Parallel Programming Summary

Gregory Rozanski

July 17, 2020

0.1 Mutual Exclusion

0.1.1 Definitions

Concurrence:
A form of computing in which several computations are executed during overlapping time periods i.e "concurrently" instead of "sequentially". A concurrent system is one where a computation can advance without waiting for all other computations to complete.

Concurrency Control:
Concurrency control ensures that correct results for concurrent operations are generated while getting those results as quickly as possible.

Race condition:
A race condition or race hazard is the condition of an electronics,software, or other system where the system's substantive behavior is dependent on the sequence or timing of other uncontrollable events.

Scheduling:
Scheduling is the method by which work is assigned to resources that complete the work. Schedulers are often implemented so they keep all computer resources busy, allow multiple users to share system resources effectively, etc.

Thread of execution:
A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler.

Critical section:
Concurrent accesses to shared resources can lead to unexpected or erroneous behaviour so parts of the program where the shared resource is accessed need to be protected in ways that avoid the concurrent access. This protected section is the critical section/region. It cannot be executed by more than one process at a time.

Deadlock:
Problem Description:
The problem which mutual exclusion addresses is a problem of resource sharing: how can a software system control multiple processes access to a shared resource, when each process needs exclusive control of that resource while doing its work?
Solution:
The mutual-exclusion solution to this makes the shared resource available only while the process is in the critical section. It controls access to the shared resource by controlling each mutual execution of that part of its program where the resource would be used.

Language features vs. parallelism: Guidelines

- Keep variables as local as possible: global variables can be accessed by various parallel activities
- If possible, avoid aliasing of references: aliasing can lead to unexpected updates to memory through a process that accesses a seemingly unrelated variable (named differently)
- If possible avoid mutable state, in particular when aliased: aliasing is no problem if the shared object is immutable

Multitasking Concurrent execution of multiple tasks/processes. If you only have one CPU its called multiplexing. If we switch fast enough between the processes multiplexing gives us the feeling that the processes are running in parallel. This is advantegous because the CPU usually waits for inputs and outputs of the memory, hence we can run other processes during this waiting period to increase efficiency.

Process context A process is a program executing inside an Operating System. Each running instance of a program is a seperate process. Each process has a context:

- Instruction counter: points to next instruction
- Values in registers, stack and heap
- Resource handles
- ...

When switching between context we have to temporarily save the context of the current process and load the context of the next process

process lifecycle

1. Process created (load from disk)
2. Process in waiting state (Pool of processes which can be executed)
3. Scheduler picks the process and puts it in a Running state
4. Process can enter Blocked state (usually because of I/O, hence it cannot be executed). When block is released it returns to a waiting state
5. Process enters Terminated state, where context is deleted

Each process demands a certain amount of the main memory. In the case where there isnt enough left, swapping takes place in which the context of the current process is put on the Hard Disk creating space for another process. (Slows down the system).

Context Switch When switching between two processes, the OS interrupts the first one captures its state, loads the state of the second process and executes it. There is alot of overhead generated when switching between processes, hence switching alot between processes is inefficient.

Threads Threads are:

- independent sequences of execution
- running in the same OS process

Multiple threads share the same address space hence they execute different code but share the same memory. Threads have the advantage that they are not controlled by protocols and can read and write freely (this also makes it more vulnerable to programming mistakes) .

- Threads are not shielded from each other
- Threads share resources and can communicate more easily

Context switching between threads is efficient

- no change of address space
- no automatic scheduling
- no saving/reloading of PCB (OS process) state

Multithreading 1 vs. many CPU's When multiple threads share a single CPU then the threads take turns executing and the others are put in a waiting state. With multiple CPU's e.g 3 threads, 3 CPUs all threads can run constantly increasing performance.

Java Threads JVM implementation of the thread concepts i.e parallel execution. It is a set of instructions to be executed one at a time, in a specific order. Thread class is part of the core language. Every Java program has at least one execution thread (first one calls main()). A Program ends when all threads finish. Threads can continue to run even if main() returns. Creating a Thread object or calling run() does not start a thread we need to call start().

java.lang.Thread

- start() : method called to spawn a new thread (causes JVM to call run() method on object)
- interrupt() : freeze and throw exception to thread (used to terminate a thread at time of call)
- sleep(int num) : puts thread to sleep for num ms
- getID(): gets the thread's ID
- getName() : gets the name of the currentThread
- setName() : sets the name of the currentThread
- currentThread() : returns current Thread
- setPriority(int num) : 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. The Scheduler decides wether or not to regard the priority of the threads.
- getState() : Denotes the status the thread is in
- join(): thread finishes and returns the result to the sleeping main thread (May throw InterruptedException)
- wait() : Consumer goes to sleep i.e status NOT RUNNABLE. If the thread has the lock and is in a state where it cant do anything productive, wait is called such that other threads can access the resource (can only be used if the thread holds the lock). It is recommended to use a while loop around the condition, in order to see that the thread returned from the wait() at a valid time. When not specifying myObject.wait() then wait() = this.wait().
- notify()/notifyAll() : Changes the state of all threads waiting on the resource to Runnable (can only be used if the thread holds the lock i.e in synchronized block). notify() wakes the highest-priority thread closest to front of object's internal queue. When not specifying myObject.notify() then notify() = this.notify()

0.2 Creating Java Threads

OPTION 1: Instantiate a subclass of java.lang.Thread class

- Override run method
- 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!

```
class ConcurrWriter extends Thread {
      public void run() {
            // code here executes concurrently with caller;
      }
}
ConcurrWriter writerThread = new ConcurrWriter();
writerThread.start(); // calls ConcurrWriter.run()
```

OPTION 2: Use Runnable Interface

- single method: public void run()
- class implements Runnable

```
public class ConcurrWriter implements Runnable {
      public void run() {
            // code here executes concurrently with caller;
      }
}
ConcurrReader readerThread = new ConcurrReader();
Thread t = new Thread(readerThread);
t.start(); // calls ConcurrWriter.run()
```

Here there it is distinguished between how the programm is executed (the thread) and what is being executed (Runnable)

Busy Waiting: By spinning(looping) until each worker's state is TERMINATED. Join (sleep, wakeup) typically incurs context switch overhead. If worker threads are short-lived, busy waiting may perform better.

Exceptions: Exceptions in a single threaded (sequential) program terminate the program, if not caught. If a worker thread throws an exception, the exception is shown on the console, the behaviour of thread.join() is unaffected, hence the main thread may not be aware of an exception inside a worker thread. Implementing UncaughtExceptionHandler interface allows us to handle unchecked exceptions. Three options:

- Register exception handler with Thread object
- Register exception handler with ThreadGroup object
- Use setDefaultUncaughtExceptionHandler() to register handler for all threads Handler can then record which threads terminated exceptionally or restart them, or ...

```
public class ExceptionHandler implements UncaughtExceptionHandler {
      public public Set<Thread> threads = new HashSet<>();{


@Override
public void uncaughtException(Thread thread, Throwable throwable){
      println("An exception has been captured");
      println(thread.getName());
      println(throwable.getMessage());
      ...
      threads.add(thread);
      }
}


public class Main {
      public static void main(String[] args) {
      ...
      ExceptionHandler handler = new ExceptionHandler();
      thread.setUncaughtExceptionHandler(handler);
      ...
      thread.join();
      if (handler.threads.contains(thread)){
        //bad
      } else {
        // good
      }
  }
}
```

Thread Safety: This implies program safety and refers to "nothing bad ever happens", in any possible interleaving.

Liveness: "eventually something good happens" (e.g endless loops are an example of liveness hazards in sequential programming). Threads makes liveness hazards more frequent: If ThreadA holds a resource(e.g a file handle) exclusively, then ThreadB might be waiting for that resource forever. Hence liveness means that progress will be made.

Examples of safety properties:

- absence of data races
- mutual exclusion
- linearizability
- atomicity
- schedule-deterministic
- absence of deadlock
- custom invariants

Synchronized Multiple threads may read/write the same data (shared objects,global data). To avoid bad interleaving we use explicit synchronization. In Java, all objects have an internal lock, called intrinsic/monitor lock. Synchronized operations lock the object, hence no other thread can successfully lock and use the object and must wait until the lock is freed. Generally if accessing shared memory, make sure itis done under a lock, if not the code is prone to a data race.

Synchronized Methods: A synchronized method grabs the object or class's lock at the start , runs to completion, then releases the lock. This is useful for methods whose entire bodies are critical sections, and thus should not be entered by multiple threads at the same time. A synchronized method is a critical section with guaranteed mutual exclusion

```
// synchronized method: locks on "this" object
public synchronized type name(parameters) {...}

// synchronized static method: locks on the given class
public static synchronized type name(parameters) {...}


Synchronized Blocks:

synchronized (object) {
    statement(s); //critical sections
}
```

Synchronized Blocks: Enforces mutual exclusion with regards to some object. Every Java object can act as a lock for concurrency: A thread T_1 can ask to run a block of code, synchronized on a given object O. The synchronized block makes sure there is no interleavings of the statements inside the block, it does not prevent other threads from executing statements outside of the block, hence it is still possible for bad interleavings to happen with statements outside the block

- If no other thread has locked O, then T_1 locks the object and proceeds
- If another thread T_2 has already locked O, then T_1 becomes blocked and must wait until T_2 is finished with O (that is, unlocks O). Then, T_1 is woken up, and can proceed

Reentrant: Locks are recursive. A thread can request to lock an object it has already locked, and will lock it, the thread will then release the lock multiple times.

Synchronization granularity: Using multiple locks to allow multiple threads to work on code while still being protected.

Synchronized and Exception If an exception is triggered in the middle of a synchronized bock, then the lock released, as if the synchronized scope ends right at the point where the exception is thrown. When the exception is caught, then the exception handler is executed. If there is no exception handler, then the exception is propagated back down to the caller of the method. Any side effects are not reverted, they do take effect even if exceptions are thrown.

Producer-Consumer: The Producer puts items into a shared buffer (shared resource), the consumer takes them out, consumption is only possible if buffer isn't empty.

Pseudo-Code Implementation of synchronized block :

0.3 Parallel Architectures

Parallelism: Use extra resources to solve a problem faster

Concurrency: Correctly and efficiently manage access to shared resources.

Distributed computing: Physical separation, administrative separation, different domains, multiple systems

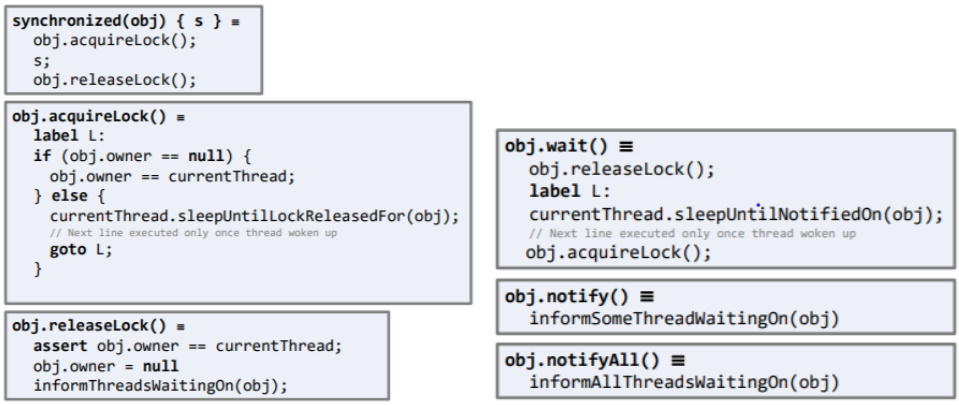


Figure 1: Pseudo implementation of synchronized block

Von Neumann architecture: Program data and program instructions share the same memory.

CPUs and Memory Hierarchies Caches are preloaded data readily available to speed up access time.

- Goal: Allow cores to work int parallel, on their own, fast memory
- CPU reads/writes values from/to main memory, to compute with them, with a hierarchy of memory caches in between. Faster memory is more expensive, hence smaller: L1 is 5x faster than L2, which is 30x faster than main memory, which is 350x faster than disk.
- Synchronisation between caches is taken care of by cache coherence protocols(e.g MESI see notes page 3)
- Concurrency Hazard: cores may pre-/postpone reads/writs from/to cache; memory barriers needed to prevent problems with parallel code. (In Java memory barriers are automatically insterted if e.g synchronized is used.)

Vectorization:

- Goal: improve performance by using specialized vector instructions
- SIMD: Single Instruction, applied to Multiple Data
- Requires vectorised code: code that uses the vector instructions provided by the target platform (CPU)
- Compilers(C++, JVMs JIT,...) attempt to detect vectorization opportunities → fully automated, but little or no control over if/where/how
- platform specific libraries (intrinsics,C/C++) expose vector instructions to developers → manual effort, but full control
- Poses no (additional) safety risk to concurrency

Instruction Stream : Instructions given to the CPU to execute

Instruction Level Parallelism (ILP)

- Goal: improve CPU performance by internal parallelisation
- CPU/Core detects independent operations in its instruction stream
- These may be executed in parallel inside the CPU if enough functional units (e.g floating-point unit,...) are available
- Various measures to increase potential for instruction parallelization. E.g speculatively execute instructions in parallel, even if result may not be used
- Concurrency hazard: cores only locally consider dependencies in their instruction stream, not globally across all cores. (Java e.g synchronized automatically adds memory barriers to prevent problematic reordering)
- Compilers may also reorder instructions; similar problems, same solution

0.3.1 Pipelining

Balanced Pipeline All steps require the same time

Throughput The amount of work that can be done by a system in a given period of time (How much can go through the pipeline in a given time)

- In CPUs : # of instructions completed per second
- The larger the throughput the better

Throughput bound $= \frac{1}{max(computationtime(stages))}$
1:= unit of work (e.g one instruction, one network package, ...)
max(computationtime(stages)) := the time of the longest step in the pipeline
The bound gives the throughput when the pipeline is at full utilization i.e it ignores lead-in and lead-out time

Latency Time needed to perform a given computation (I.e how long does it take one item to go through the pipeline)

- In CPU: time required to execute a single instruction in the pipeline
- Lower is better
- Pipeline latency is only constant over time if the pipeline is balanced (i.e each step takes the same time)
- more input means our bound is less exact

Latency bound $= \#stages \cdot max(computationtime(stages))$

Optimizing an unbalanced pipeline (E.g Clothes Washing) w: 5s d:10s f: 5: c:10, the given pipeline is unbalanced because drying and putting clothes in the closet takes more than the washing and folding. An attempt to balance the pipeline to get a constant latency would be to artificially increase the length of all steps to 10s, but in this case we would decrease the throughput. The other option is to add additional functional units i.e another dryer and closet increasing the total number of steps from 4 to 6 w:5s d1:4s d2:6 f:5 c1:4 c2:6, we then increase the duration of all steps to the duration of the longest step i.e 6, hence the pipeline is balanced and the throuput increased.

Throughput vs Latency Pipelining typically adds constant time overhead between individual stages (synchronization,communication), hence infinitely small pipeline steps are not practical and time it takes to get one complete task through the pipeline may take longer than with a serial implementation.

0.4 Basic Concepts in Parallelism

Expressing Parallelism The goal is to split up work of a single program into parallel tasks. This can be done Explicitly/Manually(task/thread parallelism) or Implicitly i.e Done automatically by the system (user expresses an operation and the system does the rest.

Work Partitioning & **Scheduling**

- work partitioning (task/thread decomposition)
  - split up work into parallel tasks/threads
  - done by user
  - A task is a unit of work
  - number of parititions should be larger than the number of processors
- scheduling
  - assign tasks to processors
  - typically done by the system
  - goal is full utilization i.e no processor is ever idle

Coarse vs Fine granularity

- Fine granularity
  - more portable (can be executed in machines with more processors
  - better for scheduling
  - but: if scheduling overhead is comparable to a single task → overhead dominates
- Task granularity guidelines
  - As small as possible but, significantly bigger than scheduling overhead

Scalability An overloaded concept: e.g how well a system reacts to increased load, for example clients in a server. In parallel programming:

- speedup when we increase processors
- what happens if $\#processors \rightarrow \infty$
- program scales linearly → linear speedup

Parallel Performance Sequential execution time: T_1
Execution time T_p on p CPUs

- $T_p = T_1/p$ (Perfect Case)
- $T_p > T_1/p$ (Performance loss,what normally happens)
- $T_p < T_1/p$ (Can happen but unusual)

Parallel Speedup Speedup S_p on p CPUs $S_p = T_1/T_p$:

- $S_p = p$ linear speedup (Perfect Case)
- $S_p < p$ sub-linear speedup (Performance loss,what normally happens)
- $S_p > p$ super-linear speedup (Can happen but unusual)

Speedup is not only dependant on the program but also on the input.
Why $S_p < p$?
Programs may not contain enough parallelism (some parts might be sequential)
Overheads introduced by parallelization (typically associated with synchronization)
Architectural limitations (e.g memory contention)

Efficiency S_p/p how efficient is a multicore system for a given task

Amdahl's Law Execution time T_1 of a program falls into two categories:

- Time spent doing non-parallelizable serial work
- Time spent doing parallelizable work

Denoted: W_{ser}, W_{par}
Given P workers available to do parallelizable work, the times for sequential execution and parallel execution are:
$T_1 = W_{ser} + W_{par}$
Resulting in a bound on speed up: $T_p \geq W_{ser} + \frac{W_{par}}{P}$
$\Rightarrow$ Amdahls Law: $S_p \leq \frac{W_{ser}+W_{par}}{W_{ser}+\frac{W_{par}}{P}}$
We define f as the non-parallelizable serial fractions of the total work. The following equalities hold:

- $W_{ser} = fT_1$
- $W_{par} = (1-f)T_1$

$\Rightarrow S_p \leq \frac{1}{f+\frac{1-f}{P}} \quad \Rightarrow S_\infty \leq \frac{1}{f}$

Gustafson's Law Observations:

- consider problem size
- run-time, not problem size, is constant
- more processors allows to solve larger problems in the same time
- parallel part of a program scales with the problem size

f : sequential part, $T_{wall} = availabletime$
$W = p(1-f)T_{wall} + fT_{wall}$
$S_p = \frac{S_p}{S_1} = f + p(1-f) = p - f(p-1)$

0.5 Divide and Conquer

fork/join Style of programming using start,run,join methods. They create a "happens before before relation", the ordering of the memory access is important and must be considered.

Approach to Divide and Conquer In theory you can divide down to single elements, do all your resultcombining in parallel and get optimal speedup. In practice, creating all those threads and communicating swamps the savings hence:

- Use a sequential cutoff, typically around 500-1000 (eliminates almost all the recursive thread creation(bottom levels of tree)
- Do not create two recursive threads, create one and do the other "yourself"
- If given enough processors, total time is height of the tree $\mathcal{O}(logn)$
- Often relies on operations being associative

```
public class SumThread extends Thread {
int[] xs;
int h, l;
int result;

public SumThread(int[] xs, int l, int h){
        super();
        this.xs = xs;
        this.h = h;
        this.l =l;
}

public void run(){
        /*Do computation and write to result*/
        return;
}
```

(a) creating task

```
public void run(){
        int size = h-l;
        if (size < SEQ_CUTOFF)
                for (int i=l; i<h; i++)
                        result += xs[i];
        else {
           int mid = size / 2;
           SumThread t1 = new SumThread(xs, l, l + mid);
           SumThread t2 = new SumThread(xs, l + mid, h);
           t1.start();
           t2.start();
           t1.join();
           t2.join();
           result = t1.result + t2.result;
        }
}
```

(b) creating executer,submitting

```
// wasteful: don't
SumThread t1 = …
SumThread t2 = …
t1.start();
t2.start();
t1.join();
t2.join();
result=t1.result+t2.result;
```

```
// better: do
// order of next 4 lines
// essential - why?
t1.start();
t2.run();
t1.join();
result=t1.result+t2.result;
```

(c) creating executer,submitting

Executor Service: Manages asynchronous tasks. ExecutorService is a Java Class which takes in a users submitted task and returns a "Future" object. Two ways to submit a task to the ExecutorService:

- .submit(Callable $< T >$ task) $\rightarrow$ Future$< T >$ (Returns result)
- .submit(Runnable task) $\rightarrow$ Future$<?>$ (Does not return result)

Beispiel:

```
static class HelloTask implements Runnable {

      String msg;

      public HelloTask(String msg) {
            this.msg = msg;
      }

      public void run() {
            long id = Thread.currentThread().getId();
            System.out.println(msg + " from thread:" + id);
      }
}
```

(a) creating task

```
int ntasks = 1000;
ExecutorService exs = Executors.newFixedThreadPool(4);

for (int i=0; i<ntasks; i++) {
  HelloTask t = new HelloTask("Hello from task " + i);
  exs.submit(t);
}

exs.shutdown();
```

(b) creating executer,submitting

Recursive Sum with ExecutorService:

```
public Integer call() throws Exception {
  int size = h - l;
  if (size == 1)
    return xs[l];

  int mid = size / 2;
  sumRecCall c1 = new sumRecCall(ex, xs, l, l + mid);
  sumRecCall c2 = new sumRecCall(ex, xs, l + mid, h);

  Future<Integer> f1 = ex.submit(c1);
  Future<Integer> f2 = ex.submit(c2);

  return f1.get() + f2.get();
}
```

Figure 4

The get method blocks until the method which the Future object refers to is finished. The above implementation does not

work because the ExecutorService is bound to a certain number of threads, hence we will eventually run out of threads and the tasks will end up waiting. The ExecutorService is not meant to be used when you need to wait for results of other tasks (divide and conquer). A possible approach is to decouple work partitioning from solving the problem. We split the array into chunks and create a task per chunk, we submit these into the ExecutorService and combine the results. When one task is finished the thread is freed and assigned to another task. I.e flat patterns (threads arent waiting) are good for the ExecutorService.

Cilk-style: Tasks:

- execute code
- spawn other tasks
- wait for results from other tasks

A graph is formes based on spawning tasks. There is an edge from node u to node v if task v was created by task u. Source vertice must finish first before destination starts. With Cilk there is no waiting for a certain task, but instead we wait for all tasks created until now to complete. There are no deadlocks in Cilk style programming (The task graphs are directed acyclic graphs).

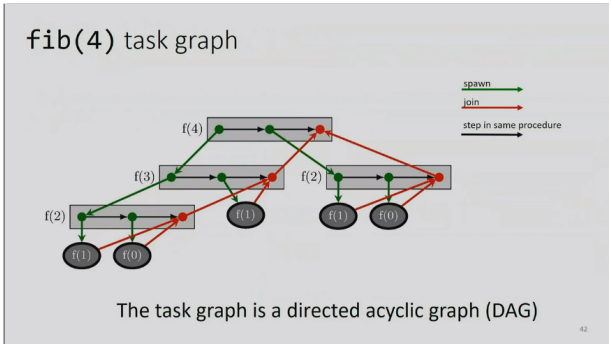


Figure 5

Task Parallelism:

- Tasks can execute in parallel, but they dont have to. The assignment of tasks to CPUs/Cores is up to the scheduler
- The task graph is dynamic and unfolds as execution proceeds(input dependent). A wide task graph means more parallelism

Performance Model: Tasks become available as computation progresses. We can execute the graph on p processors, the scheduler assigns tasks to the processors, hence the execution time T_p can vary depending on the scheduler being used.

- T_p execution time on p processors
- T_1 work(total amount of work) i,e the sum of the time cost of all nodes in graph (as if we executed graph sequentially)
- $\frac{T_1}{T_p} \rightarrow$ speedup
- T_∞ span, critical path,computational depth: Time it takes on infinite processors i.e the longest path from root to sink
- $\frac{T_1}{T_\infty} \rightarrow$ parallelism i.e maximum possible speedup
- Lower bounds:
  - $T_p \geq \frac{T_1}{p}$
  - $T_p \geq T_\infty$
- $T_p \approx \frac{T_1}{p} + T_\infty$

Scheduler is an algorithm for assigning tasks T_p depends on the the scheduler. $\frac{T_1}{P}$ and T_∞ are fixed

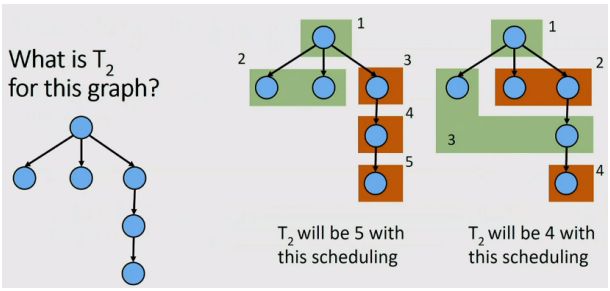


Figure 6

The above figure shows that different schedulers can have different T_p. The boxes represent what is scheduled in each step.

0.6 ForkJoin Framework & Task Parallel Algorithms

ForkJoin Framework: Designed to meet the needs of divide-and-conquer fork-join parallism.

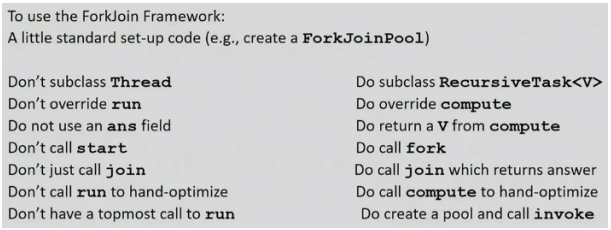


Figure 7

- .fork() → create a new task (Computation Graph: Ends a node and makes two outgoing edges i.e new thread and continuation of current thread)
- .join() → return result when task is done (Computation Graph: Ends a node and makes a node with two incoming edges i.e task just ended last node of thread joined on)
- .invoke() → submits task and waits until it is completed
- .submit() → submits task (recieves a Future)

Recursive sum with ForkJoin: The ForkJoinPool creates a number of threads equal to the number of available processors.

```
class SumForkJoin extends RecursiveTask<Long> {
    int low;
    int high;
    int[] array;

    SumForkJoin(int[] arr, int lo, int hi) {
        array = arr;
        low   = lo;
        high  = hi;
    }

    protected Long compute() { /*…*/ }
```

(a)

```
protected Long compute() {
      if(high - low <= 1)
          return array[high];
      else {
          int mid = low + (high - low) / 2;
          SumForkJoin left  = new SumForkJoin(array, low, mid);
          SumForkJoin right = new SumForkJoin(array, mid, high);
          left.fork();
          right.fork();
          return left.join() + right.join();
      }
  }
```

(b)

Recursive sum with ForkJoin (use)

Default # of processors

```
class Globals {
    static ForkJoinPool fjPool = new ForkJoinPool();
}

static long sumArray(int[] array) {
    return Globals.fjPool.invoke(new SumForkJoin(array,0,array.length));
}
```

(c)

The code aboce performs poorly in java. The following fix is possible:

```
protected Long compute() {
      if(high - low <= SEQUENTIAL_THRESHOLD) {
          long sum = 0;
          for(int i=low; i < high; ++i)
              sum += array[i];
          return sum;
       } else {
          int mid = low + (high - low) / 2;
          SumForkJoin left  = new SumForkJoin(array, low, mid);
          SumForkJoin right = new SumForkJoin(array, mid, high);
          left.fork();
          long rightAns = right.compute();
          long leftAns  = left.join();
          return leftAns + rightAns;
       }
   }
```

Figure 9

Reductions: Produce a single answer from collection via an associative operator (e.g max, count, leftmost,rightmost,...) (non examples: median, subtraction, exponentiation). (Recursive) results dont have to be a single number or strings. They can be arrays or objects with multiple fields. (e.g Histogram of test results is a variant of sum). But some things are inherently sequential i.e how we process arr[i] may depend entirely on the result of processing arr[i-1].

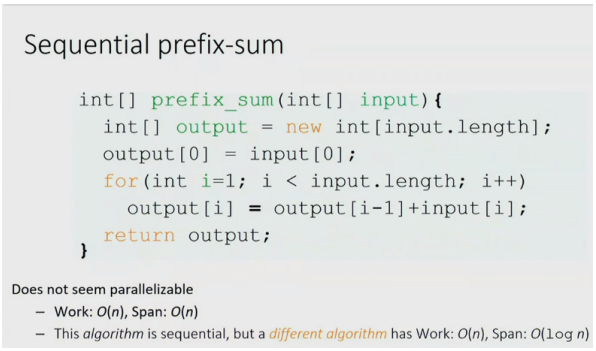
Maps: A map operates on each element of a collection independently to create a new collection of the same size, hence there is no combining results.

When to use Maps or Reduction:

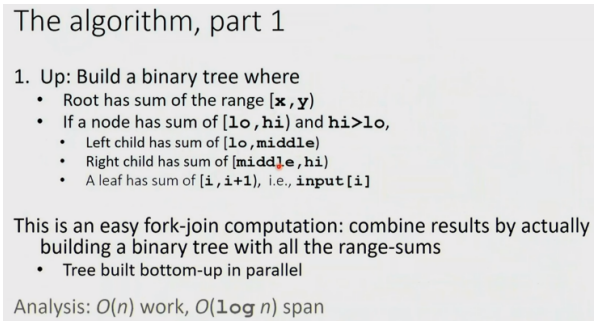
- Data structure matters!
- Parallelism is still beneficial for expensive per-element operations on a sequential Datastructure (e.g Linked Lists)
- For parallelism, balanced trees are generally better than lists so that we can get to all the data exponentially faster $\mathcal{O}(logn)$ vs $\mathcal{O}(n)$

The prefix-sum problem: Example used to show that inherently sequential programs can in fact be made parallel. Problem Statement:

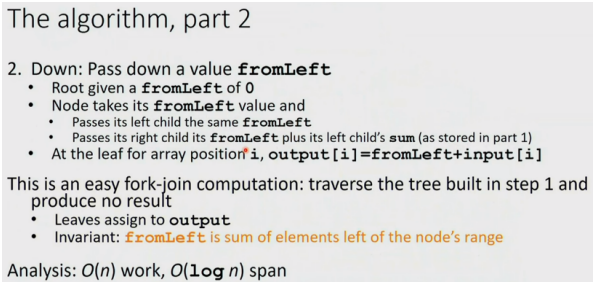
- Given int[] input
- Produce int[] output
- output[i] = input[0] + input[1] + ... + input[i]



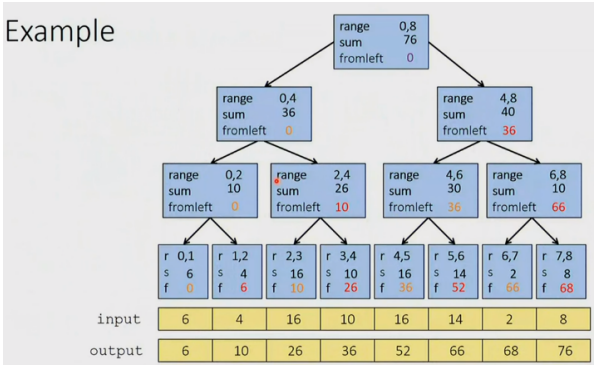
(a)



(b)



(a)



(b)

We get a parallel speedup at the expense of using more memory.

Pack Problem: Given an array input, produce an array output containing only elements such that f(elt) is true (i.e elements such that some property holds e.g elt ¿ 10). How is this Parallelizable? The work is $\mathcal{O}(n)$. Difficulty arises when trying to find the position of the current element in the result as its position depends on how many elements before it satisfy the condition. Solution (Using condition elt ¿ 10):

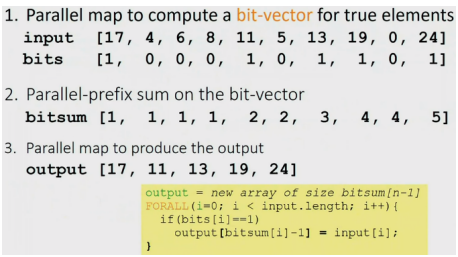


Figure 12

0.7 Shared memory concurrency, locks and data races

Managing State

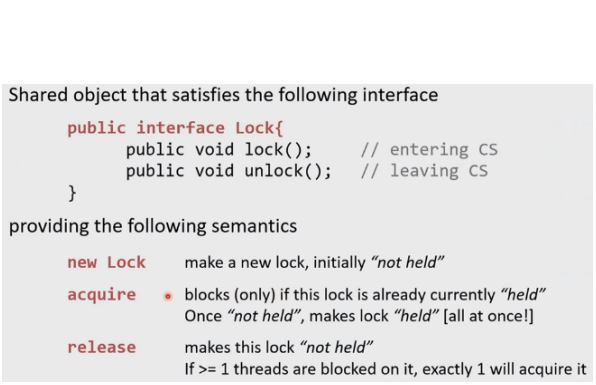
- Immutability Data does not change. This is the best option and should be used when possible
- Isolated Mutability Data can change, but only one thread/task can access them
- Mutable/Shared data Data can change, multiple Tasks/Threads can potentially access the data

Mutable/Shared data: This is present in shared memory architectures. Concurrent accesses may lead to inconsistencies, hence we must protect the state by allowing only one thread/task access the memory at a time. We can achieve this by using the following methods:

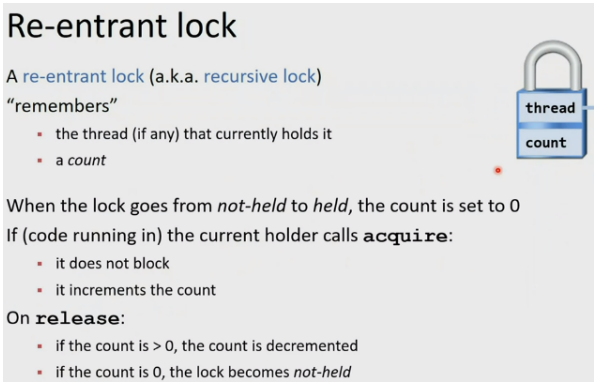
- Locks: Mechanism to ensure exclusive access/atomicity (Assume that there will be other threads that will try to modify the memory)
- Transactional memory: Programmer describes a set of actions that need to be atomic (Perform actions and only after completion do we check if there was a conflict, if a conflict occured we rollback)

Mutual Exclusion: When one thread uses a resource another thread must wait until its free. (The resource is known as a critical section). Implementing critical sections is done by the programmer as the compiler is not capable of recognizing them and bad interleavings can occur.

Lock Object in Java: Locks ensure that given simultaneous acquires and/or releases, a correct thing will happen. Class



(a)



(b)

for Reentrant Locks: java.util.concurrent.locks.ReentrantLock

Races: A race condition occurs when the computation result depends on the scheduling (how threads are interleaved). There is no interleaved scheduling with only one thread but interleaved scheduling with only one processor is possible.

Data Race vs. Bad Interleaving:

- **Data Race:** [aka Low Level Race Condition] 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.
- **Bad Interleaving:**[aka High Level Race Condition] Erroneous program behavior caused by an unfavorable execution order of a multithreaded algorithm that makes use of otherwise well synchronized resources.

3 options to avoid data races: For every memory location in your program, you must obey atleast one of the following:

- Thread-Local: Do not use the location for more than 1 threas
- Immutable: Do not write to the memory location
- Synchronized: Use synchronization to control access to the location

Thread-Local: Whenever possible, do not share resources.

- It is easier to have each thread have its own thread-local copy of a resource than to have one with shared updates
- This is only correct if threads do not need to communicate through the resource
- Because each call-stack is thread-local we do not need to synchronize on local variables

Immutable: Whenever possible do not update objects, instead make new objects. This helps to avoid side-effects and helps in a concurrent setting. If a location is read only then no synchronization is necessary (simultaneous reads are not races and not a problem.

The Rest: After minimizing the amount of memory that is thread-shared and mutable, we need guidelines for how to use locks to keep other data consistent. Guidlines:

1. No data races: Never allow two threads to read/write or write/write to the same location at the same time and do not make any assumptions on the orders of reads or writes.
2. Consistent Locking: For each location needing synchronization, have a lock that is always held when reading or writing the location. The lock "guards" the location and the same lock can guard multiple locations. (It is important to clearly document the guard for each location). Consistent locking is not sufficient, it prevents all data races but still allows bad interleavings.
3. Lock granularity: Start with coarse-grained and move to fine-grained only if contention on the coarser locks becomes an issue.
    - Coarse-grained: Fewer locks i.e more objects per lock (e.g one lock for an array). Coarse grained locking is simpler to implement and faster/easier to implement operations that access multiple locations. Also much easer to implement operations that modify the data-structures shape.
    - Fine-grained: More locks i.e fewer objects per lock (e.g one lock per array index). Fine grained locking allows for a more simultaneous access (performance when coarse-grained would lead to unnecessary blocking)

4. Critical-section granularity: Do not do expensive computations or I/O in critical sections, but also dont introduce race conditions. A second orthogonal granularity issue is critical section size. If the critical sections run for too long then performance will be lost because of other threads being blocked. On the other hand if critical sections are too short then bugs can be created because other threads see intermediate states they shouldnt and performance can be lost because of frequent thread switching and cache trashing.
5. Atomicity: Think in terms of what operations need to be atomic. An operation is atomic if no other thread can see it partly executed ("appears" invisible). Make the critical sections just long enough to preserve atomicity, then design the locking protocol to implement the critical sections correctly i.e Think about atomicity first and locks second.

Memory Reordering: The Compiler and hardware are allowed to make changes that do not affect the semantics of a sequentially executed program. What gets reordered depends on hardware e.g AMD86 is different than ARM.

- Software view: Modern compilers do not give guarantees that a global ordering of memory accesses is provided. Some memory accesses may be optimized away completely.
- Hardware view: Modern multiprocessors do not enforce global ordering of all instructions because of performance gains. Most processors have a pipelined architecture and can execute multiple instructions simultaneously. They can (and will) reorder instructions internally. Each processor has a local cache, and thus loads/stores to shared memory can become visible to other processors at different times.

There are some language constructs that forbid such reordering. (in Java synchronized and volatile)

Memory Models: The exact behaviour of threads interacting via shared memory usually depends on hardware,runtime system, and programming language. A memory model provides guarantees for the effects of memory operations, leaving open optimization possibilities for hardware and compiler, but including guidelines for writing correct multithreaded programs.

Java Memory Model(JMM) :

- JMM restricts allowable outcomes of programs
- JMM defines Actions: read/write e.g read(x):1 "read variable x, the value read is 1"
- Exections combine actions with ordering:
  - Program Order (Order in which statements are executed)
  - Synchronizes-with (Order of observed synchronizing memory actions across threads)
  - Synchronization Order (oder of synchronizing memory actions in the same thread)
  - Happens-before (union(transitive closure) of PO and SW)

Program Order(PO):

- Program order is a total order of intra-thread actions. Program statements are NOT a total order across threads.
- Program order does not provide an ordering guarantee for memory accesses
- Intra-thread consistency: Per thread, the PO order is consistent with the threads isolated execution

Synchronization Actions(SA):

- Read/write of a volatile variable
- Lock monitor, unlock monitor
- First/last action of a thread
- Actions which start a thread
- Actions which determine if a thread has terminated

underlineSynchronization Order(SO): formed by the synchronization actions:

- SO is a total order (all threads see the same order)
- all threads see SA in the same order
- SA within a thread are PO
- SO is consistent, all reads in SO see the last writes in SO

Synchronizes-With (SW)/ Happens-Before (HB) order:

- SW only pairs the specific actions which see eachother
- A volatile write to x synchronizes with subsequent read of x
- 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

0.8 Behind Locks: Implementation of Mutual Exclusion

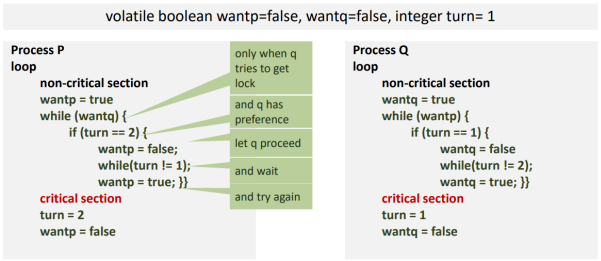
Assumptions:

- Atomic reads and writes of variables of primitive type
- no reordering of read and write sequences (not true in practice!)
- threads entering a critical section will leave it eventually
- otherwise we assume a multithreaded environment where processes can arbitrarily interleave
- we make no assumptions for progress in non critical secion

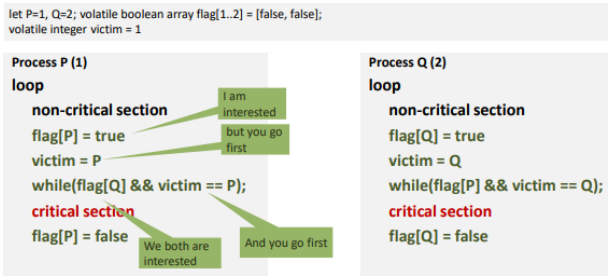
Critical Sections: Pieces of code with the following conditions:

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

Decker's Algorithm:



Peterson Lock: When implementing Peterson in Java setting an array to volatile doesnt work. Volatile will be the



reference to the array and not an array of volatile variables, instead we use Java's AtomicInteger and AtomicIntegerArray.

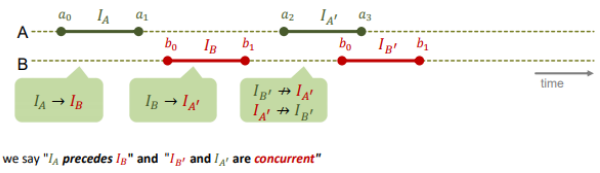
Events and precedence: Threads produce a sequence of events

$$\text{P produces events } p_0, p_1$$
$$\text{e.g } p_1 = \text{"flag[P] = true"}$$

j-th occurence of event i in thread P: p_i^j (e.g p_5^3 = "flag[P] = false" in the third iteration.)
Precedence relation: we write $a \to b$ when a occurs before b (the precedence relation is a total order for events)

Intervals:

$$(a_0, a_1) : \text{interval of events } a_0, a_1 \text{ with } a_0 \to a_1$$
$$\text{With } I_A = (a_0, a_1) \text{ and } I_B = (b_0, b_1) \text{ we write } I_A \to I_B \text{ if } a_1 \to b_0$$



we say "I_A *precedes* I_B" and "$I_{B'}$ and $I_{A'}$ are *concurrent*"

Atomic register: A Register is a basic memory object which can be shared or not (i.e in this context register $\neq$ register of a CPU). A Register r has two operations: r.read() and r.write(v). Atomic Register has the following structure:

- An invocation J of r.read or r.write takes effect at a single point $\tau(J)$ in time (i.e no two reads or writes will happen simultaneously)
- $\tau(J)$ always lies between start and end of the operation J
- Two operations J and K on the same register always have a different effect time $\tau(J) \neq \tau(K)$

These assumptions for Atomic Registers justify to treat operations on them as events taking place at a single point in tiime. Even with atomic registers there can still be non determinism of programs because nothing is said about the order of effect times for concurrent operations.

Filter Lock: Extension of Peterson's lock to n processes. Every thread t knows his level in the filter level[t]. In order to enter 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 one victim victim[l] that has to let others pass in case of conflicts.

```
import java.util.concurrent.atomic.AtomicIntegerArray;
class FilterLock{
   AtomicIntegerArray level;
   AtomicIntegerArray victim;
   volatile int n;

   FilterLock(int n) {
      this.n = n;
      level = new AtomicIntegerArray(n);
      victim = new AtomicIntegerArray(n);
   }
   ...
```

```
...
   // ∃k ≠ me: level[k] >= i (lev)
   boolean Others(int me, int lev) {
      for (int k = 0; k < n; ++k)
         if (k != me && level.get(k) >= lev) return true;
      return false;
   }
   public void Acquire(int me) {
      for (int lev = 1; lev < n; ++lev) {
         level.set(me, lev);
         victim.set(lev, me);
         while(me == victim.get(lev) && Others(me,lev));
      }
   }
   public void Release(int me) {
      level.set(me, 0);
   }
}
```

Again: I (as a thread) can make progress if
(a) Another thread wants to enter my level or
(b) No more threads are in front of me
This works because there are at most n threads in the system.

Fairness: Divide lock implementation into two parts:

- Doorway interval D: finite number of steps
- Waiting interval W: unbounded number of steps

A lock algorithm is first-come-first-served when for two processes A and B holds that if $D_A^j \rightarrow D_B^k$ then $CS_A^J \rightarrow CS_B^k$

0.9 Spinlocks,Deadlocks Semaphores

Bakery Algorithm: Each thread is given a label and a flag indicating it wether or not it wants to access the critical section. When a thread wants to acquire the critical section it gets assigned the lowest label number not already given to any other thread. If there is no thread with a lower label wanting to access the CS then the thread can acquire it. The

```
class BakeryLock
{
   AtomicIntegerArray flag; // there is no
AtomicBooleanArray
   AtomicIntegerArray label;
   final int n;

   BakeryLock(int n) {
      this.n = n;
      flag = new AtomicIntegerArray(n);
      label = new AtomicIntegerArray(n);
   }

   int MaxLabel() {
      int max = label.get(0);
      for (int i = 1; i<n; ++i)
         max = Math.max(max, label.get(i));
      return max;
   }
   ...
```

```
   boolean Conflict(int me) {
      for (int i = 0; i < n; ++i)
         if (i != me && flag.get(i) != 0) {
            int diff = label.get(i) - label.get(me);
            if (diff < 0 || diff == 0 && i < me)
               return true;
         }
      return false;
   }

   public void Acquire(int me) {
      flag.set(me, 1);
      label.set(me, MaxLabel() + 1);
      while(Conflict(me));
   }

   public void Release(int me) {
      flag.set(me, 0);
   }
}
```

Time and space complexity is $\mathcal{O}(n)$. Stackoverflow possible with the current implementation.
Shared memory locations (atomic registers) come in different variants:

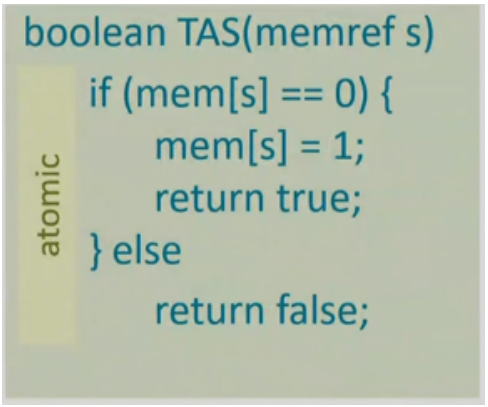
- Multi-Reader-Single-Writer (flag[], label[] in Bakery)
- Multi-Reader-Multi-Writer (victim in Peterson)

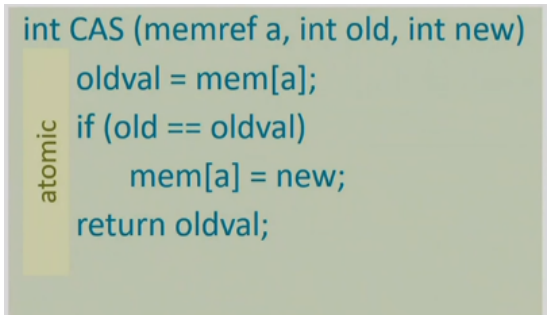
The problem with atomic registers can only have one value which cant be read and written at the same time, it can only be overwritten.
⇒ If S is a atomic read/write system with at least two processes and S solves mutual exclusion with global progress(deadlock-freedom), then S must have at least as many variables as processes

Hardware support for atomic operations: Different architectures use different methods to handle parallelism and atomics. For these architectures there is a common set of instructions used. Atomic instructions arent always used because they are typically much slower than simple read and write operations. Typical instructions are:

- Test-and-Set (TAS)
- Compare-And-Swap(CAS)

Semantics of TAS and CAS: They are Read-Modify-Write (atomic) operations and enable implementation of mutex with $\mathcal{O}(1)$. TAS and CAS are needed for lock-free programming.





Implementation of spinlock using simple atomic operations:

- Test and set
  - Init(lock) → lock = 0;
  - Acquire(lock) → while !TAS(lock); //wait
  - Release(lock) → lock = 0;
- Compare and Swap (CAS)
  - Init(lock) → lock = 0;
  - Acquire(lock) → while (CAS(lock,0,1) != 0); //wait
  - Release(lock) → CAS(lock, 1,0);

  High Level support for atomic operations: In Java there is the java.util.concurrent.atomic.AtomicBoolean (and AtomicInteger,AtomicLong, etc) with the following operations:
  - boolean set(); writes in the variable
  - boolean get(); reads from the variable
  - boolean compareAndSet(boolean expect, boolean update); The memory reference is the object the method is being called on. If the object is the same as expected then we update the object
  - getAndSet(boolean newValue); sets newValue and returns the previous value

The JVM bytecode does not offer atomic operations like CAS, but there is a class sun.misc.Unsafe offering direct mappings from java to underlying machine/OS. Direct mapping to hardware is not guaranteed hence operations on AtomicBoolean are not guaranteed lock-free.

TAS Lock in Java: When state is True the lock is being used. The lock is designed in a first come first serve fashion. The first thread to take the lock can access the CS while the other threads are going through the while loop (hence the name spin lock). The problem is the sequential bottleneck that arises when all threads are competing for the lock (contention: threads fight for the bus during call of getAndSet()) The cache coherency protocol invalidates cached copies of the lock variables on other processors. To solve this problem we can us the textbfTest-and-Test-and Set (TATAS) Lock, which adds a while loop before the compareAndSet to check if the resource can even be accessed hence we have a read-only access instead of an expensive read-write access. TATAS works but Memory ordering leads to race-conditions! (Also known as Double-Checked Locking) Observation:

- (too) many threads fight for access to the same resource
- slows down progress globally and locally

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);
    }
    ...
}
```

Spinlock:

Try to get the lock.
Keep trying until the lock is acquired (return value is false).

unlock
release the lock (set to 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);
}
```

Solution: Make threads go to sleep for a random duration. When the threads exit the while loop they recieve a random number indicating how long they should sleep for. There will be one thread with the lowest number and hence only one thread will try to access the resource at a time. Since all threads get a number at random with the same probability the whole design is fair. This is known as a **Backoff Lock:**

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);
      }
}
```

Graphically determining Deadlocks: Def: Two or more processes are mutually blocked because each process waits for another of these processes to proceed. Graphically threads are denoted T_i and Resources R_i. There is an arrow from a Thread to a lock if the thread attempts to acquire the lock. There is an arrow from a lock to the thread if the thread owns the lock. A deadlock for threads $T_1, ..., T_n$ occurs when the directed graph describing the relation of $T_1, ..., T_n$ and resources $R_1, ..., R_m$ contains a cycle. Deadlocks can, in general not be healed. Releasing locks generally leads to inconsistent state.

Deadlock avoidance:

- Two-phase locking with retry (release when failed) Usually used in databases where transactions can be aborted without consequence
- resource ordering. Usually in parallel programming where global state is modified

When no globally unique ordering is available the following can be done:

```
class BankAccount {
   private static final AtomicLong counter = new AtomicLong();
   private final long index = counter.incrementAndGet();
   ...
   void transferTo(int amount, BankAccount to) {
      if (to.index < this.index)
      ...
   }
}
```

Starvation: the repeated but unsuccesful attempt of a recently unblocked process to continue its execution

0.10 Beyond Locks II: Semaphores, Barrier, Producer-/ Consumer, Monitors

Locks provide menas to enforce atomicity via mutual exclusion, but hey the lack the menas for threads to communicate about changes (e.g changes in the state). Thus they provide no order and are hard to use (if threads A and B lock object X, it is not determined who comes first)

Semaphore: Integer-valued abstract data type S with some initial value $s \geq 0$ and the following operations:

- acquire(S) (atomicly: wait until $S > 0$ then dec(S))
- release(S) (atomicly: inc(S))

Rendezvouz: P and Q executing code. A Rendezvous are locations in code, where P and Q wait for the other to arrive, i,e its a synchronization at a specific point in code. This can be done well with Semaphores: Q can only continue with

Assume Semaphores P_Arrived and Q_Arrived

	P	Q
init	P_Arrived=0	Q_Arrived=0
pre	...	...
rendezvous	release(P_Arrived) acquire(Q_Arrived)	release(Q_Arrived) acquire(P_Arrived)
post	...	..

the code once it acquires P which only becomes possible when P releases and increases S from 0 to 1, hence we have a spinning. We can use blocking queues to implement Semaphores without spinning.

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: Used to synchronize a number of processes. All threads have the same state with respect to the barrier i.e all threads are before or have crossed the barrier. Creating a barrier with Semaphore:

- First Attempt: There are two problems with this implementation, the first one is that we could get a race condition

Synchronize a number (n) of processes.
Semaphore barrier. Integer count.

	P1	P2	...	Pn
init	barrier = 0; volatile count = 0			
pre	...			
barrier	count++ if (count==n) release(barrier) acquire(barrier)	←	←	←
post	...			

Race Condition !

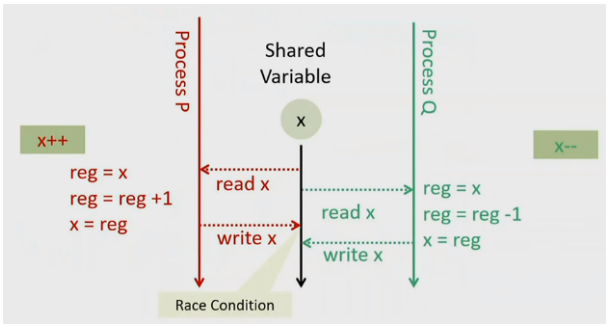
WRONG

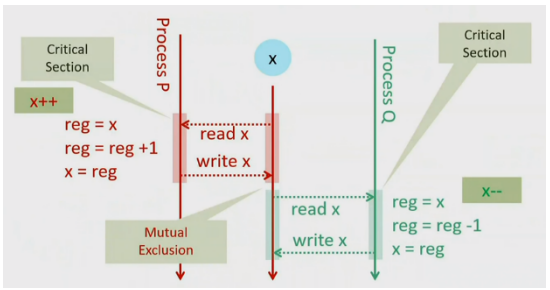
Some wait forever!

if two threads read/write to count at the same time incrementing it once instead of twice. The second problem is, only one release is being done, hence some threads will wait forever! There are 4 Invariants which must be fulfilled for the Barrier to work correclty:

1. Each of the processes eventually reaches the acquire statement
2. The barrier will be opened if and only if all the processes have reached the barrier
3. Count provides the number of processes that have passed the barrier (violated)
4. When all processes have reached the barrier then all waiting processes can continue (violated)

- Second Attempt: We can fix the race condition by adding another Semaphore around count ++. The 4th invariant can be fixed by adding another release(barrier) statement allowing another thread to get through creating a "turnstile" like flow. This works for one iteration but the value of barrier is unknown as we dont know how many threads enter the if statement.
- Third attempt: We want to guarantee that the barrier is in the same state as it was at the beginning once the threads leave. We do this by decreasing the counter back to 0: We now have 3 new Invariants:
  1. Only when all processes have reached the turnstyle it will be opened the first time
  2. When all processes have run through the barrier then count = 0





Semaphores `barrier`, `mutex`. **Integer** `count`.

	P1	P2	...	Pn
init	`mutex = 1; barrier = 0; count = 0`			
pre	`...`			

barrier	`acquire(mutex)`
	`  count++`
	`release(mutex)`
	`if (count==n) release(barrier)`
	`acquire(barrier)`
	`release(barrier)`
		←	←	←

post	`...`			

turnstile

	P1	...	Pn
init	`mutex = 1; barrier = 0; count = 0`		
pre	`...`		

barrier	`acquire(mutex)`
	`  count++`
	`release(mutex)`
	`if (count==n) release(barrier)`
	
	`acquire(barrier)`
	`release(barrier)`
	
	`acquire(mutex)`
	`  count--`
	`release(mutex)`
	`if (count==0) acquire(barrier)`
		←	←

post	`...`		

3. When all processes have run through the barrier then barrier = 0 (violated: race conditions at the if(count==n) and if(count == 0) statements)

- 4th attempt: we add the two if statements into the atomic region (Acquire(mutext)) hence we guarantee they will only be executed once. We also want the barrier to work if applied in loops i.e we have two new Invariants:

1. When all processes have passed the barrier, it holds that barrier = 0;
2. Even when a single process has passed the barrier it holds that barrier = 0 (violated: if a thread in a loop gets further and acquires mutex it can then increment the counter hence it is possible that the barrier will never be 0)

	P1	...	Pn
init	`mutex = 1; barrier = 0; count = 0`		
pre	`...`		

barrier	`acquire(mutex)`
	`  count++`
	`  if (count==n) release(barrier)`
	`release(mutex)`
	
	`acquire(barrier)`
	`release(barrier)`
	
	`acquire(mutex)`
	`  count--`
	`  if (count==0) acquire(barrier)`
	`release(mutex)`		

post	`...`		

Invariants

«When all processes have passed the barrier, it holds that barrier = 0"

« Even when a single process has passed the barrier, it holds that barrier = 0» (violated)

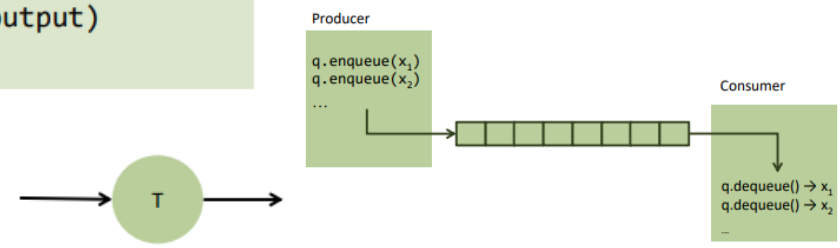
- Final Solution: **underlineTwo-Phase Barrier** By adding another variable "barrier2" we can duplicate and mirror the acquire/release behavior for both barriers also guaranteeing that the Invariants hold.

Producer/Consumer Patter: Thread T_0 computes X and passes it to T_1 which inturn uses X. No synchronization is needed for X because, at any point in time only one thread accesses X we however need a synchronized mechanism to pass X from T_0 to T_1. We can use the pipeline pattern to build data-flow parallel programs.

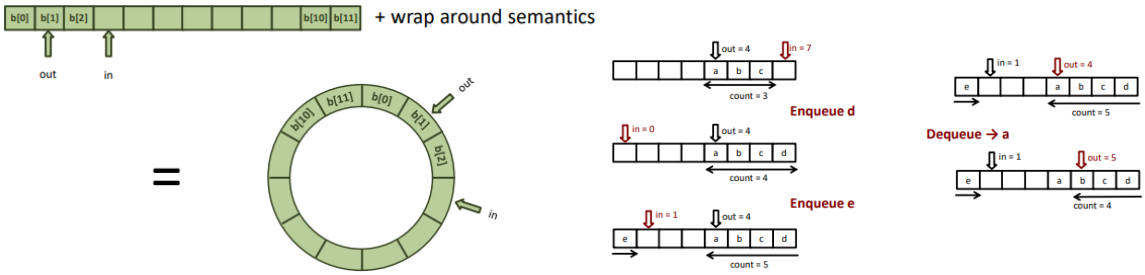
```
init      mutex=1; barrier1=0; barrier2=1; count=0

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
```

Pipeline Node: Has the same structure as a pipeline. The relation between what gets produced and consumed is FIFO (First In First Out)

```
while (true) {
    input = q_in.dequeue();
    output = do_something(input);
    q_out.enqueue(output)
}
```



Bounded FIFO as Circular Buffer: The Queue is implemented with FIFO wraparound semantics. In the helper function isFull() we do not use one element, the benefit is that by not using this element we reduce the total number of variables, reducing the risk of bugs.



0.11 Readers/Writers Lock, Lock Granularity: Coarse Grained, Fine Grained, Optimal, and lazy synchronization

Starvation-freedom a.k.a Fairness: Every thread that tries to make progress, makes progress eventually

Creating Producer/Consumer queues with Ringbuffer: We start out by creating the unsafe version which we then in steps try to make thread safe:

```
public void doEnqueue(long item) {
  buffer[in] = item;
  in = next(in);
}
public boolean isFull() {
  return (in+1) % size == out;
}
```

```
public long doDequeue() {
  long item = buffer[out];
  out = next(out);
  return item;
}
public boolean isEmpty() {
  return in == out;
}
```

- First attempt: We synchronize both the enqueue and dequeue methods but this causes a infinite loop as we have the lock for the queue and have blocked the other action from happening.

```
public synchronized void enqueue(long item) {
     while (isFull())
        ; // wait
     doEnqueue(item);
}
```

```
public synchronized long dequeue() {
     while (isEmpty())
        ; // wait
     return doDequeue();
}
```

Do you see the problem?

→ Blocks forever
infinite loops with a lock held ...

- Second Attempt: We use an unconditional while loop, lock the queue and check if we can enqueue otherwise we sleep without the lock for a given amount of time. Difficulty arises when trying to determine how long the timeout should be.

```
public void enqueue(long item) throws InterruptedException {
  while (true) {
    synchronized(this) {
      if (!isFull()) {
        doEnqueue(item);
        return;
      }
    }
    Thread.sleep(timeout); // sleep without lock!
  }
}
```

What is the proper value for the timeout? Ideally we would like to be notified when the change happens! **When is that?**

- Third Attempt: We use 3 semaphores; nonEmpty,nonFull, manipulation(representing a lock i.e initialized with 1) The above code results in a deadlock (e.g if dequeue is called first then it will acquire manipulation but will wait for

```
import java.util.concurrent.Semaphore;

class Queue {
    int in, out, size;
    long buf[];
    Semaphore nonEmpty, nonFull, manipulation;

    Queue(int s) {
        size = s;
        buf = new long[size];
        in = out = 0;
        nonEmpty = new Semaphore(0); // use the counting feature of semaphores!
        nonFull = new Semaphore(size); // use the counting feature of semaphores!
        manipulation = new Semaphore(1); // binary semaphore
    }
}
```

```
void enqueue(long x) {

    try {
        manipulation.acquire();
        nonFull.acquire();
        buf[in] = x;
        in = (in+1) % size;
    }
    catch (InterruptedException ex) {}
    finally {
        manipulation.release();
        nonEmpty.release();
    }
}
```

```
long dequeue() {
    long x=0;
    try {
        manipulation.acquire();
        nonEmpty.acquire();
        x = buf[out];
        out = (out+1) % size;
    }
    catch (InterruptedException ex) {}
    finally {
        manipulation.release();
        nonFull.release();
    }
    return x;
}
```

nonEmpty to be greater than 0, enqueue will not be able to acquire manipulation.) This can be fixed be switching the order of acquire's:

```
void enqueue(long x) {

    try {
        nonFull.acquire();
        manipulation.acquire();
        buf[in] = x;
        in = next(in);
    }
    catch (InterruptedException ex) {}
    finally {
        manipulation.release();
        nonEmpty.release();
    }
}
```

```
long dequeue() {
    long x=0;
    try {
        nonEmpty.acquire();
        manipulation.acquire();
        x = buf[out];
        out = next(out);
    }
    catch (InterruptedException ex) {}
    finally {
        manipulation.release();
        nonFull.release();
    }
    return x;
}
```

Problems occuring with Semaphores: Semaphores are unstructured and correct use requires high level of discipline. This means that deadlocks can occur quite quickly. I.e we need a lock that we can temporarily escape from when waiting on a condition.

Monitors: Abstract data structure equipped with a set of operations 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

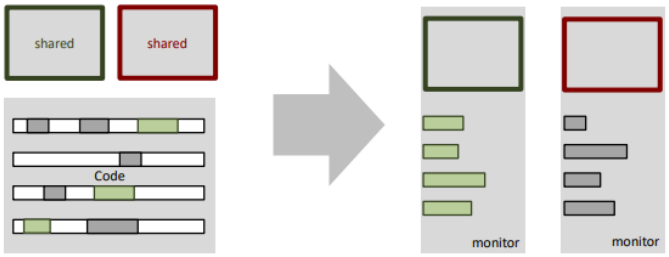
- Release the monitor lock
- Wait for the condition to become true
- Signaling mechanism to avoid busy-loops (spinning)

Uses the intrinsic lock (synchronized) of an object and wait/notify/notify All:

- 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

Different instances of an object have different monitors.

Monitors vs. Semaphores/Unbound Locks: With Semaphores we need to distinguish and guarantee that we use the proper lock when executing certain code. We define Monitors which contain the code which is protected by a given lock.



Producer/Consumer with monitors in Java: We must use a while loop otherwise we risk defying mutual exclusivity. Because we dont know how many threads will be responsible for enqueueing or dequeueing objects we must call notifyAll().

```
synchronized void enqueue(long x) {

   while (isFull())
      try {
         wait();
      } catch (InterruptedException e) { }
   doEnqueue(x);
   notifyAll();
}
```

```
synchronized long dequeue() {
    long x;
    while (isEmpty())
        try {
            wait();
        } catch (InterruptedException e) { }
    x = doDequeue();
    notifyAll();
    return x;
}
```

Wouldn't an if be sufficient?

(Why) can't we use notify()?

Thread States in Java: Diagram indicating how a thread changes states. Green states, are states independent from other states, red ones indicate dependencies

```
R synchronized void enter() {
    if (number <= 0)
       try { wait(); } Q
       catch (InterruptedException e) { };
    number--;
 }

 synchronized void exit() {
P  number++;
    if (number > 0)
       notify();
 }
```

Scenario:

1. **Process P has previously entered the semaphore and decreased number to 0.**
2. **Process Q sees number = 0 and goes to waiting list.**
3. **P is executing exit. In this moment process R wants to enter the monitor via method enter.**
4. **P signals Q and thus moves it into wait entry list (signal and continue!). P exits the function/lock.**
5. **R gets entry to monitor before Q and sees the number = 1**
6. **Q continues execution with number = 0!**

Inconsistency!

Monitor Queues: By definition only one thread can be in the monitor (synchronized). What can happen is that wait is called in the monitor. There are two waiting states:

- waiting entry: when a thread is waiting for the monitor under the assumption that it can do something.
- waiting condition: when a thread knows it cant do anything with the current state of the monitor and is waiting for the monitors state to change (e.g inserting an object into a full queue)

Signal and wait: The signaling process exits the monitor (goes to waiting entry queue) and passes the monitor lock to the signaled process.

Signal and continue: The signaling process continues running and moves the signaled process to waiting entry queue

Implementing a Semaphore: The problem with the the code below is that using an if statement can lead to race conditions, hence we need a while loop.
Rule of Thumb: wait() should always be called in a while loop. What the while loop does is, check at the end of the loop

```
class Semaphore {
   int number = 1; // number of threads allowed in critical section

   synchronized void enter() {
      if (number <= 0)
         try { wait(); } catch (InterruptedException e) { };
      number--;
   }

   synchronized void exit() {
      number++;
      if (number > 0)
         notify();
   }
}
```

Looks good, doesn't it? But there is a problem. Do you know which?

if the condition is met (i.e number ¡= 0). Since the method is synchronized it happens atomicly and hence we cannot get a race condition on the condition number ¡= 0. If, additionally different threads evaluate different conditions, the notification has to be a notifyAll().

Java Interface Lock: Intrinsic Locks ("synchronized") with objects provide a good abstraction and should be the first choice, but there are limitations:

- one implicit lock per object
- we must use them in blocks
- limited flexibility

Java offers the Lock interface for more flexibility:

$$\text{final Lock lock} = \text{new ReentrantLock();}$$

Java Locks provide conditions that can be instantiated (condition interface):

$$\text{Condition notFull} = \text{lock.newCondition();}$$

Java conditions offer:

- .await(): the current thread waits until condition is signaled. This method is called with the lock held, it atomically releases the lock and waits until the thread is signaled. When it returns, it is guaranteed to hold the lock. The thread always needs to check the condition.
- .signal(): wakes up one thread waiting on this condition. This method is called with the lock held.
- .signalAll(): wakes up all threads waiting on this condition. This method is called with the lock held.

Producer/Consumer with explicit Lock: The Producer/Consumer cant be implemented like this with monitors. The disadvantage of this solution is that notFull and notEmpty signal will be sent in any case, even when no threads are

```
class Queue{
    int in=0, out=0, size;
    long buf[];
    final Lock lock = new ReentrantLock();
    final Condition notFull  = lock.newCondition();
    final Condition notEmpty = lock.newCondition();

    Queue(int s) {
        size = s;
        buf = new long[size];
    }

...
}
```

```
void enqueue(long x){

   lock.lock();
   while (isFull())
      try {
         notFull.await();
      } catch (InterruptedException e){}
   doEnqueue(x);
   notEmpty.signal();
   lock.unlock();
}
```

```
long dequeue() {
   long x;
   lock.lock();
   while (isEmpty())
      try {
         notEmpty.await();
      } catch (InterruptedException e){}
   x = doDequeue();
   notFull.signal();
   lock.unlock();
   return x;
}
```

waiting.
Sleeping Barber Variant: The Barber cuts hair, when he is done he checks the waiting room if nobody is left he sleeps. When a client arrives he either enqueues or wakes the sleeping barber. Problems arise when the barber checks, the waiting room is empty and a client enqueus while the barber is sleeping. One solution is to notify the barber regardless of wether he is sleeping or not. This is not idea. The more efficient solution is to add additional counters to check wether processes are waiting:

- $m \leq 0 \iff$ buffer full & $-\,m$ producers (clients) are waiting
- $n \leq 0 \iff$ buffer empty & $-\,n$ consumers (barbers) are waiting

$\Rightarrow$ Producer Consumer, Sleeping Barber Variant:

```
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();

   Queue(int s) {
      size = s; m=size-1;
      buf = new long[size];
   }
...
}
```

sic! cf. slide 27

```
void enqueue(long 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;
}
```

Guidelines for using condition waits:

- Always have a condition predicate
- Always test the condition predicate i.e before calling wait and after returning from wait
- Always call wait in a loop
- Ensure state is protected by lock associated with condition

Read/Writer Locks: An abstract data type for synchronization. This locks state falls into three categtories.

- not held
- held for writing by one thread
- held for reading by one or more threads

i.e we have the following conditions:

- $0 \leq$ writers ≤ 1
- $0 \leq readers$
- $writers \cdot readers == 0$

The Reader/Writer Lock Interface has the following structure:

- new: make a new lock, intitially not held
- acquire_write: block if currently held for reading or writing otherwise make it hold for writing
- release_write: make it not held
- acquire_read: block if currently held for writing otherwise make/keep the hold for writing and increment the readers count
- release_read: decrement readers count, if its 0 make it not held

A simple Monitor- based Implementation: This implementation gives priority to readers: When a reader reads, other

```
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();
 }
}
```

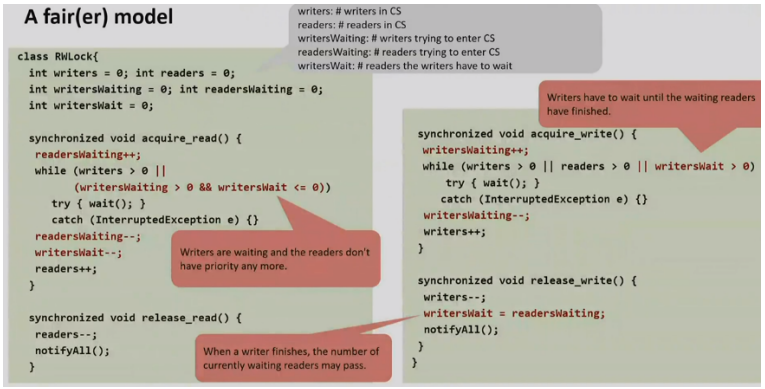
readers can enter and no writer can enter, hence this implementation isnt fair. First attempt to make it fair is add a counter indicating if writers are trying to write and not allowing readers to read until they have written, this results in the exact situation as above with the difference being that now writers have a higher priority. In order to write a fair

```
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();
 }
}
```

implementation we must first define whats fair in this context. For example:

- 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)



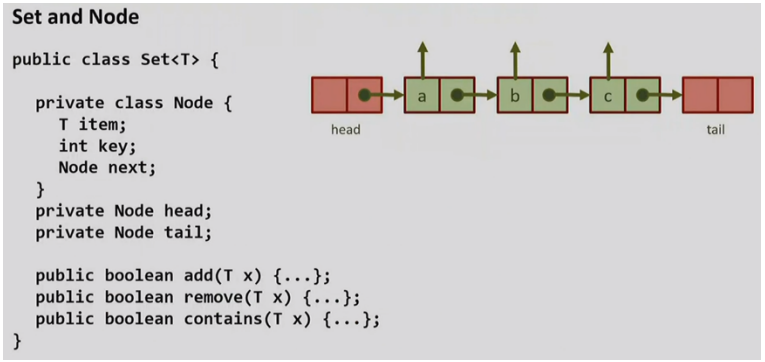
In this implementation the ratio of readers to writers is fair but the ordering of readers is not necessarily given i.e a reader that has been waiting can get passed by a new reader. To get ordering one can implement a queue.
Java's synchronized statement does not support readers/writers, instead use the library:

java.util.concurrent.locks.ReentrantReadWriteLock

It does not have writer priority nor support upgrades from Reader to Writer locks. readLock and writeLock return objects that themselves have lock and unlock methods.

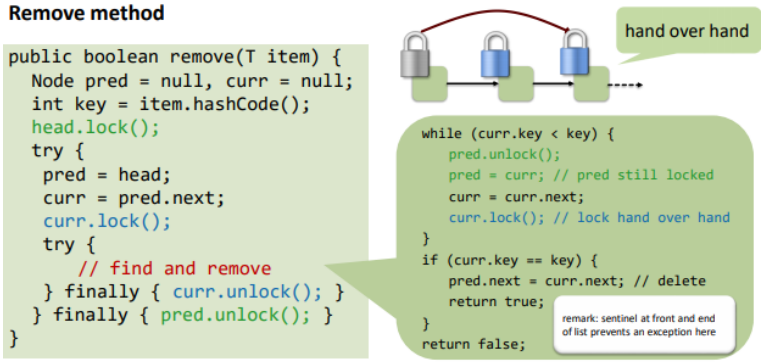
0.12 Lock Granularity

Examples in this section will be done with the datastructure "Sequential List Based Set" which supports Add Remove and Find unique elements in a sorted linked List:



Coarse Grained Locking: We make the list thread safe by synchronizing all the methods. This is simple and effective but creates a bottleneck for all threads (only one thread can work on the list at a time but we have no parrallelism)

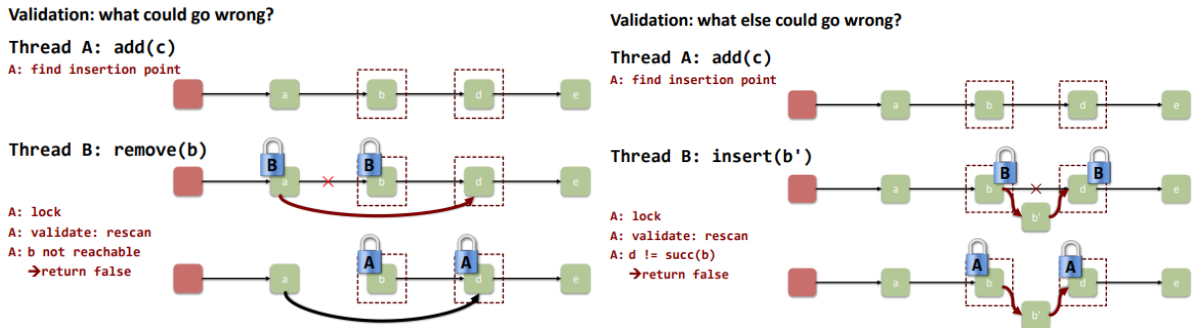
Fine Grained Locking: This technique is often more intricate(elaborate) than visible at a first sight, hence it requires careful consideration of special cases. The idea behind it is splitting the object into pieces with seperate locks (e.g each node in the list gets its own lock). If for example we try to delete a node it is not enough to lock just one node and change its pointer to skip the one which will be deleted, because another thread can try removing the node we locked hence making the original threads delete obsolete. A thread needs to lock both the predecessor and the node to be deleted (hand-over-hand locking). Disadvantages:



- Potentially long sequence of acquire/release before the intended action can take place

- One (slow) thread locking "early nodes" can block another thread wanting to acquire "late nodes"

Optimistic Synchronization: The goal is to try and get as far as possible without locks. We are optimistic and say that the list will remain in the state as we saw it and then check if this is true. Idea: Find the node without locking, then lock nodes and check that everything is ok (validation). This idea doesn't work because before we lock an object with a thread then another thread can delete it. Hence we need to validate that our current element is connected to the beginning of the list i.e we go through the list and see if we can reach the element. We also need to validate that when we insert an



element the two elements we locked are still connected. Hence our validation function becomes: Pros of Optimistic List:

```
private Boolean validate(Node pred, Node curr) {
  Node node = head;
  while (node.key <= pred.key) { // reachable?
     if (node == pred)
          return pred.next == curr; // connected?
     node = node.next;
  }
  return false;
}
```

- No contention on traversals
- Traversals are wait-free (Every call finishes in a finite numver of steps i.e Never waits for other threads)
- Less lock acquisitions

Cons of Optimistic List:

- Need to traverse list twice
- The contains() method needs to acquire locks
- Not starvation-free (threads can block each other preventing other threads from completing)

Lazy Synchronization: Similar to optimistic list but we scan the list only once and the contains() method never locks. We implement this using deleted- markers (markers are fields which can be turned on and off atomicly) and the invariant that every unmarked node is reachable from the head and its predecessor. The nodes are removed lazily i.e the thread that marks the node might not be the same thread that deletes the node. Hence our Lazy list Remove method has the following structure:

- Scan List (as before)
- Lock predecessor and current (as before)
- Mark current node as removed
- Redirect predecessor's next

```
public boolean remove(T item) {
  int key = item.hashCode();
  while (true) { // optimistic, retry
    Node pred = this.head;
    Node curr = head.next;
    while (curr.key < key) {
      pred = curr;
      curr = curr.next;
    }
    pred.lock();
    try {
      curr.lock();
      try {
        // remove or not
      } finally { curr.unlock(); }
    } finally { pred.unlock(); }
  }
}
```

```
if (!pred.marked && !curr.marked &&
     pred.next == curr) {
  if (curr.key != key)
     return false;
  else {
     curr.marked = true;    // logically remove
     pred.next = curr.next; // physically remove
     return true;
  }
}
```

We can now implement a wait free Contains() (No thread can prevent you from working). This reading can now happen while the remove function is active.

Wait-Free Contains

```
public boolean contains(T item) {
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key) {
    curr = curr.next;
  }
  return curr.key == key && !curr.marked;
}
```

Skip List: A practical representation for sets. What we want:

- A datastructure that can be traversed in $\mathcal{O}(n)$ and also parallel
- Collection of elements (without duplicates)
- Interface:
  - add: adds an element
  - remove : remove an element
  - find: search for an element
- Assume there will be many calls to find() and much fewer calls to remove()

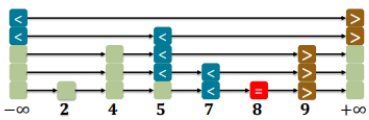
Balanced Trees dont work because:

- rebalancing after add and remove expensive
- rebalancing is a global operation (potentially changing the whole tree)
- particularly hard to implement in a lock- free way

Skip Lists solve theses challenges probabilistically (Las Vegas Style). Skip lists are sorted multi-level lists where the node height is determined probabilisticly e.g $Pr[height = n] = 0.5^n$, no rebalancing is done. Higher level lists are always contained in lower-level lists. Lowest level is the entire list.

Sequential find

```
// find node with value x
// return -1 if not found, node level, succ, and pre otherwise
// pre = array of predecessor node for all levels
// succ = array of successor node for all levels
int find(T x, Node<T>[] pre, Node<T>[] succ)

e.g., x = 8
returns 0
```



The nodes 2 and 4 are not contained int pre as they were never visited during the search. Add, remove and contains work as follows:

Lock Free Programming: Implementing Parallel programs without locks.

Problems with spinlocks:

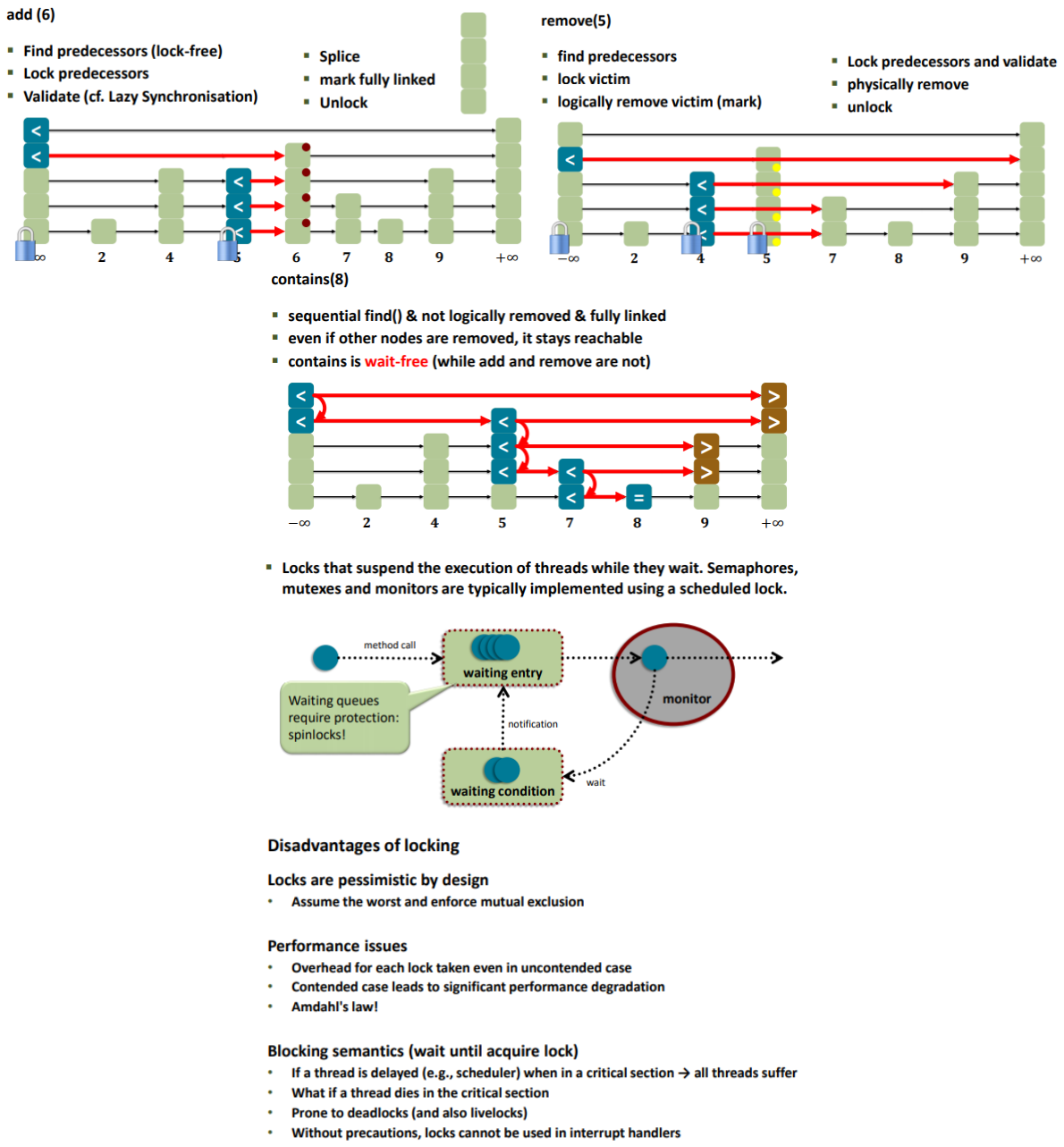
- Scheduling fairness/missing FIFO behavior
- Computing resources wasted, overall performance degraded, particularly for long-lived contention
- No notification mechanism

Locks with waiting/scheduliing: These kinds of locks require support from the runtime system (OS, scheduler) which is expensive. Data structures for scheduled locks need to be protected against concurrent access, again using spinlocks, if not implemented lock-free. Scheduled locks have a higher wakeup latency (need to involve some scheduler)

Lock Performance:

- Uncontended case: When threads do not compete for the lock, their implementation tries to have a minimal overhead (usually just the cost of an atomic operation)
- Contended case: When threads do compete for the lock this can lead to significant performance degradation and starvation

When locking a CS this part of the code becomes sequential and performance is bounded by Amdahls Law!
Disadvantages of locks:



Lock-free synchronisation: Lock-freedom means atleast one thread always makes progress even if other threads run concurrently i.e we have system-wide progress (but not freedom from starvation.) Wait-Freedom means all threads eventually make progress, this implies freedom from starvation. In a non-blocking algorithm failure or suspension of one thread

Progress conditions with and without locks

	Non-blocking (no locks)	Blocking (locks)
Everyone makes progress	Wait-free	Starvation-free
Someone make progress	Lock-free	Deadlock-free

cannot cause failure or suspension of another thread! For non-blocking algorithms we will usually use atomic datatypes or volatile. The following is an example of a non-blocking counter: If two processes see the same value, one of them will win and update it first. Deadlocks are not possible because we arent using locks. Starvation is possible if the thread always loses trying to increment v. If one part(thread) fails the counter still works, this is the key difference to locks. If a thread fails in a lock then are program will fail. Even if we get a positive result of CAS, it is not always true that no other thread has written i.e there can be an overflow and v is incremented until it becomes v again. (ABA problem)

Non-blocking counter

Deadlock/Starvation?

```
public class CasCounter {
    private AtomicInteger value;

    public int getVal() {
       return value.get();
    }

    // increment and return new value
    public int inc() {
       int v;
       do {
             v = value.get();
       } while (!value.compareAndSet(v, v+1));
       return v+1;
    }
}
```

What happens if some processes see the same value?

Assume one thread dies. Does this affect other threads?

Mechanism
(a) read current value v
(b) modify value v'
(c) try to set with CAS
(d) return if success
restart at (a) otherwise

Positive result of CAS of (c) *suggests* that no other thread has written between (a) and (c)

Lock-Free Stack: The first implementation is a Stack using locks where we just synchronize teh methods push and pop. The Lock-free implementation uses Compare and Set method to check if the stack has been "unchanged" (could be that a node is removed and then put back on top) and adds/removes the node. By design Lock free implementations are Deadlock free, but they are not starvation free. A lock-free algorithm does not automatically provide better performance than its blocking equivalent, because Atomic operations are expensive and contention can still be a problem (this can be solved using Backoff)

Stack Node

```
public static class Node {
   public final Long item;
   public Node next;

   public Node(Long item) {
       this.item = item;
   }

   public Node(Long item, Node n) {
       this.item = item;
       next = n;
   }
}
```

Blocking Stack

```
public class BlockingStack {
   Node top = null;

   synchronized public void push(Long item) {
      top = new Node(item, top);
   }

   synchronized public Long pop() {
      if (top == null)
         return null;
      Long item = top.item;
      top = top.next;
      return item;
   }
}
```

Non-blocking Stack

```
public class ConcurrentStack {
   AtomicReference<Node> top = new AtomicReference<Node>();

   public void push(Long item) { … }
   public Long pop() { … }
}
```

Pop

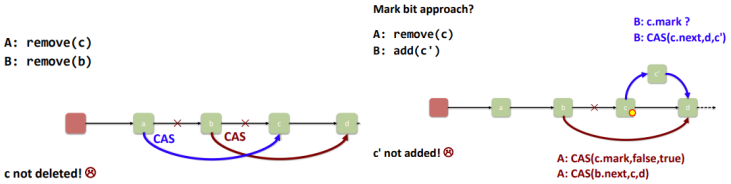
```
public Long pop() {
   Node head, next;

   do {
      head = top.get();
      if (head == null) return null;
      next = head.next;
   } while (!top.compareAndSet(head, next));

   return head.item;
}
```

Push

```
public void push(Long item) {
       Node newi = new Node(item);
       Node head;

       do {
              head = top.get();
              newi.next = head;
       } while (!top.compareAndSet(head, newi));
}
```

Lock Free List Set: Our first attempt is to use CAS. When two threads want to perform an action on the list CAS will decide which one of them will win (the other will try again). Problem arises when trying to remove two items from the list simultaneously one item might not get deleted. Our next approach is to use the Mark bit indicating if the Node is to be deleted or not. We can now succesfully remove two Nodes but problems arise when we try to remove and add Nodes simultaneously. Are key difficulties are:

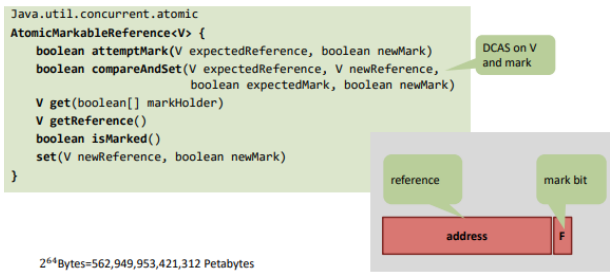
- We cannot/dont want to synchronize via locks
- We want to atomically establish consistency of two things (Mark bit and Next-pointer)



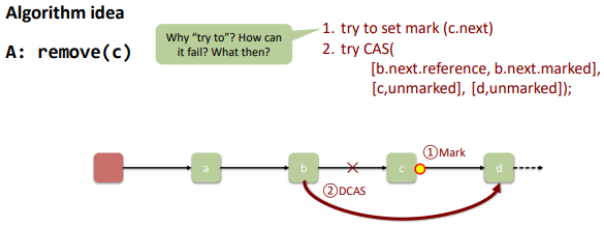
Hence we need a CAS that atomically processes 2 values, unfortunetly this doesnt exist. The solution:
In java we use 64 bit memory addresses, we use the last bit as the Mark bit and the rest as memory address. This works because the JVM knows that this last bit will be used for another purpose. This has been standardised and implemented as:

$$AtomicMarkableReference < V >$$

Because the way we split the memory address we can mimic a DCAS i.e we CAS the Reference and the Mark atomicly. We can now Atomically swing the reference and update the flag. Removing elements happens in two steps, set mark

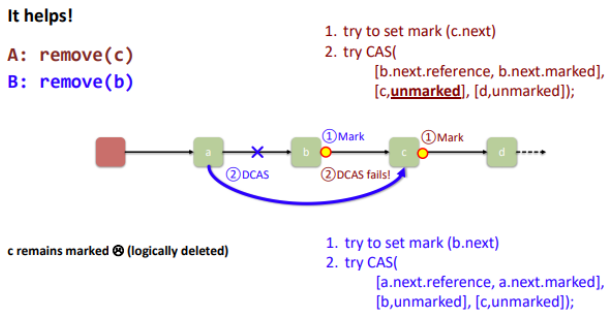


bit in the next field and redirect predecessors pointer. "Try to" fails when another thread inserts a node. 2. works as follows:



- b.next.reference is the object b is pointing to
- b.next.marked is the reference to the pointer

Hence we are checking if b still points to c and if b has not been marked if so we update b's pointer to d and leave b unmarked. When we try to delete two adjacent nodes, one of the DCAS will fail and a node will remain in the list logically deleted (syntactically correct but a waste of space) to fix this we use another thread which was not involved in the process to update a's pointer to d and thus physically removing c aswell. We update the code for the functions find and remove.



The find function does not just find the node as it did in the past but also updates the list in the process by removing marked nodes that weren't physically removed.

Find node

```
public Window find(Node head, int key) {
   Node pred = null, curr = null, succ = null;
   boolean[] marked = {false}; boolean snip;
   while (true) {
      pred = head;
      curr = pred.next.getReference();
      boolean done = false;
      while (!done) {
         marked = curr.next.get(marked);
         succ = marked[1:n]; // pseudo-code to get next ptr
         while (marked[0] && !done) { // marked[0] is marked bit
            if pred.next.compareAndSet(curr, succ, false, false) {
               curr = succ;
               succ = curr.next.get(marked);
            }
            else done = true;
         }
         if (!done && curr.key >= key)
            return new Window(pred, curr);
         pred = curr;
         curr = succ;
} } }
```

loop over nodes until position found

if marked nodes are found, delete them, if deletion fails restart from the beginning

```
class Window {
   public Node pred;
   public Node curr;
   Window(Node pred, Node curr) {
      this.pred = pred;
      this.curr = curr;
   }
}
```

Remove

```
public boolean remove(T item) {
   Boolean snip;
   while (true) {
      Window window = find(head, key);
      Node pred = window.pred, curr = window.curr;
      if (curr.key != key) {
         return false;
      } else {
         Node succ = curr.next.getReference();
         snip = curr.next.attemptMark(succ, true);
         if (!snip) continue;
         pred.next.compareAndSet(curr, succ, false, false);
         return true;
      }
   }
}
```

Find element and prev element from key

If no such element -> return false

Otherwise try to logically delete (set mark bit). ①

If no success, restart from the very beginning

Try to physically delete the element, ignore result ②

Add

```
public boolean add(T item) {
   boolean splice;
   while (true) {
      Window window = find(head, key);
      Node pred = window.pred, curr = window.curr;
      if (curr.key == key) {
         return false;
      } else {
         Node node = new Node(item);
         node.next = new AtomicMarkableRef(curr, false);
         if (pred.next.compareAndSet(curr, node, false, false))
            return true;
      }
   }
}
```

Find element and prev element from key

If element already exists, return false

Otherwise create new node, set next / mark bit of the element to be inserted

and try to insert. If insertion fails (next set by other thread or mark bit set), retry

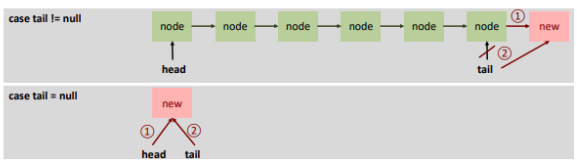
Lock Free Unbounded Queue: In comparison with the Stack we now have two ends. We need queues for implementing Operating systems e.g the scheduler in Linux uses 4 queues to decide which process should be next. Data structures of a runtime or kernel need to be protected against concurrent access by threads and intrerrupt service routines. on different cores. Conventionally (spin)locks are employed for protection. If we want to protect scheduling queues in a lock-free way we need:

- an implementation of a lock-free unbounded queue
- If we want to use queues in a schedulaer usually we cannot rely on Garbage Collection but need to reuse elements of the queue. (ABA Problem)

The Node class has the same structure as the Stack. Blocking Queue is implemented by implementing a sequential version then synchronzing the methods.

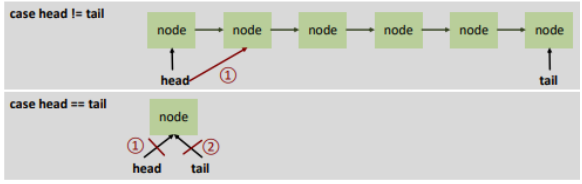
Enqueue

```
public synchronized void Enqueue(T item) {
   Node<T> node = new Node<T>(item);
   if (tail != null)
      tail.next = node;
   else
      head = node;
   tail = node;
}
```



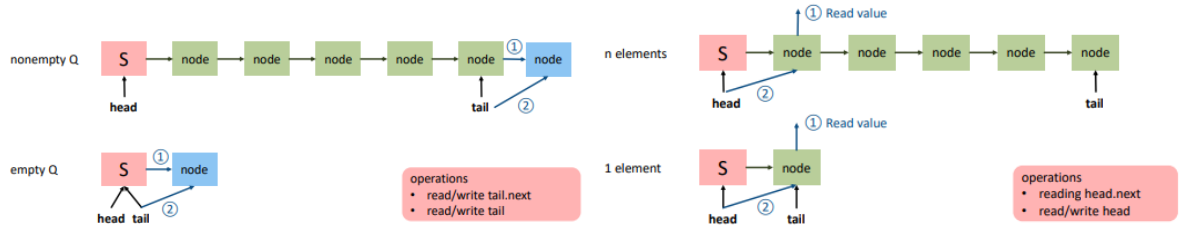
Dequeue

```
public synchronized T Dequeue() {
   T item = null;
   Node<T> node = head;
   if (node != null) {
      item = node.item;
      head = node.next;
      if (head == null) tail = null;
   }
   return item;
}
```



When trying to implement this lock free we run into problems because of potentially simultaneous updates of head, tail and tail.next. To solve this we add a Sentinal (An element which is physically there but logicaly ignored). When using the Sentinal we dont have to worry about head and tail being null at the same time when enqueing an object. When dequeueing an object we replace the sentinal with the Node to be deleted. The Problems that exist in this current idea are:

- We have to update two pointers at a time



- There is possible inconsistency tail might (transiently) not point to the last element
- We dont want any thread to have to wait for consistency to be established (this would be locking camouflages)

We can solve these by using the helper thread principle. Our updated Queue class: The following figures show our initial

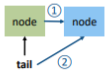
```
public class Node<T> {
      public T item;
      public AtomicReference<Node> next;

      public Node(T item) {
            next = new AtomicReference<Node>(null);
            this.item = item;
      }

      public void SetItem(T item) {
            this.item = item;
      }
}
```

```
public class NonBlockingQueue extends Queue {
  AtomicReference<Node> head = new AtomicReference<Node>();
  AtomicReference<Node> tail = new AtomicReference<Node>();

  public NonBlockingQueue() {
     Node node = new Node(null);
     head.set(node); tail.set(node);
  }

  public void Enqueue(T item);

  public T Dequeue();
}
```

approach for enqueue and dequeue and highlight some of the problems/incosistency aswell as the proper implementation for enqueue and dequeue. In the enqueue method both the if statements are necessary. The first if statement checks if another thread has added another item. The second if statement is to check if we won the race for the last pointer because multiple threads can compete for possesion. Key take away in lock free implementation using CAS is that we only have one chance to break the current progress after that we depend on the help of other threads to restore incomplete actions.

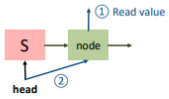
Enqueuer

- read `tail` into `last`
- then tries to set `last.next`:
  CAS(last.next, null, new)
- If unsuccessful retry!
- If successful, try to set tail without retry
  CAS(tail, last, new)



Dequeuer

- read head into `first`
- read `first.next` into `next`
- if `next` is available, read the item value of `next`
- try to set head from `first` to `next`
  CAS(head, first, next)
- If unsuccessful, retry!

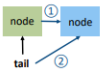


Enqueuer

- read `tail` into `last`
- then tries to set `last.next`:
  CAS(last.next, null, new)
- If unsuccessful retry!
- If successful, try to set tail without retry
  CAS(tail, last, new)

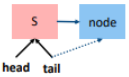
How can this be unsuccessful?
1. Some other thread has written last.next just before me
2. I have read a stale version of tail either
   a) because I just missed the update of other thread
   b) **because the other thread failed in updating tail, for example because it has died**

If the thread dies before calling this, tail is never updated.

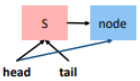


One more possible inconsistency

- Thread A enqueues an element to an empty list, but has not yet adapted tail



- Thread B dequeues (the sentinel)

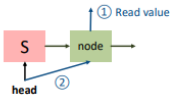


- Now tail points to a dequeued element.

Dequeuer

- read head into `first`
- read `first.next` into `next`
- if `next` is available, read the item value of `next`
- try to set head from `first` to `next`
  CAS(head, first, next)
- If unsuccessful, retry!

How can this be unsuccessful?
1. another thread has already removed next



Final solution: enqueue

```
public void enqueue(T item) {
   Node node = new Node(item);
   while(true) {
      Node last = tail.get();
      Node next = last.next.get();
      if (next == null) {
         if (last.next.compareAndSet(null, node)) {
            tail.compareAndSet(last, node);
            return;
         }
      }
      else
         tail.compareAndSet(last, next);
   }
}
```

Create the new node

Read current tail as last and last.next as next

Try to set last.next from null to node, if success then try to set tail

Ensure progress by advancing tail pointer if required and retry

Help other threads to make progress !

Final solution: dequeue

```
public T dequeue() {
   while (true) {
      Node first = head.get();
      Node last = tail.get();
      Node next = first.next.get();
      if (first == last) {
         if (next == null) return null;
         else tail.compareAndSet(last, next);
      }
      else {
         T value = next.item;
         if (head.compareAndSet(first, next))
            return value;
      }
   }
}
```

Help other threads to make progress !

Read head as first, tail as last and first.next as next

Check if queue looks empty
(1) really empty: return
(2) next available: advance last pointer

If queue is not empty, memorize value on next element and try to remove current sentinel

Retry if removal was unsuccessful

0.13 Reuse and the ABA Problem

When using CAS we always "memorize" a global variable (in a local variable) then update it only if the global variable is the same as our memory

Node Reuse: Garbage collection and "new" are slow operations. The solution is to create a node pool. E.g when using a stack instead of popping and waiting for garbage collection, we add the Node to the NodePool such that its available, hence giving us control of the memory management. We can implement NodePool as a stack, hence we move Nodes back and forth from the Stack and NodePool. The image to the left is the get and put methods for NodePool, the right image is the changes which need to be made to our stack push and pop methods. When we run this we get about a double speedup,

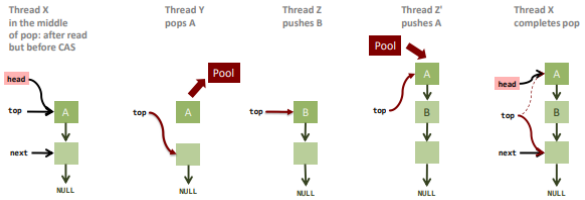
NodePool put and get

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

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

```
public void push(Long item) {
   Node head;
   Node new = pool.get(item);
   do {
      head = top.get();
      new.next = head;
   } while (!top.compareAndSet(head, new));
}

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

but there are cases where a Runtime Exception is thrown or our methods dont return.

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. The example above



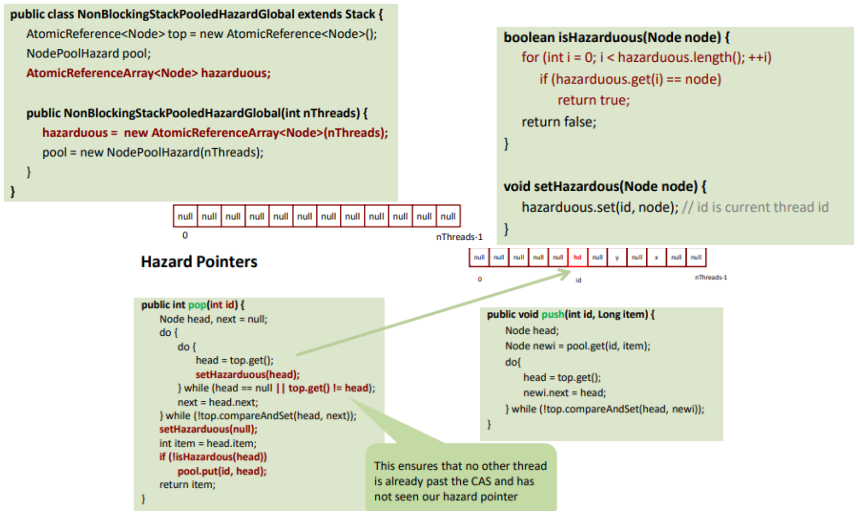
illustrates the ABA problem. Thread X creates the head Pointer to A, before the CAS Thread Y pops A (i.e A is on its way to the pool). Thread Z comes before A reaches the pool and pushes a Node B. A reaches the Pool, and is then pushed on the stack by Z. X now gets the chance to complete the pop. Because head still points to A it sets the top pointer to the next element which it read when it started the pop, hence B is unwantingly deleted from the stack.
Solutions:

- DCAS(double compare and swap): Not available on most platforms
- Garbage Collection: Just dont use the NodePool and implement a lock-free Garbage Collection. This wouldnt work because it would result in a cyclic dependancy
- Pointer Tagging(Half solution): This solution does not cure the problem, rather delay it. It can be practical but does have an overflow problem.
- Hazard Pointers
- Transactional memory

Pointer Tagging: ABA problem usually occurs with CAS on pointers. Aligned addresses(values of pointers) make some bits available for pointer tagging. E.g if a pointer is aligned modulo 32 then 5 bits are available for tagging. Each time a pointer is stored in a data structure we increase the tag by one. We can access the data structure via the address x-(x mod 32). The CAS will fail because the pointer is changed. This makes the ABA problem less probable because now 32 versions of each pointer exist. (A problem could occur when it wraps around and hence the pointer is back to 0)

Hazard Pointers: The ABA problem stems from reuse of a pointer P that has been read by some thread X but not yet written with CAS by the same thread. Modification takes place meanwhile by some other thread Y. Solution:

- before X reads P, it marks it hazardous by entering it in one of the n (number of threads) slots of an array associated with the data structure. When finished (after the CAS), process X removes P from the array. Befor a process Y tries to reuse P it checks all entries of the hazard array.



The above implementation protects the stack from the ABA problem, but our Node Pool is still vulnerable. This is easier to fix because our Node Pool is not a global object. Two solutions for the Node Pool are:

- Thread-local node pools. That means no protection is necessary but also no guarantee for well balanced push/pop operations
- Hazard pointers on the global node pool: this is an expensive operation for node reuse. Its equivalent to the code above i.e node pool returns a node only when it is not hazardous

In practice we use a hybrid of both. Hence each thread has its own pool, when it runs out then we check the global pool, only if the global pool is empty to we allocate new memory.

Method Calls: A 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. This is where we see the difference between sequential and concurrent programming. "Global Clock" means that all observers see the same object in the same state. "Object Clock" on the other

Sequential	Concurrent
Meaningful state of objects only **between method calls**.	Method **calls can overlap**. Object might **never** be **between** method calls. Exception: periods of *quiescence*.
Methods described in **isolation**.	All possible **interactions** with concurrent calls must be taken into account.
Can **add new methods** without affecting older methods.	Must take into account that everything can **interact** with everything else.
"**Global** clock"	"**Object** clock"

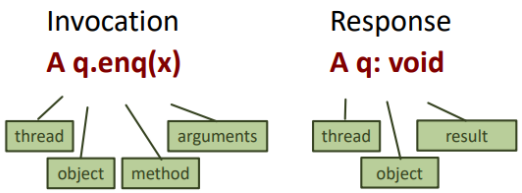
hand refers to the fact that in concurrent programming different observers may see the same object in different states.

0.14 Linearizability

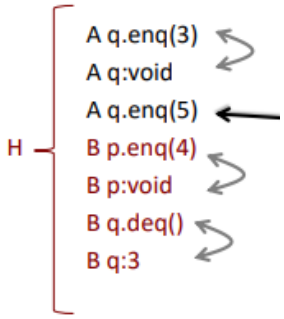
Linearizability: Each method should appear to take effect instantaneously between invocation and response events. An object for which this is true for all possible executions is called linearizable. The object is correct if the associated sequential behavior is correct. We talk about executions in order to abstract away from actual method content. The linearization points can often be specified, but may depend on the execution (not only the source code) e.g if the queue is empty a dequeue may fail, while it does not fail with a non-empty queue. CAS are usually linearization points. The code below has 2 linearization points. One at the if condition and the other at items.get(...). Formaly:

```
public int deq() throws EmptyException {
  if (tail == head)
    throw new EmptyException();
  int x = items.get(head++ % capacity);
  return x;
}
```

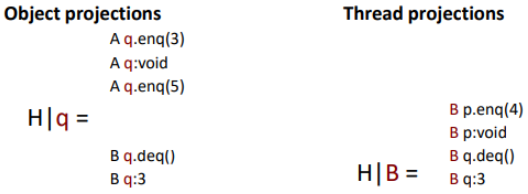
Split method calls into two events. Notation:



History: History H = sequence of invocations and responses. Invocations and response match if thread names agree and object names agree. An invocation is pending if it has no matching response. A subhistory is complete when it has no pending responses. In the image below A.q.end(5) is pending.



Projections: We can either project on objects or Threads. When we project a history on an object or thread we only look at the history involving that object/thread



Sequential histories: Method calls of different threads do not interleave. A final pending invocation is ok.

Well formed histories: Per thread projections are sequential i.e threads invoke and respond. You can check if a history is well formed by creating the projections of the threads and checking if the projections are sequential (if they are then it is well formed/correct)

Equivalent histories: If for two histories the projections are equivalent. i.e

$$H|A = G|A \text{ bzw } H|B = G|B$$

Legal histories: Sequential specification tells if a single-threaded, single object history is legal (e.g pre-/post conditions). A sequential history H is legal, if for every object x $H|x$ adheres to the sequential specification of x.

Precedence: A method call A precedes another method call B if the response event of A precedes the invocation event of B. If no precedence then method calls overlap. Notation:

- History H and method executions m_0, m_1 on H
- $m_0 \rightarrow_H m_1$ means m_0 precedes m_1
- $\rightarrow_H$ is a relation and implies a partial order on H. The order is total when H is sequential (i.e no overlapping methods)

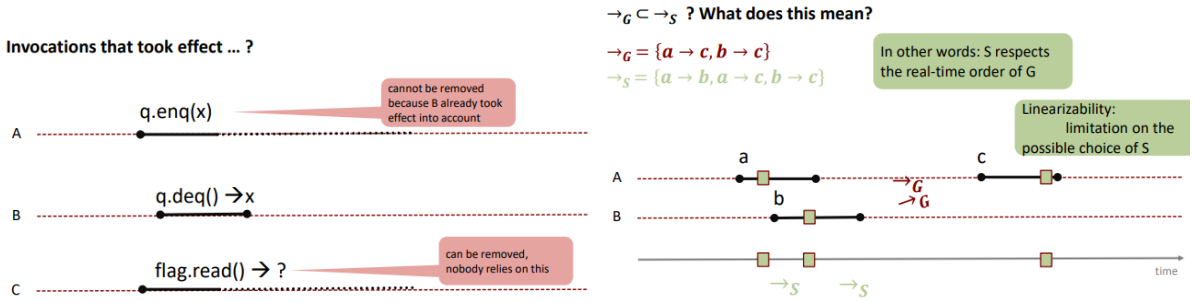
Linearizability: History H is linearizable if it can be extended to a history G by:

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

such that G is equivalent to a legal sequential history S with:

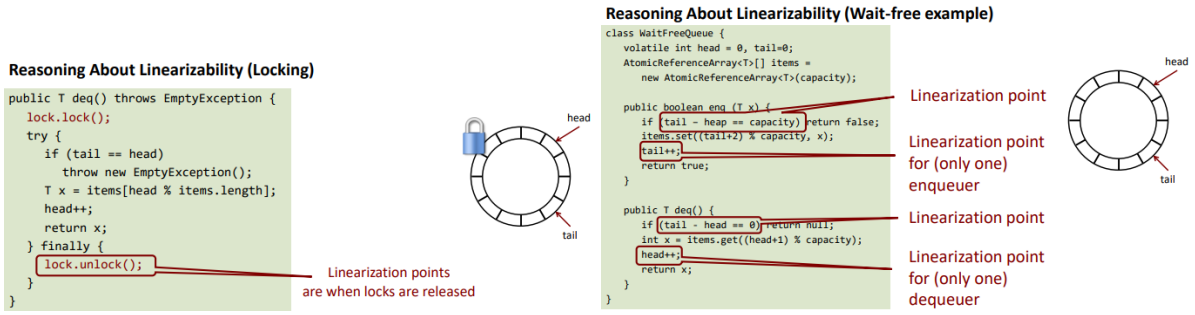
$$\rightarrow_G \subset \rightarrow_S$$

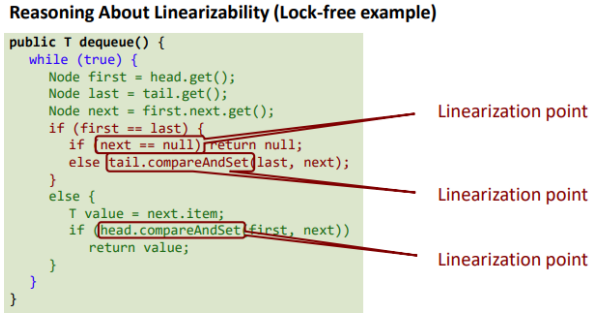
An Invocation took effect if there is dependencies between the threads e.g: (in the above example we chose that a returns before b invokes)



0.15 Composability

Composability Theorem: History H is linearizable iff for every object x $H|x$ is linearizable. Hence we have Modularity of linearizable objects. The Linearizability of objects can be proven in isolation. Independently implemented objectes can be composed(modular). E.g Atomic Registers:
They are memory locations for values of primitive types with the operations read and write. Atomic Registers are linearizable with a single linearization point i.e they are sequentially consistent, every read operation yields the most recently written value. for non-overlapping operations, the realtime order is respected (if operation a is before b then in realtime this will also be true)More examples: The linearization point, is the point where the method "happens" i.e the change is visible





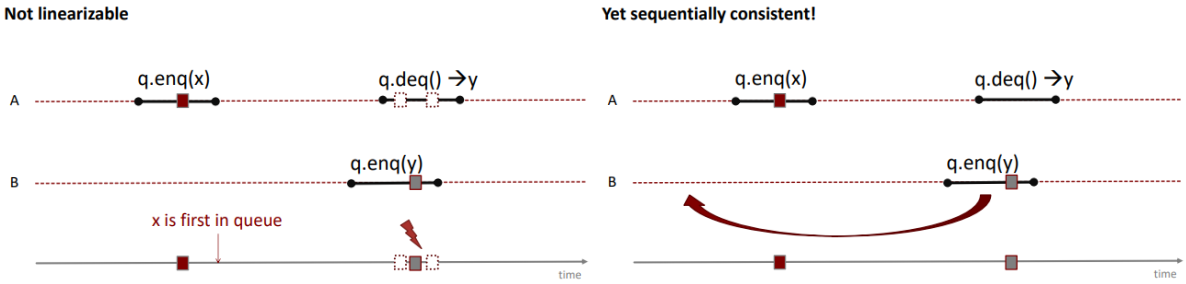
for the other threads. In the above examples, when using locks the linearization points are when the locks are released(locks are not released upon exceptions being thrown), for the wait-free queues its when the head/tail is updated(before the return) because then the other thread sees a change or at the failed attempt(i.e queue full or empty). Lock-free can be distinguished from wait free by the while loop that the lock-free implementation has.

0.16 Sequential Consistency

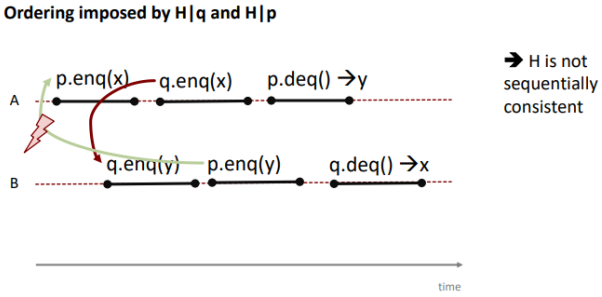
Sequential Consistency: Is weaker version of Linearizability. The difference is that:

$$\rightarrow_G \subset \rightarrow_S$$

is not required i.e no order across threads required. Sequential Consistency requires that operations done by one thread respect program order.this means there is no need to preserve real-time order hence we cannot re-order operations done by the same thread but we can reorder non overlapping operations done by different threads. The following example of a FIFO queue shows the "weakness" of the sequential consistency model

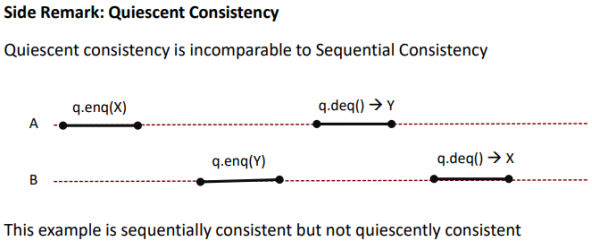


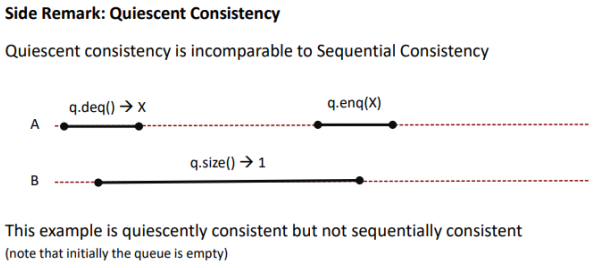
Sequential Consistency is not a local property hence we lose composability. Example: This example shows that $H|q, H|p$



are sequentially consistent but there composition is not (cycle generated). We use Sequential Consistency in order to ignore the parallel interleavings of the code.

Quiescent Consistency: Idea: Programs should respect real-time order of algorithms separated by periods of quiescence i.e quiescent consistency requires non-overlapping methods to take effect in their real-time order. Quiescent consistency is incomparable to sequential consistency. Examples If there is no quiescence than anything may happen, hence it is



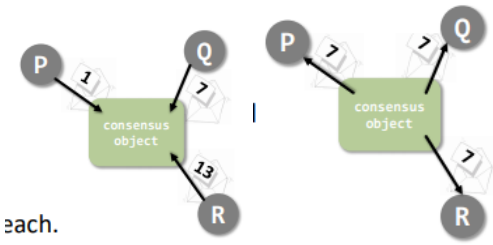


quiescently consistent (example to the right)

0.17 Consensus

Consensus: An object with the following interface:

public interface Consensus< T > {
T decide (T value);
}



A number of threads call c.decide(v) with an input value v each. Each Consensus object has a Consensus protocol with the following requirements:

- **wait-free:** consensus returns in finite time for each thread
- **consistent:** all threads decide the same value
- **valid:** the common decision value is some thread's input.

⇒ Linearizability of consensus must be such that first threads decision is adopted for all threads

Consensus number: A class C solves n-thread consensus if there exists a consensus protocol using any number of objects of class C and any number of atomic registers. Consensus number of C: largest n such that C solves n-thread consensus.
Theorem: Atomic Registers have a consensus number 1
Corollary: There is no wait-free implementation of n-thread consensus, n¿1, from read-write registers. Theorem: Compare-And-Swap/Set has infinite consensus number. Theorem: There is no wait-free implementation of a FIFO queue with atomic

Proof by construction

```
class CASConsensus {
   private final int FIRST = -1;
   private AtomicInteger r = new AtomicInteger(FIRST); // supports CAS
   private AtomicIntegerArray proposed; // suffices to be atomic register

   … // constructor (allocate array proposed etc.)

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

registers.
Proof:

- Given: A consensus protocol from queue and registers
- Assume: There exists a queue implementation from atomic registers
- Substitution yields: A wait-free consensus protocol from atomic registers. Atomic registers have consensus number 1⇒ CONTRADICTION

Consensus Hierarchy: Objects with a lower Consensus number can be implemented with objects of a higher Consensus number but not the other way around.

0.18 Transactional Memory

Motivation: Programming with locks is too difficult and Lock-free programming is even more difficult. **Goal:** Remove the burden of synchronization from the programmer and place it in the system(hardware/software)

Problems with Locks:

- **Deadlocks:** Threads attempt to take common locks in different orders
- **Convoying:** Thread holding a resource R is descheduled while other threads queue up waiting for R
- **Priority Inversion:** Lower priority thread holds a resource R that a high priority thread is waiting on

Locks are not composable.

atomic blocks/transactions: The programmer says which block should be performed atomically and let the software/hardware take care of the how. i.e programmer is concerned with:

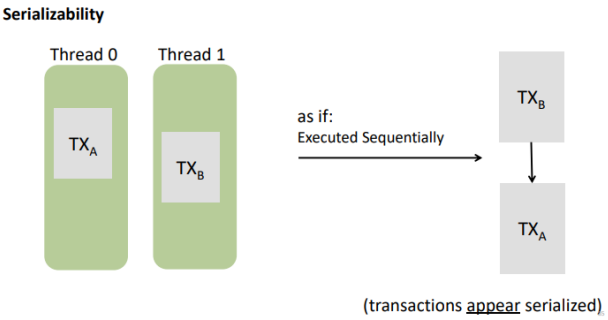
- what: what operations should be atomic
- but not how

Benefits of Transactional Memory (TM):

- simpler and less error-prone code
- higher-level (declarative) semantics (what vs how)
- composable
- optimistic by design (does not require mutual exclusion)

TM semantics:

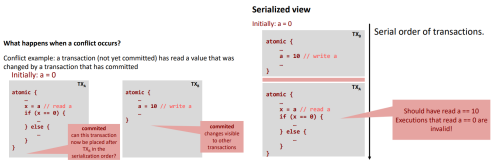
- **Atomicity:** Changes made by a transaction are made visible atomically other threads preserve either the initial or the final state, but not any intermediate states. Locks enforce atomicity via mutual exclusion, while transactions do not require mutual exclusion
- **Isolation:** Transactions run in isolation. While a transaction is running, effects from other transactions are not observed (as if transaction takes a snapshot of the global state when it begings and then operates on that snapshot)
- **Serializability:**



TM implementation:

- First Attempt: Big lock around all atomic sections, gives (nearly all i.e durability is missing) desired properties but not scalable. (not done in practice for obvious reasons)
- Second Attempt: Keep track of operations performed by each transaction i.e we have concurrency control and the system ensures atomicity and isolation properties.

Transactions can be aborted. Issues like the following are handled by Concurrency Control(CC) mechanism. When a transaction aborts, it can be retried automatically or the user is notified. The transactional memory system guarantees



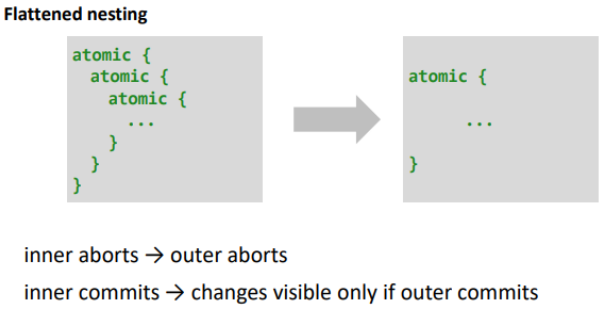
that consistent data will always be seenh by a running transaction. Possible solution is to take a snapshot when a transaction is starting and in the case of an inconsistency we abort early. Hardware TM (HTM) is fast but, bounded by resources and can often not handle big transactions. Software TM (STM) is done in the programming language, it offers greater flexibility but achieving good performance might be challenging (because at some point locks or atomics will be necessary)

Strong vs. weak isolation: What happens when shared state accessed by a transaction, is also accessed outside of a transaction? Are the transaction guarantees still maintained?

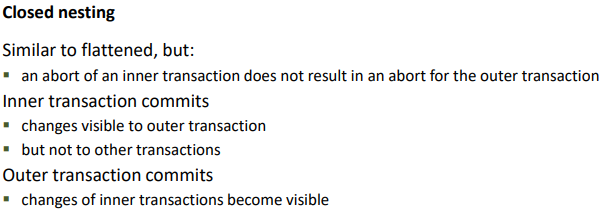
- Strong isolation: Yes (It is easier for the programmer because they dont need to check whats in the atomic blocks, but its difficult to implement)
- Weak isolation: No

Nesting: What are the semantics of nested transactions?

- Flat nesting



- Closed nesting



Reference-based STMs: Mutable state is put into special variables, which can only be modified inside a transaction. Everything else is immutable (or not shared). Callable is used to return a type.

Bank account (scala-stm)

```
class AccountSTM {
  private final Integer id;       // account id
  private final Ref.View<Integer> balance;

   AccountSTM(int id, int balance) {
      this.id      = new Integer(id);
      this.balance = STM.newRef(balance);
  }

}
```

Real world: bank account in scala-stm

```
void withdraw(final int amount) {
     // assume that there are always sufficient funds...
     STM.atomic(new Runnable() { public void run() {
         int old_val = balance.get();
         balance.set(old_val - amount);
     }});
}

void deposit(final int amount) {
     STM.atomic(new Runnable() { public void run() {
         int old_val = balance.get();
         balance.set(old_val + amount);
     }});
}
```

Bank account transfer

```
static void transfer(final AccountSTM a,
                     final AccountSTM b,
                     final int amount) {
     atomic {
          a.withdraw(amount);
          b.deposit(amount);
     }
}
```

What if account a does not have enough funds?
How can we wait until it does in order to retry the transfer?
locks → conditional variables
TM → retry

Ideal world: bank account using atomic keyword

```
void withdraw(final int amount) {
     // assume that there are always sufficient funds...
     atomic {
         int old_val = balance.get();
         balance.set(old_val - amount);
     }
}

void deposit(final int amount) {
     atomic {
         int old_val = balance.get();
         balance.set(old_val + amount);
     }
}
```

```
public int getBalance() {
  int result = STM.atomic(
    new Callable<Integer>() {
    public Integer call() {
      int result = balance.get();
      return result;
    }
  });
  return result;
}
```

"atomic"

Bank account transfer with retry

```
static void transfer_retry(final AccountSTM a,
                           final AccountSTM b,
                           final int amount) {

     atomic {
          if (a.balance.get() < amount)
               STM.retry();
          a.withdraw(amount);
          b.deposit(amount);
     }
}
```

retry: abort the transaction and retry when conditions change

Retry: Implementations need to track what reads/writesd a transaction performed to detect conflicts. This is typically called read-/write-set of a transaction. When retry is called the transaction aborts and will be retried when any of the variables that were read change. (above when a.balance is updated the transaction will be retried)

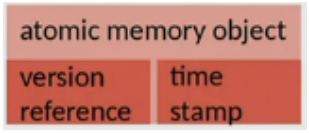
STM implementation: Theads that run transactions need three states:

- Active: Transaction is running
- Aborted: Transaction was aborted
- Comitted: Transaction was comitted and now waiting for the next one

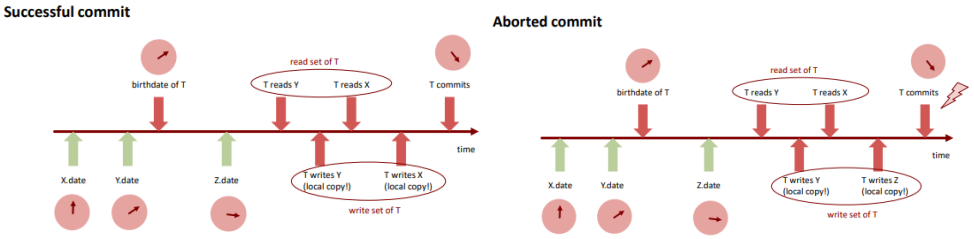
We need Objects representing state stored in memory (the variables affected by a transaction). These objects have read, write and copy methods.
Commit:

- Lock all objects of read- and write-set (in some defined order to avoid deadlocks)
- Check that all objects in the read set provide a time stamp $\leq$ birthdate of the transactionm otherwise return abort
- Increment and get the value T of current global clock
- Copy each element of the write set back to global memory with timestamp T
- Release all locks and return "commit" (at this point its visible to all other threads)

Clock-based STM System: We have a global clock that moves each time a transaction is comitted, this way we can give each transaction a time stamp.



Atomic Objects: Each transaction uses a local read-set and a local write-set holding all locally read and writtend objects (the objects themselves are global) When the transaction calls read we check if the object is in the write set, if yes then we do not take the original version in memory, but the this newest version. The written version can only come from our write-set i.e which variables have we written to and what value was written. Hence when we call read we check our transaction and if we have already written it. Otherwise we check if the objects time stamp $\leq$ transactions birthdate (time stap when we started the transaction), if not then we know another thread has completed a transaction so we throw an aborted exception. If time stamp $\leq$ transactions birthdate then we add a new copy of the object to the read set.
When a Transaction calls a write, if it is not in the write set we create a copy of it in the write set.



Dining Philosophers: Solves the problem using TM:

Dining philosophers

- 5 philosophers
- 5 forks
- each philosopher requires 2 forks to eat
- forks cannot be shared

Solution that can lead to deadlock

Philosopher:

- think
- lock left
- lock right
- eat
- unlock right
- unlock left

P_1 takes F_1, P_2 takes F_2, P_3 takes F_3, P_4 takes F_4, P_5 takes F_5
→ Deadlock

Dining Philosophers Using TM

```
Fork[] forks = new Fork[tableSize];

for (int i = 0; i < tableSize; i++)
      forks[i] = new Fork();

PhilosopherThread[] threads =
      new PhilosopherThread[tableSize];

for (int i = 0; i < tableSize; i++)
      threads[i] = new PhilosopherThread(
                        forks[i],
                  forks[(i + 1) % tableSize]);
```

Dining Philosophers Using TM

```
private static class Fork {
      public final Ref.View<Boolean> inUse = STM.newRef(false);
}
class PhilosopherThread extends Thread {
      private final int meals;
      private final Fork left;
      private final Fork right;

      public PhilosopherThread(Fork left, Fork right) {
            this.left = left;
            this.right = right;
      }

      public void run() { … }
}
```

Dining Philosophers Using TM

```
class PhilosopherThread extends Thread {
   …
      public void run() {
            for (int m = 0; m < meals; m++) {
                  // THINK
                  pickUpBothForks();
                  // EAT
                  putDownForks();
            }
      }
   …
}
```

Dining Philosophers Using TM

```
class PhilosopherThread extends Thread {
   …
      private void pickUpBothForks() {
            STM.atomic(new Runnable() { public void run() {

                  if (left.inUse.get() || right.inUse.get())
                        STM.retry();

                  left.inUse.set(true);
                  right.inUse.set(true);

            }});
      }
   …
}
```

Dining Philosophers Using TM

```
class PhilosopherThread extends Thread {
   …
      private void putDownForks() {
            STM.atomic(new Runnable() { public void run() {

                  left.inUse.set(false);
                  right.inUse.set(false);

            }});
      }
   …
}
```

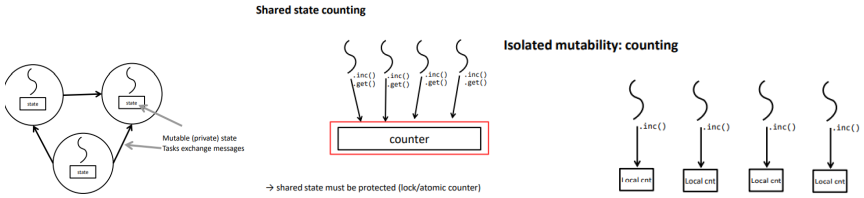
Issues with transactions: It is not clear what are the best semantics for transactions. Getting good performance can be challenging. I/O cannot be done in transactions (when do we print??)

0.19 Distributed Memory & Message Passing

Many of the problems of parallel/concurrent programming come from sharing state. (Complexity of locks, race conditions,...). Some Alternatives:

- Functional Programming: Immutabel state i.e variables either have a value or dont. Each calculation needs a new object, hence there are memory problems and one needs a good garbage collecter.
- Message Passing: Isolated mutable state: State is mutable locally, but not shared i.e each thread/task has its private state. Since different tasks dont see what the other is doing, we need to communicate via message passing. The idea is that we do not write in the memory of the other thread, but leave a message at a designated location. The thread can check if there is update when it wants.

Shared vs Distributed memory: In shared memory architectures (e.g multicores) both message passing and shared states are used. Message passing in shared memory can be slower than sharing data but is easy to implement and reason about.
In Distributed memory architectures (e.g datacenters,supercomputer, clusters) state shareing is challenging and often inefficient. Almost only use message passing. There are additional concerns which must be taken into account e.g failures.



Rethinking managing state

Bank account
- Sequential programming
  - *Single balance*
- Parallel programming: sharing state
  - *Single balance + protection*
- Parallel programming: distributing state
  - *Each thread has a local balance (a budget)*
  - *Threads exchange amounts at coarse granularity*

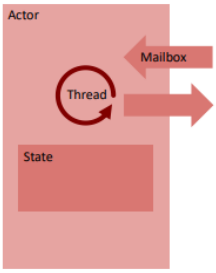
Synchronous Messaging: Sender blocks until the message is recieved

Asynchronous Messaging: Sender does not block and puts message into a buffer for reciever to get.

The Actor Model: Actor = computational agent that maps communication to:

- a finite set of communications sent to other actors (messages)
- a new behavior (State)
- a finite set of new actors created (dynamic reconfigurability)

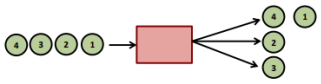
The actor model has an undefined global ordering (i.e each actor can be looked/ran at seperatly) and Asynchronous Message Passing. The Actor model provides a dynamic interconnection topology which allows to dynamically configure the graph during runtime(add channels) and dynamically allocate resources. An actor sends messages to other actors using direct naming withot indirection via port/channel/queue etc.



example distributor:

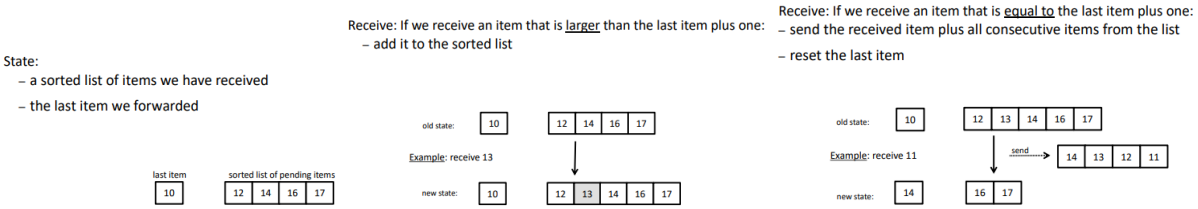
- Forward received messages to a set of nodes in a round-robin fashion

State:
- an array of actors
- the array index of the next actor to forward a message



Receive:
- messages → forward message and increase index (mod)
- control commands (e.g., add/remove actors)

example Serializer:



Event-driven programming model: Typically actors react to messages. A program is written as a set of event handlers for events (events can be seen as recieved messages)

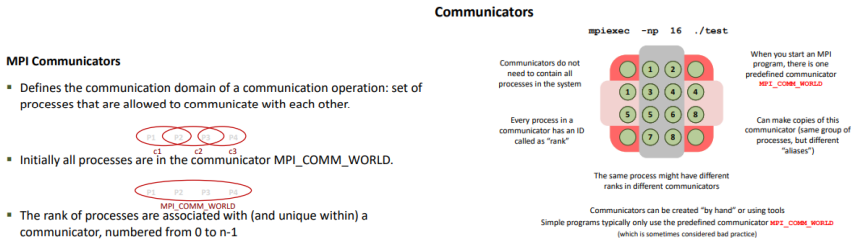
Communicating Sequential Processes (CSP): Formal language defining a process algebra for concurrent systems. The difference from the actor model, is that in the actor model, messaging is done between the actors whereas in CSP we create a channel that connects the actor with a process and send the message through this process. A process can have multiple input/output channels. Channels can also be send i.e let another actor use the channel. This allows us to organise the channels i.e have channels for specific message types.

0.20 Message Passing Interface (MPI)

MPI is a standard API. It is implemented as a library which can be used by multiple programming languages. The Intuition behind the MPI is that you write your program which uses the MPI library which can then be used for standard or specialized connections. MPI processes can be collected into groups:

- Each group can have multiple colors (sometimes called context)
- Group + color == communicator (a name for the group. It is like an object which includes a group of processes)
- When an MPI application starts, the group of all processes is initially given a predefined name called MPI_COMM_WORLD

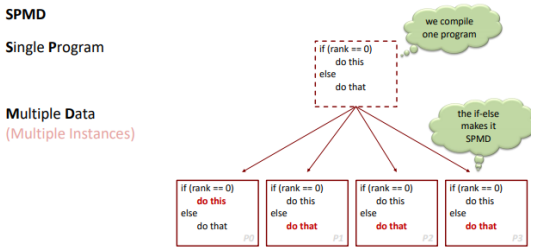
rank: The unique number identifying a process within each communicator. For two different communicators, the same process can have two different ranks:so the meaning of a "rank" is only defined when you specify the communicator.



Every MPI programm must start with: MPI.Init(args)
(This is directly after the public static void main(String args[])) and ends with MPI.Finalize();
We must declare these methods because MPI is a library (are language does not know that we want to use the library).
Methods:

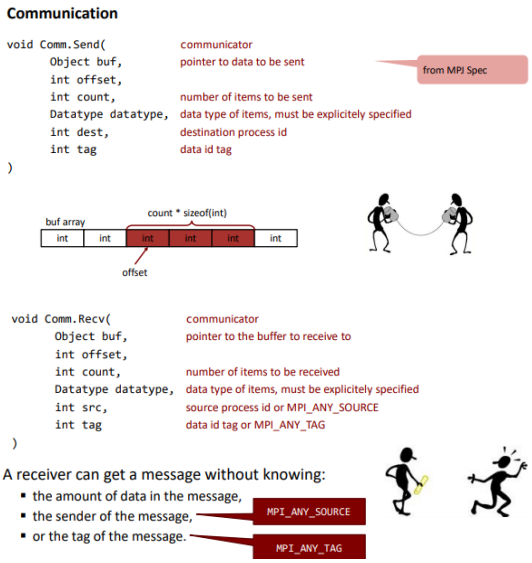
- MPI.COMM_WORLD.Size(); returns the number of processes
- MPI.COMM_WORLD.Rize(); returns the rank of each process (unique for each process)

Single Program Multiple Data (SPMD): With the rank we can implement SPMD i.e use the same code but use multiple instances which will execute differently depending on the rank of the process.



Communication: With message passing variables are not shared between processes (no race conditions!) We send messages between processes using the Comm.send(...) method We recieve messages using Comm.Recv

Message Tags: Communicating processes may need to send several messages between each other. Message tags are integers which allow us to differentiate between different messages being sent.



Message matching: For a process to send a message to a reciever validly then three things must be fulfilled:

- The communicator of sender must be identical with the communicator of the reciever
- The tags must be identical
- The destination and source must align

Synchronous Message Passing: Synchronous send (Ssend) waits until complete message can be accepted by receiving process before completing the send.
Synchronous receive (Recv), waits until expected message arrives.(recieve is always synchronous)
Synchronous routines can perform two actions: transfer data and synchronize processes

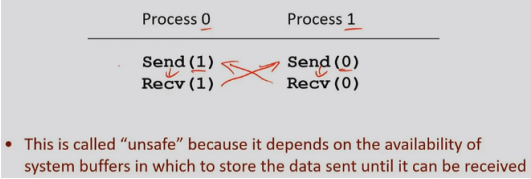
Asynchronous Message Passing: Send does not wait for actions to complete before returning. This requires local storage for messages, somtimes explicit (programmer needs to care) and sometimes implicit(transparent to the programmer). In general no synchronisation which allows local progress to be made.

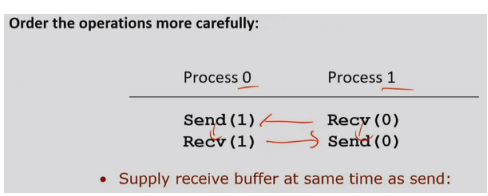
Blocking: Return after local actions are complete, though the message transfer may not habe been completed

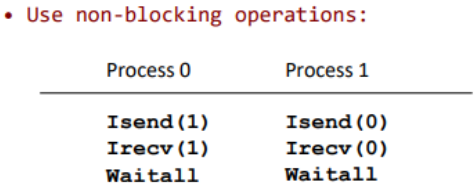
Non-Blocking: Return immediately. Assumes that data storage to be used for transfer is not modified by subsequent statements until transfer complete (Isend, Irecv) MPI Send and receive are blocking by default and synchrony for send

is implementation dependent (depends on existence of buffering, performance considerations etc. Deadlocks can occur in MPI usually not noticable when sending small messages.

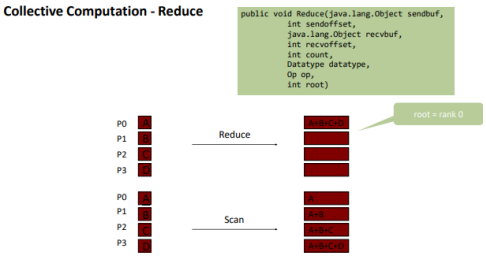
Sources of deadlocks: Sending a large messages between processes. If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a recieve) which creates a cycle in the program order hence a deadlock. Possible Solutions:

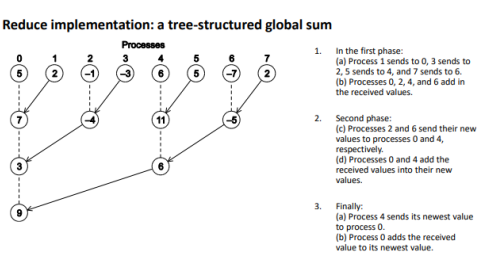


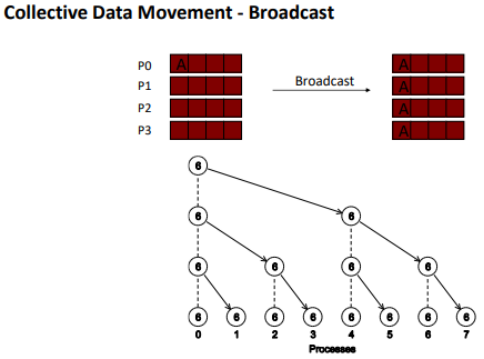


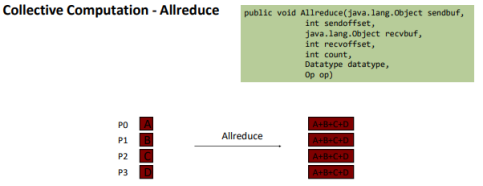


Group Communication: MPI supports communications among groups of processors (e.g sum all values in a group) e.g broadcast,gather,scatter,reduce,barrier,...



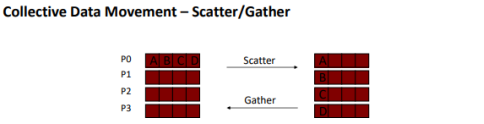




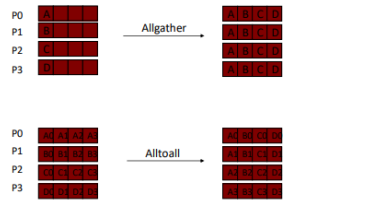


Useful in a situation in which all of the processes need the result of a global sum in order to complete some larger computation.

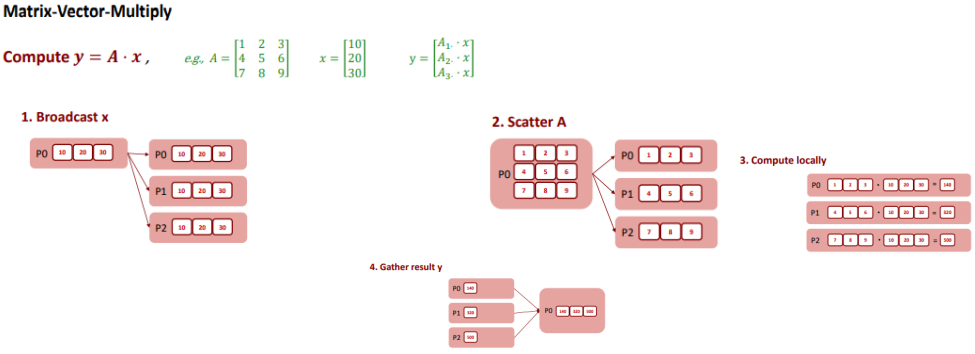
More Collective Data Movement – some more (16 functions total!)



- Scatter can be used in a function that reads in an entire vector on process 0 but only sends the needed components to each of the other processes.
- Gather collects all of the components of the vector onto destination process, then destination process can process all of the components.

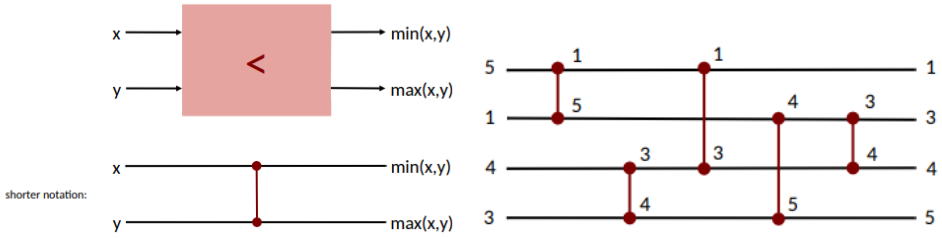


Example: Matrix-Vector-Multiply



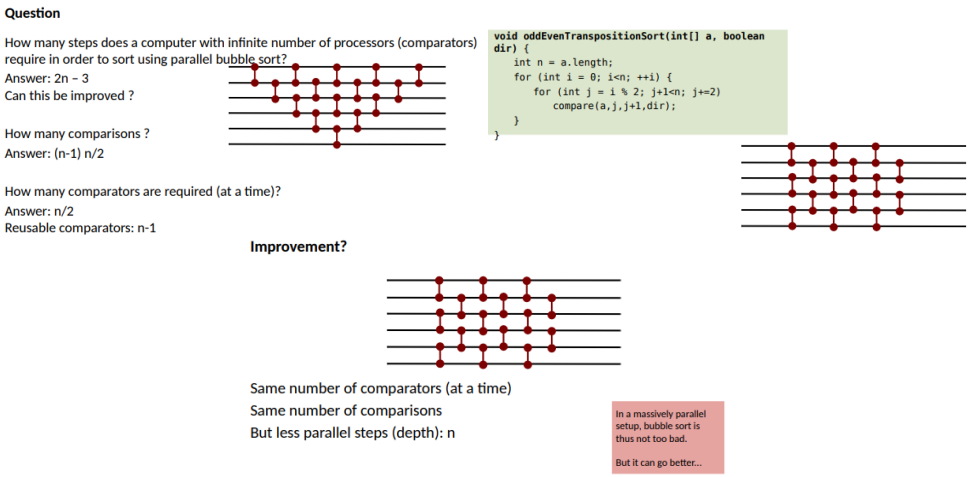
0.21 Parallel Sorting

Sorting Networks: Have one base operation the Comparator. It takes inputs x and y and outputs the min and max of both: Sorting Networks are **Data-Oblivious** i.e they perform the same operations regardless of the input. Data-oblivious

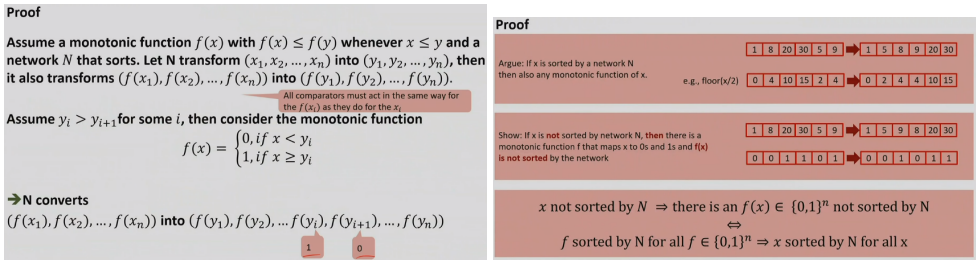


dont have a worst-case because worst-case = best-case = average case.

With parallelism insertion sort = bubble sort = selection sort. We try to improve parallel Bubble sort with odd-even Transposition Sort but get less parallel steps.



Zero-one-principle: If a network with n input lines sorts all 2^n sequences of 0s and 1s into non-decreasing order,it will sort any arbitrary sequence of n numbers in non-decreasing order.

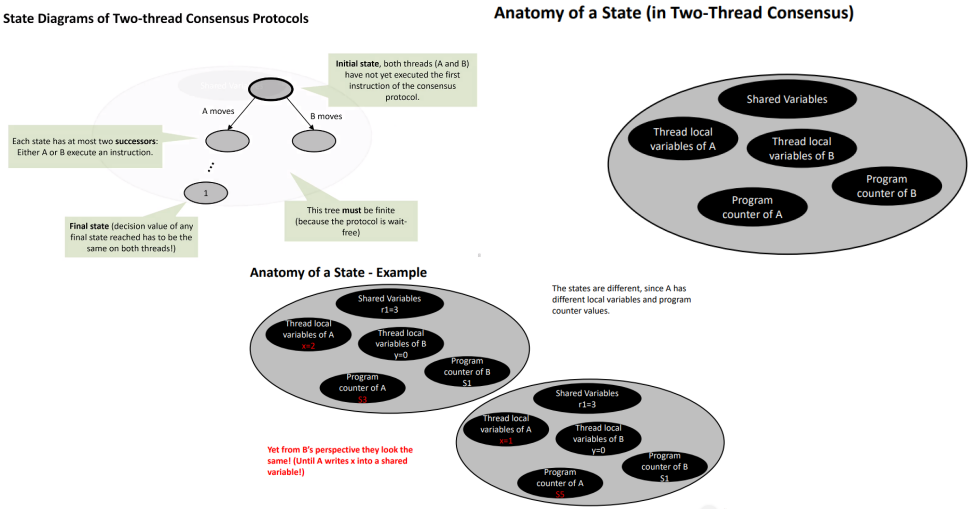


0.22 Wait-Free Consensus & Parallel Algorithms Primer

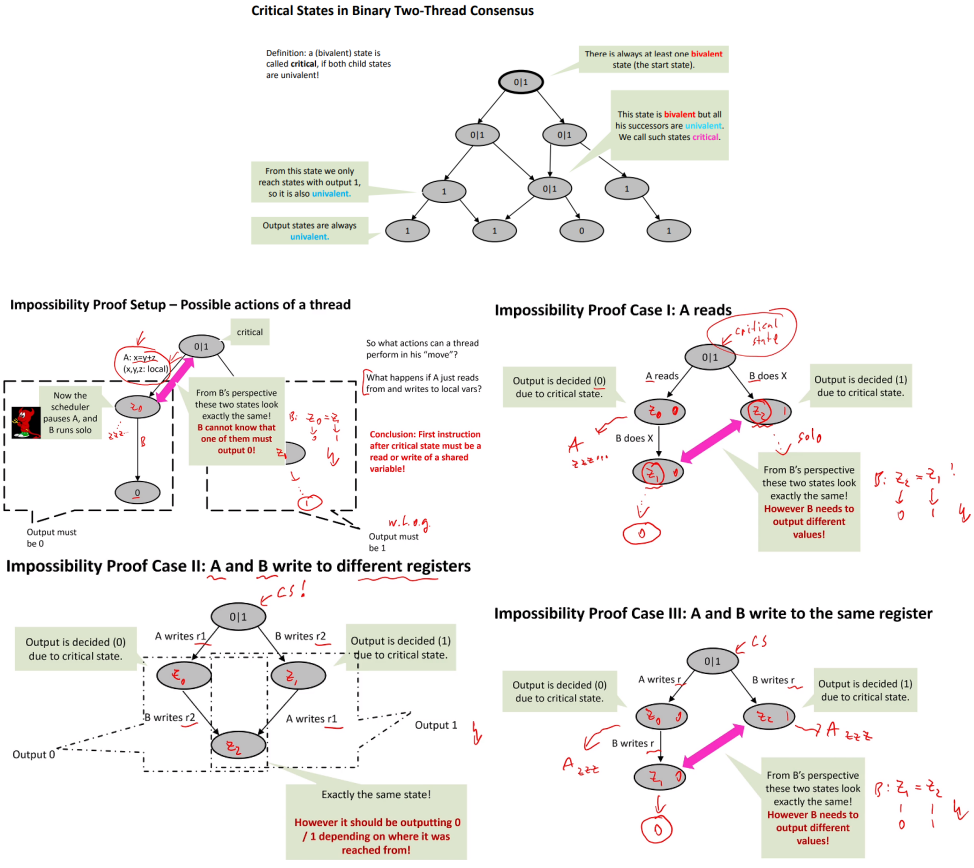
Wait-Free Consensus Protocols Each thread has a number (propose(i)). The decision must be a valid input and it must be consistent i.e all threads agree on the same number. We simplify the consensus by only considering two threads. The consensus needs to be wait-free i.e All threads finish after a finite number of steps, independent of other threads
Binary Consensus: Instead of proposing an integer, every thread now proposes either 0 or 1. This is still Equivalent to the normal consensus for two threads.

Concepty of Valency At some point during the execution (i.e a state) each thread will decide what to return. A state where a thread has decided on one is called **1-valent**, and a state where a thread has decided on zero **0-valent**. Undecided states are called bivalent, decided states are called univalent. Es gilt folgende Lemma:

- **Lemma 1:The initial state is bivalent** Proof: Consider initial state with A has input 0 and B has input 1. If A finished before B starts, we must decide 0 and if B finishes before A starts we must decide 1 (because it only knows the thread's input) Hence the initial state must be bivalent.
- **Lemma 2: Every consensus protocol has a critical state** Proof: From (bivalent) start state, let the threads only move to other bivalent states. If it runs forever the protocol is not wait free. If it reaches a position where no moves are possible this state is critical



Critical: A bivalent state for which both child states are univalent $\Rightarrow$ Binary consensus is not possible



Primer for Parallel Algorithms: Recall:

- Work W: number of operations performed when executing the algorithm (= sequential running time for P= 1)
- Depth D: Minimal number of operations for any parallel execution (= parallelo running time for p $=\infty$). Depth is also the longest path in the computational DAG

For a good parallel algorithm the Depth is usually much smaller than the Work