Übung : Semaphore

Die Anlage besteht aus zwei Fertigungsmodulen („Linear“ und „Vertikal“). Diese arbeiten auf ein gemeinsames Transportband, das von einer eigenen Steuerung („Band“) gesteuert wird. Auf dem MES-System laufen im Multithreading (für diese Betrachtung identisch mit Multitasking) für jedes Modul unabhängig parallel Threads. Die Codestücke hierfür lauten so :

Linearmodul :

```
Private Sub linearmodul()
Do
      order = 8                 //Auftrag (8 = Öffne Schranke1)
      write_order()             //Auftrag schreiben
      start_module()            //Modul starten
      Do                        //warten, bis fertig
          read_state()
      Loop Until state = "ready
```

Vertikalmodul :

```
Private Sub vertikalmodul()
Do
      order = 9                 //Auftrag (9 = Öffne Schranke2)
      write_order()             //Auftrag schreiben
      start_module()            //Modul starten
      Do                        //warten, bis fertig
          read_state()
      Loop Until state = "ready"
```

Aufgaben :

Versuchen Sie, diesen Visual-Basic ähnlichen Code soweit zu verstehen, daß Sie die Steuerlogik für das Bandmodul durchschauen.

```
order = 9                    //Auftrag (9 = Öffne Schranke2)
write_order()                //Auftrag schreiben
start_module()               //Modul starten
Do                           //warten, bis fertig
       read_state()
Loop Until state = "ready"
```

--> Die Routine setzt die Variable order und beauftragt damit das Bandmodul, eine bestimme Tätigkeit auszuführen. Dazu wird die Order mit der Methode write_order() offenbar an die Band-SPS geschickt. Mit der folgenden Methode start_module() wird die Aktion der Band-SPS dann wahrscheinlich gestartet. Nun wartet das Programm, bis die angeordnete Aktion fertig ist. Dazu wird in einer Schleife dauernd die Methode read_state aufgerufen, die wohl von der Band-SPS abfragt, ob sie „ready“ ist, was wahrscheinlich bedeutet, daß die Aktion gelaufen ist.

Überlegen Sie, wie es hier zu einer Race-Condition oder einem Deadlock kommen könnte.

```
Private Sub linearmodul()
Do
      order = 8                 //Auftrag (8 = Öffne Schranke1)
      write_order()             //Auftrag schreiben

      //hier unterbricht der Scheduler.
      //Er wählt in der Folge irgendwann das vertikalmodul :

      order = 9                 //Auftrag (9 = Öffne Schranke2)
      write_order()             //Auftrag schreiben

      //hier unterbricht der Scheduler wieder.
      //Er wählt in der Folge irgendwann das linearmodul :

      start_module()            //Modul starten
      Do                        //warten, bis fertig
          read_state()
      Loop Until state = "ready"
```

--> Das Linearmodul ist gelaufen, hat die Order 8 gesetzt (irgendeine Mechanikaktion), aber es ist etwas anderes passiert, nämlich die Tätigkeit, die von Order=9 ausgelöst wird !!

Identifizieren Sie die critical sections !

```
Private Sub linearmodul()
Do

      //-----------------------------------------------------
      order = 8                //Auftrag (8 = Öffne Schranke1)
      write_order()            //Auftrag schreiben
      start_module()           //Modul starten
      //-----------------------------------------------------

      Do                      //warten, bis fertig
          read_state()
      Loop Until state = "ready
```

--> Der rote Teil ist die critical section. Es wäre fatal, hier den Scheduler zu deaktivieren (critical section wird dann nicht mehr unterbrochen), weil das zu kooperativem Multitasking führt !

Die Lösung ist, den Eintritt in die critical Section bei allen Threads, die diese „gemeinsame Resource“ order benutzen, durch einen Mutex abzusichern. Nur ein Thread darf zugreifen (vergleiche mit eingleisiger Bahnlinie !).

Nun setzen Sie geeignete Semaphore, um dies zu verhindern !

```
Dim BandOrderMutex As New System.Threading.Mutex

Private Sub linearmodul()
Do
   BandOrderMutex.WaitOne
      order = 8                    //Auftrag (8 = Öffne Schranke1)
      write_order()                //Auftrag schreiben
      start_module()               //Modul starten
   BandOrderMutex.ReleaseMutex()
      Do                           //warten, bis fertig
          read_state()
      Loop Until state = "ready
```

Praxisbezug :

Quelle : Beckhoff Informations-System

https://infosys.beckhoff.com

Lesen Sie den Text. Mit dem Wissen, das Sie in den ersten Paketen DVT erworben haben, sollte es Ihnen möglich sein, die wesentlichen Aussagen zu verstehen und somit in einer möglichen Anwendung zu entscheiden, wie sie die Werkzeuge nutzen.

Die Testfragen am Ende sollten Sie beantworten können !

Multitask-Datenzugriffs-Synchronisation in der SPS

Wenn von mehreren Tasks auf dieselben Daten zugegriffen wird, kann es je nach Task-/Echtzeitkonfiguration vorkommen, dass die Tasks gleichzeitig auf dieselben Daten zugreifen. Wenn die Daten dabei von mindestens einer der Tasks geschrieben werden, können die Daten während oder nach einer Änderung einen inkonsistenten Zustand haben. Um dies zu verhindern, müssen alle konkurrierenden Zugriffe synchronisiert werden, sodass zu einem Zeitpunkt nur von höchstens einer Task auf die gemeinsam genutzten Daten zugegriffen werden kann.

Zu diesen konkurrierenden Zugriffen aus mehreren Tasks, bei denen eine Synchronisation nötig ist, gehören beispielsweise die folgenden Fälle:

- Direkter Zugriff auf globale oder andere nicht-temporäre Variablen, beispielsweise mittels Operatoren
- Indirekter Zugriff auf globale oder andere nicht-temporäre Variablen, beispielsweise innerhalb von Funktionen, Methoden oder anderen POU-Aufrufen (z. B. besonders häufig, wenn eine Funktionsbausteininstanz global instanziiert ist)

Kurz: Wenn von mehreren Tasks auf dieselben Daten zugegriffen wird und bei mindestens einem dieser Zugriffe die Daten geschrieben werden, müssen alle lesenden und schreibenden Zugriffe synchronisiert werden. Dies gilt unabhängig davon, ob die Tasks auf einem oder mehreren CPU Kernen laufen.

⚠ WARNUNG

Inkonsistenzen und weitere Gefahren durch ungesicherten Datenzugriff

Werden konkurrierende Zugriffe nicht synchronisiert, so besteht die Gefahr eines inkonsistenten oder ungültigen Datensatzes. Je nachdem wie die Daten im weiteren Programmverlauf genutzt werden, kann dies ein Fehlverhalten des Programms, eine ungewünschte Achsbewegung oder auch den plötzlichen Programmstillstand zur Folge haben. Abhängig von der gesteuerten Anlage können Schäden an Anlage und Werkstücken entstehen oder Gesundheit und Leben von Personen gefährdet werden.

Um ein Gefühl für die Notwendigkeit der Zugriffs-Synchronisation zu erhalten, finden Sie bei den Beispielprogrammen zu den MUTEX-Verfahren jeweils Funktionstests mit entsprechender Erläuterung.

Synchronisationsmöglichkeiten

Zur Synchronisation der Zugriffe stehen u. a. die folgenden Möglichkeiten zur Verfügung:

- Mutex-Verfahren (TestAndSet, FB_IecCriticalSection) zum Absichern von kritischen Bereichen
  - Die Anzahl der kritischen Bereiche ist immer möglichst klein zu halten.
  - Die Länge der kritischen Bereiche ist kurz zu halten.
  - Bei vergleichsweise kurzen kritischen Bereichen empfiehlt sich meist die Verwendung von FB_IecCriticalSection.
  - Bei vergleichsweise langen kritischen Bereichen empfiehlt sich meist die Verwendung von TestAndSet.

Mutex-Verfahren (TestAndSet, FB_IecCriticalSection) zum Absichern von kritischen Bereichen

Bei der Verwendung von Mutex-Verfahren bzw. bei der Implementierung eines gegenseitigen Ausschlusses werden die Bereiche, in denen konkurrierende Zugriffe stattfinden, als kritische Bereiche (engl. Critical Sections) bezeichnet. Diese Bereiche können mithilfe der Funktion TestAndSet() oder des Funktionsbausteins FB_IecCriticalSection (beide aus der SPS-Bibliothek Tc2_System) synchronisiert werden, sodass die Bereiche unter gegenseitigen Ausschluss gestellt werden und zu einem Zeitpunkt jeweils nur eine Task auf die gemeinsam genutzten Daten zugreifen kann.

Das Betreten eines kritischen Bereichs kann von einer oder mehrerer Bedingungen abhängen. Zudem können verschiedene kritische Bereiche von jeweils unterschiedlichen Bedingungen abhängen.

Blockierung

- TestAndSet(): Mit der Funktion kann die Belegung eines kritischen Bereiches markiert und geprüft werden. Die Funktion blockiert jedoch nicht und es besteht die Möglichkeit, dass der Bereich in einem Zyklus nicht durchlaufen werden kann.
- FB_IecCriticalSection: Wenn eine weitere Task durch Aufruf der Methode Enter() einen bereits belegten kritischen Abschnitt betreten will, wird sie durch den TwinCAT Scheduler blockiert. **Die Task wird angehalten, bis der Bereich wieder freigegeben ist.**

Fragen dazu :

- Welche Werkzeuge gibt es bei Beckhoff-SPS zur Vermeidung von deadlocks und race conditions ?

  FB_lecCriticalSection
  TestAndSet

- Welche Methode entspricht der „Lehrbuch"-Variante Mutex ?

  FB_lecCriticalSection, weil die task auf „blocked" geht

- Welche Variante würden Sie benutzen, um die im Skript beschriebene Problematik der Steuerung des Transportbands mit seinen Stoppern aus verschiedenen Modulen zu lösen ?

  FB_lecCriticalSection, weil dann die Behandlung der Mutex-Situation vom BS gelöst wird.
  Wenn ich „TestAndSet" benutze, muß ich das Programm so schreiben, daß der Mutex immer wieder geprüft wird, damit es dann da weitergehen kann.