Hochschule **Rosenheim**
University of Applied Sciences

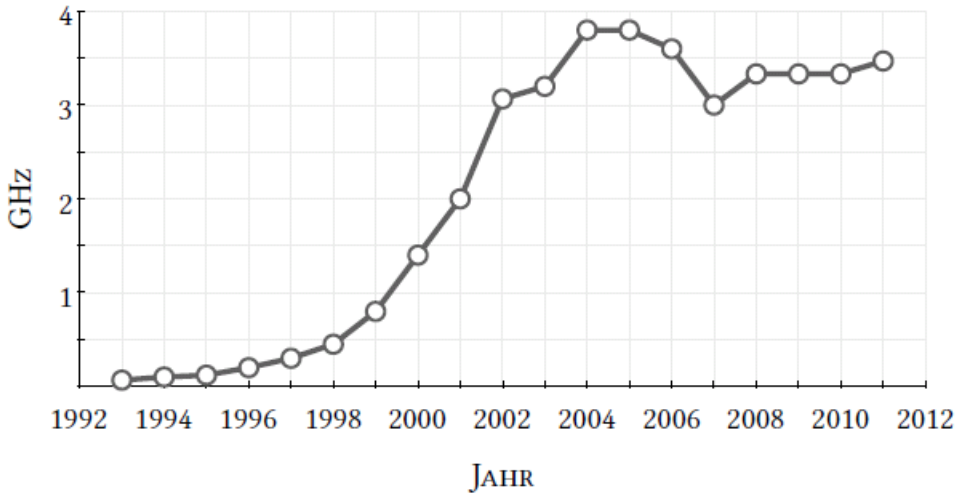
Verteilte Verarbeitung

Kapitel 4

Threadprogrammierung und Nebenläufigkeit

Wozu ist Nebenläufigkeit?

- ***Aktuelle Hardware erfordert nebenläufige Programmierung***
- Abbildung: Höchste Taktrate der jeweils neu erschienenen Intel-Prozessoren *)



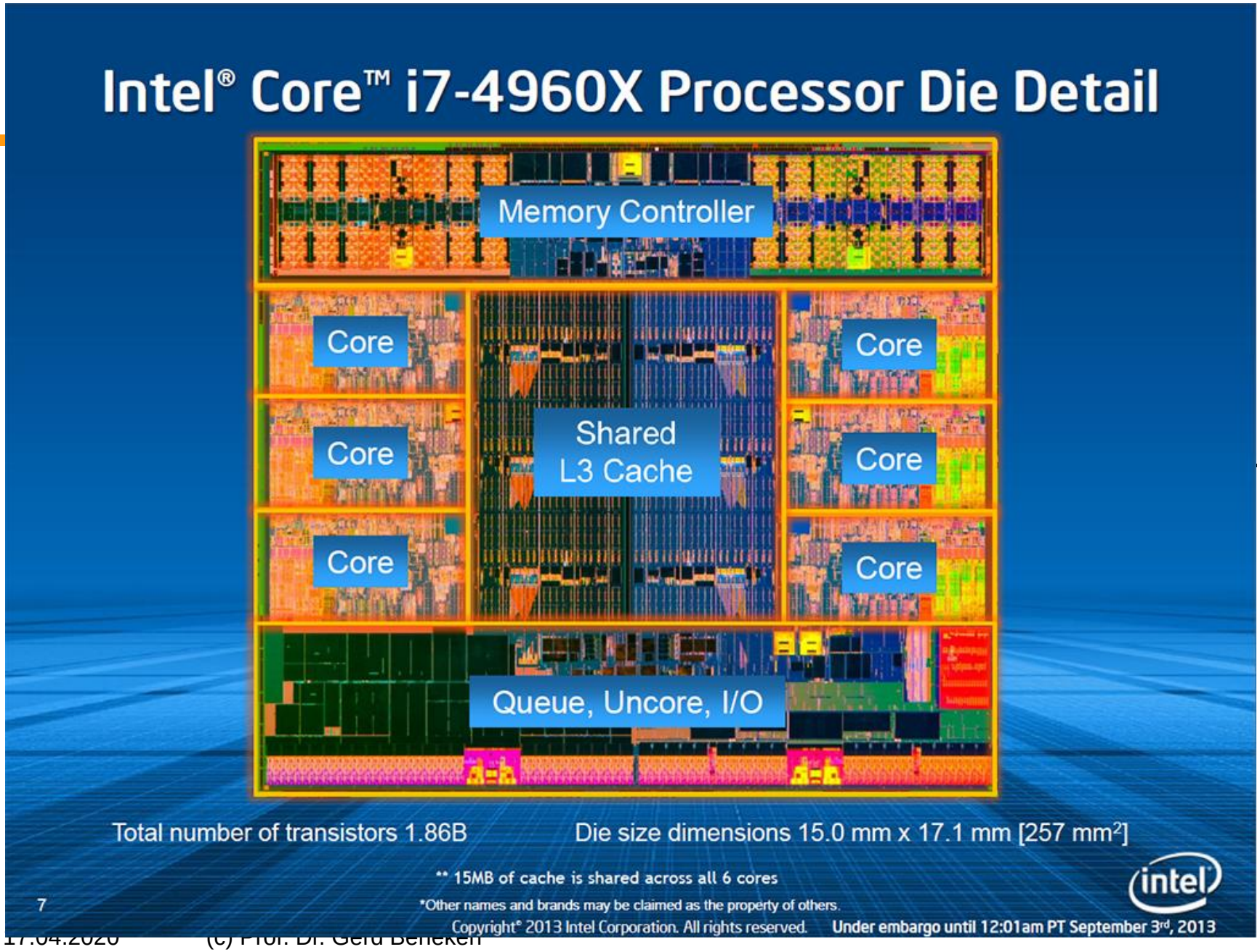
- Grund: Taktfrequenz kann nicht mehr erhöht werden, da Signal in einem Takt nicht mehr durch den Prozessor kommt (Licht schafft in einem Takt bei 3GHz gerade 10 cm)

*) Geklaut von Prof. Sergei Gorlatch, Vorlesung Paralle Programmierung

Nebenläufigkeit und Parallelität

- Parallelität
  - auf mehreren CPUs oder CPU-Kernen oder Hyperthreading
  - mehrere Kontrollflüsse, die ***gleichzeitig*** ausgeführt werden
  - Multiprocessing

- Nebenläufigkeit (-> allgemeiner als Parralelität)
  - Auf einer CPU, die sich die Prozesse / Threads teilen (Multitasking), d.h. Kontrollflüsse verzahnt ***nach einander*** oder
  - auf mehreren CPUs oder CPU-Kernen oder Hyperthreading
  - Paralleler (gleichzeitig) oder verzahnter (nacheinander) Ablauf

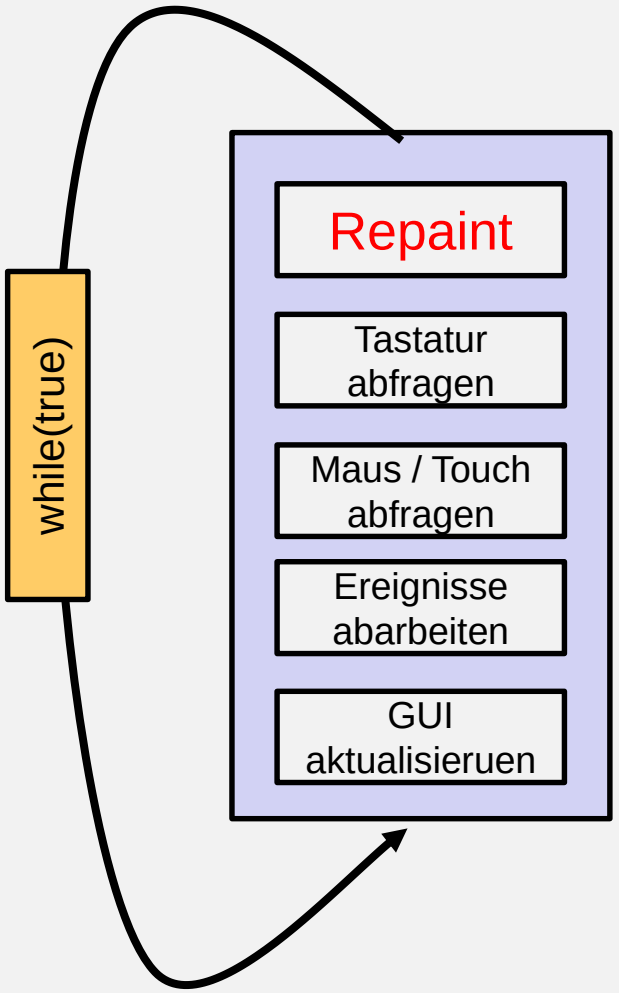

Intel® Core™ i7-4960X Processor Die Detail
Memory Controller
Core
Core
Core
Core
Core
Core
Shared
L3 Cache
Queue, Uncore, I/O
Total number of transistors 1.86B
Die size dimensions 15.0 mm x 17.1 mm [257 mm²]
** 15MB of cache is shared across all 6 cores
*Other names and brands may be claimed as the property of others.
Copyright© 2013 Intel Corporation. All rights reserved.
Under embargo until 12:01am PT September 3rd, 2013
intel

Wozu ist Nebenläufigkeit?

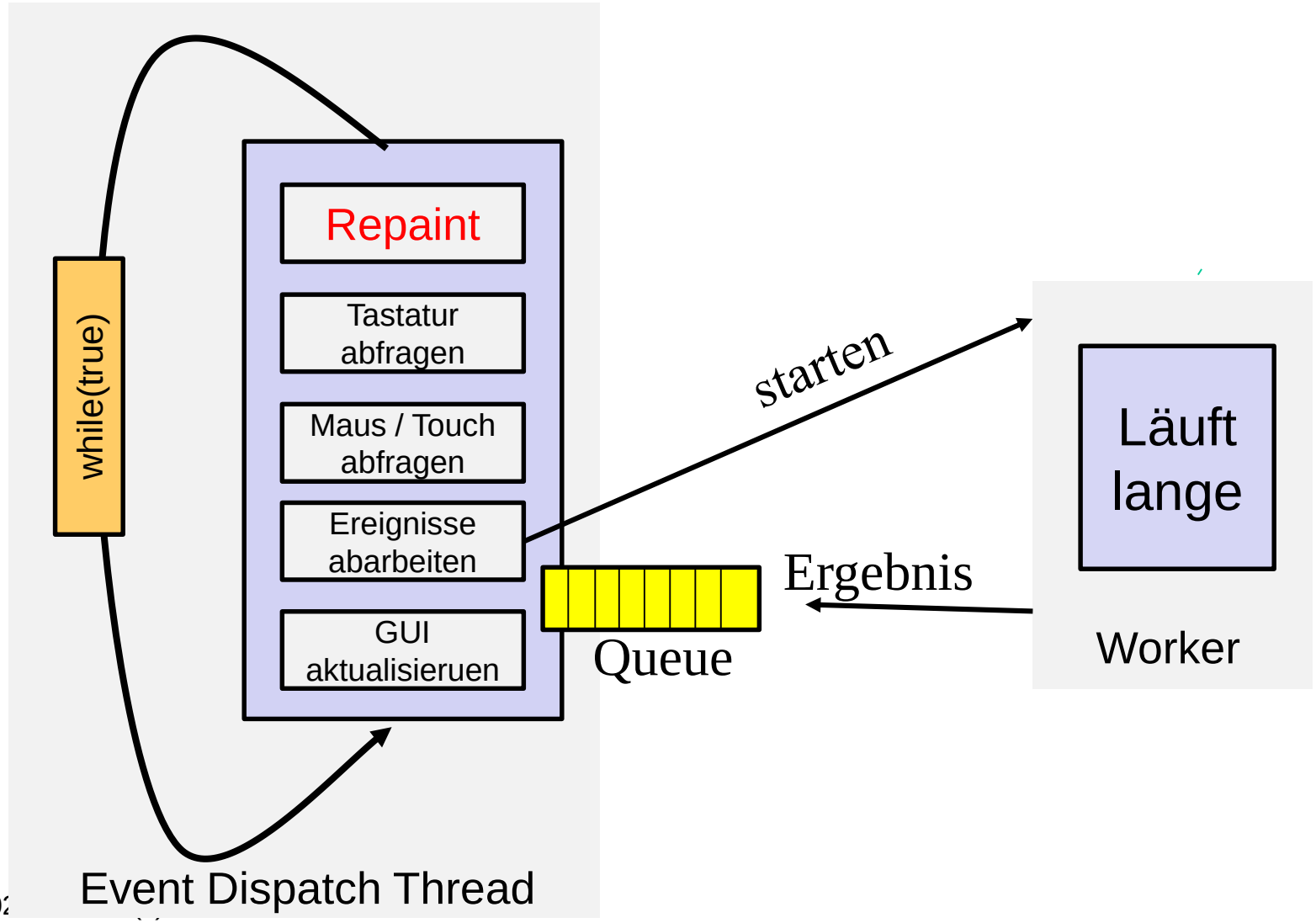
- Verteilte Systeme sind inhärent nebenläufig!
  - Mindestens zwei Prozesse/Threads: Client und Server oder 2 Peers
  - Pro Rechner zwangsläufig ein Prozess/Thread
- ***Server sind idR. multithreaded***
  - z.B. REST-Server mit vielen Clients,
    App.Server kümmert sich um Nebenläufigkeit
- ***Clients sind idR. auch multithreaded***
  - Häufig ein Ereignis-Verarbeitungs-Thread (Swing: EDT,)
  - Client, der in einem Thread auf einen langsamen Server wartet
  - Verlagerung aufwendiger Jobs (z.B. Sortieren, Berechnungen, …) in den Hintergrund als Thread
  - ***Warten auf langsame Ein/Ausgabe in eingenem Thread***
- Java Programme haben Standard-Threads: GC, finalizer, …

GUI-Programmierung inhärent Nebenläufig (und ***technisch*** asynchron)

- GUI Häufig Endlosschleife
  - = Event Dispatch Thread
  - = Fragt Tastatur / Maus / Touch ab
  - = Verarbeitet Ereignisse
- Problem: Ereignis- Bearbeitung dauert zu lange
  - GUI-Toolkit häufig nicht thread-save
  - -> Oberfläche friert ein
  - -> Benutzer glaubt GUI kaputt
- Lösung: Lange laufende Abfragen in eigene Threads



GUI-Programmierung inhärent Nebenläufig (und ***technisch*** asynchron)



Was sind Prozesse und was sind Threads?

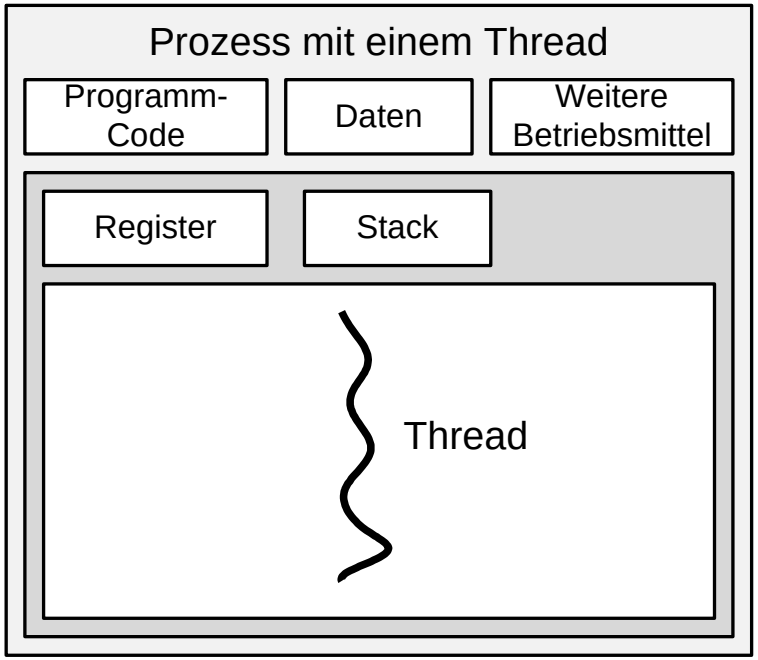
Threads, Prozesse und Tasks (Wiederholung Betriebssysteme)

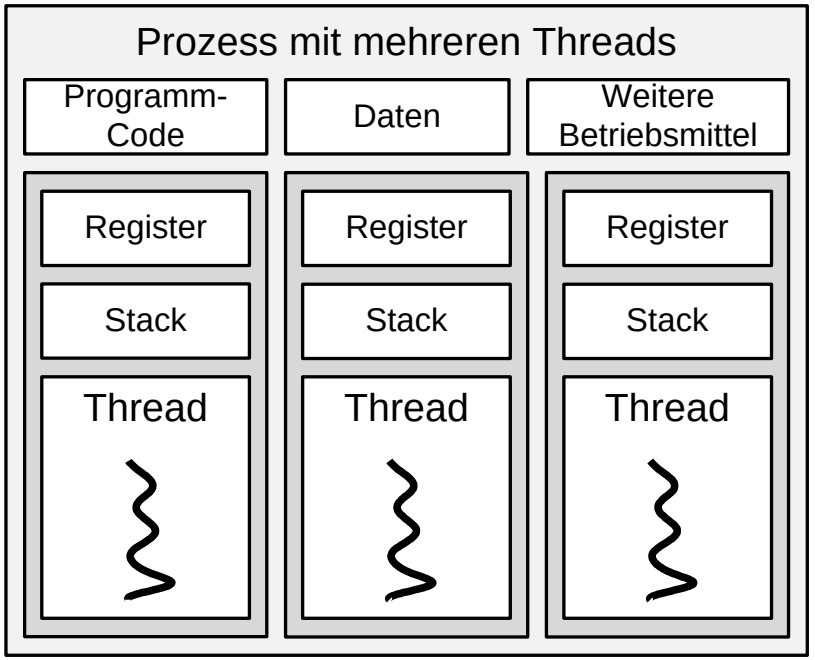
- ***Prozess*** = Heavy Weight Ressource
  - Verwaltet vom Betriebssystem (idR. wenige Prozesse pro BS)
  - Eigener Speicherbereich, Eigene Ressourcen
  - Kommuniziert mit anderen Prozessen ***nur*** über Pipes, Shared Memory, RMI/RPC, Messaging, …
- ***Thread*** = Light Weight Ressource
  - Teil eines Prozesses
  - Verwaltet von modernen Betriebssystemen (beliebig viele Threads)
  - Java: Eigene Thread-Verwaltung
  - ***Teilt sich Speicher, Ressourcen mit anderen Threads***
  - Aber: ***Eigener Programmkontext*** (Stack, Lokale Variable)
- ***Task*** = Aufgabe, die in einem Thread/Prozess ausgeführt wird

(Achtung: Task wird in Vorlesungen BS und RA anders definiert)

Prozesse und Threads

(Sicht aus Silberschatz: Betriebssysteme)





Zwei Sorten von Nebenläufigkeitsproblemen

- ***Kontrollfluss – Nebenläufigkeit*** (HIER!!)
  - Task dauert lange, wird in eigenen Thread ausgelagert
  - Verschiedene Tasks verarbeiten verschiedene Daten
  - Lösung Java: Executor Framework (Runnable/Callable)
  - Ergebnis des Tasks soll im Haupt-Thread (GUI, while(true)) weiterverarbeitet werden (-> Futures)
- ***Datenfluss – Nebenläufigkeit*** (vgl. Prog. 3)
  - Ein Task soll Daten parallel verarbeiten (Vektor, Matrix-Rechnung) für Maschinelles Lernen, Computer Grafik, Bitcoin-Schürfen
  - Lösung: Java 8 Streams

Tasks als Runnable und Threads in Java

Interface **`Runnable`** für ***Tasks***

- Interface für den Funktionsumfang eines Threads, den Task

```
public interface Runnable () {
    void run();
}
```

- **`run`** – Methode kapselt Funktionalität, die nebenläufig ausgeführt werden soll
- ***Thread terminiert, sobald run() „fertig“***
- Klasse **`Thread`** implementiert auch **`Runnable`**
- ***Kein Rückgabewert, keine checked Exceptions***
  (dazu wurde das Interface **`Callable`** eingeführt)

Die Klasse **Thread**

- Konstruktoren:

```
public Thread();
public Thread(String name);
public Thread(Runnable target);
```

- Wichtigste Methode:

```
public void start(); // Starten eines Threads
```

- Terminiert, wenn main() terminiert (deamon) ?

```
public void setDeamon(boolean); // User / Deamon
```

- Warten auf einen anderen Thread (Barrier-Synchronization)

```
public void join(); // User / Deamon
```

- Ein Thread beendet sich durch Verlassen der **run** Methode z.B. mit return

```
public void interrupt();
public boolean isInterrupted();
```

Laufverhalten eines Threads
Achtung: Executor Framework verwenden!

1. Ableiten von Thread und überschreiben der Methode „run“
   (class Thread implements Runnable)

```
public class MyThread extends Thread {
    public void run() {
        // Hier das Verhalten
    }
}
Thread t = new MyThread();
```

2. Besser: Implementieren der Runnable-Schnittstelle, sehr elegant ab Java 8 mit einem Lambda-Ausdruck (später mehr)

```
Thread t = new Thread( new Runnable() {
        public void run() {
            // Hier das Verhalten
        }
    });
```

User Threads und Daemon Threads

- User Thread terminiert wenn run() Methode beendet.
- Wenn User Thread dauerhaft laufen soll: while(true)
- Main-Thread kann terminieren bevor alle User Threads terminiert sind (Achtung: GUI kann inkonsistent wirken: System läuft halb)

- Lösung: Deamon-Thread (**`setDeamon(true)`**), diese Threads terminieren, wenn GUI Terminiert

Threads terminieren und Fehlerbehandlung

Threads manuell terminieren ist schwierig

- Einiges ist depricated, da unsafe / nicht stabilisierbar (z.B. Thread.stop(), resume(), suspend())
- Direktes Terminieren eines Threads ist kaum möglich:
  - **`interrupt()`** setzt interrupted flag auf true, abgefragt mit isInterrupted()
  - Interrupt() unterbricht sleep() mit InterruptedException, diese setzt auch interrupted Flag auf false
- Sicheres Behandeln eines Abbruchs also
  - Statt **`while(true) {…}`** z.B.
    `while(!Thread.currentThread().isInterrupted()) { … }`
  - Und bei catch(InterruptedException x) {
    - Entweder: Thread.currentThread().interrupt() }
    - Oder gleich: return;}

Fehlerbehandlung

- In run() nur Runtime-Exceptions möglich
- Fangen der Exceptions über **`UncaughtExceptionHandler`**
- ***Wichtig wegen „Frühem und sicherem Scheitern“, den Server / den Client beenden können, nicht nur den Thread mit der Exception***
- Beispiel

```
Thread.setDefaultUncaughtExceptionHandler(
   new Thread.UncaughtExceptionHandler() {

     public void uncaughtException(Thread t, Throwable e) {
         // …
     }
});
```

Agenda

Wie gehen wir wirtschaftlich mit Threads um?

Entwurfsproblem: Wirtschaftlicher Umgang mit Threads

Pooling

- Problem: Für jeden Task neuen Thread erzeugen
  - = teuer, kostet beim Erzeugen Laufzeit
  - = Ressourcen Verschwendung, da Thread von VM verwaltet werden muss
- Idee: Threads geschickt verwalten: Viele Tasks werden von wenigen durchgängig laufenden Threads bearbeitet
- Lösung: Ressourcen werden in einen ***Pool*** gestellt
  - Pool kann eine feste Zahl von Threads enthalten
  - Wenn Task gerechnet werden muss: Thread aus entsprechendem Pool heraus holen
  - Wenn Task bearbeitet: Ressourcen zurück in die entsprechenden Pool
- Implementierung: z.B. Executor Framework (ab JDK 5)

Lösung in Java 5: Executor Framework

- Basisinterface führt Runnable aus: **`Executor`**

```
public interface Executor {
     void execute(Runnable command);
}
```

- Lebenszyklusüberwachung mit **`ExecutorService`**

```
public interface ExecutorService extends Executor {
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(
         long timeout, TimeUnit unit);
    // …
```

Diskussion des Beispiels

- Executor hat ***Queue mit Tasks***, die er abarbeitet
- Task (Runnable) so lange in der Queue, bis Executor diesen Task abarbeitet
- ***Abarbeitungsstrategie frei wählbar***
  - Ein Thread arbeitet alle Tasks ab
  - Ein Thread-Pool mit fester oder variabler Größe arbeitet Tasks ab
  - Ein Thread-Cache arbeitet die Tasks ab
- Abarbeitung: Executor holt Task (Runnable) aus der Queue und führt in einem seiner Threads dessen run() Methode aus:
  - in der run() Methode des jeweiligen Executor-Threads [die nicht(!) terminiert] wird
  - die run() Methode des Tasks [die terminieren muss, z.B. mit return]

Executors Implementierungen

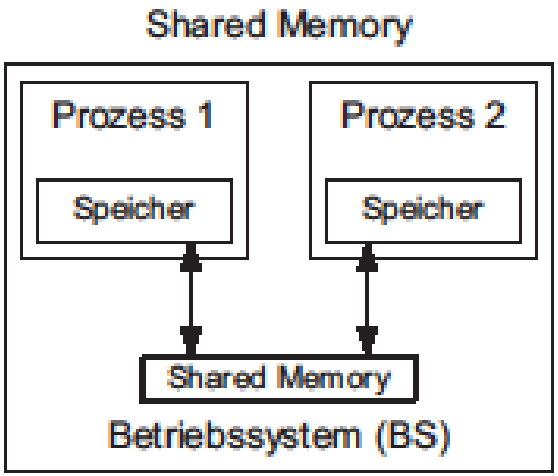
- **`Executors.newFixedThreadPool`**
  - Executor hält Thread-Zahl konstant
  - wenn ein Thread beendet wird, wird ein neuer erzeugt
- **`Executors.newCachedThreadPool`**
  - Erzeugt bei hoher Last mehr neue Threads
  - Thread Zahl ist nicht begrenzt
- **`Executors.newSingleThreadExecutor`**
  - Nur ein einziger Thread
  - Abarbeitungsstrategie durch interne Queue bestimmt (LIFO, FIFO, priority)
- **`Executor.newScheduledThreadPool`**
  - Executor hält Thread Zahl konstant
  - Wiederkehrende Ereignisse können ge-scheduled werden (ähnlich Timer)

Agenda

Wie kommunizieren Threads am besten?

Allgemeine Möglichkeiten (vgl. BS-Vorlesung)

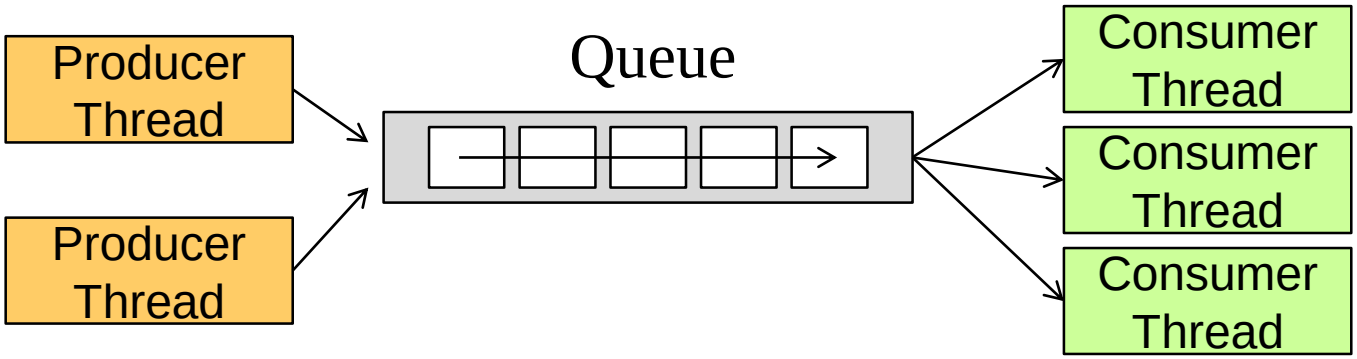
- Shared Memory (BS,VV)
  - Muss synchronisiert sein (wg. „Race Conditions“, später)
  - Threads eines Prozesses haben „Shared Memory“
- Pipes des Betriebssystems (-> BS Vorlesung)
- Sockets (-> Rechnernetze, später genauer)



Shared Memory
Hier gehen wir erstmal nicht ins Detail!

- Ein Problem, ***NUR WENN sich die gemeinsamen Daten ändern***
  - Race Conditions: gem. Zugriff verletzt interne Konsistenzbedingungen / Invarianten (vgl. Anomalien aus DB-Vorlesung), später
  - Immutable Daten (String, Integer) sind kein Problem!
- Lösung allgemein
  - Mutexe / Semaphore
- Lösung Java: sog. „Monitore“ (-> BS-Vorlesung)
  - Gegenstand des Sperrens = Objekt
  - Sperre anfordern / freigeben automatisch (daher Monitor) mit **`synchronized`**

Shared Memory: Producer/Consumer-Pattern



- Queue = Synchronisiertes Shared Memory (-> BS)
- Producer, z.B. die Applikation(en)
  - Schreiben fortlaufend in Queue
  - Waiting (=schlafen) wenn Queue ***voll***
- Consumer, z.B. der/die Logger
  - lesen fortlaufend aus Queue
  - Waiting (=schlafen) wenn Queue ***leer***

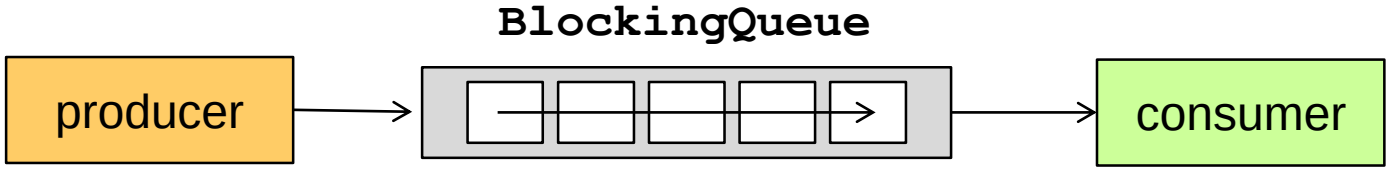
Queues ab JDK 5

- Basisschnittstelle für Queues: **`BlockingQueue<T>`**
- Alle Unterklassen Thread Save, intern synchronisiert
- Gedacht für Producer / Consumer Varianten

```
interface BlockingQueue<T> {
   void put(T t); // Einfuegen
   T take(); // Lesen und Löschen, wartet

   // Mit Timeout:
   boolean offer(T t, long time, TimeUnit u) // wie put
   T poll(long time, TimeUnit u) // wie take
   // … }
```

- Implementierungen
  - **`ArrayBlockingQueue, LinkedBlockinQueue`**
  - **`PriorityBlockingQueue`**

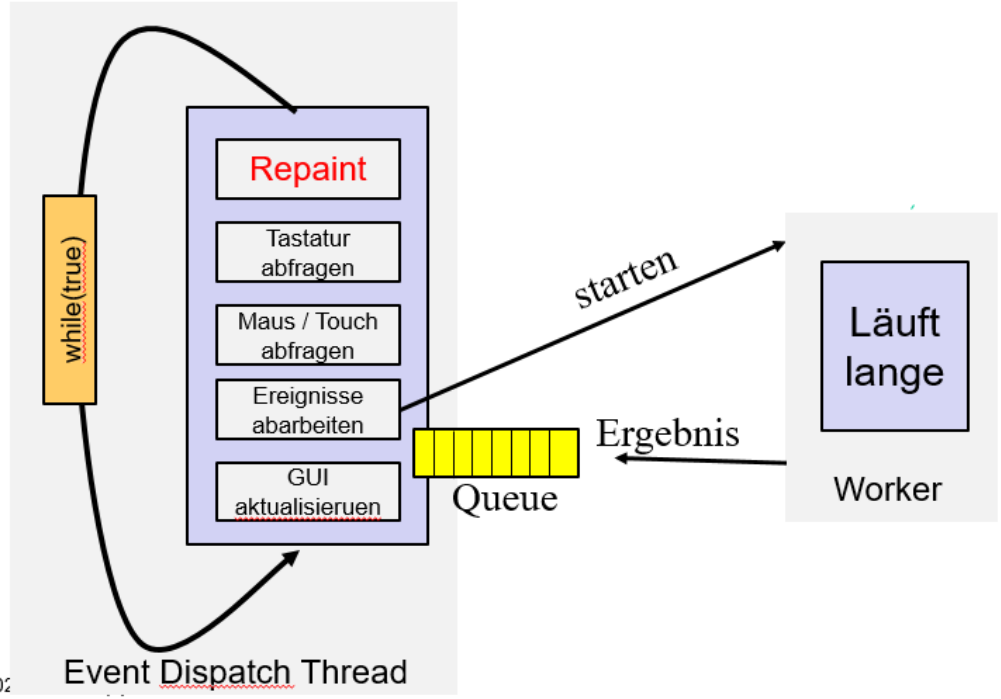
Beispiel




```
BlockingQueue<String> orders = new ArrayBlockingQueue<>(5);
ExecutorService exe = Executors.newCachedThreadPool();

Runnable producer = () -> {
      // …
      orders.put("Order");
      // …
   };

   Runnable consumer = () -> {
      // …
      String order = orders.take();
      // …
};

exe.execute(producer);
exe.execute(consumer);
```

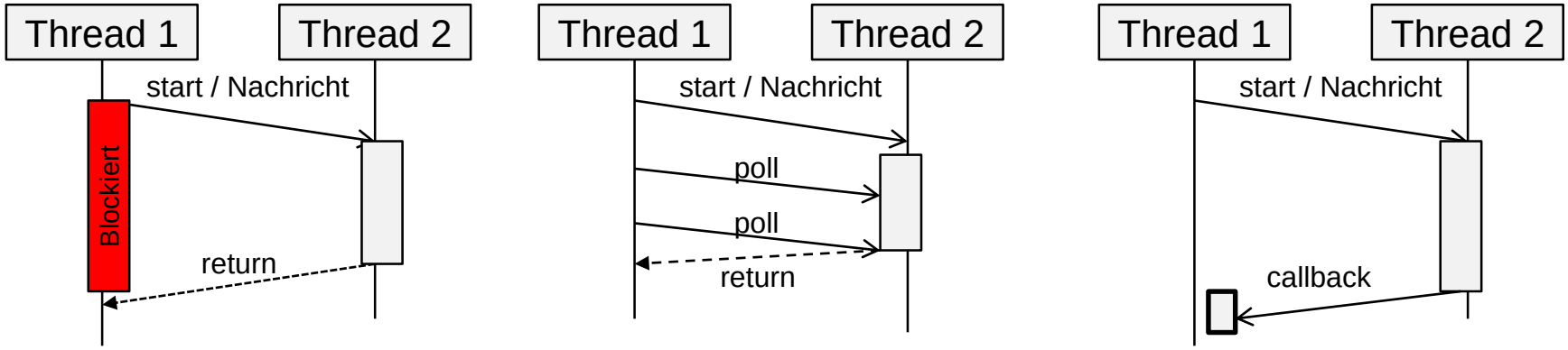
Willkommen in der asynchronen Programmierung!

Methoden mit Ergebnis in Threads auslagern



Drei Strategien

- Drei Strategien: Synchron, Polling und Callbacks



- Synchron (Achtung: kein „Methodenaufruf“, blockierend, join())
- Polling: Häufig Aufrufe mit Timeout (nicht blockierend)
- Callback: Häufig über EventListener
  - Achtung: *Callback-Methode (Listener) läuft im aufgerufenen Thread*
  - Aufrufender Thread muss die Ergebnisdaten aus gemeinsamem Speicher (z.B. Queue) lesen und weiterverwerten
  - Alternative: Runnable an den anderen Thread schicken, der arbeitet das z.B. in seiner „Event-Queue“ aub

Threads liefern Ergebnisse
Future<T> und Callable<T>

- Interface für Codeblock mit Rückgabewert und möglicher CheckedException:

```
public interface Callable<T> {
   T call() throws Exception;
};
```

- Interface zur Auswertung des Rückgabewertes und des Lebenszyklus des Callable

```
public interface Future<T> {
     boolean cancel(boolean mayInterruptIfRunning);
     boolean isCancelled();
     boolean isDone();
     T get() throws InterruptedException,
            ExecutionException, CancellationException;
     T get(long timeout, TimeUnit unit)
            throws TimeOutException // Rest Ex.wie get()
};
```

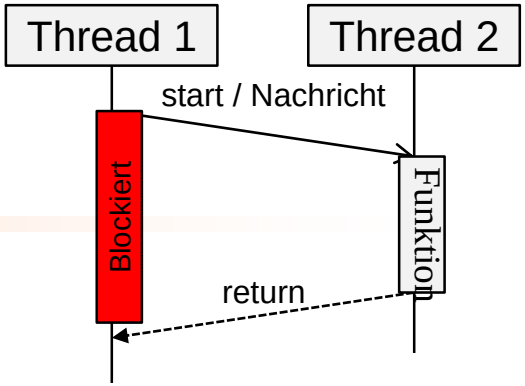
`FutureTask<T>` Lebenszyklus von Callables

- Klasse für Lebenszyklus-Management von Callables: **`FutureTask`**
  - implements **`Future<T>`** und **`Runnable`**
  - Sychron = get() blockiert bis Callable fertig
  - Polling = get(long timeout, TimeUnit t) blockiert, dann TimeOutException oder isDone()
  - Abbrechen über cancel(…)
  - Status über isCancelled(), isDone()

- Thread erzeugen z.B. mit

```
Thread t = new Thread(new FutureTask …);
t.start();
```

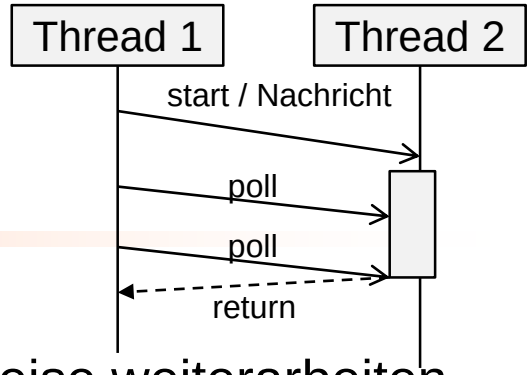
Strategie: Blockieren



- Idee: Funktion wird als Callable implementiert
- Ausführung über Executor
- Ergebnis dann als Future verfügbar
- Blockierend, wenn die get() Methode von Future verwendet wird
- Zwischeh submit() und get() kann Thread 1 was sinnvolles tun.

```
ExecutorService exe = Executors.newSingleThreadExecutor();

Callable<String> serverCall = () -> {
    String result = slow.serverCall("Some Input");
    return result;
};

Future<String> futureResult = exe.submit(serverCall);
// Hier kann ich noch was Machen

System.out.println("Berechnet: " + futureResult.get());
```

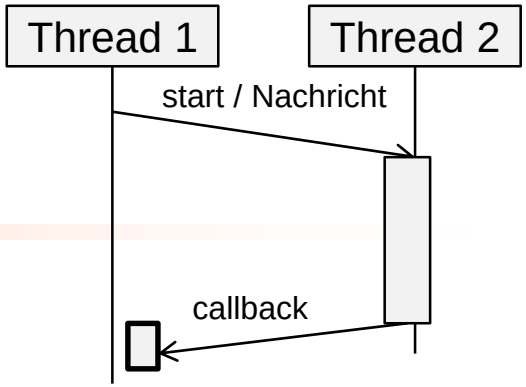
Strategie Polling



- Polling = zyklisches Abfragen
- Kostet Rechenleistung, Thread 1 kann teilweise weiterarbeiten (schwer zu Programmieren)

```
ExecutorService exe = Executors.newSingleThreadExecutor();

  Callable<String> serverCall = () -> {
    String result = slow.serverCall(input);
    return result;
  };

  Future<String> futureResult = exe.submit(serverCall);

  while(!futureResult.isDone()){
    Thread.currentThread().sleep(1000);
    System.out.println("Poll");
  }

  System.out.println("Berechnet: " + futureResult.get());
```

Strategie Callback



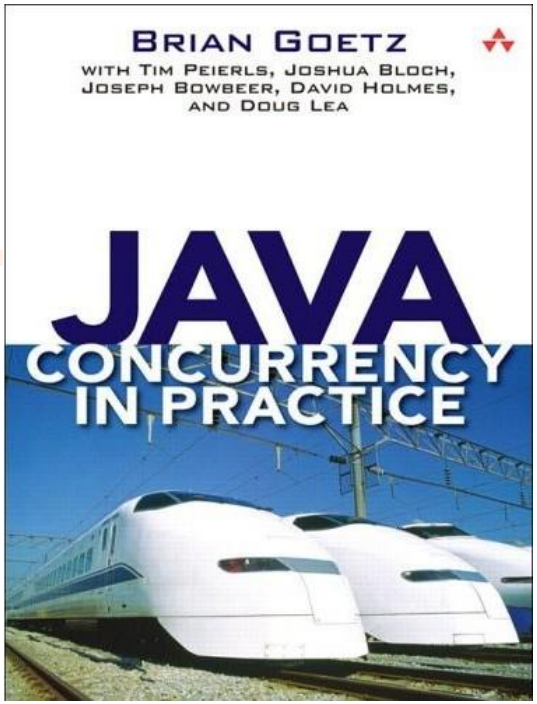
- Seit Java 8: CompleteableFuture
- Verkettung asynchron ausgeführter Verarbeitungsschritte

```
CompletableFuture<Void> serverCall =
    CompletableFuture.supplyAsync(
        () -> { String result = slow.serverCall(input);
                return result;
              }, exe)
    .thenAccept(
        (s) -> { System.out.println("Ergebnis = " + s);});
```

Diskussion des Beispiels Vollst. asynchrones Arbeiten

- Achtung: GUI ist immer ***logisch synchron*** (Benutzer wartet auf Antwort)
- Hintergrund moderner GUI ist in der Regel ***technisch asynchron*** umgesetzt (Hintergrund Threads)
- Wenn wir Ergebnisse der asynchronen Threads brauchen:
  - ***Z.B. aus GUI wird Server oder Datenbank aufgerufen***
  - ***Programmierung deutlich aufwendiger, da das Ergebnis des Aufrufs irgendwann verarbeitet werden muss***
- Wenn wir die Ergebnisse ***nicht*** brauchen, also auch der Benutzer wartet nicht auf eine direkte Antwort
  - Z.B. Bestätigung der Verarbeitung kommt später per Mail, Synchron wird nur ein ACK (Auftrag angekommen) erwartet
  - ***Programmierung genauso einfach wie synchron***

Literatur



- Brian Goetz: Java Concurrency in Practice, Addison-Wesley 2006
  (Stand: Java 5, dort war der größte Umbau bei Kontrollflussparallelität, insbes. Memory Model)
- Subramaniam: Functional Programming in Java, Pragmatic Programmer, 2014
  (Stand: Java 8: dort war der größte Umbau bei Datenparallelität, Streams und Fork/Join Framework, autom. Multi-Threading)