Contents

1 Multi-Threading Basics

async VS. parallel *Attention*: Asynchron (Concurrent) ist not the same as parallel (see 1). Concurrent means that one core runs multiple threads. One thread makes progress while the other thread waits for IO. Parallel means that n cores run n threads at the exactly same time.

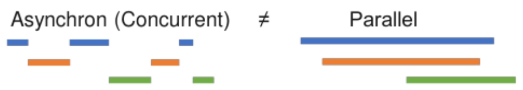


Figure 1: Async vs Parallel

User-Level Threadings User-Level Threads are implemented in a kind of runtime. The runtime itself runs in a process. Using this approach no real parallelism is possible, only one process).

Kernel-Level Threading Today you normally work with Kernel-Level Threads. The thread mechanism is implemented directly in the kernel. The programming language only interacts with the kernel API.

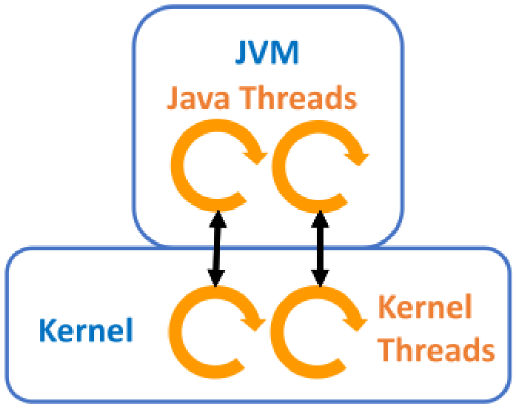


Figure 2: Kernel-Level Threads

Processor Sharing In modern system you have more threads than processors. That implies that not every thread can work at any time it wants. Therefore, the threads must *share* the processor.

Multi-Tasking The processor has two strategies how the processor sharing should work.

- cooperative
- preemptive

The preemptive is the used model on the processor. However, the most changes are cooperative because the thread has to wait for other resources (memory, IO, ...).

Cooperative Multi-Tasking In this model the thread works as long as it wants. If the thread want to wait, it must initiate the process by itself. A scheduler can not interrupt a running thread!

This model is not very common on modern systems.

Preemptive Multi-Tasking The scheduler uses a *Timer Interrupt* to interrupt a running thread. Each Thread can work for a specific max. interval. After that interval is over and the thread is not finished, the thread is interrupted and is added to the *Ready-Queue*.

This is the common model which is today used.

Thread State

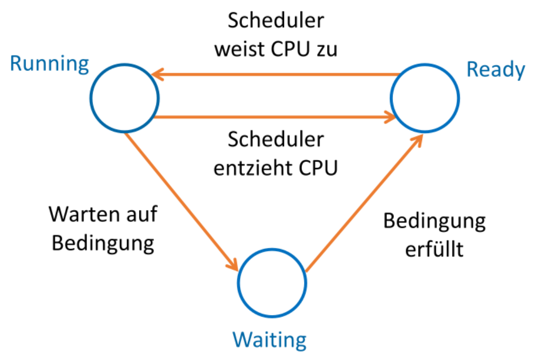


Figure 3: Thread States

JVM Thread Model The JVM is a single process system and has many threads (main, daemon threads, ...).

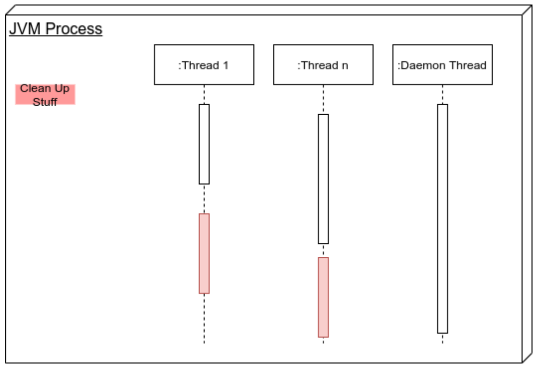


Figure 4: The JVM Thread Model

JVM Termination The JVM runs as long as at least one non-daemon thread is running. If the last non-daemon thread is terminated then the JVM is shutdown and all daemon threads are killed uncontrolled. Therefore, it is not possible to implement something like goodbye message when the thread terminates.

If inside a thread an uncaught exception occurs the other threads are **not** terminated and keep working.

InterruptedException Java In Java `join` and `sleep` can throw (theoretically) a checked `InterruptedException`. This exception is never thrown except you call `myThread.interrupt()`. Therefore, this exception should only be used if you implement a cooperative canceling. Otherwise, `xx.interrupt()` and the exception are indices for a code smell.

2 Monitor Synchronization

synchroniyed keyword In Java every object has a lock called Monitor-Lock. At the beginning of a `synchronized` block you acquire the lock as long the lock is free. If the lock is already acquired then you have to wait. After the `synchronized` block the lock is release (in **every** case)

```
public class Test {
    synchronized void do_something() {
        /* do stuff */
    }

    static synchronized void
        do_other_stuff() { }

}

public class Test2 {
    static void do_other_stuff() {
        synchronized(Test2.class) {}
    }

    void do_something() {
        synchronized(this) {}
    }
}
```

Monitor lock The monitor is a synchronisation mechanism which use the *Wait & Signal* mechanism.

1. Fight for lock
   - winner: enters the monitor (acquire the lock)
   - others: wait for entrance
2. Thread in monitor does work
   - if done: leaves monitor and wakes up other threads (`signal` / `signalAll()`)
   - condition not satisfied: leaves monitor and goes to the right and wake up other threads (Wait on signal)
3. Go to 1.

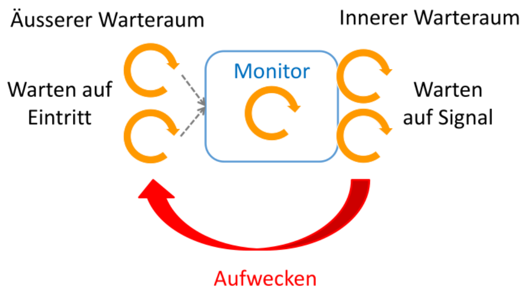


Figure 5: A monitor

Wait Function in Loop The function `wait()` must be always called inside a loop. In a simple if condition the following could happen:

1. thread 1: acquire lock, but condition is not fulfilled -> wait()
2. thread 2: acquire lock, change something, but thread 1 condition not fulfilled
3. thread 1: acquire lock again, make progress even the condition is not fulfilled

notify vs notifyAll A single notify can only be used when:

1. every thread has the same condition
2. only one **single** thread can make progress (One-In/One-Out)

Thread Wake Up A thread can be woken up by

1. notifyAll(), notify()
2. InterruptedException
3. Spurious Wake up (falsely wake up POSIX Thread API)

Implement Monitor Lock

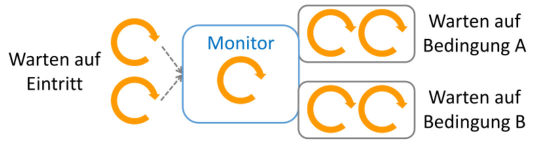
```
class BankAccount {
    private int balance = 0;

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

    public synchronized boolean withdraw
        (int amount) throws
        InterruptedException {
        int timeout = 0;
        while(amount > balance) {
            if (timeout >= 3) {
                return false;
            }
            wait();
        }

        balance -= amount;
        return true;
    }

    public synchronized int getBalance()
         {
        return balance;
    }
}
```

3 Specific Synchronization Mechanism

Semaphore A semaphore is a synchronisation mechanism. A semaphore is basically a counter which can not count down below 0.

- `acquire()`:
  - acquire resource from semaphore
  - wait, as long the counter is <= 0
  - otherwise, decrease counter
- `release()`:
  - release ressource
  - increment counter

Lock and Condition In lock & condition you have a monitor, but instead of one queue you have a queue for each condition. This has the benefit that not always all threads must be woken up. Only the threads where the condition may be now fulfilled.



:

fig:lock-and-conditions

Lock and Condition Example

```
import java.util.concurrent.locks.
    Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.
    ReentrantLock;
```

```
public class WarehouseWithLockCondition
    {
    private Lock lock;
    private Condition nonEmpty;
    private Condition nonFull;

    public WarehouseWithLockCondition(
        int capacity, boolean fair) {
        this.capacity = capacity;
        store = 0;
        lock = new ReentrantLock(fair);
        nonEmpty = lock.newCondition();
        nonFull = lock.newCondition();
    }

    @Override
    public void put(int amount) throws
        InterruptedException {
        lock.lock();
        try {
            while (store + amount >
                capacity) {
                nonFull.await();
            }
            store += amount;
            nonEmpty.signalAll();
        } finally {
            lock.unlock();
        }
    }

    @Override
    public void get(int amount) throws
        InterruptedException {
        lock.lock();
        try{
            while (store - amount < 0) {
                nonEmpty.await(); }
            store -= amount;
            nonFull.signalAll();
        }
        finally {
            lock.unlock();
        }
    }
}
```

Read-Write Locks Mutual exclusion is too strong when only reading happens. Mutual exclusion is only required when minimal one thread wants to write.

Parallel	Read	Write
Read	Yes	No
Write	No	No

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

Problems With Read-Write Locks The benefit of read-write locks is that you not have always mutual exclusion. But you can use the read lock only then, when **NO** write access happens on a shared object. But in common languages like Java it is not possible to encode the read only property into the API. However, Rust encodes the mutability of a function directly into the function arguments.

Count-Down Latch The count down latch is synchronisation mechanism. It's a counter which counts down to zero.
Using `await()` you wait until the counter is 0. Using `countDown()` you decrement the counter.

```
CountDownLatch waitForAll = new
    CountDownLatch(this.CARS);
protected void waitForAllToBeReady()
    throws InterruptedException {
    waitForAll.await();
}

@Override
public void readyToStart() {
    waitForAll.countDown();
}
```

Cyclic Barrier A cyclic barrier is a synchronisation mechanism. The cyclic barrier is a meeting point for a fixed number of threads. You wait with the function `await()`. In contrast to the count down latch a cyclic barrier can be reused.

```
var gameRound = new CyclicBarrier(5);

/* 5 different players / threads */
while (true) {
    gameRound.await();
}
```

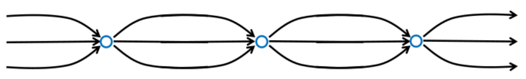


Figure 6: Cyclic Barrier

Rendez-Vous A rendez-vous is a special type of cyclic barrier with only 2 participants. Often you want to exchange data between those two threads. For this an `Exchanger` is used.

- Without exchange: `new CyclicBarrier(2)`
- With exchange: `Exchanger.exchange(something);`

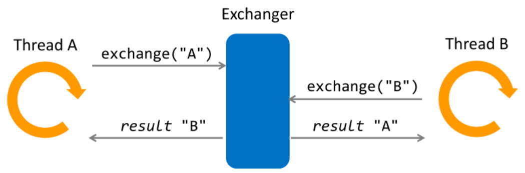


Figure 7: Exchanger Workflow

4 Threats of Concurrency

Race Conditions A race condition occurs then when multiple threads try to access the same resource without proper synchronization. This could result in wrong results or wrong behaviour. Often the reason is a data race. However, not always.

Deadlock A deadlock is when two or more threads waits on each other. This happens when thread 1 locks resource *a* and then resource *b* and thread 2 first resource *b* and then resource *a*.

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

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

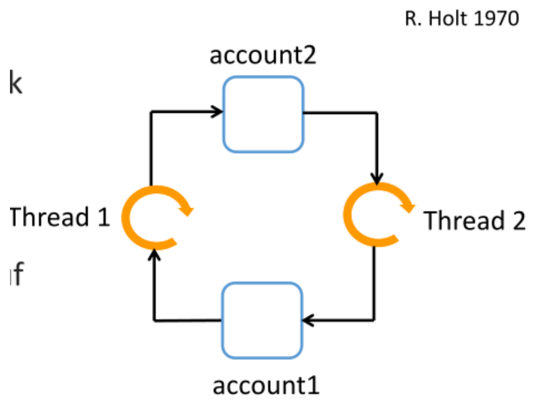


Figure 8: Deadlock detecion using resource graph

Starvation When a thread never can acquire the lock for a resource, even when the lock is release from time to time, we call this *starvation*. Therefore, it is **not** a deadlock but a fairness problem.

Data Race When unsynchronized access happens (Race Condition) on the same memory location we call this a data race. For example, when a thread reads from a variable and another thread writes to the same variable. A read-read access is **not** data race (it is also not dangerous).

```
/* Thread 1*/
balance += 100;

/* Thread 2*/
balance += 50;
```

Race Condtion withoput data race In the following code example we see, that no data race is possible. However, a race condition is still possible. The read and the subsequent write are not atomic. The thread can be interrupted between the read and write action. And the action from another thread gets lost (Lost Update).

```
class BankAccount {
    int balance = 0;
    synchronized int getBalance() {
        return balance; }
    synchronized void setBalance(int x)
        { balance = x; }
}

/* On multiple threads */
account.setBalance(account.getBalance()
    + 100);
```

No synchronization needed Synchronization is not required when:

- the object is immutable (read only access)
- with confinement

Confinement Under confinement, we understand a structure that guaranties that only one thread can access an object at time.

- *Thread Confinement*: an object belongs to a single thread and is used only be this thread
- *Object Confinement*: an object is encapsulated in an already synchronized object

```
class ProductDatabase {
    private HashMap<String, Product>
        productMap = new HashMap<>();
    public synchronized void addProduct(
        String name, String details) {
        productMap.put(name, new Product
            (details));
    }

    public synchronized
        getProductDetails(String name) {
        return productmap.get(name).
            getDetails(); // save,
            because String is immutable
    }

    public synchronized void notifySale(
        String name) {
        productMap.get(name).
            increaseSales();
    }
}
```

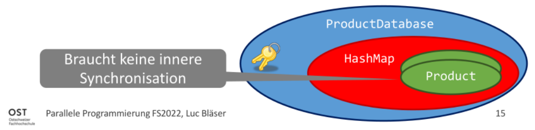

ProductDatabase
Braucht keine innere Synchronisation
HashMap
Product
OST
Parallele Programmierung FS2022, Luc Bläser
15

Figure 9: Example of object confinement

Livelock A live lock is a special case of a deadlock. While in a deadlock all relevant threads are sleeping in a live lock the threads are doing some waiting instruction. During these waiting instructions the CPU is working (busy wait).

Avoiding deadlocks To avoid deadlocks you have two possibilities:

- with a linear lock order
- coarse-grained locking (when linear lock order is not possible)

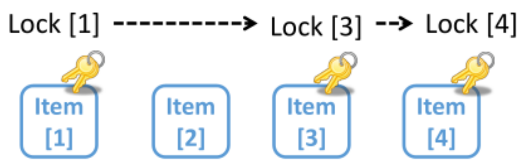


Figure 10: Linear Lock Order

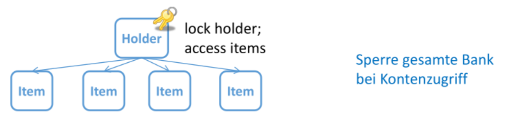
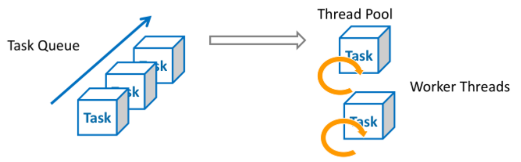

Holder
lock holder; access items
Item
Item
Item
Item
Sperre gesamte Bank bei Kontenzugriff

Figure 11: Coarse Grained Lock

5 Thread Pools

Thead Pool A thread pool consists of a task queue and n worker threads. A new task is inserted in the queue. The new free thread from the thread pool takes the first task from the queue and process it.



:

fig:thread-pool

Number of Threads in Thread Pool A thread pool has as much threads as processors (cores) and little bit more. A little bit more than the number of processors is ideally because when one thread has to wait for I/O another thread can use the processor:

$$\begin{aligned}&\#\text{Worker Threads} = \\ &\#\text{Prozessoren} + \#\text{Pendente I/O-Aufrufe}\end{aligned} \tag{1}$$

Thread Pool Java Today in modern Java you should only use the `ForkJoinPool`. If an exception occures then the exception is returnted to the caller of `get()`.

```
var threadPool = new ForkJoinPool();
// submit is non-blocking
Future<Integer> future = threadPool.
    submit(() -> {
        int value = 1;
        // long calculation
        return value;
    });

// get blocks / same as await in Rust
Integer i = future.get();
```

Task Parallelism .NET In .NET you have one thread pool for:

- task parallelization
- data parallelization
- async programming

```
Task task1 = Task.Run(() => { /* Do some
     stuff */ });
task1.Wait(); // blocking

// Task with return value
Task task2 = Task.Run(() => { return
    3;});
Console.Write(task.Result); // blocking
```

```
// Task with Sub Tasks
Task.Run(() => {
    var left = Task.Run(() => Count(
        leftPart));
    var right = Task.Run(() => Count(
        rightPart));
    int result = left.Result + right.
        Result;
    return result;
});

// Parallele Statements
Parallel.Invoke(
    () => MergeSort(l, m),
    () => MergeSort(m, r)
);

// Parallel Loop
Parallel.ForEach(list, file => Convert(
    file));

// Parallel For - only if iterations are
     independend
Parallel.For(0, array.Length, i =>
    DoComputation(array[i]));

// PLINQ
from book in bookCollection.AsParallel()
    where book.Title.Contains("
        Concurrency")
    select book.ISBN;

from number in inputList.AsParallel().
    AsOrdered()
    select IsPrime(number);
```

Parallel Loop .NET Often a loop has just a very short body. It is inefficient to execute every iteration in its own task. Therefore, the TPL groups automatically multiple bodies to a single task.

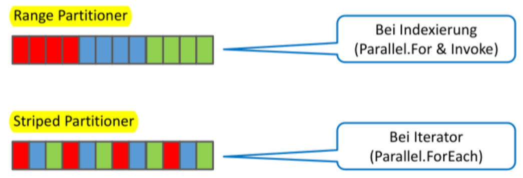

Range Partitioner
Bei Indexierung (Parallel.For & Invoke)
Striped Partitioner
Bei Iterator (Parallel.ForEach)

Figure 12: Partitioning in Parallel Loops

Work Stealing Pool Each Worker Thread has also its own task queue. A worker thread takes some task from the global queue and move it to the local queue. If the another worker thread wants to work but the global queue is empty, it steals it from other worker threads.

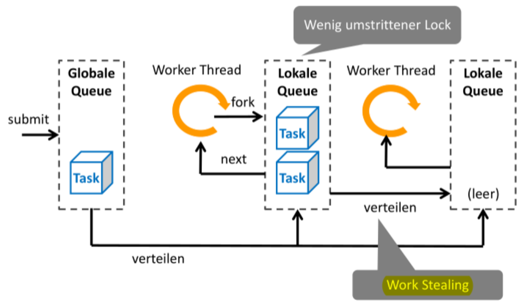


Figure 13: Work Stealing Thread Pool

Thread Injection When I injection new Threads in the thread pool during runtime is called *Thread Injection*. This can mitigate possible deadlocks when two task are not independent (only if not max threads is set).

TODO Some Stuff

6 Async Programming

Types of asynchrony You differ between two types of asynchrony:

- caller centric (pull)
  - The caller waits for task end and pulls the result
- callee centric (push)
  - task forwards the result to the successor task

Continuation in C#

```
Task.Run(task1).
    ContinueWith(task2).
    COntinueWith(task3);
```

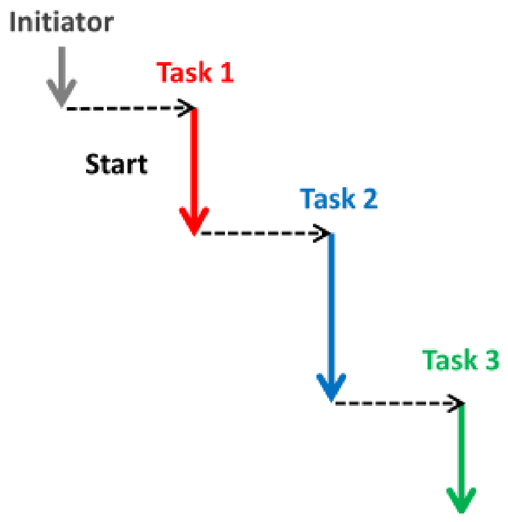


Figure 14: Task Continuations

```
Task.WhenAll(task1, task2).
    ContinueWith(continuation);
```

```
Task.WhenAny(task1, task2).
    ContinueWith(continuation);
```

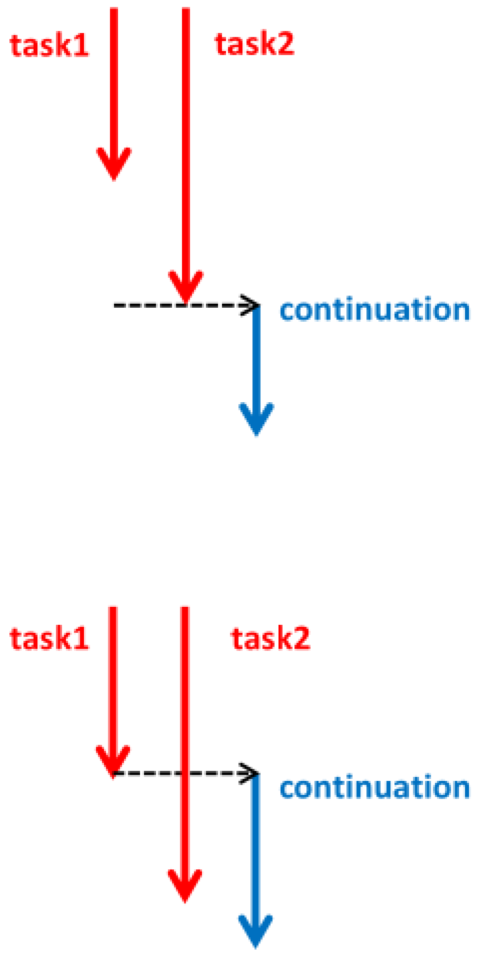


Figure 15: Multi Continuation

Exception Handling Continuation Exception in a Fire & Forget Tasks are ignored.

```
Task.Run(() => {
    throw new Exception("Some␣text"); //
        This exception is ignored by the
        system
});
```

To handle exceptions you could wait synchrony for finishing the task continuations (listening 1). Another option would be to subscribe for the `UnobservedTaskException`. However, this event is only triggered when the task is clean upped by the Garbage Collection. This can be at every point in time or even never (not deterministic).

```
task1.
    ContinueWith(task2).
    ContinueWith(task3).
    Wait();
```

Listing 1: Synchron Wait for finishing task continuation

CompletableFuture Java `CompletableFuture` is how you write async programs in modern Java and is the Java part of `Task` from C#.
The `CopletableFuture` has the methods listed in **??**.

For Multi Continuation you have the following methods:

- `CompletableFuture.allOf(future1, future2)`
- `CompletableFuture.any(future1, future2)`

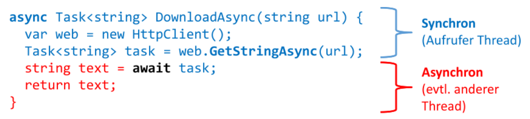
Other than .NET you can perform proper exception handling in continuations with `exceptionally()`.

Update UI .NET In WPF only the UI Thread is allowed to update the UI. Using a Dispatcher you can run an operation on the UI Thread from the background. In a WPF Project you can access the dispatcher:

- Code Behind: `this.Dispatcher`
- Other classes: `Application.Current.Dispather`

Aysnc in .NET A async function in .NET can only have the following return types:

- `Task`
- `Task<T>`
- `void` (fire and forget, should only be used in exceptional cases)

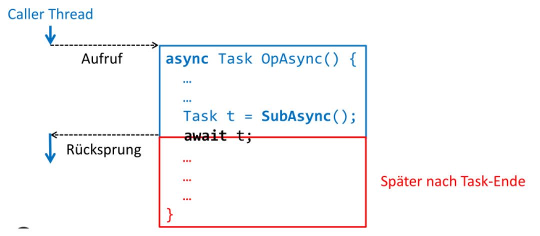


```
async Task<string> DownloadAsync(string url) {
  var web = new HttpClient();
  Task<string> task = web.GetStringAsync(url);
  string text = await task;
  return text;
}
```

Figure 16: Async / Await Execution Modell

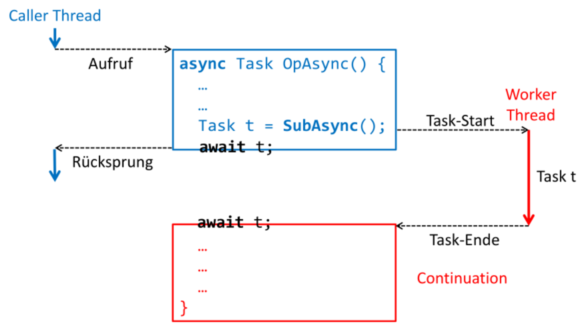
async / await execution model .NET The compiler splits the async function in multiple parts:

1. section before await: this section is executed synchrony
2. section after await: This is executed **after** the task is finished as a continuation.



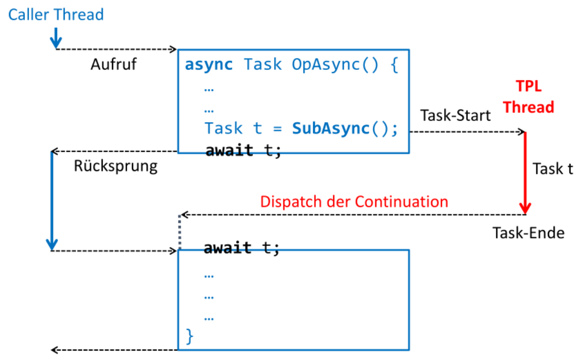
```
async Task OpAsync() {
  …
  …
  Task t = SubAsync();
  await t;
  …
  …
  …
}
```

Figure 17: Async Method Call



```
async Task OpAsync() {
  …
  …
  Task t = SubAsync();
  await t;

  await t;
  …
  …
  …
}
```

(a) Non-UI Thread Execution



```
async Task OpAsync() {
  …
  …
  Task t = SubAsync();
  await t;

  await t;
  …
  …
  …
}
```

(b) UI Thread Execution

Figure 18: Thread Execution

7 Memory Model

Lock Free Programming Lock Free Programming is a type of programming where you write concurrent programs without any lock mechanism. In lock free programming you use the guaranties of the memory model. The goal of lock free programming is to create efficient synchronization.

Weak Consistency The memory access to the same memory address is seen differently by different threads. The possible output for j and i in 19 are:

- i = 0, j = 0
- i = 1, j = 1
- i = 0, j = 1
- i = 1, j = 0

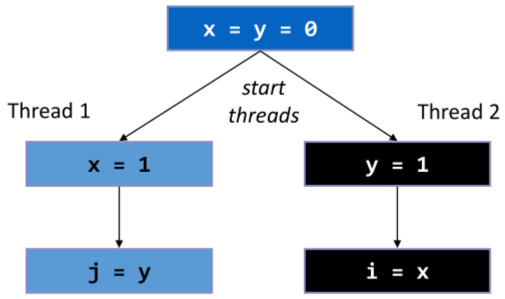


Figure 19: Example for Weak Consitency Example

The reason for this is that the compiler, runtime systems, CPUs, etc. can reorder instructions for optimization purpose. **Expect**, you use synchronization or memory barriers. This is called **weak consistency**.

Atomicity Guaranty Java Single reads / writes are atomic for:

- primitive data types until 32 bits
- object references
- long and double only with the `volatile` keyword

Visibility Guaranty Java
Java guaranties the following visibility:

- changes before release are visible at acquire (20)
- changes until write are visible at read (21)
- the thread sees the correct start values and Join the output of the thread
- initialization of final variables (only relevant if you get the object from a data race!)

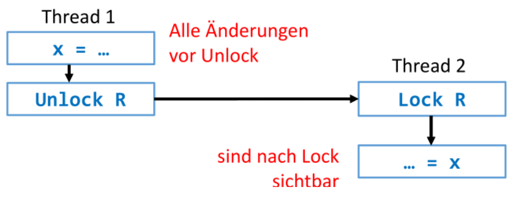


Figure 20: Visibility Lock / Unlock

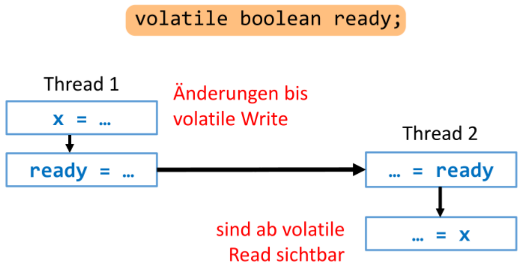
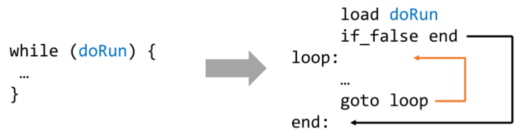


Figure 21: Visibility Volatile Write / Read



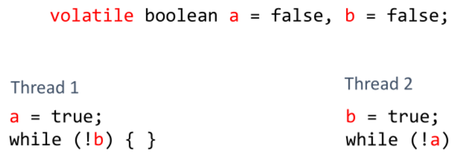
```
while (doRun) {
  …
}
```

```
     load doRun
     if_false end
loop:
     …
     goto loop
end:
```

Figure 22: Visibility Problems

Ordering Guaranty Java The order of the visibility is the same as in visibility. Additional:

- synchronization instructions are never reordered to each other
- Lock/Unlock, volatile, thread start / join are never reordered
- if everything is a synchronization mechanism than we talk about total order



```
volatile boolean a = false, b = false;
```

Thread 1

```
a = true;
while (!b) { }
```

Thread 2

```
b = true;
while (!a) { }
```

Figure 23: Total Order Rendez-Vous

Atomic Actions Java Java has for each type an Atomic class. This class provides many functions which are atomic.

- `getAndSet()`
- `compareAndSet()`
- `set()`
- …

Optimistic Synchronization Java In optimistic synchronization you perform the operation on a snapshot. If the snapshot is still the same value after the (long) operation you can update it. Otherwise, you have to do the same steps again until the fetch value is equal.

```
do {
    oldValue = var.get();
    newValue = caculateChanges(oldValue)
        ;
} while(!var.compareAndSet(oldValue,
    newValue));


// possible implementation for
    updateAndGet on the AtomicXXX
void updateAndGet(lambda
    calculateChanges) {
    do {
        oldValue = this.get();
        newValue = calculateChanges(
            oldValue);
    } while (!var.compareAndSet(oldValue
        , newValue));
}
```

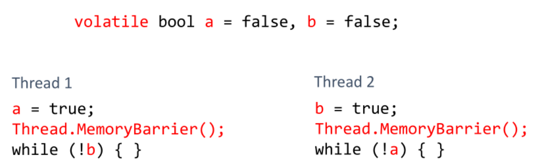
.NET Memory Model In general the .NET Memory Model is the same as in Java with a few exepctions:

- atomicity: long / double are not atomic, event with volatile
- visibility: not defined, implicit defined by ordering
- ordering: only half and full fences

The .NET version of the Atomic classes in Java is the `Interlocked` class.

```
Thread.MemoryBarrier();
```

Listing 2: .NET Full Fence

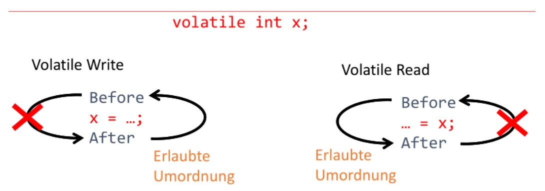


```
volatile bool a = false, b = false;

Thread 1                           Thread 2
a = true;                          b = true;
Thread.MemoryBarrier();            Thread.MemoryBarrier();
while (!b) { }                     while (!a) { }
```

Figure 24: Rendez-Vous in .NET

Volatile Half Fences .NET The `volatile` keyword in .NET is only a Half Fence. That means

- volatile writes: previous access stay before
- volatile reads: following access remain after



```
volatile int x;
```

Figure 25: .NET Volatile Half Fences

8 GPU Parallelization:

Lacency vs throughput

latency how long does it take to execute a single instruction / operation

throughput number of instructions / operations completed per second

High throughput: lots of pizzas per hour

compute & memory bound The performance of a function is defined by memory bandwidth, compute bandwidth and latency. Total time for a function is `memory access + compute time`. If the compute time takes longer we say the function is *compute bound*. If the memory time takes longer we say the function is *memory bound*.

When to use the GPU The GPU is good if you can run the same instruction on multiple data at the same time. Therefore, GPU parallelism is only then a good idea when the function / code is compute bound. If the problems is the memory access then it is not a good idea to use the GPU.

Arithmetic Intensity The arithmetic intensity describes the ration between computing and memory access. The higher, the better: efficient utilization of modern parallel processors.

$$\begin{aligned} t_c &> t_m \\ \frac{ops}{BW_c} &> \frac{bytes}{BW_m} \\ \frac{ops}{bytes} &> \frac{BW_C}{BW_m} \\ \text{Arithmetic intensity} &= \frac{BW_c}{BW_m} \end{aligned} \tag{2}$$

If your arithmetic intensity is low, then you may not want to use the GPU. Instead, you want to optimize the memory access.

Roof line model
The roof line model describes visually where your applications has performance issues.
In point O_1 we can not fully use the GPU because the memory is too slow. However, in O_2 the CPU is can not calculate faster and in this point it is a good idea to use GPU.

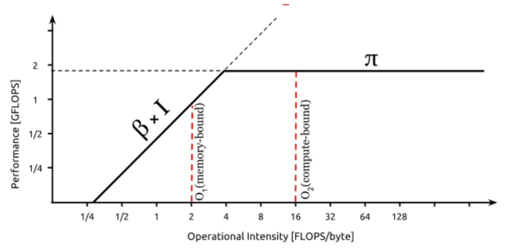
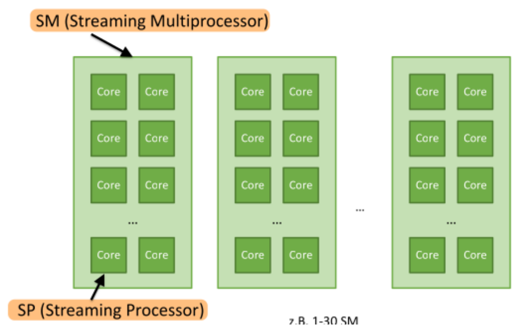


Figure 26: Roof line model

GPU Structure The big difference between the CPU and the GPU are the numbers of transistors for computing. The GPU has a lot more, but they are that good as the CPU ones.
A GPU consists of many **Streaming Multiprocessors (SM)**. And each SM consist of many **Streaming Processors (SP)** (Cores).



:
fig:gpu-structure

SIMD Single Instruction Multiple Data (SIMD) is performed on a Streaming Multiprocessor. Each core performs the exact same instructions but with different data.

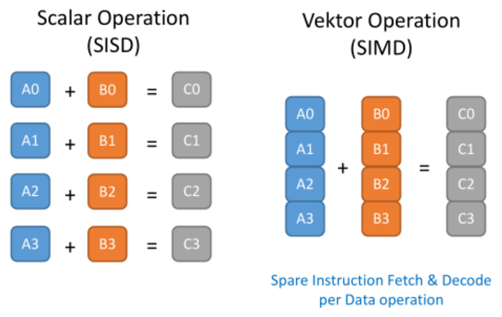


Figure 27: SIMD Vector parallelism

NUMA The NUMA model (Non-Uniform Memory Access) describes a model where two processors (CPU or GPU) do **NOT** share the memory. Therefore, one processor has to copy the data into other memory and back from there. The access time to the local memory (on-device) is much faster than the access to the remote one.

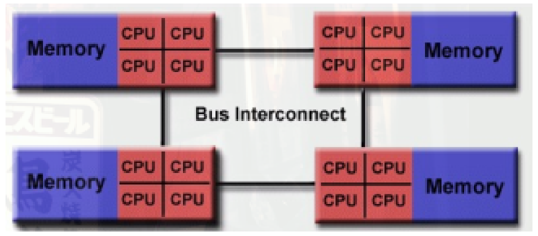


Figure 28: The NUMA model

Thermilogy

CUDA	OpenCL	Desc
Kernel	Kernel	Function to run on GP
Host Program	Host Program	Application which calls
Thread	Work Item	A single thread which r
Block	Work Group	A collection of threads
Grid	NDRange	A collection of blocks

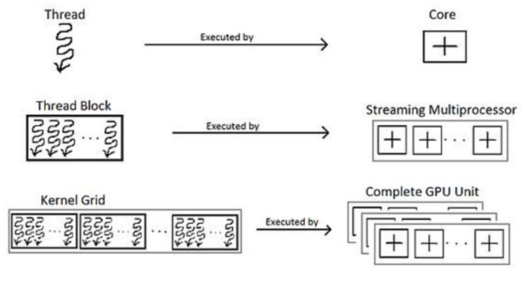


Figure 29: CUDA concepts on the GPU

Thread Hierarchy A Kernel is executed in a single thread. The single thread is executed simultaneously to other threads in a block. Many blocks are executed at the same time on a grid.
Internally, the threads are grouped into warps. Theses, are executed in a block.
Important: Each block can only execute a specific number of threads. And a grid can only execute a specific number of blocks.

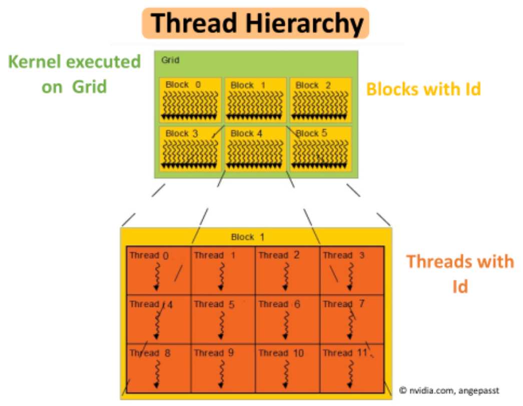


Figure 30: Thread Hierarchy on GPU

9 Performance Optimizations

Compilation Workflow The same applies to OpenCL (probably).
In CUDA you separate the device code from the host code. The device code is compiled into an assembly form (PTX code). The host codes loads and launches the compiled kernel (PTX code). The PTX code can be loaded by an application at runtime and is then **just-in-time compiled**. This has the benefits that you can benefit from the improvements of the compiler in the newer driver versions.

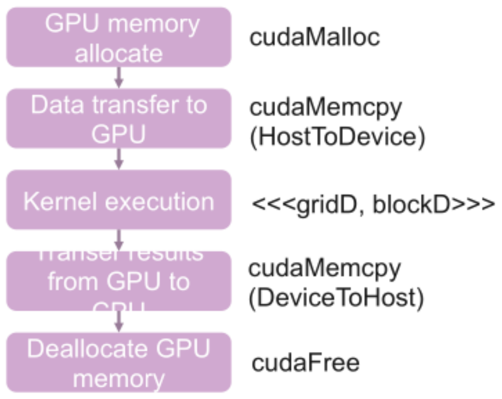


Figure 31: The CUDA execution

Ceiling Integer

$$\text{blocks} = \frac{N + (m-1)}{m} \tag{3}$$

$$\text{blocks} = \frac{N + 1023}{1024} \tag{4}$$

Warp A block is internally in **Warps** allocated. One Warp consists of 32 Threads. Each of these 32 threads executeds the same instructions (same branch in same kernel).

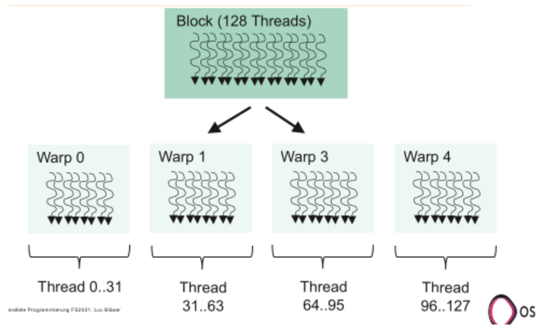
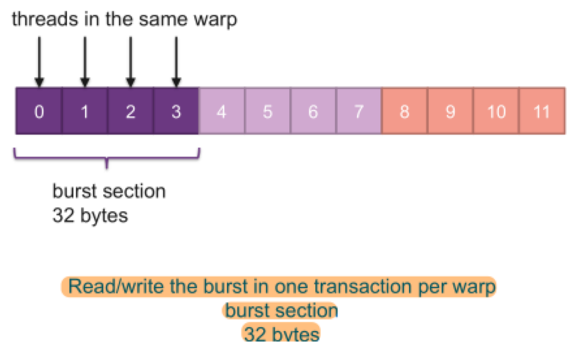


Figure 32: Block in Warp allocated

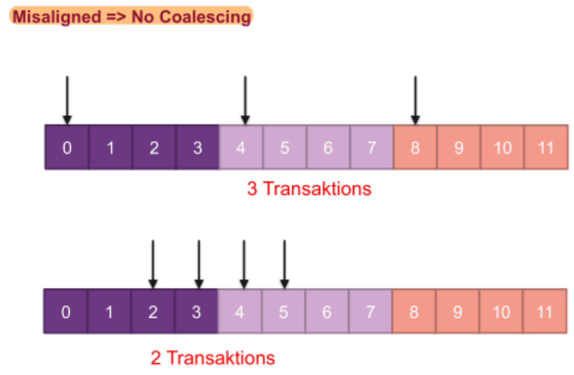
Memory Coalescing The pattern how a thread access the memory is critical. All threads in a warp should access the same ares (burst). Then a combined access is possible. Otherwise, you have to do expensive individual accesses.
If you have a an access on memory use always the `threadIdx.x/y/z` to align the memory access.

```
data[(Expression without threadIdx.x) +
    threadIdx.x]
```

Listing 3: Align memory access (coalescing)



(a) Memory Coalescing



(b) Non-Memory Coalescing

CUDA Memory Model All threads have access to the same global memory. Each thread block has shared memory, which is visible to all threads of the block. The shared memory is much faster because it is on-chip. The variables in the kernel are normally stored in the regiester (fastest access). Each thread has private local memory which resides in device memory (slow). Only when all register are used the GPU uses the local memory.

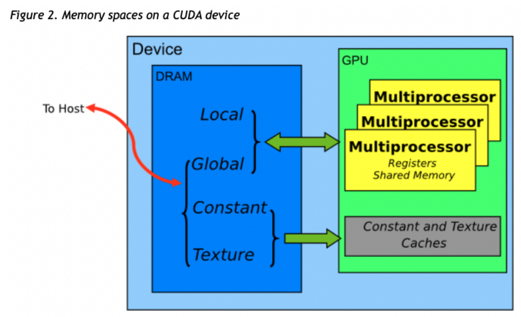


Figure 34: CUDA memory model

Shared Memory In matrix multiplication the same row / column is used multiple times. If the matrix are too big then can not load the whole row / column into memory. Therefore, the multiplication has to split up.

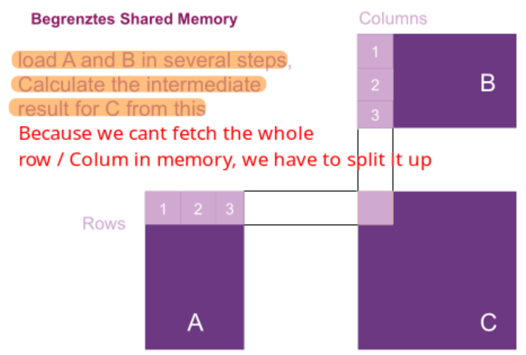

Begrenztes Shared Memory
load A and B in several steps,
Calculate the intermediate
result for C from this
Because we cant fetch the whole
row / Colum in memory, we have to split it up
Columns
1
2
3
B
Rows
1 2 3
A
C

Figure 35: Tiling Matrix Multiplication

```
float sum = 0.0;
for (int tile = 0; tile < nofTiles; tile
    ++) {
  // read tile from a and b in shared
      memory
  // each thread reads an element from
      each tile
  __syncthreads();
  // Multiply row of A–Tile by
  // Column of B–Tile from shared memory
  sum += partialProduct;
  __syncthreads();
}
C[row * M + col] = sum;
```

10 Cluster Parallelization

11 End