

Peter Rene Dietmüller, Michael Sonntag

KV Betriebssysteme Synchronisation

Institut für Informationsverarbeitung und
Mikroprozessortechnik (FIM)
Johannes Kepler Universität Linz, Österreich

E-Mail: sonntag@fim.uni-linz.ac.at
<http://www.fim.uni-linz.ac.at/staff/sonntag.htm>

© Michael Sonntag 2004

Synchronisation – Was ist das?

- Sobald auf einem Rechner mehr als ein Prozess läuft, kann es zu Problemen kommen:
- Wenn zwei Prozesse auf die selben Daten zugreifen, können Inkonsistenzen entstehen!
 - Beide Lesen: Unproblematisch!
 - Einer liest und einer schreibt: Vorher oder nachher lesen?
 - » Scheduling, Prioritäten, ...
 - Beide schreiben: Wer darf nachher schreiben (d. h. ändern)?
 - » Schlecht entworfenes Programm, Transaktionen, ...
- Das Problem entsteht **NUR** bei "shared memory"!
 - Kein gemeinsamer Speicher: Nachrichten, Blackboards, ...
 - » Das Problem existiert auch dort, wird aber auf die Kommunikationsebene verlagert!

Michael Sonntag Synchronisation 2

Synchronisation – Was ist das?

- In der Praxis ist es jedoch so, dass **beide** Prozesse **lesen** und gleichzeitig **beide schreiben** wollen!
 - A liest, B liest, A schreibt, B schreibt
 - » Ergebnis: Komplett falsch!
 - » Weder gleich wie wenn zuerst A gelesen und geschrieben hätte, wie wenn zuerst B alle Aktionen durchgeführt hätte!
- Beispiel:

| | | |
|---------------|---|----------------------------|
| → Start: a=0; | Prozess 1: a=a+2; | Prozess 2: a=a+3; |
| → Ablauf: | P1: register _x =a | {register _x =0} |
| → | P2: register _y =a | {register _y =0} |
| → | P1: register _x =register _x +2 | {register _x =2} |
| → | P2: register _y =register _y +3 | {register _y =3} |
| → | P1: a=register _x | {a=2} |
| → | P2: a=register _y | {a=3} |

Michael Sonntag Synchronisation 3

Race Conditions

- Diese Art von Problemen nennt man "race condition"
- Diese Fehler sind besonders schwer zu finden
 - Sie treten nicht immer sondern nur ab und zu auf
 - Die Probleme treten dann an vielen verschiedenen Stellen auf
 - Beim debuggen tritt der Fehler meist gar nicht auf
 - Fügt man Log-Statements ein, tritt der Fehler oft gar nicht auf
 - Keine Reproduzierbarkeit (winzige Zeitunterschiede wichtig)
- Unbedingt von vornherein vermeiden:
 - Immer wenn mehrere Prozesse auf dieselben Daten zugreifen
 - Irgendeine Art von Synchronisierung einbauen
 - » Das Programm wird langsamer, auch wenn sie tatsächlich nicht gebraucht würde
 - » Das Programm ist SEHR viel sicherer!

Michael Sonntag Synchronisation 4

"Critical Region"

- Programme die Synchronisation benötigen besitzen sogenannte "critical regions"
- Dies sind die Teile, in denen auf dem gemeinsamen Speicher gearbeitet wird
- Es darf sich als Definition immer nur ein einziger Prozess in einer critical region befinden
 - Alle anderen müssen am Beginn dieses Programmteils solange aufgehalten werden, bis dieser Teil verlassen wurde
 - Diese Region sollte so klein wie möglich gehalten werden!
- Ist garantiert, dass sich immer nur ein Prozess in einer critical region befindet, so sind race conditions ausgeschlossen!

Michael Sonntag Synchronisation 5

"Critical Region"

- Critical regions beheben nur das Problem der race conditions
 - Die Reihenfolge von critical regions (siehe oben; zwei Prozesse wollen schreiben: wer darf nachher schreiben) wird **nicht** geregelt oder beeinflusst
- Die restlichen Probleme müssen selbst durch die Programmlogik behoben werden!
 - Je nach der Bedeutung, Priorität der Aktionen oder Benutzer...
- Critical regions sorgen nur dafür, dass das Programm bei **gegebener Eingabe immer die gleichen Ergebnisse** liefert
 - "Nur" keine Überlappung von Instruktionen
 - Deren Reihenfolge wird **nicht** garantiert

Michael Sonntag Synchronisation 6

Lösung für "Gegenseitigen Ausschluss"

- Um critical regions zu implementieren müssen drei Probleme gelöst werden:
 - Mutual exclusion: Keine zwei Prozesse dürfen sich gleichzeitig in einer critical region befinden
 - » Die Basis-Voraussetzung und das Ziel
 - Progress: Ist kein Prozess in einer critical region so wird in endlicher Zeit genau einer der Prozesse als berechtigt ausgewählt, diese zu betreten
 - » Irgendwann kommt der nächste Prozess dran (Kein deadlock)
 - Bounded waiting: Es gibt einen Grenzwert wie oft andere Prozesse ihre critical region betreten dürfen nachdem eine anderer Prozess seinen Wunsch dafür angemeldet hat
 - » Alle kommen irgendwann einmal dran (Keine starvation)

Michael Sonntag Synchronisation 7

Grundlegende Konzepte

- Es gibt viele verschiedene Arten, Synchronisation zu erreichen. Bekannte Grundkonzepte sind:
 - Semaphore: Integer-Variable
 - » Analogie: Züge auf einem Eisenbahngleis müssen vor dem Signal warten, bevor sie weiterfahren dürfen
 - Critical region: Boolesche Variable
 - » Analogie: Wer den "Stab des Sprechers" hält, darf reden
 - Monitor: "Objektorientierte critical region"
 - » Analogie: Immer nur eine Person darf mit der Geisterbahn fahren; ist aber niemand da, fährt der Wagen eben leer
- Die Grundkonzepte sind äquivalent, d. h. sie können jeweils durch die anderen implementiert werden

Michael Sonntag Synchronisation 8

Das "Bounded Buffer" Problem

- Zur Erläuterung der verschiedenen Konstrukte wird ein übliches Problem verwendet:
 - Ein Buffer, in den ein Prozess schreibt
 - und aus dem ein anderer Prozess liest
 - Dieser Buffer ist in seiner Größe begrenzt ("bounded")
 - Ist er voll, muss der Produzent warten
 - Ist er leer, muss der Konsument warten

Michael Sonntag Synchronisation 9

"Bounded Buffer": Ohne Synchronisation (1)

- Definition:


```
final static int BUFFER_SIZE=...;
Object[] buffer=new Object[BUFFER_SIZE];
int count=0;
int in=0;
int out=0;
```
- Producer:


```
while(true) {
    while(count==BUFFER_SIZE);
    buffer[in]=nextProduced;
    in=(in+1)%BUFFER_SIZE;
    count++;
}
```

Michael Sonntag Synchronisation 10

"Bounded Buffer": Ohne Synchronisation (1)

- Consumer:


```
while(true) {
    while(count==0);
    nextConsumed=buffer[out];
    out=(out+1)%BUFFER_SIZE;
    count--;
}
```
- Wird jeweils ein ganzer Schleifendurchlauf von Producer oder Consumer ausgeführt, so ist diese Lösung korrekt
 - Aber was passiert, wenn count++ (vorige Seite; P) und count-- (diese Seite; C) gleichzeitig ausgeführt werden?
 - Oder wenn count vor dem Hineinschreiben des neuen Wertes erhöht wird?

Michael Sonntag Synchronisation 11

Semaphor

- Eine Integer-Variable, auf die nur über zwei spezielle Funktionen zugegriffen werden kann:
 - s.wait(): Warten bis der Wert größer als 0 ist und dann um eins verringern
 - s.signal(): Wert um eins erhöhen. Wird der Wert größer als 0, so wird ein wartender Prozess gestartet.
 - Wert<=0: Anzahl der wartenden Prozesse
 - Wert>0: Anzahl der Prozesse die ein wait ausführen können ohne blockiert zu werden
- Diese Aktionen sind **nicht teilbar** und werden auf jeden Fall zur Gänze ausgeführt
 - Dies wird vom Betriebssystem/Hardware garantiert!

Michael Sonntag Synchronisation 12

Semaphore

- Ein Semaphore kann "auf Vorrat" mit signals versehen werden und kann viele Prozesse gleichzeitig im Wartezustand haben
- Für critical regions wird der Wert auf 1 gesetzt: Ein Prozess kann hinein, alle anderen müssen warten!
`s.wait();`
... critical region
`s.signal();`
- Bei einem Startwert von N wird garantiert, dass höchstens N Prozesse sich gleichzeitig in der critical region befinden
- Andere Verwendungen (wait oder signal alleine) sind auch möglich, aber gut zu überlegen (Deadlocks, ...)!

Michael Sonntag Synchronisation 13

Semaphore Beispiel

- Lösung von Bounded Buffer mit Semaphoren
- Initialisierung:
`Semaphor empty=new Semaphor(BUFFER_SIZE);`
`Semaphor full=new Semaphor(0);`
`Semaphor mutex=new Semaphor(1);`
- Producer:
`empty.wait();`
`mutex.wait();`
`buffer[in]=nextProduced;`
`in=(in+1)%BUFFER_SIZE;`
`count++;`
`mutex.signal();`
`full.signal();`

Michael Sonntag Synchronisation 14

Semaphore Beispiel

- Consumer:
`full.wait();`
`mutex.wait();`
`nextConsumed=buffer[out];`
`out=(out+1)%BUFFER_SIZE;`
`count--;`
`mutex.signal();`
`empty.signal();`
- "Mutex" ist nötig um sicherzustellen, dass immer nur ein Prozess sich in der critical region befindet
- Full und empty dienen dazu den Füllstand abzubilden und gleichzeitig die Prozesse stillzulegen, bis Platz frei/Daten vorhanden sind

Michael Sonntag Synchronisation 15

Critical region

- Eine globale Variable (=Token), die jeweils nur von einem Prozess "besessen" werden kann
 → Nur innerhalb einer critical region ist Zugriff darauf möglich
- Beispiel:
 → v: *shared Type*;
 → region v do *Statement*;
 » Nur innerhalb von "Statement" kann auf v zugegriffen werden
 » Immer nur ein einziger Prozess kann für eine bestimmte "shared"-Variable in einem region-Statement sein
 » Dies wird von Programmiersprache/Betriebssystem garantiert
 → Alternative: region v when *boolean_expression* do *Statement*;
 » Zusätzlich muss noch der boolsche Ausdruck "Wahr" ergeben
 » Wenn nicht, wird solange mit der Ausführung gewartet, bis er es ist (d. h. ein anderer Prozess muss diese "wahr-machen")!

Michael Sonntag Synchronisation 16

Critical region Beispiel

- Lösung von Bounded Buffer mit critical regions:
- Initialisierung:

```
class BufferObj {
    Object buffer[]=new Object[BUFFER_SIZE];
    int count=0; int in=0; int out=0;
};
shared BufferObj myBuffer=new BufferObj();
```
- Producer:

```
region myBuffer when (count<BUFFER_SIZE) {
    buffer[in]=nextProduced;
    in=(in+1)%BUFFER_SIZE;
    count++;
}
```

→Achtung: Nicht wirklich Java-Code!

Michael Sonntag Synchronisation 17

Critical region Beispiel

- Consumer:

```
region myBuffer when (count>0) {
    nextConsumed=buffer[out];
    out=(out+1)%BUFFER_SIZE;
    count--;
}
```
- Hier entweder busy-waiting oder nicht
 → Je nach konkreter Implementierung!
 → Kommt darauf an, wie die Bedingung geprüft wird

Michael Sonntag Synchronisation 18

Monitor

- Bei einem Monitor wird eine Klasse als solche definiert
- Es dürfen dann nur Klassenmethoden auf die eigenen Variablen zugreifen
 - Und diese dürfen nur auf die eigenen Variablen und Parameter zugreifen, nicht jedoch auf externe Daten!
- Zusätzlich besitzt ein Monitor eine Warteschlange für Threads, die gerne (irgend-)eine Methode des Monitors ausführen möchten
- Es darf immer nur ein einziger Thread eine Methode der Klasse ausführen
 - Ganz egal welche, rekursiver Aufruf, etc.
 - Verschiedene Methoden von verschiedenen Threads aufgerufen → NEIN!

Michael Sonntag Synchronisation 19

Monitor Operationen

- x.wait()
 - Der derzeitige Prozess wird suspendiert
 - » In den Wartezustand gesetzt
- x.signal()
 - Genau ein Prozess wird aus dem Wartezustand herausgeholt und auf "runnable" gesetzt
 - Wartet kein Prozess, ergibt sich keine Änderung
 - » Unterschied zu Semaphore: Dort hat "signal" immer eine Auswirkung!
- x ist entweder der Monitor selber oder eine spezielle Variable (z. B. "condition x;")
 - Je nach Implementierung; bei speziellen Variablen gibt es dann jeweils eine gesonderte Warteschlange

Michael Sonntag Synchronisation 20

Monitor Beispiel

- Lösung von Bounded Buffer mit Monitor:
- Initialisierung:


```
monitor BoundedBuffer {
```
- Producer:


```
{
    if(count==BUFFER_SIZE)
        wait();
    buffer[in]=nextProduced;
    in=(in+1)%BUFFER_SIZE;
    count++;
    signal();
}
```

Michael Sonntag Synchronisation 21

Monitor Beispiel

- Consumer


```
{
    if(count==0)
        wait();
    nextConsumed=buffer[out];
    out=(out+1)%BUFFER_SIZE;
    count--;
    signal();
}
```
- "Überschüssige" signal sind kein Problem
 - Sie sind ohnehin wirkungslos
- Bei signal nichts prüfen: Entweder es warten NUR producer oder NUR consumer (oder gar niemand)
 - Gemischtes Warten (>=1 Producer UND >=1 Consumer warten ist unmöglich!)

Michael Sonntag Synchronisation 22

Implementation in Java Monitor

- Das einzige Synchronisationskonzept das Java direkt unterstützt
 - Seit Java 1.5: Mehr in der Klassenbibliothek (siehe später)
- Leichte Modifikation: Es wird nicht eine Klasse als "Monitor" definiert sondern eine frei wählbare Teilmenge der Methoden
 - Diese können auch auf andere (externe) Variablen oder Objekte zugreifen (Aber Vorsicht!)
 - Kennzeichnen einer Methode als "synchronized"
 - `synchronized (obj) {}` Blöcke um explizit zu synchronisieren
 - Jeweils nur eine Warteschlange (=1 Monitor) pro Objekt
 - » Alle synchronized Methoden eines Objektes besitzen mutual exclusion; keine "geteilten Subsets" davon sind möglich!

Michael Sonntag Synchronisation 23

Implementation in Java Monitor

- wait und wait(long millisSecondsTimeout)
 - Thread wartet bis er aufgeweckt wird bzw. das Timeout abläuft
- notify() zum Aufwecken (=signal()); nur anderer Name)
- notifyAll() um alle wartenden Threads aufzuwecken
- wait und notify können nur aufgerufen werden, wenn der Monitor im Besitz des Threads ist
 - D. h. in synchronized Methoden oder in Blöcken `synchronized(obj)`, wobei obj das zu bearbeitende Objekt ist
 - » Beispiel: `synchronized(obj) { ... obj.wait(); ... }`
 - » Beispiel: `public synchronized void put(...); { ... notify(); ... }`
 - Entspricht "this.notify();"

Michael Sonntag Synchronisation 24

Implementation in Java Critical region

- Meistens in der Form von "Mutex" (=mutual exclusion)
- lock(): Beginn einer kritische Region
- unlock(): Ende einer kritischen Region
- Es kann immer nur ein Prozess den lock besitzen und daher immer nur einer in der critical region sein
- Implementation trivial, bis auf ein extrem trickreiches Problem (sonst ev. mehrere Threads gleichzeitig drin!)

[Mutex.java](#)

Michael Sonntag Synchronisation 25

Implementation in Java Semaphore

- Semaphore-Implementation in Java ist nicht besonders kompliziert
 - Darauf achten, was bei interrupts passiert
 - » Java ermöglicht es, wartende Threads auch zu unterbrechen
 - InterruptedException wird ausgeworfen
 - Nicht möglich: "Abschießen" von Threads
 - » Wäre zwangsweises beenden von einem anderen Thread aus
 - » Thread-Kommunikation stattdessen erforderlich
 - Thread muss sich freiwillig selbst beenden

[Semaphore.java](#)

Michael Sonntag Synchronisation 26

Java Threads

- Java unterstützt Threads
 - Threads besitzen Prioritäten
 - » Betriebssystem-unabhängig!
 - Threads können hierarchisch gruppiert werden
 - » Verwaltungs-Vereinfachung; keine Auswirkung auf Scheduling
- Zustandsmodell wie bei Prozessen, aber Methoden zur Überführung stark eingeschränkt!
 - Stop, suspend, resume, destroy: Deprecated und wohl bald endgültig beseitigt
 - » Diese Methoden können zu enormen Problemen führen
 - "destroy" war z.B. überhaupt nie implementiert!

Michael Sonntag Synchronisation 27

Java Threads Zustände

- Unabhängig vom Prozess-Modell (running, waiting, suspended, ...) können Java Threads weitere Zustände besitzen:
 - Alive: Gestartet und noch nicht beendet
 - » =running, waiting oder suspended
 - Daemon: "Hintergrund-Thread"
 - » Die JVM wird beendet, sobald nur noch Daemon-Threads vorhanden sind
 - Interrupted: Der Thread war im Wartezustand und wurde unterbrochen
 - » Achtung: Die Abfrage setzt den Status zurück!

Michael Sonntag Synchronisation 28

Java Threads Methoden

- Erzeugen: Jeder Thread benötigt einen Namen
 - Keiner festgelegt, automatisch eindeutiger erzeugt
- Thread.currentThread(): Derzeit ausgeführter Thread
- join: Warten (ev. bestimmte Zeit) auf Ende des Thread
- Thread.sleep: Wartezustand für bestimmte Zeitspanne
 - Achtung: Monitore werden nicht freigegeben!
- Thread.yield(): Neu-Scheduling auslösen, wobei dieser Thread (hoffentlich) übergangen wird
 - D.h. freiwillig die CPU abgeben
 - Von Bedeutung ev. für Optimierungen
 - » Keinerlei Garantien!
- Hauptmethode: public void run()
 - Ausführen der eigentlichen Arbeit

Michael Sonntag Synchronisation 29

Java Threads wait vs. sleep

- Beides legt den aktuellen Thread "schlafen"
- Monitore:
 - sleep behält alle Monitore
 - wait verliert den "eigenen", behält aber alle anderen
- Zeitablauf:
 - Wait kann sofort, nach festgelegter Zeit oder nie zurückkehren
 - Sleep ist immer zeitlich begrenzt
- Beide Wartezustände können durch "interrupt" unterbrochen werden
 - Sofortige Rückkehr mit "InterruptedException"

Michael Sonntag Synchronisation 30

Java Threads Beispiel

```

public class MyThread extends Thread {
    public void run() {
        while(!Thread.interrupted()) {
            doSomething();
        }
    }
}

MyThread t=new MyThread();
t.start();
...
t.interrupt();
...
t.join(5000);
    
```

Ausführung von run() beginnt
"Signal", dass Thread beendet werden soll (läuft aber noch weiter!)
Warten auf tatsächliches Ende (5 Sekunden lang)

Michael Sonntag Synchronisation 31

Java Threadgroups

- Hierarchische Gruppierung von Threads
→ Baumstruktur
- Dient einfacherer Verwaltung
- Sicherheits-Unterstützung
→ Ein Thread kann nur auf seine eigene ThreadGroup sowie auf untergeordnete Gruppen zugreifen
- Geringe Bedeutung, außer zur inhaltlichen Strukturierung (interne Verwaltung bei vielen Threads)

Michael Sonntag Synchronisation 32

ThreadLocal Variablen

- Einmal deklariert, werden für jeden Thread separat angelegt
→ Normal: private static Variable
- Jeder Thread der darauf zugreift, sieht seinen eigenen Wert
→ Achtung: 1 Variable ⇔ x Werte!
→ "Lazy initialization": Ohen Zugriff wird nichts angelegt
- Verwendung: Klassen reentrant machen
→ "Stateful class" soll "thread-safe" werden
→ Alternative: Daten in den Thread hineinverlagern
» Ist in diesen Fällen aber meist schwierig!
- Bsp: Ein Objekt wird von mehreren Threads verwendet
→ Für jeden Thread aber ein anderer Zustand

Michael Sonntag Synchronisation 33

Java 1.5: Synchronisation

- Spezielle Synchronisationsklassen neu eingeführt
→ java.util.concurrent: Allgemein verwendbare Klassen
→ java.util.concurrent.atomic: Klassen für atomaren Zugriff auf einzelne Variablen bzw. Arrays
→ java.util.concurrent.locks: Hilfsklassen und grundlegende Primitive
- Enthalten viele nützliche Klassen, teilweise auch mit Erweiterungen
→ Beispiel: Semaphor unterstützt auch "fair waiting"
» Wer als erster wartet, kommt als erster heraus
» "Normales" wait(): Keine derartige Garantie ("irgendeiner")!

Michael Sonntag Synchronisation 34

java.util.concurrent: Überblick

- Synchronisierte Queues, Hashmaps, Lists, Sets
- Unterstützung für Thread-Pooling und Scheduling
→ Executor: Ähnlich einem Thread
» Entkopplung der Aufgabe davon, wie sie tatsächlich ausgeführt wird (wann, welcher Thread, ...)
→ Können auch asynchron ausgeführt werden
- Semaphore: Klassisch, zusätzlich Fairness möglich
- CountdownLatch: Einmaliges warten, bis eine gewisse Anzahl von Threads einen Punkt erreicht haben, dann werden alle gleichzeitig freigesetzt ("Rendezvous")
- CyclicBarrier: Wie CountdownLatch, aber rücksetzbar
- Exchanger: Synchronisationspunkt für zwei Threads zum Austauschen zweier Datenobjekte

Michael Sonntag Synchronisation 35

java.util.concurrent.atomic: Überblick

Achtung: Hier werden teilweise generische Typen verwendet (auch neu in Java 1.5)!

- Atomar: Zugriff darauf ist "unteilbar"
→ Race conditions können innerhalb nicht auftreten
→ Zwischen verschiedenen Zugriffen jedoch immer noch!
- Atomare Datenelemente: Boolean, Integer, Long
- Atomare Arrays: Integer, Long
→ Atomizität betrifft nur den Zugriff auf einzelne Elemente!
- Atomare Referenz, Referenz-Array und Referenz mit einem Markierungsbit oder einem Markierungs-Integer
→ Referenz auf einen beliebigen Datentyp
» Typischer aufgrund generischen Typs

Michael Sonntag Synchronisation 36

 **java.util.concurrent.atomic:
Überblick**

- Über Reflection und Spezialklassen Atomizität für Zugriffe auf Elemente existierender Klassen möglich
 - Element muß als *volatile* definiert sein
 - Funktioniert nur für Zugriffe über diese Klasse
 - » Dann immer diese Zugriffsart verwenden; sonst Probleme!
- Beispiele (AtomicInteger):
 - `addAndGet`: Wert dazuaddieren, Ergebnis zurückgeben
 - `get`, `set`, `compareAndSet`, `decrementAndGet`, `incrementAndGet`, `getAndAdd`, `getAndSet`, ...
 - `weakCompareAndSet`: Wenn der Wert gleich einem erwarteten Wert ist, wird er auf einen neuen Wert gesetzt
 - » Kann jederzeit (auch ohne sichtbaren Grund) fehlschlagen
 - » Implementierung viel effizienter

Michael Sonntag Synchronisation 37

 **java.util.concurrent.locks:
Überblick**

- `ReentrantLock`: Wie der interne Monitor bei Verwendung von *synchronized*, aber mit Zusatzfunktionen
 - Eigene Queue für Fairness
- `ReentrantReadWriteLock`: Beliebige viele gleichzeitige Lesezugriffe möglich, aber zu jedem Zeitpunkt nur ein einziger Schreibzugriff (und währenddessen keine Lesezugriffe)
 - Reentrant, Fairness möglich, Write → Read möglich
 - » Um Werte während der Änderung abfragen zu können
 - Read → Write unmöglich
 - » Read lock freigeben, dann neu Write lock anfordern
- Interne Hilfsklassen

Michael Sonntag Synchronisation 38