HAHN-MEITNER-INSTITUT FÜR KERNFORSCHUNG BERLIN GMBH

Bereich Datenverarbeitung und Elektronik

DM 49

HMI-B 236

Juli 1977

STRUKTURIERTER BETRIEBSSYSTEMENTWURF MIT CONCURRENT PASCAL

Christian Lengauer

Berlin-Wannsee

Mein besonderer Dank gilt Ass. Prof. Dr. K.-P. Löhr und Prof. Dr. H.-J. Töpfer, deren wertvolle Unterstützung bei fachlichen und organisatorischen Problemen die Fertigstellung dieser Arbeit möglich gemacht hat.
H. Goullon, R. Isle und M. Steinacker danke ich für hilfreiche Diskussionen zum Thema Verifikation und G. Sowada für die Umstellung des benutzten Skriptprogramms auf meine Bedürfnisse.

INHALT

1 Zielsetzung

Die Entwicklung eines Betriebssystems sollte aus folgenden Phasen bestehen:

a) Spezifikation
b) Design
c) Implementation
d) Verifikation

Die Spezifikation ist der Ausgangspunkt für die Konstruktion eines Systems. Sie legt die Aufgaben fest, die es durchführen soll. Damit wird eine Menge von möglichen Programmen definiert, die am Ende der Systementwicklung stehen können.
In der ersten Phase der Konstruktion, dem Design, wird die Struktur des Betriebssystems entworfen, etwa seine modulare Zergliederung, die nebenläufigen Aktivitäten, usw. Die Menge der möglichen Programme wird also durch zusätzliche Forderungen eingeschränkt.
Die Implementation ist die Erstellung eines Programms mit der beim Design festgelegten Struktur. Dieses Programm ist der Endpunkt der Entwicklung, das System.
Die Entwicklung eines Betriebssystems stellt also eine schrittweise Konkretisierung dar, die Einschränkung einer durch die Aufgaben des Systems definierten Menge von Programmen auf ein Element daraus. Unter der Voraussetzung, daß man sich auf dem Konstruktionsweg an einmal gefällte Entscheidungen hält, ist die Zugehörigkeit des Zielprogramms zu der Ausgangsmenge, dh. seine Konsistenz mit der Spezifikation garantiert.
Die Verifikation ist ein Mittel, diese Übereinstimmung zwischen Spezifikation und Implementation nachzuweisen. Ein geradliniger Konstruktionsweg drückt sich in einer überschaubaren Verifikation aus.

Diese vier Entwicklungsphasen lassen sich nicht strikt voneinander trennen, sondern es gibt im allgemeinen Abhängigkeiten zwischen ihnen. So wird sich z. B. das Design nach dem Instrument der Implementation, der verwendeten Sprache, richten. Die Implementation wiederum wird, sofern keine zu großen Effizienzverluste die Folge sind, mit Rücksicht auf eine bequeme Verifikation vorgenommen werden.

Für die drei Konstruktionsphasen sind in den letzten Jahren strukturierende Konzepte entwickelt worden. Ziel dieser Arbeit ist es, Erkenntnisse über die Eignung einiger dieser Ansätze zu gewinnen. Um praktische Erfahrungen zu sammeln, wird mit ihrer Hilfe ein kleines Kommunikationssystem (COCO = COncurrent COmmunication system) spezifiziert, entworfen, implementiert und zum Teil verifiziert. Bei dem Design wird die Verwendung einer nichtsequentiellen Programmiersprache vorausgesetzt. Für die Implementation wird die Sprache Concurrent Pascal gewählt. Die Verwendung einer geeigneten Sprache ist wesentlich für den Erfolg der Systementwicklung. Es ist sinnvoll auf eine Sprache zurückzugreifen, die speziell für die Programmierung von Betriebssystemen konzipiert ist. Das Fehlen unpassender Sprachstrukturen verhindert trickreiches Programmieren und erleichtert die Verifikation und Dokumentation des Systems.

2 Einleitung

2.1 Spezifikation: Rechner-Netzwerke

Die Verbindung mehrerer Rechnereinheiten zu einem Netz hat verschiedene Vorteile ([DEC74]):

a) resource sharing

Die im Netz enthaltenen Hardware- und Softwareeinrichtungen sind mehreren am Netz angeschlossenen Benutzern zugänglich. So können kleinere Experimentierstationen etwa von der Rechenkapazität eines Großrechners Gebrauch machen oder auf große am Netz angeschlossene Datenspeicher zugreifen (device sharing). Ebenso können mehrere Netzkomponenten zentrale Programm- und Dateibibliotheken verwenden (program sharing, file sharing).

b) Dezentralisierung

Statt die Durchführung vieler unter Umständen völlig verschiedenartiger Aufgaben in einem Rechner zusammenzufassen, wird jedes Problem ökonomischer und übersichtlicher in einer für seine Bearbeitung geeigneten Konfiguration behandelt. In einer Netzkomponente auftretende Hardware- oder Softwarefehler haben weniger Auswirkungen auf die Funktionsfähigkeit des gesamten Netzes. Veränderungen der Netzstruktur können während des Rechenbetriebes vorgenommen werden.

Die für ein Kommunikationsnetz charakteristischen Aktivitäten betreffen die Übertragung von Daten zwischen den einzelnen Netzkomponenten.

In dieser Arbeit wird nur ein einfacher Spezialfall eines Kommunikationsnetzes betrachtet, nämlich ein sternförmiges Netz (Abb. 2.1.a), in dem ein Knotenrechner alle weiteren Netzkomponenten (DTE = Data Terminal Equipment) miteinander verbindet. Die Verbindung zwischen dem Knotenrechner und eines DTE heißt Datenleitung oder Kanal (im weiteren Sinne). Der Knotenrechner hat die Aufgabe, die Kommunikation zwischen den DTEs zu steuern und zu überwachen. Er übernimmt aber im allgemeinen keine Bearbeitungen von Problemen, die nicht die Datenübertragung betreffen (computations).

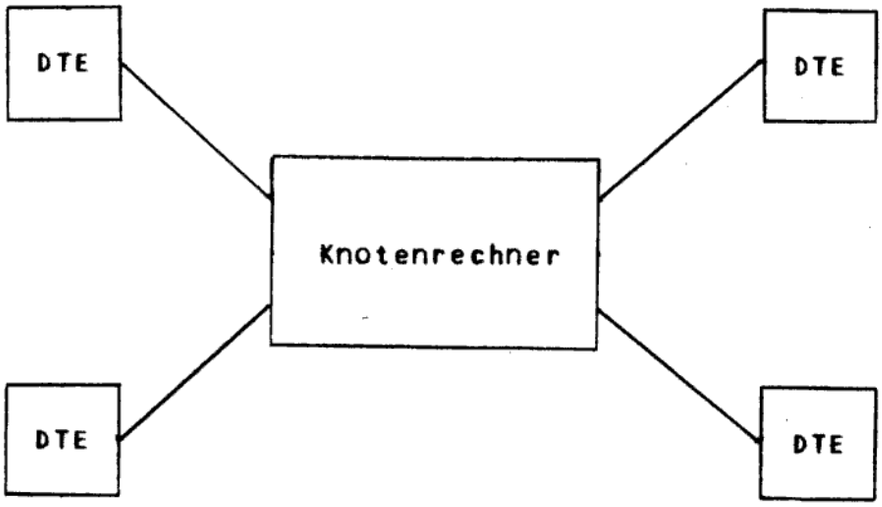

Abb. 2.1.a Sternförmiges Kommunikationsnetz

Die wesentliche Aufgabe, die der Knotenrechner eines sternförmigen Netzes zu erfüllen hat, ist die Verknüpfung der Quellen- und Zielkomponente für einen Datentransfer. Sie besteht aus zwei Teilproblemen:

a) routing

Für die Datenübertragung muß ein geeigneter Weg durch das Netz festgelegt werden. Dazu sind die Netzkomponenten, die die Übertragungsdaten passieren müssen, und für jede dieser Komponenten eventuelle Datenbehandlungen zu bestimmen (external bzw. internal routing).

b) switching

Dieser Begriff umfaßt die verschiedenen Möglichkeiten, wie lange eine einmal etablierte Verbindung aufrecht erhalten werden kann. Je länger dies der Fall ist, desto weniger Pseudoparallelität (interleaved transmission) ist bei Datenübertragungen im Netz möglich. Man kann für die Übertragung eines vollständigen Dialoges ein und dieselbe Verbindung benutzen (circuit switching) oder für mehr oder weniger große Übertragungseinheiten (packets) jeweils eine neue Verbindung etablieren (packet switching).

COCO ist ein Betriebssystem für einen solchen Knotenrechner. Seine Spezifikation wird umfassen:
a) die Bestimmung (internal routing) und Durchführung der geeigneten Behandlungen der Übertragungsdaten und ihre Leitung zum richtigen Ausgabekanal (external routing),
b) ein event-gesteuertes packet-switching unter Berücksichtigung von Realzeitanforderungen.

2.2 Design: Konstruktionskonzepte

In diesem Kapitel werden einige von Parnas ([Par74, Par75]) empfohlene Konzepte angesprochen, die beim Design des COCO Systems verwendet werden:

1.) Programmfamilien

Eine Programmfamilie ist eine Menge von Programmen, für die es sich lohnt, zunächst die gemeinsamen Eigenschaften ihrer Elemente zu betrachten, bevor man sich Unterschieden zuwendet. Die Spezifikation eines Systems definiert eine Programmfamilie. Bei der Konstruktion gefällte Entwurfsentscheidungen führen über immer kleiner werdende Unterfamilien zum Zielelement. Eine Änderung, dh. ein Übergehen von einer auf eine andere Unterfamilie sollte nur über den Rückschritt zu einer beide umfassenden Unterfamilie erfolgen. Konkret: Statt eine Designentscheidung in Richtung auf eine andere abzuändern, sollte man den Entwurfsprozeß an einem Punkt neu beginnen, wo diese Entscheidung noch nicht gefällt ist, und geradlinig auf die andere gewünschte Entscheidung zusteuern. Dieses Vorgehen setzt eine gewissenhafte Dokumentation aller Entwurfsentscheidungen voraus.

Es gibt mehrere Möglichkeiten, Programmfamilien zu entwickeln:

a) stepwise refinement

Diese von Dijkstra ([DDH72]) eingeführte Vorgehensweise reflektiert die schrittweise Konkretisierung, die zur Entwicklung von Programmfamilien notwendig ist. Abstrakte Statements werden durch immer konkretere ersetzt, bis eine der Syntax einer Sprache genügende Statementfolge entstanden ist (man kann die abstrakten Statements auch als Prozeduraufrufe verstehen). Die Grenzen dieser Methode bestehen in einer vonvornherein durch die Aufeinanderfolge der Statements festgelegten Reihenfolge, die eine unerwünschte Einschränkung der Programmfamilie bedeuten kann.

b) Moduln

Ein Modul ist im Sinne von Parnas eine Aktionseinheit und zugleich die Konkretisierung einer oder mehrerer wichtiger Entwurfsentscheidungen. Diese Definition beruht auf dem Wunsch, Entwurfsentscheidungen möglichst konzentriert vornehmen zu können. Eine Programmfamilie wird dann durch eine Menge von Moduln repräsentiert, die die Entwurfsentscheidungen bezüglich der abstrakten Eigenschaften der Familie darstellen. Bei geeigneter Modularisierung ist die Aufeinanderfolge der durch die einzelnen Moduln zur Verfügung gestellten Aktionen nicht unnötig eingeschränkt. Die Grenzen dieser Methode bestehen in der Notwendigkeit, äußerste Disziplin bei der Definition der Moduln walten zu lassen, um eine "geeignete" Modularisierung zu erhalten.

In der Praxis sollten beide Prinzipien dem aktuellen Fall angemessen miteinander verschmolzen werden.

2.) Hierarchische Struktur

Eine Menge M trägt eine hierarchische Struktur, wenn zwischen je zwei Elementen $a,b \in M$ eine Relation R(a,b) existiert, sodaß es Untermengen (Ebenen) $M(i) \subset M$ $(i=0,1,2,\ldots)$ gibt mit

i) $M(0) := \{a \mid a \in M$ und es gibt kein $b \in M$ mit $R(a,b)\}$

ii) $M(i) := \{a \mid a \in M$ und es gibt $b \in M(i-1)$ mit $R(a,b)$ und für alle $b \in M$ mit $R(a,b)$ gilt $b \in M(j)$ mit $j<i\}$, $(i>0)$

Hierarchische Strukturen können durch ihre einschränkenden Forderungen an die betrachtete Relation helfen, die Abhängigkeiten zwischen den Elementen der betrachteten Menge (etwa irgendwie gearteten Teilen von Betriebssystemen) einfach zu halten.

Beispiel: Die Programm-Hierarchie

M := Menge der als Prozeduren aufrufbaren Unterprogramme

R(a,b) :<=> a ruft b auf und a wird als inkorrekt angesehen, wenn b inkorrekt arbeitet (Interpretation nach Parnas: Prozedur a benutzt Prozedur b)

Gliedert die Relation "benutzt" ein aus Unterprogrammen aufgebautes Programmsystem gemäß i) und ii), so wird das System hierarchisch strukturiert genannt.

Zum Entwurf von Programmsystemen mit dieser Hierarchie gibt es zwei konträre Ansätze:

a) outside-in

Ausgehend von einer genauen Vorstellung über die Aufgaben des Systems wird schrittweise dessen innere Struktur ermittelt. Dies garantiert einen geradlinigen Konstruktionsweg unter Berücksichtigung aller erwünschten Systemeigenschaften.

b) inside-out

Die Operationen einer zugrundegelegten abstrakten Maschine werden zu immer umfassenderen Operationen verknüpft. Die Zieloperationen stellen das gewünschte System dar.

Beide Alternativen ermöglichen eine direkte Konstruktion, gehen aber von unterschiedlichen Voraussetzungen aus. Während sich beim outside-in Ansatz die Konstruktion nach der Systemspezifikation richtet und die Eigenheiten der verwendeten abstrakten Maschine (etwa einer Programmiersprache) vernachlässigt, sind diese Ausgangspunkt beim inside-out Ansatz, der dafür nicht auf die Eigenheiten der Spezifikation eingeht.

Diese Diskrepanz zwischen Ausgangspunkt und Ziel läßt sich zwar nicht vermeiden, aber durch eine geschickte Verknüpfung beider Ansätze an die Stelle im Konstruktionsweg verschieben, an der man die meisten Konzessionen machen kann.

2.3 Implementation: Concurrent Pascal

Es wird nur auf die wichtigsten Merkmale der Sprache eingegangen. Einzelheiten entnehme man der angegebenen Literatur.

Concurrent Pascal ist eine Modifikation der High-level Sprache Pascal ([JeWi75]) zur sicheren Programmierung nebenläufiger Algorithmen. Die Sicherheit wird durch weitgehende Kontrollmöglichkeiten über die Zugriffsrechte der einzelnen Programmteile zur Kompilationszeit erreicht. Diese Kontrollmöglichkeiten erwachsen aus der Bildung von Zugriffshierarchien aus Variablen (Objekten, instances) spezieller Datentypen (Systemtypen, system types), die mit dem für Pascal entwickelten Verfahren nach bestimmten Mustern definiert werden.

Es gibt drei solcher Muster, die alle eine ähnliche Struktur tragen: PROCESS, MONITOR, und CLASS. Sie setzen sich zusammen aus

a) einem Satz von Zugriffsrechten (access rights), in dem (als formale Parameter) andere Datenobjekte angegeben sind, auf die ein Objekt des betrachteten Systemtyps (Systemobjekt) zugreifen darf,
b) den im Systemtyp verborgenen geschützten Daten (data),
c) Prozeduren und Funktionen (handles), die die auf den Daten zulässigen Operationen definieren,
d) einer Statementfolge (initial operation), die bei der Initialisierung eines Objektes des betrachteten Systemtyps ausgeführt wird.

	PROCESS	MONITOR	CLASS
access rights	other data types	other data types	other data types
data	private	shared	private
handles	interface procedures and functions	(mutually exclusive synchronizing) operations on monitor data	operations on class data
initial operation	process operation	initial state definition	initial state definition

Der Prozeß (PROCESS):

Objekte dieses Systemtyps sind die aktiven Bestandteile eines Concurrent Pascal Programms. Sie dürfen nicht als Zugriffsparameter spezifiziert werden. Sie definieren die sequentiellen Aktivitäten des nebenläufigen Programms. Die Initialisierung eines PROCESS Objektes bedeutet den Start des Prozesses.

Der Monitor (MONITOR):

Dieser Systemtyp ermöglicht die Kommunikation und Synchronisation der aktiven in sich sequentiellen Programmteile (Prozesse). Seine handles sind unteilbare Operationen (mutually exclusive); auf Benutzung wartenden Prozessen wird nach dem FIFO-Prinzip Zugriff auf ein MONITOR Objekt gewährt. Diese Eigenschaft erlaubt die Definition von mehreren Prozessen gemeinsamen Daten (shared data) als Monitor und damit die Kommunikation zwischen Prozessen. Die Synchronisation von Prozessen geschieht über (nur in einem Monitor erlaubte) Warteschlangenvariablen des Standardtyps QUEUE, die mit Standardoperationen DELAY und CONTINUE manipuliert werden können. Sie ermöglichen die Unterbrechung eines Prozesses, der auf eine von einem anderen Prozeß herbeizuführende Bedingung warten muß, und seine Wiederaufnahme, wenn die Bedingung herbeigeführt ist.
Ein Monitor kann also zwei Arten von scheduling enthalten: Ein short-term scheduling, das sich dem Einfluß des Programmierers entzieht, und ein mid-term scheduling, für das der Programmierer verantwortlich ist.

Die Klasse (CLASS):

Mit diesem Systemtyp können geschützte Datenstrukturen nach dem zur strukturierten Programmierung empfohlenen Muster ([BrHa73]) definiert werden. Eine Klasse kann den Teil sowohl eines Prozesses als auch eines Monitors repräsentieren. Wegen ihrer sequentiellen Natur ist der gegenseitige Ausschluß ihrer handles überflüssig.

Die Definition dieser Systemtypen macht die Kontrolle der Zugriffe auf Datenobjekte möglich. Zugriffsrechte, die nicht von unvorhersagbaren Laufzeitbedingungen abhängen (privater Datenschutz, bestimmte Verklemmungsarten), werden schon in der Kompilationsphase geprüft. Wenn solche Bedingungen Voraussetzung für einen Zugriff sind (shared data, Synchronisation), werden sie vor dem Zugriff während des Programmablaufs überprüft. Die Sicherheit, die Concurrent Pascal bietet, hängt eng mit dem statischen Charakter der Sprache zusammen. Es gibt keine Blockschachtelung. Alle Datenstrukturen existieren vom Start des Programms bis zu seinem Ende, sodaß die Zugriffsrechte übersichtlich beim Programmentwurf festgelegt werden können.

Zur Demonstration der Zugriffsrechte der verschiedenen Systemobjekte eines Concurrent Pascal Programms wird eine in [BrHa75b] vorgeschlagene Darstellung, der Zugriffsgraph (access graph), verwendet. Ein Pfeil versinnbildlicht die Relation "hat Zugriff auf" zwischen zwei Systemobjekten, die durch mit der entsprechenden Typkennung versehene Kreise dargestellt werden.

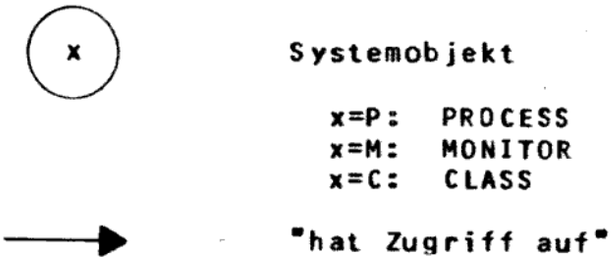

Abb. 2.3.a Elemente des Zugriffsgraphen

Das Verbot von Vorwärtsreferenzen in Concurrent Pascal bewirkt, daß die Relation "hat Zugriff auf" auf der Menge der Systemtypen eines Concurrent Pascal Programms eine Hierarchie definiert:

1. Die Ebene 0 ist die Menge der Systemtypen S, die auf keine weiteren Systemtypen zugreifen.

2. Die Ebene i besteht aus Systemtypen S, für die gilt:
   a) Es gibt einen Systemtyp auf der Ebene i-1, auf den S zugreift, und
   b) alle Systemtypen, auf die S zugreift, liegen auf Ebenen j mit j<i.

Folglich liegen die aktiven Komponenten des Programms, die PROCESS Typen, auf der höchsten Ebene der Hierarchie, während sich die passiven Komponenten, die MONITOR und CLASS Typen, auf die anderen Ebenen verteilen.

Die hierarchische Struktur besagt, daß eine Ebene nur Annahmen über niedrigere Ebenen macht. Für die Korrektheit einer Ebene ist also nur die Korrektheit aller niedrigeren Ebenen Voraussetzung. Fehler in höheren Ebenen können sich nicht in eine einmal korrekt implementierte niedrigere Ebene hinein fortpflanzen. Die Relation "hat Zugriff auf" entspricht somit der Relation "benutzt" auf der Menge der Systemtypen (vgl. Kap. 2.2).

Abb. 2.3.b zeigt als Beispiel den Zugriffsgraphen eines Systems von zwei über einen Monitor A kommunizierenden Prozessen B und C. Die verschiedenen Ebenen werden als übereinanderliegende Schichten von Systemobjekten dargestellt. Man beachte, daß niedrigere Ebenen in der Zugriffshierarchie höheren Ebenen im Zugriffsgraphen entsprechen. Da die Zugriffshierarchie auf der Menge der Systemtypen definiert ist, liegen alle Objekte ein und desselben Systemtyps im Zugriffsgraphen auf derselben Ebene.

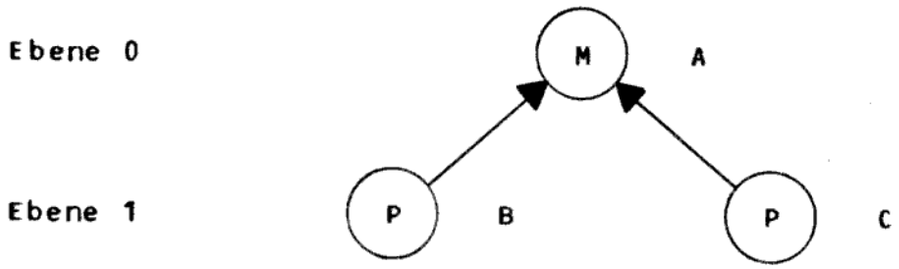

Abb. 2.3.b Beispiel eines Zugriffsgraphen

High-level Sprachen sollen die logische Struktur eines Problems wiederspiegeln, indem sie helfen, von für diese Struktur unbedeutenden Teilproblemen zu abstrahieren. Concurrent Pascal abstrahiert von der Implementation der Ein/Ausgabe und der Synchronisation nebenläufiger Programmteile. Sie wird in einem unterliegenden run-time System (Kernel) festgelegt, das möglichst wenig Änderungen unterworfen sein sollte, wenn man effektiv strukturiert programmieren will. Für die Entwicklung von COCO lag das im California Institute of Technology entwickelte run-time System für die PDP-11/45 zugrunde ([BrHa75a]).

2.4 Verifikation: Methoden der Korrektheitsargumentation

Man kann die Funktionsfähigkeit von Programmcode auf verschiedene Weise begründen. Das Verhalten des COCO Systems ist auf zwei Arten untersucht worden:

1.) Systematische Tests

Die herkömmliche Methode ist, die Auswirkungen des zu prüfenden Codes unter bestimmten Bedingungen zu testen. Die Bedeutung der Testergebnisse hängt dabei entscheidend von der Wahl der Testbedingungen ab. Auf diese Weise lassen sich zwar Fehler eliminieren. Eine positive Aussage über die Korrektheit des Codes kann jedoch nicht gemacht werden, solange nicht sichergestellt ist, daß alle im Einsatzfall möglichen Situationen im Test simuliert worden sind. In vielen Fällen ist gerade bei Betriebssystemen diese Garantie aber nicht gegeben (z. B. bei Zeitabhängigkeiten). Bei Concurrent Pascal Programmen ist die Wahl geeigneter Testbedingungen gleichzusetzen mit der sinnvollen Simulation der auf den zu testenden Systemtyp zugreifenden Systemtypen.

Ein Beispiel für die Verifikation eines Concurrent Pascal Betriebssystems durch systematische Test findet sich in [BrHa75d].

2.) Formale Verifikation

Dieses von Hoare ([Hoa69]) vorgeschlagene Verfahren soll zu einem Überblick über sämtliche denkbaren Verhaltensweisen des Programmcodes verhelfen. Das Prinzip ist, sich über die Auswirkungen jedes einzelnen Befehls und damit über das Zusammenwirken aller Befehle klarzuwerden. Die Aussagen, die interessieren, sind Zustände von Variablen vor bzw. nach der Ausführung eines Codestücks (Eingangs- bzw. Ausgangsbedingungen, pre- bzw. postconditions). Sie entsprechen den Testbedingungen bzw. den Testergebnissen. Analog ist für sinnvolle Ausgangsbedingungen die geeignete Wahl von Eingangsbedingungen erforderlich. Sie entscheidet, welche Eigenschaft des Codestücks bewiesen wird.

Der Korrektheitsbeweis, dh. die Überführung der Eingangs- in die Ausgangsbedingungen erfolgt durch schrittweise Überführung in Zwischenbedingungen. Dazu müssen Codeeinheiten (Statements) festgelegt und für sie axiomatisch Überführungsregeln formuliert werden, deren Gültigkeit vorausgesetzt ist. Im Beweis werden die Bedingungen in Kommentarklammern eingerahmt vor bzw. hinter dem Statement aufgeschrieben:

"Eingangsbedingungen" Statement "Ausgangsbedingungen"

Die Ausgangsbedingungen eines Statements und die Eingangsbedingungen des darauffolgenden Statements im Codestück werden durch die Regeln der Prädikatenlogik verknüpft. Die Überführung einer Bedingung 1 in eine Bedingung 2 durch Implikation (rule of consequence) wird geschrieben:

"Bedingung 1" "Bedingung 2"

(Mitunter werden die Axiome, auf die sich ein Beweisschritt stützt, hinter der resultierenden Bedingung in Klammern angegeben.)

Bei dieser Art des Korrektheitsbeweises können nur Aussagen gemacht werden, solange der Rechner für die Durchführung des verifizierten Programms nur eine begrenzte Zeit benötigt (partielle Korrektheit). Sonst ist die Formulierung von Ausgangsbedingungen sinnlos. Die Frage der Termination des Programms ist zusätzlich zu klären.

In der vorliegenden Arbeit wird Concurrent Pascal Code formal verifiziert. Für jedes Concurrent Pascal Statement wird eine Überführungsregel vorausgesetzt.

Die Regeln für die Pascal Statements sind in [HoWi73] beschrieben und werden hier nicht erläutert.

Die Regeln für die nicht in Pascal enthaltenen Sprachelemente, dh. für die Systemtypen und ihre Benutzung gehen auf Vorschläge von Howard ([How76a, How76b]) nach einer Idee von Hoare ([Hoa74]) zurück.

Will man Code beweisen, der Zugriffe auf derartige Sprachstruk-

turen enthält, so braucht man Eingangs- und Ausgangsbedingungen für jede auf dieser Struktur definierte Operation. Die Überführung der Eingangs- in die Ausgangsbedingungen wird beim Beweis des Codestücks vorausgesetzt und muß in einem gesonderten Korrektheitsbeweis für die Datenstruktur begründet werden.
Bei synchronisierenden Datenstrukturen (Monitoren) müssen zusätzlich auch Beweisregeln für die Synchronisationsoperationen angegeben werden.

Aussagen, die allen Eingangs- und Ausgangsbedingungen gemeinsam sind, bilden eine Invariante. Sie gilt zu jedem Zeitpunkt, zu dem nicht auf die Datenstruktur zugegriffen wird. Die Invariante einer Datenstruktur spiegelt also ihre Natur losgelöst von einer speziellen Verwendung wieder.
Oftmals wird die Natur der Datenstruktur klarer, wenn man Aspekte ihrer zeitlichen Entwicklung hinzunimmt, die nicht in der Implementation zum Ausdruck kommen. In diesem Fall kann man zur Formulierung einer reichhaltigen Invariante die Struktur mit zusätzlichen Variablen (Hilfsvariablen, history variables) anreichern, die diese Entwicklung festhalten. Sie dienen nur zur Charakterisierung der Struktur und brauchen nicht implementiert zu werden.

Zur Charakterisierung der Concurrent Pascal Systemtypen sind Aussagen zu suchen, die den folgenden Beweisregeln genügen:

CLASS:

Sei J eine Aussage über die Klassendaten. Es ist nachzuweisen:

(C1) "true" initial operation "J"
(C2) "J" each class procedure/function "J"

MONITOR:

Concurrent Pascal kennt nur einelementige Prozeßwarteschlangen (Datentyp QUEUE). Es bietet sich an, die von Howard eingeführten CONDITION Variablen ([How76a]) durch ein QUEUE Feld mit von einer Klasse bereitgestellter FIFO-Verwaltung zu simulieren (FIFO CLASS, Kap. 6.1).

Howards Signaloperation (SIGNAL) unterscheidet sich in zwei Punkten semantisch von Concurrent Pascal (CONTINUE):
a) Sie darf nur bei nichtleerer Warteschlange ausgeführt werden.
b) Sie erzwingt keinen Rücksprung aus dem Monitor.
Wird Howards Semantik im Sinne von Concurrent Pascal um einen Monitorrücksprung erweitert, so entsprechen sich folgende Notationen:

Howards Monitor	Concurrent Pascal Monitor	
	einelementige	mehrelementige Warteschlange
Q: CONDITION;	Q: QUEUE;	Q: ARRAY (.1..N.) OF QUEUE; NEXT: FIFO;
Q.WAIT;	DELAY(Q);	DELAY(Q(.NEXT.ARRIVAL.));
IF Q.QUEUE THEN Q.SIGNAL;	CONTINUE(Q);	IF NOT NEXT.EMPTY THEN CONTINUE(Q(.NEXT.DEPARTURE.));

Wegen ihrer größeren Klarheit wird in dieser Arbeit auf die Notation von Howard zurückgegriffen.

Sei Q eine Prozeßwarteschlange mit FIFO-Auswahl (Datentyp CONDITION), B(Q) für nichtleere Warteschlangen die Bedingung, auf die die in Q eingetragenen Prozesse warten, J eine Aussage über die Monitordaten und E eine Aussage, die J zu J&E => ¬B(Q) für alle nichtleeren Warteschlangen Q des Monitors ergänzt. Unter der Voraussetzung, daß für alle Q gilt:

(M1) "J & E" Q.WAIT "J & B(Q)"
(M2) "J & B(Q)" Q.SIGNAL "false"

ist nachzuweisen:

(M3) "true" initial operation "J & E"
(M4) "J & E" each monitor procedure/function "J & E"

Wie in Kap. 2.3 ausgeführt, gestattet die hierarchische Struktur der Zugriffsbeziehungen in Concurrent Pascal Programmen eine schrittweise Verifikation der einzelnen durch die Relation "hat Zugriff auf" untereinander in Beziehung stehenden Systemtypen ausgehend von der untersten Ebene.

3 Spezifikation

3.1 Hardware-Konfiguration

Zugrunde liegt ein sternförmiges Kommunikationsnetz, bestehend aus n sende- und empfangsfähigen DTEs und einem sie verbindenden Knotenrechner. Der individuelle Charakter der einzelnen DTEs ist nicht festgelegt. Sie können Rechner, Experimentstationen, Terminals, Speichergeräte und vieles mehr sein. Die Kanäle sind Vollduplex-Leitungen mit getrennten Ein- und Ausgabeinterfaces, die beide den Prozessor des Knotenrechners unterbrechen können.

3.2 Kommunikationsdaten

Das System arbeitet mit einem eigenen Datenformat (Block). Nachrichten, dh. die Dateneinheiten der kommunizierenden DTEs können aus mehreren Blöcken bestehen. Diese höhere Struktur ist für das Knotenrechnersystem jedoch transparent. Über die Leitungen übertragene Daten brauchen nicht dem Systemformat zu entsprechen. Sie werden nach dem Empfang bzw. vor der Ausgabe (durch Hardware oder Software) angepaßt.

Für die Kommunikationsdaten können spezielle Behandlungen gewünscht werden (etwa Code- oder Formatumwandlungen), die vor ihrer Ausgabe vom System in festgelegter Reihenfolge nacheinander vorgenommen werden. Eine durch eine Datenbehandlung verursachte Expansion eines Datenblocks über eine Maximallänge hinaus ist nicht vorgesehen.

Die Blöcke bekommen beim Eintreffen im System einen Speicherplatz (frame) zugewiesen, den sie bis zum Verlassen des Systems behalten. Vom System selbst können keine Blöcke generiert, jedoch eingegebene Blöcke als Träger von Systemnachrichten benutzt werden.

Zur Übertragung der Daten ist eine routing Information notwendig, die den Daten vorangeht. Sie wird teilweise vom Sender und teilweise vom System spezifiziert und ist zum Teil für das System und zum Teil für den Empfänger gedacht. Die eigentlichen Daten sind im System nur für die Datenbehandlung relevant und für den Rest transparent.

3.3 Scheduling Strategie

Die Hauptaufgabe des Systems besteht in einer sich nach der Dringlichkeit der Kommunikationsdaten richtenden und durch die Auslastung der Kanäle (dh. interrupt-) gesteuerten Datenübertragung. Diese Spezifikation erfordert die Lösung zweier Probleme:

1.) Die dynamische Abarbeitung event-gesteuert eintreffender Daten:

Die Lösung dieses Problems lehnt sich an ein in [BaSch74] beschriebenes Konzept an.

Jeder Datenblock bringt beim Eingang in das System eine feste Blockpriorität mit, die seine Dringlichkeit definiert. Sie kann entweder vom Sender-DTE oder bei der Anpassung an das Systemformat festgelegt werden. Sie sollte aber im Anwendungsfall nicht vom Benutzer des DTE spezifiziert werden können, da sich die im folgenden beschriebene Strategie auf eine faire Verteilung der Blockprioritäten stützt. Die eintreffenden Blöcke werden gemäß ihrer Dringlichkeit in einer Warteschlange gepuffert (für Blöcke gleicher Priorität gilt das FIFO-Prinzip). Die Datenbehandlungs- und Ausgabeaktionen werden für den jeweils dringendsten Block im System durchgeführt.

Diese Konzeption gestattet die Berücksichtigung der aktuellen Dringlichkeitsverteilung bei der Abarbeitung der Daten, obwohl durch die event-gesteuerte Ein/Ausgabe diese Verteilung zeitlich nicht vorhersagbar ist.

2.) Die Vermeidung von starving:

Bei der in 1.) beschriebenen Strategie kann nicht garantiert werden, daß jeder ins System kommende Block das System nach endlicher Zeit wieder verläßt. Blöcke geringerer Dringlichkeit können ewig gezwungen sein, vor der Abfertigung dringenderer Blöcke zurückzutreten. Dieses Phänomen heißt starving (Verhungern).

Um es zu vermeiden, müssen die Blöcke einem "Alterungs"prozeß unterliegen, der ihre Dringlichkeit während ihres Aufenthaltes im System in bestimmten Zeitabständen erhöht. Am effektivsten geschieht das durch Inkrementieren einer dem System eigenen dynamischen Priorität. Die Systempriorität eines Blocks, die seine Dringlichkeit definiert, setzt sich dann aus seiner Blockpriorität plus der im Augenblick seines Eintreffens im System gegebenen dynamischen Priorität zusammen. Wenn man fordert, daß die Dringlichkeit der Blöcke nach oben beschränkt ist, ist starving ausgeschlossen.

Die hohe Effizienz dieses Verfahrens resultiert daraus, daß man die Dringlichkeit eines Blocks als Summe eines für jeden Block individuellen und eines allen Blöcken gemeinsamen Wertes darstellt. Durch eine Veränderung des allen Blöcken gemeinsamen Wertes erreicht man über eine einzige Größe das Altern sämtlicher Blöcke im System.

Die Funktionsweise der Strategie wird am ehesten klar, wenn man sich die Prioritäten als Zeitaussagen vorstellt. Diese Interpretation macht das beschriebene System zu einem Kommunikationssystem, das Realzeitanforderungen berücksichtigt.

Die Blockpriorität ist als Forderung zu verstehen, wie lange der Datenblock im System maximal bis zu seiner Bearbeitung warten

sollte. Dementsprechend ist ihr niedrigster Wert sinnvollerweise Null (niedrige Werte bedeuten hohe Dringlichkeit). Er besagt, daß der Block möglichst ohne Verzug behandelt werden soll. Die dynamische Blockpriorität repräsentiert die Realzeit. Damit stellt die Systempriorität eines Blocks als Summe aus der Realzeit (dynamische Priorität) und seiner maximalen Wartezeit (Blockpriorität) den Zeitpunkt dar, zu dem der Block spätestens bearbeitet werden sollte. Dieser Wert gibt die Dringlichkeit des Blocks an. Wenn die Realzeit ihn erreicht hat, ist ein starving des betreffenden Blocks unmöglich.

Es muß hervorgehoben werden, daß diese Strategie eine Durchsetzung der Realzeitforderungen nicht garantiert. Sie sichert zwar, daß jeder Block in endlicher Zeit das System passiert, über den genauen Zeitpunkt seiner Bearbeitung kann aber keine Aussage gemacht werden. Insofern hat die Realzeitinterpretation ihre Grenzen.

Beispiel:

> Beim Eintreffen eines Datenblocks mit der Blockpriorität 10 habe die dynamische Priorität den Wert 5000. Die Systempriorität des Blocks ist dann 5010.
> Interpretation: Der Block sollte möglichst spätestens nach 10 Zeiteinheiten, also zur Zeit 5010 bearbeitet werden. Nach dem Zeitpunkt 5010 können keine neu eintreffenden Datenblöcke mehr diesem Block vorgezogen werden. Trotzdem können noch viele dringendere Blöcke vor ihm bearbeitet werden. Der genaue Zeitpunkt der Bearbeitung des betrachteten Blocks steht also nicht fest.

Die Tatsache, daß in einem Rechner nur endlich viele Zahlen dargestellt werden können, mag eine zyklische Struktur der dynamischen Priorität nahelegen. Eine solche Konzeption hätte aber Ineffizienzen zur Folge, da sie einen Sprung in der für den zentralen Alterungsvorgang wesentlichen linearen Ordnung erzwingt. Unter der Voraussetzung, daß das Concurrent Pascal run-time System genügend unterschiedliche positive Prioritätswerte für einen hinreichend langen Systemlauf darstellen kann und daß eine im Vergleich dazu verschwindend geringe obere Schranke der Blockprioritäten existiert, ist eine zyklische Struktur überflüssig.

3.4 Systemfunktion

Die Funktion des Systems wird am Beispiel der Übertragung eines Blocks erläutert (vgl. Abb. 3.4.a):

Die Eingabe der Daten in den Knotenrechner erfolgt per Interrupt in einem von der Kanalart bestimmten Format. Die eingetroffenen Daten werden zur Einheit des Systems, dem Block, zusammengefaßt (Datenanpassung). Der Block erhält einen Speicherplatz (frame) und seine Systempriorität, wird in die Warteschlange eingereiht und wartet auf seine Bearbeitung (Datenbehandlung, falls gewünscht, bzw. Ausgabe). Sobald alle dringenderen Blöcke im System abgefertigt sind, ist er an der Reihe. Im Fall einer Datenbehandlung werden die vorher im frame stehenden durch die umgewandelten Daten ersetzt. Die verschie-

denen Datenbehandlungen für einen Block werden sequentiell in festgelegter Reihenfolge durchgeführt. Nach der letzten Behandlung wird der aufbereitete Block zur Ausgabe eingereiht. Wenn er im Format des entsprechenden Kanals ausgegeben ist, wird sein frame zur Aufnahme eines neu eingehenden Blocks freigegeben.

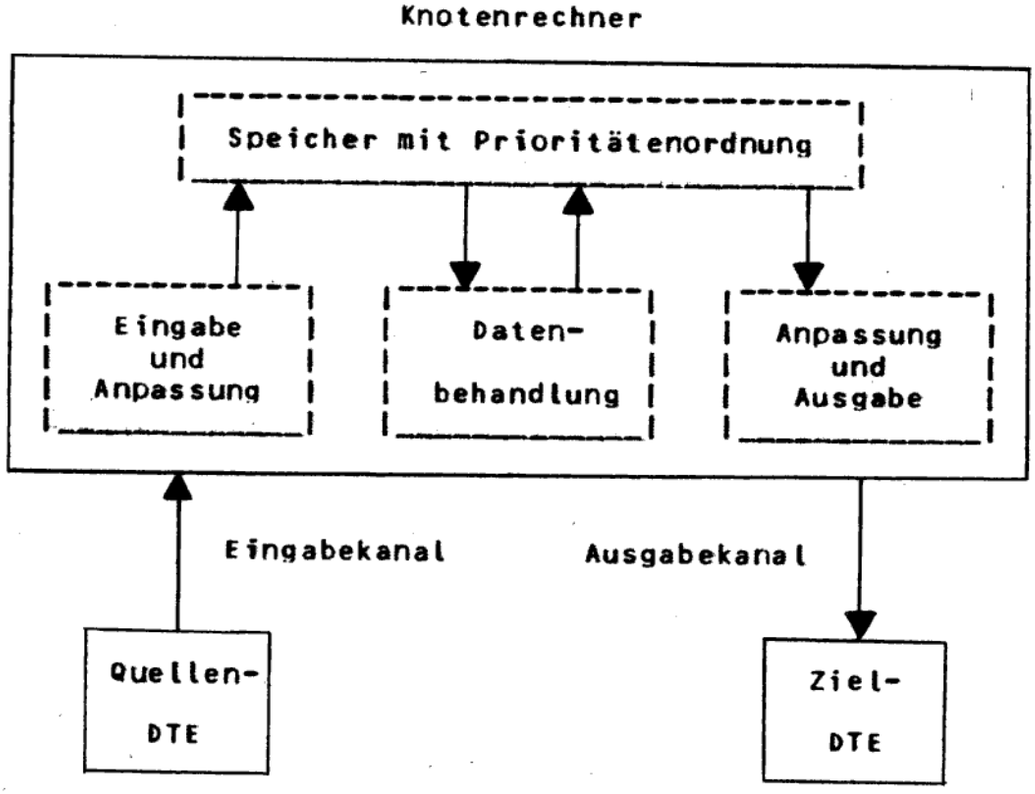

Abb. 3.4.a Kommunikationssystem

4 Design und Implementation

Wegen ihrer gegenseitigen Abhängigkeit ist den beiden Entwicklungsphasen Design und Implementation ein gemeinsamer Abschnitt gewidmet. Die hinzugefügten Bemerkungen fassen die gewonnenen Erkenntnisse über die verwendeten Designkonzepte (siehe Kap. 2.2) und die Sprache Concurrent Pascal (siehe Kap. 2.3) zusammen. Der Programmtext des COCO Systems ist im Anhang (Kap. 6.1) aufgeführt.

4.1 Datenformat des Systems

Ein Block besteht aus zwei Teilen: dem Kopf, der die routing Informationen enthält, und den Daten.

Der Kopf (Blockheader) enthält die folgenden Angaben:

a) Blocktyp

Status des Blocks im System (wird vom System spezifiziert). Es gibt fünf Blocktypen:
TRAFFIC: auf (bisher) reibungsloser Durchreise
SYSTEM: enthält Systemnachricht
LOGIC: logischer Fehler aufgetreten
DATA: Datenfehler aufgetreten
CHANNEL: Kanalfehler aufgetreten
Diese Angabe dient dem Empfänger als Information über den Ursprung des Blocks.

b) Quellen- und Zielinformation

Sender- und Empfängerleitung sowie bei entsprechendem DTE Sender- und Empfängerprozeß.

c) Blockname

Für den Empfänger bestimmte Blockkennung.

d) Blockpriorität

Eingangspriorität des Blocks. Kann vom Sender oder vom Empfangsteil des Systems spezifiziert werden.

e) Systempriorität

Die Dringlichkeit des Blocks definierende Priorität. Wird vom System spezifiziert.

f) Datenbehandlungen

Angabe über die vor der Ausgabe des Blocks notwendigen Datenmanipulationen. Es ist nur eine Pseudobehandlung implementiert.

g) Fehlertyp

Definiert die Behandlungsweise des Blocks bei Auftritt eines Fehlers während des Transfers. Es werden drei Fehlertypen unterschieden:
FORGET: Der Block wird vergessen.
SAVE: Der Block wird an den Sender zurückgeschickt. (Der Grund ist aus dem Blocktyp ersichtlich.)
MESSAGE: Der Datenteil des Blocks wird durch eine detaillierte Systemnachricht ersetzt, die an den Sender ergeht.

h) Blocklänge

Aktuelle Länge des Datenteils.

Der Datenteil darf eine maximale Länge nicht überschreiten. Er ist für das gesamte System außer für die Datenbehandlung transparent.

4.2 Transferprozesse

Design

Man sollte sich genau überlegen, in wieviele Prozesse man ein Betriebssystem zerlegt. Beim Aufbau einer nebenläufigen Struktur ist durchaus Zurückhaltung angebracht.
Prozesse heißen nebenläufig, wenn sie oder Teile von ihnen logisch voneinander unabhängig sind, dh. parallel ablaufen können. In vielen Fällen ermöglicht jedoch die Hardware ein solches Verhalten nur in eingeschränktem Sinn oder gar überhaupt nicht.

Es gibt zwei Arten, Nebenläufigkeit hardwaremäßig zu unterstützen:

1.) Ein unterbrechbarer Prozessor

ermöglicht "Pseudoparallelität". Zu jedem Zeitpunkt ist nur ein Prozeß aktiv, es ist aber nicht festgelegt, welcher. Die Synchronisation der wechselseitigen Aktivitäten wird von der Hardware mit Hilfe von Interrupts ermöglicht und bei Bedarf nach einer gewünschten durch Software realisierten Strategie durchgeführt.

2.) Mehrere Prozessoren

ermöglichen das echt parallele Ablaufen von Prozessen. Die Notwendigkeit der Unterstützung durch die Hardware bei der Synchronisation entfällt.

Ideale Voraussetzungen bieten mehrere unterbrechbare Prozessoren. Sie gestatten eine optimale Abwicklung nebenläufiger Aktivitäten durch echte Parallelität und die Vermeidung von busy waiting.

Die nebenläufige Struktur eines Betriebssystems sollte genau auf die Hardware-Gegebenheiten abgestimmt sein. Für eine optimale Geräteaus-

nutzung ist folgende Faustregel empfehlenswert:

Genau jeder Prozessor und jedes interruptfähige Gerät sollte durch einen eigenen Prozeß bedient werden. Weniger Prozesse nutzen die Möglichkeiten zur Nebenläufigkeit, die die Hardware bietet, nicht voll aus. Mehr Prozesse belasten die Software mit unnötigen scheduling Aufgaben von ohnehin sequentiell ablaufenden Aktivitäten.

Diese Regel wird auf das vorliegende System angewandt:

Zu betrachten sind Transferaktivitäten, also solche, die mit der Übertragung von Kommunikationsdaten zu tun haben. Sie lassen sich entsprechend der Systemspezifikation in drei Gruppen klassifizieren:

a) Eingabeaktivitäten
b) Datenbehandlungsaktivitäten
c) Ausgabeaktivitäten

Man kommt zur folgenden Einteilung der Prozesse:

Da jede Ein- und Ausgabeleitung die Fähigkeit zum Interrupt besitzt, gibt es für jeden Kanal einen Eingabeprozeß und einen Ausgabeprozeß. Für den (einzigen) Prozessor bleibt ein Prozeß, der alle Datenbehandlungsaktivitäten enthält.

Implementation

Der dem Prozessor zugeteilte Prozeß ist

der DATAPROCESS PROCESS.

Es ist nur die Rahmenstruktur ohne Datenbehandlungen implementiert.

Jeder Kanaltyp benötigt seine eigene Prozeßspezifikation, jeder Ein- und Ausgabekanal sein eigenes Objekt des entsprechenden Prozeßtyps. In den Ein- und Ausgabeprozessen sind alle kanalspezifischen Handlungen (etwa Datenanpassungen zwischen dem Kanal- und dem Systemformat) definiert. Allen Kanälen gemeinsame Ein/Ausgabeaktionen sind in einer die Prozesse unterstützenden Klasse untergebracht. Dadurch wird die Wartung der Ein/Ausgabelogik bei Variationen der Netzkonfiguration auf ausschließlich notwendige Arbeiten beschränkt.

Diese Schnittstelle zwischen dem kanalabhängigen und dem nicht kanalabhängigen Teil des Übertragungssystems ist

die SYSTEMGUARD CLASS.

Sie enthält alle systemweiten Handlungen, die ein Ein/Ausgabeprozeß durchführen muß.
Die Prozedur

ENTER (VAR ELEMENT: BLOCK; LINE: COMPONENTS)

bringt einen Datenblock ins System ein. Wenn die Zielleitung funk-

tionsfähig ist, wird anhand der Blockpriorität und der augenblicklichen dynamischen Priorität seine Systempriorität festgelegt und der Block abgespeichert.
Die Prozedur

CHECK (VAR ELEMENT: BLOCK; SUCCESS: BOOLEAN)

überprüft den Erfolg der Ausgabe eines Blocks und führt Buch über die Funktionsfähigkeit der Ausgabeleitungen.
Beide Prozeduren übergeben bei Erkennung eines Fehlers den Block an das Fehlerbehandlungssystem.

Für folgende DTEs sind Ein/Ausgabeprozesse implementiert:

a) Terminal:

Der TERMINALINPUT PROCESS

nimmt an einem Terminal eingegebene Meldungen entgegen. Der Benutzer des Terminals kann eine Eingabe per Steuerzeichen (ctrl g) anfordern und seinen Zielkanal spezifizieren. Diesem wird zunächst ein Block zugesandt, der als Daten den Quellenkanal enthält. Terminaldaten erhalten wegen ihrer geringen Sendefrequenz höchste Priorität. (Durch die FIFO-Strategie bei der Behandlung gleich priorisierter Blöcke bleibt die Eingabereihenfolge auf jeden Fall erhalten.) Die Meldungen werden zeilenweise übertragen. Das Ende der Eingabe deutet ein einfaches carriage return Kommando an.

Der TERMINALOUTPUT PROCESS

gibt für seinen Kanal bestimmte Blöcke aus. Eine Fehlerkontrolle erfolgt nicht.

Da die Blockdateneinheit nicht mit der Charactereinheit in Concurrent Pascal übereinstimmt, muß ein TERMINALDRIVER eine Anpassung vornehmen. Auf eine nähere Beschreibung wird an dieser Stelle verzichtet (siehe Kap. 6.1).

b) PDP-11 Computer:

Der PDPINPUT PROCESS

übernimmt vom Kernel einen Block im Systemformat und bringt ihn in das System ein.

Der PDPOUTPUT PROCESS

übernimmt vom System einen Block und übergibt ihn dem Kernel zur Ausgabe. Anschließend vergewissert er sich über ihren Erfolg.

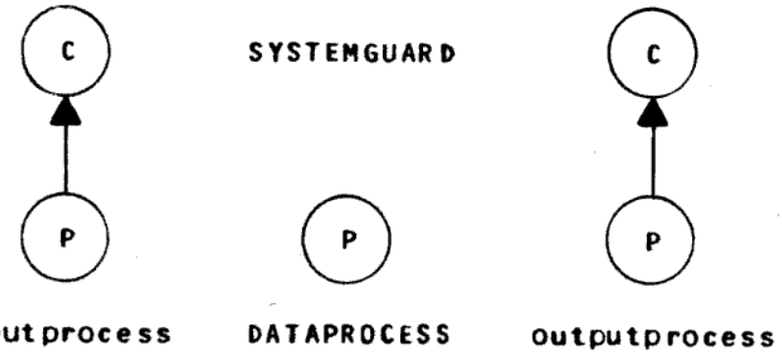

Abb. 4.2.a Transferprozesse

Bemerkungen

1.) Der Datenbehandlungsprozeß ist ein Beispiel für den sinnvollen Verzicht auf Nebenläufigkeit:

Die Eigenständigkeit der verschiedenen Datenbehandlungen für einen Block könnte dazu verleiten, sie als individuelle Prozesse zu definieren und ihr Zusammenwirken von einem Monitor überwachen zu lassen. Da die verschiedenen Behandlungen (etwa Umcodierung und Umformatierung) als in bestimmter Reihenfolge sequentiell ablaufend spezifiziert wurden, genügt es jedoch, die einzelnen Aktivitäten als Prozeduren zu definieren. Ihre Reihenfolge, sozusagen das in die Phase des Programmentwurfs vorgezogene scheduling, übernimmt ein umrahmender Prozeß.

Es werden die unterschiedlichen Aufgaben der Systemtypen MONITOR und PROCESS deutlich:

Der MONITOR ist nicht einfach ein Mittel zur Kommunikation zwischen eigenständigen Aktivitäten. Er wird erst in dem Moment notwendig, wenn Nebenläufigkeit (dh. gegenseitige Unterbrechbarkeit oder Parallelität) ins Spiel kommt. Aktivitäten, die nacheinander ablaufen müssen, können effektiver als Prozeduren in dem sequentiellen Systemtyp PROCESS zusammengefaßt werden.

2.) Ein Beispiel für einen nicht sinnvollen Verzicht auf Nebenläufigkeit:

Würden mehrere Eingabekanäle von einem gemeinsamen Prozeß bedient, so würde die Realzeitsteuerung der (pseudo-) parallelen Eingabeaktivitäten zugunsten einer von der Software oktroyierten Reihenfolge verloren gehen. Unter der Voraussetzung, daß das run-time System die Nebenläufigkeit der Kanalübertragungen unterstützt und für jeden Kanal eine individuelle Schnittstelle zur Sprache (IO call) vorsieht, hätte ein ruhender Kanal sogar katastrophale Folgen. Denn der Prozeß würde vom ihm erst ablassen, nachdem er einmal aktiviert wurde, und hätte keine Möglichkeit, ihn als ruhend zu erkennen und zu übergehen.

3.) Das zugrundeliegende run-time System ist nicht für die Programmierung von Kommunikationssystemen geeignet. Es unterstützt keine Interfaces für Datenleitungen. Neben der Definition der Prozeßtypen muß für eine Implementation von COCO der Kernel geändert werden.

Dem Implementator stellt sich die Frage, auf welcher Abstraktionsebene er die Schnittstelle zwischen dem Kernel und Concurrent Pascal ansiedeln soll. Je intelligenter das unterliegende run-time System ist, desto effektiver aber auch unflexibler kann programmiert werden.

Bei der Erweiterung des Kernels für eine Programmierung von Kommunikationssystemen muß für jeden Kanaltyp individuell entschieden werden, welchen Teil der kanalspezifischen Handlungen der Kernel übernehmen soll. Dieser Teil definiert die statischen Eigenschaften des betreffenden Kanaltyps. Eigenschaften, die Änderungen unterliegen könnten, sollten in Concurrent Pascal bestimmt werden.

Die Schnittstellen zwischen dem run-time System und der Sprache sind bei Concurrent Pascal zu verwaschen! Neben den üblichen Standardfunktionen (IO, DELAY, CONTINUE, usw.) gibt es kernelabhängige Typdefinitionen (I/O types, usw.), was zu erheblicher Verwirrung führen kann.

4.) Die Syntaxregeln für die Konstantendefinition und im Gegensatz zu Pascal auch für die Darstellung von SET Konstanten entsprechen in Concurrent Pascal nicht den Prinzipien einer strukturierenden Sprache: Neudefinitionen von Programmkonstanten nötigen mitunter zu Änderungen an mehreren Stellen des Programms. Zum Beispiel verlangt die Neudefinition der Kanalmenge (Typ IODEVICE, Untertyp COMPONENTS) oder der Datenbehandlungsmenge (Typ DATAHANDLERS) zusätzlich eine Änderung der Abfragen in der SYSTEMGUARD CLASS. Bei Zahlenkonstanten sollten arithmetische Ausdrücke erlaubt sein, bei SETs die Darstellung einer Aufzählung (enumeration) durch das erste und letzte Element. Dadurch würde eine zentrale Angabe dieser Größen und damit eine größere Flexibilität der Programme gegenüber Änderungen der Spezifikation (Adaptivität) möglich.

4.3 Speicherstruktur

Design

In diesem Kapitel wird mit einem inside-out Ansatz eine dem spezifizierten Prioritätenkonzept genügende Speicherstruktur entwickelt. Ausgehend von der Systemspezifikation gelangt man durch schrittweise Konkretisierung von abstrakten Forderungen zu der Implementation, einer Hierarchie von Systemtyp-Definitionen.
Der Ausgangspunkt des Designprozesses ist eine Warteschlange, in der sich nach ihrer Systempriorität geordnet die "durchreisenden" Datenblöcke befinden. Im Sinne des inside-out Ansatzes ist zunächst ein

Blockspeicher zu schaffen und diesem dann die gewünschte Ordnungsstruktur aufzuprägen.

1.) Der Blockspeicher

Die Datenblöcke sind für alle Transferprozesse in einem gemeinsamen Speicher (shared buffer) zugreifbar. Die Operationen auf einem solchen Speicher teilen sich in zwei Gruppen:

a) Speicherzugriff

Zugriffs- (Kopier-) Operationen, die jeweils nur ein Speicherelement, also eine Unterstruktur des Speichers bearbeiten.

b) Zugriffsverwaltung

Operationen, die den Status des Speichers definieren, also den Speicher als strukturloses Objekt bearbeiten.

Die Speicherverwaltung muß in jedem Fall vor parallelem Zugriff geschützt, dh. als kritischer Abschnitt implementiert werden (short-term scheduling) und die vom Speicherstatus abhängenden scheduling Entscheidungen (Speicher voll/leer) enthalten (mid-term scheduling). Ist jedoch garantiert, daß der Zugriff auf ein und dasselbe Speicherelement sequentiell erfolgt, so brauchen die Kopieroperationen selbst nicht unteilbar zu sein. Dies trifft z. B. für einen mailbox Speicher zu, auf den nur über die Speicherverwaltung zugegriffen werden darf.

Die in der Spezifikation von COCO festgelegte streng sequentielle Abhandlung des Blocktransfers legt nahe, auch Speicherzugriffe ohne Umweg über die Zugriffsverwaltung zuzulassen. Der Status des Speichers ändert sich nämlich ausschließlich bei der Ein- oder Ausgabe eines Blocks (vgl. Kap. 3.2 und Kap. 3.4). Eine solche Konzeption funktioniert aber nur, wenn eine sinnvolle Koordination von Speicherzugriff und Zugriffsverwaltung sichergestellt ist. Es darf nur auf Speicherelemente zugegriffen werden, die von der Verwaltung als zugreifbar deklariert sind.

2.) Die Ordnungsstruktur

Eine Ordnungsstruktur auf einem Speicher wird durch eine Speicherliste dargestellt, die seine Elemente zueinander in Beziehung setzt. Im vorliegenden Fall ist eine einfach verzeigerte Liste von nach Prioritäten geordneten Elementen (Blöcken) geeignet, wobei das Element mit der größten Dringlichkeit den Listenkopf bildet. Um die Listenoperationen (Einfügen und Herausnehmen von Elementen) möglichst effektiv zu gestalten, dh. unnötiges Suchen (scanning) zu vermeiden, sollten bei jedem Zugriff nur relevante Listenteile bearbeitet werden. Dies wird durch eine Gliederung in Teillisten erreicht, die alle Blöcke mit demselben Ziel zusammenfassen. Aus jeder Teilliste nimmt also genau ein Prozeß Elemente heraus.

Die Speicherliste setzt sich demnach aus einer einzigen Eingabeliste sowie je einer Ausgabeliste für jeden Ausgabeprozeß (dh. für jeden Kanal) zusammen (Abb. 4.3.a). Um die Anpassungsfähigkeit des Systems an unterschiedliche Belastungen einzelner Kanäle zu erhöhen, bedienen sich alle Teillisten eines gemeinsamen Elementepools, wobei jede Teilliste jedes Element für sich beanspruchen darf, solange es nicht in einer anderen steht. Das Einfügen und Herausnehmen von Elementen muß als Operation auf einer gemeinsamen Datenstruktur unteilbar sein.

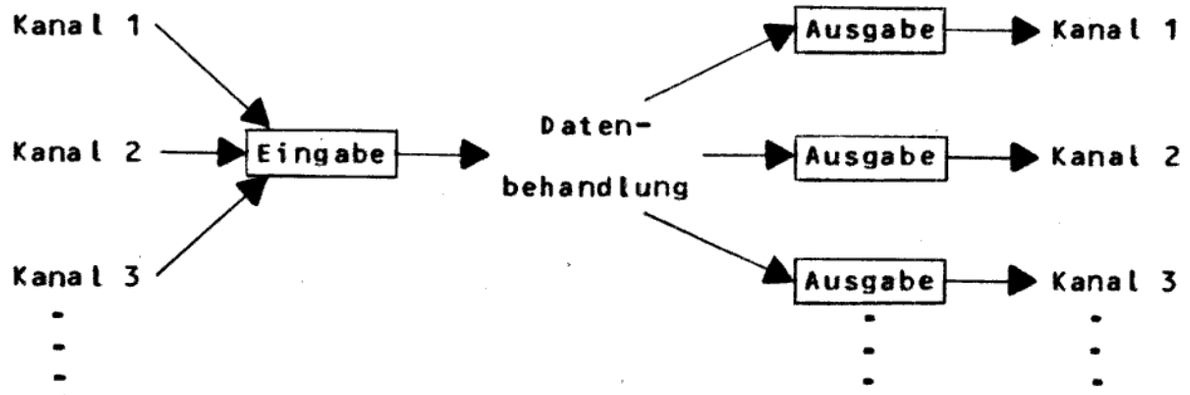

Abb. 4.3.a Struktur der Speicherliste

Implementation

1.) Der Blockspeicher

Der Blockspeicher besteht aus zwei Einheiten:

Der SHAREDBUFFER MONITOR

enthält die Speicherelemente (frame pool) und faßt die darauf zugreifenden Operationen zusammen. Er reflektiert die innere Struktur des Speichers.
Die Kopieroperationen sind

PUT (ELEMENT: BLOCK; ELEMENTINDEX: BUFFERINDEX)

zum Schreiben eines Datenblocks und

GET (VAR ELEMENT: BLOCK; ELEMENTINDEX: BUFFERINDEX)

zum Lesen eines Datenblocks. Sie sollen nur auf frames ausgeführt werden, die

der BUFFERGUARD MONITOR

als zugriffsfähig deklariert hat. Er enthält die den Status des

Speichers definierenden Operationen.
Über die Prozedur

REQUEST (VAR ELEMENTINDEX: BUFFERINDEX)

kann ein frame geschaffen (dh. als zugriffsbereit deklariert) werden, über

RELEASE (ELEMENTINDEX: BUFFERINDEX)

kann ein frame vernichtet (dh. als nicht zugriffsbereit deklariert) werden.
Zwei Funktionen für Testzwecke geben Auskunft über den Zustand des Speichers.

FULL: BOOLEAN

gibt an, ob der Speicher voll ist, dh. keine neuen frames geschaffen werden können,

EMPTY: BOOLEAN

gibt an, ob der Speicher leer ist, dh. keine frames vernichtet werden können.
Anfangs ist der Speicher als leer deklariert.

2.) Die Ordnungsstruktur

Der LISTSTRUCTURE MONITOR

gibt dem Blockspeicher seine Ordnungsstruktur. Es wird die modulare Version erläutert (vgl. Kap. 6.1 und Bem. 5).

Die PRIORITYLIST CLASS

stellt die beschriebene Substruktur dar. Sie enthält eine einfach verzeigerte nach Prioritäten geordnete Liste von Speicherindizes, die bei Initialisierung leer ist.
Mit der Prozedur

ENTER (ELEMENTINDEX: BUFFERINDEX; PRIORITY: REAL)

kann ein neuer Index gemäß der Systempriorität des im frame enthaltenen Blocks eingefügt werden, mit

REMOVE (VAR ELEMENTINDEX: BUFFERINDEX)

wird der Index des frames mit dem dringendsten Block herausgenommen.
Eine Funktion

EMPTY: BOOLEAN

gibt Auskunft über den Zustand der Liste.

Die gesamte Ordnungsstruktur besteht aus entsprechenden Objekten

der Substruktur und einem Satz sie sinnvoll manipulierender Prozeduren.

INENTER (ELEMENTINDEX: BUFFERINDEX; PRIORITY: REAL)

fügt einen frame in die Eingabeliste ein,

INREMOVE (VAR ELEMENTINDEX: BUFFERINDEX)

entnimmt der Eingabeliste einen frame,

OUTENTER (ELEMENTINDEX: BUFFERINDEX; PRIORITY: REAL;
LINE: COMPONENTS)

fügt einen frame in die im Aufruf spezifizierte Ausgabeliste ein,

OUTREMOVE (VAR ELEMENTINDEX: BUFFERINDEX; LINE: COMPONENTS)

entnimmt der im Aufruf spezifizierten Ausgabeliste einen frame.

3.) Zugriffssicherheit

Der Verlust an Kontrolle über den Speicherzugriff - der Preis, den man für die Aufteilung der Speicherstruktur zahlen muß - wird durch

die BLOCKSTREAM CLASS

teilweise aufgefangen. Ihre Prozeduren definieren die sinnvollen Möglichkeiten von Zugriffen auf die Speicherstruktur.
Die Prozedur

ENTER (ELEMENT: BLOCK; LIST: IOOPERATION)

schafft einen neuen frame, füllt ihn mit einem Block und reiht in die Eingabeliste oder die von der Zielleitung des Blocks bestimmte Ausgabeliste ein.

REMOVE (VAR ELEMENT: BLOCK; LINE: COMPONENTS)

entnimmt den dringendsten ausgabebereiten Block aus der angegebenen Liste und

CLOSE

vernichtet seinen frame. Die Trennung von Entnahme und Vernichtung ist aus Gründen der Fehlerbehandlung notwendig (siehe Kap. 4.5).
Diese Prozeduren werden von den Ein- und Ausgabeprozessen benutzt, also den Prozessen, die den Speicher dynamisch verändern (durch Kreation und Vernichtung von frames bei der Ein- und Ausgabe von Blöcken). Der Datenbehandlungsprozeß, der im System befindliche Blöcke bearbeitet, ohne frames zu generieren oder zu vernichten, gebraucht die folgenden Operationen:

GET (VAR ELEMENT: BLOCK)

liest den dringendsten zur Datenbehandlung anstehenden Block aus seinem frame.

PUT (ELEMENT: BLOCK)

schreibt einen ausgabebereiten Block in einen existierenden frame.
(Die Fehlerbehandlung kompliziert die beschriebene Zuordnung zwischen den BLOCKSTREAM Operationen und den Transferprozessen etwas.)

Solange Benutzer nur Kenntnis von den Prozeduren dieser Klasse haben, nicht aber von der sich dahinter verbergenden Speicherstruktur (Prinzip des information hiding, [Par75]), ist eine größere Zugriffssicherheit gewährleistet.

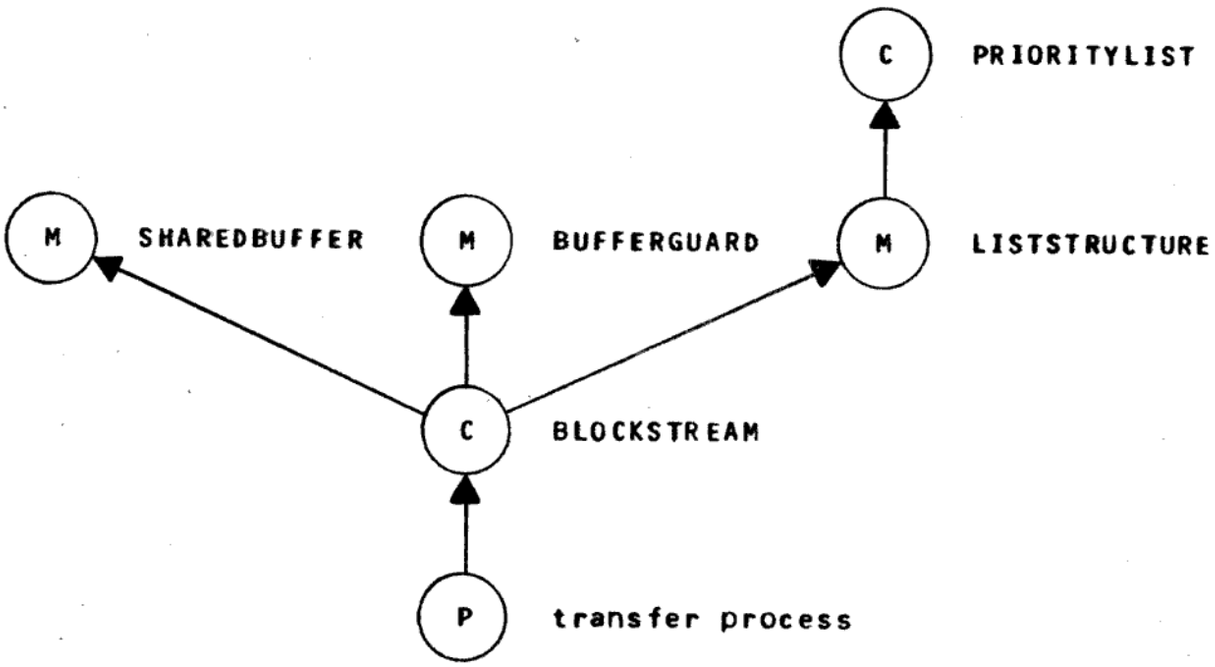

Abb. 4.3.b Speicherstruktur

Bemerkungen

1.) Der Systemtyp MONITOR ist die Implementation des kritischen Abschnittes und zugleich das einzige Mittel zur Kommunikation zwischen Prozessen in Concurrent Pascal. Das Beispiel des Blockspeichers zeigt jedoch, daß bei der Kommunikation zwischen Prozessen nicht immer Nebenläufigkeit Voraussetzung ist. Prozesse müssen auf gemeinsamen Daten nicht unbedingt unter gegenseitigem Ausschluß arbeiten. Es würde z. B. genügen, den Systemtyp SHAREDBUFFER als Klasse zu definieren, wenn eine Klasse gemein-

same Daten beherbergen dürfte. Durch die Verquickung von Zugriff auf gemeinsame Daten und gegenseitigem Ausschluß macht es Concurrent Pascal aber zugunsten einer hohen Zugriffssicherheit unmöglich, den privaten Charakter von Elementen eines gemeinsamen Datenobjektes zu erhalten.

Die Spaltung des Blockspeichers in Speicherzugriff (SHAREDBUFFER) und Zugriffsverwaltung (BUFFERGUARD) ist die Simulation eines Pointerzugriffs, den es in Concurrent Pascal nicht gibt. Mit Pointervariablen, die durch eine von der Sprache bereitgestellte flexible Zugriffsverwaltung geschützt sind, könnte nicht nur der primitive aber zeitraubende Kopiervorgang im SHAREDBUFFER entfallen, sondern insbesondere bei minimalem gegenseitigen Ausschluß eine vollständige Zugriffskontrolle erreicht werden. Den Zugriff disziplinierende Maßnahmen (wie bei COCO die BLOCKSTREAM CLASS) würden damit überflüssig.

Weiterhin macht der Monitor keine Unterschiede zwischen Lese- und Schreiboperationen. Das führt dazu, daß Leseoperationen auch untereinander unnötigerweise einem gegenseitigen Ausschluß unterliegen.

Das Konzept zur Koordination von nebenläufigen Aktivitäten in Concurrent Pascal (Monitorkozept) ist folglich zwar sicher, aber zu unkompliziert um den verschiedenartigen Anforderungen bei der Programmierung von Systemen mit Nebenläufigkeitsstruktur gerecht zu werden.

2.) Die angegebene Implementation wird der Spezifikation nur in bedingtem Maße gerecht, und zwar nur, wenn man den Datenverkehr über einen festen Ausgabekanal betrachtet. Es gibt nämlich im LISTSTRUCTURE Monitor kein mid-term scheduling der Ausgabeprozesse, sondern ihre Verwaltung stützt sich ausschließlich auf die unterliegende im Kernel definierte short-term Strategie (FIFO). Das bedeutet: Das Dringlichkeitsverhältnis von Ausgabeblöcken mit verschiedenen Zielleitungen wird nicht berücksichtigt.

Ein Weg, der FIFO-Diktion des Kernels zu entgehen, wäre, statt seiner einen weiteren unterliegenden Monitor, der anstelle von FIFO eine geeignetere Strategie verfolgt, als Grundlage für das short-term scheduling der Zugriffe auf die gemeinsamen Daten der Kanalprozesse zu verwenden.

3.) Man betrachte den in Abb. 4.3.c dargestellten alternativen Entwurf der Speicherverwaltung von COCO.

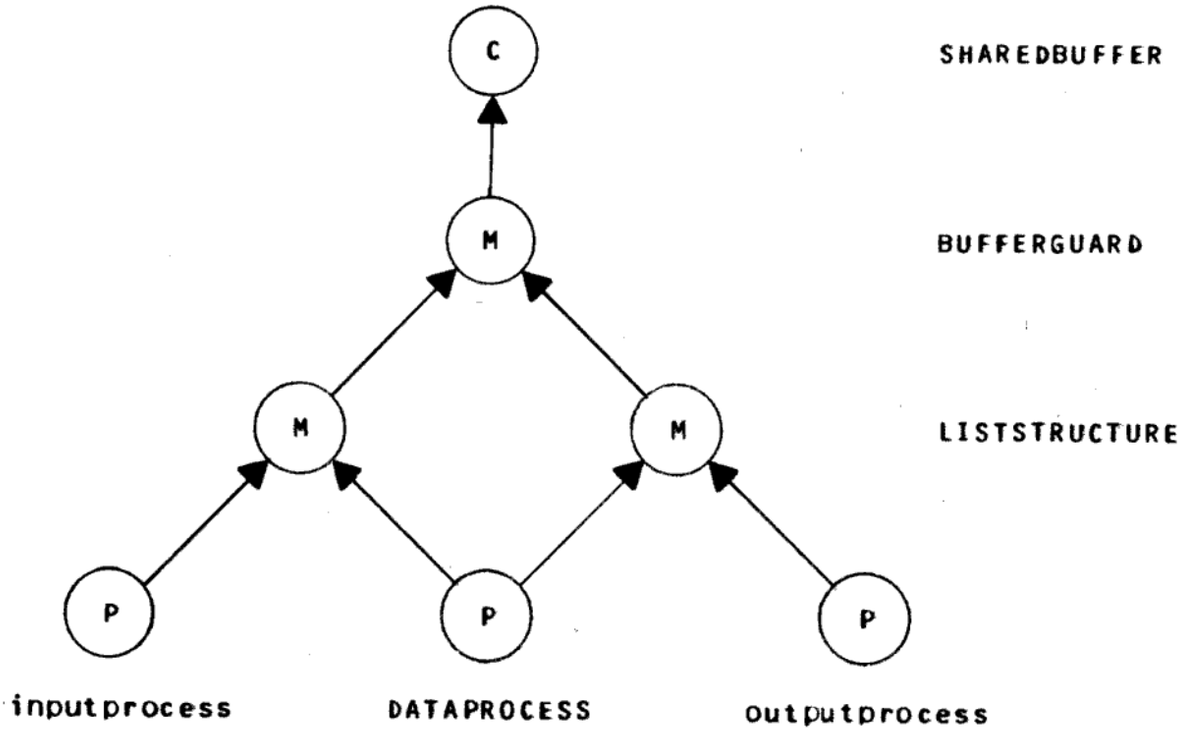

Abb. 4.3.c Alternative Speicherstruktur

Die Eingabeprozesse, der Datenbehandlungsprozeß und die Ausgabeprozesse laufen in drei Prozessoren. Daher sind die Eingabeliste und die Ausgabelisten in zwei getrennten kritischen Abschnitten untergebracht. Gemäß der hierarchischen Zugriffsbeziehungen ist eine Listenmanipulation automatisch mit der Kreation eines frames (beim Einfügen in die Liste) bzw. Vernichtung eines frames (beim Herausnehmen aus der Liste) verbunden. Abgesehen davon, daß hierbei selbst bei einer Erweiterung der Klassenstruktur im Sinne von Bem. 1 der private Charakter der einzelnen frames unberücksichtigt bleibt, weil der SHAREDBUFFER dem gegenseitigen Ausschluß des BUFFERGUARD Monitors unterliegt, eignet sich diese Konzeption zum Studium der Schwächen von Monitor-Zugriffshierarchien in Concurrent Pascal:

Mid-term scheduling Entscheidungen in einer niedrigeren Ebene der Hierarchie können Monitore in einer höheren Ebene blockieren und dadurch Verklemmungen hervorrufen. Dies ist etwa der Fall, wenn der Datenbehandlungsprozeß einen Block in eine Ausgabeliste einfügen will, aber auf die Freigabe eines frames warten muß und im LISTSTRUCTURE Monitor blockiert wird. Die einzigen weiteren frames freigebenden Prozesse im System, die Ausgabeprozesse, haben keine Möglichkeit, an ihre Listen heranzukommen und eine Freigabe zu erwirken.
In solchen Fällen müssen mid-term scheduling Entscheidungen in Monitore höherer Ebene verpflanzt werden. Das führt aber ebenfalls zur Verklemmungsgefahr, z. B. wenn Eingabeprozesse bei leerer Eingabeliste aber vollem Speicher auf die Freigabe eines frames warten. Diese kann nur von Ausgabeprozessen erwirkt

werden, die aber nicht in der Lage sind, die Eingabeprozesse zu reaktivieren, weil sie auf den betreffenden Monitor nicht zugreifen.

Folgende Schlußfolgerung ist zu ziehen:

Essentiell ist, daß alle Prozesse, die sich gegenseitig beeinflussen, auch miteinander kommunizieren können. Deshalb sollten in einem Monitor nur die mid-term scheduling Entscheidungen getroffen werden, die sich ausschließlich auf Zustände der von diesem Monitor geschützten Daten beziehen. Monitor-Hierarchien sind auf Verklemmungsgefahr hin zu überprüfen oder zu vermeiden.

Diese Überlegungen führten zu der beschriebenen Speicherstruktur.

4.) Die Zusammenfassung der verschiedenen Teillisten (repräsentiert durch PRIORITYLIST Objekte) zu einem Systemobjekt (LISTSTRUCTURE) ist ein Kompromiß, um die Flexibilität des Systems gegenüber Änderungen der Hardware-Konfiguration (Adaptivität) zu erhöhen. Dadurch wird erreicht, daß die Addition bzw. Subtraktion von Kanälen durch eine einfache Umdefinition der entsprechenden Kernelschnittstelle, des Datentyps IODEVICE, vorgenommen werden kann. Ohne diese Hülle könnte man die Menge der Teillisten, die mit der Menge der Kanäle zusammenhängt, nicht geschlossen behandeln. Denn Concurrent Pascal läßt keinen strukturierten Datentyp als Zugriffsparameter eines Systemtyps zu. Man beachte jedoch, daß im Falle einer Mehrprozessor-Konfiguration alle Listen, die von in unterschiedlichen Prozessoren laufenden Prozessen bearbeitet werden, in getrennten Monitor-Hüllen untergebracht sein müssen, um die Parallelität nicht zu beschneiden. Die vorliegende Implementation ist also nur für ein Einprozessorsystem geeignet.

5.) Der verwendete Concurrent Pascal Compiler moniert Variablen vom Typ QUEUE in einem Objekt, das kein Monitor ist, selbst im Fall, daß der Typ des Objektes lokal zu einem Monitor definiert ist. Dies trifft auf die im LISTSTRUCTURE MONITOR definierte PRIORITYLIST CLASS zu.
Die syntaktisch korrekte Version des LISTSTRUCTURE Monitors stellt sich für den Benutzer vollkommen identisch dar. Die Verschmelzung der Unterstrukturen führt aber zu einer Adaptivitätseinbuße. Der Speicherverbrauch durch die Listen ist etwas verringert worden. Beide Alternativen sind in Kap. 6.1 angegeben.

Die Definition der Sprache ist an dieser Stelle nicht hinreichend dokumentiert ([BrHa75a]). Eine derart generelle Einschränkung von QUEUE Variablen auf Monitore würde ihre Modularität über Gebühr einschränken.

6.) Über die geeignete Größe des Blockspeichers wurden keine Untersuchungen angestellt. Sie wird von der Natur der Kanäle im Anwendungsfall abhängen.

4.4 Realzeitkontrolle

Die Alterung der im System auf ihre Übertragung wartenden Blöcke geschieht durch Erhöhen der dynamischen Priorität mittels Unterbrechung durch eine Uhr.

Der DYNPRIORITY MONITOR

enthält die dynamische Priorität als gemeinsame Variable mehrerer Prozesse und definiert die zulässigen Operationen darauf.
Über die Prozedur

INCREMENT (PRIORITYSTEP: REAL)

kann der Prioritätswert um einen spezifizierten Betrag erhöht werden,
die Funktion

COUNT: REAL

gibt Auskunft über den augenblicklichen Stand der dynamischen Priorität.
Der Anfangswert ist Null.

Der TIMERPROCESS PROCESS

übernimmt die dynamische Veränderung des Prioritätswertes. Möglich sind Inkrementierungen um einen definierten Wert in Abständen, die ein ganzzahliges Vielfaches der vom run-time System durch die Standardprozedur WAIT zur Verfügung gestellten Minimalfrequenz darstellen.

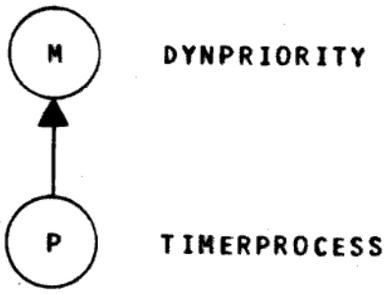

Abb. 4.4.a Realzeitkontrolle

Bemerkungen

1.) Wegen der geringen Wortlänge der PDP-11/45 (16 Bit) können für eine nichtzyklische Prioritätsstruktur nicht genügend verschiedene positive Zahlen durch INTEGER Werte dargestellt werden (vgl. Kap. 3.3). Deshalb wurde für die dynamische Priorität eine REAL Darstellung gewählt. Damit ist das System bei einer Inkre-

mentierungsfrequenz von 100 Hz (als Vielfachem der Kernelfrequenz) und 10**12 verschiedenen Prioritätswerten (vgl. real-time definitions in Kap. 6.1) über 100 Jahre lauffähig.

2.) Die vom zugrundeliegenden Kernel generierte Frequenz beträgt ca. 1 Hz. Für ein Kommunikationssystem wäre eine höhere Grundfrequenz wünschenswert (vgl. Bem. 1). Das beweist erneut die geringe Eignung dieses Kernels für die Entwicklung solcher Systeme.

4.5 Fehlerbehandlung

Design

Es gibt drei Fehlerarten:

a) LOGIC

Vor dem Einbringen eines neu eingegebenen Blocks in das System wird das Format der routing Information (Blockheader) überprüft. Eine Nicht-Übereinstimmung mit Systemformat bedeutet einen logischen Fehler.

b) DATA

Vor einer Datenbehandlung wird das Format der Blockdaten überprüft. Nicht-Übereinstimmung mit dem für die Datenbehandlung vorausgesetzten Format bedeutet einen Datenfehler.

c) CHANNEL

Ein Kanalfehler tritt auf, wenn die Zielleitung eines Blocks nicht funktionsfähig ist.

Fehler werden dezentral im System erkannt:

a) logische Fehler bei der Eingabe
b) Datenfehler bei der Datenbehandlung
c) Kanalfehler bei der Ein- und Ausgabe

Die Behandlung von Blöcken, bei deren Übertragung ein Fehler auftritt, wird zentral vorgenommen. Sie hängt vom Fehlertyp des Blocks ab (siehe Kap. 4.1).

Zur Erkennung von Kanalfehlern benötigt man eine mehreren Prozessen zugängliche Tabelle, die den Zustand der einzelnen Kanäle angibt.

Ein Konsolenprozeß gibt dem Operateur die Möglichkeit, Informationen über den augenblicklichen Zustand der Datenleitungen zu erhalten und ihn gegebenenfalls zu verändern.

Implementation

Die Zustandstabelle beherbergt

der STATETABLE MONITOR.

Sie enthält für jeden Ausgabekanal die Information "funktionsfähig" (TRUE) oder "nicht funktionsfähig" (FALSE).
Über die Prozedur

SETSTATE (LINE: COMPONENTS; RESULT: BOOLEAN)

kann dieser Zustand definiert werden,
über die Funktion

CONTAIN (LINE: COMPONENTS): BOOLEAN

erhält man die Aussage, ob der im Aufruf spezifizierte Ausgabekanal funktionsfähig ist, also vom System unterstützt werden soll oder nicht. Bei Initialisierung der Tabelle werden alle Kanäle als funktionsfähig deklariert.

Der Fehlerbehandlungsmodul ist

die ERRORHANDLER CLASS.

Sie besteht nur aus der Prozedur

HANDLE (VAR ELEMENT: BLOCK; CODE: SYSTEMERROR; CALLER: IOOPERATION)

,die den Block der Art des Fehlers und seinem Fehlertyp gemäß behandelt.
Da die beschriebenen Implementation keine Datenbehandlung enthält (vgl. Kap. 4.2), können Fehler vom Typ DATA nicht auftreten. Systemnachrichten in Abhängigkeit vom ermittelten Fehler sind vorgesehen aber nicht ausgearbeitet worden.

Fehlermeldungen beziehen sich oft auf Kanäle, also Skalare vom Typ COMPONENTS. Daher muß eine Klasse für die Ein/Ausgabe von Werten dieses Typs vorgesehen werden. Sie wird hier nicht näher erläutert .

Der OPERATORPROCESS PROCESS

stellt die Kommunikation mit dem Operateur her. Bei seiner Initialisierung informiert er über den zu diesem Zeitpunkt definierten Zustand aller im System spezifizierten Kanäle und gibt dem Operateur sofort die Möglichkeit zur Umdefinition. Die Bereitschaft zur Entgegennahme eines Kommandos zeigt der Prozeß durch die Meldung

OPERATOR>

an. Folgende Kommandos werden akzeptiert:

+Kanalname: Einbringen eines Ausgabekanals in das System,
-Kanalname: Entfernen eines Ausgabekanals aus dem System,
STATE: Angabe der augenblicklichen Zustände von aller im System spezifizierten Ausgabekanäle,
return: Abschluß der Kommandoeingabe.

Andere Kommandos werden mit der Meldung

ILL CMD!

zurückgewiesen.
Die Anforderung des Operateurs zur Kommunikation mit dem System erfolgt durch die Eingabe des Steuerzeichens ctrl g (BEL).

Bemerkungen

1.) Die ERRORHANDLER Spezifikation ist ein Beispiel für den häufig auftretenden Mißbrauch des Systemtyps CLASS als Bibliothek von globalen Prozeduren, die mehreren Prozessen gemeinsame Handlungen durchführen, ohne auf geschützten Daten zu operieren. Der Grund dafür ist die in Concurrent Pascal vorgesehene Hierarchie des Definitionsteils im Programm: Datentypen (so auch Prozesse) müssen vor Prozeduren definiert werden. Ein Merkmal derartig mißbrauchter Klassen ist ein leeres Initialstatement.

2.) Es gibt zwei Möglichkeiten, ein Zugriffsrecht auf ein Systemobjekt zu spezifizieren:

a) durch Deklaration des Objektes als Variable
b) durch Angabe des Objektes als Zugriffsparameter

Zueinander nebenläufige Programmteile müssen eigene CLASS Objekte deklarieren, da die Klasse sequentiellen Charakter hat. Aber in einem Teil der Zugriffshierarchie, der nur aus CLASS Objekten besteht, dh. keine Nebenläufigkeit enthält, darf die Klasse als Zugriffsparameter auftreten. Wenn auf sie von mehreren Ebenen der Hierarchie zugegriffen wird, ist diese "Pseudoglobalität" der Klasse als Mittel zur Kommunikation der zugreifenden Elemente untereinander sinnvoll. Die Klasse erfüllt also im sequentiellen Bereich ähnliche Aufgaben wie der Monitor im nebenläufigen Bereich (vgl. Kap. 4.2, Bem. 1 und Kap. 4.3, Bem. 1).

Ein Beispiel für eine solche Situation ist die BLOCKSTREAM CLASS (Abb. 4.5.a). Die zueinander nebenläufigen SYSTEMGUARD Objekte deklarieren ihre eigenen Objekte dieser Klasse, während in dem auf einer niedrigeren Zugriffsebene angesiedelten ERRORHANDLER der Zugriff über einen Parameter spezifiziert werden kann.

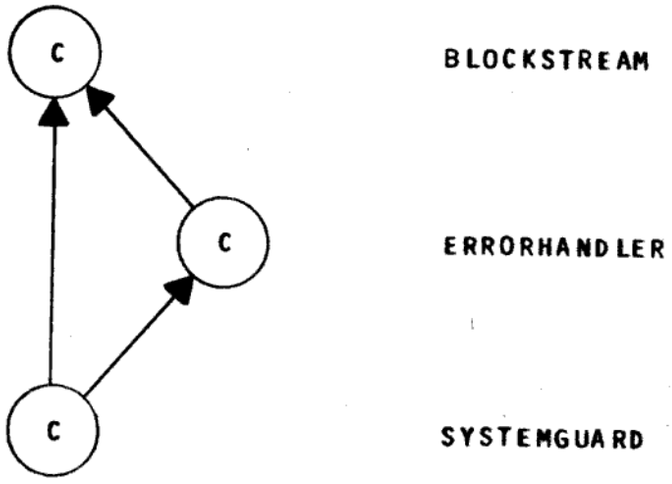

Abb. 4.5.a Sequentielle Zugriffshierarchie

4.6 Virtuelle Geräte

Periphere Geräte, die einen sequentiellen Zugriff erfordern, aber von nebenläufigen Prozessen benutzt werden (etwa Terminals), müssen einer Sequenzierungskontrolle unterliegen, die z. B. durch die Semaphore bereitgestellt wird.

Der RESOURCE MONITOR

ist die Implementation einer boolschen Semaphore mit den Prozeduren

REQUEST

als P-Operation,

RELEASE

als V-Operation
und der üblichen Initialisierung.
Bekanntlich ist der Benutzer für den sinnvollen Gebrauch der Semaphore-Operationen verantwortlich.

Im COCO System werden die Konsole und die Terminals nach dem in Abb. 4.6.a skizzierten Zugriffsschema virtuell verwaltet.

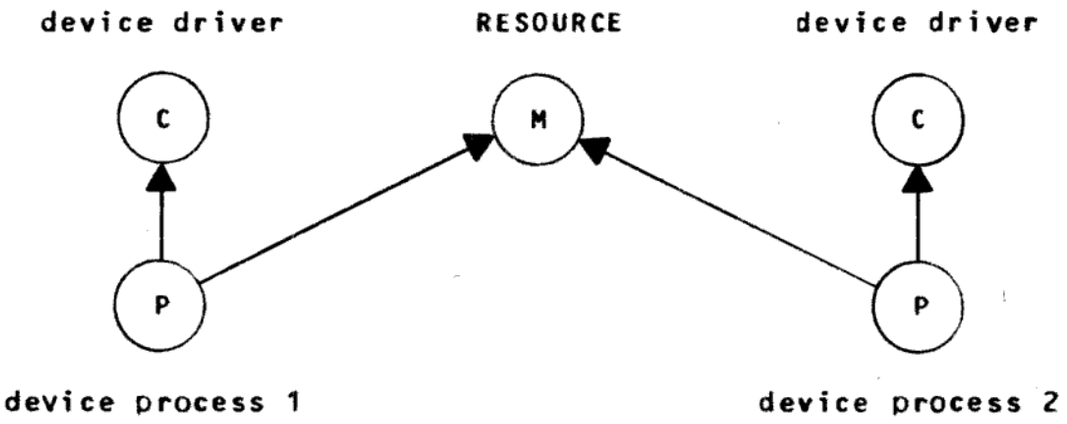

Abb. 4.6.a Virtuelle Geräteverwaltung

Bemerkungen

1.) Solange ein ein virtuelles Gerät verwaltender Monitor eine andere als die im Kernel fixierte scheduling Strategie (FIFO) verfolgt, sollte er wirklich ausschließlich Verwaltungsaufgaben und keine Ein/Ausgabe durchführen. Die im Monitor definierte mid-term scheduling Strategie kommt nämlich nur zur Geltung, wenn die Buchführung über das Gerät betreffende Anforderungen (pseudo-) parallel zur Ein/Ausgabe vorgenommen werden kann. Befände sich die Ein/Ausgabe mit dem scheduling im selben kritischen Abschnitt (Monitor), würde diese (Pseudo-) Parallelität unterbunden und das scheduling würde sich auf die short-term Ebene verlagern (ein ähnliches Problem wird in Kap. 4.3, Bem. 2 angesprochen).

   Wenn bei der Verwaltung eines virtuellen Gerätes die Kernel-Strategie verfolgt werden soll, kann auf ein mid-term scheduling verzichtet und die Ein/Ausgabe in einem nicht synchronisierenden Monitor untergebracht werden, sodaß der Gerätebenutzer jeglicher Verantwortung für die Geräteverwaltung enthoben ist. Will man jedoch den Zugriff auf das Gerät nicht in MONITOR Prozeduren festlegen, so muß der gegenseitige Ausschluß im allgemeinen mit Hilfe der RESOURCE Operationen erwirkt werden.

4.7 Konsolenkontrolle/Feldverwaltung

Die Systemtypen zur Verwaltung der Konsole wurden in Anlehnung an frühere Systeme als eine Hierarchie von Drivern zunehmender Intelligenz konzipiert.

Die Feldverwaltung folgt ebenfalls veröffentlichten Vorschlägen. Über CLASS Prozeduren können vom Benutzer definierte Feldvariablen mit einer FIFO- bzw. LIFO- (Stack-) Strategie bearbeitet werden. Die

Verantwortung für eine dem Status des Feldes entsprechende Benutzung der CLASS Operationen obliegt dem Benutzer.

Auf eine weitergehende Dokumentation wird an dieser Stelle verzichtet (vgl. [BrHa75c, BrHa75d]).

4.8 Systemstruktur

Der Definitionsteil der Datentypen bestimmt eine durch die Zugriffshierarchie der darin enthaltenen Systemtypen charakterisierte Menge von Kommunikationsbetriebssystemen, die der Spezifikation genügen, von der ausgegangen wurde. Das Endziel, eine in der vom Typdefinitionsteil bestimmten Systemmenge enthaltene Implementation, wird durch den sich daran anschließenden Initialprozeß (initial process) festgelegt. Er enthält die Deklarationen und Initialisierungen geeigneter Objekte der definierten Systemtypen.

Der Zugriffsgraph des Systems (Abb. 4.8.a) zeigt die Zugriffsbeziehungen zwischen den Objekten der in diesem Abschnitt beschriebenen Systemtypen.

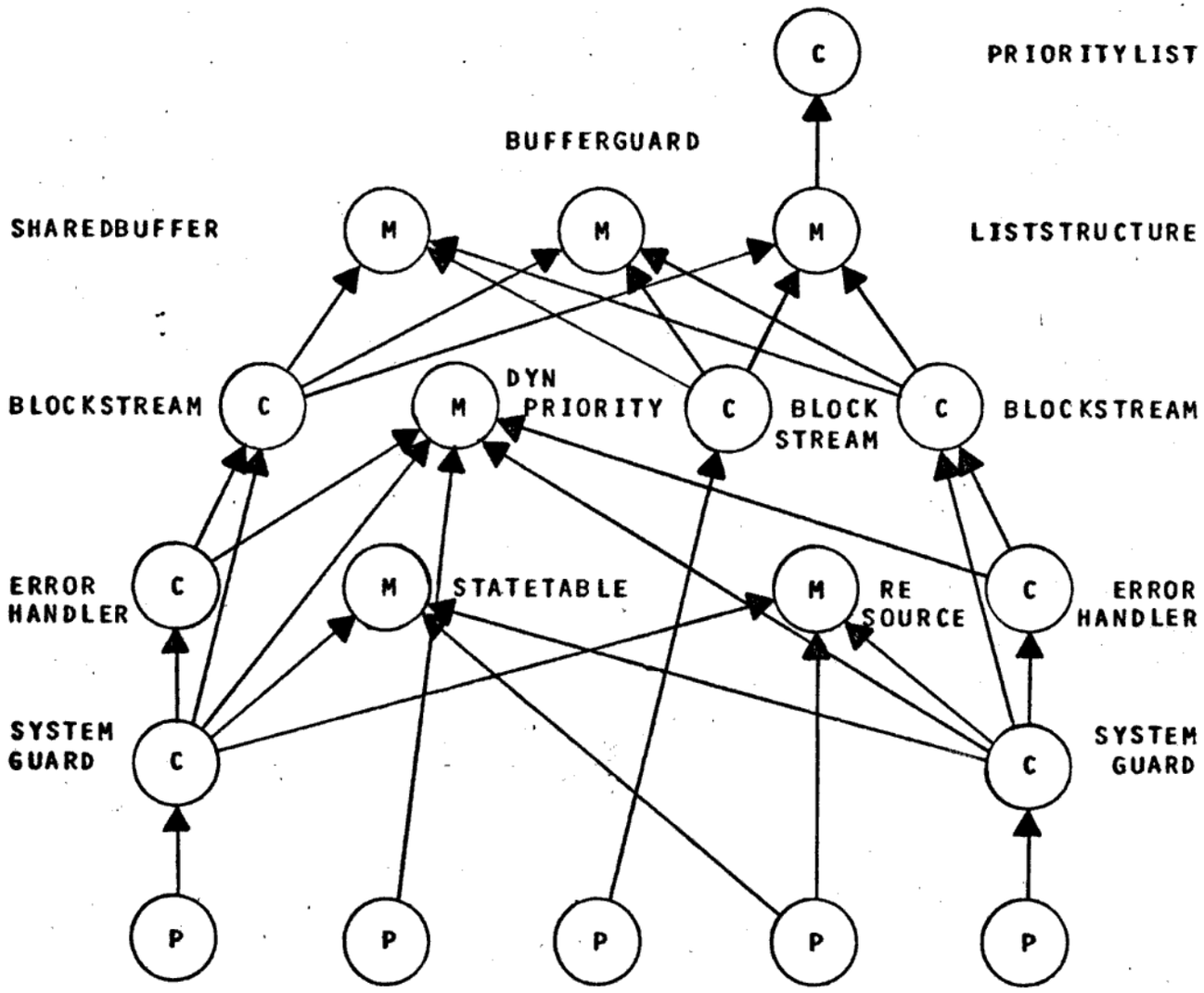

Abb. 4.8.a Systemstruktur

Bemerkungen

1.) Die Parnas'sche Definition der Zugriffshierarchie (siehe Kap. 2.3) ergibt nur zwischen funktional zusammenhängenden Systemobjekten einen Sinn. Bei einer globaleren Betrachtung lassen sich die verschiedenen Ebenen nicht mehr geschlossen interpretieren (siehe Abb. 4.8.a). Deshalb ist auch ihre Kennzeichnung mit absoluten Indizes sinnlos und unterlassen worden. Nur die relativen Ordnungsabhängigkeiten sind von Bedeutung.

5 Verifikation

Da ein entsprechendes run-time System nicht zur Verfügung stand, ist die Funktionsfähigkeit der Transferprozesse nur in eingeschränktem Maße überprüft worden. Zunächst wurde die Terminaldriver-Hierarchie auf in Kap. 2.4 beschriebene Weise systematisch getestet und dann eine stark vereinfachte Implementation von COCO entwickelt, die mit dem verwendeten run-time System auskommt. (Sie betreibt ein Netz, in dem die Konsole des Knotenrechners zugleich das einzige DTE darstellt.)

Für die Verifikation der Speicherstruktur sind folgende Eigenschaften zu untersuchen:

a) das sequentielle Verhalten
b) das Synchronisationsverhalten
c) das scheduling Verhalten

Die Eigenschaften a) und b) wurden mit Hilfe der formalen Verifikation untersucht. Das Ergebnis sind Beweisregeln für die auf der Speicherstruktur definierten Operationen. Es werden nur die wesentlichen Teile der Struktur betrachtet:

1.) Blockspeicher

    Es sind Aussagen über die Art der Speicherverwaltung zu formulieren. Sie wird durch den BUFFERGUARD MONITOR repräsentiert, der sich seinerseits zur Verwaltung der Freispeicherliste (FREELIST) der STACK CLASS bedient. Der SHAREDBUFFER MONITOR wird nicht betrachtet. Er reflektiert nur die spezielle Natur der Speicherelemente, die für das Verhalten des Speichers nicht relevant ist.

2.) Ordnungsstruktur

    Es wird nur der Kern der Ordnungsstruktur, die Datenstruktur PRIORITYLIST betrachtet. Sie wird als Monitor angenommen (vgl. Kap. 4.3, insbes. Bem. 4).

Für die Untersuchung von Eigenschaft c) muß man die Speicherstruktur in Zusammenhang mit ihrer Verwendung sehen. Durch Simulation der auf sie zugreifenden Systemobjekte wurden Übertragungssituationen nachgebildet und getestet.

Einzelheiten über die Verifikation der Speicherstruktur und dabei gewonnene Erkenntnisse über die verwendete Methode (siehe Kap. 2.4) enthält dieser Abschnitt. Zur Termination wird nur in Fällen Stellung genommen, in denen sie problematisch erscheint.

Die Realzeitkontrolle wurde mit Hilfe eines look-up Prozesses überprüft, der auf Anforderung über den augenblicklichen Stand der dynamischen Priorität informiert.

Die Fehlerbehandlungsmodule wurden ebenfalls mit systematischen

Tests untersucht (den nur andeutungsweise implementierten ERRORHANDLER ausgenommen).

5.1 Sequentielles Verhalten der Speicherstruktur

1.) Der Blockspeicher

Folgende Beweisregeln für die STACK Operationen sind trivial nachzuweisen:

(S1) "true" INIT "pointer=last+1"
(S2) "pointer=p" PUSH "pointer=p+1 & push=p"
(S3) "pointer=p" POP "pointer=p-1 & pop=p-1"

Für die STACK Statusfunktionen brauchen an dieser Stelle keine Beweisregeln formuliert zu werden, da sie das sequentielle Verhalten der Speicherverwaltung nicht beeinflussen.

Bei Nicht-Berücksichtigung der Synchronisationsoperationen wird der BUFFERGUARD MONITOR zu

```
TYPE BUFFERGUARD = MONITOR

     VAR FREELIST: ARRAY (.BUFFERINDEX.) OF BUFFERINDEX;
         NEW: STACK;
         I: BUFFERINDEX;

     PROCEDURE ENTRY REQUEST (VAR EL: BUFFERINDEX);
     BEGIN EL := FREELIST(.NEW.POP.);
     END;

     PROCEDURE ENTRY RELEASE (EL: BUFFERINDEX);
     BEGIN FREELIST(.NEW.PUSH.) := EL;
     END;

     BEGIN FOR I := MININDEX TO MAXINDEX
            DO FREELIST(.I.) := I;
           INIT NEW(MININDEX,MAXINDEX);
     END;
```

Folgende Aussagen sollten garantiert werden:

i) Nach der Initialisierung enthält die Freispeicherliste eine Basismenge von Speicherelementen (frame pool). Alle Operationen des Monitors arbeiten ausschließlich mit Elementen dieser Menge.

ii) Die Operation REQUEST verringert die Menge der freien frames um ein Element. Voraussetzung ist, daß vor ihrer Ausführung ein freies Speicherelement existiert. In Kap. 5.2 wird gezeigt werden, daß diese Voraussetzung durch die den Prozedureingang überwachende Warteoperation erzwungen wird.

iii) Die Operation RELEASE addiert einen frame zu der Freispeicherliste. Voraussetzung ist hier, daß der frame vor Ausführung der Operation nicht leer ist, aber zur Grundmenge gehört. Die Gültigkeit dieser Voraussetzung kann nicht durch den Monitor erzwungen, sondern muß bei der Verifikation der auf die Speicherverwaltung zugreifenden Systemtypen nachgewiesen werden. Im COCO System ist sie durch die Tatsache garantiert, daß die Operationen REQUEST und RELEASE abwechselnd ausgeführt werden (angefangen mit REQUEST) und daß auf ihr Argument vom zugreifenden Systemtyp keine Zuweisung erfolgt.

Diese Forderungen lassen sich folgendermaßen formalisieren und nachweisen:

Behauptung:

Seien

B := minindex..maxindex die Basismenge der frames,
F := {j| es ex. i∈minindex..pointer-1 mit freelist(.i.) = j}
  die Menge der freien frames

i)	"true"	INIT	"F=B"
ii)	"∅<>f⊂B & F=f"	REQUEST(EL)	"el=a mit a∈f & F=f-{a}"
iii)	"F=f & el=a∈B-f"	RELEASE(EL)	"F=f∪{a}"

Beweis:

Seien

a := freelist(.p-1.)
f := {j| es ex. i∈minindex..p-1 mit freelist(.i.) = j}

i) "freelist(.j.) = j für alle j∈minindex..i-1"
FREELIST(.I.) := I
"freelist(.j.) = j für alle j∈minindex..i"
=>
"true"
FOR I:= MININDEX TO MAXINDEX
DO FREELIST(.I.) := I
"freelist(.j.) = j für alle j∈minindex..maxindex"
INIT NEW(MININDEX,MAXINDEX)
"freelist(.j.) = j für alle j∈minindex..maxindex
& pointer=maxindex+1 (S1)"
"F=B"

ii) "∅<>f⊂B & F=f,
pointer=p, minindex<p<=maxindex+1"
EL := FREELIST(.NEW.POP.)
"el=a∈f & pointer=p-1 (S3)"
"F=f-{a}"

iii) "F=f & el=a∈B-f,
pointer=p, minindex<=p<maxindex+1"
FREELIST(.NEW.PUSH.) := EL
"freelist(.p.) = a ∈B-f & pointer=p+1 (S2)"
"F=f∪{a}"

q.e.d.

Eine Invariante ist offensichtlich die Aussage
minindex<=pointer<=maxindex+1 & ∅⊂F⊂B.
Für eine nichttriviale Invariante ist die Struktur zu einfach.

Man beachte, daß die bei der Verwaltung der Freispeicherliste verfolgte LIFO-Strategie in den bewiesenen Aussagen nicht zum Ausdruck kommt. Sie ist irrelevant, da alle freien Speicherelemente gleichberechtigt sind.

2.) Die Ordnungsstruktur

Abstrahiert man von den Synchronisationsoperationen, so ist die folgende Datenstruktur zu untersuchen:

```
TYPE PRIORITYLIST = MONITOR;

     CONST EXTENSION = 0;

     TYPE LISTINDEX        = EXTENSION..MAXINDEX;

          PRIORITYELEMENT = RECORD SYSTEMPRIORITY: REAL;
                                   SUCC: LISTINDEX
                            END;

          PRIORITYQUEUE    = ARRAY (.LISTINDEX.)
                               OF PRIORITYELEMENT;

     VAR LIST: PRIORITYQUEUE;

     PROCEDURE ENTRY ENTER (EL: BUFFERINDEX; PRI: REAL);
     VAR I: LISTINDEX;
     BEGIN I := EXTENSION;
           WHILE LIST(.LIST(.I.).SUCC.).SYSTEMPRIORITY <= PRI
              DO I := LIST(.I.).SUCC;
           LIST(.EL.).SYSTEMPRIORITY := PRI;
           LIST(.EL.).SUCC := LIST(.I.).SUCC;
           LIST(.I.).SUCC := EL;
     END;

     PROCEDURE ENTRY REMOVE (VAR EL: BUFFERINDEX);
     BEGIN EL := LIST(.EXTENSION.).SUCC;
           LIST(.EXTENSION.).SUCC :=
             LIST(.LIST(.EXTENSION.).SUCC.).SUCC;
     END;

     BEGIN LIST(.EXTENSION.).SYSTEMPRIORITY := MAXCOUNT;
           LIST(.EXTENSION.).SUCC := EXTENSION;
     END;
```

Sei a ein Wert eines strukturierten Datentyps, dann bezeichne a[a.x<-y] den Wert, der sich durch die Zuweisung a.x := y ergibt.

Seien list eine Prioritätsliste (PRIORITYQUEUE), el ein Eintrag einer Prioritätsliste (LISTINDEX) und pri eine Priorität. In Anlehnung an [How76a] wird definiert:

(D1) list.first := list(.extension.).succ
ist das erste Element von list.

(D2) list.rest := list[list(.extension.).succ<-extension]
ist list ohne das erste Element.

(D3) el.pri := list(.el.).systempriority
ist die Priorität des Listeneintrages el.

(D4) el.succ := list(.el.).succ
ist der Nachfolger des Listeneintrages el.
el.succ(0) := el
el.succ(n+1) := el.succ(n).succ, n>=0

(D5) el in list :<=>
el <> extension und
es ex. n>=0 mit list.first.succ(n) = el und
für alle m mit 0<=m<n ist list.first.succ(m) <> extension
Die Menge {el| el in list} wird ebenfalls mit list bezeichnet.

(D6) list.min := min{el.pri| el in list}
list.max := max{el.pri| el in list}

(D7) Die leere Prioritätsliste empty ist bestimmt durch
empty.first = extension
empty.rest = empty
empty.min := $-\infty$
empty.max := $+\infty$

Folgende Forderungen sind sinnvoll:

i) In Augenblicken, in denen auf die Liste nicht zugegriffen wird, sind ihre Einträge nach Prioritäten geordnet.

ii) Nach der Initialisierung ist die Liste leer.

iii) Die Operation ENTER fügt ein Element in die Liste so ein, daß die Ordnung der Prioritäten erhalten bleibt. Voraussetzung ist, daß für das neue Listenelement noch Speicherplatz zur Verfügung steht. Die Gültigkeit dieser Voraussetzung kann nicht durch den Monitor erzwungen, sondern muß bei der Verifikation der auf die Ordnungsstruktur zugreifenden Systemtypen nachgewiesen werden. Im COCO System ist sie dadurch gesichert, daß die Liste alle in Frage kommenden Elemente (frames) aufnehmen kann, dh. eine Beschränkung

nicht besteht.

iv) Die Operation REMOVE entfernt das höchstpriorisierte Element aus der Liste. Vorausgesetzt wird, daß die Liste vor Ausführung der Operation nicht leer ist. In Kap. 5.2 wird gezeigt werden, daß diese Voraussetzung durch die den Prozedureingang überwachende Warteoperation erzwungen wird.

Formalisiert werden diese Aussagen zur

Behauptung:

i) Sei l eine Prioritätsliste.
J(l) := (j.pri <= j.succ.pri für alle j in l)
Insbesondere: J(l) => l.min=l.first.pri
Dann gilt:
J := J(list) ist Monitorinvariante.

ii)	"true"	INIT	"list=empty & J"
iii)	"J & list=l"	ENTER(EL,PRI)	"list=l∪{el} & J"
iv)	"J & list=l<>empty"	REMOVE(EL)	"el=l.first & list=l.rest & list.min>=l.min & J"

Beweis:

i) ergibt sich aus den Beweisen von ii), iii) und iv).

ii) "true" LIST(.EXTENSION.).SYSTEMPRIORITY := MAXCOUNT;
LIST(.EXTENSION.).SUCC := EXTENSION
"list=empty (D1,D7)" "list=empty & J"

iii) P := (J & list=l & (i in list => i.pri<=pri))
B := (i.succ.pri <= pri)

"P & B"
"J & list=l & (i in list => i.pri<=pri) & i.succ.pri<=pri"
I := LIST(.I.).SUCC
"J & list=l & i.pri<=pri" "P"
=>
"J & list=l"
I := EXTENSION
"J & list=l & ¬(i in list)" "P"
WHILE LIST(.LIST(.I.).SUCC.).SYSTEMPRIORITY <= PRI
DO I := LIST(.I.).SUCC
"P & ¬B"
"J & list=l & (i in list => i.pri<=pri) & pri<i.succ.pri"
LIST(.EL.).SYSTEMPRIORITY := PRI
"J & list=l & (i in list => i.pri<=el.pri)
& el.pri<i.succ.pri"
LIST(.EL.).SUCC := LIST(.I.).SUCC;
LIST(.I.).SUCC := EL
"J(l) & list=l∪{el} (D5)
& (i in list => i.pri<=el.pri) & el.pri<el.succ.pri"
"list=l∪{el} & J"

```
iv)  "J & list=l<>empty"
      EL := LIST(.EXTENSION.).SUCC
     "el=l.first (D1) & J & el.pri=l.min"
      LIST(.EXTENSION.).SUCC :=
        LIST(.LIST(.EXTENSION.).SUCC.).SUCC
     "el=l.first & list=l.rest (D2) & J(l) & el.pri=l.min"
     "el=l.first & list=l.rest & list.min>=l.min & J"
```

q.e.d.

Die Termination der Prozedur ENTER wird durch die Tatsache garantiert, daß der Nachfolger des letzten Listenelementes zu jeder Zeit eine geringere Dringlichkeit hat als jeder Transferblock (im Rahmen des für die Realzeitkontrolle vorgegebenen Wertebereichs; vgl. Kap. 4.4, Bem. 1). Bei der Initialisierung wird ihm nämlich die für das System geltende obere Schranke der dynamischen Prioritäten zugewiesen. für jeden beliebigen Block ist also mindestens die Eintragung als letztes Listenelement erfolgreich.

5.2 Synchronisationsverhalten der Speicherstruktur

Die Problematik gleicht sich bei beiden Strukturen, dem Blockspeicher (BUFFERGUARD MONITOR) und der Ordnungsstruktur (PRIORITYLIST MONITOR). Die Synchronisationsbedingung ist in beiden Fällen das Nicht-Vorhandensein von Listenelementen, beim Blockspeicher der Freispeicherliste, bei der Ordnungsstruktur der Prioritätsliste.

Ein geeignetes Konzept für die Synchronisation bei der Verwaltung einer begrenzten Anzahl ununterscheidbarer Objekte ist die Semaphore. In der nachstehenden Implementation gibt die Variable S die Anzahl der verfügbaren Objekte, QL die Anzahl der in der Schlange Q wartenden Prozesse an:

```
TYPE SEMAPHORE = MONITOR (N: "POSITIVE" INTEGER);

     VAR S, QL: INTEGER;
         Q: CONDITION;

     PROCEDURE ENTRY P;
     BEGIN QL := QL + 1;
           IF QL > S THEN Q.WAIT;
           QL := QL - 1;
           S := S - 1;
     END;

     PROCEDURE ENTRY V;
     BEGIN S := S + 1;
           IF QL > 0 THEN Q.SIGNAL;
     END;
```

```
BEGIN S := N;
      QL := 0;
END;
```

Folgende Forderungen sollten außerhalb von Zugriffen auf den Monitor erfüllt sein:

i) Es gibt keine negativen Bestände von Objekten.

ii) Es gibt keine negativen Bestände von wartenden Prozessen.

iii) Die Menge der verfügbaren Objekte oder die der wartenden Prozesse ist leer.

iv) Ein wartender Prozeß wird aufgeweckt, sobald ein Objekt verfügbar ist.

Die Forderungen iii) und iv) in Zusammenhang mit der FIFO-Auswahlstrategie des Warteschlangentyps CONDITION garantieren, daß unnötiges Warten (starving) vermieden wird.

Das führt zur

Behauptung:

i) s>=0
ii) ql>=0
} J := (min(s,ql)>=0)

iii) E := (min(s,ql)=0)

iv) B(Q) := (ql>0 & s=1)

Unter der Voraussetzung von (M1) und (M2) genügen die Aussagen J, E und B(Q) den Beweisregeln (M3) und (M4) für den Monitor (siehe Kap. 2.4).

Beweis:

J & E <=> E,
J & B(Q) <=> B(Q)

Es wird vorausgesetzt:

"min(s,ql)=0" Q.WAIT "ql>0 & s=1" (M1)
"ql>0 & s=1" Q.SIGNAL "false" (M2)

(M3) ergibt sich trivial:

```
"true" S := N;
       QL := 0
"min(s,ql)=0"
```

Für die Gültigkeit von (M4) ist zu zeigen

```
"J & E" P "J & E":

P := (min(s,ql-1)=0)
B := (ql>s)
Q := (min(s,ql)-1=0)

"P & B"
"min(s,ql-1)=0 & ql>s"
"min(s,ql)=0"
 Q.WAIT
"ql>0 & s=1 (M1)"
"min(s,ql)=1" "Q"
und
P & ¬B => min(s,ql-1)=0 & ql<=s
       => min(s,ql)=ql & ql-1=0
       => min(s,ql)=1
       => Q
=>
"min(s,ql)=0"
 QL := QL + 1
"min(s,ql-1)=0"
 IF QL > S THEN Q.WAIT
"min(s,ql)-1=0"
 QL := QL - 1;
 S := S - 1;
"min(s,ql)=0"
```

```
"J & E" V "J & E":

P := (min(s-1,ql)=0)
B := (ql>0)
Q := (min(s,ql)=0)

"P & B"
"min(s-1,ql)=0 & ql>0"
"ql>0 & s=1"
 Q.SIGNAL
"false (M2)" "Q"

und
P & ¬B => min(s-1,ql)=0 & ql<=0
       => ql=0 & s-1>=0
       => min(s,ql)=0
       => Q
=>
"min(s,ql)=0"
 S := S + 1
"min(s-1,ql)=0"
 IF QL > 0 THEN Q.SIGNAL
"min(s,ql)=0"
```

q.e.d.

Es bleibt zu zeigen, daß P und V den Synchronisationsoperationen im Blockspeicher und in der Ordnungsstruktur entsprechen. Dazu müssen die beiden Hilfsgrößen ql und s in den Monitoren BUFFERGUARD und PRIORITYLIST identifiziert werden:

Die Analogie von V und RELEASE (Blockspeicher) bzw. ENTER (Ordnungsstruktur) ergibt sich, wenn man die Monitore um die entsprechenden Zuweisungen auf der in der FIFO CLASS definierten Hilfsvariablen LENGTH erweitert (vgl. Kap. 6.1). Dann entspricht die Abfrage auf ql>0 der in den Monitoren gemachten Abfrage auf ¬(length=0).

Definiert man

```
Blockspeicher:      s := pointer-first
                    s=0 <=> pointer=first

Ordnungsstruktur: s := n mit list.first.succ(n) = extension
                         und, sofern n>0,
                             list.first.succ(n-1) in list
                    s=0 <=> list.first.succ(0) = extension
```

,so folgt die Analogie von P mit REQUEST (Blockspeicher) bzw. REMOVE (Ordnungsstruktur) wenn die Abfragen vor der Warteoperation denselben Effekt haben. Das gilt aber mit der

Behauptung:

Unmittelbar vor der Warteoperation in P gilt ql>s <=> s=0.

Beweis:

Unmittelbar vor P gilt die Invariante min(s,ql)=0.

Zwei Fälle sind zu unterscheiden:

a) ql=0 & s>=0 =>
   ql=1 & s>=0 vor der Warteoperation.
   => (ql>s => s=0)
   & (ql<=s => s<>0)

b) ql>=0 & s=0 =>
   ql>0 & s=0 vor der Warteoperation.
   => (ql>s <=> s=0)

q.e.d.

Da die Synchronisationsoperationen im vorliegenden Fall nicht auf Daten zugreifen, die das sequentielle Verhalten der Monitore bestimmen, ist die in dem letzten und diesem Kapitel durchgeführte getrennte Betrachtung gerechtfertigt.

Bemerkungen

1.) Zur Verifikation der betrachteten Monitore mit Hilfe der Beweisregeln (M1) bis (M4) ist eine Erweiterung um die Zuweisungen auf der Variablen QL unerläßlich. In einer Monitoroperation, die nicht in jedem Fall zum Warten führen soll, muß es vor der Warteoperation die Möglichkeit einer Zuweisung auf eine in die Wartebedingung eingehende Variable geben. Diesen Zweck erfüllt QL. Will man eine solche Einschränkung des Monitordesigns vermeiden, so darf die Monitorinvariante nicht mit der Eingangsbedingung der Warteoperation zusammenfallen. Dann kann man die Zuweisungen auf der Variablen QL vollständig in der Semantik der Synchronisationsoperationen verbergen und sie als von der Sprache zur Verfügung gestellte read-only Variable konzipieren (vgl. [How76b]).

2.) Die angegebene Semaphorenimplementation ist leicht aus dem Implementationsvorschlag in [How76a] herzuleiten, der auf Habermanns Charakterisierung der Semaphore basiert ([Hab72]). (Daraus ergibt sich insbesondere die wenig naheliegende Abfrage auf ql>s vor der Warteoperation in P. Sie stimmt wie bewiesen mit der an dieser Stelle sinnvollen Abfrage auf s=0 überein.)

   Die drei Größen

   na: Anzahl der versuchten P-Operationen
   np: Anzahl der ausgeführten P-Operationen
   nv: Anzahl der ausgeführten V-Operationen

werden reduziert zu

s := nv-np Semaphorenwert
ql := na-np Warteschlangenlänge

(In [How76a] kann nv allgemeiner mit einem Wert größer gleich Null initialisiert werden, der dem Anfangswert n von s entspricht.)

Trotz einer Reduktion der Variablen kann der Beweis analog durchgeführt werden. Die von Habermann zum Zwecke der Charakterisierung angegebene Definition der Semaphore läßt sich also implementationsgerecht mit weniger aber für die Implementation relevanteren Größen formulieren.

5.3 Scheduling Verhalten der Speicherstruktur

Die wichtigste sich auf die Verwendung des Speichers in COCO beziehende Aussage ist:

Beim Zugriff auf den Speicher kann keine Verklemmung auftreten.

Dazu ist das short-term und das mid-term scheduling bezüglich der zugreifenden Prozesse zu untersuchen:

Das short-term scheduling kann keine Verklemmung verursachen, denn es gibt keinen geschachtelten Monitorzugriff: Einen rekursiven Zugriff gestattet die Sprache nicht, und eine Zugriffshierarchie wurde beim Entwurf umgangen.

Beim mid-term scheduling gibt es zwei Bedingungen, die zum Warten von Prozessen führen können:

a) Bei einem Eingabeversuch ist kein Platz für einen weiteren Block im Speicher vorhanden. Diese Bedingung betrifft den Blockspeicher. Sie wird im BUFFERGUARD MONITOR behandelt.

b) Bei einem Ausgabeversuch ist kein geeigneter Ausgabeblock vorhanden, dh. die Prioritätsliste, auf der eine Ausgabe versucht wird, ist leer. Diese Bedingung wird im PRIORITYLIST MONITOR behandelt.

Damit sind die in Kap. 4.3, Bem. 3 angesprochenen Verklemmungsquellen (Monitor-Zugriffshierarchien und nicht problemgerechte Verteilung der mid-term scheduling Entscheidungen) vermieden worden.

Zur Überprüfung der obigen Argumentation wurden die Transferprozesse durch Simulationsprozesse ersetzt, die nur ihre Zugriffe auf die Speicherstruktur durchführen und über die Blöcke Auskunft geben, die sie übertragen. Auf diese Weise wurde ein Transfer über mehrere Leitungen mit und ohne Datenbehandlung einschließlich der Situation eines vollen Speichers simuliert. Da die Ein/Ausgabe auf dem Prozessor nachgebildet wurde, werden Realzeiteinflüsse durch die peripheren Geräte mit diesen Tests nicht abgedeckt.

6 Anhang

6.1 Das COCO System

```
" CHRISTIAN LENGAUER

  DATENVERARBEITUNG/MATHEMATIK
  HAHN-MEITNER-INSTITUT BERLIN

  THE COCO SYSTEM
   29 APRIL 1977  "


"####################
#                   #
#  I/O DEFINITIONS  #
#                   #
####################"


CONST NUL = '(:0:)';
      BEL = '(:7:)';
      NL  = '(:10:)';
      FF  = '(:12:)';
      CR  = '(:13:)';
      EM  = '(:25:)';

CONST IDLENGTH   = 12; "LENGTH OF IDENTIFIER"
      LINELENGTH = 72; "LENGTH OF TEXT LINE"

TYPE IDENTIFIER = ARRAY (.1..IDLENGTH.) OF CHAR;
     LINE       = ARRAY (.1..LINELENGTH.) OF CHAR;

TYPE IODEVICE = ("PERIPHERAL DEVICES:"
                  TYPEDEVICE,DISKDEVICE,TAPEDEVICE,
                  PRINTDEVICE,CARDDEVICE,
                 "DATA TERMINAL EQUIPMENTS:"
                  TERMINAL1,TERMINAL2,TERMINAL3,
                  TERMINAL4,TERMINAL5,
                  PDP1,PDP2,PDP3);

CONST FIRSTTERMINAL  = TERMINAL1;
      LASTTERMINAL   = TERMINAL5;
      FIRSTPDP       = PDP1;
      LASTPDP        = PDP3;
      FIRSTCOMPONENT = FIRSTTERMINAL;
      LASTCOMPONENT  = LASTPDP;
      TRANSFERPROCS  = 16;
                     "= NUMBER OF TRANSFER PROCESSES - 1"
                     "= NUMBER OF LINE PROCESSES"
```

```
TYPE TERMINALCOMPONENTS = FIRSTTERMINAL..LASTTERMINAL;
     PDPCOMPONENTS      = FIRSTPDP..LASTPDP;
     COMPONENTS         = FIRSTCOMPONENT..LASTCOMPONENT;

TYPE IOOPERATION = (INPUT,OUTPUT,MOVE,CONTROL);
     IOARG       = (WRITEEOF,REWIND,UPSPACE,BACKSPACE);
     IORESULT    = (COMPLETE,INTERVENTION,TRANSMISSION,FAILURE,
                    ENDFILE,ENDMEDIUM,STARTMEDIUM);

     IOPARAM     = RECORD OPERATION: IOOPERATION;
                          STATUS: IORESULT;
                          ARG: IOARG
                   END;

CONST PROCESSCOUNT = 17;
                   "= MAXIMAL NUMBER OF PROCESSES
                      SHARING A DEVICE - 1"
                   "= NUMBER OF TRANSFER PROCESSES,
                      SINCE AT MOST THEY SHARE THE CONSOLE
                      TOGETHER WITH THE OPERATOR PROCESS"

TYPE PROCESSQUEUE = ARRAY (.1..PROCESSCOUNT.) OF QUEUE;


"############################
#                          #
#  REAL-TIME DEFINITIONS   #
#                          #
############################"


CONST SYSTEMCOUNT =  1.0; "SYSTEM COUNT STEP"
      MAXCOUNT    = 1E12; "MAX SYSTEM COUNT"
      CLOCKTICK   =  1.0; "KERNEL TICK FREQUENCY"
      SYSTEMTICK  =  1.0; "SYSTEM TICK FREQUENCY"


"##############################
#                            #
#  SYSTEM FORMAT DEFINITIONS #
#                            #
##############################"


CONST MAXINT = 32767;

      MINPRIORITY = 100; "LOWEST BLOCKPRIORITY"
      MAXPRIORITY =   0; "HIGHEST BLOCKPRIORITY"
      MAXLENGTH   = 256; "MAX LENGTH OF BLOCKDATA"

      MININDEX    =   1; "MIN BUFFER INDEX"
      MAXINDEX    =  50; "MAX BUFFER INDEX"

TYPE BUFFERINDEX  = MININDEX..MAXINDEX;

TYPE DONTKNOW = INTEGER;
```

```
      WORD      = INTEGER;

TYPE  BLOCKKIND       = (TRAFFIC,SYSTEM,LOGIC,DATA,CHANNEL);
      SYSTEMERROR     = LOGIC..CHANNEL;
      TASKNAME        = DONTKNOW;

      BLOCKADDRESS    = RECORD LINE: COMPONENTS;
                               TASK: TASKNAME
                        END;

      BLOCKNAME       = DONTKNOW;
      BLOCKPRIORITY   = MAXPRIORITY..MINPRIORITY;
      DATAHANDLERS    = (DUMMYHANDLER);
      BLOCKHANDLING   = SET OF DATAHANDLERS;
      BLOCKERROR      = (FORGET,SAVE,MESSAGE);
      BLOCKLENGTH     = 1..MAXLENGTH;
      BLOCKDATA       = ARRAY (.BLOCKLENGTH.) OF WORD;

      BLOCK           = RECORD KIND: BLOCKKIND;
                               SRC, DEST: BLOCKADDRESS;
                               NAME: BLOCKNAME;
                               PRIORITY: BLOCKPRIORITY;
                               SYSTEMPRIORITY: REAL;
                               HANDLING: BLO KHANDLING;
                               ERROR: BLOCKERROR;
                               LENGTH: BLOCKLENGTH;
                               DATA: BLOCKDATA
                        END;


"####################
#                    #
#   ARRAY CONTROL:   #
#                    #
#   FIFO  CLASS      #
#   STACK CLASS      #
#                    #
####################"


TYPE  FIFO = CLASS (LIMIT: INTEGER);

      VAR HEAD, TAIL, LENGTH: INTEGER;

      FUNCTION ENTRY ARRIVAL: INTEGER;
      BEGIN ARRIVAL := TAIL;
            TAIL := TAIL MOD LIMIT + 1;
            LENGTH := LENGTH + 1;
      END;

      FUNCTION ENTRY DEPARTURE: INTEGER;
      BEGIN DEPARTURE := HEAD;
            HEAD := HEAD MOD LIMIT + 1;
            LENGTH := LENGTH - 1;
      END;
```

```
      FUNCTION ENTRY EMPTY: BOOLEAN;
      BEGIN EMPTY := (LENGTH = 0);
      END;

      FUNCTION ENTRY FULL: BOOLEAN;
      BEGIN FULL := (LENGTH = LIMIT);
      END;

      BEGIN HEAD := 1; TAIL := 1; LENGTH := 0;
      END;

TYPE STACK = CLASS (FIRST, LAST: INTEGER);

      VAR POINTER: INTEGER;

      FUNCTION ENTRY PUSH: INTEGER;
      BEGIN PUSH := POINTER;
            POINTER := POINTER + 1;
      END;

      FUNCTION ENTRY POP: INTEGER;
      BEGIN POINTER := POINTER - 1;
            POP := POINTER;
      END;

      FUNCTION ENTRY EMPTY: BOOLEAN;
      BEGIN EMPTY := (POINTER = FIRST);
      END;

      FUNCTION ENTRY FULL: BOOLEAN;
      BEGIN FULL := (POINTER = LAST + 1);
      END;

      BEGIN POINTER := LAST + 1;
      END;


"######################
#                      #
#  CONSOLE I/O:        #
#                      #
#  BELLKEY    CLASS    #
#  TYPEWRITER CLASS    #
#  TELETYPE   CLASS    #
#                      #
######################"


TYPE BELLKEY = CLASS (UNIT: IODEVICE);

      VAR PARAM: IOPARAM;

      PROCEDURE ENTRY AWAIT;
      BEGIN IO(PARAM,PARAM,UNIT);
      END;
```

```
      BEGIN PARAM_OPERATION := CONTROL;
      END;

TYPE TYPEWRITER = CLASS (UNIT: IODEVICE);

      PROCEDURE ENTRY WRITE (C: CHAR);
      VAR PARAM: IOPARAM; X: CHAR;
      BEGIN X := C;
            PARAM_OPERATION := OUTPUT;
            IO(X,PARAM,UNIT);
      END;

      PROCEDURE ENTRY READ (VAR C: CHAR);
      VAR PARAM: IOPARAM;
      BEGIN PARAM_OPERATION := INPUT;
            IO(C,PARAM,UNIT);
      END;

      BEGIN END;

TYPE TELETYPE = CLASS (UNIT: IODEVICE);

     TYPE NUMBER = ARRAY (_1__8_) OF CHAR;

      VAR DEVICE: TYPEWRITER;

      PROCEDURE ENTRY WRITE (C: CHAR);
      BEGIN DEVICE_WRITE(C);
      END;

      PROCEDURE ENTRY READ (VAR C: CHAR);
      BEGIN DEVICE_READ(C);
      END;

      PROCEDURE ENTRY WRITETEXT (TEXT: LINE);
      VAR I: INTEGER; C: CHAR;
      BEGIN I := 1; C := TEXT(_1_);
            WHILE C <> NUL DO
              BEGIN DEVICE_WRITE(C);
                    I := I + 1;
                    C := TEXT(_I_)
              END;
      END;

      PROCEDURE ENTRY READID (VAR ID: IDENTIFIER);
      VAR LENGTH: INTEGER; C: CHAR;
      BEGIN ID := '              ';
            LENGTH := 0;
            REPEAT DEVICE_READ(C);
                   LENGTH := LENGTH + 1;
                   ID(_LENGTH_) := C;
             UNTIL (C = NL) OR (LENGTH = IDLENGTH);
            IF LENGTH < IDLENGTH
               THEN ID(_LENGTH_) := ' '
               ELSE DEVICE_WRITE(NL);
      END;
```

```
PROCEDURE ENTRY WRITEINT (INT: UNIV INTEGER);
VAR DIGITS: NUMBER;
    REM, LENGTH, ZERO: INTEGER;
BEGIN REM := INT; LENGTH := 0; ZERO := ORD('0');
      REPEAT LENGTH := LENGTH + 1;
             DIGITS(.LENGTH.) :=
               CHR(ABS(REM MOD 10) + ZERO);
             REM := REM DIV 10
       UNTIL REM = 0;
      LENGTH := LENGTH + 1;
      IF INT < 0 THEN DIGITS(.LENGTH.) := '-'
                 ELSE DIGITS(.LENGTH.) := ' ';
      FOR LENGTH := LENGTH DOWNTO 1
       DO DEVICE.WRITE(DIGITS(.LENGTH.));
END;

PROCEDURE ENTRY READINT (VAR INT: UNIV INTEGER);
VAR DIGITS: NUMBER; C: CHAR;
    FAC, LENGTH, ZERO: INTEGER; ORDER: REAL;
BEGIN FAC := 0; LENGTH := 0;
      ZERO := ORD('0'); ORDER := 1.0;
      REPEAT DEVICE.READ(C);
             LENGTH := LENGTH + 1;
             DIGITS(.LENGTH.) := C
       UNTIL (C = NL) OR (LENGTH = 8);
      IF DIGITS(.1.) = NL
         THEN INT := 0
         ELSE BEGIN
                FOR LENGTH := LENGTH - 1 DOWNTO 2 DO
                  BEGIN
                    FAC := (ORD(DIGITS(.LENGTH.)) - ZERO)
                           * TRUNC(ORDER) + FAC;
                    ORDER := ORDER * 10.0;
                  END;
                IF NOT (DIGITS(.1.) IN (.'+','-'.))
                   THEN FAC := (ORD(DIGITS(.1.)) - ZERO)
                               * TRUNC(ORDER) + FAC
                   ELSE IF DIGITS(.1.) = '-'
                           THEN FAC := -FAC;
                INT := FAC;
              END;
END;

BEGIN INIT DEVICE(UNIT);
END;
```

```
"############################
#                          #
#  VIRTUAL DEVICE CONTROL: #
#                          #
#  RESOURCE MONITOR        #
#                          #
############################"

TYPE RESOURCE = MONITOR

     VAR FREE: BOOLEAN;
         Q: PROCESSQUEUE;
         NEXT: FIFO;

     PROCEDURE ENTRY REQUEST;
     BEGIN IF FREE
              THEN FREE := FALSE
              ELSE DELAY(Q(_NEXT_ARRIVAL_));
     END;

     PROCEDURE ENTRY RELEASE;
     BEGIN IF NEXT_EMPTY
              THEN FREE := TRUE
              ELSE CONTINUE(Q(_NEXT_DEPARTURE_));
     END;

     BEGIN FREE := TRUE;
           INIT NEXT(PROCESSCOUNT);
     END;


"##########################
#                        #
#  MEMORY MANAGEMENT:    #
#                        #
#  SHAREDBUFFER  MONITOR #
#  BUFFERGUARD   MONITOR #
#  LISTSTRUCTURE MONITOR #
#  BLOCKSTREAM   CLASS   #
#                        #
##########################"

TYPE SHAREDBUFFER = MONITOR

     VAR POOL: ARRAY (_BUFFERINDEX_) OF BLOCK; "BLOCK BUFFER"

     PROCEDURE ENTRY PUT (ELEMENT: BLOCK;
                          ELEMENTINDEX: BUFFERINDEX);
     BEGIN POOL(_ELEMENTINDEX_) := ELEMENT;
     END;

     PROCEDURE ENTRY GET (VAR ELEMENT: BLOCK;
                          ELEMENTINDEX: BUFFERINDEX);
     BEGIN ELEMENT := POOL(_ELEMENTINDEX_);
```

```
      END;

      BEGIN END;

TYPE BUFFERGUARD = MONITOR

      VAR FREELIST: ARRAY (.BUFFERINDEX.) OF BUFFERINDEX;
                    "BUFFER ADMINISTRATION LIST"
          NEW: STACK; "ADMINISTRATION LIST CONTROL"
          ON_FULL: ARRAY (.1..TRANSFERPROCS.) OF QUEUE;
                   "WAITING QUEUE"
          NEXT: FIFO; "WAITING QUEUE CONTROL"
          I: BUFFERINDEX;

      PROCEDURE ENTRY REQUEST (VAR ELEMENTINDEX: BUFFERINDEX);
      BEGIN IF NEW.EMPTY THEN DELAY(ON_FULL(.NEXT.ARRIVAL.));
            ELEMENTINDEX := FREELIST(.NEW.POP.);
      END;

      PROCEDURE ENTRY RELEASE (ELEMENTINDEX: BUFFERINDEX);
      BEGIN FREELIST(.NEW.PUSH.) := ELEMENTINDEX;
            IF NOT NEXT.EMPTY "IF ANY PROCESS WAITING"
               THEN CONTINUE(ON_FULL(.NEXT.DEPARTURE.));
      END;

      FUNCTION ENTRY FULL: BOOLEAN; "FULL BUFFER CONDITION"
      BEGIN FULL := NEW.EMPTY;
      END;

      FUNCTION ENTRY EMPTY: BOOLEAN; "EMPTY BUFFER CONDITION"
      BEGIN EMPTY := NEW.FULL;
      END;

      BEGIN FOR I := MININDEX TO MAXINDEX
             DO FREELIST(.I.) := I; "FREELIST FULL"
            INIT NEW(MININDEX,MAXINDEX),
                 NEXT(TRANSFERPROCS);
      END;

--------- Ergänzung: Modulare Version der Ordnungsstruktur ---------

TYPE LISTSTRUCTURE = MONITOR

      TYPE PRIORITYLIST = CLASS

           CONST EXTENSION = 0;
                           "= MININDEX - 1"

           TYPE LISTINDEX       = EXTENSION..MAXINDEX;

                PRIORITYELEMENT = RECORD SYSTEMPRIORITY: REAL;
                                         SUCC: LISTINDEX
                                  END;

                PRIORITYQUEUE   = ARRAY (.LISTINDEX.)
                                    OF PRIORITYELEMENT;
```

```
    VAR LIST: PRIORITYQUEUE;
        ON_EMPTY: QUEUE;

    FUNCTION EMPTY: BOOLEAN; "EMPTY LIST CONDITION"
    BEGIN EMPTY := (LIST(.EXTENSION.).SUCC = EXTENSION);
    END;

    PROCEDURE ENTRY ENTER (ELEMENTINDEX: BUFFERINDEX;
                           PRIORITY: REAL);
    VAR I: LISTINDEX;
    BEGIN I := EXTENSION;
          WHILE LIST(.LIST(.I.).SUCC.).SYSTEMPRIORITY
                  <= PRIORITY
                "WHILE STILL ANY SUCCESSOR WITH
                 HIGHER SYSTEMPRIORITY..."
             DO I := LIST(.I.).SUCC; "...SCAN"
          LIST(.ELEMENTINDEX.).SYSTEMPRIORITY := PRIORITY;
          LIST(.ELEMENTINDEX.).SUCC := LIST(.I.).SUCC;
          LIST(.I.).SUCC := ELEMENTINDEX; "INSERT ELEMENT"
          CONTINUE(ON_EMPTY);
    END;

    PROCEDURE ENTRY REMOVE (VAR ELEMENTINDEX: BUFFERINDEX);
    BEGIN IF EMPTY THEN DELAY(ON_EMPTY);
          ELEMENTINDEX := LIST(.EXTENSION.).SUCC; "GET AND..."
          LIST(.EXTENSION.).SUCC := "...FORGET LIST HEAD"
            LIST(.LIST(.EXTENSION.).SUCC.).SUCC;
    END;

    BEGIN LIST(.EXTENSION.).SYSTEMPRIORITY := MAXCOUNT;
          LIST(.EXTENSION.).SUCC := EXTENSION;
    END;

VAR INLIST: PRIORITYLIST;
    OUTLIST: ARRAY (.COMPONENTS.) OF PRIORITYLIST;
    COMP: COMPONENTS;

PROCEDURE ENTRY INENTER (ELEMENTINDEX: BUFFERINDEX;
                         PRIORITY: REAL);
BEGIN INLIST.ENTER(ELEMENTINDEX,PRIORITY);
END;

PROCEDURE ENTRY INREMOVE (VAR ELEMENTINDEX: BUFFERINDEX);
BEGIN INLIST.REMOVE(ELEMENTINDEX);
END;

PROCEDURE ENTRY OUTENTER (ELEMENTINDEX: BUFFERINDEX;
                          PRIORITY: REAL; LINE: COMPONENTS);
BEGIN OUTLIST(.LINE.).ENTER(ELEMENTINDEX,PRIORITY);
END;

PROCEDURE ENTRY OUTREMOVE (VAR ELEMENTINDEX: BUFFERINDEX;
                           LINE: COMPONENTS);
BEGIN OUTLIST(.LINE.).REMOVE(ELEMENTINDEX);
END;
```

```
    BEGIN INIT INLIST;
          FOR COMP := FIRSTCOMPONENT TO LASTCOMPONENT
           DO INIT OUTLIST(.COMP.);
    END;
```

```
TYPE LISTSTRUCTURE = MONITOR

    CONST EXTENSION = 0;
                    "= MININDEX - 1"

    TYPE LISTINDEX        = EXTENSION..MAXINDEX;

         PRIORITYELEMENT = RECORD SYSTEMPRIORITY: REAL;
                                  SUCC: LISTINDEX
                           END;

         PRIORITYQUEUE    = ARRAY (.LISTINDEX.)
                              OF PRIORITYELEMENT;

    VAR LIST: PRIORITYQUEUE;
        INPUTLIST: LISTINDEX; "HEAD OF INPUT LIST"
        OUTPUTLIST: ARRAY (.COMPONENTS.) OF LISTINDEX;
                    "HEADS OF OUTPUT LISTS"
        ON_INEMPTY: QUEUE; "WAITING QUEUES"
        ON_OUTEMPTY: ARRAY (.COMPONENTS.) OF QUEUE;
        I: LISTINDEX; "AUXILIARY VARIABLES"
        COMP: COMPONENTS;

    FUNCTION INEMPTY: BOOLEAN; "EMPTY INPUT LIST CONDITION"
    BEGIN INEMPTY := (INPUTLIST = EXTENSION);
    END;

    FUNCTION OUTEMPTY (LINE: COMPONENTS): BOOLEAN;
             "EMPTY OUTPUT LIST CONDITION"
    BEGIN OUTEMPTY := (OUTPUTLIST(.LINE.) = EXTENSION);
    END;

    PROCEDURE INSERT (VAR LISTHEAD: LISTINDEX;
                      ELEMENTINDEX: BUFFERINDEX; PRIORITY: REAL);
    BEGIN LIST(.EXTENSION.).SUCC := LISTHEAD;
          I := EXTENSION;
          WHILE LIST(.LIST(.I.).SUCC.).SYSTEMPRIORITY
                  <= PRIORITY
                "WHILE STILL ANY SUCCESSOR WITH
                 HIGHER SYSTEMPRIORITY..."
             DO I := LIST(.I.).SUCC; "...SCAN"
          LIST(.ELEMENTINDEX.).SYSTEMPRIORITY := PRIORITY;
          LIST(.ELEMENTINDEX.).SUCC := LIST(.I.).SUCC;
          LIST(.I.).SUCC := ELEMENTINDEX; "INSERT ELEMENT"
          LISTHEAD := LIST(.EXTENSION.).SUCC;
    END;

    PROCEDURE EXTRACT (VAR LISTHEAD: LISTINDEX;
```

```
                        VAR ELEMENTINDEX: BUFFERINDEX);
    BEGIN ELEMENTINDEX := LISTHEAD;          "GET AND..."
          LISTHEAD := LIST(.LISTHEAD.).SUCC; "...FORGET LIST HEAD"
    END;

    PROCEDURE ENTRY INENTER (ELEMENTINDEX: BUFFERINDEX;
                             PRIORITY: REAL);
    BEGIN INSERT(INPUTLIST,ELEMENTINDEX,PRIORITY);
          CONTINUE(ON_INEMPTY);
    END;

    PROCEDURE ENTRY INREMOVE (VAR ELEMENTINDEX: BUFFERINDEX);
    BEGIN IF INEMPTY THEN DELAY(ON_INEMPTY);
          EXTRACT(INPUTLIST,ELEMENTINDEX);
    END;

    PROCEDURE ENTRY OUTENTER (ELEMENTINDEX: BUFFERINDEX;
                              PRIORITY: REAL; LINE: COMPONENTS);
    BEGIN INSERT(OUTPUTLIST(.LINE.),ELEMENTINDEX,PRIORITY);
          CONTINUE(ON_OUTEMPTY(.LINE.));
    END;

    PROCEDURE ENTRY OUTREMOVE (VAR ELEMENTINDEX: BUFFERINDEX;
                               LINE: COMPONENTS);
    BEGIN IF OUTEMPTY(LINE) THEN DELAY(ON_OUTEMPTY(.LINE.));
          EXTRACT(OUTPUTLIST(.LINE.),ELEMENTINDEX);
    END;

    BEGIN INPUTLIST := EXTENSION; "LISTS EMPTY"
          FOR COMP := FIRSTCOMPONENT TO LASTCOMPONENT
           DO OUTPUTLIST(.COMP.) := EXTENSION;
          LIST(.EXTENSION.).SYSTEMPRIORITY := MAXCOUNT;
    END;

TYPE BLOCKSTREAM = CLASS (SYSTEMBUFFER: SHAREDBUFFER;
                          FRAME: BUFFERGUARD;
                          SYSTEMLIST: LISTSTRUCTURE);

    VAR ELEMENTINDEX: BUFFERINDEX;

    PROCEDURE ENTRY ENTER (ELEMENT: BLOCK; LIST: IOOPERATION);
    BEGIN FRAME.REQUEST(ELEMENTINDEX);
          SYSTEMBUFFER.PUT(ELEMENT,ELEMENTINDEX);
          CASE LIST
            OF INPUT:  SYSTEMLIST.INENTER(ELEMENTINDEX,
                                          ELEMENT.SYSTEMPRIORITY);
               OUTPUT: SYSTEMLIST.OUTENTER(ELEMENTINDEX,
                                           ELEMENT.SYSTEMPRIORITY,
                                           ELEMENT.DEST.LINE)
          END;
    END;

    PROCEDURE ENTRY GET (VAR ELEMENT: BLOCK);
    BEGIN SYSTEMLIST.INREMOVE(ELEMENTINDEX);
          SYSTEMBUFFER.GET(ELEMENT,ELEMENTINDEX);
    END;
```

```
      PROCEDURE ENTRY PUT (ELEMENT: BLOCK);
      BEGIN SYSTEMBUFFER.PUT(ELEMENT,ELEMENTINDEX);
            SYSTEMLIST.OUTENTER(ELEMENTINDEX,
                                ELEMENT.SYSTEMPRIORITY,
                                ELEMENT.DEST.LINE);
      END;

      PROCEDURE ENTRY REMOVE (VAR ELEMENT: BLOCK;
                              LINE: COMPONENTS);
      BEGIN SYSTEMLIST.OUTREMOVE(ELEMENTINDEX,LINE);
            SYSTEMBUFFER.GET(ELEMENT,ELEMENTINDEX);
      END;

      PROCEDURE ENTRY CLOSE;
      BEGIN FRAME.RELEASE(ELEMENTINDEX);
      END;

      BEGIN END;


"##########################
#                        #
#   REAL-TIME CONTROL:   #
#                        #
#   DYNPRIORITY  MONITOR #
#   TIMERPROCESS PROCESS #
#                        #
##########################"


TYPE DYNPRIORITY = MONITOR

      VAR DYNCOUNT: REAL;

      PROCEDURE ENTRY INCREMENT (PRIORITYSTEP: REAL);
      BEGIN DYNCOUNT := DYNCOUNT + PRIORITYSTEP;
      END;

      FUNCTION ENTRY COUNT: REAL;
      BEGIN COUNT := DYNCOUNT;
      END;

      BEGIN DYNCOUNT := 0.0;
      END;

TYPE TIMERPROCESS = PROCESS (TICKSTEP, PRIORITYSTEP: REAL;
                             DYN: DYNPRIORITY);

      VAR COUNT: REAL;

      BEGIN CYCLE COUNT := 0.0;
                  REPEAT WAIT;
                         COUNT := COUNT + CLOCKTICK
                   UNTIL COUNT >= TICKSTEP;
                  DYN.INCREMENT(PRIORITYSTEP)
```

```
            END;
      END;


"##########################
#                         #
#  ERROR HANDLING:        #
#                         #
#  STATETABLE   MONITOR   #
#  ERRORHANDLER CLASS     #
#  LINEREPORT   CLASS     #
#  OPERATOR     PROCESS   #
#                         #
##########################"


TYPE STATETABLE = MONITOR

      VAR TABLE: ARRAY (.COMPONENTS.) OF BOOLEAN;
          I: COMPONENTS;

      PROCEDURE ENTRY SETSTATE (LINE: COMPONENTS;
                                RESULT: BOOLEAN);
      BEGIN TABLE(.LINE.) := RESULT;
      END;

      FUNCTION ENTRY CONTAIN (LINE: COMPONENTS): BOOLEAN;
      BEGIN CONTAIN := TABLE(.LINE.);
      END;

      BEGIN FOR I := FIRSTCOMPONENT TO LASTCOMPONENT
            DO TABLE(.I.) := TRUE;
      END;

TYPE ERRORHANDLER = CLASS (SYSTEMSTREAM: BLOCKSTREAM;
                           DYN: DYNPRIORITY);

      PROCEDURE ENTRY HANDLE (VAR ELEMENT: BLOCK; CODE: SYSTEMERROR;
                              CALLER: IOOPERATION);
      BEGIN WITH ELEMENT
              DO CASE ERROR
                   OF FORGET:  CASE CALLER
                                 OF CONTROL,
                                    OUTPUT:  SYSTEMSTREAM.CLOSE
                               END;
                      SAVE,
                      MESSAGE: BEGIN
                                 SYSTEMPRIORITY :=
                                   CONV(PRIORITY) + DYN.COUNT;
                                 DEST := SRC;
                                 CASE ERROR
                                   OF SAVE:    KIND := CODE;
                                      MESSAGE: BEGIN
                                                 KIND := SYSTEM;
                                                 "SET UP MESSAGE IN
                                                  BLOCK DATA FIELD
```

462

```
                                        ACCORDING TO CODE
                                        PARAMETER"
                                    END
                                END;
                                CASE CALLER
                                  OF INPUT:   SYSTEMSTREAM.ENTER
                                                (ELEMENT,OUTPUT);
                                      CONTROL,
                                      OUTPUT:  SYSTEMSTREAM.PUT
                                                (ELEMENT)
                                END;
                              END
                    END;
          END;

          BEGIN END;

TYPE LINEREPORT = CLASS (OPERATOR: TELETYPE);

          PROCEDURE ENTRY LINEID (COMP: COMPONENTS; VAR ID: IDENTIFIER);
          BEGIN CASE COMP
                  OF TERMINAL1: ID := 'TERMINAL1   (:0:)';
                     TERMINAL2: ID := 'TERMINAL2   (:0:)';
                     TERMINAL3: ID := 'TERMINAL3   (:0:)';
                     TERMINAL4: ID := 'TERMINAL4   (:0:)';
                     TERMINAL5: ID := 'TERMINAL5   (:0:)';
                     PDP1:      ID := 'PDP1        (:0:)';
                     PDP2:      ID := 'PDP2        (:0:)';
                     PDP3:      ID := 'PDP3        (:0:)'
                END;
          END;

          PROCEDURE ENTRY IDLINE (ID: IDENTIFIER; VAR COMP: COMPONENTS;
                                  VAR SUCCESS: BOOLEAN);
          BEGIN SUCCESS := TRUE;
                IF ID = 'TERMINAL1    ' THEN COMP := TERMINAL1 ELSE
                IF ID = 'TERMINAL2    ' THEN COMP := TERMINAL2 ELSE
                IF ID = 'TERMINAL3    ' THEN COMP := TERMINAL3 ELSE
                IF ID = 'TERMINAL4    ' THEN COMP := TERMINAL4 ELSE
                IF ID = 'TERMINAL5    ' THEN COMP := TERMINAL5 ELSE
                IF ID = 'PDP1         ' THEN COMP := PDP1      ELSE
                IF ID = 'PDP2         ' THEN COMP := PDP2      ELSE
                IF ID = 'PDP3         ' THEN COMP := PDP3      ELSE
                SUCCESS := FALSE;
          END;

          BEGIN END;

TYPE OPERATORPROCESS = PROCESS (TYPEUSE: RESOURCE;
                                CHANNELS: STATETABLE);

          VAR OPERATOR: TELETYPE;
              BELL: BELLKEY;
              COMP: LINEREPORT;
              COMMAND: IDENTIFIER;
              I: COMPONENTS;
```

```
PROCEDURE HELP;
BEGIN OPERATOR.WRITETEXT('            ILL CMD! (:10:)(:0:)');
END;

PROCEDURE LOOKUP (LINE: COMPONENTS);
BEGIN IF CHANNELS.CONTAIN(LINE)
         THEN OPERATOR.WRITE('+')
         ELSE OPERATOR.WRITE('-');
END;

PROCEDURE STATE;
VAR SAVE: IDENTIFIER;
BEGIN FOR I := FIRSTCOMPONENT TO LASTCOMPONENT DO
        BEGIN LOOKUP(I);
              COMP.LINEID(I,SAVE);
              WITH OPERATOR DO
                BEGIN WRITETEXT(SAVE);
                      WRITE(NL);
                END;
        END;
END;

PROCEDURE SETCHANNEL (CH: IDENTIFIER);
VAR B, SUCCESS: BOOLEAN;
    N: INTEGER; SAVE: IDENTIFIER;
BEGIN CASE CH(.1.)
        OF '+': B := TRUE;
           '-': B := FALSE
      END;
      FOR N := 2 TO IDLENGTH
       DO SAVE(.N-1.) := CH(.N.);
      SAVE(.IDLENGTH.) := ' ';
      COMP.IDLINE(SAVE,I,SUCCESS);
      IF SUCCESS
         THEN CHANNELS.SETSTATE(I,B)
         ELSE HELP;
END;

BEGIN INIT OPERATOR(TYPEDEVICE), BELL(TYPEDEVICE);
      INIT COMP(OPERATOR);
      TYPEUSE.REQUEST;
      WITH OPERATOR DO
        BEGIN WRITETEXT('SYSTEM:(:10:)(:10:)(:0:)');
              WRITETEXT('STATE (:10:)-----(:10:)(:0:)');
              STATE; WRITE(NL);
        END;
      TYPEUSE.RELEASE;
      CYCLE REPEAT
              TYPEUSE.REQUEST;
              WITH OPERATOR DO
                BEGIN
                  WRITETEXT('OPERATOR>  (:0:)');
                  READID(COMMAND);
                  IF COMMAND = 'STATE       ' THEN STATE
                  ELSE IF COMMAND(.1.) IN (.'+','-'.) THEN
```

```
                        SETCHANNEL(COMMAND)
                      ELSE IF COMMAND <> '                ' THEN HELP;
                    END;
                  TYPEUSE.RELEASE;
                UNTIL COMMAND = '                ';
                BELL.AWAIT;
            END;
      END;


"################################
#                               #
#   TRANSFER PROCESSES:         #
#                               #
#   CHANNEL INDEPENDENT:        #
#   SYSTEMGUARD         CLASS   #
#   DATAPROCESS         PROCESS #
#                               #
#   TERMINAL CHANNEL:           #
#   TERMINALCONVERSION CLASS    #
#   TERMINALDRIVER      CLASS   #
#   TERMINALINPUT       PROCESS #
#   TERMINALOUTPUT      PROCESS #
#                               #
#   PDP CHANNEL:                #
#   PDPINPUT            PROCESS #
#   PDPOUTPUT           PROCESS #
#                               #
################################"


TYPE SYSTEMGUARD = CLASS (SYSTEMBUFFER: SHAREDBUFFER;
                          BUFFERCONTROL: BUFFERGUARD;
                          SYSTEMLIST: LISTSTRUCTURE;
                          DYN: DYNPRIORITY; CHANNELS: STATETABLE;
                          TYPEUSE: RESOURCE);

      VAR FAULT: ERRORHANDLER;
          OPERATOR: TELETYPE;
          SYSTEMSTREAM: BLOCKSTREAM;
          COMP: LINEREPORT;

      PROCEDURE ENTRY ENTER (VAR ELEMENT: BLOCK; LINE: COMPONENTS);
      BEGIN WITH ELEMENT DO
              BEGIN
                SRC.LINE := LINE;
                IF (PRIORITY < MAXPRIORITY)
                OR (PRIORITY > MINPRIORITY)
                   THEN PRIORITY := MINPRIORITY;
                IF NOT (ERROR IN (.FORGET,SAVE,MESSAGE.))
                   THEN ERROR := FORGET;
                IF  (DEST.LINE IN (.TERMINAL1,TERMINAL2,TERMINAL3,
                     TERMINAL4,TERMINAL5,PDP1,PDP2,PDP3.))
                AND (HANDLING <= (.DUMMYHANDLER.))
                AND ((LENGTH > 0) AND (LENGTH < MAXLENGTH))
                    THEN IF CHANNELS.CONTAIN(DEST.LINE)
```

```
                                THEN BEGIN
                                       SYSTEMPRIORITY :=
                                         CONV(PRIORITY) + DYN.COUNT;
                                       KIND := TRAFFIC;
                                       IF HANDLING <> (..)
                                          THEN SYSTEMSTREAM.ENTER
                                                 (ELEMENT,INPUT)
                                          ELSE SYSTEMSTREAM.ENTER
                                                 (ELEMENT,OUTPUT);
                                    END
                                ELSE FAULT.HANDLE(ELEMENT,CHANNEL,INPUT)
                        ELSE FAULT.HANDLE(ELEMENT,LOGIC,INPUT);
                END;
        END;

        PROCEDURE ENTRY CHECK (VAR ELEMENT: BLOCK;
                               SUCCESS: BOOLEAN);
        VAR ID: IDENTIFIER;
        BEGIN IF NOT SUCCESS
                  THEN BEGIN
                         IF CHANNELS.CONTAIN(ELEMENT.DEST.LINE) THEN
                           BEGIN
                             TYPEUSE.REQUEST;
                             WITH OPERATOR DO
                               BEGIN
                                 WRITETEXT('SYSTEM: -(:0:)');
                                 COMP.LINEID(ELEMENT.DEST.LINE,ID);
                                 WRITETEXT(ID); WRITE(NL);
                               END;
                             TYPEUSE.RELEASE;
                           END;
                         CHANNELS.SETSTATE(ELEMENT.DEST.LINE,FALSE);
                         FAULT.HANDLE(ELEMENT,CHANNEL,OUTPUT);
                       END
                  ELSE SYSTEMSTREAM.CLOSE;
        END;

        PROCEDURE ENTRY REMOVE (VAR ELEMENT: BLOCK;
                                LINE: COMPONENTS);
        BEGIN SYSTEMSTREAM.REMOVE(ELEMENT,LINE);
        END;

        BEGIN INIT SYSTEMSTREAM(SYSTEMBUFFER,BUFFERCONTROL,SYSTEMLIST),
                   FAULT(SYSTEMSTREAM,DYN),
                   OPERATOR(TYPEDEVICE);
              INIT COMP(OPERATOR);
        END;

TYPE DATAPROCESS = PROCESS (SYSTEMBUFFER: SHAREDBUFFER;
                            BUFFERCONTROL: BUFFERGUARD;
                            SYSTEMLIST: LISTSTRUCTURE);

        VAR ELEMENT: BLOCK;
            SYSTEMSTREAM: BLOCKSTREAM;

        BEGIN INIT SYSTEMSTREAM(SYSTEMBUFFER,BUFFERCONTROL,SYSTEMLIST);
```

```
            CYCLE SYSTEMSTREAM_GET(ELEMENT);
                  SYSTEMSTREAM_PUT(ELEMENT);
            END;
      END;

TYPE BLOCKCHAR = ARRAY (_1__2_) OF CHAR;

TYPE TERMINALCONVERSION = CLASS

      PROCEDURE ENTRY STRETCH (C: CHAR; VAR BC: UNIV BLOCKCHAR);
      BEGIN BC(_1_) := C;
            BC(_2_) := NUL;
      END;

      PROCEDURE ENTRY SHRINK (VAR C: CHAR; BC: UNIV BLOCKCHAR);
      BEGIN C := BC(_1_);
      END;

      BEGIN END;

TYPE TERMINALDRIVER = CLASS (DEVICE: TELETYPE);

      VAR CHARACTER: TERMINALCONVERSION;

      PROCEDURE ENTRY STRETCH (C: CHAR; VAR BC: UNIV BLOCKCHAR);
      BEGIN CHARACTER_STRETCH(C,BC);
      END;

      PROCEDURE ENTRY SHRINK (VAR C: CHAR; BC: UNIV BLOCKCHAR);
      BEGIN CHARACTER_SHRINK(C,BC);
      END;

      PROCEDURE ENTRY WRITEBLOCK (ELEMENT: BLOCK);
      VAR LENGTH: INTEGER; C: CHAR;
      BEGIN LENGTH := 0;
            REPEAT LENGTH := LENGTH + 1;
                   CHARACTER_SHRINK(C,ELEMENT_DATA(_LENGTH_));
                   DEVICE_WRITE(C);
             UNTIL LENGTH = ELEMENT_LENGTH;
      END;

      PROCEDURE ENTRY READBLOCK (VAR ELEMENT: BLOCK);
      VAR LENGTH: INTEGER; C: CHAR;
      BEGIN LENGTH := 0;
            REPEAT LENGTH := LENGTH + 1;
                   DEVICE_READ(C);
                   CHARACTER_STRETCH(C,ELEMENT_DATA(_LENGTH_));
             UNTIL (C = NL) OR (LENGTH = MAXLENGTH);
            ELEMENT_LENGTH := LENGTH;
      END;

      BEGIN INIT CHARACTER;
      END;

TYPE TERMINALINPUT = PROCESS (INPUTLINE: TERMINALCOMPONENTS;
                              SYSTEMBUFFER: SHAREDBUFFER;
```

```
                              BUFFERCONTROL: BUFFERGUARD;
                              SYSTEMLIST: LISTSTRUCTURE;
                              DYN: DYNPRIORITY;
                              CHANNELS: STATETABLE;
                              TYPEUSE, TERMINALUSE: RESOURCE);

VAR SYSTEM: SYSTEMGUARD;
    BELL: BELLKEY;
    TERMINAL: TERMINALDRIVER;
    DEVICE: TELETYPE;
    COMP: LINEREPORT;
    ELEMENT: BLOCK;
    ID: IDENTIFIER;
    LINE: COMPONENTS;
    C: CHAR; SUCCESS: BOOLEAN;

PROCEDURE COPYID (ID: IDENTIFIER; VAR ELEMENT: BLOCK);
VAR I: INTEGER;
BEGIN WITH TERMINAL, ELEMENT DO
        BEGIN FOR I := 1 TO IDLENGTH
                DO STRETCH(ID(.I.),DATA(.I.));
              LENGTH := IDLENGTH + 1;
              STRETCH(NL,DATA(.LENGTH.));
        END;
END;

BEGIN LINE := INPUTLINE;
      INIT SYSTEM(SYSTEMBUFFER,BUFFERCONTROL,SYSTEMLIST,
                  DYN, CHANNELS, TYPEUSE),
           DEVICE(LINE), BELL(LINE),
           TERMINAL(DEVICE), COMP(DEVICE);
      WITH DEVICE, TERMINAL, COMP, TYPEUSE, ELEMENT
        DO CYCLE
             BELL.AWAIT;
             REQUEST;
             WRITETEXT('TERMINAL> (:10:)(:0:)');
             REPEAT
               WRITETEXT('DESTINATION  (:0:)');
               READID(ID);
               IDLINE(ID,DEST.LINE,SUCCESS);
               IF NOT SUCCESS THEN WRITETEXT
                 ('              ILL DEST!(:10:)(:0:)');
             UNTIL SUCCESS;
             RELEASE;
             ERROR := SAVE;
             PRIORITY := MAXPRIORITY;
             LINEID(LINE,ID);
             COPYID(ID,ELEMENT);
             SYSTEM.ENTER(ELEMENT,LINE);
             REPEAT
               REQUEST;
               READBLOCK(ELEMENT);
               RELEASE;
               SYSTEM.ENTER(ELEMENT,LINE);
               SHRINK(C,DATA(.1.));
             UNTIL C = NL;
```

```
                END;
        END;

    TYPE TERMINALOUTPUT = PROCESS (OUTPUTLINE: TERMINALCOMPONENTS;
                                   SYSTEMBUFFER: SHAREDBUFFER;
                                   BUFFERCONTROL: BUFFERGUARD;
                                   SYSTEMLIST: LISTSTRUCTURE;
                                   DYN: DYNPRIORITY;
                                   CHANNELS: STATETABLE;
                                   TYPEUSE, TERMINALUSE: RESOURCE);

        VAR SYSTEM: SYSTEMGUARD;
            TERMINAL: TERMINALDRIVER;
            DEVICE: TELETYPE;
            ELEMENT: BLOCK;
            LINE: COMPONENTS;

        BEGIN LINE := OUTPUTLINE;
              INIT SYSTEM(SYSTEMBUFFER,BUFFERCONTROL,SYSTEMLIST,
                          DYN,CHANNELS,TYPEUSE),
                   DEVICE(LINE), TERMINAL(DEVICE);
              CYCLE SYSTEM.REMOVE(ELEMENT,LINE);
                    TERMINALUSE.REQUEST;
                    TERMINAL.WRITEBLOCK(ELEMENT);
                    TERMINALUSE.RELEASE;
                    SYSTEM.CHECK(ELEMENT,TRUE);
              END;
        END;

    TYPE PDPINPUT = PROCESS (INPUTLINE: PDPCOMPONENTS;
                             SYSTEMBUFFER: SHAREDBUFFER;
                             BUFFERCONTROL: BUFFERGUARD;
                             SYSTEMLIST: LISTSTRUCTURE;
                             DYN: DYNPRIORITY; CHANNELS: STATETABLE;
                             TYPEUSE: RESOURCE);

        VAR ELEMENT: BLOCK;
            SYSTEM: SYSTEMGUARD;
            PARAM: IOPARAM;
            LINE: COMPONENTS;

        BEGIN LINE := INPUTLINE;
              INIT SYSTEM(SYSTEMBUFFER,BUFFERCONTROL,SYSTEMLIST,
                          DYN,CHANNELS,TYPEUSE);
              PARAM.OPERATION := INPUT;
              CYCLE IO(ELEMENT,PARAM,LINE);
                    SYSTEM.ENTER(ELEMENT,LINE);
              END;
        END;

    TYPE PDPOUTPUT = PROCESS (OUTPUTLINE: PDPCOMPONENTS;
                              SYSTEMBUFFER: SHAREDBUFFER;
                              BUFFERCONTROL: BUFFERGUARD;
                              SYSTEMLIST: LISTSTRUCTURE;
                              DYN: DYNPRIORITY; CHANNELS: STATETABLE;
                              TYPEUSE: RESOURCE);
```

```
    VAR ELEMENT: BLOCK;
        SYSTEM: SYSTEMGUARD;
        PARAM: IOPARAM;
        LINE: COMPONENTS;
        SUCCESS: BOOLEAN;

    BEGIN LINE := OUTPUTLINE;
          INIT SYSTEM(SYSTEMBUFFER,BUFFERCONTROL,SYSTEMLIST,
                      DYN,CHANNELS,TYPEUSE);
          PARAM.OPERATION := OUTPUT;
          CYCLE SYSTEM.REMOVE(ELEMENT,LINE);
                IO(ELEMENT,PARAM,LINE);
                SUCCESS := (PARAM.STATUS <> FAILURE);
                SYSTEM.CHECK(ELEMENT,SUCCESS);
          END;
    END;


"####################
#                  #
#  INITIAL PROCESS #
#                  #
####################"


VAR TYPEUSE: RESOURCE;
    TERMINALUSE: ARRAY (.TERMINALCOMPONENTS.) OF RESOURCE;
    SYSTEMBUFFER: SHAREDBUFFER; BUFFERCONTROL: BUFFERGUARD;
    SYSTEMLIST: LISTSTRUCTURE;
    DYN: DYNPRIORITY; TIMER: TIMERPROCESS;
    CHANNELS: STATETABLE;
    DATAHANDLING: DATAPROCESS;
    INTERMINAL: ARRAY (.TERMINALCOMPONENTS.) OF TERMINALINPUT;
    OUTTERMINAL: ARRAY (.TERMINALCOMPONENTS.) OF TERMINALOUTPUT;
    INPDP: ARRAY (.PDPCOMPONENTS.) OF PDPINPUT;
    OUTPDP: ARRAY (.PDPCOMPONENTS.) OF PDPOUTPUT;
    OPERATOR: OPERATORPROCESS;
    COMP: COMPONENTS;

BEGIN INIT TYPEUSE;
      FOR COMP := FIRSTTERMINAL TO LASTTERMINAL
       DO INIT TERMINALUSE(.COMP.);
      INIT SYSTEMBUFFER, BUFFERCONTROL, SYSTEMLIST,
           DYN, TIMER(SYSTEMTICK,SYSTEMCOUNT,DYN),
           CHANNELS;
      INIT OPERATOR(TYPEUSE,CHANNELS),
           DATAHANDLING(SYSTEMBUFFER,BUFFERCONTROL,SYSTEMLIST);
      FOR COMP := FIRSTTERMINAL TO LASTTERMINAL
       DO INIT INTERMINAL(.COMP.)
                 (COMP,SYSTEMBUFFER,BUFFERCONTROL,SYSTEMLIST,
                  DYN,CHANNELS,TYPEUSE,TERMINALUSE(.COMP.)),
              OUTTERMINAL(.COMP.)
                 (COMP,SYSTEMBUFFER,BUFFERCONTROL,SYSTEMLIST,
                  DYN,CHANNELS,TYPEUSE,TERMINALUSE(.COMP.));
      FOR COMP := FIRSTPDP TO LASTPDP
```

```
        DO INIT INPDP(_COMP_)
                   (COMP,SYSTEMBUFFER,BUFFERCONTROL,SYSTEMLIST,
                    DYN,CHANNELS,TYPEUSE),
                OUTPDP(_COMP_)
                   (COMP,SYSTEMBUFFER,BUFFERCONTROL,SYSTEMLIST,
                    DYN,CHANNELS,TYPEUSE);
END_
```

6.2 Literatur

BaSch74 Baum, D.; Schroedter, H.-D.
EIN KOMMUNIKATIONSBETRIEBSSYSTEM FÜR EIN STERNFÖRMIGES RECHNERNETZ
GI - 4.Jahrestagung, Berlin 9.-12.10.1974,
Lecture Notes in Computer Science 26, G.Goos, E. Hartmanis (Eds.), Springer Verlag, Berlin, 1975, 428-435

BrHa73 Brinch-Hansen, P.
OPERATING SYSTEM PRINCIPLES
Prentice Hall, Englewood Cliffs, New Jersey, 1973, 366 pp.

BrHa75a Brinch-Hansen, P.
CONCURRENT PASCAL REPORT
Information Science, California Institute of Technology, June 1975, 55 pp.

BrHa75b Brinch-Hansen, P.
THE PROGRAMMING LANGUAGE CONCURRENT PASCAL
IEEE Transactions on Software Engineering
Vol. SE-1, No. 2 (June 1975), 199-207

BrHa75c Brinch-Hansen, P.
THE SOLO OPERATING SYSTEM
Information Science, California Institute of Technology, November 1975, 87 pp.

BrHa75d Brinch-Hansen, P.
A REAL-TIME SCHEDULER
Information Science, California Institute of Technology, November 1975, 50 pp.

DDH72 Dahl, O.-J.; Dijkstra, E.W.; Hoare, C.A.R.
STRUCTURED PROGRAMMING
A.P.I.C. Studies in Data Processing No.8,
Academic Press, London, 1972, 220 pp.

DEC74 Digital Equipment Corporation
INTRODUCTION TO MINICOMPUTER NETWORKS
Edward E. Stelmach (Ed.), Digital Equipment Corporation, Maynard, Massachusetts, 1974

Hab72 Habermann, A.N.
SYNCHRONIZATION OF COMMUNICATING PROCESSES
Comm. ACM 15, 3 (March 1972), 171-176

Hoa69 Hoare, C.A.R.
AN AXIOMATIC BASIS FOR COMPUTER PROGRAMMING
Comm. ACM 12, 10 (October 1969), 576-580, 583

Hoa74 Hoare, C.A.R.
MONITORS: AN OPERATING SYSTEM STRUCTURING CONCEPT
Comm. ACM 17, 10 (October 1974), 549-557
Corrigendum: Comm. ACM 18, 2 (February 1975), 95

How76a Howard, J.H.
PROVING MONITORS
Comm. ACM 19, 5 (May 1976), 273-279

How76b Howard, J.H.
SIGNALING IN MONITORS
Proceedings of the 2nd International Conference
on Software Engineering 13.-15.10.1976,
San Francisco, California, 47-52

HoWi73 Hoare, C.A.R.; Wirth, N.
AN AXIOMATIC DEFINITION OF THE PROGRAMMING
LANGUAGE PASCAL
Acta Informatica 2 (1973), 335-355

JeWi75 Jensen, K.; Wirth, N.
PASCAL USER MANUAL AND REPORT
Springer Study Edition, Springer Verlag,
New York, 1975, 167 pp.

Par74 Parnas, D.L.
ON A "BUZZWORD": HIERARCHICAL STRUCTURE
Information Processing 74, Proceedings of the
IFIP Congress, Stockholm, 5.-10.8.74, No. 2 (Software),
North-Holland Publishing Company, 1974, 336-339

Par75 Parnas, D.L.
ON THE DESIGN AND DEVELOPMENT OF PROGRAM FAMILIES
T.H. Darmstadt, Fachbereich Informatik,
FG Betriebssysteme I, Forschungsbericht BS I 75/2,
2.7.1975, 25 pp.