Techniken der Projektentwicklung

Threading & Netzwerkprogrammierung

Ingo Lütkebohle

Termin 13

Themen heute

Kurze Wiederholung von und Praxis-Beispiele zu:

Threading	Synchronisation zwischen Threads
HTTP	Verwendung von HTTP und URLs
Sockets	Netzwerkkommunikation mit TCP-Sockets

Motivation: Threading

Anwendungsfälle für *unabhängige Abläufe mit geteilten Daten*.

GUIs Benutzerinteraktion während zeitraubender Hintergrundaktivitäten

Server Ein Netzwerk-Server, mehrere Clients gleichzeitig

Performance Verteilung von Berechnungen über mehrere CPUs (wenn vorhanden)

Threads erzeugen

Thread Unabhängiger Ausführungs*kontext*

Runnable Das, was ausgeführt wird.

1. In eigener Klasse `Runnable` implementieren
2. `Thread` erzeugen, Instanz der Klasse aus (1) übergeben
3. Thread starten

http://java.sun.com/docs/books/tutorial/essential/threads/

Komplikationen

- Gleichzeitige Verwendung von Datenstrukturen aus mehreren Threads kann zu Konflikten führen, sog. *Race Conditions*
  $\longrightarrow$ Synchronisation notwendig
- Synchronisation beinhaltet warten
  fehlerhafte Synchronisation $\longrightarrow$ sog. *Deadlock*
- „Gleichzeitige“ Ausführung auf einer CPU nur angenähert, Scheduler unterbricht nur an bestimmten Punkten (z.B. IO)
  $\longrightarrow$ `Thread.yield()`

Race Conditions

Textproblem Zwischen zwei atomaren Operationen können andere Threads Daten ändern. Ein Trivialbeispiel:

Thread 1

```
if(val < 0) {
    int exp = 2^val;
    ...
}
```

Thread 2

```
if(val < 0) {
    val = abs(val);
    ...
}
```

Was passiert in der Zeit zwischen Test und Verwendung der Variable?

Einfache Synchronisation

- *Critical Section*
  nur ein Thread kann gleichzeitig in der critical section sein
- In Java: `synchronized` objektbasierte critical sections
  Pro Instanz kann nur ein Thread in *irgendeiner* critical section *dieser Instanz* sein
- `wait`/`notify`
  `wait()` hält an, bis `notify()` aufgerufen wird
  $\rightarrow$ muss in critical section ausgeführt werden

Dining Philosophers

Situation *n* Philosophen sitzen um einen runden Tisch. Zwischen jedem liegt *ein* Besteckteil. Jeder benötigt zwei, um essen zu können. Wie sollten sie sich organisieren?

Ansatz Jeder greift zuerst nach dem rechten Besteckteil, dann nach dem Linken. Hat er beide, nimmt er einen Bissen, legt beide Teile wieder ab und wartet eine zufällige Zeit bevor er von vorne beginnt. Hat er nur ein Besteckteil, wartet er mit diesem in der Hand solange, bis das Andere verfügbar wird.

`/vol/tdpe/share/material/session13/examples/thread/dining/`

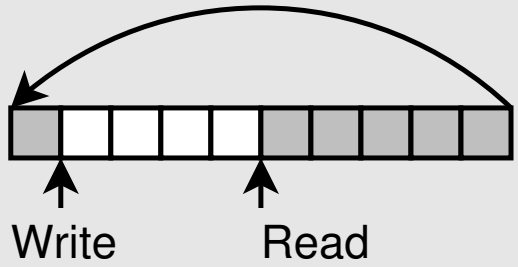
Finde das Problem und implementiere eine bessere Lösung!

Aufgabe: Zugriff auf Datenstruktur synchronisieren

Problem

- Ring-Buffer, zirkulär Schreiben und Lesen
- Mögliche Probleme:
  - Datenverlust (Producer zu schnell)
  - Mehrfaches Lesen (Consumer zu schnell)

Ringbuffer



Aufgabe

1. Definiere Bedingungen für Schreib-/Leseposition
2. Implementiere die nötigen Synchronisationsmechanismen

Siehe `/vol/tdpe/share/material/session13/examples/thread/race/`

HyperText Transfer Protocol

Aufbau

- Details RFC 2616 (v1.1)
- Request/Response-Schema
- einheitliches Nachrichtenformat

Anfrage

- Befehl, Request-URI, Protokoll
- Header
- optional Body

Beispielrequest

```
POST /foo%20?n=v HTTP/1.1
Host: www.techfak.uni-bielefeld.de
Content-Length: 12

Hello World!
```

HyperText Transfer Protocol

Aufbau

- Details RFC 2616 (v1.1)
- Request/Response-Schema
- einheitliches Nachrichtenformat

Anfrage

- Befehl, Request-URI, Protokoll
- Header
- optional Body

Beispielrequest

```
POST /foo%20?n=v HTTP/1.1
Host: www.techfak.uni-bielefeld.de
Content-Length: 12

Hello World!
```

HyperText Transfer Protocol

Aufbau

- Details RFC 2616 (v1.1)
- Request/Response-Schema
- einheitliches Nachrichtenformat

Anfrage

- Befehl, Request-URI, Protokoll
- Header
- optional Body

Beispielrequest

```
POST /foo%20?n=v HTTP/1.1
Host: www.techfak.uni-bielefeld.de
Content-Length: 12

Hello World!
```

HyperText Transfer Protocol

Aufbau

- Details RFC 2616 (v1.1)
- Request/Response-Schema
- einheitliches Nachrichtenformat

Anfrage

- Befehl, Request-URI, Protokoll
- Header
- optional Body

Beispielrequest

```
POST /foo%20?n=v HTTP/1.1
Host: www.techfak.uni-bielefeld.de
Content-Length: 12

Hello World!
```

HyperText Transfer Protocol

Aufbau

- Details RFC 2616 (v1.1)
- Request/Response-Schema
- einheitliches Nachrichtenformat

Anfrage

- Befehl, Request-URI, Protokoll
- Header
- optional Body

Beispielrequest

```
POST /foo%20?n=v HTTP/1.1
Host: www.techfak.uni-bielefeld.de
Content-Length: 12

Hello World!
```

Befehle und Ergebnisse

- Wichtige Befehle:
  - GET ohne Seiteneffekte, idempotent
  - POST Seiteneffekte erlaubt, mit Request-Body
  - HEAD wie GET, liefert nur Header
- HTTP-Status: Numerischer Code + Nachricht
  - 2xx OK, Inhalt folgt (z.B. `200 OK`)
  - 3xx Inhalt woanders (siehe `Location`-Header)
  - 4xx Client-Fehler (z.B. `404 No resource at URL`)
  - 5xx Server-Fehler (z.B. `500 Internal Server Error`)

URI/URL

```
http://www.techfak.uni-bielefeld.de:80/foo%20bar#c09?name=value&n2=v2
```

Bestandteile

1. Protokoll
2. Hostname
3. Port
4. Pfad
5. Fragment
6. Query-String
7. Seperatoren

Hinweise

- Erforderlich: Protokoll, Host, Pfad
- Rest ist optional
- Leerzeichen und viele Andere: encoden!
  → `java.net.URI`
- Java beherrscht mindestens: http, ftp, file
- Länge der URI prinzipiell unbeschränkt
  ab 255 Zeichen Vorsicht

Siehe auch `RFC 1783`: Uniform Resource Locators

URI/URL

```
http://www.techfak.uni-bielefeld.de:80/foo%20bar#c09?name=value&n2=v2
```

Bestandteile

1. Protokoll
2. Hostname
3. Port
4. Pfad
5. Fragment
6. Query-String
7. Seperatoren

Hinweise

- Erforderlich: Protokoll, Host, Pfad
- Rest ist optional
- Leerzeichen und viele Andere: encoden!
  → `java.net.URI`
- Java beherrscht mindestens: http, ftp, file
- Länge der URI prinzipiell unbeschränkt
  ab 255 Zeichen Vorsicht

Siehe auch `RFC 1783`: Uniform Resource Locators

URI/URL

`http://www.techfak.uni-bielefeld.de:80/foo%20bar#c09?name=value&n2=v2`

Bestandteile

1. Protokoll
2. Hostname
3. Port
4. Pfad
5. Fragment
6. Query-String
7. Seperatoren

Hinweise

- Erforderlich: Protokoll, Host, Pfad
- Rest ist optional
- Leerzeichen und viele Andere: encoden!
  → `java.net.URI`
- Java beherrscht mindestens: http, ftp, file
- Länge der URI prinzipiell unbeschränkt
  ab 255 Zeichen Vorsicht

Siehe auch `RFC 1783`: Uniform Resource Locators

URI/URL

```
http://www.techfak.uni-bielefeld.de:80/foo%20bar#c09?name=value&n2=v2
```

Bestandteile

1. Protokoll
2. Hostname
3. Port
4. Pfad
5. Fragment
6. Query-String
7. Seperatoren

Hinweise

- Erforderlich: Protokoll, Host, Pfad
- Rest ist optional
- Leerzeichen und viele Andere: encoden!
  → `java.net.URI`
- Java beherrscht mindestens: http, ftp, file
- Länge der URI prinzipiell unbeschränkt
  ab 255 Zeichen Vorsicht

Siehe auch `RFC 1783`: Uniform Resource Locators

URI/URL

`http://www.techfak.uni-bielefeld.de:80/foo%20bar#c09?name=value&n2=v2`

Bestandteile

1. Protokoll
2. Hostname
3. Port
4. Pfad
5. Fragment
6. Query-String
7. Seperatoren

Hinweise

- Erforderlich: Protokoll, Host, Pfad
- Rest ist optional
- Leerzeichen und viele Andere: encoden!
  → `java.net.URI`
- Java beherrscht mindestens: http, ftp, file
- Länge der URI prinzipiell unbeschränkt
  ab 255 Zeichen Vorsicht

Siehe auch `RFC 1783`: Uniform Resource Locators

URI/URL

```
http://www.techfak.uni-bielefeld.de:80/foo%20bar#c09?name=value&n2=v2
```

Bestandteile

1. Protokoll
2. Hostname
3. Port
4. Pfad
5. Fragment
6. Query-String
7. Seperatoren

Hinweise

- Erforderlich: Protokoll, Host, Pfad
- Rest ist optional
- Leerzeichen und viele Andere: encoden!
  → `java.net.URI`
- Java beherrscht mindestens: http, ftp, file
- Länge der URI prinzipiell unbeschränkt
  ab 255 Zeichen Vorsicht

Siehe auch `RFC 1783`: Uniform Resource Locators

URI/URL

`http://www.techfak.uni-bielefeld.de:80/foo%20bar#c09?name=value&n2=v2`

Bestandteile

1. Protokoll
2. Hostname
3. Port
4. Pfad
5. Fragment
6. Query-String
7. Seperatoren

Hinweise

- Erforderlich: Protokoll, Host, Pfad
- Rest ist optional
- Leerzeichen und viele Andere: encoden!
  → `java.net.URI`
- Java beherrscht mindestens: http, ftp, file
- Länge der URI prinzipiell unbeschränkt
  ab 255 Zeichen Vorsicht

Siehe auch `RFC 1783`: Uniform Resource Locators

URI/URL

```
http://www.techfak.uni-bielefeld.de:80/foo%20bar#c09?name=value&n2=v2
```

Bestandteile

1. Protokoll
2. Hostname
3. Port
4. Pfad
5. Fragment
6. Query-String
7. Seperatoren

Hinweise

- Erforderlich: Protokoll, Host, Pfad
- Rest ist optional
- Leerzeichen und viele Andere: encoden!
  → `java.net.URI`
- Java beherrscht mindestens: http, ftp, file
- Länge der URI prinzipiell unbeschränkt
  ab 255 Zeichen Vorsicht

Siehe auch `RFC 1783`: Uniform Resource Locators

URI/URL

`http://www.techfak.uni-bielefeld.de:80/foo%20bar#c09?name=value&n2=v2`

Bestandteile

1. Protokoll
2. Hostname
3. Port
4. Pfad
5. Fragment
6. Query-String
7. Seperatoren

Hinweise

- Erforderlich: Protokoll, Host, Pfad
- Rest ist optional
- Leerzeichen und viele Andere: encoden!
  → `java.net.URI`
- Java beherrscht mindestens: http, ftp, file
- Länge der URI prinzipiell unbeschränkt
  ab 255 Zeichen Vorsicht

Siehe auch `RFC 1783`: Uniform Resource Locators

Form-Encoding

- „Verpackung“ für Argumentliste
- Ein Name kann mehrmals mit unterschiedlichen Werten vorkommen!
- Name und Wert durch = getrennt
- Parameter durch & verkettet
- Verwendet URL-Encoding **vor** dem Verketten
  → `java.net.URLEncoder`
- MIME-Typ, für `POST`:
  `application/x-www-form-urlencoded`

Beispiel: URL-Retrieval

Beispielstand

Gibt URL, Response-Header und 10 Zeilen Inhalt aus.

Ort `/vol/tdpe/share/material/session13/examples/`

Aufruf `java network.GetURL <url>`

Übung

Aufgabe Ergänze Programm um Übergabe von Parametern!

Aufruf `java network.GetURL <url> name=value n2=v2 ...`

Optional Verwende `HEAD` als Request-Methode!

Wiederholung: TCP

TCP: Transmission Control Protocol

- Stellt Punkt-zu-Punkt Verbindung her
  Eine Seite wartet auf Verbindungen, die andere initiiert sie.
- Zuverlässige Paketzustellung in der richtigen Reihenfolge
  Fehlerbehandlung nur bis Timeout!
- Unabhängige Kanäle in beide Richtungen

Endpunkte

- **IP-Adresse** und **Port**
- Auflösung Name in IP durch Domain Name Service (DNS)
- Ports unter 1024 sind reserviert, z.B. 80 für HTTP
- (Quell-Host:QPort, Ziel-Host:ZPort) eindeutig

Verbindungen aufbauen

- Siehe `java.net.Socket`
- insbesondere `connect`, `getInputStream`, `getOutputStream`
- Streams **blockieren** wenn keine Daten (lesen) bzw. Buffer voll (Senden)
- `Buffered*`-Streams verwenden, um Betriebssysteminteraktion zu minimieren

Verbindungen annehmen

1. IP-Adresse und Port festlegen
   Default: Alle verfügbaren Adressen, beliebiger freier Port
2. `ServerSocket` erzeugen
3. Auf Verbindungen warten (`accept`)
4. Datenaustausch über erzeugten Socket

EchoServer & Client

/vol/tdpe/share/material/session13/examples/network/EchoServer.java

Stand

Server nimmt Verbindungen auf freiem Port an und „echoed“

Aufruf	`java network.EchoServer`
Ausgabe	IP und Port des Sockets
Test	per `telnet` auf angegebenen Port `Ctrl-]` `quit` beendet telnet

Hausaufgabe

- Ergänze den Server um Threading, so dass er mehrere Clients gleichzeitig verarbeiten kann!
- Schreibe einen Echo-Client der von der Standardeingabe liest! Client beenden mit `Ctrl-D`.