Betriebssysteme (BS)

VL 11.3 – Fadensynchronisation – Windows

Volkmar Sieh / Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 21 – 17. Januar 2022

https://www4.cs.fau.de/Lehre/WS21/V_BS

Agenda

Fadensynchronisation unter Windows

- Windows treibt die Idee der Warteobjekte sehr weit
  - **Jedes** Kernobjekt ist (auch) ein Synchronisationsobjekt!
    - explizite Synchronisationsobjekte: Event, Mutex, Timer, Semaphore, ...
    - implizite Synchronisationsobjekte: File, Socket, Thread, Prozess, ...
  - Semantik des Wartens hängt vom Objekt ab
    - Faden wartet auf „signalisiert"-Zustand
    - Zustand wird gegebenenfalls durch erfolgreiches Warten geändert
- Einheitliche, mächtige Systemschnittstelle für alle Objekttypen
  - Jedes Kernobjekt wird repräsentiert durch ein `HANDLE`
  - `WaitForSingleObject(hObject, dwMillisec)`
    - Wartet auf ein Synchronisationsobjekt mit Timeout
  - `WaitForMultipleObjects(nCount, hObjects[], bWaitAll, dwMillisec)`
    - Wartet auf einen Vektor von Synchronisationsobjekten mit Timeout (und/oder Warten, je nach `bWaitAll` = true/false)

Synchronisationsobjekte unter Windows

Objekt	ist signalisiert, wenn	erfolgreiches warten bewirkt
Event	Ändern des Zustands erfolgt explizit durch `SetEvent()` / `ResetEvent()`	zurückgesetzen des Events (nur bei AutoReset-Events)
Mutex	der Mutex verfügbar ist	Besitzname des Mutex
Semaphore	der Zähler der Semaphore > 0 ist	vermindern des Wertes der Semphore um 1
Waitable Timer	ein bestimmter Zeitpunkt erreicht wurde	zurücksetzen des Timers (nur bei AutoReset-Timern)
Change Notification	eine bestimmte Änderung im Dateisystem stattfand	keine Änderung des Zustands
Console Input	Eingabedaten zur Verfügung stehen	keine Änderung, solange Zeichen verfügbar sind
Process	der Prozess terminiert ist	keine Änderung des Zustands
Thread	der Thread terminiert ist	keine Änderung des Zustands
File	eine asynchrone Dateioperation abgeschlossen wurde	keine Änderung des Zustands, bis eine neue Dateioperation begonnen wird
Serial device	Daten verfügbar sind / Dateioperation abgeschlossen wurde	keine Änderung des Zustands, bis eine neue Operation begonnen wird
NamedPipe	eine asynchrone Operation abgeschlossen wurde	keine Änderung des Zustands, bis eine neue Dateioperation begonnen wird
Socket	eine asynchrone Operation abgeschlossen wurde	keine Änderung des Zustands, bis eine neue Operation begonnen wird
Job	Prozesse des Jobs terminiert sind	keine Änderung des Zustands
...		

Kosten der Fadensynchronisation

- Synchronisationsobjekte werden im Kern verwaltet
  - interne Datenstrukturen (Scheduler) $\leadsto$ Schutz
  - interne Synchronisation $\leadsto$ Konsistenz
- Das kann ihre Verwendung sehr teuer machen
  - für jede Zustandsänderung muss in den Kern gewechselt werden
  - Benutzer-/Kernmodus-Transitionen sind sehr aufwändig
  - Bei IA32 kommen schnell einige tausend Takte zusammen!
- Bei kurzen kritischen Gebieten mit geringer Wettstreitigkeit (*Contention*) schlägt dies besonders ins Gewicht
  - Die benötigte Zeit, um den Mutex zu akquirieren und freizugeben ist oft ein Vielfaches der Zeit, die das kritische Gebiet belegt ist.
  - Eine tatsächliche Konkurrenzsituation (Faden will in ein bereits belegtes kritisches Gebiet) tritt nur selten auf.

Kosten der Fadensynchronisation

- **Ansatz:** Mutex soweit wie möglich im Benutzermodus verwalten
  - Minimieren der Kosten im Normalfall
    - Normalfall $\mapsto$ kritisches Gebiet ist frei
    - Speziallfall $\mapsto$ kritisches Gebiet ist belegt
- Einführen eines *fast path* für den Normalfall
  - Test, Belegung, und Freigabe erfolgt im Benutzermodus
    - Konsistenz wird algorithmisch / durch atomare CPU-Befehle sichergestellt
  - Warten erfolgt im Kernmodus
    - für den Übergang in den passiven Wartezustand wird der Kern benötigt
  - weitere Optimierung für Multiprozessormaschinen
    - vor dem passiven Warten für begrenzte Zeit aktiv warten
      $\rightsquigarrow$ hohe Wahrscheinlichkeit, dass das kritische Gebiet vorher frei wird

Windows: CRITICAL_SECTION

- Struktur für einen *fast mutex* im Benutzermodus [8, 9]
  - verwendet intern ein Event (Kernobjekt), falls gewartet werden muss
  - Event wird „lazy“ (erst bei Bedarf) erzeugt
- Eigene Systemschnittstelle
  - `EnterCriticalSection(pCS)` / `TryEnterCriticalSection(pCS)`
    - k.G. belegen (blockierend) / versuchen zu belegen (nicht-blockierend)
  - `LeaveCriticalSection(pCS)`
    - kritisches Gebiet verlassen
  - `SetCriticalSectionSpinCount(pCS, dwSpinCount)`
    - Anzahl der Versuche für aktives Warten festlegen (nur auf MP-Systemen)

```
typedef struct _CRITICAL_SECTION {
  LONG LockCount;       // Anzahl der wartenden Threads (-1 wenn frei)
  LONG RecursionCount; // Anzahl der erfolgreichen EnterXXX-Aufrufe
  DWORD OwningThread;  // des Besitzers (OwningThread)
  HANDLE LockEvent;     // internes Warteobjekt, bei Bedarf erzeugt
  ULONG SpinCount;      // Auf MP-Systemem: Anzahl der busy-wait
                        // Versuche, bevor im Kern passiv gewartet wird
} CRITICAL_SECTION, *PCRITICAL_SECTION;
```

Windows: CRITICAL_SECTION

- Struktur für einen *fast mutex* im Benutzermodus [8, 9]
  - verwendet intern ein Event (Kernobjekt), falls gewartet werden muss
  - Event wird „lazy“ (erst bei Bedarf) erzeugt
- Eigene Systemschnittstelle
  - `EnterCriticalSection(pCS)` / `TryEnterCriticalSection(pCS)`
    - k.G. belegen (blockierend) / versuchen zu belegen (nicht-blockierend)
  - `LeaveCriticalSection(pCS)`
    - kritisches Gebiet verlassen
  - `SetCriticalSectionSpinCount(pCS, dwSpinCount)`
    - Anzahl der Versuche für aktives Warten festlegen (nur auf MP-Systemen)

```
typedef struct _CRITICAL_SECTION {
  LONG LockCount;       // Anzahl der wartenden Threads (-1 wenn frei)
  LONG RecursionCount; // Anzahl der erfolgreichen EnterXXX-Aufrufe
  DWORD OwningThread;  // des Besitzers (OwningThread)
  HANDLE LockEvent;    // internes
  ULONG SpinCount;     // Auf MP-Sy
                       // Versuche,
} CRITICAL_SECTION, *PCRITICAL_SECT
```

Unter Linux gibt es ab Kernel 2.6 mit Futexes (*F*ast *u*ser-mode mu*texes*) ein vergleichbares, noch deutlich mächtigeres Konzept. [1, 5]

Agenda

Zusammenfassung: Fadensynchronisation

- Programmfäden können jederzeit verdrängt werden
  - präemptives, probabilistisches Multitasking
  - keine run-to-completion–Semantik
  - Zugriff auf geteilten Zustand muss gesondert synchronisiert werden
- Fadensynchronisation: Ein Markt der Möglichkeiten
  - Mutex für gegenseitigen Ausschluss
  - Semaphore für Erzeuger-/Verbraucher-Szenarien
  - viele weitere Abstraktionen möglich: Leser-/Schreiber–Sperren, Vektorsemaphoren, Bedingungsvariablen, Timeouts, ...
- Grundlage ist ein BS-Konzept für passives Warten
  - Fundamentale Eigenschaft von Fäden: Sie können warten
  - aktives Warten und harte Fadensynchronisation sind (nur) in Ausnahmefällen sinnvoll