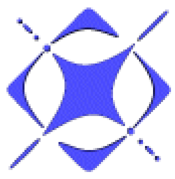
Fachhochschule
Münster University of
Applied Sciences

Java Threads

Mutex, Race Condition, Deadlock, Starvation und Co..

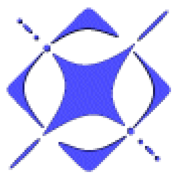
Prof. Dr. Nikolaus Wulff

Java Threads

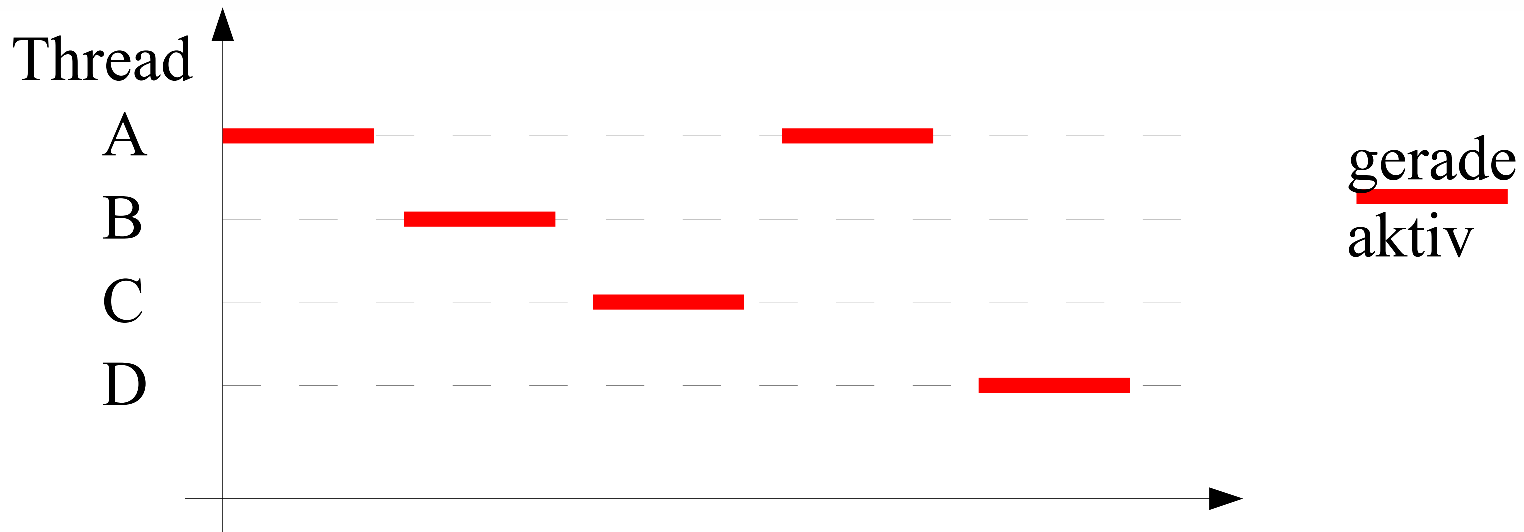


- Threads sind „leichtgewichtige Prozesse“. Sie benötigen keine Betriebsystemressourcen wie ein echter Prozess. Die OS-Ressourcen hält die JVM...
- Jeder Thread durchläuft seriell seinen Programmcode.
- Ein Thread kann blockieren (z.B. System.in.read()) oder aber auch gezielt „schlafen“ gelegt werden:
  - z.B. this.wait(1000) oder Thread.sleep(1000)
- Ein Thread kann einen anderen wieder aufwecken:
  - this.notify() oder this.notifyAll()
- Damit dies funktioniert, muss der Thread in einer synchronisierten „Critical Section“ laufen.

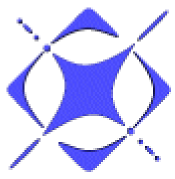
Zeitlicher Ablauf von Threads



- Auf Einprozessormaschinen laufen die Threads scheinbar – auf Multicoreprozessoren real – parallel.
- I.A. wird es immer mehr Prozesse & Threads als Prozessoren geben.
- Jeder Thread erhält ein Zeitfenster zum Laufen.



Multitasking Paradigma



- Mehrere Programme (Prozesse) laufen unabhängig voneinander, können jedoch mit einander kommunizieren.
  - Z.B. im Dateimanager wird per Drag & Drop eine Datei auf Notepad/Acrobat Reader etc. abgelegt und dort geöffnet...

Betrieb-System

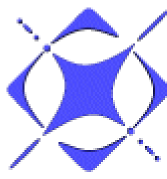
Anwendung #1 → Lokaler Speicher

Anwendung #2 → Lokaler Speicher

Anwendung #3 → Lokaler Speicher

Shared Memory

Multithreading Paradigma



- Eine der Anwendungen ist die JVM, die ihrerseits mehrere Threads verwaltet und synchronisiert.

Betrieb-System

Anwendung #1 → Lokaler Speicher

Java JVM → Lokaler Speicher

JVM

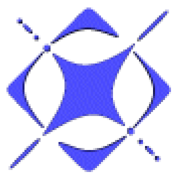
Thread #1 → Lokale Variablen

Thread #2 → Lokale Variablen

Thread #3 → Lokale Variablen

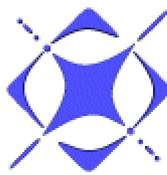
Globale Variablen

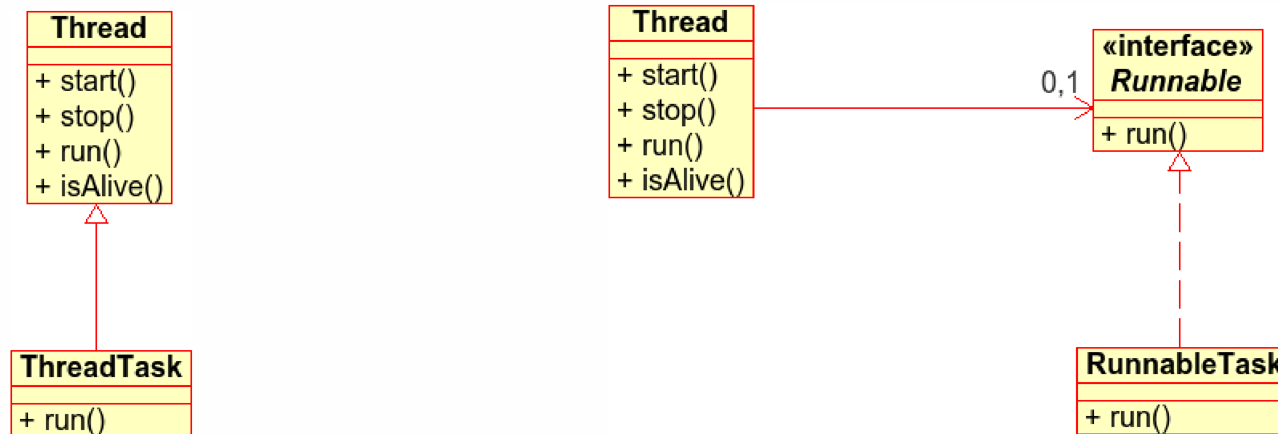
Eigene Threads



- Java Threads sind Instanzen der Klasse Thread.
- Diese werden mit der „start“ Methode gestartet, ab dann wird vom Thread dessen „run“-Methode ausgeführt, die entsprechend überladen werden muss.
- Alternative kann ein Thread mit einem „Runnable“ Objekt parametrisiert werden, dessen „run“-Methode dann ausgeführt wird.
- Der Thread „lebt“ solange bis die run Methode beendet ist, entweder wg. einer Exception oder des natürlichen Endes.

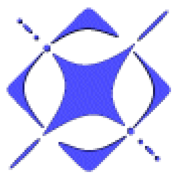
Thread Verwendung





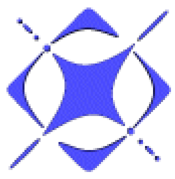
- Zwei Möglichkeiten einen Thread zu verwenden: Durch Vererbung oder durch Parametrisierung mit einem Runnable Objekt.
- In beiden Fällen muss eine geeignete run-Methode implementiert werden.

Thread Life-Cycle



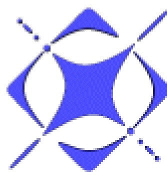
- Threads werden in Java als Instanzen der Klasse java.lang.Thread implementiert.
- Der Thread Lebenszyklus wird geregelt durch die Methoden:
  - *start*: Beginn der asynchronen Ausführung, diese Methode startet die run Methode der jeweiligen Thread Instanz.
  - *run*: Die eigentliche Aufgabe des Thread wird in dieser Methode geleistet. Sobald diese Methode beendet wird, entweder durch return oder Exception ist der Thread „tot“.
  - *stop*: Beendet die run Methode und stoppt den Thread. Diese Methode soll nicht mehr verwendet werden, da sie nicht thread-safe ist...

Threads als FSM



- Threads können (scheinbar) gleichzeitig parallel auf einem Prozessor laufen.
- Hierbei hat jeder Thread seinen eigenen Stack für lokale Variablen und muss sich den Code und Datenbereich der JVM mit allen anderen Threads teilen.
- Java Threads werden als Zustandsautomat modelliert.
- Es gibt die fünf wichtigen Zustände CREATED, TERMINATED, RUNNING, RUNABLE und NONE-RUNABLE, die durch geeignete Statusübergänge mit Hilfe von Thread Methoden erreicht werden können.

Thread Statusdiagramm



Aktiv

Schlafend

Tot

CREATED

start

RUNNING

run

sleep | suspend

NONE-RUNNABLE

suspend

yield

dispatch

resume

stop

stop | end

stop

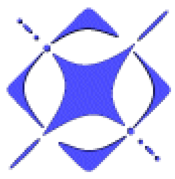
TERMINATED

stop

RUNNABLE

- Statusübergänge werden durch entsprechende Methoden der Thread Klasse ausgelöst.

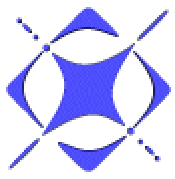
Threadsteuerung



- Methoden zum Steuern von Threads sind in den Klassen Objekt und Thread zu finden:
- Object: wait, notify, notifyAll
- Thread: sleep, interrupt, join, yield, ~~destroy~~, ~~stop~~, ~~suspend~~, ~~resume~~
- Einige Methoden sind „***Deprecated***“ und sollen nicht mehr verwendet werden. D.h. einige der Thread-Stati müssen ohne obige Methoden erreicht werden! Infos hierzu liefert die JDK Dokumentation unter:

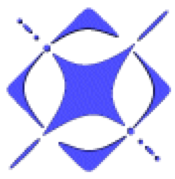
  Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?

Ressourcen



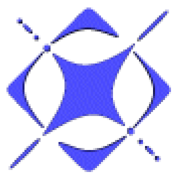
- Programme verwenden Ressourcen, diese lassen sich grob in drei Kategorien unterteilen:
  - sharable / teilbar
  - consumable / verbrauchbar
  - serially resusable / exklusiv wiederverwendbar
- In diesem Kontext ist die letzte Gruppe interessant. Nur ein Teilnehmer (≡ Prozess oder Thread) zur Zeit darf diese verwenden.
- Typische Beispiele sind: Datenbank- und Socketverbindungen, der Drucker, die Festplatte,...

Problematisches Teilen



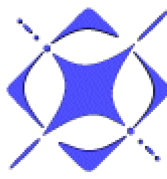
- Race Condition / Inferenz
  - Verschiedene Teilnehmer manipulieren quasi gleichzeitig, bzw. zeitlich überlappend, die selbe Ressource. Um dies zu verhindern werden Sperrmechanismen, wie Critical Section, Mutex und Semaphore, eingeführt.
- Deadlock / Verklemmung
  - Die Sperren können dazu führen, das sich zwei Teilnehmer derart blockieren, das keiner von beiden weiter ausgeführt werden können...
- Life Lock / Starvation / Verhungern
  - Massnahmen gegen Deadlock führen bei ungerechter Verteilung dazu, dass einige Teilnehmer nie die Ressource erhalten...

Vier Bedingungen zur Verklemmung



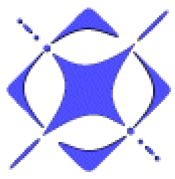
- Vier Bedingungen sind notwendig, damit es zu einer Verklemmung kommt:
  - (1) **Wechselseitiger Ausschluss**: Nur ein Teilnehmer zur Zeit hat exklusiven Zugriff auf Ressource A.
  - (2) **Halten und Warten**: Der Teilnehmer hält und blockiert Ressource A während er auf Ressource B wartet.
  - (3) **Keine Verdrängung**: Der Teilnehmer kann nur selbst die Ressource A wieder freigeben, ist er nicht gutmütig gibt es keine Chance ihm diese zu entziehen
  - (4) **Zyklische Wartesituation**: Die Teilnehmer können in eine zyklische Abhängigkeitssituation kommen.

Vier Strategien gegen Verklemmung

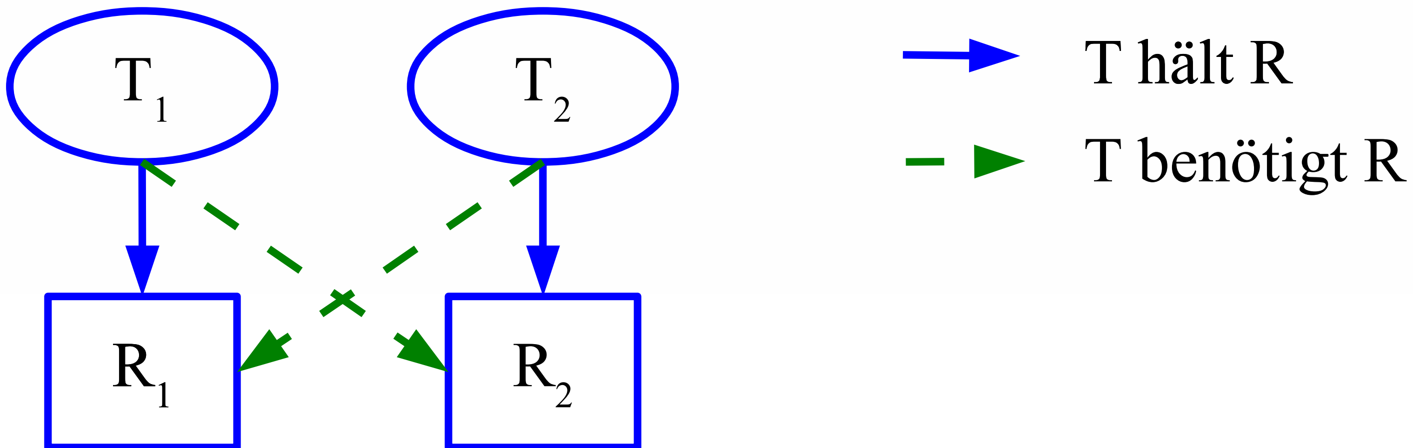


- Vorbeugen
  - Durch Einführen einer Hierarchie und Vorgabe diese immer in gleicher Reihenfolge anzufordern wird (4) durchbrochen.
- Vermeiden
  - Einem Teilnehmer wird verboten eine Ressource zu belegen, nur um anschließend zu merken das noch eine weitere fehlt. Dadurch wird (2) durchbrochen.
- Entdecken
  - Analyse ob es zur Situation (4) kommen kann.
- Auflösen
  - Kommt es zu einer Verklemmung wird die Ressource wieder entzogen und somit (3) durchbrochen.

Teilnehmer-Ressourcen Graph

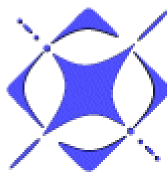


- Teilnehmer-Ressourcen Abhängigkeiten lassen sich als gerichteter Graph präsentieren:

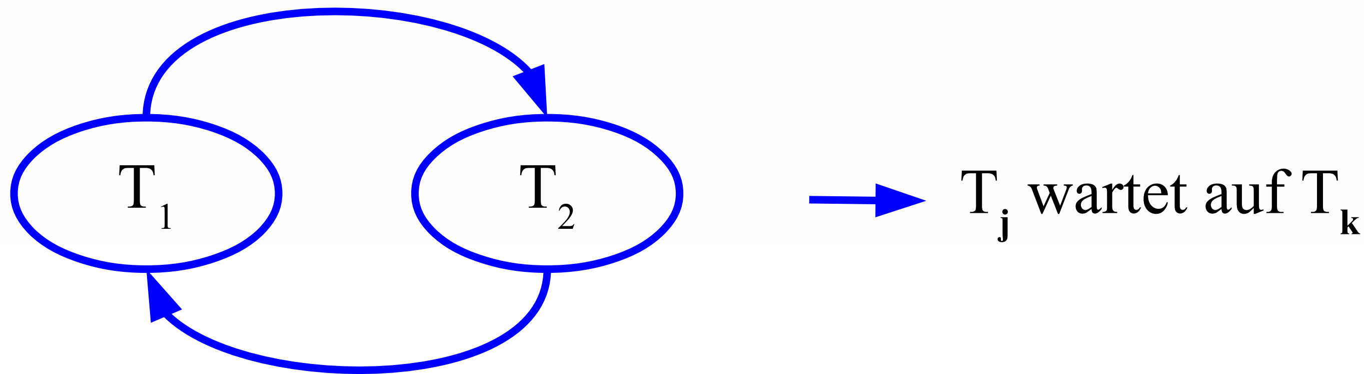


- Graph einer zyklischen Abhängigkeit: T_1 hält R_1 und wartet auf R_2 während T_2 auf R_1 wartet und R_2 hält.

Wait-For Graph

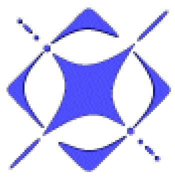


- Abstraktion der Ressourcen zeigt die reine Abhängigkeit der Teilnehmer voneinander:



- An Hand eines solchen WFG Graphen lassen sich zyklische Abhängigkeiten identifizieren.

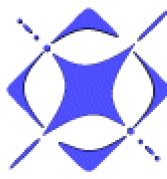
Identifizierung von Zyklen



- Es gibt verschiedene Möglichkeiten einen Zyklus in einem Wait-For Graphen zu entdecken:

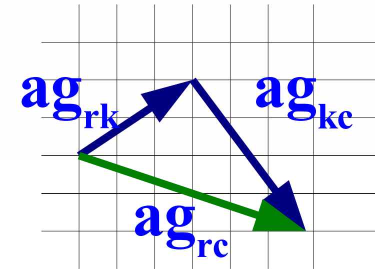
1) **Auswertung der Adjazenzmatrix** A_G und deren Transitiven Hülle R_G. Gibt es in der Diagonalen von R_G Einsen, so liegt eine Verklemmung vor.
2) **Rekursive Reduktion des Wait-For Graphen**. Bleiben nach der Reduktion noch Teilnehmer T über so liegt eine Verklemmung vor.

Warshall Algorithmus



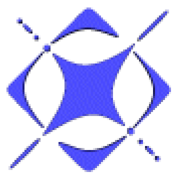
- Die Bestimmung von $R_G = \cup B^k$ erfordert eine aufwendige Berechnung aller Potenzen von A_G.
- Einfacher ist der Algorithmus nach Warshall (1962):

```
for(k=0; k<N; k++)
    for(r=0; r<N; r++)
        if( ag[r][k]==true) {
            for(c=0; c<N; c++)
                ag[r][c] |= ag[k][c];
        }
```



- Die Elemente ag_{rc} der Matrix A_G werden als boolsche Variablen mit Werten {false,true} betrachtet.

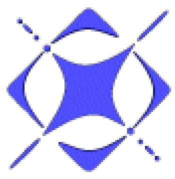
Rekursive Reduktion des WFG



- Der folgende Pseudocode eliminiert rekursiv alle Teilnehmer T_k die ordnungsgemäß terminieren, d.h. keine Abhängigkeiten haben und somit ihre Ressourcen freigeben.

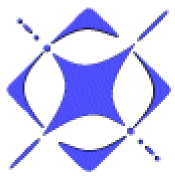
```
for(; ;) {
    find node Tk with no outgoing edges;
    if( no such node )
        break;
    erase Tk and all incoming edges;
}
if (any Tk are left)
    there is a deadlock;
```

Thread Speicherveraltung



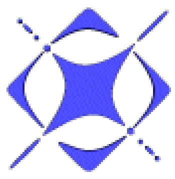
- Java Threads nutzen wo möglich die Fähigkeiten der modernen Multi-Core CPUs aus.
- CPUs verwenden z.B. schnelle L1und L2 Caches für Speicherzugriffe, dies wird auch in Threads realisiert.
- Threads verwenden lokale Kopien globaler Variablen, d.h. Modifikationen der Variablen in einem Thread sind im Anderen u.U. nicht (sofort) sichtbar!
- Zur atomaren Synchronisation von Primitiven gibt es u.A. das Java Schlüsselwort ***volatile***.
- Größere atomare Code Abschnitte werden durch ***synchronizied*** Blöcke geschützt.

Java Thread Synchronisierung



- Shareable Ressourcen müssen über Thread-Grenzen hinweg synchronisiert werden.
- In der Java Sprache sind hierzu das ***synchronizied*** Schlüsselwort sowie die zugehörigen Methoden ***wait***, ***notify*** und ***notifyAll*** zur Inter-Thread-Kommunikation in der Klasse Objekt vorgesehen.
- Weitere Möglichkeiten sind seit dem JDK 1.5+ in den neuen Paketen ***java.util.concurrent.**** zu finden:
  - Callable<V> Runnable mit Ergebnis vom Typ V.
  - Future<V> Resultat einer asynchronen Berechnung.
  - sowie zahlreiche Implementierungen und Hilfsklassen...

Java Synchonisierungsmechanismen

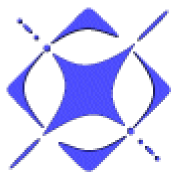


- Jedes Java Objekt besitzt ein zugehöriges Monitor Objekt, das von der JVM als Sperre verwendet wird.
- Die Sperre wird per synchronized Schlüsselwort an dem Objekt gesetzt. Nur ein Thread zur Zeit kann eine solche **Critical Section** exklusiv durchlaufen.

```
public synchronized void foo() {
    // begin critical section
        doSomething_Critical();
}   // end critical section
```

- Intern wird im Bytecode der JVM ein *monitorenter* und *monitorexit* Opcode erzeugt.

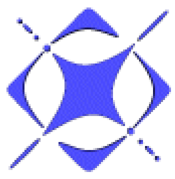
Java synchronized Block



- Die Sperre ist immer mit dem zugehörigen Java Objekt verbunden.
- Dies ist bei einer Methode das „this“-Objekt.
- Oder in einem synchronized Block das explizit angegebene Objekt (this) oder eine Instanz:

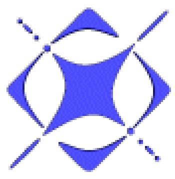
```
public void bar() {
    doSomething_noneCritical();
    // begin critcal section block
    synchronized(anObject) {
      doSomething_Critical();
    } // end critical section block
    doMore_noneCritical();
}
```

Zusammenfassung



- Java Threads laufen analog unabhängigen Prozessen innerhalb der JVM.
- Threads haben intern einen Lebenszyklus der sich als Statusmaschine modellieren lässt und verstehen einen festen Satz von Nachrichten.
- Die Verwendung gemeinsam geteilte Ressourcen muss geeignet geschützt werden.
- Hierzu bietet die JVM „critical sections“ markiert durch das synchronized Schlüsselwort.
- Mit diesem primitiven Sprachkonstrukt lassen sich Threads synchronisieren.

Literatur



- „Java Threads“, S. Oaks und H. Wong, O'Reilly, 1997
  - Das gesamte Buch handelt von Konzepten zur Thread Synchronisierung.
- „Threads und Netzwerk-Programmierung mit Java“, H. Kredel und A. Yoshida, dpunkt, 1999
  - Konzepte zur Parallelverarbeitung für Netzwerk- und Thread-Programmierung.
- „Inside the Virtual Machine“, B. Venners, McGraw-Hill, 1998.
  - Kapitel 20 erläutert die Critical Section innerhalb der JVM
- „Die Programmiersprache Java“, K. Arnold und J. Gosling, Addison-Wesley, 1996