Design of Parallel and High-Performance Computing
Fall 2017
Lecture: Locks (contd.) and Lock-Free

Motivational video: https://www.youtube.com/watch?v=jhApQIPQquw

Instructor: Torsten Hoefler & Markus Püschel
TA: Salvatore Di Girolamo

Administrivia
- Final project presentation: Monday 12/18 during last lecture
- Report will be due in January!
- Still, starting to write early is very helpful — write — rewrite — rewrite (no joke!)
- Some more ideas what to talk about:
  - What tools/programming language/parallelization scheme do you use?
  - Which architecture? (we only offer access to Xeon Phi, you may use different)
  - How to verify correctness of the parallelization?
  - How to argue about performance (bounds, what to compare to?)
  - (Somewhat) realistic use-cases and input sets?
  - What are the key concepts employed?
  - What are the main obstacles?

Review of last lecture
- Language memory models
  - Java/C++ memory model overview
  - Synchronized programming
- Locks
  - Broken two-thread locks
  - Peterson
  - N-thread locks (filter lock)
  - Many different locks, strengths and weaknesses
  - Lock options and parameters
- Formal proof methods
  - Correctness (mutual exclusion as condition)
  - Progress

Goals of this lecture
- More N-thread locks!
  - Hardware operations for concurrency control
- More on locks (using advanced operations)
  - Spin locks
  - Various optimized locks
- Even more on locks (issues and extended concepts)
  - Deadlocks, priority inversion, competitive spinning, semaphores
- Case studies
  - Barrier, reasoning about semantics
- Locks in practice: a set structure

Lock Fairness
- Starvation freedom provides no guarantee on how long a thread waits or if it is "passed"!
- To reason about fairness, we define two sections of each lock algorithm:
  - Doorway D (bounded # of steps)
  - Waiting W (unbounded # of steps)

void lock() {
    int j = 1 - tid;
    flag[tid] = true; // I'm interested
    victim = tid;    // other goes first
    while (flag[j] && victim == tid) {};
}

FIFO locks:
- If T_a finishes its doorway before T_b the CR_a → CR_b
- Implies fairness
**Lamport’s Bakery Algorithm (1974)**
- Is a FIFO lock (and thus fair)
- Each thread takes a number in *doorway* and threads enter in the order of their number!

```java
volatile int flag[n] = {0,0,...,0};
volatile int label[n] = {0,0,...,0};

void lock() {
  flag[tid] = 1; // request
  label[tid] = max(label[0], ..., label[n-1]) + 1; // take ticket
  while ((3k != tid) && (flag[k] && (label[k],k) <* (label[tid],tid))) {};
}
public void unlock() {
  flag[tid] = 0;
}
```

**Advantages:**
- Elegant and correct solution
- Starvation free, even FIFO fairness

**Not used in practice!**
- Why?
- Needs to read/write N memory locations for synchronizing N threads
- Can we do better?
  - Using only atomic registers/memory

**A Lower Bound to Memory Complexity**
- Theorem 5.1 in [1]: “If S is a [atomic] read/write system with at least two processes and S solves mutual exclusion with global progress [deadlock-freedom], then S must have at least as many variables as processes”

- So we’re doomed! Optimal locks are available and they’re fundamentally non-scalable. Or not?

**Beyond Atomic Registers – New HW ops?**
- Hardware atomic operations:
  - Test&Set (int old)(int *mem)
  - Atomic swap (bool succ)(int reg, int *mem)
  - Fetch&Op (int old)(int reg, int *mem)
  - Compare&Swap (int old)(int *mem, int cmp, int new)
  - Load-linked/Store-Conditional (LL(*mem)) // (bool succ)(SC(*mem)
  - Intel TSX (transactional synchronization extensions) XBEGIN(void *fallback)/XEND

**Relative Power of Synchronization**
- Design-Problem I: Multi-core Processor
  - Which atomic operations are effective (simple to implement and fast for apps)?
- Design-Problem II: Complex Application
  - What atomic should a programmer use?
- Concept of “consensus number” C if a primitive can be used to solve the “consensus problem” in a finite number of steps (even if threads stop)
  - atomic registers have C=1 (thus locks have C=11)
  - TAS, Swap, Fetch&Op have C=2
  - CAS, LL/SC, TM have C=∞

**Test-and-Set Locks**
- Test-and-Set semantics
  - Memoize old value
  - Set fixed value TASval (1)
  - Return old value
- After execution:
  - Post-condition is a fixed (constant) value!
Test-and-Set Locks

- Assume TASval indicates “locked”
- Write something else to indicate “unlocked”
- TAS until return value is \( \neq \) TASval (1)

When will the lock be granted?
Does this work well in practice?

volatile int lock = 0;
void lock() {
    while (TestAndSet(&lock) == 1);
}
void unlock() {
    lock = 0;
}

Contention

- On x86, the XCHG instruction is used to implement TAS
- For experts: x86 LOCK is superfluous!
- Cacheline is read and written
  - Ends up in exclusive state, invalidates other copies
  - Cacheline is “thrown” around uselessly
  - High load on memory subsystem

\texttt{movl} $1, %eax
\texttt{xchg} %eax, (%ebx)

Test-and-Test-and-Set (TATAS) Locks

- Spinning in TAS is not efficient
- Spin on cache line in shared state
  - All threads at the same time, no cache coherency/memory traffic

Danger!
- Efficient but use with great care!
- Generalizations are dangerous

volatile int lock = 0;
void lock() {
    do {
        while (lock == 1);
    } while (TestAndSet(&lock) == 1);
}
void unlock() {
    lock = 0;
}

Warning: Be careful with generalizations!

Example: Double-Checked Locking

Problem: Memory ordering leads to race-conditions!

TAS Lock with Exponential Backoff

- Exponential backoff eliminates contention statistically
  - Locks granted in unpredictable order
  - Starvation possible but unlikely
  - How can we make it even less likely?

volatile int lock = 0;
void lock() {
    while (TestAndSet(&lock) == 1) {
        wait(time);
        time *= 2; // double waiting time
    }
}
void unlock() {
    lock = 0;
}

Contention?

- Do TATAS locks still have contention?
- When lock is released, k threads fight for cache line ownership
  - One gets the lock, all get the CL exclusively (serially!)
  - What would be a good solution? (think “collision avoidance”)

volatile int lock = 0;
void lock() {
    do {
        while (lock == 1);
    } while (TestAndSet(&lock) == 1);
}
void unlock() {
    lock = 0;
}
TAS Lock with Exponential Backoff

- Exponential backoff eliminates contention statistically
  - Locks granted in unpredictable order
  - Starvation possible but unlikely
  - Maximum waiting time makes it less likely

```c
volatile int lock = 0;
const int maxtime = 1000;

void lock() {
    while (TestAndSet(&lock) == 0) {
        wait(time);
        time = min(time * 2, maxtime);
    }
}
```

```
void unlock() {
    lock = 0;
}
```

Comparison of TAS Locks

![Comparison of TAS Locks](image)

Improvements?

- Are TAS locks perfect?
  - What are the two biggest issues?
  - Cache coherency traffic (contending on same location with expensive atomics)
  
    -- OR --

  - Critical section underutilization (waiting for backoff times will delay entry to CR)

- What would be a fix for that?
  - Critical section underutilization (waiting for backoff times will delay entry to CR)
  - How is this solved at airports and shops (often at least)?

- Queue locks -- Threads enqueue
  - Learn from predecessor if it’s their turn
  - Each threads spins at a different location
  - FIFO fairness

Array Queue Lock

- Array to implement queue
  - Tail-pointer shows next free queue position
  - Each thread spins on own location
  - Critical padding!
  - index[] array can be put in TLS

- So are we done now?
  - What’s wrong?
  - Synchronizing M objects requires Θ(NM) storage
  - What do we do now?

```c
typedef struct qnode {
    struct qnode *prev;
    int succ_blocked;
} qnode;

qnode *lck = new qnode; // node owned by lock

void lock(qnode *lck, qnode *qn) {
    qn->succ_blocked = 1;
    lck->prev = FetchAndSet(lck, qn);
    qn->prev = lck;
    qn->succ_blocked = 0;
}
```

```c
void unlock(qnode **qn) {
    qnode *pred = (*qn)->prev;
    (*qn)->succ_blocked = 0;
    *qn = pred;
}
```

CLH Lock (1993)

- List-based (same queue principle)
- Discovered twice by Craig, Landin, Hagersten 1993/94
- 2N+3M words
- N threads, M locks
- Requires thread-local qnode pointer
- Can be hidden!

```c
typedef struct qnode {
    struct qnode *prev;
    int succ_blocked;
} qnode;

qnode *lck = new qnode; // node owned by lock

void lock(qnode *lck, qnode *qn) {
    qn->succ_blocked = 1;
    lck->prev = FetchAndSet(lck, qn);
    qn->prev = lck;
    qn->succ_blocked = 0;
}
```

```c
void unlock(qnode **qn) {
    qnode *pred = (*qn)->prev;
    (*qn)->succ_blocked = 0;
    *qn = pred;
}
```

CLH Lock (1993)

- Qnode objects represent thread state!
  - succ_blocked == 1 if waiting or acquired lock
  - succ_blocked == 0 if released lock

- List is implicit!
  - One node per thread
  - Spin location changes
  - NUMA issues (cacheless)

- Can we do better?

```c
typedef struct qnode {
    struct qnode *prev;
    int succ_blocked;
} qnode;

qnode *lck = new qnode; // node owned by lock

void lock(qnode *lck, qnode *qn) {
    qn->succ_blocked = 1;
    lck->prev = FetchAndSet(lck, qn);
    qn->prev = lck;
    qn->succ_blocked = 0;
}
```

```c
void unlock(qnode **qn) {
    qnode *pred = (*qn)->prev;
    (*qn)->succ_blocked = 0;
    *qn = pred;
}
```
MCS Lock (1991)

- Make queue explicit
  - Acquire lock by appending to queue
  - Spin on own node until locked is reset
- Similar advantages as CLH but
  - Only 2N + M words
  - Spinning position is fixed! Benefits cache-less NUMA
- What are the issues?
  - Releasing lock spins
  - More atomics!

```
typedef struct qnode {
  struct qnode *next;
  int locked;
} qnode;
qnode *lck = NULL;

void lock(qnode *lck, qnode *qn) {
  qn->next = NULL;
  qnode *pred = FetchAndSet(lck, qn);
  if(pred != NULL) {
    qn->locked = 1;
    pred->next = qn;
    while(qn->locked);
  }
}

void unlock(qnode *lck, qnode *qn) {
  if(qn->next == NULL) { // if we're the last waiter
    if(CAS(lck, qn, NULL)) return;
  }
  qn->next->locked = 0; // free next waiter
  qn->next = NULL;
}

typedef struct qnode {
  struct qnode *next;
  int locked;
} qnode;
qnode *lck = NULL;

void lock(qnode *lck, qnode *qn) {
  qn->next = NULL;
  qnode *pred = FetchAndSet(lck, qn);
  if(pred != NULL) {
    qn->locked = 1;
    pred->next = qn;
    while(qn->locked);
  }
}

void unlock(qnode *lck, qnode *qn) {
  if(qn->next == NULL) { // if we're the last waiter
    if(CAS(lck, qn, NULL)) return;
  }
  qn->next->locked = 0; // free next waiter
  qn->next = NULL;
}
```

Lessons Learned!

- Key Lesson:
  - Reducing memory (coherency) traffic is most important!
  - Not always straight-forward (need to reason about CL states)
- MCS: 2006 Dijkstra Prize in distributed computing
  - “an outstanding paper on principles of distributed computing, whose significance and impact on the theory and/or practice of distributed computing has been evident for at least a decade”
  - “probably the most influential practical mutual exclusion algorithm ever”
  - “vastly superior to all previous mutual exclusion algorithms”
  - fast, fair, scalable → widely used, always compared against!

Time to Declare Victory?

- Down to memory complexity of 2N+M
  - Probably close to optimal
- Only local spinning
  - Several variants with low expected contention
- But: we assumed sequential consistency
  - Reality causes trouble sometimes
  - Sprinkling memory fences may harm performance
  - Optimize to a given architecture?
  - Open research on minimally-synching algorithms!
    - Come and talk to me if you’re interested in research
- But: atomics can still contend
  - Algorithmic issue
  - Seems very hard to fix (counting networks/trees?)
    - Come and talk to me if you’re interested in research

More Practical Optimizations

- Let’s step back to “data race”
  - (recap) two operations A and B on the same memory cause a data race if one of them is a write (“conflicting access”) and neither A→B nor B→A
  - So we put conflicting accesses into a CR and lock it!
    - Remember: this also guarantees memory consistency in C++/Java!

- Let’s say you implement a web-based encyclopedia
  - Consider the “average two accesses” – do they conflict?

Reader-Writer Locks

- Allows multiple concurrent reads
  - Multiple reader locks concurrently in CR
  - Guarantees mutual exclusion between writer and writer locks and reader and writer locks
- Syntax:
  - `read_unlock()`
  - `write_unlock()`

```
// reader
int read_lock() {
  AtomicAdd(lock, 1);
}

// writer
void write_lock() {
  AtomicAdd(lock, W);
}

// reader
int read_unlock() {
  AtomicAdd(lock, -1);
}
```

A Simple RW Lock

- Seems efficient!?
  - Is it? What’s wrong?
  - Polling CAS!
- Is it fair?
  - Readers are preferred!
  - Can always delay writers (again and again again)

```
const W = 1;
const R = 2;
volatile int lock=0; // LSB is writer flag!

void read_lock(lock_t lock) {
  AtomicAdd(lock, R);
  while(lock & W);
}

void write_lock(lock_t lock) {
  AtomicAdd(lock, W);
}

void read_unlock(lock_t lock) {
  AtomicAdd(lock, -R);
}

void write_unlock(lock_t lock) {
  AtomicAdd(lock, -W);
}
```

Reader-Writer Locks

- Allows multiple concurrent reads
  - Multiple reader locks concurrently in CR
  - Guarantees mutual exclusion between writer and writer locks and reader and writer locks
- Syntax:
  - `read_unlock()`
  - `write_unlock()`

```
// reader
int read_lock() {
  AtomicAdd(lock, 1);
}

// writer
void write_lock() {
  AtomicAdd(lock, W);
}

// reader
int read_unlock() {
  AtomicAdd(lock, -1);
}
```

Lessons Learned!

- Key Lesson:
  - Reducing memory (coherency) traffic is most important!
  - Not always straight-forward (need to reason about CL states)
- MCS: 2006 Dijkstra Prize in distributed computing
  - “an outstanding paper on principles of distributed computing, whose significance and impact on the theory and/or practice of distributed computing has been evident for at least a decade”
  - “probably the most influential practical mutual exclusion algorithm ever”
  - “vastly superior to all previous mutual exclusion algorithms”
  - fast, fair, scalable → widely used, always compared against!

Time to Declare Victory?

- Down to memory complexity of 2N+M
  - Probably close to optimal
- Only local spinning
  - Several variants with low expected contention
- But: we assumed sequential consistency
  - Reality causes trouble sometimes
  - Sprinkling memory fences may harm performance
  - Optimize to a given architecture?
  - Open research on minimally-synching algorithms!
    - Come and talk to me if you’re interested in research
- But: atomics can still contend
  - Algorithmic issue
  - Seems very hard to fix (counting networks/trees?)
    - Come and talk to me if you’re interested in research

More Practical Optimizations

- Let’s step back to “data race”
  - (recap) two operations A and B on the same memory cause a data race if one of them is a write (“conflicting access”) and neither A→B nor B→A
  - So we put conflicting accesses into a CR and lock it!
    - Remember: this also guarantees memory consistency in C++/Java!

- Let’s say you implement a web-based encyclopedia
  - Consider the “average two accesses” – do they conflict?

Reader-Writer Locks

- Allows multiple concurrent reads
  - Multiple reader locks concurrently in CR
  - Guarantees mutual exclusion between writer and writer locks and reader and writer locks
- Syntax:
  - `read_unlock()`
  - `write_unlock()`

```
// reader
int read_lock() {
  AtomicAdd(lock, 1);
}

// writer
void write_lock() {
  AtomicAdd(lock, W);
}

// reader
int read_unlock() {
  AtomicAdd(lock, -1);
}
```

A Simple RW Lock

- Seems efficient!?
  - Is it? What’s wrong?
  - Polling CAS!
- Is it fair?
  - Readers are preferred!
  - Can always delay writers (again and again again)
Fixing those Issues?

- Polling issue:
  - Combine with MCS lock idea of queue polling
- Fairness:
  - Count readers and writers

The final algorithm (Alg. 4) has a flaw that was corrected in 2003!

Deadlocks

- Kansas state legislature: “When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.”
  
  [according to Botkin, Harlow “A Treasury of Railroad Folklore” (pp. 381)]

What are necessary conditions for deadlock?


Deadlocks

- Necessary conditions:
  - Mutual Exclusion
  - Hold one resource, request another
  - No preemption
  - Circular wait in dependency graph
- One condition missing will prevent deadlocks!
  - Different avoidance strategies (which?)

Issues with Spinlocks

- Spin-locking is very wasteful
  - The spinning thread occupies resources
  - Potentially the PE where the waiting thread wants to run requires context switch!
- Context switches due to
  - Expiration of time-slices (forced)
  - Yielding the CPU

What is this?


Why is the 1997 Mars Rover in our lecture?

- It landed, received program, and worked ... until it spuriously rebooted!
  - watchdog
- Scenario (vxWorks RT OS):
  - Single CPU
  - Two threads A, B sharing common bus, using locks
  - (independent) thread C wrote data to flash
  - Priority: A>C>B (A highest, B lowest)
  - Thread C would run into a lifelock (infinite loop)
  - Thread B was preempted by C while holding lock
  - Thread A got stuck at lock 😞
Priority Inversion

- If busy-waiting thread has higher priority than thread holding lock ⇒ no progress!
- Can be fixed with the help of the OS
  - E.g., mutex priority inheritance (temporarily boost priority of task in CR to highest priority among waiting tasks)

Fighting CPU waste: Condition Variables

- Allow threads to yield CPU and leave the OS run queue
  - Other threads can get them back on the queue!
- cond_wait(cond, lock) – yield and go to sleep
- cond_signal(cond) – wake up sleeping threads
- Wait and signal are OS calls
  - Often expensive, which one is more expensive?
    - Wait, because it has to perform a full context switch

Condition Variable Semantics

- Hoare-style:
  - Signaler passes lock to waiter, signaler suspended
  - Waiter runs immediately
  - Waiter passes lock back to signaler if it leaves critical section or if it waits again
- Mesa-style (most used):
  - Signaler keeps lock
  - Waiter simply put on run queue
  - Needs to acquire lock, may wait again

When to Spin and When to Block?

- What is a “while”?
- Optimal time depends on the future
  - When will the active thread leave the CR?
  - Can compute optimal offline schedule
    - Q: What is the optimal offline schedule (assuming we know the future, i.e., when the lock will become available)?
    - Actual problem is an online problem
- Competitive algorithms
  - An algorithm is c-competitive if for a sequence of actions x and a constant a holds:
    \[ C(x) \leq c \cdot C_{opt}(x) + a \]
  - What would a good spinning algorithm look like and what is the competitiveness?

Competitive Spinning

- If T is the overhead to process a wait, then a locking algorithm that spins for time T before it blocks is 2-competitive!
- If randomized algorithms are used, then e/(e-1)-competitiveness (~1.58) can be achieved
  - See paper above!
Generalized Locks: Semaphores

- Controlling access to more than one resource
  - Described by Dijkstra 1965
- Internal state is an atomic counter \( C \)
- Two operations:
  - \( P() \) – block until \( C > 0 \); decrement \( C \) (atomically)
  - \( V() \) – signal and increment \( C \)
- Binary or 0/1 semaphore equivalent to lock
  - \( C \) is always 0 or 1, i.e., \( V() \) will not increase it further
- Trivia:
  - If you’re lucky (aehem, speak Dutch), mnemonics: Verhogen (increment) and Prolaag (probeer te verlagen = try to reduce)

Semaphore Implementation

- Can be implemented with mutual exclusion!
  - And can be used to implement mutual exclusion
- … or with test and set and many others!
- Also has fairness concepts:
  - Order of granting access to waiting (queued) threads
    - strictly fair (starvation impossible, e.g., FIFO)
    - weakly fair (starvation possible, e.g., random)

Case Study 1: Barrier

- Barrier semantics:
  - No process proceeds before all processes reached barrier
  - Similar to mutual exclusion but not exclusive, rather “synchronized”
- Often needed in parallel high-performance programming
  - Especially in SPMD programming style
- Parallel programming “frameworks” offer barrier semantics (pthread, OpenMP, MPI)
  - MPI_Barrier() (process-based)
  - pthread_barrier
  - #pragma omp barrier
  - …
- Simple implementation: lock xadd + spin
  - Problem: when to re-use the counter?
  - Cannot just set it to 0 \( \Rightarrow \) Trick: “lock xadd -1” when done

Case Study 2: Reasoning about Semantics

- Comments on a Problem in Concurrent Programming Control
  - Dear Editor:
    - I would like to comment on Mr. Dijkstra’s solution [Distillation of a problem in concurrent programming control. Comm. ACM 8 (Sept. 1965), 900] to a neat problem that is hardly academic. We are using it now on a multiple computer complex.
    - When there are only two computers, the algorithm may be simplified to the following:
    - Basham array \( h(i, j) \); integer \( i, j \); comment: This is the program for computer \( i \), which may be either 0 or 1, computer \( j \neq i \) is in the other one, 0 or 1;
    - \( h(0, 0) := \text{false}; \)
    - \( h(0, 0) := \text{false}; \)
    - \( h(1, 1) := \text{false}; \)
    - \( h(1, 1) := \text{false}; \)
  - Mr. Dijkstra has come up with a clever solution to a really practical problem.

C ACM
Volume 9 Issue 1, Jan. 1966

Case Study 2: Reasoning about Semantics

- Is the proposed algorithm correct?
  - We may proof it manually
    - Using tools from the last lecture
  - Or use automated proofs (model checking)
  - E.g., SPIN (Promela syntax)
Case Study 2: Reasoning about Semantics

- Spin tells us quickly that it found a problem
  - A sequentially consistent order that violates mutual exclusion!
- It’s not always that easy
  - This example comes from the SPIN tutorial
  - More than two threads make it much more demanding!
- More in the recitation!

Locks in Practice

- Running example: List-based set of integers
  - S.insert(v) – return true if v was inserted
  - S.remove(v) – return true if v was removed
  - S.contains(v) – return true iff v in S
- Simple ordered linked list
  - Do not use this at home (poor performance)
  - Good to demonstrate locking techniques
    - E.g., skip lists would be faster but more complex

Set Structure in Memory

- This and many of the following illustrations are provided by Maurice Herlihy in conjunction with the book “The Art of Multiprocessor Programming”

Sequential Operations

- add()
  - a  b  c  d
  - Sorted with Sentinel nodes (min & max possible keys)
- remove()
  - a  b  c

Sequential Set

```c
typedef struct {
  int key;
  node *next;
} node;
```

Concurrent Sets

- What can happen if multiple threads call set operations at the “same time”?
  - Operations can conflict!
- Which operations conflict?
  - (add, remove), (add, add), (remove, remove), (remove, contains) will conflict
  - (add, contains) may miss update (which is fine)
  - (contains, contains) does not conflict
- How can we fix it?

```c
boolean add(S, x) {
  node *pred = S.head;
  node *curr = pred.next;
  while(curr.key < x) {
    pred = ...
    curr = curr.next;
  }
  if(curr.key == x)
    return true;
  return false;
}
```

```c
boolean remove(S, x) {
  node *pred = S.head;
  node *curr = pred.next;
  while(curr.key < x) {
    pred = ...
    curr = curr.next;
  }
  if(curr.key == x) {
    pred.next = curr.next;
    free(curr);
    return true;
  }
  return false;
}
```

```c
boolean contains(S, x) {
  int *curr = S.head;
  while(curr.key < x) curr = curr.next;
  if(curr.key == x)
    return true;
  return false;
}
```
Coarse-grained Locking

```c
boolean add(S, x) {
    lock(S);
    node *pred = S.head;
    node *curr = pred.next;
    while(curr.key < x) {
        pred = curr;
        curr = curr.next;
    }
    if(curr.key == x) {
        unlock(S);
        return true;
    }
    unlock(S);
    return false;
}
```

```c
boolean remove(S, x) {
    lock(S);
    node *pred = S.head;
    node *curr = pred.next;
    while(curr.key < x) {
        pred = curr;
        curr = curr.next;
    }
    if(curr.key == x) {
        node *next = curr.next;
        free(curr);
        unlock(S);
        return true;
    }
    unlock(S);
    return false;
}
```

Coarse-grained Locking

- Correctness proof?
  - Assume sequential version is correct
    - Alternative: define set of invariants and proof that initial condition as well as all transformations adhere (pre- and post conditions)
    - Proof that all accesses to shared data are in CRs
      - This may prevent some optimizations
- Is the algorithm deadlock-free? Why?
  - Locks are acquired in the same order (only one lock)
- Is the algorithm starvation-free and/or fair? Why?
  - It depends on the properties of the used locks!

Coarse-grained Locking

- Is the algorithm performing well with many concurrent threads accessing it?
  - No, access to the whole list is serialized
  - BUT: it's easy to implement and proof correct
    - Those benefits should never be underestimated
    - May be just good enough
    - “We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified” — Donald Knuth (in Structured Programming with Goto Statements)

Coarse-grained Locking

How to Improve?

- Will present some “tricks”
  - Apply to the list example
  - But often generalize to other algorithms
  - Remember the trick, not the example!
- See them as “concurrent programming patterns” (not literally)
  - Good toolbox for development of concurrent programs
  - They become successively more complex

Tricks Overview

1. Fine-grained locking
   - Split object into “lockable components”
   - Guarantee mutual exclusion for conflicting accesses to same component
2. Reader/writer locking
3. Optimistic synchronization
4. Lazy locking
5. Lock-free
Tricks Overview

1. Fine-grained locking
2. Reader/writer locking
   - Multiple readers hold lock (traversal)
   - \( \text{contains()} \) only needs read lock
   - Locks may be upgraded during operation
     \( \text{Must ensure starvation-freedom for writer locks!} \)
3. Optimistic synchronization
4. Lazy locking
5. Lock-free

Hand-over-Hand (fine-grained) locking

Trick 1: Fine-grained Locking

- Each element can be locked
  - High memory overhead
  - Threads can traverse list concurrently like a pipeline
- Tricky to prove correctness
  - And deadlock-freedom
  - Two-phase locking (acquire, release) often helps
- Hand-over-hand (coupled locking)
  - Not safe to release x’s lock before acquiring x.next’s lock
    \( \text{will see why in a minute} \)
  - Important to acquire locks in the same order

typedef struct {
  int key;
  node *next;
  lock_t lock;
} node;
Removing a Node

- Why lock target node?

Concurrent Removes

- remove(b)
- remove(c)
Uh, Oh

- Uh, Oh

Insight
- If a node x is locked
  - Successor of x cannot be deleted!
- Thus, safe locking is
  - Lock node to be deleted
  - And its predecessor!
  - → hand-over-hand locking

Hand-Over-Hand Again
- Hand-Over-Hand Again

Hand-Over-Hand Again
- Hand-Over-Hand Again

Bad news, c not removed
- Bad news, c not removed
Hand-Over-Hand Again

Hand-Over-Hand Again

Hand-Over-Hand Again

Hand-Over-Hand Again

Removing a Node

Removing a Node

Removing a Node

Removing a Node
Removing a Node

Must acquire Lock for b

remove(b)
remove(c)
Removing a Node

1. Waiting to acquire lock for b
2. Remove(c)
3. Wait!
4. Proceed to remove(b)
5. Remove(b)
6. Remove(b)
7. Remove(b)
What are the Issues?

- We have fine-grained locking, will there be contention?
  - Yes, the list can only be traversed sequentially, a remove of the 3rd item will block all other threads!
  - This is essentially still serialized if the list is short (since threads can only pipeline on list elements)
- Other problems, ignoring contention?
  - Must acquire $O(|S|)$ locks

Trick 2: Reader/Writer Locking

- Same hand-over-hand locking
  - Traversal uses reader locks
  - Once add finds position or remove finds target node, upgrade both locks to writer locks
  - Need to guarantee deadlock and starvation freedom!
- Allows truly concurrent traversals
  - Still blocks behind writing threads
  - Still $O(|S|)$ lock/unlock operations

Trick 3: Optimistic synchronization

- Similar to reader/writer locking but traverse list without locks
  - Dangerous! Requires additional checks.
- Harder to proof correct

Optimistic: Traverse without Locking

Optimistic: Lock and Load

Optimistic: Lock and Load
What could go wrong?

What could go wrong?

What could go wrong?

What could go wrong?
What could go wrong?

Validate – Part 1

What Else Could Go Wrong?

What Else Could Go Wrong?
**What Else Could Go Wrong?**

![Diagram showing possible errors in synchronization]

**Validate Part 2**
(while holding locks)

![Diagram showing validation process]

**Optimistic synchronization**
- One MUST validate AFTER locking
  1. Check if the path how we got there is still valid!
  2. Check if locked nodes are still connected
  - If any of those checks fail? Start over from the beginning (hopefully rare)
- Not starvation-free
  - A thread may need to abort forever if nodes are added/removed
  - Should be rare in practice!
- Other disadvantages?
  - All operations require two traversals of the list!
  - Even contains() needs to check if node is still in the list!

**Trick 4: Lazy synchronization**
- We really want one list traversal
- Also, contains() should be wait-free
  - Is probably the most-used operation
- Lazy locking is similar to optimistic
  - Key insight: removing is problematic
  - Perform it “lazily”
- Add a new “valid” field
  - Indicates if node is still in the set
  - Can remove it without changing list structure!
  - Scan once, contains() never locks!

**Lazy Removal**

![Diagram showing lazy removal]

```c
typedef struct {
  int key;
  node *next;
  lock_t lock;
  boolean valid;
} node;
```
Lazy Removal

How does it work?

- Eliminates need to re-scan list for reachability
  - Maintains invariant that every unmarked node is reachable!
- Contains can now simply traverse the list
  - Just check marks, not reachability, no locks
- Remove/Add
  - Scan through locked and marked nodes
  - Removing does not delay others
  - Must only lock when list structure is updated
    - Check if neither pred nor curr are marked, pred.next == curr
remove(b)
a not marked
a still points to b
Logical delete
physical delete
Business as Usual

Problems with Locks
- What are the fundamental problems with locks?
  - Blocking
    - Threads wait, fault tolerance
    - Especially when things like page faults occur in CR
  - Overheads
    - Even when not contended
    - Also memory/state overhead
  - Synchronization is tricky
    - Deadlock, other effects are hard to debug
  - Not easily composable

Lock-free Methods
- No matter what:
  - Guarantee minimal progress
    - i.e., some thread will advance
  - Threads may halt at bad times (no CRs! No exclusion!)
    - i.e., cannot use locks!
  - Needs other forms of synchronization
    - E.g., atomics (discussed before for the implementation of locks)
    - Techniques are astonishingly similar to guaranteeing mutual exclusion

Trick 5: No Locking
- Make list lock-free
- Logical succession
  - We have wait-free contains
  - Make add() and remove() lock-free!
    - Keep logical vs. physical removal
- Simple idea:
  - Use CAS to verify that pointer is correct before moving it

Lock-free Lists
- (1) Logical Removal
  - Use CAS to verify pointer is correct
- (2) Physical Removal
  - Not enough! Why?
Problem...

(1) Logical Removal

(2) Node added

(3) Physical Removal
d

The Solution: Combine Mark and Pointer

(1) Logical Removal

(3) Physical Removal

CAS of two noncontiguous locations
Well, not many machines support it 😐
Any still alive?

(2) Fail CAS: Node not added after logical Removal

Mark-Bit and Pointer are CASed together!

Practical Solution(s)

- Option 1:
  - Introduce "atomic markable reference" type
  - "Steal" a bit from a pointer
  - Rather complex and OS specific 😐

- Option 2:
  - Use Double CAS (or CAS2) 😊
  - CAS of two noncontiguous locations
  - Well, not many machines support it 😐
  - Any still alive?

- Option 3:
  - Our favorite ISA (x86) offers double-width CAS
    Contiguous, e.g., lock cmpxchg16b (on 64-bit systems)

- Option 4:
  - TM!
    E.g., Intel's TSX (essentially a cmpxchg64b (operates on a cache line))

Removing a Node

remove b

remove c

Removing a Node

failed

remove b

remove c
Dealing With Zombie Nodes

- Add() and remove() “help to clean up”
  - Physically remove any marked nodes on their path
  - I.e., if curr is marked: CAS (pred.next, mark) to (curr.next, false) and remove curr
    - If CAS fails, restart from beginning!
- “Helping” is often needed in wait-free algs
- This fixes all the issues and makes the algorithm correct!

Comments

- Atomically updating two variables (CAS2 etc.) has a non-trivial cost
- If CAS fails, routine needs to re-traverse list
  - Necessary cleanup may lead to unnecessary contention at marked nodes
- More complex data structures and correctness proofs than for locked versions
  - But guarantees progress, fault-tolerant and maybe even faster (that really depends)

More Comments

- Correctness proof techniques
  - Establish invariants for initial state and transformations
    - E.g., head and tail are never removed, every node in the set has to be reachable from head, ...
  - Proofs are similar to those we discussed for locks
    - Very much the same techniques (just trickier)
    - Using sequential consistency (or consistency model of your choice 😊)
    - Lock-free gets somewhat tricky
- Source-codes can be found in Chapter 9 of “The Art of Multiprocessor Programming”