Ada 95

Tasking

Gottlieb Bossert

Gliederung meines Vortrags

1. Nebenläufige Prozesse
2. Tasking im Ada 95 Sprachkern
3. Das Tasking Konzept
4. Protected Objects

Nebenläufige Prozesse

- Wofür benötigt man nebenläufige Prozesse?
  - Viele Aufgaben lassen sich effizient parallel ausführen
  - Systeme mit mehreren Prozessoren eignen sich gut für verteilte Aufgaben
  - Echtzeit Interaktionen mit der Realwelt müssen oft parallel bearbeitet werden

Tasking im Ada 95 Sprachkern

- Warum Tasking als Teil der Sprach-spezifikation?
  - ➔ Besser Portierbarkeit, da Betriebssystem-schnittstellen zwar oft vorhanden, aber nicht genormt
  - ➔ Komfortablere Verwendung, dadurch effizientere Anwendung

Tasking im Ada 95 Sprachkern

- Warum Tasking als Teil der Sprach-spezifikation?
  - ➔ Mehr Möglichkeiten zur Kontrolle
  - ➔ Bessere Zeitsteuerung
  - ➔ Größeres Optimierungspotential

Das Tasking Konzept

- Ada 95 bietet zwei unterschiedliche Konstrukte für nebenläufige Prozesse:
  - ***Tasks*** – Aktive Struktur zur parallelen Bearbeitung von Aufgaben
  - ***Protected Objects*** – Passive Konstruktion zur Datensyncronisation zwischen parallelen Prozessen

Task Definition

- Form ähnlich dem Paket: Unterteilung in Spezifikation und Body

```
task T is                    -- specification
   ...
end T;

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

Task Definition

- Ein Task ohne Interfaces sieht z.B. so aus:

```
task T;

task body T is
begin
   Do_Something;
end T;
```

Task Definition

- Ein Task kann auch in andere Strukturen eingebunden werden
- Dies kann auch ein anderer Task sein
- Der Zugriff auf private Bereiche ist hierdurch möglich

```
procedure Foo is

   task T;

   task body T is
   begin
      Do_Something;
   end T;

begin
   Do_Something_Else;
end Foo;
```

Task Definition

- Ein Task kann auch als Typ definiert werden
- Typ ist **limited** und Zuweisung ist Konstante
- Ein Task Typ kann Diskriminanten besitzen

```
task type T(I, J: Item);

task body T is
begin
   Do_Something(I, J);
end T;
```

```
declare
   Test: T(A, B);
begin
   ...
```

Task Definition

- Auf einen Task Typ kann auch ein Zeiger deuten
- Wie bei anderen Zeigern sind Zuweisungs- und Vergleichsopera-toren gestattet

```
task type T(I, J: Item) is
   entry E(...);
end T;

...
```

```
declare
   type Ref_T is access T;
   TX: Ref_T := new T(A, B);
begin
   TX.E(...);

...
```

Interaktion zwischen Tasks

- Interaktion zwischen Tasks durch direkten Nachrichtenaustausch
- Dieser Mechanismus wird in Ada „Rendezvous“ genannt
- Beim Rendezvous wird der Einstiegspunkt (**entry** / **accept**) eines Tasks direkt aufgerufen

Einstiegspunkt (Entry / Accept)

```
task One is
   entry E(...);
end One;

task body One is
begin
   accept E(...) do
      Do_Something;
   end E;
end One;
```

```
task Two;

task body Two is
begin
   One.E(...);
end Two;
```

```
procedure Foo is
begin
   One.E(...);
end Foo;
```

Einstiegspunkt (Entry / Accept)

- Ein **entry** erlaubt Parameter wie eine Prozedur (**in** / **out** / **in out**)
- Der Aufrufende Task wird unterbrochen, bis die **entry** Anweisung abgearbeitet ist
- Einsprungspunkte werden abgewartet und squentiell abgearbeitet; ohne **loop** Schleife wird der Task danach beendet

Einstiegspunkt (Entry / Accept)

```
task T is
   entry Put(X: in Item);
   entry Get(X: out Item);
end T;

task body T is
   V: Item;
begin
   loop
      accept Put(X: in Item)
      do
         V := X;
      end Put;
      Modify(Thing);
      accept Get(X: out Item)
      do
         X := V;
      end Get;
   end loop;
end T;
```

Die „Select“ Anweisung

- Mittels der **select** Anweisung können alternative Einsprungpunkte / Dienste angeboten werden

```
loop
  select
    accept Put(X: in ...)
    do
      V := X;
    end Put;
  or
    accept Get(X: out ...)
    do
      X := V;
    end Get;
  end select;
end loop;
```

Die „Select“ Anweisung

- Erfolgen mehrere Aufrufe eines Einsprungpunkts, so werden die Aufrufe in die Warteschlange des Einsprungpunkts gestellt und sukzessive abgearbeitet
- Haben mehrere Einsprungpunkte eines Tasks Elemente in der Warteschlange wird eine willkürliche Auswahl getroffen

Die „Select“ Anweisung

- Im vorherigen Beispiel kann etwas ausgelesen werden, bevor es dem Task übergeben wurde oder während es übergeben wird... dies ist natürlich nicht sinnvoll!
- Mit **when** lassen sich Restriktionen für die **select** Anweisung festlegen

Die „Select“ Anweisung

```
Init: Boolean := False;
...
loop
   select
      accept Put(X: in ...)
      do
         V := X;
      end Put;
      Init := True;
```

```
   or
      when Init and
         Put'Count = 0 =>
      accept Get(X: out ...)
      do
         X := V;
      end Get;
   end select;
end loop;
```

Die „Delay“ Anweisung

- Mit der **delay** Anweisung lässt sich ein Task für eine angegebene Zeit schlafen legen

```
declare
   Minutes: constant Duration := 60.0
begin
   loop
      delay 5 * Minutes;
      Do_Something;
   end loop;
end;
```

Die „Delay“ Anweisung

- Das vorherige Beispiel ist nicht Optimal, da...
  - ... die **loop** Anweisung einen Overhead produziert
  - ... die **Do_Something** Anweisung eine bestimmte Zeit zur Ausführung benötigt
  - ... ein höher priorisierter Task uns warten lässt

Die „Delay Until“ Anweisung

- Aus diesen Gründen kommt es bei dem vorherigen Beispiel zu einer wachsenden Verzögerung
- Besser wäre daher eine geplante Ausführung zu einem festgelegten Zeitpunkt
- Dies lässt sich mit der **delay until** Anweisung erreichen

Die „Delay Until“ Anweisung

```
declare
   use Calendar;
   Interval: constant Duration := 5 * Minutes;
   Next_Time: Time := Start_Time;
begin
   loop
      delay until Next_Time;
      Do_Something;
      Next_Time := Next_Time + Interval;
   end loop;
end;
```

Erweiterungen für „Select“

- Wartezeit: Wenn eine bestimmte Zeit lang kein Einstiegspunkt aufgerufen wurde, wird ein alternativer Anweisungsblock ausgeführt

```
select
   accept Signal;
or
   delay 10 * Minutes;
   Do_Something;
end select;
```

Erweiterungen für „Select“

- Statt einer **or delay** Anweisung ist auch ein **else** Block möglich. Dieser wird bei nicht-aufruf der Einstiegspunkts sofort ausgeführt und ist mit **or delay 0.0** vergleichbar.

```
select
   accept Signal;
else
   Do_Something;
end select;
```

Erweiterungen für „Select“

- Das **select ... else** Konstrukt ist durchaus sinnvoll, da mit select auch eine Ausnahme für einen Einsprungspunktaufruf (*also auf Seite des Aufrufers*) definiert werden kann:

```
select
   Resource.Lock;                    -- entry call
else
   Put(„Fehler: Die Resource ist bereits in
         Verwendung“);
end select;
```

Erweiterungen für „Select“

- Auch **select ... or delay** lässt sich für eine Ausnahme zu einem Einsprungspunkaufruf (*auf Seite des Aufrufers*) verwenden:

```
select
   Resource.Lock;                                  -- entry call
or
   delay 1 * Minutes;
   Put(„Fehler: Die Resource war die letzte Minute
         bereits in Verwendung“);
end select;
```

Erweiterungen für „Select“

- Mittels „asynchronous transfer of control“ (ATC) können laufende Berechnungen auch untebrochen werden
- Der Zeitpunkt des Abbruchs kann mittels Zeitgeber (**delay** / **delay until**) oder einer erfolgreichen **entry** Rückgabe erfolgen

Erweiterungen für „Select“

- Beispiele für Zeit- und Ereignisgesteuerte ATC:

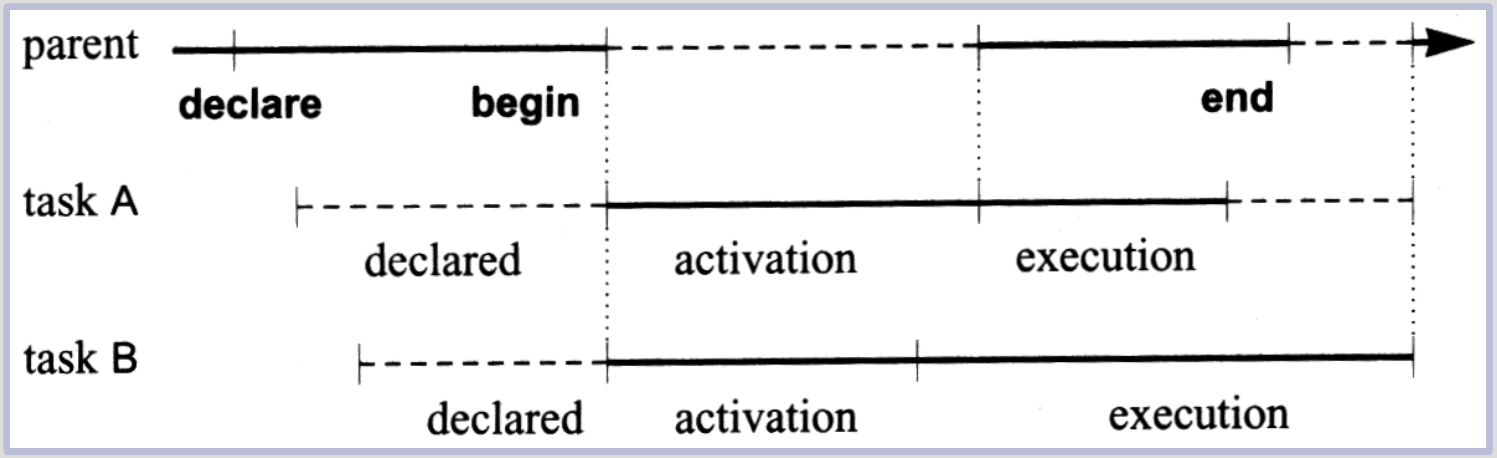
```
select
   delay 5 * Minutes;
   Put(„Zeitüber-
         schreitung“);
then abort
   Calculation.Exec(X);
end select;
```

```
select
   Trigger.Wait;
   Put(„Genauigkeit
         erreicht“);
then abort
   Calculation.Iterate(X);
end select;
```

Aktivierung von Tasks

- Tasks werden erst deklariert.
- Die Aktivierung und Ausführung der Tasks erfolgt nach der **begin** Anweisung.
- Der Block wird erst nach der Aktivierung aller Tasks ausgeführt.

```
declare
   task A;
   ...
   B: T;
begin
...
end;
```



aus Barnes, James: Programming in Ada 95, 2nd Edition, 1998, S. 436

Aktivierung von Tasks

- *Anmerkung:* Zeiger auf Tasks verhalten sich bei der Aktivierung anders:

  - Deutet ein Zeiger (**access type**) auf einen Task, so wird der Task schon bei der Evaluierung des Zeigers aktiviert.

  - Der Zeiger deutet dann auf den aktivierten Task

Terminierung von Tasks

- Tasks ohne Schleife werden nach ihrer Abarbeitung beendet
- Tasks mit Endlosschleife können eine **select ... or terminate** Anweisung enthalten, damit sie automatisch beendet werden können.
- Ohne die **terminate** Anweisung kann der aufrufende Block nicht beendet werden

Terminierung von Tasks

- Auf die **terminate** Alternative im **select** Block können keine weiteren Anweisungen folgen
- Sinnvoll ist es daher zusätzlich eine Stop Routine zu implementieren, damit der Task sauer beendet werden kann

Ausführen

```
select
   ...
or
   accept Stop do
      ...
   end Stop;
or
   terminate;
end select;
```

Terminierung von Tasks

- Tasks könne mit **abort** auch gewaltsam beendet werden
- Bei **abort** werden alle aufgerufenen Unterprogramme des Tasks ebenfalls abgebrochen
- Befindet sich der Task oder eines der Unterprogramme in einem Rendezvous treten Komplikationen auf

Terminierung von Tasks

- Wird der aufrufende Task in einem Rendezvous mit **abort** beendet, so wird der aufgerufenen Task nicht beeinflusst
- Wird der aufgerufene Task beendet, so wird eine **Tasking_Error** Eception propagiert.
- Dieses Vorgehen soll verhindern, dass kritische Serverprozesse Daten verlieren

Terminierung von Tasks

- Neben dem Rendezvous kommt es auch dann zu Komplikationen, wenn der zu beendende Task mit einem Protected Object kommunizert
- Prinzipiell sollte die Verwendung der **abort** Anweisung auf Ausnahmesituationen begrenzt werden
- (Der Zustand eine Tasks lässt sich abfragen)

Terminierung von Tasks

- Eine Ausnahmesituation für die **abort** Anweisung kann die **Exception**-Behandlung sein, da ein Block nur verlassen werden kann wenn alle seine Tasks beendet sind

Ausnahmebehandlung

- Die Ausnahme **Tasking_Error** wird erhoben...
  - ... bei Fehlern während der Akivierung eines Tasks
  - ... beim Terminieren unter bestimmten Bedingungen
  - ... wenn sich beim Beenden noch Tasks in einer Wartschlange befinden

Ausnahembehandlung

- Tritt eine Ausnahme bei einem Rendezvous in einer **accept** Anweisung auf und wird nicht intern behandelt, so wird diese Ausnahme an beide Tasks propagiert
- Dies kann z.B. dazu eingesetzt werden den Aufrufer über einen Fehler zu Informieren

Unzulänglichkeiten

- Tasks eigenen sich nicht besonders gut zur Syncronisation von Daten, da sie in diesem Falle...

  - ... als aktive Komponenten einen unnötigen Overhead produzieren, was zu einer schlechten Performance führt

  - ... keine sicheren Zugriffsbedingungen bieten (*z.B. Entry'Count*) {*Anm: Mit vorgelagerten „Proxy“-Tasks unter Performance- und Komfortverlust trotzdem realisierbar*}

Protected Objects

- Protected Objects wurden in Ada 95 neu eingeführt um diese Unzulänglichkeiten zu beheben
- Es handelt sich bei diesen um passive Strukturen
- Einem Protected Object ist kein Task zugeordent. Es wir vom Laufzeitsystem verwaltet.

Protected Object Definition

- Wie auch beim Task ähnelt die Form dem Paket, aber...

```
protected P is                    -- specification
   ...
end P;

protected body P is               -- body
   ...
end P;
```

Protected Object Definition

- ... Daten müssen im privaten Teil der Spezifikation definiert werden

```
protected P is                    -- specification
   ...
private
   V: Item := Initial_Value;
end P;
```

Protected Object Definition

- Ein Protected Object kann ebenso in andere Strukturen eingebunden werden
- Es kann auch als Typ deklariert werden
- Auf einen Protected Object Typ kann auch ein Zeiger deuten (**access type**)

Prozeduren, Funktionen, „Entry“ Anweisungen

- Ein Protected Object kann wie ein Paket Prozeduren und Funkionen enthalten
- Des weiteren können **entry** Anweisungen mit Barrierekonditionen definiert werden
- ➔ Alle drei Konstrukte verhalten sich jedoch anders als gewohnt

Prozeduren

- Prozeduren in Protected Objects besitzen im gegensatz zu Prozeduren in anderen Strukturen eine Exklusivität:
- Während des Zugriffs auf eine Prozedur ist das gesamte Protected Object gesperrt; weitere Zugriffsversuche werde in eine Wartschlange gestellt
- Eine Prozedur hat Schreibrechte

Funktionen

- Neben Prozeduren gibt es auch Funktionen im Protected Object
- Funktionen besitzen nur Leserechte
- Solange ein Protected Object nicht durch eine Prozedur oder ein Entry gesperrt ist können mehrere Funktionen parallel ausgeführt werden

Die „Entry“ Anweisung

- Oft ist es notwendig, dass Aufrufte nur unter bestimmten Bedingungen bearbeitet werden können
- **entry** Anweisungen in Protected Objects verlangen die Festlegung von Barriere-konditionen
- **entry** Anweisungen bieten wie auch Prozeduren Exklusivität und Warteschlangen

Die „Entry“ Anweisung

```
protected P is
   entry Put(X: in Item);
   entry Get(X: out Item);
private
   V: Item;
   Empty: Boolean := True;
end P;

protected body P is
   entry Put(X: in Item)
      when Empty is
   begin
      V := X;
      Empty := False;
   end Put;

   entry Get(X: out Item)
      when not Empty is
   begin
      X := V;
      Empty := True;
   end Get;
end P;
```

Barrierekonditionen

- Im Beispiel kann immer ein Element in P gelegt und dann wieder herausgenommen werden
- Bei jeder Aktion im Beispiel wird dazu die Variable für die Barrierekondition invertiert
- Alle Aktionen finden exklusiv statt, nach jeder Aktion werden alle Barrieren neu überprüft und wartende Anfragen bearbeitet

Die „Requeue“ Anweisung

- Manchmal ist es sinnvoll eine erfolgreiche Anfrage wieder in eine Wartschlange einzufügen.
- Dies lässt sich mit der **requeue** Anweisung realisieren
- Die **requeue** Anweisung kann nur aus meinem **entry** heraus aufgerufen werden

Die „Requeue“ Anweisung

- Ein Task dessen Anfrage mit **requeue** wieder in eine Warteschlange gelegt wurde kann nicht mit **abort** abgebrochen werden
- Soll dies möglich sein, so muss die Anfrage mit **requeue *Entry* with abort** behandelt werden
- Dies kann jedoch zu Komplikationen führen...

„Requeue“ Beispiel

```
protected Trigger is
   entry Wait;
   entry Signal;
private
   entry Reset;
   Go: Boolean := False;
end Trigger;

protected body Trigger is
   entry Wait when Go is
   begin
      null;
   end Wait;
   entry Signal when True is
   begin
      Go := True;
      requeue Reset;
   end Signal;

   entry Reset when
      Wait'Count = 0 is
   begin
      Go := False;
   end Reset;

end Trigger;
```

„Requeue“ Beispiel

- Würde man im vorherigen Beispiel den Abbruch eines Tasks in der Reset Warteschlange mittels **requeue *Reset* with abort** gestatten, so könnte dies dazu führen, dass der Trigger „kleben“ bleibt.

Anmerkungen

- Wenn möglich werden die Tasks in Ada auf die Taskingschnittstellen des Betriebs-systems abgebildet
- Für den Echtzeitbereich existiert ein Annex, welcher Erweiterungen zum Scheduling definiert
- Mit **pragma** Anweisungen lässt sich das Multitasking Verhalten von Ada anpassen (z.B. **pragma** Atomic)

Schlußfolgerungen

- Mit den gebotenen Tasking Fähigkeiten lassen sich relativ komfortabel nebenläufige Prozesse einsetzen
- Mit den Protected Objects lassen sich Daten zwischen parallelen Prozessen gut syncroniseren

Fragen

Vielen Dank für Eure Aufmerksamkeit!