

Multithreading Grundlagen

Hyper-threading

Multi-Core

Multi-Processor

Rechner-Netzwerk

Distanz der Cores

1. jede proz. Aufgabe bekommt mind. 1. Kugel aus 2. Kugeln-Satz

Processor Chip

Reg. Set

Reg. Set

Core Unit

Processor Chip

Core

Core

Core

Core

Computer

CPU

GPU

CPU

Phi

Computer

Computer

Computer

Computer

Parallelität: Aufteilung in Teilabläufe, laufen gleichzeitig auf mehreren Prozessoren. **Nebenläufigkeit:** Gleichzeitig oder verzahnt ausführbare Abläufe, greifen auf gemeinsame Ressourcen zu. **Prozess:** Parallel laufende Programm-Instanz im System. Eigener Adressraum. **Thread:** Parallele Ablaufsequenz innerhalb eines Programms. Teilen gleichen Adressraum.

Thread-Implementationen

User-Level Threads: Im Prozess implementiert. Keine echte Parallelität durch mehrere Prozessoren. **Kernel-Level Threads:** Im Kernel implementiert (Multi-Core Ausnutzung). Kontextwechsel vom Prozess per SW-Interrupt. **Thread Scheduling:** Processor Sharing - #Threads > #Prozessoren.

Prozessor Multiplexing

Verzahnte Ausführung: Instruktionen von mehreren Threads in Teilsequenzen. Illusion der Parallelität. **Kontextwechsel:** Synchron (Freiwillige abgabe), Thread wechselt zu wait. *Asynchron:* (gezwungene Abgabe) Begrenzte Laufzeit für Threads.

Multi-Tasking:

Kooperativ: Threads initiieren Kontextwechsel synchron (freiwillig). Scheduler kann Thread nicht unterbrechen. **Preemptiv:** Scheduler kann Thread mit Timer-Interrupt asynchron unterbrechen (Time-Sliced-Scheduling) (*Standard heute*). **Thread Zustände:** Ready, Waiting, Running.

Multi-Thread Programmierung

JVM Thread Modell

Java ist ein Single Process System. JVM ist ein Prozess im Bsys. **Main-Thread** wird beim Aufstarten der JVM anhand *main()* Methode erzeugt. JVM läuft, solange Threads laufen (Ausnahme Daemon Threads).

```
var myThread = new Thread(() -> { /* ... */ });
myThread.start();
```

Thread wird erst bei *start()* erzeugt. Führt *run()*-Methode des Runnable Interface aus. Thread endet beim Verlassen von *run()*. **Nicht-Determinismus:** Threads laufen ohne Vorkehrungen beliebig verzahnt oder parallel.

Explizite Runnable-Implementation:

```
class SimpleLogic implements Runnable {
    @Override
    public void run() { /* ... */ }
}

var myThread = new Thread(new SimpleLogic()).start();
```

Sub-Klasse von Thread

```
class SimpleThread extends Thread {
    @Override
    public void run() { /* ... */ }
}

var myThread = new SimpleThread().start();
```

Thread Join

Warten auf Beendigung eines Threads. *t2.join()* blockiert, solange t2 läuft.

Thread Passivierung

Thraed.sleep(ms): Laufender Thread geht in Wartezustand, dann ready. **Thread.yield():** Gibt Prozessor frei, direkt ready.

Monitor Synchronisation

Synchronisation: Einschränkung der Nebenläufigkeit. **Gemeinsame Ressourcen:** Threads teilen sich Adressraum und Heap.

Java Synchronized Methoden

synchronized: Modifizier für Methoden. Body ist ein kritischer Abschnitt. Wird unter gegenseitigem Ausschluss ausgeführt. **Funktionsweise:** Jedes Objekt hat einen Lock (Monitor-Lock). Max 1 Thread hat Lock. *synchronized* belegt den Lock des Obj. Besetzt bei Eintritt oder warten bis frei. Beim Austritt der Methode wieder freigegeben.

```
synchronized void f() { /* ... */ } // Object Lock
static synchronized void g() { /* ... */ } // Class Lock
```

Monitor

Objekt mit internem gegenseitigem Ausschluss. Nur 1 Thread operiert im Monitor. Alle äusseren Methoden *synchronized*. **Wait & Signal Mechanismus:** Threads können im Monitor auf Bedingung warten und wartende aufwecken (signal).

```
public synchronized void withdraw(int a) {
    while (amout > balance) { wait(); }
}
```

```
balance -= a;
}

public synchronized void deposit(int a) {
    balance += amount; notifyAll();
}
```

wait(): gibt Monitor-Lock temp. frei damit anderer Thread Bedingung erfüllen kann. **notify()** / **notifyAll():** Weckt beliebigen / alle Threads die warten. Notify: wenn Bedingung jeden Thread interessiert. **Pauschales wait & signal:** Wartende müssen selber schauen, ob sie ein Signal interessiert. **Signal and Continue:** Signalisierender Thread behält Monitor nach notify. Aufgeweckter Thread muss um Monitor-Eintritt kämpfen.

Spezifische Synchronisationsprimitiven

Semaphoren

Vergabe einer beschränkten Anzahl freier Ressourcen. Objekt mit Zähler (Anzahl freier Ressourcen). **acquire():** Bezieht Ressource. Warten falls keine Verfügbar, sonst Zähler - 1. **release():** Ressource freigeben, Zähler + 1.

Arten

Allgemeine: Zähler zwischen 0 bis N. *new Semaphore(N)*. **Binäre:** Zähler 0 oder 1. *new Semaphore(1)*. (Gegenseitiger Ausschluss) **Faire Semaphoren:** *new Semaphore(N, true)*, FIFO Warteschlange, langsamer.

```
private Semaphore upperL = new Semaphore(CAP, true);
private Semaphore lowerL = new Semaphore(0, true);
public void put(T item) throws InterruptedException {
    upperL.acquire();
    synchronized (queue) { queue.add(item); }
    lowerL.release();
}

public T get() throws InterruptedException {
    T item; lowerL.acquire();
    synchronized (queue) { item = queue.remove(); }
    upperL.release(); return item;
}
```

Multi-Acquire/Release: *acquire(N)*.

Lock & Condition

Monitor mit mehreren Wartelisten für verschiedene Bedingungen. **Lock-Objekt:** Sperre für Eintritt in Monitor. **Condition-Objekt:** Wait & Signal für bestimmte Bedingung.

```
private Lock monitor = new ReentrantLock(true); // fair
private Condition nonFull = monitor.newCondition();
private Condition nonEmpty = monitor.newCondition();
public void put(T item) throws InterruptedException {
    monitor.lock();
    try {
        while (queue.size() == Capacity) { nonFull.await(); }
        queue.add(item);
        nonEmpty.signal();
    } finally { monitor.unlock(); }
}

public T get() throws InterruptedException {
    monitor.lock();
    try {
        while (queue.size() == 0) { nonEmpty.await(); }
        T item = queue.remove();
        nonFull.signal();
        return item;
    } finally { monitor.unlock(); }
}
```

Read-Write Locks

Gegenseitiger Ausschluss ist unnötig streng für lesende Abschnitte. Erlaube parallele Lese-Zugriffe.

```
private Collection<String> names = new HashSet<>();
private ReadWriteLock rwLock = new ReentrantReadWriteLock();
public boolean exists(String pattern) { // Read-only accesses
    rwLock.readLock().lock();
    try {
        return names.steram().anyMatch(n -> n.matches(pattern));
    } finally { rwLock.readLock().unlock(); }
}

public void insert(String name) { // Write / Read accesess
    rwLock.writeLock().lock();
    try {
        names.add(name);
    } finally { rwLock.writeLock().unlock(); }
}
```

Count Down Latch

Synchronisationsprimitive mit Count Down Zähler. Threads können warten, bis Zähler <= 0 wird. **await():** warten bis Count Down 0 ist. **countDown():** Zähler - 1.

```
var ready = new CountDownLatch(N); // Warte auf N cars
var start = new CountDownLatch(1); // Einer gibt signal
// N Cars:
ready.countDown(); start.await();
// RaceControl:
```

```
ready.await(); start.countDown();
}

Cyclic Barrier
Treffpunkt für fixe Anzahl Threads. Anzahl treffender Threads muss vorgegeben sein. Ist wiederverwendbar (mehrere Runden).
var start = new CyclicBarrier(N); // Treffende Autos
// N Cars:
start.await(); // braucht kein Race Control mehr

var gameRound = new CyclicBarrier(N);
// N Players:
while (true) {
    gameRound.await(); // play concurrently with others
}

// Mit Austausch
Exchanger.exchange(something);
// Blockiert bis anderer Thread auch exchange() aufruft.
```

Gefahren der Nebenläufigkeit

Neue Arten von Programmierfehlern, die es bei single-Threading nicht gibt. Können sporadisch oder selten auftreten. Sehr schwierig durch Tests zu finden.

Race Condition

Mehrere Threads greifen auf gemeinsame Ressourcen ohne genügend synchronisation zu. Mögliche falsche Resultate oder falsches Verhalten. Ursache oft ein Data Race, nicht immer.

Data Race

Unsynchronisierter Zugriff auf gleichen Speicher. Selbe Variable oder Array Element (min. 1 schreibender Zugriff).

Race Condition ohne Data Race

Critical Sections nicht geschützt. Data Races mit Synchronisation eliminiert, aber nicht genügend grosse synchronisierte Blöcke.

```
synchronized int getBalance() { return balance; }
synchronized void setBalance(int x) { balance = x; }
// Mehrere Threads, Kein Atomares Inc – Lost Update moeglich
account.setBalance(account.getBalance() + 100);
```

Kombinationen

	Race Condition	keine Race Condition
Data Race	Fehlerhaftes Programmverhalten	Programm verhält sich zwar korrekt, dennoch formal falsch
kein Data Race	Fehlerhaftes Programmverhalten	Richtig

Alles Synchronisieren? Hilft nichts. Race Condition trotzdem möglich. Weitere Nebenläufigkeitsfehler. Synchronisationskosten sind relativ teuer.

Synchronisation: Verzichtbare Fälle

Immutability (Unveränderlichkeit): Objekte mit nur lesendem Zugriff. **Confinement (Einspernung):** Objekt gehört nur einem Thread zu einer Zeit.

Immutable Objects

Instanzvariablen alle *final*. Primitive Datentypen. Referenzen wiederum auf Immutable Objekte. Methoden mit nur lesendem Zugriff. Konstruktor initialisiert Instanzvariablen. Nach Konstruktor kann Objekt ohne Synchronisation von Threads verwendet werden.

Confinement

Struktur garantiert, dass auf ein Objekt nur durch einen Thread zur gleichen Zeit zugegriffen wird. **Thread Confinement:** Objekt gehört nur einem Thread und wird nur von demjenigen verwendet. **Object Confinement:** Objekt in anderem bereits synchronisierten Objekt eingekapselt.

Kapselungsbrüche: 1. Inneres Objekt ist aussen zugreifbar. 2. Rückgabe einer Referenz auf inneres Objekt. 3. Holder installiert selber Referenz ausserhalb. 4. Inneres Objekt gibt selber *this* raus.

Thread Safety

Klassen / Methoden, die intern synchronisiert sind. Keine Race Conditions innerhalb dieses Codes. Kritischer Abschnitt nur pro Methode erfüllt. **Aber:** Kein kritischer Abschnitt über mehrere Methodenaufrufe. Andere Nebenläufigkeitsfehler möglich.

Java Collections - Thread Safety

Alte Java 1.0 Collections (Vector, Stack, Hashtable): **JA**. Moderne Collections (HashSet, TreeSet, ArrayList, etc.): **NEIN**. Concurrent Collections (ConcurrentHashMap, etc.): **JA**.

Verstecktes Multi-Threading

Finalizers: Laufen über separaten Finalizer-Thread. **Timers:** Handler durch separaten Thread ausgeführt (ausser GUI). **Externe Libraries & Frameworks:** z.B. Abarbeitung von Web-Service Aufrufen.

Deadlock

Beide Threads sperren sich gegenseitig aus:

```
synchronized(listA) { // Thread 1
    synchronized(listB) {
        listB.addAll(listA);
    }
}
```

```
}

synchronized(listB) { // Thread 2
    synchronized(listA) {
        listA.addAll(listB);
    }
}
```

Spezialfall: Livelocks

Threads haben sich gegenseitig permanent blockiert. Führen aber noch Warteinstruktionen aus. Verbrauchen CPU während Deadlock.

```
// Thread 1
b = false; while (!a) { } b = true;
// Thread 2
a = false; while (!b) { } a = true;
```

Deadlock Erkennung

Betriebsmittelgraph

– Thread T wartet auf Lock von Ressource R

– Thread T besitzt Lock auf Ressource R

Deadlock = Zyklus im Betriebsmittelgraph

Deadlock Voraussetzungen: Geschachtelte Locks, Zyklische Wartebhängigkeiten

Deadlock Vermeidung

Lineare Sperrordnung der Ressourcen einführen. Nur geschachtelt in aufsteigender Reihenfolge sperren. Eliminiert zyklische Wartebhängigkeiten.

Großgranulare Locks wählen. Wenn lineare Sperrordnung nicht möglich/sinnvoll ist. Sperre gesamte Bank bei Kontenzugriff. Eliminiert Schachtelung von Locks.

Starvation

Ein Thread kriegt nie die Chance, auf eine Ressource zuzugreifen, obwohl sie immer wieder frei wird. Andere Threads überholen andauernd. Liveness/Fairness Problem.

```
do { // Starvation moeglich
    success = account.withdraw(100);
} while (!success);
```

Vermeidung

Faire Synchronisationskonstrukte (bei Semaphore, Lock & Condition, ReadwriteLock möglich). Java Monitor hat ein Fairness Problem (Starvation anfällig).

Parallelität Korrektheitskriterien

Safety: Keine Race Conditions, Keine Deadlocks. **Liveness:** Keine Starvation.

Thread Pool

Tasks: Implementieren potentiell parallele Arbeitspakete. Werden in Warteschlange eingereiht. **Thread Pool:** Beschränkte Anzahl von Worker-Threads. Holen Tasks aus der Warteschlange und führen sie aus.

Vorteile

Beschränkte Anzahl von Threads: Viele Threads verlangsamen das System oder überschreiten Speicherlimit. **Recycling der Threads:** Spare Thread-Erzeugung und Freigabe. **Höhere Abstraktion:** Trenne Task-Beschreibung von Task-Ausführung. **Anzahl Threads pro System konfigurierbar:** #Worker Threads = #Prozessoren + #1/O-Aufrufe

Einschränkung

Tasks dürfen nicht aufeinander warten, sonst Deadlock. **Run to Completion:** Task muss zu Ende laufen, bevor Worker Thread anderen Task ausführen kann. **Ausnahme:** geschachtelte Tasks.

Java Fork-Join-Pool

```
var threadPool = new ForkJoinPool();
Future<Integer> future = threadPool.submit(() -> { });
Int result = future.get(); // Blockiert bis Task beendet
```

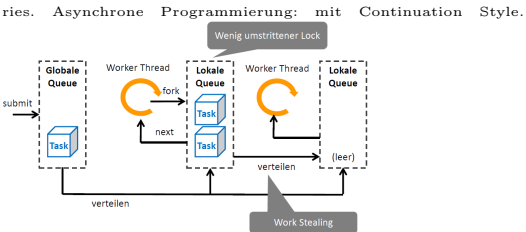
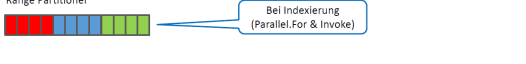
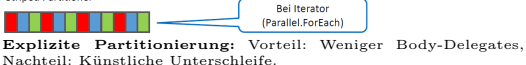
Future Konzept

Repräsentiert ein zukünftiges Resultat. Proxy auf Resultat, das evtl. noch nicht bekannt ist. Muss Ende der Berechnung abwarten, bevor Resultat zurückgegeben wird.

Rekursive Task Erstellung

```
class CountTask extends RecursiveTask<Integer> {
    // Constructor
    @Override
    protected Integer compute() {
        // no / single element ==> return result
        // Calculate lower, middle, upper
        var left = new CountTask(lower, middle);
        var right = new CountTask(middle, upper);
        left.fork(); right.fork();
        return right.join() + left.join();
    }
}
```

<pre>} // Ausfuehrung var threadPool = new ForkJoinPool(); int res = threadPool.invoke(new CountTask(2, N)); // blockiert</pre>
Keine Über-Parallelisierung
Tuning mit Schwellwert durch Programmierer. Verhältnis zwischen Arbeitspaketgröße und Anzahl Tasks optimieren.
Fork Join Pool Internals
Automatischer Parallelitätsgrad: Default: #Worker Threads = #Prozessoren Dynamisches Hinzufügen / Wegnehmen von Threads Common Pool: Verhindert Engpässe durch zu viele Thread Pools.
Asynchrone Programmierung
Unnötige Synchronität: Langlaufende Rechnungen, I/O Aufrufe.
Asynchroner Aufruf: Aufrufer soll während der Operation weiterrmachen.
<i>// Klassisch</i> Future<long> future = threadPool.submit(() -> {}); <i>// other work</i> process(future.get()); <i>// Modern</i> CompletableFuture<long> future = CompletableFuture.supplyAsync(() -> {}); <i>// other work</i> process(future.get());
Ende des async Aufrufs
Caller-zentrisch (Pull): Caller warted auf Task-Ende und holt sich Resultat, Future abfragen. Callee-zentrisch (Push): Async Operation informiert direkt über Resultat. Completion Callback.
Continuation
Folgeaufgabe an asynchrone Aufgabe anhängen. <i>// thenApply() fuer Continuation mit Rueckgabe</i> future.thenAccept(res -> System.out.println(res));
Ausführung: durch beliebigen Thread, durch Initiator, wenn Future bereits Resultat hat. Asynchrone Continuations: <i>thenAcceptAsync()</i> bzw. <i>thenApplyAsync()</i> .
Multi-Continuation
CompletableFuture.allOf(f1, f2).thenAcceptAsync(() -> {}); CompletableFuture.any(f1, f2).thenAcceptAsync(() -> {});
Fire and Forget
Task starten, ohne das Ende abzuwarten. Submitter ruft kein <i>get()</i> oder <i>join()</i> auf. CompletableFuture.runAsync(() -> {});
Daemon Workers: Workers Threads in Fork-Join-Pools sind Daemon. Anwendung kann vor Task-Ende stoppen. Ingorierte Exceptions: Exceptions in Fire & Forget Task werden ignoriert.
Task Parallel Library
.NET Threads
Keine Vererbung: Delegate bei Konstruktor. Exception in Thread: Abbruch des Programs. var myThread = new Thread () => { /* ... */ }; myThread.start(); /* ... */ myThread.join(); C# Lambda kann umgebende Variablen zugreifen (auch schreibend). Monitor in .NET FIFO Warteschlange, Pulse informiert längst Wartenden. <i>Wait()</i> in Schlaufe. <i>PulseAll()</i> bei mehreren Bedingungen oder Erfüllungen mehrerer Threads. Synchronisation mit HilfsObj als Best Practice. private object syncObject = new (); <i>// Monitor auf HilfsObj</i> public void Widthdraw(decimal amount) { lock (syncObject) { while (amout > balance) { Monitor.Wait(syncObject); } balance -= amount; } public void Deposit(decimal amount) { lock (syncObject) { balance += amount; Monitor.PulseAll(syncObject); } }
.NET Synchronisationsprimitiven
Fehlen: kein Fairnessflag, kein Lock & Condition. Zusätzlich: ReadwriteLockSlim für Upgradable Read/Write. Semaphoren auch auf OS-Stufe nutzbar. Mutex (binärer Semaphor). Collections nicht Thread-safe: Ausser <i>System.Collections.Concurrent</i>
.NET Task Parallel Library (TPL)
Work Stealing Thread Pool. Verschiedene Abstraktionsstufen: Task Parallelization: Explizite Tasks starten und warten. Data Parallelization: Parallele Statements und Que-

ries. Asynchrone Programmierung: mit Continuation Style.

Thread Injection
TPL fügt zur Laufzeit neue Worker Threads hinzu. Hill Climbing Algorithmus: Misst Durchsatz & variiert Anzahl Worker Threads. Kein Deadlock bei Task Abhängigkeiten, aber ineffizient, nicht dafür gemacht.
Task Parallelisierung
Task task = Task.Run () => { }; /* ... */ task.Wait(); <i>// Task mit Rueckgabe</i> Task <int> task = Task.Run () => { return 0; }; Console.WriteLine(task.Result); <i>// Blockiert</i> <i>// Geschachtelte Tasks</i> Task.Run () => { var left = Task.Run () => { }; var right = Task.Run () => { }; int res = left.Result + right.Result; };
Parallele Statements
Menge an Statements potentiell parallel ausführen. Als Task starten. Barriere der Tasks am Ende.
Parallel.Invoke (() => MergeSort(1, m); () => MergeSort(m,1););
Parallele Loop
Schlaufen-Bodies potentiell parallel ausführen. Gruppierung der Bodies in Tasks. Barriere dieser Tasks am Ende. Parallel.ForEach (list, file => Convert(file)); Parallel.For (0, array.Length, i => Calc(array[i]));
Parallele Loop Partitionierung
Schlaufe mit vielen sehr kurzen Bodies ist ineffizient. TPL gruppiert automatisch mehrere Bodies zu Task. Aufteilung gemäss verfügbaren Worker Threads.
Range Partitioner 
Bei Indexierung (Parallel.For & Invoke)
Striped Partitioner 
Bei Iterator (Parallel.ForEach)
Explizite Partitionierung: Vorteil: Weniger Body-Delegates, Nachteil: Künstliche Unterschleife. Parallel.ForEach (Partitioner.Create(0, array.Length), (range, _) => { for (int i = range.Item1; i < range.Item2; i++) { Calc(array[i]); } })
Parallel LINQ
Parallelisierung von Langauge-Integrated Query. Pendant zu Java Steam API. Keyword: <i>AsParallel()</i> from book in bookCol.AsParallel() where book.Title.Contains("Java") select book.ISBN
Asynchrone Programmierung mit TPL
Task Continuation
task1.ContinueWith(task2).ContinueWith(task3); <i>// Multi-Continuation</i> Task .WhenAll(task1, task2).ContinueWith(continuation); Task .WhenAny(task1, task2).ContinueWith(continuation);
GUI and Threading
GUI Frameworks erlauben nur Single-Threading. UI Thread: Loop zur Ausführung der Ereignisse aus einer Queue.
UI Thread Confinement
Wiso basieren GUI-Frameworks auf Single-Thread Modell? Synchronisationskosten: Locking in allen Komponenten und Methoden relativ teuer. Deadlock-Risiko: Bei zyklischen geschachtelten Aufrufen (z.B. MVC).
UI Thread Model - Einschränkungen
Keine lange Operationen in UI Events, blockiert sonst das UI. Kein Zugriff auf UI-Elemente durch fremde Threads, sonst Race Condition. UI Operationen müssen als Events in die UI Event Queue eingereiht werden.

Swing: Dispatching an UI Thread
Komponentenzugriffe an UI Thread delegieren. <i>// Benutzung der Klasse SwingUtilities</i> static void invokeLater(Runnable doRun); <i>// Async</i> static void invokeAndWait(Runnable doRun); <i>// Synchron</i> <i>// Example</i> button.addActionListener(event-> { new Thread () -> { var text = readHugeFile(); SwingUtilities.invokeLater(() -> { textArea.setText(text); }); }.start(); });
Swing Background Worker
Hilfsklasse für Hintergrund Arbeiten. Zeitaufwendige Operationen als Task in Thread Pool <i>doInBackground()</i> . UI-Zugriffe durch <i>DispatchThread done()</i> . <i>// <return, zwischen Res></i> class BgCalc extends SwingWorker<Integer, Void> { @Override public Integer doInBackground() { return longCalc(); } @Override protected void done() { try { inst res = get(); label .setText(res); } catch (InterruptedException ExecutionException e) { } } }
Android Async Task
<i>// <input, zwischen Res, return></i> class BgCalc extends AsyncTask<Void, Void, Integer> { @Override public Integer doInBackground(Void... input) { return longCalc(); } @Override public void onPostExecute(Integer res) { view.setText(res); } }
.NET UI Threading Modell
Gleiches Prinzip wie Java. UI Thread: Aufrufer von <i>Application.Run()</i> . UI Event Dispatching: WPF: <i>control.Dispatcher.InvokeAsync(action)</i> WinForm: <i>control.BeginInvoke(delegate)</i> Async / Await
Async Methode läuft teilweise synchron, teilweise asynchron. Aufrufer führt Methode solange synchron aus bis ein <i>await</i> anliegt. Compiler zerlegt Methode in Abschnitte. Abschnitt nach Await läuft später nach Task-Ende (Continuation). Methode läuft synchron bis <i>await</i> , springt dann zurück zum Aufrufer. Verschiedene Ausführungen: Fall 1: Aufrufer ist normaler Thread, Abschnitt wird durch TPL Worker-Thread ausgeführt. Fall 2: Aufrufer ist UI-Thread, Abschnitt wird als Event vom UI-Thread ausgeführt. public async Task <int> LongOperationAsync() { } Task <int> task = LongOperationAsync(); /* ... */ int res = await task; <i>// Warte auf Beendigung</i>
Async Rückgabetypen: <i>void:</i> fire-and-forget. <i>Task:</i> Kein Rückgabetyp erlaubt warten. <i>Task<T></i> Rückgabetyp T. async Task <string> ConcatAsync(string url1, string url2) { HttpClient client = new HttpClient(); Task <string> d1 = client.GetStringAsync(url1); Task <string> d2 = client.GetStringAsync(url2); string sitel = await d1; string site2 = await d2; return sitel + site2; }
Memory Models
Lock-Freie Programmierung: Korrekte nebenläufige Interaktionen ohne Locks. Garantien des Speichermodells nutzen.
Ursachen für Probleme
Weak Consistency: Speicherzugriffe werden in verschiedenen Reihenfolgen aus verschiedenen Threads gesehen. Ausnahme: Synchronisationen/Speicherbarrieren Optimierungen: Compiler, Laufzeitsystem und CPUs, Instruktionen werden umgeordnet/wegoptimiert.
Java Memory Model
Atomicity
Einzelnes Lesen / Schreiben ist atomar für: Primitive Datentypen, Obj-Referenzen, long und double nur mit <i>volatile</i> Keyword.
Unteilbarkeit ≠ Sichtbarkeit: Nach Write sieht anderer Thread vit. noch alten Wert.
Visibility
Garantierte Sichtbarkeit zwischen Threads: Locks Release & Acquire, Volatile Variable, Thread/Task-Start und Join, Initialisierung von final Variablen.
Ordering
Program Order: 'as-if-serial'. Sequentielles Verhalten jedes Threads bleibt erhalten. (Andere Threads dürfen es anders sehen) Synchronization Order (Total Order): Synchronisationsbefehle werden zueinander nie umgeordnet. Happens-Before Relation

(Partial Order): Alles andere kann umgeordnet werden, ausser es gibt garantierte Sichtbarkeit unter den Threads.
Java Synchronization Order
<i>// Keine Umordnung in Java, weil alles volatile</i> volatile boolean a = false , b = false ; a = true ; while (!b) { } <i>// Thread 1</i> b = true ; while (!a) { } <i>// Thread 2</i> <i>// Nicht korrekt, da nicht atomar</i> private volatile boolean locked = false ; public void acquire() { while (locked) { } locked = true ; } <i>// Spin-Lock mit atomarer Operation</i> private AtomicBoolean locked = new AtomicBoolean(false); public void acquire() { while (locked.getAndSet(true)) { } } public void release() { locked.set(false); } <i>// Atomares Compare and Set, setzt Update falls Wert gleich expect</i> boolean compareAndSet(boolean expect, boolean upadte); <i>// Optimistische Synchronization</i> do { oldValue = var.get(); newValue = calcChange(oldValue); } while (!var.compareAndSet(oldValue, newValue));
.NET Memory Model
Unterschied zu Java: Atomicity: long/double auch ohne volatile atomar. Visibility: Nicht definiert, implizit durch Ordering. Ordering: nur Half und Full Fences. Atomare Instruktionen: <i>Interlocked Klasse</i> Volatile Half Fences Volatile Write: Vorangehende Zugriffe bleiben davor. (Release Semantik) Volatile Read: Nachfolgende Zugriffe bleiben danach. (Acquire Semantik) Full Fence: Memory Barrier Thread.MemoryBarrier(); <i>// Verbieet Umordnung in beide Richtungen</i>
Actor Model
Substantiell anderes Programmierkonzept. Aktive Objekte: haben nebenläufiges Innenleben. Kommunikation: Objekte senden und empfangen Nachrichten. Kein Shared Memory: Nur Austausch von Nachrichten.
CSP
Communicating Sequential Processes: Prozesse kommunizieren indirekt über Channels. Austausch erfolgt unmittelbar und synchron. Unterschied: Actor hat keine Channels, Senden ist immer asynchron, keine garantierte Reihenfolge des Empfangs.
Vorteile: Actor, CSP
Inhäerente Nebenläufigkeit: Alle Obj (Actors) laufen nebenläufig. Maschine kann Grad an Nebenläufigkeit ausnutzen. Keine Race Conditions: Kein Shared Memory. Nachrichtenaustausch synchronisiert implizit. Gute Verteilbarkeit: Kein Shared Memory. Nachrichtenaustausch für Netz prädestiniert.
Akka
Actor Model für JVM. Actors haben privaten Zustand, kann aber auf JVM nicht enforced werden. Eine Mailbox pro Actor: Ein Buffer für alle ankommenden Nachrichten. Asynchones Senden. Empfangsverhalten: Reaktion auf ankommende Nachricht. Behandlungsmethode wird ausgeführt. Effekte per Behandlung (Ändere Zustand, Sende Nachricht, Erzeuge neue Actors). Intern sequentiell, nur eine Nachricht auf einmal behandelbar. public class Printer extends UntypedActor { public void onReceive(final Object message) { if (message instanceof Integer) { } } <i>// Erzeugen und Senden</i> ActorSystem system = ActorSystem.create("System"); ActorRef p = system.actorOf(Props.create(Printer. class)); for (int i = 0; i < 100; i++) { printer.tell(i, ActorRef.noSender()); <i>// simple async</i> } System.shutdown();
Anwendung: Alternative zu Threads, Transaction-Processing, Backend für Service, Kommunikations-Hub.
Actor Hierarchies: Passend zu URL Adressierungsschema. Erzeuger ist Parent. <i>ActorSelection</i> selektiert Teilbaum, broadcast möglich.
Actor Remoting: Remote Lookup: <i>system.actorSelection</i> mit URL. Leichtgewichtiger als <i>ActorRef</i> . Kann 0..n Actors umfassen und zu <i>ActorRef</i> aufgelöst werden. Remote Erzeugen: <i>system.actorOf()</i> , <i>application.conf</i> spezifiziert, wo Actor erstellt wird. Keine Codeänderungen.
System Shutdown
getContext().stop(actorRef); <i>// Stoppt nach Bearbeitung</i> getContext().stop(getSelf()); <i>// Rekursiv</i> getContext().system().terminate(); actor.tell(PoisonPill.getInstance(), sender);

GPU Parallelisierung
512, 1024, 3584, 5760 Cores. Sehr spezifische langsamere Prozessoren.
GPU Aufbau
SP (Streaming Processor): 8-192 SPs pro SM SM (Streaming Multiprocessor): z.B. 1-30 SM SIMD: SM ist prinzipiell SIMD (Single Instruction Multiple Data), Vektorparallelität Alle Cores führen gleiche Instruktion aus, Einzelne können sie auch nicht ausführen.
GPU vs. CPU
GPU: hohe Datenparallelität, wenig Verzweigungen, Kein beliebiges Warten, Kleine Caches pro Core. Ziel: Hoher Gesamtdurchsatz CPU: Gegenteil, Ziel: Niedrige Latenz pro Thread
NUMA Modell
Non-Uniform-Memory-Access: Kein gemeinsamer Hauptspeicher zwischen GPU und CPU. Explizites Übertragen.
CUDA - Computer Unified Device Architecture
CUDA Blocks: Threads sind in Blöcke gruppiert. Blöcke sind im gleichen SM. Threads können innerhalb Block interagieren. Ausführungsmodell: Thread: virt. Skalarprozessor. Block: virt. Multiprozessor, Blöcke müssen unabhängig sein. Run To Completion. Beliebige Ausführungsreihenfolge. Grad der Parallelität durch GPU bestimmt. Automatische Skalierung, Ablauf: 1. Auf GPU allokieren <i>cudaMalloc</i> , 2. Daten au GPU transferieren <i>cudaMemcpy</i> , 3. Kernel ausführen, 4. Rücktransfer, 5. Auf GPU deallokieren
Datenaufteilung
threadIdx.x: ThreadId im Block. blockIdx.x: Nummer des Blocks. blockDim.x: Blockgrösse. Programmierende modellieren Datenaufteilung selber.
Boundary Check
Falls Mehr Threads als zu bearbeitende Daten. Threads mit $i \geq N$ dürfen nicht auf Daten zugreifen.
Unified Memory
Automatischer Transfer CPU - GPU.
Speicherstufen
Shared Memory: Per SM. Schnell ca 4 Zyklen. Nur zwischen Threads innerhalb Block sichtbar. Paar KB. <i>__shared__ float x;</i> Global Memory: Main Memory in GPU. Langsam, ca. 400-600 Zyklen. Allen Threads sichtbar. Mehrere GB. <i>cudaMalloc()</i>
Block Barriere
<code>__syncThreads();</code>
In if/else nur falls für alle Threads eines Blocks.
Warp
Block wird intern in Warps zerlegt (je 32 Threads). Ausführung: SIMD. SM kann alle Warps eines Blocks beherbergen. Nur wenige laufen echt parallel (1 bis 24).
Divergenz
Unterschiedliche Verzweigungen im selben Warp. <i>if/switch/while/-do/for</i> . SM führt Instruktionen der einen Verzweigung durch, dann Instruktionen einer anderen Verzweigung. Performance Problem Schlechter Fall: Divergenz innerhlab derselben Warp. Guter Fall: Gleiche Verzweigung im Warp.
Memory Coalescing
Zugriffsmuster für Performance optimieren. Burst: Falls Threads auf 32-Byte-Bereiche zugreifen. Sonst teure Einzel-Zugriffe. (je 400 Zyklen pro Global Memory)
Cluster Parallelisierung
Motivation: möglichst hohe Beschleunigung. Viele CPU Cores statt nur viele GPU Cores. Computer Cluster: Verbund leistungsfähiger Rechenknoten. Meist gleichartig und fest verbunden an einem Standort. Sehr schnelles Interconnect. HPC Anleitung: Programcode uploaden, Auf Cluster Kompilieren, HPC-Job lancieren, Job-Ende abwarten, Job-Resultat anschauen. Verteiltes Programmiermodell: Programm auf mehreren Nodes ausführen. Kein Shared Memory (NUMA) zwischen nodes, nur für Cores im Node (SMP). Message Passing Interface (MPI): Basiert auf Actor/CSP Prinzip.
<pre>MPI_Init(&argc, &argv); // MPI Initialisierung MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Prozess Identifikation MPI_Finalize(); // MPI Finalisierung MPI_Send(&val, 1, MPI_INT, recRank, tag, MPI_COMM_WORLD); MPI_Recv(&val, 1, MPI_INT, senderRank, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE); MPI_Barrier(MPI_COMM_WORLD); // Wartet auf alle Prozesse MPI_Allreduce(&val, &total, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD); // Aggregation von Teilresultaten MPI_Reduce(&val, &total, 1, MPI_INT, MPI_SUM, recRank, MPI_COMM_WORLD); // Effizienter, kein Broadcast</pre>
SPMD: Single Program Multiple Data. MPI Programm wird in mehrere Prozesse gestart. Prozesse können untereinander kommunizieren. Communicator: Gruppe von MPI-Prozessen. Communicator World: Alle Prozesse einer MPI-Programmausführung.
Concurrency in Python
GIL: Global Interpreter Lock. Nur ein Thread kann Python Byte-Code ausführen. Kein Speedup für CPU-Bount Operationen möglich. Data Races durch Reordering dennoch möglich, Visibility nicht garantiert. Kein definiertes Memory Model.

<pre>from threading import Thread from multiprocessing import Process if __name__ == '__main__': t = Thread(# p = Process target=fibonacci, args=(10,)).start(); t.join(); # Shared Memory res = Value('i', -1, lock=False); // Typ, InitialWert, lock</pre>
Pools
Threads: <i>concurrent.futures.thread.ThreadPoolExecutor</i> Prozesse: <i>concurrent.futures.process.ProcessPoolExecutor / multiprocessing.Pool</i>
<pre>if __name__ == '__main__': with ProcessPoolExecutor() as pool: future = pool.submit(fib, 10) # or: map(fib, [1,2,3]) print(future.res()); # Blockiert bis Task-Ende</pre>
Asynchrone Programmierung (asyncio)
Coroutine-Functions: werden erst beim <i>await</i> ausgeführt. Keine parallele Ausführung. Ausnahme: Coroutine wird als Task verpackt.
<pre>async def sub_routine(n): await asyncio.sleep(n); if __name__ == 'main__': asyncio.run(sub_routine(1))</pre>
JavaScript Concurrency
Grundsätzlich Single-Threaded mit einem Event-Loop. Kein Schutz vor Race-Condition.
<pre>function delay(ms) { let promise = new Promise((resolve, reject) => { setTimeout(() => resolve(), ms); }); } async function countTo(n) { for (let = 1; i <= n; i++) { await delay(1000); } }</pre>
Web-Worker
Entsprechen einem Thread im Browser. Datenaustausch primär über Messaging. Werden mit Quelldatei gestartet. Langlebiger Prozess mit eigenem Event-Loop.
<pre>onmessage = event => { // Worker definieren const n = event.data; const res = fib(n); postMessage(res); } const worker = new Worker('w.js'); // Worker verwenden worker.onmessage = event => { console.log(event.data); } worker.postMessage(42);</pre>