

Betriebssysteme (BS)

VL 10 – Fadensynchronisation

Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen Nürnberg

WS 09 – 13. Januar 2009

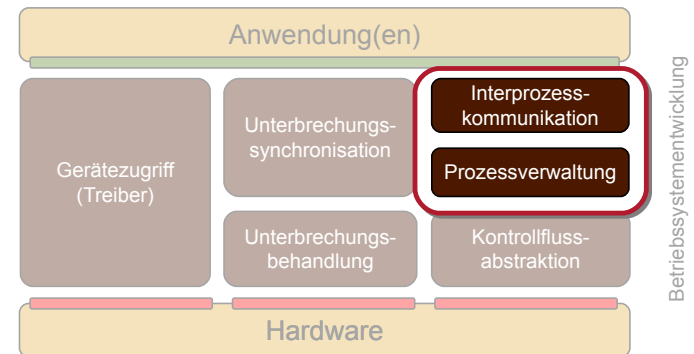


Agenda

- Motivation / Problem
- Kontrollflussebenenmodell mit Fäden
- Fadensynchronisation
 - Randbedingungen
 - Mutex, Implementierungsvarianten
 - Konzept des passiven Wartens
 - Semaphore
- Beispiel: Synchronisationsobjekte unter Windows
- Zusammenfassung



Überblick: Einordnung dieser VL



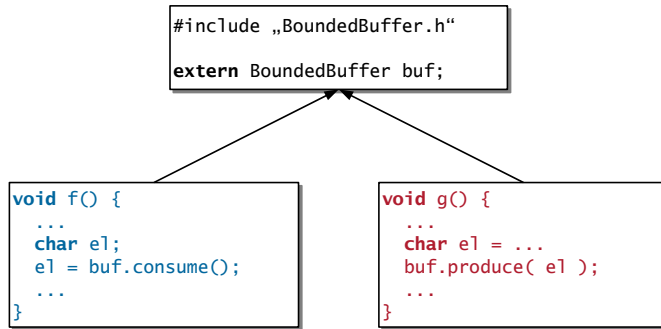
Agenda

- **Motivation / Problem**
- Kontrollflussebenenmodell mit Fäden
- Fadensynchronisation
 - Randbedingungen
 - Mutex, Implementierungsvarianten
 - Konzept des passiven Wartens
 - Semaphore
- Beispiel: Synchronisationsobjekte unter Windows
- Zusammenfassung



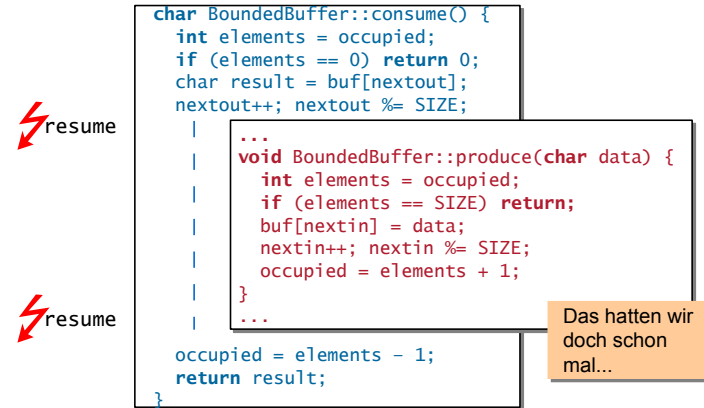
Motivation: Szenario

- Gegeben: Programmfäden <f> und <g>
 - präemptives Round-Robin – Scheduling
 - Zugriff auf gemeinsamen Puffer buf



Motivation: Konsistenzprobleme

- Gegeben: Programmfäden <f> und <g>
 - **Problem: Pufferzugriffe können überlappen**

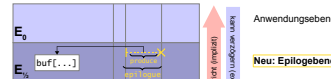


Rückblick: VL-5: Unterbrechungssynchronisation

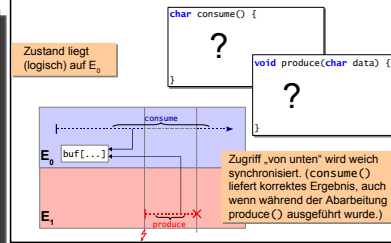
Was ist diesmal anders?

Prolog/Epilog-Modell – Ansatz

- **Idee:** Wir fügen eine **weitere Ebene** E_0 zwischen der Anwendungsebene E_1 und den UB-Ebenen $E_{1..n}$ ein
 - UB wird **zweigeteilt** in **Prolog** und **Epilog**
 - Prolog arbeitet auf Unterbrechungsebene $E_{1..n}$
 - Epilog arbeitet auf der neuen (Software-)Ebene E_0 (Epilogebe)
 - Zustand liegt (so weit wie möglich) auf der Epilogebe
 - eigentliche Unterbrechungsbehandlung wird nur noch kurz gesperrt



Bounded Buffer – Ansatz mit weicher Synchronisation



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

- Motivation / Problem
- **Kontrollflussebenenmodell mit Fäden**
- Fadensynchronisation
 - Randbedingungen
 - Mutex, Implementierungsvarianten
 - Konzept des passiven Wartens
 - Semaphore
- Beispiel: Synchronisationsobjekte unter Windows
- Zusammenfassung

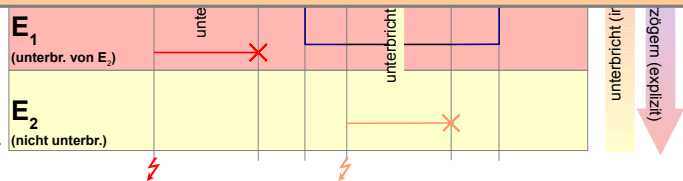
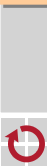


Kontrollflussebenenmodell

- Kontrollflüsse auf Ebene E_l sind
 1. **jederzeit unterbrechbar** durch Kontrollflüsse von E_m (für $m > l$)
 2. **nie unterbrechbar** durch Kontrollflüsse von E_k (für $k \leq l$)
 3. **sequentialisiert** mit weiteren Kontrollflüssen von E_l

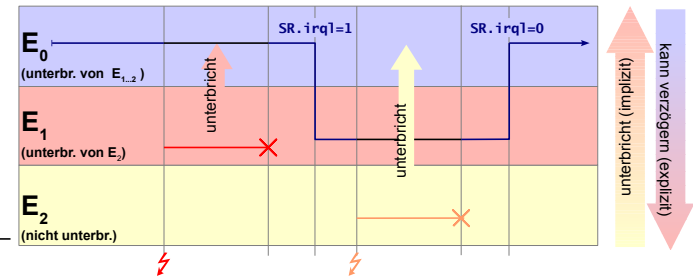
Mit der Unterstützung **präemptiver Fäden** können wir diese **Annahme** nicht länger aufrechterhalten!

- keine *run-to-completion*-Semantik mehr
- Zustandszugriffe (aus derselben Ebene) sind nicht mehr implizit sequentialisiert
- gilt für alle Ebenen, die Verdrängung (*Preemption*) und echte Parallelität von Kontrollflüssen erlauben; üblicherweise ist das die Anwendungsebene E_0



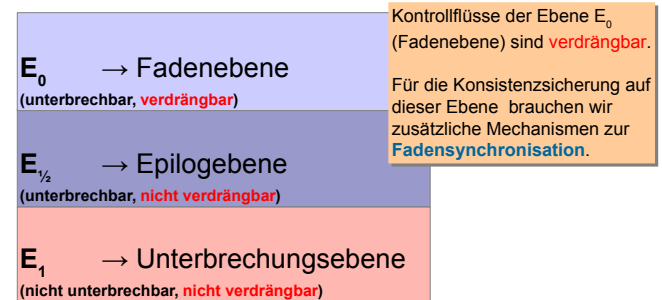
Kontrollflussebenenmodell

- Kontrollflüsse auf Ebene E_l sind
 1. **jederzeit unterbrechbar** durch Kontrollflüsse von E_m (für $m > l$)
 2. **nie unterbrechbar** durch Kontrollflüsse von E_k (für $k \leq l$)
 3. **sequentialisiert** mit weiteren Kontrollflüssen von E_l



Erweitertes Kontrollflussebenenmodell

- Kontrollflüsse auf Ebene E_l sind
 1. **jederzeit unterbrechbar** durch Kontrollflüsse von E_m (für $m > l$)
 2. **nie unterbrechbar** durch Kontrollflüsse von E_k (für $k \leq l$)
 3. **jederzeit verdrängbar** durch Kontrollflüsse von E_l (für $l = 0$)



Fadensynchronisation: Annahmen

- Fäden können **unvorhersehbar** verdrängt werden
 - zu jeder Zeit (auch durch externe Ereignisse)
 - Unterbrechungen
 - von beliebigen anderen Fäden (Fortschrittsgarantie)
 - höherer, gleicher oder niedrigerer Priorität
- Annahmen sind typisch für Arbeitsplatzrechner
 - *probabilistic, interactive, preemptive, online CPU scheduling*
 - andere Arten des Scheduling werden hier nicht betrachtet

Es vor allem die **Fortschrittsgarantie**, die uns das Leben schwer macht.

In rein prioritätengesteuerten Systemen, in denen die Fäden innerhalb einer Prioritätsstufe sequentiell abgearbeitet werden, könnten wir das Prioritätsebenenmodell der Unterbrechungsbehandlung einfach auf Fadenprioritäten ausdehnen und mit vergleichbaren Mechanismen (expliziter Ebenenwechsel, algorithmisch) synchronisieren.
(→ ereignisgesteuerte Echtzeitsysteme, → VL *Echtzeitsysteme*)

10 – 13

Fadensynchronisation: Überblick

- Ziel (für den Anwender):
 - Koordinierung des Zugriffs auf **Betriebsmittel**
 - Koordinierung des exklusiven Zugriffs auf wiederverwendbare Betriebsmittel → **Mutex**
 - Interaktion / Koordinierung von konsumierbaren Betriebsmitteln → **Semaphore**
- Implementierungsansatz (für den BS-Entwickler):
 - Koordinierung der CPU-Zuteilung an **Fäden**
 - Bestimmte Fäden werden zeitweise von der Zuteilung der CPU ausgenommen
→ „Warten“ als BS-Konzept

Im Folgenden befassen wir uns mit der Perspektive des BS-Entwicklers



Agenda

- Motivation / Problem
- Kontrollflussebenenmodell mit Fäden
- **Fadensynchronisation**
 - **Randbedingungen**
 - **Mutex, Implementierungsvarianten**
 - Konzept des passiven Wartens
 - Semaphore
- Beispiel: Synchronisationsobjekte unter Windows
- Zusammenfassung

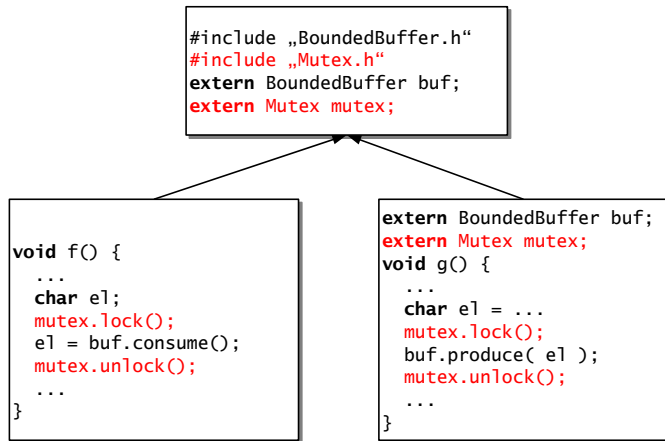


Mutex – gegenseitiger Ausschluss

- **Mutex**: Kurzform vom *mutual exclusion*
 - historisch: Name für einen zweiwertigen Semaphore [2]
 - allgemein: ein Algorithmus für die Sicherstellung von gegenseitigem Ausschluss in einem kritischen Gebiet
 - hier: eine 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**: $\sum_{exec} lock() - \sum_{exec} unlock() \leq 1$
 - zu jedem Zeitpunkt befindet sich maximal ein Faden im kritischen Gebiet



Mutex: Verwendung



Mutex: mit aktivem Warten (spin lock)

- Implementierung rein auf der Benutzerebene
- Ansatz:
 - markiere Belegung in boolescher Variable (0=frei, 1=belegt)
 - warte in lock() aktiv bis Variable 0 wird

```
// __sync_lock_test_and_set ist ein gcc builtin für
// (CPU-spezifisches) test-and-set (ab gcc 4.1)
```

```
class SpinningMutex {
public:
    volatile int locked;
    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),%eax
11: mov    $0x1,%eax
    xchg  %eax,(%edx)
    sub   $0x1,%eax
    je    11
    repz ret
unlock:
    mov    0x4(%esp),%eax
    movl  $0x0,(%eax)
    ret
```

Bewertung: Mutex mit aktivem Warten

- 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“

Aktives Warten ist, wenn überhaupt, nur auf Multiprozessormaschinen eine echte Alternative.

Mutex: mit „harter Synchronisation“

- Implementierung mit „harter Fadensynchronisation“
- Ansatz:
 - deaktiviere Multitasking vor Betreten des kritischen Gebiets
 - reaktiviere Multitasking nach Verlassen des kritischen Gebiets
- erfordert Möglichkeit, präemptives Verdrängen zu unterbinden
 - Spezielle Operationen: forbid(), permit()

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

Mutex: mit „harter Synchronisation“

- Implementierung von `forbid()` und `permit()`
 - z.B. durch den Scheduler
 - spezielle nicht verdrängbare „Echzeitpriorität“
 - eigene Prioritätsebene $E_{\frac{1}{2}}$ für den Scheduler
 - `resume()` schaltet einfach immer wieder zum Aufrufer zurück
- oder ganz einfach auf Epilogebeine
 - Fadenumschaltung ist üblicherweise auf der Epilogebeine angesiedelt
 - Voraussetzung: Kontrollflüsse der Epilogebeine sind sequenzialisiert
 - solange ein Faden auf der Epilogebeine ist, kann er also nicht verdrängt werden
 - Folge: Sequentialisierung auch mit Epilogen!

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



Bewertung: Mutex mit „harter Synchronisation“

- **Vorteile**
 - Konsistenz ist sichergestellt, Korrektheitsbedingung wird erfüllt
 - Einfach zu implementieren
- **Nachteile**
 - Breitbandwirkung
 - es werden pauschal alle Fäden (und eventuell sogar Epiloge!) verzögert
 - Prioritätsverletzung
 - es werden Kontrollflüsse verzögert, die eine höhere Priorität haben
 - prophylaktisches Verfahren
 - Nachteile werden in Kauf genommen, obwohl die Wahrscheinlichkeit, dass tatsächlich eine Kollision eintritt, sehr klein ist.

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



Agenda

- Motivation / Problem
- Kontrollflussebenenmodell mit Fäden
- **Fadensynchronisation**
 - Randbedingungen
 - Mutex, Implementierungsvarianten
 - **Konzept des passiven Wartens**
 - **Semaphore**
- Beispiel: Synchronisationsobjekte unter Windows
- Zusammenfassung



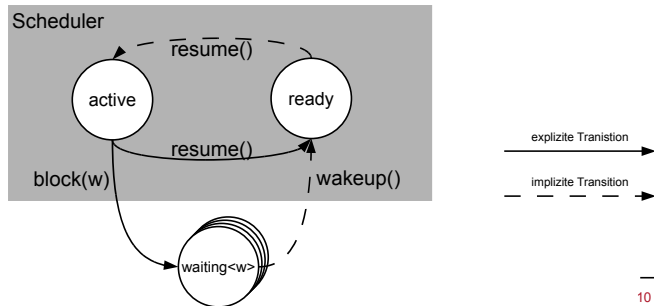
passives Warten

- Bisherige Mutex-Implementierungen sind nicht ideal
 - Mutex mit aktivem Warten: Verschwendung von CPU-Zeit
 - Mutex mit harter Synchronisation: grobgranular, prioritätsverletzend
- Besserer Ansatz: Faden 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 → von CPU-Zuteilung ausgeschlossen sein
 - Neuer Fadenzustand: wartend (auf Ereignis)
 - Eintreffen des Ereignisses bewirkt Verlassen des Wartezustands
 - Faden wird in CPU-Zuteilung eingeschlossen
 - Fadenzustand: bereit



BS-Konzept: passives Warten

- Erforderliche Abstraktionen:
 - Scheduler-Operationen: `block()`, `wakeup()`
 - Warteobjekt: `Waitingroom`
 - repräsentiert das Ereignis auf das gewartet wird
 - üblicherweise eine Warteschlange der wartenden Fäden



10 – 25

BS-Konzept: passives Warten

- Scheduler-Operationen
 - `block(Waitingroom& w)`
 - reihe aktiven Faden (Aufrufer) in die Schlange des Warteobjekts `w` ein
 - aktiviere anderen Faden (von Bereitliste)
 - `wakeup(Customer& t)`
 - reihe `t` in Bereitliste ein
- Waitingroom-Operationen
 - `enqueue(Customer*)`
 - `Customer* dequeue()`

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

dl Betriebssysteme (VL 10 | WS 09)

Fadensynchronisation

10 – 26

Mutex: mit passivem Warten

```
class WaitingMutex : public Waitingroom {
    int locked;
public:
    WaitingMutex() : locked (0) {}
    void lock(){
        while( __sync_lock_test_and_set( &locked, 1) == 0 )
            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...

10 – 27

Mutex: mit passivem Warten

```
class WaitingMutex : public Waitingroom {
    int volatile locked;
public:
    SpinningMutex() : 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?

dl Betriebssysteme (VL 10 | WS 09)

Fadensynchronisation

10 – 27

dl Betriebssysteme (VL 10 | WS 09)

Fadensynchronisation

10 – 28

Mutex: mit passivem Warten

```

class WaitingMutex : public Waitingroom {
    int volatile locked;
public:
    SpinningMutex() : 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!

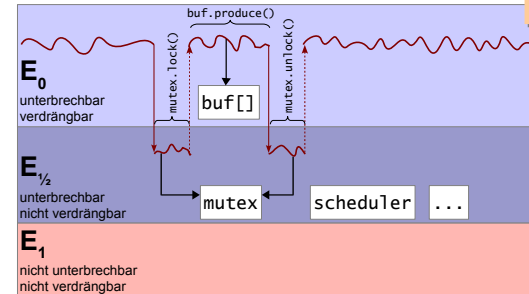
Faktisch schützt man lock() und unlock() somit, wie hier dargestellt, auf **Epiloge**ebene.



Fazit: Implementierung von Warten

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

Noch besser wäre natürlich weiche Synchronisation!



Semaphore

- Semaphore ist *das* klassische Synchronisationsobjekt
 - Edgar W. Dijkstra, 1962 [2]
 - In vielen BS: Grundlage für alle Warte-/Synchronisationsobjekte
 - Für uns: Semaphore = Warteobjekt + Zähler
- Operationen
 - zwei Standardoperationen (mit jeweils diversen Namen [2,3,5])
 - 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.

P() kann auf Fadenebene blockieren, V() blockiert jedoch nie!

Als **Erzeuger** kommt daher auch ein **Kontrollfluss** auf **Epiloge**ebene oder **Unterbrechungsebene** in Frage. (Entsprechende Synchronisation des internen Semaphore Zustands vorausgesetzt.)



Semaphore vs. Mutex: Einordnung

- Mutex wird "klassisch" als **zweiwertiger Semaphor** bezeichnet
 - Mutex → Semaphore mit initialem Zählerwert 1
 - $\text{lock}() \rightarrow P()$, $\text{unlock}() \rightarrow V()$
- 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 erhöht oder vermindert werden.

In vielen BS ist Semaphore die **Grundabstraktion** für Synchronisationsobjekte. Sie wird deshalb als **Implementierungsbasis** für Mutexe, Bedingungsvariablen, Leser-Schreiber-Sperren, ... verwendet.



Synchronisation 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 Systemschnittstelle für alle Objekttypen
 - Kernobjekt wird repräsentiert durch ein HANDLE
 - `WaitForSingleObject(hObject, dwMillisec)`
 - Warten auf ein Synchronisationsobjekt mit Timeout
 - `WaitForMultipleObjects(nCount, hObject[], bWaitAll, dwMillisec)`
 - Warten auf ein oder mehrere Synchronisationsobjekte mit Timeout (und/oder Warten, je nach `bWaitAll = true/false`)



Agenda

- Motivation / Problem
- Kontrollflussebenenmodell mit Fäden
- Fadensynchronisation
 - Randbedingungen
 - Mutex, Implementierungsvarianten
 - Konzept des passiven Wartens
 - Semaphore
- **Beispiel: Synchronisationsobjekte unter Windows**
- Zusammenfassung



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 (bei <code>AutoReset-Events</code>)
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 (bei <code>AutoReset-Timern</code>)
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 (Win 2000)	alle Prozesse des Jobs terminiert sind	keine Änderung des Zustands

Synchronisation und Kosten

- Synchronisationsobjekte werden im Kern verwaltet
 - kritische Datenstrukturen → Schutz
 - interne Synchronisation auf Epilogebeane → 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 Mutexen fällt dieser Aufwand besonders ins Gewicht
 - Die benötigte Zeit, um den Mutex zu sperren 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.



Synchronisation und Kosten

- Ansatz: Mutex soweit wie möglich im **Benutzermodus** verwalten
 - Minimieren der Kosten im Normalfall
 - Normalfall: kritisches Gebiet ist frei
 - Spezialfall: kritisches Gebiet ist belegt
- Einführen eines *fast path* für den Normalfall
 - Test, Belegung, und Freigabe im Benutzermodus
 - Konsistenz wird algorithmisch / durch atomare CPU-Befehle sichergestellt
 - Warten 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
 - hohe Wahrscheinlichkeit, dass das kritische Gebiet vorher frei wird



Windows: CRITICAL_SECTION

- Struktur für einen *fast mutex* im Benutzermodus [8]
 - 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]
 - 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
    LONG RecursionCount; // Anzahl der
    DWORD OwningThread; // des Besitzers
    HANDLE LockEvent; // internes Wa
    ULONG SpinCount; // Auf MP-Syste
                    // Versuche, b
} CRITICAL_SECTION, *PCRITICAL_SECTION;
```

Unter Linux gibt es ab Kernel 2.6 mit **Futexes** (Fast user-mode mutexes) ein vergleichbares, noch deutlich mächtigeres Konzept. [7,6]



Agenda

- Motivation / Problem
- Kontrollflussebenenmodell mit Fäden
- Fadensynchronisation
 - Randbedingungen
 - Mutex, Implementierungsvarianten
 - Konzept des passiven Wartens
 - Semaphore
- Beispiel: Synchronisationsobjekte unter Windows
- **Zusammenfassung**



Zusammenfassung

- 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



Bibliographie

- [1] K. R. Apt. Edsger Wybe Dijkstra (1930 – 2002): A Portrait of a Genius. <http://arxiv.org/pdf/cs.GL/0210001>, 2002.
- [2] E. W. Dijkstra. Multiprogramming en de X8, 1962. [4].
- [3] E. W. Dijkstra. Cooperating Sequential Processes. Technical report, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 1965. (Reprinted in Great Papers in Computer Science, P. Laplante, ed., IEEE Press, New York, NY, 1996).
- [4] E. W. Dijkstra. EWD Archive: Home. <http://www.cs.utexas.edu/users/EWD>, 2002.
- [5] P. B. Hansen. Betriebssysteme. Carl Hanser Verlag, erste edition, 1977. ISBN 3-446-12105-6.
- [6] Ulrich Drepper. Futexes are tricky. <http://people.redhat.com/drepper/futex.pdf>, 2005
- [7] Hubertus Franke, Rusty Russell, Matthew Kirkwood, Fuss, futexes and furwoks: Fast Userlevel Locking in Linux, Ottawa Linux Symposium. http://www.linux.org.uk/~ajh/ols2002_proceedings.pdf.gz, 2002.
- [8] Matt Pietrek, Russ Osterlund. Break Free of Code Deadlocks in Critical Sections Under Windows. MSDN Magazine <http://msdn.microsoft.com/msdnmag/issues/03/12/CriticalSections/default.aspx#S2>, 2003

