Concurrent Programming 1

Thomas Böhm, 5CHELI 2009/10

Threads

Multitasking ermöglicht das gleichzeitige Ausführen verschiedener Anwendungen (Prozesse) innerhalb eines BS. Als zweite Ebene der Parallelität ist es möglich, selbst einen Prozess in weitere parallele Abläufe zu splitten. Diese weiteren Aufteilungen innerhalb eines Prozesses werden als Threads bezeichnet. Threads teilen sich die Zeitscheibe, die dem Prozess zugeordnet wurde. Damit wird es einer Anwendung möglich, unterschiedliche Aufgaben innerhalb der Anwendung parallel auszuführen.

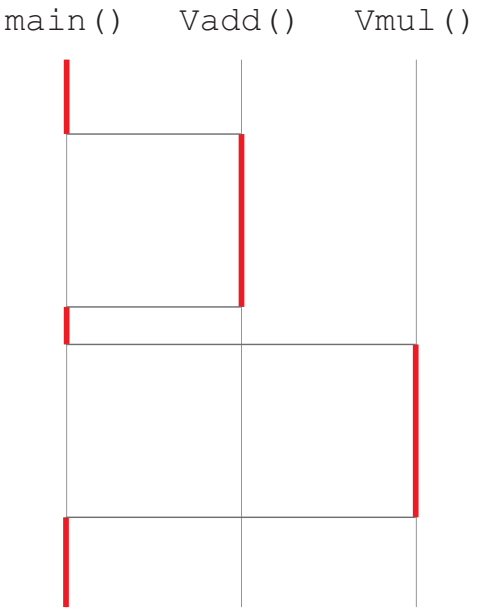
Beispiel:
Eine Anwendung mit einer komplexen und langwierigen Berechnung, könnte ohne die Existenz von Threads z.B. während der Berechnung nicht mehr auf Benutzereingaben reagieren. So ließe sich die Berechnung nicht abbrechen oder die Anwendung beenden. Abhilfe in so einem Fall wäre, die Berechnung und die Benutzerdialoge in jeweils separaten Threads durchzuführen. Somit müsste der Benutzer nicht erst auf das Ende der Berechnung warten und kann in der Zwischenzeit mit der Anwendung weiter arbeiten.

Sequentielle Abarbeitung

Wenn Funktionen sequentiell abgearbeitet werden, wird eine Funktion nach der anderen ausgeführt. Funktionen werden immer erst gestartet, nachdem die davor aufgerufene Funktion bereits beendet ist.

```
main()
{
      Vadd(C,A,B);

      Vmul(D,A,B);
}
```

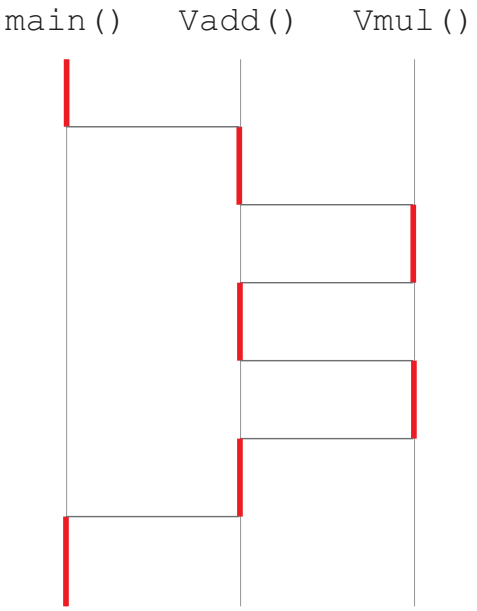


Parallel mit 1-Prozessor-Maschine

Werden Funktionen auf einer 1-Prozessor-Maschine in Threads aufgerufen, werden ihnen vom „Monitor“ sogenannte Timeslices zugewiesen. Nachdem das Timeslice für einen Thread abgelaufen ist, ist der nächste Thread an der Reihe (usw.). So wird zwischen den Threads hin- und hergewechselt (Threadswitching).

```
main()
{
	StartThread(Vadd());

	StartThread(Vmul());
}
```

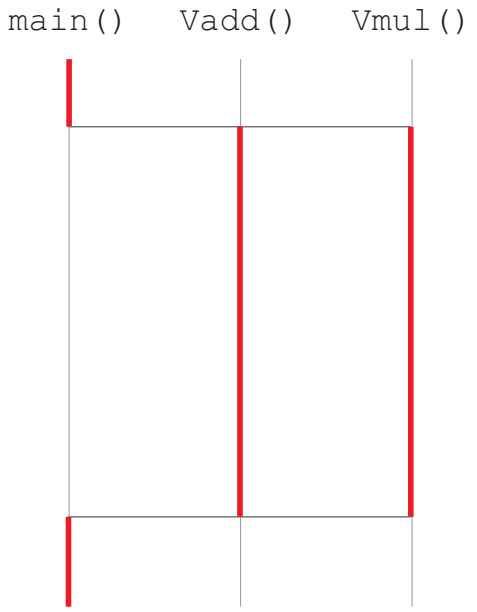


Parallel mit Multi-Prozessor-Maschine (Multicore)

Bei einer Multiprozessor-Maschine wird (wenn verfügbar) jedem Thread ein Prozessor zugewiesen, sodass die Threads parallel ausgeführt werden.

```
main()
{
	StartThread(Vadd());

	StartThread(Vmul());
}
```



Threads in C#

Beispielprogramm (*ThreadDemo1.cs*):

```
using System;
using System.Collections;
using System.Text;
using System.Threading;


namespace prj
{
  class ThreadDemo1
  {
    static void Main(string[] args)
    {
      Thread ta, tb;
      ta = new Thread(ThrFuncA); ta.Priority = ThreadPriority.Lowest;
      tb = new Thread(ThrFuncB); tb.Priority = ThreadPriority.Lowest;

      Console.WriteLine("Hit Enter to terminate ...");
      ta.Start(); tb.Start();
      Console.ReadLine();
      ta.Abort(); tb.Abort();
    }

    static void ThrFuncA()
    {
      int cnt = 0;
      while (true)
      {
        Console.WriteLine("A: {0}", cnt++);
        Thread.Sleep(1000);
      }
    }

    static void ThrFuncB()
    {
      int cnt = 0;
      while (true)
      {
        Console.WriteLine("B: {0}", cnt++);
        Thread.Sleep(1000);
      }
    }
  }
}
```

Erklärung:

Mit `Thread ta, tb;` werden 2 Threadobjekte (`ta`, `tb`) zur Kontrolle der Threads deklariert.

`ta = new Thread(ThrFuncA);` verbindet den Thread `ta` mit der Funtkion `ThrFuncA()`, die er ausführen soll.

`ta.Priority = ThreadPriority.Lowest;` setzt die Scheduling-Priorität von `ta` auf `Lowest`.

Mit `ta.Start();` wird `ta` gestartet, mit `ta.Abort();` wird er wieder beendet.

Selbes gilt für `tb`.

Mutex

Mutexe dienen nur der Verwaltung der gegenseitigen Ausschlusses von irgendeiner gemeinsam genutzten Ressource oder eines Codestücks. Sie sind einfach und effizient zu realisieren, was sie besonders in Thread-Paketen nützlich macht, die komplett im Benutzerraum realisiert sind.
Ein Mutex ist eine Variable, die zwei Zustände annehmen kann: nicht gesperrt oder gesperrt. Folglich wir nur 1 Bit benötigt, um sie darzustellen. In der Praxis wird häufig eine Ganzzahl verwendet, bei der 0 nicht gesperrt bedeutet und alle anderen Werte gesperrt bedeuten. Zwei Prozeduren werden mit Mutexen verwendet. Wenn ein Thread (oder ein Prozess) Zugang zu einer kritischen Region braucht, ruft er `mutex_lock` auf. Falls der Mutex gerade nicht gesperrt ist (was bedeutet, dass die kritische Region verfügbar ist), ist der Aufruf erfolgreich und dem aufrufenden Thread steht es frei, in die kritische Region einzutreten.
Falls der Mutex jedoch bereits gesperrt ist, wird der aufrufende Thread so lange gesperrt, bis der Thread in der kritischen Region fertig ist und `mutex_unlock` aufruft. Wenn mehrere Threads wegen des Mutex gesperrt sind, wird einer von ihnen per Zufall ausgewählt und ihm erlaubt, die Sperre zu erwerben.
Da Mutexe so einfach sind, können sie leicht im Benutzerraum realisiert werden, wenn eine **TSL**-Anweisung vorhanden ist. Der Code für `mutex_lock` und `mutex_unlock` für den Gebrauch mit Benutzeradressraum-Thread-Paketen wird unten gezeigt.

```
mutex_lock:
        TSL REGISTER, MUTEX          ;kopiere Mutex in Register, Mutex = 1
        CMP REGISTER, #0             ;war Mutex Null?
        JZE ok                       ;wenn Null, Mutex war belegt, Rücksprung
        CALL thread_yield            ;Mutex belegt; führe anderen Thread aus
        JMP mutex_lock               ;versuche es später
ok:     RET                          ;Rücksprung, in kritischen Bereich eingetreten

mutex_unlock:
        MOVE MUTEX, #0               ;speichere 0 im Mutex
        RET                          ;Rücksprung
```

Die **TSL**-Anweisung (**TSL** = **T**est and **S**et **L**ock) funktioniert wie folgt: Sie liest den Inhalt des Speicherwortes `MUTEX` ins Register `REGISTER` und speichert dann einen Wert ungleich Null an die Speicheradresse von `MUTEX`. Das Lesen und das Schreiben des Wortes sind garantiert unteilbare Operationen – kein anderer Prozess kann auf das Speicherwort zugreifen, bis die CPU die **TSL**-Anweisung ausführt, wird der Speicherbus gesperrt, um anderen CPUs den Zugriff auf den Speicher so lange zu verbieten, bis er fertig ist.
Um die **TSL**-Anweisung zu verwenden, wird die gemeinsam genutzte Variable `MUTEX` verwendet, um den Zugriff auf gemeinsam genutzten Speicher zu koordinieren. Wenn `MUTEX` 0 ist, könnte jeder Thread sie auf 1 setzen, indem er die **TSL**-Anweisung benutzt, und dann den gemeinsam genutzten Speicher lesen oder beschreiben. Wenn dies erledigt ist, setzt der Thread `MUTEX` mit Hilfe einer gewöhnlichen **MOVE**-Anweisung zurück auf 0.

Mutex in C#

Code für einen der beiden Threads welche mit wechselseitigem Ausschluss das Array befüllen (*Mutex1.cs*):

```
void ThrFuncA()
{
        int num2 = 0;
        while (true)
        {
                Monitor.Enter(mutex); // kein anderer Thread kann FillArray() jetzt aufrufen
                FillArray("A", num2++);
                Monitor.Exit(mutex); // FillArray() kann von anderen Threads wieder aufger.
                                        werden
        }
}
```

Die Codesequenzen `Monitor.Enter(mutex);` und `Monitor.Exit(mutex);` können kompakter und eleganter auch so formuliert werden:

```
void ThrFuncB()
{
        int num2 = 0;
        while (true)
        {
                lock (mutex)
                {
                        FillArray("B", num2++);
                }
        }
}
```