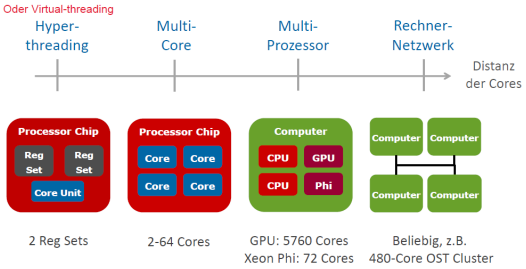
1 Multi-Threading Grundlagen



Parallelität (Parallelism): Mehrere Teilabläufe, welche gleichzeitig auf mehreren Prozessoren/Cores laufen. → Schnellere Programme
Nebenläufigkeit (Concurrency): Gleichzeitig oder verzahnt ausführbare Abläufe, welche auf gemeinsame Ressourcen zugreifen (Logisch unabhängig) → Einfachere Programme
User-Level Threads: Im Prozess implementiert (keine echte Parallelität)
Kernel-Level Threads: Im Kernel implementiert (Multi-Core Ausnutzung) → Kontextwechsel vom Prozess per SW-Interrupt

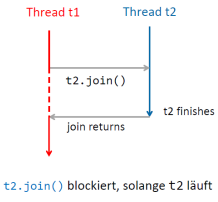
1.1 Thread Scheduling

Processor Sharing: Mehr Threads als Prozessoren und bei Wartebidingung Proz. an anderen bereiten Thread abgeben.
Verzahnte Ausführung: Prozessor führt Instruktionen von mehreren Threads in Teilsequenzen aus. → Quasiparallelität
Synchron: Warten auf Bedingung → Waiting Threads
Asynchron: Zeitablauf → Nach gewisser Zeit Proz. abgeben
Kooperativ: Threads müssen explizit beim Scheduler in Abständen Kontextwechsel synchron initiieren
Preemptiv: Scheduler kann per Timer-Interrupt den laufenden Thread asynchron unterbrechen

1.2 Multi Thread Programmierung

JVM ist ein Prozess im Betriebssystem. → Programmierer kann weitere Threads starten
Die JVM läuft, solange Threads laufen.
Ausnahme: Deamon Threads (z.B. Garbage Collector). Mit System.exit()/Runtime.exit() kann die JVM direkt terminiert werden (unsauber)

```
// A,B Ausgaben können durcheinander sein:
public class MultiThreadTest {
   public static void main (String[] args ) {
      var a = new Thread(Thread(() -> multiPrint("A"));
      var b = new Thread(Thread(() -> multiPrint("B"));
      a.start(); b.start();
      System.out.println("main finished");
   }
   static void multiPrint (String label) {
      for (int i = 0; i < 10; i++) {
         System.out.println(label + ": " + i);
      } } }
// Explizite Runnable-Implementation:
class SimpleLogic implements Runnable {
   @Override
   public void run () {
      // thread behavior
   } }
var myThread = new Thread(new SimpleLogic());
myThread.start();
// Sub-Klasse von Thread
class SimpleThread extends Thread {
   @Override
   public void run ()
      // thread behavior
   } }
var myThread = new SimpleThread(); myThread.start();
```



t1 wartet, bis t2 fertig ist.
Warten:
Thread.sleep(milliseconds)
Prozessor freigeben:
Thread.yield()
Von aussen Unterbrechen:
myThread.interrupt()
Aktuelle Thread-Instanz:
Thread currentThread()
Als Deamon markieren:
void setDeamon(boolean on)

2 Thread synchronisation

Threads teilen sich Adressraum und Heap.

2.1 Race Condition

Wenn mehrere Threads das Selbe Objekt lesen und anhand vom Resultat dieses überschreiben. (z.B. deposit: zuerst lesen dann += amount → Zwischen lesen und schreiben könnte ein anderer Thread die balance geändert haben)
Lösung: Gegenseitiger Ausschluss (Mutual Exclusion)
→ Geht nicht mit einfachem locked Boolean (Keine atomaren Instruktionen)

2.2 Synchronized (Monitor Lock)

Das Keyword synchronized belegt einen Lock für das Objekt.
→ Nur ein Thread kann eine synchronized Methode in derselben Instanz zur gleichen Zeit ausführen.

```
// deposit und withdraw sind im gegenseitigem Ausschluss
class BankAccount {
   private int balance = 0;
   public synchronized void deposit (int amount) {
      this.balance += amount }
   public synchronized boolean withdraw (int amount) {
      if (amount <= this.balance) {
         this.balance -= amount;
         return true; } {else {return false; } } }
// synchronized(object){ statements } ist eine explizite
      Angabe, auf welcher Instanz gelockt wird:
public void deposit(int amount) {
   synchronized(this) {this.balance += amount;} }
```

Problem: Wenn man auf Bedingung warten will. → sleep() und yield() geben den Monitor-Lock nicht frei.
Lösung: Wait & Signal Mechanismus. → Threads können im Monitor auf Bedingung warten oder können Wartende Threads aufwecken (Änderung signalisieren)
wait() gibt lock frei und wartet auf Signal. notify() signalisiert nur einen beliebigen (zufälligen), wartenden Thread. → Wartet vlt auf andere Bedingung und kann zu ewigem Warten führen
notifyAll() signalisiert alle wartenden Threads.
Wichtig:

- Signalisierender Thread behält Monitor
- Thread ist im inneren Warteraum, bis Signal kommt
- Bei notify() kommt Thread wieder in äusseren Warteraum

```
class BoundedBuffer <T> {
  private Queue<T> queue = new LinkedList<>();
  private int limit = 1; // or initialize in constructor
  public synchronized void put(T item) throws
        InterruptedException {
    while (queue.size () == limit) {
      wait(); // await non full
    } queue.add(item); notifyAll(); } // signal non empty
  public synchronized T get() throws
        InterruptedException {
    while (queue.size () == 0) {
      wait();// await non empty
    } var item = queue.remove(); notifyAll(); // non full
    return item; } }
```

3 Spezifische Synchronisationsprimitiven

3.1 Semaphor

Ziel: Vergabe einer beschränkten Anzahl freier Ressourcen. Semaphor ist also ein Objekt mit Zähler.
acquire(): Bezieht freie Ressource. Wartet, falls keine verfügbar (Zähler <= 0). Dekrementiert Zähler wenn erfolgreich.
release(): Ressource freigeben. Zähler inkrementieren.
Verwendung: new Semaphore(N); oder new Semaphore(N, true); → true für Fairness (FIFO-Prinzip), aber langsamer

```
// Mutex Semaphore als alternative zu synchronized
class BoundedBuffer <T> {
  private Queue<T> queue = new LinkedList<>();
  private Semaphore upperLimit=new Semaphore(cap, true);
  private Semaphore lowerLimit = new Semaphore(0, true);
  // private Semaphore mutex = new Semaphore(1, true);
  public void put(T item) throws InterruptedException {
    upperLimit.acquire();
    synchronized (queue) {queue.add (item);}
    // mutex.acquire(); queue.add(item); mutex.release();
    lowerLimit.release(); }
  public T get() throws InterruptedException {
    T item; lowerLimit.acquire();
    synchronized (queue) { item = queue.remove(); }
    // mutex.acquire(); T item = queue.remove(); mutex.
         release();
    upperLimit.release(); return item; } }
```

3.2 Lock & Condition

Ziel: Monitor mit mehreren Wartelisten für verschiedene Bedingungen. Braucht kein synchronized, wait, notify, notifyAll

```
class BoundedBuffer<T> {
  private Queue<T> queue = new LinkedList<>();
  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(); } }
```

3.3 Read-Write Lock

Ziel: Gegenseitiger Ausschluss ist unnötig streng für rein lesende Abschnitte. → Erlaube parallele Lese-Zugriffe, Gegenseitiger Auschluss bei Schreiben.

```
var rwLock = new ReentrantReadWriteLock(true);
rwLock.readLock().lock(); // read-only accesses
rwLock.readLock().unlock();
rwLock.writeLock().lock(); // read and write accesses
rwLock.writeLock().unlock();
```

3.4 Count Down Latch

Ziel: Synchronisationsprimitive mit Count Down Zähler. Threads können warten, bis Zähler <= 0 ist. countDown(), um Zähler zu dekrementieren aber kein countUp().

```
// CountDown Latch
var ready = new CountDownLatch(N); //Warte auf N Threads
var start = new CountDownLatch(1); // Einer gibt Signal
// Cyclic Barrier
var start = new CyclicBarrier(N); // Kein start nötig
```

4 Gefahren der Nebenläufigkeit

4.1 Race Condition

Problem: Ungenügend synchronisierte Zugriffe auf gemeinsame Ressourcen. → Falsche Resultate oder falsches Verhalten möglich. (meistens wegen Data-Race)
Ohne Data Race weil: Critical Sections nicht geschützt.
Kombinationen:
Data Race + Race Condition: Fehlerhaftes Programmverhalten
Data Race ohne Race Condition: Korrektes Programmverhalten, aber formal falsch
Race Condition ohne Data Race: Fehlerhaftes Prog.verhalten
Weder noch: Richtig
Wann kann man auf Synchronisation verzichten?
Immutability (Unveränderlichkeit): Instanzvariablen sind alle final, Methoden mit nur Lesezugriff → Konstruktor initialisiert die Instanzvariablen.
Confinement (Einsperrung): Objekt gehört nur einem Thread zu einer Zeit. Oder Objekt ist in anderem bereits synchronisiertem Objekt eingekapselt

4.2 Deadlocks

Problem: Gegenseitiges Aussperren von Threads.

```
class BankAccount {
  private int balance;
  public synchronized void transfer(Acc to, int amount)
       {
    balance -= amount;
    to.deposit(amount); } // implizit geschachtelter Lock
  public synchronized void deposit (int amount) {
    balance += amount; } }
// Thread 1          Thread 2
a.transfer(b, 20);   b.transfer(a, 50);
```

Deadlock Vermeidung:

- Lineare Sperrordnung der Ressourcen einführen → Nur geschachtelt in aufsteigender reihenfolge sperren
- Grobgranulare Locks wählen → Wenn lineare nicht möglich

4.3 Starvation

Problem: Kontinuierliche Fortschrittsbehinderung von Threads wegen Fairness-Problem. → Ist ein Liveness/Fairness Problem
Starvation Vermeidung:

- Länger wartende Threads haben Vortritt
- Fairness einschalten mit Semaphore, Lock & Condition, rtead-Write Lock
- Java Monitor hat ein Fairness-Problem