Spezielle Anwendersprachen
Concurrent Programming

Dr. Werner Pohlmann

Sommersemester 1998

Vorwort

Dieses Skriptum ist während der Vorlesung „Spezielle Anwendersprachen – Concurrent Programming“ im Sommersemester 1998 entstanden. Der Inhalt wurde direkt von den Folien des Dozenten Dr. Werner Pohlmann übernommen, wobei kleinere Korrekturen während der Vorlesung berücksichtigt wurden. Getippt wurde dieser Text von Ronald Blaschke im Satzprogramm LaTeX. Die aktuelle Version dieses Dokumentes kann von seiner Homepage [1] bezogen werden.

Ein herzliches Dankeschön ergeht an dieser Stelle an den Dozenten Dr. Werner Pohlmann für seine freundliche Unterstützung und an Peter Meerwald für sein umständliches Korrekturlesen dieses Skriptums.

Inhaltlich wurde die Erstellung dieses Skriptums am 02.07.1998 mit Revision 1.9 abgeschlossen.

RCS Information

```
$Revision: 1.10 $
$Date: 2001/06/02 10:14:13 $
```

[1] http://www.cosy.sbg.ac.at/~rblasch

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Worum geht's?

- Zwei Handlungen sind *sequentiell* zueinander, wenn die eine die andere voraussetzt; sonst sind sie *konkurrent* (*nebenläufig*).
  Beispiel
  - Kaffee kochen, Kaffe trinken: sequentiell
  - Kaffee trinken, Kuchen essen: konkurrent
- Sequentielle Handlungen müssen *seriell* (hintereinander) ausgeführt werden, konkurrente Handlungen können *parallel* (zeitlich überlappt) ausgeführt werden.

Also: Konkurrent = *potentiell* parallel !

D. h. *Abstraktion* von tatsächlicher Ausführungsweise, die in Abhängigkeit von pragmatischen Zwecken und technischen Gegebenheiten *verschiedene* Formen annehmen kann.

1.1.1 Technische Gegebenheiten

- 1 Prozessor („Ausführer“),
- 1 Prozessor + Hilfsprozessoren (z. B. für I/O),
- mehrere Prozessoren mit gemeinsamen Speicher,
- mehrere Prozessoren ohne gemeinsamen Speicher (insbesondere „verteilte“ Ausführung mit relativ zeitaufwendiger Kommunikation über Netze).

1.1.2 Ausführungsmodi

- Parallel: „echt parallel“
- Serialisiert
- Willkürliche Verzahnung („interleaving“) wenn zerlegbar in Teil-Handlungen : „pseudo-parallel“; verbreitet im 1-Prozessor-Fall, dabei Strategie nötig (Betriebssystem) wie z. B. „round robin“ (Zeitscheibenverfahren); die Strategie der Prozessorzuteilung sollte *fair* sein (schwieriger Begriff, von uns nicht detailiert betrachtet; nicht notwendig: Jeder kriegt gleich viel Prozessorzeit, aber: Keiner wird unendlich benachteiligt, also eher qualitative als quantitative Perspektive).

1.1.3 Konkurrente Programme, konkurrentes Programmieren

= Es gibt konkurrente (parallel ausführbare) Programmteile.

Übliche Programme sind sequentiell, Reihenfolge durch übliche Semantik (z. B. Folge von Anweisungen, getrennt durch „;“, oder rechte/linke Seite von Zuweisung) festgelegt. !

Typisch organisiert als:
Kollektion von *Prozessen* (tasks, aktive Objekte) mit eigenen Daten und Handlungen und Kontrollfluß; Prozeß in sich sequentiell, aber konkurrent (überwiegend, s . u.) zu anderen.

Das Problem dann:

1.1.4 *Interaktion* von Prozessen

Prozesse i. A. nicht unabhängig voneinander, sondern

- in *Kooperation* zur Bewältigung einer gemeinsamen Aufgabe.
- in *Konkurrenz* um gemeinsame Betriebsmittel.

Nicht verwechseln:

- **„Konkurrent“ (engl: concurrent) = nebenläufig, potentiell parallel**
- **„Konkurrenz“ (engl: competition) = Wettbewerb** !

Das erfordert:

- *Synchronisation*, z. B. Zustand erreicht, Betriebsmittel frei, gemeinsamer Start, ...
- *Kommunikation*, z. B. Austausch von Zwischenergebnissen

Synchronisation ist *keine* Frage von gemeinsamer Zeit, also von Intervallänge, Dauer, u. ä., sondern einer der Logik; Zeitfehler („race conditions“) entstehen, wenn der konkurrente Programmierer sich darauf verläßt, daß Handlung A schneller fertig wird als Handlung B: Das mag stimmen, aber beim nächsten Betriebssystem-Release, der nächsten Hardware-Aufrüstung ungültig werden ... !

1.1.5 Zwecke, Anwendungsbereiche

Grundsätzlich 2 Ziele möglich:

1. Zeitgewinn (oder Steigerung der behandelbaren Problemgröße),
2. Problemadäquate (logische Struktur, technischer Kontext) Systemstruktur,

sowie Mischformen.

Wir verfolgen Ziel 2! !

Ziel 1: vgl. VL „Paralleles Rechnen“; setzt *echte* Parallelität voraus; typisch: Feinkörnige Parallelität mit großer Regelmäßigkeit, spezielle Algorithmen, spezielle Programmiermodelle (z. B. PRAM-, Datenfluß ...) und Rechnerarchitekturen (massiv parallele Rechner nach SIMD und MIMD Prinzip);
Klassisches Anwendungsfeld: z. B. Numerik für physik.-technische Aufgaben wie Wettervorhersage oder Bildverarbeitung.

1.1.6 Beispiele/Anwendungsgebiete für konkurrentes Programmieren

- Informationsverarbeitung in Stufen (Fließbandprinzip), vgl. Unix pipes
- Benutzer-(Dialog-) und Anwendungssysteme
- Mehrbenutzersysteme
- Verteilte Transaktionen in Datenhaltungssystemen
- Allg. Client-Server-Systeme
- Betriebssysteme
- Rechnernetze, Telematik
- Eingebettete („embedded“) Systeme, Echtzeitsysteme

Betriebssysteme
= historisch *das* Gebiet der Entwicklung konkurrenten Programmierens

Eingebettete Systeme
= Software kontolliert technische Prozesse z. B. Fahrzeugsteuerung, Flugüberwachung ...

Technisches System enthält *„natürliche“ Parallelität* (z. B. Flugzeug = Höhenruder, Seitenruder ...) die abgebildet wird auf konkurrente Softwarekomponenten. !

1.2 Kleines Beispiel

Eine *Gebäudeheizung* werde wie folgt gesteuert:

- Wenn die Temperatur im *Heizkessel* ein vorgegebenes Intervall nach unten (oben) verläßt, wird der *Brenner* ein- (aus-)geschaltet;
- Wenn die Temperatur im *Gebäude* ein vorgegebenes Intervall nach unten (oben) verläßt, wird die *Umwälzpumpe* ein- (aus-)geschaltet;
- Alle diese Aktivitäten werden ausführlich auf dem *Monitor* des Hausmeisters angezeigt.

Klar:
Softwarelösung sollte reales System widerspiegeln, also aus *3 Komponenten* „Kessel“, „Haus“ und „Monitor“ bestehen, die die offensichtlichen Verantwortlichkeiten und Kollaborationen haben:

Kessel	
Kesselsteuerung	KesselTemp. Sensor
	Brenner
	Monitor

Haus	
Haussteuerung	HausTemp. Sensor
	Pumpe
	Monitor

Monitor	
anzeigen	

1.2.1 Lösung mit *passiven* Objekten

Kessel
Low: Temperature
High: Temperature
Control

control =

```
Actual := Kessel_Sensor.Read;
if Actual < Low then
   Brenner.On;
   Monitor.Write(Clock, Actual, "on");
elsif Actual > High then
   Brenner.Off;
   Monitor.Write(Clock, Actual, "off");
end if;
```

Haus
Low: Temperature High: Temperature
Control

Control =
wie oben

Monitor
Write()

und Hauptschleife („cyclic executive“):

```
loop
   Kessel.Control;
   Haus.Control;
end loop;
```

1.2.2 Kritik

Kontrolle von Kessel und Haus der Sache nach unabhängig voneinander, aber hier *eng gekoppelt*
⇒

- Beide Aktivitäten werden mit *derselben Häufigkeit* (Rate) ausgeführt; aber möglicherweise sinnvoll, verschiedene oder sogar dynamisch änderbare Raten einzuführen (z. B. Kessel genauer überwachen, wenn Brenner angeschaltet ist); diese Forderung ist nur sehr schwer in die Schleifenkonstruktion aufzunehmen!
- Fehler in einer Komponente wirkt auf andere Komponente, z. B. Aufruf von Haus.control terminiert nicht *und* Kessel eingeschaltet ⇒ Bumm!
- Schwierig, weitere Aktivitäten in das Programm zu integrieren.

1.2.3 Konkurrente Lösung mit *aktiven* Objekten (Prozessen)

= Kontrollfluß (Schleife) in Komponenten!

```
process Kessel_Control is
   Low: Temperature := ...;
   High: Temperature := ...;
   Actual: Temperature;

   begin
      loop
         Actual := ...;
         if Actual < Low then
            ...                -- wie oben!
     end loop;
end process;

process Haus_Control is
   ...                       -- wie Kessel
end process;
```

und Hauptprogramm =

```
start Kessel_Control;
start Haus_Control;
```

1.2.4 Kritik

logisch überzeugende Lösung, aber noch *unvollständig*!

- Aufnahme von delay-statements (Verzögerungen) in die Schleife zur Herstellung der gewünschten Wiederholrate; bei 1-Prozessor-time-slicing: Unschärfen unvermeidbar (⇒ später)
- Monitor ist jetzt *konkurrent* genutzte *Ressource*, aber verwandtes Schreiben erfordert vermutlich *exklusiven* Besitz
  ⇒ spezielles *Protokoll* nötig, um Chaos zu vermeiden; evtl. *Pufferung* von Output, um Wartezeiten zu vermeiden.

(*Protokoll:* Vereinbarung über *Form* (Parameter u. ä.) und *Reihenfolge* von *Operationsaufrufen* an Schnittstellen von Objekten ...)

1.3 Vorgehensweise und Plan der Vorlesung

- Grundsätzlich 2 Vorgehensweisen bei konkurrenter Programmierung möglich:
  - sequentielle Programmiersprache *plus* Dienste eines geeigneten Betriebssystems
  - Konkurrente Programmiersprache

  Unentschieden: Vor- und Nachteile (z. B. Portabilität, Effizienz) beider Vorgehensweisen.
  Einigermaßen klar: Variante 2 sorgt für besser les- und wartbare Programme!

  Wir wählen deshalb Variante 2!
- Es gibt mehrere Sprachen für konkurrente Programmierung, mit unterschiedlichen Mitteln, zum Teil nur akademisch, z. B. Modula (und Pascal Varianten), CHILL, Mesa, CSP, occam, PEARL, LINDA, Ada ...

  Wir nehmen Ada95
  - Relativ wichtig in der Praxis auf diesem Gebiet, vor allem für „eingebettete Systeme“,
  - Unterstützt OOP und verteilte Ausführung (Ada95!),
  - Implementierung für gebräuchliche Maschinen/BS (auch PC/Windows95) vorhanden und frei zugänglich (GNAT),
  - Im Haus gelehrt und benutzt.
- Plan der Vorlesung
  - Falls nötig: Kurze Einführung in Ada,
  - Grundprobleme, Lösungen und Konzepte konkurrenter Programmierung,
  - Ada-spezifische Mittel (tasks, rendezvous, protected objects) und ihr Gebrauch,
  - Soweit Zeit reicht: Einblicke in Themen wie real time programming, conncurrency & objects, distributed computing.

1.3.1 Literatur

Skriptum Kopien dieser Folien schritthaltend mit VL,

Zu Ada

- J. Barnes: „Programming in Ada95“, 1996
- J. Skanskolm: „Ada from the Beginning“, 2nd ed, 1994
- Website: http://www.adahome.com/, http://www.epfl.ch/Ada/

Wichtigste Quellen zur VL

- M. Ben-Ari: „Principles of Concurrent Programming“, 1982
- A. Burns, A. Wellings: „Concurrency in Ada“, 1995

Andere Quellen zu Concurrency

- G. R. Andrews: „Concurrent Programming“, 1991
- J. Bacon: „Concurrent Systems“, 1993
- C.A.R. Hoare: „Communcating Sequential Processes“, 1985
- A. Skipo: „Concurrent Programming“, 1989
- O. Whiddett: „Conncurrent Programming for Software Engineers“, 1987
- T. Axford: „Concurrent Programming“, 1989

Kapitel 2

Ada Tasks I

Hier: Nur ganz kurze Darstellung (mehr ⇒ später)
Zweck: Einfache Beispiele programmieren zu können !

- Tasks können im Deklarationsteil eines Hauptprogramms eingeführt werden;
- Syntax:
  Spezifikationsteil, Schnittstelle nach außen

  ```
  task T is
     ...
  end T;
  ```

  Falls Schnittstelle leer genügt die Deklaration

  ```
  task T;
  ```

  Rumpf, enthält (ähnlich Prozeduren) lokale Deklarationen und eine Folge von Anweisungen (also ein „Programm“)

  ```
  task body T is
     ...
  end T;
  ```

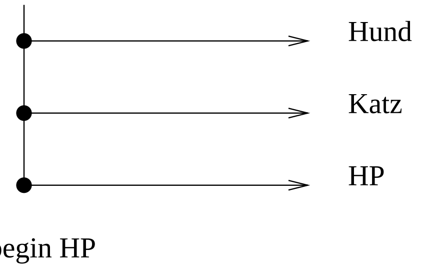
2.1 Beispiel

```
with Stringpack; use Stringpack;
procedure Tierwelt is

   task Hund;
   task body Hund is
   begin
      Print("wau");
      Print("wau");
   end Hund;

   task Katz;
   task body Katz is
   begin
```

```
      Print("miau");
   end Katz;

begin
   Print("hello world");
end Tierwelt;

----------------------------------------------------------------------
6 pohlmann@pavian:Tierwelt
wau
wau
miau
hello world
```

Wirkung

- Tasks Hund und Katz werden *aktiviert* zugleich mit Hauptprogramm; d. h. alles geht los bei „begin“ des Hauptprogramms.



- Tasks Hund und Katz *terminieren* jeweils bei *end* ihres Rumpfs; Hauptprogramm terminiert bei end seines Anweisungsteils *und* nachdem die beiden tasks beendet sind;

- Hauptprogramm und die beiden tasks sind *konkurrent* zueinander;

- Tatsächliche Ausführungsweise hängt von technischen Umständen ab; bei 1-Prozessor-System wahrscheinlich pseudoparallele (= verzahnte) Ausführung nach Zeitscheibenverfahren in willkürlicher Reihenfolge

  Ada schreibt Verfahren nicht vor; Ada Implementierung/System kann Optimierungen vornehmen, z. B. einfache tasks ganz abarbeiten. !

Ergebnis des Beispielprogramms tatsächlich unsicher: Alle drei konkurrente Teile greifen zu auf Stringpack/task.io/standard.io ..., d. h. laufen Gefahr, sich dabei gegenseitig zu stören. !

Kapitel 3

Wechselseitiger Ausschluß und gemeinsame Variablen

Die in diesem Kapitel benutzten Programmiertechniken und -lösungen taugen nichts; von ihrem Gebrauch wird dringend abgeraten! !

Warum werden sie hier benutzt?

1. Historisch-didaktische Gründe: Problemlage kennenlernen
2. Hier noch nicht adäquate Ada- (oder sonstwas) Konzepte einführen, aber trotzdem Beispiele programmieren wollen.

3.1 Wechselseitiger Ausschluß („mutual exclusion“)

Ist eines der wichtigsten und fundamentalsten Probleme der konkurrenten Programmierung.
Problem
Zwei (oder mehr) konkurrente Prozesse benutzen dasselbe Betriebsmittel; dieses Betriebsmittel kann aber nicht von mehreren gleichzeitig benutzt werden.
Beispiel

Auf gerätetechnischer Ebene massenhaft, z. B. Floppies kopieren bei nur 1 Laufwerk

auf Programmebene z. B. Programmvariable oder komplizierte Datenstruktur (Container-Typ), I/O ...

⇒ *Problem*
Konkurrentes Verhalten adäquat einschränken (also so konkurrent wie möglich, aber nicht mehr), d. h. sicherstellen, daß kein zweiter zugreift (also notfalls nix tut = wartet), wenn der erste grad zugreift.
⇒ Prozesse *synchronisieren* (koordinieren)

In Ermangelung von etwas Besserem benutzen wir in diesem Kapitel lediglich !

3.2 Gemeinsame Variablen („shared variables“)

zur Prozeßkommunikation (und dann Synchronisation), d. h. Prozeß A teilt Prozeß B was mit, indem er die Information in eine *beiden* zugängliche Programmvariable schreibt.
Klar: Diese Lösungstechnik unterliegt selbst dem Problem.

3.3 Terminologie/Problemstruktur

- *Kritischer Abschnitt* („critical section“) = Teil der Handlungsfolge eines Prozesses, der unter *Ausschluß* aller anderen Prozesse von der Ausführung ihrer entsprechenden Handlungsfolge ausgeführt werden muß; d. h. die kritischen Abschnitte verschiedener Prozesse sind sequentiell zueinander.

- Zur Sicherstellung des Ausschlusses ist ein Protokoll (entsprechende Anweisungen vor und nach kritischem Abschnitt) nötig, also exemplarische Struktur eines Prozesses:

```
loop
   Non_Critical_Part1;
   Pre_Protocol;
   Critical_Section;
   Post_Protocol;
   Non_Critical_Part2;
end loop;
```

- Wobei critical_section und Protokollausführung *kurz* sein sollten im Verhältnis zur übrigen Rechnung.

  Nie etwas in kritische Abschnitte setzen, was da nicht unbedingt sein muß. !

- Wobei critical_section und Protokollteile besonders *sicher* sein sollten. (z. B. Prozeß darf nicht im Besitz des gemeinsamen Betriebsmittels sterben)

3.4 Beispiel: Gemeinsame Variablen (Shared_Demo1)

- Das folgende Programm enthält Prozesse, die *nur* aus kritischen Abschnitten bestehen, nämlich eine gemeinsame Variable modifizieren.
- Effekt war ohne das Zwischenspeichern in lokaler Variable nicht zu sehen, und auch erst für lange Läufe.
- Technik des „busy waiting“ in Hauptprogramm ist ungut, mehr dazu später.

```
with Stringpack; use Stringpack;
procedure Shared_Demo1 is

N: Integer :=0; --shared variable
Anz: Positive := 1_000_000;

task Up;
task body Up is
   My_N: Integer;
begin
   for I in 1..Anz loop
      My_N := N;        -- -+
      My_N := My_N+1;   --  | Kritischer Abschnitt: Zugriff auf n
      N := My_N;        -- -+
   end loop;
end Up;

task Down;
task body Down is
   My_N: Integer;
begin
   for I in 1..Anz loop
      My_N := N;        -- -+
      My_N := My_N-1;   --  | Kritischer Abschnitt
      N := My_N;        -- -+
   end loop;
end Down;

begin
```

```
   while not(Up'Terminated and Down'Terminated) loop
      null;
   end loop; -- sehr schlechter Stil !!!!!!!!!!!!!!!

   Print("n= " & N);
end Shared_Demo1;

--------------------------------------------------------------------
39 pohlmann@pavian:shared_demo1
n= 517520
40 pohlmann@pavian:shared_demo1
n= 6204
41 pohlmann@pavian:shared_demo1
n= -97962
42 pohlmann@pavian:shared_demo1
n= 433532
```

Programm sollte als „Netto“-Effekt die Variable n wieder auf 0 setzen (= 1 Mio. inkrementieren/dekrementieren) aber offenbar unsicher;
Grund: Ungeschützter Gebrauch der gemeinsamen Variablen; also Abläufe möglich wie

	task up	task down
$\boxed{n=0}$		
	My_N := N;	
	My_N := My_N+1;	
		My_N := N;
		My_N := My_N-1;
		N := My_N;
$\boxed{n=-1}$		
	N := My_N;	
$\boxed{n=1}$		

D. h. 1 mal dekrementieren ist durch die Art der Überlappung mit Inkrementieren wirkungslos geworden!

3.5 Hausaufgabe

Programmieren/untersuchen Sie ähnliche Beispiele, z. B.

1. Zwei Prozesse machen Einträge in eine gemeinsam benutzte Liste (Warteschlange, Keller)

2. Zwei Prozesse benutzen Stringpack oder direkt task_io für Ein-/Ausgaben.

3.6 Anmerkung zum Hauptprogramm

Zweck Endergebnis ausdrucken, *nachdem* die beiden tasks beendet sind.

Technik *„busy waiting“* (geschäftiges Warten) d. h. Hauptprogramm fragt ständig nach, ob tasks schon fertig.
⇒ furchtbar viel Rechenaufwand für Nichts-tun!
⇒ andere Synchronisationstechniken müssen her! (⇒ später)

3.7 Wie Problem lösen?

- Wir betrachten im folgenden einige Ideen (von denen erst die letzte wichtig ist).

- Die alle *keine* besondere Unterstützung durch z. B.

  – spezielle Programmiersprachen-Konstrukte

  – spezielle Maschinenbefehle

  – Betriebssystemdienste

verwenden;

- Was bleibt sind dann:
  - Gewöhnliche Programmvariable + übliche Konstruktoren und Anweisungen!
  - Wobei man voraussetzen muß:
    1. Schreiben und Lesen von Variablen primitiver Datentypen (Boolean, Integer) erfolgt *atomar* (ungeteilt, ununterbrochen) und unter Ausschluß eines gleichzeitigen anderen Zugriffs.
    2. Die Prozesse benutzen die Variablen selbst und *nicht* lokale Kopien davon, die sie anfangs erstellen und später zurückschreiben.

1 Kann man als durch die Hardware garantiert ansehen – keine 2 gleichzeitigen Speicherzugriffe!

2 Geht durch Optimierungen des Compilers oft verloren!

Der Compiler sorgt dafür, daß Variablenwert in Register übernommen und dort über mehrere Bearbeitungsschritte hinweg gehalten wird, z. B.

Programm	⇒	Maschineninstruktionen
x := x+1;		load x into register;
		add 1 to register;
if x = 10		compare register with 10;
then x := 0;		if equal then set register to 0;
end if;		store register into x;

D. h. Äquivalenz zum Quellcode entsteht erst am *Ende* des Programmstücks!

Um solche Optimierungen zu verhindern und 2 sicherzustellen, kann man in Ada ein „pragma“ (= Compilerdirektive) verwenden:

```
pragma Volatile (x);
```

Will man ganz sicher gehen (für 1 und 2) schreibt man

```
pragma Atomic (x);
```

3.8 Idee 1

Jeder Prozeß zeigt an, daß er sich im kritischen Abschnitt befindet, und wartet (busy waiting), wenn der andere sich grad dort befindet:

task 1

```
loop
   ...
   Flag1 := True;
   while Flag2 loop
      null;
   end loop;
   Critical_Section;
   Flag1 := False;
   ...
end loop;
```

task 2

```
loop
   ...
   Flag2 := True;
   while Flag1 loop
      null;
   end loop;
```

```
   Critical_Section;
   Flag2 := False;
   ...
end loop;
```

Wobei global vereinbart ist:

```
Flag1, Flag2: Boolean := False;
```

Es gilt:

1. Das Verfahren ist *sicher* („safe“)
2. Das Programm kann in eine *Verklemmungssituation* („deadlock“) geraten

safety Es kann nichts Böses getan werden, d. h. Form von partieller Korrektheit.

deadlock Zwei (oder mehrere) Prozesse sind verklemmt, wenn *jeder* auf ein Ergebnis wartet, das erst durch den Fortschritt des anderen eintreten kann, z. B. p1 wartet auf p2 während p2 auf p1 wartet.

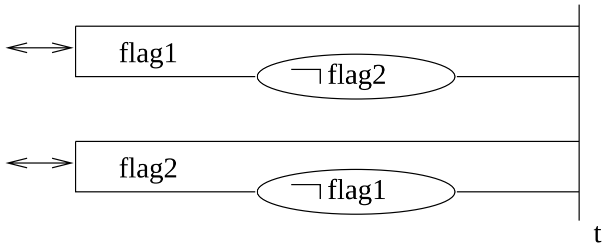
3.8.1 Safety des Verfahrens

Nimm an, es gäbe Zeitpunkt t sodaß *beide* Prozesse in critical_section
$\Rightarrow \exists t_1, t_1', t_2, t_2', t_1 < t_1' < t, t_2 < t_2' < t$ sodaß

- flag1 im Intervall (t_1, t] *und*
- flag2 im Intervall (t_2, t] *und*
- ¬flag2 in t_1' *und*
- ¬flag1 in t_2'.

Dies ist unmöglich denn:
Entweder
$t_1 < t_2 \Rightarrow t_2' \in (t_1, t]$
oder
$t_1 \geq t_2 \Rightarrow t_1' \in (t_2, t]$.

Bildlich: 2 Gummibänder, fixiert am rechten Ende (= t) repräsentieren Gültigkeitsbereich von flag1 und flag2. 2 Marken auf den Bändern (= t_1', t_2') repräsentieren daß ¬flag1 und ¬flag2 gefunden werden. *Unmöglich*, die Bänder so zu dehnen daß *beide* Marken *vor Beginn* des jeweils anderen Bandes!



Korrektheitsbeweise für konkurrente Programme sind

- sehr wünschenswert, weil man die möglichen Abläufe noch schlechter übersieht als im sequentiellen Fall,
- besonders schwierig, weil man immer an mehreren Programmstellen zugleich ist.

Methoden

- Gewöhnliche Prädikatenlogik, wobei man außer Zustandsgrößen auch noch über Zeitpunkte (oder Programmstellen, -zähler) reden muß, (s. o.)
- Temporäre Logik, die spezielle Konstrukte wie „letztendlich“, „als nächstes“ u. ä. verwendet und dadurch die explizite Verwendung von Zeit erspart
- (In einfachen Fällen) Zustandsdiagramme, endliche Automaten
- Spezielle Formalismen zur Algorithmenbeschreibung wie Petri-Netze oder CSP und zugehörige Beweistechniken.

3.8.2 Verklemmung des Verfahrens

Möglicher Ablauf:

task 1	task 2
flag1 := True;	
	flag2 := True;
	while flag1 loop
	null;
	end loop;
while flag2 loop	
null;	
end loop;	

Ab jetzt beide Tasks in Warteschleife.

Verklemmung schwierig durch Tests zu entdecken weil

1. eintreten von Zufällen abhängig (timing, Verzahnung),
2. oft durch andere Aktivitäten im System verdeckt.

3.9 Idee 2

Verbessere Idee 1 dadurch, daß Prozesse zeitweilig ihre Absicht preisgeben.

task 1

```
loop
   ...
   Flag1 := True;
   while Flag2 loop
      Flag1 := False;
      delay 1.0;
      Flag1 := True;
   end loop;
   Critical_Section;
   Flag1 := False;
   ...
end loop;
```

task 2 verläuft analog.

Es gilt

1. *Safety*, aus ähnlichen Gründen wie bei Idee 1
2. *Deadlockfrei*, jedenfalls in dem Sinn, daß das erwartete Ergebnis (¬flag1 bzw. ¬flag2) immer wieder angeboten wird.
3. *Lockout* möglich!

Lockout/Starvation (Verhungern) Prozeß wird unbestimmt (ewig) lange von seinem Ziel abgehalten, wenn ein bestimmtes timing gerade eingehalten wird.

	flag1	flag2
init	False	False
p1 sets	True	False
p2 sets	True	True
p1 tests	True	True
p2 tests	True	True
p1 sets	False	True
p2 sets	False	False
p1 sets	True	False
p2 sets	True	True

Dieser Fall ist weniger schwerwiegend als deadlock, da die Blockierung jetzt nicht zwangsläufig, sondern nur durch sehr *unwahrscheinliche* Aufrechterhaltung der Reihenfolge der Aktionen resultiert, aber praktisch insofern störend, als beliebig große Verzögerungen resultieren können; deadlock und starvation sind Verstöße gegen *liveness* (= Lebendigkeit) des Verfahrens.

Safety Es passiert nix Böses, z. B. beide Prozesse in kritischem Abschnitt.

liveness Letztendlich passiert was Gutes (= Zweck der Veranstaltung), z. B. jeder Prozeß kriegt einmal das gewünschte Betriebsmittel.

3.9.1 Hausaufgabe

Konstruieren Sie ein anderes lockout-Szenario für das Idee 2-Verfahren unter der Annahme, daß Prozeß 1 immer (und nur) in seinem kritischen Abschnitt unterbrochen wird.

3.10 Idee 3

Striktes *Abwechseln*, totsicher aber (zu) unflexibel.
<u>task 1</u>

```
loop
   ...
   while Turn = 2 loop
      null;
   end loop;
   Critical_Section;
   Turn := 2;
   ...
end loop;
```

<u>task 2</u>

```
loop
   ...
   while Turn = 1 loop
      null;
   end loop;
   Critical_Section;
   Turn := 1;
   ...
end loop;
```

Verfahren ist sicher und frei von deadlock/starvation, bedeutet aber *enge Kopplung* beider Prozesse.

- Wenn einer aussteigt kann der andere nicht mehr weiter.
- Beide arbeiten mit *derselben Rate*.

Keine adäquate Lösung des Problems!

3.11 Idee 4

Kombination von Ideen 1 und 2. Dekker 1965, Peterson 1981 (Vereinfacht).

Gemeinsame Variablen

```
Flag1, Flag2: Boolean := False;
Turn: Positive range 1..2;
```

<u>task 1</u>

```
loop
   ...
   Flag1 := True;
   Turn := 1;
   while Flag2 and Turn = 1 loop
      null;
   end loop;
   Critical_Section;
   Flag1 := False;
   ...
end loop;
```

task 2

```
loop
   ...
   Flag2 := True;
   Turn := 2;
   while Flag1 and Turn = 2 loop
      null;
   end loop;
   Critical_Section;
   Flag2 := False;
   ...
end loop;
```

Es gilt

3.11.1 *Kombination vermeidet deadlock*-Gefahr von Idee 1

Angenommen, beide Prozesse wären ewig in Warteschleife.

⇒ Keine Änderung der Variablenwerte

⇒ dauerhaft (flag2 and turn = 1) und (flag1 and turn = 2)

Dies ist unmöglich, da turn nicht gleichzeitig den Wert 1 und 2 haben kann!

3.11.2 *Kombination vermeidet* strikte *Abwechslung* wie in Idee 3

Angenommen, Prozeß 2 befindet sich längere Zeit (oder ewig) im *unkritischen* Teil „...“ seines Programms.

⇒ längere Zeit oder ewig: ¬flag2

⇒ längere Zeit oder ewig: Bedingung der Warteschleife von Prozeß 1 ist False!

Korrektheitsbeweis für das Verfahren ist kompliziert und aufwendig, deshalb hier unterlassen. Für Interessierte: Das Buch von T. Axford (p. 203-205) präsentiert einen auf Zustandsdiagramme gestützten Beweis.

3.11.3 Beispiel (Shared_Demo2)

```
with Stringpack; use Stringpack;
procedure Shared_Demo2 is

N: Integer := 0; --shared variable
Anz: Positive := 1_000_000;

TURN: POSITIVE range 1..2;
FLAG1, FLAG2: BOOLEAN := FALSE;
pragma ATOMIC(TURN); --sicher ist sicher
pragma ATOMIC(FLAG1);
```

```
pragma ATOMIC(FLAG2);

task Up;
task body Up is
   My_N: Integer;
begin
   for I in 1..Anz loop
      FLAG1 := TRUE;
      TURN := 1;
      while FLAG2 and TURN = 1 loop null; end loop;
      My_N := N;
      My_N := My_N+1;
      N := My_N;
      FLAG1 := FALSE;
   end loop;
end Up;

task Down;
task body Down is
   My_N: Integer;
begin
   for I in 1..Anz loop
      FLAG2 := TRUE;
      TURN := 2;
      while FLAG1 and TURN = 2 loop null; end loop;
      My_N := N;
      My_N := My_N-1;
      N := My_N;
      FLAG2 := FALSE;
   end loop;
end Down;

begin
   while not(Up'Terminated and Down'Terminated) loop
      null;
   end loop; -- sehr schlechter Stil !!!!!!!!!!!!!!!

   Print("n= " & N);
end Shared_Demo2;

----------------------------------------------------------------------
70 pohlmann@pavian:shared_demo2
n= 0
```

3.11.4 Kritik an Lösung und verwendeten Mitteln

Lösung ist

1. Schwer zu *verstehen*, *fehleranfällig* im Gebrauch, schwer zu *variieren* (z. B.
   - Verallgemeinerung auf n Prozesse,
   - Verallgemeinerung auf verteilte Systeme).
2. (Sehr) *ineffizient*, da Warten als geschäftiges Warten („busy waiting") realisiert wird, d. h. wartender Prozeß beansprucht Prozessor und verzögert damit, daß anderer Prozeß vorankommt und insbesondere Wartebedingung auflöst. (Illustration siehe unten, mit Zählern in Warteschleifen.) Diese Kritik immer dann berechtigt, wenn wartender Prozeß nicht Prozessor für sich alleine hat!

```
with Stringpack; use Stringpack;
procedure Shared_Demo2a is

N: Integer := 0; --shared variable
Anz: Positive := 1_000_000;

TURN: POSITIVE range 1..2;
FLAG1, FLAG2: BOOLEAN := FALSE;
pragma ATOMIC(TURN); --sicher ist sicher
pragma ATOMIC(FLAG1);
pragma ATOMIC(FLAG2);

BUSY1, BUSY2: Natural := 0;

task Up;
task body Up is
   My_N: Integer;
begin
   for I in 1..Anz loop
      FLAG1 := TRUE;
      TURN := 1;
      while FLAG2 and TURN = 1 loop BUSY1 := BUSY1+1; end loop;
      My_N := N;
      My_N := My_N+1;
      N := My_N;
      FLAG1 := FALSE;
   end loop;
end Up;

task Down;
task body Down is
   My_N: Integer;
begin
   for I in 1..Anz loop
      FLAG2 := TRUE;
      TURN := 2;
      while FLAG1 and TURN = 2 loop BUSY2 := BUSY2+1; end loop;
      My_N := N;
      My_N := My_N-1;
      N := My_N;
      FLAG2 := FALSE;
   end loop;
end Down;

begin
   delay 120.0;
   Print("BUSY1= " &BUSY1 & " BUSY2= " & BUSY2);
end Shared_Demo2a;

----------------------------------------------------------------------
pohlmann@pavian:shared_demo2a
BUSY1= 79802288 BUSY2= 78124617
^C

...
begin
   while not(Up'Terminated and Down'Terminated) loop
      null;
```

```
   end loop; -- sehr schlechter Stil !!!!!!!!!!!!!!!!

   Print("BUSY1= " &BUSY1 & " BUSY2= " & BUSY2);
end Shared_Demo2a;

----------------------------------------------------------------------
rbla@bravo:shared_demo2a
BUSY1= 2147483647 BUSY2= 2147483647
```

3.11.5 Konsequenz daraus

- Erfinde *programmiersprachliche* Konstrukte, die Details der nötigen Protokolle verbergen.
- *Implementiere* solche Konstrukte so, daß *kein* busy waiting, vielmehr: benutze Prozeßverwaltung des *Betriebssystems* mit Diensten wie „Prozeß schlafen legen“/“Prozeß aufwecken“.

3.11.6 Hausaufgabe

Studieren Sie einige der Varianten im Ben-Ari, p. 43-49.

Kapitel 4

Semaphore und Monitore

4.1 Semaphore

- Wurden 1965 von Dijkstra eingeführt.
- Gibt es in mehreren Varianten, hier eine:

4.1.1 Ein Sempaphor S ist eine Variable mit Natural-Werten und zwei Operationen

Wait(S)

```
if S > 0 then
   S := S-1;
else
   setze ausführenden Prozeß wartend;
end if;
```

Signal(S)

```
if es gibt auf S wartende Prozesse then
   lasse ersten (FIFO) dieser Prozesse weitermachen;
else
   S := S+1;
end if;
```

- Wait und signal werden *atomar* (= unteilbar) ausgeführt, d. h. keine andere Aktion zwischen Test und zugehöriger Handlung;
- wait und signal sind neben einer Initialisierung vor Prozeßstart die *einzigen* Operationen auf dem Semaphor.

Statt `wait(S)` und `signal(S)` sagt man oft auch `P(S)` und `V(S)` – niederländisch (Passeren und Vrijmaken)!

4.1.2 Für Sempaphor S gilt folgende Invariante

$$S \geq 0 \ and \ S = S_{init} + \#signals - \#compl\_wait$$

Beweis

compl_wait Prozeß ist über das wait-statement hinausgekommen, hat Operation abgeschlossen.

Formel richtig falls signal/wait Zuweisung an S machen, andernfalls: signal verändert S nicht gdw. wartender Prozeß wird aufgeweckt gdw. wait abgeschlossen ohne Veränderung von S.

4.1.3 Beispiel: Wechselseitiger Ausschluß

```
S: Semaphore := 1;
```

process 1

```
loop
   ...
   Wait(S);
   Critical_Section;
   Signal(S);
   ...
end loop;
```

process 2 Genau wie bei process 1.

Lösung erweiterbar auf n $\geq$ 2 Prozesse. !

4.2 Korrektheit der Lösung

Sei #CS = Zahl der Prozesse im kritischen Abschnitt, dann nach Programmstruktur #CS = #compl_waits - #signals.
Invariante $\Rightarrow$ S = 1 - #CS. (*)

1. **Safety** #CS > 1 wegen (*) unmöglich.
2. **Deadlockfrei** Beide Prozesse warten
   $\Rightarrow$ S = 0 and #CS = 0
   $\Rightarrow$ 1 = 0 (aus (*))
3. **No starvation** Process 1 wartet
   $\Rightarrow$ S = 0
   $\Rightarrow$ #CS = 1 (wegen (*))
   $\Rightarrow$ Process 2 in critical_section
   $\Rightarrow$ Process 2 macht als nächstes signal
   $\Rightarrow$ Process 1 wird aufgeweckt!

4.2.1 Varianten

Boolesche Semaphore Einschränkung des Wertebereichs auf 0..1.

Schwächere Fairneßforderung Z. B. Auswahl des wartenden Prozesses unbestimmt (nicht FIFO-Garantie); oder signalisierender Prozeß kann sofort wieder wait ausführen.

Implementierung Durch Betriebssystemdienst oder (problematisch) busy waiting.

Korrektheit der Anwendung hängt ab von solchen Eigenschaften! Z. B. schwächere Fairneß kann bei obiger Lösung des mutual-exclusion Problems zu starvation führen! !

4.3 Producer-Consumer Problem

Muster eines Kommunikationsproblems

***Producer*-Prozeß** erzeugt Daten.

***Consumer*-Prozeß** verarbeitet Daten.

Puffer Datenstruktur für Zwischenspeicherung, gleicht vorübergehende(!) Unterschiede in Geschwindigkeit aus.

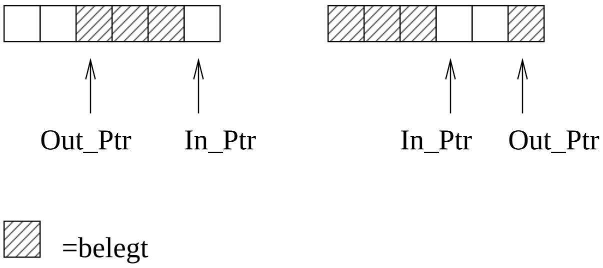
Dabei *Synchronisation* nötig.

- Consumer kann Datenelement erst lesen, wenn producer es geschrieben hat.
- Producer kann Datenelement erst schreiben, wenn Puffer freien Platz enthält.
- Ferner: Je nach Implementierung des Puffers sind Zugriffe unter wechselseitigen Ausschluß zu stellen!

4.3.1 Beispiel

Implementierung Puffer

- **Puffer** Vektor mit n > 0 Plätzen und zyklischer (circular buffer) Verwendung (i := (i+1) mod n);
- Private Zeiger (Indizes) für producer und consumer, Lesezeiger folgt Schreibzeiger, beide zeigen jeweils auf nächste Position



Also

```
N: constant Positive := ...;
type Data is ...;
Buffer: array(0..N-1) of Data;
In_Ptr, Out_Ptr: Natural := 0;
Elements: Semaphore := 0; -- Daten da?
Spaces: Semaphore := N; -- noch Platz?
```

<u>producer</u>

```
D: Data;
begin
   loop
      Produce(D);
      Wait(Spaces);
      Buffer(In_Ptr) := D;
      In_Ptr := (In_Ptr+1) mod N;
      Signal(Elements);
   end loop;
end Producer;
```

<u>consumer</u>

```
D: Data;
begin
   loop
      Wait(Elements);
      D := Buffer(Out_Ptr);
      Out_Ptr := (Out_Ptr+1) mod N;
      Signal(Spaces);
      Consume(D);
   end loop;
end Consumer;
```

beachte Reihenfolge

- **Erst Element schreiben, dann Signal(Element)**
- **Erst Element lesen, dann Signal(Spaces)** !

Schleifeninvariante

- **Elements = #(noch nicht gelesene Elemente)**
- **Spaces = N - #(noch nicht gelesene Elemente)** !

Daraus folgt die Korrektheit

- Puffer kann nicht gelesen (geschrieben) werden, wenn er leer (voll) ist
- **No deadlock** Spaces = Elements = 0 $\Rightarrow$ N = 0 (aus Invariante).
- **No starvation** Klar: Einer verhungert $\Rightarrow$ der andere auch $\Rightarrow$ deadlock.

4.3.2 Hausaufgabe

Überlegen Sie, ob die folgende Lösung, die busy-waiting benutzt, korrekt ist.
producer

```
loop
   Produce(D);
   while((In_Ptr+1) mod N = Out_Ptr) loop
      null;
   end loop;
   Buffer(In_Ptr) := D;
   In_Ptr := (In_Ptr+1) mod N;
end loop;
```

consumer

```
loop
   while(Out_Ptr = In_Ptr) loop
      null;
   end loop;
   D := Buffer(Out_Ptr);
   Out_Ptr := (Out_Ptr+1) mod N;
   Consume(D);
end loop;
```

4.4 Monitore

Wurden von *Hoare* 1974 vorgeschlagen und in verschiedenen Pascal-Abkömmlingen realisiert. Ausgangspunkt ist *Kritik* an *Semaphoren*: Zusammengehörendes *Paar* von Operationen, aber *verteilt* über Programmtext/Prozesse, also *Fehlergefahr* durch

- Operationsaufruf auslassen.
- Operation (wait/signal) verwechselt.
- Operand (S) verwechselt.

Stattdessen: Zuständigkeit für Schutz- und Synchronisationsmaßnahmen wird dem *Objekt* übertragen, das von mehreren Prozessen benutzt wird, ohne Kopplung.

Schnittstelle Zugriffsoperationen, *kein* Protokoll auf Prozeßseite

intern Implementierung durch Operationen *inklusive* Synchronisation.

Eine solche Datenstruktur/Objekt heißt *Monitor*.

Monitore sind *passiv*, d. h. werden bei Aufruf ihrer Operationen tätig, sind *nicht* selbst Prozesse. !

4.4.1 Beispiel: Producer-Consumer

Ada ähnliche Syntax, aber *nicht* Ada!

```
Monitor Buffer is
  procedure Insert(D: Data);
  procedure Take(D: out Data);
end Buffer;

Monitor body Buffer is
  N: Positive := ...;
  B: array(0..N-1) of Data;
  In_Ptr, Out_Ptr: natural := 0;
  Count: Natural := 0; -- Anzahl Elemente
  Not_Full, Not_Empty: Condition; -- Bedingungsvariablen

  procedure Insert(D: Data) is
  begin
     if Count = N then
        Wait(Not_Full);
     end if;
     B(In_Ptr) := D;
     In_Ptr := (In_Ptr+1) mod N;
     Count := Count+1;
     Signal(Not_Empty);
  end Insert;

  procedure Take(D: out Data) is
  begin
     if Count = 0 then
        Wait(Not_Empty);
     end if;
     D := B(Out_Ptr);
     Out_Ptr := (Out_Ptr+1) mod N;
     Count := Count-1;
     Signal(Not_Full);
  end Take;
end Buffer;
```

Gebrauch unit:
producer

```
D: Data;

begin
   loop
      Produce(D);
      Insert(D);
   end loop;
end;
```

consumer

```
D: Data;

begin
   loop
      Take(D);
      Consume(D);
   end loop;
end;
```

d. h. Synchronisation ist *nicht* Sache der Klienten!

4.4.2 Semantik der Monitor-Konstruktion

- Jeweils nur 1 Monitor Operation in Ausführung, d. h. *mutual exclusion.*
- Operationen auf *condition variables* (C)

  Wait(C) Ausführender Prozeß wird suspendiert und in eine mit C assozierte FIFO-queue eingereiht; andere Prozesse jetzt zugelassen zur Operationsausführung.

  Signal(C) Falls die mit C assozierte FIFO-queue nicht leer ist, wird der erste der wartenden Prozesse aufgeweckt.

  Is_Empty(C) True gdw. Queue für C nicht leer (reine Abfrage!)

4.4.3 Interpretation von Signal

Heikler Punkt! Wenn der erste auf die Bedingung wartende Prozeß aufgeweckt wird, sollte er *sofort* arbeiten können und *nicht* auf Beendigung des signalisierenden Prozesses warten (sonst: Bedingung vielleicht wieder False). Aber: Was tun mit signalisierendem Prozeß, wie hält man Ausschlußeigenschaft des Monitors aufrecht?

Sinnvolle Konvention: Signal *letztes* statement einer Prozedur und intern blockierte Prozesse haben Priorität über extern aufrufende!

Mehrere Festlegungen und Konventionen im Gebrauch! Wir verfolgen die Sache hier nicht weiter, sondern studieren (später) die Ada-Lösung! !

4.4.4 Kritik an Monitoren

Schwierig, eine Strategie für die Bedienung von Operationsaufrufen zu realisieren.

Kapitel 5

Zwischenbilanz, Ausblick Ada

- Bisher haben wir betrachtet: Prozesse interagieren mit Hilfe *gemeinsamer* Variablen ⇒ Protokolle und Schutzmaßnahmen nötig!
- Technischer/historischer Ausgangspunkt dafür: Rechner mit 1 Prozessor (meist, oder mehrere) und *1 Speicher*.

Aber: Technische Alternative ist ein *verteiltes* System mit *Nachrichtenaustausch* (message passing).

⇒ plausibel, konkurrente Programmierung an diesem technischen Paradigma zu orientieren.

Technische Basis nicht zwingend, man kann

- **eine Struktur durch eine anderen *simulieren*,**
- **eine aus Gründen der *Programmierung* bevorzugen!** !

5.1 Analogie

shared variables & processes	~	global variables & procedures
↓		↓
? *message passing* ? Typ der Nachricht, wer an wen, wenn ...	~	*explizite Schnittstelle* Parameterübergabe

5.2 Message passing

Möglich in verschiedenen Grundformen.

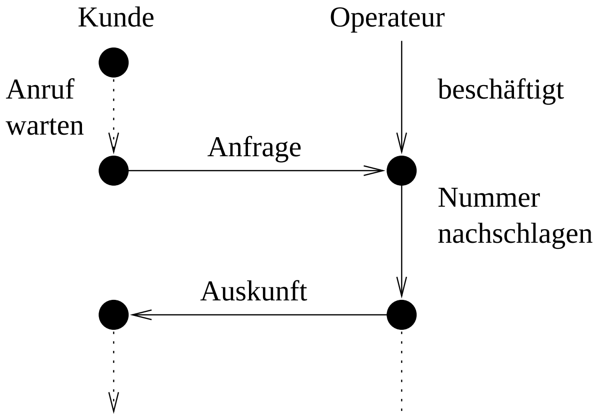
1. **asynchron (nicht blockierend)** Sender wartet nicht auf Empfang, sondern macht weiter.
   - **Vorteil** Kein Zeitverlust
   - **Nachteil** Rückfluß von Informationen nicht im selben Akt möglich; Info aktuell.
   - **Beispiel** Briefpost
2. **synchron** Gemeinsamer Akt von Sender und Empfänger, jeder muß auf anderen warten.
   - **Vorteil/Nachteil** siehe oben
   - **Beispiel** Telephon
3. Fest vorgegebene (uni-, bidirektionale) *Kanäle*.
4. Sender kennt Empfänger„anschrift“, nicht umgekehrt.
5. Broadcast („Rundfunk“, Info öffentlich)

5.2.1 Beispiele

2 + 3:occam
1 + 5:Linda
2 + 4:Ada Rendezvous

5.2.2 Die Techniken in Ada

Rendezvous Synchrone Kommunikation mit Möglichkeit zur Antwort und Hilfsberechnung; z. B. Telephonauskunft.



Protected objects Gekapselte passive Struktur, ähnlich Monitor.

Die verschiedenen für konkurrente Programmierung vorgeschlagenen / benutzten Konstrukte (Semaphore, Monitore, asynchrone/synchrone Nachrichten, ...) sind gleichwertig in dem Sinn, daß sie sich wechselseitig simulieren können; also nur Unterschiede in

- **Effizienz (je nach Kontext und Problemstellung)**
- **Sicherheit, Klarheit im Gebrauch**
- **Modellierungsqualität** !

Ada hatte ursprünglich nur rendezvous; protected objects in Ada95 weil manchmal

- **Klarer und einfacher im Gebrauch.**
- **Effizienter zu implementieren.** !

5.2.3 Hausaufgabe

- *Monitor* simuliert *Semaphore*.
- *Asynchrone* Nachrichtenkommunikation simuliert synchrone.

Kapitel 6

Ada Tasking II

Immer noch nicht volle Syntax, sondern nur Basisform von rendezvous und protected objects; mehr später. !

6.1 Tasks

- Haben ähnlich wie packages *Spezifikationsteil* und *Rumpf*, müssen aber in übergeordnetem Programmteil (z. B. Hauptprogramm, Paket) deklariert werden, sind also selber *nicht* Bibliothekseinheiten!
- Beinhalten wie Unterprogramme eigene *Deklarationen* und eine *Anweisungsfolge*, werden aber nicht durch Aufruf gestartet (und im Kontrollfluß des Aufrufers ausgeführt), sondern implizit und konkurrent zu anderen.
- Lassen sich wie Datenstrukturen unterscheiden in *Typ* (z.Ḃ. `type vector is array(1..n) of ...`) und zugehörige *Objekte* (z. B. `v1, v2: vector`), sind aber nicht passiv, sondern haben eigenen Kontrollfluß!

Wir betrachten den letzten Punkt. Bisher nur „Einmal-tasks“, die einen *anonymen* Typ haben!

```
task Hund;
task body Hund is
   print("wau wau");
end Hund;
```

Allgemein deklariert man den Typ mit

```
task type T is
   ...
end T;

task body T is
   ...
end T;
```

und die Objekte davon mit

```
t1, t2: T;
```

Task-Typen sind *limited*, d. h. Vergleich und Zuweisung ist nicht erlaubt. (Task Objekte also insbesondere konstant). !

Task Typen erlauben wie andere Typen die Bildung von *zusammengesetzten* Datenstrukturen, z. B.

```
T_Vector: array(1..10) of T;
```

und *dynamische Erzeugung*, z. B.

```
type T_pointer is access T;
p: T_pointer;
   ...
begin
   ...
   p := new T;
   ...
end;
```

6.1.1 Beispiele

- **Array of tasks** Gebäudeheizung mit *Sensor*-task für jeden Raum.
- **Dynamische Erzeugung** Luftraumüberwachung mit task für jedes gesteuerte Flugzeug.

6.1.2 Verschiedene Tasks desselben Typs

- Zeigen *gleichartiges* Verhalten.
- Haben ihre *eigenen* Daten und Kontrollfluß (speziell: 2 Tasks rufen dieselbe Prozedur auf ⇒ 2 Inkarnationen der Prozedur, wie bei Rekursion).

6.2 Aktivierung, Ausführung, Terminierung

- Ist ein komplexes Thema, weil so viele Kontexte möglich.
- Wir beschränken uns auf einfache/fundamentale Fälle (und warnen vor exotischen Konstruktionen).

6.2.1 Task Objekt deklariert in Hauptprogramm

z. B.

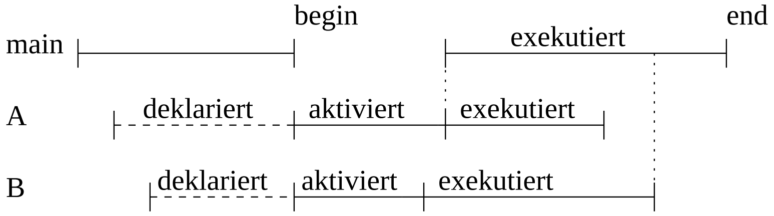
```
procedure Main is
  ...
  task type T;
  A: T;
  B: T;
  ...
begin
   ...
end;
```

dann

- Hauptprogramm wird angesehen als von einer „Umgebungs-Task" aufgerufen und ausgeführt. Dies ist die „*Mutter*" (der „ *Vater*") der im Hauptprogramm deklarierten Tasks, die von ihr *abhängen* („are dependent on").
- Deklarierte Task Objekte werden

  aktiviert Ihre internen Deklarationen werden elaboriert.

  exekutiert Anweisungsfolge wird ausgeführt.



Hauptprogramm arbeitet erst selbst, wenn Aktivierung aller Tasks gelungen ist. Grund: Fehler auffangen. !

Terminierung in 2 Stufen

completed Task erreicht Ende ihrer Anweisungsfolge.

terminated Completed + alle abhängigen Tasks sind terminiert ⇒ Hauptprogramm terminiert als letztes)

Spezialfälle z. B. Abort-statement (= andere Task abbrechen). **Unschön und wegen komplexer Semantik fehleranfällig.** Gibt's in Ada als letztes Mittel zur Behandlung von Ausnahmesituationen (= Task läuft Amok). Aber: Angenommen, es gibt böse Menschen (Räuber, Mörder, ...). Frage: Soll man dann Waffenerwerb *erleichtern* oder *erschweren*? !

6.2.2 Task deklariert in (library-) Package

- Wird aktiviert mit „begin" des Paketrumpfs; gibt es dort keinen Anweisungsteil, wird „`begin null; end;`" angenommen.
- Main-Environment-Task terminiert erst, wenn
  1. Hauptprogramm terminiert.
  2. Library (package) Task terminiert.

6.2.3 Task wird dynamisch erzeugt durch Allokator

- Aktivierung während Ausführung der Allokationsanweisung, dann Ausführung konkurrent zum Erzeuger.
- Solche Tasks hängen *nicht* ab vom Erzeuger, sondern von Programmeinheit, die den zugehörigen *access-Typ* eingeführt hat!

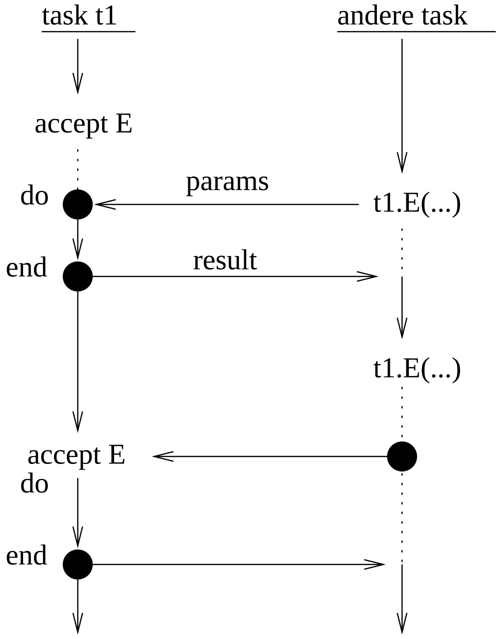
6.3 Rendezvous

- Ist Mittel zur Synchronisation *und* Kommunikation.
- Geschieht über im Task-Spezifikationsteil deklarierte „entries"

```
task type T is
   entry E(<Param. Spec.>);
end T;

task body T is
   ...
   accept E(<Param. Spec.>) do
     ...
   end E;
end T;
```

- **Gebrauch** Sei `T1:T;`, dann Aufruf des entries mit `T1.E(<Actuals>);`
- **Wirkung**
  – Parameterübergabe (in, out, inout) ähnlich Prozedur.
  – Aufrufer wartet während Ausführung des accept-statements, ähnlich Prozedur.
  – Aufrufer wartet, *bis* Aufgerufener accept-statement ausführt.

– Aufgerufener wartet, bis jemand entry aufruft.



- Task body kann *mehrere* accept-statements für dasselbe entry enthalten (insbesondere in Schleife – typischer Fall einer Server-Task).
- Entry kann von mehreren Tasks gerufen werden ⇒ Warteschlange!
- Accept-statement muß direkt im task body stehen, aber nicht in Unterprogramm. (Aufruf aber durchaus in Unterprogramm. Möglich und manchmal sinnvoll: `procedure ...`*`renames`*` entry ...`).
- Accept statement sollte nur Anweisungen enthalten, die für das Rendezvous (Ergebnisrückgabe) unbedingt nötig sind – also den Anrufer nicht sinnlos verzögern.
- Entries *ohne* Parameter sind nützlich für Synchronisationszwecke, z. B.

```
----------------------------------------------------------------------
-- Demo: rendezvous zur Synchronisation
----------------------------------------------------------------------

with stringpack; use Stringpack;
procedure Tierwelt2 is

   task type Hund is
      entry Los;
   end Hund;

   task body Hund is
   begin
      accept Los;
      Print("wau");
      Print("wau");
   end Hund;

   Rex, Pluto, Fifi: Hund;

   task Katz is
      entry Los;
      entry Nochmal;
   end Katz;
```

```
      task body Katz is
      begin
         accept Los;
         Print("miau");
         accept Nochmal;
         Print("miau");
      end Katz;

   begin
      Print("hello world");
      Rex.Los; Fifi.Los; Katz.Los; Pluto.Los; Katz.Nochmal;
   end Tierwelt2;
```

Synchronisation mittels Rendezvous kann man benutzen, um

- Andere Synchronisationsmittel wie Semaphore zu realisieren, oder
- die mit z. B. Semaphoren lösbaren Aufgaben *direkt* anzugehen.

Letzteres ist vorzuziehen, wobei – wie später besprochen –

- **Verfeinerung der Rendezvous Konstruktion**
- **Protected types**

wichtig sind für gute Lösung. !

Hier einstweilen eine *schlechte* Lösung.

- Imitiere (Binäres-) Semaphor.
- Löse damit Problem des wechselseitigen Ausschlusses.

6.3.1 Beispiel: Imitation einer binären Semaphore

Programm von 3.4 (Seite 13); Tasks „Up“ und „Down“ manipulieren gemeinsame Variable.

So nicht! !

```
with Stringpack, Ada.Calendar;
use Stringpack, Ada.Calendar;

procedure Shared_Demo3 is

   Start_Work, End_Work: Time;

   N: Integer := 0;  --shared variable
   Anz: Positive := 1_000_000;

   task Binary_Semaphore is
      entry Wait;
      entry Signal;
   end Binary_Semaphore;

   task body Binary_Semaphore is
   begin
      loop
         accept Wait;
```

```
         accept Signal;
      end loop;
   end Binary_Semaphore;

   task Up is
      entry Start;
      entry Finish;
   end Up;

   task body Up is
      My_N: Integer;
   begin
      accept Start;
      for I in 1..Anz loop
         Binary_Semaphore.Wait;
         My_N := N;
         My_N := My_N+1;
         N := My_N;
         Binary_Semaphore.Signal;
      end loop;
      accept Finish;
   end Up;

   task Down is
      entry Start;
      entry Finish;
   end Down;

   task body Down is
      My_N: Integer;
   begin
      accept Start;
      for I in 1..Anz loop
         Binary_Semaphore.Wait;
         My_N := N;
         My_N := My_N-1;
         N := My_N;
         Binary_Semaphore.Signal;
      end loop;
      accept Finish;
   end Down;

begin
   Start_Work := Clock;  -- Zeiterfassung
   Up.Start; Down.Start;
   Up.Finish; Down.Finish;
   Print("n= " & N);
   End_Work := Clock;    -- Zeiterfassung
   Print("time spent= " & Float(Ada.Calendar."-"(End_Work, Start_Work)));
   abort Binary_Semaphore;  -- sehr schlechter Stil !!!!!!!!!!!!!!!
end Shared_Demo3;

----------------------------------------------------------------------
11 pohlmann@pavian:shared_demo3
n= 0
time spent= 1086.25

rbla@bravo:shared_demo3
```

```
n= 0
time spent= 424.29 (AMD K6, 200MHz, GNAT, Linux)
```

6.3.2 Zur Zeitmessung

Hier: Einfache, ungenaue(!) Methode: Gemessen wird „*Wall-clock-time*“, also *nicht* die Rechenzeit, die speziell auf *mein* Programm entfällt.

Methode Ada.Calendar

mit `type time` (Datum, Uhrzeit) wobei Uhrzeit = Sekunden und einschlägige Arithmetik.

und `function clock` current time

```
package Ada.Calendar is

-> type Time is private;

   subtype Year_Number  is Integer range 1901 .. 2099;
   subtype Month_Number is Integer range 1 .. 12;
   subtype Day_Number   is Integer range 1 .. 31;

   subtype Day_Duration is Duration range 0.0 .. 86_400.0;

-> function Clock return Time;

   function Year    (Date : Time) return Year_Number;
   function Month   (Date : Time) return Month_Number;
   function Day     (Date : Time) return Day_Number;
   function Seconds (Date : Time) return Day_Duration;

   procedure Split
     (Date    : Time;
      Year    : out Year_Number;
      Month   : out Month_Number;
      Day     : out Day_Number;
      Seconds : out Day_Duration);

   function Time_Of
     (Year    : Year_Number;
      Month   : Month_Number;
      Day     : Day_Number;
      Seconds : Day_Duration := 0.0)
      return    Time;

   function "+" (Left : Time;     Right : Duration) return Time;
   function "+" (Left : Duration; Right : Time)     return Time;
   function "-" (Left : Time;     Right : Duration) return Time;
-> function "-" (Left : Time;     Right : Time)     return Duration;

   function "<"  (Left, Right : Time) return Boolean;
   function "<=" (Left, Right : Time) return Boolean;
   function ">"  (Left, Right : Time) return Boolean;
   function ">=" (Left, Right : Time) return Boolean;

   Time_Error : exception;

private
```

```
   -- implementation-dependent
end Ada.Calendar;
```

Bessere Lösung ⇒ später.

6.4 Beispiel für Kooperation von Tasks

Berechne $\sum_{i=1}^{n} i$ durch tatsächliches Addieren, und verteile die Arbeit unter mehreren Task!

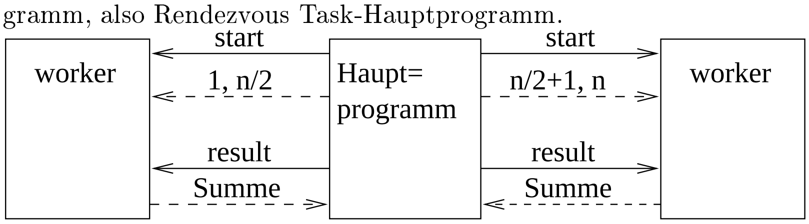
Beispiel ist an sich *nicht* sinnvoll weil

- **Formel,**
- **Arbeitsteilung nur bei echter Parallelität schneller,**

aber hier Zweck: Zeige verschiedene Möglichkeiten der Interaktion durch Rendezvous! !

6.4.1 Beispiel 1

2 Tasks, jeder verantwortlich für halbe Summe, Start und Zusammenfassung der Ergebnisse durch Hauptprogramm, also Rendezvous Task-Hauptprogramm.

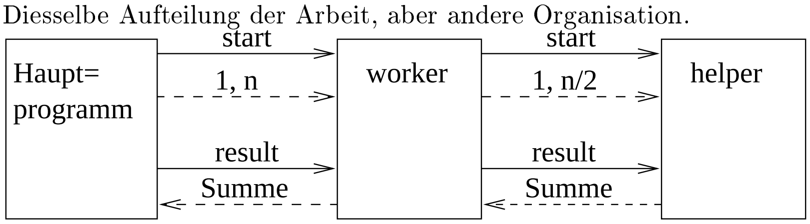



```
----------------------------------------------------------------------
-- Aufgabe: summiere 1..n.
-- Methode: tasks fuer Teilsummen 1..n/2, n/2+1..n,
--          Start u. Zusammenfassung durch Hauptprogramm
----------------------------------------------------------------------
with Stringpack; use Stringpack;
procedure Arbeitsteilung is

   task type Worker is
      entry Start(From, To: Integer);
      entry Result(Res: out Integer);
   end Worker;

   task body Worker is
      Local_From, Local_To, Local_Res: Integer;
   begin
      accept Start(From, To: Integer) do
         Local_From := From;
         Local_To := To;
      end Start;
      Local_Res := 0;
      for I in Local_From..Local_To loop
         Local_Res := Local_Res+I;
      end loop;
      accept Result(Res: out Integer) do
         Res := Local_Res;
      end Result;
   end Worker;

   W1, W2: Worker;
```

```
   N, Sum1, Sum2: Integer;

begin
   Print("give natural");
   N := Getint;
   W1.Start(1, N/2);
   W2.Start(N/2 +1, N);
   W1.Result(Sum1);
   W2.Result(Sum2);
   Print("Summe= " & (Sum1 + Sum2));
end Arbeitsteilung;

----------------------------------------------------------------------
7 pohlmann@pavian:Arbeitsteilung
give natural
100
Summe= 5050
```

6.4.2 Beispiel 2

Diesselbe Aufteilung der Arbeit, aber andere Organisation.




```
----------------------------------------------------------------------
-- Aufgabe: summiere 1..n.
-- Methode: tasks fuer Teilsummen 1..n/2, n/2+1..n
--          erste task startet die zweite und bildet Gesamtergebnis
----------------------------------------------------------------------

with Stringpack; use Stringpack;
procedure Arbeitsteilung2 is

   task Worker is
      entry Start(From, To: Integer);
      entry Result(Res: out Integer);
   end Worker;

   task Helper is
      entry Start(From, To: Integer);
      entry Result(Res: out Integer);
   end Helper;

   task body Worker is
      Local_From, Local_To, Local_Res, Remote_Res: Integer;
   begin
      accept Start(From, To: Integer) do
         Local_From := From;
         Local_To := To;
      end Start;
      Helper.Start(Local_From, (Local_From + Local_To)/2);
      Local_Res := 0;
      for I in ((Local_From + Local_To)/2 +1)..Local_To loop
         Local_Res := Local_Res+I;
```

```
      end loop;
      Helper.Result(Remote_Res);
      accept Result(Res: out Integer) do
         Res := Local_Res + Remote_Res;
      end Result;
   end Worker;

   task body Helper is
      Local_From, Local_To, Local_Res: Integer;
   begin
      accept Start(From, To: Integer) do
         Local_From := From;
         Local_To := To;
      end Start;
      Local_Res := 0;
      for I in Local_From..Local_To loop
         Local_Res := Local_Res+I;
      end loop;
      accept Result(Res: out Integer) do
         Res := Local_Res;
      end Result;
   end Helper;

   N, Sum: Integer;

begin
   Print("give natural");
   N := Getint;
   Worker.Start(1, N);
   Worker.Result(Sum);
   Print("Summe= " & Sum);
end Arbeitsteilung2;

----------------------------------------------------------------------
16 pohlmann@pavian:Arbeitsteilung2
give natural
100
Summe= 5050
```

6.4.3 Beispiel 3

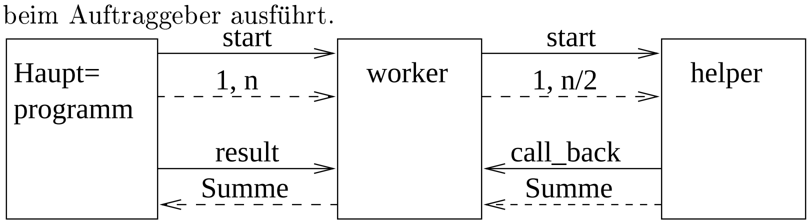
Dieselbe Teilung der Arbeit, Organisation und Datenfluß fast genauso: Aber jetzt „Rendezvous zu Dritt" beim abschließenden Resultat einsammeln.

```
----------------------------------------------------------------------
-- Aufgabe: summiere 1..n.
-- Methode: tasks fuer Teilsummen 1..n/2, n/2+1..n
--          erste task startet die zweite und bildet Gesamtergebnis
--                                        mit three-way-rendezvous
----------------------------------------------------------------------

with Stringpack; use Stringpack;
procedure Arbeitsteilung2a is

   task Worker is
      entry Start(From, To: Integer);
      entry Result(Res: out Integer);
   end Worker;
```

```
   task Helper is
      entry Start(From, To: Integer);
      entry Result(Res: out Integer);
   end Helper;

   task body Worker is
      Local_From, Local_To, Local_Res, Remote_Res: Integer;
   begin
      accept Start(From, To: Integer) do
         Local_From := From;
         Local_To := To;
      end Start;
      Helper.Start(Local_From, (Local_From + Local_To)/2);
      Local_Res := 0;
      for I in ((Local_From + Local_To)/2 +1)..Local_To loop
         Local_Res := Local_Res+I;
      end loop;
      accept Result(Res: out Integer) do  --Rendezvous zu dritt
         Helper.Result(Remote_Res);
         Res := Local_Res + Remote_Res;
      end Result;
   end Worker;

   task body Helper is
      Local_From, Local_To, Local_Res: Integer;
   begin
      accept Start(From, To: Integer) do
         Local_From := From;
         Local_To := To;
      end Start;
      Local_Res := 0;
      for I in Local_From..Local_To loop
         Local_Res := Local_Res+I;
      end loop;
      accept Result(Res: out Integer) do
         Res := Local_Res;
      end Result;
   end Helper;

   N, Sum: Integer;

begin
   Print("give natural");
   N := Getint;
   Worker.Start(1, N);
   Worker.Result(Sum);
   Print("Summe= " & Sum);
end Arbeitsteilung2a;

----------------------------------------------------------------------
30 pohlmann@pavian:Arbeitsteilung2a
give natural
100
Summe= 5050
```

6.4.4 Beispiel 4

Dieselbe Teilung der Arbeit und Grundstruktur, aber jetzt gibt die Helper-Task das Ergebnis *aktiv* zurück. Dazu erhält die Helper-Task beim Start eine sog. „call-back“ Prozedur, die Rückruf (bei entsprechendem entry) beim Auftraggeber ausführt.




```
----------------------------------------------------------------------
-- Aufgabe: summiere 1..n.
-- Methode: tasks fuer Teilsummen 1..n/2, n/2+1..n
--          erste task startet die zweite und bildet Gesamtergebnis
--                                    mit call-back procedure
----------------------------------------------------------------------

with Stringpack; use Stringpack;
procedure Arbeitsteilung2b is

   task Worker is
      entry Start(From, To: Integer);
      entry Result(Res: out Integer);
      entry Call_Back(Res: Integer);
   end Worker;

   type Call_Back_Proc is access procedure(X: Integer);

   procedure Call_Back_Worker(R: Integer) is --proc must not be deeper than access type
   begin
      Worker.Call_Back(R);
   end Call_Back_Worker;

   task Helper is
      entry Start(From, To: Integer; Cb: Call_Back_Proc);
   end Helper;

   task body Worker is
      Local_From, Local_To, Local_Res, Remote_Res: Integer;
   begin
      accept Start(From, To: Integer) do
         Local_From := From;
         Local_To := To;
      end Start;
      Helper.Start(Local_From, (Local_From + Local_To)/2, Call_Back_Worker'access);
      Local_Res := 0;
      for I in ((Local_From + Local_To)/2 +1)..Local_To loop
         Local_Res := Local_Res+I;
      end loop;
      accept Call_Back(Res: Integer) do
         Remote_Res := Res;
      end Call_Back;
      accept Result(Res: out Integer) do
         Res := Local_Res + Remote_Res;
      end Result;
   end Worker;
```

```
   task body Helper is
      Local_From, Local_To, Local_Res: Integer;
      My_Call_Back_Proc: Call_Back_Proc;
   begin
      accept Start(From, To: Integer; Cb: Call_Back_Proc) do
         Local_From := From;
         Local_To := To;
         My_Call_Back_Proc := Cb;
      end Start;
      Local_Res := 0;
      for I in Local_From..Local_To loop
         Local_Res := Local_Res+I;
      end loop;
      My_Call_Back_Proc(Local_Res);
   end Helper;

   N, Sum: Integer;

begin
   Print("give natural");
   N := Getint;
   Worker.Start(1, N);
   Worker.Result(Sum);
   Print("Summe= " & Sum);
end Arbeitsteilung2b;

----------------------------------------------------------------------
47 pohlmann@pavian:Arbeitsteilung2b
give natural
100
Summe= 5050
```

Klar: Helper Task könnte auch direkt entsprechendes entry von Worker Task aufrufen, aber dann wäre das Rückgabe-Ziel fest in den Programmtext von Helper zu schreiben, d. h. Helper könnte nicht von *mehreren* Klienten-Tasks benutzt werden!

6.4.5 Beispiel 5

Ähnlich Beispiel 2, aber Helper nach Bedarf *dynamisch* erzeugt.

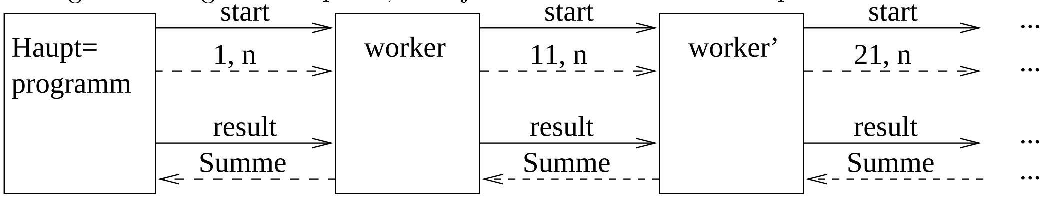
```
----------------------------------------------------------------------
-- Aufgabe: summiere 1..n.
-- Methode: tasks fuer Teilsummen 1..n/2, n/2+1..n falls n>50,
--          erste task erzeugt und startet die zweite und bildet
--          Gesamtergebnis
----------------------------------------------------------------------

with Stringpack; use Stringpack;
procedure Arbeitsteilung3 is

   task Worker is
      entry Start(From, To: Integer);
      entry Result(Res: out Integer);
   end Worker;

   task type Helper is
      entry Start(From, To: Integer);
      entry Result(Res: out Integer);
   end Helper;
```

```
   type Helper_Ptr is access Helper;

   task body Worker is
      Local_From, Local_To: Integer;
      Local_Res, Remote_Res: Integer := 0;
      H: Helper_Ptr;
   begin
      accept Start(From, To: Integer) do
         Local_From := From;
         Local_To := To;
      end Start;
      if Local_To-Local_From >50 then
         H := new Helper;
         H.Start(Local_From, (Local_From+Local_To)/2);
         for I in ((Local_From + Local_To)/2 +1)..Local_To loop
            Local_Res := Local_Res+I;
         end loop;
         H.Result(Remote_Res);
      else
         for I in Local_From+Local_To..Local_To loop
            Local_Res := Local_Res + 1;
         end loop;
      end if;
      accept Result(Res: out Integer) do
         Res := Local_Res + Remote_Res;
      end Result;
   end Worker;

   task body Helper is
      Local_From, Local_To, Local_Res: Integer;
   begin
      accept Start(From, To: Integer) do
         Local_From := From;
         Local_To := To;
      end Start;
      Local_Res := 0;
      for I in Local_From..Local_To loop
         Local_Res := Local_Res+I;
      end loop;
      accept Result(Res: out Integer) do
         Res := Local_Res;
      end Result;
   end Helper;

   N, Sum: Integer;

begin
   Print("give natural");
   N := Getint;
   Worker.Start(1, N);
   Worker.Result(Sum);
   Print("Summe= " & Sum);
end Arbeitsteilung3;

----------------------------------------------------------------------
21 pohlmann@pavian:Arbeitsteilung3
give natural
```

```
100
Summe= 5050
```

6.4.6 Beispiel 6

Verallgemeinerung von Beispiel 5, aber jetzt zusätzliche Tasks entsprechend Bedarf *rekursiv* erzeugt.




```
----------------------------------------------------------------------
-- Aufgabe: summiere 1..n.
-- Methode: tasks fuer Teilsummen aus 10 Elementen,
--          tasks nach Bedarf rekursiv erzeugt und gestartet,
--          Ergebnis wandert zurueck durch Erzeugertasks
----------------------------------------------------------------------

with Stringpack; use Stringpack;
procedure Arbeitsteilung4 is

   task type Worker is
      entry Start(From, To: Integer);
      entry Result(Res: out Integer);
   end Worker;

   type Worker_Ptr is access Worker;

   function Get_New_Worker return Worker_Ptr is --Hilfsfunktion, erzeugt neue task
   begin
      return new Worker;
   end Get_New_Worker;

   task body Worker is
      Local_From, Local_To: Integer;
      Local_Res, Remote_Res: Integer := 0;
      Colleague: Worker_Ptr;
   begin
      accept Start(From, To: Integer) do
         Local_From := From;
         Local_To := To;
      end Start;
      if Local_To-Local_From >10 then
         Colleague := Get_New_Worker; --task type cannot be used as type mark within its own body
         Colleague.Start(Local_From+10, Local_To);
         for I in Local_From..(Local_From+9) loop
            Local_Res := Local_Res+I;
         end loop;
         Colleague.Result(Remote_Res);
      else
         for I in Local_From..Local_To loop
            Local_Res := Local_Res+I;
         end loop;
      end if;
      accept Result(Res: out Integer) do
         Res := Local_Res+Remote_Res;
```

```
      end Result;
   end Worker;

   N, Sum: Integer;
   W: Worker_Ptr := new Worker;

begin
   Print("give natural");
   N := Getint;
   W.Start(1, N);
   W.Result(Sum);
   Print("Summe= " & Sum);
end Arbeitsteilung4;

----------------------------------------------------------------------
49 pohlmann@pavian:Arbeitsteilung4
give natural
100
Summe= 5050
53 pohlmann@pavian:Arbeitsteilung4
give natural
1000
Summe= 500500
54 pohlmann@pavian:Arbeitsteilung4
give natural
1000000
^C       -- nach 5 Minuten warten ...................................
```

Problem

1. Zeit?
2. System verträgt nicht so viele (100000) Prozesse?

```
----------------------------------------------------------------------
-- Demo: Anzahl tasks
----------------------------------------------------------------------

with Stringpack, Ada.Calendar;
use Stringpack, Ada.Calendar;
procedure Task_Nr_Demo is

   N: Natural;
   Start_Work, End_Work: Time;

   task type T is
      entry Start;
      entry Finish;
   end T;

   task body T is
   begin
      accept Start;
      accept Finish;
   end T;

begin
   Print("give natural");
```

```
   N := Getint;
   Start_Work := Clock;
   declare
      Ts: array(1..N) of T;
   begin
      End_Work := Clock;  --for declare tasks
      Print("time spent on decl= " & Float(Ada.Calendar."-"(End_Work, Start_Work)));
      for I in Ts'Range loop Ts(I).Start; end loop;
      for I in Ts'Range loop Ts(I).Finish; end loop;
   end;
   End_Work := Clock;     --for everything
   Print("time spent on everything= " & Float(Ada.Calendar."-"(End_Work, Start_Work)));
end Task_Nr_Demo;

----------------------------------------------------------------------
10 pohlmann@pavian:task_nr_demo
give natural
100
time spent on decl= 0.16
time spent on everything= 0.34
11 pohlmann@pavian:task_nr_demo
give natural
1000
time spent on decl= 2.29
time spent on everything= 7.57
12 pohlmann@pavian:task_nr_demo
give natural
10000
^C      -- nach 15 min...............................................
```

6.4.7 Hausaufgabe

Schreiben Sie ein Programm, in dem mehrere Tasks zu einer *Fließband-* (oder pipeline-) *Verarbeitung* kombiniert sind, also

in ⟶ $\boxed{\text{t1}}$ ⟶ $\boxed{\text{t2}}$ ⟶ ··· $\boxed{\text{tn}}$ ⟶ out

Beispiel für sowas könnte sein:

Textverarbeitung

- Überflüssige blanks und Formatierungsinformationen entfernen,
- neue Formatierung vornehmen.

6.5 Protected types & protected objects

Objekte mit prozedualer Schnittstelle und Aufrufkontrolle (mutual exclusion).

6.5.1 Syntax für Deklaration

```
protected type P is
  <protected Operations>
  private
     <protected Data Declarations>;
end P;

protected body P is
```

```
   <Impl. protected Operations>
end P;
```

- Es gibt wieder „einmal“ Objekte vom anonymen Typ.
- Der private-Teil mit den Attributen des Objekts kann auch fehlen.
- Gebrauch wie üblich mit Typen – z. B. in Objektdeklaration `p1: P;` – **Protected types sind limited, d. h. kein Vergleich, keine Zuweisung.** !

Beispiel

```
protected Flag is
  procedure Up;
  procedure Down;
  function Read return Boolean;
private
  The_Flag: Boolean := True;
end Flag;

protected body Flag is
   procedure Up is
   begin
      The_Flag := Ture;
   end;
   ...
end Flag;
```

6.5.2 Semantik

Protected procedures Erlauben zustandsändernden Zugriff und werden unter *wechselseitigem Ausschluß* ausgeführt, d. h. Prozedur hat Objekt für sich allein, andere Aufrufe einer Operation warten.

Protected functions Mehrere Aufrufe gleichzeitig möglich, da nur „read-only“. **Aber nicht zugleich mit Prozedur!** !

Mit protected objects lassen sich mutual exclusion Probleme einfach und effizient lösen, siehe Beispiel auf Seite 48.

Beispiel

```
with Stringpack, Ada.Calendar;
use Stringpack, Ada.Calendar;
procedure Shared_Demo4 is

   Start_Work, End_Work: Time;
   Anz: Positive := 1_000_000;

   protected Shared_N is
      procedure Incr_N;
      procedure Decr_N;
      function Give_N return Integer;
   private
      N: Integer := 0;
   end Shared_N;

   protected body Shared_N is
```

```
      procedure Incr_N is
      begin
         N := N+1;
      end Incr_N;

      procedure Decr_N is
      begin
         N := N-1;
      end Decr_N;

      function Give_N return Integer is
      begin
         return N;
      end Give_N;
   end Shared_N;

   task Up is
      entry Start;
      entry Finish;
   end Up;

   task body Up is
   begin
      accept Start;
      for I in 1..Anz loop
         Shared_N.Incr_N;
      end loop;
      accept Finish;
   end Up;

   task Down is
      entry Start;
      entry Finish;
   end Down;

   task body Down is
   begin
      accept Start;
      for I in 1..Anz loop
         Shared_N.Decr_N;
      end loop;
      accept Finish;
   end Down;

begin
   Start_Work := Clock;  --Zeiterfassung
   Up.Start; Down.Start;
   Up.Finish; Down.Finish;
   Print("n= " & Shared_N.Give_N);
   End_Work := Clock;  --Zeiterfassung
   Print("time spent= " & Float(Ada.Calendar."-"(End_Work, Start_Work)));
end Shared_Demo4;

----------------------------------------------------------------------
3 pohlmann@pavian: shared_demo4
n= 0
time spent= 14.45
```

6.5.3 Wichtige Einschränkung !

Protected operations dürfen *nicht „potentiell blockierend“* sein, d. h. nicht dazu führen, daß die aufrufende task suspendiert werden muß, also ein Prozeßwechsel nötig ist.
Beispiel Protected operation ruft *entry* einer task auf! (denn dann muß die andere task dran kommen).

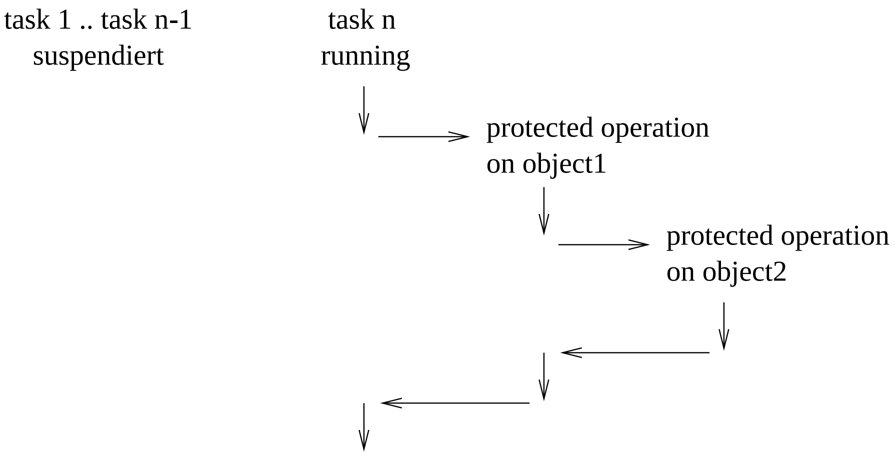
Grund Andernfalls Komplikationen mit

Programmlogik z. B. deadlock Gefahr.

***Implementierung* des wechselseitigen Ausschlusses für protected operations** Unter der genannten Einschränkung kann 1-Prozessor Implementierung darin bestehen, daß die, eine protected Operation aufrufende, task bis zur Beendigung der Operation *nicht* unterbrochen wird. ⇒ Effizienz.

aber erlaubt

- Aufruf von protected operation (eines *anderen* protected objects) *innerhalb* einer protected operation, denn das alles bei 1-Prozessor Implementierung ununterbrechbar.



Verstöße gegen die Regel können nicht immer erkannt werden (wenn erkannt: Warnung des Compilers, Laufzeitfehler program error). !

6.5.4 Bedingungssynchronisation (conditional synchronisation)

Operation kann nur ausgeführt werden, wenn betroffenes Objekt in einem bestimmten Zustand ist.

Beispiel: Puffer schreiben erfordert: Puffer nicht voll.
Dafür erlauben protected objects:

Protected entries Ähnlich wie protected procedures (Parameter, Ausschluß) aber zusätzlich mit *Barriere* („barrier“, „guard“ = Boolscher Ausdruck) die true sein muß, damit Operation ausgeführt wird.

Beispiel

```
protected Bahnuebergang is
   procedure Oeffne;
   procedure Schliesse;
   entry Fahre_Weiter;

private
   Offen: Boolean := True;
end Bahnuebergang;

protected body Bahnuebergang is
```

```
   procedure Oeffne is
   begin
      Offen := True;
   end Oeffne;

   Procedure Schliesse is
   begin
     Offen := False;
   end Schliesse;

   entry Fahre_Weiter
     when Offen is        --Barriere
   begin
      null;
   end;
end Bahnuebergang;
```

Wirkung:

- Bei Aufruf des entry wird Barriere *ausgewertet.*

  ```
  falls true
     dann Rumpf des entry ausführen;
     sonst Aufrufer wartend setzen;
  ```

- Bei Änderung des Zustands (= Ausführung von protected procedure oder entry) wird Barriere *wieder ausgewertet.*

  ```
  wenn true
     dann lasse einen der darauf wartenden Prozesse weitermachen;
  ```

  Unbedingt beachten: tasks, die „innerhalb“ des protected objects auf Barriere warten, haben *Vorrang* vor tasks, die „von außen“ protected objects aufrufen. !

Barrieren (Boolsche Ausdrücke)

- Dürfen *nicht* Parameter des entry benutzen.
- Dürfen globale Variablen enthalten, was *nicht* zu empfehlen ist!

Beispiel *Producer-Consumer-Problem* mit endlichem *Puffer* (vgl. 4.3, Seite 25).

Producer kann nur schreiben, wenn Puffer nicht voll,

Consumer kann nur lesen, wenn Puffer nicht leer.

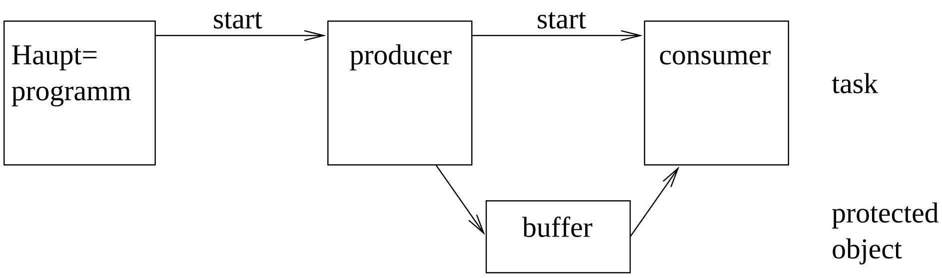
im Beispiel:

Producer erzeuge 1...n

Consumer summiere Zahlen

```
---------------------------------------------------------------------
-- Aufgabe: producer-consumer (summation_of_naturals) with bounded buffer
-- Methode: tasks for producer-consumer (simple sum_of_naturals problem)
--          protected object for buffer
---------------------------------------------------------------------

with Stringpack; use Stringpack;
procedure Bounded_Buffer is

   Length: constant Natural := 10;
   type Buff_Type is array(0..Length-1) of Natural;

   protected Buffer is
      entry Insert(D: Natural);
      entry Take(D: out Natural);
   private
      B: Buff_Type;
      In_Ptr, Out_Ptr: Natural := 0;
      Count: Natural := 0;
   end Buffer;

   task Producer is
      entry Start(How_Many: Natural);
   end Producer;

   task Consumer is
      entry Start(Break: Natural);
   end Consumer;

   task body Producer is
      Local_How_Many: Natural;
   begin
      accept Start(How_Many: Natural) do
         Local_How_Many := How_Many;
      end Start;

      Consumer.Start(Local_How_Many+1);

      for I in 1.. (Local_How_Many+1) loop
         Buffer.Insert(I);
      end loop;
   end Producer;

   task body Consumer is
      Over, Item, Result: Natural;
   begin
      accept Start(Break: Natural) do
         Over := Break;
      end Start;

      Result := 0;
      Buffer.Take(Item);
      while Item /= Over loop
         Result := Result+Item;
         Buffer.Take(Item);
      end loop;
      Print("Summe= " & Result);
   end Consumer;
```

```
   protected body Buffer is
      entry Insert(D: Natural)
        when Count < Length is
      begin
         B(In_Ptr) := D;
         In_Ptr := (In_Ptr +1) mod Length;
         Count := Count +1;
      end Insert;

      entry Take(D: out Natural)
        when Count > 0 is
      begin
         D := B(Out_Ptr);
         Out_Ptr := (Out_Ptr +1) mod Length;
         Count := Count -1;
      end Take;
   end Buffer;

   N: Natural;

begin
   Print("give natural");
   N := Getint;
   Producer.Start(N);
end Bounded_Buffer;

----------------------------------------------------------------------
11 pohlmann@pavian:bounded_buffer
give natural
100
Summe= 5050
```



6.6 Tasking & Exceptions

Ist wieder hochkomplizierte Materie; hier die fundamentals. Als Neuerung gibt es:

6.6.1 tasking-error

Ausgelöst („raised") wenn

Beliebige exception *während task Aktivierung* Dann:

- Task wird sofort *completed* (= Totgeburt)
- Tasking_error wird ausgelöst im *Eltern-task* (⇒ Begräbnis organisieren)

Beispiel:

```
with Stringpack; use Stringpack;
procedure Exception_Demo1 is

   task T;

   task body T is
      function F return Natural is begin return 0; end F;
      N: Natural := 1 mod F;         -- 1/0 in init!
   begin
      null;
   end T;

begin
   null;
exception
   when Tasking_Error => Print("Fehler bei Aktivierung");
end Exception_Demo1;

----------------------------------------------------------------------
6 pohlmann@pavian:exception_demo1
Fehler bei Aktivierung
```

Tasking_error wird weiters ausgelöst bei

Kommunikationsproblemen Aufruf eines entry einer task, die schon „completed ist", also den Aufruf nicht mehr entgegennehmen kann (= kein Anschluß unter dieser Nummer).

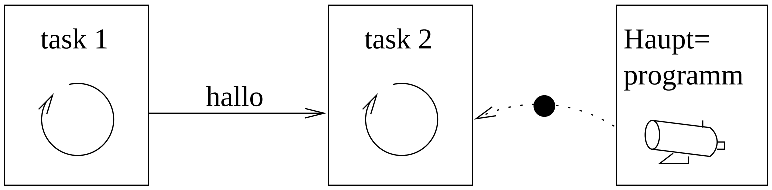
Dann: *Tasking_error* wird ausgelöst bei *Aufrufer*.

Für diesen Fall ist es egal, ob aufgerufener task *normal* endete oder z. B. mit „*abort*" abgeschossen wurde. !

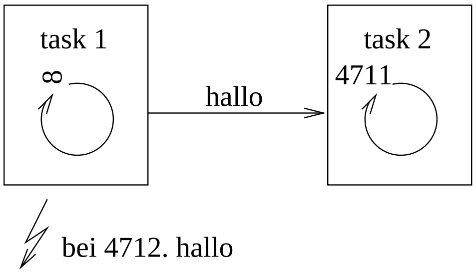
Beispiel

```
with Stringpack; use Stringpack;
procedure Exception_Demo2 is

   task T1;
   task T2 is
      entry Hallo;
   end T2;

   task body T1 is
   begin
      loop
         T2.Hallo;
      end loop;

   exception
      when Tasking_Error => Print("Fehler bei entry call");
   end T1;

   task body T2 is
   begin
      loop
         accept Hallo;
      end loop;
   end T2;
```

```
begin
   abort T2;
end Exception_Demo2;

----------------------------------------------------------------------
9 pohlmann@pavian:exception_demo2
Fehler bei entry call
```




```
with Stringpack; use Stringpack;
procedure Exception_Demo2a is

   task T1;
   task T2 is
      entry Hallo;
   end T2;

   task body T1 is
   begin
      loop
         T2.Hallo;
      end loop;

   exception
      when Tasking_Error => Print("Fehler bei entry call");
   end T1;

   task body T2 is
   begin
      for I in 1..4711 loop
         accept Hallo;
      end loop;
   end T2;

begin
   null;
end Exception_Demo2a;

----------------------------------------------------------------------
3 pohlmann@pavian:exception_demo2a
Fehler bei entry call
```



Jetzt umgekehrte Perspektive: wie *wirkt* das Auslösen einer exception *auf* task?

Exception ausgelöst durch gewöhnliche *Anweisung* im *Rumpf* einer task

```
falls task besitzt passenden handler
   dann führe handler aus;
        beende task;
   andernfalls task stirbt;
```

Man beachte: Im 2. Fall stirbt die task sang- und klanglos, *ohne* letzte Worte, Testament, etc., *ohne* Propagation der exception irgendwohin (Grund: task $\sim$ selbständiges Programm). $\Rightarrow$ zumindest zum *Debuggen* versehe man jede task mit einem exception handler, der Tod und Todesursache anzeigt! !

```
with Stringpack; use Stringpack;
procedure Exception_Demo3 is

   task T;

   task body T is
      function F return Natural is begin return 0; end F;
      N: Natural;
   begin
      N := 1 mod F;     -- 1/0 in body => exiting
      Print("alles ok");
   end T;

begin
   Print("fertig");

exception
   when others => Print("Fehler in task");
end Exception_Demo3;

----------------------------------------------------------------------
5 pohlmann@pavian:exception_demo3
fertig
```

Keiner hat's gemerkt! !

```
with Stringpack; use Stringpack;
procedure Exception_Demo4 is

   task T;

   task body T is
      function F return Natural is begin return 0; end F;
      N: Natural;
   begin
      N := 1 mod F;     -- 1/0 in body => exiting
      Print("alles ok");

   exception         -- handler in task
      when others => Print("Fehler in body");
   end T;

begin
   Print("fertig");
```

```
exception
   when others => Print("Fehler in task");
end Exception_Demo4;

----------------------------------------------------------------------
13 pohlmann@pavian:exception_demo4
Fehler in body
fertig
```

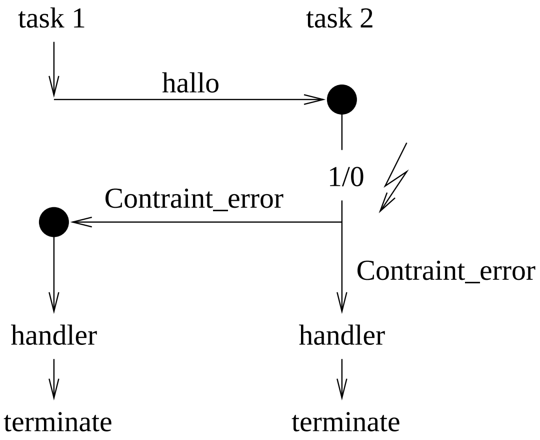
So sollte man's machen! !

Exception tritt auf *während Rendezvous* (d. h. innerhalb der Ausführung des accept-statements).

Fall 1 Accept statement enthält selbst keinen handler.
⇒ Exception wird propagiert zu beiden Beteiligten, genauer: Wieder ausgelöst auf der Ebene des entry-accept *und* des entry-calls.
⇒ Dann unabhängige Behandlung durch die beiden tasks, also im Idealfall: Beide haben eigenen handler.
Beispiel

```
with Stringpack; use Stringpack;
procedure Exception_Demo5 is

   task T1;
   task T2 is
      entry Hallo;
   end T2;

   task body T1 is
   begin
      loop
         T2.Hallo;
      end loop;

   exception
      when Constraint_Error => Print("t1: Fehler in rendezvous");
   end T1;

   task body T2 is
      function F return Natural is begin return 0; end F;
      N: Natural;
   begin
      loop
         accept Hallo do
            N := 1 mod F;     -- 1/0 in Rendezvous
         end Hallo;
      end loop;

   exception
      when Constraint_Error => Print("t2: Fehler in rendezvous");
   end T2;

begin
   null;
end Exception_Demo5;

----------------------------------------------------------------------
```

```
15 pohlmann@pavian:exception_demo5
t2: Fehler in rendezvous
t1: Fehler in rendezvous
```



Fall 2 Accept statement enthält geeigneten handler ⇒ Exception wird lokal behandelt, und alles ist gut!

Folgendes Beispiel erbrachte nicht den gewünschten Effekt, sondern (bei 2 Unix Implementierungen von GNAT) einen Programmabbruch. !

wegen

„Floating Exception“ nix Ada! Vermutlich: System versucht doch noch exception zuzustellen und weiß nicht wo.

6.6.2 Hausaufgabe

Probieren Sie dieses Beispiel und/oder Varianten auf Ihrer Maschine aus; suchen Sie in der Literatur oder (besser) im Netz nach Aufklärung!

```
with Stringpack; use Stringpack;
procedure Exception_Demo6 is

   task T1;
   task T2 is
      entry Hallo;
   end T2;

   task body T1 is
   begin
      for I in 1..3 loop
         T2.Hallo;
      end loop;

   exception
      when others => Print("t1: Fehler in rendezvous");
   end T1;

   task body T2 is
      function F return Natural is begin return 0; end F;
      N: Natural;
   begin
      for I in 1..3 loop
         accept Hallo do
            N := 1 mod F;     -- 1/0 in Rendezvous
```

```
         exception
            when Constraint_Error => Print("t2: Fehler in rendezvous");
         end Hallo;
      end loop;
   end T2;

begin
   null;
exception
   when Constraint_Error => Print("main: Fehler in rendezvous");
end Exception_Demo6;

----------------------------------------------------------------------
37 pohlmann@pavian:exception_demo6
t2: Fehler in rendezvous
Floating exception
```

Anmerkung des Tippers Ich habe das Beispiel auf meinem Rechner(Linux Kernel 2.0.33, gcc version egcs-2.90.23 980102 (egcs-1.0.1 release), GNAT 3.10p) ausprobiert und erhielt das gewünschte Ergebnis:

```
rbla@bravo:exception_demo6
t2: Fehler in rendezvous
t2: Fehler in rendezvous
t2: Fehler in rendezvous
```

6.7 Exkurs: Diskriminanten (= Parameter für Typen)

- sind Typkomponenten mit speziellen Eigenschaften.
- gibt es für zusammengesetzte Typen (außer für arrays!).
- müssen selber von *diskreten* oder *access* Typ sein!

6.7.1 Klassischer Fall: *Record* Typ mit Diskriminante

Beispiel

```
type Matrix_Type is
   array(Positive range <>, Positive range <>) of Float;

type Square_Matrix(size: Positive) is record
   Matrix: Matrix_Type(1..Size, 1..Size);
end record;
```

Also:

- Record Typ mit *zwei* Komponenten: 1. Size, 2. Matrix
- **1. Komponente** Diskriminante
  2. Komponente gewöhnliche record Komponente aber *abgestützt* auf Diskriminante.

Anwendung z. B. so:

```
M1: Square(3);
M2: Square(Size => 3);
M3: Square := (3, (1..3 => (1..3 => 0.0)));
```

Zugriff auf Diskriminante wie sonst auf Komponente, z. B

```
M2.Size
```

Klar: Wert von Diskriminante ist bei Objektvereinbarung nötig und später nicht mehr änderbar. !

Wichtiger Spezialfall:

Variante Records Struktur des Records hängt ab von der Diskriminante.

z.B.

```
type Sex is (Male, Female);

type Person(Which: Sex) is record
   Name: ...;
   case Which is
      when Female =>
         Children: Integer;
      when Male =>
         null;
   end case;
end record;
```

Variante Records sind primitiver Ersatz für Klassen/Vererbung ⇒ Literatur für Näheres. !

6.7.2 Task types mit Diskriminante

Ähnlich: protected objects.
Diskriminanten sind insbesondere zur Spezialisierung in Initialisierung nützlich:

Beispiel

- Initialisierung durch entry

```
task type Hund is
   entry Start(Wie_Oft: Positive);
end Hund;

task body Hund is
   N: Positive;
begin
   accept Start(Wie_Oft: Positive) do
      N := Wie_Oft;
   end Start;

   for I in 1..N loop
      Print("wau wau");
   end loop;
end Hund;
```

- Initialisierung über Diskriminante

```
task type Hund(Wie_Oft: Positive);

task body Hund is
begin
   for I in 1..Wie_Oft loop
```

```
        Print("wau wau");
     end loop;
  end Hund;
```

Unterschiede:

- Entry kann Werte von beliebigem Typ einbringen,
- Entry ist synchronisierend,
- Diskriminante oft bequemer.

Kapitel 7

Rendezvous mit Alternativen

Problem Prozeß, der *entry call* oder *accept* Anweisung ausführt, ist dadurch *festgelegt*: Er *muß* warten, bis Rendezvous zustande kommt; aber z. .B möglich

- daß kein *Partner* mehr *vorhanden* ($\Rightarrow$ Tasking_error),
- daß Rendezvous nur sinnvoll, wenn *sofort* oder *bald* stattfindet,
- daß Prozeß *verschiedene* Rendezvous eingehen könnte.

Also mehr Flexibilität nötig!
Wir beginnen mit entsprechdener Differenzierung beim accept-statement.

7.1 Accept-Alternativen

```
<selective_accept> ::=
   select
     [when <condition> =>]
     <selective_accept_alternative>

  or
     [when <condition> =>]
     <selective_accept_alternative>

  [else
     <statement_sequence>]
   end select;
```

Wobei

```
<selective_accept_alternative> ::=
   <accept_alternative>
 | <delay_alternative>
 | <terminate_alternative>
```

Wir betrachten nun diese drei Möglichkeiten.

7.1.1 accept_alternative

```
<accept_alternative> ::=
   <accept_statement> [<statement_sequence>]
```

Erlaubt warten auf *mehrere* Rendezvous und Akzeptieren der *ersten* Möglichkeit; typisch nötig für *server tasks*, die „ewig“ *verschiedene Dienste* ausüben.

```
task Server is
   entry Service1(...);
   ...
   entry ServiceN(...);
end Server;

task body Server is
   ...
begin
   loop
      select
         accept Service1(...) do
            ...
         end Service1;
         -- internal work 1 --
      or
         ...
      or
         accept ServiceN(...) do
           ...
         end ServiceN;
         -- internal work N --
      end select;
   end loop;
end Server;
```

Wirkung

- Bei jedem Schleifendurchgang 1 Rendezvous,
- wenn *kein* call vorhanden ⇒ Server wartet, bis ein(!) Anruf kommt,
- wenn calls für *mehrere* entries vorhanden ⇒ akzeptiere *einen* (unbestimmt welcher).

7.1.2 Guards

Erlauben, die select-Alternativen durch Bedingungen zu schützen.

```
select
   when -- Dienst möglich -- =>
      accept Service1(...) do
         ...
      end Service1;
      -- interne Arbeit --
or
   ...
end select;
```

Wirkung

- Bedingungen werden ausgewertet bei *Beginn* der Ausführung des select-statements.
- Alternativen *ohne* Bedingungen oder mit *wahren* Bedingungen heißen *offen* (open), die anderen *geschlossen* (closed). **Program_error, wenn alle Alternativen geschlossen.** !
- Offene Alternativen werden behandelt wie beim einfachen select.
- Wie bei guards bei protected objects gilt: Guard darf nicht entry Parameter benutzen, guard sollte nicht globale Variablen benutzen.

7.1.3 terminate Alternative

`<terminate_alternative> ::= terminate;`

Erlaubt *Terminierung* einer „endlosen“ server-task, wenn keine Klienten mehr *da sind.* Z. B.

```
loop
   select
      accept Service1(...) do
         ...
      end Service1;
   or
      ...
   or terminate;
   end select;
end loop;
```

Wirkung Server wird *completed* wenn

- Elterntask ist completed.
- Alle anderen task mit derselben Elterntask sind schon fertig *oder* warten selber in einem select-statement mit terminate-Alternative.

7.1.4 Methodische Bemerkung

Bei Entwurf/Programmierung von *Servern* hat man 2 Möglichkeiten:

1. *protected object* mit service = protected operation oder entry
2. *task* mit Endlosschleife + select accept service

Variante 1 (= *passives* Objekt) ist oft besser (einfaches Programm, effiziente Realisierung), es sei denn der Server muß *zwischen* den Diensten lokal Arbeit leisten (aufräumen, nächsten Dienst vorbereiten, ...), wodurch der Klient nicht verzögert werden sollte.

7.1.5 Beispiel: server task

Vergleiche Lösung Seite 51.

```
----------------------------------------------------------------------
-- Aufgabe: producer-consumer (summation_of_naturals) with bounded buffer
-- Methode: tasks for producer-consumer (simple sum_of_naturals problem)
--          AND task for buffer
----------------------------------------------------------------------

with Stringpack; use Stringpack;
procedure Bounded_Buffer2 is

   task Buffer is                -- server task
      entry Insert(D: Natural);
      entry Take(D: out Natural);
   end Buffer;

   task Producer is
      entry Start(How_Many: Natural);
   end Producer;

   task Consumer is
      entry Start(Break: Natural);
```

```
   end Consumer;

   task body Producer is
      Local_How_Many: Natural;
   begin
      accept Start(How_Many: Natural) do
         Local_How_Many := How_Many;
      end Start;

      Consumer.Start(Local_How_Many +1);
      for I in 1..(Local_How_Many +1) loop
         Buffer.Insert(I);
      end loop;
   end Producer;

task body Consumer is
   Over, Item, Result: Natural;
begin
   accept Start(Break: Natural) do
      Over := Break;
   end Start;

   Result := 0;
   Buffer.Take(Item);
   while Item /= Over loop
      Result := Result +Item;
      Buffer.Take(Item);
   end loop;

   Print("Summe= " & Result);
end Consumer;

task body Buffer is
   Length: constant Natural := 10;
   B: array(0..Length-1) of Natural;
   In_Ptr, Out_Ptr: Natural := 0;
   Count: Natural := 0;
begin
   loop
      select
         when Count < Length =>
            accept Insert(D: Natural) do
               B(In_Ptr) := D;
            end Insert;

            In_Ptr := (In_Ptr +1) mod Length;
            Count := Count +1;
      or
         when Count > 0 =>
            accept Take(D: out Natural) do
               D := B(Out_Ptr);
            end Take;

            Out_Ptr := (Out_Ptr +1) mod Length;
            Count := Count -1;
      or
         terminate;
      end select;
```

```
   end loop;
end Buffer;

N: Natural;
begin
   Print("give natural");
   N := Getint;
   Producer.Start(N);
end Bounded_Buffer2;

----------------------------------------------------------------------
45 pohlmann@mandrill:bounded_buffer2
give natural
100
Summe= 5050
```

7.1.6 Beispiel: Primzahlen durch Siebmethode

In der VL Programmieren 3, Funktionale Programmierung, haben wir Programmierung mit „*unendlichen Listen*“ kennengelernt; für solche unendlichen Listen läßt sich ein (mehr oder weniger vollständiges) Äquivalent schaffen durch *Prozesse*, die auf Aufforderung das *nächste Element* der Liste erzeugen; Realisierung durch:

- Passive Objekte mit internem Zustand + give_next-Operation.
- Aktive Objekte wie Ada task mit give_next-entry.

Hier: Illustration mittels task.

Anwendungsbeispiel: Primzahlerzeugung durch Sieben
Dazu:
Beginne mit Zahlenfolge $s_1 = 2, 3, 4, 5, \ldots$;
Erzeuge s_{i+1} durch Elimination aller Vielfachen des ersten Elements von s_i aus s_i;
Die Kopfelemente dieser Folgen sind die Primzahlen.
Also:

```
prim|
[ 2,| 3,  4,  ... ]  Startfolge s1
[ 3,| 5,  7,  ... ]             s2
[ 5,| 7, 11,  ... ]             s3
...                             ...
```

in *Haskell*

```
crossout :: [Int]->[Int]
crossout (x:xs) = filter (\y -> y `rem` x /= 0) xs

primeNumbers :: [Int]
primeNumbers = map head (iterate crossout [2..])
```

Dieser elegante 4-Zeiler läßt sich mit Ada tasks ungefähr so nachmachen:

task type seq repräsentiert Folge ganzer Zahlen, entweder 2, 3, 4, ... (Start) oder crossout (Vorgängerfolge).

task seq_of_seqs repräsentiert Folge *von* solchen Folgen.

```
----------------------------------------------------------------------
-- Primzahlen durch Siebmethode:
-- erzeuge Folge von Folgen [[2,3,4,5,...],[3,5,7,9],...]
-- durch Elimination aller Fielfachen des 1. Elements der Vorgaengerfolge;
-- die jeweils ersten Elemente bilden Folge der Primzahlen
```

```
----------------------------------------------------------------------

with Stringpack; use Stringpack;
procedure Primes is

   ----------------------------------------------- Typ der Zahlenfolge
   type Seq;  -- incomplete type declaration to allow recursive construction
   type A_Seq is access Seq;

   task type Seq(Pred: A_Seq) is     -- discriminant names predecessor sequence
      entry Head(Item: out Integer); -- gives first element
      entry Next(Item: out Integer); -- gives all elements on iterated calls
   end Seq;

task body Seq is
   First, Elem, Prime: Integer;
begin
   if Pred = null then  -- base case: seq= 2,3,4,5,...
      First := 2; Elem := 2;
      loop
         select
            accept Head(Item: out Integer) do
               Item := First;
            end Head;
         or
            accept Next(Item: out Integer) do
               Item := Elem;
            end Next;

            Elem := Elem +1;
         or
            terminate;
         end select;
      end loop;

   else  -- general case: seq = crossout multiples in predecessor seq
      Pred.Head(Prime); Pred.Next(Elem);
      while (Elem mod Prime = 0) loop
         Pred.Next(Elem);
      end loop;
      First := Elem;

      loop
         select
            accept Head(Item: out Integer) do
               Item := First;
            end Head;
         or
            accept Next(Item: out Integer) do
               Item := Elem;
            end Next;

            Pred.Next(Elem);
            while (Elem mod Prime = 0) loop
               Pred.Next(Elem);
            end loop;
         or
            terminate;
```

```
         end select;
      end loop;
   end if;
end Seq;  --------------------------------------------------------

----------------------------------------------- Folge von Zahlenfolgen
task Seq_Of_Seqs is
   entry Next(Item: out A_Seq);  -- gives all elements on iterated calls
end Seq_Of_Seqs;

task body Seq_Of_Seqs is
   Previous, Elem: A_Seq;  -- previous = null on initialization!
begin
   loop
      Elem := new Seq(Previous); -- create next seq (with a link to previous sequence)
      select
         accept Next(Item: out A_Seq) do
            Item := Elem;
         end Next;
      or
         terminate;
      end select;

      Previous := Elem;
   end loop;
end Seq_Of_Seqs; -------------------------------------------------

Res, From, To: Integer;
Current_Seq: A_Seq;
begin
   Print("give from and to");
   From := Getint;
   To := Getint;
   for I in 1..From-1 loop  -- skip
      Seq_Of_Seqs.Next(Current_Seq);
   end loop;

   for I in From..To loop
      Seq_Of_Seqs.Next(Current_Seq);
      Current_Seq.Head(Res);
      Print(">" & Res);
   end loop;
end Primes;

----------------------------------------------------------------------
25 pohlmann@mandrill:primes
give from and to
1 3
>2
>3
>5
26 pohlmann@madrill:primes
100 100
>541
```

Klar: Lösung nicht praktikabel für große Primzahlen.

7.1.7 Hausaufgabe

1. Die tasks, die die unendlichen Folgen repräsentieren, können nur *einen* Klienten bedienen (entry next = Einmal-Durchlauf). Überlegen Sie sich eine Realisierung, die beliebig viele Klienten bedienen kann (und dazu bereits erzeugte Folgenelemente jeweils speichert).
2. In der Literatur (z. B. Barnes, Burns/Wellings) wird die Primzahl/Sieb-Aufgabe meist anders gelöst, nämlich mit umgekehrter Aufrufstruktur: Zahlenfolge stößt *Nachfolger*-Zahlenfolge an. Betrachten (realisieren) Sie eine solche Lösung und überlegen Sie Vor- und Nachteile!

7.1.8 Delay-Alternative

```
<delay_alternative> ::=
   <delay_statement> [<sequence_of_statements>]

<delay_statement> ::=   delay <expr>;
                      | delay until <expr>;
```

Wird für *timeouts* mit relativer/absoluter Zeitangabe benutzt, d. h. task kann ihre Bereitschaft zu Rendezvous beenden.

Wirkung Task führt delay-Alternative aus, falls keines der Rendezvous des select-statements in der angegebenen Frist/ bis zum angegebenen Zeitpunkt möglich ist.

Typische Anwendung

- Führe default-Handlung aus, wenn nichts anderes gewünscht wurde (z. B. unterbliebene Initialisierung)
- Löse Alarm aus, wenn andere task sich nicht als „lebendig“ erweist (d. h. meldet).

```
----------------------------------------------------------------------
-- Demo: delay alternative
----------------------------------------------------------------------

with Stringpack; use Stringpack;
procedure Watchdog_Demo is

   task type Get_Info;
   type A_Get_Info is access all Get_Info;

   task type Watchdog(For_Whom: A_Get_Info) is
      entry Start(T: Duration);
      entry Ok;
   end Watchdog;

   G: aliased Get_Info;
   W: Watchdog(G'Access);

   task body Get_Info is
      Vs: Varstring;
   begin
      Print("give info");
      W.Start(60.0);
      Vs := Gettext;
      W.Ok;
   end Get_Info;

   task body Watchdog is
      D: Duration;
```

```
   begin
      accept Start(T: Duration) do
         D := T;
      end Start;

      select
         accept Ok;
         Print("got it");
      or
         delay D;
         abort For_Whom.all;
         Print("time expired");
      end select;
   end Watchdog;

begin
   null;
end Watchdog_Demo;

----------------------------------------------------------------------
40 pohlmann@mandrill:watchdog_demo
give info
hallo
got it
41 pohlmann@mandrill:watchdog_demo
give info
time expired
```

Anmerkung des Tippers Ich habe das Beispiel auf meinem Rechner(Linux Kernel 2.0.33, gcc version egcs-2.90.23 980102 (egcs-1.0.1 release), GNAT 3.10p) ausprobiert und erhielt:

```
rbla@bravo:watchdog_demo
give info
hallo
got it
rbla@bravo:watchdog_demo
give info
time expired

raised STORAGE_ERROR
```

Delay-Alternative *nicht verwechseln* mit normalem delay-statement (= *verzögere* task). !

Beispiele

- Verzögerung um 10 Sekunden.

```
...
accept E;
delay 10.0;
...
```

- 10 Sekunden auf Rendezvous warten.

```
...
select
   accept E;
or delay 10.0;
end select;
...
```

- 20 Sekunden auf Rendezvous warten.

```
...
select
   accept E;
or
   delay 20.0;
end select;
...
```

- 10 Sekunden auf Rendezvous warten; falls keines kommt weitere 10 Sekunden verzögern.

```
...
select
   accept E;
or
   delay 10.0;
   delay 10.0;
end select;
...
```

7.1.9 else-Teil des select-statements

Wird ausgeführt, wenn Rendezvous *nicht sofort* möglich ist, also äquivalent:

```
...
select
   accept E;
or
   delay 0.0;
   Statement;
end select;
...
```

```
...
select
   accept E;
else
   Statement;
end select;
...
```

Man vermeide „*polling*“ d. h. ständiges Abfragen (busy waiting). !

```
...
loop
   select
      accept E;
   else
      null;
   end select;
end loop;
...
```

Sinnvolle Anwendungen von „else“ sind selten! !

7.2 Entry-Call mit Alternative

(hier: für Tasks, gilt analog für protected objects)

Anders als bei selective-accept ist bei Aufrufen *nur 1 Alternative* möglich. !

Nämlich:

7.2.1 Timed entry call

```
<timed_entry_call> ::=
   select
      <entry_call_alternative>
   or
      <delay_alternative>
   end select;
```

wobei

```
<entry_call_alternative> ::=
   <entry_call_statement>
      [<statement_sequence>]
```

Also z. B.:

```
loop
   select
      Buffer.Take(X);
      Consume(X);
   or
      delay 5.0;
      raise Alarm;   -- exception
   end select;
end loop;
```

Wirkung Aufrufer wartet für die angegebene Zeit, daß Aufruf angenommen wird, also wieder time-out.

Time-out betrifft nur Beginn des Rendezvous, *nicht* Ende (insbesondere Informationsübertragung). !

7.2.2 Conditional entry call

```
<conditional_entry_call> ::=
   select
      <entry_call_alternative>
   else
      <statement_sequence>
   end select;
```

Wirkung Rendezvous sofort oder gar nicht, enspricht `timed_entry_call` mit `delay 0.0;`

Insbesondere in Schleife (= polling) nur sinnvoll, wenn Aufrufer wirklich was zu tun hat:
`else null;` **⇒ busy waiting, kontraproduktiver Rechenzeitverbrauch.** !

7.2.3 Auswahl unter mehreren entry calls

Ist in Ada *nicht* vorgesehen! Grund: im Allgemeinen großer Aufwand für Implementierung. Man kann aber selber *Ersatzlösungen* bauen:

1. Umkehrung der Aufrufrichtung: call ↔ accept.
   Oft unnatürlich!
2. Einführung von Hilfstask als Zwischenglied, kann Probleme bei 1 lösen.
3. *Wiederversuch* (polling) mit „sinnvollem“ Abstand (ungleich 0, eventuell random).

z. B.:

```
task type Server is
   entry Service;
end Server;

Server_List: array(1..N) of Server;

task Client;

task body Client is
begin
L: loop
       for I in Server_List'Range loop
          select
             Server(I).Service;
             exit L;
          else
             null;
          end select;
       end loop;

       delay Some_Ticks;
end loop L;
end Client;
```

- Ähnlich: Mischung von entry-calls für tasks un protected objects.
- Klar: Ersatz ist nicht dasselbe wie ein ordentliches select:
  Kleines Warteintervall⇒Ineffizient
  Großes Warteintervall⇒Langes Warten möglich (Prüfung *selten*)

7.3 Deadlock revisited

Zur Erinnerung: Wir haben *Verklemmung* (deadlock) in Abschnitt 4 in Zusammenhang mit wechselseitigem Ausschluß kennengelernt.
Dort: Gebrauch von gemeinsamen Variablen, z. B. 2 tasks haben jeweils sperrendes Signal gesetzt.
Hier: Verklemmungsgefahr bei *Rendezvous*.

1.
```
task T1 is                task T2 is
   entry E1;                 entry E2;
end T1;                   end T2;

task body T1 is           task body T2 is
begin                     begin
   T2.E2; (*)           (**) T1.E1;
   accept E1; (**)       (*) accept E2;
end T1;                   end T2;
```

2.
```
...                              ...
task body T1 is                  task body T2 is
begin                            begin
   accept E1;                       accept E2;
   T2.E2;                           T1.E1;
end T1;                          end T2;
```

Unterschied von 1 und 2:

1 Beide tasks unweigerlich verklemmt.

2 Keine Verklemmung, wenn es dritte task gibt, die E1 oder E2 aufruft. Allerdings: Den letzten beißt der Hund ...

Wichtig im *Kampf gegen deadlocks*:

- Große Sorgfalt bei Entwurft/Verfeinerung von Algorithmen.
- Vorher Nachdenken (möglicherweise Beweise führen) anstatt auf Tests verlassen. Deadlock kann selten sein, kann durch anderweitige Systemaktivität verdeckt werden, und grundsätzlich: „ewige“ Inaktivität kann empirisch nicht von „langer“ aber endlicher Wartezeit unterschieden werden.
- Anwendung *methodischer Regeln*, falls für konkreten Fall passend, z. B.: Aufteilung in tasks, die *ausschließlich* entry-*call*-statements enthalten („clients“) und tasks, die *ausschließlich* entry-*accept*-statements enthalten („server“).

Kapitel 8

Beispiele

8.1 Readers-Writers-Problem

Klassisches Problem-"Muster" im Bereich wechselseitiger Ausschluß/Bedingungssynchronisation, mit vielen möglichen Verfeinerungen und auch praktischen Anwendungen.

8.1.1 Gegeben

Ressource wie „Datenbasis" (Tabelle, Datei, ...) und zwei Klassen von Nutzern:

reader Können *konkurrent* (gleichzeitig, überlappend) zugreifen.

writer Benötigen *exklusiven* Besitz der Ressource.

8.1.2 Gesucht

Geeignetes Protokoll.
Offensichtliche Lösung in Ada95:
Ressource = *protected object* mit
`procedure write(...)` ⇒ exklusiv, darf Zustand ändern
`function read(...)` ⇒ nicht exklusiv. Aber schließt gleichzeitiges write aus.

8.1.3 Illustration: Ressource = gemeinsame Variable

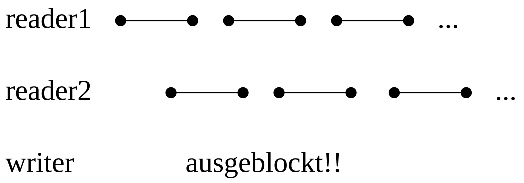
```
----------------------------------------------------------------------
-- Readers-Writers Demo, mit simpler integer-Variablen
--
-- Version 1: protected object, => no preference for writer,
--                                 no potentially blocking ops
----------------------------------------------------------------------

with Stringpack; use Stringpack;
procedure Readers_Writers1 is

protected Variable is
   procedure Write (X : Integer);
   function Read return Integer;
private
   V : Integer := 0; -- na ja
end Variable;

protected body Variable is
   procedure Write (X : Integer) is
      begin V := X; end Write;
   function Read return Integer is
```

```
      begin return V; end Read;
end Variable;

task type Reader;
task body Reader is
      Local_X : Integer;
   begin
      for I in 1..100000 loop
           Local_X := Variable.Read;
      end loop;
      Print("> " & Local_X);
end Reader;

task type Writer;
task body Writer is
      Local_X : Integer := 1;
   begin
      for I in 1..100000 loop
           Variable.Write(Local_X);
           Local_X := Local_X + 1;
      end loop;
end Writer;

Anz_R, Anz_W : Natural;

begin

   Print("give #readers, #writers");
   Anz_R := Getint; Anz_W := Getint;
   declare
       R : array(1..Anz_R) of Reader;
       W : array(1..Anz_W) of Writer;
   begin null; end;

end Readers_Writers1;

----------------------------------------------------------------------
26 pohlmann@mandrill:readers_writers1
give #readers, #writers
3 1
> 97951
> 100000
> 100000
27 pohlmann@mandrill:

43 pohlmann@mandrill:readers_writers1
give #readers, #writers
3 1
> 90700
> 90700
> 90700
44 pohlmann@mandrill:
```

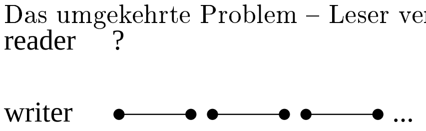
Nichtdeterminismus des Ablaufs und daher Nichtdeterminiertheit des Ergebnisses. Ergebnis = Letzter gelesener Wert bei 100000 Zugriffen. !

8.1.4 Mängel der Lösung

- Ada erlaubt keine „blockierenden Operationen in protected operations, also z. B. I/O.
- *Starvation* (= Aushungern) der Schreiber möglich:



Ergebnis unseres Laufs brauchbar, da 1-Prozessor Implementierung vorliegt und reader wahrscheinlich *nicht* während seiner read-operation *unterbrochen*, bei Prozeßwechsel aber jeder Prozeßtyp gleichermaßen möglich.
Das umgekehrte Problem – Leser verhungern wegen Unfairness – ist auch denkbar:



Erfordert aber „bösartiges“ Verhalten der Prozeßverwaltung und genügend dichte (häufige) Aufrufe von write; praktisch weniger wichtiges Problem und von uns *nicht* behandelt!
Aber jetzt Verbesserung:

- Wenn writer Ressource benutzen will, werden *keine neuen* Lesewünsche mehr zugelassen.
- Protokoll realisiert duch Kontroll-Task, die über Ressourcenbenutzung Buch führt und ähnlich wie Semaphor-Kombination wirkt.

```
----------------------------------------------------------------------
-- Readers-Writers Demo, mit simpler integer-Variablen
--
-- Version 2: control task in package =>  preference for writer,
--                                        potentially blocking ops allowed
----------------------------------------------------------------------

with Stringpack; use Stringpack;
procedure Readers_Writers2 is

   package Variable is
      procedure Write (X : Integer);
      function Read return Integer;
   end Variable;

   package body Variable is

      V : Integer := 0; -- na ja

      task Control is
         entry Start_Read;
         entry Finish_Read;
         entry Start_Write;
         entry Finish_Write;
      end Control;

      task body Control is
         Readers, Writers : Natural := 0;
      begin
         loop
            select
               when Writers = 0  and Start_Write'Count = 0 => -- no writer inside or waiting
```

```
                accept Start_Read;
                Readers := Readers + 1;
          or
             accept Finish_Read;
             Readers := Readers - 1;
          or
             when Readers = 0 and Writers = 0  =>  -- nobody inside
                accept Start_Write;
                Writers := 1;
          or
             accept Finish_Write;
             Writers := 0;
          or
             terminate;
          end select;
       end loop;
    end Control;

    procedure Write (X : Integer) is
    begin
       Control.Start_Write;
       V := X;
       Control.Finish_Write;
    end Write;

    function Read return Integer is
       X : Integer;
    begin
       Control.Start_Read;
       X := V;
       Control.Finish_Read;
       return V;
    end Read;

 end Variable;

 task type Reader;
 task body Reader is
    Local_X : Integer;
 begin
    for I in 1..100000 loop
       Local_X := Variable.Read;
    end loop;
    Print("> " & Local_X);
 end Reader;

 task type Writer;
 task body Writer is
    Local_X : Integer := 1;
 begin
    for I in 1..100000 loop
       Variable.Write(Local_X);
       Local_X := Local_X + 1;
    end loop;
 end Writer;

 Anz_R, Anz_W : Natural;
```

```
begin

   Print("give #readers, #writers");
   Anz_R := Getint; Anz_W := Getint;
   declare
      R : array(1..Anz_R) of Reader;
      W : array(1..Anz_W) of Writer;
   begin null; end;

end Readers_Writers2;

----------------------------------------------------
35 pohlmann@mandrill:readers_writers2
give #readers, #writers
3 1
> 100000
> 100000
> 100000
36 pohlmann@mandrill:
```

Beachte: !

- Zugangskontrolle durch task, aber eigentliche Nutzung der Ressource außerhalb.
- Alles verpackt in package mit read/write-ops als Schnittstelle, aber Protokollimplementierung vor Klient geschützt.
- Guard vor accept-read berücksichtigt Anzahl der bereits wartenden Schreiber vermöge des *Attributs* „start_write'count“ = Zahl der aktuell wartenden Aufrufer dieses entry.

Zum Vergleich: Die folgende Programmversion unterläßt die Bedingung „`when start_write'count = 0`“ und läßt im Beispiel-Lauf die Leser *öfter* zum Zuge kommen:

```
----------------------------------------------------------------------
-- Readers-Writers Demo, mit simpler integer-Variablen
--
-- Version 2: control task in package => *no* preference for writer,
--                                       potentially blocking ops allowed
----------------------------------------------------------------------

with Stringpack; use Stringpack;
procedure Readers_Writers2a is

   package Variable is
      procedure Write (X : Integer);
      function Read return Integer;
   end Variable;

   package body Variable is

      V : Integer := 0; -- na ja

      task Control is
         entry Start_Read;
         entry Finish_Read;
         entry Start_Write;
         entry Finish_Write;
      end Control;
```

```
   task body Control is
      Readers, Writers : Natural := 0;
   begin
      loop
         select
            when Writers = 0   => -- no writer inside, no preference for writers
               accept Start_Read;
               Readers := Readers + 1;
         or
            accept Finish_Read;
            Readers := Readers - 1;
         or
            when Readers = 0 and Writers = 0  =>  -- nobody inside
               accept Start_Write;
               Writers := 1;
         or
            accept Finish_Write;
            Writers := 0;
         or
            terminate;
         end select;
      end loop;
   end Control;

   procedure Write (X : Integer) is
   begin
      Control.Start_Write;
      V := X;
      Control.Finish_Write;
   end Write;

   function Read return Integer is
      X : Integer;
   begin
      Control.Start_Read;
      X := V;
      Control.Finish_Read;
      return V;
   end Read;

end Variable;

task type Reader;
task body Reader is
   Local_X : Integer;
begin
   for I in 1..100000 loop
      Local_X := Variable.Read;
   end loop;
   Print("> " & Local_X);
end Reader;

task type Writer;
task body Writer is
   Local_X : Integer := 1;
begin
   for I in 1..100000 loop
```

```
         Variable.Write(Local_X);
         Local_X := Local_X + 1;
      end loop;
   end Writer;

   Anz_R, Anz_W : Natural;

begin

   Print("give #readers, #writers");
   Anz_R := Getint; Anz_W := Getint;
   declare
      R : array(1..Anz_R) of Reader;
      W : array(1..Anz_W) of Writer;
   begin null; end;

end Readers_Writers2a;

-----------------------------------------------------
37 pohlmann@mandrill:readers_writers2a
give #readers, #writers
3 1
> 3751    -+
> 3751     | Also Leser viel schneller fertig als Schreiber!
> 4274    -+
38 pohlmann@mandrill:
```

Zur *Korrektheit* der Lösung (= „readers_writers2“):

Safety (= „writer hat exklusiven Gebrauch“) Ist klar, wegen guards und Zählern.

No deadlock Klar, wenn Gebrauch der Ressource nur endlich dauert.

No starvation Auch klar, wenn Ada-Implementierung offene selective-accept-Alternativen fair bedient.

Angegebene Lösung hat *erheblichen Zeitbedarf*, weil pro Zugriff (read/write) *2 Rendezvous* nötig. !

Kleine Verbesserungsmöglichkeit: Eigentliche write-Tätigkeit in die task integrieren und den Zähler für writers sowie finish_write einsparen:

```
select
   ...
or
   when Readers = 0 =>
      accept Start_Write(X: Integer)
        do V := X; end Start_Write;
or
   terminate;
end select;
```

Also ausnutzen, daß task selber schon den wechselseitigen Ausschluß herstellt.

8.1.5 Alternative

Statt Kontroll-Task ein *protected-object* mit analoger Struktur nehmen:

```
-----------------------------------------------------------------------
-- Readers-Writers Demo, mit simpler integer-Variablen
--
```

```
-- Version 3: protected object in package =>  preference for writer,
--                                             potentially blocking ops allowed
----------------------------------------------------------------------

with Stringpack; use Stringpack;
procedure Readers_Writers3 is

   package Variable is
      procedure Write (X : Integer);
      function Read return Integer;
   end Variable;

   package body Variable is

      V : Integer := 0; -- na ja

      protected Control is
         entry Start_Read;
         procedure Finish_Read;
         entry Start_Write;
         procedure Finish_Write;
      private
         Readers, Writers : Natural := 0;
      end Control;

      protected body Control is

                                                     -- no writer waiting or working
         entry Start_Read when Writers = 0 and Start_Write'Count = 0 is
         begin Readers := Readers + 1; end Start_Read;

         procedure Finish_Read is
         begin Readers := Readers - 1; end Finish_Read;

         entry Start_Write when Readers = 0 and Writers = 0   is   -- nobody inside
         begin Writers := 1; end Start_Write;

         procedure Finish_Write is
         begin Writers := 0; end Finish_Write;

      end Control;

      procedure Write (X : Integer) is
      begin
         Control.Start_Write;
         V := X;
         Control.Finish_Write;
      end Write;

      function Read return Integer is
         X : Integer;
      begin
         Control.Start_Read;
         X := V;
         Control.Finish_Read;
         return V;
      end Read;
```

```
   end Variable;

   task type Reader;
   task body Reader is
      Local_X : Integer;
   begin
      for I in 1..100000 loop
         Local_X := Variable.Read;
      end loop;
      Print("> " & Local_X);
   end Reader;

   task type Writer;
   task body Writer is
      Local_X : Integer := 1;
   begin
      for I in 1..100000 loop
         Variable.Write(Local_X);
         Local_X := Local_X + 1;
      end loop;
   end Writer;

   Anz_R, Anz_W : Natural;

begin

   Print("give #readers, #writers");
   Anz_R := Getint; Anz_W := Getint;
   declare
      R : array(1..Anz_R) of Reader;
      W : array(1..Anz_W) of Writer;
   begin null; end;

end Readers_Writers3;

----------------------------------------------------
36 pohlmann@mandrill:readers_writers3
give #readers, #writers
3 1
> 71215
> 89096
> 100000
44 pohlmann@mandrill:readers_writers3
give #readers, #writers
3 1
> 44358
> 45480
> 52115
```

8.1.6 Unterschiede

(bei wenigen Testläufen) Variante mit protected-objects

- ist *schneller*: Einige Sekunden statt Minuten.
- streut stärker im Resultat, mit Tendenz zur Bevorzugung von Lesern!

Korrektheit (Schreiber verhungert nicht) bleibt natürlich erhalten, trotzdem möglich, daß „endliche" Bevorzugung von Lesern:

- Wegen anderer task-Verwaltung, also Frage von Implementierung, Scheduling, ..., insbesondere Prozeßwechsel bei Rendezvous.
- Beachte auch Ada-Semantik: Bei Ende von protected procedure/entry werden guards neu ausgewertet *und* dann ausführbare entries *sofort* (also nicht in Konkurrenz zu *neuen* Ankünften) ausgeführt.

⇒ Zur Detail-Gestaltung von Reihenfolgen u. ä. gibt es Hilfsmittel im „Realtime Annex“.

8.2 Dining Philosophers

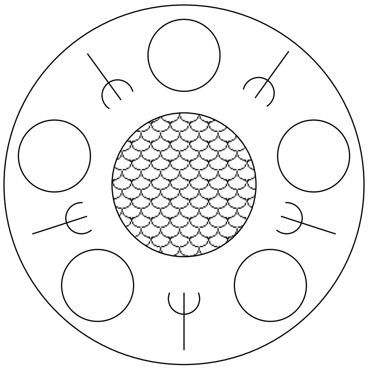
Traditionsreiches akademisches Problem (Dijkstra ’71), oft benutzt, um Nützlichkeit von Sprachkonstrukten und Programmiertechniken zu illustrieren.

Praktischer Hintergrund: Prozesse benötigen *mehrere* exklusive Betriebsmittel gleichzeitig – dann kann ungeschickte Zuteilung zu *Verklemmung* führen, z. B. 2 Köche in 1 Küche, 1 Topf, 1 Löffel, jeder braucht beides. Koch 1 ergreift Löffel, Koch 2 ergreift Topf ⇒ Deadlock.

Gebräuchliche *Vermeidungsstrategie*: Lege fest, daß Prozesse die Betriebsmittel in *derselben Reihenfolge* belegen. Z. B. 1. Löffel, 2. Topf ⇒ Wenn Koch 1 Löffel hat, muß Koch 2 warten, bis Koch 1 fertig.
„Dining philosophers“ Problem ist so konstruiert, daß diese Lösung nicht möglich.

8.2.1 Gegeben

- Gemeinschaft von (z. B.) 5 Philosophen, die ihr Leben mit *Denken* und *Essen* zubringen.
- Speisesaal mit Tisch mit 5 Tellern, 5 Gabeln und Spaghettihaufen (unerschöpflich).



- Regel: Jeder Philisoph hat seinen Platz (Teller), benötigt aber *beide* danebenliegenden Gabeln (wobei er eine Gabel nicht mehr freigibt, bis er fertig gegessen hat)!

8.2.2 Gesucht

Protokoll, das alle Philosophen vor dem Verhungern bewahrt.
(Fragen der Hygiene, des Anstands usw. spielen keine Rolle; es ist aber zugesichert, daß kein Philosoph endlos ißt oder über seinem Teller stirbt! Anmerkung des Tippers: Auch der Diebstahl einer Gabel sei ausgeschlossen!)

8.2.3 Offensichtlich nötig

Stelle exklusiven Gebrauch der Gabeln sicher, z. B. Gabel = protected-object mit

```
entry Pick_Up when Frei is
begin Free := False; end Pick_Up;

protected Put_Down is
begin Free := True; end Put_Down;
```

= binäres Semaphor.

Klar Safety (= keine 2 Philosophen haben dieselbe Gabel).

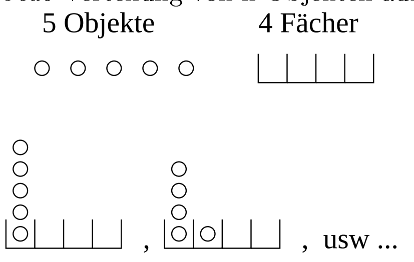
Aber *Verklemmung* möglich: Jeder Philosoph ergreift linke Gabel und findet dann rechte Gabel benutzt.

Lösung Erlaube *höchstens 4* Philosophen im Speisesaal!

⇒ liveness, denn

8.2.4 Schubfachprinzip (=pigeonhole principle)

Jede Verteilung von n Objekten auf n-1 Fächer muß in mindestens 1 Fach mindestens 2 Objekte unterbringen:



Identifiziere: Objekt = Gabel, Fach = Philosoph und beschränke Verteilungsmöglichkeit auf die für unser Problem legalen Fälle (linke, rechte Gabel pro Philosoph).

Das folgende Programm führt zur Beschränkung der im Speisesaal anwesenden Philosophen ein weiteres protected-object ein (task wäre auch möglich); zur Dokumentation wird bei jedem Ereignis (= Gabel ergreifen oder niederlegen) der aktuelle Stand protokolliert und ausgegeben:

```
----------------------------------------------------------------------
-- Dining Philosophers
--
-- synchronisation by protected objects room and fork (1..#phils):
--               room (always less than #phils inside) => deadlock prevention,
--                                                fork => mutual exclusion;
-- for demo, phils call trace task whenever having seized/released a fork
----------------------------------------------------------------------

with Stringpack; use Stringpack;
procedure Philosophers is

   How_Many : constant Natural := 5;
   subtype Phil_Range is Integer range 0..How_Many-1;
   How_Often : constant Natural := 5;         -- Zahl der Speisungen/Philosoph

   task Trace is --------------------------------------------------------
      entry Up(Where : Phil_Range);
      entry Down(Where : Phil_Range);
   end Trace;

   task body Trace is
      Busy : array(Phil_Range) of Boolean := (others => False);
   begin
      loop
         Outbuffer := Varnull;
         select
            accept Up(Where : Phil_Range)
            do Busy(Where) := True; end Up;
            for J in Phil_Range loop
               if Busy(J) then Outbuffer := Outbuffer & " X";
               else Outbuffer := Outbuffer & " +"; end if;
            end loop;
            Print;
```

```
      or
         accept Down(Where : Phil_Range)
      do Busy(Where) := False; end Down;
         for J in Phil_Range loop
            if Busy(J) then Outbuffer := Outbuffer & " X";
            else Outbuffer := Outbuffer & " -"; end if;
         end loop;
         Print;
      or
         terminate;
      end select;
   end loop;
end Trace;  -----------------------------------------------------------



protected Room is  ----------------------------------------------------
   entry Enter;
   procedure Leave;
private
   Present : Natural := 0;
end Room;

protected body Room is
   entry Enter when Present < How_Many -1 is
   begin Present := Present +1; end Enter;
   procedure Leave is
   begin Present := Present -1; end Leave;
end Room;  ------------------------------------------------------------

protected type Forks is  ----------------------------------------------
   entry Pick_Up;
   procedure Put_Down;
private
   Free : Boolean := True;
end Forks;

protected body Forks is
   entry Pick_Up when Free is
   begin Free := False; end Pick_Up;
   procedure Put_Down is
   begin Free := True; end Put_Down;
end Forks;

Fork : array(Phil_Range) of Forks; ------------------------------------

task type Philosophers (Me : Phil_Range);   --------------------------

task body Philosophers is
begin
   for I in 1..How_Often loop
      delay 0.0001; -- think
      Room.Enter;
      Fork(Me).Pick_Up;  --left fork
      Trace.Up(Me);
      Fork((Me + 1) mod How_Many).Pick_Up; --right fork
      Trace.Up((Me + 1) mod How_Many);
```

```
         delay 0.0001; -- eat
         Fork(Me).Put_Down;  --left fork
         Trace.Down(Me);
         Fork((Me + 1) mod How_Many).Put_Down; --right fork
         Trace.Down((Me + 1) mod How_Many);
         Room.Leave;
      end loop;
   end Philosophers;

   type A_Philosophers is access Philosophers;
   Philosopher : array(Phil_Range) of A_Philosophers;  ------------------

begin  -----------------------------------------------------------main

   for I in Phil_Range loop Philosopher(I) := new Philosophers(I); end loop;

end Philosophers;  ---------------------------------------------------

----------------------------------------------------------------------
Legende: Spalte = fork-nr, 'X' = belegt, '+' und '-' = frei,
                wobei ' -' =Freigabeereignis, '+' = Belegungsereignis.

33 pohlmann@pavian:philosophers
 Spalte 1       Spalte 2       Spalte 3       Spalte 4
 X + + + +      - - X X -      - X - - -      X - - X -
 X X + + +      + X X X +      - - - - -      - - - X -
 - X - - -      - X - X -      + X + + +      + + + X X
 - - - - -      - X - - -      + X X + +      - - - - X
 + X + + +      + X X + +      - - X - -      - - - - -
 + X X + +      X X X + +      - - - - -      + + + + X
 X X X + +      X - X - -      + + X + +      X + + + X
 X - X - -      X - - - -      + + X X +      X + + X X
 X - - - -      X + X + +      - - - X -      X - - X -
 X + X + +      X + X X +      - - - - -      - - - X -
 X + X X +      X X X X +      + + + X +      + + + X X
 X X X X +      X X - X -      + + + X X      - - - - X
 - X X X -      X X - - -      - - - - X      - - - - -
 - - X X -      - X - - -      - - - - -      + + + + X
 - - - X -      - - - - -      + + + + X      X + + + X
 - - - - -      + X + + +      X + + + X      X + + X X
 + X + + +      + X X + +      X + + X X      X - - X -
 + X X + +      X X X + +      X - - X -      - - - X -
 X X X + +      X - X - -      - - - X -      + + + X X
 X - X - -      X - - - -      + + + X X      - - - - X
 X - - - -      X + X + +      - - - - X      - - - - -
 X X + + +      X + X X +      - - - - -      + + + + X
 X X X + +      X X X X +      + + + + X      X + + + X
 X X X X +      X X - X -      X + + + X      X - - - -
 - X X X -      X X - - -      X + + X X      - - - - -
weiter in      weiter in      weiter in
Spalte 2       Spalte 3       Spalte 4
34 pohlmann@pavian:
```

8.3 Asynchrone Iteration

Zur Lösung eines *Fixpunktproblems* der Form

$$\begin{aligned} x_1 &= f_1(x_1, \ldots, x_n) \\ &\vdots \\ x_n &= f_n(x_1, \ldots, x_n) \end{aligned}$$

kann man eine Iteration wie z. B.

$$\begin{aligned} x_1^{s+1} &= f_1(x_1^s, \ldots, x_n^s) \quad (Jakobi) \\ &\vdots \\ x_n^{s+1} &= f_n(x_1^s, \ldots, x_n^s) \end{aligned}$$

oder auch

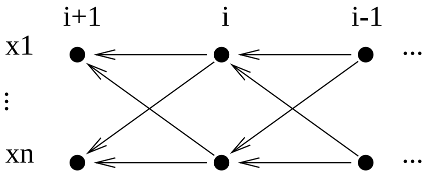
$$\begin{aligned} x_1^{s+1} &= f_1(x_1^s, \ldots, x_n^s) \qquad (Gauss-Seidl) \\ x_2^{s+1} &= f_2(x_1^{s+1}, x_2^s, \ldots, x_n^s) \\ &\vdots \\ x_n^{s+1} &= f_n(x_1^{s+1}, \ldots, x_{n-1}^{s+1}, x_n^s) \end{aligned}$$

verwenden.

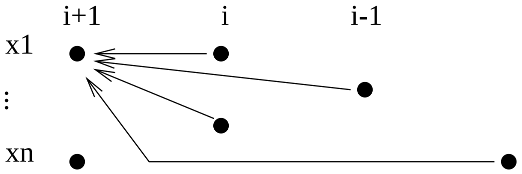
Solche Iterationen legen fest, welche *Vor-Versionen* von Werten zur Berechnung der *nächsten* Version x_i^{s+1} zu benutzen sind und heißen deshalb *synchron*.

Zur *Parallelisierung* – z. B. für jede Komponente x_i eigenen Prozeß – kann man sich andere sinnvolle Reihenfolgen ausdenken, *oder* die Reihenfolge überhaupt *freigeben* = *asynchrone* oder *chaotische Iteration*.

8.3.1 Standard Iteration



8.3.2 Chaotische Iteration



Bezüge so, wie sie sich *faktisch* ergeben, also durch relative Geschwindigkeit der beteiligten Prozesse bzw. Verfügbarkeit von Werten!! („asynchron“ = ohne festen zeitlichen Bezug, insbesondere Reihenfolge).

Wir betrachten ein

8.3.3 Beispiel

vgl. Barnes 95, p. 443 ff.

Gegeben
Partielle Differentialgleichung wie
$\frac{\partial^2 P(x,y)}{\partial x^2} + \frac{\partial^2 P(x,y)}{\partial y^2} = F(x,y)$
auf Gebiet mit gegebenen Randwerten.

Methode zu numerischen Lösung.
Diskretisierung durch Erfüllung von *Gitterpunkten* und Übergang zu *Differenzengleichung*:
P(i-1, j) - P(i, j) + P(i+1, j) - P(i, j) + P(i, j-1) - P(i, j) + P(i, j+1) - P(i, j) = F(i, j)
$\Longleftrightarrow$
P(i, j) = $\frac{1}{4}$[P(i-1, j) + P(i+1, j) + P(i, j-1) + P(i, j+1) - F(i, j)]

also spezielles lineares Gleichungssystem.

Lösung durch Iteration, hier: Ordne jedem Punkt (i, j) eine *task* zu und rechne *asynchron* unter Verwendung von *globalen* (*gemeinsamen*) Variablen.

```
---------------------------------------------------------------------
-- Aufgabe: partielle Differentialgleichung wie zB Laplace,
--                diskretisiert ueber Gitter: 4*P(i,j) =   P(i-1,j) + P(i+1,j)
--                                                       + P(i,j-1) + P(i,j+1)
--                                                       - F(i,j)
-- Methode: tasks fuer Gitterpunkte, asynchrone Iteration, shared variables
--                Hauptprogramm ueberwacht Konvergenz
-- Quelle:    Barnes95,p.443...
--                                   !!! problematic termination method !!!!
---------------------------------------------------------------------

with Stringpack; use Stringpack;
procedure Grid is
   N: constant := 5;
   subtype Full_Grid is Integer range 0 .. N;  -- grid with border
   subtype Grid is Full_Grid range 1 .. N-1;   -- grid without border
   type Real is digits 7;  -- declare our own type
   type Matrix is array(Integer range <>, Integer range <>) of Real;
   pragma Atomic_Components(Matrix);  -- shared!
   P: Matrix(Full_Grid, Full_Grid)  := (others=>(others=>0.0));
   Delta_P: Matrix(Grid, Grid) := (others=>(others=>1.0));
   Tolerance: constant Real := 0.0001;
   Error_Limit: constant Real := Tolerance * Real(N-1)**2;
   Converged: Boolean := False;
   pragma Atomic(Converged);  -- shared!
   Error_Sum: Real;


   function F(I, J: Grid) return Real is   -- keep it simple
   begin return 0.0; end F;

   task type Iterator is
      entry Start(I, J: in Grid);
   end;

   Process: array (Grid, Grid) of Iterator;

   task body Iterator is
      I, J: Grid;
      New_P: Real;
   begin

      accept Start(I, J: in Grid) do
         Iterator.I := Start.I;
         Iterator.J := Start.J;
      end Start;

      loop
         New_P := 0.25 * (P(I-1, J) + P(I+1, J) + P(I, J-1)
                          + P(I, J+1) -  F(I, J));
         Delta_P(I, J) := New_P - P(I, J);
         P(I, J) := New_P;
         exit when Converged;
      end loop;
```

```
   end Iterator;

begin            -- main subprogram, Iterator tasks now active

   for I in Full_Grid loop  --init P;
      P(I,0) := 1.0;
   end loop;

   for I in Grid loop
      for J in Grid loop
         Process(I, J).Start(I, J);  -- tell them who they are
      end loop;
   end loop;

   loop  -- Konvergenzprüfung
      Error_Sum := 0.0;
      for I in Grid loop
         for J in Grid loop
            Error_Sum := Error_Sum + Delta_P(I, J)**2;
         end loop;
      end loop;

      Converged := Error_Sum < Error_Limit;  -- problematic method !!!!!!!!!!!!
      exit when Converged;
   end loop;

   for I in Full_Grid loop        -- output results
      Outbuffer := Varnull;
      for J in Full_Grid loop
         Outbuffer := Outbuffer & Cvfs(Float(P(I,J)),1,3) & " ";
      end loop;
      Print;
   end loop;


end Grid;

-----------------------------------------------------------------------
20 pohlmann@pavian:grid
1.000 0.000 0.000 0.000 0.000 0.000
1.000 0.250 0.063 0.016 0.004 0.000
1.000 0.313 0.094 0.027 0.008 0.000
1.000 0.328 0.105 0.033 0.010 0.000
1.000 0.332 0.109 0.036 0.011 0.000
1.000 0.000 0.000 0.000 0.000 0.000
21 pohlmann@pavian:grid


-----------------------------------------------------------------------
-- Aufgabe: partielle Differentialgleichung wie zB Laplace,
--                diskretisiert ueber Gitter: 4*P(i,j) =   P(i-1,j) + P(i+1,j)
--                                                       + P(i,j-1) + P(i,j+1)
--                                                       - F(i,j)
-- Methode nur zum Vergleich: Standard Iteration(Gauss Seidel Typ)
-----------------------------------------------------------------------

with Stringpack; use Stringpack;
procedure Gridsynchro is
```

```
   N: constant := 5;
   subtype Full_Grid is Integer range 0 .. N;
   subtype Grid is Full_Grid range 1 .. N-1;   -- without border
   type Real is digits 7;  -- declare our own type
   type Matrix is array(Integer range <>, Integer range <>) of Real;
   P: Matrix(Full_Grid, Full_Grid)  := (others=>(others=>0.0));
   Delta_P: Matrix(Grid, Grid) := (others=>(others=>1.0));
   Tolerance: constant Real := 0.0001;
   Error_Limit: constant Real := Tolerance * Real(N-1)**2;
   Converged: Boolean := False;
   Error_Sum: Real;
   New_P: Real;


   function F(I, J: Grid) return Real is   -- keep it simple
   begin return 0.0; end F;


begin            -- main subprogram

   for I in Full_Grid loop  --init P;
      P(I,0) := 1.0;
   end loop;

   loop  -- siehe folgender Text
      Error_Sum := 0.0;
      for I in Grid loop
         for J in Grid loop
            New_P := 0.25 * (P(I-1, J) + P(I+1, J) + P(I, J-1)
                             + P(I, J+1) -  F(I, J));
            Delta_P(I, J) := New_P - P(I, J);
            P(I, J) := New_P;
            Error_Sum := Error_Sum + Delta_P(I, J)**2;
         end loop;
      end loop;
      Converged := Error_Sum < Error_Limit;
      exit when Converged;
   end loop;

   for I in Full_Grid loop        -- output results
      Outbuffer := Varnull;
      for J in Full_Grid loop
         Outbuffer := Outbuffer & Cvfs(Float(P(I,J)),1,3) & " ";
      end loop;
      Print;
   end loop;


end Gridsynchro;

----------------------------------------------------------------------
45 pohlmann@pavian:gridsynchro
1.000 0.000 0.000 0.000 0.000 0.000
1.000 0.438 0.203 0.093 0.037 0.000
1.000 0.573 0.302 0.148 0.061 0.000
1.000 0.577 0.307 0.153 0.063 0.000
1.000 0.446 0.212 0.101 0.041 0.000
1.000 0.000 0.000 0.000 0.000 0.000
```

Für alle Gitterpunkte gilt:
$P^{t+1}(i,j) = \frac{1}{4}[P^{t+1}(i-1,j) + P^t(i+1,j) + P^{t+1}(i,j-1) + P^t(i,j+1) - F(i,j)]$

Ergebnisse sehen anders aus als bei asynchronem Programm!
Möglich:

- Konvergenzkriterium zu lax, na ja.
- *Terminierungsprotokoll* in asynchronem Fall *ungeeignet*! Klar: Wenn process(i, j) *zweimal* mit *demselben* Argumenten rechnet, dann ist Delta_P(i, j) = 0 !

  Aber:

  - Generell möglich, daß dies eintritt.
  - Bei 1-Prozessor Implementierung (interleaving) sogar wahrscheinlich.

  Also z. B. Reihenfolge der Ausführung:

```
process(1,1);
process(1,1);
process(1,2);
process(1,2);
...
process(n-1,n-1);
process(n-1,n-1);
main;  -- findet System konvergiert obwohl process(1,1) nie von
          process(1,2)... process(n-1,n-1) beeinflußt wurde.
```

Offenbare Abhilfe
Überprüfe im Falle von Konvergenz (im obigen Sinne), ob Ergebnis (innerhalb der gewünschten Genauigkeit) Gleichung löst, d. h. wende jeden Iterator noch einmal an!

```
----------------------------------------------------------------------
-- Aufgabe: partielle Differentialgleichung wie zB Laplace,
--                diskretisiert ueber Gitter: 4*P(i,j) =   P(i-1,j) + P(i+1,j)
--                                                       + P(i,j-1) + P(i,j+1)
--                                                       - F(i,j)
-- Methode: tasks fuer Gitterpunkte, asynchrone Iteration, shared variables
--                Hauptprogramm ueberwacht Konvergenz
-- Modifikation von Barnes95,p.443...:
--    Terminierung erst wenn Fehlerschranke simultan unterschritten
--
----------------------------------------------------------------------

with Stringpack; use Stringpack;
procedure Grid2 is
   N: constant := 5;
   subtype Full_Grid is Integer range 0 .. N;
   subtype Grid is Full_Grid range 1 .. N-1;   -- without border
   type Real is digits 7;  -- declare our own type
   type Matrix is array(Integer range <>, Integer range <>) of Real;
   pragma Atomic_Components(Matrix);
   P: Matrix(Full_Grid, Full_Grid)  := (others=>(others=>0.0));
   Delta_P: Matrix(Grid, Grid) := (others=>(others=>1.0));
   Tolerance: constant Real := 0.0001;
   Error_Limit: constant Real := Tolerance * Real(N-1)**2;
   Converged, NoChange: Boolean := False;
   pragma Atomic(Converged); pragma Atomic(NoChange);
```

```
   Error_Sum: Real;

   I : Positive := 1;  -- for counting interim output


   function F(I, J: Grid) return Real is   -- keep it simple
   begin return 0.0; end F;

   task type Iterator is
      entry Start(I, J: in Grid);
      entry Begin_Verify_Result;
      entry End_Verify_Result;
   end;

      Process: array (Grid, Grid) of Iterator;

      task body Iterator is
         I, J: Grid;
         New_P: Real;
      begin

         accept Start(I, J: in Grid) do
            Iterator.I := Start.I;
            Iterator.J := Start.J;
         end Start;

         loop
            New_P := 0.25 * (P(I-1, J) + P(I+1, J) + P(I, J-1)
                             + P(I, J+1) -  F(I, J));
            Delta_P(I, J) := New_P - P(I, J);
            P(I, J) := New_P;
            if Converged
            then accept Begin_Verify_Result  do  -- do it again!
               New_P := 0.25 * (P(I-1, J) + P(I+1, J) + P(I, J-1)
                                + P(I, J+1) -  F(I, J));
               Delta_P(I, J) := New_P - P(I, J);
               P(I, J) := New_P;
            end Begin_Verify_Result;
            accept End_Verify_Result;  -- synchro
            exit when NoChange;
            end if;
         end loop;

      end Iterator;

begin            -- main subprogram, Iterator tasks now active

   for I in Full_Grid loop  --init P;
      P(I,0) := 1.0;
   end loop;

   for I in Grid loop
      for J in Grid loop
         Process(I, J).Start(I, J);  -- tell them who they are
      end loop;
   end loop;

   loop
```

```
      Error_Sum := 0.0;
      for I in Grid loop
         for J in Grid loop
            Error_Sum := Error_Sum + Delta_P(I, J)**2;
         end loop;
      end loop;
      Converged := Error_Sum < Error_Limit;
      if Converged
      then                   -- make sure that error bounds hold simultaneously
         Error_Sum := 0.0;
         for I in Grid loop
            for J in Grid loop
               Process(I,J).Begin_Verify_Result;
               Error_Sum := Error_Sum + Delta_P(I, J)**2;
            end loop;
         end loop;
         Converged := False;
         NoChange := Error_Sum < Error_Limit;
         for I in Grid loop
            for J in Grid loop
               Process(I,J).End_Verify_Result;  -- synchro
            end loop;
         end loop;
      end if;
      if NoChange
      then exit;
      else
         if I < 5 then
            I := I + 1;
            Print("--------------------------------------------------");
            for I in Full_Grid loop        -- output state
               Outbuffer := Varnull;
               for J in Full_Grid loop
                  Outbuffer := Outbuffer & Cvfs(Float(P(I,J)),1,3) & " ";
               end loop;
               Print;
            end loop;
         end if;
      end if;
   end loop;

   Print("results: ----------------------------------------------");
   for I in Full_Grid loop        -- output results
      Outbuffer := Varnull;
      for J in Full_Grid loop
         Outbuffer := Outbuffer & Cvfs(Float(P(I,J)),1,3) & " ";
      end loop;
      Print;
   end loop;


end Grid2;

----------------------------------------------------------------------
rbla@bravo:grid2
--------------------------------------------------
1.000 0.000 0.000 0.000 0.000 0.000
1.000 0.429 0.190 0.083 0.032 0.000
```

```
1.000 0.527 0.255 0.114 0.043 0.000
1.000 0.518 0.248 0.083 0.032 0.000
1.000 0.414 0.166 0.062 0.000 0.000
1.000 0.000 0.000 0.000 0.000 0.000
--------------------------------------------------
1.000 0.000 0.000 0.000 0.000 0.000
1.000 0.429 0.190 0.083 0.032 0.000
1.000 0.527 0.255 0.114 0.043 0.000
1.000 0.518 0.248 0.083 0.032 0.000
1.000 0.414 0.166 0.062 0.000 0.000
1.000 0.000 0.000 0.000 0.000 0.000
--------------------------------------------------
1.000 0.000 0.000 0.000 0.000 0.000
1.000 0.429 0.190 0.083 0.032 0.000
1.000 0.527 0.255 0.114 0.043 0.000
1.000 0.518 0.248 0.083 0.032 0.000
1.000 0.414 0.166 0.062 0.000 0.000
1.000 0.000 0.000 0.000 0.000 0.000
--------------------------------------------------
1.000 0.000 0.000 0.000 0.000 0.000
1.000 0.429 0.190 0.083 0.032 0.000
1.000 0.527 0.255 0.114 0.043 0.000
1.000 0.518 0.248 0.083 0.032 0.000
1.000 0.414 0.166 0.062 0.000 0.000
1.000 0.000 0.000 0.000 0.000 0.000
results: --------------------------------------------
1.000 0.000 0.000 0.000 0.000 0.000
1.000 0.446 0.212 0.099 0.040 0.000
1.000 0.582 0.311 0.154 0.064 0.000
1.000 0.583 0.313 0.157 0.065 0.000
1.000 0.448 0.215 0.103 0.042 0.000
1.000 0.000 0.000 0.000 0.000 0.000
```

Ergebnis entspricht dem der synchronen Iteration, aber ziemlich viel überflüssige Arbeit!

Grundsätzlich gilt für asynchrone Itertation:

- Kann
  - Arbeit/Zeit sparen:
    - Kein Warten darauf, daß anderer Prozeß fertig mit Schritt.
    - Kein Synchronisationsaufwand.
  - Arbeit/Zeit vermehren:
    - Benutzung evtl. veralteter Werte $\Rightarrow$ unproduktive, evtl. schädliche Berechnungsschritte.

  Also asynchrone Iteration *nicht* sinnvoll für 1-Prozessor-Implementierung, da Spekulation auf Zeitersparnis entfällt. !

- *Konvergenz* ist i. A. schwieriger zu beweisen, insbesondere beachte man: Standard-Iteration konvergiert $\not\Rightarrow$ Asynchrone-Iteration konvergiert.

  Hausaufgabe
  Überlegen Sie sich, warum unser Beispiel konvergiert!
  Hinweis:

  1. Klar, daß Werte für P bzw. Delta_P beschränkt.

2. Nimm an, daß Nicht-Konvergenz in einem Punkt, d. h. Differenz von P-Werten nicht unter eine gewisse *Schwelle* zu drücken; betrachte Punkt mit *maximalem* Schwellwert und erschließe Widerspruch.

- *Terminierung* (d. h. praktische Anwendung eines Konvergenzkriteriums) ist schwieriges Problem, zu dem allerdings Lösungen in der Literatur existieren. Unsere Lösung ist sehr grob und läuft auf wiederholte asynchrone Phasen mit Synchro-Punkten dazwischen hinaus.
- *Brauchbarkeit*
  - Als Ausführungsmodus am ehesten brauchbar für *verteilte* Systeme mit sehr hohen Synchronisationskosten. Beispiel: Verteilte Berechnung von routing-Tafeln (= shortest-path Problem) u. ä. in Netzen.
  - Als algorithmische Vorstufe im Programmentwurf: Erfinde *erst* asynchrones Verfahren für gegebenes Problem *und* füge *dann* Synchronisation entsprechend technischen Gegebenheiten, Effizienzbedürfnissen u. ä., hinzu.

8.3.4 Literatur

Asynchrone Iteration als

Programmentwicklungs-Philosophie Clandy/Misra: „Parallel Program Design“, 19..

Technik für numerische Probleme Bertsekas/Tsitsiktis: „Parallel & Distributed Computations: Numerical Methods“, 1984

8.3.5 Hausaufgabe

n Professoren müssen für das Semester (Tag 1 . . . k) einen Termin für die Fakultätssitzung festlegen. Da sie alle einen Terminkalender mit vielen Einträgen haben, selten da sind und auch nicht gern miteinander reden, wird folgendes Verfahren gewählt:

- Im Sekretariat wir ein *Terminvorschlag* ausgehängt.
- Jeder Professor kommt sooft wie möglich im Sekretariat vorbei, vergleicht seinen Terminkalender mit dem aktuellen Vorschlag und verschiebt, *wenn nötig*, letzteren auf den *frühesten individuell möglichen späteren Termin*.
- Alle sind bereit, die Sitzung am *letzten* Tag des Semesters abzuhalten (weil sonst die Fakultät geschlossen wird), so daß das Problem eine Lösung hat.

1. Überlegen Sie sich, daß das beschriebene Verfahren die früheste für alle Profs mögliche Lösung findet.
2. Realisieren Sie das Verfahren mit Ada-tasks und statten Sie das Programm mit einem geeignetem Terminierungsprotokoll aus. (z. B. Terminvorschlag gilt, wenn von allen n Profs *gesehen und nicht geändert*).

Kapitel 9

Ada „requeue" Konstrukt

Bedingungs-Synchronisation mit Hilfe von Wächtern (guards, barriers) vor entries – `when <condition>` – erlaubt, die Erfüllung eines Bedienwunsches vom *Zustand des Bedieners* abhängig zu machen; aber Wächter darf *nicht* Parameter des entry-call benutzen

⇒ Erfüllung des Bedienwunsches kann *nicht* durch Barriere an *Art des Wunsches* (z. B. *Umfang*, *Dringlichkeit*) geknüpft werden.

Klar: Wenige(!) diskrete Fälle können durch Vermehrung der entries gehandhabt werden.

Allgemein:

9.1 Abhilfe

= *doppelte* Interaktion:

1. Sagen, was man will.
2. Dafür anstehen, daß man das kriegt.

Mit unseren bisherigen Mitteln bedeutet das, daß der Klient auch *zwei entry-calls* benötigt; das folgende Programm illustriert die Technik, wobei der Server/Ressourcenverwalter durch ein protected-object repräsentiert wird (ähnlich: task);

9.2 Prinzip der Lösung

Klient, der nicht sofort bedient werden kann (= Entscheidung bei Aufruf der Prozedur „`request(...)`") reiht sich durch Aufruf von entry „`wait`" in eine Warteschlange ein; alle unter „`wait`" aufgeführten Bedienwünsche werden auf Erfüllbarkeit überprüft, sobald eine Rückgabe von Betriebsmitteln erfolgt; nicht befriedigte Klienten müssen sich immer *neu* anstellen!

```
----------------------------------------------------------------------
-- resource management for varying requests
--
-- structure:  requesters = task, resourcepool(+mangement) = protected object
-- version1: double interaction
----------------------------------------------------------------------

with Stringpack, Ada.Numerics.Float_Random; use Stringpack, Ada.Numerics.Float_Random;
procedure Resources1 is

   G : Generator;
   Number_Of_Clients : constant Natural := 3;
   Number_Of_Resources : constant Natural := 5;
```

```
How_Often : constant Natural := 5;


protected Resource_Manager  is ---------------------------------------
   procedure Request(How_Many : Positive; Ok : out Boolean); --+ double interaction
   entry Wait (How_Many : Positive; Ok : out Boolean);       --+
   procedure Release(How_Many : Positive);
private
   Free : Natural := Number_Of_Resources;
   Release_Event : Boolean := False;
   Wanted : Natural := 0;     -- for trace only, offene Wünsche
end Resource_Manager;

protected body Resource_Manager is

   procedure Trace is
      Not_Free : Natural := Number_Of_Resources - Free;
   begin
      Outbuffer := Varnull;
      for I in 1..Number_Of_Resources loop
         if I <= Not_Free then Outbuffer := Outbuffer & " X";
         else Outbuffer := Outbuffer & " -";
         end if;
      end loop;
      Outbuffer := (Outbuffer & "  wanted = ") & Wanted;
      Print;
   end Trace;

   procedure Request(How_Many : Positive; Ok : out Boolean) is
      -- this could be an entry with "when free > 0"
   begin
      if How_Many <= Free
      then Free := Free - How_Many; Ok := True;  --allocate
      else Ok := False;
      Wanted := Wanted + How_Many;   -- for trace
      end if;
      Trace;
   end Request;

   entry Wait (How_Many : Positive; Ok : out Boolean) when Release_Event  is
   begin
      if Wait'Count = 0 then Release_Event := False; end if;
      -- => loop over all waiting requesters !!!
      if How_Many <=  Free
      then Free := Free - How_Many; Ok := True;  --allocate
      Wanted := Wanted - How_Many;   -- for trace
      Trace;
      else Ok := False;
      end if;
   end Wait;

   procedure Release(How_Many : Positive) is
   begin
      Free := How_Many + Free;  -- release
      if Wait'Count > 0 then Release_Event := True; end if;
      Trace;
   end Release;
```

```
   end Resource_Manager;  -----------------------------------------------

   task type Clients is   -----------------------------------------------
      entry Start(How_Many : Natural);
   end Clients;

   task body Clients is
      My_Need : Natural;
      Done : Boolean;
   begin
      accept Start(How_Many : Natural)
      do My_Need := How_Many; end Start;
      for I in 1..How_Often loop
         delay Duration(0.001* Random(G));
         Resource_Manager.Request(My_Need, Done);
         while not Done loop                              --+ immer neu "wait"
            Resource_Manager.Wait(My_Need, Done); end loop;  --+ aufrufen!
         delay Duration(0.001 * Random(G));
         Resource_Manager.Release(My_Need);
      end loop;
   end Clients;

   Client : array(1..Number_Of_Clients ) of Clients;  ----------------

begin  -----------------------------------------------------------main

   Client(1).Start(1); Client(2).Start(2); Client(3).Start(5);

end Resources1;  ----------------------------------------------------

---------------------------------------------------------------------
32 pohlmann@pavian:resources1
X - - - -  wanted = 0
X - - - -  wanted = 5  ! Client 3 fordert 5 Einheiten !
X X X - -  wanted = 5
X X - - -  wanted = 5
- - - - -  wanted = 5
X - - - -  wanted = 5  ! Client 3 wird von Client 1 überholt !
X X X - -  wanted = 5
X X - - -  wanted = 5
- - - - -  wanted = 5
X X X X X  wanted = 0
X X X X X  wanted = 1
X X X X X  wanted = 3
- - - - -  wanted = 3
X - - - -  wanted = 2
X X X - -  wanted = 0
X X X - -  wanted = 5
X - - - -  wanted = 5
- - - - -  wanted = 5
X X X X X  wanted = 0
X X X X X  wanted = 2
X X X X X  wanted = 3
- - - - -  wanted = 3
X X - - -  wanted = 1
X X X - -  wanted = 0
```

```
X X X - -  wanted = 5
X X - - -  wanted = 5
- - - - -  wanted = 5
X X X X X  wanted = 0
X X X X X  wanted = 1
X X X X X  wanted = 3
- - - - -  wanted = 3
X - - - -  wanted = 2
X X X - -  wanted = 0
X X - - -  wanted = 0
X X - - -  wanted = 5
- - - - -  wanted = 5
X X X X X  wanted = 0
- - - - -  wanted = 0
X X X X X  wanted = 0
- - - - -  wanted = 0
33 pohlmann@pavian:
```

Anmerkung des Tippers Auf meinem Rechner (Linux Kernel 2.0.33, gcc version egcs-2.90.23 980102 (egcs-1.0.1 release), GNAT 3.10p) konnte ich diesen Überholvorgang nicht nachvollziehen, was aber nicht weiter ungewöhnlich ist. Schließlich ändert allein die Art der Ausgabe (Bildschirm/Datei) die Reihenfolge komplett:

```
rbla@bravo:resources1
 X - - - -  wanted = 0
 X X X - -  wanted = 0
 X X X - -  wanted = 5
 X X - - -  wanted = 5
 - - - - -  wanted = 5
 X X X X X  wanted = 0
 X X X X X  wanted = 1
 - - - - -  wanted = 1
 X - - - -  wanted = 0
 X X X - -  wanted = 0
 X X - - -  wanted = 0
 - - - - -  wanted = 0
 X X X X X  wanted = 0
 X X X X X  wanted = 1
 - - - - -  wanted = 1
 X - - - -  wanted = 0
 X X X - -  wanted = 0
 X X - - -  wanted = 0
 - - - - -  wanted = 0
 X X X X X  wanted = 0
 X X X X X  wanted = 1
 - - - - -  wanted = 1
 X - - - -  wanted = 0
 X X X - -  wanted = 0
 X X - - -  wanted = 0
 - - - - -  wanted = 0
 X X X X X  wanted = 0
 X X X X X  wanted = 2
 X X X X X  wanted = 3
 - - - - -  wanted = 3
 X X - - -  wanted = 1
 X X X - -  wanted = 0
 X - - - -  wanted = 0
 - - - - -  wanted = 0
 X X X X X  wanted = 0
```

```
rbla@bravo:resources1 >output
 X - - - -  wanted = 0
 X X X - -  wanted = 0
 X X X - -  wanted = 5
 X - - - -  wanted = 5
 - - - - -  wanted = 5
 X X X X X  wanted = 0
 - - - - -  wanted = 0
 X - - - -  wanted = 0
 X X X - -  wanted = 0
 X - - - -  wanted = 0
 - - - - -  wanted = 0
 X X X X X  wanted = 0
 - - - - -  wanted = 0
 X - - - -  wanted = 0
 X X X - -  wanted = 0
 X - - - -  wanted = 0
 - - - - -  wanted = 0
 X X X X X  wanted = 0
 - - - - -  wanted = 0
 X - - - -  wanted = 0
 X X X - -  wanted = 0
 X - - - -  wanted = 0
 - - - - -  wanted = 0
 X X X X X  wanted = 0
 - - - - -  wanted = 0
 X - - - -  wanted = 0
 X X X - -  wanted = 0
 X - - - -  wanted = 0
 - - - - -  wanted = 0
 X X X X X  wanted = 0
 - - - - -  wanted = 0
```

```
 - - - - -  wanted = 0
```

9.2.1 Kritik

Verfahren mit doppelter Interaktion ist

- Umständlich + fehleranfällig
  $\Rightarrow$ Operation für Betriebsmittel anfordern in Paket verkapseln, sodaß Klient-Programm nur 1 Aufruf.
- Ineffizient (wiederholter Aufruf von „`wait`“, aber aufrufende task jedesmal sinnlos aktivieren).
- Unsicher im Hinblick auf *Reihenfolge* der Betriebsmittelzuteilung (d. h. $\neq$ FIFO), siehe Beispiel-Protokoll:

  client 3 $\xrightarrow{5}$ request
  $\xleftarrow{notok}$
  client 1 $\longrightarrow$ release
  client 2 $\longrightarrow$ release
  client 1 $\xrightarrow{1}$ request
  $\xleftarrow{ok}$
  client 2 $\xrightarrow{2}$ request
  $\xleftarrow{ok}$
  client 3 $\longrightarrow$ wait

  d. h. Problem, daß Klient *zwischen* request und wait *unterbrochen* wird.

Abhilfe durch neues *Sprachmittel*:
„*private entry*“ und „*requeue*“ erlauben, die Doppel-Aktion allein durch den entry-Akzeptor ausführen zu lassen, d. h. Klient wird nicht tätig, Server kann Klienten *ohne* Gefahr des Überholens in Warteschlange einfügen, d. h. als atomare Aktion.

client $\longrightarrow$ request
$\downarrow$ requeue
wait

Anstatt:
client $\longrightarrow$ request
$\xleftarrow{notok}$
client $\longrightarrow$ wait

```
----------------------------------------------------------------------
-- resource management for varying requests
--
-- structure:  requesters = task, resourcepool(+mangement) = protected object
-- version2:   requeue unserviced requests
----------------------------------------------------------------------

with Stringpack, Ada.Numerics.Float_Random; use Stringpack, Ada.Numerics.Float_Random;
procedure Resources2 is

   G : Generator;
   Number_Of_Clients : constant Natural := 3;
   Number_Of_Resources : constant Natural := 5;
   How_Often : constant Natural := 5;


   protected Resource_Manager  is -----------------------------------
      entry Request(How_Many : Positive);
      procedure Release(How_Many : Positive);
   private
      entry Wait (How_Many : Positive);           -- private entry for requeue
      Free : Natural := Number_Of_Resources;
```

```
      Release_Event : Boolean := False;
      To_Try : Natural := 0;  -- # requests to be tried on release; <= wait'count
      Wanted : Natural := 0;  -- for trace only
   end Resource_Manager;

   protected body Resource_Manager is

      procedure Trace is
         Not_Free : Natural := Number_Of_Resources - Free;
      begin
         Outbuffer := Varnull;
         for I in 1..Number_Of_Resources loop
            if I <= Not_Free then Outbuffer := Outbuffer & " X";
            else Outbuffer := Outbuffer & " -";
            end if;
         end loop;
         Outbuffer := (Outbuffer & "  wanted = ") & Wanted;
         Print;
      end Trace;

      entry Request(How_Many : Positive) when True is  -- could be "when free > 0"
      begin
         if How_Many <= Free
         then Free := Free - How_Many;   --allocate
         Trace;
         else  Wanted := Wanted + How_Many;   -- for trace
         Trace;
         requeue Wait;
         end if;
      end Request;

      entry Wait (How_Many : Positive) when Release_Event  is
      begin
         To_Try := To_Try - 1; --  loop over all waiting requesters !!!
         if To_Try = 0 then Release_Event := False; end if;
         if How_Many <=  Free
         then Free := Free - How_Many;   --allocate
         Wanted := Wanted - How_Many;   -- for trace
         Trace;
         else requeue Wait;
         end if;
      end Wait;

      procedure Release(How_Many : Positive) is
      begin
         Free := How_Many + Free;  -- release
         if Wait'Count > 0 then To_Try := Wait'Count; Release_Event := True; end if;
         Trace;
      end Release;


   end Resource_Manager;  --------------------------------------------


   task type Clients is   --------------------------------------------
      entry Start(How_Many : Natural);
   end Clients;
```

```
   task body Clients is
      My_Need : Natural;
   begin
      accept Start(How_Many : Natural)
      do My_Need := How_Many; end Start;
      for I in 1..How_Often loop
         delay Duration(0.001* Random(G));
         Resource_Manager.Request(My_Need);
         delay Duration(0.001 * Random(G));
         Resource_Manager.Release(My_Need);
      end loop;
   end Clients;

   Client : array(1..Number_Of_Clients ) of Clients;  ----------------

begin  -----------------------------------------------------------main

   Client(1).Start(1); Client(2).Start(2); Client(3).Start(5);

end Resources2;  -----------------------------------------------------

----------------------------------------------------------------------
35 pohlmann@pavian:resources2
 X - - - -  wanted = 0
 X - - - -  wanted = 5
 X X X - -  wanted = 5
 X X - - -  wanted = 5
 - - - - -  wanted = 5
 X X X X X  wanted = 0  ! Client 3 (mit Bedienversuch 5) kommt durch, weil
 X X X X X  wanted = 1    sofort im Anschluß "requeue" in Warteschlange von
 X X X X X  wanted = 3    "wait" eingefügt !
 - - - - -  wanted = 3
 X - - - -  wanted = 2
 X X X - -  wanted = 0
 X - - - -  wanted = 0
 - - - - -  wanted = 0
 X X X X X  wanted = 0
 X X X X X  wanted = 2
 X X X X X  wanted = 3
 - - - - -  wanted = 3
 X X - - -  wanted = 1
 X X X - -  wanted = 0
 X X - - -  wanted = 0
 - - - - -  wanted = 0
 X X X X X  wanted = 0
 X X X X X  wanted = 1
 X X X X X  wanted = 3
 - - - - -  wanted = 3
 X - - - -  wanted = 2
 X X X - -  wanted = 0
 X - - - -  wanted = 0
 X - - - -  wanted = 5
 - - - - -  wanted = 5
 X X X X X  wanted = 0
 X X X X X  wanted = 2
 X X X X X  wanted = 3
 - - - - -  wanted = 3
 X X - - -  wanted = 1
```

```
 X X X - -  wanted = 0
 X X X - -  wanted = 5
 X X - - -  wanted = 5
 - - - - -  wanted = 5
 X X X X X  wanted = 0
 - - - - -  wanted = 0
36 pohlmann@pavian:
```

Anmerkung des Tippers Zum Vergleich hier wieder das Ergebnis auf meiner Maschine:

```
rbla@bravo:resources2
 X - - - -  wanted = 0
 X X X - -  wanted = 0
 X X X - -  wanted = 5
 X X - - -  wanted = 5
 - - - - -  wanted = 5
 X X X X X  wanted = 0
 X X X X X  wanted = 1
 - - - - -  wanted = 1
 X - - - -  wanted = 0
 X X X - -  wanted = 0
 X X - - -  wanted = 0
 X X - - -  wanted = 5
 - - - - -  wanted = 5
 X X X X X  wanted = 0
 X X X X X  wanted = 1
 - - - - -  wanted = 1
 X - - - -  wanted = 0
 X X X - -  wanted = 0
 X X - - -  wanted = 0
 X X - - -  wanted = 5
 - - - - -  wanted = 5
 X X X X X  wanted = 0
 X X X X X  wanted = 1
 - - - - -  wanted = 1
 X - - - -  wanted = 0
 X X X - -  wanted = 0
 X X - - -  wanted = 0
 X X - - -  wanted = 5
 - - - - -  wanted = 5
 X X X X X  wanted = 0
 X X X X X  wanted = 1
 - - - - -  wanted = 1
 X - - - -  wanted = 0
 X X X - -  wanted = 0
 X X - - -  wanted = 0
 X X - - -  wanted = 5
 - - - - -  wanted = 5
 X X X X X  wanted = 0
 - - - - -  wanted = 0
```

```
rbla@bravo:resources2
 X - - - -  wanted = 0
 X X X - -  wanted = 0
 X X X - -  wanted = 5
 X - - - -  wanted = 5
 - - - - -  wanted = 5
 X X X X X  wanted = 0
 - - - - -  wanted = 0
 X - - - -  wanted = 0
 X X X - -  wanted = 0
 X - - - -  wanted = 0
 - - - - -  wanted = 0
 X X X X X  wanted = 0
 - - - - -  wanted = 0
 X - - - -  wanted = 0
 X X X - -  wanted = 0
 X - - - -  wanted = 0
 - - - - -  wanted = 0
 X X X X X  wanted = 0
 - - - - -  wanted = 0
 X - - - -  wanted = 0
 X X X - -  wanted = 0
 X - - - -  wanted = 0
 - - - - -  wanted = 0
 X X X X X  wanted = 0
 - - - - -  wanted = 0
 X - - - -  wanted = 0
 X X X - -  wanted = 0
 X - - - -  wanted = 0
 - - - - -  wanted = 0
 X X X X X  wanted = 0
 - - - - -  wanted = 0
```

9.3 Zur Syntax und Semantik von „requeue“

- Gibt’s für tasks und protected-objects, oft für „*private*“ entry.
- Entry von requeue muß *dasselbe* Parameterprofil haben wie das ürsprüngliche entry oder muß *parameterlos* sein. Parameter müssen bei requeue nicht extra genannt werden.

- Requeue kann auch entries *anderer* Objekte ansprechen.
- Mit Ausführung des „requeue“ ist der Aufruf des ursprünglichen entry erstmal zuende (d. h. insbesondere, daß jetzt andere Aufrufe des protected-objects durchkommen können).
- Task, die auf ein entry (insbesondere infolge requeue wartet) hat die übliche Präferenz gegen Aufrufer von außen: Nach jeder Zustandsänderung des Objekts werden die Wächter neu ausgewertet und die darauf wartenden tasks bedient: „Internal-Progress-First Rule“.

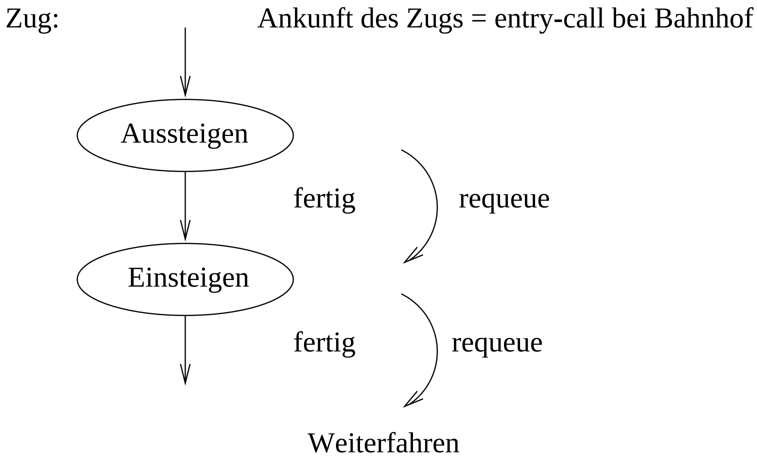
Requeue ist bequemes Mittel zu Lösung kompilzierter *Synchronisationsprobleme*: *Ein* Aufruf einer protected-operation von außen wird *intern*, kann *intern zerlegt* werden in eine Folge *geschützter Teiloperationen*, die in genau festgelegter Weise mit anderen Operationen *verzahnt* sind.

9.4 Beispiel

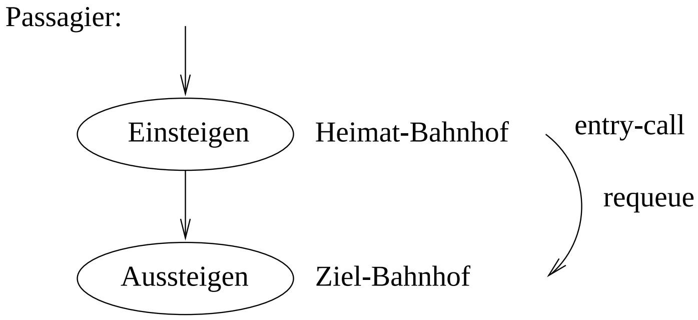
Zu modellieren ist ein *Verkehrssystem* bestehend aus

- einer ringförmigen U-Bahn(o. ä.)-Linie mit *Bahnhöfen*.
- einer dieser Linien (in fester Richtung) bedienender *Zug*.
- Kunden.

Wir realisieren den *Zug* und die *Passagiere* durch *tasks* und den *Bahnhof* (naheliegend) als *protected-object*; im Bahnhof werden Zug *und* Passagiere *gemeinsam* aktiv, Aufenthalt des Zugs im Bahnhof ist charakterisiert durch *2 Phasen*:



Außerdem modellieren wir die *Fahrt* eines *Passagiers* durch *requeue* bei der gewünschten *Ziel-Station*:




```
----------------------------------------------------------------------
-- metro  - illustrates use of requeue
--
-- problem: model circular metro line with 1 train and stations where
--          passengers enter/leave
--
-- structure: passengers & train = task, stations = protected objects
-- requeue is used to 1. delay passengers for trip to their destination,
```

```
--                              2. guide train through phases
--                                 (passengers alight/board) at stations
-- cf: Burns/Wellings ch.8.6 (does not use 2.)
--
-- !!! start protocol ( cf.main ) necessary for initialisation !!!
-- !!! program does NOT terminate (just stops producing output) !!!
----------------------------------------------------------------------

with Stringpack, Ada.Numerics.Discrete_Random; use Stringpack;
procedure Metro is

   Number_Of_Stations : constant Natural := 5;
   subtype Station_Range is Integer range 1..Number_Of_Stations;

   package Random_Destination is new Ada.Numerics.Discrete_Random(Station_Range);
   use Random_Destination;
   G : Generator;

   Number_Of_Clients : constant Natural := 100; -- small town
   Capacity : Integer  := 50;  -- (relatively) big train

   How_Often : constant Natural := 4;  -- traced rounds in sample run

   ---------------------------------------------------------------trace

   type Trace_Type is record
      Train_At_Station : Boolean := False;
      Clients_At_Station : Natural := 0; -- initially there or alighted
      Passenger_Count : Natural := 0;  -- in train on leaving station
   end record;
   Station_Trace : array(Station_Range) of Trace_Type;      --global!!

   procedure Trace is
   begin
      Outbuffer := Varnull;
      for I in Station_Range  loop
         Outbuffer := Outbuffer & Cvis(Station_Trace(I).Clients_At_Station,3);
         if Station_Trace(I).Train_At_Station
         then Outbuffer := Outbuffer &
           " |" & Cvis(Station_Trace(I).Passenger_Count,2) &"> ";
         else Outbuffer := Outbuffer & "      ";
         end if;
      end loop;
      Print;
   end Trace;

   -------------------------------------------------------------------

   protected type Stations  is --------------------------------------
      procedure Start(Id : Station_Range);
      entry Train_Comes_In(On_Board : in out Natural);
      entry Board_The_Train(Where : Station_Range);
   private
      My_Id : Station_Range;                         -- must be set initially!!!
      entry Alight_At_Destination;           -- private entry to requeue client
                                              -- private entry to requeue train
      entry Passengers_Board(On_Board : in out Natural);
                                              -- private entry to requeue train
```

```
      entry Close_Doors(On_Board : in out Natural);
      Passenger_Count : Natural := 0;                    -- # Passagiere in Zug
      Boarding, Alighting : Boolean := False;            --  Phase
   end Stations;

   Station : array(Station_Range) of Stations;

   protected body Stations is

      procedure Start(Id : Station_Range) is
      begin My_Id := Id; end Start;

      entry Board_The_Train(Where : Station_Range)
      when Boarding and then Passenger_Count < Capacity is
      begin
         Passenger_Count := Passenger_Count + 1;                  -- fuelle Zug
         Station_Trace(My_Id).Clients_At_Station :=
           Station_Trace(My_Id).Clients_At_Station - 1;   -- trace
         requeue Station(Where).Alight_At_Destination;        -- Ziel der Reise
      end Board_The_Train;

      entry Train_Comes_In(On_Board : in out Natural)
      when True is                           -- requeue nur in entry erlaubt!!!
      begin
         Station_Trace(My_Id).Train_At_Station := True; -- trace
         Passenger_Count := On_Board;
         Alighting := True;
         requeue Passengers_Board;
      end Train_Comes_In;

      entry Alight_At_Destination when Alighting is
      begin
         Passenger_Count := Passenger_Count - 1;   -- leave train
         Station_Trace(My_Id).Clients_At_Station :=
           Station_Trace(My_Id).Clients_At_Station + 1;  -- trace
      end Alight_At_Destination;

      entry Passengers_Board(On_Board : in out Natural)
      when Alight_At_Destination'Count = 0 is
      begin
         Alighting := False;
         Boarding := True;
         requeue Close_Doors;
      end Passengers_Board;

      entry Close_Doors(On_Board : in out Natural)
      when  Boarding and (Board_The_Train'Count = 0 or Passenger_Count = Capacity) is
      begin
         Boarding := False;
         On_Board := Passenger_Count;  -- out Parameter von train_comes!!
         Station_Trace(My_Id).Passenger_Count := Passenger_Count;  --trace
         Trace;
         Station_Trace(My_Id).Train_At_Station := False; -- trace
      end Close_Doors;

   end Stations; -----------------------------------------------------

   task type Clients is   --------------------------------------------
```

```
      entry Start;
   end Clients;

   task body Clients is
      From, To : Station_Range;
   begin
      accept Start;
      From := Random(G);
      Station_Trace(From).Clients_At_Station :=
        Station_Trace(From).Clients_At_Station + 1;  -- trace
      loop  -- infinite loop !!!!
         To := Random(G);
         Station(From).Board_The_Train(To);
         From := To;
      end loop;
   end Clients;

   Client : array(1..Number_Of_Clients ) of Clients;  ----------------

   task Train is   ---------------------------------------------------
      entry Start;
   end Train;

   task body Train is
      Nr_Passengers : Natural := 0;
   begin
      accept Start;
      for I in 1..How_Often loop  --sample run
         for J in Station_Range loop
            Station(J).Train_Comes_In(Nr_Passengers);
         end loop;
      end loop;
   end Train;


begin  ------------------------------------------------------------main

   for I in Station_Range loop Station(I).Start(I); end loop;
   for I in 1..Number_Of_Clients loop Client(I).Start; end loop;
   Train.Start;

end Metro;  -----------------------------------------------------------

-----------------------------------------------------------------------
24 pohlmann@pavian:metro
  0 |32>  20       20       18       10
  0        7 |45>  20       18       10
  0        7       15 |50>  18       10
  0        7       15       18 |50>  10
  0        7       15       18       12 |48>
 27 |21>   7       15       18       12
 27       11 |17>  15       18       12
 27       11       12 |20>  18       12
 27       11       12        7 |31>  12
 27       11       12        7        9 |34>
 20 |41>  11       12        7        9
 20        9 |43>  12        7        9
 20        9       14 |41>   7        9
```

```
 20          9         14         14 |34>    9
 20          9         14         14         8 |35>
 16 |39>     9         14         14         8
 16          8 |40>    14         14         8
 16          8         15 |39>    14         8
 16          8         15         14 |39>    8
 16          8         15         14         6 |41>
^C
25 pohlmann@pavian
```

= Situation bei *Abfahrt* des Zugs:
#(Passagiere) in Stationen (nicht notwendig in queue!), #(Passagiere) im Zug `|>`, $\sum = 100$.

9.5 Hausaufgabe

Das vorstehende Programm ist verbesserungsfähig:

1. Der Gebrauch der Zustandsvariablen/private entries ist etwas übertrieben für den Anlaß; vereinfachen Sie das Programm.

2. Überlegen Sie sich Möglichkeiten, das Programm ordentlich terminieren zu lassen! Schwierigkeit ist offenbar, daß clients auf ein entry warten können, wenn der Zug Feierabend macht. Diskutieren/probieren Sie Lösungen, bei denen

   - der Ablauf in „station“ durch weitere Zustandsgrößen/Handlungsalternativen verfeinert wird.
   - die Klienten früher Schluß machen (nach n Fahrten) als der Zug.
   - von uns noch nicht behandelte Sprachmittel Adas (insbesondere „`requeue ...with abort`“ sowie „asynchronous transfer of control“, kurz ATC) eingesetzt werden.

Kapitel 10

Concurrency and Objects

10.1 Grundsätzliches

Nebenläufigkeit (tasks, protected objects) und Vererbung (extendible types) sind in Ada *getrennte Konzepte*. Grund: Prinzipielle semantische Schwierigkeiten, Effizienzprobleme. !

⇒ Der Programmierer muß fallweise selbst geeignete *Kombination* von tasks/protected objects und Typhierachien bilden, wobei

- Tasks/protected objects definieren allgemeines Verhaltens-/Synchronisationsschema.
- Damit gekapselte Objektdefinitionen beinhalten Spezialisierungen, Varianten.

Offenbar 2 Möglichkeiten der *Kombination*.

1. Objekt hat task/protected object als Attribut.
2. Task/protected object ist über Diskriminante (access Typ!) mit task/protected object gekoppelt.

Grundsätzlich gibt es 2 Anwendungszwecke, mit fließenden Übergängen.

1. Mache Objekte *aktiv*, d. h. statte sie mit eigenem Verhalten aus (vgl. z. B. Simula Prozesse).
   ∼ Prozeß + Vererbung, wobei letzters problematisch.
2. Erlaube, daß *passives* Objekt in Umgebung mit *Nebenläufigkeit* benutzt wird, z. B. mehrere tasks greifen auf Objekt zu, d. h. statte Objekt mit *Synchronisationsmechanismus* aus.
   ∼ Protected object + Vererbung.

Wir betrachten im folgenden die zweite Zielsetzung!

10.2 Einfache Synchronisationsmechanismen für Objektzugriffe

Aufgabe Zugriffe auf Objekt-Operationen sollen unter *wechselseitigem Ausschluß* erfolgen.

10.2.1 Objekte enthalten Synchronisationsmechanismen als Attribut

Sei definiert

```
protected type Mutex is
   entry Lock;
   procedure Unlock;

private
   Open: Boolean := True;
end Mutex;

protected body Mutex is
```

```
   entry Lock when Open is
   begin
      Open := False;
   end Lock;

   procedure Unlock is
   begin
      Open := True;
   end Unlock;
end Mutex;
```

Dann bilde *Wurzeltyp* für beabsichtigte Typen:

```
package Lockable_Pack is
   type Lockable is abstract tagged limited private; -- Wurzeltyp

                          -- eine Methode, geschützt durch mutex!
   procedure Synchronized_Op(Object: in out Lockable´Class);

private
   type Lockable is tagged limited record
      Synchronizer: Mutex;
   end record;

   procedure Op(Object: in out Lockable) is abstract;
end Lockable_Pack;

package body Lockable_Pack is
   procedure Synchronized_Op(Object: in out Lockable´Class) is
   begin
      Object.Synchronizer.Lock;
      Op(Object);  -- dispatch to actual operation
      Object.Synchronizer.Unlock;
   end Synchronized_Op;
end Lockable_Pack;
```

Also

- Umgebe tatsächliche Operation mit Protokolloperation lock/unlock.
- Mache tatsächliche Operation *privat* und exportiere *nur* die um das Protokoll bereicherte Operation.
- Tatsächliche Operation ist hier noch abstrakt und kann später bei Typableitung deklariert werden.

Kleines Problem Wenn abstrakte Operation op *privat*, wie kann sie dann für Typerweiterung sichtbar gemacht werden?

Ada-Ausweg

Child-packages Pakete, die privaten Teil des Eltern-Pakets anschauen dürfen.
Notation: `<Elternname>.<Paketname>`

Also

```
package Lockable_Pack.Child is
   type Lockable_Concrete is new Lockable with private;

private
   procedure Op(Object: in out Lockable_Concrete);
```

```
end Lockable_Pack.Child;

package body Lockable_Pack.Child is
   type Lockable_Concrete is new Lockable with null record;

   procedure Op(Object: in out Lockable_Concrete) is
   begin
      ...
   end Op; -- override abstract op
end Lockable_Pack.Child;
```

Natürlich kann man auch weitere Dinge hinzufügen, z. B.

```
package Lockable_Pack.Child2 is
   type Extended_Lockable is new Lockable with private;

   procedure Synchronized_Op2(Object: in out Extended_Lockable);

private
   type Extended_Lockable is new Lockable with record
      ...  -- new state variables
   end record;

   procedure Op(Object: in out Extended_Lockable);  -- override
   procedure Op2(Object: in out Extended_Lockable);
end Lockable_Pack.Child2;
```

Und im Rumpf dann Synchronized_Op2 wieder unter Anwendung des Protokolls definieren.

```
package body Lockable_Pack.Child2 is
   procedure Op...
   procedure Op2...

   procedure Synchronized_Op2(Object: in out Extended_Lockable) is
   begin
      Object.Synchronizer.Lock;
      Op2(Object);
      Object.Synchronizer.Unlock;
   end Synchronized_Op2;
end Lockable_Pack.Child2;
```

Nachteil: Klient greift nicht selbst (nicht direkt) auf protected object zu, sodaß z. B. Timeouts schwieriger zu realisieren sind. !

10.2.2 Protected object ist über Diskriminante mit zu schützendem Object verbunden

```
package Lockable_Pack is
   type Objects is abstract tagged null record;  -- Wurzeltyp

   protected type Mutex_Plus(O: access Objects´Class) is
      procedure Synchronized_Op;
   end Mutex_Plus;

private
   procedure Op(Object: in out Objects) is abstract;
end Lockable_Pack;
```

```
package body Lockable_Pack is
   protected body Mutex_Plus is
      procedure Synchronized_Op is
         begin
            Op(O.all);
         end Synchronized_Op;
   end Mutex_Plus;
end Lockable_Pack;
```

Anwendung dann wieder so, daß im child-package eine konkrete Version für op definiert wird.

```
package Lockable_Pack.Child is
   type Concrete_Objects is private;

private
   type Concrete_Objects is new Objects with null record;
                                    -- könnte natürlich auch was dazukommen!
   procedure Op(Object: in out Concrete_Objects);  -- override abstract op

   O: aliased Concrete_Objects;  -- erzeuge Objekt,
   Synchronizer: Mutex_Plus(O´access);  -- und kopple mit mutex!
end Lockable_Pack.Child;

package body Lockable_Pack.Child is
   procedure Op(Object: in out Concrete_Objects) is
   begin
      ...
   end Op;
end Lockable_Pack.Child;
```

Großer Nachteil der Methode: Zu Beginn, d. h. bei Definition des protected objects „mutex_plus“ muß Anzahl (und Art) der Operationen (Methoden) der Objektfamilie schon feststehen – man kann später nichts mehr hinzufügen zu dein Operationen des protected_objects. !

Was bleibt dann bei dieser Methode noch für Typerweiterung übrig? Man kann:

- Beteiligte Datentypen *variieren* (z. B. Keller für Integer, Char, . . . = generics!)
- Implementierung *variieren* (z. B. Keller durch array, linked list)

und diese Varianten als zu *einem* Typ gehörig behandeln und verwalten (Unterschied zu generics).

10.3 „Bounded Buffer“ im Stil 10.2.2 (Code aus Burns/Wellings)

```
package Buffers is
   type Data abstract tagged null record;  -- lasse Elementtyp offen
   type Bufer is abstract tagged private;  -- lasse Implementierung offen

   -- Basic protected Buffer
   protected type Buffer_Controller(B: access Buffer´Class) is
      entry Put(D: in Data´Class);  -- entries, weil condition snychro
      entry Get(D: out Data´Class);
   end Buffer_Controller;

   procedure Put(D: Data´Class; B: access Buffer) is abstract;
   procedure Get(D: Data´Class; B: access Buffer) is abstract;
   function Buffer_Not_Full(B: access Buffer) return Boolean is abstract;
   function Buffer_Not_Empty(B: access Buffer) return Boolean is abstract;
```

```
private
   type Buffer is abstract tagged null record;
end Buffers;

package body Buffer is
   protected body Buffer_Controller is
      entry Put(D: in Data´Class) when Buffer_Not_Full(B) is
         -- note Buffer_Not_Full(B) is a dispatching operation
      begin
         Put(D, B); -- dispatching operation
      end Put;

      entry Get(D: in Data´Class) when Buffer_Not_Empty(B) is
         -- note Buffer_Not_Empty(B) is a dispatching operation
      begin
         Get(D, B); -- dispatching operation
      end Get;
   end Buffer_Controller;
end Buffers;
```

Jetzt Typerweiterung: Puffer soll als Vektor realisiert werden und kriegt hier entsprechende Attribute.

```
package Buffers.Array_Based_Buffers is
   type Array_Buffer is abstract new Buffer with private;
   Buffer_Size: constant Natural := 10;

   procedure Put(D: Data´Class; B: access Array_Buffer) is abstract;
   procedure Get(D: out Data´Class; B: access Array_Buffer) is abstract;

   function Buffer_Not_Full(B: access Array_Buffer) return Boolean is abstract;
   function Buffer_Not_Empty(B: access Array_Buffer) return Boolean is abstract;

private
   -- The package could be made generic, and the size passed as a generic parameter
   subtype Index is mod Buffer_Size;
   subtype Count is Natural range 0..Buffer_Size;

   type Array_Buffer is abstract new Buffer with record
      First: Index := Index´First;
      Last: Index := Index´First;
      Number_In_Buffer: Count := 0;
   end record;
end Buffers.Array_Based_Buffers;
```

Hier noch als *abstrakt*, *keine* Implementierung, da *Elementtyp* noch nicht feststeht. !

Zwischenschritt ist sinnvoll, wenn man array-basierte Puffer für verschiedene Elementtypen haben will. Implementierung: Integer Elemente

```
package Buffers.Array_Based_Buffers.Integer_Buffers is
   type Integer_Data is new Data with record
      X: Integer;
   end record;

   type Integer_Array_Buffer is new Array_Buffer with private;

private
   -- a bounded buffer
```

```
   type Integer_Array is array(Index) of Integer_Data;
   -- child package has visibility of the private part of the parent

   type Integer_Array_Buffer is new Array_Buffer with record
      Mb1: Integer_Array;
   end record;

   procedure Put(D: Data´Class; B: access Integer_Array_Buffer);
   procedure Get(D: out Data´Class; B: access Integer_Array_Buffer);

   function Buffer_Not_Full(B: access Integer_Array_Buffer) return Boolean;
   function Buffer_Not_Empty(B: access Integer_Array_Buffer) return Boolean;
end Buffers.Array_Based_Buffers.Integer_Buffers;
```

Jezt Implementierung möglich.
Implementierung:

```
package body Buffers.Array_Based_Buffers.Integer_Buffers is
   procedure Put(D: Data´Class; B: access Integer_Array_Buffer) is
   begin
      B.Mb1(B.Last) := Integer_Data(D);  -- may generate Constraint_Error
      B.Last := B.Last +1;
      B.Number_In_Buffer := B.Number_In_Buffer +1;
   exception
      when Contraint_Error =>
         -- potential error recovery
         raise;
   end Put;

   procedure Get(D: out Data´Class; B: access Integer_Array_Buffer) is
   begin
      D := Data´Class(B.Mb1(B.First));
      B.First := B.First +1;
      B.Number_In_Buffer := B.Number_In_Buffer -1;
   end Get;

   function Buffer_Not_Full(B: access Integer_Array_Buffer) return Boolean is
   begin
      return B.Number_In_Buffer = Buffer_Size;
   end Buffer_Not_Full;

   function Buffer_Not_Empty(B: access Integer_Array_Buffer) return Boolean is
   begin
      return B.Number_In_Buffer /= 0;
   end Buffer_Not_Empty;
end Buffers.Array_Based_Buffers.Integer_Buffers;
```

Eleganz der Gesamtkonzeption ist wie oft bei OO verbunden mit Laufzeit-Unsicherheit: Weil Put/Get für Datenelemente Parameter von *klassenweiten* Typ haben, ist Aufruf mit anderem (von Data abgeleitetem) Typ ebenfalls möglich. !

10.4 Inheritance Anomaly

Die beiden oben dargestellten Techniken sind in ihrer Anwendbarkeit sehr beschränkt:

- 10.2.2 verlangt, daß vorweg klar ist, welche Operationen der Synchronisation durch das protected object unterworfen werden sollen,

- 10.2.1 erlaubt zwar, weitere Operationen hinzuzufügen, ermöglicht aber *keine freie Gestaltung/Änderung/Bedingungssynchronisation.*

In dem in 10.2.1 beschriebenem Verfahren kann man beliebig viele Operationen op2, op3 usw. dazunehmen und dem wechselseitigen Ausschluß unterwerfen (oder eben nicht), aber es ist *unmöglich*, daß z. B.

- op1 muß allein sein, op2, op3 zusammen.
- Zustandsabhängige Synchronisation/Ausschluß (z. B. Puffer voll/leer).

Solche Änderungen sind nur unter *Aufhebung* des *Geheimnisprinzips* und *Änderung* früher definierter Operationen möglich. = Inheritance anomaly!

Im folgenden Diskussion des Problems unabhängig von Ada.

Klassische Übersicht

- S. Matsoka, A. Yonezawa: „Analysis of Inheritence Anomaly in OO Concurrent Languages“, Proc. Research Directions in concurrent OOP (Ayha, Wegno, ed.), MIT, 1993

Ada95-spezifischer Artikel

- G. Schumacher, W. Nebel: „How to Avoid the Inheritance Anomaly in Ada“, Proc. Ada-Europe ´98, Springer LNCS 1411, 1998
  Ziemlich weitreichende Lösung in Ada95, aufwendig und zum Teil unschön, z. B. massiver Gebrauch von exceptions zur Ablaufsteuerung.

Allgemeine Aufgabe

- Methoden eines Objekts unterliegen hinsichtlich Aufrufbarkeit *Synchronisationsbedingungen*, also zu jedem Zeitpunkt immer nur Teilmenge der existierenden Methoden aufrufbar, um Integrität des Objekts zu schützen: Konkurrenter (mehrere Klienten Prozesse) Zugriff kann z. B. erfordern:
  - Wechselseitiger Ausschluß
  - Objekt in bestimmtem Zustand.
- Deshalb enthält Klassen- (Objekt-) Definition *Synchronisationscode* (insbesondere als Teil von Methodenimplementierungen) zur Realisierung der Synchro-Bedingungen.
- Dann *Konflikt* Vererbung ↔ Synchronisation: Neueinführung/Überschreibung einer Methode kann zu massiven Änderungen scheinbar unbeteiligter Elemente in der Klassenhierarchie führen!

Problem hat noch keine befriedigende und allgemein akzeptierte Lösung und ist Gegenstand aktueller Forschung und Diskussion; aber beachte: Abhängig von Sprachmitteln, je nach Formulierung verschieden schwerwiegend. Wir betrachten *Beispiele*, die alle beruhen auf dem *Anwendungsfall* „bounded buffer“ mit Get/Put Operationen.

10.4.1 Objekte mit Rumpf (body)

Aktives Objekt mit eigenem Kontrollfluß (= spezielle Methode, „Hauptprogramm“ des Objekts, Prozeß).
Rumpf *kontrolliert* Methodenausführung, vgl. Ada: Service_Task (= Endlosschleife mit selektivem accept).
Dabei kann dann akzeptierter Methodenaufruf konkurrent oder innerhalb des Rumpf-Prozesses ausgeführt werden.

Offenbar: Für jede Unterklasse, die neue Methoden einführt oder Synchronisationsbedingungen ändert, muß *body neu definiert* werden. !

10.4.2 Akzeptanz-Mengen („enabled sets“)

Am Ende der Ausführung einer Methode wird jeweils explizit die Menge der Methoden festgelegt, die als nächstes akzeptiert werden; dabei:

- wechselseitiger Ausschluß unterstellt
- Bezeichner für enabled sets

Beispiel

```
Class Bounded_Buffer is
  procedure Put(...);
  procedure Get(...);

  Enabled Sets:
    Empty := Put,
    Partial := Put, Get,
    Full := Get;

Implementation
  In, Out: Integer := ...;
  Buf: array(Size) of ...;

  procedure Put(...) is
  begin
     ...
     if In = Out then
       Become Full;
     else
       Become Partial;
     end if;
  end Put;

  procedure Get(...) is
  begin
     ...
     if In = out then
       Become Empty;
     else
       Become Partial;
     end if;
  end Get;
end Bounded_Buffer;
```

Jetzt neue Operation: `procedure last(...)` entnimmt *letzten* Eintrag.

```
Class X_Buffer Extends Bounded_Buffer is
  procedure Last(...);

  Enabled Sets:
    Partial := Put, Get, Last,
    Full := Get, Last;

Implementation
  procedure Last(...) is
  begin
    ...
    if In = out then
      Become Emtpy;
    else
      Become Partial;
    end if;
  end Last;
end X_Buffer;
```

D. h. einfache *Erweiterung* von enabled-set! Aber das ist *nicht immer* möglich:

Beispiel Führe neue Operation „Get2“ ein.
Bedeutung: Hole 2 Elemente auf einmal; das ist offenbar *nicht* durch zweimaliges Get zu machen, weil anderer Klienten-Prozeß intervenierend Puffer leeren könnte ...
⇒ Get erfordert *Aufspaltung* eines vorhandenen enabled sets, d. h. *Verfeinerung* von Partial – nur 1 Element - 2 oder mehr Elemente - vorhanden.
⇒ Get/Put müssen in ihrem *becomes*-Teil geändert werden.

Immerhin nur lokalisierte Änderung, *ferner*: become-Teil kann *interne Methode* sein, die man *überschreibt.* !

10.4.3 Wächter (guards) für Methoden

Ähnlich in Problemen und Möglichkeiten wie letzter Ansatz. Wächter: Vgl. Ada entries.
Anwendung auf Beispiel: Klar, z. B. `procedure Get(...) when (In >= Out +1) is ...`

Kann Fälle wie „Get2“ vereinfachen, wichtig auch hier: Guard überschreibbar? !

Aber: Methode versagt, wenn vorhandene *Zustandsgrößen* die gewünschte Bedingung *nicht auszudrücken* vermögen. !

Typisches Beispiel: Zustand = Details der Vor-Geschichte, z. B. Reihenfolge früherer Methodenaufrufe.

Natürlich gilt immer: Zustand = Akkumulierte Vorgeschichte, aber tatsächliche Wahl von Zustandsbegriff immer bezogen auf Funktion, d. h. Abstraktion/Äquivalenzklassenbildung; Vorgeschichte = *feinster* Zustandsbegriff. !

Beispiel Führe Operation „Gget“ ein, Bedeutung: Wie Get, *aber nur ausführbar, wenn* letzte vorangegangene Operation auch ein Get war.

Offenbar nicht ausdrückbar mit vorhandenen Variablen (Zustandsattributen), man muß zusätzlich festhalten, ob letzte Operation Get oder Put war. !

Dies ist immerhin noch als *Verfeinerung* des Zustandsraums zu machen: Gget enabled ⇒ Get enabled, aber auch möglich, *inkompatibles Verhalten* einzuführen: Operation *Lock/Unlock*, die Ausführung anderer Operationen *sperren* oder wieder freigeben, d. h. Get/Put nicht immer zulässig unter alten Bedingungen, sondern nur unter *Einschränkungen* davon.
⇒ Neue Guards für Get/Put nötig.