ETH Zürich
Parallel Programming

Lukas Wolf

Summer2019

Contents

1 Introduction

2 things to make sure in parallel programming:

1. Mutual Exclusion (Safety property): nothing bad ever happens (95% of checks, finite behaviour)

2. No Lockout (Liveness property): eventually something good happens (5% of checks in programs, infinite behaviour)

3 Stories of concurrency / parallel programming:
1. Mutual Exclusion: In computer science, mutual exclusion is a property of concurrency control, which is instituted for the purpose of preventing race conditions; it is the requirement that one thread of execution never enters its critical section at the same time that another concurrent thread of execution enters its own critical section.

2. Producer vs. Consumer: In computing, the producer–consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

3. Readers vs. Writers: In computer science, the readers-writers problems are examples of a common computing problem in concurrency. There are at least three variations of the problems, which deal with situations in which many threads (small processes which share data) try to access the same shared resource at one time. Some threads may read and some may write, with the constraint that no process may access the shared resource for either reading or writing while another process is in the act of writing to it (In particular, we want to prevent more than one thread modify the shared resource simultaneity and allow for two or more readers to access the shared resource at the same time). A readers-writer lock is a data structure that solves one or more of the readers-writers problems.

2 Threads

Threads are independent sequences of execution. Multiple threads share the same address space. They are not shielded from each other, share resources and can communicate more easily. Context switching between threads is efficient (No change of address space, no automatic scheduling, no saving or reloading of (heavyweight OS process) state.
Usage of Multithreading: Reactive systems, more responsive to user input, GUI application can interrupt a time-consuming task, server can handle multiple clients simultaneously, parallel processing.

Java Threads:
A thread is a set of instructions to be executed one at a time, in a specified order. A special Thread class is part of the core language.

Some methods of class java.lang.Thread:

1. start: method called to spawn a new thread, causes JVM to call run() method on object
2. interrupt: freeze and throw exception to thread

How to create a Java Thread (two options):

1. Instantiate a subclass of java.lang.Thread class
   Override the run method (must be overridden). run() is called when execution of that thread begins. A thread terminates when run() returns. start() method invokes run(). Calling run() does not create a new thread.
2. Instantiate a class that implements java.lang.Runnable. We need a single method: public void run(). Java does not allow multiple inheritance but multiple implementing. If we want to implement more than one interface we use this way to create a thread.

Why should we call start() instead of run() ? run is executed in the thread which calls it. start() allocates resources and starts a new thread internally.

Useful: join() function: will wait until the thread which is joined is finished with its execution.

Java Threads: some key points

1. Every Java program has at least one execution thread: First execution thread calls main()
2. Each call to start() method of a Thread object creates an actual execution thread
3. The program ends when all threads (non-daemon threads) finish

4. Threads can continue to run even if main() returns
5. Creating a Thread object does not start a thread
6. Calling run() doesn't start a thread either (need to actually call start() !)

Useful Thread attributes and methods:

1. ID: This attribute denotes the unique identifier for each Thread:
   Thread t = Thread.currentThread(); // get the current thread
   System.out.printf("Thread ID:" + t.getId()); // prints the current ID.
2. Name: this attribute denotes the name of Thread
   t.setName("PP" + 2019); // can be modified like this
3. Priority: denotes the priority of the thread. Threads can have a priority between 1 and 10: JVM uses the priority of threads to select the one that uses the CPU at each moment.
   t.setPriority(Thread.MAX_PRIORITY); // updates the threads priority
4. Status: denotes the status the thread is in: one of new, runnable, blocked, waiting, time waiting, or terminated
   if (t.getState() == State.TERMINATED) // check if thread's status is terminated

Threads Safety Hazards: Discussion
Informally: Thread Safety → program correctness. More formally, thread safety tends to refer to nothing bad ever happens irrespective of thread interleavings (a safety property). Of course, there is also the liveness property which says that: eventually something good happens. Typically as programmers we think more of safety properties. In general, ensuring safety or liveness properties requires careful design with parallel execution in mind from the beginning.

Correctness of parallel programs:
Examples of safety properties: absence of data races, mutual exclusion, linearizability, atomicity, schedule-deterministic, absence of deadlock, custom invariants (e.g., age ¿ 15).
To ensure the parallel program satisfies such properties, we need to correctly synchronize the interaction of parallel threads so to make sure they do not step on each others toes.

Share memory interaction between threads:
2 Threads may assign/read the same variable. The programmer is responsible for avoiding interleaving issues by explicit synchronization.
In Java, all objects have an internal lock (inherited from java.lang.Object). Synchronized operations lock the object. While locked, no other thread can successfully lock the object.

Synchronized Blocks:
Internally, the JVM implements synchronized by using native, operating system primitives (and low level architecture instructions, e.g., from x86). This means the implementation of synchronized will look different on different OS/architecture combinations.

synchronized (object) {...}
Uses the given object as a lock.
Enforces mutual exclusion w.r.t. to locks on the same object. As long as a thread locks the object another thread becomes blocked and must wait.
Every Java Object can act as a lock for concurrency.

Synchronized Methods:
public synchronized type name(parameters){...}
Locks the this object which calls the method. The parameters of the locked method are not locked. A synchronized method is a shorthand for wrapping the entire body of the method in a synchronized(this)... block. Useful for methods whose entire bodies should not be entered by multiple threads at the same time.

Synchronized static method:
public static synchronized type name (parameters) {...}
Locks the given (static) class, which is also an object.

Synchronized:
synchronized(r){...}
Locks the object referenced by r. If synchronized(this)..., then only one thread can enter this block at a time.

volatile keyword:
The Java volatile keyword guarantees visibility of changes to variables across threads.

Locks are recursive. A thread can request to lock an object it has already locked.

Synchronized & Exceptions:
If an exception is thrown, the synchronized on the this object will be released. The exception is caught and the exception handler is executed (if no handler then propagating to calling method). The code after the exception will not be executed. Note that any side effects from computations before the exception are not reverted, they do take effect, even if the exception is thrown.

Beyond synchronization: Wait, Notify, NotifyAll

Object.wait() : releases object lock, thread waits on internal queue. thread puts itself in the waiting list of the current object

Object.notify() : wakes the highest-priority thread closest to front of object's internal queue

Object.notifyAll() : wakes up all waiting threads; may not be fair, low-priority threads may never get access, the JVM does not guarantee who takes the lock (if same priority)

Producer / Consumer template:
wait and notify methods used within synchronized code to make sure that no other thread has called a method on the same object. wait causes the thread to give up its lock and sleep until notified: this allows another thread to obtain the lock and continue processing.
Why do we need a (while) loop and synchronized when using wait/notify?
Loop: There could be something to consume (producer returns true) shortly after the producer returned false. If producer never produces again we will resume blocked forever. In Java if wait() is called without synchronized on that object, an exception is thrown.

Java.lang.Object methods: **wait(), notify(), synchronized**
Java.lang.Thread extends Object. methods:
start(), join(),
interrupt(): Syntax: t.interrupt(); // requests a thread interruption of thread t.
An interrupt is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate. This is the usage emphasized in this lesson. A thread sends an interrupt by invoking interrupt on the Thread object for the thread to be interrupted. For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption.

Synchronizing with join():
One thread can wait (with or without a timeout) for another thread (the target) to terminate by issuing the join() method call on the target's object. After t.join(); the main thread joins t and waits for it to die.
The isAlive() method allows a thread to determine if the target thread has terminated.

Thread States: Summary
Thread is created when an object derived from the Thread class is created. At this point, the thread is not executable, it is in a New state.
Once the start() method is called, the thread becomes eligible for execution by the scheduler.

If the thread calls the wait method in an Object, or calls the join method in another thread object, the thread becomes Blocked and no longer eligible for execution.
It becomes executable as a result of an associated notify method being called by another thread, or if the thread with whic h it has requested a join, becomes dead.
A thread enters the dead state, either as a result of the run method exiting (normally, or as a result of an unhandled exception) or because its destroy method has been called.
In the latter case, the thread is abruptly moved to the dead state and does not have the opportunity to execute any finally clauses associated with its execution; it may leave other objects locked.

3 Parallel Architectures

Parallel vs. concurrent vs. distributed:
Common assumption for parallel and concurrent: "one system".
Distributed computing:
Physical separation, different domains, multiple systems.

Key concerns:
Parallelism: Use extra resources to solve a problem faster
Concurrency: Correctly and efficiently manage access to shared recources

Basic principles of today's computers:
Based on the Von Neumann architecture:
program data and program instructions in the same memory. This architecture matches imperative programming languages such as Java: statement executes, then another statement, then a third one and so on.
How can we make computations faster?
Parallel execution, i.e., additional execution units (that are acutally used). CPU cores have different level of caches. They have their own level 1 and level 2 (usually bigger) cache, and an even bigger level 3 cache that is shared by all cores.

3 approaches to apply parallelism to improve sequential processor performance:

Vectorization (exposed to the programmer):
example: simple addition of two arrays of the same size. We can iterate in 2-entry or 4-entry steps and each of the 2, respectively 4 entrys are added by a different core. There are special vectorization instructions for processors (better availability for lower level PL).

Instruction Level Paralellism (ILP) (done by processor itself):
Modern CPUs exploit ILP. The sequential program, an instruction stream ap-

pears like if the program was executed sequentially. The CPU uses pipelining, superscaling (multiple instructions per cycle / multiple functional units), Out-of-Order (OoO) execution (potentially change execution order of instructions) and speculative execution (predict results to continue execution).

Pipelining (using software libraries and techniques to transfer this hardware topic into the software layer):
Washing clothes example with different steps of the process that need resources for a fixed amount of time.

Balanced Pipeline:
All stages take the same time to process.
lead-in: time until all stages are busy.
lead-out: time until all stages are free (again).
If the latency is constant, the pipeline is balanced.

Throughput:
The throughput is the amount of work that can be done by a system in a given period of time. In CPUs: number of instructions completed per second. The larger the better.
Pipeline throughput approx. $= \frac{1}{max(computationtime(ofallstages)}$
(ignoring lead-in and lead-out time in pipeline with large number of states)
The computation time per stage probably has to be divided by the number of execution units.

Latency:
The Latency is the time to perform a computation (e.g., a CPU instruction). How long does it take to execute a single computation (CPU instruction) in the pipeline. Pipeline latency only constant over time if pipeline is balanced: sum of execution times of each stage. Lower latency is better.
upper bound: number of stages * max(time-per-stage)

Unbalanced Pipeline:
Each stage takes different time to process / pass. With an unbalanced pipeline we cannot bound the latency.
How to balance:
We can make a pipeline balanced by increasing time for each stage to match longest stage. This is a bit wasteful, but the latency is bound.
We can take the longest stage and cut it into more than one stage such that each stage takes the same time.

Throughput vs. Latency:
Throughput optimization may increase the latency. Example: pipelining can add overheads if stages must now synchronize and communicate. $\rightarrow$ Time it takes to get one complete task through the pipeline may take longer than with

a serial implementation. Increases in parallelism can hide latency.

CPU Pipeline:
Multiple stages: 1. Instruction Fetch (fetches the instructions from the memory) - 2. Instruction Decode (prepares the instruction for execution) - 3. Execution (actual computation, e.g., addition or shifting) - 4. Data Access (writes to memory if needed) - 5. Writeback (writes to CPU registers)
Parallelism (multiple hardware functional units) leads to faster execution of sequential programs.

For a long time CPU architects improved sequential execution by exploiting Moore's law and ILP, but now archtitecture hits walls: power dissipation wall (expensive to cool), memory wall (CPUs faster than memory access) and ILP wall (limits in inherent program's ILP; complexity). Therefore it is no longer affordable to increase sequential CPU performance.

Parallel Architectures:
Shared / Distributed memory architectures:
Simultaneous multithreading, multicores, symmetric multiprocessor system (SMP), non-uniform memory access (NUMA) (Lecture 4 Slides)

Summary:
Parallelism is used to improve performance (at all levels). Architects cannot improve sequential CPU performance anymore. Nowadays (multicore era) programmers need to write parallel programs.

4 Basic Concepts in Parallelism

Work partitioning means splitting up work of a single program into parallel tasks. This can be done explicitly (i.e., manually) (task/thread parallelism). Implicit parallelism is done automatically by the system.
We want to split up the work into parallel tasks / threads and schedule them such that no processor is ever idle (scheduling is typically done by the system).

Task/Thread Granularity:
Fine granularity of tasks is more portable (can be executed in machines with more processors) and is better for scheduling.
But: if scheduling overhead is comparable to a single task, then the overhead dominates.
Guideline: As small as possible but significantly bigger than scheduling overhead.

Scalability:
An overloaded concept: e.g., how well a system reacts to increased load, like clients on a server. In parallel programming we calculate the speedup (scalability) when we increase the processors. What will happen if processors $\rightarrow \infty$?
If a program scales linearly, then we have linear speedup.

Parallel Performance:
Execution time:
T_p on p CPUs:
Sequential execution time: T_1
$T_p = T_1/p$ (perfection)
$T_p > T_1/p$ (performance loss, what normally happens)
$T_p < T_1/p$ (sorcery!)
Parallel Speed-up (absolute speed-up):
(parallel) speedup S_p on p CPUs:
$S_p = T_1/T_p$
$S_p = p$ (linear speedup)
$S_p < p$ (sub-linear speedup)
$S_p > p$ (super-linear speedup, sorcery)
Efficiency (relative speed-up per processor):
S_p/p
Absolute versus relative Speed-up
Relative speedup (efficiency):
relative improvement from using P execution units (baseline: serialization of the parallel algorithm). Sometimes there is a better serial algorithm that does not parallelize well. In these cases it is fairer to use that algorithm for T_1 (absolute speedup). Using an unnecessarily poor baseline artificially inflates speedup and efficiency.
Why $S_p < p$?
Programs may not contain enough parallelism, e.g. some parts of the program might be sequential. Overheads can be introduced by parallelization (typically associated with synchronization). We also have architectural limitations, e.g., memory contention.

Amdahl's Law
... the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude.
W_{ser} : time spent doing non-parallelizable serial work
W_{par} : time spent doing parallelizable work
Given P workers available to do parallelizable work, the times for sequential execution and parallel execution are:
$T_1 = W_{ser} + W_{par}$
Therefore we get the following bound on speed-up:
$T_P \geq W_{ser} + \frac{W_{par}}{P}$

Plugging these relations into the definition of speedup yields Amdahl's Law:
$S_p = \frac{W_{ser}+W_{par}}{W_{ser}+\frac{W_{par}}{P}}$

Corollary:
If f is the non parallelizable serial fraction of the total work, then the following equalities hold:
$W_{ser} = f \cdot T_1$
$W_{par} = (1-f) \cdot T_1$
which gives:
$S_p \leq \frac{1}{f+\frac{1-f}{P}}$
What happens if we have infinite workers: $S_\infty \leq \frac{1}{f}$
Amdahl's law means "get same work done in less time".

Remarks about Amdahl's Law
It concerns maximum speedup (Amdahl is optimist), architectural constraints will make factors worse. But this law is mostly bad news (as it puts a limit on scalability).
Takeaway: All non-parallel parts of a program (no matter how small) can cause problems.
Amdahl's law shows that efforts required to further reduce the fraction of the code that is sequential may pay off in large performance gains. Hardware that achieves even a small decrease in the percent of things executed sequentially may be considerably more efficient.

Gustavson's Law
An alternative (optimistic) view to Amdahl's Law)
Observations: Consider problem size. For this law, the run-time, not the problem size is constant. More processors allow to solve larger problems in the same time. The parallel part of a program scales with the problem size.
Gustavson's law means: "get more work done in the same time - raise the parallel work".
f: sequential part (no speedup)
$W = p(1-f) \cdot T_{wall} + f \cdot T_{wall}$
$S_p = \frac{W_P}{W_1} = f + p \cdot (1-f) = p - f \cdot (p-1)$
Therefore Gustavsons law is proportional to p.

Amdahl's law sees the percentage of non-parallelizable code as a fixed limit for the speedup. So even if we had an infinite amount of processors, according to Amdahl's law, the speedup would never be greater than 2.
On the other hand Gustafson's law assumes that the parallel part of the program increases with the problem size and the sequential part stays fixed.

5 Divide and Conquer, Cilk-style bounds

Divide-and-conquer can be implemented to parallelize recursive calls. A fundamental pattern in parallel programming is called recursive splitting. It means you divide the problem until a unitary solution can be calculated (and is returned) and combine the solutions afterwards.
Java threads are actually quite heavyweight. Java threads are mapped to OS threads. In general using one thread per (small task) is highly inefficient.
In theory, you can divide down to single elements, do all your result-combining in parallel and get optimal speedup. In practice, creating all those threads and communicating swamps the savings. Therefore we have to:
Use a sequential cutoff, typically around 500-1000. This eliminates almost all the recursive thread creation (bottom levels of tree).
Do not create two recursive threads. Create one and do the other half "yourself" (e.g. call "run" method which does not create an actual thread). This cuts the number of threads created by another 2x.

The key is divide-and-conquer parallelizes the result-combining. If you have enough processors, the total time is height of the tree: $O(log(n))$ (optimal, exponentially faster than sequential). Often relies on operations being associative (like +). We will write all our parallel algorithms in this style, but we will use special libraries engineered for this style.

Alternative approach:
schedule tasks on threads with a threadpool. Java's executor service offers managing asychnronous tasks.

Java's Executor Service:
ExecutorService is a framework provided by the JDK which simplifies the execution of tasks in asynchronous mode. Generally speaking, ExecutorService automatically provides a pool of threads and API for assigning tasks to it.
The ExecutorService is good to use for independent tasks (no deadlock can occur).
Tasks can be assigned to the ExecutorService using several methods, including execute(), which is inherited from the Executor interface, and also submit(), invokeAny(), invokeAll(). The execute() method is void, and it doesn't give any possibility to get the result of task's execution or to check the task's status (is it running or executed).
submit() submits a Callable or a Runnable task to an ExecutorService and returns a result of type Future.
The shutdown() method doesn't cause an immediate destruction of the ExecutorService. It will make the ExecutorService stop accepting new tasks and shut down after all running threads finish their current work.
The submit() and invokeAll() methods return an object or a collection of objects of type Future, which allows us to get the result of a task's execution or to check the task's status (is it running or executed). The Future interface

provides a special blocking method get() which returns an actual result of the Callable task's execution or null in the case of Runnable task. Calling the get() method while the task is still running will cause execution to block until the task is properly executed and the result is available.
After the release of Java 7, many developers decided that the ExecutorService framework should be replaced by the fork/join framework. This is not always the right decision, however. Despite the simplicity of usage and the frequent performance gains associated with fork/join, there is also a reduction in the amount of developer control over concurrent execution.
ExecutorService gives the developer the ability to control the number of generated threads and the granularity of tasks which should be executed by separate threads. The best use case for ExecutorService is the processing of independent tasks, such as transactions or requests according to the scheme "one thread for one task."

Note: Callable vs. Runnable:
If we implement the interface Runnable, we override the run method, which is void and does not return a result. If we implement the Interface Callable, we get a return value. We submit a task and can save a value, e.g. of the type Future(e.g., Integer).

How to use the ExecutorService:
First we create a new ExecutorService with a fixed number of threads like:
ExecutorService executor = Executors.newFixedThreadPool(int number of threads);
Then we have to create a Callable object, e.g. a Callabe Integer object. In this class, we have to override the call method which starts the work and returns a variable of our type (here: Integer).
We then create a Future(Integer) reference and submit our task:
Future(Integer) future = executor.submit(callableTask);
To save the result, we create a new (Integer) variable and assign: future.get(). this method waits until the executorservice finished the task.
The future.get() has to be called after the other half is computed by the current thread itself. future.get() waits until it the CallableTask is finished, therefore otherwise we would not have concurrency at all. In the end, we try executor.shutdown(), catch an InterruptedException and finally execute executor.shutdownNow().
More can be found in the solution 5 of the exercises.

Task Parallel Programming (Cilk-style)
Tasks: execute code, spawn other tasks, wait for results from other tasks. We get a task graph which is a DAG (directed acyclic graph). Edges mean that tasks were created by each other. Tasks can be executed in parallel, but they don't have to. The assignment of tasks to CPUs/cores is up to the scheduler. The task graph is dynamic and unfolds as the execution proceeds.
Intuition: wider task graph $\rightarrow$ more parallelism.

T_∞: span, critical path; time it takes on infinite processors, longest path from root to sink
Lower bounds:
$T_p \geq T_1/p$
$T_p \geq T_\infty$

The scheduler is an algorithm for assigning tasks to processors. Note that T_p depends on the scheduler, whereas T_1/P and T_∞ are fixed.
A bound on how fast you can get on p processors with a greedy scheduler is: $T_p \leq T_1/P + T_\infty$.

Work stealing scheduler:
First used in MIT's Cilk, now a standard method.
Provably: $T_p = T_1/P + O(T_\infty)$, Empirically: $T_p \approx T_1/P + T_\infty$

6 ForkJoin Framework & Task Parallel Algorithms

We need a new framework that supports Divide and Conquer style parallelism, since Java's ExecutorService gets deadlocked sometimes. That is, when a task is waiting, it should be suspended (from a thread) and other tasks are allowed to run.
ForkJoin is used for dependent tasks.

How to use the ForkJoin Framework:
The ForkJoinPool is the heart of the framework. It is an implementation of the ExecutorService that manages worker threads and provides us with tools to get information about the thread pool state and performance. Worker threads can execute only one task at the time, but the ForkJoinPool doesn't create a separate thread for every single subtask. Instead, each thread in the pool has its own double-ended queue (or deque, pronounced deck) which stores tasks. This architecture is vital for balancing the thread's workload with the help of the work-stealing algorithm.

Work stealing algorithm:
Simply put – free threads try to "steal" work from deques of busy threads. By default, a worker thread gets tasks from the head of its own deque. When it is empty, the thread takes a task from the tail of the deque of another busy thread or from the global entry queue, since this is where the biggest pieces of work are likely to be located. This approach minimizes the possibility that threads will compete for tasks. It also reduces the number of times the thread will have to go looking for work, as it works on the biggest available chunks of work first.

Different Terms to Threads:
Instead of subclassing Thread we have to subclass RecursiveTask (type V). Then we override the compute() method from which we return a V. To start the task we call fork() on it. Calling join() will return the answer. If we want to hand-optimize, we can directly call compute(). First of all, we create a pool of tasks and then we call invoke() to execute each task.
There are two types of a ForkJoinTask: the RecursiveTask returns a value, the RecursiveAction does not.
The ForkJoinPool() contructor creates a number of threads equal to the number of available processors.

How to do it:
First we need to create a Task (which extends e.g. RecursiveTask):
SearchAndCountForkJoin countapp = new SearchAndCountForkJoin(input, 0, input.length, cutOff, wt); Then we create a ForkJoinPool with a fixed number of threads:
ForkJoinPool fjp = new ForkJoinPool(noThreads);
Then we call invoke on our task and save the result:
int result = fjp.invoke(countapp);
In the compute function of our task, we divide the problem further. We fork() one of the two tasks and call compute() on one half (direct call) and join() on the other (forked) one. Then we return the result.

Results:
If we do experiments with ForkJoin we may observe that it performs poorly. The reasons have to do with the ForkJoin framework implementation and how it was retrofitted to Java. We have to make sure that each task has 'enough' to do. The recommended sequential threshold is 100-5000 basic operations in each 'piece' of the algorithm. The library needs to warm up and we may see slow results before the JVM re-optimizes the library internals.

Examples on when to use ForkJoin:
Maximum or minimum element, is there an element satisfying some property (e.g., first 17), left-most element satisfying a property, counting.
Computations of this form are called reductions. They produce a single answer from a collection via an associative operator. (Recursive) results don't have to be single numbers or strings, they can be arrays or objects with multiple fields.

DAG representation of ForkJoin:
A program execution using fork an join can be seen as a DAG. Nodes: pieces of work, edges: source must finish before destination starts. A fork "ends a node" and makes to outgoing edges (new thread). A join "ends a node" and makes a node with two incoming edges (task just ended).

Runtime Analysis:

An asymptotically optimal execution would be: $T_p = O((T_1/p) + T_\infty)$. The first term dominates for small p, the second one for large p. The FJ framework gives an expected-time guarantee of asymptotically optimal.

Maps (Data Parallelism):
A map operates on each element of a collection independently to create a new collection of the same size. Results are not combined. For arrays, this is so trivial that some hardware has direct support (e.g., vector addition).
Maps and reductions are by far the two most important and common parallel programming patterns.

Pack
Pack is a non-standard terminology. Given an array input, produce an array output containing only elements such that f(elt) is true. example: f: is elt > 10.
The parallel prefix example can be used to compute solutions for this and similar problems. The first pass builds a tree bottom-up: the "up" pass. The second pass traverses the tree top-down: the "down" pass.
Analysis: O(n) work, O(log(n)) span.
Parallelized packs will help us parallelize quicksort.

7 Data Races

The stategy with ForkJoin won't work if memory accessed by threads is overlapping or unpredictable. Threads are doing independent tasks needing access to same resources (rather than implementing the same algorithm).

Approaches:
Immutability: Data does not change, best option, should be used when possible.
Isolated Mutability: Data can change, but only one thread / task can access them.
Mutable / Shared Data: Data can change / all tasks / threads can potentially access them; present in shared memory architectures

Dealing with mutable / shared state:
locks: mechanism to ensure exclusive access / atomicity (ensuring good performance / correctness with locks can be hard)
transactional memory: programmer describes a a set of actions that need to be atomic (easier for the programmer, but getting good performance might be challenging)

Interleaving:
If second call starts before first ends, we say the calls interleave. This could happen even with one processor since a thread can be pre-empted at any point for time-slicing.

Mutual exclusion:
One thread using a resource (e.g., a bank account) means that another thread must wait (a.k.a. critical sections, which technically have other requirements). Mutual exclusion means an algorithm that implements a critical section (piece of code that may be executed by at most one thread).

Lock Object:
Shared object that satisfies the Lock interface:
public interface Lock {
public void lock();
public void unlock();
}

Lock semantics:
new Lock: make a new lock, initially "not held"
acquire: blocks (only) if this lock is already currently "held", once "not held", makes lock "held"
release: makes this lock "not held", if >= 1 threads are blocked on it, exactly 1 will acquire it
How can this be implemented? Uses special hardware and OS support.

Example: Bank Account
One lock object for each account. The lock locks the object from which it is called (bank account of that person).
Possible mistakes: Use different locks for withdraw and deposit, use same lock for every bank account, forget to release a lock (blocks other threads forever).

Re-entrant Lock:
Remembers the thread that currently holds it. When the lock goes from not-held to held, the count is set to 0. If (code running in) the current holder calls acquire, it does not block, it increments the count. On release, if the count is > 0, the count is decremented, if the count is 0, the lock becomes not held.
Java has some built-in support for re-entrant locks: focus on the synchronized statement.
Java Class: java.util.concurrent.locks.ReentrantLock
Often use try ... finally to avoid forgetting to release the lock if there's an exception

Races:
A Race condition occurs when the computation result depends on the scheduling (how threads are interleaved).
Note: In this lecture there is a big distinction between data races and bad interleavings, both kinds of race-condition bugs.
Data Race (low level race condition / low semantic level):
Erroneous program behavior caused by insufficiently synchronized accesses of a

shared resource by multiple threads, e.g. simultaneous read/write or write/write of the same memory location. Always an error, due to compiler and HW.
Bad Interleaving (high level race condition / high semantic level):
Erroneous program behavior caused by an unfavorable execution order of a multithreaded algorithm that makes use of otherwise well synchronized resources. "Bad" depends on your specificatoin.

For every memory location in your program, you must obey at least one of the following:

1. Thread-local: Do not use the location in > 1 thread
2. Immutable: Do not write to the memory location
3. Synchronized: Use synchronization to control access to the location

Lock Granularity:
Coarse-grained: fewer locks, i.e., more objects per lock. Examples: one lock for entire data structure (e.g., array), one lock for all bank accounts.
Fine-grained: more locks, i.e., fewer objects per lock. Examples: one lock per data element (e.g. array index), per lock per bank account.

Trade-Offs:
Coarse-grained advantages: simpler to implement, faster / easier to implement operations that access multiple locations (because all guarded by the same lock), much easier: operations that modify data-structure shape
Fine-grained advantages: more simultaneous access (performance when coarse-grained would lead to unnecessary blocking)

A second, orthogonal granularity issue is the critical section size (how much work to do while holding locks). If the critical section runs too long, we have a performance loss because other threads are blocked. If critical sections are too short, we can get bugs because you broke up something where other threads should not be able to see intermediate state. We can also get an performance loss because of frequent thread switching and cache trashing.

Guidelines for how to use locks

1. No Data Races: Never allow two threads to read/write or write/write the same location at the same time. Do not make any assumptions on the orders of reads or writes.
2. For each location needing synchronization, have a lock that is always held when reading or writing the location. The same lock can (and often

should) guard multiple locations. In Java, often the guard is the object containing the location.

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

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

5. Think in terms of what operations need to be atomic. Typically we want ADT operations atomic, even to other threads running operations on the same ADT. Make critical sections just long enough to preserve atomicity.

Summary of first half of the semester
Java Threads: wait, notify, join, synchronized, producer - consumer
Parallelism: vectorization, ILP, pipelining, latency / throughput
Concepts: T_1, speedup / efficiency, Amdahl / Gustafson, T_P, T_∞, Task Graphs / DAGs
Divide-And-Conquer: Threads, ForkJoin, Prefix Scan
Races: high / low level races

8 Data Races, Solving Mutual Exclusion with Atomic Registers

Why mutual exclusion can still fail:
Rule of thumb: compiler and hardware are allowed to make changes that do not affect the semantics of a sequentially executed program. Modern compilers do not give guarantees that a global reordering of memory accesses is provided. This gives a huge potential for optimizations - and for errors, when you make wrong assumptions.

Memory models:
The exact behavior of threads interacting via shared memory usually depends on hardware, runtime system and programming language. A memory model (e.g., of a programming language like Java) provides (often minimal) guarantees for the effects of memory operations. This leaves open optimization possibilities for hardware and compiler but including guidelines for writing correct multi-threaded programs.

Volatile fields:
Java has volatile fields, accesses to volatile fields do not count as data races. They are implemented slower than regular fields, but faster than locks. They are really for experts, we should avoid to use them and use standard libraries instead. Volatile tells the compiler that the order of access should not be changed. A volatile field access is not interchanged with a normal field access as well.

Architectural memory models
The compiler, the CPU and the DRAM reorders the instructions that are to be executed for better performance. It depends on the hardware what will be reordered, e.g., AMD86 is different than ARM. For example Intels x86 architecture allows loads reordered after loads, stores reordered after stores (and more), but no atomic reordering.

Java Memory Model (JMM):
We saw that if we don't have these operations (volatile, synchronized, etc.), the outcome can be arbitrary / unexpected. Roughly we can say that memory operations will not be reordered with respect to accesses to volatile variables or synchronized blocks.

Executions combine actions with ordering:

1. Program order
2. Synchronizes-with
3. Synchronization Order
4. Happens-before

JMM: Program Order (PO): order in which statements are executed (meaning the actions resulting from statements)
Order in which statements are executed (meaning the actions resulting from statements). The program order is a total order of intra-thread actions, this means per thread, the PO order is consistent with the threads isolated execution.

JMM: Synchronization Actions (SA) and Synchronization Order (SO):
Synchronization Actions (SA) are: order of synchronizing memory actions (in the same thread)

1. read / write of a volatile variable
2. lock / unlock monitor
3. first / last action of a thread (synthetic)
4. actions which start a thread
5. actions which determine if a thread has terminated

Synchronization Actions form the Synchronization Order (SO): SO is the order of synchronizing memory actions (in the same thread)
SO is a total order, all threads see SA in the same order, SA within a thread are in PO, SO is consistent: all reads in SO see the last writes in SO.

JMM: Synchronizes-With (SW) / Happens-Before (HB) orders: SW: order of observed synchronizing memory across threads, HB: the union of PO and SW
SW is the order of observed synchronizing memory actions across threads.
SW only pairs the specific actions which "see" each other. A volatile write to x synchronizes with subsequent read of x (subsequent in SO). The transitive closure of PO and SW forms HB. HB consistency: when reading a variable, we see either the last write (in HB) or any other unordered write (this means races are allowed).

9 Behind Locks: Implementation of Mutual Exclusion

Critical Sections
Pieces of code with the following conditions:

1. Mutual exclusion: statements from critical sections of two or more processes must not be interleaved
2. Freedom from deadlock: if some processes are trying to enter a critical section then one of them must eventually succeed
3. Freedom of starvation: if any process tries to enter its critical section, then that process must eventually succeed

State space diagram
Shows possible transitions between states and if mutual exclusion is supported or if a piece of code is free of starvation.

Dekker's Algorithm (for mutual exclusion):
If two processes attempt to enter a critical section at the same time, the algorithm will allow only one process in, based on whose turn it is. If one process is already in the critical section, the other process will busy wait for the first process to exit. This is done by the use of two flags, $wantsToEnter[0]$ and $wantsToEnter[1]$, which indicate an intention to enter the critical section on the part of processes 0 and 1, respectively, and a variable turn that indicates who has priority between the two processes.
Dekker's algorithm can be expressed in pseudocode, as follows.

```
    variables
        wants_to_enter : array of 2 booleans
        turn : integer

    wants_to_enter[0] ← false
    wants_to_enter[1] ← false
    turn ← 0   // or 1
```

```
p0:
   wants_to_enter[0] ← true
   while wants_to_enter[1] {
      if turn ≠ 0 {
         wants_to_enter[0] ← false
         while turn ≠ 0 {
           // busy wait
         }
         wants_to_enter[0] ← true
      }
   }

   // critical section
   ...
   turn ← 1
   wants_to_enter[0] ← false
   // remainder section
```

```
p1:
   wants_to_enter[1] ← true
   while wants_to_enter[0] {
      if turn ≠ 1 {
         wants_to_enter[1] ← false
         while turn ≠ 1 {
           // busy wait
         }
         wants_to_enter[1] ← true
      }
   }

   // critical section
   ...
   turn ← 0
   wants_to_enter[1] ← false
   // remainder section
```

Peterson Lock:
More concise than Dekker. See pseudocode.

```
bool flag[2] = {false, false};
int turn;
```

```
P0:       flag[0] = true;
P0_gate: turn = 1;
         while (flag[1] == true && turn ==
1)
         {
             // busy wait
         }
         // critical section
         ...
         // end of critical section
         flag[0] = false;
```

```
P1:       flag[1] = true;
P1_gate: turn = 0;
         while (flag[0] == true && turn ==
0)
         {
             // busy wait
         }
         // critical section
         ...
         // end of critical section
         flag[1] = false;
```

Atomic register:
Register: basic memory object, can be shared or not, i.e., in this context a register is not a register of a CPU.
For atomic registers an invocation of read or write takes effect at a single point in time.
Assumptions for atomic registers justify to treat operations on them as events taking place at a single point in time.

Filter Lock (inefficient):
Extension of Peterson's lock to n processes. Every thread t knows his level in the filter level[t]. In order to enter a critical section (CS), a thread has to elevate all levels. For each level, we use Peterson's mechanism to filter at most one thread, if other threads are at higher level. For every level l there is on victim victim[l] that has to let others pass in case of conflicts. See algorithm and lecture 15 for additional lock methods.
Remember: Filter lock is not fair, it is not FIFO.

```
class FilterLock{
  AtomicIntegerArray level;
  AtomicIntegerArray victim;
  volatile int n;
  FilterLock(int n) {
   this.n = n;
   level = new AtomicIntegerArray(n);
   victim = new AtomicIntegerArray(n);
  }
}
Pseudocode for lock and unlock:
lock(me) {
  for (int i = 1; i < n; i++)
   level[i] = i;
   victim[i] = me;
   while(there is k ≠ me : level[k] >= i  victim[i] == me);
  }
}
unlock(me) {
  level[me] = 0;
}
```

Algorithm 1: Filter Lock Java code

Safe SWMR (Single Writer Multiple Reader Register)
Allows only one concurrent write but multiple conucrrent reads. Any read not concurrent with a write returns the current value of r. Any read concurrent with a write can return any value of the domain of r. The notion "safe" is historically motivated but actually misleading.

Bakery Algorithm: Mutual exclusion for n processes:
A process is required to take a numbered ticket with value greater than all outstanding tickets. CS entry when own ticket number is the lowest.

Bakery algorithm (two processes, simplified)

```
volatile int np = 0, nq = 0
```

```
Process P
loop
    non-critical section
    np = nq + 1
    while (nq != 0 && nq < np);
    critical section
    np = 0
```

Q also wants access

and Q has an earlier ticket

```
Process Q
loop
    non-critical section
    nq = np + 1
    while (np != 0 && np <= nq)
    critical section
    nq = 0
```

np == nq can happen ➔ global ordering of processes

Theorem:
If S is a (atomic) read/write system with at least two processes and S solves mutual exclusion with global progress (deadlock-freedom), then S must have at least as many variables as processes.

10 Beyond Locks

Hardware Support for Parallelism (atomics):
Typical instructions: Test-And-Set (TAS), Compare-and-swap (CAS), Load Linked.

Semantics of TAS and CAS
TAS and CAS are read-modify-write (atomic) operations. They enable implementation of a mutex with O(1) space (in contrast to filter lock, bakery lock, etc.). These operations are needed for lock-free programming. See for test-and-set (TAS) lock and test-and-test-and-set (TATAS) lock.

TASLock in Java

```
public class TASLock implements Lock {
   AtomicBoolean state = new
AtomicBoolean(false);

   public void lock() {
      while(state.getAndSet(true)) {}
   }

   public void unlock() {
      state.set(false);
   }
   ...
}
```

Test-and-Test-and-Set (TATAS) Lock

```
public void lock()
{
   do
      while(state.get()) {}
   while (!state.compareAndSet(false, true));
}

public void unlock()
{
   state.set(false);
}
```

Spinlock:
Try to get the lock until the lock is acquired

Lock with backoff
Probably the best lock in this lecture.

Lock with Backoff

```
public void lock() {
    Backoff backoff = null;
    while (true) {
        while (state.get()) {};       // spin reading only (TTAS)
        if (!state.getAndSet(true))   // try to acquire, returns previous val
            return;
        else { // backoff on failure
            try {
                if (backoff == null)   // allocation only on demand
                    backoff = new Backoff(MIN_DELAY, MAX_DELAY);
                backoff.backoff();
            } catch (InterruptedException ex) {}
        }
    }
}
```

exponential backoff

```
class Backoff
{...
      public void backoff() throws InterruptedException {
            int delay = random.nextInt(limit);
            if (limit < maxDelay) { // double limit if less than max
                  limit = 2 * limit;
            }
            Thread.sleep(delay);
      }
}
```

Deadlocks
A deadlock for threads $T_1, ..., T_n$ occurs, when the directed graph describing the relation of the threads and resources $R_1, ..., R_m$ contains a cycle (edge from thread to resource: thread wants to acquire, edge from resource to thread: thread holds this resource).
Deadlock detection in systems is implemented by finding cycles in the dependency graph.
Deadlock avoidance amounts to techniques to ensure a cycle can never arise: either two phase locking with retry (release when failed)(usually in databases where transactions can be aborted without consequence), or resource ordering (usually in parallel programming where global state is modified).
Programming trick: if no globally unique order is available, then generate one. For each object that is created, give each object an index and then increment a counter (atomic).

Global Ordering of resources:
Often we need a unique global ordering to avoid deadlocks. The whole program has to obey this order to avoid cycles. We can easily generate a global order using an AtomicLong counter which gives each object (e.g., bank account) an unqiue index.

Semaphores
Locks lack the means for threads to communicate about changes. Thus, they provide no order and are hard to use.
A semaphore is an integer-valued abstract data type with some initital value $s \geq 0$ and the following (atomic) operations:
aquire(S): wait until $S > 0$, dec(S)
release(S): inc(S)

Example: Rendezvous with semaphores:
synchronize processes P and Q at one location, after both processes arrived, we continue (or one of them).

Implementing Semaphores without Spinning (blocking queues)

Consider a process list Q_S associated with semaphore S

acquire(S)

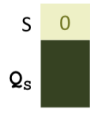
atomic

```
{if S > 0 then
    dec(S)
 else
    put(Qs, self)
    block(self)
 end }
```

release(S)

atomic

```
{if Qs == ∅ then
    inc(S)
 else
    get(Qs, p)
    unblock(p)
 end }
```



Barrier

Synchronize a number of processes. Barrier will be opened if and only if all processes have reached the barrier. The count provides the number of processes that have passed the barrier. When all processes have reached the barrier then all waiting processes can continue. The two-phase barrier prevents overtaking.

Solution: Two-Phase Barrier

init

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

barrier

```
acquire(mutex)
  count++;
  if (count==n)
     acquire(barrier2); release(barrier1)
release(mutex)

acquire(barrier1); release(barrier1);
// barrier1 = 1 for all processes, barrier2 = 0 for all processes
acquire(mutex)
  count--;
  if (count==0)
      acquire(barrier1); release(barrier2)
signal(mutex)

acquire(barrier2); release(barrier2)
// barrier2 = 1 for all processes, barrier1 = 0 for all processes
```

Producer-Consumer Pattern

Thread0 computes X and passes it to Thread1, which uses X. We do not need synchronization on X, because at any point in time only one thread accesses X. However, we need a synchronized mechanism to pass X.
For multiple producers and consumers we can use queues, e.g., a bounded FIFO as circular buffer.

Monitors

A monitor is an abstract data structure (or simply a java object) equipped with a set of operatoins that run in mutual exclusion.
Monitors provide, in addition to mutual exclusion, a mechanism to check conditions with the following semantics:
if a condition does not hold, 1. release the monitor lock, 2. wait for the condition to become true, 3. signaling mechanism to avoid busy-loops (spinning).
Monitors in Java use the instrinsic lock (synchronized) of an object. We can use any Java object as a monitor. We can call several methods on a monitor object such as wait, notify, notifyAll and more. Wait makes a thread wait on the object's monitor. Notify wakes a single thread that is waiting on this object's monitor, notifyAll wakes all threads and one wins.
Monitor queues have different states: if wait is called, the thread goes to the waiting condition. If the thread is notified, it goes to the waiting entry.

There are several semantics of monitors:
1. signal and wait: signaling process exits the monitor (goes to waiting entry queue), signaling process passes monitor lock to signaled process
2. signal and continue: signaling process continues running, signaling process moves signales process to waiting entry queue
other semantics: signal and exit, signal and urgent wait, ...

Producer / Consumer
If we implement a producer / consumer queue we have to be careful implementing the enqueue and dequeue methods. We can either use the Thread.sleep() method in the enqueue method if the queue is full. We can also implement a queue using semaphores nonEmpty, nonFull and a binary manipulation semaphore to control the concurrent access.
Producer / Consumer can also be implemented using monitors or conditions on a lock. We then have 3 different states/groups: 1. the monitor (only one object can obtain), 2. waiting for condition to become true, 3. waiting entry (waiting to get the monitor)

Condition Interface
Java locks provide conditions that can be instantiated. Conditions are always associated with a lock. Java conditions offer:
1. .await() - the current thread waits until condition is signaled (lock must be held to call this method), atomically releases the lock and waits until thread is signaled. When returns, it is guaranteed to hold the lock. Thread always needs to check condition.
2. .signal() - wakes up one thread waiting on this condition (called with the lock held)
3. signalAll() - wakes up all threads waiting on this condition (called with the lock held)

Guidelines for using condition waits:
Always have a condition predicate, always test the condition predicate, always call wait in a loop, ensure state is protected by lock associated with the condition.

11 Readers-Writers Locks

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.
A readers-writers lock allows multiple readers or a single writer to enter the critical section concurrently. Therefore we need a readLock and a writeLock which operate differently.
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.
In Java the synchronized statement does not support reader / writer. Instead, we have the ReentrantReadWriteLock. This uses a different interface in which the methods readLock an writeLock return objects that themselves have lock and unlock methods.

12 Lock Granularity

The Five-Fold Path: Coarse-grained, Fine-grained, Optimistic Synchronization, Lazy Locking, Lock-Free-Sync

Coarse-grained Locking
Ensure that each method call acquires and releases the given lock. It often works well but there are important cases where it does not. Coarse-Grained locking works well when the levels of concurrency are low. If many threads try to access the object, then the object becomes a sequential bottleneck, forcing threads to wait in line for access.

Fine-grained synchronization
Instead of using a single lock to synchronize every access to an object, we split the object into independently synchronized components, ensuring that method calls interfere only when trying to access the same component at the same time. Therefore we do not have mutex for threads on disjoint pieces.
Hand-over-hand-locking is an example for fine-grained-sync.
Disadvantages: we have a potentially long sequence of acquire/release calls before the intended action takes place. One slow thread locking early nodes may block another thread wanting to acquire later nodes.

Optimistic synchronization
One way to reduce the cost of fine-grained locking is to search without acquiring any locks at all. If the method finds the sought-after component, it locks that component, and then checks that the component has not changed in the interval between when it was inspected and when it was locked (validate method).
This technique is worthwhile only if it succeeds more often than not, which is why we call it optimistic.
Idea:
1. Find nodes without locking, lock them, 2. validate that everything is okay (not trivial to check)
Good: no contentions on traversals, traversals are wait-free, less lock acquisitions
Bad: need to traverse list twice, contains method needs to acquire locks, not starvation free

Lazy synchronization
Sometimes it makes sense to postpone hard work. For example, the task of removing a component from a data structure can be split into two phases: the component is logically removed simply by setting a tag bit, and later, the component can be physically removed by unlinking it from the rest of the data structure.
It is similar to Optimistic synchronization but instead of locking we use a mark bit (i.e., AtomicBoolean).
We perform the following Ops:
1. Scan list (as before), 2. Lock pred and curr, 3. Logical delete (mark curr node), 4. Physical delete (redirect pred's next)
Therefore the main difference to optimistic sync is that in our validate method we do not rescan but only check marked bits.

Nonblocking synchronization
Sometimes we can eliminate locks entirely, relying on built-in atomic operations such as CAS or TAS for synchronization.

13 Skip Lists

A skip list is a practical representation for sets. The assumptions are that we have many calls to find() and fewer calls to add() and much fewer calls to remove().
A skip list is a sorted multi-level list. Lists on lower level always contain the lists on higher level. We have less entries in higher level lists. Therefore we skip lots of entries on the higher levels while searching for elements.
There is a node height probabilistic, p is often 1/2 or 1/4.
Sublist relationship between levels: higher level lists are always contained in lower-level lists. Lowest level is entire list.
Searching: logrithmic search (with high probability)
Add: find precedessors (lock-free), lock them, validate (cf. lazy synchronization)
Remove: find preds, lock victim, logically remove victim
Contains: sequential find not logically removed fully linked, even if other nodes are removed, it stays reachable. Contains is wait-free (while add and remove are not)

14 Lock-Free

Locks with waiting / scheduling
Locks that suspend the execution of threads while they wait. Semaphores, mutexes and monitors are typically implemented using a scheduled lock.
They require support from the runtime system (OS, scheduler)
Data structures for scheduled locks need to be protected against concurrent access, again using spinlocks, if not implemented lock-free.
Such locks have a higher wakeup latency (need to involve some scheduler)

Locks performance:
Uncontended case: when threads do not compete for the lock; lock implementations try to have minimal overhead
Contended case: threads do compete; can lead to significant performance degradation, starvation

Performance issues:
Overhead for each lock taken even in uncontended case
Amdahl's law!

Blocking semantics:
If a thread is delayed when in a critical section, all threads suffer
A thread can die in a critical section
Prone to deadlocks

Lock-Free Programming
Recap: Defintions
Deadlock: group of two or more competing processes are mutually blocked because each process waits for another blocked process in the group to proceed
Livelock: competing processes are able to detect a potential deadlock but make no observable progress while trying to resolve it
Starvation: repeated but unsuccessful attempt of a recently unblocked process to continue its execution

Definitions for Lock-free synchronization:
Lock-freedom: at least one thread always makes progress even if other threads run concurrently. Implies system-wide progress but not freedom from starvation
Wait-freedom: all threads eventually make progress (in bounded time). Implies freedom from starvation.
Wait-freedom implies lock-freedom !

Progress conditions with and without locks
Everyone makes progress non-blocking: wait-free
Everyone makes progress blocking: starvation-free
Someone makes progress non-blocking: lock-free
Someone makes progress blocking: deadlock-free

Non-blocking: failure or suspension of one thread cannot cause failure or suspension of another thread!

Compare-and-set (CAS)
Example: Non-blocking counter:
read current value v, modify value v', try to set with CAS, return if success, otherwise restart.

Handle CAS with care:
A positive result of CAS suggests that no other thread has written. It is not always true (ABA problem). However, CAS is the best mechanism to check for exclusive access in lock-free programming (maybe transactional memory will become competitive at some point).

More examples for lock-free: lock-free stack (implemented with AtomicReference and CAS)

Performance of lock-free:
A lock-free algorithm does not automatically provide better performance than its blocking equivalent! Atomic operations are expensive and contention can still be a problem (better performance with backoff).

Example: Lock-free list set
The difficulty that arises in this and many other problems is:
We cannot (or don't want to) use synchronization via locks. We still want to atomically establish consistency of two things. Here: mark bit next-pointer
The Java solution: AtomicMarkableReference

AtomicMarkableReference:
Each address of an AMR object has an address and a mark bit. We can atomically swing the reference and update the flag (mark bit). To remove an element from the lock-free list, we set the mark bit in the next field (the element to be deleted) and redirect predecessors's pointer.
Algorithm idea: 1. try to set mark(next), 2. try DCAS (double CAS) until successful

Lock-free Queue:
If we want to implement a lock-free queue (such as we know it), we run into problems because of potentially simultaneous updates of head, tail, or tail.next. We solve this problem with using AtomicReferences for head and tail as well as for the next pointer.

15 Memory Reuse and ABA Problem

Memory Reuse:
Example: Stack. Assume we do not want to allocate memory for each push and maintain a node pool instead. We can implement a NodePool as a second stack to which we push and put nodes that we (currently) do not need anymore. This reduces runtime quite a lot (example: factor 2).
Sometimes the factor is worse. Problem: ABA.

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.
Example: Thread X is in the middle of popping an element A (after read but before CAS). Meanwhile a Thread Y pops A, stores A in the NodePool, A Thread Z pushes B and another Thread Z' pushes A again, which comes from the NodePool. Therefore, Thread X completes the pop (CAS successful) and does not recognize the new state.

How to solve the ABA problem:

DCAS: We can check whether bothheadandhead.nextare as expected, but doesn't exist on most platforms
Garbage Collection: Would eliminate the need for aNodePool, but is very slow and doesn't always exist either

Pointer Tagging: Aligned adresses make some bits availabe for pointer tagging. By incrementing the address bits, 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

Transactional Memory: (next chapter)

16 Linearizability and Sequential Consistency

The aim of this section is to properly define notions of correctness.

Method call: is the interval that starts with an invocation and ends with a response. A method call is called pending between invocation and response. A method call precedes another method call if the response event precedes the invocation event. If therer is no precedence, then the calls overlap.
Notation: $m_0 \rightarrow_H m_1$: history H and method m0 precedes m1.
History: is a finite sequence of method invocations and responses. A subhistory is a subsequence of a history.

Sequential Consistency:
History H is sequentially consistent, if it can be extended to a history G such that G is equivalent to a legal sequential history S.
To be sequentially consistent, a program must fulfill two requirements:
1. method calls should appear to happen in a one-at-a-time sequential order. This means that given a history H, every thread subhistory is a sequential history, i.e., every method call returns before the next one starts.
2. method calls should appear to take effect in program order.

Theorem:
Sequential consistency is not a local property. (and thus we loose composability)

Linearizability:
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.
A history H is linearizable if it has an extension H' such that H' is complete and there is a legal sequential history S such that: 1. complete(H') is equivalent to S, and 2. if $m_0 \rightarrow_H m_1$ then $m_0 \rightarrow_S m_1$.
Composability Theorem:
H is linearizable if, and only if, for each object x, $H|x$, i.e., the subhistory of H with only all method calls concerning object x, is linearizable.

Recall Atomic Registers: they are linearizable with a single linearization point. They are sequentially consistent, every read operation yields most recently written value. For non-overlapping operations, the realtime order is respected.

Linearizability vs. Sequential Consistency:
Linearizability: Operation takes effect instantaneously between invocation and response, uses sequential specification, locality implies composability, good for high level objects
Sequential Consistency: Not composable, harder to work with in software development, good way to think about hardware models

17 Consensus

We want to define a notion of how strong a certain synchronization primitive is.
Idea: 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.
A consensus object provides a single method decide(). Each thread calls the decide method with its input v at most once. The objects decide method will then return a value being consistent (all threads decide the same value) and valid (the common decision value is some thread's input).
In other words, a concurrent consensus object is linearizable to a sequential consensus object in which the thread whose value was chosen completes its call to decide first.
The consensus number is the largest n such that the consensus solves a n-thread-consensus.

Requirements on a consensus protocol:
1. wait-free, 2. consistent, 3. valid Linearizability of consensus must be such that first thread's decision is adopted for all threads.

Theorem: Atomic registers have consensus number 1.
Corollary: There is no wait-free implementation of n-thread-consensus ($n > 1$) from read-write registers.
Theorem: CAS has infinite consensus number.
Theorem: There is no wait-free implementation of a FIFO queue with atomic registers.

18 Transactional Memory

Motivation: programming with locks is difficult, programming without locks even more.

Problems with locks:
Deadlocks: threads might attempt to acquire locks in a different order, thereby introducing a cyclic dependency.
Convoying: A thread might be descheduled while other threads queue up waiting for it to continue.
Priority Inversion: Lower priority threads might hold a resource that a higher priority thread is waiting for
Convention: Association of data and locks, i.e., a correct use of the program by future developers depends on reasonable documentation.
Non-composable: Changing anything about the locking scheme requires changing the whole program. You cannot combine a lock for two threads to have a n thread-safe operation.
Pessimistic by design: assumes the worst and introduces expensive mutual exclusion as a consequence

Goal: remove the burden of synchronization from the programmer and place it in the system (HW/SW)
In transactional memory we define code sections, called transactions, that guarantee:
1. Atomicity: changes made by transactions are made visible atomically, i.e., no other thread observes a state in between. This is solved without mutex.
2. Isolation: while running a transaction, changes made by other threads aren't observed.

Transactions are executed speculatively: as a transaction executes, it makes tentative changes to object. If it completes without a synchronization conflict, it commits or it aborts.
Transactional memory can be implemented in hardware (faster but unflexible) or in software (in programming languages, flexible).

Concurrency Control:
To guarantee that a running transaction always sees consistent data, TM implements a concurrency control (CC) mechanism. The CC will abort a transaction when a conflict occurs. An example of a conflict is when a transaction that hasn't yet committed has read a value that is later changed by a committed transaction. Once a transaction is aborted, it can either be retried automatically or the user is notified.

Zombie transaction: transactions that run even after it is possible for them to commit (conflict already occured).

TM organizes mutable state into atomic objects, to which it associates timestamps, which indicates when the object was last changed by a committed transaction.

atomic block: the difference to normal locks is the execution which is behind it.
atomic {
a.withdraw(amount);
b.deposit(amount);
}

A transaction will keep a local read-set and a local write-set, holding all locally read and written objects, respectively. When a transaction calls read, it will check if the object is in the write-set and use this new version if it is. If not it will check if the object's time stamp is smaller than the transaction's birthdate. If the object has been changed by some other transaction after the transaction's birthdate, it will abort. Otherwise it'll add a new copy of the object to the read-set. When a transaction calls write, it'll create a copy of the object in the write-set, if there isn't one already.
When a transaction attempts to commit, all objects of the read-set and write-set are locked, and the timestamps of the objects in the read-set are compared to the birthdate of the transaction. If any of the objects were changed after the transaction started, it's aborted. Otherwise all objects in the write-set are copied back to global memory with their new timestamps. all locks are released and a commit is returned.

TM-benefits:
simple, composable, optimistic by design (no mutex), higher-level semantics

Nested transactions:
Flat nesting: inner aborts -¿ outer aborts, inner commits -¿ changes visible only if outer commits
closed nesting: similar to flattened nesting but an abort of an inner transaction does not result in an abort for the outer transaction
other approaches(open nesting)

Reference-based STMs: mutable state is put into special variables which can only be modifies inside a transaction (everything else is immutable (or not shared). This is the model we discuss.

Scala-STM:
Scala-STM is a Java API through which we can access the methods provided by the Scala STM library. ScalaSTM is a reference-based STM (mutable state can only be modifies inside a transaction and is put into special variables).
private final Ref.View<Integer> count = STM.newRef(0);
Arrays can be declared as:
private TArray.View<E> items = STM.newTArray(capacity);
Everything else is immutable, which means any other variable accesses inside an atomic block must be declared final.
We can declare 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.

19 Message passing

Main problem of concurrent processing: sharing state.
Idea of message passing: have processes run in isolation and only communicate via messages which must be explicitly received.
Alternative: functional programming (immutable state -¿ no sync needed)

Message Passing Interface:
The Actor model is a model for concurrent computation. It uses actors as computational agents which react to received messages. Received messages are mapped to a set of messages sent to other actors, a new behavior and a new set of actors created. In other words, an actor is a thread which reacts to received messages and has a local state.
Note that the actor model messages are sent in an asynchronous 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.
The java library for it is called the Message Passing Interface (MPI).
MPI collects processes into groups, where each group can have multiple "colors" and 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 number to identify it by, called a rank.
Distributor: type of actor which forwards received messages to a set of actors in a round-robin fashion.

Point-to-Point Communication:
The methods to send and receive messages are declared as follows:
public void Send(Object buf, int offset, int count, Datatype datatype, int dest, int tag);
public void Recv(Object buf, int offset, int count, Datatype datatype, int source, int tag);
Note that in the Recv method it's not necessary to declare src or tag, instead one could use MPI_ANY_SOURCE or *TAG. both methods are declared in the COMM class, i.e. can only be used in combination with a communicator.
The two methods are blocking operations, therefore do not return until the action has been completed locally. Their non-blocking versions recv and send return immediately. We can also send synchronous messages, i.e. the operation blocks until the message has been received using Ssend.
Note that the Recv method declared above is already synchronous.

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)

Group Communication:
MPI does not only support communications between two processes but also among groups.
MPI.COMM.Reduce(int sendoffset, Object recvbuf, int recvoffset, int count, Datatype datatype, Op op, int root); takes an array of input elements on each process and returns an array of output elements to the specified root process. MPI.COMM.ReduceAll() does the same thing, only returning the result to all processes instead of a single root process.
MPI.COMM.Scan() takes an array of input elements on each process and returns distinct arrays of output elements to each process, where the operation is applied iteratively, with the final result array being returned to the specified root process.
Using MPI.COMM.Bcast() one process can broadcast, i.e. send the same data to all processes in a communicator. Note that both the receiver and sender processes call the same method.
MPI.COMM.Scatter() involves a root process sending data to all processes in a communicator. The main difference to Bcast() is that Scatter() sends chunks of an array to different processes while Bcast() sends the entire array to all.

MPI.COMM.Gather() does the exact inverse of Scatter(), in that it takes data from many processes and gathers them to a single root process.