probucon Business Consulting
Your Business and Project Consulting Experts

Bernd Ua | probucon Business Consulting GmbH&Co KG

Thread Synchronisation

Vierte deutsche Coderage 2019

Vorstellung

- Bernd Ua
  - Geschäftsführer von probucon
  - Trainer, Consultant und nicht zuletzt Entwickler
  - zwei Jahrzehnte Erfahrung im Delphi Umfeld
  - Program Chair der Delphi Entwicklerkonferenz Ekon
  - Embarcadero MVP

Threads

- Parallel laufende Programmfäden innerhalb eines Prozesses
- Zeitscheibenverwaltung erfolgt analog Prozessen über das OS
  (pre-emptives Multitasking)
- Teilen sich im Gegensatz zu Prozessen einen Adressraum
- haben einen eigenen Stack und Registersatz

Delphi Threading Support

- Direkte Thread-Programmierung über Nachfahren der Klasse TThread
  (Vorlage in Objektgalerie)
- Indirekte Programmierung mittels PPL mit TTask/TFuture<T>
- Verwendung von Komponenten mit Threading-Support (z.B. Indy, HTTP-Komponenten)

UI und Threads

- Parallele Threads dürfen nicht direkt in den UI-Thread schreiben
- Gilt für Tasks ebenso wie für Threads
- Die einfache Lösung TThread.Synchronize
  - Als auskommentiertes Codefragment bereits in der Threadvorlage enthalten

Synchronize und Queue

- Synchronize stoppt den Thread und führt den Code im Kontext des Hauptthreads aus
  - Unperformant durch gestoppte Threads
- Queue stoppt den TThread nicht
  - dafür sind aber Thread-inhalte nicht mehr ohne Synchronisation zugreifbar
  - Queue-Aufrufe gehen verloren, wenn der Thread vor Abarbeitung beendet wird

Probleme

- Gefahren beim Arbeiten mit mehreren Threads
  - Unkoordinierter Zugriff auf gemeinsame Variablen
  - Gleichzeitiger Zugriff auf Ressourcen
  - Austausch von Daten mit begrenzter Gültigkeit
  - Verwendung statischer lokaler Variablen
  - Falsche Annahmen über den verwendeten Thread-Kontext

Synchronisation

- Synchronisation ist notwendig,
  - Für den Zugriff auf gemeinsame Daten aus mehreren Threads
  - um Code gegebenenfalls Thread-sicher zu machen
  - die Arbeit von Threads zu koordinieren

Warum überhaupt Synchronisation ?

- Threads können jederzeit zwischen Maschinencode Anweisungen unterbrochen werden
- SMP (symmetrisches Multiprocessing) mit mehreren Prozessoren und /oder Kernen Systeme ist heute Standard auf Server, Desktop und Mobilgeräten
- Durch den CPU-Cache für mehrere Kerne oder Prozessoren sind auch scheinbar atomare Operationen nicht mehr atomar

Synchronisation des CPU-Cache

- Folgende Operationen leiten eine Synchronisierung der CPU-Caches ein
  - Eintreten in und Verlassen von Critical Sections
  - Signalisieren von Synchronisationsobjekten
  - Wartefunktionen
  - Atomic(Interlocked)-Funktionen

Regeln für den Datenzugriff

- Datenänderung nur durch einen Thread zur gleichen Zeit
- während Datenänderung darf kein zweiter Thread die Daten lesen
- während des Lesens darf kein zweiter Thread Daten ändern
- während des Lesens dürfen andere Threads auch lesen

Möglichkeiten

- TThread.Synchronize
- TThread.Queue
- InterLocked-bzw Atomic Funktionen
- Synchronisationsobjekte
- Postmessage
- Events

Synchronisationsobjekte

- CriticalSections
- Mutexes/Semaphoren
- TMonitor
- TMultiReadSingleWriteSynchronizer (TMREWS)
- TSpinLock

Interlocked-Funktionen

- Die WinAPI bietet threadsichere Funktionen für einfache Integeroperationen an :
  - InterlockedIncrement, InterlockedDecrement
  - InterlockedExchange
  - InterlockedExchangeAdd
  - InterlockedCompareExchange u.a.

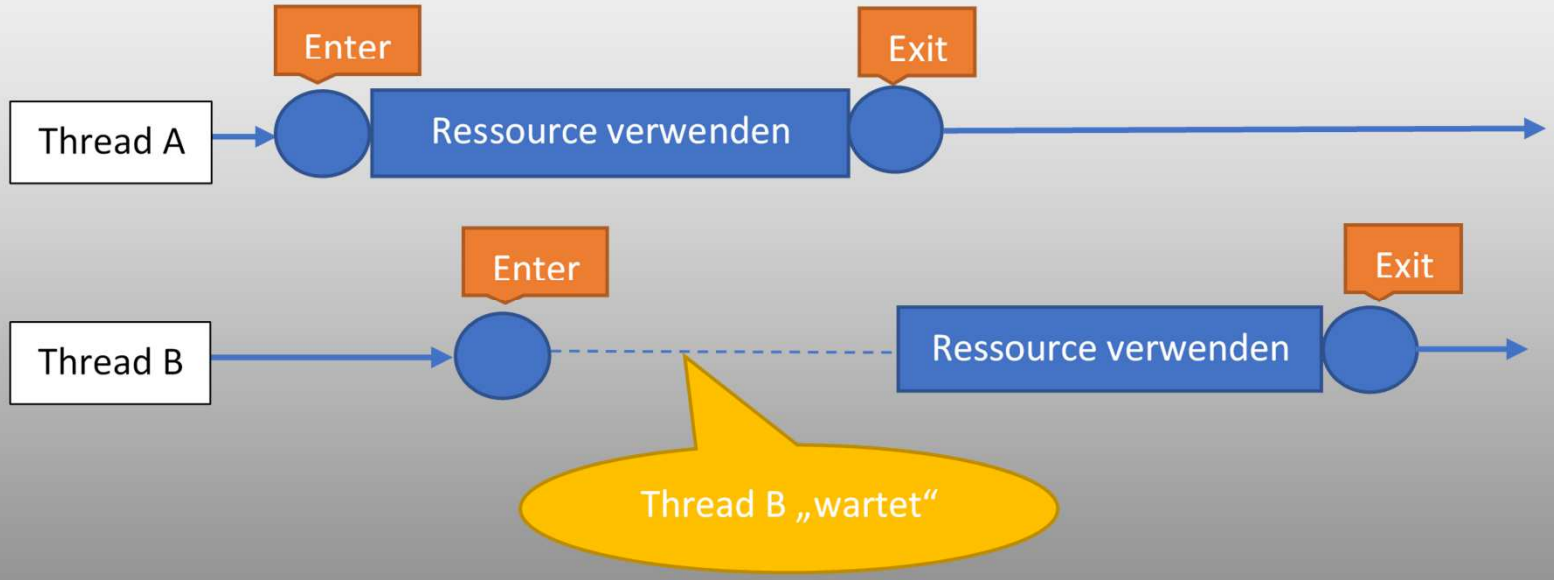
Interlocked versus Atomic

- In der Unit System finden sich jetzt analoge Funktionen
  - AtomicIncrement
  - AtomicDecrement
  - AtomicCmpExchange
  - AtomicExchange
- Stehen Crossplatform zur Verfügung !

Generelle Funktionsweise

- Synchronisationsobjekte funktionieren analog einer Sicherheitsschleuse, die nur eine Person betreten kann
- Alle Nutzer einer Ressource benutzen diese nur über den Umweg des Sync-Objektes
- Das Sync-Object stellt sicher, dass immer nur ein Nutzer das Objekt erhält

Synchronisationsobjekte schematisch



Was ist anders ?

- Die einzelnen Synchronisationsobjekte unterscheiden sich hinsichtlich
- der Methodennamen
- der Geschwindigkeit
- der Optionen (Timeout, TryEnter etc)

Kritische Abschnitte

- Kritische Sektionen verhindern, dass ein Codeabschnitt von mehreren Threads ausgeführt wird
- Kritische Sektionen sind nur innerhalb eines Prozesses gültig
- Nur ein Thread zur Zeit kann einen kritischen Abschnitt betreten

Verwendung

- Über TRTLCriticalSection direkt mit API-Funktionen
- In Form der Klasse TCriticalSection aus der Unit SyncObjs
- Verwendung von API und Delphi gleichwertig
- Bei TRTLCriticalSection den Pointer verwenden !
- TryEnterCriticalSection versucht den Lock zu erhalten und kehrt mit true zurück, wenn er erhalten wurde

System.TMonitor

- Verfügbar ab Delphi 2009
- Funktionsweise analog .net TMonitor
- Vorteil gegenüber der CriticalSection
  - TMonitor wird ein Objekt zur Synchronisation übergeben
  - Zusätzliche Funktionen
- „Nachteil“ gegenüber der Criticalsection
  - Bis XE2/Upd3 einige Bugs
  - Stark beschleunigt erst ab XE5

TMonitor

- enthält zusätzliche Funktionen (analog .net Monitorklasse)
  - Benachrichtigungsmechanismus mit Wait, Pulse, PulseAll
- verhindert, dass ein Codeabschnitt von mehreren Threads ausgeführt wird
- ist nur innerhalb eines Prozesses gültig
- Nur ein Thread zur Zeit kann den geschützten Abschnitt betreten

Mutexes

- Mutex = mutually exclusive access to shared ressource
- Mutexes sind prozessübergreifend gültig
- Mutexes können benannt werden
- Mit Hilfe des Namens lassen sich mehrere Handles auf ein Mutex erhalten

Mutexes verwenden

- Über die API mit CreateMutex/OpenMutex oder mittels Klasse SyncObjs.TMutex
- bInitialOwner steuert ob der erzeugende Thread Eigentümer wird
- lpName ist der optionale Name
  - Existiert ein Mutex dieses Namens wird von CreateMutex ein Handle auf den bestehenden Mutex zurückgeliefert
  - OpenMutex schlägt fehl, wenn der Mutex nicht bereits existiert

Mutexes (API) verwenden

- Threadsichere Code wird mit
- WaitForSingleObject und ReleaseMutex geschützt
- Die Kontrolle über einen Mutex erhält ein Thread durch Aufruf von WaitForSingleObject()
- ReleaseMutex gibt den Mutex frei

Waitfunktionen

- Die Wait-Funktionen veranlassen einen Thread, sich freiwillig so lange in den Ruhezustand zu begeben, bis ein bestimmtes Kernel-Objekt signalisiert wird
- Während der Thread wartet, braucht er keine Rechenzeit (er wird bei der Zeitscheibenzuteilung gar nicht erst berücksichtigt)
- das Warten ist also äußerst ressourcenschonend

WaitForSingleObject

- Wartet auf den Status Signaled eines Objekts (Mutex, Prozess, Thread)
- `function WaitForSingleObject(hHandle: THandle; dwMilliseconds: DWORD): DWORD; stdcall;`
- Die Wartezeit kann in ms angegeben werden oder unendlich sein (INFINITE)

WaitForMultipleObjects

- Wartet auf mehrere Ereignisse
  - function WaitForMultipleObjects(
    nCount: DWORD;
    lpHandles: PWOHandleArray;
    bWaitAll: BOOL;
    dwMilliseconds: DWORD): DWORD; stdcall;

MsgWaitForMultipleObjects

- Kehrt nicht nur bei Signalsierungen zurück sondern auch bei den angegebenen Nachrichten
  - function MsgWaitForMultipleObjects(nCount: DWORD; var pHandles; fWaitAll: BOOL; dwMilliseconds, dwWakeMask: DWORD): DWORD; stdcall;
- In dwMask können verschiedene Nachrichten angegeben werden

Rückgabewerte

- WAIT_ABANDONED es wurde auf einen Mutex gewartet dessen Besitzer-Thread beendet wurde ohne den Mutex freizugeben
- WAIT_OBJECT_0 das Objekt hat den Signalstatus erreicht
- WAIT_TIMEOUT die angegebene Wartezeit ist abgelaufen

Der Status Signaled

- Der Status Signaled unterschiedet sich je nach Objekt
  - Ein Prozess/Thread erreicht diesen Status wenn er beendet wird
  - Ein Mutex erreicht diesen Status, wenn er erzeugt wurde oder ein Thread den Mutex mit ReleaseMutex freigegeben hat
  - Ein Event erreicht diesen Status, wenn es ausgelöst wurde

Mutexes versus kritische Abschnitte

- CriticalSections sind deutlich schneller
  (ca 10 CPU Zyklen versus 600)
- Mutexes sind prozessübergreifend gültig
- Daher innerhalb einer Anwendung im Zweifelsfall kritische Abschnitte verwenden

Semaphoren

- Ähnlich einem Mutex
- Bieten zusätzlich Ressourcenzählung
- `function CreateSemaphore`
- `  (lpSemaphoreAttributes :PSecurityAttributes;`
- `   lInitialCount, lMaximumCount: Longint; lpName: PChar): THandle;stdcall;`
- Einsatzszenarien
  - Mehrere Threads berechnen Teilaufgaben und die Ausführung soll erst fortgesetzt werden, wenn alle x Threads soweit sind
  - Mehrere Threads warten auf Einträge

TMultiReadExclusiveWriteSynchronizer

- (TMREWS)
- Verwaltet mehrere lesende Zugriffe und blockierenden Schreibzugriff
- BeginRead/EndRead zum Lesen
- BeginWrite/EndWrite zum Schreiben
- Zum Beispiel verwendet von der VCL via IReadWriteSync in Forms (GlobalNamespace)

Verwendung

```
var
  global: TMultiReadExclusiveWriteSynchronizer;
  ...
  global.BeginRead;
  try
  // read something
  finally
    global.EndRead;
  end;
  ...
  global.BeginWrite ...
```

Neue Synchro-Objekte

- Unit SyncObjects seit Delphi XE beständig erweitert
- Größtenteils Crossplatform-fähige Erweiterungen
- Viel Klassen analog zu .net 4.0-Klassen gebaut

TInterlocked

- Kapselt threadsichere Variablenzugriff analog den Interlocked-Funktionen
- Nur Klassenmethoden und sealed
- Increment, Decrement, Exchange, CompareExchange etc

TSpinWait

- Implementiert „BusyWaiting“ – d.h. regelmäßiges nachschauen mit CPU-Zeit „verbrennen“
- Überladene Varianten der Klassenmethode SpinUntil warten auf einen anonymen Codeblock
- `class procedure SpinUntil(const ACondition: TFunc<Boolean>); overload; static;`
- Entspricht .NET 4.0 System.Threading.SpinWait

TSpinLock

- Verwendet keinen Lock zum Warten sondern verbrennt auch CPU Zeit
- Ist für kurze Wartezyklen effektiver als einen Lockimplementierung
- Methoden unter anderem
  Enter, Exit, TryEnter
- Aber Vorsicht: **nicht reentrant** !!!
- Entspricht .NET 4.0 System.Threading.SpinLock

TLightweightSemaphore

- Baut ein Semaphore mit TInterlocked/TMonitor nach
- Ist effektiver, wenn der Semaphorenzähler regelmäßig größer Null ist
- Entspricht .NET 4.0 System.Threading.SemaphoreSlim

Vergleich für das Beispiel

	Zeit (ms)	Bemerkungen
Keine Threads, seriell erhöhen und erniedrigen	30	
Threads, keine Synchronisation	15-20	
Atomic-Funktion	450	
Criticalsection (API)	1000	
TCriticalSection	1000	
Monitor	500	
Mutex (API)	76000	unbenannt
TMutex	76000	benannt
TSpinLock	400	

PostMessage

- Für einfache Ausgaben in den MainThread
- Nur unter Windows
- Nachrichten können verloren gehen
- Dafür einfach und schnell

Ereignisse für Threads

- Auch Threads kennen Ereignisse
- Eher Benachrichtigungsmechanismus für Threads als Synchronisationsobjekt

Events

- „Ereignisse für Threads“
- In Delphi gekapselt durch SyncObjs.TEvent
- TEvent.WaitFor wartet, dass ein andere Thread das Signal setzt
- ResetEvent setzt das Signal des Events zurück
- SetEvent setzt das Signal des Events

TLightweightEvent

- Baut ein ManualResetEvent mit TInterlocked/TMonitor nach
- Wenn die nicht-signalisierten Intervalle relativ kurz sind, effektiver als TSimpleEvent
- Entspricht .NET 4.0 System.Threading.ManualResetEventSlim

TCountdownEvent

- Signalisiert bei Erreichen von NULL
- Ist Null erreicht kann das Signal nur durch Reset zurückgesetzt werden
- AddCount funktioniert nur bei Zähler<>Null
- Entspricht .NET 4.0 System.Threading.CountdownEvent

Fragen …

- Gleich im Chat
- Oder Bernd.Ua@probucon.de
- Quellcodes/Slides über die Coderage Seite oder https://probucon.de/
- Vielleicht sieht man sich auf der nächsten Entwicklerkonferenz Ekon 23
- vom 28. bis 30. Oktober in Düsseldorf
  - https://entwickler-konferenz.de/de/