Übung 1 - Lösungen

Aufgabe 1 *Cobegin in Ada*

Zeigen Sie, wie eine Cobegin-Anweisung in Ada implementiert werden kann.

Gehen Sie z.B. davon aus, das `A` und `B` zwei Ada-Anweisungen sind, die nebenläufig ausgeführt werden sollen. Schreiben Sie dazu ein Ada-Programm, das in seinem Verhalten

```
cobegin A; B; coend;
```

entspricht.

Lsg.:

```
 1 procedure Cobegin is
 2
 3 task tA;
 4 task tB;
 5
 6 task body tA is begin A end ;
 7 task body tB is begin B end ;

 8 begin
 9   null;
10 end Cobegin ;
```

Aufgabe 2 *Fork & join in Ada*

Wie kann in Ada *Fork & Join* realisiert werden?

Lsg:

Fork kann einfach erreicht werden durch dynamische Task-Erzeugung. Z.B. das Pseudo-Code-Fragment von S. 4-19

```
function F return is ...;
procedure P;
  ...
  C:= fork F; -- F wird nebenläufig zu P gestartet
  ...
  J:= join C; -- P muss Terminierung von F abwarten
  ...
end P;
```

könnte nur mit dem Fork in Ada so programmiert werden

```
procedure Fork is
task type F_Type; task body F_Type is ...
type FPtr is access F_Type;  -- Typ FPtr ist "Zeiger auf" F_Type
F: FPtr;

task P;
task body P is
  begin
    ...
    F = new FPtr; -- "Fork F"
    ...
  end P;
begin  -- P wird automatisch gestartet, dadurch auch F;
  null
end Fork;
```

P muss allerdings am Ende nicht warten, bis F fertig ist, da der Master von F nicht P ist, sondern die Prozedur Fork (dort ist der Access-Typ definiert).

Ein Join innerhalb von P lässt sich mit folgenden Techniken realisieren:

- Über das Attribut `T'Terminate` kann abgefragt werden, ob eine Task `T` treminiert hat.
- An der Stelle des Joins kann in den Rumpf von P eine Warteschleife eingebaut werden, die wartet, solange F nicht terminiert hat, etwa so:

```
task body P is
  begin
    ...
    F = new FPtr; -- "Fork F"
    ...
    while not F.all'Terminated loop null end loop;
    ...
  end P;
```

Diese Lösung leidet aber unter den üblichen Nachteilen von "busy waitung". Mit Hilfe von Inter-Task-Kommunikation gibt es auch andere Realisierungsmöglichkeiten für ein "Join".

Aufgabe 3 *Task-Hierarchien in Ada*

Bestimmen Sie im folgenden Ada-Programm für jede Task ihre Eltern-Task und ihre Master-Task sowie, falls vorhanden, ihre Kinder-Tasks und die abhängigen Tasks.

```
procedure Main is
  procedure Hierarchy is
    task A;
    task type B;

    type Pb is access B;
    Pointerb : Pb;

    task body A is
      task C;
      task D;
      task body C is
      begin
        -- sequence of statements including
        Pointerb := new B;
      end C;
      task body D is
        Another_Pointerb : Pb;
      begin
        -- sequence of statements including
        Another_Pointerb := new B;
      end D;
    begin
      -- sequence of statements
    end A;

    task body B is
    begin
      -- sequence of statements
    end B;

  begin
    -- sequence of statements
  end Hierarchy;

begin
  -- sequence of statements
end Main;
```

Lsg:

Task	Parent	Children	Master	Dependant
A	Main	C, D	Hierarchy	C, D
Pointerb.all	C	-	Hierarchy	-
Another_Pointerb.all	D	-	Hierarchy	-
C	A	Pointerb.all	A	-
D	A	Another_Pointerb.all	A	-
Main	-	A	-	-
Hierarchy	-	-	-	A, Pointerb.all Another_Pointerb.all

Aufgabe 4 *Prozesssteuerung in C/Real-Time POSIX*

Implementieren Sie das "einfache eingebettete System" aus der Vorlesung (S. 2-41 ff) in C/Real-Time-POSIX.

Lsg: (*von Alexander Grünwald und Vitali Koschewoi*)

```
/*
 * Main.c
 *
 *  Created on: 22.10.2012
 */

#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

//Records
struct Temp_Reading     { int temp;};
struct Pressure_Reading { int press;};
enum   Heater_Setting   {OFF=0, ON};
struct Pressure_Setting {int p_sett;};

#define MIN_TEMP 10
#define MAX_TEMP 500

#define MIN_PRESS 0
#define MAX_PRESS 750

//Helper Method
int rand_between(int min, int max) {
      int random = (rand() % ((max + 1) - min)) + min;
      return random;
}

//Read/Write operations
void Read_Temp(struct Temp_Reading *tr) {
      //Generates random temperature between 10 and 500
      tr->temp = rand_between(MIN_TEMP, MAX_TEMP);
      printf("[Temp-Read]: %d \n", tr->temp);
}

void Read_Press(struct Pressure_Reading *pr) {
      pr->press = rand_between(MIN_PRESS, MAX_PRESS);
      printf("[Press-Read]: %d \n", pr->press);
}

void Write_HS(enum Heater_Setting *hs) {
      (*hs == ON)? printf("[Write-HS]: ON \n"): printf("[Write_HS]: OFF \n");
}

void Write_PS(struct Pressure_Setting * ps) {
      printf( "[Write_PS]: %d\n", ps->p_sett);
}

void Write_TR(struct Temp_Reading *tr) {
      printf("[Write_TR]: %d\n", tr->temp);
```

```
}

void Write_PR(struct Pressure_Reading *pr) {
     printf("[Write_PR]: %d\n", pr->press);
}

//Controlling operations
void Temp_Convert(struct Temp_Reading *tr, enum Heater_Setting *hs) {
     *hs = (tr->temp < 250)? ON: OFF;
}

void Pressure_Convert(struct Pressure_Reading *tr,
                      struct Pressure_Setting *pr) { }

//Thread defintion

void *Temp_Controller(void *ptr) {
     struct Temp_Reading TR;
     enum Heater_Setting HS = OFF;
     printf("Tempthread \n");
     while(1) {
          Read_Temp(&TR);
          Temp_Convert(&TR, &HS);
          Write_HS(&HS);
          Write_TR(&TR);
     }
}

void *Pressure_Controller(void *ptr) {
     struct Pressure_Reading PR;
     struct Pressure_Setting PS;
     printf("Pressurethread\n");
     while(1) {
          Read_Press(&PR);
          Pressure_Convert(&PR, &PS);
          Write_PS(&PS);
          Write_PR(&PR);
     }
}



int main() {
     pthread_attr_t attrTempController;
     pthread_attr_t attrPressController;

     pthread_t      TempControllerThread;
     pthread_t      PressControllerThread;

     //init with default values, exit if something goes wrong
     if( pthread_attr_init(&attrTempController)  != 0) exit(EXIT_FAILURE);
     if( pthread_attr_init(&attrPressController) != 0) exit(EXIT_FAILURE);

     //start execution of Temp_Controller and Pressure_Controller Threads,
     //exit if something goes wrong
     if( pthread_create(&TempControllerThread, &attrTempController,
                                               Temp_Controller, NULL) != 0)
     exit(EXIT_FAILURE);
```

```
      if( pthread_create(&PressControllerThread, &attrPressController,
                                          Pressure_Controller, NULL) != 0)
      exit(EXIT_FAILURE);

      pthread_join( TempControllerThread, NULL);
      pthread_join( PressControllerThread, NULL);
      return EXIT_SUCCESS;
}
```

Aufgabe 5 *Zufahrtskontrolle zu einem Parkplatz*

Skizzieren Sie mit Hilfe nebenläufiger Tasks die Struktur eines Programms zur Steuerung der Zufahrt auf einen Parkplatz. Der Parkplatz ist mit einer Einfahrtsschranke, einer Ausfahrtsschranke und einem Signal, das den Parkplatz als voll belegt oder frei anzeigt, ausgestattet.

Zeigen Sie, wie diese Struktur in Ada und in C/Real-Time POSIX realisiert werden kann.

Lsg.: (in Ada)

Entsprechend dem Stand in der Vorlesung (Kapitel 2) hier zunächst eine Lösung ohne Synchronisation der Tasks und ohne gegenseitigem Ausschluss (kann in der Praxis zu Problemen führen!).

(Die Code-Fragmente stammen teilweise aus einer Lösung von *Peter Hirsch und Thomas Kosak*)

Datentypen:

```
package Data_Types is
   type Parkplaetze is new Integer range 0..50;
   type Einfahrt is (Open, Close);
   type Ausfahrt is (Open, Close);
   type Signal is (Free, Full);
   type EAnfrage is new Boolean;
   type AAnfrage is new Boolean;
   type EDurchfahrt is new Boolean;
   type ADurchfahrt is new Boolean;
end Data_Types;
```

Ein-/Ausgabe (Zugriff auf Schranken, Parkplatz und Singal)

```
with Data_Types; use Data_Types;
package MyIO is
   procedure Read(EA: out EAnfrage);    -- Fahreug möchte einfahren?
   procedure Read(AA: out AAnfrage);    -- Fahrzeug möchte ausfahren?
   Procedure Read(ED: out EDurchfahrt); -- Fahrzeug in Einfahrtbereich?
   Procedure Read(AD: out ADurchfahrt); -- Fahrzeug in Ausfahrtbereich?
   procedure Write(E: Einfahrt);        -- Einfahrtschranke öffnen/schließen
   procedure Write(A: Ausfahrt);        -- Ausfahrtschranke öffenen/schließen
   procedure Write(S: Signal);          -- Signal setzen (Free/Full)
end MyIO;
```

Hauptprogramm mit 3 Tasks

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

   EA : EAnfrage := False;
   AA : AAnfrage  := False;
   ED : EDurchfahrt := False;
   AD : ADurchfahrt  := False;
   PP : Parkplaetze := 0;     -- Anzahl der belegten Parkplätze
   AOpen : Ausfahrt := Open;
   AClose : Ausfahrt := Close;
   EOpen : Einfahrt := Open;
   EClose : Einfahrt := Close;

   task Einfahrt;
   task body Einfahrt is
   begin
      loop
         Read(EA);
         if EA = True then -- Wenn Anfrage Einfahrt
           if PP<50 then      -- Wenn Parkplatz nicht voll
              Write(EOpen);   -- Einfahrt öffnen
              ED = True;      -- Durchfahrt beginnt
              PP = PP+1;      -- Ein Fahrzeug mehr auf dem Parkplatz
              while (ED = True) loop Read(ED); endloop; -- Durchfahren lassen
              Write(EClose);  -- Einfahrt schließen
            end if;
         end if;
      end loop;
   end Einfahrt;

   task Ausfahrt;
   task body Ausfahrt is
   begin
      loop
         Read(AA);
         if EA = true then -- Wenn Anfrage Ausfahrt
              Write(AOpen);  -- Ausfahrt öffnen
              AD = True;     -- Durchfahrt beginnt
              PP = PP-1;     -- Ein Fahrzeug weniger auf dem Parkplatz
              while (AD = True) loop Read(AD); endloop; -- Durchfahren lassen
              Write(AClose); -- Ausfahrt schließen
         end if;
      end loop;
   end Einfahrt;

   task Signal;
   task body Signal is
     loop
       if PP < 50 then write(Free);
       else write(Full);
       end if;
     end loop;
   end Signal;

begin   null;  end Main;
```