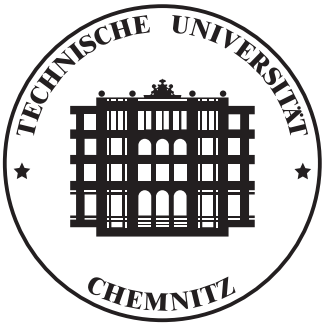
Technische Universität Chemnitz
Fakultät für Informatik



Ein durchgängiges Architekturkonzept für Anwendungs- und Betriebssysteme

Die Grundlage für Entwurf und Implementierung des *CheOpS*-Kerns

Dissertation

zur Erlangung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

vorgelegt

der Fakultät für Informatik

der Technischen Universität Chemnitz

von: Dipl. Inf. Sven Graupner

geboren am: 23. Januar 1967 in Chemnitz

Gutachter:

Prof. Dr. Ing. habil. Winfried Kalfa, Technische Universität Chemnitz

Prof. Dr. sc. nat. Christoph Polze, Humboldt-Universität zu Berlin

Prof. Dr. rer. nat. Hermann Härtig, Technische Universität Dresden

Chemnitz, den 3. November 1997

Kurzfassung

Die zunehmende Einsatzvielfalt von Hard- und Softwaresystemen führt zu einem wachsenden Bedarf, Betriebssysteme als (Software-) Infrastrukturen für Anwendungen zweckorientiert anzupassen bzw. herzustellen. Das Anwendungsgebiet reicht dabei über das der universellen Betriebssysteme für universelle Rechensysteme hinaus. Anwendungsentwicklung findet nicht mehr notwendigerweise ausschließlich im Anwendungsbereich statt, sondern kann sich auch auf unterliegende Systemschichten beziehen. Infrastruktur muß dafür einerseits offen sein, und andererseits muß es auch dort geeignete Strukturen geben, welche die Herstellung bzw. Anpassung von Infrastrukturkomponenten für Anwendungshersteller auch geeignet unterstützen.
Dem Gesamtsystem sollte eine Architektur zugrunde liegen, welche strukturell, funktional und methodisch alle Systembereiche bzw. verschiedenartige Zielsysteme in einer einheitlichen, durchgängigen Weise erfaßt und heute vorhandene Strukturbrüche überwindet. Gleichzeitig muß der Spezifik unterschiedlicher Systemumgebungen Rechnung getragen werden.

In dieser Dissertation wird eine dafür geeignete Architektur aus der Analyse des Gegenstandsbereichs dynamischer ablaufender Systeme hergeleitet und begründet. Die praktische Umsetzbarkeit wird anhand der Implementierung des *CHEOPS*-Kerns gezeigt und bewertet.

Um der Verschiedenartigkeit von Systemen bzw. Systembereichen gerecht zu werden, ist ein hohes Maß an Skalierbarkeit der Architekturmerkmale erforderlich. Dies wird durch Trennung universeller Merkmale von konkreten Ausprägungen in jeweiligen Zielumgebungen erreicht. In der *generalisierten Architektur* wird das unterlegte, durchgängig anwendbare Architekturkonzept festgelegt. Es ist durch *Schichten* als vertikale Grundstruktur gekennzeichnet. Innerhalb von Schichten bilden *(Verarbeitungs-) Instanzen* das zentrale Strukturelement, um semantisch zusammengehörige Teilverarbeitungen identifizierbaren, aktiv dienstausführenden Elementen zuzuordnen. Das Vorbild dafür ist das aus dem Anwendungsbereich bekannte *Client-Server-Modell*. Die Anpassung an jeweilige Systemumgebungen erfolgt dann durch explizites Ableiten *spezieller Ausprägungen* von Elementen und Beziehungen aus den generellen Architekturmerkmalen und der Zuordnung jeweils geeigneter *Ausführungs-* und *Ablaufeigenschaften*.
In dieser hier explizit vorgenommenen Differenzierung liegt ein wesentlicher Unterschied zu anderen Architekturvorschlägen. Erst dadurch wird das Grundmuster des *Client-Server-Modells* auf alle Infrastrukturschichten in speziell angepaßten Ausprägungen übertragbar und damit Anwendungsanpaßbarkeit strukturell auch für Infrastruktur unterstützt.

Mit der Implementierung des *CHEOPS*-Kerns konnte gezeigt werden, daß sich das Strukturmuster von Instanzen selbst für die unterste Systemschicht der Unterbrechungsverarbeitung anwenden läßt und sich daraus neben strukturellen auch vorteilhafte Ablaufeigenschaften ergeben. Dieses neuartige Implementierungsprinzip auf Basis sogenannter *iproc-Instanzen* wird im zweiten Teil der Dissertation im Detail vorgestellt und bewertet.

Bibliographische Angaben: Dissertation, 182 Seiten, 60 Abbildungen, 296 Literaturverweise,
Technische Universität Chemnitz, Fakultät für Informatik, Professur Betriebssysteme,
eingereicht am 25. Februar 1997, Tag der öffentlichen Verteidigung: 3. November 1997.

Danksagung

An erster Stelle möchte ich mich bei Herrn Prof. Dr. Winfried Kalfa bedanken, der diese Dissertation gefördert und den dafür notwendigen Freiraum gewährt hat. Durch das Forschungsprojekt *CHEOPS* wurde an der Professur Betriebssysteme der fachliche Rahmen für diese Arbeit geschaffen.

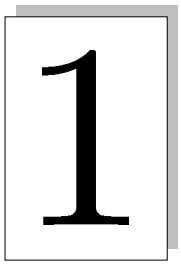
Auch den beiden externen Gutachtern, Herrn Prof. Dr. Christoph Polze von der Humboldt-Universität zu Berlin und Herrn Prof. Dr. Hermann Härtig von der Technischen Universität Dresden soll an dieser Stelle für die Bewertung der Dissertation gedankt sein.

Den Kollegen an der Professur Betriebssysteme und den beteiligten Studenten sei ebenfalls gedankt. Dank gebührt auch Herrn Gerd Liefländer für die Durchsicht und Diskussion des ersten Teils des Manuskripts und Holger Trapp für orthographische Hinweise.

Bedanken möchte ich mich auch bei früheren Betreuern und Kollegen. Sie haben maßgeblichen Anteil an meinem Werdegang. Genannt seien Dr. Eckhard Einert, Dr. Klaus Müller und Prof. Dr. Klaus Mätzel.

Insgesamt möchte ich mich bei allen bedanken, die im wissenschaftlichen und privaten Umfeld zum Gelingen dieser Dissertation beigetragen haben.

Inhaltsverzeichnis



Kapitel 1

Einführung – Überblick

Anwendungsorientierte Anpaßbarkeit ist gegenwärtig ein dominierender Forschungsgegenstand auf dem Gebiet der Betriebssysteme. Das belegt eine neue Generation von Forschungsprojekten, die vor allem in den USA vorangebracht werden[1]: *SPIN* an der University of Washington [Bersh95], *Exokernel* am M.I.T. [Kaash95], *Bridge* an der Carnegie Mellon University [Lucco93], *V++/Cache Kernel* an der Stanford University [Cheriton94], *Scout* an der University of Arizona [Peterson94] oder *VINO* an der Harvard University [Seltzer94].

Die an diesen Projekten beteiligten Wissenschaftler und Institutionen waren in der Vergangenheit federführend bei der Entwicklung neuer Technologien, wie die der μ-Kerne oder verteilter Betriebssysteme. Die neuen Forschungsprojekte sind Ausdruck einer gewandelten Welt in der zweiten Hälfte der neunziger Jahre, deren Grundtendenz unter das Schlagwort *Diversifizierung von Hard- und Software* gefaßt werden kann. Der allgegenwärtige Trend nach mehr Kunden- und Anwendungsorientierung wird auch für Betriebssysteme immer bestimmender. Das Ziel sind wirtschaftlich herstellbare, an verschiedenartige Zielumgebungen und Einsatzzwecke anpaßbare Betriebssysteme ($\rightarrow$ *customizable operating systems*) als zugeschnittene Softwareinfrastrukturen für Anwendungssysteme.
Auch in Deutschland gab es und gibt es Aktivitäten mit diesem Ziel: PANDA [ABB+93a] und GENESYS [NS95] an der Universität Kaiserslautern, PEACE [Schrö94] an der GMD-FIRST Berlin oder *PM* [Klei94] an der Universität Erlangen-Nürnberg. Auch im Bereich kleiner Echtzeitkerne gibt es Arbeiten, wie *L4* an der GMD Birlinghoven [Lie95] oder *PXROS* der Firma *HighTec* aus Saarbrücken [Mau96]. Beispiele für auf Parallelrechner zugeschnittene Betriebs- und Laufzeitsysteme sind *COSY* [BBH96] und *MARS* (PC^2-Lab) [GR96] von der Universität Paderborn.

Diese Dissertation ist Teil des *CHEOPS*-Projekts an der Professur Betriebssysteme der Technischen Universität Chemnitz im Rahmen des von der DFG unterstützten Forschungsprojekts *"Untersuchung rekursiver und reflexiver Methoden zur Anpassung in objektorientierten verteilten Betriebssystemen an unterschiedliche Hardware und Software"* [Kal93] .

1.1 Trends für Betriebssysteme

Vergangenheit und Gegenwart sind von der Dominanz universeller Betriebssysteme für universelle Rechner geprägt. Lediglich Echtzeitbetriebssysteme konnten sich als spezielle Gattung in einem Teilbereich etablieren. In der breiten kommerziellen Anwendung findet man die seit Jahren bekannten Betriebssysteme. In der Forschung konzentrierte man sich auf universelle

[1] vgl. Panel-Diskussion auf der OSDI'94 *"Radical Operating System Structures for Extensibility"* [BCK+94] bzw. eine Zusammenfassung in [Grau95d]

μ-Kerne und universell anwendbare verteilte (Betriebs-) Systeme bzw. Programmiersysteme für verteilte Anwendungen [Bor92]. Gegenwärtig verstärken sich jedoch Trends, die von Universalität wegführen. Betriebssystemtechnologie wird in Zukunft ein breiteres Anwendungsspektrum erfahren und mehr Menschen tangieren, als das heute der Fall ist.

1. Der Trend nach Spezialisierung in einem vernetzten Umfeld

Vernetzung und die gemeinsame Nutzung von Ressourcen führen zu Aufgabenteilung und Spezialisierung von Maschinen als File-, Compute- oder Datenbank-Server, als Terminals oder E/A-Geräte. Spezialisierung schlägt sich heute in Ausstattung und Konfiguration von Maschinen nieder, nicht aber in spezialisierten Betriebssystemvarianten.
Telematik und Multimedia erfordern eine spezielle Ressourcenverwaltung [Wolf95, SteiN95, Herr91], andererseits können und müssen nicht alle Maschinen für diese Zwecke ausgelegt sein. In der Spezialisierung, vor allem auch der Betriebssysteme, liegt ein großes Potential für neue Anwendungen und für Leistungszuwachs, vielleicht mehr noch als in immer schnellerer Hardware.
Bei *Plan9* gibt es bereits für drei Klassen von Aufgaben drei Varianten von Kernen für drei spezialisierte Maschinentypen: CPU-Server, File-Server und Terminals [PPTT91][2].

2. Der Trend nach Mobilität

Für immer mobilere Menschen wird der entfernte Zugriff auf ihre Daten über verschiedene Wege immer wichtiger. Mobile Geräte sollen handlich sein, man akzeptiert dabei Beschränkungen von Leistung und Ressourcen. Die Chiptechnologie erlaubt heute, eine breite Palette solcher Geräte herzustellen: Personal Diaries, Personal Digital Assistants, Palmtops, auch Handies, intelligente Chipkarten u.a. All diese Geräte sind softwaregesteuert. Für die Ressourcenverwaltung spielen bislang untypische Kriterien eine Rolle, wie Stromaufnahme ($\rightarrow$ energiesparendes CPU-Scheduling [WWDS94]) oder das Gewicht und der Platzbedarf von Speicher [DKM+94].
Heute sind für diese Klasse von Systemen Betriebssystemaufgaben mit in der Anwendungssoftware enthalten. Mit zunehmenden Ressourcen, wachsender Einsatzvielfalt und steigendem Umfang der Software wird es aber sinnvoll und notwendig werden, Ablaufsteuerung und Ressourcenverwaltung aus Anwendungen herauszulösen und in kleinen, dedizierten Laufzeit- oder Betriebssystemen zu konzentrieren. Es werden letztlich dieselben Gründe sein, die schon in den sechziger Jahren zur Entstehung der Betriebssysteme für Universalrechner geführt haben. Die Verschiedenartigkeit wird hier aber keine universellen Systeme zulassen.
Betriebssystemtechnologie wird deshalb in diesem Bereich auch eine wichtige softwaretechnologische Bedeutung erlangen, da Anwendungen entlastet und einfacher herstellbar werden. Existierende Beispiele sind *Apple's Newton OS* [WSW+94] oder die Betriebssysteme für *HP's Omnibook* und den *PSION*. *Sun* plant für *Java* eigene Prozessoren mit Systemsoftware in drei Skalierungen: *pico-*, *micro-* und *UltraJava* [Sun96b]. Mittlerweile wurde für *Java* auch ein stand-alone *JavaOS* für kleine mobile Geräte für das *Internet* vorgestellt [Sun96a].

3. Der Trend nach Steuerung durch Software – *Embedded Systems*

Es ist in vielen Fällen wirtschaftlicher, Steuerungsaufgaben durch Software auf Basis billiger Standard-Hardware zu erbringen, als fest verschaltete Steuerungen aufzubauen. Bei sich angleichenden technischen Parametern findet Wettbewerb in vielen Bereichen zunehmend über die "Intelligenz" der Produkte statt, was komplexere, programmierbare Steuerungen zur Folge hat. Man findet diesen Trend in der Konsumelektronik, in der Medizintechnik, in Steuerungen für Fahrzeuge, Maschinen und Anlagen und in vielen anderen Bereichen. Die Herstellung von Software wird zu einem wichtigen Kostenfaktor, und es wird ebenso relevant werden, anwendungsbezogene Aufgaben eines Softwaresystems von der eigenen Ablaufsteuerung und Ressourcenverwaltung zu entkoppeln und eine Softwareinfrastruktur für Anwendungen zu schaffen.

[2] vgl. auch Untersuchung und Bewertung von *Plan9* in [Suhr95b, Grau95c]

4. Der Trend nach Prozeßorientierung bei Anwendungen

Nicht nur für Betriebssysteme ist die Entkopplung separater Abläufe in Prozesse wichtig, auch die traditionelle Anwendungsentwicklung wird zunehmend davon erfaßt.
In parallelen oder verteilten Systemen sind Prozesse die Grundeinheiten des Systems, nicht Prozeduren, auch wenn man die Illusion dafür herstellen kann. Der Wechsel des Paradigmas von sequentiellen Prozeduren zu parallelen Prozessen hat in den breit angewandten Methoden der Softwaretechnologie noch keinen entsprechenden Niederschlag gefunden. Es bedarf insofern auch hier einer Neuorientierung, die für Anwendungsentwicklung mehr Betriebssystemtechnologie zugänglich macht, wie Fragen der Steuerung und Interaktion von Prozessen. Es genügt nicht mehr, Systemstruktur allein auf Code und Daten zu beziehen (Module, Instanzierungen von Typen oder Klassen). Die Ablaufstruktur für Prozesse gehört für verteilte, parallele Anwendungen ebenfalls dazu und verlangt eine gleichwertige Betrachtung.

Unter dem Schlagwort *Diversifizierung* kann man drei Entwicklungen zusammenfassen:

- Der Bereich vernetzter Universalsysteme ist von Spezialisierung geprägt, die sich auch im Betriebssystem niederschlägt und dort Freiheitsgrade für Anpassung erfordert.
- Es entsteht eine neue Klasse dedizierter Betriebssysteme mit stark verschiedenen Eigenschaften und für sehr spezielle Einsatzzwecke.
- Anwendungsentwickler werden verstärkt mit Betriebssystemtechnologie konfrontiert:
  - für die Anpassung von Betriebssystemen, die immer mehr zum Anwendungsentwickler bzw. Endnutzer des Systems verlagert wird,
  - für die Herstellung oder Anpassung von Infrastruktur für eigene Anwendungen und
  - für verteilte und parallele Anwendungen, deren Grundeinheiten Prozesse sind.

1.2 μ-Kerne – Ein neuer Aufbruch

Es scheint, daß gegenwärtig in der Forschung zu Betriebssystemen, insbesondere zu Betriebssystemkernen, ein neuer Aufbruch erfolgt. Nach 15 Jahren Entwicklung zu μ-Kernen muß konstatiert werden, daß μ-Kerne die traditionellen monolithischen Betriebssystemstrukturen in der breiten Anwendung nicht ablösen konnten. Die Gründe dafür sind heute bekannt (vgl. Abschnitt 2.5, [Ous90, Joy91]). Mit μ-Kernen gelang es zwar, Kerne funktional abzurüsten, ein signifikanter Effizienzgewinn konnte jedoch nicht erreicht werden, weil das von *Unix* geprägte, teure Ausführungsmodell übernommen wurde: preemptive Prozesse mit getrennten Adreßräumen und virtueller Speicher. Im Gegenteil, μ-Kerne erreichen aufgrund der höheren internen Kommunikationsaktivität im Kontext eines Endsystems nicht einmal die Leistungswerte vergleichbarer monolithischer Systeme. Werte von Messungen sind zahlreich veröffentlicht [BALL90, BDDR91, vEick92]. Die potentielle Wählbarkeit von Strategien zur anwendungsorientierten Optimierung des Gesamtsystems, wie beispielsweise für die aus dem Kern ausgelagerte Paging-Steuerung, konnte die Leistungsbilanz nicht wesentlich verbessern und wurde auch nicht wirklich praktiziert. Diese ernüchternde Einschätzung gab unter anderem auch den Anlaß, das *Mach*-Projekt an der *CMU* 1994 einzustellen. Generell klingen die Aktivitäten zu den klassischen, an *Mach* orientierten μ-Kernen ab.
Eine Zunahme verzeichnen dafür neuartige Ansätze in der Forschung, die man gegenwärtig in den USA an einer neuen Generation von Experimentalsystemen erkennen kann. Einige dieser Systeme brechen radikal mit gewohnten Eigenschaften, indem getrennte Adreßräume nicht mehr notwendig zu den Ablaufeigenschaften von Anwendungsprozessen gehören müssen. Auch die Illusion "unbegrenzten" virtuellen Speichers wird auf den Prüfstand gestellt, verbunden mit der Argumentation, daß ein signifikanter Effizienzgewinn nur durch Abrüstung der teuren

Ausführungsmodelle erreicht werden kann. In der ***Skalierbarkeit der Ausführungsmodelle*** wird eine wichtige Quelle für Leistungszuwachs und daher eine Schlüsseleigenschaft künftiger Betriebssysteme gesehen, auch mit der Option, Effizienz zu Lasten von "Komfort" zu erzielen:

- "unbegrenzte" virtuelle Ressourcen $\leftrightarrow$ reale Ressourcen,
- ad hoc Anforderung und Belegung von Ressourcen $\leftrightarrow$ vorausgeplante, intelligente Strategien,
- idealisierte Hardware $\leftrightarrow$ Zugang zu realer Hardware,
- Schutz, Isolierbarkeit, Redundanz für Ausfallsicherheit $\leftrightarrow$ Verzicht auf diese Eigenschaften usw.

Die potentielle ***Wahlmöglichkeit von Ausführungsmodellen*** bedarf der Konfigurierung der Infrastruktur durch Anwendungshersteller auf einem elementaren Niveau und stellt eine völlig neue Herangehensweise für Betriebssysteme dar, die damit auch ihren universellen Charakter verlieren. Lediglich Anpaßbarkeit bleibt als universelle Eigenschaft. Als ***Ausführungsmodell*** sollen hier die in einem Gegenstandsbereich vorherrschenden Ablauf- und Ausführungseigenschaften verstanden werden, die sich aus der Art von Elementen und der Wirkungsweise von Verarbeitungsprozessen ergeben. Dies sind reale Eigenschaften. Der Begriff des Ausführungs*modells* hat sich jedoch in der englischsprachigen Literatur dafür etabliert ($\rightarrow$ *execution model*).

Nun mögen die genannten Beweggründe und Vorstellungen zwar wünschenswert sein, eine praktikablen Realisierung scheint aber in weiter Ferne. Es ist in der Tat auch nicht so, daß alle Forschungsprojekte derart extreme Ziele verfolgen wie beispielsweise *Exokernel* (vgl. Abschnitt 2.7). Es mag auch sein, daß sich manche Entwicklung als Irrweg erweist und in der Praxis ganz andere Erfordernisse bestehen. Es wird aber ein neuer Aufbruch in der Forschung zu Betriebssystemen deutlich, der von neuen und unkonventionellen Ideen vorangetrieben wird.

Das globale Ziel ist dagegen klar. Es geht um mehr Anwendungsorientierung in Betriebssystemen. Eine Folge davon wird sein, daß mehr Entwicklungskompetenz für das Betriebssystem zu Anwendungsherstellern und Endnutzern verlagert wird, damit diese die gewünschten Modifikationen auch tatsächlich ausführen können. Das wird sich auf die Konfigurierung des Betriebssystems beziehen, wie es in Grenzen schon immer der Fall war, aber zunehmend auch auf die Modifikation und Herstellung innerster Kern-Komponenten, um neue oder veränderte Infrastruktureigenschaften (Dienste und Ausführungsmodelle) zu bewirken.
Das setzt natürlich eine plausible Ordnung in der Infrastruktur, d.h. im Betriebssystem, voraus, die heute kaum gegeben ist. Diese wichtige Frage wird von den genannten Forschungsprojekten jedoch kaum beachtet, es steht die Kern-Integrität bei Erweiterungen im Vordergrund.

Das Ziel dieser Arbeit besteht darin, eine Architektur für ein Gesamtsystem mit dem Hintergrund anwendungsorientierter Anpaßbarkeit zu finden. Es geht dabei um die Architektur für ein ablaufendes System, nicht primär um die Frage der Struktur der Software zu seiner Herstellung. Das ist ein wesentlicher Unterschied zu Ansätzen, die unter das Stichwort *objektorientierte Software für Betriebssysteme* gefaßt werden können. Die hier vorgenommene Abgrenzung zwischen Software- und Ablaufstruktur wird im folgenden kurz und in Kapitel 2 ausführlich diskutiert.

1.3 Softwareeigenschaften versus Ablaufeigenschaften

Die Infrastruktur bestimmt die Ablaufeigenschaften für Anwendungen ($\rightarrow$ *Dienste, Ausführungsmodell*), unabhängig davon, nach welcher softwaretechnologischen Methode Anwendungen hergestellt wurden: prozedur-, modul- oder objektorientiert. Auf einer Infrastruktur *Unix* gibt es ausschließlich *Unix*-Prozesse. Es ist folglich eine grundlegende Differenzierung:

$$\textbf{\textit{Softwareeigenschaften}} \quad \text{vs.} \quad \textbf{\textit{Ablaufeigenschaften}},$$
$$\textit{Softwarestruktur} \quad \text{vs.} \quad \textit{Ablaufstruktur},$$

die man auf den Anwendungsbereich, aber auch auf die Infrastruktur oder das Betriebssystem

beziehen kann. Es ergeben sich ***vier Gegenstandsbereiche***, die klar zu unterscheiden sind:

Herstellung der Software, ablaufendes System	×	*Anwendungsbereich, Infrastruktur*

- Geht es um Methoden zur Herstellung oder Anpassung von Anwendungssoftware?
- Geht es um Ablaufeigenschaften für Anwendungen, die von der Infrastruktur vorgegeben sind?
- Geht es um die Herstellung oder Anpassung von Software für die Infrastruktur?
- Geht es um die Ablaufeigenschaften innerhalb der Infrastruktur?

Für jeden dieser vier Gegenstandsbereiche kann man Aussagen treffen,

- worin das *Ziel* für einen Anpassungsprozeß besteht,
- was der *Gegenstand* einer Anpassung ist,
- welche *Mittel und Methoden* dafür vorhanden sind und
- *wer* eine Anpassung initiieren und *wer* diese ausführen kann.

Anpassung setzt dabei die ***Offenheit*** des Gegenstandsbereichs voraus. Das bedeutet, den Zugang und die Mittel zur Manipulation für den Ausführenden bereitzustellen, aber auch eine Gliederung in abgrenzbare, identifizierbare Einheiten, d.h. ***Modularisierung***. Modularisierung und Beziehungen schaffen ***Struktur*** im Gegenstandsbereich. Offenheit ist eine notwendige Bedingung für Anpassung, Modularisierung und Struktur sind sinnvolle Bedingungen, um Anpassung für den Ausführenden auch praktikabel zu gestalten. Offenheit, Modularisierung und Struktur sind für die vier genannten Gegenstandsbereiche natürlich verschieden ausgeprägt. Offenheit von Software kann man als Verfügbarkeit der Quellen interpretieren, Offenheit eines ablaufenden Systems als Möglichkeit, neue Elemente während des Ablaufs hinzuzufügen oder zu ändern.

Die Eigenschaften verschiedener Methoden zur Softwareherstellung sind bekannt und sollen hier nicht diskutiert werden [Mey88, Rumb91, Pomb93, Mü92a]. Sie führen zu einer bestimmten *Softwarestruktur*. Die Ablaufeigenschaften werden von der ***Infrastruktur***[3] bestimmt und bewirken eine *Ablaufstruktur*, die nicht mit der Softwarestruktur gleichgesetzt werden kann. Für beide Bereiche läßt sich eine weitere Gliederung in Funktionseinheiten (→ ***Instanzen***[3]) angeben. Die Art ihrer Ausprägung und die Möglichkeiten für ihre Beziehungen und ihr Zusammenwirken (→ ***Interaktion***[3]) werden vom Ausführungsmodell vorgegeben, welches einer Programmiersprache zugrunde liegt oder welches für ein ablaufendes System von der Infrastruktur bestimmt wird. In dieser Arbeit werden die genannten Begriffe[3] auf das ***ablaufende System*** bezogen.

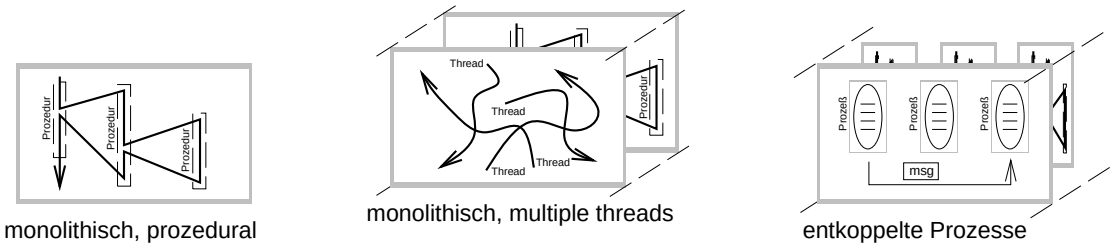


Abb.1.1 Typische Ablaufstrukturen in heutigen Systemen

Ein System wird als ***monolithisch*** betrachtet, wenn die spätere Ablaufstruktur bereits bei der Herstellung fixiert wird und während des Ablaufs nicht mehr änderbar ist, z.B. durch feste Bindung von Adressen. Monolithische Systeme entstehen typischerweise aus Sprachen mit Prozeduren als Ablaufform, wie *C*, *Modula-2*, aber auch *C++*[4]. Diese Struktur ist effizient von

[3] Die Terminologie wurde zutreffenderweise aus dem Lehrbuch *"Systemarchitektur"* [Wett93] übernommen.

[4] Unabhängig von Objektorientierung sind Methodenaufrufe von Objekten Prozeduraufrufe. Im Fall von Aufrufen virtueller Methoden über Basisklassenzeiger auf Objekte sind es indirekte Prozeduraufrufe.

heutigen Prozessoren umsetzbar. Sie bedingt einen globalen Adreßraum für Code und Daten und damit einen globalen Gesamtzustand eines monolithischen Systems. Die genannten Programmiersprachen entsprechen dem sequentiellen von-Neumann–Modell. Für Betriebssysteme und zunehmend auch für Anwendungssysteme ist dieses wegen real vorhandener paralleler Prozesse, aber auch schon bei der Interaktion mit "äußeren" Prozessen inadäquat (nichtblockierende E/A-Operationen, Unterbrechungen, ereignisgesteuerte Anwendungen u.a.).
Um auch für solche Ablaufeigenschaften die gewohnten Sprachen zu verwenden, ist ein oft beschrittener Weg, parallele Prozesse in Form von ***Threads*** [CD88] in einem monolithischen System zu initiieren. Diese Technik ist einfach umsetzbar und stellt in gewisser Weise ein Zwischenstück zu den entkoppelten Prozessen dar. In [Grau95b] wird gezeigt, wie man für die Sprache *C* mit einfachen Mitteln drei nichtprozedurale Ablaufformen erzeugen kann: Coroutinen, entkoppelte Coroutinen[5] und preemptive Threads. Das Gesamtsystem bleibt aber monolithisch.

Entkoppelte Prozesse besitzen dagegen a priori keine gemeinsamen Zustände und Steuerungen. Die Trennung ist typischerweise durch separate Adreßräume gegeben. Aufgrund der Nutzungsanforderungen ist diese Struktur seit langem für den Anwendungsbereich vorherrschend. Für die innere Struktur von Betriebssystemen wurde dieses Modell mit den μ-Kernen relevant. Für verteilte Systeme ohne gemeinsamen Speicher ist es sogar das einzig mögliche Ausführungsmodell. Allerdings ist dieses Modell auf Monoprozessorsystemen aufwendig umzusetzen, so daß es nur für grobgranulare Funktionseinheiten sinnvoll ist. Für Multiprozessorsysteme sinken diese Kosten aber mit einem zunehmenden Verhältnis von Prozessoren zu Prozessen.
Bei Multiprozessorsystemen gibt es eine enge Beziehung zwischen dem monolithischen Modell und dem *Shared-Memory-Modell (SMM)*, und das Modell entkoppelter Prozesse ist eng mit dem *Message-Passing-Modell (MPM)* verbunden (→ *Dualität von Prinzipien*).

1.4 Problemkreis: Spezialisierung universeller Betriebssysteme

Für universelle Betriebssysteme hat sich ein bestimmtes Anforderungs- und Nutzungsprofil herausgebildet. Einige Eigenschaften sind:

- entkoppelte, parallele Ausführung unabhängiger Anwendungen von mehreren Nutzern,
- bekanntes, breit akzeptiertes Ausführungsmodell für den Anwendungsbereich,
- Verbindlichkeit von Funktion und Schnittstelle für die Ablauffähigkeit von Anwendungen,
- Effizienz und Integrität des Kerns.

Aus diesem Profil und den Eigenschaften von Universalrechnern sind die heute typischen Strukturmerkmale entstanden:

- Zweiteilung in Anwendungs- und Kern-Bereich,
- monolithischer, nicht änderbarer Kern mit festen Regeln zur Ressourcenverwaltung,
- festgelegte Dienste des Kerns,
- festes Ausführungsmodell im Anwendungsbereich: preemptive Prozesse mit separaten Adreßräumen, "faires" Scheduling, Dateien als einzige Form dauerhafter Speicherung u.a.

Eng mit der Zweiteilung ist eine strenge Aufgabenteilung für den Betriebssystem- und den Anwendungsbereich verbunden. Die Entwicklung und Herstellung universeller Betriebssysteme ist bei wenigen Herstellern in der Welt konzentriert. Für die Anwendungsentwicklung hat sich daraus eine Herangehensweise etabliert, die in [EW95] als *bottom-up design* bezeichnet wird. Anwendungsentwicklung findet ausschließlich im Anwendungsbereich auf einer gegebenen, fixierten Infrastruktur statt. Anwendungen müssen bei ihrer Herstellung an die Erfordernisse

[5] Die Folge-Coroutine wird nicht bei TRANSFER() angegeben, sondern durch einen Scheduler bestimmt.

dieser Infrastruktur angepaßt werden. Es besteht heute keine Option für die umgekehrte Richtung, ausgehend von der Anwendungen, eine dafür zugeschnittene Infrastruktur anzupassen oder herzustellen (*top-down design*). Hierin liegt eines der Ziele der genannten amerikanischen Forschungsprojekte, die anwendungsspezifische Erweiterbarkeit des Kerns unter Wahrung seiner Integrität zu erlauben.

Der Prozeß der Diversifizierung wird dazu führen, daß mehr Beteiligte aktiv an der Entwicklung, Erweiterung und Anpassung von Betriebssystemen mitwirken werden. Betriebssystemhersteller müssen dafür die Mittel bereitstellen und geeignete Strukturen in Betriebssystemen schaffen.

Als strukturelle Probleme universeller Betriebssysteme können für den Anwendungsbereich **A** und die Infrastruktur **I** identifiziert werden:

!	- strenge *Zweiteilung:* **A** / **I**	➞ Verallgemeinerung: potentiell n Schichten,
!	- *Strukturbruch:* - entkoppelte Prozesse in **A** - monolithisch in **I**	➞ Homogenisierung: *gleiche* Grundstruktur über alle Schichten des Systems,
!	- **I** für **A** *fest*, nicht änderbar: - festgelegte Dienste - starres Ausführungsmodell	➞ *Öffnung* von **I** für Anpassung durch Dritte: - flexible Dienste, - optional anpaßbares Ausführungsmodell,

Aus den strukturellen Problemen folgen methodische Probleme:

!	- strenge *Aufgabenteilung* für die Herstellung und Anpassung von **I** und **A**	➞ Öffnung: **I** wird zunehmend *Entwicklungsgegenstand* für **A**nwendungshersteller,
!	- strenges *bottom-up* Vorgehen bei der Anwendungsentwicklung	➞ Option für *top-down* Vorgehen.

1.5 Problemkreis: Dedizierte Betriebssysteme

Es wird in Zukunft notwendig werden, für die Software sehr spezieller Zielsysteme, wie für Embedded Systems oder für mobile Geräte, die Ablaufsteuerung und Ressourcenverwaltung in einer Softwareinfrastruktur zu konzentrieren. Es entsteht eine neue Klasse kleiner dedizierter Laufzeit- oder Betriebssysteme für sehr spezielle Aufgaben und den Einsatz in bislang artfremden Umgebungen. Auch hier stößt man heute auf offene strukturelle und methodische Fragen.

Als strukturelle Probleme können für dedizierte Betriebssysteme zusammengefaßt werden:

!	- bislang *keine* Trennung von **I** und **A**	➞ *Trennung* von Funktion und Infrastruktur,
!	- *monolithisches* Gesamtsystem (ggf. multi-threaded)	➞ *Entkopplung, Kapselung* getrennter Abläufe und Verarbeitungseinheiten,
!	- Standard-Betriebssysteme wegen starrer Eigenschaften unmöglich	➞ *"skalierbare"* Eigenschaften und eine *"skalierbare"* Betriebssystemarchitektur.

Aus den strukturellen Problemen folgen auch hier methodische Probleme:

!	- Struktur wird rein *softwaretechnologisch* betrachtet (OOP: Code+Daten)	→ Einbeziehung und Kapselung von *Abläufen* (Code + Daten + Aktivität(en)),
!	- Reduktion auf *Herstellung durch Variantengenerierung* innerhalb einer Ablaufschicht des Systems,	→ bewußtes Wahrnehmen *mehrfacher Ebenen* von Infrastrukturen und Anwendungsbereichen (→ *Schichtenarchitektur*).

1.6 Fazit

Es wurden Fragen und Probleme genannt, die nicht oder nicht zufriedenstellend gelöst sind. Bei den heute vorgeschlagenen Antworten und Lösungen fällt auf, daß die Problematik oft rein softwaretechnologisch aus der Sicht der Herstellung von Software für Betriebssysteme betrachtet wird, unter Nutzung aktueller Technologien (*Objektorientierung – OOP*). Der Schwerpunkt liegt im Erreichen eines hohen Wiederverwendungsgrades von Code und einfacher Variantengenerierung für Betriebssystemsoftware. Das Gesamtsystem, seine Ablaufeigenschaften und Funktionen rücken dabei oft in den Hintergrund. Modularisierung und Struktur beziehen sich in der Regel auf Quellcode. Der Widerspruch zwischen Softwarestruktur und Ablaufstruktur und die Inadäquatheit gebräuchlicher sequentieller Programmiersprachen zur Abbildung der Realität ablaufender Systeme werden nicht bewußt wahrgenommen. Man geht den anderen Weg, real vorhandene, aber nicht in das Ausführungsmodell einer Sprache passende Eigenschaften "vor dem Programmierer zu verbergen". Dafür gibt es verschiedene Techniken: (Klassen-) Bibliotheken mit Seiteneffekten oder Quellcode-Manipulatoren (Präprozessoren), mit denen parallele Prozesse, der Übergang in andere Schutzdomänen oder entfernte Methodenaufrufe umgesetzt werden. Es gibt zahlreiche Beispiele dafür, heute vor allem für *C++*.

Ein zweiter Gesichtspunkt ist, daß die hierarchische Struktur eines Gesamtsystems durch aufeinander aufsetzende Softwareschichten im Sinne *abstrakter Maschinen* mit jeweils verschiedenen Ausführungseigenschaften nicht erfaßt wird. Es erscheint im Gegenteil erstrebenswert, dem Programmierer eine "einheitliche (Objekt-) Welt" darzustellen, d.h. wiederum verschiedenartige Ausführungseigenschaften zu verbergen: *"Closing the Gap Between Different Object Models"* [Son92] oder *"Unification of Active and Passive Objects in an Object-Oriented Operating System"* [MHM+95]. Auch hier werden die genannten Techniken eingesetzt, um die Realität im ablaufenden System als gedankliches Modell anders erscheinen zu lassen, als sie tatsächlich ist.

Aus Sicht der Benutzung sind das durchaus geeignete Methoden für Abstraktion. Der oft gebrauchte Begriff *Transparenz* bedeutet hier Idealisierung durch Verbergen innerer Kompliziertheit [PS75]. Soll aber Anpassung im Inneren erfolgen, ist die Offenlegung und das Verständnis des realen Geschehens notwendig. ***Transparenz*** in diesem Sinne heißt *Offenheit und Durchschaubarkeit* innerer Strukturen und Beziehungen, damit Anpassung praktikabel ausführbar ist. Das Ziel sollte hier darin bestehen, nicht innere Kompliziertheit durch weitere Komplexität zu verbergen, sondern auch im Inneren eines Systems klare, verständliche *reale* Strukturen und Abläufe herzustellen. Das gilt insbesondere für Betriebssysteme, wenn diese zum Gegenstand anwendungsorientierter Anpassungen werden sollen, die nicht mehr vom Betriebssystemhersteller, sondern von Dritten ausgeführt werden. Es geht daher nicht allein um die Struktur von Quellcode, sondern dem Gesamtsystem muß eine geeignete Architektur zugrunde liegen, die auch unterschiedliche Ausführungseigenschaften mit erfaßt, die es real in verschiedenen Schichten eines Systems gibt. Das Leitmotiv ist Transparenz im Sinne durchschaubarer innerer Strukturen für *alle* Schichten des Systems, was letztlich erst die Grundlage für verständliche Software schafft.

Der rein softwaretechnologisch geprägte Ansatz ist aus folgenden Gründen unzureichend:

! - *Inadäquatheit* sprachlicher Ausdrucksmittel für die Beschreibung der Realität im ablaufenden System (→ *Softwarestruktur* ≠ *Ablaufstruktur*).

! - *Modularisierung* (Kapselung) bezieht sich auf Code und Daten, wie es in vielen Programmiersprachen vorgegeben ist, parallele Abläufe (Prozesse) werden nicht erfaßt.

! - Die *Schichtung* in realen Systemen und die Realität mehrerer, aufeinander aufsetzender Softwareinfrastrukturen mit jeweils verschiedenartigen Ausführungseigenschaften für übergeordnete Instanzen ist auf *Sprachebene nicht beschreibbar.*

! - Die *Idealisierung* des realen Ablaufs und des Erscheinens von Instanzen unter verschiedenen Ausführungseigenschaften ist für Anpassungen nicht immer vorteilhaft.

In der klassischen Betriebssystemtechnologie geht es dagegen um das reale Geschehen in ablaufenden Systemen. Softwaretechnologische Fragen stehen nicht im Vordergrund. Ablaufsteuerung und Ressourcenverwaltung werden auf Forderung, Zuteilung und Nutzung von Betriebsmitteln für Prozesse innerhalb einer Schicht und ***Dienste*** zwischen Schichten zurückgeführt. In Schichten werden durch Prozesse neue virtuelle oder logische Betriebsmittel, Dienste und wiederum Prozesse für übergeordnete Schichten transformiert und erzeugt [Kal90]. Auf diese Weise werden in der Infrastruktur auch die ***Ausführungseigenschaften*** für übergeordnete Schichten hergestellt. Diese Sicht von Betriebssystemen sagt jedoch nichts über die Art der Herstellung aus, sondern betrachtet allein das ablaufende System. Insofern gibt es auch hier Unzulänglichkeiten:

! - Betriebssystemstruktur besteht aus Schichten von Prozessen und Betriebsmitteln. Es gibt a priori *keine weitere Strukturierung* innerhalb einer Schicht (Kapselung).

! - Fragen der *Herstellung* von Software für Betriebssysteme werden *nicht betrachtet.*

1.7 Abgrenzung und Ziel dieser Arbeit

Es zeigt sich, daß weder die reine Softwaretechnologie noch die klassische Lehre von Betriebssystemen die alleinige Antwort auf heutige Fragen liefern. Es kommt vielmehr auf eine geeignete ***Verknüpfung aktueller Techniken*** aus beiden Bereichen und die Aufnahme von Erkenntnissen aus der Vergangenheit an.

In dieser Dissertation wird für das *CHEOPS*-Projekt die Ablaufstruktur in den Vordergrund gestellt, die Frage der adäquaten Widerspiegelung programmiersprachlicher Elemente objektorientierter Sprachen im ablaufenden System wird in einer zweiten Dissertation behandelt [Schu96].

Hier ist das Ziel, eine weithin *skalierbare Architektur* für dynamisch anpaßbare Betriebssysteme abzuleiten, die ein *homogenes Grundmuster über alle Schichten* aufweist. Es soll gezeigt werden, daß sich ein Betriebssystem auf dieser Basis praktisch aufbauen läßt.

Die wesentlichen Ziele der vorgeschlagenen Architektur sind zusammengefaßt:

- ***homogene Grundstruktur*** eines ablaufenden Systems über ***alle Schichten***, um Anpassung zu unterstützen und den Strukturbruch zu überwinden;
- ***Skalierbarkeit*** der Grundstruktur für verschiedene Schichten und Systemumgebungen;
- ***Modularisierung*** innerhalb einer Schicht auf Basis abgrenzbarer, selbständig agierender Funktionseinheiten; Entkopplung von Abläufen: → ***Instanzen***;
- explizites Herausstellen von ***Schichten*** mit verschiedenen ***Ausführungseigenschaften***;
- potentielle ***Offenheit*** aller Schichten für ***dynamische Anpassung***, was die Möglichkeit zur dynamischen Änderung von Infrastruktureigenschaften bedeutet;
- explizite ***Abbildung*** programmiersprachlicher Elemente in Ausführungseinheiten des ablaufenden Systems zur Unterstützung der Herstellung.

1.8 Grundüberlegung, prinzipielles Vorgehen

Die Grundüberlegung zum Erreichen der genannten Ziele orientiert sich an den Fragen:

Was ist ***universell*** (gleichbleibend, übertragbar) ?

Was ist ***speziell*** (für einen konkreten Zweck zugeschnitten, nicht übertragbar) ?

Was ist ***spezialisierbar*** (für den Zweck übernehm- und anpaßbar) ?

Dabei kommt es auf den Bezugsbereich für diese Fragen an. Dieser wird hier weiter gefaßt, als es üblicherweise für die Herstellung von Software erfolgt. Hier sollen auch strukturelle, inhaltliche und methodische Aspekte einfließen, die Architektur also umfassend betrachtet werden.

Anpaßbarkeit ist in einem übergreifenden Sinn die Eigenschaft eines Systems, eines Konzepts, einer Methode oder eben einer Architektur, in eine fremde Umgebung übertragbar und dort anwendbar zu sein. Das heißt auch, nicht paßfähige Elemente oder Eigenschaften anzupassen, zu ersetzen oder neu zu schaffen. Eine Vorgehensweise oder Methodik ist dabei ebenso übertragbar und anpaßbar wie eine Grundstruktur oder eine Architektur. Generelle Merkmale können unverändert übernommen werden, andere müssen neu hinzugefügt werden und weitere sind für das Zielsystem zu konkretisieren, d.h. zu spezialisieren.

Für wiederzuverwendende Elemente eines Systems müssen drei Randbedingungen betrachtet werden, und es ist abzuwägen, ob erforderliche Anpassungen sinnvoll und mit vertretbarem Aufwand durchführbar sind. Zu betrachten sind:

- ▷ *funktionale Anforderungen (**f()**)*, die eine Komponente für einen Zweck erfüllen muß,
- ▷ *Beziehungen zur Umgebung (**u**)*, d.h., welche anderen Komponenten müssen in der Umgebung auf gleicher Ebene zum Funktionieren der Komponente vorhanden sein, und
- ▷ *infrastrukturelle Voraussetzungen (**i**)* für die Existenz und Funktion der Komponente.

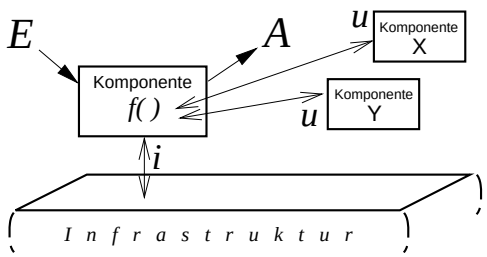


Abb.1.2 Umgebung und Infrastruktur

Anpassung ist ein Mittel, Eigenschaften der drei Kategorien ***f()***, ***u*** und ***i*** in Einklang zu bringen. Eine Komponente kann funktional an eine gewandelte Umgebung oder an eine veränderte Infrastruktur (→ *bottom-up*) angepaßt werden. Aber auch der umgekehrte Weg sollte prinzipiell gangbar sein, eine Umgebung oder eine Infrastruktur an die Erfordernisse einer Komponente anzupassen (→ *top-down*).

Rekursion kommt ins Spiel, wenn Infrastruktur selbst zum Anpassungsgegenstand wird. Idealerweise sollte innerhalb der Infrastruktur eine möglichst *ähnliche Struktur* bestehen, was in den o.g. Zielen für die hier angestrebte homogene Grundstruktur zum Ausdruck kommt. Natürlich kann es nicht exakt dieselbe Struktur mit gleichen Ausführungseigenschaften innerhalb einer Infrastruktur geben, die von dieser Infrastruktur für übergeordnete Schichten erst geschaffen wird. Aber es sollte andererseits auch nicht der krasse Strukturbruch wie in heutigen Systemen vorliegen. Es muß auch nicht immer nur die zwei Schichten für Kern und Anwendung geben, sondern die Architektur sollte potentiell n Schichten mit verschiedenen Ausführungseigenschaften zulassen. Anwendungssoftware kann in eine solche Architektur ebenso eingeordnet werden wie Betriebssystemsoftware, mehr noch, dieser Unterschied wird im Prinzip aufgehoben. Der Begriff *Rekursion* beschreibt den Sachverhalt dabei treffend, weil er Endlichkeit und Abbau bis zu einem elementaren Niveau impliziert. Für Softwaresysteme erfolgt der Abbruch pragmatisch an der Hardwareinfrastruktur. Prinzipiell läßt sich die Rekursion auch innerhalb der Hardware fortsetzen, da es aus Sicht des Modells nur eine Frage der Ausprägung ist, ob eine Funktionseinheit oder Instanz als Hardware oder Software real existiert.
Neben der Rekursion von Infrastrukturen und Instanzbereichen gibt es die Aggregationsrekursion zur Auflösung in innere Teilkomponenten:

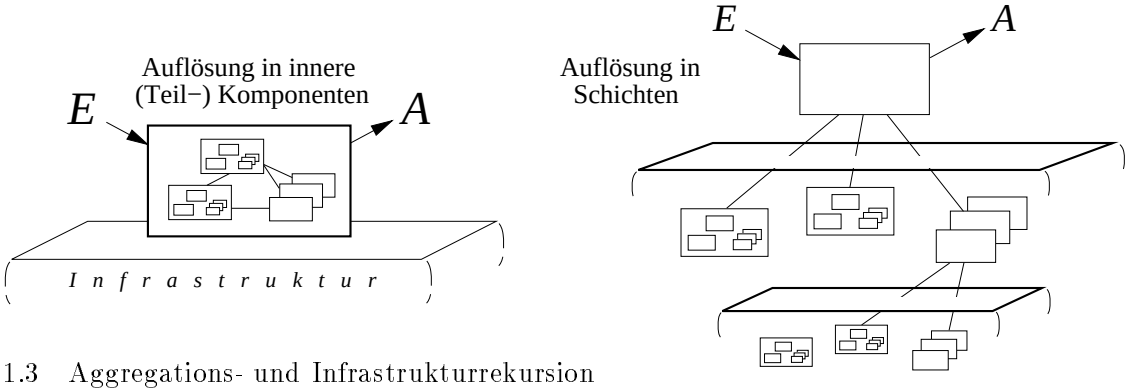


Abb.1.3 Aggregations- und Infrastrukturrekursion

Aggregations- und Infrastrukturrekursion führen zu ***Hierarchiebildung*** und sind wichtige Mittel für Struktur, Abstraktion und die Reduktion von Komplexität. Sie müssen in der Systemarchitektur berücksichtigt sein.
Für Anpaßbarkeit ist wesentlich, inwieweit die oben genannten Bedingungen Offenheit, Modularisierung und Struktur für welche Beteiligten in welchen Schichten gegeben sind.

1.8.1 Spezialisierbare Systemarchitektur

In der Architektur widerspiegeln sich die Grundzüge eines Systems:

1. *strukturell:* Systemgliederung in Schichten (Infrastrukturen), Elemente und Beziehungen,
2. *methodisch:* generelle Vorgehensweise bei der Konstruktion des Systems,
3. *inhaltlich:* Zweck, Funktionen und Eigenschaften des Systems und seiner Elemente.

Auch für eine Architektur sind die drei Fragen zu beantworten: Was ist universell? Was ist speziell? Was ist spezialisierbar? Eine Architektur kann speziell auf ein Zielsystem zugeschnitten

sein, sie ist dann nicht übertragbar. Das Ziel hier ist aber keine "universelle Architektur", passend für jedes Zielsystem, sondern eine *spezialisierbare Architektur*, die universelle, für ein breites Spektrum übertragbare Grundzüge enthält und die für ein konkretes Zielsystem spezialisierbar ist. Die Idee ist, übertragbare strukturelle und methodische Grundzüge in einer ***generalisierten Architektur*** festzuschreiben und inhaltliche Ausprägungen für eine Schicht oder ein konkretes Zielsystem erst in einem zweiten Schritt daraus abzuleiten, d.h. zu ***spezialisieren***. Das bedeutet, Ausprägungen von Elementen und Infrastrukturen und damit die Ausführungseigenschaften von Instanzen erst bei der Spezialisierung zuzuordnen.

Man findet derzeit ähnliche Überlegungen auf anderen Gebieten der Informatik, allgemeine, für viele Fälle übertragbare Grundprinzipien, Vorgehensweisen und "Entwurfsmuster" abzuleiten. Für die Softwareherstellung ist das Buch *"Design Patterns"* [Gam94] in der Diskussion. Auch der Vorschlag der *OMG* für eine allgemeine *"Object Management Architecture"* [Sol92] zielt in eine ähnliche Richtung. Für das Gebiet der Betriebssysteme wäre das Buch *"Systemarchitektur"* [Wett93] zu nennen. Trotz unterschiedlicher Zielbereiche ist diesen Ansätzen gemeinsam, aus der Erfahrung immer wiederkehrende Wirkprinzipien, Strukturen und Vorgehensweisen zu identifizieren und zu verallgemeinern. Für ihre Anwendung sind diese dann zu konkretisieren, zu spezialisieren. In gewisser Weise hat diese Arbeit eine ähnliche Intention, den Versuch zu unternehmen, generelle Prinzipien für die Konstruktion anpaßbarer Betriebssysteme abzuleiten und Realisierungsmöglichkeiten am Beispiel des *CHEOPS*-Kerns zu untersuchen.

Erst die Trennung eines Grundmusters von der konkreten Ausprägung für ein spezielles Zielsystem eröffnet die Chance, ein "skalierbares" Grundmusters aufzustellen. Für Betriebssysteme sind traditionell beide Aspekte vermischt. Eine gegebene Architektur impliziert ein konkretes Ausführungsmodell für Instanzen. Instanzen des *Unix*-Kerns besitzen stets getrennte Adreßräume und unterliegen preemptivem Scheduling. Hier sind in der Architektur bereits fundamentale Ausführungseigenschaften festgelegt. Eine Übertragung in eine artfremde Umgebung, etwa für eine Chipkarte, wäre mangels Ressourcen unmöglich und für diesen Zielbereich auch nicht sinnvoll. Ein weiterer Aspekt ist, daß Universalität für neue Betriebssysteme oft dadurch angestrebt wird, daß jeweils die allgemeinsten und oft teuersten Mechanismen für Eigenschaften umgesetzt werden. Auf der Infrastruktur von *BirliX* [Här92] sind *alle* Instanzen charakterisiert durch: multi-threading (schließt single-threading ein), prinzipiell Kommunikation über *RPC* und die Persistenz aller Instanzen. Das hat natürlich Folgen für die Effizienz.

Es gilt vielmehr der Umkehrschluß, nicht Betriebssysteme werden immer universeller, sondern Anpaßbarkeit wird zur universellen Eigenschaft für Betriebssysteme, und das gilt bereits für die Architektur eines Systems.

> " A true application-oriented system evolves, since extensions are only made on demand, namely, when needed to implement a specific system feature that supports a specific application. Design decisions are postponed as long as possible. In this process, system construction takes place bottom-up but is controlled in a top-down (application-driven) fashion. ...
> In its last consequence, applications become the final system extension. The traditional boundary between application and operating system disappears. " [Schrö94], S. 20-22.

1.8.2 Struktureller Entwurf: Überblick

Um die genannten Ziele auch strukturell wiederzugeben, wird eine ***Schichtenarchitektur*** angewandt, die im folgenden in ihre grundlegenden Bestandteile aufgelöst wird.

Modellierungsgegenstand sind die realen Verarbeitungseinheiten eines ablaufenden Systems (→ *Instanzen*), die selbständig Aufträge (→ *Dienste*) ausführen. Instanzen einer Schicht schaffen die Infrastruktur für Instanzen übergeordneter Schichten (→ *Rekursion*). Instanzen weisen

dabei unterschiedliche Ausführungseigenschaften und technische Ausprägungen in verschiedenen Schichten auf. Für das Zusammenwirken (→ *Interaktion*) sind Steuerung und Informationsaustausch (→ *Protokolle*) erforderlich. Verarbeitung und Steuerung sind inhärent aktive Prozesse, die ihrerseits einer Steuerung bedürfen. Daraus resultiert eine Hierarchie von Steuerung und Verarbeitung. Aufträge an das informationsverarbeitende System aus der Umgebung werden intern rekursiv in Teilaufgaben für Steuerung und Verarbeitung in Teilsystemen in verschiedenen Schichten zerlegt.

Die daraus resultierende Systemgliederung läßt sich grob wie folgt beschreiben:

- **vertikale Systemgliederung:**
  - *Rekursion* aufeinander aufsetzender Schichten von *Infrastrukturen* und *Instanzbereichen*;
- **horizontale Systemgliederung:**
  - jede Schicht besteht aus *Instanzbereichen*;
  - ein Instanzbereich ist eine Gruppierung von *Instanzen* innerhalb einer Schicht;
  - eine *Instanz* ist eine autonome, von anderen Instanzen abgrenzbare, dienstausführende Funktionseinheit des Systems; jede Instanz kapselt eigene Daten (Zustände), Programme (Steuerungsinformation) und Prozesse zur selbständigen Ausführung von Diensten;

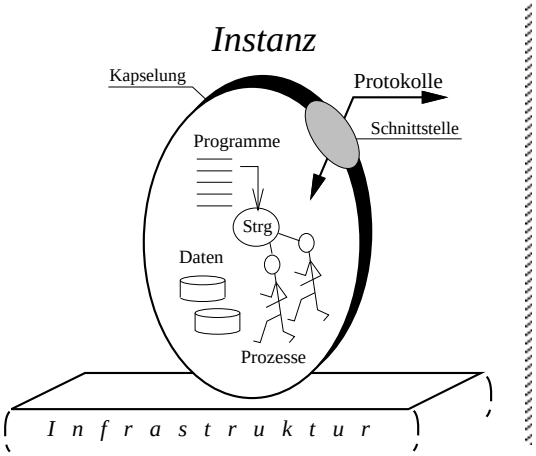


Abb.1.4 Merkmale einer Instanz

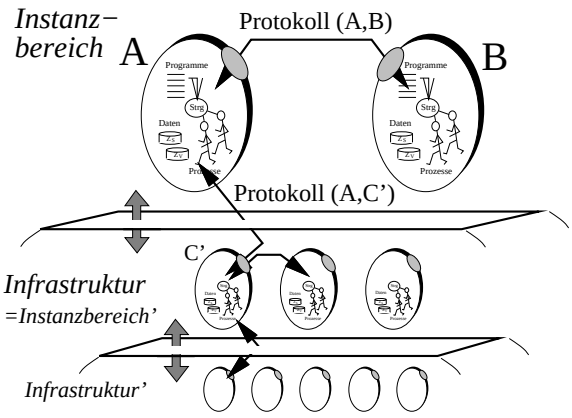


Abb.1.5 Rekursion: Instanzbereiche – Infrastrukturen

(Software-) Instanzen weisen über alle Schichten gemeinsame Merkmale auf:

- ▷ Instanzen sind abgrenzbar (→ *Kapselung*) und damit *identifizierbar*, bezogen auf:
  - Daten (innere Zustände),
  - Programme (Steuerungsinformation),
  - Prozesse (Steuerung und innere Aktivitäten).
- ▷ Die Herstellung einer Instanz erfolgt anhand von Beschreibungsinformation (→ *Typinformation*), die vorher ebenfalls herzustellen und in das System einzubringen ist.

Die ***technische Umsetzung*** von Kapselung, Zuständen, Steuerung und Prozessen ist ***Gegenstand der Spezialisierung*** und aus der generalisierten Architektur ausgenommen !

Instanzen sind als Ausführungseinheiten in konkreten Ausprägungen prinzipiell bekannt: *Unix-Prozesse*, *Mach-Tasks* oder *BirliX-Teams*. Im objektorientierten Umfeld findet man *aktive Objekte* oder *Concurrent Objects* [YT87], die eine vergleichbare Ablaufstruktur darstellen. Man kann sich auch auf Hewitt's *Actors* [Hew77, Agh85] beziehen, die letztlich eine ähnliche Modellierung darstellen. Auch das Zurückführen von Informationsverarbeitung auf Diensterbringung wird seit langem praktiziert. Darin liegt das Grundprinzip des *Client-Server-Modells* und der *Schichtenstruktur* in Betriebssystemen. Das ***Client-Server-Modell** erscheint **in Verbindung mit einer Schichtenstruktur*** als übergreifendes Grundmuster sowohl für den Anwendungs- als auch für den Betriebssystembereich attraktiv. Es greift als Strukturprinzip ablaufender Systeme auch Elemente der Softwaretechnik auf (Kapselung, Abstraktion: Code + Daten), erweitert diese um Abläufe (Prozesse: → *Dienste*) und gibt eine Methode für das Zusammenwirken (Interaktion) vor (Steuerungsebenen, Protokolle).

Insofern setzt die hier vorgestellte Architektur auf akzeptierten und weithin bekannten Strukturprinzipien auf. Das Neue liegt dabei nicht in den Einzelaspekten der beteiligten Strukturen, sondern in deren ***Zusammenführung** und **Kombination***, die es mit der hier angestrebten Konsequenz für Betriebssysteme bislang nicht gibt. Für den *CHEOPS*-Kern wird diese Architektur bis zur Hardwareinfrastruktur ausgeführt. So gibt es beispielsweise eine spezielle Art von Instanzen, die in der untersten Schicht Dienste zur Behandlung von Unterbrechungen ausführt.

Traditionell wird in einem System oft nur eine Ebene (Schicht) betrachtet, in der es eine bestimmte Grundabstraktion des Ausführungsmodells gibt (z.B. *Unix-Prozesse* oder *aktive Objekte*). Mit der jeweiligen Grundabstraktion sind stets konkrete Ausführungseigenschaften verbunden, die eine Übertragung in andere Ebenen oder in artfremde Zielsysteme unmöglich machen. Genau hier liegt der Ansatzpunkt, die Grundabstraktion von konkreten Ausprägungen zu trennen und für einen konkreten Zielbereich erst in einem *zweiten Schritt zu spezialisieren*.

Aus dem geeigneten Zusammenführen verschiedener Grundmodelle ergeben sich Synergieeffekte:

- ***Client-Server-Modell:*** abgeschlossene, aktive Einheiten, die Dienste ausführen; vor allem von *Unix* und den μ-Kernen bekannt; ***neu:*** auch innerhalb der Infrastruktur; "skalierbar" durch Abstraktion und Spezialisierbarkeit konkreter Ausführungseigenschaften;
- ***Schichtenmodell:*** [Dij71], "vertikale" Systemgliederung in Schichten von Prozessen und Betriebsmitteln (→ *abstrakte Maschinen*); ***neu:*** "horizontale" Gliederung innerhalb einer Schicht in Instanzen, die innere Zustände und Verarbeitung kapseln; auf diese Weise wird die traditionelle Trennung: Prozeß–Betriebsmittel in Instanzen auflösbar;
- ***Instanzen/Infrastruktur:*** [Wett93], Modellierung von Systemen als interagierende Instanzen auf einer Infrastruktur; Abstraktion von konkreten Ausführungseigenschaften; ***neu:*** mehrstufige Rekursion von Instanzbereichen und Infrastrukturen;
- ***Concurrent Objects / Actors:*** [YT87, Hew77], selbständig agierende Objekte kapseln neben Code und Daten auch Prozesse; ***neu:*** Einbeziehung der Infrastruktur, d.h. Ausdehnung auf mehrere Schichten mit verschiedenartigen Ausführungseigenschaften;
- ***Meta-Objekte, Meta-Objekt-Protokolle (MOP):*** [Kicz91], dynamische Änderbarkeit von Ausführungseigenschaften durch Manipulation in der Infrastruktur, so daß auch reflexive Techniken umsetzbar werden; ***neu:*** "aktive" Meta-Objekte in Form von Instanzen (vgl. auch *Apertos* [Yok93]).

Die aktuelle Diskussion um *Meta-Level-Architekturen* zeigt das gegenwärtige Interesse an solchen Architekturen [IWOOOS95] und vor allem an *Wegen für ihre praktische Umsetzung*.

1.8.3 Absehbare Vorteile

Idealerweise sollte der Entwurf und die Herstellung eines Systems nach folgenden Schritten ausgeführt werden können:

1. Welche Anforderungen stellt die Anwendung?
2. Welche Art von Ausführungseinheiten (Instanzen) ist am besten geeignet?
3. Welche Infrastruktur bzw. Umgebung ist dafür nötig?
4. Wie läßt sich Infrastruktur dafür herstellen, wiederverwenden oder anpassen?

Durch die im vorangegangenen Abschnitt kurz charakterisierte Architektur wird methodisch keine Trennung mehr zwischen Anwendungs- und Infrastrukturentwicklung vorgenommen. Aus dieser Architektur ergeben sich einige vorteilhafte Eigenschaften:

- Die *homogene Grundstruktur über alle Ebenen* des Systems schafft Übersicht und überwindet den Modell- und Strukturbruch in traditionellen Systemen.
- Durch Abtrennung struktureller Eigenschaften von Ausführungseigenschaften wird eine weitreichende Skalierbarkeit ermöglicht (→ *"scalable Client-Server Architecture"*).
- Die oft gegebene konzeptionelle Mehr-Ebenen-Struktur von Anwendungen kann direkt in Schichten des ablaufenden Systems abgebildet werden. Durch das explizite Herausstellen von Schichten werden *verschiedenartige Ausführungsmodelle* in Systemen integrierbar, was der Strategie des Verbergens von Verschiedenartigkeit entgegengesetzt ist.
- Infrastrukturen können durch *statischen oder dynamischen Einbau von Instanzen* auf Anwendungen zugeschnitten werden. Infrastrukturen müssen im ablaufenden System nur die tatsächlich benötigten Komponenten enthalten. Mehrere Anwendungen im selben System können mit *individuellen Infrastrukturen* ausgestattet werden.
- In der Anwendung sind *Effizienzvorteile* zu erwarten, wenn die Option wahrgenommen werden kann, kritische Komponenten auch in tieferen Schichten eines Systems zu plazieren bzw. zugeschnittene, optimierte Infrastrukturen herzustellen.
- Instanzen sind sowohl *Einheiten der Modellierung* als auch *Einheiten des ablaufenden Systems*. Die Herstellung von Software wird dahingehend unterstützt, daß Elemente einer Sprache explizit in Ausführungseinheiten (Instanzen) abgebildet werden und damit eine *Integration von Software- und Ablaufstruktur* möglich wird. Code und Daten einer Instanz können nach objektorientierten Methoden entworfen und hergestellt werden. Es erfolgt dann eine *explizite Abbildung:* Objekte → Instanzen.
  Für die Architektur steht aber das Gesamtsystem im Vordergrund: Schichten von Instanzen bilden rekursiv die Infrastrukturen für jeweils übergeordnete Instanzbereiche.

1.9 Einordnung in das *CHEOPS*-Projekt

Das *CHEOPS*-Projekt [Kal93] befaßt sich mit strukturellen Fragen für anwendungsanpaßbare Betriebssysteme und mit deren praktischer Umsetzbarkeit. Es sollen Praktikabilität, aber auch Probleme und Schwächen aufgezeigt werden. Zu diesem Zweck wird der *CHEOPS*-Kern[6] implementiert.

[6] früher COSAI-Kern für *"Customizable Operating System with Adaptable Multi-Layered Infrastructures"*, vgl. auch Beitrag zur *IWOOOS'95* [Grau95a]

Neben der in dieser Dissertation behandelten Thematik sind folgende Themen weitere Schwerpunkte des *CHEOPS*-Projekts [Kal96a]:

- adäquate Widerspiegelung von *C++*-Sprachelementen im ablaufenden System [Schu96],
- feiner granulare dynamische Anpaßbarkeit für Objekte innerhalb von Instanzen,
- dynamisches Rekonfigurierungsmanagement für adaptives Verhalten [Woh96].

In dieser Dissertation wird die grundlegende Architektur entwickelt und deren praktische Umsetzbarkeit in einer realen Betriebssystem-Implementierung gezeigt. Die weiterführenden Themen werden an diesem Experimentalbetriebssystem untersucht.

1.10 Implementierung: Der *CHEOPS*-Kern

Der *CHEOPS*-Kern ist ein Experimentalkern zur Untersuchung der Umsetzbarkeit der vorgestellten Architektur und deren Wirkungen. Es stehen nicht primär die Funktionen dieses Kerns im Vordergrund, sondern Implementierungsaspekte.

Der *CHEOPS*-Kern ist als traditioneller μ-Kern konzipiert, der auf Basis von Instanzen dynamische Anpaßbarkeit unterstützt. Die Idee ist, Instanzen innerhalb der Kern-Infrastruktur ebenso dynamisch hinzuzufügen oder zu entfernen, wie es für den Start und das Ende von Anwendungsprozessen bekannt ist. Da es sich hier aber um Instanzen der Infrastruktur handelt, ergeben sich weitreichende Wirkungen für den übergeordneten Instanzbereich. Einer Infrastruktur können durch den dynamischen Start von Instanzen neue, anwendungsbezogene Dienste hinzugefügt werden. Dieser Grundgedanke liegt auch den eingangs genannten amerikanischen Forschungsprojekte zugrunde. Der Schwerpunkt liegt dort bei der Sicherung der Integrität dynamisch erweiterbarer Kerne, die in den meisten Fällen monolithisch ausgeprägt sind. In dieser Arbeit stehen strukturelle Fragen und die Umsetzung in eine rekursive Instanzen-Infrastruktur-Architektur im Vordergrund.

Der *CHEOPS*-Kern wird als stand-alone Kern auf zwei Hardwareplattformen implementiert:

- *CADMUS-9700* Workstation (asymmetrische 2-CPU-Maschine auf Basis *m68020*) und
- *PC* (Intel 486, protected mode).

Zum Abschluß dieses Überblicks über den Gegenstand, das Ziel und die Einordnung dieser Dissertation sollen zwei Zitate aus [Kicz91] zu Arbeiten am *M.I.T.* über Meta-Objekt-Protokolle für *CLOS* angegeben werden, die ein Grundanliegen charakterisieren, das hier für Betriebssysteme aufgegriffen wird:

> "Traditionally, (language-) designers are expected to produce languages with well-defined, fixed behaviors (or 'semantics'). Users are expected to treat these languages as immutable black-box abstractions, and to derive any needed flexibility or power from constructs built on top of them."

> "The metaobject protocol approach, in contrast, is based on the idea that one can and should 'open languages up', allowing users to adjust the design and implementation to suit their particular needs. In other words, users are encouraged to participate in the language design process."

Kapitel 2

Anpassung in Betriebssystemen – eine Bestandsaufnahme

Anpaßbarkeit ist die Eigenschaft eines Systems, eines Konzepts, einer Architektur, in eine andere Umgebung oder einen anderen Kontext übertragbar zu sein bzw. auf veränderte Bedingungen oder Anforderungen in einem Umfeld durch Anpassung von Eigenschaften reagieren zu können. Anpassung hat mit Wieder- oder Weiterverwendung übertragbarer, universeller Eigenschaften und mit dem Herstellen der Paßfähigkeit einzelner, spezifischer Merkmale zu tun.

Betriebssysteme waren und sind permanenter Gegenstand von Anpassungsprozessen unterschiedlicher Art. Anpassung betrifft Betriebssysteme sogar in höherem Maße als andere Software. Einerseits sollen Betriebssysteme technologische Fortschritte für Anwendungen erschließen und veränderten Nutzungserfordernissen Rechnung tragen, andererseits besteht der legitime Anspruch von Anwendern und Kunden, daß Betriebssysteme funktional und strukturell, d.h. in bezug auf Architektur, weitgehend invariant gegenüber dem Wandel im Umfeld sind, damit bestehende Anwendungen weiterhin ablaufen können. Anforderungen an Betriebssysteme ändern sich aber auch im Lauf der Zeit. Für universelle Betriebssysteme sind Dezentralisierung, Verteilung und Spezialisierung in heterogenen Landschaften aktuelle Trends. Das erfordert Offenheit und Interaktionsfähigkeit von Systemen verschiedener Hersteller. Hinzu kommen Technologien, die neue Anwendungsmöglichkeiten schaffen, wie mobile und multimediale Systeme.

In der Integration der vielfältigen, teils divergierenden technologischen Entwicklungen und Nutzungsarten lag in der Vergangenheit einer der Hauptgründe zur Schaffung einer absorbierenden Softwareinfrastruktur zur Entkopplung der Anwendungen von diesen Einflüssen. Diese Zielstellung besteht nach wie vor, damit Investitionen von Kunden in Anwendungen auf längere Sicht Bestand haben können. Das ist seit vielen Jahren die Geschäftsgrundlage der etablierten proprietären Betriebssysteme in der kommerziellen Datenverarbeitung: *Siemens BS2000* [Gö90], *IBM MVS* [Eld93], *IBM OS/400* [Law93] oder *DEC VMS* [Mil92]. Dem Betriebssystem kommt dabei eine Doppelrolle zu, einerseits ein System über lange Zeiträume und Generationen von Hardware- und Softwaretechnologien hinweg, funktional zu konservieren, andererseits aber neue Technologien und verbesserte Leistungsparameter für Anwendungen zu erschließen. Der erste Aspekt begründet, warum Betriebssysteme äußerst *langlebige Systeme* sind, der zweite belegt, warum Betriebssysteme *permanenter Gegenstand von Anpassungsprozessen* sind. Bereits aus dieser globalen Betrachtung wird die stete Aktualität von Anpassung in Betriebssystemen deutlich.
Die Methode dafür ist die Schaffung einer *abstrahierten, idealisierten Ausführungsmaschine* als Infrastruktur für Anwendungen. Die Mittel sind die Zentralisierung der Verwaltung von Betriebsmitteln, die Transformation neuer Betriebsmittel und das Erzeugen und Verwalten paralleler Abläufe in Form von Prozessen [Kal90]. Die Funktion und die Eigenschaften dieser Ausführungsmaschine sind in hohem Maße invariant gegenüber Änderungen im Umfeld.

Für Betriebssysteme können drei Entkopplungsfunktionen zusammengefaßt werden:

- Entkopplung bestehender Anwendungen von technologischen Fortschritten im Sinne weiterer Ablauffähigkeit von Anwendungen (→ funktionale Invarianz der abstrakten Maschine über lange Zeiträume, Kompatibilität),
- Entkopplung der Anwendungen von der Kompliziertheit und dem Technologiewandel der Hardware durch Idealisierung und Abstraktion von Diensten und Eigenschaften der abstrakten Maschine,
- Entkopplung gleichzeitig ablaufender Anwendungen voneinander durch zentrale Betriebsmittelvergabe und Ablaufsteuerung.

Auf die Aufgabenteilung für die Herstellung, Wartung und den Betrieb von Systemen wurde bereits hingewiesen. Anpassungsprozesse für ein Betriebssystem konzentrieren sich beim Hersteller. Er gibt vor, welche Anpassungen durchgeführt werden und welche Möglichkeiten Dritten dafür eröffnet werden. Es finden nur Anpassungen statt, die ein breites Kundeninteresse finden. Das steht im Widerspruch zu einer Welt, die durch fortschreitende Diversifizierung geprägt ist. Andererseits ist es aber auch nicht sinnvoll, übergreifende Betriebssysteme aufzugeben und ausschließlich spezielle Systeme herzustellen. Der aktuelle Trend ist aber dadurch geprägt, mehr Anpassungskompetenz zu Anwendungsherstellern und Endanwendern zu verlagern.

2.1 Facetten für Anpassung in Betriebssystemen

Zunächst sollen traditionelle Anpassungsprozesse für universelle Betriebssysteme weiter untersetzt werden. Die Einflußgrößen können grob in drei Kategorien eingeteilt werden:

- Anpassung eines Betriebssystems an den vorgesehenen Einsatzzweck,
- Anpassung der Funktion und der Eigenschaften der abstrakten Maschine,
- softwaretechnologische Anpassungen bei der Herstellung der Betriebssystemsoftware.

In Abbildung 2.1 sind Einflußfaktoren für Anpassungsprozesse dargestellt. Für Anwendungen ist wesentlich, ob Anpassungen am Betriebssystem für sie konfliktfrei bleiben, d.h. sich in besseren Betriebssparametern niederschlagen, aber keine Auswirkungen auf die weitere Ablauffähigkeit von Anwendungen haben. Das Betriebssystem bestimmt dabei als Infrastruktur nicht nur die Funktionen und Dienste für Anwendungen, sondern auch das generelle Ausführungsmodell.

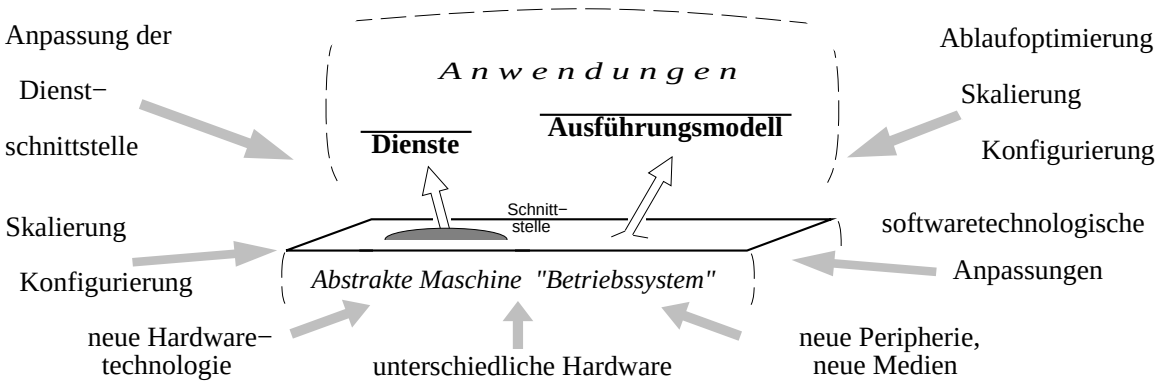


Abb.2.1 Einflußfaktoren für Anpassungsprozesse in Betriebssystemen

2.1.1 Anpassung an den vorgesehenen Einsatzzweck

Schon die Herstellung von *Anwendungssoftware* kann für ein universelles Rechensystem (Hardware + Betriebssystem) als Anpassungsprozeß an den Einsatzzweck betrachtet werden.
Zu Anpassung in dieser Kategorie zählt jedoch auch die *Ablaufoptimierung* für bestimmte Anwendungsklassen zur Erschließung weiterer Leistungspotentiale im Gesamtsystem. Hierfür müssen Betriebssystemhersteller Mittel bereitstellen. Ein Beispiel ist das *Performance Control Subsystem* (*PCS*) für *BS2000* [Mes94]. Aus Beobachtung und dem Bestimmen von Anforderungsprofilen von Anwendungen können optimierte Systemparameter abgeleitet und eingestellt werden. Beeinflußbare Größen können auch Verwaltungsstrategien für Betriebsmittel sein.
Eine weitere Art von Anpassung wird als *Skalierung* bezeichnet und bedeutet die für einen Einsatzzweck angemessene Wahl der Verarbeitungsleistung eines Systems. Das spiegelt sich vor allem in unterschiedlichen Ausstattungsmerkmalen der Hardware wider. Anwender sollen dabei die Option haben, mit zunehmenden Verarbeitungsvolumen in höhere Leistungsklassen "mitwachsen" zu können, unter Wahrung der Ablauffähigkeit bestehender Anwendungen auf unterschiedlich leistungsfähigen Maschinen. Skalierung bezieht sich auf die Anzahl von Prozessoren, Platten oder den Hauptspeicherausbau eines Systems.
Anpassung an den Einsatzzweck ist eng mit *Konfigurierung* verbunden. Je nach Einsatzfall können verschiedene Komponenten in das Betriebssystem eingebunden werden oder entfallen, beispielsweise zur Bedienung spezieller Peripherie. Konfigurierung ist heute die typische Form, über die Hersteller erlauben, individuelle Kundenanforderungen in vorgegebenen Grenzen zu berücksichtigen. Neben der Konfigurierung ist *Parametrisierung* eine Möglichkeit, das Verhalten des Betriebssystems zu beeinflussen und damit anzupassen.

Die vier wesentlichen Aspekte von Anpassung in dieser Kategorie sind zusammengefaßt:

- Herstellung und Ablauf von *Anwendungen* auf der *Infrastruktur Betriebssystem*,
- *Ablaufoptimierung* für spezielle Anwendungsklassen (Tuning),
- *Skalierung* von Ausstattung und Leistung der Hardware und auch des Betriebssystems,
- *Konfigurierung* und *Parametrisierung* des Betriebssystems.

2.1.2 Anpassung der abstrakten Maschine "Betriebssystem"

Eine ganz andere Art von Anpassung resultiert aus zunehmender Dezentralisierung und Vernetzung von Informationsverarbeitung. Rechensysteme sollen in einer heterogenen Landschaft interaktionsfähig sein, unabhängig vom Hersteller und vom Mainframe bis hin zum PC.
Das Zeitalter *Offener Systeme* begann in den siebziger Jahren mit *Unix* [RT74]. Kennzeichnend für diesen Prozeß ist die Abkopplung des Betriebssystems von der Hardware eines Herstellers. Diese Art Universalität hat Vorteile für Kunden und trägt zu einer wachsenden Akzeptanz dieser Systeme bei. Ein *Offenes System* wird durch eine *abstrakte Ausführungsmaschine* definiert (Steuerung, Dienste, Ausführungseigenschaften, Schnittstellen, Protokolle), deren Implementation und Leistungswerte jedoch variieren können. Bei *Offenen Systemen* geht es um zwei wesentliche Aspekte. Zum einen spielt *Portabilität von Anwendungssoftware* durch Definition herstellerübergreifender Programmierschnittstellen (*API*) auf Basis fester Ausführungseigenschaften eine Rolle [Moo90], zum anderen geht es um *Interoperabilität* oder Interaktionsfähigkeit ablaufender Anwendungen in einem Verbund heterogener Systeme.

Für *Unix* haben sich im Verlauf seiner Entwicklung eine Reihe von Anpassungen der abstrakten Maschine an jeweilige Gegebenheiten ergeben, ausgehend vom "*Ur-Unix*" [RT74], über *System V Unix* [Bac86] und *BSD Unix* [Leff88] bis zum aktuellen *System V Release 4* [GC94] [1]. Dieser Anpas-

[1] In [Bac86], [Leff88] und [GC94] ist die Historie von *Unix* umfassend dargestellt.

sungsprozeß wurde von Auseinandersetzungen verschiedener Organisationen und Konsortien um herstellerübergreifende Standards begleitet [Bol89]. Dazu zählen die frühen Ansätze */usr/group* des */usr/group Standards Committee* [group84], der Vorschlag des *ANSI X3J11 Language Subcommittee* für die Sprache C [ANSI84], der später zum *ANSI-C* Standard *X3.159-1989* [ANSI89] wurde und als verbindliche Festlegung der Sprache C und der Standard-C-Bibliothek gilt, die auch Systemrufe von *Unix* enthält. Vom ursprünglichen Initiator von *Unix*, *AT&T*, stammte *SVID*[2] [AT&T85]. Später erlangten die *POSIX*[3]*-Suite*[4] der *IEEE* [Zlo91] und die Vorschläge der *X/Open*[5] praktische Relevanz. Aktuell sind *IEEE 1003.x (POSIX)* [IEEE88], *Unix System V Release 4* [AT&T90] und der *X/Open Portability Guide (XPG4)* [X/Open92] von Bedeutung.

Die Auseinandersetzung um herstellerübergreifende Standards ist letztlich auch ein Anpassungsprozeß zur Festlegung der Eigenschaften und Funktionen der abstrakten Maschine. Er stellte in dieser Form ein Novum in der Geschichte der Betriebssysteme dar. Andere Standardisierungsvorschläge zu Dienstschnittstellen fanden keine breite Akzeptanz, wie der des japanischen *TRON*-Projekts[6] [Sak87] oder der *IEEE-Standard 855*[7] [IEEE85].

Ein *Offenes System* bezieht sich aber nicht allein auf das Betriebssystem, sondern in gleicher Weise auf Kommunikationsprotokolle, Netzwerkinfrastruktur, Dienste und Anwendungen bis hin zur Nutzerschnittstelle [Ein89]. Für das *Internet* sind eine Vielzahl von Diensten notwendig, die unabhängig und nicht Teil der Definition eines Betriebssystems sind. Der Prozeß zur Schaffung *Offener Systeme* schließt auch Programmier- und Verwaltungssysteme ein, z.B. *DCE* der *OSF* [OSF92], *CORBA* der *OMG* [OMG91, OMG95] oder *CSA* [CSA90].

Der Trend hin zu *Offenen Systemen* ist heute so stark, daß selbst die klassischen Mainframe-Betriebssysteme ihren Weg in diese Welt suchen. Für das Betriebssystem *BS2000* wurde das 1992 mit der Fortführung unter dem Namen *BS2000/OSD* (*OSD – Open Systems Direction*) deutlich gemacht. In der Folge wurde 1994 eine *OSF/DCE*-Anbindung hergestellt und 1995 ein *POSIX*-Subsystem (1003.1/2) und eine *NFS*-Anbindung für *BS2000/OSD* geschaffen [Sti95]. Andere Hersteller von Mainframe-Betriebssystemen unternehmen ähnliche Anstrengungen, z.B. *IBM* mit *POSIX* für *MVS*. Auch für *Windows NT* gibt es ein *POSIX*-Subsystem [Cust93].

Um speziellen Anforderungen gerecht zu werden, gab es und gibt es für *Unix* eine Reihe spezialisierter Ableitungen (*Derivate*), die an bestimmte Einsatzzwecke angepaßt sind, etwa für Echtzeitanwendungen, z.B. *REAL/IX* [FGG+91]. Derivate füllen bestimmte Freiheitsgrade der abstrakten Maschine und passen diese für einen Einsatzzweck an, wie Eigenschaften des Schedulings, der Unterbrechungsbehandlung oder der Interprozeß-Kommunikation (IPC) für Echtzeitfähigkeit. Sie erfüllen sonst aber vollständig die funktionalen Anforderungen der Maschinendefinition.

Neben Derivaten gibt es auch speziell für Echtzeitaufgaben hergestellte kleine Echtzeitkerne, die vorwiegend in Steuerungen für Maschinen und Anlagen eingesetzt werden (→ *Embedded Systems*). Beispiele sind *PXROS* [Mau96], *QNX* [Hil92] oder *pSOS* [PSOS93]. Diese Systeme sind modular aufgebaut und bieten verschiedene Aufsätze für einen μ-Kern, um Interaktionsfähigkeit zu *Unix*-Systemen herzustellen oder selbst *Unix*-Dienste für lokale Anwendungen auszuführen. Der Anwender hat die Wahl, den Ausbaugrad des Systems in Richtung *Unix*-Fähigkeit selbst festzulegen. Für *QNX* belegt deshalb der reine μ-Kern nur ca. 10 kByte.

Herstellerübergreifende Standardisierungen sind auf der einen Seite äußerst wichtig und werden von Kunden gefordert, auf der anderen Seite tragen sie eine Tendenz nach *umfassender Universalität* in sich, was sich im großen Umfang der Standards zeigt und sich in der Folge auch im

[2] *System V Interface Definition (SVID)*
[3] *Portable Operating System Interface (POSIX)*
[4] *IEEE Std 1003.1-1988* [IEEE88] bzw. *IEEE Std 1003.1-1990, ISO/IEC 9945-1: 1990* [IEEE90]
[5] *X/Open Portability Guide (XPG3, Issue 2 [X/Open87] and Issue 3 [X/Open89])*
[6] *The Realtime Operating System Nucleus (TRON)*
[7] *Trial-Use Standard Specification for Microprocessor Operating System Interfaces (MOSI)*

Umfang der Betriebssysteme niederschlägt, die diese Standards umsetzen. Diese Betriebssysteme erfordern eine Mindestverarbeitungsleistung, die das Einsatzspektrum auf eine bestimmte Klasse von Systemen beschränkt. Der schwerfällige Standardisierungsprozeß steht auch dem hohen Innovationstempo technischer Entwicklungen entgegen. Da die beteiligten Hersteller im Wettbewerb stehen, wird versucht, früh de-facto Standards für neue technische Entwicklungen zu etablieren, die an längerfristigen Zielen von Standardisierungsgremien vorbeigehen können.
Die Anpassung der abstrakten Maschine betrifft sowohl innere Eigenschaften, die nichts an der Funktion ändern als auch Eigenschaften, die nach außen durch neue oder veränderte Dienste wirksam werden. Ein Beispiel innerer Anpassung der abstrakten Maschine für *Unix* ist der Übergang von Einprozessor- zu Multiprozessor-Technologie. In [Schim94] werden dafür wesentliche Voraussetzungen und Techniken genannt. Auch äußere Anpassungen lassen sich für das Beispiel *Unix* angeben, wie das spätere Hinzufügen der *System V IPC*-Mittel (*shared memory*, *messages*, *semaphores*) und der *Streams* durch *AT&T* oder der *sockets* für *BSD-Unix*.

Die *abstrakte Maschine Betriebssystem* ist Gegenstand vielfältiger Anpassungsprozesse:

- *Offene Systeme* erfordern die herstellerübergreifende Definition einer abstrakten, idealisierten Ausführungsmaschine (Steuerung, Dienste, Ausführungseigenschaften, Schnittstellen, Protokolle), um *Interoperabilität* und *Portabilität* von Anwendungen zu erreichen.
- Der Definitionssprozeß dieser Maschine unterliegt dabei teils widerstrebenden Herstellerinteressen und steht außerdem im Zwang, dem permanenten technischen und anwendungsbedingten Wandel zu folgen.
- Für Anwender ist relevant, ob Anpassungen der abstrakten Maschine
  - konfliktfrei für Anwendungen sind (Dienste, Ausführungsmodell) oder ob sie auch
  - Änderung in den Anwendungen hervorrufen bzw. neue Anwendungen erfordern.

2.1.3 Softwaretechnologische Aspekte

Betriebssysteme sind nicht nur Infrastruktur für den Ablauf von Anwendungen, sondern bestehen aus Software und sind damit *Gegenstand der Herstellung von Software*. Sie unterliegen damit generellen softwaretechnologischen Gesetzmäßigkeiten und verlangen die *"praktische Anwendung wissenschaftlicher Erkenntnisse für die wirtschaftliche Herstellung und den wirtschaftlichen Einsatz qualitativ hochwertiger Software."* [Pomb93], S. 3.
Software wird zu einem Zeitpunkt unter einem bestimmten Stand der Technologie hergestellt. Es ist später nur mit großem Aufwand möglich, einmal hergestellte Software mit der fortschreitenden Technologie mitzuführen (neue Sprachen, Beschreibungs- und Verwaltungsmethoden, Werkzeuge etc.). Betriebssysteme sind langlebige Systeme, über Generationen von Hardware- und Softwaretechnologien hinweg. Einzelkomponenten von Betriebssystemen wurden zu verschiedenen Zeiten nach dem jeweiligen Stand der Technologie hergestellt und müssen immer wieder in neue Technologien überführt werden (→ *Redesign-Phasen*). Diese softwaretechnologischen Anpassungsprozesse tangieren primär den Betriebssystemhersteller und sind sehr aufwendig, ohne daß sich ein bezahlter Vorteil für Kunden ergibt. Andererseits verschärft sich das Problem für den Hersteller mit zunehmendem Umfang der Software und läßt sich ab einem bestimmten Punkt praktisch nicht mehr auflösen. Das Schlagwort *Softwarekrise* umschreibt diesen Zustand und betrifft insbesondere auch lange existierende Betriebssystemsoftware.
Die in 2.1.1 und 2.1.2 genannten Gründe für Anpassungsprozesse in Betriebssystemen schlagen sich natürlich in der Anpassung, Änderung bzw. dem Neuerstellen von Betriebssystemsoftware nieder. Das gilt insbesondere für die Software *Offener Betriebssysteme*, die nicht nur die Portierbarkeit von Anwendungen erlauben muß, sondern selbst in hohem Maße portabel sein sollte, um sie auf immer neue Hardware zu übertragen. Generell spielt daher auch für Betriebssysteme *Wiederverwendbarkeit* und die Struktur von Software eine wichtige Rolle.

Softwaretechnologie ist eng mit *Modularisierung, Struktur und Verwaltung der Software* verbunden. Modularisierung beginnt bei der Analyse (Dekomposition eines Problembereichs, Identifikation abgrenzbarer Einheiten und Beziehungen), setzt sich über die Modellierung und den Entwurf, die Programmierung und Komposition der Komponenten eines Programmsystems fort. Die Grundlage ist, funktional abgeschlossene Einheiten zu identifizieren und beschreibungstechnisch abzugrenzen. Die Kriterien dafür werden durch die Entwurfsziele bestimmt [Par72]. Modularisierung setzt adäquate Beschreibungsmittel voraus, mit denen sich die Abgrenzung von Modulen und die Beziehungen zwischen Modulen ausdrücken lassen (→ *Benutzt-Relation* (*uses*), *Aggregations-Relation* (*is part of*), *Bezugnahme-Relation* (*references*)). Ein guter modularer Entwurf zeichnet sich dadurch aus, daß es möglichst wenige Beziehungen zwischen Modulen gibt bzw. daß diese einer Ordnung unterliegen, etwa einer Hierarchie, um zyklische Abhängigkeiten zu vermeiden. Modularisierung enthält nicht nur den strukturellen Aspekt, sondern ist auch unter dem Begriff *information hiding* ein Mittel für Abstraktion (Schnittstellen, Verbergen der Implementation). Die Techniken dafür sind bekannt [Par72, Mey88, Coad91, Rumb91].

Für die Implementierung von Betriebssystemen setzt das maschinenunabhängige Sprachen voraus (z.B. *Unix* → *C* [KR78], *Spring* → *C++* [SE90] oder *BirliX* → *Modula-2* [Wir88]). Besonders im Mainframe-Bereich liegt schon darin ein Problem, da große Teile der Altsoftware in Assembler implementiert und damit auf einen bestimmten Prozessortyp fixiert sind. Im Projekt *SUNRiSE* [Mes95] wird dieses Problem aktuell für die Migration (Portierung) des Betriebssystems *BS2000* weg von der ursprünglichen, *IBM-390*-kompatiblen Hardware auf Standard-RISC-Hardware (*MIPS*-Multiprozessor, *SCSI*-Peripherie, *Unix*-Subsysteme für E/A) behandelt. Es werden vielfältige Techniken angewandt, wie Assembler-Quellcode-Transformation, die *IBM-390*-Emulation für den Ablauf bestehender Assembler-Anwendungen und die Ausführung von *IBM-390*-E/A-Aktivitäten durch autonome *Unix*-Subsysteme.
Es besteht jedoch nicht allein die Abhängigkeit von der jeweiligen CPU, die man durch Hochsprachen weitgehend eliminieren kann. Die Verschiedenartigkeit der anderen Hardwarekomponenten ist in viel höherem Maße relevant. Die Abgrenzung hardwareabhängiger von hardwareunabhängigen Komponenten der Betriebssystemsoftware ist daher ein wichtiger technologischer Aspekt.

Für die Software von Betriebssystemen ist es daher sinnvoll, eine Ordnung über den Modulen nach dem Grad ihrer Hardwareabhängigkeit herzustellen, etwa in Form funktionaler Schichten. Beispiele aus dem *System V Release 4* [GC94] auf Basis der Sprache *C* sind der *Hardware Address Translation Layer* (*HAT* → *"abstrakte MMU"*), die *vnode*-Abstraktion für Files oder das *Device Driver Interface (DDI)* bzw. das *Device Kernel Interface (DKI)*. In *Windows NT* [Cust93] ist all das in einer funktionalen Schicht *Hardware Abstraction Layer (HAL)* zusammenfaßt. Die Entkopplungsfunktion des Betriebssystems für Anwendungen von der Hardware gilt folglich in abgeschwächter Form auch für Komponenten innerhalb des Betriebssystems.

Bei objektorientierten Sprachen kann Hardwareunabhängigkeit durch Ausfaktorisieren in Oberklassen (Generalisierung) mehrstufig widergespiegelt werden (*is_a*-Relation zwischen Klassen). Die Umkehrung ist das Ableiten von Unterklassen (Spezialisierung). Die Technik der *Klassenbibliotheken* oder *Klassen-Frameworks* wurde aus der Softwaretechnik für die Herstellung von Betriebssystemsoftware übernommen und u.a. für *Choices* [Rus91] auf Basis von *C++* und für *ETHOS* [Szy92] auf Basis *Oberon* angewandt. Bei nicht-objektorientierten Sprachen kann die mehrstufige Gliederung auch außerhalb der Sprache umgesetzt werden, wie es beispielsweise für *BirliX* [Här92] mit *Modula-2* praktiziert wurde (*VCS* [Här90b]). Auch der Entwurf von Betriebssystemen nach objektorientierten Methoden wurde und wird weiter untersucht [CLK93].

Fazit, die Herstellungs- und Verwaltungstechnologie der Betriebssystemsoftware

- sagt nichts über die Funktion und die Eigenschaften der abstrakten Maschine aus, und
- beim heutigen Stand der Technik hat Modularisierung und Struktur der Software nichts mit der Struktur im später ablaufenden System zu tun.

Softwaretechnologische Aspekte für die Herstellung von Betriebssystemen sind zusammengefaßt:

- ▷ Betriebssystemsoftware entsteht wie andere Software unter einem bestimmten Stand der Herstellungstechnologie. Aufgrund der Langlebigkeit dieser Software ist es schwierig, angesammelte Bestände immer wieder mit der Technologiefolge mitzuführen.
- ▷ Betriebssystemanpassungen bedingen auch Anpassungen der Betriebssystemsoftware.
- ▷ Der Wiederverwendungsgrad von Software kann erhöht werden, wenn eine klare Modularisierung in funktional abgrenzbare Komponenten vorliegt. Außerdem ist eine Klassifizierung nach dem Grad an Hardwareabhängigkeit sinnvoll, und es können Einschränkungen für die Relationen zwischen Modulen getroffen werden (Hierarchien, Schichten). Die Möglichkeiten dafür werden maßgeblich von den verwendeten Programmiersprachen und Verwaltungssystemen bestimmt.
- ▷ Die Herstellung und Verwaltung der Betriebssystemsoftware hat aber nichts mit den Eigenschaften des Betriebssystems und der Struktur im später ablaufenden System zu tun.

2.2 Einordnung

2.2.1 Softwarestruktur versus Ablaufstruktur

Damit sind die zwei grundlegenden Problembereiche genannt:

- Das Betriebssystem schafft als ablaufendes System eine abstrakte Ausführungsmaschine (Infrastruktur) für Anwendungen und bestimmt deren Eigenschaften, und es unterliegt dabei selbst einer inneren ***Ablaufstruktur***.
- Das Betriebssystem besteht aus Software und ist damit Gegenstand von Softwareherstellungsprozessen. Die ***Softwarestruktur*** hat ursächlich nichts mit der Ablaufstruktur der Ausführungsmaschine zu tun.

Aus diesem Unterschied leiteten sich bereits früh die zwei grundlegenden Herangehensweisen für die Konstruktion und Strukturierung von Betriebssystemen ab, entweder ***Elemente des real ablaufenden Systems*** in den Vordergrund zu stellen oder softwaretechnologische Elemente, Methoden und Strukturmerkmale bei der ***Herstellung der Programme*** für Betriebssysteme. Man findet beides bei Dijkstra und Habermann Anfang der siebziger Jahre.

Dijkstra's *THE*-System wies zur Laufzeit eine *hierarchische Prozeßstruktur* auf [Dij68b], S. 341:

> " A multiprogramming system is described in which all activities are divided over a number of sequential processes. These sequential processes are placed at various hierarchical levels, in each of which one or more independent abstractions have been implemented.".

Die dominierenden Struktureinheiten waren Schichten von Prozessen im ablaufenden System [Dij71]. Mit jeder Schicht wurde dem System eine neue abstrakte Maschine hinzugefügt, deren Verarbeitung intern durch einen Prozeß ausgeführt wurde.
Einige Jahre später wurde der Einfluß von Fortschritten in der Softwaretechnologie in Richtung Modularisierung in prozeduralen Sprachen auch für Betriebssysteme erkennbar. Habermann's *funktionale Hierarchie* betrachtete Modularisierung auf programmiersprachlicher Ebene. Abgrenzbare Einheiten waren Prozeduren (Funktionen) [Hab76], S. 267:

> " ... the hierarchical structuring is based upon functions – not processes as employed in the "THE" system. ... It is the system *design* which is hierarchical, not its implementation.".

Die Schichtung des Systems wird auf die Zuordnung einer Menge von Funktionen zu Mengen von Modulen und Schichten zurückgeführt, wobei sich die Schichtung der Funktionen aufgrund wechselseitiger Bezugnahmen nicht notwendigerweise in der Zuordnung zu Modulen widerspiegeln muß. Ablaufstruktur und Prozesse spielen eine untergeordnete Rolle: *"Roughly speaking, a process can be thought of as a flow of control which passes among various modules, ..."* [Hab76], S. 271.

Diese beiden grundlegend verschiedenen Herangehensweisen bei der Konstruktion von Betriebssystemen, entweder die Ablaufstruktur oder die Softwarestruktur in den Vordergrund zu stellen, sind bis heute gültig, wenngleich eine andere Begriffswelt verwendet wird. Serverstrukturierte oder serverbasierte Systeme ($\rightarrow$ *μ-Kerne*) können zur ersten Kategorie gezählt werden, die meisten objektorientierten Betriebssysteme zur zweiten (*Choices*, *ETHOS*).

Bei traditionellen Systemen werden beide Gegenstandsbereiche auch begrifflich klar getrennt: Programm (Herstellung) $\leftrightarrow$ Prozeß (Ablauf). Mit *Objektorientierung* ist es zwar möglich, eine homogene Methode über alle Phasen des Softwareherstellungsprozesses zu spannen, von der Analyse [Rumb91] über den Entwurf [Coad91] bis hin zur Programmierung [Mey88], die Welt des Ablaufs eines Systems ist jedoch eine völlig andere. Die naheliegende Fortsetzung dieser Kette auch für den Ablauf wurde aber bislang allenfalls begrifflich vollzogen, indem Elemente ablaufender Systeme ebenfalls als Objekte bezeichnet wurden: Prozesse, Adreßräume, Maschinen u.a. Oft sind zugehörige Verwaltungsdatenstrukturen gemeint. Diese für Betriebssysteme aber wesentlichen Elemente passen in die "saubere" objektorientierte Betrachtungswelt nicht recht hinein, wie es in dem klassischen Zitat von H.H. Ingalls deutlich zum Ausdruck kommt: *"An operating system is a collection of things that doesn't fit into a language. There shouldn't be one."* [Ing81], S. 286.

Andererseits hat "Objektorientierung" für Betriebssysteme auch eine lange Tradition, wenn auch nicht unmittelbar im Sinne objektorientierter Methoden, sondern im Sinne der Homogenisierung vielgestaltiger Ressourcen in Betriebssystemen. *Hydra* (1974) ist ein frühes Beispiel:

> " This philosophy is realized through the introduction of a generalized notion of "resource", both physical and virtual, called "object". Mechanisms are presented for dealing with objects, including the creation of new types, specification of new operations applicable to a given type, sharing, and protection of any reference to a given object against improper application of any of the operations defined with respect to that type of object. The mechanisms provide a coherent basis for extension of the system in two directions: the introduction of new facilities, and the creation of highly secure systems. " [WCC+74], S. 337.

Die begriffliche Gleichsetzung beider Prinzipien, objektorientierte Methoden zur Softwareherstellung und "Objekte" als normiertes Handhabungsmittel für verschiedene Elemente in Betriebssystemen (Prozesse, Geräte, Dateien etc.), hat fatale Folgen für die Klarheit und Überschaubarkeit, zumindest solange der Unterschied zwischen Software- und Ablaufstruktur besteht[8]. Aus diesem Grund wird hier der Begriff "Objekt" vermieden, wenn es um Verarbeitungsinstanzen geht. Der Bezug zu Objekten im Verständnis objektorientierter Programmiersprachen wird auf eine explizite *Abbildung* von Objekten in Verarbeitungsinstanzen zurückgeführt, ohne die Abgrenzung zu verwischen. *CORBA* ist einer der wenigen objektorientierten Ansätze, bei dem dieser signifikante Unterschied ebenfalls anerkannt wird. Es wird ein ähnliches Vorgehen unter dem Begriff *activation policy* praktiziert. Objekte können in *CORBA* auf vier Arten in Server-Prozesse abgebildet werden ($\rightarrow$ *object implementation*) [OMG91].

Das Problem der Nichtadäquatheit programmiersprachlicher Beschreibungen und der Realität ablaufender Systeme existiert aber nicht erst, seit Objektorientierung aktuell wurde. In der Vergangenheit zeigte sich das insbesondere im Fehlen von Ausdrucksmitteln sequentieller Programmiersprachen für parallele Prozesse, die für Betriebssysteme unabdingbar sind. Es wurden

[8] Sind *Unix*-Prozesse, *Mach-Tasks*, *BirliX-Teams* "Objekte"? (sicher nicht im Sinne von Methoden der OOP)

deshalb schon frühzeitig Arbeiten geleistet, gebräuchliche Programmiersprachen für Betriebssysteme geeignet anzupassen, indem "parallele Dialekte" oder neue Systemsprachen geschaffen wurden. Diese enthielten mindestens Beschreibungsmittel für parallele Prozesse und für die Koordination kritischer Abschnitte. Beispiele sind: Hoare's *Monitore* [Hoa74] und *CSP* [Hoa78], *Concurrent Pascal* [Han72, Han75] oder *Mesa* [GMS77] als von *Pascal* abgeleiteter Dialekt für die Systemprogrammierung des *Xerox-Alto*. Eine andere Entwicklung setzte auf die formale Beschreibung und Verifizierung nichtdeterministischer Prozeßsysteme, wie Dijkstra's *Guarded Commands* [Dij75], was sich aber als nicht praktikabel erwies.

In der Praxis setzten sich für Betriebssysteme andere, aus den genannten Gründen eigentlich unzureichende Sprachen durch, wie *C* und *C++* für die *Unix*-Welt. Es werden verschiedene Techniken angewandt, die Unzulänglichkeiten dieser Sprachen für die Beschreibung von Betriebssystemen zu überwinden (→ Bibliotheksfunktionen mit "Seiteneffekten" für Prozesse, für das Überwinden von Schutzdomänen u.a.).

2.2.2 Wirkungen von Anpassung auf das Betriebssystem

Anpassung kann sich auf ein ablaufendes Betriebssystem beziehen, aber auch auf die Software zu seiner Herstellung. Für beide Kategorien ergeben sich Wirkungen für übergeordnete Anwendungen in zwei Richtungen.

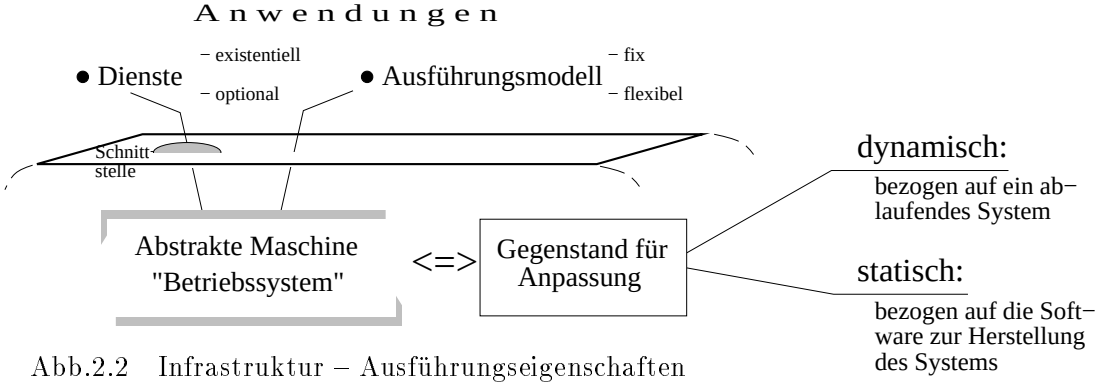


Abb.2.2 Infrastruktur – Ausführungseigenschaften

Das Betriebssystem bestimmt als Infrastruktur die *Ausführungseigenschaften* für Anwendungen:

- Das sind zum einen ***Dienste***, die Anwendungen bekannt sind und *explizit* in Anspruch genommen werden:
  - *Existentielle Dienste* des Betriebssystems müssen für den Ablauf von Anwendungen notwendig in der Infrastruktur vorhanden sein. Diese Dienste schaffen die Existenzgrundlage für übergeordnete Instanzen:
    - Erzeugung (Herstellung) und Beseitigung von Instanzen,
    - Mittel zur Ablaufsteuerung von Instanzen und
    - Mittel zur Interaktion von Instanzen.
  - *Optionale Dienste* können prinzipiell auch im Anwendungsbereich angesiedelt sein, sind aber in der Infrastruktur besser plaziert (z.B. Effizienz, Hardwarezugriff).
- Das Betriebssystem bestimmt das generelle ***Ausführungsmodell*** für Anwendungen:
  - Dieses ist in weiten Bereichen *fixiert* und nicht änderbar, z.B. die Eigenschaft getrennter Adreßräume.
  - Es kann jedoch auch die Option bestehen, diese Eigenschaften in Grenzen *flexibel* zu gestalten. Ein Beispiel wäre die Änderung einer Scheduling-Strategie.

2.3 Anpassung des ablaufenden Systems

Dynamische Anpassung ist ein Prozeß und bedarf einer Steuerung. Man kann drei grundlegende Varianten dynamischer Anpassung in einem System nach der Stellung der Steuerung unterscheiden, die einen Anpassungsprozeß veranlaßt (Abbildung 2.3):

- *dynamisch adaptierbar:* auf äußere Veranlassung (veranlassende Steuerung $\notin$ System),
- *dynamisch adaptiv:* Anpassungssteuerung durch einen inneren Regelkreis und
- *dynamisch reflexiv:* Regelkreis mit Rückwirkung auf die Anpassungssteuerung.

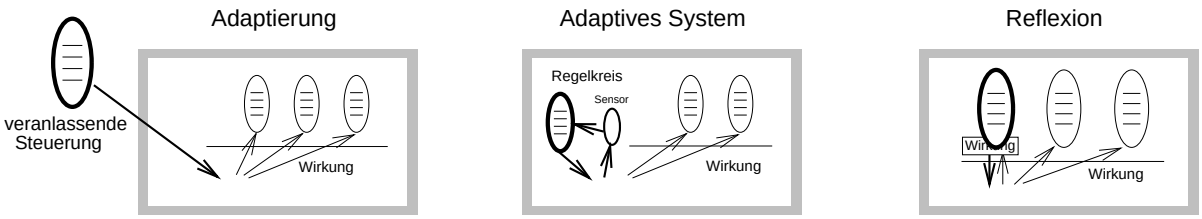


Abb.2.3 Varianten für dynamische Anpassung

2.3.1 Dynamische Adaptierbarkeit

Dynamische Adaptierbarkeit ist letztlich die Voraussetzung zur Schaffung adaptiver und reflexiver (Betriebs-) Systeme. Auf äußere Veranlassung (= Anpassungssteuerung) werden Veränderungen im System vorgenommen, die sich auf das Verhalten seiner Komponenten auswirken. In Abschnitt 2.3.4 werden drei Techniken dafür erklärt: Parametrisierung, (Re-) Konfigurierung und Meta(objekt)-Protokolle.

2.3.2 Dynamische Adaptivität

Bei adaptiven Systemen wird Anpassung durch das System selbst gesteuert, d.h. veranlaßt und ausgeführt. Ein adaptives System wird aus einer sensorischen Komponente gebildet, die Betriebswerte des Systems beobachtet und einer Entscheidungskomponente zuführt, die anhand der beobachteten Werte und eingebauter Regeln den Betriebsablauf des Systems beeinflußt.
Die Frage nach Adaptivität in Betriebssystemen wurde ebenfalls früh gestellt [BR76]. In der Dissertation *"Adaptive Betriebssystemkonzepte"* [Heck91] wird sie ebenfalls behandelt, und es werden Wirkungen adaptiver Ressourcenverwaltungsstrategien an einem Simulationssystem gezeigt. Es gibt auch Beispiele für adaptive Techniken in realen Betriebssystemen, wie das "faire" Prozeß-Scheduling in *Unix* [Bac86], adaptive Lastausgleichsverfahren [ELZ86, BK88] und Auswahlstrategien [GS89], adaptive Routingprotokolle [NRTT89] oder Benutzerschnittstellen. Dennoch sind die Mittel und Möglichkeiten für adaptive Techniken in Betriebssystemen heute wenig entwickelt. Das liegt auch daran, daß dynamische Adaptierbarkeit von Betriebssystemherstellern aus verständlichen Gründen nur eingeschränkt unterstützt wird (→ *Schutz, Kern-Integrität*).

Typischer ist dagegen die "off-line"–Adaption, bei der ein System beobachtet wird und aus den gesammelten Daten Konfigurierungshinweise abgeleitet werden. Der Konfigurierungsvorgang selbst kann nicht dynamisch ausgeführt werden. Diese Methode ist heute die allgemeine Praxis zur Systemoptimierung. Für *Unix* wird dieses Gebiet in [Lou91, Coc95] behandelt.
In [Heck91] werden Systeme genannt, die zum Teil auf der Basis von Expertensystemen beobachtete Daten in Konfigurierungshinweise umsetzen. Diese Art der "off-line"–Adaption gehört aber eher zur Kategorie der Herstellung optimierter Software für Betriebssysteme.

2.3.3 Dynamische Reflexion

Bei *Reflexion* wird eine Veränderung im System mit der Folge veranlaßt, daß die Veränderung auf die veranlassende Komponente zurückwirkt und damit auf sie reflektiert wird.

> " Computational reflection is the activity performed by a system when doing computation about (and by that possibly affecting) its own computation." [Mae86], S. 21.

Maßgebende Arbeiten dazu sind die Dissertationen *"Reflection and semantics in a procedural language"* [Smi82] und *"Computational Reflection"* [Mae87], die Untersuchungen zu *ABCL/R* an der Universität Tokio [WY88, Yon90] und die Arbeiten zu Reflexion in *Smalltalk-80* [FJ89] und *Meta-Objekt-Protokollen (MOP)* für *CLOS* am *Xerox-PARC* [Kicz91].
Reflexive Techniken wurden ursprünglich für interpretierte funktionale Sprachen untersucht, indem Zugangsmöglichkeiten eröffnet wurden, von interpretativ gesteuerten Prozessen einen definierten Einfluß auf den Interpreter und damit auf die Interpretation des eigenen Programms zu nehmen. Auch für andere Sprachen gibt es reflexive Erweiterungen, u.a. auch für *C++* [CIKM92].

> " The benefits of computational reflection are the abilities to reason and alter the dynamic behavior of computation from within the language framework." [MWY91], S. 231.

Diese Techniken finden primär auf dem Gebiet der Künstlichen Intelligenz Anwendung, um intelligentes ("lernfähiges") Verhalten zu erreichen, indem sich ein System nicht allein nach festen Regeln verhält, sondern diese auf Einwirkung von außen auch selbständig ändern kann und das als eine Änderung seines Verhaltens wieder nach außen reflektiert wird.

In gewisser Weise sind Analogien zu selbstmodifizierendem Code zu erkennen, d.h., ein Prozeß führt selbst Änderungen an seiner Steuerungsinformation aus. Bei Reflexion werden jedoch ***Ebenen*** unterschieden (→ ***Meta-Level Architektur*** [Mae86]), mindestens ein *Proto-Niveau* und ein zugeordnetes *Meta-Niveau*. Elemente des Meta-Niveaus repräsentieren und implementieren die Elemente des Proto-Niveaus. In objektorientierten Systemen sind das typischerweise Objekte auf Proto- (Objekte o_p) bzw. Meta-Niveau (Meta-Objekte $\uparrow o_m$)[9]. Zwischen Objekten o_p und Meta-Objekten $\uparrow o_m$ besteht eine *Kausal-Relation* derart, daß explizite Änderungen in $\uparrow o_m$ veränderte Zustände bzw. verändertes Verhalten für Objekte o_p bewirken. ***Reflexion*** findet statt, wenn Objekte o_p selbst Änderungen in Meta-Objekten $\uparrow o_m$ veranlassen, zu denen für sie eine Kausal-Relation besteht. Das dafür notwendige Protokoll zwischen Objekten und Meta-Objekten ist das *Meta-Objekt-Protokoll (MOP)* [Kicz91]. Eine Übersicht über den Stand und die Fortschritte auf diesem Gebiet wurde auf dem Workshop *"Advances in Metaobject Protocols and Reflection"* [Zim96] zur *ECOOP'95*, Aarhus, Dänemark gegeben.
Natürlich läßt sich die Zweistufigkeit auf n-Ebenen erweitern, indem es zu Meta-Objekten $\uparrow o_m$ Meta-Meta-Objekte $\uparrow\uparrow o_m$ gibt, was zu der potentiell "unendlichen Säule" (infinite tower) oder ***"Rekursion der Meta-Level Architektur"*** führt. Der Begriff Rekursion charakterisiert den Sachverhalt besser, weil Rekursion Endlichkeit und damit den notwendigen Abbau der Rekursion über mehrere Ebenen einschließt, bis hin zu einem elementaren Niveau, an dem die Rekursion letztendlich abbricht.

Die Grundidee von Reflexion erscheint auch für Betriebssysteme attraktiv. Die Analogie ist offensichtlich. Auch hier gibt es Ebenen (→ *Schichten*), bei denen untergeordnete Schichten die Elemente übergeordneter Schichten implementieren und deren Abläufe steuern.
Apertos ist derzeit das einzige Betriebssystem, das konsequent nach der Methode der Meta-Level Architektur strukturiert ist [Yok92] (vorher *Muse* [YTM91]), wenngleich viele Fragen der Implementierung und der Wirkung noch zu untersuchen sind.

[9] Der Operator $\uparrow$ drückt in [WY88] die Meta-Relation aus.

In der Dissertation *"Adaptierbarkeit durch Reflexion"* [Son93] werden reflexive Eigenschaften auch für das Betriebssystem *BirliX* [Här92] untersucht und realisiert, indem die Zustellung von Nachrichten zwischen *BirliX-Instanzen* (*Teams*) als grundlegender Betriebssystem-Mechanismus für Teams selbst manipulierbar wird.
Trotz dieser Arbeiten muß eingeschätzt werden, daß Fragen der praktischen Relevanz und der Umsetzung reflexiver Techniken für Betriebssysteme noch weitgehend offen sind.

2.3.4 Techniken für dynamische Anpassung

Es lassen sich prinzipiell drei Techniken für dynamische Anpassung (Adaptierung, Adaptivität, Reflexion) identifizieren:
- dynamische Parametrisierung,
- dynamische (Re-) Konfigurierung und
- Meta(objekt)-Protokolle in Meta-Level-Architekturen.

• Dynamische Parametrisierung

Die einfachste Form dynamischer Anpassung basiert auf der *dynamischen Parametrisierung* des Systems. Über Parameter (= Daten) können innere Zustände und damit die Steuerung des Systems beeinflußt werden. Anpassung dieser Art ist auf Manipulation der Zustände beschränkt. Die Steuerungsinformation (= Code) kann nicht beeinflußt werden, so daß sich die Wirkung von Zustandsänderungen auf die Auswahl a priori vorgegebener Steuerungsfolgen reduziert.

• Dynamische (Re-) Konfigurierung

Bei *dynamischer (Re-) Konfigurierung* wird diese Bandbreite wesentlich erweitert, indem ganze Komponenten (= Daten + Code) dynamisch in das System eingebracht bzw. entfernt werden können. Es erschien attraktiv und naheliegend, den Übergang in der Softwaretechnik zur Abgrenzung funktional und semantisch abgeschlossener Module [Par72] und deren einfache Austauschbarkeit auf Quelltextniveau auch für ablaufende Systeme nachzuvollziehen [Fab76]. Eine frühe Arbeit mit diesem Ziel war *DAS* (*Dynamic Alterable System*) an der TU Berlin [Löhr78].
Das dynamische Einbinden neuen Codes ist für Betriebssysteme nach wie vor aktuell [IM93, Eul96, PB96], gegenwärtig unter dem Blickwinkel der Sicherung der Integrität des Kerns.
Für Betriebssysteme bestand anfangs das Ziel darin, daß nur die Elemente Ressourcen belegen sollten, die tatsächlich benötigt wurden. Später kam hinzu, langfristig ablaufende Systeme auf nicht absehbare Änderungen einstellen zu können. Dynamische (Re-) Konfigurierbarkeit ist daher eine wichtige Eigenschaft in vernetzten Umgebungen [Mü92c, Paps93].

• Meta-Protokolle (MP) – Meta-Objekt-Protokolle (MOP)

Meta-Protokolle stehen in Verbindung zu Meta-Level-Architekturen. Meta-Level-Architekturen sind nicht notwendigerweise an Objektorientierung gebunden, so daß Meta-Objekt-Protokolle (MOP) als spezielle Kategorie von Meta-Protokollen (MP) für objektorientierte Meta-Level-Architekturen aufgefaßt werden können. Hinter beiden steht dasselbe Grundprinzip. Explizit benannt wurde diese Technik erstmals im Umfeld funktionaler interpretierter Sprachen und wurde am *Xerox-PARC* für *CLOS* untersucht.

> " Metaobject protocols are interfaces to the language that give users the ability to incrementally modify the language's behavior and implementation, as well as the ability to write programs within the language." [Kicz91], S. 1.

Generell gesagt, über ***Meta-Protokolle (MP)*** veranlassen Instanzen eines Proto-Niveaus Änderungen in Instanzen des Meta-Niveaus, zu denen kausale Beziehungen bestehen, so daß diese Manipulationen auf die veranlassenden Instanzen zurückwirken. Werden in einem System Objekte als Instanzen betrachtet, gilt dies für entsprechende ***Meta-Objekt-Protokolle (MOP)*** zwischen Objekten des Proto-Niveaus und Meta-Objekten.
Auf Betriebssysteme übertragen heißt das nichts anderes, als daß Instanzen eines Anwendungsbereichs Veränderungen in ihren eigenen Repräsentationen in der Infrastruktur vornehmen können. Man findet solche Techniken partiell auch in traditionellen Betriebssystemen, wie z.B. die Veränderung seines *nice*-Wertes durch einen *Unix*-Prozeß. Über M(O)P wird Infrastruktur "programmierbar". Als generelle Methode erscheint dieses Konzept aber sehr komplex. Man beginnt gerade damit, die Umsetzbarkeit für Betriebssysteme breiter zu untersuchen [CG95].

In *Apertos* [Yok93] werden Objekte von Meta-Objekten unterschieden, die eine Ausführungs-Infrastruktur (= Meta-Räume) für Objekte herstellen. Diese Unterscheidung setzt sich rekursiv fort und endet bei Hardware–Meta^n-Objekten. Dadurch werden Ebenen in das System eingeführt und Objekte ihrer Art nach durch Relation zu einem Meta-Raum differenziert. Für dynamische Anpaßbarkeit gibt es über Meta-Objekt-Protokolle zwischen Objekten und sogenannten Reflektoren in Meta-Objekten zwei Varianten, weitreichende Wirkungen zu erzielen. Es gibt einen generalisierten Migrationsmechanismus von Objekten zwischen Meta-Räumen, der auch den Wechsel zu neuen Ausführungseigenschaften bedeutet. Die zweite Variante besteht darin, Eigenschaften von Meta-Räumen durch Konfigurierung der zugehörigen Meta-Objekte zu manipulieren. Beide Varianten sind ihrer Konzeption nach sehr mächtig. Welche Möglichkeiten ihrer praktikablen Umsetzung tatsächlich bestehen, bleibt aktueller Untersuchungsgegenstand.

2.4 Anpassung aus softwaretechnologischer Sicht

An dieser Stelle soll nicht das umfangreiche Gebiet der Softwaretechnik behandelt werden, das natürlich auch für die Herstellung von Software für Betriebssysteme relevant ist. Der Zweck der Diskussion liegt in der Gegenüberstellung und Abgrenzung der beiden Betrachtungswelten: *ablaufendes System* ↔ *Herstellung der Software* für das System.

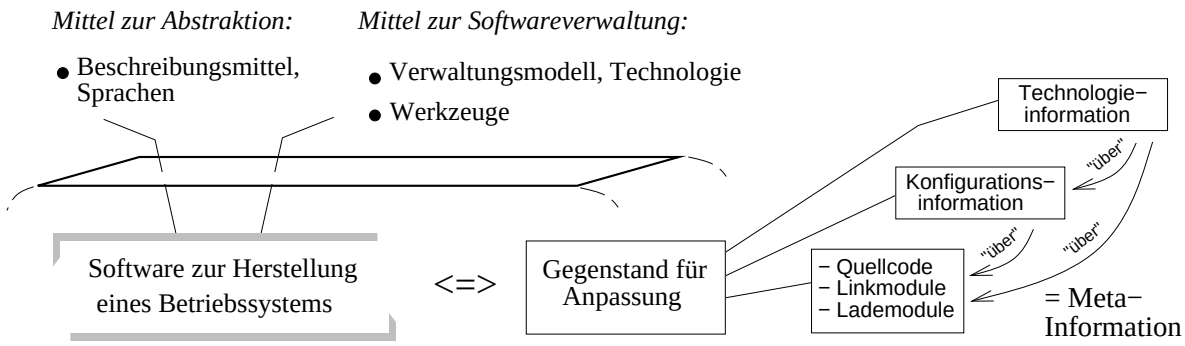


Abb.2.4 Softwaretechnologische Aspekte für Betriebssysteme

Neben den Phasen der Softwareentwicklung (Analyse, Entwurf, Programmierung) kann man drei technologische Ebenen unterscheiden, die ebenfalls Gegenstand für Anpassungen sind:

- allgemeine Technologie- und Verfahrensinformationen zur Softwareherstellung,
- Konfigurierungsinformationen für ein konkretes herzustellendes Zielsystem,
- elementares Niveau von Quellcode-, Link- und Lademodulen.

Dabei enthält jede Stufe ("Meta-") Informationen über die jeweils darunterliegende. Diese Art "Meta-Ebenen" sind natürlich völlig anderer Natur als die im letzten Abschnitt vorgestellten.
Für die Programmierung spielt die Implementierungssprache eine wichtige Rolle. Sie gibt die Mittel für Abstraktion, Modularisierung und Struktur für das Programmieren *"im Kleinen"* vor. Andererseits muß es ein Verwaltungsmodell und eine Technologie für das Zusammenfügen einzelner Programmteile geben (Programmieren *"im Großen"* oder *"packaging"* [Cox86]). Werkzeuge sollten diesen Vorgang unterstützen.
Oft werden Programmierungsaspekte in den Vordergrund gestellt. Die Steuerung des technologischen Herstellungsprozesses, die Pflege und Verwaltung der Software wird weniger beachtet, obwohl sie für umfangreiche Systeme mindestens den gleichen Stellenwert besitzt. Es geht hier um Versionsverwaltung und Variantengenerierung für Betriebssystemsoftware.
Dabei können ebenso moderne, objektorientierte Techniken angewandt werden, wie es vom Autor für das *Unix*-Werkzeug *make* bei der Verwaltung und Variantengenerierung der Software für die Betriebssysteme *BirliX* und *CHEOPS* in einer Verzeichnishierarchie gezeigt wurde [Grau93a].

2.4.1 Das Konzept der Betriebssystem-Familien

Man kann sagen, daß im Verlauf der Zeit fast alle allgemeinen Technologien und Paradigmen zur Softwareherstellung auch für die Herstellung von Betriebssystemsoftware angewandt wurden, mindestens in der Forschung untersucht wurden. Auf die besonderen Probleme kommerzieller Betriebssystemsoftware wurde bereits hingewiesen (→ *Langlebigkeit*).

Aus dem breiten Spektrum soll das *Konzept der Betriebssystem-Familien* herausgestellt werden, weil es bis heute gültige und wünschenswerte Eigenschaften von Betriebssystemsoftware treffend charakterisiert. Das Grundanliegen dieses Ansatzes kommt unter dem jeweiligen Stand der Softwaretechnologie in verschiedenen Formen immer wieder auf und ist bis heute aktuell.

Im Anschluß werden vier Techniken vorgestellt, die heute auch für die Herstellung von Betriebssystemsoftware in der Diskussion sind:

- Klassen-Frameworks,
- Generische Betriebssysteme,
- Offene Implementationen (*Open Implementations – OI*) und
- Entwurfsmuster (*Design Patterns*).

Betriebssystem-Familien gehen auf Habermann zurück [Hab76]. Die Idee bestand darin, Betriebssysteme bei der Herstellung aus mehrfach verwendbaren Software-Grundbausteinen zusammenzusetzen, ähnlich wie man es bei Hardware praktiziert. Das Ziel war, spezialisierte Betriebssysteme auf Basis einer gemeinsamen Bibliothek von Grundbausteinen einfacher herzustellen. Schon damals ging es um Variantengenerierung und das Finden wiederverwendbarer Grundbausteine. Dieses Ziel besteht unter den Möglichkeiten der jeweiligen Softwaretechnologie bis heute fort. Aus Kombinationen von Grundbausteinen zusammengesetzte Betriebssystemvarianten gehörten aufgrund ihrer gemeinsamen Herkunft zu einer "Familie".

> " A general purpose system cannot be as efficient in any of its roles as would be a system specifically designed for one particular purpose. Unfortunately, the development costs of even a unipurpose system usually precludes the construction of several independent such systems. ...
>
> Members of a system family are developed as far as possible along common lines to avoid as much redesign and recoding as possible. The software system family concept is somewhat analogous to the hardware concept ... " [Hab76], S. 266.

Das Familienkonzept ist auch eng mit Habermann's *funktionaler Hierarchie* verbunden, bei der Betriebssysteme als Schichten von Modulen und Funktionen (Grundbausteinen) betrachtet wurden. Variantengenerierung und Komposition bedeutete das schichtweise Zusammensetzen von Modulen und Funktionen (→ *"incremental machine design"* [Hab76], S. 266).

Zehn Jahre später lebte diese Idee mit dem Aufkommen von Objektorientierung wieder auf. Die *"Software-IC's"* von Cox hatten im Grunde dasselbe Ziel [Cox86], wenngleich sie allgemein auf Softwareherstellung bezogen waren und nicht speziell auf die Software für Betriebssysteme. Die Vision war es, und sie ist es nach wie vor, das Vorbild der Hardwaretechnik für Softwaresysteme zu übernehmen, eine überschaubare Menge funktional orthogonaler, möglichst standardisierter Grundbausteine zu schaffen und diese vielfach zu kombinieren und wiederzuverwenden. Wegen der einfachen Reproduzierbarkeit von Software sollten die Kosten deutlich reduziert werden.

In der Softwaretechnologie stellen ***Bibliotheken*** das Pendant zu Chipsätzen oder Baureihen der Hardwaretechnik dar. Bibliotheken werden seit Jahrzehnten angewandt. Ihre Ausprägungen folgen der technologischen Entwicklung (Macro-, Funktions-, Klassenbibliotheken).
Dennoch wurde und wird die Situation als unbefriedigend empfunden, weil das Grundproblem nicht bei den jeweiligen Mitteln zur Schaffung von Bibliotheken liegt, sondern im Identifizieren vielfach benötigter Komponenten, die funktional redundanzfrei und überschaubar sind [Par72]. Hardware ist aufwendiger zu fertigen und die Zahl der Hersteller und angewandter Grundtechnologien ist im Vergleich zu Software klein. Bei Hardwarebausteinen besteht für Anwender praktisch keine Alternative der Eigenentwicklung. Für Software ist diese in viel höherem Maße gegeben, so daß Wiederverwendung nur eine Option ist, die es zudem nicht ohne Aufwand gibt. Die Diskussion führt letztlich auf generelle softwaretechnologische Fragen zu Wiederverwendung [End88]. Das primäre Ziel besteht in der wirtschaftlichen Herstellung von Software, und Wiederverwendung *kann* ein Mittel dafür sein. Weithin akzeptierte Sätze von Bibliotheken entstehen oft nur für häufig benutzte Grundalgorithmen und für die Anbindung der Infrastruktur (Betriebssystemdienste, Programmierschnittstelle – *API*).

Auch für die Herstellung von Betriebssystemen ist es seit langem ein Ziel, eine möglichst minimale Menge orthogonaler, aber funktional vollständiger Grundbausteine zu identifizieren und in Bibliotheken zusammenzufassen: *"Design and Specification of the Minimal Subset of an Operating System Family"* [PHW76] bzw. minimale Betriebssytemschnittstellen zu identifizieren [Grau92a].

Ein Betriebssystem, das sich stark an dem Konzept der Betriebssystem-Familien orientiert, ist PEACE [Schrö93, Schrö94]. Der Zielbereich für PEACE sind Betriebssysteme für die Knoten eines Parallelrechners (ursprünglich SUPRENUM, später MANNA), für die traditionelle μ-Kerne *"zu schwer"* waren und auch unangemessene Eigenschaften aufwiesen, wie preemptives Multitasking oder virtuellen Speicher [Schrö94]. Der Ansatz ist, die schon bei μ-Kernen abgerüstete Kern-Funktionalität für Prozeßverwaltung und IPC weiter abzurüsten, aber nicht in einem funktionalen Sinn, sondern auf das ***Ausführungsmodell*** bezogen, d.h. auf die Ausprägung von Prozessen (→ nicht notwendigerweise mit separaten Adreßräumen und preemptivem Multitasking) und IPC (→ Nachrichten mit minimalen Startup-Zeiten). Beides spielt für den Wirkungsgrad von Parallelrechnern eine entscheidende Rolle. Das Ausführungsmodell soll für beide (minimalen) Basismechanismen *"featherweight"* sein. Aufbauend auf einem Kern (→ *minimal subset of system functions*) werden Systemerweiterungen (→ *system extensions*) für jeweils unterschiedliche Funktionen und Ausprägungen von Funktionen aufgesetzt, aus denen das Betriebssystem zusammengefügt wird (→ *system composition*).
Dieses Prinzip bezieht sich auf die Generierung verschiedener Betriebssystemvarianten auf der Ebene von Quellcode. Die Analogie zur Idee der Betriebssystem-Familien ist offensichtlich und bestätigt die These der fortwährenden Aktualität dieses Konzepts.
Eine Ausdehnung auch auf das ablaufende System wurde für PEACE in [Schm95] untersucht.

2.4.2 Aktuelle Techniken zur Herstellung von Betriebssystemsoftware

Für die Umsetzung der Idee von Betriebssystem-Familien bietet sich die objektorientierte Programmierung geradezu an, wie es für PEACE in einem Artikel ausdrücklich deutlich gemacht wird: *"Object-Oriented Operating System Design and the Revival of Program Families"* [Schrö91] .

• Klassen-Frameworks

Frameworks sind Bibliotheken, die als "Rahmen" für die Herstellung von Anwendungen gedacht sind. Ihr Zweck ist, bei der Anwendungsherstellung Elemente des Frameworks direkt wiederzuverwenden, zu ergänzen und zu spezialisieren. Diese Eigenschaft unterscheidet ein Framework von anderen (geschlossenen) Bibliotheken. Hinter Frameworks steckt also nicht nur eine Bibliothek, sondern auch eine Methode für die Spezialisierung bei der Herstellung von Anwendungen. Frameworks sind typischerweise mit objektorientierten Methoden realisiert. Der Vorteil ist, daß Gemeinsamkeiten von Elementen des Frameworks durch mehrstufige hierarchische Relationen zwischen Typen (Klassen) ausdrückbar sind. Universelle Elemente und Eigenschaften können zentral und damit in höherem Maße redundanz- und konfliktfrei beschrieben werden. Im Idealfall lassen sich Elemente unmittelbar wiederverwenden. Neue Elemente werden durch Ableiten von Unterklassen erzeugt (→ *programming by difference*). Frameworks müssen offen sein und eine innere Struktur aufweisen, die Anpassungen auch praktikabel unterstützt.

Objektorientierte Klassen-Frameworks sind eine nahezu ideale Technik zur Umsetzung des Konzepts der Betriebssystem-Familien. *Choices* war das erste (stand-alone) Betriebssystem, das vollständig als Klassen-Framework implementiert wurde [Camp87, Rus91, Camp93]. Eine typischer Ableitungspfad für die Klasse des Bausteins zur Adreßumsetzung ist [Rus91] :

```
class AddressTranslation { ... };   // abstract class

  class TwoLevelPageTable : public AddressTranslation { ... };

    class NS32332Translation : public TwoLevelPageTable {...};
    class i386Translation    : public TwoLevelPageTable {...};
    class MC68030Translation : public TwoLevelPageTable {...};
```

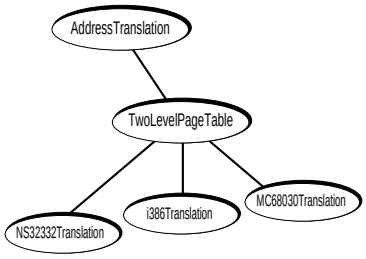


Abb.2.5 Klassen-Framework: Ableitungspfad für Adreßumsetzung

Nach dieser Methode kann der gesamte Quellcode eines Betriebssystems strukturiert werden. Die Herstellung eines konkreten Betriebssystems dieser Familie erfordert "lediglich" die Auswahl entsprechender Unterklassen zur Instanzierung entsprechender Objekte: *"Object-oriented programming is not so much a coding technique as it is a code packaging technique, ..."* [Cox86], S. 13. Diese Auswahl muß allerdings außerhalb der Sprache über die Konfigurierung gesteuert werden.

An der Professur Betriebssysteme wurden eigene Erfahrungen mit dem objektorientierten Entwerfen und Implementieren von Betriebssystemen gesammelt. In [Mü90a, Mü90b] und [MüKT90] wird gezeigt, wie die Schichten eines Betriebssystems in eine Klassenhierarchie abgebildet werden können und wie Vererbung zur Anpassung an die Hardware genutzt wird. Dieser Ansatz bildete die Grundlage für die Re-Implementierung des Betriebssystems *MINIX* [Tan90]. Das Betriebssystem *MINIX++* [Geb93, Wiat93, Kern95] wurde nach dieser Methode neu implementiert, ohne aber die Ablaufstruktur zu ändern. Die Erfahrungen aus diesen Arbeiten sind ein wichtiger Hintergrund für das *CHEOPS*-Projekt.

Für Betriebssysteme, die auf Klassen-Frameworks basieren, spielt die verwendete Implementierungssprache eine entscheidende Rolle. Für *Choices* ist es *C++*. Ein anderes Beispiel in dieser Kategorie ist *ETHOS* von der *ETH Zürich* [Szy92], das in *Oberon-2* [Wirth92] implementiert ist. Klassen-Frameworks sind dann sinnvoll, wenn die Zielsysteme in hohem Maße ähnlich sind.

• Generische Betriebssysteme

Der Name dieser Technik kennzeichnet es bereits, bei generischen Betriebssystemen geht es um die *"...anwendungsspezifische Generierung maßgeschneiderter Betriebssystem-Varianten"* [Neh95], S. 39. Der Zielbereich sind hier vor allem kleine, dedizierte Betriebssysteme für spezielle Anwendungen. Es wird die Technik der Klassen-Frameworks angewandt und um Laufzeitbibliotheken oder kleine (*pico-*) Kerne ergänzt [ABB+93b], die eine Anpassung des Sprachmodells an das Ablaufmodell der Hardwareinfrastruktur des Zielsystems vornehmen.
Die Aktualität dieses Ansatzes kann man für die Klasse dedizierter Betriebssysteme anhand aktueller Forschungsprojekte belegen, wie dem GENESYS-Projekt an der Universität Kaiserslautern [NS95] oder dem *Tigger*-Projekt am Trinity College Dublin [CHJ+94].

• Offene Implementationen (Open Implementations – OI)

Offene Implementationen (OI) kann man in gewisser Weise als Gegenstück zu Meta(objekt)-Protokollen auf der Seite der Softwaretechnologie ansehen. Es geht darum, das Prinzip der Kapselung und des Verbergens der Implementation von Modulen oder Objekten aufzuweichen, um den Grad an Wiederverwendbarkeit durch Anpassung auch der Implementation zu erhöhen. Für objektorientierte Techniken stellt es eine Alternative zur Unterklassenbildung dar, indem bei Spezialisierung nicht abgeleitet wird, sondern die Definition und Teile der Implementation einer bestehenden Klasse einfach geändert werden. Diese Technik verletzt eigentlich die Prinzipien von Kapselung und *information hiding*. Kiczales benennt aber das in der Praxis erkannte Problem treffend: *"Why are Black Boxes so Hard to Reuse? "* [Kic94] . Am *Xerox-PARC* konzentrieren sich gegenwärtig die Aktivitäten zu OI.

> " Software has traditionally been constructed according to the principle that a module should expose its functionality but hide its implementation. This principle, informally known as *black-box abstraction*, is a basic tenet of software design, underlying our approaches to portability, reuse, and many other important issues in computing.
> Although black-box abstraction has many attractive qualities, exposing only the functionality of a module can lead to serious performance difficulties when reusing it. These problems have led a new community of researchers to reconsider the question of what a module should expose to clients. ... They call this new design principle *open implementation*.
>
> ... the client often knows best how the module should be implemented. Black-box abstraction forces the implementor to decide early on what the implementation will be, and then locks that decision into the black box. ... Some of these issues are crucial implementation-strategy decisions that will inevitably bias the performance of the resulting implementation. Module implementations must somehow be opened up to allow clients control over these issues ... "
> `http://www.parc.xerox.com/spl/projects/oi/`, September 1996, oder [Kic96], S. 8–10.

Die Manipulierbarkeit von Implementationen beschränkt sich dabei nicht auf Quellcode. In *OMOS* (*Object/Meta-Object Server*) [OM92] wird Offenheit auch für übersetzte Module (.o-Files) erlaubt, indem diese auf dem *OMOS*-Server verwaltet werden und aus ihnen anhand einer Konfigurierungssprache Varianten für ein zu ladendes Programm generiert werden können. Das Grundkonzept basiert darauf, daß externe Referenzen für Funktionsaufrufe in den .o-Files beim Linken nicht direkt aufgelöst werden, sondern über *"wrapper-functions"* eingeschlossen werden, damit Funktionsaufrufe zur Laufzeit manipuliert werden können.

Offenheit der Implementation (OI) ist auch das Leitmotiv für *Open C++*, bei dem die Implementation des *C++*-Methodenaufrufs manipulierbar gestaltet wird: *"... a variant of C++ called Open C++, in which the programmer can alter the implementation of method calls to obtain new language functionalities suitable for the programmer's applications. "* [CM93], S. 482. Die Codegenerierung

erfolgt mit einem *Open C++*-Compiler anhand von Quellcode-Attributierung. Durch Indirektion kann dann während der Laufzeit in Abhängigkeit der Plazierung eines Objektes ausgewählt werden, ob für das Objekt ein lokaler oder ein entfernter Methodenaufruf ausgeführt wird. Verteilte Systeme sind auch der Zielbereich für diese Entwicklung. Der Fortschritt gegenüber dem klassischen *Proxy*-Prinzip [Sha86, Mü92b] besteht darin, daß *Proxy*-Beziehungen statischer Natur sind, so daß Aufrufe in jedem Fall, auch im lokalen, über den *Proxy* auf Nachrichten abgebildet werden. Erst dort finden ggf. Optimierungen statt. Der Aufwand für die Konvertierung auf bzw. von Nachrichten ist aber stets vorhanden und ist nicht unerheblich (z.B. für *DCE* [Rich94]).

Obwohl die Beeinflussung der Ablauf- oder "Meta"-Eigenschaften zur Laufzeit erfolgt, muß der Code für alle Varianten bereits bei der Herstellung eingebaut werden. Zur Laufzeit besteht dann lediglich die Option der Auswahl einer Variante für eine Meta-Eigenschaft. Deshalb werden Entwicklungen dieser Art hier in die Kategorie der Herstellung eingeordnet.

- **Entwurfsmuster (Design Patterns)**

In dem Buch *"Design Patterns"* [Gam94] werden aus Sicht der Autoren 23 immer wiederkehrende und damit verallgemeinerbare *"Entwurfsmuster"* zur Herstellung objektorientierter Software benannt und beschrieben. Zehn Jahre Erfahrung mit objektorientiertem Entwerfen und Programmieren haben gezeigt, daß es für einen guten Softwareentwurf und für die Implementierung nicht genügt, die bekannten objektorientierten Mittel auf beliebige Art und Weise einzusetzen, sondern daß es viel wichtiger ist abzuwägen, für welchen Zweck welche Mittel geeignet sind, welche Konsequenzen sich ergeben und wie man sie vorteilhaft anwendet[10]. Erst der geschickte Umgang mit diesen Mitteln macht die *"Kunst"* von Objektorientierung aus, die bislang nur als *"untold design experience"* existiert. Den Autoren geht es um das explizite Benennen von Stilelementen und deren Wirkungen. Dem Buch von *Coplien "Advanced C++ Programming Styles and Idioms"* [Cop92] liegt eine ähnliche Intention zugrunde. Entwurfsmuster sind allgemeiner Natur und können prinzipiell für jede objektorientierte Sprache angewandt werden [Gam93].
Für Betriebssysteme werden *Design Patterns* aktuell an der Universität Kaiserslautern für die Strukturierung der Software für *Unix*-Betriebssystemdienste untersucht [BMS96].

Man kann in gewisser Weise eine Analogie von *Design Patterns* zum Ziel in dieser Arbeit sehen, ein abstraktes Grundmuster als generelle Struktur über verschiedenartige Ebenen eines ablaufenden Systems zu legen und dieses in konkreten Ausprägungen anzuwenden, zu spezialisieren.

2.5 Ablaufstrukturen in Betriebssystemen – Kern-Architektur

Die mannigfaltigen Gründe für Anpassungsprozesse in Betriebssystemen haben eine Vielfalt von Lösungen hervorgebracht. Letztendlich ist die gesamte Entwicklung von Betriebssystemen ein Anpassungsprozeß an jeweilige Gegebenheiten und Erfordernisse. Einige von ihnen wurden in den vorangegangenen Abschnitten vorgestellt, und es wurden die zwei wesentlichen Kategorien herausgearbeitet, auf die sich Anpassung beziehen kann: auf die Herstellung von Software für ein Betriebssystem bzw. auf ein Betriebssystem als ablaufende Maschine.

Während sich in der Softwaretechnologie eine folgerichtige Entwicklung vollzogen hat[11]:

▷ Subroutinen, globale Daten → Prozeduren, Datenstrukturen → Module → abstrakte Datentypen → Typhierarchien (OOP),

[10] z.B. Fragen wie *Vererbung vs. Aggregation* oder *Polymorphismus* durch *Vererbung* oder *dynamisches Binden*
[11] Die Folgerichtigkeit der Entwicklung in der Softwaretechnologie wird anschaulich in [Marty88] dargestellt.

standen für Betriebssysteme die zwei grundlegenden Modelle für Ablaufstrukturen seit Anfang der siebziger Jahre fest, die es in verschiedenen Variationen bis heute gibt:

$$\triangleright \quad \frac{\text{prozeßorientiert}}{\text{monolithisch, prozedural}} \quad \rightarrow \quad \text{Zwei–Schichten–Ablaufstruktur}$$

Die Argumentation für und wider einer prozeßorientierten oder einer monolithischen Struktur wurde schon bei Dijkstra's Schichten von Prozessen [Dij68b] und Habermann's funktionaler Hierarchie [Hab76] geführt. Es besteht heute weitgehende Übereinstimmung darin, daß die Prozeßstruktur die für Betriebssysteme preferierte Variante ist. Sie hatte damals und hat auch auf heutigen Maschinen einen nicht zu vernachlässigenden Preis:

> " One of the arguments against the "THE" design is the ***overhead*** associated with interlevel communication among processes. In a functional hierarchy where functions may actually be macros, a sequence of function calls may result in a single machine instruction (or possibly non at all) when the system is compiled. It is the system ***design*** which is hierarchical, not its implementation. " [Hab76], S. 267.

Darin liegt bis heute der wesentliche Grund, weshalb innerhalb von Betriebssystemen monolithische Strukturen vorherrschen. Strukturelle Fragen werden auf die Ebene des Entwurfs und der Programmierung gehoben und führen zu dem genannten Widerspruch zwischen Software- und Ablaufstruktur. Nur im Anwendungsbereich haben sich unabhängige Prozesse aufgrund der Nutzungsanforderungen etabliert.
Der ***Betriebssystemkern-Ansatz*** manifestiert diese Struktur. Der Kern ist monolithisch implementiert, daher effizient auf heutigen Maschinen umsetzbar und stellt für den Anwendungsbereich Prozesse bereit. Diese Zweiteilung des Kern-Ansatzes bildet die grundlegende Ablaufstruktur in Betriebssystemen, von den Anfängen in den sechziger Jahren bis heute.

P.B. Hansen versuchte Anfang der siebziger Jahre, eine möglichst kleine Menge orthogonaler Grundfunktionen eines Kerns zu identifizieren und in einer monolithischen Basisschicht *Nucleus* zusammenzufassen (Speicher- und Prozeßverwaltung und Nachrichtenkommunikation):

> " ... to concentrate on the fundamental aspects of the control of an environment consisting of parallel, cooperating processes. ... The purpose of the system nucleus is to implement these fundamental concepts: simulation of processes; communication among processes; creation, control, and removal of processes. " [Han70], S. 238.

Dieser Ansatz bildete einige Jahre später die Grundlage für μ-Kerne. In dieser Grundstruktur bleibt für Anpaßbarkeit letztlich die Frage, welche Möglichkeiten es gibt, weitere Funktionalität in welcher Schicht und in welcher Ausprägung – als Prozedur oder Prozeß – anzuordnen.

Betriebssystemfunktionen sollen hier nicht mit der im Kern angesiedelten Funktionalität gleichgesetzt werden. Es sollen alle Funktionen zur Kategorie Betriebssystem gezählt werden, die zum Ablauf und zum Betreiben einer Anwendung notwendig sind, die aber nicht im unmittelbaren Herstellungsprozeß der Anwendung mit entstehen, sondern als Infrastruktur oder Umgebung für den späteren Ablauf der Anwendung vorausgesetzt werden. Dazu zählen insbesondere auch Laufzeitsysteme von Programmiersprachen, Betriebssystem-Aufsätze und auch Instanzen der Umgebung in derselben Schicht der Anwendung. Für *Unix* gibt es eine Vielzahl von "Systemprozessen", die für den Ablauf von Anwendungen unabdingbar sind, die aber aus Sicht des Kerns als reguläre Anwendungsprozesse ablaufen.

Für die Kern-Architektur ergeben sich vier Varianten, wo Komponenten eines Betriebssystems im genannten Sinn plaziert werden können:

- **(a)** als weitere Instanzen in der *Umgebung der Anwendung*,
- **(b)** als *funktionale Schicht* innerhalb einer Anwendungsinstanz (→ *Laufzeitsystem*),
- **(c)** als permanenter (statischer) *Bestandteil des Kerns* oder
- **(d)** als statische oder dynamische *Erweiterung des Kerns*.

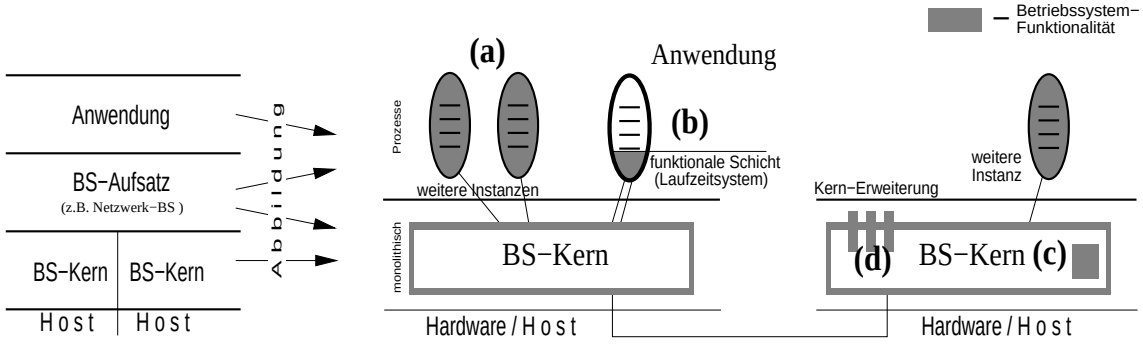


Abb.2.6 Plazierung von Betriebssystemfunktionen in der Kern-Architektur

In Abbildung 2.6 ist noch ein zweiter Aspekt dargestellt, der zeigt, wie konzeptionelle Schichten von Systemen auf die Realität der zwei Schichten des Kern-Ansatzes abgebildet werden:

→ *" It is the system **design** which is hierarchical, not its implementation. "* [Hab76], S. 267 .

In dieses Szenarium läßt sich eine Vielzahl von Systemen einordnen. Einem System werden konzeptionell neue Schichten (→ *abstrakte Maschinen*) hinzugefügt, indem sie auf eine Kombination der vier genannten Varianten in die Realität eines ablaufenden Systems mit Kern-Architektur abgebildet werden. Heute werden vor allem die Varianten (a) und (b) praktiziert, da Kern-Erweiterungen nur wenig praktikabel unterstützt werden. Die Frage nach Wahrung der Integrität des Kerns spielt dabei eine zentrale Rolle. Gegenwärtig stehen deshalb Untersuchungen zur anwendungsorientierter, dynamischer Erweiterbarkeit von Kernen wieder im Mittelpunkt des Interesses (vgl. Abschnitt 2.7). Man kann zwar vereinzelt Beispiele für anwendungsorientierte, dynamische Erweiterungen von Betriebssystemkernen in der Praxis finden, die nicht unmittelbar vom Betriebssystemhersteller initiiert wurden, z.B. die Erweiterungen des *Unix*-Kerns für *AFS* [SK89]. Der typische Fall ist das aber nicht.

Der Hauptgrund für die mangelnde Nutzung anwendungsorientierter Erweiterungen von Kernen liegt einmal in der oft fehlenden Offenheit der Kerne. Selbst dort, wo diese Offenheit prinzipiell gegeben ist, z.B. in vielen modernen *Unix*-Systemen durch die dynamische Nachladbarkeit von Modulen in den Kern, ergibt sich keine praktikable Lösung, weil kerninterne Strukturen für Dritte nicht offengelegt sind. Insofern wird klar, daß für anwendungsorientierte Erweiterungen vor allem die Varianten (a) und (b) praktiziert werden, für die es auch zahlreiche Beispiele gibt. Mit dem Aufkommen der Vernetzung wurden den lokalen Betriebssystemkernen Aufsätze hinzugefügt, um aus Sicht der Anwendung Dienste für entfernte oder verteilte Verarbeitung bereitzustellen. Auf diese Weise entstand eine konzeptionelle Schicht *"Netzwerkbetriebssystem"*, die über (a) und (b) auf die Zweiteilung der Kern-Architektur abgebildet wurde. Beispiele für derartige Systeme sind: *Newcastle Connection* als früher Vertreter für *Unix* [BMR82], *COMANDOS* [CBHR93] oder *DACNOS* [GH90], aber auch verteilte Programmiersysteme, wie *OSF/DCE* [OSF92], *OMG/CORBA* [OMG91], *MPI* [GLS94] oder *PVM* [Sund90] kann man hier einordnen.

Im Projekt *NOW* (*Network of Workstations*) [And95][12] an der University of Berkeley ist es das Ziel, möglichst alle Ressourcen eines Workstationnetzes zu einem übergreifenden *"Supercomputer"*

[12] siehe auch Bericht in [Grau95d]

zusammenzufassen. Für die Granularität von *Unix*-Prozessen gelingt das bereits, wie *MPI* oder *PVM* (*Parallel Virtual Machine*) zeigen (→ *Ressource CPU*). Für andere Ressourcen oder für feiner granulare Verteilungseinheiten ist das für Workstations nicht gelöst, weil man hier ohne Eingriffe in den Kern nicht auskommt.

Die Akzeptanz neuer Betriebssysteme hängt wesentlich von der weiteren Ablauffähigkeit bestehender Anwendungen ab. Bei der *Emulation von Betriebssystemen* wird die Funktion eines Betriebssystems im Sinne einer abstrakten Maschine auf Basis eines anderen Betriebssystems erbracht. In dieser Technik lag eine wesentliche Voraussetzungen für den erreichten Akzeptanzgrad der μ-Kerne, daß Anwendungen, vor allem aus dem Umfeld von *Unix*, ablauffähig waren. Bei gleichen Prozessoren kann sogar Binärkompatibilität erreicht werden. Für den *Mach3.0*-μ-Kern gibt es Emulationen für das ursprüngliche *4.3BSD-Unix* [DFGR90], den *OSF/1-Kern* [OSF93], *OS/2* [AP93] und auch für *MSDOS* [BGMR91]. Andere Beispiele sind *CHORUS/MiX* [GG92], bei dem ein *Unix System V* – Subsystem auf den *CHORUS-V3*-Kern [RAA+88] abgebildet wird.

Ein anderer Weg ist, die Hardware als *virtuelle Maschine* in einer Schicht herzustellen, auf der ein unverändertes Betriebssystem ablaufen kann. Diese Technik wurde in den siebziger Jahren eingeführt, um auf einer Maschine zeitgleich Anwendungen unter verschiedenen Betriebssystemen zu betrieben bzw. schnell zwischen Betriebssystemen zu wechseln. Beispiele sind *IBM VM/370* [SM79], *VM2000* für *BS2000* [NP95] oder die *"Software-PC's"* für *Unix*-Workstations oder für *NT*.

• Klassifikation von Betriebssystemkernen

Analysiert man Ausprägungen von Betriebssystemkernen, kann man fünf Kategorien angeben:

- schwere Kerne,
- monolithische μ-Kerne,
- μ-Kerne,
- Kerne mit *"In-Kernel" Servern*,
- nano- und pico-Kerne.

Der Unterschied zwischen ***schweren Kernen*** und μ-Kernen wird an der Funktionalität des Kerns festgemacht. Schwere Kerne enthalten alle typischen Betriebssystemelemente, wie Prozeßverwaltung, Dateisysteme, Gerätetreiber u.a. (→ *Unix*).
Bei *μ-Kernen* war die Idee, die Funktionalität des Kerns als Infrastruktur für den Anwendungsbereich auf ein Minimum zu beschränken [Han70]:

- Prozeßverwaltung (PROC),
- Speicherverwaltung für Prozesse (VMM) und
- Interprozeß-Kommunikation (IPC).

Damit waren die minimalen Betriebssystemfunktionen bestimmt: *Prozesse, IPC* und *Speicher*, allerdings wurde keine Aussage über das ***Ausführungsmodell*** und damit die Ausprägung dieser Grundfunktionen getroffen. Es wurde vielmehr das von *Unix* geprägte Modell übernommen, d.h., alle Instanzen haben separate Adreßräume, das Scheduling erfolgt zeitscheibengesteuert preemptiv, es gibt "unbegrenzten" virtuellen Speicher usw. Insofern sagt die reine Funktion eines Kerns noch nicht viel über seine Komplexität und Struktur, die Ausführungseigenschaften und den Ressourcenbedarf aus. Es verwundert daher nicht, daß der Umfang von μ-Kernen extrem schwankt, obwohl sie im Prinzip die gleichen Grundfunktionen enthalten, jedoch mit unterschiedlichen Ausführungsmodellen.
Man vergleiche den Umfang eines *Mach3.0*-μ-Kerns von 300-500 kByte [Zim93] mit den "kleinen" μ-Kernen *QNX* – 10 kByte [Hil92], *PXROS* – 27 kByte [Mau96] oder *L4* – ca. 30 kByte [Lie95].

Die verbleibende Funktionalität ist oberhalb des μ-Kerns angesiedelt. Bei μ-Kernen muß jedoch die Art der Umsetzung in einem realen System unterschieden werden. Der frühe *Mach*-Kern war als *"new foundation for Unix"* [ABB+86] für SMM-Multiprozessormaschinen konzipiert und als funktionale Schicht (→ ***monolithischer μ-Kern***) für ein aufgesetztes *4.3BSD-Unix* realisiert. *Unix*-Mechanismen wurden auf Basis der *Mach*-Primitiven im Kern implementiert. Es wurden auch kommerzielle Betriebssysteme auf dieser Basis aufgebaut, beispielsweise *NextStep, Release 3* [NeX92] auf Basis *Mach2.0* oder *OSF/1* [OSF93] auf Basis *Mach2.5*.
Erst *Mach3.0* [DFGR90] wird als ***"echter" μ-Kern*** angesehen, bei dem die *Unix*-Funktionalität vollständig aus dem Kern gelöst wurde. *Chorus* war von Anfang an als ein solcher μ-Kern konzipiert, im Unterschied zu *Mach* aber für lose gekoppelte MPM-Systeme [BCG+80] gedacht.

Neben der teuren Realisierung "echter" μ-Kerne, Betriebssystemfunktionen als Server-Prozesse im Anwendungsbereich auszuführen, besteht auch die Möglichkeit, diese Struktur effizienter wieder im Kern-Bereich herzustellen. Die Technik der ***"In-Kernel Server"*** unterscheidet sich von *"Out-Kernel Servern"* der μ-Kernen dadurch, daß Server-Prozesse innerhalb des Kerns im selben Kern-Adreßraum ausgeführt werden. Dadurch vereinfacht sich die Handhabung der Prozesse erheblich, was der Effizienz zugute kommt.
Als erste Ausprägung von *In-Kernel Servern* können die *Protected Subsystem Actors* von *Chorus* [RAA+88] angesehen werden, die im Gegensatz zu sonstigen *Chorus*-Aktoren im Kern-Adreßraum liegen. Andere Beispiele für Systeme mit *In-Kernel Servern* sind *UNICOS/mk* von *CRAY* [Bro95] oder *Mach-INKS* [Lepr93] für *Mach3.0*. Die Unterschiede zwischen Kern-Servern und Kern-Threads, die es auch in monolithischen Kernen gibt, sind:

- die Server-Prozesse weisen den typischen Zyklus auf (*main-message-dispatch-loop*),
- Server-Prozesse interagieren über Nachrichten und
- Server-Prozesse haben a priori keine gemeinsamen Zustände, obwohl Schutz im selben Kern-Adreßraum erst einmal nicht gegeben ist.

In-Kernel Server können als wegweisende Technologie für μ-Kerne angesehen werden (vgl. Abschnitt 2.7). Der große Vorzug ist, daß dasselbe Grundmodell aus dem Anwendungsbereich für einen Teil des Kern-Bereichs *mit anderen Ausführungseigenschaften* übernommen wird. Die Verallgemeinerung dieses Prinzips ist ein wichtiger Ansatzpunkt in dieser Arbeit.

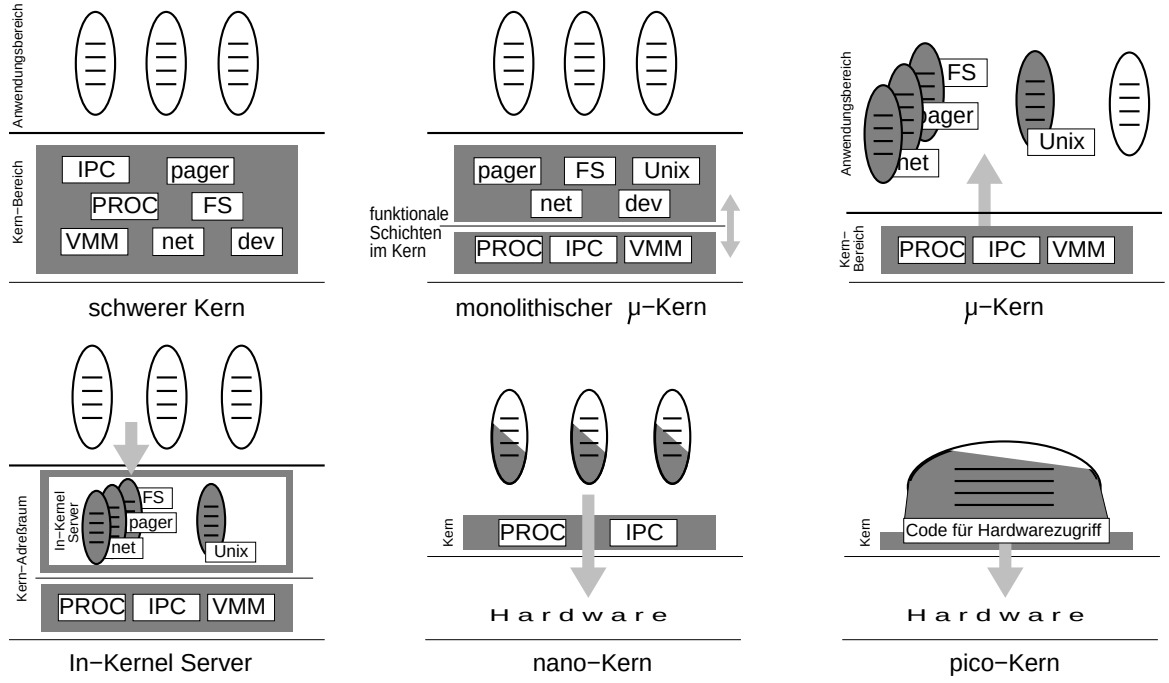


Abb.2.7 Ausprägungen von Betriebssystemkernen

Die zuletzt genannte Kategorie der ***nano- und pico-Kerne*** zielt nicht auf universelle Rechnensysteme, sondern auf spezielle Zielumgebungen für dedizierte Betriebssysteme. Besonders in ressourcenarmen Systemen geht es um extrem leichtgewichtige Ausführungsmodelle (→ *"featherweight"* [Schrö94]), wenn man es sich überhaupt leistet, Betriebssystemfunktionen von Anwendungen abzutrennen. Eingangs wurde zumindest der zukünftige Bedarf dafür begründet.
Die Begriffsbildung ist für diese Art von Systemen noch nicht in der Weise abgeklärt, wie es für μ-Kerne der Fall ist. Der Begriff Betriebssystem kann hier auch mit Ablauf- oder Laufzeitsystem umschrieben werden [Neh95, Schrö94]. Dennoch kann man eine Differenzierung angeben, daß nano-Kerne noch eine leichtgewichtige Prozeßabstraktion und IPC für Anwendungen bereitstellen, während es bei pico-Kernen lediglich noch höhere Zugriffsfunktionen zur unmittelbaren Hardware gibt. Kerne dieser Art umfassen nur einige kByte. Die Abgrenzung eines nano-Kerns von der Anwendung ist allenfalls noch im Sinne einer funktionalen Schicht möglich, während man für pico-Kerne aufgrund des anderen Ausführungsmodells – Prozesse statt Zugriffsprozeduren – diese Grenze noch ziehen kann. Nano-Kerne könnten somit auch in die Kategorie der Softwareherstellung eingeordnet werden.
Systeme, die für sich selbst die Bezeichnungen pico- bzw. nano-Kern verwenden, sind: PANDA [ABB+93a, Ass96], *KeyKOS* [Bom92] oder *μChoices* [CRT95].

• Anmerkung zur Effizienz

Zum real vorhandenen Effizienznachteil der prozeßorientierten Struktur ist zu sagen, daß dieser eng mit der heutigen Rechnerarchitektur zusammenhängt (→ *"Monoprozessor-Flaschenhals"*). Als Ursachen für signifikante Leistungsverluste können identifiziert werden:

- *ready*-Wartezeiten, wenn es mehr arbeitsbereite Prozesse als Prozessoren gibt,
- Aufwand für Werterettung der Prozeßzustände in der Hardware bei Umschaltungen[13],
- die Kommunikation zwischen Prozessen ist stets mit Prozeßumschaltungen verbunden,
- das Überwinden von Schutzdomänen zwischen Prozessen erfordert zusätzlichen Aufwand.

Es gibt eine breite Palette von Optimierungen, vor allem aus dem Umfeld von *Mach*[14], um Prozeßumschaltungen, Interprozeß-Kommunikation und den gesicherten Übergang zwischen Schutzdomänen für μ-Kerne zu beschleunigen. Einige Beispiele sind: *Continuations* [BDDR91], *Scheduler Activations* [BML93], *Lightweight RPC* zwischen Server-Prozessen [BALL90], *Active Messages* [vEick92], *Redirecting System Calls* [Pat93] und letztlich auch die *In-Kernel Server* [Lepr93].

Man kann aber auch überlegen, ob durch neue Rechnerarchitekturen Gründe für den Effizienznachteil obsolet werden, weil aufgrund mehrerer CPU's ein Großteil der Prozeßumschaltungen wegfallen und *ready*-Wartezeiten deutlich reduziert werden können. High-End–Systeme verfügen über n-Prozessoren, die nicht mehr notwendigerweise über einen Bus und damit gemeinsamen Speicher gekoppelt sind[15]. Multiprozessorsysteme setzen sich dabei nicht nur im High-End–Bereich durch. Bei mehreren Prozessoren gibt es a priori n-Prozesse ohne Effizienznachteil, so daß sich der Effizienznachteil gegenüber monolithischen Systemen mit der Anzahl der Prozessoren vermindern wird. Im Idealfall könnte man jedem Prozeß eine eigene reale CPU zuteilen, wie es für Prozesse verteilter Anwendungen in Workstation-Clustern im Prinzip schon gegeben ist. Das *Client-Server-Modell* aus dem Anwendungsbereich, wo es seit langem bekannt und akzeptiert ist, ist geradezu ideal auch für Betriebssystemkerne von Multiprozessorsystemen geeignet. Auch dieser Aspekt motiviert die in dieser Arbeit vorgeschlagene Systemarchitektur, unabhängig von den heute vorhandenen Effizienznachteilen auf Monoprozessormaschinen.

[13] vor allem für RISC aufwendig, wenn mehrere Registersätze der CPU zu retten sind [KT91]
[14] *"The Increasing Irrelevance of IPC Performance for μ-kernel-based Operating Systems"* [Bersh92]
[15] z.B. *Siemens RM1000* mit bis zu 214 MIPS-RISC-Prozessoren, hybrides SMM und MPM-Modell [Pyr96]

2.6 Zusammenfassung der Konzepte

In Abschnitt 2.3 wurden drei Prinzipien für dynamische Anpassung in einem ablaufenden System herausgestellt, und es wurden drei Techniken zu deren Umsetzung gezeigt.

- ***Anpassung des ablaufenden Systems:***

Prinzipien:		Techniken:
○ Adaptierung ○ Adaptivität ○ Reflexion	×	▷ dynamische Parametrisierung ▷ dynamische (Re-) Konfigurierung ▷ Meta(objekt)-Protokolle

---

Für die Seite der Softwaretechnologie wurde in Abschnitt 2.4 das Konzept der Betriebssystem-Familien aufgegriffen, und es wurden vier aktuelle Techniken vorgestellt, mit denen es heute für die Herstellung von Betriebssystemsoftware umgesetzt wird.

- ***Anpassung aus softwaretechnologischer Sicht:***

Prinzip:		Techniken:
○ Betriebssystem-Familien	×	▷ Klassen-Frameworks ▷ generische Betriebssysteme ▷ Offene Implementationen (*Open Implementations*) ▷ Entwurfsmuster (*Design Patterns*)

Zur Einordnung von Ablaufstrukturen wurden in Abschnitt 2.5 die zwei grundlegenden Varianten gegenübergestellt: monolithisch, prozedural (ggf. multi-threaded) und unabhängige Prozesse, aus denen Systeme schichtweise aufgebaut sind. Beide findet man im heute allgegenwärtigen Betriebssystemkern-Ansatz, dessen Ausprägungen in fünf Klassen eingeordnet wurden.

- ***Ablaufstrukturen in Betriebssystemen:***

Grundmuster:		Klassen von Betriebssystemkernen:
○ entkoppelte Prozesse ○ monolithisch, prozedural	×	▷ schwere Kerne ▷ monolithische μ-Kerne ▷ μ-Kerne ▷ Kerne mit *"In-Kernel" Servern* ▷ pico- und nano-Kerne

Zur Plazierung von Anwendungs- oder Betriebssystemfunktionen ergeben sich für die Kern-Architektur vier prinzipielle Möglichkeiten.

- ***Plazierung von Betriebssystemfunktionen in der Kern-Architektur:***
  - als weitere Instanzen in der Umgebung der Anwendung
  - als funktionale Schicht innerhalb einer Anwendungsinstanz (Laufzeitsystem)
  - als permanenter (statischer) Bestandteil des Kerns
  - als statische oder dynamische Erweiterung des Kerns

2.7 Dynamisch erweiterbare Kerne: *State-of-the-Art*

Eingangs wurden neue Forschungsprojekte aus den USA genannt, bei denen dynamische Erweiterbarkeit im Vordergrund steht. Ausgehend von den Erfahrungen mit μ-Kernen wird gegenwärtig durch dynamisch erweiterbare Kerne versucht, die Flexibilität des μ-Kern-Ansatzes mit der Effizienz monolithischer Kerne zu verbinden, indem Systemfunktionen wahlweise in den Kern-Bereich zurückgebracht werden können. Das Resultat sind dynamisch erweiterbare oder rekonfigurierbare Kerne. Sowohl Anwendungs- als auch Betriebssystemfunktionen sollen dabei dynamisch in den Kern eingebracht werden. Das bedeutet Offenheit des Kerns zur Laufzeit, und das erfordert klare Strukturen im Kern, damit es für Anwender auch praktikabel ist. Ein Schwerpunkt der Forschung liegt bei der Wahrung der Integrität des Kerns, die durch Offenheit erst einmal aufgegeben wird. Man versucht durch geeignete Techniken, einen Kompromiß zwischen Schutz (Integrität), Flexibilität und Effizienz zu finden.
Zur Diskussion des *State-of-the-Art* für dynamische Anpassung sollen fünf ausgewählte Projekte kurz charakterisiert werden:

- *SPIN*, University of Washington,
- *Exokernel*, M.I.T.,
- *Bridge*, Carnegie Mellon University,
- *V++ / Cache Kernel*, Stanford University,
- *In-Kernel Servers (INKS)* für *Mach3.0*, University of Utah.

▷ *SPIN* – University of Washington, [Bersh95]

SPIN stammt aus Umfeld von *Mach3.0*, wird jedoch als eine eigenständige Entwicklung fortgeführt. Dienste können in *SPIN* als:
- externe Server (wie bei μ-Kernen üblich),
- anwendungsspezifische Bibliotheken innerhalb einer Anwendung oder als
- Kern-Erweiterung (→ *spindle: SPIN – Dynamic Load Extension*)

implementiert werden (→ *service partitioning* [BCE+94]). Die dritte Variante bedeutet die dynamische Erweiterung des Kerns. Der Schutz der Integrität des Kerns wird ausschließlich auf die Mittel und Restriktionen der Sprache *Modula-3* zurückgeführt (→ *extension language*). Durch strenge Typprüfung wird auf Sprachniveau durchgesetzt, daß ausschließlich Code für erlaubte Zugriffe generiert werden kann, indem es nur für bestimmte Kern-Daten und Kern-Prozeduren Schnittstellendefinitionen gibt, z.B. für Datenstrukturen der Adreßumsetzungen oder für Kern-Threads. Der Compiler überprüft statische Bezugnahmen und generiert Prüfcode für dynamische Zugriffe über Zeiger und Feldindizes. *Spindle*-Code wird erst zum Ladezeitpunkt übersetzt, so daß die Restriktionen der Sprache auch wirksam werden und kein *Spindle*-Code auf anderem Wege generiert oder verfälscht werden kann.

▷ *Exokernel* – M.I.T., [Kaash95]

In *Exokernel* geht man noch einen Schritt weiter, indem nicht ein Kern mit festen Grundfunktionen erweitert werden kann, sondern daß im Kern lediglich Abstraktionen realer Hardware existieren, auf die sich Anwendungen abstützen. Es wird die reale Hardware über Zugriffsfunktionen nach oben angeboten. Der Kern unterstützt lediglich preemptive Prozesse in einem Adreßraum, die Verwaltung des physischen Speichers, den Netzwerkzugriff und das Laden und Starten von Anwendungen. Alle weiteren Dienste und Eigenschaften, beispielsweise auch separate Adreßräume und virtueller Speicher, werden in aufsetzbaren Bibliotheken als Teil der Anwendungen umgesetzt. In den Bibliotheken der Anwendungen werden die entsprechenden Belegungen der Hardware ausgeführt. Natürlich müssen alle Bibliotheken für einen sinnvollen

Ablauf kooperieren. Wichtig erscheint hier die mögliche Entkopplung des Ausführungsmodells von den Grundfunktionen, d.h., es muß keine separaten Adreßräume und keinen virtuellen Speicher geben, wenn alle (!) Anwendungen darauf abgestimmt sind. Bibliotheken werden als nicht vertrauenswürdig angesehen. Sicherheit wird durch Prüfen der Hardwarezugriffe in den Zugriffsfunktionen des Kerns unterstützt. In der Literatur sind keine Schutzmechanismen fest angegeben. Der Kern kapselt Hardware-Komponenten und kann sie exklusiv Anwendungen zuteilen bzw. auch wieder entziehen. Aus diesem Grund wurde auch das preemptive Prozeßmodell fest in den Kern eingebaut. Nach [Kaash95] gibt es derzeit zwei Kerne: *Aegis* für *DEC-MIPS* und *Glaze* für *Sparc-SMMP* und zwei Bibliotheken für Anwendungen: *ExOS* für *Aegis* (Unix-Schnittstelle) und *PhOS* für *Glaze* – *"a parallel operating system"* .

▷ *Bridge* – Carnegie Mellon University, [Lucco93]

Bei *Bridge* werden Erfahrungen zum softwareseitigen Schutz von Prozessen innerhalb eines Adreßraumes von der Universität Berkeley: *"Efficient Software-Based Fault Isolation"* [Lucco93] mit dem *Integrated Multiserver*-Projekt auf Basis *Mach3.0* an der CMU verbunden. *Integrated Multiserver* sind *In-Kernel Server*, bei denen innerhalb des Kern-Adreßraumes Schreibzugriffe von Server-Prozessen in Speicher anderer Server-Prozesse sicher unterbunden werden sollen.
Das könnte auch innerhalb des Kern-Adreßraumes hardwareseitig durch Setzen der Seitenzugriffsrechte erfolgen, allerdings würde das wieder mehr Aufwand bei Umschaltungen zwischen Kern-Prozessen bedeuten, da die entsprechenden Seitenzugriffsrechte extra verwaltet und mit umgeschaltet werden müßten. Statt dessen wird eine softwarebasierte Lösung angewandt, bei der ebenfalls sicher verhindert wird, daß Prozesse auf beliebige Adressen schreibend zugreifen.

Auf Sprachebene werden Zugriffe auf globale Daten oder Funktionen des Kerns durch den Compiler über definierte Symbolnamen und Schnittstellen geprüft. Um den übersetzten Code vor späterer Manipulation zu schützen, wird er mit einer digitalen Signatur versehen, die zum Ladezeitpunkt geprüft wird. Für dynamische Zugriffe wird beim Übersetzen Prüfcode generiert, für den in [Lucco93] zwei Methoden gegenübergestellt werden: *segment-matching* und *sandboxing*. Beiden ist gemeinsam, daß jedem Prozeß ein "privates Segment" innerhalb des Kern-Adreßraumes zugewiesen wird, in dem für ihn beliebige Zugriffe erlaubt sind. Außerhalb des Segments dürfen keine Schreibzugriffe erfolgen. Das gilt für alle Kern-Prozesse.
Der zusätzliche Laufzeitaufwand sollte minimal sein. Es wird angenommen, daß der Anteil durch Prüfcode abzusichernder Zugriffe klein ist. Lesezugriffe benötigen keine Schutzvorkehrungen, da sie die Integrität nicht verletzen. Schreibzugriffe erfolgen größtenteils auf lokale Variablen, die auf den privaten Stacks der ausführenden Server-Prozesse liegen und relativ adressiert werden. Schreibzugriffe auf globale Daten werden über Symbole durch den Compiler und Linker sicher auf feste Adressen aufgelöst. Direkte Assemblerprogrammierung und das Verändern von Registerinhalten ist durch die verwendete Sprache ausgeschlossen. Es bleiben letztlich Schreibzugriffe über Zeiger und Indizes in Feldern, für die Prüfcode zu generieren ist.
Bei *segment-matching* wird die referenzierte Adresse mit den Segmentgrenzen verglichen und bei Überschreitung eine Segmentverletzung erzeugt. Der Aufwand dafür wird bei *sandboxing* weiter reduziert, indem einfach gültige Segmentbits auf jede Adresse bei Schreibzugriffen aufgeprägt werden. Damit unterliegen die Segmentgrößen zwar den Ausrichtungsschranken der Bitmuster, und es können auch keine Segmentverletzungen mehr erkannt werden, aber es wird zumindest sichergestellt, daß nicht über die Segmentgrenzen hinweg schreibend zugegriffen wird.

```
(a): if( not_in_segment( ptr))        (b): ptr &= seg_mask;
       raise( segment_fault);                ptr |= seg_bits;
     *ptr = x;                               *ptr = x;
```

Abb.2.8 Software-Based Fault Isolation: `(a)`–segment-matching, `(b)`–sandboxing

In [Lucco93] sind Zahlen für den Overhead für beide Varianten gegenübergestellt. Der in Abbildung 2.8 angegebene Prüfcode könnte beim Laden in den Kern in den Maschinencode eingefügt werden, was allerdings aufwendig ist. Für *Bridge* wurde der *gcc*-Compiler für entsprechende Codegenerierung modifiziert. Der übersetzte Code wird als sicher angesehen, da Fälschungen durch die digitale Signatur sicher erkannt werden sollen.

▷ *V++ / Cache Kernel* – Stanford University, [Cheriton94]

V++ ist der Nachfolger des *V*-Systems [Che88]. Lokale μ-Kerne bildeten die Basis für das verteilte System *V*. Analog ist der *Cache Kernel* die Basis für *V++*. In [Cheriton94] werden auch grundlegende Defizite der "klassischen" μ-Kerne genannt (vgl. auch Abschnitt 2.8).
Die Architektur von *V++* ist dreigeteilt: (1.) *Anwendungen* auf Basis eines oder mehrerer (2.) *Application Kernels*, die auf dem (3.) *Cache Kernel* aufsetzen. Der *Cache Kernel* ist im Wortsinn als "Cache" für Systemdatenobjekte zu verstehen, die aus den Anwendungskernen in den *Cache Kernel* geladen bzw. entfernt werden können. Der *Cache Kernel* arbeitet als einziges Element im Kern-Modus. Der *Cache Kernel* kennt drei grundlegende Objekttypen: *Address Spaces* als Abstraktionen für Speicher, preemptive *Threads* für parallele Prozesse und sogenannte Anwendungskerne oder *Application Kernels*. Datenobjekte dieser drei Typen werden dynamisch in den *Cache Kern* geladen und sind dort Repräsentanten für Abbildungsbeziehungen virtueller Adreßräume, für Steuerblöcke von Prozessen, oder sie repräsentieren einen Anwendungskern.
Das Verhalten des *Cache Kernels* ist funktional fixiert: preemptives Scheduling von Prozessen, deren Datenobjekte im *Cache Kernel* enthalten sind. Bei Prozeßumschaltungen werden dann die Adreßraumumschaltungen ausgeführt, die in entsprechenden Datenobjekten beschrieben sind. Eine weitere Funktion besteht in der Signalisierung von Anwendungskernen, damit diese bei äußeren Ereignissen die Cache-Konfigurierung entsprechend ändern können. Das Blockieren eines Anwendungsprozesses wird nicht vom *Cache Kernel* ausgeführt, sondern vom zugehörigen Anwendungskern, der das repräsentierende Datenobjekt für den Prozeß aus dem *Cache Kernel* entfernt. Es haben nur bereite Prozesse eine Datenrepräsentation im *Cache Kernel*, so daß dieser keine Prozeßzustände unterscheiden muß. Wird ein blockierter Prozeß wieder bereit, wird das entsprechende Datenobjekt vom Anwendungskern wieder in den *Cache Kernel* eingetragen.
Es können verschiedenartige Anwendungskerne kooperativ gleichzeitig ablaufen, und es werden alle Hardwareressourcen in die zwei Grundabstraktionen Speicher und Prozeß eingeordnet, auch periphere Geräte. Dieses Herangehen resultiert aus der Zielplattform für *V++*, der *ParaDiGM*-Maschine (4-CPU, Motorola 68040, MPM mit je 2 MByte lokalem RAM). Motorola-Prozessoren verwenden *memory-mapped-I/O*, so daß die Speicherabstraktion auch für Ein- bzw. Ausgabe paßt. Dieses Prinzip ist aber nicht auf beliebige Prozessoren übertragbar.
V++ / Cache Kernel ist zu *Exokernel* ähnlich, bei dem der Kern auch lediglich Hardwareabstraktionen bereitstellt und die Betriebssystemfunktionen in Bibliotheken enthalten sind, die bei *Exokernel* allerdings im Kern-Modus ausgeführt werden, woraus sich die genannten Sicherheitsprobleme für *Exokernel* ergeben. Für den *Cache Kernel* wird lediglich Parametrisierung über Datenobjekte zugelassen, es ist kein neuer Code für den Kern ladbar. Der *Cache Kernel* selbst wird als integer angesehen. Seine Funktion ist nicht durch Software manipulierbar, da der Code in einem *PROM* abgelegt ist. Der Kern umfaßt ca. 130 kByte.

▷ *In-Kernel Servers (INKS) für Mach3.0* – University of Utah, [Lepr93]

Auch in diesem Projekt liegt der Schwerpunkt auf der Minderung der Effizienznachteile von μ-Kernen. Am Beispiel des *OSF/1-Single-Servers* wird gezeigt, daß *Unix*-Systemrufe auf *Mach* um den Faktor 3 schneller ausgeführt werden können, wenn der *Unix*-Server nicht in der Anwendungsschicht angesiedelt ist, sondern im Kern plaziert wird und die Interaktionsschnittstelle von *Mach-RPC's* auf *traps* reduziert wird. Dieses Ergebnis ist intuitiv leicht nachzuvollziehen, es

wundert eher, daß *In-Kernel Server* nicht früher für *Mach* angewandt wurden, zumal es in *Chorus* das Pendant in Form der *Protected Subsystem Actors* schon von Anfang an gab [RAA+88]. Ein Grund dafür liegt in der Aufgabe der Schutzfunktion getrennter Adreßräume für Server. In *INKS* steht aber die Frage der Realisierbarkeit für *Mach* und der mögliche Effizienzgewinn im Vordergrund. Die Abschottung von *In-Kernel Servern* bleibt vorerst offen. Der *OSF/1-Unix*-Server wird als vertrauenswürdig angesehen, was natürlich nicht für beliebige *In-Kernel Server* verallgemeinerbar ist. In *Bridge* wurden aber Wege für softwareseitige Abschottung gezeigt.
Für *INKS* wurde der *Mach*-Kern dahingehend modifiziert, daß die Option besteht, einen im Quellcode unveränderten *OSF/1*-Server wahlweise als *"In-"* oder *"Out-Kernel" Server* ablaufen zu lassen. Ein modifizierter Stub-Generator *KMIG* erzeugt entsprechend veränderten Code für Systemrufe (→ *traps* statt *Mach-RPC*) für den *In-Kernel Server*. Gegenwärtig kann zum Bootzeitpunkt die zu ladende Server-Variante ausgewählt werden. Erweiterungen für mehrere *In-Kernel Server*, das dynamische Laden neuer *In-Kernel Server* und die Behandlung der offenen Schutzfrage sind gegenwärtig aber nicht vorgesehen.

Als Zusammenfassung kann die folgende Übersicht für die genannten fünf Forschungsprojekte angeben werden:

	SPIN	*Exokernel*	*Bridge*	*Cache Kernel*	*Mach-INKS*
dynamisch Code laden Parametrisierung	spindles[16] –	libraries[16] –	libraries[16] –	– Objekte	server –
Kern-Dienste Ausführungsmodell	variabel fest	variabel variabel[17]	variabel fest	fest fest	fest fest
Sprache spezieller Compiler	Modula-3 trusted comp.	C/C++ –	C/C++ modifiz. *gcc*	C++ –	C/C++ –
Schutz ▷ Sprache, statisch ▷ Prüfcode, dynam. ▷ Merkmale	 ja ja Übersetzen und Prüfen zur Ladezeit	 ja ja nicht angegeben: "Hardwareschutz"	 ja ja Prüfcode und .o-Files mit dig. Signatur	 ja nein keine, da kein Code geladen wird	 ja nein keine, der OSF/1-Server "ist integer"
Kern-Kategorie	erweiterbarer μ-Kern	pico-Kern	erweiterbarer μ-Kern	parametr. pico-Kern	INKS-μ-Kern

Abb.2.9 Übersicht zu dynamisch erweiterbaren μ-Kernen

[16] als Teil der Anwendungen
[17] z.B. mit oder ohne getrennte Adreßräume oder virtuellem Speicher

• Single Address Space Operating Systems

Es sei eine Anmerkung zu *Single Address Space Operating Systems* gestattet, die eine interessante Entwicklung für die Zukunft aufzeigen. Betriebssysteme ohne getrennte Adreßräume waren seit jeher im Gebrauch, weil die Hardware dieses Konzept nicht unterstützte, weil für spezielle Anwendungen dafür keine Unterstützung nötig war oder der Preis für die Verwaltung mehrerer Adreßräume nicht angemessen erschien. Die "alten" Betriebssysteme (*Multics* [DD68], *THE* [Dij68b], *Hydra* [WCC+74], *Pilot* [RDH+80], *Monads* [Kee89] u.v.a.) hatten aufgrund der fehlenden Hardwareunterstützung für indirekte Adreßumsetzung von vorn herein nur einen Adreßraum, der gegebenenfalls in Segmente eingeteilt war. Die Schutzfunktion für Speicher einzelner Prozesse wurde beispielsweise über Speicherschlüssel realisiert. Vielleicht wird die Idee der Speicherschlüssel mit einer Unterstützung in den Speicherbausteinen in Zukunft auch wieder aufgegriffen. In Betriebssystemen und Prozessoren könnte viel Aufwand gespart werden, wenn man auf getrennte Adreßräume verzichtet. Das bezieht sich nicht nur auf die Umschaltzeiten zwischen Adreßräumen, die nach [Lie95] nicht mehr in dem Maße relevant sind, es gehört auch die Verwaltung umfangreicher Datenstrukturen dazu, die einen signifikanten Aufwand darstellt. Wenn im Kern natürlich nur die Umschaltmechanismen enthalten sind und die Verwaltungsaufgaben außerhalb des Kerns erfolgen, wird dieser Aufwand bei isolierter Betrachtung des Kerns nicht spürbar. Im Kontext eines Endsystems fällt er umso mehr ins Gewicht. Die gleiche Aussage gilt auf einer anderen Ebene auch für virtuellen Speicher. Ein Großteil der Komplexität, die *Mach*-ähnlichen μ-Kernen zugeschrieben werden muß, resultiert aus diesen beiden Speichertechniken, die nach wie vor als eine Grundanforderung an universelle Betriebssysteme angesehen werden. Aber gerade in der (optionalen!) Reduktion teurer Ausführungsmodelle liegt eine wichtige Quelle für Effizienzgewinn, die man gerade (wieder) entdeckt und für heutige Anforderungen zumindest in der Forschung zu untersuchen beginnt. Eine gleichwertige Abschottung der Prozesse voreinander kann auch auf andere Art erfolgen, Methoden dafür wurden genannt.

Es muß allerdings gesagt werden, daß die neue Generation von *Single Address Space Operating Systems* primär eine andere Zielstellung verfolgt als die Vereinfachung der Kerne. Es geht um die global eindeutige Identifikation von Objekten in 64-Bit Adreßräumen für verteilte Systeme. Untersuchungen in diesem Umfeld erfolgen gegenwärtig in: *Opal* [CLFL94], *Nemesis* [Ros94] oder *Pegasus* [LMM93]. Dennoch, ein großes Potential dieses Ansatzes liegt auch in der möglichen Abrüstung der Ausführungsmodelle der Kerne.

2.8 Zusammenfassung und Fazit

Nach 15 Jahren Forschung zu μ-Kernen ist der Hoffnung auf Ablösung monolithischer Betriebssysteme in der breiten Anwendung Ernüchterung gefolgt. Das zeigt sich u.a. daran, daß 1994 die Arbeiten zu *Mach* an der CMU eingestellt wurden, sie werden auf einigen Gebieten an anderen Einrichtungen in kleinerem Umfang fortgesetzt.
μ-Kerne sind bezüglich Effizienz nur dann zu monolithischen Kernen konkurrenzfähig, wenn sie als *monolithische μ-Kerne* ausgeprägt sind (vgl. Abbildung 2.7), oder sie sind auf Basis von *In-Kernel Servern* implementiert. Die zuerst genannte Variante ist sicher nicht befriedigend (*Mach* bis Version *2.5*), die zweite erscheint als ein gangbarer Weg für die Zukunft. Sie wurde aber bislang von der *Mach*-Euphorie überschattet und kommt erst jetzt wieder stärker zur Geltung. In *Chorus* waren *In-Kernel Server* von Anfang an vorgesehen.
Bedenkt man zudem, daß in der Regel *Unix*-Anwendungen auf μ-Kernen ablaufen, muß aus Sicht der Anwender die Frage beantwortet werden, welchen Vorteil es für sie hat, ein emuliertes *Unix* anstelle eines effizienteren monolithischen *Unix*-Kerns zu verwenden. Oft wird die Flexibilität der μ-Kerne in die Diskussion gebracht. Aber wird sie von Anwendern genutzt? Werden

optimierte Server für Betriebssystemdienste tatsächlich hergestellt und eingesetzt? Sicher gibt es Beispiele, aber der Normalfall ist es nicht. μ-Kerne haben aber gezeigt, daß man Systeme auf einer minimalen funktionalen Basis aufbauen kann ([Han70] : *Prozesse, Speicher, IPC*).

Bei μ-Kernen fällt sofort die Analogie zur Prozessortechnik auf, bei der es mit *RISC*-Prozessoren gelang, die *CISC*-Technologie für eine Phase der Entwicklung abzulösen. Auch hier wurden im Verhältnis wenig benutzte Funktionen (Instruktionen) aus den Prozessoren entfernt, allerdings nicht mit dem Ziel funktionaler Minimalität, sondern höherer Effizienz der verbleibenden Instruktionen (1 Zyklus). Der Erfolg von *RISC* läßt sich durch eine einfache Bilanzgleichung begründen, daß eine gleiche Anwendungsaufgabe zwar mit mehr *RISC*- als *CISC*-Instruktionen ausgeführt wird, das aber im Endeffekt oder im Kontext eines Endsystems schneller erfolgt. Der quantitative Mehraufwand wird durch schnellere Ausführungszeiten überkompensiert. In dieser Überkompensation, die durch geschickte Compilertechniken verstärkt wird, liegt der Effizienzvorteil und damit der Grund für den Erfolg von *RISC*. In Zukunft kann sich das Blatt wenden, oder beide Technologien fließen wieder zusammen, wenn immer komplexere Instruktionen in einem oder in wenigen Zyklen ausgeführt werden können.

In gewisser Weise kann man die funktionale Abrüstung der μ-Kerne als ein Pendant zur *RISC*-Idee betrachten, obwohl *RISC*-Prozessoren später und unabhängig vom μ-Kern-Ansatz entstanden. Auch μ-Kerne erfordern durch reduzierte Funktion Mehraufwand, vor allem mehr IPC. Von Kompensation, geschweige Überkompensation durch schnellere Ausführung kann jedoch keine Rede sein. Hier liegt das Problem des μ-Kern-Ansatzes, das bis heute nicht geeignet gelöst ist. Auf dem Dagstuhl-Workshop *"Operating Systems of the 90s and Beyond"* [KN91] gab *Bill Joy* (*Sun*) seine Einschätzung zu jüngeren objektbasierten- und objektorientierten Betriebssystem-Entwicklungen: *" If we seek an analogy to "RISC" in operating systems, I don't think most object-based systems will be RISC-like. We may have to look elsewhere first to get a truly simple and powerful system."* [Joy91], S. 39[18]. Die folgende Ansicht drückt gewissermaßen gegenteilige Zweifel aus: *"Are object systems really object oriented?"* [Hor89], S. 69. Beide Auffassungen zeigen auch die Unwägbarkeiten, die mit objektorientierten Betriebssystemen bis heute einher gehen.

Die Kritik an μ-Kernen und die Gründe für die fehlende Akzeptanz werden gegenwärtig breit diskutiert. Die Ursachen für diese Situation werden heute klarer erkannt und auch benannt.

> " The advantages in modularity and power of microkernel-based operating systems such as Mach 3.0 are well known. The existing performance problems of these systems, however, are significant. Much of the performance degradation is due to the cost of maintaining separate protection domains, traversing software layers, and using semantically rich inter-process communication mechanism. " *Mach-INKS* [Lepr93], S. 39.

> " Operating system research has endeavored to develop micro-kernels to provide modularity, reliability and security improvements over conventional monolithic kernels. However, the resulting kernels have been slower, larger and more error-prone than desired. ...
> Micro-kernels to date have not provided compelling advantages over the conventional monolithic operating system kernel for several reasons.
> First, micro-kernels are larger than desired because of the complications of a modern virtual memory system ... Moreover, performance problems have tended to force services originally implemented on top of a micro-kernel back into the kernel, increasing its size. ...
> Second, micro-kernels do not support domain-specific resource allocation policies any better than monolithic kernels, an increasingly important issue with sophisticated applications and application systems. ... Placement of conventional operating system kernel services in a micro-kernel-based server does not generally give the application any more control because the server is a fixed protected system service. " *V++ / Cache Kernel* [Cheriton94], S. 179.

[18] *Sun's* Prototypkern *Spring* löst sich jedoch auch nicht von traditionellen μ-Kern-Konzepten [Schö96].

> " The problem with micro-kernel based systems is that they tightly couple modularity and protection – servers are implemented in separate protection domains, and consequently, the communication mechanisms are designed for a cross-domain case. " *Lipto* [DPH92], S. 512.

Aus diesem Zusammenhang läßt sich der neue Aufbruch in der Betriebssystemforschung erklären, mit neuen Ideen und unkonventionellen Ansätzen Lösungswege zu suchen. Aktuelle Beispiele wurden für einige Projekte gezeigt.

Es geht jetzt im Grunde darum, Kerne nach der Abrüstung auf Grundfunktionen auch in den ***Ausführungsmodellen*** abzurüsten bzw. die ***Wahlmöglichkeit*** für Anwendungen zu eröffnen, angepaßte Ausführungsmodelle herzustellen. Das heißt auch, gewohnten Komfort gegen Effizienz eintauschen zu können, wenn dies erwünscht oder notwendig ist. Mittel dafür können sein, die "Illusion unbeschränkter Ressourcen" aufzugeben, indem kein virtuelles Speicherkonzept umgesetzt wird, vorausgeplante oder statische Ressourcenzuteilung zu benutzen oder auf Schutz für bestimmte Anwendungsfälle zu verzichten.
Entscheidend ist, daß die Option besteht, nicht nur Strategien aus dem Kern zu lösen, sondern daß auch Mechanismen auf Anforderungen von Anwendungen zuschneidbar sind. Die Chance, universelle monolithische Systeme abzulösen, besteht letztlich nur darin, Leistungspotentiale durch Anpassung an konkrete Anwendungen in einem Maße zu erschließen, das für universelle Betriebssysteme nicht möglich ist. Wenn das auf praktikablem Wege gelingt, ist auch ein signifikanter Effizienzvorteil zu erwarten, der dieser Art von Systemen zum Durchbruch verhelfen kann. Der Weg bis zu diesem Ziel ist jedoch noch weit, zumal in der Praxis auch noch andere Randbedingungen bestehen (→ *Kompatibilität, Interaktionsfähigkeit* u.a.).

Exokernel und *Cache Kernel* bieten die interessantesten Ansätze. Die anderen Systeme führen die Historie von *Mach* mit sich, von der sie sich langfristig jedoch lösen wollen (*Spin*, *Bridge*).
Neben den Ideen und Lösungsvorschlägen für anwendungsspezifische Erweiterbarkeit von Betriebssystemkernen und dem Streben nach Effizienz, die gegenwärtig den Schwerpunkt darstellen, wird dabei eine wesentliche Frage sein, daß die Ausschöpfung der Leistungspotentiale durch anwendungsbezogene Gestaltung von Betriebssystemen die Mitarbeit der Anwendungsentwickler erfordert, die unter Nutzung der Basismechanismen diese Anpassungen vornehmen sollen.

Es ist damit eine mindestens gleichwertige Forderung an offene, durch Anwender erweiterbare oder modifizierbare Betriebssysteme, daß diese nicht nur prinzipielle Möglichkeiten bereitstellen, sondern auch eine ***plausible innere Ablaufstruktur*** aufweisen, die es Anwendern tatsächlich ermöglicht, sinnvolle Eingriffe in das System in akzeptabler Zeit auszuführen. Dabei geht es nicht allein um die Offenheit und Struktur von Quelltext, sondern primär um das Verstehen von Grundelementen, Zusammenhängen und Vorgängen im ablaufenden System. In Kapitel 2 wurde der Unterschied zwischen Software- und Ablaufstruktur ausführlich begründet.
Auf dem Dagstuhl-Workshop *"Operating Systems of the 90s and Beyond"* [KN91] wurde dieser seit langem bekannte, aber nach wie vor offene Wunsch wieder angesprochen: *"The Explainable Operating System"* [Wett91]. Einfache Lösungen gibt es dafür sicher nicht. Es erscheint aber naheliegend, und vielleicht ist es ein gangbarer Weg, im Inneren eines Betriebssystems prinzipiell die gleiche Ablaufstruktur herzustellen, die im Anwendungsbereich seit langem bekannt und breit akzeptiert ist. Genau hier liegt der Ansatzpunkt dieser Arbeit.

Gelingt es nicht, Plausibilität im Inneren des Betriebssystems herzustellen, bleibt Offenheit ein abstraktes Konzept, wie es heute für eine Reihe von *Unix*-Systemen der Fall ist, die zwar das dynamische Einbringen neuer Module in den Kern erlauben, die Herstellung dieser Module allerdings detaillierte Kenntnis kerninterner Strukturen und Abläufe voraussetzt, die heute Anwendungsentwicklern, und damit der Zielgruppe für anwendungsorientierte Anpassungen, nicht oder nur schwer zugänglich oder vertraut ist und einen hohen Anfangsaufwand erfordert.

Die Probleme des Schutzes und der Integrität bei anwendungsspezifischer Erweiterung von Kernen sind wichtige Fragestellungen, die gegenwärtig in verschiedenen Forschungsprojekten im Mittelpunkt stehen. Aber auch für die hier angesprochenen Problemstellungen müssen anwendungsanpaßbare Betriebssysteme neue Lösungen anbieten. Ein Vorschlag unter diesem Blickwinkel wird in dieser Arbeit gemacht. Es wird dabei die Gesamtarchitektur des ablaufenden Systems in den Vordergrund gestellt. In vielen aktuellen Forschungsprojekten spielt diese Frage eine untergeordnete Rolle. Dort geht es um die prinzipielle Machbarkeit, den Grad der Effizienzverbesserung und die Lösung der offenen Schutzfrage. Insofern soll in dieser Arbeit eine andere Facette mit folgenden Schwerpunkten betrachtet werden:

- Überwindung des Strukturbruchs typischer Kerne durch ein homogenes Grundmuster in allen Schichten des Systems, welches auch innerhalb des Kerns eine Ordnung für Elemente und Abläufe schafft, die an der bekannten und akzeptierten Ordnung im Anwendungsbereich orientiert ist;

- explizite Beachtung verschiedener Ausführungsmodelle in verschiedenen Schichten des Systems; Transparenz im Sinne von Durchschaubarkeit von Strukturen und Abläufen als eine wichtige Voraussetzung für anwendungsanpaßbare Infrastrukturen;

- das *Client-Server-Modell* bietet sich als Grundmuster an, welches allerdings für verschiedene Ausführungsmodelle zu skalieren ist; Skalierung bedeutet, die starre Granularität von Ausführungseinheiten und die in der Infrastruktur fixierten Ausführungseigenschaften aufzuheben, um dieses Modell tatsächlich auf *alle Schichten* eines Systems übertragen zu können; derzeit gibt es keine konsequente Anwendung dieses Modells für alle Schichten in einem Betriebs- oder Anwendungssystem;

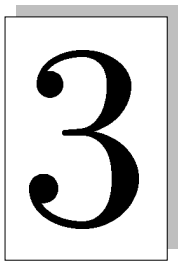
- die explizite Abbildung programmiersprachlicher Einheiten in Ausführungseinheiten einer Maschine, um das Problem des Unterschieds von Software- und Ablaufstruktur zu klären; dabei spielt insbesondere die Einbettung von Elementen objektorientierter Sprachen in Ausführungseinheiten der ablaufenden Maschine eine wichtige Rolle;

- im *Client-Server-Modell* liegt das Potential, objektorientierte Softwareentwicklung durch eine geeignete Ablaufstruktur zu unterstützen und damit eine Brücke zwischen Software- und Ablaufstruktur zu schlagen; die Schlüsselidee des *CHEOPS*-Projekts liegt im Zusammenfließen von objektorientierten Technologien zum Entwurf und zur Herstellung von Software für Betriebssysteme mit einer Ablaufstruktur auf Basis von Schichten, die innerhalb nach dem *Client-Server-Modell* strukturiert sind.

Diese Dissertation befaßt sich mit der Architektur, um Anwendungsanpaßbarkeit in Betriebssystemen mit den genannten Schwerpunkten zu unterstützen. Auf dieser Grundlage wird dann im *CHEOPS*-Projekt versucht, für einige der angesprochenen Fragen Lösungen zu erarbeiten. Anhand der Implementierung des *CHEOPS*-Kerns wird gezeigt, inwieweit sich das Ziel der Homogenisierung der Grundstruktur auch innerhalb des Kerns erreichen läßt und welcher Preis auf heutigen Maschinen dafür aufzuwenden ist.

Im folgenden Kapitel schließt sich eine Analyse wichtiger Erkenntnisse, Sichtweisen und Beobachtungen zu Betriebssystemen an, die als Grundlage und zur Begründung der im nachfolgenden Kapitel entworfenen Architektur dient.

Anschließend wird die Umsetzung dieses Modells anhand der Implementierung des *CHEOPS*-Kerns gezeigt und bewertet.



Kapitel 3

Modellierungsgegenstand Betriebssystem

In diesem Kapitel sollen wesentliche Aspekte des Ablaufs und der Struktur von Betriebssystemen als Teilbereich allgemeiner informationsverarbeitender Systeme modellhaft zusammengefaßt werden. Die Diskussion dient der Verdeutlichung des Diskursbereichs und der Begründung der im nächsten Kapitel vorgeschlagenen Architektur. Sie basiert auf Erkenntnissen und Sichtweisen aus *"Betriebssysteme"* [Kal90] und *"Systemarchitektur"* [Wett93] und wurde um Aspekte der Abgrenzung von Teilsystemen ergänzt, die als Verarbeitungsinstanzen modelliert werden und die selbständig Dienste ausführen. Verarbeitungsinstanzen werden in zwei Dimensionen rekursiv zueinander in Beziehung gesetzt:

- *Aggregationsrekursion:* Verarbeitungsinstanzen enthalten Verarbeitungsinstanzen und
- *Infrastrukturrekursion:* die Infrastruktur als Existenzgrundlage für Verarbeitungsinstanzen besteht selbst aus Verarbeitungsinstanzen einer elementareren Art.

Für die Modellbetrachtung hier und die Diskussion der Architektur im nächsten Kapitel soll keine Trennung zwischen Anwendungs- und Betriebssystem mehr vorgenommen werden, sondern es soll ein Gesamtsystem in einer Systemarchitektur erfaßt werden. Es steht das ablaufende System im Vordergrund, nicht die Struktur der Software zu seiner Herstellung.

3.1 Informationsverarbeitende Systeme

Der Zweck eines informationsverarbeitenden Systems ist es, informations- und steuerungsbezogene Aufgaben in einem Problembereich eines Anwenders zu lösen. Informationsverarbeitung kann als Spezialform allgemeiner Verarbeitungs- und Produktionsprozesse angesehen werden mit der Besonderheit, daß Informationen aus dem Problembereich eines Anwenders Gegenstand und Ergebnis der Verarbeitung sind. Insofern gelten allgemeine Prinzipien von Verarbeitungs- und Produktionsprozessen auch für Informationsverarbeitung:

- **Beziehung zur Umgebung:** Ein System ist von seiner *Umgebung* räumlich, technisch und organisatorisch abgegrenzt. Aus der Umgebung werden ihm *Verarbeitungsgegenstände* (hier Informationen aus dem Problembereich) zugeführt, verarbeitet und als Endprodukt wieder an die Umgebung abgegeben. Der Verarbeitungsprozeß wird aus der Umgebung heraus initiiert und gesteuert. Man kann ein System auf seine Beziehungen zur Umgebung reduziert betrachten, so daß der innere Verarbeitungsprozeß verborgen bleibt.

- **Aktivität und Steuerung:** Der Prozeß der Verarbeitung ist durch *aktives Handeln* mindestens eines Elements des Systems geprägt (→ *Handlungsträger*). Handeln setzt dabei *Aktivität und Steuerung* voraus, so daß mit Verarbeitungsgegenständen selbständig eine Folge vorgesehener Verarbeitungsschritte ausgeführt wird. Auch in der Umgebung muß es Handlungsträger geben, die mit dem System arbeiten, es versorgen und steuern (→ *Protokolle*). Verarbeitung ist ein zielgerichteter, d.h. gesteuerter Vorgang. Im System muß es somit eine *Steuerung der Aktivität(en)* geben. Steuerung erfordert *Steuerungsinformation* über den geplanten Ablauf, die in das System bei der Herstellung oder während des Ablaufs eingebracht werden muß. Steuerung ist selbst ein aktiver Prozeß, der wiederum zu steuern ist. Systemsteuerung gliedert sich in zwei Kategorien:
  - *"Steuerung von außen"* (aus der Umgebung): legt fest, *was* mit Verarbeitungsgegenständen unter *welchen Bedingungen*, beispielsweise über den zeitlichen Ablauf, geschehen soll (funktionaler und konditionaler Aspekt der äußeren Steuerung),
  - *"innere Steuerung":* gibt durch eine Auflösung in Teilschritte an, *wie* dies erfolgt.

- **Innere Gliederung:** Ein System kann aufgabenbezogen gegliedert werden. Man kann fünf grundlegende Aufgabenklassen ableiten:
  - *Entgegennahme* bzw. *Abgabe* von Verarbeitungsgegenständen und Steuerungsinformation aus der bzw. an die Umgebung,
  - *Transport* von Verarbeitungsgegenständen im System,
  - *Lagerung* von Verarbeitungsgegenständen im System,
  - *Verarbeitung* (Teilschritte, Teilprozesse) und
  - die *Steuerung* des Ablaufs und des Zusammenspiels der Elemente.

  Weiterhin kann man drei grundlegende Kategorien von Systemelementen zusammenfassen:
  - passive *Grundelemente*,
  - aktive *Handlungsträger* und
  - abgeschlossene *Teilsysteme*.

  Grundelemente sind in (Teil-) Systemen Gegenstand der Verarbeitung durch Handlungsträger. Sie sind aber nicht selbständig handlungsfähig. ***Handlungsträger*** führen auf Grundlage gesteuerter Aktivität Teilprozesse mit Grundelementen aus bzw. initiieren Abläufe in ***Teilsystemen***, die diese dann intern selbständig ausführen (→ *Rekursion von Teilsystemen*, → ***Aggregationsrekursion***). Teilsysteme erfordern eigene Handlungsträger mit innerer Steuerung und innerer Aktivität. Die äußere Steuerung ist durch die Handlungsträger des umgebenden Systems gegeben. Aus der Auflösung eines Systems in Teilsysteme folgt auch eine rekursive Auflösung der Steuerung in Steuerungsebenen.

$$System_i = \{ Grundelemente_{i,i} \} \cup \{ Handlungsträger_{i,j} \} \cup \{ Teilsysteme_{i,k} \}$$

Rekursion: $Teilsystem_{i,k} = System'_k$, Abbruch bei: $System'_k = \emptyset$.

- **Infrastruktur:** Existenz und Arbeitsfähigkeit eines Systems setzen eine ***Infrastruktur*** voraus, welche auch die Beziehungen zur Umgebung vermittelt. Die Infrastruktur muß weiterhin den internen Verarbeitungsprozeß des Systems und die Funktion seiner Elemente sichern, wie die Versorgung mit Energie, räumliche und technische Gegebenheiten usw.
  Infrastruktur besteht selbst aus Grundelementen, Handlungsträgern und Teilsystemen, die ihrerseits einer Infrastruktur bedürfen. Orthogonal zur Aggregationsrekursion entsteht somit eine ***Infrastrukturrekursion***, was auch bedeutet, daß Infrastruktur prinzipiell nach demselben Muster modelliert werden kann.

- **Herstellung:** Informationsverarbeitung ist selbst ein Herstellungsprozeß von Endinformationen aus Ausgangsinformationen. Voraussetzung für diesen Herstellungsprozeß ist die vorherige Herstellung:
  - der Verarbeitungsgegenstände (Ausgangsinformationen),
  - der Steuerungsinformation der Verarbeitungsprozesse (Technologien, Verfahren, Algorithmen),
  - des Systems und seiner Elemente (Grundelemente, Handlungsträger, Teilsysteme),
  - von Werkzeugen und Hilfsmitteln und
  - der Infrastruktur.

  Die Herstellungsprozesse dafür sind selbst wieder Verarbeitungsprozesse und sind dem Verarbeitungsprozeß des Systems vorausgegangen (→ ***Rekursion der Herstellung***). Diese dritte Art von Rekursion für die Herstellung soll hier aber nicht weiter betrachtet werden.

Information vermittelt Aussagen über Erkenntnisse und Abläufe in einem Teilbereich der umgebenden Realität von Menschen (*Betrachtungswelt*). Unter ***Verarbeitung von Information*** soll primär die *Verknüpfung kodierter Information* verstanden werden. *Speicherung* und *Übertragung* sind inhärent damit verbunden und werden deshalb mit unter Verarbeitung gefaßt. Für Verarbeitung oder Herstellung ist Steuerungsinformation erforderlich. Umgekehrt ist Steuerungsinformation selbst Verarbeitungsgegenstand in informationsverarbeitenden Systemen. Dieses Phänomen wurde 1946 von John von Neumann als Äquivalenz von Steuerung und Information erkannt.

Eine andere Besonderheit zeichnet informationsverarbeitende Systeme aus: Softwareelemente eines Systems bestehen selbst nur aus Information. Handlungsträger und Teilsysteme dieser Art führen dabei ebenso Verarbeitungsprozesse aus und sind real vorhanden, wenn auch nicht materiell faßbar. Sie existieren in Form kodierter Information im System. Für diese Elemente gelten aber dieselben Aussagen, wie sie oben für Systeme und Teilsysteme getroffen wurden. Die Differenzierung in Hard- und Softwareelemente ist somit für die strukturellen und funktionalen Wesenszüge eines informationsverarbeitenden Systems nicht wesentlich. Es besteht natürlich die Abhängigkeit der Softwareelemente von der Existenz der Hardwareelemente als Infrastruktur. Die Herstellung von Softwareelementen erfolgt in einem informationsverarbeitenden System als Resultat von Verarbeitungsschritten anderer Elemente desselben Systems, in der Regel über Transformations- und Erzeugungsprozesse in der (Software-) Infrastruktur.

Die enge Verflechtung zwischen Verarbeitung, Steuerung und Herstellung, die sich alle auf den Gegenstand *"kodierte Information – Daten"* beziehen, macht die Besonderheit informationsverarbeitender Systeme aus und muß berücksichtigt werden, wenn es um die Festlegung einer Systemarchitektur geht. Es ist insofern relevant, die Natur informationsverarbeitender Systeme weiter zu untersuchen, um daran die später entwickelte Architektur zu orientieren.

3.2 Äußere Charakteristik

Ein informationsverarbeitendes System kann von außen als eine umschließende Verarbeitungseinheit oder Verarbeitungsinstanz[1] (***VI***) angesehen werden. Die Beziehung einer VI zu ihrer Umgebung (***U***) ist gekennzeichnet durch den:

- *Datenfluß* – Verarbeitungsgegenstände und Ergebnisse und den
- *Steuerfluß* – Information für bzw. über den Ablauf der Verarbeitung.

[1] Innere Zustände von Verarbeitungsinstanzen werden in Abschnitt 3.3 betrachtet.

Informationen aus dem Problembereich einer Anwendung werden der VI über die Umgebung als Eingabedaten kodiert zugeführt. Ausgabedaten werden an die Umgebung zurückgegeben und dort entsprechend interpretiert, auch zur Steuerung von Abläufen in der Umgebung.

Gleichzeitig ist eine Steuerung der VI aus der Umgebung erforderlich. Sie gibt die funktionale Wirkung des Verarbeitungsprozesses (*"was"* soll getan werden) und die Bedingungen für den Ablauf an. Diese Steuerungsebene beinhaltet nicht die interne Auflösung in Teilprozesse (*"wie"*). Das Steuerflußprotokoll ist bidirektional, d.h., Steuerungsinformationen werden aus der Umgebung an die VI gegeben und von der VI auch an die Umgebung zurückgeliefert, beispielsweise wenn eine Verarbeitung beendet ist. Die Art der Kodierung dieser Steuerungsinformation kann sehr unterschiedlich ausgeprägt und auch an die Ein- bzw. Ausgabedaten gekoppelt sein. So kann bereits das Vorliegen von Eingabedaten als Steuerungsinformation zum Anstoßen der Verarbeitung interpretiert werden, wie es bei Datenflußmaschinen zutrifft.

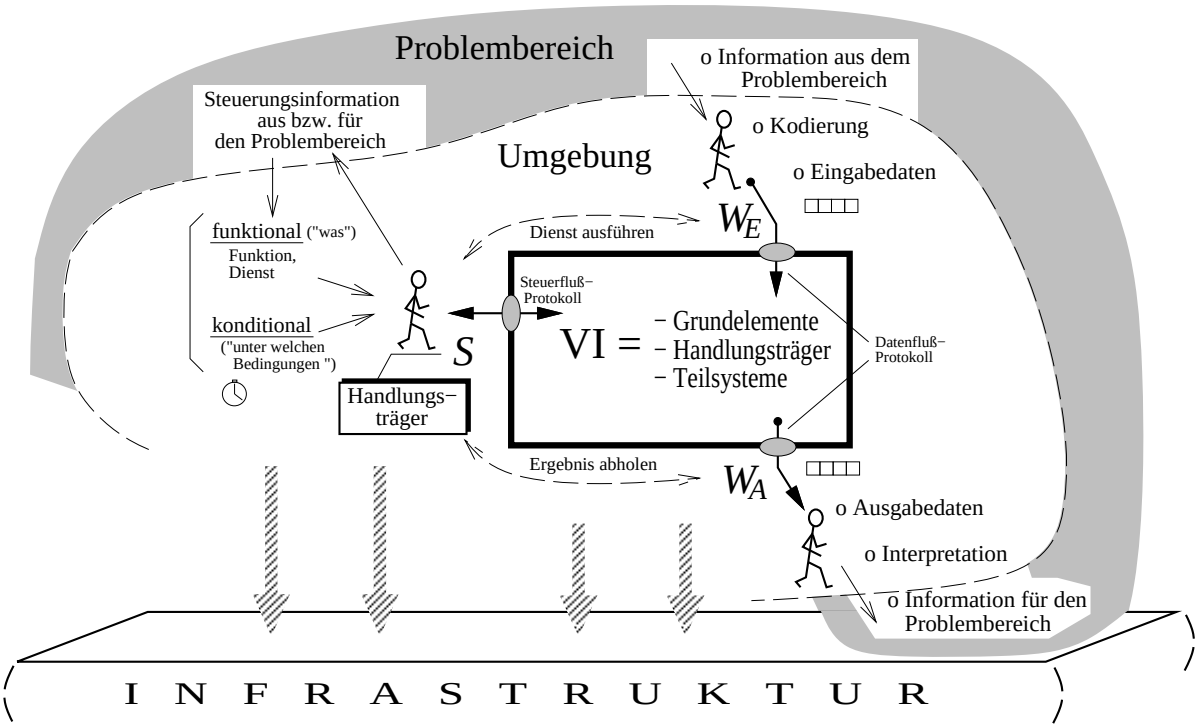


Abb.3.1 Informationsverarbeitung: Problembereich – Umgebung – Infrastruktur

Sowohl für den Datenfluß als auch für den Steuerfluß muß es in der Umgebung einer VI Handlungsträger geben, die Protokolle für Daten und Steuerung mit der VI abwickeln:

- *Protokoll für den Datenfluß* (Handlungsträger der Umgebung $\longleftrightarrow$ VI)
  - VI mit Eingaben versorgen (Handlungsträger: $\mathcal{W}_\mathcal{E} \rightarrow$ VI)
  - Ausgaben von der VI abholen (Handlungsträger: VI $\rightarrow \mathcal{W}_\mathcal{A}$),
- *Protokoll für den Steuerfluß* (Handlungsträger der Umgebung $\longleftrightarrow$ VI)
  - zum Veranlassen der Verarbeitung (Handlungsträger: $\mathcal{S}_\mathcal{E} \rightarrow$ VI)
    - funktional (Auswahl einer Verarbeitungsfunktion)
    - konditional (Ablaufbedingungen)
  - über den Verarbeitungsprozeß (Handlungsträger: VI $\rightarrow \mathcal{S}_\mathcal{A}$)
  - Steuerfluß : $\mathcal{S} = \mathcal{S}_\mathcal{E} \cup \mathcal{S}_\mathcal{A}$.

3.2.1 Dienste, Operationen

Ausgehend von dieser Beobachtung kann eine Verarbeitungsinstanz in einer idealisierten Form dargestellt werden, die ihren eigentlichen Zweck in den Vordergrund stellt, auf Veranlassung von außen Verarbeitungsleistungen in Form von ***Diensten*** auszuführen. Ein Dienst soll hier als eine semantisch zusammengehörige Verarbeitungsleistung durch eine VI verstanden werden, die durch ggf. mehrere elementare ***Operationen*** durch eine VI erbracht wird. Die Trennung zwischen Diensten und Operationen wurde vorgenommen, um zeitlich verzahnte bzw. parallele Dienstausführungen zu erfassen (vgl. Abschnitt 3.2.3). Ein Dienst kann natürlich auch durch eine Operation erbracht werden.

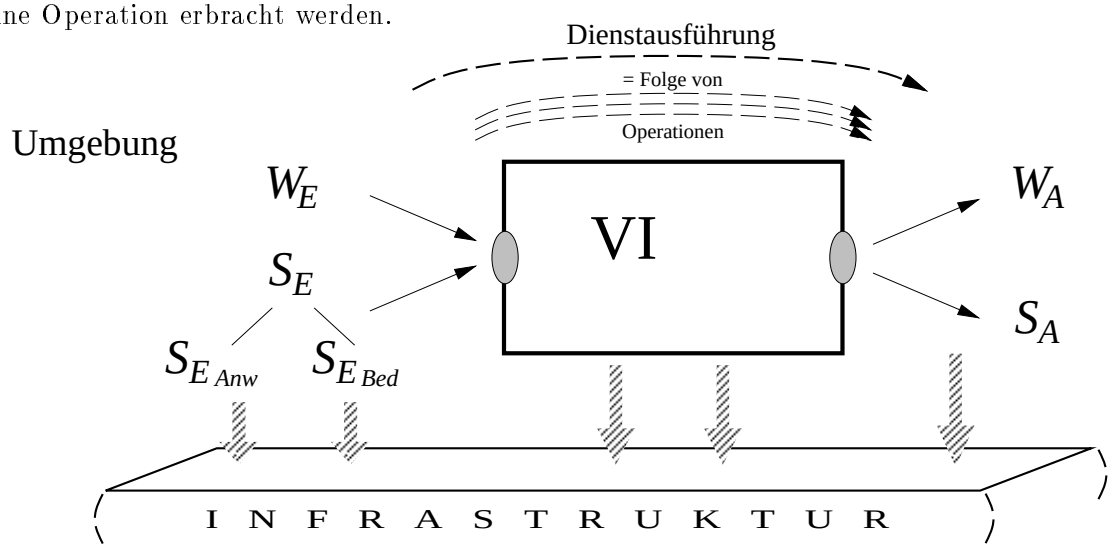


Abb.3.2 Idealisiertes Modell einer Verarbeitungsinstanz

Ein Dienst wird durch eine Folge von Operationen ausgeführt. Aus Sicht von außen erscheint eine Operation somit als ein Verarbeitungs- oder Prozeßschritt, der durch die VI selbständig ausgeführt wird. Die rekursive Auflösung in innere Teilverarbeitungen ist in der Umgebung der VI nicht sichtbar ($\rightarrow$ *Kapselungsprinzip*). Die Ausführung eines Dienstes kann folglich als ein äußerer Verarbeitungsprozeß durch eine Folge von Operationen angesehen werden.

Die Ausführung einer Operation ist eine Abbildung von Eingabewerten $\mathcal{W}_\mathcal{E}$ und Steuerungsinformation $\mathcal{S}_\mathcal{E}$ in eine Menge von Ausgabewerten $\mathcal{W}_\mathcal{A}$ und Steuerungsinformation $\mathcal{S}_\mathcal{A}$ an die Umgebung. Die Menge der Steuerungsinformation $\mathcal{S}_\mathcal{E}$ kann weiter unterteilt werden in eine Menge Anweisungen $\mathcal{S}_{\mathcal{E}_{Anw}}$, die von der VI als Operationen ausgeführt werden können, und in eine Menge von Bedingungen $\mathcal{S}_{\mathcal{E}_{Bed}}$, unter denen Operationen ausgeführt werden. Die Menge möglicher Operationen $\mathcal{OP}$ enthält Abbildungen:

$$\mathcal{OP} : \quad \{ \quad \mathcal{S}_\mathcal{E} \times \mathcal{W}_\mathcal{E} \quad \} \qquad \longrightarrow \quad \{ \quad \mathcal{S}_\mathcal{A} \times \mathcal{W}_\mathcal{A} \quad \} \qquad \text{bzw.}$$

$$\mathcal{OP} : \quad \{ \quad \mathcal{S}_{\mathcal{E}_{Anw}} \times \mathcal{S}_{\mathcal{E}_{Bed}} \times \mathcal{W}_\mathcal{E} \quad \} \quad \longrightarrow \quad \{ \quad \mathcal{S}_\mathcal{A} \times \mathcal{W}_\mathcal{A} \quad \}.$$

Informationsverarbeitung durch eine Instanz wird auf eine Folge von Dienstausführungen $\varphi\mathcal{D}$ zurückgeführt, wobei eine Dienstausführung $\mathcal{D}_i \in \varphi\mathcal{D}$ eine zeitlich und semantisch geordnete Folge von Operationen $\varphi op_i \in \varphi\mathcal{OP}$ ist und einen Verarbeitungsprozeß durch eine Verarbeitungsinstanz widerspiegelt:

$$\varphi\mathcal{D} \quad = \quad \{ \mathcal{D}_i \} \quad = \quad \{ \mathcal{D}_1, \mathcal{D}_2, \ldots \mathcal{D}_n \},$$

$$\mathcal{D}_i \quad = \quad \varphi op_i \quad = \quad \{ op_1^{t_1}, op_2^{t_2}, \ldots op_m^{t_m} \} \qquad \varphi op_i \in \varphi\mathcal{OP}, \; op_j^{t_j} \in \mathcal{OP},$$

$$\text{mit} \quad op_k^{t_k} \prec op_l^{t_l} \quad - \quad \text{"zeitlich vor"}, \quad \text{mit} \quad t_k \prec t_l \quad \text{für} \quad k < l,$$
$$\text{wenn auch gilt "semantisch vor"} \quad \text{für} \quad k < l.$$

Operationen und Dienste sind somit inhärent aktive Verarbeitungsprozesse einer VI. Sie geben die *Dynamik von Verarbeitung* wieder und begründen den Zusammenhang zwischen informationsverarbeitenden Prozessen und der Dienstausführungen durch Instanzen. Informationsverarbeitung kann folglich neben dem Wirken gesteuerter Aktivitäten ($\rightarrow$ *Prozesse, Handlungsträger*) auch über die Dienstausführungen durch Instanzen erfolgen. Damit werden auch Instanzen zu Handlungsträgern im System. Das ist auch deshalb folgerichtig, weil Instanzen im Inneren gesteuerte Aktivitäten enthalten und diese lediglich nach außen kapseln. Die Wirkung dieser Aktivitäten wird im System durch die selbständige Ausführung von Diensten sichtbar. Die *Kapselung innerer Prozesse* und die Herstellung *"höherer" Dienste* durch Instanzen sind wesentliche Strukturmittel für ablaufende Systeme. Mit Instanzen wird das auch in anderen technischen Bereichen angewandte Prinzip autonomer Teilsysteme auf ablaufende Softwaresysteme übertragen.

3.2.2 Protokolle

Ein ***Protokoll*** ist der nach festen Regeln gesteuerte Informationsaustausch zwischen Handlungsträgern in einem System für Koordination oder Kooperation über ein gemeinsames Medium. Ein Protokoll wird über eine uni- oder bidirektionale ***Folge von Protokolleinheiten*** zwischen Handlungsträgern eines Systems abgewickelt. Die Regeln definieren die Struktur, Kodierung und Interpretation der Protokolleinheiten und die Steuerung des Ablaufs.

Der für Operationen notwendige Informationsaustausch einer Instanz mit Handlungsträgern der Umgebung (Steuerungsinformationen $\mathcal{S_E}$, $\mathcal{S_A}$ und Werte zur Verarbeitung $\mathcal{W_E}$, $\mathcal{W_A}$) wird ausschließlich über Protokolle abgewickelt. Die übermittelten Informationen sind für Operationen grob in vier Klassen einteilbar (vgl. auch Abbildung 3.2):

$\mathcal{W_E}$	–	Eingabewerte für Operationen,
$\mathcal{W_A}$	–	Ausgabewerte von Operationen,
$\mathcal{S_E}$	–	Steuerungsinformationen an die VI: $\mathcal{S_E} = \mathcal{S}_{\mathcal{E}_{Anw}} \cup \mathcal{S}_{\mathcal{E}_{Bed}}$ und
$\mathcal{S_A}$	–	Steuerungsinformationen der VI an die Umgebung.

Wie bereits angegeben, läßt sich die Menge $\mathcal{S_E}$ in eine Teilmenge funktionaler Steuerungsinformationen $\mathcal{S}_{\mathcal{E}_{Anw}}$ (Anweisungen) unterteilen, die angeben, "was" mit Verarbeitungsgegenständen geschehen soll, und in eine Teilmenge von Bedingungen $\mathcal{S}_{\mathcal{E}_{Bed}}$, unter denen Operationen ausgeführt werden. Diese Bedingungen können sich auf den zeitlichen Ablauf, das Eintreten von Ereignissen oder Zuständen beziehen. Durch Ablaufbedingungen soll sowohl zeitlich als auch funktional eine korrekte Operationsfolge hergestellt werden.

Die Struktur der genannten vier Mengen läßt sich weiter verfeinern. Den Ausgangspunkt bildet die Menge der Operationen $\mathcal{OP}$. Die Wirkung einer Operation wird durch eine Anweisung definiert, d.h., zu jeder Operation $op_j \in \mathcal{OP}$ gehört genau eine $anw_j \in \mathcal{S}_{\mathcal{E}_{Anw}}$.

$$\mathcal{S}_{\mathcal{E}_{Anw}} = \{ anw_1, anw_2, \ldots anw_n \},$$
$$\forall\ op_j \in \mathcal{OP}: \quad \exists\ anw_j \in \mathcal{S}_{\mathcal{E}_{Anw}} .$$

Neben der Anweisung gehört zu einer Operation $op_j \in \mathcal{OP}$ eine Liste von Eingabewerten (Argumentliste) $\lambda w_{\mathcal{E}_j} \in \mathcal{W_E}$ und eine Liste von Ausgabewerten $\lambda w_{\mathcal{A}_j} \in \mathcal{W_A}$. Beide Listen bestehen aus elementaren Eingabe- bzw. Ausgabewerten (ew_n bzw. aw_m).

$$\lambda w_{\mathcal{E}_j} = \{ ew_1, ew_2, \ldots ew_n \}, \qquad \lambda w_{\mathcal{A}_j} = \{ aw_1, aw_2, \ldots aw_m \},$$
$$\forall\ op_j \in \mathcal{OP}: \quad \exists\ \lambda w_{\mathcal{E}_j} \in \mathcal{W_E} \quad \text{und} \quad \exists\ \lambda w_{\mathcal{A}_j} \in \mathcal{W_A} .$$

Eine Operation wird ausgeführt, wenn alle Bedingungen dafür erfüllt sind [Dij75]. Im Unterschied zu Ein- und Ausgabewerten ($\lambda w_{\mathcal{E}}$, $\lambda w_{\mathcal{A}}$), die ausschließlich über Protokolle mit einer Instanz ausgetauscht werden, sind die Bedingungen nicht an ein solches Protokoll gebunden. Vielmehr können Ablaufbedingungen auch über die Infrastruktur "implizit" wirken, beispielsweise, indem eine Operation erst beim Eintreffen eines äußeren Ereignisses, nach dem Eintreten einer Koordinierungsbedingung oder nach Zuteilung einer Aktivität ausgeführt wird. Andererseits können Ablaufbedingungen auch als Elemente von Protokolleinheiten an die Instanz übermittelt werden, daß etwa eine Operation erst nach Ablauf einer bestimmten Zeit ausgeführt werden soll. Für Protokolle ist nur diese zweite Art von Bedingungen interessant und soll als Liste von Vorbedingungen $\lambda b_{\mathcal{E}_j} \in \mathcal{S}_{\mathcal{E}_{Bed}}$ für eine Operation $op_j \in \mathcal{OP}$ in das Modell aufgenommen werden. Analoges gilt für die Steuerungsinformation $\mathcal{S}_{\mathcal{A}}$ einer Instanz an die Umgebung.

$$\lambda b_{\mathcal{E}_j} = \{ be_1, be_2, \ldots be_n \}, \qquad \lambda b_{\mathcal{A}_j} = \{ ba_1, ba_2, \ldots ba_m \},$$
$$\forall\ op_j \in \mathcal{OP}: \quad \exists\ \lambda b_{\mathcal{E}_j} \in \mathcal{S}_{\mathcal{E}_{Bed}} \quad \text{und} \quad \exists\ \lambda b_{\mathcal{A}_j} \in \mathcal{S}_{\mathcal{A}}\ .$$

Damit sind die relevanten Mengen zur Beschreibung von Protokolleinheiten eingeführt. Die Darstellung läßt sich zusammenfassen. Für jede Operationen $op_j \in \mathcal{OP}$ gibt es:

$\mathcal{S}_{\mathcal{E}_{Anw}}$	$= \{ anw_j \}$	–	eine Anweisung anw_j aus $\mathcal{S}_{\mathcal{E}_{Anw}}$,
$\mathcal{S}_{\mathcal{E}_{Bed}}$	$= \{ \lambda b_{\mathcal{E}_j} \}$	–	eine Liste Ablaufbedingungen aus der Umgebung,
$\mathcal{W}_{\mathcal{E}}$	$= \{ \lambda w_{\mathcal{E}_j} \}$	–	eine Liste von Eingabewerte aus $\mathcal{W}_{\mathcal{E}}$,
$\mathcal{S}_{\mathcal{A}}$	$= \{ \lambda b_{\mathcal{A}_j} \}$	–	eine Liste Steuerungsinformationen an die Umgebung,
$\mathcal{W}_{\mathcal{A}}$	$= \{ \lambda w_{\mathcal{A}_j} \}$	–	eine Liste von Ausgabewerte aus $\mathcal{W}_{\mathcal{A}}$.

Damit können Protokolleinheiten für die Protokolle mit einer Instanz näher beschrieben werden. Es wird zwischen Protokolleinheiten für die Erteilung von Aufträgen aus der Umgebung an eine Verarbeitungsinstanz ($U \rightarrow VI:\ pe_{\mathcal{E}} \in PE_{\mathcal{E}}$) und dem Abholen von Ergebnissen ($VI \rightarrow U:\ pe_{\mathcal{A}} \in PE_{\mathcal{A}}$) unterschieden. In der eingangs angegebenen Abbildungsrelation für Operationen kommen bereits die Protokolleinheiten zum Ausdruck:

$$\mathcal{OP}: \quad \{ \underbrace{\mathcal{S}_{\mathcal{E}_{Anw}} \times \mathcal{S}_{\mathcal{E}_{Bed}} \times \mathcal{W}_{\mathcal{E}}}_{\hookrightarrow pe_{\mathcal{E}}} \} \longrightarrow \{ \underbrace{\mathcal{S}_{\mathcal{A}} \times \mathcal{W}_{\mathcal{A}}}_{\hookrightarrow pe_{\mathcal{A}}} \},$$

$$PE_{\mathcal{E}} \subseteq \{ \mathcal{S}_{\mathcal{E}_{Anw}} \times \mathcal{S}_{\mathcal{E}_{Bed}} \times \mathcal{W}_{\mathcal{E}} \} \text{ und } PE_{\mathcal{A}} \subseteq \{ \mathcal{S}_{\mathcal{A}} \times \mathcal{W}_{\mathcal{A}} \},$$
$$pe_{\mathcal{E}} \in PE_{\mathcal{E}} \text{ und } pe_{\mathcal{A}} \in PE_{\mathcal{A}}\ .$$

Eine Protokolleinheit für die Eingabe ist ein Tripel $pe_{\mathcal{E}_j} = (anw_j, \lambda b_{\mathcal{E}_j}, \lambda w_{\mathcal{E}_j})$ aus $\mathcal{S}_{\mathcal{E}_{Anw}} \times \mathcal{S}_{\mathcal{E}_{Bed}} \times \mathcal{W}_{\mathcal{E}}$. Eine Ausgabeprotokolleinheit einer Instanz an die Umgebung ist dementsprechend ein zugehöriges Tupel $pe_{\mathcal{A}_j} = (\lambda b_{\mathcal{A}_j}, \lambda w_{\mathcal{A}_j})$ aus $\mathcal{S}_{\mathcal{A}} \times \mathcal{W}_{\mathcal{A}}$.

$$pe_{\mathcal{E}_j} = (anw_j, \lambda b_{\mathcal{E}_j}, \lambda w_{\mathcal{E}_j}) \quad \text{mit} \quad anw_j \in \mathcal{S}_{\mathcal{E}_{Anw}},\ \lambda b_{\mathcal{E}_j} \in \mathcal{S}_{\mathcal{E}_{Bed}},\ \lambda w_{\mathcal{E}_j} \in \mathcal{W}_{\mathcal{E}},$$
$$pe_{\mathcal{A}_j} = (\lambda b_{\mathcal{A}_j}, \lambda w_{\mathcal{A}_j}) \quad \text{mit} \quad \lambda b_{\mathcal{A}_j} \in \mathcal{S}_{\mathcal{A}},\ \lambda w_{\mathcal{A}_j} \in \mathcal{W}_{\mathcal{A}}\ .$$

Die tatsächlichen Wege von Protokolleinheiten können verschieden sein. Eine Anweisung mit Eingabeparametern wird normalerweise auf einem anderen Weg als die Informationen über die Ablaufbedingungen zu einer Instanz gelangen. Da aber Wege und Realisierungen von Protokolleinheiten für die Modellbetrachtung hier keine Rolle spielen, sollen Protokolleinheiten verallgemeinert werden, so daß zu jeder Operation op_j genau eine "abstrakte" Eingabeprotokolleinheit $pe_{\mathcal{E}_j} = (anw_j, \lambda b_{\mathcal{E}j}, \lambda w_{\mathcal{E}j})$ mit $pe_{\mathcal{E}_j} \in PE_{\mathcal{E}}$ und genau eine Ausgabeprotokolleinheit $pe_{\mathcal{A}j} = (\lambda b_{\mathcal{A}j}, \lambda w_{\mathcal{A}j})$ mit $pe_{\mathcal{A}j} \in PE_{\mathcal{A}}$ gehört. Für eine Operation op_j werden beide Protokolleinheiten zu pe_j zusammengefaßt op_j: $(pe_{\mathcal{E}_j} \rightarrow pe_{\mathcal{A}j}) = pe_j$.
Protokolleinheiten werden in diesem Modell folglich allgemeiner gefaßt als Protokolleinheiten im Sinne von Nachrichten über Kommunikationskanäle.

$$\begin{array}{lllll} \forall\ op_j: & \exists\ pe_{\mathcal{E}_j} & \text{mit} & (anw_j,\ \lambda b_{\mathcal{E}j},\ \lambda w_{\mathcal{E}j}), & \\ & \text{Ausführung bei:} & & \forall\ be_k \in \lambda b_{\mathcal{E}j}: \quad be_k \rightarrow true, & \\ \forall\ op_j: & \exists\ pe_{\mathcal{A}_j} & \text{mit} & (\lambda b_{\mathcal{A}j},\ \lambda w_{\mathcal{A}j}), & \end{array}$$

$$\Rightarrow\ op_j: \quad (anw_j,\ \lambda b_{\mathcal{E}j},\ \lambda w_{\mathcal{E}j}) \rightarrow (\lambda b_{\mathcal{A}j},\ \lambda w_{\mathcal{A}j}), \qquad pe_{\mathcal{E}_j} \in PE_{\mathcal{E}},\ pe_{\mathcal{A}j} \in PE_{\mathcal{A}},$$

$$op_j: \quad (pe_{\mathcal{E}_j} \rightarrow pe_{\mathcal{A}j}) = pe_j \qquad pe_j \in PE,\ PE \subseteq \{ PE_{\mathcal{E}} \times PE_{\mathcal{A}} \}.$$

• Zusammenfassung zu Diensten, Operationen, Protokollen

Damit können die wichtigsten Aussagen des Modells zu Protokollen zusammengefaßt werden. Der ***Zweck*** einer Verarbeitungsinstanz besteht darin, eine ***Folge von Diensten*** $\varphi\mathcal{D}$ auszuführen, die jeweils in ***Folgen von Operationen*** φop_i aufgelöst werden. Bei der Betrachtung einer Verarbeitungsinstanz von außen ist eine Operation op_j genau ein Verarbeitungs- oder Prozeßschritt, der durch die Abbildung $op_j: (anw_j, \lambda b_{\mathcal{E}j}, \lambda w_{\mathcal{E}j}) \rightarrow (\lambda b_{\mathcal{A}j}, \lambda w_{\mathcal{A}j})$ beschrieben werden kann[2]. Für eine Operation op_j sind ***Protokolleinheiten*** mit äußeren Handlungsträgern auszutauschen, die hier verallgemeinert werden, so daß es nur jeweils eine Ein- und Ausgabeprotokolleinheit pro Operation gibt. Für das äußere Verhalten einer Instanz gilt zusammengefaßt:

$$\begin{array}{lllll} \bullet\ \varphi\mathcal{D} = \{ \mathcal{D}_i \} = \{ \mathcal{D}_1,\ \mathcal{D}_2,\ \ldots\ \mathcal{D}_n \}, & & & & \\ \bullet\ \mathcal{D}_i = \varphi op_i = \{ op_1^{t_1},\ op_2^{t_2},\ \ldots\ op_m^{t_m} \} & & & & \varphi op_i \in \varphi\mathcal{OP},\ op_j^{t_j} \in \mathcal{OP}, \\ \bullet\ op_j: & (anw_j,\ \lambda b_{\mathcal{E}j},\ \lambda w_{\mathcal{E}j}) & \rightarrow & (\lambda b_{\mathcal{A}j},\ \lambda w_{\mathcal{A}j}) & op_j \in \mathcal{OP}, \\ - \ \forall\ op_j: \exists\ pe_{\mathcal{E}_j} & = & (anw_j,\ \lambda b_{\mathcal{E}j},\ \lambda w_{\mathcal{E}j}) & & pe_{\mathcal{E}_j} \in \{ \mathcal{S}_{\mathcal{E}_{Anw}} \times \mathcal{S}_{\mathcal{E}_{Bed}} \times \mathcal{W}_{\mathcal{E}} \}, \\ - \ \forall\ op_j: \exists\ pe_{\mathcal{A}j} & = & (\lambda b_{\mathcal{A}j},\ \lambda w_{\mathcal{A}j}) & & pe_{\mathcal{A}_j} \in \{ \mathcal{S}_{\mathcal{A}} \times \mathcal{W}_{\mathcal{A}} \}. \end{array}$$

Eine Dienstausführung $\mathcal{D}_i$ kann gleichberechtigt zur Folge von Operationen φop_i auch als eine Folge von Protokolleinheiten φpe_i angesehen werden, die durch eine Instanz verarbeitet werden:

$$\begin{array}{lllll} \circ\ \mathcal{D}_i = \varphi pe_i = \{ pe_1^{t_1},\ pe_2^{t_2},\ \ldots\ pe_m^{t_m} \} & & & & \varphi pe_i \in \varphi PE,\ pe_j^{t_j} \in PE, \\ & & & & PE \subseteq \{ PE_{\mathcal{E}} \times PE_{\mathcal{A}} \}, \\ \circ\ \mathcal{D}_i = \{ (pe_{\mathcal{E}}{}_1^{t_1} \rightarrow pe_{\mathcal{A}}{}_1^{t_1+d_1}),\ (pe_{\mathcal{E}}{}_2^{t_2} \rightarrow pe_{\mathcal{A}}{}_2^{t_2+d_2}),\ \ldots\ (pe_{\mathcal{E}}{}_m^{t_m} \rightarrow pe_{\mathcal{A}}{}_m^{t_m+d_m}) \}. & & & & \end{array}$$

[2] Innere Zustandsübergänge werden später betrachtet (vgl. Abschnitt 3.3.1).

3.2.3 Dynamik von Operationsfolgen und Protokollen

In der zeitlichen Abfolge von Operationen bzw. Protokolleinheiten kommt das dynamische Verhalten einer Verarbeitungsinstanz zum Ausdruck. Verarbeitungsinstanzen können je nach Auslegung genau einen oder mehrere Dienste gleichzeitig erbringen. Das kann intern parallel oder durch eine verzahnte Operationsfolge für mehrere Dienste erfolgen. Zur Beschreibung dieses Sachverhalts soll für eine Operation $op_j^{t_j}$ der Operationsfolge eines Dienstes $\mathcal{D}_i$ ein Intervall $\mathcal{T}[\, op_j \,]$ definiert werden, welches den Start der Operation, wenn alle Komponenten aus $pe_{\mathcal{E}_j}$ vorliegen und alle Vorbedingungen erfüllt sind, bis zu deren Ende umfaßt. Man kann Intervalle gleichberechtigt auch auf Protokolleinheiten beziehen.

$$\text{Zeitintervall für } x\text{:} \quad \mathcal{T}[\, x \,] \;=\; [\, ts(x)\,,\;\; te(x) \,] \qquad \begin{array}{l}\text{mit} \quad ts(x) \;-\text{ Startzeit von } x \\ \text{und} \quad te(x) \;-\text{ Endezeit von } x,\end{array}$$

$$\mathcal{T}[\, \mathcal{D}_i \,] \;=\; \mathcal{T}[\; \varphi op_i \;=\; \{\; op_1^{\,t_1},\; op_2^{\,t_2},\; \ldots\; op_m^{\,t_m} \;\}\;] \;=\; [\, ts(\, op_1^{t_1}\,)\,,\, te(\, op_m^{t_m}\,) \,].$$

Instanzen sollen in zwei Kategorien bezüglich ihres dynamischen Verhaltens beim Erbringen von Diensten eingeteilt werden. Ein Instanz wird als ***sequentiell*** bezeichnet, wenn in einem beliebigen Zeitintervall höchstens ein Dienst durch eine Folge von Operationen erbracht werden kann. Können mehrere Dienste gleichzeitig in Arbeit sein, d.h., die zugehörigen Intervalle für mindestens zwei Dienste überlappen sich, soll diese Art von Instanzen als ***parallele*** Instanzen bezeichnet werden. Parallele Instanzen können weiter unterteilt werden. Bei zeitlich *verzahnter Ausführung von Operationen* können in einem Zeitintervall zwar mehrere Dienste in Arbeit sein, die Intervalle von Operationen dieser Dienste überlappen sich aber nicht. Die zweite Art paralleler Instanzen ist dagegen auch zur *parallelen Ausführung von Operationen* in der Lage.

Zur Formalisierung werden zwei Funktionen eingeführt: $\mathcal{V}_{op}(\, t \,)$ und $\mathcal{V}_d(\, t \,)$, die für den Zeitpunkt t angeben, wieviele Operationen bzw. Dienste in einer Instanz bearbeitet werden:

$$\mathcal{V}_{op}(\, t \,) \;=\; \Big|\; \{\; op_j \quad \text{mit} \quad \mathcal{T}[\, op_j \,]\,: \quad ts(op_j) \prec t \quad \wedge \quad te(op_j) \succ t \quad \}^3 \;\Big|,$$

$$\mathcal{V}_d(\, t \,) \;=\; \Big|\; \{\; \mathcal{D}_i \quad \text{mit} \quad \mathcal{T}[\, \mathcal{D}_i \,]\,: \quad ts(\mathcal{D}_i) \prec t \quad \wedge \quad te(\mathcal{D}_i) \succ t \quad \} \;\Big|.$$

Aus der Auflösung von $\mathcal{D}_i = \varphi op_i \;=\; \{\; op_1^{t_1},\; op_2^{t_2},\; \ldots\; op_m^{t_m} \;\}$ folgt:

$$=\; \Big|\; \{\; \mathcal{D}_i \quad \text{mit} \quad \mathcal{T}[\, op_j \,]\,: \quad ts(op_1) \prec t \quad \wedge \quad te(op_m) \succ t \quad \} \;\Big|.$$

Die Klassifizierung von Instanzen läßt sich damit wie folgt zusammenfassen:

- ***sequentielle Instanz***

$$\forall \;\; t_i\,: \quad \mathcal{V}_d(\, t_i) \;\leq 1 \quad \wedge \quad \mathcal{V}_{op}(\, t_i) \;\leq 1,$$

- ***parallele Instanz***

$$\exists \;\; t_i\,: \quad \mathcal{V}_d(\, t_i) \;> 1, \qquad \text{und es gilt bei:}$$

  - *zeitlich verzahnter Operationsfolge:* $\quad \forall \;\; t_i\,: \quad \mathcal{V}_{op}(\, t_i) \;\leq 1,$
  - und bei *parallelen Operationen:* $\quad \exists \;\; t_i\,: \quad \mathcal{V}_{op}(\, t_i) \;> 1.$

Im Fall paralleler Operationen muß die Instanz über mehrere innere Handlungsträger verfügen, die anderen Fällen sind prinzipiell auch mit nur einem Handlungsträger realisierbar.

[3] Operatoren: '$\prec$' – zeitlich vor, '$\succ$' – zeitlich nach

3.2.4 Schnittstellen – Schnittstellendefinition

Eine Instanz ist nach außen von ihrer Umgebung abgegrenzt. Sie muß daher für die Abwicklung von Protokollen mit Handlungsträgern der Umgebung bzw. der Infrastruktur gemeinsame Elemente als ***Schnittstellen*** besitzen. In informationsverarbeitenden Systemen gibt es Schnittstellen global in zwei Ausprägungen, als

- *gemeinsame Speicherelemente* oder als
- *gemeinsame Kommunikationskanäle.*

Schnittstellen bestehen entweder direkt zwischen den beteiligten Partnern einer Interaktion, oder die Interaktion wird über die Infrastruktur vermittelt. In diesem Fall muß es Schnittstellen zur Infrastruktur geben. Unter einer Schnittstellendefinition sollte man eigentlich die Definition der Schnittstelle, d.h. des gemeinsamen Elements zur Abwicklung von Protokollen verstehen. Üblicherweise wird aber etwas anderes darunter verstanden. In der Schnittstellendefinition soll das Verhalten einer Instanz zum Ausdruck kommen. Eine vollständige Schnittstellendefinition müßte dabei umfassen:

- alle ausführbaren Dienste $\mathcal{D}_i$ der Instanz,
- alle für die Ausführung von Diensten benötigten Protokollfolgen φop_i für Operationen, einschließlich der Ordnung ihrer zeitlichen Abfolge,
- alle Protokolleinheiten für alle Operationen $op_j \in \varphi op_i$ mit jeweils
  - einer Anweisung $anw_j \in \mathcal{S}_{\mathcal{E}_{Anw}}$,
  - einer Liste von Vorbedingungen $\lambda b_{\mathcal{E}j} \in \mathcal{S}_{\mathcal{E}_{Bed}}$,
  - einer Liste von Eingabewerten $\lambda w_{\mathcal{E}j} \in \mathcal{W}_{\mathcal{E}}$,
  - einer Liste von Ausgabesteuerungsinformation $\lambda b_{\mathcal{A}j} \in \mathcal{S}_{\mathcal{A}}$,
  - einer Liste von Ausgabewerten $\lambda w_{\mathcal{A}j} \in \mathcal{W}_{\mathcal{A}}$ und
- die Art und den Zugang zu Schnittstellen.

In typischen Programmiersprachen sind nur Teile einer Schnittstellendefinition ausdrückbar: die Menge der Anweisungen $anw_j \in \mathcal{S}_{\mathcal{E}_{Anw}}$ und die Listen für Ein- bzw. Ausgabewerte $\lambda w_{\mathcal{E}j} \in \mathcal{W}_{\mathcal{E}}$ bzw. $\lambda w_{\mathcal{A}j} \in \mathcal{W}_{\mathcal{A}}$. Die Art der Steuerung der Protokolle ist implizit durch das Programmier- oder Ausführungsmodell der Sprache festgelegt, das im typischen Fall strenger Sequentialität trivial ist und deshalb nicht gesondert betrachtet wird. Dieses Modell wird in der Regel über Prozedurrufe umgesetzt. Es ist leicht zu verstehen und für viele Anwendungsszenarien angemessen. Für Betriebssysteme ist es aber wegen paralleler Abläufe unzureichend. Hier spielt die Koordination von Handlungen (Prozessen) eine zentrale Rolle. Fragen der Koordination von Abläufen werden im Modell in die Ablaufbedingungen $\lambda b_{\mathcal{E}j} \in \mathcal{S}_{\mathcal{E}_{Bed}}$ und $\lambda b_{\mathcal{A}j} \in \mathcal{S}_{\mathcal{A}}$ für Operationen (Prozeßschritte) eingeordnet.

In Programmiersprachen können Instanzen durch eine Menge möglicher Operationen mit Anweisungen und zugehörigen Ein- und Ausgabewerten beschrieben werden. Hierfür wurde das Typkonzept eingeführt, welches im Fall abstrakter Typen (ADT) die "vollständige" Definition einer Instanz ausschließlich über die Menge möglicher Operationen erlaubt.
Die ***Schnittstellendefinition*** (= ***Typ***) einer Instanz i soll in diesem Sinne über eine Liste von j *Operationsbeschreibungen* opd_j, d.h. Anweisungen mit zugehörigen Ein- und Ausgabewerten eingeführt werden: $\mathcal{TYP}_i = \lambda opd_i = \{ opd_j \}$ mit $opd_j = (anw_j, \lambda w_{\mathcal{E}j}, \lambda w_{\mathcal{A}j})$.

Der dynamische Aspekt der Abfolge von Protokolleinheiten ist im Falle typischer Programmiersprachen auf strenge Sequentialität beschränkt. *Strenge Sequentialität* bedeutet hier, daß

nicht nur die Abfolge von Operationen für eine Instanz sequentiell erfolgt, sondern daß es in einer Steuerungsebene eines Systems überhaupt nur genau einen Prozeß gibt, der zwischen den beteiligten passiven Funktionseinheiten übergeht. Diesen Fall findet man bei der prozeduralen Ablaufform. Eine noch schärfere Form den Sequentialität liegt vor, wenn es in allen Steuerungsebenen eines Systems nur noch einen Prozeß gibt. Diese Form soll als *absolute Sequentialität* bezeichnet werden. Abläufe in Betriebssystemen sind aber durch parallele Prozesse geprägt. Die Beschreibungen sequentieller Programmiersprachen erfassen daher nur einen Teil dieser Realität.

Es gibt aber auch für den dynamischen Aspekt der Abfolge von Protokolleinheiten Beschreibungsmittel. Die Regeln für den Protokollablauf einer Instanz können beispielsweise über Automaten beschrieben werden. Es kann selbst eine Protokollsteuerung aus formalen Automatenbeschreibungen erzeugt werden [Oels88]. Über Automaten wird jedoch nur das Verhalten intern sequentieller Instanzen in ihrer Reaktion auf synchron empfangene Protokolleinheiten erfaßt. Ein Gesamtsystem mit mehreren Prozessen läßt sich nur (aufwendig) durch gekoppelte Automaten beschreiben. Andere Beschreibungsmittel sind beispielsweise *path expressions* [CH74] oder *Petri-Netze* [CH70]. Auch *Hoare's CSP*-Modell [Hoa78] kann man u.U. dazu zählen.
Dennoch, die Beschreibung des dynamischen Systemverhaltens ist heute kaum praktikabel gelöst.

In der Praxis beschreitet man oft den Weg, Sequentialität auch für parallele Systeme künstlich aufrechtzuerhalten. Ein Beispiel ist *RPC* in verteilten Systemen. Um aber auch hier Parallelität nutzen zu können, sind für viele *RPC*-Systeme verschiedene Ausführungs- oder Aufrufsemantiken im Programmiermodell festgelegt und für Anwendungen wählbar. Ein Beispiel sind die Aufrufsemantiken von *DCE-RPC: asynchron*, *idempotent* oder *maybe* [OSF92].
Das sequentielle Modell wird aber auch bereits verlassen, wenn ein einziger sequentieller Anwendungsprozeß mit äußeren Prozessen zusammenwirkt. Daraus ergeben sich Seiteneffekte über nichtblockierende Dienstaufrufe bei Ein- und Ausgaben, Unterbrechungen oder Ausnahmen.

Das Erfassen möglicher Ablaufszenarien ist eines der schwierigsten Probleme für die Herstellung und Verifikation von Softwaresystemen überhaupt[4]. Das gilt insbesondere auch für Betriebssysteme [Suhr95a].

3.3 Innere Charakteristik

Die äußere Charakteristik beschreibt das Verhalten einer Instanz aus Sicht der jeweils äußeren Umgebung. Es ist durch die Ausführung von Diensten gekennzeichnet. Die innere Charakteristik gibt an, *wie* Dienste erbracht werden, welche inneren Elemente der Steuerung und Verarbeitung daran beteiligt sind und wie diese zusammenwirken.

3.3.1 Zustände und Zustandsübergänge in Instanzen

Informationen werden als Daten kodiert in Speicherelementen (= Grundelemente) von Instanzen aufbewahrt, die dadurch einen Wert erhalten, welcher den ***Zustand*** dieser Elemente ausmacht. Daten oder Zustände können ihrem Zweck nach in verschiedene Kategorien eingeordnet werden:

- $\mathcal{Z}_{\mathcal{W}}$ – Zustände der Werte für Verarbeitung (Verarbeitungsgegenstände aus $\mathcal{W}_{\mathcal{E}}$, $\mathcal{W}_{\mathcal{A}}$),
- $\mathcal{Z}_{\mathcal{S}}$ – Zustände der Werte für Steuerungsinformation (aus $\mathcal{S}_{\mathcal{E}}$ und $\mathcal{S}_{\mathcal{A}}$, Anweisungen),
- $\mathcal{Z}_{\mathcal{V}}$ – Zustände der Abarbeitung einer zustandsbehafteten Steuerung,
- $\mathcal{Z}_{\mathcal{E}}$ – Zustände zur Repräsentation von Softwareelementen in der Infrastruktur.

[4] Auch das Fiasko des *Ariane 5* Starts am 4. Juni 1996 resultierte aus einer fehlerhaften Ausnahmebehandlung, siehe auch `http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html`, Juli 1996.

Der mögliche Zustandsraum einer Instanz wird in der Menge $\mathcal{Z}$ vereint:

$$\mathcal{Z} \quad = \quad \mathcal{Z}_{\mathcal{W}} \quad \cup \quad \mathcal{Z}_{\mathcal{S}} \quad \cup \quad \mathcal{Z}_{\mathcal{V}} \quad \cup \quad \mathcal{Z}_{\mathcal{E}} \; .$$

Operationen führen innerhalb von Instanzen zu ***Zustandsübergängen***, die nach außen nicht sichtbar sind. Zustandsübergänge haben jedoch vielfältige Wirkungen auf die Operationsausführung: $\mathcal{Z}_{\mathcal{W}}$ – für den Steuerfluß, z.B. für bedingte Verzweigungen, $\mathcal{Z}_{\mathcal{S}}$ – für die Modifikation von Anweisungen, $\mathcal{Z}_{\mathcal{V}}$ – etwa für das Rücksetzen der Abarbeitung oder $\mathcal{Z}_{\mathcal{E}}$ – bei Operationen über Meta-Protokolle. Aus der Folge von Operationen resultiert somit auch eine Folge von Zustandsübergängen bei der Ausführung von Operationen durch eine Instanz:

$$\begin{array}{lcl} op_j^{t_j} : & z_k^{t_j} & \longrightarrow \quad z_l^{t_j+d}, \\ \varphi op_i \; : & \varphi\mathcal{Z} \quad = \quad \{ \; z_{\;1}^{\;t_1}, \; z_{\;2}^{\;t_2}, \; \ldots \; z_{\;n}^{\;t_n} \; \} & \qquad z_k^t, \; z_l^t \; \in \; \mathcal{Z} \; . \end{array}$$

3.3.2 Rekursion von Steuerung und Verarbeitung

Steuerung ist die im voraus geplante, gezielte Einflußnahme auf das Wirken aktiver Elemente in einem System, die sich durch Vorgabe einer zeitlichen Abfolge auszuführender Operationen in Abhängigkeit von internen Zuständen ergibt. Steuerung ist damit ein aktiver Verarbeitungsprozeß eines Steuerungselements im System. Die notwendige Steuerungsinformation ist in einem Programm beschrieben. Das Steuerungselement selbst ist ein aktives Element, dessen Verarbeitungsleistung in der Steuerung der Operationsfolge einer Aktivität besteht. Auf diese Weise werden zwei Steuerungsebenen in Systemen in Beziehung gesetzt. Das ist zum einen die Ebene der Steuerung der Aktivität des Steuerungselements und zum anderen, als deren Wirkung, die Ebene der Steuerung der Verarbeitungsaktivität bei der Ausführung von Operationen.
Bei fest verschalteten Steuerungen bildet das "Programm" die Grundlage für die Konstruktion oder Verschaltung. Programmierbare Steuerungen unterscheiden sich von diesen Steuerungen im Grundprinzip lediglich darin, daß Steuerungsinformationen nicht in Form fester Verschaltungen in ein System enthalten, sondern als kodierte Daten in einem ladbaren oder fest programmierten Speicher abgelegt sind. Sie geben durch ein interpretierendes Steuerungselement das Verhalten einer Aktivität vor. Zu den zwei Steuerungsebenen kommt somit noch eine weitere Ebene hinzu, die der Steuerung des Interpretationsvorganges (Steuerschleife: sequentielle Abfolge der Anweisungen im Speicher). Zur Steuerung dieses Ablaufs ist diese dritte Steuerungsebene erforderlich.

Sinnvolle Verarbeitung bedingt Steuerung, und Steuerung ist selbst ein aktiver Verarbeitungsprozeß in einem System, der selbst eine Steuerung erfordert. Diese Rekursion erzeugt mehrere ***Steuerungsebenen*** im System, die sich auch in Steuerungsprotokollen $\mathcal{S}$ und $\mathcal{S}'$ widerspiegeln. Die Betrachtungsweise der Steuerungsebenen läßt sich auch auf Operationen beziehen, daß Operationen op_S der Ebene $\mathcal{S}$ in eine Folge von Operationen φop_S in der Ebene $\mathcal{S}'$ aufgelöst werden, oder umgekehrt, aus einer Folge von Operationen $\varphi op_{S'}$ der Ebene $\mathcal{S}'$ wird eine höherwertige Operation op_S in der Ebene $\mathcal{S}$ erzeugt. Auf diese Weise wird der Zusammenhang zwischen Steuerung (= Verarbeitungsprozeß der Ebene $\mathcal{S}'$) und Verarbeitung (= Wirkung einer auf diese Weise gesteuerten Aktivität) hergestellt. Das gilt rekursiv für beliebige Steuerungsebenen, im Extremfall bis auf die Ebene elementarer Schaltvorgänge in digitalen Grundbausteinen, deren Steuerung dann auf fester Verschaltung beruht und deren Aktivität durch einen Schalttakt initiiert wird. In dieser Ebene erfolgt der endgültige Abbruch dieser rekursiven Auflösung. Für Betriebssysteme wird der Abbruch sinnvollerweise an der Grenze des Softwaresystems zur Hardwareinfrastruktur vollzogen. Für ein Gesamtsystem kann man die Betrachtung aber auch auf die Ebene der Betriebssystem- oder gar der Bedienschnittstelle reduzieren.

Für alle Betrachtungsebenen gilt im Modell aber dasselbe Grundprinzip. Der Zweck eines Systems oder eines Teilsystems ist es, auf Veranlassung von außen Dienste durch eine Folge von Operationen auszuführen. Die veranlassenden Informationen werden über Protokolleinheiten des Steuerungsprotokolls $\mathcal{S}$ an die Instanz übermittelt. In den Eingabeprotokolleinheiten ist je eine Anweisung enthalten, die angibt, welche Operation ausgeführt werden soll, und es sind die Bedingungen enthalten, unter denen die Operation ausgeführt wird.
Im Inneren der Instanz wird die Anweisung der Protokolleinheit durch eine Steuerung ($Strg'$ in Abbildung 3.3) interpretiert und in eine Folge von Teiloperationen $\varphi op_{S'}$ über das Steuerungsprotokoll $\mathcal{S}'$ an eine innere Verarbeitungsinstanz (= rekursive Auflösung von Teilsystemen) oder einen inneren Verarbeitungsprozeß (= elementare Ausführung) aufgelöst. Es besteht kein funktionaler Unterschied zwischen dem Ausführen einer Operationsfolge durch einen inneren Prozeß oder durch ein inneres Teilsystem. Es besteht natürlich der strukturelle Unterschied. In Abbildung 3.3 ist der erste Fall dargestellt. Protokolle werden in innere Protokollfolgen $\varphi pe_{S'}$ aufgelöst, Operationen in innere Folgen $\varphi op_{S'}$, und auch für jede Anweisung $anw_S \in pe_S$ gibt es eine innere Auflösung in eine Folge von $anw'_{S'}$. Es läßt sich zusammenfassen:

- **äußere Steuerung** → **Eingabeprotokolleinheit** pe_S **über** $\mathcal{S}$ → **Operation** op_S **:**
  - Anweisung ("was") – $anw_S \in pe_S$,
  - Bedingungen – $\lambda b_{\mathcal{E}S} \in pe_S$, unter denen die Operation ausgeführt wird,
  - Operationsausführung – wenn: $\forall \; be_k \in \lambda b_{\mathcal{E}S} : \; be_k \rightarrow true$,
- **innere Auflösung** ("wie") → $\varphi pe_{S'}$ **über** $\mathcal{S}'$ → **Operationsfolge** $\varphi op_{S'}$ **:**
  - einer inneren Verarbeitungsinstanz (VI') → Teilsystem,
  - einer unmittelbar gesteuerten inneren Aktivität → innerer Prozeß.

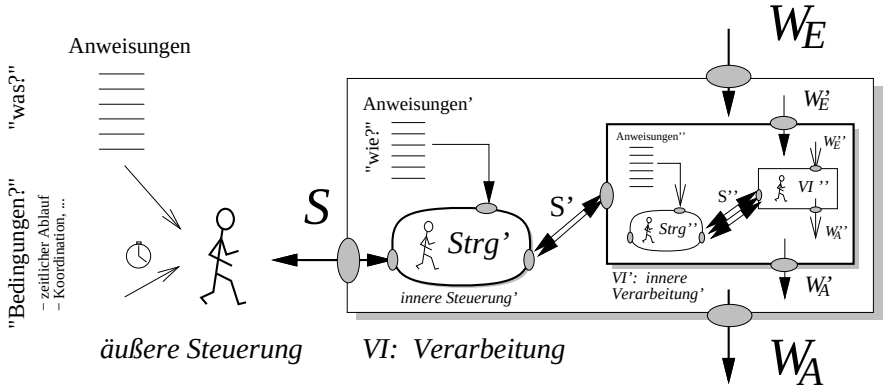


Abb.3.3 Auflösung von Steuerung und Verarbeitung in Verarbeitungsinstanzen

Eine ***Anweisung*** ist die kodierte Form einer Steuerungsinformation zur Interpretation durch ein Steuerungselement. Innerhalb einer Instanz gibt es eine Anweisungsfolge für die Auflösung und Ausführung einer Folge von Teiloperationen $\varphi op_{S'}$ über das Protokoll $\mathcal{S}'$. Für jede Anweisung anw_S muß es eine Anweisungsfolge $\{ \; anw'_1, \; anw'_2, \; \ldots \; anw'_n \; \}$ für die innere Steuerung geben. Der Interpretationsvorgang von Anweisungen bei der Ausführung von Operationen kann über das Eintreten von Bedingungen bzw. über spezielle Operationen von außen beeinflußt werden, was zum Halt bzw. Wiederanlauf, zur Unterbrechung oder zum Fortsetzen der Interpretation von Anweisungen an einer anderen Stelle im Speicher führen kann. Die Steuerung dieser Vorgänge erfolgt über die Infrastruktur, indem beispielsweise Ablaufbedingungen eintreten. Es können auch spezielle Wirkungen durch Operationen über $\mathcal{S}$ veranlaßt werden.

Zwei benachbarte Steuerungsebenen spiegeln sich damit auch in zwei Arten von Anweisungen wider. Anweisungen können ihrer Funktion nach in Klassen eingeteilt werden:

- (1) Anweisungen an eine Instanz aus pe_S über $\mathcal{S}$:
  - anw_α – für Verarbeitung (Verknüpfung, Speicherung, Transport),
  - anw_β – Steuerungsanweisungen (bedingte, unbedingte Verzweigungen u.a.),
  - anw_γ – Ändern, Ergänzen der Folge von anw' in der Instanz von außen,
  - anw_δ – Steuerung des Interpreters über $\mathcal{S}$ (Synchronisation, Stop, Abgabe u.a.),

- (2) innere Anweisungen der Instanz für $\varphi pe_{S'}$ über $\mathcal{S}'$:
  - anw'_α – Steuerung der internen Verarbeitung über $\mathcal{S}'$,
  - anw'_β – Steuerung der Steuerschleife (Steuerungsanweisungen anw_β ausführen),
  - anw'_γ – Ändern, Ergänzen von anw' durch $anw_\gamma{}'$ (selbstmodifizierender Code),
  - anw'_δ – Steuerung des Interpreters, welche über die Infrastruktur initiiert und durch anw' über $\mathcal{S}'$ ausgeführt wird (Unterbrechung, Wiederanlauf u.a.),
  - anw'_ϵ – für die Interaktion zwischen Prozessen bzw. Instanzen.

Es können drei Aspekte für die Steuerung zusammengefaßt werden:

- Steuerung wirkt entweder direkt auf eine Aktivität (→ *elementarer Prozeß*) oder als äußere Steuerung auf eine Instanz, woraufhin im Inneren Operationen ausgeführt werden.
- Steuerung ist ein aktiver Verarbeitungsprozeß von Steuerungselementen, der über mehrere Ebenen in elementarere Steuerungen und Verarbeitungen aufgelöst wird.
- In der Rekursion von Steuerungsebenen stehen je zwei benachbarte Ebenen in Beziehung:
  - Operation op_S → innere Operationsfolge $\varphi op_{S'}$,
  - Protokolleinheit pe_S über $\mathcal{S}$ → Folge innerer Protokolleinheiten $\varphi pe_{S'}$ über $\mathcal{S}'$,
  - Anweisung anw → innere Folge von Anweisungen anw'.

Die Gliederung eines Systems in Steuerungsebenen ist ein wesentliches Mittel zur Schaffung von Struktur und Abstraktion, da bei Instanzen die innere Auflösung von Verarbeitung und Steuerung auf einer übergeordneten Ebene verborgen bleibt.

3.3.3 Aktivität, Handlungsträger, sequentieller Prozeß

Es bleibt die Frage, wer die Steuerschleife im Inneren einer Verarbeitungsinstanz antreibt und auf diese Weise die geschilderten Abläufe und damit Aktivität im System bewirkt.

Aktivität wird in einem digitalen informationsverarbeitenden System erzeugt, wenn von einem Taktgeber oder von äußeren Ereignissen initiiert, Schaltvorgänge in digitalen Logik-Grundbausteinen ausgelöst werden. Hier liegt die unterste Steuerungsebene, in welcher Aktivität inhärent vorhanden ist. Diese Schaltelemente sind die initialen ***Träger von Aktivität***. Während Dienste von außen über verschiedene Steuerungsebenen bis auf das unterste Niveau aufgelöst werden, wird die a priori gegebene Aktivität in der untersten Steuerungsebene nach oben transformiert, um in jeder Steuerungsebene Verarbeitungsschritte für dort angesiedelte Prozesse auszuführen. Auch Elemente der Hardware (CPU, Controller u.a.) basieren auf dem

sinnvollen Zusammenschalten dieser aktiven Grundbausteine und werden damit in der jeweiligen Steuerungsebene selbst zu abgegrenzten aktiven Elementen (→ *Instanzen*). Aus elementareren aktiven und passiven Elementen werden neue, funktional höherwertige aktive Elemente hergestellt. Dieses Prinzip läßt sich auch für Softwareelemente fortsetzen und begründet die Folgerichtigkeit der hier vorgeschlagenen Modellvorstellung, daß für die *strukturellen Wesenszüge* der Architektur eines Systems kein Unterschied zwischen Hardware- und Softwareinstanzen besteht.

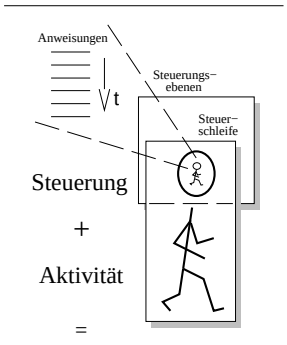


Abb.3.4

Ungesteuerte Aktivität hat für ein technisches System keinen Sinn. Aktivität wird zu einer zielgerichteten, geplanten ***Handlung***, wenn Aktivität gesteuert wird. Das Ziel ist die selbständige Ausführung von Operationen und Diensten. Aktivitätsträger werden durch Steuerung zu ***Handlungsträgern:***

Handlung	=	Aktivität + Steuerung,
Handlungsträger	=	Aktivitätsträger + Steuerung.

Dabei ist unwesentlich, ob Steuerung von Aktivität auf fester Verschaltung oder auf der Ausführung von Programmanweisungen beruht. Man kann deshalb zu der programmbasierten Form verallgemeinern.

Eine Handlung im Sinne einer gesteuerten Verarbeitung wird als sequentielle Handlung verstanden, d.h., zu einem Zeitpunkt kann höchstens eine Operation für eine Handlung in Ausführung sein $\forall\ t_i : \ \mathcal{V}_{op}(\ t_i) \ \leq 1$. In der Literatur werden *Handlungen* oder *"Handlungsfäden"* oft auch als *"thread of control"* bezeichnet. In Betriebssystemen ist dafür seit langem der Begriff ***sequentieller Prozeß*** definiert [Dij68a], obwohl der Prozeßbegriff in Betriebssystemen durchaus auch mit unterschiedlichen Bedeutungen gebraucht wird, etwa in Verbindung mit Adreßräumen [Hof96]. Prozesse in der Hardware oder adreßraum- und maschinenübergreifende Verarbeitungsprozesse werden in dieser Sichtweise nicht erfaßt, es sei denn, man verallgemeinert Adreßräume auf die in 4.3.3 eingeführten Wirkungsbereiche von Prozessen. Deshalb soll hier unter einem sequentiellen Prozeß tatsächlich nur die gesteuerte Abfolge von Verarbeitungsschritten oder Operationen verstanden werden, die als Wirkung einer gesteuerten Aktivität entsteht. In Systemen gibt es typischerweise mehrere gleichzeitige Handlungen, die durch mehrfache sequentielle oder parallele, d.h. gleichzeitig im System existierende Prozesse modelliert werden. Das heißt nicht notwendigerweise, daß zu jedem Zeitpunkt immer alle Handlungen auch tatsächlich voranschreiten müssen. Es ist vorerst keine Aussage über den zeitlichen Ablauf der Prozesse gegeben. Auf dieser Weise läßt sich ein komplexes Gesamtgeschehen in einem System systematisch in n sequentielle Prozesse zerlegen. Hierin liegt ein wichtiges Mittel für die Strukturierung komplexer Abläufe in Systemen [Dij71], welches ebenfalls in der Architektur zum Ausdruck kommen muß.

3.3.4 Passive Grundelemente – Aktive Handlungsträger

In der Einführung dieses Kapitels wurde die globale Systemgliederung für eine Steuerungsebene angegeben: Grundelemente, Handlungsträger und Teilsysteme. Aktive Handlungsträger führen Handlungen (→ *Verarbeitungsprozesse*) über passiven Grundelementen aus, interagieren mit anderen Handlungsträgern oder veranlassen Teilsysteme (→ *Instanzen*), Operationen auszuführen.

Aktivität oder Passivität sind Eigenschaften von Elementen eines Systems, die relativ zu Steuerungsebenen zu sehen ist. Ein Element ist in einer Steuerungsebene *aktiv*, wenn es dort im Sinne eines Prozesses selbständig Handlungen ausführt. Die anderen Elemente sind *passiv*. Dabei kann es sich um *absolute Passivität* von Elementen handeln, oder Passivität wird für ein intern aktives Element in einer höheren Steuerungsebene künstlich hergestellt. Diese Form soll als *relative Passivität* bezeichnet werden. Ein Beispiel dafür ist der Hauptspeicher. Auf der

Ebene des Ablaufs von Anwendungsprozessen ist Hauptspeicher passiv. Betrachtet man aber die Ebene der Speichersteuerung und des Bussystems, dann ist Hauptspeicher ein aktives Element, welches Protokolle über das Bussystem mit anderen Elementen abwickelt und selbständig Adressierungs-, Lese- oder Schreiboperationen ausführt. Verarbeitungsprozesse dieser Art sind aber bewußt in übergeordneten Steuerungsebenen verborgen. Passivität ist deshalb auch ein Mittel zur Abstraktion und zum Vereinfachen komplizierter Systeme.

3.3.5 Umschalten von Aktivität zwischen Handlungsträgern

Ein Aktivitätsträger wird durch Steuerung zu einem Handlungsträger. Es kann in einem System sinnvoll sein, mehrere Handlungen auf Grundlage weniger Aktivitäten oder gar nur einer Aktivität zu erzeugen. Ein Grund kann darin liegen, die komplexe Steuerung einer Handlung in Teilsteuerungen für Teilhandlungen zu zerlegen. Ein anderer Grund ist, Wartezeiten in einer Handlung durch Umschalten auf eine andere Handlungen zu nutzen.
Umschalten bedeutet den zeitlichen Wechsel einer Aktivität zwischen mehreren Steuerungen. Dabei sind Zustände der Steuerung im Aktivitätsträger zu retten, um sie bei einer späteren Rückschaltung wiederherstellen zu können. Die Umschaltung von Handlungen ist selbst eine gesteuerte Handlung, die veranlaßt und ausgeführt werden muß. Eine Umschaltung kann nicht vom selben Handlungsträger vorgenommen werden, da er seine Aktivität dabei verliert. Es muß folglich mindestens einen weiteren Handlungsträger geben, der die Umschaltung vornimmt.

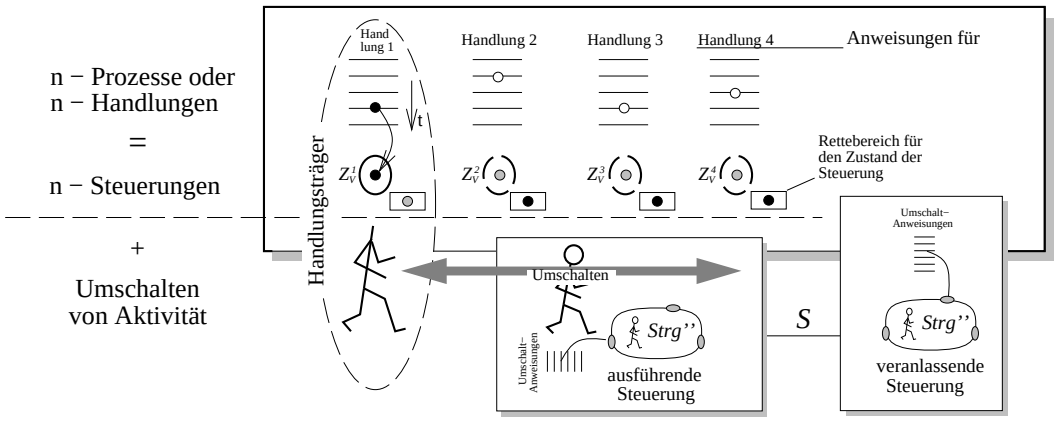


Abb.3.5 Umschalten von Aktivität zwischen Handlungen bzw. Prozessen

Ein Prozeß kann eine Umschaltung selbst veranlassen, indem er anw_δ -Anweisungen ausführt. Gibt es ausschließlich Umschaltung dieser Art, bleibt das Gesamtverhalten des Systems in der Ebene der betrachteten Prozesse determiniert. Anders ist es, wenn Umschaltungen auch durch andere als die betroffenen Prozesse initiiert werden können, etwa durch Prozesse in der Infrastruktur aufgrund von Ereignissen (anw'_δ). Ein Prozeß kann dann prinzipiell an jeder Stelle unterbrochen werden und eine Umschaltung stattfinden (→ ***Entzug, preemption***). Auch die Ziele von Umschaltungen (→ ***Schedule***) müssen gesteuert werden. Im Fall der selbstveranlaßten Umschaltung kann diese Information Teil der anw_δ-Anweisungen sein. Dann sind alle Umschaltungen sowohl bezüglich der Umschaltpunkte als auch der jeweiligen Umschaltziele vorgegeben (→ *Coroutinen*, determiniert). Die Schedule-Steuerung kann aber auch in einem separaten Element zentralisiert sein (→ *Scheduler*), und die Auswahl kann dynamisch erfolgen. Sind die Umschaltpunkte nicht mehr in den Anweisungen der Prozesse programmiert, werden die beteiligten Handlungen auch auf Basis einer Aktivität soweit voneinander entkoppelt, daß unabhängige Handlungen oder Prozesse entstehen (→ *preemptive Prozesse*, nichtdeterminiert).

3.3.6 Strukturierte Ablaufformen

Bei komplexen Steuerungen ist auch die Menge der Anweisungen und Zustandsinformationen kompliziert und umfangreich. Für die Herstellung der zugehörigen Beschreibungen ist daher Struktur notwendig. Das Gebiet der Softwaretechnik befaßt sich mit der Herstellung von strukturierten Beschreibungsinformationen: Anweisungen (Programme), Beschreibungen von Zustandsinformationen (Datenstrukturen) und Anfangsbelegungen. Beschreibungen erfolgen in formalen Sprachen verschiedener Abstraktionsgrade und werden ggf. über mehrere Stufen manuell oder maschinell übersetzt, bis eine Form vorliegt, die von einem informationsverarbeitenden System zur Herstellung der Steuerung für einen Handlungsträger geladen und interpretiert werden kann. Formalen Sprachen ist ein Modell unterlegt, welches dem Softwareentwickler Mittel für Abstraktion und Strukturierung bietet. Das Modell umfaßt drei Aspekte:

- das ***Daten-*** und ***Funktionsmodell*** – Typen, Datenstrukturen und zugehörige Operationen,
- das ***Kompositionsmodell*** – Modularisierung in Programmbausteine und
- das ***Ablaufmodell*** – Verhalten der Aktivität(en) im später ablaufenden System: → *strukturierte Ablaufformen.*

Imperative Programmiersprachen sind die heute für Betriebssysteme relevante Sprachkategorie. Ein (abstrakter) Typ wird durch eine Datenstruktur und zugehörige Operationen definiert. Bei der Umsetzung werden Operationsbeschreibungen in Folgen von Anweisungen aufgelöst. In Sprachen muß auch das arbeitsteilige Entwickeln von Software durch geeignete Möglichkeiten des Kompositionsmodells unterstützt werden.

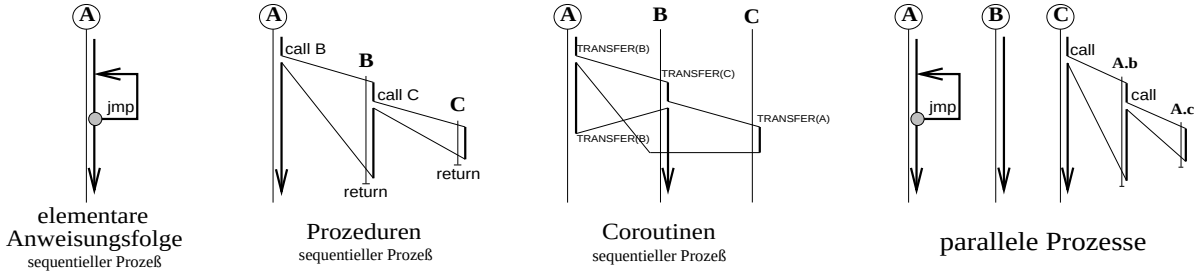


Abb.3.6 Strukturierte Ablaufformen

Das Ablaufmodell gibt das Zusammenspiel und die Struktur von Abläufen bei Operationsausführungen an. Als elementare Formen gibt es die sequentielle Ausführung von Anweisungen in der Reihenfolge der Speicherung und bedingte bzw. unbedingte Verzweigungen. Mit der strukturierten Programmierung wurden diese elementaren Mittel auf hierarchisch zusammensetzbare Programmblöcke und die drei Steuerungsarten: Sequenz, Alternative und Wiederholung verallgemeinert. Gleichzeitig konnten Programmblöcke auch zu Prozeduren bzw. Funktionen zusammengefaßt werden. Prozeduren sind dabei nicht nur ein Element des Kompositionsmodells, sondern bewirken auch eine bestimmte Art oder Struktur des Ablaufs.

Die in Abbildung 3.6 gezeigten Ablaufformen können sich überlagern. Die Basis sind stets elementare Anweisungsfolgen eines sequentiellen Prozesses, denen Prozeduren oder Coroutinen überlagert sein können. Innerhalb einer Coroutine können wiederum Prozeduren ablaufen.
Parallele Prozesse sind voneinander unabhängig. Bei Coroutinen spielt der Aspekt der Kopplung bezüglich des Umschaltpunktes und der Auswahl der Folge-Coroutine eine wichtige Rolle. Entkoppelte Coroutinen sind nur bezüglich des Umschaltpunktes aneinander gekoppelt, die Auswahl der Folge-Coroutine trifft ein zentraler Scheduler. Wirkprinzipien und Realisierungsmöglichkeiten der Ablaufformen für Coroutinen und für preemptive Prozesse (*preemptive Threads*) werden für die Sprache *C* ausführlich in [Grau95b] vorgestellt.

Als die zwei wesentlichen Kategorien lassen sich für das Ablaufmodell angeben:

- ***sequentieller Prozeß*** in den Ausprägungen:
  - elementare Anweisungsfolge (Sequenz) und Verzweigungen,
  - strukturierte Programmierung: hierarchische Programmblöcke,
  - Prozeduren / Funktionen,
  - Coroutinen (gekoppelt bzw. entkoppelt) oder
- ***parallele Prozesse***, d.h. parallel zueinander ausgeführte sequentielle Prozesse, die jeweils mit einer der genannten sequentiellen Ablaufformen überlagert sind.

3.3.7 Interaktion: Signalisierung zur Koordination von Prozessen

Bei der Koordination von Prozessen gibt es in den sonst unabhängigen Prozessen A und B Operationen op_A und op_B, zwischen denen Abhängigkeiten bei der Ausführung bestehen:

$$op_A \quad \longleftarrow \quad op_B : \qquad ts(op_A) \quad \succeq \quad te(op_B) \; .$$

Die Operation op_A in Prozeß A kann erst ausgeführt werden, wenn die Operation op_B aus Prozeß B vollständig ausgeführt ist ($\rightarrow$ ***Synchronisation***). Prozeß A muß gegebenenfalls auf das Eintreten dieser Bedingung warten. Die Abhängigkeit kann auch wechselseitig gelten:

$$op_A \leftarrow op_B \quad \wedge \quad op_B \leftarrow op_A : \qquad ts(op_A) \succeq te(op_B) \quad \wedge \quad ts(op_B) \succeq te(op_A) \; .$$

Ablaufbedingungen für die Synchronisation werden hier mit unter die $\lambda b_{\mathcal{E}} \in \mathcal{S}_{\mathcal{E}_{Bed}}$ gefaßt.

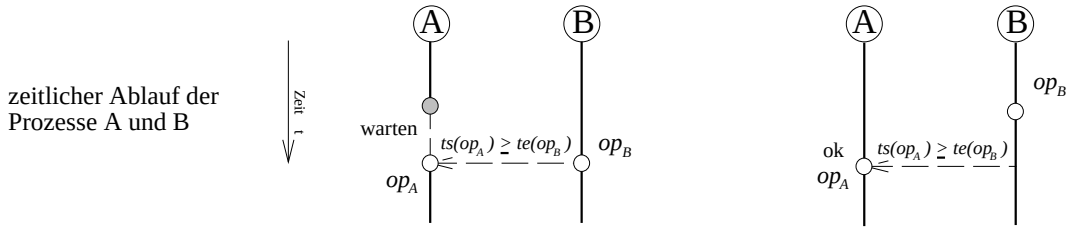


Abb.3.7 Synchronisation des Ablaufs von Prozessen

Für Synchronisation wird Steuerungsinformation über Protokolle übertragen. Die Wirkung dieser Steuerungsinformation auf einen Prozeß soll als ***Signalisierung*** bezeichnet werden. Signalisierung zum Zweck der Synchronisation umfaßt dabei drei Aspekte. Ein *Initiator-Prozeß* in der Umgebung, in der Infrastruktur oder außerhalb des Systems *signalisiert* einem *Rezeptor-Prozeß* das Eintreten einer Bedingung, worauf dieser ggf. *wartet* und dann *reagiert*.
Es gibt dabei sehr unterschiedliche Ausprägungen dieser drei Grundmuster. Das Warten kann in Form eines programmierten Wartezyklus oder durch Abgabe der Aktivität des Prozesses umgesetzt sein. Auch das Reagieren kann verschiedenartig erfolgen, z.B. durch programmiertes Verzweigen anhand einer Bedingung im Wartezyklus oder durch Unterbrechen der regulären Abarbeitung und dem Fortsetzen an einer anderen Stelle im Programm. Signalisierung kann auch zum Wiederzuteilen der Aktivität eines Prozesse führen, wenn dieser auf ein Signal gewartet hat. Signalisierung ist das gerichtete Zusammenwirken ($\rightarrow$ *Interaktion*) zwischen zwei Prozessen: Prozeß A (Rezeptor) $\leftarrow$ Prozeß B (Initiator). Signalisierung ist dabei Interaktion als reine Ablaufsteuerung zum Zweck der Koordination zwischen Prozessen.
Die Schnittstellen, über die Protokolle zur Übermittlung der Signalisierungsinformation abgewickelt werden, können unterschiedlich ausgelegt sein. Die beiden Grundformen sind:

- (a) ***gemeinsamer Speicher*** oder ein
- (b) ***gemeinsamer Kommunikationskanal***.

Für beide Formen sind Algorithmen bekannt: [Dij65, Knu66] für (a) und [Lamp78, Han78] für (b).

In Abbildung 3.8 sind zwei Beispiele für Signalisierungsszenarien angegeben. Im linken Teil erfolgt die Übermittlung der Signalisierungsinformation über einen gemeinsamen Speicher oder einen gemeinsamen Kanal. Der Rezeptor prüft periodisch den Wert dieses Elements und verzweigt anhand der Prüfbedingung, um auf ein Signal zu reagieren ($\rightarrow$ *polling*). Die Steuerungen (*Strg*) für dieses Verhalten sind als Anweisungsfolgen mit den Anweisungsfolgen der eigentlichen Verarbeitung in beiden Prozessen vermischt, d.h., der Prüfzyklus in Prozeß A muß in den Anweisungen dieses Prozesses programmiert sein. Diese Vermischung führt zu zusätzlicher Kompliziertheit des Programms, vor allem, wenn mehrere Koordinierungsbedingungen einzuhalten sein. Andererseits ist kein gesondertes Steuerungselement dafür im System notwendig. Dennoch, die Vermischung semantisch verschiedener Dinge (Koordination und Verarbeitung) in einer Anweisungsfolge ist aus konzeptioneller und struktureller Sicht nicht wünschenswert. Man findet sie deshalb nur in einfachen Steuerungsszenarien, z.B. für die Koordination von Gerätetreiberprozessen mit Controller-Prozessen. Das Fehlen einer separaten Steuerung verhindert auch, das Warten durch Abgabe oder Abschalten der Aktivität zu realisieren, so daß nur programmierte Wartezyklen bleiben, die Verarbeitungsleistung vergeuden, wenn es mehr arbeitsbereite Prozesse als Prozessoren gibt.

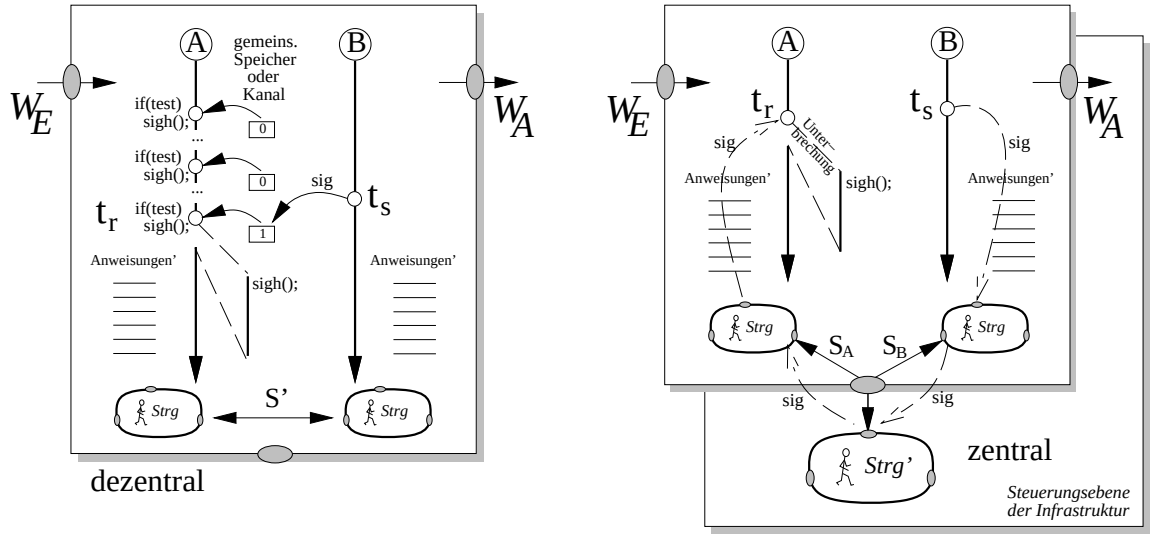


Abb.3.8 Beispiele für Signalisierungsszenarien

In [Kal90] wird wegen des Fehlens eines zentralen Steuerungselements diese Ausprägung als ***dezentrale Steuerung*** bezeichnet. Im Fall einer ***zentralen Steuerung*** ist ein solches Element für Koordinationsaufgaben zwischen Prozessen im System vorhanden. Damit können auch die Anweisungsfolgen der Prozesse von Aufgaben der Ablaufsteuerung entlastet werden. Eine Koordination muß nur noch veranlaßt, aber nicht mehr von den Prozessen selbst ausgeführt werden. Das zentrale Steuerungselement ist sinnvollerweise in einer anderen Steuerungsebene im System plaziert, da zum einen die Ablaufsteuerung semantisch nichts direkt mit der Verarbeitungsleistung der Prozesse zu tun hat und zum anderen die Ablaufsteuerung aufgrund der Ab- bzw. Umschaltung von Aktivität in der unterliegenden Schicht enthalten sein muß. Aus der Sicht der Prozesse befindet sich die Ablaufsteuerung daher in der Infrastruktur und wirkt für die Prozesse implizit. In Abbildung 3.8 veranlaßt Prozeß B ein Signal, welches zur Ablaufsteuerung $Strg'$ in der Infrastruktur gelangt und dort eine Unterbrechung des Prozesses A bewirkt. Daraufhin wird die Abarbeitung auf ein vorher für diesen Fall eingerichtetes Programmstück `sigh` umgelenkt. Es sind auch andere Szenarien denkbar, daß z.B. Prozeß A auf ein Signal durch Abgabe seiner Aktivität wartet (*sleep*) und beim Eintreffen des Signals wieder "aufwacht" (*wakeup*). Prozeß A veranlaßt in diesem Fall sein Warten, die Umsetzung muß die Ablaufsteuerung in der Infrastruktur vornehmen. Dafür müssen in Prozeß A Anweisungen anw_δ ausgeführt werden, durch die

der Prozeß die Ablaufsteuerung informiert und in deren Wirkung der genannte Effekt für den Prozeß eintritt. Analog verhält es sich beim Senden eines Signals durch einen Initiatorprozeß, das ebenfalls über die Ablaufsteuerung in der Infrastruktur an den Empfänger vermittelt wird.

Es läßt sich eine große Bandbreite heute angewandter Prinzipien zur Ablaufsteuerung zwischen Prozessen auf Signalisierung mit den drei Aspekten: *Signalisieren, Warten* und *Reagieren* zurückführen. Dabei ergeben sich vier Grundmuster für Synchronisationsprotokolle zwischen einem Initiator- und einem Rezeptor-Prozeß [Wett93], die in ihrer Kombination 16 Varianten ergeben:

I \ R	synchron	asynchron	versuchend	umlenkend
synchron				
asynchron				
versuchend				
umlenkend				

Abb.3.9 Synchronisationsprotokolle

- ***synchron:*** *Initiator* signalisiert und wartet auf Reaktion des Rezeptors; *Rezeptor* wartet auf Signal;
- ***asynchron:*** *Initiator* signalisiert und setzt Arbeit fort; *Rezeptor* testet auf Signal und fährt fort;
- ***versuchend:*** *Initiator* signalisiert, setzt seine Arbeit fort und testet periodisch auf das Eintreffen der Reaktion des Rezeptors; *Rezeptor* testet periodisch auf eine Signalisierung;
- ***umlenkend:*** *Initiator* signalisiert, setzt seine Arbeit fort und wird bei einer Reaktion des Rezeptors unterbrochen, d.h., die Steuerung wird an eine andere Stelle im Programm umgelenkt; der *Rezeptor* wird beim Eintreffen eines Signals unterbrochen und kann in der Behandlung der Unterbrechung auf das Signal reagieren.

Weitere Merkmale bestehen darin, ob bzw. wieviele Signale zwischengespeichert werden können, wenn Signale mehrfach eintreffen und der Rezeptor zwischenzeitlich nicht reagiert.
Freiheitsgrade entstehen auch bei Mehrfach- oder Gruppensignalisierung, wenn mehr als zwei Prozesse beteiligt sind. Für Gruppen von Prozessen ($\mathcal{G}_I$ – Initiatoren, $\mathcal{G}_R$ – Rezeptoren) gibt es vier Relationen für das Senden bzw. Empfangen:

$$\mathcal{G}_I \quad \rightarrow \quad \mathcal{G}_R \qquad \text{mit:} \qquad |\,\mathcal{G}_I\,| \;:\; |\,\mathcal{G}_R\,| \qquad \longleftrightarrow \qquad \begin{cases} (a) & 1 : 1 \\ (b) & 1 : m \\ (c) & n : 1 \\ (d) & n : m\ . \end{cases}$$

Der Fall (a) entspricht dem oben gezeigten Fall einer Signalisierung zwischen zwei Prozessen. Im Fall (b) signalisiert ein Prozeß an eine Gruppe anderer Prozesse (*Multicast-Signalisierung*), und bei Fall (c) erwartet ein Prozeß Signale von mehreren Prozessen (*Multigather-Signalisierung*). Hier kann man unterscheiden, nach welcher Funktion Signalisierungen verschiedener Prozesse verknüpft werden. Von dieser Verknüpfung hängt es ab, ob bereits das Signal eines Initiators für die Signalisierung ausreicht (*OR*-Verknüpfung) oder ob dafür Signale von allen Prozessen aus $\mathcal{G}_I$ vorliegen müssen (*AND*-Verknüpfung). Andere booleschen Verknüpfungen sind auch denkbar, aber in der Regel nicht sinnvoll. Der Fall (c) vereint die Fälle (a) – (c).

3.3.8 Interaktion: Kommunikation

Bei reiner Signalisierung wird lediglich Steuerungsinformation übertragen, die sich auf den Ablauf von Prozessen auswirkt. Kommen Verarbeitungsdaten hinzu und besteht zwischen den beteiligten Prozessen ein Protokoll für den Austausch dieser Daten, resultiert ***Kommunikation*** zwischen diese Prozessen. Signalisierung widerspiegelt den Aspekt der Ablaufsteuerung und Kommunikation den Aspekt des Datenaustauschs bei Kooperation von Prozesse oder Instanzen. Kommunikation unterliegt für die Ablaufsteuerung prinzipiell denselben Grundmustern wie Signalisierung. Signalisierung und Kommunikation sind die zwei wesentlichen Elemente für das Zusammenwirken oder die ***Interaktion*** von Prozessen oder Instanzen.

3.4 Kombination und Aggregation von Verarbeitungsinstanzen

Es kann mehrere Verarbeitungsinstanzen in einer Steuerungsebene geben, die gemeinsam eine Verarbeitungsleistung für das Gesamtsystem erbringen und deren Zusammenwirken zu steuern ist. Das bedeutet insbesondere die Auflösung von Operationen op_S in eine Folge von Operationen $\varphi op_{S'}$, die auf einzelne Verarbeitungsinstanzen aufgeteilt werden ($\rightarrow$ *dispatching*).

Auch hier kann man eine Differenzierung in eine zentrale bzw. dezentrale Variante angeben. Bei zentraler Steuerung existiert eine separate Steuerungsinstanz, die das Steuerungsprotokoll $\mathcal{S}$ mit der Umgebung abwickelt und das Zusammenwirken der VI' über die Steuerungsprotokolle $\mathcal{S}'_i$ koordiniert. Das Verhalten dieser Steuerung $Strg'$ ist durch anw' zentral beschrieben. Die inneren VI'_i und die Aufteilung der Operationen sind nach außen nicht sichtbar.

Bei dezentraler Steuerung wird die Koordination der VI'_i von den einzelnen Steuerungen der VI'_i selbst erbracht, d.h., die Steuerungsprotokolle $\mathcal{S}'_i$ bestehen untereinander und sind in Anweisungsfolgen anw'' der $Strg''$ in den VI'_i beschrieben. Man findet hier dieselbe Vermischung von Anweisungen zur Steuerung der Verarbeitung und zur Steuerung der Koordination, wie sie im vorangegangenen Abschnitt für die dezentrale Signalisierung angegeben wurde. Jede VI'_i hat ein Steuerungsprotokoll $\mathcal{S}_i$ zur Umgebung, wo sie aus diesem Grund auch bekannt sein muß. Insofern erscheint die umschließende Instanz nach außen als eine Gruppierung mehrere VI'_i.

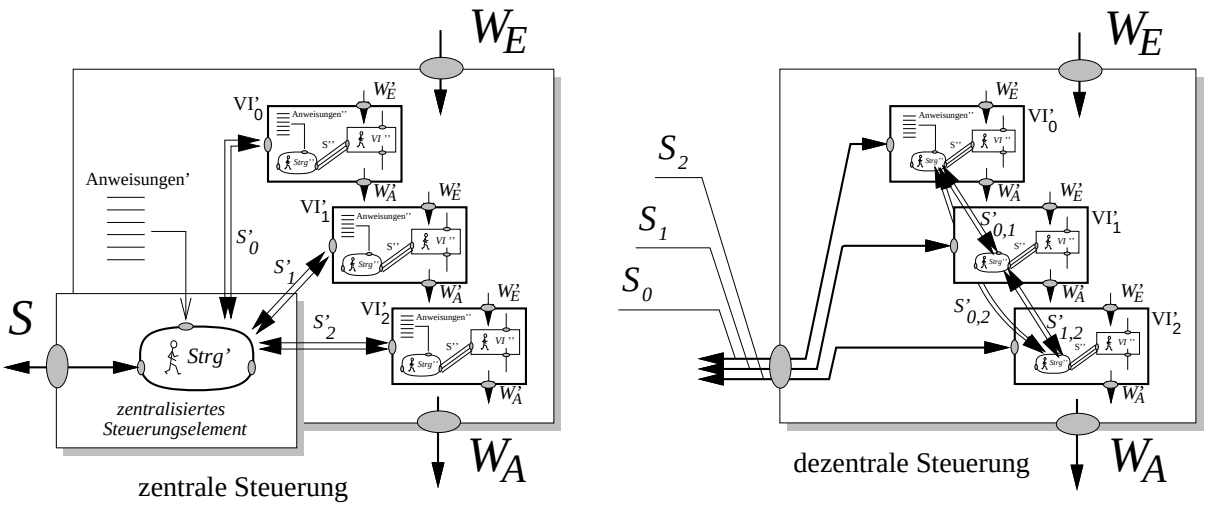


Abb.3.10 Kombination von Instanzen mit zentraler bzw. dezentraler Steuerung

VI'_i besitzen jeweils eigene Aktivität(en) und sind damit zur selbständigen Ausführung von Operationen bzw. Diensten in der Lage. Das bedeutet gleichzeitig, daß mehrere VI'_i parallel und unabhängig zueinander arbeiten. Für die Organisationsform dieser Parallelität gibt es ebenfalls verschiedene Varianten, beispielsweise zur parallelen Ausführung einzelner $op_{S'}$. Man kann VI'_i auch als Fließband- oder Pipeline hintereinanderschalten, indem Ausgaben von VI'_i als Eingaben an VI'_{i+1} weitergereicht werden. Auch Mischformen sind möglich.

Je nach Ausprägung des Systems können neue VI'_i als Softwareelemente dynamisch durch die Infrastruktur erzeugt werden. Es kann im Extremfall sogar sinnvoll sein, für jede Operation $op_{S'}$ eine Instanz VI'_i zu erzeugen und nach Ausführung dieser Operation die Instanz wieder zu beseitigen. Für die Implementierung des *CHEOPS*-Kerns wird ein solches Verfahren zur Behandlung von Unterbrechungen angewandt, da es sich in dieser Ebene mit vertretbarem Laufzeitaufwand umsetzen läßt und sich daraus einige vorteilhafte Eigenschaften bei der Unterbrechungsbehandlung ergeben, die mit der traditionellen Methode nicht erreicht werden können. Der Entwurf, die Implementierung und die Bewertung dieses Verfahrens werden in Kapitel 5 behandelt.

Bei der Kombination von Verarbeitungsinstanzen befinden sich die beteiligten Instanzen in der gleichen Schicht. Sie besitzen dieselbe Infrastruktur und damit dieselben infrastrukturellen Eigenschaften, und sie bilden nach außen ein geschlossenes System. Das umschließende System kann als eine Verarbeitungsinstanz VI angesehen werden, welche die VI'_i enthält. Daraus leitet sich die prinzipielle Äquivalenz von Kombination und Aggregation ab, so daß im weiteren die Kombination von Instanzen durch die Aggregation aufgehoben werden kann.

3.5 Kombination – Aggregation – Infrastruktur

Für die Gliederung von Systemen gibt es drei wesentliche Prinzipien, um abgeschlossene Teilsysteme oder Verarbeitungsinstanzen in Beziehung zu setzen:

- ***Kombination*** – Instanzen befinden sich auf gleichem Niveau nebeneinander,
  und
  Aggregation – Instanzen enthalten innere Verarbeitungsinstanzen,
- ***Infrastruktur*** – Instanzen bilden die Infrastruktur für übergeordnete Instanzen.

Mit Aggregation und Infrastruktur werden hierarchische Ebenen in eine Systemarchitektur eingeführt, die wesentliche Mittel für Abstraktion und zur Schaffung von Übersicht sind. Beide Hierarchien sind orthogonal zueinander.

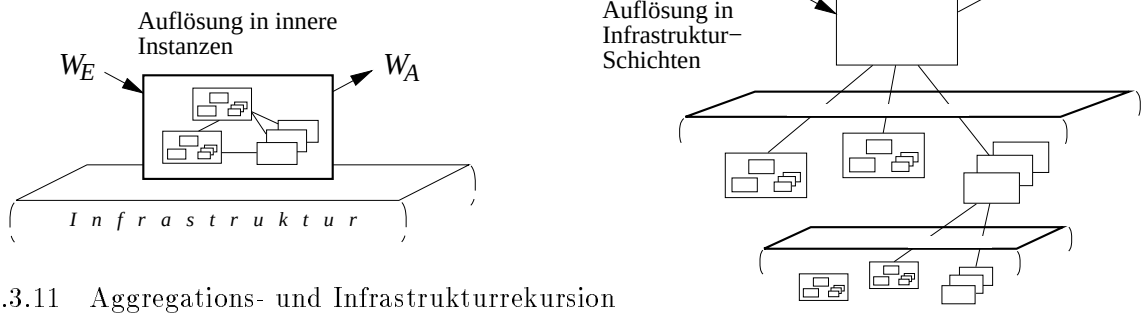


Abb.3.11 Aggregations- und Infrastrukturrekursion

Die Kombination und Aggregation von Instanzen wurde im vorangegangenen Abschnitt behandelt. Es bleibt die Frage, an welchen Kriterien man die Abgrenzung von Instanzen einer Infrastruktur von denen eines übergeordneten Instanzbereichs festmacht.
Die Abgrenzung von Hardware- und Softwareinstanzen ist a priori gegeben und intuitiv verständlich. Hier geht es aber darum, ein ablaufendes System auch in Schichten von Softwareinfrastrukturen zu gliedern. In der Praxis findet man das in Form der Schicht Betriebssystem seit langem. Das Ziel hier ist, dieses Prinzip zu verallgemeinern und die Frage zu beantworten, wodurch Schichten von Softwareinstanzen voneinander abgegrenzt werden können.

Eine Infrastruktur stellt die Existenz-, Funktions- und Interaktionsgrundlagen für Instanzen einer übergeordneten Schicht bereit. Eine Besonderheit von Softwaresystemen ist, daß Verarbeitungsinstanzen als Ergebnis von Verarbeitungsprozessen anderer Instanzen im System geschaffen werden. Auch die Eigenschaften der Existenz-, Funktions- und Interaktionsgrundlagen resultieren aus Verarbeitungsprozessen einer anderen Gruppe von Instanzen im System.
Damit ist eine Aufgabenteilung zwischen mindestens zwei Gruppen von Instanzen in einem System gegeben. Eine Gruppe von Instanzen stellt durch Verarbeitungsprozesse andere Instanzen und deren Ablaufeigenschaften her. Daraus folgt auch eine existentielle Abhängigkeit der einen

Gruppe von der anderen. Aber das wesentliche Unterscheidungsmerkmal ist, daß Verarbeitungsprozesse in der Infrastruktur beliebig gestaltbar sind und auf diese Weise auch neue Ablaufeigenschaften für übergeordnete Instanzen herstellbar sind. Über ***neuartige Ablaufeigenschaften*** soll hier auch die ***Abgrenzung von Infrastrukturschichten*** vorgenommen werden. Der Sinn dieses Abgrenzungskriteriums begründet sich auch daraus, daß innerhalb der Infrastruktur die hergestellten Ablaufeigenschaften nicht wirksam sind, da die zugehörigen Verarbeitungsprozesse nicht unter denselben Eigenschaften ablaufen können, wie sie als Wirkung dieser Prozesse erst entstehen, d.h., es herrscht innerhalb der Infrastruktur ein elementareres Ausführungsmodell vor. Daraus leitet sich auch der notwendige Abbau der Ausprägung des Ausführungsmodells hin zu elementareren Niveaus ab, wie es auch im Begriff ***Rekursion von Infrastrukturschichten*** zum Ausdruck kommt. Der Abbruch der Rekursion erfolgt für Softwareinfrastrukturen an der Hardwareinfrastruktur.

In der Herstellung spezifischer, für einen Zweck besser geeigneter Ablaufeigenschaften liegt der Sinn für die Schaffung von Softwareinfrastrukturen. Das Betriebssystem wurde als Beispiel schon genannt. Aber auch innerhalb des Betriebssystems oder in einer anderen Softwareschicht kann dieses Vorgehen in einer feineren Auflösung sinnvoll sein, um spezielle, an einen Einsatzfall angepaßte Infrastrukturen softwareseitig herzustellen.

Auch aus softwaretechnologischer Sicht haben Softwareinfrastrukturen Sinn, wenn es gelingt, durch Konzentration gemeinsamer Steuerungs- und Verwaltungskomponenten in der Infrastruktur, Komplexität aus Instanzen in die Infrastruktur zu verlagern, für die dann im Anwendungsbereich auch keine Software hergestellt werden muß. In Abschnitt 3.3.7 wurde das Beispiel der Anordnung der Koordinierungssteuerung für Prozesse diskutiert, die im dezentralen Fall mit den Anweisungen des Prozesses vermischt beschrieben ist oder die im zentralen Fall in der Infrastruktur enthalten ist. Damit werden die Programme der Prozesse und die Abläufe in der übergeordneten Schicht vereinfacht. Das bezieht sich nicht nur auf typische Betriebssystemdienste, sondern kann auf beliebig gestaltete Infrastrukturen für verschiedenste Anwendungen verallgemeinert werden.
Softwareinfrastrukturen sind damit auch ein wichtiges *softwaretechnologisches Element*, dessen Bedeutung bei der Herstellung von Systemen noch gar nicht entsprechend erkannt und angewandt wird, weil auch die Voraussetzungen dafür heute kaum gegeben sind.

Den Vorteilen stehen Nachteile gegenüber, die vor allem in einem höheren Verwaltungsaufwand liegen, der offensichtlich ist, denn Softwareinstanzen sind Ergebnis von Verarbeitungsprozessen in der Infrastruktur, deren Zweck die Herstellung von Instanzen und Ablaufeigenschaften, aber nicht die Erbringung der eigentlich vorgesehenen Verarbeitungsleistung des Systems ist. Dieser Mehraufwand wird aber bei heutigen Betriebssystemen auch akzeptiert. Es ist unstrittig, daß die Vorteile überwiegen. Das Argument des Effizienznachteils von Softwareinfrastrukturen ist damit relativ zu bewerten. In anpaßbaren Betriebssystemen geht es vor allem auch darum, den Verarbeitungsmehraufwand durch bestmögliche Anpassung an die Anwendung zu reduzieren.

Aus diesen Zusammenhängen folgt, daß Schichten von Softwareinfrastrukturen:

- nicht nur semantisch, sondern durch unterschiedliche Ausführungseigenschaften auch *technisch abgegrenzt* sind,
- mit Notwendigkeit *rekursiv* hin zu elementareren Ausführungsniveaus abgebaut werden, und die Rekursion an der Hardwareinfrastruktur abbricht,
- ein wichtiges *softwaretechnologisches Element* für die Konstruktion übersichtlicher Systeme sind, und daß
- der *Mehraufwand* für Softwareinfrastrukturen *relativ* zu bewerten ist und dessen Gewichtung von der *Zielstellung* abhängt, die bei der Schaffung eines Systems verfolgt wird.

3.6 Zusammenfassung und Fazit

Der Zweck der Diskussion zum Modellierungsgegenstand Betriebssystem bestand darin, generelle Strukturmerkmale, Funktionen und Abläufe in Systemen, speziell in Betriebssystemen, zu analysieren, um im folgenden Kapitel den Ausgangspunkt und die Begründung für die vorgestellte Architektur daran zu orientieren.

Fragen der strukturellen Gliederung eines Systems sind eng mit dynamischen Eigenschaften des ablaufenden Systems verbunden. Informationsverarbeitung wurde als Teilbereich allgemeiner Verarbeitungsprozesse eingeordnet, und es wurde die enge Verflechtung zwischen Verarbeitung, Steuerung und Herstellung herausgearbeitet. Als generelles Strukturmuster wurde die Gliederung eines Systems in passive Grundelemente, aktive Handlungsträger und die rekursive Auflösung in Teilsysteme angegeben. Handlungsträger wurden als gesteuerte Aktivitäten oder *Prozesse* erklärt und Teilsysteme als *Verarbeitungsinstanzen (VI)* erfaßt, deren Zweck es ist, Verarbeitungsleistungen in Form von Diensten zu erbringen. Verarbeitungsleistungen kann ausschließlich durch gesteuerte, aktive Elemente erbracht werden. Im Modell hier sind das elementare Prozesse und selbständig agierende Verarbeitungsinstanzen (→ *Teilsysteme*).

Das *äußere Verhalten* von Instanzen wird durch Dienste, Operationen, Protokolle und Schnittstellen beschrieben. Die *innere Charakteristik* erfaßt Zustände und Zustandsübergänge bei der Ausführung von Operationen. Es wurde gezeigt, wie mehrere Handlungen oder Prozesse durch Umschalten von Aktivität zwischen Steuerungen hergestellt werden können, und es wurden strukturierte Ablaufformen für Prozesse vorgestellt.

Für das Zusammenwirken oder die *Interaktion* wurden die zwei grundlegenden Prinzipien zusammengefaßt: Signalisierung zum Zweck der Koordination von Prozessen oder Instanzen und Kommunikation für Zusammenarbeit oder Kooperation.

Neben der elementaren Gliederung eines Systems in Grundelemente, Handlungsträger und Teilsysteme wurden Teilsysteme oder Instanzen eines Systems in zwei Varianten zueinander in Beziehung gesetzt:

- ***Aggregation – Kombination***,
- ***Infrastruktur***.

Der Schwerpunkt der Modellbetrachtung wurde auf das dynamische Verhalten eines ablaufenden Systems, seiner Elemente und Beziehungen gelegt. Darin liegt ein fundamentaler Unterschied zu einer softwaretechnologisch dominierten Herangehensweise, statische Aspekte der Struktur von Programmen und Daten und Aufrufbeziehungen in den Vordergrund zu stellen, wie es insbesondere mit dem Hintergrund von Objektorientierung in vielen Forschungsarbeiten zu Betriebssystemen anzutreffen ist. Betriebssysteme sind in erster Linie dynamische Systeme, bei denen es um Prozesse, Ablaufsteuerung und Ressourcenverwaltung geht.
Fragen der Programmstruktur sind ebenfalls wichtig, sollten aber nicht die eigentlichen Wesenszüge von Betriebssystemen überlagern. An dieser Stelle haben heute manche objektorientierte Ansätze in Betriebssystemen Defizite.

Die Abgrenzung autonomer Teilsysteme in Form von *Instanzen*, die Zusammenfassung von Instanzen über Aggregation und die Abgrenzung von *Softwareinfrastrukturen* bilden die wesentlichen Strukturprinzipien, die in grundlegender Weise auch in der Architektur zum Ausdruck kommen müssen.

Im nächsten Kapitel wird die in dieser Dissertation vorgeschlagene Systemarchitektur vorgestellt.

Kapitel 4

Architektur

Nachdem im vorangegangen Kapitel wesentliche Eigenschaften des Ablaufs und der Gliederung informationsverarbeitender Systeme vorgestellt wurden, sollen im weiteren diese Eigenschaften als Grundlage für eine Architektur aufgegriffen werden, die für das *CHEOPS*-Projekt eine geeignete Basis für den Entwurf und die Implementierung eines anpaßbaren Betriebssystems darstellt. Mit diesem Experimentalsystem wird auch die Plattform für weiterführende Arbeiten im Rahmen des *CHEOPS*-Projekts geschaffen. Die Architektur soll für ein breites Spektrum unterschiedlicher Zielbereiche anwendbar sein. Die Idee ist, übertragbare Grundmuster und Eigenschaften in einer generalisierten Architektur festzuschreiben und diese dann für konkrete Zielbereiche zu spezialisieren, indem Ausprägungsvarianten der Architektur und Ausführungseigenschaften von Elementen festgelegt werden.

4.1 Architekturziele

Eine Architektur umfaßt Aspekte der Systemgliederung, die sich nach Funktion und Zweck eines Systems richten. Bereits die Architektur eines Systems unterliegt teils in Konflikt stehenden Einflußfaktoren.

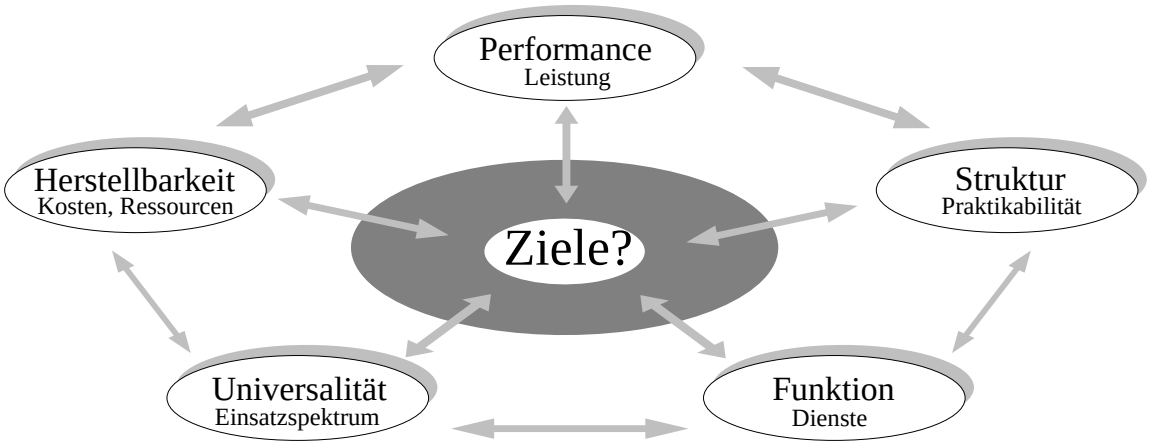


Abb.4.1 Einflußfaktoren und eine Auswahl genereller Architekturziele

Ein klassischer Konflikt besteht oft zwischen Strukturiertheit, Universalität und Effizienz. Man kann zahlreiche Beispiele dafür angeben. Kosten beschränken Ressourcen und damit Leistung und Funktion. Es können aber auch die Kosten für Struktur und Gliederung unangemessen sein, z.B. für lange Planungs- und Entwurfsphasen, für Redesigns oder auch wegen Effizienz.

Es lassen sich viele derartige Zusammenhänge herstellen, hinter denen letztendlich die Frage von *Aufwand und Nutzen* steht. Der Zweck bestimmt für die Architektur, auf welche Weise die einzelnen Faktoren gewichtet werden und an welchen Stellen Kompromisse zu schließen sind.

Für Betriebssysteme dominieren heute die Faktoren Effizienz und Universalität, die zum Teil auch in Konflikt zueinander stehen. Im Faktor Effizienz lag der hauptsächliche Grund, weshalb monolithische Strukturen in Betriebssystemen für Monoprozessormaschinen nicht abgelöst werden konnten. Auch für die in Kapitel 2 diskutierten Forschungsprojekte aus den USA liegt die primäre Motivation in höherer Effizienz. Die Quellen dafür werden in der wahlweisen Abrüstbarkeit teurer Ausführungsmodelle und in der anwendungsspezifischen Erweiterbarkeit der Kerne gesehen, um auf Effizienz optimierte Endsysteme (Betriebssystem + Anwendung) herzustellen.

Der zuletzt genannte Aspekt markiert auch eine neue, umfassendere Sichtweise für Betriebssysteme in der Forschung. Im kommerziellen Bereich war die Effizienz von Endsystemen schon immer ein wesentlicher Wettbewerbsfaktor. Das Betriebssystem wird in den Kontext des Gesamtsystems eingeordnet. In der Forschung besteht dagegen oft das Problem, daß der Hintergrund einer konkreten Anwendung fehlt. Es liegt daher nahe, den Betriebssystemkern isoliert zu betrachten und auch isoliert zu optimieren. Man konzentriert sich auf schnellere Signalisierungs- und Kommunikationsverfahren, auf schnelle Unterbrechungsbehandlungen, auf die effiziente Umsetzung von Systemrufen u.v.a. All das sind in der Tat auch effizienzkritische Mechanismen. Es folgt aber nicht mit Notwendigkeit daraus, daß suboptimierte Betriebssystemkerne auch die *optimale Infrastruktur* für Anwendungen sind, sondern der Kontext eines Endsystems ist das entscheidende Kriterium, und das schließt Anwendungen und im Fall adaptierbarer Systeme auch Anpassungen durch Anwender ein.

An *Mach* orientierte μ-Kerne sind ein Beispiel für eine isolierte Herangehensweise. μ-Kerne verfügen heute in der Tat über hocheffiziente IPC-Mechanismen. In Abschnitt 2.5 wurden Arbeiten zu effizienzsteigernden Optimierungen für *Mach* angegeben. Durch Auslagerung von Betriebssystemfunktionen in den Anwendungsbereich kann aber die Kommunikationsaktivität über das Maß ansteigen, welches aus der Beschleunigung der IPC-Mechanismen durch kleine hochoptimierte Kerne[1] resultiert. Der Gewinn wird überkompensiert, die Folge ist eine negative Leistungsbilanz. Wenn das Zielsystem *Unix* ist, dann ist eine Struktur im Sinne echter μ-Kerne mit ausgelagertem *Unix*-Server unangemessen. In *Mach-INKS* wird der *Unix*-Server deshalb auch in den Kern-Bereich zurückgebracht [Lepr93].

Für *L4* [Lie95] ist Effizienz das ausschließliche Ziel, ohne Kompromisse für Struktur und Portabilität der Kern-Software einzugehen. Der *L4*-Kern ist in Assembler unter Nutzung jeglicher Hardwarespezifik implementiert. Wegen des geringen Umfangs der Kern-Software stellt es nach Meinung des Autors kein Problem dar, diesen Kern für neue Maschinen jeweils neu zu implementieren. Für das Ziel von *L4* ist diese Herangehensweise konsequent und auch legitim.

Es ist aber auch legitim, Betriebssystemarchitekturen zu untersuchen, bei denen der Untersuchungsschwerpunkt an einer anderen Stelle liegt. Es kann dabei durchaus möglich sein, daß für ein optimiertes Endsystem vielleicht sogar eine bessere Effizienz erzielt werden kann als auf Basis universeller, suboptimierter Kerne. Das in der Forschung oft anzutreffende Streben nach bestmöglicher Kern-Effizienz ist somit in einen Gesamtzusammenhang einzuordnen, in dem auch andere Faktoren eine wichtige Rolle spielen, daß etwa Offenheit mit einer plausiblen Modularisierung verbunden ist und daß innere Strukturen verständlich sind, damit anwendungsspezifische Optimierungen durch Anwendungsentwickler mit vertretbarem Aufwand auch tatsächlich ausgeführt werden können. An diesen Stellen liegen Defizite in heutigen Ansätzen (vgl. auch Diskussion in Abschnitt 2.7), die durch das *CHEOPS*-Projekt aufgedeckt werden sollen und für die hier ein Lösungsansatz vorgestellt und begründet wird.

[1] z.B. durch Kerne, die aufgrund ihres geringen Umfangs in den CPU-Caches Platz finden (aber nicht *Mach*)

Das ist der Hintergrund und die Motivation für die *CHEOPS*-Architektur, die von einer homogenen Grundstruktur über alle Systemschichten und der separaten Betrachtung spezieller Ausführungseigenschaften geprägt ist. Die Grundstruktur ist dabei bewußt an die Struktur typischer Anwendungsbereiche angelehnt. Kern-Effizienz ist insofern ein Ziel, daß die angegebene Architektur durch bestmögliche Implementierung auch effizient umgesetzt wird, aber nicht unter Aufgabe der primären Architekturziele, denn das hieße, zu den bekannten, monolithischen Kern-Strukturen zurückzukehren, die natürlich bezüglich einer isoliert betrachteten Kern-Effizienz auf Monoprozessorsystemen nicht zu übertreffen sind.
Inwieweit ein nach der hier vorgeschlagenen Architektur hergestelltes Endsystem (Betriebssystem + Anwendung) vielleicht auch unter dem Blickwinkel Effizienz suboptimierten Kern-Architekturen überlegen sein kann, ist ein langfristiges Untersuchungsziel des *CHEOPS*-Projekts. Diese These läßt sich vorerst damit begründen, daß in der anwendungsspezifischen Optimierung des Betriebssystems unbestritten ein großes Leistungspotential liegt, das bislang kaum erschlossen ist, weil heutige Kerne keinen geeigneten Zugang für derartige Anpassungen erlauben, denn das setzt nicht nur Offenheit, sondern auch Plausibilität und klare innere Strukturen voraus, nicht nur in der Software, vor allem auch im ablaufenden System.

In dieser Arbeit wird eine dafür geeignete Architektur entworfen, und es wird der Nachweis der Implementierbarkeit anhand des *CHEOPS*-Kerns erbracht. Anschließend erfolgt eine Bewertung der Implementierung dieses Kerns.

4.2 Generalisierte Architektur

Gemäß dem Anliegen in dieser Arbeit wird eine weithin skalierbare, für verschiedene Umgebungen anwendbare generalisierte Architektur vorgestellt, die für konkrete Zielsysteme spezialisierbar ist. Die explizite Abtrennung generalisierter Architekturmerkmale von konkreten Ausprägungen in verschiedenartigen Zielbereichen ist ein wesentliches Prinzip und stellt in der hier vorgeschlagenen Form eine neue Herangehensweise für Betriebssysteme dar.

Die primäre Betrachtungswelt für die Architektur ist das ablaufende System, nicht die Struktur der Software zu seiner Herstellung. Die Beziehung zur Softwareherstellung nach objektorientierten Methoden wird für das *CHEOPS*-Projekt in einer zweiten Dissertation untersucht [Schu96]. Der Weg dafür liegt in der Schaffung einer Abbildung von Objektstrukturen der Sprachebene in eine adäquate Ablaufstruktur (vgl. Abschnitt 4.4).

Auch für die Architektur stellen sich die Fragen:

- Was ist *universell* ?
- Was ist *speziell* ?
- Was ist *spezialisierbar* ?

Die generalisierte Architektur ist *universeller* Natur. Sie wird für ein konkretes Zielsystem *spezialisiert*, indem Ausprägungsvarianten abgeleitet und Ausführungseigenschaften zugeordnet werden. Die im Ergebnis entstehende Architektur ist dann für ein Zielsystem *speziell* ausgelegt.
Erst diese Vorgehensweise ermöglicht das hier angestrebte Maß an Skalierbarkeit der Architektur, um sie tatsächlich in allen Schichten eines Zielsystems bzw. in verschiedenartigen Zielsystemen wiederzuverwenden und als homogene Grundstruktur auf das Gesamtsystem aufzuprägen.
Ausgangspunkt ist die im vorangegangenen Kapitel gegebene Modellbetrachtung für informationsverarbeitende Systeme. Als grundlegendes Strukturmerkmal wurden (Teil-) Systeme herausgearbeitet, die in der Architektur als Verarbeitungsinstanzen dargestellt werden. Ihr Zweck ist die selbständige Ausführung von Diensten auf Veranlassung von außen durch innere Verarbeitungsprozesse über inneren Daten.

4.2.1 Grundelemente: Daten, Prozesse, Instanzen – Infrastruktur

Das Ziel der Analyse in Kapitel 3 war, Wesensmerkmale informationsverarbeitender Systeme herauszuarbeiten, um die hier vorgeschlagene Architektur daran zu orientieren und zu begründen.

Verarbeitungsgegenstände sind Daten, die kodierte Informationen über Zustände und Steuerung enthalten (vgl. $\mathcal{Z}_{\mathcal{W},\mathcal{S},\mathcal{V},\mathcal{E}}$ aus Abschnitt 3.3.1). Gesteuerte Aktivitäten ($\rightarrow$ *Prozesse*) führen Operationen über diesen Daten aus. ***Daten*** und ***Prozesse*** sind die elementaren Merkmale von Informationsverarbeitung. In den Abbildungen 4.2 und 4.3 ist weiterhin die Abgrenzung autonomer *Teilsysteme* gezeigt. Damit ist ein wesentliches Strukturprinzip informationsverarbeitender Systeme gegeben, das in der Architektur über ***Verarbeitungsinstanzen*** zum Ausdruck kommt. Verarbeitungsinstanzen bilden für sich abgeschlossene Teilsysteme und lassen sich intern rekursiv nach dem gleichen Muster weiter zerlegen ($\rightarrow$ *Aggregationsrekursion*). Der grundlegende Zusammenhang zur Gliederung informationsverarbeitender Systeme aus Kapitel 3:

$$System_i = \{\, Grundelemente_{i,i} \}\cup\{\, Handlungsträger_{i,j} \}\cup\{\, Teilsysteme_{i,k} \}$$
Rekursion: $Teilsystem_{i,k} = System'_k$, Abbruch bei: $System'_k = \varnothing$,

wird in der Architektur adäquat wiedergegeben:

$$System_i = Instanz_i = \{\, Daten_{i,i} \} \cup \{\, Prozesse_{i,j} \} \cup \{\, Instanzen_{i,k} \}$$
Rekursion: $Instanz_{i,k} = Instanz'_k$, Abbruch bei: $Instanz'_k = \varnothing$.

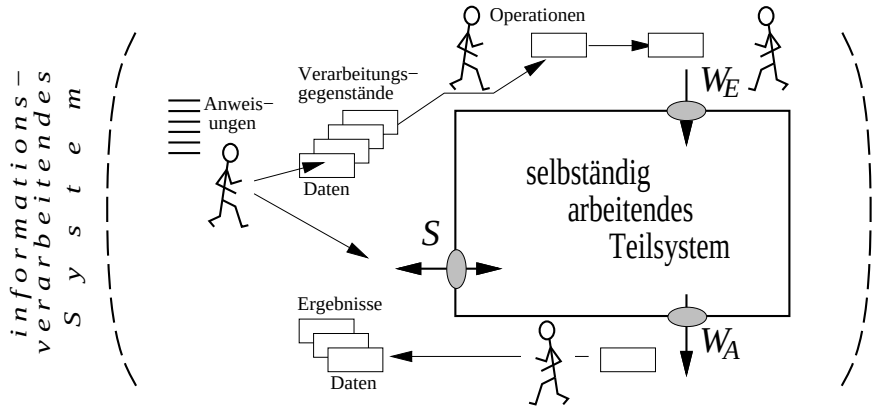


Abb.4.2 Modellbetrachtung: Informationsverarbeitung

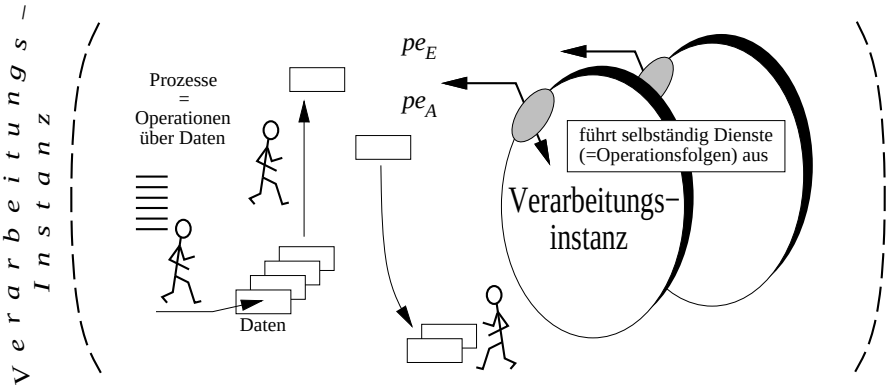
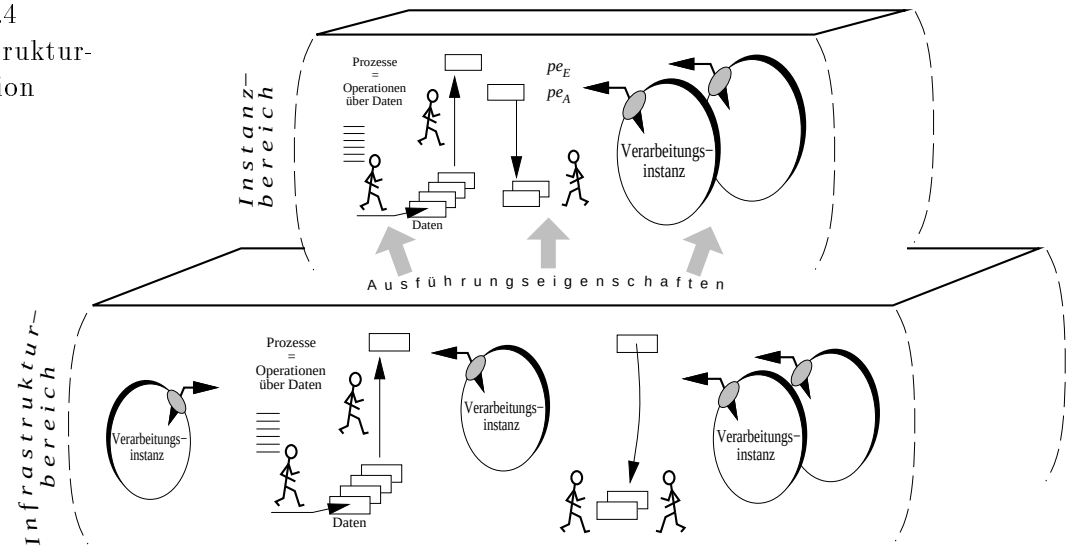


Abb.4.3 Widerspiegelung in der Architektur: Daten, Prozesse, Verarbeitungsinstanzen

Um Existenz- und Verarbeitungsgrundlagen zu schaffen, muß es eine geeignete ***Infrastruktur*** geben, in der wiederum Verarbeitungsprozesse ausgeführt werden. Durch die prinzipielle Ähnlichkeit des Geschehens in der Infrastruktur liegt es nahe, dort eine analoge Gliederung, d.h. die gleichen Grundelemente und Beziehungen anzuwenden.

Abb.4.4
Infrastruktur-
rekursion



Die Notwendigkeit des Abbaus dieser Rekursion durch immer elementarere Ausprägungen von Grundelementen wurde bereits begründet, auch der Begriff Rekursion macht das deutlich.

Die Gliederungsmittel Instanz und (Software-) Infrastruktur müssen aber nicht notwendig Anwendung finden, Prozesse und Daten bleiben die alleinige Voraussetzung für Verarbeitung. Instanzen (*Aggregationsrekursion → Kapselung*[2]) und Infrastrukturen (*Infrastrukturrekursion → Kapselung*[3]) sind aber wegen der Kapselungseigenschaft zwei wichtige Prinzipien für den Aufbau klarer Ablaufstrukturen in informationsverarbeitenden Systemen.

Von den Grundelementen der Architektur (Daten, Prozesse, Instanzen, Infrastrukturen) sollen Daten und Prozesse wegen ihres elementaren Charakters als ***Basiselemente*** bezeichnet werden. Aus der Menge der Daten sollen die Steuerungsinformationen für Prozesse herausgestellt werden, so daß im weiteren unter dem Begriff *Daten* die eigentlichen Verarbeitungsgegenstände verstanden und von *Programmdaten* getrennt werden. ***Instanzen*** und ***Infrastruktur*** sind höherwertige Elemente, die aus Gruppierungen von Basiselementen erzeugt werden.

Fazit: Die Merkmale der vorgeschlagenen Architektur sind an Beobachtungen der Realität ablaufender Systeme orientiert und können über drei Kategorien zusammengefaßt werden.

- Die ***Basiselemente*** für Verarbeitung sind
  - *Daten:* → Verarbeitungsgegenstände,
  - *Programme:* → Daten mit Steuerungsinformationen für Prozesse und
  - *Prozesse:* → gesteuerte Aktivitäten, die Verarbeitungen über den Daten ausführen.
- Über die *Kapselung von Basiselementen* ist ein Mittel gegeben, identifizierbare und autonom arbeitende ***Instanzen*** herzustellen. Die ***Aggregationsrekursion*** gibt an, daß Instanzen intern wiederum Instanzen enthalten können.
- Mit der *Rekursion von* ***Infrastrukturen*** kann ein gleichartig strukturierter Bereich abgegrenzt werden, mit dem neuartige Elemente und Ausführungseigenschaften für einen übergeordneten Bereich *erzeugt, transformiert und gesteuert* werden.

[2] Kapselung wird hier im Sinne funktionaler, semantischer und auch technischer Abgrenzung verstanden.
[3] Auch infrastrukturelle Aufgaben werden in dieser Hinsicht abgegrenzt und damit gekapselt.

4.2.2 Instanzen

- **Gründe für Instanzen**

Die Abgrenzung von Instanzen stellt eine Option bei der Herstellung von Systemen dar, die nicht notwendigerweise Anwendung finden muß. Aus der Betrachtung eines Systems von außen ist es unerheblich, ob für das System intern eine Abgrenzung von Teilsystemen oder Teilverarbeitungsinstanzen erfolgt, d.h. eine solche innere Strukturiertheit vorliegt. Für die Benutzung eines Systems sind Funktion, Leistung und Bedienung die maßgebenden Faktoren. Innere Strukturierung wird interessant, wenn Anpassungen im System vorgenommen werden sollen.
Instanzen sind dafür ein wichtiges Strukturelement. Durch Instanzen werden Steuerungsebenen in das System eingeführt, die relativ zu einer Instanz durch eine äußere und eine innere Steuerungsebene gebildet werden. Über die ***äußere Steuerungsebene*** veranlassen Prozesse oder andere Instanzen die Ausführung von *Diensten* $\mathcal{D}$ durch *Folgen von Operationen* φop in einer Instanz. Die ***innere Steuerungsebene*** gibt die Auflösung in Teilverarbeitungen bzw. Teiloperationen $op \rightarrow \varphi op'$ an und ist nach außen nicht sichtbar. Auf diese Weise werden höherwertige Dienste in Systemen hergestellt, mit denen man einfacher umgehen kann und die jeweils näher an der Zweckbestimmung der Anwendung orientiert sind.
Die Herstellbarkeit höherwertiger Dienste, das Verbergen oder Kapseln interner Verarbeitung und die selbständige Arbeit ohne weitere Aufwendungen von außen sind die Vorteile von Instanzen. Sie bilden eine wichtige Grundlage für Struktur und Abstraktion in ablaufenden Systemen. Abgegrenzte Instanzen sind auch einfacher herzustellen, zu testen und wiederverwenden.

- **Wesenszüge von Instanzen**

äußere Charakteristik: Dienste, Operationen, Protokolle, Identifikation, Typ bzw. Schnittstellendefinition, Schnittstelle, Interaktion

Wenn Instanzen auf äußere Veranlassung Dienste ausführen, müssen sie nach außen identifizierbar sein. Die Identität beruht letztlich auch auf der Abgrenzbarkeit von Instanzen. Die Funktion einer Instanz i kommt in einer Menge von Operationsbeschreibungen mit zugehörigen Eingabe- bzw. Ausgabewerten zum Ausdruck: $\mathcal{TYP}_i = \lambda opd_i = \{ opd_j \}$ mit $opd_j = (anw_j , \lambda w_{\mathcal{E}j} , \lambda w_{\mathcal{A}j})$. Anweisungen werden als Operationen ausgeführt. Dafür sind Protokolle notwendig. Die Beziehungen zwischen ***Diensten, Operationen*** und ***Protokollen*** wurden in Abschnitt 3.2.2 gezeigt:

- $\varphi\mathcal{D} = \{ \mathcal{D}_i \} = \{ \mathcal{D}_1, \mathcal{D}_2, \ldots \mathcal{D}_n \},$
- $\mathcal{D}_i = \varphi op_i = \{ op_1^{t_1}, op_2^{t_2}, \ldots op_m^{t_m} \}$ $\quad \varphi op_i \in \varphi\mathcal{OP}, \; op_j^{t_j} \in \mathcal{OP},$
- $op_j : \quad (anw_j, \lambda b_{\mathcal{E}j}, \lambda w_{\mathcal{E}j}) \rightarrow (\lambda b_{\mathcal{A}j}, \lambda w_{\mathcal{A}j})$ $\quad op_j \in \mathcal{OP},$
- $\forall\ op_j : \exists\ pe_{\mathcal{E}j} = (anw_j, \lambda b_{\mathcal{E}j}, \lambda w_{\mathcal{E}j})$ $\quad pe_{\mathcal{E}j} \in \{ \mathcal{S}_{\mathcal{E}_{Anw}} \times \mathcal{S}_{\mathcal{E}_{Bed}} \times \mathcal{W}_{\mathcal{E}} \},$
- $\forall\ op_j : \exists\ pe_{\mathcal{A}j} = (\lambda b_{\mathcal{A}j}, \lambda w_{\mathcal{A}j})$ $\quad pe_{\mathcal{A}j} \in \{ \mathcal{S}_{\mathcal{A}} \times \mathcal{W}_{\mathcal{A}} \}.$

- $\mathcal{D}_i = \varphi pe_i = \{ pe_1^{t_1}, pe_2^{t_2}, \ldots pe_m^{t_m} \}$ $\quad \varphi pe_i \in \varphi PE, \; pe_j^{t_j} \in PE, \; PE \subseteq \{ PE_{\mathcal{E}} \times PE_{\mathcal{A}} \},$
- $\mathcal{D}_i = \{ (pe_{\mathcal{E}}{}_1^{t_1} \rightarrow pe_{\mathcal{A}}{}_1^{t_1+d_1}), (pe_{\mathcal{E}}{}_2^{t_2} \rightarrow pe_{\mathcal{A}}{}_2^{t_2+d_2}), \ldots (pe_{\mathcal{E}}{}_m^{t_m} \rightarrow pe_{\mathcal{A}}{}_m^{t_m+d_m}) \}.$

Die Anforderungen an ein System bestimmen, welche statischen oder dynamischen Aspekte zum ***Identifikationsschema*** für Instanzen gehören, ob nur Dienste oder auch Operationen bzw. Protokolleinheiten identifizierbar sind. Identifikation heißt die Abgrenzung von Elementen eines Gegenstandsbereichs anhand unterscheidender Eigenschaften und die Zuordnung eineindeutiger *Identifikatoren* zu diesen Elementen. Welche Eigenschaften für die Identifikation relevant sind, hängt vom Gegenstandsbereich ab. Es ist auch möglich, dieselben Dinge nach mehreren Unterscheidungsmerkmalen auch verschieden zu identifizieren. Im Unterschied zu ***Namen*** wird von Identifikatoren die Eineindeutigkeit im Gegenstandsbereich gefordert. Die Benennung von Elementen ist dann sinnvoll, wenn Menschen mit diesen Elementen umgehen sollen. Über Namen kann in Grenzen auch etwas Semantik über ein Element transportiert werden.
Identifikation ist ein wesentliches Merkmal in einem strukturierten Gegenstandsbereich. Benennung, Adressierung oder Lokalisierung sind sekundärer Natur, da sie identifizierte Elemente voraussetzen. Identifikations-, Namens- oder Adreßräume sind dabei oft verschieden strukturiert.

In jedem strukturierten Gegenstandsbereich muß daher mit Notwendigkeit ein Identifikationsschema existieren. Für Instanzen kann man verschiedene Identifikationsschemata angeben.

- ***(1)*** – *Identifikation statischer Merkmale der* ***Beschreibung***[4]:
  - (a) Identifikation des Typs einer Instanz i : $\mathcal{TYP}_i = \lambda opd_i = \{ opd_j \}$, Schnittstellendefinition,
    - (b) Identifikation einer Instanz i in der Beschreibung,
      - (c) Identifikation von Operationsbeschreibungen: $opd_j = (anw_j, \lambda w_{\mathcal{E}j}, \lambda w_{\mathcal{A}j})$,
        - (d) Identifikation der Beschreibungen für: anw_j, $\lambda w_{\mathcal{E}j}$, $\lambda w_{\mathcal{A}j}$;

- ***(2.a)*** – *Identifikation elementbezogener Merkmale im* ***ablaufenden System:***
  - (a) Identifikation der Schnittstelle(n) einer Instanz i zur Protokollabwicklung,
    - (b) Identifikation der Instanz i im ablaufenden System,
      - (c) Identifikation möglicher Dienste und Operationen,
        - (d) Identifikation von Werten für Zustandsinformation (Daten), Aktivitäten und Steuerungen (Prozesse) ;

- ***(2.b)*** – *Identifikation verarbeitungsbezogener Merkmale im* ***ablaufenden System:***
  - (a) Identifikation einer Dienstausführung: $\mathcal{D}_i = \varphi op_i = \{ op_j \}$,
    - (b) Identifikation einer Operation: $op_j : (anw_j, \lambda b_{\mathcal{E}j}, \lambda w_{\mathcal{E}j}) \rightarrow (\lambda b_{\mathcal{A}j}, \lambda w_{\mathcal{A}j})$,
      - (c) Identifikation von Protokolleinheiten: $pe_{\mathcal{E}}$, $pe_{\mathcal{A}}$.

In Systemen sind nicht immer alle genannten Merkmale für die Identifikation notwendig und sinnvoll. In parallelen Systemen ist es aber beispielsweise erforderlich, im ablaufenden System nicht nur Instanzen und mögliche Dienste zu identifizieren, sondern auch individuelle Dienstausführungen, um diese bei paralleler Ausführung unterscheiden und richtig zuordnen zu können.

Für ***Schnittstellen*** und ***Schnittstellendefinitionen*** gelten die Aussagen aus Abschnitt 3.2.4, daß Schnittstellen einer Instanz durch gemeinsame Elemente mit anderen Instanzen oder Prozessen der Umgebung bzw. der Infrastruktur gebildet werden, über die Protokolle ablaufen.

[4] angelehnt an sequentielle Programmiersprachen, bei denen keine Abläufe oder Prozesse identifizierbar sind

Interaktion schließt den Steuerungsaspekt (→ *Signalisierung*, Abschnitt 3.3.7) und den Austausch von Verarbeitungsdaten (→ *Kommunikation*, Abschnitt 3.3.8) ein. Interaktion bezieht sich auf das Zusammenwirken von Instanzen oder Prozessen für Koordination bzw. Kooperation bei der Verarbeitung. Interaktion erfolgt über Protokolle, für deren Ablauf wiederum Interaktion in anderen Steuerungsebenen erforderlich ist.

innere Charakteristik: = (äußere Charakteristik)' und
Daten (Zustände), Prozesse, innere Instanzen, Abgrenzung, Kapselung

Aufgrund der rekursiven Auflösung von Instanzen in innere Instanzen, Daten und Prozesse gelten die oben getroffenen Aussagen zu Diensten, Operationen, Protokollen, Identifikation und Interaktion auch für das Innere von Instanzen, mit dem Unterschied, daß es sich hier um eine andere Steuerungsebene handelt, die aber im Grunde denselben Strukturmerkmalen unterliegt. Instanzen sind zu selbständiger Ausführung von Operationen fähig, wozu sie eigene innere Prozesse und innere Zustände benötigen. Instanzen können intern rekursiv wiederum Instanzen enthalten (vgl. Abschnitt 3.3).

Instanzen sind identifizierbar und müssen somit voneinander abgrenzbar sein. Die Unterscheidungsmerkmale für ***Abgrenzung*** *oder* ***Kapselung*** sind für die generalisierte Architektur zum einen *funktional oder semantisch* gegeben, daß von einer Instanz eine bestimmte Menge von Diensten ausgeführt werden kann und diese Funktion an das Element "Instanz" geknüpft ist. Abgrenzung kann auch an das Merkmal der Steuerungsebenen gebunden werden, daß Dienste von außen veranlaßt und intern in Teilverarbeitungen aufgelöst werden.
Neben der funktionalen Abgrenzung sollen Instanzen auch *technisch* abgegrenzt sein, was sich in der generalisierten Architektur darin ausdrückt, daß innere Prozesse nur Übergänge innerer Zustände bewirken können, nicht aber auf äußere Zustände oder Zustände anderer Instanzen direkten Einfluß nehmen können. Auch die Umkehrung gilt, daß äußere Prozesse keine inneren Zustände einer Instanz verändern können. Das Prinzip der technischen Kapselung wird für die generalisierte Architektur an der ***Beschränkung der Einflußsphäre von Prozessen*** festgemacht. Die Umsetzung für ein konkretes Zielsystem ist dann Gegenstand der Spezialisierung (vgl. Abschnitt 4.3).

Das wesentliche Abgrenzungsmerkmal für die Identifikation von Instanzen liegt in der Verknüpfung von Prozessen mit zugehörigen Werten (Zuständen) zum Zweck der selbständigen Ausführung von Diensten. Auf diese Weise erhalten Instanzen in einem System eine Identität, die funktional und auch technisch begründet ist.

Unterschied: Instanzen – Prozesse

Instanzen bilden durch die autonome Ausführung von Diensten ein funktional, semantisch und technisch abgegrenztes (Teil-) System. Prozesse benötigen Verarbeitungsgegenstände in ihrer Umgebung, sie bilden für sich allein kein sinnvolles System. Die technische Abgrenzung von Instanzen wird an der Beschränkung der Einflußsphäre von Prozessen festgemacht. Prozesse *per se* unterliegen keiner solchen Beschränkung. Durch die Auflösung von Diensten in Instanzen bzw. durch das Herstellen höherwertiger Dienste werden Steuerungsebenen in ein System eingeführt, die zur Übersicht beitragen. Prozesse bilden dagegen nur jeweils eine Steuerungsebene.

Für Betriebssysteme eröffnet sich damit die Möglichkeit, die klassische Sichtweise von separaten ***Prozessen und Betriebsmitteln*** um dienstausführende ***Instanzen*** zu ergänzen [Kal92b] und damit ein wichtiges Strukturelement zu gewinnen, welches es ermöglicht, das Prinzip autonomer Teilsysteme auf Betriebssysteme zu übertragen. Dieses Prinzip wird in Systemen schon lange mehr oder weniger explizit angewandt (z.B. die unterbrechungsgesteuerte und damit autonome Arbeitsweise von Plattentreibern bei asynchronem Lesen bzw. Schreiben von Blöcken, auch die

Server-Prozesse der μ-Kernen gehören dazu). Für Anwendungsbereiche findet man in den meisten Systemen ohnehin Instanzen als alleinige Ausführungseinheiten vor. Eingangs wurde es als eine wichtige Zielstellung der Architektur genannt, daß die Merkmale typischer Anwendungsbereiche auch im Inneren von Betriebssystemen zum Ausdruck kommen sollten. Es ist daher nur folgerichtig, Instanzen zu einem expliziten und dominierenden Strukturmerkmal zu erklären.

Unterschied: Instanzen – Objekte

Objektorientierung ist eine Methode zur strukturierten Beschreibung von Analysen aus Problembereichen, von Entwürfen und von Programmen. Das Ziel ist die Herstellung strukturierter Software. In Abschnitt 2.2.1 wurde der Unterschied zwischen Software- und Ablaufstruktur erklärt. Objekte im Verständnis der Softwaretechnik beschreiben den statischen Aspekt der Steuerungs- und Zustandsinformation. Programme und Daten von Instanzen können natürlich über Objekte modelliert und beschrieben werden, das Wesen von Instanzen schließt jedoch auch die dynamische Seite realer Verarbeitungsprozesse, deren Koordination und Kommunikation ein. Objekte können selbständig keine Dienste ausführen. Ein System von Objekten bedarf mindestens eines "äußeren" Prozesses, der die eigentliche Verarbeitungsleistung erbringt und der von den Programmen (Methoden) und Zuständen (Daten) der Objekte gesteuert wird.
Die mit selbständig, d.h. parallel arbeitenden Instanzen einhergehenden Fragen von Kooperation bzw. Koordination sind in den heute etablierten objektorientierten Sprachen nicht berücksichtigt und müssen außerhalb der Sprache und damit außerhalb des Konzepts umgesetzt werden.
In der Forschung findet man zwar seit langem Ansätze: *aktive Objekte*, *Concurrent Objects*, *Aktoren* usw. (vgl. Abschnitt 1.8.2), der Übergang in breit angewandte Methoden in der Praxis wurde aber noch nicht vollzogen, nicht zuletzt deshalb, weil man die gewohnte Sequentialität als ausschließliche Ablaufform nicht aufgeben will. Dafür gibt es Gründe, und es sind bislang zuwenig Einsatzfelder, wo dieser Schritt zwangsläufig erfolgen muß. Dazu zählen aber verteilte Systeme, deren Grundeinheiten Prozesse (hier im Sinne von Instanzen) sind.

In der frühen objektorientierten Begriffswelt lassen sich jedoch Interpretationen finden, die nicht notwendig mit einem sequentiellen Ablauf verknüpft sind: *"The system is composed of objects that interact only by sending and receiving messages ... that invoke methods in objects."* [Xer81], S. 36, 46. Es ist nichts darüber ausgesagt, daß es in einem System zu einem Zeitpunkt höchstens ein Objekt geben kann, welches entweder eine Nachricht empfängt, sendet oder eine Methode ausführt[5].
Im Gegenteil, die ursprüngliche Denkweise ging sogar davon aus, daß diese Abläufe durchaus in mehreren Objekten gleichzeitig ablaufen können. So gab es in *Smalltalk-80* von Anfang an ein Mehrprozeßkonzept[6]. Das hängt auch damit zusammen, daß eine Ahnenlinie von Objektorientierung in der Diskreten Ereignissimulation liegt, zumindest an vielen Stellen in der Literatur dort gesehen wird. In dem Artikel *"SIMULA – an ALGOL-Based Simulation Language"* [DN66] ist zwar von Objekten[7], die ein bestimmtes Verhalten bei Ereignissen zeigen, und Prozessen[8](!), da Ereignisse mehrere Objekte gleichzeitig beeinflussen, nicht aber von Klassen, Vererbung oder Polymorphie die Rede. Diese Elemente kamen erst später als folgerichtige Fortentwicklung des *ADT*-Konzepts hinzu [Marty88]. Das ist aber eine andere, von der zuerst genannten völlig unabhängige Entwicklungslinie, die aber heute das dominiert, was mehrheitlich unter Objektorientierung verstanden wird. Die Frage nach Prozessen stellte sich hier nicht mehr – und sie stellt sich noch nicht wieder nachdrücklich genug. Für die Zukunft ist es aber wichtig, an die erstgenannte Entwicklungslinie anzuknüpfen. Das *Client-Server-Modell* (→ *Instanzen*) ist hierfür ein geeigneter Ausgangspunkt, auch für *"Objektorientierung in Betriebssystemen"* [Grau93b].

[5] Abgesehen von der Tatsache, daß nicht Objekte, sondern Prozesse Methoden ausführen.
[6] *"... processes are provided by classes ProcessorScheduler, Process, and Semaphore."* [Xer81], S. 47
[7] im Text, nicht als explizites Sprachkonstrukt
[8] Prozesse sind dagegen über das Schlüsselwort `activity` ein explizites Sprachkonstrukt.

Betriebssysteme sind ablaufende, aktive, *operierende Systeme*, die eine dynamische Betrachtung verlangen. In [Son93, Grau93b] wird daher unterschieden, ob ein Betriebssystem objektorientiert entworfen und implementiert wurde (→ *Herstellung: Choices*, *ETHOS*) oder ob es sich um ein *Objekt Management System* handelt (→ *Ablauf:* kernunterstützt: *BirliX*, *Clouds* bzw. aufgesetzte Schicht: *SOS*, *Emerald*). *Apertos* [Yok93] (*Sony CSL* und *Keio University*) versucht, beides zu vereinen. Es wurde mit *Concurrent Objects* [YT87] konzipiert und implementiert.

Fazit:

Um den Unterschied zu Objekten und Objektorientierung im heute mehrheitlich akzeptierten Verständnis begrifflich deutlich zu machen, werden in dieser Arbeit autonom arbeitende Elemente eines ablaufenden Systems als Verarbeitungsinstanzen und nicht als Objekte bezeichnet. Instanzen sind nicht mit "reinen" Prozessen im Sinne gesteuerter Aktivitäten oder mit Objekten im Sinne der objektorientierten Modellierung und Programmierung gleichzusetzen. Man kann sie ggf. als eine Kombination aus beidem betrachten. Instanzen sind auch kein neues Konzept, *Unix*-Prozesse, *Mach*-Tasks oder *BirliX*-Teams sind Beispiele für bekannte Ausprägungen. Instanzen sind aber ein herausragendes Architekturmerkmal für ablaufende Systeme.

4.2.3 Instanzbereiche

- **Gründe und Wesenszüge für Instanzbereiche**

Ein Instanzbereich wird durch eine Gruppe von Instanzen in einer Systemschicht mit jeweils gleichen Ausführungseigenschaften gebildet. Das Element der Instanzbereiche wurde eingeführt, damit Gruppen von Instanzen unterschieden werden können, die auch innerhalb einer Schicht verschiedene Infrastruktureigenschaften besitzen. Das bezieht sich vor allem auf unterschiedliche Infrastrukturdienste, da nicht alle Instanzen alle Infrastrukturdienste gemeinsam besitzen müssen, oder umgekehrt, daß Gruppen von Instanzen mit individuellen Infrastrukturdiensten ausgestattet werden können. Die Gruppierbarkeit über Instanzbereiche ist auch deshalb sinnvoll, um aus technischen Gründen unterschiedlich ausgeprägte Infrastrukturdienste für Instanzen differenzieren zu können. Ein Beispiel ist der Unterschied zwischen lokaler und maschinenübergreifender Kommunikation zwischen Instanzen. Obwohl Instanzen in beiden Fällen in derselben Schicht in einem verteilten Gesamtsystem existieren, sind die Mechanismen in der Infrastruktur für beide Fälle natürlich verschieden ausgelegt, auch mit Rückwirkungen auf die Instanzen, was beispielsweise ein verändertes Laufzeitverhalten oder Fehlermöglichkeiten betrifft.
Daraus ergeben sich unterschiedliche Ausführungseigenschaften für die beteiligten Instanzen, die in der generalisierten Architektur über das Mittel der Instanzbereiche differenzierbar sind. Es entspricht auch dem Anliegen anwendungsanpaßbarer Betriebssysteme oder Infrastrukturen, bestimmten Gruppen von Instanzen individuelle Infrastrukturdienste zuordnen zu können.

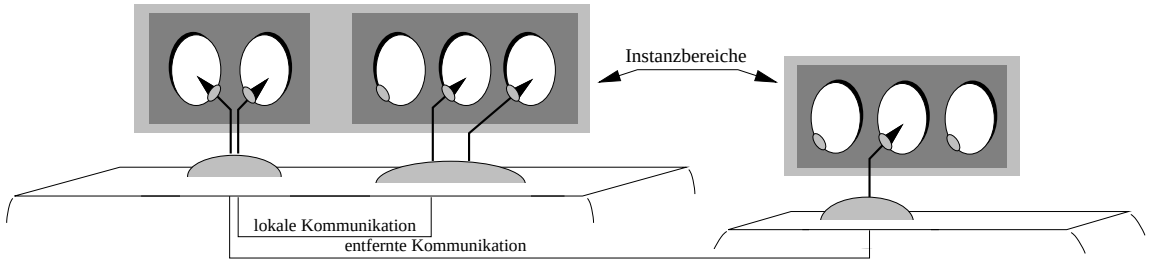


Abb.4.5 Instanzbereiche

Instanzbereiche sind nicht notwendig nach außen identifizierbar, sie bieten lediglich ein Mittel, Gruppen von Instanzen nach bestimmten Kriterien zusammenzufassen. Eine sinnvolle Anwendung für identifizierbare Instanzbereiche ist die Gruppenkommunikation. Eine Nachricht an einen Instanzbereich, d.h. an eine Gruppe von Instanzen, erreicht alle Mitglieder dieser Gruppe.

4.2.4 Systemschichten – Infrastrukturrekursion

- **Gründe für die Abgrenzung von Systemschichten – Softwareinfrastrukturen**

Einige Gründe für den Sinn der Abgrenzung von Softwareinfrastrukturen (→ *Infrastrukturrekursion*) wurden bereits in Abschnitt 3.5 erörtert:

- Das ist zum einen die Herstellung höherwertiger, für Anwendungen besser geeigneter Infrastrukturdienste und Ausführungsmodelle (→ *abstrakte Maschinen*).
- Ein anderer Zweck liegt im Herstellen oder Erzeugen geeigneter Verarbeitungseinheiten (Prozesse, Instanzen) für übergeordnete Instanzbereiche.
- Softwareinfrastrukturen sind auch ein wesentliches softwaretechnologisches Element zur Reduktion von Komplexität (→ Kapselungseigenschaft der Infrastruktur, Herstellen höherwertiger Dienste und Ausführungsmodelle, Zentralisierung von Aufgaben der globalen Ablaufsteuerung und Ressourcenverwaltung u.a.).
- Konzeptionelle Schichtungen von Softwaresystemen können unmittelbar in Schichten des ablaufenden Systems abgebildet werden (vgl. Abbildung 2.6). Damit kann der Unterschied zwischen Entwurfs- und Implementierungsstruktur überbrückt werden.
- Da in einer Softwareinfrastruktur letztlich auch Verarbeitungsprozesse stattfinden, ist es naheliegend, die gleichen Modellierungsprinzipien auch innerhalb der Infrastruktur anzuwenden. Daraus resultiert eine prinzipielle Gleichartigkeit in allen Schichten eines Systems, die auch in der Architektur des Gesamtsystems deutlich zum Ausdruck kommen sollte. Der notwendige Abbau der Rekursion erfolgt über Schichten durch jeweilige Ausprägungsvarianten und Ausführungseigenschaften der Elemente hin zu elementareren Niveaus.

Der Preis für diese wünschenswerten Eigenschaften liegt in einem Mehraufwand für Verwaltung und Verarbeitung, um den sich die Nutzleistung des Systems reduziert. Dieser Mehraufwand ist jedoch relativ zu bewerten. Seine Inkaufnahme hängt von der Zielstellung für ein System ab.

Mit Softwareinfrastrukturen ist neben der Kapselung und Aggregation von Instanzen ein weiteres wichtiges Mittel für Hierarchiebildung und Struktur in der Architektur gegeben. Es geht aber nicht darum, möglichst viele verschiedenartige Softwareinfrastrukturen herzustellen, sondern dieses Mittel *sinnvoll* unter Abwägung von Aufwand und Nutzen einzusetzen. In der Architektur muß aber die prinzipielle Möglichkeit dafür vorgesehen sein.

- **Wesenszüge von Systemschichten – Softwareinfrastrukturen**

äußere Charakteristik:	Ausführungsmodell, Dienste, Interaktion, Protokolle, Schnittstellen

Softwareinfrastrukturen stellen die Existenz- und Arbeitsgrundlagen für übergeordnete Instanzbereiche her. Daraus leiten sich fünf prinzipielle Aufgaben einer Softwareinfrastruktur ab:

- Herstellung der ***Elemente*** zur Verarbeitung (Daten, Prozesse – Instanzen),
- Herstellung eines ***Ausführungsmodells*** der Verarbeitung (Eigenschaften der Elemente),
- Bereitstellung von ***Diensten*** in der Infrastruktur,
- Ausführung der globalen ***Ablaufsteuerung*** für übergeordnete Instanzbereiche und die
- Vermittlung von ***Interaktionsbeziehungen*** zwischen verarbeitenden Elementen (Prozesse, Instanzen) untereinander, zur Umgebung und zur Infrastruktur.

Eine Softwareinfrastruktur erbringt diese Leistungen und kapselt nach außen ihre inneren Elemente und Verarbeitungen. Insofern kann eine Softwareinfrastruktur auch als ein abgeschlossenes System betrachtet werden, das Verarbeitungsleistungen erbringt. Die Analogie zum Wesen von Instanzen ist offensichtlich und begründet erneut die Folgerichtigkeit der im Grundprinzip gleichen inneren und äußere Modellierung von Softwareinfrastrukturen und Instanzen.

innere Charakteristik: Herstellen und Transformation von Verarbeitungselementen, Repräsentation, Identität, Interaktion, Dienste, Ausführungsmodell

Der primäre Zweck von Infrastruktur ist die Herstellung von Verarbeitungselementen, Diensten und Ausführungseigenschaften und die Vermittlung von Interaktionsbeziehungen für übergeordnete Instanzbereiche.

Das Herstellen und Beseitigen von Verarbeitungselementen bezieht sich auf Aktivitäten und Steuerungen (→ *Prozesse*), auf Verarbeitungsgegenstände (Daten) und Instanzen. Die Existenz, das Herstellen und Beseitigen dieser Verarbeitungselemente wird durch ***Repräsentationen*** dieser Elemente in der Infrastruktur umgesetzt. Eine Repräsentation umfaßt einen Zustand, über dem in der Infrastruktur Operationen ausgeführt werden. Die Repräsentation bestimmt in der Regel auch die ***Identität*** eines Elements und ist Grundlage für das Identifikationsschema.

Durch die beliebige Gestaltungsmöglichkeit von Repräsentationen und Verarbeitungsprozessen in der Infrastruktur können für übergeordnete Instanzbereiche neuartige Elemente mit neuen Ausführungseigenschaften hergestellt werden. Da es in allen Schichten prinzipiell gleiche Elemente gibt (Prozesse, Daten, Instanzen), muß dieser Herstellungsprozeß als ***Transformation*** höherwertiger Ausprägungen von Eigenschaften dieser Elemente aus elementareren Eigenschaften anderer Elemente erfolgen. Diese Transformationsprozesse sind eine primäre Aufgabe der Infrastruktur. Transformationsprozesse sind auch Herstellungsprozesse für neuartige Elemente im System. Nach [Kal90] lassen sie sich in zwei Kategorien einteilen. Elemente der Schicht i können durch zeitliche oder räumliche Teilung *virtuell* für die Schicht $i+1$ vervielfacht werden. Sie haben prinzipiell gleiche Eigenschaften. Es können aber auch neuartige, *logische* Elemente mit neuen Eigenschaften hergestellt werden. Beispiele für die erste Kategorie sind die Abbildung virtuellen Speichers auf Hauptspeicher und Externspeicher oder das Herstellen "virtueller" Prozessoren durch Umschalten von Aktivität zwischen Steuerungen. Die Abbildung von Dateisystemen auf blockstrukturierte Datenträger ist ein Beispiel für die zweite Kategorie.

Da Elemente einer Schicht $i+1$ Repräsentationen in der Infrastruktur i besitzen, ist es konsequent und naheliegend, die Mittel für die ***globale Ablaufsteuerung*** und für ***Interaktion*** in der Infrastruktur zu zentralisieren. Dezentrale Varianten sind prinzipiell auch möglich, führen aber wegen der Vermischung von Verarbeitungs- und Steuerungsaufgaben zu einer komplizierten Struktur (vgl. Abschnitt 3.3.7). Interaktion erfolgt prinzipiell nur zwischen aktiven Elementen eines Systems (hier: elementare Prozesse oder Instanzen). Es gibt drei Abstufungen für Interaktion nach der Position der beteiligten Partner im System, d.h. Interaktion zwischen

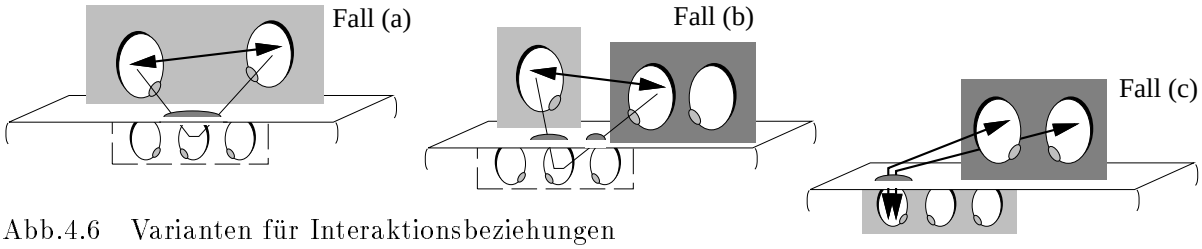


Abb.4.6 Varianten für Interaktionsbeziehungen

Partnern im selben Instanzbereich (Umgebung – Fall (a)), in verschiedenen Instanzbereichen (Fall (b)) oder in unterschiedlichen Schichten (Infrastruktur – Fall (c)). Interaktion wird in allen drei Fällen über die Infrastruktur vermittelt, die dafür entsprechende Mittel bereitstellen muß.

Infrastruktur wirkt in zwei Richtungen auf übergeordnete Instanzbereiche. Da sind zum einen ***Infrastrukturdienste***. Dienste werden als Folgen von Operationen über Protokolle mit Instanzen der Infrastruktur vermittelt und in der Infrastruktur ausgeführt. Dazu muß es für Instanzen Zugangspunkte, d.h. eine Schnittstelle mit der Infrastruktur geben. Dienste sind in Instanzbereichen identifizierbar und müssen explizit aufgerufen werden. Dienste können dabei in zwei Kategorien eingeteilt werden (vgl. Abschnitt 2.2.2). Es gibt

- ***existentielle Dienste***, welche für die Existenz von Elementen und die Verarbeitung in einem Instanzbereich unabdingbar sind, für die:
  - Erzeugung bzw. Beseitigung von Verarbeitungselementen, für die
  - Interaktion von Prozessen bzw. Instanzen und für die
  - globalen Steuerungsaufgaben, oder es handelt sich um
- ***optionale Dienste***, die prinzipiell auch im Instanzbereich angesiedelt sein könnten, aber in der Infrastruktur aus verschiedenen Gründen besser plaziert sind.

Die andere Wirkung von Infrastruktur liegt in den Eigenschaften des ***Ausführungsmodells***, welche implizit die Existenz- und Verarbeitungsgrundlagen für einen Instanzbereich vorgeben. Das Ausführungsmodell wird durch die Art der Elemente und die Eigenschaften der Verarbeitung bestimmt. Für das Ausführungsmodell wird in der generalisierten Architektur keine weitere Aussage getroffen, da hier nur die grundlegenden Elemente und ihre Beziehungen Gegenstand der Betrachtung sind. Eigenschaften des Ausführungsmodells werden erst bei der Spezialisierung zugeordnet (vgl. Abschnitt 4.3).

Für Infrastrukturen kann zusammengefaßt werden, daß deren Hauptzweck darin besteht, durch Verarbeitungsprozesse von Elementen in einer Schicht, höherwertige Elemente für übergeordnete Schichten herzustellen. Die prinzipielle Gleichartigkeit der Elemente und des Geschehens (→ *Verarbeitungsprozesse*) erlaubt die Modellierung der Infrastrukturschicht nach demselben Grundmuster wie bei Instanzbereichen. Das ist die Grundlage für das hier gewählte Vorgehen, ein gleichartiges Strukturmuster rekursiv über jeweils benachbarte Schichten von Instanzbereichen und Infrastrukturen zu legen. Dafür ist allerdings notwendig, die generellen Architekturmerkmale von individuellen Ausführungseigenschaften und Ausprägungen zu abstrahieren, da diese für einzelne Schichten verschieden sind. Über Ausführungseigenschaften und die Ausprägung von Elementen erfolgt auch der notwendige Abbau der Infrastrukturrekursion.

Unterschied: Aggregation – Infrastruktur

Wenn Infrastruktur letztlich auch als Zusammenfassung und Abgrenzung von Verarbeitungselementen (Daten, Prozesse, Instanzen) modelliert wird und von außen als ein abgeschlossenes Teilsystem (→ *Instanz*) erscheint, stellt sich die Frage nach dem grundlegenden Unterschied zwischen Aggregation und der Kapselung von Infrastruktur.
Dieser besteht darin, daß Infrastrukturen nicht nur Instanzen und andere Verarbeitungselemente enthalten, sondern für einen übergeordneten Bereich auch *herstellen*. Die Wirkung einer Infrastruktur ist, neue Instanzbereiche mit *anderen, höherwertigen* Ausführungseigenschaften zu erzeugen. Die Aggregationsrekursion bezieht sich nur auf eine Softwareschicht, und Elemente werden über die Enthaltensein-Relation in Instanzen lediglich nach außen gekapselt. Es entsteht zwar eine neue Steuerungsebene, aber keine neue Schicht im System.

Infrastruktur ist in dieser Betrachtungsweise Aggregation plus das Herstellen einer neuen Systemschicht mit neuen Verarbeitungselementen und Ausführungseigenschaften. Aus diesem Zusammenhang ergibt sich auch die weitreichende Wirkung infrastruktureller Anpassungen für Instanzbereiche, nicht nur in Hinsicht auf neue oder veränderte Dienste, sondern auch bezogen auf Eigenschaften des Ausführungsmodells, um beispielsweise teure Ausführungsmodelle nach Wahl abzurüsten oder gänzlich andere Ausführungsmodelle oder neuartige Dienste herzustellen.

4.2.5 Zusammenfassung der strukturellen Architekturmerkmale

Eine Funktionseinheit wird zu einem System, wenn innere Funktionseinheiten und Elemente abgrenzbar und damit identifizierbar werden und deren Beziehungen und Zusammenwirken beschreibbar sind. Gemäß diesem Grundsatz wurden für den Betrachtungsgegenstand eines ablaufenden Systems die wesentlichen Strukturelemente der Architektur vorgestellt und begründet. Sie sollen hier noch einmal zusammengefaßt werden:

(1) ***Basiselemente*** für Verarbeitung in einer Ebene: (→ Daten, Prozesse, Programme),

(2) Kapselung von Basiselementen zu ***Instanzen*** (→ innere, äußere Steuerungsebene),

(3) Gruppierung von Instanzen zu ***Instanzbereichen*** (→ gleiche Steuerungsebene),

(4) ***Aggregationsrekursion*** von Instanzen (→ Instanzen enthalten innere Instanzen),

(5) ***Systemschichten*** und ***Infrastrukturrekursion***.

Basiselemente *(1)* bilden die strukturelle Grundlage der Architektur. Die anderen Merkmale *(2)–(5)* haben einen *erzeugenden Charakter* für höherwertige Elemente in der Architektur. Durch diese Erzeugungsrelationen entstehen drei orthogonale Prinzipien, ***hierarchische Steuerungsebenen*** in Systemen einzuführen und auf diese Weise Struktur herzustellen:

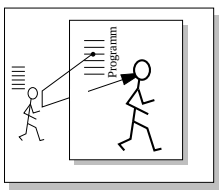
- innere und äußere Steuerungsebenen durch Kapselung von Basiselementen zu Instanzen,
- rekursive innere Steuerungsebenen durch die Aggregationsrekursion von Instanzen,
- Steuerungsebenen durch Systemschichten, die rekursiv jeweils Infrastrukturen für übergeordnete Schichten sind.

Die Gruppierung und Kapselung von Elementen und das Bilden von Hierarchien sind die grundlegenden Mittel für Abstraktion. Durch sie werden neue Elemente mit höherwertigen Funktionen und Diensten und damit höherwertige Steuerungsebenen erzeugt. ***Steuerungsebenen*** *sind* ***Abstraktionsebenen*** *in ablaufenden Systemen.*

Es gibt weitere Prinzipien, die für die Zusammenfassung der strukturellen Architekturmerkmale jedoch nicht vordergründig betrachtet werden. Das sind die

- *Herstellungsrekursion* von Elementen durch Verarbeitungsprozesse anderer Elemente im selben System, welche vorher selbst (rekursiv) herzustellen sind, und die
- *Auflösungsrekursion* von Elementen, d.h., auch Basiselemente werden aus Elementen bzw. dem Wirken von Elementen in anderen Steuerungsebenen gebildet.

Steuerungsebenen werden für *Basiselemente* in der Form wirksam, daß Speicherelemente und Prozesse auf irgendeine Weise selbst erzeugt werden müssen, so daß Daten, Programme und Prozesse zusammenwirken können. Die Auflösungsrekursion von Elementen fällt mit der Infrastrukturrekursion zusammen, wenn Elemente durch Infrastruktur erzeugt werden, z.B. virtueller Speicher oder die Erzeugung von Prozessen auf Basis einer Aktivität.

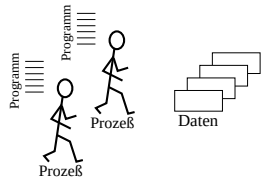


In anderen Fällen paßt diese Gleichsetzung mit der Infrastrukturrekursion nicht. Basiselemente können auch aus sich selbst heraus bestimmte Funktionen haben, ohne daß in der Infrastruktur dafür besondere Vorkehrungen notwendig sind. Die Infrastruktur muß natürlich Verbindungen zu diesen Elementen herstellen und ihre Benutzung erlauben, aber sie ist nicht für die eigentliche Existenz dieser Elemente erforderlich. Die Auflösungsrekursion von Elementen kann somit in beide Richtungen geführt werden. Die Aggregationsrekursion wird aus diesem Grund hier nur auf gekapselte Instanzen und nicht auch auf Basiselemente bezogen.

Für die generalisierte Architektur können als Basiselemente $\mathcal{BE}$ einer Steuerungsebene Daten, Prozesse und Programme festgeschrieben werden.

(1) – Basiselemente:

- ***Basiselemente*** – $\mathcal{BE} = \{ \mathcal{Z}, \mathcal{P}, \mathcal{P}_{rog} \}$
  - *Daten* $\mathcal{Z}$ - Zustände (Werte) von Speicherelementen
  - *Prozesse* $\mathcal{P}$ - Aktivitäten + Steuerungen
  - *Programme* $\mathcal{P}_{rog}$ - Steuerungsinformation für $\mathcal{P}$



Aus Sicht von außen ist ein System von $\mathcal{BE}$ abgeschlossen. Verarbeitungen werden durch innere Prozesse $\mathcal{P}$ ausgeführt. Die Wirkung eines informationsverarbeitenden Systems $\mathcal{IVS}$ ist:

$\mathcal{P}: \{ \mathcal{S}_\mathcal{E} \times \mathcal{W}_\mathcal{E} \} \rightarrow \{ \mathcal{S}_\mathcal{A} \times \mathcal{W}_\mathcal{A} \}$ von außen; bzw. $\{ \mathcal{W}_\mathcal{E} \times \mathcal{Z}_t \} \times \mathcal{P} \rightarrow \{ \mathcal{W}_\mathcal{A} \times \mathcal{Z}_{t+1} \}$ im Inneren.

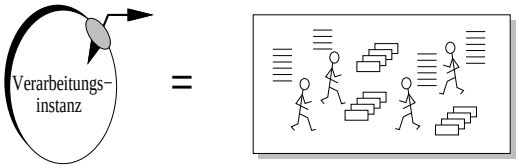
Eine Instanz kapselt primär Basiselemente einer Steuerungsebene und erhält dadurch den Charakter eines informationsverarbeitenden (Teil-) Systems.

(2) – Instanzen:

- ***Instanz*** $\mathcal{I} = \{ \mathcal{BE} \}$

  $\mathcal{I}$ – $\mathcal{IVS}'$ mit

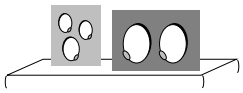
  $\{ \mathcal{Z}, \mathcal{P}, \mathcal{P}_{rog}, \{\mathcal{W}_\mathcal{E}, \mathcal{Z}_t\} \times \mathcal{P} \rightarrow \{\mathcal{W}_\mathcal{A}, \mathcal{Z}_{t+1}\} \}$



Instanzbereiche sind eine (ggf. hierarchische) Gruppierung von Instanzen in derselben Schicht. Die Gruppierung definiert sich anhand von Eigenschaften, welche Gruppenmitgliedern gemeinsam sind, aber zwischen Gruppen unterschieden werden.

(3) – Instanzbereiche:

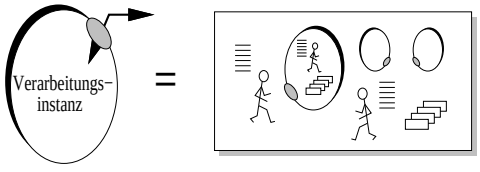
- ***Instanzbereich*** $\mathcal{IB} = \{ \mathcal{I} \} \cup \{ \mathcal{IB}' \}$



Die Aggregationsrekursion von Instanzen widerspiegelt die Eigenschaft der Kapselung autonomer Teilsysteme im Inneren von Instanzen, d.h., Instanzen können neben Basiselementen wiederum Instanzen $\mathcal{I}'$ enthalten. Diese Instanzen befinden sich in derselben Systemschicht wie die umschließende Instanz. Wegen der Kapselung bei Aggregation entstehen mit jeder neuen Aggregationsstufe auch neue Steuerungsebenen innerhalb von Instanzen.

(4) – Aggregationsrekursion von Instanzen:

- ***Instanz*** $\mathcal{I} = \{ \mathcal{BE} \} \cup \{ \mathcal{I}' \}$

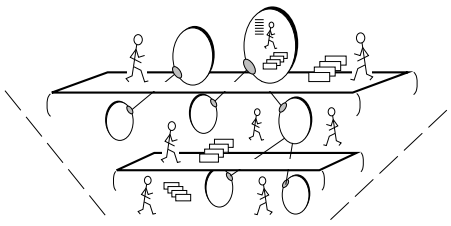


Für Softwaresysteme soll ausdrücklich auch die Schaffung von Infrastrukturschichten berücksichtigt werden. In Abschnitt 4.2.4 wurden Gründe und Wesenszüge von Softwareinfrastrukturen ausführlich erörtert. Eine Softwareinfrastruktur bildet eine Systemschicht, auf der andere Systemschichten aufsetzen. Mit Systemschichten werden ebenfalls Steuerungsebenen im Sinne von Infrastruktur (→ *Dienste*, *Ausführungsmodell*) eingeführt. Durch Verarbeitungsprozesse in der Infrastruktur lassen sich neuartige Elemente und Ausführungseigenschaften für übergeordnete Schichten herstellen. Wegen der prinzipiellen Gleichartigkeit der Elemente und Vorgänge innerhalb der Infrastruktur wird dasselbe Modellierungsprinzip angewandt. Auch hier gibt es Basiselemente (Daten, Programme, Prozesse) und deren mögliche Zusammenfassung zu Instanzen. Der Unterschied zwischen Infrastruktur und der von ihr hergestellten übergeordneten Schicht wird ausschließlich von Ausprägungsvarianten und Ausführungseigenschaften bestimmt.

***(5)** – Systemschichten und Infrastrukturrekursion:*

- ***Systemschicht*** – $\mathcal{L} = \{ \mathcal{BE}, \mathcal{I}, \mathcal{IB} \}$
- ***Infrastrukturrekursion***
  – $\mathcal{L}_i = \mathcal{I}_{nfra} (\mathcal{L}'_{i+1})$

  mit der Wirkung: $\mathcal{L}_i$ stellt $\mathcal{L}'_{i+1}$ her.



Infrastruktur ist folglich auch als ein informationsverarbeitendes (Teil-) System anzusehen, mit der Aufgabe, ein neuartiges übergeordnetes System von Elementen und (Interaktions-) Beziehungen herzustellen. Infrastruktur bekommt dadurch auch den Charakter einer Instanz und unterliegt prinzipiell den gleichen Strukturmerkmalen. Diese Argumentation kann man auch auf ein gesamtes $\mathcal{IVS}$ anwenden, da es ebenfalls nach außen wie eine abgeschlossene Instanz wirkt.

Die Strukturmerkmale der generalisierten Architektur können noch einmal in einer formalen Darstellung zusammengefaßt werden. Für konkrete Ausprägungen der Architektur müssen nicht alle der genannten Mengen tatsächlich Elemente enthalten. Leere Mengen einzelner Merkmale sind für Steuerungsebenen auch notwendig, um die beteiligten Rekursionen abzubrechen.

• ***Basiselemente***	$\mathcal{BE}$	=	$\{ \mathcal{Z}, \mathcal{P}, \mathcal{P}_{rog} \}$
• ***Instanzen***	$\mathcal{I}$	=	$\{ \mathcal{BE} \}^a \cup \{ \mathcal{I}' \}^b$
• ***Instanzbereiche***	$\mathcal{IB}$	=	$\{ \mathcal{I} \}^c \cup \{ \mathcal{IB}' \}^d$
• ***Systemschicht***	$\mathcal{L}$	=	$\{ \mathcal{BE}, \mathcal{I}, \mathcal{IB} \}$
• ***Infrastrukturrekursion***	$\mathcal{L}_i$	=	$\mathcal{I}_{nfra} (\mathcal{L}'_{i+1})$
• ***System***	$\mathcal{IVS}$	=	$\bigcup_{\forall i} \mathcal{L}_i$, $\mathcal{IVS}' = \mathcal{I}$ [e]

[a] Kapselung von Basiselementen zu Instanzen
[b] Aggregationsrekursion von Instanzen
[c] Gruppierung von Instanzen
[d] hierarchische Gruppierung von Instanzen
[e] prinzipielle Gleichsetzung eines $\mathcal{IVS}'$ mit einer ganzheitlich umschließenden Instanz $\mathcal{I}$

Damit sind die strukturellen Merkmale der generalisierten Architektur vorgestellt. Im folgenden geht es um Ausprägungsvarianten und Ausführungseigenschaften für die Spezialisierung der Architektur, um sie auf Maßgaben konkreter Zielsysteme zuschneiden zu können.

4.3 Spezialisierung der Architektur

Bei der Spezialisierung der generalisierten Architektur geht es um die Anpassung der universellen Architekturmerkmale an konkrete Einsatzfälle. Spezialisierung bezieht sich auf zwei Bereiche.

Einerseits müssen nicht mit Notwendigkeit alle Merkmale der generalisierten Architektur für jeden konkreten Einzelfall angemessen und sinnvoll sein, beispielsweise die Existenz mehrstufiger Softwareinfrastrukturen. Auch eine Systemgliederung in Instanzen muß nicht in jedem Fall nützlich sein. Es ist durchaus berechtigt, wenn es lediglich global agierende Prozesse über globalen Daten gibt, oder es gibt gar nur einen Prozeß und höhere Strukturmerkmale fehlen. Diese elementaren Ausprägungen sind sogar notwendig, um die Rekursionen abzubrechen.
Die generalisierte Architektur spannt aufgrund der Rekursionen einen potentiell unendlichen Variantenraum auf. Für eine konkrete Architektur ist daraus eine Variante oder ein Element aus dem Variantenraum auszuwählen. Diese Variante beschreibt für die Infrastrukturrekursion, welche Schichten es in dem Zielsystem geben soll und welche Elemente für die Schichten vorgesehen sind. Für die Aggregationsrekursion von Instanzen wird deren Tiefe festgelegt. Das Ableiten einer konkreten Architektur im Sinne der Auswahl der konkreten Gestalt der Architektur soll als ***Ausprägung der Architektur*** bezeichnet werden.
Die andere Dimension bezieht sich auf die ***Zuordnung von Ausführungseigenschaften*** für die Elemente einer spezialisierten Architektur. Bei der generalisierten Architektur ging es um die Beschreibung "reiner" Strukturelemente. Ausführungseigenschaften werden durch *Dienste der Infrastruktur* und das *Ausführungsmodell* für Elemente in einer Systemschicht bestimmt. Damit werden gleichzeitig die Anforderungen an die dafür notwendige Infrastruktur festgelegt.
In der strikten Trennung genereller Architekturmerkmale von Ausprägungen und Ausführungseigenschaften für einzelne Schichten und für eine Bandbreite von Systemen liegt das Neue in der Herangehensweise. Bislang sind in vielen Systemen beide Aspekte vermischt, so daß bereits in der Architektur Ausprägungs- und Ausführungsaspekte implizit mit fixiert sind, die eine Übertragung in andere Schichten oder auf andere Zielsysteme stark einschränken oder verhindern.

Die Architektur von *BirliX* ist ein Beispiel für eine Architektur mit feststehender Variantenbildung [Här90a]. Das System besteht aus vernetzen Maschinen (→ *Hosts*), mit lokalen Kernen als Infrastruktur (→ *Nucleus*), auf denen es Instanzen (→ *Teams*) gibt. Innerhalb der *Teams* gibt es parallele Prozesse (→ *Agents, Natives*), die global über den Daten der *Teams* operieren. Es gibt keinen Spielraum in dieser Architektur, weitere Schichten hinzuzufügen oder eine Schichtenstruktur innerhalb des (ablaufenden) Kerns einzuführen. Man kann dies zwar prinzipiell im System implementieren, aber in der Architektur ist dieser Fall nicht vorgesehen.
Und es kommt ein zweites, schwerwiegenderes Problem hinzu, daß in der Architektur auch Ausführungseigenschaften der Elemente und generelle Mechanismen mit festgeschrieben sind: prinzipiell *RPC* für Inter-*Team*-Kommunikation, separate Adreßräume für *Teams*, Persistenz aller *Teams* usw. Es handelt sich folglich um eine spezielle Architektur für das konkrete System *BirliX*. Sie ist speziell auf die Anforderungen dieses Systems zugeschnitten. Das war auch die Intention der Autoren, aber sie ist dadurch nicht auf andere Zielbereiche übertragbar[9].

Es könnten andere Beispiele für ähnliche Herangehensweisen genannt werden. Anwendungsanpaßbarkeit kann nur im Rahmen der jeweils fest vorgegebenen Architektur umgesetzt werden. Wird dieser Rahmen verlassen, und die zunehmende Diversifizierung in (Betriebs-) Systemen wird dazu führen, sind neue Wege zu beschreiten. Deshalb soll hier ein solcher Weg gegangen werden, universelle Strukturmerkmale von Ausprägungsvarianten und Ausführungseigenschaften zu trennen und erst beim Ableiten einer speziellen Architektur für ein konkretes Zielsystem zuzuordnen. In dieser Vorgehensweise liegt auch der neue *methodischer Ansatz* der hier vorgestellten Architektur.

[9] etwa "*BirliX* für die Kaffeemaschine" – Anmerkung auf dem Treffen der *BirliX*-Gruppe, März 1992.

4.3.1 Ausprägungsvarianten der Architektur

Bei den Ausprägungsvarianten der Architektur geht um die Auswahl einer geeigneten Variante aus dem möglichen Variantenraum der generalisierten Architektur. Die strukturelle Gliederung des Systems erfolgt anhand von Steuerungsebenen, welche durch die Kapselung von Basiselementen zu Instanzen, die Aggregationsrekursion und die Infrastrukturrekursion gebildet werden können. Für ein Zielsystem ist eine Variante mit sinnvollen Steuerungsebenen auszuwählen:

- **pro System** – $\mathcal{IVS}$:
  - Wieviele Schichten gibt es ($\rightarrow$ Tiefe der Infrastrukturrekursion, $|\mathcal{L}_i| > 1$) ?
- **pro Schicht** – $\mathcal{L}_i$:
  - Gibt es ausschließlich Basiselemente (globale Daten, global agierende Prozesse) ?
  - Gibt es auch Instanzen $\mathcal{I}_j$?
  - Gibt es ausschließlich Instanzen $\mathcal{I}_j$?
  - Gibt es Instanzbereiche $\mathcal{IB}$?
- **pro Instanz** – $\mathcal{I}_j$:
  - Gibt es im Inneren ausschließlich Basiselemente?
  - Gibt es auch innere Instanzen $\mathcal{I}'_{j,k}$ ($\rightarrow$ Aggregationsrekursion) ?
  - Gibt es ausschließlich innere Instanzen $\mathcal{I}'_{j,k}$ ($\rightarrow$ Aggregationsrekursion) ?
  - [Gibt es auch innere Schichten $\mathcal{L}'_{j,k}$ ($\rightarrow$ innere Infrastrukturrekursion) ?] [10]

Ausprägungsvarianten entstehen dadurch, daß bestimmte Merkmale nicht vorhanden sind. Die Mengen für die betroffenen Merkmale sind leer. Es gibt außerdem Regeln, welche Mengen in Abhängigkeit anderer Mengen leer sein können, damit dennoch ein sinnvolles System entsteht.

Um die Vorgehensweise zu verdeutlichen, sind in Abbildung 4.7 drei mögliche Szenarien dargestellt. In allen drei Varianten wird lediglich eine Systemschicht betrachtet. Bei der Ausprägung in Variante A gibt es ausschließlich Basiselemente. Die Mengen für Instanzen $\mathcal{I}$, Instanzaggregation $\mathcal{I}'$ und Instanzbereiche $\mathcal{IB}$ sind leer. Bei Variante B gibt es dagegen ausschließlich Instanzen ohne Aggregation ($\mathcal{BE} = \mathcal{I}' = \emptyset$, $\mathcal{IB} = \{ \mathcal{I}_i \}$). Variante C ist eine Mischform aus A und B.

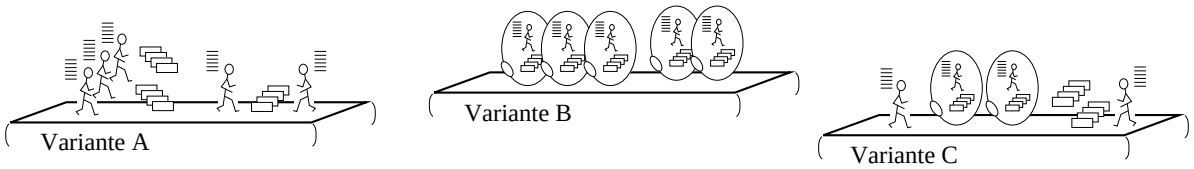


Abb.4.7 Ausprägungsvarianten für Basiselemente und Instanzen in einer Systemschicht

Damit wird deutlich, auf welchem Prinzip die Gestaltung von Ausprägungsvarianten beruht. Für den Anwendungsbereich von *Unix* kommt nur eine Ausprägung in Variante B in Frage, es gibt ausschließlich *Unix*-Prozesse. Auch für verteilte Systeme unter *Unix* gilt diese Struktur, nur daß in diesem Fall die Infrastruktur nicht allein den lokalen Kern umfaßt, sondern auch die maschinenübergreifende Infrastruktur des Netzes einschließt.
Ein System unterliegt in gewisser Weise einer extremen Ausprägung, wenn es global nur einen einzigen Prozeß mit einem Programm und einer Menge von Daten gibt. Das ist aber gerade der typische Fall für viele Anwendungssysteme und sogar auch für monolithische – *single-threaded* – Betriebssysteme.

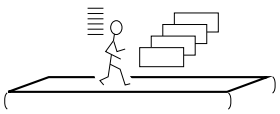


Abb.4.8 Extreme Ausprägung als 1–Prozeß–System

[10] wegen der prinzipiellen Gleichsetzung von $\mathcal{IVS}$ und $\mathcal{I}$ in der generalisierten Architektur (vgl. Seite 88)

- **Varianten pro Systemschicht**

Zur Beschreibung von Varianten sollen die beteiligten Mengen dahingehend klassifiziert werden, ob sie kein, genau ein oder beliebig viele (n, $n > 1$) Elemente enthalten können ($\mathcal{K}$ – Mengen).

	- Daten	$\mid \mathcal{Z} \mid$	$\rightarrow$	$\mathcal{KZ}$	$=$	$\{0,\ 1,\ n\}^a$
○ Basiselemente $\mathcal{BE}$	- Prozesse	$\mid \mathcal{P} \mid$	$\rightarrow$	$\mathcal{KP}$	$=$	$\{0,\ 1,\ n\}$
	- Programme	$\mid \mathcal{P}_{rog} \mid$	$\rightarrow$	$\mathcal{KP}_{rog}$	$=$	$\{0,\ 1,\ n\}^b$
○ Instanzen $\mathcal{I}$		$\mid \mathcal{I} \mid$	$\rightarrow$	$\mathcal{KI}$	$=$	$\{0,\ 1,\ n\}$
○ Instanzbereiche $\mathcal{IB}$		$\mid \mathcal{IB} \mid$	$\rightarrow$	$\mathcal{KIB}$	$=$	$\{0,\ 1,\ n\}$

[a] 0 – kein globaler Zustand, 1 – globaler Zustand $\forall\ \mathcal{P}_i \in \mathcal{P}$, $n - \forall\ \mathcal{P}_i\ \exists$ je ein $\mathcal{Z}_i \in \mathcal{Z}$
[b] 1 – $\forall\ \mathcal{P}_i \in \mathcal{P}\ \exists$ nur ein globales $\mathcal{P}_{rog}$, $n - \forall\ \mathcal{P}_i \in \mathcal{P}\ \exists$ je ein $\mathcal{P}_{rog_i} \in \mathcal{P}_{rog}$

Für die Auswahl von Elementen für sinnvolle Steuerungsebenen gelten dabei einige Regeln:

$\mathcal{KZ} = 0 \ \vee\ \mathcal{KP}_{rog} = 0 \ \rightarrow\ \mathcal{KP} = 0$.	Prozesse kann es nur geben, wenn es $\mathcal{Z}$ und $\mathcal{P}_{rog}$ gibt.
$\mathcal{KP} = 0 \ \rightarrow\ \mathcal{KI} \neq 0$.	Wenn es keine Prozesse gibt, muß es Instanzen geben.
$\mathcal{KI} = 0 \ \rightarrow\ \mathcal{KP} \neq 0$.	Wenn es keine Instanzen gibt, muß es Prozesse geben.
$\mathcal{KI} \neq 0 \ \rightarrow\ \mathcal{KIB} \neq 0$.	Wenn es Instanzen gibt, gibt es mindestens einen $\mathcal{IB}$.

Für die in den Abbildungen 4.7 und 4.8 gezeigten Beispiele kann man ebenfalls die jeweiligen Bedingungen formulieren.

Abbildung 4.7, Variante A:	$\mathcal{KP} = \mathcal{KP}_{rog} = \mathcal{KZ} = n,\quad \mathcal{KI} = \mathcal{KIB} = 0$.
Abbildung 4.7, Variante B:	$\mathcal{KP} = \mathcal{KP}_{rog} = \mathcal{KZ} = 0,\quad \mathcal{KI} = n,\quad \mathcal{KIB} = 1$.
Abbildung 4.7, Variante C:	$\mathcal{KP} = \mathcal{KP}_{rog} = \mathcal{KZ} = n,\quad \mathcal{KI} = n,\quad \mathcal{KIB} = 1$.
Abbildung 4.8:	$\mathcal{KP} = \mathcal{KP}_{rog} = \mathcal{KZ} = 1,\quad \mathcal{KI} = \mathcal{KIB} = 0$.

Eine reine Instanzstruktur innerhalb einer Systemschicht, wie sie auch dem *Client-Server-Modell* zugrunde liegt, ist folglich eine spezielle Ausprägungsvariante der generalisierten Architektur (Abbildung 4.7, Variante B). Andere Ausprägungsvarianten sollen aber nicht ausgeschlossen werden, auch wenn Instanzen hier wegen ihrer vorteilhaften Eigenschaften bevorzugt werden.

- **Komposition von Systemschichten zu einem Gesamtsystem**

Für jede Systemschicht ist eine der gezeigten Varianten auszuwählen und in der Komposition zu einem Gesamtsystem zusammenzufügen.
Hierbei sei nochmals auf die erweiterte Betrachtung von Instanzen in der generalisierten Architektur verwiesen, daß ein gesamtes informationsverarbeitendes System als eine ganzheitlich umschließende Instanz aufgefaßt werden kann. Auch die Umkehrung gilt, eine innere Instanz als ein inneres $\mathcal{IVS}'$ aufzufassen, in dem wiederum alle Merkmale der generalisierten Architektur anwendbar sind, also eine innere Gliederung in $\mathcal{BE}'$, $\mathcal{I}'$ und auch in ***innere Schichten*** $\mathcal{L}'_j$.
Aus dieser Gleichsetzung ($\mathcal{I} = \mathcal{IVS}'$) entsteht eine dritte, den anderen hier betrachteten Rekursionen – *Aggregations-* und *Infrastrukturrekursion* – übergeordnete Rekursion.
Aber daraus folgt kein neues Prinzip, denn diese Rekursion kann auf eine Kombination der beiden anderen Rekursionen zurückgeführt werden und muß deshalb nicht gesondert betrachtet werden. Der Ansatz ist, eine Instanz in einer Systemschicht separat als $\mathcal{IVS}'$ zu betrachten und eine gleichwertige Gliederung anhand der Architekturdefinition für dieses $\mathcal{IVS}'$ anzuwenden.

- **Typische Ausprägungsvarianten**

Zur Begründung der praktischen Relevanz der gezeigten Ausprägungsvarianten sollen einige typische Systeme eingeordnet werden. Instanzen, Aggregationsrekursion und Infrastrukturrekursion sind die drei Erzeugungsmöglichkeiten für Steuerungsebenen in der generalisierten Architektur. Bei der ***vertikalen Systemgliederung*** werden die Systemschichten als eine Form der Steuerungsebenen bestimmt. Für jede Systemschicht gibt die ***horizontalen Systemgliederung*** die Auflösung in innere Basiselemente, Instanzen, Instanzbereiche und auch innere Schichten an. Über die Aggregationsrekursion lassen sich auch innere Instanzen weiter rekursiv auflösen. In realen Systemen gibt es in der Regel wenige derartige Ebenen, sowohl für die Aggregations- als auch für die Infrastrukturrekursion. In Tabelle 4.9 sind vier Vertreter realer Systeme in das angegebene Schema zur Charakterisierung von Ausprägungsvarianten eingeordnet.

Beispiel	vertikale Gliederung (***Schichten***)	horizontale Gliederung (***Aggregation***) 1.Ebene	2.Ebene
ursprüngliches *Unix*, [RT74]	*Anwendungen:* →	Instanzen → (*Unix*-Prozesse)	Basiselemente ($\mathcal{KP}=\mathcal{KZ}=\mathcal{KP}_{rog}=1$)
	Kern: →	Basiselemente ($\mathcal{KP}=\mathcal{KZ}=\mathcal{KP}_{rog}=1$)	
Unix mit reentrantem Kern	*Anwendungen:* →	Instanzen → (*Unix*-Prozesse)	Basiselemente ($\mathcal{KP}=\mathcal{KZ}=\mathcal{KP}_{rog}=1$)
	Kern: →	Basiselemente ($\mathcal{KP}=n$, $\mathcal{KZ}=\mathcal{KP}_{rog}=1$)	
Unix mit multi-threaded Prozessen	*Anwendungen:* →	Instanzen → (*Unix*-Prozesse)	Basiselemente ($\mathcal{KP}=n$, $\mathcal{KZ}=\mathcal{KP}_{rog}=1$)
	Kern: →	Basiselemente ($\mathcal{KP}=n$, $\mathcal{KZ}=\mathcal{KP}_{rog}=1$)	
μ-Kern mit *In-Kernel Servern*	*Anwendungen:* →	Instanzen → (multi-threaded)	Basiselemente ($\mathcal{KP}=n$, $\mathcal{KZ}=\mathcal{KP}_{rog}=1$)
	Kern-Prozesse: →	Instanzen → (single-threaded)	Basiselemente ($\mathcal{KP}=\mathcal{KZ}=\mathcal{KP}_{rog}=1$)
	μ-Kern: →	Basiselemente ($\mathcal{KP}=\mathcal{KZ}=\mathcal{KP}_{rog}=1$)	

Abb.4.9 Beispiele für typische Ausprägungsvarianten in realen Systemen

Das primäre Merkmal ist die vertikale Gliederung in Schichten, wobei jede Schicht in der horizontalen Gliederung in ihre Grundbestandteile zerlegt wird (1. Auflösungsebene). Gibt es Instanzen, so können diese nach der Aggregationsrekursion weiter aufgelöst werden (2. Auflösungsebene). Für die genannten Beispiele bricht die Aggregationsrekursion nach zwei Stufen, d.h. mit der inneren Gliederung von Instanzen ab. Der Abbruch erfolgt bei elementaren Basiselementen. Die Infrastrukturrekursion bricht ebenfalls nach zwei und im Fall der μ-Kerne mit *In-Kernel Servern* nach drei Stufen ab. Für den *CHEOPS*-Kern wird eine vierte Infrastrukturschicht für Instanzen zur Behandlung von Unterbrechungen hinzugefügt (vgl. Kapitel 5).

4.3.2 Ausführungseigenschaften – Dienste der Infrastruktur

Dienste der Infrastruktur wurden in Abschnitt 2.2.2 in zwei Kategorien eingeteilt.

- *Existentielle Dienste* sind für den Ablauf von Anwendungen in der Infrastruktur notwendig. Über diese Dienste wird ein Minimum an Funktionen als Existenz- und Arbeitsgrundlage für die Elemente der übergeordneten Schicht bereitgestellt:
  - Erzeugen (Herstellen) und Beseitigen von Elementen,
  - Ablaufsteuerung und Interaktion.
- *Optionale Dienste* bieten die Möglichkeit, auch anwendungs- und systembezogene Aufgaben in der Infrastruktur zu zentralisieren. Erweiterbare Betriebssysteme zeichnen sich gerade durch diese Eigenschaft aus.

Eine Infrastruktur ist als Ganzes nach außen abgegrenzt, kapselt die innere Verarbeitung und führt (Infrastruktur-) Dienste aus. Infrastruktur wirkt damit selbst wie eine Instanz. Daher können die Aussagen aus Abschnitt 4.2.2 zu Diensten von Instanzen auch auf Infrastrukturen übertragen werden. Das betrifft die Merkmale:

- Dienste, Operationen, Protokolle,
- Schnittstellen und Schnittstellendefinitionen und das
- Identifikationsschema.

Zur Ausführung von Diensten sind mit der Infrastruktur Protokolle abzuwickeln. Das Identifikationsschema gibt an, ob die Infrastruktur in der übergeordneten Schicht als globaler Interaktionspartner, d.h. als "eine Instanz" auftritt oder ob auch innerhalb der Infrastruktur dienstspezifische Interaktionspartner ($\rightarrow$ *Aggregation innerer Instanzen*) nach außen bekannt und damit identifizierbar sind. Auch für Schnittstellen und Schnittstellendefinitionen (Dienstdefinitionen) lassen sich die Aussagen von Instanzen auf Infrastruktur übertragen.

4.3.3 Ausführungseigenschaften – Ausführungsmodell

Neben Diensten stellt Infrastruktur auch das *Ausführungsmodell* für Elemente und Abläufe in der übergeordneten Schicht her. Das ist die zweite Facette von Infrastruktur, die oft nicht die angemessene Beachtung erfährt, weil der Schwerpunkt bei Diensten und Schnittstellen liegt.
Die Ausprägungsvarianten der generalisierten Architektur bieten bereits eine große Variantenvielfalt, die aber noch formal beschreibbar ist (vgl. Seite 88). Auch Infrastrukturdienste lassen sich einfach erfassen und beschreiben. Für Ausführungsmodelle gibt es aber einen weitaus größeren Variantenreichtum, der sich nicht mehr so leicht einordnen und klassifizieren läßt. Eine vollständige Beschreibung aller Varianten ist unmöglich und wäre sicher auch unpraktisch. Zum Vergleich der Problemklasse kann man vielleicht auf die formale Beschreibung von Semantik verweisen, die ebenfalls nicht praktikabel und schon gar nicht vollständig für reale Anwendungsszenarien möglich ist. Aus diesem Grund soll hier eine informale Darstellung einiger typischer Grundvarianten für Ausführungseigenschaften angegeben werden, die sich an Beobachtungen in realen Systemen orientiert. Weil das Ausführungsmodell durch eine Infrastruktur hergestellt wird, bezieht sich die Darstellung auf jeweils genau eine Softwareschicht in einem System.
Für das Ausführungsmodell lassen sich grob zwei Kategorien angeben, welche die Elemente und Beziehungen für Verarbeitungsprozesse wiedergeben:

- die ***Art von Elementen*** ($\mathcal{BE}$: $\mathcal{Z}$, $\mathcal{P}$, $\mathcal{P}_{rog}$ und $\mathcal{I}$, $\mathcal{IB}$) und
- die ***Wirkung von Prozessen*** ($\mathcal{P}$, $\mathcal{I}$, Ablaufsteuerung, Wirkungsbereich).

• Die Art von Elementen

Als Elemente einer Schicht wurden *Basiselemente* $\mathcal{BE}$ (Daten $\mathcal{Z}$, Prozesse $\mathcal{P}$, Programme $\mathcal{P}_{rog}$) und *Instanzen* $\mathcal{I}$ bzw. $\mathcal{IB}$ eingeführt. Im Ausführungsmodell werden die Existenzgrundlagen für diese Elemente angegeben, welche dann auch die *"Art dieser Elemente"* ausmachen.

- ***Installation und Erzeugung von Elementen, Repräsentation:***

  Damit Softwareelemente in einer Infrastruktur hergestellt werden können, sind entsprechende Beschreibungen (→ *Typbeschreibungen*) notwendig, die entweder a priori in der Infrastruktur enthalten sind oder während des Ablaufs geladen werden. Sind Typbeschreibungen ladbar (→ *Installation*), so können neuartige Elemente in ein System eingebracht werden. Die Herstellung der Typbeschreibungen ist Aufgabe und Gegenstand der Softwareherstellung[11].
  Bei der Erzeugung eines Elements wird in der Infrastruktur eine *Repräsentation* aus Elementen der Infrastruktur hergestellt, welche einen Zustand dieses Elements in der Infrastruktur bewirken. Herstellung ist ein aktiver Verarbeitungsprozeß in der Infrastruktur, dessen Steuerung durch die Interpretation der Typbeschreibung erfolgt.
  Es lassen sich prinzipiell beliebige Elemente eines Typs oder einer Klasse herstellen. Im Ausführungsmodell kann aber auch eine *Elementquantifizierung* festgelegt sein, daß:

  - *genau ein Exemplar* [12] eines Prozesses, eines Datenelements oder einer Instanz,
  - *beliebig viele Exemplare* feststehender oder variabler Anzahl mit einem
  - *globalen oder beschränkten* Wirkungsbereich existieren können.

  Für *C++* ist es eine grundlegende Eigenschaft des Ausführungsmodells, daß es genau einen Prozeß gibt. Die Differenzierung nach dem eingeschränkten Wirkungsbereich soll erfassen, daß Teilmengen von Prozessen nur jeweils Teilmengen von Zuständen beeinflussen können. Daran wurde in Abschnitt 3.5 auch das Kapselungsprinzip für Instanzen festgeschrieben.

- ***Lebensdauer von Elementen:***

  Die Existenz eines Elements beginnt mit der Erzeugung und endet mit der Beseitigung seiner Repräsentation in der Infrastruktur. Bezogen auf die Ablaufzeit eines Systems kann man die Lebensdauer von Elementen in drei Klassen einordnen:

  - *permanent* - ein Element existiert während der gesamten Ablaufzeit des Systems,
  - *temporär* - ein Element wird innerhalb der Ablaufzeit zu einem beliebigen Zeitpunkt dynamisch erzeugt und spätestens am Ende wieder beseitigt,
  - *persistent* - ein Element existiert über die Ablaufzeit des Systems hinaus.

  Für Prozesse und Instanzen soll Persistenz so verstanden werden, daß sie wiederanlauffähig sind, wofür entsprechende (persistente) Repräsentationen aufzubewahren sind (→ *BirliX*).

- ***Örtliche Bindung:***

  Auch die örtliche Bindung von Elementen ist während ihrer Lebensdauer an die Repräsentation der Elemente in der Infrastruktur gebunden. Man kann zwei Varianten für diese Kategorie angeben, ob eine Repräsentation und damit ein Element:

  - *fest* oder *migrierbar* im System ist, oder ob die Repräsentation eines Elements
  - *lokal*, *verteilt*, *repliziert,* also einmalig oder mehrfach im System existiert.

[11] Als Typbeschreibungen werden hier ladbare Module aufgefaßt (z.B. *Unix: a.out*).
[12] nach [Gam94] → *"singleton"* (hier aber in einer verallgemeinerten Bedeutung)

- ***Identifikation:***

  Das Identifikationschema ist in der Regel auch an die Repräsentation eines Elements geknüpft. In Abschnitt 4.2.2 wurden Merkmale des Identifikationschemas für Instanzen angegeben. Sie gelten im Prinzip für beliebige Elemente in einem System:

  - *statische Merkmale der Beschreibung* Typinformation,
  - *elementbezogene Merkmale* eines Exemplars im ablaufenden System,
  - *verarbeitungsbezogene Merkmale* zur Identifikation einer Dienst- oder Operationsausführungen durch $\mathcal{P}$ oder $\mathcal{I}$.

  Auf dem elementaren Identifikationschema basieren letztendlich auch die Merkmale für *Benennung*, *Adressierung* und *Lokalisierung*, wenn es diese für eine Steuerungsebene gibt.

- **Die Wirkung von Prozessen**

Neben der *"Art von Elementen"*, liegt der zweite wesentliche Aspekt des Ausführungsmodells in der *"Art der Verarbeitung"* oder der Ausprägung von Verarbeitungsprozessen. Zwischen beiden Bereichen bestehen enge Wechselbeziehungen. Elemente müssen natürlich für bestimmte Verarbeitungsformen ausgelegt sein und bewirken andererseits auch eine bestimmte Form der Verarbeitung. Unter diesem Blickwinkel sollen vier Bereiche diskutiert werden, welche die *"Art von Verarbeitung"* in einem System näher charakterisieren:

- *die "Art von Prozessen" – Prozeßmodell,*
- *Interaktion und globale Ablaufabhängigkeiten,*
- *der Wirkungsbereich von Prozessen* und
- *Schutzeigenschaften.*

Ein (Verarbeitungs-) Prozeß ist eine Folge von Operationen, die entweder direkt durch eine gesteuerte Aktivität oder durch eine Instanz ausgeführt werden. Beide Elemente wurden als aktive, verarbeitende Elemente für die Architektur eingeführt. Über Instanzen konnten innere Prozesse und damit innere Verarbeitungen nach außen gekapselt werden. Auf diese Weise wurde eine Art der Steuerungsebenen erklärt (vgl. Seite 86). Prozesse wirken immer in einer Steuerungsebene, die gleichzeitig vorgibt, auf welche Weise Operationen ausgeführt werden, wie die Steuerung der Prozesse erfolgt und auf welche Art Aktivität erzeugt wird. Durch die Art dieser Vorgänge wird das Ausführungsmodell für Verarbeitungsprozesse bestimmt.

- ***Die "Art von Prozessen" – Prozeßmodell:***

  Für das Ausführungsmodell von Prozessen wurden oben bereits einige Merkmale bei der Charakterisierung der *"Art von Elementen"* genannt. Ein Prozeß läßt sich als Basiselement der Architektur natürlich in diese Merkmale einordnen:

  - *Erzeugung* → Repräsentation, Elementquantifizierung ($n = 1$, $n \geq 1$),
  - *Lebensdauer* → permanent, temporär, persistent,
  - *Örtliche Bindung* → fest oder migrierbar – lokal, verteilt oder repliziert,
  - *Identifikation* → Merkmale der Beschreibung ($\mathcal{Z}$, $\mathcal{P}_{rog}$), des Prozesses (z.B. *pid*) oder der Verarbeitung (Operationen, mögliche Sperren u.a.).

  Persistenz, Verteilung und Migrierbarkeit wurden für Prozesse auf Repräsentationen und Wiederanlauffähigkeit bezogen. Unter replizierten Prozessen kann man verstehen, daß mehrere Prozesse örtlich getrennt gleiche Operationsfolgen über gleichen Daten ausführen.

Die *"Art von Prozessen"* wird vor allem durch die folgenden drei Kriterien bestimmt:

- *Elementquantifizierung*, es gibt - *genau einen Prozeß* oder
  - *beliebig viele Prozesse (fest, variabel)*,
- *Erzeugung der Aktivität(en)* für Prozesse $\mathcal{P}$ auf Basis von Aktivitätsträgern AT :
  - $1\ AT \rightarrow 1\ \mathcal{P}$
  - $1\ AT \rightarrow n\ \mathcal{P}$ (*)
  - $n\ AT \rightarrow n\ \mathcal{P}$
  - $n\ AT \rightarrow m\ \mathcal{P}$ (* bei $n < m$).

  Im ersten Fall wird ein Prozeß auf Basis eines Aktivitätsträgers hergestellt. In den mit (*) markierten Fällen müssen sich mehr Prozesse in weniger Aktivitätsträger teilen. Das erfolgt durch - *konkurrierendes* (Entzug in der Infrastruktur)
  oder - *kooperierendes* (durch $\mathcal{P}$ selbst veranlaßt)
  Umschalten der Aktivität von AT zwischen Steuerungen (vgl. Abschnitt 3.3.5). Die Art der Umschaltung und die Auswahl des Folge-Prozesses (→ *Scheduling*) bestimmen entscheidend das Prozeß- und damit das gesamte Ausführungsmodell.
- *Erzeugung der Steuerung*, Installation der Steuerungsinformation (Programm) :
  - fest programmiert, nicht änderbar,
  - beim Start einmalig ladbar, nicht änderbar,
  - beliebig ladbar, änderbar, erweiterbar,
  - durch $\mathcal{P}$ selbst änderbar.

Gibt es in einer Steuerungsebene a priori nur genau einen Prozeß, so entfallen alle Fragen zur Koordination und Ablaufsteuerung. Dieses Modell liegt sequentiellen Programmiersprachen und den daraus resultierenden Ablaufsystemen zugrunde. Für E/A-Operationen muß dieser Prozeß aber beispielsweise mit "äußeren" Geräte- und Treiberprozessen koordiniert werden. Damit die Sequentialität erhalten bleibt, werden blockierende (→ *synchrone*) Operationen benutzt.
Gibt es in einem System mehrere Prozesse, ist es ein wesentliches Merkmal des Ausführungsmodells, ob Prozesse mit mehreren Aktivitätsträgern "echt" parallel zueinander arbeiten, ob durch preemptives Umschalten "Pseudoparallelität" hergestellt wird oder ob Prozesse durch wechselseitige Abgabe des Aktivitätsträgers kooperieren. Alle drei Szenarien bewirken verschiedene Ablaufformen, die sich bis auf die Ebene der Programmierung auswirken. Im Fall kooperativer Zusammenarbeit von Prozessen gibt es keine kritischen Abschnitte, wenn die Programme korrekt sind. Bei preemptiven Umschaltungen konkurrierender Prozesse mit einem Aktivitätsträger können kritische Abschnitte in Prozessen z.B. dadurch koordiniert werden, daß durch eine Sperre im Umschaltmechanismus in der Infrastruktur Umschaltungen unterbunden werden. Bei mehreren Aktivitätsträgern ist dieses Prinzip natürlich nicht mehr anwendbar (z.B. bei symmetrischen Multiprozessormaschinen). Der Wechsel des Ablaufmodells war einer der Gründe, weshalb der Übergang zu Multiprozessormaschinen für *Unix* mit so vielen Schwierigkeiten behaftet war. *Unix* war zwar von Anfang an ein *Multiprocessing-System*, aber auf Basis einer CPU konnten kerninterne Koordinierungsaufgaben einfach mit Interruptsperren gelöst werden (die oft verwendeten `splx`-Routinen in den *Unix*-Kern-Quellen). Damit wurde die Ursache für Umschaltungen kurzzeitig ausgeschaltet, so daß kein anderer Prozeß aktiv werden konnte. Bei mehreren CPU's entfällt diese Annahme und erzwingt umfangreiche Korrekturen im Kern. Damit Interruptsperren vorerst weiter benutzt werden konnten, war in den ersten Multiprozessor-*Unix*-Systemen der gesamte Kern-Bereich für Prozesse exklusiv. Auch im Anwendungsbereich stößt man für die verschiedenen *multi-threading*-Modelle auf die gleiche Problematik (→ *reentrante Bibliotheken für kernunterstützte Threads*). Fragen dieser Art werden in [Schim94] näher behandelt.

Diese Darstellung zeigt die immense Bedeutung des Prozeßmodells für Anwendungs- und Systembereiche. Auf die verschiedenen zeitlichen Ablaufformen (→ *Echtzeit, kontinuierliche Ströme* [Wolf95, SteiN95] u.a.) wurde dabei an dieser Stelle noch gar nicht eingegangen.

- ***Interaktion und globale Ablaufabhängigkeiten:***

  Beim Ablauf von Verarbeitungsprozessen in einer Steuerungsebene bestehen vielfältige Abhängigkeiten für den Ablauf, die Bestandteil des Ausführungsmodells sind:

  - *explizite Abhängigkeiten* → *Interaktion:* Koordination, Kooperation,
  - *implizite Abhängigkeiten* → *Prozeßmodell:* Erzeugung und Steuerung.

  Explizite Ablaufabhängigkeiten resultieren aus Anforderungen von Anwendungen für Koordination und Kooperation von Prozessen. Zum Ausführungsmodell gehören die ***Mittel***, welche eine Infrastruktur für diese Zwecke bereitstellt.

  Die andere Art impliziter Ablaufabhängigkeiten ist durch das Prozeßmodell bedingt, wenn es beispielsweise zu *ready*-Wartezeiten kommt. Sie schlagen sich in verzögerten Reaktionszeiten bei Unterbrechungen oder beim Empfang von Nachrichten nieder. Bei konkurrierenden Prozeßmodellen ist der zeitliche Ablauf in der Regeln nicht bestimmt, wenn es mehr arbeitsbereite Prozesse als Aktivitätsträger gibt.
  In diesen Situationen entstehen Konflikte (→ *implizite Ablaufabhängigkeiten*), die durch jeweilige Zuteilungsstrategien (→ *Scheduling*) nicht widerspruchsfrei aufgelöst werden können. Echtzeit- oder Multimediasysteme stellen hierbei besondere Anforderungen, die sich nicht so leicht in die Ablaufmodelle universeller Betriebssysteme integrieren lassen.
  Eine dritte Facette des Ausführungsmodells ist, ob Ablaufabhängigkeiten in der Ebene der Prozesse sichbar und eventuell in gewissen Grenzen beeinflußbar sind.

- ***Der Wirkungsbereich von Prozessen:***

  Der Wirkungsbereich $\mathcal{WB}_i$ eines Prozesses i umfaßt die Zustandsmenge, in welcher von diesem Prozeß Zustandsübergänge bewirkt werden können. An der Abgrenzung des Wirkungsbereichs innerer Prozesse wurde auch das Kapselungsprinzip bei Instanzen erklärt.
  Der Begriff Wirkungsbereich trifft den Sachverhalt besser als der Begriff *Adreßraum*, der in der Literatur an vielen Stellen für die hier beschriebenen Merkmale verwendet wird. Ein Adreßraum bezieht sich immer auf Hauptspeicher, der von einer CPU direkt oder über eine Umsetzung durch eine *MMU* adressiert werden kann. Prozesse in anderen aktiven Elementen, adreßraum- oder maschinenübergreifende Verarbeitungsprozesse oder durch Prozesse bewirkte Zustandsübergänge in Nicht-Hauptspeicher–Elementen, z.B. auch in Dateien, werden durch die Beschreibung von Adreßräumen nicht erfaßt. Adreßräume begrenzen natürlich auch den Wirkungsbereich von "CPU-Prozessen". Sie sind zwar eine typische, aber nicht die einzig mögliche Abgrenzungsform. Deshalb soll hier auf Wirkungsbereiche von Prozessen verallgemeinert werden. Man kann folgende Varianten für Beziehungen zwischen Prozessen und getrennten bzw. gemeinsamen Wirkungsbereichen angeben:

  - *nur ein* $\mathcal{P}$ — $\mathcal{P}$ wirkt global: $\mathcal{P} \rightarrow \mathcal{WB}_{glob}$,
  - *mehrere* $\mathcal{P}_i$
    - *vollständig gemeinsamer* $\mathcal{WB}$: $\forall\ \mathcal{P}_i \rightarrow \mathcal{WB}_{glob}$,
    - *partiell gemeinsamer* $\mathcal{WB}$: $\forall\ \mathcal{P}_i \rightarrow \mathcal{WB}_{gem},\ \forall\ \mathcal{P}_i \rightarrow \mathcal{WB}_i$, $\mathcal{WB}_{glob} = \mathcal{WB}_{gem} \cup \{ \bigcup_{\forall i} \mathcal{WB}_i \}$,
    - *vollständig getrennte* $\mathcal{WB}_i$: $\forall\ \mathcal{P}_i \rightarrow \mathcal{WB}_i,\ \mathcal{WB}_i \cap \mathcal{WB}_j = \oslash$, $\mathcal{WB}_{glob} = \oslash$.

  Wirkungsbereiche von Prozessen sind dabei nicht notwendigerweise an Schutzvorkehrungen gebunden. Wirkungsbereiche können auch durch "korrekte" Steuerung der Prozesse, d.h. durch "sichere" Programme und den Einbau von Prüfcode eingegrenzt werden.

- ***Schutzeigenschaften:***

  Schutzverletzungen sind nicht gewollte Zugriffsoperationen auf Elemente eines Systems. Schutzverletzungen können folglich nur von Prozessen ausgehen. Das Merkmal Schutz gibt an, welche Vorkehrungen in einem System vorhanden sind, um Schutzverletzungen zu verhindern oder zu erkennen und entsprechend darauf zu reagieren. Man kann ein allgemeines Schema für die Behandlung von Schutzverletzungen angeben:

- *Auslesen von Daten, Informationen*		→ *Verhindern,*
- *Manipulieren von Daten, Informationen*	×	→ *Erkennen und Reagieren,*
		→ *keine Maßnahmen.*

  Schutzvorkehrungen können bei Prozessen (Aktivitäten, Steuerungen) oder bei den zugegriffenen Elementen angesiedelt sein. Bei Prozessen kann geprüft und gesichert werden, daß nur "legale" Anweisungen die Prozesse steuern und auf diese Weise Schutzverletzungen ausgeschlossen werden (vgl. *SPIN* oder *Bridge* in Abschnitt 2.7). Es kann auch Prüfcode in die Programme eingefügt werden. Eine andere Möglichkeit ist, Schutz an den zugegriffenen Elementen festzumachen (z.B. Speicher über getrennte Adreßräume oder *page protection*, *ACL's* in Dateisystemen u.a.). Für Schutzmerkmale kann man zusammenfassen:

  - *Prozesse* → *Steuerung:* Sichern der Korrektheit und Legalität der Programme,
    → *Aktivität:* Privileg des Aktivitätsträgers (z.B. *CPU-Mode*),
    → *Sicherung getrennter Wirkungsbereiche:* Adreßräume, Maschinengrenzen, Identifikations- und Zugriffsdomänen,
  - *Elemente* → *Zugriffskontrolle:* Rechte, Zugriffslisten, notwendige Privilegstufen, Authentifizierung, Autorisierung, Berechtigungsnachweise,
    → *Erkennen illegaler Zugriffe:* Registrierung, Beobachtung,
    → *Kodierung der Information:* Formate, Protokolle, Verschlüsselung.

  Die heute typische Methode für Speicherschutz ist die Herstellung getrennter Adreßräume für Prozesse. Es können jedoch auch andere Lösungen in Verbindung mit dem leichtgewichtigeren *Single Address Space* Ausführungsmodell angewandt werden. Das ist vor allem für die sichere dynamische Erweiterbarkeit von Kernen ein aktueller Forschungsschwerpunkt.
  Auch für das Erkennen von Manipulationen gibt es zahlreiche Methoden (Prüfsummen, Werten von *Hash*-Funktionen u.a.). Eine einfache Art des Schutzes vor dem Auslesen von Informationen ist die Verschlüsselung von gespeicherten oder übertragenen Daten.

  Die Kapselung von Verarbeitungsprozessen in Instanzen ist dabei nicht mit Schutz gleichzusetzen, sondern Kapselung wurde an der Beschränkung der $\mathcal{WB}_i$ für innere Prozesse i über inneren Daten erklärt. *Kapselung ist ein Merkmal der generalisierten Architektur, Schutzeigenschaften sind dagegen ein Merkmal der Spezialisierung eines Ausführungsmodells.*

  Die Einschränkung des Wirkungsbereichs kann bei korrekten Programmen von Prozessen auch ohne Schutzvorkehrung eingehalten werden. Wenn Schutz aber als notwendig erachtet und der Aufwand dafür akzeptiert wird, können Maßnahmen in einem System dafür vorgesehen werden. Wirken sie generell, sind auch sie Bestandteil des Ausführungsmodells.

4.3.4 Zusammenfassung zu Spezialisierung

Die Idee der generalisierten Architektur bestand darin, allgemeine und übertragbare Architekturmerkmale für ein breites Spektrum von Zielbereichen zusammenzufassen. Durch Spezialisierung erfolgt dann die notwendige Anpassung an die Maßgaben eines konkreten Zielsystems.

Spezialisierung bezieht sich dabei auf die Festlegung von ***Ausprägungsvarianten*** der Architektur und die Zuordnung von ***Ausführungseigenschaften***. Bei Ausprägungsvarianten ging es um die Auswahl der strukturellen Architekturmerkmale, die für ein Zielsystem nützlich und angemessen sind. Die Ausführungseigenschaften werden in der Infrastruktur durch *Dienste* und durch das *Ausführungsmodell* hergestellt.

- **Ausprägungsvarianten der Architektur**

Die Angemessenheit und Zweckmäßigkeit bestimmt, welche Ausprägungen Anwendung finden.

- pro System – $\mathcal{IVS}$:	Wieviele Schichten gibt es ?
- pro Schicht – $\mathcal{L}_i$:	Gibt es ausschließlich Basiselemente $\mathcal{BE}$? Gibt es auch oder ausschließlich Instanzen $\mathcal{I}_j$ und Instanzbereiche $\mathcal{IB}$?
- pro Instanz – $\mathcal{I}_j$:	Gibt es im Inneren ausschließlich Basiselemente ? Gibt es auch innere Instanzen (→ Aggregationsrekursion) ?

In einem Gesamtsystem gibt es drei Arten von Steuerungsebenen, wobei für jede Steuerungsebene eine Ausprägung durch Einschränkung der Merkmalsmengen angegeben werden kann (→ *$\mathcal{K}$-Mengen:* $\{ 0, 1, n \}$ – Elemente für ein Merkmal). Es gelten außerdem Plausibilitätsregeln. Leere Mengen (→ $\mathcal{K}$-Menge = $\{ 0 \}$) waren auch erforderlich, um die erzeugenden Rekursionen der Architekturdefinition abzubrechen (→ *Aggregations-* und *Infrastrukturrekursion*). In den Abbildung 4.7 – 4.9 wurden Beispiele für Ausprägungsvarianten angegeben.

- **Ausführungseigenschaften – Dienste der Infrastruktur**

Infrastruktur wirkt auch als Verarbeitungsinstanz in einem System, so daß die Merkmale von Instanzen auf Infrastruktur übertragen werden können. Es gelten somit die Aussagen zu Diensten, Operationen, Protokollen, zum Identifikationsschema, zu Schnittstellen und Definitionen aus Abschnitt 4.2.2 auch für Infrastrukturen. Ein wichtiger Punkt war, ob Infrastruktur als globale Instanz in Erscheinung tritt oder ob auch innere Elemente nach außen identifizierbar sind.

- **Ausführungseigenschaften – Ausführungsmodell**

Wesentlich schwieriger läßt sich der zweite Aspekt des Ausführungsmodells erfassen. Die Ausführungseigenschaften wurden auf Basiselemente ($\mathcal{Z}$, $\mathcal{P}$, $\mathcal{P}_{rog}$) und *Instanzen* $\mathcal{I}$ bezogen:

- *Installation*	–	Typ, Repräsentation, Elementquantifizierung ($n = 1$, $n \geq 1$),
- *Lebensdauer*	–	permanent, temporär, persistent,
- *Örtliche Bindung*	–	fest oder migrierbar – lokal, verteilt oder repliziert,
- *Identifikation*	–	Typ, Element, Verarbeitung.

Die Art der Verarbeitung wurde durch die Wirkung von Prozessen in einer Steuerungsebene bestimmt. Es wurden vier Bereiche für diese Facette des Ausführungsmodells angegeben:

- *Prozeßmodell*	–	1 $\mathcal{P}$, n $\mathcal{P}$*:* konkurrierend, kooperierend, Scheduling-Strategie,
- *Wirkungsbereich* $\mathcal{WB}$	–	separat, partiell oder global gemeinsam, ($\approx$ *Adreßraum*),
- *Ablaufbedingungen*	–	explizit (→ *Interaktion*), implizit (→ *Prozeßmodell*),
- *Schutzeigenschaften*	–	bei Prozessen ($\mathcal{P}_{rog}$, $\mathcal{WB}$, Privileg), bei Elementen.

Damit sind die wesentlichen Aspekte von Ausführungsmodellen noch einmal zusammengefaßt. Eine vollständige Charakterisierung war dabei nicht beabsichtigt und ist auch nicht möglich.

4.4 Abbildung von Objekten in Instanzen

Die Unterstützung objektorientierter Anwendungen und einer objektorientierten Programmiermethodik ist heute eine wichtige Forderung an Betriebssysteme. In Kapitel 2 wurde aber der signifikante Unterschied zwischen "Denk-", Programmier- und Ablaufstruktur herausgearbeitet. Objektstrukturen widerspiegeln sich in vielen populären Sprachen oder Systemen nicht oder auf gänzlich andere Weise im ablaufenden System. Zur Aufhebung dieses "zweiten" Strukturbruchs[13] sollen kurz mögliche Abbildungen von Objekten in Ablaufstrukturen dargestellt werden. Für *CHEOPS* wird diese Problematik in einer weiteren Dissertation für *C++* umfassend behandelt. Es geht dort vor allem auch um die Repräsentation der Klassenzugehörigkeit von Objekten und von Vererbungsrelationen im ablaufenden System über sogenannte *Klassenobjekte* [Schu96].
Man findet bereits Systeme, bei denen eine explizite Abbildung von Objekten in Ausführungseinheiten des ablaufenden Systems vorgenommen wird und diese beiden "Welten" auch explizit unterschieden werden. In der Definition von *CORBA* [OMG91] sind vier Abbildungsvarianten vorgesehen. Anpassungen erfolgen durch den *Basic Object Adapter (BOA)*. Auch für *Sun's* Kern *Spring* und die zugehörige Programmiermethodik werden Objekte und Objektimplementierungen ausdrücklich unterschieden [SunSoft94, Schö96]. Objekte werden für die Implementierung mit den Attributen *serverbasiert* (vgl. Fall (a) in Abbildung 4.10) oder *serverlos* (Fall (b)) versehen. Systeme dieser Art sind von vorn herein als verteilte Systeme konzipiert, deren grundlegende Ausführungseinheiten Prozesse (→ *Instanzen*) sind. Die Verbindung zwischen Objektorientierung in der Welt von Analyse, Entwurf und Programmierung zur Welt ablaufender Systeme erscheint über die Abbildungsvarianten in Bild 4.10 sinnvoll möglich zu sein. Übertragen in die Terminologie hier bedeutet das: ***Instanzen sind Ausführungseinheiten für Objekte.***
Man kann es umgekehrt betrachten, daß Programme und Daten für Instanzen durch Objekte beschrieben und implementiert werden. Bei einer 1:1–Abbildung kann man eine Instanz auch als aktives Objekt auffassen. Es handelt sich hier aber nur um eine mögliche Abbildungsvariante.

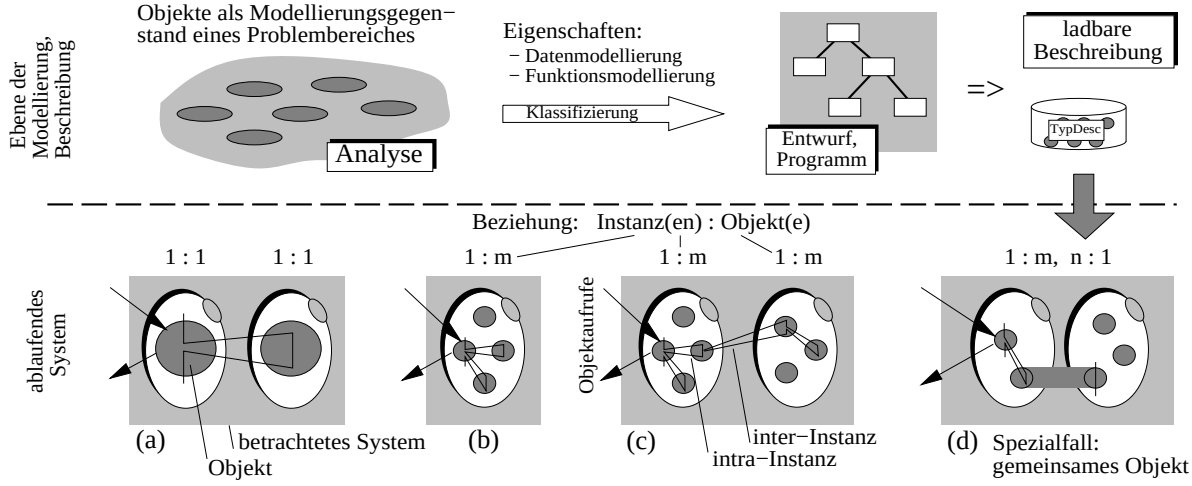


Abb.4.10 Abbildungsvarianten für Objekte in Instanzen

Über die Phasen der Analyse, des Entwurfs und der Programmierung finden manuelle oder maschinelle Transformationen von Beschreibungen statt, bis eine Form vorliegt, die zur Herstellung eines Abbilds in einer Maschine zur Steuerung von Verarbeitungsprozessen interpretiert werden kann. Man kann Relationen zwischen Objekten und Instanzen angeben. In Fall (a) besteht ein System aus n Instanzen mit je einem Objekt, in (b) gibt es nur eine Instanz mit m-Objekten (→ *C++*), bei (c) gibt es n-Instanzen mit je m-Objekten und (d) ist ein Spezialfall mit einem gemeinsamen Objekt in mehreren Instanzen (gemeinsame *Z:* → *Interaktionsobjekt*).

[13] Der "erste" Strukturbruch bestand zwischen Anwendungs- und Kern-Bereich (vgl. Abschnitt 2.5).

4.5 Zusammenfassung und Fazit

> " Leider werden heute noch zu oft Systeme konzipiert, worin der Programmierung und Integration größere Aufmerksamkeit beigemessen wird als einer übergeordneten Architektur." [Koc96], S. 34.

Eine Architektur für ein technisches System beschreibt ein übergreifendes Konzept, welches die Grundlage für die Gliederung des Systems in seine Elemente und ihre Beziehungen bildet. Die Architektur leitet sich aus Zielen und Anforderungen ab, welche an das System für seine spätere Benutzung gestellt werden. Die Funktion steht somit für technische Systeme im Vordergrund. Eine Architektur sollte einfach und übersichtlich sein und nur den Rahmen für das System vorgeben. Das ist eine wichtige Eigenschaft für die Skalierbarkeit, Anpaßbarkeit und Erweiterbarkeit. Eine Architektur sollte auch das Gesamtsystem und nicht nur spezielle Teilbereiche erfassen.

Der durch die Architektur gegebene Gesamtzusammenhang begründet eine Vielzahl von Entwurfsentscheidungen, welche Elemente in welchen Ausführungen für welche Funktionen in einem System enthalten sind. Er ist für das Verständnis des Systems wesentlich. Anwendungsanpaßbare Betriebssysteme erfordern daher unabdingbar eine Architektur für das Gesamtsystem, wenn das gesamte System potentiell Gegenstand für Anpassungen sein soll.

Der Bezugsbereich für die hier vorgeschlagene Architektur ist die Realität eines ablaufenden informationsverarbeitenden Systems. Sie orientiert sich an einem Modell dieser Realität, welches in Kapitel 3 abgeleitet und begründet wurde und die Grundlage dieser Architektur bildet.

Eingangs wurde auf die breiter werdende Diversifizierung in Betriebssystemen hingewiesen. Daraus leitet sich auch das Ziel ab, trotz zunehmender Verschiedenartigkeit von Zielsystemen, übergreifende Strukturmuster möglichst oft, aber für den jeweiligen Zweck in angemessener Form, anzuwenden. Eine gute Systemstruktur ist durch Einfachheit, Zweckmäßigkeit, Erweiterbarkeit und Skalierbarkeit für verschiedene Zielbereiche gekennzeichnet. Innere Teilkomponenten sollten klar abgegrenzt sein und zwischen ihnen plausible Beziehungen bestehen. Diese Faktoren bestimmen die Ziele einer Architektur für anwendungsanpaßbare (Betriebs-) Systeme.

Die Architektur widerspiegelt sich auch in Dokumenten, Darstellungen und Beschreibungen, welche für den Entwurf und die Herstellung eines Systems und seiner Elemente notwendig sind. An dieser Stelle trifft die Architektur des (ablaufenden) Zielsystems mit Modellen und Architekturen der Softwaretechnologie zusammen. Beide Betrachtungswelten sind heute nicht deckungsgleich und überlagern sich folglich. Es ist dabei wesentlich, daß die Architektur des ablaufenden Zielsystems die dominierende Rolle spielt und nicht die Umkehrung gilt, daß die Softwarestruktur oder auch die Software-Architektur die realen Abläufe und Zusammenhänge im eigentlichen Zielsystem verdeckt. Diese Problematik wurde in Kapitel 2 ausführlich dargestellt.

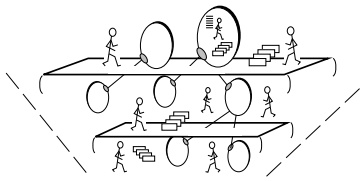
Für das *CHEOPS*-Projekt werden beide Betrachtungswelten klar unterschieden und voneinander getrennt. In dieser Dissertation steht die Architektur des ablaufenden Zielsystems im Mittelpunkt. Diese explizite Differenzierung ist ein wesentlicher Unterschied zu heute oft softwaretechnologisch geprägten Ansätzen in Betriebssystemen, bei denen die tatsächlichen Ablaufstrukturen in den Hintergrund treten oder explizit verborgen werden sollen.
In Abschnitt 4.4 wurde ein Weg gezeigt, eine Brücke ($\rightarrow$ *Integration*) zwischen der Welt objektorientierter Software und der Realität ablaufender Systeme zu schlagen.

Die Architektur umfaßt aber nicht nur strukturelle Aspekte der Systemgliederung. Mit jeder Architektur ist auch eine Methodik oder Vorgehensweise für den Entwurf und die Herstellung von Systemen verbunden. Das kommt hier für die Architektur selbst durch die anfangs gestellten Fragen zum Ausdruck: Was ist für die Architektur universell? Was ist speziell? Was ist spezialisierbar? Aus der Beantwortung dieser Fragen resultiert die hier vorgenommene Zweiteilung in

eine generalisierte Architektur, die in hohem Maße universelle Züge trägt, und deren Merkmale für jeweils konkrete Zielsysteme zu spezialisieren sind, um sie für diese passend festzulegen.

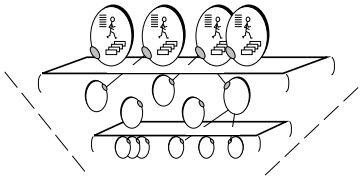
Für die ***generalisierte Architektur*** wurden fünf Elemente eingeführt.

◦ *Basiselemente*	$\mathcal{BE}$	=	$\{ \mathcal{Z},\ \mathcal{P},\ \mathcal{P}_{rog} \}$
◦ *Instanzen*	$\mathcal{I}$	=	$\{ \mathcal{BE} \} \cup \{ \mathcal{I}' \}$
◦ *Instanzbereiche*	$\mathcal{IB}$	=	$\{ \mathcal{I} \} \cup \{ \mathcal{IB}' \}$
◦ *Systemschicht*	$\mathcal{L}$	=	$\{ \mathcal{BE},\ \mathcal{I},\ \mathcal{IB} \}$
◦ *Infrastrukturrekursion*	$\mathcal{L}_i$	=	$\mathcal{I}nfra\ (\ \mathcal{L}'_{i+1}\ )$



Basiselemente schaffen die Grundlage für informationsverarbeitende Systeme. Die auch in anderen Bereichen anzutreffende Gliederung komplexer Systeme in autonome Teilsysteme wird in dieser Architektur über die Zusammenfassung von Basiselementen zu Instanzen modelliert. Instanzen sind abgegrenzte, autonom arbeitende Teilsysteme, die selbständig Dienste erbringen.

Die Architekturdefinition erlaubt mit zwei Rekursionen die Erzeugung höherer Strukturelemente: ***Aggregationsrekursion*** *und* ***Infrastrukturrekursion***. Die Rekursion von Infrastrukturen dient zur Gliederung in aufeinander aufsetzende Schichten, welche infrastrukturelle Aufgaben für jeweils übergeordnete Schichten ausführen. Eine Softwareschicht oder Infrastruktur kann als Ganzes wiederum als abgeschlossene Instanz angesehen werden, mit der Besonderheit, daß nicht nur Infrastrukturdienste ausgeführt werden, sondern daß auch der übergeordnete Bereich erst durch das Wirken dieser Infrastruktur-Instanz hergestellt wird.



Eine ***"reine Instanzstruktur"*** sollte als bevorzugte Form angesehen werden, da hier die sonst uneingeschränkten Beziehungen zwischen Basiselementen einer Softwareschicht über Instanzen klar geordnet werden können und auch die Beziehungen zur Instanz "Infrastruktur" gleichartig behandelt werden können. Eine solche Ausprägung der Architektur entspricht genau dem ***Client-Server-Modell***, welches seit langem für Anwendungsbereiche bekannt und akzeptiert ist. Dieses Modell eignet sich auch für die inneren Schichten des Betriebssystems, so daß damit die eingangs gewünschte homogene Grundstruktur über alle Schichten des Systems durch eine reine Instanzmodellierung hergestellt werden kann. Voraussetzung dafür ist ein hohes Maß an Skalierbarkeit des Grundmodells für Instanzen, von der Behandlung von Unterbrechungen bis hin zu verteilten Systemen. Diese Skalierbarkeit wird hier durch Abstraktion und Trennung konkreter Ausführungseigenschaften für unterschiedliche Arten von Instanzen vom allgemeinen Grundmuster für Instanzen erreicht.

Die ***Spezialisierung*** der Architektur erfolgt in zwei Richtungen. Bei ***Ausprägungsvarianten*** wird festgelegt, welche Merkmale es für ein Zielsystem geben soll, d.h. welche Schichten es gibt und welche Elemente in den Schichten existieren.
Die oben beschriebene reine Instanzstruktur ist dabei eine spezielle Ausprägungsvariante. Die zweite Dimension der Spezialisierung legt die ***Ausführungseigenschaften*** der Elemente in einer Schicht fest: *Dienste der Infrastruktur* und *das Ausführungsmodell* (die Art von Elementen und die Wirkung von Prozessen).

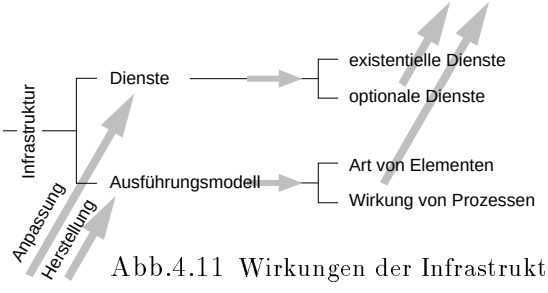
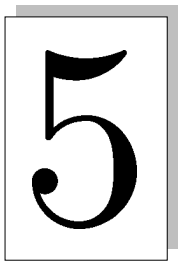


Abb.4.11 Wirkungen der Infrastruktur

Die vorgeschlagene Architektur ist einfach, da nur wenige Grundelemente vorkommen und über wenige Regeln kombiniert werden. Sie ist zweckmäßig, weil die Grundstruktur von Ausprägungsvarianten und Ausführungseigenschaften getrennt und über Spezialisierung für individuelle Zwecke anpaßbar ist. Die Architektur ist durch die zwei Rekursionen auch theoretisch unbegrenzt erweiterbar. ***Instanzen*** sollten als dominierendes Merkmal favorisiert werden.

Im anschließenden Kapitel wird diese Architektur für den *CHEOPS*-Kern angewandt.



Kapitel 5

Der *CHEOPS*-Kern

Die Implementierung des *CHEOPS*-Kerns zeigt die praktische Seite des *CHEOPS*-Projekts. Mit dem *CHEOPS*-Kern werden zwei grundsätzliche Ziele verfolgt:

- die Untersuchung und Bewertung der Implementierbarkeit eines Betriebssystemkerns anhand der vorgeschlagenen Architektur und
- die Schaffung der experimentellen Basis für weitere Untersuchungen zu dynamisch rekonfigurierbaren und adaptiven (Betriebs-) Systemen im Rahmen des *CHEOPS*-Projekts.

5.1 Ziele und Anforderungen an den *CHEOPS*-Kern

Aus beiden Zielstellungen folgt eine Spezifik, die den *CHEOPS*-Kern von anderen Entwicklungen unterscheidet. Es geht *nicht* um die Herstellung einer weiteren, für bestimmte Einsatzbereiche effizienteren oder mit besonderen Eigenschaften ausgestatteten Implementierung eines Vorbildbetriebssystems, wie es für viele μ-Kerne (z.B. *Mach* $\rightarrow$ *Unix*) zutrifft. Der Schwerpunkt liegt bei der Untersuchung und Bewertung eines Systems, welches anhand der vorgeschlagenen Architektur entworfen und implementiert wird, um damit Rückschlüsse auf die Eignung der Architektur zu ziehen. Es geht vorrangig um die Untersuchung der Folgen neuartiger Implementierungsprinzipien, die sich aus der Architektur für den Kern ergeben, beispielsweise der Behandlung von Interrupts durch eine spezielle Art äußerst leichtgewichtiger Instanzen. In Abschnitt 5.3 wird dieser Aspekt im Detail vorgestellt. Auch für die anderen Forschungsziele stehen Untersuchungen zur Implementierbarkeit der im *CHEOPS*-Projekt entwickelten Konzepte und Ideen zu dynamisch anpaßbaren und adaptiven Systemen im Vordergrund. In diesem Sinne ist der *CHEOPS*-Kern als offene Experimentalplattform für die Untersuchungsziele des *CHEOPS*-Projekts konzipiert.

Aus dem Ziel einer konsequenten Anwendung der Architektur für alle Systemschichten folgte auch, daß der *CHEOPS*-Kern als *stand-alone* System auf der Hardware aufgesetzt werden sollte. Auch hierin unterscheidet sich der *CHEOPS*-Kern von anderen Arbeiten, die als Aufsätze auf Wirtsbetriebssysteme implementiert sind. In Kapitel 2 wurden eine Reihe von Beispielen dafür genannt, die man vor allem im Bereich verteilter Systeme antrifft.

Diese strategischen Entscheidungen haben weitreichende Konsequenzen. Es folgt vor allem daraus, daß für den *CHEOPS*-Kern beim Stand dieser Dissertation keine Kompatibilität zu existierenden Betriebssystemen angestrebt ist. Für die Zukunft soll das jedoch nicht grundsätzlich ausgeschlossen sein. Der Vorteil dieses Vorgehens ist, daß man sich auf die primären Untersuchungsgegenstände konzentrieren und die vorhandenen Ressourcen dafür einsetzen kann. Diese Entscheidung war auch noch aus einem anderen Grunde notwendig, um nämlich tatsächlich neuartige Konzepte umsetzen zu können, ohne die Hypothek vorgegebener Funktionen und daraus

folgender Implementierungsprinzipien seit langem bekannter Vorbildsysteme tragen zu müssen. Eine alternative Implementierung müßte auch sofort den Effizienzvergleich zu seinem Vorbildsystem antreten. Die Historie der μ-Kerne zeigt, daß dieser Umstand in der Vergangenheit entweder gegen neue Entwicklungen entschieden wurde oder daß derartig gravierende Kompromisse einzugehen waren, daß letztendlich das angestrebte Ziel in Frage stand. Als Beispiel könnten die "unechten" monolithischen μ-Kerne gelten ($\rightarrow$ *Mach2.x*). Am Ende des Kapitels 2 wurde die Situation im Bereich der μ-Kerne ausführlich diskutiert.

Es hat sich trotz vielfältiger Bemühungen seitens der Forschung herausgestellt, daß die traditionellen Betriebssysteme in der traditionellen Art auf heutigen Maschinen in gewisser Weise optimal implementiert sind. Betriebssystemhersteller unternahmen auch große Anstrengungen dafür. Bezüglich Effizienz gibt keine "bessere Implementierung" von *Unix* als die eines monolithischen, ggf. multi-threaded Kerns. Anders läßt sich die fehlende Akzeptanz neuer Betriebssystemprinzipien in ihrer tatsächlichen Anwendung nicht erklären.
Für die Forschung sollte daher nicht primär die Frage im Vordergrund stehen, bewährte Systeme neu zu (re-) implementieren, sondern neue Systeme mit neuartigen Eigenschaften nach neuen Methoden herzustellen und deren Eigenschaften zu untersuchen. Für die Akzeptanz in der Praxis ist es dann in einem zweiten Schritt aber wichtig, die Vorteile klar herauszustellen, und es sind Wege für eine Migration existierender Software auf diese neuen Systeme aufzuzeigen.
Die Erfahrung aus den an *Mach* orientierten μ-Kernen ist ein Beispiel eines (vorerst) gescheiterten Versuchs, Ablaufkompatibilität ($\rightarrow$ zu *Unix*) und eine neue Systemarchitektur ($\rightarrow$ μ-Kern mit *tatsächlich* aus dem Kern ausgelagerten Betriebssystemfunktionen) zu vereinen und obendrein noch im Wettbewerb der Effizienz zu bestehen. Diese Erkenntnis begründet auch den gegenwärtigen Trend, der in Abschnitt 1.2 als neuer Aufbruch in der Forschung zu Betriebssystemen bezeichnet wurde. In Abschnitt 2.7 wurden fünf aktuelle Entwicklungen vorgestellt.

Aus dieser Argumentation leitet sich auch das Vorgehen für das *CHEOPS*-Projekt ab, Kompatibilität zu bestehenden Systemen vorerst nicht herzustellen, sondern die hier vorgeschlagenen und untersuchten Konzepte in den Vordergrund zu stellen. Der Preis dieser Entscheidung ist natürlich, daß auf absehbare Zeit keine bestehende Anwendungssoftware ablauffähig sein wird. Das hat auch zur Folge, daß für den *CHEOPS*-Kern eine eigene einfache Arbeits- und Ablaufumgebung herzustellen war (*stand-alone shell – sash* – vgl. Abschnitt 5.5.1). Es ist auch vorerst ausgeschlossen, daß der *CHEOPS*-Kern selbst als Entwicklungsumgebung fungiert. Alle Entwicklungsarbeiten finden auf anderen Systemen als *Cross-Entwicklungen* statt.

Zur Einordnung der Kerns in das Klassifikationsschema aus Abschnitt 2.5 orientiert sich der *CHEOPS*-Kern an der Kategorie der *In-Kernel Server*, da diese Ausprägung als aussichtsreichste Variante einer Kern-Architektur anzusehen ist. Sie erlaubt es wohl am besten, Flexibilität und innere Struktur mit Effizienz zu verbinden. Es gibt keine vordefinierte Funktionsschnittstelle für den *CHEOPS*-Kern, da seine Herstellung nicht die Umsetzung einer bestimmten Funktionalität für Anwendungen zum Ziel hat, sondern im Kern, je nach Untersuchungsziel, neue Funktionen bzw. andere Mechanismen eingebaut und untersucht werden sollen. Es gibt somit keinen vordefinierten Kern, der *mit festen Funktionen* ausgestattet ist und *auf dessen Basis* die Experimente für das *CHEOPS*-Projekt durchgeführt werden, sondern der Kern ist vor allem *selbst Gegenstand für Experimente*. Dieser Kern muß daher in besonderem Maße flexibel und für die Zwecke der Untersuchungen anpaßbar sein. Er ist in diesem Sinne eine ideale Anwendung für eine anwendungsanpaßbare Systemarchitektur.

Der *CHEOPS*-Kern wird – ausgehend von der Hardware – in Schichten (vgl. Abschnitt 5.2.1) aufgebaut. Natürlich gibt es auch für den *CHEOPS*-Kern invariante Grundfunktionen, welche eine Schicht als Infrastruktur für die übergeordneten Schichten erbringt (vgl. Abschnitte 5.2.2 und 5.2.3). Diese Grundfunktionen resultieren aus der Arbeitsweise der verwendeten Zielmaschinen und den daraus folgenden Ablaufformen. Es gibt daher notwendigerweise ein Teilsystem zur

Behandlung von Interrupts, und es muß Komponenten geben, die parallele Prozesse herstellen und die Ressourcen der Hardware verwalten und für übergeordnete Schichten transformieren. Es wird aber ausdrücklich angestrebt, die dafür notwendigen Strategien und auch die erforderlichen Mechanismen in allen Schichten des Systems in hohem Maße offen, flexibel und austauschbar zu gestalten. Selbst die Anzahl und Art der Schichten ist nicht zwingend vorgegeben. Es wird auch nicht statisch festgelegt, welche Arten von Prozessen es in welchen Schichten gibt, nach welchen Strategien die Verwaltung und Zuteilung von Ressourcen erfolgt, welche Mechanismen dafür in der Infrastruktur vorgesehen sind oder welche Interaktionsprinzipien zur Anwendung kommen. Es hängt von den jeweiligen Untersuchungszielen im Rahmen des *CHEOPS*-Projekts ab, welche Funktionen in welcher Schicht benötigt werden und dann dafür herzustellen sind.
Dieses hohe Maß an Offenheit unterscheidet den Ansatz des *CHEOPS*-Kerns gravierend von anderen Systementwürfen und Implementierungen. Anwendungsanpaßbarkeit des *CHEOPS*-Kerns bedeutet hier, je nach Zielstellung einer Untersuchung, neue Komponenten in das System einzufügen bzw. bestehende zu entfernen, zu ergänzen oder zu verändern. Dies kann im Sinne der Herstellung der Software für spezielle Kern-Varianten, aber auch während des Ablaufs des System geschehen, wenn es um Untersuchungen zu dynamischer Anpaßbarkeit und adaptivem Verhalten geht, wie es als ein Schwerpunkt für das *CHEOPS*-Projekt formuliert ist.

Um aber vorerst eine Ausgangsbasis zu schaffen, wird in Abschnitt 5.2 für den *CHEOPS*-Kern eine Architektur abgeleitet, die eine praktische Umsetzung erlaubt. Als ein Aspekt wird in Abschnitt 5.3 eine alternative Art der Behandlung von Interrupts untersucht und bewertet.

Für die Implementierung wurden für den *CHEOPS*-Kern zwei Maschinentypen gewählt:

- *CADMUS-9700* Workstation (2 x m68020-CPU, asymmetrisch),
- *PC* (*Intel 486*, protected mode).

Die Arbeiten wurden vom Autor initial auf der zuerst genannten Maschine ausgeführt, da Vorarbeiten eines vorausgegangenen Projekts zur Portierung des Betriebssystems *BirliX* auf diese Maschine existierten [Grau91b, Grau92b, Grau92c]. Es war naheliegend, diese Ergebnisse in Form eines ablauffähigen stand-alone Kerns weiterzuverwenden. Aufgrund der überalterten Hardware der *CADMUS*-Workstation wurde 1995 entschieden, den initialen Kern auf die PC-Plattform zu übertragen und alle Folgeentwicklungen auf dieser Basis fortzusetzen [VW96].

Zusammenfassend lassen sich folgende Punkte für die Zielstellung des *CHEOPS*-Kerns angeben.

- **Anwendbarkeit der Architektur** – Es soll untersucht werden, inwieweit sich die hier vorgeschlagene Architektur zur Schaffung eines Betriebssystems eignet.
- **Basis für experimentelle Untersuchungen** – Das *CHEOPS*-Projekt ist weiter gefaßt als die Untersuchung eines Architekturvorschlags, der in dieser Dissertation im Mittelpunkt steht. Der *CHEOPS*-Kern ist die experimentelle Basis für diese weiterführenden Arbeiten.
- **vorerst keine Kompatibilität zu Vorbildsystemen** – Diese Eigenschaft begründet sich aus Erfahrungen anderer Entwicklungen und der Konzentration der vorhandenen Ressourcen auf die eigentlichen Untersuchungsziele des *CHEOPS*-Projekts.
- **hohes Maß an Flexibilität** – Aufgrund der verschiedenartigen Untersuchungsziele muß der *CHEOPS*-Kern in hohem Maße flexibel sein, wobei sich diese Flexibilität auch auf innerste Kern-Mechanismen und verschiedenartige Ausführungsmodelle bezieht.
- **stand-alone Implementierung** – Da innerste Kern-Strukturen und -funktionen betrachtet werden, ist eine Implementierung auf einem Wirtsbetriebssystem ausgeschlossen.

- **dynamische Anpaßbarkeit** – Ein wichtiges Untersuchungsziel ist für das *CHEOPS*-Projekt die dynamische Anpaßbarkeit des Kerns bzw. von Systemkomponenten, so daß diese Eigenschaft für den *CHEOPS*-Kern mit vorzusehen ist.

  Dynamische Anpaßbarkeit soll in drei Stufen verwirklicht werden:
  - 1. Stufe – Erweiterung des Kerns durch Laden und Starten neuer Instanzen im Kern,
  - 2. Stufe – Kern-Rekonfigurierung durch Austausch von Kern-Instanzen bzw. Teilen davon,
  - 3. Stufe – dynamisch adaptives Verhalten des Kerns und von Systemkomponenten.

Der *CHEOPS*-Kern fungiert somit in zweifacher Hinsicht als Untersuchungsgegenstand für ein anwendungsanpaßbares System. Zum einen ist er ein Beispiel für eine Anwendung oder Anpassung der generalisierten Architektur für die experimentellen Zwecke des *CHEOPS*-Projekts. Zum anderen soll der *CHEOPS*-Kern selbst für die Durchführung verschiedener Experimente in hohem Maße anpaßbar sein. Das Anwendungsgebiet dieses Kerns liegt somit bei experimentellen Untersuchungen, welche im *CHEOPS*-Projekt durchgeführt werden [Kal96a]:

- Abbildung objektorientierter Sprachelemente von *C++* in das ablaufende System und Objekt-Rekonfigurierung [Schu96],
- dynamische Anpassung von Kernen in den oben genannten drei Abstufungen,
- dynamisches Rekonfigurierungsmanagement für adaptives Verhalten [Woh96].

Der *CHEOPS*-Kern ist an der Klasse der μ-Kerne mit *In-Kernel Servern* orientiert. Diese Kerne sind dadurch gekennzeichnet, daß sie nicht nur oberhalb des Kerns eine Instanzstruktur aufweisen, sondern auch innerhalb des Kerns, aber in einer *"leichteren" Ausführungsform*. Auf diese Weise lassen sich die ursprünglichen Ziele der μ-Kerne mit einer effizienten Ablaufform am besten verbinden. Für anwendungsanpaßbare Kerne ist dabei wesentlich, daß mit *In-Kernel Servern* das gleiche Strukturprinzip (→ *Server*, *Instanzen*) des Anwendungsbereichs für einen Teil des Kern-Bereichs übernommen wird. Damit kommen Kerne mit *In-Kernel Servern* der eingangs in dieser Arbeit genannten Zielstellung nahe, innerhalb des Kerns dieselbe Struktur wie im Anwendungsbereich aufzuweisen, um eine homogene Grundstruktur in möglichst allen Schichten des Systems zu schaffen und damit die Herstellung oder Anpassung von Kern-Komponenten auch für Dritte auf praktikablem Wege zu ermöglichen.

Das *Client-Server-Modell* hat sich als Ablaufform und auch als Programmierparadigma im Anwendungsbereich seit vielen Jahren bewährt. Es ist daher nur folgerichtig, dieses Modell auch innerhalb des Kerns anzuwenden. Kerne mit *In-Kernel Servern* setzen aber das Prinzip einer homogenen, schichtübergreifenden Grundstruktur im Kern- und Anwendungsbereich nicht mit Konsequenz um:

- Sie sind noch stark an den klassischen Eigenschaften der μ-Kerne orientiert, d.h., es gibt fixierte Dienste und ein festes Ausführungsmodell der Kern-Infrastruktur für den Anwendungsbereich (→ separate Adreßräume, preemptives "faires" Prozessorscheduling, virtueller Speicher u.a.). Andere Prinzipien sind nicht vorgesehen.
- Die Infrastruktur für die *In-Kernel Server* innerhalb des Kerns ist selbst monolithisch ausgeführt. Dieser verbleibende Kern-Bereich ist zwar wesentlich kleiner als in einem monolithischen Gesamtkern, es stellt sich aber die Frage, ob nicht das Grundmuster der Server (→ *Instanzen*) auch in diesem Teil des Kerns anwendbar ist. Dessen primäre Aufgabe besteht darin, die Interruptverarbeitung auszuführen, die übergeordneten Kern-Prozesse zu erzeugen, ihren Ablauf zu steuern und den Speicher des Kern-Adreßraumes zu verwalten.
- *In-Kernel Server* sind in der Regel nicht in der Weise flexibel, daß Server-Prozesse im Kern tatsächlich austauschbar sind (vgl. *Mach-INKS* [Lepr93]).

- Das Ausführungsmodell und die Dienste des Kerns werden stets für alle Prozesse des Anwendungsbereichs auf gleiche Weise wirksam. Es besteht keine Möglichkeit einer differenzierten Zuordnung individueller Infrastruktureigenschaften für einzelne Gruppen von Anwendungsinstanzen (→ *Instanzbereiche*).

Der *CHEOPS*-Kern geht in diesen vier Punkten weiter als andere Systeme mit *In-Kernel Servern*. Er ist damit nicht nur Untersuchungsgegenstand für die in dieser Dissertation vorgeschlagene anwendungsanpaßbare Architektur, und er ist auch nicht allein die experimentelle Basis für das *CHEOPS*-Projekt, sondern stellt auch neue Prinzipien für Betriebssystemkerne zur Diskussion.

- Vor allem für den Anwendungsbereich wird kein festes Ausführungsmodell und keine Menge fester Dienste in der Infrastruktur des Kerns vorgeschrieben. Es muß damit nicht notwendigerweise getrennte Adreßräume für Instanzen des Anwendungsbereichs geben. Das Prozessorscheduling kann, wenn es für bestimmte Zwecke sinnvoll ist, auch ohne Entzug oder für zeitkritische Phasen mit abschaltbarer Entziehbarkeit erfolgen. Damit wird der Mechanismus des Prozessorscheduling in weit höherem Maße flexibilisiert, als das allein durch anpaßbare Schedulingstrategien (z.B. feste Prioritäten oder "fair") möglich wäre. Dieselbe Flexibilität sollte auch für die Speicherverwaltung gelten, indem es nicht notwendigerweise getrennte Adreßräume und virtuellen Speicher geben muß. Das Kriterium für die Wahl eines Ausführungsmodells ist seine Angemessenheit bzw. sind die Anforderungen für den Zweck eines konkreten Zielsystems.
- Der verbleibende monolithische Teil des Kerns von *In-Kernel Servern* wird weiter reduziert, indem auch die Reaktion und die Behandlung von Interrupts in das Grundschema von Instanzen eingeordnet und dementsprechend implementiert wird (vgl. Anschnitt 5.3).
- Für den *CHEOPS*-Kern soll es als erste Stufe dynamischer Anpaßbarkeit tatsächlich ermöglicht werden, Kern-Instanzen (→ *In-Kernel Server*) dynamisch in das System zur Laufzeit einzubringen und zu entfernen und auf diese Weise weitreichende Wirkungen für den übergeordneten Bereich der Anwendungsinstanzen zu erreichen.
- Eine wichtige Eigenschaft ist auch, daß Gruppen von Anwendungsinstanzen (→ *Instanzbereiche*) von der Infrastruktur des *CHEOPS*-Kerns mit individuellen Diensten und Ausführungseigenschaften ausgestattet werden können.

Die beiden zuletzt genannten Punkte zielen vor allem auf Flexibilität und die Gestaltungsmöglichkeit des Kerns. Durch Einbau bestimmter Instanzen in den Kern können weitreichende Wirkungen für den übergeordneten Instanzbereich erzielt werden, denn die Instanzen im Kern bilden dessen Infrastruktur. Die Implementierung konzentriert sich aber vorerst auf die beiden ersten Punkte, um die Voraussetzungen dafür zu schaffen. Für die Ableitung und Spezialisierung der Architektur für den *CHEOPS*-Kern müssen jedoch diese Aspekte berücksichtigt werden, woraus sich auch die hier im Vorfeld geführte umfassende Diskussion begründet.

5.2 Spezialisierung der Architektur für den *CHEOPS*-Kern

Anhand der Zielstellung für den *CHEOPS*-Kern wird eine konkrete Architektur nach dem Muster der generalisierten Architektur abgeleitet bzw. spezialisiert. Dies erfolgt in den zwei Schritten:

1. Ableiten einer ***Ausprägungsvariante*** für den *CHEOPS*-Kern (→ *Schichten, Elemente in Schichten*) und

2. Zuordnen von ***Ausführungseigenschaften*** für die Elemente in den Schichten (→ *Dienste der Infrastruktur, Ausführungsmodell*).

5.2.1 Ausprägungsvarianten für den *CHEOPS*-Kern

Für die Festlegung einer Ausprägungsvariante bei der Spezialisierung einer konkreten Architektur wurde in Abschnitt 4.3.1 folgendes Schema angegeben.

- **pro System** – $\mathcal{IVS}$:
  - Wieviele Schichten $\mathcal{L}_i$ gibt es ?
- **pro Schicht** – $\mathcal{L}_i$:
  - Gibt es ausschließlich Basiselemente (globale Daten, Programme, Prozesse) ?
  - Gibt es auch oder ausschließlich Instanzen $\mathcal{I}_j$ bzw. auch Instanzbereiche $\mathcal{IB}$?
- **pro Instanz** – $\mathcal{I}_j$:
  - Gibt es im Inneren ausschließlich Basiselemente ?
  - Gibt es auch oder ausschließlich innere Instanzen $\mathcal{I}'_{j,k}$ (Aggregation) ?

Für den *CHEOPS*-Kern wird eine Architektur bestehend aus ***vier Schichten*** festgelegt. Innerhalb jeder der vier Schichten wird eine Instanzstruktur favorisiert. In den vorangegangenen Kapiteln wurden die Vorteile von Instanzen herausgearbeitet.

- Instanzen spiegeln wegen ihrer selbständigen Arbeitsweise das in anderen Bereichen der Technik oft anzutreffende Prinzip autonomer Teilsysteme wider:
  - Schaffung von Steuerungs- und damit von Abstraktionsebenen innerhalb einer Schicht ($\rightarrow$ *äußere, innere*);
  - lokale Konzentration und Kapselung semantisch zusammengehöriger (Teil-) Verarbeitungen; Entkopplung der Benutzung von der Realisierung der Funktionen;
  - Schaffung höherwertiger, komplexerer Elemente in einem System; Reduktion und Ordnung von Beziehungen zwischen diesen Elementen;
  - Förderung von Wiederverwendbarkeit; einfachere Testbarkeit.
- Instanzen sind ein wichtiges Mittel zur Strukturierung und für Abstraktion in ablaufenden Systemen ($\rightarrow$ *identifizierbare Einheiten, höhere Dienste*).
- Instanzen bieten für Betriebssysteme auch die Möglichkeit, eine Brücke in die Welt der Softwareherstellung zu schlagen. Das *Client-Server-Modell* ist seit langem ein bekanntes und akzeptiertes Programmierparadigma für verteilte und parallele Systeme. Ein Weg zur Integration der objektorientierten Programmierung wurde in Abschnitt 4.4 gezeigt.
- Durch Abtrennung der Ausführungseigenschaften vom Grundmodell der Instanzen wird das notwendige Maß an Skalierbarkeit hergestellt, um Instanzen als gleichbleibendes, dominierendes Strukturelement in *allen* Schichten des Systems anwenden zu können. Für anwendungsanpaßbare Betriebssysteme ist das eine wünschenswerte Eigenschaft.
- Instanzen sind sowohl für MPM- als auch für SMM-Programmiermodelle geeignet.
- Nicht zuletzt kann man auch auf eine gewisse Ästhetik dieser Architekturform verweisen (vgl. auch die Darstellung der *CHEOPS*-Architektur in Abbildung 5.1), obwohl dieser Aspekt für Architekturen technischer Systeme nicht im Vordergrund steht.

Das dominierende Merkmal der Architektur des *CHEOPS*-Kerns bilden Instanzen in vier Schichten, die in jeder Schicht unterschiedliche Ausführungseigenschaften aufweisen.

Schicht 4:	*aproc's*	– Anwendungsinstanzen,
Schicht 3:	*kproc's*	– Kern-Instanzen für ursächliche Betriebssystemfunktionen,
Schicht 2:	*iproc's*	– Kern-Instanzen zur Behandlung von Interrupts,
Schicht 1:	*ictrl*	– Instanz zur Steuerung der Interruptverarbeitung.

Zur Kennzeichnung der konkreten Architektur für den *CHEOPS*-Kern sollen diese vier Schichten näher charakterisiert werden.

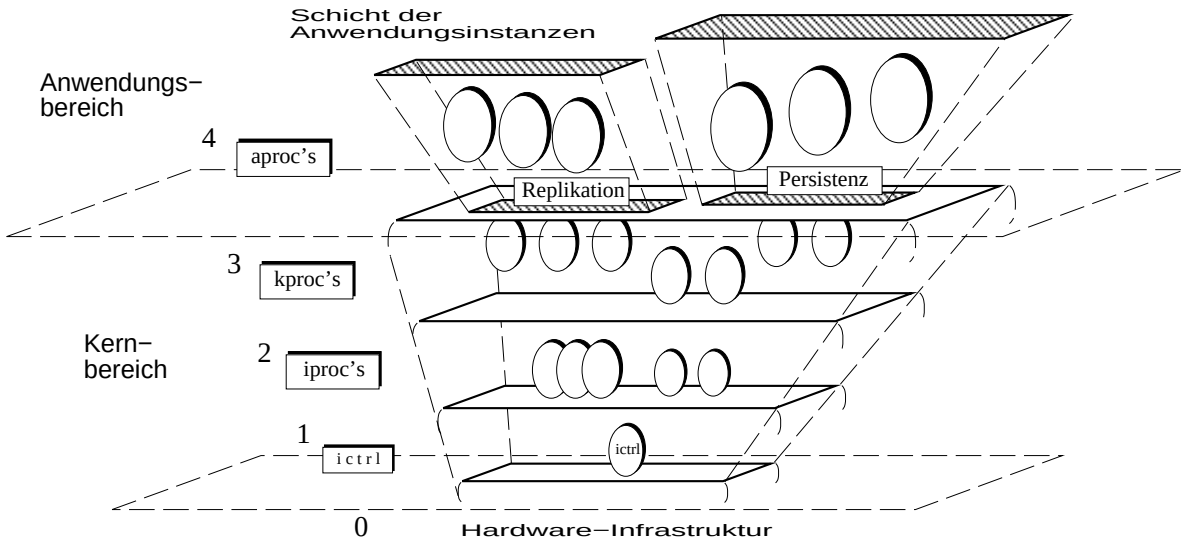


Abb.5.1 Vier-Schichten-Architektur des *CHEOPS*-Kerns

4 – *aproc's* : Für Instanzen des Anwendungsbereichs sind die Ausführungseigenschaften und Dienste des Kerns nicht in der Art festgelegt, wie man es bei anderen Kernen antrifft. Es sollen vielmehr verschiedenartige Formen durch entsprechende Gestaltung der Infrastruktur (→ *kproc*-Instanzen) offen sein. Als einzige Festlegungen der Architektur gilt für Anwendungsinstanzen, daß sie als Instanzen entsprechend der Definition umgesetzt sind und im *User-Mode* des Prozessors ablaufen. Konkrete Ausführungsmodelle werden dann durch die Infrastruktur des Kerns festgelegt und orientieren sich am Einsatzzweck.
Denkbare Ausführungsmodelle könnten sich an den Modellen bekannter Betriebssystemkerne orientieren (vgl. *Unix-Prozessen*, *Mach-Tasks*, *BirliX-Teams* o.a.), d.h. mit separaten Adreßräumen, preemptivem "fairem" Scheduling, virtuellem Speicher und im Falle von *BirliX-Teams* mit grundsätzlich persistenten und im System verteilten Instanzen.
Der Vorteil der Konzeption des *CHEOPS*-Kerns ist es gerade, daß im Prinzip alle diese Modelle durch entsprechende Gestaltung der Infrastruktur ***mögliche Optionen*** sind. Es lassen sich ebensogut leichtgewichtigere Ausführungsmodelle mit einem gemeinsamen Adreßraum, softwarebasiertem Zugriffschutz für Speicherbereiche von Instanzen (vgl. *Bridge*) oder mit Schutz der Kern-Integrität durch *page protection* für den Kern-Bereich umsetzen. Wenn es sinnvoll ist, sollte auch auf Zugriffschutz gänzlich verzichten werden können. Dieser Spielraum macht genau den höheren Grad der ***Flexibilität der Ausführungsmodelle*** aus, welcher den *CHEOPS*-Kern von anderen wesentlich unterscheidet.
Durch Gestaltung der Schicht *kproc's* läßt sich auch eine Infrastruktur für verteilte Systeme herstellen, so daß die heute übliche Form ersetzt werden könnte, Dienste für verteilte Systeme allein im Anwendungsbereich anzusiedeln und der Kern lediglich Kommunikationskanäle zwischen Prozessen bereitstellt (vgl. auch Abbildung 2.6).

In Abbildung 5.1 ist auch dargestellt, daß für den *CHEOPS*-Kern verschiedene Gruppen von Anwendungsinstanzen mit jeweils individuellen Infrastruktureigenschaften, vor allem für spezielle Dienste der Infrastruktur, ausgestattet werden können (in der Abbildung: Replikation und Persistenz). Auch diese vorteilhafte Eigenschaft resultiert aus der grundlegenden Entwurfsentscheidung für den *CHEOPS*-Kern, *keine Dienste* der Kern-Infrastruktur und auch *kein Ausführungsmodell* für Instanzen des Anwendungsbereichs fest vorzugeben, sondern die Kern-Infrastruktur einem Zweck nach gestalten zu können, indem entsprechende *kproc*-Instanzen in den Kern eingebracht werden.

3 – *kproc's* : Instanzen der Schicht drei befinden sich im Kern, teilen sich den gemeinsamen Kern-Adreßraum und arbeiten im *Kernel-Mode* des Prozessors. Diese Art von Instanzen ist vor allem für unmittelbare Funktionen des Betriebssystems als Infrastruktur für den übergeordneten Anwendungsbereich ausgelegt. Durch *kproc's* werden die Infrastrukturdienste und das Ausführungsmodell für den Anwendungsbereich umgesetzt.
Eine besondere Charakteristik des *CHEOPS*-Kerns ist, daß prinzipiell beliebige *kproc*-Instanzen für diese Schicht hergestellt werden können. Daraus resultiert das hohe Maß an Flexibilität des *CHEOPS*-Kerns für den Anwendungsbereich, da auf diese Weise Infrastruktur in wesentlichen Teilen für konkrete Anwendungszwecke angepaßt oder speziell hergestellt werden kann.

Kern-Instanzen oder *kproc's* sind für Gerätetreiber, zur Speicherverwaltung und zur Verwaltung und Herstellung von *aproc's* vorgesehen. Sie entsprechen den *In-Kernel Servern* in anderen Systemen. Sie sind als zyklische Instanzen konzipiert, die auf Anforderung Dienste ausführen. Die Interaktion erfolgt über Nachrichten und Signale. *kproc's* sind dynamisch startbar (vergleichbar zu Prozessen unter *Unix*, hier aber im Kern), wodurch dem Kern neue Dienste hinzugefügt werden. *kproc's* können auch dynamisch wieder entfernt werden. Eine Grundmenge *kproc's* existiert über die gesamte Laufzeit des Systems, z.B. Instanzen für bestimmte Gerätetreiber ($\rightarrow$ *Terminal*, *Platte*, *Netz*), für die Verwaltung der *aproc's* ($\rightarrow$ *Scheduler*, *Speicherverwaltung*) und für das dynamische Laden und Starten von *kproc*-Instanzen.

Zur Herstellung einer *kproc*-Instanz ist zur Laufzeit eine Beschreibungsinformation notwendig ($\rightarrow$ *Typinformation*: Code, initiale Daten $\rightarrow$ aus *C++* compilierte `.o`-Datei). Durch die Interpretation dieser Beschreibung kann ein neues Exemplar einer *kproc*-Instanz im System hergestellt werden. Typbeschreibungen sind dynamisch ladbar, so daß auch neuartige Instanzen während der Laufzeit in das System eingebracht werden können. Zur Herstellung individueller Dienste oder Ausführungseigenschaften für Teile des Anwendungsbereichs können *kproc's* gruppiert werden.

2 – *iproc's* : Jeder Interrupt wird von einer eigens dynamisch erzeugten *iproc*-Instanz behandelt. Die Behandlung eines Interrupts bedeutet die Ausführung genau eines Dienstes durch eine *iproc*-Instanz, die im Anschluß wieder entfernt wird. Dieses Vorgehen resultiert aus der Implementierungstechnik und hat Vorteile gegenüber der Variante mit zyklischen Instanzen (vgl. Abschnitt 5.3).
Instanzen für *iproc's* befinden sich mit *kproc's* im selben Kern-Adreßraum. Ihre Verwaltung und Ablaufsemantik ist auf die Behandlung von Interrupts spezialisiert, was sie von *kproc's* unterscheidet und eine separate Schicht im Kern für diese Aufgabe rechtfertigt. Die Steuerung der *kproc's* wird zu einem wesentlichen Teil von *iproc*-Instanzen bestimmt, da sie den wichtigen Teil des Interruptsystems zur Interaktion von Systemprozessen mit Prozessen anderer aktiver Elemente der Hardware bilden. Es gibt eine spezielle Instanz in dieser Schicht $\mathcal{I}_{ksched}$, welche die Repräsentationen der übergeordneten *kproc's* enthält und deren Ablaufsteuerung umsetzt.

1 – *ictrl* : Es gibt genau eine solche Instanz im Kern, bei der alle Interrupts eintreffen. Es wird je ein *iproc* für einen Interrupt erzeugt und die Abarbeitung der *iproc's* steuert. Diese Instanz existiert permanent während der gesamten Laufzeit des Kerns und wird bei jedem Interrupt durch den Interruptzyklus des Prozessors aktiviert.

0 – Instanzen der Hardware: Abbruch der Rekursion von Schichten für den *CHEOPS*-Kern.

Die Ausprägungsvarianten für den *CHEOPS*-Kern lassen sich anhand der in Abschnitt 4.3.1 eingeführten Beschreibung über $\mathcal{K}$–Mengen auch formal zusammenfassen.

	Schichten des *CHEOPS*-Kerns	innere Gliederung 1.Ebene (Instanzen – $\mathcal{I}$)	2.Ebene (innerhalb von $\mathcal{I}$)
Anwendungsbereich	4.a: ***aproc's*** $\mathcal{L}_4 = \{ \mathcal{I}^i_{aproc} \}$	$\mathcal{KI} = n$ (separate Adreßräume) $\mathcal{KP} = \mathcal{KZ} = 0,\ \mathcal{KP}_{rog} = \{1, n\}$ $\mathcal{KIB} = \{0, n\}$	$\mathcal{KP} = \{1, n\},\ \mathcal{KP}_{rog} = \mathcal{KZ} = 1$ (single / multi threaded)
	4.b: ***aproc's*** $\mathcal{L}_4 = \{ \mathcal{I}^i_{aproc} \}$	$\mathcal{KI} = n$ (gemeinsamer Adreßraum) $\mathcal{KP} = 0,\ \mathcal{KP}_{rog} = \mathcal{KZ} = \{1, n\}$ $\mathcal{KIB} = \{0, n\}$	$\mathcal{KP} = \mathcal{KZ} = \{1, n\},\ \mathcal{KP}_{rog} = 1$
Kernbereich	3: ***kproc's*** $\mathcal{L}_3 = \{ \mathcal{I}^i_{kproc} \}$	$\mathcal{KI} = n$ (zyklische Kern-Instanzen) $\mathcal{KP} = 0,\ \mathcal{KP}_{rog} = \{1, n\},\ \mathcal{KZ} = \{0, 1, n\}$ $\mathcal{KIB} = \{0, n\}$	$\mathcal{KP} = \mathcal{KP}_{rog} = 1,\ \mathcal{KZ} = \{1, n\}$ (single threaded)
	2: ***iproc's*** $\mathcal{L}_2 = \{ \mathcal{I}^i_{iproc} \} \cup \{ \mathcal{I}_{ksched} \}$	$\mathcal{KI} = n$ (Interruptbehandlungen) $\mathcal{KP} = 0,\ \mathcal{KP}_{rog} = 1,\ \mathcal{KZ} = \{1, n\}$ $\mathcal{KIB} = 0$	$\mathcal{KP} = \mathcal{KP}_{rog} = 1,\ \mathcal{KZ} = \{1, n\}$ (single threaded)
	1: ***ictrl*** $\mathcal{L}_1 = \{ \mathcal{I}^1_{ictrl} \}$	$\mathcal{KI} = 1$ ($\rightarrow$ *"singleton"*-Instanz) $\mathcal{KP} = \mathcal{KZ} = \mathcal{KP}_{rog} = 0$ $\mathcal{KIB} = 0$	$\mathcal{KP} = \mathcal{KZ} = \mathcal{KP}_{rog} = 1$ (single threaded)

Abb.5.2 Architekturbeschreibung für Ausprägungsvarianten des *CHEOPS*-Kerns

In ***Schicht*** $\mathcal{L}_1$ befindet sich genau eine Instanz $\mathcal{I}_{ictrl}$ ($\rightarrow \mathcal{KI} = 1$). Diese Instanz bildet gleichzeitig die gesamte Schicht und enthält alle Daten dieser Schicht ($\rightarrow \mathcal{KP} = \mathcal{KZ} = \mathcal{KP}_{rog} = 0$) und keine Instanzbereiche ($\rightarrow \mathcal{KIB} = 0$). Innerhalb von *ictrl* gibt es genau einen Prozeß, der durch den Interruptzyklus des Prozessors aktiviert wird. Es gibt innere Datenstrukturen und inneren Programmcode ($\rightarrow \mathcal{KP} = \mathcal{KZ} = \mathcal{KP}_{rog} = 1$).
Nach diesem Muster lassen sich die strukturellen Merkmale der *CHEOPS*-Architektur auch für die anderen Schichten beschreiben.

In ***Schicht*** $\mathcal{L}_2$ gibt es *iproc*-Instanzen zur Behandlung von Interrupts ($\rightarrow \mathcal{KI} = n$). Sie werden von *ictrl* erzeugt und gesteuert. Außer *iproc's* gibt es keine weitere Prozesse in dieser Schicht ($\rightarrow \mathcal{KP} = 0$). Die Programme der *iproc's* sind die Interruptbehandlungsroutinen, die für gleichartige Interrupts mehrfach verwendet werden, so daß ein globales Programm mit allen Interruptbehandlungsroutinen in dieser Schicht existiert ($\rightarrow \mathcal{KP}_{rog} = 1$). Die Zustandsmenge in dieser Schicht ist zum Teil für Interaktionszwecke global. Das betrifft insbesondere eine spezielle Instanz $\mathcal{I}_{ksched}$, die nicht zum Zweck der Behandlung von Interrupts existiert, sondern die übergeordneten *kproc's* steuert. Diese Instanz hat mit den anderen *iproc's* gemeinsame Zustände, damit *kproc's* bei Interruptbehandlungen Ereignisse signalisiert werden können, indem durch *iproc's* Zustände in den Repräsentationen von *kproc's* in der Instanz $\mathcal{I}_{ksched}$ verändert werden (z.B. das Aktivieren wartender *kproc's*). Aus diesem Grund umfaßt die Zustandsmenge sowohl private Zustände der *iproc's* als auch gemeinsame Zustände ($\rightarrow \mathcal{KZ} = \{1, n\}$).
Innerhalb von *iproc's* gibt es jeweils nur einen Prozeß, ein Programm ($\rightarrow$ Behandlungsroutine) und private bzw. mit $\mathcal{I}_{ksched}$ geteilte Zustände ($\rightarrow \mathcal{KP} = \mathcal{KP}_{rog} = 1,\ \mathcal{KZ} = \{1, n\}$).

In ***Schicht*** $\mathcal{L}_3$ befinden sich *kproc*-Instanzen als *Server*-Instanzen im Kern-Bereich, um dort typische Betriebssystemaufgaben auszuführen. Sie werden in der darunterliegenden Schicht $\mathcal{L}_2$ innerhalb der Instanz $\mathcal{I}_{ksched}$ repräsentiert und von dieser global gesteuert. Instanzen dieser Art können, wie in den darunterliegenden Schichten, gemeinsamen Programmcode besitzen und ausführen. Der Programmcode kann sich permanent im Kern-Adreßraum befinden oder für spezielle *kproc*-Instanzen individuell in den Kern geladen werden ($\rightarrow \mathcal{KP}_{rog}=\{1, n\}$).
Für die Zustandsmenge sind alle Varianten zugelassen ($\rightarrow \mathcal{KZ}=\{0, 1, n\}$), d.h., *kproc's* können sowohl gemeinsame Zustände mit anderen *kproc's* oder auch mit *iproc's* als auch private Zustände im Sinne der Definition von Instanzen besitzen. Dies ist beispielsweise für Gerätetreiberinstanzen sinnvoll, die eng mit Instanzen der Interruptbehandlung zusammenwirken. Außerdem soll in dieser Schicht die Möglichkeit der Gruppierung von *kproc*-Instanzen in Instanzbereiche $\mathcal{IB}$ möglich sein ($\rightarrow \mathcal{KI}=\{0, n\}$).
Innerhalb von *kproc's* gibt es genau einen Prozeß ($\rightarrow \mathcal{KP}=\mathcal{KP}_{rog}=1$). Die Zustandsmenge kann privat oder für Interaktionszwecke partiell gemeinsam mit anderen *kproc*- oder *iproc*-Instanzen sein ($\rightarrow \mathcal{KZ}=\{1, n\}$). Ähnlich wie in Schicht $\mathcal{L}_2$ gibt es auch in Schicht $\mathcal{L}_3$ eine oder mehrere *kproc*-Instanzen zur Repräsentation und Ablaufsteuerung der übergeordneten Schicht $\mathcal{L}_4$.

Für die ***Schicht*** $\mathcal{L}_4$ werden zwei Varianten (4.a und 4.b) angegeben. Im Gegensatz zu anderen Systemen bestand ein Entwurfsziel für den *CHEOPS*-Kerns gerade darin, nicht nur *eine Variante* mit genau *einem Ausführungsmodell* bereitzustellen, sondern durch variable Gestaltung der darunterliegenden Schichten, ***alternative Varianten und Ausführungsmodelle*** für den Anwendungsbereich zu ermöglichen.

Variante (4.a) orientiert sich an den klassischen μ-Kernen. Das wesentliche Merkmal ist, daß alle *aproc*-Instanzen separate Adreßräume und damit keine gemeinsamen Zustände besitzen. Außerhalb von *aproc*-Instanzen gibt es keine anderen Prozesse. Anwendungsinstanzen sind dynamisch startbar und können zu Instanzbereichen gruppiert werden. Eine wichtige Neuerung ist, daß Gruppen von *aproc*-Instanzen mit individuellen Eigenschaften der Infrastruktur ausgestattet werden können, d.h. vor allem mit individuellen Diensten, aber auch mit verschiedenen Ausführungsmodellen. In Abbildung 5.1 wurde dieses Prinzip an einem Beispiel gezeigt, daß eine Gruppe von *aproc*-Instanzen die Ausführungseigenschaft Persistenz und die andere Gruppe die Eigenschaft einer replizierten Existenz auf mehreren Knoten in einem verteilten System aufwies. Um diese Wirkungen zu erreichen, müssen in der Infrastruktur entsprechende *kproc*-Instanzen vorhanden sein. Ein Ziel des *CHEOPS*-Projekts liegt genau bei dieser Art der Flexibilisierung von Ausführungsmodellen. In diese Kategorie könnte auch ein "wahlweise zuschaltbarer" virtueller Speicher als Teil eines anderen Ausführungsmodells eingeordnet werden.
Innerhalb einer *aproc*-Instanz kann es einen oder mehrere innere Prozesse ($\rightarrow$ *Threads*) geben, die eine gemeinsame Zustandsmenge besitzen.

Variante (4.b) ist dadurch gekennzeichnet, daß *aproc*-Instanzen keine separaten Adreßräume und damit ein einfacheres Ausführungsmodell als in Variante (4.a) aufweisen. Eine Zielsetzung für dieses Modell kann darin bestehen, inwieweit die Vereinfachung des Ausführungsmodells zur Reduktion des Umfangs des Kerns und auch zur Steigerung der Effizienz beitragen kann und welche Möglichkeiten bestehen, die Schutzwirkung getrennter Adreßräume auch anders für Instanzen zu erreichen. In [Lucco93, NL96] werden Wege für softwareseitigen Schutz gezeigt (vgl. Abschnitt 2.7). Zugriffschutz auf Speicherbereiche könnte aber auch über *page protection* innerhalb eines Adreßraumes umgesetzt werden. Eine andere Frage ist, ob der Aufwand für die Abschottung von Speicherbereichen für Anwendungsinstanzen in jedem Fall gerechtfertigt ist.

Die Kategorie der *Single Address Space Operating Systems* wurde ebenfall in Abschnitt 2.7 als eine (wieder) aktuelle Entwicklung diskutiert, welche allerdings eine andere Zielrichtung verfolgt, nämlich die neuen Möglichkeiten von 64-Bit Adreßräumen für die globale Identifikation

in verteilten Systemen einzusetzen. Für *CHEOPS* stünde ein anderer Aspekt im Vordergrund, inwieweit die Vereinfachung des Ausführungsmodells zur Effizienzsteigerung führen kann bzw. welche Konsequenzen sich für den Betrieb solcher Systeme ergeben. Es sei daran erinnert, daß es in der Vergangenheit wegen mangelnder Hardwareunterstützung eine Vielzahl von Betriebssystemen gab, die keine getrennten Adreßräume umsetzen konnten. Heute ist Hardwareunterstützung für die Adressierung zwar gegeben, aber der Verwaltungsaufwand im Betriebssystem ist dennoch beachtlich und trägt nicht unwesentlich zur Komplexität der Kerne bei.

Es stellt sich die Frage, worin letztendlich noch der Unterschied zwischen *kproc*- und *aproc*-Instanzen liegt, wenn es keine getrennten Adreßräume in Variante *(4.b)* für *aproc's* gibt. Der Unterschied begründet sich aus der Schichtenstruktur, daß *kproc*-Instanzen die Infrastruktur für *aproc's* herstellen. Diese Infrastruktur schließt nicht allein das Prozeßmodell ein, welches in dieser Variante dem Modell der *kproc's* in der Tat sehr ähnlich ist. Aber selbst hier kann es Unterschiede für das Scheduling und die Art der Umschaltung geben. Es gilt auch, daß die CPU für *aproc's* im *User-Mode* und für *kproc's* im *Kernel-Mode* arbeitet.
Auf diese Weise entstehen durchaus verschiedene Eigenschaften des Prozeßmodells, auch wenn das Merkmal eines gemeinsamen Adreßraumes für beide – *kproc's* und *aproc's* – gleich ist.

Die Untersuchungsmöglichkeiten für beide Varianten in der Schicht *aproc's* sind Optionen für die Fortsetzung des *CHEOPS*-Projekts. Sie sind nicht Bestandteil dieser Dissertation. Hier soll aber die Grundlage dafür geschaffen werden, weshalb diese möglichen Ziele an dieser Stelle mit berücksichtigt und diskutiert werden sollten. Sie bestimmen auch maßgeblich die Entwurfsentscheidungen für den Kern.

5.2.2 Ausführungseigenschaften – Dienste der Infrastruktur

Die Spezialisierung der Ausführungseigenschaften für die Architektur des *CHEOPS*-Kerns soll nach einem Schema vorgenommen werden, welches drei Grundmerkmale beschreibt, die im Zusammenhang mit Infrastrukturdiensten wesentlich sind.

- Welche ***Dienste*** gibt es in den zwei Kategorien (vgl. Abschnitt 2.2.2):
  - *existentielle Dienste* (Erzeugung, Beseitigung von Elementen, Ablaufsteuerung, Interaktion),
  - *optionale Dienste*?
- Welche ***Identifikation*** gibt es für Dienste, Dienstausführungen und Dienstleister (→ *gesamte Schicht* oder auch *individuelle Instanzen* in der Infrastruktur)?
- Wie erfolgt die ***Interaktion*** zur Veranlassung und Ausführung von Diensten?
  - Wie ist das *Protokoll* zwischen Auftraggeber und Dienstleister gestaltet?
  - Welche *Schnittstelle* existiert für diesen Zweck mit der Infrastruktur?

Für die einzelnen Schichten sollen diese Merkmale diskutiert werden, soweit sie zum unmittelbaren Grundgerüst des *CHEOPS*-Kerns gehören, d.h., es werden im einzelnen die Schichten $\mathcal{L}_1$ (*ictrl*) bis $\mathcal{L}_3$ (*kproc's*) diskutiert. Spezielle Erweiterungen des *CHEOPS*-Kerns, optionale Dienste und mögliche Varianten der Anwendungsschicht (4.a und 4.b aus Abschnitt 5.2.1) werden nicht gezeigt. Es handelt sich bei diesen um künftige Untersuchungs- und Gestaltungsvarianten des *CHEOPS*-Kerns, die gegenwärtig teilweise noch in der Phase der Diskussion sind. Das Grundgerüst des *CHEOPS*-Kerns besteht aus wenigen Elementen und elementaren Mechanismen, wie der Behandlung von Interrupts über *iproc*-Instanzen. Dieses Grundgerüst wird während des ***Bootprozesses*** (vgl. Abschnitt 5.5) aufgebaut und existiert während der gesamten Laufzeit des Kerns. Auch die Mittel für Interaktion sind auf ein elementares Maß beschränkt.

Schicht $\mathcal{L}_1$ – *ictrl*

- *Zweck:*
  - Zustand der unterbrochenen Instanz im Prozessor retten und Prozessorentzug,
  - *iproc* erzeugen und anschließend den Interruptzyklus der CPU beenden,
  - *Re-Scheduling* und Umschalten zur nachfolgenden Instanz;
- *Charakteristik:*
  - es gibt genau eine Instanz $\mathcal{I}_{ictrl}$ ($\mathcal{L}_1 = \mathcal{I}_{ictrl}$),
  - $\mathcal{I}_{ictrl}$ existiert permanent während der gesamten Laufzeit des Kerns;

- ***Dienste***

Dienste der Schicht $\mathcal{L}_1$ bzw. Dienste der Instanz *ictrl* wirken implizit durch das Verhalten des Interruptsystems der CPU. Dienstausführungen werden nicht explizit aufgerufen, sondern als Reaktion des Prozessors auf eintreffende Interrupts veranlaßt. Diese *impliziten Dienste* sind:

- `isavregs` – Retten der in der CPU enthaltenen Zustände der unterbrochenen Instanz,
- `inew` / `ifree` – Erzeugen und Beseitigen von *iproc's* bei Interrupts,
- `isched` / `iswitch` – Scheduling und Umschalten zu *iproc's*.

Innerhalb von *ictrl* gibt es eine Datenstruktur `icbs` *(iproc control blocks)*, über der die Dienste `inew` / `ifree` ausgeführt werden. Während des Interruptzyklus des Prozessors wird lediglich der Zustand in der CPU der unterbrochenen Instanz gerettet und ein neuer *iproc* angelegt. Danach wird der Interruptzyklus beendet, aber nicht zur unterbrochenen Instanz zurückgekehrt, sondern es wird ein Re-Scheduling als Dienst der Instanz *ictrl* ausgeführt, bei welchem die folgende zu aktivierende Instanz bestimmt und dann umgeschaltet wird (vgl. Abschnitt 5.3).
Es können keine *iproc's* auf anderem Wege als durch Interrupts erzeugt werden, so daß es keine "expliziten" Dienste für Instanzen höherer Schichten dafür gibt. Es gibt zwei Ausnahmen, zum einen die Operation `itrap` und zum anderen die Instanz $\mathcal{I}_{ksched}$. Durch `itrap` wird ein Interruptzyklus des Prozessors synchron durch eine übergeordnete Instanz ausgelöst, für den in analoger Weise ein *iproc* erzeugt wird. Die Instanz $\mathcal{I}_{ksched}$ befindet sich in derselben Schicht wie *iproc's*. Sie wird während des Bootprozesses erzeugt und existiert permanent für die Laufzeit des Kerns. Ihre Aufgabe besteht in der Verwaltung und Ablaufsteuerung von *kproc's*.

Vier Dienste können von *iproc*- und *kproc*-Instanzen *explizit* "aufgerufen" werden:

- `iEI` / `iDI` – Interrupts für die CPU erlauben bzw. verhindern,
- `itrap( args )` – Interruptzyklus der CPU auslösen, *iproc* erzeugen und Re-Scheduling,
- `ilock` / `iunlock` – Interrupts sind erlaubt, es werden auch *iproc's* erzeugt, aber die Steuerung kehrt direkt, d.h. *ohne Re-Scheduling* an die unterbrochene Instanz zurück,
- `iassign( no, (* code)(), ... )` – Behandlungsroutine für den Interrupttyp `no` zuweisen.

`iEI`/`iDI` und `itrap` sind Maschinenbefehle, die von allen Instanzen im Kern ausgeführt werden können. Die Dienste `ilock`/`iunlock` sind durch atomares Setzen bzw. Rücksetzen einer global im Kern bekannten Variablen in *ictrl* (= gemeinsamer Zustand mit anderen Kern-Instanzen) implementiert. Ihr Wert entscheidet in *ictrl*, ob nach dem Erzeugen eines *iproc's* ein Re-Scheduling erfolgt oder ob sofort, d.h. noch *vor* der eigentlichen Behandlung des Interrupts durch den *iproc*, zur vorher unterbrochenen Instanz zurückgekehrt wird. Über `iassign` kann der Programmcode für Behandlungsroutinen von *iproc's* zugewiesen werden (Eintrag in der Interruptvektortabelle).

- ***Identifikation*** – *Dienste, Dienstausführungen und Dienstleister*

Die impliziten Dienste von *ictrl* sind für andere Instanzen nicht identifizierbar, da ihre Ausführung durch das Interruptsystem des Prozessors veranlaßt wird. Die expliziten Dienste

sind ebenfalls nicht im Sinne der Identifikation eines gemeinsamen Zugriffspunktes für einen Dienstaufruf identifizierbar. Die Identifikation ist durch die Maschinenbefehle und durch Kenntnis interner Adressen gegeben (`lock`-Variable und Interruptvektor, vgl. Abbildung 5.3).

- ***Interaktion*** – *Protokolle und Schnittstellen*

Für die Interaktion gibt es in dieser Schicht nur die elementare Form atomarer Maschinenbefehle für die expliziten Dienste, deren Aufruf in *C++* über `inline static member`-Methoden objektloser Klassen beschrieben wird. Der Dienstaufruf und auch die Ausführung sind folglich beide in der aufrufenden Instanz enthalten. Die impliziten Dienste werden über das Interruptsystem aktiviert. Die Schnittstelle als gemeinsames Element wäre in diesem Fall die gemeinsame `lock`-Variable in *ictrl* und der Interruptvektor. Diese elementaren (minimalen) Dienst- und Interaktionsformen sind in der Schicht *ictrl* für die hier unterstellten Zielplattformen ausreichend. Andere Interaktionsformen wären zu aufwendig und sind daher unangemessen.

Schicht $\mathcal{L}_2$ – *iproc's*

- *Zweck:*
  - jeder Interrupt wird durch eine eigens erzeugte *iproc*-Instanz behandelt,
  - *iproc's* führen den Code von Interruptbehandlungsroutinen aus (= Dienst),
  - die Instanz $\mathcal{I}_{ksched}$ dient zur Repräsentation und Ablaufsteuerung von *kproc's*;
- *Charakteristik:*
  - *iproc's* sind äußerst leichtgewichtige Instanzen,
  - *iproc's* sind prinzipiell wie andere Instanzen unterbrechbar;

- ***Dienste***

In der Schicht der *iproc's* werden zwei wesentliche Funktionen erbracht. Zum einen erfolgt hier die Interruptverarbeitung durch Ablauf von dynamisch erzeugten *iproc*-Instanzen, zum anderen werden durch eine spezielle Instanz $\mathcal{I}_{ksched}$ in dieser Schicht die übergeordneten *kproc*-Instanzen verwaltet und gesteuert. Der Ablauf von *iproc*-Instanzen kann als ein impliziter Dienst dieser Schicht angesehen werden. Für die Herstellung und Steuerung von *kproc*-Instanzen gibt es aber auch explizite Dienste, die in der Instanz $\mathcal{I}_{ksched}$ und damit für *kproc's* wirksam werden.

- *Dienste zur Erzeugung und Beseitigung von Elementen:*
  - `knew( code, prio )` → `kid` – Erzeugen eines *kproc*,
  - `kdelete( kid )` – Entfernen eines *kproc*,
  - `kmyid` → `kid` – liefert den `kid` des aufrufenden *kproc*,
- *Dienste zur Steuerung des Ablaufs:*
  - `kprio( kid, prio )` – Setzen der Priorität von *kproc*-Instanzen,
- *Dienste für Signalisierungs-Interaktion zwischen kproc's:*
  - `kwaits` – Warten auf ein Signal,
  - `ksig( kid )` – Senden eines Signals an den *kproc* `kid` ohne Re-Scheduling,
  - `ksigrs( kid )` – Senden eines Signals an den *kproc* `kid` mit Re-Scheduling.

Diese Dienste für *kproc's* sind Operationen über einer Datenstruktur `kcbs` *(kproc control blocks)*. Anhand dieser Datenstruktur wird beim Scheduling von *kproc's* in der Instanz $\mathcal{I}_{ksched}$ entschieden, welcher *kproc* als nächster arbeiten wird. Für das Prozessorscheduling von *kproc's* werden *statische Prioritäten* verwandt, wobei diese Prioritäten über `kprio( kid, prio )` für *kproc's* zu beliebigen Zeitpunkten gesetzt werden können. Für die Interaktion wird ein einfacher Signalisierungsmechanismus mit untypisierten Signalen eingesetzt.

Diese Operationen können nicht nur von *kproc's* benutzt werden, sondern auch von *iproc's*. Das ist insbesondere für die Signalisierung von *iproc's* an *kproc's* wichtig, so daß wartende *kproc's* nach Interrupts aktiviert werden können. Der Unterschied zwischen `ksig` und `ksigrs` besteht darin, daß mit `ksig` lediglich der Übergang `WAITING` → `READY` für den Ziel-*kproc* in der Verwaltungsdatenstruktur vorgenommen wird, wohingegen bei `ksigrs` zusätzlich ein Re-Scheduling über `itrap` veranlaßt wird, bei dem die initiierende Instanz die Steuerung verliert.

- ***Identifikation*** – *Dienste, Dienstausführungen und Dienstleister*

Identifizierbare Einheiten sind in der Schicht *iproc's* intern die *iproc's* selbst. Für übergeordnete Schichten sind *iproc's* nicht identifizierbar. Die zentrale Instanz zur Verwaltung von *kproc's* ist $\mathcal{I}_{ksched}$. Da es nur eine solche Instanz gibt, ist sie nach außen zwar bekannt, aber nicht in einem unterscheidenden Sinne identifizierbar. Die expliziten Dienste dieser Schicht sind für *iproc's* und *kproc's* aufrufbar und deshalb auch identifizierbar. Wegen der besonderen Interaktionsart sind individuelle Dienstausführungen nicht identifizierbar.

- ***Interaktion*** – *Protokolle und Schnittstellen*

Die Interaktion zum Aufruf und zur Ausführung von Diensten für *kproc's* ist einfach gestaltet. Diese Dienste bewirken ausschließlich Veränderungen in der Verwaltungsdatenstruktur `kcbs`. Die Operationen `kdelete`, `kwaits` und `ksigrs` veranlassen an ihrem Ende zusätzlich ein Re-Scheduling. Die anderen Operationen werden erst bei einem folgenden Re-Scheduling wirksam, welches durch den nächsten Interrupt veranlaßt wird.

In der Schicht *iproc's* gibt es die Instanz $\mathcal{I}_{ksched}$, welche das Scheduling und die Umschaltungen für *kproc's* anhand der Datenstruktur `kcbs` vornimmt. Aus struktureller Sicht wäre es naheliegend, daß diese Instanz auch die Ausführung der genannten Dienste für *kproc's* vornimmt. Zu diesem Zweck müßten jedoch Umschaltungen *aufrufende Instanz* $\rightleftharpoons \mathcal{I}_{ksched}$ ausgeführt werden. Diese Umschaltungen ließen sich auch einfach umsetzen, indem z.B. mit einem `itrap` ein *iproc* erzeugt wird, dessen Zweck es aber lediglich wäre, sich zu beenden, um danach zu $\mathcal{I}_{ksched}$ weiterschalten zu können. Dieses Vorgehen ist zu aufwendig. Aus diesem Grunde wurde eine Entkopplung der Dienstausführungen von der Instanz $\mathcal{I}_{ksched}$ vorgenommen.
Da es sich lediglich um Operationen über der Verwaltungsdatenstruktur `kcbs` für *kproc's* handelt, findet keine aufwendige Interaktion zwischen der veranlassenden Instanz und $\mathcal{I}_{ksched}$ statt, so daß $\mathcal{I}_{ksched}$ diese Operationen ausführen könnte, sondern die veranlassende Instanz führt diese Operationen selbst über `kcbs` aus. Damit wird die Datenstruktur `kcbs` zu einem gemeinsamen Zustand innerhalb des Kerns. Auch $\mathcal{I}_{ksched}$ arbeitet über dieser Datenstruktur für das Prozessorscheduling von *kproc's*. Da es sich um eine gemeinsame Datenstruktur handelt, ist `kcbs` nicht mehr Teil der Instanz $\mathcal{I}_{ksched}$. Die "sauberere" Lösung mit einer ausschließlich lokalen Datenstruktur innerhalb dieser Instanz hätte jedoch zu dem oben erklärten hohen Aufwand bei der Instanz-Interaktion geführt.
Die Datenstruktur `kcbs` wird als ***Monitor*** umgesetzt, welcher die genannten Dienste als exklusive Operationen implementiert. Jene drei Operationen, die ein Re-Scheduling bewirken, führen an ihrem Ende die Operation `itrap` aus, so daß über den Interruptzyklus des Prozessors das Re-Scheduling veranlaßt wird. Der Monitor `kcbs` bildet damit auch die Schnittstelle für die Dienste der Schicht *iproc's* für Schicht *kproc's*.

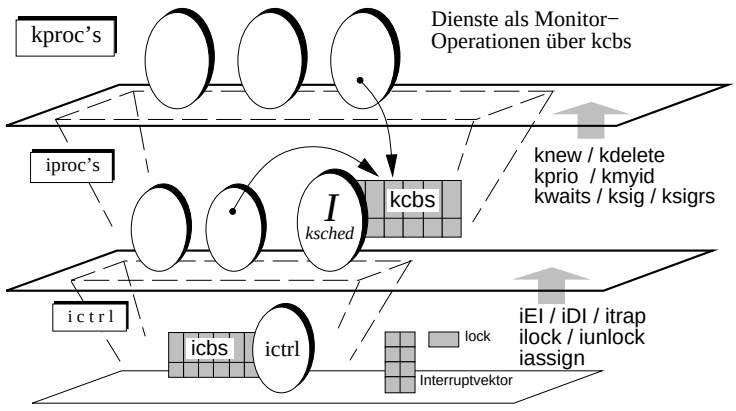


Abb.5.3 Interaktion für Dienste der Schicht *iproc's*

Die Interaktionsform zur Ausführung von Diensten der Schicht *iproc's* über Monitore ist eine pragmatische Gestaltung für die speziellen Gegebenheiten der Zielplattformen des *CHEOPS*-Kerns. Eine flexible, anpaßbare Architektur bietet diesen Gestaltungsspielraum. Insofern sind Instanzen zwar die hier favorisierte Gliederungsform innerhalb von Schichten, aber sie sind nicht die alleinige. Globale Daten oder – wie im Fall der Verwaltungsdatenstruktur für *kproc's* – globale Monitore sind ebenfalls geeignete Mittel, Interaktionsmöglichkeiten für Diensterbringungen zwischen Schichten innerhalb des *CHEOPS*-Kerns auf effektivem Wege umzusetzen.

Die in Kapitel 4 eingeführte generalisierte Architektur besitzt den hierfür erforderlichen Gestaltungsspielraum, indem es neben Instanzen auch gemeinsame Basiselemente geben kann. Bei der Spezialisierung einer konkreten Architektur kann dieses Merkmal dann angewandt werden, wenn dies für die Eigenschaften des Zielsystems vorteilhaft ist, wie an dieser Stelle für den gemeinsamen Monitor. Im Fall des *CHEOPS*-Kerns konnte daher die Interaktionsform für Dienstnutzungen innerhalb des Kerns äußerst leichtgewichtig gestaltet werden.

Die Exklusivität der Monitor-Operationen wird durch die Dienste `iEI/iDI` erreicht, da die Zielmaschinen nur jeweils einen Prozessor besitzen, auf dem der *CHEOPS*-Kern läuft und daher kein anderer als der dienstausführende Prozeß eine Monitor-Operation ausführen kann. Auch diese Lösung ist an die Gegebenheiten des *CHEOPS*-Kerns und an die der Zielmaschinen angepaßt.

- ***Weitere Dienste*** – *Interaktion über Nachrichten*

Die oben genannten Dienste der Schicht *iproc's* sind für die Ablaufsteuerung der übergeordneten Schicht *kproc's* ausreichend. Mit Signalisierung und gemeinsamen Zuständen lassen sich sinnvolle Interaktionsformen zwischen *kproc's* implementieren. Um die Interaktionsmöglichkeiten zu verbessern, wird in der Schicht *iproc's* ein einfacher Mechanismus für unidirektionale Nachrichten eingeführt. Das Grundprinzip ist den *Message Queues* von *Unix* [Bac86, GC94] nachempfunden, aber in einer wesentlich einfacheren Form, welche speziell für die Kommunikation von Instanzen im *CHEOPS*-Kern angepaßt wurde.
Es werden Nachrichten mit fester Länge (4 Byte) verwandt, die zwischen einem oder mehreren Sendern (*iproc's* oder *kproc's*) und einem oder mehreren Empfängern (*kproc*) über einen gemeinsam identifizierten Kanal (→ *Message Queue*) verschickt werden. Das Nachrichtensystem besitzt eine feste Menge an Kanälen, über die Instanzen Nachrichten (a)synchron senden können.

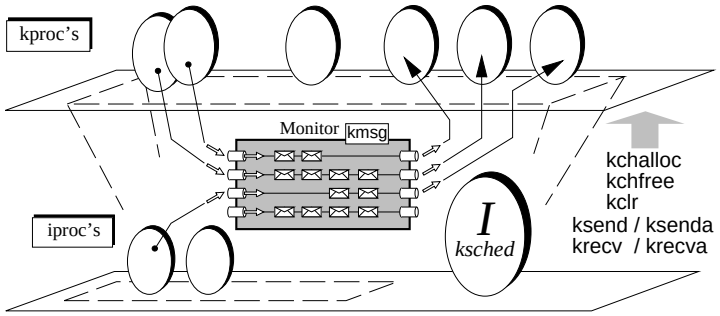


Abb.5.4 Interaktion im Kern über Nachrichten

Das Absenden von Nachrichten ist für alle *iproc's* und *kproc's* möglich. Nachrichten empfangen können dagegen nur *kproc's* und $\mathcal{I}_{ksched}$. Sowohl die Anzahl der Kanäle (→ N_k) als auch deren maximale Größe (→ *Kanalkapazität* K_i) werden statisch konfiguriert und sind nicht dynamisch änderbar. Die feste Anzahl von Kanälen kann aber von den Instanzen dynamisch belegt und freigegeben werden. Es gibt einen Dienst zur Anforderung und Belegung eines Kanals mit einer vorgegebenen Mindestkapazität. Belegte Kanäle werden markiert (Markierung $\neq 0$ – belegt, 0 – frei). Vergabe und Bedeutung der Marken liegen in der Verantwortung der Instanzen. Identifiziert werden Kanäle über `chid`'s. Kommunikationspartner werden nicht identifiziert.
Sende- und Empfangsoperationen gibt es jeweils in einer *synchronen* und *asynchronen* (versuchenden) Variante, so daß im synchronen Fall ein Sender bzw. Empfänger blockiert, wenn der Kanal voll belegt bzw. leer ist. Das Blockieren wird auf den oben erklärten Signalisierungsmechanismus für *kproc's* abgebildet. *iproc*-Instanzen dürfen keine blockierenden Operationen

ausführen, nur asynchron Nachrichten an *kproc's* senden. Die Interaktion von *iproc's*- bzw. *kproc's* zum Senden oder Empfangen von Nachrichten erfolgt mittels exklusiver Operationen des ***Monitors*** `kmsg`, d.h., die Sende- und Empfangsoperationen werden durch die beteiligten Instanzen selbst ausgeführt. Pro Kanal gibt es zwei Speicherplätze für `kid`'s eventuell wartender *kproc's* bei synchronem Senden bzw. Empfangen. Hebt eine folgende Sende- oder Empfangsoperation eines Kommunikationspartners die Wartebedingung auf, wird der blockierten Instanz dieses Ereignis signalisiert. Auch hier bietet das Nachrichtensystem nur diesen primitiven Mechanismus. Instanzen müssen in einer sinnvollen und koordinierten Weise davon Gebrauch machen.

Diese auf den ersten Blick harten Einschränkungen sind aber auf den Zielbereich einer leichtgewichtigen Nachrichten-Interaktion zwischen Instanzen im Kern (→ *iproc's* und *kproc's*) zugeschnitten. Jeder weitere "Komfort", wie Kanäle mit dynamisch veränderbarer Kapazität oder dynamisch erzeugbare Kanäle, verkompliziert das Nachrichtensystem in der Schicht *iproc's* und geht zu Lasten von Einfachheit und Effizienz. Kanäle mit fester Kapazität, d.h. mit einer festen Anzahl von Speicherplätzen für Nachrichten á 4 Byte, können einfach über Felder implementiert werden. Das Senden und Auslesen von Nachrichten erfolgt durch Operationen über zwei zyklischen Feldindizes eines Kanals $\texttt{ch}_{chid}$:

`send( chid, msg ):` $\texttt{msg} \rightarrow \texttt{ch}_{chid}\texttt{[si = ++si \% N]; n++;}$ — `N` – feste Kanalkapazität K_{chid},

`recv( chid ) → msg:` $\texttt{ch}_{chid}\texttt{[ri = ++ri \% N]} \rightarrow \texttt{msg; n--;}$ — `n` – aktueller Füllstand [0..N-1].

Listenoperationen und eine dynamische Speicherverwaltung für Speicherplätze von Nachrichten sind nicht notwendig. Die Sende- und Empfangsoperationen sind dadurch sehr einfach, so daß deren Exklusivität in analoger Weise zu `kcbs` durch "kurze kritische Abschnitte" in den Monitor-Operationen über `iEI/iDI` hergestellt werden kann.

In der Schicht *iproc's* gibt es für den Nachrichtenmechanismus die folgenden Dienste:

- *Dienste für Nachrichten-Interaktion zwischen iproc's → kproc's, kproc's ⇌ kproc's:*
  - `ksend( chid, msg )`[1] – synchrones Senden eine Nachricht `msg` in den Kanal `chid`; falls der Kanal vollständig belegt ist, blockiert der Sender;
  - `ksenda( chid, msg )` → { 0, -1 } – asynchrones, nichtblockierendes Senden; falls der Kanal vollständig belegt ist, wird -1 geliefert, sonst 0;
  - `krecv( chid )`[1] → `msg` – synchrones Empfangen eine Nachricht; falls keine Nachricht vorhanden ist, blockiert der Empfänger;
  - `krecva( chid, *msg )`[1] → { 0, -1 } – asynchrones, nichtblockierendes Empfangen; falls der Kanal leer ist, wird -1 geliefert, sonst 0;
  - `kclr( chid )` – Löschen aller Nachrichten des Kanals `chid`;
- *Dienste für die dynamische Zuteilung von Kanälen:*
  - `kchalloc( cap, mark )`[1] → { `chid`, -1 } – Belegen eines freien Kanals der Mindestkapazität `cap` mit der Marke `mark`; es wird `chid` oder -1 im Fehlerfall zurückgegeben;
  - `kchfree( chid )`[1] – Freigabe des Kanals `chid` (0 → `chid.mark`);
  - `kchmark( chid )`[1] → `mark` – Rückgabe der Markierung des Kanals `chid`.
  - `kchid( mark )`[1] → `chid` – Rückgabe des `chid` eines Kanals mit der Marke `mark`.

Die sinnvolle Anwendung dieses einfachen Nachrichtenmechanismus geht von einem kooperativen Verhalten aller beteiligten Instanzen aus.

[1] Dieser Dienst darf nur von *kproc's* benutzt werden.

Um die Flexibilität dieses einfachen Nachrichtenmechanismus zu zeigen, soll eine komplexere Interaktionsform als Beispiel aufgebaut werden. In Abbildung 5.5 ist eine synchrone n:1-Nachrichtenkommunikation zwischen n Sendern und einer Empfänger-Instanz mit n Kanälen gezeigt. Der Empfänger soll solange blockieren, bis auf mindestens einem der Kanäle eine Nachricht anliegt. Diese Semantik entspricht typischen *n–Client* : 1–*Server* Szenarien, die unter *Unix* über `select()` realisiert werden. In [Wett93] heißt diese Interaktionsform n-fach–*EINZEL-Kanal*. Die Realisierung erfolgt auf Basis von n `msg`-Kanälen, wobei für alle n Kanäle die empfangende Instanz auch das Ziel der Signalisierung ist, wenn sie beim Empfang blockiert. Sobald einer der n Sender eine Nachricht abschickt, erhält der Empfänger ein Signal und kann im folgenden Lesezyklus asynchron alle Kanäle auf das Vorhandensein einer Nachricht prüfen. Ist das geschehen, und alle Nachrichten sind verarbeitet, blockiert die empfangende Instanz wieder.

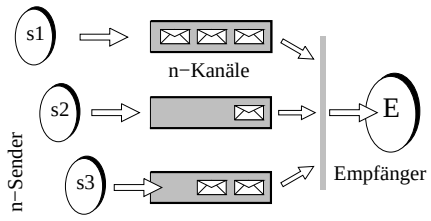


Abb.5.5 Mehrfachkommunikation

Die genannten Dienste der kern-internen Schicht *iproc's* bilden die Grundlage für die *kproc*-Instanzen, welche die eigentlichen Betriebssystemfunktionen des Kerns erbringen. Auf der mit *iproc's* geschaffenen Grundlage sollte es einfach möglich sein, *kproc*-Instanzen herzustellen und damit eine angepaßte Infrastruktur für übergeordnete *aproc*-Anwendungsinstanzen herzustellen.

Schicht $\mathcal{L}_3$ – *kproc's*

- *Zweck:*
  - Server-Instanzen im Kern zur Ausführung typischer Betriebssystemaufgaben,
  - *kproc's* sind individuell gestaltbar, so daß Kern-Infrastruktur anpaßbar wird,
  - eine Kategorie von *kproc's* ist auch dynamisch ladbar;
- *Charakteristik:*
  - zyklische (Server-) Instanzen als Infrastruktur für *aproc's*,
  - Interaktion erfolgt über Signale, gemeinsame Zustände, Nachrichten;

- ***Dienste***

Auf Basis der Dienste aus Schicht $\mathcal{L}_2$ können alle verbleibenden Kern-Funktionen als *kproc's* umgesetzt werden. Es gibt eine variable Anzahl von *kproc*–Server-Instanzen, welche in drei Gruppen eingeteilt werden können.

[**a**] *Existentielle Instanzen* müssen für den Betrieb des Systems unabdingbar vorhanden sein. Sie werden vom Bootprozeß erzeugt und existieren für die gesamte Laufzeit des Kerns.

[**b**] Sogenannte *notwendig optionale Instanzen* werden ebenfalls vom Bootprozeß erzeugt und existieren ebenfalls während der gesamten Laufzeit des Kerns. Diese Instanzen bestimmen im wesentlichen das Ausführungsmodell für die übergeordnete Schicht *aproc's*.

[**c**] *Optionale Instanzen* sind dagegen dynamisch in den Kern ladbar und können dynamisch gestartet und beendet werden. Sie bieten die Möglichkeit, zusätzliche Dienste während der Laufzeit in der Schicht *kproc's* zu installieren.

Instanzen der Kategorie [**a**] erfüllen lebenswichtige Funktionen des *CHEOPS*-Kerns: die Verwaltung des gesamten Kern-Speichers, Funktionen der wichtigsten Gerätetreiber (Terminal, Platte, Netz) und die Zeitüberwachung. Eine wichtige Instanz ist der Lader von Typbeschreibungen für das dynamische Einbinden neuartiger Instanzen. Dadurch können neue *kproc*-Instanzen der Kategorie [**c**] im Kern hergestellt werden.

Instanzen der Kategorie [**b**] stellen das Ausführungsmodell für die übergeordnete Schicht *aproc's* her. Vorn wurden zwei der für den *CHEOPS*-Kern möglichen Varianten *(4.a)* und *(4.b)* genannt.

Hierzu gehört eine Instanz für die Verwaltung und Steuerung der Anwendungsinstanzen, welche in ähnlicher Weise wie *ictrl* und $\mathcal{I}_{ksched}$ arbeitet, und es gehört die Verwaltung des Anwendungsspeichers dazu. Für Variante *(4.a)* muß diese Instanz getrennte Adreßräume für *aproc's* umsetzen. Variante *(4.b)* ist einfacher, da bei diesem Modell alle *aproc*-Instanzen im selben Adreßraum liegen. Virtueller Speicher ist für beide Fälle nicht vorgesehen, wäre aber prinzipiell durch abgewandelte Instanzen zur Verwaltung des Anwendungsspeichers herstellbar.

Es sollen hier nicht alle Dienste alle Server-Instanzen erklärt werden, sondern nur die wesentlichen Funktionen und Instanzen für die drei Kategorien genannt werden:

Art	*kproc*	Funktion
[a]	`kmm`	Verwaltung des Kern-Speichers und des Kern-Adreßraumes,
	`ktim`	Zeitüberwachung,
	`knewi`	Erzeugung neuer *kproc*-Instanzen,
	`ktyp`	Typ-Manager, Laden und Linken von Typbeschreibungen (`.o`-Files),
	`kcfs`	einfaches Filesystem für Typbeschreibungen,
	`kns`	Namensdienst für die Abbildung: Instanznamen → `kid`'s,
	`sash`	*stand-alone shell* für eine einfache Nutzerinteraktion,
	Gerätetreiber:	
	`ktty`, `kdisk`, `knet`, `kicc`[2],	
[b]	`anewi`	Erzeugung einer neuen *aproc*-Instanz,
	`asched`	Verwaltung und Steuerung von *aproc's*,
	`amm`	Verwaltung des Anwendungsspeichers,
	`asysc`	Empfängt und delegiert *aproc*-Systemrufe,
[c]	–	anwendungsspezifisch.

Abb.5.6 Übersicht über *kproc*-Instanzen des *CHEOPS*-Kerns

In Abbildung 5.7 sind die Benutzungsbeziehungen im groben zwischen *kproc*-Instanzen dargestellt. Eine Spezifik der *CADMUS*-Workstation ist der *ICC*[2] als Schnittstelle zum autonom arbeitenden E/A-System dieser Maschine.

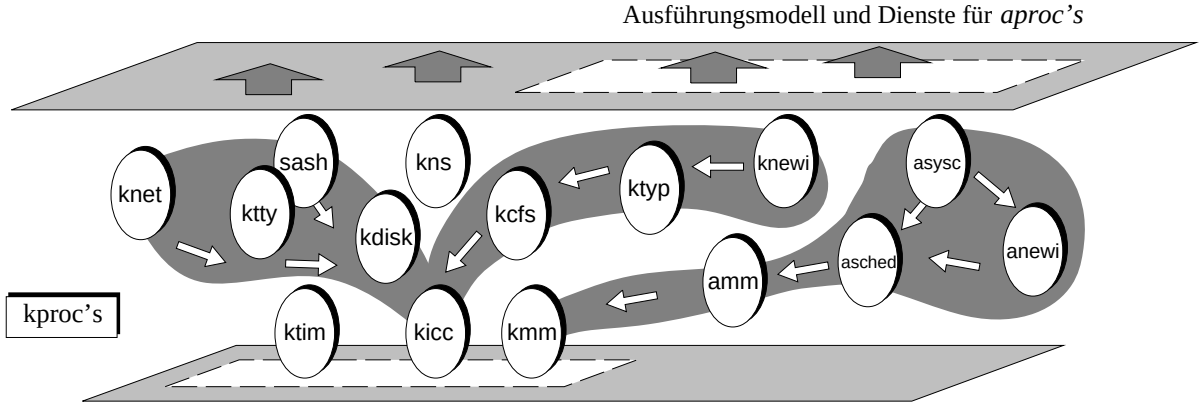


Abb.5.7 Benutzungsbeziehungen für Instanzen der Schicht *kproc's* im *CHEOPS*-Kern

[2] *Intelligent Communications Controller* – Anbindung des E/A-Subsystems der *CADMUS*-Workstation

- ***Identifikation*** – *Dienste, Dienstausführungen und Dienstleister*

Im Gegensatz zu den Schichten *ictrl* und *iproc's*, bei denen keine individuellen Dienstleister für Dienste der Infrastruktur in übergeordneten Schichten identifizierbar waren, sind in der Schicht *kproc's* Dienstleister als individuelle Instanzen nach außen bekannt. Es gibt folglich keine globale Menge von Diensten der Infrastruktur, sondern alle Infrastrukturdienste sind einzelnen dienstausführenden *kproc*-Instanzen zugeordnet und werden auch als solche von *aproc's* identifiziert. *kproc*-Instanzen sind als zyklische (→ *Server*-) Instanzen ausgebildet, die auf das Eintreffen von Nachrichten oder Signalen warten und daraufhin Operationen und damit Dienste ausführen.
Die Definition der Dienste erfolgt auf der Ebene der Beschreibung über Schnittstellendefinitionen von *C++*-Klassen (Beschreibungen für: *Dienste*, *Parameter* und *Rückgabewerte*), so daß entsprechender Code für die Dienstaufrufe generiert werden kann. Die Identifikation des Diensterbringers wird über einen `kid` hergestellt, der entweder a priori bekannt ist oder indirekt über Auflösungsdienste von Namen zur Laufzeit ermittelt wird. Das ist die Aufgabe der Instanz `kns`, deren `kid` im voraus bekannt sein muß. Alle *kproc* -Instanzen sind in `kns` als Tupel (name, `kid`) mit ggf. zusätzlichen Attributen vermerkt. Dynamisch erzeugte Instanzen werden zur Laufzeit in `kns` eingetragen. Über den Namensdienst kann auch die Sichtbarkeit und damit die Benutzbarkeit von Dienstleistern gesteuert werden. Das Merkmal der Instanzbereiche wird für die Kategorie der Infrastrukturdienste an der selektiven Sichtbarkeit von Namen und damit der Verfügbarkeit von Dienstleistern festgemacht. Schutz, wie mit *capabilities* von *Amoeba* [MRT+90], ist damit aber nicht verbunden, sondern wäre eine speziell herzustellende Eigenschaft.

- ***Interaktion*** – *Protokolle und Schnittstellen*

Für die Interaktion müssen zwei Fälle unterschieden werden. Zum einen können *kproc*-Instanzen Dienste anderer *kproc*-Instanzen in der gleichen Schicht nutzen, zum anderen sind Dienste von *kproc*-Instanzen auch Infrastrukturdienste für übergeordnete *aproc's*.

Die Beziehung $\boxed{kproc's \rightleftharpoons kproc's}$ wird über Nachrichten, Signale und gemeinsamen Speicher umgesetzt. Je nach Auslegung und der Art des Dienstleisters bestimmt dieser die Interaktionsform, welche in der darunterliegenden Schicht *iproc's* implementiert ist und vorn erklärt wurde.

Die Beziehung $\boxed{aproc's \rightleftharpoons kproc's}$ ist komplexerer Natur. Anwendungsinstanzen können nicht einfach Nachrichten im Kern initiieren, um damit Dienstausführungen in der Schicht *kproc's* zu veranlassen. Sie laufen im *User-Mode* des Prozessors ab, und der Kern-Bereich des *CHEOPS*-Kerns ist geschützt. Für die Interaktion zwischen Anwendungs- und Kern-Instanzen bleibt damit nur der traditionelle Weg, Kern-Dienste über `traps` anzufordern. Für die hier relevanten Zielplattformen besteht hierin die einzige Möglichkeit, den Übergang vom Anwendungs- in den Kern-Bereich und vom *User-* zum *Kernel-Mode* des Prozessors zu veranlassen. Der Prozessor führt einen Interruptzyklus aus, und es wird ein *iproc* erzeugt. Jeder Dienstaufruf einer Anwendungsinstanz führt damit auch zum Entzug des Prozessors für die Dauer der Dienstausführung. Der erzeugte *iproc* kann, wie jeder andere *iproc*, Nachrichten an eine *kproc*-Instanz senden und damit die Ausführung des Dienstes veranlassen. Der veranlassende *aproc* wird vorher noch als wartend in der Verwaltungsdatenstruktur der *aproc's* (→ `acbs`) markiert. Anschließend wird der *iproc* beendet. Die dienstausführende *kproc*-Instanz muß dann sicherstellen, daß der wartende *aproc* nach der Dienstausführung wieder aktiviert wird und bei einem späteren Wiederanlauf die bei `trap` gerettete Zustandsmenge mit den Rückgabewerten der Dienstausführung wiederhergestellt wird. Daraufhin läuft die *aproc*-Instanz nach der Dienstausführung wieder an. Es entsteht ein synchrones Interaktionsverhalten zwischen *aproc*- und *kproc*-Instanzen.
Diese Interaktionsform bezieht sich auf "kleine" Nachrichtenmengen zum Aufruf von Diensten. Durch `trap` können nur wenige Parameter in Registern mitgegeben werden, welche dann vom *iproc* ausgewertet und an den *kproc* weitergeleitet werden. Größere Datentransfers erfolgen durch die dienstausführende Instanz mit entsprechenden Speicherkopieroperationen.

5.2.3 Ausführungseigenschaften – Ausführungsmodelle

Als zweite Facette der Ausführungseigenschaften wurden die Ausführungsmodelle zum Teil schon bei der Erläuterung der Infrastrukturdienste für *ictrl*, *iproc's* und *kproc's* mit erklärt. Die wesentlichen Eigenschaften sollen deshalb hier nur kurz noch einmal in einer Tabelle zusammengefaßt werden.

	ictrl	*iproc's*	*kproc's*	*aproc's (a)*	*aproc's (b)*
Installation (Typ)	Bootprozeß	Bootprozeß	Bootprozeß, dynamisch	Bootprozeß, dynamisch	Bootprozeß, dynamisch
Erzeugung	Bootprozeß einmalig	in *ictrl* bei Interrupts	Bootprozeß, dynamisch	Bootprozeß, dynamisch	Bootprozeß, dynamisch
Lebensdauer	permanent	temporär	permanent, temporär	permanent, temporär	permanent, temporär
örtliche Bindung	fest	fest	fest	fest ggf. migrierbar	fest ggf. migrierbar
Identifikation:					
- Instanzen	implizit: *ictrl*	implizit: *iproc's*	`kid`: – *kproc's*	`aid`: – *aproc's*	`aid`: – *aproc's*
- globale Daten	`lock, irv`	`kcbs`	`kmsg`	–	–
- Dienste	Interrupts IR +iE/DI,itrap,lock	IR-Behandlung + *kproc's*, Signale	Kern-Dienste *aproc's*, Nachrichten	Anwendung	Anwendung
Quantifizierung	genau 1	*n*	*n*	*n*	*n*
Prozeßmodell	1, nichtpreemptiv	*n*, preemptiv	*n*, preemptiv	*n*, preemptiv	*n*, preemptiv
Wirkungsbereich	nur in *ictrl*	nur in *iproc's*	*iproc's*, *kproc's*	nur in *aproc's*	nur in *aproc's*
Schutz	Kern-Adreßraum (AR) *page protection* des Kern-Adreßraum für Zugriffe im *Kernel Mode* des Prozessors			separate AR *User Mode* des Prozessors	ein gem. AR
Interaktion	IR	Signale, Nachrichten, gem. Daten	Signale, Nachrichten, gem. Daten	`trap`	`trap`

Abb.5.8 Ausführungseigenschaften in den Schichten des *CHEOPS*-Kerns

5.2.4 Zusammenfassung zu Spezialisierung

Die für den *CHEOPS*-Kern spezialisierte Architektur ist an die Anforderungen dieses Kerns und an die der Zielplattformen angepaßt. Sie ist nicht mehr abstrakt, wie die generalisierte Architektur, sondern fixiert spezielle, an den Zweck angepaßte Eigenschaften. Es gibt vier Schichten, wobei in den drei Kern-Schichten Instanzen das dominierende, aber nicht das alleinige Strukturmerkmal sind. Globalzustände in Form der Monitore gehören auch dazu. Damit wird das Instanz-Prinzip aber nicht aufgehoben, sondern eine an die Eigenschaften der Zielsysteme angepaßte Form der Interaktion hergestellt. Auch die Ausführungsmodelle orientieren sich an den Zielsystemen mit je einem Prozessor für den *CHEOPS*-Kern[3] und einem traditionellen Interruptsystem, so daß zum Beispiel `iEI/iDI` für die Exklusivität der Monitoroperationen ausreicht. Die zwei Prozessormodi: *User/Kernel* und der über *page protection* geschützte Kern-Adreßraum führen dazu, daß die Interaktion zwischen Anwendungs- und Kern-Instanzen über `trap` und damit über *ictrl* und *iproc's* erfolgen muß. Diese Beispiele zeigen, wie sich Eigenschaften der Zielmaschinen in einer konkreten Architektur widerspiegeln. Die weiteren Ausführungen zur Implementierung des *CHEOPS*-Kerns beziehen sich ausschließlich auf die *CADMUS*-Plattform.

[3] Der zweite Prozessor der *CADMUS*-Workstation übernimmt E/A-Aufgaben. Auf ihm läuft ein eigener Kern.

5.3 *Iproc's* – Ein instanzbasierter Ansatz zur Behandlung von Interrupts

> " Interrupts are an unpleasant fact of life. They should be hidden away, deep in the bowels of the operating system, so that as little of the system as possible knows about them."
> [Tan92], S. 212.

Diese Einschätzung von A.S. Tanenbaum aus seinem Buch *"Modern Operating Systems"* charakterisiert eine Grundeinstellung, welche heute dem Bereich der Interruptverarbeitung in Betriebssystemen entgegengebracht wird. Als Ursachen gelten die Asynchronität und Nichtberechenbarkeit des Eintretens von Unterbrechungen, die den sonst geordneten (determinierten) Ablauf stören und zu Wettlaufbedingungen und kritischen Abschnitten führen.
Ein anderer Grund kann aber auch darin gesehen werden, daß das "Phänomen" der Unterbrechungen nicht recht in manche Modelle von Betriebssystemen paßt und somit ein Restbereich im Kern übrigbleibt, welcher nicht den sonst geltenden Modellierungs- und Architekturprinzipien des Systems unterliegt. Dieser sollte dann konsequenterweise so klein wie möglich und von den anderen Komponenten des Betriebssystems weitgehend isoliert sein.
Man kann aber auch umgekehrt herangehen und versuchen, eine geeignete Modellbetrachtung zu finden, die den Umstand der Unterbrechungen integriert, so daß die gleichen Funktions- und Wirkungsprinzipien der anderen Komponenten des Systems auch für die Unterbrechungsverarbeitung anwendbar sind. Der oben genannte Restbereich im Betriebssystem könnte entfallen, wenn es keine prinzipiellen Unterschiede in Struktur und Wirkungsweise zwischen Komponenten innerhalb und außerhalb des Interruptsystems mehr gibt.

Für den *CHEOPS*-Kern wird das aus zwei Gründen angestrebt. Zum einen erfordert die Zielstellung einer gleichbleibenden, homogenen Grundstruktur in *allen* Schichten des Systems die Integration des Interruptsystems in das Strukturmuster der anderen Schichten. Zum anderen könnte damit gezeigt werden, daß die Interruptverarbeitung keinen besonderen, nicht in die hier angewandte Modellvorstellung zu Betriebssystemen einzuordnenden Merkmalen unterliegt. Für den *CHEOPS*-Kern bedeutet das konkret die Anwendung einer Instanzstruktur in der Schicht der Interruptverarbeitung und eine entsprechende Implementierung der Verarbeitung von Interrupts durch Instanzen. Es geht dabei nicht um eine neue Interpretation bekannter Vorgänge in einer Begriffswelt von Instanzen, sondern diese Art der Interruptverarbeitung erfordert auch eine besondere Gestaltung, welche von der bisherigen Interruptbehandlung abweicht. Sie wird im folgenden im Detail beschrieben.

Der *CHEOPS*-Kern bietet damit auch in dieser Hinsicht einen neuen Ansatz, durch eine geeignete *Modellierung* (Modellvorstellung, Interpretation) und *Gestaltung* (Umsetzung durch Spezialisierung der Architektur, Implementierung) der Unterbrechungsverarbeitung in

- zwei separaten Schichten: *ictrl* und *iproc's* und
- mit *Instanzen* als dominierenden Elementen in diesen Schichten[4]

eine Lücke in Betriebssystemen zu schließen, welche heute allgemein besteht. Es trifft aber in der Tat zu, daß die Interruptverarbeitung anderen Ausführungsmerkmalen unterliegt, als sie für die übrigen Kern-Instanzen (→ *kproc's*) gelten. Diese Unterschiede begründen auch die Abgrenzung in Schichten, welche jedoch alle dem gemeinsamen Grundmuster genügen. Die Integration der Unterbrechungsverarbeitung in die hier vorgeschlagene Architektur bzw. umgekehrt die Anpassung der Architektur für diesen Bereich des Betriebssystems war ein wichtiges Anliegen des *CHEOPS*-Projekts und bildet den Schwerpunkt des praktischen Teils dieser Dissertation.

[4] Neben Instanzen gibt es auch globale Zustände für Interaktionszwecke (vgl. `kcbs` für Signalisierung und `kmsg` für Nachrichten in Abschnitt 5.2.2).

5.3.1 Das Interruptsystem

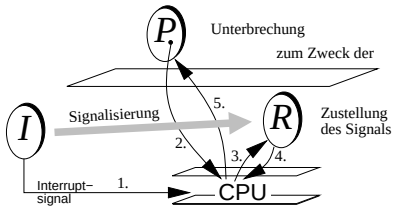
Der technische Vorgang bei Unterbrechungen ist im Prinzip seit Jahrzehnten gleich. Prozessoren unterscheiden sich nur in Nuancen bei der Unterbrechungsverarbeitung. Man trifft aber bei der *Einordnung* von Unterbrechungen in Betriebssystemkonzepte auf unterschiedliche Sichtweisen, wie es sie auch für andere Konzepte in Betriebssystemen gibt, z.B. für das der Prozesse. Für die Einordnung der Sichtweisen zu Unterbrechungen kann man grob zwei Kategorien identifizieren:

- eine ***prozessororientierte*** Sichtweise, bei der Interrupts asynchrone Signale an den Prozessor sind und die daraufhin ausgeführten Interruptzyklen "nichtprogrammierte" Aufrufe von Unterbrechungsbehandlungsroutinen bewirken (vgl. Lehrbücher: [Wett93], S. 21, [Dav92], S. 100 ff., [Dei90], S. 57 ff., [Kra88], S. 34 ff., [Sil91], S. 30 ff., [TB93], S. 14 ff.) und
- eine ***prozeßorientierte*** Sichtweise, welche versucht, die *Wirkung* von Unterbrechungen in den Kontext des umgebenden Betriebssystems einzuordnen, daß "Interrupts" letztendlich Mittel zur *Signalisierung zwischen Prozessen*[5] sind, wobei für den Empfang des Signals einem anderen aktiven Prozeß der Prozessor kurzzeitig entzogen und damit dieser Prozeß unterbrochen werden muß[6] (vgl. Lehrbuch: [Kal90], S. 43, S. 221 ff.).

Bei der ersten Betrachtungsweise wird der Vorgang im Prozessor bei einem Interruptzyklus weitgehend isoliert beschrieben. Diese Betrachtung findet man in der Regel für Systeme ohne Prozeßstruktur im Kern (vgl. [GC94], S. 216, [Schim94], S. 4, [Rob96], S. 8). Für Multiprozessorsysteme sind Interrupts auch ein wichtiges Mittel zur *Interprozessor-Kommunikation* [JPN84].

Bei der zweiten Kategorie liegt der Schwerpunkt bei der Einordnung der Wirkung in den Kontext des umgebenden Betriebssystems. Der eigentliche Empfänger eines Signals ist nicht der Prozessor, sondern ein wartender Prozeß im Betriebssystem. Nur die Übermittlung des Signals benötigt den gemeinsamen Prozessor, so daß diesem ein Interruptzyklus signalisiert werden muß, um den laufenden Prozeß kurzzeitig zu unterbrechen, damit das eigentliche Signal zugestellt werden kann. Unterbrechungsverarbeitung ist damit eine spezielle Ausprägung der *Signalisierung* zwischen Prozessen ($\rightarrow$ *umlenkend* [Wett93], vgl. auch Abschnitt 3.3.7).

Durch folgendes Schema können die Zusammenhänge verallgemeinert werden und als Vorgabe für die Modellierung und für die Implementierung dienen.



- Es gibt drei beteiligte Prozesse:
  - einen *Initiator* der Signalisierung – I,
  - einen *Rezeptor* der Signalisierung – R,
  - einen unterbrochenen Prozeß – P.
- R wartet auf das Signal, P und R sind voneinander unabhängig, es besteht aber eine indirekte Kopplung über den gemeinsamen Prozessor.
- Trifft das Signal ein, wird P unterbrochen und R aktiviert. Gäbe es keinen Unterbrechungsmechanismus, müßten P oder R (dann mit Umschaltungen $P \rightleftharpoons R$) zyklisch das Eintreffen des Signals prüfen und entsprechend verzweigen ($\rightarrow$ *versuchend* [Wett93] oder *polling*). Durch den Interruptzyklus entsteht nur ein minimaler zeitlicher Verzug für die Reaktion auf ein Signal, d.h., es gibt eine definierte Reaktionszeit für den Beginn der Interruptbehandlung, und es ist keine Rechenleistung für die Prüfzyklen nötig. Die Prüfzyklen entfallen aber nicht, sie werden eine Ebene tiefer im Prozessor nach jeder Instruktion ausgeführt.

[5] Ein *"nicht-CPU"*–Prozeß kann auf einem Geräte-Controller ablaufen und einem *"CPU–Gerätetreiberprozeß"* ein Ereignis signalisieren, es können aber auch zwei *"CPU-Prozesse"* über das Unterbrechungssystem Signale schicken. (Interrupt-) Signale können *nur von Prozessen* initiiert bzw. empfangen werden.

[6] Als Wirkung der Signalisierung kann dem unterbrochenen Prozeß der Prozessor auch ganz entzogen werden.

In Rechensystemen wird das Interruptsystem für eine Vielzahl weiterer Aufgaben benutzt:

- zur Reaktion auf synchrone und asynchrone Ausnahmen:
  - der Hardware (*power fail, machine exception, parity error* u.a.),
  - beim Ablauf eines Prozesses (*division by zero, page fault, illegal instruction* u.a.),
  - "programmierte" Ausnahmen für *Debugging* oder *Tracing*;
- für den Entzug des Prozessors bei konkurrierenden Prozeßsystemen, denn das Betriebssystem kann die Ablaufsteuerung nur durch den Entzug des Prozessors über das Interruptsystem umsetzen; das Interruptsystem ist der letztendliche Prozeßumschalter [Kal90];
- für Systemrufe – der einzigen Möglichkeit, gesichert Dienste des Kerns zu nutzen ($P{=}I$);
- zur Umsetzung virtuellen Speichers, von Interprozeß- und Interprozessor-Interaktion u.a.

All diese Varianten lassen sich sowohl in eine prozessor- als auch in eine prozeßorientierte Betrachtung einordnen. In der Literatur wird zwischen *synchronen* und *asynchronen* Unterbrechungen unterschieden. In der prozeßorientierten Sichtweise erklärt sich dieser Unterschied daraus, ob Initiator- und Rezeptorprozeß auf demselben Prozessor ablaufen oder auf verschiedenen asynchron zueinander arbeitenden Prozessoren. Der Begriff des Prozessors wurde in Kapitel 3 als selbständiger Aktivitätsträger für eine unabhängige Steuerung in einer Ebene eingeführt. Jedes Element, welches autonom in einer Steuerungsebene Zustandsübergänge bewirken kann, ist in diesem Sinne ein Prozessor relativ zu dieser Ebene. Selbständig arbeitende Controller, der DMA-Baustein oder der Taktgeber sind neben den "CPU-Prozessoren" ebenfalls aktive Prozessoren in dieser Ebene, auf denen Verarbeitungsschritte und damit Prozesse ablaufen, die ggf. über das Interruptsystem mit "CPU-Prozessen" zu synchronisieren sind.

Es ist klar, daß die allgemeinere Vorstellung die von Prozessen ist, denn zwischen dem empfangenden und dem unterbrochenen Prozeß besteht keine semantische Beziehung, außer die indirekte Kopplung über das gemeinsame Betriebsmittel Prozessor. Sonst sind die Abläufe unabhängig, was die Betrachtung als separate Prozesse rechtfertigt. In der Konsequenz heißt das aber auch, daß bei jedem Interruptzyklus keine Aufrufe von Behandlungsroutinen erfolgen, sondern mindestens zwei ***Prozeßwechsel*** $P \rightleftharpoons R$ stattfinden (*"context switches"* bezüglich des Prozessors [Kra88]). Die Behandlungs*routine* ist der Programmcode für den Behandlungs*prozeß*, der bei gleichen, zeitlich verschachtelten Unterbrechungsbehandlungen auch mehrfach ausgeführt werden kann. Es gibt in dieser Betrachtung keine "nichtprogrammierten" Aufrufe von Unterbrechungsbehandlungsroutinen, sondern Entzug und Zuteilung des Prozessors zwischen Prozessen, welcher durch Unterbrechungssignale ausgelöst wird. Nur an wenigen Stellen in der Literatur wird das klar herausgestellt [Kal90]. Bei einem Interruptzyklus muß, wie auch bei anderen leichtgewichtigen Prozeßumschaltungen, die Zustandsmenge des unterbrochenen Prozesses im gemeinsamen Prozessor gerettet werden, welche durch den Interruptprozeß überschrieben wird.

Die Ausführung einer Unterbrechungsbehandlungsroutine ist damit ein *separater Prozeß*. Die beiden Betrachtungsweisen unterscheiden sich folglich dadurch, ob dem unterbrochenen Prozeß durch den gemeinsamen Prozessor eine semantische Beziehung zum Behandlungsprozeß zugesprochen wird, so daß der "nichtprogrammierte" Ablauf der Behandlungsroutine als eingeschobene Fortsetzung "in dessen Kontext" angesehen werden kann. Der Begriff Behandlungs*routine* stützt dabei die Vorstellung des Ablaufs einer Prozedur. Auch bei der Programmierung wird eine Prozedur implementiert, deren Ausführung an einer definierten Stelle beginnt und die am Ende zurückkehrt. Diese Merkmale sprechen durchaus für eine prozedurale Ablaufform.
Der einzige Unterschied zu Prozeduren besteht darin, daß die "Aufrufe", außer bei "programmierten" Ausnahmen (z.B. `trap`'s), nicht programmiert und damit nicht in Programmen sichtbar sind und asynchron erfolgen können. Aber genau darin liegt die Ursache für Wettlaufbedingungen und kritische Abschnitte, und es ist schwierig, alle Stellen für Koordinierungen in Programmen zu identifizieren, weil es diese Probleme bei prozeduralen Abläufen sonst natürlich nicht gibt.

Es war eine wichtige Erkenntnis von Dijkstra Ende der sechziger Jahre [Dij68a, Dij71], zeitgleich existierende, aber semantisch nicht zusammengehörige Verarbeitungsfolgen in getrennte sequentielle ***Prozesse*** aufzuspalten, auch wenn sie auf einem gemeinsamen Prozessor ablaufen.
Was für Anwendungsprozesse seit langem Praxis ist und sich mittlerweile auch in modernen Betriebssystemen in Form einer Prozeßstruktur im Kern (→ *multi-threaded kernels*[7]) durchgesetzt hat, gilt bis auf wenige Ausnahmen noch nicht für den "Restbereich" der Unterbrechungsverarbeitung. Der Hauptgrund dafür ist die fehlende Unterstützung durch Prozessoren[8].

Die Vorteile einer prozeßorientierten Sichtweise liegen auf der Hand. Wettlaufbedingungen und kritische Abschnitte lassen sich auf natürliche Weise einordnen und sind keine abnormen Seiteneffekte mehr. Auf konzeptioneller Ebene muß keine Unterscheidung mehr zwischen Signalisierungsarten in verschiedenen Schichten des Systems getroffen werden. Der Mechanismus der *Unix*-Signale hat beispielsweise *im Prinzip* eine gleiche Wirkung und Funktion wie die Behandlung von Interrupts im Kern. In beiden Fällen besteht das Grundprinzip in der Signalisierung eines Prozesses. Die Unterschiede liegen in den Ausführungseigenschaften, welche aber bei der *CHEOPS*-Architektur über das Mittel der Schichten explizit eingeordnet werden können. Es wird somit keine generelle Gleichsetzung vorgenommen, sondern das *Grundprinzip* wird von seinen konkreten *Ausführungen* in bestimmten Schichten verallgemeinert. Diese generelle Vorgehensweise in dieser Dissertation läßt sich auch für die Gestaltung des Interruptsystems anwenden.

In der Literatur kann man in letzter Zeit ebenfalls eine Wiederbelebung der altbekannten Problematik von Unterbrechungen beobachten. Dabei geht es vor allem auch um die Auseinandersetzung der beiden Grundauffassungen. Die Tendenz geht dabei klar in Richtung der prozeßorientierten Betrachtungsweise. In *"Structured Interrupts"* [Hil93] wird festgestellt:

> " Even though the concept of *interrupts* has been with us for quite some time, I believe that the theoretical understanding of interrupts is still weak. There are clues to this weakness. For instance, in most operating systems it is necessary to construct interrupt handlers outside the normal process structure of a computer system." [Hil93], S. 51.

Mit dem "schwachen theoretischen Verständnis" ist nicht der technische Vorgang bei Unterbrechungen gemeint, sondern genau die Einordnung der Unterbrechungsverarbeitung in allgemeine Betriebssystemkonzepte, wie eben hier das der Prozesse.

In *"Interrupts as Threads"* [KE95] wird von Entwicklern des *Solaris 2* Betriebssystems von *SunSoft* die Integration der Interruptverarbeitung in eine spezielle Klasse von Kern-Threads gezeigt.

> " Most operating system implementations contain two fundamental forms of asynchrony: processes (or equivalently, internal threads) and interrupts. ...
> In the Solaris 2 implementation ..., these two forms are unified into a single model, threads. Interrupts are converted into threads using a low overhead technique. This allows a single synchronization model to be used throughout the kernel. In addition, it lowers the number of times in which interrupts are locked out, it removes the overhead of masking interrupts, and allows modular code to be oblivious to the interrupt level it is called at." [KE95], S. 21.

Auch in *"Prozeßsignalisierungen statt Unterbrechungen"* [GK92] wird das Wesen von Unterbrechungen herausgearbeitet und ein Vorschlag für eine prozeßorientierte Gestaltung des Interruptsystems von Prozessoren abgeleitet. Der Ansatz ist aber (leider) akademischer Natur, denn man trifft bei den Konstruktionsprinzipien des Unterbrechungssystems von Prozessoren auf

[7] *Unix* hatte diese Eigenschaft nicht, erst das breite Aufkommen von Multiprozessormaschinen führte Ende der achtziger Jahre zu dieser grundlegenden Restrukturierung des *Unix*-Kerns [Schim94, GC94]. Auch Echtzeitsysteme erfordern eine jederzeitige Unterbrechbarkeit des Kerns, welches ohne Prozeßstruktur schwierig ist.
[8] Für Schutzkonzepte und virtuellen Speicher wurde Prozessoren aber auch Unterstützung hinzugefügt.

den bekannten Sachverhalt, daß die Gestaltung ausschließlich von der zugrundeliegenden (hier Hardware-) Infrastruktur bestimmt wird (→ *bottom-up*). Eine effiziente softwareseitige Gestaltung der Komponenten zur Unterbrechungsverarbeitung im Betriebssystem muß sich an den Eigenschaften der Prozessoren für die Behandlung von Interrupts orientieren, woraus sich die heute angewandte Ablauf- und Verarbeitungsform ergibt. Andererseits werden daraus wieder die Anforderungen an neue Prozessorgenerationen abgeleitet, so daß durch diese wechselseitige Beziehung herkömmliche Strukturen und Wirkprinzipien über lange Zeit festgeschrieben werden. Der Wert der Forschungsarbeit [GK92] liegt deshalb vor allem darin, einmal aus Betriebssystemsicht Anforderungen an eine geeignete Prozessor-Infrastruktur zu formulieren (→ *top-down*).
Die Anforderungen an heutige Universalprozessoren definieren sich natürlich nicht allein aus Betriebssystemsicht. Leistungsgewinn wird heute vor allem mit steigender Arbeitsgeschwindigkeit und umfangreicheren Ressourcen bei gleichbleibender, kompatibler Technologie erzielt. Neue Gestaltungsprinzipien, auch aus der Betriebssystemforschung, fließen nur langsam darin ein.

Dennoch, auch für die Gestaltung von Prozessoren wird in Zukunft ein verstärktes *top-down* Vorgehen, eine stärkere Orientierung an Anwendungen und eine daraus folgende Spezialisierung ein wichtiger Trend sein. Er ist im Teilbereich anwendungsspezifischer Spezial- und Kundenschaltkreise auch schon einige Zeit Realität. Man kann sich durchaus auch eine Art spezialisierte "Betriebssystem-Prozessoren" vorstellen, die auf die Belange in Betriebssystemen zugeschnitten sind. Multiprozessorsysteme bestehen heute aus n gleichartigen, symmetrischen CPU's. Man könnte von einer "horizontalen" Anordnung gleicher Prozessoren sprechen. Spezialprozessoren findet man bei der Peripherie (Netz- und Grafikkarten u.a.), nicht aber für die ebenfalls effizienzbestimmenden Abläufe im Betriebssystem. Zur Unterstützung der Unterbrechungsverarbeitung wurde in [GK92] ein Vorschlag unterbreitet. Das wichtigste Merkmal eines auf Betriebssysteme spezialisierten Prozessors wäre die Unterstützung von Prozessen, nicht allein von Prozeduren. Man kann sich dafür schnelle atomare Listenoperationen oder hardwareseitig realisierte leichtgewichtige Synchronisationsmittel (z.B. *Signale, Semaphore*) vorstellen. Gleitkomma-Arithmetik könnte im Gegenzug entfallen. Zur horizontalen Anordnung von "Anwendungs-Prozessoren" käme eine vertikale Zuordnung spezialisierter Prozessoren für Infrastrukturschichten hinzu. Schritte zur Realisierung dieser Vorstellung sind aber für heutige Prozessoren noch nicht erkennbar, der Trend bleibt bei immer schnelleren Universalprozessoren.
Eingangs wurde aber derselbe Sachverhalt für die Beziehung zwischen Betriebssystem und Anwendungssystemen konstatiert, woraus sich ein Beweggrund allgemein in der Forschung und auch für das *CHEOPS*-Projekt für mehr Anwendungsanpaßbarkeit in Betriebssystemen ableitet.

Die Realisierung des *CHEOPS*-Kerns stützt sich natürlich auf real vorhandene Prozessoren ab, es wird allerdings die Softwareumgebung der Unterbrechungsverarbeitung in der untersten Schicht *ictrl* derart modifiziert, daß eine prozeßorientierte Behandlung von Interrupts durch eine spezielle Gattung von Instanzen (→ *iproc's*) entsteht. Durch die Schicht *ictrl* muß die Lücke zwischen dem Interruptsystem des Prozessors und einer prozeßorientierten Behandlung von Unterbrechungen durch *iproc's* geschlossen werden. Daraus resultieren einige vorteilhafte Eigenschaften, die in Abschnitt 5.3.6 zusammengefaßt sind. Die fehlende Hardwareunterstützung hat aber auch einen gewissen Mehraufwand gegenüber der konventionellen Lösung zur Folge (vgl. Abschnitt 5.3.5).

Die Darstellung sollte zeigen, daß auch in anderen Arbeiten das Thema Unterbrechungsverarbeitung wieder in der Diskussion ist. Das markante Merkmal des *CHEOPS*-Kerns einer instanzbasierten Gestaltung der Unterbrechungsverarbeitung hat daher nicht nur für die Architektur zur Homogenisierung der strukturellen und ablaufbedingten Merkmale in allen Schichten – eines der wesentlichen Ziele des *CHEOPS*-Projekts – Bedeutung, sondern setzt auch ein prozeßorientiertes Konzept zur Behandlung von Unterbrechungen um, welches in anderen Ausführungen und mit anderen Zielen auch in anderen Arbeiten ein aktueller Gegenstand von Untersuchungen ist und in einigen kommerziellen Systemen bereits Anwendung findet, z.B. in *Solaris 2* [KE95].

5.3.2 Die Behandlung von Interrupts im *CHEOPS*-Kern

Unterbrechungsbehandlungen sollen prozeßorientiert, d.h. hier durch *iproc*-Instanzen ausgeführt werden. Prozessoren unterstützen dieses Prinzip nicht, so daß die dafür notwendige Infrastruktur als Software in einer separaten Schicht *ictrl* zu implementieren ist. Aus der Notwendigkeit einer Softwarelösung für die Schicht *ictrl* resultiert auch der Mehraufwand gegenüber einer direkten Behandlung von Interrupts während des Interruptzyklus des Prozessors. Könnte man Anforderungen an die Gestaltung eines spezialisierten Betriebssystemprozessors formulieren, so wären zur Unterstützung einer prozeßorientierten Behandlung von Unterbrechungen durch *iproc's* die Infrastrukturaufgaben der Softwareschicht *ictrl* durch diesen Prozessor zu implementieren. Damit könnten die Kosten für die jetzige Softwarelösung reduziert werden, und die Vorteile könnten eine Akzeptanzgrundlage für diese Technik schaffen.

Die getrennten Aufgaben von Infrastruktur (→ *ictrl*) und Instanzbereich (→ *iproc's*) werden für den *CHEOPS*-Kern in zwei Schichten geteilt. Die Abgrenzung in zwei Schichten ergibt sich aber nicht nur aus der Beziehung: Infrastruktur – Instanzbereich. Sie ist auch wegen unterschiedlicher Ablauf- und Ausführungseigenschaften gerechtfertigt, und sie widerspiegelt den Ablauf einer prozeßorientierten Unterbrechungsverarbeitung im Betriebssystem in zwei Phasen:

Schicht 1 / Phase 1 – *ictrl* – Diese Instanz (hier gleichzeitig Schicht) ist während des Interruptzyklus des Prozessors aktiv. Ihre Aufgabe ist das Retten der Registerwerte der unterbrochenen Instanz P in den Rettebereich von P. Anschließend wird eine *iproc*-Instanz erzeugt und anhand des Interrupttyps der Code der später als *iproc* auszuführenden Behandlungsroutine zugewiesen. Anschließend wird ein Re-Scheduling veranlaßt.
Ist ein *iproc* – ggf. auch ein anderer als der gerade erzeugte – ausgewählt, wird die Interruptbehandlung aus Sicht des Prozessors beendet (→ `rte`[9]). Es wird aber nicht zu P zurückgekehrt, sondern die vom Scheduler ausgewählte Instanz aktiviert. Dies kann auch wieder P sein, noch bevor der Interrupt behandelt wurde. Die Entscheidung trifft allein der Scheduler, sie ist nicht mehr von der fest im Prozessor eingebauten Strategie abhängig.

Schicht 2 / Phase 2 – *iproc's* – Mit der Ausführung der Behandlungsroutine als *iproc*-Instanz erfolgt die eigentliche Reaktion auf das durch den Interrupt angezeigte Signal.

Durch Interrupts vermittelte Signale werden folglich in zwei voneinander entkoppelten Phasen in zwei Schichten behandelt (→ *two level interrupt processing*). Während des Interruptzyklus des Prozessors wird in *ictrl* lediglich eine *iproc*-Instanz erzeugt (1. Phase). Danach wird der Interruptzyklus des Prozessors beendet und eine ausgewählte Instanz aktiviert (2. Phase). Dies ist in der Regel, aber nicht notwendigerweise, der erzeugte *iproc*. Durch die Entkopplung des Interruptzyklus des Prozessors von der prozeßorientierten Reaktion auf das über den Interrupt empfangene Signal werden auch Steuerungsebenen und damit Mittel zur Abstraktion für das Unterbrechungssystem eingeführt. Man findet dieses Prinzip auch in anderen Systemen in Form von *first-* und *second-level Behandlungen*. Die Operationen der *first-level* Behandlung werden während des Interruptzyklus des Prozessors ausgeführt. Diese Operationsfolge sollte kurz sein, um das Interruptsystem des Prozessors nicht zu lange zu belegen. Die *second-level* Behandlung erfolgt später an ausgezeichneten Ablaufpunkten (→ *preemption points*) im Betriebssystem.

In *"Needles and Links or Dealing with Interrupts"* [SSC94] wird ebenfalls ein Zwei-Phasen Konzept zur Behandlung von Interrupts vorgestellt (1. Phase – *needles*, 2. Phase *links*). Der Gegenstand ist hier aber eine Erweiterung des Coroutinenkonzepts von *Modula-2* für eine schnelle Reaktion auf Interrupts durch Coroutinen-Umschaltungen (→ `IOTRANSFER`). Die Neuerung ist, daß kein

[9] `rte` – `return from interrupt exception`, *m68020*-Instruktion zum Abschluß des Interruptzyklus der CPU

`IOTRANSFER` mehr zur behandelnden Coroutine stattfindet, sondern der Interrupt "wird auf dem Stack der aktiven Coroutine" behandelt (→ *needle*), und es werden mit einem *link* das Auftreten des Interrupts und einige Argumente in einer globalen Datenstruktur vermerkt, so daß die eigentliche Behandlungs-Coroutine nach dem Interruptzyklus durch `IOTRANSFER` aktiviert werden kann. Damit wird ebenfalls eine Entkopplung des Interruptzyklus von der Reaktion auf den Interrupt erreicht, so daß `IOTRANSFER` außerhalb des Interruptzyklus ausgeführt werden kann. Coroutinen sind aber für die Unterbrechungsverarbeitung nur der halbe Weg hin zu Prozessen.

Im *CHEOPS*-Kern kommt das Zwei-Phasen-Prinzip explizit durch zwei Schichten mit Instanzen als Verarbeitungselementen zum Ausdruck. Der Interruptzyklus des Prozessors ist damit auf ein notwendiges Minimum beschränkt: Zustand retten → *iproc* erzeugen → Re-Scheduling. Es sind keine interruptspezifischen Verarbeitungschritte notwendig, so daß die Reaktionszeit für einen Interrupt weitgehend nur von der Scheduling-Strategie in *ictrl* abhängt und damit durch das Betriebssystem gestaltbar ist. Instanzen der Schicht *iproc's* sind autonom und selbst "unterbrechbar", d.h., ihnen kann der Prozessor durch *ictrl* entzogen werden. Damit arbeiten *iproc*-Instanzen nach einem gleichen Grundprinzip wie Instanzen in anderen Schichten.

5.3.3 *iproc's* – eine alternative Form der Unterbrechungsbehandlung

Für den *CHEOPS*-Kern ist es folgerichtig, Unterbrechungen explizit in Form einer spezialisierten Art von Instanzen (→ *iproc's*) zu behandeln. Zunächst soll der Unterschied zur herkömmlichen Art der Interruptbehandlung in Betriebssystemen deutlich gemacht werden.
Wenn man Entwurfsprinzipien und Implementationen von Unterbrechungsbehandlungen in verschiedenen Betriebssystemen analysiert[10], kann man zwei Merkmale identifizieren, welche eine prozeßorientierte Behandlung von Unterbrechungen verhindern.

- Es wird ein ***gemeinsamer "Interruptstack"*** für verschiedene Aufgaben benutzt:
  - für das Ablegen von Informationen über das eingetroffene Interruptsignal (`irf` – *Interrupt Frame:* Rückkehradresse, ggf. Parameter, wie z.B. die *page fault address*),
  - für das Retten der Registerwerte der unterbrochenen Instanz, welche durch die Behandlungsprozedur ggf. überschrieben werden – `regs`,
  - für den Ablauf der Behandlungsprozedur selbst (lokale Variablen, Rückkehradresse, weitere Aufrufe von Prozeduren u.a.) – `sibr`.
- Ablaufende Unterbrechungsbehandlungen haben keine ***explizite Repräsentation***, etwa in Form der Verwaltungsdatenstrukturen für Kern-Prozesse. Sie haben damit auch keine Identität für Dienste zur Steuerung des Ablaufs durch den Kern. Die gesamte Ablaufsteuerung wird durch das im Prozessor fixierte Verfahren bestimmt. Für das Betriebssystem bleibt als Einflußmöglichkeit lediglich das Sperren der Annahme weiterer Interruptsignale.

Diese beiden Merkmale haben bei der Behandlung von Unterbrechungen einige problematische Ablaufeigenschaften zur Folge, wobei sich diese zwangsläufig ergeben und daher heute nicht als Probleme, sondern als die notwendige Art und Weise der Behandlung von Unterbrechungen in Betriebssystemen angesehen werden. Diese paßt jedoch vor allem in "sequentiellen" (*single-threaded*) Betriebssystemkernen nicht zum sonstigen Ablauf von Kern-Prozeduren, woraus sich die Probleme bei der Behandlung asynchroner Interrupts ergeben (vgl. Zitat [Tan92] auf Seite 123). Die Ursache ist aber nicht die Asynchronität an sich, sondern ein fehlender adäquater Umgang mit dieser Erscheinung. Dieser führt notwendigerweise zum Konzept von Prozessen, nicht nur, aber unter anderem auch im Bereich der Unterbrechungsverarbeitung in Betriebssystemen.

[10] Literaturverweise wurden zu Beginn des Abschnitts 5.3.1 auf Seite 124 angegeben.

Als Konsequenzen der beiden genannten Merkmale ergeben sich die typischen Ablaufeigenschaften bei der Behandlung von Unterbrechungen.

- Verschachtelter Interrupts müssen "prozedural" in der umgekehrten Reihenfolge ihres Eintreffens abgearbeitet werden. Die Abhängigkeit ergibt sich aus der Speicherungsfolge der "Kontexte" auf dem gemeinsamen Stack.
- Das Interruptsystem des Prozessors ist während der gesamten Ausführungszeit einer Behandlungsroutine belegt. Behandlungsroutinen sollten aus diesem Grund kurz sein. Weitere Interruptsignale sind entweder gesperrt und gehen ggf. verloren, oder aber bei höherpriorisierten Interrupts entsteht die bekannte prozedurähnliche Arbeitsfolge.
- Interrupts müssen in jedem Fall sofort behandelt werden, unabhängig von der Wichtigkeit der Signalisierung in Relation zum gerade ablaufenden Prozeß, z.B. auch die Behandlung eines anderen Interrupts. Als einzige Alternative bleibt das selektive Sperren von Interrupts nach Prioritätsebenen (Arbitrierung, Kaskadierung von Interrupts).

In Prozessoren ist dieses Verfahren seit langem fest eingebaut, so daß Betriebssystementwickler kaum die Chance haben, diesen Ablauf ohne Effizienzverlust zu verändern. Für den *CHEOPS*-Kern soll aber das Konzept der *iproc's* umgesetzt werden.

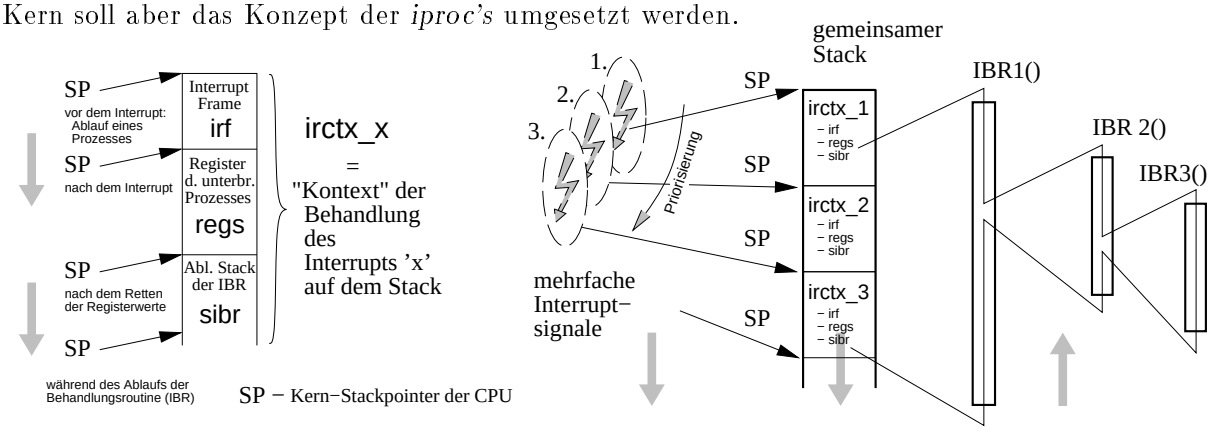


Abb.5.9 Heute gebräuchliche Art der Interruptbehandlung

In Abbildung 5.9 ist diese Art der Unterbrechungsbehandlung dargestellt. Kennzeichnend ist der gemeinsame Stack für alle Unterbrechungsbehandlungen. Bei manchen Prozessoren ist dies ein separater Interruptstack, aber in der Regel ist es der globale Kern-Stack, über den bei monolithischen Kernen auch sämtliche Kern-Routinen ausgeführt werden. Das Ablaufmuster ist damit notwendigerweise an Prozeduren angelehnt. Die Folge ist eine nicht änderbare Reihenfolge der Abarbeitung von Unterbrechungsbehandlungsroutinen anhand des Auftretens der zugehörigen Interruptsignale. Die *Reihenfolge des Eintreffens* von Interruptsignalen bestimmt *allein die Abarbeitungsreihenfolge*. Interrupts müssen auch in jedem Fall sofort behandelt werden, wenn man Interruptsperren ausnimmt. Eine andere softwareseitige, an den Strategien des Betriebssystems orientierte Einflußnahme auf die Verarbeitung von Interrupts gibt es nicht.

Nicht ohne Grund ist das Interruptsystem auch eine zentrale Komponente bei Systemen mit Echtzeitanforderungen. Bei diesen Systemen findet man daher die ausgeklügeltsten Interruptverarbeitungsstrategien [Hil92, Mau96, PSOS93, FGG+91]. All diese Systeme sind natürlich an die Fähigkeiten und Restriktionen der Prozessoren gebunden. Eine vollständig unter der Steuerung des Betriebssystems stehende und damit weitgehend anwendungsanpaßbare Strategie kann aber wegen der fehlenden Prozessorunterstützung aus Zeitgründen nicht eingesetzt werden.
Der *CHEOPS*-Kern ist nicht als Echtzeitsystem ausgelegt, hier stehen Fragen der Architektur und Untersuchungen zu alternativen Prinzipien und neuartigen Konzepten im Vordergrund.

Die genannten Einschränkungen für den Ablauf von Unterbrechungsbehandlungsroutinen in einer prozeduralen Form lassen sich durch eine prozeßorientierte Ablaufform aufheben. Der hauptsächliche Vorteil liegt in der Entkopplung der Arbeitsreihenfolge von der durch die stackorganisierten Speicherung der "Kontexte" bedingten prozedurähnlichen Form. Aus der angegebenen Analyse der herkömmlichen Unterbrechungsverarbeitung leiten sich einige Schlußfolgerungen ab, um auch auf verfügbaren Prozessoren einen prozeßorientierten Ablauf herzustellen:

- *Aufgabe des gemeinsamen Interruptstacks (!)*;
- *Trennung* der bei einer Unterbrechung anfallenden *Daten* (vgl. Abbildung 5.9):
  - `irf` – Informationen über den Interrupt (→ *Interrupt Frame*),
  - `regs` – Rettebereich für Zustände der unterbrochenen Instanz (→ *P*),
  - `sibr` – Stackbereich für den Ablauf der Behandlungsprozedur;

  daraus folgt eine *Entkopplung* semantisch nicht zusammengehöriger Daten, und es bietet sich damit auch die Möglichkeit zur *Entkopplung der Abläufe*;
- Herstellen *expliziter Repräsentationen* für ablaufende Unterbrechungsbehandlungen ähnlich den Verwaltungsdaten für Instanzen, damit das Betriebssystem über ihnen Dienste und damit eine Ablaufsteuerung umsetzen kann, insbesondere eine freie Scheduling-Strategie.

Als Randbedingung soll natürlich auch für den *CHEOPS*-Kern gelten, daß das vorgestellte Konzept möglichst effizient umgesetzt wird, aber nicht unter Aufgabe der Ziele.

Das Vorbild zur Entkopplung der bislang stark prozedural geprägten Unterbrechungsverarbeitung sind reguläre Kern-Instanzen. Es war auch ein Ziel des *CHEOPS*-Kerns, dieselben Grundprinzipien in allen Schichten des Systems wiederzuwenden, d.h. auch für die Unterbrechungsverarbeitung. Instanzen besitzen eigene abgegrenzte Verarbeitungszustände, was sie von "reinen" Prozessen unterscheidet, die ausschließlich in einem globalen Verarbeitungsraum operieren. Für *iproc*-Instanzen bedeutet dies, daß sie einen eigenen Stack und u.U. einen eigenen separaten Datenbereich besitzen. Ein privater Stack ist auch deshalb notwendig, weil die Behandlungsroutine nach wie vor als Prozedur abläuft[11], aber nicht als "eingefügte" Prozedur in den Kontext einer unbeteiligten Instanz, sondern als Prozedur innerhalb einer eigens für diesen Ablauf erzeugten *iproc*-Instanz. Die Behandlungsroutine ist lediglich der Programmcode dieser Instanz.

Zur expliziten Repräsentation von *iproc*-Instanzen müssen Datenstrukturen verwaltet werden. Operationen über diesen Datenstrukturen bewirken eine globale Steuerung des Ablaufs. Auch hierfür findet man das Vorbild beispielsweise in Form der *process control blocks – PCB* in anderen Betriebssystemen. Für *iproc's* des *CHEOPS*-Kerns wird es aufgegriffen und an die Bedingungen zur Unterbrechungsbehandlung angepaßt. Die Datenstruktur zur Repräsentation eines *iproc* heißt deshalb konsequenterweise auch *iproc control block*. Die Verwaltung erfolgt in der zu *iproc's* gehörenden Infrastruktur, d.h. in der Schicht *ictrl*. Jeder `icb` verfügt über eine Komponente `regs`, in welcher Werte aus Registern gespeichert werden können, wenn der zugehörige *iproc* selbst unterbrochen wird. In der Komponente `istk` ist ein Verweis auf den privaten Stack des *iproc* enthalten, und in `sinfo` kann Information für den *iproc*-Scheduler enthalten sein, z.B. eine Priorität. In `gs` wird der globale Zustand eines *iproc* vermerkt. Es sind zwei Zustände ausreichend: `CREAT` und `RUN`. Schließlich gibt es noch zwei Verweise für das Einhängen des `icb` in eine doppelt verkettete Ringliste arbeitsbereiter *iproc's*, über welcher der Scheduler in *ictrl* operiert. Im Bild ist das durch die beiden Verweispfeile angedeutet. Insgesamt läßt

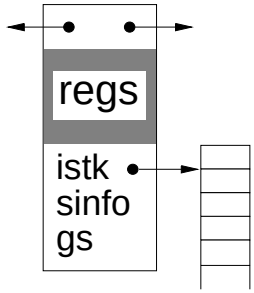


Abb.5.10 `icb` – *iproc control block*

[11] Der *C++*-Compiler kann keinen anderen Code als den für Prozeduren erzeugen.

sich eine prinzipielle Ähnlichkeit mit Repräsentationen anderer Kern-Instanzen oder mit *PCB* in anderen Systemen erkennen, hier allerdings in einer sehr elementaren, an die Bedingungen der Unterbrechungsverarbeitung angepaßten Form, was ganz im Sinne dieser Dissertation ist.
Als weitere notwendige Maßnahme für den prozeßorientierten Ablauf von Unterbrechungsbehandlungen muß das Retten der Zustände der unterbrochenen Instanz beim Entzug des Prozessors bei einem Interrupt dahingehend geändert werden, daß diese Werte nicht mehr wie bislang auf den Stack gerettet werden, sondern in den sowieso für jede Instanz vorhandenen Rettebereich in der dazugehörigen Verwaltungsdatenstruktur (!). Dafür ist ein Zeiger `*savregs` ausreichend, weiterer Mehraufwand entsteht nicht.

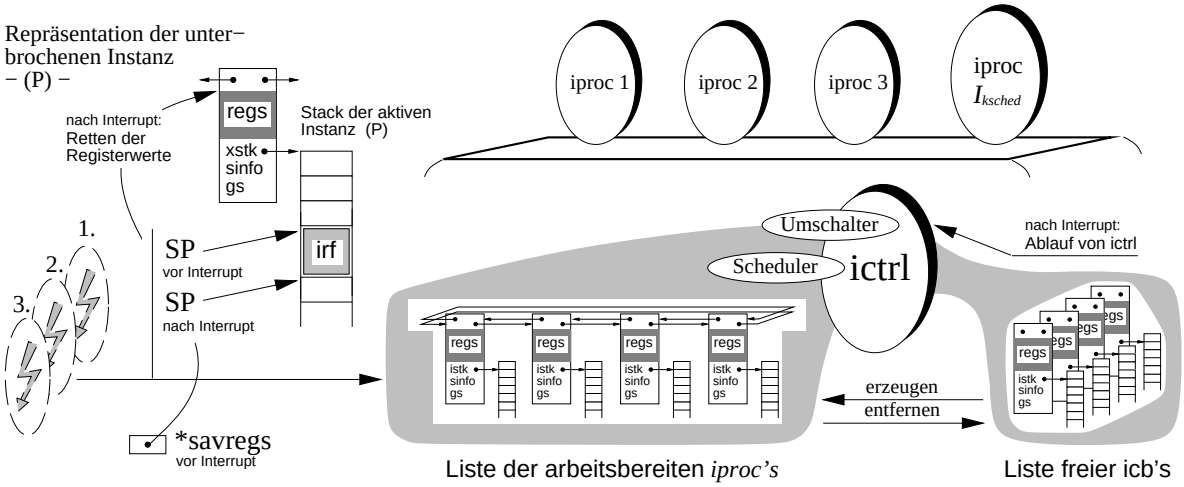


Abb.5.11 Unterbrechungsbehandlung mit *iproc's*

In Abbildung 5.11 ist der grundlegende Aufbau der Unterbrechungsverarbeitung des *CHEOPS*-Kerns mit der dafür vorgesehenen Software-Infrastrukturschicht *ictrl* gezeigt. Es wird der Unterschied zur herkömmlichen Technik in Abbildung 5.9 deutlich. Die zentrale Rolle spielen `icb`'s zur Repräsentation von *iproc's*. Die Steuerung des Ablaufs übernimmt der Scheduler in *ictrl*.

Der Ablauf in *ictrl* bei einem Interrupt kann in sechs Phasen [I]–[VI] eingeteilt werden:

[I] ***PREEMPT – P:*** weitere Interrupts für [I]–[III] in *ictrl* sperren (→ `iDI`) und den Zustand der unterbrochenen Instanz *P* nach *P.regs* retten (→ `*savregs`);

[II] ***CREATE:*** einen `icb` für einen neuen *iproc* anlegen:
- Interrupt anhand des *Interrupt-Frames* (→ `irf`) auf dem Stack von *P* analysieren,
- `icb` aus der Freiliste in die Bereit-Liste umketten,
- `icb` initialisieren: `regs`, `istk`, `sinfo`, `gs`;

[III] *Interrupt-Frame* (→ `irf`) derart manipulieren, daß nach `rte` nicht zu *P* zurückgekehrt, sondern bei ***SCHED*** fortgesetzt wird;
`rte` – Ende der Interruptbehandlung durch den Prozessor;

[IV] ***SCHED:*** der Scheduler in *ictrl* wählt eine Instanz aus der Bereit-Liste aus;

[V] Umschalten zu dieser Instanz (=Ende von *ictrl*), Interrupts wieder erlauben: `iEI`;

RUN: Ablauf der ausgewählten *iproc*-Instanz,
bei Ende: → Umschalten zu ***END*** in *ictrl*;

[VI] ***END:*** Fortsetzen nach dem Ende einer abgelaufenen *iproc*-Instanz, `iDI`,
Freigeben des `icb` durch Rückketten in die Freiliste, weiter bei ***SCHED***.

Der Prozessor arbeitet nur in [I]–[III] im Interruptmodus, am Ende von [III] steht `rte`. Vorher wird aber der *Interrupt-Frame* derart manipuliert, daß nicht zur unterbrochenen Instanz zurückgekehrt, sondern bei *SCHED* forgesetzt wird. Die Phasen [IV]–[VI] laufen außerhalb des Interruptmodus, so daß das Interruptsystem des Prozessors nur "minimal" ([I]–[III]) belegt ist.

Für den Ablauf wird vorausgesetzt, daß stets mindestens eine arbeitsbereite *iproc*-Instanz in der Bereit-Liste existiert. Dies ist durch die permanent existierende Instanz $\mathcal{I}_{ksched}$ gegeben, welche für das Scheduling von *kproc's* verantwortlich ist. Es hängt von der Scheduling-Strategie in *ictrl* ab, wann $\mathcal{I}_{ksched}$ aktiviert wird und damit übergeordnete *kproc*-Instanzen wieder zum Zuge kommen. Es ergibt sich eine hierarchische Abhängigkeit des Prozessor-Schedulings in Schichten, auf die in Abschnitt 5.4 noch näher eingegangen wird. In der Regel wird das Scheduling in *ictrl* so gewählt sein, daß zuerst alle *iproc's* ablaufen und $\mathcal{I}_{ksched}$ erst wieder an die Reihe kommt, wenn es keine anderen *iproc's* mehr gibt. Dieser Ablauf wird aber nicht erzwungen. Es besteht auch die Option, weniger wichtige Unterbrechungsbehandlungen erst verzögert ablaufen zu lassen und jederzeit zu unterbrechen.
iproc-Instanzen werden im Prinzip wie andere Kern-Instanzen repräsentiert und gesteuert, so daß sie im Prinzip auch gleiche Ablaufeigenschaften besitzen. Die Ablaufeigenschaften sind aber nicht mit denen von *kproc's* identisch, sondern unterscheiden sich in einigen Punkten, beispielsweise darin, daß *iproc's* nicht blockieren können. Es gibt keinen Globalzustand `BLOCKED`, *iproc*-Instanzen sind stets arbeitsbereit. Die Arbeitsreihenfolge legt der Scheduler in *ictrl* fest.

Wichtig ist auch, daß sich das geschilderte Prozessor-Scheduling bei Unterbrechungen tatsächlich nur auf das Betriebsmittel *Prozessor* bezieht, d.h. keine anderen Betriebsmittel, wie Adreßräume oder Speicher umgeschaltet werden. Diese hängen von den Ausführungseigenschaften in jeweiligen Schichten ab und müssen bei sogenannten *"horizontalen Umschaltungen"* (vgl. Abschnitt 5.4) von der Umschalt-Instanz in der jeweils dazugehörigen Infrastruktur umgesetzt werden. Die durch Interrupts veranlaßten *"vertikalen Umschaltungen"* des Prozessors über Schichten hinweg tangieren nicht die Globalzustände von Instanzen in übergeordneten Schichten. Einer dort ablaufenden Instanz wird bei einem Interrupt nur der Prozessor entzogen, alle anderen Betriebsmittel bleiben vorerst zugeteilt. Auch der Globalzustand dieser Instanz ändert sich nicht. Er gilt relativ in einer Schicht. Das bedeutet, daß eine unterbrochene Instanz auch bei *vertikalem Entzug* des Prozessors durch einen Interrupt *in ihrer Schicht (pseudo-) aktiv bleibt.*
Die Scheduler in den einzelnen Schichten erhalten die Steuerung entweder beim Blockieren oder bei Beendigung der von ihnen gesteuerten Instanzen, oder es wird über die Scheduler-Hierarchie die Steuerung nach einem Interrupt, ausgehend vom Scheduler in *ictrl*, wieder "nach oben gereicht". Jeder Interrupt bewirkt, daß unmittelbar *ictrl* aktiviert wird. Diese Instanz ist damit auch der am höchsten stehende Prozessor-Scheduler im *CHEOPS*-Kern. Er steuert die Scheduler der nächsten übergeordneten Schichten ($\rightarrow$ hier nur $\mathcal{I}_{ksched}$ für *kproc's*).

Um das Prinzip der *iproc's* trotz Mehraufwand effizient zu gestalten, werden einige Vorkehrungen getroffen bzw. ergeben sich einige Eigenschaften aus der Vorgehensweise:

- außer `CREAT` und `RUN` gibt es keine weiteren Globalzustände für *iproc's*, wodurch die Verwaltung vereinfacht wird; *iproc's* können aus diesem Grund nicht blockieren;
- es wird in der Regel wenige *iproc*-Instanzen geben, so daß der Scheduler in *ictrl* nur über einer kurzen Liste arbeitsbereiter *iproc's* operieren wird;
  oft wird es nur einen *iproc* und $\mathcal{I}_{ksched}$ geben, so daß dieser Sonderfall optimierbar ist;
- Verwaltungsdatenstrukturen werden im voraus vom Bootprozeß angelegt, so daß sich der Laufzeitaufwand beim Erzeugen und Löschen von `icb` auf das Umketten und Initialisieren reduziert; die Anzahl `icb` ist fixiert, es sind aber auch neue `icb` in die Freiliste einhängbar;
- vertikale Prozeßumschaltungen bei Interrupts betreffen nur die im Prozessor enthaltene Zustandsmenge ($\rightarrow$ *"processor context"*), andere Betriebsmittel werden nicht berührt.

Bislang wurde unterstellt, daß bei jedem Interrupt eine neue *iproc*-Instanz angelegt wird, indem ein `icb` aus der Freiliste in die Bereit-Liste in *ictrl* umgekettet und anschließend die vier `icb`-Einträge mit relevanten Werten für einen konkreten Interrupt initialisiert werden. Diese Form der ***pop up-Instanzen*** entspricht dabei eher dem Wesen von Unterbrechungsbehandlungen: das Interruptsignal bewirkt den Ablauf einer Unterbrechungsbehandlung. Dieser Ablauf ist der einzige Dienst einer *iproc*-Instanz. Die Alternative wären ***zyklische Instanzen***, je eine pro *Interrupttyp*, die permanent existieren und auf das Betriebsmittel Interruptsignal warten.

```
(a): Instanz erzeugen;
     {
         Behandlungscode;
     }
     Instanz loeschen;
```

```
(b): while(1) {
       wait_on_signal();

       Behandlungscode;

     }
```

Abb.5.12 Unterbrechungsbehandlungen: `(a)`–*pop up-Instanz*, `(b)`–*zyklische Instanz*

In der wenigen Literatur zu prozeßorientierter Behandlung von Interrupts findet man beides. In [Kal90, GK92, SSC94] wird die Variante mit einer festen Anzahl zyklischer Interrupt-Prozesse vorgeschlagen, die jeweils auf Interruptsignale warten.
In [KE95] (→ *Solaris*) bzw. auch in [Grau94d] (→ *CHEOPS*) wird die zuerst genannte Variante favorisiert, weil sie einige Vorteile gegenüber zyklischen Interruptbehandlungsprozessen aufweist.

- Es kann beliebige *iproc*-Instanzen desselben Interrupttyps zu einem Zeitpunkt geben. Eine zyklische Instanz kann immer nur einen Interrupt eines Typs bearbeiten. Weitere Interruptsignale gleichen Typs gehen verloren oder müssen zwischengespeichert werden, was ebenfalls zu einem nicht unerheblichen Mehraufwand führt.
- Die Anzahl im System vorhandener *iproc*-Instanzen ändert sich dynamisch und entspricht genau der zu einem Zeitpunkt tatsächlich in Behandlung befindlichen Unterbrechungen. Das werden in der Regel sehr viel weniger sein, als für jeden Interrupttyp (*typisch: 256*) je eine Instanz mit ihrer zugehörigen Repräsentation (→ `icb`) zu verwalten. Für *pop up-Instanzen* sind somit wesentlich weniger `icb`'s notwendig. Für den *CHEOPS*-Kern sind acht `icb` ausreichend.
- Für zyklische Instanzen muß ein dritter Globalzustand `BLOCKED` (*on interrupt signal*) eingeführt werden, welcher die Verwaltung verkompliziert, *iproc's* können nicht blockieren.
- Interruptbehandlungen haben vor und nach ihrem Ablauf keinen Steuerungszustand, so daß dieser auch nicht zu verwalten ist. Im Fall der zyklischen Instanzen ist das aber notwendig. Der "zustandslose" Start eines *iproc* ist billiger als der "zustandsbehaftete" Wiederanlauf einer zyklischen Instanz an der Stelle `wait_on_signal()`. Auch der Rücksprung am Ende von `while` ist unnötig.
- Es sind keine expliziten Operationen `wait_on_signal()` bzw. `send_signal()` notwendig, wobei `send_signal()` während des Interruptzyklus des Prozessors auszuführen wäre.

Insgesamt ergibt sich für *pop up-Instanzen* ein geringerer Speicher- und Laufzeitaufwand. Auch durch dieses spezifische, "leichtere" Ausführungsmodell unterscheiden sich *iproc's* von den vorwiegend zyklischen *kproc*-Instanzen. Darin spiegelt sich eine Anpassung an bzw. eine Ausnutzung von speziellen Ablaufeigenschaften bei Unterbrechungsbehandlungen mit *iproc's* wider.

Fazit: *iproc's* sind keine neue Interpretation des herkömmlichen Ablaufs von Interruptbehandlungsroutinen, sondern sie sind eine spezialisierte Form von Instanzen für die Unterbrechungsverarbeitung. Sie erfordern eine spezielle Gestaltung, ordnen sich aber homogen in die Architektur des *CHEOPS*-Kerns ein. Sie werden im Prinzip wie Instanzen anderer Schichten verwaltet und

bekommen dadurch im Prinzip gleiche Ablaufeigenschaften. Das unterscheidet sie von der traditionellen Art der Interruptbehandlung. Andererseits besitzen *iproc's* spezifische Eigenschaften, die sie von anderen Instanzen unterscheiden und eine Abgrenzung als Schicht erfordern.

iproc's besitzen damit alle notwendigen Eigenschaften von Instanzen (vgl. Abschnitt 4.2.2):

- *äußere Merkmale von Instanzen:*
  - *iproc's* führen jeweils genau einen Dienst zur Behandlung eines Interruptsignals aus;
  - sie sind wegen der Repräsentation durch `icb` im System identifizierbar;
  - sie besitzen einen Typ (Interrupttyp), welcher die Dienstausführung bestimmt;
  - Interaktion erfolgt über Schnittstellen und Protokolle mit *kproc*-Instanzen bzw. durch direkte Einflußnahme auf deren Repräsentation in $\mathcal{I}_{ksched}$, z.B. zum Reaktivieren einer auf ein Interruptsignal wartenden *kproc*-Instanz;
- *innere Merkmale von Instanzen:*
  - während der Zeit ihrer Existenz haben *iproc's* eigene innere Zustände der Verarbeitung (→ *Variablen auf dem lokalen Stack*) und der Steuerung (→ `gs`, `regs` im `icb`), wodurch *iproc's* voneinander entkoppelt werden und wie andere Instanzen ablaufen;
  - *iproc's* haben genau einen eigenen, unabhängigen inneren Prozeß mit:
    - eigener Aktivität – durch Scheduling und Ablaufsteuerung in *ictrl* transformiert,
    - eigener Steuerungsinformation – Programmcode der Behandlungsroutine.

Damit sind die wesentlichen Unterschiede und Neuerungen zur Behandlung von Unterbrechungen mit *iproc's* in ihren Grundzügen und weitgehend unabhängig von einer konkreten Implementierungsplattform vorgestellt. Im folgenden Abschnitt wird die Implementation der Infrastrukturschicht *ictrl* zur Realisierung von *iproc's* für die *CADMUS*-Workstation im Detail gezeigt.

5.3.4 Die Realisierung von *iproc's* – Die Infrastrukturschicht *ictrl*

Für die *iproc*-Implementation wird im folgenden auf die *CADMUS-9700* Workstation Bezug genommen. Diese Maschine besitzt zwei Prozessoren *Motorola m68020* [Mot85], wobei auf einem Prozessor das E/A-System (*ICC – Intelligent Communication Controller*) mit einem speziellen Kern (*icckernel*[12]) abläuft. Auf dem anderen Prozessor arbeitet der *CHEOPS*-Kern. Die Kommunikation zwischen beiden Kernen erfolgt über einen Bereich gemeinsamen Speichers und mittels Signalisierung über das Interruptsystem. Der Prozessor *m68020* unterstützt keine instanzbasierte Art der Unterbrechungsbehandlung, so daß *ictrl* als Softwareschicht zu implementieren ist. Diese Schicht muß an das *m68020*–Interruptsystem angebunden werden.

- ***Die Anbindung von** ictrl **an das** m68020–**Interruptsystem***

Man könnte diese Anbindung wiederum als eine Infrastruktur für *ictrl* auffassen und als separate Schicht abgrenzen. Der Sinn einer solchen Abgrenzung wäre jedoch fragwürdig, weil die einzige Aktivität in diesem Bereich das Interruptsystem des Prozessors ist. Hardwarekomponenten sollten aber nicht Bestandteile von Softwareschichten sein, so daß allein zwei Datenstrukturen `_cpuv[ ]` und `_irdv[ ]` als Softwareelemente verblieben, die eine Abgrenzung als Schicht nicht rechtfertigen. Deshalb werden `_cpuv[ ]` und `_irdv[ ]` mit in die Schicht *ictrl* eingeordnet.

Der Prozessor *m68020* kennt 256 Interrupttypen. Im Vektor `_cpuv[256]` steht für jeden Interrupttyp die Adresse der zugehörigen Interruptbehandlungsroutine, welche bei einem Interruptzyklus

[12] Der *icckernel* stammt vom Hersteller der *CADMUS*-Workstation und wurde unverändert übernommen.

"aufgerufen" wird. Die Basisadresse dieses Vektors steht in einem speziellen Prozessorregister `vbr` – *Vector Base Register*. Damit *ictrl* bei relevanten Interrupts aktiviert wird, muß in den entsprechenden Einträgen von `_cpuv`[256] die Startadresse des Programmcodes von *ictrl:* `_ictrl()` eingetragen sein. Für nichtzubehandelnde Interrupttypen ist die Adresse von `_ictrl_zom()` enthalten, wobei in `_ictrl_zom()` lediglich ein `rte` erfolgt (vgl. Abbildung 5.14).
Im zweiten Vektor `_irdv`[256] sind für alle Interrupttypen weitere Informationen enthalten, welche für den Ablauf von Behandlungsroutinen als *iproc's* erforderlich sind. Es sind dies die Komponenten: `_code` – die Startadresse der Behandlungsroutine für den *iproc*, `_sinfo` – eine Information für den *iproc*-Scheduler, z.B. eine statische Priorität, welche einem Interrupttyp zugeordnet sein kann und `_this` – dieser Eintrag ist aufgrund der Implementierung in *C++* notwendig, da Interruptbehandlungsroutinen für *iproc's* als nichtstatische *C++*-Member-Methoden implementiert sind, die einen `this`-Zeiger als erstes Argument erwarten [SE90].

Die Einträge in `_irdv`[no] können zur Laufzeit durch den Dienst von *ictrl:*

```
iassign( no, (* code)(), sinfo, _this)
```

neu belegt werden (vgl. Abschnitt 5.2.2), so daß sich beispielsweise neuer Code für den Ablauf neu erzeugter *iproc's* anbinden läßt, oder die Scheduling-Priorität kann geändert werden. Beide Vektoren `_cpuv[ ]`, `_irdv[ ]` und `vbr` werden vom Bootprozeß mit Anfangsbelegungen initialisiert.

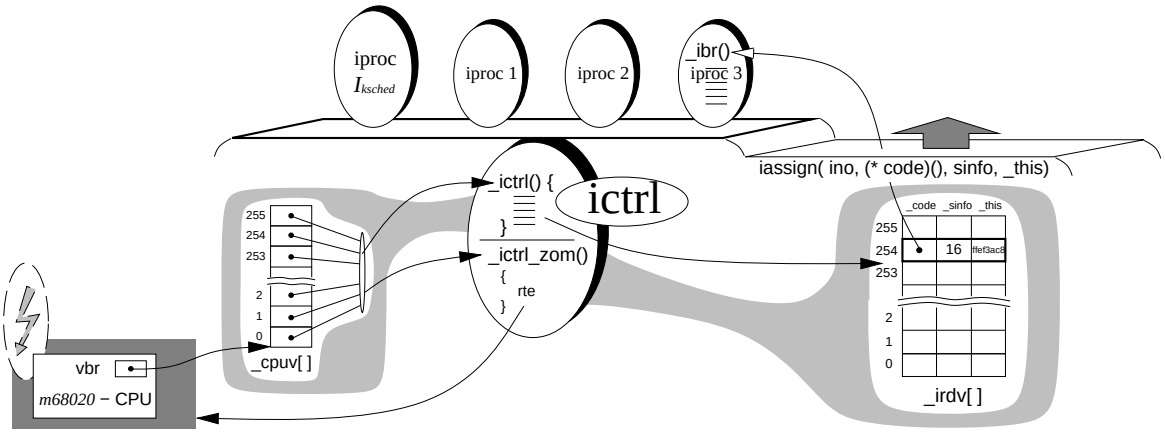


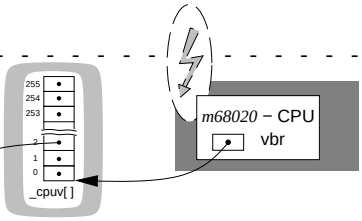
Abb.5.13 Die Anbindung von *ictrl* an das Interruptsystem des *m68020*-Prozessors

- ***Die Implementation von** ictrl — **Die Phasen*** [I] – [VI]

Die Implementation von *ictrl* soll anhand des *C++*-Quelltextes gezeigt werden, der zur besseren Übersicht in die vorn genannten sechs Phasen gegliedert wurde. Auf Seite 144 zeigt dann der Assemblertext das tatsächliche Geschehen auf der Ebene von Maschinenbefehlen bei einem Interrupt in *ictrl*. Die Folgen ausgeführter Maschinenbefehle bilden dann auch die Grundlage der Effektivitäts- bzw. Aufwandsbetrachtung für die sechs Phasen in Abschnitt 5.3.5.

- **Phase** [I] – Retten der Registerwerte der unterbrochenen Instanz P

Zuerst wird die reale Startadresse des Programmcodes für *ictrl* definiert (`_ictrl:`). Dies muß nach der Definition der statischen Funktion `_ictrl()` nochmals über `asm( '_ictrl:' )` erfolgen, um den vom Compiler generierten Vorspann bei Funktionen für die Interruptbehandlung auszuschalten. Anschließend werden für *ictrl* alle weiteren Interrupts gesperrt: `CPU::iDI()`.
`CPU` ist eine objektlose Klasse, welche softwareseitig ausführbare Steuerungsoperationen des Prozessors als statische `inline`-Methoden definiert, z.B. das Freigeben und Sperren von Interrupts. Statische `inline`-Methoden haben die Wirkung von Makros, d.h., der Methodencode wird an der Aufrufstelle eingefügt. Im erzeugten Assemblertext auf Seite 144 ist das erkennbar.




```
#include <iproc.h>
#include <ir.h>
#include <regs.h>
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

extern "C" void ecode();
extern iprocsC icbs;


static void _ictrl() {

    //--(Phase -I-)-------------------------------------------------------------------

        asm( ".globl _ictrl     ");
        asm( "_ictrl:           ");                 // Startadresse von ictrl()     [I]

        CPU::iDI();                                 // Interrupts sperren
        IRsaveR();                                  // Register nach P.regs retten

    //--(Phase -II-)------------------------------------------------------------------

        irfT *irf = (irfT *)CPU::sp();                  // irf analysieren
        irvC::irdT *ird = irv.get_ird( irf->no / 4);

        icbT *id = icbs.inew( ird->_code, ird->_sinfo); // neuen icb anlegen
                                                                                     [II]
        if( id) {                                       // und initialisieren
                id->istk.push( (int) ird->_this);
                id->istk.push( (int) &ecode);   // Fortsetzen am iproc-Ende
                id->regs.sp =  (int) id->istk.sp();
                id->regs.sr =   CPU::_iEI;      // iproc laeuft mit EI
        };

    //--(Phase -III-)-----------------------------------------------------------------
                                    // irf aendern, um bei _catch fortzusetzen
        asm( " movew    #0x2700, a7@    ");     // SR: Interrupts weiter gesperrt
        asm( " movel    #_catch, a7@(2) ");     // PC: auf _catch setzen
        asm( " rte                      ");     // Ende des Interruptzyklus der CPU
        //      |                                   |
        //      V                                   V
                                                                                     [III]
        asm( "_catch:           ");             // Fortsetzen nach rte
        IRsaveSP();                             // SP nach P.regs retten

    //--(Phase -IV-)------------------------------------------------------------------

    asm( "_sched:           ");
                                                // Re-Scheduling?               [IV]
        if( ! icbs.locked())                    // Scheduler aufrufen
                icbs.curip = icbs.schedule();   // curip: id des Folge-iproc

    //--(Phase -V-)-------------------------------------------------------------------

        if( icbs.curip->gs==icbT::CREAT) {      // Umschalten zu *curip
                                                                                     [V]
                icbs.curip->gs = icbT::RUN;

                startR(  & ( icbs.curip->regs));  // starte iproc

        } else  resumeR( & ( icbs.curip->regs));  // resume iproc

    //--(Phase -VI-)------------------------------------------------------------------
                                    // Abschlusscode am Ende eines iproc
        asm( "_ecode:           ");
        CPU::iDI();                             // ictrl: Interrupts sperren    [VI]
        icbs.ifree( icbs.curip);                // icb freigeben
        icbs.unlock();                          // Re-Scheduling freigeben
        asm( " jmp  _sched      ");             // -> Re-Scheduling
//------------------------------------------------------------------------------------
};


static void _ictrl_zom() {      // nichtbehandelte Interrupts abfangen
        //-------------------------------
        asm( ".globl _ictrl_zom ");
        asm( "_ictrl_zom:       ");
        asm( "  rte             ");
        //-------------------------------
};
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```


Abb.5.14 *C++*-Quelltext der *m68020*-Implementation für *ictrl*

Anschließend werden die Werte aller Register (außer `SP`) in den Rettebereich der unterbrochenen Instanz `P.regs` gespeichert, auf den stets der Zeiger `*savregs` verweist. Er wird bei jeder Prozessorumschaltung auf den Eintrag `P'.regs` in der Verwaltungsdatenstruktur der Ziel-Instanz *P'* gesetzt. Damit ist bei einem Interrupt sofort bekannt, wo sich der Rettebereich `P.regs` der unterbrochenen Instanz *P* befindet. Über die Methode `IRsaveR()` werden am Ende von Phase [I] die Werte aller Register in diesen Bereich über `*savregs` geschrieben. Prozessoren unterstützen diese Operation in der Regel durch spezielle Maschinenbefehle (*m68020:* `moveml #regmask, adr`). Der Wert des Stackpointers `SP` bleibt davon ausgenommen, weil er durch den auf dem Stack der unterbrochenen Instanz abgelegten *Interrupt-Frame* `irf` verfälscht ist und nicht dem Wert an der Unterbrechungsstelle in *P* entspricht (vgl. Abbildung 5.11 – `SP` nach dem Interrupt). Der Wert von `SP` kann dabei nicht einfach um die Größe von `irf` vermindert werden, weil diese vom Interrupttyp abhängt. Deshalb wird `SP` erst nach `rte` am Ende von Phase [III] gerettet, denn die `rte`-Instruktion stellt `SP` entsprechend dem Interrupttyp auf den ursprünglichen Wert vor dem Interrupt und damit auf den Wert an der Unterbrechungsstelle in *P* zurück.

Das Umschalten des Prozessors zwischen Instanzen erfolgt im *CHEOPS*-Kern jeweils anhand des aktuellen Zeigerwerts von `*savregs` und fünf zugehörigen Operationen, welche wiederum als `inline`-Methoden implementiert sind, so daß deren Code an den jeweiligen Aufrufstellen eingefügt wird. Da dieser Mechanismus zur Prozessorumschaltung grundlegender Natur ist, soll auch seine Implementation für *m68020* hier gezeigt werden. Damit wird auch der in Abschnitt 5.2.2 genannte implizite Dienst `isavregs` für die Schicht *ictrl* erklärt. Implizite Dienste können nicht an einer Instanz oder Schicht lokalisiert werden, sondern der Code für die Steuerung des Dienstes wird durch die Makro-Technik in die Programme der aufrufenden Instanzen eingefügt.

```
#ifndef __REGS_H_
#define __REGS_H_
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

typedef struct {        // Registersatz des m68020
  int   d[ 8];                  // D0..D7  - data registers
  int   a[ 6];                  // A1..A6  - address registers
  int   sp;                     // A7 = SP - kernel stackpointer
  int   a0;                     // A0      - address register
  int   usp;                    // USP - user stackpointer
  int   pc;                     // PC  - program counter
  short sr;                     // SR  - status register
} regsT;

                                // zeigt stets auf den Rettebereich
extern regsT *_savregs;         // der aktuell ablaufenden Instanz,
                                // wird nur in startR/resumeR gesetzt


inline void startR( regsT *from) {      // nur SP, PC, SR
        //-------------------------------
        _savregs = from;        // savregs beim Hinschalten setzen
        //-------------------------------
        asm( "  movel   %0, a7          ": :"g"( from->sp));
        asm( "  movel   %0, a7@-        ": :"g"( from->pc));
        asm( "  movew   %0, sr          ": :"g"( from->sr));
        asm( "  movel   %0, a0          ": :"g"( from->a0));
        asm( "  rts                     ");
        //-------------------------------
};


inline void resumeR( regsT *from) {     // alle Register
        //-------------------------------
        _savregs = from;        // savregs beim Hinschalten setzen
        //-------------------------------
        asm( "  movel   %0, a1          ": :"g"( from->usp));
        asm( "  movec   a1, usp         ");
        asm( "  moveml  a0@, #0xfeff    ");     // a1-a7,d0-d7
        asm( "  movel   %0, a7@-        ": :"g"( from->pc));
        asm( "  movew   %0, sr          ": :"g"( from->sr));
        asm( "  movel   %0, a0          ": :"g"( from->a0));
        asm( "  rts                     ");
        //-------------------------------
};
```

```
inline int saveR( regsT *to) {  // alle Register
  //-------------------------------
  asm( " movel  a0,a7@-        ");
  asm( " movel  %0,a0          ": :"g"( to));
  asm( " moveml #0x7eff, a0@   ");
  asm( " movel  a7@+, %0       ": :"g"( to->a0));
  asm( " movel  a7, %0         ": :"g"( to->sp));
  asm( " movec  usp, a1        ");
  asm( " movel  a1, %0         ": :"g"( to->usp));
  asm( " movew  sr, %0         ": :"g"( to->sr));
  //-------------------------------
  if( to) {
        asm( " movel  #___l0, %0": :"g"( to->pc));
        return( 1);
  } else        {
        asm volatile( "___l0: ");
        return( 0);
  };
  //-------------------------------
};

inline int IRsaveR() {          // alle, ausser SP
  //-------------------------------
  asm( " movel  a0,a7@-        ");
  asm( " movel  %0,a0          ": :"g"( _savregs));
  asm( " moveml #0x7eff, a0@   ");
  asm( " movel  a7@+, %0       ": :"g"( _savregs->a0));
  asm( " movec  usp, a1 ");
  asm( " movel  a1, %0         ": :"g"( _savregs->usp));
  asm( " movew  a7@, %0        ": :"g"( _savregs->sr));
  asm( " movel  a7@(2), %0     ": :"g"( _savregs->pc));
  //-------------------------------
};

inline int IRsaveSP() {         // nur SP
  //-------------------------------
  asm( " movel  a7, %0         ": :"g"( _savregs->sp));
  //-------------------------------
};
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
#endif __REGS_H_
```

Abb.5.15 Das Umschalten des Prozessors im *CHEOPS*-Kern

Die Speicherrepräsentation der Registerwerte wird durch die Struktur `regsT` beschrieben. Anhand dieser Struktur können dann auch Manipulationen von Registerwerten in den `regs`-Komponenten der Verwaltungsdatenstrukturen von Instanzen vorgenommen werden.
Für das Retten von Registerwerten nach `*savregs` gibt es zwei Methoden. Die allgemeine Form `saveR()` erfaßt alle Register. Die Rückkehrwerte entsprechen jenen von `setjmp()` der Standard-*C*-Bibliothek. Für das Retten der Registerwerte nach einem Interrupt gibt es wegen des oben genannten Problems mit `SP` eine Aufspaltung in zwei Methoden: `IRsaveR()` und `IRsaveSP()`.

Die eigentlichen Umschaltungen werden durch das Rückspeichern von vorher geretteten bzw. manipulierten Registerwerten (z.B. für den Start von Instanzen) aus dem Speicher in den Prozessor über `*savregs` ausgeführt. Die Methoden `startR()` und `resumeR()` stellen die einzigen Dienste dar, über die alle Umschaltungen stattfinden. Nur hier wird der Zeiger `*savregs` auf die jeweiligen Umschaltziele gesetzt, so daß `*savregs` stets auf den Register-Rettebereich der aktuell ablaufenden Instanz P zeigt und bei einem Interrupt für `IRsaveR()` sofort bekannt ist.

- **Phase** [II] – Erzeugen und Initialisieren eines `icb` für den neuen *iproc*

In Phase [II] wird der eingetroffene Interrupt analysiert, indem Werte aus dem *Interrupt-Frame* auf dem Stack von P ausgelesen werden. Zur Erzeugung eines `icb` ist nur der Interrupttyp erforderlich, der über die Interruptnummer `irf→no` kodiert ist. In `_irdv[no]` können dann:

`_code`, `_sinfo` und `_this`

indiziert und anschließend zur Initialisierung des erzeugten `icb` verwendet werden. Für die Erzeugung eines `icb` wird die Methode `inew( _code, _sinfo )` des Objektes `icbs` verwendet. Dieses Objekt existiert genau einmal in *ictrl* und stellt die Zusammenhänge zwischen den in *ictrl* verwalteten Daten her, vor allem zwischen `icb`'s. Die zugehörige Klassendefinition `iprocsC` beschreibt die dafür nötigen Elemente und Methoden.

```
#ifndef __IPROC_H_
#define __IPROC_H_

#include <types.h>
#include <tmpl.h>
#include <regs.h>

// - - - - - - - - - - - - - - - - - - - - - - - - -


struct icbT;
class   ilkC    : public cdl_listTMPL <icbT,0>  { };
class   istkC   : public stackTMPL <int>        { };
class   freelC  : public stackTMPL <icbT *>     { };


struct icbT {   // iproc control block
  ilkC::dlinkT     lk;  // doppelt verkettete RListe
  int           sinfo;  // icb-Scheduler-Info
  enum {CREAT,RUN} gs;  // icb-Globalzustand
  regsT          regs;  // Register Rettebereich
  istkC          istk;  // privater iproc-Stack
};

class iprocsC;
extern iprocsC icbs;    // Verwaltung aller icb und
                        // Re-Scheduling


// - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

class iprocsC {                 // Verwaltung von icb's in ictrl
    freelC      freel;          // icb-Freiliste
    int         _locked;        // Umschaltungen gesperrt?
    icbT *(*ischeduler)(ilkC &); // Zeiger auf Scheduler-Funktion

  public:
    icbT        *curip;         // aktuell ablaufender iproc
    ilkC        schedlist;      // Bereit-Liste fuer iprocs

    void lock()   { _locked = 1;};      // Umschaltungen sperren
    void unlock() { _locked = 0;};      // Umschaltungen erlauben
    int  locked() { return( _locked);};

    inline icbT *schedule() {    // Aufruf des iproc-Schedulers
        return( ischeduler( schedlist));
    };

    inline void set_scheduler( icbT *(*schedfunc)(ilkC &)) {
        ischeduler = schedfunc;
    };
    inline icbT *inew( fpT _code, int _sinfo) {
        if( freel.isempty())
            return( (icbT *) 0);
        icbT *id = freel.pop();  // icb aus Freiliste nehmen

        schedlist.insert( id);   // in Bereit-Liste einhaengen

        id->regs.pc =(int)_code; // icb initialisieren
        id->sinfo       = _sinfo;
        id->gs          = icbT::CREAT;
        return( id);
    };

    inline void ifree( icbT *id) {
        schedlist.remove( id);   // icb aus Bereit-Liste nehmen
        freel.push( id);         // in Freiliste einhaengen
        id->regs.sp = (int)id->istk.reset();
    };                           // istk beim Freigeben ruecksetzen
    void setup();                // icbs zu Beginn initialisieren
};
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
#endif __IPROC_H_
```

Abb.5.16
iproc-Verwaltungsdatenstrukturen in *ictrl* : `icbT`, `iprocsC` und `icbs`

In der Abbildung ist auch die Typbeschreibung für `icb` – `icbT` dargestellt (vgl. Abbildung 5.10). Analog zu dem oben genannten impliziten Dienst `isavregs`, welcher durch `inline`-Methoden zum Retten bzw. Rückspeichern von Registerinhalten realisiert war, implementieren auch die Methoden `inew` und `ifree` in `iprocsC` die in Abschnitt 5.2.2 genannten gleichnamigen impliziten Dienste für Operationen über Repräsentationen von *iproc's*. Am Ende von Phase [II] existiert ein neuer, komplett initialisierter `icb` in der Bereit-Liste von ictrl (→ `icbs.schedlist`).

- **Phase** [III] – Interruptzyklus des Prozessors beenden

In dieser Phase wird der Interruptzyklus des Prozessors durch die `rte`-Instruktion beendet. Die `rte`-Rückkehradresse (`PC`) ist im *Interrupt-Frame* auf dem Stack der unterbrochenen Instanz P abgelegt. Dieser Wert muß dort mit der Adresse von `_catch` überschrieben werden, damit nach `rte` in *ictrl* bei `_catch` fortgesetzt wird. Die Interrupts bleiben nach `rte` für *ictrl* weiter gesperrt, weshalb auch der Wert des Statusregisters `SR` in `irf` entsprechend belegt werden muß. Erst nach `rte` kann dann auch der ursprüngliche Wert des Stackpointers `SP` nach `P.regs` gerettet werden.

- **Phase [IV]** – Scheduling von *iproc's*

Der Scheduler arbeitet über der Bereit-Liste `icbs.schedlist` und wählt anhand einer Strategie einen darin enthaltenen `icb` aus, zu welchem dann in [V] umgeschaltet wird. Es muß stets mindestens ein `icb` in dieser Liste existieren, was durch die permanente Existenz der Instanz des *kproc*-Schedulers $\mathcal{I}_{ksched}$ gegeben ist. Der Code des *iproc*-Schedulers wird innerhalb der Instanz *ictrl* abgearbeitet. Um die Scheduling-Strategie aber auch softwaretechnisch vom Scheduling-Mechanismus zu trennen, wird der Scheduler als eine separate Funktion beschrieben, in welcher lediglich eine doppelt verkettete Ringliste von `icb`-Einträgen bekannt ist. Für das Scheduling kann der Wert von `icb.sinfo` benutzt werden. In der Verwaltungsdatenstruktur für alle *iproc's* – `icbs` – gibt es einen Zeiger, über den die Schedulerfunktion indirekt aufgerufen wird. Damit kann zur Laufzeit über `iprocsC::set_scheduler( icbT *(*schedfunc)(ilkC &) )` auch auf eine andere Schedulerfunktion umgeschaltet werden.

Eine mögliche Implementation eines *iproc*-Schedulers ist in Abbildung 5.17 mit der Funktion `icbT *isched( ilkC &rlist )` gezeigt, die als Argument eine Referenz auf die Bereit-Liste von *iproc's* hat. Hinter dem Typ `ilkC` verbirgt sich die Definition einer doppelt verketteten Ringliste, welche anhand eines *C++-Templates* für Elemente des Typs `icbT` definiert wurde. Auch das Element `iprocsC::schedlist` ist von diesem Typ (vgl. Abbildung 5.16). In Betriebssystemen hat man es häufig mit Listenstrukturen (zyklisch, einfach bzw. doppelt verkettet, sortiert u.a.) mit jeweils verschieden typisierten Elemente zu tun. Im Fall der Bereit-Liste von *iproc's* sind es Elemente `icb` vom Typ `icbT`. In *C++* eignen sich *Templates* für das Zusammenfassen allgemeiner Merkmale in einer *Template*-Definition, um daraus die entsprechenden typspezifischen Definitionen abzuleiten. *Templates* oder *generische Typen* sind eine einfache Alternative zu Vererbung. Für den *CHEOPS*-Kern wurden einige dieser *Templates* definiert, aus denen u.a. auch der Typ (`cdl_listTMPL <icbT,0>`, vgl. Abbildung 5.16) für die *iproc*-Bereit-Liste abgeleitet wurde. In den *Template*-Definitionen für Listen sind jeweils auch *Iteratoren* zum Durchmustern der Listen definiert, von denen einer für den in Abbildung 5.17 gezeigten *iproc*-Scheduler benutzt wird. Die Operationen über Listen sind ebenfalls komplett als `inline`-Methoden ausgeführt, so daß keine Prozedurrufe erforderlich sind.
Für das Scheduling von *iproc's* wird bei diesem Scheduler ein Verfahren mit statischen Prioritäten angewandt, wobei die Priorität anhand des Interrupttyps bestimmt werden. Das Element `icb.sinfo` wird als Prioritätsangabe interpretiert. Gibt es mehrere gleichtypisierte und damit gleichpriorisierte *iproc's*, wird der jeweils am längsten existierende *iproc* gewählt (→ *FIFO*-Ordnung in der Bereit-Liste). Diese Scheduling-Strategie orientiert sich damit am Vorbild statischer Priorisierung oder Privilegierung von Interrupts in herkömmlichen Interruptsystemen. Der wesentliche Unterschied ist aber, daß bei *iproc's* diese Strategie nur *eine Option*, bei der traditionellen Weise aber eine hardwareseitig bedingte Notwendigkeit ist.

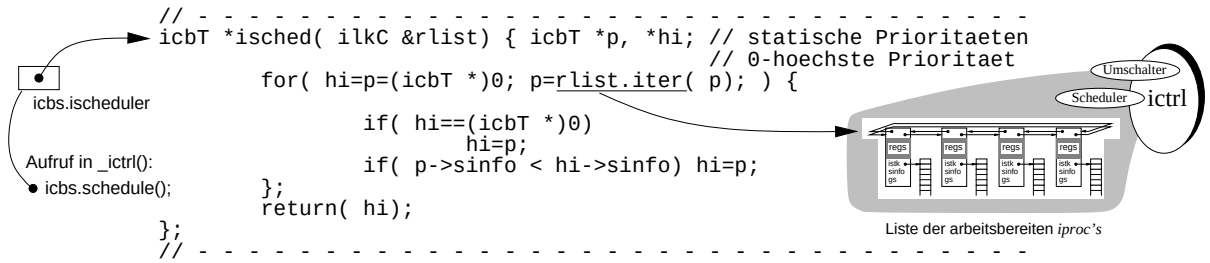


Abb.5.17 Beispiel für einen einfachen *iproc*-Scheduler mit statischen Prioritäten

Die Bereit-Liste ist nicht nach Prioritäten geordnet, so daß kein Aufwand beim Einfügen von `icb` nach Interrupts entsteht. Der Aufwand fällt hier beim Scheduling für das Bestimmen des höchstpriorisierten *iproc* mit einer linearen Suche in der Bereit-Liste an. Diese Aufwandsverteilung erscheint abwegig, aber sie kann hier durchaus angewandt werden, da Einfüge- und Such-

operationen in der Bereit-Liste in etwa im gleichen Verhältnis auftreten. Jeder Interrupt bewirkt ein Einfügen und ein Suchen. Außerdem enthält diese Liste in der Regel nur wenige Elemente. Oft werden es nur genau zwei Elemente sein: $\mathcal{I}_{ksched}$ und ein gerade erzeugter *iproc*. Dieser häufige Sonderfall läßt sich natürlich optimieren, was nicht in Abbildung 5.17 gezeigt ist. Auch diese Eigenschaften unterscheiden *iproc's* von *kproc's*, so daß durch die getrennte Behandlung Optimierungen für *iproc's* möglich werden.

- **Phase** [V] – Starten bzw. Umschalten zum nächsten *iproc*

In dieser Phase erfolgt die Umschaltung zu dem in [IV] ausgewählten *iproc*, indem mittels `startR` bzw. `resumeR` Registerwerte aus dem Rettebereich `icb.regs` dieser Instanz in den Prozessor zurückgeladen werden. Es wird dabei zwischen dem Start einer neuen (→ `startR`) und dem Wiederanlauf, d.h. dem Fortsetzen einer unterbrochenen *iproc*-Instanz (→ `resumeR`) unterschieden. Die Implementation beider Operationen wurde in Abbildung 5.15 gezeigt. Für neu gestartete *iproc's* ist es ausreichend, nur Registerwerte für `PC`, `SR` und `SP` in den Prozessor zu laden, während im Fall des Fortsetzens eines unterbrochenen *iproc* der komplette Registersatz wiederherzustellen ist. Allein daraus ergeben sich auch die beiden Globalzustände für *iproc's:* `CREAT` und `RUN`. Beide Umschaltoperationen bewirken das Verlassen von *ictrl*.

- **Phase** [VI] – Fortsetzen nach dem Ende eines *iproc*

Nach dem Ablauf eines *iproc* muß an dessen Ende eine Rückschaltung zu *ictrl* erfolgen. Dieser Effekt wird erreicht, indem beim Initialisieren des zugehörigen `icb` in Phase [II], d.h. vor dem Ablauf des *iproc*, die Adresse `_ecode` auf dem Laufzeitstack des *iproc* abgelegt wird, so daß sich eine `return`-Operation am Ende der *iproc*-Behandlungsroutine auf diese Adresse bezieht und das Rückschalten zu *ictrl* bewirkt. Für *iproc's* ist das nicht nur eine Rückkehr aus der Behandlungsroutine, sondern bewirkt auch das Ende des *iproc* und die Freigabe des `icb`. Beendete *iproc's* besitzen keinen zu rettenden Ablaufzustand mehr, so daß für die Umschaltung nach *ictrl* kein Ablaufzustand zu retten ist. Auch *ictrl* hat keinen Ablaufzustand, so daß das Setzen von `PC` auf die Adresse `_ecode` für das Umschalten ausreicht. Dies erfolgt durch die vom Compiler generierte `return`-Operation am Ende der Behandlungsroutine für die Rückkehradresse `_ecode`. Das Register `SP` muß dabei auch nicht zurückgeschaltet werden, da *ictrl* keinen eigenen Stackbereich besitzt. Am Ende eines *iproc* zeigt `SP` noch auf den Stack der beendeten Instanz, so daß dieser für den Ablauf des Re-Schedulings und das anschließende Weiterschalten in *ictrl* noch benutzt wird. An der Stelle `_ecode` werden die Interrupts wieder für *ictrl* gesperrt und der `icb` durch Umketten in die Freiliste freigegeben.
Dieses "zustandslose" (außer `PC`) Rückschalten zu *ictrl* am Ende eines *iproc* ist ein wesentlicher Unterschied zu der eingangs diskutierten alternativen Form zyklischer Interruptbehandlungsprozesse. Diese können nach dem Durchlauf einer Interruptbehandlung nicht "zustandslos" beendet werden, sondern der Ablaufzustand muß beim Blockieren in `wait_on_signal()` im folgenden Zyklus komplett (!) gerettet werden. Das Erzeugen und Beseitigen von `icb` als Repräsentanten für *iproc's* ist unter diesen speziellen Bedingungen mit weniger Aufwand verbunden als das Retten und Wiederherstellen des Ablaufzustands zyklischer Instanzen beim Blockieren auf `wait_on_signal()`. Zudem kommen die hier verwendeten *pop up*-Instanzen ohne den Blockierzustand aus, was deren Verwaltung vereinfacht. Auch hierin lag ein wichtiger Grund für die Entkopplung von *iproc's* und *kproc's* in separate Schichten. Diese Trennung ist damit nicht nur aus den hier verfolgten Architekturzielen begründet, sondern ergibt sich auch aus dem Beweggrund einer spezialisierten und damit optimierten Ablaufform bei der Verwaltung von *iproc's*.

Fazit: Mit *iproc's* kann gezeigt werden, daß sich auch für das seit langem im Prinzip gleiche Behandlungsmuster für Interrupts neue Konzepte anwenden lassen, deren Vorteile in Abschnitt 5.3.6 kompakt zusammengefaßt werden, aber deren Aufwand auf Basis heutiger Prozessoren auch nicht verschwiegen werden soll. Er ist im folgenden Abschnitt Gegenstand der Untersuchung.

5.3.5 Aufwandsbetrachtung für *iproc's*

Die Unterbrechungsverarbeitung mit *iproc's* bietet ein höheres Maß an Flexibilität, bringt aber auch einen höheren Aufwand hinsichtlich Speicher und auch Laufzeit mit sich.

Der ***Speicheraufwand*** für Verwaltungsdaten `icb` kann als unkritisch angesehen werden. Die privaten Stacks von *iproc's* haben in der hier vorgestellten Implementation eine statisch fixierte Größe von je 512 Byte. Pro `icb` sind damit 92+512=604 Byte notwendig. In der Freiliste sind acht `icb` enthalten, so daß sich ein Gesamtspeicherbedarf für `icb` von 4832 Byte ergibt. Es kommen die Datenstrukturen `icbs` (36), `savregs` (4) und die beiden Vektoren `_cpuv[ ]` (1024) und `_irdv[ ]` (3072) hinzu. Es ergibt sich ein Gesamtspeicherbedarf für Daten von 8968 Byte. Die Programmcodes in *ictrl* für `_ictrl()` (464) `_ictrl_zom()` (6) umfassen 470 Byte.
Damit ergibt sich ein Gesamtspeicherbedarf für *ictrl* von ca. ***10 kByte***, der bei heutigen Speicherausbauten von Maschinen als akzeptierbar angesehen werden kann.

Wesentlich schwieriger ist die Bestimmung des ***Laufzeitaufwands*** und die Quantifizierung eines Mehraufwands gegenüber der traditionellen Art der Interruptbehandlung. Die Schwierigkeit ist zum einen inhaltlicher Art. *iproc's* bieten mehr Flexibilität und können auch strukturell als vorteilhaft gegenüber herkömmlichen Interruptbehandlungen angesehen werden. Die Vorteile von *iproc's* werden im folgenden Abschnitt noch einmal kompakt zusammengestellt. Der Preis ist ein Mehraufwand. Insofern liegt es in der Natur der Sache, daß ein direkter Laufzeitvergleich mit herkömmlichen Interruptbehandlungen an sich nicht stimmig ist. Direkte Effizienzvergleiche sind in der Regel nur sinnvoll, wenn ein gleiches Prinzip auf verschiedene Weise implementiert wird. Hier soll es daher darum gehen, den Aufwand für *iproc's* aufgeteilt auf die oben eingeführten sechs Phasen in *ictrl* zu bestimmen. Erst diese feinere Auflösung läßt eine Aussage darüber zu, wodurch der Laufzeit-Mehraufwand bei *iproc's* verursacht wird, in welchen Größenordnungen er liegt und wo Optimierungsmöglichkeiten bestehen.
Ein zweites Problem bei der Bestimmung des Ablaufaufwands von Interruptbehandlungen ist technischer Art. Softwaretechnische Zeitmessungen sind unmöglich, weil deren feinste Auflösung im Zeitbereich des Zeitinterrupts und damit wesentlich über dem hier zu bestimmenden Zeitintervall liegt. Bei der betrachteten Maschine beträgt die Auflösung des Zeitinterrupts $20ms$ (50 Hz). Die Ablaufaufzeit in *ictrl* wird sich in in einem Bereich von ca. $50\mu s$ bewegen, d.h., sie ist um den Faktor $4 * 10^2$ feiner als die Auflösung des Zeitinterrupts. Diese Zeiten sind folglich softwaretechnisch nicht zu messen. Die *CADMUS*-Workstation verfügt auch nicht über eine hochauflösende interne Hardwareuhr, wie sie heute in Systemen für feingranulare Zeitmessungen eingesetzt wird. Die Methode des Hardwaremonitorings und der Bestimmung von Signallaufzeiten auf dem Systembus für ausgeführte Befehle kam wegen des damit verbundenen technischen Aufwands nicht in Betracht.

Das hier gewählte Verfahren für die Bestimmung des Ablaufaufwands basiert auf der Bestimmung der Ausführungszeiten der in *ictrl* ausgeführten Maschinenbefehle. Dieser pragmatische Ansatz hat den Vorteil, daß er mit einfachen Mitteln realisierbar ist. Aber er kann auch nicht völlig unkritisch gesehen werden.
Ein erstes Problem besteht darin, die tatsächliche Folge von ausgeführten Maschinenbefehlen zu bestimmen. Diese hängt von dynamischen Parametern ab, beispielsweise von der Länge der Bereit-Liste für das *iproc*-Scheduling. Der Verarbeitungsprozeß in *ictrl* muß daher in eine Folge linearer Befehlssequenzen (Teilprozesse) aufgespaltet werden, deren Ablauf von dynamisch bedingten Zyklen und Verzweigungen abhängt. Ein zweites Kriterium für die Aufspaltung in Befehlssequenzen ist die Gruppierung nach den sechs Verarbeitungsphasen in *ictrl*, so daß diesen später Verarbeitungszeiten als Maße für den Ablaufaufwand zugeordnet werden können. Eine Befehlssequenz wird somit durch eine Phase und eine Teilaktion (a, b, c, ...) innerhalb dieser Phase gekennzeichnet: [I].a – [VI].c. Welche Folge von Befehlssequenzen tatsächlich ausgeführt wird,

hängt von dynamischen Parametern ab und ist nicht vorhersagbar. Es müßten daher alle (!) möglichen Kombinationen von Folgen betrachtet und bewertet werden. Um den Aufwand dafür zu reduzieren, werden hier nur einige wenige, aber häufig vorkommende Folgen erfaßt. So werden bspw. alle Fehlersituationen ausgeschlossen, weil sie nicht den typischen Ablauf repräsentieren. Auch das Scheduling wird auf den typischen Fall reduziert, daß es genau einen *iproc* und die Instanz $\mathcal{I}_{ksched}$ für das Scheduling von *kproc's* gibt, so daß die ausgeführte Befehlsfolge für diesen Fall feststeht.

Sind die Folgen von Befehlssequenzen auf die typischen Fälle eingeschränkt, liegt ein zweites Problem bei der Bestimmung ihrer Ausführungszeit aus der Summe der Ausführungszeiten der einzelnen Maschinenbefehle. Das keine trivial Aufgabe. Einerseits gibt es für *CISC*-Prozessoren eine breite Vielfalt von Adressierungsarten, die mit der Mehrzahl der Befehlen kombiniert werden können und zu unterschiedlichen Ausführungszeiten führen. Zum anderen kommt wesentlich schwerwiegender hinzu, daß durch Optimierungstechniken in Prozessoren die Ausführungszeit eines Maschinenbefehls nicht feststeht, sondern von der Folge der ausgeführten Maschinenbefehle und weiteren Randbedingungen abhängt. Prozessoren verwenden heute – und auch schon der *Motorola m68020* – Befehls-Caches, deren Inhalt nicht vorherbestimmbar ist. Zudem werden Befehle im Voraus gelesen und parallel zur Ausführung vorausgehender Instruktionen vorverarbeitet (*Befehls-pipelining*), so daß sich eine gleichzeitige, verzahnte Ausführung mehrerer Maschinenbefehle ergibt. Auch die Daten- oder Operanden-Caches spielen für die notwendige Taktanzahl eine wichtige Rolle, bis alle Operanden im Prozessor vorliegen und ein Maschinenbefehl ausgeführt werden kann. Vor allem bei hochgetakteten Prozessoren sind auch die Zugriffszeit zum Hauptspeicher und der Verkehr auf dem Bussystem wesentlich. In [Mot85] werden für den Prozessor *Motorola m68020* drei wesentliche Einflußgrößen auf die Ausführungszeit von Maschinenbefehlen genannt (der *m68020* besitzt nur einen Befehls-, keinen Operanden-Cache):

- *On-Chip Instruction Cache* und *Instruction Prefetch*,
- *Operand Misalignment* (Operanden liegen nicht auf 4-Byte ausgerichteten Adressen und können nicht in einem Speicherzyklus geladen bzw. gespeichert werden) und
- *Instruction Execution Overlap* – verursacht durch *Prefetch* und die teilweise parallele Ausführung von Instruktionen.

In [Mot85] sind daher keine determinierten Ausführungszeiten für Maschinenbefehle enthalten, sondern es werden Taktzahlen für die Befehlsausführung in drei Abstufungen angegeben:

`BC`	– *Best Case*	Der Befehl befindet sich im Befehls-Cache und kann mit einem maximalen Parallelitätsgrad ausgeführt werden, d.h., er wird u.U. bereits vollständig parallel zum vorausgegangenen Befehl ausgeführt, was durch eine Ausführungszeit von 0 Takten gekennzeichnet wird (z.B. `addl d1,d0` mit `0/2/3` in Zeile 24 in Abbildung 5.18).
`CC`	– *Cache Case*	Der Befehl ist bereits im Befehls-Cache, aber es kann kein Teil des Befehls parallel ausgeführt werden.
`WC`	– *Worst Case*	Der Befehle steht nicht im Befehls-Cache, und es findet keine parallele Ausführung statt.

Aus den drei Taktzahlen folgt ein Muster `BC/CC/WC`, welches ein Profil für die Ausführung eines Befehls beschreibt. Die zweite Dimension für die Bestimmung der Ausführungszeiten bei *CISC*-Prozessoren wurde mit der Vielzahl von Adressierungsarten bereits genannt. Die Taktanzahl ergibt sich aus einem Maschinenbefehl in Kombination mit einer oder zwei Adressierungsarten für Operanden. Sie wurden hier anhand der in [Mot85] angegebenen Tabellen errechnet.

Für die Bewertung der Ausführungszeiten der einzelnen Phasen werden die in Abbildung 5.18 eingetragenen typischen Befehlssequenzen [I].a – [VI].c herangezogen.

```
 1 ;---------------------------------(1)-
 2 gcc2_compiled (version 2.3.2):
 3
 4 .globl _ictrl
 5 _ictrl:                          [I].a
 6   movew #9984, sr                      ;8/10/14
 7   movel  a0,a7@-                 [I].b ;3/ 5/ 6
 8   movel  __savregs,a0                  ;3/ 6/10
 9   moveml #0x7eff, a0@                 ;66/68/70
10   movel __savregs,a0                   ;3/ 6/10
11   movel  a7@+, a0@(60)                 ;7/ 7/11
12   movec  usp, a1                       ;3/ 6/ 7
13   movel  a1, a0@(64)                   ;3/ 5/ 7
14   movew  a7@, a0@(72)                  ;3/ 5/ 7
15   movel  a7@(2), a0@(68)               ;6/ 8/13
16
17 ;---------------------------------(2)-
18   movel sp, a0                 [II].a  ;0/ 2/ 3
19   movew a0@(6),d1                      ;3/ 7/ 9
20   andl #4095,d1                        ;0/ 4/ 6
21   lsrl #2,d1                           ;1/ 4/ 4
22   movel d1,d0                          ;0/ 2/ 3
23   addl d0,d0                           ;0/ 2/ 3
24   addl d1,d0                           ;0/ 2/ 3
25   asll #2,d0                           ;5/ 8/ 8
26   movel d0,a3                          ;0/ 2/ 3
27   addl #_irv+1024,a3                   ;0/ 6/ 8
28   movel a3@,d1                         ;3/ 6/ 7
29   movel a3@(8),d2                      ;3/ 7/ 9
30   movel _icbs+4,d0             [II].b  ;3/ 6/10
31   subl _icbs,d0                        ;3/ 6/10
32   asrl #2,d0                           ;3/ 6/ 6
33   subl _icbs+8,d0                      ;3/ 6/10
34   negl d0                              ;0/ 2/ 3
35   tstl d0                              ;0/ 2/ 3
36   jle L212                             ;3/ 6/ 9
37   movel _icbs+4,a0                     ;3/ 6/10
38   movel a0@,a1                         ;3/ 6/ 7
39   addql #4,_icbs+4                     ;6/ 8/13
40   movel _icbs+24,a2                    ;3/ 6/10
41   cmpw #0,a2                           ;1/ 6/ 7
42   jeq L208                             ;3/ 6/ 9
43   movel a2@,a1@                        ;6/ 7/ 9
44   movel a2,a1@(4)                      ;3/ 5/ 7
45   movel a2@,a0                         ;3/ 6/ 7
46   movel a1,a0@(4)                      ;3/ 5/ 7
47   movel a1,a2@                         ;3/ 4/ 5
48   jra L206                             ;3/ 6/ 9
49 L208:
50   movel a1,a1@(4)                      ;3/ 5/ 7
51   movel a1,a1@                         ;3/ 4/ 5
52   movel a1,_icbs+24                    ;5/ 6/ 9
53 L206:
54   movel d1,a1@(84)                     ;3/ 5/ 7
55   movel d2,a1@(8)                      ;3/ 5/ 7
56   clrl a1@(12)                         ;6/ 9/12
57   cmpw #0,a1                   [II].c  ;1/ 4/ 4
58   jeq L212                             ;3/ 6/ 9
59   movel a3@(4),d0                      ;3/ 7/ 9
60   subql #4,a1@(94)                     ;6/ 9/12
61   movel a1@(94),a0                     ;3/ 7/ 9
62   movel d0,a0@                         ;3/ 4/ 5
63   subql #4,a1@(94)                     ;6/ 9/12
64   movel a1@(94),a0                     ;3/ 7/ 9
65   movel #_ecode,a0@                    ;3/ 8/ 7
66   movel a1@(94),a1@(72)                ;6/ 8/13
67   movew #8192,a1@(88)                  ;3/ 7/ 7
68
69 ;---------------------------------(3)-
70 L212:
71   movew #0x2700, a7@           [III].a ;3/ 6/ 5
72   movel #_catch, a7@(2)                ;3/ 9/ 9
73   rte                          [III].b ;20/21/24
74
75 _catch:
76   movel __savregs,a0           [III].c ;3/ 6/10
77   movel  a7, a0@(56)                   ;3/ 5/ 7
78 ;-------------------------------------
```

```
79 ;---------------------------------(4)-
80 _sched:
81   tstl _icbs+12          [IV].c [IV].a ;3/ 6/10
82   jne L221                             ;3/ 6/ 9
83   pea _icbs+24                  [IV].b ;6/11/13
84   movel _icbs+16,a0                    ;3/ 6/10
85   jbsr a0@                      isched() ;3/ 7/13
86   movel d0,_icbs+20                    ;5/ 6/ 9
87 ;---------------------------------(5)-
88 L221:
89   movel _icbs+20,a0            [V].b   ;3/ 6/10
90   tstl a0@(12)                 [V].a   ;3/ 7/ 9
91   jne L226                             ;3/ 6/ 9
92   moveq #1,d3                          ;0/ 4/ 3
93   movel d3,a0@(12)                     ;3/ 5/ 7
94   movel _icbs+20,a4                    ;3/ 6/10
95   lea a4@(16),a0                       ;4/ 4/ 6
96   movel a0,__savregs                   ;5/ 6/ 9
97   movel a0@(56), a7                    ;3/ 7/ 9
98   movel a0@(68), a7@-                  ;6/ 8/11
99   movew a0@(72), sr                   ;11/13/17
100  movel a0@(60), a0                    ;3/ 7/ 9
101  jra L246                             ;3/ 6/ 9
102 L226:
103  movel _icbs+20,a4                    ;3/ 6/10
104  lea a4@(16),a0                       ;4/ 4/ 6
105  movel a0,__savregs                   ;5/ 6/ 9
106  movel a0@(64), a1                    ;3/ 7/ 9
107  movec a1, usp                        ;9/12/13
108  moveml a0@, #0xfeff                 ;68/70/72
109  movel a0@(68), a7@-                  ;6/ 8/11
110  movew a0@(72), sr                    ;8/10/14
111  movel a0@(60), a0                    ;3/ 7/ 9
112 L246:
113  rts                                  ;9/10/12
114 ;-------------------------------------
115
116
117 ;---------------------------------(6)-
118 _ecode:                        [VI].a
119   movew #9984, sr                      ;8/10/14
120   movel _icbs+20,a1            [VI].b  ;3/ 6/10
121   movel a1@(4),a0                      ;3/ 7/ 9
122   movel a1@,a0@                        ;6/ 7/ 9
123   movel a1@,a0                         ;3/ 6/ 7
124   movel a1@(4),a0@(4)                  ;6/ 8/13
125   cmpl _icbs+24:l,a1                   ;4/ 8/11
126   jne L235                             ;3/ 6/ 9
127   cmpl a1@,a1                          ;4/ 8/ 8
128   jne L237                             ;3/ 6/ 9
129   clrl _icbs+24                        ;6/10/13
130   jra L235                             ;3/ 6/ 9
131 L237:
132   movel a1@,_icbs+24                   ;8/ 9/13
133 L235:
134   subql #4,_icbs+4                     ;6/ 8/13
135   movel _icbs+4,a0                     ;3/ 6/10
136   movel a1,a0@                         ;3/ 4/ 5
137   movel a1@(98),d0                     ;3/ 7/ 9
138   asll #2,d0                           ;5/ 8/ 8
139   addl a1@(90),d0                      ;3/ 7/ 9
140   movel d0,a1@(94)                     ;3/ 5/ 7
141   movel d0,a1@(72)                     ;3/ 5/ 7
142   clrl _icbs+12                [VI].c  ;6/10/13
143   jmp _sched                           ;1/ 6/ 9
144 ;-------------------------------------
145
146   moveml a6@(-20),#0x1c0c
147   unlk a6
148   rts
149
150 ;-------------------------------------
```

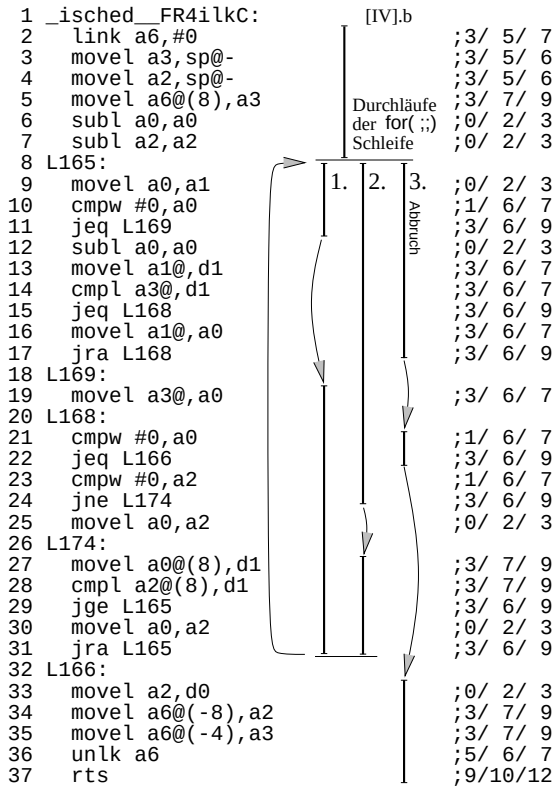
Abb.5.18 *m68020*-Assemblertext für *ictrl*

In Abbildung 5.18 ist der aus *C++* (vgl. Abbildung 5.14) mit *gcc2.3.2* generierte Assemblertext gezeigt, und es sind die typischen Befehlssequenzen für die sechs Verarbeitungsphasen in *ictrl* eingetragen. Jedem Maschinenbefehl ist eine Taktzahl nach dem Muster `BC/CC/WC` zugeordnet. Es werden nicht alle möglichen Befehlssequenzen betrachtet, sondern nur ein fehlerfreier Durchlauf für die Erzeugung, das Scheduling, die Umschaltung und das Beenden eines *iproc*. Der Code von *ictrl* wird dabei vorwiegend linear durchlaufen. Es gibt keine Zyklen, und die meisten Verzweigungsmöglichkeiten resultieren aus dem Prüfen auf mögliche Fehlerzustände, daß es beispielsweise keine freien `icb` mehr in der Freiliste gibt. Aus dem dafür nötigen Test in [II]:

```
id = icbs.inew( ...);
if( id) {
    /* ok */
};
```

```
57  cmpw #0,a1          ;Vergleich
58  jeq L212            ;bedingter Sprung
59  movel a3@(4),d0
60  ...
```

ergeben sich zwei mögliche Folgen von Maschinenbefehlen (vgl. Zeile 58 im Assemblertext). Fehlersituationen sind aber keine typischen Fälle, sondern Ausnahmesituationen, für die zwar Befehlssequenzen vorzusehen sind, welche aber im regulären Ablauf nicht durchlaufen werden und für die Aufwandsbetrachtung ignoriert werden können. Es wird nur an einer Stelle eine Verzweigung und damit ein Paar alternativer Befehlssequenzen betrachtet, nämlich für die Unterscheidung des Starts bzw. des Wiederanlaufs einer *iproc*-Instanz in Phase [V], da hier während des Ablaufs durchaus beide Fälle eintreten können, wenngleich der Fall des Wiederanlaufs eines unterbrochenen *iproc* seltener ist als der Start eines neuen *iproc*. Beide Varianten werden deshalb durch zwei alternative Befehlssequenzen [V].a (→ `startR`) bzw. [V].b (→ `resumeR`) erfaßt.




```
 1 _isched__FR4ilkC:
 2   link a6,#0                ;3/ 5/ 7
 3   movel a3,sp@-             ;3/ 5/ 6
 4   movel a2,sp@-             ;3/ 5/ 6
 5   movel a6@(8),a3           ;3/ 7/ 9
 6   subl a0,a0                ;0/ 2/ 3
 7   subl a2,a2                ;0/ 2/ 3
 8 L165:
 9   movel a0,a1               ;0/ 2/ 3
10   cmpw #0,a0                ;1/ 6/ 7
11   jeq L169                  ;3/ 6/ 9
12   subl a0,a0                ;0/ 2/ 3
13   movel a1@,d1              ;3/ 6/ 7
14   cmpl a3@,d1               ;3/ 6/ 7
15   jeq L168                  ;3/ 6/ 9
16   movel a1@,a0              ;3/ 6/ 7
17   jra L168                  ;3/ 6/ 9
18 L169:
19   movel a3@,a0              ;3/ 6/ 7
20 L168:
21   cmpw #0,a0                ;1/ 6/ 7
22   jeq L166                  ;3/ 6/ 9
23   cmpw #0,a2                ;1/ 6/ 7
24   jne L174                  ;3/ 6/ 9
25   movel a0,a2               ;0/ 2/ 3
26 L174:
27   movel a0@(8),d1           ;3/ 7/ 9
28   cmpl a2@(8),d1            ;3/ 7/ 9
29   jge L165                  ;3/ 6/ 9
30   movel a0,a2               ;0/ 2/ 3
31   jra L165                  ;3/ 6/ 9
32 L166:
33   movel a2,d0               ;0/ 2/ 3
34   movel a6@(-8),a2          ;3/ 7/ 9
35   movel a6@(-4),a3          ;3/ 7/ 9
36   unlk a6                   ;5/ 6/ 7
37   rts                       ;9/10/12
```

Abb.5.19 *iproc*-Scheduling in `isched()`

Im Assemblertext von *ictrl* ist der Code für den in Abbildung 5.17 gezeigten einfachen *iproc*-Scheduler nicht enthalten, in Zeile 85 steht lediglich der indirekte Aufruf der Schedulerfunktion `isched()`. Der Ablauf der Befehlssequenz für das Scheduling in [IV].b ist in nebenstehend in Abbildung 5.19 dargestellt. Es wird dabei der häufige Fall betrachtet, daß es genau zwei *iproc*-Instanzen gibt: $\mathcal{I}_{ksched}$ und einen gerade erzeugten *iproc*. Es gibt in diesem Fall genau zwei Durchläufe der Iterator-Schleife über der Bereits-Liste `icbs.schedlist`. Dieser Sonderfall kann natürlich optimiert werden, um auf das Suchen in der Bereits-Liste zu verzichten, da der vom Scheduler ausgewählte Kandidat bereits im voraus feststeht. Hier soll aber ein typischer Aufwand bestimmt werden, so daß diese Optimierungsmöglichkeit hier nicht betrachtet wird. Der genannte typische Ablauf in der Schedulerfunktion `isched()` bewirkt zwei Schleifendurchläufe (`1.`, `2.`), da die Bereit-Liste genau zwei `icb`-Elemente enthält. Der `3.`Ablauf bedeutet den Abbruch der Schleife und damit das Ende von `isched`. Faßt man alle Abläufe für diesen typischen Fall des Schedulings von zwei *iproc*-Instanzen zusammen, läßt sich eine Taktzahl von 124/294/390 als Aufwand für die Befehlssequenz [IV].b des Schedulings von *iproc's* ermitteln.

Die berechneten Taktzahlen können für die *CADMUS-9700* Workstation direkt in Ausführungszeiten umgerechnet werden. Die beiden *m68020*-Prozessoren werden mit 25 MHz getaktet, so daß sich eine Ausführungszeit pro Takt von $40ns$ ergibt. Um eine Abschätzung für die Ablaufzeit

eines Maschinenbefehls zu erhalten, wird eine durchschnittliche Ausführungszeit $t_{\bar{i}}$ definiert, welche sich aus dem gewichteten Mittel der Takte für BC/CC/WC nach folgender Formel ergibt.

$$t_{\bar{i}} \quad = \quad (BC * 0.15 + CC * 0.30 + WC * 0.55) \ * \ 40ns$$

Die Gewichte wurden intuitiv gewählt, da in [Mot85] keine Angaben über typische Verteilungen enthalten sind. Der kleine Anteil für BC bzw. der relativ große Anteil für WC wird damit begründet, daß die Cache-Wirkung bei Interrupts nicht in dem Maße gegeben ist, weil die Maschinenbefehle natürlich nicht im *prefetch* gelesen und vorverarbeitet werden können. Andererseits kommt der Befehls-Cache im Verlauf der Verarbeitung in *ictrl* wieder zum Wirken, so daß der Cache-Effekt nicht völlig ausgeschlossen werden soll und etwa nur der *Worst Case* zu betrachten wäre.

Unter den genannten Randbedingungen und Einschränkungen läßt sich eine realistische Näherung der Ausführungszeiten der Befehlssequenzen in den einzelnen Phasen in *ictrl* angeben, welche in der folgenden Tabelle 5.20 zusammengefaßt ist.

Befehls-sequenz	vgl. Abbildung 5.14	Ablaufbeschreibung	$\sum_{Takte}^{(BC/CC/WC)}$	$t_{\bar{i}}$
[I].a	`CPU::iDI();`	Interrupts sperren	8/10/14	0.48 μs
[I].b	`IRsaveR();`	Registersatz nach `P.regs` retten	97/116/141	5.08 μs
[II].a	`CPU::sp();` `irv.get_ird(...);`	`irf` analysieren	15/52/66	2.17 μs
[II].b	`icbs.inew(...);`	`icb` erzeugen und	67/124/177	5.78 μs
[II].c	`if(id){ ... };`	`icb` initialisieren	40/76/96	3.26 μs
[III].a	`asm('...');`	`irf` manipulieren	6/15/14	0.52 μs
[III].b	`asm('rte');`	CPU-Interruptzyklus beenden	20/21/24	0.90 μs
[III].c	`IRsaveSP();`	`SP` nach `P.regs` retten	6/11/17	0.54 μs

[IV].a	`_sched:`
	`if( ! locked()) {`
	`  icbs.schedule();`
		Scheduler aufrufen ?	23/42/64	2.05 μs

[IV].b	`  /*run isched(); */;`	→ Schedulerablauf	124/294/390	12.85 μs
[IV].c	`};`	`else:` kein Re-Scheduling	6/12/19	0.60 μs
[V].a	`if(CREAT) {` `  startR(...);`	starte *iproc*	59/95/130	4.35 μs

[V].b	`} else {`
	`  resumeR(...);`
	`};`
		setze *iproc* fort	127/159/193	6.92 μs

[VI].a	`CPU::iDI();`	Interrupts für *ictrl* sperren	8/10/14	0.48 μs
[VI].b	`icbs.ifree(icb);`	`icb` freigeben	57/98/136	4.51 μs
[VI].c	`icbs.unlock();` `asm('jmp _sched');`	erneutes Re-Scheduling	7/16/22	0.72 μs

Abb.5.20 Näherungen für Ausführungszeiten der Befehlssequenzen in *ictrl*

Der Aufwand für die einzelnen Phasen läßt sich nochmals in einer kompakten Form zusammenfassen, so daß sich ein Bild über den Gesamtaufwand und die Verteilung auf die Einzelphasen ergibt. Diese zusammengefaßte Aufwandsverteilung ist in der folgenden Tabelle 5.21 dargestellt.

Phase	Ablaufbeschreibung	$\sum_{Takte}^{(BC/CC/WC)}$	$t_{\bar{i}}$	Anteil[13]
[I]	Zustand von *P* nach *P.regs* retten	105/126/155	5.5 μs	10 %
[II]	`icb` für *iproc* erzeugen und initialisieren	122/252/339	11.2 μs	25 %
[III]	CPU-Interruptzyklus beenden	32/47/55	2.0 μs	5 %
[IV]	*iproc*-Scheduling	153/348/473	15.5 μs	35 %
[V].a	Umschalten zum *iproc* (Start)	59/95/130	4.4 μs	10 %
[V].b	Umschalten zum *iproc* (Wiederanlauf)	127/159/193	6.9 μs	(15 %)
[VI]	*iproc* beenden	72/124/172	5.7 μs	15 %
	$\sum$ - [V].a – Start eines *iproc* - [V].b – Wiederanlauf eines *iproc*	543/992/1324 611/1056/1387	44.3 μs 46.8 μs	

Abb.5.21 Zusammengefaßte Ausführungszeiten für Befehlssequenzen in *ictrl*

Ein typischer Durchlauf in *ictrl* erfordert auf der betrachteten Maschine und unter den genannten Randbedingungen einen ***absoluten Laufzeitaufwand*** von ca. 1150 Takten oder von ca. 46μs.

In der Tabelle ist auch die Verteilung des Aufwands für die sechs Phasen ablesbar. Den dominierenden Einfluß haben das Scheduling (Phase [IV]: 35 %) und die Verwaltung der `icb` (Phasen [II] und [VI]: 40 %). Diese Anteile charakterisieren auch den ***Mehraufwand*** gegenüber der herkömmlichen Art der Interruptbehandlung. Sie machen etwa 3/4 des gesamten Ablaufaufwands aus. Die verbleibenden Phasen [I], [III] und [V].a sind auch bei der traditionellen Interruptverarbeitung notwendig und stellen damit keinen Mehraufwand dar. Es ließe sich diskutieren, ob die hier in Betracht gezogene Optimierung des Starts gegenüber einem Wiederanlauf eines *iproc* in den Phasen [V].a bzw. [V].b im Gesamtaufwand (-5 %) so entscheidend ins Gewicht fällt. Es dominieren die Verwaltung der `icb` und der Aufwand für das Scheduling von *iproc's*.

In der Literatur findet man wegen des mit der Bestimmung verbundenen Aufwands nur wenige Zahlenangaben für die Interruptverarbeitung. Oftmals werden zur Charakterisierung des Laufzeitaufwands elementarer Systemmechanismen Werte angegeben, die auch Verarbeitungszeiten für Interrupts mit enthalten (z.B. (Null-) Systemrufe, (Null-) IPC-Zeiten, Reaktionszeiten auf externe Ereignisse, Prozeßumschaltzeiten u.a.), es erfolgt aber keine Aufschlüsselung mit konkreten Werten für die Interruptverarbeitung. Konzentriert man die Betrachtung dann noch auf die kleine Menge von Systemen mit einer prozeßorientierten Unterbrechungsverarbeitung, schränkt sich die Auswahl weiter ein. Vergleichende Untersuchungen sind deshalb und auch wegen der Unterschiedlichkeit der Maschinen und der angewandten Prinzipien nur schwer möglich.

Dennoch soll ein System diskutiert werden. In [KE95] sind Werte für die *Dispatch-Latenzzeit* für die thread-basierte Behandlung von Interrupts in *Solaris 2* angegeben. Diese Werte wurden unter Last mit der im μs-Bereich auflösenden Hardwareuhr einer *SparcStation 2* im realen Betrieb gemessen. Außerdem sind die Anzahl zusätzlicher Befehle und der Speicheraufwand angegeben. Als Latenzzeit wird hier das Zeitintervall vom Eintreffen eines niedrigst priorisierten Interruptsignals (hier: *clock*) bis hin zum Aktivieren eines *user realtime threads* auf dieses Ereignis hin definiert. Damit ist klar, daß damit mehr als die primäre Reaktion auf das Interruptsignal erfaßt wird. Die angegebenen Werte schwanken daher um den Bereich von etwa 300 μs.
Die Anzahl zusätzlicher *Sparc*-Maschinenbefehle wird mit 40 angegeben. Allein aus der Anzahl läßt sich jedoch nicht auf den Ablaufaufwand schließen: *"... passivating the interrupt thread involves flushing the register windows which may contain data from the pinned threads."* [KE95], S. 24. Bei

[13] Für die Aufwandsanteile wird der Start eines *iproc* als typischer Fall angenommen, der Wiederanlauf eines *iproc* in Phase [V].b verändert die Relationen nur unwesentlich.

Sparc-Prozessoren müssen (mit einem Maschinenbefehl) *alle* Registersätze (*register windows*) gerettet werden, da diese für Prozedur-Kontexte innerhalb von *Threads* benutzt werden[14].
Der Speicheraufwand ist dagegen durchaus mit *iproc's* vergleichbar. Es werden ebenfalls acht *Thread*-Einträge mit den zugehörigen "kleinen" statischen Stacks reserviert, woraus ein Speicheraufwand von ca. 8 kByte für die Verwaltung von *Interrupt-Threads* in *Solaris 2* entsteht.

Ein anderer Aspekt ist, daß Interrupts zwar durch eine spezielle Art von *Threads* behandelt werden, d.h., es gibt einen preallokierten *Thread*-Eintrag und einen eigenen Stack, sie unterliegen aber keinem Scheduling, sondern bei einem Interrupt wird ein *Interrupt-Thread* erzeugt, indem ein *Thread*-Eintrag zugewiesen und initialisiert wird, dieser *Thread* muß aber *sofort* ausgeführt werden. Es gibt in diesem Sinne kein Scheduling von *Interrupt-Threads*. Insofern lehnt sich die in [KE95] vorgestellte Lösung an die herkömmliche Art der Interruptbehandlung an. Bei verschachtelten Interrupts gilt dann ebenfalls das traditionelle Prinzip, daß nur höherpriorisierte Interrupts den Ablauf niedriger priorisierter *Interrupt-Threads* unterbrechen können.
Die Variante mit *iproc's* des *CHEOPS*-Kerns ist konsequenter, was jedoch bei der Architektur heutiger Prozessoren mit einem höheren Ablaufaufwand (vor allem für Scheduling) erkauft wird.

Die Ausführung sollte zeigen, daß es äußerst problematisch ist, Vergleiche zur Aufwandsbetrachtung für *iproc's* mit anderen Lösungen und Systemen zu ziehen. Lediglich in [KE95] wurden überhaupt Zahlenwerte für *Interrupt-Threads* für *SparcStation 2* angegeben. Diese sind aber an sich nicht mit der hier untersuchten Lösung für *iproc's* auf *CADMUS*-Workstation vergleichbar, da es sich zum einen um Maschinen handelt, zwischen denen ein technologischer Entwicklungszeitraum von ca. 10 Jahren liegt und weil sich auch die umgesetzten Konzepte in signifikanten Eigenschaften unterscheiden, z.B. der des *iproc*-Schedulings. In der anderen Literatur zu prozeßorientierter Behandlung von Interrupts [GK92, Hil93, SSC94, Kal90] sind dagegen keine Zahlenwerte enthalten, es werden lediglich die Konzepte vorgestellt.

Aus der Untersuchung zu *iproc's* folgt, daß die hauptsächlichen Anteile des Mehraufwands bei der Verwaltung der Datenstrukturen (→ `icb`) und beim Scheduling von *iproc's* liegen. Dieser Aufwand ist auch der Art und Weise der Unterbrechungsverarbeitung mit *iproc's* inhärent, d.h., er läßt sich durch feinere Optimierungen, vor allem beim Scheduling, sicher verringern, aber nicht um die Größenordnungen verbessern, damit auf *heutigen Prozessoren*[15] ein vergleichbarer Ablaufaufwand zu der heute gebräuchlichen Art der Interruptbehandlung resultiert. Diese Prämisse war von Anfang an klar. Optimierungen sind allerdings möglich und schließen dabei auch das Umfeld von *iproc's* ein. Beispiele sind:

- kein *iproc*-Scheduling für den häufigen Fall: $\mathcal{I}_{ksched}$ + ein *iproc*, da das Ergebnis feststeht,
- "kurze" *iproc*-Behandlungen nach *ictrl* verlagern, z.B. das Zählen der *clock* oder das Aktivieren eines *kproc* (Vermischen von Aufgaben: *ictrl* – *iproc's* zugunsten von Effizienz),
- das Umschalten zu $\mathcal{I}_{ksched}$ erfordert keine Umschaltung des kompletten Registersatzes, Aktivierungen können unterbleiben, wenn kein Re-Scheduling für *kproc's* ansteht, so daß aus *ictrl* unmittelbar zu dem vorher aktiven *[ika]proc* zurückgeschaltet werden könnte u.a.

Es schließt sich der Kreis zur eingangs getroffenen These, daß auf Grundlage althergebrachter Technologie und des gewohnten *bottom-up* Vorgehens nur die seit langem angewandten Prinzipien tragfähig sind, die ihrerseits die zugrundeliegende Technologie über lange Zeiträume bestimmen. Neue Technologien haben es schwer, eine Akzeptanzgrundlage zu finden. Die *Interrupt Threads* in *Solaris 2* sind in dieser Hinsicht ebenfalls ein Kompromiß. Der Ansatz des *CHEOPS*-Kerns mit *iproc's* geht konzeptionell weiter, weil das eingangs erklärte Ziel für eine anwendungsanpaßbare Systemarchitektur bestand und weil *iproc's* auch Vorteile bieten.

[14] Die Compiler erzeugen entsprechenden Code für die Umsetzung von Prozeduren auf *Sparc*-Prozessoren.
[15] Hier sind technische Grundprinzipien gemeint, die seit langem und auch für den veralteten *m68020* gelten.

5.3.6 Welche Vorteile bieten *iproc's* ?

Nachdem der Aufwand für die Unterbrechungsbehandlung mit *iproc's* untersucht wurde, sollen die vorteilhaften Eigenschaften von *iproc's* an dieser Stelle noch einmal zusammengefaßt werden.

- ***Vorteile struktureller Art:***

Der primäre Beweggrund für die Idee zu *iproc's* bestand in der eingangs formulierten Zielstellung der Homogenisierung der Abläufe und Strukturelemente in allen Schichten eines Systems. In Verbindung mit *iproc's* können aber noch andere Vorteile struktureller Art, für den Ablauf und auch softwaretechnischer Natur herausgestellt werden.

- Mit *iproc's* kann gezeigt werden, daß sich auch die Unterbrechungsverarbeitung in das hier vorgeschlagene Architekturkonzept von Schichten mit Instanzen als dominierenden Elementen einpaßt. Hinter diesem Prinzip steckt auch Dijkstra's Idee, semantisch unkorrelierte, aber gleichzeitig stattfindende Abläufe in Systemen in separate sequentielle Prozesse aufzuspalten, wobei hier Prozesse zu Instanzen erweitert werden (+ eigene Verarbeitungszustände, Lokalisierung und Kapselung einer Teilverarbeitung in einem autonomen Teilsystem), welche für die Schicht der Unterbrechungsverarbeitung als *iproc*-Instanzen ausgeprägt sind.
- Mit *iproc's* kann die Lücke geschlossen werden, welche in Betriebssystemen sonst für das Interruptsystem bestand. Der Strukturbruch zwischen der Interruptverarbeitung mit asynchron "aufgerufenen" Behandlungsroutinen und Prozeßinstanzen in höheren Schichten des Kerns bzw. im Anwendungsbereich wird aufgehoben.
- Durch Kapselung und Zentralisierung in der Infrastruktur ($\rightarrow$ *ictrl*) kann der anwendungsseitige Ablauf von Unterbrechungsbehandlungen ($\rightarrow$ *iproc's*) von der Ablaufsteuerung entkoppelt werden. Damit werden Steuerungsebenen in zweierlei Hinsicht eingeführt:
  - durch explizite Trennung der Verwaltung und Ablaufsteuerung vom eigentlichen Ablauf der Unterbrechungsbehandlungen in zwei Schichten: *ictrl* und *iproc's*,
  - durch explizite Trennung und Identifizierung verschiedener Unterbrechungsverarbeitungen in separaten, voneinander unabhängig ablaufenden Verarbeitungsinstanzen mit eigenen Verarbeitungs- und Steuerungszuständen.
- Durch die prinzipiell ähnliche Modellierung und Implementierung von Interruptverarbeitungsinstanzen zu Instanzen anderer Schichten des Systems ergeben sich auch prinzipiell ähnliche Ablaufprinzipien. Damit kann die als eine wichtige Zielstellung dieser Dissertation angestrebte Homogenisierung von Verarbeitungsprinzipien mit *iproc's* auch für die Schicht der Unterbrechungsverarbeitung als umsetzbar angesehen werden.

- ***Vorteile für den Ablauf:***

Strukturelle Vorteile sind für die Entwickler von Systemen wesentlich, und sie sind auch für diejenigen wichtig, die Anpassungen an Systemen vornehmen. Mit *iproc's* ergeben sich jedoch auch darüber hinausgehende vorteilhafte Eigenschaften für den Ablauf von Unterbrechungsverarbeitungen. Durch die vollständige Entkopplung der Abläufe von *iproc's* (speziell von der traditionellen *stacking order*) ergeben sich weitreichende Konsequenzen:

- Entkopplung des Ablaufs von Interruptbehandlungen von der Reihenfolge des Eintreffens der Interruptsignale und eine
- daraus folgt eine freie Gestaltbarkeit des Schedulings von *iproc's* durch das Betriebssystem ($\rightarrow$ *Anwendungsanpaßbarkeit*), wie es auch für andere Instanzen des Systems zutrifft.

- Interruptbehandlungen belegen nicht mehr das Interruptsystem des Prozessors, sie werden außerhalb des CPU-Interruptzyklus ausgeführt, so daß ein höherer Grad der Verfügbarkeit für die Annahme von Interruptsignalen entsteht.
- Die Annahme eines Interruptsignals wird vom Zwang der sofortigen Verarbeitung entkoppelt, so daß der *iproc*-Scheduler allein die Entscheidung für den Ablauf der nächsten Instanz treffen kann, *iproc's* können verzögert ablaufen.
- Instanzen der Schicht *iproc's* sind wie andere Instanzen selbst jederzeit unterbrechbar, und das Prinzip von *iproc's* läßt sich auch auf die Signalisierung in Multiprozessorsystemen übertragen, weil der unabhängige (→ *parallele*) Ablauf von Anfang an ein wesentliches Merkmal der Modellierung und Gestaltung von (*iproc*-) Instanzen war.
- Interruptbehandlungen mit *iproc's* unterliegen damit auch nicht mehr dem Zwang, "kurz" sein zu müssen, um das Interruptsystem nicht zu lange zu belegen. Daraus folgt auch, daß die für "umfangreichere" Reaktionen auf Interrupts in Betriebssystemen anzutreffende Trennung in eine *first-* und *second-level* Behandlung hier klar den Aufgaben der Schichten *ictrl* und *iproc's* zugeordnet werden kann und nicht mehr nach zufälligen Gesichtspunkten der Implementierung erfolgt.
- Es kann auf das Sperren der Annahme weiterer Interrupts durch die CPU (→ `DI`) für *iproc's* verzichtet werden, da statt dessen die Umschaltsteuerung in *ictrl* gesperrt werden kann (→ `locked`), so daß auch innerhalb kritischer Abschnitte in *iproc's* zumindest die Annahme weiterer Interruptsignale gesichert bleibt, für welche *iproc's* erzeugt, aber erst verzögert nach dem Ende des kritischen Abschnitts verarbeitet werden können. Damit vermindert sich das Risiko des Verlusts von Interruptsignalen bei gesperrten Interrupts.
  Die in Prozessoren fehlende Speicherung mehrfacher Interruptsignale wird durch Erzeugen und Speichern von `icb` in der Bereit-Liste in *ictrl* nachgebildet. Dieser Aspekt wird insbesondere auch in [KE95] als wesentlich herausgestellt.

Mit dem letzten Punkt ist aber noch nicht das Problem der Prioritätsumkehr (*priority inversion*) gelöst, welches darin besteht, daß bei gesperrten Umschaltungen in *ictrl* zwar Unterbrechungssignale jederzeit angenommen und *iproc's* in die Bereit-Liste eingeordnet werden, aber erst frühestens nach Freigabe der Sperre ausgeführt werden können. Ein niedrigpriorisierter *iproc* kann damit einen zwischenzeitlich eingetroffenen, höherpriorisierten *iproc* verzögern.
Der Ausweg für dieses Problem ist die Aufgabe des globalen Sperrens von Umschaltungen in *ictrl* als Koordinierungsmittel. Statt dessen müßten kritische Abschnitte bezüglich gemeinsamer Ressourcen unterschieden werden, so daß die Sperre nicht global, sondern nur selektiv für die betroffenen Ressourcen wirkt. Dies ist aber wiederum mit einem höheren Verwaltungsaufwand in *ictrl* verbunden. Aus diesem Grunde wurde die hier verwendete einfache Lösung mit einer globalen Umschaltsperre in *ictrl* gewählt, da das Problem der Prioritätsumkehr vorwiegend bei Echtzeitsystemen eine Rolle spielt, aber für den *CHEOPS*-Kern nicht im Mittelpunkt stand und damit der Aufwand in *ictrl* auf ein Minimum beschränkt werden konnte.
Es ist aber eine wesentliche Intention der *CHEOPS*-Architektur, eine diesbezügliche Umstellung in der Schicht *ictrl* bei Bedarf vornehmen zu können.

- ***softwaretechnologische Vorzüge:***

Nicht zuletzt ist die Frage nach Ordnung und Struktur auch eine softwaretechnologische Frage. Hierbei geht es nicht nur um die effektive Herstellung von Software anhand geeigneter Beschreibungssprachen, sondern die Softwaretechnologie wird auch wesentlich von den vorherrschenden Ablaufeigenschaften und dem zugrundeliegenden Ablaufmodell beeinflußt. Gerade bei der Unterbrechungsverarbeitung ist es mit den Fähigkeiten heute angewandter Programmiersprachen nicht

möglich, Wettlaufbedingungen und kritische Abschnitte zu erfassen und adäquat zu beschreiben. Die Folge ist, daß es hinter dem Programmiermodell noch ein zweites Ablaufmodell geben muß, mit dem diese Effekte dann erfaßbar sind. In heutigen Betriebssystemen ist dieses Modell für den Bereich der Unterbrechungsverarbeitung im Prinzip nicht vorhanden, und die Mittel für Koordination reichen über das Sperren von Interrupts an verschiedenen Stellen im Code nicht hinaus. Insofern bieten *iproc's* dieses fehlende Ablaufmodell in Form des aus dem Anwendungsbereich und aus übergeordneten Betriebssystemschichten bekannten ***Client-Server-Modells*** auf Basis elementarer *iproc*-Instanzen auch für die Unterbrechungsverarbeitung.

- Das *Client-Server-Modell* wird durch *iproc's* in einer speziellen, sehr elementaren Ausprägung auch für die Unterbrechungsverarbeitung anwendbar.
- Verarbeitungen werden durch eine explizite Zuordnung zu *iproc*-Funktionseinheiten identifizierbar und in ihrem Ablauf voneinander entkoppelt.
- Die Programmiermethodik ändert sich, indem der Maßstab nicht mehr die Herstellung einer Behandlungsroutine (→ *Prozedur*) ist, sondern der Ablauf als *iproc*-Instanz im Vordergrund steht (*Programm* → *Prozeß*). Das *Client-Server-Modell* bietet genau dieses hinter dem Programmiermodell stehende Ablaufmodell. Die identifizierbaren Einheiten der *Modellierung* und des *Ablaufs* werden mit *iproc*-Instanzen gleichgestellt.
- Wettlaufbedingungen und kritische Abschnitte können explizit eingeordnet werden, so daß keine kritischen Abschnitte in Programmen konkurrierend zueinander ausgeführter Behandlungs- bzw. Kern-Routinen lokalisiert werden müssen, sondern statt dessen gemeinsame Ressourcen in *Monitoren* gekapselt werden können. Auch das ist ein wichtiges softwaretechnisches Prinzip, wenn man die Wirkungen von Unterbrechungen als prozeßorientiertes Geschehen erkennt und mit einem entsprechenden Ablaufmodell behandelt.
- Aus der Zentralisierung des Ablaufmechanismus in *ictrl* folgt auch eine einfachere Softwarestruktur bei der Herstellung der als *iproc's* ablaufenden Behandlungsroutinen.
- Die Interaktion mit anderen Instanzen des Systems erfolgt explizit durch Signale oder Nachrichten. Schwer zu durchschauende implizite Seiteneffekte in parallel zueinander ausgeführten Behandlungsroutinen (→ *"hidden interactions"*) können vermieden werden.

Auch in [KE95] werden die Vorteile von *Interrupt Threads* in *Solaris 2* herausgestellt. Da sie im Prinzip auch für *iproc's* gelten, soll diese Einschätzung hier mit angegeben werden.

"Implementing interrupts as asynchronously generated threads yields the following advantages:

1) Single model of asynchrony and synchronization.
   Code is written using only the normal thread synchronization primitives to protect against all forms of asynchrony: from other processors, preemption, or interrupts.
2) Eliminate time to enable/disable interrupts from mutexes.
3) Reduces interrupt lock out time.
4) Code doesn't need to know the highest interrupt level that calls it.
5) Use priority inheritance instead of maximum calling interrupt priority.
   The use of thread synchronization primitives allows the application of techniques such as priority inheritance. This mechanism prevents priority inversions ...
6) Adaptive mutexes instead of explicitly spinning or explicitly blocking mutexes.
   There is little need for the code to explicitly request spin blocking or sleep blocking in synchronization primitives since they are only used by threads. This allows optimizations like adaptive mutexes ...
7) Low overhead.
8) Interrupt handlers see a less restricted environment." [KE95], S. 25

5.4 Hierarchisches Prozessor-Scheduling über Schichten

Aus der vorgeschlagenen Schichtenarchitektur ergibt sich unmittelbar ein hierarchisches Schema für das Prozessor-Scheduling. Pro Schicht gibt es mindestens einen Prozessor-Scheduler. Für die Schicht der *iproc's* ist dieser Scheduler[16] Teil von *ictrl*. In der Schicht *iproc's* bildet dann $\mathcal{I}_{ksched}$ den Scheduler für die übergeordneten *kproc's*, und in der Schicht *kproc's* sind dann die Scheduler für die Anwendungsinstanzen *aproc's* angesiedelt. Es kann durchaus mehrere Scheduler in dieser Schicht geben, um beispielsweise verschiedenen Anwendungsklassen nach unterschiedlichen Scheduling-Strategien zu behandeln. In dem in Kapitel 4 entwickelten Architekturkonzept wurde für diesen Zweck das Mittel der Instanzbereiche $\mathcal{IB}$ eingeführt. In Abbildung 5.22 ist diese Art Scheduling dargestellt. Das Prozessor-Scheduling ist eine wesentliche Aufgabe der Infrastruktur, so daß sich auch aus der Beziehung zwischen jeweiligen Infrastrukturschichten eine hierarchische Struktur ergibt. Systeme können sich aber darin unterscheiden, ob das Scheduling prozessor- oder schichtbezogen oder anhand von Instanzbereichen erfolgt. Bei prozessororientiertem Scheduling gibt es für jeden Prozessor einen Scheduler, welcher für die Instanzen aller übergeordneten Schichten die Prozessorzuteilung umsetzt. Dieser Prozessor-Scheduler muß dann auch alle Strategien für unterschiedliche Anforderungen in höheren Systemschichten kennen und umsetzen. Für *Unix System V Release 4* werden beispielsweise die Scheduling-Klassen für Echtzeit-Threads, reguläre, zeitscheibenbasierte Prozesse im Anwendungsbereich und System-Threads unterschieden [Jus93, GC94]. Es gibt aber genau einen zentralen Scheduler für alle Instanzen. Ein anderer Weg, der auch für den *CHEOPS*-Kern beschritten wird, ist der eines hierarchischen Schedulings über mehrere Scheduler. In der Abbildung gibt es beispielsweise zwei *aproc*-Scheduler für zwei Instanzbereiche in der Anwendungsschicht. Die *aproc*-Scheduler sind als reguläre *kproc*-Instanzen implementiert. Die Einflußsphären der jeweiligen Scheduler sind in der Abbildung durch unterschiedliche Schattierungen kenntlich gemacht. Die Lösung mit hierarchischen Schedulern hat den Vorteil, daß unterschiedliche Scheduling-Strategien auch unterschiedlichen Systemkomponenten in der jeweils vorgesehenen Systemschicht zugeordnet werden können. Das Scheduling steht nicht außerhalb der Schichtenarchitektur. Scheduler-Instanzen werden in der jeweiligen Infrastrukturschicht als reguläre Instanzen behandelt. Sie können nach den dort geltenden Eigenschaften gestaltet, geändert und statisch oder dynamisch plaziert werden. Ihre Aufgabe ist das Scheduling des Prozessors für die Instanzen eines Bereichs in der übergeordneten Schicht. Damit wird eine Aufgabenteilung und eine Dezentralisierung von Aufgaben in einem System erreicht. Die zu betrachtenden Komponenten sind kleiner, leichter herstellbar und betreffen nur einen Teilbereich des Systems.

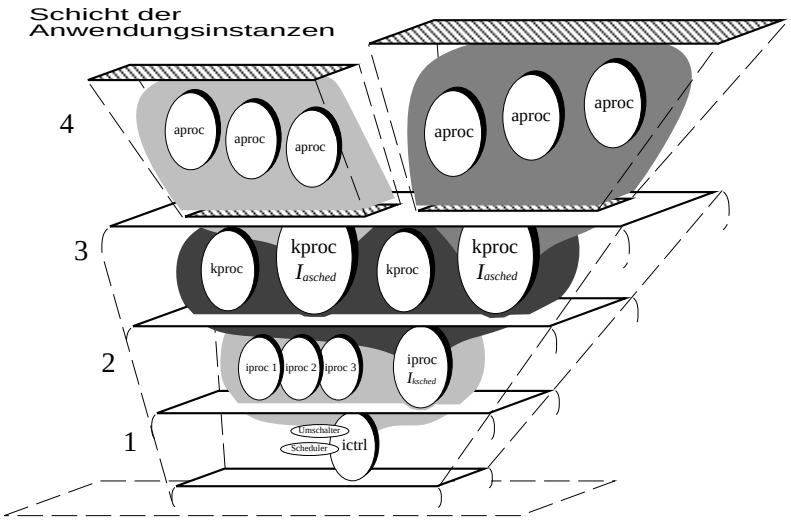


Abb.5.22 Hierarchisches Prozessor-Scheduling

Diese Vorzüge haben einen Preis, der sich wiederum in einem erhöhten Ablaufaufwand bei Systemen heutiger Bauart niederschlägt. Die wesentliche Ursache dafür ist, daß Scheduler selbst auf einem hierarchischen Weg den Prozessor zugeteilt bekommen müssen, so daß bis zur Auswahl

[16] Wenn im folgenden von Schedulern bzw. von Scheduling gesprochen wird, soll sich das primär auf die Zuteilung des Betriebsmittels Prozessor für den Ablauf von Instanzen beziehen.

der letztendlich auszuführenden Instanz u.U. mehrere Scheduler-Instanzen beteiligt sind, die im Vorfeld als reguläre Instanzen ablaufen müssen. Jedes Interruptsignal bewirkt eine Umschaltung zu dem am höchsten stehenden Scheduler in *ictrl*. Es wird ein *iproc* erzeugt, ausgewählt und ausgeführt. Anschließend wird $\mathcal{I}_{ksched}$ ausgeführt, um den Prozessor einer Instanz in der Schicht *kproc's* zuzuteilen und zu dieser umzuschalten. Dies kann wiederum eine Scheduler-Instanz sein, welche ihrerseits diesen Vorgang für die nächsthöhere Schicht oder den übergeordneten Instanzbereich wiederholt. Ein solches Vorgehen ist auf einem Einprozessorsystem unpraktikabel.
Aus diesem Grund werden heute auch kaum hierarchische Scheduler in Betriebssystemen eingesetzt, sondern alle Scheduling-Aufgaben werden in einem Prozessor-Scheduler konzentriert. Dieses Prinzip ist seit langem bekannt, und es muß daher nicht für den *CHEOPS*-Kern nachempfunden werden, dessen Ziel es ist, alternative Konzepte und Wirkprinzipien zu untersuchen.

Es gibt drei Wege, einen Scheduler für einen Bereich von Instanzen zu aktivieren, damit ein Re-Scheduling erfolgen und eine andere Instanz den Prozessor zugeteilt bekommen kann:

(1.) beim *Ende* oder *Blockieren* einer übergeordneten Instanz ($\rightarrow$ *Abgabe "von oben"*),

(2.) bei einem synchronen *Dienstaufruf an die Infrastruktur* ($\rightarrow$ *Abgabe "von oben"*),

(3.) bei einem Interrupt, wenn der Prozessor von unteren Schichten wieder nach oben zugeteilt wird ($\rightarrow$ *Zuteilung "von unten"*).

Die Aussagen, welche hier zum Prozessor-Scheduling getroffen werden, lassen sich in geeigneter Weise auch auf hierarchische Scheduling-Prinzipien für andere Betriebsmittel übertragen.

Für die Verarbeitung von Interrupts wird tatsächlich nur das Betriebsmittel Prozessor zwischen den beteiligten Instanzen $P \rightleftharpoons R$ umgeschaltet. Andere Betriebsmittel sind nicht betroffen. Gleich, in welcher Schicht die unterbrochene Instanz P abläuft, ihr bleiben dort alle anderen Betriebsmittel zugeteilt, d.h., die unterbrochene Instanz P bleibt in ihrer Schicht *"pseudo-aktiv"*. Die "vertikalen" Umschaltungen des Prozessors bei Unterbrechungen haben somit keinen direkten Einfluß auf die sonstige Betriebsmittelzuteilung, so daß auch keine anderen Verwaltungsinstanzen einbezogen werden müssen. Der Globalzustand einer unterbrochenen Instanz ändert sich nicht. Veranlaßt aber das Interruptsignal auch eine horizontale Umschaltung, muß der für die betroffene Instanz verantwortliche Scheduler aktiviert werden. Diese Unterscheidung wird in $\mathcal{I}_{ksched}$ getroffen, damit die verantwortliche Scheduler- und Umschaltinstanz in der nächsthöheren Schicht aktiviert wird. Dort wird dann der notwendige Wechsel von Zuständen und zugeteilten Betriebsmitteln für die Umschaltung ausgeführt.

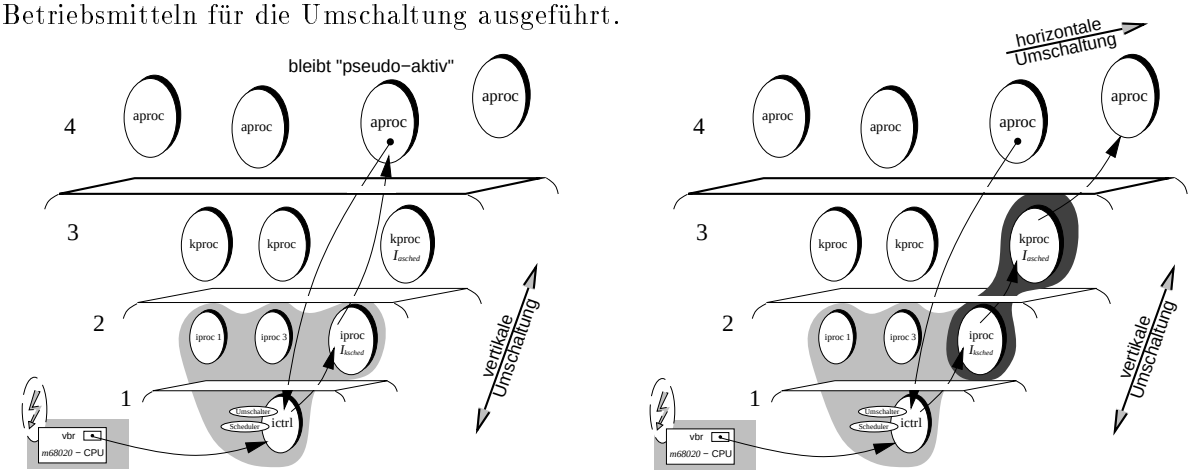


Abb.5.23 Vertikales und horizontales Umschalten zwischen Instanzen

Man kann zusammenfassen, daß "vertikale" Umschaltungen für Instanzen einen kurzzeitigen Entzug des Betriebsmittels Prozessor durch das Interruptsystem dieses Prozessors bedeuten, hingegen "horizontale" Umschaltungen durch Softwareinstanzen einer Infrastrukturschicht vorgenommen werden. Dabei werden alle dafür notwendigen Betriebsmittel und Zustände erfaßt.

Der hauptsächliche Mehraufwand resultiert aus den notwendigen Prozessorumschaltungen zum Ablauf der verschiedenen beteiligten Instanzen für das Scheduling von Betriebsmitteln, speziell hier für das Betriebsmittel Prozessor. Dieser Aufwand läßt sich bis zu einem gewissen Grade reduzieren, aber nicht eliminieren. Dies wäre theoretisch erst dann gegeben, wenn das Betriebsmittel Prozessor keine knappe Ressource mehr wäre, sondern im Idealfall jeder Instanz ein eigener Prozessor zur Verfügung stünde. Dieser Sachverhalt wurde auch schon für die Interruptverarbeitung in Betriebssystemen festgestellt, bei der auch nur zum Zweck der Zustellung einer Signalisierung ein ablaufender Prozeß unterbrochen wurde und zwei Prozessorumschaltungen ausgeführt wurden ($P \rightleftharpoons R$). Diese Umschaltungen könnten entfallen, wenn es für die Zustellung der über Interrupts angezeigten Signale einen eigenen Prozessor gäbe. Das Signal wäre parallel zu dem Ablauf von P zustellbar. Analog könnte man sich vorstellen, daß auch alle Scheduler einer Hierarchie von Schichten mit einem eigenen Prozessor ablaufen könnten, so daß deren Ablauf von dem der anderen Instanzen entkoppelt wird. Dies wäre eine alternative Gestaltungsmöglichkeit für ein Multiprozessorsystem. Heute werden mehrere Prozessoren primär symmetrisch, d.h. "horizontal" in einem System eingesetzt. Sie sollen vor allem den parallelen Ablauf von Anwendungen erlauben. Da aber Anwendungen nicht nur parallel zueinander, sondern auch parallel zum Betriebssystem ablaufen, ließe sich auch in dieser Hinsicht eine Prozessoranordnung "vertikal" zu den Schichten vornehmen, um eine Entkopplung der Verwaltungsaufgaben des Betriebssystems vom Ablauf der Anwendungsinstanzen zu erreichen. Je mehr Prozessoren zur Verfügung stehen, desto feiner könnte die Zuordnung von Prozessoren zu Gruppierungen von Instanzen erfolgen. In der Literatur wird diese aufgabenbezogene Zuordnung von Prozessoren zu Instanzen auch als *Prozessor-Affinität* bezeichnet [FS96].

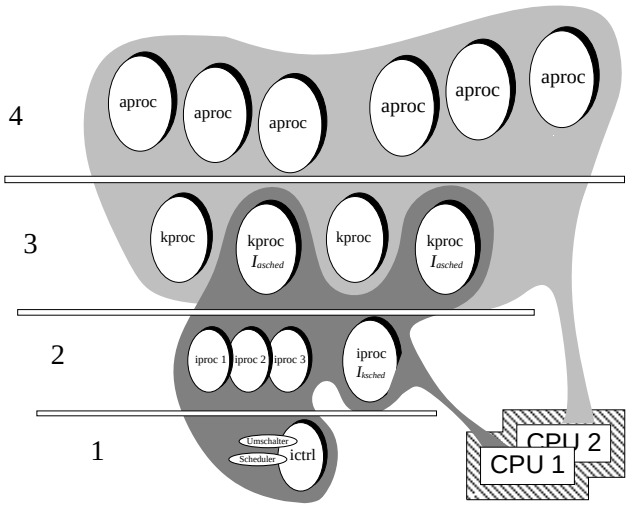


Abb.5.24 Vertikale Zuordnung zweier Prozessoren

In Abbildung 5.24 werden die Prozessoren nicht gleichberechtigt, d.h. symmetrisch zwischen Anwendungsinstanzen zugeteilt, sondern sind asymmetrisch Instanzen der Infrastruktur bzw. der übergeordneten Schicht zugeordnet. Auf diese Weise kann beispielsweise eine Unterbrechung des Ablaufs einer Anwendungsinstanz vermieden werden, wenn das durch einen Interrupt angezeigte Signal von dem anderen Prozessor verarbeitet werden kann. Die Wirkung dieses Signals kann dann natürlich dazu führen, daß sich auch für den Ablauf dieser Anwendungsinstanz Wirkungen ergeben. Ein Prozessor ($\rightarrow$ *CPU 1*) könnte die dafür nötigen Verwaltungsaufgaben ausführen und dann durch ein Interruptsignal eine Umschaltung auf dem zweiten Prozessor veranlassen. Die Idee eines solchen Arbeitsprinzips ist es, die Verwaltungsaufgaben des Betriebssystems weitgehend vom Ablauf der Anwendungsinstanzen zu entkoppeln und parallel auszuführen. Das im Bild mit zwei Prozessoren dargestellte Prinzip läßt sich auf n Prozessoren verallgemeinern, so daß die weiteren Prozessoren für den parallelen Ablauf von Anwendungsinstanzen nutzbar sind.

Dieses Prinzip würde es erlauben, den mit einer hierarchischen Betriebsmittelvergabe verbundenen Mehraufwand durch häufige Umschaltungen zwischen vielen beteiligten Instanzen nicht auf die Anwendungsebene auswirken zu lassen. Es ist zugleich eine interessante Alternative zur heute allgemein angewandten symmetrischen Betriebsweise[17] von Prozessoren in einem Multi-

[17] Darunter soll in diesem Zusammenhang die gleichartige Nutzung von Prozessoren in einem Multiprozessorsystem verstanden werden, nicht das symmetrische Multiprocessing mit gemeinsamen Speicher.

prozessorsystem. Dies erfordert aber eine entsprechend ausgelegte Hardware-Infrastruktur, und die Vor- und Nachteile müßten praktisch näher untersucht werden. An dieser Stelle sollte lediglich auf die prinzipielle Gestaltungsmöglichkeit eines solchen Systems hingewiesen werden, um das Argument des hohen Ablaufaufwands hierarchischer Betriebsmittelzuteiler zu relativieren.

Weitere Argumente für dieses Prinzip können auch aus der aktuellen Literatur angegeben werden. Auf der *OSDI'96* [OSDI96] wurden einige Vorschläge und Systeme mit hierarchischen Zuteilungsstrategien vorgestellt, z.B. in *"CPU Inheritance Scheduling"* [FS96]: *"This paper presents CPU inheritance scheduling, a novel processor scheduling framework in which arbitrary threads can act as schedulers for other threads"*. Diese Beziehung gilt rekursiv und ist dem hier gezeigten Prinzip ähnlich, außer daß "Scheduler-Threads" nicht durch eine Infrastrukturschicht abgegrenzt sind. Es entsteht eine Scheduling-Hierarchie für Threads. Dieser Vorschlag bietet eine Alternative zu der heute in Systemen üblichen Methode fest eingebauter Verfahren in zentralen (Prozessor-) Schedulern, wenngleich diese durch Einführung von Scheduling-Klassen bereits etwas flexibilisiert wurden. Auch ein zweiter Beitrag behandelt unmittelbar diesen Themenkomplex: *"A Hierarchical CPU Scheduler for Multimedia Operating Systems"* [GGV96]. Weitere Beispiele könnten folgen.

5.5 Weitere Implementierungsaspekte

Für den *CHEOPS*-Kern wurde in dieser Dissertationsschrift die Implementierung der Schichten *ictrl* und *iproc's* sehr detailliert beschrieben, begründet und bewertet. Dieses Merkmal ist auch eine markante Eigenschaft des *CHEOPS*-Kerns, welche diesen Kern von anderen unterscheidet. Der *CHEOPS*-Kern umfaßt aber mehr Komponenten, und es werden auch im Rahmen weiterer Dissertations- und Forschungsarbeiten Zielstellungen behandelt, die nicht Gegenstand dieser Dissertation sind. Auf sie wurde eingangs bei der Einordnung und Abgrenzung dieser Arbeit verwiesen. Insofern ist die Entwicklung des *CHEOPS*-Kerns nicht abgeschlossen, sie wird anhand der Ziele weiterführender Arbeiten fortgesetzt. Das in dieser Dissertation entwickelte Architekturkonzept und die implementierte Plattform bilden den Ausgangspunkt für diese Arbeiten.
An dieser Stelle sollen deshalb nur noch die Dinge kurz angesprochen werden, die im Rahmen dieser Dissertation und im Umfeld angesiedelter studentischer Arbeiten ausgeführt wurden.

5.5.1 Werkzeuge für eine stand-alone Umgebung

Die Arbeit in einem stand-alone Umfeld ist aus vielerlei Gründen kompliziert. Das schwerwiegendste Problem ist, daß ein ablauffähiges Initialsystem nicht *bottom-up* entwickelt werden kann, sondern als bootfähiges, funktionierendes Gesamtsystem herzustellen ist, welches bereits alle vitalen Aufgaben der Hardwaresteuerung, der Steuerung des eigenen Ablaufs und der Ein- und Ausgabe erfüllen muß. Das betrifft das Interruptsystem, die Herstellung und Steuerung von Prozessen, den Kern-Adreßraum, die Terminalsteuerung u.v.a. Damit wird der gesamte "untere" Kern-Bereich erfaßt, in dem ein isoliertes Entwickeln und Testen kaum möglich ist.

Damit eine Interaktion mit einem solchen System erfolgen kann, muß die Terminalsteuerung arbeiten und eine die Eingaben interpretierende Instanz existieren. Diese Instanz wurde für den *CHEOPS*-Kern *sash – stand-alone shell* genannt. Damit das System nach dem Booten definiert arbeitet, muß die Hard- und Software vom *Bootprozeß bootp* initialisiert werden. Der *CHEOPS*-Kern ist in *C++* implementiert, so daß stand-alone Versionen für die benötigten Funktionen aus *C++*-Bibliotheken herzustellen waren (*libsa*.a). Weiterhin mußten Modifikationen am Laufzeitsystem vorgenommen werden, so daß z.B. die Konstruktoren für die Initialisierungen statischer Datenobjekte in *ictrl* bzw. *iproc's*[18] vom Bootprozeß ausgeführt werden konnten.

[18] z.B. die Verwaltungsdatenstrukturen für Instanzen `icbs` und `kcbs`, das einfache Nachrichtensystem `kmsg` u.a.

- ***Stand-Alone Shell*** – *sash*

Die *stand-alone shell* stellt eine einfache Dialog- und Test-Funktionalität für den *CHEOPS*-Kern bereit. Sie wurde mit dem Ziel entworfen und implementiert, auf einer "minimalen" Infrastruktur ablauffähig zu sein, um sie bereits in einem frühen Entwicklungsstadium des *CHEOPS*-Kerns einsetzen zu können. Mit dem Fortschreiten der Entwicklung und den daraus resultierenden besseren Infrastruktureigenschaften sollte sie in andere Ausführungsumgebungen "mitwachsen" können. Konkret bedeutete dies, daß anfangs lediglich die Schichten *ictrl* und *iproc's* arbeitsfähig waren. Die *stand-alone shell* mußte in diesem Umfeld mit einer einfachen Terminalsteuerung als fest in den Kern gebundene "Dialogfunktion" auskommen. Mit der Verfügbarkeit von *kproc's* konnte die *stand-alone shell* in diese Umgebung als *kproc*-Instanz migrieren. Die Anforderungen der *stand-alone shell* an die Umgebung bzw. Infrastruktur sind minimal. Es genügt eine Anbindung an die Terminal-Ein- bzw. Ausgabe und der von ihr benötigte statische Speicher.

Die Funktionalität der *stand-alone shell* und die Implementation sind in [Grau96b, Grau94c] beschrieben. Der Funktionsumfang der *stand-alone shell* ist natürlich eingeschränkt. Er kann grob mit drei Funktionsklassen umschrieben werden:

- Inspektion und Änderung der Werten von Datenstrukturen des *CHEOPS*-Kernspeichers,
- "Aufrufbarkeit" von Kern-Funktionen über symbolische Namen mit Parameterübergabe,
- Ausführbarkeit von Test-Skripts, welche während des Ablaufs eingegeben oder als Text in Form von Zeichenkettenkonstanten mit in die *sash* eingebunden werden.

Damit werden natürlich nicht die Anforderungen an ein vollwertiges Entwicklungs- oder Testwerkzeug erfüllt, wie diese etwa aus dem Anwendungsbereich bekannt sind. Für die Implementierung des *CHEOPS*-Kerns ist *sash* jedoch ein unverzichtbares Werkzeug, welches unter den Randbedingungen von Aufwand und Zweckmäßigkeit entworfen und implementiert wurde. Die Interpreter-Komponenten wurden mit Hilfe von *lex* und *yacc* hergestellt und sind erweiterbar.

- ***Der Bootprozeß*** – *bootp*

Die Aufgabe des Bootprozesses ist es, die initial benötigten Komponenten der Hard- und Software mit definierten Werten zu initialisieren. Das betrifft vor allem die Komponenten des Interruptsystems (Interruptsteuerung des *m68020*, die Vektoren `_cpuv[ ]` und `_irdv[ ]`, *ictrl* u.a.), der Adreßumsetzung des *m68020* für den Kern-Adreßraum und alle weiter benötigten Verwaltungsdatenstrukturen für Instanzen bzw. für die Interaktions-Monitore (vgl. Abschnitt 5.2.2). Desweiteren muß für die *CADMUS*-Workstation die Kommunikation mit dem Ein- bzw. Ausgabesystem *icckernel* auf dem zweiten Prozessor initialisiert und etabliert werden, über die auch bereits die elementare Terminalsteuerung erfolgt. Zu diesem Zweck muß der gemeinsame Speicher zwischen dem *CHEOPS*-Kern und dem *icckernel* eingerichtet werden, und die entsprechenden Behandlungen für Interrupts müssen vom Interruptsystem anhand des Kommunikationsprotokolls mit dem *icckernel* korrekt ausgeführt werden.

- ***Stand-Alone Bibliothek*** – *libsa.a* ***und*** ***C++-Laufzeitsystem*** – *saccrts.a*

Diese Bibliotheken sind notwendig, da die Standard-Bibliotheken für *C++* natürlich nicht für den stand-alone Einsatz vorgesehen sind, sondern sich auf eine vorausgesetzte Betriebssystemschnittstelle abstützen. Natürlich wurden nicht alle Funktionen dieser Bibliotheken an standalone Bedingungen angepaßt, sondern nur die jeweils notwendigen Funktionen nach *libsa.a* übernommen. Diese beziehen sich vor allem auf Konvertierungsfunktionen für die Terminalein- bzw. ausgabe, auf Zeichenkettenfunktionen für die Symboltabelle u.a.). Als Übersetzungssystem wurde der *GNU-C++*-Compiler *gcc2.3.2* mit den dazugehörigen Bibliotheken verwendet.
Die Modifikation des *C++*-Laufzeitsystems bezog sich vor allem auf die Aufrufbarkeit der Konstruktor-Methoden statischer Datenobjekte durch den Bootprozeß.

5.5.2 Im Umfeld angesiedelte Arbeiten

Darüber hinaus waren und sind studentische Arbeiten in Form von Projekt-, Studien- und Diplomarbeiten mit der Weiterentwicklung und Vervollständigung des *CHEOPS*-Kerns befaßt.

Im Umfeld des *CHEOPS*-Kerns wurden vom Autor unmittelbar die folgenden Arbeiten betreut. Zur Ansteuerung der Platte mußte eine *"Integration des Plattentreibers in den CHEOPS-Kern"* [Hart96] (Projektarbeit) erfolgen (→ *kproc*-Instanz `kdisk`, vgl. Abschnitt 5.2.2), auf welche aufbauend ein einfaches Dateisystem (→ `kcfs`) hergestellt wurde [WG95].
In der Projektarbeit *"Architektur der Speicherverwaltung"* [Fran96a] wurde ein Konzept erarbeitet, mit welchem die Ausführungseigenschaft getrennter Adreßräume für eine Variante von *aproc*-Instanzen in der dazugehörigen Infrastrukturschicht als *kproc*-Instanz umgesetzt werden kann (→ `amm`). Das entstandene Konzept wurde als Prototyp auf Basis von *Unix* implementiert und getestet. Es soll als Vorlage für die Umsetzung in den *CHEOPS*-Kern dienen.
Das *"Dynamische Laden und Starten von kproc-Instanzen für den CHEOPS-Kern"* [Hart97] soll in einer Diplomarbeit erlaubt werden, so daß auch während der Laufzeit Typbeschreibungen für neuartige Instanzen in den *CHEOPS*-Kern eingebracht werden können. Diese Arbeit bezieht sich vor allem auf die Herstellung des Typ-Managers `ktyp` und der Erzeugungsinstanz `knewi`.

Nicht vom Autor betreute, aber ebenfalls im Umfeld des *CHEOPS*-Kerns angesiedelte Arbeiten waren bzw. sind: *"Dynamisches typsicheres Linken von C++-Programmen"* [Eul96] (Diplomarbeit), *"Basisdienste für die Interobjektkommunikation in einem objektorientierten, verteilten Betriebssystem"* [Bec96] (Studienarbeit) und die *"Entwicklung des Kommunikationsteilsystems für ein objektorientiertes, verteiltes Betriebssystem"* [Bec97] (Diplomarbeit).

5.6 Zusammenfassung und Fazit

In diesem Kapitel wurde die Spezialisierung des allgemeinen Architekturkonzepts aus Kapitel 4 als Grundlage für den Entwurf und die Implementierung des *CHEOPS*-Kerns vorgestellt. Es konnte gezeigt werden, daß sich ausgehend von der experimentellen Zielstellung des *CHEOPS*-Kerns eine geeignete Ausprägungsvariante bestehend aus vier Schichten mit jeweiligen Ausführungseigenschaften ableiten läßt: *ictrl*, *iproc's*, *kproc's* und *aproc's*.

Die spezialisierte Architektur bildet den Hintergrund für eine Reihe von Entwurfsentscheidungen über Struktur und Eigenschaften des *CHEOPS*-Kerns. Gemäß dem Anliegen dieser Arbeit wurden dabei nicht die sonst üblicherweise für Betriebssysteme verfolgten Ziele (Effizienz, Kompatibilität u.a.) zugrunde gelegt, sondern es wurde das Anliegen nach Anwendungsanpaßbarkeit und einem durchgängigen Architekturkonzept in den Vordergrund gestellt. Ein zentraler Punkt war auch die praktische Umsetzbarkeit dieses Architekturvorschlags anhand der Implementierung eines Betriebssystem-Kerns. Mit dem *CHEOPS*-Kern wurde dieser Versuch aus Gründen einer Vorgeschichte auf zwei Hardwareplattformen unternommen. Im Nachhinein muß eingeschätzt werden, daß eine Konzentration auf nur eine Implementierungsplattform von Vorteil gewesen wäre. Die Arbeiten auf der *CADMUS*-Workstation wurden mit dem hier beschriebenen Stand eingestellt. Alle weiterführenden Arbeiten und Untersuchungen finden auf der PC-Basis statt.

Der Aufbau eines Betriebssystem-Kerns beginnt mit den untersten Schichten, so daß auch diese den Schwerpunkt der praktischen Implementierung in dieser Dissertation und speziell in diesem Kapitel darstellen. In den Mittelpunkt der Betrachtung wurde dabei die besondere Art der Behandlung von Unterbrechungen mit Hilfe von *iproc*-Instanzen und die dafür notwendige Softwareinfrastruktur *ictrl* gestellt. Es wurden die Ursachen für die Probleme der Interruptbehandlung nach der heute üblichen Methode identifiziert und durch eine entsprechende neuartige Gestaltung aufgehoben. Die Entscheidung für diese Art der Umsetzung wurde primär aus der

Architektur abgeleitet, aber es wurden auch besondere Ablaufeigenschaften herausgestellt, welche sich durch die hier gezeigte Lösung ergeben und welche sich von der sonst angewandten Technik der Behandlung von Interrupts wesentlich unterscheiden. Die Umsetzung der Behandlung von Unterbrechungen über *iproc's* erfaßt dabei auch das Wesen von Unterbrechungen in einer verarbeitungsbezogenen, prozeßorientierten Weise. Die in Prozessoren heute festgelegte Ablaufreihenfolge für Interruptbehandlungen wird durch das Prinzip der *iproc's* softwareseitig gestaltbar und damit anwendungsanpaßbar. Strukturell bieten *iproc's* den Vorzug, daß sie sich gleichförmig in das hier vorgeschlagene durchgängige Architekturkonzept von hierarchischen Schichten mit Instanzen als dominierenden Elementen einordnen, so daß der bisher vorliegende Strukturbruch in Betriebssystemen für die Unterbrechungsverarbeitung aufgehoben wird.

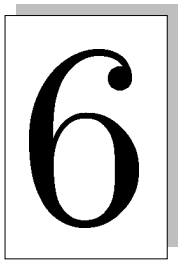
Innerhalb des Kerns wird damit ebenfalls eine konsequente Schichtung umgesetzt, aber nicht nur aus Gründen der Konformität zur vorgegebenen Architektur, sondern auch um den besonderen Ablaufeigenschaften von Unterbrechungsbehandlungen Rechnung tragen zu können:

- *kproc's:* größere Anzahl (typisch > 10), lange Lebensdauer, zyklisch, Blockierzustand;
- *iproc's:* kleine Anzahl (typisch 1–3), kurze Lebensdauer, "pop up", stets arbeitsbereit, eine spezielle *iproc*-Instanz verwaltet die übergeordneten *kproc*-Instanzen;
- *ictrl:* genau eine Instanz, welche gleichzeitig die Infrastrukturschicht für *iproc's* ist.

Instanzen aller Schichten werden im Prinzip gleich verwaltet und erhalten damit im Prinzip gleiche Ablaufeigenschaften. Die Differenzierbarkeit von Ausführungseigenschaften über das Mittel der Schichten stellt dabei den *Gegenpol zur angestrebten Homogenisierung* des Erscheinens und des Ablaufs in möglichst allen Bereichen eines Systems dar. Dieser Gegenpol fehlt in vielen anderen Systemen, die ebenfalls das Ziel der Vereinheitlichung oder Gleichstellung (→ *"unification"*) von Architekturkonzepten verfolgen, vor allem in Verbindung mit Objektorientierung.

Trotz Spezialisierung und damit Anpassung an die jeweiligen Gegebenheiten ist mit den Vorteilen von *iproc's* auf heutigen Maschinen ein nicht zu vernachlässigender Ablaufaufwand verbunden. Er wurde unter einigen Randbedingungen für die Implementation der *CADMUS*-Workstation untersucht und quantifiziert. Dieser Aufwand kann, durch das angewandte Arbeitsprinzip bedingt, nur in Grenzen gemindert, aber nicht beseitigt werden. Dazu wären vor allem eigene "reale" Aktivitätsträger in den beteiligten Schichten notwendig, so daß die häufigen Umschaltungen des hier einzigen Aktivitätsträgers zwischen den beteiligten Instanzen reduziert werden könnten. Dieser Entwicklungsschritt könnte durchaus in der Zukunft vollzogen werden, wenn im Zuge der weiteren Spezialisierung von Prozessoren für bestimmte Einsatzgebiete in Rechensystemen auch das für die Effektivität eminent wichtige Betriebssystem über eigene (Spezial-) Prozessoren verfügen würde. Am Ende dieses Kapitel wurde eine solche Option unter dem Blickwinkel hierarchischer Zuteilungen von Betriebsmitteln als "vertikale" Zuordnung von Prozessoren zu bestimmten Einsatzbereichen in Rechensystemen kurz diskutiert.

Insgesamt kann der Schluß gezogen werden, daß die hier vorgestellte Lösung der Unterbrechungsverarbeitung mit *iproc's* auf Basis der Infrastruktur *ictrl* in geeigneter Weise die eingangs dieser Arbeit formulierten Architekturziele auch für den untersten Bereich eines Betriebssystems erfüllt. Der damit verbundene Mehraufwand muß für die heute verfügbare Technik allerdings akzeptiert werden, wenngleich weitere Optimierungsmöglichkeiten bestehen, einige wurden genannt. Es bestätigt sich auch hier wieder die These, daß die heute angewandten Prinzipien in Betriebssystemen auf Basis der heutigen Technik bezüglich Funktion und Effizienz in gewisser Weise optimal sind. Forschungsvorhaben, welche darüber hinausgehen und mögliche künftige Entwicklungen abzuleiten und umzusetzen versuchen, sollten daher nicht nur anhand der heute möglichen Implementierbarkeit gemessen werden. Die künftige Entwicklung wird auch eine neue, veränderte Hardwareinfrastruktur hervorbringen, mit welcher sich dann vielleicht die hier aufgezeigten und untersuchten Techniken effizienter und vorteilhafter umsetzen lassen.



Kapitel 6

Zusammenfassung, Fazit und Ausblick

" The lack of operating system openness prevents a lot of innovation in computer science and applications." Jim Mitchell, Sun Fellow and Leader of the Spring Project [Mit95].

Das Ziel nach mehr Offenheit, nicht nur in einem technischen Sinn, sondern auch in bezug auf Zugänglichkeit, Durchschaubarkeit und Plausibilität, ist für Betriebssysteme nach wie vor eines des vorrangigen Ziele. Mit der in dieser Dissertation vorgeschlagenen Architektur wird dieser Problembereich angesprochen. Eine universelle Lösung gibt es nicht. Es ist Aufgabe der Forschung, verschiedene Wege und Lösungsansätze zu untersuchen. Dabei ordnet sich diese Dissertation in eine Reihe von Forschungsarbeiten in Deutschland und auch international zu Architekturfragen von Betriebs- und Anwendungssystemen ein, welches u.a. in aktuellen Dissertationen zum Ausdruck kommt [Heck91, Schm95, Ass96]. Die Ansätze unterscheiden sich. Das in dieser Dissertationen vorgestellte Architekturkonzept ist gekennzeichnet durch:

- die primäre Betrachtung der Elemente und Beziehungen eines dynamischen *ablaufenden Systems*, nicht vordergründig von Methoden zur Softwareherstellung,
- die Anwendung eines *Schichtenmodells*,
- eine *durchgängige Architektur* über alle Schichten,
- die lokale Konzentration von Teilverarbeitungen in identifizierbaren, *aktiven Verarbeitungsinstanzen*, deren Funktion es ist, selbständig *Dienste* auszuführen,
- die Trennung universeller und struktureller Merkmale von speziellen Ausprägungs- bzw. Ausführungseigenschaften individueller Zielbereiche in eine *generalisierte Architektur* und
  - in daraus für Zielbereiche abgeleitete *Ausprägungsvarianten* (Schichten, Elemente)
  - mit differenziert zugeordneten *Ausführungseigenschaften* (Dienste der Infrastruktur, Art von Elementen, Wirkung von Prozessen, Interaktion),
- die Verbindung existierender, anerkannter und akzeptierter Strukturmodelle:
  - *Client-Server-Modell* (→ *Instanzen*),
  - *Schichtenmodell* (→ *Rekursion von Infrastrukturen und Instanzbereichen*),
- die potentielle *Offenheit* aller Systemschichten für Anpassungen, so daß die Option eröffnet wird, auch infrastrukturelle Eigenschaften (*Dienste*, *Ausführungseigenschaften*) nach den Erfordernissen von Anwendungen (→ *top-down*) gestalten zu können.

Ein wesentlicher Unterschied zu anderen Ansätzen, welche eine möglichst einheitliche, gleichartige "Welt" zum Ziel haben, besteht hier in der Anerkennung der Notwendigkeit zur expliziten Differenzierung verschiedener Ausprägungs- und Ausführungseigenschaften von Elementen unterschiedlicher Systeme oder Systemschichten in ablaufenden Systemen. Mit der hier vorgeschlagenen Architektur wird keine Idealisierung in einer konzeptionellen Ebene vorgenommen, sondern der Betrachtungsgegenstand ist das ablaufende System mit den dort vorkommenden Elementen und Beziehungen, welche nach entsprechenden Merkmalen gestaltet werden können.

Es wird dabei ein kritischer Standpunkt zur undifferenzierten Übertragung einer objektorientierten Begriffswelt in die völlig andere Welt ablaufender Systeme bezogen. Objektorientierung ist eine vorteilhafte Methode zur Softwareherstellung und ein folgerichtiges Ergebnis der Entwicklung auf diesem Gebiet. Betriebssysteme haben eine andere Tradition und auch der Gegenstandsbereich ist ein anderer, wenngleich auch für Betriebssysteme die erforderliche Software herzustellen ist und dafür objektorientierte Methoden sinnvoll angewandt werden können.

Ein wesentliches, prägendes Merkmal der hier vorgestellten Architektur besteht in der expliziten Anwendung eines Schichtenmodells zur Abgrenzung und Differenzierung unterschiedlicher Ausführungseigenschaften innerhalb eines Systems. Erst durch diese Differenzierbarkeit wird das erforderliche Maß an Skalierbarkeit erreicht, um die vorgesehenen Architekturmerkmale tatsächlich auf verschiedene Systeme oder Systemschichten übertragen zu können.

In anderen Ansätzen kann dies nur für Teilbereiche geschehen, weil die notwendige Differenzierbarkeit nicht berücksichtigt wurde. Die angestrebte Struktur läßt sich nur in einer Systemschicht herstellen, für die das jeweilige Ausführungsmodell paßt. Der Rest des Systems unterliegt anderen Strukturprinzipien, welche nicht betrachtet werden. Dieser Restbereich sollte möglichst klein sein. Dieser Ansatz ist geeignet, wenn in der Tat nur die Elemente einer Systemschicht erfaßt werden sollen, z.B. für Betriebssystem-Aufsätze für verteilte Systeme. Wenn es aber auch um anpaßbare Infrastruktur geht, müssen zumindest zwei und im allgemeinen Fall alle Systemschichten einbezogen werden. Für das *CHEOPS*-Projekt trifft genau dieser Sachverhalt zu.
Für eine anwendungsanpaßbare Systemarchitektur muß das Gesamtsystem in einer durchgängigen, möglichst einheitlichen Weise erfaßt werden, um bestehende Strukturbrüche aufzuheben:

- zwischen prozeßorientiertem Anwendungs- und einem monolithischen Betriebssystem,
- innerhalb des Betriebssystems (z.B. zur Unterbrechungsverarbeitung) und
- zwischen der konzeptionellen Ebene und der Welt ablaufender (Betriebs-) Systeme bzw. zwischen Softwarestruktur und Ablaufstruktur.

Der zuletzt genannte Punkt wird mit der vorgeschlagenen Architektur dahingehend berücksichtigt, daß *Instanzen* sowohl konzeptionelle Elemente der Systemgliederung sind als auch später im ablaufenden System als identifizierbare Einheiten in jeweiligen Schichten existieren.

Systemarchitekturen mit hierarchischen Schichten wurden von Dijkstra Ende der sechziger Jahre und mit *Abstrakten Maschinen* in den siebziger Jahren eingeführt. Die hier vorgeschlagene Architektur rekursiver Schichten ordnet sich in diese Traditionslinie in Betriebssystemen ein und ergänzt diese dadurch, innerhalb *aller* Schichten ein durchgängiges Strukturmuster anzuwenden (→ *Instanzen*) und dabei ausdrücklich auch die untersten Systemschichten einzubeziehen.

Schichtenarchitekturen sind aber nicht nur eine Entwicklung aus der Vergangenheit, sie spielen auch in der aktuellen Forschung zu Betriebssystemen wieder eine Rolle, wie man es in aktuellen Veröffentlichungen verfolgen kann, beispielsweise auf der *OSDI'96* in *"Microkernels Meet Recursive Virtual Machines"* [FHL+96]. Ähnliches gilt für hierarchische Betriebsmittelzuteilungen, für die in der Zusammenfassung zu Kapitel 5 Beispiele angegeben wurden.

Kombiniert man eine Architektur rekursiver Schichten von Infrastrukturen und Instanzbereichen mit der für Anpassungen notwendigen *Offenheit*, ergeben sich weitreichende Konsequenzen:

- durch potentielle Offenheit aller Schichten werden Infrastruktureigenschaften anwendungsorientiert anpaßbar; bislang starre Dienste und Ausführungsmodelle werden gestaltbar;
- es besteht die reale Option für *top-down* Vorgehen bei der Anwendungsherstellung, d.h., Infrastruktur kann anwendungsgerecht angepaßt bzw. hergestellt werden;
- die bislang zweigeteilte Beziehung zwischen Betriebs- und Anwendungssystem wird durch die rekursiv in Beziehung stehenden Schichten von Infrastrukturen und Instanzbereichen ersetzt, die Rollenverteilung zwischen System- und Anwendungsherstellung wird im Prinzip aufgehoben, und Anpassungsarbeiten verlagern sich mehr zum Anwendungshersteller.

Damit können die eingangs dieser Dissertation aufgestellten Anforderungen an eine anwendungsanpaßbare Architektur als umgesetzt angesehen werden.

Als entscheidendes Problem muß der höhere Ablaufaufwand der hier vorgeschlagenen Architektur auf heutigen Rechnersystemen bewertet werden. Trotz der in der Architektur verankerten Spezialisierbarkeit ist eine instanzbasierte Verarbeitungsform vor allem in den unteren Systemschichten aufwendiger umzusetzen. Man könnte diesen Sachverhalt aber auch umgekehrt betrachten und die These formulieren, daß gegenwärtige (meist monolithische) Betriebssysteme mit heutigen Funktionen auf Basis der verfügbaren Rechnerarchitekturen bezüglich Effizienz in gewisser Weise *optimal implementiert* sind. Es besteht ein Zusammenhang zwischen Monoprozessorsystemen und monolithischen Betriebssystemen. Vorteilhaftere Systemarchitekturen bedingen in der Regel einen höheren Ablaufaufwand (→ *Prozesse, Instanzen, Schichten*).
In der Praxis vollziehen sich strukturelle Umbrüche erst, wenn ein tatsächlicher Bedarf dafür entsteht. So war der Auslöser für den Übergang zu einer Prozeßstruktur im Inneren des *Unix*-Kerns das Aufkommen von Multiprozessormaschinen, obwohl Dijkstra eine Prozeßstruktur für Betriebssysteme bereits vor gut 30 Jahren vorgeschlagen und angewandt hatte.

Es besteht aber eine Legitimation für die Forschung, auch über den Bereich des gegenwärtig Praktizierbaren hinauszugehen, denn

- künftige Rechnerarchitekturen werden andere Eigenschaften aufweisen, welche heute hemmende Ressourcenengpässe aufheben (mehrere, ggf. asymmetrisch zugeordnete, spezialisierte Prozessoren), so daß diese erwarteten Verbesserungen in der Grundlagenforschung zu Betriebssystemen durchaus vorweg genommen werden können, und
- durch die wahlweise Abrüstbarkeit der Ausführungsmodelle bzw. durch die Spezialisierbarkeit von Softwareinfrastrukturen für bestimmte Anwendungen können sich sogar auch vorteilhafte Eigenschaften auf der gegenwärtigen Grundlage ergeben.

Gerade der zuletzt genannte Aspekt relativiert den heute real vorhandenen Effizienznachteil neuer Software-Architekturen dahingehend, daß sich durch infrastrukturelle Anpaßbarkeit im Kontext eines Gesamtsystems durchaus Leistungsvorteile ergeben können. Im Gegenzug können auch suboptimierte monolithische Betriebssystem-Kerne mit festgelegten Infrastrukturen für Anwendungssysteme negative Leistungsbilanzen aufweisen (vgl. *μ-Kerne*). Die Untersuchung dieser Zusammenhänge kann als eine weiterführende Forschungsrichtung angesehen werden.

Der ausgebliebene Durchbruch bei an *Mach* orientierten μ-Kernen hängt entscheidend mit der Frage der Ausführungsmodelle zusammen. Nur wenn die Ausführungseigenschaften in dem Maße vereinfacht werden, daß der wegen einer vorteilhafteren Struktur bedingte Mehraufwand durch Beschleunigung (über-) kompensiert wird, ergibt sich eine Akzeptanzgrundlage. Ein Beispiel aus einem anderen Gebiet, wo das für eine Phase der Entwicklung gelang, waren *RISC*-Prozessoren.

In der Gestaltbarkeit von Ausführungseigenschaften liegt generell auch für gegenwärtige Systeme ein großes Leistungspotential, wenn je nach Anforderung wahlweise einfachere und effizientere Ausführungsmodelle zu Lasten starrer, "komfortabler" (=teurer) Ausführungsmodelle heute angewandter Universalbetriebssysteme eingesetzt werden können. Diese Quelle für Leistungsgewinn wird gegenwärtig wegen fehlender Anpassungsmöglichkeiten nicht erschlossen.

Das *CHEOPS*-Projekt und die Entwicklung des *CHEOPS*-Kerns stellt daher Fragen in den Vordergrund, die nicht auf die Re-Implementierung bestehender, bekannter Arbeitsprinzipien abzielen, sondern es sollen neue Prinzipien untersucht werden, auch wenn diese im direkten Effizienzvergleich vorerst nicht bestehen können. Eine bestmögliche Implementierung wird dabei natürlich angestrebt, aber nicht unter Aufgabe der primären Untersuchungs- und Architekturziele. Der *CHEOPS*-Kern sollte bewußt frei von dem sonst oft bestehenden Zwang nach Kompromissen aufgrund von Effizienz- und Kompatibilitätskriterien gehalten werden. Für ein Forschungsvorhaben ist dies legitim.

Neben allgemeinen Fragestellungen zur Architektur war ein zentraler Punkt des *CHEOPS*-Projekts die praktische Umsetzung des Architekturvorschlags anhand der Implementierung eines Betriebssystem-Kerns, der gleichzeitig als experimentelle Basis für andere Forschungsvorhaben im Rahmen des *CHEOPS*-Projektes dient. Diese weiterführenden Forschungsziele wurden in Abschnitt 1.9 genannt. Sie beziehen sich vor allem auf Experimente in unteren Systemschichten, d.h., aus diesen Zielstellungen leitete sich eine unmittelbare Anforderung nach Anpassungsmöglichkeiten auch in diesen Systemschichten ab. Der *CHEOPS*-Kern ist damit ein ideales Anwendungsfeld für eine durchgängige, anwendungsanpaßbare Systemarchitektur.

Für die praktische Seite der Implementierung wurde in dieser Dissertation die Gestaltung der Schicht der Unterbrechungsverarbeitung mit einer speziellen Art von Instanzen – *iproc's* – in den Mittelpunkt gestellt. Mit *iproc's* kann gezeigt werden, daß auch der Bereich der Unterbrechungsverarbeitung anhand der hier favorisierten Verarbeitungsform von Instanzen gestaltet werden kann und damit eine konsequente, durchgängige Umsetzung des Architekturkonzepts für alle Schichten eines Systems möglich ist. Instanzen zur Behandlung von Unterbrechungssignalen werden im Prinzip wie andere Instanzen verwaltet und erhalten dadurch ähnliche Ablaufeigenschaften. Durch Abgrenzung in einer separaten Schicht mit einer eigenen Infrastruktur wird dabei der Spezifik der Unterbrechungsverarbeitung Rechnung getragen. Es ergibt sich aber ein zusätzlicher Ablaufaufwand aus der dafür nötigen (Software-) Infrastrukturschicht *ictrl*, welcher für die Implementierungsplattform der *CADMUS*-Workstation in Abschnitt 5.3.5 quantifiziert und bewertet wurde. Neben den strukturellen Vorzügen ergeben sich aber auch vorteilhafte Ablaufeigenschaften, die in Abschnitt 5.3.6 diskutiert wurden.

Insgesamt kann der Schluß gezogen werden, daß sich mit der hier vorgeschlagenen und untersuchten Architektur ein größeres Maß an Übertragbarkeit, Skalierbarkeit und Offenheit und damit auch an Anwendungsanpaßbarkeit erreichen läßt, wenngleich nicht alle Probleme gelöst sind und sich auch neue Fragen ergeben, die weiterführende Arbeiten begründen. Diese werden zum Teil auch im Rahmen der Fortsetzung des *CHEOPS*-Projekts untersucht:

- Umsetzbarkeit und Praktikabilität verschiedener Ausführungsmodelle für Anwendungen und ein möglicher Leistungsgewinn im Kontext eines Gesamtsystems,
- Zugriffschutz zwischen Anwendungen auch ohne getrennte Adreßräume,
- Wirkungen dynamischer Austauschbarkeit von Instanzen in der Infrastruktur,
- Abbildung von Elementen objektorientierter Sprachen in ablaufende Systeme,
- feiner granulare dynamische Austauschbarkeit von Objekten innerhalb von Instanzen,
- dynamisches Rekonfigurierungsmanagement für adaptives Verhalten.

Literaturverzeichnis

[**ABB+86**] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. Mach: A New Kernel Foundation For UNIX Development. In *Proceedings of the Usenix Summer Conference, Atlanta, GA, USA, June 9-13*, pages 93–112, June 1986.

[**ABB+93a**] Assenmacher, H., Breitbach, T., Buhler, P., Hübsch, V., and Schwarz, R. PANDA – Supporting Distributed Programming in C++. *Proceedings of ECOOP'93*, In [**ECOOP93**]:361–383, July 1993.

[**ABB+93b**] Assenmacher, H., Breitbach, T., Buhler, P., Hübsch, V., and Schwarz, R. The PANDA System Architecture – A Pico-Kernel Approach. *Proceedings of the Third Workshop on Future Trends in Distributed Systems FTDS'93*, 1993.

[**Agh85**] Agha, G. A Model of Concurrent Computation in Distributed Systems. Technical Report TR 844, M.I.T., Artificial Intelligence Laboratory, 1985.

[**And95**] Anderson, T., Culler, D., Patterson, D., and the NOW-Team. A Case for NOW (Networks of Workstations). *IEEE Micro*, pages 54–64, February 1995.

[**ANSI84**] ANSI. *Draft Proposed Standard – Programming Language C.* X3J11 Language Subcommittee, American National Standards Institute, New York, USA, July 1984.

[**ANSI89**] ANSI. *American National Standard – Programming Language C (X3.159-1989).* American National Standards Institute, New York, USA, 1989.

[**AP93**] Arendt, J.W. and Phelan, J.M. An OS/2 Personality on Mach. In *Proceedings of the Third Usenix Mach Symposium, Santa Fe, New Mexico, USA*, pages 191–202, April 1993.

[**Ass96**] Assenmacher, H. *Ein Architekturkonzept zum Entwurf flexibler Betriebssysteme.* Dissertation, 174 S., Universität Kaiserslautern, Fachbereich Informatik, Februar 1996, erschienen im Shaker Verlag Aachen, 1996.

[**AT&T85**] AT&T. *System V Interface Definition (SVID), Issues 2 and 3.* American Telephone and Telegraph Company, Morristown, New Jersey, USA, UNIX Press, 1986.

[**AT&T90**] AT&T. *UNIX System V Release 4, Volumes 1-8.* AT&T UNIX System Laboratories, Inc., Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1990.

[**Bac86**] Bach, M.J. *The Design of the UNIX Operating System.* Software Series, 471 p. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1986.

[**BALL90**] Bershad, B.N., Anderson, T.E., Lazowska, E.D., and Levy, H.M. Lightweight Remote Procedur Call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[**BBH96**] Butenuth, R., Burke, W., and Heiß, H.U. Cosy – An Operating System for Highly Parallel Computers. *ACM Operating Systems Review*, 30(2):81–91, April 1996.

[**BCE+94**] Bershad, B.N., Chambers, C., Eggers, S., Maeda, C., McNamee, D., Pardyak, P., Savage, S., and Sirer, E.G. SPIN – An Extensible Microkernel for Application-specific Operating System Services. Technical Report 94-03-03, University of Washington, February 1994.

[**BCG+80**] Banino, J.-S., Caristan, A., Guillemont, M., Morisset, G, and Zimmermann, H. CHORUS: An Architecture for Distributed Systems. Technical Report No. 42, 68 p., Rapports de Recherche, Institut National de Recherche en Informatique et en Automatique (INRIA), Novembre 1980.

[**BCK+94**] Bershad, B., Cheriton, D., Kaashoek, F., Leach, P., Lucco, S., and Peterson, L. Panel: Radical Operating System Structures for Extensibility. *Proceedings of OSDI'94*, In [**OSDI94**]:195–200, November 1994.

[**BDDR91**] Bershad, B.N., Dean, R., Draves, R., and Rashid, R. Using Continuations to Implement Thread Management and Communication in Operating Systems. *Proceedings of the 13th SOSP*, In [**SOSP91**]:122–136, October 1991.

[**Bec96**] Becher, M. Basisdienste für die Interobjektkommunikation in einem objektorientierten, verteilten Betriebssystem. Projektarbeit, 47 S., Technische Universität Chemnitz, Fakultät für Informatik, Professur Betriebssysteme, Oktober 1996.

[**Bec97**] Becher, M. Entwicklung des Kommunikationsteilsystems für ein objektorientiertes, verteiltes Betriebssystem. Diplomarbeit, Technische Universität Chemnitz, Fakultät für Informatik, Professur Betriebssysteme, (in Bearbeitung), 1997.

[**Bersh92**] Bershad, B.N. The Increasing Irrelevance of IPC Performance for Micro-kernel-based Operating Systems. *Proceedings of the Usenix Workshop on Micro-Kernels and Other Kernel Architectures*, pages 205–212, April 1992.

[**Bersh95**] Bershad, B.N., Savage, S., Pardyak, P., Sirer, E.G., Fiuczynski, M.and Becker, D., Chambers, C., and Eggers, S. Extensibility, Safety and Performance in the SPIN Operating System. *Proceedings of the 15th SOSP*, In [**SOSP95**]:267–284, December 1995.

[**BGMR91**] Baron, R., Golub, D., Malan, G., and Rashid, R. DOS as a Mach3.0 Application. In *Proceedings of the Second Usenix Mach Symposium*, pages 27–40, November 1991.

[**BK88**] Bonomi, F. and Kumar, A. Adaptive Optimal Load Balancing in a Heterogenous Multiserver System with a Central Job Scheduler. *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 500–508, June 1988.

[**BML93**] Barton-Davis, P., McNamee, D., and Lazowska, E.D. Adding Scheduler Activations to Mach 3.0. In *Proceedings of the Third Usenix Mach Symposium, Santa Fe, New Mexico, USA*, pages 119–136, April 1993.

[**BMR82**] Brownbridge, D.R., Marshall, L.F., and Randell, B. The Newcastle Connection or UNIXes of the World Unite! *Software-Practice & Experience*, 12:1147–1162, 1982.

[**BMS96**] Baentsch, M., Molter, G., and Sturm, P. Modellierung von Betriebssystemdiensten mit Design-Patterns. Vortrag zum Treffen der GI-Fachgruppen *"Betriebssysteme"*, *"Fehlertolerierende Rechensysteme"* und *"Echtzeitsysteme"* zum Thema *"Responsive Systems"*, Universität Kaiserslautern, 28.-29. März 1996.

[**Bol89**] Boldyreff, C. UNIX-Standardization - An Overview. *Proceedings of the EUUG Spring Conference*, 1989.

[**Bom92**] Bomberger, A.C., et al. The KeyKOS Nanokernel Architecture. *Proceedings of the Usenix Workshop on Micro-Kernels and Other Kernel Architectures*, April 1992.

[**Bor92**] Borghoff, U.M. *Catalogue of Distributed File/Operating Systems.* 214 p. Springer-Verlag, Berlin Heidelberg New York Tokyo, 1992.

[**Bor96**] Borrmann, L. Kleine und kleinste Kerne: Betriebssysteme mit Mikro- und Nanokernen. *it+ti, Nr. 2/96*, 38(2):18–25, April 1996.

[**BR76**] Blevins, P.R. and Ramamoorthly, C.V. Aspects of a Dynamically Adaptive Operating System. *IEEE Transactions on Computers*, 25(7):713–724, July 1976.

[**Bro95**] Broner, G. UNICOS/mk: A Distributed Operating System with Single System Image. *Presentation at Cray Users Group*, 1995.

[**Camp87**] Campbell, R.H., Johnston, G., and Russo, V. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). *ACM Operating Systems Review*, 21(3):9–17, July 1987.

[**Camp93**] Campbell, R.H., Islam, N., Raila, D., and Madany, P. Designing and Implementing Choices: An Object-Oriented System in C++. *Communications of the ACM*, 36(9):117–126, September 1993.

[**CBHR93**] Cahill, V., Balter, R., Harris, N.R., and Rousset de Pina, X., (Eds.). *The COMANDOS Distributed Application Platform.* Research Reports ESPRIT, Project 2071, Volume 1, 312 p. Springer-Verlag, Berlin Heidelberg New York Tokyo, 1993.

[**CD88**] Cooper, E.C. and Draves, R.P. C Threads. Carnegie Mellon University, Technical Report CMU-CS-88-154, July 1988.

[**CG95**] Cahill, V. and Gowing, B. Making Meta-Object Protocols Practical for Operating Systems. *Proceedings of IWOOOS'95*, In [**IWOOOS95**]:52–55, August 1995.

[**CH70**] Commoner, F. and Holt, A.W. Events and Conditions. *in: Record Project MAC Conference on Concurrent Systems and Parallel Computation*, Woods Hole, Massachusetts, pp. 3-52, 1970.

[**CH74**] Campbell, R.H. and Habermann, A.N. The Specification of Process Synchronization by Path Expressions. *in*, Lecture Notes in Computer Science (LNCS 16), pages 89-102, Springer-Verlag, Berlin Heidelberg New York Tokyo 1974.

[**Che88**] Cheriton, D.R. The V Distributed System. *Communications of the ACM*, 31(3):314–333, March 1988.

[**Cheriton94**] Cheriton, D.R. and Duda, K.J. A Caching Model of Operating System Kernel Functionality. *Proceedings of OSDI'94*, In [**OSDI94**]:179–193, November 1994.

[**CHJ+94**] Cahill, V., Hogan, C., Judge, A., O'Grady D., and Tangney, B. Extensible Systems – The Tigger Approach. *Proceedings of SIGOPS European Workshop*, 1994.

[**CIKM92**] Campbell, R.H., Islam, N., Kougiouris, P., and Madany, P. Practical Examples of Reification and Reflection in C++. *Proceedings of the International Workshop on New Models for Software Architecture*, 1992.

[**CLFL94**] Chase, J.S., Levy, H.M., Feeley, M.J., and Lazowska, E.D. Sharing and Protection in a Single Address Space Operating System. *ACM Transactions on Computer Systems*, 12(4), November 1994.

[**CLK93**] Chen, D.J., Lee, P.J., and Ku, K.L. A Modeling Approach to the Construction of Object-Oriented Operating Systems. *Journal of Object-Oriented Programming (JOOP)*, pages 52–66, November-December 1993.

[**CM93**] Chiba, S. and Masuda, T. Designing an Extensible Distributed Language with a Meta-Level Architecture. *Proceedings of ECOOP'93*, In [**ECOOP93**]:482–501, July 1993.

[**Coad91**] Coad, P. and Yourdon, E. *Object-Oriented Design*. Yourdon Press Computing Series, 197 p. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1991.

[**Coc95**] Cockcroft, A. *Sun Performance And Tuning – SPARC & Solaris*. 253 p. Sun Microsystems, Inc., Mountain View, California, USA, 1995.

[**Cop92**] Coplien, J.O. *Advanced C++ Programming Styles and Idioms*. 520 p. Addison-Wesley Publishing Company, Reading, Massachusetts, USA, 1992.

[**Cox86**] Cox, B.J. *Object-Oriented Programming – An Evolutionary Approach*. 274 p. Addison-Wesley Publishing Company, Reading, Massachusetts, USA, 1986.

[**CRT95**] Campbell, R., Raila, D.K., and Tan, S.M. An Object-Oriented *nano-kernel* for Operating System Hardware Support. *Proceedings of IWOOOS'95*, In [**IWOOOS95**]:220–223, August 1995.

[**CSA90**] CSA. *Communications Systems Architecture – Architectural Description*. 236 p., Esprit Project No.237, R&D Area 4.3.1: Communication Systems, Mari Applied Technologies, Ltd., March 1990.

[**Cust93**] Custer, H. *Inside Windows NT*. 397 p. Microsoft Press, Redmond, Washington, USA, 1993.

[**Dav92**] Davis, W.S. *Operating Systems – A Systematic View*. 672 p. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, USA, fourth edition, 1992.

[**DD68**] Daley, R.C. and Dennis, J.B. Virtual Memory, Processes, and Sharing in MULTICS. *Communications of the ACM*, 11(5):306–312, May 1968.

[**Dei90**] Deitel, H.M. *An Introduction to Operating Systems*. 853 p. Addison-Wesley Publishing Company, Reading, Massachusetts, USA, second edition, 1990.

[**DFGR90**] Dean, R., Forin, A., Golub, D., and Rashid, R. Unix as an Application Program. In *Proceedings of the Summer Usenix Conference, Anaheim, California, USA, June 11-15*, pages 87–96, June 1990.

[**Dij65**] Dijkstra, E.W. Solution of a Problem in Concurrent Programming. *Communications of the ACM*, 8(9):569–574, September 1965.

[**Dij68a**] Dijkstra, E.W. Cooperating Sequential Processes. *in: Programming Languages (F. Genuys, ed.)*, pages 43–112, Academic Press, London and New York, 1968.

[**Dij68b**] Dijkstra, E.W. The Structure of THE – Multiprogramming System. *Communications of the ACM*, 11(5):341–346, May 1968.

[**Dij71**] Dijkstra, E.W. Hierarchical Ordering of Sequential Processes. *Acta Informatica*, 1(1):115–138, January 1971.

[**Dij75**] Dijkstra, E.W. Guarded Commands, Nondeterminancy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[**DKM+94**] Douglis, F., Kaashoek, F., Marsh, B., Cáceres, R., Li, K., and Tauber, J.A. Storage Alternatives for Mobile Computers. *Proceedings of OSDI'94*, In [**OSDI94**]:25–37, November 1994.

[**DN66**] Dahl, O.-J. and Nygaard, K. SIMULA – an ALGOL-Based Simulation Language. *Communications of the ACM*, 9(9):671–678, September 1966.

[**DPH92**] Druschel, P., Peterson, L.L., and Hutchinson, N.C. Beyond Micro-Kernel Design: Decoupling Modularity and Protection in Lipto. In *Proceedings of the 12th International Conference on Distributed Computing Systems, Yokohama, Japan, June 9-12,*, pages 512–520. IEEE Computer Society Press, 1992.

[**ECOOP91**] America, P., (Ed.). *ECOOP'91, European Conference on Object-Oriented Programming*, Geneva, Switzerland, July 15-19, 1991, Proceedings in: Lecture Notes in Computer Science (LNCS 512), 396 p., Springer-Verlag, Berlin Heidelberg New York Tokyo, 1991.

[**ECOOP93**] Nierstrasz, O.M., (Ed.). *ECOOP'93, Object-Oriented Programming, 7th European Conference*, Kaiserslautern, Germany, July 26-30, 1993, Proceedings in: Lecture Notes in Computer Science (LNCS 707), 530 p., Springer-Verlag, Berlin Heidelberg New York Tokyo, 1993.

[**Ein89**] Einert, E. *Strukturelle und funktionale Aspekte von Fenstersystemen für UNIX-Betriebssysteme.* Dissertation, 121 S., Technische Universität Karl-Marx-Stadt, Mai 1989.

[**Eld93**] Elder-Vass, D. *MVS Systems Programming.* 530 p. McGraw-Hill, London, U.K., 1993.

[**ELZ86**] Eager, D.L., Lazowska, E.D., and Zahorjan, J. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, May 1986.

[**End88**] Endres, A. Software-Wiederverwendung: Ziele, Wege und Erfahrungen. *Informatik-Spektrum*, 1988 (11):85–95, November 1988.

[**Eul96**] Eulitz, A. Dynamisches typsicheres Linken von C++-Programmen. Diplomarbeit, 111 S., Technische Universität Chemnitz, Fakultät für Informatik, Professur Betriebssysteme, September 1996.

[**EW95**] Eckert, C. and Windisch, H.-M. A Top-down Driven, Object-Based Approach to Application-Specific Operating System Design. *Proceedings of IWOOOS'95*, In [**IWOOOS95**]:153–156, August 1995.

[**Fab76**] Fabry, R.S. How to Design a System in which Modules Can be Changed on the Fly. *Proceedings of 2nd International Conference on Software Engineering*, 1976.

[**FGG+91**] Furth, B., Grostick, D., Gluch, D., Rabbat, G., Parker, J., and McRoberts, M. *REAL-TIME UNIX SYSTEMS – Design and Application Guide.* The Kluwer International Series in Engineering and Computer Science, SECS 121, 316 p. Kluwer Academic Publishers, Boston London Dordrecht, 1991.

[**FHL+96**] Ford, B., Hibler, M., Lepreau, J., Tullmann, P., Back, G., and Clawson, S. Microkernels Meet Recursive Virtual Machines. *Proceedings of OSDI'96*, In [**OSDI96**]:137–151, October 1996.

[**FJ89**] Foote, B. and Johnson, R.E. Reflective Facilities in Smalltalk-80. *Proceedings of OOPSLA'89, SIGPLAN Notices, ACM Press*, 24(10):327–335, October 1989.

[**Fran96a**] Franke, J. Objektorientierte Architektur der Speicherverwaltung eines μ-kern Betriebssystems. Projektarbeit, 51 S., Technische Universität Chemnitz, Fakultät für Informatik, Professur Betriebssysteme, Januar 1996.

[**FS96**] Ford, B. and Susarla, S. CPU Inheritance Scheduling. *Proceedings of OSDI'96*, In [**OSDI96**]:91–105, October 1996.

[**Gam93**] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns – Abstraction and Reuse of Object-Oriented Design. *Proceedings of ECOOP'93*, In [**ECOOP93**]:406–431, July 1993.

[**Gam94**] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns – Elements of Reusable Object-Oriented Software.* Professional Computing Series, 395 p. Addison-Wesley Publishing Company, Reading, Massachusetts, USA, 1994.

[**GC94**] Goodheart, B. and Cox, J. *The Magic Garden Explained, The Internals of UNIX System V Release 4, An Open Systems Design.* 664 p. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1994.

[**GCS96**] Goldstein, S.C., Culler, D., and Schauser, K.E. Enabling Primitives for Compiling Parallel Languages. In Szymanski, B.K. and Sinharoy, B., (Eds.), *Languages, Compilers and Run-Time Systems for Scalable Computers*, pages 153–168. Kluwer Academic Publishers, Boston London Dordrecht, Workshop Proceedings, Troy, New York, USA, May 1995, book appeared in 1996.

[**Geb93**] Gebhard, J. Architektur objektorientierter Betriebssysteme. Diplomarbeit, 95 S., Technische Universität Chemnitz, Fakultät für Informatik, Professur Betriebssysteme, Oktober 1992.

[**GG92**] Gien, M. and Grob, L. Micro-kernel Based Operating Systems: Moving UNIX onto Modern System Architectures. In *Proceedings of the UniForum'92 Conference*, San Francisco, USA, January 22-24, 1992.

[**GGV96**] Goyal, P., Guo, X., and Vin, H.M. A Hierarchical CPU Scheduler for Multimedia Operating Systems. *Proceedings of OSDI'96*, In [**OSDI96**]:107–121, October 1996.

[**GH90**] Geihs, K. and Hollberg, U. Retrospective on DACNOS. *Communications of the ACM*, 33(4):439–448, April 1990.

[**GK92**] Gittinger, J and Krüger, U. Prozeßsignalisierungen statt Unterbrechungen. Forschungsbericht, 14/92, 37 S., Universität Karlsruhe, Institut für Betriebs- und Dialogsysteme, Mai, 1992.

[**GLS94**] Gropp, W., Lusk, E., and Skjellum, A. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* The MIT Press, Cambridge, Massachusetts, USA, 1994.

[**GMS77**] Geschke, C., Morris, J., and Satterthwaite, E. Early Experiences with Mesa. *Communications of the ACM*, 20(8):540–553, August 1977.

[**Gö90**] Görling, H. *Das Mainframe-Betriebssystem BS2000.* R. Oldenbourg Verlag, München Wien, 1990.

[**GR96**] Gehring, J. and Reinefeld, A. MARS – A Framework for Minimizing the Job Execution Time in a Metacomputing Environment. *Future Generation Computer Systems*, pages 87–99, 1996.

[**Grau96a**] Graupner, S., Gläß, A., and Naumann, T. rse – eine Umgebung für entfernte Script-Ausführung. *SIWORK'96, Workstations und ihre Anwendungen*, 1996. Universität Zürich, Schweiz, 14.-15. Mai 1996, In: Cap, C. (Hrsg.), *Proceedings zur Fachtagung*, Seiten 327-330, vdf Hochschulverlag AG, ETH Zürich, 1996.

[**Grau96b**] Graupner, S. sash – A stand-alone Shell for the *CheOpS*-Kernel. Forschungsbericht, In [**Kal96b**]: 57-69, März 1996.

[**Grau95a**] Graupner, S. and Kalfa, W. Adaptable Infrastructures for Operating Systems. *Proceedings of Fourth IEEE International Workshop on Object-Orientation in Operating Systems, IWOOOS'95, University of Lund, Sweden, August 14-15, 1995*, In [**IWOOOS95**]:162–165, 1995.

[**Grau95b**] Graupner, S. *Nichtprozedurale Ablaufformen in imperativen Sprachen – Coroutinen und preemptive Threads in C.* Chemnitzer Informatik-Berichte 95/07, 51 S., August 1995.

[**Grau95c**] Graupner, S. and Suhr, A. Plan9 – erste Erfahrungen mit dem neuen Betriebssystem von AT&T, Bell Labs. *Offene Systeme*, Band 4, Heft 3, Seiten 140-147, Springer Verlag, August 1995.

[**Grau95d**] Graupner, S. Neue Trends in Betriebssystemen. *Offene Systeme*, Band 4, Heft 2, Seiten 98-103, Springer Verlag, Mai 1995.

[**Grau95e**] Graupner, S. and Kalfa, W. Designing and Implementing Dynamic Adaptation for the COSAI-Kernel. Forschungsbericht, 6 S., November 1995.

[**Grau95f**] Graupner, S. Dynamische Adaption von Betriebssystemen durch objektorientierte Methoden in einer rekursiven Systemarchitektur. Forschungsbericht, 15 S., Oktober 1995.

[**Grau95g**] Graupner, S. Algorithmen für dezentrale Prozeßsteuerung. Forschungsbericht, 22 S., März 1995.

[**Grau94a**] Graupner, S., Kalfa, W., and Schubert, F. Multi-Level Architecture for Object-Oriented Operating Systems. Technical Report TR-94-056, 28 p., International Computer Science Institute (ICSI), Berkeley, California, USA, November 1994.

[**Grau94b**] Graupner, S. Strukturformen von Servern verteilter Anwendungen unter UNIX. *Offene Systeme*, Band 3, Heft 4, Seiten 208-215, Springer Verlag, November 1994.

[**Grau94c**] Graupner, S. An interactive Test Support for stand-alone Environments. Forschungsbericht, 13 S., August 1994.

[**Grau94d**] Graupner, S. and Kalfa, W. Structuring Interrupts in μ-kernel Operating Systems. Forschungsbericht, 7 S., Juni 1994.

[**Grau94e**] Graupner, S. The Essence of Objects and Object-Orientation. Forschungsbericht, 20 S., Mai 1994.

[**Grau94f**] Graupner, S. Coroutinen und preemptive Threads in C – Mechanismen und Realisierbarkeit. Forschungsbericht, 50 S., April 1994.

[**Grau93a**] Graupner, S. Mechanismen zur Verwaltung heterogener Software-Projekte unter Anwendung objektorientierter Technologien. *Offene Systeme*, Band 2, Heft 1, Seiten 23-31, Springer Verlag, März 1993.

[**Grau93b**] Graupner, S. Objektorientierung in Betriebssystemen. Forschungsbericht, 24 S., Oktober 1993.

[**Grau92a**] Graupner, S. Hardwarenahe Betriebssystemschnittstellen. Forschungsbericht, 25 S., Juni 1992.

[**Grau92b**] Graupner, S., Bähnsch, K., Müller, K., Müller, T., and Schubert, F. Das Betriebssystem "BirliX" – Konzepte und Realisierung. *Wissenschaftliche Zeitschrift der TU Chemnitz*, Ausgabe 34, 2/92, Seiten 29-44, Februar 1992.

[**Grau92c**] Graupner, S. Dokumentation der Portierungsschritte von BirliX auf CADMUS-Workstations. Forschungsberichte, (1.) *Erste Schritte bei der Portierung der stand-alone Variante des Testers*, August 1990, (2.) *Die Portierung der Clock*, Januar 1991, (3.) *Zugang zum ICC für Console-IO*, Februar 1992, (4.) *Zugang zum ICC via ICC-Kernel für Console-IO*, März 1992, (5.) *Low Level Disk Interface via ICC*, April 1992.

[**Grau92d**] Graupner, S. Die Evolution von Betriebssystem-Kernen – Betrachtungen zur Portierung des BirliX-Betriebssystems. Forschungsbericht, 14 S., Mai 1992.

[**Grau91a**] Graupner, S. Zuverlässigkeit in speziellen verteilten Betriebssystemen. Band zum Workshop *"Verteilte Betriebssysteme"* der RWTH Aachen, Teil IV, Beitrag 15, 28 S., 30. September – 5. Oktober 1991, Söllerhaus, Riezlern im Kleinwalsertal, Österreich, Oktober 1991.

[**Grau91b**] Graupner, S. and Müller, T. Konzepte des Betriebssystems BirliX. *GUUG Nachrichten*, Band 8, Heft 27, Seiten 5-13, Oktober 1991.

[**Grau91c**] Graupner, S. Device Management in microkernel-Betriebssystemen. Diplomarbeit, 152 S., Technische Universität Chemnitz, Fachbereich Informatik, Lehrstuhl Betriebssysteme, Juli 1991.

[**Grau90**] Graupner, S. Realisierung des geräteabhängigen Serverteils des Fenstersystems X Window. Ingenieurbeleg, 59 S., Technische Universität Chemnitz, Fachbereich Informatik, Februar 1990.

[**group84**] /usr/group. *1984 /usr/group Standard.* /usr/group Standards Committee, UniForum, Santa Barbara, California, USA, 1984.

[**GS89**] Gopinath, P. and Schwan, K. CHAOS: Why One Cannot Have Only An Operating System for Real-Time Applications. *ACM Operating Systems Review*, 23(3):106–125, July 1989.

[**Hab76**] Habermann, A.N., Flon, L., and Cooprider, L. Modularization and Hierarchie in a Family of Operating Systems. *Communications of the ACM*, 19(5):266–272, May 1976.

[**Han70**] Hansen, P.B. The Nucleus of a Multiprogramming System. *Communications of the ACM*, 13(4):238–250, April 1970.

[**Han72**] Hansen, P.B. Structured Multiprogramming. *Communications of the ACM*, 15(7):574–578, July 1972.

[**Han75**] Hansen, P.B. The Programming Language Concurrent Pascal. *IEEE Transactions on Software Engineering*, SE-1:199–207, June 1975.

[**Han78**] Hansen, P.B. Distributed Processes: A Concurrent Programming Concept. *Communications of the ACM*, 21(11):934–941, November 1978.

[**Här90a**] Härtig, H., Kühnhauser, W., Kowalski, O., Lux, W., Reck, W., Streich, H., and Goos, G. Architecture of the BirliX Operating System. Technical Report, German National Research Center for Computer Science (GMD), Birlinghoven, March 1990.

[**Här90b**] Härtig, H. and Reck, W. Software Configuration Management for Medium-Size Systems. *Proceedings of CAiSE'90*, 1990.

[**Här92**] Härtig, H., Kühnhauser, W., Lux, W., and Reck, W. Operating System(s) on Top of Persistent Object Systems. *Proceedings of 25th HICSS*, pages 790–799, January 1992.

[**Hart96**] Harte, P. Integration des Plattentreibers in den *CheOpS*-Kern. Projektarbeit, 24 S., Technische Universität Chemnitz, Fakultät für Informatik, Professur Betriebssysteme, Mai 1996.

[**Hart97**] Harte, P. Dynamisches Laden und Starten von *kproc*-Instanzen für den *CheOpS*-Kern. Diplomarbeit, Technische Universität Chemnitz, Fakultät für Informatik, Professur Betriebssysteme, (in Bearbeitung), 1997.

[**Heck91**] Heck, A. *Adaptive Betriebssystemkonzepte.* Dissertation, 215 S., Technische Hochschule Darmstadt, Fachbereich für Informatik, August 1991.

[**Herr91**] Herrtwich, R.G., (Ed.). *Network and Operating System Support for Digital Audio and Video, Second International Workshop*, Heidelberg, Germany, November 18-19, 1991, Proceedings in: Lecture Notes in Computer Science (LNCS 614), 403 p., Springer-Verlag, Berlin Heidelberg New York Tokyo, 1991.

[**Hew77**] Hewitt, C.E. Viewing Control Structures as Patterns of Passing Messages. *Journal of Artificial Intelligence*, 8(3):323–364, June 1977.

[**Hil92**] Hildebrand, D. An Architectural Overview of QNX. *Proceedings of Micro-kernel and Other Kernel-Architectures Usenix Workshop, Seattle, Washington, USA*, pages 113–126, April 1992.

[**Hil93**] Hills, T. Structured Interrupts. *ACM Operating Systems Review*, 27(1):51–68, January 1993.

[**Hoa74**] Hoare, C.A.R. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[**Hoa78**] Hoare, C.A.R. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[**Hof96**] Hofmann, F. Was bieten Betriebssysteme den Anwendern? *it+ti, Nr. 2/96*, 38(2):6–11, April 1996.

[**Hor89**] Horn, C. Is Object Orientation a Good Thing for Operating Systems. *Proceedings of European Workshop on Progress in Distributed Operating Systems and Distributed Systems*, In [**Schrö89**]:60–74, April 1989.

[**Hü90**] Hüsken, V. *Objekt-Orientierung und Parallelität in Betriebssystem und Programmiersprache.* Dissertation, 137 S., RWTH Aachen, Fakultät für Elektrotechnik, Februar 1990.

[**IEEE85**] IEEE. *Trial-Use Standard Specification for Microprocessor Operating System Interfaces (Std. 855).* Institute of Electrical and Electronics Engineers, Inc., New York, USA, 1985.

[**IEEE88**] IEEE. *Portable Operating System Interface for Computer Environments (Std. 1003.1).* Institute of Electrical and Electronics Engineers, Inc., New York, USA, 1988.

[**IEEE90**] IEEE. *Information Technology – Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API, C Language), IEEE Std 1003.1-1990 and ISO/IEC 9945-1:1990.* Institute of Electrical and Electronics Engineers, Inc., New York, USA, 356 p., appr. 1990 by the International Organization for Standardization (ISO) and by the International Electrotechnical Commission (IEC), 1st edition, 1990.

[**IM93**] Inverardi, P. and Mazzanti, F. Experimenting with Dynamic Linking with Ada. *Software-Practice & Experience*, 23(1):1–14, January 1993.

[**Ing81**] Ingalls, H.H. Design Principles Behind Smalltalk. *BYTE*, 6(8):286–298, August 1981.

[**IWOOOS95**] Theimer, M. and Cabrera, L.-F., (Eds.). *Fourth International Workshop on Object-Orientation in Operating Systems, IWOOOS'95*, Lund, Sweden, August 14-15, 1995. Proceedings, 242 p., IEEE Computer Society Press, 1995.

[**Joy91**] Joy, B. Future Operating Systems: An Environmental Scenario and Consequential Challenges. *Dagstuhl Workshop* Operating Systems of the 90s and Beyond, In [**KN91**]:34–39, July 1991.

[**JPN84**] Joseph, M., Prasad, V.R., and Natarajan, N. *A Multiprocessor Operating System.* Series in Computer Science, 477 p. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1984.

[**Jus93**] Justice, B. Echtzeit-Eigenschaften von SunOS 5.x. *iX Multiuser Multitasking Magazin*, 93(6):136–139, Juni 1993.

[**Kaash95**] Engler, D.R., Kaashoek, M.F., and O'Toole Jr., J. Exokernel: An Operating System Architecture for Application-Level Resource Management. *Proceedings of the 15th SOSP*, In [**SOSP95**]:251–266, December 1995.

[**Kal90**] Kalfa, W. *Betriebssysteme.* 400 S. Akademie-Verlag, Berlin, 2. Auflage, 1990.

[**Kal92a**] Kalfa, W. Proposal of an External Processor Scheduling in Micro-Kernel based Operating Systems. Technical Report TR-92-028, 14 p., International Computer Science Institute (ICSI), Berkeley, California, USA, May 1992.

[**Kal92b**] Kalfa, W. From Process-Ressource Paradigm to the Object-Oriented Paradigm of Operating Systems. Presentation, International Computer Science Institute (ICSI), Berkeley, California, USA, April 13, 1992.

[**Kal92c**] Kalfa, W. and Stainov, R. A Light Weight Kernel Server. *Microprocessing and Microprogramming*, North-Holland(35):39–46, 1992.

[**Kal93**] Kalfa, W. Untersuchung rekursiver und reflexiver Methoden zur Anpassung in objektorientierten verteilten Betriebssystemen an unterschiedliche Hardware und Software. *Bewilligter Antrag auf Gewährung einer Sachbeihilfe durch die DFG*, 1993.

[**Kal96a**] Graupner, S., Kalfa, W., Schubert, F. Vogel, R., Werner, J., and Wohlrab, L. Dynamische Adaption in Betriebssystemen – Das *CheOpS*-Projekt. Forschungsbericht, In [**Kal96b**]:5-17, März 1996.

[**Kal96b**] Kalfa, W., (Ed.). *Dynamische Adaption in Betriebssystemen – Das CheOpS-Projekt.* Chemnitzer Informatik-Berichte, CSR-96-03, 69 S., März 1996.

[**KE95**] Kleiman, S. and Eykholt, J. Interrupts as Threads. *ACM Operating Systems Review*, 29(2):21–26, April 1995.

[**Kee89**] Keedy, J.L. A History of the MONADS Project: 1976–1988. Technical Report 10/89, 18 S., Forschungsbericht des Studiengangs Informatik, Universität Bremen, Oktober 1989.

[**Kern95**] Kerner, U. Objektorientierte Architektur einer persistenten Datenverwaltung. Diplomarbeit, 53 S., Technische Universität Chemnitz, Fakultät für Informatik, Professur Betriebssysteme, Februar 1995.

[**Kic94**] Kiczales, G. Why are Black Boxes so Hard to Reuse? (Towards a New Model of Abstraction in the Engineering of Software). Invited Talk at OOPSLA'94, 1994.

[**Kic96**] Kiczales, G. Beyond the Black Box: Open Implementation. *IEEE Software*, pages 8–11, January 1996.

[**Kicz91**] Kiczales, G., Riviéres, J. des, and Bobrow, D.G. *The Art of the Metaobject Protocol.* 335 p. The MIT Press, Cambridge, Massachusetts, USA, 1991.

[**Klei94**] Kleinöder, J. PM Systemarchitektur. *In: Wedekind, H. (Hrsg.): Verteilte Systeme, Grundlagen und zukünftige Entwicklungen aus der Sicht des SFB 182*, pages 357–370, BI-Wissenschafts Verlag, Mannheim 1994.

[**KN91**] Karshmer, A.I. and Nehmer, J., (Eds.). *Operating Systems of the 90s and Beyond*, Dagstuhl Castle International Workshop, Germany, July 8-12, 1991, Proceedings in: Lecture Notes in Computer Science (LNCS 563), 284 p., Springer-Verlag, Berlin Heidelberg New York Tokyo, 1991.

[**Knu66**] Knuth, D.E. Additional Comments on a Problem in Concurrent Programming. *Communications of the ACM*, 9(5), June 1966.

[**Koc96**] Koch, A. System-Architektur und Software-Engineering. *Offene Systeme*, Band 5, Heft 1, Seiten 30-34, Springer Verlag, Februar 1996.

[**KR78**] Kernighan, B.W. and Ritchie, D.M. *The C Programming Language.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1978.

[**Kra88**] Krakowiak, S. *Principles of Operating Systems.* 469 p. The MIT Press, Cambridge, Massachusetts, USA, 1988.

[**KT91**] Karshmer, A.I. and Thomas, J.N. Are Operating Systems at RISC. *Dagstuhl Workshop* Operating Systems of the 90s and Beyond, In [**KN91**]:48–52, July 1991.

[**Lamp78**] Lamport, L. Time, Clocks, and the Ordering of Events in Distributed Systems. *Communications of the ACM*, 21(7):558–564, July 1978.

[**Law93**] Lawrence, J.T. *AS/400 Architecture and Application.* John Wiley & Sons, Inc., New York, USA, 1993.

[**Leff88**] Leffler, S.J., McKusick, M.K., Karels, M.J., and Quarterman, J.S. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Series in Computer Science, 471 p. Addison-Wesley Publishing Company, Reading, Massachusetts, USA, 1988.

[**Lepr93**] Lepreau, J., Hibler, M., Ford, B., and Law, J. In-Kernel Servers on Mach3.0: Implementation and Performance. In *Proceedings of the Third Usenix Mach Symposium, Santa Fe, New Mexico, USA*, pages 39–56, April 1993.

[**Lie95**] Liedtke, J. On μ-Kernel Construction. *Proceedings of the 15th SOSP*, In [**SOSP95**]:237–250, December 1995.

[**LMM93**] Leslie, I.M., McAuley, D.R., and Mullender, S.J. Pegasus – Operating System Support for Distributed Multimedia Systems. *ACM Operating Systems Review*, 27(1):69–78, January 1993.

[**Löhr78**] Löhr, K.P., Isle, R., and Goullon, H. Dynamic Restructuring in an Experimental Operating System. *IEEE Transactions on Software Engineering*, SE-4(4):10, July 1978.

[**Lou91**] Loukides, M. *System Performance Tuning.* 313 p. O'Reilly & Associates, Inc., Sebastopol, California, USA, 1991.

[**Lucco93**] Anderson, T.E., Graham, S.L., Lucco, S., and Wahbe, R. Efficient Software Based Fault Isolation. *Proceedings of the 14th SOSP*, In [**SOSP93**]:203–216, December 1993.

[**Mae86**] Maes, P. Issues in Computational Reflection. *In Meta-Level Architectures and Reflection*, In [**Mae86**]:21–35, 1986.

[**Mae86**] Maes, P. and Nardi, D., (Eds.). *Meta-Level Architectures and Reflection*, Selected Papers, 355 p., Presented at Workshop *Meta-Level Architectures and Reflection*, Alghero, Italy, October 27-30 1986. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, The Netherlands. 1988.

[**Mae87**] Maes, P. *Computational Reflection.* Dissertation, Vrije Universiteit Brussel, Artificial Intelligence Laboratory, Brussels, Belgium, January 1987.

[**Marty88**] Marty, R. Von der Subroutinentechnik zu Klassenhierarchien. Berichte des Instituts für Informatik, Nr. 88.04, 44 S., Universität Zürich, 1988.

[**Mau96**] Maurer, D. PXROS – ein skalierbarer Mikrokern mit besonderen Echtzeiteigenschaften. *it+ti, Nr. 2/96*, 38(2):26–32, April 1996.

[**Mes94**] Messing, W. Prozessor- und Arbeitsspeicherverwaltung (Basissystem Teil 1). In P. Jilek, (Ed.), *BS2000/OSD – Technische Beschreibung.* 192 S., Siemens Nixdorf Informationssysteme AG, München, Februar 1994.

[**Mes95**] Messing, W. BS2000 SUNRiSE: Ein Portierungsprojekt. In P. Schlichtiger, (Ed.), *3. BS2000-Seminar, Positionierung und aktuelle Entwicklungsvorhaben.* 14.-19. September 1995, Teil 4, 15 S., Siemens Nixdorf Informationssysteme AG, München, September 1995.

[**Mey88**] Meyer, B. *Object-Oriented Software Construction.* International Series in Computer Science, 534 p. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1988.

[**MHM+95**] Murata, K., Horspool, R.N., Manning, E.G., Yokote, Y., and Tokoro, M. Unification of Active and Passive Objects in an Object-Oriented Operating System. *Proceedings of IWOOOS'95*, In [**IWOOOS95**]:68–71, August 1995.

[**Mil92**] Miller, D.D. *VAX/VMS Operating System Concepts.* 550 p. Digital Press, Maynard, 1992.

[**Mit95**] Mitchell, J. Präsentation auf der Video-Produktion *The Spring Distributed, Object-Oriented Operating System*, University Video Communications, Stanford, California, USA, aufgezeichnet im März 1995.

[**Moo90**] Mooney, J.D. Strategies for Supporting Application Portability. *IEEE Computer*, 23(11):59–70, November 1990.

[**Mot85**] Motorola Inc. *MC68020 32-Bit Microprocessor User's Manual.* 216 p. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, second edition, 1985.

[**MRT+90**] Mullender, S.J., Rossum, G.v., Tanenbaum, A.S., Renesse, R.v., and Staveren, H.v. Amoeba – A Distributed Operating System for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[**Mü90a**] Müller, T. Ansätze für den objektorientierten Entwurf von Betriebssystemen. *edv-aspekte 4/90*, pages 24–30, April 1990.

[**Mü90b**] Müller, T. Untersuchung zum Einsatz der objektorientierten Programmierung für den Entwurf und Implementation von Betriebssystemen. Diplomarbeit, 62 S., Technische Universität Chemnitz, Fachbereich Informatik, Juli 1990.

[**Mü92a**] Müller, K. *Objektorientierte Programmierung – Eine Einführung.* Technische Universität Chemnitz, Fachbereich Informatik, Bericht 223, 6. Jahrgang, 16 S., 1992.

[**Mü92b**] Müller, K. *Realisierung verteilter Objektstrukturen mit C++.* Technische Universität Chemnitz, Fachbereich Informatik, Bericht, 83 S., Oktober 1992.

[**Mü92c**] Müller, K. Dynamische Rekonfiguration und Migration in heterogenen Umgebungen. Positionspapier zum Treffen der GI-Fachgruppe "Betriebssysteme" zum Thema "Architektur von Betriebssystemen", Technische Universität Dresden, 5.-6. Oktober 1992.

[**MüKT90**] Müller, K. and Müller, T. An Object-Oriented Architecture Model for Microkernel Based Systems. *Position Paper for the OSF Microkernel Applications Workshop, Grenoble, France, November 27-29*, 1990.

[**MWY91**] Matsuoka, S., Watanabe, T., and Yonezawa, A. Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming. *Proceedings of ECOOP'91*, In [**ECOOP91**]:231–250, July 1991.

[**Neh95**] Nehmer, J. Betriebssysteme – Stand der Forschung und Entwicklungstendenzen. unveröffentlichtes Positionspapier, 51 S., Universität Kaiserslautern 1995.

[**NeX92**] NeXT Computer, Inc. *NeXTSTEP Operating System Software.* Addison-Wesley Publishing Company, Reading, Massachusetts, USA, 1992.

[**NL96**] Necula, G.C. and Lee, P. Safe Kernel Extensions Without Run-Time Checking. *Proceedings of OSDI'96*, In [**OSDI96**]:229–243, October 1996.

[**NP95**] Nimberger, F. and Pohl, H. VM2000. In P. Jilek, (Ed.), *BS2000/OSD – Technische Beschreibung.* Publicis MCD Verlag, Erlangen, 155 S., Siemens Nixdorf Informationssysteme AG, München, Juli 1995.

[**NRTT89**] Nishiguchi, O., Ramamoorthly, C.V., Tsai, W.K., and Tsai, W.T. An Adaptive Hierarchical Routing Protocol. *IEEE Transactions on Computers*, 38(8):1059–1075, August 1989.

[**NS95**] Nehmer, J. and Sturm, P. Generating Dedicated Runtime Platforms for Distributed Applications. *Proceedings of the Fifth Workshop on Future Trends in Distributed Computing Systems FTDCS'95, Korea*, pages 50–55, 1995.

[**Oels88**] Oelsner, T. Anschluß eines 16-bit-PC an 32-bit-Rechner zur Terminalemulation. Diplomarbeit, 67 S., Technische Universität Dresden, Informatik Zentrum, Professur Betriebssysteme, 1988.

[**OM92**] Orr, D.B. and Mecklenburg, R.W. OMOS – An Object Server for Program Execution. *Proceedings of IWOOOS'92, Paris, France, November, 1992*, pages 200–209, 1992.

[**OMG91**] OMG and X/Open. *The Common Object Request Broker: Architecture and Specification, CORBA.* published by Object Management Group (OMG) and X/Open, 177 p., Reading Berkshire RGI 1AX, U.K. and Framingham, Massachusetts, USA, OMG Document Number 91.12.1, Revision 1.1, 1991.

[**OMG95**] OMG. *The Common Object Request Broker: Architecture and Specification, Revision 2.0.* published by Object Management Group (OMG), 463 p., Framingham, Massachusetts, USA, July, 1995.

[**OSDI94**] Lepreau, J., (Ed.). *Operating System Design and Implementation, OSDI'94*, The Usenix Association, Symposium Proceedings, 280 p., Monterey, California, USA, November 14-17, 1994.

[**OSDI96**] Petersen, K. and Zwaenepoel, W., (Eds.). *Second USENIX Symposium on Operating System Design and Implementation, OSDI'96*, Seattle, Washington, USA, October 28-31, 1996. Special Issue of ACM Operating Systems Review, 291 p., Vol. 30, Winter 1996.

[**OSF92**] OSF. *Introduction to OSF DCE: Revision 1.0.* OSF DCE series. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, Open Software Foundation, Inc., Cambridge, Massachusetts, USA, 1992.

[**OSF93**] OSF. *Design of the OSF/1 Operating System.* OSF DCE series. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, Open Software Foundation, Inc., Cambridge, Massachusetts, USA, 1993.

[**Ous90**] Ousterhout, J.K. Why Aren't Operating Systems Getting Faster as Fast as Hardware? *Proceedings of the Usenix Summer Conference, Anaheim, California, USA*, pages 247–256, June 1990.

[**Paps93**] Papsdorf, T. Migration in heterogenen verteilten Umgebungen. Diplomarbeit, 86 S., Technische Universität Chemnitz, Fakultät für Informatik, Professur Betriebssysteme, April 1993.

[**Par72**] Parnas, D.L. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[**Pat93**] Patience, S. Redirecting System Calls in Mach 3.0: An Alternative to the Emulator. In *Proceedings of the Third Usenix Mach Symposium, Santa Fe, New Mexico, USA*, pages 55–74, April 1993.

[**PB96**] Pardyak, P. and Bershad, B.N. Dynamic Binding for an Extensible System. *Proceedings of OSDI'96*, In [**OSDI96**]:201–212, October 1996.

[**Peterson94**] Peterson, L.L., Montz, A.N., Mosberger, D., O'Malley, S.W., Proebsting, T.A., and Hartmann, J.H. Scout: A Communication-Oriented Operating System. Technical Report TR 94-20, Arizona State University, June 1994.

[**PHW76**] Parnas, D.L., Handzel, G., and Würges, H. Design and Specification of the Minimal Subset of an Operating System Family. *IEEE Transactions on Software Engineering*, SE-2, July 1976.

[**Pomb93**] Pomberger, G. and Blaschek, G. *Software Engineering – Prototyping und objektorientierte Software-Entwicklung.* 337 p. Carl Hanser Verlag, München Wien, 1993.

[**PPTT91**] Pike, R., Presotto, D., Thompson, K., and Trickey, H. Plan9, A Distributed System. *Proceedings of the Spring 1991 EurOpen Conference*, pages 43–50, May 1991.

[**PS75**] Parnas, D.L. and Siewiorek, D.P. Use of the Concept of Transparency in the Design of Hierarchically Structured Systems. *Communications of the ACM*, 18(7):401–408, July 1975.

[**PSOS93**] pSOS. *pSOSystem – System Concepts, Release 2.0.* Integrated Systems, Inc., Santa Clara, California, USA, 1993.

[**Pyr96**] Pyramid, Inc., A Siemens-Nixdorf Company. RM1000 – Product Overview. http://www.pyramid.com/, San José, California, USA, April 1996.

[**RAA+88**] Rozier, M., Abrossimov, V., Armand, F, Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C, Rozier, M., Langlois, S., Léonard, P., and Neuhauser, W. CHORUS Distributed Operating Systems. *Computing Systems Journal*, 1(4):305–370, December 1988.

[**RDH+80**] Redell, D.D., Dalal, Y.K., Horsley, T.R., Lauer, H.C., Lynch, W.C., McJones, P.R., Murray, H.G., and Purcell, S.C. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, 23(2):81–91, February 1980.

[**Rich94**] Richter, B. Performance Evaluation von OSF/DCE. Diplomarbeit, 70 S., Technische Universität Chemnitz, Fakultät für Informatik, Professur Betriebssysteme, Juni 1994.

[**Rob96**] Robbins, K.A. and Robbins, S. *Practical UNIX Programming – A Guide to Concurrency, Communication, and Multithreading.* 658 p. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1996.

[**Ros94**] Roscoe, T. Linkage in the Nemesis Single Address Space Operating System. *ACM Operating Systems Review*, 28(4):48–55, October 1994.

[**RT74**] Ritchie, D.M. and Thompson, K. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7):365–375, July 1974.

[**Rumb91**] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. *Object-Oriented Modeling and Design.* 500 p. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1991.

[**Rus91**] Russo, V.F. *An Object-Oriented Operating System.* Dissertation, 154 p., University of Illinois at Urbana-Champaign, 1991.

[**Sak87**] Sakamura, K., (Ed.). *TRON-Project 1987: Open-Architecture Computer Systems*, The Third TRON Project Symposium, Tokyo, November 13, 1987, Proceedings, 308 p. Springer-Verlag, Berlin Heidelberg New York Tokyo, 1987.

[**Sch95a**] Schalm, M. *Verwaltung persistenter Daten in einem verteilten objektbasierten Betriebssystem.* Dissertation, 130 S., Technische Universität Chemnitz, Fakultät für Informatik und Technische Universität Dresden, April 1995.

[**Schim94**] Schimmel, C. *UNIX Systems for Modern Architectures.* Professional Computing Series, 396 p. Addison-Wesley Publishing Company, Reading, Massachusetts, USA, 1994.

[**Schm95**] Schmidt, H. *Dynamisch veränderbare Betriebssystemstrukturen.* Dissertation, 141 S., Universität Potsdam, Mathematisch-Naturwissenschaftliche Fakultät und GMD-FIRST Berlin, Mai 1995.

[**Schö96**] Schöniger, F. Spring: Sun's neuer Kern für objektorientierte, verteilte Anwendungen. Projektarbeit, 27 S., Technische Universität Chemnitz, Fakultät für Informatik, Professur Betriebssysteme, Juli 1996.

[**Schrö89**] Schröder-Preikschat, W. and Zimmer, W., (Eds.). *Progress in Distributed Operating Systems and Distributed Systems*, European Workshop, Berlin, Germany, April 18-19, 1989, Proceedings in: Lecture Notes in Computer Science (LNCS 433), 205 p., Springer-Verlag, Berlin Heidelberg New York Tokyo, 1989.

[**Schrö91**] Schröder-Preikschat, W. and Cordsen, J. Object-Oriented Operating System Design and the Revival of Program Families. *Proceedings of IWOOOS'91*, pages 24–28, October 17-18, 1991, Palo Alto, California, USA 1991.

[**Schrö93**] Schröder-Preikschat, W. Design Principles of Parallel Operating Systems – A Peace Case Study. Technical Report TR-93-020, International Computer Science Institute, Berkeley, California, USA, April 1993.

[**Schrö94**] Schröder-Preikschat. *The Logical Design of Parallel Operating Systems.* Series in Innovative Technology, 370 p. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1994.

[**Schu96**] Schubert, F. Dynamische Adaptierbarkeit in Betriebssystemen auf Basis objektorientierter Methoden. Forschungsbericht, In [**Kal96b**]:19-36, März 1996.

[**SE90**] Stroustrup, B. and Ellis, M.A. *The Annotated C++ Reference Manual, ANSI Base Document.* 447 p. Addison-Wesley Publishing Company, Reading, Massachusetts, USA, 1990.

[**Seltzer94**] Seltzer, M., Endo, Y., Small, C., and Smith, K.A. An Introduction to the VINO Kernel. In "VINO: The 1994 Fall Harvest", Technical Report TR-34-94, Harvard University, 1994.

[**Sha86**] Shaprio, M. Structure and Encapsulation in Distributed Systems: The Proxy Principle. *Proceedings of the Sixth International Conference on Distributed Computing Systems*, pages 198–204, Boston, Massachusetts, USA, May 1986.

[**Sil91**] Silberschatz, A., Peterson, J., and Galvin, P. *Operating System Concepts.* 696 p. Addison-Wesley Publishing Company, Reading, Massachusetts, USA, third edition, 1991.

[**SK89**] Spector, A.Z. and Kazar, M.L. Wide Area File Service and The AFS Experimental System. *Unix Review*, 7(3), March 1989.

[**SM79**] Seawright, L.H. and MacKinnon, R.A. VM/370 - A Study of Multiplicity and Usefulness. *IBM Systems Journal*, 18(1):4–17, 1979.

[**Smi82**] Smith, B.C. *Reflection and Semantics in a Procedural Language.* Dissertation, M.I.T., TR-272, Cambridge, Massachusetts, USA, 1982.

[**Sol92**] Soley, R.M., (Ed.). *Object Management Architecture Guide.* 98 p. published by Object Management Group, OMG TC Document 92.11.1, Revision 2.0, Framingham, Massachusetts, USA, September 1992.

[**Son92**] Sonntag, S. Closing the Gap Between Different Object Models. *Proceedings of IWOOOS'92, Paris, France, November*, pages 378–383, 1992.

[**Son93**] Sonntag, S. *Adaptierbarkeit durch Reflexion.* Dissertation, 103 S., Technische Universität Chemnitz, Fakultät für Informatik und GMD Birlinghoven, Juni 1993.

[**SOSP91**] Lazowska, E.D., (Ed.). *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, Asilomar Conference Center, Pacific Grove, California, USA, December 13-16, 1991. Special Issue of ACM Operating Systems Review, 253 p., 25(5), December 1991.

[**SOSP93**] Liskov, B., (Ed.). *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, The Grove Park Inn and Country Club, Asheville, North Carolina, USA, December 5-8, 1993. Special Issue of ACM Operating Systems Review, 283 p., 27(5), December 1993.

[**SOSP95**] Weiser, M., (Ed.). *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain Ressort, Colorado, USA, December 3-6, 1995. Special Issue of ACM Operating Systems Review, 324p., 29(5), December 1995.

[**SSC94**] Stras, A., Stras, R., and Chrobot, S. Needles and Links or Dealing with Interrupts. Technical Report, 11 S., Kuweit University, Department of Mathematics, 1994.

[**SteiN95**] Steinmetz, R. and Nahrstedt, K. *Multimedia: Computing, Communications & Applications.* Innovative Technology Series, 854 p. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1995.

[**Sti95**] Stiegler, H. Systemübersicht. In P. Jilek, (Ed.), *BS2000/OSD – Technische Beschreibung.* Publicis MCD Verlag, Erlangen, 2. Auflage, 137 S., Siemens Nixdorf Informationssysteme AG, München, August 1995.

[**Suhr95a**] Suhr, A. Eine Lastmaschine zur Qualitätssicherung des Kerns von UNIX System-V/R4. Diplomarbeit, 53 S., Technische Universität Chemnitz, Fakultät für Informatik, Professur Betriebssysteme und Siemens-Nixdorf Informationssysteme AG, Paderborn, August 1995.

[**Suhr95b**] Suhr, A. Plan9 – Ein alternatives Betriebssystem von AT&T. Projektarbeit, 25 S., Technische Universität Chemnitz, Fakultät für Informatik, Professur Betriebssysteme, Januar 1995.

[**Sun96a**] Sun. JavaSoft stellt JavaOS vor. *Sun News*, page 5, September 1996.

[**Sun96b**] Sun. Neu: Auf Java optimierte low-cost Prozessoren. *Sun News*, page 14, März 1996.

[**Sund90**] Sunderam, V.S. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice & Experience*, 2(4):315–339, December 1990.

[**SunSoft94**] SunSoft. *A Spring Collection – A Collection of Papers on the Spring Distributed Operating System.* Sun Microsystems, Inc., Mountain View, California, USA, September 1994.

[**Szy92**] Szyperski, C.A. *Insight ETHOS: On Object-Orientation in Operating Systems.* Informatik-Dissertationen ETH Zürich, Nr. 40, 232 S. Verlag der Fachvereine Zürich, 1992.

[**Tan90**] Tanenbaum, A.S. *Betriebssysteme, Teile 1 und 2.* Carl Hanser Verlag, München Wien, 1990.

[**Tan92**] Tanenbaum, A.S. *Modern Operating Systems.* 728 S. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1992.

[**TB93**] Theaker, C.J. and Brookes, G.R. *Concepts of Operating Systems.* 226 p. The Macmillan Press LTD, Great Britain, 1993.

[**vEick92**] von Eicken, T., Culler, D.E., Goldstein, S.C., and Schauser, K.E. Active Messages: a Mechanism for Integrated Communication and Computation. *Proceedings of the 19th International Symposium on Computer Architecture*, Mai 1992.

[**VW96**] Vogel, R. and Werner, J. Die Hardware-Abstraktionsschicht für *CheOps*. Forschungsbericht, In [**Kal96b**]:49-56, März 1996.

[**WCC+74**] Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., and Pollack, F. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM*, 17(6):337–345, June 1974.

[**Wett91**] Wettstein, H. The Explainable Operating System. *Dagstuhl Workshop Operating Systems of the 90s and Beyond*, In [**KN91**]:247–250, July 1991.

[**Wett93**] Wettstein, H. *Systemarchitektur*. Hanser Studienbücher der Informatik, 514 S. Carl Hanser Verlag, München Wien, 1993.

[**WG95**] Wohlrab, L. and Graupner, S. Machine-Level Class Representation for an Adaptive Object-Oriented Operating System. Forschungsbericht, März 1995.

[**Wiat93**] Wiatrowski, J. Architektur objektorientierter Betriebssysteme. Diplomarbeit, 55 S., Technische Universität Chemnitz, Fakultät für Informatik, Professur Betriebssysteme, Mai 1993.

[**Wir88**] Wirth, N. *Programming in Modula-2*. Springer-Verlag, Berlin Heidelberg New York Tokyo, 1988.

[**Wirth92**] Wirth, N. and Gutknecht, J. *Project OBERON, The Design of an Operating System and Compiler*. ACM Press Books, 548 p. Addison-Wesley Publishing Company, Reading, Massachusetts, USA, 1992.

[**Woh96**] Wohlrab, L. Ruling the complexities of OS design and maintenance using object-oriented and AI technologies. Forschungsbericht, In [**Kal96b**]:37-47, März 1996.

[**Wolf95**] Wolf, L.C. *Resource Management for Distributed Multimedia Systems*. Dissertation, 145 S., Technische Universität Chemnitz, Fakultät für Informatik und IBM-ENC Heidelberg, Dezember 1995, erschienen bei Kluwer Academic Publishers, Boston London Dordrecht, 1996.

[**WSW+94**] Welland, R., Seitz, G., Wang, L.-W., Dyer, L., Harrington, T., and Culbert, D. The Newton Operating System. *Proceedings of the 1994 IEEE Computer Conference*, San Francisco, 1994.

[**WWDS94**] Weiser, M., Welch, B., Demers, A., and Shenker, S. Scheduling for Reduced CPU Energy. *Proceedings of OSDI'94*, In [**OSDI94**]:13–23, November 1994.

[**WY88**] Watanabe, T. and Yonezawa, A. Reflection in an Object-Oriented Concurrent Language. *Proceedings of OOPSLA'88, SIGPLAN Notices, ACM Press*, 23:306–315, September 1988.

[**X/Open87**] X/Open. *X/Open Portability Guide, Issue 2*. X/Open Company, Ltd., Elsevier Science Publishers B.V. (North-Holland), Amsterdam, The Netherlands, 1987.

[**X/Open89**] X/Open. *X/Open Portability Guide, Issue 3*. X/Open Company, Ltd., Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1989.

[**X/Open92**] X/Open. *X/Open Systems and Branded Products: XPG4*. 5 Volumes, X/Open Comp. Ltd., Berkshire, U.K., 1992.

[**Xer81**] The Xerox Learning Research Group. The Smalltalk-80 System. *Byte*, pages 36–48, August 1981.

[**Yok92**] Yokote, Y. The Apertos Reflective Operating System: The Concept and Its Implementation. *Proceedings of OOPSLA'92*, pages 414–434, October 1992.

[**Yok93**] Yokote, Y. Kernel Structuring for Object-Oriented Operating Systems: The Apertos Approach. *Proceedings of ISOTAS'93*, November 1993.

[**Yon90**] Yonezawa, A., (Ed.). *ABCL – An Object-Oriented Concurrent System: Theory, Language, Programming Implementation, and Application*. The MIT Press, Cambridge, Massachusetts, USA, 1990.

[**YT87**] Yonezawa, A. and Tokoro, M., (Eds.). *Object-Oriented Concurrent Programming*. Computer Systems Series, 282 p. The MIT Press, Cambridge, Massachusetts, USA, 1987.

[**YTM91**] Yokote, Y., Teraoka, F., and Mitsuzawa, A. The Muse Object Architecture: A New Operating System Structuring Concept. *ACM Operating Systems Review*, 25(2):22–46, April 1991.

[**Zim93**] Zimmermann, C. and Kraas, A.W. *MACH - Konzepte und Programmierung*. 192 S. Springer-Verlag, Berlin Heidelberg New York Tokyo, 1993.

[**Zim96**] Zimmermann, C., (Ed.). *Advances in Object-Oriented Meta-Level Architectures and Reflection*. Collection of Workshop-Papers: *"Advances in Metaobject Protocols and Reflection"* at ECOOP'95, Aarhus, Denmark, August 7-11, 1995. 352 p., CRC Press, Inc. Lewis Publishers, Boca Raton, Florida, USA, 1996.

[**Zlo91**] Zlotnick, F. *The POSIX.1 Standard: A Programmer's Guide*. 379 p. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, USA, 1991.

Abbildungsverzeichnis

Thesen

1. Eine ***Architektur*** gibt für technische Systeme die *strukturellen*, *methodischen* und *inhaltlichen (funktionalen)* Grundzüge eines Systems an. Sie bildet damit eine wichtige Grundlage für Vorgehens-, Entwurfs- und Implementierungsentscheidungen. Für technische Systeme steht der funktionale Aspekt der Architektur im Vordergrund, welcher sich aus den ***Architekturzielen*** ableitet.

2. ***Anpaßbarkeit*** ist eine Eigenschaft von Elementen, eines Systems, eines Konzepts, einer Methode oder Struktur, einer Grundidee oder auch einer Architektur, in eine neue Umgebung mit veränderten Anforderungen und Eigenschaften übertragbar zu sein. ***Anpassung*** ist der Prozeß der Angleichung nichtpaßfähiger Eigenschaften an die Erfordernisse der Zielumgebung. Anpassungsfähigkeit unterstützt damit in entscheidender Weise die *Wiederverwendung* von Elementen, Konzepten, Methoden oder von Architekturmerkmalen.

3. Anwendungsanpaßbarkeit und ***Anwendungsorientierung*** sind, wie für andere Produkt- und Dienstleistungsbereiche, auch für Betriebssysteme in zunehmender Weise bestimmende Trends, die gegenläufig zur Entwicklung übergreifender Universalbetriebssysteme sind.

4. Die Steuerungssoftware für vielfältigste Geräte wird komplexer. Dies führt zu einem Bedarf, infrastrukturelle Aufgaben der Ressourcenverwaltung und Ablaufsteuerung von den Anwendungsaufgaben zu trennen. Die entstehenden ***"Softwareinfrastrukturen"*** sind dabei nichts anderes als kleine und kleinste Betriebssysteme, die für Anwendungssoftware in sehr speziellen Einsatzumgebungen zugeschnitten sind. Es wiederholt sich hier eine Entwicklung, die in den sechziger Jahren bei universellen Rechenanlagen mit der Entkopplung von Anwendungs- und Betriebssystemen stattfand.

5. Die Ausführung von Anpassungen an Betriebssystemen bzw. die Herstellung neuer Betriebssysteme wird immer mehr zum Hersteller von Anwendungen verlagert. Die Grenze zwischen Anwendungssystem und Betriebssystem und damit die traditionelle Rollenverteilung von Anwendungs- und (wenigen) Betriebssystemherstellern wird unschärfer.

6. Die Anwendung von Betriebssystemtechnologien oder (allgemeiner) die Abgrenzung, Anpassung und Herstellung von Softwareinfrastrukturen wird auch eine größere Bedeutung in der ***Softwaretechnologie*** erlangen. Anwendungssoftware ist auf Basis geeigneter und ggf. speziell hergestellter bzw. zugeschnittener Infrastrukturen leichter zu entwickeln und zu testen. Generelle Aufgaben können in der Infrastruktur konzentriert werden, und es lassen sich bessere ***Ausführungseigenschaften*** für Anwendungen erreichen.

7. Die Softwareherstellung wird zunehmend von ***"Prozeßorientierung"*** und damit von Betriebssystemtechnologie geprägt sein. Die Ursache liegt im wachsenden Aufkommen paralleler und verteilter Anwendungen, deren Grundeinheiten parallel ablaufende Prozesse sind, die über verschiedene, auch nichtsynchrone, Interaktionsszenarien kooperieren.

8. Anwendungsanpaßbarkeit setzt die ***Offenheit*** des Bezugsbereichs voraus. Für Betriebssysteme genügt Offenheit allein jedoch nicht, wenn kein Konzept und keine geeignete (anwendungsanpaßbare) ***Architektur*** dahintersteht, die es idealerweise ermöglichen sollte, Anpassungen potentiell in allen Systembereichen mit vertretbarem Aufwand auszuführen.

9. Der oft gebrauchte Begriff ***Transparenz*** sollte nicht in dem Sinne umgesetzt werden, komplizierte innere Strukturen durch weitere Komplexität zu verdecken, sondern auch im Inneren von Systemen ***durchschaubare***, offengelegte und zugängliche Strukturen zu schaffen. Dies ist ein immer wieder anzustrebendes Ideal für den Entwurf und die Herstellung von Systemen, vor allem auch von Betriebssystemen bzw. von Softwareinfrastrukturen.

10. Bei einer durchgängigen Systemarchitektur über alle Schichten eines Systems spielt die Wechselbeziehung zwischen Uniformität und Differenzierbarkeit die wesentliche Rolle. Es ist abzuwägen, welche Merkmale der Architektur ***universeller*** Natur und damit für andere Systemschichten oder für andere Systeme übertragbar sind, welche Elemente für einen konkreten Zielbereich ***speziell*** zugeschnitten sind oder ob es auch ***spezialisierbare*** Elemente zur Anpassung universeller Architekturmerkmale gibt.

11. Eine anwendungsanpaßbare Systemarchitektur sollte das ***Gesamtsystem***, in einer durchgängigen, möglichst einheitlichen Weise erfassen, um Strukturbrüche aufzuheben:
    - zwischen (prozeßorientierten) Anwendungs- und (monolithischen) Betriebssystemen,
    - innerhalb von Betriebssystemen (z.B. zur Unterbrechungsverarbeitung) und
    - zwischen der konzeptionellen Ebene des Entwurfs, der Implementierung und ablaufenden (Betriebs-) Systemen, d.h. zwischen ***Software- und Ablaufstruktur***.

12. Wenn ein ablaufendes System der Gegenstandsbereich für Anpassungen ist, müssen auch dessen Elemente die Architektur bestimmen. Es geht primär um ***Verarbeitungsprozesse***, um ihre Erzeugung und Steuerung, nicht in *erster Linie* um Fragen der Herstellung der Software für diese Elemente. Dies folgt in einem *zweiten Schritt*, der nicht minder wichtig, aber der Architektur des Gesamtsystems untergeordnet ist.

13. Die Abgrenzung von Infrastrukturen und Anwendungsbereichen führt konsequenterweise zu hierarchischen oder ***rekursiven Schichten*** als dominierendem Architekturmerkmal. Schichtgrenzen werden festgelegt, wenn durch Verarbeitungsprozesse von Elementen einer Infrastrukturschicht neue Elemente in anderen Ausprägungen und mit anderen, höherwertigen Ausführungseigenschaften in einer darüberliegenden Schicht hergestellt werden.

14. Mit dem Strukturelement ***Instanz*** lassen sich semantisch zusammengehörige Verarbeitungsvorgänge sowohl konzeptionell als auch in der Realität des ablaufenden Systems in identifizierbaren Verarbeitungselementen zusammenfassen. Die Funktion von Instanzen ist die selbständige Ausführung von ***Diensten***. Das in vielen technischen Bereichen angewandte Gliederungsprinzip selbständiger Teilsysteme wird damit für alle Schichten anwendbar, und es lassen sich höherwertige Dienste von Teilsystemen herstellen.

15. Neben der Rekursion von Systemschichten spielt auch die Kapselungseigenschaft für innere Verarbeitungen eine wichtige Rolle. Die hierarchische Enthaltensein-Relation von Elementen führt zu einer Aggregationsrekursion. ***Beide Arten von Rekursion*** sind aufgrund der Einführung von ***Steuerungsebenen*** wichtige Strukturmittel für ablaufende Systeme.

16. Instanzen grenzen innerhalb einer Schicht semantisch zusammengehörige Teilverarbeitungen in identifizierbaren Elementen ab. Sie kapseln eigene innere Verarbeitungs- und Steuerungszustände, besitzen für ihre autonome Arbeitsweise mindestens einen eigenen inneren

Aktivitätsträger und eine eigene (Programm-) Steuerung. Instanzen sind das hier favorisierte Strukturelement neben anderen Elementen, z.B. für gemeinsame Zustände.
Die Kapselungseigenschaft für innere Verarbeitungen in Instanzen ist dabei nicht unmittelbar mit Zugriffsschutz in Beziehung zu bringen. Zugriffsschutz ist eine spezielle Eigenschaft des Ausführungsmodells, wogegen Kapselung im Sinne der Abgrenzung semantisch zusammengehöriger Verarbeitungsvorgänge ein generelles Architekturmerkmal ist.

17. Das Konzept von Instanzen in allen Systemschichten reflektiert genau die im Anwendungsbereich seit langem bekannten und akzeptierten Grundelemente des *Client-Server-Modells*. Es war eine wesentliche Intention dieser Arbeit, die bekannte und ***vertraute Welt des Anwendungsbereichs*** konsequent auf alle Schichten des Systems zu übertragen.

18. Die in dieser Dissertation vorgeschlagene Architektur basiert damit auf einer Verallgemeinerung und Kombination ***anerkannter Grundkonzepte***:
    - dem *Client-Server-Modell* in Verbindung mit
    - dem Modell *rekursiver Schichten* bzw. dem Modell *Abstrakter Maschinen*.

    Aus dieser Kombination folgt einerseits eine Begründung der Folgerichtigkeit, und es ergeben sich andererseits *Synergien* für die Architektur, einschließlich der gewählten Vorgehensweise über Generalisierung und Spezialisierung von Architekturmerkmalen.

19. Die in Systemen anzutreffende Zweiteilung in einen geschlossenen System- und einen offenen Anwendungsbereich kann durch die ***Integration in eine Gesamtarchitektur*** rekursiver, offener Schichten mit weitreichenden Konsequenzen aufgehoben werden:
    - durch potentielle Offenheit aller Schichten werden Infrastruktureigenschaften anwendungsorientiert anpaßbar; bislang starre Dienste und Ausführungsmodelle der Infrastruktur werden gestaltbar;
    - es eröffnet sich die Option für ein *top-down* Vorgehen bei der Anwendungsherstellung, d.h., Infrastruktur kann anwendungsgerecht angepaßt bzw. hergestellt werden;
    - die Rollenverteilung zwischen System- und Anwendungsherstellung wird unschärfer.

20. Für dieses Ziel muß das Grundkonzept von konkreten Ausführungseigenschaften in jeweiligen Schichten oder Zielbereichen getrennt werden.
    Universelle Merkmale werden in einer ***generalisierten Architektur*** festgeschrieben, aus welcher dann unter Bezugnahme auf die konkreten Bedingungen *Ausprägungsvarianten* in Form von Schichten und Elementen abgeleitet und durch eine jeweilige Infrastruktur hergestellte *Dienste* und *Ausführungseigenschaften* für die Art der Elemente, deren Arbeitsweise und Zusammenwirken zugeordnet werden.

21. Erst durch diese Trennung wird das notwendige hohe Maß an ***"Skalierbarkeit"*** der Architekturmerkmale erreicht, um sie tatsächlich in allen Systemschichten bzw. in unterschiedlichen Systemen anwenden und jeweils herrschenden Bedingungen anpassen zu können.

22. In dieser Kombination kommt sowohl das Ziel nach ***Homogenität*** einer durchgängigen, für viele Zielumgebungen passenden Architektur als auch die ***Differenzierbarkeit*** und damit Anpassungsfähigkeit an spezielle Umgebungen (Schichten, Systeme) zum Ausdruck.

23. Es muß die ***Balance*** zwischen Homogenität und Differenzierbarkeit für ein anwendungsanpaßbares, durchgängiges Architekturkonzept hergestellt werden. Bei vielen anderen Ansätzen in der Forschung wird dagegen oft Uniformität als alleiniges Ziel angestrebt.

24. Das wird vor allem beim Trend nach ***Objektorientierung*** in Betriebssystemen deutlich. Die Übertragung einer softwaretechnischen Methode in die (dynamische) Welt der Betriebssysteme muß beim heutigen Stand der Entwicklung kritisch betrachtet werden.
Die Idealvorstellung *"everything are objects"* hat sich als ungeeignet erwiesen, wenn grundverschiedene Elemente oder Eigenschaften nicht mehr begrifflich unterschieden werden, d.h. Prozesse, Programme, Nachrichten, Adreßräume oder Rechner "im Prinzip Objekte" sind.

25. Objektorientierte Methoden (Klassenhierarchien, Datenkapselung, Vererbung u.a.) sind vorteilhaft für die ***Herstellung der Software*** von Elementen eines Systems anwendbar. Die Konzepte zur objektorientierten Herstellung der *Software für Elemente* sind aber nicht mit den *Elementen selbst* und ihren Eigenschaften gleichzusetzen.

26. Parallele Prozesse, nichtsynchrone Interaktionsszenarien, unterschiedliche Ausprägungs- und Ausführungseigenschaften von Elementen in verschiedenen Schichten oder andere für Betriebssysteme wesentlichen Merkmale sind mit objektorientierten Methoden nicht modellierbar. Die Diskrepanz und die ***begriffliche Vermischung*** *"beider Welten":*

    konzeptionelle Ebene $\longleftrightarrow$ *ablaufendes System*

    ist nach wie vor ein schwerwiegendes Problem, nicht nur in Betriebssystemen.

27. Objekte der konzeptionellen Ebene (Analyse, Entwurf, Programmierung) müssen in einer geeigneten Weise in das ablaufende System abgebildet werden, um dort Verarbeitungsprozesse zu steuern, worin deren letztendlicher Zweck besteht.
Diese ***Objekt-Implementierung*** geht für verteilte oder parallele (Betriebs-) Systeme über das triviale Ablauf- und Ausführungsmodell vieler bekannter objektorientierter Sprachen hinaus (Objekte=Daten, globales Programm, eine Aktivität im Gesamtsystem u.a.). Für komplexere Umgebungen muß es daher eine explizite Abbildung geben:

    Objekt(e) $\rightarrow$ *Ausführungseinheit(en).*

28. Instanzen eignen sich als ***Ausführungseinheiten*** für Objekte. *CORBA* bietet bereits einige Abbildungsvarianten. Es besteht jedoch noch Forschungsbedarf, den Strukturbruch zwischen der konzeptionellen Ebene und der Realität ablaufender Systeme zu überwinden.

29. In der Gestaltbarkeit von Ausführungseigenschaften liegt ein großes ***Leistungs-*** und ***Skalierungspotential***, wenn je nach Anforderung wahlweise einfachere und effizientere Ausführungsmodelle zu Lasten starrer, "komfortabler" (=teurer) Dienste und Ausführungsmodelle gegenwärtiger Universalbetriebssysteme eingesetzt werden können. Dies kann auch Eigenschaften einschließen, die heute als fundamental angesehen werden, z.B. virtueller Speicher oder auch getrennte Adreßräume. Diese mögliche Quelle für Leistungsgewinn wird heute wegen fehlender Anpassungsmöglichkeiten nicht erschlossen. Auch der Aspekt der Skalierbarkeit, d.h. der Übertragbarkeit in andere Systemumgebungen, ist ein wesentliches Kriterium.

30. Der ausgebliebene Durchbruch bei an *Mach* orientierten ***μ-Kernen*** hängt in entscheidender Weise mit der Frage der Ausführungsmodelle zusammen. Der Kern wurde funktional abgerüstet, jedoch wurde das teure ($\rightarrow$ *"komfortable"*) Ausführungsmodell von *Unix* übernommen. Die durch ausgelagerte Funktionen größere Kommunikationsaktivität führt bei etwa gleichen Ausführungskosten zu einer ***negativen Leistungsbilanz***, vor allem dann, wenn wegen Kompatibilität Vorbildsysteme ($\rightarrow$ *Unix*) auf Basis von μ-Kernen emuliert werden. Die Leistungswerte der (meist monolithischen) Vorbildsysteme sind auf diese Weise natürlich nicht erreichbar. Diese ernüchternde Erkenntnis wird heute allgemein anerkannt.

31. Nur wenn die Ausführungseigenschaften der beteiligten Elementen und ihre Interaktion in dem Maße vereinfacht werden, daß der strukturell bedingte Mehraufwand durch Beschleunigung (über-) kompensiert wird, ergibt sich ein ***Leistungsgewinn***. Ein Beispiel aus einem anderen Gebiet, bei dem das für einen Entwicklungsabschnitt gelang, waren *RISC*-Prozessoren.

32. Der in jüngerer Zeit mit 64-Bit Adressierungsbreite aufkommende Trend "*Single Address Space Operating Systems*" bietet beispielsweise auch die Option zur Vereinfachung des durchaus aufwendigen Ausführungsmodells getrennter Adreßräume, deren Schutzfunktion auch auf andere Weise hergestellt werden kann.

33. Das Ziel sind folglich an der Anwendung orientierte, dem jeweiligen Zweck entsprechende ***"skalierbare" Dienste*** *und* ***Ausführungseigenschaften***, indem Infrastruktur im Sinne eines *top-down* Vorgehens bei der Herstellung von Anwendungen ebenfalls zu einem Anpassungs- und Entwicklungsgegenstand werden kann.

34. Heutige Betriebssysteme sind auf Basis der gegenwärtigen Rechnerarchitekturen bezüglich Effizienz in gewisser Weise ***optimal implementiert***. Es besteht ein Zusammenhang zwischen Monoprozessorsystem und monolithischem Betriebssystem. Vorteilhaftere Software-Architekturen bedingen heute in der Regel einen höheren Ablaufaufwand.

35. In der ***Praxis*** vollziehen sich strukturelle Übergänge erst dann, wenn ein tatsächlicher Bedarf dafür besteht. So war der Auslöser für den Übergang zu einer Prozeßstruktur im Inneren des *Unix*-Kerns das Aufkommen von Multiprozessormaschinen, obwohl Dijkstra eine Prozeßstruktur für Betriebssysteme bereits vor gut 30 Jahren vorgeschlagen hatte.

36. Die Motivation für ***Forschungsvorhaben*** zu strukturellen und Architekturfragen von Betriebs- und Anwendungssystemen leitet sich vor allem aus zwei Gesichtspunkten ab.
    - Künftige Rechnerarchitekturen werden andere Eigenschaften aufweisen, welche heute bestehende Ressourcenengpässe überwinden, so daß diese in der Grundlagenforschung auch vorweg genommen werden können (z.B. mehrfache, asymmetrisch zugeordnete, ggf. spezialisierte Prozessoren).
    - Durch die wahlweise Abrüstbarkeit der Ausführungsmodelle bzw. durch die Spezialisierbarkeit von Softwareinfrastruktur für bestimmte Anwendungen können sich aber auch auf der gegenwärtigen Basis vorteilhafte Eigenschaften ergeben.

    Gerade der zuletzt genannte Aspekt relativiert den heute real vorhandenen Effizienznachteil neuer Software-Architekturen dahingehend, daß sich in bezug auf die ***Effizienz eines Gesamtsystems*** durch Spezialisierung der Infrastruktur auch Leistungsvorteile ergeben können. Im Gegenzug können suboptimierte, monolithische Betriebssystem-Kerne (vgl. *μ-Kerne*) mit festen Ausführungseigenschaften im Kontext eines Gesamtsystems auch eine negative Leistungsbilanz bewirken. Die Untersuchung dieser Zusammenhänge kann als eine weiterführende Forschungsrichtung angesehen werden, die nicht im Rahmen dieser Dissertation behandelt wurde.

37. Das *CHEOPS*-***Projekt*** und die Entwicklung des *CHEOPS*-Kerns stellt daher Fragen in den Vordergrund, die nicht auf die Re-Implementierung bestehender, bekannter Arbeitsprinzipien abzielen, sondern es sollen ***neue Prinzipien*** untersucht werden, auch wenn diese im direkten Effizienzvergleich vorerst nicht bestehen können. Eine bestmögliche Implementierung wird natürlich angestrebt, aber nicht unter Aufgabe der primären Untersuchungs- und Architekturziele. Der *CHEOPS*-Kern sollte bewußt frei von dem sonst oft bestehenden Zwang nach zielverfälschenden Kompromissen aufgrund von Effizienz- und Kompatibilitätskriterien gehalten werden. Für ein Forschungsvorhaben ist dies legitim.

38. Die Gestaltung der Schicht der ***Unterbrechungsverarbeitung*** mit einer speziellen Art von Instanzen – *iproc's* – ist ein Beispiel dafür. Mit *iproc's* kann gezeigt werden, daß auch dieser Bereich anhand der hier favorisierten Verarbeitungsform von Instanzen gestaltet werden kann und damit auch eine konsequente, durchgängige Umsetzung des Architekturkonzepts für alle Schichten eines Systems möglich ist. Instanzen in höheren Schichten gibt es bereits in verschiedenen Systemen, z.B. bei *In-Kernel Servern*. Für *iproc's* folgt aber ein zusätzlicher Ablaufaufwand aus der nötigen (Software-) Infrastrukturschicht *ictrl*.

39. Neben den strukturellen Vorzügen ergeben sich durch instanzbasierte Unterbrechungsbehandlungen auch ***vorteilhafte Ablaufeigenschaften***, die für bestimmte Anwendungsfälle neben dem Architekturaspekt sinnvoll angewandt werden können.

40. Instanzen zur Behandlung von Unterbrechungssignalen werden im Prinzip wie andere Instanzen verwaltet und erhalten dadurch ähnliche Ablaufeigenschaften. Durch Abgrenzung in einer separaten Schicht mit einer eigenen Infrastruktur wird jedoch gleichzeitig der Spezifik der Unterbrechungsverarbeitung Rechnung getragen.

41. Darin zeigt sich die in dieser Dissertation angestrebte und in der Architektur umgesetzte ***Dualität*** zwischen Differenzierbarkeit und Spezialisierung von durchgängig anwendbaren Struktur- und Architekturmerkmalen.

42. Insgesamt kann der Schluß gezogen werden, daß sich mit der hier vorgeschlagenen und untersuchten Architektur ein größeres Maß an Übertragbarkeit, Skalierbarkeit und Offenheit und damit auch an Anwendungsanpaßbarkeit erreichen läßt, wenngleich nicht alle Probleme gelöst sind und sich auch neue Fragen ergeben, die weiterführende Arbeiten begründen. Diese werden zum Teil auch im Rahmen der Fortsetzung des *CHEOPS*-Projekts untersucht:

    - Umsetzbarkeit und Praktikabilität verschiedener Ausführungsmodelle für Anwendungen und ein möglicher Leistungsgewinn im Kontext eines Gesamtsystems,
    - Zugriffschutz zwischen Anwendungen auch ohne getrennte Adreßräume,
    - Wirkungen dynamischer Austauschbarkeit von Instanzen in der Infrastruktur,
    - Abbildung von Elementen objektorientierter Sprachen in ablaufende Systeme,
    - feiner granulare dynamische Austauschbarkeit von Objekten innerhalb von Instanzen,
    - dynamisches Rekonfigurierungsmanagement für adaptives Verhalten.

Lebenslauf und wissenschaftlicher Werdegang

Dipl. Inf. Sven Graupner

---

23. Januar 1967	in Chemnitz geboren
1973 – 8/1985	Polytechnische Oberschule "Georg Weerth" und Erweiterte Oberschule "Friedrich Engels" in Chemnitz, Abitur
9/85 – 8/86	einjähriges Praktikum im Forschungszentrum des Werkzeugmaschinenbaus "Fritz Heckert", Chemnitz
9/86 – 2/90	Studium der Informatik an der Technischen Universität Chemnitz
3/90 – 7/90	Studium der Informatik an der Universität Stuttgart
8/90 – 9/90	achtwöchiges Praktikum im Elektronik Meßzentrum der Mercedes-Benz AG, Stuttgart
10/90 – 8/91	Technische Universität Chemnitz, Studienabschluß Diplom der Informatik (Prädikat "sehr gut")
seit 9/91	wissenschaftlicher Assistent an der Professur Betriebssysteme der Technischen Universität Chemnitz Forschungsschwerpunkt: Anpaßbare Betriebssysteme
11/94	vierwöchiger Aufenthalt am International Computer Science Institute (ICSI), Berkeley, USA
Fremdsprachen:	Englisch (SKA 2a), Russisch (SKA 2a), Tschechisch

---