Softwareentwicklung - Anwendungsentwicklung

Aufgabenblätter - 01.09.2021

Dipl.-Ing. Msc. Paul Panhofer BSc.[1*]

1 ZID, TU Wien, Taubstummengasse 11, 1040, Wien, Austria

Abstract:

MSC: paul.panhofer@gmail.com

Keywords:

.

.

*E-mail: paul.panhofer@tuwien.ac.at

1.) Aufgabenblatt - Parallele Programmierung

Kompetenzen

Thread: Threadsynchronisation, Semaphoren

1.) Beispiel - Threadsynchronisation (10.Punkte)

In einer Fabrik werden Werkstücke hergestellt. Die Bearbeitungsanlage besteht dabei aus einem **Kran** einer **Maschine A** und einer **Maschine B**.

Der Bearbeitungsprozess für Werkstücke durchläuft folgende Schritte:

Definition: Prozessablauf

- Ein Werkstück wird vom **Kran** von Lager1 zu **MaschineA** transportiert.
- Das Werkstück wird von **MaschineA** bearbeitet. Dazu wird die `Process()` Methode von **MaschineA** aufgerufen.
- Das Werkstück wird anschließend vom **Kran** von **MaschineA** zu **MaschineB** transportiert.
- Das Werkstück wird nun von **MaschineB** bearbeitet. Dazu wird die `Process()` Methode von **MaschineB** aufgerufen.
- Zuletzt wird das Werkstück vom **Kran** von **MaschineB** ins Lager2 transportiert.

Hinweis: Führen Sie **MaschineA**, **MaschineB** und den **Kran** jeweils in einem eigenen Thread aus. Synchronisiere Sie die einzelnen Threads um den geforderten Prozessablauf abzubilden.

Codebeispiel: Programm.cs

```
// ----------------------------------------
//  Prozessablauf:
// ----------------------------------------
// 1.Step: Crane->Move("Storage", "MachineA")
// 2.Step: MachineA->Process()
// 3.Step: Crane->Move("MachineA", "MachineB")
// 4.Step: MachineB->Process()
// 5.Step: Crane->Move("MachineB", "Storage")


// ----------------------------------------
//  Klasse: MachineA
// ----------------------------------------
public class MachineA {

     public void Run(){
         while(true){
             ...
             Process();
             ...
         }
     }

    private void Process(){
        Thread.Sleep(100);
        Console.WriteLine(
            "MachineA: finished work"
        );
    }
}

// ----------------------------------------
//  Klasse: MachineB
// ----------------------------------------
public class MachineB {

     public void Run(){
         while(true){
             ...
             Process();
             ...
         }
     }

    private void Process(){
        Thread.Sleep(150);
        Console.WriteLine(
            "MachineB: finished work"
        );
    }
}
```

```
53 // ---------------------------------------
54 //  Klasse: Crane
55 // ---------------------------------------
56 public class Crane {
57
58     public void Run(){
59          while(true){
60              ...
61              Move("Storage", "MachineA");
62              ...
63              Move("MachineA", "MachineB");
64              ...
65              Move("MachineB", "Storage");
66              ...
67          }
68      }
69
70     private void Move(
71         string from, string to
72     ){
73         Thread.Sleep(200);
74         Console.WriteLine(
75             $"moving from {from} to {to}"
76         );
77     }
78 }
```

.

❑

3.) Aufgabenblatt - Parallele Programmierung

Kompetenzen

Thread: Threadsynchronisation, Semaphoren

1.Beispiel) Semaphoren (10.Punkte)

Im Rahmen einer Mars Mission werden Dronen eingesetzt um Rohstoffe zu finden bzw. abzubauen. Die Marsmission umfasst dabei 4 Harvesterdronen und 2 Sentineldronen.

Simulieren Sie den im Prozessablauf beschriebenen Vorgang in der Main Methode. Synchronisieren Sie dazu die folgenden Threadtypen: `Sentinel, Harvester`

Definition: Prozessablauf

- Zu Beginn der Mars Mission werden 2 Sentineldronen ausgeschickt, um die Marsoberfläche nach Rohstoffen abzusuchen.

  `Sentinel->ScanningSurface`

- Findet die Drone Rohstoffe signalisiert sie einer Harvesterdronen (`Sentinel->Signal()`) die gefundenen Rohstoffe abzubauen.
- Die Sentineldrone wartet dabei an der Fundstelle bis ein Harvester ihr Signal bestätigt. (`Harvester->Acknowledge`) Danach fährt sie fort die Oberfläche abzusuchen.
- Harvesterdronen warten in der Marsbasis bis sie von einer Sentineldrone die Koordinaten betreffend eines Rohstoffundes übertragen bekommen.
- Erhält einer der Harvester ein Signal von einem Sentinel bestätigt er das Signal und fährt zur Fundstelle (`Harvester->Harvest()`) um die Rohstoffe abzubauen.
- Hat die Drone die Rohstoffe abgebaut bringt sie sie ins Lager zurück. (`Harvester->Store`)
- Im Lager können sich immer nur maximal 2 Harvester aufhalten.
- Nach der Einlagerung warten die Harvester wieder auf das Signal einer Sentineldrone.

Codebeispiel: Mars Mission

```
// ----------------------------------------
//  Prozessablauf
// ----------------------------------------
// Sentinel->ScanningSurface()
// Sentinel->Signal()
// Harvester->Acknowledge()
// Harvester->Harvest()
// Harvester->Store()

// ----------------------------------------
//  Sentinel.cs
// ----------------------------------------
public class Sentinel {

    public string Code { get; set; }

    public Sentinel(string code) {
        Code = code;
    }

    public void Run() {
        while(true) {
         ...
            ScanningSurface();
            ...
            Signal();
        }
    }

    private void ScanningSurface(){
        Console.WriteLine(
            $"{Code}: Scanning Surface"
        );

        Thread.Sleep(500);
    }

    private void Signal(){
        Thread.Sleep(800);
        Console.WriteLine(
              $"{Code}: Found raw material"
        );
    }



}
```

```
//-----------------------------------------
//  Harvester.cs
//-----------------------------------------
public class Harvester {

  public string Code { get; set; }

    public Harvester(string code) {
        Code = code;
    }

    public void Run(){
        while(true) {
            ...
            Acknowledge();
            ...
            Harvest();
            ...
            Store();
        }
    }

    private void Acknowledge(){
        Console.WriteLine($"{Code}:
            Acknowledging signal");
        Thread.Sleep(100);
    }
    private void Harvest(){
        Console.WriteLine($"{Code}: Harvesting
            resources");
        Thread.Sleep(1000);
    }
    private void Store(){
        Console.WriteLine($"{Code}: Storing
            resources");
        Thread.Sleep(200);
    }
}
```

.

❑

2.) Aufgabenblatt - Parallele Programmierung

Kompetenzen

Parallele Programmierung: Threadsynchronisation

1.) Beispiel - Semaphoren (10.Punkte)

Kebappeppi ist eine neue aufstrebende Fastfoodkette für den gehobenen Anspruch. Sie werden beauftragt die folgenden Prozesse mit Hilfe eines Programms zu synchronisieren.

Hinweis: Verwalten Sie die folgenden Objekte: `Customer`, `Cook`, `Cashier`

Definition: Kebappeppi

- Jede Filiale hat 3 Mitarbeiter: Ali, Abdul und Hakan.
- Abdul und Hakan sind Köche während Ali die Bestellungen der Kunden abrechnet.
- Abdul und Hakan warten bis Kunden Bestellung (**Customer->Order**) aufgegeben. Sobald sie eine Bestellung aufgenommen haben, bereiten sie die entsprechenden Gerichte zu. (**Cook->Cook**)
- Jeder der Köche nimmt dabei immer die gesamte Bestellung eines Kunden auf.
- Sobald die Gerichte einer Bestellung zubereitet sind, wird die Bestellung (**Customer->Pay**) von Ali verrechnet. Sobald der Kunde bezahlt hat, bekommt er von Ali den Kassenbeleg (**Cashier->Confirm**).
- Ali wartet solange keine Kunden an ihn treten. Er kann immer nur einen Kunden aufeinmal abrechnen.

Codebeispiel: kebappeppi.cs

```
1  // ----------------------------------------
2  //  Prozessablauf:
3  // ----------------------------------------
4  // 1.Step: Customer->Order
5  // 2.Step: Cook->Cook
6  // 3.Step: Customer->Pay
7  // 4.Step: Cashier->Confirm
8
9
10 // ----------------------------------------
11 //  Klasse: Customer
12 // ----------------------------------------
13 public class Customer {
14
15     public string Name { get; set; }
16
17     public Customer(string name) {
18         Name = name;
19     }
20
21     public void Run(){
22       ...
23         Order();
24         ...
25         Pay();
26         ...
27     }
28
29     private void Order(){
30         Console.WriteLine(
31             $"{Name}: ordering food"
32         );
33         Thread.Sleep(1917);
34     }
35
36     private void Pay(){
37         Console.WriteLine(
38             $"{Name}: paying order"
39         );
40         Thread.Sleep(1989);
41     }
42
43 }
```

```
// ----------------------------------------
//  Klasse: Cook
// ----------------------------------------
public class Cook {
    public string Name { get; set; }

    public Cook(string name) {
        Name = name;
    }

    public void Run(){
        while(true){
         ...
         PrepareMeal();
         ...
        }
    }

    public void PrepareMeal(){
        Console.WriteLine("cooking food");
        Thread.Sleep(1818);
    }

}

// ----------------------------------------
//  Klasse: Cashier
// ----------------------------------------
public class Cashier{
   public string Name { get; set; }

    public Cashier(string name) {
        Name = name;
    }

    public void Run(){
        while(true){
            ...
            Confirm();
            ...
        }
    }
    public void Confirm(){
        Console.WriteLine(
            $"{Name}: confirm payment"
        );
        Thread.Sleep(1818);
    }
}
```

.

❑

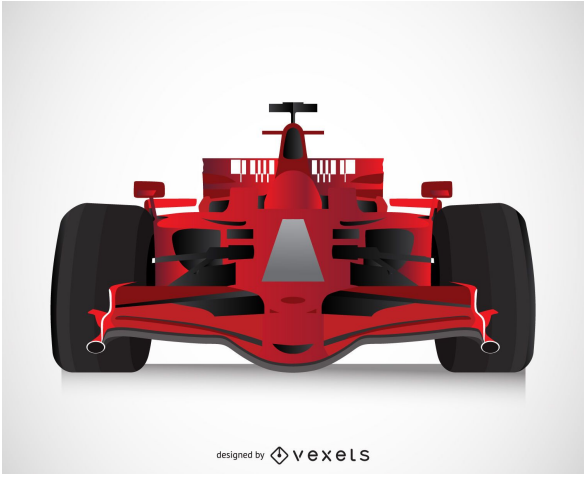
4.) Aufgabenblatt - Parallele Programmierung

Kompetenzen

Parallele Programmierung: Threadsynchronisation

1.) Beispiel - Semaphoren (10.Punkte)

Schreiben Sie ein Programm zur Simulation eines F1 Rennens. Synchronisieren Sie dazu die folgenden Prozesse: `Car, Race`.



Definition: racing

- An einem Rennen nehmen immer 5 Autos teil.
- Ein F1 Auto wartet auf seiner Startposition bis das Rennen (`Car->WaitForSignal`) gestartet wird.
- Der Streckenwart gibt das Signal zum Starten sobald alle 5 Autos ihre Startpositionen eingenommen haben (`Race->Start`).
- Daraufhin beginnt das Rennen (`Car->Race`).
- Während des Rennen führt jedes Auto einen Boxenstopp durch (`Car->TakingPitstop`). In der Boxenstrasse dürfen sich gleichzeitig nur 3 Autos aufhalten.
- Nach dem Boxenstopp fährt (`Car->Race`) der Fahrer das Rennen zu Ende.
- Das Rennen ist zu Ende `Race->End` sobald alle 5 Autos fertig sind.

Codebeispiel: Car.cs

```
// ----------------------------------------
// Prozessablauf
// ----------------------------------------
// 1.Step: Car->WaitForSignal();
// 2.Step: Race->Start();
// 3.Step: Car->Race();
// 4.Step: Car->Pitstop();
// 5.Step: Car->Race();
// 6.Step: Race->End();



// ----------------------------------------
// Klasse: Car
// ----------------------------------------
public class Car {
    public string Racer { get; set; }

    public Car(string racer) {
        Racer = racer;
    }

    public void Run(){
        //...
        WaitForSignal();
        //...
        Race();
        //...
        TakingPistop();
        //...
        Race();
        //...
    }
    private void WaitForSignal(){
        Console.WriteLine(
            $"{Racer}: Waiting for Start
                Signal"
        );
        Thread.Sleep(200);
    }
    private void Race(){
        Console.WriteLine($"{Racer}: Racing");
        Thread.Sleep(1500);
    }
    private void TakingPistop(){
        Console.WriteLine(
            $"{Racer}: Taking Pit stop"
         );
        Thread.Sleep(500);
    }
}
```

▸ Codebeispiel: Race.cs ▾

```
1  // ---------------------------------------
2  //  Klasse: F1Race
3  // ---------------------------------------
4  public class F1Race {
5
6     public void Run(){
7        // ...
8         Start();
9         // ...
10        End();
11        // ...
12    }
13
14    private void Start(){
15       Console.WriteLine("Starting Race");
16       Thread.Sleep(1000);
17    }
18
19    private void End(){
20       Console.WriteLine("Race finished");
21    }
22 }
```

❑

.

6.) Aufgabenblatt - Parallele Programmierung

Kompetenzen

Thread: Threadsynchronisation, Semaphoren

1.) Beispiel - Semaphoren (10.Punkte)

In einem Hafen wird die Ladung anlegender Schiffe gelöscht.

Definition: Hafen

- Im **Bereich** des Hafens dürfen sich zur selben Zeit maximal 5 Schiffe aufhalten
- Mit einer Glocke signalisiert (`Signal()`) der Hafenmeister einem der Schiff im Hafenbereich zu einer Anlegestelle zu fahren. Die Glocke wird erst geläutet, wenn eine der Anlegestellen im Hafen frei geworden ist.
- Im Hafen gibt es 3 Anlegestellen.
- Durch den Aufruf der `Unload()` Methode wird ein Schiff entladen. Wurde die `Unload()` Methode aufgerufen verläßt dass Schiff den Hafen. Der Hafenmeister kann dem nächsten Schiff signalisieren auszuladen.
- Ein Schiff kann nur entladen werden wenn es zuvor angelegt hat (die `Signal` Methode wurde bereits aufgerufen).
- Verläßt ein Schiff den Hafen, kann ein anderes Schiff einfahren.

Codebeispiel: Programm.cs

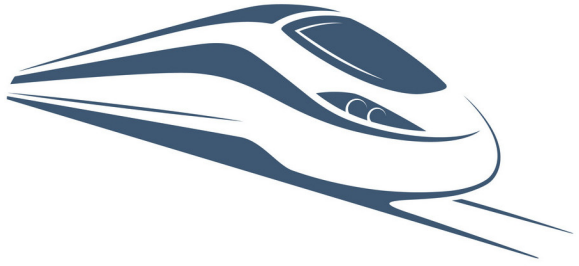
```
// ----------------------------------------
// Prozessablauf
// ----------------------------------------
// 1.Step: StagingArea->Signal()
// 2.Step: Ship->Unload()
// 3.Step: StagingArea->Signal()
// 4.Step: Ship->Unload()

// ----------------------------------------
// Klasse: StagingArea
// ----------------------------------------
public class StagingArea {

    public void Run(){
        while(true){
            ...
            Signal();
            ...
        }
    }

    private void Signal(){
        Console.WriteLine(
            "Stagingarea: signaling ship"
        );
    }

}

// ----------------------------------------
// Klasse: Ship
// ----------------------------------------
public class Ship {
    public string Id { get; set; }

    publis void Run(){
        ...
        Unload();
        ...
    }

    private void Unload(){
        Thread.Sleep(500);
        Console.WriteLine(
            $"unloading ship: {Id}"
        );
    }

}
```

2.) Beispiel - Threadsynchronisation (4.Punkte)

Zur Abwicklung des Zugverkehrs zwischen 3 Zugstationen soll eine Simulation geschrieben werden.

Definition: Zugsimulation: Shinkansen

- Die Zugstrecke führt von **Kyoto** über **Shizuoka** nach **Tokyo**.
- Im Bahnhof **Kyoto** gibt es 7 Bahnsteige, im Bahnhof **Shizuoka** 5 und in **Tokyo** 11.
- Zwischen **Kyoto** und **Shizuoka** sind jeweils 3 Gleise verlegt, zwischen **Shizuoka** und **Tokyo** jeweils 5.
- Auf einer **Bahnstrecke** (**Gleis**) zwischen 2 Bahnhöfen kann immer nur ein Zug fahren. Die Strecke kann jedoch in beiden Richtungen befahren werden.
- Züge können in einen Bahnhof einfahren solange ein **Bahnsteig** frei ist.
- Synchronisieren Sie die folgenden Züge auf möglichst effiziente Weise!



Codebeispiel: Programm.cs

```
// ----------------------------------------
// Klasse: ATrain
// ----------------------------------------
public abstract class ATrain {

    public static int indexer = 0;

    public static Semapore lock = new
        Semaphore(1,1);

    private int _id;

    public ATrain(){
       lock.WaitOne();
       this._id = ++indexer;
       lock.Release();
    }
    ...

// ----------------------------------------
// Klasse: ATrain
// ----------------------------------------
   public void EnterKyoto(){
       Console.WriteLine(
          $"Train {_id}: Entering Kyoto"
       );
       Thread.Sleep(500)
   }

   public void EnterShizuoka(){
       Console.WriteLine(
          $"Train {_id}: Entering Shizuoka"
       );
       Thread.Sleep(300);
   }

   public void EnterTokyo(){
       Console.WriteLine(
          $"Train {_id}: Entering Tokyo"
       );
       Thread.Sleep(700);
   }
}

// ----------------------------------------
// Klasse: TrainA
// ----------------------------------------
public class TrainA : ATrain{
    public void Run(){
       EnterKyoto();
       ...
       EnterShizuoka();
       ...
       EnterTokyo();
    }
}

// ----------------------------------------
// Klasse: TrainB
// ----------------------------------------
public class TrainB : ATrain{
    public void Run(){
       EnterTokyo();
       ...
       EnterShizuoka();
       ...
       EnterKyoto();
    }
}
```

❑

5.) Aufgabenblatt - Parallele Programmierung

Kompetenzen

Thread: Threadsynchronisation, Semaphoren

1.) Beispiel - Threadsynchronisation (10.Punkte)

Der Bearbeitungsprozess in der Fabrik wird umgestellt, sodaß statt des Krans ein Förderband installiert wird.

Definition: Bearbeitungsprozess

- Das Förderband wird verwendet um Werkstücke aus dem Lager zu **MaschineA** und **MaschineB** zu transportieren.

  Das Förderband wird dabei über die `Move()` Methode gesteuert.

- Der Aufruf der `Move()` Methode startet dabei folgenden Prozess:
  - ein Werkstück wird aus dem **Lager** zu **MaschineA** transportiert.
  - ein Werkstück wird von **MaschineA** zu **MaschineB** und
  - ein Werkstück von **MaschineB** ins **Lager** transportiert.

- Maschinen verarbeiten Werkstücke indem Sie die `Process()` Methode aufrufen.

- Eine Maschine darf die `Process()` Methode nur aufrufen, wenn ein unbearbeitetes Werkstück bereit liegt.

- Die Maschinen müssen in der Lage sein, unabhängig voneinander, parallel Werkstücke zu verarbeiten.

- Das Förderband darf nur bewegt werden, wenn beide Maschinen ihre Bearbeitung beendet haben.

- Beachten Sie dass beim Hochfahren des Systems keine Werktücke auf dem Förderband liegen, und somit die Maschinen solange warten müssen, bis ein Werkstück durch das Föderband zu ihnen transportiert worden sind.

Codebeispiel: Programm.cs

```
// ---------------------------------------
//  Prozessablauf
// ---------------------------------------
// 1.Step: ConveyorBelt->Move()
// 2.Step: MachineA->Process()

// 3.Step: ConveyorBelt->Move()

// 4.Step: MachineA->Process()
//         MachineB->Process()

// 5.Step: ConveyorBelt->Move()

// 6.Step: MachineA->Process()
//         MachineB->Process()


// ---------------------------------------
//  Klasse: MachineA, MachineB
// ---------------------------------------
public class MachineA {

     public void Run(){
         while(true){
             Process();
         }
     }

    private void Process(){
        Thread.Sleep(100);
        Console.Writeln(
             "MachineA: finished process"
        );
    }
}

public class MachineB {

     public void Run(){
         while(true){
             Process();
         }
     }

    private void Process(){
        Thread.Sleep(150);
        Console.Writeln(
            "MachineB: finished process"
        );
    }
}

public class ConveyorBelt {

     public void Run(){
         while(true){
             ...
             move();
             ...
         }
     }

}
```

❑

7.) Aufgabenblatt - Parallele Programmierung

Kompetenzen

Asynchrone Programmierung: Task

1.) Beispiel - Taskprogrammierung (10.Punkte)

Ihr Pate hat Sie gebeten für ihn eine Programm zur Simulation von Banküberfälle zu schreiben.

Definition: Heist

- Während eines Heists müssen mehrere Aufgaben durchgeführt werden.
- Jede Aufgabe (`EnterBank, HireCrew, ..`) soll in einem eigenem Task ausgeführt werden.
- Manche der Aufgaben können dabei gleichzeitig, andere hingegen müssen sequentiell durchgeführt werden.
- Reihenfolge der Tasks:
  - **1.Schritt:** `HIRING_CREW, GETTING_BANK_PLAN, BRIBE_EMPLOYEES, BUY_GETAWAY_CAR`
  - **2.Schritt:** `ENTER_BANK`
  - **3.Schritt:** `ROBBING_COUNTER_1, ROB_COUNTER_2, ROB_COUNTER_3`
  - **4.Schritt:** `LEAVE_BANK`
  - **5.Schritt:** `LOOSE_POLICE`

Hinweis: Verwenden sie folgende Methoden: `Result, Wait, WhenAll`

Codebeispiel: Heist.cs

```
// ----------------------------------------
// Klasse: Heist
// ----------------------------------------
public class Heist {

  /*
   * Implementieren Sie die EvaluateHeist
   * Methode
   */
  public void EvaluateHeist(){
      // 1.Protokollieren Sie wielange ein
      //   Heist dauert


      // 2. Protokolieren Sie in welcher
      //    Abfolge die Tasks ausgefuhrt
      //    werden
  }

   private string AggregateLog(string
       message) {
      StringBuilder str = new StringBuilder();

      str.Append(message);
      str.AppendLine();

      return str.ToString();
   }

   private string AggregateLog(string[]
       messages) {
      StringBuilder str = new
          StringBuilder();
      foreach (var message in messages) {
         str.Append(message);
         str.Append(" ");
      }
      str.AppendLine();

      return str.ToString();
   }

   private string BribeBankEmployee() {
      Thread.Sleep(300);
      return
          ELog.BRIBE_BANK_EMPLOYEE.ToString();
   }

   private string BuyGetawayCar() {
      Thread.Sleep(200);
      return ELog.BUY_GETAWAY_CAR.ToString();
   }
```

```
49
50      private string GetBankPlans() {
51          Thread.Sleep(200);
52          return ELog.GET_BANK_PLAN.ToString();
53      }
54
55      private string HireCrew() {
56          Thread.Sleep(400);
57          return ELog.HIRE_CREW.ToString();
58      }
59
60      private string EnterBank() {
61          Thread.Sleep(100);
62          return ELog.ENTER_BANK.ToString();
63      }
64
65      private string RobCounter1() {
66          Thread.Sleep(300);
67          return ELog.ROB_COUNTER_1.ToString();
68      }
69
70      private string RobCounter2() {
71          Thread.Sleep(300);
72          return ELog.ROB_COUNTER_2.ToString();
73      }
74
75      private string RobCounter3() {
76          Thread.Sleep(300);
77          return ELog.ROB_COUNTER_3.ToString();
78      }
79
80      private string LeaveBank() {
81          Thread.Sleep(120);
82          return ELog.LEAVE_BANK.ToString();
83      }
84
85      private string LoosePolice() {
86          Thread.Sleep(300);
87          return ELog.LOOSE_POLICE.ToString();
88      }
89  }
```

❑

.

6.) Aufgabenblatt - Parallele Programmierung

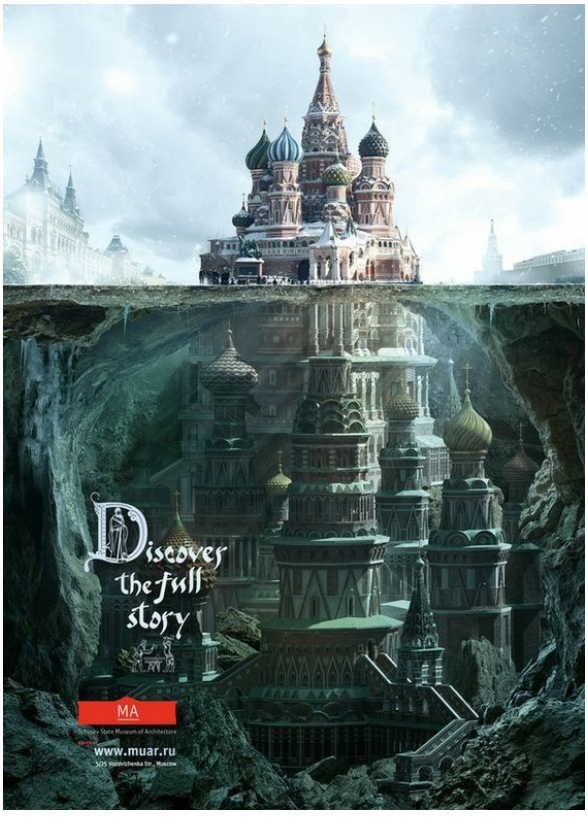
Kompetenzen

Parallele Programmierung: Threadsynchronisation

1.) Beispiel - Semaphoren (10.Punkte)

Die kommunistische Partei Graz hat Sie beauftragt ein Programm zur Abbildung der Prozesse innerhalb der zukünftigen Stadtregierung zu schreiben.

Hinweis: Verwalten Sie die folgenden Objekte: `Comrade`, `Maulsoleum`, `Task`



Definition: Aufgaben der Stadtregierung

- Die Zahl der Beamten (**Comrade**) die im Dienst der Stadtregierung stehen variiert je nach Bedarf.
- Zu Beginn des Arbeitstages begibt sich der Genosse in das Mausoleum, um sich für meherere Stunden dem Studium des kommunistischen Manifests zu widmen.

- Dazu entlehnt (**Mausoleum->Fetch**) er ein Exemplar des Manifests aus der Bibliothek des Mausoleums.

  Im Mousoleums Thread darf die `Fetch` Methode nur aufgerufen werden, wenn ein Comrade tatsächlich auf das kommunistische Manifest wartet.

- Im Mausoleum werden 3 Exemplare des Manifests aufbewahrt.

- Hat ein Genosse ein Manifest kann er sich in seine Studien vertiefen (**Comrade->Study**).

- Nach dem Studium des Manifests, warten Genossen auf die Zuweisung einer Aufgabe (**Task**).

- Im Laufe eines Arbeittages können Beamten mehrere Aufgaben zugeordnet werden. An einer Aufgabe kann immer nur ein Beamter arbeiten.

  Ein Beamter kann gleichzeitig immer nur eine Aufgabe bearbeiten.

- Mit der `Request` Methode (**Task->Request**) wird einem wartendem Genossen eine neue Aufgabe zugeordnet. Die Methode darf nur aufgerufen werden wenn es einen wartenden Genossen gibt.

- Durch den Aufruf der `Response` Methode kann ein Genosse die gestellte Aufgabe bearbeiten (**Comrade->Response**).

Codebeispiel: government.cs

```
// ----------------------------------------
// Klasse: Government
// ----------------------------------------
Ablauf:
    Mausoleum->Fetch
    Comrade->Study
    Task->Request
    Comrade->Response

public class Comrade {

    public void Run(){
        ...
        Study();
        ...
        while(true){
            ...
            Response();
          ...
        }
    }

    private void Study(){
        Console.WriteLine("comrade studying");
        Thread.Sleep(1917);
    }


    private void Response(){
        Console.WriteLine("comrade responding
            to request");
        Thread.Sleep(1989);
    }
}


public class Mausoleum {

    public void Run(){
          while(true){
               ...
               Fetch();
               ...
          }
    }

    public void Fetch(){
       Console.WriteLine("fetching holy
           book");
       Thread.Sleep(1818);
    }
}

public class Task{

    public void Run(){
        ...
        Request();
        ...
    }

    public void Request(){
       Console.WriteLine("requesting work");
       Thread.Sleep(2010);
    }
}
```

❑