Übung 3 – Lösungen

Aufgabe 1 *Prozesssteuerung in Ada mit Hilfe von Rendezvous*

Das "einfache eingebettete System" aus der Vorlesung (S. 2-41 ff) kann durch eine dritte Task ergänzt werden, die für das Entgegennehmen der gemessenen Druck und Temperaturwerte sowie deren Darstellung an der Konsole zuständig ist (vgl. Übung 2, Aufgabe 4).

Realisieren Sie das Package `IO` und die Task `Console` so, dass die sie mit den `Controller`-Tasks über eine Rendezvous kommunizieren.

Lsg.: (In Teilen von Manfred Koch)

Das Package IO benötigt keine Write-Prozeduren mehr für die Ausgabe auf der Konsole. Stattdessen bekommt die Konsole-Task entsprechende Eintrittspunkte.

```
with Data_Types; use Data_Types;
package IO is
  -- procedures for data exchange with the environment
  procedure Read(TR : out Temp_Reading); -- from DAC
  procedure Read(PR : out Pressure_Reading); -- from DAC
  procedure Write(HS : Heater_Setting);   -- to switch.
  procedure Write(PS : Pressure_Setting); -- to DAC
  -- entfällt:
  -- procedure Write(TR : Temp_Reading);     -- to console
  -- procedure Write(PR : Pressure_Reading); -- to console
end IO;
```

Die folgende Prozedur `Controller` definiert die drei Tasks `Temp_Controler`, `Pressure_Controler` und `Console` und startet sie nebenläufig. `Temp_Controler` und `Pressure_Controler` kommunizieren mit `Console` über ein Rendezvous.

```
with Data_Types; use Data_Types;
with IO; use IO;
with Control_Procedures; use Control_Procedures;
procedure Contoller is

  task Temp_Contoller;
  task Pressure_Contoller;

  task Console is
    entry Call_Temp(TR_temp : Temp_Reading); -- von Task Temp_Controller zu rufen
    entry Call_Pressure(PR_temp : Pressure_Reading);
                                             -- vom Task Pressure_Controller zu rufen
  end Console;


  task body Temp_Contoller is
    TR : Temp_Reading;
    HS : Heater_Setting;
  begin
    loop
      Read(TR);
      Temp_Convert(TR,HS);
      Write(HS);
      Console.Call_Temp(TR); -- sendet den Temperaturwert an Task Console
    end loop;
  end Temp_Controller;

  task body Pressure_Controller is
    PR : Pressure_Reading;
    PS : Pressure_Setting;
  begin
    loop
```

```
      Read(PR);
      Pressure_Convert(PR,PS);
      Write(PS);
      Console.Call_pressure(PR_temp); -- sendet den Druckwert an Task Console
    end loop;
  end Pressure_Controller;


  task body Console is
    PR : Pressure_Reading;
    TR : Temp_Reading;
  begin
    loop
      select
        accept Call_Temp(TR : Temp_Reading) do
          -- TR ausgeben
        end Call_Temp;
      or
        accept Call_Pressure(PR : Pressure_Reading) do
          -- PR ausgeben
        end Call_Pressure;
      end select;
    end loop;
  end Console;

begin
  null;       -- Die drei Tasks werden gestartet
end Controller;
```

Aufgabe 2 *Zufahrtskontrolle zu einem Parkplatz*

Lösen Sie das Parkplatzproblem (Vgl. Übung 1, Aufgabe 5 und Übung 2, Aufgabe 5) mit Hilfe dreier nebenläufiger Tasks: eine steuert die Einfahrtsschranke, eine steuert die Ausfahrtsschranke und eine steuert das Signal.

Skizzieren Sie eine Lösung in Ada, bei der sich Die Tasks über Rendezvous synchronisieren.

Hinweis: Im Gegensatz zur entsprechenden Aufgabe in Übung 2 muss die Anzahl der Fahrzeuge auf dem Parkplatz hier keine gemeinsame Ressource sein. Es genügt, wenn die Signal-Task diese Anzahl kennt und das Signal auf Rot setzt, wenn der Parkplatz voll ist.

Lsg.:

Die Packages `Data_Types` und `MyIO` sind dieselben wie in der Lösung zu Übung1, Aufgabe 5 1 (vgl. Übung1-lsg.pdf)

```
with Data_Types; use Data_Types;
with MyIO; use MyIO;
procedure Main is

   task Signal is
     entry einfahrt();
     entry ausfahrt();
   end;

   task Einfahrt;
   task body Einfahrt is
     EA : EAnfrage := False;
     EOpen : Einfahrt := Open;
     EClose : Einfahrt := Close;
     ED : EDurchfahrt := False;
```

```
  begin
    loop
      Read(EA);
      if (EA = True) then -- Wenn Anfrage Einfahrt
        Signal.einfahrt() ; -- Signal informieren, ggfs. warten
        Write(EOpen);   -- Einfahrt öffnen
        ED = True;      -- Durchfahrt beginnt
        while (ED = True) loop Read(ED); endloop; -- Durchfahren lassen
        Write(EClose);  -- Einfahrt schließen
      end if;
    end loop;
  end Einfahrt;

  task Ausfahrt;
  task body Ausfahrt is
    AA : AAnfrage  := False;
    AD : ADurchfahrt  := False;
    AOpen : Ausfahrt := Open;
    AClose : Ausfahrt := Close;
  begin
    loop
      Read(AA);
      if (AA = true) then -- Wenn Anfrage Ausfahrt
        Signal.ausfahrt() - Signal informieren
        Write(AOpen);  -- Ausfahrt öffnen
        AD = True;     -- Durchfahrt beginnt
        while (AD = True) loop Read(AD); endloop; -- Durchfahren lassen
        Write(AClose); -- Ausfahrt schließen
      end if;
    end loop;
  end Ausfahrt;

  task body Signal is
    PP: Integer := 0;
  begin
    write(Free); -- Signal auf grün
    loop
      select
        when PP < 50 =>
          accept einfahrt do
            PP := PP+1;
            if PP = 50 then write(Full); -- Signal auf Rot setzen
          end einfahrt
      or
         accept ausfahrt do
           if PP = 50 then write(Free); -- Signal auf Grün setzen
           PP := PP-1;
         end ausfahrt
      or
         terminate; -- Falls Einfahrt und Ausfahrt nicht mehr aktiv sind
                    -- Wird auch die Signal-Task beendet.
     end select;
    end loop

  end Signal;

begin   null;  end Main; -- Einfahrt, Ausfahrt und Signal werden gestartet
```

Aufgabe 3

Eine Server-Task habe die folgende Ada-Spezifikation:

```
task Server is

  entry Service_A;
  entry Service_B;
  entry Service_C;

end Server;
```

Schreiben Sie den Rumpf (body) der Task Server, so dass die folgenden Abläufe gewährleistet sind:

- Solange für alle drei Eintrittspunkte Client-Tasks Rendezvous-bereit sind, sollen diese in einer zyklischen Reihenfolge akzeptiert werden (A, B, C, A, B, C,…)
- Falls es für einige (1 oder 2) der Eintrittspunkte keine Rendezvous-bereiten Client-Tasks gibt, sollen die restlichen Eintrittspunkte (für die es Rendezvous-bereite Client-Tasks gibt) in zyklischer Reihenfolge abgearbeitet werden. Die Server-Task sollte nie blockiert werden, solange es Rendezvous-bereite Tasks gibt.
- Wenn es keine Rendezvous-bereiten Tasks gibt, soll die Server-Task kein "busy-waiting" durchführen, sondern blockiert werden, bis ein Eintrittspunkt gerufen wird.
- Wenn alle möglichen Clients terminiert haben, soll die Server-Task auch terminieren.

Hinweis: Für einen Eintrittspunkt E kann man mit dem Attribut `E'count` die Anzal der an diesem Eintrittspunkt Rendezvous-bereiten Tasks bestimmen (d.h. die Anzahl der Tasks, die den Eintrittspunkt aufgerufen haben und auf ein `accept` warten).

Lsg.: (nach [Burns & Wellings 2009])

```
1 task body SERVER is
2   type SERVICE is (A, B, C);
3   next : SERVICE := A;
4 begin
5   loop
6     select
7       when NEXT = A or
8         (next = B and SERVICE_B 'count = 0
9          and SERVICE_C 'count = 0) or
10        (next = C and SERVICE_C 'count = 0) =>
11           accept SERVICE_A do next := B; end ;
12    or
13      when NEXT = B or
14        (next = C and SERVICE_A 'count = 0
15         and SERVICE_C 'count = 0) or
16        (next = A and SERVICE_A 'count = 0) =>
17            accept SERVICE_B do next := C; end ;
1     or
2       when NEXT = C or
3         (next = A and SERVICE_A 'count = 0
4          and SERVICE_B ' count = 0) or
5         (next = B and SERVICE_B 'count = 0) =>
6            accept SERVICE_C do next := A; end ;
7     or
8       terminate ;
9     end select ;
10   end loop;
11 end SERVER ;
```