Mutual Exclusion

Introduction

Overview

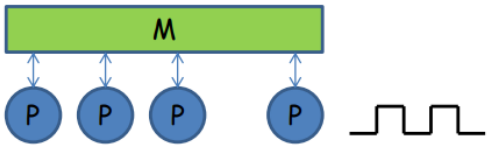
Slide 1-9

Parallel computing: efficiently using parallel resources to solve a given computational problem - focused on performance and efficiency

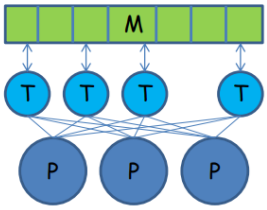
Concurrent computing: managing and reasoning about interacting processes that may take place simultaneously - focused on coordination and correctness

Abstractions

PRAM (Parallel Random Access Machine): assumes that all processes act under the same clock → good for studying parallel algorithms and establishing lower bounds on speedup based on the number of processors.



Symmetric Shared-Memory Computer: no shared clock for the processes → threads aren't synchronized and it might be difficult to reason about correctness and performance.



Most modern (parallel) processors are asynchronous.

Speedup, Performance and Throughput

Slide 10-20

Sequential time: time needed to solve a problem sequentially by best-known algorithm

Parallel time: time needed by slowest processor to solve problem using *p* processors

Absolute speedup: sequential time/parallel time

Relative speedup: parallel time with one processor/parallel time with *p* processors

Best possible speedup is O(p), perfect speedup is equal to p.

Throughput: number of operations that can be carried out in a given time

Latency: time it takes to carry out a given number of operations

Shared Memory

Slide 21-37

Multiple processors will communicate and coordinate through a shared memory. Processors (hardware entity) will execute processes that execute threads (software entities). Threads are uncoordinated, asynchronous, and can be interrupted, so it's impossible to make assumptions about when the action of another thread will take place.

Writes to the shared memory may be delayed or reordered, so threads might see values in a different order than they were written.

Caches

Small, fast memory spaces. Each processor typically has its own caches. The caches should communicate with each other to maintain cache coherence and "push" updates to other memory levels to maintain memory consistency.

Write buffers are small buffers that store pending writes

Sequential Consistency

Slide 39-56

Sequential consistency: the result of concurrent execution of a set of threads follows the thread's program order, i.e. updates to the memory are visible to other threads in the order of execution.

Typically this is not the case in modern shared-memory multiprocessors. In practice, we can make programs sequentially consistent enough by relying on atomic ordering instructions and memory fences (for this there's a balance between having too many or

too few fences - too few make the program incorrect and too many make the program slow).

Mutual Exclusion

Slide 57-66

In practice, we often use **locks** to protect the critical sections (CS) of our program. They must guarantee:

Mutual exclusion: at most one thread in the CS at once

Deadlock freedom: when multiple threads try to acquire the lock, one will always succeed

Starvation freedom: a thread trying to acquire the lock will eventually get it

→ critical sections are not concurrent

Starvation freedom implies deadlock freedom

Locks

Peterson Lock

Slide 67-79

Ensures mutual exclusion on 2 threads and based on:

- Each thread has its own flag to indicate that it wants to enter CS → check the other thread's flag before entering

  This does not satisfy deadlock freedom if both threads set their flags to true before both read the other's flag.

- Use a victim variable to indicate which thread should go first

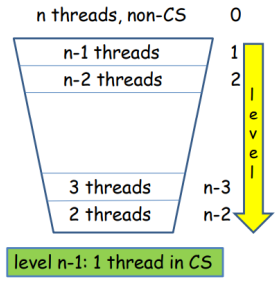
  This deadlocks if one thread starts and finishes completely before the other start. The lock depends on the presence and cooperation of the other thread.

A combination of the two ideas ensures mutual exclusion and starvation freedom (proof by contradiction). Deadlock freedom follows starvation freedom.

Filter Lock

Slide 80-90

Ensures mutual exclusion on n threads. There will be n-1 locking levels - at least one thread trying to enter a level will succeed, but one thread will be blocked if more than one thread tries to enter the same level:



- The Boolean flag[2] in the Peterson-lock is replaced by a level[n] array which indicates the level at which a thread is trying to enter.
- The victim variable is changed to an array that keeps track of the last thread entering a level j.

This satisfies mutual exclusion and starvation freedom (induction hypothesis + proof by contradiction), and therefore also deadlock freedom.

However: many shared integer variables and starvation freedom only mean “eventually” - we want better than that.

Dekker’s Algorithm

Slide 91-92

Seems like there’s nothing important here. Another 2-thread lock that is a bit similar to the Peterson lock.

Lamport’s Bakery

Slide 93-108

Ensures mutual exclusion and **fairness** (first-come-first-served). Each thread will take a ticket and wait for its turn, i.e. when the ticket is the smallest. There will be two arrays: flag[n] and label[n].

As label computation isn’t atomic, there is the risk that two or more threads that try to acquire the lock at the same time generate the same label, in which case there needs to be a tie-breaker, for example: prioritize according to thread ID.

First-Come-First-Serve: whenever thread A finishes its doorway before thread B, A cannot be overtaken by B.

The lock satisfies mutual exclusion (proof by contradiction), FCFS, starvation freedom, and deadlock freedom.

However: many shared integer variables and labels grow without bound (in practice: overflow issues etc.). Possible to use a timestamping system instead of labels

General Remarks and Discussion on Registers

Slide 109-121

For correct implementation of the register locks, we must ensure correct event ordering and data race freedom.

Theorem: any deadlock-free algorithm that solves mutual exclusion for n threads by reading/writing to registers must use at least n distinct registers.

Black-White Bakery Lock

Slide 122-127

Solves the problem of Lamport’s Bakery Algorithm with unbounded labels using ticket colors. There is a global color (black/white, just a Boolean). A thread that wants to enter CS takes a colored ticket and waits until it has the smallest ticket with that color.

Registers

Introduction and Hardware Resources

Slide 1-18

Registers: memory locations that can be read and written to/from by different threads. Can handle Single/Multiple Readers/Writers (→ SRSW, MRSW, SRMW, MRMW). An **M-valued register** can store M values. **Sequential** registers will have an order of operations that follows the program order, i.e. each read will return the most recently written value.

Safe register: if a read interval doesn't overlap a write interval, it will return the most recently written value. Otherwise, it may return *any* value in the register's range.

Regular register: if a read interval doesn't overlap a write interval, it will return the most recently written value. Otherwise, it may return the most recent value *or* any of the overlapping written values.

In the safe and regular definitions, it's assumed that the registers are SW registers.

Atomic registers: reads and writes look as if they are totally ordered events where each read returns the most recently written value.

Linearization points: the instant at which an operation takes effect.

Construction of Atomic MRMW from Safe SRSW

Slide 19-55

Theorem: atomic MRMW registers are not more powerful than safe SRSW registers

Proof by construction:

- **Safe SRSW to Safe MRSW**

  An array of n safe SRSWs, each thread reads only a single register in the array so it will read the most recent value unless overlapping write.

- **Safe Boolean MRSW to Regular Boolean MRSW**

These are the same if the new value only will be written if it differs from the old value.

- **Regular Boolean MRSW to Regular M-valued MRSW**

Use M regular Boolean registers to represent the regular M-valued register. Keep an array to verify which values have been updated - this will be checked in the reading.

- **Regular SRSW to Atomic SRSW**

Assume that registers can store time-stamped values, this will prevent an old value from being returned.

- **Atomic SRSW to Atomic MRSW (→M-valued)**

The previous array construction doesn't as a thread with higher threadID has delayed updating. Instead, an n-by-n matrix of atomic SRSW registers can be used to ensure that later reading threads get the correct value. The writer will write to the diagonal of the array and the reader will read its corresponding column and write the updated value to its corresponding row:

- **Atomic MRSW to Atomic MRMW**

An array of atomic MRSWs with time stamps keeps track of the write order.

Note: the construction is not practical both in terms of memory requirements and time-stamping.

Progress Conditions

Slide 56-60

Non-Blocking Progress Conditions

Wait-free: in any concurrent execution, **each** thread can finish in a finite number of steps, independent of the actions of other threads.

Lock-free: in any concurrent execution, **a** thread can finish in a finite number of steps, independent of the actions of other threads.

Wait-free implies lock-free.

Blocking Progress Conditions

Starvation-free: in any execution, **any** thread will eventually be able to complete, provided that the other threads take steps (→ no thread will be “left out”).

Deadlock-free: in any execution, **some** thread will eventually be able to complete, provided that other threads take steps (→ the program will not “freeze”).

Starvation-free implies deadlock-free.

Non-Blocking but Dependent Progress Condition

Obstruction-free: a thread that executes in isolation can finish in a finite number of steps.

Atomic snapshots

Slide 61-65

Atomic snapshot: the state of the whole set of atomic registers at some point in the global time. It should therefore be possible to find a *valid* total order of operations such that sequential specifications are satisfied.

The ordering will be a **linearization** of the history. A simplified version of such a snapshot could be implemented by update and scan operations: each thread has a register which can be updated with a value or scanned. The scan operation takes a snapshot of all thread’s registers and copies them into a local array.

Obstruction-Free Snapshot

Slide 66-68

Each thread has a local time-stamp which is incremented on each update. A collect operation copies the registers thread into a local array, so if two successive collects return the same value, then the collected value can have had no update in this interval. A scan is completed when the last double collect is clean.

Wait-Free Snapshot

Slide 69-80

To not be dependent on other threads, a wait-free solution uses a helper scheme: updating threads will call scan before updating the register. When a scan fails, the thread can use the other thread's snapshot. To ensure that the other thread's snapshot occurred within the interval of the scanning thread, the scanning thread will perform a double collect. If the double-collect is dirty, it's known that that thread's update occurred within the interval, and can therefore be used.

This is a little bit fuzzy, not sure if my understanding of this protocol is completely correct, but it's probably not that important to know in detail…

Extension

An alternative wait-free snapshot to the one above is by letting the updating register take the snapshot after the update. The correctness proof is a bit more involved though.

Consensus Problem and Correctness

> 💡 “Excluding the detailed, universal construction, but you are supposed to know the main theorem”

Consensus Problem

Consensus Protocol

Slide 3-16

Consensus problem: n threads need to agree on a common value among local values provided by the threads, and no thread should possibly block another thread.

A **consensus object** has a decide-method that can be called at most once per thread with some local value v. The method must return a value that is consistent with all other threads, and valid (i.e. it must have been proposed by some thread).

A **consensus protocol** implements consensus in a wait-free manner. The **consensus number** of a class is the largest number of threads for which the class solves the consensus problem.

Theorem: atomic registers have consensus number 1 (proof by contradiction using state machinery) and are therefore insufficient to construct wait-free implementations of any concurrent object with consensus number ≥1.

(m,n)-Assignment Registers

Slide 17-33

An (m,n)-assignment register has n fields and can atomically write m (index, value) pairs to m indexed fields. **n≥m>1** and proof by construction:

We have a half-triangle of registers. Initially, all registers are set to a special value. When a thread calls decide, it will assign its corresponding row and column to the

suggested value. A value is decided on by finding the first thread that executed decide. The decision is made by scanning the diagonal and checking if the row/column value was overwritten by another thread, in which case the overwritten value came before the other one.

Concurrent Objects, Correctness, Consistency

Slide 34-39

Mandatory properties of concurrent objects:

- **Safety:** decided correctness requirement must be fulfilled
- **Liveness:** there must be progress

If two consistent objects also are consistent together, they are **compositional**.

Sequential Consistency

Slide 40-66

Sequential consistency: method calls should appear to happen in a one-at-a-time sequential order. For each thread, methods should appear to take effect in program order. This is not a real-time condition. In practice, it means that you can slide (but not reorder) the method calls in time to meet the object's sequential correctness specification. Sequential consistency is not compositional but non-blocking.

Skipping some definitions and technicalities on histories in this section.

Quiescent Consistency

Slide 67-71

Quiescent Consistency: method calls should appear to happen in a one-at-a-time sequential order. Method calls separated by a period of quiescence should appear to take effect in real-time order. Therefore less strict to follow program order for overlapping method calls, and a real-time component is introduced. Quiescent consistency is compositional.

Linearizability

Slide 72-81

Linearizability: method calls should appear to happen in a one-at-a-time sequential order. Each method call should appear to take effect in real-time order at some point between invocation and response. This point is called a **linearization point**.

Linearizability can be proved for some given history or for an object/algorithm as a whole. Often there are several options and possibilities, and it can be difficult to find/prove linearization by just looking at code.

Linearizability is compositional and a non-blocking property. Every linearizable execution is sequentially consistent.

Slide 82-115

Skipping some definitions, technicalities, and proof for the consensus number of a FIFO Queue in this section.

Hardware

Read Modify Write

Slide 116-124

Atomically reads and updates a register:

getandset(v) → overwrites

getandinc() → commutes

getandadd(k) → commutes

get()

Theorem: any RMW register in common has consensus number 2. A set of functions are common if they commute or overwrite each other. FIFO queues and Stacks have the same consensus number as common RMW, so we can construct wait-free queues/stacks using common RMW registers and atomic registers.

SWAP

Slide 125-127

Atomic SWAP operations swap the contents of two registers.

Theorem: SWAP has consensus number n (proof by construction with a wait-free decide-method).

CAS

Slide 128-134

An extended RMW operation on a register. It atomically compares the current content of the register to an expected value. If the values match, the register will be updated by the update value.

Theorem: CAS has consensus number ∞.

Universality of Consensus

Slide 135-153

Universal objects: if we use enough classes of an object, we can construct wait-free, linearizable implementations of any sequentially specified concurrent objects. Objects with consensus number ∞ (such as CAS) are universal.

Consensus can be used to linearize concurrent operations on the sequentially specified object - each thread locally performs the sequence of operations on a local copy of the object to compute the response. A linearized execution history can be represented by a list of nodes from which a response can be computed after a number of invocations. A new invocation means that a new node is added at the end of the list.

Theorem: this construction correctly implements the concurrent object and is lock-free.

In this construction, any thread can be overtaken an indefinite number of times. In a wait-free construction, each thread must be able to complete in a finite number of steps,

so helper schemes can be implemented to ensure this.

Extended CAS

Slide 154-155

Some small remark on multi-word CAS.

Transactional Memory

Slide 156-158

Transaction: a sequence of instructions that are executed atomically

Practical Locks

Introduction

Slide 1-14

Spin lock: thread keeps actively trying to acquire the lock (Peterson/Baker lock for example). Suitable if critical section is short, but keeps processor busy.

Blocking lock: thread suspends, scheduler reawakens thread when lock becomes free. Better for long critical sections.

Locks

Test and Set Locks

Slide 15-19

Test-and-Set: one atomic flag per lock, thread acquires lock by changing the flag from true to false. Bad performance, high contention.

Test-and-Test-and-Set: only do test-and-set if there's a chance to succeed → less contention. Possible to implement with a random back-off time to avoid that threads wake up at the same time.

Both are clearly unfair and have big memory contention issues.

Ticket Lock

Slide 20-22

A ticket lock with an atomic counter → thread will wait for its turn. FCFS and space-efficient. Risk of false sharing if both ticket and counter are on the same cache-line.

Array Lock

Slide 23-30

A circular array of atomic memory locations (one per thread), lock spin on the next free slot. All slots will be false except the current thread. FCFS, is not space-efficient, and

false sharing may occur.

CLH Queue Lock

Slide 31-39

Linked list where all threads except the first are locked and will stay locked until the one before (i.e. first in line) is no longer locked. Sentinel node for the tail. Easy memory management, FCFS, not much false sharing since each thread spins on its own node (i.e. on a separate location).

MCS Queue Lock

Slide 40-48

Unlike CLH, it maintains the head instead of the tail. To unlock, you instead modify the lock of the next node. Also FCFS.

Queue Locks with Timeout (try-lock)

Slide 49-57

It may be a good idea to include timeouts: you won’t deadlock or end up with a long delay in case a thread falls asleep or fail. One approach is to mark the node as abandoned and let the successor clean up the node.

Composite Lock

Slide 58-66

Combines back-off lock with queue lock by preallocating a fixed number of nodes which will be randomly acquired by threads before being queued. This can reduce contention.

Hierarchical Locks

Slide 67-77

Made to minimize lock migration (see slide 67).

Back-off Lock

Length of back-off is dependent on distance to lock owner, so local threads are more likely to acquire lock → benefit from faster access due to locality. This may starve remote threads.

CLH Lock

One global queue for all processor clusters, and one local queue per cluster. Node is added to the local queue, and the first node in the local queue is the cluster master that joins the local queue with the global queue.

Slide 78-82

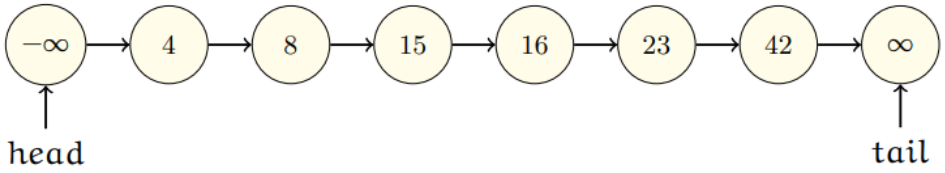
Lock cohorting: a technique to construct NUMA-aware locks from normal locks.

Concurrent Data Structures Part 1

List-Based Set

Slide 1-10

The list-based set will have an add, remove, and contain function. It's implemented by a sorted, linked list of node elements that are identified by unique keys from an ordered space. The head and tail are sentinel nodes containing the minimum and maximum key values ($\pm\infty$).



Locking

Coarse-Grained Locks

Slide 11-12

Put a single global lock at the beginning of each method call which is released in the end - essentially works as the sequential implementation.

Reader-Writer Locks

Slide 13-14

RW lock instead of normal lock → concurrent read is allowed so the performance is improved sine contains is the most common operation.

Fine-Grained Locks

Slide 15-42

The data structure can be slit into independently synchronized components, using a lock per node instead of a global lock.

Add: propagate lock-pairs over the list until the correct value-pair has been reached, then insert the new item between the locks before unlocking from behind.

Remove: similar fashion but the item is removed after locking. The behind node is relinked before the removed node is unlinked from the list.

Synchronization

Optimistic Synching

Slide 43-62

Instead of locking while searching for the components, we will only lock the found nodes. After locking, they must be verified → if nodes changed between locking and searching they will be released and the search will be reinitiated. This means that removed nodes may still be pointed to by some other thread, so memory reclamation is tricky. Progress is not guaranteed.

Lazy Synching

Slide 63-89

Since the list is traversed twice, there's a lot of overhead in the list with optimistic synching. Lazy synchronization postpones some hard operations or gets help from other threads. The remove operation is then split into a logical removal where it's marked as removed, but will only be physically removed (unlinked) at a later stage. Nodes are still searched for without locking, but verification is only done by checking the marked flags of the item, instead of by going through the list a second time.

Lock-Free

Slide 90-102

CAS operations can be used to update the next-pointer when adding and removing items to make the set lock-free. This further requires the next-pointer and the marked-flag to be treated as a single atomic unit to avoid errors (see slide 90). Add and remove will be lock-free since *some* thread will have made progress even if the CAS operations fail.

Slide 103-105

Conclusion: the most suitable implementation depends on the use-case. Coarse-grained synchronization can be used with any data structure but can easily become a bottleneck when the data structure is used a lot. Other implementations require deep knowledge of the implementation. The best theoretical progress is obtained by the lock-/wait-free data structures, but the atomic operations can be expensive so that’s a trade-off too.

Queues and Stacks

Slide 1-3

Nothing special in these slides, just an explanation on what queues/stacks are basically…

Concurrent FIFO Queues

Slide 4-9

FIFO queues support enqueues and dequeues and is typically implemented with a linked list of items (nodes) and a sentinel node for the head. See slides for how sequential implementation works.

Bounded with Fine-Grained Locking

Slide 10-16

The capacity and current size of the queue is maintained by the queue object. Enqueues will wait for a dequeue is the capacity is met, likewise, a dequeue will wait for enqueues when the queue is empty. The list can be replaced with a circular array since the queue is bounded anyways.

The enq/deq locks are called upon an operation. The locks operate with condition variables: if the thread needs to wait until it can perform its operation, it will wait. The wait is managed by condition variables - since spurious wakeups are possible it’s important to recheck the condition.

Unbounded with Fain-Grained Locking

Slide 17-18

Similar to the bounded implementation but without the waiting. Progress is not ensured because of locking.

Unbounded and Lock-Free

Slide 19-30

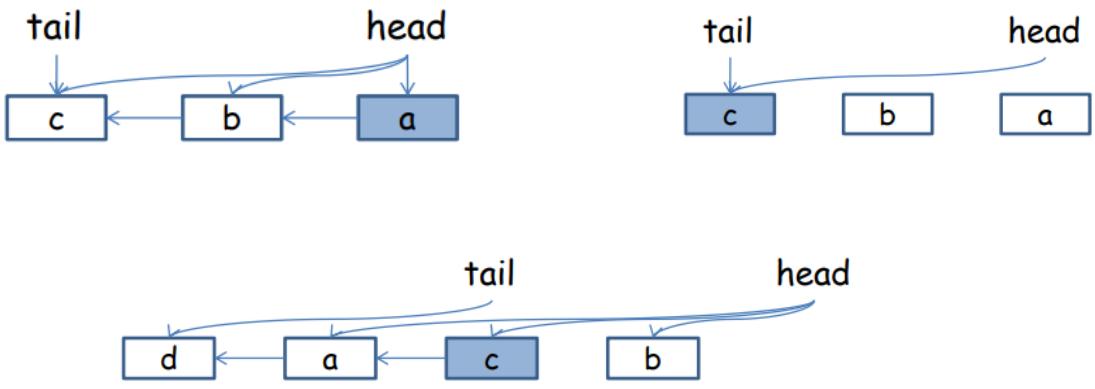
Uses CAS to update the list. Since the enqueue requires the tail and next-field of tail to be updated, other threads may encounter unfinished updates, so we must use a helper scheme. If the queue is empty, the tail will be helped by the dequeue operation before advancing the head (dequeuing). This is to avoid that the enqueuer

ABA Problem

Slide 31-42

The ABA problem/recycling problem occurs when a value changes from A to B, and then back to A. Because the CAS expects an A, it succeeds, even though the context and entity have changed completely, for example:

1. Thread A reads that the head==a and head.next==b, however the CAS has not taken place yet.
2. Another thread dequeue b successfully, so both a and b are freed while c is left as the sentinel node.
3. Node a is recycled and enqueued again, another node d is enqueued as well.
4. c is dequeued and a is now the sentinel node.
5. Thread A wakes up and performs a CAS on the head. Since the head is still a, the CAS will succeed and the head will be updated to head.next=b even though b already has been freed and is unlinked from the rest of the queue.



This is a frequent problem with CAS operations where only a reference/pointer is compared, not the state as a whole. LL/SC atomics (load-linked/store conditional)

doesn’t have this problem and may therefore be more suitable for some algorithms, since SC fails if there has been a change on the address since LL.

Another possible solution is the use of timestamps, then again we might not have enough bits to store timestamps in our single-word, in which case we need multi-word CAS operations.

Skipping some remarks on lock-free CAS algorithms on slide 41-42.

Work-Stealing Dequeue

Slide 43-46

Work-stealing is a technique for load-balancing where each thread has its own task dequeue. It will add and remove tasks from the bottom of the dequeue. When the local dequeue is empty it will steal tasks from the top of another dequeue instead. The most frequent operations will be the local operations to the bottom of the dequeue, so the concurrent operations are minimized. No atomics are needed for the local access (unless there is only one element in the dequeue), but a CAS is required for stealing access. Can be implemented as an unbounded array.

Lock-Free Array Implementation

Slide 47-58

Uses CAS → ABA problem may occur but can be solved with timestamps. Popping from top may further fail spuriously even when dequeue has a lot of items left (not really explained why in the slides?).

To avoid using an unbounded array we can use an array of size N, where we don’t allow pushes when the dequeue is full.

Concurrent LIFO Stacks

A LIFO stack supports push and pop operations and is implemented with a linked list and a reference to the top.

Lock-Free Stack

Slide 59-79

Pushes and pops can be implemented with CAS operations → thread will back off upon failure. If popped items can be reused, the stack has an ABA problem which can be repaired with timestamp bits.

In both the stack and the queue we have a sequential bottleneck at the head/tail/top, since all threads must compete for the top of the stack.

Backoff Stack

Slide 80-94

Overlapping pushes and pops could be paired together without having to access the stack. This can be implemented using an array: when pushing, a thread will put its value into a random index where it waits for a pop. If it times out it will try the stack instead. The same goes for pop-operations: a random index is chosen and the thread will wait for a corresponding push to that index until time-out. This requires a lot of tuning though: how big is the array, how long is the time-out, etc.

Relaxed Linearization

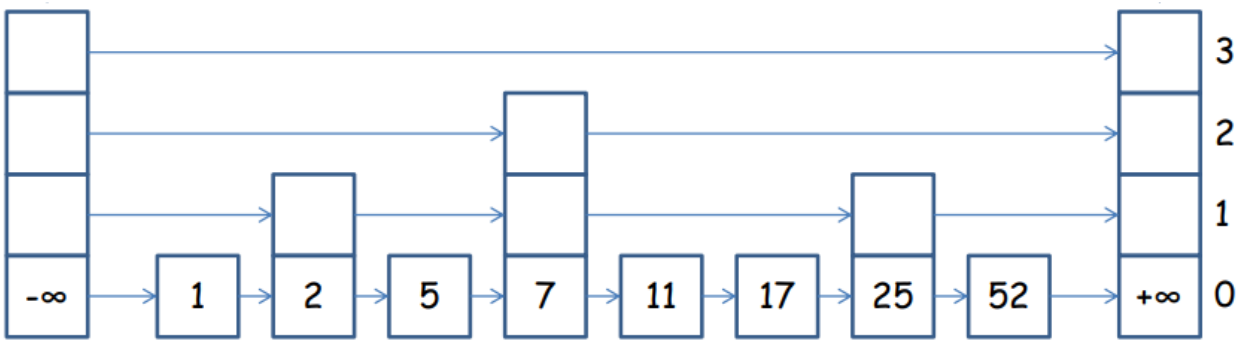
Slide 95-97

Queues and stacks are implemented with strong ordering/fairness guarantees, which might be too strict for concurrent data structures as it gives little potential for parallelization. An alternative could be to pop elements that are close enough to the top.

Skiplist

Slide 98-108

An efficient, randomized, list-based search structure. It’s essentially an ordered list with multiple higher-level “sub”-lists for faster navigation. The lowest level represents the whole list, and each level has fewer elements → the storage is O(n) with some irrelevant constant. Searches start at the highest level to find the range within which the value should be. The search goes down each level until the value is found (or not). When new nodes are inserted, it will generate a random number of levels on which it’s inserted. All operations are O(log n).



Lazy Skiplist

Slide 109-134

Utilizes fine-grained locking, wait-free contains operations, and lazy removal of elements using marked-flags.

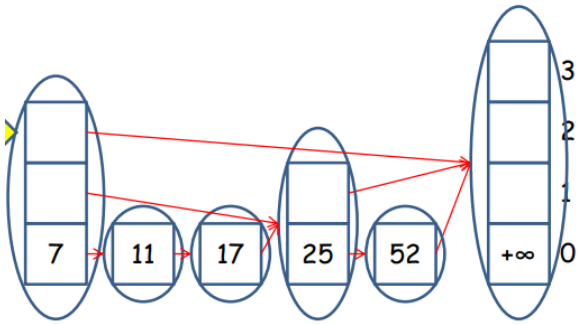
Add: linking from level 0 and up. Similar to previous queue, you will lock the predecessors (up to the level of the new item) and validate that they haven't changed before linking. Once linked, a "fully-linked"-flag should be set.

Remove: remove from the max level and works down to the lowest level. The node to be removed will be locked then marked. The predecessors will then also be locked and validated before the node is linked out. There will also be checks to see if the victim was not yet fully linked in, or is being concurrently unlinked by another thread.

Lock-free Skiplist

Slide 135-167

Aims to implement lock-free lists at each level. Since it's only possible to update the list on one level at a time (single-word CAS), it's not possible to maintain the skiplist property. Instead, the list at level L will be a shortcut into the list at level L-1 (some sort of abstract pointer set-up kind of - see fig).



Find: physically unlinks marked nodes at each level with CAS operations

Contains: skip marked links on each level, just make sure that the item eventually is found at level 0.

Add: the new node is linked starting from the lowest level. If the CAS fails it means that the predecessor has changed (either by getting marked or by another node being inserted). When the node has been added to the lowest level, the linking will proceed until all levels have been linked. This may however violate the sublist property if the new node is currently being removed by another thread.

Remove: if the node is found, it will be marked starting from the highest level. The operation succeeds if marking succeeds at the bottom level. If CAS fails, it means that the node has been concurrently linked out by some other thread, so it will regardless be physically linked out by the following find operation.

Priority Queue

Slide 168-175

Priority queues can be implemented using skiplists by adding atomic candidate flags to the nodes.

Slide 176-186 not on exam

Memory

Slide 1-5

Memory models regard software whereas memory behavior is observed in hardware. How the actual memory behavior works can be *very* confusing 🙃

Sequential Computing and Cache Coherence

Slide 6-25

We expect loads and stores to take place in a sequential program order, for example only the most recently written value should be returned from a register. It’s possible that the value of a register differs between cache and memory, but sequential semantics allow the use of coherent caches. This means that the caching is functionally invisible (as if caches didn’t exist).

Write buffers write to memory in batches to postpone the latency of writes.

Cache Coherence

Definition 1: all cores’ loads and stores to a single memory location must happen in a total order that respects the program order of each thread.

Definition 2: every store eventually becomes visible to each core. Stores to the same memory location are serialized.

Definition 3: a load from memory location A returns the value of the most recent store to A by the same core, unless another core stored to A in between. If another core stored to A in between, that value must be returned provided that the two operations are “sufficiently separated in time”. Stores to the same memory locations should furthermore be serialized.

Definition 4: assumes the lifetime of a memory location A is divided into separate epochs. The cache is coherent if registers are MRSW at any given time, and the value of the memory location at the start of an epoch is the same as the value at the end of the last write epoch.

Different Memory Locations

Slide 26-33

These definitions take into account multiple operations on a single register, but when looking at multiple registers the coherence may not hold (see slides).

Consistency Problems and Models

Slide 34-42

Memory consistency models constraints what a core can observe on load and store operations, or specifies the allowed load/store behavior of a multi-core program with shared memory. These are implemented either in hardware or software.

Total Store Order

Slide 43-48

Total Store Order means that the processor uses write buffers to hold committed stores until the memory system can process them. The write buffer essentially hides the latency of a missed store by postponing them and ordering them after the loads (or the other way around).

Shitty explanation in the slides, think it becomes a bit more clear when looking at FENCEs and stuff…

Memory Consistent Models

Slide 49-54

TSO is weaker than sequential consistency, meaning that executions allowed by SC is a subset of the executions allowed by TSO (SC is more constraining than TSO).

FENCE: a hardware mechanism that enforces order - it implies that operations cannot be reordered according to the fence. Since the only difference between TSO and SC is that stores can be postponed and ordered after loads (or the other way around), FENCEs can be inserted to obtain SC behavior.

Atomic Operations

Slide 55-61

Atomic load/store operations take place at the same time for all threads and is usually implemented by exploiting the cache-system (not by using special hardware).

Remark: C/C++ weren't designed for multi-threading, so the compilers are mostly unaware of the threads and might perform optimizations that break expectations on the memory model. **Volatile** enforces load from memory.

Other Consistency Models

Slide 62-88

Skipped some confusing stuff here…

Terminology:

- Coherence: all operations to any single location appear to occur in some single, total order from the perspective of all threads
- Global order: synchronizing operations happen in some total order in which operations by the same thread happen in program order
- Program order: operations happen in the order specified by the compiled program from the perspective of the issuing thread
- Local order: ordinary operations may be reordered with respect to synchronizing operations by the issuing thread
- Value read: any read operation returns the value most recently written in the order observed by the thread

Locks should ensure that it's actually mutually exclusive, this should be enforced with FENCEs. Most libraries already make this guarantee.

Memory Consistency in Programming Languages

Slide 89-90

Programming languages should make the memory consistency model explicit

Race Condition

Slide 91-95

Race Condition: two or more threads read/write to a memory location at the same time, and essentially leads to undefined behavior. This is avoided by protecting all shared memory locations with locks or other things.

A program is **data race free** if it has no data races when executed under sequential consistency, this is up for the programmer to ensure.

C11/C++11 Memory Model

Slide 96-114

Atomic objects act as synchronization operations which is transitive between threads. Operations before atomic stores cannot be ordered after the store, and operations after atomic load cannot be ordered before, unless we work with acquire-release. There are a bunch of memory order definitions that can be applied to the store/load/exchange atomic operations to specify how the memory order should be interpreted.

Best to go through these slides again.

Java, OpenMP/pthreads and MPI

Slide 115-119

Barely any info here.

Memory Reclamation

Slide 120-123

Non-garbage collected languages like C/C++ free pointers to give them back to the memory manager. It can be worth using reference counters to ensure that the pointer is no longer used before freeing it.

Memory reclamation schemes aims to be:

- **General:** should work for any data structure and have no constraints on the structure of the allocated nodes
- **Portable:** should require no OS support
- **Thread oblivious:** no fixed upper bound on number of threads

- **Strong progress guarantees:** wait-free, lock-free, and a crashed thread shouldn’t prevent the reclamation of an unbounded amount of memory
- **Efficient:** constant-time overhead per reclaimed node

Lock-Free Reference Counting

Slide 124-126

Memory is allocated and freed as nodes → add a reference count to each node, when the count drops to 0, the node may be reclaimed. Drawback can be that nodes in cyclic data structures may never be freed → memory leaks.

Hazard Pointers

Slide 127-128

Sensitive nodes are referenced through special pointers (Hazard pointers). Each thread has some fixed number of hazard pointers, and it’s the way for a thread to communicate to other threads that it’s holding a reference to a shared node. When a thread wants to reclaim a node it must first scan all the other threads’ hazard pointers.

Stamp-it

Slide 129-149

Holds a local retire list of nodes to be reclaimed, a global retire list, and a stamp pool of thread elements. The programmer has to mark critical regions (NOT critical section) where nodes may be shared. The stamp pool holds time-stamped elements to keep track of which elements can be retired. All nodes with a stamp less than the currently lowest stamp will be reclaimed.