Advanced Multiprocessor Programming

Jesper Larsson Träff
traff@par.tuwien.ac.at
Research Group Parallel Computing, 191-1
Faculty of Informatics, Institute of Computer Engineering
Vienna University of Technology (TU Wien)

©Jesper Larsson Träff

"With the notable exception of simple atomic counters, lock-free programming is for specialists."

B. Stroustrup, The C++ Programming Language, 4th Ed., 2013

"The Art of multiprocessor programming": Two concerns

From "Intro. Parallel Computing"

Parallel computing:
The discipline of efficiently utilizing dedicated parallel resources (processors, memories, …) to solve a single, given computational problem.

Performance, efficiency

Concurrent computing:
The discipline of managing and reasoning about interacting processes that may (or may not) take place simultaneously

Coordination, correctness

Abstraction 1: PRAM (Parallel Random Access Machine)

PRAM natural generalization of the RAM

1. Synchronous processors in lock step
2. Unit time memory access (Conflicts: EREW, CREW, CRCW)

Good for studying parallel algorithms and especially establishing lower bounds (we do: lecture in WS)

But (some say): Not realistic. This lecture: Different model, different concerns, close(r) to current multi-core, shared-memory processors

PRAM algorithms and complexity

- Algorithms: With p PRAM processors, solve given problem (sorting, connected components, convex hulls, linear equations, …) p times faster than best possible/best known sequential algorithm
- Data structures: With p processors, make individual or bulk operations p times faster
- Complexity: Establish lower bounds on possible improvements under various assumptions about the memory system; study problems that may be hard to parallelize (no fast algorithms known)

PRAM model: Interesting, relevant, non-trivial results in all 3 areas

Abstraction 2: (Symmetric) Shared-memory computer

Issues/problems:

- Threads need to synchronize and coordinate
- How can we reason about progress and complexity (speed-up)?
- How can we ensure/reason about correctness?
- Scheduling tasks to threads, treads to processors (cores)

1. Threads executed by processors, $|T| \geq |P|$
2. Threads are not synchronized
3. Memory access not unit time
4. (Memory accesses not ordered, memory updates not necessarily in program order)

Shared-memory algorithms and complexity

- Algorithms: Algorithms that are correct for p asynchronous threads; faster than sequential algorithm under optimistic assumptions on thread progress and absence of resource conflicts.
- Data structures: p threads can work independently and each perform individual operations on data structure, complexity not too far from best known sequential implementation, guarantees on progress (for instance, deadlock freedom)
- Complexity: Lower bounds on resource requirements to achieve desired properties

Fact

All current, modern (general-purpose) processors are parallel processors (multi-core cpu, GPU, …), mostly asynchronous

For CPUs:

- Clock-frequency has not been increasing (much) since ca. 2005
- Power-consumption: Same
- Single-core performance no longer following Moore's "law" (performance version)

- Number of transistors still increasing exponentially (Moore's observed and extrapolated "law")
- Number of cores per chip increasing

Kunle Olukotun, 2010, Karl Rupp (TU Wien), 2015

Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

©Jesper Larsson Träff

The first concern: Speeding up time to solution of given problem

Definitions:

Tseq(n): Time (or number of operations) needed to solve given problem sequentially by best possible or best known algorithm and implementation on given machine

Tp(n): Time (or number of operations), by slowest processor, needed to solve given problem of size n using p processors (given algorithm on given machine)

Absolute speed-up: SU(p) = Tseq(n)/Tp(n)

Relative speed-up: T1(n)/Tp(n)

Can time to solve problem be improved?

Does parallel algorithm scale?

Recall:

SU(p) =Θ(p) is best possible (while relative to best known sequential algorithm): linear speed-up

for p ≤ maxP(n)

SU(p) = p: perfect speed-up

The larger the p for which linear/perfect speed-up is achieved, the better

Constants normalized out, assume same constant in both O

Example:
Tp(n) = O(n/p+log n) has linear speed-up for p in O(n/log n), assuming Tseq(n) = O(n), namely (normalizing constants)

SU(p) = n/(n/p+log n) = p/(1+(p log n)/n) > p/(1+ε) for p≤εn/log n

©Jesper Larsson Träff

Practical remarks:

Tseq(n), Tp(n) measured times for some (good) implementation of the algorithms, for some input (Worst-case? Average case? Average over many inputs?)

Measuring time is not trivial. Tp(n) usually defined as time of last processor to finish, assuming all p processors start at the same time (temporal synchronization problem)

T1(n) ≥ Tseq(n), since Tseq(n) is time of best known/possible algorithm

Reporting only relative speed-up with baseline T1(n) compared to Tseq(n) can be grossly misleading

Analogies

Tseq(n) sometimes called the "work" required to solve the problem of size n (in number of "operations")

A good parallel algorithm will effectively divide this work over the available p processors, such that the total parallel work is O(Tseq(n)). In this case

Tp(n) = O(Tseq(n)/p) with linear speed-up

Challenges:

- Load balancing (the work that has to be done must be evenly distributed, no processors idle for too long)
- Little extra work (redundancy, parallelization overhead)
- Small overhead (synchronization, shared data structures)

<u>Definition</u> (Throughput):

Number of operations that can be carried out in some given amount of time t

ThroughputSU(t) = Throughputp(t)/Throughput1(t)

<u>Definition</u> (latency):

Time taken to carry out some given number of operations n

Relevant measures when benchmarking data structures. Important to decide what the operations are and in what distribution

Example: Performance

„painting 5 room flat…" (from Herlihy-Shavit book, apologies)

Amount of work, Tseq(n), here 5 rooms

If the amount of work can be trivially divided (5 rooms over 5 people) into p parts, Tp(n) = Tseq(n)/p, and

SU(p) = Tseq(n)/Tp(n) = p

<u>Trivially (embarrassingly) parallel</u>:

- No coordination overhead
- No extra work done

©Jesper Larsson Träff

Example: Performance

„painting 5 room flat…" (from Herlihy-Shavit book, apologies)

5 people, $s=2/5$, $SU = 1/(2/5+3/5/5) = 25/13 \approx 1.9 < 2.5 = 5/2$

Amount of work, Tseq(n), here 5 rooms

But… some part of the work may not be divisible (one large room), say, some fraction s of the total work, so that $Tp(n) = sTseq(n)+(1-s)Tseq(n)/p$. Then

Amdahl's law

$SU(p) = Tseq(n)/Tp(n) = 1/(s+(1-s)/p) \longrightarrow 1/s$ for $p \longrightarrow \infty$, independently of n

Example: Performance

„painting 5 room flat…" (from Herlihy-Shavit book, apologies)

Scalable parallel algorithm:

- No constant fraction that cannot be parallelized
- Find ways to address the seemingly sequential parts: coordination

Amdahl's law: If there is a constant fraction (independent of n) of the total work Tseq(n) that cannot be parallelized, then

$SU(p) = Tseq(n)/Tp(n) = 1/(s+(1-s)/p) \longrightarrow 1/s$ for $p \longrightarrow \infty$

Example: Performance

„painting 5 room flat…" (from Herlihy-Shavit book, apologies)

Assume sequential part is constant (or slowly growing with n)

$Tp(n) = (Tseq(n)-k)/p + k$

$SU(n) = Tseq(n)/((Tseq(n)-k)/p+k) = p/(1+(p-1)k/Tseq(n))$

$\longrightarrow p$ for $Tseq(n) \longrightarrow \infty$ and fixed p

Not Amdahl

Scaled speed-up: A certain problem size needed for required speed-up

Example: Performance

„painting 5 room flat…" (from Herlihy-Shavit book, apologies)

Non-parallelizable part may be distributed over the algorithm

- Sequential data structures
- Bad coordination constructs: locks (serialization)
- Threads unable to do useful work because blocked (waiting) by other threads
- Work done by "master thread"

Master thread assigned to innocent looking bookkeeping

©Jesper Larsson Träff

Example: Performance

„painting 5 room flat…" (from Herlihy-Shavit book, apologies)

Non-parallelizable part may be distributed over the algorithm

Good parallel algorithm:

- No constant sequential fraction
- Work (total number of operations by all threads) is still O(Tseq(n))

©Jesper Larsson Träff

Example: Coordination

„A and B's dog and cat…" (again from Herlihy-Shavit intro)

- A and B need to access infinitely often
- Either A or B has access, but not both (Mutual exclusion)
- If A and B want to access either A or B will succeed (deadlock-freedom)
- If A wants to access, A will eventually get access, independently of B; same for B (starvation freedom; fairness)

Example: Coordination

„A and B's dog and cat…" (again from Herlihy-Shavit intro)

A flag solution, two flags:

A's protocol:

1. Set flagA=1
2. Wait until flagB==0: ENTER
3. Set flagA=0

B's protocol

1. Set flagB=1
2. While flagA==1:
   a) Set flagB=0
   b) Wait until flagA==0
   c) Set flagB=1
3. ENTER
4. Set flagB=0

©Jesper Larsson Träff

Example: Coordination

„A and B's dog and cat…" (again from Herlihy-Shavit intro)

A's protocol:

1. Set flagA=1
2. Wait until flagB==0: ENTER
3. Set flagA=0

B's protocol

1. Set flagB=1
2. While flagA==1:
   a) Set flagB=0
   b) Wait until flagA==0
   c) Set flagB=1
3. ENTER
4. Set flagB=0

Satisfies mutual exlusion: Assume not, assume both in yard, and consider last time A and B checked flags. A must have entered before B assigned flagB=1. B's last check must have been after A checked, and assigned flagA=1. Contradicts that B has ENTERED

Example: Coordination

„A and B's dog and cat…" (again from Herlihy-Shavit intro)

A's protocol:

1. Set flagA=1
2. Wait until flagB==0: ENTER
3. Set flagA=0

B's protocol

1. Set flagB=1
2. While flagA==1:
   a) Set flagB=0
   b) Wait until flagA==0
   c) Set flagB=1
3. ENTER
4. Set flagB=0

Satisfies deadlock freedom: If both wants to enter, B will eventually see flagA==1, set flagB=0, and A can enter

Example: Coordination

„A and B's dog and cat…" (again from Herlihy-Shavit intro)

A's protocol:

1. Set flagA=1
2. Wait until flagB==0: ENTER
3. Set flagA=0

B's protocol

1. Set flagB=1
2. While flagA==1:
   a) Set flagB=0
   b) Wait until flagA==0
   c) Set flagB=1
3. ENTER
4. Set flagB=0

Does not satisfy starvation freedom: B always yields to A…

See also discussion in D. E. Knuth, The art of computer programming, Volume 4, Satisfiability, Addison-Wesley 2015, p. 20-24

Example: Coordination

„A and B's dog and cat…" (again from Herlihy-Shavit intro)

- B produces, A consumes
- Mutual exclusion: when B produces (delivers to resource), A cannot consume (receive from resource); when A consumes, B cannot deliver
- Starvation freedom: if B can produce infinitely often, and A consume infinitely often, both A and B can proceed
- Correctness: A will not consume unless B has produced

Example: Coordination

„A and B's dog and cat…" (again from Herlihy-Shavit intro)

A flag solution, one flag

A's protocol:

1. Wait until flag==0: ENTER
2. Set flag=1

B's protocol

1. Wait until flag==1: ENTER
2. Set flag=0

Satisfies mutual exclusion, starvation freedom, correctness

Example: Coordination

„A and B's dog and cat…" (again from Herlihy-Shavit intro)

A's protocol:

1. Wait until flag==0: ENTER
2. Set flag=1

B's protocol

1. Wait until flag==1: ENTER
2. Set flag=0

Mutual exclusion: Initially flag is either 0 or 1. Assume it is 0, then only A can enter, and eventually sets flag to 1 upon exit; A will not enter again before flag is 0, so mutually exclusion holds. In order for flag to become 0, B must have exited (and will not enter again before flag is 1), so mutual exclusion holds. There are no other possibilities for flag to change value…

Shared-memory multi-(core-)processor

Homogeneous cores ("processors", P's) that communicate and coordinate through a shared memory; possibly with some hardware support for coordination and synchronization

Each processor has own local program, some own local storage, is not synchronized with oher processors: MIMD or SPMD

- Processors (hardware entities) execute processes, processes execute threads (software entities)
- A process can execute zero or more threads
- Threads are uncoordinated and asynchronous. Execution of (processes and) threads can be arbitrarily interleaved, threads can be preempted, interrupted, …

©Jesper Larsson Träff

Threads are uncoordinated and asynchronous: Execution of threads can be arbitrarily interleaved, threads can be preempted, interrupted, …

A thread cannot make any assumptions about when an action ("event") of another thread will happen

©Jesper Larsson Träff

A thread cannot make any assumptions about when an action ("event") of another thread will happen

… even in the dedicated case where each processor executes only one thread

©Jesper Larsson Träff

- Memory network
- Banked memory

- NUMA: Non-uniform memory access, access time to memory locations may differ per location and per thread

©Jesper Larsson Träff

Writes to memory may be delayed or reordered: One thread may see values from other threads in a different order than written in program

BUT: Possibility to force writes by special flush/memory fence operations

©Jesper Larsson Träff

Caches: Small (Kbytes to Mbytes), fast memory with copies of most recently used locations

- Updates in cache of one processor propagated to caches of other processors (cache coherence)?
- Updates in cache propagated to memory (memory consistency)?

A bit more on when/how/performance issues with caches: Intro. Par. Comp. and HPC lecture

Write buffer: Small buffer storing pending writes (few K)

- Are writes kept in order (and which?)
- When are writes committed to memory (write buffer flush)?

Computer architecture/later this lecture

Modern multi-core processor (TU Wien: "Ceres") ca. 2010

- 4x16 cores, 3.6GHz, 1TB memory
- HW support for 8 threads/core
- Programmer sees 512 threads

- HW thread waiting for memory transfer can be preempted

- Out-of-order execution, branch prediction, prefetching, …

©Jesper Larsson Träff

Thread A

```
a = 0;
…
a = 1;
if (b==0) { <body> }
```

Thread B

```
b = 0;
…
b = 1;
if (a==0) { <body> }
```

Impossible for both threads to execute <body>:

If A executes body it has set `a = 1`, and read `b==0` so B cannot read `a==0`. Vice versa…

(BUT: Can easily happen that neither A nor B executes body)

BUT only if writes to a and b are visible in that order to the other thread (and not delayed): Sequential consistency

©Jesper Larsson Träff

Sequential consistency (informally):

The result of a concurrent execution of a set of threads is the result of some interleaving of the instructions of the threads as written in the threads' programs (program order)

Memory consistency model formalizes what can and cannot be observed in memory for a multithreaded computation

Sequential consistency is the most well-behaved model: Updates to memory are visible (instantaneous? not necessarily) to other threads in the order as executed by the thread

Modern, shared-memory multiprocessors typically do NOT provide sequential consistency!

(write buffers, caches, memory networks, …; and the compiler!)

Hardware memory consistency models, and how to deal with weaker models:

- Weak consistency
- Relaxed consistency
- Processor consistency
- Release consistency
- Total store order
- …

- Memory fences/barriers
- Compiler options

AMP:
We will study this in more detail!

For the "principles": We (mostly) assume sequential consistency

In "practice" make execution sequentially consistent enough by relying on atomic ordering instructions, and inserting memory fences that enforce writes to take effect

Java: **`volatile`**, **`synchronized`** keywords, and other means
C/C++: **`volatile`**, memory fences inserted by hand, C++11 memory model

CAUTION:

- Too few fences make programs incorrect, with bugs that are sometimes very hard to find
- Too many fences make programs slow (cache invalidations, flushing of write buffers, …)

Assumptions:

Reads and writes ("events") are not simultaneous. Events can always be ordered, one before or after the other

Memory updates are valid, well-formed, consistent:

If thread A writes values a, b, c to variable x which already contains d, and thread B reads x, then B will see either a, b, or c, or d, not some mashup/undefined value

AMP:
We will study this in more detail!

Performance and coordination: An example

Task: compute the primes from 1 to $n = 10^9$

p threads.

Idea 1 (trivial parallelization): divide the interval [1..n] evenly among the p threads, each thread checks for primes in own interval

```
int i = ThreadID.get(); // get local thread ID
block = n/p; // each thread checks a block of integers
for (j=i*block+1; j<(i+1)*block; j++) {
  if (isPrime(j)) { <take action> }
}
```

This AMP lecture: Pseudo-Java

©Jesper Larsson Träff

Drawbacks:

1. Primes are not evenly distributed. Prime number theorem(*) says many more primes in [1…n/p] than in [(p-1)*n/p+1…n]

2. Time for `isPrime()` varies (fast for non-primes with small prime factors, slow for large primes)

Thus: No reason to expect good load-balance, some task my be unnecessarily idle

(*) Number of primes smaller than x approx. x/ln x

Idea 2 (coordination): shared "work pool", each thread gets next integer to check from work pool

Use a shared counter to manage work pool

```
class Counter {
  private int value;
  public Counter(int c) { // constructor
    value = c;
  }

  public int getandinc() {
    return value++;
 }
}
```


©Jesper Larsson Träff

```
Counter counter = new Counter(1);

int i = 0;
while (i<n) {
  i = counter.getandinc();
  if (isPrime(i)) { <take action> }
}
```

Does this work?

©Jesper Larsson Träff

```
return value++;
```

compiles into (something like)

```
int temp = value;
value = temp+1; // may itself be several instructions
return temp;
```

Thread 0

```
temp = value;


value = temp+1;
return temp;
```

Thread 1

```
temp = value;


value = temp+1;
return temp;
```

time

Both threads return the same value!

…and even worse

Thread 0

```
temp = value;

value = temp+1;
return temp;
```

Obsolete value returned

Thread 1

```
temp = value;
value = temp+1;
return temp;

temp = value;
value = temp+1;
return temp;

temp = value;
value = temp+1;

return temp;
```

All increments by thread 1 lost

Classical solution:
Encapsulate the dangerous increment in "critical section", only one thread at a time can be in critical section and perform increment: Mutual exclusion property. Enforce m.e. by locking:

```
public interface Lock {
  public void lock(); // enforce mutual exclusion
                      // acquire lock
                      // enter critical section
  public void unlock(); // leave critical section
                        // release lock
}
```

A thread acquires the lock by executing `lock();` at most one thread can hold the lock at a time; a thread releases the lock by `unlock();`

"Atomic" counter with locks

```
class Counter {
  private int value;
  private Lock lock; // use lock for CS
  public Counter(int c) { // constructor
    value = c;
  }

  public int getandinc () {
    int temp;
    lock.lock(); // enter CS
    try {
      temp = value; // increment alone
      value = temp+1;
      return temp;
    } finally {
      lock.unlock(); // leave CS
    }
  }
}
```

AMP:
We will study this in more detail!

Problems:

- How can lock/unlock be implemented?
- What do locks provide? Will a thread trying to acquire the lock eventually get the lock?
- Are locks a good programming mechanism? Sufficient?

Properties:
Correctness/safety: Lock must guarantee mutual exclusion, at most one thread at a time in critical section
Liveness:
Deadlock freedom: If some thread tries to aquire lock, then some (other) thread will acquire lock. If a thread does not succeed, then other threads must be succeeding infinitely often
Starvation freedom: A thread trying to acquire the lock will eventually get the lock

The problems with locks

Thread 0

```
lock.lock();

<long and
complicated update
of shared data
structure (FIFO,
Stack, list,
Priority queue, hash
map, …>
```

Thread 1

```
lock.lock(); // will idle
```

Threads waiting for lock make no progress

Locks/critical sections easily become a sequential bottleneck.
Bad: Amdahl's law

Possible to do better for specific data structures?

What if `lock.unlock();` forgotten?
What if Thread 0 fails? Or thread is preempted indefinitely?

The problems with locks: Assume Lock is a perfectly good lock (correct, fair, fast, …) that is used correctly in different parts of the program

Thread 0

```
Lock lock1, lock2;

lock1.lock();

lock2.lock();
… // work here
lock2.unlock();
lock1.unlock();
```

CORRECT (for some reason, two locks are needed)

The problems with locks: Assume `Lock` is a perfectly good lock (correct, fair, fast, …) that is used correctly in different parts of the program

CORRECT (for some reason, two locks are needed)

Thread 1

```
Lock lock1, lock2;

lock2.lock();

lock1.lock();
… // work here
lock1.unlock();
lock2.unlock();
```


©Jesper Larsson Träff

The problems with locks: Assume `Lock` is a perfectly good lock (correct, fair, fast, …); put together correctness is lost

Thread 0

```
Lock lock1, lock2;

lock1.lock();


lock2.lock();
```

Thread 1

```
Lock lock1, lock2;

lock2.lock();

lock1.lock();
```

DEADLOCK!

Locks are error-prone, non-modular: Programs for Thread 0 and Thread 1 could have been written at different times by different programmers following different conventions…

©Jesper Larsson Träff

Hardware based solution (to prime computation):

Provide `getandinc(&value);` as special instruction that reads and increments the counter (value) in one atomic, indivisible step

Perfect solution: No locks, each available thread will immediately get next value to work on, no waiting/idling

But global resource (counter)!

Questions:

- Are such instructions possible?
  - YES (see advanced computer architecture book, getandinc, CAS, …)
- Are such instructions equally powerful?
  - NO! These lectures
- How can such instructions be used to provide better data structure support than locks? We will see

Mutual exclusion (Chap. 2)

- Some machinery to reason about correctness
- Two classical solutions with "registers" (memory locations)
- An impossibility result

<u>Assumption</u>:
Atomic registers (see later) or sequential consistency:
Reads and writes are totally ordered events, writes to memory locations (registers) appear in program order, what a thread reads has been written by some other thread previously

Newtonian (not Einsteinian) time:

There is a common, global time against which the actions of the threads can be ordered

Note: threads normally cannot refer to this time, and will not have to. Time is not an accessible, global clock/timer.

Actions (updates of variables, reading of variables: global state changes) by threads are called events. Events are instantaneous. Events can be ordered temporally, no two events take place at the same time, either is before or after the other: Total order

Historically:
Events affect special variables called registers. Updates to registers by one thread are seen in that order by other threads

j'th occurrence of event i

Total "precede" order on events, e -> f:

- Not a -> a (irreflexive)
- If a -> b, then not b -> a (antisymmetric)
- If a -> b and b -> c, then a -> c (transitive)

- Either a -> b, or b -> a (total)

If b0 -> b1, then interval B = (b0,b1) is the duration between b0 and b1

A = (a0,a1), B = (b0,b1), then A -> B iff a1 -> b0

Partial "precede" order on intervals:

- Not A -> A (irreflexive)
- If A -> B, then not B -> A (antisymmetric)
- If A -> B and B -> C, then A -> C (transitive)

- But NOT (either A -> B or B -> A): intervals may overlap and be concurrent (interval order is partial)

- A1 -> A2
- Not A0 -> A1, and not A1 -> A0: Intervals A0 and A1 concurrent
- A0 -> B0
- B0 -> B1, implies A0 -> B1
- Not A1 -> B0, and not B0 -> A1: Intervals A1 and B0 concurrent

©Jesper Larsson Träff

Critical Section (CS):

A section of code that can be executed by only one thread at a time: An interval (code) that must not be executed concurrently with certain other intervals (same code).

A program can want to execute CS many times. A program can have different CS, CS', CS"; different critical sections may be executed by different threads concurrently (but each by at most one thread)

E. W. Dijkstra: Solution of a problem in concurrent programming control. Comm. ACM, 8(9), p. 569, 1965.
E. W. Dijkstra: Co-operating sequential processes. In: Programming languages, pp. 43-112. Academic Press, 1965

Let CS(A,k) be the interval in which thread A is in the critical section CS for the k'th time

<u>Mutual exclusion property</u>:
For any threads A and B, and any j, k, either CS(A,k) -> CS(B,j), or CS(B,j) -> CS(A,k): critical section intervals are never concurrent

Challenge: Devise lock-algorithms (lock-objects) that can enforce mutual exclusion, and are

- Correct: At most one thread executes CS (intervals never concurrent, mutual exclusion property)
- Deadlock free: If some thread tries to execute CS, some thread will succeed
- Starvation free: If a thread tries to execute CS it will eventually succeed
- Resource efficient: As few registers as possible

Let CS(A,k) be the interval in which thread A is in the critical section CS for the k'th time

<u>Mutual exclusion property</u>:
For any threads A and B, and any j, k, either CS(A,k) -> CS(B,j), or CS(B,j) -> CS(A,k): critical section intervals are never concurrent

©Jesper Larsson Träff

<u>Mutual exclusion property</u>:
For any threads A and B, and any j, k, either CS(A,k) -> CS(B,j), or CS(B,j) -> CS(A,k): critical section intervals are never concurrent

©Jesper Larsson Träff

Definitions (dependent or blocking liveness properties):

An operation on a shared object is

- Deadlock free, if when some threads tries to execute CS, some (possibly other) thread will succeed. Conversely, if a thread never succeeds, other threads will succeed infinitely often
- Starvation free, if when a thread tries to execute CS it will eventually succeed,

provided that (all) threads takes steps.

Observation: Starvation freedom implies deadlock freedom

The Peterson Lock

A lock for mutual exclusion on 2 threads

Combination of two ideas. "Building block" for generalization to n threads

G. Peterson: Myths about the mutual exclusion problem.
Information Processing Letters, 12(3): 115-116, 1981

Idea 1: Each thread has an own flag to indicate that it wants to enter CS; check that other thread is not entering before entering

```
class Lock1 implements Lock {
  private boolean[] flag = new boolean[2];

  public void lock() {
    int i = ThreadID.get(); // get own thread id
    int j = 1-i; // other thread
    flag[i] = true;
    while (flag[j]) {} // wait for other thread
  }

  public unlock() {
    int i = ThreadID.get();
    flag[i] = false;
  }
}
```

i either 0 or 1

<u>Lemma:</u> Lock1 satisfies mutual exclusion

Proof: By contradiction.

Assume there exists two concurrent CS intervals, i.e., not(CS(A,j) -> CS(B,k)) and not(CS(B,k) -> CS(A,j)).

Consider A's and B's last execution of `lock()` before entering CS(A,j) and CS(B,k)

write(A,flag[A]=true) -> read(A,flag[B]==false) -> CS(A,j)
write(B,flag[B]=true) -> read(B,flag[A]==false) -> CS(B,k)

write(A,flag[A]=true) -> read(A,flag[B]==false) -> CS(A,j)
write(B,flag[B]=true) -> read(B,flag[A]==false) -> CS(B,k)

Also:

read(A,flag[B]==false) -> write(B,flag[B]=true)

since once flag[B] is set to true it remains true, and A could not have read flag[B]==false. By transitivity

write(A,flag[A]=true) -> read(B,flag[A]==false)

Contradiction!

BUT:
Deadlocks if the execution of the two threads is interleaved

write(A,flag[i]=true) -> write(B,flag[j]=true) ->
read(A,flag[j]==true) -> read(B,flag[i]==true) -> FOREVER

Both threads are stuck in while loop repeating the read-events, since no further write events can happen

Idea 2: Use one variable to trade entry to the lock ("victim" for some historical reasons, intuition is "not me" or "you first")

```
class Lock2 implements Lock {
  private volatile int victim;

  public void lock() {
    int i = ThreadID.get();
    victim = i;
    while (victim==i) {} // wait for other thread
  }

  public unlock() {}
}
```


©Jesper Larsson Träff

<u>Lemma:</u> Lock2 satisfies mutual exclusion

Proof: By contradiction. Again, consider A's and B's last execution of `lock()` before entering CS

write(A,victim=A) -> read(A,victim==B) -> CS(A,j)
write(B,victim=B) -> read(B,victim==A) -> CS(B,k)

For A to read victim==B it must be that

write(A,victim=A) -> write(B,victim=B)

since this assignment is the last write by B. And since B now reads, and there are no other writes, this read cannot return victim==A. Contradiction!

BUT:
Deadlocks if one thread runs completely before the other. The lock depends cooperation by the other thread

Thread A

```
lock();



unlock(); return;
```

Thread B

```
lock(); unlock();
lock(); // thread hangs
```


©Jesper Larsson Träff

The Peterson lock combines the two ideas and overcomes deadlock

```
class Peterson implements Lock {
  private boolean[] flag = new boolean[2];
  private volatile int victim;

  public void lock() {
    int i = ThreadID.get();
    int j = 1-i; // other thread
    flag[i] = true;
    victim = i;
    while (flag[j] && victim==i) {} // wait
  }
  public unlock() {
    int i = ThreadID.get();
    flag[i] = false;
  }
}
```

<u>Proposition:</u> The Peterson lock satisfies mutual exclusion

Proof: By contradiction. Look at what happens the last time A and B enter CS:

write(A,flag[A]=true) -> write(A,victim=A) -> read(A,flag[B]) -> read(A,victim) -> CS(A,j)

write(B,flag[B]=true) -> write(B,victim=B) -> read(B,flag[A]) -> read(B,victim) -> CS(B,k)

at the moment not knowing the values read for victim and flag.

If A (wlog) was the last to write victim, then

write(B,victim=B) -> write(A,victim=A)

For A to be in CS, it must have read flag[B]==false, so

write(A,victim=A) -> read(A,flag[B]==false)

Here we use atomicity/sequential consistency: events occur/appear in program order

By transitivity

write(B,flag[B]=true) -> write(B,victim=B) -> write(A,victim=A) -> read(A,flag[B]==false)

Contradiction, since there were no other writes to flag[B]

<u>Proposition:</u> The Peterson lock is starvation free

Proof: By contradiction. Assume thread A is waiting forever in `lock();` it waits for either flag[B]==false or victim==B.

3 cases:
Either B is outside of CS, or B is repeatedly entering CS. Or B is also waiting in `lock();`

- B outside of CS: flag[B]==false. Contradiction.
- B reentering CS: it sets victim=B, so A can enter. Contradiction.
- B waiting in `lock();`: victim cannot be both ==A and ==B at the same time, so either thread must enter. Contradiction.

<u>Corollary:</u> The Peterson lock is deadlock free

Proof:
Starvation freedom always implies deadlock freedom

©Jesper Larsson Träff

The filter Lock

Extending the idea to n threads. Let there be n-1 locking levels. At each level, roughly

- At least one thread trying to enter level j succeeds
- If more than one thread is trying to enter level j, one is blocked

The Boolean `flag[2]` is replaced by `level[n]`: the level at which thread i is trying to enter. For each level j there is a `victim[j]` keeping track of the last thread that did not enter level j

```
class Filter implements Lock {
  private int[] level;
  private int[] victim;

  public Filter(int n) {
    level = new int[n];
    victim = new int[n];
    for (i=0; i<n; i++) level[i] = 0;
  }
```

Object constructor…

```
class Filter implements Lock {
  private int[] level;
  private int[] victim;

  public void lock() {
    int i = ThreadID.get();
    for (int j=1; j<n; j++) { // try to enter level j
      level[i] = j;
      victim[j] = i;
      while (EXIST k≠i:level[k]>=j && victim[j]==i);
  }

  public unlock() {
    int i = ThreadID.get();
    level[i] = 0; // through the filter
  }
}
```

Note:
Checking n level values and a victim is not assumed to be done atomically

```
while (EXIST k≠i:level[k]>=j && victim[j]==i);
```

is shorthand for

```
for (k=0; k<n; k++) {
  if (k==i) continue;
  while (level[k]>=j && victim[j]==i);
}
```

level n-1: 1 thread in CS

Intuition:

At most n-j threads can complete level j, j≥1, and proceed to level j+1.

At each level, any thread attempting CS will eventually succeed (proceed to next level)

Note:
For n=2, Filter lock equivalent to two-thread Peterson

©Jesper Larsson Träff

Proposition: The Filter lock satisfies mutual exclusion

Proof:
Thread i has reached level j when it starts iteration j of the for loop, and has completed level j when it exits

```
while (EXIST k≠i: level[k]≥j && victim[j]==i);
```

When thread i completes level j, it can reach level j+1. By induction, we show that for j, $0 \leq j < n$, at most n-j threads can complete level j.

Observe that j=n-1 implies mutual exclusion in CS

Induction hypothesis: At most n-j threads can complete level j.

Base case, j=0, trivial, the n threads not in CS

Assume hypothesis, at most n-(j-1) threads have completed level j-1, j≥1. At level j, at most n-(j-1) = n-j+1 threads enter. Have to show that at least one thread cannot complete level j. Assume the contrary: All n-j+1 threads complete level j.

Let A be the last thread to write `victim[j]`, so for any other B:

write(B,level[B]=j) ->
write(B,victim[j]=B) -> write(A,victim[j]=A)

B has now reached level j or higher, so A will read level[B]>=j and therefore cannot complete level j. Contradiction.

<u>Proposition:</u> The Filter lock is starvation free

Proof:
Will have to show that any thread that wants to enter CS will eventually succeed. By induction on levels in reverse order

Induction hypothesis: Every thread that enters level j or higher, will eventually enter CS

Base case, level j=n-1 contains only one thread, in CS

Assume thread A remains stuck at level j. Thus victim[j]==A and level[B]≥j for at least one B.

Two cases:

- Some thread B sets `victim[j] = B`. Since A is the only thread that can set `victim[j] = A`, this contradicts that A is stuck

- No thread sets `victim[j] = B`. Thus, there is no thread entering level j (such B would set victim[j] = B) from below. By the induction hypothesis, all threads at level j or higher will eventually enter CS. When this happens, eventually level[B] becomes smaller than j (first reset to 0), contradicting that level[B]≥j, so A can enter level j+1

Corollary: The Filter lock is deadlock free

(since starvation freedom implies deadlock freedom)

However:

Proposition: The Filter lock is not fair (what does that mean?)

Exercise: Show that threads can overtake up to a certain number of times

Peterson and Filter locks have nice properties (mutual exclusion correctness, starvation free)

BUT:

1. 2n shared integer variables for n-thread mutual exclusion
2. Weak liveness properties (starvation only means "eventually")

Can this be improved?

Aside: Dekker's algorithm (as quoted by Dijkstra, ca. 1965)

```
class Dekker implements Lock {
  private Boolean[] flag = new Boolean[2];
  private volatile int turn = 0;

  public void lock() {
    int i = ThreadID.get(); // get own thread id
    int j = 1-i; // other thread
    flag[i] = true;
    while (flag[j]) {
      if (turn!=i) {
        flag[i] = false;
        while (turn!=i) {}
        flag[i] = true;
      }
    }
  }
```


©Jesper Larsson Träff

Home exercise: Prove correctness and liveness of Dekker's algorithm

```
class Dekker implements Lock {
  private Boolean[] flag = new Boolean[2];
  private volatile int turn = 0;

  public unlock() {
    int i = ThreadID.get();
    turn = 1-i;
    flag[i] = false;
  }
}
```

Similar to Peterson; according to Dijkstra, the(?) first correct solution to the mutual exclusion problem (before 1965)

Aside: Mutual exclusion and bypassing

Peterson's/Dekker's algorithms makes it possible for processes to overtake each other (see exercises); this can be avoided, e.g.,

K. Alagarsamy: A mutual exclusion algorithm with optimally bounded bypasses. Inf. Process. Lett. 96(1): 36-40 (2005)

Lamport's Bakery algorithm

A mutual exclusion with stronger fairness guarantees (first-come-first-served, FIFO, …)

Idea: Take a ticket that is larger than the ones already in the bakery (or having been served); wait until my ticket is smallest

L. Lamport: A new solution of Dijkstra's concurrent programming problem. Comm. ACM, 17(5): 543-545, 1974
See also: research.microsoft.com/en-us/um/people/lamport

Note: The "Bakery algorithm" in book and here is NOT quite Lamport's Bakery algorithm, more like this:

G. L. Peterson: Observations on l-exclusion. Allerton Conf. on Communication, Control and Computing, pp.568-577, 1990.

Array of n flags for each thread to signal need to enter CS; array of n labels for Bakery tickets

```
class Bakery implements Lock {
  private boolean[] flag;
  Label[] label; // unbounded integer label
  public Bakery(int n) {
    flag = new boolean[n];
    label = new Label[n];
    for (int i=0; i<n; i++) {
      flag[i] = false; label[i] = 0;
    }
  }
  …
}
```


©Jesper Larsson Träff

```
class Bakery implements Lock {
 private boolean[] flag;
  Label[] label; // unbounded integer label
  …
  public void lock() {
    int i = ThreadID.get();
    flag[i] = true;
    label[i] = max(label[0],…,label[n-1])+1;
    while (EXIST k≠i:
           flag[k] &&
           (label[k],k)<<(label[i],i)) { }
  }

  public unlock() {
    flag[ThreadID.get()] = false;
  }
}
```

Take ticket

Wait till smallest

©Jesper Larsson Träff

Both computations can be done in arbitrary order, e.g.,

```
max(label[0],label[1],…,label[n-1]);
```

as

```
max = label[0];
for (k=1; k<n; k++) if (max<label[k]) max = label[k];
```


©Jesper Larsson Träff

Observations:

The sequence of labels for each thread are strictly increasing

Two (or more) threads trying to acquire the lock may generate the same label. Need to break ties between such threads:

Standard tie-breaking idea: Use thread id

Note:
The label computation is not atomic ("snapshot"); a thread may use labels of others set at different times, thus compute its label from a set of labels that were never in memory at the same time…

Rachel Ruysch (1664-1750),
Akad. Bild. Künste, Wien

Note:
The label computation is not atomic ("snapshot"); a thread may use labels of others set at different times, thus compute its label from a set of labels that were never in memory at the same time…

```
(label[k],k)<<(label[i],i)
```

means lexicographic order, holds iff

```
label[k]<label[i]
```

or

```
label[k]==label[i] && k<i
```

Standard tie-breaking scheme:
If two threads have the same label, the thread with smaller ID "wins"

```
while (EXIST k≠i:flag[k]&&(label[k],k)<<(label[i],i));
```

is not supposed to be atomic, and can be implemented as

```
for (k=0; k<n; k++) {
  if (k==i) continue;
  while (flag[k]&&
         (label[k]<label[i]||
         (label[k]==label[i]&&k<i)));
}
```

<u>Proposition:</u> The Bakery lock is deadlock free

Proof: Assume all threads waiting to enter CS. No labels change. There is a unique least `(label[A],A)` pair. The corresponding thread can acquire the lock. Contradiction

©Jesper Larsson Träff

Proposition: The Bakery lock satisfies mutual exclusion

Proof: By contradiction. Assume threads A and B in critical section, and let labeling(A), labeling(B) denote the (non-atomic) sequences of instructions generating the labels. Assume (wlog) that (label[A],A)<<(label[B],B). When B entered it must therefore have read flag[A]==false, so

labeling(B) -> read(B,flag[A]==false) -> write(A,flag[A]=true) -> labeling(A)

which contradicts (label[A],A)<<(label[B],B), since A's label would be at least label[B]+1

Note:
Even though the labeling(A) steps are not atomic, the algorithm is correct (satisfies mutual exclusion).

Even if two threads compute the same `label[i]`, they will be strictly ordered (lexicographically), e.g.,

write(A,flag[A]=true) -> read(A,label) -> write(B,flag[B]=true) -> read(B,label) -> write(A,label[A]) -> read(A,flag[B]==true) -> read(A,label[B]) ->
write(B,label[B]) -> read(B,flag[A]==true) -> read(B,label[A]) ->
… -> read(A,label[B]) -> CS(A,j)

assuming (wlog) that A<B

Fairness: Informally, would like that if A calls `lock();` before B then B cannot overtake A, that is B cannot enter the critical section before A

Divide the lock method into two sections:

- A doorway, whose execution interval D has a bounded number of steps (greater than one)
- A waiting room, whose execution interval W may be unbounded

Definition: A lock is first-come-first-served if, whenever thread A finishes its doorway before thread B starts its doorway, then A cannot be overtaken by B:

If D(A,j) -> D(B,k) then CS(A,j) -> CS(B,k)

```
class Bakery implements Lock {
  private boolean[] flag;
  Label[] label; // unbounded integer label
  …
  public void lock() {
    int i = ThreadID.get();
    flag[i] = true;
    label[i] = max(label[0],…,label[n-1])+1;
    while (EXIST k≠i:
           flag[k] &&
           (label[k],k)<<(label[i],i)) { }
  }
  public unlock() {
    flag[ThreadID.get()] = false;
  }
}
```

Doorway

Waiting room

©Jesper Larsson Träff

<u>Proposition:</u> The Bakery lock is first-come-first-served

Proof:
If D(A) -> D(B), then label[A]<label[B] since

write (A,flag[A]) -> write(A,label[A]) ->
read(B,label[A]) -> write(B,label[B]) -> read(B,flag[A]==true)

so B must wait as long as flag[A]==true, since label[B]≥label[A]+1

Corollary: The Bakery lock is starvation free (because any algorithm that is both deadlock free and FCFS is starvation free: Exercise)

Note: Bakery doorway takes $\Omega(n)$ operations

Bakery lock has nice properties (mutual exclusion correctness, first-come-first-served)

BUT:

1. 2n shared integer variables for n-thread mutual exclusion
2. Labels grow without bounds

Can this be improved?

Ad 2: It is possible to construct a bounded, concurrent, wait-free timestamping system, such that a thread can read the other threads time stamp and assign itself a later timestamp. A sequential solution is described in Herlihy/Shavit book.

Bakery lock has nice properties (mutual exclusion correctness, first-come-first-served)

BUT:

1. 2n shared integer variables for n-thread mutual exclusion
2. Labels grow without bounds

Can this be improved?

Ad 2: There are Bakery-like algorithms where labels are bounded, e.g. Black-White Bakery

G. Taubenfeld: The black-white bakery algorithm. Proc. DISC. LNCS 3274, pp. 56-70, 2004

Other remarks:

1. Filter lock: all threads write to victim[i]
2. Bakery: each location flag[i] and label[i] written by only one thread but read by many)
3. Both: even in the absence of contention $\Omega(n)$ locations are read

1-2: Bakery can do with MRSW (Multiple Readers, Single Writer – see later) (atomic) registers, Filter requires MRMW

3: Some „fast path" mutual exclusion algorithms exist

L. Lamport: A fast mutual exclusion algorithm. ACM TOCS 5(1): 1-11, 1987

Aside: The original Bakery (from Taubenfeld)

```
public void lock() {
  int i = ThreadID.get();
  flag[i] = true;
  label[i] = max(label[0],…,label[n-1])+1;
  flag[i] = false;
  for (int j=0; j<n; j++) {
    while (flag[j]);
    while (label[j]>0 && (label[j],j)<<(label[i],i)));
  }
}
public unlock() {
  label[ThreadID.get()] = 0;
}
```

- Labels can still grow unboundedly
- More complicated correctness proof
- Careful with maximum computation, can break correctness

Exercise

Maximum (correct)

```
max = 0; k = 0;
for (i=0; i<n; i++) {
  m = label[i];
  if (max<m) max = m;
}
```

Maximum (wrong)

Home exercise

```
k = 0;
for (i=0; i<n; i++) {
  if (label[k]<label[i]) k = i;
}
max = label[k];
```


©Jesper Larsson Träff

Aside: Implementing register based mutual exclusion

For correct implementation of the register locks in C/C++, correct event ordering and data race freedom (see later) must be ensured. Technically, necessary to use, eg.

```
#include <stdatomic.h>

atomic_int flag[2];
atomic_int victim;
```


©Jesper Larsson Träff

Theorem:
Any deadlock-free algorithm that solves mutual exclusion for n threads by reading and writing memory locations (registers) must use at least n distinct locations

Problem with registers: Any value written by a thread can be overwritten, without other threads seeing the first value

Filter and Bakery are thus optimal (within a factor 2)

Exercise: Reduce number of locations to n for the Bakery algorithm

To do better, stronger mechanisms are needed: "hardware support" (later lecture)

Observations for any mutual exclusion algorithm with registers:

1. Any thread A entering CS must write a value to at least one register, otherwise other threads would have no way of determing that A is in CS.

2. If only single-writer registers are used (as in Bakery), at least n such are required

©Jesper Larsson Träff

System state: state of all threads and all objects

Object state: state (values) of all object fields

Local thread state: state of all local variables and program counter

©Jesper Larsson Träff

Proof (2 threads):
By contradiction, assume 1 register x suffices for 2 threads.
Find a scenario where both threads A and B can enter CS.

There is a "covering state" where thread A is about to write to x, but the lock (state) still looks as if no thread is in or trying to enter CS.

Let A be in the covering state, let B run, and enter CS (it can, because the algorithm satisfies mutual exclusion and deadlock freedom). Now, A is resumed, writes x – overwriting whatever B has written to x. Since there is no trace of B anymore in x, A can likewise enter CS. Contradiction!

Base case for induction proof for n threads (not here, but)…

Proof sketch (3 threads):
By contradiction, assume 2 locations suffice for 3 threads.

Claim: There is a "covering state" where thread A and B are about to write to the two locations, but the lock still looks as if no thread is in or trying to enter CS.

Let thread C run alone and enter CS. Then let A and B continue, each updates each of the locations, overwriting what C wrote. Now, neither A nor B can determine that C is already in CS.

A or B could enter CS, contradicting mutual exclusion.

Locations: R1, R2. LA either R1 or R2 (the location A is about to write to), likewise LB is either R1 or R2

"Covering state": Both locations about to be written, but still possible for thread C to enter CS

©Jesper Larsson Träff

"Covering state": Consider execution where B runs through CS 3 times, and consider the first location that B writes to. Since there are only 2 locations, there must be such a first location that is written two times. Call this LB

Let B run to the point just before it writes LB for the first time. This is a state where both A and C can possibly enter CS. If A runs now, it can enter CS since B has not yet written. Let A run to the point just before it writes the other location LA (if it wrote only LB, then B would overwrite, and B would not be able to tell that A is in CS).

Still, A could have written to LB before writing to LA (which could have effect for C). Let B run through CS at most three times until it again covers LB. Then it would have overwritten whatever A might have written to LB, and be in the state where both A and C (and B) could enter CS

Covering state: A and B about to write LA and LB, but no thread in or entering CS

©Jesper Larsson Träff

Proof (n threads):

By induction, it is possible to find a covering state that covers k registers, $k \leq n-1$ …

Full proof (combinatorial "covering argument"):
See Nancy Lynch: Distributed Algorithms. Morgan-Kaufman 1996.

Burns, Lynch: Bounds on shared memory for mutual exclusion. Information&Computation 107(2): 171-184, 1993.

Black-white bakery lock

A (simple) solution to the problem of Lamport's bakery algorithm with unbounded labels

Idea: tickets (label's) are colored. A thread that wants to enter CS takes a colored ticket, and waits until it is smallest with that color. Algorithm needs to maintain a global color (bit).

G. Taubenfeld: The black-white bakery algorithm. DISC, LNCS 3274, 56-70, 2004

```
class BlackWhite implements Lock {
  private boolean[] flag;
  Label[] label; // bounded integer label
  Color[] color; Color c;// global color
  public Bakery(int n) {
    flag  = new boolean[n];
    color = new Color[n];
    label = new Label[n];
    c = white;
    for (int i=0; i<n; i++) {
      flag[i] = false; label[i] = 0; color[i] = white;
    }
  }
  public void lock() {
    …
  }
  public unlock() {
    …
  }
}
```


©Jesper Larsson Träff

```
public void lock() {
  int i = ThreadID.get();
  flag[i] = true;
  color[i] = c; // set own color from global color
  label[i] =
    max(label[j] | 0<=j<n&&color[j]==color[i])+1;
  flag[i] = false;

  for (j=0; j<n; j++) {
    while (flag[j]);
    if (color[j]==color[i]) {
      while (label[j]>0 && (label[j],j)<<(label[i],i) &&
             color[j]==color[i]);
    } else {
      while (label[j]>0 &&
             color[i]==c && color[j]!=color[i]);
    }
  }
}
```

```
public unlock() {
  int i = ThreadID.get();
  if (color[i]==black) c = white; else c = black;
  label[i] = 0;
}
```


©Jesper Larsson Träff

3 threads, all wants to enter CS

©Jesper Larsson Träff

Theorem: Black-white bakery algorithm

- satisfies mutual exclusion
- is first-come, first-serve
- uses only bounded registers (label[i] at most n, for n threads)

Proof: Exercise (or Taubenfeld book/paper)

Other bounded register fair locks:

Alex A. Aravind, Wim H. Hesselink: Nonatomic dual bakery algorithm with bounded tokens. Acta Inf. 48(2): 67-96 (2011)
Alex A. Aravind: Yet Another Simple Solution for the Concurrent Programming Control Problem. IEEE Trans. Parallel Distrib. Syst. 22(6): 1056-1063 (2011)
Boleslaw K. Szymanski: A simple solution to Lamport's concurrent programming problem with linear wait. ICS 1988: 621-626
Prasad Jayanti, King Tan, Gregory Friedland, Amir Katz: Bounding Lamport's Bakery Algorithm. SOFSEM 2001: 261-270

Advanced Multiprocessor Programming

Jesper Larsson Träff
traff@par.tuwien.ac.at
Research Group Parallel Computing
Faculty of Informatics, Institute of Information Systems
Vienna University of Technology (TU Wien)

©Jesper Larsson Träff

The power of registers (Chap. 4)

Hardware resource ("register" for some historical reasons; not every memory location might have the desired properties) or just memory location that can be read and written by different threads

Single/Multiple Readers/Writers

- SRSW
- MRSW
- SRMW
- MRMW

M-valued register: Can store values 0, … M-1. Special case: Boolean register, two values (**true/false**)

Mutual exclusion constructions have assumed "atomic" MRMW registers: Totally ordered, instantaneous read/write "events"

Question: Are "atomic" registers reasonable? Is it possible to construct stronger registers out of weaker ones? How weak can the weaker registers be?

Plan: Construct atomic registers out of weak ones such that operations (intervals) look like events

Constructions must be wait-free: Any thread can complete any read/write in a finite (bounded) number of own operations, independently of the operations of other threads

Sequential register:
A single thread performs read and write operations. There is trivially a total order on the operations: Program order. Each read operation returns the value written by the most recent write operation

x=0, y=9 (not y=7, not y=0)

©Jesper Larsson Träff

SRSW: A single thread reads, and a(nother) single thread writes; but reads and writes can overlap (partial order between read and write operations)

Read x does not overlap any write; read y etc. overlap write's

©Jesper Larsson Träff

MRSW: Many threads read, and a single thread writes; and reads and writes can overlap (partial order on read and write operations)

©Jesper Larsson Träff

Definition:

A MRSW is safe if a read interval that does not overlap a write interval returns the most recently written value; a read interval that is concurrent with a write interval may return any value in the register's range 0, …, M-1

x=0, y=?? because Read y overlaps Write 23

©Jesper Larsson Träff

Definition:

A MRSW is safe if a read interval that does not overlap a write interval returns the most recently written value; a read interval that is concurrent with a write interval may return any value in the register's range 0, …, M-1

x=??, y=?? (but ?? is in the range of the register)

©Jesper Larsson Träff

Definition:

A MRSW is regular if a read interval that does not overlap a write interval returns the most recently written value; a read that overlaps one or more writes returns either the most recently written (preceding) value or any of the values being written by the concurrent (overlapping) writes

x=either of 0,9,1,3; y=either of 9,1,3,23

©Jesper Larsson Träff

Note:
In these definitions, safe and regular registers are always SW

©Jesper Larsson Träff

Definition:

A MRMW register is atomic if reads and writes look as if they are totally ordered events where each read operation returns the value written by the most recent write operation:

- Each read and write operation appears to have taken effect at some instant (event) between the start and the end of the operation's interval
- The resulting sequence of read/write operations ordered by the instants in which they take effect could have been executed by a single thread on a sequential register, that is, each read returns the value of the most recent write

Notes:

The instants where an operation "takes effect" or "becomes visible" are called linearization points (next lecture)

An atomic register is such that any history (next lecture) generated with that register is linearizable, that is equivalent to a (correct) sequential execution

©Jesper Larsson Träff

Write order:

Let W0, W1, …, Wi be the write operations on a register, with total write order => such that Wi => W(i+1)

For SW registers, the write order is the interval order ->, since there are no overlapping writes (in particular, safe and regular registers). For MW registers, impose order by the time instant in which a write "takes effect"

Name the read operations, such that Ri is the read operation that returns value x==xi that was written by Wi

Note: It is assumed (wlog) that each Wi writes a unique value. Each Wi is unique, but there can be several read operations Ri, R'i, … that returns value xi written by Wi

A MRSW register is regular iff

- Not Ri -> Wi: A read never returns a value from the future
- Not Wi => Wj -> Ri: A read never returns a(n overwritten) value from the distant past

©Jesper Larsson Träff

Definition:
A MRMW is atomic if it behaves as if reads and writes are totally ordered events such that each read returns the most recently written value

Not:
y==9, x==1

A register is atomic iff it is regular and additionally

- If Ri -> Rj, then i≤j: An earlier read must return a value that was written no later than the value returned by a later read

To show that a register is atomic: determine the write order, and verify the atomic conditions

©Jesper Larsson Träff

Definition:
A MRMW is atomic if it behaves as if reads and writes are totally ordered events such that each read returns the most recently written value

Not:
y==9, x==1

"linearization point": write becomes "visible" here, x==9, or (x==1 and y==1)

©Jesper Larsson Träff

Definition:
A MRMW is atomic if it behaves as if reads and writes are totally ordered events such that each read returns the most recently written value

Not:
y==9, x==1

"linearization point": write becomes "visible" here, x==9, and y==1 or y==9

©Jesper Larsson Träff

Definition:
A MRMW is atomic if it behaves as if reads and writes are totally ordered events such that each read returns the most recently written value

Linearization points determine total write order:
Write 6 => Write 9 => Write 2 => Write 1

©Jesper Larsson Träff

Definition:
A MRMW is atomic if it behaves as if reads and writes are totally ordered events such that each read returns the most recently written value

Not:
y==9, x==1

Theorem:
Atomic MRMW are no more powerful than safe SRSW

Proof: Construct atomic MRMW out of a number of safe SRSW as in the following slides (with a large number of registers)

©Jesper Larsson Träff

The construction

1. SRSW safe
2. MRSW safe (Boolean)
3. MRSW Boolean regular
4. MRSW regular
5. SRSW atomic (from SRSW regular)
6. MRSW atomic
7. MRMW atomic

©Jesper Larsson Träff

Safe SRSW to safe MRSW

Construction:

```
Array c[n] of safe SRSW
read(thread i): return c[i]
write(x): for (i=0; i<n; i++) c[i] = x;
```

<u>Lemma</u>: The construction is a safe MRSW

Proof: Each thread reads only a single register c[i]. If a read does not overlap write, it returns the most recently written value; if a read overlaps a write it may return any value, either an old c[i], a new c[i] just written, or if the write to c[i] overlaps the read of c[i], any value since c[i] is safe.

Safe SRSW to safe MRSW

©Jesper Larsson Träff

Safe Boolean MRSW to regular Boolean MRSW

Observation: A regular Boolean register differs from a safe Boolean register only when the value x written (during an overlapping read) is the same as the old value already in the register: a regular register can only return x, while a safe register can return either true/false.

Construction: Associate an extra, local variable (old) with the register; write x only if x≠old (and update old)

Note: An M-valued register, M>2, can be turned into a Boolean register by reading and writing modulo 2

<u>Lemma</u>: The construction is a Boolean regular MRSW

©Jesper Larsson Träff

<u>Lemma</u>: The construction is a Boolean regular MRSW

same value, do nothing, x==f,
both reads return f (safe)

©Jesper Larsson Träff

<u>Lemma</u>: The construction is a Boolean regular MRSW

Read y fulfills regularity condition

©Jesper Larsson Träff

Regular Boolean MRSW to regular M-valued MRSW

Represent $0 \leq x < M$ in unary, use M regular Boolean regular registers to represent the M-valued regular register

Construction:

```
Array v[M] of regular Boolean MRSW registers
Init: v[0] = true;
read(thread i): for (j=0; j<M; j++) if (v[j]==true) return j;
write(x): v[x] = true; for (i=x-1; i≥0; i--) v[i] = false;
```

<u>Lemma</u>: The construction is a regular M-valued MRSW

Proof: When a write(x) completes, v[i]==**false** for all i<x; the smallest j with v[j]==**true** is the most recently written value.

If write(x) completes before read, read will eventually find v[x]==**true** and return x.

If reader reads v[j]==**false**, then most recent write(x) had x>j

©Jesper Larsson Träff

<u>Lemma</u>: The construction is a regular M-valued MRSW

Proof: Assume the register has been initialized, some v[i]=**true**.

When a write(x) completes, v[i]==**false** for all i<x; the smallest j with v[j]==**true** is the most recently written value.

If read(y) completes before write(x) (v[x]=**true**), then y<x, and y is the most recently (previous) written value

©Jesper Larsson Träff

Proof (con't):

If read overlaps write, consider the cases where reader and writer access the same Boolean SRSW register v[j], writer coming from some index x≥j, reader from 0≤j

Example: y may be any of 7, 2, 5, 9, but nothing else

©Jesper Larsson Träff

Proof (con't): 4 cases to consider

a) Reader reads v[j]==**false**, writer writes v[j]=**false**. Then the write operation is to an x>j, so ok for the reader to continue
b) Reader reads v[j]==**true**, writer writes v[j]=**false**. Then there must have been a previous (overlapping or most recent non-overlapping) write(j), since v[j] is a regular register. Ok for reader to return j

©Jesper Larsson Träff

Proof (con't): 4 cases to consider

c) Reader reads v[j]==**true**, writer writes v[j]=**true**; correct for the reader to return j
d) Reader reads v[j]==**false**, but writer writes[j]=**true**. Then, since register v[j] is regular, there must have been a most recent previous write(k) with k>j, so the reader may continue

©Jesper Larsson Träff

Regular SRSW to atomic SRSW

Not:
y==9, x==1

Idea:
Use time stamps to prevent that a later read gets an earlier value than a preceding read. Assumption: Registers can store time stamped values.

Construction: Maintain an old, time-stamped value; reader reads register and compares to old value, returns most recent value of either and updates; writer increments the time stamp

```
private:
unbounded timestamp stamp;   // incremented by writer
old: pair of <timestamp,value>; // updated by reader

Regular SRSW v of <timestamp,value> pairs; // register

Init: stamp = 0;

read():
w = v; // read regular SRSW register
if (old.timestamp<w.timestamp) old = w; // write takes effect
return old.value;

write(x):
stamp += 1; // increment time-stamp
v = <stamp,x>; // write regular SRSW register
```


©Jesper Larsson Träff

<u>Lemma</u>: The construction is an atomic M-valued SRSW

Time stamps are strictly increasing. Reader cannot return an earlier value than a preceding read, since that would have a lower time stamp

xi==1 or 9 for regular registers. In the construction, the moment xi==1, also xj==1 for j≥i

©Jesper Larsson Träff

Note:
This and the following constructions use unbounded time stamps, that cannot be represented with bounded, M-valued registers.
The proof-chain collapses…

The problem has been addressed with other constructions (not in book), e.g.:

Haldar, Vidaysankar: Constructing 1-writer multireader multivalued atomic variables from regular variables. JACM, 42(1): 186-203, 1995.
Israeli, Li: Bounded time-stamps. Distributed Computing, 6(4): 205-209, 1993.
Singh, Anderson, Gouda: The elusive atomic register. JACM, 41(2): 311-339, 1994.
Li, Tromp, Vitanyi: How to share concurrent wait-free variables. JACM, 43(4): 723-746, 1996.

Atomic SRSW to atomic MRSW

The single-array construction (safe SRSW to safe MRSW) does not work:

read(thread 0) returns c[0] and any later read(thread i) for i>0 could return an earlier value, if the write(x) was delayed when updating c[i] after having written c[0]

Working idea:
nxn matrix of atomic SRSW, reading thread helps later reading threads to get the correct value

©Jesper Larsson Träff

Each entry c[i][j] is a <stamp,value> pair of an atomic SRSW

<t,v>	<t,v>	<t,v>	<t,v>
<t,v>	<t,v>	<t,v>	<t,v>
<t,v>	<t,v>	<t,v>	<t,v>
<t,v>	<t,v>	<t,v>	<t,v>

write(x): t = t+1; write <t,x> in diagonal c[i][i]

©Jesper Larsson Träff

Each entry c[i][j] is a <stamp,value> pair of an atomic SRSW

<t+1,x>	<t,v>	<t,v>	<t,v>
<t,v>	<t+1,x>	<t,v>	<t,v>
<t,v>	<t,v>	<t,v>	<t,v>
<t,v>	<t,v>	<t,v>	<t,v>

Example: writer is delayed after i=1,...

write(x): t = t+1; write <t,x> in diagonal c[i][i]

©Jesper Larsson Träff

Each entry c[i][j] is a <stamp,value> pair of an atomic SRSW

<t+1,x>	<t,v>	<t,v>	<t,v>
<t,v>	<t+1,x>	<t,v>	<t,v>
<t,v>	<t,v>	<t,v>	<t,v>
<t,v>	<t,v>	<t,v>	<t,v>

max = <t+1,x>

write(x): t = t+1; write <t,x> in diagonal c[i][i]

read(thread i): find value w with max timestamp in column i

©Jesper Larsson Träff

Each entry c[i][j] is a <stamp,value> pair of an atomic SRSW

<t+1,x>	<t,v>	<t,v>	<t,v>
<t+1,x>	<t+1,x>	<t+1,x>	<t+1,x>
<t,v>	<t,v>	<t,v>	<t,v>
<t,v>	<t,v>	<t,v>	<t,v>

max = <t+1,x>

write(x): t = t+1; write <t,x> in diagonal c[i][i]

read(thread i): find value w with max timestamp t' in column i
write <t',w> in row i (except for c[i][i] since SW)
return w

<t+1,x>	<t,v>	<t,v>	<t,v>
<t+1,x>	<t+1,x>	<t+1,x>	<t,v>
<t,v>	<t,v>	<t,v>	<t,v>
<t,v>	<t,v>	<t,v>	<t,v>

Thread 1 delayed

Thread 0 (writer) delayed

©Jesper Larsson Träff

Valid event ordering (linearization)

<t+1,x>	<t,v>	<t,v>	<t,v>
<t+1,x>	<t+1,x>	<t+1,x>	<t,v>
<t,v>	<t,v>	<t,v>	<t,v>
<t,v>	<t,v>	<t,v>	<t,v>

Thread 1 delayed

Thread 0 (writer) delayed

©Jesper Larsson Träff

<t+1,x>	<t,v>	<t,v>	<t,v>
<t+1,x>	<t+1,x>	<t+1,x>	<t+1,x>
<t,v>	<t,v>	<t,v>	<t,v>
<t,v>	<t,v>	<t,v>	<t,v>

©Jesper Larsson Träff

<u>Lemma</u>: The construction is an atomic M-valued MRSW

Proof: Time-stamp t of writer is strictly increasing with each write. The maximum time stamp in any row or column is strictly increasing in time.

A reader clearly returns a value written by an overlapping or most recently completed write (no value from the future). If a write operation with timestamp t precedes a read operation, then the read will return this or some later value (no value from the distant past). If a read by thread A precedes a read by thread B, then A has written a time-stamped value in B's column, so the value returned cannot be a value written before the value read by A (atomicity).

Atomic MRSW to atomic MRMW

Idea: Array of atomic MRSWs with time stamps

read(thread i):
Scan through array, return the value with the maximum time stamp in the array

write(thread i, value x):
Scan through array, find latest time stamp t, write <t+1,x> to register at index i

<u>Lemma</u>: The construction is an atomic M-valued MRMW

Time stamping as in Bakery algorithm, for each writing thread, time stamps are strictly increasing.

Write order: Define that a write by A with time stamp tA precedes a write by B, A => B, with time stamp tB if tA<tB, or tA==tB and A<B (lexicographic order).

©Jesper Larsson Träff

<u>Lemma</u>: The construction is an atomic M-valued MRMW

Time stamping as in Bakery algorithm, for each writing thread, time stamps are strictly increasing.

Write order: Define that a write by A with time stamp tA precedes a write by B, A => B, with time stamp tB if tA<tB, or tA==tB and A<B (lexicographic order).

<u>Lemma</u>: The construction is an atomic M-valued MRMW

Time stamping as in Bakery algorithm, for each writing thread, time stamps are strictly increasing.

Write order: Define that a write by A with time stamp tA precedes a write by B, A => B, with time stamp tB if tA<tB, or tA==tB and A<B (lexicographic order).

<u>Lemma</u>: The construction is an atomic M-valued MRMW

Write order: Define that a write by A with time stamp tA precedes a write by B with time stamp tB if tA<tB, or tA==tB and A<B (lexicographic order).

If write A => write B (temporal order) then tA < tB. If write A and write B overlap, one will be ordered before the other either by time-stamp or by thread id.

No value from the future: A writing thread B which is after a reading thread A, A -> B, writes a time-stamp that is at least one larger than the time-stamp read by A. Thus, the value read by A cannot be the value from the future written by B.

<u>Lemma</u>: The construction is an atomic M-valued MRMW

No value from the distant past: Assume write(thread A) => write(thread B) -> read(thread C). If A=B, then B has overwritten the time-stamped value written by A, so C will return B's value. If A≠B, since A's time-stamp is smaller than B's, C will return the value written by B (or some other later thread after A). Thus, no value from the distant past can be returned.

©Jesper Larsson Träff

<u>Lemma</u>: The construction is an atomic M-valued MRMW

Atomicity, values read in write order: Assume read(thread A) -> read(thread B), and write(thread C) => write(thread D) in write order (the two writes may be concurrent). It must shown that if A returns the value written by D, then B cannot return the value written by C.

If tC<tD, then A read tD from index D, and since B also reads index D, it cannot return the value written by C (at index C). If tC=tD (concurrent writes) then C<D, and B must also return the value from index D (or a later one)

©Jesper Larsson Träff

Theorem:
Atomic MRMW are no more powerful than safe SRSW

Proof (now completed):
Constructions implemented an atomic MRMW out of a (large) number of safe SRSW

BUT:
Constructions are not practical. High space requirements. The atomic constructions rely on unbounded time stamps

On the register constructions:

1. No RW operation by some thread blocks RW operations of other threads
2. RW operations by any thread can progress independently of actions by other threads
3. RW operations always complete in a finite (bounded) number of steps, $O(n)$, $O(n^2)$, $O(M)$, … independently of actions by other threads

Algorithms, object methods, … with property 3 are called (bounded) wait-free

Lock-free: Infinitely often, some method/execution will finish in a finite number of steps (or: when several threads invoke some method, some thread will finish)

Progress conditions

Lock-freeness and wait-freeness are non-blocking, independent progress conditions

Blocking (dependent) progress conditions (progress of thread i can be delayed/blocked arbitrarily dependent on actions or non-actions of other threads):

- Starvation freedom
- Deadlock freedom

Definitions (non-blocking, independent progress):

- A concurrent algorithm (object, method call) is wait-free, if in any concurrent execution, each thread can finish in a finite (bounded) number of steps, independent of the actions of other threads
- A concurrent algorithm (object, method call) is lock-free, if in any execution some thread can finish in a finite (bounded) number of steps (or, in long execution, equivalently, some thread can complete a method call infinitely often), independent of the actions of other threads

A wait-free algorithm whose bounds do not depend on the number of threads is called population oblivious

Definitions (blocking, dependent progress):

- A concurrent algorithm (object, method call) is starvation free, if in any execution, any thread will eventually be able to complete, provided that other threads take steps
- A concurrent algorithm (object method call) is deadlock free, if in any execution, some thread will eventually be able to complete, provided that other threads take steps

Degrees of starvation freedom: Fairness

Main examples: locks, monitors, …: progress possible if threads relinquish the shared resource (unlock), in particular do not stall (crash) indefinitely

Blocking	Non-blocking
	Obstruction-free
Deadlock-free	Lock-free
Starvation-free	Wait-free

Note:

- Starvation-free implies deadlock-free
- Wait-free implies lock-free

Question:

Are wait- or lock-free implementations of concurrent data structures/objects/methods/algorithms always possible? Needed?

Non-blocking but dependent progress condition:

Obstruction freedom: A thread that executes in isolation (no other threads taking steps), can finish in a finite number of steps

Remark:
An obstruction free algorithm can sometimes be turned into a useful, "wait-free" algorithm by introducing back-offs: A thread that detects some kind of conflict with other threads, will step back and yield (processor) to another thread

Atomic snapshots

Assume a whole set of atomic registers describing the state of some algorithm…
Might be beneficial for any thread to be able to read this state atomically, i.e. take a consistent snapshot(*)

Simplified version (MRSW):

- Each thread has a register, update(value) operation writes value to threads register (SW)
- scan() operation takes snapshot of all thread's registers (MR) and copies into local array

Is this possible in a wait-free manner?

(*) state that actually exist(ed) at some point in the global time

©Jesper Larsson Träff

Assume a whole set of atomic registers describing the state of some algorithm…
Might be beneficial for any thread to be able to read this state atomically, i.e., take a consistent snapshot

Correct: Possible to find a total order on the operations, such that sequential specification is satisfied, and such that the snapshot was valid in the interval of the operation, that is, it reflects the state after the most recent preceding or overlapping update

©Jesper Larsson Träff

Assume a whole set of atomic registers describing the state of some algorithm…
Might be beneficial for any thread to be able to read this state atomically, i.e., take a consistent snapshot

Correct: Possible to find a total order on the operations, such that sequential specification is satisfied, and such that the snapshot was valid in the interval of the operation, that is, it reflects the state after the most recent preceding or overlapping update

Linearization point: operation "takes effect"

Such an ordering is called a linearization

©Jesper Larsson Träff

Assume a whole set of atomic registers describing the state of some algorithm…
Might be beneficial for any thread to be able to read this state atomically, i.e., take a consistent snapshot

Correct: Possible to find a total order on the operations, such that sequential specification is satisfied, and such that the snapshot was valid in the interval of the operation, that is, it reflects the state after the most recent preceding or overlapping update

Wrong if second scan does not see second update

©Jesper Larsson Träff

Blocking (lock-based, sequential) implementation:

```
class LockSnapshot<T> implements Snapshot<T> {
public synchronized void update(T v) {
    regarray[ThreadID.get()] = v; // set my state value
}
public synchronized void T[] scan() {
  T[] snapshot = (T[]) new atomic reg[regarray.length];
  for (int i=0; i<regarray.length; i++) {
    snapshot[i] = regarray[i];
  }
  return snapshot;
}
}
```


©Jesper Larsson Träff

An obstruction-free snapshot

Each register is MRSW atomic. Each thread maintains a local time-stamp that is incremented at each update operation, <timestamp,value> pairs are written in register

```
update(value):
increment local timestamp and write <timestamp,value>
```

<u>Lemma</u> (observation): The update operation is wait-free

Note:
Timestamps here just ensure that written values are unique

©Jesper Larsson Träff

A collect operation simply copies the thread registers into a local array (in some order)

Observation: If two successive collects return the same value (array contents), then this interval can have had no update, so the collected value (array) is a correct snapshot. Call this a clean double collect.

scan(): Perform collect until the last double collect is clean

Lemma: The scan operation is obstruction-free, but not wait-free. A scan can always be disturbed by an overlapping update. But as soon as there are no concurrent updates (scanning thread executes in isolation), it can complete correctly.

Not possible to linearize scan with update if only a single collect is done (could try to linearize scan either before or after update, however, some later update may or may not be seen; or another concurrent scan could be incompatible with this order)

Clean double collect implies no concurrent update

Scan might "see" either of the updates

©Jesper Larsson Träff

A wait-free snapshot

Previous solution is dependent on other threads (to not make updates). For the wait-free solution: helper scheme. Updating threads shall help concurrent or next scan

An update takes a snapshot by calling scan, before it updates its register

Idea:
If a scanning thread fails to make a clean double collect, use the snapshot taken by the thread that performed overlapping update

…does not quite work

regarray: the registers constituting the state, one register per thread

```
public void update(T value) {
  int i = TheadID.get();
  T[] snap = scan(); // help by taking snapshot
  int t = regarray[i].stamp;
  Stamped<T> newval = new Stamped(t+1,value,snap);
  regarray[i] = newval;
}
```

Registers now store a triplet: <timestamp,value,reference to snapshot>

Java note: memory management (free'ing allocated memory) is left to garbage collection system (must then be wait-free etc.!!)

A scan that repeatedly fails uses the snapshot taken by one of the concurrent updates

A thread is said to move if it completes an update() operation.

Idea: When a scan by A fails, because B has moved, take B's snapshot? Does not work since B could have completed its scan before A started scanning, so the snapshot was outside A's interval

©Jesper Larsson Träff

Repair: If A sees B move twice while performing collects, then B's most recent snapshot is within A's interval

Scan: Perform double collect, if not clean, let some concurrent thread move twice and use this thread's snapshot

©Jesper Larsson Träff

```
public T[] scan() {
  Stamped<T> oldsnap, newsnap;
  boolean[] moved = new boolean[threads]; // all false

  oldsnap = collect();
  collect: while (true) {
    newsnap = collect(); // double collect
    for (j=0; j<threads; j++) { // clean?
      if (oldsnap[j].stamp!=newsnap[j].stamp) {
        if (moved[j]) return newsnap[j].snap;
        else {
          moved[j] = true;
          oldsnap = newsnap;
          continue collect;
        }
      }
    } // clean: allocate and copy into local array, done
} }
```

Observations:

- If a thread makes a clean double collect, the snapshot is correct (a state that existed in the registers in the interval of the scan)
- If A sees a change in B's register (time-stamp) during two different collects, then the value in B's register read during the last collect was written by an update() that began some time after the first collect started.
- It follows that the snapshot returned by a scan() corresponds to a state in the interval of the scan (clear if A made a clean double collect; otherwise, B's scan(), which is inside A's interval was from a clean double collect, or some other embedded scan by C made a clean double collect… But there can be at most n-1 embedded updates, n being the number of threads)

<u>Lemma</u>: Every scan() or update() operation completes after at most $O(n^2)$ and at least $\Omega(n)$ register reads or writes, where n is the number of threads

Proof: For any scan() after at most n+1 double collects, either one is clean, or some thread is observed to move twice (in which case its snapshot is returned); each collect takes n reads

<u>Corollary</u>: The snapshot implementation is wait-free

Extension and simplification

Problems:

- MRSW registers
- Triple <stamp,value,reference> in registers

The idea can easily be extended to MRMW registers: Keep track of which tread updated a register.

Let `regarray[]` be an array of m MRMW registers

©Jesper Larsson Träff

Each thread has a reference to a snapshot (array of m values)

```
public void update(int i, T value) {
  int w = TheadID.get();
  int t = regarray[i].stamp;
  Stamped<T> newval = new Stamped(t+1,w,value);
  regarray[i] = newval;
  snap[w] = scan(); // help after update
}
```

The register array now stores triples <stamp,thread id,value> which might fit better in register with finite number of bits (next stamp could also be maintained thread locally).

The helper snapshot is taken after the register update (a reference to the array of values computed by scan())

```
public T[] scan() {
  Stamped<T> oldsnap, newsnap;
  boolean[] moved = new boolean[threads]; // all false

  oldsnap = collect();
  collect: while (true) {
    newsnap = collect(); // double collect
    for (j=0; j<m; j++) { // all m registers clean?
      if (oldsnap[j]!=newsnap[j]) {
        int w = newsnap[j].w; // which thread moved?
        if (moved[w]) return snap[w];
        else {
          moved[w] = true;
          oldsnap = newsnap;
          continue collect;
        }
      }
    } // clean: allocate and copy into local array, done
} }
```

The correctness proof is a bit more involved, but establishes strong "freshness" properties of the snapshot. Also extends to "partial snapshots" (a subset of the registers is read)

Damien Imbs, Michel Raynal: Help when needed, but no more: Efficient read/write partial snapshot. J. Parallel Distrib. Comput. 72(1): 1-12 (2012)

Issue: Memory management, when can a snapshot be thrown away?

Other snapshot results and algorithms

Yehuda Afek, Gideon Stupp, Dan Touitou: Long-lived and adaptive atomic snapshot and immediate snapshot (extended abstract). PODC 2000: 71-80
James H. Anderson: Composite Registers. Distributed Computing 6(3): 141-154 (1993)
James H. Anderson: Multi-Writer Composite Registers. Distributed Computing 7(4): 175-195 (1994)
Hagit Attiya, Ophir Rachman: Atomic Snapshots in O(n log n) Operations. SIAM J. Comput. 27(2): 319-340 (1998)
Elizabeth Borowsky, Eli Gafni: Immediate Atomic Snapshots and Fast Renaming (Extended Abstract). PODC 1993: 41-51
Prasad Jayanti: An optimal multi-writer snapshot algorithm. STOC 2005: 723-732

Advanced Multiprocessor Programming

Jesper Larsson Träff
traff@par.tuwien.ac.at
Research Group Parallel Computing
Faculty of Informatics, Institute of Computer Engineering
Vienna University of Technology (TU Wien)

©Jesper Larsson Träff

The relative power of synchronization primitives (Chap. 5)

- What registers cannot do: The basic result!
- Concurrent correctness
- More powerful synchronization primitives

Maurice Herlihy: Wait-free synchronization. ACM Transactions on Programming Languages and Systems (TOPLAS), 13(1):124-149, 1991
Maurice Herlihy, Jeannette Wing: Linearizability: A correctness condition for concurrent objects. ACM TOPLAS 12(3): 463-492, 1990

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

More formally:
A consensus object has a method `decide(v)` that can be called at most once per thread with some thread local value v.
`decide(v)` returns a value that is

- Consistent: All threads return the same value
- Valid: One of the values provided in the calls is returned

The `decide()` method is wait-free: Any call to the method will finish in a finite number of operations, independently of other treads actions

Consensus protocol:
A class (algorithm) that implements consensus in a wait-free manner

A class solves the n-thread consensus problem, if there exists a consensus protocol for n threads using any number of objects of the class and any number of atomic registers. The class can be implemented in software (algorithm) or hardware (special registers and instructions)

Definition:
The consensus number of a class (algorithm/instruction) is the largest number of threads n for which the class solves the consensus problem. If no largest (finite) n exists, the consensus number is ∞

Consensus interface:

```
public interface Consense<T> {
  T decide(T v); // decide on v provided by each thread
}
```

Generic consensus protocol:

```
public abstract class Consensus<T> implements Consense {
  protected T[] proposed = (T[])new Object[N];
  void propose(T v) {
    proposed[ThreadID.get()] = v;
  }
  abstract public T decide(T v);
}
```

Not a solution (why?) to the consensus problem, but useful idea: first thread that calls `decide()` determines value to be returned by other threads

```
class LOCKconsensus extends Consensus {
  private int who = -1; // special non-thread ID
  private Lock lock;
  public Object decide(Object value) {
    lock.lock();
    try {
      propose(value); // store value
      if (who==-1) who = ThreadID.get(); // first thread
      return proposed[who];
    } finally {
      lock.unlock();
    }
  }
}
```

State:
state (x=v) of all threads, and all shared objects/variables

Move:
A method call/state update by some thread

Initial state: Before any move

Final state:
Protocol state after all threads have finished. Decision value d is the value decided in that state.

<u>Lemma</u>: Protocol is wait-free, so tree is finite

©Jesper Larsson Träff

Special case:
Binary consensus, each thread provides value true/false, 0/1

In the final state of a binary consensus protocol, the decision value d is therefore also either 0 or 1

Note:
The decision value d is not necessarily some explicit, actual register where the decision can be read off. A state has decided d=0 or d=1 if in that state the value that will eventually be returned to each thread is fixed.

Binary consensus state is bivalent, if decision value is not fixed in that state: different move sequences can lead to either d=0 or d=1

Univalent: d is now fixed
1-valent: d=1
0-valent: d=0

©Jesper Larsson Träff

<u>Lemma</u>:
Every n-thread consensus protocol has an initial bivalent state

Proof:
Consider an initial state where thread A contributes value 0, all other threads value 1.

If A finishes the protocol (it can: wait-free) before the remaining threads make a move, d=0, because A cannot know anything about the other threads value (0/1) before they make at least one move. Similarly, if some other thread B finishes before A or any other thread makes a move, d=1.

Thus, the initial state is bivalent

The "last" bivalent state is critical: A bivalent state is said to be critical, if any move by any thread will make the next state univalent

Lemma:
Any consensus protocol has a critical state

Proof:
Start in the initial bivalent state. As long as there is a thread that can move without making the state univalent, make the move. Since the protocol is wait-free, the state transition tree is finite. Therefore, eventually a state is reached where no such move is possible. This state is critical

Theorem:
Atomic registers have consensus number 1

Proof:
By contradiction. Suppose there exist a binary consensus 2-thread protocol using only atomic registers.

Run the protocol into a critical state. Assume thread A's next move leads to a 0-valent state, thread B's next move to a 1-valent state.

Three possibilities:
1. A's next move will read a register, B's will read or write
2. A's and B's next move will write to different registers
3. A's and B's next move will write to same register

1. Thread A's move will read

Contradiction: States s' and s'' are indistinguishable to B, since A only read (local state change for thread A cannot have any effect for B), therefore B would have had to make the same decision in s' and s''

2. Thread A and B will write to different registers

Contradiction: Since neither thread can determine which went first (the two writes "commute"), both sequences should lead into the same (univalent) state; but by critical state assumption, do not.

©Jesper Larsson Träff

3. Thread A and B will write to same register

Contradiction: s' and s'' are indistinguishable to B, since in s' B overwrote A's write to R. Therefore, B must decide the same value from s' and s''

<u>Main Corollary</u>:
Atomic registers alone are insufficient to construct wait-free implementations of any concurrent object with consensus number greater than one

Therefore:
For constructing wait- (and lock-) free data structures/objects, (hardware support for) synchronization primitives that are more powerful than atomic reads and writes are needed

Stronger (m,n)-assignment registers

Let $n \geq m > 1$. An (m,n)-assignment register has n fields, and can atomically write m (index,value) pairs to m indexed fields.

The value for a given index can be read atomically.

Note: (m,n)-assignment is the "dual" of atomic snapshots

Lemma: The atomic snapshot object has consensus number 1 (since it can be implemented with atomic registers)

©Jesper Larsson Träff

Example: lock-based implementation of a (2,3)-assignment register

```
public class Assign23 {
  int[] r23 = new int[3]; // use array for the 3 values
  public Assign23(int init) {
    r23[0] = init; r23[1] = init; r23[2] = init;
  }
  public synchronized void assign(T v0, T v1,
                                  int i0, int i1) {
  // atomically write 2 out of 3
  r[i0] = v0; r[i1] = v1;
  }
  public synchronized int read(int i) {
    return r[i];
  }
}
```

Java `synchronized` makes method call atomic (critical section)

<u>Theorem</u>:
Atomic (n,n(n+1)/2)-register assignment for n>1 has consensus number at least n

Proof: By construction, by giving a consensus algorithm for n threads

(n+1)n/2 registers, i=0,1,…,n-1, and i≤j

Initially, all registers are set to special † value

©Jesper Larsson Träff

```
decide(v):
```

1. Thread i assigns "row" rij = v, i≤j, and "column" rji = v, j<i

Initially, all registers are set to special † value

`decide(v)`:

1. Thread i assigns "row" rij = v, i≤j, and "column" rji = v, j<i

Thread 2 calls `decide(v2)`

Initially, all registers are set to special † value

©Jesper Larsson Träff

```
decide(v):
```

1. Thread i assigns "row" rij = v, i≤j, and "column" rji = v, j<i

Thread 2 calls `decide(v2)`
Thread 3 calls `decide(v3)`

Initially, all registers are set to special † value

```
decide(v):
```

1. Thread i assigns "row" rij = v, i≤j, and "column" rji = v, j<i

Thread 2 calls `decide(v2)`
Thread 3 calls `decide(v3)`
Thread 0 calls `decide(v0)`

Initially, all registers are set to special † value

©Jesper Larsson Träff

`decide(v)`:

1. Thread i assigns "row" rij = v, i≤j, and "column" rji = v, j<i

Thread 2 calls `decide(v2)`
Thread 3 calls `decide(v3)`
Thread 0 calls `decide(v0)`

Note:
Only thread i writes to rii. If rjj≠† then thread j has called `decide()`

Initially, all registers are set to special † value

`decide(v)`:

1. Thread i assigns "row" rij = v, i≤j, and "column" rji = v, j<i

Thread 2 calls `decide(v2)`
Thread 3 calls `decide(v3)`
Thread 0 calls `decide(v0)`

Idea:
Find the first thread that executed `decide()`, return this value as consensus value

Initially, all registers are set to special † value

```
decide(v):
```

1. Thread i assigns "row" rij = v, i≤j, and "column" rji = v, j<i
2. Scan diagonal, read rii, for i=0, 1, …:

©Jesper Larsson Träff

```
decide(v):
```

1. Thread i assigns "row" rij = v, i≤j, and "column" rji = v, j<i
2. Scan diagonal, read rii, for i=0, 1, …:
3. Let j be first index i where rii≠†; j=i is candidate for result

```
decide(v):
```

1. Thread i assigns "row" rij = v, i≤j, and "column" rji = v, j<i
2. Scan diagonal, read rii, for i=0, 1, …:
3. Let j be first index i where rii≠†; j=i is candidate for result
4. Continue scan, if rii=†, j can still be result

©Jesper Larsson Träff

```
decide(v):
```

1. Thread i assigns "row" rij = v, i≤j, and "column" rji = v, j<i
2. Scan diagonal, read rii, for i=0, 1, …:
3. Let j be first index i where rii≠†; j=i is candidate for result
4. Continue scan, if rii=†, j can still be result
5. If rii≠rjj, then if rji=rjj, set j = i, new candidate for result

```
decide(v):
```

1. Thread i assigns "row" rij = v, i≤j, and "column" rji = v, j<i
2. Scan diagonal, read rii, for i=0, 1, …:
3. Let j be first index i where rii≠†; j=i is candidate for result
4. Continue scan, if rii=†, j can still be result
5. If rii≠rjj, then if rji=rjj, set j = i, new candidate for result

Thread 2 was first, its value r22 is returned as decided

6. Return rjj: consensus value

©Jesper Larsson Träff

Simplified example: 2-consensus with a (2,3)-register

```
class MULTI2consensus extends Consensus {
  private final int NULL = -1;
  Assign23 value = new Assign23(NULL);
  public T decide(T v) {
    propose(v);
    int i = ThreadID.get();
    value.assign(i,i,i,i+1); // (2,3)-assignment
    int other = value.read((i+2)%3);
    if (other==NULL || other==value.read(1)) {
      return proposed[i]; // thread won
    } else return proposed[1-i]; // thread lost
  }
}
```

Thread 0, 3 cases:

0	0	†

0 alone, d=0

0	1	1

0 first, d=0

0	0	1

0 last, d=1

©Jesper Larsson Träff

Corollary:
(2,3)-assignment registers have consensus number at least 2

Corollary:
It is not possible to construct an (m,n)-assignment register out of atomic registers

Reminder:
Atomic snapshot could be constructed from atomic registers; the dual, atomic multi-assignment cannot. Atomic multi-assignment is more demanding than snapshot

Remark:
Have (m,n)-assignment registers ever been implemented in hardware (Exercise: Check this)?

Digression: Concurrent objects, correctness, consistency (Chap. 3)

System state: all objects states + all thread states, including program counter

Concurrent method calls change state of object (and system state)

Local thread state

©Jesper Larsson Träff

Mandatory properties of concurrent objects

Safety (correctness, whatever that means): "Nothing bad (violation of some requirements) can ever happen"

Counterpart of sequential correctness (e.g. of data structure like stack, queue, PQ, …)

Liveness: Progress, conditions under which a thread or the execution as a whole can make progress; "something good will (eventually) happen"

What does it mean that a concurrent execution of a concurrent object is correct?

- Method calls take time (intervals, not events) and can overlap (no total order on intervals)
- Relate to sequential specification of object

Example (useful concurrent object): FIFO queue that supports (sequentially specified) enqueue/dequeue operations in a wait-free manner

- `q.enq(a)`: enqueues a in q (append to tail)
- `q.deq(b)`: if q nonempty, dequeues b from q (remove head), otherwise throw exception (or similar action, e.g. return †)
- Item dequeued is least recently item enqueued (whatever that means)

State of q described by the sequence of elements in q

Example:
Lock-based FIFO queue implementation, each queue operation is done in a critical section, protected by a lock

Such an implementation can be deadlock free, starvation free, may satisfy other dependent progress criteria, depending on the lock properties

Such an implementation is blocking: A delay by one thread can (will) cause other treads to wait.

It is not wait-free: It is not guaranteed that every method call will finish in a finite (bounded) number of steps

It is not lock-free: It is not guaranteed that some method call will always finish in a finite number of steps

```
class LockQueue<T> {
  int head, tail;
  T[] queue;
  Lock lock;
  public void enq(T x) throws Full {
    lock.lock();
    try {
      if (tail-head==queue.length) throw new Full();
      // insert at tail ……
    } finally {
      lock.unlock();
    }
  }
  public T deq() throws Empty {
    lock.lock();
    // … if empty throw exception, else return head
    lock.unlock();
  }
}
```

Classical sequential correctness/specification:
If object in state s' satisfying precondition C', then method call will result in state s'' satisfying postcondition C''

Concurrent correctness condition:
Relates any concurrent execution to the sequential specification (preferably in such a way not to enforce blocking/dependent progress). Such a condition is called consistency (Examples: sequential consistency, quiescent consistency, linearizable consistency, …)

Informally: A concurrent execution is correct if it is consistent with some sequential execution

Desirable: If objects A and B are consistent (according to some condition), then any system using A and B together is also consistent. Such a consistency condition is called compositional.

Example: Sequential specification of queue class with methods enq() and deq(); represent queue state S = <x1,x2,x3,…> by a finite sequence of queued elements

Precondition: <Q>

```
q.enq(a)
```

Postcondition: <Q.a>

Precondition: <e.Q>

```
v = q.deq()
```

Postcondition: v==e, <Q>

Precondition: <>

```
v = q.deq()
```

Postcondition: EMPTY exception

Note: This specification is total, queue either returns value or throws exception (alternative: Returns special † value)

Sequential consistency:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order (but not necessarily in real time)
- For each thread, method calls should appear to take effect in program order

Example:

Program A:
…
q.enq(a);
…
x = q.deq():
…

Program B:
…
q.enq(b);
…

Program order: The order in which a single thread executes method calls.

Note: Method calls by different threads are unrelated by program order

©Jesper Larsson Träff

ACM Turing Award 2014

Lamport: …
The result of a concurrent execution is as if all the operations were executed in some sequential order (interleaving), and the operations of each thread appear in the order specified by the program

Example:

Program A:
…
q.enq(a);
…
x = q.deq():
…

Program B:
…
q.enq(b);
…

Leslie Lamport: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Trans. Computers 28(9): 690-691 (1979)

Sequential consistency:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- For each thread, method calls should appear to take effect in program order

Should not happen; either dequeue a or b ("one-at-a-time sequential order"), not consistent with any sequential execution

©Jesper Larsson Träff

Sequential consistency:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- For each thread, method calls should appear to take effect in program order

What about this?

Sequentially consistent!

Sequential consistency is not a real-time condition

©Jesper Larsson Träff

Sequential consistency:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- For each thread, method calls should appear to take effect in program order

Still sequentially consistent

Anticipation:
This history would not be linearizable

Sequential consistency is not a real-time condition

©Jesper Larsson Träff

Sequential consistency:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- For each thread, method calls should appear to take effect in program order

Because: A (real-time) sequentially correct execution can be constructed by moving the threads' method calls in time (enq(b) before enq(a))

©Jesper Larsson Träff

Method calls take time…

…modeled by the interval between invocation and response…

…invocations and responses are events (totally ordered)

©Jesper Larsson Träff

Some technicalities (part I): Histories, definitions

H: Finite sequence of (totally ordered) invocation and response events

H: <q.enq(a) A>,<q.enq(b) B>,<q:ok A>,<q:ok B>,<q.deq B>,<q:b A>

- Invocation of method m on object x with arguments a* by thread A: <x.m(a*) A>
- Response by object x to thread B with values v*: <x:v* B>

©Jesper Larsson Träff

Note:
The response to thread A to invocation of method m on object x by thread A is always to the most recent, preceding invocation of m (in other words, a thread that invokes a method waits for the response)

A response <x:v* A> matches an invocation <x.m(a*) A> in H: same object and same thread

A method call on method m of object x by thread A in H is a pair (interval) consisting of an invocation <x.m A> and the next matching response event <x:v A>

Operations, notation:

A subhistory H' of some history H is a subsequence of H

- H|A: the (sub)history of H consisting of invocations and responses by thread A (thread subhistory)
- H|x: the (sub)history of H consisting the invocations and responses on object x (object subhistory)

An invocation in H for which there is no matching response is pending; complete(H) is the (sub)history where all pending invocations have been removed.

An extension of a(n incomplete) history H is a history H' constructed by possibly appending matching responses to pending invocations

Definition:
Histories H and G are equivalent if H|A = G|A for all threads A
(Exercise: Prove that this is an equivalence relation)

Definition:
A history H = $e_1, e_2, e_3, \ldots, e_n$ is sequential if e_1 is an invocation, and each invocation e_i is immediately followed by a matching response e_{i+1}

Note: A sequential history is a history with no overlapping method calls

A history is well-formed if each thread subhistory is sequential (no overlapping method calls by a single thread). Only well-formed histories considered here

Note: Object subhistories H|x of a well-formed history need not be sequential (there can be concurrent calls to x)

Definition:
A (sequential) specification of an object is the set of those sequential object histories that are correct; each such object history is legal

A sequential history S is legal if all object subhistories H|x are legal

End of technicalities (part I)

Sequential consistency:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- For each thread, method calls should appear to take effect in program order

Means:
In any concurrent execution, there is a way to adjust (slide) the method calls in time (no swapping) such that
- they (for each thread) are consistent with program order
- meet the objects sequential (correctness) specification

More formally:
Any execution history H is equivalent to some correct sequential execution history S that satisfies program order for each thread

Sequential consistency:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- For each thread, method calls should appear to take effect in program order

Example:

Program A:	Program B:
…	…
q.enq(a);	q.enq(b);
…	…
x = q.deq();	y = q.deq();
…	…

Possible under sequential consistency to have x=b and y=a?

Sequential consistency:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- For each thread, method calls should appear to take effect in program order

… is sequentially consistent:

Two possible executions that fulfill the conditions

©Jesper Larsson Träff

Sequential consistency:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- For each thread, method calls should appear to take effect in program order

Sequentially consistent execution 1:

<q.enq(b) B> -> <q.enq(a) A> -> <q.deq(b) A> -> <a.deq(a) B>

Sequential consistency:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- For each thread, method calls should appear to take effect in program order

Sequentially consistent execution 2:

<q.enq(a) A> -> <q.enq(b) B> -> <q,deq(b) B> -> <q.deq(b) A>

Key observation: Sequential consistency is not compositional

Example: Two queues p and q in two threads

Program A:	Program B:
…	…
p.enq(a);	q.enq(b);
q.enq(a);	p.enq(b);
…	…
x = p.deq():	y = q.deq();
…	…

Possible that x=b and y=a? – as could happen with p or q in isolation?

Key observation: Sequential consistency is not compositional

Definition: A correctness property/consistency condition P is compositional if, whenever each object (and method) of the system satisfies P, then the system as a whole satisfies P.

If all object subhistories of some execution history satisfy P (e.g. sequential consistency) then the execution history as a whole should satisfy P

Compositionality is desirable:

Different data structures can be developed independently, different objects verified independently

Unfortunately, sequential consistency is not compositional (see why on next slides)

©Jesper Larsson Träff

<u>Key observation</u>: Sequential consistency is not compositional

Two queues p and q; both sequences on p and q can be put in sequentially consistent order

©Jesper Larsson Träff

<u>Key observation</u>: Sequential consistency is not compositional

Two queues p and q; both sequences on p and q can be put in sequentially consistent order

©Jesper Larsson Träff

<u>Key observation</u>: Sequential consistency is not compositional

Two queues p and q; both sequences on p and q can be put in sequentially consistent order

More formally: Both p and q object subhistories sequentially consistent

BUT…

©Jesper Larsson Träff

<u>Key observation</u>: Sequential consistency is not compositional

Queues are FIFO, so

<p.enq(b) B> -> <p.enq(a) A> (because of <p.deq(b) A>) and <q.enq(a) A> -> <q.enq(b) B>

Note: Here -> means "has completed before"

©Jesper Larsson Träff

<u>Key observation</u>: Sequential consistency is not compositional

Now <p.enq(b) B> -> <p.enq(a) A> and <q.enq(a) A> -> <q.enq(b) B>

Program order states
<p.enq(a) A> -> <q.enq(a) A> and <q.enq(b) B> -> <p.enq(b) B>

©Jesper Larsson Träff

<u>Key observation</u>: Sequential consistency is not compositional

<p.enq(b) B> -> <p.enq(a) A> and <q.enq(a) A> -> <q.enq(b) B>
<p.enq(a) A> -> <q.enq(a) A> and <q.enq(b) B> -> <p.enq(b) B>

implies <p.enq(b) B> -> <q.enq(b) B> contradicting program order for B (likewise for A)

<u>Key observation</u>: Sequential consistency is not compositional

Although both p and q object subhistories of the example execution history were sequentially consistent, the history as a whole violated program order.

This means: Sequential consistency is not compositional

©Jesper Larsson Träff

Quiescent consistency:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- Method calls separated by a period of quiescence (no activity, no overlapping method calls,…) should appear to take effect in real-time order

Quiescent consistency relaxes the strict adherence to program order (for overlapping method calls) AND introduces a real-time component

Example:
A and B enqueues x and y, the queue object becomes quiescent, C enqueues z: The queue will hold x and y in some order, followed by z

Difficulty: How long should the period of quiescence be?

Theorem (without proof here): Quiescent consistency is compositional.

Exercises

Note:
Quiescent and sequential consistency are incomparable: There are executions that are sequentially consistent by not quiescently consistent, and vice versa

Problem with sequential consistency:

- Intuitive outcomes of programs, attractive for reasoning about memory (too restrictive for real hardware): Updates by any thread become visible to other threads in program order, but delays are allowed
- Does not respect real-time order

is sequentially consistent, but not consistent with the way the method calls happened in time

- Sequentially consistent, AND
- Possible to find points in time where method calls take effect such that the execution is sequentially correct and respects the real-time order in which the method calls happened

©Jesper Larsson Träff

- Sequentially consistent, AND
- Possible to find points in time where method calls take effect such that the execution is sequentially correct and respects the real-time order in which the method calls happened

©Jesper Larsson Träff

Linearizability (linearizable consistency, atomicity):

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- Each method call should appear to take effect in real-time order, instantaneously at some point between invocation and response event

Terminology:
The points where method calls take effect are called linearization points

Proving linearizability

- Of some given history: Identify possible linearization points in the history, such that the linearization order is a correct sequential history. For any given history, there may be many different linearizations (see exercises and later)

- Of an object/algorithm: Show that any concurrent history that can be generated by the object is linearizable. Linearization points often correspond to the execution of some atomic operation (see later). Different executions may generate different linearizations. But: Linearization point may not always be the same instruction, and may depend on the actual history

Example: Wait-free, linearizable ("atomic") counter

```
class AtomicCounter {
  int counter[N]; // N number of threads. Initially 0
  public void inc() {
    counter[ThreadID.get()]++; // non-atomic inc
  }
  public int read() {
    int value = 0;
    int i;
    for (i=0; i<N; i++) value += counter[i];
    return value;
  }
}
```

Uses only registers

Idea: Counter represented as an array of "per thread counts"

©Jesper Larsson Träff

Claim: Counter history is linearizable (behaves atomically).
Question: When does read() linearize?

©Jesper Larsson Träff

Lemma: Both inc() and read() operations are wait-free

©Jesper Larsson Träff

Lemma: The inc() operation for thread i linearizes at the point where counter[i] is updated

©Jesper Larsson Träff

Lemma: Let k be the value of the counter before the read operation starts (consider all inc() -> read()); let v = k+x be the value returned by read(). Then read() linearizes between the x'th and the (x+1)'th overlapping inc() operation

Possible linearization point if x=2

Even if thread 1 has not seen thread 0's (and/or thread 2's) update(s)

Lemma: Let k be the value of the counter before the read operation starts (consider all inc() -> read()); let v = k+x be the value returned by read(). Then read() linearizes between the x'th and the (x+1)'th overlapping inc() operation

Recall: Increment (++) is commutative

Proving linearizability of given data structure/algorithm

Linearization point: Some (virtual) point during the method call where the delivered result would have been correct. A proof argues that this point can always be found; but it may depend on other, concurrent operations

©Jesper Larsson Träff

Linearizability:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- Each method call should appear to take effect in real-time order, instantaneously at some point between invocation and response event

Theorem:
Linearizability is compositional

Theorem:
Linearizability is a non-blocking property. A pending method call never has to wait for another pending method call to respond

Corollary:
Every linearizable execution is sequentially consistent (not vice versa)

Technicalities (part II): Formal definition and proofs

Method call in H: Pair of matching invocation and response events.
m = …<x.m A>, …, <x:v A>, ….

Method call m0 is before method call m1, written m0 -> m1, in H if H=…<x.m0 A> … <x:v A>… <x'.m1 A'>… (response event of m0 before invocation event of m1)

Since the interval of method call m0 may overlap the interval of the method call m1, -> is a partial order

Important: For a sequential history S, the relation -> is total. Since there are no overlapping method calls in S, either m0 -> m1 or m1 -> m0 for any method calls m0, m1 in S

Definition:
History H is linearizable if there is an extension H' and a legal sequential history S such that
1. complete(H') is equivalent to S
2. if m0 -> m1 in H for method calls m0, m1, then m0 -> m1 in S.

S is called a linearization of H

- A method call that precedes another in H ("real time") must also do so in S. S totally orders (linearizes) the method calls in H, and S is a legal sequential history.
- It is allowed to add responses to some pending invocations in H ("these have already taken effect") in order to obtain the linearization S

<u>Theorem</u>: History H is linearizable iff for each object x, H|x is linearizable

Thus: Linearization is compositional

Proof (only if):
If H is linearizable to S, then S|x is a linearization of H|x for each object x

<u>Theorem</u>: History H is linearizable iff for each object x, H|x is linearizable

Proof (if):
Induction on the number of method calls in H. For each object x in H, pick a linearization Sx of H|x. Each Sx may include some responses not in H. Let Rx be all such responses, and H' = HRx.
Induction base: If there is only one method call in H', H' is itself a linearization.
Induction step: Assume claim for fewer than k method calls. For each x consider last method call in Sx. One of these m must be maximal in H', that is there is no m' with m -> m'. Let G' be H' with m removed. Now, H' is equivalent to G'm (because m was last and maximal). By induction hypothesis G' is linearizable to some S', and both H and H' are linearizable to S'm.

Proof (illustration):

©Jesper Larsson Träff

Proof (illustration): By assumption

- H|x linearizable
- H|y linearizable

Remove m from H' to get G'; H' is equivalent to G'm (because of maximality); G' has one less method call, by induction hypothesis G' is linearizable

©Jesper Larsson Träff

G'm clearly equivalent to H (Gm|A = H|A for all threads A)

©Jesper Larsson Träff

But for the G' obtained by removing a last, but not maximal m, it does not hold that G'm is equivalent to H (thread history for lower thread has changed)

©Jesper Larsson Träff

Proof of maximality claim: "For each x consider last method call in Sx. One of these m must be maximal in H', that is there is no m' with m -> m'."

Induction on the number of objects x.

Induction base: If there is only one object x in H, then the last call m in the linearization Sx is maximal; if not, that is m -> m' for some m' in H, then also m -> m' in Sx and m could not have been last (contradiction).

Induction step: Pick an object x and remove all calls on x from H; call this history H''. The last call m in the linearization of Hx is maximal in Hx. By the induction hypothesis, there is a last call m'' in either of the object linearizations for H'' that is maximal in H''. If m'' -> m, m is maximal in H', if m -> m'', m'' is maximal in H', and if m and m'' are overlapping either is maximal in H'.

<u>Theorem</u>: If <x.m A> is a pending invocation of method m of x in a linearizable H, then there exist a matching response <x:v A> such that H <x:v A> is linearizable

Thus, linearizability does not force a thread (here A) to block; linearizable H can be extended with matching responses

Proof:
Let S be a linearization of H. If S already includes <x:v A>, then S is also linearization of H <x:v A>. Otherwise, S does not include <x.m A> (linearizations do not have pending invocations). There exists a response such that S <x.m A> <x:v A> is legal. This is a linearization of H <x:v A>

Note: Assumed that method calls are total, a response per sequential specification always possible

Example:
History H of LIFO stack operations, two pending Push operations, Pop 3 after Pop 2. Is this history linearizable?

©Jesper Larsson Träff

Example:
History H of LIFO stack operations, two pending Push operations, Pop 3 after Pop 2. For the linearization, H must first be extended with responses to the two push operations

©Jesper Larsson Träff

Example:
History H of LIFO stack operations, two pending Push operations, Pop 3 after Pop 2. A possible (only?) linearization is this sequential history:

Example:
History H of LIFO stack operations, two pending Push operations, Pop 3 after Pop 2. The linearization has linearization points in this order:

©Jesper Larsson Träff

Definition:

History H is sequentially consistent if there is an extension H' and a legal sequential history S such that

1. complete(H') is equivalent to S
2. if m0 -> m1 in H|A for method calls m0, m1 for some thread A, then m0->m1 in S

Recall, linearization has:

2. if m0 -> m1 in H for method calls m0, m1, then m0 -> m1 in S.

Corollary:
Linearizability implies sequential consistency (not vice versa)

<u>Theorem</u> (without proof): Sequential consistency is non-blocking

But sequential consistency is not compositional (see previous counterexamples)

©Jesper Larsson Träff

The proof of compositionality (if-part) for linearizability breaks down for sequential consistency:

The linearization points of each H|x may change (slide) the real-time history for object x. Thus no guarantee that any last m in an Sx will be maximal in H

©Jesper Larsson Träff

The proof of compositionality (if-part) for linearizability breaks down for sequential consistency:

The linearization points of each H|x may change (slide) the real-time history for object x. Thus no guarantee that any last m in an Sx will be maximal in H

©Jesper Larsson Träff

The proof of compositionality (if-part) for linearizability breaks down for sequential consistency:

The linearization points of each H|x may change (slide) the real-time history for object x. Thus no guarantee that any last m in an Sx will be maximal in H

©Jesper Larsson Träff

Linearizability:

- Method calls should appear to happen in a one-at-a-time ("atomic"), sequential order
- Each method call should appear to take effect instantaneously at some point between invocation and response event

Definition:
History H is linearizable if there is an extension H' and a legal sequential history S such that
1. complete(H') is equivalent to S
2. if m0 -> m1 in H for method calls m0, m1, then m0->m1 in S

S is called a linearization of H

End of technicalities (part II)

End of digression

What "composable" does not mean

Composite operations consisting of component operations on different, linearizable objects are not necessarily linearizable. Most often not

Example: Accounts A and B with operations deposit(), withdraw(), balance()

Transfer:

```
...
A.withdraw(100);
B.deposit(100);
...
```

Total:

```
...
total = A.balance();
total += B.balance();
...
```


©Jesper Larsson Träff

Transfer:

```
...
A.withdraw(100);


B.deposit(100);
...
```

Total:

```
...
total = A.balance();
total += B.balance();
...
```

Indicated history is not linearizable (correct)

Example: Start with 500 in both A and B, total will be 900, and incorrect

Would need to create new data structure with new operations transfer(A,B,amount), and total(A,B)
Transactions can help (put the two operations into transactions)

Some desirable concurrent objects/data structures:

FIFO, LIFO queues, lists, … with operations like their sequential counterparts (enqueue/dequeue, push/pop, insert/search/delete, …)

SEE NEXT LECTURES FOR HOW TO IMPLEMENT THESE

Assume such data structures:
Can they solve the consensus problem? In other words: What are their power to solve concurrent coordination problems; and what is required to implement them?

<u>Theorem</u>:
Wait-free FIFO queues have consensus number at least 2

Proof:
By construction. Enqueue two values WIN and LOOSE, decide ("who was first"?) by dequeuing and checking value

©Jesper Larsson Träff

```
class FIFOconsensus extends Consensus {
  private static final int WIN = 0; // elt for 1st thread
  private static final int LOSE = 1; // elt for 2nd thread
  Queue queue;
  public FIFOconsensus() {
    queue = new Queue();
    queue.enq(WIN); queue.enq(LOOSE);
  }
  public T decide(T value) {
    propose(value);
    int i = ThreadID.get();
    int elt = queue.deq();
    if (elt==WIN) return proposed[i]; // thread won
    else return proposed[1-i]; // thread lost
  }
}
```


©Jesper Larsson Träff

Corollary:
Atomic registers do not suffice to construct a wait-free FIFO queue!

By implication; similar constructions (exercise):
Atomic registers do not suffice to construct wait-free lists, stacks, sets, bags, priority queues, …

Proofs: See Exercises…

Theorem:
FIFO queues have consensus number exactly 2

Proof:
By contradiction. Assume a binary consensus protocol for 3 threads A, B, C exists.

The protocol has a critical state. Let A's next move lead to a 0-valent state, B's next move to a 1-valent state (C indifferent).

Observe: A and B moves cannot commute (otherwise contradiction, indistinguishable state, cannot be both 0- and 1-valent), so A and B moves must be method calls of same shared resource. A and B cannot be about to read/write registers.

Thus A and B's moves must be method calls of a single shared FIFO queue object

3 possibilities:

1. A and B both call `deq()`
2. A calls `enq(a)`, B calls `deq()`
3. A and B both call `enq()`

See also:

M. Raynal: Concurrent programming: Algorithms, principles and foundations. Springer, 2013

with more careful arguments

©Jesper Larsson Träff

1. A and B both calls deq( )

Two sequences:

1. A deq( ), B deq( ) 0-valent s', at most 2 items gone
2. B deq( ), A deq( ) 1-valent s'', at most 2 items gone

But states s' and s'' are indistinguishable to C, so C must decide same value. Contradiction!

2. A calls `enq(a)` and B calls `deq()`

If queue is nonempty, the two operations commute, so both sequences

1. A `enq(a)`, B `deq(x)`, 0-valent s'
2. B `deq(x)`, A `enq(a)`, 1-valent s''

lead to states s' and s'' that are again indistinguishable to C, so C must decide same value from both. Contradiction

If queue is empty, the sequences

1. A `enq(a)`, 0-valent s'
2. B `deq()`, A `enq(a)`, 1-valent s''

Contradiction again

lead to states s' and s'' that are indistinguishable to C.

3. A and B both call `enq()`

1. First A `enq(a)`, then B `enq(b)`. Let A run until `deq(a)`, then run B until `deq(b)`. State s' is 0-valent (A moved first)
2. First B `enq(b)`, then A `enq(a)`. Let A run until `deq(b)`, then run B until `deq(a)`. State s" is 1-valent (B moved first)

Ad:

1. <x1,x2,…,xk,a> is the FIFO state where global state became 0-valent
2. <x1,x2,…,xk,b> is the FIFO state where global state became 1-valent

The two states are indistinguishable to A and B until the elements x1, x2, …, xk have been dequeued. In order for A or B to know what the decision value is, k+1 values must be dequeued.

A, B, C make moves from initial state, assume k enq( ) operations

But none of these enq( ) operations make state univalent

Critical state, next move by A will lead to 0-valent state, next move by B to 1-valent state (C could be either

FIFO queue with k elements

d=0/1

A enq(a) <x1,x2,…,xk,a>

d=0

B enq(b), A runs until deq(a) <x1,x2,…,xk,a,b>

<b,y1,y2,…>

d=0

B deq(b) <y1,y2,…>

s'

©Jesper Larsson Träff

A, B, C make moves from initial state, assume k enq( ) operations

But none of these enq( ) operations make state univalent

Critical state, next move by A will lead to 0-valent state, next move by B to 1-valent state (C could be either

FIFO queue with k elements

©Jesper Larsson Träff

3. A and B both call `enq()`

1. First A `enq(a)`, then B `enq(b)`. Let A run until `deq(a)`, then run B until `deq(b)`. State s' is 0-valent (A moved first)
2. First B `enq(b)`, then A `enq(a)`. Let A run until `deq(b)`, then run B until `deq(a)`. State s" is 1-valent (B moved first)

In order for A to return d=0 (in case 1) it must observe the state of the queue by `deq(x)`, and x=a since A moved first. Similarly for case 2.

But states s' and s" are indistinguishable to C, since the same number of elements have been dequeued (and enqueued, and updates to registers cannot help), so C must decide same value from both. Contradiction

Hardware support(I)

R(ead)M(odify)W(rite) operations: Special instructions/methods that atomically read and update a register r. Examples

`r.getandset(v):`	returns value of r, replaces r with v
`r.getandinc():`	returns value of r, increments r
`r.getandadd(k):`	returns value of r, replaces r with r+k
`r.get():`	returns value of r

Sometimes called: test-and-set, fetch-and-add, …

Formally: A method is an RMW method for a function class F, if there is some f in F, such that it atomically returns current register value r and replaces r with f(r)

`r.getandset(v)`:	returns value of r, replaces r with v F: constant functions f.v(x) = v
`r.getandinc()`:	returns value of r, increments F: f(x) = x+1
`r.getandadd(k)`:	returns value of r, replaces r with r+k F: f.k(x) = x+k
`r.get()`:	returns value of r F: f(x) =x trivial

Formally: A method is an RMW method for a function class F, if there is some f in F, such that it atomically returns current register value r and replaces r with f(r)

An RMW method/register is nontrivial if at least one f in F is not the identity function

Theorem:
Any non-trivial RMW register has consensus number at least 2

Proof: by construction. Consensus protocol for 2 threads

```
class RMWconsensus extends Consensus {
  // initialize to v such that f(v)!=v
  private RMWRegister r = new RMWRegister(v);
  public Object decide(Object value) {
    propose(value);
    int i = ThreadID.get();
    if (r.RMW()==v) return proposed[i]; // thread won
    else return proposed[1-i]; // thread lost
  }
}
```

Specific RMW methods

A set of functions F is in COMMON iff for any fi and fj in F, either

- fi(fj(x)) = fj(fi(x)) (fi and fj commute)
- fi(fj(x)) = fi(x) or fj(fi(x)) = fj(x) (either overwrites the other)

COMMON RMW methods:

`r.getandset(v)`: returns value of r, replaces r with v
f.v(x) = v overwrites

`r.getandinc()`: returns value of r, increments
f(f(x)) = x+1+1 commutes

`r.getandadd(k)`: returns value of r, replaces r with r+k
f.k(f.j(x)) = x+k+j commutes

Theorem:
Any RMW register in COMMON has consensus number 2

Proof:
By contradiction. Assume a 3-consensus protocol for threads A, B and C exists.

A critical state exists. The next move by either thread cannot be register read/write, and the moves cannot commute. Thus either next move must be an RMW update to a single register r with some function f in COMMON.

Assume A is about to update r with fA, B about to update with fB

Two possible cases:

1. fB overwrites fA, fB(fA(x)) = fB(x)
2. fA and fB commute, fA(fB(x)) = fB(fA(x))

©Jesper Larsson Träff

1. fB overwrites fA, fB(fA(x)) = fB(x)

Consider two sequences

- A moves, fA(x), B moves, fB(fA(x)) = fB(x), 0-valent s'
- B moves, fB(x), 1-valent s"

However, states s' and s" are indistinguishable to C, since the value of r is the same and only the internal states of A and B can differ. Thus C should decide same value in both s' and s".
Contradiction

2. fA and fB commute, fA(fB(x)) = fB(fA(x))

Consider two sequences

- A moves fA(x), then B moves fB(fA(x)), 0-valent s'
- B moves fB(x), then A moves fA(fB(x)), 1-valent s''

Again, states s' and s'' that can differ only in the internal states of A and B, are indistinguishable to C, so C should decide same value in both. Contradiction.

<u>Corollary</u>:
FIFO queues (stack, …) have the same consensus number as COMMON RMW, namely 2.

This means that it might well be possible to construct a wait-free queue (stack, …) using only COMMON RMW registers and atomic registers.

Hardware support(II)

An atomic SWAP operation swaps the contents of two registers

```
r1.SWAP(r2): atomically:
temp = r1; r1 = r2; r2 = temp;
```

The atomic SWAP is not an RMW operation, it operates on two registers

Theorem: SWAP has consensus number n

Proof: By construction. Consensus protocol for n threads

```
class SWAPconsensus extends Consensus {
  private AtomicInteger R = new AtomicInteger(1);
  private AtomicInteger A[n] =
    new AtomicInteger[n](0);
  public Object decide(Object value) {
    propose(value);
    int i = ThreadID.get();
    A[i].SWAP(R); // first i gets 1
    for (i=0; i<n; i++)
      if (A[i]==1) return proposed[i];
  }
}
```


©Jesper Larsson Träff

The `decide()` method is clearly wait-free (bounded loop)

First thread to call `decide()` and execute swap has A[i]==1 (from the initial value of R), all other threads A[j]==0, regardless of whether they have called `decide()` or not

Alternatively:
The algorithm maintains the invariant that $R+\sum_{0\leq i<n}A[i]=1$

Note:
The number of threads n must be known in advance and remain fixed.

Hardware support(III)

Compare-and-swap (CAS): Extended RMW operation on register r that atomically compares current content of r to an expected value e and if same replaces with an update value u, returns true if update was performed, false if not (alternatively: returns current content)

```
r.CAS(e,u): atomically:
if (r.get()==e) { r.set(u); return true; }
else return false;
```

Sometimes called test-and-set, compare-and-exchange, …., and sometimes returns old value of register r

CAS supported by Intel, AMD, Sun/Oracle Sparc, …

Alternative compare-and-exchange/load (CEX) atomically compares current content of r to an expected value e and if same replaces with an update value u, returns true if update was performed, returns false if not and updates u (reference) to e (reference) atomically

```
r.CASL(e,u): atomically:
if (r.get()==e) { r.set(u); return true; }
else { u.set(e); return false; }
```

CEX semantics of the C11/C++ `atomic_compare_exchange` operation

LL/SC (load linked, store conditional):

LL reads (loads) address a; the subsequent SC stores a new value to a, if a has not changed since the LL; if a has changed SC fails, and nothing happens to a

Note:
In practical, HW implementations of LL/SC there may be other conditions leading SC to fail (distance/time between LL and SC too long)

LL/SC supported by IBM PowerPC, MIPS 11, ARM, …

RISC-V LR/SC (load reserved/store conditional)

```
# emulated RISC-V CAS
0:      lr.w a3,(a0)          # load reserved
        bne a3,a1,80          # not equal?
        sc.w a3,a2,(a0)       # store conditional
                              # (succeed if no change)
        bnez a3,0             # retry (wait-free?)
80:
```

ISA design argument for not supporting CAS: has three source registers, would require new operand instruction format

David Patterson, Andrew Waterman: The RISC-V Reader: An open architecture atlas. Strawberry Canyon LLC, 2017

Other RISC-V atomics (amo: atomic memory operation)

```
amo{add, and, or, swap, xor, max, min}.word
```

all with acquire/release bits (see later)

©Jesper Larsson Träff

Theorem: CAS has consensus number ∞

Proof: By construction. Consensus protocol for any number of threads (number threads does not need to be known in advance)

```
class CASconsensus extends Consensus {
  private int first = -1;
  private AtomicInteger r = new AtomicInteger(first);
  public Object decide(Object value) {
    propose(value);
    int i = ThreadID.get();
    if (r.CAS(first,i))
      return proposed[i]; // thread won
    else return proposed[r.get()]; // thread lost
  }
}
```

<u>Theorem:</u> LL/SC has consensus number ∞

Proof: By construction. Exercise

©Jesper Larsson Träff

Universality of consensus (Chap. 6)

Fundamental result: There exist classes of objects, such that given enough of them it is possible to construct wait-free, linearizable implementations of any sequentially specified concurrent object

An object with this property is called universal

Theorem: Objects with consensus number ∞ are universal

Corollary: CAS is universal, LL/SC is universal

Proof by construction; constructions explicit but very inefficient

Idea:
Use consensus to linearize concurrent operations on the sequentially specified object; each thread locally performs the sequence of operations on a local copy of the object to compute response.

The deterministic, sequential object is specified by a generic interface. The invoke() method is the invocation which returns a response

```
public interface SeqObject {
  public abstract Response invoke();
}
```

Linearized execution history represented by a list of Node's

```
public class Node {
  public Invoc invoc; // parameters
  public Consensus<Node> decideNext;
  public Node next;
  public int seqnum;
  public Node(Invoc invoc) {
    invoc = invoc;
    decideNext = new Consensus<Node>;
    seqnum = 0;
  }
  public static Node max(Node[] array) {
    return maximum(array[i].seqnum for some i);
  }
}
```


©Jesper Larsson Träff

A history of k invocations

Thread i:
To compute response after k invocations, allocate local object, start with initial invoc-parameters, follow next pointers through list, update object state at each node

©Jesper Larsson Träff

A history of k invocations

Sentinel node

```
SeqObject object = new SeqObject();
Node current = tail.next;
while (current!=prefer) {
  object.invoke(current.invoc);
  current = current.next;
}
return object.invoke(current.invoc);
```

Lock-free universal construction

Invocation by thread i:
Allocate node, store arguments, use consensus to agree when it can be appended to list

©Jesper Larsson Träff

Lock-free universal construction

Invocation by thread i:
Allocate node, store arguments, use consensus to agree when it can be appended to list

Head-array shared by all threads for each thread references threads last head

©Jesper Larsson Träff

Initializing head: Each thread points to fixed tail node

```
public class LockFreeUniversal {
  private Node[] head;
  private Node tail; // sentinel Node
  public Universal() {
    tail = new Node();
    tail.seqnum = 1;
    for (i=0; i<n; i++) head[i] = tail;
  }
  public Response invoke(Invoc invoc);
}
```

Invariant: `head[i]` is last Node that thread i appended; seqnum's in list are successive (from tail.seqnum==1)

To append new invocation by thread i (= linearize operation):

- Get overall last node (max seqnum in head[j] over all j)
- Apply consensus using this node's consensus object to find out which node shall be appended
- Append to list (note: more than one thread may do so, but they will do the same updates)

Repeat until own node is successfully appended (seqnum has been set)

```
public Response invoke(Invoc invoc) {
  int i = ThreadID.get();
  Node prefer = new Node(invoc); // new Node
  while (prefer.seqnum==0) {
    Node before = Node.max(head);
    Node after = before.decideNext(prefer);
    before.next = after;
    after.seqnum = before.seqnum+1;
    head[i] = after;
  }
  SeqObject object = new SeqObject();
  Node current = tail.next;
  while (current!=prefer) {
    object.invoke(current.invoc);
    current = current.next;
  }
  return object.invoke(current.invoc);
}
```

Get last

Agree

Append

prefer Node successfully appended; compute result

©Jesper Larsson Träff

<u>Theorem</u>: The construction correctly implements the concurrent object and is lock-free

Proof:
On concurrent invocations, the use of the last Node's consensus object linearizes by choosing either of the invocations. This node is inserted correctly. Thread's "loosing" the consensus, reinsert a Node already in the list, but this is no harm.
Each thread calls each consensus object at most once: the outcome of the consensus is stored in head[i], so a different object will be used in the next iteration.
Since a thread will iterate only when it looses consensus to some other thread, some thread is guaranteed to make progress and the construction is lock-free.

Note:
The values proposed to the consensus objects are Node references

Swept under the rug:
Memory management, must be wait-free (can be handled)

Wait-free universal construction

In the lock-free construction, any thread can be overtaken an indefinite number of times. In a wait-free construction, each thread must be able to complete in a finite number of steps, regardless of what other threads do (outcome of consensus protocol)

To prevent this, use helper scheme: Each thread announces the node it wants to append, if unsuccessful in consensus, some other thread will insert the node

```
public class Universal {
  private Node[] announce; // announced Nodes
  private Node[] head;
  private Node tail = new Node(); tail.seqnum = 1;
  for (j=0; j<n; j++) {
    head[j] = tail; announce[j] = tail;
  }
  public Response invoke(Invoc invoc);
}
```

```
public Response invoke(Invoc invoc) {
  int i = ThreadID.get();
  announce[i] = new Node(invoc);
  head[i] = Node.max(head);
  while (announce[i].seqnum==0) {
    Node before = head[i];
    Node help = announce[(before.seqnum+1)%n];
    if (help.seqnum==0) prefer = help;
    else prefer = announce[i];
    Node after = before.decideNext(prefer);
    before.next = after;
    after.seqnum = before.seqnum+1;
  }
  SeqObject object = new SeqObject();
  Node current = tail.next;
  while (current!=announce[i]) {
    object.invoke(current.invoc);
    current = current.next;
  }
  head[i] = announce[i];
  return object.invoke(current.invoc);
}
```

Announce

Need help?

Try to help

prefer Node successfully appended; compute result

<u>Theorem</u>: The construction correctly implements the concurrent object and is wait-free

Proof: For details, see Herlihy-Shavit book

Some fundamental papers upon which this is all based:

Maurice Herlihy, Jeannette M. Wing: Linearizability: A Correctness Condition for Concurrent Objects. ACM Trans. Program. Lang. Syst. 12(3): 463-492 (1990)
Maurice Herlihy: Wait-Free Synchronization. ACM Trans. Program. Lang. Syst. 13(1): 124-149 (1991)

Great for seminar talks

More efficient, universal constructions

Panagiota Fatourou, Nikolaos D. Kallimanis: Highly-Efficient Wait-Free Synchronization. Theory Comput. Syst. 55(3): 475-520 (2014)
Panagiota Fatourou, Nikolaos D. Kallimanis: The RedBlue family of universal constructions. Distributed Comput. 33(6): 485-513 (2020)
Yehuda Afek, Dalia Dauber, Dan Touitou: Wait-free made fast (Extended Abstract). STOC 1995: 538-547
Alex Kogan, Erez Petrank: A methodology for creating fast wait-free data structures. PPOPP 2012: 141-150
James H. Anderson, Mark Moir: Universal Constructions for Large Objects. IEEE Trans. Parallel Distrib. Syst. 10(12): 1317-1332 (1999)
Andreia Correia, Pedro Ramalhete, Pascal Felber: A wait-free universal construction for large objects. PPoPP 2020: 102-116

Lower consensus operations in terms of higher consensus

Fetch-and-add can easily be implemented using compare-and-swap

```
class FAA extends RMWregister {
  private RMWRegister r = new RMWRegister();
  public int fetchandadd(int k) {
    int v = r.get();
    while (!r.CAS(v,v+k)) v = r.get();
    return ri;
  }
}
```

The implementation is correct and lock-free, but not wait-free. When a CAS fails for thread A, some other thread has successfully added its k. Challenge: Wait-free FAA by CAS

Challenge: Wait-free FAA by CAS.

See

Faith Ellen, Philipp Woelfel: An Optimal Implementation of Fetch-and-Increment. DISC 2013: 284-298

for a partial solution using LL/SC

Extended compare-and-swap

Atomically, compare n values to n registers, and update all

```
class CASn {
  private RMWRegister r1, r2, …, rn;
  public boolean CASn(r1,r2,…,rn,
                      e1,e2,…,en,u1,u2,…,un) {
    // atomically, synchronized
    if (e1==r1.get()&&e2==r2.get()&&…en==rn.get()) {
      r1.set(u1);
      r2.set(u2);
      …
      rn.set(un);
      return true;
    } else return false;
  }
}
```

Atomically, compare k values to k registers, and update one of the registers

```
class kCAS {
  private RMWRegister r[k];
  public Boolean kCAS(r[],e[],j,u) {
    for (i=0; i<k; i++) if (e[i]!=r[i].get())
      return false;
   r[j].set(u);
   return true;
}
```

These operations are sometimes useful (needed) for concurrent data structures, but usually not (?) supported in hardware.
Alternative: Emulate (challenge), or rely on transactional memory

Transactional memory (example)

Transactional memory enables linearizability at a higher granularity.

Transaction: Sequence of instructions that are (semantically) executed atomically.

Transactions must be serializable, that is appear to execute sequentially in a one-a-at-time order, that is appear to take place instantaneously. Transactions are non-blocking.

Not Java, but what transactional models provide

```
atomic {
  x = q0.deq();
  q1.enq(x);
}
```

Transaction

Even when transactions occur concurrently, no thread can observe that x has been removed from q0 but is not present in q1, or that x is present in both q0 and q1, or …

©Jesper Larsson Träff

Implementation (hardware or software)

Transactions are executed speculatively, at the end of a transaction it is checked that atomicity is not violated, and if so, the transactions commits; if atomicity is violated, the transaction aborts

Hardware transactional support, e.g., in Intel, from Haswell architecture (THX).

Advanced Multiprocessor Programming: Locks

Jesper Larsson Träff, Martin Wimmer

TU Wien

April 24th, 2023

Locks we have seen so far

- Peterson Lock
- Filter Lock
- Bakery Lock

These locks ...

... use atomic registers

... have time and space complexity $O(n)$

... spin (busy waiting)

n: Number of threads

Problems with locks, recap

- Not compositional: Error prone, can lead to deadlocks if different locking conventions are used
- Priority inversion: In interaction with OS scheduler, a low priority thread holding a lock can block a high priority thread indefinitely by not being allowed to run because of a medium priority thread being scheduled
- Lock convoying: In interaction with OS scheduler, a descheduled thread holding a lock can cause other threads to queue up waiting for the lock; context switches between queued threads can be expensive

A typical lock interface

```
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *mutexattr);

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            const struct timespec *abs_timeout);

int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

To spin or not

Two types of lock implementations:

- Spin lock: Thread actively keeps trying (test some condition) to acquire lock (Peterson lock, Bakery lock, etc. are spin locks)
- Blocking lock: Thread suspends, scheduler (OS) reawakens thread when lock becomes free

Performance considerations:

- Spin lock: Good if critical section short (fast), low overhead; but keeps processor busy ("burns cycles")
- Blocking lock: Good for long critical sections, high (OS) overhead, processor/core can do something else

For coarse grained locking (pthreads), combinations often used: Spin for some time, then back off, then block

Additional pthread locks

```
#include <pthread.h>

int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);

int pthread_spin_unlock(pthread_spinlock_t *lock);
```

(Strange) Design decision: Programmer needs to rewrite code to use spin locks

Alternative (why not?): Use attribute at initialization to fix pragmatics

Higher consensus operations

- Consensus number 1
  - Atomic registers
- Consensus number 2
  - (flag) test-and-set
  - get-and-set (aka: exchange, swap)
  - fetch-and-add (also: fetch-and-inc, fetch-and-X)
  - wait-free FIFO queues (also: stack, list, ...)
- Consensus number n
  - swap (aka: exchange)
- Consensus number ∞
  - compare-and-swap (cas, aka: compare-exchange)
  - load-linked - store-conditional (ll/sc)

C11 atomics

```
#include <stdatomic.h>

typedef /* unspecified */ atomic_flag;
_Bool atomic_flag_test_and_set( volatile atomic_flag* obj );
_Bool atomic_flag_test_and_set_explicit( volatile atomic_flag* obj, memory_order
     order );

C atomic_exchange( volatile A* obj, C desired );
C atomic_exchange_explicit( volatile A* obj, C desired, memory_order order );


_Bool atomic_compare_exchange_strong( volatile A* obj,
                                      C* expected, C desired );
_Bool atomic_compare_exchange_weak( volatile A *obj,
                                    C* expected, C desired );
_Bool atomic_compare_exchange_strong_explicit( volatile A* obj,
                                               C* expected, C desired,
                                               memory_order succ,
                                               memory_order fail );
_Bool atomic_compare_exchange_weak_explicit( volatile A *obj,
                                             C* expected, C desired,
                                             memory_order succ,
                                             memory_order fail );
C atomic_fetch_add( volatile A* obj, M arg );
C atomic_fetch_add_explicit( volatile A* obj, M arg, memory_order order );
C atomic_fetch_sub( volatile A* obj, M arg );
C atomic_fetch_sub_explicit( volatile A* obj, M arg, memory_order order );
// also: or, xor, and

void atomic_thread_fence( memory_order order );
```

Weak and strong and explicit compare-exchange

C/C++ beware: A failed compare-exchange is not a no-op, but loads the contents of the object into the location given for the expected value!

A strong compare-exchange must succeed if expected and memory values are indeed equal. A weak compare-exchange is allowed to not succeed and return false, even in case expected and memory values are equal ("spuriously"), but not indefinitely.

Rationale: Some hardware sacrifices strong compare-and-swap for sometimes better performance.

Compare-exchange operations are often used in loops, in some cases weak compare-exchange is correct and can give better performance. But not always.

Explicit atomics explicitly specify conditions on the memory model behavior (later lecture).

Header <atomic>

```
template< class T > struct atomic;
template<> struct atomic<Integral>;
template< class T > struct atomic<T*>;

// Member functions
(constructor) // constructs an atomic object
operator= // stores a value into an atomic object
is_lock_free // checks if the atomic object is lock-free
store // atomically replaces the value of the atomic object with a non-atomic
      argument
load //atomically obtains the value of the atomic object
operator T // loads a value from an atomic object
exchange // atomically replaces the value of the atomic object and obtains the
      value held previously
compare_exchange_weak // atomically compares the value of the atomic object with
      non-atomic argument and performs atomic exchange if equal or atomic load if
      not
compare_exchange_strong
```

Header <atomic>

```
// Specialized member functions
fetch_add // atomically adds the argument to the value stored in the atomic object
      and obtains the value held previously
fetch_sub // atomically subtracts the argument from the value stored in the atomic
      object and obtains the value held previously
fetch_and // atomically performs bitwise AND between the argument and the value of
      the atomic object and obtains the value held previously
fetch_or // atomically performs bitwise OR between the argument and the value of
      the atomic object and obtains the value held previously
fetch_xor // atomically performs bitwise XOR between the argument and the value of
      the atomic object and obtains the value held previously

operator++
operator-- // increments or decrements the atomic value by one
operator+=
operator-=
operator&=
operator|=
operator^= // adds, subtracts, or performs bitwise AND, OR, XOR with the atomic
      value
```

C11/C++11 threads

Native thread-model (somewhat like pthreads) with locks (mutexes) since C++11/C11. Use this (or pthreads)

```
#include <thread>

void myfunction()
{
  // make sure all threads start more or less at the same time
  // use mutex-condition variable, or (experimental) barrier construct

}
int main()
{
  // create and start threads (tree like, for larger number of threads
  std::thread t1(myfunction), t2(myfunction), t3(myfunction), t4(myfunction);

  // threads running, join back at end
    t1.join();
    t2.join();
    t3.join();
    t4.join();
}
```

Threads with OpenMP for C/C++

OpenMP: Fork-join thread-model programming language extension. Can be used to easily start and synchronize threads.

```
#include <omp.h>

void benchmark() // called by master thread
{
  double threadtime[omp_get_max_threads()];

#pragma omp parallel // further clauses to control number of threads etc.
  {
    // threads active, per thread scope
    int p = omp_get_num_threads();
    int t = omp_get_thread_num();
    double start, stop;

#pragma omp barrier
    start = omp_get_wtime();

    // code to be benchmarked

    stop = omp_get_wtime();
    threadtime[t] = stop-start; // and other statistics
  }
  // only master thread active
}
```

Threads with OpenMP for C/C++

OpenMP: Fork-join thread-model programming language extension. Can be used to easily start and synchronize threads.

For a concrete example, see the code for

> Jesper Larsson Träff, Manuel Pöter: A more pragmatic implementation of the lock-free, ordered, linked list. PPoPP 2021: 457-459

which is available via
`https://github.com/parlab-tuwien/lockfree-linked-list`

Cost of atomic operations

- All O(1), constant time, and wait-free instructions, imlemented efficiently in hardware (CA: How?)
- But how large are the constants? And same O(1) under all circumstances?
- Reasonable for `exhange` and `fetch_add`?
  - These must always succeed, high contention could cause delays

For a paper attempting to assess costs of atomic operations, see Hermann Schweizer, Maciej Besta, Torsten Hoefler: Evaluating the Cost of Atomic Operations on Modern Architectures. PACT 2015: 445-456

Higher consensus operations in action:

- Test-And-Set Lock
  - Single flag field per lock
  - Acquire lock by changing flag from false to true → locked on success
  - Reset flag to unlock

```
bool locked = false; // atomic register

void lock() {
  while (test_and_set(&locked));
}

void unlock() {
  locked = false;
}
```

- Performance (surprisingly?) bad (why?)
  - Each test-and-set call invalidates cached copies for all threads
  - High contention on memory interconnect

Test-And-Test-And-Set lock

- Test-and-set only if there is a chance of success
- Cache invalidated less often
- Still contention with more threads
- Relies on cache-coherence protocol implementation
- Beware of compiler optimizations (`volatile`)!

```
volatile bool locked = false;

void lock() {
  do {
    while (locked);
    if (!test_and_set(&locked)) return;
  } while (true);
}

void unlock() {
  locked = false;
}
```

Exponential Backoff

- On failure to acquire lock:
  → Backoff with random duration
- Increase time to wait exponentially
  - Reduces contention
  - Try less often on high contention
  - Randomization ensures that threads wake up at different times
- Threads might wait longer than necessary!
- C++ note: Don't use rand()
  → Not thread-safe: Uses locks inside!

  (Here: Thread-safe pseudo-function `rnd()`)

```
class Backoff {
  int limit = MIN_DELAY;

  void backoff() {
    int delay = rnd() % limit;
    limit =
      min(MAX_DELAY, limit*2);
    sleep(delay); // suspend for
          some time
  }
}
```

```
volatile bool locked = false;

void lock() {
  Backoff bo;
  do {
    while (locked);
    if (!test_and_set(&locked))
      return;
    bo.backoff();
  } while (true);
}

void unlock() {
  locked = false;
}
```

Test-And-Set-Locks: Summary

- Space complexity $O(1)$ for ∞ threads
  - Possible by test-and-set (consensus number 2)
- Unlock wait-free $O(1)$

- Problem with memory contention
  - All threads spin on a single memory location
    (cache coherence traffic)
- Threads might wait longer than necessary due to backoff
- Unfair, and not starvation free
- "Fault-tolerant": Stalled (crashed) thread not having lock will not prevent other threads from using lock

Thread programming needs thread safety

All functions called by a thread must be thread-safe: Can be called concurrently, outcomes depend only on thread-local state:

- Pure functions: No side-effects, no (global) state
- All functions called must likewise be thread-safe

Fair locks: Ticket Lock

- Atomic ticket counter
- Non-atomic served counter
- Lock: Take ticket and wait for served
- Unlock: Serve next

```
int ticket = 0; // atomic
volatile int served = 0;

void lock() {
  int next = fetch_add(&ticket,1)
  while (served<next) {};
}

void unlock() {
  served++;
}
```

Ticket Lock: Properties

- First-come, first-served, linearization by fetch-and-add
- Space efficient: $O(1)$ per lock!
- Unlock wait-free $O(1)$
- Ticket grows unboundedly
- Each thread spins on a local copy of a variable (in cache)
- False sharing might occur (if both served and ticket variables on same cache-line)
- Slow (stalled, crashed) thread will stall later threads. Not "fault-tolerant"

Too strong fairness; the following locks will have the same problem.
Problem: all threads before calling thread must be served.

Fair locks: Array Lock

- Circular array of atomic memory locations
- Lock: Spin on next free slot
- Unlock: Release slot

```
bool flags[n] =
  {true, false, false, false, ...}
int tail = 0; // atomic
thread_local int mySlot;

void lock() {
  mySlot = fetch_add(&tail,1) % n;
  while(!flags[mySlot]) {};
}

void unlock() {
  flags[mySlot] = false;
  flags[(mySlot + 1) % n] = true;
}
```

$\texttt{tail} = \boxed{0}$

Fair locks: Array Lock

- Circular array of atomic memory locations
- Lock: Spin on next free slot
- Unlock: Release slot

```
bool flags[n] =
  {true, false, false, false, ...}
int tail = 0; // atomic
thread_local int mySlot;

void lock() {
  mySlot = fetch_add(&tail,1) % n;
  while(!flags[mySlot]) {};
}

void unlock() {
  flags[mySlot] = false;
  flags[(mySlot + 1) % n] = true;
}
```

$\texttt{tail} = \boxed{1}$

Fair locks: Array Lock

- Circular array of atomic memory locations
- Lock: Spin on next free slot
- Unlock: Release slot

```
bool flags[n] =
  {true, false, false, false, ...}
int tail = 0; // atomic
thread_local int mySlot;

void lock() {
  mySlot = fetch_add(&tail,1) % n;
  while(!flags[mySlot]) {};
}

void unlock() {
  flags[mySlot] = false;
  flags[(mySlot + 1) % n] = true;
}
```

$\texttt{tail} = \boxed{2}$

Fair locks: Array Lock

- Circular array of atomic memory locations
- Lock: Spin on next free slot
- Unlock: Release slot

```
bool flags[n] =
  {true, false, false, false, ...}
int tail = 0; // atomic
thread_local int mySlot;

void lock() {
  mySlot = fetch_add(&tail,1) % n;
  while(!flags[mySlot]) {};
}

void unlock() {
  flags[mySlot] = false;
  flags[(mySlot + 1) % n] = true;
}
```

$\texttt{tail} = \boxed{3}$

Fair locks: Array Lock

- Circular array of atomic memory locations
- Lock: Spin on next free slot
- Unlock: Release slot

```
bool flags[n] =
  {true, false, false, false, ...}
int tail = 0; // atomic
thread_local int mySlot;

void lock() {
  mySlot = fetch_add(&tail,1) % n;
  while(!flags[mySlot]) {};
}

void unlock() {
  flags[mySlot] = false;
  flags[(mySlot + 1) % n] = true;
}
```

$\texttt{tail} = \boxed{3}$

Fair locks: Array Lock

- Circular array of atomic memory locations
- Lock: Spin on next free slot
- Unlock: Release slot

```
bool flags[n] =
  {true, false, false, false, ...}
int tail = 0; // atomic
thread_local int mySlot;

void lock() {
  mySlot = fetch_add(&tail,1) % n;
  while(!flags[mySlot]) {};
}

void unlock() {
  flags[mySlot] = false;
  flags[(mySlot + 1) % n] = true;
}
```

$\texttt{tail} = \boxed{3}$

Fair locks: Array Lock

- Circular array of atomic memory locations
- Lock: Spin on next free slot
- Unlock: Release slot

```
bool flags[n] =
  {true, false, false, false, ...}
int tail = 0; // atomic
thread_local int mySlot;

void lock() {
  mySlot = fetch_add(&tail,1) % n;
  while(!flags[mySlot]) {};
}

void unlock() {
  flags[mySlot] = false;
  flags[(mySlot + 1) % n] = true;
}
```

$\texttt{tail} = \boxed{3}$

Array Lock: Properties

- First-come, first-served, linearization by fetch-and-add
- Unlock wait-free $O(1)$
- Less contention than simple test-and(-test-and)-set lock
  - Each thread spins on a local copy of a variable
  - False sharing might occur, resolvable by padding
- Not space efficient: $O(n)$ per lock!
- Not fault-tolerant

False sharing: Unrelated addresses on same cache-line still entail remote cache invalidation/updates.

CLH Queue Lock

- Independently by Craig & Landin and Hagersten
- Linked list
- Single Sentinel node
- Spin on locked flag of next node

```
Node* tail = new Node();
tail->locked = false; // unlocked
thread_local Node* node;

void lock() {
  node = new Node();
  node->locked = true;
  node->next = exchange(&tail,node);
  while (node->next->locked) {}
}

void unlock() {
  delete node->next;
  node->locked = false;
}
```

CLH Queue Lock

- Independently by Craig & Landin and Hagersten
- Linked list
- Single Sentinel node
- Spin on locked flag of next node

```
Node* tail = new Node();
tail->locked = false; // unlocked
thread_local Node* node;

void lock() {
  node = new Node();
  node->locked = true;
  node->next = exchange(&tail,node);
  while (node->next->locked) {}
}

void unlock() {
  delete node->next;
  node->locked = false;
}
```

CLH Queue Lock

- Independently by Craig & Landin and Hagersten
- Linked list
- Single Sentinel node
- Spin on locked flag of next node

```
Node* tail = new Node();
tail->locked = false; // unlocked
thread_local Node* node;

void lock() {
  node = new Node();
  node->locked = true;
  node->next = exchange(&tail,node);
  while (node->next->locked) {}
}

void unlock() {
  delete node->next;
  node->locked = false;
}
```

CLH Queue Lock

- Independently by Craig & Landin and Hagersten
- Linked list
- Single Sentinel node
- Spin on locked flag of next node

```
Node* tail = new Node();
tail->locked = false; // unlocked
thread_local Node* node;

void lock() {
  node = new Node();
  node->locked = true;
  node->next = exchange(&tail,node);
  while (node->next->locked) {}
}

void unlock() {
  delete node->next;
  node->locked = false;
}
```

CLH Queue Lock

- Independently by Craig & Landin and Hagersten
- Linked list
- Single Sentinel node
- Spin on locked flag of next node

```
Node* tail = new Node();
tail->locked = false; // unlocked
thread_local Node* node;

void lock() {
  node = new Node();
  node->locked = true;
  node->next = exchange(&tail,node);
  while (node->next->locked) {}
}

void unlock() {
  delete node->next;
  node->locked = false;
}
```

CLH Queue Lock

- Independently by Craig & Landin and Hagersten
- Linked list
- Single Sentinel node
- Spin on locked flag of next node

```
Node* tail = new Node();
tail->locked = false; // unlocked
thread_local Node* node;

void lock() {
  node = new Node();
  node->locked = true;
  node->next = exchange(&tail,node);
  while (node->next->locked) {}
}

void unlock() {
  delete node->next;
  node->locked = false;
}
```

CLH Queue Lock

- Independently by Craig & Landin and Hagersten
- Linked list
- Single Sentinel node
- Spin on locked flag of next node

```
Node* tail = new Node();
tail->locked = false; // unlocked
thread_local Node* node;

void lock() {
  node = new Node();
  node->locked = true;
  node->next = exchange(&tail,node);
  while (node->next->locked) {}
}

void unlock() {
  delete node->next;
  node->locked = false;
}
```

CLH Queue Lock: Implementation notes

- C++ only supports static thread_local variables
  - Either pass on some data on lock and unlock

    (if allowed by interface)
  - Or implement thread-local object storage yourself

    (Or use `boost::thread_specific_ptr`)

- Implementation (and field names) here differs slightly from Herlihy/Shavit
  - Next field stored in node
  - Manual memory management
  - Next can safely be deleted at unlock

    (no other thread accessing it)

CLH Queue Lock: Properties

- First-come, first-served, linearization by `exchange`
- Unlock wait-free $O(1)$
- $O(L)$ space, where L is the number of threads currently accessing the lock
  - More space depending on implementation of thread_local data
  - Herlihy/Shavit implementation requires $O(n)$
- Each thread spins on a separate location
  - Allocated locally by each thread, reduces false sharing
- Potential problem on NUMA architectures:
  - Locked field is in remote location
- Not fault-tolerant

MCS Queue Lock

- Maintain head instead of tail in list
- Spin on field in own node
- To unlock, modify locked field of next node
- If no successor exists, reset head

```
Node* head = nullptr;
thread_local Node* node = new Node();
node->next = nullptr;

void lock() {
  Node* n = node;
  Node* pred = exchange(&head,n);
  if (pred != nullptr) {
    n->locked = true;
    pred->next = n;
    while (n->locked);
  }
}

void unlock() {
  Node* n = node;
  if (n->next == nullptr) {
    if (compare_exchange(&head,n,
                         nullptr))
      return;
    // Wait for next thread
    n = node; // Be careful with C/C++
    while (n->next == nullptr);
  }
  n->next->locked = false;
  n->next = nullptr;
}
```

MCS Queue Lock

- Maintain head instead of tail in list
- Spin on field in own node
- To unlock, modify locked field of next node
- If no successor exists, reset head

```
Node* head = nullptr;
thread_local Node* node = new Node();
node->next = nullptr;

void lock() {
  Node* n = node;
  Node* pred = exchange(&head,n);
  if (pred != nullptr) {
    n->locked = true;
    pred->next = n;
    while (n->locked);
  }
}

void unlock() {
  Node* n = node;
  if (n->next == nullptr) {
    if (compare_exchange(&head,n,
                         nullptr))
      return;
    // Wait for next thread
    n = node; // Be careful with C/C++
    while (n->next == nullptr);
  }
  n->next->locked = false;
  n->next = nullptr;
}
```

MCS Queue Lock

- Maintain head instead of tail in list
- Spin on field in own node
- To unlock, modify locked field of next node
- If no successor exists, reset head

```
Node* head = nullptr;
thread_local Node* node = new Node();
node->next = nullptr;

void lock() {
  Node* n = node;
  Node* pred = exchange(&head,n);
  if (pred != nullptr) {
    n->locked = true;
    pred->next = n;
    while (n->locked);
  }
}

void unlock() {
  Node* n = node;
  if (n->next == nullptr) {
    if (compare_exchange(&head,n,
                         nullptr))
      return;
    // Wait for next thread
    n = node; // Be careful with C/C++
    while (n->next == nullptr);
  }
  n->next->locked = false;
  n->next = nullptr;
}
```

MCS Queue Lock

- Maintain head instead of tail in list
- Spin on field in own node
- To unlock, modify locked field of next node
- If no successor exists, reset head

```
Node* head = nullptr;
thread_local Node* node = new Node();
node->next = nullptr;

void lock() {
  Node* n = node;
  Node* pred = exchange(&head,n);
  if (pred != nullptr) {
    n->locked = true;
    pred->next = n;
    while (n->locked);
  }
}

void unlock() {
  Node* n = node;
  if (n->next == nullptr) {
    if (compare_exchange(&head,n,
                         nullptr))
      return;
    // Wait for next thread
    n = node; // Be careful with C/C++
    while (n->next == nullptr);
  }
  n->next->locked = false;
  n->next = nullptr;
}
```

MCS Queue Lock

- Maintain head instead of tail in list
- Spin on field in own node
- To unlock, modify locked field of next node
- If no successor exists, reset head

```
Node* head = nullptr;
thread_local Node* node = new Node();
node->next = nullptr;

void lock() {
  Node* n = node;
  Node* pred = exchange(&head,n);
  if (pred != nullptr) {
    n->locked = true;
    pred->next = n;
    while (n->locked);
  }
}

void unlock() {
  Node* n = node;
  if (n->next == nullptr) {
    if (compare_exchange(&head,n,
                         nullptr))
      return;
    // Wait for next thread
    n = node; // Be careful with C/C++
    while (n->next == nullptr);
  }
  n->next->locked = false;
  n->next = nullptr;
}
```

MCS Queue Lock

- Maintain head instead of tail in list
- Spin on field in own node
- To unlock, modify locked field of next node
- If no successor exists, reset head

```
Node* head = nullptr;
thread_local Node* node = new Node();
node->next = nullptr;

void lock() {
  Node* n = node;
  Node* pred = exchange(&head,n);
  if (pred != nullptr) {
    n->locked = true;
    pred->next = n;
    while (n->locked);
  }
}

void unlock() {
  Node* n = node;
  if (n->next == nullptr) {
    if (compare_exchange(&head,n,
                         nullptr))
      return;
    // Wait for next thread
    n = node; // Be careful with C/C++
    while (n->next == nullptr);
  }
  n->next->locked = false;
  n->next = nullptr;
}
```

MCS Queue Lock

- Maintain head instead of tail in list
- Spin on field in own node
- To unlock, modify locked field of next node
- If no successor exists, reset head

```
Node* head = nullptr;
thread_local Node* node = new Node();
node->next = nullptr;

void lock() {
  Node* n = node;
  Node* pred = exchange(&head,n);
  if (pred != nullptr) {
    n->locked = true;
    pred->next = n;
    while (n->locked);
  }
}

void unlock() {
  Node* n = node;
  if (n->next == nullptr) {
    if (compare_exchange(&head,n,
                         nullptr))
      return;
    // Wait for next thread
    n = node; // Be careful with C/C++
    while (n->next == nullptr);
  }
  n->next->locked = false;
  n->next = nullptr;
}
```

MCS Queue Lock: Properties

- First-come, first-served
- $O(n)$ space total
  - More space depending on implementation of thread_local data
- Each thread spins on its own memory location
  - Updated by other thread
- No additional memory management

- Requires compare-and-swap
  (consensus number ∞)
- Unlock not wait-free any more!
  (Waiting for next lock owner to set `next` pointer)
- Not fault-tolerant

Space Requirements in Array and Queue Locks

- These locks are starvation free (first-come, first-served fair)
- Array and queue locks all need $\Omega(n)$ space (thread local nodes, slot in array); shown by Hendler/Fich/Shavit
- In this sense, higher consensus operations do not improve on the simple register locks

Queue Locks with timeouts (trylock)

- Abandoning is easy for Test-And-Set lock
  - Just stop trying to acquire lock
  - Timing out is wait-free
- More difficult for queue locks
  - If we just exit, the following thread will starve
  - Can't just unlink the node

    (other thread might be accessing it)
- Lazy approach
  - Mark node as abandoned
  - Successor is responsible for cleanup

Queue Locks with timeouts (here: CLH Queue Lock)

- Add flag **abandoned** to nodes
- To abandon, set flag to **true**
- Next node is responsible for clean-up

```
Node* tail = new Node();
thread_local Node* node;

bool try_lock(int timeout) {
  node = new Node();
  node->locked = true;
  node->abandoned = false;
  node->next = exchange(&tail,node);
  int start = time();
  while (node->next->locked) {
    if (node->next->abandoned) {
      Node* pp = node->next->next;
      delete node->next;
      node->next = pp;
    }
    if (start + timeout <= time() {
      node->abandoned = true;
      return false;
    }
  }
  return true;
}

void unlock() {
  delete node->next;
  node->locked = false;
}
```

Queue Locks with timeouts (here: CLH Queue Lock)

- Add flag **abandoned** to nodes
- To abandon, set flag to **true**
- Next node is responsible for clean-up

```
Node* tail = new Node();
thread_local Node* node;

bool try_lock(int timeout) {
  node = new Node();
  node->locked = true;
  node->abandoned = false;
  node->next = exchange(&tail,node);
  int start = time();
  while (node->next->locked) {
    if (node->next->abandoned) {
      Node* pp = node->next->next;
      delete node->next;
      node->next = pp;
    }
    if (start + timeout <= time() {
      node->abandoned = true;
      return false;
    }
  }
  return true;
}

void unlock() {
  delete node->next;
  node->locked = false;
}
```

Queue Locks with timeouts (here: CLH Queue Lock)

- Add flag **abandoned** to nodes
- To abandon, set flag to **true**
- Next node is responsible for clean-up

```
Node* tail = new Node();
thread_local Node* node;

bool try_lock(int timeout) {
  node = new Node();
  node->locked = true;
  node->abandoned = false;
  node->next = exchange(&tail,node);
  int start = time();
  while (node->next->locked) {
    if (node->next->abandoned) {
      Node* pp = node->next->next;
      delete node->next;
      node->next = pp;
    }
    if (start + timeout <= time() {
      node->abandoned = true;
      return false;
    }
  }
  return true;
}

void unlock() {
  delete node->next;
  node->locked = false;
}
```

Queue Locks with timeouts (here: CLH Queue Lock)

- Add flag **abandoned** to nodes
- To abandon, set flag to **true**
- Next node is responsible for clean-up

```
Node* tail = new Node();
thread_local Node* node;

bool try_lock(int timeout) {
  node = new Node();
  node->locked = true;
  node->abandoned = false;
  node->next = exchange(&tail,node);
  int start = time();
  while (node->next->locked) {
    if (node->next->abandoned) {
      Node* pp = node->next->next;
      delete node->next;
      node->next = pp;
    }
    if (start + timeout <= time() {
      node->abandoned = true;
      return false;
    }
  }
  return true;
}

void unlock() {
  delete node->next;
  node->locked = false;
}
```

Queue Locks with timeouts (here: CLH Queue Lock)

- Add flag **abandoned** to nodes
- To abandon, set flag to **true**
- Next node is responsible for clean-up

```
Node* tail = new Node();
thread_local Node* node;

bool try_lock(int timeout) {
  node = new Node();
  node->locked = true;
  node->abandoned = false;
  node->next = exchange(&tail,node);
  int start = time();
  while (node->next->locked) {
    if (node->next->abandoned) {
      Node* pp = node->next->next;
      delete node->next;
      node->next = pp;
    }
    if (start + timeout <= time() {
      node->abandoned = true;
      return false;
    }
  }
  return true;
}

void unlock() {
  delete node->next;
  node->locked = false;
}
```

Queue Locks with timeouts (here: CLH Queue Lock)

- Add flag **abandoned** to nodes
- To abandon, set flag to **true**
- Next node is responsible for clean-up

```
Node* tail = new Node();
thread_local Node* node;

bool try_lock(int timeout) {
  node = new Node();
  node->locked = true;
  node->abandoned = false;
  node->next = exchange(&tail,node);
  int start = time();
  while (node->next->locked) {
    if (node->next->abandoned) {
      Node* pp = node->next->next;
      delete node->next;
      node->next = pp;
    }
    if (start + timeout <= time() {
      node->abandoned = true;
      return false;
    }
  }
  return true;
}

void unlock() {
  delete node->next;
  node->locked = false;
}
```

Queue Locks with timeouts (here: CLH Queue Lock)

- Add flag **abandoned** to nodes
- To abandon, set flag to **true**
- Next node is responsible for clean-up

```
Node* tail = new Node();
thread_local Node* node;

bool try_lock(int timeout) {
  node = new Node();
  node->locked = true;
  node->abandoned = false;
  node->next = exchange(&tail,node);
  int start = time();
  while (node->next->locked) {
    if (node->next->abandoned) {
      Node* pp = node->next->next;
      delete node->next;
      node->next = pp;
    }
    if (start + timeout <= time() {
      node->abandoned = true;
      return false;
    }
  }
  return true;
}

void unlock() {
  delete node->next;
  node->locked = false;
}
```

Queue Locks with timeouts (here: CLH Queue Lock)

- Add flag **abandoned** to nodes
- To abandon, set flag to **true**
- Next node is responsible for clean-up

```
Node* tail = new Node();
thread_local Node* node;

bool try_lock(int timeout) {
  node = new Node();
  node->locked = true;
  node->abandoned = false;
  node->next = exchange(&tail,node);
  int start = time();
  while (node->next->locked) {
    if (node->next->abandoned) {
      Node* pp = node->next->next;
      delete node->next;
      node->next = pp;
    }
    if (start + timeout <= time() {
      node->abandoned = true;
      return false;
    }
  }
  return true;
}

void unlock() {
  delete node->next;
  node->locked = false;
}
```

Composite Lock

- Combines Backoff lock and Queue lock
- Preallocate fixed number of nodes $< n$
- Acquire a random node
  - Only one thread may use a certain node
  - On failure back off
- As soon as a node is acquired, enqueue it
- Can be augmented with a fast path for low contention
  - If queue is empty, try fast path, on failure use normal path

```
Node nodes[k]; // preallocated

void lock() {
  Node* node;
  while(!(node =
      acquireNode(rnd() % k)))
    backoff();

  enqueueNode(node);
  waitForPredecessor(node);
}
```

Composite Lock

- Combines Backoff lock and Queue lock
- Preallocate fixed number of nodes $< n$
- Acquire a random node
  - Only one thread may use a certain node
  - On failure back off
- As soon as a node is acquired, enqueue it
- Can be augmented with a fast path for low contention
  - If queue is empty, try fast path, on failure use normal path

```
Node nodes[k]; // preallocated

void lock() {
  Node* node;
  while(!(node =
      acquireNode(rnd() % k)))
    backoff();

  enqueueNode(node);
  waitForPredecessor(node);
}
```

Composite Lock

- Combines Backoff lock and Queue lock
- Preallocate fixed number of nodes $< n$
- Acquire a random node
  - Only one thread may use a certain node
  - On failure back off
- As soon as a node is acquired, enqueue it
- Can be augmented with a fast path for low contention
  - If queue is empty, try fast path, on failure use normal path

```
Node nodes[k]; // preallocated

void lock() {
  Node* node;
  while(!(node =
      acquireNode(rnd() % k)))
    backoff();

  enqueueNode(node);
  waitForPredecessor(node);
}
```

Composite Lock

- Combines Backoff lock and Queue lock
- Preallocate fixed number of nodes $< n$
- Acquire a random node
  - Only one thread may use a certain node
  - On failure back off
- As soon as a node is acquired, enqueue it
- Can be augmented with a fast path for low contention
  - If queue is empty, try fast path, on failure use normal path

```
Node nodes[k]; // preallocated

void lock() {
  Node* node;
  while(!(node =
      acquireNode(rnd() % k)))
    backoff();

  enqueueNode(node);
  waitForPredecessor(node);
}
```

Composite Lock

- Combines Backoff lock and Queue lock
- Preallocate fixed number of nodes $< n$
- Acquire a random node
  - Only one thread may use a certain node
  - On failure back off
- As soon as a node is acquired, enqueue it
- Can be augmented with a fast path for low contention
  - If queue is empty, try fast path, on failure use normal path

```
Node nodes[k]; // preallocated

void lock() {
  Node* node;
  while(!(node =
      acquireNode(rnd() % k)))
    backoff();

  enqueueNode(node);
  waitForPredecessor(node);
}
```

Composite Lock

- Combines Backoff lock and Queue lock
- Preallocate fixed number of nodes $< n$
- Acquire a random node
  - Only one thread may use a certain node
  - On failure back off
- As soon as a node is acquired, enqueue it
- Can be augmented with a fast path for low contention
  - If queue is empty, try fast path, on failure use normal path

```
Node nodes[k]; // preallocated

void lock() {
  Node* node;
  while(!(node =
      acquireNode(rnd() % k)))
    backoff();

  enqueueNode(node);
  waitForPredecessor(node);
}
```

Composite Lock

- Combines Backoff lock and Queue lock
- Preallocate fixed number of nodes $< n$
- Acquire a random node
  - Only one thread may use a certain node
  - On failure back off
- As soon as a node is acquired, enqueue it
- Can be augmented with a fast path for low contention
  - If queue is empty, try fast path, on failure use normal path

```
Node nodes[k]; // preallocated

void lock() {
  Node* node;
  while(!(node =
      acquireNode(rnd() % k)))
    backoff();

  enqueueNode(node);
  waitForPredecessor(node);
}
```

Composite Lock

- Combines Backoff lock and Queue lock
- Preallocate fixed number of nodes $< n$
- Acquire a random node
  - Only one thread may use a certain node
  - On failure back off
- As soon as a node is acquired, enqueue it
- Can be augmented with a fast path for low contention
  - If queue is empty, try fast path, on failure use normal path

```
Node nodes[k]; // preallocated

void lock() {
  Node* node;
  while(!(node =
      acquireNode(rnd() % k)))
    backoff();

  enqueueNode(node);
  waitForPredecessor(node);
}
```

Composite Lock

- Combines Backoff lock and Queue lock
- Preallocate fixed number of nodes $< n$
- Acquire a random node
  - Only one thread may use a certain node
  - On failure back off
- As soon as a node is acquired, enqueue it
- Can be augmented with a fast path for low contention
  - If queue is empty, try fast path, on failure use normal path

```
Node nodes[k]; // preallocated

void lock() {
  Node* node;
  while(!(node =
      acquireNode(rnd() % k)))
    backoff();

  enqueueNode(node);
  waitForPredecessor(node);
}
```

Taking the memory hierarchy into account

- Most modern architectures are NUMA architectures
- Some processors are near to each other
  - → Smaller memory access times
- Hierarchical locks minimize lock migration[1]
- Look at the architecture of your own machine
  - `lstopo` tool (part of hwloc)
  - Try it on mars/saturn and ceres

[1] A lock L migrates if two threads running on a different NUMA clusters (nodes) acquire L one after the other - from Dice et al., PPoPP 2012

Hierarchical Backoff Lock

- Original idea by Zoran Radovic and Erik Hagersten
- Based on Backoff Lock
- Length of backoff dependent on distance to lock owner

  (2-level hierarchy common)
- Local threads are more likely to acquire lock
- May starve remote threads!

```
int noowner = 1;
int owner = noowner;

void lock() {
  int current_owner;
  do {
    current_owner = noowner;
    if (compare_exchange(
          &owner,&current_owner,
                thread_id()))
    break;

    /* C11 convention:
       Failed compare_exchange loads
       old value of owner into
       current_owner
    */

    int distance =
      memory_distance(current_owner,
                      thread_id());
    backoff(distance);
  } while (1);
}
```

Hierarchical CLH Lock

- By Victor Luchangco, Dan Nussbaum and Nir Shavit

- Multiple clusters of processors
- 1 global queue
- 1 local queue per cluster

- Add node to local queue
- First node in local queue is the cluster master and splices local queue into global queue
- Others wait until they have the lock or become cluster master

```
thread_local Node* pred, node;
// cluster local queues
cluster_local Node* lq;
Node* gq; // global queue

void lock()
{
  // Splice into local queue
  pred = exchange(&lq,node);

  if (pred != nullptr) {
    bool lock =
      wait_lock_or_master(pred);
    if (lock) // Lock acquired
      return;
  }

  // Thread is now cluster master

  // Splice local into global queue
  tail = *lq;
  pred = exchange(&gq,tail);
  node->pred = pred;

  // Successor is new cluster master
  tail->newmaster = true;

  // Wait for lock
  while (pred->locked);
}
```

Hierarchical CLH Lock

Hierarchical CLH Lock

Hierarchical CLH Lock

Hierarchical CLH Lock

Hierarchical CLH Lock

Hierarchical CLH Lock

Hierarchical CLH Lock

Hierarchical CLH Lock

Lock Cohorting

- Introduced in 2012 by David Dice, Virendra J. Marathe and Nir Shavit
- General technique to construct NUMA-aware locks from normal locks
- 1 global lock (not NUMA aware)
- 1 local lock per cluster
  - Requires cohort detection ability

    (Other threads from same cluster wait for global lock)
- Threads must acquire both locks
  - Global lock can be passed around inside NUMA cluster
  - Release global lock at some point for fairness
- Supports abortable locks

Final remarks

- There is no one lock for every application
- Choice depends on congestion in application
- NUMA-awareness improves performance on modern systems

- Reader-Writer Locks not discussed in this lecture
  - See Herlihy-Shavit book
  - A recent result: NUMA-aware reader-writer locks by Calciu et al., PPoPP 2013

Pthreads readers-writers locks

Many readers, at most one writer in critical section.

```
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                        const pthread_rwlockattr_t *restrict attr);

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

Lock summary

- Test-And-Set Locks
  - + Low overhead
  - + Abortable by design
  - - Unfair
  - - All threads spin on same memory location
  - - Threads may wait longer than necessary with backoff
- Queue Locks
  - + Fair
  - + Threads spin on different locations
  - - Hard to abort
  - - Cleanup often difficult without garbage collection
  - - Need $\Omega(n)$ space
- Composite Lock
  - + Tries to take advantages of both TAS locks and Queue locks
  - + Good at high congestion
  - - High overhead on low congestion
  - - Quite complex
- Hierarchical Locks
  - + Higher performance on NUMA systems due to less lock migrations
  - + Cohorting allows to use all lock types for NUMA-aware locks
  - - Added complexity

References

- John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst., 9(1): 21–65, 1991.
- Travis S. Craig. Queuing spin lock algorithms to support timing predictability. In Proceedings of the Real-Time Systems Symposium, pages 148–157, 1993.
- Peter S. Magnusson, Anders Landin, and Erik Hagersten. Queue locks on cache coherent multiprocessors. In Proceedings of the 8th International Symposium on Parallel Processing, pages 165–171, 1994.
- David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: a general technique for designing NUMA locks. In Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), pages 247–256, 2012.
- Faith Ellen, Danny Hendler, Nir Shavit On the Inherent Sequentiality of Concurrent Objects. SIAM J. Comput., 41(3): 519–536, 2012.

Advanced Multiprocessor Programming: Data-Structures Part 1

Martin Wimmer, Jesper Larsson Träff

TU Wien

April 24th, 2023

Concurrent data structures, Part I

A first data structure: List-based set

General issues and concerns:

- Coarse-grained vs. fine-grained synchronization
- Safety and liveness properties
- Progress guarantees
- Data structure: Algorithms and complexity

Coarse-grained synchronization

The trivial solution: Use lock to protect all data structure operations

- Acquire lock when data structure is accessed (especially updated): Each method call acquires and releases
- Likely major scalability bottleneck!
  - Only one thread can access data-structure at any time
    - → Even for read accesses!
  - No progress guarantees
    - → Thread holding a lock may block other threads indefinitely
  - But linearizable (when lock is acquired)

Can we do better?

Fine-grained synchronization

- Decouple parts of the data-structure
  - $\rightarrow$ multiple locks
- Get rid of locks altogether
  - $\rightarrow$ Use higher-consensus operations instead (atomics)
- Provide progress guarantees

Liveness properties

- Deadlock-free
- Starvation-free
- Non-blocking
  - No thread can block other threads

    (Threads not actively participating at the moment shall not hinder other threads from progressing)
  - All threads may require an unbounded number of steps
- Lock-free
  - At least one thread progresses in a bounded number of steps
- Wait-free
  - All threads progress in a bounded number of steps

Correctness condition: Linearizability

- Operation appears to take effect as a single, atomic step somewhere during the execution
- Operations on data-structure appear as atomic operations to the outside
  - (other, concurrent method calls)

- Tricky to achieve
  - → The combination of two linearizable operations is not necessarily linearizable (an insert combined with a delete will not be atomic to the outside observer)

Proving linearizability

For each implemented operation:

- Look at the outcome (return value)
- Show that there is some instant (a linearization point) during the execution of the operation where this outcome is correct in the linearized history (sequence of linearization points)
- The linerarization point is sometimes a specific operation, but sometimes depends on the history (concurrent execution of other operations)

The data structure: List-based set

Set data-structure operations:

- `bool add(item)`
  - Add item to the set (if not already in set)
  - Return `true` if item was added, `false` if already existing
- `bool remove(item)`
  - Remove item from set
  - Return `true` on success, `false` if item was not found
- `bool contains(item)`
  - Check whether item exists in set

Implementation as ordered, linked list, items identified by key drawn from an ordered universe. Note: In code snippets, we do not distinguish between `item` and `key`

In terms of sequential complexity, not a good data structure, all operations are $O(n)$ and some require $\Omega(n)$

List-based set: Data structure

- Sorted, linked list of node elements
- Elements identified by unique key from ordered universe
- Items stored in nodes are comparable by key
- Sentinel nodes for head and tail
- Head and tail contain maximum and minimum key values ($-\infty$ and ∞)

List-based set: Assumptions and Invariants

- Assumptions
  - Freedom from interference

    (only operations `add`, `remove` and `contains` may modify nodes)
  - Garbage collection available

- Invariants: Properties that hold initially and are maintained by any (concurrent) operation; no thread can take a step that makes an invariant false.

  - Sentinel nodes are neither added nor removed
  - Nodes are sorted by item (key)
  - Items are unique
  - Item is in the set iff it is reachable from the head

List-based set with coarse-grained locking

- Single, global lock
- Acquired at beginning of each method call
- Released at the end
- Similar to a sequential implementation

```
Node* head;
mutex m;

bool add(T item) {
  // C++11 style locking
  lock_guard<mutex> g(m);

  // Search for item or successor
  Node* pred = head;
  Node* curr = pred->next;
  while (curr->item < item) {
    pred = curr;
    curr = curr->next;
  }

  // Item alread in set
  if (item == curr->item) return false;

  // Add item to set
  Node* n = new Node(item, curr);
  pred->next = n;

  return true; // Done
} // end of lock scope
```

List-based set with coarse-grained locking: Properties

- Obviously correct
- Execution is essentially sequential

- Linearization point
  - The instant the lock is acquired

- Liveness
  - Same as the lock implementation

- No progress guarantees!

  (Thread holding the lock may stall, progress of all other threads blocked)

Relaxation: Reader-Writer Lock

- Use Reader-Writer lock instead of a normal lock
- Allows concurrent read accesses
- Better performance if `contains` is the most common operation
  - → Typically the case

```
Node* head;
rw_mutex m;

bool contains(T item) {
  // Pseudocode
  lock_guard<mutex> g(m.read_lock());

  // Search for item
  Node* curr = head;
  while (curr->item < item) {
    curr = curr->next;
  }

  if (item == curr->item)
    return true; // element in set
  else
    return false; // not in set
} // end of lock scope
```

Reader-Writer Lock: Properties

- Multiple `contains` checks can be performed at the same time
- Linearization point
  - The instant the lock is acquired
- Liveness
  - Same as the lock implementation
  - Still no progress guarantees!

    (A single writer may stall all other threads)
  - Special case: No writers → wait-free

    (if reader-lock implementation is wait-free)

List-based set with fine-grained locking

- Split object into independently synchronized components
- Method calls only interfere when trying to access the same component at the same time

→ Instead of a global lock, use a lock per node

List-based set with fine-grained locks: `add(16)`

- Locks acquired in hand-over-hand manner
- New item inserted between locks

List-based set with fine-grained locks: `add(16)`

- Locks acquired in hand-over-hand manner
- New item inserted between locks

List-based set with fine-grained locks: `add(16)`

- Locks acquired in hand-over-hand manner
- New item inserted between locks

List-based set with fine-grained locks: `add(16)`

- Locks acquired in hand-over-hand manner
- New item inserted between locks

List-based set with fine-grained locks: `add(16)`

- Locks acquired in hand-over-hand manner
- New item inserted between locks

List-based set with fine-grained locks: `add(16)`

- Locks acquired in hand-over-hand manner
- New item inserted between locks

List-based set with fine-grained locks: `add(16)`

- Locks acquired in hand-over-hand manner
- New item inserted between locks

List-based set with fine-grained locks: `add(16)`

- Locks acquired in hand-over-hand manner
- New item inserted between locks

List-based set with fine-grained locks: `add(16)`

- Locks acquired in hand-over-hand manner
- New item inserted between locks

List-based set with fine-grained locks: `add(16)`

- Locks acquired in hand-over-hand manner
- New item inserted between locks

List-based set with fine-grained locks: `add(16)`

- Locks acquired in hand-over-hand manner
- New item inserted between locks

List-based set with fine-grained locks: `remove(16)` call

- Locks acquired in hand-over-hand manner
- Item removed after locking

List-based set with fine-grained locks: `remove(16)` call

- Locks acquired in hand-over-hand manner
- Item removed after locking

List-based set with fine-grained locks: `remove(16)` call

- Locks acquired in hand-over-hand manner
- Item removed after locking

List-based set with fine-grained locks: `remove(16)` call

- Locks acquired in hand-over-hand manner
- Item removed after locking

List-based set with fine-grained locks: `remove(16)` call

- Locks acquired in hand-over-hand manner
- Item removed after locking

List-based set with fine-grained locks: `remove(16)` call

- Locks acquired in hand-over-hand manner
- Item removed after locking

List-based set with fine-grained locks: `remove(16)` call

- Locks acquired in hand-over-hand manner
- Item removed after locking

List-based set with fine-grained locks: `remove(16)` call

- Locks acquired in hand-over-hand manner
- Item removed after locking

List-based set with fine-grained locks: `remove(16)` call

- Locks acquired in hand-over-hand manner
- Item removed after locking

List-based set with fine-grained locks: `remove(16)` call

- Locks acquired in hand-over-hand manner
- Item removed after locking

List-based set with fine-grained locks: `remove(16)` call

- Locks acquired in hand-over-hand manner
- Item removed after locking

List-based set with fine-grained locks: `remove(16)` call

- Locks acquired in hand-over-hand manner
- Item removed after locking

List-based set with fine-grained locks

```
Node* head;
struct Window {
  Node* pred;
  Node* curr;
}

Window find(T item) {
  // Search for item or successor
  Node* pred = head;
  pred->lock();

  // Exception safety for locks ignored
       for simplicity
  Node* curr = pred->next;
  curr->lock();

  while (curr->item < item) {
    pred->unlock();
    pred = curr;
    curr = curr->next;
    curr->lock();
  }

  return Window(pred, curr);
}

void unlock(Window w) {
  pred->unlock();
  curr->unlock();
}
```

```
bool contains(T item) {
  Window w = find(item);
  bool found = item == w.curr->item;
  unlock(w);
  return found;
}

bool add(T item) {
  Window w = find(item);
  if (item == w.curr->item) {
    unlock(w);
    return false;
  }
  Node* n = new Node(item, w.curr);
  w.pred->next = n;
  unlock(w);
  return true;
}

bool remove(T item) {
  Window w = find(item);
  if (item != w.curr->item) {
    unlock(w);
    return false;
  }
  w.pred->next = w.curr->next;
  unlock(w);
  delete(w.curr); // safe to delete
  return true;
}
```

List-based set with fine-grained locking: Invariants

- Sentinels are never added or removed
- Nodes are sorted by item
- Each item occurs exactly once
- An item is in the set iff it is reachable from head

From now on: The `find` operations locate position in list such that `pred->item` $<$ `item` $\leqslant$ `curr->item`

List-based set with fine-grained locking: Liveness

- Locks are always acquired in the same order
  $\rightarrow$ deadlock-free
- If the locks are starvation-free, so is the set
- No progress guarantees
- Limited concurrency

  (A thread having lock to a small item early in the list blocks other threads looking for larger items)

List-based set with fine-grained locking: Linearization

Linearization points:

Outcome: Item in set

- Linearize when item containing element is locked
  (Cannot be removed while locked)

Outcome: Item not in set

- Linearize when node containing next-higher item is locked
  (Predecessor node cannot be inserted while an item is locked)

(Earliest possible linearization point: When lock on head is acquired? As soon as head lock has been acquired, thread cannot be overtaken)

List-based set with optimistic synchronization

Idea:

- Search for a component without acquiring locks
- Lock the found nodes
- Confirm whether the nodes are correct: Validate that they are still reachable, and that `pred` points to `curr`
- On failure, release locks and start over
- Requires good validation!

List-based set with optimistic synchronization: `add(16)`

- Search for item without locking

List-based set with optimistic synchronization: `add(16)`

- Search for item without locking

List-based set with optimistic synchronization: `add(16)`

- Search for item without locking

List-based set with optimistic synchronization: `add(16)`

- Search for item without locking

List-based set with optimistic synchronization: `add(16)`

- Search for item without locking

List-based set with optimistic synchronization: `add(16)`

- Search for item without locking
- Lock item

List-based set with optimistic synchronization: `add(16)`

- Search for item without locking
- Lock item

List-based set with optimistic synchronization: `add(16)`

- Search for item without locking
- Lock item
- Validate by searching for item again

List-based set with optimistic synchronization: `add(16)`

- Search for item without locking
- Lock item
- Validate by searching for item again

List-based set with optimistic synchronization: `add(16)`

- Search for item without locking
- Lock item
- Validate by searching for item again

List-based set with optimistic synchronization: `add(16)`

- Search for item without locking
- Lock item
- Validate by searching for item again

List-based set with optimistic synchronization: `add(16)`

- Search for item without locking
- Lock item
- Validate by searching for item again
- Add item

List-based set with optimistic synchronization: `add(16)`

- Search for item without locking
- Lock item
- Validate by searching for item again
- Add item
- Unlock

List-based set with optimistic synchronization: `add(16)`

- Search for item without locking
- Lock item
- Validate by searching for item again
- Add item
- Unlock

List-based set with optimistic synchronization

```
Node* head;
struct Window {
  Node* pred;
  Node* curr;
}

void lock(Window w) {
  pred->lock();
  curr->lock();
}

void unlock(Window w) {
  pred->unlock();
  curr->unlock();
}

bool validate(Window w) {
  Node* n = head;
  while (n->item <= w.pred->item) {
    if (n == w.pred)
      return n->next == w.curr;

    n = n->next;
  }
  return false;
}
```

```
Window find(T item) {
  while (true) {
    // Search for item or successor
    Node* pred = head;
    Node* curr = pred->next;

    while (curr->item < item) {
      pred = curr;
      curr = curr->next;
    }

    Window w(pred, curr);
    lock(w);
    if (validate(w)) return w;
    unlock(w);
  }
}
```

List-based set with optimistic synchronization

```
bool contains(T item) {
  // Identical to fine-grained locking:

  Window w = find(item);
  bool found = item == w.curr->item;
  unlock(w);
  return found;
}

bool add(T item) {
  // identical to fine-grained locking

  Window w = find(item);
  if (item == w.curr->item) {
    unlock(w);
    return false;
  }
  Node* n = new Node(item, w.curr);
  w.pred->next = n;
  unlock(w);
  return true;
}
```

```
bool remove(T item) {
  Window w = find(item);
  if (item != w.curr->item) {
    unlock(w);
    return false;
  }
  w.pred->next = w.curr->next;
  unlock(w);

  // Deletion of nodes would be unsafe!
  // delete(w.curr);

  return true;
}
```

List-based set with optimistic synchronization: Properties

- Nodes may be traversed even though they have been removed
- Rely on garbage collection for removed nodes
  - Manual memory reclamation is difficult

    (How can we know that no thread currently reads a node?)

- Previous invariants still apply
  - Sentinels are never added or removed
  - Nodes are sorted by the item
  - Each item occurs exactly once
  - An item is in the set iff it is reachable from head

    → But only if validated!

- Freedom of interference essential: `next` field only changed by operations on the list

List-based set with optimistic synchronization: Liveness

- Deadlock-free
  - Locks are always acquired in the same order
  - On failed validation both locks are freed
- Not starvation-free!
  - Validation might fail all the time as other threads add and remove nodes
- No progress guarantees!

  (But many threads can work concurrently on different items in list)
- Each validation traverses list from head!

List-based set with optimistic synchronization

Linearization points:

Outcome: Item in set

- Linearize when item containing element is successfully validated

Outcome: Item not in set

- Linearize when node containing next-higher item is validated

List-based set with lazy synchronization

Disadvantages of optimistic synchronization (aka: Speculation):

- Need to acquire locks for `contains` call

  (`contains` calls are typically more common than `add` or `remove` calls)
- Even though `contains` does not modify the list
- List is traversed twice, so computational overhead is doubled

List-based set with lazy synchronization

Idea: Be lazy and postpone some hard operations, or get help from other threads

- Split `remove` operation into two parts
  - Logical removal
    - Introduce additional marked flag for each node
    - A node is in the set iff it is reachable and not marked
  - Physical removal
    - Redirect next pointer of predecessor node

List-based set with lazy synchronization: `remove(16)` call

- Search for item without locking

List-based set with lazy synchronization: `remove(16)` call

- Search for item without locking

List-based set with lazy synchronization: `remove(16)` call

- Search for item without locking

List-based set with lazy synchronization: `remove(16)` call

- Search for item without locking

List-based set with lazy synchronization: `remove(16)` call

- Search for item without locking

List-based set with lazy synchronization: `remove(16)` call

- Search for item without locking
- Lock item

List-based set with lazy synchronization: `remove(16)` call

- Search for item without locking
- Lock item

List-based set with lazy synchronization: `remove(16)` call

- Search for item without locking
- Lock item
- Verify by checking marked flags of items

List-based set with lazy synchronization: `remove(16)` call

- Search for item without locking
- Lock item
- Verify by checking marked flags of items
- Mark item as logically removed

List-based set with lazy synchronization: `remove(16)` call

- Search for item without locking
- Lock item
- Verify by checking marked flags of items
- Mark item as logically removed
- Unlink item

List-based set with lazy synchronization: `remove(16)` call

- Search for item without locking
- Lock item
- Verify by checking marked flags of items
- Mark item as logically removed
- Unlink item
- Unlock

List-based set with lazy synchronization: `remove(16)` call

- Search for item without locking
- Lock item
- Verify by checking marked flags of items
- Mark item as logically removed
- Unlink item
- Unlock

List-based set with lazy synchronization

```
Node* head;
struct Window {
  Node* pred;
  Node* curr;
}

Window find(T item) {
  while (true) {
    // Search for item or successor
    Node* pred = head;
    Node* curr = pred->next;

    while (curr->item < item) {
      pred = curr;
      curr = curr->next;
    }

    Window w(pred, curr);
    lock(w);
    if (validate(w)) return w;
    unlock(w);
  }
}

bool validate(Window w) {
  return !w.pred->marked &&
         !w.curr->marked &&
          w.pred->next == curr;
}
```

```
bool contains(T item) {
  Node* n = head;

  while (n->item < item) n = n->next;

  return n->item == item &&
         !n->marked;
}

bool add(T item) {
  // identical to fine-grained locking
}

bool remove(T item) {
  Window w = find(item);
  if (item != w.curr->item) {
    unlock(w);
    return false;
  }
  // logical remove (mark)
  w.curr->marked = true;

  // physical unlink
  w.pred->next = w.curr->next;
  // Careful with memory order!

  unlock(w);
  return true;
}
```

List-based set with lazy synchronization: Invariants

- Nodes may be traversed even though they have been removed
- We rely on garbage collection for removed nodes
- Sentinels are never added or removed
- Nodes are sorted by the item
- Each item occurs exactly once

- An item is in the set iff
  - it is reachable from head
  - it is not marked

List-based set with lazy synchronization: Liveness

- Deadlock-free
  - Locks are always acquired in the same order
  - On failed validation both locks are freed
  - No locks at all for `contains`
- `add` and `remove` not starvation-free!
  - Validation might fail all the time as other threads add and remove nodes
- Wait-free `contains` call
- Validation does not longer re-traverse list.

List-based set with lazy synchronization: Linearization

Linearization points:

Outcome: Item in set

`add`	Successful validation
`remove`	Flag `marked` set to `true`
`contains`	Read of `marked` flag

Outcome: Item not in set

`add`	Successful validation of next node
`remove`	Successful validation of next node
`contains`	Earlier of • Point where a removed matching node, or a node with `node->item` > `item` is found • Point immediatly before a new matching node is added to list

List-based set with lazy synchronization

Linearization of unsuccessful `contains(16)` call

- Search for item without locking

List-based set with lazy synchronization

Linearization of unsuccessful `contains(16)` call

- Search for item without locking

List-based set with lazy synchronization

Linearization of unsuccessful `contains(16)` call

- Search for item without locking

List-based set with lazy synchronization

Linearization of unsuccessful `contains(16)` call

- Search for item without locking

List-based set with lazy synchronization

Linearization of unsuccessful `contains(16)` call

- Search for item without locking

List-based set with lazy synchronization

Linearization of unsuccessful `contains(16)` call

- Search for item without locking
- Node is unlinked by another thread

List-based set with lazy synchronization

Linearization of unsuccessful `contains(16)` call

- Search for item without locking
- Node is unlinked by another thread
- Same value node is linked by another thread

List-based set with lazy synchronization

Linearization of unsuccessful `contains(16)` call

- Search for item without locking
- Node is unlinked by another thread
- Same value node is linked by another thread
- `contains` call must be linearized before the check of the marked flag!

List-based set with lazy synchronization: Linearization

Item not in list

`add`	Successful validation of next node
`remove`	Successful validation of next node
`contains`	Earlier of • Point where a removed matching node, or a node with `node->item` > `item` is found • Point immediatly before a new matching node is added to list

Linearization at the point where the marked flag is checked would be wrong, since this can be after a new node with same key is added. The point immediately before the new node is added is correct, since this `add` operation must be concurrent with the `contains` call (otherwise, `contains` would have found the item).

Lesson: Linearization point is not always some predetermined instruction!

Making the list-based set lock-free

Idea: Use `compare_exchange` to update `next`-pointer when adding and removing items.

Does not work!

- Case 1: Thread A removes first item, thread B adds new second item. Both successfully perform `compare_exchange`, but B's item is not added
- Case 2: Thread A removes first item, thread B removes second item. Both successfully perform `compare_exchange`, but only the first item is removed

Concurrent `remove(4)`, `remove(8)`

Making the list-based set lock-free

Idea: Use `compare_exchange` to update `next`-pointer when adding and removing items.

Repair:

- Treat `next`-pointer and `marked`-flag as an atomic unit.
- `compare_exchange` operation succeeds only if both pointer and flag have the expected value
- Marked nodes cannot be traversed, but must be linked out during traversal by `add` and `remove` operations

In absence of double-word compare-and-swap, steal a bit from the pointer and use that as flag. Addresses are normally less than full 64 bits (48 bits, 32 bits, ...)

Marked pointer implementation

Take some free (upper) bits from pointer (`next` field in `Node`); use bit operations to read out and modify flag.

```
void *getpointer(void *); // mask out pointer part
bool getflag(void *);     // return stolen bit from next pointer
void setflag(void **);    // set bit in next pointer
void resetflag(void **);  // reset bit in next pointer
```

Note: In code snippets masking out the pointer is mostly left out

Java: Use atomic marked reference (see next lectures)

Lock-free list-based set

```
Node* head;
struct Window {
  Node* pred;
  Node* curr;
}

bool contains(T item) {
  // same as lazy implementation
  // except marked flag is part of next
       pointer
  Node* n = head;

  while (n->item < item)
    n = getpointer(n->next);

  return n->item == item &&
         !getflag(n->next);
}
```

```
Window find(T item) {
  // Search for item or successor
retry:
  Node* pred = head;
  Node* curr = getpointer(pred->next);

  while (true) {
    Node *succ = getpointer(curr->next);
    while (getflag(curr->next)) {
      // link out marked item (curr)

      resetflag(&curr);
      resetflag(&succ);
      if (!compare_exchange(&pred->next,
                            &curr, succ))
        goto retry;

      curr = succ;
      succ = getpointer(succ->next);
    }

    if (curr->item >= item) {
      Window w(pred, curr);
      return w;
    }
    pred = curr;
    curr = getpointer(curr->next);
  }
}
```

Lock-free list-based set

```
bool add(T item) {
  Window w;

  Node *n = new Node(item);

  while (true) {
    w = find(item);
    Node *pred = w.pred;
    Node *curr = w.curr;

    if (curr->item == item) {
      delete(n); // was never in list
      return false;
    }

    n->next = curr;

    // unmark new node
    resetflag(&n->next);
    resetflag(&curr);

    if (compare_exchange(&pred->next,
                         &curr, n))
      return true;
  }
}
```

```
bool remove(T item) {
  Window w;
  while (true) {
    w = find(item);
    if (item != w.curr->item)
      return false;

    Node *succ = w.curr->next;
    Node *markedsucc = succ;
    // mark as deleted
    setflag(&markedsucc);
    resetflag(&succ);
    if (!compare_exchange(&w.curr->next,
                          &succ,
                          markedsucc))
      continue;

    // attempt to unlink curr
    compare_exchange(&w.pred->next,
                     &w.curr, succ);
    return true;
}
```

Lock-free list-based set: Invariants

- Nodes may be traversed even though they have been removed
- We rely on garbage collection for removed nodes
- Sentinels are never added or removed
- Nodes are sorted by the item
- Each item occurs exactly once
- An item is in the set iff it is reachable from head and not marked

Lock-free list-based set: Liveness

- `add` is lock-free
- `remove` is lock-free
- `contains` is wait-free

Argument: If a `compare_exchange` in `add` or `remove` fails, then some other thread has made progress.

- Each failure to mark/unlink or add item causes restart from head!

Lock-free list-based set: Linearization

Linearization points:

Outcome: Item in set

`add` Earlier of

- Point where found node is checked against `item`
- Point immediatly before node is marked (concurrent `remove`)

`remove` Compare-exchange on marked flag

`contains` Read of `marked` flag

Lock-free list-based set: Linearization

Linearization points:

Outcome: Item not in set

`add` Compare-exchange

`remove` Earlier of

- Point where found node is checked against `item`
- Point immediatly before new node is inserted (concurrent `add`)

`contains` Earlier of

- Point where a removed matching node, or a node with `node->item` > `item` is found
- Point immediatly before a new matching node is added to list

Lock-free list-based set: Improvements

Instead of restarting from head on compare-exchange (CAS) failure, maintain approximate backward links, traverse backwards and restart from close predecessor node instead. Can save (significantly?) on the cost of restart when list is long.

- Mikhail Fomitchev, Eric Ruppert: Lock-free linked lists and skip lists. PODC 2004: 50-59

(with no benchmark results)

Lock-free list-based set: Simple improvement (find)

The find operation needs to restart only if predecessor node has been deleted (marked).

```
Window find(T item) {
  // Search for item or successor
retry:
  Node* pred = head;
  Node* curr = getpointer(pred->next);

  while (true) {
    Node *succ = getpointer(curr->next);
    while (getflag(curr->next)) {
      resetflag(&curr);
      resetflag(&succ);
      if (!compare_exchange(&pred->next,
                            &curr, succ))
      {
        if (getflag(pred->next))
          goto retry;
        else succ = pred->next;
      }

      curr = succ;
      succ = getpointer(succ->next);
    }

    ... as before
```

Lock-free list-based set: Simple improvement (`delete`)

The `delete` operation only needs to ensure that the found item has become marked. Only one traversal of list.

Linearizes earlier (no `find` to clean up and discover that other thread successfully deleted).

```
bool remove(T item) {
  Window w;

  w = find(item);
  if (item != w.curr->item) return false;

  Node *succ = w.curr->next;
  Node *markedsucc = succ;
  // mark as deleted
  setflag(&markedsucc);
  resetflag(&succ);
  while (!compare_exchange(&w.curr->next,
                           &succ,
                           markedsucc)) {
    if (getflag(w.curr->next))
      return false;
  }

  // attempt to unlink curr
  compare_exchange(&w.pred->next,
                   &w.curr, succ);
  return true;
}
```

Simple improvements

The simple improvements were suggested and have been implemented in:

> Jesper Larsson Träff, Manuel Pöter: A more pragmatic implementation of the lock-free, ordered, linked list. PPoPP 2021: 457-459

with concrete code available via `https://github.com/parlab-tuwien/lockfree-linked-list`.

Another observation: For finding the point from which to start `find`, only aproximate predecessor pointers are needed.

Can these improvements and observations be used for similar data structures (skip lists, hash tables)?

Summary

Which implementation works best depends on the use-case, all ideas have merits and drawbacks.

- Coarse-grained synchronization (lock, synchronized regions, ...) can be used with any (sequentially efficient!) data structure (black box). Can easily become bottleneck when data structure operations are frequent
- All other implementations require deep knowledge of the data structure implementation, need care to achieve correctness, liveness (deadlock-freedom), and performance
- Lock- and wait-free data structures provide theoretically best progress guarantees, but come at a cost (expensive atomic operations, frequent need to restart validation)

References

- Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. SPAA 2002: 73–82.
- Timothy L. Harris: A Pragmatic Implementation of Non-blocking Linked-Lists. DISC 2001: 300–314
- John D. Valois: Lock-Free Linked Lists Using Compare-and-Swap. PODC 1995: 214–222
- Jesper Larsson Träff, Manuel Pöter: A more pragmatic implementation of the lock-free, ordered, linked list. PPoPP 2021: 457–459.

Advanced Multiprocessor Programming

Jesper Larsson Träff
traff@par.tuwien.ac.at
Research Group Parallel Computing
Faculty of Informatics, Institute of Computer Engineering
Vienna University of Technology (TU Wien)

©Jesper Larsson Träff

This is confusing!

©Jesper Larsson Träff

It's all about what can be observed!

©Jesper Larsson Träff

Memory behavior (hardware) and memory models (software)

Two issues:

1. Which memory behaviors do actual hardware allow? What can be observed? "What set of values is a read allowed to return"? Is this reasonable?

2. How can possible behaviors be modeled or controlled such that it becomes possible to reason about correctness of real programs? What is desirable? How can undesirable behaviors be constrained?

This can all be very confusing. Many subtleties. No entirely satisfactory answers to all questions

Desirable(?)

Something like sequential consistency (SC) for memory behavior:

- Reads and writes are observed by any thread in the order issued, but can be delayed
- Outcome of concurrent program is some interleaving of the core's instructions in program order for each core

This is not what hardware and compilers actually provide

Sequential computing: Programs on Random Access Machine

```
// initially A==0, B==0
register int x = A;

A = 7;
A = 27;
register int x = A;
A = 35;
register int y = A;
assert(x==27);
assert(y==35);
```

Not 7, not 35

Loads and stores take place in program order: Load from address A returns the value of the most recent store to A. No values from the distant past (earlier stores are overwritten) or the near future (no speculation, "out of thin air")

Sequential computing: Programs on Random Access Machine

```
// initially A==0, B==0
register int x = A;

A = 7;
A = 27;
register int x = A;
A = 35;
register int y = A;
assert(x==27);
assert(y==35);
```

Not 7, not 35

This makes it "easy" to reason about program properties (correctness): Assertions and invariants

©Jesper Larsson Träff

Sequential program order:

Running program P imposes a total order $<_P$ on loads and stores
L(A), S(B), A and B may be same or different memory location.

- L(A) $<_P$ S(B) if L(A) is executed before S(B) by P

And also between multiple loads and stores L1, L2, S1, S2:

- L1(A) $<_P$ L2(B) if L1(A) is executed before L2(B) by P
- S1(A) $<_P$ S2(B) if S1(A) is executed before S2(B) by P

Program: Sequence of instructions/code (C, Java, C++, machine code) executed by a thread running on a core
Hidden assumption: Compiler essentially preserves program order (where it matters, reordering must be observationally transparent)

Sequential semantics

```
// initially A==0, B==0
register int x = A;

A = 7;
A = 27;
register int x = A;
A = 35;
register int y = A;
assert(x==27);
assert(y==35);
```

$S(A,7) <_P S(A,27) <_P x=L(A) <_P S(A,35) <_P y=L(A)$

For any memory location A:
L(A) returns the value written by $MAX\{S(A) \mid S(A) <_P L(A)\}$

Sequential semantics allows the use of coherent caches

```
// initially A==0, B==0
register int x = A;

A = 7;
A = 27;
register int x = A;
A = 35;
register int y = A;
assert(x==27);
assert(y==35);
```

If A is stored in cache, all S(A) must update cache, L(A) must return value in cache

Sequential semantics allows the use of coherent caches

```
// initially A==0, B==0
register int x = A;

A = 7;
A = 27;
register int x = A;
A = 35;
register int y = A;
assert(x==27);
assert(y==35);
```

A coherent cache is observationally transparent (caching is functionally invisible): There is no way to determine by analyzing the outcomes of loads and stores whether the processor has cache(s)

Except by timing the operations (pragmatics)

©Jesper Larsson Träff

Sequential semantics permits compiler optimizations

```
// initially A==0, B==0
register int x = A;

A = 7; B = 13;
A = 27;
register int x = A;
A = 35;
register int y = A;
register z = B;
assert(z==13);
assert(x==27);
assert(y==35);
```

©Jesper Larsson Träff

Sequential semantics permits compiler optimizations

```
// initially A==0, B==0
register int x = A;

A = 7; B = 13;
register z = B;
A = 27;
register int x = A;
A = 35;
register int y = A;
assert(z==13);
assert(x==27);
assert(y==35);
```

and other hardware optimizations (prefetching)… as long as functional dependencies are respected (A and B must be loaded before C = A+B; cannot be reordered after store to C)

Sequential semantics permits further hardware optimizations

```
// initially A==0, B==0
register int x = A;

A = 7; B = 13;
register z = B;
A = 27;
register int x = A;
A = 35;
register int y = A;
assert(z==13);
assert(x==27);
assert(y==35);
```

Write buffer: Batches writes to memory (FIFO or other order).
Any load A must return latest value in write buffer (or cache)

Why hardware optimizations

Cost of actual, direct memory access order(s) of magnitudes slower than performing an operation on registers (and cache lines)

Caches, write buffers (prefetching, instruction reordering) can reduce average cost of memory accesses:

- Caches: Exploit (temporal and spatial) locality
- Write buffers: Hide/postpone latency of writes

Sequential semantics: What is not observed (aka dependencies), does not matter (speculative, superscalar hardware)

Template multi-core parallel processor

©Jesper Larsson Träff

©Jesper Larsson Träff

Cache coherence: Keeping caches observationally transparent

©Jesper Larsson Träff

Cache coherence: Keeping caches observationally transparent

©Jesper Larsson Träff

Cache coherence: Keeping caches observationally transparent

Cache incoherent:
If core 1 never sees the update to A

Desirable(?):
Eventually core 1 will see A==43

```
register int x = A;
A = x+1;
```

```
register int x = A;
register int y = A; // y==?
```


©Jesper Larsson Träff

<u>Definition 1</u> (SC like):
A cores' loads and stores to a single memory location must happen in a total order that respects the program order of each thread

<u>Definition 2</u> (Gharachorloo):
- Every store eventually becomes visible to each core
- Stores to the same memory location are serialized

<u>Definition 3</u> (Hennessy&Patterson):
1. A load from memory location A returns the value of the most recent store to A by the same core, unless another core stored to A inbetween
2. A load from memory location A returns the value of the most recent store from another core, provided the load and the store are "sufficiently separated in time"
3. Stores to the same memory location are serialized

<u>Definition 4</u> (epoch invariants): Assume lifetime of a memory location A divided into separate epochs

Cache is coherent if it maintains the following invariants:

- **Single-Writer, Multiple-reader**: For any memory location A, at any given time, either a single core may write to A (and read from A), or some number of cores may read from A
- **Data-value**: The value of the memory location at the start of an epoch is the same as the value at the end of the last write epoch.

Definitions 1-4 are equivalent.

Cache coherence protocol implements cache coherence (in hardware, transparently)

Cache coherence protocol (e.g, invalidate-based, MESI, …) maintains the invariants of Definition 4.

©Jesper Larsson Träff

Reminder: Cache granularity (performance and pragmatics)

Caches (normally) works on larger units than words, or programming language objects (int's, char's, double's, …):

- Cache block (cache line: cache block + meta information)

Coherence is at the level of cache blocks; this may lead to false sharing in coherent caches

Other reminders (performance factors):
Cache capacity, associativity, replacement strategy, …

Cache coherence vs. memory consistency

Cache coherence is about the behavior of operations on a single memory location:

- Coherent: Eventually, other threads will "see" the stores, and stores will be seen in interleaved program order

What about operations on different locations?

Cache coherence and different memory locations

Cache coherent, invalidation/update on A delayed, after B

```
A = 43;
B = 27;
```

```
register int y = B;
register int x = A;
```

Cache coherence and different memory locations

NOT $S(A,43) <_M S(B,27) <_M L(B==27) <_M L(A==42)$

Cache coherent, but possibly core 1 sees update to B first

Violates SC!

```
A = 43;
B = 27;
```

```
register int y = B; // B==27
register int x = A; // A==42
```


©Jesper Larsson Träff

Initially A and B are 0

Thread 0

```
A = data;
B = true;
```

Thread 1

```
while (!B);
register y = A;
```

Desirable (SC):
Thread 1 has y==data (and not y==0)

Common pattern:
Flag synchronization between two threads

Basic synchronization pattern between two threads:
Flag B determines when one thread has finished and the other can take over, all data written by thread 0 should be available to thread 1

©Jesper Larsson Träff

Initially A and B are 0

Thread 0

```
A = data;
B = true;
```

Thread 1

```
while (!B);
register y = A;
```

Store to A goes to write buffer, is delayed, and effectively reordered after store to B

Thread 1 could read A==0 into y

With non-FIFO write buffer, a reorder of both writes to A and B could be possible

©Jesper Larsson Träff

Initially A and B are 0

Thread 0

```
A = value1;
register int x = B;
```

Thread 1

```
B = value2;
register int y = A;
```

Obviously possible outcomes:

- (x,y) = (value2,value1)
- (x,y) = (0,value1)
- (x,y) = (value2,0)

- Threads run in lock-step
- Thread 0 before thread 1
- Thread 1 before thread 0

Initially A and B are 0

Thread 0

```
A = value1;
register int x = B;
```

W

Thread 1

```
B = value2;
register int y = A;
```

W

Obviously possible outcomes:

- (x,y) = (value2,value1)
- (x,y) = (0,value1)
- (x,y) = (value2,0)

- Threads run in lock-step
- Thread 0 before thread 1
- Thread 1 before thread 0

Is (x,y) = (0,0) possible?

Yes: Write buffers (even FIFO) may delay the (single) write per thread

©Jesper Larsson Träff

Initially A and B are 0

Thread 0

```
A = 1;
if (B==0) {
  // alone?
}
```

Thread 1

```
B = 1;
if (A==0) {
  // alone?
}
```

Simplified Dekker's algorithm: Is mutual exclusion guaranteed?

No. Write buffers may delay both stores to A and B

Initially A and B are 0

Non-hardware issue

Thread 0

```
A = value1;
register int x = B;
```

Thread 1

```
B = value2;
register int y = A;
```

Compiler orders load early to hide latency

```
register int x = B;
A = value1;
```

```
register int y = A;
B = value2;
```

Is (x,y) = (0,0) possible?

Yes: Compiler could reorder independent assignments

Consistency problems, consistency models

Hardware aspects: Cores read from and write to memory.

What can other cores observe? Why? How to model and axiomatize possible hardware behaviors

Software aspects: Threads read from and write to memory as determined by their programs.

What can threads observe? Why (hardware and compiler)? How to reason about possible behaviors? How write correct and portable programs?

Memory consistency model

Definition 1:
A memory consistency model constrains what a core (or thread) can observe on load and store operations to different locations

Definition 2 (Sorin, Hill, Wood):
A memory consistency model is a specification of the allowed load/store behavior of a multi-core (multi-threaded) program executing with shared memory.

An implementation (in hardware, or software) of a memory consistency model ensures that only allowed behaviors can happen (safety), and that some allowed behavior will happen (liveness)

Sequential consistency (SC)

Lamport:

A multi-processor (-core) system is sequentially consistent if "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program"

Leslie Lamport: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Trans. Computers 28(9): 690-691 (1979)

"Some sequential order…"

- All operations of all processor-cores can be totally ordered
- All operations (loads and stores to memory locations) in this order take place instantaneously

Call the total order of the operations on the (fictitious) global memory the memory order:

- $S(x) <_M L(y)$, or $L(y) <_M S(x)$, and
- $L1(x) <_M L2(y)$, or $L2(y) <_M L1(x)$, or
- …

for any loads and stores by any processor-cores

Sequential consistency (definition):

"…in the order specified by its program" (program order, $<_P$)

1. $L(x) <_P L(y)$ implies $L(x) <_M L(y)$
2. $L(x) <_P S(y)$ implies $L(x) <_M S(y)$
3. $S(x) <_P S(y)$ implies $S(x) <_M S(y)$
4. $S(x) <_P L(y)$ implies $S(x) <_M L(y)$

Furthermore, a load most return the value from memory written by the most recent store

$L(a)$ returns value written by $MAX\{S(a) \mid S(a) <_M L(a)\}$

Sequential consistency (definition):

"…in the order specified by its program" (program order, $<_P$)

Another way to specify SC: The following instruction (program) orders by a single processor must be preserved, not reordered:

1. Load -> Load
2. Load -> Store
3. Store -> Store
4. Store -> Load

"cannot be reordered":
If a thread performs loads and stores in order 1-4, no thread must be able to observe these loads and stores in a different order

SC: No reordering of memory instructions

Flag-synchronization works under SC (initially A, B = (0,0))

Thread 0

```
A = value1;
register int x = B;
```

Thread 1

```
B = value2;
register int y = A;
```

Under SC (x,y) = (0,0) is not possible!

Assume for the sake of contradiction that (x,y)==(0,0).

Then

- $L(B) <_M S(B)$
- $L(A) <_M S(A)$

By program order

$S(A) <_P L(B)$

$S(B) <_P L(A)$

Contradiction, by transitivity and SC: $L(B) <_M S(B) <_M L(A) <_M S(A)$

Implementing sequential consistency

Can SC be implemented in hardware? At what cost/penalty?

- Trivial implementation 1: Software multi-threading
- Trivial implementation 2: Funnel access to memory through switch (one core at a time has control over memory)
- Non-trivial implementations: Use cache-coherency protocol

Many believe that SC is too constraining on desired hardware optimizations (e.g. comes with a too high penalty). But not all:

Milo M. K. Martin, Mark D. Hill, Daniel J. Sorin: Why on-chip cache coherence is here to stay. Commun. ACM 55(7): 78-89 (2012)

The fictions of shared memory and global time (god&Newton)

The mental model of SC (and later TSO) is an actual, shared memory, and some globally ordered sequence of operations on this memory

However:

- Storing information takes physical space: Memory is always distributed (over the chip, over several chips, …)
- Updating memory takes (signal propagation) time, and the time to reach different physical locations may be different

- The order in which one processor observes updates may therefore be different from the order in which another/any other processor observes updates

Total Store Order (TSO)

Allow the use of FIFO write buffer by allowing stores to be ordered after loads

Write buffer:

- Stores are sent to the write buffer, and later written to memory
- Load L(A) must return value most recently written to A by processor (bypassing)

Thus: A store can be postponed till after a load in the memory order

Further non-FIFO write buffer optimizations

- Coalescing
- Sorting
- …

See later remarks

Memory receives from top of FIFO, processor writes to bottom, and reads (searches) from bottom (bypass)

```
A = 12;
…
B = 17;
A = 20;
A = 0;
register int z = A;
assert(z==0); // bypass
```

Coalescing would maintain only one entry for A

©Jesper Larsson Träff

Total Store Order (definition):

1. $L(x) <_P L(y)$ implies $L(x) <_M L(y)$
2. $L(x) <_P S(y)$ implies $L(x) <_M S(y)$
3. $S(x) <_P S(y)$ implies $S(x) <_M S(y)$
4. NOT: $S(x) <_P L(y)$ implies $S(x) <_M L(y)$

1. Load -> Load
2. Load -> Store
3. Store -> Store
4. NOT Store -> Load

Furthermore, a load must return the value written by the most recent store whether from memory or from write buffer

$L(a)$ returns value of $MAX\{S(a) \mid S(a) <_M L(a) \text{ or } S(a) <_P L(a)\}$

Flag based synchronization works under TSO:

1. Thread 0 reads and writes A0, A1, …, An
2. Thread 0 writes synchronization flag F

3. Thread 1 waits on synchronization flag F
4. Thread 1 reads what thread 0 has written in A0, A1, …, An

Even if the write to F of thread 0 is postponed, all loads and stores before must be observed as before F by thread 1

©Jesper Larsson Träff

Initially A and B are 0

Thread 0

```
A = 1;
if (B==0) {
  // alone?
}
```

Thread 1

```
B = 1;
if (A==0) {
  // alone?
}
```

Simplified Dekker's algorithm. Is mutual exclusion guaranteed?

Not under TSO

©Jesper Larsson Träff

Initially A and B are 0

Thread 0

```
A = value1;
register int a = A;
register int b = B;
```

Thread 1

```
B = value2;
register int x = B;
register int y = A;
```

Possible that (a,x) = (value1,value2) even if (b,y)==(0,0)?

Natural counter-argument: If (b,y)==(0,0) then the stores must have taken place after the loads into a and x, thus the values of a and x should also be 0

However, bypassing makes it possible for a = A; to load value1 (from core 0's write buffer) and x = B; to load value2, and load b = B; and y = A; from memory, thus (a,x)==(value1,value2) and (b,y)==(0,0)

Memory consistency models

A memory consistency model MC1 is weaker than a memory consistency model MC2 if the executions allowed by MC2 is a subset of the executions allowed by MC1 (the stronger model is more constraining on allowed executions)

Theorem:
TSO is weaker than SC

Two memory models are incomparable if neither is weaker than the other (weaker than only a partial order)

Enforcing order: The memory FENCE

A mechanism for enforcing order (preventing reordering) is needed

Memory FENCE (hardware instruction): Core-local memory operation across which other memory operations cannot be reordered

Bad names (but often used): Memory barrier, sync

FENCE is core/thread local, does not involve other cores/threads, it is not a synchronization operation

Total Store Order FENCE (definition):

1. $L(x) <_P FENCE$ implies $L(x) <_M FENCE$
2. $S(y) <_P FENCE$ implies $S(y) <_M FENCE$
3. $FENCE <_P FENCE$ implies $FENCE <_M FENCE$
4. $FENCE <_P L(x)$ implies $FENCE <_M L(x)$
5. $FENCE <_P S(y)$ implies $FENCE <_M S(y)$

For TSO, it suffices to state

1. $S(y) <_P FENCE$ implies $S(y) <_M FENCE$
2. $FENCE <_P L(x)$ implies $FENCE <_M L(x)$

Total Store Order FENCE (definition):

Equivalently, the following orders must be preserved

1. Load -> FENCE
2. Store -> FENCE
3. FENCE -> FENCE
4. FENCE -> Load
5. FENCE -> Store

In addition to

1. Load -> Load
2. Load -> Store
3. Store -> Store

But NOT Store -> Load

©Jesper Larsson Träff

Using FENCE for SC with TSO

In TSO, Store can be postponed and ordered after Load (same: Load can be ordered before Store), only difference to SC model

To get SC behavior, FENCE must be inserted after each Store that may be followed by a Load (conservative, but expensive: after each Store)

TSO in hardware allows the use of FIFO write buffers

TSO seems to be the memory model of x86-processors

- No formal specification published
- Vendors (Intel) do not reveal all details

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, Magnus O. Myreen: x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM 53(7): 89-97 (2010)

Exercise:
Write and execute a program that shows that your x86 is not SC (but hopefully TSO)

Atomic operations (registers)

- Load
- Store
- RMW (read-modify-write)

Atomic load/store operations: Totally ordered, when some thread observes the effect of an atomic operation, all threads can observe the same effect (atomicity)

RMW: Atomically read and update memory location (globally ordered) without any intervening updates to the location

Implementing atomic RMW operations

- Special hardware (combining circuits), rare
- Exploiting cache-system: Get exclusive access to cacheline for the duration of the atomic operation, cache lines of other threads must be invalidated

Write buffer: Must be drained

On some hardware, atomic operations often entails a (or some type of) FENCE, but not always so

C11 and C++11 atomic operations

exchange (get-and-set), compare_exchange (CAS), fetch_add (fetch-and-add), …

See earlier lecture

```
#include <stdatomic.h>
void atomic_thread_fence(memory_order order);
```

```
#include <atomic>
extern "C"
void atomic_thread_fence(std::memory_order order);
```


©Jesper Larsson Träff

Neither C nor C++ were designed for multi-treading(!):

- Compilers were/are mostly thread-unaware and might perform optimizations that break expectations on memory model
- Threading support was originally added through libraries
- Thread libraries provide synchronization primitives to prevent data races (see later)
- Compilers explicitly forbidden to reorder across synchronization primitives

Hans-Juergen Boehm: Threads cannot be implemented as a library. PLDI 2005: 261-268

With C11 and C++11, threading support and explicit memory models were built into the languages

Enforcing load from memory: `volatile`

"volatile" means that the location can be modified from outside (other thread or device). Volatile accesses not reordered inside threads. But "volatile" does not mean "atomic". Use case:

```
volatile bool locked;

void lock() { // test-and-test-and-set lock
  while (true) {
    while (locked);
    if (!atomic_exchange(&locked,true)) return;
  }
}

void unlock() {
  locked = false;
}
```

Does this work without volatile?

```
bool locked;

void lock() { // test-and-test-and-set lock
  while (true) {
    while (locked);
    if (!atomic_exchange(&locked,true)) return;
  }
}
```

Compiler might optimize load away

```
bool locked;

void lock() {
  while (true) {
    if (locked) while (true);
    if (!atomic_exchange(&locked,true)) return;
  }
}
```

Infinite loop

©Jesper Larsson Träff

We will also need this (volatile pointer):

```
node_t *volatile pred, *volatile curr,
       *volatile * succ;
```

e.g., list-based set, skip list, pointers with stolen bits (marks, time stamps) that may/will be updated by other threads

Even weaker (relaxed) consistency models: Motivation

Flag synchronization does actually not require TSO:

1. Thread 0 reads and writes A0, A1, …, An
2. Thread 0 writes synchronization flag F

3. Thread 1 waits on synchronization flag F
4. Thread 1 reads what thread 0 has written in A0, A1, …, An

Load/Store order in Step 1 (and 4) does not have to be respected. It is only required that thread 1 "sees" all updates after the flag has been set

Some hardware/memory organizations permit (almost) all reorderings

Flag synchronization does actually not require TSO:

1. Thread 0 reads and writes A0, A1, …, An
2. Thread 0 writes synchronization flag F

3. Thread 1 waits on synchronization flag F
4. Thread 1 reads what thread 0 has written in A0, A1, …, An

Required order for correct flag synchronization (hand-over):

$L(Ai),S(Ai) <_M S(F) <_M L(F) <_M L(Ai),S(Ai)$

©Jesper Larsson Träff

Further hardware optimization potential with less constraints

1. Coalescing write buffer

```
A = 27;
B = 45;
```

©Jesper Larsson Träff

Further hardware optimization potential with less constraints

1. Coalescing write buffer

Other threads may see A = 18; before B = 45;

- NOT Store -> Store

NUMA (Non-Uniform Memory Access): Memory affinity

Shared memory divided physically into modules, each module is closer to some cores than to other cores

Performance/Pragmatics: Placement of program memory can be controlled (to some extent), e.g., first-touch policy

©Jesper Larsson Träff

NUMA (Non-Uniform Memory Access): Memory affinity

Initially A and B are 0

NOT Store -> Store

May see A==0, B==32 (if write to A in other module is slow)

©Jesper Larsson Träff

NUMA (Non-Uniform Memory Access): Memory affinity

Initially A and B are 0

(x,y) = (0,0) may happen, even without write buffer (writes to the locations far away slow, reading the close locations fast):
NOT Store -> Load

©Jesper Larsson Träff

Independent Reads of Independent Writes (IRIW)

There are architectures (POWER, ARM) where this can happen

Core 0:

```
A = 27;
```

Core 1:

```
B = 35;
```

Core 2:

```
register int x = A;
register y = B;
```

Core 3:

```
register int b = B;
register a = A;
```

Assume x==27 and b==35.

- If core 1 is farther away from core 2 than core 0, core 2 may load y==0
- If core 0 is farther away from core 3 than core 1, core 3 may load a==0

(x,b) = (27,35) but (y,a) = (0,0)

- Core 2 observes: S(A) < S(B)
- Core 3 observes: S(B) < S(A)

No (total) memory order!

Independent Reads of Independent Writes (IRIW) under SC

Core 0:

```
A = 27;
```

Core 1:

```
B = 35;
```

Core 2:

```
register int x = A;
register y = B;
```

Core 3:

```
register int b = B;
register a = A;
```

Assume x==27 and b==35. Assume further y==0

Then $S(A,27) <_M L(A==27) <_M L(B==0) <_M S(B,35) <_M L(B==35) <_M L(A)$, so it must be that a==27

Under SC (and TSO), it cannot happen that both (y,a)==(0,0)

Independent Reads of Independent Writes (IRIW)

Core 0:

```
A = 27;
```

Core 1:

```
B = 35;
```

Core 2:

```
register int x = A;
register y = B;
```

Core 3:

```
register int b = B;
register a = A;
```

Assume x==27 and b==35.

FENCE operations cannot prevent y==0, a==0 (without further axioms), core 2 and core 3 can read A==27 and B==35 before the FENCEs that would have prevented both y==0 and a==0

©Jesper Larsson Träff

Independent Reads of Independent Writes (IRIW)

Core 0:

```
A = 27;
```

Core 1:

```
B = 35;
```

Core 2:

```
register int x = A;
register y = B;
```

Core 3:

```
register int b = B;
register a = A;
```

Assume x==27 and b==35.

FENCE operations cannot prevent y==0, a==0 (without further axioms)

Problem: Stores do not take place immediately, possible that some threads observe the outcome of a store, but others not (yet)

Write atomicity (store atomicity):
Stores/writes take place immediately: If some core can observe the result of a store, then so can all cores

IRIW example:
Weak hardware memory models (e.g, NUMA systems) can violate write atomicity

Write atomicity implies that IRIW is correctly handled:
If x==27 and b==35, and furthermore y==0, then $S(A,27) <_M L(A==27) <_M L(B==0) <_M S(B,35) <_M L(B==35)$ which implies L(A==27) by core 3, so a=27

Causality: If a location is read by one thread, a strictly later thread will see the same (or later) value

```
A = 27;
```

```
while (A!=27);
FENCE;
B = 35;
```

```
while (B!=35);
FENCE;
register int x = A;
```

By causality: $S(A,27) <_M L(A) <_M S(B) <_M x=L(A)$ and $x=27$

Observations:

- Write atomicity implies causality
- Causality does not imply write atomicity

©Jesper Larsson Träff

Causality does not imply write atomicity (counter example)

Core 0, hyperthread 0:

```
A = 27;
```

W

Core 0, hyperthread 1:

```
register int x = A;
register y = B;
```

Core 1, hyperthread 0:

```
B = 35;
```

W

Core 1, hyperthread 1:

```
register int b = B;
register a = A;
```

Hyperthreads share write buffer (and other hw): Possible that x==27 and b==35 (bypassing), and y==0 and a==0 (stores are in write buffer), so write atomicity does not hold.

Core 0, hyperthread 0:

```
A = 27;
```

W

Core 1, hyperthread 0:

```
while (B!=35); FENCE;
register int x = A;
```

W

Core 1, hyperthread 1:

```
while (A!=27);
FENCE;
B = 35;
```

On this example architecture, causality holds, A must have left write buffer before core 1, hyperthread 1 writes to B, and can therefore be read also by hyperthread 0

©Jesper Larsson Träff

Core 0, hyperthread 0:

```
A = 27;
```

W

Core 0, hyperthread 1:

```
while (A!=27);
FENCE;
B = 35;
```

Core 1, hyperthread 0:

```
while (B!=35); FENCE;
register int x = A;
```

W

Core 1, hyperthread 1:

Causality holds, FENCE enforces L(A==27) $<_M$ S(B,35) in memory order observed by core 1

Not a proof, example(s) from

Daniel J. Sorin, Mark. D. Hill, David A. Wood: A primer on memory consistency and cache coherence. Synthesis Lectures on Computer Architecture. Morgan&Claypool, 2011

Generic, weak memory consistency model XC (definition):

1. $L(x) <_P FENCE$ implies $L(x) <_M FENCE$
2. $S(y) <_P FENCE$ implies $S(y) <_M FENCE$
3. $FENCE <_P FENCE$ implies $FENCE <_M FENCE$
4. $FENCE <_P L(x)$ implies $FENCE <_M L(x)$
5. $FENCE <_P S(y)$ implies $FENCE <_M S(y)$

The (total) $<_M$ relation describes what can be observed, not a physical memory.

Loads and stores to same location respect

1. $L(a) <_P L'(a)$ implies $L(a) <_M L'(a)$
2. $L(a) <_P S(a)$ implies $L(a) <_M S(a)$
3. $S(a) <_P S'(a)$ implies $S(a) <_M S'(a)$

Load gets value of last local or global store

$L(a)$ returns value of $MAX\{S(a) \mid S(a) <_M L(a) \text{ or } S(a) <_P L(a)\}$

Equivalently, the following orders are preserved

1. Load -> FENCE
2. Store -> FENCE
3. FENCE -> FENCE
4. FENCE -> Load
5. FENCE -> Store

For accesses to same location (same as TSO):

1. Load -> Load
2. Load -> Store
3. Store -> Store

Model	Impl.	Load/Store	Fence
SC	Optional/Default	L->L, L->S, S->L, S->S	F->S, F->L, L->F, S->F, F->F
TSO (Processor Consistency)	x86, IBM 370, DEC VAX, (Sparc)	L->L, L->S, S->S	Same
Partial Store Order	Sparc	L->L, S->S	Same
Weak Ordering	PowerPC (IBM)		Same
Release Consistency	Alpha, MIPS		FA->S, FA->L, L->FA, W->FR, FA->FA, FA->FR, FR->FA,FR->FR

From Hennessy/Patterson, "Computer Architecture", 2nd Ed. 1996

©Jesper Larsson Träff

Release consistency

FENCE operations for "ending a load store sequence" (RELEASE FENCE, FR) and "beginning a load store sequence" (ACQUIRE FENCE, FA):

1. Load, Store -> RELEASE
2. ACQUIRE -> Load, Store
3. ACQUIRE -> ACQUIRE
4. ACQUIRE -> RELEASE
5. RELEASE -> ACQUIRE
6. RELEASE -> RELEASE

Scott's notation

Distinguish between synchronizing (atomic, fences) and ordinary memory operations (non-atomic, load, store). Together, call these "operations"

From:

Michael L. Scott: Shared-memory synchronization. Synthesis Lectures on Computer Architecture. Morgan&Claypool, 2013

Terminology, assumptions:

- **Coherence**: All operations (synchronizing, ordinary) to any single location appear to occur in some single, total order from the perspective of all threads
- **Global order**: Synchronizing operations happen in some total order, in which operations by the same thread happen in program order
- **Program order**: Operations happen in the order specified by the (compiled) program from the perspective of the issuing thread; other threads may see operations in a different order
- **Local order**: Ordinary operations may be reordered wrt. synchronizing operations by the issuing thread, except when explicitly forbidden
- **Value read**: Any read operation returns the value most recently written value in the order observed by the thread

Local order of ordinary operations wrt. synchronizing operations is specified explicitly with the following annotation on the synchronizing operation

x.load/store/rmw/fence(arg,P||S)

||: A fence across which the specified kinds of local operations cannot be reordered
P: Preceding, local R(ead) and/or W(rite)
S: Succeeding, local R(ead) and/or W(rite)

```
a1 = x1; a2 = x2; x3 = a3;
a = x.load(R||W); // atomic
b = x4;
```

Reads of x1, x2 cannot be ordered after read of x (write to x3 can). Write to b cannot be ordered before read of x

©Jesper Larsson Träff

Locks (and data structures):

Lock release must ensure that all data read and written in the critical section must be visible (all load/stores) completed. Lock release must have at least a RW|| fence (as part of a synchronizing store)

Lock acquire should ensure that the thread cannot read/write shared data before it has the lock. Acquire should be prefixed by a ||RW fence (as part of synchronizing load/store)

Lock implementations (pthreads library, OpenMP, C++) usually make this guarantee

Annotations and hardware memory consistency

Hardware SC: No || annotations needed, hardware guarantees global order consistent with program order

Hardware TSO: R||R, R||W and W||W annotations not needed, these orderings are guaranteed by hardware; but W||R must be asserted when needed

Relaxed models: Any reorderings are possible, all required orders must be asserted

fence(RW||RW): Full fence

Portable (pseudo-)code: Make all ordering requirements explicit

Example: Test-and-test-and-set lock

```
int f = 0;

void lock() {
  while (test_and_set(&f))
    while (f); // spin on f
  fence(||RW);
}

void unlock() {
  f.store(0,RW||);
}
```


©Jesper Larsson Träff

Memory consistency in programming languages

Concurrent/multi-threaded programming language must present a clear memory model to the programmer (which may or may not be identical to the model provided by the hardware):

- Portability
- (Formal) Reasoning about correctness
- Shielding from (too) strange effects of hardware model
- Control and constrain compiler optimizations (reordering, register usage)
- Safety (Java)

Adopting SC as the model is generally thought too restrictive and expensive (but feasible, compiler inserts FENCEs)

C++/Java approach

- Programming language makes memory consistency model explicit (and threading model as well)
- Memory model can be used by programmer to enforce required load/store order

- Memory model must be respected by compiler optimizations

- Desirable if weaker (but faster) hardware memory consistency can be exploited (portably) by programmer

Data race, race condition

Data race (race condition):

Two or more cores (threads) read/write to a memory location with ordinary read/write operations, and at least one operation is a write

Outcome of a program with a race condition depends on the relative order of execution and of reads/writes, and is considered bad (non-deterministic, undefined behavior…)

Another view: Order of memory operations matter only where threads interact, through accessing shared memory locations (threads locally obey program order). Control order by controlling the races…

Race conditions avoided by protecting all updates to shared memory locations by synchronization constructs (e.g., locks) or by using atomic operations that establish an inter-thread order

Data Race Free (DRF):

A program is DRF if it has no data races when executed under SC: All accesses to shared locations are either via synchronization operations (atomics), or separated (ordered) by synchronization operations.

Alternatively:
A program is properly synchronized if all accesses to shared locations are ordered by synchronization operations

A DRF program can be treated as if it was executed under SC

- Between synchronization points, SC is established by local rules on program order.
- At synchronization points, FENCES may have to be added to ensure that memory operations are properly ordered

Sequential Consistency for Data Race Free programs: SC for DRF

- Programmer must ensure that the program is DRF if executed under SC
- Implementer must ensure that all executions of a DRF program behave like an SC execution (by inserting FENCEs)

Note: Optimal placement of FENCEs is undecidable; conservative placement not (extreme: FENCE after every load/store)

SC for DRF:

- Programmer must ensure that the program is DRF if executed under SC
- Implementer must ensure that all executions of a DRF program behave like an SC execution (by inserting FENCEs)

Advantage:
SC for DRF allows (almost) all common compiler and hardware optimizations between synchronization points, order must be enforced at synchronization points possibly with fence operations

In particular, SC for DRF can be implemented on hardware with only weak memory ordering

Definition: SC for DRF

- Operations are either access (Load/Store) or synchronization
- Two data operations Di and Dj conflict if they are from different cores (threads), access the same location and at least one is a Store
- Two synchronization operations conflict if they are from different cores (threads), if they access the same location, and are not compatible
- Synchronization conflict is transitive
- Two data operations race if they conflict, and they are not separated by a conflicting synchronization
- An SC execution is Data Race Free (DRF) if no operations race
- A program is DRF if all its SC executions are DRF
- A memory consistency model supports "SC for DRF" if all executions of all DRF programs are SC executions

C++11/C11 Memory model

- SC for DRF, undefined semantics for programs with data races (unlike Java)
- Atomic objects act as synchronization operations (establish order)
- Load/store/atomic (RMW) operations with explicit memory model
- Default model (for atomic loads and stores) is SC

Hans-Juergen Boehm, Sarita V. Adve: Foundations of the C++ concurrency memory model. PLDI 2008: 68-78

Resource: www.cplusplus.com

Establishing order in C/C++

- Atomic operations by different threads establish synchronizes-with relationship

- Synchronizes-with is transitive between threads
- Local operations has a happens-before (local) order(*)

- Inter-thread happens-before:
- Operation A (by thread 0) happens-before S (by thread 0) synchronizes-with S (by thread 1) happens-before B (by thread 1): A inter-thread happens-before B

- Memory operations are visible to threads in happens-before order

(*) not quite that: happens-before by sequenced-before

C++ flag synchronization with atomics

```
#include <atomic>
#include <thread>
std::atomic<bool> f;

…
a = data;
f.store(true);
…
```

synchronizes-with

```
#include <atomic>
#include <thread>
std::atomic<bool> f;

…
while (!f.load());
b = a;
```

Store to a happens-before load from a, no race

Atomic store synchronizes with atomic load (under the default memory model: SC)

©Jesper Larsson Träff

Explicit memory model orderings on atomic variables:

- Acquire-Release
- Consume (?)
- Relaxed (no order, used in combination with other operations)

Load

```
memory_order_seq_cst
memory_order_relaxed
memory_order_acquire
memory_order_consume
```

Store

```
memory_order_seq_cst
memory_order_relaxed
memory_order_release
```

RMW

```
memory_order_seq_cst
memory_order_relaxed
memory_order_acquire
memory_order_release
memory_order_acq_rel
memory_order_consume
```

Synchronizes-with relationship (approximate):

- `memory_order_seq_cst` with `memory_order_seq_cst`
- `memory_order_acquire` (on load/RMW) with `memory_order_release` (on store/RMW)
- `memory_order_acq_rel` (on RMW) with `memory_order_release` (on store)
- `memory_order_acq_rel` (on RMW) with `memory_order_acquire` (on load)
- `memory_order_acq_rel` with `memory_order_acq_rel` (on RMW)
- `memory_order_release` (on store) with `memory_order_consume` (on load) (*)
- `memory_order_relaxed` does not synchronize with anything

(*) The `memory_order_consume` model may be deprecated (2016)

```
#include <atomic>
#include <thread>
std::atomic<bool> f;

…
a = data;
f.store(true,
        memory_order_release);
…
```

```
#include <atomic>
#include <thread>
std::atomic<bool> f;

…
while (!f.load(memory_order_acquire));
b = a;
…
```


©Jesper Larsson Träff

```
#include <atomic>
#include <thread>
std::atomic<bool> f;

…
a = data;
f.store(true,
        memory_order_release);
A = someotherdata;
```

Operations before atomic store cannot be ordered after

With acquire-release, the atomic store may be ordered after succeeding local operations

```
#include <atomic>
#include <thread>
std::atomic<bool> f;

…
while (!f.load(memory_order_acquire));
b = a;
B = A; // ?? No happens-before
…
```


©Jesper Larsson Träff

```
#include <atomic>
#include <thread>
std::atomic<bool> f;

…
a = data;
A = someotherdata;
f.store(true,
        memory_order_release);
```

Operations after atomic load cannot be ordered before

With acquire-release, the preceding local operations may be ordered after the atomic load

```
#include <atomic>
#include <thread>
std::atomic<bool> f;

…
B = A; // ?? No happens-before
while (!f.load(memory_order_acquire));
b = a;
…
```


©Jesper Larsson Träff

Scott's notation:

- `.store(val,memory_model_release)` like `.store(val,RW||)`
- `.load(memory_model_acquire)` like `.load(||RW)`

C++ IRIW under memory_order_seq_cst

```
A.store(27,
        memory_order_seq_cst);
```

```
B.store(35,
        memory_order_seq_cst);
```

```
while (A.load(memory_order_seq_cst)!=27);
y = B.load(memory_order_seq_cst);
```

```
while (B.load(memory_order_seq_cst)!=35);
a = A.load(memory_order_seq_cst);
```

Under SC, either y==35, or a = 27 (or both), since the stores are totally ordered

©Jesper Larsson Träff

C++ IRIW under memory_order_acq_rel

```
A.store(27,
        memory_order_release);
```

```
B.store(35,
        memory_order_release);
```

```
while (A.load(memory_order_acquire)!=27);
y = B.load(memory_order_acquire);
```

```
while (B.load(memory_order_acquire)!=35);
a = A.load(memory_order_acquire);
```

Possible that y==0 and a==0, because no happens-before between stores to A and B. Acquire-release does not establish a total order, and does not guarantee atomicity

Acquire-release guarantees causality

```
A.store(27,memory_order_release);
f1.store(true,memory_order_release);
```

```
while (!f1.load(memory_order_acquire));
f2.store(true,memory_order_release);
```

```
while (!f2.load(memory_order_acquire));
a = A.load(memory_order_acquire);
```

Store to A happens-before load from A, since synchronizes with is transitive

©Jesper Larsson Träff

Acquire-release guarantees causality (also under relaxed store/load)

```
A.store(27,memory_order_relaxed);
f1.store(true,memory_order_release);
```

```
while (!f1.load(memory_order_acquire));
f2.store(true,memory_order_release);
```

```
while (!f2.load(memory_order_acquire));
a = A.load(true,memory_order_relaxed);
```

Store to A happens-before load from A, since "synchronizes with" is transitive

Acquire-release guarantees causality (also under relaxed store/load)

```
A =27;
f1.store(true,memory_order_release);
```

```
while (!f1.load(memory_order_acquire));
f2.store(true,memory_order_release);
```

```
while (!f2.load(memory_order_acquire));
a = A;
```

Store to A happens-before load from A, since "synchronizes with" is transitive

©Jesper Larsson Träff

```
#include <atomic>
#include <thread>
std::atomic<bool> f;

…
a = data;
atomic_thread_fence(memory_order_release);
f.store(true,memory_order_relaxed);
…
```

The two fences synchronizes -with each other, and establish happens-before on a

```
#include <atomic>
#include <thread>
std::atomic<bool> f;

…
while (!f.load(memory_order_relaxed));
atomic_thread_fence(memory_order_acquire);
b = a;
…
```


©Jesper Larsson Träff

CLH lock in C++11

```
struct Node {
  Node *pred;
  atomic<bool> locked;
};
void lock() {
  node = new Node();
  node->locked.store(true,memory_order_seq_cst);
  node->pred = tail.exchange(node,memory_order_seq_cst);
  while (node->pred->locked.load(memory_order_seq_cst));
}
void unlock() {
  delete node->pred;
  node->locked.store(false,memory_order_seq_cst);
}
```

SC is the default, can be left out

©Jesper Larsson Träff

First relaxation: Release for writes, acquire for reads

```
struct Node {
  Node *pred;
  atomic<bool> locked;
};
void lock() {
  node = new Node();
  node->locked.store(true,memory_order_release);
  node->pred = tail.exchange(node,memory_order_acq_rel);
  while (node->pred->locked.load(memory_order_acquire);
}
void unlock() {
  delete node->pred;
  node->locked.store(false,memory_order_release);
}
```


©Jesper Larsson Träff

Second relaxation: Relax first write (will be released by second)

```
struct Node {
  Node *pred;
  atomic<bool> locked;
};
void lock() {
  node = new Node();
  node->locked.store(true,memory_order_relaxed);
  node->pred = tail.exchange(node,memory_order_acq_rel);
  while (node->pred->locked.load(memory_order_acquire);
}
void unlock() {
  delete node->pred;
  node->locked.store(false,memory_order_release);
}
```


©Jesper Larsson Träff

Further reading

Sarita V. Adve, Hans-Juergen Boehm: Memory models: a case for rethinking parallel languages and hardware. Commun. ACM 53(8): 90-101 (2010)
Hans-Juergen Boehm, Sarita V. Adve: You don't know jack about shared variables or memory models. Commun. ACM 55(2): 48-54 (2012)

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, Peter Sewell: The Problem of Programming Language Concurrency Semantics. ESOP 2015: 283-307
Hans-Juergen Boehm, Sarita V. Adve: Foundations of the C++ concurrency memory model. PLDI 2008: 68-78

Java

Adapts SC for DRF. But Java also needs to provide semantics for programs that are not DRF:

- safe and secure language
- must be able to run untrusted code

Arbitrary behavior is not acceptable

New memory model with Java 5 (earlier models were broken)

```
Singleton get() {
  if (singleton==null) {
    synchronized (this) {
      if (singleton==null)
        singleton = new Singleton();
    }
  }
  return singleton;
}
```

Double checked locking (idea: Prevent expensive lock if singleton already initialized), broken in old Java memory model: Pointer might be set before constructor has been executed

©Jesper Larsson Träff

OpenMP, pthreads

Idea (similar to MPI one-sided communication):
Conceptually separated local and global views on memory, threads operate on local view, OpenMP constructs propagate memory updates to global view

Greg Bronevetsky, Bronis R. de Supinski: Complete Formal Specification of the OpenMP Memory Model. International Journal of Parallel Programming 35(4): 335-392 (2007)
Jay P. Hoeflinger, Bronis R. de Supinski: The OpenMP Memory Model. IWOMP 2005: 167-177

MPI one-sided communication model

Conceptually, local and global views of memory; global view visible to other threads, closing and opening of epoch determine when local views become global, and global view locally visible; additional synchronization (fence) constructs

Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, Keith D. Underwood:
Remote Memory Access Programming in MPI-3. TOPC 2(2): 9:1-9:26 (2015)

Further reading (and there's lots more):

Daniel J. Sorin, Mark. D. Hill, David A. Wood: A primer on memory consistency and cache coherence. Synthesis Lectures on Computer Architecture. Morgan&Claypool, 2011

Michael L. Scott: Shared-memory synchronization. Synthesis Lectures on Computer Architecture. Morgan&Claypool, 2013

Anthony Williams: C++ Concurrency in Action. Manning, 2012

S. V. Adve, K. Gharachorloo: Shared Memory Consistency Models: A Tutorial. IEEE Computer, 29(12):66-76, 1996

Bjarne Stroustrup: The C++ Programming Language. 4th Ed., Addison-Wesley, 2013

The concurrent memory reclamation problem

In non-garbage collected languages (C, C++)

```
free(p);
```

gives node pointed to by p back to memory manager

With only one thread:
(mostly; if not, use reference counting) easy to know that there is only one reference (beware: circular references!), node can safely be given back, since it will never be referenced again

©Jesper Larsson Träff

With more than one thread:
Safe to give node back only when no threads are referencing.
How can free'ing thread know that no other threads reference the node?

©Jesper Larsson Träff

With more than one thread:
Safe to give node back only when no threads are referencing.
How can free'ing thread know that no other threads reference the node?

If A frees node, information (next pointer) needed by B will be corrupted

©Jesper Larsson Träff

Memory reclamation scheme wish list:

- General, should work for any data structure, no constraints on structure of allocated nodes, no special coding discipline
- Portable, should require no OS support, no not commonly supported hardware instructions (DCAS etc.)
- Thread oblivious, no fixed upper bound on number of threads
- Strong progress guarantees (wait-free, lock-free, not reclamation blocking)
- Efficient, constant-time overhead per reclaimed node
- …

Reclamation blocking: A stalled (crashed) thread can prevent reclamation of an unbounded amount of memory

Lock-Free Reference Counting (LFRC)

Memory allocated and free'd as "nodes".

Add a reference count to each node. When a thread references a node, the reference count is incremented, when a thread drops the reference, the reference count is decremented. When a reference count drops to 0, this node may be reclaimed, reclaiming thread sets "reclaim" bit.

Drawback of reference counting:
Nodes in cyclic data structures may never be freed (memory is leaked)

LFRC

- Get reference: FAA on reference count, validate reference with CAS (on failure, decrease reference count, drop reference, retry)
- Drop reference: CAS on reference count, if 0 reclaim (by setting reclaim bit with the CAS)

Because getting a reference involves two steps, LFRC can only be used when nodes are being reused in the same data structure, the reference count must be available indefinitely

LFRC: Not general, portable, thread oblivious, reclamation blocking, not efficient

John D. Valois: Lock-free data structures. PhD thesis, Rensselaer Polytechnic Institute, 1995
Maged M. Michael, Michael L. Scott: Correction of a memory management method for lock-free data structures. TR, University of Rochester, 1995

Håkan Sundell: Wait-Free Reference Counting and Memory Management. IPDPS 2005
Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, Philippas Tsigas: Efficient and Reliable Lock-Free Memory Reclamation Based on Reference Counting. IEEE Trans. Parallel Distrib. Syst. 20(8): 1173-1187 (2009)

Hazard Pointers (HP)

Sensitive nodes are referenced through special pointers: Hazard pointers.

Each thread has some fixed number K of hazard pointers (total number of hazard pointers is pK when p threads are running). The hazard pointers is the way for a thread to communicate to other threads that it is holding a reference to a shared node. When a thread wants to reclaim a node, it must scan the hazard pointers of all other threads to find out whether there are other references to the node

HP: Not general, portable, not thread oblivious, not reclamation blocking, not efficient. And patented(!)

Maged M. Michael: Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. IEEE Trans. Parallel Distrib. Syst. 15(6): 491-504 (2004)

Stamp-it: A new, epoch-based reclamation scheme

- General: Little special coding discipline required
- Portable: Uses FAA, CAS, EX, no OS support
- Thread oblivious
- Good progress: lock-less (but reclamation blocking)
- Amortized constant-time reclamation per node

Design goal: Avoid inspection of all threads on reclamation

Ingredients:
- Local retire list of nodes to be reclaimed
- Global retire list
- Stamp Pool of thread elements

Epoch-based schemes (ER, NER, IBR, …)*:

- Programmer has to mark critical regions (not to be confused with critical sections…) where nodes may be shared. Can be supported with C++ interface (not yet in standard)
- Scheme tracks entry to and exit from critical regions

Stamp-it key idea:
Stamp Pool data structure to order entry to critical regions

*

Keir Fraser: Practical lock-freedom. University of Cambridge, UK 2004
Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, Jonathan Walpole: Performance of memory reclamation for lockless synchronization. J. Parallel Distrib. Comput. 67(12): 1270-1285 (2007)

Stamp Pool:
Maintains time-stamped elements (one per thread), stamps monotonically increasing

©Jesper Larsson Träff

Stamp Pool:
Maintains time-stamped elements (one per thread), stamps monotonically increasing

Implementation:
Lock-free doubly linked list (CAS), based on Sundell/Tsigas, O(1) operations in absence of conflicts. New stamp by FAA

Correct in C++ memory model (acquire-release)

Håkan Sundell, Philippas Tsigas: Lock-free deques and doubly linked lists. J. Parallel Distrib. Comput. 68(7): 1008-1020 (2008)

©Jesper Larsson Träff

Thread A enters, Pool insert

©Jesper Larsson Träff

©Jesper Larsson Träff

Thread A retires node, get highest

retire

Thread B in critical region (1)

©Jesper Larsson Träff

Thread A retires node

Thread B in critical region (1)

©Jesper Larsson Träff

©Jesper Larsson Träff

Thread C in critical region (2)

Thread A in critical region

Thread B in critical region (1)

©Jesper Larsson Träff

©Jesper Larsson Träff

Scan local retire list, all retired nodes with stamp smaller than lowest can be freed

©Jesper Larsson Träff

Thread C in critical region (2)

Thread A leaves critical region

retire

Scan local retire list, all retired nodes with stamp smaller than lowest can be freed

©Jesper Larsson Träff

Thread C leaves critical region

©Jesper Larsson Träff

Thread C leaves critical region, delete, get lowest, local retire

retire

©Jesper Larsson Träff

If local retire list gets too large, move to global retire list

©Jesper Larsson Träff

Properties:

- Nodes in local retire lists are in increasing stamp order. Reclaiming all nodes with stamp less than currently lowest stamp in O(1) per reclaimed node
- Global retire list is list of ordered sublists. Carefully chosen threshold for amortization argument

Retire list operations:

- Append local retire list to global retire list: CAS
- Reclaim from global retire list: Last thread (remove=**true**) grabs list with EX

Claim 1:
Stamp-it is reclamation safe (correct). A node is reclaimed only when it is referenced by no thread

Claim 2:
Stamp-it reclaims any node in amortized constant time in the number of data structure (Stamp Pool) operations

Claim 3:
Stamp-it is lock-less. All Stamp Pool operations and other operations for entering/leaving critical regions are lock-free. If threads do not conflict, all operations are O(1)

Claim 4:
The Stamp-it implementation is correct under the C++ memory model

Practically efficient:

Comparison to 6 other general schemes
- LFRC, HP, ER, NER, QSR, DEBRA

on 4 different architectures with quite a lot of hardware threads (48-core AMP Opteron, 160-thread Intel Xeon, 244-tread Intel Xeon Phi, 512-thread SPARC)

Different benchmarks, various criteria (speed, reclamation efficiency)

Stamp-it performs on par with, or (considerably) better in almost all cases

Details:

Manuel Pöter, Jesper Larsson Träff: Brief Announcement: Stamp-it, a more Thread-efficient, Concurrent Memory Reclamation Scheme in the C++ Memory Model. SPAA 2018: 355-358
Manuel Pöter, Jesper Larsson Träff: Stamp-it, amortized constant-time memory reclamation in comparison to five other schemes. PPOPP 2018: 413-414
Manuel Pöter, Jesper Larsson Träff: Stamp-It: A more Thread-efficient, Concurrent Memory Reclamation Scheme in the C++ Memory Model. CoRR abs/1805.08639 (2018)

Manuel Pöter: Effective Memory Reclamation for Lock-free Data Structures in C++. Master's thesis, TU Wien (2018)

Full code and benchmarks: https://github.com/mpoeter/emr

Advanced Multiprocessor Programming

Jesper Larsson Träff
traff@par.tuwien.ac.at
Research Group Parallel Computing
Faculty of Informatics, Institute of Computer Engineering
Vienna University of Technology (TU Wien)

©Jesper Larsson Träff

Abstract "pool" data stuctures

Pool supports operations on collections of items (that can appear more than once):

Insert (or: set): Put an item into the pool
Delete (or: get): Retrieve an item from the pool

Contains: Check if some item is in pool (not necessarily supported)

Note: Empty? (pool empty?) predicate not considered essential

Pool variants and properties

- Delete on an empty pool returns no element (exception, special value or behavior)
- Bounded vs. unbounded. A bounded pool has an associated capacity, and cannot contain more items than the capacity
- Partial vs. total. A total pool method always succeeds (returns value, or throws exception), a partial pool method may have to wait for some condition (non-empty, or non-full), by definition blocking

- Fairness: pool items are deleted according to some criterion

Queue: first-in, first-out fairness (FIFO)
Stack: last-in, first-out fairness (LIFO)
Priority Queue: fairness according to some order on item keys

Concurrent FIFO Queues (Chap. 10)

Instance of a pool with FIFO fairness. Items are maintained in order of insertion, the oldest item is returned on delete. Often the oldest and the most recent item are referred to as tail and head, respectively (aka: double-ended queue, deque)

Queue operations:

- enq(x): add item to tail of queue
- y = deq(): return item from head of queue
- (but no operations on specific items, e.g. del(item))

(Semi-formal) Sequential semantics: State of queue described by (ordered) sequence of elements in queue

Our implementation: Linked list of items, plus sentinel element

Sentinel: Logically invisible place-holder item (blue), avoids many special cases; here, sentinel is not a fixed element

Head points to sentinel element. Queue is empty when sentinel has no successor (head.next=NULL)

Invariant: Item in queue, iff reachable from head

©Jesper Larsson Träff

enq(x)

©Jesper Larsson Träff

enq(x)

©Jesper Larsson Träff

y = deq()

y = head.next.value;

©Jesper Larsson Träff

y = deq()

y = head.next.value;

Memory management:
Recycle old sentinel

©Jesper Larsson Träff

Concurrent queues

Enqueuers (threads) and dequeuers can work concurrently and independently, synchronization between enqueuers and dequeuers only needed when queue is close to empty (or full)

Example 1:
Bounded, partial queue with fine-grained locking

Use condition variables:
A thread having a lock may release the lock and wait until woken up by other thread; when woken up, it again has the lock (as the sole lock owner: no violation of CS)

`pthreads`: mutex and condition variables

Java implementation:

Queue class maintains capacity and current size of queue

enq(x): if queue is full (size==capacity) wait until other thread dequeues, insert at tail of list, increment size and wakeup waiting dequeuers if needed

deq(): if queue is empty wait until other thread enqueues, delete from head, decrement size and wakeup waiting enqueuers if needed

Linked list implementation. List could/should be replaced with circular array (head and tail indices, head=array[headidx%size])

```
class BoundedQueue {
  ReentrantLock enqlock, deqlock;
  Condition notEmpty, notFull;
  AtomicInt size; // current size of queue
  int capacity;
  volatile Node head, tail;
  public BoundedQueue (int C) {
    capacity = C; // set capacity
    head = new Node(null);
    tail = head;
    size = new AtomicInt(0);
    enqlock = new ReentrantLock();
    deqlock = new ReentrantLock();
    notFull = enqlock.newCondition();
    notEmpty = deqlock.newCondition();
  }
}
```

Condition variables

```
public void enq(T x) {
  boolean wakeup;
  enqlock.lock();
  try {
    while (size.get()==capacity) notFull.await();

    Node e = new Node(x);
    tail.next = e; tail = e;
    if (size.getAndIncrement()==0) wakeup = true;
  } finally enqlock.unlock();

  if (wakeup) {
    deqlock.lock();
    try {
      notEmpty.signalAll();
    } finally deqlock.unlock();
  }
}
```

Condition associated with lock: Acquire

```
public void deq() {
  T x;
  boolean wakeup;
  deqlock.lock();
  try {
    while (size.get()==0) notEmpty.await();
    x = head.next.value; head = head.next;
    if (size.getAndDecrement()==capacity) wakeup = true;
  } finally deqlock.unlock();

  if (wakeup) {
    enqlock.lock();
    try {
      notFull.signallAll();
    } finally enqlock.unlock();
  }
  return x;
}
```


©Jesper Larsson Träff

Condition variable reminder:
When woken up, always recheck condition. It could have changed if wakeup is not performed in CS, on spurious wakeups, and if more threads are woken up

This implementation:
Contention on atomic size variable between enqueuers and dequeuers; can be remedied (idea: two counters, maintain upper/lower bound on number of elements)

Note:
Abstract queue's head and tail not always equals `head` and `tail`. The linearization point of enq(x) is the point where tail.next is set to the new item (not the point where tail is updated)

Synchronization constructs (reminder)

- Lock with condition variables (Java, pthreads, C++ threads,…)
- Monitor: Language construct, additional structure, safety guarantees that can be ensured by compiler
- Semaphore (binary, counting)

All three constructs equally powerful, either can be emulated in terms of the others

Counting semaphores are no more powerful than binary semaphores

Example 2:
Unbounded, total queue with (not so) fine-grained locking

- Liveness: Properties inherited from lock implementation, e.g. starvation free
- Progress: Blocking
- Correctness: Linearizable (enq(x) linearizes at the point where tail.next is set; deq() linearizes when head is updated to head.next)

```
public void enq(T x) {
  enqlock.lock();
  try {
    Node e = new Node(x);
    tail.next = e; tail = e;
  } finally enqlock.unlock();
}
```

```
public void deq() {
  T x;
  deqlock.lock();
  try {
    if (head.next==null)
      throw new EmptyException();
    x = head.next.value; head = head.next;
  } finally deqlock.unlock();
  return x;
}
```

Example 3:
Unbounded, total, lock-free queue

Idea:
Use CAS (Java: compareAndSet) to update list (pointers in C++)

```
class Node {
  public T value;
  public AtomicReference<Node> next;
  public Node(T x) {
    this.value = x;
    next = new AtomicReference<Node>(null);
  }
}
```

Difficulties:

- enq(x) consists of two operations
  - Updating next-field of tail
  - Updating tail
- Other threads may encounter unfinished updates

Solution: Make lazy with helper scheme

enq(x)

Try to update tail.next to x with CAS. Condition: tail.next must be **null**

©Jesper Larsson Träff

enq(x)

Try to update tail.next to x with CAS. Condition: tail.next must be **null**

Update tail to x with CAS

©Jesper Larsson Träff

enq(x)

Try to update tail.next to x with CAS. Condition: tail.next must be **null**

If not, some other thread is performing a concurrent enq, help by advancing tail with CAS, try again

©Jesper Larsson Träff

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


©Jesper Larsson Träff

y = deq()

If queue is empty (head==tail, and no item reachable from head), throw exception

If there is an item reachable from head (by concurrent enq), help by advancing tail with CAS

©Jesper Larsson Träff

y = deq()

Otherwise, return head.next, if head can be correctly advanced by CAS

©Jesper Larsson Träff

y = deq()

Otherwise, return head.next, if head can be correctly advanced by CAS

©Jesper Larsson Träff

```
public void deq() {
  while (true) {
    Node first = head.get();
    Node last  = tail.get();
    Node next  = first.next.get();
    if (first==last) {
      if (next==null) throw new EmptyException();
      tail.compareAndSet(last,next); // help
    } else {
      T x = next.value;
      if (head.compareAndSet(first,next)) return x;
    }
  }
}
```

The need for help

1. Enqueuer has succeed in setting tail.next
2. Dequeuer updates head

3. Tail would refer to a removed node

©Jesper Larsson Träff

Properties of lock-free queue

- Linearizable
  - enq: when CAS setting of next field succeeds
  - deq: when CAS updating of head succeeds
- Lock-free, but not wait-free (if CAS fails, some other thread must have succeeded)

Check:
Without helper scheme, enq would not be lock-free (other enq'er could go to sleep before updating tail, tail would not advance)

©Jesper Larsson Träff

The ABA problem

Instead of giving old sentinel nodes back to memory allocator, they could be explicitly reused:

Keep thread-local free-lists, on deq link old sentinel node into free-list, on enq get free node from free-list
For load balance: Threads could steal free nodes from other nodes' free-lists…

Implicit garbage collection: Nice, but
- Expensive
- Not in every programming language (e.g., C++)
- Rarely (never?) lock- or wait-free

Recycling problem (example):

1. Thread A: reads head (==a) and head.next (==b), falls asleep just before CAS
2. Thread B: deq (b), CAS succeeds
3. Thread C: deq (c), CAS succeeds

4. Nodes a and b are put in free-lists

©Jesper Larsson Träff

Recycling problem (example):

1. Thread A: reads head (==a) and head.next (==b), falls asleep just before CAS
2. Thread B: deq (b), CAS succeeds
3. Thread C: deq (c), CAS succeeds

4. Nodes a and b are put in free-lists

©Jesper Larsson Träff

Recycling problem (example):

5. Node a is recycled and enqueued again, a new node d is enqueued
6. Node c is dequeued

7. Thread A wakes up: CAS on head (again a) succeeds (== a, but obsolete) sets head to head.next (==b)

Head now refers to a node not in the queue, but in the free-list of thread B (or somewhere else): Disaster!

The ABA problem:

- A value changes from a to b and back to a
- A CAS operations expects a, succeeds, although the a may no longer refer to the same entity (different semantics, different content)

Often a problem with compare and swap (CAS): Only a reference/pointer is compared, not what is referenced/pointed to

Put differently:
CAS is used to ensure that state (of the data structure) has not changed, but state is represented by a single word, and this word may have been changed back (A->B->A) even though state has changed

Note:
LL/SC (load-linked/store conditional) atomics does not have this problem (and may therefore be better for some algorithms).

SC fails if there has been a change on the address since LL.

But, LL/SC is not so commonly supported as CAS (exception: IBM Power; Risc-V)

Remedies:
- Multi-word CAS
- Transactional memory

Solution (hack) with time stamps

- Encode a time-stamp with the reference/pointer, steal some bits
- Increment time-stamp with every reference

Obvious problem:
What if there is not enough bits to steal from the pointer?

Need multi-word compare-and-swap (n-CAS, k-CAS)

Java: `AtomicStampedReference<T>` class

```
public boolean compareAndSet(T expectedReference,
                             T newReference,
                             int expectedStamp,
                             int newStamp)
public T get(int stamp[]);
```

C++:
Implementation by hand, steal upper bits from 64-bit pointer

x86 architecture:
64-bit addresses, currently only lower 48 bits used; 16 bits available for marks

```
public void deq() {
  int[] = lastX new int[];  // time stamps
  int[] = firstX new int[];
  int[] = nextX new int[];
  while (true) {
    Node first = head.get(firstX);
    Node last  = tail.get(lastX);
    Node next  = first.next.get(nextX);
    if (first==last) {
      if (next==null) throw new EmptyException();
      tail.compareAndSet(last,next,
                         lastX[0],lastX[0]+1);
    } else {
      T x = next.value;
      if (head.compareAndSet(first,next,
                             firstX[0],firstX[0]+1))
        return x;
    }
  }
}
```

Herlihy-Shavit book adopted the lock-free queue from

Maged M. Michael, Michael L. Scott: Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. PODC 1996: 267-275

©Jesper Larsson Träff

Lock-free CAS (compare and swap) algorithms

CAS loop:

1. CAS to check that pointer (state) is as expected, update atomically
2. Retry on CAS failure

Lock-free if CAS failure means that other, competing thread has updated correctly and finished loop

On high contention, progress may be limited to a single/a few threads.
Possible remedy: Use atomic operation that always succeeds: FAA (fetch-and-add)

Adam Morrison, Yehuda Afek: Fast concurrent queues for x86 processors. PPOPP 2013: 103-112

Some work delegated to FAA, but also uses a double-word CAS (available on x86)

Chaoran Yang, John M. Mellor-Crummey: A wait-free queue as fast as fetch-and-add. PPOPP 2016: 16:1-16:13

Work-stealing dequeue: A special-purpose lock-free queue

Applications: Load-balancing by work-stealing (later), free-lists.

There is a collection of independent tasks to be executed:

- Each thread maintains own queue of tasks
- New tasks generated by thread are put at the bottom of queue
- When thread needs task, it is taken from the bottom
- If thread local queue empty, steal task from top of other queue

Asymmetric data structure:
Local access to bottom only, frequent; less frequent concurrent steal access to top; steal may spuriously fail (return null even if an item was present)

Operations:

push_bottom(x): Add element to bottom, only one thread (owner)
y = pop_bottom(): Remove element from bottom, only one thread
z = pop_top(): Remove element from top, many threads (thieves)

Further observations:

- Work-stealing queue is typically well-filled, no conflict between local and steal accesses. No atomics needed for local access, CAS for steal access
- One element remaining: both local access and steal must CAS

Unbounded array implementation

©Jesper Larsson Träff

Unbounded array implementation

©Jesper Larsson Träff

Lock-free, unbounded array implementation (in pseudo-C++)

`atomic int bottom`: next free slot in array from bottom
`atomic int top`: current top element in array (only increasing)

Initially, `bottom = 0; top = 0;`

Dequeue is empty when `top>=bottom`

```
void push_bottom(T item) {
  data[bottom++] = item;
}

T pop_bottom() {
  if (bottom==top) return null_value<T>;

  T value = data[--bottom];
  int t = top;
  if (bottom>t) return value;

  if (bottom==t) {
    if (compare_exchange(top,t,t) return value;
    bottom++; // restore
  } else bottom = t;
  return null_value<T>;
}
```


©Jesper Larsson Träff

```
T pop_top() {
  int t = top;
  if (bottom<=t) return null_value<T>;
  T value = data[t];

  if (compare_exchange(top,t,t+1) return value;
  else return null_value<T>;
}
```


©Jesper Larsson Träff

B: pop_top()

1. Reads top
2. Reads data
3. CAS succeeds
4. Return value

A: pop_top()

1. Reads top
2. Reads data
3. CAS fails
4. Return null

Spurious fail: The dequeue has plenty of elements

©Jesper Larsson Träff

B: pop_bottom()

1. Reads data
2. Reads top
3. CAS fails
4. Restore bottom
5. Return null

A: pop_top()

1. Reads top
2. Reads data
3. CAS succeeds
4. Return value

©Jesper Larsson Träff

B: pop_bottom()	A: pop_top()
1. Reads data	1. Reads top
2. Reads top	2. Reads data
3. CAS succeeds	3. CAS succeeds
4. Return value	4. Return value

Wrong:
ABA problem! top has been updated by CAS (because bottom changed), but value has not changed

©Jesper Larsson Träff

Properties:

- Suffers from ABA problem. Standard repair: Time stamps
- Local pop_bottom(): Decrement bottom, read top; when only one element remaining, CAS on top – fails if other thread modified top, but that means queue empty
- Steal pop_top(): Read top, read bottom, CAS on top. Only one thread may succeed before a decrement to bottom becomes visible

Linearization:

- push_bottom(): Increment of bottom
- pop_bottom():
  - Decrement of bottom, if top<bottom
  - CAS, if top==bottom
- pop_top(): CAS on top

Progress and liveness:

- push_bottom(): Wait-free
- pop_bottom():
  - wait-free, if top<bottom
  - if top==bottom: wait-free whether CAS fails or succeeds
- pop_top(): Wait-free, whether CAS fails or succeeds, but may fail spuriously (stealing therefore becomes lock-free)

Drawbacks:

1. Unbounded array!
2. Ignores ABA problem
3. (pop_top() may fail spuriously)

Nimar S. Arora, Robert D. Blumofe, C. Greg Plaxton: Thread Scheduling for Multiprogrammed Multiprocessors. Theory Comput. Syst. 34(2): 115-144 (2001)

Drawbacks. Fixes

1. Use array of size N, index with (bottom%N) and (top%N), when full either do not push_bottom(), or extend array. Recall array lock
2. Bounded time stamp, or hack
3. Not a problem for work-stealing context, could also try pop_top() some fixed number of times before selecting new victim

Nimar S. Arora, Robert D. Blumofe, C. Greg Plaxton: Thread Scheduling for Multiprogrammed Multiprocessors. Theory Comput. Syst. 34(2): 115-144 (2001)

David Chase, Yossi Lev: Dynamic circular work-stealing deque. SPAA 2005: 21-28

Bit stealing hack: Steal k (=8, =16?) bits from address

- Increment "time-stamp" on pop_bottom()
- Reset "time stamp" on pop_top()

```
T pop_bottom() {
  if (bottom==index_bits(top)) return null_value<T>;

  T value = data[--bottom];
  if (bottom>index_bits(top)) return value;

  int t = top;
  int new_t = inc_mark_bits(t);
  if (bottom==index_bits(t)) {
    if (compare_exchange(top,t,new_t) return value;
    bottom++;
  } else bottom = index_bits(t);
  return null_value<T>;
}
```

```
T pop_top() {
  int t = top;
  if (bottom<=index_bits(top)) return null_value<T>;
  T value = data[t];

  int new_t = index_bits(t); // reset "time stamp"
  if (compare_exchange(top,t,new_t+1) return value;
  else return null_value<T>;
}
```

```
#define index_bits(a) (a & 0xFFFFFF)
#define inc_mark_bits(a) \
((((a>>48)+1) & 0xFF)<<48) | index_bits(a))
```

Wait-free queues (lists, stacks)

Some form of help needed to make list and queues (and stacks) wait-free (instead of only lock-free).

Alex Kogan, Erez Petrank: Wait-free queues with multiple enqueuers and dequeuers. PPOPP 2011: 223-234
Chaoran Yang, John Mellor-Crummey: A wait-free queue as fast as fetch-and-add. PPOPP 2016: 16:1-16:13
Shahar Timnat, Anastasia Braginsky, Alex Kogan, Erez Petrank: Wait-Free Linked-Lists. OPODIS 2012: 330-344

Seep Goel, Pooja Aggarwal, Smruti R. Sarangi: A Wait-Free Stack. ICDCIT 2016: 43-55

Concurrent LIFO stacks (Chap. 11)

Stack is a pool type with last-in, first-out (LIFO) fairness semantics

Operations:

- push(x)
- x = pop()

The item returned by a pop() is the most recently pushed item

Implementation: Linked list with top reference/pointer

©Jesper Larsson Träff

```
class Node {
  public T value;
  public Node next;
  public Node (T x) {
    value = x;
    next = null;
  }
}
```

©Jesper Larsson Träff

Implementation:

- push(x): create new stack node, try to push with CAS. If CAS fails, there is contention on top, back off and try again
- y = pop(): remove top node with CAS. If CAS fails, there is contention on top, back off and try again

```
class LockFreeStack {
  AtomicReference<Node> top = new …
  static final int MIN_DELAY, MAX_DELAY;
  Backoff backoff =
    new Backoff(MIN_DELAY,MAX_DELAY);

  public void push(T x);

  public T pop();
}
```

Values for Backoff class, recall lock lecture

©Jesper Larsson Träff

```
protected boolean try_push(Node node) {
  Node oldtop = top.get();
  node.next = oldtop;
  return top.compareAndSet(oldtop,node);
}

public void push(T x) {
  Node node = new Node(x);
  while (true) {
    if (try_push(node)) return;
    else backoff.backoff();
  }
}
```


©Jesper Larsson Träff

```
protected boolean try_pop()
  throws EmptyException {
  Node oldtop = top.get();
  if (oldtop==null) throw new EmptyException();
  Node newtop = oldtop.next;
  if (top.compareAndSet(oldtop,newtop))
    return oldtop;
  else return null;
}

public T pop() throws EmptyException {
  while (true) {
    Node node = try_pop();
    if (node!=null) return node.value;
    else backoff.backoff();
  }
}
```

Properties of lock-free stack

- Linearizable
  - push: successful CAS on top
  - pop: successful CAS on top, or read of null top-reference

- Lock-free, but not starvation free (if CAS fails, some other thread must have succeeded)

The Java implementation, relying on garbage collection does not have an ABA problem:
The push/pop interface passes values only (items), nodes are maintained (allocated) internally. The same address thus cannot reappear at one thread when some other threads references this address

If pop'ed items can be reused, the stack has an ABA problem

Thread A: pop(), newtop=b

Thread A delayed

Thread B: pop(), pop()

©Jesper Larsson Träff

Thread A: pop(), newtop=b

Thread B: pop(), pop(), push(a)

©Jesper Larsson Träff

Thread A: pop(): newtop=b, CAS succeeds

Thread B: pop(), pop(), push(a)

©Jesper Larsson Träff

Repair (in C): Claim some bits for time stamp

```
void *pop() {
  void *oldtop;
  do {
    oldtop = top;
    if (address_bits(oldtop)==NULL) return NULL;
    newtop = oldtop->next;
    set_mark_bits(newtop,get_mark_bits(oldtop)+1);
  while (!compare_exchange(top,oldtop,newtop);
  return oldtop;
}
```

Observation:
It suffices to increment the timestamp only on pop() (or on push())

©Jesper Larsson Träff

Repair (in C): Claim some bits for time stamp

```
void push(void *newtop) {
  void *oldtop;
  do {
    oldtop = top;
    newtop->next = address_bits(oldtop);
    set_mark_bits(newtop,get_mark_bits(oldtop));
  while (!compare_exchange(top,oldtop,newtop);
  return oldtop;
}
```

R. Kent Treiber: Systems Programming: Coping with Parallelism. Technical Report 5118, IBM Almaden Research Center, 1986

Problem with lock-free stack (and queue)

Access to top reference/pointer is a sequential bottleneck (even worse than the double-ended queue), all threads compete for the top of the stack, linearization enforces serialization

Possible to improve?

Highly sensitive to back off tuning

Stack linearization(*)

(*) Example due to Martin Wimmer

Stack linearization

©Jesper Larsson Träff

Stack linearization: Could have happened like this

©Jesper Larsson Träff

Stack linearization: But also differently…

©Jesper Larsson Träff

Stack linearization: …with these real-time operations

©Jesper Larsson Träff

Can this history be linearized?

Yes: Two concurrent push and pop operations can be paired

©Jesper Larsson Träff

Can this history be linearized?

Yes: Two concurrent push and pop operations can be paired

©Jesper Larsson Träff

New idea: Elimination (backoff stack)

A push can be cancelled by an overlapping pop; this pair of operations do not have to access the stack at all, and are said to eliminate each other

Use elimination array (size: tuning parameter):

- push(x): choose random index, put x into index, wait for pop, try stack on time-out
- pop(): choose random index, wait for item at index, try stack on time-out

Successful elimination:
A and B pairs up

Unsuccessful elimination:
- Either or both C, D to stack
- E to stack

©Jesper Larsson Träff

Linearization with elimination

Linearization via elimination array

Linearization via elimination array

Linearization via lock-free stack

©Jesper Larsson Träff

Linearization with elimination (different history)

Linearization via elimination array

Linearization via elimination array

Linearization via lock-free stack

©Jesper Larsson Träff

Implementation (straightforward, but tedious):

Elimination array of socalled exchangers each with a slot that allows exchange of some value between two threads using CAS.
Then:

- Try stack first
- If unsuccessful, try elimination
- Backoff/time-out
- Repeat until success

Highly sensitive to tuning!

- Size of the elimination array (too large: no elimination; too small: to many conflicts)
- Back-off
- …

```
class LockFreeExchanger<T> {
  static final int EMPTY=…, WAITING=…, BUSY=…;
  AtomicStampedReference<T> slot =
    new AtomicStampedReference<T>(null,EMPTY);

  public exchange(T item, long timeout, TimeUnit unit)
  throws Timeoutexception {
    long timebound =
      System.nanotime()+unit.toNanos(timeout)
    int[] stamp = {EMPTY};

    while (true) {
      // try to exchange until timeout
      … (next slide)
    }
  }
}
```

```
// try to exchange until timeout
if (System.nanoTime()>timebound)
  throw new Timeoutexception();
  // timed out, no exchange
… (next slide)
```

Thread tries to grab slot with CAS. Three cases:

1. Slot is EMPTY, no one to exchange with, thread becomes WAITING until either timeout (change back to EMPTY) or someone shows up
2. Someone else is WAITING, change to BUSY and let other thread exchange
3. Already BUSY, slot cannot be used

```
T otheritem = slot.get(stamp); // read the slot
switch (stamp[0]) {
case EMPTY:
  if (slot.compareAndSet(otheritem,item,
                         EMPTY,WAITING)) {
    while (System.nanoTime()<timebound) {
      otheritem = slot.get(stamp);
      if (stamp[0]==BUSY) {
        slot.set(null,EMPTY);
        return otheritem;
      }
    }
  }
  if (slot.compareAndSet(item,null,WAITING,EMPTY))
    throw new TimeoutException();
  else {
    otheritem = slot.get(stamp);
    slot.set(null,EMPTY);
    return otheritem;
  }
```

```
case EMPTY:
  … (previous slide)
break;
case WAITING:
  if (slot.compareAndSet(otheritem,item,
                         WAITING,BUSY))
    return otheritem;
  break;
case BUSY:
  break;
default: // cannot be
}
```

```
class EliminationArray<T> {
  static final int duration=…; // tuning parameter
  LockFreeExchanger<T>[] exchanger;
  Random random;
  public EliminationArray(int capacity) {
    exchanger = (LockFreeExchanger<T>[])
      new LockFreeExchanger<T>();
    for (int i=0; i<capacity; i++)
      new LockFreeExchanger<T>();
    random = new Random();
  }
  public T visit(T value, int range)
  throws TimeoutException {
    int slot = random.nextInt(range);
    return (exchanger[slot].exchange(value,duration,
                                     TimeUnit.MILLISECONDS));
  }
}
```


©Jesper Larsson Träff

```
public class EliminationBackoffStack<T> extends
LockFreeStack<T> {
  static final int capacity = …; //maximum capacity
  static final int range = … ; // here: fixed range
  // book provides for adaptive range selection
  EliminationArray<T> eliminationarray =
    new EliminationArray<T>(capacity);

  public void push(T value);

  public T pop();
}
```

```
public void push(T x) {
  Node node = new Node(x);
  while (true) {
    if (try_push(node)) return;
    else try {
      T y = eliminationarray.visit(x,range);
      if (y==null) return; // successful elimination
    }
  }
}
```

On unsuccessful elimination (timeout): catch the exception, perhaps use this to adjust the range (book provides for such policy change). Also update range/policy on successful elimination

```
public T pop() throws EmptyException {
  while (true) {
    Node node = try_pop();
    if (node!=null) return node.value;
    else try {
      T y = eliminationarray.visit(null,range);
      if (y!=null)
        return y; // successful elimination
    }
  }
}
```

On unsuccessful elimination (timeout): catch the exception, perhaps use this to adjust the range (book provides for such policy change). Also update range/policy on successful elimination

Linearization:

Any successful push()/pop() that completes via the lock-free stack, linearizes at the successful CAS in the stack

Any pair of eliminated push()/pop() operations, linearize at their elimination point. Such a pair leaves the stack unchanged, and can linearize anywhere within their concurrent invocations, pop before push (regardless of other, concurrent push() and/or pop() operations).

All push()/pop() operations complete either by successful elimination or via the lock free stack

Other uses of exchangers and elimination

William N. Scherer III, Doug Lea, Michael L. Scott: Scalable synchronous queues. Commun. ACM 52(5): 100-111 (2009) (*)

Irina Calciu, Hammurabi Mendes, Maurice Herlihy: The Adaptive Priority Queue with Elimination and Combining. DISC 2014: 406-420

Anastasia Braginsky, Nachshon Cohen, Erez Petrank: CBPQ: High Performance Lock-Free Priority Queue. Euro-Par 2016: 460-474

(*) Announced at PPoPP 2006

Are queues and stacks good, concurrent data structures?

Queues and stacks are sequential data structures… strong ordering/fairness guarantees (FIFO, LIFO, Priority, …)

Leads to serialization on concurrent updates

⇒ Little potential for parallelization, little that can be done locally by individual threads

Problem:
Linearization enforces sequential order. Is this too strict?

Relaxed linearization

Some applications do not require exact ordering:

- Element close to the top good enough (relaxed stack)
- k'th smallest element (Priority queue)
- …

As long as some bound on the deviation from a linearizable order can be given

Yehuda Afek, Guy Korland, Eitan Yanovsky: Quasi-Linearizability: Relaxed Consistency for Improved Concurrency. OPODIS 2010: 395-410

…

Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, Philippas Tsigas: The lock-free k-LSM relaxed priority queue. PPOPP 2015: 277-278

Some applications do not require exact ordering:

- Quiescent consistency gives more freedom for concurrent operations (might be easier to guarantee, see later)

But no "fairness" guarantees

Often used relaxed PQ example(s):

- SSSP a la Dijkstra is robust, an "incorrect" min element does not harm correctness, but may lead to unnecessary, duplicate, superfluous (speculative) work
- Concurrent tree search, e.g., branch-and-bound

Skiplist (Chap. 14)

Efficient, randomized, list-based search structure (abstract set)

- O(log n) expected operations for n element skiplist
  - add(x),
  - y = remove()
  - contains(z)?
- O(n) space
- Compared to tree-based search structures: No need for rebalancing

William Pugh: Skip Lists: A Probabilistic Alternative to Balanced Trees. Comm. ACM 33(6): 668-676 (1990)

Ordered list (with sentinels at beginning and end), linear time

©Jesper Larsson Träff

Skiplist: Multiple lists for fast navigation

©Jesper Larsson Träff

Skiplist properties and invariants

- Collection of ordered lists, organized in levels 0, 1, …, MAX_LEVEL
- List at bottom level 0 represents set
- Ceiling(log n) levels for n element set
- Sublist property: List at level l, l>0, sublist of list at level l-1 (list at level l is a shortcut into the list at level l-1)
- Number of elements at level l is expected some constant fraction 1/f (often: f=2) of number of elements at level l-1
- Probability that an element appears at level l is $(1/f)^l$
- Expected space consumption is
  - $O(\sum_{0 \le l < MAX_LEVEL} n\,(1/f)^l) = O(n)$

Why?

For the following, we do not distinguish between item and key

Skiplist operations, implementation

- find(x) and contains(x): Start at current maximum level l, search for elements pred and curr (== pred.next) such that pred.item<x≤curr.item, decrease l; until l==0. Found iff curr.item==x
- add(y): if not find(x), create new node with random number of levels k, insert in lists at all levels from 0 to k
- remove(z): if find(x), link out x from all lists from level 0 to level of x

All operations are O(log n), if the skiplist properties are fulfilled

Why?

Example: find(14)

Set
preds[3] = pred; succs[3] = curr;

©Jesper Larsson Träff

Example: find(14)

Set
preds[3] = pred; succs[3] = curr;
preds[2] = pred; succs[2] = curr;

©Jesper Larsson Träff

Example: find(14)

Set
preds[3] = pred; succs[3] = curr;
preds[2] = pred; succs[2] = curr;
preds[1] = pred; succs[1] = curr;

©Jesper Larsson Träff

Example: find(14)

Set
preds[3] = pred; succs[3] = curr;
preds[2] = pred; succs[2] = curr;
preds[1] = pred; succs[1] = curr;
preds[0] = pred; succs[0] = curr;

©Jesper Larsson Träff

Example: add(14)

preds[3] = pred; succs[3] = curr;
preds[2] = pred; succs[2] = curr;
preds[1] = pred; succs[1] = curr;
preds[0] = pred; succs[0] = curr;

©Jesper Larsson Träff

Example: add(14)

preds[3] = pred; succs[3] = curr;
preds[2] = pred; succs[2] = curr;
preds[1] = pred; succs[1] = curr;
preds[0] = pred; succs[0] = curr;

©Jesper Larsson Träff

Lazy Skiplist

Idea:
Fine-grained locking, wait-free contains operation, remove elements lazily by setting marked flag: Each level is essentially as in the lazy list-based set algorithm.

To maintain skiplist property, insertion of new node is done by linking in from level 0 up to maximum level of new node, at deletion, links must be removed from maximum level of node down to 0

To maintain skiplist property, insertion of new node is done by linking in from level 0 up to maximum level of new node, at deletion, links must be removed from maximum level of node down to 0

In order have a wait-free contains() operation, locking only in add() and remove() operations, no lock in find().

Requires lock per node (note: reentrant/recursive lock convenient), marked flag per node, fully linked flag per node

Invariant:
An element is in the skiplist iff it is unmarked and fully linked

The Shavit-Herlihy implementation

A (new) node in the skiplist with k levels is represented by an array of k next pointers

find(14) method:
Builds node arrays preds[] and succs[] with preds[l].item<14 and 14≤succs[l].item, for all l; returns first foundlevel from top such that item==succs[foundlevel].item

find(x): found if x==succs[foundlevel].item

©Jesper Larsson Träff

add(x):

1. find(x)
2. If not marked, wait for fully linked
3. Lock predecessors (but only up to level of new element x) from bottom to top
4. Validate predecessors
5. Link in from bottom to top
6. Set fully linked flag

remove(y):

1. find(y)
2. Lock node
3. Set marked flag
4. Lock predecessors (only up to level of y), from bottom to top
5. Validate predecessors
6. Link out from top to bottom

```
private static final class Node<T> {
  final Lock lock = new ReentrantLock();
  final T item; // use as key… (sloppy)
  final Node<T>[] next;
  volatile boolean marked = false;
  volatile boolean fullylinked = false;
  private int k; // lastlevel for node

  public Node(T x, int levels) {
    item = x;
    next = new Node[levels+1];
    k = levels;
  }
  …
}
```

Reentrant (recursive) locks are needed. Why? See next slides…

Note:

- Next references are to Node's
- Locks are per node
- Flags (marked, fullylinked) are per node, not per level

k: last level for node

Reentrant (recursive) locks are needed (lock is per node)

```
int find(T item, Node<T>[] preds, Node<T>[] succs) {
  int foundlevel = -1; // not found at any level≥0
  Node<T> pred = head;
  for (int l = MAX_LEVEL; l>=0; l--) {
    volatile Node<T> curr = pred.next[l];
    while (item>curr.item) {
      pred = curr; curr = curr.next[l];
    }
    if (foundlevel==-1&&item==curr.item)
      foundlevel = l; // found at level l
    preds[l] = pred; succs[l] = curr;
  }
  return foundlevel;
}
```

```
int contains(T item) {
  Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL+1];
  Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL+1];

  int foundlevel = find(x,preds,succs);
  return (foundlevel>=0 &&
          succs[foundlevel].fullylinked &&
          !succs[foundlevel].marked);
}
```

contains(x) is wait-free: No locks, no spinning

Example: remove(5)

©Jesper Larsson Träff

Example: remove(5)

"victim" for removal: check unmarked and fully linked
lock victim, mark, lock predecessors (k=0 for this victim)

©Jesper Larsson Träff

Example: remove(5)

"victim" for removal: check unmarked and fully linked
lock victim, mark, lock predecessors (k=0 for this victim)

©Jesper Larsson Träff

Example: remove(5) successful

"victim" for removal: check unmarked and fully linked
lock victim, mark, lock predecessors
unlink, unlock

©Jesper Larsson Träff

```
boolean remove(T y) {
  Node<T> victim = null;
  boolean marked = false; int k = -1;
  Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL+1];
  Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL+1];
  while (true) {
    int f = find(y,preds,succs);
    if (f>=0) victim = succs[f];
    if (marked||
        (f>=0&&victim.fullylinked&&victim.k==f&&
         !victim.marked)) {
      if (!marked) { // mark victim only once
        k = victim.k;
        victim.lock.lock();
        if (victim.marked) {
          victim.lock.unlock(); return false;
        }
        victim.marked = true; marked = true;
      }
      // now marked, try to link out … (next slide)
    } else return false;
} }
```

See next slide

?

©Jesper Larsson Träff

Check for `victim.k==f` is an optimization:

If `victim` was found at a level `f<victim.k`, either victim is not yet fully linked (by concurrent add()) or is concurrently being unlinked by concurrent remove() operation, and validation would fail anyway

©Jesper Larsson Träff

```
// try to link out (previous slide)
int highlock = -1;
try { // validate
  Node<T> pred, succ; boolean valid = true;
  for (l=0; valid&&(l<=k); l++) {
    pred = preds[l];
    pred.lock.lock();
    highlock = l;
    valid = !pred.marked&&pred.next[l]==victim;
  }
  if (!valid) continue;
  for (l=k; l>=0; l--)
    preds[l].next[l] = victim.next[l];
  victim.lock.unlock();
  return true;
} finally
  for (l=0; l<=highlock; l++) preds[l].lock.unlock();
```

Validation failed

Unlink downwards

©Jesper Larsson Träff

Example: remove(7)

"victim" for removal: check unmarked and fully linked
lock victim, mark, lock predecessors (k==2 for this victim)

©Jesper Larsson Träff

Example: remove(7)

"victim" for removal: check unmarked and fully linked
lock victim, mark, lock predecessors (k==2 for this victim)

©Jesper Larsson Träff

Example: remove(5), with concurrent add(40)

Find, lock and validate (predecessor and successor unmarked, pred.next==succ). Here: new element with k=0
Link in, bottom to top to maintain skiplist property

©Jesper Larsson Träff

Example: remove(5), with concurrent, unsuccessful add(17)

Assume 17 not yet fully linked: Wait (spin) for node to become fully linked (what will happen if not?)

©Jesper Larsson Träff

```
boolean add(T x) {
  int k = random_level(MAX_LEVELS);
  Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL+1];
  Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL+1];
  while (true) {
    int f = find(x,preds,succs);
    if (f>=0) {
      Node<T> Found = succs[f];
      if (!Found.marked)
        while (!Found.fullylinked) {} // spin
        return false;
      }
      continue;
    }
    // not in skiplist, try to add (next slide)
}
```

```
// try to add
highlock = -1; // highest level locked
try {
  Node<T> pred, succ;
  boolean valid = true;
  for (l=0; valid&&(l<=k); l++) { // validate
    pred = preds[l]; succ = succs[l];
    pred.lock.lock();
    highlock = l;
    valid =
     !pred.marked&&!succ.marked&&pred.next[l]==succ;
  }
  if (!valid) continue; // failed, retry
  Node<T> newNode = new Node(x,k); // allocate new
  for (l=0; l<=k; l++) { // link in updwards
    newNode.next[l] = succs[l];
    preds[l].next[l] = newNode;
  }
  newNode.fullylinked = true;
} finally
  for (l=0; l<highlock; l++) preds[l].lock.unlock();
```

Finding the right number of levels for new node

Number of levels for each node is chosen randomly, such that that the skiplist properties are preserved (constant fraction of nodes at next level)

```
int random_level(int maxlevel);
  int i = 0;
  for (i=0; i<maxlevel; i++)
    // choose random number in [0,1[ uniformly
    if (random(0,1)<1/f) break;
  return i;
}
```

With f=2, probability of k=0 is constant (1/2), probability that k=i, i≥1, is $2^{-(i+1)}$, probability that k is 31 is 2^{-32} (cf. Herlihy-Shavit)

Properties and invariants:

Locks (add and remove) always acquired in same order, bottom to top: no deadlock

- Maintains skiplist property: List at level l is sublist of list at level l-1, lists are ordered (add sets links from bottom to top)
- An element is in the skiplist iff it is reachable, fully linked (linked on all levels) and not marked

When add(x) finds unmarked, but not fully linked node, it must wait for node to become fully linked (here: spin) to make operation linearizable while maintaining the skiplist property

Non-linearizable history when not waiting for fully linked

Let add return false (no waiting)

Remove would wrongly return false

Item found at some level, but yet not fully linked

New node is fully linked

©Jesper Larsson Träff

Correctness: All operations are linearizable

Unsuccessful add(x): Check for unmarked and fully linked succeeds
Successful add(x): After linking in the new node and setting it to fully linked

Unsuccessful remove(y): The found node is already marked (being deleted by concurrent thread), or not found, or not fully linked
Successful remove(y): Marking the found node

Successful contains(z): Predecessors next reference unmarked and fully linked
Unsuccessful contains(z): Node not found, or found marked, or before linearization of concurrent add (see lazy list)

Liveness:

Contains obviously wait-free

Add and remove not starvation free (even if locks are)

Lock-free Skiplist

Idea: Lock free lists at each level; need to maintain reference/pointer AND mark as one unit

Java: Each next field is an `AtomicMarkableReference<Node>`

Note:
Can update lists only one level at a time (single-word CAS), therefore not possible to maintain skiplist property: List at level l will no longer be a sublist of list at level l-1

Instead:
List at level l is a shortcut into list at level l-1.
Property of abstract set: An element is in the set iff it is in the list at level 0

```
private static final class Node<T> {
  final T item; // use as key… (sloppy)
  final AtomicMarkableReference<Node<T>>[] next;
  private int k; // number of levels for node

  public Node(T x, int levels) {
    item = x;
    next = (AtomicMarkableReference<Node<T>>[])
      new AtomicMarkableReference[levels+1];
    k = levels;
    for (l=0; l<=k; l++) next[l] = new
      AtomicMarkableReference<Node<T>>(null,false);
  }
  …
}
```


©Jesper Larsson Träff

Note:

- No locks, no fully linked flag for the nodes.
- Each reference to a next node is a marked reference, mark indicates that node link is logically deleted and should be skipped/short-cut: Node item is not present at this level

©Jesper Larsson Träff

find(x, preds, succs):

As in lock-free list, must physically link out marked nodes at each level (with CAS: compareAndSet in Java). Returns only whether item was found or not

To make contains(z) wait-free: Jump over marked nodes, as in lock-free list

```
boolean find(T x, Node<T>[] preds, Node<T>[] succs) {
  int b = 0;
  boolean[] marked = {false};
  Node<T> pred = null, curr = null, succ = null;
  retry:
    while (true) {
      pred = head;
      for (l=MAX_LEVEL; l>=0; l--) { // go down levels
        curr = pred.next[l].getReference();
        while (true) {
          succ = curr.next[l].get(marked);
          while (marked[0]) { // link out marked nodes
            if (!pred.next[l].
                 compareAndSet(curr,succ,false,false))
              continue retry;
            curr = pred.next[l].getReference();
            succ = curr.next[l].get(marked);
          }
```


©Jesper Larsson Träff

```
            if (curr.item<x) {
              pred = curr; curr = succ;
            } else break;
          }
          preds[l] = pred; succs[l] = curr;
        }
        return (curr.item==x);
      }
    }
}
```

Linearization point when l==0

©Jesper Larsson Träff

```
boolean contains(T item) {
  int bottom = 0;
  boolean[] marked = {false};
  Node<T> pred = head; curr = null; succ = null;
  for (int l=MAX_LEVEL; l>=bottom; l--) {
    curr = pred.next[l].getReference();
    while (true) {
      succ = curr.next[l].get(marked);
      while (marked[0]) {
        curr = curr.next[l].getReference();
        succ = curr.next[l].get(marked);
      }
      if (curr.item<item) {
        pred = curr; curr = succ;
      } else break;
    }
  }
  return (curr.item==item);
}
```

(sometimes) linearization point when l==0

add(x):

find(x), link in with CAS, on failure restart

remove(y):

find(y), attempt to mark from top to bottom (setting mark atomically), restart on failure. Find again to clean up lists

©Jesper Larsson Träff

```
boolean add(T x) {
  int k = random MAX_LEVELS); int b = 0;
  Node<T>[] pred de<T>[]) new Node[MAX_LEVEL+1];
  Node<T>[] suc <T>[]) new Node[MAX_LEVEL+1];
  while (true) {
    if (find(x,preds,succs)) return false;
    else { // prepare new node
      Node<T> newNode = new Node(x,k);
      for (l=b; l<=k; l++) {
        Node<T> succ = succs[l];
        newNode.next[l].set(succ,false);
      }
      Node<T> pred = preds[b];
      Node<T> succ = succs[b];
      if (!pred.next[b].compareAndSet(succ,newNode,
                                      false,false))
        continue;
    }
    // remaining levels
  } }
```

Lineari zation on fail

References set early

Linearization on success

©Jesper Larsson Träff

```
// remaining levels
for (l=b+1; l<=k; l++) {
  while (true) {
    pred = preds[l]; succ = succs[l];
    if (pred.next[l].compareAndSet(succ,newNode,
                                   false,false))
      break;
    find(x,preds,succ);
  }
}
return true;
```


©Jesper Larsson Träff

```
boolean remove(T y) {
  int b = 0;
  Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL+1];
  Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL+1];
  while (true) {
    if (!find(y,preds,succs)) return false;
    else {
      // shortcut lists from k down to b+1
      Node<T> remNode = succs[b];
      for (l=remNode.k; l>=b+1; l--) {
        boolean marked[] = {false};
        succ = remNode.next[l].get(marked);
        while (!marked[0]) {
          remNode.next[l].compareAndSet(succ,succ,
                                        false,true);
          succ = remNode.next[l].get(marked);
        }
      }
      // level 0 list
} } }
```

Lineari
zation
on fail

©Jesper Larsson Träff

```
// level 0 list
boolean[] marked = {false};
succ = remNode.next[b].get(marked);
while (true) {
  boolean done =
    remNode.next[b].compareAndSet(succ,succ,
                                  false,true);
  succ = succs[b].next[b].get(marked);
  if (done) {
    find(y,preds,succs); // clean up (optimization)
    return true;
  } else if (marked[0]) return false;
}
```

Linearization on success

©Jesper Larsson Träff

```
// level 0 list
boolean[] marked = {false};
succ = remNode.next[b].get(marked);
while (true) {
  boolean done =
    remNode.next[b].compareAndSet(succ,succ,
                                  false,true);
  succ = succs[b].next[b].get(marked);
  if (done) {
    find(y,preds,succs); // clean
    return true;
  } else if (marked[0]) return fals
}
```

Linearization on success

Fail by concurrent remove

©Jesper Larsson Träff

Key points (I)

add(x):
New node is linked in starting from bottom level 0, if CAS fails predecessor has changed (marked or new node inserted concurrently), sublist property maintained. When new node has been added, repeat until it has been linked at all levels; may lead to violation of sublist property if new node is concurrently removed by other thread

remove(y):
If found, node is marked starting from highest level, success if marking succeeds at bottom level 0; if CAS failure, node has been concurrently linked out by other thread. Node physically removed (linked out) by following find operation

Key points (II)

contains(x):
At each level, skip marked links, makes sure that item in set (at level 0) is eventually found

©Jesper Larsson Träff

Example: add(7)

add(7): find

©Jesper Larsson Träff

Example: add(7)

add(7): find

©Jesper Larsson Träff

Example: add(7)

add(7): find

©Jesper Larsson Träff

Example: add(7)

add(7): find

©Jesper Larsson Träff

Example: add(7)

add(7): find

©Jesper Larsson Träff

Example: add(7)

©Jesper Larsson Träff

Example: add(7)

add(7): find, alloc, link

©Jesper Larsson Träff

Linearization point
of successful add

Example: add(7)

©Jesper Larsson Träff

Linearization point
of successful add

Example: add(7)

©Jesper Larsson Träff

Linearization point
of successful add

Example: add(7)

©Jesper Larsson Träff

Linearization point
of successful add

Example: add(7)

©Jesper Larsson Träff

Example: add(7)

©Jesper Larsson Träff

Example: add(7)

©Jesper Larsson Träff

Example: add(7)

add(7): find, alloc, link, level 0 | levels 1…,2

remove(7): find, mark levels k…, level 0 | find

Node 7 correctly unlinked at level 0, 1, but present (marked reference) at level 2

©Jesper Larsson Träff

Correctness: All operations are linearizable

Successful add(x): The moment the element is in the list at level 0: successful CAS
Unsuccessful add(x): Linearization point of find

Successful remove(y): The point where the node containing the element is marked (CAS)
Unsuccessful remove(y): Not found (linearization point of find), or concurrent remove sets marked

find(z): node with item>=z found at level 0

contains(z): on overlapping add(z) call, either point where node is linked in at level 0, or if not found, before other thread links node in

Liveness:

contains() obviously wait-free

add() and remove()

- Lock-free
- But not starvation free

©Jesper Larsson Träff

Lazy vs. lock-free skiplist

Skiplist data structure robust, it does not need the strict sublist property: Key observation for lock-free algorithm

Lock-free list in some way simpler than lazy, fine-grained locking based algorithm

Memory management: By garbage collection

Which performs better?

Another interesting algorithm for multi-linked data structures: Maintain property only for a subset of the links, other links may violate property, but serve as useful hints:

Håkan Sundell, Philippas Tsigas: Lock-free deques and doubly linked lists. J. Parallel Distrib. Comput. 68(7): 1008-1020 (2008)

List in one direction always correct (item in data structure iff it is reachable in this direction), links in other direction only approximate. Threads help each other to get other direction up to date

Priority-queue based on Skiplist (chap. 15)

Often needed sequential data structure (pool) with fairness guarantees based on key-value (here item = key):

- insert(key)
- key = delete_min()

Sometimes:

- decrease_key(item_reference,new_key)
- delete(item_reference)

Skiplist based implementation (lock-free or lazy)

- Add atomic candidate flag to nodes
- delete_min() scans through level 0 list for first (=smallest key) non-candidate item, sets candidate flag (CAS), and deletes the node by calling skiplist remove()

```
public Node<T> findmincandidate() {
  Node<T> curr = null;
  curr = head.next[0].getReference();
  while (curr!=tail) {
    if (curr.candidate.compareAndSet(false,true))
      return curr;
    else curr = curr.next[0].getReference();
  }
  return null; // no candidates for delete_min
}
```

Note:
Could save some CAS operations by first reading the candidate flag (as in TTAS lock)

A linearizable execution cannot return key 10 before key 3 is returned

©Jesper Larsson Träff

Properties of skiplist based priority queue

Progress guarantees (lock-freeness) inherited from skiplist

BUT:

- Not linearizable
- Instead, quiescently consistent

Alternative/better skiplist based priority queues

Nir Shavit, Itay Lotan: Skiplist-Based Concurrent Priority Queues. IPDPS 2000: 263-268
Håkan Sundell, Philippas Tsigas: Fast and lock-free concurrent priority queues for multi-thread systems. J. Parallel Distrib. Comput. 65(5): 609-627 (2005)
Jonatan Lindén, Bengt Jonsson: A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention. OPODIS 2013: 206-220

Relaxed priority queues

Dan Alistarh, Justin Kopinsky, Jerry Li, Nir Shavit: The SprayList: a scalable relaxed priority queue. PPOPP 2015: 11-20
Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, Philippas Tsigas: The lock-free k-LSM relaxed priority queue. PPOPP 2015: 277-278
Tingzhe Zhou, Maged M. Michael, Michael F. Spear: A Practical, Scalable, Relaxed Priority Queue. ICPP 2019: 57:1-57:10

MultiQueue: A simple, randomized, relaxed priority queue

With p threads, use cp sequential priority queues, constant c>0 tuning parameter (each thread maintains c queues)

Idea:

- Insert: Choose queue randomly, try to lock, insert
- Delete minimum element: Choose two queues at random, try to lock the one with the smallest element, delete

Hamza Rihani, Peter Sanders, Roman Dementiev: Brief Announcement: MultiQueues: Simple Relaxed Concurrent Priority Queues. SPAA 2015: 80-82

```
boolean insert(T *x)
{
  int i; // queue index
  do {
    i = UniformRandom(0,cp);
  } while (!PQ[i].trylock())
  PQ[i].insert(x);
  PQ[i].unlock();
  return true;
}
```


©Jesper Larsson Träff

```
boolean delete_min(T *x)
{
  int i, j; // two queue indices
  int k;
  do {
    i = UniformRandom(0,cp);
    j = UniformRandom(0,cp);
    // sample without locking
    if (PQ[i].findmin()<PQ[j].findmin())
      k = i; else k = j;
  } while (!PQ[k].trylock())
  PQ[k].delete_min(x);
  PQ[k].unlock();
  return true;
}
```

Proposition:
MultiQueue insert and delete_min are wait-free when c>1

Proof: Since at most p queues can be locked at any time, the success probability of finding an unlocked queue is constant

Corrollary:
MultiQueue insert and delete_min perform expected O(1) trylock operations, and complete in time determined by the sequential priority queue operation

No strict bound for the relaxation (how far can the returned element of a delete_min operation be from the smallest element in queue at that time?), but two attempts is (much) better than one:

Yossi Azar, Andrei Z. Broder, Anna R. Karlin, Eli Upfal: Balanced Allocations. SIAM J. Comput. 29(1): 180-200 (1999)

Petra Berenbrink, Artur Czumaj, Angelika Steger, Berthold Vöcking: Balanced Allocations: The Heavily Loaded Case. SIAM J. Comput. 35(6): 1350-1385 (2006)

Michael Mitzenmacher: The Power of Two Choices in Randomized Load Balancing. IEEE Trans. Parallel Distrib. Syst. 12(10): 1094-1104 (2001)

New result

Dan Alistarh, Justin Kopinsky, Jerry Li, Giorgi Nadiradze:
The Power of Choice in Priority Scheduling. PODC 2017: 283-292

does give a tight bound on the relaxation quality (rank) of the MultiQueue

Benchmarking priority queues

Throughput benchmark often used in literature:

- Some mix of insertions and deletions (usually 50/50%)
- All threads perform same mix of operations
- Insert keys chosen uniformly from some (large) range, no dependency between deleted and inserted keys

Prefill queue, run for some interval (10 seconds), report total number of operations

Scalable PQ:
Throughput increases linearly with p (in some range)

Question:

Is this a good benchmark?

> Jakob Gruber, Jesper Larsson Träff, Martin Wimmer: Brief Announcement: Benchmarking Concurrent Priority Queues. Performance of k-LSM and Related Data Structures. SPAA 2016, to appear

k-LSM:

- Linearizable, relaxed priority queue built on Log-Structured Merge trees, proposed by Martin Wimmer ("Variations for Shared Memory Systems", PhD thesis, TU Wien, 2014)
- Standalone C++ implementation by Jakob Gruber ("k-LSM: A relaxed, lock-free priority queue", MSc. Thesis, TU Wien, 2016)
- Hybrid of local and global LSM component, relaxation guarantee kp for chosen parameter k

Uniformly drawn random 32-bit keys, same mix of insert and delete

TUW "mars", 80-core Intel Xeon E7-8850, 2GHz; `gcc 5.2.1` with `-O3` and `-flto`; 30 repetitions. Also results on 48-core AMD, 64-core Sparc T5, 61-core Intel Xeon Phi

Descending dependent keys, same mix of insert and delete

©Jesper Larsson Träff

Ascending dependent keys, same mix of insert and delete

©Jesper Larsson Träff

Split mix of insert and delete, uniform 32-bit keys

- Many (most?) concurrent PQ's do not scale universally
- Locality important for performance/scalability
- Simple throughput benchmark can be very misleading
- Good to have: standard framework for PQ's

Advanced Multiprocessor Programming

Jesper Larsson Träff
traff@par.tuwien.ac.at
Research Group Parallel Computing
Faculty of Informatics, Institute of Computer Engineering
Vienna University of Technology (TU Wien)

©Jesper Larsson Träff

Set/Dictinary based on hash tables (chap. 13)

Dictionary (set) data structure supports

- add(x)
- remove(x)
- contains(x)

for item (key) x. Again, no distinction between keys and items

Assume existence of hash function h(x) that maps item/key to integer, with good properties:

- $h(x) \neq h(y)$ for $x \neq y$ with high probability
- h(x) can be evaluated efficiently in O(1) steps, and fast

Currently much activity on fast, good hashing (Thorup ao.)

Idea:
Store items in hash table of some size (capacity), item x at position h(x) mod capacity. All operations in O(1) expected/high probability?

Much potential for concurrency: Disjoint-access parallelism

What to do on collisions, h(x)=h(y)?

Two possibilities:
- Closed addressing: Items with hash key h(x) stored in bucket at index h(x) mod capacity
- Open addressing: At most one item at index h(x) mod capacity

What to do on when table (buckets) become too full?

Solution: Resize, move item to new table

Closed address hash table

Each table entry maintains (small) bucket of elements (implemented as list-array)

To ensure constant O(1) access time, the buckets must have constant maximum size. When some bucket exceeds maximum (or average bucket size exceeds maximum), table is resized and items redistributed

Buckets as array lists (any suitable data structure can be used)

```
public abstract class BaseHashTable<T> {
  protected List<T> table;
  protected int size; // total number of items in set
  protected int capacity; // number of indices in table
  public BaseHashTable(int c) {
    size = 0;
    capacity = c;
    table = (List<T>[]) new List[capacity];
    for (int i=0; i<capacity; i++)
      table[i] = new ArrayList<T>(); // bucket
  }
  …
}
```

Implementation choice

Contains method from ArrayList

```
public boolean contains(T x) {
  acquire(x);
  try {
   return table[h(x)%capacity].contains(x);
  } finally {
    release(x);
  }
}
```

Acquire and release functions give exclusive access to the necessary parts of the hash table, ideally only the needed index (and the size)

©Jesper Larsson Träff

```
public boolean add(T x) {
  int result = false;
  acquire(x);
  try {
    int ix = h(x)%capacity;
    if (!table[ix].contains(x)) {
      table[ix].add(x); // insert into bucket
      result = true;
      size++;
    }
  } finally {
    release(x);
  }
  if (policy()) resize();
  return result;
}
```

Policy function determines when to resize (to maintain O(1) access time)

```
public boolean remove(T x) {
  int result = false;
  acquire(x);
  try {
    int ix = h(x)%capacity;
    if (table[ix].contains(x)) {
      table[ix].remove(x); // delete bucket
      result = true;
      size--;
    }
  } finally {
    release(x);
  }
  // no shrinking here (exercise)
  return result;
}
```

Lock-based closed-address hash tables

Trivial solution: Coarse-grained locking, one lock for whole hash table

```
public class CoarseHashTable<T>
       extends BaseHashTable<T> {
  final Lock lock;
  public CoarseHashTable(int capacity) {
    super(capacity);
    lock = new ReentrantLock();
  }
  public final void acquire(T x) {
    lock.lock();
  }
  public final void release(T x) {
    lock.unlock();
  }
}
```


©Jesper Larsson Träff

Resize:
To maintain constant size buckets (and constant time access), when average load exceeds some constant THRESHOLD, double the number of buckets

Implemented in policy and resize functions. Resizing acquires lock for whole table and "stops the world" (like most garbage collectors)

```
public boolean policy() {
  return size/capacity > THRESHOLD; // avg. load
}

public void resize() {
  lock.lock();
  try {
    if (!policy()) return;

    // double table capacity (next slide)
    …
  } finally lock.unlock();
}
```

Somebody else may have resized, recheck condition

©Jesper Larsson Träff

```
// double table capacity

capacity = 2*capacity;
List<T>[] oldtable = table;
table = (List<T>[]) new List[capacity];
for (int i=0; i<capacity; i++)
  table[i] = new ArrayList<T>();
for (int i=0; i<capacity; i++)
  for (int j=0; j<oldtable[i].length; j++)
    int x = oldtable[i][j];
    table[h(x)%capacity].add(x);
delete oldtable;
```


©Jesper Larsson Träff

Better solution: Striped hash set, fixed array of locks

```
public class StripedHashTable<T>
       extends BaseHashTable<T> {
  final ReentrantLock[] locks;
  public StripedHashTable(int capacity) {
    super(capacity);
    locks = new Lock[capacity];
    for (int i=0; i<capacity; i++)
      locks[i] = new ReentrantLock();
  }
  public final void acquire(T x) {
    locks[h(x)%locks.length].lock();
  }
  public final void release(T x) {
    locks[h(x)%locks.length].unlock();
  }
}
```

Not same as table capacity

After 1st resize, locks[i] protects j=h(x)%capacity with j%locks.length=i

©Jesper Larsson Träff

Resize (table only!): Easy, needs to acquire all locks of the striped lock array. Done in ascending order, no deadlock

Drawback: Resize "stops the world", lock table is not changed, granularity gets larger and larger as table increases

```
public void resize() {
  int oldcapacity = table.length;
  for (Lock lock: locks) lock.lock();
  try {
    if (oldcapacity!=table.length) return;
    // already done by other thread
    int newcapacity = 2*oldcapacity;
    List<T>[] oldtable = table;
    table = (List<T>[])new List[newcapacity];
    for (int i=0; i<newcapacity; i++)
      table[i] = new ArrayList<T>();
    for (List<T>bucket: oldtable) // copy table elements
      for (T x: bucket)
        table[h(x)%table.length].add(x);
  } finally {
    for (Lock: lock) lock.unlock();
  }
}
```

Must be in fixed order

©Jesper Larsson Träff

But:
Changing the lock array in the resize operation does not work:

Some thread may be trying to acquire lock assuming table capacity has not changed. This lock may either not exist, or not protect the right entry.

(Also note that a resize operation may recursively trigger nested resize calls (by the add); but this still works correctly (why?).
For this reason, reentrant locks are needed)

Next refinement: Refinable hash table (resizable locks).

Introduce global marked reference (owner thread and flag) to indicate whether resize is in progress and by which thread. Acquire operation checks marked reference, and acquires lock only if resize is not in progress, spins otherwise. Release just releases the lock. Resize atomically sets flag and reference, and can now resize both table and lock array

Drawback:
Resize operation still "stops the world", no concurrency when resize in progress (for both striped and refinable hash table)

```
public class RefinableHashTable
       extends BaseHashTable<T> {
  AtomicMarkableReference<Thread> owner;
  volatile ReentrantLock[] locks;
  public RefinableHashTable(int capacity) {
    super(capacity);
    locks = new ReentrantLock[capacity];
    for (int i=0; i<capacity; i++) {
      locks[i] = new ReentrantLock();
    }
    owner =
      new AtomicMarkableReference<Thread>(null,false);
    …
}
```


©Jesper Larsson Träff

```
public void acquire(T x) {
  boolean[] mark = {true};
  Thread t = Thread.currentThread();
  Thread w;
  while (true) {
    do { // spin while resizing in progress
      w = owner.get(mark);
    } while (mark[0]&&w!=t);
    ReentrantLock oldlocks = locks;
    ReentrantLock oldlock =
      oldlocks[h(x)%oldlocks.length];
    oldlock.lock();
    w = owner.get(mark);
    if ((!mark[0]||w==t)&&locks==oldlocks) return;
    else oldlock.unlock();
   // resizing has taken place, retry
  }
}
```

```
public void resize() {
  int oldcapacity = table.length;
  boolean[] mark = {false};
  int newcapacity = 2*oldcapacity;
  Thread t = Thread.currentThread();
  if (owner.compareAndSet(null,t,false,true) {
    try {
      if (table.length!=oldcapacity) return;
      waitforquiescence();
      List<T>[] oldtable = table;
      table = (List<T>[])new List[capacity];
      for (int i=0; i<newcapacity; i++)
        table[i] = new ArrayList<T>();
      locks = new ReentrantLock[newcapacity];
      for (int j=0; j<locks.length; j++)
        locks[j] = new ReentrantLock();
      … // copy table elements to new table
    } finally owner.set(null,false);
  }
}
```


©Jesper Larsson Träff

Resizing thread must wait for all operations on hash table to complete (quiescence)

```
protected void waitforquiescence () {
  for (ReentrantLock lock: locks) {
    // spin until lock is released
    while (lock.isLocked()) { }
  }
}
```

How good is this (in real applications)? Resizing may be a major bottleneck. Shrinking of tables and locks is not done

Lock-free closed address hash table

Fine-grained locks on buckets may not be too problematic(?); but "stop-the world" on resize could be a major bottleneck

Ideas:

- Use lock-free list to store all buckets, with references into bucket list from hash table
- Make resize purely local operation, breaking up too long bucket
- Be lazy

To make this work, the lock-free list must be organized such that a hashed item does not have to change position on resize.
How is this possible?

Maintain list in a certain order

©Jesper Larsson Träff

©Jesper Larsson Träff

Resize:
Double table size, split sublists without changing order of elements

©Jesper Larsson Träff

Splitting sublists without changing order possible if bucket list is maintained in sorted order on $reverse_w(h(x))$

See HPC lecture

$reverse_w(y)$:
Bit reversal of y, e.g., with w=8 reverse(01101101) = 10110110, reverse(11110000) = 00001111, reverse(00111100) = 00111100

3-bit reverse (example):

©Jesper Larsson Träff

Technicalities:

- Lock-free list works best with sentinel head and tail elements (never changed)
- The recursive split-ordered list has several sentinel elements (corresponding to non-empty buckets), distinguish between sentinel and normal items by upper bit (set for normal, non-sentinel items)
- Implementations (next slides) for 32-bit hash keys
- Each sublist a lock-free BucketList
- If sentinel for some sublist is not present, it is added lazily

- Reuse find-method of lock-free list-based set. Find operations returns a window structure with pred and curr references, such that pred.key<x≤curr.key

©Jesper Larsson Träff

```
public class BucketList<T> {
  Node head;
  static final int HIGH = 0x80000000;
  static final int MASK = 0x00FFFFFF;
  public BucketList() {
    head = new Node(0);
    head.next =
      new AtomicMarkedReference<Node>(
        new Node(Integer.MAX_VALUE),false);
  }
  public int makeNormalKey(T x) {
    int key = h(x) & MASK; // lowest bytes
    return reverse(key | HIGH);
  }
  public int makeSentinelKey(int key) {
    return reverse(key & MASK);
  }
  …
}
```

32-bit reverse function

Properties of item keys:

Sentinel smaller than all items belonging to bucket (e.g., sentinel for bucket 20, after reverse, has least significant bit 0, with same upper bits as all normal items of bucket 20, which have least significant bit 1)

All items in bucket smaller than sentinel of next bucket

Property of table size:

Starting from $1=2^0$, always a power of 2

```
public class BucketList<T> {
  private Window find(Node head, T x);

  public boolean contains(T x) {
    int key = makeNormalKey(x);
    Window window = find(head,x);
    Node curr = window.curr;
    return (curr.key==x);
  }

  public boolean add(T x); // also key reverse
  public boolean remove(T x);
}
```

All operations reverse key by makeNormalkey. Find operation from lock-free list-based set; returns window of pred and curr elements

```
public BucketList<T> getSentinel(int index) {
  int key = makeSentinelKey(index);
  while (true) {
    Window window = find(head,key);
    Node pred = window.pred;
    Node curr = window.curr;
    if (curr.key==key)
      return new BucketList<T>(curr);
    else {
      Node node = new Node(key);
      node.next.set(pred.next.GetReference(),false);
      if (pred.next.compareAndSet(curr,node,
                                  false,false)) {
        return new BucketList<T>(node);
      else continue;
    }
  }
}
```


©Jesper Larsson Träff

Lock-free hash table is an array of bucket lists. Array indices either sentinel (head) of list, or uninitialized (null)

Note: This implementation preallocates whole table of maximum capacity. This can be improved

©Jesper Larsson Träff

```
public class LockFreeHashTable<T> {
  protected BucketList<T>[] table;
  protected AtomicInteger tablesize;
  protected AtomicInteger size;
  public LockFreeHashTable(int c) {
    table = (BucketList<T>[]) new BucketList[c];
    table[0]  = new BucketList<T>();
    tablesize = new AtomicInteger(1);
    size      = new AtomicInteger(0);
  }
  … // private functions to get and init buckets
}
```

The lock-free hash table consist of (for now: fixed) table of bucket lists.
Maintain current table size (assumption: Always smaller than capacity c), and current number of items (size) in hash table

```
public boolean add(T x) {
  int bucket = h(x)%tablesize.get();
  BucketList<T> b = getBucket(bucket);
  if (!b.add(x)) return false;
  int s = size.getAndIncrement();
  int t = tablesize.get();
  if ((s+1)/t>THRESHOLD)
    tablesize.compareAndSet(t,2*t);
  return true;
}
```

Resize: Just double table size

Adding new element: Look for current table entry of bucket

Increase table size (implicit resize) when average bucket load beyond THRESHOLD

```
private BucketList<T> getBucket(int bucket) {
  if (table[bucket]==null) initializeBucket(bucket);
  return table[bucket];
}
```

```
private void initializeBucket(int bucket) {
  int parent = getParent(bucket);
  if (table[parent]==null) initializeBucket(parent);
  BucketList<T> b =
    bucket[parent].getSentinel(bucket);
  if (b!=null) table[bucket] = b;
}
```

```
private void getParent(int bucket) {
  int parent = tablesize.get();
  do {
    parent = parent>>1;
  } while (parent>bucket);
  parent = bucket-parent;
  return parent;
}
```

Parent is the index of a bucket to be split. Should be close to new bucket, but not lead to too many recursive initializations:

Find largest power of 2 not greater than bucket index. Use this to set most significant bit to zero in bucket index. E.g., 7 = 00000111_2 becomes 00000011_2=3

Initializing lock-free hash table with at most 8 entries

©Jesper Larsson Träff

Some add(x) operations trigger resize.

Up till then: get bucket returns table[0] bucket, lock-free list add maintains sorted order (reverse-bit of key)

Size atomically incremented to 3, bucketsize to 2

©Jesper Larsson Träff

add(6): 6%2 = 0, so in table[0], does not split lists

©Jesper Larsson Träff

add(3): 3%2 = 1, so in table[1], uninitialized

Parent of 1 is 0, find returns node 1

©Jesper Larsson Träff

add(3): 3%2 = 1, so in table[1]

New sentinel 1, after which 3 is inserted

©Jesper Larsson Träff

Analysis: (non-trivial)

Initializing empty bucket (table[bucket]==**null**) can recursively trigger $\log_2$(tablesize) initializations
Can be shown that expected depth of recursion is constant
Overall (expected) complexity of all operations is O(1) steps

Note also:
Large buckets where all h(x)%newtablesize go into same bucket remain large

Contains and remove operations also initialize buckets (needed?)

©Jesper Larsson Träff

```
public boolean remove(T x) {
  int bucket = h(x)%tablesize.get();
  BucketList<T> b = getBucket(bucket);
  if (b.remove(x)) return false;

  // no shrinking (should perhaps be done)
  return true;
}

public boolean contains(T x) {
  int bucket = h(x)%tablesize.get();
  BucketList<T> b = getBucket(bucket);
  return b.contains(x);
}
```

Properties:

- Lock-freedom inherited from lock-free list
- Linearizable; linearization points for concurrent updates to same bucket are the linearization points of corresponding list operations

Drawback: Fixed (large), preallocated table; not shrinkable (can be fixed)

Ori Shalev, Nir Shavit: Split-ordered lists: Lock-free extensible hash tables. J. ACM 53(3): 379-405 (2006)

Open addressing

All items stored in hash table, at most one item at h(x) mod capacity

Linear probing: To find item x, search linearly from index h(x) mod capacity until x found or empty slot

Some results, some open problems:

Chris Purcell, Tim Harris: Non-blocking Hashtables with Open Addressing. DISC 2005: 108-121
Hui Gao, Jan Friso Groote, Wim H. Hesselink: Lock-free dynamic hash tables with open addressing. Distributed Computing 18(1): 21-42 (2005)

Open address hashing: Cuckoo

Idea:
A newly added item pushes out previous item in same slot (Cuckoo)

Use two hash tables (same size), and two hash functions h0(x) and h1(x)

©Jesper Larsson Träff

Example: add(x)

©Jesper Larsson Träff

Example: add(x)

©Jesper Larsson Träff

Example: add(x)

©Jesper Larsson Träff

Example: add(x)

©Jesper Larsson Träff

Example: add(x)

Done when free slot eventually found.

©Jesper Larsson Träff

Example: add(x), add(y) with h0(x)=h0(y)

©Jesper Larsson Träff

Example: add(x), add(y) with h0(x)=h0(y)

Observation: x either in table[0][h0(x)] or in table[1][h1(x)]

©Jesper Larsson Träff

Example: add(x), add(y), add(z) with h0(z)=h0(y) and h1(y)=h1(x)

©Jesper Larsson Träff

Example: add(x), add(y), add(z) with h0(z)=h0(y) and h1(y)=h1(x)

©Jesper Larsson Träff

Example: add(x), add(y), add(z) with h0(z)=h0(y) and h1(y)=h1(x)

©Jesper Larsson Träff

Example: add(x), add(y), add(z) with h0(z)=h0(y) and h1(y)=h1(x)

©Jesper Larsson Träff

Example: add(x), add(y), add(z) with h0(z)=h0(y) and h1(y)=h1(x)

Done when free slot eventually found. Or cycle found. Or chain of kick-outs too long

©Jesper Larsson Träff

```
public boolean add(T x) {
  if (contains(x)) return false;

  for (int i=0; i<LIMIT; i++) {
    if ((x=swap(0,h0(x),x))==null) return true;
    if ((x=swap(1,h1(x),x))==null) return true;
  }
  resize(); // failed: too long chain or cycle
  return add(x); // try again
}

public boolean contains(T x) {
  if (table[0][h0(x)]==x) return true;
  if (table[1][h1(x)]==x) return true;
  return false;
}
```

Resize operation increases table sizes, and chooses new hash functions

Sequential cuckoo hashing:
Constant time contains and remove operations.
Average number of displacements in add operations expected constant

Rasmus Pagh: Cuckoo Hashing. Encyclopedia of Algorithms 2016: 478-481
Rasmus Pagh, Flemming Friche Rodler: Cuckoo hashing. J. Algorithms 51(2): 122-144 (2004)

Concurrent cuckoo hashing:
Phased, striped lock-based implementation, refine to use refinable locking

Phased, lock-based Cuckoo hashing:

Problem is to avoid having to lock long sequences of table indices due to long sequences of swaps.

Possible solution: Do the relocation in phases, maintain small, fixed buckets of items for each index in the two hash tables.

PROBE_SIZE: Fixed size of bucket, cannot be exceeded
THRESHOLD (<PROBE_SIZE): Size of bucket that will trigger relocation

©Jesper Larsson Träff

```
public abstract class PhasedCuckooHashTable<T> {
  volatile int capacity;
  volatile List<T>[][] table;
  public PhasedCuckooHashTable(int c) {
    capacity = c;
    table = (List<T>[][])new ArrayList[2][c]; // util
    for (int i=0; i<2; i++) {
      for (int j=0; j<c; j++) {
        table[i][j] = new ArrayList<T>(PROBE_SIZE);
      }
    }
  }
}
```


©Jesper Larsson Träff

remove: Item is in either table[0] or table[1], remove from either if present
contains: Check table[0] and table[1]
add: If below THRESHOLD, add to fixed bucket, otherwise, try other table. If above THRESHOLD, trigger relocation of items. Resize hash table[0] and table[1] if both buckets full (PROBE_SIZE). Try relocation only up to LIMIT (cycle or chain too long)

```
public boolean remove(T x) {
  acquire(x);
  try {
    List<T> set0 = table[0][h0(x)%capacity];
    List<T> set1 = table[1][h1(x)%capacity];
    if (set0.contains(x)) {
      set0.remove(x);
      return true;
    }
    if (set1.contains(x)) {
      set1.remove(x);
      return true;
    }
    return false;
  } finally release(x);
}
```

```
public boolean add(T x) {
  T y = null;
  acquire(x);
  int i = -1, h = -1; boolean mustresize = false;
  try {
    if (contains(x)) return false;
    List<T> set0 = table[0][h0(x)%capacity];
    List<T> set1 = table[1][h1(x)%capacity];
    if (set0.size()<THRESHOLD) {
      set0.add(x); return true;
    } else if (set1.size()<THRESHOLD) {
      set1.add(x); return true;
    } else if (set0.size()<PROBE_SIZE) {
      set0.add(x); i = 0; h = h0(x)%capacity;
    } else if (set1.size()<PROBE_SIZE) {
      set1.add(x); i = 1; h = h1(x)%capacity;
    } else mustresize = true;
  } finally release(x);
  // resize and/or relocate (next slide)
```

```
  // resize and/or relocate
  if (mustresize) {
    resize(); add(x);
  } else if (!relocate(i,h)) resize();
  return true; // x must have been present
}
```

Note: Locks have been released after (unsuccessful) insertion, other threads may use table concurrently

Which set? Hashed index in set

```
public boolean relocate(int i, int hi) {
  int hj;
  int j = 1-i;
  for (int r=0; r<LIMIT; i++) {
    List<T> seti = table[i][hi];
    T y = seti.get(0);
    if (i==0) hj = h1(y)%capacity;
    else hj = h0(y)%capacity;
    acquire(y);
    List<T> setj = table[j][hj];

    // relocate y from seti to setj
  }
}
```

Try to relocate oldest item from set

```
// relocate y from seti to setj
try {
  if (seti.remove(y)) {
    if (setj.size()<THRESHOLD) {
      setj.add(y); return true;
    } else if (setj.size()<PROBE_SIZE) {
      setj.add(y); i = 1-i; hi = hj; j = 1-j;
    } else {
      seti.add(y); return false;
    }
  } else if (seti.size()>=THRESHOLD) continue;
  else return true;
} finally release(y);
```

Could have been removed by other thread

Both sets full, add back and trigger resize

y gone, but still to do

… and back to the loop (LIMIT)

©Jesper Larsson Träff

Locks for the items as for the simple, closed hash table

```
public class StripedCuckooHashTable<T>
       extends PhasedCuckooHashTable {
  final ReentrantLock[] lock;
  public StripedCuckooHashTable(int capacity) {
    super(capacity);
    lock = new ReentrantLock[2][capacity];
    for (int i=0; i<2; i++) {
      for (int j = 0; j<capacity; j++) {
        lock[i][j] = new ReentrantLock();
      }
    }
  }
  …
}
```

Locks for both table[0] and table[1] needed (not refinable here)

```
public final void acquire(T x) {
  lock[0][h0(x)%capacity].lock();
  lock[1][h1(x)%capacity].lock();
}

public final void release(T x) {
  lock[0][h0(x)%capacity].unlock();
  lock[1][h1(x)%capacity].unlock();
}
```


©Jesper Larsson Träff

```
public void resize() {
  int oldcapacity = capacity;
  for (Lock lock0s: lock[0]) lock0s.lock();
  try {
    if (capacity!=oldcapacity) return; // other did it
    List<T>[][] oldtable = table;
    capacity = 2*capacity;
    table = (List<T>[][])new List[2][capacity];
    for (List<T>[] set: table) {
      for (int i=0; i<set.length(); i++) {
        set[i] = new ArrayList<T>(PROBE_SIZE);
      }
    }
    … // move items from old to new table
  } finally {
    for (Lock lock0s: lock[0]) lock0s.unlock();
  }
}
```

Concurrent cuckoo hashing:
Lock-free algorithms exist

Nhan Nguyen, Philippas Tsigas: Lock-Free Cuckoo Hashing. ICDCS 2014: 627-636

©Jesper Larsson Träff

An adaptive open-address hash table, with elements of linear-probing and cuckoo-hashing (and attention to good cache behavior)

Maurice Herlihy, Nir Shavit, Moran Tzafrir: Hopscotch Hashing. DISC 2008: 350-364

Not lock-free (still open question?)

Recent, simple basic algorithms (open addressing, linear probing), with lots of practical considerations, comparisons to a number of other schemes

Tobias Maier, Peter Sanders, Roman Dementiev: Concurrent Hash Tables: Fast and General(?). TOPC 5(4): 16:1-16:32 (2019)

Advanced Multiprocessor Programming
Vorbesprechung

Jesper Larsson Träff
Thomas Schlögl, Stephan Felber
traff@par. …
Research Group Parallel Computing 191-4
Faculty of Informatics, Institute of Computer Engineering
TU Wien

©Jesper Larsson Träff

The takeaway

Lectures:
Mondays, 13:00 (c.t)-15:00, EI 5 "Hohenegg".

Exercises:
Two batches, exercises from the book(s), hand-in 24.4 and 8.5

Exercise Presentations, projects and progress presentation:
Thursdays, after Easter, 11:00 (c.t) – 13:00

Project hand-in 19.6, NO EXTENSION. Machine account 27.3.
Exam: 26.6 to 30.6. Sign up in TISS (May)

Further information, all material: TUWEL (and TISS)

Team

- Jesper Larsson Träff: Lectures, everything
- Thomas Schlögl, Stephan Felber: Exercises, Projects, Exam

- Markus Hinkel: Technical support

The facts and the problems

Modern multi-core processors (2, 4, …, 80 cores + multi/hyper-threading)

- do not really correspond to standard theoretical models (PRAM)
- are very, very difficult to program effectively and efficiently: Performance and correctness

This course:

- Advanced programming techniques in theory (the possible and the impossible) and practice for modern multi-core processors (not GPUs):
- How to implement traditional constructs like locks and barriers efficiently
- How to program without locks and barriers: Data structures and algorithms

Formalities

VU (Lecture-Exercises-Project)

4.5 ECTS (=112.5 hours of work)

Breakdown:

- Lecture 1.5 ECTS
- Exercises 1.0 ECTS
- Programming Project: 2.0 ECTS

Participation MANDATORY, credit given based on Participation, Blackboard Exercises, Programming Project, and Exam

Detailed break-down

- Planning, intro ("Vorbesprechung"): 2h
- Lectures: 15 x 2h = 30h
- Preparation: 15 x 2.5h = 22.5h
- Project/Exercises: 50h
- Exam, including preparation: 8h

Total: 112.5h = 4.5 ECTS

Lecture:

Monday, 13:00 – 15:00, EI 5, "Hohenegg"
Thursdays (from 27.4), 11:00 – 13:00, EI 5, "Hohenegg"

"Sprechstunde" Jesper Larsson Träff, Thomas Schlögl, Stephan Felber: by appointment

Email: traff@par. …, Thomas.e191-02.schloegl@ … stephan.felber@ …

©Jesper Larsson Träff

Sign-up required (deadline 31.3, in TISS)

Sign-out if you don't follow the lecture (before 24. April)

- Theory exercises should be done individually (discussions encouraged…)
- Project in groups of ≤2 members

Now: ≤3

- Get machine account via TUWEL: 27.3 (will be enabled soon, TUWEL exercise to upload 4K public ssh key)

Topics, Goals

Basic understanding of principles and practice of thread-based shared-memory multiprocessor parallel programming

Principles/theory:

- Synchronization and coordination mechanisms
- Scope and limitations
- Correctness: safety and liveness

C/C++, threads, OpenMP, CilkPlus, …

Practice:

- Implementation of basic synchronization mechanisms
- Fundamental (lock- and wait-free) data structures
- Memory models

Supporting higher-level shared memory programming models:

- Task parallel models by work-stealing
- (Transactional memory)

Literature/Material

Book:

Maurice Herlihy (Brown), Nir Shavit (Tel Aviv): The Art of Multiprocessor Programming. Morgan Kaufmann Publishers, 2008, revised 1st edition, 2012, second edition, 2021 (now with Victor Luchangco and Michael Spear)

Recommended: buy it! …despite Elsevier

Lecture slides, additional papers… all on TUWEL)

Approx. Coverage

Chapters 1-5 (6), Chapters 7, 9, 10, 11, (12?), 13-16, (17?)
Work-stealing and memory models from other sources

Prerequisites:
- "Introduction to Parallel Computing"
- Algorithms and data structures
- C/C++ (or Java) programming

Possible follow-up:
- Parallel Algorithms (PRAM, Scheduling)
- HPC
- Distributed Algorithms (Ulrich Schmid)
- Seminars, Project, Master's thesis

Michel Raynal:
Concurrent programming: Algorithms, Principles, and Foundations. Springer, 2013

Gadi Taubenfeld:
Synchronization Algorithms and Concurrent Programming.
Pearson/Prentice Hall 2006

Synthesis lectures on computer architecture. Morgan&Claypool

Michael L. Scott: Shared-Memory Synchronization, 2013

Daniel J. Sorin, Mark. D. Hill, David A. Wood: A Primer on Memory Consistency and Cache Coherence, 2011, second edition 2020

Parallel computing background (also wikipedia.org)

©Jesper Larsson Träff

Exercises/Project

"Theoretical" exercises from book, hand-in and discussion/presentation on blackboard

Two slots

Small programming project:
Implementation and benchmarking (comparison) of lock-free data structure(s) and other material from the lectures

Implementation in C++ threads or C with (p)threads, OpenMP, possibly with CilkPlus (or other C-based framework)

Latex template will be available. Follow instructions on how/what to hand in

Exercises: 2 batches, hand-in

- 27.3 -> 24.4 (Monday):
- 24.4 -> 8.5 (Monday):

Mandatory to pass course (and at least 50% correct)

Project is done in groups of ≤2 (since 10.3: ≤3)

Project:

- 17.4 (Monday): Project topic presentation (by me)
- 24.4 (Monday): Project commit (by you)
- Ca. 1.6: Project status presentation (by you: each group gives a 5-10 minute overview of what it is doing)
- 19.6: Project hand-in (fixed deadline, no extension)

EXAM: From late June (26.6-30.6)

Start early on the project!

System

Possible to start developing on own PC/laptop (no lab access)

Benchmarking/testing: Nebula, new shared-memory node at TUWien

- 2 AMD EPYC 7351P 32-core processors, 2-way hyperthreading, 1.2GHz, total 64 cores, 256G main memory
- Possibly ARM system (Medusa)

Start early on the project!

More later… (get account via TUWEL on 27.3)

Grading/participation

- Attending lectures and exercises (MANDATORY)
- Active participation
- Solving the exercises, presentation on the blackboard (theoretical exercises, hand-in of practical programming exercise, MANDATORY)
- Examination based on project but can cover whole material
- NO ChatGPT

NOTE:
- You only learn by doing exercises and project by yourself.
- Copying/plagiarism will result in grade 5
- Discussion with other groups encouraged, but hand in your own solution

Don't forget: EVALUATE THE COURSE by end of semester (TISS)

Project hand-in:

- Short description of problem, your solution
- Some argument for correctness, testing procedure…
- The required tests/benchmark comparisons (plots, tables)

Both correctness and performance are important!

Grade weighting: $\frac{1}{4}$ for exercises, $\frac{1}{2}$ for project, $\frac{1}{4}$ for exam

Start early on the project!

Solving in group:

- Active collaboration, "2*100%", NOT "2*50%"
- Both members get same grade (unless blatantly different)
- Both members must understand all aspects of solutions

Project hand-in:

- Short description of problem, your solution
- Some argument for correctness, testing procedure…
- The required tests/benchmark comparisons (plots, tables)

Both correctness and performance are important!

If done in group, only one hand-in counts (the worst…); groups can hand in two solutions, that should then be identical, or just one (more risky…)

Follow instructions, hand-in via TUWEL

Exam

Oral examination based on project, but can cover whole lecture

Ca. $\frac{1}{2}$ hour

Now: Might be written exam instead

Sign-up in TISS later (group exam or individual to be decided)

If you are signed up for the exam, but do not show up without (or extremely late) notice, grade is 5

Detailed lecture plan (subject to change), Mondays

6.3: "Vorbesprechung". Intro. Mutual Exclusion problem
13.3: Constructions of atomic registers, register snapshot
20.3: Relative power of synchronization operations, correctness conditions
27.3: Relative power of synchronization operations, universality
17.4: Projects (description). Relative power of synchronization operations. Wrap-up.
24.4: Practical lock implementations
(27.4, Thursday, TBA)
8.5: Data structures (I): List-based set
15.5: Data structures (II): Queues and stacks
22.5: Memory consistency models, memory reclamation
5.6: Data structures (III): Skiplist, priority queues
12.6: Data structures (IV): Hash tables
19.6: Barrier synchronization, work-stealing theory (TBA)

Easter: 3.4-14.4
Whitsun: 29.5

1st of May 1.5

Detailed exercise plan (subject to change), Thursdays

27.3 to 24.4 (Monday): Exercise 1, hand-in
24.4 to 8.5 (Monday): Exercise 2, hand-in

4.5 (Thursday): Exercise feedback 1
11.5 (Thursday): Exercise feedback 2

17.4: Project topic presentation
25.5: Project Q&A
1.6: Project presentations, state-of-the-art… (by you)
15.6: Project Q&A

Follow-up

- Projects (6.0+6.0 ECTS)
- Seminar in WS23
- Parallel Algorithms (WS23: VU, 3.0 ECTS)
- High Performance Computing (WS23: VU, 4.5 ECTS)
- Master's Thesis (30.0 ECTS)