

Multithreading Grundlagen

Hyper-threading

Multi-Core

Multi-Processor

Rechner-Netzwerk

Distanz der Cores

1. Alle prozessorinterne Ressourcen werden ausgenutzt

2. Benutzen des Netzes

Processor Chip

Processor Chip

Computer

Computer

Computer

Computer

Reg. Set

Reg. Set

Core Unit

Core

Core

CPU

GPU

CPU

Phi

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 accesses
    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 möglich
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 möglich
    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();
    }
}
```

```
}  
// Ausführung  
var threadPool = new ForkJoinPool();  
int res = threadPool.invoke(new CountTask(2, N)); // blockiert
```

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 weitermachen.

```
// Klassisch  
Future<long> future = threadPool.submit(() -> { });  
// other work  
process(future.get());  
// Modern  
CompletableFuture<long> future =  
    CompletableFuture.supplyAsync(() -> { });  
// other work  
process(future.get());
```

Ende des async Aufrufs
Caller-zentrisch (Pull): Caller wartet 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.
// thenApply() fuer Continuation mit Rueckgabe
future.thenAccept(res -> System.out.println(res));

Ausführung: durch beliebigen Thread, durch Initiator, wenn Future bereits Resultat hat.
Asynchrone Continuations: thenAcceptAsync() bzw. thenApplyAsync().

Multi-Continuation
CompletableFuture.allOf(f1, f2).thenAcceptAsync(() -> { });
CompletableFuture.any(f1, f2).thenAcceptAsync(() -> { });

Fire and Forget
Task starten, ohne das Ende abzuwarten. Submitter ruft kein get() oder join() auf.

CompletableFuture.runAsync(() -> { });

Daemon Workers: Workers Threads in Fork-Join-Pools sind Daemon. Anwendung kann vor Task-Ende stoppen.
Ignorierte 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. Wait() in Schlaufe. PulseAll() bi mehreren Bedingungen oder Erfüllungen mehrerer Threads. Synchronisation mit HilfsObj als Best Practice.

```
private object syncObject = new(); // Monitor auf HilfsObj  
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);  
    }  
}
```

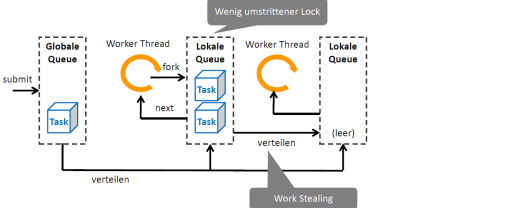
.NET Synchronisationsprimitiven

Fehlen: kein Fairnessflag, kein Lock & Condition.
Zusätzlich: ReadwriteLockSlim für Upgradeable Read/Write. Semaphoren auch auf OS-Stufe nutzbar. Mutex (binärer Semaphor).
Collections nicht Thread-safe: Ausser System.Collections.Concurrent

.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();  
// Task mit Rueckgabe  
Task<int> task = Task.Run(() => { return 0; });  
Console.WriteLine(task.Result); // Blockiert  
// Geschachtelte Tasks  
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.

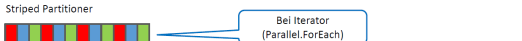
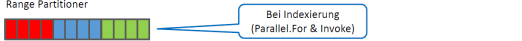
```
Parallel.Invoke(  
    () => MergeSort(1, m); () => MergeSort(m, 1);  
);
```

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.



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 Language-Integrated Query. Pendant zu Java Steam API. Keyword: AsParallel()

```
from book in bookCol.AsParallel()  
where book.Title.Contains("Java") select book.ISBN
```

Asynchrone Programmierung mit TPL

Task Continuation

```
task1.ContinueWith(task2).ContinueWith(task3);  
// Multi-Continuation  
Task.WhenAll(task1, task2).ContinueWith(continuation);  
Task.WhenAny(task1, task2).ContinueWith(continuation);
```