Ch. 1 Introduction

- parallelism -> using multiple processors to work on a single task
- shared-memory multiprocessor -> multiple processors that work in parallel and have access to the same memory
    - often produces many problems
        - cache coherence
        - synchronization
        - memory consistency
- multi-core processor -> a single computing component with two or more independent processing units that read and execute program instructions simultaneously
- aysnchronous -> program tasks do not necessarily have to execute sequentially, i.e. the results of one task are not necessary as inputs to the next task
- threads -> the smallest sequence of programmed instructions that can be managed independently by a scheduler
    - a component of a process
    - multiple threads can exist within one process, executing concurrently and sharing resources such as memory
    - the threads of a process share its executable code and the values of its global variables at any given time
- concurrent program -> a program in which several tasks or computations are executed within overlapping time periods
- concurrent algorithm -> an algorithm in which several tasks or computations are executed within overlapping time periods
- program correctness -> the state of a program where it is possible to provably show that a program adheres to some formal specification or property
- safety property -> the definitive assurance that for a every possible execution, nothing bad happens for a component of computation (e.g. program, task, method, etc)
- transactional memory -> a concurrency control mechanism that ensures load and store operations will occur in an atomic way (think atomics in C++)

Shared Objects and Synchronization

- race conditions are caused by indeterminate interleaved reads and writes from multiple threads
- modern processors provide read-modify-write instructions that allow threads to perform these operations atomically

- mutual exclusion - technique for making sure that only one thread at a time can execute a particular block of code at a time
- understanding mutual exclusion is necessary to understand concurrent computation (and key for deadlock, bounded fairness, and blocking vs. nonblocking synchronization)

A Fable

- coordination protocol - agreed method for sharing access to a resource
- basic idea (mutual exclusion protocol)
    - actor 1
        a. set flag
        b. if actor 2's flag is unset, perform action
        c. on completion, unset flag
    - actor 2
        a. set flag
        b. if actor 1's flag is set
            a. unset flag
            b. wait until actor 1's flag is unset
            c. set flag
        c. on flag set (when actor 1's flag is unset) perform action
        d. on completion, unset flag
    - flag principal -
        - if actors 1 & 2
            a. set own flag
            b. check other's flag
        - then at least one will see the other's flag set and will not act
        - NOTE: does not prove actions never occur concurrently
        - to prove this, assume actions are occurring concurrently
            - for actor 1 to act, actor 2's flag was unset on check
            - for this, actor 2 must not have completed setting before actor 1 checked
            - this means when actor 2 checked last, it was after actor 1 set the flag
            - this means actor 2 would not have acted and thus negates the possibility that they acted concurrently
- this description assumes nothing about the time period for setting or getting the state of the flag
- only concerned with start and end point of setting and getting, i.e. critical point of change

Properties of Mutual Exclusion

- correctness of concurrent solutions depends on many properties, one of which is mutual exclusion

- deadlock-freedom - property that dictates that at least one action will always occur
- starvation-freedom (aka lockout-freedom) - property that dictates that one actors intention to act means that it will eventually act
- waiting - property that dictates that an actor will wait indefinitely according to the coordination protocol
- waiting is related to fault tolerance
  - https://users.ece.cmu.edu/~koopman/des_s99/sw_fault_tolerance/
- all mutual exclusion protocols require waiting
- other coordination protocols do not require waiting

The Moral

- two kinds of communication occur naturally in concurrent systems
  - transient - both parties participate at same time
  - persistent - each party can participate at different times (using some form of persistent state)
- mutual exclusion requires persistent communication
- interrupts - common method for one thread to signal another (achieved by setting a bit at a predetermined location and checking that bit between threads)

The Producer-Consumer Problem

- the producer's job is to generate data repetitively
- the consumer is consuming the data one piece at a time concurrently
- need to ensure the producer won't add data when it isn't possible and that the consumer won't try to remove data when it isn't possible
- basic idea (producer-consumer protocol)
  - producer uses a flag and sets it when data is available
  - consumer
    a. waits for set flag
    b. acts on data
    c. on action completion, resets flag if possible
  - producer
    a. waits for flag unseet
    b. produces data
    c. sets flag
  - flag reflects data availability (set means available, unset = unavailable)
- properties
  - mutual exclusion - producer and consumer are never acting on data at same time
  - starvation-freedom - if producer continues to produce data and the consumer continues to consume data the consumer will always have data to consume

    - producer-consumer - the consumer will not attempt consume if there is no data and the producer will not produce data if the previously produced data has not yet been consumed
- the producer-consumer protocol cannot be used for mutual exclusion
    - mutual exclusion requires deadlock freedom which is not possible with the producer-consumer protocol because the consumer cannot consume if the producer has not produced and the producer cannot produce if the consumer has not consumed
- proofs
    - mutual exclusion
        - the flag has two states and changes only occur after transitions
        - if the flag is set, only the consumer can consume
        - in order for the flag to be unset, the consumer must complete consumption
        - if the flag is unset, only the producer can produce
        - in order for the flag to be set, the producer must complete
        - mutual exclusion holds in each state
    - starvation-freedom
        - the consumer must be willing to consume infinitely often and the flag cannot be unset
        - the flag then must be set, and since the consumer is willing to consume, the consumer will consume until completion, thus unsetting the flag
    - producer-consumer
        - mutual exclusion implies the producer will never produce while the consumer consumes
        - the producer will not produce until until the consumer unsets the flag which can only be done if consumer is complete
        - the consumer will not consume until the flag is set which will not happen until the producer has completed production
- waiting
    - if producer produces but does not set the flag the consumer will never consumer
    - if the consumer consumes and does not unset the flag, the producer will never produce
- producer-consumer appears in almost all parallel and distributed systems
- it is the method used for communication buffers for transmission across a network or across a shared bus

The Readers-Writers Problem

- actors want to communicate through a shared resource but cannot access the resource at the same time
- consider the shared resource to be composed of multiple elements
    - the writer can write and the reader may read an incomplete message before the writer writes again

- can use the mutual exclusion protocol to ensure only complete messages are read but does not ensure all messages will be read
- can use a flag to signal when a message can be read
- the readers-writers problem does not have solutions that do not require waiting

The Harsh Realities of Parallelization

- increasing parallel processing should ideally increase computational power linearly
- most real-world computational problems do not display this linear increase because parallelizing computations incurs the cost of IPC and coordination
- Amdahl's Law - any potential possible speed increase of a complex task is limited by the required sequential execution of the job's component tasks
- speedup -> the speedup, S, of a job is the ratio between the time it takes one processor to complete the job vs the time it takes n concurrent processors to complete the same job
- Amdahl's law formally
  - characterizes the max speedup, S, that can be achieved by n processors collaborating on an application (computation, task, algorithm, etc)
  - p - the fraction of the job that can be executed in parallel
  - assume takes normalized time of 1 for a single processor to complete the job
  - parallel part takes time p/n
  - sequential part takes time 1 - p
  - parallelized computation takes 1 - p + p/n
  - result
    - S = 1 / (1 - p + p / n)
- extrapolated result
  - if only able to parallelize 90% of a computation, the resulting speedup is only 5x (as opposed to 10x)
- goal of this book is to help programmers understand the tools and techniques for creating communication and synchronization in order to facilitate the ability to parallelize as much of their code as possible (e.g. the remaining 10%)

Parallel Programming

- no cookbook recipe for identifying components of an algorithm that can be parallelized

Ch. 2 Mutual Exclusion

- covers classical algorithms that work by reading and writing shared memory

Time

- concurrency requires reasoning about time and how intervals overlap
- threads share a common time (not necessarily a common clock)
- thread is a state machine
- events - state transitions
- events are instantaneous - they occur at a single instant in time
- convenient to require that events are never simultaneous and events that occur close in time can be assumed to be in any order
- a thread A has a set of events in running time a sub 0 ... a sub i and loops are thought of as a single event denoted by a sub i sup j
- a -> b means a precedence relation
- interval is duration between events and concurrent intervals have overlap between that duration

Critical Sections

- a block of code that can be executed by only one thread at a time (aka mutual exclusion)
- standard way to achieve mutual exclusion is through a Lock object

```
class Lock {
public:
    void lock();
    void unlock();
};
```

- acquires/locks, releases/unlocks
- thread is well formed if
    i. each critical section is associated with a unique Lock object
    ii. thread acquires the critical section at the start of each critical section
    iii. the thread releases the lock when it leaves the critical section
- in Java use a try...finally block to ensure the lock is always unlocked

```
mutex.lock();
try {
    // do work with shared data
} finally {
```

```
    mutex.unlock();
}
```

- properties for a lock algorithm
  - mutual exclusion - running of critical sections within different threads does not overlap (i.e. interval/duration between lock event and unlock event)
  - freedom from deadlock - some thread will always succeed in acquiring the lock when one or more threads are attempting to acquire the lock (if a thread attempts to acquire the lock and never does it means other threads are completing infinitely many critical sections)
  - freedom from starvation (aka lockout freedom) - every thread that attempts to acquire the lock eventually succeeds (i.e. every call to lock eventually returns

Thread Solutions

The LockOne Class

- 2-thread lock
- code here
- generally use the volatile keyword to ensure the compiler does not reorder or optimize out instructions
- use write sub A(x = v) to denote a write event and read sub A(x == v) to denote a read event
- Lemma 2.3.1 -> LockOne class satisfies mutual exclusion
- provides proof here
- deadlocks if thread executions are interleaved
- if one thread runs before the other then no deadlock occurs

The LockTwo Class

- 2-thread lock
- code here
- Lemma 2.3.2 -> LockTwo satisfies mutual exclusion
- provides proof here
- deadlocks if one thread runs completely ahead of the other
- if the threads run concurrently, lcok() succeeds
- LockOne and LockTwo complement each other, one succeeds under condition that another fails

The Peterson Lock

- combining LockOne and LockTwo creates a starvation-free Lock algorithm

- very elegant and succinct two-thread mutual exclusion algorithm
- code here
- Lemma 2.3.3 -> Peterson lock satisfies mutual exclusion
- provides proof here
- Lemma 2.3.4 -> Peterson lock is starvation-free
- provides proof here
- Corollary 2.3.1 -> Peterson lock is deadlock-free

The Filter Lock

- two solutions that work for n > 2 threads
- Filter lock uses n - 1 waiting levels that a thread must traverse before acquiring a lock
- levels satisfy the following properties
  - at least one thread attempting to satisfy level l succeeds
  - if more than one thread is trying to enter level l then at least one is blocked (i.e. will wait at level)
- Peterson lock uses a two-element boolean flag array to indicate whether a thread is trying to enter the critical section
- Filter lock generalizes this idea with an n-element integer level array
- `level[A]` indicates the highest level that thread A is currently trying to enter
- levels = levels of exclusion
- each level has a distinct `victim[l]` field to filter out a thread excluding it from the next level
- Lemma 2.4.1 -> for j between 0 and n - 1, there are at most n - j threads at level j
- proof here
- Corollary 2.4.1 -> the Filter lock satisfies mutual exclusion
- Lemma 2.4.2 -> the Filter lock is starvation-free
- proof here
- Corollary 2.4.2 -> the Filter lock is deadlock-free

Fairness

- starvation-freedom guarantees every thread that calls lock() eventually enters the critical section but does not guarantee duration in critical section
- informally, if A calls lock() before B it should enter the critical section before B
- cannot determine which thread called lock() first with current implementation
- to facilitate this, split lock into two sections
  i. doorway -> execution interval D sub A consists of a bounded number of steps
  ii. waiting -> execution interval W sub A can consist of an unbounded number of steps

- bounded wait-free progress property -> requirement that doorway will always finish in a bounded number of steps

Fairness

- Definition 2.5.2 -> a lock is first-come-first-served if, when thread A finishes its doorway before thread B finishes its doorway, then A cannot be overtaken by B

Lamport's Bakery Algorithm

- maintains the first-come-first-served property by using a distributed version of the number-dispensing machines (i.e. machines that distribute an order to elements in a queue)
- each thread takes receives a number when it enters the doorway and then waits until no thread with an earlier number is trying to enter the doorway
- code here
- `flag[A]` is a boolean flag indicating whether A wants to enter the critical section, and `label[A]` is an integer that indicates the thread's relative order when entering the doorway of the critical section, for each thread A
- generates a new label by
  i. reads all other threads' labels in any order
  ii. reads all other threads' labels in some arbitrary order and generates a label that is one greater than the maximal label
- uses lexicographic ordering on pairs of labels and thread ids
- labels are strictly increasing
- labels are read asynchronously so the set of labels read by a thread in a section may not have existed in memory at the same time but the algorithm still works
- Lemma 2.6.1 -> Bakery lock algorithm is deadlock-free
- proof here
- Lemma 2.6.2 -> Bakery lock algorithm is first-come-first-served
- any algorithm that is both deadlock-free and first-come-first-served is also starvation free
- Lemma 2.6.3 -> Bakery algorithm satisfies mutual exclusion
- proof here

Bounded Timestamps

- labels of Bakery grow without bound which will eventually result in overflow
- overflow will negate the first-come-first-served property
- labels in the Bakery lock act as timestamps
- threads need two capabilities
  - read the other threads' timestamps (i.e. scan)
  - assign itself a later timestamp (i.e. label)

- code for Timestamp system here
- must be wait-free (see references for how to construct wait-free concurrent timestamp system)
- simpler problem:
  - construct a sequential timestamping system
  - threads perform scan-and-label sequentially (as if with mutual exclusion)
- underlying principles of sequential vs. concurrent timestamping are similar but differ in detail
- think of timestamps as nodes in a directed graph (called a precedence graph)
- edge from a -> b means b occurs after a
- irreflexive -> no edge from any node a to itself
- antisymmetric -> if edge a -> b, no edge b -> a
- no requirement of transitivity -> edges a -> b and b -> c do not imply an edge a -> c exists
- assigning a timestamp to a thread = placing the thread's token on the timestamp's node
- see this section for a more detailed description of the n-thread labeling system
- note -> two threads can never form a cycle

Lower Bounds on the Number of Locations

- Bakery algorithm is not practical because need to read and write n locations, where n is the maximum number of concurrent threads (which may be very large)
- no lock algorithm avoids this overhead
- any deadlock-free Lock algorithm requires allocating and then reading or writing at least n distinct locations in the worst case
- limitation of memory accessed solely by reads/writes (aka loads/stores):
  - any information written by a thread to a given location can be overwritten without any other thread observing it
- requires definitions of state for memory
  - object's state -> state of object's fields
  - thread's local state -> state of program counters and local variables
  - global/system state -> state of all global objects + local state of threads
- Definition 2.8.1
  - inconsistent state -> a lock object's state, s, is inconsistent in any global state where some thread is in the critical section, but the lock state is compatible with a global state in which no thread is in the critical section or is trying to enter the critical section
- Lemma 2.8.1 -> no deadlock-free lock algorithm can enter an inconsistent state
- proof here
- Definition 2.8.2
  - covering state -> a covering state for a lock object is one in which there is at least one thread about to write to each shared location, but the Lock object's locations present the

appearance that the critical section is empty (i.e. the locations' states appear as if there is no thread either in the critical section or trying to enter the critical section)

- Theorem 2.8.1 -> any lock algorithm that, by reading and writing memory, solves deadlock-free mutual exclusion for three threads must use at least three distinct memory locations
- proof here
- it follows that n-thread deadlock-free mutual exclusion requires n distinct memory locations
- the Peterson and Bakery locks are optimal within a constant factor
- limitation of read and write operations:
  - information written by a thread may be overwritten without any other thread reading it
- modern architectures provides instructions that overcome this limitation

Ch. 3 Concurrent Objects

- safety and liveness properties aka correctness and progress

Concurrency and Correctness

- shows lock-based concurrent FIFO queue that stores a lock and locks in both the enqueue and dequeue methods
- implementation allows multiple enqueuers/dequeuers
- shows lock-free/wait-free concurrent queue
- implementation presents a single enqueuer/single dequeuer queue which means the queue can be accessed from two separate threads, one thread which performs enqueueing and one thread that performs dequeueing
- a result of Amdahl's law is that concurrent objects that use exclusive locks within their methods are worse than ones that use finer-grained locking and those are worse than ones that do not use locks at all
- easier to reason about concurrent objects if their execution can be mapped to a sequential model
- this principal is key to correctness properties

Sequential Objects

- precondition -> some required state that an object (including global state when also accounting for arguments) must be in before executing a method
- postcondition -> the state of an object (and global state) after a method's execution, including return value
- side effect -> change to an object's state (or the global state)
- note: references to global state above includes expanding containing scopes but this is not mentioned in the book

- sequential specification -> documentation of the full set of preconditions and postconditions possible for a method (can ignore intermediate states)
- cannot assume so much when considering method interactions etc. when there is concurrent access to an object (mainly due to the potential for method overlap, which means intermediate states must also be considered)

Quiescent Consistency

- method calls take time an start with an invocation event and a response event (which should appear to take effect instantaneously)
- sequential method calls will never overlap but concurrent method calls may overlap
- pending -> a method call whose invocation event has occurred but whose response event has yet to occur
- register -> object version of a read-write memory location (i.e. a data member or property) NOTE: this name persists due to historical reasons despite confusing relationship
- Principle 3.3.1 ->
  - method calls should appear to happen in a one-at-a-time sequential order
- quiescent -> an object is quiescent if it has no pending method calls
- Principle 3.3.2 ->
  - method calls separated by a period of quiescence should appear to take effect in their real-time order
- quiescent consistency -> any time an object becomes quiescent then the execution so far is equivalent to some sequential execution of the completed calls
- a quiescently consistent counter is an index distribution mechanism that can be used as a loop counter but the clients of the loop counter must not be concerned about the order in which the indexes are used

Remarks

- quiescent consistency never requires one method call to block waiting for another to complete
- total method -> defined for every object state
- partial method -> not defined for every object state
- in any concurrent execution, for any pending invocation of a total method, there exists a quiescently consistent response
- quiescent consistency is a nonblocking correctness condition
- compositional -> a correctness property, P, is compositional if, whenever each object in the system satisfies P, the system as a whole satisfies P
- compositionality is important for large systems and closely related to modularity (using hidden implementations and exposed interfaces)
- the consistency guarantees are exposed through the interface as well

- quiescent consistency is compositional

Sequential Consistency

- program order -> the order in which a single thread issues method calls
- method calls by different thread are unrelated by program order
- Principle 3.4.1 ->
  - method calls should appear to take effect in program order
- sequential consistency -> a correctness property that combines principles 3.3.1 and 3.4.1, e.g.
  - sequential consistency requires that method calls act as if they occurred in a sequential order consistent with program order
- in any concurrent execution, there is a way to order methods call sequentially so that they
  i. are consistent with program order
  ii. meet the object's sequential specification

Remarks

- sequential consistency and quiescent consistency are incomparable
  - there exist sequentially consistent executions that are not quiescently consistent, and vice versa
- in most modern multiprocessor architectures, memory reads are not sequentially consistent
  - the can be typically reordered (without notification)
- memory barriers or fences -> special instructions provided by multiprocessor architectures that instruct the processor to propagate updates to and from memory to ensure that reads and writes interact correctly
- sequential consistency
  - nonblocking
  - not compositional

Linearizability

- the result of composing sequentially consistent components is not itself necessarily sequentially consistent
- Principle 3.5.1 ->
  - each method call should appear to take effect instantaneously at some moment between its invocation and response
- real time behavior of method calls must be preserved
- linearizability -> Principle 3.5.1
- every linearizable execution is sequentially consistent but not vice versa

Linearization Points

- to show that a concurrent object is linearizable is to identify, for each method, a linearization point
- for lock based implementations, each method's critical section can serve as its linearization
- for implementations that do not use locking, the linearization point is typically a single step where the method call becomes visible to other method calls

Remarks

- sequential consistency is useful for describing standalone systems
- linearizability is useful for describing components of large systems where components must be implemented and verified independently
- linearizabilty is compositional and nonblocking (and as a result does not limit concurrency)

Formal Definitions

- informally, linearizability means each method call appears to take effect instantaneously at some moment between that method's invocation and return events
- history -> a finite sequence of method invocation and response events
- subhistory -> a subsequence of events within a history
- invocation -> <x.m(a*)A>
  - x is an object
  - m a method name
  - a* a sequence of arguments
  - A is a thread
- response -> <x:t(r*)A>
  - t is OK or an exception
  - r* is a sequence of result values
- step of A -> an event labeled with thread A
- response matches an invocation if they have the same object and thread
- method call (formally) -> in a history H, a method call is a pair consisting of an invocation and the next matching response in H
- pending -> an invocation is pending in H if no matching response follows the invocation
- extension -> an extension of a history H is a history constructed by appending responses to zero or more pending invocations of H
- complete(H) -> the subsequence of H consisting of all matching invocations and responses, i.e. ignore all pending invocations

- a history H is sequential if the first event of H is an invocation, and each invocation, except possibly the last, is immediately followed by a matching response (i.e. method calls do not overlap)
- thread subhistory -> a history of H is the subsequence of all events in H whose thread names are equivalent
- object subhistory -> a history of H is the subsequence of all events in H whose object is equivalent
- equivalent (for histories) -> two histories (H and H') are equivalent if for every thread, A, in H, H|A = H'|A
- well formed (history) -> if each thread subhistory is sequential
- sequential specification (for an object) -> a set of sequential histories for the object
- legal (subhistory) -> a sequential history, H, is legal if each object subhistory is legal for that history
- partial order
  - irreflexive and transitive
  - never true that x -> x
  - when x -> y and y -> z then z -> z
- total order (on X)
  - a partial order such that for all distinct x and y in X either x < y or y < x
- any partial order can be extended to a total order
- Fact 6.3.1 ->
  - if -> is a partial order on X, then there exists a total order "<" on X such that if x -> y, then x < y
- a method call m0 precedes method call m1 in history H if m0 finished before m1 (i.e. m0's response event occurs before m1's invocation event)

Linearizability

- basic idea -> event concurrent history is equivalent, according to the following sense, to some sequential history
  - if one method call precedes another, then the earlier call must have taken effect before the later call
  - by contrast, if two method calls overlap, then their order is ambiguous, and they can be ordered in any convenient way
- Definition 6.3.1 ->
  - a history is linearizable if it has an extension H' and there is a legal sequential history S such that
    - complete(H') is equivalent to S
    - if method call m0 precedes method call m1 in H, then the same is true in S
- S is a one (of potentially many) linearizations in S

Compositional Linearizability

- Theorem 3.6.1 ->
  - H is linearizable if and only if for each object x, Hlx is linearizable
- proof here
- compositionality is important for composing modular concurrent systems
- basing a concurrent system on a noncompositional correctness property means it must rely on a centralized scheduler for all objects or it must satisfy additional constraints placed on objects to ensure that they follow compatible scheduling protocols

The Nonblocking Property

- a pending invocation of a total method is never required to wait for another pending invocation to complete
- Theorem 3.6.2 ->
  - let inv(m) be an invocation of a total method
  - if is a pending invocation in a linearizable history H, then there exists a response such that H * is linearizable
- proof here
- linearizability by itself never forces a thread with a pending invocation of a total method to block
- the nonblocking property does not rule out blocking in situations where it is explicitly intended

Progress Conditions

- blocking -> a delay (possibly unexpected) can prevent others from making progress
- delays can be caused by
  - cache misses
  - page faults
  - preemption
- wait-free -> a call finishes its execution in a finite number of steps
- bounded wait-free -> a bound exists on the number of steps a method call can take
- population-oblivious -> a wait-free method whose performance does not depend on the number of active threads
- nonblocking (progress condition) -> being wait-free
  - an arbitrary and unexpected delay by one thread (possibly one holding a lock) does not necessarily prevent the others from making progress
- wait-free property is a good one because it guarantees that every thread that takes steps makes progress
  - can be inefficient

- lock-free (method) -> a method that guarantees that infinitely often some method call finishes in a finite number of steps
- wait-free is lock-free but lock-free does not mean wait-free
- lock-free algorithms can lead to starvation
    - a fast lock-free algorithm may be more attractive than a slower wait-free algorithm

Dependent Progress Conditions

- wait-free and lock-free nonblocking progress conditions guarantee that the computation as a whole makes progress, independently of how the system schedules threads
- dependent progress conditions -> progress occurs only if the underlying platform (i.e. OS) guarantees
    - every thread eventually leaves every critical section
    - (useful when the OS guarantees that every thread eventually leaves every critical section in a timely manner)
- the deadlock-free and starvation-free properties are dependent progress conditions
- lock-based synchronization can guarantee, at best, dependent progress conditions
- isolated execution -> a method call executes in isolation if no other threads take steps
- Definition 3.7.1 ->
    - obstruction-free -> a method is obstruction-free if, from any point after which it executes in isolation, it finishes in a finite number of steps
- the obstruction-free property
    - is a dependent nonblocking progress condition
    - rules out the use of locks
    - only requires pausing threads that conflict, i.e. they call the same shared object's methods
- back-off mechanism -> a thread that detects a conflict pauses to give an earlier thread time to finish
- the simplest way to exploit an obstruction-free algorithm is to introduce a back-off mechanism

The Java Memory Model

- does not guarantee linearizability or sequential consistency when reading or writing fields of shared objects
    - caused by compiler optimizations that reorder reads/writes
        - register allocation
        - common subexpression elimination
        - redundant read elimination

- satisfies the Fundamental Property of relaxed memory models
    - for a sequentially consistent execution, if certain rules are followed, then every execution of that program in the relaxed memory model will still be sequentially consistent
- all rules are not covered
- double-checked locking -> a once-common idiom in Java that checks for a null instance, enters a synchronized block, and checks for a null instance again before performing the action
- see 3.10 for double-checked locking with a Singleton
- it does not work as expected due to the lack of sequential consistency in Java's memory model because of instruction reordering
- objects reside in a shared memory and each thread has a private working memory that contains cached copies of fields it has read or written
- each JVM can keep cached fields in threads consistent but it is not required to
- synchronization events -> statements that in other contexts imply atomicity or mutual exclusion but within the JVM implies reconciling a thread's working memory with the shared memory
    - update shared memory with cached memory changes
    - invalidate cached values and reread from shared memory
- synchronization events are linearizable
    - they are totally ordered, and all threads agree on that ordering

Locks and Synchronized Blocks

- a thread achieves mutual exclusion
    - by entering a synchronized block or method (which acquires an implicit lock)
    - or by acquiring an explicit lock (e.g. ReentrantLock, FileLock, StampedLock, ReadWriteLock)
- if all accesses to a particular field are protected by the same lock then reads-writes to that field are linearizable

Volatile Fields

- linearizable
- reading is like acquiring a lock
    - working memory is invalidated and volatile field's current value is reread from memory
- writing is like releasing a lock
    - the volatile field is immediately written back to memory
- multiple reads/writes are not atomic
- may still need locks

Final Fields

- final is const and cannot be modified once it has been initialized
- initialized in ctor
- reads/writes to fields are linearizable if either the field is volatile, or the field is protected by a unique lock which is acquired by all readers and writers

Remarks

- different methods for a concurrent object can have different progress conditions
- frequently called time-critical, read method should be wait-free
- infrequent writes can use mutual exclusion

Ch. 4 Foundations of Shared Memory

- foundations of sequential computing in Church-Turing Thesis
  - anything that can be computed can be computed by a Turing Machine or equivalently by Church's Lambda Calculus
  - anything that is unsolvable by a Turing Machine is universally considered unsolvable
- remember -> it is a thesis, not a theorem
- progress of classical theory of sequential computing
  - finite-state automata
  - pushdown automata
  - Turing Machines
- move through shared memory concurrency in the same manner
- no effort to make models efficient

The Space of Registers

- threads communicate by reading/writing shared memory
- good to abstract away and think about communication as happening through shared concurrent objects
- keep in mind
  - safety
    - consistency conditions
  - liveness
    - defined by liveness conditions
- read-write register (or just register) -> an object that encapsulates a value that be observed by a read() method and modified by a write() method (load and store)

- boolean register
- M-valued register -> register for range of of M integer values
- for non-overlapped calls, read and write should work as expected
- for overlapped calls
  - could use a mutex but does not make sense to use another entity to provide the progress property
  - better to use wait-free approach
    - rules out mutual exclusion
    - guarantees independent progress (without OS scheduler)
- require register implementations to be wait-free
- atomic register -> linearizable implementation of the expected sequential register implementation
- important to specify how many readers and writers are expected
- SRSW - single-reader, single-writer
- MRSW - multi-reader, single-writer
- MRMW - multi-reader, multi-writer
- fundamental question ->
  - can any data structure implemented from strongest registers also be implemented from weakest
- useful inter-thread communications must be persistent (i.e. outlives active participation of sender)
- register guarantees have more or less power
  - range of values representable by register
  - number of readers and writers supported
  - degree of consistency they provide
- safe register ->
  - SWMR
  - a read that does not overlap a write returns the value of the most recent write
  - a read that overlaps a write can return any value within the register's allowed ranges of values
- safe is a misleading term because safe registers are unsafe
- regular register
  - MRSW
  - writes are not atomic, e.g.
  - safe, so provides safety as above
  - if a read call overlaps one or more write calls
    - let v0 be the value written by the latest preceding write() call
    - let v1 ... vk be the sequence of values written by overlapping write() calls
    - the read() call may return any of the vi, for any i in the range 0 ... k
- simply stated, the value read may flicker between allowed values

- regular registers are quiescently consistent but not vice versa
- a regular register is a quiescently consistent single-writer sequential register
- useful to rephrase definitions directly in terms of object histories to help reason about algorithms for implementing regular and atomic registers
- write order -> any register implementation (safe, regular, or atomic) defines a total order on the write calls
- for safe and regular registers write order is trivial because only allow one writer at a time
- for atomic registers method calls have a linearization order
- for SRSW or MRSW safe or regular registers the write order is exactly the same as the precedence order
- write call indices -> W0, W1, ... Wi
- read call indices -> R0, R1, ... Ri
- returned value indices -> v0, v1, ... vi
- regular register:
  - 4.1.1 -> it is never the case that Ri -> Wi
  - 4.1.2 -> it is never the case that for some j, Wi -> Wj -> Ri
- atomic register:
  - 4.1.1
  - 4.1.2
  - 4.1.3 -> if Ri -> Rj then i <= j

Register Constructions

MRSW Safe Registers

- code for SafeBooleanMRSWRegister
- Lemma 4.2.1 ->
  - code for SafeBooleanMRSWRegister is a safe MRSW register
- proof presented

A Regular Boolean MRSW Register

- uses safe boolean MRSW register
- only difference arises when newly written value is the same as the old
  - regular returns x
  - safe may return either
- only writes on change
- code for RegBooleanMRSWRegister
- Lemma 4.2.2 ->
  - code for RegBooleanMRSWRegister is a regular boolean MRSW register

- proof here

A Regular M-Valued MRSW Register

- implemented as an array of M boolean registers
- initially set to 0, indicated by setting the 0th bit to true
- write sets xth index to true and then sets all lower indexes to false
- reading method reads locations in ascending order until it encounters a true
- Lemma 4.2.3 ->
  - the read call in the RegMRSWRegister code always returns a value corresponding to a bit 0...M-1 set by some write call
- proof here
- Lemma 4.2.4 ->
  - the RegMRSWRegister code is a regular M-valued MRSW register
- proof here

An Atomic SRSW Register

- uses unbounded timestamps and a regular SRSW register
- must satisfy the conditions 4.1.1, 4.1.2, and 4.1.3
- only way 4.1.3 can be violated is if two reads that overlap the same write read values out-of-order, the first returning vi and that latter returning vj, where j < i
- each read remembers the most recent timestamp/value pair ever read, so that it is available to future reads
  - if a later read then reads an earlier value it ignores that value and uses the remembered latest value
- the writer remembers the latest timestamp it wrote, and tags each newly written value with a later timestamp
- code for StampedValue and AtomicSRSWRegister
- the algorithm requires the ability to read/write a value and a timestamp as a single unit
  - in a language such as C
    - treat both the value and the timestamp as uninterpreted raw bits and use bit-shifting and masks to pack and unpack values
  - easier to use timestamp/value pair
- Lemma 4.2.5 ->
  - the code AtomicSRSWRegister is an atomic SRSW register
- proof here

An Atomic MRSW Register

- uses atomic SRSW registers to construct an MRSW register
- presents intermediate construction that is not multi-reader

- code for AtomicMRSWRegister
- Lemma 4.2.6 ->
  - the code for AtomicMRSWRegister is a MRSW atomic register
- proof here

An Atomic MRMW Register

- uses one atomic MRSW register per thread to construct atomic MRMW register
- to write to the register
  - thread A reads all the array elements, chooses a timestamp higher than any it has observed
  - writes a stamped value to array element A
- to read the register
  - a thread reads all the array elements
  - returns the one with the highest timestamp
- timestamp algorithm is the one used by the Bakery algorithm
- code for AtomicMRMWRegister
- Lemma 4.2.7 ->
  - the code AtomicMRMWRegister is an atomic MRMW register
- shows proof here
- these algorithms are generally not practical in use but showing the construction of one from the other shows how to implement algorithms with strong synchronization properties from ones with weak synchronization properties

Atomic Snapshots

- constructs an instantaneous view of an array of of atomic registers
- array of atomic MRSW registers, one per thread
- methods
  - `void update(T v)`
  - `T[] scan()`
- goal: goal is to construct a wait-free implementation that is equivalent (linearizable) to the sequential specification outlined in code for SeqSnapshot class

An Obstruction-Free Snapshot

- update - wait-free, scan - obstruction-free
- use StampedValue as in the atomic MRSW register
- collect -> non-atomic act of copying the register values one-by-one into an array
- clean double collect -> performing two collects (one after the other) and reads return equivalent values
- result of second collect is a snapshot of the system state immediately after the end of the first collect (know there was an interval between first and second in which no update was made)
- see SimpleSnapshot code

A Wait-Free Snapshot

- to allow for wait-free scan, each update call helps a scan it may interfere with by snapshotting before modifying so that a failing scan can use the snapshot from an interfering update as its own
  - must make sure the snapshot from the update can be linearized within the scan's own execution
- moves (thread moves) -> call to update completes within thread
- if a scanning thread A sees a thread B move twice while it is performing repeated collects, then B executed a complete update() call within the interval of A ’s scan(), so it is correct for A to use B ’s snapshot
- code for StampedSnap and WFSnapshot here

Correctness Arguments

- Lemma 4.3.1 ->
  - if a scanning thread makes a clean double collect, then the values it returns were the values that existed in the registers in some state of the execution
- proof here
- Lemma 4.3.2 ->
  - if a scanning thread A observes changes in another thread B ’s label during two different double collects, then the value of B ’s register read during the last collect was written by an update() call that began after the first of the four collects started
- proof here
- Lemma 4.3.3 ->
  - the values returned by a scan() were in the registers at some state between the call’s invocation and response
- proof here
- Lemma 4.3.4 ->
  - every scan() or update() returns after at most O(n2) reads or writes

- Theorem 4.5.1 ->
  - if c = c1 ... cm is always written from right to left, then a read from left to right obtains a sequence of values c1 sup k1,l1 ... cm sup km,lm with k1 <= l1 <= k2 <= km <= lm
- Theorem 4.5.2 ->
  - let c = c1 ... cm and assume that c0 <= c1 <= ...
  - if i1 <= ... <= im <= i then c1 sup i1 ... cm sup im <= ci
  - if i1 >= ... >= im >= i then c1 sup i1 ... cm sup im >= ci
- Theorem 4.5.3 ->
  - let c = c1 ... cm and assume that c0 <= c1 <= ... and the digits ci are atomic
    a. if c is always written from right to left, then a read from left to right obtains a value c sup k,l <= c sup l
    b. if c is always written from left to right, then a read from right to left obtains a value c sup k,l >= cl

Ch. 5 The Relative Power of Primitive Synchronization Operations

- outlines core set of primitive atomic instructions that should be seen as those necessary to solve important synchronization problems
- synchronization primitives -> object whose exported methods are primitive synchronization instructions
- consensus problem -> a problem that requires agreement among a number of processes (or agents) for a single data value
  - some of the processes (agents) may fail or be unreliable in other ways, so consensus protocols must be fault tolerant or resilient
  - processes must somehow put forth their candidate values, communicate with one another, and agree on a single consensus value
- consensus number -> maximum number of threads for which objects of the class can solve the consensus problem

Consensus Numbers

- consensus object has one method decide where each thread calls decide with an input v at most once

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

- a concurrent consensus object is linearizable to a sequential consensus object in which the thread whose value was chosen completes decide first

- binary consensus -> consensus problems where all inputs are either 0 or 1
- focused on wait-free concurrent solutions to consensus == consensus protocol
- Definition 5.1.1 ->
  - a class, C, solves n-thread consensus if there exists a consensus protocol using any number of objects of class C and any number of atomic registers
- Definition 5.1.2 ->
  - the consensus number of a class C is the largest n for which that class colves n-thread consensus
  - if no largest n exists, the consensus number of the class is infinite
- Corollary 5.1.1 ->
  - supposition: an object of class C can be implemented from one or more objects of class D together with some number of atomic registers
  - if class C solves n-consensus, then so does class D

States and Valence

- simplest case
  - inputs: 0 or 1
  - 2 threads
  - move -> method call to shared object
  - protocol state -> combined state of threads and shared objects
  - initial state -> a protocol state before any thread has moved
  - final state -> a protocol state after all threads have finished
  - decision value -> the value decided by all threads in a specific final state
  - each thread makes moves until it decides on a value
- wait-free protocol's set of possible states forms a tree
  - each node represents a protocol state
  - each edge represents a move by some thread

We refer to s′ as a successor state to s.

- successor state ->
  - an edge for A from node s to node s' means that if A moves in protocol state s, then the new protocol state is s' (s' is a successor state to s)
- bivalent (protocol state) -> the decision value is not yet fixed
  - there is some execution stating from that state in which threads decide 0 and one where threads decide 1
- univalent (protocol state) -> outcome is fixed
  - every execution starting from that state decides the same value
- 1-valent (0-valent, protocol state) -> univalent and decision value is 1 (or 0)
- Lemma 5.1.1 ->
  - every 2-thread consensus protocol has a bivalent initial state

- proof here
- Lemma 5.1.2 ->
  - every n-thread consensus protocol has a bivalent initial state
- proof here
- critical (protocol state) ->
  - bivalent
  - if any thread moves, the protocol state becomes univalent
- proof here

Atomic Registers

- cannot solve consensus using atomic registers
- Theorem 5.2.1 ->
  - atomic registers have consensus number 1
- proof here
- Corollary 5.2.1 ->
  - it is impossible to construct a wait-free implementation of any object with consensus number greater than 1 using atomic registers
- this corollary explains why hardware must provide primitive synchronization operations other loads/stores (reads/writes) to be able to implement lock free concurrent data structures on modern multiprocessors

Consensus Protocols

- synchronization objects that uses array of atomic registers in relation to a decide method which proposes an input value and executes a sequence of steps to decide on one of the proposed values

FIFO Queues

- is a wait-free implementation of a FIFO queue that supports multiple enqueuers and dequeuers possible?
- is a wait-free implementation of a two-dequeuer FIFO queue using atomic registers possible?
- Theorem 5.4.1 ->
  - the two-dequeuer FIFO queue class has consensus number at least 2
- proof here
- code for QueueConsensus that is wait-free
- can be used for
  - stack

    - priority queue
    - list
    - set
- Corollary 5.4.1 ->
    - it is impossible to construct a wait-free implementation of a queue, stack, priority queue, set, or list from a set of atomic registers
- Theorem 5.4.1 ->
    - FIFO queues have consensus number 2
- proof here
    - can be applied to show sets, stacks, double-ended queues, and priority queues have consensus number 2

Multiple Assignment Objects

- multiple assignment or m, n-assignment -> for n >= m > 1, given an object with n fields (sometimes an n-element array)
    - atomically assigns m values (vi, i contained in 0, ..., m - 1) and m index values (ij, j contained in 0, ..., m - 1, ij contained in 0, ..., n - 1) via the assign method (atomically assigns vj to array index ij)
    - read takes an index and returns the ith array element
- code for lock-based implementation of (2,3) assignment
- Theorem 5.5.1 ->
    - there is no wait-free implementation of an (m, n)-assignment object by atomic registers for any n > m > 1
- proof here
- code for MultiConsensus here
- Theorem 5.5.2 ->
    - atomic (n, n*(n + 1)/2)-register assignment for n > 1 has a consensus number of at least n
- proof here
- multiple assignment solves consensus for any m > n > 1 threads while its dual structures and atomic snapshots have consensus number at mose one
- shows that writing atomically to multiple memory locations requires more computational power than reading atomically

Read-Modify-Write Operations

- many, if not all, of the classical synchronization operations provided by multiprocessors in hardware can be expressed as read–modify–write (RMW) operations, or, as they are called in their object form, read–modify–write registers

- Theorem 5.6.1 ->
  - any nontrivial RMW register has a consensus number of at least 2
- proof here
- code for RMWConsensus here
- Corollary 5.6.1 ->
  - it is impossible to construct a wait-free implementation of any nontrivial RMW method from atomic registers for two or more threads

Common2 RMW Operations

- RMW registers that correspond to many of the common synchronization primitives provided by processors in the late 20th century
- Common2 registers have consensus number exactly 2 (limited synchronization power)
- not common any more
- Definition 5.7.1 ->
  - a set of functions, F, belongs to Common2 if for all values v and all fi and fj in F, either
    - fi and fj are commutative -> fi(fj(v)) == fj(fi(v))
    - one function overwrites the other -> fi(fj(v)) == fi(v) or fj(fi(v)) == fj(v)
- Definition 5.7.2 ->
  - an RMW register belongs to Common2 if its set of functions, F, belongs to Common2
- Theorem 5.7.1 ->
  - any RMW register in Common2 has consensus number (exactly) 2
- proof here

The compareAndSet() Operation

- related to instruction provided by many modern architectures (CMPXCHG on Intel)
- aka compare-and-swap
- expected and update args
- if current == expected then current = update, returns bool indicated whether value changed
- Theorem 5.8.1 ->
  - a register providing compareAndSet() and get() methods has an infinite consensus number
- code for CASConsensus here
- Corollary 5.8.1 ->
  - a register providing only compareAndSet() has an infinite consensus number

Ch. 6 Universality of Consensus

Introduction

- used proofs of form "no wait-free implementation of X by Y"
- derived hierarchies in which no object at one level can implement one at a higher level
- consensus number -> max number of threads for which an object can solve the consensus problem
- impossible to construct a wait-free implementation of an object with consensus number < n in a system of n or more threads

```
consensus number    |   object
------------------------------------------------------------------------------
1                   |   atomic registers
2                   |   getAndSet(), getAndAdd(), Queue, Stack
...                 |   ...
m                   |   (m, m(m + 1)/2)-register assignment
...                 |   ...
inf                 |   memory-to-memory move, compareAndSet(), load-linked/store
------------------------------------------------------------------------------
```

- chapter shows that there exist classes of objects that are universal
  - given sufficiently many of them, one can construct a wait-free linearizable implementation of any concurrent object
- programming language or architecture is powerful enough to support arbitrary wait-free synchronization if and only if it provides objects of a universal class as primitives
- universal construction -> use of consensus objects used to implement any concurrent object

Universality

- universal (class) -> class C is universal if can construct a wait-free implementation of any object from some number of objects of C and some number of read-write registers
- use an unlimited number of of read-write registers and consensus objects (without describing how to recycle memory)
- start with lock-free and move to wait-free

A Lock-Free Universal Construction

```
public interface SeqObject {
    public abstract Response apply(Invocation invoc);
}
```

- generic definition for a sequential object based on invocation-response of ch/ 3
- code in 6.3 and 6.4 (Node class and LFUniversal class) shows a construction that transforms any sequential object into a lock-free linearizable concurrent object
- assumes sequential objects are deterministic
    - application of a method to an object in a particular start state always only returns one response and results in one particular object state
- can represent any object as a combination of a sequential object in its initial state and a log
    - log is a linked list of nodes representing the sequence of method calls applied to an object (i.e. the sequence of state transitions)
    - thread applies method call by appending to head of list
    - traverses from tail to head applying transitions to a private copy of the object
    - thread returns the result of applying its own operation to the private object
    - note: only final call is mutable, all previous operations and start state are immutable
    - make concurrent by allowing a thread to construct a node containing its call
    - then run an n-thread consensus protocol to determine the node to append operation to log next
    - consensus objects can only be used once
    - to locate head of the log, use a construction similar to Bakery algorithm
        - n-entry array, `head[]` , where `head[i]` is the last node in the list that thread i has observed
        - initially all entries refer to tail sentinel node
        - head is node with max sequence number (i.e. timestamp) within the array
        - completes appending in finite number of steps but a thread can fail to add its node if other threads succeed in appending theirs so starvation is possible so the construction is only lock-free

A Wait-Free Universal Construction

- must guarantee that apply() call always completes in finite number of steps so no thread starves
- to guarantee this, progressing threads must help non-progressing threads to complete
- code for Universal class here

- for apply
  - thread announces its new node (by adding to announce array)
  - ensure that other thread will append node on its behalf if current thread does not succeed
  - proceeds as in lock-free universal construction and retries main loop until its node is appended
  - difference:
    - a thread first checks to see if there is a node that needs help ahead of it in the announce array
    - attempts to help threads ahead of it in increasing order by using timestamp modulo width (size) n of the announce array
  - removing the helping step would mean each thread could be overtaken an arbitrary number of times
  - must make sure no node is appended twice
    - avoided by order of read max in head array and read sequence (timestamp) in announce array
- linearizable because no node is added twice
- proof of wait-free requires showing that any announced node will eventually be appended to head and that that allows the announcing thread to complete
- auxiliary variable (ghost variable) -> one that does not appear explicitly in code, does not alter the program's behavior in any way, but aids in reasoning about the behavior of the algorithm
- use the following aux vars:
  - concur(A) is the set of nodes that have been stored in the `head[]` array since thread A's last announcement
  - start(A) is the sequence number of `max(head[])` when thread A last announced
- shows wait-free Universal code with aux vars for clarity
- Lemma 6.4.1 ->
  - for all threads A, the following claim is always true
    - `|concur(A)| > n => announce[A] contained in head[]`
- proof here
- Lemma 6.4.2 ->
  - the following property always holds
    - `max(head[]) >= start(A)`
- proof here
- Lemma 6.4.3 ->
  - the following is a loop invariant for loop in wait-free universal construction (i.e. holds for each iteration)
    - `max(head[A], head[j] ,..., head[n] - 1) >= start(A)`
    - j is loop index
- proof here

- Lemma 6.4.4 ->
  - the following assertion holds before line 10 in code
    - `head[A].seq >= start(A)`
- proof here
- Lemma 6.4.5 ->
  - the following property always holds
    - `|concur(A)| >= head[A}.seq - start(A) >= 0`
- proof here
- Theorem 6.4.1 ->
  - the wait-free universal construction algorithm is correct and wait-free
- proof here

Ch. 7 Spin Locks and Contention

- multiprocessor programming necessitates an understanding of the underlying architecture
- mutual exclusion protocols always require determining what to do if the lock cannot be acquired
- spin lock -> a lock that is repeatedly retried if locking is not possible until it is possible (spinning or busy waiting)
  - only possible on multiprocessors
  - best to use for shorter waitrs
- blocking -> suspending the thread attempting to acquire the lock and signaling the OS scheduler to schedule another thread on the suspended thread's processor
  - only good for longer delays

Welcome to the Real World

- use Java java.util.concurrent.locks package and implement a lock object called mutex
- details issues with locks, memory buffers, and instruction reordering and the weak memory consistency guarantees caused by these
- memory barrier (memory fence) -> used to force outstanding operations (for things like outstanding write buffering) to take effect
- barriers are expensive and up to programmer to know where to add them

Test-And-Set Locks

- shows code for TASLock here
- shows code for TTASLock here
- both guarantee deadlock free mutual exclusion
- TTASLock shows better performance

- bottlenecks and variance from ideal lock performance is generally due to the delays caused by memory access and the relationship to cache accesses
- when a processor reads from an address in memory
  - first checks whether that address and its contents are present in its cache
  - if so
    - the processor has a cache hit, and can load the value immediately
  - if not,
    - the processor has a cache miss, and must find the data either in the memory, or in another processor's cache
    - processor then broadcasts the address on the bus
    - other processors snoop on the bus
    - if one processor has that address in its cache, then it responds by broadcasting the address and value
    - if no processor has that address, then the memory itself responds with the value at that address

TAS-Based Spin Locks Revisited

- on a shared-bus architecture, each getAndSet() call is broadcast on the bus, delaying all threads, and resulting in a cache miss almost every time
- when the lock is released, it may be delayed due to spinning threads
- TTASLock re-reads are cache hits so does not slow down other threads and lock release is not delayed by spinning locks
- local spinning -> threads repeatedly reread cached values instead of repeatedly using the bus

Exponential Backoff

- contention -> when multiple threads try to acquire a lock at the same time
- high contention -> when many threads try to acquire a lock at the same time
- low contention -> when only a few threads try to acquire a lock at the same time
- key observation ->
  - in the TTASLock algorithm, if some other thread acquires the lock between the first and second check then there is high contention for the check
- good rule of thumb ->
  - larger number of retries for a lock, the higher the contention, longer the thread should back off
  - when thread sees lock is free but fails to acquire, back off before retrying
  - each time thread tries and fails, double back off time
- code for Backoff here

- code for BackoffLock here

Queue Locks

- more complicated scalable spin locks than backoff locks but more portable
- problems with BackoffLock
  - cache-coherence traffic
    - all threads spin on the same shared location causing cache-coherence traffic on every successful lock access
    - critical section underutilization
      - threads delay longer than necessary, causing critical section to be underutilized
- with threads in a queue, each thread can check for its turn if predecessor has returned
- provides first-come-first-served fairness (similar to Bakery)

Array-Based Locks

- uses atomic int tail field
- to acquire lock, each thread increments tail field
- maintains boolean `flag[]` and if `flag[j]` is true, then thread with slot j has permission to acquire the lock
- threads spin until flag at their slot is true and releases by setting it to false
- code for ALock here
- each thread's slotIndex is thread-local
  - not stored in shared memory
  - no need for synchronization
  - do not generate cache coherence traffic
- contention may still occur because of false sharing
- false sharing -> occurs when adjacent data items (such as array elements) share a single cache line
- in languages like C and C++ can avoid false sharing by padding adjacent elements to the size of a cache line

The CLH Queue Lock

- impovements
  - reduces invalidations to a minimum
  - minimizes interval between when a lock is freed by one thread and when it is acquired by another
  - guarantees no starvation
  - provides first-come-first-serve fairness

- ALock is not space efficient
- CLHLock uses nodes with a boolean locked field
- lock is represented by a virtual linked list because list is implicit (threads refer to link through pred field)
- code for CLHLock here
- TODO: more details of algorithm
- advantages:
  - releasing lock only invalidates successor's cache
  - requires less space
  - does not require knowledge of the number of threads that might access the lock
  - provides first-come-first-served fairness
- disavantages:
  - performs poorly on cache-less NUMA architectures

The MCS Queue Lock

- uses explicit linked list
- code for MCSLock here
- advantages:
  - shared with CLHLock
  - better suited to cache-less NUMA architectures
- disadvantages:
  - releasing a lock requires spinning
  - requires more reads, writes, and compareAndSet() calls than CLHLock

A Queue Lock with Timeouts

- tryLock allows specifying a timeout or max duration to wait to acquire lock and returns bool specifying success of acquiring lock
- uses virtual linked list of nodes and each thread spins on predecessor's node
- when a thread times out it marks its node as abandoned
- its successor in queue (if it exists) observes the node on which it has been spinning has been abandoned and spins instead on abandoned node's predecessor
- code for TOLock here
- advantages:
  - shares many of CLHLock's advantages
    - local spinning on a cached location
    - quick detection that the lock is free
  - wait-free timeout of BackoffLock

- disadvantages:
  - need to allocate a new node per lock access
  - a thread spinning on the lock may have to move through a chain of timed-out nodes before accessing the critical section

A Composite Lock

- spin-lock algorithms impose trade-offs
  - queue locks provide
    - first-come-first-served fairness
    - fast lock release
    - low contention
  - queue locks require
    - nontrivial protocols for recycling abandoned nodes
  - backoff locks
    - support trivial timeout protocols
  - however, backoff locks
    - are inherently not scalable
    - may have slow lock release if timeout parameters are not well-tuned
- in queue lock only threads at front need to perform lock handoffs
- can keep small number of threads in queue and have rest use exponential backoff
- code for CompositeLock here
- TODO: more description of code
- advantages:
  - reduce contention because when they back off, access different locations
  - hand-off is fast
  - abandoning a lock request is trivial for threads in backoff stage
  - abandoning a lock request is straightforward for threads that have acquired queue nodes
  - requires only O(L) space in worst case for L locks and n threads

A Fast-Path Composite Lock

- the previously described composite lock does not perform well in the absence of concurrency
- fast path -> shortcut through a complex algorithm used for a single thread in the absence of concurrency
- add a an extra state to distinguish between a lock held by an ordinary thread and a lock held by a fast-path thread
- when lock is free, thread attempts fast-path and if it fails, follows previous algorithm by enqueueing

- code for CompositeFastPathLock here
- uses FAST_PATH flag, fastPathLock private method, and calls to super's method (inherits from CompositeLock)

Hierarchical Locks

- many cache-coherent architectures organize processors in clusters
- communication within a cluster is significantly faster
- want to design locks to be aware to these difference
- hierarchical (locks) -> take into account the architecture's memory hierarchy and access costs
- can have many hierarchical levels but keep it to 2 for simplicity

A Hierarchical Backoff Lock

- TTASLock can be easily adapted to exploit clustering
- reduce backoff time for locks that are in the same cluster
- code for HBOLock here
- disadvantage:
  - may be too successful in exploiting locality causing threads in same cluster to exchange lock causing threads in other clusters to starve

A Hierarchical CLH Queue Lock

- need to favor threads in same cluster and still ensure some degree of fairness
- uses set of local queues, one per cluster, and a single global queue
- code for HCLHLock here
- TODO: more code description here
- advantages:
  - favors sequences of local threads
  - use of implicit references minimizes cache misses and threads spin on locally cached copies of their successor's node state

One Lock To Rule Them All

- no single algorithm is ideal for all applications

Ch. 8 Monitors and Blocking Synchronization

Introduction

- monitor -> structured method for combining synchronization and data (within an abstraction)
- allow queues to manage own synchronization for threads

Monitor Locks and Conditions

- hold, acquire, release
- if a thread cannot immediately acquire a lock, it can either spin, repeatedly testing whether the desired event has happened, or it can block, giving up the processor for a while to allow another thread to run
- often makes sense to combine spinning and blocking
- spinning only works on multiprocessors, while blocking works on uniprocessors also
- general Lock API
  - lock -> blocks caller until it acquires the lock
  - lockInterruptibly -> throws an exception if the thread is interrupted while lock call is waiting
  - unlock -> releases lock
  - newCondition -> factory that creates a Condition object
  - tryLock -> acquires the lock if free and returns boolean indicating whether lock was acquired (can be configured with timeout)

Conditions

- associated with a lock and, when created, if the await method is called, causes the calling thread to release the lock and suspend
- when calling thread awakens, it can reacquire the lock (by competing if necessary)
- with interruptible threads, if thread is interrupted during an await call, an exception is thrown
- work with Conditions has a protocol that should be followed
- code for Condition interface here
- reentrant -> thread holding a lock can acquire it again without blocking
- code for LockedQueue using locks and conditions
- two condition objects
  - notEmpty -> notifies waiting dequeuers when the queue goes from empty to nonempty
  - notFull -> notifies waiting enqueuers when the queue goes from full to not full
- using two conditions is more efficient

- monitor -> object demonstrated in LockedQueue that combines methods, mutual exclusion locks, and condition objects to facilitate concurrent access

The Lost-Wakeup Problem

- lost wakeups -> one or more threads wait forever without realizing that the condition for which they are waiting has become true
- Condition objects are vulnerable to lost-wakeups
- no solution for finding all cases that can cause lost-wakeups
- methods to minimize vulnerability to lost-wakeups
  - always signal all processes waiting on a condition, not just one
  - specify a timeout when waiting
- each imposes a small performance penalty which can be viewed as negligible when considering the possibility of a lost-wakeups
- synchronized methods and the wait, notify, and notifyAll methods provide the functionality of monitors

Readers-Writers Locks

- readers -> shared object method calls that return information about the object's state without modifying the object
- writers -> shared object method calls that modify the object
- most shared object methods are readers while only a few are writers
- readers do not need to synchronize whereas writers must lock readers and writers
- readers-writers lock -> allows multiple readers or a single writer to enter the critical section concurrently
- interface
  - Lock readLock
    - no thread can acquire the read lock while any thread holds the write lock
    - multiple threads may hold the read lock at the same time
  - Lock writeLock
    - no thread can acquire the write lock while any thread holds either the write lock or the read lock

Simple Readers-Writers Lock

- code for SimpleReadWriteLock
- keeps count of readers that have acquired the lock and bool indicating if a writer has acquired the lock
- code for ReadLock
- code for WriteLock

Fair Readers-Writers Lock

- many readers can cause writer to be locked out for an excessively long time
- when writer calls writeLock method, block readers from calling readLock which causes readers to eventually drop to 0 and allows writer to acquire lock
- code for FifoReadWriteLock (and ReadLock/WriteLock)
- to notify writer
    - when a writer tries to acquire the write lock, acquires the lock object
    - a releasing reader also acquires lock object and calls lock's signal method if all readers have released

Our Own Reentrant Lock

- using locks described in ch2/7, a thread that attempts to reacquire a lock it already holds will deadlock with itself (arises in nesting/recursion)
- reentrant lock -> a lock that can acquired many times by the same thread
- code for SimpleReentrantLock class
- stores thread ID of the last thread to acquire the lock and keeps track of holdCount to indicate when a lock is free

Semaphores

- semaphore -> generalization of a mutual exclusion lock that has a capacity property that indicates the number of threads that can enter a Semaphore's critical section at any given time
- methods
    - acquire -> suspends until capacity is available
    - release
- code for Semaphore here

Ch. 9 Linked Lists

- coarse-grained synchronization -> give each data structure a scalable lock and ensure each method call acquires and releases the lock
- works well for low levels of concurrency but does not scale
- fine-grained synchronization -> split an object into independently synchronized components and acquire and release locks based on component access
- optimistic synchronization -> in trees, lists, etc, scan for desired object without locking and then check for modification in the interim between scan start and scan success after locking (only worthwhile if scan succeeds more often than not)

- lazy synchronization -> can split all modifying tasks into logical steps and delay the more work intensive component (i.e. by logically performing action by tagging or flagging and then physically performing action, like add, remove, etc, later)
- nonblocking synchronization -> can sometimes remove locks entirely by relying on built-in atomic operations for synchronization
- use set interface for linked list
    - add
    - remove
    - contains

List-Based Sets

- Node class as expected
- Set class uses two sentinel nodes, head and tail, as dummy nodes with min and max integer as values

Concurrent Reasoning

- skill that can be learned
- understand invariants
    - conditions that hold on creation
    - conditions that must be maintained on each operation
- freedom from interference -> assurance that only modifications are centralized to known locations (i.e. no unexpected side effects)
- for set, even requires freedom from interference for nodes that have been removed because a node can be unlinked by a thread while others are traversing it
- abstract value -> the higher level organization/representation achieved by using an abstraction such as a class
- concrete representation -> the means by which the higher level abstraction is achieved
- understanding abstract value vs. concrete representation is essential for reasoning
- representation invariant -> the characterization of which representations make sense as abstract values
- set invariants
    - sentinels are neither added nor removed
    - nodes are sorted by keys
    - keys are unique
- abstraction map ->
- safety property to use -> linearizability
    - identify linearization point -> point where method call takes effect (single atomic step)
- different list algorithms with different guarantees

- summary
  - wait-free -> call finishes in finite number of steps
  - lock-free -> some call always finishes in finite number of steps (not free from starvation)

Coarse-Grained Synchronization

- simpler and more obviously correct
- code for CoarseList here
- satisfies same progress condition as the lock used
  - if starvation-free, list is starvation-free
- list implementation works well for low contention
- does not work well for high contention even if the lock performs well under contention this list implementation will not perform well

Fine-Grained Synchronization

- as a thread traverses the list, locks each node as it visits and releases some time later
- permits pipelined concurrent traversal
- lock coupling -> uses a hand-over-hand order for acquiring locks (i.e. acquire lock for a node only while holding the lock for the predecessor)
- no obvious way to implement lock coupling using Java's synchronized blocks
- code for FineList add and remove here
- important that all methods acquire locks in the same order to guarantee progress
- the linearization point for an add(a) call depends on whether the call was successful
  - a successful call (a absent) is linearized when the node with the next higher key is locked (either Line 7 or 13)
- for remove, an unsuccessful call (a present) is linearized when the predecessor node is locked (Lines 36 or 42)
- for remove, An unsuccessful call (a absent) is linearized when the node containing the next higher key is locked (Lines 36 or 42)
- FineList is starvation-free

Optimistic Synchronization

- fine grained locking is an improvement but may still impose a long sequence of lock acquisitions and releases
- optimistic synchronization -> assume locks are not needed until a target is found then check correctness and retry in case of error
- code for OptimisticList here

- traversing lock based data structures without locks requires careful consideration
  - use some form of validation
  - guarantee freedom from interference
- path from nodes can change, necessitates need for validation to ensure reachability
- OptimisticList
  - not starvation-free, but starvation is rare
  - individual nodes are starvation-free
  - works best if the cost of traversing the list twice without locking is significantly less than the cost of traversing the list once with locking

Lazy Synchronization

- refinements to OptimisticLock
  - make contains() wait-free
  - in add() and remove(), while blocking, traverse list only once in absence of contention
  - use boolean marked field and maintain invariant that every unmarked node is reachable
  - remove() node is lazy (marks for removal then removes later)
- code for LazyList here
- advantages
  - can separate logical steps that are non-modifying from physical steps that are modifying
  - to gain more benefit, physical steps can be delayed and batched
- disadvantages
  - add and remove calls are blocking and may result in delays

Non-Blocking Synchronization

- can extend lazy marking to remove locks altogether
- cannot use compare and set to change the next field
- main need is the ability to ensure a node is not modified after it has logically been marked for removal
  - treat next and marked fields as a single atomic unit, if marked is true, attempts to update next will fail
- Pragma 9.8.1 ->
  - can use AtomicMarkableReference from java.util.concurrent.atomic
- code for LockFreeList here

Discussion

- gradually reduced frequency and granularity of locking in list-based locks
- optimistic and lazy approaches are often used in the path to lock-free data structures

- LockFreeList guarantees progress in spite of arbitrary delays
- cost of progress
  - atomic operations on a reference and a boolean add performance cost
  - add and remove must concurrently remove marked nodes (which adds contention and can lead to restarting traversal)
- LazyList does not guarantee progress but does not require atomic markable field or traversal to clean-up logically removed nodes

Ch. 10 Concurrent Queues and the ABA Problem

- pool -> similar to a concurrent multi-set, i.e. does not necessarily provide the the contains() method and allows repeated elements
- appear often in concurrent systems
  - producer-consumer
    - jobs
    - input (keystrokes)
    - purchases
    - packets
- bursty -> suddenly and briefly producing items faster than consumers can consume them
- can place buffer between producers and consumers to handle bursty producers
- pools can be used as these buffers
- pool varieties
  - bounded vs. unbounded -> limited by capacity or unlimited capacity
  - total (method) -> methods do not wait for certain conditions to become true to complete
  - partial (method) -> method calls may wait for conditions before completing
  - synchronous -> waits for a condition (another method call's interval to overlap) to complete
  - fairness guarantees -> LIFO, FIFO, or another policy

Queues

- concurrent queues are linearizable to a sequential queue and are pools where enqueue = set and dequeue = get

A Bounded Partial Queue

- if queue is neither empty or full, enqueue and dequeue should be able to operate without interference

- code for BoundedQueue
- similar to linked list code but sentinel node is replaced
- uses an enqueue lock and a dequeue lock and two condition variables
- must carefully avoid lost-wakeup
    - enqueuer encounters full queue in two steps (checks size and then waiits on not full condition variable)
- the queue's actual head and tail fields are not always the same as its logical head and tail fields
- disadvantage
    - enqueue and dequeue calls interfere with each other and could cause a bottleneck
- can split counter into two

An Unbounded Total Queue

- simpler than bounded algorithm
- code for UnboundedQueue
- cannot deadlock because each method acquires only one lock

An Unbounded Lock-Free Queue

- code for LockFreeQueue
- prevents starvation by having faster calls assist slower calls
- enqueue method is lazy (completes in two steps) so each other method must be able to handle an incomplete enqueue call
- subtle issue with dequeue
    - must make sure tail is not left referring to sentinel node before advancing head
    - test for head == tail
- lock-free because some method call always completes and each call checks for an incomplete enqueue
- lock-free queues significantly outperform blocking ones

Memory Reclamation and the ABA Problem

- can use a pool to handle custom memory reclamation to avoid garbage collector or expensive construction/destruction
- lock-free queue will not work for most logical method of recycling using a pool (see figure 10.14 for details)

- known as the ABA problem
    - reference about to modified using compare-and-set is changed from a to b by another thread and then back to a again a resulting in a successful call even though it shouldn't
- the ABA problem occurs when multiple threads (or processes) accessing shared data interleave
    - process P1 reads value A from shared memory
    - P1 is preempted, allowing process P2 to run
    - P2 modifies the shared memory value A to value B and back to A before preemption
    - P1 begins execution again, sees that the shared memory value has not changed and continues
- can tag each atomic reference with a unique stamp
- (can use intermediate nodes)
- (can use deferred reclamation)
- Pragma 10.6.1 ->
    - an AtomicStampedReference object encapsulates both a reference to an object of Type T and an integer stamp (which can be atomically updated together or separately)
- ABA problem occurs in many concurrent scenarios

A Naive Synchronous Queue

- multiple producers and multiple consumers, producer blocks until a consumer consumes the produced data
- code for SynchronousQueue
- uses monitors
- has a high synchronization cost
    - where one thread might wake another, enqueuers and dequeuers wake up all threads
    - number of wakeups is quadratic in number of waiting threads
    - can use condition objects but still necessary to block on every call

Dual Data Structures

- alternative synchronous queue that splits enqueue and dequeue calls
    - dequeue
        - add reservation object to queue to indicate dequeuer is waiting for an enqueuer to rendezvous and spins

        - when enqueuer discovers reservation, fulfills reservation by enqueueing and notifying the dequeuer
    - enqueue
        - wait for rendezvous partner by adding reservation and spinning on reservation flag
- dual data structure -> data structure with methods that take effect in two stages, reservation and fulfillment
- can use locally cached flag for spinning
- linearizable
- code for SynchronousDualQueue

Ch. 11 Concurrent Stacks and Elimination

- stacks are not inherently sequential

An Unbounded Lock-Free Stack

- use a linked list with a top field
- code for LockFreeStack
- lock free because if a thread fails on a push or a pop it means there are infinitely many successful modifications to top from various threads

Elimination

- LockFreeStack performs poorly, not because of contention, but because of sequential bottlenecking
- backoff can reduce contention but it doesn't reduce sequential bottlenecking
- can try to cause concurrent pairs of pushes and pops to cancel out (elimination)

The Elimination Backoff Stack

- must avoid allowing a thread to try to coordinate for elimination with more than one thread
- use an EliminationArray
- exchangers -> objects that allow exactly two threads to rendezvous for exchange
- use spinning rather than blocking

A Lock-Free Exchanger

- code for LockFreeExchanger
- uses states of EMPTY, BUSY, and WAITING

- lock-free because overlapping exchange() calls with sufficient time to exchange will fail only if other exchanges are repeatedly succeeding

The Elimination Array

- max capacity for array
- thread picks array entry at random, calls exchange, and provides its own exchange value
- uses visit() method with timeout
- code for EliminationArray
- code for EliminationBackoffStack
- EliminationBackoffStack is a linearizable stack
- performs like LockFreeStack at low loads
- has potential to scale over LockFreeStack
- reduces contention over LockFreeStack

Ch. 12 Counting, Sorting, and Distributed Coordination

- measuring concurrent data structures requires measuring latency and throughput

Shared Counting

- stacks and queues can be viewed as pools that provide different semantics as additional fairness guarantees
- course-grained locking for pools leads to sequential bottlenecks and hot spots
- can use cyclic array with two counters for put and get location
- removes one bottleneck
- two bottlenecks are better than one!
    - need to avoid memory contention
    - need to achieve real parallelism

Software Combining

- combining ->
    - achieved through binary tree (node-based) with counter in root
    - to increment, thread ascends from leaf to root
    - if a thread encounters another at a node, combine increments, and one active thread continues forward
    - when active thread increments, all related passive threads are notified that the increment has been completed

- disadvantage
  - latency increases has increment increases
- advantage
  - provides much better throughput
  - can be adapted to apply any commutative function to a value
- consider solutions that improve throughput over concern for latency

Overview

- complex implementation
- split into two classes for clarity
- short term synchronization provided by Nodes
- long term synchronization provided by boolean lock field to exclude threads from a node
- each tree node has a combining status
  - FIRST -> one active thread has visited and will return to check for another passive thread to have visited
  - SECOND -> a passive second thread has visited and stored a value to be combined but the combination has not yet occurred
  - RESULT -> both thread's operations have been combined and completed and result is stored in node
  - IDLE -> not in use
  - ROOT -> node is root and must be treated specially
- code for CombiningTree
- Pragma 12.3.1 ->
  - always provide an arm for each enumeration value (program defensively)
- code for Node class' combine(), op(), and distribute() methodd

An Extended Example

- TODO: more here

Ch 16 Futures, Scheduling, and Work Distribution

Introduction

- demonstrate decomposition of algorithms/computations into parallel components (tasks)
- consider matrix multiplication
  - Worker class creates a thread to perform the multiplication for a single cell

- creating, scheduling, and destroying threads takes substantial computation and approach above becomes inefficient for large matrices
- more effective to use thread pooling and assign tasks to threads
- thread pools also allow the pool to work in conjunction with hardware limitations on the number of threads, improving performance hits that also occur when an application attempts to create too many threads
- executor service -> thread pool (java.util.ExecutorService)
    - submit task
    - wait for submitted task
    - task cancellation
- tasks are represented by Runnable objects combined with Callable objects
- submitting a task to an executor service returns a Future
- promise -> contract provided by Future guaranteeing to return the result of an asynchronous computation
- for Java, submitting a Runnable returns a different type of Future than a Callable
    - can call get() to block until Runnable based future returns
- all submissions to executor service are advisory, indicating that it may execute tasks in parallel but not being given any guarantee that they will be executed in in parallel
- code for Matrix, MatrixTask, MulTask, and FibTask

Analyzing Parallelism

- multithreaded computations can be viewed as a DAG
    - links predecessor to successor
    - future creates two successors
        - node that is successor in same thread
        - node that is successor as first task in future's computation
    - link between child to parent when value is returned (i.e. get() call)
- some computations are inherently more parallel than others
- latency -> minimum time required to execute a parallel program from start to finish on a set of dedicated processors
- critical-path length -> given that a program has a set number of steps to execute, the number of steps to execute the program on an infinite number of processors
- linear speedup
- parellelism -> maximum possible speedup or average amount of work available at each step along the critical path

Realistic Multiprocessor Scheduling

- practical multiprocessors run many jobs and have to balance between jobs

- applications typically have no control over scheduling of user-level threads
- three-level model
  - application decomposes parallel computations into short-lived tasks
  - user-level scheduler maps tasks to a finite number of threads
  - kernel maps threads onto hardware processors (depending on availability)
- greedy -> a schedule that executes as many ready nodes on as many available processors as possible
- Theorem 16.3.1 ->
  - consider a multithreaded program with work T1, critical-path length T inf, and P user-level threads
  - any greedy execution has length T which is at most T1/PA + Tinf(P - 1)/PA
- proof here

Work Distribution

- multithreaded computations create and destroy tasks dynamically, sometimes unpredictably
- work distribution -> algorithm used to assign ready tasks to idle threads as efficiently as possible
- work dealing -> overloaded thread offloads tasks to less overloaded threads
- flaw is that overloaded threads may waste extra work searching for less overloaded threads
- work stealing -> a free thread tries to steal work from other busy threads
- avoids flaw above

Work Stealing

- use double-ended queue for each threads task pool
- when queue is empty, thread randomly chooses another thread to check for work to steal from
- code for WorkStealingThread
- to avoid endless attempts to steal, use a termination-detecting barrier (ch 17)

Yielding and Multiprogramming

- multiprogrammed environment ->
  - more threads than processors
  - not all threads can run at the same time
  - any thread can be preemptively suspended at any time
- must ensure that working threads are not delayed by thief threads
  - each thief calls thread.yield() before attempting to steal allowing thread's processor to be yielded to a working thread

Work-Stealing DEqueues

- linearizable with pop always returning a task if possible

A Bounded Work-Stealing DEqueue

- code for BDEQueue here

An Unbounded Work-Stealing Deque

- BDEQueue's limitation is that the queue has a fixed size which may be difficult to predict and maxing out for each thread is a waste of space
- unbounded deque resizes dynamically using a cyclic array
- code for CircularArray and UnboundedDEQueue
- both algorithms become complex but this complexity is justifiable for an application such as an executor pool which may be a performance critical center point of a concurrent multithreaded application

Work Balancing

- threads balance workloads with a randomly chosen partner
- thread’s probability of balancing is inversely proportional to the number of tasks in the thread’s queue
    - threads with few tasks are likely to rebalance
    - threads with nothing to do are certain to rebalance
- advantages
    - provides strong fairness guarantees
    - balancing of tasks over threads reduces contention because there will not be a synchronization overhead for each individual task
- code for WorkSharingThread

Ch. 17 Barriers

- soft real-time -> has a real-time requirement but occasional failure to meet that requirement is not catastrophic
- tasks like window display may take drastically varying times when split amongst threads
- phase -> splitting concurrent tasks into grouped sets of computations where no concurrently executing thread should start the ith set (phase) until others have finished it
- barrier -> mechanism for enforcing phases, or forcing asynchronous threads to act almost like a synchronous process, blocking at a phase until all others have completed
- barriers have similar performance issues to spin locks + more

- notification time -> interval between when some thread has detected that all threads have reached the barrier and when that specific thread leaves the barrier

Barrier Implementations

- code for SimpleBarrier
- breaks because is not reusable between phases

Sense-Reversing Barrier

- sense -> true for even-numbered barriers, false for odd
- thread caches current sense value and either reverses current barrier's sense to continue or spins waiting for current barrier's sense field to change to match its own local sense
- code for SenseBarrier here

Pragma 17.3.1 ->

Combining Tree Barrier

- reduce memory contention at cost of increased latency
- split a large barrier into a tree of smaller barriers
  - have threads combine requests going up the tree
  - distribute notifications going down the tree
- code for TreeBarrier and Node
- reduces memory contention by spreading memory accesses across multiple barriers
- may or may not reduce latency, depending on whether it is faster to decrement a single location or to visit a logarithmic number of barriers

Static Tree Barrier

- barriers so far
  - simple and sense reversing suffer from contention
  - combining tree requires excessive communication
- static tree barrier
  - each thread is assigned to a node in a tree
  - thread at a node waits until all nodes below it in the tree have finished
  - then informs parent
  - then spins waiting for the global sense bit to change

    - on root notification of children completion
        - toggle the global sense bit to notify the waiting threads that all threads are done
    - on a cache-coherent multiprocessor
        - completing the barrier requires log(n) steps moving up the tree
        - notification simply requires changing the global sense (propagated by the cache-coherence mechanism)
    - on machines without coherent caches
        - threads propagate notification down the tree like the combining barrier
- code for StaticTreeBarrier

Termination Detecting Barriers

- work stealing but with termination detection for threads to finish spinning once all threads have finished all tasks
- threads are active or inactive
- termination cannot be solved by having each thread announce that it has become inactive
- use setActive(bool) and isTerminated() methods to increment an atomic integer with a count of the number of active threads
- code for SimpleTDBarrier
- works only if used correctly
    - safety and liveness properties need to be satisfied
    - safety
        - no active thread ever declares itself inactive
        - isTerminated() must only return true if computations really have terminate
    - liveness
        - if computation terminates than isTerminated will eventually return true

Ch. 18 Transactional Memory

Introduction

- tools for multiprocessor programming are flawed

What is Wrong with Locking?

- difficult for inexperience programmers
- priority inversion -> a lower-priority thread is preempted while holding a lock needed by higher-priority threads

- convoying -> thread holding a lock is descheduled, perhaps by exhausting its scheduling quantum by a page fault, or by some other kind of interrupt
    - while the thread holding the lock is inactive, other threads that require that lock will queue up, unable to progress
- deadlocks -> occur if threads attempt to lock the same objects in different orders
- no one really knows how to organize and maintain large systems built on locking

What is Wrong with compareAndSet()?

- difficult to devise and understand synchronization primitive based algorithms
- no obvious way to implement a multi-compare and set on conventional architectures

What is Wrong with Compositionality?

- all synchronization mechanisms are not easily composed
- generally requires atomic actions that do not exist
- solutions are generally ad hoc

What can We Do about It?

- problem summary
    - locks are hard to manage effectively, especially in large systems
    - atomic primitives are too granular and result in complex algorithms
    - it is difficult to compose multiple calls to multiple objects into atomic units
- transactional memory is an emerging solution

Transactions and Atomicity

- transaction -> a sequence of steps executed by a single thread
    - must be serializable
- serializability is a kind of coarse-grained version of linearizability
    - linearizability defined atomicity of individual objects by requiring that each method call of a given object appear to take effect instantaneously between its invocation and response
    - serializability defines atomicity for entire transactions (blocks of code that may include calls to multiple objects)
        - ensures that a transaction appears to take effect between the invocation of its first call and the response to its last call
- transactions do not deadlock or livelock
- livelock -> similar to deadlock but each thread takes action that then continues to cause the other thread(s) to not progress, resulting in starvation (common in deadlock avoidance algorithms)
- atomic keyword (hypothetical language feature) delimits a transaction

- speculative execution -> as a transaction executes, makes tentative changes to objects
    - completion without a synchronization conflict results in a commit
    - otherwise aborts and discards changes
- transactions are nestable and modular
- for conditional synchronization, the atomic block has a corresponding orElse block
- transactional memory can be implemented in hardware (HTM) or software (STM)

Software Transactional Memory

- implemented with libraries

Transactions and Transactional Threads

- thread local Transaction object
    - active
    - aborted
    - committed
- transactional thread
    - handlers
        - onCommit
        - onAbort
        - validation handler
    - takes a callable object

Zombies and Consistency

- some aborted threads still continue to run as zombie transactions
- zombies must be prevented from seeing inconsistent states

Atomic Objects

- concurrent transactions communicate through shared atomic objects

Dependent or Independent Progress

- transactional memory
    - free the programmer from worrying about starvation, deadlock, and many other issues that locking is prone to
- must decide on a progress condition for STM
- wait-free and lock-free STM systems can be designed but are not efficient enough to be practical
- two practical approaches
    - nonblocking -> obstruction-free

        - blocking, lock-based -> deadlock-free
- obstruction freedom -> not all threads can be blocked by delays or failures of other threads
- deadlock-freedom -> does not guarantee progress if threads stop in critical sections
- for both, progress is guaranteed by a contention manager
    - uses spinning or yielding to delay contending threads so that some thread can always make progress

Contention Managers

- advises a transaction that may cause a synchronization conflict on whether to abort the other transaction immediately or to stall to allow the other transaction an opportunity to complete
- code for ContentionManager (abstract)
- types
    - backoff -> a stalling thread repeatedly backs off (doubling time limit)
    - priority -> transactions are timestamped, older timestamps abort other transaction and wait otherwise
    - greedy -> timestamped, A aborts B if A is older or B is waiting for another transaction (eliminates chains of waiting)
    - karma -> transactions track work accomplished and more work accomplished takes precedence
- code for BackoffManager

Implementing Atomic Objects

- transactional implementation of an atomic object must provide getter and setter methods that invoke transactional synchronization and recovery
- implement from abstract AtomicObject

An Obstruction-Free Atomic Object

- any thread that runs by itself will eventually make progress
- overview
    - owner field
    - old version field
    - new version field
    - logical fields -> may not be implemented as fields
    - transaction access means opening and possibly resetting fields
    - for B as previous owner
        - if B committed, new version is current
            - A is new owner
            - A sets old = new

            - sets new to a copy of previous new or sets new to new value
    - for B as aborted
        - old version is current
        - A is new owner
        - old = old
        - new = old copy if setter call
        - new = old if getter call
    - for B active
        - A and B conflict
        - A consults contention manager
        - one transaction aborts other based on contention manager
- why it works
    - serializable because if A active -> committed it must still be owner because it would've been aborted otherwise
    - follows that none of objects read or wrote have been updated since A first accessed them, so A is updating a snapshot of the objects it accessed
- in detail
    - code for FreeObject and openWrite method here

A Lock-Based Atomic Object

- obstruction-free is inefficient
    - writes continually allocate locators and versions
    - reads go through two levels of indirection
- overview
    - reads optimistically and later detects conflicts
    - detects conflicts with a global version clock
    - stamp field, lock field, and version field
    - transaction virtually executes (stores cached thread local version and modifies)
    - validate check that the object's timestamp is not greater than the transaction's read stamp
    - steps to commit
        - lock objects in write set using timeouts to avoid deadlock
        - compareAndSet global version clock (serialization point)
        - transaction checks each object in read set for lock and stamps that are less than transaction's and commits if so
        - transaction updates the stamp field of each object in its write set then releases lock
        - aborts at any failed test
- why it works
    - transactions are serializable in the order they increment the global version clock

    - if A reads x and A later commits, then x could not have changed between the time A first reads x and the time A increments the global version clock
    - if A with read stamp r observes x is unlocked at time t, then any subsequent modifications to x will give x a stamp larger than r
    - if a transaction B commits before A, and modifies an object read by A, then A's validation handler will either observe that x is locked by B, or that x's stamp is greater than r, and will abort either way
- in detail
    - code for WriteSet, VersionClock, LockObject openRead, openWrite, validate and OnValidate and OnCommit handlers

Hardware Transactional Memory

- most cache coherence protocols already implement requirements for transactional memory at a hardware level
- only requires a few minor changes

Cache Coherence

- caches store cache lines and can map addresses to lines
- processors and memory communicate over a bus
- cache lines are tagged with state information
- MESI protocol states
    - modified -> eventually requires write back due to modification, not cached elsewhere
    - exclusive -> not modified and not cached elsewhere (typical default on load)
    - shared -> not modified, may be cached elsewhere
    - invalid -> no meaningful data in line
- detects synchronization conflicts and ensures processors agree on shared state
- cache coherence protocols can be complex
    - a requested load to exclusive causes other copies to be invalidated and modified copies to write before load
    - a requested load to shared causes exclusives to be switched to shared and modifieds must write back before load
    - full cache may cause eviction which will discard shared or exclusive but write back modified

Transactional Cache Coherence

- add transactional bit to each cache line, defaulted to unset
- when cache is used for transaction, set
- ensure modified transactional lines are not written back to memory

- invalidating a transactional cache lines aborts the transaction (represents synchronization conflict -> two stores or load and store)
- modified line is invalidated or evicted, discard value
- if cache evicts a transactional line, abort transaction
- commit when transaction is complete if no lines have been invalidated or evicted

Enhancements

- limitations
  - size of transaction is limited to size of cache line
  - best suited for short, small transactions
  - many caches are direct-mapped so a transaction that accesses addresses mapped to same cache line will fail
  - some caches are set-associative so a transaction that accesses addresses mapped to k + 1 addresses will fail
  - few caches are fully-associative
- can split cache
  - large, direct-mapped main cache and small fully associative victim cache
  - large set-associated non-transactional cache, and a small fully-associative transactional cache for transactional lines

See Appendix B for Hardware Information and a description of cache-conscious programming

Additional Notes

C++ Memory Model