

Multithreading Grundlagen

Hyper-threading

Multi-Core

Multi-Processor

Rechner-Netzwerk

Distanz der Cores

Falsch: jede Aufgabe bekommt mind. 1. eigenen Sockel

2. eigener Sockel

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) (*Standart 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 freigeben.

```
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 Programmierfehler, 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). External 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 Warteeabhängigkeiten
Deadlock Vermeidung
Lineare Sperrordnung der Ressourcen einführen. Nur geschachtelt in aufsteigender Reihenfolge sperren. Eliminiert zyklische Warteeabhängigkeiten.
Grobgranulare Locks wählen. Wenn lineare Sperrordnung nicht möglich/sinvoll 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.
```