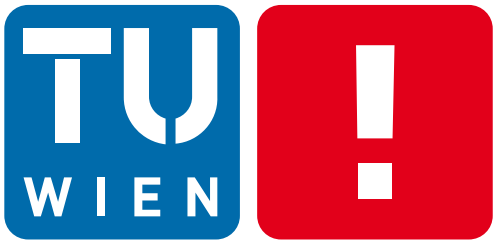
Advanced Multiprocessor Programming: Locks

Martin Wimmer, Jesper Larsson Träff

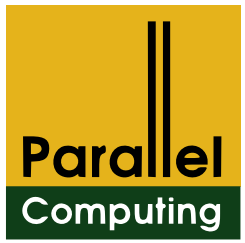
TU Wien

2nd May, 2016



FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics



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
- 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 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 ∞
  - compare-and-swap (aka: compare-exchange)
  - load-linked - store-conditional

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 );
```

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
```

Cost of atomic operations

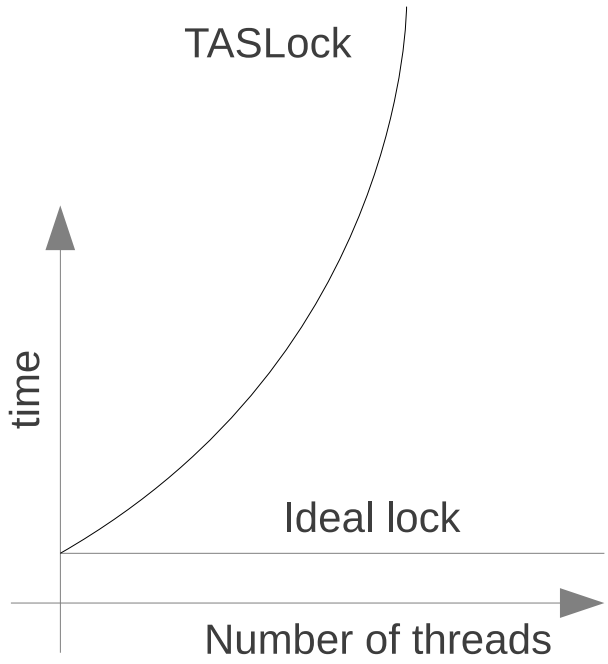
- All O(1), constant time, and wait-free instructions
- Reasonable for `exhange` and `fetch_add`?
  - These must always succeed, high contention could cause delays

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;
}
```

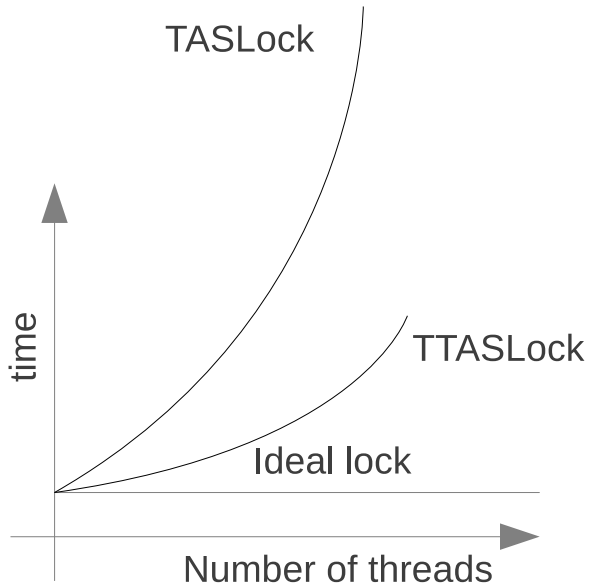
- Performance (surprisingly?) bad (why?)
  - Each test-and-set call invalidates cached copies for all threads
  - High contention on memory interconnect



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
- 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)
- 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: not starvation free

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:

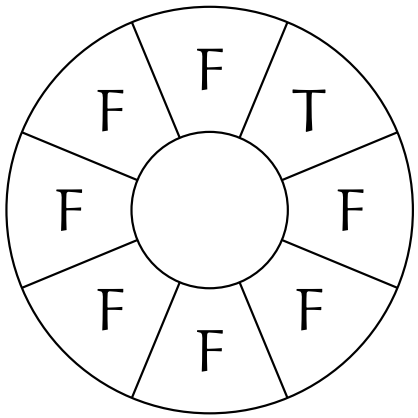
- Pure functions: no side-effects, no (global) state
- All functions called must likewise be thread-safe

Array Lock

- First come – first served
- Less contention
  - Each thread spins on a local copy of a variable
  - False sharing might occur, can be resolved with padding
- Not space efficient: $O(n)$ per lock!

```
bool flags[n] =
  {true, false, false, false, ...}
int tail = 0;
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;
}
```

$tail = \boxed{0}$

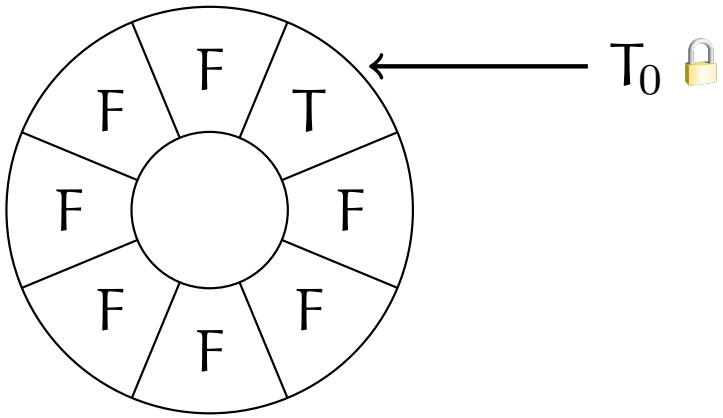


Array Lock

- First come – first served
- Less contention
  - Each thread spins on a local copy of a variable
  - False sharing might occur, can be resolved with padding
- Not space efficient: $O(n)$ per lock!

```
bool flags[n] =
  {true, false, false, false, ...}
int tail = 0;
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;
}
```

$tail = \boxed{1}$

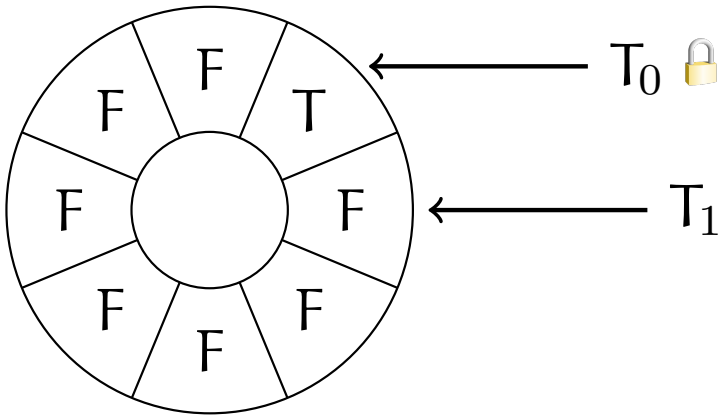


Array Lock

- First come – first served
- Less contention
  - Each thread spins on a local copy of a variable
  - False sharing might occur, can be resolved with padding
- Not space efficient: $O(n)$ per lock!

```
bool flags[n] =
  {true, false, false, false, ...}
int tail = 0;
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;
}
```

$\mathtt{tail} = \boxed{2}$

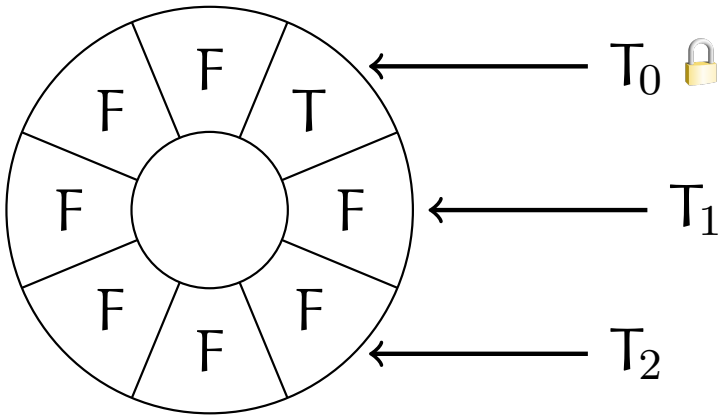


Array Lock

- First come – first served
- Less contention
  - Each thread spins on a local copy of a variable
  - False sharing might occur, can be resolved with padding
- Not space efficient: $O(n)$ per lock!

```
bool flags[n] =
  {true, false, false, false, ...}
int tail = 0;
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;
}
```

$\mathtt{tail} = \boxed{3}$

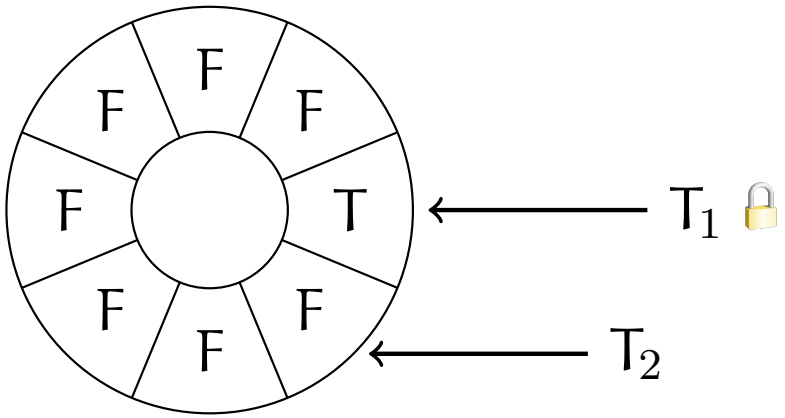


Array Lock

- First come – first served
- Less contention
  - Each thread spins on a local copy of a variable
  - False sharing might occur, can be resolved with padding
- Not space efficient: $O(n)$ per lock!

```
bool flags[n] =
  {true, false, false, false, ...}
int tail = 0;
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;
}
```

$\mathrm{tail} = \boxed{3}$

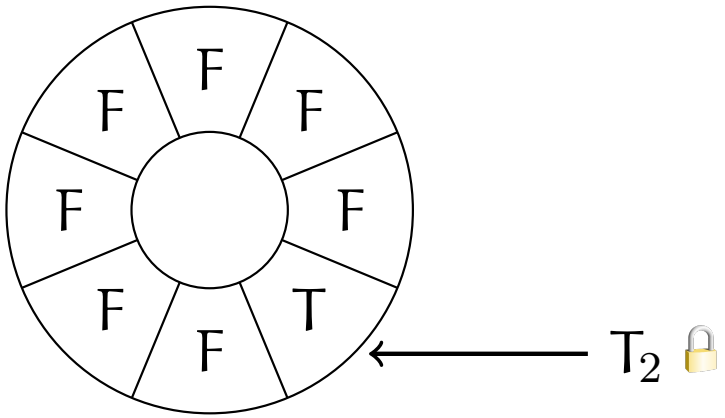


Array Lock

- First come – first served
- Less contention
  - Each thread spins on a local copy of a variable
  - False sharing might occur, can be resolved with padding
- Not space efficient: $O(n)$ per lock!

```
bool flags[n] =
  {true, false, false, false, ...}
int tail = 0;
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;
}
```

$\mathtt{tail} = \boxed{3}$

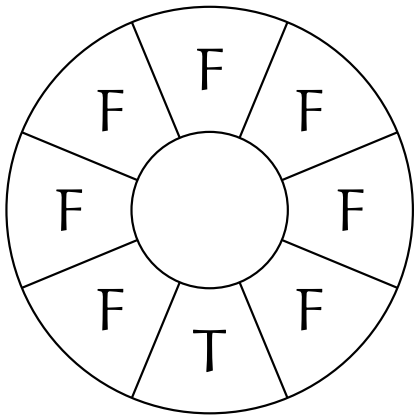


Array Lock

- First come – first served
- Less contention
  - Each thread spins on a local copy of a variable
  - False sharing might occur, can be resolved with padding
- Not space efficient: $O(n)$ per lock!

```
bool flags[n] =
  {true, false, false, false, ...}
int tail = 0;
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;
}
```

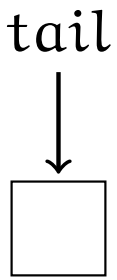
$\mathtt{tail} = \boxed{3}$



CLH Queue Lock

- Independently by Craig & Landin and Hagersten
- Linked list
- Single Sentinel node
- Spin on locked flag of previous node

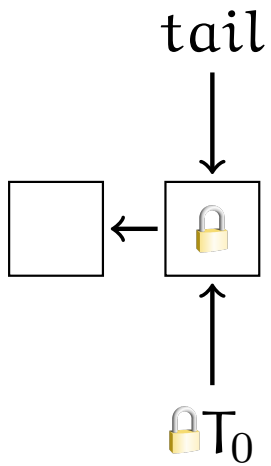
```
Node* tail = new Node();
tail->locked = false; // unlocked
thread_local Node* node;

void lock() {
  node = new Node();
  node->locked = true;
  node->pred = exchange(&tail,node);
  while (node->pred->locked) {}
}

void unlock() {
  delete node->pred;
  node->locked = false;
}
```



CLH Queue Lock

- Independently by Craig & Landin and Hagersten
- Linked list
- Single Sentinel node
- Spin on locked flag of previous node

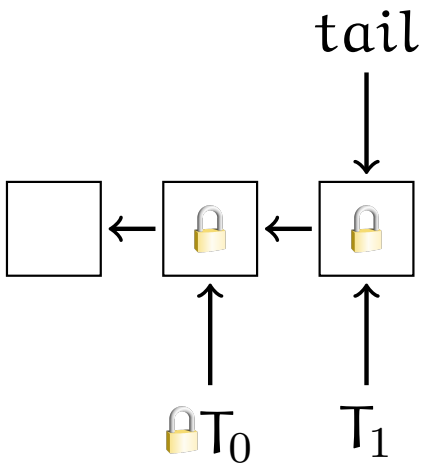
```
Node* tail = new Node();
tail->locked = false; // unlocked
thread_local Node* node;

void lock() {
  node = new Node();
  node->locked = true;
  node->pred = exchange(&tail,node);
  while (node->pred->locked) {}
}

void unlock() {
  delete node->pred;
  node->locked = false;
}
```



CLH Queue Lock

- Independently by Craig & Landin and Hagersten
- Linked list
- Single Sentinel node
- Spin on locked flag of previous node

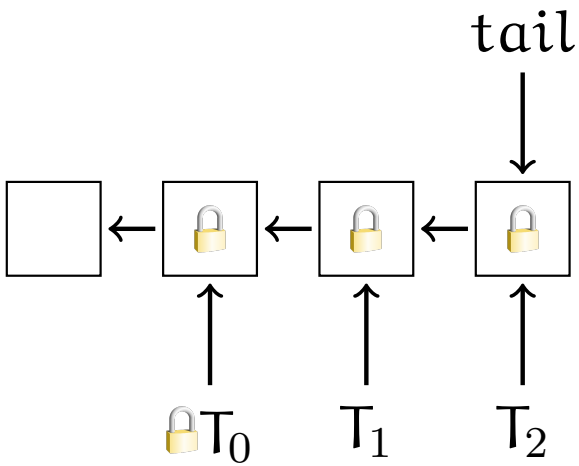
```
Node* tail = new Node();
tail->locked = false; // unlocked
thread_local Node* node;

void lock() {
  node = new Node();
  node->locked = true;
  node->pred = exchange(&tail,node);
  while (node->pred->locked) {}
}

void unlock() {
  delete node->pred;
  node->locked = false;
}
```



CLH Queue Lock

- Independently by Craig & Landin and Hagersten
- Linked list
- Single Sentinel node
- Spin on locked flag of previous node

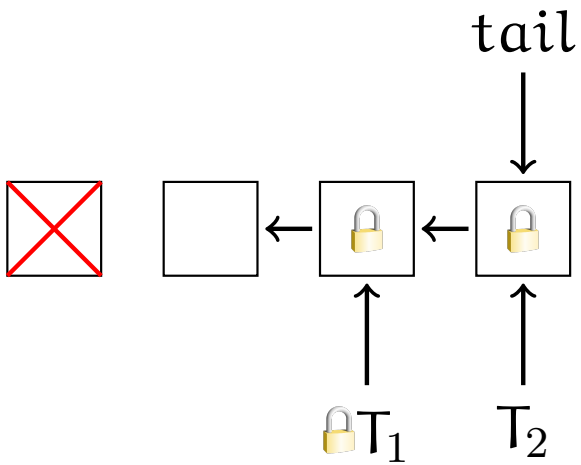
```
Node* tail = new Node();
tail->locked = false; // unlocked
thread_local Node* node;

void lock() {
  node = new Node();
  node->locked = true;
  node->pred = exchange(&tail,node);
  while (node->pred->locked) {}
}

void unlock() {
  delete node->pred;
  node->locked = false;
}
```



CLH Queue Lock

- Independently by Craig & Landin and Hagersten
- Linked list
- Single Sentinel node
- Spin on locked flag of previous node

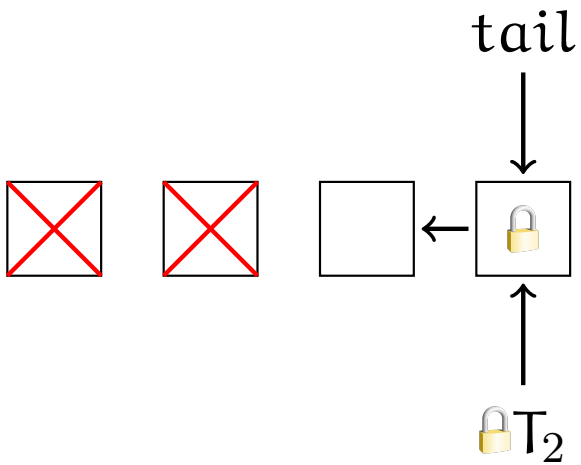
```
Node* tail = new Node();
tail->locked = false; // unlocked
thread_local Node* node;

void lock() {
  node = new Node();
  node->locked = true;
  node->pred = exchange(&tail,node);
  while (node->pred->locked) {}
}

void unlock() {
  delete node->pred;
  node->locked = false;
}
```



CLH Queue Lock

- Independently by Craig & Landin and Hagersten
- Linked list
- Single Sentinel node
- Spin on locked flag of previous node

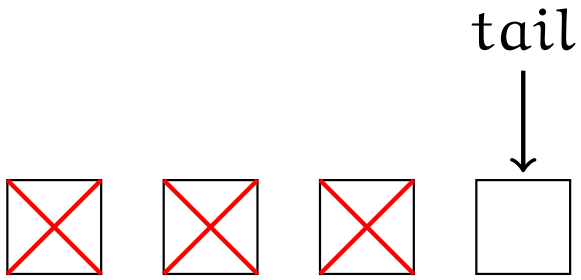
```
Node* tail = new Node();
tail->locked = false; // unlocked
thread_local Node* node;

void lock() {
  node = new Node();
  node->locked = true;
  node->pred = exchange(&tail,node);
  while (node->pred->locked) {}
}

void unlock() {
  delete node->pred;
  node->locked = false;
}
```



CLH Queue Lock

- Independently by Craig & Landin and Hagersten
- Linked list
- Single Sentinel node
- Spin on locked flag of previous node

```
Node* tail = new Node();
tail->locked = false; // unlocked
thread_local Node* node;

void lock() {
  node = new Node();
  node->locked = true;
  node->pred = exchange(&tail,node);
  while (node->pred->locked) {}
}

void unlock() {
  delete node->pred;
  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`)

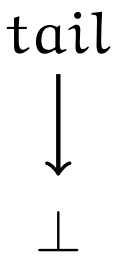
- Differs slightly from Herlihy/Shavit
  - Predecessor stored in node
  - Manual memory management
  - Predecessor can safely be deleted at unlock
    (no other thread accessing it)

CLH Queue Lock: Properties

- First-come-first-served
- 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

MCS Queue Lock

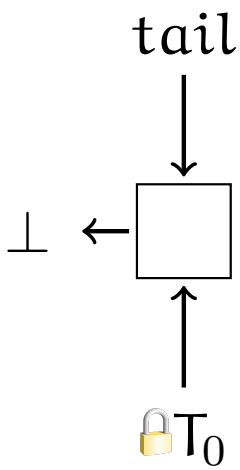
- Reverse list
- To unlock, modify locked field of next node
- If no successor exists, reset tail




```
Node* tail = nullptr;
thread_local Node* node = new Node();

void lock() {
  Node* my = node;
  Node* pred = exchange(&tail,my);
  if (pred != nullptr) {
    my->locked = true;
    pred->next = my;
    while (my->locked);
  }
}

void unlock() {
  Node* my = node;
  if (my->next == nullptr) {
    if (compare_exchange(&tail,my,
                         nullptr))
      return;
    // Wait for next thread
    while (my->next == nullptr);
  }
  my->next->locked = false;
  my->next = nullptr;
}
```

MCS Queue Lock

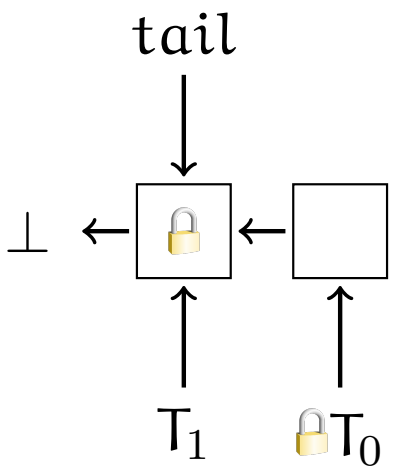
- Reverse list
- To unlock, modify locked field of next node
- If no successor exists, reset tail




```
Node* tail = nullptr;
thread_local Node* node = new Node();

void lock() {
  Node* my = node;
  Node* pred = exchange(&tail,my);
  if (pred != nullptr) {
    my->locked = true;
    pred->next = my;
    while (my->locked);
  }
}

void unlock() {
  Node* my = node;
  if (my->next == nullptr) {
    if (compare_exchange(&tail,my,
                         nullptr))
      return;
    // Wait for next thread
    while (my->next == nullptr);
  }
  my->next->locked = false;
  my->next = nullptr;
}
```

MCS Queue Lock

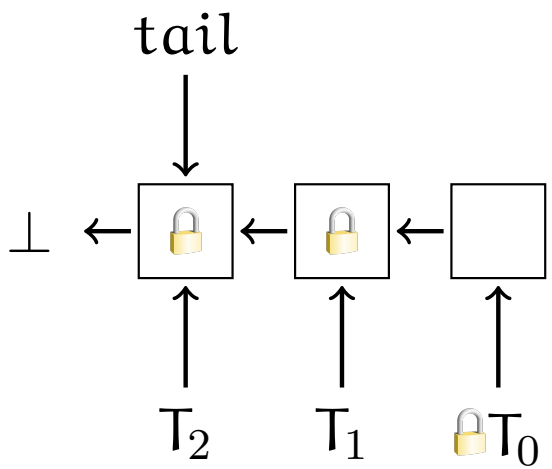
- Reverse list
- To unlock, modify locked field of next node
- If no successor exists, reset tail




```
Node* tail = nullptr;
thread_local Node* node = new Node();

void lock() {
  Node* my = node;
  Node* pred = exchange(&tail,my);
  if (pred != nullptr) {
    my->locked = true;
    pred->next = my;
    while (my->locked);
  }
}

void unlock() {
  Node* my = node;
  if (my->next == nullptr) {
    if (compare_exchange(&tail,my,
                         nullptr))
      return;
    // Wait for next thread
    while (my->next == nullptr);
  }
  my->next->locked = false;
  my->next = nullptr;
}
```

MCS Queue Lock

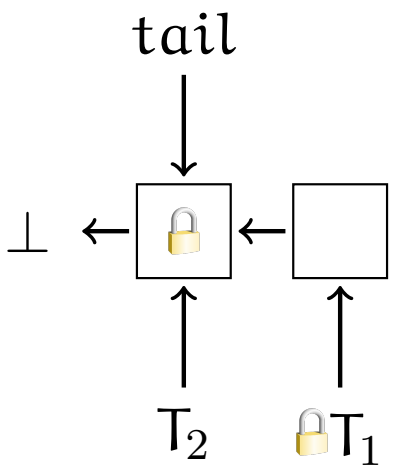
- Reverse list
- To unlock, modify locked field of next node
- If no successor exists, reset tail




```
Node* tail = nullptr;
thread_local Node* node = new Node();

void lock() {
  Node* my = node;
  Node* pred = exchange(&tail,my);
  if (pred != nullptr) {
    my->locked = true;
    pred->next = my;
    while (my->locked);
  }
}

void unlock() {
  Node* my = node;
  if (my->next == nullptr) {
    if (compare_exchange(&tail,my,
                         nullptr))
      return;
    // Wait for next thread
    while (my->next == nullptr);
  }
  my->next->locked = false;
  my->next = nullptr;
}
```

MCS Queue Lock

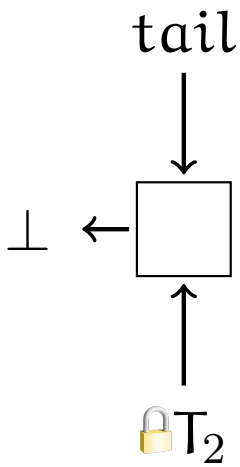
- Reverse list
- To unlock, modify locked field of next node
- If no successor exists, reset tail




```
Node* tail = nullptr;
thread_local Node* node = new Node();

void lock() {
  Node* my = node;
  Node* pred = exchange(&tail,my);
  if (pred != nullptr) {
    my->locked = true;
    pred->next = my;
    while (my->locked);
  }
}

void unlock() {
  Node* my = node;
  if (my->next == nullptr) {
    if (compare_exchange(&tail,my,
                         nullptr))
      return;
    // Wait for next thread
    while (my->next == nullptr);
  }
  my->next->locked = false;
  my->next = nullptr;
}
```

MCS Queue Lock

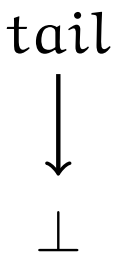
- Reverse list
- To unlock, modify locked field of next node
- If no successor exists, reset tail




```
Node* tail = nullptr;
thread_local Node* node = new Node();

void lock() {
  Node* my = node;
  Node* pred = exchange(&tail,my);
  if (pred != nullptr) {
    my->locked = true;
    pred->next = my;
    while (my->locked);
  }
}

void unlock() {
  Node* my = node;
  if (my->next == nullptr) {
    if (compare_exchange(&tail,my,
                         nullptr))
      return;
    // Wait for next thread
    while (my->next == nullptr);
  }
  my->next->locked = false;
  my->next = nullptr;
}
```

MCS Queue Lock

- Reverse list
- To unlock, modify locked field of next node
- If no successor exists, reset tail




```
Node* tail = nullptr;
thread_local Node* node = new Node();

void lock() {
  Node* my = node;
  Node* pred = exchange(&tail,my);
  if (pred != nullptr) {
    my->locked = true;
    pred->next = my;
    while (my->locked);
  }
}

void unlock() {
  Node* my = node;
  if (my->next == nullptr) {
    if (compare_exchange(&tail,my,
                         nullptr))
      return;
    // Wait for next thread
    while (my->next == nullptr);
  }
  my->next->locked = false;
  my->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 is not wait-free any more!
  (Waiting for next lock owner to set `next` pointer)

Space Requirements in Array and Queue Locks

- These locks are starvation free (first-come, first serve 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)

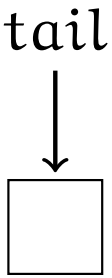
- Abandoning is easy for Test-And-Set lock
  - Just stop trying to acquire lock
  - Timing out is wait-free
- More difficult for queue locks
  - If we just exit, the following thread will starve
  - Can't just unlink the node

    (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)

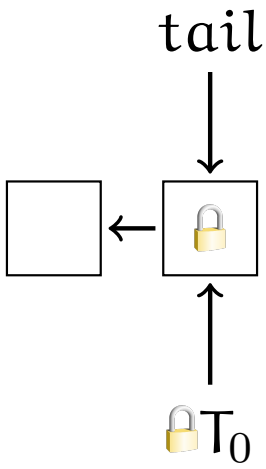
- Add flag **abandoned** to nodes
- To abandon, set flag to **true**
- Next node is responsible for clean-up




```
Node* tail = new Node();
thread_local Node* node;

bool try_lock(int timeout) {
  node = new Node();
  node->locked = true;
  node->abandoned = false;
  node->pred = exchange(&tail,node);
  int start = time();
  while (node->pred->locked) {
    if (node->pred->abandoned) {
      Node* pp = node->pred->pred;
      delete node->pred;
      node->pred = pp;
    }
    if (start + timeout <= time() {
      node->abandoned = true;
      return false;
    }
  }
  return true;
}

void unlock() {
  delete node->pred;
  node->locked = false;
}
```

Queue Locks with timeouts (here: CLH Queue Lock)

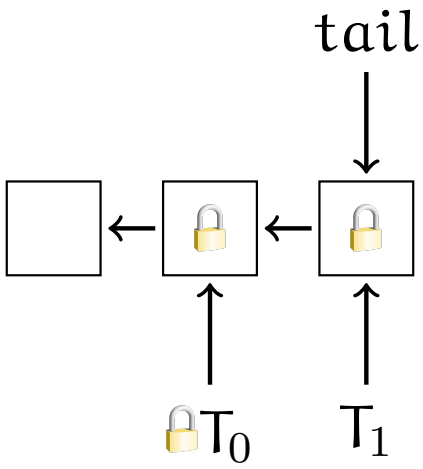
- Add flag `abandoned` to nodes
- To abandon, set flag to `true`
- Next node is responsible for clean-up




```
Node* tail = new Node();
thread_local Node* node;

bool try_lock(int timeout) {
  node = new Node();
  node->locked = true;
  node->abandoned = false;
  node->pred = exchange(&tail,node);
  int start = time();
  while (node->pred->locked) {
    if (node->pred->abandoned) {
      Node* pp = node->pred->pred;
      delete node->pred;
      node->pred = pp;
    }
    if (start + timeout <= time() {
      node->abandoned = true;
      return false;
    }
  }
  return true;
}

void unlock() {
  delete node->pred;
  node->locked = false;
}
```

Queue Locks with timeouts (here: CLH Queue Lock)

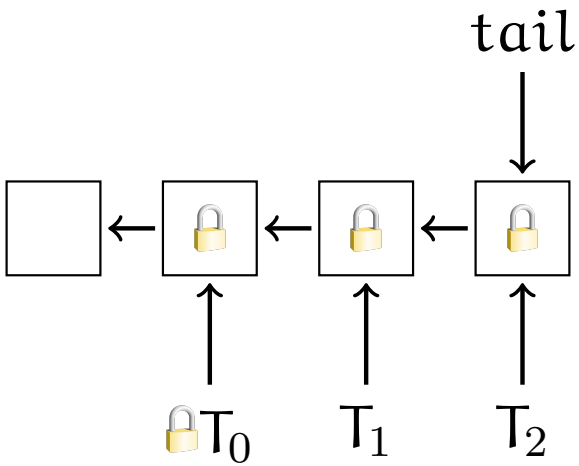
- Add flag **abandoned** to nodes
- To abandon, set flag to **true**
- Next node is responsible for clean-up




```
Node* tail = new Node();
thread_local Node* node;

bool try_lock(int timeout) {
  node = new Node();
  node->locked = true;
  node->abandoned = false;
  node->pred = exchange(&tail,node);
  int start = time();
  while (node->pred->locked) {
    if (node->pred->abandoned) {
      Node* pp = node->pred->pred;
      delete node->pred;
      node->pred = pp;
    }
    if (start + timeout <= time() {
      node->abandoned = true;
      return false;
    }
  }
  return true;
}

void unlock() {
  delete node->pred;
  node->locked = false;
}
```

Queue Locks with timeouts (here: CLH Queue Lock)

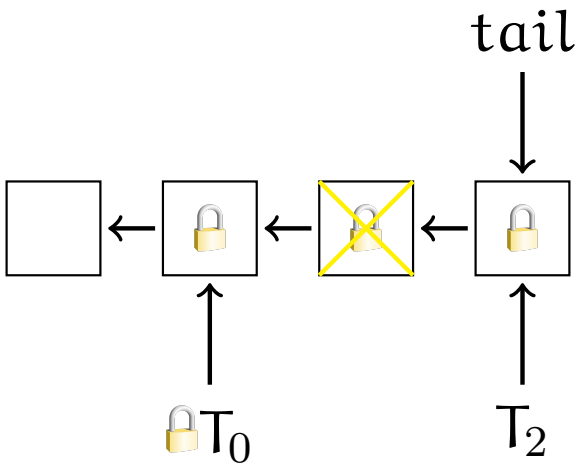
- Add flag **abandoned** to nodes
- To abandon, set flag to **true**
- Next node is responsible for clean-up




```
Node* tail = new Node();
thread_local Node* node;

bool try_lock(int timeout) {
  node = new Node();
  node->locked = true;
  node->abandoned = false;
  node->pred = exchange(&tail,node);
  int start = time();
  while (node->pred->locked) {
    if (node->pred->abandoned) {
      Node* pp = node->pred->pred;
      delete node->pred;
      node->pred = pp;
    }
    if (start + timeout <= time() {
      node->abandoned = true;
      return false;
    }
  }
  return true;
}

void unlock() {
  delete node->pred;
  node->locked = false;
}
```

Queue Locks with timeouts (here: CLH Queue Lock)

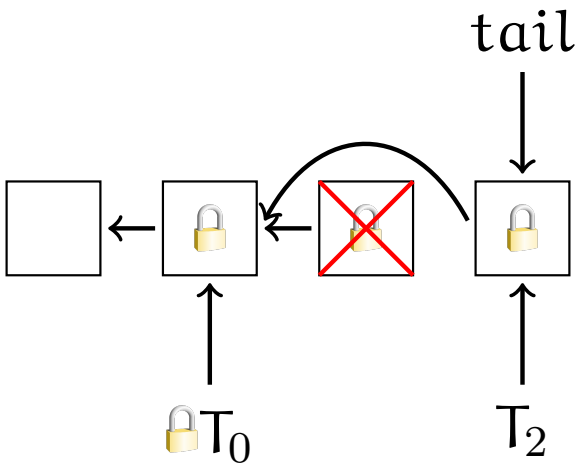
- Add flag **abandoned** to nodes
- To abandon, set flag to **true**
- Next node is responsible for clean-up




```
Node* tail = new Node();
thread_local Node* node;

bool try_lock(int timeout) {
  node = new Node();
  node->locked = true;
  node->abandoned = false;
  node->pred = exchange(&tail,node);
  int start = time();
  while (node->pred->locked) {
    if (node->pred->abandoned) {
      Node* pp = node->pred->pred;
      delete node->pred;
      node->pred = pp;
    }
    if (start + timeout <= time() {
      node->abandoned = true;
      return false;
    }
  }
  return true;
}

void unlock() {
  delete node->pred;
  node->locked = false;
}
```

Queue Locks with timeouts (here: CLH Queue Lock)

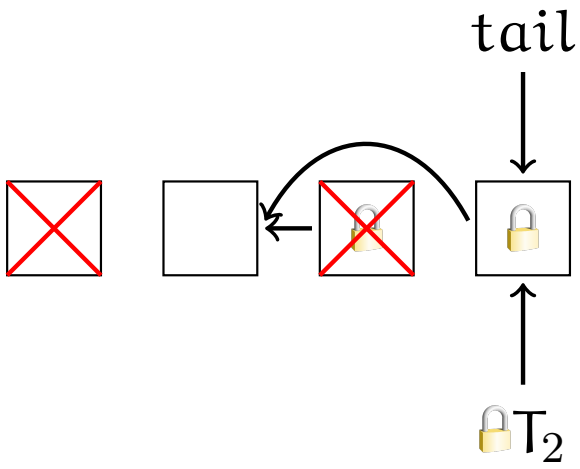
- Add flag **abandoned** to nodes
- To abandon, set flag to **true**
- Next node is responsible for clean-up




```
Node* tail = new Node();
thread_local Node* node;

bool try_lock(int timeout) {
  node = new Node();
  node->locked = true;
  node->abandoned = false;
  node->pred = exchange(&tail,node);
  int start = time();
  while (node->pred->locked) {
    if (node->pred->abandoned) {
      Node* pp = node->pred->pred;
      delete node->pred;
      node->pred = pp;
    }
    if (start + timeout <= time() {
      node->abandoned = true;
      return false;
    }
  }
  return true;
}

void unlock() {
  delete node->pred;
  node->locked = false;
}
```

Queue Locks with timeouts (here: CLH Queue Lock)

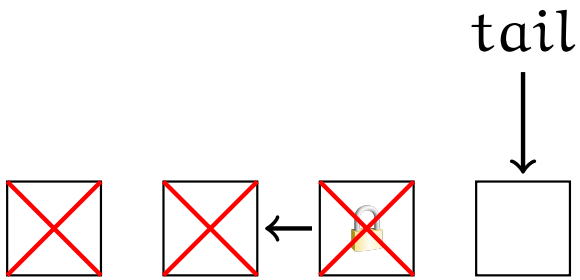
- Add flag **abandoned** to nodes
- To abandon, set flag to **true**
- Next node is responsible for clean-up




```
Node* tail = new Node();
thread_local Node* node;

bool try_lock(int timeout) {
  node = new Node();
  node->locked = true;
  node->abandoned = false;
  node->pred = exchange(&tail,node);
  int start = time();
  while (node->pred->locked) {
    if (node->pred->abandoned) {
      Node* pp = node->pred->pred;
      delete node->pred;
      node->pred = pp;
    }
    if (start + timeout <= time() {
      node->abandoned = true;
      return false;
    }
  }
  return true;
}

void unlock() {
  delete node->pred;
  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

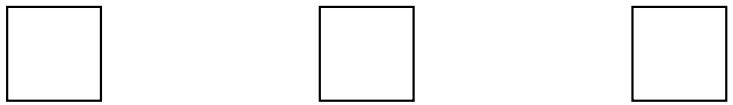



```
Node* tail = new Node();
thread_local Node* node;

bool try_lock(int timeout) {
  node = new Node();
  node->locked = true;
  node->abandoned = false;
  node->pred = exchange(&tail,node);
  int start = time();
  while (node->pred->locked) {
    if (node->pred->abandoned) {
      Node* pp = node->pred->pred;
      delete node->pred;
      node->pred = pp;
    }
    if (start + timeout <= time() {
      node->abandoned = true;
      return false;
    }
  }
  return true;
}

void unlock() {
  delete node->pred;
  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

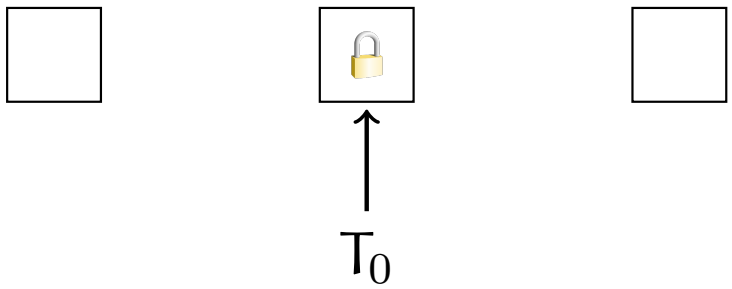
```
Node nodes[k]; // preallocated

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

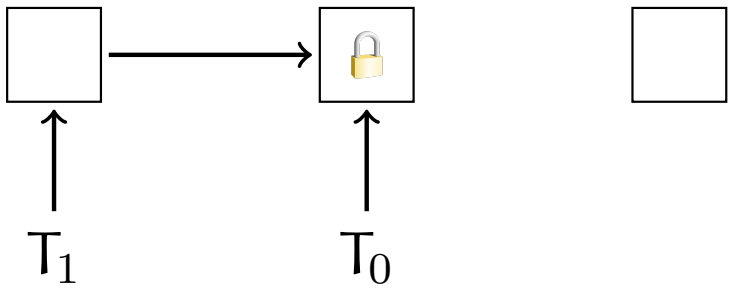
```
Node nodes[k]; // preallocated

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

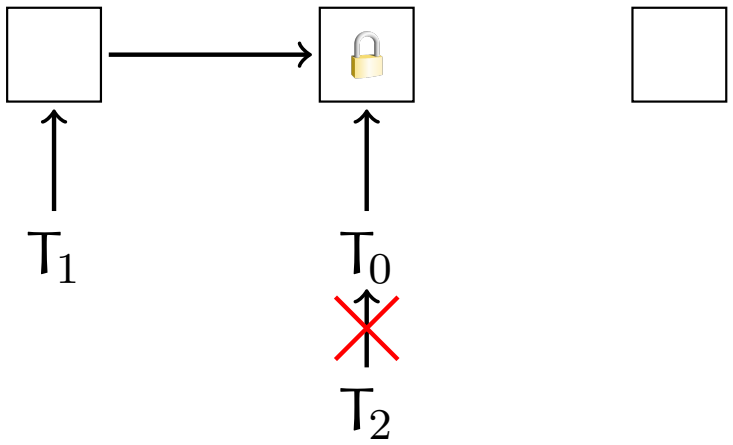
```
Node nodes[k]; // preallocated

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

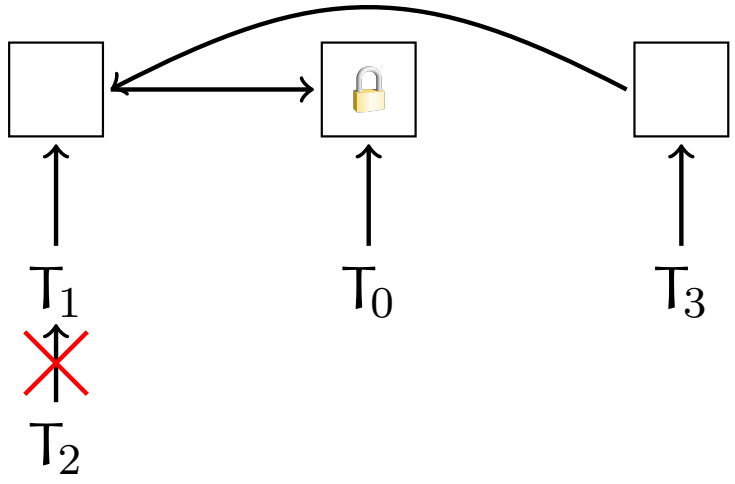
```
Node nodes[k]; // preallocated

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

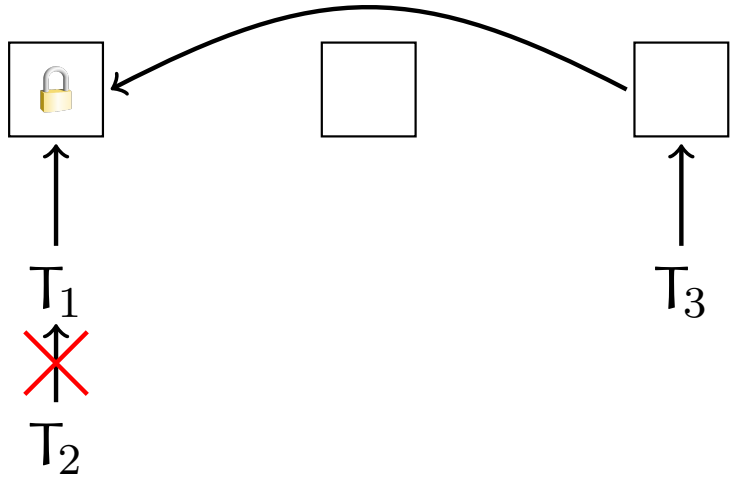
```
Node nodes[k]; // preallocated

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

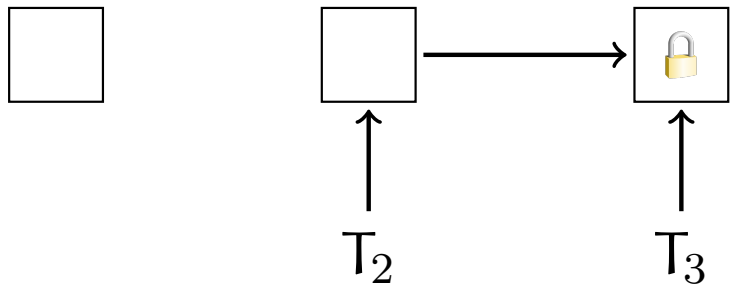
```
Node nodes[k]; // preallocated

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

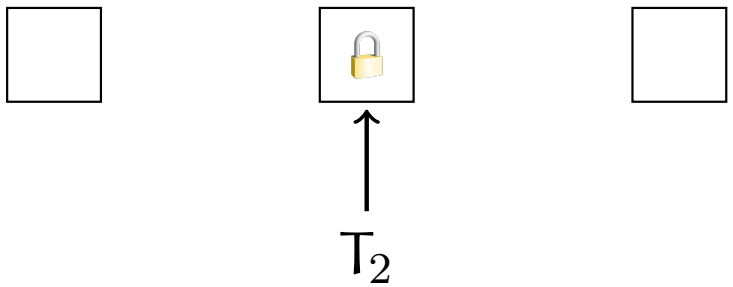
```
Node nodes[k]; // preallocated

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

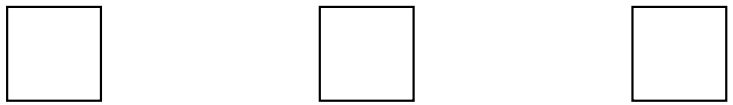
```
Node nodes[k]; // preallocated

void lock() {
  Node* node;
  while(!(node =
       acquireNode(rnd() % k)))
    backoff();

  enqueueNode(node);
  waitForPredecessor(node);
}
```



Composite Lock

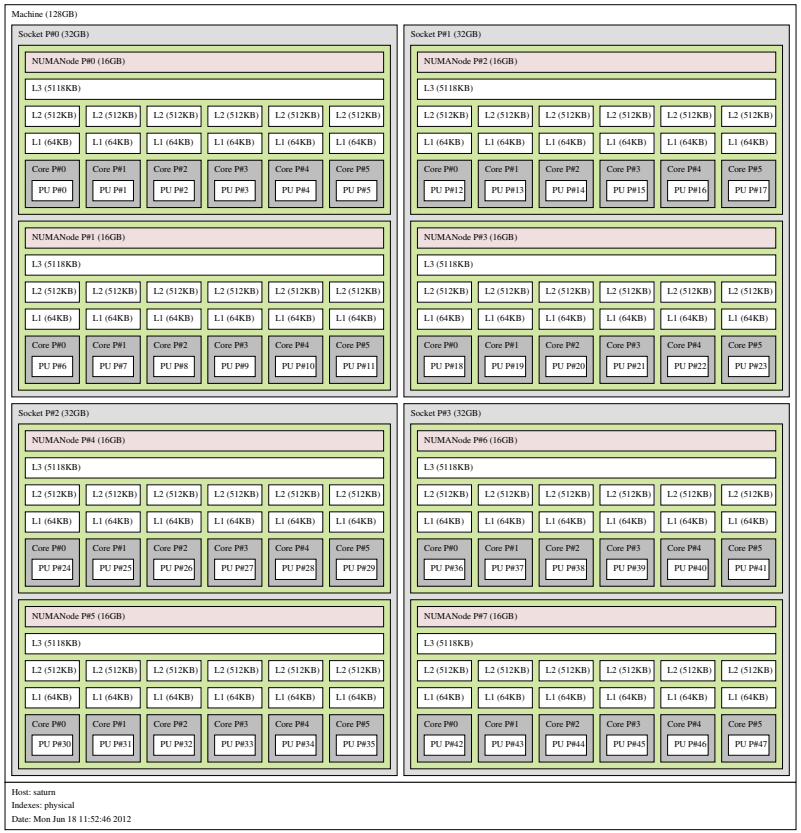
- Combines Backoff lock and Queue lock
- Preallocate fixed number of nodes $< n$
- Acquire a random node
  - Only one thread may use a certain node
  - On failure back off
- As soon as a node is acquired, enqueue it
- Can be augmented with a fast path for low contention
  - If queue is empty, try fast 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;
  while ((current_owner =
          compare_exchange(&owner,
                           &noowner,
                                 thread_id
                                 ()))
          != thread_id()) {
    int distance =
      memory_distance(current_owner,
            thread_id());
    backoff(distance);
  }
}
```

Hierarchical CLH Lock

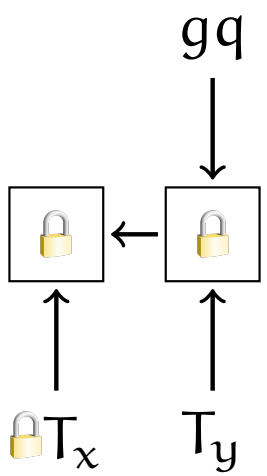
- By Victor Luchangco, Dan Nussbaum and Nir Shavit

- 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;
  }

  // Thre 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



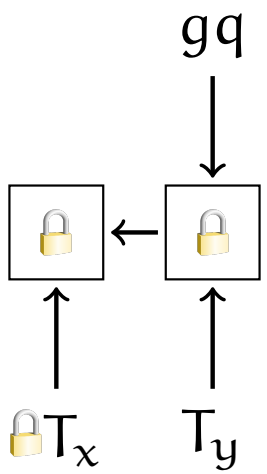
lq_1

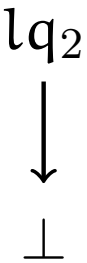
$\perp$

lq_2

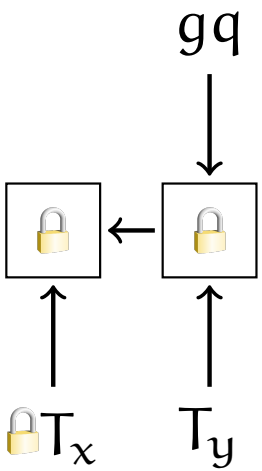
$\perp$

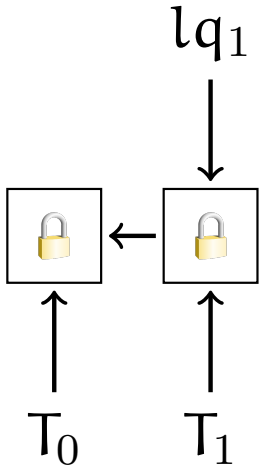
Hierarchical CLH Lock



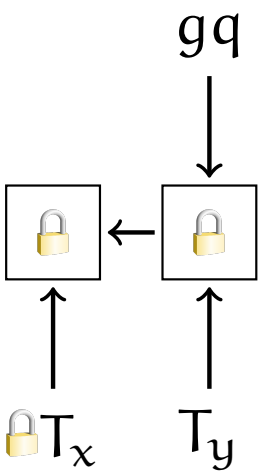


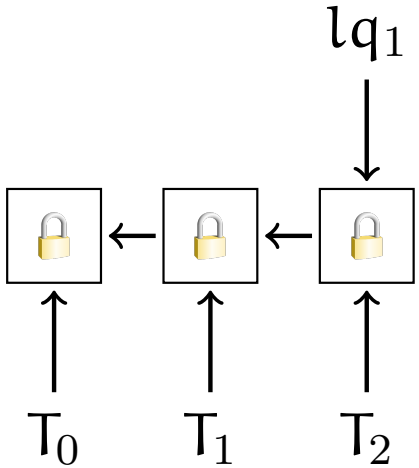
Hierarchical CLH Lock

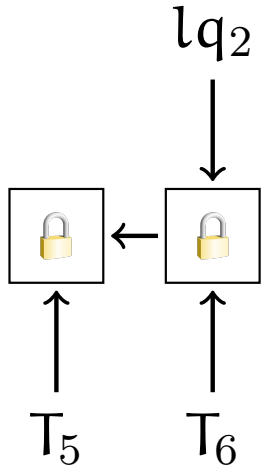




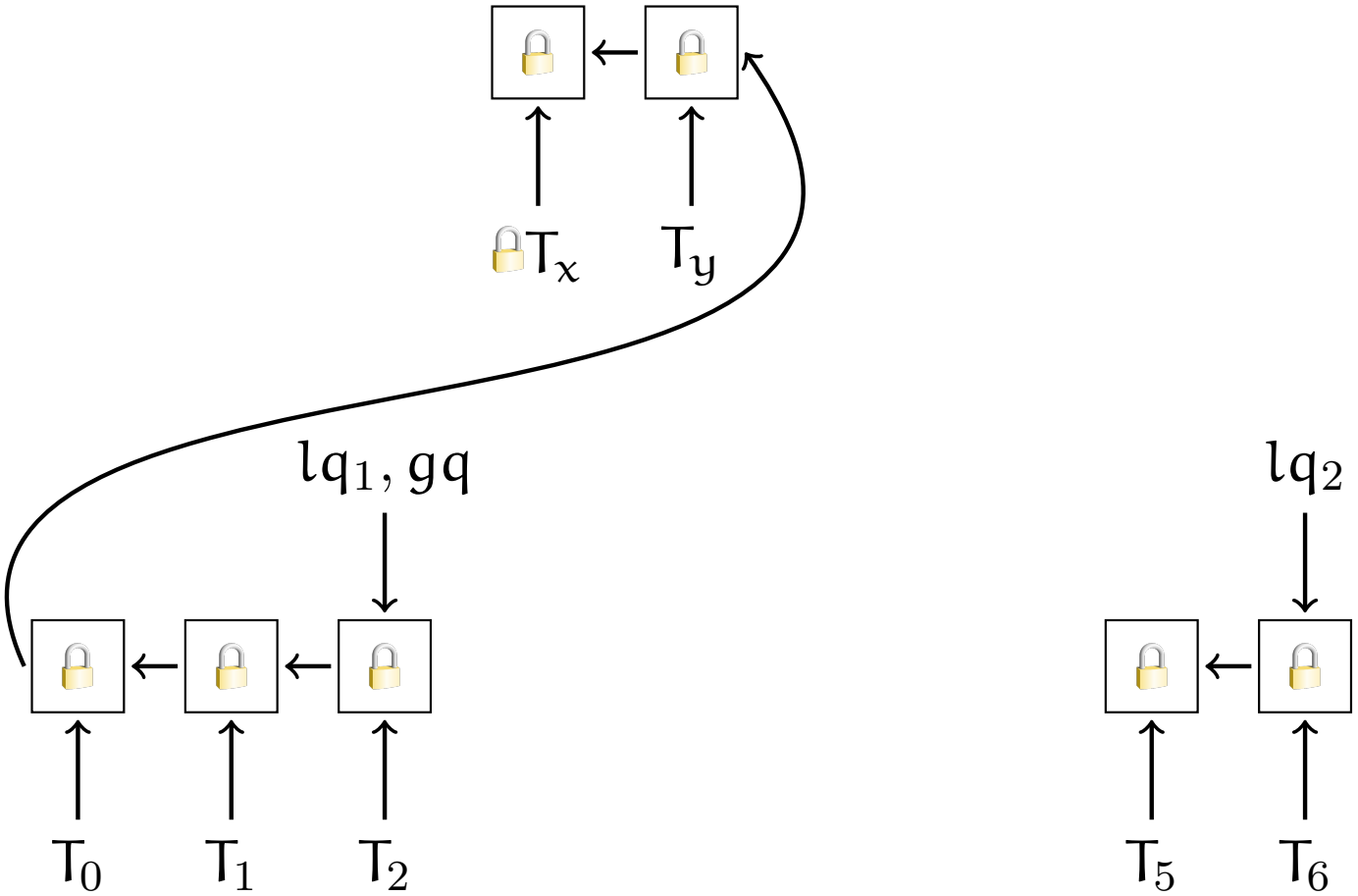
Hierarchical CLH Lock



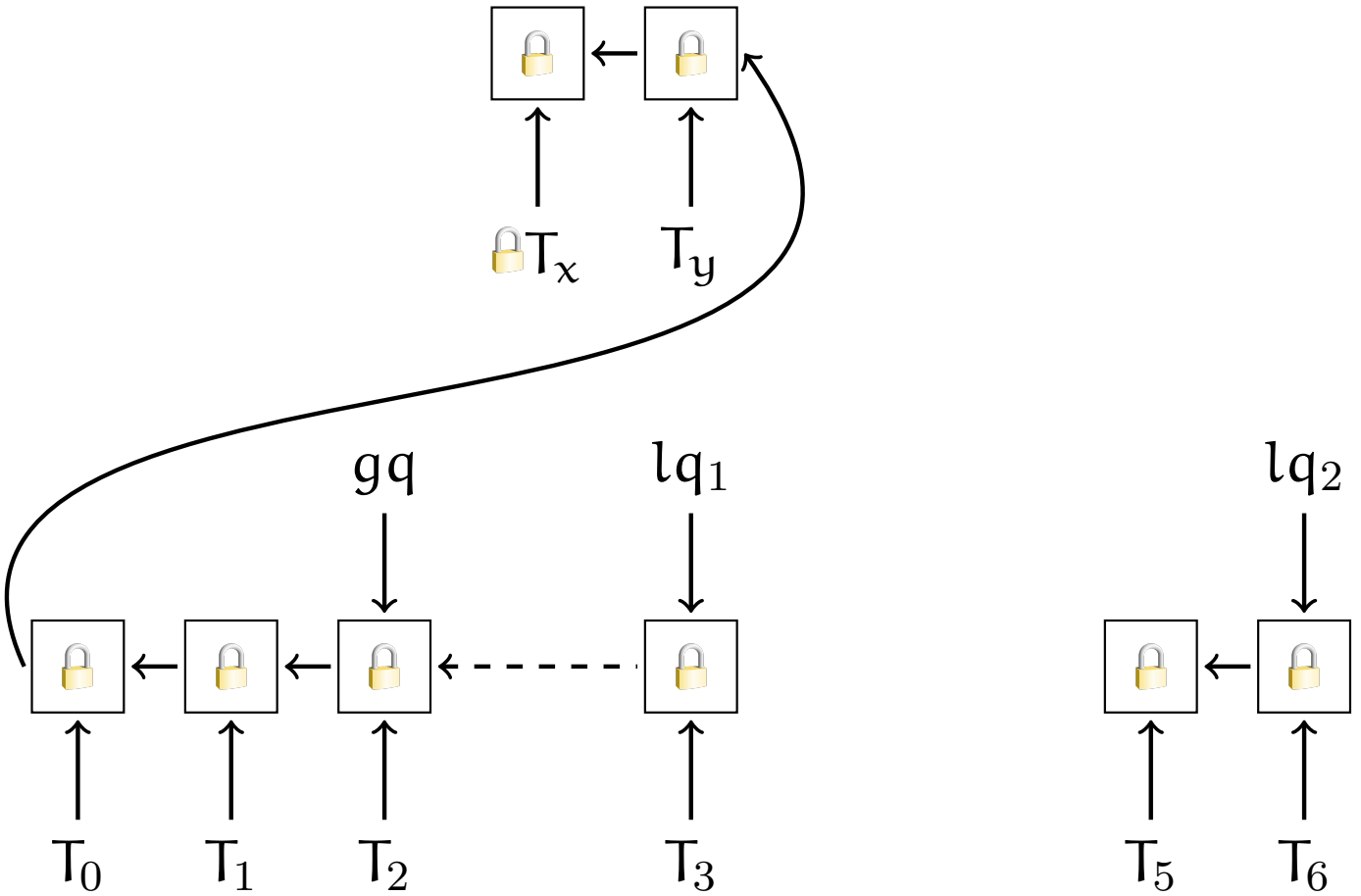




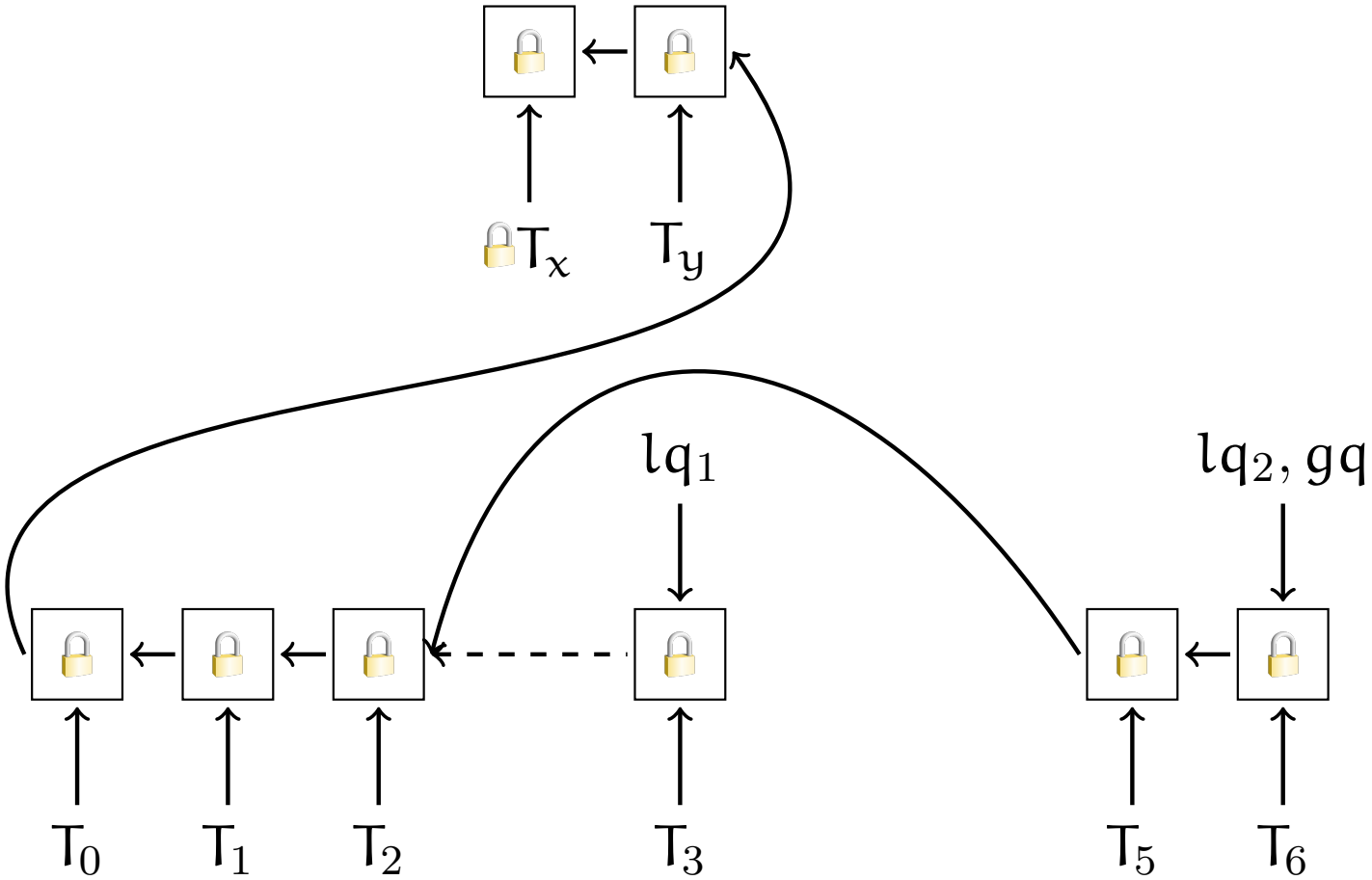
Hierarchical CLH Lock



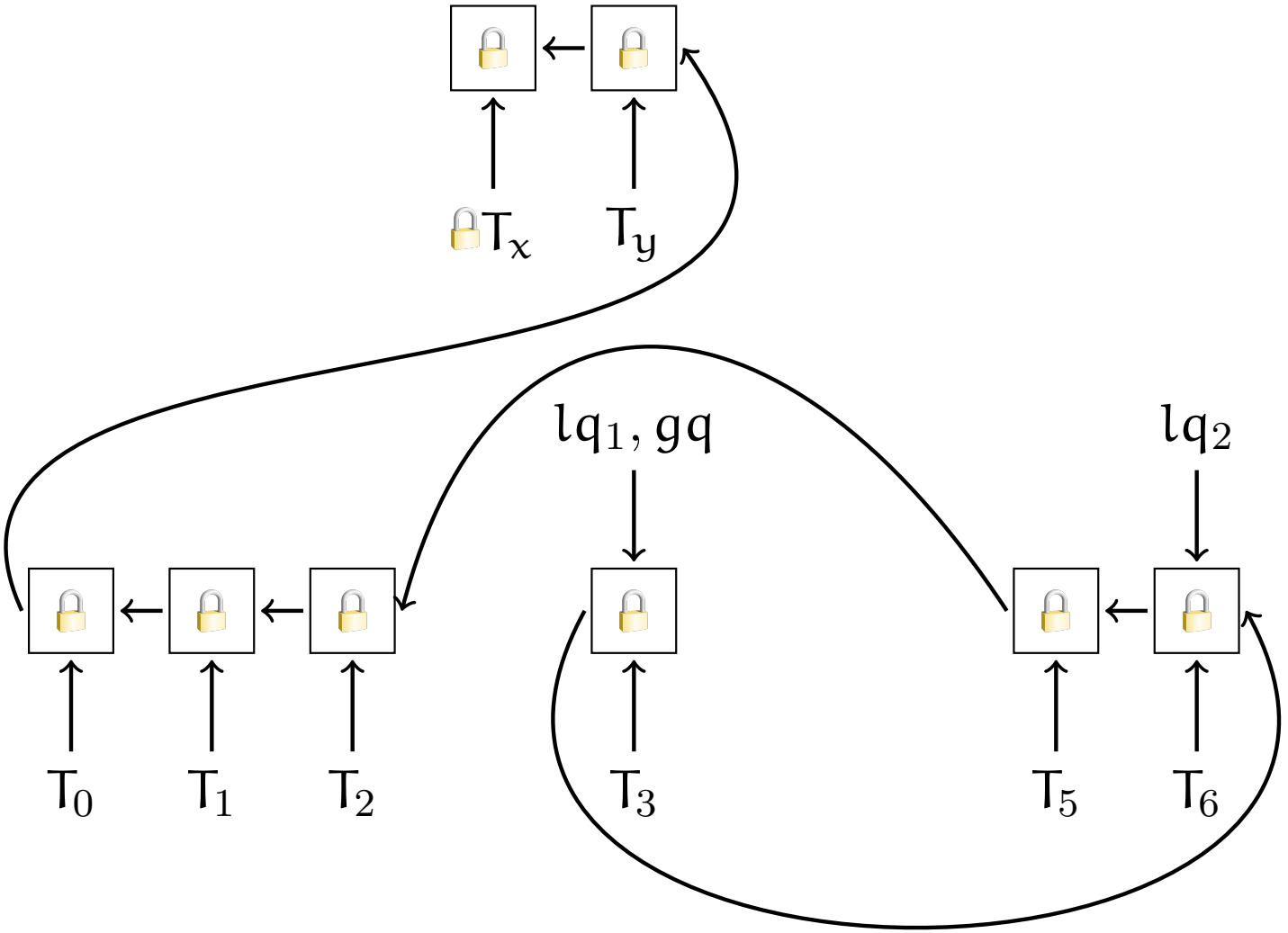
Hierarchical CLH Lock



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.