08-Threads-02

Objektorientierte Programmierung | Matthias Tichy

Software Engineering
Programming Languages

universität ulm

Lernziele

- Synchronisierung von Threads
- Worker

Warten, warten, warten

- Manchmal muss ein Threadabschnitt auf eine Nachricht oder eine Information warten, bevor er weiter machen kann
  ➔ Guarded Block

- Wie kann aktives Warten (active polling, busy-wait) vermieden werden?

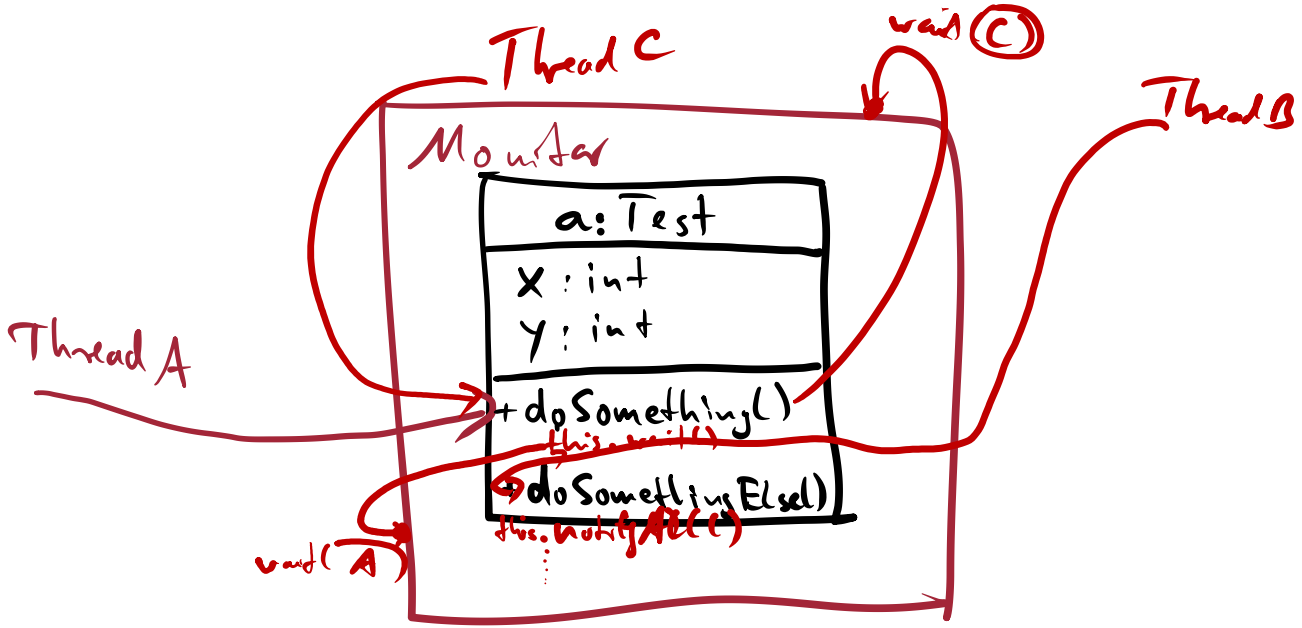
- Objektmechanismen **`wait`** und **`notify`**

Warten, warten, warten

Verschiedenes Warten und Wecken

- Unterschiedliche **`wait`**s:
  - **`wait()`**
  - **`wait(long ms)`**
  - **`wait(long ms, int nanos)`**
- Unterschiedliche **`notify`**s:
  - **`notify()`**
  - **`notifyAll()`**
- **`wait`** und **`notify()`** dürfen nur innerhalb eines Monitors aufgerufen werden

Warten und Wecken



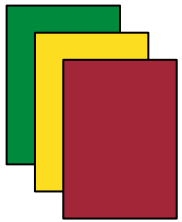
Schlafen oder Warten?

sleep	wait
Methoden der Klasse Thread	Methoden der Klasse Object
Klassenmethoden (static)	Instanzmethoden (non-static)
keine Bedingung für Aufruf	kann nur in gesperrtem Objekt verwendet werden
gesperrte Objekte bleiben gesperrt	Sperre des aktuellen Objekts wird aufgehoben (andere Sperren bleiben gesetzt) nach Ende des Wartens wird die Sperre wieder gesetzt (inkl. warten auf die Freigabe des Objekts)
wartet bis • vorgegebene Zeit oder • bis Interrupt	wartet bis • vorgegebene Zeit oder • bis Interrupt oder • bis Aufwecken durch notify

Das Warten hat ein Ende

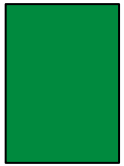
- **`notify`** weckt einen beliebigen auf dieses Objekt wartenden Thread auf
- **`notifyAll`** weckt alle auf dieses Objekt wartenden Threads auf

- Aufwecken bedeutet nicht, dass das gewünschte Ereignis (z.B. Wertänderung einer Variablen) wirklich eingetreten ist
  ➔ immer nochmal Bedingung prüfen!



Welcher Fall ist häufiger?

notify



notifyAll

Typische Anwendung

- Consumer-Producer:
  Ein Thread produziert Daten, ein anderer verarbeitet diese

- Beispiele:
  - Drucker-Spooler bekommt Aufträge von verschiedenen Anwendungen und arbeitet sie ab
  - Event-getriebene Anwendungen bekommen viele Ereignisse von verschiedenen Quellen (Tastatur, Maus) und verarbeiten diese nach und nach

einfaches Beispiel: Parkhaus

- ein Parkhaus hat N freie Plätze
- Autos werden als Threads modelliert

- bei jeder Einfahrt eines Autos, werden die freien Plätze um eins verringert
- bei jeder Ausfahrt eines Autos werden die freien Plätze um eins erhöht

einfaches Beispiel: Parkhaus

- Falls das Parkhaus voll ist, muss auf eine Änderung der freien Plätze gewartet werden ➔ evtl. **`wait`** bei Einfahrt

- Änderung der freien Parkplätze kann nur bei Ausfahrt auftreten
  ➔ **`notifyAll`** bei Ausfahrt

konkretes Consumer-Producer

- ein Thread schreibt Nachrichten in einen Buffer (Producer)
- ein weiterer Thread liest Nachrichten aus diesem Buffer (Consumer)
- Wenn Buffer voll, dann **`wait`** von Producer
- Sobald Nachricht gelesen, **`notify`** von Consumer
- Wenn Buffer leer, dann **`wait`** von Consumer
- Sobald Nachricht geschrieben, **`notify`** von Producer

Konkurrenz!

1. Die Verbraucher c1, c2 und c3 laufen und werden alle blockiert, da kein Wert im Puffer ist
2. Der Erzeuger p1 legt einen Wert im Puffer ab; c1 wird geweckt.
3. p1 läuft weiter und wird ebenfalls blockiert, weil der Puffer noch nicht ausgelesen wurde.
4. p2 wird aktiv und wird sofort blockiert, anschließend wird p3 aktiv und ebenfalls blockiert.
5. Nur noch c1 kann ausgeführt werden; c1 entnimmt den Wert aus dem Puffer; c2 wird aus der Warteschlange geweckt.
6. c1 arbeitet weiter; da kein Wert mehr im Puffer ist, wird c1 blockiert.
7. Der einzige lauffähige Thread ist c2; c2 versucht, einen Wert aus dem Puffer zu lesen und wird blockiert.

Anmerkung zu notifyAll

- **`notifyAll`** weckt zwar zu viele Threads, da Wartebedingung aber sofort wieder überprüft, keine Auswirkung auf Korrektheit des Programms
- ABER: zu häufiges Verwenden von **`notifyAll`** schadet der Effizienz

Anmerkung zu notifyAll

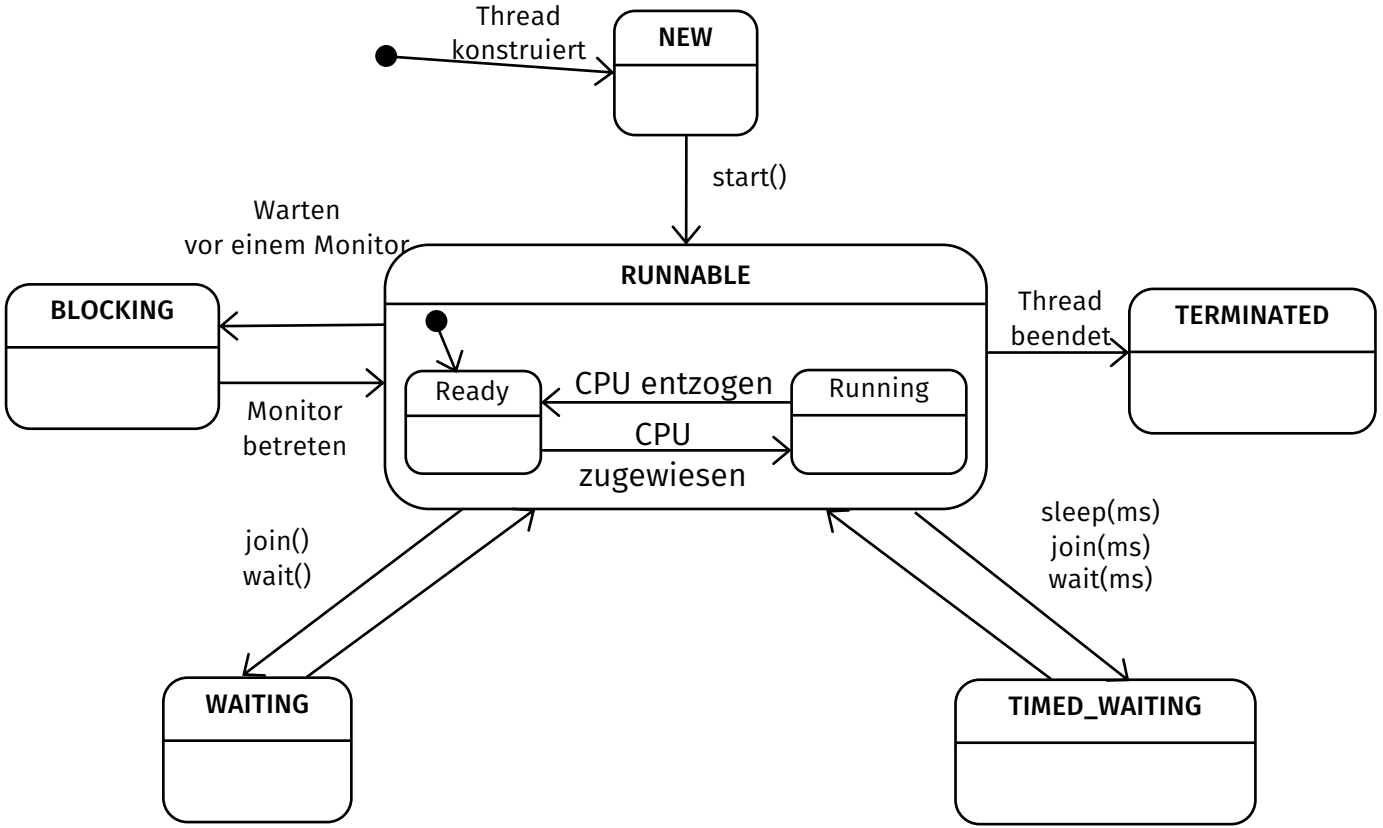
- **`notifyAll`** muss statt **`notify`** verwendet werden, wenn einer der beiden Fälle gilt:
  - Threads mit unterschiedlichen Wartebedingungen; sonst Gefahr, dass „falscher“ Thread geweckt wird (wie im Consumer-Producer-Bsp.)
  - Durch Veränderung des Zustands eines Objekts können mehrere Objekte weiterlaufen
    (Beispiel: Ampel, an der mehrere Autos warten; schaltet die Ampel auf grün, können alle wartenden Autos weiterfahren)

Anmerkung zu Buffer

- In der Realität sollte man **`Buffer`** nicht selbst schreiben, sondern vorhandene Daten-strukturen aus **`java.util.concurrent`** verwenden

- Hier z.B. **`BlockingQueue<>`**

Zustände eines Threads

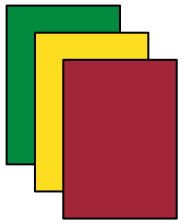


Performanz von Threads

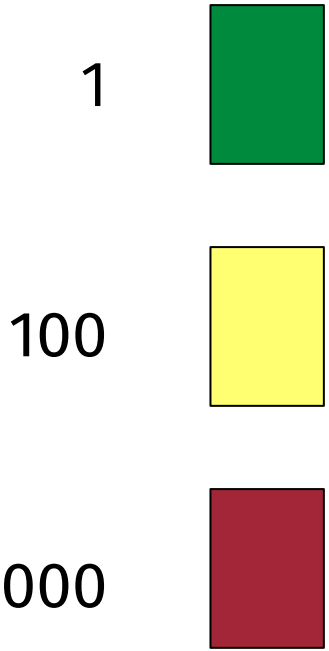
- große Liste von booleschen Werten
- Aufgabe: Wie viele true-Werte kommen vor?

- Aufteilung des Zählens auf eine gewisse Anzahl von Threads
- Zusammenfassen der Teilergebnisse

- Mit wie vielen Threads am schnellsten?



Am schnellsten mit … Threads



stop, suspend, resume

- In der API Funktionen **`stop()`**, **`suspend()`** und **`resume()`** (aus historischen Gründen) vorhanden, aber Benutzung nicht empfohlen
- Probleme mit plötzlichem Abbruch/Pausieren von Threads

- Empfohlen:
  Funktionalität durch Flags nachbauen

stop, suspend, resume

- Zugriffe auf Flags müssen synchronisiert werden
- suspend wird durch **`wait()`** abgebildet, bis resume mit **`notify()`** wieder weckt

- siehe Beispielcode

Bessere Locks

- **`synchronized`** liefert einfache Locking-Mechanismen
- Flexiblere Lock-Möglichkeiten mit **`java.util.concurrent.locks`** package
- Hauptschnittstelle: **`Locks`**

Was kann man besser machen?

- getrennte Read und Write-Locks:
  - mehrere Reader dürfen gleichzeitig zugreifen, so lange keiner schreibt
- Nicht-blockierende Lock-Versuche:
  - `boolean tryLock()`
  - `boolean tryLock(time, timeUnit)`
  - `boolean lockInterruptibly()`
- Information über beteiligte Threads:
  - `Thread getOwner()`
  - `Collection<Thread> getQueuedThreads()`

Atomare Variablen

- Idee: Synchronisation nicht im Ablaufcode, sondern direkt bei den Variablenzugriffen

- Built-In Mechanismus:
  - Schlüsselwort **`volatile`** sorgt dafür, dass Variablenschreibzugriffe direkt an andere Threadkopien durchgeschrieben werden
  - Keine Lösung für ReadAndUpdate-Problematik!

- Wrapperklassen für einfache Datentypen mit höherwertigen Zugriffsfunktionen

Atomare Variablen

Beispiel: **`AtomicInteger`**

- `int addAndGet(int delta)`
- `boolean compareAndSet(int expect, int update)`
- `int decrementAndGet()`
- `void set(int newValue)`
- `int get()`
- `updateAndGet(IntUnaryOperator updateFunction)`

Warum überhaupt ändern?

- Probleme kommen daher, dass Daten geteilt verändert werden
- Mögliche (andere) Lösung:
  keine Daten mehr verändern!
- Bei einer Zuweisung wird immer eine Kopie des Objekts angelegt und mit dieser weitergearbeitet
  ➔ Immutable Objects

Jeder für sich

- Wenn keine echten Änderungen mehr, wie dann Kommunikation zwischen Threads?

- Lösung: Nachrichtenaustausch

➔ Actor-Modell:
  - Jeder Prozess (Actor) hat seinen eigenen Speicherbereich und kommuniziert nur über unveränderbare Nachrichten mit anderen Aktoren
  - bereits seit 1973 Idee bekannt
  - Typische Sprachvertreter: Scala, Erlang, D, Io

Nebenläufigkeit in GUIs

- In JavaFX mehrere Threads aktiv
- Thread-sichere GUI ist nicht einfach zu entwickeln und sehr anfällig für Verklemmungen
- Wie vorgehen, so dass Anwendung immer noch "responsive" bleibt?

Nebenläufigkeit in GUIs

Regeln:

- alle Anweisungen, die etwas in der GUI manipulieren, müssen im "JavaFX Application Thread" ablaufen
- die Abarbeitung von Ereignissen soll typischerweise kurz sein, damit GUI responsive bleibt
- für längere Arbeiten: Worker Threads

Nebenläufigkeit in GUIs

- Ausführung in Application Thread

```
Platform.runLater(new Runnable() {
  public void run() {
    // do something in Application Thread;
  }
});
```

Arbeiten, arbeiten, arbeiten

- Worker Threads implementieren typischerweise das Worker Interface:

```
public interface Worker<V> {
      boolean     cancel()
      Throwable   getException()
      String      getMessage()
      double      getProgress()
      Worker.State    getState()
      String      getTitle()
      double      getTotalWork()
      V           getValue()
      double      getWorkDone()
      boolean     isRunning()
      …
}
```

Arbeiten, arbeiten, arbeiten

- `Worker` bietet einige Vorteile:
  - Rückgabe eines Ergebnisses
  - Rückgabe von Zwischenständen (Progress)

- Implementierungen von Worker-Interface:
  - Task
  - Service (führt Tasks aus)

Concurrency Hell

- (Korrekte) Nebenläufige Programmierung schwierig
- Immer wichtiger werdender Zweig der Softwaretechnik
- Andere Programmiermodelle:
  - Stateless (Funktionale) Programmierung
  - Aktor-Modell
  - Parallele Iteratoren
  - Sprachen nutzen, deren Compiler falschen nebenläufigen Zugriff verbietet → Rust
- Vorlesung im Master:
  - Konzepte für nebenläufige, parallele, verteilte Programmierung

Lernziele

- Synchronisierung von Threads
- Worker