

Betriebssysteme (BS)

VL 10 – Fadensynchronisation

Daniel Lohmann / Volkmar Sieh

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 16 – 12. Januar 2017

https://www4.cs.fau.de/Lehre/WS16/V_BS



Agenda

Einleitung

Motivation
Erstes Fazit

Prioritätsebenenmodell mit Fäden

Mechanismen

Randbedingungen
Mutex, Implementierungsvarianten
Passives Warten
Semaphore

Beispiel: Windows

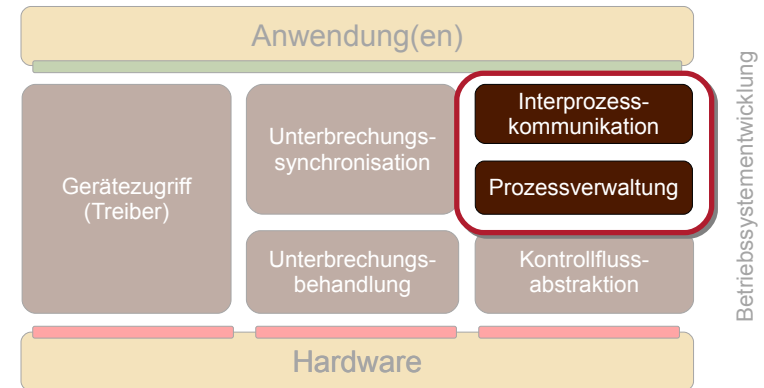
Warteobjekte
Optimierungen für Mehrkernsysteme

Zusammenfassung

Referenzen

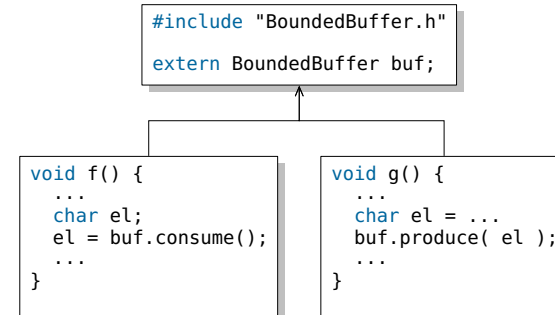


Überblick: Einordnung dieser VL



Motivation Szenario

- Gegeben: Programmfäden $\langle f \rangle$ und $\langle g \rangle$
 - Präemptives Scheduling (z. B. *round robin*)
 - Zugriff auf gemeinsame Datenstruktur *buf*



Motivation Szenario

- Gegeben: Programmfäden $\langle f \rangle$ und $\langle g \rangle$
 - Problem:** Pufferzugriffe können überlappen

```
char BoundedBuffer::consume() {
    int elements = occupied;
    if (elements == 0) return 0;
    char result = buf[nextout];
    nextout++; nextout %= SIZE;
}
```

⚡ resume $\langle g \rangle$

```
void BoundedBuffer::produce(char data) {
    int elements = occupied;
    if (elements == SIZE) return;
    buf[nextin] = data;
    nextin++; nextin %= SIZE;
    occupied = elements + 1;
}
```

⚡ resume $\langle f \rangle$

```
occupied = elements - 1;
return result;
}
```

Das hatten wir doch schon mal...

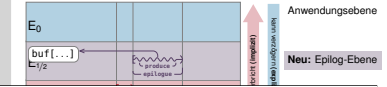


Rückblick: VL 5 – Unterbrechungssynchronisation

Was ist diesmal anders?

Prolog/Epilog-Modell – Ansatz

- Ansatz:** Latenzverbergung durch zusätzliche Ebene
 - Wir fügen eine weitere *logische Ebene* ein: $E_{1/2}$
 - $E_{1/2}$ liegt zwischen der Anwendungsebene E_0 und den UB-Ebenen E_1, \dots
 - Unterbrechungsbehandlung wird *zweigeteilt* in *Prolog* und *Epilog*
 - Prolog arbeitet auf Unterbrechungsebene E_1, \dots
 - Epilog arbeitet auf der neuen (Software-)Ebene $E_{1/2}$ (Epilogebe)
 - Zustand liegt (so weit wie möglich) auf der Epilogebe
 - eigentliche Unterbrechungsbehandlung wird nur noch kurz gesperrt

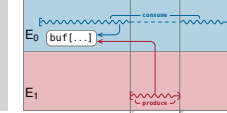


Bounded Buffer – Ansatz mit weicher Synchronisation

Zugriff „von unten“ wird weich synchronisiert: consume() liefert ein korrektes Ergebnis, auch wenn während der Abarbeitung produce() ausgeführt wurde.

```
char consume() {
    ?
}

void produce(char data) {
    ?
}
```



Zustand liegt (logisch) auf E_0



Erstes Fazit

- Bisher:** Konsistenzsicherung bei Zugriffen von Kontrollflüssen aus *verschiedenen* Ebenen
 - Zustand wurde auf einer Ebene „platziert“
 - Sicherung entweder „von oben“ (hart) oder „von unten“ (weich)
 - Innerhalb einer Ebene wurde implizit sequenzialisiert
- Nun:** Konsistenzsicherung bei Zugriffen von Kontrollflüssen aus *derselben* Ebene
 - Fäden können jederzeit durch andere Fäden verdrängt werden
 - Fäden können echt parallel arbeiten (bei mehreren CPUs)

Das ist ja auch der Sinn von Fäden!



Agenda

- Einleitung
 - Motivation
 - Erstes Fazit
- Prioritätsebenenmodell mit Fäden**
- Mechanismen
 - Randbedingungen
 - Mutex, Implementierungsvarianten
 - Passives Warten
 - Semaphore
- Beispiel: Windows
 - Warteobjekte
 - Optimierungen für Mehrkernsysteme
- Zusammenfassung
- Referenzen



Fadensynchronisation: Annahmen

- Fäden können **unvorhersehbar** verdrängt werden
 - zu jedem beliebigen Zeitpunkt
 - von beliebigen anderen Fäden höherer, gleicher, oder niedrigerer Priorität (\leftrightarrow Fortschrittgarantie)
- Annahmen typisch für Arbeitsplatzrechner \sim VL 8
 - *probabilistic, interactive, preemptive, online CPU scheduling*
 - andere Arten des Scheduling werden im Folgenden nicht betrachtet

Problematisch ist hier die **Fortschrittgarantie**

Bei rein **prioritätsgesteuertem Scheduling** (Fäden innerhalb einer Prioritätsstufe werden sequentiell abgearbeitet) könnten wir das Ebenenmodell der Unterbrechungsbehandlung (\sim VL 5) einfach auf Fadenprioritäten ausdehnen und vergleichbaren Mechanismen (expliziter Ebenenwechsel, algorithmisch unter der Annahme von *run-to-completion*) synchronisieren.

- typisch für ereignisgesteuerte Echtzeitsysteme \sim [EZS]
- in Windows/Linux: Bereich der Echtzeitprioritäten \sim VL 8
- bei mehreren Kernen bleibt das Problem der echten Parallelität!



Fadensynchronisation: Überblick

- **Ziel:** aus Anwendersicht **Koordinierung** und **Interaktion**
 - Koordinierung des exklusiven Zugriffs auf wiederverwendbare Betriebsmittel (gegenseitiger Ausschluss) \sim **Mutex**
 - Interaktion / Koordinierung von konsumierbaren Betriebsmitteln (Synchronisation) \sim **Semaphore**
- **Implementierungsansatz:** für den BS-Entwickler Steuerung der CPU-Zuteilung an **Fäden**
 - Fäden werden zeitweise von der Zuteilung ausgenommen
 - „**Warten**“ als BS-Konzept

Im Folgenden befassen wir uns mit der Perspektive der BS-Entwicklerin

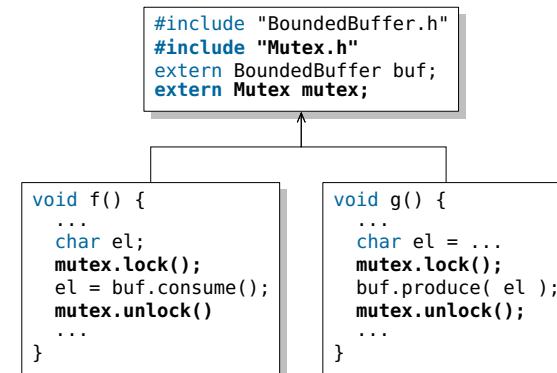


Mutex

- **Mutex** \mapsto Kurzform von **mutual exclusion**
 - Ursprung: Bezeichnername eines zweiwertigen Semaphor, eingesetzt für gegenseitigen Ausschluss [2]
 - allgemein: Algorithmus für die Sicherstellung von gegenseitigem Ausschluss in einem kritischen Gebiet
 - hier: Systemabstraktion `class Mutex`
- Schnittstelle
 - `void Mutex::lock()`
 - Betreten und Sperren des kritischen Gebiets
 - Faden kann blockieren
 - `void Mutex::unlock()`
 - Verlassen und Freigeben des kritischen Gebiets
- Korrektheitsbedingung
 - Es befindet sich maximal ein Faden im kritischen Gebiet
 - Für ausgeführte Operationen gilt: $\sum_{\text{lock}()} - \sum_{\text{unlock}()} \leq 1$



Mutex: Verwendung



Mutex mit aktivem Warten

(\mapsto *spin lock*)

- Implementierung rein auf der Benutzerebene
 - markiere Belegung in boolescher Variable (0 \mapsto frei, 1 \mapsto belegt)
 - warte in `lock()` aktiv, bis Variable 0 wird

```
// __sync_lock_test_and_set ist ein gcc builtin fuer
// (CPU-spezifisches) test-and-set (ab gcc 4.1)
class SpinningMutex {
    volatile int locked;
public:
    SpinningMutex() : locked (0) {}
    void lock(){
        while( __sync_lock_test_and_set(
            &locked, 1) == 1 )
            ;
    }
    void unlock() {
        locked = 0;
    }
};

// g++-4.2 -O3
// -fomit-frame-pointer
lock:
    mov 0x4(%esp),%edx
l1: mov $0x1,%eax
    xchg %eax,%edx
    sub $0x1,%eax
    je l1
    repz ret
unlock:
    mov 0x4(%esp),%eax
    movl $0x0,(%eax)
    ret
```



Mutex mit aktivem Warten: Bewertung

- **Vorteile**
 - Konsistenz ist sichergestellt, Korrektheitsbedingung wird erfüllt
 - unter der Voraussetzung von Fortschrittsgarantie für alle Fäden
 - Synchronisation erfolgt ohne Beteiligung des Betriebssystems
 - keine Systemaufrufe erforderlich
- **Nachteile**
 - aktives Warten verschwendet viel CPU-Zeit
 - mindestens bis die Zeitscheibe abgelaufen ist
 - bei Zeitscheiben von 10–800 msec ganz erheblich!
 - Faden wird eventuell vom Scheduler „bestraft“ (\sim VL 8)

Fazit

Aktives Warten ist – wenn überhaupt – nur auf **Multiprozessormaschinen** eine Alternative.



Mutex mit harter Synchronisation

- Implementierung mit „harter Fadensynchronisation“
 - deaktiviere Verdrängbarkeit vor Betreten des kritischen Gebiets
 - neue Systemoperation: `forbid()`
 - reaktiviere Verdrängbarkeit nach Verlassen des kritischen Gebiets
 - neue Systemoperation: `permit()`

```
class HardMutex {
public:
    void lock(){
        forbid(); // schalte Multitasking ab
    }
    void unlock(){
        permit(); // schalte Multitasking wieder an
    }
};
```

In der Welt der Echtzeitsysteme steht dieses Verfahren hinter dem *non-preemptive critical section (NPCS) protocol* [6, EZS].



Mutex mit harter Synchronisation: Implementierung

- Implementierung durch den Scheduler, z. B. über
 - spezielle nicht verdrängbare Prioritätsklasse
 - OSEK OS / AUTOSAR OS: Ressource `RES_SCHED` [7]
 - eigene Prioritätsebene $E_{1/4}$ für den Scheduler
 - war faktisch so in AmigaOS realisiert
 - `resume()` schaltet einfach zum Aufrufer zurück
- oder ganz einfach durch Betreten der Epilogebe
 - Fadenumschaltung ist üblicherweise auf der Epilogebebe angesiedelt
 - so lange ein Faden auf der Epilogebebe ist kann er nicht verdrängt werden
 - Voraussetzung: Kontrollflüsse der Epilogebebe werden sequenzialisiert
 - \sim **Sequentialisierung auch mit Epilogen!**

```
void forbid() {
    enter();
}
void permit(){
    leave();
}
```



Mutex mit harter Synchronisation: Bewertung

Vorteile

- Konsistenz ist sichergestellt, Korrektheitsbedingung wird erfüllt
- einfach zu implementieren

Nachteile

- Breitbandwirkung
 - alle Fäden (und ggfs. sogar Epiloge!) werden pauschal verzögert
- Prioritätsverletzung
 - „unbeteiligte“ Kontrollflüsse mit höherer Priorität werden verzögert
- prophylaktisches Verfahren
 - Nachteile werden in Kauf genommen, auch wenn die Wahrscheinlichkeit einer tatsächlichen Kollision sehr klein ist.

Fazit

Fadensynchronisation auf Epilogebe hat viele Nachteile. Sie ist nur auf Einprozessorsystemen für kurze, selten betretene kritische Gebiete geeignet – oder wenn sowieso mit Epilogen synchronisiert werden muss.



Passives Warten: Motivation

Bisherige Mutex-Implementierungen sind nicht ideal

- Mutex mit aktivem Warten
 - ~ Verschwendung von CPU-Zeit
- Mutex mit harter Synchronisation
 - ~ grobgranular, prioritätsverletzend

Besserer Ansatz: Fäden so lange von der CPU-Zuteilung ausschließen, wie der Mutex belegt ist.

Erfordert neues BS-Konzept: passives Warten

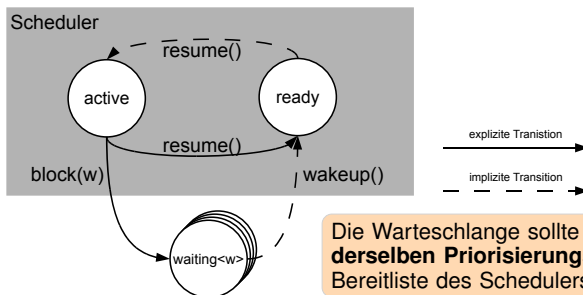
- Fäden können auf ein Ereignis „passiv warten“
 - passiv warten \mapsto von CPU-Zuteilung ausgeschlossen sein
 - (Neuer) Fadenzustand: *wartend* (auf Ereignis)
- Eintreffen des Ereignisses bewirkt Verlassen des Wartezustands
 - Faden wird in CPU-Zuteilung eingeschlossen
 - Anschließender Fadenzustand: *bereit*



Passives Warten: Implementierung

Erforderliche Abstraktionen:

- Operationen: `block()`, `wakeup()`
 - Betreten bzw. Verlassen des Wartezustands
- Warteobjekt: `Waitingroom`
 - repräsentiert das Ereignis auf das gewartet wird
 - enthält üblicherweise eine Warteschlange der wartenden Fäden



Die Warteschlange sollte sinnvollerweise mit derselben **Priorisierungsstrategie** wie die Bereitliste des Schedulers verwaltet werden!



Mutex mit passivem Warten: Implementierung

```
class WaitingMutex : public Waitingroom {
    int locked;
public:
    WaitingMutex() : locked (0) {}
    void lock(){
        while( __sync_lock_test_and_set( &locked, 1) == 1 )
            scheduler.block( *this );
    }
    void unlock() {
        locked = 0;
        // Maximal einen wartenden Thread holen und aufwecken
        Customer* t = dequeue();
        if( t ) scheduler.wakeup( *t );
    }
};
```

Bei dieser Lösung gibt es noch ein Problem...



Mutex mit passivem Warten: Implementierung

```
class WaitingMutex : public Waitingroom {
    int volatile locked;
public:
    WaitingMutex() : locked (0) {}
    void lock(){
        mutex.lock();
        while( locked == 1 )
            scheduler.block( *this );
        locked = 1;
        mutex.unlock();
    }
    void unlock() {
        mutex.lock();
        locked = 0;
        // Maximal einen wartenden Thread holen und aufwecken
        Customer* t = dequeue();
        if( t ) scheduler.wakeup( *t );
        mutex.unlock();
    }
};
```

lock() und unlock()
bilden ein eigenes
kritisches Gebiet

Kann man dieses
kritische Gebiet mit
einem Mutex schützen?



Mutex mit passivem Warten: Implementierung

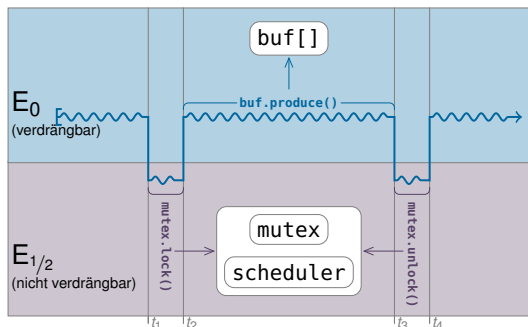
```
class WaitingMutex : public Waitingroom {
    int volatile locked;
public:
    WaitingMutex() : locked (0) {}
    void lock(){
        enter();
        while( locked == 1 )
            scheduler.block( *this );
        locked = 1;
        leave();
    }
    void unlock() {
        enter();
        locked = 0;
        // Maximal einen wartenden Thread holen und aufwecken
        Customer* t = dequeue();
        if( t ) scheduler.wakeup( *t );
        leave();
    }
};
```

Mit einem HardMutex ginge es!
Faktisch schützt man lock()
und unlock() somit, wie hier
dargestellt, auf **Epiloge**bene.



Mutex mit passivem Warten: Fazit

- Mutex-Zustand liegt nun **im Kern** auf der Epilogebeine
 - genauer: auf derselben Ebene wie der **Scheduler Zustand**
- Das ist ein **allgemein verwendbares Prinzip**
 - Implementierung der Synchronisationsmechanismen für E_0 -Kontrollflüsse wird auf $E_{1/2}$ synchronisiert.



Noch besser wäre natürlich **weiche Synchronisation**.
Dazu mehr in [CS]!



Semaphore

- Semaphore ist das klassische Synchronisationsobjekt
 - Edgar W. Dijkstra, 1963 [3]
 - In vielen BS: Grundlage für alle Warte-/Synchronisationsobjekte
 - Für uns: Semaphore \rightarrow Warteobjekt + Zähler
- Zwei Standardoperationen (mit jeweils diversen Namen [2-4])
 - `prolaag()`, `P()`, `wait()`, `down()`, `acquire()`, `pend()`
 - wenn zähler > 0 vermindere Zähler
 - wenn zähler ≤ 0 warte bis Zähler > 0 und probiere es noch einmal
 - `verhoog()`, `V()`, `signal()`, `up()`, `release()`, `post()`
 - erhöhe Zähler
 - wenn Zähler = 1 wecke gegebenenfalls wartenden Faden
- Es gibt vielfältigste Varianten

Implementierung der
Standardvariante
erfolgt in der Übung!



Semaphore: Verwendung

- Semantik der Semaphore eignet sich besonders für die Implementierung von Erzeuger/Verbraucher-Szenarien
 - Also für den geordneten Zugriff auf **konsumierbare Betriebsmittel**
 - Zeichen von der Tastatur
 - Signale, die auf Fadenebene weiterverarbeitet werden sollen
 - ...
 - Interner Zähler repräsentiert die Anzahl der Ressourcen
 - Erzeuger ruft $V()$ auf für jedes erzeugte Element.
 - Verbraucher ruft $P()$ auf, um ein Element zu konsumieren
 - ~ wartet gegebenenfalls.

Beachte!

- $P()$ kann auf Fadenebene blockieren, $V()$ blockiert jedoch nie!
- Als **Erzeuger** kommt daher auch ein Kontrollfluss auf Epilogebe oder Unterbrechungsebene in Frage. (Entsprechende Synchronisation des internen Semaphorzustands vorausgesetzt.)



Semaphore vs. Mutex: Einordnung

- Mutex wird „klassisch“ als binärer Semaphor bezeichnet [2]
 - $\text{Mutex} \mapsto \text{Semaphore mit initialem Zählerwert } 1$
 - $\text{lock}() \mapsto P(), \text{unlock}() \mapsto V()$
- Die Semantik ist (heute) jedoch i. a. deutlich strenger:
 - Ein belegter Mutex hat (implizit oder explizit) einen **Besitzer**.
 - Nur dieser Besitzer darf $\text{unlock}()$ aufrufen.
 - Muteximplementierungen in z. B. Linux oder Windows überprüfen dies.
 - Ein Mutex kann (üblicherweise) auch **rekursiv** belegt werden
 - Interner Zähler: *Derselbe* Faden kann mehrfach $\text{lock}()$ aufrufen; nach der entsprechenden Anzahl von $\text{unlock}()$ -Aufrufen ist der Mutex frei
 - Eine Semaphore kann hingegen von *jedem* Faden verändert werden.

Semaphore als Basis aller Dinge?

In vielen BS ist Semaphore die **Grundabstraktion** für Fadensynchronisation. Sie wird deshalb in der Literatur oft als (notwendige) **Implementierungsbasis** für Mutex, Bedingungsvariable, Leser-Schreiber-Sperre etc. angesehen.



Agenda

Einleitung

Motivation

Erstes Fazit

Prioritätsebenenmodell mit Fäden

Mechanismen

Randbedingungen

Mutex, Implementierungsvarianten

Passives Warten

Semaphore

Beispiel: Windows

Warteobjekte

Optimierungen für Mehrkernsysteme

Zusammenfassung

Referenzen



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 <code>SetEvent()</code> / <code>ResetEvent()</code>	zurücksetzen des Events (nur bei <code>AutoReset</code> -Events)
Mutex	der Mutex verfügbar ist	Besitzname des Mutex
Semaphore	der Zähler der Semaphore > 0 ist	vermindern des Wertes der Semaphore um 1
Waitable Timer	ein bestimmter Zeitpunkt erreicht wurde	zurücksetzen des Timers (nur bei <code>AutoReset</code> -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) \rightsquigarrow Schutz
 - interne Synchronisation \rightsquigarrow 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 \rightsquigarrow kritisches Gebiet ist frei
 - Spezialfall \rightsquigarrow 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_SECTION;
```

Unter Linux gibt es ab Kernel 2.6 mit **Futexes** (*Fast user-mode mutexes*) ein vergleichbares, noch deutlich mächtigeres Konzept. [1, 5]



Agenda

- Einleitung
 - Motivation
 - Erstes Fazit
- Prioritätsebenenmodell mit Fäden
- Mechanismen
 - Randbedingungen
 - Mutex, Implementierungsvarianten
 - Passives Warten
 - Semaphore
- Beispiel: Windows
 - Warteobjekte
 - Optimierungen für Mehrkernsysteme
- Zusammenfassung
- Referenzen



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–Sperrern, 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



Referenzen

- [1] Ulrich Depper. *Futexes are tricky*. Techn. Ber. (Version 1.5). Red Hat Inc., Aug. 2009. URL: <http://www.akkadia.org/drepper/futex.pdf> (besucht am 06.01.2011).
- [2] Edsger Wybe Dijkstra. *Cooperating Sequential Processes*. Techn. Ber. (Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996). Eindhoven, The Netherlands: Technische Universiteit Eindhoven, 1965. URL: <http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF>.
- [3] Edsger Wybe Dijkstra. „Multiprogramming en de X8“. circulated privately. Juni 1963. URL: <http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD57.PDF> (besucht am 06.01.2011).
- [4] Per Brinch Hansen. *Betriebssysteme*. München: Carl Hanser Verlag, 1977. ISBN: 3-446-12105-6.
- [5] Matthew Kirkwood Hubertus Franke Rusty Russell. „Fuss, futexes and furwocks: Fast Userlevel Locking in Linux“. In: *Proceedings of the Ottawa Linux Symposium*. (Ottawa, OT, Canada, 26.–29. Juni 2002). Hrsg. von Andrew J. Hutton, Stephanie Donovan und C. Craig Ross. Juni 2002, S. 479–495.
- [6] Aloysius K. L. Mok. „Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment“. Diss. Cambridge, MA, USA: Massachusetts Institute of Technology, MIT, Mai 1983.



Referenzen (Forts.)

- [7] OSEK/VDX Group. *Operating System Specification 2.2.3*. Techn. Ber. <http://portal.osek-vdx.org/files/pdf/specs/os223.pdf>, visited 2014-09-29. OSEK/VDX Group, Feb. 2005.
- [8] Matt Pietrek und Russ Osterlund. „Break Free of Code Deadlocks in Critical Sections Under Windows“. In: *MSDN Magazine* 18 (12 Dez. 2003). ISSN: 1528-4859. URL: <http://msdn.microsoft.com/en-us/magazine/cc164040.aspx> (besucht am 06.01.2011).
- [9] Jeffrey Richter. *Windows via C/C++ (5th edition)*. Microsoft Press. ISBN: 978-0735624245.
- [CS] Wolfgang Schröder-Preikschat. *Concurrent Systems*. Vorlesung mit Übung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4, 2015 (jährlich). URL: https://www4.cs.fau.de/Lehre/WS15/V_CS.
- [EZS] Peter Ulbrich. *Echtzeitsysteme*. Vorlesung mit Übung. Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl für Informatik 4, 2015 (jährlich). URL: https://www4.cs.fau.de/Lehre/WS15/V_EZS.

