Parallele Programmierung

Bsc. Informatik ETH Zürich

Laurin Seeholzer

May 31, 2026

Contents

1 Threads and Processes

1.1 Processes

Definition 1.1. A process is a program executing inside an OS (operating-system). Processes share CPU but have seperate memory space.

Definition 1.2. The **context** of a process is the set of utilities it uses and or operates on. That is:

- **Hardware context:** Instruction counter, Registers, Execution State, Flags ...
- **Memory context:** Virtual Memory, Heap-Space, Stack ...
- **OS-level context:** Process ID, open files ...

The lifecylce of a process is managed by the OS and can be described by the following states:

- **CREATED:** The process is loaded from the disc
- **WAITING:** The process is waiting in the main-memory for execution.
- **RUNNING:** The process is executed
- **BLOCKED:** The process is waiting for access to I/O devices
- **TERMINATED:** The process has finished its execution

Definition 1.3. Context Switch: The process of saving the context of a process and loading the context of another process. This happens for example when a process recives the RUNNING state and another process gets BLOCKED.

All processes are managed by the OS, which schedules, synchronizes, starts and terminates processes, allows comminication and manages ressources. The OS uses so called Process control blocks (PCB).

Definition 1.4. A **process-control-block (PSB)** is a table-entry storing relevant process information such as Program counter, Registers, Adress Space, Open files ...

1.2 Parallelism

Definition 1.5. Parallelism is when multiple processes execute simultaneously on diffrent CPU-Cores.

Definition 1.6. Concurrency is when multiple processes execute interleaved on a single CPU-Core.

Remark 1.7. Observe that parallelism implies concurrency, but concurrency does not imply parallelism.

1.3 Multithreading

Definition 1.8. A thread is an independent sequence of exection running in the same OS-process.

Remark 1.9. While **processes can only communicate through main-memory** and have diffrent memory spaces, **threads share the same adress space**. This leads to faster and more efficient context-switches as adressspace and PCB loading is not necessary.

There are diffrent levels of threads:

- **User-Level Threads:** Threads managed by the application using a thread library, ex. Java-Threads managed by JVM
- **Kernel-Level Threads:** Threads managed by to OS, which can schedule them on diffrent cores.
- **CPU-Level Threads:** Threads managed by the CPU, which can schedule them on the physical cores. Ex. A CPU might have 4 physical cores, which apear to the OS as 8 logical cores.

There are two ways to create a thread in java. We can directly extend the Thread class and implement our run method in the Thread.

Code Example 1.10

```
class MyThread extends Thread {
    ...
    public void run() {
        ...
    }
}
MyThread thread = new MyThread();
thread.start(); // calls MyThread.run()
```

Or we can use the Runnable Interface to create a runnable object which we pass to the Thread.

Code Example 1.11

```
class MyRunnable implements Runnable {
    ...
    public void run() {
        ...
    }
}
MyRunnable runnable = new MyRunnable();
Thread thread = new Thread(runnable);
thread.start(); // calls MyRunnable.run()
```

Remark 1.12. The .run() method is never called directly, because then the method is executed by the current thread (e.g. the main-thread). To span a new thread and run it we have to use thread.start().

In its lifecylce a thread can have the follwoing states:

- **NEW:** The thread has been created but not yet started.
- **RUNNABLE:** The thread is ready to run and waiting for the OS to schedule it.
- **RUNNING:** The thread is currently executing.
- **BLOCKED:** The thread is waiting for access to I/O devices
- **WAITING:** The thread is waiting to be notified
- **TIMED WAITING:** The thread is waiting for a specified amount of time before it continues.
- **TERMINATED:** The thread has finished its execution

Remark 1.13. Every Thread has an **ID, Name, Priority and State** which might be usfull for certain tasks.

To wait for a thread to finish, we can use the .join() method on the thread we want to wait on. This lets the main-thread sleep until the joined thread terminates and wakes it up again.

Code Example 1.14

```
Thread t = new MyThread();
t.start(); //start t
t.join(); //sleep until t is finished
continueWork(); // this is only called after t finishes
```

Remark 1.15. Using the **.join() method might not always be the best option**, as it produceses a **significant overhead**. If the execution time of a thread is short, it might be better to let the main-thread busy wait in a while loop.

Important 1.16. If a worker-thread throws an exception the main-thread will be woken up, but it **will not know of the error**, which might lead to false results. To make sure we don't produce false results we have to **set an UncaughtExeption-Handler** using a try-catch block when joining threads. The same holds for **threads that run indefinetly** we need to make sure to **have a timer task** that interupts threads or manually interupt them.

Code Example 1.17

```
Thread t = new MyThread(); //worker thread
Thread timer = new MyTimerThread(); //timer thread

t.start();
timer.start(t, 1000);

try {
    t.join();
} catch (UncaughtExeptionHandler) {
    System.out.println("Error in thread t"); //handle the error
}
continueWork();

MyTimerThread extends Thread {
    run(Thread thread, int time) {
        timer.sleep(time);
        thread.interupt(); //interrupt indefinetly running thread
    }
}
```

Important 1.18. Thread scheduling is not deterministic. This means that we can't rely on the order (interleaving) in which threads are executed.

Important 1.19. The program execution is finished if all non-deamon threads have terminated, not necessarily when the main method or main thread terminates.

2 Synchronization

Example 2.1. A famous example of thread-safety issues is the counter. When a counter c is incremented using c++ or ++c the value of c first gets read, then incremented and then stored. The counter fails if one thread stores his result between the time another reads and stores again.

Definition 2.2. A **Bad Interleaving** is an interleaving of operations from multiple threads that leads to an incorrect program state.

Definition 2.3. A **Critical Section** is a codeblock that might lead to bad interleavings. To ensure program correctness we need the enforce mutual exclusion on them.

Definition 2.4. Thread safty refers to nothing bad happeing in all possible interleavings and thus implies program safety.

Definition 2.5. Thread Liveness refers to something good happening eventually, which implies the correctness of a program.

2.1 Race Conditions

Definition 2.6. A **Race condition** occurs in concurent programming when the correctness of the system depends on the specific interleaving or ordering of execution of operations.

Definition 2.7. A **Low-level race condition** is a race condition that occurs because of insufficient synchronization of memory access (Data Races).

Definition 2.8. A **High-level race condition** is a race condition that occurs because of a wrong decomposition/algorithm. The program might ensure mutual exclusion on the shared resources but the program might still produce wrong results due to **bad interleavings**.

Generally we want one of three things when it comes to shared ressources:

- **Thread local (Isolated Mutability):** We dont use the ressource of other threads.
- **Read only (Immutable):** We only read and do not change the ressource

- **Mutual exclusion (Synchronization):** We use the ressource of other threads but we ensure that only one thread accesses it at a time.

2.2 Locks and Synchronized

Every Java-Object has an **intrinsic lock or lock monitor**. These locks are unique and can be optaind by threads which allows us to lock a certain codeblock. If a thread wants to access such a codeblock and execute the code within it first has to optain the lock, while it has the lock, no other thread can accuire it.

Code Example 2.9

```
synchronized(lockObject) {
    i++;
}
```

This ensures mutual exlusion on the synchronized codeblock. It is also possible to use synchronized as a keyword in a function to sync. the whole function.

> **Remark 2.10.** By default synchronized blocks use "this" as the lock-object and unlock when an exception occures.

> **Remark 2.11.** Java-Locks are **reentrant**, meaning they can be locked multiple times by the same thread recursively (it will then have to release the lock as many times as it acquired it).

For incrementing and decrementing integers there is a special DataType AtomicInteger which has built in synchronization as incrementing/decrementing is done in a single step.

Code Example 2.12

```
AtomicInteger counter = new AtomicInteger(0);
counter.incrementAndGet();
```

2.3 Wait, notify, notifyAll

We can use wait() on a lock-object to tell the current thread to sleep until further notice if the lock is not available. This allows use to save ressources.

Code Example 2.13

```
synchronized(lockObject) {
    while(!condition) {
        lockObject.wait();
    }
    // do work
}
```

To wake one or multiple threads and tell them that the lock is now available we can use notify() or notifyAll() on the lock-object to wake up an aribtrary waiting thread or all of them.

Important 2.14. Always **wait inside a while loop**, as this ensures checking the condition after beeing notified.

Code Example 2.15

```
synchronized(lockObject) {
    while (!condition) {
        lockObject.wait();
    }
    //do work
    notifyAll();
}
```

2.4 Lock API

Example 2.16. The problem with the below code is that it is **not thread-safe**. If two threads try to withdraw money at the same time, they might both read the same balance and then both write to it, resulting in a wrong or even negative final balance. We need mutual exclusion in that critical section.

Code Example 2.17

```
class BankAccount {
    private int balance;

    int balance() {return balance;}

    void setBalance (in x) {balance = x;}

    void withdraw(int amount) {
        int b = balance();
        if (b less than amount) {
            throw new Exception("Not enough money");
        }
        setBalance(b-amount);
    }
}
```

Instead of using built-in synchronized blocks we can also use locks from the java.util.concurrenct.locks package.

Code Example 2.18

```
class BankAccount {
    private Lock lock = new ReentrantLock();
    ...
    void withdraw(int amount) {
        lock.lock();
```

```
6           try {
7               ...
8           } finally {
9               lock.unlock();
10          }
11      }
12  }
```

Important 2.19. Always use a try-finally block to ensure the lock is released even if an error occures.

Remark 2.20. There is also a **lock.tryLock(100, TimeUnit.MILLISECONDS)** method which tries to accuire the lock only for a given amount of time. This is useful for avoiding deadlocks (ex. when two threads have each one lock and wait on the other to get released)

Aspect	synchronized	Lock API
Lock handling	Automatic (JVM managed)	Manual (lock() / unlock())
Lock release	Automatic	Must be done explicitly (finally!)
Scope	Method or block scoped	Flexible (can span multiple methods)
Waiting behavior	Thread blocks indefinitely	tryLock() avoids blocking
Interruptibility	Not interruptible while waiting	lockInterruptibly() supports interrupt
Flexibility	Low	High
Complexity	Simple, less error-prone	More complex, error-prone
Deadlock risk	Lower	Higher (if unlock forgotten)
Typical use	Simple synchronization	Advanced concurrency control

2.5 Guidelines for Concurrent programming

Important 2.21. Thread-Locality: Minimize the sharing of mutable state between threads. Only share ressources if needed.

Important 2.22. Immutability: Use immutable data structures whenever possible. They are inherently thread-safe.

Important 2.23. Consitent locking: If you need to share mutable state, use locks to ensure mutual exclusion.

Definition 2.24. Lock granularity refers to the amount of code that is proteted by a lock in ways:

- Size of synchronized blocks
- Number of objects a lock is used for

Theorem 2.25. Coarse-grained locking is easier to implement but leads to less concurrency and therfore performance loss. **Fine-grained locking** is harder to implement without creating bugs but leads to more concurrency and therfore performance gain.

Important 2.26. Start with coarse-grained (simpler) and move to fine-grained (performance) only if contention on the coarser locks becomes an issue.

Important 2.27. Do not do expensive computations or I/O in critical sections, but also don't introduce race conditions

Definition 2.28. Atomicity: An operation is atomic if it is performed as a single, indivisible unit. In other words, it either completes entirely or not at all, without any intermediate states being visible to other threads.

3 Optimization

Theorem 3.1. Moore's Law states that the number of transistors on a microchip doubles approximately every two years.

By moores law architects improved sequential execution every year. Sequential programs were becoming exponentialy faster. But in the early 2000's we began hitting walls:

- **Power-Consumption**: Expensive cooling limited further increase of clock-speed.
- **Memory-Wall**: CPU's became faster than memory and had to wait (stall) on load/store operations.

3.1 Caching

Faster CPU's does not mean faster execution. The CPU might compute almost instantly but the loading and storing of the exectution data/result forms a bottleneck, as this is one of the slowest operations (high latency).

Definition 3.2. Cache is a fast memory that preloads and stores frequently used data form the memory to reduce latency.

The usage of cache reduces stalls (times in which the CPU can not execute as the data is not yet present).

> **Remark 3.3.** Programmers can influence the cache usage by organizing their data in a cache-friendly way. That means increasing the locality of data access so that the cache can efficiently prefetch data.

3.2 Vectorization

With vectorized data we can use SIMD to exectute the same operation on multiple data points all at once.

Definition 3.4. SIMD (Single Instruction Multiple Data) is a type of parallel processing where the same instruction is applied to multiple data points simultaneously by using multiple ALUs inside the CPU. The same instruction gets broadcasted to multiple ALU which execute in parallel.

3.3 Instruction Level Parallelism (ILP)

Definition 3.5. Superscalar execution refers to out of order execution of independent instructions in parallel using multiple ALUs.

Definition 3.6. Speculative execution refers to the execution of instructions before the actual need for them is known.

3.4 Pipelining

Pipelining increases throughput by overlapping the execution of multiple instructions in diffrent stages.

Example 3.7. As soon as I am finished washing my laundry and start hanging it, my neighbour can already use the machine and start wahsing his clothes. My neighbour does not have to wait for me to wash, hang, dry, fold, store my clothes.

Definition 3.8. Throughput is the amout of work that can be done by a system in a given period of time. It can be bounded by

$$tp = \frac{1}{max(computationTime(stages))}$$

Definition 3.9. Latency is the time needed to perform a given computation. It can be bound by

$$L = numStages * max(computationTime(stages))$$

Definition 3.10. A pipeline is **balanced** if all stages take the same amount of time.

Balancing a pipeline can increase throughput.

Example 3.11. Assume washing is the stage that takes the longest. Having two small washing machines (one to prewash, one to rinse) instead of a large one can increase through put as now my neighbour can start prewashing as soon as I start rinsing and my neighbours neighbour can start prewahsing as soon as i am finished rinsing. This balances the pipeline.

3.5 Parallel Performance

Definition 3.12. Scalability describes how performance improves with additional resources (cores, memory, etc.).

Definition 3.13. Speedup is the ratio of the execution time of the sequential algorithm to the execution time of the parallel algorithm.

$$S_p = \frac{T_1}{T_p}$$

Definition 3.14. Efficiency is the ratio of speedup to the number of processors.

$$E_p = \frac{S_p}{p}$$

Remark 3.15. Speedup depends on the definition of T_1 (**Baseline**), is it the execution time of the optimized sequential algorithm or the unoptimized one?

Theorem 3.16. Amdahl's Law sets a limit on speedup, as the sequential part of the algorithm can not be parallelized.

$$S(p) = \frac{1}{f + \frac{1-f}{p}} \implies {}_{p\to\infty} S(p) \leq \frac{1}{f}$$

Theorem 3.17. Gustafson's Law proposes an optimistic view compared to Amdahl. It shows that we might not be able to shrink execution time but we can perform more operations in that same execution time.

$$S_p = f + P(1 - f)$$

4 Parallel Patterns

4.1 Divide and Conquer

Definition 4.1. Divide and Conquer refers to the strategy of splitting a problem into smaller subproblems, solving them recursively, and combining the results.

This concept can be used in many parallel algorithms. Until now we have used manual forking .start() and joining .join() of Threads. We can continue using manual fork/join with DC, but this can lead to several Problems.

- Poor reusability of code
- We need to commit to a number of threads a priori and can not dynamically adjust.
- Subproblems are not automatically assigned to a thread.

Remark 4.2. There are some fixes that can improve manual fork/join such as **sequential-cutoffs** or running one subproblem on the thread itself (**invoke/compute**), instad of creating two new threads.

4.1.1 Executor Service

The Executor Service framework provides the way to schedule and run tasks on a thread-pool automatically.

Definition 4.3. Runnable: A task without a return value. Passed to execute() (fire-and-forget) or submit() (returns an empty future).

Definition 4.4. Callable: A task that returns a result of given type T and can throw checked exceptions. Passed to submit(Callable) and returns a Future of type T.

Remark 4.5. There are diffrent **Pool Types**: Executors.newFixedThreadPool(n) keeps n worker threads alive. newCachedThreadPool() scales up and down dynamically depending on load.

Code Example 4.6

```
1 ExecutorService executor = Executors.newFixedThreadPool(4);
2
3 class MyRunnable implements Runnable {
4     public void run() {...}
5 }
6
7 class MyCallable implements Callable<Integer> {
```

```
8       public Integer call() throws Exception {...}
9   }
10
11  executor.execute(new MyRunnable());
12
13  Future<Integer> future = executor.submit(new MyCallable());
14  int res = future.get(); // Blocks until result is returned
15
16  executor.shutdown();
```

Pros and Cons of Executor service:

- Not suited for: Recursive subproblems that depend on each other and have a deep fork-tree. As the service will eventually run out of threads.
- Suited for: flat structures that can run independently in parallel.

4.1.2 Fork-Join

The fork-join framework is built for divide and conquer parallelism. It manages a pool of threads using a **work-stealing strategy**: each thread maintains a local task-queue, and when finished with its own tasks, steals work from others to minimize idle time.

Definition 4.7. RecursiveTask: A task that returns a result of type given Type T (equivalent to Callable).

Definition 4.8. RecursiveAction: A task that does not return a result (equivalent to Runnable).

Key Methods of the Fork-Join Framework:

- **fork()**: Asynchronously pushes the task to the local thread's queue.
- **join()**: Blocks until the result is ready. While waiting, the thread does not truly idle but uses work-stealing to execute other tasks.
- **compute()**: The main logic method that must be overridden.

Code Example 4.9

```
1   class MyTask extends RecursiveTask<Integer> {
2       int start, end;
3       MyTask(int start, int end) { this.start = start; this.end = end
            ; }
4
5       protected Integer compute() {
6           if (end - start <= SEQUENTIAL_CUTOFF) {
7               return baseCase();
8           } else {
9               int mid = (start + end) / 2;
10              MyTask left = new MyTask(start, mid);
11              MyTask right = new MyTask(mid, end);
```

```
12
13              left.fork(); // Async execute left half
14              int rightRes = right.compute(); // Optimize: caller
                    thread computes right half
15              int leftRes = left.join();      // Wait for left half
16
17              return leftRes + rightRes;
18          }
19      }
20  }
```

Important 4.10. Make sure that every task has enough work to do (aprox. 100-10'000 ops.), else the overhead will be larger than the parallel speedup gain.

4.1.3 Task graphs

The execution of parallel programs using Fork-Join can be represented as a Task-Graph (DAG).

Definition 4.11. T_1 is the execution time on a sequential machine and can be visualized as the sum of all task durations in our DAG.

Definition 4.12. T_∞ is the execution time assuming unlimited hardware. This is called the **span** and can be visualized as the sum of the execution times of the longest dependent path in our DAG.

Remark 4.13. The maximum speedup is limited by $\frac{T_1}{T_\infty}(AmdahlsLaw)$.

It is the programmers responsibility to find the optimal decomposition/alogrithm for the problem to ensure a minimal span (a balanced tree). **Fork-join will guarantee optimal scheduling of the tasks with minimal idletime.**

4.2 Parallel Patterns

Definition 4.14. A **reduction** is a parallel operation that produces a single answer from a collection via an associative operator using for example DC. (The order of operations does not matter).

Example 4.15. Finding the Max, Min, Sum or Product over all entries of an array.

Definition 4.16. A **map** applies a function to each elements of a collection producing an output of the same size.

Example 4.17. Vector addition can be done using a map pattern. We split the vectors into dim(v)/2 recursively until we have dim(v)=1 then sum the vectors and write the result directly into the result vector.

Remark 4.18. Maps and Reductions have (when parallelized) an optimal span of $O(\log n)$.

Definition 4.19. A **stencil** is an extension of the map pattern that applies a function to each element of a collection producing an output of the same size. The function uses the element itself and its neighbours.

Example 4.20. A smoothing kernel (ex. 3x3) that calculates the average value is sled over an image to smooth/blur the image.

Remark 4.21. Note that arrays and balanced trees are usually parallelizable while linked lists are not even if the per element operation is parallelizable.

4.3 Parallel Algorithms

4.3.1 Prefix-Sum

Remark 4.22. Prefix-Sum (calculating input[1] + input[2] + ... + input[n]) seams to be unparallisable. But infact it is!

Algorithm 4.23

```
upTask(); //divide problem and build tree with sums of subproblems
downTask(); //fill in the correction values (left_sum +
    parent_correction)
```

Theorem 4.24. The span of the prefix sum algorithm is $O(\log n)$. This leads to a parallelism of $\frac{n}{\log n}$.

4.3.2 Pack

Another application of this problem is the **Pack** algorithm which removes all elements from an array that satisfy a certain condition and packs them into a seperate output.

Algorithm 4.25

```
applyMap(); //applies a map to the input
//results in a binary vector [1, 0, 1, 1, 0, ...]

parallelPrefixSum(); //prefix sum of the binary vector
//results in [1, 1, 2, 3, 3, ...]

applyPackMap(); //applies map to the result of the prefix sum
//results in [0, 2, 3, ...] (indices of output elements)
```

4.3.3 Quicksort

We can reduce the limiting factor of the problem division from a span of $O(n)$ to a span of $O(\log n)$ using the pack-pattern we have seen before.

Algorithm 4.26

```
applyMaps(); //apply maps for greater and smaller than pivot
parallelPrefixSum(); //num. of smaler/greater elems. come bofore
applyMap(); //index = numLessThan || numTotalLessThan +
    numGreaterThan
```

We can do this by using two pack-patterns to partition the data. This leads to a partitioning of $O(\log n)$ levels times $O(\log n)$ for prefixSum hence $O(\log^2 n)$ total span. This means a parallelism of $\frac{n \log n}{\log^2 n} = \frac{n}{\log n}$.

5 Memory Models

5.1 Memory Reordering and Hierarchy

Remark 5.1. In a concurrent environment, programs often fail even if they appear logically sound in a sequential context. This is primarily due to memory reordering.

Definition 5.2. Memory Reordering: Compilers and hardware are permitted to change the order of memory operations as long as the semantics of a **sequentially executed program** remain unaffected.

Remark 5.3. Memory reordering offers significant performance benefits through optimizations like dead code elimination, register hoisting, and locality optimization, but it introduces errors when multi-threaded consistency is assumed.

Code Example 5.4

```
int x;
// Thread A: Waiting for a signal
void wait() {
    x = 1;
    while (x == 1);
}
// Thread B: Providing the signal
void arrive() {
    x = 2;
}
```

Consider how the **wait()** function might be compiled: **Naive Assembly (Standard):** The value of **x** is re-read from memory in every iteration.

Code Example 5.5

```
movl 0x1, x
test:
mov x, %eax    // Load x from memory
cmp 0x1, %eax // Compare x to 1
je test        // Jump back if equal
```

Optimized Assembly (The "Failure" Case): The compiler assumes **x** cannot change within the loop (since Thread A doesn't modify it there) and hoists the value into a register once or replaces the check with an infinite jump.

Code Example 5.6

```
movl 0x1, x
test:
jmp test       // Infinite loop: Thread A never sees Thread B's
    update!
```

Remark 5.7. Because L1 caches and registers are **private to each core**, a **store** operation to memory on Core 1 might be buffered locally and only become visible to Core 2 at a later, unpredictable time.

Remark 5.8. Use **volatile** in Java to prevent memory reordering by updating the local cache/buffer every time the variable changes.

Important 5.9. Volatile arrays only guarantee that the array reference is volatile, not the elements.

Remark 5.10. Almost all common architectures allow the reordering of **Stores after Loads** to maximize performance, unless explicitly prevented by memory barriers.

5.2 The Java Memory Model (JMM)

Definition 5.11. A **Memory Model** is a contract between the programmer and the system. It provides the minimal guarantees for memory operation effects while allowing hardware/compiler optimizations.

Remark 5.12. Java has synchronization primitives like **synchronized** and **volatile** that provide stronger memory guarantees than basic reads/writes.

Definition 5.13. Program Order (PO): A total order of actions within a **single** thread as defined by the execution trace. Crucially, PO does **not** provide a global ordering guarantee across threads.

Definition 5.14. Synchronization Actions (SA): Actions like reading/writing **volatile** variables or acquiring/releasing locks.

Remark 5.15. Synchronization actions establish HB dependencies between actions in different threads.

- Write to volatile in thread $A \rightarrow$ Read from volatile in thread B

- Lock release in thread $A \rightarrow$ Lock acquire in thread B

Definition 5.16. Synchronization Order (SO): A total order of all SA. All threads see these actions in the same global order.

Definition 5.17. Happens-Before (HB): The transitive closure of PO and **Synchronizes-With** (SW). If action A happens-before B, B is guaranteed to see the results of A.

Remark 5.18. HB-Consistency: The JMM explicitly allows "Data Races" (when two threads access the same variable without HB-ordering and at least one is a write). In an **HB-consistent execution**, a read operation is allowed to see either the temporally last write ordered before it via the HB-relation, or any other write that is not strictly ordered by HB. This is why unsynchronized reads can observe stale or surprising values.

6 Locks

6.1 Hardware Support for Parallelism

Remark 6.1. Software algorithms such as Bakery or Filter locks have **linear space complexity** $O(n)$, making them impractical for large numbers of processes. Hardware support is needed to implement locks efficiently.

Remark 6.2. Moder architectures offer **atomic instructions** that allow for operations to happen in a single atomic step. This allows mutex-implementation with **constant space complexity** $O(1)$ and can be used for lock-free programming.

6.1.1 Typical Hardware Instructions

Definition 6.3. Compare and Swap (CAS): Atomically compares the value of a memory location with a given value. If they are equal, it updates the memory location with a new value. Otherwise, it does nothing.

Definition 6.4. Test and Set (TAS): Atomically reads the value of a memory location and sets it to 1.

Definition 6.5. Load Linked / Store Conditional (LL/SC): LL reads the value of a memory location and sets a "link" to it. SC writes a new value to the memory location only if the link is still valid (i.e., no other process has written to that location since the LL).

Remark 6.6. Java offers these funcitonalities via the classes **AtomicInteger**, **AtomicLong**, etc.

6.1.2 Locks using atomic operations

Code Example 6.7

```
1  //TAS LOCK IN Java
2  public class TASLock implements Lock {
3      AtomicBoolean state = new AtomicBoolean(false);
4
5      public void lock() {
6          while (state.getAndSet(true)) {
7              //busy wait
8          }
9      }
10
```

```
11      public void unlock() {
12          state.set(false);
13      }
14  }
```

Remark 6.8. This lock consistently checks the state variable with an atomic write operation, which causes a lot of cache coherence traffic. To avoid this we can use TATAS lock (checking if the state is false before trying to acquire the lock).

Code Example 6.9

```
1  //TAS LOCK IN Java
2  public class TATASLock implements Lock {
3      AtomicBoolean state = new AtomicBoolean(false);
4
5      public void lock() {
6          do {
7              while (state.get());
8          } while (state.getAndSet(true));
9      }
10
11     public void unlock() {
12         state.set(false);
13     }
14 }
```

Remark 6.10. To further optimize this lock we can use a backoff strategy (e.g. exponential backoff) to reduce the cache coherence traffic. This means that if a process fails to acquire the lock, it will wait for a certain amount of time (ex. exponentially increasing) before trying again.

6.2 Locking algorithms

To implement a Lock (Mutex) correctly using only basic memory reads/writes, three conditions must be met:

1. **Mutual Exclusion:** No two processes can be in the critical section (CS) simultaneously.
2. **Freedom from Deadlock:** If processes try to enter the CS, at least one must eventually succeed.
3. **Freedom from Starvation:** Every process that attempts to enter must eventually succeed.

6.2.1 State Space Diagrams

Definition 6.11. A **State Space Diagram** is a formal tool used to track every possible configuration of a concurrent program. **States** are typically represented as a tuple, e.g., $[p, q, \text{wantp}, \text{wantq}]$, where p and q are the current program counters (line numbers) of the threads.

Remark 6.12. We can use state space diagrams to prove the correctness of lock implementations and understand the follwoing algorithms and idenfy failures

- **No Mutex:** Reachable state where both processes are in the CS.
- **Deadlock:** A state with no outgoing transitions where threads are stuck in the protocol.
- **Starvation:** An execution path where one thread never reaches the CS despite trying.

6.2.2 Attempts to mutual exclusion

Algorithm 6.13

```
volatile int flag[2] = {0, 0};

// Thread 0
while (flag[1] != 1) {}
flag[0] = 1;
// Critical section
flag[0] = 0;
```

Observe that this algorithm fails, as both thread might check the others flag and see that it is 0, and then both enter the critical section.

Algorithm 6.14

```
volatile int flag[2] = {0, 0};

// Thread 0
flag[0] = 1;
while (flag[1] != 1) {}
// Critical section
flag[0] = 0;
```

Observe that this algorithm also fails, as both thread might set their flag to 1 and then check the others flag and see that it is 1 and therefore deadlock.

Algorithm 6.15

```
volatile int turn = 1;

// Thread 0
while (turn == 1) {}
// Critical section
turn = 0;
```

Observe that this algorithm also fails as we do not guarantee that a thread resets the turn variable if it crahses or does not use the critical section. Furthermore this algorithm would ensure strict alternation (round-robin) between the threads.

6.2.3 Successful Algorithms

The first correct solution to the mutual exclusion problem for two processes. It combines the flag system with a **turn** variable to break ties. If both want to enter, the **turn** variable decides who must temporarily retract their interest.

Code Example 6.16

```
// Process P
wantp = true;
while (wantq) {
    if (turn == 2) {
        wantp = false;
        while (turn == 2); // wait for turn
        wantp = true;
    }
}
// Critical Section
turn = 2;
wantp = false;
```

Peterson's algorithm is a more concise version of Dekker's. Instead of retracting interest, a process simply "gives way" by setting a **victim** variable to itself.

Code Example 6.17

```
// Process P
wantp = true;
victim = 1; // P is process 1
while (wantq && victim == 1);
// Critical Section
wantp = false;
```

Theorem 6.18. Peterson's Algorithm provides Mutual Exclusion, Deadlock Freedom, and Starvation Freedom, and works elegantly using only atomic registers.

Definition 6.19. An event A is said to precede B denoted as $A \prec B$ if the execution interval of A does not overlap with the execution interval of B. If they do not overlap, then A and B are called concurrent.

Definition 6.20. An **atomic register** is a register that performs operations at a single point in time hence two operations on the same register always have diffrent effect times.

Proof. **Mutual Exclusion of Peterson's Algorithm:** By contradiction: assume concurrent CS_P and CS_Q. Assume wlog:

$$W_Q(\text{victim} = Q) \to W_P(\text{victim} = P)$$

From the program code:

- $W_P(\text{flag}[P] = \text{true}) \to W_P(\text{victim} = P) \to R_P(\text{flag}[Q]) \to R_P(\text{victim}) \to CS_P$
- $W_Q(\text{flag}[Q] = \text{true}) \to W_Q(\text{victim} = Q) \to R_Q(\text{flag}[P]) \to R_Q(\text{victim}) \to CS_Q$

Since $W_Q(\text{victim} = Q) \to W_P(\text{victim} = P)$, thread P must read P for victim. For thread P to enter its critical section (CS_P), the condition while condition must be false.

Because $victim == P$ is true, it follows that $R_P(\text{flag}[Q])$ must have read **false**. However, we have the sequence:

$$W_Q(\text{flag}[Q] = \text{true}) \to W_Q(\text{victim} = Q) \to W_P(\text{victim} = P) \to R_P(\text{flag}[Q])$$

By the transitivity of the precedence relation $\to$, $R_P(\text{flag}[Q])$ must read **true**. This is a contradiction. Therefore, mutual exclusion is satisfied. □

Proof. **Proof: Freedom from Starvation:** Assume wlogthat thread P runs forever in its lock loop, waiting until flag[Q] is flase or victim is not P.

There are three possibilities for thread Q:

1. Q **is stuck in its non-critical section:** Then flag[Q] = false, and P can continue. $\implies$ Contradiction.
2. Q **repeatedly enters and leaves its critical section:** Each time Q enters, it sets $victim = Q$. Since P does not change victim again, victim remains Q, allowing P to continue. $\implies$ Contradiction.
3. Q **is also stuck in its lock loop:** Q would be waiting until flag[P] is false or victim is not Q. However, the conditions victim == P and victim == Q cannot hold at the same time. $\implies$ Contradiction.

In all cases, we reach a contradiction, proving the algorithm is starvation-free. □

Remark 6.21. The Peterson's algorithm is correct, but it is not scalable. It is only proven for 2 processes. We can use the so called **Filter Lock** where every process has a level and in order to enter the critical section a process has to go trough all lower levels where on each level we use peterson's algorithm to eliminate one victim and let the others pass. This ensures mutual exclusion.

Remark 6.22. Another algorithm for mutual exclusion of n processes is the **Bakery Algorithm**. Every process has to take a numbered ticket with a number greater than all outstanding tickets and waits until it has the lowest number.

Theorem 6.23. If S is a [atomic] read/write system with at least two processes and S solves mutual exclusion with global progress [Deadlock freedom], then S must have at leas as many variables as processes.

Remark 6.24. But how can there exist computers with thousands to millions of threads? (Solution: Mutex is not implemented as we thought)

6.3 Locking Strategies

6.3.1 Fine-Grained Locking

Remark 6.25. Instead of locking everything coarse-grained (the whole data structure) we can lock individual parts of it. This is called **fine-grained locking** and allows for higher concurrency.

6.3.2 Read-Write Locks

Remark 6.26. With coarse- and fine-grained locking we have complete mutual exclusion on resources but what if we only need exclusive access for writing?

Definition 6.27. A **read-write lock** allow fair access to resources by allowing one writer (every N reads) or multiple readers at the same time.

Remark 6.28. To implement a RWL we need to keep track of the number of readers waiting, the number of writers waiting and the number of readers a writer has to wait until it can write (this ensures fairness).

6.3.3 Optimistic Locking

Remark 6.29. When for example traversing a list, finding a node with **hand-over-hand locking** requires locking all nodes on the way which consumes time and energy. But how high is it that two threads edit the same node at the same time?

Definition 6.30. Optimistic Locking: Instead of locking every node we could traverse the list without locking and only lock the node we are interested in. If we lock and notice that the node has been modified and our locks on (predecessor/successor) are nolonger valid we restart the operation.

Remark 6.31. Optimistic locking allows us for less overhead on locking but might require multiple traversals if there is high contention.

6.3.4 Lazy Locking

Definition 6.32. Lazy locking is a technique where data-elements are not immediately updated, but rather marked as deleted (or inserted) and only updated when a thread traverses the data structure again. This saves the overhead of multiple traversals.

6.3.5 Lock-Free Data Structures

Remark 6.33. Using atomic operations (mostly CAS) we can implement data structures without locks.

Code Example 6.34

```
class ConcurrentCounter {
    private AtomicLong value;

    public ConcurrentCounter() {
        value = new AtomicLong(0);
    }

    public void increment() {
        long expected;
        do {
            expected = value.get();
        } while (!value.compareAndSet(expected, expected + 1));
    }

    public long getValue() {
        return value.get();
    }
}
```

Remark 6.35. A lock free algorithm is always deadlock free

Definition 6.36. Lock-Free: System wide progress is guaranteed. This means that at least one process can always make progress.

Definition 6.37. Wait-Free: Any operation on the data structure will terminate in a bounded number of steps.

6.3.6 The ABA Problem

Remark 6.38. Assume we have a stack that we try to implement lock free and to save computation we dont fully garbage collect (reclaim memory) of popped elements. Instead we store them in a pool.

Example 6.39. Assume we pop element a from the stack and store it in the pool. Then we push some element b and then a again (which we got from the pool, hence has the same adress). A thread that read the last element before poping a and now checks again will not notice the change of the stack.

Important 6.40. The **ABA problem** is a common issue in lock-free programming where an atomic operation (like compare-and-swap) succeeds because the value has changed twice (e.g., A -¿ B -¿ A), making it appear unchanged to the operation. This can lead to incorrect program states.

Definition 6.41. We can use **Tagged Pointers** to significantly reduce the probability of the ABA problem by incrementing a **tag** along with the pointer in the CAS operation to detect up to 2^k changes where k is the number of bits used for the tag.

Definition 6.42. Hazard Pointers: A common technique to solve the ABA problem is to use hazard pointers. A hazard pointer is a pointer to a node that is currently being accessed by a thread. So when a thread reads the state of a datastructure it will store the address of the nodes it reads in its hazard pointer. Anyone trying to edit it will see the hazard pointer and not modify the nodes.

7 Beyond Locks

7.1 Deadlocks

Remark 7.1. Deadlock detection in systems is implemented by finding cycles in the dependency graph.

Remark 7.2. Deadlock avoidance amounts to techniques to ensure a cylce can never arise in the dependency graph.

- **imposing an ordering** on the resources and requiring processes to acquire resources in that order
- **two-phase locking protocol** with retry and release on failure

Code Example 7.3

```
1 //Programming trick to create a global ordering
2 class Resource {
3     private static final AtomicLong counter = new AtomicLong();
4     private final long index = counter.incrementAndGet();
5     ...
6 }
```

7.2 Semaphores

Definition 7.4. A **semaphore** is a synchronization primitive that acts as a thread-safe counter that can be used to control access to a shared resource by multiple processes.

Code Example 7.5

```
1  public class Semaphore {
2      private int n, S;
3      ...
4      acquire() {
5          wait until S > 0;
6          S--;
7      }
8
9      release() {
10         S++;
11     }
12 }
```

Code Example 7.6

```
1 //initialize with n permits
2 Semaphore semaphore = new Semaphore(n);
3
```

```
semaphore.acquire();
    //no more than n threads can be here at once
    //the semaphore will block until a thread calls release
semaphore.release();
```

Remark 7.7. Semaphores are a generalization of locks and can be used to implement locks, as well as other synchronization primitives.

Definition 7.8. A **binary semaphore** is a semaphore that can only have values 0 and 1 this can be used to implement locks.

Definition 7.9. A **rendezvous** is a location in the code where threads have to wait until all threads have reached that location before they can continue.

Remark 7.10. For two threads, a rendezvous can be implemented with two binary semaphores.

Code Example 7.11

```
void thread1() {
    A.acquire();
    B.release();

    B.acquire();
    A.release();
}

void thread2() {
    B.acquire();
    A.release();

    A.acquire();
    B.release();
}
```

Important 7.12. Using Semaphores is often problematic as the programmer has to have deep knowlegde of the algorithm to correctly implement the synchronization.

7.3 Barriers

Definition 7.13. A **barrier** is a synchronization primitive that is used to coordinate the execution of multiple threads. It is essentially a counter that is used to keep track of the number of threads that have reached the barrier.

Remark 7.14. Barriers are used to implement parallel algorithms that require threads to synchronize at certain points in their execution. Wait until all threads have reached the barrier, then release them all at once.

Definition 7.15. A **two-phase barrier** is a barrier that is implemented with two semaphores and a counter.

Code Example 7.16

```
mutex=1, barrier1=0, barrier2=1, count=0;

aquire(mutex);
    count++;
    if (count == n) {
        accuire(barrier2);
        release(barrier1);
    }
release(mutex);

acuire(barrier1); release(barrier1);

aquire(mutex);
    count--;
    if (count == 0) {
        accurie(barrier1);
        release(barrier2);
    }
release(mutex);
```

7.4 Producer Consumer Pattern

Definition 7.17. The **producer consumer pattern** is a classic synchronization problem that illustrates the use of semaphores to coordinate the execution of multiple threads.

Example 7.18. We can prodcuer-consumer patterns ot build data flow parallel programs for example pipelines.

$$T_0 \to T_1 \to T_2 \dots T_n$$

where T_0 is a producer, T_n is a consumer, and $T_1, \dots, T_{n-1}$ are consumers of the previous thread and producers of the next thread.

Remark 7.19. Note that in a basic pipeline T_i can only start producing once $T_(i+1)$ is ready to consume, as there is not buffer inbetween them. An **improvement** would be to have buffer queues between each node where T_i can produce into and $T_(i+1)$ can consume from. We stall T_i iff the queue Q_i is full.

Definition 7.20. Such a buffer-queue is called **Producer-Consumer Queue**.

Remark 7.21. To implement a PC Queue we can use Semaphores that signal the producers when the queue has space and signal the consumers when the queue has data.

Code Example 7.22

```
void enqueue(long x) {
    try {
        nonFull.acquire();
        manipulation.acquire();
        buffer[in] = x;
        in = (in + 1) % n;
    } catch (Exception e) {}
    finally {
        manipulation.release();
        nonEmpty.release();
    }
}
```

Code Example 7.23

```
long dequeue() {
    long x = 0;
    try {
        nonEmpty.acquire();
        manipulation.acquire();
        x = buffer[out];
        out = (out + 1) % n;
    } catch (Exception e) {}
    finally {
        manipulation.release();
        nonFull.release();
    }
    return x;
}
```

7.5 Monitors

Definition 7.24. A **monitor** is a synchronization primitive that allows threads to wait on condition variables to be notified about changes in the state of the monitor.

Definition 7.25. A **condition variable** is condition that threads can wait on. Other threads can signal the condition variable to notify waiting threads that the condition has changed.

Code Example 7.26

```
Monitor buffer = new BufferMonitor(10);
Condition nonFull = buffer.newCondition();
Condition nonEmpty = buffer.newCondition();

void producer() {
    buffer.lock();
    try {
        while (buffer.isFull()) {
            nonFull.await();
        }
        buffer.enqueue(i);
        nonEmpty.signal();
    } finally {
        buffer.unlock();
    }
}

void consumer() {
    buffer.lock();
    try {
        while (buffer.isEmpty()) {
            nonEmpty.await();
        }
        buffer.dequeue();
        nonFull.signal();
    } finally {
        buffer.unlock();
    }
}
```

8 Concurrency Theory

8.1 Linearizability

Definition 8.1. A **history** is a partially ordered set of invocations and responses of operations that appear in a distributed program.

Definition 8.2. A history H is **linearizable** if it can be extended to a history G by

- appending zero or more responses to pending invocations that took effect
- discarding zero or more pending invocations that did not take affect

such that G is equivalent to a legal sequantial history S with $\rightarrow_G \subseteq \rightarrow_S$.

Remark 8.3. Note that $\rightarrow_G$ is a total order of the operations in G that satisfies the leagal sequential order of S.

Theorem 8.4. History H is linearizable $\iff$ for every object x $H|x$ is linearizable.

Remark 8.5. The consequence of this theorem is that linearizability can be proven in isolation for each object and then composed.

8.2 Sequential Consistency

Definition 8.6. A history H is **sequentially consistent** if it can be extended to a history G by

- appending zero or more responses to pending invocations that took effect
- discarding zero or more pending invocations that did not take affect

such that G is equivalent to a legal sequential history S.

Remark 8.7. Sequential consitancy does not require the preservation of the real-time order and only has to respect program order of each thread.

Theorem 8.8. Sequential consitency is not a local property thus we lose composability.

Remark 8.9. Another idea: Programs should respect real-time order of algorithms separated by periods of quiescence. Quiescence consitency requires non-overlapping methods to take effect in real-time order.

8.3 Consensus

Definition 8.10. A **consensus object** has a function *decide*(*value*) that can be invoked by multiple threads and returns the same value for all threads where value is one of the values v_i proposed by the threads.

Remark 8.11. The consensus object allows threads to agree on a single and valid output value.

Definition 8.12. The **Consensus Number** is the maximum number of nodes that can solve consensus validly, sequentially consitent and wait-free.

Theorem 8.13. Atmoic Registers have a consensus number of 1 while Test and Set and Get and Set have consensus number of 2 and Compare and Swap has an infinite consensus number.

Proof. **CAS has infinite consensus number:** This can be proven by implementing a valid, sequentially consitent and wait-free consensus object using compare and swap. □

Code Example 8.14

```
class CASConsensus {
    private final int FIRST = -1;
    private AtomicInteger r = AtomicInteger(FIRST);
    private AtomicIntegerArray proposed;

    public int decide(int value) {
        int i = ThreadID.get();
        proposed.set(i, value);
        if (r.compareAndSet(FIRST, i)) {
            return proposed.get(i);
        } else {
            return proposed.get(r.get());
        }
    }
}
```

Example 8.15. Lets prove that there is no wait free implementation of a FIFO Queue using atomic registers.

Proof. Assume there is a wait free implementation. Then i can use the FIFO Queue to solve $n = 2$ consensus which is a contradiction as atomic registers have a consensus number of 1. □

Remark 8.16. We can solve consensu with FIFO Queue by setting $q[0] = 1$ and $q[1] = 0$ if p_1 wants to output 1 and p_2 wants to output 0, then they both try to dequeue the value who ever gets the value 1 will win and the other will lose.

8.4 Transactional Memory

Remark 8.17. Note that synchronization with locks is hard to implement, not composable and pessimistic.

Remark 8.18. The idea behind transactional memory is to remove the burden of synchronization from the programmer and make the hardware (HTM) or the underlaying software (STM) responsible to enforce synchronization where declared/needed.

Definition 8.19. In transactional memory the programer can declare sections of his code as **atomic** and leave it to the system and hardware to ensure this code block takes effect atomically (also called a **transaction**).

Lemma 8.20. Transactions run isolated and are consitent and atomic, thus other threads can only observe the start or the end state of the transaction. Where HTM offers fast execution at limited size while STM might offer larger transactions at higher time/enery consumption.

Definition 8.21. In TM we define for each transaction the **read set** of read variables during the transaction and the **write set** of written variables during the transaction.

Remark 8.22. Conflicts can occure when the write set of transation A overlaps with the read or write set of transaction B.

Remark 8.23. Transactional Memory can be implemented on a Hardware level (HTM) (often using the existing cache coherence infrastructure) or on the Software level (STM).

Definition 8.24. In STM we have to distinguish between **Weak isolation** and **strong isolation**.

- **Weak isolation** only isolates Transactions from each other.
- **Strong isolation** isolates Transactions from each other and from operations outside the transaction.

Remark 8.25. STMs are still actively being developed. There is a lot of reseach happening in the field of Hybrid (STM and HTM) implementations.

Definition 8.26. Nesting in a TM implementation referes to how transactions in transactions behave (ex. if an inner transaction aborts).

Definition 8.27. Flat Nesting is the simplest form of nesting where:

- inner transactions are invisible to the outside world
- if an inner transaction aborts the outer transaction aborts

Remark 8.28. We will use Scala's Software Transactional Memory (STM) to implement and demonstrate our ideas.

8.5 Distributed Memory and Message Passing

Remark 8.29. Shared mutable data is difficult to handle. A possible solution is functional programming. Alternatively, we can use local mutable data where threads send data to each other via message passing.

Definition 8.30. Message Passing can be either synchronous or asynchronous:

- **Synchronous**: Sender blocks until the message is received.
- **Asynchronous**: Fire and forget (messages are buffered).

Remark 8.31. In contrast to shared memory, distributed memory architectures rely on isolated mutability.

Definition 8.32. MPI (Message Passing Interface) is a standard for distributed memory programming. Processes are grouped into **communicators**, and each process is assigned a unique identifier called a **process rank**.

Definition 8.33. Message passing operations can be:

- **Blocking**: The operation blocks until the data is safely sent/received.
- **Non-blocking**: The operation returns immediately, allowing for overlap between computation and communication.

Definition 8.34. Apart from send/receive operations MPI offers:

- **Barriers** to synchronize processes
- **Broadcast** to send data from one process to all processes
- **Reduce** to perform operations on the data from all processes and return the result to one process
- **Scatter** to send data from one process distributed in chunks to all processes
- **Gather** to receive data from all processes distributed in chunks and combine and send it to one process

8.5.1 Actor Model

Definition 8.35. Actor Model is a model of distributed computing where actors are the fundamental units of computation. Actors communicate with each other by sending messages.

Remark 8.36. Actors are isolated from each other and can only communicate with each other by sending messages. This eliminates the need for locks or other synchronization primitives.