KERNFORSCHUNGSZENTRUM

KARLSRUHE

September 1973 KFK 1866

Institut für Datenverarbeitung in der Technik

Dispatcher-Elementarfunktionen für symmetrische Mehrprozessor-DV-Systeme

J. Nehmer

GESELLSCHAFT
FÜR
KERNFORSCHUNG M.B.H.

KARLSRUHE

GESELLSCHAFT FÜR KERNFORSCHUNG M. B. H.
KARLSRUHE

KERNFORSCHUNGSZENTRUM KARLSRUHE

KFK 1866

Institut für Datenverarbeitung in der Technik

Dispatcher-Elementarfunktionen für symmetrische Mehrprozessor-DV-Systeme

Jürgen Nehmer

von der Fakultät für Informatik der Universität Karlsruhe genehmigte Dissertation

Gesellschaft für Kernforschung mbH, Karlsruhe

Kurzfassung

In dem Bericht wird auf der Basis einer Schichtenstruktur für Betriebsorganisationen ein universeller Satz von Dispatcher-Elementarfunktionen definiert und verschiedene Implementierungen entwickelt.

Die Implementierungsalternativen werden in Simulationsversuchen auf ihr dynamisches Verhalten untersucht, wobei dem Problem der Prozessorkonflikte besondere Beachtung gewidmet wird.

Dispatcher Primitives for Symmetric Multiprocessor Systems

Abstact

This report describes on the basis of a layer structure for operating systems a universal set of dispatcher primitives. Some implementation alternatives will be compared by simulation, spending special attention to the problem of processor conflicts.

Inhaltsverzeichnis Seite

1. Erläuterung der Aufgabenstellung

Die Fortschritte, die in den letzten Jahren auf dem Gebiet der Betriebsorganisation von DV-Systemen erzielt wurden, nehmen sich vergleichsweise bescheiden aus, mißt man sie z. B. mit technologischen Erfolgen auf dem Halbleiter-Sektor. Neben einer Vielzahl partieller Ergebnisse, die insgesamt zu einer Verfeinerung der konstruktiven Hilfsmittel beitrugen, wurden in jüngster Zeit zunehmende Anstrengungen zur Entwicklung geeigneter Generierungsverfahren für Betriebssysteme unternommen [13, 14, 32, 42, 44]. Sie stellen den Versuch dar, auch ohne Existenz einer Strukturtheorie für Betriebssysteme Hilfsmittel für ihre rationelle Produktion zu entwickeln. Dagegen sind nur wenige Ansätze zu verallgemeinernden Konzepten von Betriebsorganisationen bekannt, die als Ausgangspunkt einer Standardisierung dienen könnten. Erschwerend wirkt sich hier vor allem die schnelle technologische Entwicklung aus, die eine Stabilisierung der Strukturen - auch auf dem abhängigen Bereich der maschinennahen Software - für die nahe Zukunft nicht erkennen läßt.

Eine der richtungsweisenden Arbeiten auf dem Gebiet der Betriebsorganisationen stammt von Saltzer [39], dessen 'Traffic Controller' wesentliche Züge eines generalisierten Betriebssystemkerns aufweist. Seine entscheidende Idee besteht darin, die konfigurationsabhängige Hardware, die sich aus Prozessoren, Arbeitsspeichermodulen und einer Hierarchie von Hintergrundspeichern zusammensetzt, auf ein konfigurationsinvariantes System von Pseudoprozessoren mit zugeordneten virtuellen Arbeitsspeichern konstanter Größe und einem geräteunabhängigen File-System abzubilden (Abb. 1). Entsprechend setzt sich der 'Traffic Controller' Saltzers aus drei Hauptkomponenten zusammen, die in eine Reihe elementarer Betriebssystemfunktionen aufgegliedert wurden:

- dem Prozessor-Multiplexer,
- dem Memory-Multiplexer,
- dem File-System.

Saltzer hatte bei seinen Überlegungen vornehmlich die Belange von Teilnehmersystemen im Auge und paßte die Funktionen an die Architektur der GE 645 von Honeywell an, die als Zielmaschine des MULTICS-Systems diente [2, 9, 17].
Die angegebenen Verfahren für das Prozess-Switching über Bibliotheksroutinen, die im Adreßraum jedes Prozesses liegen, lassen sich effektiv nur auf einer Paging-Maschine mit multidimensionalem virtuellen Speicher vom Typ der GE 645 durchführen. Ebenso erscheint die feste Einbettung von Scheduler-Aufrufen vor dem Umschalten der Prozesse aus einem Wartezustand in dieser Form nur für Teilnehmersysteme oder DV-Anlagen mit verwandtem Betriebsverhalten praktikabel.

Die vorliegende Arbeit versteht sich als konsequente Weiterführung der bisherigen Ansätze in Richtung einer Standardisierung von Betriebssoftware.
Erste Voraussetzung ist die Definition eines strategiefreien Betriebssystemkerns, der universell einsetzbar ist und als Basis sowohl für durchsatzorientierte Stapelverarbeitungssysteme, Teilnehmersysteme als auch Realzeitsysteme dienen kann. Des weiteren sollten die Hardware-Anforderungen für die Realisierung eines derartigen Betriebssystemkerns auf ein Minimum reduziert werden, um ihn einem breiteren Spektrum von Rechnertypen zugänglich zu machen. Besonders erwähnt sei die Klasse der Klein- und Mittelrechner, die gewöhnlich nicht über virtuelle Adressierungstechniken verfügen.

Mit diesen Randbedingungen wird im ersten Teil dieser Arbeit das strukturelle Skelett einer allgemeinen Betriebsorganisation erarbeitet, in das die 'Elementarfunktionen des Dispatchers', die ihrerseits nur einen Ausschnitt des Betriebssystemkerns bilden, eingebettet sind.
Als Ausgangsbasis einer Organisationsstruktur dient die erstmals von Dijkstra angegebene und im THE-System [4] praktizierte schichtenweise Gliederung von Programmsystemen, deren Leistungsfähigkeit durch eine Reihe nachfolgender Systeme

[15, 19, 26, 40] bestätigt werden konnte. In einem Verfeinerungsprozeß wird die hardwarenächste Schicht in eine Ebene der Interruptroutinen und eine Ebene der Elementarfunktionen untergliedert und der Dispatcher als Teilsystem darin lokalisiert.

Im zweiten Teil der Arbeit werden die Grundfunktionen des Dispatchers entwickelt und vier alternative Implementierungen für symmetrische Mehrprozess rsysteme angegeben, die mit Methoden der Simulation auf ihr dynamisches Verhalten untersucht werden. Die Auswahl der Entwürfe wurde maßgeblich durch die Forderung nach einer Minimierung der Prozessor-Konflikte im Dispatcher bestimmt, die einen kritischen Systemengpaß verursachen.

Die alternativen Implementierungen aller Dispatcherfunktionen fanden letztlich ihren Niederschlag in einem PL/1-Programmpaket, das gemeinsam mit den wichtigsten Simulationsresultaten im Anhang dieser Arbeit in Form eines Konstruktionshandbuchs zusammengefaßt ist. Die Programme sind leicht lesbar und auf Grund des hohen Detaillierungsgrades direkt in eine Assembler- oder Betriebssystem-Implementierungssprache übertragbar. Sie sind Ausdruck der Erfahrung des Verfassers, daß die Schwierigkeiten in der Entwicklung von komplexen Programmsystemen vor allem im Detail liegen und sollen deshalb eine wirksame Hilfe für den Implementierer darstellen.

2. Der Rahmen einer allgemeinen Betriebsorganisation

2.1 Das Schichtenmodell nach Dijkstra

Abb. 2 zeigt in schematischer Form die schichtenweise Gliederung des gesamten Programmkomplexes einer Betriebsorganisation. Auf dem höchsten Niveau sind die Benutzerprozesse angesiedelt, während alle darunterliegenden Schichten das eigentliche Betriebssystem bilden. Die hardwarenächste Schicht wird oft als Systemnukleus bezeichnet [20], während alle darüberliegenden Ebenen des Betriebssystems den 'Supervisor' bilden. Goos [16] bezeichnet in treffender Weise die Schnittstelle zwischen zwei aufeinanderfolgenden Schichten als 'Pseudohardware' einer virtuellen Maschine. Die Mächtigkeit des Instruktionsvorrats einer Pseudohardware wird durch den (u. U. eingeschränkten) Befehlssatz der realen Maschine und einen zusätzlichen Satz von Funktionen bestimmt (Pseudoinstruktionen), die die Serviceleistung der darunterliegenden Schichten des Betriebssystems repräsentieren. Der Instruktionsvorrat des Systemnukleus als unterster Schicht des Betriebssystems stützt sich ausschließlich auf den der realen Maschine, während mit aufsteigender Schichtung die Mächtigkeit des Instruktionsvorrats wächst.

Voraussetzung für die Gültigkeit dieses Modells ist die gleichartige Behandlung von realen und Pseudo-Maschineninstruktionen durch eine virtuelle Maschine, die z. B. bei konventionellen Systemen die sequentielle Abarbeitung einer Instruktionsfolge erfordert. Abweichungen von dieser Regel, die hauptsächlich aus Effizienzgründen anzutreffen sind, werden - ebenso wie deren Auswirkungen - hier jedoch nicht näher diskutiert.

In den Schichten oberhalb des Nukleus werden alle Aufgaben gewöhnlich einheitlich in Form sequentieller Prozesse organisiert. Aus der Vielzahl existierender Definitionen [3, 6, 39] wird hier die von Saltzer angegebene benutzt: ein Prozess ist ein Programm, das sich in der Ausführung durch einen Pseudoprozessor befindet.

Es wird für jeden Prozeß im System ein Pseudoprozessor bereitgestellt (1 : 1 - Abbildung), der für die Bearbeitung der Programmfolge zuständig ist. Die Aufgabe der Prozessor-Zuteilung zu den existierenden Pseudoprozessoren ist dann offenbar eine Basisfunktion des Nukleus, auf die alle Prozesse angewiesen sind. Dieser Vorgang wird in der Literatur oft mit 'Dispatching' bezeichnet.

Pseudoprozessoren werden für betriebssysteminterne Verwaltungszwecke durch entsprechende Datenstrukturen beschrieben und können in ihrer Mächtigkeit erheblich von den realen Prozessoren des Systems abweichen. Ein spezifisches Merkmal von Pseudoprozessoren ist gewöhnlich die Unfähigkeit, externe Interrupts zu verarbeiten. Sie setzt voraus, daß sie außerhalb der Prozesse in einem zentralen Systemteil, dem Interrupthandler, abgehandelt werden und stellt für homogene Systeme (in denen jeder Interrupt dieselbe Bedeutung für alle Prozesse hat) eine sinnvolle Restriktion dar.

Es scheint vernünftig, diesen zentralen Systemdienst ebenfalls dem Nukleus zuzuordnen, der damit als Koordinator zur Außenwelt des DV-Systems fungiert und auch den eigentlichen Ein/Ausgabeverkehr kontrolliert.
Eine Erweiterung dieser Philosophie ist notwendig, wenn unter Regie eines Betriebssystems mehrere, in der Struktur voneinander abweichende Hardwaresysteme mit individuellen Betriebsorganisationen emuliert werden [28]. Überlegungen in dieser Richtung sind jedoch nicht Gegenstand dieser Arbeit.

Neben diesen geschilderten Eigenschaften der Prozesse, die nahezu zwangsläufig einen zentralen Dispatcher und Interrupthandler im Nukleus voraussetzen, werden Prozesse in der Regel mit Synchronisations- und Kommunikationshilfen ausgestattet, die eine Koordination der Zugriffe auf gemeinsame Daten und bei der Kommunikation zwischen Prozessen ermöglichen.

Es ist zwingend, auch diesen, allen Prozessen gleichberechtigt zur Verfügung stehenden Systemdienst im Nukleus zu konzentrieren.

Aus der Beschreibung der wesentlichen Eigenschaften und der Mächtigkeit von Prozessen wurden oben unmittelbar die minimalen funktionellen Anforderungen an einen Nukleus gewonnen. Er enthält danach:

- die Primitiv-Ein/Ausgabe,
- den Interrupthandler,
- den Dispatcher,
- die Datensynchronisation und Prozesskommunikation.

Ein Vergleich mit dem 'Traffic Controller' Saltzers zeigt, daß die Funktionen des Memory-Multiplexers, die integrierter Bestandteil des 'Traffic Controllers' sind, im Nukleus nicht enthalten sind. Sie werden bei schichtenweise organisierten Betriebsorganisationen gewöhnlich in die erste Schicht des Supervisors verlegt, so daß im Nukleus ausschließlich die ablaufbezogenen Funktionen zurückbleiben.
Die Vorteile einer derartigen hierarchischen Organisationsstruktur wurden schon vielfach beschrieben [5, 36, 37] und sind im betrachteten Fall:

- die Supervisorprozesse der Speicherverwaltung können selbst auf Nukleusfunktionen zurückgreifen,
- der Nukleus wird kompakter. Betriebsorganisationen mit einer festen Aufteilung des Arbeitsspeichers, wie sie oft in Prozessrechnern für spezielle Anwendungen anzutreffen sind, können dadurch bedeutend ökonomischer aufgebaut werden. Im Extremfall bestehen sie - wie dieses Beispiel beweist - ausschließlich aus dem Nukleus.
  Eine Nebenbedingung des skizzierten Systemaufbaus ist jedoch, daß der Nukleus und die an der Arbeitsspeicherverwaltung beteiligten Prozesse des Supervisors selbst resident sind, um eine rekursive Abhängigkeit voneinander zu vermeiden.

Aus demselben Grunde - der Vermeidung rekursiver Abhängigkeiten - können Prozesse, die im Nukleus ablaufen, nicht über die gleiche Mächtigkeit wie diejenigen der höheren Schichten der Betriebsorganisation verfügen. Es erscheint jedoch sinnvoll, eine zweite Klasse von Prozessen, die Nukleusprozesse, für den Bereich des Nukleus zu definieren. Ihre Zahl ist durch die hardwaremäßig realisierten Ablagebereiche für den Prozessor-Status bestimmt, die hier die Aufgabe der Pseudoprozessoren übernehmen. Nukleusprozesse führen Funktionen des Systemnukleus aus und stützen sich ausschließlich auf die Hardware der realen Maschine. Die Prozessor-Zuteilung zu den existierenden Pseudoprozessoren des Nukleus geschieht hardwaregesteuert durch das Interruptwerk einer DV-Anlage.

Neben den Eigenschaften

- ablaufbezogen und
- arbeitsspeicherresident,

die zwangsläufig aus der funktionellen Aufgliederung der Betriebssystemfunktionen zwischen Nukleus und Supervisor folgen, werden nachfolgend fünf zusätzliche Anforderungen an den Nukleus gestellt.

Die weitestgehende Forderung ist die nach Betriebsartinvarianz. Sie bedeutet, daß der Nukleus selbst weitgehend strategiefrei sein muß und damit als universelle Basis für den Aufbau von Betriebsorganisationen mit unterschiedlichen Betriebsarten dienen kann.

Weiterhin verlangen wir, daß ausschließlich der Nukleus voll privilegiert ist. Mit dieser, oft hardwaremäßig unterstützten Eigenschaft (z. B. durch die Unterscheidung eines System/Benutzer-Modus) ist gemeint, daß nur dem Nukleus das totale Kontrollrecht über alle Betriebsmittel eines DV-Systems eingeräumt wird. Er stellt deshalb vom Standpunkt der Systemzuverlässigkeit den kritischsten Teil einer Betriebsorganisation dar. Programmier-

fehler im Nukleus sind daher grundsätzlich nicht begrenzbar und führen in der Regel zum Zusammenbruch des Systems.

Eine wichtige Forderung stellt die weitgehende Entkopplung des Nukleus vom restlichen Betriebssystem dar. Insbesondere sollen keine, auf den Supervisor übergreifenden Datenbereiche existieren. Diese Restriktion scheint aus zwei Gründen notwendig:

- zur Vermeidung von Systemverklemmungen (Deadlocks) bei der Verriegelung gemeinsamer Daten. Sie können durch die Unterbrechung der beteiligten Supervisor-Prozesse inmitten eines Verriegelungsintervalls dann ausgelöst werden, wenn in der Interruptbehandlung derselbe Datenbereich nochmals verriegelt wird,
- um unterschiedliche Arbeitsspeichersysteme im Nukleus und den darüberliegenden Schichten der Betriebsorganisation nicht von vornherein auszuschließen. Ein Beispiel, das in diesem Zusammenhang große praktische Bedeutung besitzt, ist die Prozess-Liste des Schedulers, die in einer Reihe existierender Systeme [1] auch vom Dispatcher inspiziert wird. Im Falle eines virtuellen Arbeitsspeichers mit unterlegtem Paging müssen für den Scheduler - die allgemeinen Regeln einschränkende - Sondermaßnahmen getroffen werden, um die Pages der Prozess-Liste im reellen Arbeitsspeicher zu verankern (d. h. sie präventiv gegen Page Faults abzusichern).

Alle Nukleus-Prozesse müssen außerdem - im Gegensatz zu den Prozessen der darüberliegenden Schichten - während des Aufenthaltes in Verriegelungsintervallen unterbrechungsgesperrt ausgeführt werden, da es für sie keinen unterstützten Wartezustand gibt.

Schließlich verlangen wir, daß der Nukleus kompakt sei. Für diese Forderung sollen zwei Gründe genannt werden:

- die permanente Residenz im Arbeitsspeicher, die eine Einschränkung des Platzbedarfs, insbesondere bei der Anwendung der vorgeschlagenen Organisation auf Kleinrechner nahelegt,

- das Kurzhalten von unterbrechungsgesperrten Perioden, das für die schnelle Reaktion des Systems auf Realzeitbedingungen entscheidend ist.

2.2 Verfeinerung des Schichtenmodells durch Einführung von Elementarfunktionen

Im vorigen Kapitel wurden in groben Zügen die wesentlichen Merkmale schichtenweise gegliederter Betriebsorganisationen als eine komplexe Anwendung des allgemeinen, von Dijkstra angegebenen Prinzips der hierarchischen Funktionengliederung diskutiert. Ihre grundlegende Struktur wurde maßgeblich durch die Erfordernisse in konventionellen DV-Systemen beeinflußt, in denen die Tendenz besteht, scharf zwischen Prozessen der Benutzer und denen des Betriebssystems zu unterscheiden. Dieser Philosophie liegen hauptsächlich Sicherheitsüberlegungen zugrunde.
Benutzerprozesse, die in der äußersten Schicht der Betriebsorganisation ablaufen, verfügen nur über sehr indirekte Kontrollmittel zur Einflußnahme auf den Betriebsablauf. Von ihnen, durch versuchtes Verändern oder Unterlaufen der Betriebsorganisation ausgelöste Störungen können daher mit entsprechender Hardwareunterstützung meist abgefangen werden.
Anwenderprogrammpakete für Realzeitsysteme können im allgemeinen nicht in dieses Organisationsschema eingebettet werden. Benutzerspezifische, auf den speziellen Anwendungsfall zugeschnittene, Scheduling-Maßnahmen erfordern oft Eingriffe in die unteren Schichten der Betriebsorganisation.
Darüberhinaus ist in der Regel die direkte Kontrolle der externen Prozeßperipherie durch benutzereigene Treiber- und Interruptprogramme unerläßlich. Die dazu notwendigen Eingriffe in den Nukleus setzen detaillierte Kenntnisse über dessen interne Programm- und Datenorganisation voraus.

Aus den beschriebenen Gründen können Anwenderprogrammpakete für Realzeitsysteme nicht mehr einer bestimmten Schicht der Betriebsorganisation zugeordnet werden.

Anwender, die ein schichtenweises gegliedertes DV-System in dieser Weise benutzen, üben daher eine den Systemprogrammierern sehr verwandte Tätigkeit aus und müssen auch vergleichbare Kenntnisse über den internen Systemaufbau besitzen.
Diese Arbeitsweise ist den Intentionen, die hinter Betriebssystemen stehen, entgegengerichtet und stellt eine erhebliche Quelle für Betriebsstörungen dar.

In den höheren Ebenen der Betriebsorganisation können benutzereigene Programme unter Verwendung geeigneter Programmiersprachen noch relativ unproblematisch in den Gesamtkomplex integriert werden, da sie sich auf eine definierte Pseudohardware beziehen und daher abprüfbar sind.
Besonders kritisch sind dagegen Eingriffe in den Systemnukleus, in dem aus bereits erklärten Gründen Programmierfehler nicht begrenzbar sind.
Die dieser Arbeit zugrundeliegende Idee, die eine betriebssichere und rationelle Programmierung auch im Bereich des Systemnukleus ermöglichen soll, besteht darin, für Nukleusprozesse eine den übrigen Schichten der Betriebsorganisation vergleichbare Pseudohardware zu definieren. Ihr Instruktionsvorrat bestehe aus dem Befehlssatz der realen Maschine und einem Satz von Elementarfunktionen, durch die alle systemspezifischen Operationen abgedeckt werden, die im Nukleus durchgeführt werden. Vier Klassen von Elementarfunktionen können unterschieden werden:

- die interruptspezifischen Funktionen,
- die Primitiv-E/A-Funktionen,
- die Dispatcher-Funktionen,
- die Synchronisations- und Kommunikationsfunktionen.

Sie sind in Form von Reentrant-Unterprogrammen organisiert und werden bei Bedarf von Interruptroutinen, den Trägern der Nukleusprozesse, aufgerufen.

Die Programmorganisation des Nukleus zerfällt danach in zwei Schichten:

- in die Schicht der Interruptroutinen,
- in die darunterliegende Schicht der Elementarfunktionen.

In Abb. 3 ist das modifizierte Schichtenmodell dargestellt. Es verdeutlicht, daß Benutzer ihre Programme nicht mehr für eine bestimmte Pseudohardware schreiben, sondern entlang einer "Treppe" programmieren, die sich durch alle Schichten der Betriebsorganisation zieht.

Mit einem derartigen Satz von Elementarfunktionen können Anwender (vom Betriebssystemprogrammierer bis zum eigentlichen Benutzer) Interruptroutinen formulieren, ohne besondere Kenntnisse über die interne Ablauf- und Datenorganisation des Nukleus zu besitzen. Integriert man die Elementarfunktionen in eine geeignete Programmiersprache (z. B. einen Makroassembler), dann sind Interruptroutinen wie gewöhnliche Benutzerprogramme - auch hinsichtlich des Gebrauchs der Elementarfunktionen - syntaktisch abprüfbar.

Die zu Beginn dieses Kapitels geschilderten Nachteile des Schichtenmodells sind durch die vorgelegte Organisation damit weitgehend beseitigt.
Der Nachweis, daß ein Basissatz von Elementarfunktionen definierbar ist, der mit einem Minimum an Hardwareanforderungen auf ein breites Spektrum von unterschiedlichen Maschinentypen abgebildet werden kann, wird hier lediglich für die Dispatcher-Funktionen geführt, auf die sich alle nachfolgenden Untersuchungen beschränken. Die vorliegende Arbeit ist damit Teil eines übergreifenden Forschungsvorhabens [38], das die Definition und Entwicklung aller Elementarfunktionen des Nukleus zum Ziele hat.
Die gewonnenen Resultate sind jedoch weitgehend unabhängig von den Zielen des übergeordneten Projekts und können daher getrennt genutzt werden.

2.3 Der Dispatcher in symmetrischen Mehrprozessorsystemen - Definition, Lokalisierung und Problematik

Die nachfolgenden Untersuchungen konzentrieren sich auf die Klasse der symmetrischen Mehrprozessorsysteme. Wir wollen hierunter alle DV-Systeme verstehen, in denen mehrere gleichartige Prozessoren, die über einen gemeinsamen Arbeitsspeicher gekoppelt sind, gleichwertig im Rahmen der Betriebsorganisation behandelt werden (Abb. 4).
Diese, im Englischen oft mit 'distributed organization' [29 +)] bezeichnete Organisationsform von MP-Systemen hat den Vorzug, daß jeder Prozessor alle Funktionen des Betriebssystems ausführen kann und stellt einen erfolgversprechenden Lösungsansatz für DV-Systeme mit erhöhter Verfügbarkeit dar, in denen Totalausfälle von Prozessoren relativ einfach abgefangen werden können.

Die Elementarfunktionen des Dispatchers übernehmen im Rahmen dieser Betriebsform die Aufgabe, die logische Verknüpfung zwischen allen Prozessoren und Prozessen des Systems herzustellen. Da der Dispatcher damit zentrale Schaltstelle für alle Prozessoren des Systems ist, bildet er einen potentiellen Engpaß ('Bottleneck') für sie. Er wird durch den simultanen Zugriff mehrerer Prozessoren zur Datenbasis des Dispatchers ausgelöst und führt zu Prozessor-Konflikten, die in einer erheblichen Leistungsminderung eines Mehrprozessor-DV-Systems resultieren können.

Ein wesentlicher Bestandteil dieser Arbeit ist daher die Entwicklung geeigneter Datenstrukturen für den Dispatcher mit dem Ziele einer Reduzierung der Konfliktwahrscheinlichkeit. Als grundsätzliche Strategie wird dabei verfolgt, die Datenbasis des Dispatchers in mehrere unabhängige, getrennt zugängliche Bereiche zu untergliedern.

+) "verteilt" (distributed) ist hier lediglich das Kontrollrecht, das allen Prozessoren gleichermaßen eingeräumt wird. Dagegen sind die Betriebssystemfunktionen selbst zentral organisiert, d.h. Code und Datenbasis sind im Regelfall nur einmal angelegt.

Die Synchronisation der Datenzugriffe wird auf der maschinennahen Betriebsorganisationsebene, auf der die Dispatcherfunktionen ablaufen, per Hardware durchgeführt.
In Anlehnung an Dennis [6] wollen wir zu diesem Zweck die Instruktionen LOCK (v) und UNLOCK (v) einführen, in denen v eine logische Verriegelungsvariable bezeichnet, die die Werte 'true' (Datenzugriff verriegelt) oder 'false' (Datenzugriff gestattet) annehmen kann.

Die Wirkungsweise der Instruktionen sei durch die folgende PL/1-Programmsequenzen beschrieben:

LOCK (v)

```
lock : if  v  = 'true' then goto lock ;
                       else v = 'true';
```

UNLOCK (v)

```
unlock : v = 'false' ;
```

Beide Instruktionen sind - wie alle Maschinenbefehle - unteilbar und müssen den Speicher für die Ausführungsdauer blockieren.
Ein Verstoß gegen diese Vorschrift kann zu logisch falschen Abläufen führen [3].
Mit der Einführung dieser Synchronisationshilfsmittel sind diejenigen Hardwareanforderungen definiert, auf die bei der Entwicklung geeigneter Programm- und Datenstrukturen für Dispatcher in den nachfolgenden Kapiteln zurückgegriffen wird.

Die beschriebene Organisation der Dispatcherfunktionen in Unterprogramme hat den Vorteil, daß Interruptroutinen bei Unterprogrammaufrufen immer die Kontrolle zurückerhalten. Der Kontrollfluß wird dadurch übersichtlicher.

Die oft zeitaufwendigen Operationen des Ladens und Rettens von Registerinhalten können zudem aus dem Dispatcher in die Interruptroutinen verlegt, d. h. dezentralisiert werden. Die Programmlaufzeiten im Dispatcher werden damit klein gehalten und die Prozessorkonflikte reduziert.

3. Die Elementarfunktionen des Dispatchers

3.1 Definition und Wirkungsweise

Aus der globalen Definition des Dispatchers im vorangegangenen Kapitel werden nachfolgend einzelne Funktionen abgeleitet und durch eine Reihe von Zustandsänderungen an Prozessen und Prozessoren beschrieben. Zur Verdeutlichung der im Rahmen dieser Diskussion relevanten Zustandsänderungen für Prozesse dient dabei das Zustandsdiagramm in Abb. 5. In Anlehnung an [25, 27] werden vier Prozesszustände betrachtet, die im einzelnen folgende Bedeutung haben:

- SUSPENDIERT

ist ein außerhalb des Dispatchers definierter Wartezustand. Er könnte z. B. von allen Prozessen eingenommen werden, denen der überwiegende Teil ihres erforderlichen Arbeitsspeichers fehlt, und die deshalb vorerst nicht am Wettbewerb um die Zuteilung freier Prozessoren teilnehmen.
Die Zuweisung der für eine Reaktivierung erforderlichen Betriebsmittel unterliegt der Kontrolle eines Schedulers (zuständig für die Betriebsmittelplanung und -führung), der auch den Austrittszeitpunkt von Prozessen aus dem SUSPENDIERT-Zustand bestimmt.

- BEREIT

kennzeichnet den ablaufbereiten Zustand, in dem sich die Prozesse um Prozessoren im System bewerben. Sie führen zu diesem Zweck eine Ordnungskennziffer mit sich (Priorität), die ein eindeutiges Entscheidungskriterium für die Auswahl des nächst zu aktivierenden BEREIT-Prozesses durch einen untätigen Prozessor bildet.

- AKTIV

markiert diejenigen Phasen im dynamischen Ablauf der Prozesse, in denen ihnen ein Prozessor fest zugeordnet ist. Die Allokation bleibt solange bestehen, bis die Prozesse durch eine Dispatcher-Funktion in einen anderen erlaubten Zustand überführt werden.

- BLOCKIERT

ist ein Dispatcher-interner Wartezustand von Prozessen. Der wesentliche Unterschied zum Zustand SUSPENDIERT besteht darin, daß die Durchführung dieses Zustandswechsels ohne Kenntnis des Schedulers geschieht und deshalb die Betriebsmittel der Prozesse unangetastet bleiben.
Nach Befriedigung der Wartebedingungen können blockierte Prozesse durch eine entsprechende Dispatcher-Funktion direkt in den BEREIT-Zustand zurückversetzt werden.

Diese internen Prozesszustandswechsel im Dispatcher, die transparent für den Scheduler sind, bilden die Grundlage für die Realisierung variierender Kopplungen zwischen Dispatcher und Scheduler. Die Zweckmäßigkeit dieses Mechanismus wird im weiteren Verlauf dieses Kapitels an einigen Beispielen noch näher erläutert.

In ähnlicher Weise wie für Prozesse werden für die Prozessoren des Systems drei unterschiedliche Zustände definiert:

- UNTÄTIG

In diesem Zustand befinden sich alle Prozessoren, die keine nützliche Arbeit mehr leisten können. Er wird entweder durch eine Warteschleife oder eine hardwaremäßig implementierte IDLE-Instruktion realisiert, durch die der Instruktionszyklus des Prozessors abgeschaltet wird. Das Verlassen dieses Zustandes kann nur durch einen externen Interrupt erzwungen werden.

- ALLOKIERT

sind alle Prozessoren, die einem aktiven Prozeß zugeordnet sind. Die Allokation bleibt solange bestehen, bis die zugeordneten AKTIV-Prozesse durch eine Dispatcher-Funktion in einen anderen erlaubten Zustand übergehen und zwangsläufig eine Umschaltung der ausführenden Prozessoren in den WACH-Zustand veranlassen.

- WACH

In diesem Übergangszustand halten sich alle Prozessoren auf, die ihren Prozeß abgegeben haben und ihren Zielzustand (ALLOKIERT oder UNTÄTIG) noch nicht erreicht haben.

Die Definition eines WACH-Zustandes scheint zwingend, da Prozessoren in dieser Phase nicht disponierbar und deshalb grundsätzlich verschieden von allokierten oder untätigen Prozessoren zu behandeln sind.

Dies soll an einem Beispiel erläutert werden:

Ein Prozessor habe nach Ausführung der wait-Funktion seinen Prozess abgegeben, die assign-Funktion zwecks Zuteilung eines bereiten Prozesses aber noch nicht aufgerufen. Bleibt der Prozessor weiterhin im Zustand ALLOKIERT, dann ist er potentieller Kandidat einer 'Preemption'. Dies bedeutet, daß er auf Grund seiner momentanen Priorität durch einen Interrupt von einem fremden Prozessor zur Aufgabe seines Prozesses gezwungen werden kann. Der Interrupt kann aber erst nach der Aufnahme eines neuen Prozesses wirksam werden und unterbricht damit in der Regel den Prozess höchster Priorität.

Eine ähnliche unerwünschte Situation entstünde, wenn der Prozessor durch die wait-Funktion in den Zustand UNTÄTIG versetzt würde. Er ist dann potentieller Kandidat einer Neuzuteilung, die ebenfalls durch einen Interprozessor-Interrupt eingeleitet wird. Geht der adressierte Prozessor jedoch vorher durch Aufruf der assign-Funktion in den Zustand ALLOKIERT über, dann wird er nach Aufnahme des neuen Prozesses ungewollt durch den anstehenden Interrupt unterbrochen.

Ein weiterer Grund für die Einführung des WACH-Zustandes ist die erhöhte Flexibilität für den Anwender. Der Eintritt in den UNTÄTIG-Zustand erfolgt unter Kontrolle der Interruptroutinen und ist nicht an das Resultat einer anderen Dispatcher-Funktion gekoppelt

(denkbar wäre die Kopplung an die assign-Funktion, die im Falle einer leeren BEREIT-Liste den ausführenden Prozessor in einen STOP führt).

Entsprechend den betrachteten Zuständen von Prozessen und beteiligten Prozessoren wollen wir die Elementarfunktionen des Dispatchers aus den erlaubten Zustandsänderungen ableiten, die in den Abbildungen 5 und 6 durch Pfeile mit entsprechenden Funktionsnahmen gekennzeichnet sind.
Die folgende Tabelle enthält in einer Übersicht alle auf diese Weise ermittelten Dispatcher-Funktionen sowie die Zustandsänderungen, die an den zugeordneten Prozessen bzw. Prozessoren vorgenommen werden.

Elementar-funktionen	Prozeß-Zustands-wechsel	Prozessor-Zustände oder Zustandswechsel des ausführenden Prozessors
add	SUSPENDIERT → BEREIT	ALLOKIERT v WACH
retire	{BEREIT, BLOCKIERT} → SUSPENDIERT	ALLOKIERT v WACH
assign	BEREIT → AKTIV	{UNTÄTIG, WACH} → ALLOKIERT
release	AKTIV → SUSPENDIERT	ALLOKIERT → WACH
wait	AKTIV → BLOCKIERT	ALLOKIERT → WACH
ready	BLOCKIERT → BEREIT	ALLOKIERT v WACH
reready	AKTIV → BEREIT	ALLOKIERT → WACH
stop	—	WACH → UNTÄTIG

Die Aufstellung zeigt, daß mit einer Ausnahme alle eingeführten Dispatcher-Funktionen Zustandswechsel an Prozessen durchführen. Die Ausnahme bildet die stop-Funktion (Abb. 6), die unabhängig von allen Prozessen lediglich eine Umschaltung des verantwortlichen Prozessors von WACH nach UNTÄTIG bewirkt.

Bevor aus den oben definierten Elementarfunktionen des Dispatchers praktische Implementierungen abgeleitet werden können, ist eine Spezifikation der Funktionsaufrufe sowie eine detaillierte Beschreibung der Semantik erforderlich.
Zur Erkennung dynamischer Fehler, die z. B. durch eine logisch unsinnige Folge von Funktionsaufrufen entstehen könnten, erzeugen alle Elementarfunktionen einen Return-Code, der an die aufrufende Prozedur zurückgegeben wird. In den anschließenden Funktionsbeschreibungen werden daher vor Erläuterung der Wirkungsweise jeweils die Parameter der betrachteten Funktion sowie die gültigen Return-Codes erklärt.
Die Nukleus-relevanten Prozessbeschreibungen werden dabei in Prozess-Kontroll-Blöcken vorausgesetzt, auf die sich einige der Funktionen beziehen.

add (process, policy)

- Parameter

process ist eine Adresse, die auf den Prozess-Kontroll-Block (PCB) des Prozesses weist, der dem Dispatcher übergeben werden soll.

policy ist eine Ausführungsvorschrift für die add-Funktion und kann zwei Werte annehmen:

= 'nonpreemptive', bedeutet, daß als Teil der add-Funktion lediglich nach einem untätigen Prozessor gesucht wird, der - falls vorhanden - durch ein entsprechendes Interrupt-Signal aufgeweckt wird.

= 'preemptive', bedeutet, daß als Nebenwirkung der add-Funktion nach einem Prozessor gesucht wird, der auf einem niedrigeren Prioritätsniveau liegt als der neu eingereihte Prozess und - falls vorhanden - durch ein Interrupt-Signal unterbrochen wird. Sofern sich der signalisierte Prozessor nicht im UNTÄTIG-Zustand befindet, wird durch dieses Vorgehen eine Umschaltung des unterbrochenen Prozessors auf den Prozess höherer Priorität eingeleitet.

- Return Codes

  = 0, ohne Beanstandungen ausgeführt.
  = 1, Funktion aus Speicherplatzmangel für den Aufbau eines neuen Listenelements nicht ausführbar.

- Wirkungsweise der Funktion

Der unter der Adresse process angegebene PCB wird unter Berücksichtigung der Priorität des Prozesses und der Ausführungsvorschrift policy in die Datenbasis des Dispatchers aufgenommen und dort in den Zustand BEREIT versetzt.
Die Priorität wird dabei als eine im PCB an definierter Position enthaltene Größe vorausgesetzt.

retire (process_id, process, policy)

- Parameter

process_id ist die Prozessidentifikation des Prozesses, der aus der Datenbasis des Dispatchers entfernt werden soll.

process enthält nach erfolgreicher Ausführung der Funktion die Adresse des PCB für den bezeichneten Prozess.

policy ist - äquivalent der add-Funktion - eine Ausführungsvorschrift für die retire-Funktion und kann zwei Werte annehmen:

= 'nonpreemptive' , bedeutet, daß der bezeichnete Prozess im Falle AKTIV unangetastet bleibt und lediglich ein Kennzeichen hinterlassen wird, daß eine retire-Anforderung vorliegt. Sie wird beim nächsten Zustandswechsel des Prozesses berücksichtigt.

= 'preemptive', veranlaßt, daß dem allokierten Prozessor eines aktiven Prozesses ein Signal geschickt wird, das eine Unterbrechung auslöst und damit die zwangsweise Ausgliederung des Prozesses aus der Datenbasis des Dispatchers einleitet.

- Return Codes

  = 0, ohne Beanstandungen ausgeführt.

  = 1, die Ausgliederung des Prozesses konnte nur initialisiert werden, da er sich im Zustand AKTIV befindet.

  = 2, der bezeichnete Prozess ist in der Datenbasis des Dispatchers nicht auffindbar.

- Wirkungsweise der Funktion

  Der durch process_id bezeichnete Prozess wird in der Datenbasis des Dispatchers gesucht und - falls er sich im Zustand BEREIT oder BLOCKIERT befindet - ausgegliedert. Dem Parameter process wird die Adresse des PCB zugewiesen. Befindet sich der Prozess im Zustand AKTIV, dann kann seine Ausgliederung nicht schritthaltend erfolgen. Vielmehr wird in Abhängigkeit von der gewählten policy ein Kennzeichen hinterlassen, das die Ausgliederung des PCB zu einem späteren Zeitpunkt durch die release-Funktion erzwingt. In den nachfolgenden Beispielen wird dabei angenommen, daß die Prozessidentifikation jedes Prozesses im zugeordneten PCB an definierter Position enthalten ist.

assign (reg_state, process_id)

- Parameter

  reg_state ist eine Adresse, die bei erfolgreicher Ausführung der Funktion auf die für eine Wiederaufnahme des Prozesses zu ladenden Register zeigt.

  process_id enthält nach erfolgreicher Ausführung der Funktion die Prozessidentifikation des zugeordneten Prozesses.

- Return Codes

  = 0, ohne Beanstandungen ausgeführt,

  = 1, kein BEREIT-Prozess vorhanden,

  = 2, der verantwortliche Prozessor ist bereits allokiert.

- Wirkungsweise der Funktion

  Der bereite Prozess höchster Priorität wird gesucht und - falls vorhanden - in den AKTIV-Zustand versetzt. Gleichzeitig wird eine logische Verbindung zu dem verantwortlichen Prozessor (der die Funktion ausführt) hergestellt, der dadurch in den ALLOKIERT-Zustand übergeht.

release (condition, process)

- Parameter

  condition enthält in Form einer Bit-Maske die Gründe für eine Aufgabe des Prozesses. Sie wird mit einer allgemeinen Wartemaske verknüpft, die als Bestandteil des PCB vorausgesetzt wird und in der alle Wartebedingungen eines Prozesses gespeichert sind.

  process enthält nach erfolgreicher Ausführung der Funktion die Adresse des aus der Datenbasis des Dispatchers entlassenen PCB.

- Return Codes

  = 0, ohne Beanstandungen ausgeführt,

  = 1, der verantwortliche Prozessor ist nicht allokiert.

- Wirkungsweise der Funktion

  Die bestehende logische Verbindung zwischen dem ausführenden Prozessor und dem aktiven Prozess wird aufgehoben und der PCB aus der Datenbasis des Dispatchers ausgegliedert.
  Der Prozess verläßt damit den Zustand AKTIV, während der Prozessor in den Zustand WACH umgeschaltet wird.

wait (condition)

- Parameter

condition enthält in Form einer Bit-Maske die Gründe für eine Blockade des Prozesses, Sie ersetzt den momentanen Wert der allgemeinen Wartemaske, die - wie bereits unter release erläutert - Bestandteil des PCB ist.

- Return Codes

= 0, ohne Beanstandungen ausgeführt,

= 1, Funktion wird nicht ausgeführt, weil eine retire-Anforderung vorliegt,

= 2, Funktion kann nicht ausgeführt werden, da der ausführende Prozessor nicht allokiert ist.

- Wirkungsweise der Funktion

Die bestehende logische Verbindung zwischen dem ausführenden Prozessor und dem aktiven Prozess wird aufgehoben. Der Prozess wechselt in den Zustand BLOCKIERT , während der Prozessor in den WACH-Zustand übergeht.

ready (process_id, condition)

- Parameter

process_id ist die Prozessidentifikation des Prozesses, der aus einem Wartezustand befreit werden soll.

condition enthält in Form einer Bit-Maske alle Wartebedingungen, die von der allgemeinen Warte-Maske des PCB zu entfernen sind.

- Return Codes

= 0, ohne Beanstandungen ausgeführt,

= 1, der bezeichnete Prozess konnte in der Datenbasis des Dispatchers nicht gefunden werden (in diesem Falle ist dem Scheduler eine entsprechende Mitteilung zu machen).

- Wirkungsweise der Funktion

Der durch process bezeichnete Prozess wird gesucht und die in condition gekennzeichneten Wartebedingungen von der allgemeinen Wartemaske im PCB entfernt. Sind alle Wartebedingungen des Prozesses befriedigt, so wird er in den BEREIT-Zustand versetzt und bewirbt sich damit erneut um einen Prozessor.

reready

- Parameter

keine

- Return Codes

= 0, ohne Beanstandungen ausgeführt,

= 1, Funktion konnte nicht ausgeführt werden, da eine retire-Anforderung vorliegt,

= 2, der Prozessor befindet sich nicht im Zustand ALLOKIERT.

- Wirkungsweise der Funktion

Die bestehende logische Verbindung zwischen dem ausführenden Prozessor und dem aktiven Prozess wird aufgehoben. Der Prozess wechselt in den Zustand BEREIT und der Prozessor wird in den Zustand WACH überführt.

stop

- Parameter

keine

- Return Codes

= 0, ohne Beanstandungen ausgeführt,

= 1, Funktion kann nicht ausgeführt werden, da der Prozessor noch allokiert ist.

- Wirkungsweise der Funktion

  Der ausführende Prozessor wird in den Zustand UNTÄTIG versetzt.

Der damit im Detail festgelegte Satz von Elementarfunktionen des Dispatchers kann aus drei Gründen als weitgehend universell bezeichnet werden:

- er ist praktikabel, da er mit einem Minimum an Hardwareeinschränkungen auf ein breites Spektrum von unterschiedlichen Maschinentypen anwendbar ist.
Als einzige Anforderungen an die Hardware wurden bisher die Instruktionen LOCK und UNLOCK sowie eine Möglichkeit der Interprozessor-Kommunikation gefordert, die in existierenden Mehrprozessor-Systemen zur Standardausrüstung gehören.

- er ist weitgehend strategiefrei, d. h. die einzelnen Funktionen treffen keine mehrdeutig interpretierbaren Entscheidungen, sondern folgen einem Vorgehenszwang, der durch elementare Gesetze der Logik vorgegeben ist (z. B. ist die Auswahl eines BEREIT-Prozesses durch die assign-Funktion von dessen Priorität abhängig:
der bereite Prozess höchster Priorität hat den Vorrang, bei mehreren Prozessen mit gleicher Priorität entscheidet der Eintrittszeitpunkt in den BEREIT-Zustand.

Grundlage ähnlicher primitiver Entscheidungen ist oft - wie dieses Beispiel zeigt - die Priorität der Prozesse, die im Dispatcher als Konstante aufgefaßt wird,

- er ist flexibel, d. h. auf Ereignisse im Gesamtsystem, die Eingriffe des Dispatchers erforderlich machen, kann auf unterschiedliche, eine bestimmte Betriebsart sowie das spezifische Anforderungsprofil berücksichtigende Weise reagiert werden. Diese

Flexibilität wird in erster Linie durch die Wartezustände BLOCKIERT und SUSPENDIERT erreicht, die qualitativ gleichwertig sind und damit ein gewisses Überangebot an Dispatcher-Funktionen erzeugen. Zwei Beispiele sollen dies verdeutlichen:

Als erstes sei eine Betriebsorganisation mit zentraler Betriebsmittelverwaltung und fester Kopplung Scheduler-Dispatcher betrachtet. Das entsprechend reduzierte Prozess-Zustandsdiagramm (Bild 7) zeigt, daß der BLOCKIERT-Zustand von keinem der Prozesse eingenommen wird, da jedes Ereignis im System, das den Verlust oder Bedarf eines zusätzlichen Betriebsmittels für Prozesse bedeutet, ihren Übergang vom AKTIV in den SUSPENDIERT-Zustand auslöst. Der Scheduler eines derart operierenden Systems übernimmt im wesentlichen die Funktion eines zentralen Interrupthandlers.
Die Dispatcher-Funktionen wait und ready , die an den BLOCKIERT-Zustand gekoppelt sind, können hier entfallen.

Als entgegengesetztes Beispiel sei eine Betriebsorganisation mit dezentraler Betriebsmittelverwaltung und loser Kopplung Scheduler-Dispatcher betrachtet. Hier wird angenommen, daß die Prozesse nur am Anfang und Ende ihrer Existenz die Funktionen add und release benötigen, für die Dauer ihres Bestehens aber ausschließlich zwischen den Dispatcher-internen Zuständen wechseln.
Der Verlust oder Bedarf eines zusätzlichen Betriebsmittels wird immer durch die Wartestellung der betroffenen Prozesse im Zustand BLOCKIERT beantwortet.
Nach Bereitstellung des gewünschten Betriebsmittels durch einen verantwortlichen Supervisor-Prozess können die blockierten Prozesse durch Aufruf der ready-Funktion erneut in den BEREIT-Zustand versetzt werden.

Gewöhnlich werden Mischformen dieser extremen Organisationen auftreten, und die Wahl der einen oder anderen Strategie ist oftmals eine Praktikabilitätsfrage.

Der Betriebsaufwand (Overhead) wird in der Regel mit steigendem Kopplungsgrad Dispatcher-Scheduler anwachsen, wobei unter Kopplungsgrad das Verhältnis

$$k = \frac{as}{as + ab} \qquad (3.1 - 1)$$

verstanden wird, worin

as die Zahl der AKTIV ⟶ SUSPENDIERT-Übergänge und
ab die Zahl der AKTIV ⟶ BLOCKIERT-Übergänge

von Prozessen bedeuten, die über eine bestimmte Zeit gemessen wurden.

Der gegenüber Dispatcher-internen Zustandswechseln erhöhte Overhead bei steigendem Kopplungsgrad wird hauptsächlich durch den Informationsaustausch zwischen Dispatcher und Scheduler sowie den zusätzlichen Zeitbedarf des Schedulers verursacht.
Feste Kopplungen (k = 1) sind daher in allen Fällen impraktikabel, in denen die mittlere Folge von Zustandsänderungen der Prozesse in die Größenordnung eines add-release-Zyklus vorstößt. Die Ausbildung eines Systemengpasses wäre die wahrscheinliche Folge.

Neben den oben definierten Dispatcher-Funktionen ist eine Reihe weiterer vorstellbar, die entweder optimierenden Charakter haben (d.h. mit Hilfe bereits bestehender Funktionen realisierbar sind wie z.B. die Änderung der Prozess-Priorität) oder Zustandsbeschreibungen über den Dispatcher liefern, wie z. B. Belegungszustand der Listen, mittlere Aufenthaltsdauer der Prozesse in den Dispatcherlisten, Prozessorauslastung udgl., die als Entscheidungshilfen für Scheduling-Verfahren herangezogen werden könnten.

Abschließend sei darauf hingewiesen, daß das vorgestellte Dispatchermodell die Herstellung bedingter Wartezustände nicht erlaubt. Dieser bewußten Einschränkung liegt die Erkenntnis zugrunde, daß die Realisierung bedingter Wartezustände Gegenstand der Synchronisations- und Kommunikationsmechanismen ist, die die Dispatcherfunktionen als funktionellen Unterbau benutzen.

3.2 Auslegungsziele und Darstellungsart

Auf der Basis der im vorigen Kapitel eingeführten Elementarfunktionen werden nachfolgend zwei alternative Datenrepräsentationen für die Datenbasis des Dispatchers angegeben und die Elementarfunktionen algorithmisch beschrieben.

Dominierendes Kriterium bei der Auslegung beider Entwürfe, das einen wesentlichen Einfluß auf die Struktur der Datenbasis sowie die entwickelten Algorithmen hat, ist die Optimierung des Durchsatzes durch die Minimierung der Konflikte zwischen parallel im Dispatcher arbeitenden Prozessoren.
Die Untersuchungen wurden dabei grundsätzlich auf Strukturen beschränkt, die die Einrichtung ablaufkonsistenter Systemzustände zulassen. Ablaufkonsistenz ist dann hergestellt, wenn die n Prozessoren zu den n Prozessen höchster Priorität allokiert sind [43] (bei gleichen Prioritäten gelte FIFO als Ordnungskriterium). Mit dieser Randbedingung sind der angestrebten Partitionierung der Dispatcher-Datenbasis Grenzen gesetzt. Insbesondere verbieten sich alle Organisationen, die auf heuristischen Auswahlverfahren für die Bestimmung von Prozessen oder Prozessoren beruhen und damit die als Ordnungskriterium eingeführte Priorität praktisch ignorieren. Derartige Methoden können in futuristischen Mehrprozessorsystemen dagegen durchaus große Bedeutung gewinnen, in denen die Zahl der Prozessoren bereits eine statistischen Gesetzen zugängliche Größenordnung erreicht. Die Untersuchungen im Rahmen dieser Arbeit beschränken sich auf Mehrprozessorkonfigurationen mit maximal 16 Prozessoren, für die bereits in der nahen Zukunft ökonomisch wettbewerbsfähige DV-Systeme zu erwarten sind [26].

Darstellung der Algorithmen

Zur algorithmischen Beschreibung der Entwürfe sowie einer detaillierten Repräsentation der Datenstrukturen wurde eine kompakte PL/1-Darstellung gewählt, die als Vorlage für den Aufbau der Simulationsmodelle diente. Beide Dispatcher-Entwürfe sind durch abgeschlossene PL/1-Prozeduren realisiert, die dieser Arbeit als Anhang beiliegen und folgenden formalen Aufbau haben.

Im Anschluß an den Prozedurnamen wurden alle globalen Datendeklarationen sowie das Programm für die Initialisierung der Datenbasis zusammengefaßt. Im Deklarationsteil sind außerdem die Hardwareinstruktionen und Hardwareregister der Prozessoren, die von einigen Dispatcher-Funktionen benutzt werden, als externe Prozeduren oder Variable mit führendem $-Zeichen aufgeführt.

Die Instruktionen $LOCK und $UNLOCK sind die bereits beschriebenen Synchronisationshilfsmittel.
Die Instruktion $IDLE versetzt den Prozessor in einen hardwaremäßig realisierten Wartezustand, aus dem er nur durch einen Interrupt befreit werden kann. Er wird durch die $WAKEUP-Instruktion ausgelöst, die als spezielle Form der $SIGNAL-Instruktion für die Interprozessor-Kommunikation aufgefaßt wird. $IDLE und $WAKEUP können entfallen, wenn der Wartezustand durch einen dynamischen STOP realisiert wird.
Die Prozessoridentifikation wird in einem speziellen Register ($CPU-ID) angenommen, das im einfachsten Fall ein für diesen Zweck reserviertes Arbeitsregister des Prozessors ist.

Die unter den nachfolgenden acht ENTRIES aufgeführten Programme stellen die algorithmische Beschreibung der Elementarfunktionen dar. Unterprogramme, die von mehreren Elementarfunktionen gemeinsam benutzt werden, sind unter den nachfolgenden ENTRIES beschrieben.
Durch das Attribut RECURSIVE wird mit den sprachlichen Mitteln von PL/1 dokumentiert, daß das gesamte Programmsystem reentrant ist, ohne auf Einzelheiten der dazu notwendigen Speicherorganisation einzugehen. Dies trifft gleichermaßen für die Parameterübertragung zu, die mit den üblichen PL/1-Techniken behandelt wird.

Bemerkungen zu den Datenstrukturen

Die Verkettung der PCB's sowie die Ablage Dispatcher-interner Organisationsdaten erfolgt in einer unabhängigen Struktur, dem Koppelglied (in den Programmen mit CONNECTOR bezeichnet).

Man erreicht damit, daß Änderungen in der Organisation des Dispatchers ohne Einfluß auf die Struktur des PCB bleiben. Insbesondere kann die Art der Verkettung zwischen verschiedenen Entwürfen variiert werden.

Die Dimensionen der Elemente von Datenstrukturen (wie z. B. die Länge der Wartemaske des PCB) sind in den PL/1-Programmen willkürlich gewählt und müssen im Bedarfsfall an die Erfordernisse der Hardware und der Betriebsorganisation angepaßt werden.

3.3 Der Entwurf A: geschlossene Prozessliste

In Abb. 9 ist die Datenstruktur des Dispatcher-Entwurfs A dargestellt. Man erkennt im wesentlichen zwei verschiedene Datenformationen:

- die Prozessortabelle, die den Prozessorstatus für alle Prozessoren des Systems enthält und
- die Prozessliste, die alle PCB's in abnehmender Priorität enthält, die gegenwärtig dem Dispatcher bekannt sind.

Die PCB's sind - wie bereits erläutert - nicht direkt untereinander verzeigert, sondern indirekt über entsprechende Koppelglieder, die auch den Dispatcher-internen Prozesszustand aufnehmen. Aktive Prozesse sind über ihre Koppelglieder außerdem mit den allokierten Prozessoren verzeigert. Diese Organisation stellt sicher, daß ein allokierter Prozessor direkten Zugriff zum zugeordneten aktiven Prozess hat und umgekehrt zu jedem aktiven Prozess unmittelbar der allokierte Prozessor bestimmt werden kann.

Der Partitionierungsgrad wurde zugunsten einer einfachen, speicherarmen Organisation, die zu einfachen, kurzen Algorithmen führt, bewußt begrenzt.

Die Prozessliste ist nicht weiter untergliedert und muß daher beim Zugriff einer Dispatcher-Funktion immer als Ganzes verriegelt werden.
In der Prozessortabelle kann dagegen jeder Eingang separat verriegelt werden, so daß Stauungen von Prozessoren beim Durchsuchen der Prozessor-Tabelle erheblich reduziert werden. Der Suchvorgang ist immer dann erforderlich, wenn als Folge der Dispatcher-Funktionen add und ready entweder ein untätiger Prozessor oder ein allokierter Prozessor unterhalb eines vorgegebenen Prioritätsniveaus bestimmt werden soll.
Die Unterprogramme WAKE_PR und SIGNAL_PR führen diese Aufgabe durch und zeichnen sich insbesondere dadurch aus, daß die Verriegelungen der Eingänge der Prozessortabelle nach Möglichkeit sequentialisiert werden (d. h. der Eingang i + 1 wird erst dann verriegelt, wenn der i-te Eingang entriegelt ist). Der organisatorische Mehraufwand für eine partitionierte Prozessortabelle gegenüber einer geschlossenen ist nicht essentiell, so daß auf die Untersuchung noch primitiverer Organisationen verzichtet wurde.

3.4. Der Entwurf B: partitionierte Prozess - Liste mit separater BEREIT-Liste

Schon eine qualitative Analyse zeigt, daß der Entwurf A für eine größere Zahl von Prozessoren wegen der hohen Wahrscheinlichkeit für das Auftreten von Prozessorkonflikten ungeeignet ist. Die Prozessorkonflikte werden vor allem durch

- lange Suchvorgänge in der Prozessliste und
- den geringen Partitionierungsgrad der Datenbasis begünstigt.

Die beiden Einflußgrößen multiplizieren sich dabei in ihrer Wirkung, da die Verriegelungsdauer der Prozessliste maßgeblich von der Dauer des Suchvorganges nach einem bestimmten Prozess abhängt.

Folgende Suchvorgänge sind bei Anwendung geeigneter Sortierverfahren stark reduzierbar:

- die Suche nach einem Prozess vorgegebener Identifikation,
- die Suche nach dem bereiten Prozess höchster Priorität.

Die Suchzeiten für Prozesse wurden durch die Reorganisation der Prozessliste des Entwurfs A in eine wählbare Zahl von Sub-Prozesslisten vermindert, die jeweils eine stark reduzierte Teilmenge aller Prozesse aufnehmen und getrennt verriegelbar sind. Der komplette Suchvorgang nach einem vorgegebenen Prozess wird damit zweistufig: im ersten Schritt muß die Sub-Prozessliste ermittelt werden, der ein Prozess angehört, während im zweiten Schritt der Prozess innerhalb der Sub-Prozessliste gesucht wird.

Im vorliegenden Fall wurde angenommen, daß die Sub-Prozesslisten aus der Prozess-Identifikation durch ein Hashing-Verfahren bestimmt werden. Die Köpfe (Header) der Sub-Prozesslisten wurden aus diesem Grund in einer Prozess-Hash-Tabelle zusammengefaßt, die durch den aus der Prozess-Identifikation abgeleiteten Hash-Index adressiert wird.
Die Eliminierung der Suchzeiten für bereite Prozesse wurde durch die zusätzliche Einführung einer BEREIT-Liste erreicht, die alle BEREIT-Prozesse prioritätsgeordnet aufnimmt und als Ganzes verriegelt wird.

Die modifizierte Datenstruktur für den Entwurf B in Abb. 10 zeigt, daß Prozesse gleichzeitig Mitglieder zweier Datenformationen sein können: der Prozess-Hash-Tabelle und der BEREIT-Liste. Der dadurch mögliche gleichzeitige Zugriff von zwei Prozessoren auf einen Prozess über zwei verschiedene Listen erfordert zusätzlich die Verriegelung aller Prozesse und wurde durch Einführung einer LOCK-Variablen in den Koppelgliedern realisiert.
Zur Vermeidung von Deadlocks beim Zugriff zu bereiten Prozessen wurde das Prinzip angewendet, die Verriegelungsobjekte zu ordnen und Verriegelungen mehrerer Objekte nur in der festgelegten Reihenfolge zu gestatten [21].

Im vorliegenden Fall wurden die Objekte in der Reihenfolge Prozess - Hash - Tabelle→BEREIT - Liste→Prozess geordnet. Die vollständige Kontrolle über einen bereiten Prozess (und damit das Privileg, ihn aus der BEREIT-Liste auszugliedern) ist damit nur über die Verriegelung der BEREIT-Liste möglich. Ein Prozess, der durch die retire-Funktion aus der BEREIT-Liste entfernt werden soll und über die Hash-Tabelle verriegelt wurde, wird nach einem entsprechenden Vermerk noch einmal entriegelt. Über die Verriegelung der BEREIT-Liste wird der Prozess dann erneut verriegelt und kann erst jetzt aus der BEREIT-Liste und damit stufenweise aus dem Dispatcher ausgegliedert werden.

Die Organisation der Prozessor-Tabelle wurde ebenso wie die Unterprogramme WAKE_PR und SIGNAL_PR unverändert vom Entwurf A übernommen.

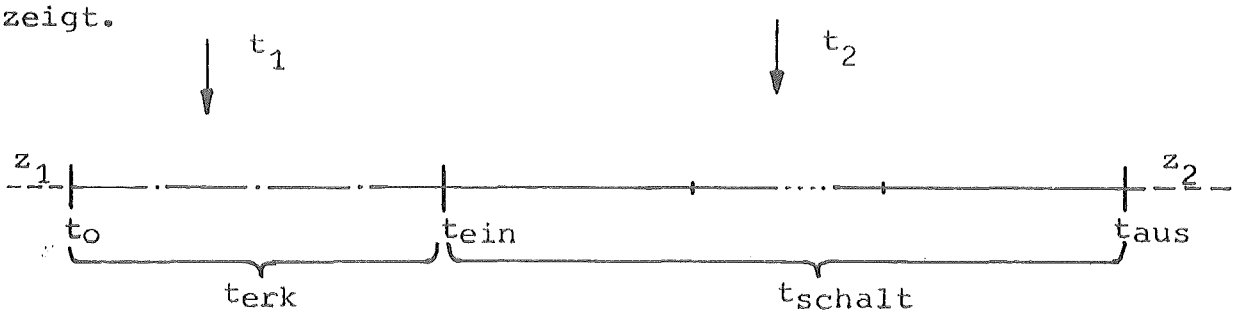
Im Idealfall, in dem die Länge der Hash-Prozess-Tabelle gleich der Zahl der Prozesse ist, werden von den Dispatcher-Funktionen gewöhnlich nur der betroffene Prozess und gegebenenfalls der ausführende Prozessor verriegelt. Ausnahmen bilden die Funktionen add, retire, assign, ready und reready, die auf die BEREIT-Liste zugreifen können und damit alle Prozesse dieser Liste blockieren. Dieser mögliche Engpaß läßt sich unter Beachtung der eingangs formulierten Randbedingung nicht vermeiden, da durch eine weitere Partitionierung der BEREIT-Liste (in z.B. Prozessor-spezifische BEREIT-Listen) keine Ablaufkonsistenz mehr hergestellt werden könnte.

Ein weiterer Engpaß kann durch das Unterprogramm SIGNAL_PR ausgelöst werden, da hier die Eingänge der Prozessortabelle nicht streng sequentiell, sondern teilweise überlappt verriegelt werden. Das Ausmaß der Verriegelung ist dabei stark vom momentanen Zustand aller Prozessoren abhängig und kann im ungünstigsten Fall zur kurzzeitigen Blockade der gesamten Tabelle führen.

Der Einsatz weniger rigoroser Strategien für die SIGNAL_PR-Routine, die letztlich einen Verzicht auf vollständige Ablaufkonsistenz bedeuten, kann hier Abhilfe schaffen. Desweiteren sei angemerkt, daß die SIGNAL_PR-Routine ausschließlich von der add-Funktion mit dem policy-Parameter " 'preemtive' aufgerufen wird. Durch eine Beschränkung preemptiver Scheduling-Disziplinen kann dieser Engpaß daher zusätzlich entschärft werden.

3.5 Ablaufstörungen und ihre Eliminierung

Unter Ablaufstörungen werden hier alle ungewollten, durch unvermeidbare Synchronisationsfehler ausgelösten Unterbrechungen aktiver Prozesse verstanden, die u. U. auch eine temporäre Störung der Ablaufkonsistenz verursachen. Ihre Entstehung sei anhand des nachstehenden Zeitdiagramms näher erläutert, das die verschiedenen Umschaltphasen eines Prozessors aus einem definierten Ausgangszustand z_1 in einen definierten Zielzustand z_2 zeigt.



Der Umschaltvorgang wird in der Regel durch einen Interrupt eingeleitet (Zeitpunkt t_o). Von dort bis zum Eintritt in den Dispatcher, der durch Aufruf einer Dispatcher-Funktion erfolgt, vergeht gewöhnlich eine von Null verschiedene Zeit, in der z. B. der Prozessorstatus gerettet wird oder andere interruptspezifische Aufgaben erledigt werden. Wir wollen diese Zeit als Erkennzeit t_{erk} bezeichnen, da der eingeleitete Zustandswechsel durch Inspektion der Dispatcher-Datenbasis (z. B. von einem fremden Prozessor zum Zeitpunkt t_1) frühestens nach Ablauf dieser Zeit erkennbar ist. Es gibt drei Quellen für Ablaufstörungen, die während des Aufenthalts von Prozessoren in der Erkennphase auftreten können.

1. Ein untätiger Prozessor, der durch einen externen Interrupt (z. B. Kanal-Ende-Interrupt) bereits geweckt wurde, empfängt vor Eintritt in den Dispatcher einen WAKEUP-Interrupt durch einen fremden Prozessor, der in der WAKE_PR- oder SIGNAL_PR-Routine tätig ist.
Die störende Wirkung dieses Interrupts kann auf drei Arten - abhängig von den Fähigkeiten der Hardware - beseitigt werden:

a) WAKEUP-Interrupts werden nur im hardwaremäßig ausgebildeten UNTÄTIG-Zustand wirksam, in den übrigen Prozessor-Zuständen sind sie maskiert und werden daher ignoriert.

b) In Prozessoren mit wenigstens zwei unabhängigen Programmstatusbereichen (ähnlich den PSW's der IBM 360) kann der WAKEUP-Interrupt im Programmstatusbereich der Interruptroutine maskiert werden.

c) Der WAKEUP-Interrupt wird nach Aufruf der assign-Funktion und vor Laden des neuen Prozesstatus gelöscht.
Diese Lösung erfordert es, daß am Prozessor bereits anstehende Interrupts wieder beseitigt werden.

2. Ein allokierter Prozessor, der durch einen Interrupt unterbrochen wurde, empfängt in der Erkennphase ein SIGNAL-Interrupt durch einen anderen Prozessor.

Ähnlich den Lösungen(b) und (c) des vorher besprochenen Falles kann die störende Wirkung dieses Interrupts auf zwei Arten - abhängig von den Fähigkeiten der Hardware - eliminiert werden:

a) In Prozessoren mit wenigstens zwei unabhängigen Programmstatusbereichen kann der SIGNAL-Interrupt im Programmstatusbereich der Interruptroutine maskiert werden.

b) Der SIGNAL-Interrupt wird vor Laden des neuen Prozesstatus gelöscht.

Beide Maßnahmen müssen jedoch unterdrückt werden, wenn der Prozessor unter Ausschaltung des Dispatchers den unterbrochenen Prozess wieder fortsetzt. Sie dürfen alo nur bei einer Prozessorumschaltung angewendet werden.

3. Ein allokierter Prozessor empfängt in der Interruptbehandlung einen weiteren Interrupt durch einen abgelaufenen TIMER. Wird durch den ersten Interrupt eine Prozessumschaltung ausgelöst, dann unterbricht der TIMER-Interrupt letztlich den neu allokierten (und damit falschen) Prozess.
Auch hier können drei Korrekturverfahren angesetzt werden, die im wesentlichen von der Leistungsfähigkeit der Prozessor-Hardware abhängen:

a) Der TIMER wird - ausgelöst durch eintreffende Interrupts - gestoppt.
   Diese reine Hardwarelösung ist am bequemsten und erfordert keine nennenswerten Aufwendungen. Sie sei daher auch als Empfehlung an Hersteller von DV-Anlagen verstanden.

b) In Prozessoren mit wenigstens zwei unabhängigen Programmstatusbereichen kann der TIMER-Interrupt im Programmstatusbereich der Interruptroutine maskiert werden.

c) Der TIMER-Interrupt wird nach Aufruf der assign-Funktion und vor Laden des neuen Prozesstatus durch die Interruptroutine gelöscht.

Die Verfahren (b) und (c) erfordern in der Regel zusätzliche Maßnahmen der TIMER-Adjustierung, in denen die korrekte Behandlung des Interrupts zu einem späteren Zeitpunkt sichergestellt wird.

Eine wesentliche Feststellung am Schluß dieser Aufstellung ist, daß alle besprochenen Korrekturverfahren außerhalb des Dispatchers anwendbar sind. Sie können als Bestandteil des Prozessor-Statuswechsels angesehen werden, der in die hier nicht behandelte Klasse der interruptspezifischen Elementarfunktionen gehört [38].

Neben diesen Dispatcher-externen Ablaufstörungen, die durch eine von Null verschiedene Erkennzeit verursacht werden, können prinzipiell auch Dispatcher-interne Ablaufstörungen auftreten.

Sie kommen dadurch zustande, daß der Zustandswechsel eines Prozessors und ggf. des mit ihm verbundenen oder zu verbindenden Prozesses nicht in einem Zuge durch globale Verriegelung der gesamten Datenbasis des Dispatchers geschieht. Stattdessen werden Teile der Datenbasis zur Vermeidung von Prozessorkonflikten sequentiell verriegelt, so daß zwischen unabhängig im Dispatcher arbeitenden Prozessoren Überholvorgänge möglich sind. Der komplette Umschaltvorgang, der durch eine Dispatcher-Funktion durchgeführt wird, zerfällt daher in der Regel in einige unabhängige Teilphasen, in denen entweder der Zustand des Prozessors oder des beteiligten Prozesses geändert wird. Nach Abschluß einer derartigen Teilphase könnte z. B. die Zustandsänderung eines Prozesses durchgeführt, der Prozessor aber noch im ursprünglichen Zustand sein (Beispiel: die assign-Funktion hat den Prozess bereits von BEREIT nach AKTIV umgeschaltet, der Prozessor aber befindet sich noch im Zustand WACH). Reaktionen der Dispatcher-Funktionen auf derartige Situationen, die in dem oben dargestellten Zeitdiagramm zum Zeitpunkt t_2 ausgelöst werden, können unlogisch sein und daher Ablaufstörungen verursachen.
Sie wurden im vorliegenden Fall nach einer kombinatorischen Gegenüberstellung der Funktionen wait, release, reready, assign mit den asynchron eingreifenden Funktionen add und retire durch die Festlegung einer Reihenfolge eliminiert, in der Zustandsänderungen an Prozessoren und Prozessen erlaubt sind. In den Funktionen wait, release, reready wird als erstes der Prozessor umgeschaltet, während in der assign-Funktion der Zustand des Prozesses zuerst geändert wird. Überholvorgänge werden dadurch ausgeschaltet und mögliche Ablaufstörungen auf die bereits behandelten Arten zurückgeführt.

4. Simulation der Dispatcher-Entwürfe

4.1 Anforderungen an die Simulationsmethode

Das Studium der verfügbaren Literatur über die Simulation von Betriebssoftware (z. B. [22, 23, 30, 31]) zeigt, daß Erfahrungen über Modellaufbau und Simulationsmethoden vorwiegend in Anwendungen vorliegen, die für die Lösung der hier gestellten Aufgabe in nur geringem Umfang genutzt werden können.

In nahezu allen untersuchten Simulationsprojekten wurde die Modellierung der Software auf einem relativ groben Niveau durchgeführt, auf dem lediglich die funktionellen Eigenschaften von Betriebssystemkomponenten nachgebildet wurden. Dagegen wurde der Eigenbedarf der modellierten Betriebssystemfunktionen und damit ihr indirekter Einfluß auf die operativen Ziele entweder garnicht oder nur stark vereinfacht in Rechnung gestellt.
Über das einzige, dem Verfasser bekannte Projekt, in dem Betriebssoftware auf der Instruktionsebene simuliert wurde, berichten Fox und Kessler [10]. Die Maschineninstruktionen des OS der IBM 360 wurden dort durch FORTRAN-CALL-Statements ersetzt und ihre Wirkung in den entsprechenden Unterprogrammen nachgebildet. Zweck dieses Experiments war es, durch Vergleich des Verhaltens des realen mit dem simulierten Betriebssystem ein besseres Verständnis für den Umgang mit dem Hilfsmittel 'Simulation' zu bekommen.

Ziel der hier vorzustellenden Simulationen ist es, die Rückwirkungen des Eigenzeitbedarfs und Parallelgrades der Dispatcher-Entwürfe auf den Durchsatz des Systems unter dem Einfluß variierender Strukturierungsgesichtspunkte der Dispatcher-Datenbasis zu studieren. Diese Aufgabenstellung verlangt eine hohe Modellierungstiefe der Betriebssystemfunktionen, die in einer mit Maschinensprachen vergleichbaren Auflösung geschehen muß.
Ein weiterer wesentlicher Faktor für die Wahl der Simulationsmethode ist, daß sie bevorzugt der Gewinnung von Vergleichsaussagen zwischen alternativen Betriebssystementwürfen dienen soll.

Sie stellen eine häufig wiederkehrende Aufgabenstellung dar, denen Betriebssystementwickler gegenüberstehen und auf Grund mangelnder softwaretechnologischer Hilfsmittel in der Regel gefühlsmäßig entschieden werden.
Die Beschränkung auf vergleichende Aussagen (z. B. Entwurf B ist um 20 % effektiver als Entwurf A) legt die Benutzung einer höheren Programmiersprache als Implementierungsbasis für Betriebssystemmodelle nahe, die allerdings eine realitätsnahe Darstellung struktureller Zusammenhänge gestatten muß.
Der dominierende Einfluß der Strukturbeziehungen auf das dynamische Verhalten eines Systementwurfs wirkt weit oberhalb des Instruktionsniveaus und bleibt damit auch bei unterschiedlichen maschineninternen Darstellungen bestimmend für die Betriebsweise. Als Beispiel können die Verriegelungsintervalle für die Dispatcher-Entwürfe A und B angeführt werden, deren mittlere zeitliche Dauer sich um bis zu drei Größenordnungen unterscheidet.

Auf die Realisierung spezifischer Hardwareabhängigkeiten zwischen dem Modell und einer Zielhardware kann deshalb verzichtet werden.
An ihre Stelle tritt ein Verfahren, in dem die Statements der ausgewählten Modell-Implementierungssprache unmittelbar mit Zeitwichtungen versehen werden. Sie können z. B. nach folgender Formel näherungsweise ermittelt werden:

$$T = \left(\sum_{i=1}^{n} a_i + \sum_{j=1}^{m} op_j \right) \cdot \tau \qquad (4.1 - 1)$$

τ Grundtaktzeit

$n + 1$ Zahl der Speicherzugriffe zum Abholen der Operanden und Abspeichern des Resultats

m Anzahl der Operationen

a_i Zahl der Grundtakte für die Durchführung des i-ten Speicherzugriffs

op_j Zahl der Grundtakte für die Durchführung der j-ten Operation

Man kommt damit unmittelbar zur Vorstellung einer virtuellen Maschine, deren Maschinencode aus Statements dieser Sprache besteht. Um eine möglichst große Annäherung von Statements der virtuellen Maschine an übliche Maschineninstruktionen zu erreichen, sollten auf der rechten Seite jedes Statements immer nur so viele Operatoren stehen, wie eine durchschnittliche reale Maschine in einer Instruktion verarbeiten kann.

4.2 Das Simulationsinstrument, eine virtuelle PL/1-Maschine

Als Basis sowohl für die Implementierung des Simulationssystems als auch für die zu simulierenden Betriebssystemprogramme wurde PL/1 aus folgenden Gründen gewählt:

- es ist für die Formulierung von Betriebssoftware, sowie die Darstellung von Datenstrukturen gut geeignet,
- PL/1-Programme sind leicht lesbar und haben deshalb einen hohen Dokumentationswert,
- der von IBM gelieferte zeitoptimierende PL/1-Compiler erzeugt Objektprogramme hoher Güte, die eine Grundvoraussetzung für angemessene Simulationslaufzeiten darstellen,
- die zunehmende Benutzung von PL/1 - auch außerhalb des IBM-Kundenkreises - ermöglicht eine wirksame Verbreitung der erzielten Ergebnisse.

Nachfolgend werden die Grundzüge des Simulationssystems sowie dessen wesentliche Merkmale und Fähigkeiten vorgestellt. Einzelheiten sind in [33, 34, 35] zu finden.

Abb. 11 zeigt den Aufbau des Simulationssystems. In der oberen Hälfte ist die Kontrollstruktur dargestellt, die in drei Ebenen untergliedert ist.

Wir unterscheiden

- die Eventsteuerung,
- die Ablaufsteuerung der PL/1-Maschine,
- das Modell.

Event- und Maschinensteuerung zusammen bilden die virtuelle PL/1-Maschine (kurz: PLIM). Das Modell besteht aus dem zu simulierenden Programmsystem und ist aus einer Kollektion von PL/1-Prozeduren (Modellprogramme) zusammengesetzt, an die hinsichtlich des Aufbaus noch näher zu spezifizierende Anforderungen gestellt werden.

Im unteren Teil der Abbildung ist der Erstellungsprozess für Modellprogramme dargestellt:
der eigentliche Quellcode wird zunächst durch einen speziellen Makroprozessor modifiziert und dann durch den standardmäßigen PL/1-Optimizing Compiler übersetzt.

Eine Besonderheit des hier realisierten Verfahrens ist, daß Modellprogramme entgegen der sonst üblichen Technik nicht interpretiert, sondern Anweisung für Anweisung von der PL/1-Maschine durchlaufen werden. Der dadurch notwendige Kontrollflußwechsel zwischen PL/1-Maschine und Modellprogrammen, der nach jedem PL/1-Statement stattfindet, erfordert eine entsprechende Aufbereitung des Quellcodes der Modellprogramme, die ebenfalls durch den bereits erwähnten Makroprozessor vorgenommen wird.

Das damit in groben Zügen skizzierte Verfahren hat folgende Vorteile:

- Anstelle eines speziellen Übersetzers, der Objektcode für die virtuelle Maschine erzeugt, wird lediglich ein vergleichsweise primitiver Makroprozessor benötigt.
- Der Aufwand für einen Codeinterpretierer entfällt.
- die Laufzeiteffizienz wird durch den Fortfall der Codeinterpretation wesentlich gesteigert.

Quellcode - Modifikationen

Abb. 12 zeigt an einem einfachen Beispiel die wesentlichen Modifikationen, die am Quellcode vorgenommen werden. Die mit einem führenden Dollarzeichen am Anfang einer Zeile angegebenen Konstanten sind die bereits erwähnten Bewertungsfaktoren, die vom Makroprozessor in PL/1-Statements umgesetzt werden müssen. Alle durch Zeitbewertungen gekennzeichneten Statements werden für eine spätere Reaktivierung mit Marken versehen, die mit L000, L001, L002 ..... durchnumeriert sind. Die Marke des nächsten auszuführenden Statements wird zusammen mit der aufgelaufenen Ausführungszeit in eine dynamische Savearea (DSA) gerettet, die in Form einer BASED-Struktur angelegt ist.

Man erkennt, daß die Ausführung eines Statements in zwei Phasen zerfällt. In der ersten Phase wird eine Zeitkorrektur durch Auswertung der Zeitbewertung durchgeführt, während in der zweiten Phase das Statement tatsächlich abgehandelt wird. Nach Ausführung jedes RETURN-Statements geht die Kontrolle an die aufrufende PL/1-Maschine zurück, die dann den nächsten Zeitpunkt für die Reaktivierung des Modellprogramms bestimmt. Voraussetzung ist jedoch, daß alle temporären Variablen entweder in einem stationären oder dynamisch kontrollierten Speicherbereich angelegt sind. Die Benutzung automatischer Variablen in PL/1 (Attribut AUTOMATIC), die eine dynamische Einrichtung und Freigabe des zugehörigen Speicherplatzes beim Aufruf oder Verlassen einer Prozedur bewirkt, ist deshalb unzulässig, Erlaubt dagegen ist die Ablage der Variablen in Speicherbereichen des Typs CONTROLLED, STATIC oder BASED. Im vorliegenden Modell wurden alle temporären Variablen der Modellprogramme in BASED-Speichern abgelegt. Sie bieten ein einfaches Mittel, um Modellprogramme auf PL/1-Ebene reentrant zu organisieren.

Anweisungen des Quellcodes, die keine Zeitbewertung mitführen, werden vom Makroprozessor unverändert übernommen. Sie tragen des-

halb weder zu Laufzeiten der Programme bei, noch beeinflussen sie den Kontrollfluß zwischen PL/1-Maschine und Modellprogrammen. Diese Transparenz ist von Vorteil, weil sie auf bequeme Weise gestattet, Kontrollausdrucke oder Zwischenauswertungen in den Quellcode einzuschieben, ohne die Eigenschaften des Meßobjektes (= Modellprogramme) zu verändern.

Die Bewertung von Statements innerhalb von Blöcken (Unterprogramme, BEGIN ... END, DO-Gruppen) ist auf Grund der beschriebenen Kontrollstruktur unzulässig, da der Wiedereintritt in einen Block nur über die zulässige Aktivierungskette aller Blöcke geringerer Schachtelungstiefe erfolgen darf. Dagegen ist es möglich, Blöcken eine Zeitbewertung zuzuordnen und sie im Sinne der PL/1-Maschine wie eine Maschineninstruktion zu behandeln.

Die zeitgerechte Aktivierung aller Funktionen der Maschinensteuerung wird durch die Eventsteuerung kontrolliert, die alle Aktionen des Systems in einer prioritäts- und zeitgeordneten Eventkette verwaltet. Da diese Form der Organisation Standard-Simulationstechnik darstellt, wird hier auf eine weitere Vertiefung verzichtet (z. B. [31, 35]).

Prozessor-Struktur

Die Struktur jedes Prozessors der PL1M wird durch eine entsprechende PL/1-Datenstruktur wiedergegeben. Sie zerfällt in 4 voneinander trennbare Unterstrukturen, die den Rechnerkernstatus des aktuellen, d. h. in der Ausführung begriffenen Programmes, des Interruptprogrammes, eines unterbrochenen Programmes und einen Bereich zur Aufnahme von Interrupt-Statusworten enthalten. Anstehende Interrupts werden - vertreten durch Interrupt-Statusworte - in einer prioritätsgeordneten Warteschlange bis zu ihrer Aktivierung gepuffert.
Außerdem tragen alle Prozessoren eine eindeutige Identifikation in einem speziellen Register, die in der Initialisierungsphase zugeteilt wird.

Abb. 13 verdeutlicht die Funktion der drei Registersätze, die jeweils einen kompletten Rechnerkernstatus enthalten.

Nach Annahme eines Interrupts wird der aktuelle Rechnerkernstatus in den Save-Bereich umgespeichert und anschließend der neue Status des Interruptsprogrammes in den aktuellen Bereich geladen. Damit wird das Interruptprogramm zum aktuellen Programm, die Interruptbearbeitung ist initialisiert. Da nur ein Interruptprogramm für die Gesamtheit aller Interrupts existiert, muß die spezifische Reaktion durch programminterne Verzweigungen organisiert werden.

Der Status des unterbrochenen Programms, der im SAVE-Bereich zur Verfügung steht, kann entweder abgespeichert oder durch eine Systeminstruktion ($Return) wieder zum aktuellen Status erklärt werden. Das Programm wird daraufhin fortgesetzt. Bekanntlich bieten doppelte Registersätze dieser Konstruktion die Möglichkeit, kurze Interruptprogramme ohne zeitraubendes Registerumladen über den Arbeitsspeicher in ein laufendes Programm einzublenden.

Der eigentliche Rechnerkernstatus enthält neben dem Programmnamen einen Hinweis auf den zugeordneten SAVE-Bereich sowie die momentane Priorität und den Modus des Rechnerkerns.

Durch die Angabe eines SAVE-Bereiches können alle Programme reentrant organisiert werden. Die Zuweisung und Verwaltung dieses Speichers ist Sache der PLIM-Hardware.

Laufmodi der Prozessoren

Vier simulationsbezogene Laufmodi der Prozessoren werden unterschieden:

a) RUNNING
   bedeutet, daß ein Prozessor Instruktionszyklen ausführt,

b) LOOPING
   bedeutet, daß sich ein Prozessor inmitten der Ausführung einer Instruktion befindet.

c) IDLE
bedeutet, daß ein Prozessor keine Instruktionszyklen ausführt. Jeder Interrupt schaltet ihn automatisch in den RUNNING-Modus zurück.

d) PSEUDORUNNING
bedeutet, daß sich ein Prozessor in einem simulierten RUNNING-Modus befindet.

Dieser für die effiziente Nutzung der PL/1-Maschine als Simulationsinstrument wichtige Modus erlaubt es, in ihren Aktionen unwesentliche Programmstücke eines Modells durch einfache Zeitstrecken nachzubilden, die durch ein einziges Event in der Eventkette vermerkt werden.
Der PSEUDORUNNING-Modus bietet damit die Möglichkeit einer Zwei-Stadien-Simulation:

- Der Modellkern wird in der Regel mikroskopisch dargestellt, alle Operationen sind algorithmisch aufgelöst.
- Die Umgebung des eigentlichen Meßobjekts wird überwiegend makroskopisch nachgebildet, in dem die typischen Ausführungszeiten von Programmen durch - z. B. statistisch verteilte - Zeitstrecken mit Hilfe von PSEUDORUNNING-Phasen angenähert werden.

Eine Aufstellung über den Maschineninstruktionsvorrat der PLIM, der sich aus Standardinstruktionen sowie System- und Pseudoinstruktionen zusammensetzt, liegt der Arbeit als Anhang bei.

Neben Maschineninstruktionen stehen im Rahmen der PL/1-Maschine Unterprogramme für ein breites Spektrum von Zufallszahlengeneratoren unterschiedlicher Verteilung zur Verfügung, deren Gebrauch ähnlich den Systeminstruktionen erfolgt.

Die durchgeführten Messungen ergaben, daß der Faktor Simulationslaufzeit/simulierter Laufzeit, der ein Maß für die Laufzeiteffizienz des Simulationsverfahrens darstellt, für mikroskopisch aufgelöste Modelle etwa 300 betrug. Durch den Einschluß von Makrophasen wird dieser Wert erfahrungsgemäß um ca. eine 10er Potenz reduziert, so daß sich für durchschnittliche Modelle ein Effi-

zienzfaktor von etwa 30 einstellt.

Das gesamte Simulationspaket ist auf der Großrechenanlage IBM 360/370 des Kernforschungszentrums Karlsruhe implementiert und im Time-Sharing Betrieb zugänglich. Es umfaßt etwa 3500 Zeilen PL/1-Code.

4.3 Das Modell

Die im vorhergehenden Abschnitt behandelte virtuelle PL/1-Maschine diente als Simulationsbasis für den Aufbau und Test eines Betriebssystemmodells, das den Dispatcher sowie alle mit ihm kooperierenden Komponenten der Betriebsorganisation enthält. Die Gesamtanordnung, die in Abb. 14 dargestellt ist, zerfällt in drei Teile:

- den Dispatcher, der in mikroskopischer Auflösung dargestellt und im wesentlichen mit den im Anhang aufgeführten PL/1-Programmpaketen identisch ist,

- der Dispatcherumgebung, bestehend aus dem Interrupthandler, den Kommunikationsfunktionen, dem Scheduler, dem Short-Wait-Handler, dem Long-Wait-Handler und den Benutzerprozessen, deren Funktionsweise nur insoweit berücksichtigt wurde, wie sie zum dynamischen Verhalten des Dispatchers beiträgt,

- den außerhalb des Betriebssystem-Modells liegenden Monitor, der ausschließlich Überwachungsaufgaben wie - Steuerung der Ablaufmodi (Test- oder Produktionsmodus), Anfertigung von Schnappschüssen und die kontrollierte Beendigung des Simulationslaufs mit Rettung des Endzustandes durchführt.

Eine ausführliche Beschreibung aller Modellkomponenten befindet sich im Anhang.

Dem simulierten Betriebsverhalten liegt ein Ablaufmodell für Benutzerprozesse zugrunde, in denen zwischen Arbeitsphasen und Wartephasen unterschieden wird:

Während der Arbeitsphasen halten sich Prozesse im Zustand AKTIV auf.
Die Wartephasen werden - unabhängig von ihrer Ursache - in Long-Wait-Phasen (LW-Phasen) und Short-Wait-Phasen (SW-Phasen) unterteilt, die zu den Prozess-Zuständen SUSPENDIERT und BLOCKIERT korrespondieren und damit unterschiedliche Reaktionen des Dispatchers hervorrufen.
Die Zeitabschnitte a, lw, sw stellen dabei Zwischenankunftszeiten dar, die als Realisierung von exponentiell verteilten Zufallsvariablen mit wählbaren Mittelwerten aus drei voneinander unabhängigen Zufallszahlengeneratoren erzeugt werden [18]. Zusätzlich wird durch einen vierten binären Zufallszahlengenerator die Verteilung zwischen Long-Wait- und Short-Wait-Phasen gesteuert, deren wählbare Häufigkeit h angibt, wieviel Short-Wait-Phasen im Mittel auf eine Long-Wait-Phase kommen.
Die Häufigkeit h ist unmittelbar mit dem durch Glg. (3.1-1) eingeführten Kopplungsgrad k durch folgende Beziehung verknüpft:

$$k = \frac{1}{1 + h} \qquad (4.3 - 1)$$

Der Kopplungsgrad k wird danach 1 , wenn h = 0 ist, d.h. keine Short-Wait-Phasen vorkommen, dagegen geht k gegen Null, wenn h gegen ∞ geht, also ausschließlich Short-Wait-Phasen existieren.

Die Bedeutung der Wartephasen ist beliebig interpretierbar und kann z.B. einen Ein/Ausgabe-Wunsch, einen Page-Fault oder Timeslice-Ende darstellen. In jedem Falle wird angenommen, daß die erzeugten Wartebedingungen nur durch eine Serviceleistung des Betriebssystems befriedigt werden können.

Anstelle der großen Zahl möglicher Funktionen des Supervisors wurden der Long-Wait-Handler und Short-Wait-Handler eingeführt. Sie führen die verlangten Serviceleistungen scheinbar aus und initialisieren die Reaktivierung der wartenden Benutzerprozesse. Diese Darstellung der Wechselwirkung zwischen Benutzerprozessen und dem Betriebssystem berücksichtigt auch die Folgereaktionen, die durch Zustandswechsel der Benutzerprozesse im Dispatcher ausgelöst werden und stellt daher eine gute Annäherung an das Verhalten realer Systeme dar.

Der Scheduler dient als Auffangstelle für Prozesse im SUSPENDIERT-Zustand, die durch die release-Funktion aus dem Dispatcher entfernt wurden. Er folgt keiner besonderen Scheduling-Disziplin, sondern entläßt auf Verlangen des Nukleus Prozesse wieder an den Dispatcher.
Diese Vorgehensweise bedeutet, daß neben der Begrenzung der Zahl der verfügbaren Prozessoren keine weitere Beschränkung von Betriebsmitteln in diesem Modell vorgesehen ist.
Insbesondere wird ein infiniter Arbeitsspeicher vorausgesetzt, um die mit der Speicherorganisation zusammenhängenden Phänome nicht mit denen des Dispatching zu vermischen.

Der Verkehr der Supervisorprozesse Scheduler, Long-Wait-Handler und Short-Wait-Handler untereinander erfolgt über Botschaftenkanäle, die über im Nukleus bereitgestellte Kommunikationsfunktionen manipulierbar sind.

Vorliegende Erfahrungen

Die Erstellung des kompletten Modells erforderte ca. 2500 Zeilen PL/1-Code, wobei etwa zwei Drittel des Aufwandes auf die Dispatcher-Umgebung entfielen.
Für das Austesten des Modells mit dem Dispatcher-Entwurf A wurden noch ca. 2 Monate benötigt, da in dieser ersten komplexen Anwendung des Simulationsinstruments logische Fehler in allen Organisationsebenen einschließlich der Modellprogrammerstellung auftraten, die zu starken Wechselwirkungen führen und nur schwer lokalisierbar sind.

Die Umstellung auf den Entwurf B konnte dagegen in wenigen Tagen durchgeführt werden. Die Vorteile eines modularen Programmaufbaus, der durch feste Aufrufkonventionen und die Entkopplung der Dispatcher-Datenbasis vom restlichen Modell gekennzeichnet ist, wurden hier deutlich sichtbar.

4.4 Das Meßverfahren

Ziel der Simulationen ist die Messung des unter vorgegebenen Lastbedingungen erzielten Durchsatzes in beiden Dispatcher-Entwürfen bei einer wechselnden Zahl von Prozessoren.
Als Gütemaß eines Dispatcherentwurfs wird der normierte Problemdurchsatz d definiert, der aus den über die simulierte Laufzeit T_{tot} aufakkumulierten Arbeitsphasen der Benutzerprozesse T_a durch folgende Beziehung verknüpft ist:

$$d = \frac{T_a}{T_{tot}} \qquad (4.4 - 1)$$

Offenbar ist unter gleichen operativen Bedingungen diejenige Dispatcher-Variante den anderen überlegen, die in einem vorgegebenen Meßintervall den größten Problemdurchsatz erzielt und daher den geringsten Engpaß darstellt.

Da im Rahmen dieser Untersuchungen ausschließlich das zeitliche Verhalten des Dispatchers und dessen Rückwirkung auf die operativen Eigenschaften des Gesamtsystems interessieren, wurden die Zeit-Beiträge aller übrigen Komponenten des Betriebssystemmodells zu Null gesetzt. Sie wirken daher nur dann auf den dynamischen Ablauf des Modells ein, wenn sie selbst Dispatcher-Funktionen aufrufen oder zu ihrer eigenen Disposition (z.B. Umschalten eines Supervisorprozesses von BLOCKIERT nach BEREIT) auf die Ausführung entsprechender Dispatcher-Funktionen angewiesen sind.

Die zeitfortschaltenden Beiträge im Modell können daher nur folgenden Quellen entstammen:

- den Arbeitsphasen der Benutzerprozesse,
- den Short-Wait- und Long-Wait-Phasen der Benutzerprozesse,
- dem Eigenzeitbedarf des Dispatchers, der durch statementweise Zeitbewertungen nach Formel (4.2-1) in allen Elementarfunktionen berücksichtigt wurde.

Die direkte Messung der Laufzeiten aller Dispatcherfunktionen wäre messtechnisch relativ umständlich zu bewerkstelligen und wurde durch die Messung der Summe aller UNTÄTIG-Phasen der Prozessoren (T_{idle}) ersetzt. Der Eigenzeitbedarf des Dispatchers T_{disp} läßt sich dann bei bekannter simulierter Laufzeit T_{tot} auf einfache Weise ermitteln:

$$T_{disp} = r \cdot T_{tot} - T_{idle} - T_a \qquad (4.4 - 2)$$

Die simulierte Laufzeit T_{tot} muß mit der Zahl r der Prozessoren multipliziert werden, da T_{idle} und T_a ebenfalls die Summe der Beiträge aller Prozessoren enthalten.

Aus den vorgegebenen bzw. gemessenen Größen T_{tot}, T_{idle}, T_a lassen sich die folgenden Beziehungen für die mittlere Prozessorbelegung b und den Dispatcher-Eigenverbrauch e (Overhead) leicht berechnen:

$$b = \frac{r \cdot T_{tot} - T_{idle}}{r \cdot T_{tot}} \qquad (4.4 - 3)$$

(mittlere Prozessorbelegung)

und

$$e = \frac{T_{disp}}{r \cdot T_{tot} - T_{idle}} = \frac{r \cdot T_{tot} - T_{idle} - T_a}{r \cdot T_{tot} - T_{idle}} \qquad (4.4 - 4)$$

(Dispatcher-Eigenverbrauch)

Es erscheint vernünftig, bei der Definition des Dispatcher-Eigenverbrauchs den Eigenzeitbedarf T_{disp} nicht auf die gesamte simulierte Laufzeit T_{tot}, sondern lediglich auf die Summe derjenigen Zeitintervalle zu beziehen, in denen die Prozessoren nicht UNTÄTIG sind. Die im nachfolgenden Kapitel diskutierten Meßergebnisse zeigen jedoch, daß UNTÄTIG-Zeiten von Prozessoren nicht nur durch verminderte mittlere Belegungen der Prozessoren zustande kommen, sondern auch durch Sättigungserscheinungen des Dispatchers in extremen Belastungs-Situationen ausgelöst werden. Der durch Glg. (4.4 - 4) ermittelte Eigenverbrauch ist daher als ein generelles Gütemaß zur Beurteilung eines Dispatcher-Entwurfs unbrauchbar und wurde in den Simulationsläufen lediglich als Kontrollgröße mitberechnet.

Der normierte Problemdurchsatz d wird - im Gegensatz zum Dispatcher-Eigenverbrauch - auf die totale simulierte Laufzeit T_{tot} bezogen und stellt bei einer vorgegebenen Prozessorbelegung eine objektive Leistungskenngröße zur Beurteilung der Dispatcher-Entwürfe dar.

Wegen der breit gestreuten Variation der Eingangsparamter wird die simulierte Laufzeit T_{tot} nicht vorgegeben, sondern nach einer definierten Stichprobenzahl aufakkumulierter Arbeitsphasen der Benutzerprozesse gemessen. Diese Methode garantiert, daß alle Simulationsläufe - unabhängig von der Konstellation der Eingangsparameter - bis zum Erreichen eines vorgegebenen statistischen Fehlers des Messergebnisses für T_a durchgeführt werden.

Der Stichprobenumfang N läßt sich nach dem Grenzwertsatz von Lindeberg-Levy [12, 24] abschätzen.
Wenn die mittlere Ankunftszeit μ der exponentiell verteilten Zufallsereignisse mit der Standardabweichung σ den relativen Fehler f mit der statistischen Sicherheit S nicht überschreiten soll, so muß

$$f \geq \frac{U_\alpha \cdot \sigma/\sqrt{N}}{\mu} \qquad (4.4 - 5)$$

gewählt werden.

Daraus läßt sich die minimale Stichprobenzahl N zu

$$N \geq \left(\frac{U_\alpha}{f}\right)^2 \cdot \left(\frac{\sigma}{\mu}\right)^2 \qquad (4.4 - 6)$$

ermitteln, wobei U_α als die Signifikanzschwelle der N (0,1) - Verteilung aus der Lösung des Integrals

$$S = \frac{1}{\sqrt{2\pi}} \int_{-U_\alpha}^{+U_\alpha} \exp\left(-\frac{1}{2}x^2\right) dx \qquad (4.4 - 7)$$

bestimmbar ist.

Da für exponentiell-verteilte Zufallszahlen $\sigma = \mu$ wird, ergibt sich unter Zugrundelegung einer statistischen Sicherheit S von 97 % und eines relativen Fehlers f von 5 % mit $U_\alpha = 2{,}21$ eine minimale Stichprobenzahl von $N \approx 1960$.
Die Zahl der Stichproben wurde daher auf 2000 festgelegt.

Neben dem normierten Problemdurchsatz wurde bei allen Simulationsläufen eine Prozess-Umschaltzeit T_{switch} gemessen, die aus der Summe der Mittelwerte für die Ausführung der assign- und wait-Funktion gebildet wurde:

$$T_{switch} = T_{assign} + T_{wait} \qquad (4.4 - 8)$$

Sie stellt die einfachste Form der Prozessumschaltung dar, in der der ausführende Prozessor die Dispatcher-Funktionsfolge wait und assign ausführen muß.

Diese Zeit ist für Realzeitanwendungen oft wichtiger als der normierte Problemdurchsatz und vermittelt außerdem einen guten Einblick in die dynamischen Verhältnisse des Dispatchers (es wäre bei zwei untersuchten Dispatchervarianten E_1 und E_2 durchaus möglich, daß E_1 einen höheren normierten Problemdurchsatz als E_2, gleichzeitig aber eine höhere Prozess-Umschaltzeit aufweist).

Der objektive Vergleich einer Reihe von Dispatchervarianten bei unterschiedlichen Parameterkonstellationen des Modells erfordert ferner die Einstellung eines konstanten 'Arbeitspunktes' in allen Simulationsläufen.

Als Bezugsgröße, die durch alle Simulationsläufe hindurch konstant zu halten ist, bietet sich die Prozessorbelegung b an. Sie kann bei Vernachlässigung des Dispatcher-Eigenverbrauchs e leicht aus der Zahl der Prozessoren und der Anzahl sowie den Kenngrößen der Prozesse abgeleitet werden. Ist

w_{sw} die Wahrscheinlichkeit dafür, daß

auf eine Aktiv-Phase eines Benutzerprozesses eine Short-Wait-Phase folgt und

w_{lw} die Wahrscheinlichkeit dafür, daß

auf eine Aktiv-Phase eine Long-Wait-Phase folgt, dann gilt wegen

$$w_{sw} + w_{lw} = 1 \qquad (4.4 - 9)$$

$$w_{sw} = \frac{h}{1 + h} \qquad (4.4 - 10)$$

$$w_{lw} = \frac{1}{1 + h} \qquad (4.4 - 11)$$

wobei h die bereits eingeführte Häufigkeit zwischen Short- Wait- und Long-Wait-Phasen darstellt.

Wird mit p die Zahl der Prozesse im System bezeichnet, dann gilt für die Prozessorbelegung b:

$$b = \frac{p}{r} \cdot \frac{a}{a + w_{sw} \cdot sw + w_{lw} \cdot lw} \qquad (4.4 - 12)$$

Bei vorgegebener Prozessorbelegung b sowie den Eingangsgrößen für a, h, sw und lw folgt aus Glg. (4.4-12) mit (4.4-10, 4.4-11):

$$p = \frac{b \cdot r}{a} \left(a + \frac{h}{1 + h} \cdot sw + \frac{1}{1 + h} \cdot lw\right) \qquad (4.4 - 13)$$

Glg. (4.4 - 13) diente als Grundlage der Modellinitialisierung, wobei b in allen Versuchen konstant gehalten wurde. Die tatsächliche Prozessorbelegung b, die sich im Modell einstellt, kann wegen der Berücksichtigung des Dispatcher-Eigenverbrauchs erheblich von dem theoretischen Erwartungswert abweichen, der durch Glg. (4.4 - 13) ausgedrückt wird.
Entsprechend dem ermittelten Wert für p wird in der Initialisierungsphase eine feste Zahl von PCB's allokiert, mit der notwendigen Kontrollinformation aufgefüllt und in die BEREIT-Liste des Dispatchers eingespeist.
Die Supervisorprozesse werden in den BLOCKIERT-Zustand initialisiert.

Da dieser Ausgangszustand im Verlauf einer Simulation stets wiederkehren kann, stellt er gleichzeitig einen der eingeschwungenen Zustände des Modells dar, so daß auf zusätzliche Maßnahmen verzichtet werden konnte.

Die Prozessoren werden schließlich durch einen Konsol-Start-Interrupt aktiviert und nehmen durch Aufruf der assign-Funktion ihre Arbeit auf.

4.5 Wahl der Eingangsparameter

Die große Zahl unabhängiger Eingangsgrößen sowie die Forderung nach ökonomisch vertretbaren Simulationskosten verlangen ein sorgfältiges Vorgehen bei der Auswahl der Parametersätze.

Aus dem Spektrum möglicher Belastungsprofile wurden vier repräsentative Parametersätze für die Benutzerprozeßkenngrößen a, sw, lw festgelegt, die eine aussagekräftige Klassifizierung ermöglichen und in der nachstehenden Tabelle zusammengefaßt sind:

Prozess-Profilklasse	a	sw = lw
Klasse I: Ein/Ausgabe - intensive Prozesse mit hoher Service-Request-Rate	1 msec	15 msec
Klasse II: Ein/Ausgabe - intensive Prozesse mit niedriger Service-Request-Rate	5 msec	75 msec
Klasse III: Rechenintensive Prozesse mit hoher Service-Request-Rate	5 msec	15 msec
Klasse IV: Rechenintensive Prozesse mit niedriger Service-Request-Rate	25 msec	75 msec

In jeder Klasse wurden ferner drei verschiedene Kopplungsgrade untersucht:

- lose Kopplung mit $k \approx 0$ ($h = 10^5$)
- gemischte Kopplung mit $k = \frac{1}{11}$ ($h = 10$)
- feste Kopplung mit $k \approx 1$ ($h = 10^{-5}$)

Die Größen sw und lw werden unabhängig von der Bedeutung der Warteabschnitte gleichgesetzt, um bei Variation des Kopplungsgrades den durch Glg. (4.4 - 13) eingestellten Arbeitspunkt nicht zu verschieben. Der quantitative Vergleich der gemessenen Kurven wird dadurch objektiviert.

Die Prozessorbelegung b wurde für alle Messreihen konstant zu 1 angenommen und entspricht daher der theoretischen Vollast des Modellsystems. Dieser Arbeitspunkt wurde deshalb gewählt, weil die Konfliktwahrscheinlichkeit der Prozessoren im Dispatcher am größten ist und die charakteristischen dynamischen Eigenschaften sehr viel deutlicher zutage treten als bei geringen Prozessorbelegungen.

Mit $b = 1$ und $lw = sw$ vereinfacht sich Glg. (4.4 - 13) zu

$$\frac{p}{r} = \frac{a + sw}{a} \qquad (4.4 - 14)$$

Für die auf die Prozessoren bezogene Zahl von Prozessen ergeben sich deshalb für die Prozess-Profilklassen I - IV folgende Werte:

Klasse I + II $\quad \frac{p}{r} = 16$,

Klasse III + IV : $\quad \frac{p}{r} = 4$

Vom Entwurf B wurden durch Variation der Länge der Prozessliste l, die als zusätzlicher Eingangsparameter existiert, drei Varianten für $l = p$, $l = \frac{p}{2}$ und $l = 1$ durchgemessen.

Die Variante für $l = 1$ stellt dabei einen interessanten Sonderfall dar, der dem Entwurf A mit der Ausnahme entspricht, daß die bereiten Prozesse zusätzlich zur übergeordneten Prozessliste in einer eigenen BEREIT-Liste formiert sind.

Es ergeben sich auf diese Weise 12 Messreihen, die bei Variation der Prozessorzahl r von 1 ... 16 und 4 Dispatcher-Varianten $4 \times 12 \times 16 = 768$ Simulationsläufe erfordern würden. Für die Varianten des Entwurfs B wurden wegen der stetigen Kurvenverläufe in der Regel nur Konfigurationen mit einer geraden Zahl von Prozessoren durchgemessen, so daß die Zahl der Simulationsläufe auf ca. 520 gesenkt werden konnte. Bei einer mittleren Programmlaufzeit von ca. 10 min pro Simulationslauf ergab sich damit ein Gesamtbedarf von etwa 80 Rechenstunden auf der IBM 370/165.

Die retire-Funktion und die add-Funktion mit dem policy-Paramter ='preemptive' waren in keiner der Simulationsläufe involviert, da sie ausschließlich ein Mittel für die Realisierung von Scheduling-Disziplinen darstellen, die nicht Gegenstand der Untersuchungen waren. Sie wurden allerdings in getrennten Testläufen auf ihre Funktionstüchtigkeit überprüft.

4.6 Diskussion der Resultate

Die Messergebnisse sind - getrennt nach normiertem Problemdurchsatz und Prozess-Umschaltzeit in den Diagrammen Abb. 15 - 18 für die 4 Prozess-Profilklassen aufgetragen. Die interessantesten Kurven sind diejenigen der Klassen I + II, die die größte Belastung des Dispatchers zeigen.

Unterstellt man, daß bei Prozess-Profilen der Klasse I jeweils ein wait (release) -activate-assign-Zyklus für einen Benutzerprozess und den eingeschalteten Supervisor-Prozess in einer Millisekunde verarbeitet werden, so ergibt sich bei Konfigurationen mit 16 Prozessoren eine mittlere theoretische Aufruffolge für Dispatcher-Funktionen von 1000/6.16 = 12 µsec.

Man entnimmt den Kurven Abb. 15-a, daß der Dispatcher-Entwurf A diesen Belastungen nur für 1 - 2 Prozessoren gewachsen ist, danach, und zwar von 3 - 4 Prozessoren aber bereits starke Sättigungserscheinungen aufweist, die sich in einem spürbaren Abflachen der Kurve bemerkbar machen.
Bei 5 Prozessoren ist dann sogar ein schroffer Rückgang des erzielten Problemdurchsatzes zu erkennen, der in Anlehnung an ähnlich gelagerte Phänomene bei der Speicherorganisation [8] in der Literatur unter dem Begriff 'Prozessor Thrashing' [39] diskutiert wird und hier erstmals meßtechnisch nachgewiesen werden konnte.

Der Übergang vom Sättigungsverhalten zum Thrashing ist im vorliegenden Fall dadurch zu erklären, daß der zusätzlich ins System eingebrachte Prozessor auf Grund des erreichten Sättigungsgrades im Dispatcher keine nützliche Arbeit mehr leisten kann, durch den Zugriff zur Datenbasis aber die übrigen Prozessoren in der Ausführung von Dispatcher-Funktionen behindert. Der Grad der Störung ist dabei stark von der Partitionierung der Datenbasis abhängig und bestimmt daher maßgeblich den Verlauf der Kurve nach Erreichen des Sättigungswertes.

Der Abfall ist am schroffsten für k = 0 und wird in allen Fällen mit steigendem Kopplungsgrad geringer. Für k = 1 (Abb. 15) ähnelt die Kurve stark einer normalen Sättigungskurve. Hier spiegelt sich das dynamische Gleichgewicht wider, in dem die überwiegende Zahl der Prozesse im SUSPENDIERT-Zustand auf ihre Reaktivierung warten. Dies bedeutet, daß ein großer Teil der Suchvorgänge nun im Scheduler durchgeführt werden, die auf Grund der Modelleigenschaften nicht in die Ablaufzeiten eingehen. Die Dispatcherlisten sind gegenüber dem Fall $k \approx 0$ kurz, so daß die Suchzeiten nach Prozessen drastisch reduziert und der Störgrad durch im Sättigungsbereich arbeitende Prozessoren vermindert wird.
Eine weitere Konsequenz dieses dynamischen Gleichgewichts ist der scheinbar höhere normierte Problemdurchsatz, der für alle Dispatcher-Varianten bei steigendem Kopplungsgrad k erzielt wurde. Da große Teile der für $k \approx 0$ im Dispatcher erledigten Suchvorgänge in den Scheduler verlegt werden (wo sie entsprechend den Modellierungsvoraussetzungen keine Zeit kosten), steigt der normierte Problemdurchsatz mit wachsendem k an. Die Kurvenscharen 15 a - c können deshalb nicht unmittelbar, sondern nur unter Berücksichtigung der sich ändernden dynamischen Situation verglichen werden.

Ein Vergleich des Entwurfs B (l = 1) mit dem Entwurf A für die Prozess-Profilklasse I zeigt, daß nur im Sättigungsbereich eine deutliche Steigerung des Problemdurchsatzes für den Entwurf B erzielt wurde. Dieses Ergebnis ist auf das zyklische Ablaufverhalten der Prozesse zurückzuführen:

da die Zustandswechsel entweder in der Folge

AKTIV → BLOCKIERT → BEREIT → AKTIV ... bzw.
AKTIV → SUSPENDIERT → BEREIT → AKTIV

erfolgen, ist es wenig effektiv, nur einen der bestehenden Engpässe in diesem Zyklus zu beseitigen. Im Falle des Entwurfs B (l = 1) wurden die Zugriffszeiten bei der Durchführung des Zustandswechsels von BEREIT→AKTIV durch die zusätzliche Ein-

führung einer BEREIT-Liste zwar weitgehend eliminiert, die Umschaltzeit eines Prozesses aus dem BLOCKIERT-Zustand wird dadurch aber nicht reduziert. Sie stellt den dominierenden Anteil der Zeiten dar, die für die Durchführung eines zyklischen Zustandswechsels aufgewendet werden muß und überdeckt damit alle übrigen Zeitbeiträge.

Eine durchgreifende Verbesserung des Dispatcher-Verhaltens wurde dagegen für die Varianten des Entwurfs B ($l = p$, $l = \frac{p}{2}$) gemessen. Die Abb. 15 a - c zeigen ein nahezu ideales Verhalten des Verlaufes der Kurven für den normierten Problemdurchsatz. Für Konfigurationen bis zu 14 Prozessoren zeigt der Dispatcher selbst für den kritischen Fall der losen Kopplung ($k = 0$) praktisch keine Sättigungserscheinungen. Erst zwischen 14 und 16 Prozessoren flacht die Kurve ab, ohne jedoch den Sättigungsendwert schon zu erreichen.

In allen vier gemessenen Profilklassen wurden nahezu identische Kurven für die Entwürfe B ($l = p$) und B ($l = \frac{p}{2}$) gemessen. Sie zeigen, daß die Wahl der Länge der Prozess-Hash-Tabelle unproblematisch ist und lediglich in der Größenordnung mit der Zahl der im System befindlichen Prozesse übereinstimmen muß.

Die Kurven für die Prozess-Umschaltzeiten vermitteln eine Erklärung für die Ursache der Sättigungs- und Thrashing-Erscheinungen. Während die Zeiten für den Entwurf A zwischen drei Größenordnungen - abhängig von der Zahl der Prozessoren schwanken, bleiben sie für die Varianten $l = p$, $l = \frac{p}{2}$ des Entwurfs B nahezu unverändert. Diese Resultate beweisen, daß sich der Partitionierungsgrad der Datenbasis und die Suchzeiten in den Listen in ihrer Wirkung multiplizieren und deshalb einen dominierenden Einfluß auf die dynamischen Eigenschaften des Dispatchers ausüben.
Im Gegensatz zu den abfallenden, durch Thrashing ausgelösten Kurvenverläufen für den Problemdurchsatz, sind die Kurven für die Prozess-Umschaltzeit reine Sättigungskurven. Ähnlich den Kurven für den Problemdurchsatz sind auch hier die charakteristischen Merkmale der Verläufe besonders deutlich für $k = 0$ aus-

gebildet, da mit wachsendem Kopplungsgrad durch die Abwanderung der Prozesse aus dem Dispatcher die gemessenen Prozess-Umschaltzeiten sinken.

Die Analyse der Kurven für die Prozess-Profilklassen II - IV in den Abb. 16 - 18 zeigt ein ähnliches Verhalten wie das der diskutierten Prozess-Profilklasse I. Der Unterschied liegt in dem verschobenen Einsatzpunkt für die Sättigungs- und Thrashing-Erscheinungen, der auf die abnehmende Belastung des Dispatchers zurückzuführen ist. Für die Profilklasse IV wurden für keine der Dispatcher-Varianten Sättigungserscheinungen gemessen. Dieses Resultat ist verständlich, wenn man bedenkt, daß der Dispatcher-Eigenverbrauch in keinem gemessenen Falle 6 % übertraf. Die geringfügigen Unterschiede in den Kurvenverläufen waren vernachlässigbar und konnten in dem Diagramm nicht mehr vollständig aufgelöst werden.

Entsprechend den gewonnenen Resultaten enthalten die im Anhang aufgeführten PL/1-Programme für die Entwürfe A und B kurze Angaben über die bevorzugten Prozess-Profilklassen, die als Entscheidungshilfe beim Einsatz der Dispatcher gedacht sind.

Abschließend sei festgestellt, daß sich die gewonnenen Meßergebnisse unempfindlich gegenüber den Änderungen einzelner Zeitbewertungen für Statements erwiesen haben.

Diese Erfahrung ist das Ergebnis einer umfangreichen Experimentier- und Testphase, in der große Teile des Modells mehrfach geändert und Zeitbewertungen (z. B. für die Zuweisung und Freigabe von Speicherbereichen für Koppelglieder) um max. den Faktor 5 variiert wurden.
Das 'vernünftige' und in weit über 500 Simulationsläufen stabile Verhalten des Modells, das qualitative Erwartungen letztlich quantitativ bestätigte, hat ein hohes Maß an subjektivem Vertrauen in das Modell und die zugrunde gelegte Simulationsmethode erzeugt. Darüberhinaus ist jedoch als wesentlicher Bestandteil

der zukünftigen Aufgaben geplant, eine - objektiven Bewertungsmaßstäben zugängliche - Gültigkeitsbestätigung (Validation) für das Modell und die Simulationsmethode durchzuführen [11, 45].

Die oft angewendete Technik der Validierung durch einen Vergleich des Modells mit existierenden Systemen ist im vorliegenden Fall wenig sinnvoll, da die dort angewendeten Organisationsprinzipien stark von den hier vorgestellten abweichen und Erfahrungen außerdem nur für Konfigurationen bis maximal 3 Prozessoren vorliegen. Sie stellen aber, wie aus der Diskussion der Resultate sichtbar wurde, den weniger kritischen Bereich des Konfigurationsspektrums dar.

Als eine weiterführende Aufgabe des Projektes ist jedoch vorgesehen, auf neuen Rechnern prototypische Implementierungen des vorgestellten Betriebsorganisationskonzepts vorzunehmen und durch die gewonnenen praktischen Erfahrungen die Simulationsmethode und die darauf aufbauenden Modelle stufenweise zu validieren.

5. Zusammenfassung

Ziel dieser Arbeit war der Entwurf alternativer Dispatcher für symmetrische Mehrprozessor-DV-Anlagen auf der Basis einer generalisierten Betriebsorganisation.

Ausgehend von dem Schichtenmodell Dijkstra's, das durch Einführung von Elementarfunktionen der Ablaufsteuerung verfeinert wurde, konnten acht Dispatcher-Funktionen anhand eines Zustandsmodells für Prozesse und Prozessoren definiert und in ihrer Wirkung beschrieben werden. Sie bildeten die Grundlage für den Entwurf von zwei alternativen Implementierungen und deren Varianten in PL/1. Auslegungskriterium war die Minimierung der Prozessorkonflikte, die durch parallel im Dispatcher arbeitende Prozessoren zustande kommen und einen starken Einfluß auf den unter vorgegebenen Belastungsprofilen erzielbaren Durchsatz ausüben. Beide Entwürfe unterscheiden sich durch den Partitionierungsgrad der Dispatcher-Datenbasis, der eine der wesentlichen Einflußgrößen für die Ausbildung von Prozessor-Konflikten darstellt.

Die PL/1-Programme dienten als Vorlage für den Aufbau eines Simulationsmodells, an dem die dynamischen Eigenschaften der Dispatcher-Entwürfe gemessen wurden. Simulationsinstrument war eine virtuelle PL/1-Maschine, die Betriebssystemmodelle in einer mit Maschinensprachen vergleichbaren Modellierungstiefe darzustellen erlaubt und Programmlaufzeitmessungen auf der Basis von zeitgewichteten PL/1-Statements zuläßt.

Die Simulationsläufe wurden mit vier alternativen Belastungsprofilen durchgeführt, die auf einem einfachen Ablaufmodell für Benutzerprozesse basieren und unterschiedliche Anforderungen an den Parallelgrad des Dispatchers stellen. Wichtigstes Resultat der Messungen ist, daß sich zwei der Dispatcher-Varianten für das gemessene Konfigurationsspektrum von 1 - 16 Prozessoren praktisch invariant gegenüber den Belastungsprofilen erwiesen.

Damit wurde gezeigt, daß die als Folge von Generalisierungen unvermeidlichen Effektivitätsverluste in Betriebsorganisationen durch einen geeigneten Systemaufbau und sorgfältige, auf quantitative Bewertungsmethoden gestützte Implementierungen, weitgehend kompensierbar sind. Dieses Ergebnis kann als Bestätigung der Bemühungen um eine Standardisierung von Betriebssoftware gewertet werden, die das langfristige Ziel dieses Forschungsvorhabens ist und mit der Definition der verbleibenden Elementarfunktionen der Ablaufsteuerung und deren Erprobung weitergeführt wird.

Schlußwort

Die Grundideen für die vorliegende Arbeit wurden während eines einjährigen Studienaufenthaltes im Jahre 1971 am Thomas J. Watson-Laboratory der IBM in Yorktown Heights, USA im Rahmen eines Fellowship-Assignment entwickelt.

Mein besonderer Dank gilt Herrn Prof. G. Krüger, der mir durch die Vermittlung dieses Aufenthaltes und eine stetige Förderung letztlich die Anfertigung der vorliegenden Arbeit ermöglichte.

Prof. G. Krüger, Prof. G. Goos und Prof. H. Wettstein haben sich in Diskussionen kritisch mit dem gewählten Thema auseinandergesetzt und durch ihre Anregungen wesentliche Impulse für den erfolgreichen Verlauf der Arbeit geliefert.
Bei der Lösung von Teilproblemen, der Codierung sowie einer übersichtlichen Darstellung des Textes waren außerdem die Mitarbeiter des IDT, Dipl.-Ing. G. Fleck, Dipl.-Ing. H. Herbstreith, Dipl.-Ing. D. Hilse, Dipl.-Phys. E. Holler und Dipl.-Inf. M. Rupp, behilflich.

Allen sei an dieser Stelle für ihre wertvolle Unterstützung gedankt.

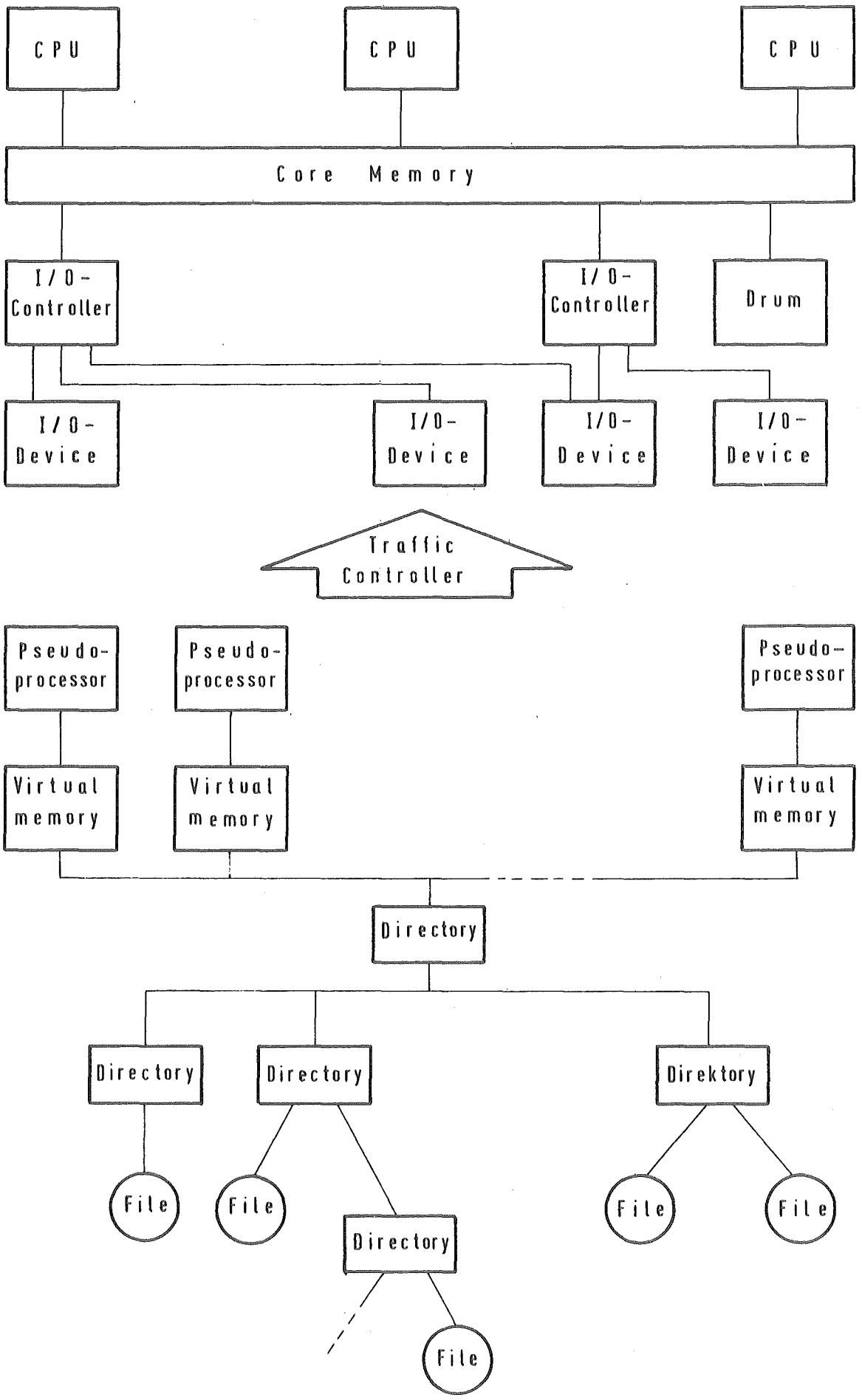


Bild 1: Die Transformation einer virtuellen Maschine in eine reelle Maschine durch den Traffic Controller nach Saltzer [39]

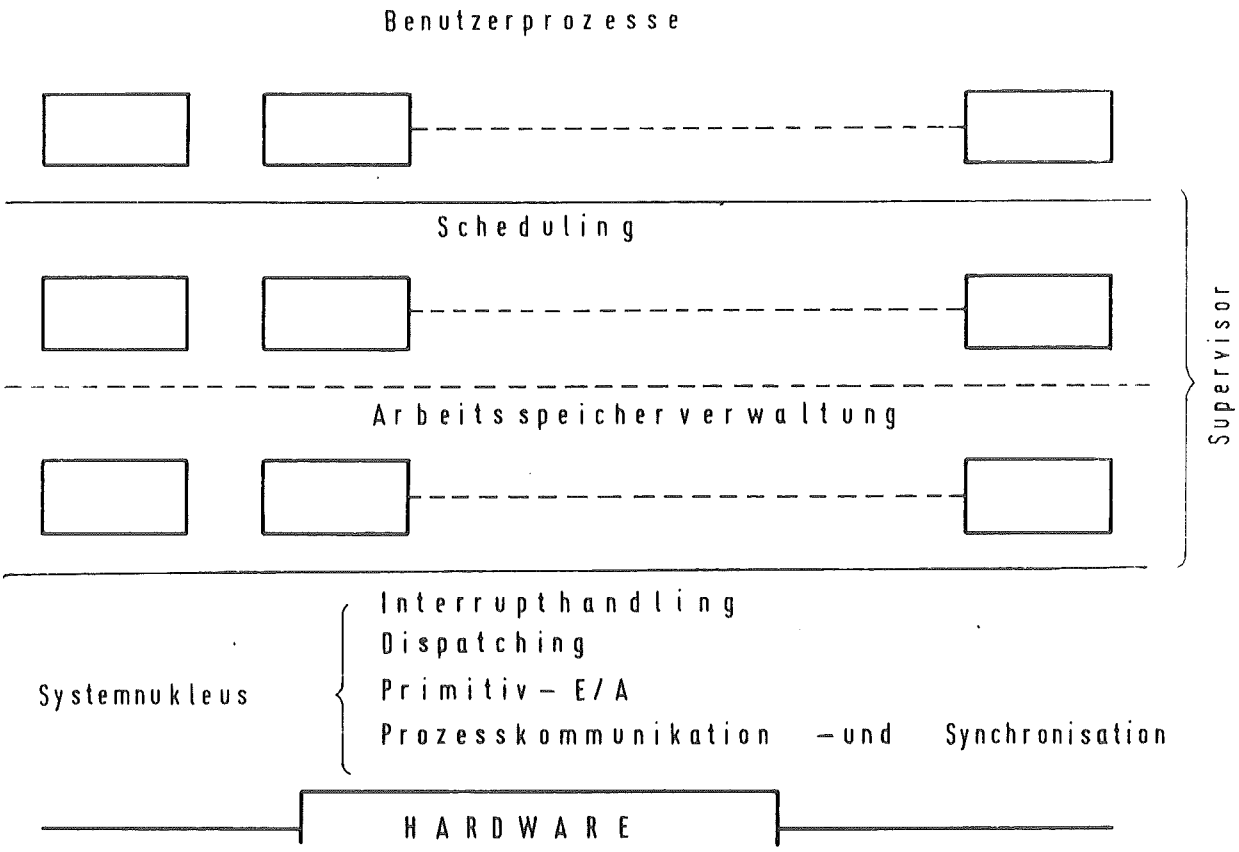


Bild 2 Schichtenstruktur einer Betriebsorganisation nach Dijkstra[4]

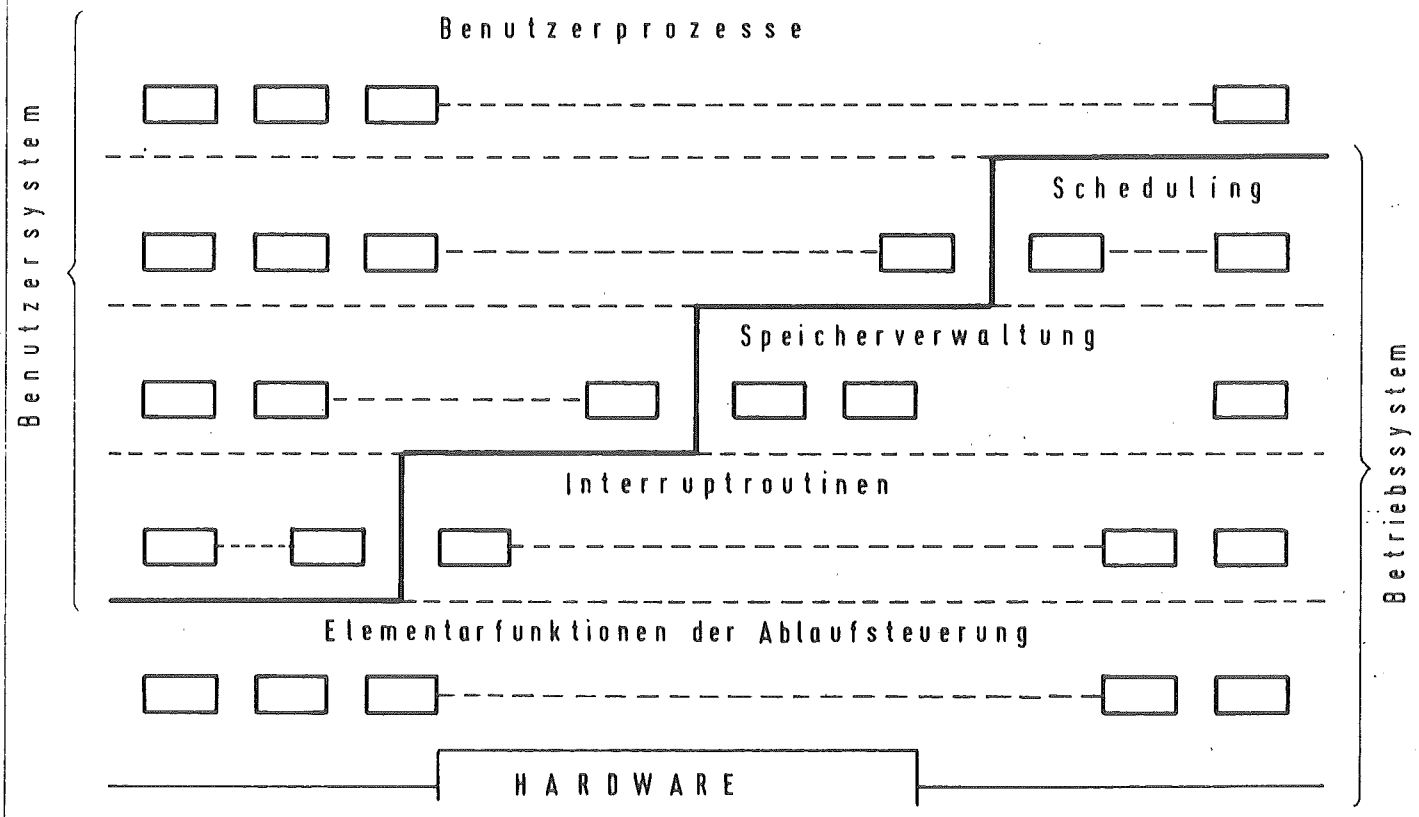


Bild 3 Modifiziertes Schichtenmodell durch Einführung von Elementarfunktionen

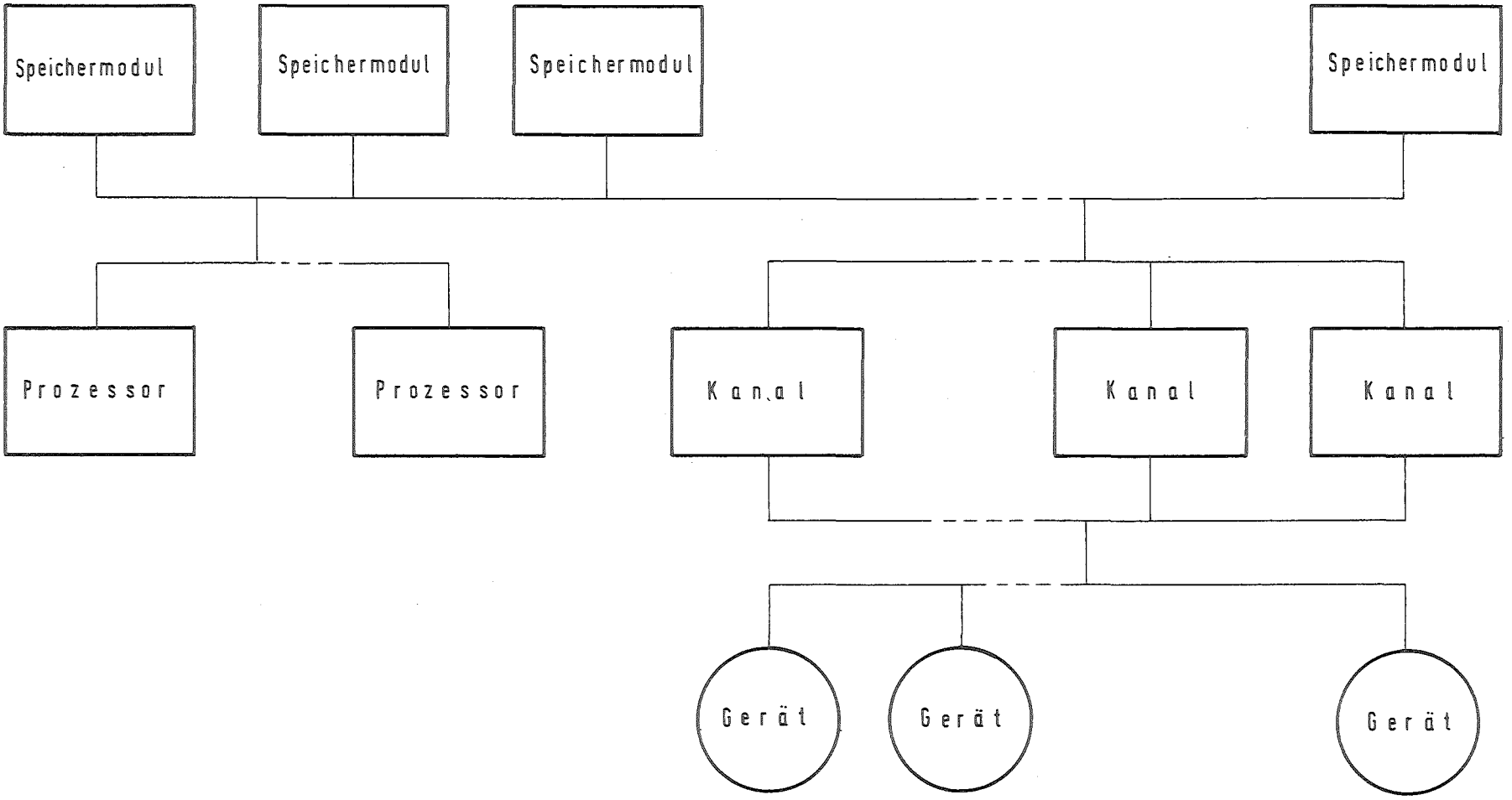


Bild 4: Schematische Darstellung eines symmetrischen Mehrprozessorsystems

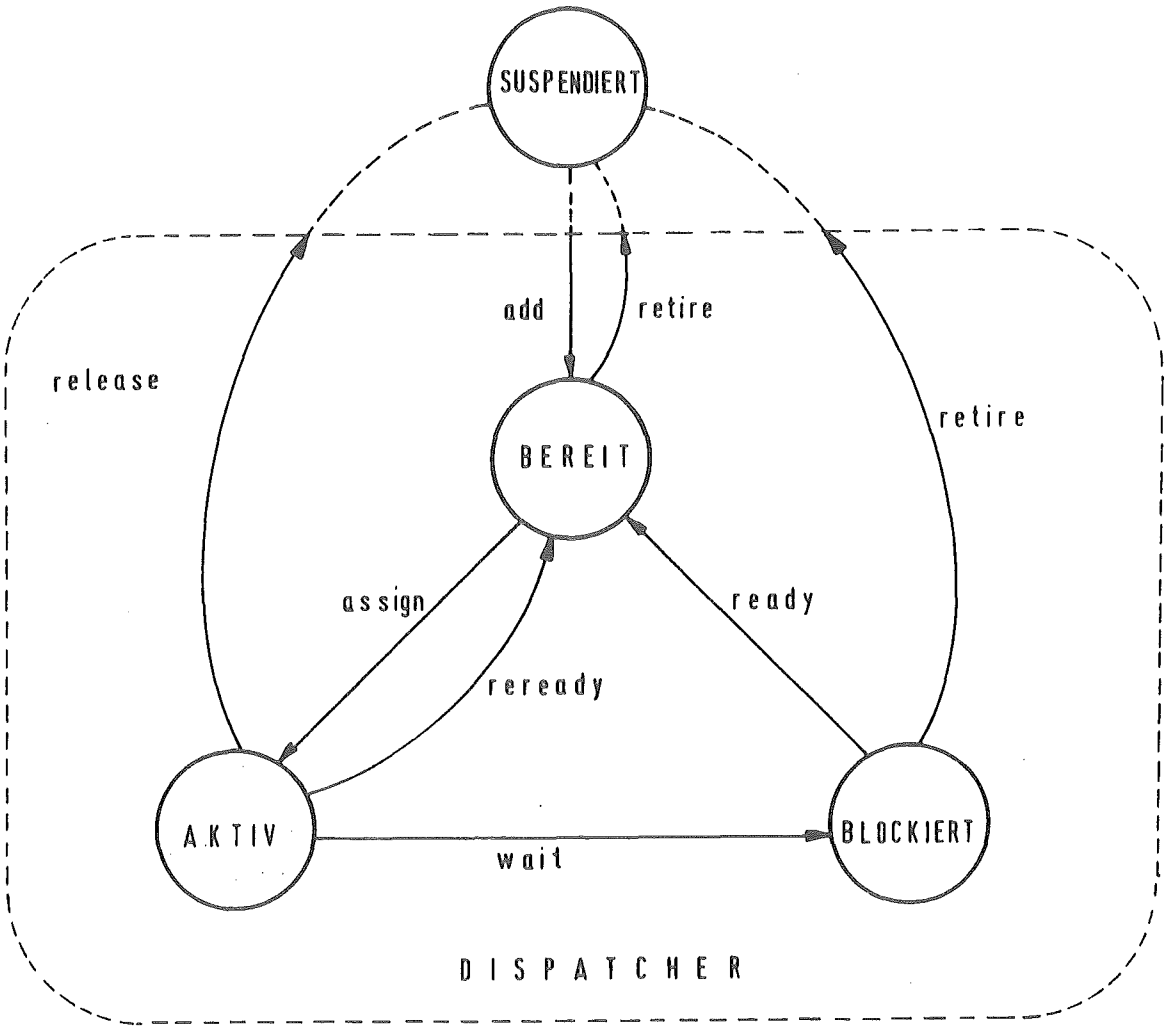


Abb. 5: Prozesszustände und Übergangsfunktionen

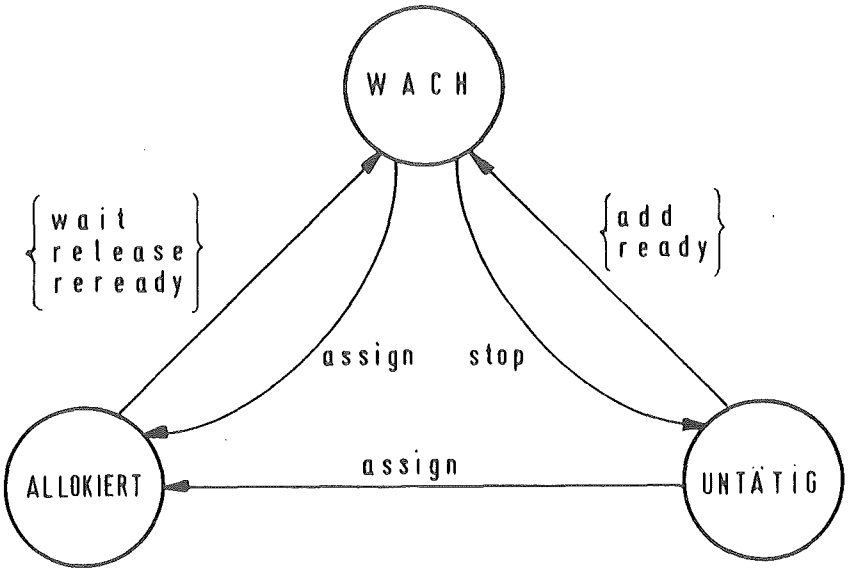


Abb. 6: Prozessorzustände und Übergangsfunktionen

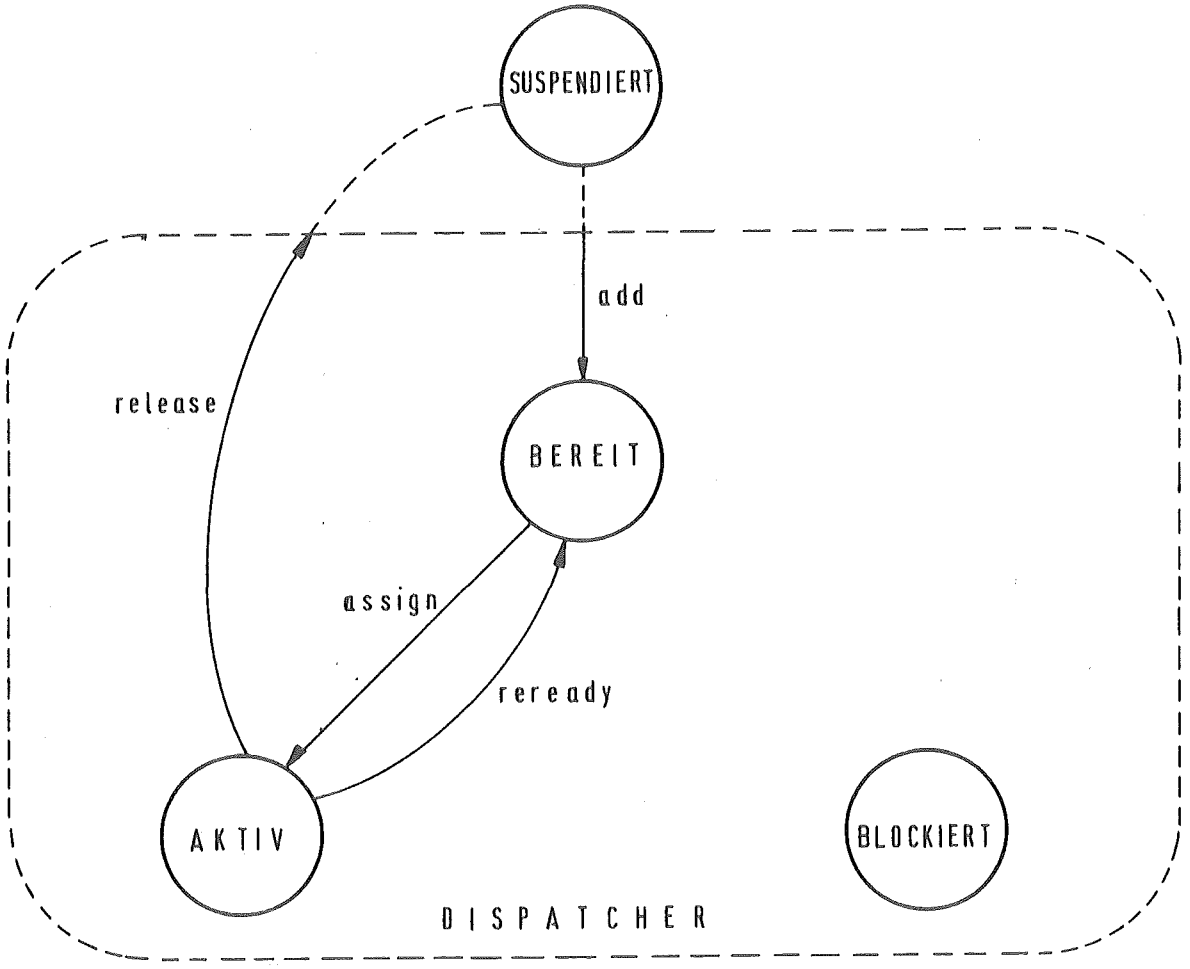


Abb.7: Involvierte Prozesszustände und Elementarfunktionen bei fester Kopplung Dispatcher – Scheduler

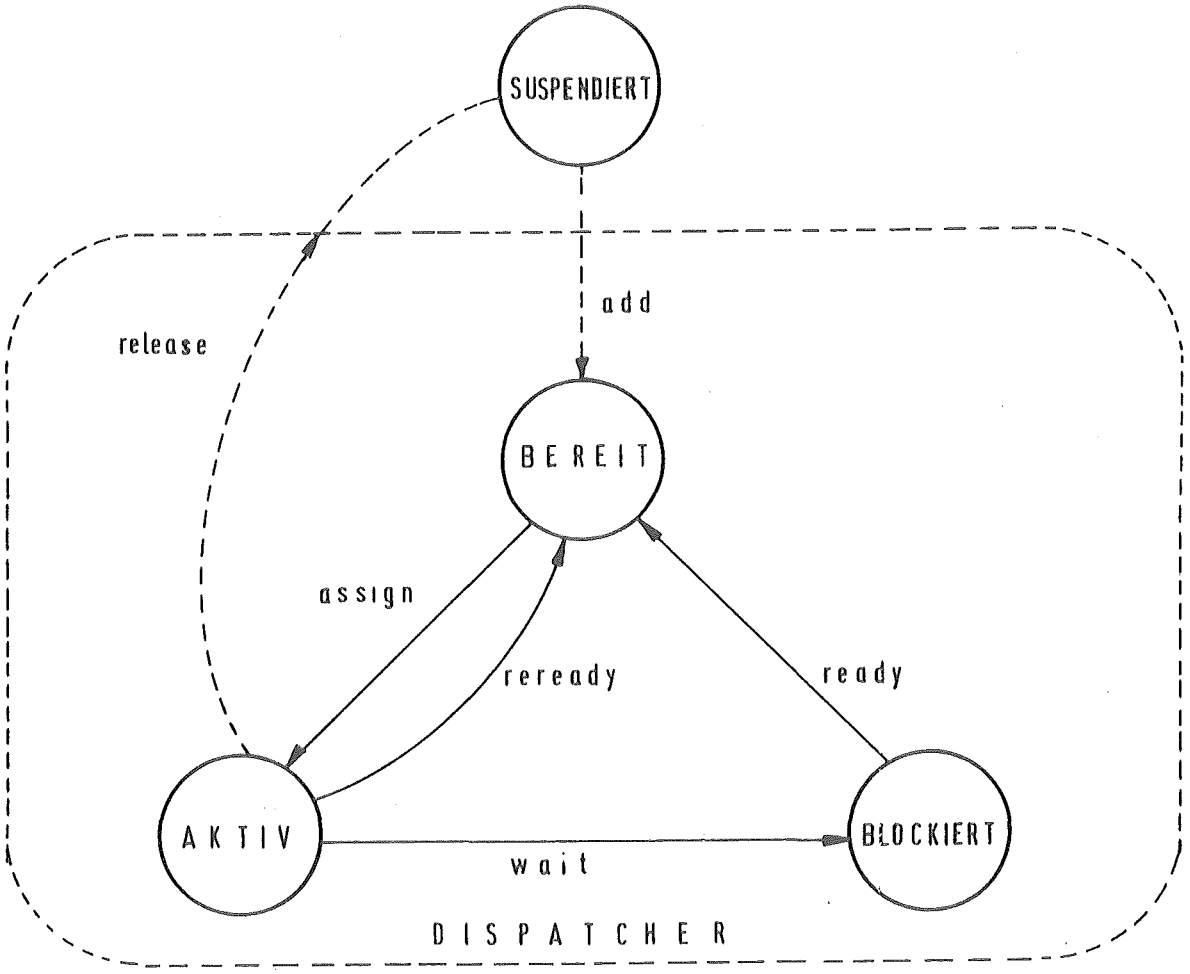


Abb 8.: Involvierte Prozesszustände und Elementarfunktionen bei loser Kopplung Dispatcher – Scheduler

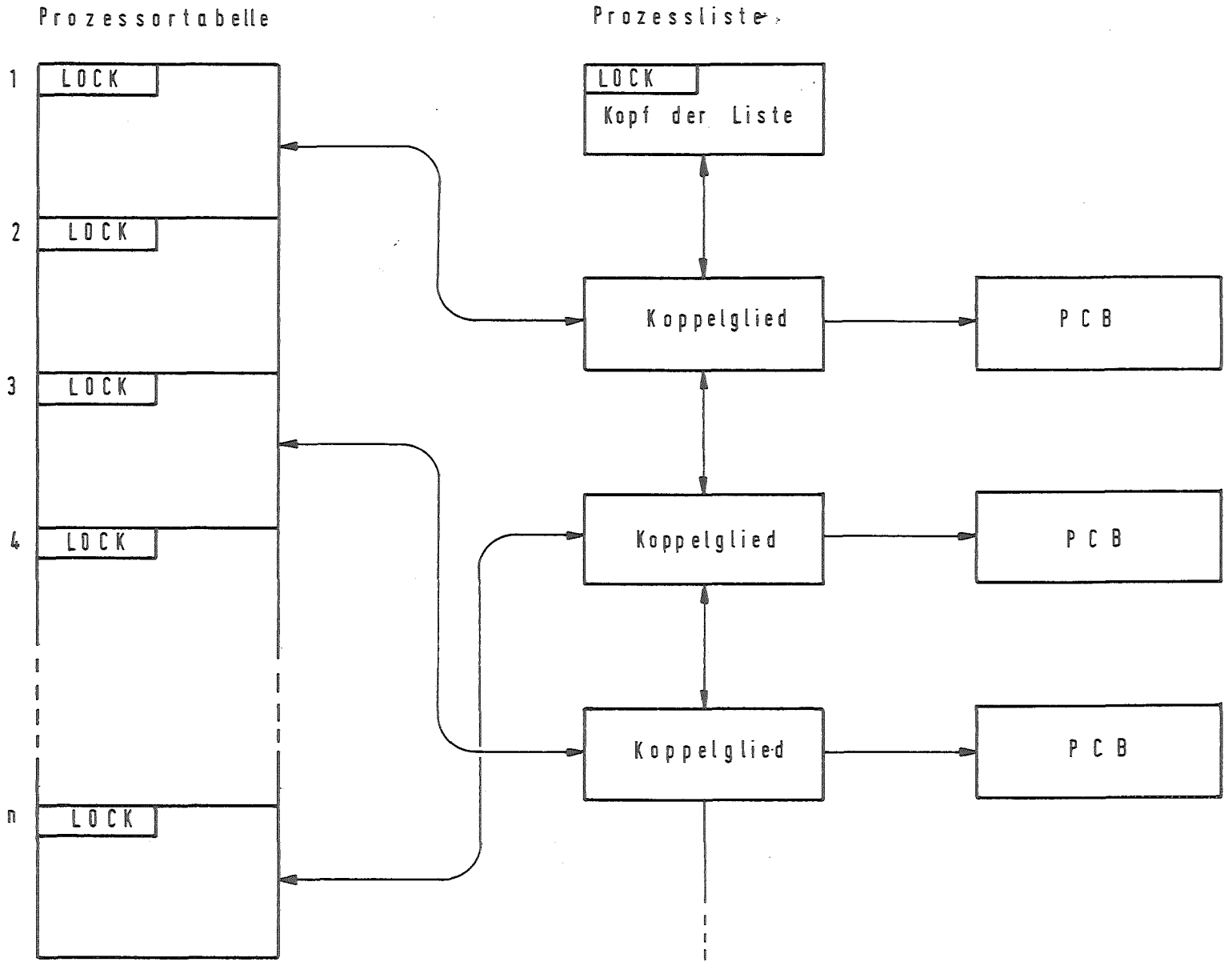


Abb. 9:

Die Datenstruktur des Dispatcher – Entwurfs A

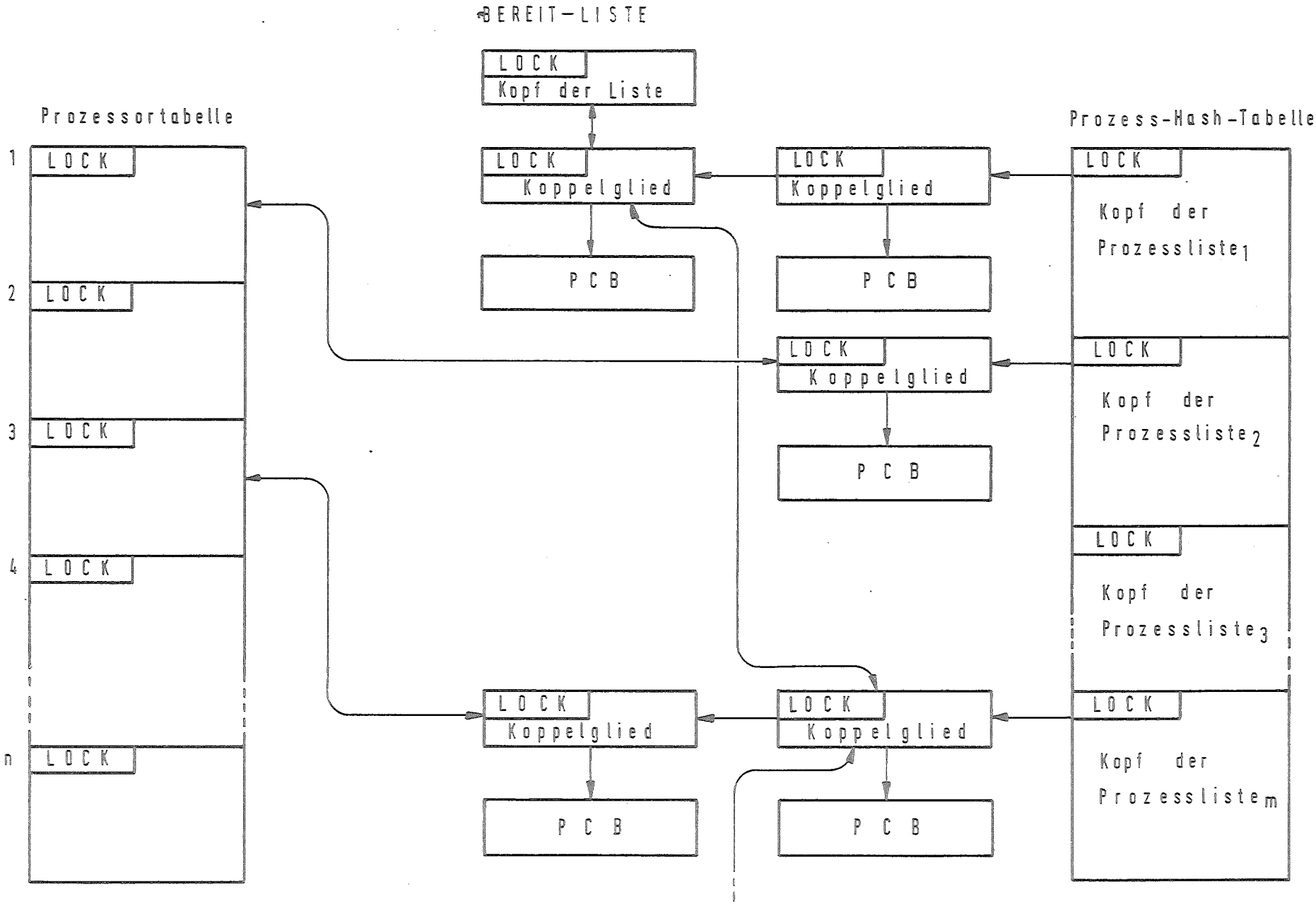


Abb. 10: Die Datenstruktur des Dispatcher-Entwurfs B

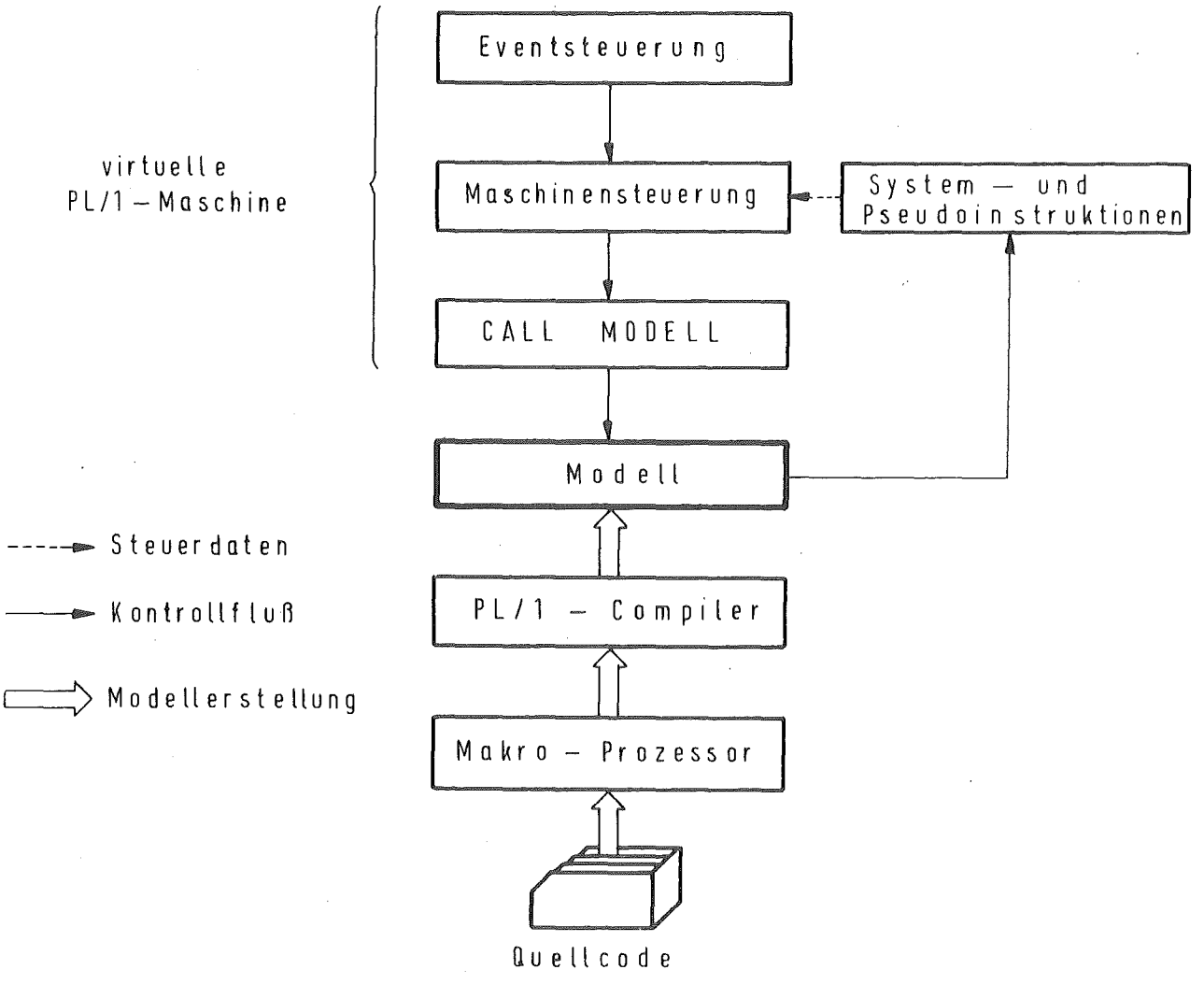
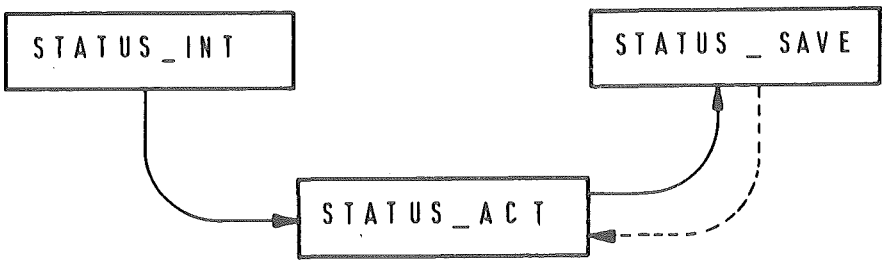


Abb.11: Organisation des Simulationssystems

Quellcode	transformierter Quellcode
	. . . GOTO $DSA.SLABEL; $L000:;
(1) $1 K=20;	$DSA.SZEITBEW=1; $DSA.SLABEL=L001; RETURN; $L001: K = 20;
(2) $1 I=0;	$DSA.SZEITBEW=1; $DSA.SLABEL=L002; RETURN; $L002: I = 0;
(3) $3 I=I+1;	$DSA.SZEITBEW=3; $DSA.SLABEL=L003; RETURN; $L003: I = I+1;
	. . .

Abb.12: Transformation des Quellcodes durch den Makroprozessor



—— Registertransporte nach Annahme eines Interrupts

------ Programmwechsel nach Ausführung einer $RETURN-Instruktion

Abb. 13:
Funktion der drei Registersätze für den Rechnerkernstatus

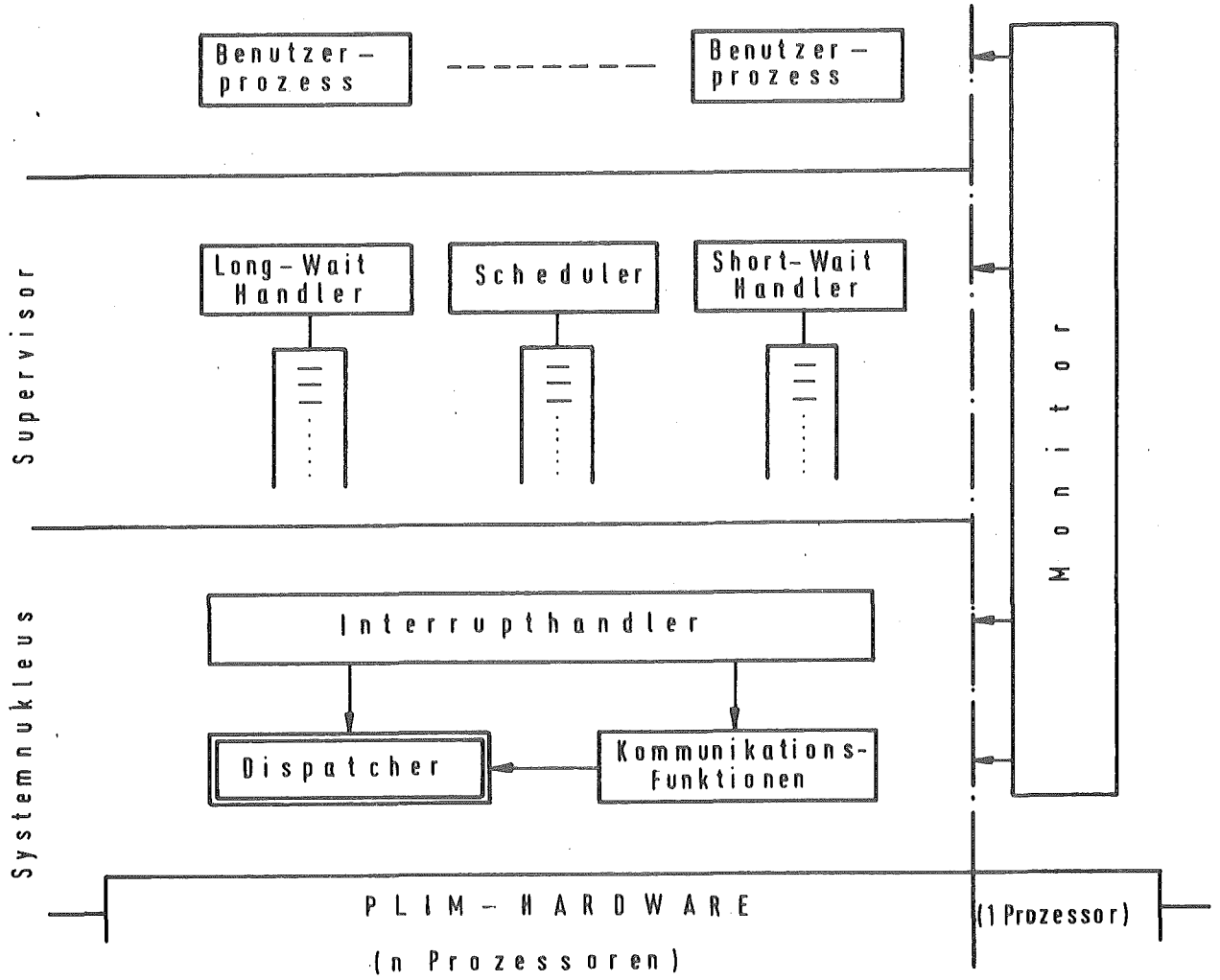
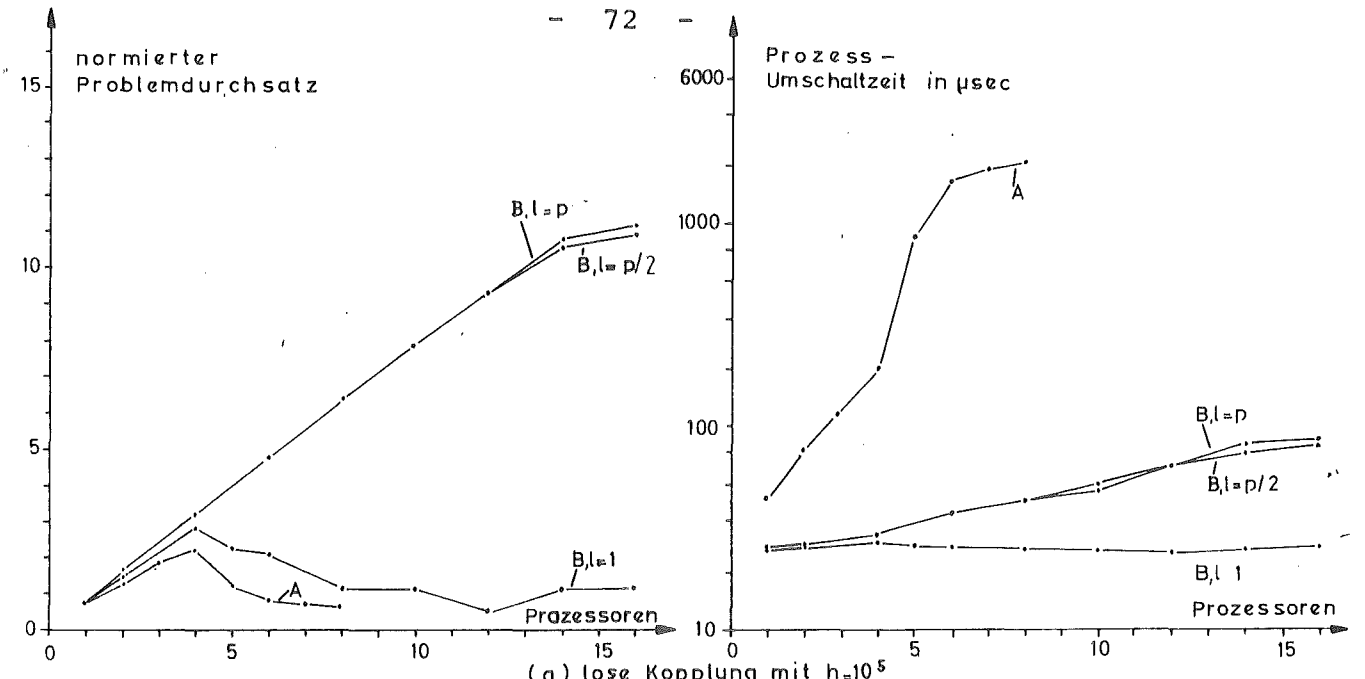
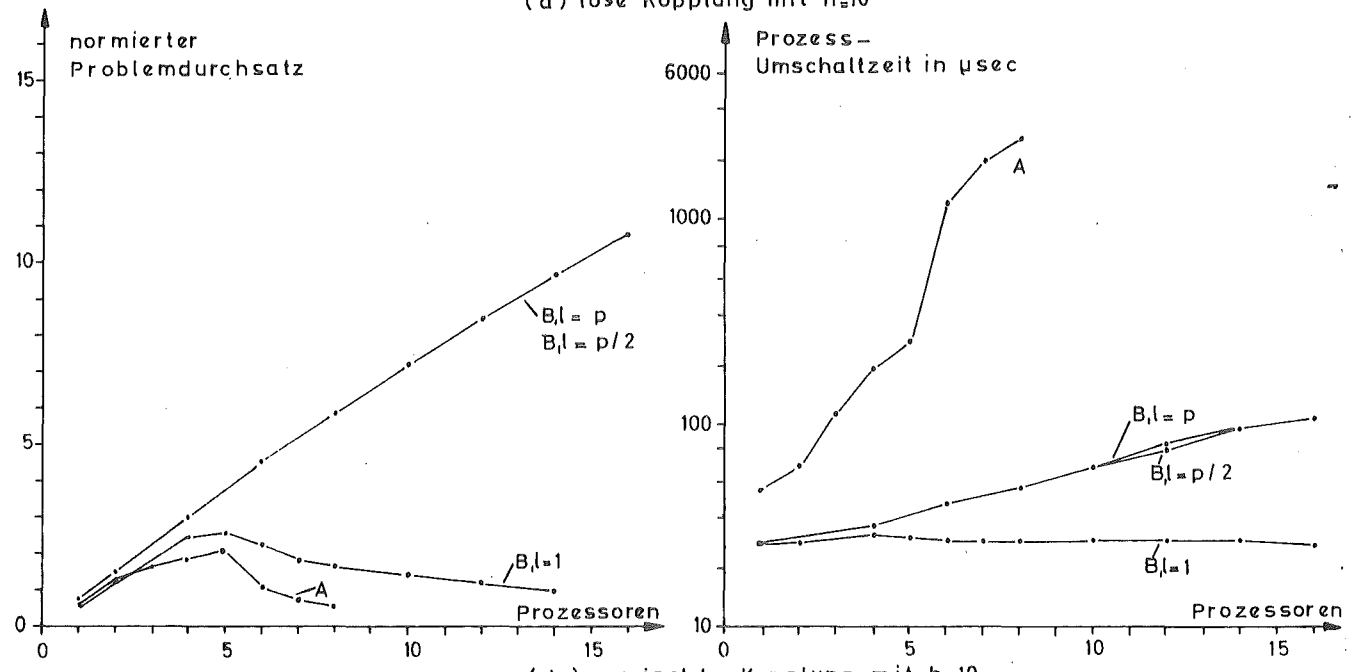


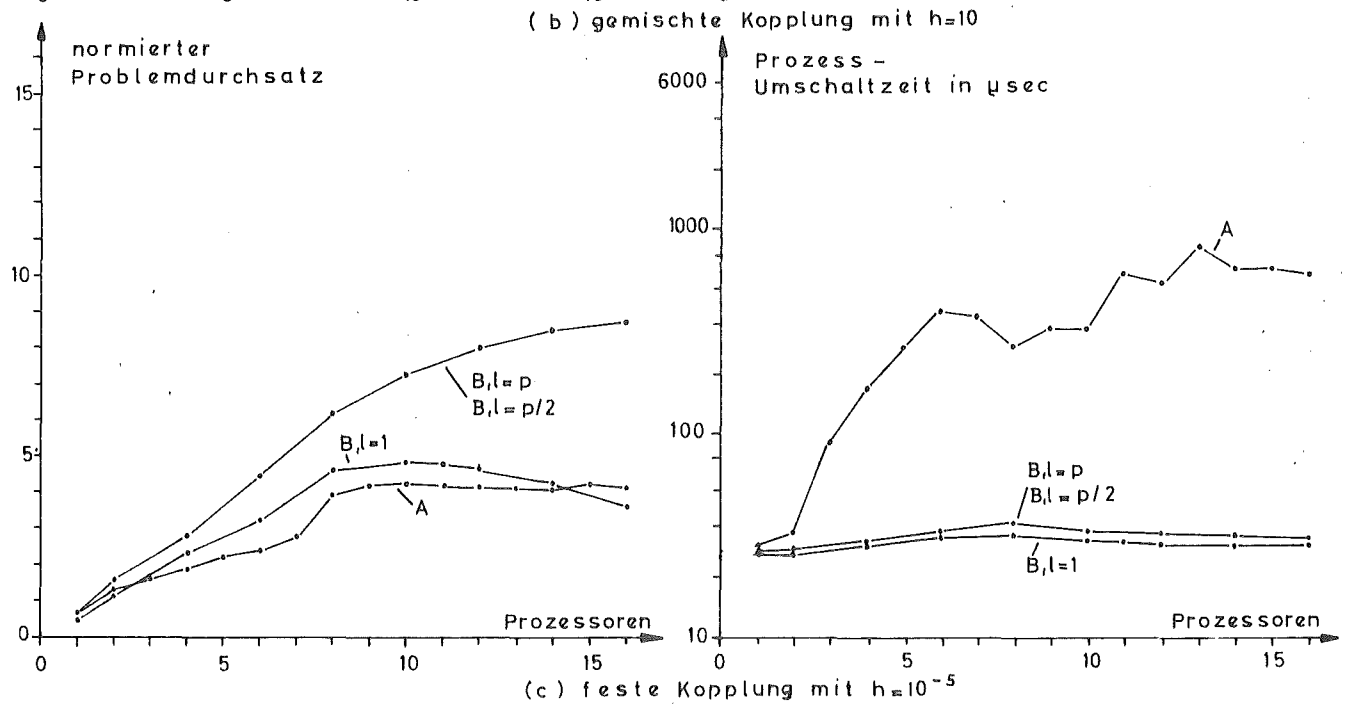
Abb. 14:
Organisation des Betriebssystemmodells



(a) lose Kopplung mit $h=10^5$



(b) gemischte Kopplung mit $h=10$



(c) feste Kopplung mit $h=10^{-5}$

Abb. 15: Gemessene Kurven für die Prozeß-Profilklasse I ($\frac{p}{r}=16$, a=1msec, sw=15msec, lw=15msec)

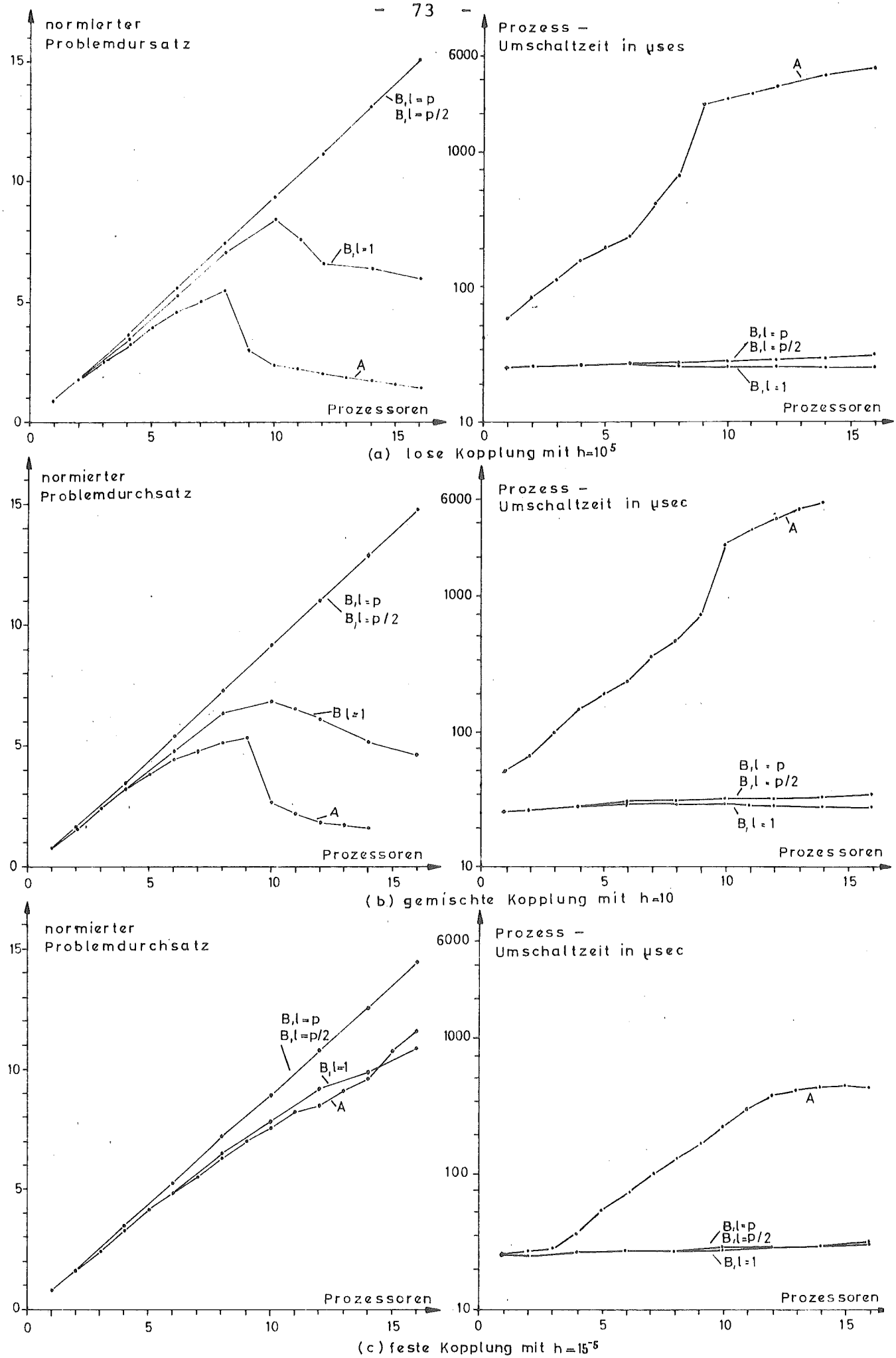
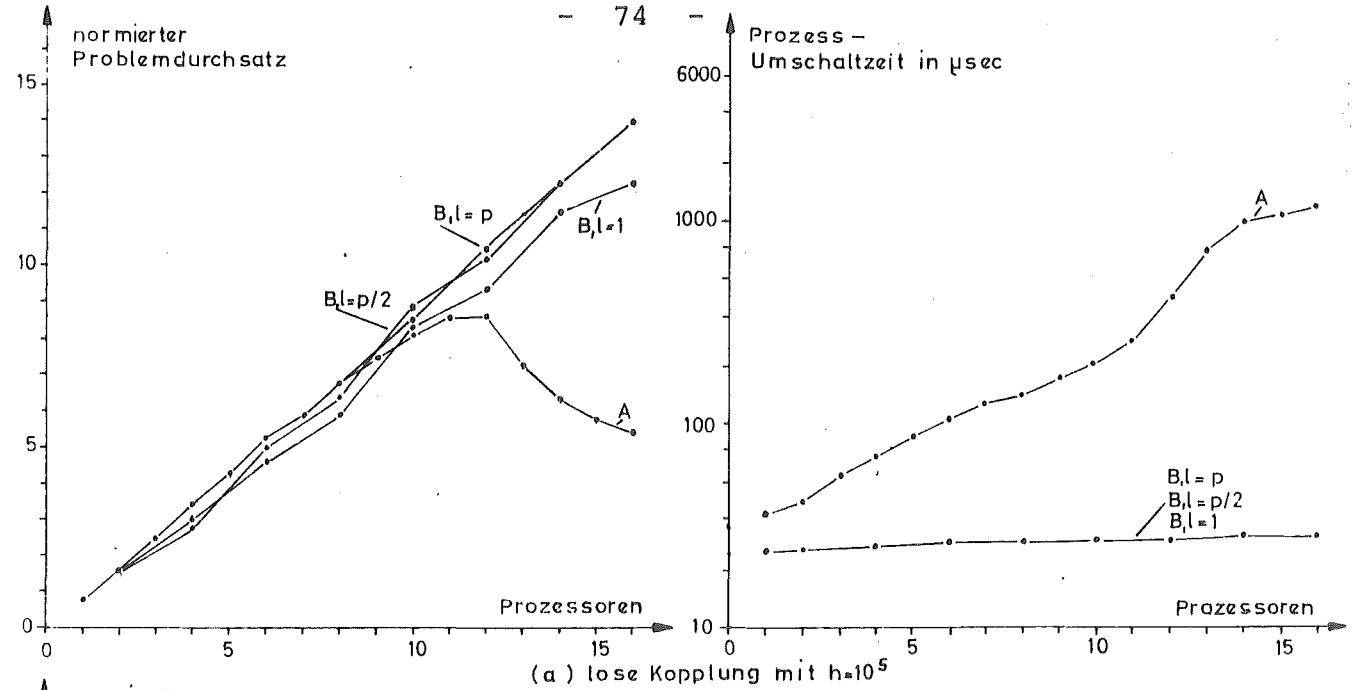
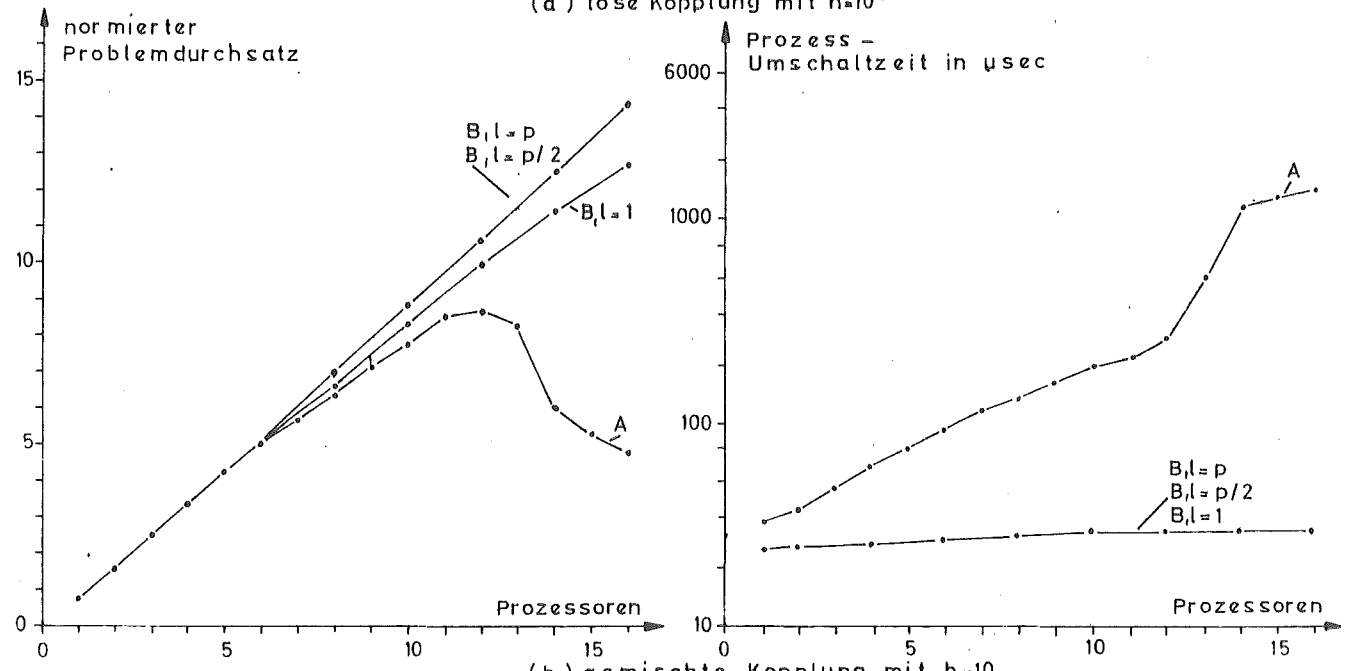


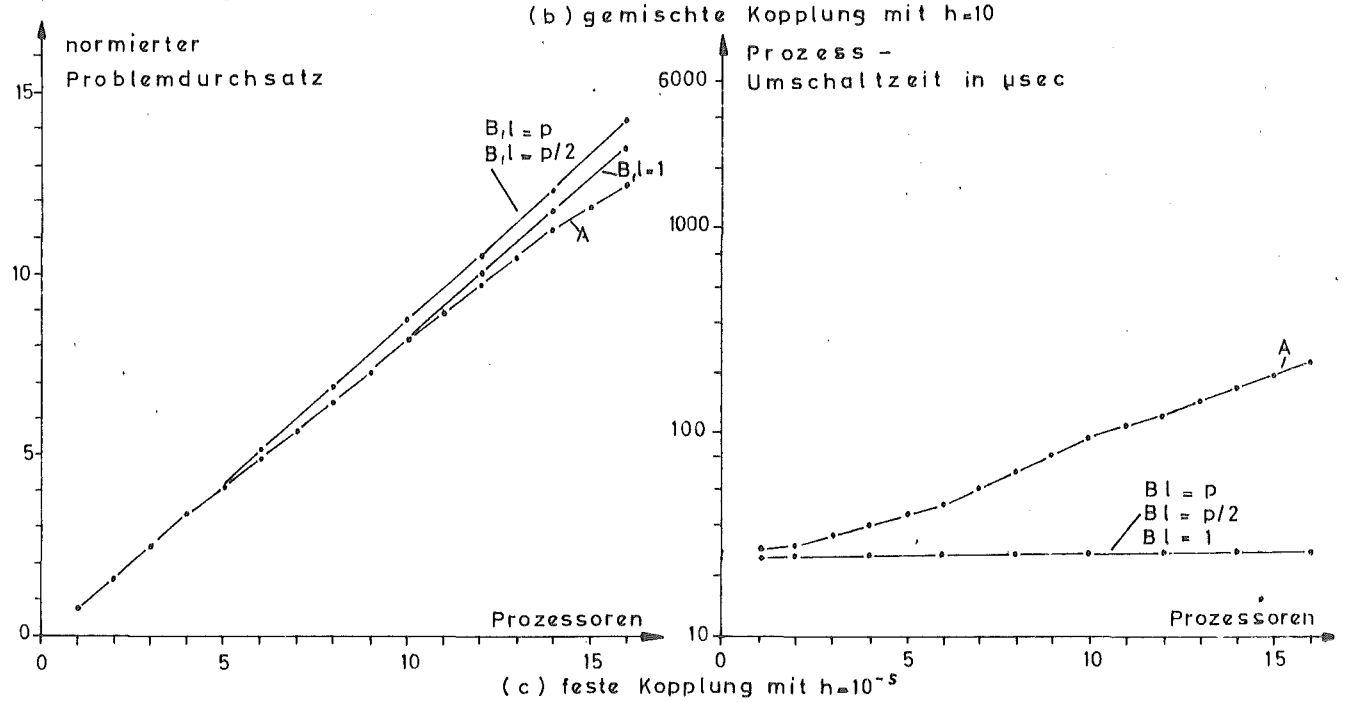
Abb. 16: Gemessene Kurven für die Prozeß-Profilklasse II ($\frac{p}{r}$=16, a=5msec, sw=75msec, lw=75msec)



(a) lose Kopplung mit $h=10^{5}$

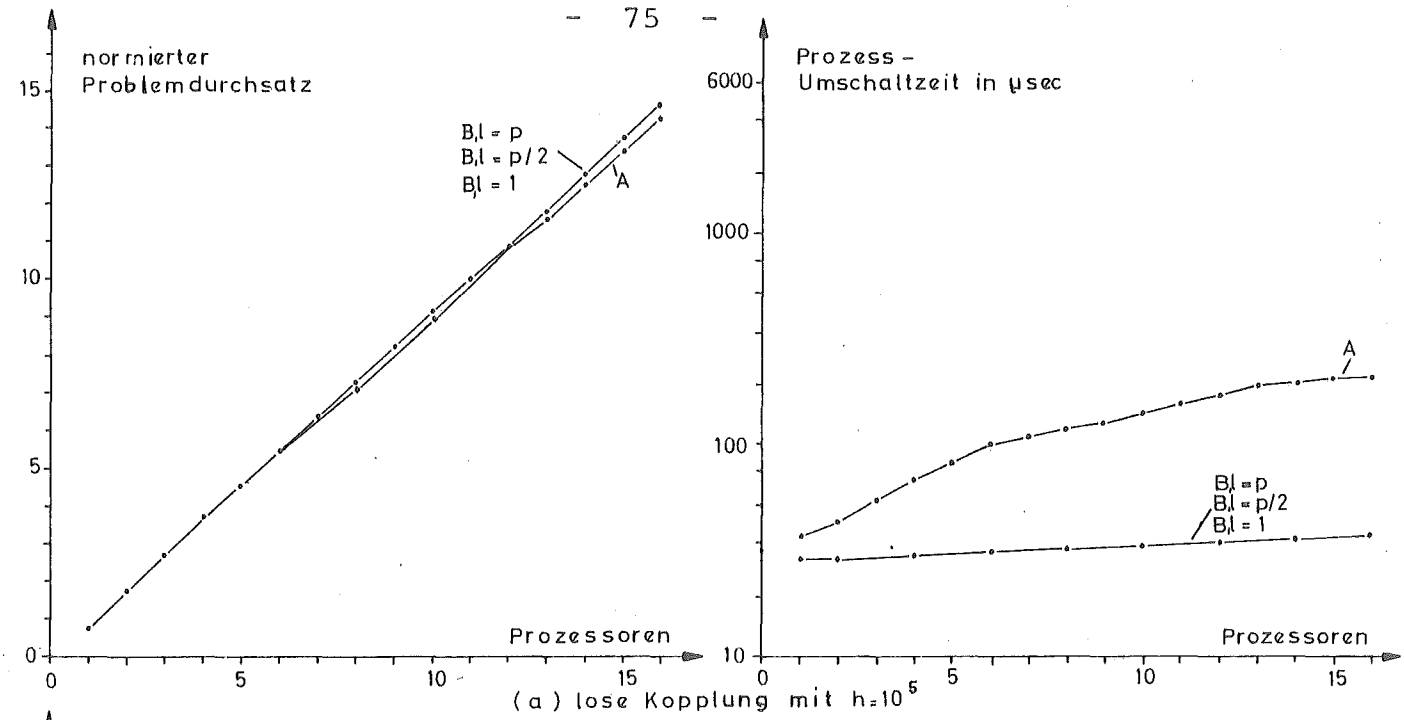


(b) gemischte Kopplung mit $h=10$

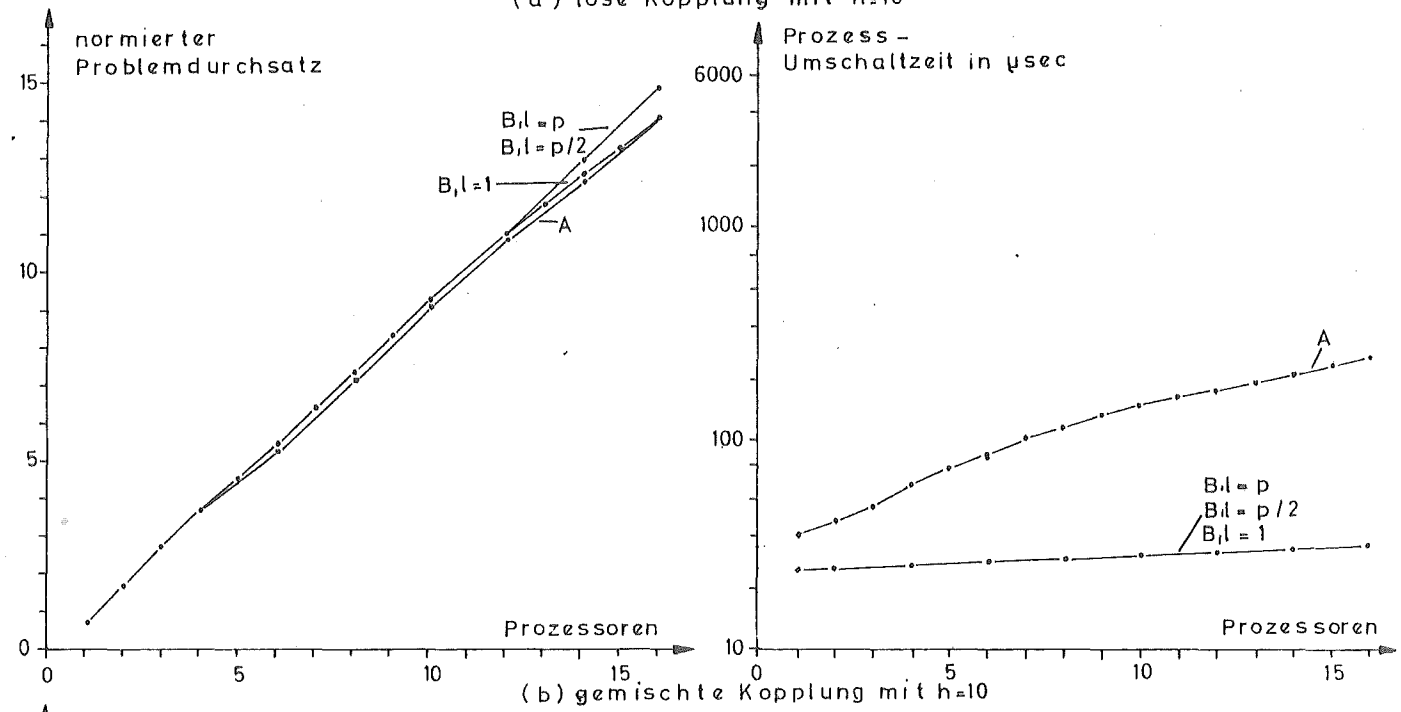


(c) feste Kopplung mit $h=10^{-5}$

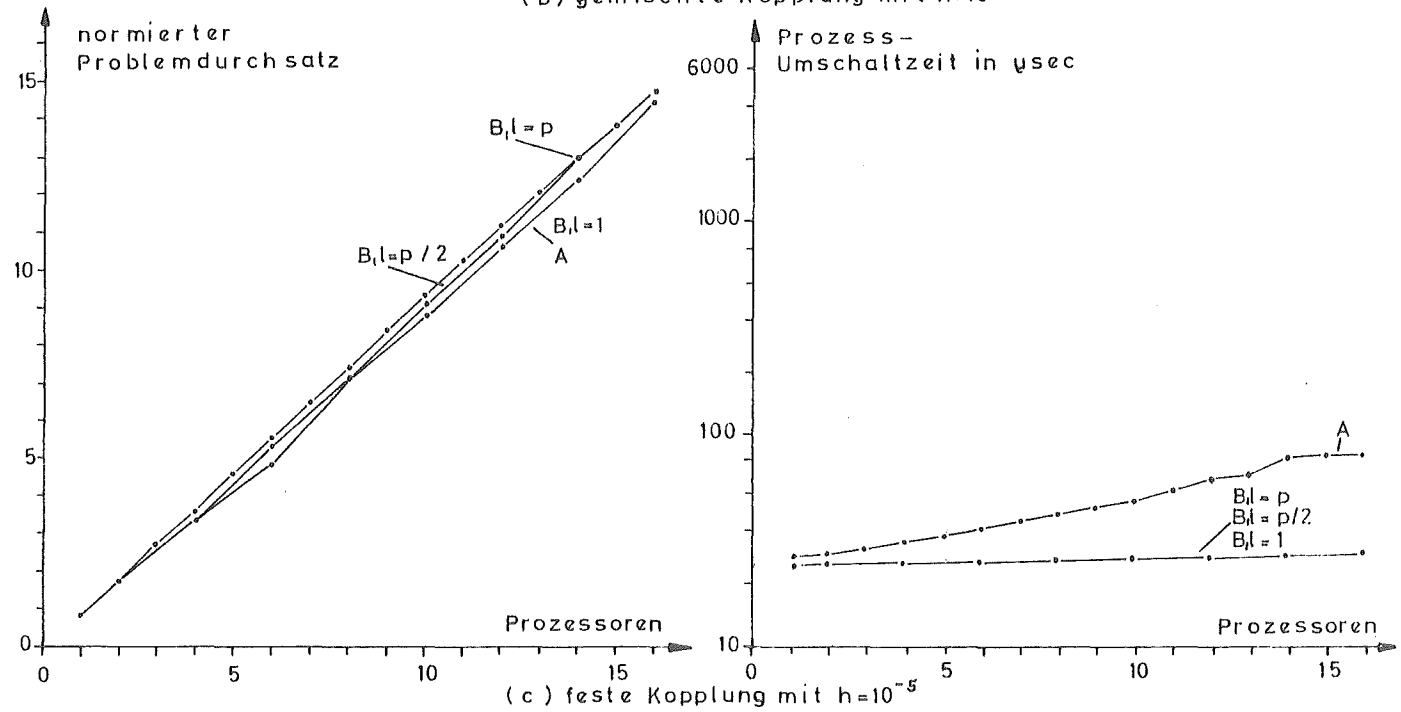
Abb.17: Gemessene Kurven für die Prozeß Profilklasse III ($\frac{p}{r}=4$ a=5msec, sw=15msec, lw=15msec)



(a) lose Kopplung mit $h=10^{5}$



(b) gemischte Kopplung mit $h=10$



(c) feste Kopplung mit $h=10^{-5}$

Abb.18: Gemessene Kurven für die Prozeß-Profilklasse IV ($\frac{p}{r}=4$, a= 25msec, sw=lw=75 msec)

Liste der wichtigsten Symbole und ihre Bedeutung

a	Arbeitsphase eines Benutzerprozesses
b	mittlere Prozessorbelegung
d	normierter Problemdurchsatz
e	Eigenverbrauch des Dispatchers (Overhead)
h	Häufigkeit, gibt an, wieviel Short-Wait-Perioden im Mittel auf eine Long-Wait-Periode kommen
k	Kopplungsgrad zwischen Dispatcher und Scheduler (lose Kopplung : $k = 0$, feste Kopplung : $k = 1$, gemischte Kopplung : $0 < k < 1$)
lw	Long-Wait-Periode eines Benutzerprozesses
p	Zahl der Benutzerprozesse
r	Zahl der Prozessoren (Rechnerkerne)
sw	Short-Wait-Periode eines Benutzerprozesses
T_a	absoluter Problemdurchsatz innerhalb des Meßintervalls
T_{assign}	mittlere Ausführungszeit für die assign-Funktion
T_{disp}	Eigenzeitbedarf des Dispatchers innerhalb des Meßintervalls
T_{idle}	Summe der UNTÄTIG-Zeiten aller Prozessoren innerhalb des Meßintervalls
T_{switch}	Prozess-Umschaltzeit
T_{tot}	simulierte Laufzeit in einem Simulationslauf
T_{wait}	mittlere Ausführungszeit für die wait-Funktion
w_{lw}	Wahrscheinlichkeit für eine Long-Wait-Periode nach einer abgeschlossenen Arbeitsphase eines Benutzerprozesses
w_{sw}	Wahrscheinlichkeit für eine Short-Wait-Periode nach einer abgeschlossenen Arbeitsphase eines Benutzerprozesses

Literaturquellen

/1/ Alexander, M. T.
Time Sharing Supervisor Programs
University of Michigan, Computing Center, May 1969, revised May 1970

/2/ Corbato, F. J.
Introduction and Overview of the MULTICS System
Proceedings-FJCC Vol. 27, Part 1, 1965, 185 - 196

/3/ Dijkstra, E. W.
Cooperating Sequential Processes
Mathematical Dep., Technical University Eindhoven, Sept. 1965

/4/ Dijkstra, E. W.
The Structure of the THE-Multiprogramming System
Comm. ACM Vol. 11, No. 5, May 1968, 341 - 346

/5/ Dijkstra, E. W.
Hierarchical Ordering of Sequential Processes
Acta Informatica 1, 1971, 115 - 138

/6/ Dennis, J. B., VanHorn, E. C.
Programming Semantics for Multiprogrammed Computations
Comm. ACM Vol. 9, No. 3, March 1966, 143 - 155

/7/ Denning, P. J.
Resource Allocation in Multiprocess Computer Systems
Ph. D. Thesis at MIT (MAC - TR - 50), May 1968

/8/ Denning, P. J.
Thrashing :Its Cause and Prevention
Proceedings-FJCC Vol. 33, Part 1, 1968, 915 - 922

/9/ Daley, R. C., Dennis, J. B.
Virtual Memory, Processes and Sharing in MULTICS
Comm. ACM, Vol. 11, No. 5, May 1968, 306 - 312

/10/ Fox, D., Kessler, J. L.
Experiments in Software Modeling
Proceedings-FJCC Vol. 31, 1967, 429 - 436

/11/ Fishman, G. S., Kiviat, P. J.
The Statistics of Discrete Event Simulation
Simulation - April 1968, 185 - 194

/12/ Fisz, M.
Wahrscheinlichkeitsrechnung und mathematische Statistik
Deutscher Verlag der Wissenschaften Berlin, 5. Auflage
1970, S. 235

/13/ Goos, G., Lagally, K., Sapper, G.
PS 440 - Eine niedere Programmiersprache
RZ der TU München, Bericht Nr. 7002, 1970

/14/ Goos, G.
Betriebssysteme - Theorie und Praxis
Nachrichtentechnische Fachberichte Band 44, 1972
VDE-Verlag GmbH Berlin

/15/ Goos, G., Jürgens, J., Lagally, K.
The Operating System BSM, Viewed as a Community
of Parallel Processes
Arbeitsgruppe für Betriebssysteme am RZ der TU München
Mai 1972

/16/ Goos, G.
Ein Strukturmodell für Betriebssysteme
NTG-Fachtagung "Rechnerstrukturen u. Betriebsprogrammierung"
Erlangen, Okt. 1970

/17/ Glaser, E. L., Couleur, J. F., Oliver, G. A.
System Design of a Computer for Time Sharing Applications
Proceedings-FJCC Vol. 27, Part 1, 1965, 197 - 202

/18/ Graham, R. M.
Performance Prediction
Advanced Course on Software Engineering, Feb. 21-March 4,
1972
Technical University of Munich, Germany

/19/ Hansen, P. B.
RC 4000 Software : Multiprogramming System
(Ed.) A/S Regnecentralen, Copenhagen 1969

/20/ Hansen, P. B.
The Nucleus of a Multiprogramming System
Comm. ACM Vol.13, No. 4, April 1970, 238 - 250

/21/ Havender, J. W.
Avoiding Deadlock in Multitasking Systems
IBM System Journal No. 2, 1968, 74 - 84

/22/ Hutchinson, G. K., Maguire, J. N.
Computer Systems Design and Analysis through Simulation
Proceedings-FJCC Vol. 27, Part 1, 1965, 161 - 167

/23/ Kimbleton, S. R.
The Role of Computer System Models in Performance Evaluation
Comm. ACM Vol. 15, No. 7, July 1972, 586 - 590

/24/ Kohlas, J.
Monte Carlo Simulation in Operations Research
Lecture Notes in Economics and Mathematical Systems 63
Springer Verlag Berlin, Heidelberg, New York 1972

/25/ Lampson, B. W.
A Scheduling Philosophy for Multiprocessing Systems
Comm. ACM, Vol. 13, No. 1, Jan. 1970, 347 - 360

/26/ Liskov, B. H.
The Design of the VENUS Operating System
Comm. ACM Vol. 15, No. 3, March 1972, 144 - 149

/27/ LTPL-Working Group
Functional Requirements and Language Features for a Procedural Language for Industrial Computers
LTPL-Report Oct. 29, 1971

/28/ Meyer, R. A., Seawright, L. H.
A Virtual Machine Time-Sharing System
IBM System Journal No. 3, 1970, 199 - 218

/29/ Mosier, R. A.
A Multiprocessor Compendium
Institute for Defense Analyses, Science and Technology Division
Research Paper P-433, Virginia June 1968

/30/ Merikallio, R. A., Holland, F. C.
Simulation Design of a Multiprocessing System
Proceedings-FJCC Vol. 33, Part 2, 1968, 1399 - 1410

/31/ MacDougall, M. H.
Computer System Simulation : An Introduction
Computing Surveys Vol. 2, No. 3, Sept. 1970, 191 - 209

/32/ Mußtopf, G.
Das Programmiersystem POLYP
Angewandte Informatik Heft 10, 1972, 441 - 448

/33/ Nehmer, J., Fleck, G.
Simulation von Betriebssoftware auf einer virtuellen PL/1-Maschine
Lecture Notes in Economics and Mathematical Systems 78
Springer Verlag Berlin, Heidelberg, New York 1972, 253-262

/34/ Nehmer, J., Fleck, G., Hilse, D., Rupp, M., Friehmelt, R.
PLIM - Eine virtuelle PL/1-Maschine
Kernforschungszentrum Karlsruhe
KFK-Bericht 1712, März 1973

/35/ Nehmer, J., Hilse, D., Rupp, M.
Ein PL/1-Unterprogrammpaket für die diskrete Eventsimulation
Kernforschungszentrum Karlsruhe
KFK-Bericht 1711, Januar 1973

/36/ Parnas, D. L.
On the Criteria to be Used in Decomposing Systems into Modules
Comm. ACM Vol. 15, No. 2, Dec. 1972, 1053 - 1058

/37/ Pyle, I. C.
Some Techniques in Multi Computer System Software Design
Software Practice and Experience Vol. 2, 1972, 43 - 54

/38/ PDV/IDT-Projektgruppe
Hardwarenahe Elementarfunktionen der Ablaufsteuerung für Prozessrechenbetriebssysteme
Kernforschungszentrum Karlsuhe
PDV - P5.2-KA-IDT/4, Okt. 1972

/39/ Saltzer, J. H.
Traffic Control in a Multiplexed Computer System
Ph. D. Thesis at MIT (MACTR-30), July 1966

/40/ System/A - Working Group
SYSTEM/A
IBM-Technical Report 2, Vol. 1, RA 24 (15670),
July 15, 1971

/41/ Taylor, J. R.
Multiprocessor Systems for Reliability - A Comparative Study
Atomic Energy Research Establishment, AERE-R7102,
Harwell, May 1972

/42/ Waite, W. M.
The Mobile Programming System : STAGE2
Comm. ACM Vol. 13, No. 7, 1970, 415 - 421

/43/ Wettstein, H.
Ablaufsteuerung in Betriebssystemen
2. Kapitel aus dem Skriptum "Betriebssystemprogrammierung",
Karlsruhe, Jan. 1973

/44/ Wulf, W. A., Russell, D. B., Habermann, A. N.
BLISS : A Language for Systems Programming
Comm. ACM Vol. 14, No. 12, 1971, 780 - 790

/45/ Wigan, M. R.
The Fitting, Calibration and Validation of Simulation Models
Simulation - May 1972, 188 - 192

Anhang A: PL/1-Programme der Dispatcher-Entwürfe

PL/1-Programm für den Dispatcher-Entwurf A

Prozess-Profilklasse	empfohlene Konfigurations-bandbreite
Klasse I: a = 1 msec, sw = lw = 15 msec	1 - 2 Prozessoren
Klasse II: a = 5 msec, sw = lw = 75 msec	1 - 6 Prozessoren
Klasse III: a = 5 msec, sw = lw = 15 msec	1 - 10 Prozessoren
Klasse IV: a = 25 msec, sw = lw = 75 msec	1 - 16 Prozessoren

```
MEMBER NAME  DISP1
 DISP_1:PROCEDURE(PR_NUMB) RECURSIVE; /*MULTIPROCESSOR VERSION*/
 /*INITIALISATION OF THE DISPATCHERS DATA BASE*/
 DCL NULL BUILTIN;
 DCL 1 PROLIST EXTERNAL ALIGNED, /*PROCESS LIST HEADER*/
       2 FW_P POINTER, /*FORWARD POINTER*/
       2 BW_P POINTER, /*BACKWARD POINTER*/
       2 LOCK CHAR(1); /*LOCK BYTE*/
 DCL 1 PR_TAB(PR_NUMB) CONTROLLED EXTERNAL ALIGNED, /*PROCESSOR TABLE*/
       2 MODE CHAR(4), /*'IDLE' OR 'WAKE' OR 'WORK'*/
       2 PRIORITY BIN FIXED(15), /*PROCESSOR PRIORITY*/
       2 REQUEST BIN FIXED(15), /*0=NO REQUEST,1=SIGNAL REQUEST*/
       2 ALLOC_P POINTER, /*POINTER TO ALLOCATED PROCESS*/
       2 LOCK  CHAR(1); /*LOCK BYTE*/
 DCL PR_NUMB BIN FIXED(15); /*NUMBER OF PROCESSORS*/
 DCL 1 PCB BASED(PCB_P) ALIGNED, /*PROCESS CONTROL BLOCK*/
       2 PROCESS_ID BIN FIXED(31), /*PROCESS IDENTIFICATION*/
       2 PRIORITY BIN FIXED(15),/*PROCESS PRIORITY*/
       2 PROC_STATE BIT(8), /*PROCESS STATE*/
       2 REG_STATE(10) BIN FIXED(31); /*SAVE AREA FOR PROCESSOR*/
                  /*REGISTERS AND OTHER INFORMATION*/
 DCL 1 CONNECTOR BASED(CONNECT_P) ALIGNED, /*FOR COUPLING PCB'S*/
       2 FW_P POINTER, /*CHAIN FORWARD POINTER*/
       2 BW_P POINTER, /*CHAIN BACKWARD POINTER*/
       2 DISP_STATE CHAR(4), /*'READ','ACTV','WAIT'*/
       2 TEXT_P POINTER, /*POINTER TO PCB*/
       2 PR_ID BIN FIXED(15), /*PROCESSOR IDENTIFICATION*/
       2 RET_REQ BIN FIXED(15); /*RETIRE REQUEST,0=NO,1=YES*/
 DCL (PRE_P,NEXT_P) POINTER; /*ORGANIZATIONAL PARAMETERS*/
 DCL (PCB_P,CONNECT_P) POINTER; /*BASED POINTER*/
 DCL $IDLE ENTRY; /*HARDWARE INSTRUCTION,IDLES A PROCESSOR*/
 DCL $SIGNAL ENTRY(BIN FIXED(15)); /*HARDWARE INSTRUCTION FOR INTER-*/
                                   /*PROCESSOR COMMUNICATION*/
 DCL $WAKEUP ENTRY(BIN FIXED(15)); /*HARDWARE INSTRUCTION FOR INTER-*/
                                   /*PROCESSOR COMMUNICATION*/
 DCL $LOCK ENTRY(CHAR(1)); /*HARDWARE INSTRUCTION TO LOCK A BYTE*/
 DCL $UNLOCK ENTRY(CHAR(1)); /*HARDWARE INSTRUCTION TO UNLOCK A BYTE*/
 DCL $CPU_ID RETURNS(BIN FIXED(15)); /*HARDWARE REGISTER CONTAINING*/
                                     /*THE PROCESSOR IDENTIFICATION*/
 DCL RT_CODE BIN FIXED(15) INITIAL(0); /*RETURN CODE*/
 DCL PROCESS_P POINTER; /*PARAMETER,POINTER TO PCB*/
 DCL PROCSTAT_P POINTER; /*PARAMETER,POINTER TO REGISTER STATE*/
 DCL POLICY CHAR(1); /*PARAMETER,'P'=PREEMPTIVE,'N'=NON PREEMPTIVE*/
 DCL PROCID BIN FIXED(31); /*PARAMETER,PROCESS IDENTIFICATION*/
 DCL CONDITION BIT(8); /*RELEASE CONDITION*/
 DCL I BIN FIXED(15); /*COUNTER*/
 ALLOCATE PR_TAB(*);
 CALL $UNLOCK(PROLIST.LOCK);
 PROLIST.FW_P=NULL;
 PROLIST.BW_P=NULL;
 DO I=1  TO PR_NUMB;
 CALL $UNLOCK(PR_TAB.LOCK(I));
 PR_TAB.MODE(I)='IDLE';
 PR_TAB.ALLOC_P(I)=NULL;
 PR_TAB.REQUEST=0;
 END;
 RETURN;

 ADD:ENTRY(PROCESS_P,POLICY) RETURNS(BIN FIXED(15));
```

```
MEMBER NAME  DISP1
 /*ADDS A NEW PROCESS TO THE DISPATCHER.THE STRATEGY MAY BE EITHER*/
 /*PREEMPTIVE OR NON PREEMPTIVE*/
 /*RETURN CODE VALUES,0=OK,1=NO SPACE*/
 DCL TEMPRIO BIN FIXED(15); /*TEMPORARY VARIABLE*/
 ON AREA BEGIN;
    RT_CODE=1;
    RETURN(RT_CODE);
         END;
 ALLOCATE CONNECTOR;
 PCB_P,CONNECTOR.TEXT_P=PROCESS_P;
 DISP_STATE='READ';
 CONNECTOR.RET_REQ=0;
 PCB.PROC_STATE='00000000'B; /*NO WAITING CONDITION*/
 TEMPRIO=PCB.PRIORITY;
 PRE_P=ADDR(PROLIST);
 CALL $LOCK(PROLIST.LOCK);
 LOOP1:;
 NEXT_P=PRE_P->CONNECTOR.FW_P;
 IF NEXT_P=NULL THEN DO;
     PRE_P->CONNECTOR.FW_P=CONNECT_P;
    CONNECTOR.FW_P=NULL;
    CONNECTOR.BW_P=PRE_P;
                      END;
                ELSE DO;
    PCB_P=NEXT_P->CONNECTOR.TEXT_P;
    IF TEMPRIO<PCB.PRIORITY THEN DO;
       CONNECTOR.FW_P=NEXT_P;
       CONNECTOR.BW_P=PRE_P;
       PRE_P->CONNECTOR.FW_P=CONNECT_P;
    NEXT_P->CONNECTOR.BW_P=CONNECT_P;
                                    END;
                               ELSE DO;
       PRE_P=NEXT_P;
       GOTO LOOP1;
                                    END;
                      END;
 CALL $UNLOCK(PROLIST.LOCK);
 IF POLICY='P' THEN DO; /* PREEMPTIVE ADD*/
                CALL SIGNAL_PR(PROCESS_P->PCB.PRIORITY);
                    END;
                ELSE DO; /*NON PREEMPTIVE ADD*/
    CALL WAKE_PR;
                    END;
 RETURN(RT_CODE);

 RETIRE:ENTRY(PROCID,PROCESS_P,POLICY) RETURNS(BIN FIXED(15));
 /*INITIALIZES RETIRING OF A PROCESS FROM DISPATCHER.IF RETIRING MAY*/
 /*BE PERFORMED IMMEDIATLY,THE PCB-POINTER IS RETURNED TO THE CALLING*/
 /*PROCEDURE*/
 /*RETURN CODE VALUES,0=OK,1=INITIALIZED,2=PROCESS NOT FOUND*/
 CALL $LOCK(PROLIST.LOCK);
 CONNECT_P=PROLIST.FW_P;
 LOOP2:;
 IF CONNECT_P¬=NULL THEN DO;
    PCB_P=CONNECTOR.TEXT_P;
    IF PCB.PROCESS_ID=PROCID THEN DO;
       IF CONNECTOR.DISP_STATE='ACTV' THEN DO; /*PROCESS IS ACTIV*/
          CONNECTOR.RET_REQ=1;
```

```
MEMBER NAME  DISP1
          CALL $UNLOCK(PROLIST.LOCK);
          IF POLICY='P' THEN DO;
             CALL $LOCK(PR_TAB.LOCK(CONNECTOR.PR_ID));
             IF (PR_TAB.MODE='ACTV')&(PR_TAB.REQUEST=0) THEN DO;
             CALL $SIGNAL(CONNECTOR.PR_ID);
                                END;
                             ELSE;
          CALL $UNLOCK(PR_TAB.LOCK(CONNECTOR.PR_ID));
       RT_CODE=1; /*RETIRING INITIALIZED*/
                                END;
                             ELSE DO; /*PROCESS IS READY OR WAITING*/
       PRE_P=CONNECTOR.BW_P;
       NEXT_P=CONNECTOR.FW_P;
       PRE_P->CONNECTOR.FW_P=NEXT_P;
       IF NEXT_P¬=NULL THEN NEXT_P->CONNECTOR.BW_P=PRE_P;
       CALL $UNLOCK(PROLIST.LOCK);
       FREE CONNECTOR;
       PROCESS_P=PCB_P;
                                END;
                                END;
                             ELSE DO; /*PROCESS NOT FOUND YET*/
       CONNECT_P=CONNECTOR.FW_P;
       GOTO LOOP2;
                                END;
                          END;
                       ELSE DO;
    RT_CODE=2; /* PROCESS NOT IN LIST*/
    CALL $UNLOCK(PROLIST.LOCK);
                          END;
RETURN(RT_CODE);

ASSIGN:ENTRY(PROCSTAT_P,PROCID) RETURNS(BIN FIXED(15));
/*ASSIGNS A READY PROCESS TO THE INVOKING PROCESSOR AND PASSES THE*/
/*POINTER TO THE REGISTER STATE AND PROCESS IDENTIFICATION BACK TO*/
 /*THE CALLING PROCEDURE*/
/*RETURN CODE VALUES,0=OK,1=NO WORK,2=PROCESSOR ALREADY*/
/*ALLOCATED*/
IF PR_TAB.MODE($CPU_ID)¬='WORK' THEN DO;
   CALL $LOCK(PROLIST.LOCK);
   CONNECT_P=PROLIST.FW_P;
LOOP3:;
   IF CONNECT_P=NULL THEN DO;
      CALL $UNLOCK(PROLIST.LOCK);
      RT_CODE=1;
                          END;
                       ELSE DO;
      IF DISP_STATE='READ' THEN DO;
         PCB_P=CONNECTOR.TEXT_P;
         PROCSTAT_P=ADDR(PCB.REG_STATE);
         PROCID=PCB.PROCESS_ID;
         DISP_STATE='ACTV';
         CONNECTOR.PR_ID=$CPU_ID;
         CALL $UNLOCK(PROLIST.LOCK);
         CALL $LOCK(PR_TAB.LOCK($CPU_ID));
         PR_TAB.ALLOC_P($CPU_ID)=CONNECT_P;
         PR_TAB.MODE($CPU_ID)='WORK';
         PR_TAB.PRIORITY($CPU_ID)=PCB.PRIORITY;
         PR_TAB.REQUEST($CPU_ID)=0;
```

```
MEMBER NAME  DISP1
          CALL $UNLOCK(PR_TAB.LOCK($CPU_ID));
                              END;
                         ELSE DO;
          CONNECT_P=CONNECTOR.FW_P;
          GOTO LOOP3;
                              END;
                              END;
                                    END;
                               ELSE DO;
   RT_CODE=2;
                                    END;
RETURN(RT_CODE);

RELEASE:ENTRY(CONDITION,PROCESS_P) RETURNS(BIN FIXED(15));
/*RELEASES THE PROCESSOR FROM ALLOCATED PROCESS AND PASSES THE PCB-*/
/*POINTER BACK TO THE CALLING PROCEDURE*/
/*RETURN CODE VALUES,0=OK,1=PROCESSOR NOT ALLOCATED*/
IF PR_TAB.MODE($CPU_ID)='WORK' THEN DO;
   CALL $LOCK(PR_TAB.LOCK($CPU_ID));
   PR_TAB.MODE($CPU_ID)='WAKE';
   CALL $UNLOCK(PR_TAB.LOCK($CPU_ID));
   CONNECT_P=PR_TAB.ALLOC_P($CPU_ID);
   CALL $LOCK(PROLIST.LOCK);
   PRE_P=CONNECTOR.BW_P;
   NEXT_P=CONNECTOR.FW_P;
   PRE_P->CONNECTOR.FW_P=NEXT_P;
   IF NEXT_P¬=NULL THEN NEXT_P->CONNECTOR.BW_P=PRE_P;
   CALL $UNLOCK(PROLIST.LOCK);
   PCB_P=CONNECTOR.TEXT_P;
   PROCESS_P=PCB_P;
   PCB.PROC_STATE=CONDITION|PCB.PROC_STATE;
                                    END;
                               ELSE DO;
   RT_CODE=1;
                                    END;
RETURN(RT_CODE);

WAIT:ENTRY(CONDITION) RETURNS(BIN FIXED(15));
/*PUTS AN ACTIVE PROCESS INTO THE WAITING STATE*/
/*RETURN CODE VALUES,0=OK,1=RETIRE REQUEST,2=PROCESSOR*/
/*NOT ALLOCATED*/
IF PR_TAB.MODE($CPU_ID)='WORK' THEN DO;
   CALL $LOCK(PR_TAB.LOCK($CPU_ID));
   PR_TAB.MODE($CPU_ID)='WAKE';
   CALL $UNLOCK(PR_TAB.LOCK($CPU_ID));
   CONNECT_P=PR_TAB.ALLOC_P($CPU_ID);
   CALL $LOCK(PROLIST.LOCK);
   PCB_P=CONNECTOR.TEXT_P;
   PCB.PROC_STATE=CONDITION;
   IF CONNECTOR.RET_REQ=1 THEN DO;
      CALL $UNLOCK(PROLIST.LOCK);
      CALL $LOCK(PR_TAB.LOCK($CPU_ID));
      PR_TAB.MODE($CPU_ID)='WORK';
      CALL $UNLOCK(PR_TAB.LOCK($CPU_ID));
      RT_CODE=1;
                              END;
                         ELSE DO;
      CONNECTOR.DISP_STATE='WAIT';
```

```
MEMBER NAME  DISP1
       CALL $UNLOCK(PROLIST.LOCK);
                                   END;
                                       END;
                                   ELSE DO;
    RT_CODE=2;
                                        END;
 RETURN(RT_CODE);

 READY:ENTRY(PROCID,CONDITION) RETURNS(BIN FIXED(15));
 /*THE NAMED PROCESS WILL BE ACTIVATED IF ALL WAITING CONDITIONS*/
 /*CAN BE REMOVED*/
 /*RETURN CODE VALUES,0=OK,1=PROCESS NOT FOUND*/
 CALL $LOCK(PROLIST.LOCK);
 CONNECT_P=PROLIST.FW_P;
 LOOP4:;
    IF CONNECT_P¬=NULL THEN DO;
    PCB_P=CONNECTOR.TEXT_P;
    IF PCB.PROCESS_ID=PROCID THEN DO;
       IF DISP_STATE='WAIT' THEN DO;
          PCB.PROC_STATE=(¬CONDITION)&(PCB.PROC_STATE);
          IF PCB.PROC_STATE='00000000'B THEN DO;
             DISP_STATE='READ';
             CALL $UNLOCK(PROLIST.LOCK);
             CALL WAKE_PR; /*WAKEUP IDLE PROCESSOR*/
                                          END;
                                    ELSE CALL $UNLOCK(PROLIST.LOCK);
                                END;
                             ELSE CALL $UNLOCK(PROLIST.LOCK);
                                 END;
                              ELSE DO;
       CONNECT_P=CONNECTOR.FW_P;
       GOTO LOOP4;
                                   END;
                          END;
                       ELSE DO;
    CALL $UNLOCK(PROLIST.LOCK);
    RT_CODE=1; /*PROCESS NOT FOUND*/
                          END;
 RETURN(RT_CODE);

 REREADY:ENTRY RETURNS(BIN FIXED(15));
 /*PUTS AN ACTIVE PROCESS INTO THE READY STATE*/
 /*RETURN CODE VALUES,0=OK,1=RETIRE REQEST,2=PROCESSOR*/
 /*NOT ALLOCATED*/
 IF PR_TAB.MODE($CPU_ID)='WORK' THEN DO;
    CALL $LOCK(PR_TAB.LOCK($CPU_ID));
    PR_TAB.MODE($CPU_ID)='WAKE';
    CALL $UNLOCK(PR_TAB.LOCK($CPU_ID));
    CONNECT_P=PR_TAB.ALLOC_P($CPU_ID);
    CALL $LOCK(PROLIST.LOCK);
    IF CONNECTOR.RET_REQ=1 THEN DO; /*RETIRE REQUEST*/
       CALL $UNLOCK(PROLIST.LOCK);
       CALL $LOCK(PR_TAB.LOCK($CPU_ID));
       PR_TAB.MODE($CPU_ID)='WORK';
       CALL $UNLOCK(PR_TAB.LOCK($CPU_ID));
       RT_CODE=1; /*RETIRE REQUEST*/
                                   END;
                              ELSE DO;
```

```
MEMBER NAME  DISP1
    CONNECTOR.DISP_STATE='READ';
    CALL $UNLOCK(PROLIST.LOCK);
                                    END;
                                        END;
                                    ELSE DO;
    RT_CODE=2; /*PROCESSOR NOT ALLOCATED*/
                                        END;
 RETURN(RT_CODE);

 STOP:ENTRY RETURNS(BIN FIXED(15));
 /*PUTS THE INVOKING PROCESSOR INTO THE IDLE STATE*/
 /*RETURN CODE VALUES,0=OK,1=PROCESSOR STILL ALLOCATED*/
 CALL $LOCK(PR_TAB.LOCK($CPU_ID));
 IF PR_TAB.MODE($CPU_ID)='WORK' THEN DO;
    CALL $UNLOCK(PR_TAB.LOCK($CPU_ID));
    RT_CODE=1;
                                        END;
                                    ELSE DO;
    PR_TAB.MODE($CPU_ID)='IDLE';
    CALL $UNLOCK(PR_TAB.LOCK($CPU_ID));
    CALL $IDLE;                         END;
 RETURN(RT_CODE);

 WAKE_PR:ENTRY; /*FIND AND WAKEUP ANY IDLE PROCESSOR*/
 DCL PR_INDEX BIN FIXED(15); /*PROCESSOR INDEX*/
 PR_INDEX=0;
 LOOP5:;
 PR_INDEX=PR_INDEX+1;
 CALL $LOCK(PR_TAB.LOCK(PR_INDEX));
 IF (PR_TAB.MODE(PR_INDEX)='IDLE')&(PR_INDEX¬=$CPU_ID) THEN DO;
    CALL $WAKEUP(PR_INDEX);
    PR_TAB.MODE(PR_INDEX)='WAKE' ;
    CALL $UNLOCK(PR_TAB.LOCK(PR_INDEX));
    IF PR_INDEX<PR_NUMB THEN GOTO LOOP5;
                        ELSE;
                                                        END;
 SIGNAL_PR:ENTRY(PROC_PRIO);
 /*SEEKS A PROCESSOR WITH A LOWER PRIORITY THAN PROC_PRIO AND*/
 /*INTERRUPTS IT BY SIGNALING*/
 DCL (PROC_PRIO,PR_VAR1,PR_VAR2) BIN FIXED(15);
 PR_VAR1=0;
 LOOP6:;
 PR_VAR1=PR_VAR1+1;
 IF PR_VAR1=$CPU_ID THEN GOTO LOOP6;
 IF PR_VAR1>PR_NUMB THEN RETURN;
 CALL $LOCK(PR_TAB.LOCK(PR_VAR1));
 IF PR_TAB.MODE(PR_VAR1)='IDLE' THEN DO;
    CALL $WAKEUP(PR_VAR1);
    PR_TAB.MODE(PR_VAR1)='WAKE';
    CALL $UNLOCK(PR_TAB.LOCK(PR_VAR1));
    RETURN;
                                        END;
 IF PR_TAB.MODE(PR_VAR1)¬='WORK' THEN DO;
    CALL $UNLOCK(PR_TAB.LOCK(PR_VAR1));
    GOTO LOOP6;
                                        END;
 IF (PR_TAB.PRIORITY(PR_VAR1)<=PROC_PRIO)|
    (PR_TAB.REQUEST(PR_VAR1)=1)  THEN DO;
```

```
MEMBER NAME  DISP1
        CALL $UNLOCK(PR_TAB.LOCK(PR_VAR1));
        GOTO LOOP6;
                                          END;
 PR_VAR2=PR_VAR1;
 LOOP7:;
 PR_VAR2=PR_VAR1+1;
 IF PR_VAR2=$CPU_ID THEN GOTO LOOP7;
 IF PR_VAR2>PR_NUMB THEN DO;
    CALL $SIGNAL(PR_VAR1);
    PR_TAB.REQUEST(PR_VAR1)=1; /*SIGNAL REQUEST*/
    CALL $UNLOCK(PR_TAB.LOCK(PR_VAR1));
                          END;
 CALL $LOCK(PR_TAB.LOCK(PR_VAR2));
 IF PR_TAB.MODE(PR_VAR2)='IDLE' THEN DO;
    CALL $WAKEUP(PR_VAR2);
    PR_TAB.MODE(PR_VAR2)='WAKE';
    CALL $UNLOCK(PR_TAB.LOCK(PR_VAR1));
    CALL $UNLOCK(PR_TAB.LOCK(PR_VAR2));
    RETURN;
                                          END;
 IF PR_TAB.MODE(PR_VAR2)¬='WORK' THEN DO;
    CALL $UNLOCK(PR_TAB.LOCK(PR_VAR2));
    GOTO LOOP7;
                                          END;
 IF (PR_TAB.PRIORITY(PR_VAR2)<=PR_TAB.PRIORITY(PR_VAR1))|
    (PR_TAB.REQUEST(PR_VAR2)=1) THEN DO;
    CALL $UNLOCK(PR_TAB.LOCK(PR_VAR2));
    GOTO LOOP7;
                                          END;
 CALL $UNLOCK(PR_TAB.LOCK(PR_VAR1));
 PR_VAR1=PR_VAR2;
 GOTO LOOP7;
 END DISP_1;
```

PL/1-Programm für den Dispatcher-Entwurf B

Angaben gelten für (1 = p, 1 = p/2)

Prozess-Profilklasse	empfohlene Konfigurations-bandbreite
Klasse I: a = 1 msec, sw = lw = 15 msec	1 - 14 Prozessoren
Klasse II: a = 5 msec, sw = lw = 75 msec	1 - 16 Prozessoren
Klasse III: a = 5 msec, sw = lw = 15 msec	1 - 16 Prozessoren
Klasse IV: a = 25 msec, sw = lw = 75 msec	1 - 16 Prozessoren

```
MEMBER NAME  DISP2
 DISP_2:PROCEDURE(PR_NUMB,T_LENGTH) RECURSIVE;
 /*INITIALISATION OF THE DISPATCHERS DATA BASE*/

 DCL NULL BUILTIN;

 DCL 1 PROTAB(T_LENGTH) CONTROLLED EXTERNAL ALIGNED,
     /*PROCESS HASH TABLE*/
       2 CHAIN_P POINTER, /*CHAINING OF PCB'S IN THE PROCESS TABLE*/
       2 LOCK CHAR(1); /*LOCK BYTE*/
 DCL T_LENGTH BIN FIXED(15); /*LENGTH OF PROCESS HASH TABLE*/

 DCL 1 PR_TAB(PR_NUMB) CONTROLLED EXTERNAL ALIGNED, /*PROCESSOR TABLE*/
       2 MODE CHAR(4), /*'IDLE' OR 'WAKE' OR 'WORK'*/
       2 PRIORITY BIN FIXED(15), /*PROCESSOR PRIORITY*/
       2 REQUEST BIN FIXED(15), /*0=NO REQUEST,1=SIGNAL REQUEST*/
       2 ALLOC_P POINTER, /*POINTER TO ALLOCATED PROCESS*/
       2 LOCK CHAR(1); /*LOCK BYTE*/
 DCL PR_NUMB BIN FIXED(15); /*NUMBER OF PROCESSORS*/

 DCL 1 PCB BASED(PCB_P) ALIGNED, /*PROCESS CONTROL BLOCK*/
       2 PROCESS_ID BIN FIXED(31), /*PROCESS IDENTIFICATION*/
       2 PRIORITY BIN FIXED(15), /*PROCESS PRIORITY*/
       2 PROC_STATE BIT(8), /*PROCESS STATE*/
       2 REG_STATE(10) BIN FIXED(31); /*SAVE AREA FOR PROCESSOR*/
                                /*REGISTERS AND OTHER INFORMATION*/

 DCL 1 CONNECTOR BASED(CONNECT_P) ALIGNED, /*FOR COUPLING PCB'S*/
       2 CHAIN_P POINTER, /*CHAINING IN PROCESS TABLE*/
       2 FW_P POINTER, /*FORWARD CHAINING IN READY-LIST*/
       2 BW_P POINTER, /*BACKWARD CHAINING IN READY-LIST*/
       2 DISP_STATE CHAR(4), /*'READ','ACTV','WAIT','UNDF'*/
       2 TEXT_P POINTER, /*POINTER TO PCB*/
       2 PR_ID BIN FIXED(15), /*PROCESSOR IDENTIFICATION*/
       2 RET_REQ BIN FIXED(15), /*RETIRE REQUEST,0=NO,1=YES*/
       2 LOCK CHAR(1); /*LOCK BYTE*/

 DCL 1 R_LIST EXTERNAL ALIGNED, /*HEADER OF READY-LIST*/
       2 DUMMY_P POINTER, /*NOT USED*/
       2 FW_P POINTER, /*FORWARD CHAINING*/
       2 BW_P POINTER, /*BACKWARD CHAINING*/
       2 LOCK CHAR(1); /*LOCK BYTE*/

 DCL (PCB_P,CONNECT_P) POINTER; /*BASED POINTER*/
 DCL (PRE_P,NEXT_P) POINTER; /*ORGANIZATIONAL PARAMETERS*/
 DCL H_INDEX BIN FIXED(15); /*HASH INDEX*/
 DCL $IDLE ENTRY; /*HARDWARE INSTRUCTION,IDLES A PROCESSOR*/
 DCL $SIGNAL ENTRY(BIN FIXED(15)); /*HARDWARE INSTRUCTION FOR INTER-*/
                                   /*PROCESSOR COMMUNICATION*/
 DCL $WAKEUP ENTRY(BIN FIXED(15)); /*HARDWARE INSTRUCTION FOR INTER-*/
                                   /*PROCESSOR COMMUNICATION*/
 DCL $LOCK ENTRY(CHAR(1)); /*HARDWARE INSTRUCTION TO LOCK A BYTE*/
 DCL $UNLOCK ENTRY(CHAR(1)); /*HARDWARE INSTRUCTION TO UNLOCK A BYTE*/
 DCL $CPU_ID RETURNS(BIN FIXED(15)); /*HARDWARE REGISTER CONTAINING*/
                                     /*THE PROCESSOR IDENTIFICATION*/
 DCL RT_CODE BIN FIXED(15) INITIAL(0); /*RETURN CODE*/
 DCL PROCESS_P POINTER; /*PARAMETER,POINTER TO PCB*/
 DCL PROCSTAT_P POINTER; /*PARAMETER,POINTER TO REGISTER STATE*/
 DCL POLICY CHAR(1); /*PARAMETER,'P'=PREEMPTIVE,'N'=NON PREEMPTIVE*/
```

```
MEMBER NAME  DISP2
 DCL  PROCID BIN FIXED(31); /*PARAMETER,PROCESS IDENTIFICATION*/
 DCL  CONDITION BIT(8); /*RELEASE CONDITION*/
 DCL  I BIN FIXED(15); /*COUNTER*/
 DCL HASH ENTRY(BIN FIXED(31)) RETURNS(BIN FIXED(15));

 ALLOCATE PR_TAB(*);
 ALLOCATE PROTAB(*);

 CALL $UNLOCK(R_LIST.LOCK);
 R_LIST.FW_P=NULL;
 R_LIST.BW_P=NULL;

 DO I=1 TO PR_NUMB;
 CALL $UNLOCK(PR_TAB.LOCK(I));
 PR_TAB.MODE(I)='IDLE';
 PR_TAB.ALLOC_P(I)=NULL;
 PR_TAB.REQUEST=0;
 END;

 DO I=1 TO T_LENGTH;
 CALL $UNLOCK(PROTAB.LOCK(I));
 PROTAB.CHAIN_P(I)=NULL;
 END;
 RETURN;

 ADD:ENTRY(PROCESS_P,POLICY) RETURNS(BIN FIXED(15));
 /*ADDS A NEW PROCESS TO THE DISPATCHER. THE STRATEGY MAY BE EITHER*/
 /*PREEMPTIVE OR NON PREEMPTIVE*/
 /*RETURN CODE VALUES,0=OK,1=NO SPACE*/

 ON AREA BEGIN;
    RT_CODE=1;
    RETURN(RT_CODE);
         END;

 ALLOCATE CONNECTOR;
 PCB_P,CONNECTOR.TEXT_P=PROCESS_P;
 DISP_STATE='UNDF'; /*UNDEFINED*/
 CONNECTOR.RET_REQ=0;
 PCB.PROC_STATE='00000000'B; /*NO WAITING CONDITION*/

 H_INDEX=HASH(PCB.PROCESS_ID);
 CALL $LOCK(PROTAB.LOCK(H_INDEX));
 CONNECTOR.CHAIN_P=PROTAB.CHAIN_P(H_INDEX);
 PROTAB.CHAIN_P(H_INDEX)=CONNECT_P;
 CALL $LOCK(CONNECTOR.LOCK);
 CALL $UNLOCK(PROTAB.LOCK(H_INDEX));
 DISP_STATE='READ';
 CALL $LOCK(R_LIST.LOCK);
 CALL INSERT(CONNECT_P);
 CALL $UNLOCK(R_LIST.LOCK);

 IF POLICY='P' THEN DO; /*PREEMPTIVE ADD*/
    CALL SIGNAL_PP(PROCESS_P->PCB.PRIORITY);
                    END;
                ELSE DO;
    CALL WAKE_PR; /*NON PREEMPTIVE ADD*/
                    END;
```

```
MEMBER NAME  DISP2
 RETURN(RT_CODE);

 RETIRE:ENTRY(PROCID,PROCESS_P,POLICY) RETURNS(BIN FIXED(15));
 /*INITIALIZES RETIRING OF A PROCESS FROM DISPATCHER. IF RETIRING MAY*/
 /*BE PERFORMED IMMEDIATELY, THE PCB-POINTER IS RETURNED TO THE */
 /*CALLING PROCEDURE*/
 /*RETURN CODE VALUES,0=OK,1=INITIALIZED,2=PROCESS NOT FOUND*/
 DCL HELP_P POINTER; /*TEMPORARY VARIABLE*/

 H_INDEX=HASH(PROCID);
 CALL $LOCK(PROTAB.LOCK(H_INDEX));
 PRE_P=ADDR(PROTAB(H_INDEX));
 CONNECT_P=PROTAB.CHAIN_P(H_INDEX);

 LOOP1:;
 IF CONNECT_P=NULL THEN DO;
    RT_CODE=2;
    RETURN(RT_CODE);
                          END;
 PCB_P=CONNECTOR.TEXT_P;
 IF PCB.PROCESS_ID¬=PROCID THEN DO;
    PRE_P=CONNECT_P;
    CONNECT_P=CONNECTOR.CHAIN_P;
    GOTO LOOP1;
                                   END;
 CALL $LOCK(CONNECTOR.LOCK);
 IF CONNECTOR.DISP_STATE='UNDF' THEN DO; /*UNDEFINED*/
                                         /*RELEASE IN PROGRESS*/
    CALL $UNLOCK(CONNECTOR.LOCK);
    CALL $UNLOCK(PROTAB.LOCK(H_INDEX));
    RT_CODE=2;
    RETURN(RT_CODE);
                                        END;
 IF CONNECTOR.DISP_STATE='WAIT' THEN DO;
    PRE_P->CONNECTOR.CHAIN_P=CONNECTOR.CHAIN_P;
    CONNECTOR.RET_REQ=1;
    CALL $UNLOCK(CONNECTOR.LOCK);
    CALL $UNLOCK(PROTAB.LOCK(H_INDEX));
    PROCESS_P=PCB_P;
    FREE CONNECTOR;
    RETURN(RT_CODE);
                                        END;
 IF CONNECTOR.DISP_STATE='READ' THEN DO;
    HELP_P=PRE_P;
    CALL $UNLOCK(CONNECTOR.LOCK);
    CALL $LOCK(R_LIST.LOCK);
    CALL $LOCK(CONNECTOR.LOCK);
    IF CONNECTOR.DISP_STATE='READ' THEN DO;
       PRE_P=CONNECTOR.BW_P;
       NEXT_P=CONNECTOR.FW_P;
       PRE_P->CONNECTOR.FW_P=NEXT_P;
       IF NEXT_P¬=NULL THEN NEXT_P->CONNECTOR.BW_P=PRE_P;
                                       END;
                                    ELSE;
    HELP_P->CONNECTOR.CHAIN_P=CONNECTOR.CHAIN_P;
    CALL $UNLOCK(R_LIST.LOCK);
    CALL $UNLOCK(CONNECTOR.LOCK);
    CALL $UNLOCK(PROTAB.LOCK(H_INDEX));
```

```
MEMBER NAME  DISP2
    PROCESS_P=PCB_P;
    FREE CONNECTOR;
    RETURN(RT_CODE);
                                          END;
 CONNECTOR.RET_REQ=1;
 CALL $UNLOCK(PROTAB.LOCK(H_INDEX));
 CALL $UNLOCK(CONNECTOR.LOCK);
 IF POLICY='P' THEN DO;
    CALL $LOCK(PR_TAB.LOCK(CONNECTOR.PR_ID));
    IF (PR_TAB.MODE(CONNECTOR.PR_ID)='ACTV')&
       (PR_TAB.REQUEST(CONNECTOR.PR_ID)=0) THEN DO;
       CALL $SIGNAL(CONNECTOR.PR_ID);
       PR_TAB.REQUEST(CONNECTOR.PR_ID)=1;
                                                 END;
                                              ELSE;
    CALL $UNLOCK(PR_TAB.LOCK(CONNECTOR.PR_ID));
                      END;
 RT_CODE=1;
 RETURN(RT_CODE);

 ASSIGN:ENTRY(PROCSTAT_P,PROCID) RETURNS(BIN FIXED(15));
 /*ASSIGNS A READY PROCESS TO THE INVOKING PROCESSOR AND PASSES THE*/
 /*POINTER TO THE REGISTER STATE AND THE PROCESS IDENTIFICATION BACK*/
 /*TO THE CALLING PROCEDURE*/
 /*RETURN CODE VALUES,0=OK,1=NO WORK,2=PROCESSOR ALREADY ALLOCATED*/

 IF PR_TAB.MODE($CPU_ID)¬='WORK' THEN DO;
    CALL $LOCK(R_LIST.LOCK);

 LOOP2:;
    CONNECT_P=R_LIST.FW_P;
    IF CONNECT_P=NULL THEN DO; /*NO WORK*/
       CALL $UNLOCK(R_LIST.LOCK);
       RT_CODE=1;
                         END;
                      ELSE DO;
       CALL $LOCK(CONNECTOR.LOCK);
       R_LIST.FW_P=CONNECTOR.FW_P;
       NEXT_P=R_LIST.FW_P;
       IF NEXT_P¬=NULL THEN NEXT_P->CONNECTOR.BW_P=ADDR(R_LIST);
       IF CONNECTOR.RET_REQ=1 THEN DO; /*RETIRE REQUEST*/
          CONNECTOR.DISP_STATE='UNDF'; /*UNDEFINED*/
          CALL $UNLOCK(CONNECTOR.LOCK);
          GOTO LOOP2;
                                    END;
                                 ELSE DO;
          PCB_P=CONNECTOR.TEXT_P;
          CONNECTOR.PR_ID=$CPU_ID;
          CALL $UNLOCK(CONNECTOR.LOCK);
          CALL $UNLOCK(R_LIST.LOCK);
          CALL $LOCK(PR_TAB.LOCK($CPU_ID));
          PR_TAB.ALLOC_P($CPU_ID)=CONNECT_P;
          PR_TAB.MODE($CPU_ID)='WORK';
          PR_TAB.PRIORITY($CPU_ID)=PCB.PRIORITY;
          PR_TAB.REQUEST($CPU_ID)=0;
          CALL $UNLOCK(PR_TAB.LOCK($CPU_ID));
          PROCSTAT_P=ADDR(PCB.REG_STATE);
          PROCID=PCB.PROCESS_ID;
```

```
MEMBER NAME  DISP2
                                        END;
                                    END;
                                        END;
                                    ELSE DO;
   RT_CODE=2;
                                        END;
 RETURN(RT_CODE);

 RELEASE:ENTRY(CONDITION,PROCESS_P) RETURNS(BIN FIXED(15));
 /*RELEASES THE PROCESSOR FROM ALLOCATED PROCESS AND PASSES THE PCB-*/
 /*POINTER BACK TO THE CALLING PROCEDURE*/
 /*RETURN CODE VALUES,0=OK,1=PROCESSOR NOT ALLOCATED*/
 DCL TEMP_ID BIN FIXED(31); /*TEMPORARY PROCESS IDENTIFICATION*/

 IF PR_TAB.MODE($CPU_ID)='WORK' THEN DO;
    CALL $LOCK(PR_TAB.LOCK($CPU_ID));
    PR_TAB.MODE($CPU_ID)='WAKE';
    CONNECT_P=PR_TAB.ALLOC_P($CPU_ID);
    CALL $LOCK(CONNECTOR.LOCK);
    CALL $UNLOCK(PR_TAB.LOCK($CPU_ID));
    PCB_P=CONNECTOR.TEXT_P;
    PCB.PROC_STATE=PCB.PROC_STATE|CONDITION;
    CONNECTOR.DISP_STATE='UNDF'; /*UNDEFINED*/
    TEMP_ID=PCB.PROCESS_ID;
    CALL $UNLOCK(CONNECTOR.LOCK);
    H_INDEX=HASH(TEMP_ID);
    CALL $LOCK(PROTAB.LOCK(H_INDEX));
    PRE_P=ADDR(PROTAB.LOCK(H_INDEX));

 LOOP3:;
    CONNECT_P=PRE_P->CONNECTOR.CHAIN_P;
    IF CONNECT_P¬=NULL THEN DO;
       PCB_P=CONNECTOR.TEXT_P;
       IF TEMP_ID=PCB.PROCESS_ID THEN DO;
          PRE_P->CONNECTOR.CHAIN_P=CONNECTOR.CHAIN_P;
          FREE CONNECTOR;
          PROCESS_P=PCB_P;
                                        END;
                                    ELSE DO;
          PRE_P=CONNECT_P;
          GOTO LOOP3;
                                        END;
                                END;
                            ELSE;
    CALL $UNLOCK(PROTAB.LOCK(H_INDEX));
                                    END;
                                ELSE RT_CODE=1;
 RETURN(RT_CODE);

 WAIT:ENTRY(CONDITION) RETURNS(BIN FIXED(15));
 /*PUTS AN ACTIVE PROCESS INTO THE WAITING STATE*/
 /*RETURN CODE VALUES,0=OK,1=RETIRE REQUEST,2=PROCESSOR NOT*/
 /*ALLOCATED*/

 IF PR_TAB.MODE($CPU_ID)='WORK' THEN DO;
    CALL $LOCK(PR_TAB.LOCK($CPU_ID));
    PR_TAB.MODE($CPU_ID)='WAKE';
    CALL $UNLOCK(PR_TAB.LOCK($CPU_ID));
```

```
MEMBER NAME  DISP2
    CONNECT_P=PR_TAB.ALLOC_P($CPU_ID);
    CALL $LOCK(CONNECTOR.LOCK);
    PCB_P=CONNECTOR.TEXT_P;
    PCB.PROC_STATE=CONDITION;
    IF CONNECTOR.RET_RFQ=1 THEN DO;
       CALL $UNLOCK(CONNECTOR.LOCK);
       CALL $LOCK(PR_TAB.LOCK($CPU_ID));
       PR_TAB.MODE($CPU_ID)='WORK';
       CALL $UNLOCK(PR_TAB.LOCK($CPU_ID));
       RT_CODE=1;
                                  END;
                               ELSE DO;
       CONNECTOR.DISP_STATE='WAIT';
       CALL $UNLOCK(CONNECTOR.LOCK);
                                  END;
                                     END;
                                   ELSE RT_CODE=2;
 RETURN(RT_CODE);

 READY:ENTRY(PROCID,CONDITION) RETURNS(BIN FIXED(15));
 /*THE NAMED PROCESS WILL BE ACTIVATED IF ALL WAITING CONDITIONS*/
 /*CAN BE REMOVED*/
 /*RETURN CODE VALUES,0=OK,1=PROCESS NOT FOUND*/

 H_INDEX=HASH(PROCID);
 CALL $LOCK(PROTAB.LOCK(H_INDEX));
 CONNECT_P=PROTAB.CHAIN_P(H_INDEX);

 LOOP4:;
 IF CONNECT_P¬=NULL THEN DO;
    PCB_P=CONNECTOR.TEXT_P;
    IF PCB.PROCESS_ID=PROCID THEN DO;
       CALL $LOCK(CONNECTOR.LOCK);
       CALL $UNLOCK(PROTAB.LOCK(H_INDEX));
       PCB.PROC_STATE=(¬CONDITION)&(PCB.PROC_STATE);
       IF PCB.PROC_STATE='00000000'B THEN DO;
          CONNECTOR.DISP_STATE='READ';
          CALL $LOCK(R_LIST.LOCK);
          CALL $UNLOCK(CONNECTOR.LOCK);
          CALL INSERT(CONNECT_P);
          CALL $UNLOCK(R_LIST.LOCK);
                                        END;
                                     ELSE DO;
          CALL $UNLOCK(CONNECTOR.LOCK);
                                        END;
                                  END;
                               ELSE DO;
       CONNECT_P=CONNECTOR.CHAIN_P;
       GOTO LOOP4;
                                  END;
                            END;
                         ELSE DO;
    CALL $UNLOCK(PROTAB.LOCK(H_INDEX));
    RT_CODE=1;
                            END;
 RETURN(RT_CODE);

 REREADY:ENTRY RETURNS(BIN FIXED(15));
```

```
MEMBER NAME  DISP2
 /*PUTS AN ACTIVE PROCESS INTO THE READY STATE*/
 /*RETURN CODE VALUES,0=OK,1=RETIRE REQUEST,2=PROCESSOR*/
 /*NOT ALLOCATED*/

 IF PR_TAB.MODE($CPU_ID)='WORK' THEN DO;
    CALL $LOCK(PR_TAB.LOCK($CPU_ID));
    PR_TAB.MODE($CPU_ID)='WAKE';
    CALL $UNLOCK(PR_TAB.LOCK($CPU_ID));
    CONNECT_P=PR_TAB.ALLOC_P($CPU_ID);
    CALL $LOCK(CONNECTOR.LOCK);
    IF CONNECTOR.RET_REQ=1 THEN DO;
       CALL $UNLOCK(CONNECTOR.LOCK);
       CALL $LOCK(PR_TAB.LOCK($CPU_ID));
       PR_TAB.MODE($CPU_ID)='WORK';
       CALL $UNLOCK(PR_TAB.LOCK($CPU_ID));
                                  END;
                              ELSE DO;
      CONNECTOR.DISP_STATE='READ';
       CALL $UNLOCK(CONNECTOR.LOCK);
                                  END;
                                       END;
                                  ELSE RT_CODE=2;
 RETURN(RT_CODE);

 STOP:ENTRY RETURNS(BIN FIXED(15));
 /*PUTS THE INVOKING PROCESSOR INTO THE IDLE STATE*/
 /*RETURN CODE VALYES,0=OK,1=PROCESSOR STILL ALLOCATED*/

 CALL $LOCK(PR_TAB.LOCK($CPU_ID));
 IF PR_TAB.MODE($CPU_ID)='WORK' THEN DO;
    CALL $UNLOCK(PR_TAB.LOCK($CPU_ID));
    RT_CODE=1;
                                       END;
                                  ELSE DO;
    PR_TAB.MODE($CPU_ID)='IDLE';
    CALL $UNLOCK(PR_TAB.LOCK($CPU_ID));
    CALL $IDLE;
                                         END;
 RETURN(RT_CODE);

 WAKE_PR:ENTRY; /*FIND AND WAKEUP ANY IDLE PROCESSOR*/
 DCL PR_INDEX BIN FIXED(15); /*PROCESSOR INDEX*/
 PR_INDEX=0;

 LOOP5:;
 PR_INDEX=PR_INDEX+1;
 CALL $LOCK(PR_TAB.LOCK(PR_INDEX));
 IF (PR_TAB.MODE(PR_INDEX)='IDLE')&(PR_INDEX¬=$CPU_ID) THEN DO;
    CALL $WAKEUP(PR_INDEX);
    PR_TAB.MODE(PR_INDEX)='WAKE' ;
    CALL $UNLOCK(PR_TAB.LOCK(PR_INDEX));
    IF PR_INDEX<PR_NUMB THEN GOTO LOOP5;
                        ELSE;
                                                              END;

 SIGNAL_PR:ENTRY(PROC_PRIO);
 /*SEEKS A PROCESSOR WITH A LOWER PRIORITY THAN PROC_PRIO AND*/
 /*INTERRUPTS IT BY SIGNALING*/
```

```
MEMBER NAME  DISP2
 DCL (PROC_PRIO,PR_VAR1,PR_VAR2) BIN FIXED(15);
 PR_VAR1=0;

 LOOP6:;
 PR_VAR1=PR_VAR1+1;
 IF PR_VAR1=$CPU_ID THEN GOTO LOOP6;
 IF PR_VAR1>PR_NUMB THEN RETURN;
 CALL $LOCK(PR_TAB.LOCK(PR_VAR1));
 IF PR_TAB.MODE(PR_VAR1)='IDLE' THEN DO;
    CALL $WAKEUP(PR_VAR1);
    PR_TAB.MODE(PR_VAR1)='WAKE';
    CALL $UNLOCK(PR_TAB.LOCK(PR_VAR1));
    RETURN;
                                        END;
 IF PR_TAB.MODE(PR_VAR1)¬='WORK' THEN DO;
    CALL $UNLOCK(PR_TAB.LOCK(PR_VAR1));
    GOTO LOOP6;
                                         END;
 IF (PR_TAB.PRIORITY(PR_VAR1)<=PROC_PRIO)|
    (PR_TAB.REQUEST(PR_VAR1)=1)  THEN DO;
        CALL $UNLOCK(PR_TAB.LOCK(PR_VAR1));
        GOTO LOOP6;
                                         END;
 PR_VAR2=PR_VAR1;

 LOOP7:;
 PR_VAR2=PR_VAR1+1;
 IF PR_VAR2=$CPU_ID THEN GOTO LOOP7;
 IF PR_VAR2>PR_NUMB THEN DO;
    CALL $SIGNAL(PR_VAR1);
    PR_TAB.REQUEST(PR_VAR1)=1; /*SIGNAL REQUEST*/
    CALL $UNLOCK(PR_TAB.LOCK(PR_VAR1));
                          END;
 CALL $LOCK(PR_TAB.LOCK(PR_VAR2));
 IF PR_TAB.MODE(PR_VAR2)='IDLE' THEN DO;
    CALL $WAKEUP(PR_VAR2);
    PR_TAB.MODE(PR_VAR2)='WAKE';
    CALL $UNLOCK(PR_TAB.LOCK(PR_VAR1));
    CALL $UNLOCK(PR_TAB.LOCK(PR_VAR2));
    RETURN;
                                        END;
 IF PR_TAB.MODE(PR_VAR2)¬='WORK' THEN DO;
    CALL $UNLOCK(PR_TAB.LOCK(PR_VAR2));
    GOTO LOOP7;
                                         END;
 IF (PR_TAB.PRIORITY(PR_VAR2)<=PR_TAB.PRIORITY(PR_VAR1))|
    (PR_TAB.REQUEST(PR_VAR2)=1) THEN DO;
    CALL $UNLOCK(PR_TAB.LOCK(PR_VAR2));
    GOTO LOOP7;
                                        END;
 CALL $UNLOCK(PR_TAB.LOCK(PR_VAR1));
 PR_VAR1=PR_VAR2;
 GOTO LOOP7;

 INSERT:ENTRY(ELEMENT_P);
 DCL TEMPRIO BIN FIXED(15);
 DCL ELEMENT_P POINTER; /*POINTER TO A CONNECTOR*/
```

```
MEMBER NAME   DISP2
 CONNECT_P=ELEMENT_P;
 PCB_P=CONNECTOR.TEXT_P;
 TEMPRIO=PCB.PRIORITY;
 PRE_P=ADDR(R_LIST);

 LOOP8:;
 NEXT_P=PRE_P->CONNECTOR.FW_P;
 IF NEXT_P=NULL THEN DO;
    PRE_P->CONNECTOR.FW_P=CONNECT_P;
    CONNECTOR.FW_P=NULL;
    CONNECTOR.BW_P=PRE_P;
                        END;
                  ELSE DO;
    PCB_P=NEXT_P->CONNECTOR.TEXT_P;
    IF TEMPRIO < PCB.PRIORITY THEN DO;
       CONNECTOR.FW_P=NEXT_P;
       CONNECTOR.BW_P=PRE_P;
       PRE_P->CONNECTOR.FW_P=CONNECT_P;
       NEXT_P->CONNECTOR.BW_P=CONNECT_P;
                                      END;
                                ELSE DO;
       PRE_P=NEXT_P;
       GOTO LOOP8;
                                      END;
                        END;
 RETURN;

 END DISP_2;
```

Anhang B

System- und Pseudoinstruktionen der PLIM

Neben den durch die PL/1-Konventionen festgelegten Standardinstruktionen setzt sich der Instruktionsvorrat für die PL/1-Maschine aus Systeminstruktionen und Pseudoinstruktionen zusammen.
Systeminstruktionen bilden das notwendige Instrument zur Konfigurations- und Zustandskontrolle der virtuellen PL/1-Maschine. Siebzehn derartige Instruktionen sind gegenwärtig implementiert, die über Prozeduraufrufe von allen Modellprogrammen benutzbar sind:

$CALL(Programmname)
$RETURN
$JUMP(Programmname)
$RESUME(Programmstatus)
$SAVE(Programmstatus)
$LOAD_PARMADDR(Parameteradresse)
$STORE_PARMADDR(Parameteradresse)
$INT_STATUS(Ablageadresse)
$ENABLE
$DISABLE
$MASK
$UNMASK
$LOCK(Lockadresse)
$UNLOCK(Lockadresse)
$SIGNAL(Prozessor, Priorität)
$IDLE
$SVC(SVC-Nummer)

Durch die Instruktionen $CALL, $RETURN, $JUMP, $RESUME, $SAVE sind kontrollierte Übergänge zwischen Modellprozeduren auf der Basis von Unterprogrammaufrufen oder Sprüngen möglich. Die dazu erforderlichen Speicherallokationen und -freigaben, Aufräumungsarbeiten und Normierungen in den SAVE-Bereichen der Modellprogramme werden automatisch durchgeführt und sind damit Bestand-

teil der Hardware der PL/1-Maschine. Der Systemprogrammierer wird dadurch von der Speicherorganisation auf der untersten Ebene der Betriebsorganisation befreit und kann sich in verstärktem Maße den algorithmischen Eigenschaften der zu untersuchenden Modelle widmen.

Bei Unterprogrammschachtelungen durch $CALL werden die SAVE-Bereiche der beteiligten Modellprozeduren automatisch gekellert und die Rücksprungsadressen gerettet.

Durch $LOAD_PARMADDR und $STORE_PARMADDR werden bei Unterprogrammaufrufen Parameter übergeben bzw. abgeholt.
Die Instruktionen $INT_STATUS, $ENABLE, $DISABLE, $MASK, $UNMASK werden für die Abfrage des Interruptstatus bzw. Verändern des Interruptmaskenregister benötigt.
$LOCK und $UNLOCK als Synchronisationshilfsmittel sowie $SIGNAL als Mittel der Interprozessor-Kommunikation wurden ebenso wie die $IDLE-Instruktion bei der Beschreibung der Dispatcher-Algorithmen bereits eingeführt.
Die $SVC-Instruktion (Supervisor Call) ist in ihrer Wirkung mit denen realer Maschinen identisch.

Fünf Pseudoinstruktionen, die ausschließlich Simulations-, Test- und Meßzwecken dienen, stehen zusätzlich zur Verfügung:

$RUN(Zeit)
$INTERRUPT(Zeit, Interrupt-Nr., Priorität)
$CLOCK
$CPU_TIME
$TEST(Modus)

Durch Ausführung der $RUN-Instruktion wird ein Rechnerkern für die Dauer der angegebenen Zeit in den PSEUDORUNNING-Modus versetzt.

Mit Hilfe der $INTERRUPT-Instruktion können beliebige Interrupts zu einem definierten Zeitpunkt an dem Rechner generiert werden, der diese Instruktion ausführt.

Beide Instruktionen ermöglichen es auf einfache Weise, Programmlaufzeiten, Ein/Ausgabe-Transferzeiten, Gerätezugriffsverzögerungen und alle Arten von Geräte- und Kanalinterrupts zu simulieren und ergänzen somit den Satz der Systeminstruktionen in sinnvoler Weise.

Die Instruktionen $CLOCK und $CPU_TIME dienen der Feststellung der absoluten simulierten Zeit (in einer wählbaren Grundtaktzeit) bzw. der Messung von Programmlaufzeiten.
Der Satz von Pseudoinstruktionen wird durch die $TEST-Instruktion vervollkommnet, die eine statementweise Protokollierung aller Programmabläufe in einem wählbaren Zeitintervall erlaubt und damit eine wertvolle Unterstützung beim Austesten der Modelle bildet.

Anhang C

Die Komponenten des Modells und ihre Funktionsweise
(vgl. Abb. 14)

Interrupthandler

Im Interrupthandler werden im wesentlichen vier Arten von Interrupts bearbeitet: SVC's, Long-Wait-Ende, Short-Wait-Ende und SIGNAL-Interrupts. Folgende SVC's werden dabei von den Benutzerprozessen bzw. Supervisorprozessen erzeugt:

- Service-Requests an den Long-Wait-Handler,
- Service-Requests an den Short-Wait-Handler,
- Synchronisationsanweisungen für Botschaften (send_message, get_message, wait_message),
- Synchronisationsanweisungen an den Dispatcher, die eine Reaktivierung wartender Prozesse auslösen.

Neben der Interruptbearbeitung, die im wesentlichen durch Dispatcher-Elementarfunktionen sowie Kommunikationsfunktionen erfolgt, werden im Interrupthandler zwei, für die Auswertung notwendigen Zeitintervalle gemessen und aufakkummuliert:
die Zeiten, in denen die Prozessoren untätig sind und der Problemdurchsatz, der durch die Benutzerprozesse erzeugt wird.

Kommunikationsfunktionen

Als Grundlage der Kommunikation zwischen den Supervisorprozessen (Scheduler, Long-Wait-Handler, Short-Wait-Handler) wurde ein einfaches Botschaftensystem implementiert, das auf Hansen [20] zurückgeht. Die Botschaftenkanäle für die drei Supervisorprozesse werden bei der Systeminitialisierung statisch angelegt und können durch drei Funktionen manipuliert werden:

- send_message (empfänger, message_id, message_buf)

Diese Funktion kopiert eine Botschaft aus dem durch message_buf bezeichneten internen Speicher des Senders in den Botschaftenkanal des Empfängers, der im Speicher des Systemnukleus liegt.

Der Empfänger wird aktiviert, wenn er im blockierten Zustand auf eine Botschaft wartet. Der sendende Prozess kann daraufhin seine Arbeit unmittelbar fortsetzen.

- wait_message (sender, message_id, message_buf)

Diese Funktion zwingt den ausführenden Prozess solange in den Zustand BLOCKIERT bzw. SUSPENDIERT bis eine Botschaft in seinem Botschaftenkanal eintrifft. Dann wird der Prozess reaktiviert, die Parameter sender, message_id übergeben und die Botschaft in einem durch message_buf angegebenen Prozess-internen Pufferbereich kopiert. Die Botschaft wird daraufhin vernichtet. Ist der Botschaftenkanal bei Ausführung der Funktion nicht leer, so wird verfahren wie unter get_message.

- get_message

Die Funktion versorgt den Empfänger-Prozess mit der nächsten, im Botschaftenkanal enthaltenen Botschaft. Die Botschaft wird in dem durch message_buf bezeichneten internen Speicherbereich abgelegt und die Identifikation des Senders (sender) und der Botschaft (message_id) dem Empfänger übergeben. Daraufhin wird die Kopie der Botschaft im Botschaftenkanal vernichtet.
Ist der Botschaftenkanal bei Ausführung der Funktion leer, so wird der Empfänger durch Rückgabe des Null-Pointers darüber informiert, bleibt aber weiterhin im Zustand AKTIV.

Scheduler

Dieser Supervisorprozess sammelt alle Prozesse, die durch die release-Funktion aus dem Dispatcher entfernt und mittels einer Botschaft an ihn weitergeleitet wurden. Sie befinden sich dort in dem Dispatcher-externen Wartezustand SUSPENDIERT und werden nach Aufforderung über eine spezielle Botschaft aus diesem Zustand entlassen. Der Scheduler erzeugt zu diesem Zweck einen SVC mit Angabe des PCB, der zum Aufruf der add-Funktion in dem zugehörigen Zweig der Interruptroutine führt.
Bei leerem Botschaftenkanal geht der Scheduler nach Durchführung seiner Aufgaben durch Aufruf der wait_message-Funktion in den BLOCKIERT-Zustand über.

Long-Wait-Handler

Dieser Supervisorprozess generiert auf Grund von Service-Requests, die in Form von Botschaften eintreffen, zufallsverteilte Long-Wait-Perioden und initialisiert die entsprechenden Long-Wait-Ende-Interrupts. Sie werden nach ihrem Eintreffen unter Angabe des auslösenden Prozesses in einer Botschaft verschlüsselt und in den Botschaftenkanal des Long-Wait-Handlers eingereiht. Der Long-Wait-Handler leitet durch einen speziellen SVC daraufhin die Reaktivierung des wartenden Prozesses ein.
Liegen keine Service-Requests im Botschaftenkanal des Long-Wait-Handlers vor, so geht er durch Aufruf der wait_message-Funktion selbst in den SUSPENDIERT-Zustand über.

Short-Wait-Handler

Die Funktionsweise dieses Supervisorprozesses entspricht der des Long-Wait-Handlers. Die generierten Short-Wait-Perioden werden allerdings durch einen unabhängigen Zufallszahlengenerator erzeugt.
Liegen keine Service-Requests im Botschaftenkanal des Short-Wait-Handlers vor, so geht er durch Aufruf der wait_message-Funktion in den BLOCKIERT-Zustand über.

Benutzerprozesse

Sie erzeugen unter Benutzung eines weiteren unabhängigen Zufallszahlengenerators Laufzeiten, die mit der $RUN-Pseudoinstruktion (Anhang B) auf dem jeweiligen Prozessor verwirklicht werden. Nach Abschluß jeder Arbeitsphase wird durch Aufruf eines binären Zufallszahlengenerators entschieden, in welchem der zwei möglichen Wartezustände (BLOCKIERT oder SUSPENDIERT) ein Prozess eintritt. Er wird durch einen entsprechenden SVC eingeleitet.
Nach Ablauf der Wartezeit und Rückgabe der Kontrolle an den Prozess wird der Zyklus von neuem gestartet.