1 Multi-Threading Basics

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!

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*.

Thread States

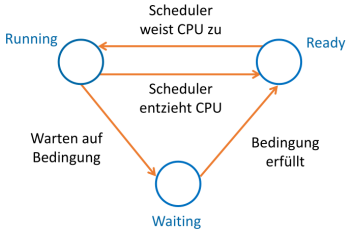

Figure 1: Thread States

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.

2 Monitor

Monitor The monitor is a synchronisation mechanism which use the *Wait & Signal / Signal & Continue* 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 (`notify()` / `notifyAll()`)
   - condition not satisfied: leaves monitor and goes to the right and wake up other threads (Wait on signal)
3. Go to 1.

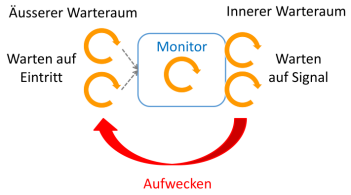

Figure 2: A monitor

Single Notify 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

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

3 Synchronization Mechanism

Semaphore

```
public class BoundedBufferSemaphore<T> {
    private Queue<T> queue = new
        LinkedList<>();

    public BoundedBufferSemaphore(int
        capacity) {
        upperLimit = new Semaphore(
            capacity, true);
    }

    public void put(T element) throws
        InterruptedException {
        upperLimit.acquire(); // counter
            --
        mutex.acquire(); queue.add(
            element); mutex.release();
        lowerLimit.release(); //counter
            ++
    }
}
```

Lock & Conditions

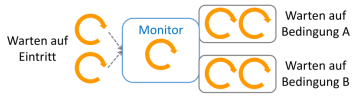

Figure 3: Lock Conditions

```
public class WarehouseWithLockCondition
    {
    public WarehouseWithLockCondition(
        int capacity, boolean fair) {
        lock = new ReentrantLock(fair);
        nonEmpty = lock.newCondition();
        nonFull = lock.newCondition();
    }
    @Override
    public void put(int amount) throws
        InterruptedException {
        lock.lock();
        try {
            nonFull.await();
            nonEmpty.signalAll();
        } finally { lock.unlock(); }
    }
}
```

Read-Write Locks

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

CountdownLatch

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

Cyclic Barrier

```
var gameRound = new CyclicBarrier(5);
/* 5 different players / threads */
while (true) gameRound.await();
```

Rendez-Vous

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

Monitor in .NET

```
class BankAccount {
    private decimal balance;
    private object syncObject = new();
    public void Deposite(decimal amount)
        {
        lock (syncObject) {
            balance += amaount;
            Monitor.PulseAll(syncObject)
                ;
            //Monitor.Wait(syncObject);
        }
    }
}
```

4 Threats

General Race Condition, Deadlock, Starvation

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

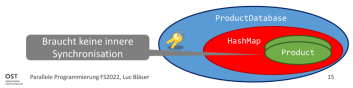

Figure 4: Example of object confinement

5 Thread Pool

Thread 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.

Work stealing 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.

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 indepentend (only if not max threads is set).

Number of Threads 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:

ForkJoinPool

```
var threadPool = new ForkJoinPool();
Future<Integer> future = threadPool.
    submit(() -> {
        int value = 1;
        return value;
});
Integer i = future.get();

class MyTask extends RecursiveTask<
    Integer> {
    boolean finished = false;
    @Override
    protected Integer compute() {
        if (finished) return 1;
        var left = new MyTask();
        var right = new MyTask();
        left.fork();
        right.fork();
        return left.join() + right.join
            ();
    }
}
var n = threadPool.invoke(new MyTask());
```

.NET

```
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]));
```

6 Async

Types 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 C# Exception in Fire & Forget are ignored. To handle exception you have to wait synchrony for finishing the task.

```
Task.Run(task1).
    ContinueWith(task2).
    ContinueWith(task3).Wait();
Task.WhenAll(task1, task2).
    ContinueWith(continuation).Wait();
Task.WhenAny(task1, task2).
    ContinueWith(continuation).Wait();
```

CompleatableFuture Java

```
var all = CompletableFuture.<Void>
      completedFuture(null);
for (int i = 0; i < LINKS.length; i++) {
    var link = LINKS[i];
    var future = downloader.
          asyncDownloadUrl(link).
          thenAcceptAsync(result -> { /*
          Stuff*/ });
    all = CompletableFuture.allOf(all,
          future);
}
all.thenAcceptAsync(voids -> { /*Stuff*/
      }).join();
```

7 Memory Modell

Atomicity Single reads / writes are atomic for:

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

Java: `AtomicBoolean` / C# `Interlocked`

Visibility Java guaranties the following visibility:

- changes before release are visible at acquire
- changes until write are visible at read
- 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!)

Ordering 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

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

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

In Java is a Full Fence (no reordering).

```
Thread.MemoryBarrier(); // Full Fence
```

8 GPU

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

throughput number of instructions / operations completed per second

Arithmetic Intensity

$$t_c > t_m$$
$$\frac{ops}{BW_c} > \frac{bytes}{BW_m}$$
$$\frac{ops}{bytes} > \frac{BW_C}{BW_m} \quad (1)$$
$$\text{Arithmetic intensity} = \frac{BW_c}{BW_m}$$

Thermilogy

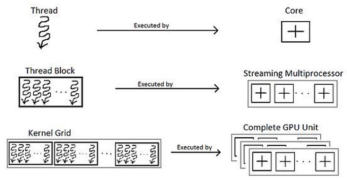

Figure 5: CUDA concepts on the GPU

Classification

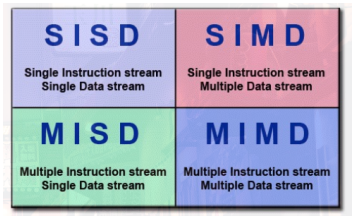

Figure 6: Flynn's Classical Taxonomy

Launch Kernel

```
__global__
void VectorAddKernel(float *a, float *b,
      float *c, int size) {
  int i = blockIdx.x * blockDim.x +
      threadIdx.x;
  if (i < size) {
    c[i] = a[i] + b[i];
  }
}
int blockSize = 1024;
int gridSize = (numElements + blockSize
    - 1) / blockSize;
VectorAddKernel<<<gridSize, blockSize
    >>>(d_a, d_b, d_c, numElements);
```

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 (What is a warp?). 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.

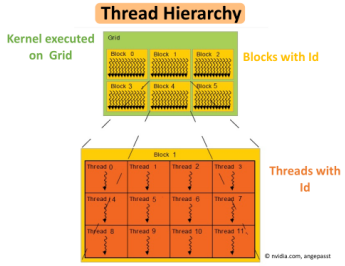

Figure 7: Thread Hierarchy on GPU

CUDA Memroy 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.

Memory Coalescing

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

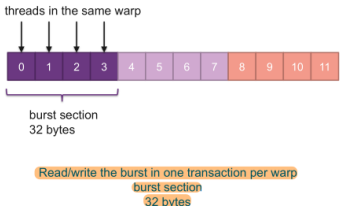

Figure 8: Memory Coalescing

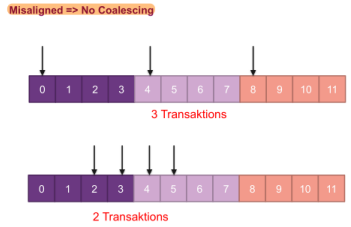

Figure 9: No Memory Coalescing

Branching / Divergenz The Streaming Multiprocessor executes the instruction of one branch per warp. All the other threads have to wait. Then the next branch is executed. Some threads have to wait again. This leads to performance problem. Single `if` is not a problem

9 MPI

Hybrid Memory Architecture Most modern supercomputers use a hybrid memory architecture. All processors in a machine can share the memory (UMA). The memory from a remote machine can be requested programmatically (NUMA).

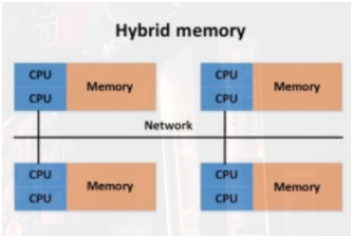

Figure 10: Hybrid Memory Architecture

SPMD / MPMD

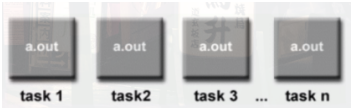

Figure 11: Single Program Multiple Data

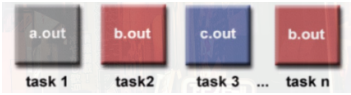

Figure 12: Multiple Program Multiple Data

Communication Modes Scatter, Gather, Broadcast

MPI Message When one process transfers data to another process the following data are sent:

- ID of the sender
- ID of the receiver
- Data type to be sent (int, float, char, ...)
- number of data items
- the data itself
- a message type identifier

Broadcast The root note sends the data to all notes. As soon as a node receives the data it sends the data to other notes. Because of this, `MPI_Bcast` is faster than `MPI_Send`.

Structure

```
int main(int argc, char* argv[]) {
  MPI_Init(&argc, &argv);
  int rank;
  int size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  if (rank == ROOT_RANK) {
    MPI_Gather(/**/);
  } else {
    MPI_Gather(/**/);
  }
  MPI_Finalize();
  return 0;
}
```

OpenMP OpenMP is an API for writing multithreaded code in C, CPP. OpenMP uses `#pragma` to insert instructions.

```
#include <stdio.h>
#include <omp.h>

int main(void)
{
#pragma omp parallel
  printf("Hello, world.\n");
  return 0;
}
```

10 Laws

Amdahls's law

$$\text{SpeedUp} = \frac{1}{s + \frac{p}{N}} \quad (2)$$

Gutafson's law

s serial part
p parallel part
N number of processes

$$\text{SpeedUp} = s + p \cdot N = s + (1 - s) \cdot N \quad (3)$$