Interner Bericht
DESY F58-86-01
Februar 1986

ENTWURF EINES SYSTEMKERNS ZUR VERWALTUNG KOOPERIERENDER PROZESSE AUF DER BASIS DES NS32000

von

Guido Pfeiffer

Entwurf eines Systemkerns zur Verwaltung kooperierender Prozesse auf der Basis des NS32000

von

Guido Pfeiffer

INHALTSVERZEICHNIS

1. EINLEITUNG

Vorliegendes Papier ist eine Diskussionsgrundlage fuer die Implementierung eines kleinen Systemkerns zur Verwaltung kooperierender bzw. konkurrierender Prozesse.
Ziel ist es,

a) ein allen praktischen Anforderungen genuegendes System zu entwerfen, ohne in den Fehler zu verfallen, ein vollstaendiges Betriebssystem zu entwickeln und

b) zu demonstrieren, dass diese Aufgabe mit Hilfe der hoeheren Programmiersprache Modula-2 zu erreichen ist, moeglichst ohne Verwendung von Assemblerprogrammierung.

Weiterhin soll das System aus wenig Grundelementen bestehen, die aber so vollstaendig sind, dass alle Anwendungen abgedeckt sind. Schliesslich gilt es auch die Forderung nach hoher Effizienz, hier Prozessumschaltung und Interruptbedienung zu erfuellen.

Im naechsten Abschnitt wird - ohne auf Hardware-Details einzugehen - eine Einteilung der Konfigurationen von Prozessoren und Prozessen gegeben, fuer die das System geeignet sein soll. Das dritte Kapitel gibt eine kurze Einfuehrung in die Grundbegriffe konkurrenter Prozesse. In Kapitel 4 wird die Loesung der wichtigsten Konzepte in Modula-2 vorgestellt und diskutiert. Kapitel 5 behandelt die Verwaltung von Interrupts. In Kapitel 6 werden Rueckschluesse auf den Modula-2-Compiler gezogen, insbesondere werden einige einfach zu realisierende Aenderungen vorgeschlagen, die zur Implementierung der vorgestellten Konzepte notwendig sind. In Kapitel 7 werden einige Beispiele diskutiert. Der Anhang schliesslich enthaelt eine vorlaeufige Implementierung, wobei nur die wichtigsten Teile wiedergegeben sind. Es soll hauptsaechlich demonstriert werden, mit wie geringem Aufwand die vorgestellten Konzepte zu realisieren sind.

2. AUFGABENSTELLUNG

Grundlage ist eine auf dem Microprozessor NS32000 basierende Micro-Platine, die von der Gruppe F58 entwickelt wurde. Da die angestrebte Loesung moeglichst unabhaengig von speziellen Prozessorstrukturen sein soll, braucht auf technische Einzelheiten der Platine hier nicht eingegangen werden. Randbedingung ist lediglich, dass mehrere Micros innerhalb eines Systems miteinander kooperieren koennen. Dazu ist das gesamte Memory so strukturiert, dass jeder Micro Zugang zu einem lokalen und einem oeffentlichen Memory hat. Zu dem lokalen Memory kann allein der zugehoerige Prozessor zugreifen, zum oeffentlichen alle im System vorhandenen Prozessoren. D.h., das oeffentlichen Memory gehoert allen Prozessoren gemeinsam. Anders gesagt, auf unterschiedlichen Prozessoren laufende Prozesse kooperieren, indem sie gemeinsame Datenbereiche benutzen. Unter einem Prozess ist eine sequentielle Folge von Aktionen zu verstehen, die parallel mit anderen Aktionen ausgefuehrt wird. Insgesamt lassen sich drei Faelle unterscheiden:

1. Das System besteht aus einem Prozessor und m kooperierenden Prozessen. Der Prozessor wird zwischen den Prozessen nach Eintreffen gewisser Bedingungen hin- und hergeschalten, d.h. die Prozesse werden quasi-gleichzeitig ausgefuehrt. Dies ist der klassische Fall der Multiprogrammierung.

2. Das System besteht aus n Prozessoren und m kooperierenden Prozessen, mit n = m, d.h. jeder Prozessor vertritt einen Prozess. Die Kopplung der Prozessoren erfolgt ueber gemeinsame Datenbereiche, hier ueber das oeffentliche Memory. Man nennt diesen Fall Multiprocessing.

3. Das System besteht wiederum aus n Prozessoren und m kooperierenden Prozessen, mit n = m. Die Kopplung der Prozessoren erfolgt jedoch ueber Leitungen, z.B. ein Netz. Man nennt diesen Fall 'Distributed Processing'.

In der Praxis treten gewoehnlich Mischfaelle auf. Meistens ist n << m, d.h. viele Prozesse sind auf einige Prozessoren verteilt, mit Kopplung der Prozessoren ueber gemeinsame Datenbereiche und Leitungen. Kennzeichnend fuer alle drei Faelle ist, dass die Prozesse lose gekoppelt sind. Die Interaktionen finden an einigen wenigen, im Programmtext genau festgelegten Stellen statt. Es ist sogar wuenschenswert, die Kopplung zwischen Prozessen so gering wie moeglich zu machen, z.B. durch Verwendung grosser Puffer.

3. LOESUNGSKONZEPTE

Bei der Interaktion von Prozessen treten grundsaetzlich zwei zu loesende Probleme auf:

- der gegenseitige Ausschluss (mutual exclusion) und
- die Synchronisation von Prozessen.

Ersteres bedeutet, dass waehrend ein Prozess einen Datenbereich bearbeitet, den er mit anderen Prozessen teilt, darf kein anderer Prozess Zugang zum gleichen Datenbereich haben. Der Datenbereich muss vor unberechtigtem Zugriff geschuetzt werden. Synchronisation bedeutet, dass sich Prozesse gegenseitig ueber ihren Ablauf informieren koennen, so dass sie - zeitlich verschraenkt - korrekt ablaufen.

Mutual Exclusion und Synchronisation koennen unmittelbar durch Verwendung von Semaphoren erreicht werden. Unmittelbar soll heissen, dass der Programmierer beides selbst ausprogrammieren muss unter Verwendung der Semaphor-Primitiven. Dies ist die niedrigste, allgemeinste aber auch fehlertraechtigste Loesungsstufe.

In einem weiter entwickelten Ansatz kooperieren Prozesse indirekt ueber Prozeduren. Man nennt dies prozedur-orientiert. Eine weitere Moeglichkeit ist, Prozesse direkt kooperieren zu lassen. Man nennt dies nachrichten-orientiert. Das prozedur-orientierte Verfahren fuehrt zum Begriff des Monitors, das Nachrichten-orientierte zum Begriff des Rendevous. (Andere Kooperationsmoeglichkeiten, wie Operations-Orientierung und Message-Passing werden hier nicht weiter diskutiert).

Im naechsten Abschnitt werden die hier erwaehnten Begriffe Semaphor, Monitor und Rendevous kurz erlaeutert. Fuer weitere Information sei auf die Literatur verwiesen, z.B. Andrews und Schneider (1983).

3.1 Sempahore

Semaphore wurden von Dijkstra (1968) eingefuehrt. Nach heutiger Terminologie wuerde man einen Semaphore als einen abstrakten Datentyp bezeichnen, fuer den zwei Operationen definiert sind, haeufig P(s) und V(s) genannt. Ein Semaphore s kann nur zwei Werte annehmen, z.B. 0,1 oder True,False oder Busy,Free etc. Zusaetzlich kann s eine Warteschlange zugeordnet sein.

Die Bedeutung von P(s) und V(s) ist:

```
P(s):  IF s > 0  THEN  s := s - 1
                 ELSE  "warte in der s zugeordneten Schlange"
                 END

V(s):  s := s + 1;
       IF Schlange(s) # leer THEN
         s := s - 1;
         "entferne einen Prozess aus der Schlange und fuehre ihn aus."
       END
```

Mit anderen Worten, ein Semaphor kann als ein Tor verstanden werden, das offen oder geschlossen ist. Wenn s geschlossen ist, wird ein Prozess, der P(s) ausfuehrt verzoegert, bis s wieder offen ist. V(s) oeffnet s. Ist s bereits offen, hat V(s) keine Wirkung, d.h. der augenblickliche Prozess laeuft weiter. Ueber die Art der Implementierung von P und V bzw. die Implementierung der Warteschlangen ist nichts gesagt. Wichtig ist nur, dass die oben beschriebene Wirkung von P und V gewaehrleistet ist (daher abstrakter Datentyp!)

Gegenseitigen Ausschluss leisten Semaphore durch Einschliessen des kritischen Codes (Zugriff zu gemeinsamen Daten) in P(s) und V(s):

Prozess 1	Prozess 2
P(s) "kritischer Code" V(s)	P(s) "kritischer Code" V(s)

Ebenso koennen Semaphore Ereignisse signalisieren:

P(s) "Prozess 1 wartet auf Prozess 2"	V(s) "Prozess 2 signalisiert Prozess 1

Mit Semaphoren lassen sich im Prinzip alle Arten kooperierender Prozesse beschreiben. Sie sind ein grundlegendes hoehersprachliches Konzept zur Formulierung von gegenseitigem Ausschluss und Prozess-Synchronisation. Sie haben jedoch eine Reihe von Nachteilen und sind daher anderen Konzepten gewichen. Zwei der mit Semaphoren verbundenen Probleme sind:

- Gegenseitiger Ausschluss und Synchronisation sind grundverschiedene Dinge, die nicht durch dasselbe Sprachmittel beschrieben werden sollten. Sie sind bei Verwendung von Semaphoren nur noch schwer - d.h. nur aus dem Kontext heraus - zu unterscheiden.

- Die Verwendung von Semaphoren erfordert grosse Disziplin des Programmierers. Eine vergessene V-Operation, bzw. eine durch Sprung oder Return nicht ausgefuehrte, resultiert in einem deadlock. Semaphore sind im grunde noch zu primitive Sprachmittel, vergleichbar mit den Goto's bei Kontrollstrukturen.

3.2 Monitore

Der hier verwendete Begriff des 'Monitors' hat nichts mit dem auch als 'Monitor' bezeichneten Teil eines Betriebssystems zu tun, der Prozesse ueberwacht und sie dem Prozessor zur Ausfuehrung uebermittelt (besser Scheduler genannt).

Monitore wurden eingefuehrt, um gegenseitigen Ausschluss und Synchronisation in einem einzigen Konzept zu vereinigen, z.B. Hoare (1974). Ein Monitor besteht aus einem 'Programmblock' oder auch Modul, mit lokalen Daten, nach aussen hin exportierten Prozeduren und einem Initialisierungsteil. Entscheidend ist dir folgende wichtige Eigenschaft des Monitors: Zu jedem beliebigen Zeitpunkt kann nur eine einzige Prozedur eines Monitors aktiv sein. D.h. wenn ein Prozess eine Monitor-Prozedur ausfuehrt, muessen andere Prozesse, die dieselbe oder eine andere Prozedur des Monitors ausfuehren wollen automatisch (!) verzoegert werden, d.h. sie muessen warten, bis die gerade aktive Monitor-Prozedur fertig und der Monitor wieder frei ist. Auf diese Weise ist gegenseitiger Ausschluss eine implizite Eigenschaft von Monitoren. Natuerlich muss dieses Verhalten durch eine entsprechende Implementierung gewaehrleistet sein.

Ueblicherweise schuetzt man Datenbereiche, ueber die Prozesse miteinander kommunizieren durch einen Monitor. Die kritischen Datenbereiche werden lokal im Monitor angelegt und sind ausschliesslich ueber Monitor-Prozeduren zugreifbar. Daher auch die Bezeichung prozedur-orientierte Kooperation von Prozessen.

Zur Synchronisation wird ein neuer Datentyp 'Condition' (Bedingung) eingefuehrt. Aehnlich wie der Semaphore ist der Typ Condition ein abstrakter Typ, auf den zwei Operationen definiert sind, WAIT(condition) und SIGNAL(condition). Anders aber als bei Semaphoren nehmen Condition-Variable keine Werte an. Auf der Abstraktionsebene des Benutzers signalisieren Conditions das Eintreffen von Bedingungen. Auf der Implementierungsebene dagegen repraesentieren sie eine Menge auf diese Bedingung wartender Prozesse.

Die Operationen WAIT und SIGNAL sind nur innerhalb von Monitoren zulaessig. Die folgende Beschreibung entspricht der urspruenglich von Hoare gegebenen Semantik:

- WAIT(c):

  Der aufrufende Prozess wird verzoegert, bis ein anderer Prozess ein SIGNAL(c) innerhalb des gleichen Monitors aussendet. Gleichzeitig mit dem WAIT wird der Monitor freigegeben, um einem anderen Prozess ueberhaupt die Moeglichkeit zu geben, in den Monitor hineinzukommen und ein Signal auszusenden.

- SIGNAL(c):

  Wenn kein Prozess auf s wartet, bleibt die SIGNAL-Operation wirkungslos. Ist jedoch die zu s gehoerende Warteschlange nicht leer, wird genau ein Prozess daraus entfernt und sofort wieder ausgefuehrt. Der signalisierende Prozess geht so lange in den Wartezustand. Er kann erst wieder aktiv werden, nachdem der Monitor freigegeben wurde, entweder durch ein weiteres WAIT oder beim Verlassen der Monitor-Prozedur.

Eine weitere Regel bzw. Empfehlung besagt, dass Monitore so schnell wie moeglich freigegeben werden sollten. D.h. nach Beendigung einer Monitorprozedur und Freigabe des Monitors sollte festgestellt werden, ob noch andere Prozesse im Wartezustand innerhalb des Monitors sind (durch zuvor ausgegebene Signal-Operationen). Sie sind vor allen anderen Prozessen zu reaktivieren, damit der Monitor wieder frei wird.

Am klassischen Beispiel des Produzenten/Konsumenten-Problems soll das Monitorkonzept verdeutlicht werden. Zwei Prozesse, Produzent und Konsument tauschen Daten ueber einen gemeinsamen Puffer aus. Je groesser der Puffer, desto loser sind die Prozesse gekoppelt. Der Produzent legt ueber den Aufruf einer Prozedur deposit Daten in den Puffer ab, der Konsument holt sie ueber eine Prozedur fetch aus dem Puffer. Waehrend der Vorgaenge 'ablegen' bzw. 'holen' darf der jeweilige andere Prozess nicht an den Puffer heran. Ablegen und Holen sind also in gegenseitigem Ausschluss durchzufuehren. Ferner koennen zwei Wartebedingungen auftreten, wenn der Puffer voll ist und der Produzent Daten ablegen bzw. wenn der Puffer leer ist und der Konsument Daten holen will. Es tauchen also die beiden Grundprobleme gegenseitiger Ausschluss und Synchronisation von Prozessen auf. Man verkapselt daher die gemeinsame Variable - den Puffer -, die Prozeduren deposit und fetch, zusammen mit einigen lokalen Variablen in einem Monitor. Folgendes Beispiel zeigt die Implementierung des Monitors Buffer:

```
MONITOR Buffer;

   CONST N = 128;                    (* Puffergroesse                    *)
   VAR   count:     [0..N];          (* Anzahl abgelegter Elemente       *)
         in,out:    [0..N-1];        (* Indizes                          *)
         nonfull:   Condition;       (* n < N, d.h. Puffer nicht voll *)
```

```
        nonempty: Condition;     (* n > 0, d.h. Puffef nicht leer *)
        buffer: ARRAY [0..N-1] OF ElementType

    PROCEDURE deposit(x: ElementType);
    BEGIN
      IF count = N THEN WAIT(nonfull) END;
      (* es gilt die Bedingung: 0 <= count < N *)
      count := count + 1;
      buffer[in] := x;
      in := (in+1) MOD N;
      SIGNAL(nonempty)
    END deposit;


    PROCEDURE fetch(VAR x: ElementType);
    BEGIN
      IF count = 0 THEN WAIT(nonempty) END;
      (* es gilt die Bedingung: 0 < count <= N *)
      count := count - 1;
      x := buffer[out];
      out := (out+1) MOD N;
      SIGNAL(nonfull)
    END fetch;

BEGIN (* Initialisierung des Monitors *)
   count := 0; in := 0; out := 0
END Buffer.
```

Diese Implementierung ist zwar noch nicht ganz effizient, da bei jedem Ablegen und Holen ein Signal gesendet wird, was eigentlich erst dann notwending ist, wenn ein Prozess wartet. Zur Demonstration ist der Algorithmus jedoch ausreichend.

Monitore sind ideal geeignet zur Realisierung der ersten beiden oben beschriebenen Systemklassen, also fuer Systeme bestehend aus n Prozessoren und m kooperierenden Prozessen, die ueber gemeinsame Datenbereiche miteinander kommunizieren. Fuer verteilte Systeme ist das als naechstes besprochene Rendevouskonzept geeigneter.

3.3 Rendevous

Der Begriff Rendevous geht auf einen Artikel von Hoare (1978) zurueck. Implementiert sind Rendevous unter anderem in der Programmiersprache ADA, vgl. z.B. Ichbiah (1979). Unter einem Rendevous zweier Prozesse versteht man, dass sich die Prozesse selbst synchronisieren, Daten austauschen und danach parallel weiter laufen. Im Prinzip geschieht dies so:

Ein Sender gibt folgende Anweisung aus

call receiver(<expression>)

Die zugehoerige Anweisung des Empfaengers ist

accept(<variable>)

Der Sender wartet, bis der Prozess receiver eine accept-Anweisung ausgibt, und umgekehrt wartet der Prozess receiver auf eine Sendeanweisung. Also, egal welcher Prozess als erster den Synchronisations- bzw. Rendevouspunkt erreicht, er wartet bis der andere Prozess eintrifft. Danach wird <variable> := <expression> ausgefuehrt, und beide Prozesse laufen unabhaengig voneinander weiter.

Diese Art von Rendevous nennt man auch unidirektionales Rendevous, da Daten nur vom Sender zum Empfaenger transportiert werden. Gibt der Empfaenger jedoch vor dem Auseinanderlaufen wieder Daten an den Sender zurueck, nennt man dies bidirektionales Rendevous.

Wollen verschiedene Prozesse gleichzeitig ein Rendevous mit einem Empfaenger eingehen, bildet sich eine Warteschlange aus. Der Empfaenger bedient dann ueblicherweise den am laengsten wartenden Prozess zuerst.

Rendevous modellieren auf natuerliche Weise Prozess-Synchronisation, besser als Condition-Variable beim Monitor. Gegenseitigen Ausschluss erhaelt man, durch Verbindung des Auschlusskonzeptes beim Monitor mit dem Rendevous (vgl. hierzu das Rendevous-Konzept in ADA).

4. REALISIERUNG IN MODULA-2

4.1 Grundlagen

Die Programmiersprache Modula-2 (Wirth 1985) enthaelt keines der oben beschriebenen Konzepte, ebensowenig eine Prozessverwaltung oder Echtzeit-Eigenschaften. Dafuer bietet Modula-2 jedoch eine Reihe elementarer Hilfsmittel, um eben dies zu erreichen. Es lassen sich Prozesse erzeugen und zwischen Prozessen hin- und herschalten. Eine besondere Anweisung ist zur Interruptsbehandlung gedacht. Das Wort 'Prozess' wird in Modula-2 im Sinne einer Koroutine verstanden.

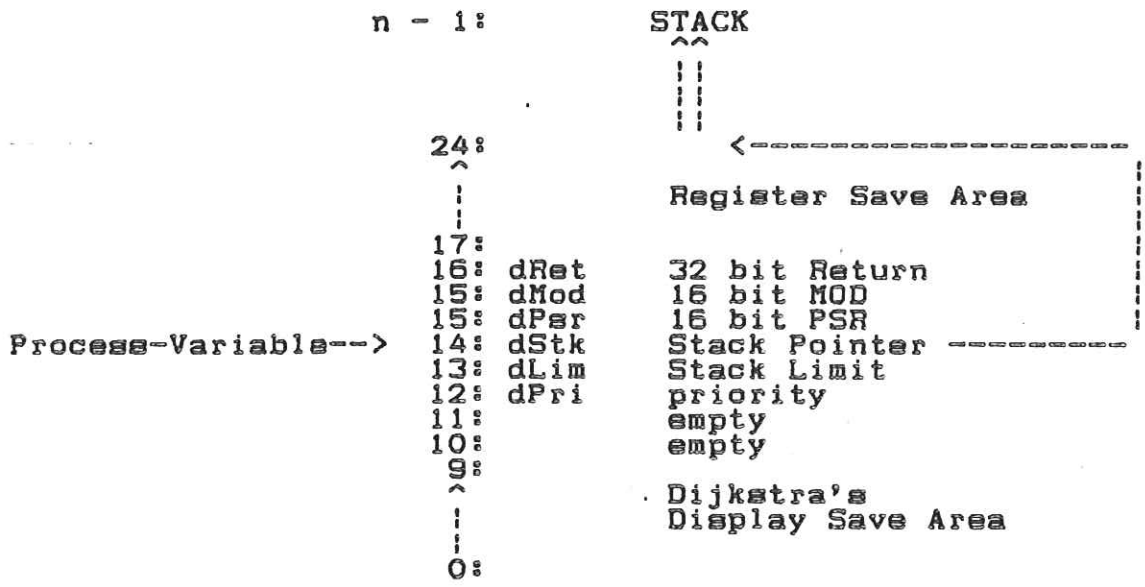
Syntaktische sehen die Anweisungen so aus:

```
PROCEDURE NEWPROCESS(P: PROC; A: ADDRESS; n: CARDINAL;
                     VAR p1: PROCESS)
```

- P ist die Prozedur, d.h. der Code, aus dem der Prozess besteht,
- A ist die Anfangsadresse eines zum Prozess gehoerenden Arbeitsspeichers,
- n ist die Groesse des Arbeitsspeichers und
- p1 ist eine Variable, die auf den Prozess verweist. Man bezeichnet sie als Prozessvariable.

NEWPROCESS legt am oberen Ende des Arbeitsspeichers einen Prozess-Kontrollblock an, der untere Bereich des Arbeitsspeichers ist der zum Prozess gehoerende Stack.

Der Prozess-Kontrollblock hat folgendes Format:



PROCEDURE TRANSFER(VAR p1,p2: PROCESS)

der gerade aktive Prozess wird angehalten, der Variablen p1 zugewiesen und danach der von p2 bezeichnete Prozess aufgenommen.

Auf die in Modula-2 enthaltene Anweisung IOTRANSFER zur Steuerung von Interrupts wird nicht eingegangen, da eine andere Interruptsbehandlung vorgeschlagen wird, als in Modula-2 vorgesehen ist.

4.2 Das Monitor-Konzept und Modula-2

Grundlage fuer die Erzeugung eines Monitors in Modula-2 ist der Begriff des Moduls. Nach Wirth erzeugt man aus einem Modul durch Angabe einer Prioritaet im Modulkopf einen Monitor, etwa so:

```
MODULE Monitor[1];

  (* Monitor-Prozeduren *)

END Monitor.
```

Gedacht ist dies so, dass durch Erhoehen der Prozessor-Prioritaet, Prozeduren innerhalb des Monitors nicht unterbrechbar sind. Man koennte dies auch durch Abschalten des Interruptsystems erreichen. Die Arbeit, gegenseitigen Ausschluss zu gewaehrleisten, ist also auf die Hardware verlagert.

Diese Loesung ist nicht befriedigend wenn Interrupts zu verarbeiten sind. Erstens sollte das Interruptssystem so wenig wie moeglich abgeschaltet werden, i. A. nur an einigen kritischen Stellen, zum zweiten kann beim NS32000 die Prozessorprioritaet nicht veraendert und die Interrupts nur durch einen Interrupt (Supervisorcall) ausgeschaltet werden.

Dies bedeutet, dass das Verhalten eines Monitors, so wie im letzten Kapitel beschrieben, in Modula-2 ausprogrammiert werden muss. Die Hilfsmittel hierzu werden in naechsten Abschnitt beschrieben.

4.2.1 Realisierung von Semaphoren und Monitoren

In einem gesonderten Modul - NsKernel genannt - werden die Datentypen Processname, Condition, Sema (Semaphore), Process, Interrupt und die Prozeduren StartProcess, P, V, WAIT, WaitTime, SIGNAL, InitCond und InitSema eingefuehrt:

```
TYPE
  ProcessName = ARRAY [0..15] OF CHAR;
  Condition;
  Sema;
  Process;
  Interrupt;

PROCEDURE StartProcess     (p: PROC; name: ProcessName;
                            n: CARDINAL; prio: CARDINAL; pr: Process);

PROCEDURE P                (VAR sm: Sema);

PROCEDURE V                (VAR sm: Sema);

PROCEDURE WAIT             (VAR sm: Sema; VAR co: Condition);

PROCEDURE SIGNAL           (VAR co: Condition);
```

```
PROCEDURE InitCond          (VAR co: Condition);

PROCEDURE InitSema          (VAR sm: Sema);
```

Die Typen Condition, Sema und Process sind nach Modula-2-Nomenklatur opak, d.h. ihre innere Struktur ist nur innerhalb des Moduls NsKernel bekannt Nach aussen hin ist sie fuer den Benutzer unsichtbar. Die Bedeutung von Condition und Sema ergibt sich von selbst. Der Typ Interrupt wird im naechsten Kapitel ueber Interrupts besprochen. Der Typ Process ist ein Pointer auf einen Prozess. Die Prozeduren P,V,WAIT und SIGNAL gehorchen der im Abschnitt ueber Semaphore und Monitore besprochenen Semantik. Neu sind die Prozeduren InitCond und InitSema zur Initialisierung von Conditions und Semaphoren. Das oben gegebene Beispiel des Produzenten/Konsumenten-Problems wuerde in Modula-2 fast genauso aussehen. Der wesentliche Unterschied besteht darin, dass der gegenseitige Ausschluss jetzt explizit mit Semaphoren ausprogrammiert ist, eine Aufgabe, die sonst der Compiler uebernimmt.

```
IMPLEMENTATION MODULE Buffer;
   FROM NsKernel IMPORT Sema,Condition,P,V,WAIT,SIGNAL,
                        InitCond,InitSema;
   EXPORT deposit,fetch;

   CONST N = 128;                    (* Puffergroesse                    *)
   VAR   count:    [0..N];           (* Anzahl abgelegter Elemente       *)
         in,out:   [0..N-1];         (* Indizes                          *)
         mutex:    Sema;             (* gegenseitiger Ausschluss         *)
         nonfull:  Condition;        (* n < N, d.h. Puffer nicht voll    *)
         nonempty: Condition;        (* n > 0, d.h. Puffef nicht leer    *)
         buffer: ARRAY [0..N-1] OF ElementType


   PROCEDURE deposit(x: ElementType);
   (*-------------------------------*)
   BEGIN
     P(mutex);
     IF count = N THEN WAIT(mutex,nonfull) END;
     (* es gilt die Bedingung: 0 <= count < N *)
     count := count + 1;
     buffer[in] := x;
     in := (in+1) MOD N;
     SIGNAL(nonempty);
     V(mutex)
   END deposit;


   PROCEDURE fetch(VAR x: ElementType);
   (*-------------------------------*)
```

```
    BEGIN
      P(mutex);
      IF count = 0 THEN WAIT(mutex,nonempty) END;
      (* es gilt die Bedingung: 0 < count <= N *)
      count := count - 1;
      x := buffer[out];
      out := (out+1) MOD N;
      SIGNAL(nonfull);
      V(mutex)
    END fetch;

BEGIN (* Initialisierung des Monitors *)
    count := 0; in := 0; out := 0;
    InitSema(mutex);
    InitCond(nonempty); InitCond(nonfull);
END Buffer.
```

4.2.2 Der Modul NsKernel

Der Modul NsKernel enthaelt die innere Struktur der Typen Condition, Sema, Process, Interrupt, die Implementierung der Monitorprozeduren, die Verwaltung der Warteschlangen und einen Prozessverwalter (Scheduler), der die Aktivierung von Prozessen aus der Warteschlange steuert.

Typ Condition

Variable des Typs Condition sind als Pointer definiert. Sie markieren den Beginn einer Warteschlange. Warteschlangen bestehen aus einer verketteten Liste von Prozess-Deskriptoren. Ein Prozess-Deskriptor (QueueItem) enthaelt u.A. folgende Felder (die vollstaendige Definition ist dem Anhang zu entnehmen):

```
  Condition = POINTER TO QueueItem;

  QueueItem = RECORD
                process: PROCESS;
                nextQ:   Condition;
                queue:   Condition;
                caller:  Condition;
                server:  Condition;
                pname:   ProcessName;
                ...
              END
```

Das Feld process ist die Startadresse des Prozesses, ueber nextQ

werden saemtliche vorhandenen Prozesse zu einem Ring verkettet, queue verkettet Warteschlangen und pname enthaelt den Namen des Prozesses. Die beiden Felder caller und server dienen der Implementierung von Rendevous. Der Aufruf von StartProcess erzeugt und initialisiert ein neues Objekt Queueltem und stellt den neuen Process in die Schlange der um die CPU konkurrierenden Prozesse (Ready-Queue).

Typ Sema

Die Definition von Semas ist:

```
Sema  =  RECORD
           open: BOOLEAN;
           cond: Condition
         END
```

Das Feld open kontrolliert den Semaphore, cond ist der Kopf einer eventuellen Warteschlange. Die Prozeduren P und V sind im Prinzip wie folgt implementiert (fuer Multiprozessorsysteme ist dies nicht korrekt!):

```
PROCEDURE P(VAR sema: SEMA);
BEGIN
  WITH sema^ DO
    IF NOT open THEN
      (* stelle laufenden *)
      (* Prozess in die   *)
      (* cond-Schlange    *)
      Schedule
    END;
    open := FALSE
  END
END P
```

```
PROCEDURE V(VAR sema: SEMA);
BEGIN
  WITH sema^ DO
    open := TRUE;
    IF cond <> NIL THEN
      (* aktiviere den 1. auf cond *)
      (* wartenden Prozess         *)
      cond := cond^.queue;
      Schedule
    END
  END
END V
```

Warteschlangen und Scheduler

Prozesse, die bereit sind CPU-Zeit zu uebernehmen werden - in dieser vorlaeufigen Implementierung - in zwei Arten von Warteschlangen gefuehrt, signalQ und readyQ, wobei letztere aus einem Array von nach Prioritaeten geordneten Warteschlangen besteht, gemaess folgenden Deklarationen:

```
TYPE Priority = [1..maxprio];

VAR  signalQ:  Condition;
     readyQ:   ARRAY Priority OF Condition;
     cp:       Condition;
```

cp ist ein Pointer, der auf den gerade laufenden Prozess zeigt (current process). Der Scheduler durchsucht zunaechst die signalQ und danach die readyQ nach absteigender Prioritaet. Ist ein Prozess gefunden, erfolgt die Umschaltung durch eine TRANSFER-Anweisung (Koroutinen-Aufruf). Fuer die signalQ z.B. lautet der Algorithmus

```
VAR sO: Condition;

BEGIN
  IF signalQ <> NIL THEN
    sO := cp; cp := signalQ;
    signalQ := cp^.queue; cp^.queue := NIL;
    TRANSFER(cO^.process,cp^.process);
    RETURN
  END
END
```

Die signalQ wird nur dann von der Prozedur SIGNAL gefuellt, wenn der laufende Prozess eine wartende Bedingung bedient und dabei selbst in den Wartezustand, naemlich die signalQ uebergeht. Durch bevorzugte Bedienung der signalQ wird dafuer gesorgt, dass der Monitor schnellstens wieder frei wird.

4.2.3 Diskussion

Wie im letzten Abschnitt gezeigt wurde laesst sich das Monitorkonzept auf einfache Weise in Modula-2 realisieren. Die angegebene Loesung hat jedoch einen grossen Nachteil, da sie von der Disziplin des Anwenders abhaengt. Der gegenseitige Ausschluss muss mit Hilfe von Semaphoren explizit ausprogrammiert werden, d.h. jede exportierte Monitor-Prozedur ist durch Semaphore zu schuetzen. Ausserdem sollte die Verwendung von WAIT und SIGNAL auf Monitor-Prozeduren beschraenkt bleiben. Beide Punkte koennen genauso gut auch vom Compiler sichergestellt werden, was allerdings bedingt, dass das Monitorkonzept Teil der Sprache ist. Fuer Modula-2 bedeutet dies, entweder den Compiler zu aendern und damit auch die Sprache oder einen Preprozessor zu implementieren, der aus dem Modul des Abschnitts 4.2.1 automatisch den Monitor des Abschnitts 3.2 generiert. Die erste Loesung der Aenderung von Modula-2 sollte aus prinzipiellen Gruenden vermieden werden. Die Implementierung eines

Preprozessors ist ein gangbarer Weg, allerdings mit dem Nachteil eines zusaetzlichen Uebersetzer-Passes. Andererseits wird sowohl gegenseitiger Ausschluss als auch die korrekte Verwendung von WAIT und SIGNAL automatisch sichergestellt.

Die SIGNAL-Operation ist nicht ganz unproblematisch, da es verschiedene Varianten der Implementierung gibt. Hier wurde auf die urspruenglich von Hoare (1978) gegebene Semantik zurueckgegriffen. Wie bereits beschrieben, wird dabei genau einer der auf das Signal wartendenden Prozesse aktiviert, waehrend der signalisierende Prozess solange wartet. Der wartende Prozess kann daher davon ausgehen, dass die zum Signal gehoerende Bedingung unbedingt erfuellt ist. Andererseits erfordert diese Loesung einen i.A. ueberfluessigen Prozesswechsel. Denn meistens ist SIGNAL die letzte Anweisung einer Monitorprozedur. Die Rueckkehr zum signalisierenden Prozess dient dann nur der endgueltigen Freigabe des Monitors.

In einer anderen Implementierung von SIGNAL behaelt der Signalisierer den Monitor besetzt. Vorlaeufig findet noch kein Prozesswechsel statt. Stattdessen werden entweder einer oder alle wartenden Prozesse in die Schlange der den Monitor kontrollierenden Semaphore gestellt. Erst wenn der signalisierende Prozess den Monitor verlaesst, wird ein wartender Prozess aktiv. Dieser kann dann allerdings nicht mehr davon ausgehen, dass die Signalisierungsbedingung noch erfuellt ist. Sie muss vor der weiteren Ausfuehrung nochmals geprueft werden. Bei dieser Loesung sollte daher das WAIT immer innerhalb einer WHILE-Schleife stehen:

```
WHILE NOT Bedingung DO WAIT(...) END
```

Der Einfachheit halber wurden zur Prozess-Synchronisierung bisher nur die beiden Prozeduren WAIT und SIGNAL eingefuehrt. In Wirklichkeit werden sicher noch mehr Funktionen benoetigt, z.B. um Prozesse eine Zeitlang schlafen zu legen (Sleep) bzw. auf eine Bedingung warten zu lassen (WaitTime) oder einen Prozess abzubrechen (Abort) oder den Status einer Warteschlange zu bestimmen (Status). Diese Funktionen sind bereits vorgesehen, aber noch nicht implementiert, vgl. Definitions-Modul von NsKernel im Anhang. Weitere Funktionen liessen sich bei Bedarf leicht in NsKernel integrieren.

Der Schedule-Algorithmus wurde bewusst einfach gehalten. Prozesse werden in der Rangfolge ihrer Prioritaeten aktiviert, bei Prozessen gleicher Prioritaet der am laengsten wartende. Die hoechste Prioritaet haben in jedem Falle Prozesse, die noch einen Monitor besetzt halten. Ein laufender Prozess kann von aussen nur durch einen Interrupt unterbrochen werden, ansonsten nur durch sich selbst (WAIT oder SIGNAL). Auf ein Zeit-Multiplex-Verfahren wurde wegen des hoeheren Verwaltungsaufwands verzichtet und ist bei den meisten Prozess-Steuerungsaufgaben auch nicht noetig.

4.3 Rendevous

Das beschriebene Monitorkonzept ist ausreichend, um alle bei kooperierenden Prozessen auftretenden Anwendungen zu beschreiben. Dennoch wird zusaetzlich ein einfaches Rendevous-Konzept vorgeschlagen, mit dem sich einige Probleme leichter formulieren lassen. Hierfuer werden zwei Prozeduren, MEET und ACCEPT eingefuehrt:

```
PROCEDURE MEET(VAR pr: Process);

PROCEDURE ACCEPT;
```

In MEET ist ein Parameter (pr) vom Typ Process anzugeben. Er bezeichnet den Prozess, mit dem das Rendevous durchgefuehrt werden soll. Dieser gibt seine Bereitschaft zum Rendevous durch ein ACCEPT bekannt. Welcher Prozess zuerst den Synchronisationspunkt MEET bzw. ACCEPT erreicht, wartet bis der andere eintrifft. MEET und ACCEPT sind nicht symmetrisch aufgebaut. Waehrend ACCEPT auf irgendeinen Prozess wartet, muss in MEET der andere Prozess explizit genannt sein. Entsprechend koennen unterschiedliche Prozesse ein Rendevous mit ein und demselben Prozess verlangen. Sie warten dann in einer Schlange auf ein ACCEPT. Beim ACCEPT tritt keine Warteschlange auf. Aus dieser unsymmetrischen Konstruktion ergibt sich, dass MEET und ACCEPT besonders zur Inanspruchnahme von Dienstleistungen geeignet sind. Eine Dienstleistung kann z.B. allen Prozessen in einem ACCEPT angeboten werden, waehrend MEET eine bestimmte Dienstleistung anfordert. Die Kommunikation zwischen zwei auf unterschiedlichen Prozessoren laufenden Prozessen koennte nach folgendem Schema ablaufen:

```
Prozess caller                 |  Prozess server
-------------------------------|-------------------------
   ...                         |
fuelle Puffer                  |    LOOP
MEET(server)                   |      ACCEPT
sende Pufferdaten              |      empfange Daten
   ...                         |      verarbeite Daten
                               |    END
```

Sind die Prozesse ueber ein gemeinsames Memory gekoppelt, entfaellt das explizite Senden und Empfangen der Daten. Evtl. muss dann aber ein Semaphor den gegenseitigen Ausschluss der Prozesse gewaehrleisten.

Die symmetrische Kommunikation zwischen zwei Prozessen mit gegenseitigem Datenaustausch (bidirektionales Rendevous) waere nach

folgendem Schema zu formulieren:

```
        Prozess 1                    |        Prozess 2
-------------------------------------|-------------------------------
          ...                        |          ...
        fuelle Puffer                |          ...
        MEET(process2)               |        ACCEPT
        sende Pufferdaten            |        empfange und verarbeite Daten
        ACCEPT                       |        MEET(process1)
        empfange Daten               |        sende Daten
          ...                        |          ...
```

Fuer die Implementierung von Rendevous werden die beiden Felder 'caller' und 'server', beide vom Typ Condition verwendet. Hinter 'caller' kann sich eine Warteschlange von MEET's aufbauen. Hinter 'server' wartet hoechstens der ACCEPT-Prozess. Tritt das Rendevous ein, werden beide Prozesse - gemaes ihrer Prioritaet - in die Ready-Queue gestellt und zwar in der Reihenfolge server caller, so dass bei gleicher Prioritaet der server zuerst ausgefuehrt wird.

4.3.1 Diskussion

Das hier vorgeschlagene Rendevous ist in seiner Leistungsfaehigkeit begrenzt, da nur die Synchronisation von Prozessen geregelt ist. Der Datenaustausch muss explizit ausprogrammiert werden. Ebenso der gegenseitige Ausschluss, wofuer wieder wie beim Monitor Semaphore zustaendig sind. Verglichen mit anderen Realisierungen von Rendevous (vgl. ADA) fehlt ausserdem ein nichtdeterministisches Element zur Auswahl von ACCEPT-Anweisungen. Fuer die Formulierung des Produzenten/Konsumenten-Problems bietet das vorgestellte Rendevous-Konzept keine Vorteile gegenueber der Monitor-Formulierung mit WAIT und SIGNAL. Andererseits lassen sich Probleme wie Doppelpuffer-Verwaltung selbst mit den hier gezeigten einfachen Rendevousprimitiven MEET und ACCEPT geschickter ausdruecken als mit einem Monitor (vgl. Beispiele). Es sollte daher erst noch weitere Erfahrungen mit Rendevous gesammelt werden, bevor ueber ihren Nutzen geurteilt werden kann.

5. Interrupts

Die Hardware erlaubt 128 Interrupts (Vector-Mode, nicht kaskadiert). Es gibt 16 Prioritaetsebenen, die fest verdrahtet sind. Durch die Software kann eine vorgegebene Prioritaet nicht veraendert werden.

Die Behandlung von Interrupts wird auf die Primitiven WAIT und SIGNAL zurueckgefuehrt. Der Benutzer sieht jedoch nur eine Prozedur AcceptInterrupt, in der alles erforderliche durchgefuehrt wird. Jedem der 128 Interrupt ist in einem Array eine Interrupt-Variable fest zugeordnet, gemaess der Definitionen

```
TYPE
  Interrupt = Condition;

VAR
  interrupt: ARRAY [0..127] OF Interrupt
```

AcceptInterrupt ist wie folgt definiert:

```
AcceptInterrupt(VAR i:  Interrupt)
```

Der Typ Interrupt ist aequivalent mit dem Typ Condition. Es koennen also Warteschlangen aufgebaut und verwaltet werden. Der Typ Interrupt wurde zusaetzlich zu Condition eingefuehrt, um bei AcceptInterrupt den Wertebereich des Condition-Typs auf die vordefinierten Interrupts-Bedingungen einzuschraenken. Hierdurch wird groessere Sicherheit bei Verwendung von AcceptInterrupt erreicht.

AcceptInterrupt stellt den aktuellen Prozess in die Schlange des spezifizierten Interrupts. Die Prozedur entspricht dem Monitor-WAIT, hat jedoch keinen Einfluss auf die Vergabe von Monitorrechten. Nach Eintreffen des Interrupts wird eine dem SIGNAL aequivalente Operation ausgefuehrt. Falls kein Prozess auf den Interrupt wartet faehrt der unterbrochene fort, andernfalls wird er in die bevorrechtigte signalQ gestellt. Danach wird der erste wartende Prozess aus der Queue entfernt und aktiviert. Im Gegensatz zur SIGNAL-Operation muss bei der Signalisierung von Interrupts der betreffende Interruptvektor durch eine RETI-Anweisung ge'cleared werden. Dies geschieht zum fruehest moeglichen Zeitpunkt, naemlich vor Aktivierung des wartenden Prozesses.

5.1 Diskussion

Es gibt zwei Ansaetze zur Behandlung von Interrupts. Bei der direkten und schnellsten Methode wird eine Prozedur direkt auf den entsprechenden Interruptvektor gesetzt. Es gibt keine Warteschlangen. Nach Eintreffen eines Interrupts wird die Prozedur nach Massgabe der Hardware-Prioritaet sofort und unabhaengig von der Umgebung gestartet. Modula-2 sieht hierfuer die Primitivoperation IOTRANSFER vor.

Bei einem anderen, hier gewaehlten Ansatz wird die Behandlung von Interrupts in die Prozessverwaltung integriert. Notwendigerweise dauert es dann etwas laenger, die betroffene Prozedur zu aktivieren. Dafuer gewinnt man eine Reihe von Vorteilen, wie einfache Benutzbarkeit (die Initialisierung der Interruptvektoren z.B. wird in die Initialisierung des Schedulers verlegt), groessere Allgemeinheit, Warteschlangen und i.A. eine schnellere Freigabe des Interruptssystems.

Um den 'overhead' so gering wie moeglich zu halten wird vorgeschlagen an dieser und nur dieser Stelle Assembler anstatt Modula-2 fuer die Implementierung zu verwenden. In einer niederen Pseudo-Sprache sieht die Interruptsbehandlung so aus (z.B. fuer Interrupt 127):

```
PROCEDURE Interrupt127;
BEGIN
  ADDR(interrupt[127]) => R0;
  GO TO InterruptService
END Interrupt127;


PROCEDURE InterruptService;
BEGIN
  IF @R0 <> NIL THEN          (* ein Prozess wartet                  *)
    InsertQ(signalQ,cp);      (* aktueller Prozess in die Queue      *)
    R1 := cp; cp := R0;       (* R1 = alter Prozess, cp = neuer      *)
    R2 := TOS; R3 := TOS;     (* Return Adr & MOD,PSR vom Stack      *)
    BISPSRW(200H);            (* umschalten zum User Stack           *)
    TOS := R3; TOS := R2;     (* MOD,PSR & Return Adr auf Stack      *)
    SAVE [R0...R7];           (* Register retten                     *)
    SPRD (SP,@R1.process);    (* rette alte Prozessvariable          *)
    LPRD (SP,@R0.process);    (* lade neue Prozessvariable           *)
    RESTORE [R0...R7];        (* restauriere Register                *)
  END;
  RETI                        (* fuehre neuen Prozess aus und        *)
END InterruptService;         (* schalte Interrupts ein              *)
```

Der Operator '@' bedeutet indirekten Zugriff, 'R0.process' bedeutet Zugriff zum Feld process des von R0 adressierten Rekords. Das Feld

'process' - die Prozessvariable - zeigt auf den Stack des jeweiligen Prozesses. Die Prozedur InsertQ besteht im einfachsten Falle - wenn die Queue leer ist - aus der Anweisung 'signalQ := cp'.

Dieses Code-Beispiel soll ein Gefuehl fuer die Effizienz der Interruptbehandlung geben, an der sich nur noch schwer ein Befehl sparen laesst. Der einzige overhead besteht darin, den unterbrochenen Prozess in die signalQ zu stellen.

6. COMPILER-AENDERUNGEN

Fuer die Implementierung der bisher beschriebenen Prozessverwaltung genuegen die normalen von Modula-2 zur Verfuegung gestellten Hilfsmittel nicht. Das erste und wichtigste Problem taucht bei der Implementierung der Semaphore in einem Multiprozessor-System mit Kommunikation ueber ein gemeinsames Memory auf. Das Pruefen und Aendern der Semaphore muss eine unteilbare Operation sein. Abschalten der Interrupts genuegt nicht, das dies immer nur fuer einen Prozessor gemacht werden kann. Der NS32000 sieht zwei Sonderbefehle vor, Set Bit Interlocked (SBITI) und Clear Bit Interlocked (CBITI). Die Befehle kopieren das addressierte Bit in das Flagbit des Processor-Status-Registers (PSR) und fuehren danach die Set- bzw. Clear-Operation durch. Beide Befehle sind zusammen unteilbar (nicht unterbrechbar). Der Originalzustand kann danach ueber die Flag abgefragt werden. Bei Implementierung der Semaphor-Operationen P und V fuer ein Multiprozessor-System muessen die Befehle SBITI und CBITI benutzt werden. Sie sollten daher dem SYSTEM-Modul von Modula-2 zugefuegt werden.

Weiterhin sollte der Aufruf von WAIT, SIGNAL etc. als ein Supervervisor-Call (SVC) implementiert sein. Fuer den Benutzer ist es am einfachsten, wenn entsprechende SVC's automatisch erzeugt werden. Hierfuer ist die Codegenerierungsphase im Compiler so zu aendern, dass anstelle eines Prozeduraufrufs direkt ein SVC (TRAP) generiert wird.

Zur Optimierung der Prozessverwaltung ist es zudem zwar nicht notwendig, aber wuenschenswert, Moeglichkeiten zur Steuerung der Verwendung von Registern vorzusehen.

Insgesamt werden folgende Erweiterungen des SYSTEM-Moduls vorgeschlagen:

INCLI(s,i)	Einfuegen des Elementes i in die Menge s (nicht unterbrechbar)
EXCLI(s,i)	Entfernen des Elementes i aus der Menge s (nicht unterbrechbar)
FLAG	Funktion des Type BOOLEAN. TRUE = Flag gesetzt, sonst FALSE
P(sema)	rufe Procedure 'P' in NsKernel via TRAP
V(sema)	-- " --
WAIT(sema,cond)	-- " --
SIGNAL(cond)	-- " --
MEET(process)	-- " --
ACCEPT	-- " --
LDREG(Register,e)	lade das Register mit dem Ausdruck e. (aequivalent zu 'Register' := e)

```
   USEREG(x)          lade Variable x in ein freies Register,
                      und ersetze alle weiteren Zugriffe zu x durch
                      einen entsprechenden Registerzugriff.
   FREEREG(x)         loese die Bindung von x an ein Register.
```

Dabei ist Register als Enumeration

```
   Register = (R0,R1,R2,R3,R4,R5,R6,R7);
```

definiert.

7. BEISPIELE

7.1 Terminaltreiber

Das bereits wiederholt besprochene Beispiel des Produzenten/Konsumenten-Problems ist ein Prototyp fuer viele unterschiedliche Anwendungen. In einer Abwandlung wuerde ein Treiber zur Ein- bzw. Ausgabe von einem Terminal als Monitor so aussehen:

```
IMPLEMENTATION MODULE Terminal;
(*****************************)

   FROM SYSTEM IMPORT WORD;
   FROM NsKernel IMPORT Sema,Condition,P,V,WAIT,SIGNAL,
                        InitCond,InitSema,AcceptInterrupt;
   FROM InterruptHandler IMPORT interrupt;

   EXPORT fetch;

   CONST N = 128;                       (* Puffergroesse                    *)
   VAR   count:    [0..N];              (* Anzahl abgelegter Elemente       *)
         in,out:   [0..N-1];            (* Indizes                          *)
         mutex:    Sema;                (* gegenseitiger Auschluss          *)
         nonempty: Condition;           (* n > 0, d.h. Puffer nicht leer *)
         buffer:   ARRAY [0..N-1]  OF CHAR;
         wsp:      ARRAY [0..177B] OF WORD;
         terminalIn: PROC;
         x[..B]:   CHAR;          (* feste Adresse der Terminal-Puffer-
                                     Registers *)


   PROCEDURE fetch(VAR ch: CHAR);
   (*---------------------------*)
   BEGIN
     P(mutex);
     IF count = 0 THEN WAIT(mutex,nonempty) END;
     count := count - 1;
     ch    := buffer[out];
     out   := (out+1) MOD N;
     V(mutex)
   END fetch;


   PROCEDURE terminalInput;
   (*----------------------*)
   BEGIN
     LOOP
       AcceptInterrupt(interrupt[nr]);
```

```
      (* nr = Nummer des Interrupts *)
      (* wenn der Puffer voll ist (n=N), keine Zeichen annehmen *)
      IF n < N THEN
        count := count + 1;
        buffer[in] := x;
        in := (in+1) MOD N;
        SIGNAL(nonempty);
      END
    END
  END terminalInput;

BEGIN (* Initialisierung des Monitors *)
  count := 0; in := 0; out := 0;
  InitSema(mutex);
  InitCond(nonempty);
  StartProcess(terminalIn,"terminal",200B,0,terminalInput);
END Terminal.
```

Der Monitor Terminal exportiert nur die eine Prozedur 'fetch'. Der Produzent 'terminalInput' ist im Monitor verkapselt. fetch leert den Puffer 'buffer' zeichenweise. Ist der Puffer leer, wartet die rufende Prozedur, bis wieder ein Zeichen da ist - signalisiert von terminalInput. Der Prozess terminalInput fuellt den Puffer bis er voll ist. Weitere Zeichen werden nicht beachtet. terminalInput wartet auf einen Interrupt, dessen Nummer hier noch nicht festgelegt ist. Ebenso muss auch die Registeradresse, wo das Zeichen abgelegt ist, noch bestimmt werden.

7.2 Doppelpuffer

Das folgende Beispiel zeigt schematisch die Implementierung eines Doppelpuffers. Events werden in einer Prozedur 'getEvent(event)' eingelesen und in einem Puffer 1 abgelegt. Ist der Puffer voll, wird auf einen Puffer 2 umgeschaltet und dieser gefuellt. Waehrenddessen soll Puffer 1 in einer Prozedur 'processBuffer' verarbeitet werden. Das Verarbeiten kann z.B. eine Vorauswertung, Schreiben auf ein Speichermedium, Uebertragung zu einem anderen Rechner, etc. sein. Die Struktur der Events soll in einem nicht naeher angegebenen Modul als Datentyp festgelegt sein. Ebenso wird ueber die zugehoerige Operation 'getEvent' nichts ausgesagt.

Es werden zwei Loesungen angegeben, eine mit Monitor, die andere mit Rendevous.

7.2.1 Monitor-Implementierung

```
IMPLEMENTATION MODULE DoubleBuffer;
(*********************************)

  FROM Storage IMPORT ALLOCATE,DEALLOCATE;
  FROM NsKernel  IMPORT Condition,Sema,P,V,Wait,Signal,AcceptInterrupt,
                        StartProcess,InitSema,InitCond;
  FROM InterruptHandler IMPORT interrupt;
  FROM Somewhere IMPORT Event,getEvent;

  EXPORT deposit;


  CONST
    N = 100;

  TYPE
    BufferSize = [0..N];

    Buffer = POINTER TO ARRAY BufferSize OF Event;

  VAR
    b1,b2,tb: Buffer;
    n: BufferSize;
    done,mutex: Sema;
    transmit: Condition;


  PROCEDURE deposit(e: Event);
  (*------------------------*)
  BEGIN
    P(mutex);
    IF n <= N THEN b1^[n] := e; INC(n)
    ELSE
      P(done);
      Signal(transmit);
      tb := b1; b1 := b2; b2 := tb;
      n := 0
    END;
    V(mutex)
  END deposit;


  PROCEDURE consume;
  (*---------------*)
  BEGIN
    LOOP
      Wait(mutex,transmit);
      processBuffer(b1)
    END
```

```
  END consume;

  PROCEDURE transmitDone;
  (*--------------------*)
  BEGIN
    LOOP
      AcceptInterrupt(interrupt[...]);
      V(done)
    END
  END transmitDone;

  PROCEDURE processBuffer(VAR b: Buffer);
  (*-----------------------------------*)
  BEGIN
  END processBuffer;


BEGIN (* DoubleBuffer (Monitor) *)
  InitSema(done); InitSema(mutex);
  InitCond(transmit);
  NEW(b1); NEW(b2); n := 0;
  StartProcess(consume,"Consumer",500,0);
  StartProcess(transmitDone,"Interrupt",500,0);

(* in der folgenden LOOP werden die Events eingelesen und im
   Puffer abgelegt.
*)

  LOOP
    AcceptInterrupt(interruptEvent);
    GetEvent(event);
    deposit (event)
  END

END DoubleBuffer.
```

In der Monitor-Initialisierung werden die beiden Prozesse 'consume' und 'transmitDone' gestartet. Beide gehen sofort in den Wartezustand. consume wartet auf den Auftrag, den Puffer zu verarbeiten (transmit), transmitDone auf einen Interrupt. Prozedur 'deposit' fuellt den aktuellen Puffer b1. Ist er voll wird versucht, auf den anderen Puffer umzuschalten. Zuvor wird mit Hilfe der Semaphore 'done' festgestellt, ob eine vorige Puffer-Verarbeitung beendet ist. Wenn nicht, muss gewartet werden. Wenn ja, wird Prozess consume signalisiert, mit der Verarbeitung zu beginnen. consume nimmt seine Arbeit unmittelbar auf, startet die Verarbeitung (processBuffer) und wartet wieder auf den naechsten Auftrag. Jetzt wird deposit wieder aktiv und schaltet die Puffer um. Waehrenddessen laeuft die Verarbeitung des anderen Puffers und so lange ist die

Semaphore 'done' gesperrt. Das Ende der Verarbeitung loest den Interrupt aus, auf den in transmitDone gewartet wird. Danach kann die Semaphore 'done' freigegeben werden, und transmitDone wartet auf den naechsten Interrupt.

7.2.2 Rendevous-Implementierung

```
IMPLEMENTATION MODULE DoubleBuffer;
(*********************************)

  FROM Storage IMPORT ALLOCATE,DEALLOCATE;
  FROM NsKernel IMPORT Condition,Process,Sema,Meet,Accept,AcceptInterrupt,
                       P,V,StartProcess,InitSema;
  FROM InterruptHandler IMPORT interrupt;
  FROM Somewhere IMPORT Event,getEvent;

  EXPORT deposit;


  CONST
    N = 100;

  TYPE
    BufferSize = [0..N];

    Buffer = POINTER TO ARRAY BufferSize OF Event;

  VAR
    b1,b2,tb: Buffer;
    n: BufferSize;
    mutex: Sema;
    transmit: Process;


  PROCEDURE deposit(e: Event);
  (*-------------------------*)
  BEGIN
    P(mutex);
    IF n <= N THEN b1^[n] := e; INC(n)
    ELSE
      Meet(consume);
      tb := b1; b1 := b2; b2 := tb;
      n := 0
    END;
    V(mutex)
  END deposit;
```

```
  PROCEDURE consume;
  (*--------------*)
  BEGIN
    LOOP
      Accept;
      processBuffer(b1);
      AcceptInterrupt(interrupt[...])
    END
  END consume;


  PROCEDURE processBuffer(VAR b: Buffer);
  (*-----------------------------------*)
  BEGIN
  END processBuffer;

BEGIN (* DoubleBuffer (Rendevous) *)
  InitSema(mutex);
  NEW(b1); NEW(b2); n := 0;
  StartProcess(consume,"Consumer",500,0,transmit);

(* in der folgenden LOOP werden die Events eingelesen und im
   Puffer abgelegt.
*)

  LOOP
    AcceptInterrupt(interruptEvent);
    GetEvent(event);
    deposit (event)
  END

END DoubleBuffer.
```

Man sieht auf Anhieb, dass diese Formulierung wesentlich kuerzer und einfacher ist. Der Prozess transmitDone und die Semaphore 'done' werden eingespart. Prozess consume fuehrt als erste Anweisung ein 'Accept' aus und geht damit in den Wartezustand. Der aktuelle Prozess fuellt den Puffer bis er voll ist und versucht dann, Prozess consume zu treffen. Dies gelingt, wenn consume bereits wartet. consume laeuft dann los, startet die Pufferverarbeitung (processBuffer) und wartet auf den Interrupt, der das Ende der Verarbeitung signalisiert. Waehrenddessen kann deposit weiterlaufen, die Puffer umschalten und wieder mit Events auffuellen. Irgendwann tritt der Interrupt ein und wird sofort bedient. consume wird dann beim Accept wieder aufgehalten bis zum naechsten Rendevous.

Versucht deposit ein Treffen mit consume und letzterer ist noch nicht bereit dazu, weil er noch auf den Interrupt wartet, so wird deposit aufgehalten bis consume so weit ist. Danach geht es wie oben beschrieben weiter.

8. LITERATUR

ANDREWS,G.R., SCHNEIDER,F.B.: "Concepts and Notations for Concurrent Programming." ACM Comp. Surv. 15,3, 1983

DIJKSTRA,E.W.: "Co-operating Sequential Processes." in Programming Languages. (F. Genuys, ed.), Academic Press, London and New York, 1968

HOARE,C.A.R.: "Monitors: An Operating System Structuring Concept." CACM, 17,8, 1974

HOARE,C.A.R.: "Communicating Sequential Processes." CACM, 21,8, 1978

ICHBIAH,J.D. et al.: "Rationale for the Design of the ADA Programming Language." SIGPLAN, 14,6, 1979

WIRTH,N.: "Programming in MODULA-2." Springer Verlag, 1985

3. ANHANG

3.1 Definition Module NsKernel

```
(*****************************************
*                                       *
*           Implementation of           *
*       Hoare's Monitor Concept for     *
*    concurrent execution of processes  *
*              and                      *
*       a simple Rendevous Concept      *
*                                       *
*****************************************)

DEFINITION MODULE NsKernel;
(*************************)

  EXPORT QUALIFIED ProcessName,Condition,Sema,Process,Interrupt,
    StartProcess,P,V,WAIT,WaitTime,SIGNAL,
    MEET,MeetTime,ACCEPT,AcceptInterrupt,
    Sleep,Abort,Status,
    InitCond,InitSema;

  TYPE
    ProcessName = ARRAY [0..15] OF CHAR;
    Condition;
    Sema;
    Process;
    Interrupt;


  PROCEDURE StartProcess     (p: PROC; name: ProcessName;
                              n: CARDINAL; prio: CARDINAL; VAR pr: Process);
  PROCEDURE P                (VAR sm: Sema);
  PROCEDURE V                (VAR sm: Sema);
  PROCEDURE WAIT             (VAR sm: Sema; VAR co: Condition);
  PROCEDURE WaitTime         (VAR sm: Sema; VAR co: Condition; time: CARDINAL);
  PROCEDURE SIGNAL           (VAR co: Condition);
  PROCEDURE MEET             (VAR rv: Process);
  PROCEDURE MeetTime         (VAR rv: Process; time: CARDINAL);
  PROCEDURE ACCEPT;
  PROCEDURE AcceptInterrupt  (VAR in: Interrupt);
  PROCEDURE Sleep            (time:   CARDINAL);
  PROCEDURE Abort            (VAR pr: Process);
  PROCEDURE Status           (VAR co: Condition): BOOLEAN;
  PROCEDURE InitCond         (VAR co: Condition);
  PROCEDURE InitSema         (VAR sm: Sema);

END NsKernel.
```

9.2 Implementation Module NsKernel

Die hier gezeigte Implementierung soll eine Vorstellung davon geben, wie eine Prozessverwaltung in Modula-2 geschrieben wird. Einige Teile sind noch nicht korrekt. Z.B. muss der Test 'IF NOT open THEN' in der Prozedur 'P(VAR sema: Sema)' fuer den Multiprocessor-Fall mit einem gemeinsamen Memory durch eine Interlock-Operation INCLI ersetzt werden. Im Multiprogrammier-Fall ist die gezeigte Implementierung jedoch korrekt. Einige triviale Prozeduren wie Sleep, Abort etc. sind noch nicht implementiert. Die Monitorprozeduren P,V,WAIT,SIGNAL,MEET und ACCEPT werden ueber Traps aktiviert. Die maskierbaren Interrupts werden dabei ausgeschaltet. Die TRANSFER-Anweisung schaltet sie indirekt ueber das Processor-Status-Register des neuen Prozesses wieder ein.

```
(*****************************************
*                                       *
*           Implementation of           *
*       Hoare's Monitor Concept for     *
*    concurrent execution of processes  *
*                and                    *
*       a simple Rendevous Concept      *
*                                       *
*****************************************)

IMPLEMENTATION MODULE NsKernel;
(*******************************)

  FROM SYSTEM IMPORT ADDRESS,PROCESS,NEWPROCESS,TRANSFER,
    TSIZE;
  FROM Storage IMPORT ALLOCATE,DEALLOCATE;

  CONST
    maxprio = 5;

  TYPE
    Priority      = [1..maxprio];

    Condition     = POINTER TO QueueItem;

    Process       = Condition;

    Interrupt     = Condition;

    Sema          = POINTER TO Semaphore;

    TimeItem  = RECORD
                  ticks: CARDINAL;
                  tQ:    Condition;
```

```
                END;

  QueueItem = RECORD
                process : PROCESS;
                nextQ   : Condition;       (* ring of processes *)
                queue   : Condition;       (* waiting queue     *)
                caller  : Condition;       (* rendevous: caller queue   *)
                server  : Condition;       (* rendevous: server process *)
                pname   : ProcessName;
                priority: Priority;
                abort   : BOOLEAN;
                timer   : TimeItem
              END;

  Semaphore = RECORD
                open : BOOLEAN;
                cond : Condition
              END;


VAR
  signalQ:    Condition;
  readyQ:     ARRAY Priority OF Condition;
  cp,timerQ:  Condition;
  interrupt:  ARRAY [0..127] OF Interrupt;
  k:          CARDINAL;
  Timer:      Process;


PROCEDURE Schedule;
(*---------------*)
  VAR k: CARDINAL; c0: Condition;
BEGIN
  c0 := cp;
  IF signalQ <> NIL THEN
    cp := signalQ;
    WITH cp^ DO
      signalQ := queue; queue := NIL;
      TRANSFER(c0^.process, process);
      RETURN
    END
  ELSE
    k := maxprio;
    WHILE k > 0 DO
      IF readyQ[k] <> NIL THEN
        cp := readyQ[k];
        WITH cp^ DO
          readyQ[k]  := queue;
          IF c0 <> cp THEN
            queue := NIL;
            TRANSFER(c0^.process, process)
```

```
          END;
          RETURN
        END
      END;
      DEC(k);
    END
  END;
  HALT          (* deadlock *)
END Schedule;

PROCEDURE InsertQ(VAR q,c: Condition);
(*-----------------------------------*)
  VAR q0,q1: Condition;
BEGIN
  IF q = NIL THEN q := c
  ELSE
    q0 := q; q1 := q0^.queue;
    WHILE q1 <> NIL DO
      q0 := q1;
      q1 := q0^.queue
    END;
    q0^.queue := c
  END
END InsertQ;

PROCEDURE StartProcess (p: PROC; name: ProcessName;
(*-------------------------------------------------*)
                        n: CARDINAL; prio: CARDINAL;
                        VAR pr: Process);
  VAR
    c0: Condition; wsp: ADDRESS;
BEGIN
  ALLOCATE(wsp,n);
  ALLOCATE(c0,TSIZE(QueueItem));
  pr := c0;
  WITH c0^ DO
    nextQ := cp^.nextQ;
    cp^.nextQ := c0;
    NEWPROCESS(p,wsp,n,process);
    queue    := NIL;
    caller   := NIL;
    server   := NIL;
    pname    := name;
    priority := prio;
    abort    := FALSE;
    WITH timer DO
      ticks := 0;
      tQ    := NIL
    END;
    InsertQ(readyQ[priority],c0);
```

```
  END;
  WITH cp^ DO
    IF prio > priority THEN
      InsertQ(readyQ[priority],cp);
      Schedule
    END
  END
END StartProcess;

PROCEDURE P(VAR sema: Sema);    (* called via trap *)
(*-------------------------*)
BEGIN
  WITH sema^ DO
    IF NOT open THEN            (* assert interlock *)
      InsertQ(cond,cp);
      Schedule
    END;
    open := FALSE
  END
END P;

PROCEDURE V(VAR sema: Sema);    (* called via trap *)
(*-------------------------*)
BEGIN
  WITH sema^ DO
    open := TRUE;
    IF cond <> NIL THEN
      InsertQ(readyQ[cond^.priority],cond);
      InsertQ(readyQ[cp^.priority],cp);
      cond := cond^.queue;
      Schedule
    END;
  END
END V;

PROCEDURE WAIT(VAR sema: Sema; VAR c: Condition);    (* called via trap *)
(*---------------------------------------------*)
BEGIN
  InsertQ(c,cp);
  WITH sema^ DO                     (*Unlock Monitor*)
    open := TRUE;
    IF cond <> NIL THEN
      InsertQ(readyQ[cond^.priority],cond);
      cond := cond^.queue;
    END
  END;
  Schedule
END WAIT;
```

```
PROCEDURE WaitTime (VAR sema: Sema; VAR c: Condition; time: CARDINAL);
(*-------------------------------------------------------------------*)
BEGIN
  (* WAIT, but not longer than 'time' clock ticks *)
END WaitTime;

PROCEDURE SIGNAL(VAR c: Condition);    (* called via trap *)
(*--------------------------------*)
  VAR c0: Condition;
BEGIN
  IF c <> NIL THEN
    InsertQ(signalQ,cp);
    c0 := cp; cp := c;
    WITH cp^ DO
      c := queue;
      TRANSFER(c0^.process,process)
    END
  END
END SIGNAL;

PROCEDURE MEET (VAR rv: Process);    (* called via trap *)
(*-------------------------------*)
BEGIN
  WITH rv^ DO
    IF server = NIL THEN  InsertQ(caller,cp)
    ELSE
      InsertQ(readyQ[server^.priority],server);  (* server *)
      InsertQ(readyQ[cp^.priority],    cp);      (* caller *)
      server := NIL
    END
  END;
  Schedule
END MEET;

PROCEDURE MeetTime (VAR rv: Process; time: CARDINAL);
(*---------------------------------------------------*)
BEGIN
  (* Try to meet rv, but wait not longer than time clock ticks *)
END MeetTime;

PROCEDURE ACCEPT;    (* called via trap *)
(*-------------*)
BEGIN
  WITH cp^ DO
    IF caller = NIL THEN  InsertQ(server,cp);
    ELSE
      InsertQ(readyQ[cp^.priority],    cp);      (* server *)
      InsertQ(readyQ[caller^.priority],caller);  (* caller *)
```

```
      caller := caller^.queue;
    END
  END;
  Schedule
END ACCEPT;

PROCEDURE AcceptInterrupt (VAR i: Interrupt);
(*-----------------------------------------*)
BEGIN
  InsertQ(i,cp);   Schedule
END AcceptInterrupt;

PROCEDURE Sleep(time:   CARDINAL);
(*--------------------------------*)
BEGIN
  (* delay current process for time clock ticks *)
END Sleep;

PROCEDURE Abort(VAR pr: Process);
(*-------------------------------*)
BEGIN
  (* Abort process pr *)
END Abort;

PROCEDURE Status (VAR co: Condition): BOOLEAN;
(*-------------------------------------------*)
BEGIN
  RETURN co = NIL
END Status;

PROCEDURE InitCond (VAR co: Condition);
(*-------------------------------------*)
BEGIN
  NEW(co);
  co := NIL
END InitCond;

PROCEDURE InitSema(VAR sema: Sema);
(*---------------------------------*)
BEGIN
  NEW(sema);
  WITH sema^ DO
    open := TRUE;
    cond := NIL
  END
END InitSema;
```

```
  PROCEDURE timer;
  (*-------------*)
  BEGIN
    LOOP
      (* AcceptInterrupt(timerInterrup); *)
      (* ProcessTimerQueue               *)
    END
  END timer;


BEGIN
  FOR k := 0 TO maxprio DO readyQ[k] := NIL  END;
  FOR k := 0 TO 127 DO  interrupt[k] := NIL  END;
  cp := NIL; timerQ := NIL; signalQ := NIL;
  StartProcess(timer,"Timer",100,0,Timer)
END NsKernel.
```