DHBW
Duale Hochschule
Baden-Württemberg

AIRBUS
DEFENCE & SPACE

Nebenläufigkeit

Lukas Abelt
lukas.abelt@airbus.com

DHBW Ravensburg
Wirtschaftsinformatik

Ravensburg
15. Mai 2019

Inhalt

Inhalt

Multithreading

Allgemeines (Vgl. [1] S. 948)

- Moderne Betriebssysteme unterstützen *Multitasking*
- Bedeutet: Mehrere Programme können gleichzeitig laufen
- Man spricht in der Regel von *Nebenläufigkeit*
- Wie diese erreicht wird steuert das Betriebssystem (und ggf. Hardware)
- Auf Mehrkernprozessoren können „echt parallel“ arbeiten
- Auf Einkernsystemen wird eine Parallelität „simulisert“ (*Quasiparallelität*)

Multithreading

Technische Umsetzung (Vgl. [1] S. 948)

- Jeder Prozessorkern kann (in der Regel) zu einem Zeitpunkt einen Prozess bearbeiten
- In der Regel gibt es deutlich mehr laufende Prozesse als Kerne
- Lösung: Aktiver Prozess wird auf den Kernen hochfrequent (im Millisekundenbereich) umgeschalten
- Umschaltung erfolgt durch den *Scheduler*
- Zur Umschaltung der Prozesse gibt es diverse Strategien mit diversen Parametern:
  - Priorität
  - Bearbeitungsdauer
  - „Fail-Count“ – Wie oft wurde der Prozess schon versucht zu bearbeiten

Prozesse

Grundlegende Eigenschaften (Vgl. [1] S. 948)

- Jeder Prozess besteht im Grunde aus:
  - Dem auszuführenden Programmcode
  - Den dazugehörigen Daten
  - Einem *eigenen* (isolierten) Speicherbereich
  - Ggf. Verwendete Ressourcen wie Dateien oder Laufwerke
- Durch die Trennung des Speicherbereichs können Prozesse nicht auf die Daten anderer Prozesse zugreifen!
- Ist doch ein Datenaustausch zwischen Prozessen erforderlich, ist ein spezielle *Shared Memory* Bereich notwendig
- Prozesse können aus mehreren parallelen Threads bestehen $\rightarrow$ Diese können die gleichen Ressourcen nutzen

Nebenläufigkeit

Geschwindigkeitsgewinn (Vgl. [1] S. 949ff)

- Nebenläufigkeit führt in der Regel zu Geschwindigkeitsgewinn
- In Mehrkernsystemen sowieso...
- ...aber auch in Einkernsystemen
- Beispiel: Software zur Erstellung von Datenbank-Reports:
  - Baue ein Fenster auf
  - Öffnen der Datenbank vom Server, lesen der Datensätze
  - Analyse der Daten, Visualisierung des Fortschritts
  - Datei öffnen, Analyseergebnisse in Datei schreiben

Nebenläufigkeit

Beispiel für Geschwindigkeitsgewinn (Vgl. [1] S. 949ff)

- Betrachten wir einmal die parallelisierbaren Abschnitte:
  - Öffnen von Fenster und Datenbank können parallel geschehen
  - Lesen neuer Datensätze und Analyse alter Datensätze kann parallel erfolgen
  - Analyse neuer Datensätze und schreiben von alten analysierten Daten kann gleichzeitig abgearbeitet werden
- Hier auch auf einem Einprozessorsystem großer Leistungsgewinn
- Da die parallelen Prozesse verschiedene *Ressourcen* belasten

Nebenläufigkeit

Beispiel für Geschwindigkeitsgewinn (Vgl. [1] S. 949ff)

- Während auf das Fertigstellen einer Ressource gewartet wird, können Aufgaben bearbeitet werden die andere Ressourcen benötigen:
  - Während der Prozessor ausgelastet ist die GUI zu erstellen kann eine Datei auf der Festplatte geöffnet werden → Dateioperationen benötigen wenig Prozessorleistung, eher durch Festplattengeschwindigkeit begrenzt
  - Während Daten z.B. aus einer Datenbank abgerufen werden wird hauptsächlich die Netzwerkressource belastet → Prozessorleistung kann ggf. anders genutzt werden
  - Parallel zu einer Prozessorlastigen Analyse können bereits analysierte Daten in eine Datei geschrieben werden
- Kurz gesagt: Wir nutzen „Wartezeiten“ von langsamen Operationen zu unserem Vorteil

Nebenläufigkeit

Fazit (Siehe [1] S. 951)

- Nebenläufigkeit muss gut geplant werden
- Insbesondere für Einkernsysteme
- Geschwindigkeitsgewinn nur vorhanden, wenn die parallelen Aktivitäten unterschiedliche Ressourcen nutzen
- Durch Nebenläufigkeit entsteht auch ggf. zusätzlicher Overhead für Synchronisation
- Zum Beispiel, wenn auf ein Teilergebnis gewartet werden muss
- Hier muss insbesondere auf konkurrierende Zugriffe und gegenseitige Wartebedingungen geachtet werden, um *Deadlocks* zu vermeiden

Inhalt

Inhalt

Thread Klasse

Überblick (Vgl. [1] S. 948f)

- Neue Threads werden über die `Thread` Klasse erzeugt und verwaltet
- In der Regel greifen diese direkt auf die Thread Funktionen des Betriebssystems zu
  - „Native Threads“
- JVM Spezifikation schreibt jedoch nicht vor, ob im Hintergrund native Threads genutzt werden oder nicht
- Java garantiert nur, dass die Ausführung der Thread Implementierung korrekt und konsistent funktioniert

Thread Klasse

Grundlegende Methoden (Vgl [2] S. 177ff)

- `Thread` verfügt über zwei grundlegende Methoden:
  - `run()` – Diese Methode wird nebenläufig ausgeführt
  - `start()` – Startet den Thread und führt die `run()` Methode nebenläufig aus
- **Achtung:** Zur nebenläufigen Ausführung muss `start()` aufgerufen werden. Wird `run()` direkt aufgerufen, dann wird der Code zwar auch ausgeführt, jedoch nicht nebenläufig!
- Eine Möglichkeit um nebenläufigen Code auszuführen ist es, eine eigene Unterklasse von `Thread` zu bilden, die die `run()` Methode überschreibt

Thread

Eigene Unterklasse (Vgl. [2] S. 180)

```
1 public class MyThread extends Thread{
2   @Override
3   public void run(){
4     for(int i=0;i<100;i++){
5       System.out.println(i);
6     }
7   }
8 }
```

Thread Klasse

Eigene Unterklasse (Vgl. [2] S. 182)

- Bilden einer Subklasse von Thread ist in der Regel nicht sinnvoll
- Da in der Regel nur das `run` Verhalten definiert werden soll
- Die eigentliche Art und Weise der Nebenläufigen Bearbeitung wird nicht verändert
- Grundsatz beim Bilden von Unterklassen: In der Regel nur, wenn die grundlegende Funktionsweise spezifiziert werden soll
- Weiterer Nachteil: Vererbung ist schon „aufgebraucht“
- Daher bietet Java für Nebenläufigkeit auch ein Interface

Runnable Interface

Grundlegendes(Vgl. [2] S. 177f)

- Das `Runnable` Interface kann auch genutzt werden um nebenläufiges Verhalten abzubilden
- `Runnable` ist ein *funktionales Interface*
  - `void run()` – Spezifiziert den nebenläufig auszuführenden Code
- Für `Thread` gibt es einen Konstruktor, der als Parameter ein `Runnable` akzeptiert:
  - `Thread(Runnable r)` – Das `Runnable` Objekt definiert, welcher Code nebenläufig ausgeführt werden soll
  - Thread muss weiterhin über `start()` gestartet werden
- Da es sich bei `Runnable` um ein funktionales Interface handelt können auch Lambda Expressions verwendet werden

Runnable

Als eigene Klasse

```
1  public class MyRunnable implements Runnable{
2    @Override
3    public void run(){
4      for(int i=0;i<100;i++){
5        System.out.println(i);
6      }
7    }
8  }
9  //Verwendung:
10 Thread t = new Thread(new MyRunnable());
11 t.start();
```

Runnable

Als Lambda Expression

```
1 Thread t = new Thread( () -> {
2     for(int i=0;i<100;i++){
3       System.out.println(i);
4     }
5   });
6 t.start()
```

Runnable

Aufgabe

Implementiert zwei `Runnable` Klassen, die (verschiedene) Daten auf der Konsole ausgeben. Erzeugt dann in eurer `main()` Methode jeweils einen Thread zum Ausführen der Runnables und startet diese direkt nacheinander.
Was für eine Ausgabe erwartet ihr auf der Konsole?
Inwiefern unterscheidet sich die tatsächliche Ausgabe von Eurer Erwartung?
Beobachtet das Verhalten der Ausgaben bei mehrfacher Ausführung des Programms.
Hinweis: Für eine bessere Beobachtung ist es empfehlenswert, die Daten mehrfach über eine Schleife (>100 Iterationen) ausgeben zu lassen

Inhalt

Eigenschaften

Von Threads (Vgl. [2] S. 182f)

- Beispiele für Eigenschaften von Threads:
  - Priorität
  - Zustand
  - Name
- Name lässt sich festlegen im Konstruktor:
  - `Thread(Runnable r, String name)`
  - `Thread(String name)`
- Oder über die entsprechende Funktion:
  - `void setName(String name)`

Eigenschaften

Zugriff auf die Eigenschaften (Vgl. [2] S. 183)

- Analog gibt es auch eine `getName()` Funktion
- *Erinnerung:* Die Methoden gehören zu der `Thread` Klasse!
- Nicht direkt in `Runnable` abrufbar!
- Jedoch lässt sich der Thread in dem gerade gearbeitet wird zurückgeben:
  - `static Thread currentThread()`
  - Aufruf: `Thread t = Thread.currentThread()`
- Wenn der aktuelle Thread mehrfach benutzt wird (z.B. in einer Schleife) sollte dieser vor der Schleife abgerufen und zwischengespeichert werden
  - `currentThread()` Aufruf ist „teuer“

Zustände

Von Threads (Vgl. [2] S. 183f)

- Lebenszyklus des Threads besteht aus verschiedenen Phasen:
  - *Nicht erzeugt*: Der Thread wurde schon definiert (über `new Thread(...)`), aber noch nicht gestartet
  - *Laufend*: Der Thread wurde gestartet (über die `start()` Methode)und wird aktuell durch den Prozessor bearbeitet
  - *Nicht Laufend*: Der Thread wurde gestartet (über die `start()` Methode), wird aber aktuell nicht bearbeitet
  - *Wartend*: Der Thread wartet auf ein bestimmtes Ereignis (Abschluss einer anderen Operation, Freiwerden einer Ressource...)
  - *Beendet*: Der Thread wurde fertig bearbeitet
- Java bildet das über die `Thread.State` Enumeration ab
- Abrufbar über die `getState()` Methode
- Methode `isAlive()` gibt zurück, ob der Thread noch arbeitet oder beendet ist

Zustände

In `Thread.State` (Siehe [2] S. 184)

Zustand	Erklärung
`NEW`	Thread ist erzeugt, aber noch nicht gestartet
`RUNNABLE`	Thread wurde gestartet und läuft in der JVM
`BLOCKED`	Thread wartet auf das freiwerden eines bestimmten Locks um beispielsweise einen synchronisierten Codeabschnitt zu betreten
`WAITING`	Wartet etwa auf ein `notify()`
`TIMED_WAITING`	Zeitgesteuertes Warten beispielsweise durch `sleep()`
`TERMINATED`	Thread wurde beendet

Inhalt

Unterbrechungen

In Threads (Vgl. [2] S. 184f)

- Threads bieten diverse Möglichkeiten zur Unterbrechung
- Simpelste: Die `sleep()` Methode
  - Dieser wird (als `int`) die Zeit in Millisekunden übergeben, die pausiert werden soll
  - Ist **nur** statisch vorhanden → Nur der eigene Thread lässt sich pausieren
  - Kann eine `InterruptedException` auslösen, die über try-catch abgefangen werden muss
  - Gleiche Funktionalität kann auch mit der `TimeUnit` Klasse erreicht werden → Dort eingängliche Definition der Zeit die gewartet wird

sleep()

Codebeispiel

```
1 try {
2   Thread.sleep(5000);
3 } catch(InterruptedException e) {}
4 //Alternativ:
5 try {
6   TimeUnit.SECOND.sleep(5);
7 } catch(InterruptedException e) {}
```

Verzicht

Von Rechenzeit (Vgl. [2] S. 186f)

- Ein Thread kann von sich aus auf Rechenzeit verzichten
- Durch Aufruf von `Thread.yield();`
- Signalisiert dem Betriebssystem, dass auf weitere ggf. bereitgestellte Zeit verzichtet wird und direkt auf einen anderen Prozess geswitched werden kann
- **Nicht deterministisch** – Wann der Thread wieder aufgenommen wird ist unklar
- Daher mit Bedacht zu verwenden
- Oft nur für sehr spezielle Anwendungsfälle wirklich praktikabel

Unterbrechen

Über Interrupt (Vgl. [2] S. 189ff)

- Häufig laufen in nebenläufigen Threads Anwendungen in Endlosschleifen um dauerhaft verfügbar zu sein
  - Beispielsweise Server, die auf ankommende Verbindungen warten
  - Durch die Endlosschleife würden diese auf natürliche Weise nicht beenden
- Deshalb kann von außen an den Thread eine *Anfrage auf Unterbrechung* gestellt werden
- Über die Methode `interrupt()`
- Anders als der Name ggf. vermuten lässt wird der Thread nicht direkt unterbrochen
- Es wird lediglich ein Flag im Thread gesetzt, die dieser eigenständig überprüfen muss
- Thread kann selbst über `isInterrupted()` prüfen, ob er unterbrochen wurde

Interrupt

Codebeispiel

```
1  Thread t = new Thread(()->{
2    while(!isInterrupted()){
3      System.out.println("Running...");
4      try{
5        Thread.sleep(500);
6      } catch (InterruptedException e){
7        interrupt();
8        System.out.println("Interrupt caught while sleep()");
9      }
10   }
11   System.out.println("Ende");
12 });
13 t.start();
14 Thread.sleep(2000);
15 t.interrupt();
```

Daemons

Speziell für Endlosthreads (Vgl. [2] S. 187f)

- Für „Endlosthreads“ im Hintergrund gibt es einen speziellen Modi
- Man kann den Thread als *Daemon* markieren
- Das bedeutet: Der Thread wird solange ausgeführt, solange noch andere Threads (die keine Daemons sind) laufen
- Sollten nur noch Daemon Threads laufen, so werden diese alle beendet
- Markierung als Daemon über die Methode `setDaemon(boolean)`

Rendezvous

Zusammenführen von Threads (Vgl. [2] S. 194f)

- Threads werden unter anderem genutzt, um nebenläufige Berechnungen durchzuführen
- Die Ergebnisse werden an späterer Stelle benötigt
- Zum Verwendungszeitpunkt muss sichergestellt sein, dass die Berechnung abgeschlossen ist
- Dafür wird die Methode `join()` genutzt
- Diese blockiert die Ausführung so lange, bis der entsprechende Thread beendet ist (Die `run()` Methode beendet wurde)

join()

Codebeispiel

```
1  public class JoinThread extends Thread{
2    public int result =0;
3    @Override
4    public void run(){
5      result=1;
6    }
7  }
8  //Anwendung
9  JoinThread t = new JoinThread();
10 t.start();
11 //t.join();
12 System.out.println(t.result);
```

Rendezvous

Überladungen

- Über verschiedene Überladungen lässt sich eine maximale Wartezeit definieren:
  - `join(long millis)` – Definiert eine Wartezeit in Millisekunden
  - `join(long millis, long nanos)` – Definiert eine Wartezeit in Milli- und Nanosekunden
- Nach `join` Aufruf mit Timeout kann über `isAlive()` geprüft werden, ob die Berechnung wirklich abgeschlossen wurde

Inhalt

Executor

Grundlegendes (Vgl. [2] S. 197f)

- Bisher haben wir die Threads immer direkt erzeugt
- Dies hat jedoch einige Nachteile:
  - Bei erzeugen muss der Ausführbare Code (Meist als `Runnable`) übergeben werden → Erzeugen eines Threads und späteres festlegen des nebenläufigen Codes nicht möglich
  - Jeder Thread kann nur einmal ausgeführt werden. Wiederholtes starten führt zu einem Fehler
  - Thread wird immer sofort nach `start()` ausgeführt - Dies ist ggf. nicht zwangsweise gewollt
  - Gesamtanzahl der Threads lässt sich nur schwer begrenzen
- Hilfreich wäre eine Abstraktion, die die *Runnables* von der technischen Umsetzung trennt

Executor

Vorteile

- Bietet eine Methode um beliebige *Runnables* nebenläufig auszuführen:
  - `submit(Runnable r)`
- Java bietet verschiedene Standard-Executor:
  - `ThreadPoolExecutor` – Nutzt eine Sammlung von Threads (*Thread-Pool*) die zur Ausführung wiederverwendet werden können
  - `ScheduledThreadPoolExecutor` – Erweiterung des normalen Pool Executors. Erlaubt die Ausführung zu einem bestimmten Zeitpunkt oder wiederholte Ausführung

Executor

Erzeugung

- Werden am einfachsten erzeugt über die entsprechenden statischen Methoden:
  - `static ExecutorService newCachedThreadPool()` – Liefert einen Thread Pool mit wachsender Größe
  - `static ExecutorService newFixedThreadPool(int nThreads)` – Liefert einen Thread Pool mit fest definierter Größe

Thread-Pools

- Großer Vorteil der Executors: Thread Pools
- Das erzeugen neuer Threads ist sehr aufwändig
- Threads in Thread Pools können wiederverwendet werden
- Dadurch muss nicht für jedes `Runnable` ein neuer Thread erzeugt werden

Callable

Rückgabewerte von Nebenläufigkeiten

- `Runnable` kann keinen Wert zurückgeben – Für Nebenläufige Berechnungen äußerst ungünstig
- Daher gibt es weiterhin das (generische) `Callable<V>`
- Auch wieder ein *funktionales Interface*:
  - `V call()` – Äquivalent zur run Methode im Runnable, jedoch jetzt mit Rückgabewert
- Auch wieder über Lambda Funktionen definierbar

Callable

Als eigene Klasse

```
1 public class MyCallable implements Callable<Integer>{
2   @Override
3   public void call(){
4     return 42;
5   }
6 }
7 //Alternativ:
8 Callable<Integer> c = ()->{return 42;};
```

Future

Ein Blick in die Zukunft

- `Callable` Objekte werden auch über `submit` an den Executor gegeben
- Dieser gibt ein `Future<V>` Objekt zurück
- Darüber lassen sich verschiedene Aspekte prüfen:
  - Ob die Berechnung abgeschlossen wurde
  - Welches Ergebnis zurückgegeben wurde
  - Ob die Bearbeitung abgebrochen wurde
- Das `Future` Objekt kann die Berechnung auch abbrechen

Future

Interface

- `V get()` – Gibt den Rückgabewert des `Callables` zurück
- `V get(long timeout, TimeUnit unit)` – Gibt den Rückgabewert des `Callables` zurück mit einer maximalen Wartezeit
- `boolean isDone()` – Überprüft, ob die Berechnung abgeschlossen ist
- `boolean cancel(boolean)` – Bricht die Arbeit ab
- `boolean isCencelled()` – Gibt an, ob die Arbeit unterbrochen wurde

Quellen I

[1] C. Ullenboom. *Java ist auch eine Insel: Das umfassende Handbuch*. Rheinwerk Computing, 2018. ISBN: 978-3-8362-5869-2.

[2] C. Ullenboom. *Java SE 8 Standard-Bibliothek: das Handbuch für Java-Entwickler ; [Nebenläufigkeit, String-Verarbeitung, Datenstrukturen und Algorithmen, XML, RMI, JDBC, Reflection, JavaFX, Swing, Grafik- und Netzwerkprogrammierung ; JNI, Sicherheit]*. Galileo Computing. Galileo Press, 2014. ISBN: 9783836228749. URL: `https://books.google.de/books?id=D3jSnQEACAAJ`.

Kontakt

- E-Mail: `lukas.abelt@airbus.com`
- GitHub: `https://www.github.com/LuAbelt`
- GitLab: `https://www.gitlab.com/LuAbelt`
- Telefon(Firma): 07545 - 8 8895
- Telegram: LuAbelt