Parallel Programming

Contents

Parallelism

- **Parallelism**: Use extra resources to solve a problem faster
- **Concurrency**: Correctly and efficiently manage access to shared resources
- **Implicit Parallelism**: Compiler, runtime or operating system identify and exploit parallelism
- **Explicit Parallelism**: Programmer has the job to identify and exploit parallelism
- **Von Neumann Architecture**: Program data and program instructions in the same memory

Sequential vs. Concurrent

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"

Parallel Execution

Vectorization

- **SIMD**: A single instruction gets applied multiple times on different data

Instruction Level Parallelism

- Multiple units for one instruction stream
- **Pipelining**: Instructions are executed in many steps
- **Superscalar CPUs**: Multiple instructions per cycle and multiple functional units
- **Out-of-Order execution**: Potentially change execution order of instructions
- **Speculative execution**: Predict results to continue execution

Pipelining

- **Throughput**: Amount of work that can be done by a system in a given period of time, measured in the number of instructions completed per second
- **Latency**: Time to perform a single computation
- Throughput optimization may increase the latency

CPU Pipelining

- CPU gets divided into multiple stages that form functional units

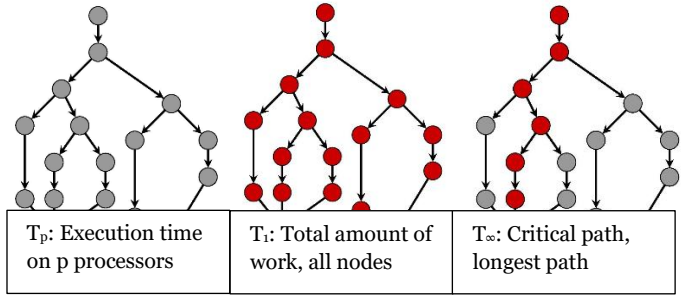
Expressing Parallelism

- **Work Partitioning**: Split up work of a single program into parallel tasks, can be done explicitly (manually by the programmer) or implicitly (automated by the system)
- **Scheduling**: Assign tasks to processors so that the processing power can be fully utilized, typically done by the system
- **Fine Granularity** is more portable, better for scheduling, but overhead can dominate in contrast to coarse granularity
- **Coarse Granularity**: granularity should be as small as possible but significantly bigger than scheduling overhead

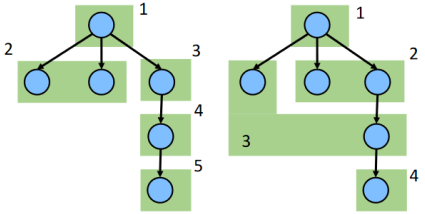
Parallel Performance

- Execution time on p CPUs is T_p, sequential execution time is T_1
- Parallel speedup S_p on p CPUs is $S_p = T_1 / T_p$
- Efficiency: S_p / p

```
Tₚ = T₁ / p, Sₚ = p (perfection)
Tₚ > T₁ / p, Sₚ < p (performance loss)
Tₚ < T₁ / p, Sₚ > p (performance gain)
```



T_p: Execution time on p processors	T_1: Total amount of work, all nodes	T_∞: Critical path, longest path



Example for T_2: **Time depends on scheduling of the tasks**

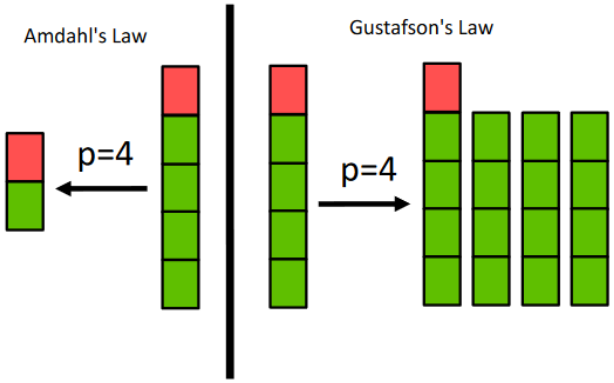
But: **T_1 and T_∞ are fixed**

- Performance loss ($S_p < p$) may be caused by programs containing not enough parallelism, overheads introduced by parallelization and architectural limitations
- **Relative speedup**: Relative improvement from using P execution units, baseline is the serialization of the parallel algorithm
- **Absolute speedup**: Using a better serial algorithm that does not parallelize
- **Amdahl's Law**: "The effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude"
    - Concerns maximum speedup
    - Puts a limit on scalability
    - Even a small decrease in the percent of things executed sequentially may pay off in large performance gains

$T_1 = W_{ser} + W_{par}$
$T_p >= W_{ser} + W_{par} / p$
$S_p <= T_1 / T_p$
$S_p <= 1 / (f + (1 - f) / p)$, $T_1 = W_{ser} + W_{par}$, $W_{ser} = f * T_1$, $W_{par} = (1 - f) * T_1$, *f is the non-parallelizable serial fractions of the total work*
$S_\infty <= 1 / f$

- **Gustafson's Law**: Runtime, not problem size is constant, more processors allows to solve larger problems in the same time, parallel part of a problem scales with the problem size

$W = p * (1 - f)\ T_{wAll} + f * T_{wAll}$
$Sp = f + p * (1 - f) = p - f * (p - 1)$



Work Horses of Parallel Programming

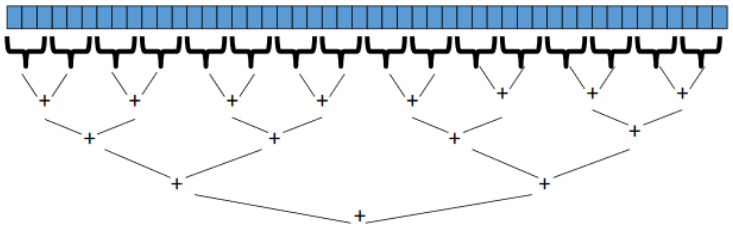
- **Reductions**: Produces a single answer from a collection via an associative operator, some things are inherently sequential
- **Maps**: Operates on each element of a collection independently to create a new collection of the same size, no combining of results

Adding Numbers

Step 1: Sequential Version

```
public static int sum(int[] input) {
        int sum= 0;
        for(int i=0; i < input.length; i++) {
                sum += input[i];
        }
        return sum;
}
```

Step 2: Parallel Version

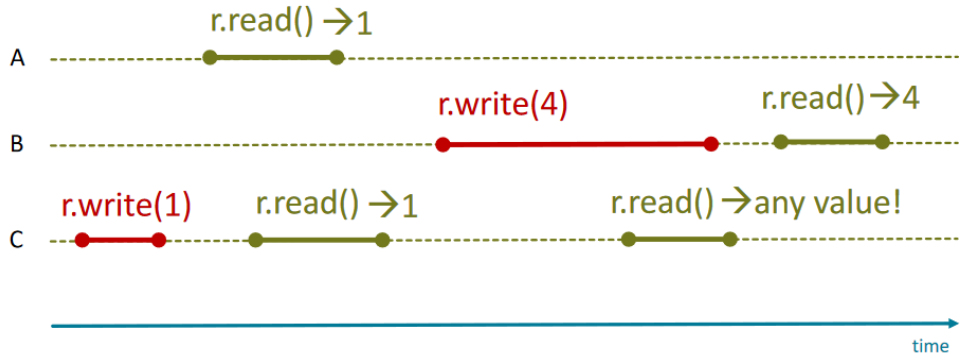


```
class SumForkJoin extends RecursiveTask <Long> {
        int low;
        int high;
        int[] array;

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

        @Override
        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;
                }
        }
```

```
}

// start process with the following method
static long sumArray(int[] array) {
        ForkJoinPool fjPool= new ForkJoinPool();
        return fjPool.invoke(new SumForkJoin(array,0,array.length));
}
```

Safe and Regular Registers

- Register: Basic memory object, can be shared or not, operations are *read* and *write*
- It is possible to construct mutual exclusion with non-atomic register
- **Safe Register** (The term *save* can be misleading)
    - Any read not concurrent with a write returns the current value of the register
    - Any read concurrent with a write can return any value of the domain of the register
    - Single Writer Multiple Reader Register: Only one concurrent write but multiple concurrent reads allowed
- **Regular Register**
    - Any read concurrent with a write can only return either the previous or the new value



Critical Sections

Pieces of code with the following conditions

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

Managing State

- **Immutability**: Data does not change, should be used whenever possible
- **Isolated Mutability**: Data can change, but only one thread can access it
- **Mutable/Shared Data**: Data can change, all threads can potentially access it

Mutable/Shared Data

- Concurrent accesses may lead to inconsistencies
- Solution: Protect state by allowing only one thread to access the critical section at a time (exclusive access)
- Mutable Data in Java: See Memory Sharing between Threads
    - Locks: Mechanism to ensure exclusive access/atomicity
    - **Transactional Memory**: Programmer describes a set of actions that need to be atomic

Alternatives to Shared Data

- **Functional Programming**: State is immutable, no synchronization required
- **Message Passing**: State is mutable, but not shared, each thread has its private state and threads cooperate via message passing
    - **Synchronous**: Sender blocks until message is received
    - **Asynchronous**: Sender does not block, placed into a buffer for receiver to get (fire-and-forget)
    - **Actor Model**: Actor sends messages to other actors
    - **Event-Driven Programming Model**: Actors react to messages, program is written as a set of handlers for events
    - **Blocking**: Return after local actions are complete, even if the message transfer may not have been completed
    - **Non-Blocking**: Return immediately

Actor Examples

- **Distributor**: Forwards received messages to a set of nodes in a round-robin fashion
- **Serializer**: If it receives an item that is larger than the last item plus one, add it to the sorted list, else if it receives an item that is equal to the last item plus one, send the sorted list and the old last item and reset the last item to the last itemsent

Communicating Sequential Processes

- Symbolic channels between sender and receiver
- Read and write requires a synchronous rendezvous
- *send*: Specifies a port to which a message is sent
- *receive*: Specifies a port and waits for the first message that arrives

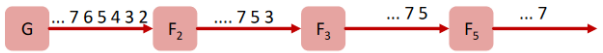
Concurrent Prime Sieve

- Algorithm to find prime numbers
- Each station removes multiples of the first element received and passes on the remaining elements to the next station

```
func Generate(ch chan<- int) {
    for i := 2; ; i++ {
        ch <- i
    }
}
```
G

```
func Filter(in <-chan int, out chan<- int, prime int) {
    for {
        i := <-in  // Receive value from 'in'.
        if i%prime != 0 {
            out <- i // Send 'i' to 'out'.
        }
    }
}
```
F_{prime}

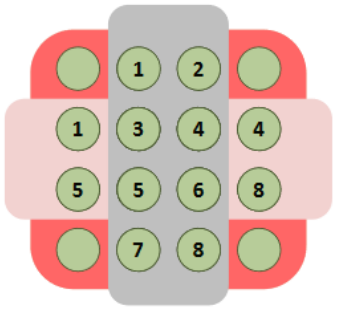
```
func main() {
    ch := make(chan int)
    go Generate(ch)
    for i := 0; i < 10; i++ {
        prime := <-ch
        fmt.Println(prime)
        ch1 := make(chan int)
        go Filter(ch, ch1, prime)
        ch = ch1
    }
}
```



Message Passing Interface

- The de-facto interface for distributed parallel computing
- MPI processes can be collected into groups, each group can have multiple colors (contexts), the name for the group consists of the group and the color
- When a MPI application starts, the group of all processes is given a predefined name (MPI_COMM_WORLD)
- A process is identified by an unique number within each communicator, called rank
- MPI communicators define a set of processes that are allowed to communicate with each other
- Initially all processes are in the communicator MPI_COMM_WORLD
- The rank of processes are associated with a communicator, numbered from 0 to n-1

Communicators do not need to contain all processes in the system



When you start an MPI program, there is one predefined communicator `MPI_COMM_WORLD`

Every process in a communicator has an ID called as "rank"

Can make copies of this communicator (same group of processes, but different "aliases")

The same process might have different ranks in different communicators

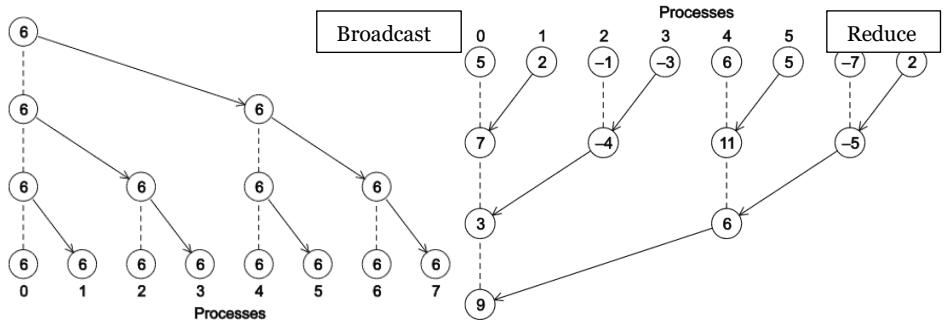
Communicators can be created "by hand" or using tools

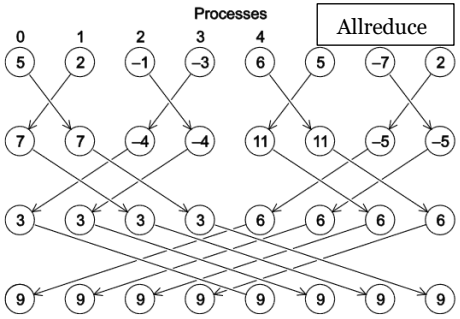
Simple programs typically only use the predefined communicator `MPI_COMM_WORLD`

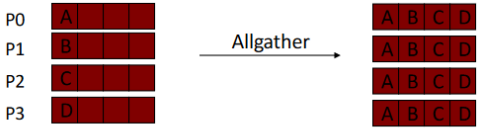
(which is sometimes considered bad practice)

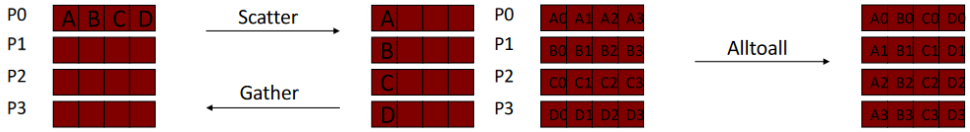
```
MPI.Init(args);
// declare and initialize variables
int size = MPI.COMM_WORLD.Size();
int rank = MPI.COMM_WORLD.Rank();
for(inti=rank; i<numSteps; i=i+size) {
        double x = (i+ 0.5) * h;
        sum += 4.0/(1.0 + x*x);
}
if(rank != 0) {
        double[] sendBuf= new double[]{sum};
        // 1-element array containing sum
        MPI.COMM_WORLD.Send(sendBuf, 0, 1, MPI.DOUBLE, 0, 10);
} else {
        // rank == 0
        double[] recvBuf = new double[1];
        for(intsrc=1 ; src<P; src++) {
                MPI.COMM_WORLD.Recv(recvBuf, 0, 1, MPI.DOUBLE, src, 10);
                sum += recvBuf[0];
        }
}
doublepi = h * sum;
// output pi at rank 0 only!
MPI.Finalize();
```

Collective Communication









- Example Matrix-Vector-Multiply: Broadcast vector to all threads, scatter matrix to all threads, compute locally in each thread, finally gather result

Safety/Liveness Properties

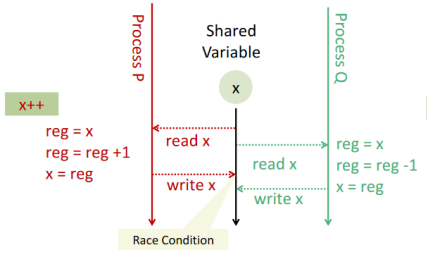
- **Safety: Nothing bad ever happens**
- **Liveness: Something good will happen in finite time**

- To ensure the parallel program satisfies the safety properties, we need to correctly synchronize the interaction of parallel threads.
- Every memory location must obey at least one of the following statements
  - **Thread-local**: Do not use the location in more than one thread, use whenever possible
  - **Immutable**: Do not write to the memory location, use whenever possible
  - **Synchronized**: Use synchronization to control access to the location

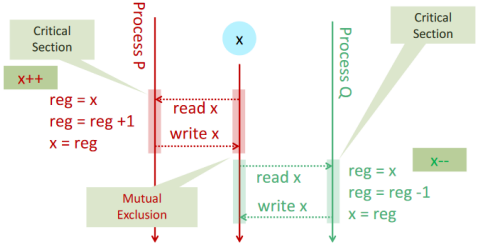
Absence of Data Races

Race Conditions

- A race condition occurs when the computation result depends on the scheduling (how threads are interleaved)
- Typically, problem is some intermediate state that messes up a concurrent thread that sees this state
- There is a difference between data races and bad interleavings



Race Condition



With Mutual Exclusion

Data Races

- Caused by insufficiently synchronized accesses of a shared resource by multiple threads, for example simultaneous read/write or write/write of the same memory location

Bad Interleavings

- If a second call starts before the first ends, we say the calls interleave
- Bad interleavings are caused by an unfavourable execution order of a multithreaded algorithm that makes use of otherwise well synchronized resources
- Can happen even with one processor since a thread can be interrupted at any point
- Consistent locking still allows bad interleavings
- Solution: **Locks**

```
public class Stack <E> {
        E peek() {
                E ans = pop();
                // inconsistent intermediate state,
```

```
                // stack property can be violatedF
                push(ans);
                return ans;
        }
}
```

Memory Reordering and Optimizing

- Compiler and hardware are allowed to make changes that do not affect the semantics of a sequentially executed program
- We can check the correctness by checking all interleavings or with a proof by contradiction
- Compilers may optimize certain code elements in such a way that they do not run in parallel anymore
- The keyword *Volatile* and *Synchronized Blocks* can be used to prevent reordering

```
int x;
void wait() {
        x = 1;
        while(x == 1); // this while loop gets optimized away
}
void arrive() {
        x = 2;
}
```

Mutual Exclusion

Requirements

- **Safety Property**: Only one thread in critical block at the same time, the other threads must wait
- **Liveness Property**: No Lockout when the critical block is free, the *acquire* process must terminate in finite time when no process executes in the critical section

Problems

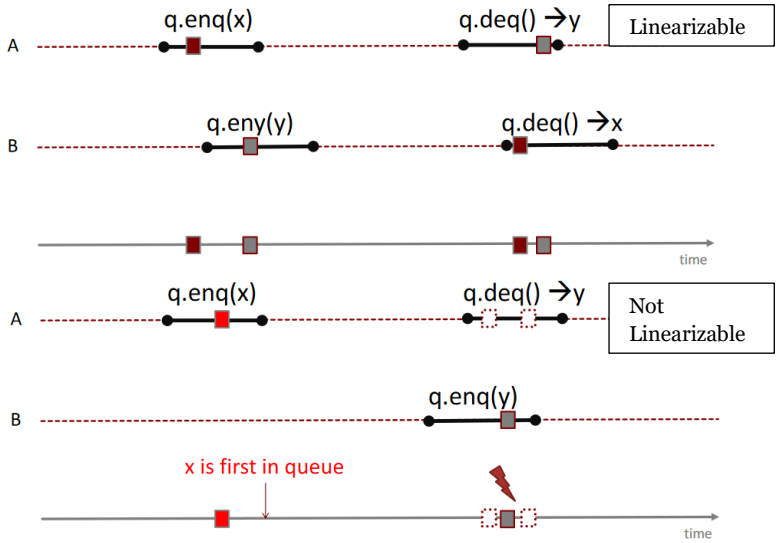
- **Deadlock**: group of two or more competing processes are mutually blocked because each process waits for another blocked process in the group to proceed
- **Livelock**: Competing processes are able to detect a potential deadlock but make no observable progress while trying to resolve it
- **Starvation**: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

Methods

- Locks

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



More Formal

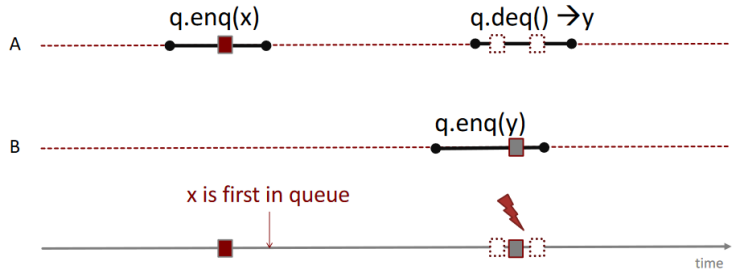
- Split method calls into two events: Incovation (A q.enq(x)) and response (A q: void)
- **History**: Sequence of invocations and responses
- Invocations and responses **match** if thread and object names agree
- An invocation is **pending** if it has no matching response
- A subhistory is **complete** when it has no pending responses
- **Object Projections**: Invocations and responses of an object
- **Thread Projections**: Invocations and responses of a thread
- **Sequential History**: Method calls of different threads do not interleave, a final pending invocation is okay
- **Well Formed History**: Per thread projections are sequential
- **Equivalent Histories**: Per thread projections of each history are equal
- **Legal History**: If the single-threaded, single object history is legal, meaning every object projection adheres to the sequential specification of the object
- A method call **precedes** another method call if the response event precedes the invocation event, otherwise the method calls **overlap**

- **A history is linearizable if it can be extended to a new history by appending zero or more responses to pending invocations that took effect and discarding zero or more pending invocations that did not take effect, such that the new history is equivalent to a legal sequential history and the order of the new history is a subset of the order of the legal sequential history, meaning an order across threads is required**
- **Composability Theorem**: A history is linearizable if and only if all object projections are linearizable, meaning linearizability of objects can be proven in isolation and independently implemented objects can be composed
- Atomic registers are linearizable with a single linearization point, meaning it is sequentially consistent and for non-overlapping operations, the realtime order is respected

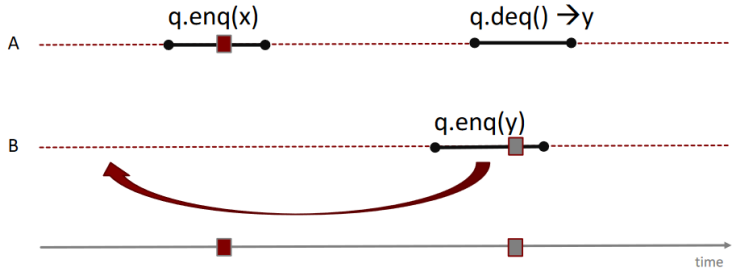
Sequential Consistency

- Alternative to linearizability
- **A history is sequentially consistent if it can be extended to a new history by appending zero or more responses to pending invocations that took effect and discarding zero or more pending invocations that did not take effect, such that the new history is equivalent to a legal sequential history, no order across threads is required**

Not linearizable

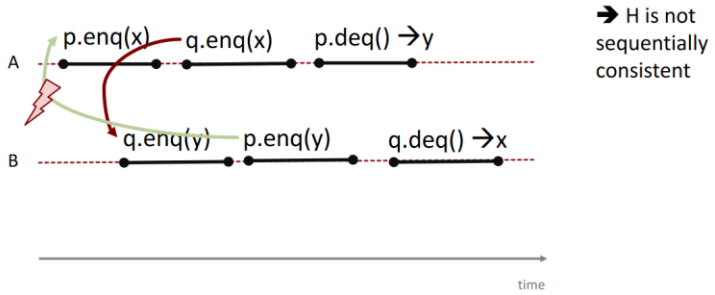


Yet sequentially consistent!



- Sequential consistency is not a local property, and thus we lose composability (In the following example: H|q and H|p are sequentially consistent, but H is not)

Ordering imposed by H|q and H|p



Quiescent Consistency

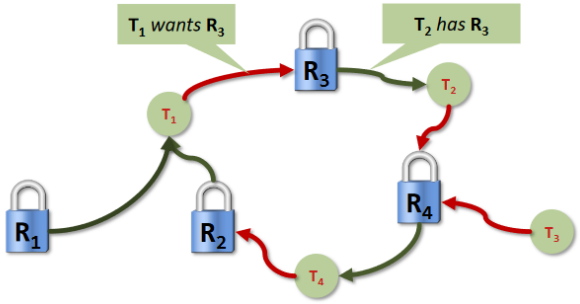
- Programs should respect real-time order of algorithms separated by periods of quiescence
- Quiescent consistency requires non-overlapping methods to take effect in their real-time order

Atomicity

- An operation is atomic if no other thread can see it partly executed, it appears indivisible

Absence of Deadlocks

- **Deadlocks**: two or more processes are mutually blocked because each process waits for another of these processes to proceed
- A deadlock for threads occurs when the directed graph describing the relation of threads and resources contains a cycle



- Deadlocks can be detected by finding circles in the dependency graph
- Deadlocks can be avoided by
  - Two-phase locking with retry, release when failed
  - Non-overlapping smaller critical sections
  - One lock for all resources
  - Resource ordering with a globally unique order

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

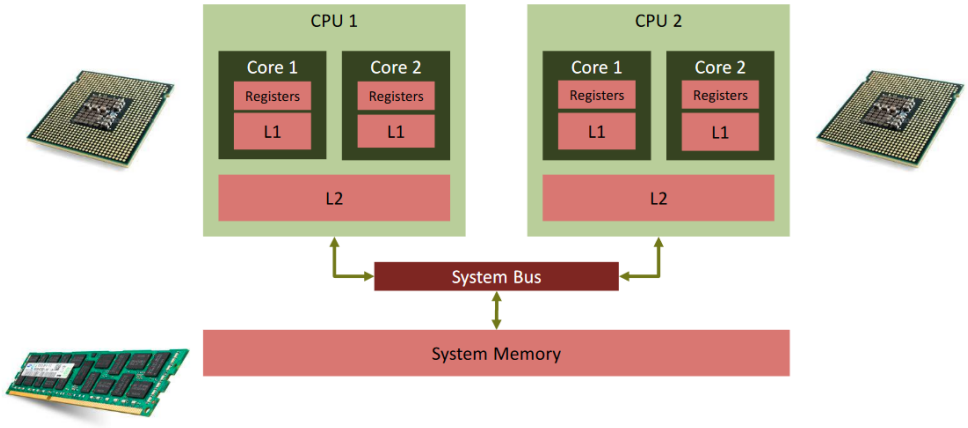
Unique global ordering required.
Whole program has to obey this order to avoid cycles.
Code taking only one lock can ignore it.

Custom Invariants

- Identify invariants in the problem domain, ensure they hold for your implementation

Hardware Support for Parallelism

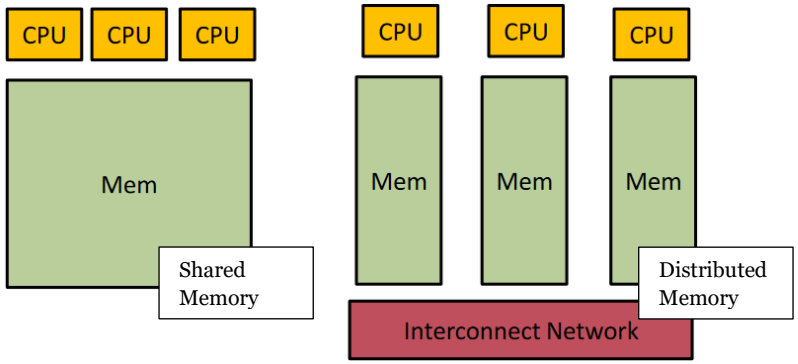
Memory Hierarchy



- A memory model of a programming language like Java provides often minimal guarantees for the effects of memory operations
  - Leaving open optimization possibilities for hardware and compiler
  - Including guidelines for writing correct multithreaded programs

Parallel Architectures

- Multicore processors: Programmers need to write parallel programs
- Shared memory systems can be programmed as distributed memory systems and the other way around



Shared Memory Architectures

- **Simultaneous Multithreading (SMT, Hyperthreading)**: Multiple units for multiple instruction streams, virtual cores, limited parallel performance but increases pipelining effects
- **Multicores**: Single chip with multiple cores, computations in parallel perform well, might share part of the cache hierarchy
- **Symmetric Multiprocessor System (SMP)**: Multiple CPUs on the same system, CPUs share memory, same cost to access memory
- **Non-Uniform Memory Access (NUMA)**: Memory is distributed, faster and slower memory, shared memory interface
- There is still a need for exchanging messages to ensure cache coherency
- Shared memory locations come in different variants
    - **Multi-Reader-Single-Writer** (flag)
    - **Multi-Reader-Multi-Writer** (victim)

Distributed Memory Architectures

- Depend on message passing

Hardware Support for Atomic Operations

- **Test-And-Set**
- **Compare-And-Swap**
- **Load Linked / Store Conditional**
- TAS and CAS are Read-Modify-Write operations, needed for lock-free programming

```
boolean TAS(memref s)
atomic
    if (mem[s] == 0) {
        mem[s] = 1;
        return true;
    } else
        return false;
```

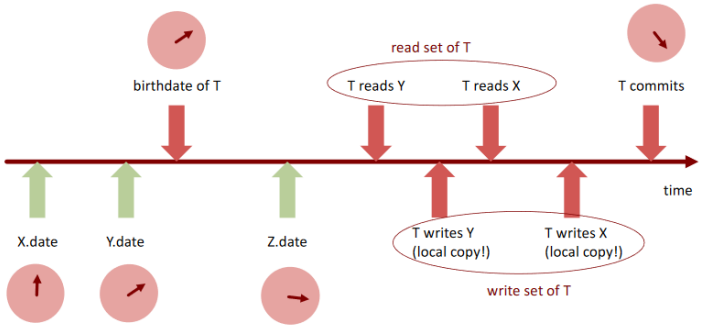
```
int CAS (memref a, int old, int new)
atomic
    oldval = mem[a];
    if (old == oldval)
        mem[a] = new;
    return oldval;
```

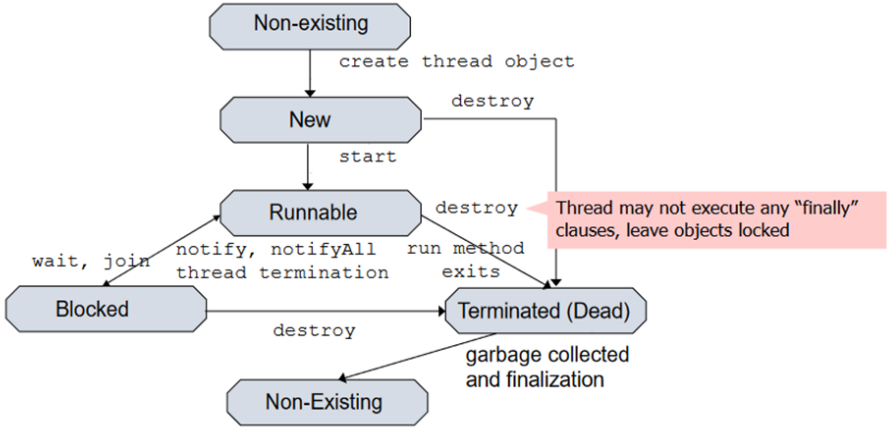
Transactional Memory

- Remove burden of synchronization from the programmer and place it in the system
- Solution: Atomic blocks (transactions) that allow multiple operations to happen atomically
- Programmer explicitly defines atomic code sections
- Benefits: Simpler, higher level semantics, composable, optimistic by design
- 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
- Transactions run in isolation, effects from other transactions are not observed
- Transactional memory is heavily inspired by database transactions (ACID properties: Atomicity, Consistency, Isolation, Durability)
- In case of a conflict, transaction can be aborted
- Hardware transactional memory can be fast, has bounded resources, can often not handle big transactions
- Software transactional memory has greater flexibility, achieving good performance is more challenging
- Nested transactions
    - Flattened Nesting: Inner aborts leads to outer aborts, inner commits are visible only if the outer transactions commit
    - Closed Nesting: Inner abort does not result in abort of outer transaction, inner transaction commits change visible to outer transaction, but not to other transactions until outer transaction commits
- What is part of a transaction
    - All program variables are protected: Easier to port existing code, but there is a need to check every memory operation
    - Reference-Based STM: Mutable state is put into special variables which only can be modified inside a transaction, everything else is immutable or not shared

Clock-Based STM System

- Each transaction uses a local read-set and a local write-set holding all locally read and written objects
- Transaction calls *read*
    - Check if the object is in the write set and return this new version
    - Otherwise check if the object's time stamp is smaller than the transaction's birthdate and add a new copy of the object to the read set, otherwise abort
- Transaction calls *write*
    - If the object is not in the write set, create a copy of it in the write set
- Transaction commits
    - Lock all objects of read-set and write-set in a defined order to avoid deadlocks
    - Check that all objects in the read set provide a time stamp that is smaller than the birthdate of the transaction, otherwise abort
    - Increment and get the value of the current global clock
    - Copy each element of the write set back to the global memory with the new timestamp
    - Release all locks and commit





Implementation of Parallelism

Threads

- Threads are not shielded from each other
- Threads share resources and can communicate more easily
- Instructions of a thread are executed in the specified order

Java Thread Implementation

- Every Java program has at least one execution thread, which calls the *main* method
- Each call to the start method of a thread object creates an execution thread
- Program ends when all non-daemon threads finish
- Daemon threads can continue to run even if the *main* method returns
- Threads can wait for another thread by calling the *join* method on the thread to wait for
- The *isAlive* method allows a thread to determine if the target thread is terminated
- Threads can be interrupted with the *interrupt* method
- It can be checked if a thread is interrupted with the *isInterrupted* method

java.lang.Thread

```
public class MyThread extends Thread {
        @Override
        public void run() {
                // statements
        }
}
MyThread myThread = new MyThread();
myThread.start();
```

java.lang.Runnable

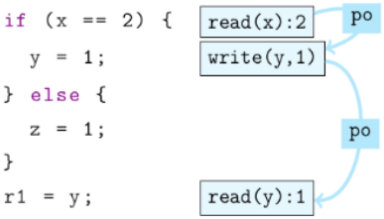
```
public class RunnableImplementation implements Runnable {
        @Override
        public void run() {
                // statements
        }
}
Thread myThread = new Thread(RunnableImplementation);
myThread.start();
```

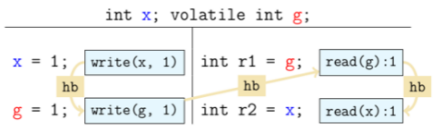
Memory Sharing between Threads

- Threads may assign and read the same variable
- The programmer is responsible for avoiding interleaving issues by explicit synchronization
- Threads can be synchronized via synchronization primitives
- All Java objects have an internal lock
- While an object is locked, no other thread can successfully lock the object
- Generally, the access of shared memory is done under a lock
- Many synchronization objects are already implemented in *java.util.concurrent*

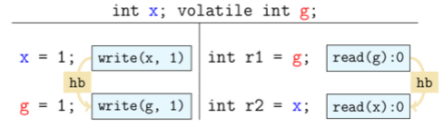
Java Memory Model

- Restricts allowable outcomes of programs
- Contract between programmer and compiler
- The JMM Defines Actions
- Program statements are not a total order across threads
- Executions combine actions with ordering
    - **Program Order**: Total order of intro-thread actions, per thread, the program order is consistent, does not provide ordering guarantee for memory accesses
    - **Synchronization Order**: Formed with synchronization actions (read/write volatile variables, lock/unlock monitor, first/last action of thread, actions which start a thread, actions which determine if a thread has terminated), is a total order, all threads see synchronization actions in the same order and in the program order within a thread, is consistent
    - **Synchronizes-With**: Pairs the specific actions to be visible to each other (volatile write to x synchronizes with subsequent read of x)
    - **Happens-Before**: Formed by transitive closure of *program order* and *synchronizes-with*, when reading a variable, we see either the last write in *happens-before* of and other unordered write
- **Memory operations will not be reordered with respect to accesses to volatile variables or synchronized blocks**

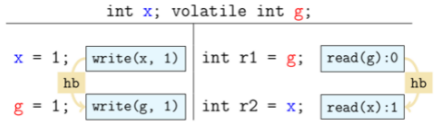




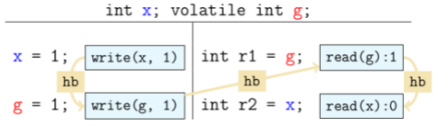
Case 1: HB consistent, observe the latest write in $\xrightarrow{hb}$
$(r1, r2) = (1, 1)$



Case 2: HB consistent, observe the default value
$(r1, r2) = (0, 0)$



Case 3: HB consistent (!), reading via race!
$(r1, r2) = (0, 1)$



Case 4: HB **in**consistent, execution can be thrown away

Locks

```
public interface Lock {
        public void lock();      // entering critical section
        public void unlock();    // leaving critical section
}
```

- Locks are reentrant: The same thread can access a critical section multiple times, the lock counts the number of accesses
- Basic synchronization primitive with operations
    - *new*: Make a new lock, initially not held
    - *acquire*: Blocks if this lock is already currently held, once not held it makes the lock held
    - *release*: Makes this lock not held, if more than one thread blocked on this lock, exactly one thread will acquire it
- The lock implementation ensures that given simultaneous acquires or releases, something correct will happen
- The lock implementation uses special hardware and OS support
- Lock Granularity
    - **Coarse-Grained**: All methods are synchronized (Fewer locks, more objects per lock, simpler to implement, less simultaneous access)
    - **Fine-Grained**: Split object into pieces with separate locks (More locks, fewer objects per lock, harder to implement, more simultaneous access)
    - **Optimistic Locking (Synchronization)**: Do not lock objects when traversing them, then lock objects and lastly check if everything is okay (Traversals are wait-free, less lock acquisitions, need to traverse list twice to validate)
    - **Lazy Locking (Synchronization)**: Mark removed objects instead of locking them when removing or validating in the end

- **Critical-Section Granularity**: Large critical sections cause performance loss, short critical sections are more difficult to implement and may cause performance loss because of frequent thread switching, make critical sections just long enough to preserve Atomicity
- **Spinlocks**: Computing resources wasted while actively waiting, no notification mechanism
- **Scheduled Locks**: Require support from the runtime system, higher wakeup latency
- Disadvantages of locking: Pessimistic by design, performance issues, blocking semantics, not composable

Lock Performance

- **Uncontended Case**: Threads do not compete for the lock, lock implementations have minimal overhead, typically just the cost of an atomic operation
- **Contended Case**: Threads compete for the lock, can lead to significant performance degradation and starvation

Condition Interface

- Java locks provide conditions that can be instantiated
- *await*: The current thread waits until condition is signalled
- *signal*: Wakes up one thread waiting on this condition
- *signalAll*: Wakes up all threads waiting on this condition

Lock-Free Programming

- **Lock-Freedom**: At least one thread always makes progress even if other threads run concurrently, this implies system-wide progress but not freedom from starvation
- **Wait-Freedom**: all threads eventually make progress, this implies lock-freedom and freedom from starvation
- **Blocking**: A thread can indefinitely delay another thread
- **Non-Blocking**: Failure or suspension of one thread cannot cause failure or suspension of another thread

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

Compare and Swap

- Compare *old* value with value at memory location
- If and only if value at memory location equals *old* value, overwrite memory location with *new* value
- CAS can be implemented wait-free by hardware
- Positive result of CAS suggests that no other thread has written, but this is not always true (ABA problem)

```
public class CasCounter{
        private AtomicIntegervalue;
        public int getVal() {
                return value.get();
        }
        public int inc() {
                int v;
                do {
                        v = value.get();
                } while (!value.compareAndSet(v, v+1));
                return v+1;
        }
}
```

Lock-Free Stack

```
public static classNode {
        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;
        }
}

public classBlockingStack {
        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;
        }
}

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

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

- A lock-free algorithm does not automatically provide better performance than its blocking equivalent
- Atomic operations are expensive and contention can still be a problem, backoff can be used to improve performance

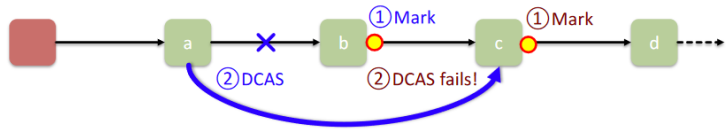
Update Reference and Mark Bit Consistently

- We want to atomically establish consistency of two things (mark bit and reference)
- We can use a variant of double compare and swap, this is possible because one bit is free in the reference

```
AtomicMarkableReference <V> {
        boolean attemptMark(V expectedReference, booleannewMark);
        boolean compareAndSet(V expectedReference, V newReference
        boolean expectedMark, boolean newMark);
        V get(boolean[] markHolder);
        V getReference();
        boolean isMarked();
        void set(V newReference, Boolean newMark);
}
```

A: remove(c)
B: remove(b)

1. try to set mark (c.next)
2. try CAS(
   [b.next.reference, b.next.marked],
   [c,**unmarked**], [d,unmarked]);

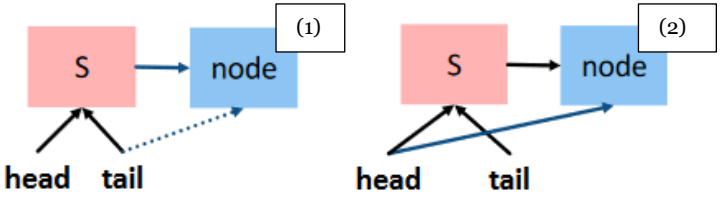


c remains marked ☹ (logically deleted)

1. try to set mark (b.next)
2. try CAS(
   [a.next.reference, a.next.marked],
   [b,unmarked], [c,unmarked]);

Update Two References Consistently

- Example: Queue
- Problem: First thread enqueues an element to an empty list by setting the next reference of the sentinel (1), but has not yet adapted the tail reference, then another thread dequeues the the sentinel, which results in the tail reference pointing to an invalid reference (2)



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 !

- By incrementing the tail pointer if the tail pointer has a next node, we help other threads to make progress

ABA Problem

- The ABA problem occurs when one activity fails to recognize that a single memory location was modified temporarily by another activity and therefore erroneously assumes that the overall state has not been changed
- Solutions
  - **Double Compare and Swap**: Not available on most platforms
  - **Garbage Collection**: Slow
  - **Pointer Tagging**: Bits available for tagging are used as a counter, this does not cure the problem, it only delays it
  - **Hazard Pointers**: Transactional memory is needed, before a thread reads a pointer, it marks it as hazardous by entering it in a slot of an array, when the CAS is finished, the thread removes the pointer from the array, before another thread tries to reuse the pointer, it checks all entries of the hazard array

```
public class NonBlockingStackPooledHazardGlobal extends Stack {
        AtomicReference <Node> top = newAtomicReference <Node> ();
        NodePoolHazard pool;
        AtomicReferenceArray <Node> hazarduous;
        public NonBlockingStackPooledHazardGlobal(int nThreads) {
                hazarduous = newAtomicReferenceArray <Node> (nThreads);
                pool = newNodePoolHazard(nThreads);
        }
        boolean isHazarduous(Node node) {
                for(int i = 0; i < hazarduous.length(); ++i)
                        if (hazarduous.get(i) == node)
                                return true;
                return false;
        }
        void setHazardous(Node node) {
                hazarduous.set(id, node);
        }
}
```

Algorithms to Provide Mutual Exclusion

Decker's Algorithm

volatile boolean wantp=false, wantq=false, integer turn= 1

```
Process P
loop
    non-critical section
    wantp = true
    while (wantq) {
        if (turn == 2) {
            wantp = false;
            while(turn!=1);
            wantp = true; }}
    critical section
    turn = 2
    wantp = false
```

only when q tries to get lock

and q has preference

let q proceed

and wait

and try again

```
Process Q
loop
    non-critical section
    wantq = true
    while (wantp) {
        if (turn == 1) {
            wantq = false
            while(turn != 2);
            wantq = true; }}
    critical section
    turn = 1
    wantq = false
```

Peterson Lock

let P=1, Q=2; volatile boolean array flag[1..2] = [false, false];
volatile integer victim = 1

```
Process P (1)
loop
    non-critical section
    flag[P] = true
    victim = P
    while(flag[Q] && victim == P);
    critical section
    flag[P] = false
```

I am interested

but you go first

We both are interested

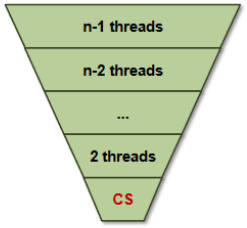
And you go first

```
Process Q (2)
loop
    non-critical section
    flag[Q] = true
    victim = Q
    while(flag[P] && victim == Q);
    critical section
    flag[Q] = false
```

The Filter Lock

- Extension of Peterson's Lock to n processes
- **Not fair**: The lock is not first-come first-serve

```
int[] level(#threads), int[] victim(#threads)


lock(me) {
  for (int i=1; i<n; ++i) {
    level[me] = i;
    victim[i] = me;
    while (∃k ≠ me: level[k] >= i && victim[i] == me) {};
  }
}

unlock(me) {
  level[me] = 0;
}
```



Other threads are at same or higher level

And I have to wait

Bakery Algorithm

- A process is required to take a numbered ticket with a value greater than all outstanding tickets, and wait until the ticket number is the lowest

```
integer array[0..n-1] label = [0,...,0]
boolean array[0..n-1] flag =  [false, ..., false]
```

SWMR «ticket number»

SWMR «I want the lock»

```
lock(me):
   flag[me] = true;
   label[me] = max(label[0], ... , label[n-1]) + 1;
   while (∃k ≠ me: flag[k] && (k,label[k]) <l (me,label[me])) {};

unlock(me):
   flag[me] = false;
```

$(k, l_k) <_l (j, l_j) \Leftrightarrow l_k < l_j \text{ or } (l_k = l_j \text{ and } k < j)$

Test-And-Set Lock

- Sequential bottleneck: Threads fight for the bus during call of *getAndSet*

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



```
}
```

Test-And-Test-And-Set Lock

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

Test-And-Test-And-Set Lock with Backoff

- Threads no longer fight for the bus during *getAndSet*, better performance

```
public class TATASBackoffLock implements Lock {
        AtomicBoolean state = new AtomicBoolean(false);
        public void lock() {
                Backoff backoff = null;
                while(true) {
                        while(state.get());
                        if(!state.getAndSet(true)) {
                                return;
                        } else {
                                try {
                                        if(backoff == null)
                                                backoff = new Backoff(MIN_DELAY,
MAX_DELAY);
                                } catch (InterruptedException e) {}
                        }
                }
        }
        public void unlock() {
                state.set(false);
        }
}
class Backoff {
        public void backoff() throws InterruptedException {
                int delay = random.nextInt(limit);
                if(limit < maxDelay) {
                        limit = 2 * limit;
                }
                Thread.sleep(delay);
        }
}
```

Read Write Lock

- Note that this lock is not 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();
        }
}
class Queue {
        int in=0, out=0, size
        n = 0; // n consumers are waiting
        m = 0; // m producers are waiting
        long buf[];
        final Lock lock= new ReentrantLock();
        final Condition notFull  = lock.newCondition();
        final Condition notEmpty = lock.newCondition();
        Queue(int s) {
                size = s;
                m = size – 1;
                buf = new long[size];
        }
        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;
        }
}
```

Synchronized Blocks

```
synchronized(object) {
        // statements
yy}
```

- Enforces mutual exclusion by locking the shared resource
- If an exception is thrown the lock on the object will be released and the exception will be thrown
- Methods inside synchronized blocks
    - *wait*: Releases the object lock, the threads waits on an internal queue
    - *notify*: Wakes the highest-priority thread closest to the front of the object's internal queue
    - *notifyAll*: Wakes up all waiting threads, they compete for access to the object

Synchronized Methods

```
public synchronized void myMethod() {
        // statements
}
```

- Enforces mutual exclusion by locking the object that contains the method

Volatile

- *volatile* makes changes visible immediately to all threads
- *volatile* variables do not get reordered

```
private volatile int x = 0;
```

```
private volatile int y = 0;
void f() {
        x = 1;
        y = 1;
}
void g() {
        int a = y;
        int b = x;
        assert(b >= a); // this would not work without volatile
}
```

Semaphores

- Allows a specific amount of threads to enter the critical section
- *acquire*: Wait until S is greater than zero, then decrement S
- *release*: Increment S

```
synchronized void enter() {
        while(number <= 0)
        try {
                wait();
        } catch (InterruptedException e) {};
        number--;
}
synchronized void exit() {
        number++;
        if(number> 0)
                notify();
}
```

Rendezvous with Semaphores

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

Semaphore Barrier

- Synchronize a number of processes

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

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
```

Monitors

- Abstract data structure equipped with a set of operations that run in mutual exclusion
- If a condition does not hold
    - Release the monitor lock
    - Wait for the condition to become true
    - Signalling mechanism to avoid busy-loops
- *wait*: The current thread waits until it is signalled via notify
- *notify*: Wakes up an arbitrary waiting thread
- *notifyAll*: Wakes up all waiting threads
- **Signal and Wait**: Signalling process exits monitor, goes to waiting queue and passes the monitor lock to signalled process
- **Signal and Continue**: Signalling process continues running, signalled process gets moved to waiting queue
- Guidelines for using condition waits
    - Always have a condition predicate
    - Always test the condition predicate before calling and returning from wait
    - Always call wait in a loop
    - Ensure the state is protected by a lock associated with the condition

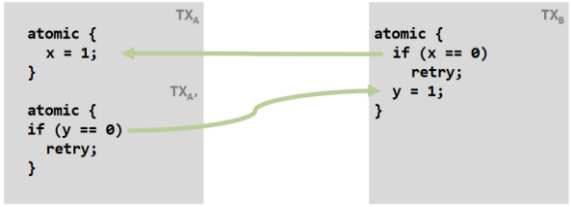
ScalaSTM

- Software Transactional Memory library built for Scala, has a Java interface
- Follows the reference-based approach

```
classAccountSTM {
        private finalInteger id;
        privatefinalRef.View<Integer> balance;
        AccountSTM(intid,intbalance) {
                this.id = newInteger(id);
                this.balance = STM.newRef(balance);
        }
        void withdraw(final int amount) {
```

```
                STM.atomic(new Runnable() { public void run() {
                        int old_val = balance.get();
                        balance.set(old_val-amount);
                }});
        }
        void deposit(finalintamount) {
                STM.atomic(new Runnable() { public void run() {
                        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;
        }
        static void transfer(finalAccountSTMa, finalAccountSTMb, finalintamount) {
                STM.atomic(new Runnable() { public void run() {
                        a.withdraw(amount);
                        b.deposit(amount);
                }});
        }
        static void transferRetry(finalAccountSTMa,finalAccountSTMb,finalintamount) {
                STM.atomic(new Runnable() { public void run() {
                        if(a.balance.get() < amount)
                                STM.retry();
                                a.withdraw(amount);
                                b.deposit(amount);
                }});
        }
}
```

- *STM.retry*: Transaction aborts and will be retried when any of the variables that read change
- Dependencies can lead to application level deadlock

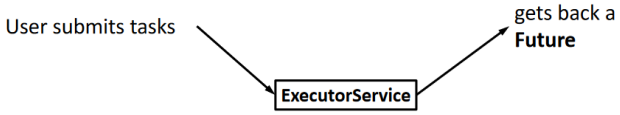


- Implementation: Threads that run transactions with the thread states (active, aborted, committed) and the objects representing variables affected by a transaction offering specific methods (constructor, read, write, copy)

Dining Philosophers

```
// main
Fork[] forks = newFork[tableSize];
for(int i = 0; i < tableSize; i++)
        forks[i] = newFork();
PhilosopherThread[] threads = newPhilosopherThread[tableSize];
for(int i = 0; i < tableSize; i++)
        threads[i] = newPhilosopherThread(forks[i], forks[(i+ 1) % tableSize]);

private static class Fork {
        public finalRef.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() {
                for(int m = 0; m < meals; m++) {
                        // think
                        pickUpBothForks();
                        // eat
                        putDownForks();
                }
        }
        private void pickUpBothForks() {
                STM.atomic(newRunnable() { publicvoid run() {
                        if(left.inUse.get() || right.inUse.get())
                                STM.retry();
                        left.inUse.set(true);
                        right.inUse.set(true);
                }});
        }
        private void putDownForks() {
                STM.atomic(newRunnable() { publicvoid run() {
                        left.inUse.set(false);
                        right.inUse.set(false);
                }});
        }
}
```

Executor Services



.submit(Callable<T> task) → Future<T>
.submit(Runnable task) → Future<?>

Don't subclass **Thread**	Do subclass **RecursiveTask<V>**
Don't override **run**	Do override **compute**
Do not use an **ans** field	Do return a **V** from **compute**
Don't call **start**	Do call **fork**
Don't just call **join**	Do call **join** which returns answer
Don't call **run** to hand-optimize	Do call **compute** to hand-optimize
Don't have a topmost call to **run**	Do create a pool and call **invoke**

Good Performance in Practice

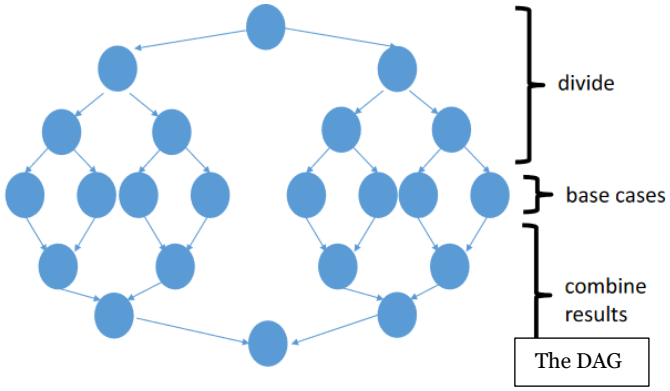
- Multiple operations in each task, documentation recommends 100-5000 basic operations
- Library needs to "warm up" so the Java Virtual Machine re-optimizes for the library

Analysing Parallel Algorithms

- **Work**: How long it would take one processor (T_1), sequentialize the recursive forking
- **Span**: How long it would take an infinite amount of processors (T_∞), the longest dependence chain

Fork-Join Framework

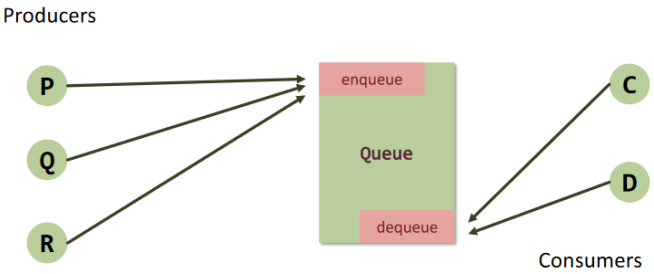
- A program execution using fork and join can be seen as a DAG
- Nodes: Pieces of work
- Edges: Source must finish before destination starts
- A fork "ends a node" and makes two outgoing edges
- A join "ends a node" and makes a node with two incoming edges



- **Work**: O(n) for simple maps and reductions
- **Span**: O(log(n)) for simple maps and reductions
- **Speed-Up**: T_1 / T_p for p processors
- **Parallelism**: Maximum possible speed-up T_1 / T_∞
- Perfect linear speed-up means doubling the amount of processors halves the running time
- At some point, adding processors won't help, this point depends on the span
- **Designing parallel algorithms is about decreasing span without increasing work too much**
- Asymptotically optimal execution would be $T_p = O((T_1 / p) + T_\infty)$
- The ForkJoin Framework gives an expected-time guarantee of asymptotically optimal
- But: Amdahl's Law implies that unparallelized parts become a bottleneck very quickly, but sometimes, things that seem sequential are actually parallelizable

Producer-Consumer

- A synchronized mechanism to pass the product is needed




```
synchronized void enqueue(T product) {
        while(isFull())
```

```
                try {
                        wait();
                } catch (InterruptedException e) {}
        doEnqueue(product);
        notifyAll();
}
synchronized T dequeue() {
        while(isEmpty())
                try {
                        wait();
                } catch (InterruptedException e) {}
        T product = doDequeue();
        notifyAll();
        return product;
}
```

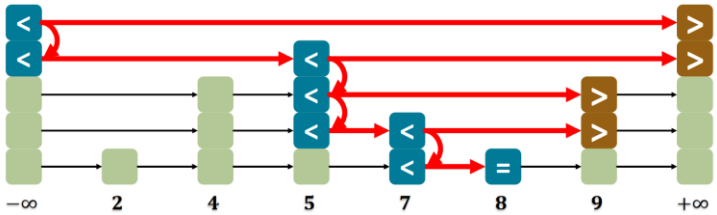
- The while loop is needed in case the consumer is notified without something to consume

Reader-Writer

- Multiple concurrent reads of same memory: Not a problem
- Multiple concurrent writes of same memory: Problem
- Multiple concurrent reads and writes of same memory: Problem
- **We could still allow multiple simultaneous readers**
- New lock with the following states
  - *Not held*
  - *Held for writing* by one thread
  - *Held for reading* by one or more threads
- Methods
  - *new*: Make a new lock, initially not held
  - *acquire_write*: Block if currently held for reading or held for writing, else make held for writing
  - *release_write*: Make not held
  - *acquire_read*: Block if currently held for writing, else make/keep held for reading and increment readers count
  - *release_read*: Decrement readers count, if readers count is zero, make not held

Skip Lists

- Collection of elements without duplicates
- Sorted multi-level list
- Node height is probabilistic, no rebalancing
- Higher level lists are always contained in lower-level lists, lowest level is entire list
- *add*: Find predecessors lock-free, lock predecessors, validate, splice, mark as fully linked, unlock
- *remove*: Find predecessors lock-free, lock victim, mark victim as removed, lock predecessors, validate, physically remove, unlock
- *contains*: Find element, check that it isn't marked as removed, check if it is marked as fully linked (wait-free)



Consensus Protocol

- A number of threads decide on a value where each thread brings an input value
- Requirements
  - Wait-Free: Returns in finite time for each thread
  - Consistent: All threads decide the same value
  - Valid: Common decision value is some thread's input
- Consensus Number: Larges n such that a consensus protocol solves an n-thread consensus
- **Theorem: Atomic registers have consensus number 1**
- **From this follows that there is no wait-free implementation of n-thread consensus with n larger than one from read-write registers**
- **Theorem: Compare-And-Swap has infinite consensus number**

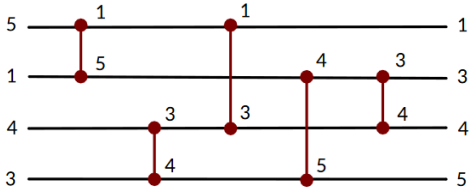
```
class CASConsensus {
        private final int FIRST = -1;
        private AtomicIntegerr = new AtomicInteger(FIRST);
        AtomicIntegerArrayproposed;
        // constructor
        public Object decide(Object value) {
                int i = ThreadID.get();
                proposed.set(i, value);
                if(r.compareAndSet(FIRST, i))
                        return proposed.get(i);
                else
                        return proposed.get(r.get());
        }
}
```

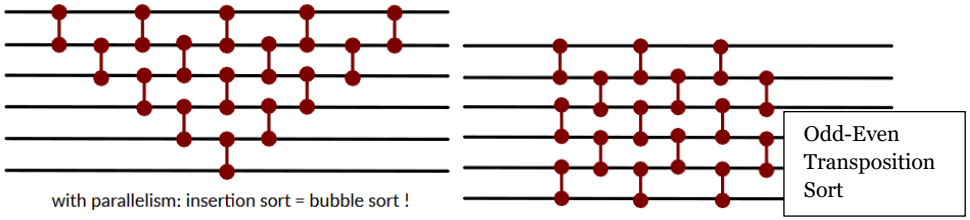
- Wait-free FIFO queues have consensus number 2
- Test-And-Set, Get-And-Set, Get-And-Increment have consensus number 2
- Compare-And-Set has consensus number ∞
- This means that wait-free FIFO queues, wait-free RMW operations and CAS cannot be implemented with atomic registers

Sorting

- Heapsort and Mergesort have O(n log n) worst-case run time, Quicksort has O(n log n) average-case run time
- Because of the tree structure of sorting, it cannot be better than O(n log n)

Sorting Networks





Odd-Even Transposition Sort

- Computer with an infinite number of processors needs to do 2n-3 steps to sort using parallel bubble sort or insertion sort
- Using Odd-Even Transposition Sort, this can further be reduced to n steps
- There exist even better parallel algorithms like Bitonic Sort that sort in $O(\log^2 n)$ parallel time
- A bitonic sequence is defined as a list with no more than one local minimum and no more than one local maximum
- A Half Cleaner divides a bitonic list in two equal halves that are itself bitonic lists
- A Bitonic Merger compares a pair and moves smaller numbers to the left and larger to the right, thus sorting a bitonic list
- A Merger takes two sorted lists and outputs a resulting sorted list