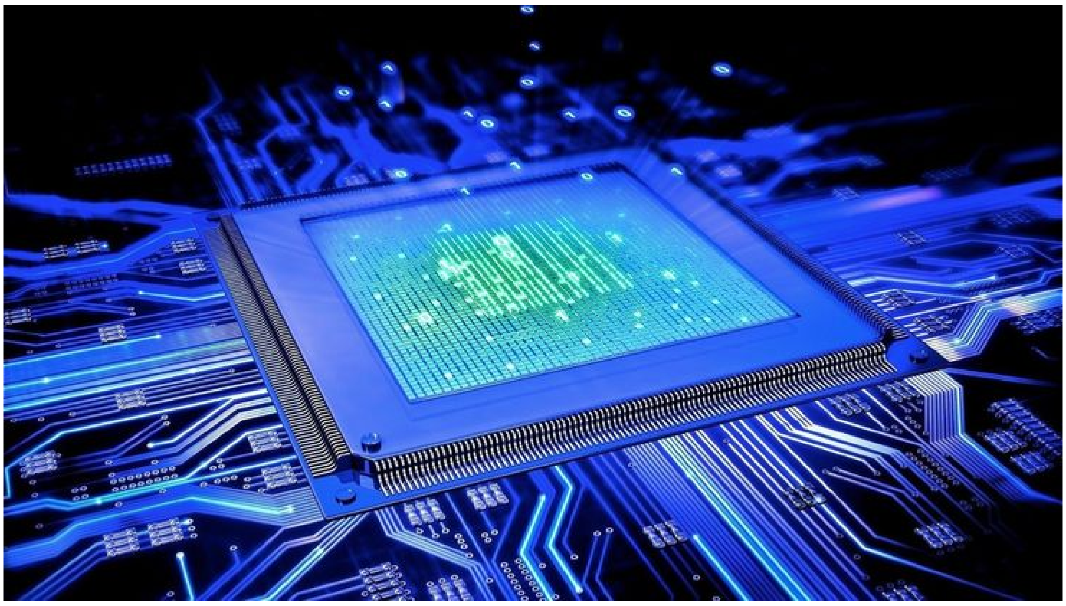
ETH Zurich

Department of Computer Science

# Parallel Programming

*Author:*
Soel Micheletti



Spring 2020

# Preface

This script is a summary of the ETH Course *Parallel Programming*. First of all, I would like to acknowledge Lasse Meinen to give me access to the repository of his PVK Skript, a very good source from where I took part of this summary. Some parts are slightly more formal than what is required in order to be succesful in this course, feel free to skip some proofs if your primary concern is the exam preparation. If you find any error or you have any suggestion, don't hesitate to contact me at *msoel@ethz.ch*: I'm happy to hear your feedbacks!

# Contents

# Introduction 1

In the past, people used to code sequential programs without concerning too much about performance. Better said, they were concerned with the efficiency of their algorithms, but not on how to implement them on hardware. This was a consequence of *Moore's Law*, *i.e.* the observation that the number of transistors doubles every two years and hence the same version of a sequential program automatically became faster with the new generation of CPU. However recently, Moore's Law has not been repealed: each year, more and more transistors fit into the same space, but their clock speed cannot be increased without overheating. In order to improve performance, manufacturers are instead turning to *multicore* architectures, in which multiple processors communicate directly through shared hardware caches. Multiprocessor chips make computing more effective by exploiting *parallelism*: harnessing multiple processors to work on a single task. In this course we focus on how to program multiprocessors that communicate via a shared memory. Such systems are often called *multicores*. The challenges (and the solutions) that we will explore, arise at all scales of multiprocessor systems: at a very small scale such as a laptop (even the ones with a single core) and on a large scale such as in the case of supercomputers or systems distributed over many machines.

Welcome to parallel programming! In this script we are going to cover some very interesting topics: some in depth and in some cases we will just scratch the surface. We are going to talk about performance and correctness aspects of parallel computation. Moreover, we will have a broad look that will touch other areas of Computer Science: algorithms, operating systems and various logical problems. We hope that you will not only get the details necessary to prepare for your exam (in this case, our number one recommendation is *do not memorize too many details, practice, practice and practice again*), but them main goal is that you will be able to build a big picture of the situation with a lot of links between the concepts, by also considering what you learn in other courses.

Before we begin our journey, we point out a very important distinction that is made in this lecture, *i.e.* the difference between *parallelism* and *concurrency*. The concept of *parallelism* implies the use of *additional computational resources* to solve a problem faster. As an example think that you have to sum all elements of an array: the idea of parallelism is that, instead of *doing everything by yourself*, you call some other friends (which, out of the analogy, we will call *threads*) and you split the work, *i.e.* each person sums of a part of the array while the other are summing theirs and at the end you sum the results together. The concept of *concurrency* might sound similar at the beginning, but is quite different: concurrency is about correctly and efficiently controlling access to multiple threads to shared resources. In the contexts that we will encounter throughout this course, we usually have that parallelism implies concurrency (this might not always be true in some pathological cases, but it holds in all real world situations), but concurrency does not always imply parallelism (this means, for example, that if you run programs with multiple threads on a single core you still have to carefully manage access to mutable shared resources).

## 1.1 Theoretical Perspective

The most common theoretical model to reason about parallelism, and particularly about *parallel algorithms*, is the *Parallel Random Access Machine* (PRAM) model, which considers a number of RAM machines, all of whom have access to some shared memory. We note that this model is most interesting from a theoretical perspective and it ignores a range of practical issues, instead trying to focus on the fundamental aspects of "parallelism in computation". While there have been a number of different theoretical models introduced throughout the years for parallel computation, PRAM has remained the primary one and it has served as a convenient vehicle for developing algorithmic tools and techniques. Many of the ideas developed in the context of PRAM algorithms have proved instrumental in several other models and they are also being used in practical settings of parallel computations. In the PRAM model, we consider $p$ number of RAM processors, each with its own local registers, which all have access to a global memory. Time is divided into synchronous steps and in each step, each processor can do a RAM operation or it can read/ write to one global memory locations. The model has four variations with regard to how concurrent reads and writes to one global memory are resolved; Exclusive Read Exclusive Write (EREW), Concurrent Read Exclusive Write (CREW), Exclusive Read Concurrent Write (ERCW) and Concurrent Read Concurrent Write (CRCW). When concurrent writes on the same memory location are allowed, there are variations on how the output is determined. A simple rule is to assume that an arbitrarily chosen one of the write operations take effect.

## 1.2 Threads

Before writing any parallel or concurrent programs , we need some way of making multiple things happen at once and some way for those different things to communicate.The programming model we will assume is *explicit threads with shared memory*.

A thread is like a running sequential program (formally we say that it is an *independent sequences of execution*), but one thread can create other threads that are part of the same program and those threads can create more threads, etc. Two or more threads can communicate by writing and reading fields of the same object. They can see the same objects because we assume that they share memory. Conceptually, all the threads that have been started but not yet terminated are ”running at once” in a program. In reality, they may be running at any particular moment, as there may be more threads than processors or a thread may be waiting for something to happen before it continues. When there are more threads than processors, it’s up to the Java implementation, with help from the underlying operating system, to find a way to let the threads ”take turns” using the available processors. This is called *scheduling* and is a major topic in operating systems. All we need to care about is that it's not under our control: we create the threads and the system schedules them.

We will now see how to create new threads in Java. The details vary in different languages. In addition to creating threads, we will need other language constructs for coordinating them. For example, for one thread to read the result of another thread's computation, the reader often needs to know whether the writer is done. Creating a new thread in Java requires that you define a new class and then perform two actions at run-time:

1. Define a subclass of *java.lang.Thread* and override the *public* method *run*, which takes no arguments and has return type *void*. The *run* method will act like "main" for threads created using this class. It must take no arguments, but the example below shows how to work around this inconvenience.
2. Create an instance of the class created in step 1. Note that this doesn't create a running thread. It just creates an object.
3. Call the *start* method of the object you created in step 2. This step does the "magic" creation of a new thread. The new thread will execute the *run* method of the object. Notice that you do not call *run*; that would just be an ordinary method call. You call *start*, which makes a new thread that runs *run*. The new thread terminates when its *run* method completes.

In general every Java program has at least one execution thread (and the first execution thread calls the *main()* method). Each call to *start()* method of a *Thread* object creates a new thread. The program ends when all non-daemon thread (*i.e.* thread that assist the operating system) finish. The various threads can continue even if *main()* returns.

**Example 1.2.1** Here is a useless Java program that starts with one thread and then creates 20 more threads:

```
public class Useless extends Thread{
    int i;
    Useless(int i){this.i = i; }
    public void run(){
        System.out.println("Thread" ++ i ++ "says hi");
        System.out.println("Thread" ++ i ++ "says bye");
    }
}
public class M{
    public static void main(String[]args){
        for(int i = 1; i < 21; i++){
            Thread t = new Useless(i);
            t.start();
        }
    }
}
```

When running this program, it will print 40 lines of output, the order of which is unpredictable. In fact, if you run the program multiple times, you will probably see the output appear in different orders every run. The example shows that there is no guarantee that threads created earlier will run earlier. Therefore, multithreaded programs are *nondeterministic*. This is an important reason why multithreaded programs are much harder to test and debug.We can also see how this program worked around the rule that *run* is not allowed to take any arguments. Any "arguments" for the new thread are passed via the constructor, which then stores them in fields so that *run* can later access them. We close with a performance consideration: creating a new thread is an expensive operation which requires time and introduce a significant *overhead*.

We mentioned previously that we'd like to make a thread wait before reading a value until another thread has finished its computations, i.e. its *run* method. We can do this with the *join* keyword, which we'll introduce through another somewhat useless example.

**Example 1.2.2** Here is a Java program that starts with one thread which spawns 20 more threads and waits for all of them to finish.

```
public class Useless extends Threads{
    int i;
    Useless(int i){this.i = i; }
    public void run(){
        System.out.println("The double of" ++ i ++ "is ++ i*2);
    }
}
public class Main{
    public static void main(String[]args){
        Threads[] threads = new Threads[20];
        for(int i = 0; i < 20; i++){
            Thread t = new Useless(i + 1);
            t.start();
            threads[i] = t;
        }
        for(int i = 0; i < 20; i++){
            try{threads[i].join(); }catch(InterruptedException e){}
        }
        System.out.println("All done!);
    }
}
```

The *join* method can throw an *InterruptedException*, which means we need to wrap it in a try-catch block.

## Thread states

If we want to be able to talk about the effects of different thread operations, we need some notion of *thread states*. In short, a Java thread typically goes through the following states:

- *Non-Existing*: Before the thread is created, this is where it is. We do not know too much about this place, as it's not actually on this plane of reality, but it's somewhere out there.

- *New*: Once the *Thread* object is created, the thread enters the new state.
- *Runnable*: Once we call *start()* on the new thread object, it becomes eligible for execution and the system can start scheduling the thread as it wishes.
- *Blocked*: When the thread attempts to acquire a lock, it goes into a blocked state until it's actually obtained the lock, upon which it returns to a runnable state. In addition, calling the *join()* method will also transfer at thread into a blocked state.
- *Waiting*: The thread can call *wait()* to go into a waiting state. It'll return to a runnable state once another thread calls *notify()* or *notifyAll()* and the thread is removed from the waiting queue.
- *Terminated*: At any point during execution we can use the *interrupt()* to signal the thread that it should stop execution. It will then transfer to a terminated state. Note that when the thread is in a runnable state, it needs to check whether its interrupted flag is set itself, it won't transfer to the terminated state automatically. Of course, exiting the *run* method is equivalent to entering a terminated state. Once the garbage collector realizes that the thread has been terminated and is no longer reachable, it will garbage collect the thread and it will return to a non-existing state, completing the cycle.

## 1.3 Bad Interleavings and Data Races

A *race condition* is a mistake in your program such that, whether the program behaves correctly or not, depends on the order that the threads execute. Race conditions are very common bugs in concurrent programming that, by definition, do not exist in sequential programming. We distinguish two types of race conditions.

One kind of race condition is a *bad interleaving*. The key point is that "what is a bad interleaving" depends entirely on what you are trying to do. Whether or not it is okay to interleave two bank-account withdraw operations depends on some specification of how a bank is supposed to behave.

**Example 1.3.1** Suppose we have the following implementation of a *peek* operation on a concurrent stack.

```
static <T> T peek(Stack<T> s){
    T ans = s.pop();
    s.push(ans);
    return ans;
}
```

Assume that the *pop* and *push* methods are implemented correctly. While *peek* might look like it's implemented correctly, the following interleaving might occur: first a thread $B$ pushes an element $x$ into the stack. Then thread $A$ begins the *peek* operation and pops $x$. After this thread $B$ pushed the element $y$ into the stack. Thread $A$ goes on with its *peek* by pushing back $x$ into the stack and returning this value. What if now thread $B$ performs a *pop()* operation? The result should be $y$, but with this interleaving, the program would (wrongly) return $x$.

The other kind of race condition are *data races*, a phenomenon that is better described as a ″simultaneous access error″, although nobody uses that term. There are two kinds of data races:

- When one thread might read an object field at the same moment that another thread writes the same field.
- When one thread might write an object field at the same moment that another thread also writes the same field.

Notice that it is not an error for two threads to both read the same object field at the same time. Our programs must never have data races even if it looks like a data race would not cause an error - if our program has data races, the execution of your program is allowed to do very strange things.

**Example 1.3.2** Let's consider an example.

```
class C{
    private int x = 0;
    private int y = 0;

    void f(){
        x = 1;
        y = 1;
    }
    void g(){
        int a = x;
        int b = x;
        assert(b>=a);
    }
}
```

Notice that $f$ and $g$ are not synchronized, leading to potential data races on fields $x$ and $y$, Therefore, it turns out that the assertion in $g$ can fail. But there is no interleaving of operations that justifies the assertion failure, as can be seen through a proof by contradiction:

Assume the assertion fails, meaning *!(b>=a)*. Then *a==1* and *b==0*. Since *a==1*, line B happened before line C. Since A must happen before B, C must happen before D, and ″happens before″ is a transitive relation, A must happen before D. But then *b==1* and the assertion holds.

There is nothing wrong with the proof except its assumption that we can reason in terms of ″all possible interleaving″ or that everything happens in certain orders. We can reason this way only if the program has no data races.

## Other Models

We've introduced a programming model of explicit threads with shared memory. This is, of course, not the only programming model for concurrent or parallel programming. Shared memory is often considered convenient because communication uses "regular" reads and writes of fields to objects . However, it's also considered error-prone because communication is implicit; it requires deep understanding of the code/-documentation to know which memory accesses are doing inter-thread communication and which are not. The definition of shared-memory programs is also much more subtle than many programmers think because of issues regarding data races, as discussed in the previous section.

Three well-known, popular alternatives to shared memory are presented in the following. Note that different models are better suited for different problems. Models can be abstracted and built of top each other as we wish or we can use multiple models in the same program (e.g. MPI with Java).

*Message-passing* is the natural alternative to shared memory. In this model, we have explicit threads, but they do not share objects. To communicate, they exchange *messages*, which sends a copy of some data to its recipient. Since each thread has its own objects, we do not have to worry about other threads wrongly updating fields. But we do have to keep track of different copies of things being produced by messages. When processes are far apart, message passing is likely a more natural fit, just like when you send email and a copy of the message is sent to the recipient.

*Dataflow* provides more structure than having "a bunch of threads that communicate with each other however they want." Instead, the programmer uses primitives to create a directed acyclic graph. A node in the graph performs some computation using inputs that arrive on its incoming edges. This data is provided by other nodes along their outgoing edges. A node starts computing when all of its inputs are available, something the implementation keeps track of automatically.

*Data parallelism* does not have explicit threads or nodes running different parts of the program at different times. Instead, it has primitives for parallelism that involve applying the same operation to different pieces of data at the same time. For example, you would have a primitive for applying some function to every element of an array. The implementation of this primitive would use parallelism rather than a sequential for-loop. Hence all the parallelism is done for you provided you can express your program using the available primitives.

We conclude this introductory chapter with a branch of parallel programming which is likely to become crucial in the following years. Since we are in the era of *Big Data*, it is likely that in the future we will not just have to exploit parallelism within a single machine, but to coordinate the work of several computers.

*Massive Parallel Computing (MPC)* considers a system composed of some $M$ number of machines, each of which has a memory of size $S$ words. This memory $S$ is typically assumed to be significantly less than the input size $N$. The input is distributed arbitrarily across the machines, *e.g.* in the case of sorting each machine holds some of the items, and in the case of graph problems each machine holds some of the edges of the graph. The computation in MPC proceeds in lock-step syncrhonous rounds 1, 2, 3, ... Per round, each machine can do some computation on the data that it holds, and then it sends messages to the other machines. The model does not impose any particular restriction on the computations that each machine can perform in one round. As for the commutation, the limitation is simple: per round, each machine can send at most $S$ words and it can receive at most $S$ words. Our algorithm needs to just describe what information should be sent from each machine to each other machine, subject to the above constraints.

# Parallelism | 2

## 2.1 Performance

In an ideal world, upgrading from a uniprocessor to an $n$−way multiprocessor should provide about an $n$−fold increase in computational power. In practice, sadly, this never happens. The primary reason for this is that most real-world computational problems cannot be effectively parallelized without incurring the costs of inter-processor communication and coordination. Consider five friends who decide to paint a five-room house. If all the rooms are the same size, then it makes sense to assign each friend to paint one room, and as long as everyone paints about the same rate, we would get a five-fold speed-up over the single painter case. The task becomes more complicated if the rooms are of different sizes. For example, if one room is twice the size of the others, then the five painters will not achieve a five-fold speedup because the overall completion time is dominated by the one room that takes the longest to paint. Another important factor that explains why we do not achieve the perfect speedup is the fraction of *non parallelizable* work: some parts of the program might not be parallelizable at all and, as we will see with a rigorous mathematical explanation shortly, this might become a bottleneck. Now we take a look at the problem from a mathematical perspective.

Before we begin, we need to introduce some terminology. Let $P$ denote the number of processors available during the execution of a program.

**Definition 2.1.1** *$T_P$ is the time to execute a program on $P$ processors. Important special cases are the sequential time $T_1$ and $T_\infty$, i.e. the execution time when infinitely many processors are available.*

**Definition 2.1.2** (Speedup) *The* speedup *$S_P$ of a program is given by*

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

### Amdahl's Law

The speedup of a parallel program can be decreased by several factors. One of them is the overhead caused by introducing parallelization in the program. As we have pointed out in the previous chapter (and as you have seen in *Assignment 2*), creating threads (a necessary ingredient to parallelize our program) is costly. Another factor that decreases the speedup is given by the *sequential part* of the program: imagine that you have infinite friends that can help you painting the house described at the beginning of this section. If you need ten minutes in order to explain them how to work, this is the sequential part of your algorithm and is definitely a bottleneck to the final speedup, in the sense that the people will definitely need at least ten minutes to clean the house. In order to derive a precise mathematical formulation for this concepts we denote with $W_{ser}$ the time spent doing non-parallelizable, serial work and $W_{par}$ denote the time spent doing parallelizable work. We have

$$T_1 = W_{ser} + W_{par}$$

What happens when we have more processors? Of course, the time spent doing serializable work stays the same (new processors are not useful for this work), but we get an improvement in the parallelizable work. We get the following lower bound

$$T_P \geq W_{ser} + \frac{W_{par}}{P}$$

Where the last equation is an instance of application of *Brent's principle*, which will be discussed later. By plugging this expression we get the speedup given by Amdahl's Law.

**Theorem 2.1.1** (Amdahl's Law)

$$S_P = \frac{T_1}{T_p} \leq \frac{W_{ser} + W_{par}}{W_{ser} + \frac{W_{par}}{P}}$$

*If we denote with $f$ the non-parallelizable, serial fraction of the toal work we get*

$$S_p \leq \frac{1}{f + \frac{1-f}{P}}$$

When we let $P$ go to infinity, we see that $S_\infty \leq \frac{1}{f}$. In order to see why this is such an important result, we can try plugging in a couple of values. Assume that 25% of a program is non-parallelizable. This means that even with the *IBM Blue Gene/P* supercomputer with its 164000 cores, we can only achieve a speedup of at most 4. While a depressing result at first sight, this makes perfect sense when we consider the fact that these 25% are completely fixed, in the sense that the execution time can not possibly be reduced past this point. The conclusion that we can draw from this result is that it's worth investing some more time into reducing the sequential fraction of our program, *e.g.* by reducing the overhead of communicating between threads or by picking a better algorithm (see the third year course *Algorithms, Probability and Computing* for some very exciting examples).

## Gustafson's Law

Amdahl's Law considered a fixed workload and provides us with an upper bound on the speedup achievable when increasing the number of processors at our disposal. Gustafson looked at the problem from another perspective: he fixed the time available and he looked at how much work was it possible to do when increasing the number of processors available. In other words, we consider the time interval to be fixed and we look at the problem size. Let $W$ denote the work done in a fixed time interval.

We get

$$W = f \cdot W + (1 - f) \cdot W$$

As we increase the number of processors at out disposal, we can only speed up the parallel fraction of our program. The serial fraction remains the same. Letting $W_P$ be the work done with $P$ processors at our disposal, we get

$$W_P = f \cdot W + P \cdot (1 - f)W$$

How much time do we need to do $W_P$ when we have a single processor available? What about the case where we have $P$ processors? We get the following lower bounds

$$T_1 \geq f \cdot W + P \cdot (1 - f)W$$

$$T_p \geq f \cdot W + \frac{P \cdot (1 - f)W}{P} = W$$

By plugging this values into the speedup formula we obtain *Gustafon's Law*

**Theorem 2.1.2** (Gustafson's Law)

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

In this case we see that, by having a very large number of processors, we always get a larger speedup. For this reason we can say that Gustafson's Law is more optimistic than Amdahl's Law.

## 2.2 Pipelining

In the course *Digital Design and Computer Architecture* you have seen several basic concepts of computer architecture. Particularly you have seen the *Von Neumann Architecture* of a processor, which assumes that the processor is made of two basic components: CPU and memory. On the one hand the CPU contains registers (a very small memory which can be used to save variables that are accessed over and over again), a program counter (that saves the address in memory of the *next instruction* that has to be executed) and an ALU which performs arithmetical operations. On the other hand the memory contains both data and programs (*i.e.* there are some addresses in memory which save instructions that will be fetched, decoded and executed by the CPU when the program counter will contain their address). How can this model become faster? There are two important solutions

- When memory becomes a bottleneck introduce a hierarchical memory system. Sometimes the CPU is very fast, but the transfer of instructions and data to/ from memory takes a long time. A solution to this issue is introducing different levels of *caches* to exploit two heuristic principles called temporal location (*i.e.* the hypothesis a memory that was accessed recently is likely to be accessed again in the near future) and spatial location (*i.e.* if we accessed a region in memory it is likely that we will access its neighbor addresses, this is particularly true for a sequential array access). There are different levels of caches with different sizes, and in general it holds that *smaller memory* implies *faster access*. By having different levels of caches one hopes that the expensive accesses to main memory will be rare and hence, at least most of the times, we will have the data that we need in one of those caches. In order to achieve this goal there are several algorithms that answer the question *what should we save in the cache?* You will get an answer to this and other interesting questions in the course *Systems Programming and Computer Architecture*, for the purpose of this course is enough to know that by organizing the memory in a hierarchical way with several caches of different size (and access time) the access to memories get a lot faster and hence they speedup the programs a lot.

- Exploiting parallelism in the hardware. In this model, which is the one that we study in this section, the programmer has *no idea* that there is parallelism going on. The programmer writes a sequential program and assumes that the hardware will execute the instructions in order, but then the hardware uses some tricks to make this sequential program run faster. There are three basic main approaches to enhance parallelism *without exposing it the programmer*. The first one is *vectorization* that is used when the same operation is executed on an aggregate of $N$ items (such as doing the same operation on all elements of an array). Instead of taking one element from memory, doing the operation and writing back the result to memory, in this case the CPU directly takes $k$ elements from memory, executes the operation on all $k$ elements and writes back the results of all $k$ elements to memory. If the operation to be executed is an arithmetic operation and the CPU has more than a single ALU, the operations can be executed in parallel, otherwise this approach still gives a minor overhead compared to the naive version. The second approach is *Instruction Level Parallelism (ILP)*, where the CPU reorders the instructions of the program in some way (and hopefully, by doing this, it will improve its performance) but then it makes sure that for the programmer it appears in the same way as if everything was executed sequentially. An important concepts is detecting which instructions are *independent* from each other and hence can be executed in parallel or reordered. In the context of ILP arise some keywords such as *superscalar CPUs* (*i.e.* CPUs with multiple functional units), speculative execution (*i.e.* when there is a conditional branch the processor might guess which branch will be taken and executing it speculatively, by gaining in performance if the guess was right and by paying a time price if the guess was wrong), *out of order execution* (*i.e.* change the execution order of instructions as far as the programmer observes the sequential order) and *pipelining*, which we discuss in greater detail here.

Pipelining is a technique where multiple independent instructions are overlapped in execution through the use of multiple execution units, provided that they are available for use. Before we dive into examples we introduce some important concepts.

**Definition 2.2.1** (Throughput) *Throughput is amount of work that can be done by a system in a given period of time. Larger is better. In CPUs it corresponds to the number of instructions completed per second and in general it can be approximated as follows*

$$Throughput \sim \frac{1}{\max(computationTime(stages)}$$

*Note that we usually consider throughput when the pipeline is fully utilized, i.e. ignoring lead-in and lead-out time.*

**Definition 2.2.2** (Latency) *Latency is the time needed to perform a computation (e.g. a CPU instruction), including wait time resulting from resource dependencies. Lower is better.*

The goal would be to have a high throughput and a low latency. However, this goals are often conflicting. Another important concept is the one of *balanced pipeline*: we say that a pipeline is balanced if its latency remains constant over time.

In order to get insights into the world of pipelining we discuss several variants of the *washing machine pipeline*. In this example we have four stages that are executed five times: the washing phase (5 seconds), the dryer phase (10 seconds), the folding phase (5 seconds) and the closet phase (10 seconds). We want to pipeline the stages in order to save time compared to the sequential version.

- First attempt: just wait until the resource you need is available. In this context the first load takes 30 seconds. The second load takes five seconds more: in facts, after the washing phase, it has to wait for the dryer to finish. The third one takes ten seconds more; in facts, after the washing phase, it has to wait for the drying phase of load number two to be over (which will be ten seconds later). Each load takes five seconds more of the previous one. The total time for all loads is 70 seconds. This pipeline can work, however it cannot bound the latency of a Load as it keeps growing. If we want to bound this latency, one approach is to make each stage take as much time as the longest one, thus balancing it.
- Second attempt: we let each phase take the same time as the longest one (in this case 10 seconds). In this case *each* load takes 40 seconds and in total we need 80 seconds (longer than before). This pipeline is a bit wasteful, but it bounds the latency. Can we somehow get a bound on latency while improving the time/throughput?
- Third attempt: if we introduce multiple units on the longest stages (*i.e.* the drying and the closet phase) we can get a better throughput while keeping the latency constant. The cost to achieve this result is buying more functional units. This is exactly the idea behind superscalar processors.

**Example 2.2.1** (CPU Pipeline) The CPU pipeline is a technique for implementing instruction-level parallelism within a single processor. Pipelining attempts to keep every part of the processor busy with some instruction by dividing incoming instructions into a series of sequential steps performed by different processor units with different parts of instructions processed in parallel. The stages of such a pipeline are:

- Instruction fetch: where the CPU fetches the next instruction by looking at the address in the CPU's instruction pointer. In this stage advanced techniques such as speculative execution can come into play.
- Instruction decode: the CPU receives the instruction from memory and *understands* what it has to do by examining the sequence of bits.
- Execution: the CPU executes the *heart* of the instruction, without memory access.
- Memory access: the CPU reads the necessary data from memory (or caches).
- Writeback: the CPU writes the necessary data in memory.

We point out that the solutions that we discussed so far are implemented *at the hardware level* and hence a programmer does not need to use those tricks in high level programming.

We have seen that originally computers followed the idea of the Von Neumann machine and that CPUs architects improved sequential execution by exploiting Moore's Law and some hardware parallelism tricks. For a long time sequential programs were becoming exponentially faster with each new CPU and hence most programmers did not worry about performance in this sense: they just waited for the next CPU generation. But CPU architects hit walls as the *power wall* (*i.e.* the fact the faster CPU consumed more energy and hence became expensive in energy terms and overheated), the *memory wall* (*i.e.* the fact that, although there are some tricks such as the memory hierarchy the CPU was way faster than memory access) and the *ILP wall* (*i.e.* with ILP we cannot have speedup beyond a certain point). Hence it is no longer affordable to increase sequential CPU performance. Nowadays, in order to have good performance, it is important to expose parallelism to software: programmers need to write parallel programs to take advantage of the new (multicore) hardwares.

## 2.3 A typical example of parallel programming: the divide and conquer paradigm

In the course *Algorithms and Data Structures* of the first semester, you have encountered the first examples of divide and conquer algorithms. For example you have talked about the *find a star* and the *maximum subarray sum* problems. However probably, the most famous example of this paradigm you might recall is MergeSort. In MergeSort you proceed in the following way in order to sort an array:

1. You split the array in two (equally long) parts
2. You sort both parts *recursively* (or, in principle, with any sorting algorithm you want)
3. You merge the resulting sorted subarray into a single, sorted array

This paradigm can be applied to many problems, and is suitable for parallelization. In facts, in all divide and conquer algorithms, you split the original task into two (or more) subtasks which you solve recursively. All this recursive tasks are independent and hence can be performed in parallel. In the case of MergeSort you can split your array in two parts and, while you sort the first part, you call a friend that sorts the second part at the same time. At the end, you just have to merge both results.

While divide and conquer algorithms seem very natural to parallelize, we currently have no easy way of determining the efficiency of these algorithms in a parallel setting and it would be quite tedious to implement them using regular Java threads. We will therefore first introduce a way of visualizing the performance of a divide and conquer algorithm and will then look at two different libraries to see how we can easily implement this paradigm.

## Task Graphs

We can describe program executions as directed acyclic graphs. We have that nodes are the pieces of work that the program performs and each node will have a constant execution time per task (in most cases we will interpret the execution time per node as one). Edges represent that the source node must complete before the target node begins, i.e. there is a data dependency along the edge. We will now connect this visual representation of the execution of a program with the terminology introduced in the previous section by introducing two additional important concepts.

**Definition 2.3.1** (Work) *The work done in total in order to solve the task is the sum of the time costs of all nodes in the graph. We denote it again as $T_1$.*

**Definition 2.3.2** (Span) *Span is the sum of the time cost of all nodes along the critical path, i.e. the longest path in the graph. The span determines the best possible execution time we can achieve by introducing an arbitrary large number of processors, i.e. $T_\infty$.*

In general we have that independent tasks (*i.e.* nodes that have no common ancestor on the path to the root) *can* execute in parallel. However they *do not have to* execute in parallel: the assignment of tasks to the CPUs is up to the scheduler (for this course you can think to the scheduler as a referee between your program and the hardware that can decide who uses the CPUs; this concept will be studied in greater details in the core course *Computer Systems*). The intuition is that *wider* task graphs can exploit more parallelism. What speedup do we have in general? We have two lower bounds on the time needed to execute the program with $P$ processors, *i.e.*

$$T_P \geq \frac{T_1}{P}$$

$$T_P \geq T_\infty$$

In general, the value that $T_P$ will have depends on the scheduling algorithm, but one can prove that with a work stealing scheduler (*i.e.* a scheduler that does not make any core idle), we have

$$T_p = \frac{T_1}{P} + \mathcal{O}(T_\infty)$$

and the hidden constant in the asymptotic notation has been empirically shown to be reasonably small.

Summarizing, we can view each computational process as a sequence of rounds, where each round consists of a (potentially large) number of computations that are independent of each other and can be performed in parallel. In this view, we refer to the total number of rounds as the *depth* of the computation and we referred to the summation of the number of computations performed over all the rounds as the total *work*. Naturally, the primary goal when designing parallel algorithms for a given task, is to have a depth as small as possible. A secondary goal is to have a small total work. Then we can relate this depth and work to the actual time measure for the computation on parallel processing systems with the equation above, also known as *Brent's principle*.

Now the natural question that comes into mind is: *how can we implement parallel algorithms with a good task graph?* In the general case, designing good parallel algorithms is challenging and we refer to other lectures (particularly, the excellent course *Algorithms, Probability and Computing*), but for the particular case of divide and conquer algorithms we present two useful frameworks that manage a lot of things for the user: the *executor service* and the *fork and join framework*.

## Executor Service

Instead of managing the threads ourselves we can use a library which manages a threadpool to which we can submit tasks. The basic idea is that we have several independent tasks that can be performed in parallel and the executor service binds each task to a thread taken from the threadpool. A task is either:

- A *Runnable* object, which implements a method *void run()* and does not return a result
- A *Callable<T>* object, which implements a method *T call()* and returns a result of type *T*

Upon submitting a task a *Future<T>* is created, which contains the result of the submitted task. Implementing a divide and conquer algorithm now looks easier, as shown in the following example.

**Example 2.3.1** We wish to submit tasks to the ExecutorService such that we can obtain some sort of result from the task's execution. We therefore need to implement a *Callable* task (i). We've additionally provided the code for creating an ExecutorService and submitting the topmost task (ii). Try running it locally and see what happens.

**Algorithm 2.1**: Sum elements in array

```
class Sum implements Callable{

    int low;                          ▷ Index where we need to start
    int high;                                        ▷ Index where we need to end
    int [] array                                     ▷ Input array
    ExecutorService ex;

    SumForkJoin(ExecutorService e, int[] ar, int lo, int hi){
        low = lo;
        high = hi;
        array = ar;
        ex = e;                                      ▷ Simple constructor
    }

    protected Integer call () throws Exception{
    if(h − l ==1){
        return arr[l]                                ▷ Base case
    }
    int mid = (h − l) / 2;
    Sum left = new Sum(ex, arr, low, low + mid);
    Sum right = new Sum(ex, arr, low + mid, h);
    Future<Integer> f1 = ex.submit(left);
    Future<Integer> f2 = ex.submit(right);
    try{
        return f1.get() + f2.get();
    }catch(Exception e){
        return 0;
    }
    }
}

public static void main(String[]args){
    int [] arr = ...
    ExecutorService ex = Executors.newFixedThreadPool(4);
    Sum top = new Sum(ex, arr, 0, arr.length);
    Future<Integer> res = ex.submit(top);
    try{
        System.out.println(res.get());
    }catch(Exception e){}
    ex.shutdown();
}
```

You should see that the program never terminates. The reason for this is that the ExecutorService limits the number of threads that can be spawned. Once all threads are occupied by tasks waiting for the result of spawned subtasks, execution is therefore halted. No threads are available to run the subtasks on, i.e. the subtasks will wait indefinitely.

With the ExecutorService, the limited thread pool size caused the program to run forever, as all currently active threads were occupied by tasks waiting for the spawned subtasks to return. While there are possible approaches that could alleviate this problem, e.g. separating the work partitioning from solving the sub-tasks,they are all non-trivial to implement and are therefore not well suited for practical purposes.

## The Fork and Join Framework

The drawbacks of Executor Service suggest that we need a framework that supports divide and conquer style parallelism. That is, when a task is waiting, it is suspended and other tasks are allowed to run. Compared to Java threads, the usage of these classes is exactly the same, but with some different names and interfaces. First, we'll introduce the required terminology.

- Instead of extending *Thread*, we extend *RecursiveTask<T>* (which returns a value) or *RecursiveAction* (which does not return any value)
- Instead of overriding *run*. we override *compute*
- Instead of calling *start*, we call *fork* (which creates a new task)
- Instead of a topmost call to *run*, we create a *ForkJoinPool* and call *invoke*
- We stil call *join*, but now it returns a value

We are deeply convinced that the best way to learn how this library works is doing a lot of exercises. Hence, for the sake of brevity, we will just expose an example here with some additional comments. The goal of the following code is simply summing all elements of the input array.

**Algorithm 2.2**: Sum elements in array

```
class  SumForkJoin extends RecursiveTask<Long>{

    int  low;                                        ▷ Index where we need to start
    int  high;                                       ▷ Index where we need to end
    int []  array                                    ▷ Input array

    SumForkJoin(int[] ar,  int  lo,  int  hi){
        low = lo;
        high = hi;
        array  = ar;                                 ▷ Simple constructor
    }

    protected Long compute(){
        if(high − low <= CUTOFF){
            int  res  = 0;
            for( int  i  = low; i  < high;  i++){
                res+= array[i ];
            }
            return  res;                             ▷ Base case, compute sequentially
        }
        else{
            int  mid = low + (high − low)  / 2;
            SumForkJoin left = new SumForkJoin(array, left, mid);
            SumForkJoin right = new SumForkJoin(array, mid, right);
    ▷ Divide the problem in two subtasks
            left .fork () ;                          ▷ Spawn a new thread for the left subproblem
            long rightAns = right.compute();  ▷ The original thread goes on on
    the right side
            return  left .join ()  + rightAns;  ▷ When results are ready, compute
    the result
        }
    }
}
```

We end this short paragraph with some general concepts:

- It is highly inefficient to divide the problem until the trivial case where there is only a single element in the array. This holds because spawning the required number of threads for a short array is more expensive than computing the result sequentially. Use a cutoff with a value around 500-1000 in order to be efficient.
- Do not fork both subtasks. This creates a new thread for each subtask, although this is not required (the original thread can be reused).
- In order to start the computation declare a *ForkJoinPool* (which uses the number of processors as default parameter) and invoke it on the task. In this way the library will automatically manage the processors in order to parallelize your task efficiently.

## 2.4 A taste of parallel algorithms

In this section we explain an algorithm used for an elementary problem: adding two $n-$bit binary numbers, $a = a_n, a_{n-1}, \dots, a_1$ and $b = b_n, b_{n-1}, \dots, b_1$, where $a_i$ and $b_i$ denote the $i-$th least significant bits of $a$ and $b$ respectively. The goal is to output $a + b$, also in binary representation. How we will see, this algorithm provides some useful techniques which are used also for other problems, such as the *prefix sum problem* that you have discussed in the lecture.

A basic algorithm for computing the summation $a + b$ is *carry-ripple*. This is probably what most of us learned in the elementary school as "the way to compute summation". In the binary case, this algorithm works as follows: we compute the output bit $s_1$ by adding $a_1$ and $b_1$ (mod 2), but on the side we also produce a carry bit $c_1$, which is 1 iff $a_1 + b_1 \geq 2$. Then, we compute the second output bit $s_2$ based on $a_2$, $b_2$ and $c_1$, and on the side also produce a carry bit $c_2$. The process continues similarly. We compute $s_i$ as a (simple) function of $a_i$, $b_i$ and $c_{i-1}$, and then we also compute $c_i$ as a side-result, to be fed to the computation of the next bit of output. A shortcoming of the above algorithm is that computing each bit of the output needs to wait for the computation of the previous bit to be finished, and particularly for the carry bit to be determined. Even if many of your friends come to your help in computing this summation, it is not clear how to make the computation finish (significantly) faster. We next discuss an adaptation of the above algorithm, known as *carry look-ahead* that is much more parallelizable.

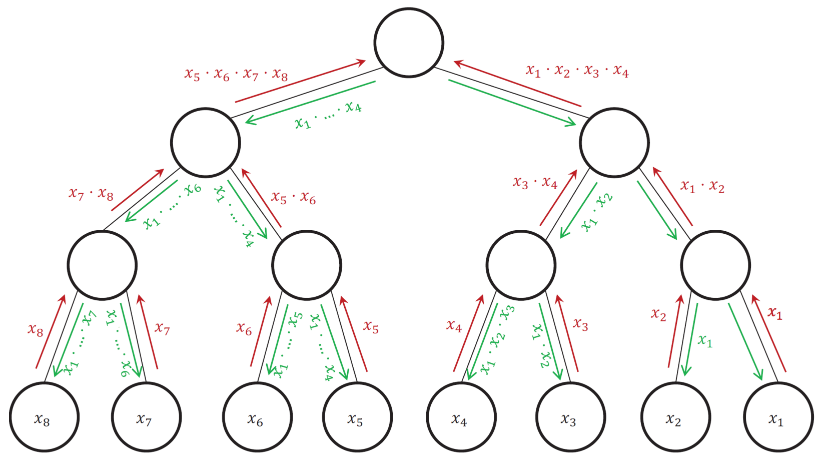
The only problematic part in the above process was the preparation of the carry bits. Once we can compute all the carry bits, we can complete the computation to find the output bit $s_i$ easily from $a_i$, $b_i$ and $c_{i-1}$. Moreover, the computations of different output bits $s_i$ are then independent of each other and can be performed in parallel .Se, we should find a better way to compute the carry bits $c_i$.

Let us re-examine the possibilities for $a_i$, $b_i$ and $c_{i-1}$ ans see what $c_i$ should be in each case. If $a_i = b_i = 1$, then $c_i = 1$; if $a_i = b_i = 0$ then $c_i = 0$, and if $a_i \neq b_i$, then $c_i = c_{i-1}$. Correspondingly, we can say that the carry bit is either *generated* (g), *killed* (k), or *propagated* (p). Given $a_i$ and $b_i$, we can easily determine an $x_i \in \{g, k, p\}$ which indicates in which of these three cases we are. Now, let us examine the impact of two consecutive bits, in the adder, on how the carry bit gets passed on. We can write the following simple multiplication table:

|   | k | p | g |
|---|---|---|---|
| k | k | k | k |
| p | k | p | g |
| g | g | g | g |

**Table 2.1:** Multiplication table

Let $y_0 = k$ and define $y_i \in \{k, p, g\}$ as $y_i = y_{i-1} \times x_i$, using the above multiplication table. Here $y_0$ indicates that there is no carry before the first bit. Once we compute $y_i$, we know the carry bit $c_i$: in particular $y_i = k$ implies $c_i = 0$, and $y_i = g$ means $c_i = 1$. Notice that we can never have $y_i = p$ because $y_0 = k$. So what remains is to compute $y_i$ for $i \in \{1, \dots, n\}$, given that $x_i \in \{k, p, g\}$ are known. Notice that each $x_i$ was calculated as a function of $a_i$ and $b_i$, and all in parallel. Computing $y_i$ is a simple task for parallelism and it is generally known as *Parallel Prefix*. We next explain a classic method for it: we build a full binary tree on the top of the indices $\{1, \dots, n\}$. then, on this tree, we pass up the product of all descendants, toward the root, in $\mathcal{O}(\log n)$ parallel steps. This way, each node in the binary tree knows the product of all $x_i$ in indices $i$ that are its descendants. Then, using $\mathcal{O}(\log n)$ extra parallel steps, each node passes to each of its children the product of all of the $x_i$ in the indices that are preceding the rightmost descendant of that child (pictorially, we are imagining the least significant bit in the rightmost part and the most significant part in the leftmost part). A the end, each leaf (index $i$) knows the product of all indices before itself and thus can compute $y_i$. We summarize this considerations in the following figure.



$x_5 \cdot x_6 \cdot x_7 \cdot x_8$
$x_1 \cdot x_2 \cdot x_3 \cdot x_4$
$x_1 \cdot \ldots \cdot x_4$
$x_7 \cdot x_8$
$x_1 \cdot \ldots \cdot x_6$
$x_1 \cdot \ldots \cdot x_4$
$x_5 \cdot x_6$
$x_3 \cdot x_4$
$x_1 \cdot x_2$
$x_1 \cdot x_2$
$x_8$
$x_1 \cdot \ldots \cdot x_7$
$x_1 \cdot \ldots \cdot x_6$
$x_7$
$x_6$
$x_1 \cdot \ldots \cdot x_5$
$x_1 \cdot \ldots \cdot x_4$
$x_5$
$x_4$
$x_1 \cdot x_2 \cdot x_3$
$x_1 \cdot x_2$
$x_3$
$x_2$
$x_1$
$x_1$
$x_8$
$x_7$
$x_6$
$x_5$
$x_4$
$x_3$
$x_2$
$x_1$

# Concurrency 3

## 3.1 A Teaser

Instead of treating coordination problems as programming exercises, we begin with an analogy as a physical problem. We now present a short story that provides insights with some of the most important concepts of this chapter.

Alice and Bob are neighbors, and they share a yard. Alice owns a cat and Bob owns a dog. Both pets like to run around in the yard, but (naturally) they do not get along. After some unfortunate experiences, Alice and Bob agree that they should coordinate to make sure that both pets are never in the yard at the same time. Of course, we rule out trivial solutions that do not allow any animals into an empty yard. How should they do it? Alice and Bob need to agree on mutually compatible procedures for deciding what to do. We call such an agreement a *coordination protocol*. The yard is large, so Alice cannot simply look out of the window and check whether Bob's dog is present. She could perhaps walk over to Bob's house and knock on the door, but that takes a long time. Alice might lean out the window and shout *Hey Bob! Can I let the cat out?* The problem is that Bob might no hear her. They could try to coordinate by cell phone, but maybe Bob is taking a shower. Alice has a clever idea. Each one sets up a flag pole, easily visible to the other. When Alice wants to release her cat, she does the following:

1. She raises her flag
2. When Bob's flag is lowered, she unleashes her cat
3. When her cat comes back, she lowers her flag

Bob's behaviour is a little bit more complicated:

- He raises his flag
- While Alice's flag is raised
  - (a) Bob lowers his flag
  - (b) Bob waits until Alice's flag is lowered
  - (c) Bob raises his flag
- As soon as his flag is raised and hers is down, he unleashes his dog
- When his dog comes back, he lowers his flag

This protocol rewards further study as a solution to Alice and Bob's problem. On an intuitive level, it works because of the following *flag principle*. If Alice and Bob each

1. Raises his or her own flag, and then
2. Looks at the other's flag

then at least one will see the other's flag raise (clearly, the last one to look will see the other's flag raised) and will not let his or her pet enter the yard. However, this observation does not *prove* that the pets will never be in the yard together. What if, for example, Alice lets her cat in and out of the yard several times while Bob is looking? To prove that the pets will never be in the yard together, assume by way of contradiction that there is a way the pets could end up in the yard together. Consider the last time Alice and Bob each raised their flag and looked at the other's flag before sending the pet into the yard. When Alice last looked, her flag was already fully raised. She must have not seen Bob's flag, or she would not have released the cat, so Bob must have not completed raising his flag before Alice started looking. It follows that when Bob looked for the last time, after raising his flag, it must have been after Alice started looking, so he must have seen Alice's flag raised and would not have released his dog, a contradiction.

To show that the flag protocol is a correct solution of Alice and Bob's problem, we must understand what properties are required of a solution, and then show that they are met by the protocol. First, we proved that the pets are excluded from being in the yard at the same time a fundamental property that we call *mutual exclusion*. Mutual exclusion in only one of several properties of interest. After all, as we noted earlier, a protocol in which Alice and Bob never release a pet satisfies the mutual exclusion property, but it is unlikely to satisfy their pets. Here is another property of central importance. First, if one pet wants to enter the yard, then it eventually succeeds. Second, if both pets want to enter the yard, then eventually at least one of them succeeds. We consider this *deadlock-freedom* property to be essential. We claim that Alice and Bob's protocol is deadlock-free. Suppose both pets want to use the yard. Alice and Bob each raise their flags. Bob eventually notices that Alice's flag is raised, and defers to her by lowering his flag, allowing her cat into the yard. Another property of compelling interest is *starvation-freedom*: if a pet wants to enter the yard, will it eventually succeed? Here, Alice and Bob's protocol performs poorly. Whenever Alice and Bob are in conflict, Bob defers to Alice, so it is possible that Alice's cat can use the yard over and over again, while Bob's dog becomes increasingly uncomfortable. The last property of interest in this example concerns *waiting*. Imagine that Alice raises her flag, and is then suddenly stricken with appendicitis. She (and the cat) are taken to the hospital, and after a successful operation, she spends the next week under observation at the hospital. Although Bob is relieved that Alice is well, his dog cannot use the yard for an entire week until Alice returns. The problem is that the protocol states that Bob (and his dog) must *wait* for Alice to lower her flag. If Alice is delayed, then Bob is also delayed (for apparently no reason).

Having reviewed both the strengths and weaknesses of Bob and Alice's protocols, we now turn our attention back to Computer Science.

In the introductory chapter we saw how multiple threads accessing the same objects can lead to incorrect or undesirable executions and dubbed such cases as *race conditions*. In this chapter we focus on the crucial aspect of protecting access to a mutable memory location that is shared among different threads. We first formalize the discussion of the previous example by defining mutual exclusion and other associated properties in a rigorous way. We'll then see different ways to implement mutual exclusion: locks (with their low level implementations and some high-level synchronization primitives such as semaphores and barriers), atomic operations (to implement spinlocks and the so called *lock free* data structures) and transactional memory (a model which is in some sense more optimistic than locking and is likely to play an important role in the future). Let's begin this fascinating journey!

## 3.2 Mutual exclusion

For all mutable shared memory locations, the programmer must ensure that no bad interleaving or data race occurs by implementing *mutual exclusion*. This is done by defining *critical sections*, *i.e.* sections of code which only one thread at a time is allowed to execute. We will now first define progress conditions and use these to explain the requirements for the correct implementation of a critical section. We will then turn our attention to finite state diagrams, which will allow us to formally define whether all these requirements hold.

When talking about concurrent algorithms, we distinguish between *blocking* and *non-blocking algorithms*. As one might think, in blocking algorithms threads might occasionally go into a blocked state, *e.g.* when attempting to acquire a lock. Among blocking algorithms, we distinguish further cases:

- ▶ Deadlock-free: at least one thread is guaranteed to proceed into the critical section at some point
- ▶ Starvation-free: all threads are guaranteed to proceed into the critical section at some point

In non-blocking algorithms, threads never enter a blocked state, *i.e.* they can always continue their execution. This mainly suggests that the algorithms do not use any locks. We again distinguish two further cases:

- ▶ Lock-free: at least one thread always makes progress
- ▶ Wait-free: all threads make progress within a finite amount of time

When comparing the different definitions listed above, we notice a few things. We see that lock-freedom and starvation-freedom both imply deadlock freedom. We further notice that wait-freedom implies both lock-freedom and starvation-freedom. We summarize the different conditions in the following table:

| | Blocking | Non-Blocking |
|---|---|---|
| Someone makes progress | Deadlock-free | Lock-free |
| Everyone makes progress | Starvation-free | Wait-free |

Of course, it is perfectly possible that an algorithm fits into none of these categories, *e.g.* when there is an execution that results in a deadlock.

The mutual exclusion property is clearly essential. Without this property, we cannot guarantee that a computation's result are correct. Mutual exclusion is a *safety property*, *i.e.* it assures that "something bad never happens". The deadlock-freedom property is important. It implies that the system never freezes. Individual threads may be stuck forever (starvation), but some threads makes progress. Deadlock-freedom is a liveness property. The starvation-freedom property, while clearly desirable, is the least compelling of the three. Later on, we will see practical mutual exclusion algorithms that fail to be starvation-free. These algorithms are typically deployed in circumstances where starvation is a theoretical possibility, but it is unlikely to occur in practice. Nevertheless, the ability to reason about starvation is essential for understanding whether it is a realistic threat. The starvation-freedom property is also weak in the sense that there is no guarantee for how long a thread waits before it enters the critical section.

As a final note, it is important to mention *livelocks*. A livelock occurs when all or at least some threads are changing state, but none of them actually enters the critical section. One might think of this visually as two people in a narrow hallway continuously trying to get past each other by stepping aside, but always stepping in the same direction. Note that a livelock does not mean that there is no possible execution which results in a thread entering the critical section, but simply that there is the possibility of the threads returning to the same state without any thread entering the critical section.

**Example 3.2.1** Consider the following class *BooleanFlags* that increments a counter 1000 times.

**Algorithm 3.1**: A wrong implementation of mutual exclusion

```
public class BooleanFlags extends Thread{
    static int cnt = 0;
    static boolean flag = false;

    public void run(){
        for(int i = 0; i < 1000; i++){
            while(flag != false);
            flag = true;
            cnt++;
            flag = false;
        }
    }
}
```

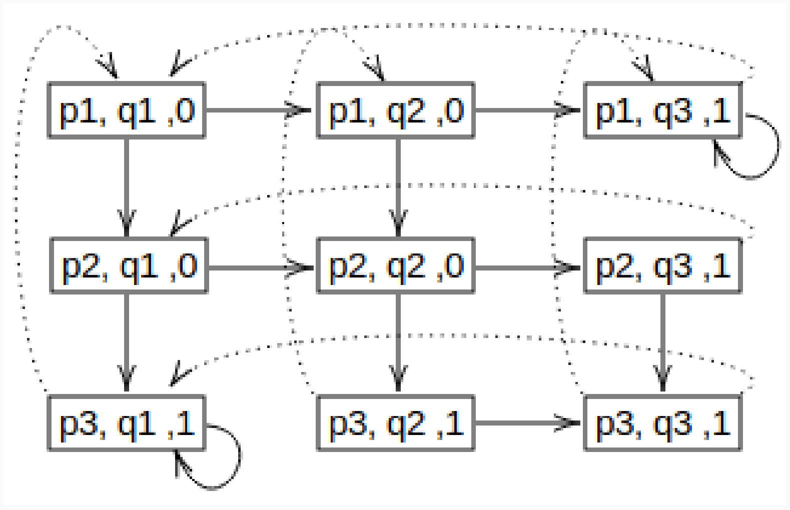
When running the program with two threads, the counter turns out to be unexpectedly low. Looking at the implementation, we see that the critical section is between the *flag* = *true* and *flag* = *false* statements. The problem here is that there is a possible execution of the two threads, namely when both read *flag* == *false* before either of them could set *flag* = *true*, where both threads enter the critical section. Therefore, the requirement of mutual exclusiveness is violated and this implementation is incorrect.

A useful tool to realise whether the properties of mutual exclusion, deadlock-freedom and starvation-freedom are satisfied are *state diagrams*. An execution state diagram visually represents the different states and state transitions between them. A program state is determined by the instructions the threads are executing and the states of global variables and concurrent objects. A state transition is then simply represented by arrows between such boxes. Using state transition diagrams, it is easy to determine whether the critical section is implemented correctly. Deadlocks can be recognized by states which have no outgoing state transitions. Livelocks are any possible cycle of state transitions in which in none of the states a threads is in the critical section. The critical section is mutually exclusive if there is no state in which more than one thread is in the critical section.

**Example 3.2.2** We continue the previous example and construct the state diagram corresponding to the given code snippet, were we restrict ourselves to two threads for simplicity. We represent a state by a tuple $(p_i, q_j, b)$ for $i, j \in \{1, 2, 3\}$ and $b \in \{0, 1\}$. $p_i$ (respectively $q_j$) represents the instruction the corresponding thread is about to execute. The represented instructions are:

- $i = 1$: *while(flag!=false);*
- $i = 2$: *flag=true;*
- $i = 3$: *cnt++;*

$b$ represents the value of *flag*.



We see that there is a path of state transitions leading to a state in which both threads are in the critical section. Therefore, the code does not provide mutual exclusion.

**Example 3.2.3** (Dining Philosophers) The Dining Philosophers Problem was proposed by Dijkstra in 1965. The problem consists of a table with five plates, five forks (or chopsticks) and a big bowl of spaghetti. Five philosophers, do nothing but think for a certain amount of time and eat a very difficult kind of spaghetti which requires two forks to eat. You can think of a philosopher as a thread, executing the following piece of pseudo-code:

```
while(true){
think();
acquire_fork_on_left_side();
acquire_fork_on_right_side();
eat();
release_fork_on_right_side();
release_fork_on_left_side();
```

**Algorithm 3.2**: Dining Philosophers

```
}
```

Assuming that the philosophers know how to think and eat, the methods to pick up the forks and put down the forks again need to satisfy the following constraints:

- Only one philosopher can hold a fork at a time and is permitted to use only the forks to his/ her immediate right and left
- A philosopher needs to acquire both forks to his left and right before he can start to eat
- It must be impossible for a deadlock to occur
- It must be impossible for a philosopher to starve waiting for a fork
- It must be impossible for more than one philosopher to eat at the same time

With a naive implementation there is a possibility to deadlock. In the first step, each philosopher acquires the fork on their left side at the same time. Now that every philosopher has one fork in their left hand, they will try to pick up the fork on their right side, but this fork is already held by their neighbor. In this state, everyone waits for their neighbor on the right to release the fork, but since nobody has acquired both forks to eat, this will never happen and the philosophers will starve. The problem is that there is a cyclic dependency in the case described above: P1 waits for P5, P5 waits for P4, P4 waits for P3, P3 waits for P2, and P2 waits for P1. How can we make deadlocks impossible? The correct solution is to break the cyclic dependency. This means that one philosopher, e.g. philosopher P5 has to pick up the forks in a different order. This could be achieved by introducing a order between the forks, and all the philosophers have to pick up the fork with the lower order first. This resolves the deadlock, because there are no cyclic dependencies any more. Note that introducing a timeout will not solve the problem: Even if the philosophers would release the fork after some time of waiting, it could still happen that they would do this at the same time. This would be a livelock, where all philosophers would constantly acquire and release the fork on their left, but nobody would get to eat.An alternative solution to avoid deadlocks while achieving the mutual exclusion property would be using *exponential backoff*. In this scenario a philosopher tries to take the forks and if it fails he sleeps for a random amount of time in the interval $(0, s)$. The second time that he fails he will sleep for a random amount of time in the interval $(0, 2s)$ and this time doubles for every successive failure. While this approach does not guarantees deadlock freedom (if we are very unlucky, a deadlock could still occur), it is a heuristic known to work very well in practice.

## 3.3 Mutex Implementation

In the previous section we discussed the importance of achieving mutual exclusion in a multithreaded environment: we must be sure that a mutable memory location which is shared by many threads will access only by a single thread at a time. Bot *how* can we actually achieve mutual exclusion (without forgetting other important properties such as deadlock freedom)? In the following we will declare some variables as *volatile*. This means that accesses to those variables do not count as data races. For more details see the dedicated section in Chapter 4.

## Peterson Lock

In the following, we assume all reads and writes are atomic. This assumption is necessary, as while we do add the *volatile* keyword to all array declarations, this only concerns the array references, not their entries. In practice, one would declare the arrays to be arrays of *AtomicIntegers*, i.e. as *AtomicIntegerArray*. This algorithm is arguably the most succint and elegant mutual exclusion algorithm. It has the big advantage of being starvation-free, but it has the drawback to work only to provide mutual exclusion to two threads.

**Algorithm 3.3**: Peterson Lock

```
class PetersonLock{
    private volatile boolean[] flag = new boolean[2];
    public volatile int victim;
    public void lock(int id){
        flag[id] = true;
        victim = id;
        while(flag[1-id] && victim == id){}
    }
    public void unlock(int id){
        flag[id] = false
    }
}
```

In a nutshell, the idea of the algorithm is the following:

- Set out flag, thereby indicating that we're interested in entering the critical section
- Indicate that the other thread is allowed to go first. The thread that arrives at this statement first will enter the critical section first
- Wait until the other thread is either no longer interested in entering the critical section or until we're allowed to go first
- Indicate that we're no longer interested

**Theorem 3.3.1** *The Peterson lock satisfies mutual exclusion*

*Proof.* Suppose not. Consider the last executions of the *lock()* method by threads $A$ and $B$. Inspecting the code we see that

$$\text{write}_A(\text{flag}[A] = true) \longrightarrow \text{write}_A(\text{victim} = A) \longrightarrow \text{read}_A(\text{flag}[B]) \longrightarrow \text{read}_A(\text{victim}) \longrightarrow \text{CS}_A$$

and

$$\text{write}_B(\text{flag}[B] = true) \longrightarrow \text{write}_B(\text{victim} = B) \longrightarrow \text{read}_B(\text{flag}[A]) \longrightarrow \text{read}_B(\text{victim}) \longrightarrow \text{CS}_B$$

Assume, without loss of generality, that $A$ was the last thread to write to the *victim* field

$$\text{write}_B(\text{victim} = B) \longrightarrow \text{write}_A(\text{victim} = A)$$

This implies that $A$ observed *victim* to be $A$. Since $A$ nevertheless entered its critical section, it must have observed the flag of $B$ to be false, so we have

$$\text{write}_A(\text{victim} = A) \longrightarrow \text{read}_A(\text{flag}[B] == false)$$

And with the transitivity of $\longrightarrow$ we get

$$\text{write}_B(\text{flag}[B] = true) \longrightarrow \text{write}_B(\text{victim} = B) \longrightarrow \text{write}_A(\text{victim} = A) \longrightarrow \text{read}_A(\text{flag}[B] == false)$$

It follows that $\text{write}_B(\text{flag}[B] - true) \longrightarrow \text{read}_A(\text{flag}[B] == false)$. This observation yields a contradiction because no other write to the flag of $B$ was performed before the critical section executions. □

**Theorem 3.3.2** *The Peterson lock is starvation free.*

*Proof.* Suppose not. Suppose (without loss of generality) that thread $A$ runs forever in the *lock()* method. It must be executing the *while* statement, waiting for the flag of thread $B$ to become false or the victim to be set to $B$. What is $B$ doing while $A$ fails to make progress? Perhaps $B$ is repeatedly entering and leaving its critical section. If so, however, then $B$ sets *victim* to $B$ as soon as it reenters the critical section. Once *victim* is set to $B$, it does not change, and $A$ must eventually return from the *lock()* method, a contradiction. So it must be that $B$ is also stuck in the *lock()* method call, waiting until either the flag of $A$ becomes false or *victim* is set to $A$.But *victim* cannot be both $A$ and $B$, a contradiction. □

Note that since we proved that Peterson lock is starvation free, this automatically implies deadlock-freedom.

## Filter Lock

We now consider the first mutual exclusion protocol that work for $n$ threads, where $n$ is greater than two: the *filter lock*, which is a direct generalization of the Peterson lock to multiple threads. The filter lock creates $n-1$ "waiting rooms", that a thread must traverse before acquiring the lock. Levels satisfy two important properties:

- At least one threads trying to enter level $l$ succeeds
- If more than one thread is trying to enter level $l$, then at least one is blocked, *i.e.* it continues to wait at that level.

**Algorithm 3.4**: Filter Lock

```
class FilterLock{
    volatile int[] level;
    volatile int[] victim;
    volatile int n;
   public FilterLock(int n){
       this.n = n;
       level = new int[n];
       victim = new int[n];
       for(int i = 0; i < n; i++){
           level[i] = 0;
11          }
12      }
13      public void lock() {
14          int  me = ThreadID.get();
15          for(int  i  = 1;  i  < n;  i++){
16              level [me] = i;
17              victim[i]  = me;
18              while((∃k!=me)(level[k]>=i && victim[i]==me){}
19          }
20      }
21      public void unlock(){
22          int  me = ThreadID.get();
23          level [me] = 0;
24      }
25  }
```

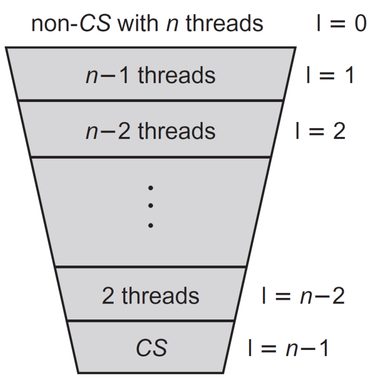


**Figure 3.1:** There are $n-1$ levels threads pass through, he last of which is the critical section. There are at most $n$ threads that pass concurrently into level 0, $n-1$ into level 1 and son on, that only one enters the critical section at level $n-1$.

The Peterson Lock uses a two element boolean *flag* array to indicate whether a thread is trying to enter the critical section. The Filter Lock generalizes this notion with an $n-$element integer *level* array, where the value of *level[A]* indicates the highest level that thread $A$ is trying to enter. Each thread must pass through $n-1$ levels of "exclusion" to enter its critical section. Each level $l$ has a distinct *victim* field used to "filter out" one thread, excluding it from the next level. Initially a thread $A$ is at level zero. We say that $A$ is at *level* $j$ for $j > 0$, when it last completes the waiting loop in line 18 with *level[A]*$\geq j$. So a thread at level $j$ is also at level $j-1$ an so on.

**Theorem 3.3.3** *For $j$ between 0 and $n-1$, there are at most $n-j$ threads at level $j$.*

*Proof.* By induction on $j$. The base case, where j = 0, is trivial. For the induction step, the induction hypothesis implies that there are at most $n-j+1$ threads at level $j-1$ To show that at least one thread cannot progress to level j, we argue by contradiction: assume there are $n-j+1$ threads at level $j$. Let $A$ be the last thread at level $j$ to write to *victim[j]*. Because $A$ is last, for any other $B$ at level $j$ we have

$$\text{write}_B(\text{victim}[j]) \rightarrow \text{write}_A(\text{victim}[j])$$

Inspecting the code, we see that $B$ writes *level[B]* before it writes to *victim[j]*, so

$$\text{write}_B(\text{level}[B] = j) \rightarrow \text{write}_B(\text{victim}[j]) \rightarrow \text{write}_A(\text{victim}[j]) \rightarrow \text{read}_A(\text{level}[B])$$

Because $B$ is at level $j$, every time $A$ reads *level[B]*, it observes a value greater than or equal to $j$, implying that $A$ could not have completed its waiting loop in line 18, a contradiction. □

By observing the statement of the previous theorem, it is immediate to realise that the Filter Lock algorithm satisfies mutual exclusion.

**Theorem 3.3.4** *The Filter Lock is starvation-free.*

*Proof.* We argue by reverse induction on the levels. The base case, level $n-1$ is trivial, because it contains at the most one thread. For the induction hypothesis, assume that every thread that reaches level $j+1$ or higher, eventually enters (and leaves) its critical section. Suppose $A$ is stuck at level $j$. Eventually, by the induction hypothesis, there are no threads at higher levels. Once $A$ sets *level[A]* to $j$, then any thread at level $j-1$ that subsequently reads *level[A]* is prevented from entering level $j$. Eventually, no more threads enter level $j$ from lower levels. All threads stuck at level $j$ are in the waiting loop at Line 18, and the values of the victim and level fields no longer change. We now argue by induction on the number of threads stuck at level $j$. For the base case, if $A$ is the only thread at level $j$ or higher, then clearly it will enter level $j+1$. For the induction hypothesis, we assume that fewer than $k$ threads cannot be stuck at level $j$. Suppose threads $A$ and $B$ are stuck at level $j$. $A$ is stuck as long as it reads *victim[j]* $= A$, and $B$ is stuck as long as it reads *victim[j]* $= B$. The victim field is unchanging, and it cannot be equal to both $A$ and $B$, so one of the two threads will enter level $j+1$, reducing the number of stuck threads to $k-1$, contradicting the induction hypothesis. □

Note that since we proved that Filter lock is starvation free, this automatically implies deadlock-freedom.

The starvation-freedom property guarantees that every thread that calls *lock()* eventually enters the critical section, but it makes no guarantees about how long this may take. Ideally (and very informally) if $A$ calls *lock()* before $B$, then $A$ should enter the critical section before $B$. Unfortunately, using the tools at hand we cannot determine which thread called *lock()* first. Instead, we split the *lock()* method into two sections of code.

1. A *doorway* section, whose execution interval $D_A$ consists of a bounded number of steps
2. A *waiting* section, whose execution interval $W_A$ may take an unbounded number of steps.

The requirement that the doorway section always finish in a bounded number of steps is a strong requirement. Here is how we define *fairness*.

**Definition 3.3.1** (Fairness) *A lock is* first-come-first-served *if, whenever, thread A finishes its doorway before thread B starts its doorway, then A* *cannot be overtaken by B: if* $D_A^j \rightarrow D_B^k$ *then* $CS_A^j \rightarrow CS_B^k$ *for threads A and B and integers j and k.*

## Bakery Algorithm

Until now we have seen the Peterson Lock (which implements a starvation-free protocol for mutual exclusion in the limited case of two threads) and the Filter Lock (which generalizes the Peterson Lock to $n$ threads). A drawback of the Filter Lock is that is not fair. Now let's take a loot at the following algorithm, known as *Lamport's Bakery Algorithm*.

**Algorithm 3.5**: Filter Lock

```
1  class Bakery implements Lock{
2      boolean[] flag;
3      label [] label;
4      public Bakery(int n){
5          flag = new boolean[n];
6          label = new label[n];
7          for(int i = 0; i < n; i++){
8              flag[i] = false;
9              label[i] = 0;
10         }
11     }
12     public void lock(){
13         int i = ThreadID.get();
14         flag[i] = true;
15         label[i] = max(label[0], ..., label[n−1])+1;
16         while(∃ k != 1 with (flag[k] && (label[k],k) << (label[i], i))
       ){}
17     }
18     public void unlock(){
19         flag[ThreadID.get()] = false;
20     }
21 }
```

The Bakery lock algorithm maintains the first-come-first- served property by using a distributed version of the number-dispensing machines often found in bakeries: each thread takes a number in the doorway, and then waits until no thread with an earlier number is trying to enter it. In the Bakery lock, *flag[A]* is a Boolean flag indicating whether $A$ wants to enter the critical section, and *label[A]* is an integer that indicates the thread's relative order when entering the bakery, for each thread $A$. Each time a thread acquires a lock, it generates a new *label[]* in two steps. First, it reads all the other threads' labels in any order. Second, it reads all the other threads' labels one after the other (this can be done in some arbitrary order) and generates a label greater by one than the maximal label it read. We call the code from the raising of the flag (Line 14) to the writing of the new *label[]* (Line 15) the doorway. It establishes that thread's order with respect to the other threads trying to acquire the lock. If two threads execute their doorways concurrently, they may read the same maximal label and pick the same new label. To break this symmetry, the algorithm uses a lexicographical ordering « on pairs of label[] and thread ids. In the waiting part of the Bakery algorithm

(Line 16), a thread repeatedly rereads the labels one after the other in some arbitrary order until it determines that no thread with a raised flag has a lexicographically smaller label/id pair. Since releasing a lock does not reset the *label[]*, it is easy to see that each thread's labels are strictly increasing. Interestingly, in both the doorway and waiting sections, threads read the labels asynchronously and in an arbitrary order, so that the set of labels seen prior to picking a new one may have never existed in memory at the same time. Nevertheless, the algorithm works.

**Theorem 3.3.5** *The Bakery Lock is deadlock free.*

*Proof.* Some waiting thread $A$ has the unique least *(label[A],A)* pair, and that thread never waits for another thread. □

**Theorem 3.3.6** *The Bakery Lock algorithm is first-come-first-served.*

*Proof.* If $A$'s doorway preeceds $B$'s $D_A \to D_B$ then $A$'s label is smaller since

$$\text{write}_A(\text{label}[A]) \to \text{read}_B(\text{label}[A]) \to \text{write}_B(\text{label}[B]) \to \text{read}_B(\text{flag}[A])$$

so $B$ is locked out while *flag[A]* is true. □

Note that any algorithm that is both deadlock-free and first-come-first-served is also starvation-free.

**Theorem 3.3.7** *The Bakery algorithm satisfies mutual exclusion.*

*Proof.* Suppose not. Let $A$ and $B$ be two threads concurrently in the critical section. Let $\text{labeling}_A$ and $\text{labeling}_B$ be the last respective sequences of acquiring new labels prior to entering the critical section. Suppose that *(label[A],A) «(label[B],B)*. When $B$ successfully completed the test in its waiting section, it must have read that *flag[A]* was false or that *(label[B],B) « (label[A],A)*. However, for a given thread, its id is fixed and its *label[]* values are strictly increasing, so $B$ must have seen that *flag[A]* was false. It follows that

$$\text{labeling}_B \to \text{read}_B(\text{flag}[A]) \to \text{write}_A(\text{flag}[A]) \to \text{labeling}_A$$

which contradicts the assumption that $(\text{label}[A], A) << (\text{label}[B], B)$ □

## Spinlock

When implementing mutual exclusion, there are two different alternative choices on what to do when we cannot immediately acquire a lock. The first choice is to continue trying to acquire the lock. This is called *spinning* or *busy waiting*. The Filter lock and the bakery lock are two examples. The second choice is called *blocking* and essentially means that if a thread cannot acquire the lock it gets de-scheduled by the operating system (it goes to sleep) and is woken up after a certain time. Which strategy should we choose? In general a pure spinlock only makes sense if the duration that any process holds the lock is short, otherwise, it's better to block. It is also possible to come up with a mixed strategy, which begins with spinning and, if it unable to acquire the lock for a certain amount of time, it blocks. Here is where the *spin-block problem* come into play: we want to come up with a strategy for how long a thread should spin waiting to acquire a lock before giving up and blocking, given particular values for the cost of blocking, and the probability distribution of lock hold times. The *competitive spinning theorem* tells that in absence of any other information about the lock hold time, spinning for a time equal to the cost of putting the thread to sleep and waking it up again results in overhead at most twice that the optimal offline algorithm (which has perfect knowledge about the future) can do.

Unfortunately, our previous lock implementations (although we proved them to logically work) won't work on most modern processors and compilers.This is because the compiler and the underlying hardware architecture do not guarantee memory operations to occur in-order. We tried to somewhat alleviate this issue by introducing the *volatile* keyword, but this only guarantees reads and writes to the array reference to be in-order, not to the entries of this array. We therefore introduce mutex implementations using two atomic operations: TAS and CAS (which will be discussed in greater detail in the further section about atomic operations). With the use of those atomic operations we obtain the following locks:

**Algorithm 3.6**: TAS Lock

```
class TASLock{
    AtomicBoolean state = new AtomicBoolean(false);
    public void lock(){
        while(state.getAndSet(true)){}
    }
    public void unlock(){
        state.set(false);
    }
}
```

**Algorithm 3.7**: TTAS Lock

```
class TTASLock{
    AtomicBoolean state = new AtomicBoolean(false);
    public void lock(){
        do{
            while(state.get()){}
        }while(!state.compareAndSet(false, true));
    }
    public void unlock(){
        state.set(false);
    }
}
```

Both implementations are correct, but unfortunately they perform poorly. The explanation for this behaviour can be found in modern hardware architecture:

- Calls to *getAndSet()* and *set()* force other processors to invalidate their cached copies of the state variable. This means that the next call to *get()* and *getAndSet()* will need to read from main memory.

Continuing on like this, it's easy to see that nearly each call will read from main memory.
- ▶ Nearly each call reading from main memory also means that the shared bus will be under heavy use.This means that all threads using the bus will be slowed down considerably.

One possibility to alleviate this problem would be to implement an exponential backoff. This means that every time we don't manage to acquire the lock, we wait for a random amount of time $m \in [0, t^x]$, where $x$ is the number of times we have failed to acquire the lock and $t$ is the initial wait time.

## 3.4 Locks: an high level perspective

In the second section of this chapter we introduced the concept of mutual exclusion and we argued that a correct implementation of a mutual exclusion algorithm has to be *deadlock-free*. Before we dive into more details about how locks can be used in practice, we take a more formal look into deadlocks. There are two purposes of doing this: first, deadlocks are one of the major (and worst) issues in parallel programs; second, it can aware the reader to the fact that one has to think about the role of locks in his code. Just putting the keyword *synchronized* somewhere can have disastrous effects. Before we set an example, we recall that a deadlock occurs when two or more processes are mutually blocked because each process waits for another of these processes to proceed.

**Example 3.4.1** (Bank account) Consider a method to transfer money between bank accounts.

**Algorithm 3.8**: Bank Account

```
class BankAccount{
    synchronized void withdraw(int amount){...}
    synchronized void deposit(int amount){...}
    synchronized void transferTo(int amount, BankAccount a){
        this.withdraw(amount);
        a.deposit(amount);
    }
}
```

The problem in this code happens in the scenario of $x$ that wants to transfer a certain amount to $y$ and, at the same time, also $y$ wants to transfer money to $y$. In this case it could happen that the bank account $x$ first acquires the lock for $x$ and then the bank account $y$ acquires the lock for $y$. If this happens we have a deadlock: $x$ is waiting for a resource held by $y$ (*i.e.* the lock for $y$) and $y$ is waiting for a resource held by $x$ (*i.e.* the lock for $x$).

How can we formally reason about this scenario? We can represent the situation visually. In our visual representation we have a graph $G = (V, E)$ where $V$ is the set of threads union the set of resources (in this case locks) and the directed edges connect a thread to a resource if a thread attempts to acquire that resource, and a resource to a thread if the resource is held by the thread. We have a deadlock for threads

$T_1, \dots, T_n$ if the graph describing the relation of $T_1, \dots, T_n$ and resources $R_1, \dots, R_m$ contains a cycle.

Now that we have understood how to formally detect a deadlock, we have to find a way to avoid it. Introducing a global lock would not be a good solution because it would lead back to the sequential execution. What one should do is *introducing a global ordering of resources*. An algorithm for the situation of the previous example would be simply to use an ID for each single account and lock accordingly, as shown in the following piece of code.

**Algorithm 3.9**: Bank Account

```
class BankAccount{
    synchronized void withdraw(int amount){...}
    synchronized void deposit(int amount){...}
    synchronized void transferTo(int amount, BankAccount a){
        if(to.accountID<this.accountID){
            syncrhonized(this){
                syncrhonized(to){
                    this.whitdraw(amount);
                    to.deposit(amount);
                }
            }
        } else{
            synchronized(to){
                synchronized(this){
                    this.withdraw(amount);
                    to.deposit(amount);
                }
            }
        }
    }
}
```

When a thread encounters the *synchronized* keyword, it will always first attempt to obtain the lock to the specified object. Until it obtains the lock, it will block. Every Java object, including classes itself, not just their instances, has an associated lock, meaning that we can use it to enforce mutual exclusion using *synchronized*. We can also include the *synchronized* in the method signature. This means that before proceeding with execution of the method body, we will first acquire the lock on the *this* object, which is either an instance of the class or the class itself if the method is declared *static*. Note that Java locks are reentrant, *i.e.* they can be acquired multiple times by the same thread. When creating a concurrent program with mutable, shared state, we think in terms of what operations need to be atomic. Locks do pretty much just that: Changes made inside of a *synchronized* block appear to other threads (provided they also acquire the lock before reading mutable, shared fields) to take place instantaneously. Nonetheless, locks are not all rainbows and sunshine. When we have large critical sections which are all protected by locks, we reduce the parallelizable fraction in our program by a lot. Thinking back to Amdahl's Law, we know that this drastically reduces the possible speedup of our program. In the remainder of this section we will see different ways to use locks properly. In the next section, instead, we will focus on *lock granularity*, *i.e.* we will reflect about the efficiency implications of our choices.

## Semaphores

As we have seen, a mutual exclusion lock guarantees that only one thread at a time can enter a critical section. If another thread wants to enter the critical section while it is occupied, then it blocks, suspending itself until another thread notifies it to try again. A *Semaphore* is a generalization of mutual exclusion locks. Each Semaphore has a capacity, denoted by $c$ for brevity. Instead of allowing only one thread at a time into the critical section, a semaphore allows at most $c$ threads, where the capacity $c$ is determined when the semaphore is initialized. The Semaphore class of the next code snippet provides two methods: a thread calls *acquire()* to request permission to enter the critical section, and *release()* to announce that it is leaving the critical section. The Semaphore itself is just a counter: it keeps track of the number of threads that have been granted permission to enter. If a new *acquire()* call is about to exceed the capacity $c$, the calling thread is suspended until there is room.When a thread leaves the critical section, it calls *release()* to notify a waiting thread that there is now room.

```
public class Semaphore{
    int capacity;
    int state;
    Lock lock;
    Condition condition;
    public Semaphore(int c){
        capacity = c;
        state = 0;
        lock = new ReentrantLock():
        condition = lock.newCondition();
    }
    public void acquire(){
        lock.lock();
        try{
            while(state == capacity){
                condition.await();
            }
            state++;
        } finally {lock.unlock(); }
    }
    public void release(){
        lock.lock();
        try{
            state--;
            condition.signalAll();
        } finally {lock.unlock(); }
    }
}
```

**Algorithm 3.10**: Semaphore Implementation

## Barriers

We now want to go one step further. We wish to create a barrier which blocks all threads up until a certain threshold $N$. Once the threshold has been reached all threads are allowed to continue execution. We distinguish between non-reusable and reusable barriers.

The non-reusable barrier has a relatively simple implementation, which can be seen in the next code snippet. Each arriving thread increments the counter (taking the lock to avoid race conditions) and attempts to acquire the *barrier* semaphore, thereby going into a blocked state. Once all threads have arrived, i.e. *count* == *threshold*, the *barrier* semaphore is set to 1, allowing one thread to pass. Whenever a thread passes the barrier it immediately releases it again so that the next thread may proceed with its execution. This method of acquiring and releasing the *barrier* semaphore is called a *turnstyle*.

```
class SimpleBarrier{
    private int threshold;
    private int count = 0;
    private Semaphore barrier = new Semaphore(0);
    public SimpleBarrier(int threshold){
        this.threshold = threshold;
    }
    public void await(){
        synchronized(this){
            count++;
        }
        if(count == threshold) barrier.release();
        try{
            barrier.acquire();
        }catch(InterruptedException e){}
        barrier.release();
    }
}
```

The reusable barrier is implemented in such a way that it can be reused once all waiting threads are released, i.e. it assumes no additional threads call the *await()* method before all waiting threads have been released.
The reusable barrier consists of two parts. First, the threads increment the counter, attempt to acquire the *barrier1* semaphore and go into a blocking state. Once the threshold is reached the *barrier1* semaphore is incremented so that the threads are allowed to pass the turnstyle and enter the second part of the barrier. Additionally, the *barrier2* semaphore is decremented, thereby making the second part of the barrier behave identically to the first part, only now decreasing the counter. In the second part of the barrier, threads will be allowed to pass the turnstyle once the counter's value is 0.Note that once all threads have exited the barrier, all values have been restored to their original state, thereby allowing the barrier to be reused again.

## Producer Consumer Pattern

In this paragraph we study the *Producer Consumer Pattern*, a fundamental parallel programming pattern that can be used to build data-flow parallel programs. We actually have already encountered such a pattern in our semaphore implementation: in that case entering the critical section would in a certain sense mean "consuming the resource" and leaving the critical section "producing the resource". Here we study another example: a *concurrent queue*, where a set of producer threads enqueue elements in the queue and a set of consumer threads dequeues elements from the queue. Here we assume that a queue has a maximum size, *i.e.* we cannot put an unbounded number of elements in the queue. A first idea is to implement it with semaphores, as shown in the following code snippets.

**Algorithm 3.12**: Concurrent queue

```
class Queue{
    int in, out, size;
    int[] buffer;
    Semaphore nonEmpty, nonFull, manipulation;

    Queue(int s){
        size = s;
        buffer = new int[size];
        in = out = 0;
        nonEmpty = new Semaphore(0);
        nonFull = new Semaphore(size);
        manipulation = new Semaphore(1);
    }
    void enqueue(int x){
        try{
            manipulation.acquire();
            nonFull.acquire();
            buffer[in] = x;
            in = (in + 1) % size;
        } catch(InterruptedException ex){}
        finally {
            manipulation.release();
            nonEmpty.release();
        }
    }
    int dequeue(){
        int x = 0;
        try{
            manipulation.acquire();
            nonEmpty.acquire();
            x = buffer[out];
            out = (out - 1) %size;
        } catch(InterruptedException ex){}
        finally {
            manipulation.release();
            nonFull.release();
        }
        return x;
39     }
40 }
```

Although at first sight this implementation might look correct, it has a (big) problem: it can lead to a deadlock in the situation where the Consumer requires *nonEmpty*, which is owned by the Producer which requires *manipulation* which is owned by the Consumer. In principle, by swapping lines 29 and 30, the code would be correct, but the goal of this example was showing that semaphores are unstructured: a correct use requires a high level of discipline. In this scenario a new abstract data type called *Monitor* comes to rescue.

## Monitors

Monitors provide, in addition to mutual exclusion, a useful mechanism to check conditions with the following semantics. If a condition does not hold:

- ▶ Release the monitor lock
- ▶ Wait for the condition to become true
- ▶ Signalling mechanisms to avoid busy loops

In Java this is given by adding to the intrinsic lock (*synchronized*) of an object the following methods:

- ▶ *wait()*: the current thread waits until it is signaled (via notify).
- ▶ *notify()*: wakes up one waiting thread (an arbitrary one).
- ▶ *notifyAll()*: wakes up *all* waiting threads

With the help of monitors, implementing the concurrent queue of the previous paragraph becomes (almost) trivial.

**Algorithm 3.13**: Concurrent queue

```
class Queue{
    int in, out, size;
    int[] buffer;

    Queue(int s){
        size = s;
        buffer = new int[size];
        in = out = 0;
    }
    void enqueue(int x){
        while(isFull()){
            try{
                wait();
            }catch(InterruptedException e){}
            doEnqueue(x);
            notifyAll();
        }
    }
    int dequeue(){
        int x;
        while(isEmpty()){
            try{
```

```
23                 wait();
24             } catch(InterruptedException e){}
25             x = doDequeue();
26             notifyAll();
27             return x;
28         }
29     }
30 }
```

Here some methods are given as a blackbox, but the intuitive (sequential) version works. Pay attention to the fact that at line 21 an *if* statement would not be sufficient: if multiple threads would enter to the wait zone, then all of them would enter to the critical section when woken up, and this would not provide mutual exclusion. Moreover, since threads are managed by the operating system, it could happen that a thread that is waiting gets woken up without a corresponding notify signal. We conclude this paragraph with another important feature of Java locks, the *Condition* interface, which offer the following methods:

- *await()*: the current thread waits until condition is signaled
- *signal()*: wakes up one thread waiting on this condition
- *signalAll()*: wakes up all threads waiting on this condition

We illustrate them by mean of an example.

**Example 3.4.2** (The Sleeping Barber) The analogy is based upon a hypothetical barber shop with one barber. The barber has one barber's chair in a cutting room and a waiting room containing a number of chairs in it. When the barber finishes cutting a customer's hair, he dismisses the customer and goes to the waiting room to see if there are others waiting. If there are, he brings one of them back to the chair and cuts their hair. If there are none, he returns to the chair and sleeps in it. Each customer, when they arrive, looks to see what the barber is doing. If the barber is sleeping, the customer wakes him up and sits in the cutting room chair. If the barber is cutting hair, the customer stays in the waiting room. If there is a free chair in the waiting room, the customer sits in it and waits their turn. If there is no free chair, the customer leaves.

In order to solve this problem we introduce two additional variables $m$ and $n$ with the following properties:

- $m \leq 0$ iff the buffer is full and $m$ clients are waiting
- $n \leq 0$ iff the buffer is empty and $n$ barbers are waiting

We get the following implementation.

**Algorithm 3.14**: Concurrent queue

```
class Queue{
    int in=0, out=0, size;
    long buf[];
    final Lock lock = new ReentrantLock();
    int n = 0; final Condition notFull=lock.newCondition();
    int m;
    final Condition notEmpty = lock.newCondition();
    public Queue(int s){
        size = s;
        m = s−1;
        buf = new long[size];
    }
    void enqueue(int x){
        lock.lock();
        m−−; if (m<0)
            while (isFull())
                try { notFull.await(); }
                catch(InterruptedException e){}
        doEnqueue(x);
        n++;
        if (n<=0) notEmpty.signal();
        lock.unlock();
    }
    long dequeue(){
        long x;
        lock.lock();
        n−−; if (n<0)
            while (isEmpty())
                try { notEmpty.await(); }
                catch(InterruptedException e){}
        x = doDequeue();
        m++;
        if (m<=0) notFull.signal();
        lock.unlock();
        return x;
    }
}
```

## Readers-Writers Lock

Many shared objects have the property that most method calls, called readers, return information about the object's state without modifying the object, while only a small number of calls, called writers, actually modify the object. There is no need for readers to synchronize with one another; it is perfectly safe for them to access the object concurrently. Writers, on the other hand, must lock out readers as well as other writers. A readers–writers lock allows multiple readers or a single writer to enter the critical section concurrently. We use an interface that exports two lock objects: the read lock and the write lock. They satisfy the following safety properties:

- No thread can acquire the write lock while any thread holds either the write lock or the read lock
- No thread can acquire the read lock while any thread holds the write lock

Naturally, multiple threads may hold the read lock at the same time.

We consider a sequence of increasingly sophisticated reader–writer lock implementations. The *SimpleReadWriteLock* class in the following code snippet uses a counter for the number of readers and another one for the number of writers. If there are no writers in the critical section, readers are allowed to enter. However a writer, has to wait until all readers are finished.

**Algorithm 3.15**: Read-Write Lock with priority to the readers

```
class RWLock{
    int writers = 0;
    int readers = 0;

    synchronized void acquire_read(){
        while(writers>0){
            try{wait(); }
            catch(InterruptedException e){}
        }
        readers++;
    }

    synchronized void release_read(){
        readers−−;
        notifyAll();
    }

    synchronized void acquire_write(){
        while(writers>0 || readers>0){
            try{wait(); }
            catch(InterruptedException e){}
        }
        writers++;
    }

    synchronized void release_write(){
        writers−−;
        notifyAll();
    }
}
```

Even though the *SimpleReadWriteLock* algorithm is correct, it is still not quite satisfactory. If readers are much more frequent than writers, as is usually the case,then writers could be locked out for a long time by a continual stream of readers. In order to alleviate this problem, we could introduce a variable that keeps track of how many writers are waiting to enter the critical section. If there are any, we pause the reader and we let the writers do their work. The idea is exposed in the following code snippet.

**Algorithm 3.16**: Read-Write Lock with priority to the writers

```
class RWLock{
    int writers = 0;
    int readers = 0;
    int writersWaiting = 0;

    synchronized void acquire_read(){
        while(writers>0 || writersWaiting>0){
            try{wait();}
            catch(InterruptedException e){}
        }
        readers++;
    }

    synchronized void release_read(){
        readers−−;
        notifyAll();
    }

    synchronized void acquire_write(){
        writersWaiting++;
        while(writers>0 || readers>0){
            try{wait(); }
            catch(InterruptedException e){}
        }
        writersWaiting−−;
        writers++;
    }

    synchronized void release_write(){
        writers−−;
        notifyAll();
    }
}
```

We have seen two different locks: one which gives strong priority to the readers and another one which gives strong priority to the writers. Of course, both models are not fair. But what notion of *fairness* should we consider? A possibility would be:

- When a writer finishes, a number $k$ of currently waiting readers may pass.
- When the $k$ readers have passed, the next writer may enter (if any), otherwise further readers may enter until the next writer enters (who has to wait until current readers finish).

This fair(er) idea is implemented in the following snippet. We have introduced a variable *writersWaiting()* which counts the number of writers trying to enter the critical section (similarly for *readersWaiting()*). Moreover we have a variable *writersWait* to keep track of the number of the readers that the writer has to wait (hence, at the beginning this variable will be set to $k$).

**Algorithm 3.17**: A fair(er) Read-Write Lock

```
class RWLock{
    int writers = 0;
    int readers = 0;
    int writersWaiting = 0;
    int readersWaiting = 0;
    int writersWait = k;

    synchronized void acquire_read(){
        readersWaiting++;
        while(writers>0 || (writersWaiting>0 && writersWait<=0)){
            try{wait();}
            catch(InterruptedException e){}
        }
        readersWaiting--;
        writersWait--;
        readers++;
    }

    synchronized void release_read(){
        readers--;
        notifyAll();
    }

    synchronized void acquire_write(){
        writersWaiting++;
        while(writers>0 || readers>0 || writersWait>0){
            try{wait(); }
            catch(InterruptedException e){}
        }
        writersWaiting--;
        writers++;
    }

    synchronized void release_write(){
        writers--;
        writersWait = k;
        notifyAll();
    }
}
```

If the parameter $k$ reflects the proportion of reads and writes in this context, then the previous lock can be considered fair (or, at least, much more fair than the previous versions we presented).

## 3.5 Lock granularity

The size of our critical section greatly influences the possible speedup we can achieve through parallelization. When programming with locks, it's common practice to start of by essentially wrapping every-thing that remotely resembles a critical section in one huge lock. This is called *coarse-grained locking*. In this section, we introduce different locking strategies which, with some customization, can be applied to many different concurrent data structures. We introduce these strategies at the hand of sorted linked lists, which implement an *add(x)*, a *remove(x)* and a *contains(x)* method.

As mentioned the first idea is to take the sequential method of the data structure and synchronize all his methods. The disadvantage here is the considerable size of the critical section. As all the methods share a single lock and start by acquiring and end by releasing it, this implementation allows for barely any concurrency whatsoever.However, this strategy does have one advantage: its simplicity. Implementing it does not really require any effort on the side of the programmer. Now we move to more involved versions.

## Fine-Grained Locking

As a first step of improving *coarse-grained locking*, we can lock individual elements instead of the entire data structure. As a thread traverses the data structure, it locks each node when it first visits it and releases it once it has acquired the lock for the next node. This method of locking is called *hand-over-hand* locking. This method allows threads to traverse the data structure in a pipelined fashion.

In order to avoid deadlocks, it's important that all threads acquire the locks in some predefined order, *e.g.* in the case of the sorted linked list, all threads start at the head and do not try to "skip" nodes.

**Example 3.5.1** In order to avoid deadlocks, it's important that all threads acquire the locks in some predefined order. In the case of sorted linked lists, this requirement is easily met by always starting at the *head* sentinel and only proceeding in a hand-over-hand fashion.

```
public boolean remove(T item){
    int key = item.hashCode();
    head.lock();
    Node pred = head;
    try{
        Node curr = pred.next;
        curr.lock();
        try{
            while(curr.key < key){
                pred.unlock();
                pred = curr;
                curr = curr.next;
                curr.lock();
            }
            if(key == curr.key){
                pred.next = curr.next;
                return true;
            } else return false;
        } finally {curr.unlock(); }
    } finally {pred.unlock(); }
}
```

**Algorithm 3.18**: Fine-grained locking remove method

We've now implemented an actual concurrent data structure, in the sense that several threads can actually operate on it simultaneously. But unfortunately, this strategy is still far from perfect. As threads iterate over this data structure in a pipelined fashion, the slowest thread sets the tempo for all threads that immediately follow it, meaning that a potentially "fast" thread might experience considerable slowdown. In addition, for large data structures, there is still a potentially long chain of lock acquisitions and releases.

## Optimistic Locking

With *optimistic locking*, we take somewhat of a risk. We iterate the data structure without taking any locks. Once we've found the required elements we lock them and check if everything is still correct. If we find that in between finding the elements and taking the locks the state of the data structure changed to one where we can not execute our operation reliably anymore, we start over. As such conflicts are rare, we consider this approach to be optimistic. Implementing such a verification method is non-trivial and requires careful thought. What state do we, as an operating thread, *expect* and how can we check that this expected state actually holds?

**Example 3.5.2** In the case of the sorted linked list, we verify the following two conditions:

- The *curr* node, i.e. the node we're operating on, is still reachable.
- *pred* still points to *curr*, i.e. the two nodes we're about to operate on are actually the ones we *should* operate on.

```
public boolean remove(T item){
    int key = item.hashCode();
    while(true){
        Node pred = head;
        Node curr = pred.next;
        while(curr.key < key){
            pred = curr;
            curr = curr.next;
        }
        pred.lock(); curr.lock();
        try{
            if(validate(pred, curr)){
                pred.next = curr.next;
                return true;
            } else return false;
        } finally {pred.unlock(); curr.unlock(); }
    }
}
public boolean validate(Node pred, Node curr){
    Node node = head;
    while(node.key <= pred.key){
        if(node == pred) return pred.next == curr;
        node = node.next;
    }
    return false;
}
```

**Algorithm 3.19**: Optimistic locking remove method

We've now been able to alleviate most issues. Nonetheless, a few remain. First, by always calling *validate*, we're essentially iterating the list twice. If we then have a high amount of thread contention, i.e. a lot of threads operating on the same area of the data structure, these *validate* will often return *false*, thereby forcing most threads to re-iterate the entire list. Finally, we would like an operation as simple as the *contains* method to not have to acquire any locks whatsoever, as this does seem like a slight overkill.

## Lazy Locking

*Lazy synchronization* builds on top of optimistic synchronization by adding a boolean *marked* field to each node, which, when false, it holds the invariant that

- ▶ the node is in the set and
- ▶ the node is reachable.

The *remove* method now first *lazily* removes a node by setting its *marked* bit, then *physically* removes it, e.g. by redirecting the pointer from the previous node. We adjust the *validate* method so that it only checks whether the *marked* bit is set and whether the local state is still as expected. Therefore, the remainder of the *add* and *remove* methods can be left the same. Finally, we change the *contains* method such that it simply iterates the data structure without taking any locks and, if it finds the specified node, checks whether it's marked or not.

**Example 3.5.3** The concrete implementation of the sorted linked list according to the lazy locking strategy now looks as follows.

**Algorithm 3.20**: Lazy locking remove method

```
public boolean remove(T item){
    int key = item.hashCode();
    while(true){
        Node pred = head;
        Node curr = pred.next;
        while(curr.key < key){
            pred = curr;
            curr = curr.next;
        }
        pred.lock(); curr.lock();
        try{
            if(validate(pred, curr){
                if(key == curr.key){
                    curr.marked = true;
                    pred.next = curr.next;
                    return true;
                } else return false;
            }
        } finally {pred.unlock(); curr.unlock(); }
    }
}
public boolean validate(Node pred; Node curr){
    return !pred.marked && !curr.marked && pred.next == curr;
}
```

## 3.6 Atomic Operations

We have already seen two main ways to implement locks. The first approach are *spinlocks*, that by actively trying to acquire the lock waste computational resources and, in the case of long-lived contention, degrade performance. The second approach is using locks that, when the thread fails to acquire them, the thread goes to sleep until something changes. This locks require the support from the operating system scheduler and introduce a significant overhead. Of course, hybrid solutions that first spin on the lock and after a while go to sleep also exist. In general locks have several disadvantages: they are *pessimistic* by design (they assume the worst and enforce mutual exclusion, *i.e.* they assume that there will be several conflicts), they introduce overhead for each lock taken even in the uncontended case (which, by Amdahl's law, causes significant performance degradation), and have several problems in special cases such as delay of the thread in the critical section. Moreover the programmer has to avoid introducing problems such as deadlocks, livelocks and starvation.

In this section we will examine another way to enforce mutual exclusion without locking: *lock-free* programming (which, as should be intuitive from the previous considerations, is also called *optimistic concurrency control*). Recall that lock-freedom means that at least one thread always makes progress even if other threads run concurrently. Lock-freedom implies system-wide progress but not freedom from starvation. A stronger concept in this sense would be *wait-freedom*, which requires that all threads eventually make progress. In non-blocking algorithms the failure or suspension of one thread cannot cause failure or suspension of another thread.

In order to write lock-free programs, we need *atomic operations*. Two very important examples are the *Test-And-Set* (TAS) and the *Compare-And-Swap* (CAS) operations. Their semantics is described in the following pseudocodes.

```
inputs
a = memory address
outputs
v: flag indicating if the test was succesful

v = mem[a]
if(v == 0){v = 1; return true}
return false
```

**Algorithm 3.21**: TAS

```
inputs
a: memory address
old: old value
new: new value

oldval = mem[a]
if(old == oldval) mem[a] = new
return oldval
```

**Algorithm 3.22**: CAS

As we have already discussed, TAS and CAS allow to implement a spinlock. In general CAS is strictly more powerful than TAS: every atomic operation can be efficiently simulated with CAS, but not with TAS, as we will see in the next chapter when we will discuss *Consensus*.

**Example 3.6.1** (Non-Blocking Counter) The following code snippet implements a correct lock-free counter. In this example we have the ABA problem (in a nutshell, although it may seem that if CAS succeeds nobody has modified the value of the variable, this may not be true).

**Algorithm 3.23**: Non-Blocking Counter

```
public class Counter{
    private AtomicInteger value;
    public int inc(){
        int v = value.get(); ;
        do{
            v = value.get();
        }while(!CAS(value, v, v+1));
        return v + 1;
    }
}
```

**Example 3.6.2** (Lock-Free Stack)

**Algorithm 3.24**: Lock-Free Stack

```
public Class ConcurrentStack{
    AtomicReference<Node> top = new AtomicReference<Node>();
    public Long pop(){
        Node head, next;
        do{
            head top.get();
            if(head == null) return null;
            next = head.next;
        }while(!top.compareAndSet(head, next));
        return head.item;
    }
    public void push(Long item){
        Node newi = new Node(item);
        Node head;
        do{
            head = top.get();
            newi.next = head;
        }while(!top.compareAndSet(head, newi));
    }
}
```

The advantage of using lock-free data structure is that they are deadlock-free by design. However, if we compare the performance of the previous example with a locked version, we would see that our lock-free programs perform poorly. Keep in mind that a lock-free algorithm does not automatically provide better performance than its blocking equivalent. Atomic operations are expensive (they require specific hardware support) and contention is still a problem. In fact, when there is high contention, the expensive atomic operations are retried several times, leading to performance degradation. In general we have that optimistic concurrency control could be useful whenever there is low data contention, thus we assume that although there will be conflicts, they will be rare. We will look for indication if two threads actually tried to update the shared resource at the same time. If this is the case, then one of the threads' operation will be discarded and retried i.e. performing an extra atomic operation. A solution that works very well in practice to alleviate this problem is introducing an exponential backoff mechanism.

Now we go back to our lock free stack implementation in order to introduce an issue that can arise in this context. Without exponential backoff, our implementation is slow. Instead of introducing a backoff, we exploit the observation that we create a new node for every single *push* operation. As an optimization, we implement a node pool which allows for reuse of the node objects:

```
public Node get(Long item){
    do{
        head = top.get();
        if(head == null) return new Node(item);
        next = head.next;
    }while(!top.compareAndSet(head, next);
    head.item = item;
    return head;
}
```

**Algorithm 3.25**: Get Node from the pool with a given item

```
public void put(Node n){
    Node head;
    do{
        head = top.get();
        n.next = head;
    }while(!top.compareAndSet(head, n))
}
```

**Algorithm 3.26**: Put Node back the pool

By using those two methods we can implement our lock-free stack as before, but without creating too many Node objects (which is not a light operation, as you will see in the *Systems Programming and Computer Architecture* course next semester). In facts, when we adjust the stack implementation to use such a *NodePool*, we do indeed see a massive improvement in execution time. However, we see that the program exhibits erroneous behavior for some runs. This leads to a very common problem in lock-free concurrent programming, the *ABA Problem*.

**Definition 3.6.1** (ABA Problem) *The ABA problem occurs when one activity fails to recognize that a single memory location was modified temporarily by another activity and therefore erroneously assumes that the overall state has not been changed.*

In the case of the lock-free stack with node reuse, this means the following scenario would exhibit an ABA-problem: We try to *pop* and observe that $head == a$ and $head.next == b$. We then try to compare-and-set *head* to $b$. Suppose however, that another thread removes both $a$ and $b$, pushes some other node $c$ and then pushes $a$. We would then observe $head == a$ as expected and set $head == b$, a node which is possibly not even in the stack at the time. In order to be clear we introduce another possible scenario in the following figure.

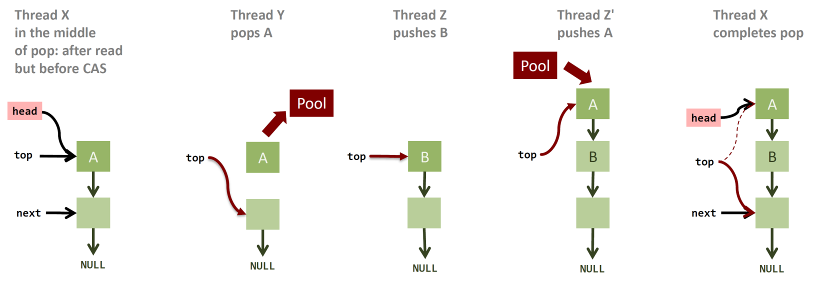


**Figure 3.2:** An example of ABA Problem

There are several possible alternatives to alleviate the ABA problem:

- DCAS: We can check whether both *head* and *head.next* are as expected, but doesn't exist on most platforms.
- Garbage Collection: Would eliminate the need for a *NodePool*, but is very slow and doesn't always exist either.
- Pointer Tagging: By incrementing the address bits made available by alignment, we can decrease the odds of the ABA problem occuring by a lot. Nonetheless, this doesn't actually alleviate the problem, only delay it.
- Hazard Pointers: We can associate an *AtomicReferenceArray<Node>* with the data structure where we temporarily store references which we've read and wish to write to in the future. Whenever we return a Node to the *NodePool*, we check whether its reference is stored in the *hazarduous* array. While this solution does work, the final product of a *NodePool* doesn't really improve performance when compared to regular memory allocation with a garbage collector.

As a conclusion to this section we do the following observation: lock-free programming alleviates some problems of parallel programming with locks. However it introduces other kind of problems such as the ABA problem, it puts burden on the programmer and it may not improve performance. In the next section we see an alternative, which is still in the category of *optimal concurrency control*, but is much easier for the programmer. This kind of solution is likely to become more and more important in the following years.

## 3.7 Transactional Memory

We saw that programming with locks is difficult and exhibits problems such as deadlocks, convoying (when a thread holding a resource $R$ is de-scheduled while other threads queue up waiting for $R$) and priority inversion (when a lower priority thread holds a resource $R$ that a high priority thread is waiting on). Association of locks and data is established by convention. The best one can do is reasonably document the code. Recall for example the bank account problem: if one wants to implement it without introducing deadlocks, he has to write several lines of synchronization for just a couple of lines describing the actual operation. Another drawback of locks is that they are not composable, *i.e.* combining $n$ thread-safe operations is not straighforward. Lock-free programming can be even worse. The goal would be to find a tool to remove the burden of synchronization from the programmer and pace it in the system (hardware and software). *Transactional memory* tries to do this by introducing atomic blocks. With transactional memory the programmer explicitly defines atomic code sections, but he's mainly concerned with *what* operations should be atomic rather than *how* this is actually achieved. Transactional memory is simpler and less error prone than locking, it allows a declarative semantics (*what* instead of *how*) and it's optimistic by design (*i.e.* similarly to optimistical concurrency control with atomic operations, it does not assumes the worst and it does not require mutual exclusion). With this model changes made by a transaction are made visibly atomically. Other threads preserve either the initial or the final state, but not any intermediate states. Locks enforce atomicity via mutual exclusion, transactions do not require mutual exclusion.

**Definition 3.7.1** (Transactional Memory) *Transactional Memory is a programming model whereby loads and stores on a particular thread can be grouped into transactions. The read set and write set of a transaction are the set of addresses read from and written to respectively during the transaction. A data conflict occurs in a transaction if another processor reads or writes a value from the transaction's write set, or writes to an address in the transaction's read set. Data conflicts cause the transaction to abort, and all instructions executed since the start of the transaction (and all changes to the write set) to be discarded.*

Transactions run in isolation: while a transaction is running, effects from other transactions are not observed. A good analogy is the one of a snapshot: transactional memory works as if transaction takes a snapshot of the global state when it begins and then operates on that snapshot.

In database transactions that are four properties that are important, and usually are recalled with the word *ACID*. The properties are atomicity, consistency (*i.e.* the database remains in a consistent state), isolation (*i.e.* no mutual corruption of data) and durability (*i.e.* transaction effects are stored in disk and hence will survive power loss). Transactional memory satisfies three of this properties: atomicity, consistency and isolation.

Transactional memory works as follows: it tries to do the transaction without enforcing mutual exclusion but, if there is a conflict, the transaction aborts and has to restart again. If there is no conflict, the transaction commits. A system to enforce this behaviour can be either implemented in software (which can be fast, but often cannot handle big transactions) or in hardware (which offers great flexibility, but achieving good performance might be challenging).

Transactional Memory is far to be part of parallel programming routine, but is likely to get more and more important in the future. Today we have *ScalaSTM*, a Java API through which we can access the methods provided by the Scala STM library. ScalaSTM is a so-called *reference-based STM*, which means that mutable state, i.e. state which can only be modified *inside a transaction*, is put into special variables.

*private final Ref.View<Integer> count = STM.newRef(0);*

Arrays can be declared as follows:

*private TArray.View<E> items = STM.newTArray(capacity);*

Everything else is immutable, which means any other variable accessed inside an atomic block must be declared *final*.

We can create an atomic block as follows:

*STM.atomic(new Runnable(){...});* or *STM.atomic(new Callable<T>(){...});*

Note that the passed *Runnable* or *Callable* object must implement the *public void run()* or the *public T call()* method, respectively.

**Example 3.7.1** We can now try to implement the *enq()* method of the bounded queue using ScalaSTM. First, we need to distinguish between mutable state and immutable state. We see that we modify the variables *count, tail* and *items*. As these variables should indeed only be accessed from within a transaction, they are part of the mutable state. The method also accepts a parameter *x*. As we only read object and do not actually try to modify, we'll include it in the immutable state. All in all, our class will look something like this (omitting the other methods for brevity):

```
public class CircularBufferSTM<T>{
    private final Ref.View<Integer> count = STM.newRef(0);
    private final Ref.View<Integer> tail = STM.newRef(0);
    private TArray.View<T> items;
    public CircularBufferSTM(int capacity){
        items = STM.newTArray(capacity);
    }
    public void enq(final T x){
        STM.atomic(new Runnable(){
            if(count.get() == items.length())STM.retry();
            items.update(tail.get(), x);
            tail.set((tail.get()+1) % items.length());
            STM.increment(count, 1);
        });
    }
}
```

# Other topics 4

## 4.1 Linearizability and Sequential Consistency

In the previous chapter we have seen many ways to implement blocking and non-blocking concurrent data structures. We have seen that those implementations are *correct*. Interestingly enough, we did all this without properly defining what a *correct parallel execution* is. This is the aim of this section, defining different notions of correctness, some more strict than others, so that we may argue and prove things about our implementations.

**Definition 4.1.1** (Object) *An object is a variable or a data structure storing information. This is a general term for any entity that can be modified, like a queue, stack, memory slot, ...*

An operation $f$ access or manipulates an object. The operation $f$ starts at clock time $f_b$ and ends at clock time $f_e$. An operation can be as simple as extracting an element from a data structure, but an operation may also be more complex, like fetching an element, modifying it and storing it again. We introduce the notation $f < g$, which means that $f_e < g_b$.

**Definition 4.1.2** ((Sequential) Execution) *An execution E is a set of operations on one or multiple objects that are executed by a set of nodes.*

*An execution restricted to a single node is a sequential execution. All operations are executed sequentially, which means that no two operations $f$ and $g$ are concurrent, i.e. we have $f < g$ or $g < f$.*

**Definition 4.1.3** (Semantic Equivalence) *Two executions are semantically equivalent if they contain exactly the same operations. Moreover, each pair of corresponding operations has the same effect in both executions. For example, when dealing with a stack object, corresponding pop operations in two different semantically equivalent executions must yield the same element of the stack.*

**Definition 4.1.4** (Linearizability) *An execution E is called linearizable if there is a sequence of operations S such that:*

- *S is correct and semantically equivalent to E*
- *Whenever $f < g$ for two operations $f, g$ in E, then also $f < g$ in S*

The idea behind linearizability is that the concurrent history is equivalent to some sequential history. The rule is that if one method call precedes another, then the earlier call must have taken effect before the later call. By contrast, if two method calls overlap, then their order is ambiguous, and we are free to order them in any convenient way.

A linearization point of operation f is some $f_\bullet \in [f_b, f_e]$.

**Theorem 4.1.1** *An execution $E$ is linearizable iff there exist linearization points such that the sequential execution $S$ that results in ordering the operations according to those linearization points is semantically equivalent to $E$.*

*Proof.* Let $f$ and $g$ be two operations in $E$ with $f_e < g_b$. Then by definition of linearization points we also have $f_\bullet < g_\bullet$ and therefore $f < g$ in $S$. □

**Definition 4.1.5** (Sequential Consistency) *An execution $E$ is called sequentially consistent, if there is a sequence of operations $S$ such that:*

- *$S$ is correct and semantically equivalent to $E$*
- *Whenever $f < g$ for two operations $f, g$ on the same node in $E$, then also $f < g$ in $S$.*

**Example 4.1.1** We are given the following history:

```
A: s.push(2)
B: s.push(1)
A: void
B: void
A: s.pop()
B: s.pop()
A: 2
B: 1
```

We can see that the first two method calls and the last two method calls overlap.
An easy visualization of histories is through the use of time lines. In this case, it could look like this:

```
A s.push(1): *---------*
B s.push(2):      *---------*
B s.pop()->1:                     *---------------*
A s.pop()->2:                           *-------------*
```

This history is both linearizable and sequentially consistent.

**Theorem 4.1.2** *Linearizability implies sequential consistency (but the opposite direction does not hold).*

*Proof.* Since linearizability (order of operations on *any* node must be respected) is stricter than sequential consistency (only order operations on the same node must be respected), the theorem follows immediately. As a counterexample to the other direction consider the following history:

```
A: s.pop(1)
A: void
B: s.push(1)
B: void
```

which is sequential consistent but not linearizable. □

A system or implementation is called linearizable if it ensures that every possible execution is linearizable.

**Example 4.1.2** You submit a comment on your favorite social media platform using your mobile phone. The comment is immediately visible on the phone, but not on your laptop. Is this level of consistency acceptable? In this context a linearizable implementation would have to make sure that the comment is immediately visible on any device, as the *read* operation starts after the *write* operation finishes. If the system is only sequentially consistent, the comment does not need to be immediately visible on every device.

**Definition 4.1.6** (Restricted Execution) *Let $E$ be an execution involving operations on multiple objects. For some object $o$ we let the restricted execution $E|o$ be the execution $E$ filtered to only contain operations involving object $o$.*

**Definition 4.1.7** (Composability) *A consistency model is called composable if the following holds: if for every object $o$ the restricted execution $E|o$ is consistent, then also $E$ is consistent. Composability enables to implement, verify and execute multiple concurrent objects independently.*

**Theorem 4.1.3** *Sequential consistency is not composable.*

*Proof.* Consider the following history as a counterexample to the composability of sequential consistency:

```
A: read(x) = 1
A: void
A: write(y)=1
A: void
B: read(y)=1
B: void
B: write(x)=1
B: void
```

□

**Theorem 4.1.4** *Linearizability is composable.*

*Proof.* Let $E$ be an execution composed of multiple resticred executions $E|x$. For any object $x$ there is a sequential execution $S|x$ that is semantically consistent to $E|x$ and in which the operations are ordered according to clock linearization points. Let $S$ be the sequential execution ordered according to all linearization points of all executions $E|x$. $S$ is semantically equivalent to $E$ as $S|x$ is semantically equivalent to $E|x$ for all objects $x$ and two object disjoint executions cannot interfere. Furthermore, if $f_e < g_b$ in $E$, then also $f_\bullet < g_\bullet$ in $E$ and therefore $f < g$ in $S$. □

**Definition 4.1.8** (Quiescent Consistency) *An execution E is called quiescent consistent, if there exist a sequence of operations S such that:*

- *S is correct and semantically equivalent to E*
- *Let t be some quiescent point, i.e. for all operations f we have $f_e < t$ or $f_b > t$. Then for every t and every pair of operations $g, h$ with $g_e < t$ and $h_b > t$ we also have $g < h$ in S*

**Theorem 4.1.5** *Linearizability implies quiescent consistency.*

*Proof.* Let $E$ be the original execution and $S$ be the semantically equivalent sequential execution. Let $t$ be a quiescent point and consider two operations $g, h$ with $g_e < t < h_b$. Then we have $g < h$ in $S$. This is also guaranteed by linearizability since $g_e < t < h_s$ implies $g < h$. □

**Theorem 4.1.6** *Sequential consistency and quiescent consistency do not imply one another.*

*Proof.* There are executions that are sequentially consistent but not quiescently consistent. An object initially has value 2. We apply two operations to this object: *inc* (increment the object by 1) and *double* (multiply the object by 2). Assume that $inc < double$, but *inc* and *double* are executed on different nodes. Then a result of 5 (first *double*, then *inc*) is sequentially consistent but not quiescently consistent. There are executions that are quiescently consistent but not sequentially consistent. An object initially has value 2. Assume to have three operations on two nodes $u$ and $v$. Node $u$ calls first *inc* then *double*, node $v$ calls *inc* once with $inc_b^v < inc_e^u < double_b^u < inc_e^v$. Since there is no quiescent point, quiescent consistency is okay with a sequential execution that doubles first, resulting in $((2 \cdot 2) + 1) + 1 = 6$. The sequential execution demands that $inc^u < double^u$, hence the result should be strictly larger than 6 (either 7 or 8). □

## 4.2 Volatile Fields

In this section, we give a brief overview of *volatile fields*. We introduce them because they are instructive and they give some insights about the Java Memory Model. However, we strongly recommend you to avoid them in practice: they are really for experts!

Consider the following scenario: there are two variables, X and Y, which have value zero. Then two threads, thread $A$ and thread $B$, run concurrently. Thread $A$ executes the following instructions: $X = 1, I = Y$. Thread $B$ executes the following: $Y = 1, J = X$. The question is: *how is it possible that one could end in a situation where both I and J have value zero, although no possible interleaving of the instructions leads to such a result?* There are two possible answers to this (apparent) paradox:

- In an environment with multiple CPUs, each CPU has its own separate cache. In this scenario is possible that, if Thread $A$ runs on CPU 1 and Thread $B$ runs on CPU 2, then $A$ writes the value of $X$ in the cache (and hence $B$ will not see it) and reads the value of $Y$ from its cache (and hence will not see the update from $B$). Similarly, $B$ writes $Y$ and reads $X$ from its local cache. By knowing what happens underneath, one can explain the apparently strange behaviour of the program.

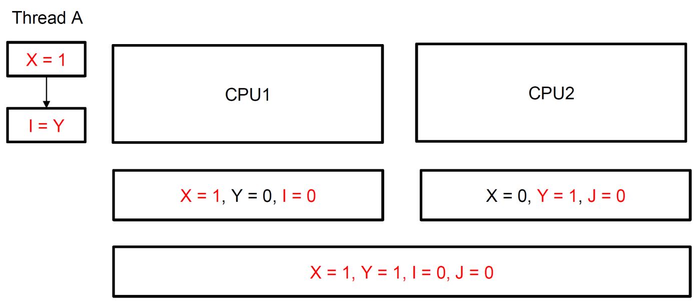


**Figure 4.1:** Program Behaviour

- Another explanation is given by the *Out-Of-Order-Execution*. Even in a single core environment, the compiler, the CPU or the hardware could reorder instructions. This is done for the sake of efficiency and the pact is that the *sequential* programmer does not see any difference from the behavior of his program. However, when there are multiple threads, such reorderings could lead to situations where bad interleavings can actually happen.

We can make a link with the previous section by stating that the Java Memory Model is very relaxed, in facts it does not either guarantees sequential consistency, even instructions reorders are possible! This choice is done to accommodate compiler optimizations. Here volatile fields come into play. In the previous chapter we stated that volatile accesses do not count as data races. But what does it actually mean? This means that the compiler does not touch (and does not reorder) volatile accesses and it forces reads and writes directly to memory (this is actually not completely true, in facts the hardware could still use some tricks to use cached value in order to save time, but now there is the guarantees that caches are consistent). Hence, by declaring $X$ and $Y$ volatile, the apparently strange behaviour of our previous example will not happen anymore. Volatile fields are weaker than locking: they give less guarantees, but they introduce less overhead. Let's see an additional example of the consequences of declaring a variable volatile.

**Example 4.2.1** The following code

```
 volatile int x;
void foo(){
    x++;
}

```

is equivalent to

```
int x;
int tmp;
```

```
void foo(){
    synchronized(x){
        tmp = x;
    }
    tmp = tmp + 1;
    synchronized(x){
        x = tmp;
    }
}
```

If this increment algorithm is executed by a single writer thread and multiple readers, than using volatile is enough. However, if there are multiple writers one needs to go back to locks or atomic operations.

We conclude this section by recalling the two common use cases of volatile variables: setting a flag and doing operations with a single writer (and multiple readers).

## 4.3 Consensus

The goal of this section is to identify a set of primitive synchronization operations powerful enough to solve synchronization problems likely to arise in practice. To this end, we need some way to evaluate the power of various synchronization primitives: what synchronization problems they can solve, and how efficiently they can solve them. The basic idea is simple: each class in the hierarchy has an associated *consensus* number, which is the maximum number of threads for which objects of the class can solve an elementary synchronization problem called consensus. The consensus problem is simple. A certain number of threads call a *decide* method which outputs a value. The requirements of this method in order to be correct are:

- *Wait-Free:* consensus returns in finite time for each thread
- *Consistent:* all threads decide on the same value
- *Valid:* the common decision value is some thread's input

The linearizability of consensus must be such that the first thread's decision is adopted for all threads.

**Definition 4.3.1** *A class C solves $n$−thread consensus if there exist a consensus protocol using any number of objects of class C and any number of atomic registers.*

**Definition 4.3.2** *The consensus number of a class C is the largest $n$ for which that class solves $n$−thread consensus. IF no largest $n$ exists, we say the consensus number of the class in infinite.*

A good place to start is to think about the simplest interesting case: binary consensus (*i.e.* inputs are either zero or one) for two threads $A$ and $B$. Each thread makes moves until it decides on a value. Here, a *move* is a method call to a shared object. A *protocol state* consists of the states of the threads and the shared objects. An *initial state* is a protocol state before any thread has moved, and a *final state* is a protocol state after all threads have finished. The *decision* value of any final state is the value decided by all threads in that state. A wait-free protocol's set of possible states forms a tree, where each node represents a possible protocol state, and each edge represents a possible move by some thread. An edge for $A$ from node $s$ to node $s'$ means that if $A$ moves in protocol state $s$, then the new protocol state is $s'$. Because the protocol is wait-free, the tree must be finite. Leaf nodes represent final protocol states, and are labeled with their decision values, either 0 or 1. A protocol state is bivalent if the decision value is not yet fixed: there is some execution starting from that state in which the threads decide 0, and one in which they decide 1. By contrast, the protocol state is univalent if the outcome is fixed: every execution starting from that state decides the same value. A bivalent state is a node whose descendants in the tree include both leaves labeled with 0 and leaves labeled with 1, while a univalent state is a node whose descendants include only leaves labeled with a single decision value.

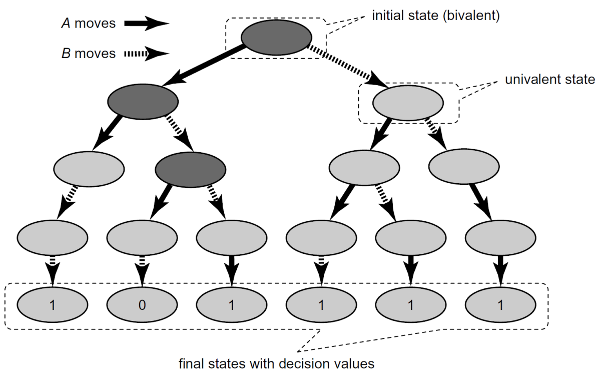


**Figure 4.2:** An execution tree for two threads $A$ and $B$. The dark shaded nodes denote bivalent states, and the lighter ones the univalent states.

A protocol state is *critical* if it is bivalent and if any thread moves, the protocol state becomes univalent.

## Atomic Registers

We begin by asking ourselves, whether we can solve consensus using atomic registers. The answer is no.

**Theorem 4.3.1** *Atomic registers have consensus number one.*

*Proof.* Suppose there exists a binary consensus protocol for two threads $A$ and $B$. Since every wait-free consensus protocol has a critical state (this is derived from the fact that there always exist a bivalent initial state), we can run the protocol until it reaches a critical state $s$. Suppose $A$'s next move carries the protocol to a zero-valent state, and $B$'s next move carries the protocol to a one-valent state. What methods could $A$ and $B$ be about to call? We now consider an exhaustive list of the possibilities: one of them reads from a register, they both write to separate registers, or they both write to the same register.

Suppose $A$ is about to read a given register ($B$ may be about to either read or write the same register or a different register). Consider two possible execution scenarios. In the first scenario, $B$ moves first, driving the protocol to a one-valent state $s'$, and then $B$ runs solo and eventually decides 1. In the second execution scenario, A moves first, then $B$ executes one operation, driving the protocol to a zero-valent state $s$. $B$ then runs solo starting in $s''$ and eventually decides zero. The problem is that the states $s'$ and $s''$ are indistinguishable to $B$ (the read $A$ performed could only change its thread-local state which is not visible to B), which means that B must decide the same value in both scenarios, a contradiction.

Suppose, instead of this scenario, both threads are about to write to different registers. $A$ is about to write to $r_0$ and $B$ to $r_1$. Let us consider two possible execution scenarios. In the first, $A$ writes to $r_0$ and then $B$ writes to $r_1$, so the resulting protocol state is zero-valent because $A$ went first. In the second, $B$ writes to $r_1$ and then $A$ writes to $r_0$, so the resulting protocol state is one-valent because $B$ went first. The problem is that both scenarios lead to indistinguishable protocol states. Neither $A$ nor $B$ can tell which move was first. The resulting state is therefore both zero-valent and one-valent, a contradiction.

Finally, suppose both threads write to the same register $r$. Again, consider two possible execution scenarios. In one scenario $A$ writes first, the resulting protocol state $s'$ is zero-valent, $B$ then runs solo and decides zero. In another scenario, $B$ writes first, the resulting protocol state $s''$ is one-valent, $B$ then runs solo and decides one. The problem is that $B$ cannot tell the difference between $s'$ and $s''$ (because in both $s'$ and $s''$ it overwrote the register $r$ and obliterated any trace of $A$'s write) so $B$ must decide the same value starting from either state, a contradiction. □

**Corollary 4.3.2** *It is impossible to construct a wait-free implementation of any object with consensus number greater than one using atomic registers.*

The aftermentioned corollary is perhaps one of the most striking impossibility results in Computer Science. It explains why, if we want to implement lock-free concurrent data structures on multiprocessors, out hardware must provide primitive synchronization operations other than loads and stores.

**Theorem 4.3.3** *Compare-And-Set has infinite consensus number.*

*Proof.* As shown in the following code snippet, threads share an *AtomicInteger* object, initialized to a constant FIRST, distinct from any thread index. Each thread calls *compareAndSet()* with FIRST as the expected value, and its own index as the new value. If thread $A$'s call returns true, then that method call was first in linearization order, so $A$ decides its own value. Otherwise $A$ reads the current *AtomicInteger* value, and takes that thread's input from the *proposed[]* array.

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

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

□

**Theorem 4.3.4** *The two-dequeuer FIFO queue class has consensus number at least two.*

*Proof.* Here, the queue stores integers. The queue is initialized by enqueuing the value WIN followed by the value LOSE. As in all the consensus protocol considered here, *decide()* first calls *propose(v)*, which stores *v* in *proposed[]*, a shared array of proposed input values. It then proceeds to dequeue the next item from the queue. If that item is the value WIN, then the calling thread was first, and it decides on its own value. If that item is the value LOSE, then the other thread was first, so the calling thread returns the other thread's input, as declared in the *proposed[]* array. The protocol is wait-free, since it contains no loops. If each thread returns its own input, then they must both have dequeued WIN, violating the FIFO queue specification. If each returns the other's input, then they must both have dequeued LOSE, also violating the queue specification. The validity condition follows from the observation that the thread that dequeued WIN stored its input in the *proposed[]* array before any value was dequeued. The next code snippet shows a two thread consensus protocol using a single FIFO queue.

```
public class QueueConsensus<T> extends ConsensusProtocol<T>{
    private static final int WIN = 0;
    private static final int LOSE = 1;
    Queue queue;
    public QueueConsensus(){
        queue = new Queue();
        queue.enqueue(WIN);
        queue.enqueue(LOSE);
    }
    public T decide(T value){
        propose(value);
        int status = queue.deq();
        int i = ThreadID.get();
        if(status == WIN) return proposed[i];
        else return proposed[i-1];
    }
}
```

□

**Theorem 4.3.5** *RMW operations have consensus number two.*

*Proof.* We do not show the theorem in its totality. We just consider the case of TAS and we consider a two thread consensus protocol with it.

One would also have to show that a three thread consensus with TAS is impossible to achieve.

```
public class TASCOnsensusProtocol<T>{
     static int X = 0;
    protected[]T proposed = (T[])new Object[2];
    void propose(T value){
        proposed[ThreadID.get()] = value;
    }
    public T decide(T value){
        propose(value);
        boolean val = TAS(X);
        if(val) return value;
        else return proposed[1−ThreadID.get()];
    }
}
```

We see that the call to TAS(X)will only return true once, namely the first time it's called. Otherwise,it'll always return false. It's therefore easy to see that the returned result is both consistent and valid. As the protocol doesn't contain any loops or other dependencies, it's wait-free. Therefore, we have provided a correct 2-thread consensus protocol and shown that test-and-set implements 2-thread consensus. Note that we can't extend this implementation to n-treads, as a "loser" thread has no way of telling which entry of the proposed array it should use. □

Given the previous theorems we have that it is impossible to implement wait-free FIFO queues, wait-free RMW operations and CAS with atomic registers. This holds because they all have a consensus number larger then one. This section also explains why in the section about atomic operations we stated that CAS is strictly more powerful than TAS: CAS has consensus number infinity, while TAS only two. Many people in the past have attempted to implement RMW operations with atomic registers, but this would be similar to trying to square the circle: a loss of time.

## 4.4 Parallel Sorting

Sorting is one of the most important computational tasks. For example Donald Knuth, in its famous *The Art of Computer Programming*, stated: *Indeed, I believe that virtually every important aspect of programming arises somewhere in the context of sorting*. In the previous semester you have seen different algorithms of sorting and some of them you implemented also in this course by exploiting the framework for the divide and conquer paradigm. We know that a sorting algorithm, with the exception of some specialized cases, cannot do better than $\mathcal{O}(n \log n)$ in the worst case. The obvious next step is to try to break this lower bound by using the multiprocessors available to us, *i.e.* by parallelizing sorting algorithms. Now we present a new class of sorting algorithms: *sorting networks*. A sorting network is a network of *comparators*. A comparator is a computing element with two input wires and two output wires, called the *top* and *bottom* wires. It receives two numbers on its input wires, and forwards the larger to its top wire and the smaller two its bottom wire. A comparator is *synchronous*, *i.e.* it outputs values only when both inputs have arrived.

A *comparison network* is an acyclic network of comparators. An input value is placed on each of its $w$ input lines. These values pass through each layer of comparators synchronously, finally leaving together on the network output wires. A comparison network with input values $x_i$ and output values $y_i$ (where all elements of the vectors $x$ and $y$ are either 0 or 1) is a valid *sorting network* if its output values are the input values sorted in descending order. The following classic theorem simplifies the process of proving (or disproving) that a given network sorts correctly.

**Theorem 4.4.1** (0-1 Principle) *If a sorting network sorts every input sequence of 0s and 1s, then it sorts any sequence of input values.*

By playing with sorting networks you can come up with some sorting algorithms. For example you can try to "simulate" *Bubble Sort* and *Insertion Sort* and you will realize that, in the context of sorting networks, they are equivalent. Or you could come up with a more efficient (but still intuitive) algorithm called *odd-even sorting*. In the remainder of the section we discuss *Bitonic Sort*, a parallel algorithm that, provided a sufficiently large amount of processors, breaks the lower bound on sorting for comparison based algorithms. The sequential time complexity of Bitonic Sort is $\mathcal{O}(n \log^2 n)$, but gets a parallel complexity of $\mathcal{O}(\log^2 n)$ with an infinite amount of processors.

**Definition 4.4.1** (Bitonic Sequence) *A* bitonic sequence *is a sequence of numbers that first monotonically increases, and then monotonically decreases, or vice versa. For example the sequence* $< 1, 4, 6, 8, 3, 2 >$ *is bitonic, the sequence* $< 9, 6, 2, 3, 5, 4 >$ *is not. Since we restrict ourselves to sequences of zeros and ones, bitonic sequences have the form* $0^i 1^j 0^k$ *or* $1^i 0^j 1^k$.

**Definition 4.4.2** (Half Cleaner) *An half cleaner is a comparison network that takes as input two bitonic sequences and returns a sequence such that:*

- *the upper and lower half are bitonic*
- *one of the halves is bitonic clean*
- *every number in upper half is lower equal than every number in the lower half*

*Hence, given a bitonic sequence, we can sort it by using half cleaners recursively.*

**Definition 4.4.3** (Bitonic Sequence Sorter) *With the help of half cleaners we can construct a bitonic sequence sorter, which sorts a bitonic sequence. The algorithm works as follows:*

1. *A bitonic sequence sorter consists of a half cleaner of width* $n$*, and then two bitonic sequence sorters of width* $n/2$ *each.*
2. *A bitonic sequence sorter of width 1 is empty.*

Clearly we want to sort arbitrary and not only bitonic sequences! To do this we need one more concept, merging networks. A merging network is simply a bitonic sequence sorter, where we replace the (first) half-cleaner by a merger. How do we sort $n$ values when we are able to merge two sorted sequences of size $n/2$? Piece of cake, just apply the merger recursively, as shown in the next figure.

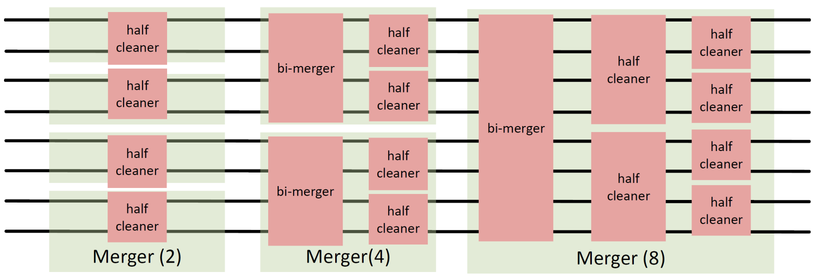


**Figure 4.3:** Bitonic Sort

The number of steps performed by the algorithm is:

$$\sum_{i=1}^{\log n} \log(2^i) \in \mathcal{O}(\log^2 n)$$

where $\log n$ is the number of mergers and $\log(2^i)$ is the number of steps performed by every merger.

## 4.5 Skip List

In this section we present a new data structure which is useful to handle a collection of elements (without duplicates) and provides an interface to add, search and remove elements. This data structure performs well when there are several searches, fewer addition and many more deletions. The advantage of this data structure (compared to other efficient data structures such as AVL trees) is that is easy to parallelize. The basic idea is to have a sorted multi-level list and the node height is probabilistic. Let's take a closer look to the sequential implementation.

Can we search in a sorted linked list in better than $\mathcal{O}(n)$ time? The worst case search time for a sorted linked list is $\mathcal{O}(n)$ as we can only linearly traverse the list and cannot skip nodes while searching. For a Balanced Binary Search Tree, we skip almost half of the nodes after one comparison with root. For a sorted array, we have random access and we can apply Binary Search on arrays. Can we augment sorted linked lists to make the search faster? The answer is *Skip List*. The idea is simple, we create multiple layers so that we can skip some nodes. See the following example list with 16 nodes and two layers. The upper layer works as an "express lane" which connects only main outer stations, and the lower layer works as a "normal lane" which connects every station. Suppose we want to search for 50, we start from first node of "express lane" and keep moving on "express lane" till we find a node whose next is greater than 50. Once we find such a node (30 is the node in following example) on "express lane", we move to "normal lane" using pointer from this node, and linearly search for 50 on "normal lane". In following example, we start from 30 on "normal lane" and with linear search, we find 50.

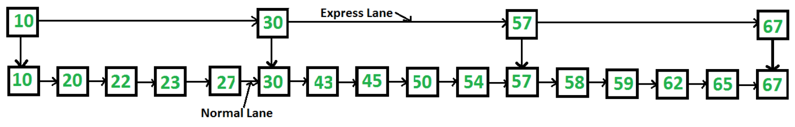


**Figure 4.4:** Example of Skip List with two levels

What is the time complexity with two layers? The worst case time complexity is number of nodes on “express lane” plus number of nodes in a segment (A segment is number of “normal lane” nodes between two “express lane” nodes) of “normal lane”. So if we have n nodes on “normal lane”, $\sqrt{n}$ nodes on “express lane” and we equally divide the “normal lane”, then there will be $\sqrt{n}$ nodes in every segment of “normal lane” . $\sqrt{n}$ is actually optimal division with two layers. With this arrangement, the number of nodes traversed for a search will be $\mathcal{O}(\sqrt{n})$. Therefore, with $\mathcal{O}(\sqrt{n})$ extra space, we are able to reduce the time complexity to $\mathcal{O}(\sqrt{n})$. The time complexity of skip lists can be reduced further by adding more layers. In fact, the time complexity of search, insert and delete can become $\mathcal{O}(\log n)$ in average case with $\mathcal{O}(n)$ extra space.

Each element in the list is represented by a node, the level of the node is chosen randomly while insertion in the list. Level does not depend on the number of elements in the node. The level for node is decided by the following algorithm:

**Algorithm 4.6**: Choose level

```
level = 1;
                                   ▷ random() that returns a random value in [0...1)
while random() < p and level < MaxLevel do
    level := level + 1
return level
```

*MaxLevel* is the upper bound on number of levels in the skip list. It can be determined as – $L(N) = \log_{p/2}(N)$. Above algorithm assure that random level will never be greater than *MaxLevel*. Here $p$ is the fraction of the nodes with level $i$ pointers also having level $i+1$ pointers and $N$ is the number of nodes in the list.

Each node carries a key and a forward array carrying pointers to nodes of a different level. A level $i$ node carries i forward pointers indexed through 0 to $i$.

## Insertion

We will start from highest level in the list and compare key of next node of the current node with the key to be inserted. Basic idea is if:

1. Key of next node is less than key to be inserted then we keep on moving forward on the same level
2. Key of next node is greater than the key to be inserted then we store the pointer to current node $i$ at *update[i]* and move one level down and continue our search

at the level 0, we will definitely find a position to insert given key.

### Searching

Searching an element is very similar to approach for searching a spot for inserting an element in Skip list. The basic idea is if:

- Key of next node is less than search key then we keep on moving forward on the same level
- Key of next node is greater than the key to be inserted then we store the pointer to current node $i$ at *update[i]* and move one level down and continue our search

At the lowest level (0), if the element next to the rightmost element (update[0]) has key equal to the search key, then we have found key otherwise failure.

### Deletion

Deletion of an element $k$ is preceded by locating element in the Skip list using above mentioned search algorithm. Once the element is located, rearrangement of pointers is done to remove element form list just like we do in singly linked list. We start from lowest level and do rearrangement until element next to *update[i]* is not $k$. After deletion of element there could be levels with no elements, so we will remove these levels as well by decrementing the level of Skip list.

## 4.6 Message Passing

In this course we have discussed several ways to ensure mutual exclusion. We talked about locks, atomic operations and transactional memory. Now we take a step back and we observe that all this techniques share a common assumptions: different proccessor (or threads) communicate by accessing the same memory. In order to ensure correctness (*i.e.* avoiding data races, bad interleavings and inconvenient program executions) we studied several ways to ensure that no two agents have access to a shared mutable resource at the same time. We particularly paid attention to write-write and read-write conflicts.

How can we exploit parallelism without the (problematic) shared memory assumption? There are two main alternatives. The first option is *functional programming*, a particular programming paradigm that works with immutable states and hence does not require synchronization. You will gain experience with this idea in the course *Functional Programming and Formal Methods* in the fourth semester. The second option is *message passing*, which we study in this section. In message passing the state is mutable, but is *not* shared among processors (or threads). Here each processor has its own memory, and hence it does not need to synchronize accesses. However, in order to make different processors cooperate, an alternative solution to shared memory is necessary. Message passing uses *messages*, which are exchanged by processors via an interconnect network.

## Message Passing Interface

In order to illustrate how message passing works, we present the *Message Passing Interface* (MPI). In order to understand MPI, we first need some preliminary concepts.

The *Actor Model* is a model for concurrent computation. *Actors* are computational agents that perform local computations and react to received messages.

**Example 4.6.1** A *distributor* is a type of actor which forwards received messages to a set of actors in a round-robin fashion. There are two questions that we need to answer to be able to model an actor. What *local state* should be kept by the actor and what should the actor do upon receiving a message? The first question can be answered quickly. We know that we have a set of actors. In order to guarantee that the messages are distributed in a round-robin fashion, we store the actors in an array and we keep an index which indicates the entry in the array which contains the next actor to send a message to. Now that we know what internal state we are going to keep, defining the behavior of the actor upon receiving a message seems obvious. The message can immediately be forwarded to the actor stored in the array entry indicated by the stored index. Then, we increment the index modular the array length. We also need to consider adding control commands which could, for example, allow us to add or remove actors from the set of stored actors.

There are different ways to communicate. On the one hand there is *synchronous vs asynchronous communication*: synchronous communication means "live" communication (*e.g.* a phone call), asynchronous communication indicates "delayed" communication (*e.g.* mail communication). On the other hand there is *blocking vs non-blocking communication*: blocking communication means "not doing anything until the message is read", while non-blocking communication means "if the message is not received, I do something else and I retry later". Those concepts are *orthogonal*, *i.e.* they don't influence each other. Concretely, we have:

- Synchronous + blocking: try to call somebody until he answers.
- Synchronous + non-blocking: try to call, if the other person does not pick up I do something else.
- Asynchronous + blocking: wait until your crush texts you back.
- Asynchronous + non-blocking: send an E-Mail and continue working until you get a response.

In the actor model, messages are sent in an *asynchronous, non-blocking fashion*, i.e. the sender places the message into the buffer of the receiver and continues execution. In contrast, when the sender sends *synchronous* messages, it blocks until the message has been received. MPI collects processes into groups, where each group can have multiple *colors*. A group paired with its color uniquely identifies a communicator. Initially, all processes are collected in the same group and communicator `MPI_COMM_WORLD`. Within each communicator, a process is assigned a unique identifier, called the *rank*.

## Point-to-Point Communication

The methods to send and receive messages are declared as follows:

```
public void Send(
    Object buf,                         ▷ Ptr to data to be sent
    int offset,
    int count,                          ▷ Number of items to be sent
    Datatype datatype,                  ▷ Datatype of items
    int dest,                           ▷ Destination process id
    int tag,                            ▷ Data id tag
);
public void Recv(
    Object buf,                         ▷ Ptr to buffer to rcv to
    int offset,
    int count,                          ▷ Number of items to be received
    Datatype datatype,                  ▷ Datatype of items
    int dest,                           ▷ Source process id
    int tag,                            ▷ Data id tag
);
```

Note that in the case of the *Recv* method it is not necessary to declare *src* or *tag*, instead one could use `MPI_ANY_SOURCE` or `MPI_ANY_TAG`. Both methods are declared in the `COMM` class, i.e. can only be used in combination with a communicator. The two methods declared above are so-called *blocking* operations, which means they will not return until the action has been completed *locally*. Their *non-blocking* variants *recv* and *send* return immediately. We can also send synchronous messages, i.e. the operation blocks until the message has been received, using *Send*. Note that the *Recv* method declared above already is synchronous. To summarize and complete what we have learned up until now, we can write a simple MPI program using the following six functions:

- `MPI.Init()`: initialize the MPI library (first routine called)
- `MPI.COMM.Size()`: get the size of a communicator `COMM`
- `MPI.COMM.Rank()`: get the rank of the calling process in the communicator
- `MPI.COMM.Send()`: Send a message to another process in the communicator
- `MPI.COMM.Recv()`: Receive a message from another process in the communicator
- `MPI.Finalize()`: Clean up all MPI state (last routine called)

**Example 4.6.2** Given an array of integers, let's compute the number of prime factors for each entry. The resulting array should be present in the process with rank 0 at the end. For simplicity, assume the array length is divisible by the number of processes.

```
public static void computePrimeFactors(int[] arr){
    int size = MPI.COMM_WORLD.Size();
    int rank = MPI.COMM_WORLD.Rank();
    int partSize = arr.length / size;
    int[] res = new int[partSize];
    for(int i = rank*partSize; j=0; i<arr.length; i++,j++){
        if(rank == 0) arr[i] = primeFactors(arr[i]);
        else res[j] = primeFactors(arr[i]);
    }
    if(rank != 0) MPI.COMM_WORLD.Send(res, 0, partSize, MPI.
    INT, 0, 42);
    else{
        for(int i = 1; i < size; i++){
            MPI.COMM_WORLD.Recv(arr, i*partSize,partSize,MPI.
    INT, i,42);
        }
    }
}
```

We see that the behaviour of the program depends on the rank of the actor. The actor with rank zero behaves differently than all other actors. Since a single program (we compile it only once) actually includes multiple programs (We can argue that rank zero executes something completely different than the other actors), we say that MPI is *Single Program Multiple Data* (SPMD).

## Group Communication

So far we considered communication between two specific processes. MPI also supports communications among groups of processes. This is not really necessary to write a program (the six methods provided above are sufficient to write any MPI program), but it is essential to performance. Here we give an overview of some methods for *group communication*:

- Broadcast: used by a processor to send a data to all other processors.
- Scatter: used by a processor to distribute an aggregate of $n$ items (*e.g.* an array) to the other $p$ processors, such that each processor has $n/p$ elements.
- Gather: used by a processor to recall the results of the computation from the other processors. A leader processor that initially holds the array can distribute the elements to the processors which, after they have performed some computation in parallel, are recalled by the leader via the gather method.
- Reduce: used to perform an operation such as sum, max, min, prod, ... to an aggregate of items. The processors perform the computation and, at the end, the processor that called reduce has the result.
- Allreduce: similar to reduce, but at the end all the processors have the result. Equivalent to a reduce followed by a broadcast.

# Bibliography

[1] L. Meinen, *PVK Skript for Parallel Programming*, 2019.

[2] M. Ghaffari, *Script of Algorithms, Probability and Computing*, Chapter 6.

[3] M. Herlihy et al, *The Art Of Multiprocessor Programming*, Morgan Kaufmann Publishers, 2014.

[4] T. Roscoe and R. Wattenhofer, *Computer Systems Script*, Chapters 5 and 19.