Übung 2 - Lösungen

Aufgabe 1 *Semaphore in C/Real-Time POSIX*

Programmieren Sie eine Lösung des Erzeuger-/Verbraucher-Problems mit Hilfe der Semaphore-Schnittstelle in C/Real-Time POSIX (vgl. Vorlesung S. 3-17 – 3-20).

Lsg.: (Von Max Maier und Manfred Koch)

```
//============================================================================
// Name        : A1_Semaphore_PCP.cpp
// Author      : Max Maier
// Version     :
// Copyright   : Your copyright notice
// Description : Don't forget to provide the "pthreadGC2.dll" libray
//                       (on windows) to the  linker
//============================================================================

#include <iostream>
#include <pthread.h>
#include <semaphore.h>
#include <time.h>

using namespace std;

// the threads
pthread_attr_t    ConsAttr, ProdAttr;
pthread_t         Cons, Prod;

// the Buffer
const int N = 3;
int Buffer[N];
int Top, Bottom;

// Semaphores
sem_t Mutex;
sem_t Item_Available;
sem_t Space_Available;

// append "object" (here only a number) to the buffer
void Append(int I){
      sem_wait(&Space_Available);
      sem_wait(&Mutex);
      Buffer[Top] = I;
      Top++;
      Top %= N;
      sem_post(&Mutex);
      sem_post(&Item_Available);
}

// take a "object" from the buffer
int Take(void){
      int retval;

      sem_wait(&Item_Available);
      sem_wait(&Mutex);
      retval = Buffer[Bottom];
      Bottom++;
      Bottom %= N;
      sem_post(&Mutex);
      sem_post(&Space_Available);
      return retval;
}
```

```
// consumer function: take an object and print the content on the console
void* Consumer(void* attributes){
	while(1){
		cout << Take() << endl;
	}
	return NULL;
}

// producer function: produce an object (here only an incrementing number)
// and insert into the buffer. - The timing functionality to keep the output
// readable is not really predictable
void* Producer(void* attributes){
	static time_t lastTime = time(NULL) ;
	static int counter = 0;
	while(1){
		if(difftime(time(NULL), lastTime) > .1){
			Append(counter);
			counter++;
			lastTime = time(NULL);
		}
	}
	return NULL;
}

// main fuction
int main() {
	// initialize the semaphores and other variables
	sem_init(&Mutex, 0, 1);
	sem_init(&Item_Available, 0, 0);
	sem_init(&Space_Available, 0, N);
	Top   = 0;
	Bottom	= 0;

	cout << "This is the PCP example:" << endl;

	// initialize the thread atributes
	if(pthread_attr_init(&ConsAttr) != 0){
		exit(EXIT_FAILURE);
	}
	if(pthread_attr_init(&ProdAttr) != 0){
		exit(EXIT_FAILURE);
	}
	// start consumer and producer thread
	if(pthread_create(&Cons, &ConsAttr, Consumer, NULL) != 0){
		exit(EXIT_FAILURE);
	}
	if(pthread_create(&Prod, &ProdAttr, Producer, NULL) != 0){
		exit(EXIT_FAILURE);
	}

	// wait to join one of the threads (here: the Producer), this should
	// never happen because both are infinite
	pthread_join(Prod, (void**)NULL);
	cout << "It has joined. This should not happen!!" << endl;
	return 0;
}
```

Aufgabe 2 *Semaphore in Ada*

Programmieren Sie eine Lösung des Problems der gemeinsamen Ressourcen (vgl. Vorlesung S. 3-22 – 3-25) in Ada mit Hilfe des auf S. 3-17 deklarierten Semaphor-Packages.

Lsg1.:

```
package Ressource is

  type Priority is (high, medium, low);
  procedure Allocate(P:Priority);
  function Deallocate return Boolean;

end Ressource;


with Semaphore_Package; use Semaphore_Package;

package body Ressource is

  mutex: Semaphore;  -- Gegenseitiger Ausschluss
  cond: array(Priority) of Semaphore; -- Bedingungssynchronisation nach Priorität
  waiting: array(Priority) of Integer := (0,0,0);
     -- Anzahl der wartenden Tasks für jede Prio
  busy: Boolean := False;

  procedure Allocate(P: Priority)
  begin

    Wait(mutex);

    if busy then
      waiting[P] := waiting[P] + 1;
      Signal(mutex); -- Mutex freigeben, damit Deallocate stattfinden kann
      Wait(cond[P]); -- Warten mit Prio
      -- Jetzt ist die Ressource Freigegeben
    end if;

    busy := True; -- Ressource belegen

    Signal(mutex);

  end Allocate;

  Function Deallocate: Boolean;
  begin

    Wait(mutex);

    if busy then
      busy = false; Ressource ist jetzt frei

      -- den wartenden Prozess mit der höchsten Priorität freigeben
      if waiting[high] > 0 then Signal(cond[high]);
                                waiting[high] := waiting[high] - 1;
      elseif waiting[medium] > 0 then Signal(cond[medium]);
                                      waiting[medium] := waiting[medium] - 1;
      elseif waiting[low] > 0 then Signal(cond[low]);
                                   waiting[low] := waiting[low] - 1;
      else Signal(mutex); -- kein wertender Prozess, Mutex freigeben
      end if;

      return True; -- Erfolgreich

    else

      return False; -- Fehler: Deallocate auf freie Ressource

    end if;

  end Deallocate;

end Ressource;
```

Lsg2: Auf das "Mitzählen" der auf eine Ressource wartenden Tasks (Array `waiting`) kann verzichtet werden, wenn man in der Lage ist, den Wert einer Semaphore-Variablen abzufragen (vgl. Funktion `sem_getvalue()` in der C/RealTime-POSIX Schnittstelle für Semaphore). Dazu müsste das `Semaphore_Package` in Ada etwa folgende Spezifikation haben:

```
package Semaphore_Package is

  type Semaphore(Initial: Natural := 1)
       is limited private;
  procedure Wait(S: in out Semaphore);
  procedure Signal(S: in out Semaphore);
  function GetValue(S: Semaphore) return Natural;

private

  type Semaphore is ...

end Semaphore_Package;
```

Damit ergibt sich folgende Implementierung des Packages `Ressource`:

```
with Semaphore_Package; use Semaphore_Package;

package body Ressource is

  mutex: Semaphore;  -- Gegenseitiger Ausschluss
  cond: array(Priority) of Semaphore; -- Bedingungssynchronisation nach Priorität
  busy: Boolean := False;

  procedure Allocate(P: Priority)
  begin

    Wait(mutex);

    if busy then
      Signal(mutex); -- Mutex freigeben, damit Deallocate stattfinden kann
      Wait(cond[P]); -- Warten mit Prio
      -- Jetzt ist die Ressource Freigegeben
    end if;

    busy := True; -- Ressource belegen

    Signal(mutex);

  end Allocate;

  Function Deallocate: Boolean;
  begin

    Wait(mutex);

    if busy then
      busy = false; Ressource ist jetzt frei

      -- den wartenden Prozess mit der höchsten Priorität freigeben
      -- cond[P] = 0 heißt, dass Prozesse mit Priorität P auf die Ressource warten
      if GetValue(cond[high]) = 0 then Signal(cond[high]);
      elseif GetValue(cond[medim]) = 0 then Signal(cond[medium]);
      elseif GetValue(cond[low]) = 0 then Signal(cond[low]);
      else Signal(mutex); -- kein wertender Prozess, Mutex freigeben
      end if;

      return True; -- Erfolgreich

    else

      return False; -- Fehler: Deallocate auf freie Ressource

    end if;

  end Deallocate;

end Ressource;
```

Aufgabe 3 *Geschützte Objekte in Ada: Implementierung eines Semaphore-Packages*

Implementieren Sie das auf S. 3-17 deklarierte Semaphor-Package von Ada

Hinweis: Implementieren Sie den Typ *Semaphore* im Privaten Teil der Package-Deklaration als geschütztes Objekt mit speziellen *Wait*- und *Signal*-Operationen als geschützten Operationen. Die öffentlichen *Wait*- und *Signal*-Operationen können dann mit Hilfe der geschützten Operationen implementiert werden.

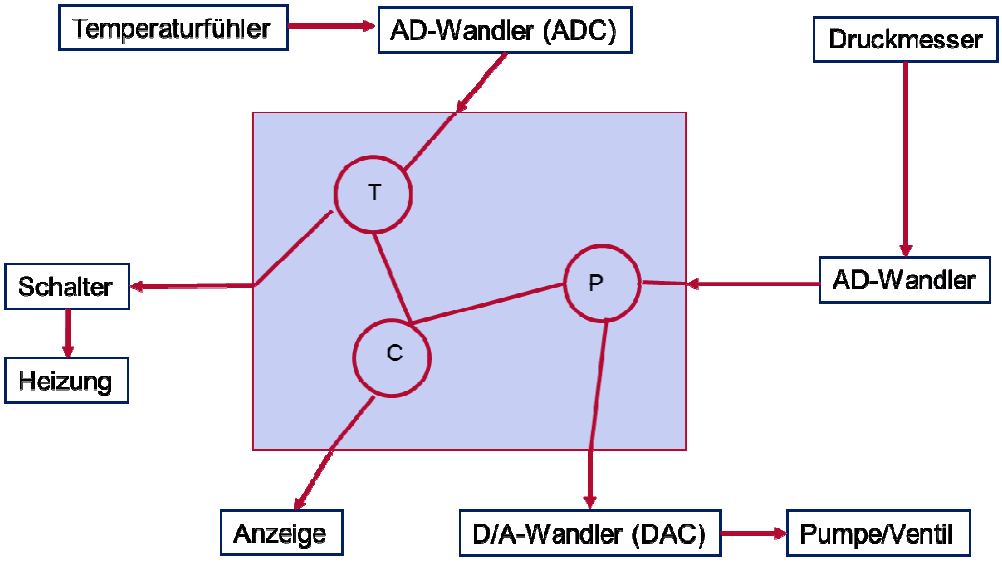
Lsg.: Das Semaphore-Package ist hier gleich erweitert um die Funktion `GetValue()` (vgl. Lösung 2 zu Aufgabe 2.)

```
package Semaphore_Package is
  type Semaphore(Initial : Natural := 1) is limited private;
  procedure Wait (S : in out Semaphore);
  procedure Signal (S : in out Semaphore);
  function GetValue(S:Semaphore): Natural;

private
  protected type Semaphore(Initial : Natural := 1) is
    entry Wait_Imp;
    procedure Signal_Imp;
    function GetValue_Imp return Natural;
  private
    Value : Natural := Initial;
  end Semaphore;
end Semaphore_Package;


package body Semaphore_Package is
  protected body Semaphore is
    entry Wait_Imp when Value > 0 is
    begin
      Value := Value - 1;
    end Wait_Imp;

    procedure Signal_Imp is
    begin
      Value := Value + 1;
    end Signal_Imp;

    function GetValue_Imp return Natural is
    begin
      return Value;
    end;
  end Semaphore;

  procedure Wait(S : in out Semaphore) is
  begin
    S.Wait_Imp;
  end Wait;

  procedure Signal(S : in out Semaphore) is
  begin
    S.Signal_Imp;
  end Signal;

  function getValue(S:Semaphore) return Natural is
  begin
    return S.GetValue;
  end;
end Semaphore_Package;
```

Aufgabe 4 *Prozesssteuerung in Ada mit Hilfe von geschützten Objekten*

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.

Die Kommunikation zwischen den Messprozessen T und P sowie dem Display-Prozess C kann über ein geschütztes Objekt *Console_Data* erfolgen. Die Tasks T und P schreiben ihre gemessenen Werte in *Console_Data*, die Task T liest immer dann die Werte aus *Console_Data*, wenn dort neue Werte geschrieben worden sind.



Skizzieren Sie eine Implementierung des Systems mit drei Tasks in Ada!

Hinweise:

Die Prozedur *Controller* mit den Tasks *Temp_Controller* und *Pressure_Controller* der nebenläufigen ADA-Lösung kann unverändert bleiben. Es ist lediglich das Package IO (S. 2-43) entsprechend zu implementieren.

```
with Data_Types; use Data_Types;
package IO is
  procedure Read(TR : out Temp_Reading); -- from ADC
  procedure Read(PR : out Pressure_Reading);
  procedure Write(HS : Heater_Setting);-- to switch
  procedure Write(PS : Pressure_Setting); -- to DAC
  procedure Write(TR : Temp_Reading);  -- to screen
  procedure Write(PR : Pressure_Reading);-- to screen
end IO;
```

Die Lese-/und Schreib-Methoden für die Sensoren (Druck und Temperatur) und Aktoren (Heizung und Ventil/Pumpe) können nach wie vor als gegeben vorausgesetzt werden.

Die letzten beiden Schreib-Methoden (auf den Bildschirm) müssen ersetzt werden durch ein Schreiben in das geschützte Objekt *Console_Data*.

Im Implementierungsteil (body) des Packages *IO* sind also das geschützte Objekt *Console_Data*, die letzten beiden Write-Methoden (schreiben nicht auf den Screen, sondern in *Console_Data*) sowie die Task *Console* zu implementieren, die *Console_Data* liest und die Werte auf dem Bildschirm ausgibt. Dabei ist zu realisieren, dass die Task *Console* nur Daten aus *Console_Data* liest, die neu geschrieben wurden, d.h. die er nicht schon gelesen hat. Zugriffe auf *Console_Data* sollten sich grundsätzlich gegenseitig ausschließen.

Lsg.:

Hilfspaket für Datentypen (vgl. S. 2-43)

```
package Data_Types is
  -- necessary type definitions
  type Temp_Reading is new Integer range 10..500;
  type Pressure_Reading is new Integer range 0..750;
  type Heater_Setting is (On, Off);
  type Pressure_Setting is new Integer range 0..9;
end Data_Types;
```

Hilfspaket für Steuerungs-Prozeduren– berechnen die Stellwerte aus Messwerten (vgl. S. 2-44)

```
with Data_Types; use Data_Types;
package Control_Procedures is
  -- procedures for converting a reading into
  -- an appropriate setting for output to
  procedure Temp_Convert(TR : Temp_Reading;
                         HS : out Heater_Setting);
  procedure Pressure_Convert(PR : Pressure_Reading;
                             PS : out Pressure_Setting);
end Control_Procedures;
```

Steuerungs-Tasks (Temp_Controller = T, Pressure_Controller = P)

```
with Data_Types; use Data_Types;
with IO; use IO;
with Control_Procedures; use Control_Procedures;

procedure Controller is
  task Temp_Controller;
  task Pressure_Controller;

  task body Temp_Controller is
    TR : Temp_Reading; HS : Heater_Setting;
  begin
    loop
      Read(TR);
      Temp_Convert(TR,HS);
      Write(HS);
      Write(TR);
    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);
      Write(PR);
    end loop;
  end Pressure_Controller;

begin
  null;   -- Temp_Controller and Pressure_Controller
          -- have started their executions
end Controller;
```

Paket mit Ein-/Ausgabe-Prozeduren (vgl. S. 2-43)

```
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
  procedure Write(TR : Temp_Reading);     -- to console
  procedure Write(PR : Pressure_Reading); -- to console
end IO;
```

Die Implementierung des Pakets erfolgt so, dass die Ausgabe von Temperatur- und Druckwerten nicht direkt auf die Konsole, sondern in ein geschütztes Objekt erfolgt. Das Geschützte Objekt verfügt über entsprechende `Write`-Prozeduren.

Gleichzeitig wird eine Task `Console` (= C) erzeugt, die über den Eintrittspunkt `Read` des geschützten Objekts `Console_Data` die immer dann (und nur dann) eingetragene Daten liest und auf der Konsole ausgibt, wenn neue Daten vorliegen. Dies wird durch einen entsprechenden Guard gesichert.

Für das Geschützte Objekt ist kein Datentyp definiert, sondern direkt ein Objekt.

```
package body IO is
  task Console;
  protected Console_Data is -- protected Object without type declaration
    procedure Write(R : Temp_Reading);
    procedure Write(R : Pressure_Reading);
    entry Read(TR : out Temp_Reading;
               PR : out Pressure_Reading);
  private
    Last_Temperature : Temp_Reading;
    Last_Pressure :Pressure_Reading;
    New_Reading : Boolean := False;
  end Console_Data;

  -- procedures for data exchange with the environment
  procedure Read(TR : out Temp_Reading) is separate; -- from DAC
  procedure Read(PR : out Pressure_Reading) is separate; -- from DAC
  procedure Write(HS : Heater_Setting) is separate;   -- to switch.
  procedure Write(PS : Pressure_Setting) is separate; -- to DAC

  task body Console is
    TR : Temp_Reading;
    PR : Pressure_Reading;
  begin
    loop
       ...
      Console_Data.Read(TR, PR);
      -- Display new readings
     end loop;
  end Console;

  protected body Console_Data is
    procedure Write(R : Temp_Reading) is
    begin
      Last_Temperature := R;
      New_Reading := True;
    end Write;
```

```
    procedure Write(R : Pressure_Reading) is
    begin
      Last_Pressure := R;
      New_Reading := True;
    end Write;

    entry Read(TR : out Temp_Reading; PR : out Pressure_Reading)
          when New_Reading is
    begin
      TR := Last_Temperature;
      PR := Last_Pressure;
      New_Reading := False;
    end Read;
  end Console_Data;

  procedure Write(TR : Temp_Reading) is
  begin
    Console_Data.Write(TR);
  end Write;     -- to screen

  procedure Write(PR : Pressure_Reading)  is
  begin
    Console_Data.Write(PR);
  end Write; -- to screen
end IO;
```

Aufgabe 5 *Zufahrtskontrolle zu einem Parkplatz*

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

Betrachten Sie die Anzahl der Fahrzeuge auf dem Parkplatz als gemeinsame Variable.

Beschreiben Sie die Synchronisationsbedingungen, die von den drei Prozessen eingehalten werden müssen.

Skizzieren Sie Lösungen des Synchronisationsproblems

(a) mit Hilfe von Mutexen und Bedingungsvariablen in C/Real-Time POSIX und

(b) mit Hilfe von geschützten Objekten in Ada.

Lsg:

Das Problem der Lösung zu Übung 1-5 (vgl. Übung1-lsg.pdf) besteht darin, dass die drei nebenläufigen Prozesse Ausfahrt, Einfahrt und Signal gleichzeitig auf die Variable PP (= Anzahl der belegten Prakpätze) zugreifen (Einfahrt inkrementiert, Ausfahrt dekrementiert und Signal liest den Wert von PP). Dabei können aufgrund der nicht atomaren Ausführung einer Zuweisung Inkonsistenzen entstehen.

Die Variable PP wird jetzt als Geschütztes Objekt mit Inkrement, Dekrement- und Leseoperation implementiert. Die Inkrement-Operation ist ein Eintrittspunkt, der nur akzeptiert wird, wenn der Parkplatz nicht voll ist. In der Einfahrts-Task wird die Schranke nur geöffnet, wenn ein Inkrement erfolgreich war. Wenn der Parkplatz voll ist, wird die Einfahrts-Task so lange blockiert, bis ein Inkrement wider möglich ist (d.h. ein zwischenzeitliches Dekrement stattgefunden hat).

Ein weiterer Nachteil bei der Lösung zu Übung 1-5 ist, dass viele Tasks "busy-waiting"-Schleifen ausführen. Insbesondere die Signal-Task prüft kontinuierlich die Zahl der freien Parkplätze und setzt das Signal entsprechend. Es würde reichen, wenn das Signal dann neu gesetzt wird, wenn sich die Anzahl der Parkplätze signifikant verändert, d.h. dann wenn der letzte Parkplatz belegt wird und dann wenn dieser Zustand wieder verlassen wird.

Dies wird erreicht durch zwei weitere Eintrittspunkte *WaitUntilFull* bzw. *WaitUntilFree*, die akzeptiert werden, wenn der Parkplatz voll ist bzw. wenn noch mindestens ein Platz frei ist.

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

   protected type Parkplatz is

     entry Inc;
     procedure Dec;
     function GetNumber return Integer;

     -- Die folgenden Entries blockieren die aufrufende Task bis
     entry WaitUntilFull; -- Parkplatz voll ist
     entry WaitUntilFree; -- es mindestens einen freien Platz gibt

   private

     PP:Integer := 0; -- Zahl der Fahrzeuge auf dem Parkplatz

   end;

   protected body Parkplatz is

     entry Inc when PP < 50 is
     begin PP := PP + 1 end;

     procedure Dec is
     begin PP := PP-1; end;

     function GetNumber return Integer is
     begin return PP; end;

     entry WaitUntilFull when PP = 50 is
     begin end;

     entry WaitUntilFree when PP < 50 is
     begin end;

   end Parkplatz;


   EA : EAnfrage := False;
   AA : AAnfrage  := False;
   ED : EDurchfahrt := False;
   AD : ADurchfahrt  := False;

   PP : Parkplatz; -- Anzahl der belegten Parkplätze (geschütztes Objekt)

   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
           PP.Inc(); -- Zahl der belegten Plätze erhöhen, sobald möglich
           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
   begin
      loop
         Read(AA);
         if (AA = true) then -- Wenn Anfrage Ausfahrt
              PP.Dec();      -- Ein Fahrzeug weniger auf dem Parkplatz
              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 Einfahrt;

   task Signal;
   task body Signal is
     loop
       PP.WaitUntilFree;
       write(Free);
       PP.WaitUntilFull;
       write(Full);
     end loop;
   end Signal;

begin   null;  end Main;
```