

Dieses Programm verhindert zwar, dass sich beide Prozesse gleichzeitig in ihrem kritischen Abschnitt befinden können, aber wenn die Anweisungsfolgen (wie oft üblich) in einer unendlichen Schleife stehen, dann können die beiden Prozesse nur abwechselnd ihren kritischen Abschnitt betreten, und beide Prozesse müssen dauernd aktiv bleiben, sonst wartet der andere Prozess ewig:

Unbrauchbare Lösung

```
[ local X, Y: integer;
  type prozess is (PA, PB); Nr: prozess := PA;

  (A:: while true loop
    X:=1; Nr := PB;
    await Nr = PA;
    write(X);
    Nr := PB; X:=X+1;
  end loop;
  B:: while true loop
    Y:=3; Nr := PA;
    await Nr = PB;
    write(Y);
    Nr := PA; Y:=Y+1;
  end loop; ) ]
```

2.2.2.9 Problemformulierung: Gesucht wird also eine Softwarelösung, bei der jeder Prozess unabhängig vom anderen ist, außer in dem Fall, dass beide in einem gleichen Zeitraum in ihren kritischen Abschnitt eintreten wollen. Hierzu gibt es diverse Lösungen in der Literatur (z.B.: T.Dekker 1965, G.L.Peterson 1981, S.Owicki und L.Lamport 1982).

Versuchen Sie zunächst selbst, eine Lösung für Vorbereitung A bzw. B und Nachbereitung A bzw. B des allgemeinen Problems zu finden:

```
[ local <Deklarationen>;
  (A:: while true loop
    Vorbereitung A;
    kritischer Abschnitt A;
    Nachbereitung A;
  end loop;
  B:: while true loop
    Vorbereitung B;
    kritischer Abschnitt B;
    Nachbereitung B;
  end loop; ) ]
```

Wir geben hier nur die Lösung von Peterson an. Jeder Prozess hat hierbei eine eigene Boolesche Variable PrA bzw. PrB, die den Wunsch, in den kritischen Abschnitt einzutreten, signalisiert. Zusätzlich gibt es eine Variable "dran", die dem anderen Prozess den Vortritt lässt, sofern dieser auch gerade in den kritischen Abschnitt will.

Diese Lösung erfüllt die geforderten Eigenschaften:

- Es kann sich zu jedem Zeitpunkt nur ein Prozess in seinem kritischen Abschnitt befinden.
- Jeder Prozess kann in seinen kritischen Abschnitt gelangen unabhängig davon, wo sich der andere Prozess befindet oder ob er noch aktiv ist.
- Es tritt keine Verklemmung auf.

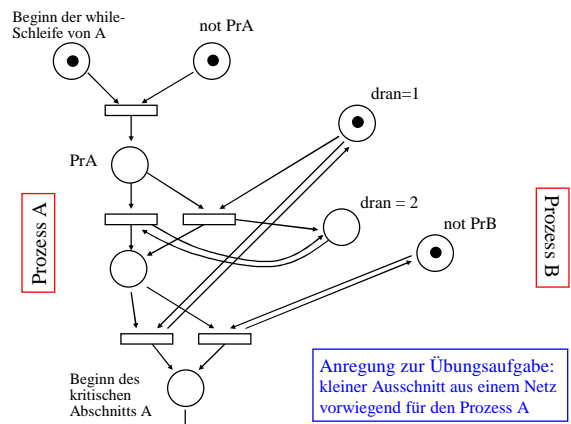
2.2.2.10 Die Lösung von Peterson (1981):
(*or ist hier das Oder in einem Booleschen Ausdruck*)

```
[ local dran: Integer:=1; PrA, PrB: Boolean:= false;
  (A:: while true loop
    PrA := true;
    dran := 2;
    await (not PrB) or (dran=1);
    < kritischer Abschnitt A >;
    PrA := false;
    ...
  end loop;
  B:: while true loop
    PrB := true;
    dran := 1;
    await (not PrA) or (dran=2);
    < kritischer Abschnitt B >;
    PrB := false;
    ...
  end loop; ) ]
```

In der Vorlesung wird die Arbeitsweise dieses Vorgehens genauer erläutert. Machen Sie sich diese Arbeitsweise an einem Ablaufdiagramm klar!

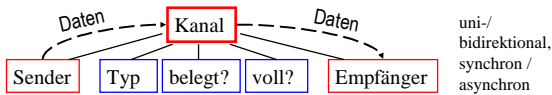
Zu diesem Programm kann man ein Stellen-Transitions-Netz zeichnen, das die Arbeitsweise genau widerspiegelt. Einen Beweis für die Korrektheit der Peterson-Lösung kann man dann über dieses S/T-Netz führen.

Übungsaufgabe: Konstruieren Sie dieses S/T-Netz zu der oben angegebenen Peterson-Lösung.



2.2.2.11 Kanäle: Wir haben einige Aspekte des Nachrichtenaustausches vorgestellt, der über einen gemeinsamen Speicherbereich erfolgt. Anders funktioniert das Telefonieren: Dort wird jedem solchen Nachrichtenaustausch ein eigener Kanal zur Verfügung gestellt, der nach dessen Beendigung einer anderen Kommunikation zugeordnet werden kann.

Anstelle des Ablegens von Informationen in einem gemeinsamen Speicherbereich betrachten wir nun also die Nutzung von Kanälen, über die eine Verbindung zwischen zwei Partnern hergestellt werden kann.



Für Kanäle erlauben wir übergreifend den Datentyp "`channel [1..max] of T`", in den Daten d vom Typ T mittels $CH \leftarrow d$ vom Sender hineingelegt und aus dem Daten dieses Typs vom Empfänger mittels $CH \rightarrow X$ der Variablen X zugewiesen werden können. Benutzen zwei Prozesse den gleichen Kanal, so muss einer **Sender** und einer **Empfänger** sein und es können Daten nur vom Sender an den Empfänger geschickt werden. Ein Kanal ist wie eine **Warteschlange** organisiert und er besitzt in der Regel eine **Kapazität**. Die Daten, die zuerst hineingesteckt werden, kommen auch als erste wieder heraus (FIFO-Prinzip), und die Warteschlange kann meist nur die begrenzte Zahl "max" von Daten aufnehmen. Mit einem Kanal muss weiterhin eine Boolesche Variable "**belegt**" verbunden sein, die einen Kanal nicht frei gibt, sofern er derzeit benutzt wird.

Die Anweisung $CH \leftarrow X$ in einem Prozess P besagt also:

Wenn der Kanal CH nicht belegt ist, so wird er als belegt gekennzeichnet und P ist der Sender für CH ;
 wenn CH belegt ist und bisher nur der Empfänger dem Kanal zugeordnet ist, so wird P der Sender von CH ;
 wenn CH belegt ist und schon zwei Partner besitzt, so muss P unter diesen Partnern der Sender sein.

Trifft eine dieser drei Bedingungen zu, so wird geprüft, ob die Daten (hier: der Wert von X) vom Typ T ist und ob CH noch Platz für die Aufnahme eines weiteren Datums besitzt.

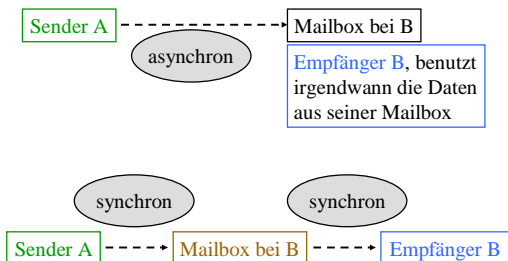
Trifft auch dies zu, so wird der Wert von X in den Kanal gelegt; anderenfalls erfolgt eine geeignete Fehlermeldung.

Analog ist die Anweisung $CH \rightarrow X$ in einem Prozess P zu interpretieren.

Will man einen Datenaustausch zwischen zwei Prozessen installieren, so muss man zwei Kanäle verwenden (wie beim Telefonieren). Will man den Zustand der Kanäle noch überwachen oder unabhängig von den Daten Kontrollinformationen übertragen, so muss man einen oder zwei weitere Kanäle hinzunehmen (wie beim Telefon).

Man muss noch festlegen, ob eine synchrone Verbindung besteht (wenn A sendet, so muss B zeitgleich empfangen; A kann erst weiterarbeiten, wenn B alle Daten empfangen hat) oder ob die Daten asynchron überliefert werden (z.B. in eine Mailbox gelegt werden; allerdings muss dann die Mailbox mit A oder mit dem Kanal synchron zusammenarbeiten).

2.2.2.12: Hieraus folgt: Eine asynchrone Kommunikation zwischen zwei Prozessen kann man durch zwei synchrone Kommunikationen zwischen drei Prozessen simulieren:

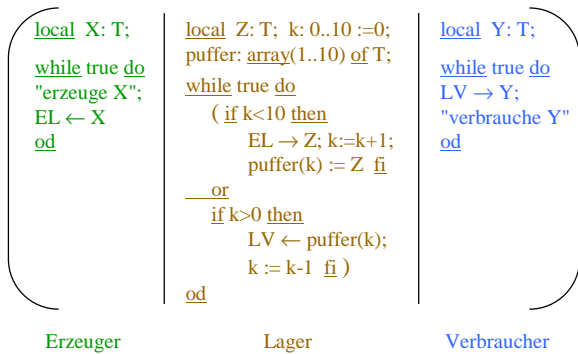


Ein solcher Fall liegt beim Erzeuger-Verbraucher-Kreislauf vor (siehe 2.2.1.3): Der Erzeuger schickt asynchron seine Produkte an den Verbraucher. Fasst man das Lager als zusätzlichen Prozess auf, so lässt sich dieser Vorgang synchron darstellen (siehe nächste Folie).

Wir wollen diese Ausführungen nun mit einem Beispiel beenden, an dem die prinzipielle Arbeitsweise von Kanälen abgelesen werden kann. Das Thema des Nachrichtenaustausches wird in Vorlesungen über Betriebssysteme, Verteilte Systeme und Sichere Systeme weiter vertieft.

Als Beispiel wählen wir den Erzeuger-Verbraucher-Kreislauf.

2.2.2.13 Beispiel EL, LV: channel [1..1] of T;



Noch einige Begriffe:

- Geblockte Übertragung: Daten werden meist nicht einzelnen, sondern in größeren Einheiten (Blöcken) übertragen. Dies erhöht vor allem die Effizienz der Übertragung.
 - Gepackte Daten: Daten werden oft noch komprimiert, damit sie weniger Platz benötigt. Beim Empfänger müssen sie dann wieder "entpackt" werden (Beispiel: zip-Files).
 - Synchron: Der Empfänger übernimmt die Daten, während der Sender sendet.
 - Asynchron: Die Daten werden irgendwo gepuffert, bis der Empfänger sie abholt. Das Puffern kann auch im Kanal integriert sein.
 - Blockierendes Senden: Der Sender kann erst weiterarbeiten, wenn alle gesendeten Daten entweder beim Empfänger angekommen sind oder vom Puffer des Kanals aufgenommen wurden.
 - Blockierendes Empfangen: Der Empfänger kann erst weiterarbeiten, wenn alle gesendeten Daten bei ihm abgespeichert sind.
- Den Datenaustausch mittels synchronem blockierendem Senden und blockierendem Empfangen bezeichnet man als **Rendezvous**. Dies ist in Ada realisiert, siehe 2.2.4.

Gliederung des Kapitels

2.2 Prozesse

- ~~2.2.1 Stellen Transitions Netze~~
- ~~2.2.2 Nachrichtenaustausch~~
- ~~2.2.3 Parallelität~~
- 2.2.4 Nebenläufigkeit in Ada95

Muss hier aus Zeitmangel entfallen; aber: Sortieren!

2.2.4 Nebenläufigkeit in Ada

2.2.4.1 Einführende Begriffe

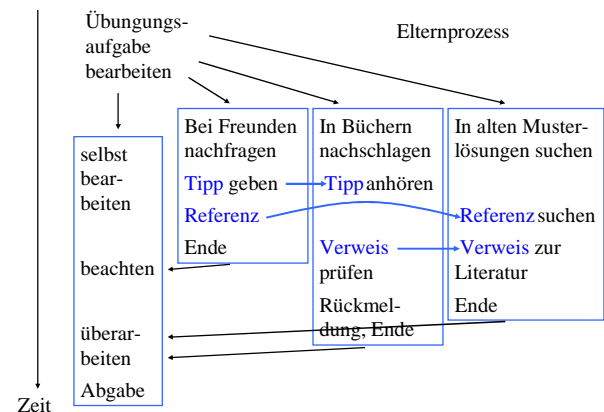
Prozess = Abarbeitung eines Algorithmus, wobei nur immer höchstens eine Stelle im Algorithmus aktiv ist.

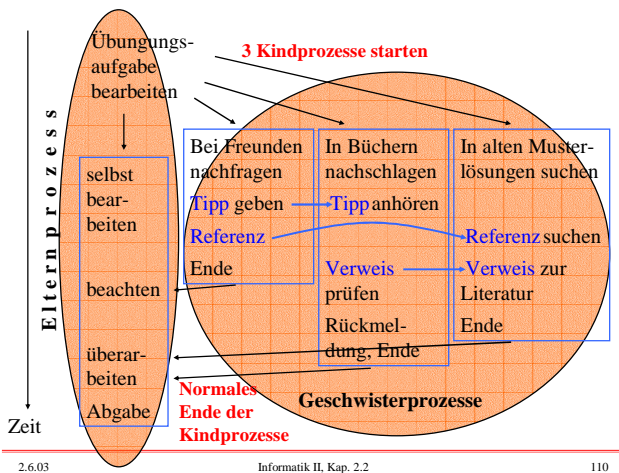
Einen Prozess kann man mit einem einzelnen Prozessor (Monoprozessor) abarbeiten. Er läuft sequentiell ab. Es können viele Prozesse gleichzeitig aktiv sein (Nebenläufigkeit).

Das Programmstück, das einen Prozess beschreibt, nennen wir *Prozesseinheit*. Schlüsselwort in Ada hierfür: task. Sie kann in einem Deklarationsteil vereinbart werden. Sie ist in die Spezifikation und die Implementierung (task body) geteilt. Der Nachrichtenaustausch zwischen Prozessen erfolgt implizit durch Kanäle (zwischen einem Entry-Aufruf und einem accept).

Eine Prozesseinheit wird in Ada gestartet, sobald ihre Deklaration im Programmablauf erreicht wird (impliziter Start); es gibt in Ada keine Anweisung der Form "starte Prozess X". Die Einheit, in der eine Prozesseinheit deklariert wird, heißt *Elternprozess*; andere im gleichen Deklarationsteil vereinbarte Prozesseinheiten heißen *Geschwisterprozesse*.

In einer Prozesseinheit kann es mehrere Stellen geben, an denen eine Synchronisation oder ein Datenaustausch erfolgen soll. Diese Stellen bezeichnet man als Entry-Schnittstellen; sie erhalten einen (Entry-) Namen und der Ablauf, der bei der Synchronisation erfolgen soll, wird in einer accept-Anweisung genau ausformuliert. Die accept-Anweisung trägt den Entrynamen; für eine Synchronisation wird sie wie eine Prozedur aufgerufen, erhält evtl. aktuelle Parameter und kann Werte über out-Variablen zurückgeben.

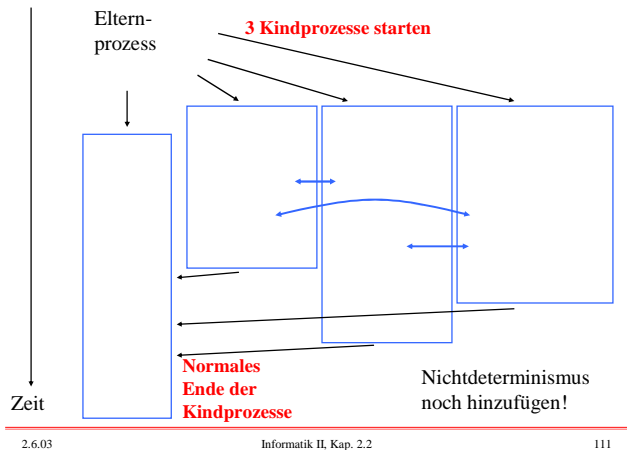




2.6.03

Informatik II, Kap. 2.2

110



2.6.03

Informatik II, Kap. 2.2

111

Wir benötigen also Sprachelemente für

Deklaration eines Prozesses: **task** (Spezifikation und Rumpf).

Starten eines Prozesses: Erfolgt implizit mit der Deklaration.

Normales Ende eines Prozesses: Erreichen von **end**
 (ein Elternprozess endet aber frühestens, wenn alle seine Kindprozesse beendet sind) oder eigene Beendigung mittels **terminate** (in select-Anweisung).

Datenaustausch zwischen den Prozessen:
entry für die Spezifikation des Austausches
 entry-Aufruf und **accept** für die Programmstellen

Gewaltsames Abbrechen eines Prozesses: **abort**

Nichtdeterministische Auswahl: **select ... or ... or ... end select**

Warten in nichtdeterministischen Alternativen: **delay [until]**

2.6.03

Informatik II, Kap. 2.2

112

2.2.4.2 a Syntax

```

single_task_declaration ::=
    task defining_identifier [is task_definition];
task_definition ::= {task_item} [private {task_item}]
    end [task_identifier]
task_item ::= entry_declaration | representation_clause
task_body ::= task_body defining_identifier is
    declarative_part
    begin
    handled_sequence_of_statements
    end [task_identifier];
  
```

2.6.03

Informatik II, Kap. 2.2

113

2.2.4.2 b Syntax (Zu "statement" siehe 1.1.6.6)

```

entry_declaration ::= entry defining_identifier
    [(discrete_subtype_definition)] parameter_profile;
(zu parameter_profile siehe 1.4.2.13)
entry_call_statement ::= entry_name [actual_parameter_part];
(actual_parameter_part beschreibt die aktuellen Parameter)
accept_statement ::=
    accept entry_direct_name [(entry_index)]
    parameter_profile
    [do handled_sequence_of_statements
    end [entry_identifier]];
(zu handled_sequence_of_statements siehe 1.4.1.1,
insbesondere ist diese Folge von Anweisungen nie leer)
entry_index ::= expression
  
```

2.6.03

Informatik II, Kap. 2.2

114

2.2.4.2 c Syntax

```

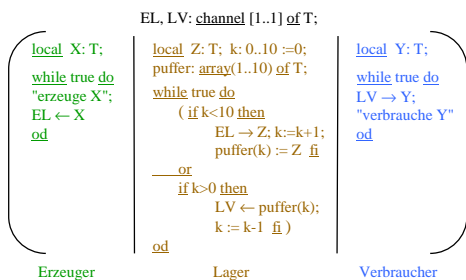
select_statement ::= selective_accept | timed_entry_call |
    conditional_entry_call | asynchronous_select
selective_accept ::= select [guard] select_alternative
    { or [guard] select_alternative }
    [else sequence_of_statements] end select;
guard ::= when condition =>
select_alternative ::= accept_alternative | delay_alternative |
    terminate_alternative
accept_alternative ::= accept_statement [sequence_of_statements]
delay_alternative ::= delay_statement [sequence_of_statements]
terminate_alternative ::= terminate;
delay_statement ::= delay_until_statement |
    delay_relative_statement
delay_until_statement ::= delay until delay_expression;
delay_relative_statement ::= delay delay_expression;
  
```

2.6.03

Informatik II, Kap. 2.2

115

2.2.4.3 Beispiel: Der Nachrichtenaustausch erfolgt in Ada über Kanäle. Wir übertragen daher Beispiel 2.2.2.13:



Das Programm wird zu `procedure ...` und die drei Prozesse werden zu `task Erzeuger`, `task Lager` und `task Verbraucher`.

2.6.03

Informatik II, Kap. 2.2

116

Die Kanäle spielen keine direkte Rolle. Ihre Namen können als Aufrufstellen im jeweiligen Prozess verwendet werden, sofern kein Namenskonflikt entsteht.

Die Sendeoperation "`←`" wird durch einen Entryaufruf zu der entsprechenden Stelle ersetzt, die Empfangsoperation "`→`" wird zu einer `accept`-Anweisung.

Verwendet man die gleichen Bezeichner, so wird aus `EL ← X`; der Aufruf `Lager.EL(X)`; und aus `EL → Z`; wird `accept EL(U: in Float) do Z := U; end`; (hier wurde willkürlich U als Name für den formalen Parameter der Aufrufstelle gewählt; dieser muss nun natürlich auch in der Spezifikation von `task Lager` für diesen entry benutzt werden).

Wir fügen für den Abbruch noch eine Boolesche Variable `Ende` und für die Interaktion eine Character-Variable `C` hinzu.

2.6.03

Informatik II, Kap. 2.2

117

```

procedure Erzeuger_Verbraucher is
  Ende: Boolean := false; C: Character;
  task Erzeuger;
  task Lager is
    entry EL(U: in Float);
  end;
  task Verbraucher;
  entry LV(W: in Float);
  < Die drei Taskrümpfe, siehe unten, hier einfügen >
begin
  Put("Erzeuger, Lager und Verbraucher sind aktiv.");
  while not Ende loop get(C); Ende := C='0'; end loop;
  Put("Erzeuger, Lager und Verbraucher enden nun.");
end Erzeuger_Verbraucher;
  
```

2.6.03

Informatik II, Kap. 2.2

118

Anstelle der Endlosschleife verwenden wir die Schleife mit der Bedingung "`not Ende`". Als "Produkt" nehmen wir reelle Zahlen.

```

task body Erzeuger is
  X: Float := 1.0;
  begin
  while not Ende loop
    X := Sin(X+1.0);
    Lager.EL(X);
  end loop;
end Erzeuger;
  
```

-- Schleife, um ständig Zahlen mit der
-- Variablen X neu zu erzeugen.
-- Entry-Aufruf: X wird ans Lager gesandt.
-- Schluss, falls "Ende" den Wert true erhält.

2.6.03

Informatik II, Kap. 2.2

119

```

task body Lager is
  -- fast wörtliche Übertragung nach Ada
  Z: Float; K: Integer range 0..10 := 0;
  Puffer: array(1..10) of Float;
  begin
  while not Ende loop
    select
      when K < 10 =>
        -- statt der Endlosschleife
        -- "(" wird zu select
        -- das if wird in ein when umgewandelt
        accept EL(U: in Float) do Z := U; end EL;
        K := K+1; Puffer(K) := Z;
      or
      when K > 0 =>
        -- das if wird in ein when umgewandelt
        Verbraucher.LV(Puffer(K));
        K := K-1;
    end select;
    -- ")" wird zu end select
  end loop;
end Lager;
  
```

-- Schluss, falls "Ende" den Wert true erhält.

2.6.03

Informatik II, Kap. 2.2

120

```

task body Verbraucher is
  Y: Float;
  begin
  while not Ende loop
    accept LV(W: in Float) do Y := W; end LV;
    < hier können Anweisungen zur Verarbeitung von x folgen >
  end loop;
end Verbraucher;
  
```

2.6.03

Informatik II, Kap. 2.2

121

2.2.4.4: Synchronisation und Kommunikation zweier Prozesse in Ada (1):

Die Interaktion zweier Prozesse erfolgt in Ada durch "Entry-Schnittstellen". Ein Entry ist eine durch `accept` bezeichnete Stelle in einer Prozesseinheit. Diese kann wie ein Unterprogramm von einem anderen Prozess aufgerufen werden, allerdings erfolgt keine Verzweigung des Programmablaufs zu dieser Stelle, sondern es wird eine Synchronisation vorbereitet: Der aufrufende Prozess wartet an der Entry-Aufrufstelle solange, bis der gerufene Prozess die Entry-Schnittstelle erreicht hat (= Synchronisation). Dann wird der hinter `accept` zwischen `do` und `end` stehende Programmteil, der wie ein Unterprogramm aufgebaut sein kann, ausgeführt. Erst danach trennen sich die beiden Prozesse wieder, d.h., sie fahren unabhängig von einander mit ihrer jeweils nächsten Anweisung fort. Diesen Vorgang bezeichnet man als *Rendezvous*.

Ein Entry kann von einem beliebigen anderen Prozess aufgerufen werden. Rufen mehrere Prozesse gleichzeitig die gleiche Entry-Schnittstelle, so werden diese Aufrufe nacheinander abgearbeitet, wobei der wechselseitige Ausschluss für den `do-end`-Programmteil garantiert wird, d.h., dieses Programmstück kann nicht durch weitere Entry-Aufrufe unterbrochen werden.

2.6.03

Informatik II, Kap. 2.2

122

Synchronisation und Kommunikation zweier Prozesse in Ada (2):

Trifft umgekehrt ein Prozess auf einen seiner Entry-Schnittstellen (also auf ein `accept`), so wartet er dort, bis ein anderer Prozess dieses Entry aufruft. Erfolgt dieser Aufruf, so wird der zwischen `do` und `end` stehende Programmteil ausgeführt (in dieser Zeit wartet der aufrufende Prozess nichtstehend) und anschließend trennen sich die beiden Prozesse wieder.

Man beachte, dass das Rendezvous mit dem Ende der `accept`-Anweisung endet. Danach folgende Anweisungen wirken sich nur auf die Wartezeit weiterer aufrufender Prozesse aus. In unserem Beispiel 2.2.4.3 endet das Rendezvous an der Entry-Schnittstelle EL in der Prozesseinheit Lager mit dem `end` nach dem `accept`. Die anschließend folgenden Anweisungen $K := K+1$; $\text{Puffer}(K) := Z$; gehören nicht mehr zum Rendezvous.

Hierdurch wird offensichtlich eine *Synchronisation* erreicht. Zugleich können durch den Entry-Aufruf, der aktuelle Parameter enthalten kann, Daten an den gerufenen Prozess übergeben werden und umgekehrt können am Ende des `do-end`-Programmstücks Daten zum aufrufenden Prozess zurückfließen, falls `out`-Variablen übergeben wurden. Dadurch wird eine *Kommunikation* zwischen den Prozessen realisiert.

2.6.03

Informatik II, Kap. 2.2

123

Synchronisation und Kommunikation zweier Prozesse in Ada (3):

Durch diesen Mechanismus kann es vorkommen, dass ein rufender Prozess oder ein Prozess, der für einen Aufruf bereit ist, ewig wartet. (Dann ist der Programmierer "selbst schuld", siehe aber unten (5).)

Man beachte die *Asymmetrie des Rendezvous-Mechanismus*: Während der rufende Prozess genau weiß, mit wem er eine Kommunikation durchführt (der Entry-Aufruf besteht ja aus dem Namen des Tasks, gefolgt von einem Punkt und dem Namen der Entry-Schnittstelle; fehlt der Name des Tasks, so ist stets der Elternprozess gemeint), kennt der gerufene Prozess den Namen seines Partners nicht. Um die rufenden Prozesse korrekt bedienen zu können, muss mit jeder Entry-Schnittstelle eine Warteschlange für rufende Prozesse verbunden sein: Ein Prozess, der auf ein `accept` trifft, führt ein Rendezvous mit dem ersten in der Warteschlange stehenden Prozess durch (oder wartet, bis ein Aufruf eintrifft).

In der Syntax gibt es die Möglichkeit, Entries als geheim einzustufen: `task_definition ::= ... [private {task_item}] ...`. Dies wird man dann tun, wenn diese Entry-Schnittstellen nur von Unterprozessen genutzt werden sollen.

2.6.03

Informatik II, Kap. 2.2

124

Synchronisation und Kommunikation zweier Prozesse in Ada (4):

Weiterhin lässt die Syntax bei der entry-Deklaration einen diskreten Untertyp [(discrete_subtype_definition)] und beim accept-statement einen [(entry_index)] zu. Diese beiden Varianten werden nur wichtig, wenn man "Entry-Familien" benutzt (dies sind viele Entries, die alle eine ähnliche Spezifikation besitzen).

Hinweise zur `accept`-Anweisung: Diese enthält keinen Deklarationsteil; man kann aber einen hinzufügen, indem man zwischen `do` und `end` Blöcke verwendet. Der `do-end`-Teil kann fehlen; in diesem Fall dient der Entry-Aufruf nur zur Synchronisation. Weiterhin kann es sinnvoll sein, die gleiche Entry-Schnittstelle an mehreren Stellen im Prozess zu platzieren; es sind daher zu einem Entry-Namen mehrere `accept`-Anweisungen erlaubt.

Wie üblich sind `goto`- und `exit`-Anweisungen, die von außen in eine `accept`-Anweisung gelangen oder eine `accept`-Anweisung verlassen, verboten. Allerdings ist ein `return` erlaubt, wodurch die umfassende Prozedur und zugleich das laufende Rendezvous beendet wird. Es gibt viele weitere Einschränkungen, siehe hierzu die Ada-Literatur.

2.6.03

Informatik II, Kap. 2.2

125

Synchronisation und Kommunikation zweier Prozesse in Ada (5):

Natürlich muss es Mechanismen geben, um ein "unverschuldetes" unendliches Warten zu beenden. In Ada sind dies für einen rufenden Prozess:

`timed_entry_call` der Form

```
select <Entry-Aufruf> <und evtl. weitere Anweisungen>
or <delay-Alternative> <und evtl. weitere Anweisungen> end select
```

Bedeutung: In der `delay`-Alternative kann man eine Zeit vorgeben; hat bis dahin das Rendezvous des "Entry-Aufruf" nicht begonnen, so wird der Aufruf abgebrochen und die hinter der `delay`-Alternative aufgeführten Anweisungen werden durchgeführt.

`conditional_entry_call` der Form

```
select <Entry-Aufruf> <und evtl. weitere Anweisungen>
or <Folge von Anweisungen, evtl. leer> end select
```

Bedeutung: Ist der gerufene Prozess nicht zum sofortigen Rendezvous bereit, so wird der `or`-Teil ausgeführt.

Ebenso gibt es Abbruchmöglichkeiten für den Prozess, der an einer Entry-Schnittstelle auf ein Rendezvous wartet.

2.6.03

Informatik II, Kap. 2.2

126

2.2.4.5: (Eingeschränkter) Nichtdeterminismus in Ada (1)

Es gibt die `select`-Anweisung mit den `or`-Alternativen. Diese Anweisung ist vorwiegend für die nichtdeterministische Behandlung von Entry-Aufrufen bzw. `accept`-Anweisungen vorgesehen, siehe Syntax 2.2.4.2 c.

In der Praxis ist Nichtdeterminismus (also willkürliches Verhalten von Prozessen an gewissen Stellen des Kontrollflusses) schwer kalkulierbar. Insbesondere können nicht alle Folgemöglichkeiten durchprobiert oder vorhergesehen werden. Entscheidet sich ein Prozess falsch, so kann er in Verklemmungen geraten.

Die `select`-Anweisung besitzt daher Varianten, um eine Entscheidung bzgl. der Kommunikation hinauszuzögern, bis bessere Informationen vorliegen, um eine Entscheidung, die nicht erfolgreich war, rückgängig zu machen oder durch eine andere zu ersetzen oder um eine eingeschlagene Alternative gewaltsam abzubrechen. Hierfür kann man einschränkende Bedingungen (sog. Wächter, "guards") vor die Alternativen setzen (`when ... => ...`), man kann die Auswahl davon abhängig machen, ob innerhalb einer Zeitspanne (`delay <Zeit in Sekunden>`) eine Alternative nicht erfolgreich war, man kann den eigenen Prozess zum Abbruch anbieten (`terminate`) oder man kann einen Prozess gewaltsam abbrechen (`abort`).

2.6.03

Informatik II, Kap. 2.2

127

(Eingeschränkter) Nichtdeterminismus in Ada (2)

Betrachte ein einfaches Beispiel:

```
select  
  accept Bildschirmausgabe (...) do ... end; ...  
or  
  when <Bildschirm an> => delay 300.0;  
  <rufe Bildschirmschoner> ...  
end select;
```

Liegt für 300 Sekunden kein Rendezvous mit dieser Entry-Schnittstelle "Bildschirmausgabe" vor, so wird die zweite Alternative ausgewählt.

Steht in einer Alternative das Sprachsymbol terminate, so wird dies einem "übergeordneten" Prozess, der zum Ende kommen möchte, als Möglichkeit angeboten, den laufenden Prozess an dieser Stelle zu beenden.

Bei den vielen Möglichkeiten muss man zwischen der rufenden (aktiver Prozess) und der gerufenen Seite (passiver Prozess) unterscheiden, für die ein Sprachelement unterschiedliche Bedeutung haben kann. Einzelheiten siehe Ada-Literatur und zum Teil im Programmierkurs.