Advanced Multiprocessor Programming

Practical lock implementations

Martin Wimmer
martin.wimmer@tuwien.ac.at
Research Group Parallel Computing
Faculty of Informatics, Institute of Information Systems
Vienna University of Technology (TU Wien)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Outline of this lecture

- General lock structure
- Bakery lock revisited
  - Common pitfalls in synchronization algorithms & motivating example
  - The unbounded timestamp problem
- Consensus (again)
- Locking algorithms based on higher consensus operations

Locks

- General structure
  - Lock operation
  - Unlock operation
    - Make sure unlock is always called even on errors
      - Java-style: in finally block
      - C++ style: RAII

```
// Java style
mutex.lock();
try {
  // critical section
} finally {
  mutex.unlock();
}
```

- General distinction
  - Spinning/busy-waiting
    - We will concentrate on those
  - Blocking

```
{ // C++ style
  std::lock_guard<std::mutex>
lock(mutex);
  // critical section
}
```

Bakery lock revisited

- As presented in previous lecture
  - Algorithm from Herlihy/Shavit book
  - Not the original Bakery algorithm
- Maintain first-come-first-served property
  - Idea: Number-dispensing machine
    - (like in bakeries)
  - Take a number. When your number is up, it is your turn.

```
// Initialize to false
bool flag[n];
// Initialize to 0
int label[n];

void lock() {
  int me = thread_id();
  flag[me] = true;
  // Acquire ticket
  label[me] =
     max(label[0], … label[n - 1]);

  // Wait until no
  // lower ticket exists
  for(int i = 0; i < n; ++i) {
     while(i != me && flag[i] &&
       (label[i] < label[me] ||
         // On same label, take
         // thread with lower id
         (label[i] == label[me] &&
          i < me)));
  }
}

void unlock() {
  int me = thread_id();
  flag[me] = false;
}
```

Bakery lock mutual exclusion proof

- Proposition: The Bakery lock satisfies mutual exclusion

- Proof: By contradiction (from previous lecture)
  - labeling(A), labeling(B)
    - sequences of instructions generating the labels.
  - Assume (label[A],A)<<(label[B],B). When B entered it must therefore have read flag[A]==false

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

  - This contradicts (label[A],A)<<(label[B],B) since A‘s label would be at least label[B]+1

Bakery lock proof

- Why do we revisit the proof again?

Bakery lock proof

- Why do we revisit the proof again?

- Because we forgot something
- The proof is not correct!
  - Fortunately the algorithm still seems to be correct after we correct the proof

Where is the error?

- Proposition: The Bakery lock satisfies mutual exclusion

- Proof: By contradiction (from previous lecture)
  - labeling(A), labeling(B)
    - sequences of instructions generating the labels.
  - Assume (label[A],A)<<(label[B],B). When B entered it must therefore have read flag[A]==false

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

  - This contradicts (label[A],A)<<(label[B],B) since A‘s label would be at least label[B]+1

The error

- Labelling is not atomic!
  - Proof ignores this fact
  - Can be split into multiple reads and a write operation

```
flag[me] = true;

// read labels
int max = label[0];
for(int i = 1; i < n; ++i) {
  if(label[i] > max)
     max = label[i];
}

// write own label
label[me] = max + 1;

// Waiting part
...
```

- This is therefore not completely correct:
  - Assume (label[A],A)<<(label[B],B). When B entered it must therefore have read flag[A]==false

    labeling(B) → read(B,flag[A]==false) → write(A,flag[A]=true) → labeling(A)
  - This path may also occur

    write(B, flag[B] = true) → read(B, labels) → write(A, flag[A]=true) → read(A, labels) → write(B, label[B]) → read(B, flag[A] == true) → read(B, label[A]) → write(A, label[A])
  - This doesn't change correctness in this case
    - As label[A] must be smaller than the new value, label[A] < label[B] still holds
    - Still very dangerous to ignore such facts

Common pitfalls in synchronization algorithms

- Forgetting that some operation isn't atomic
  - As in the Bakery lock example
- Performing a check, and then assuming that the condition holds for the rest of the algorithm
- Overflow of timestamps
  - Next slide
- Assuming sequential consistency
  - We will come to this later

- Those problems are quite common
  - Might rarely lead to errors in practice
  - Random crashes that are very difficult to track
  - Trial and error is a bad idea => Make sure your algorithms are correct

Problem with unbounded registers

- Unbounded registers don't exist!
- Labels in Bakery algorithm may overflow, invalidating the first-come-first-served property
  - Common problem in synchronization algorithms

- Possible solutions
  - Ignore problem
  - make size of label large enough
    - e.g. on 64 bit systems
  - Simple hack:
    - use unsigned data-type
    - instead of checking A > B check A – B > 0
    - Works well for many algorithms in practice, but may still fail in theory
    - Bad for Bakery, if some threads rarely access the lock
  - Construct unbounded timestamps
    - Cyclic graph providing an ordering of timestamps
    - May grow very large for larger numbers of threads
    - Quite complex to implement
  - Black-white bakery lock

Original Bakery lock

- Original Bakery algorithm
  - Flag field used to “lock” own label
- While thread is labelling, other threads wait for the updated value
  - This would allow us to treat labelling as atomic in the proof

    (at least for this special case)
- Special value for label for unlocked state
  - To unlock we reset label
- We won't cover proof here

```
// Initialize to false
bool flag[n];
// Initialize to 0
int label[n];

void lock() {
  int me = thread_id();
  flag[me] = true;
  // Acquire ticket
  label[me] =
     max(label[0], … label[n - 1]);
  flag[me] = false;

  // Wait until no
  // lower ticket exists
  for(int i = 0; i < n; ++i) {
     // Make sure thread is
     // finished chosing a label
     while(flag[i]);
     while(i != me &&
       label[i] != 0 &&
       (label[i] > label[me] ||
         // On same label, take
         // thread with lower id
         (label[i] == label[me] &&
          i < me)));
  }
}

void unlock() {
  int me = thread_id();
  label[me] = 0;
}
```

Black-white bakery lock

- Have a shared coloring bit
- Store color in addition to label
- Threads with color different to current bit come first
- After thread exits critical section, change shared coloring bit to color different to own color

- Guarantees first-come-first-served
- Only needs n values for label

```
void lock() {
  ...
  flag[me] = true;
  mycolor[me] = color;
  label[me] = ...
  flag[me] = false;

  for(int i = 0; i < n; ++i) {
     // Make sure thread is
     // finished chosing a label
     while(flag[i]);
     if(i != me) {
     if(mycolor[me] == mycolor[i]) {
       while([other wins] and
          mycolor[me] ==
            mycolor[i]);
     } else {
       while([other active] and
          mycolor[me] == color and
          mycolor[me] ==
            mycolor[i]);
     }}
  }
}

void unlock() {
  int me = thread_id();
  color = !mycolor[me];
  label[me] = 0;
}
```

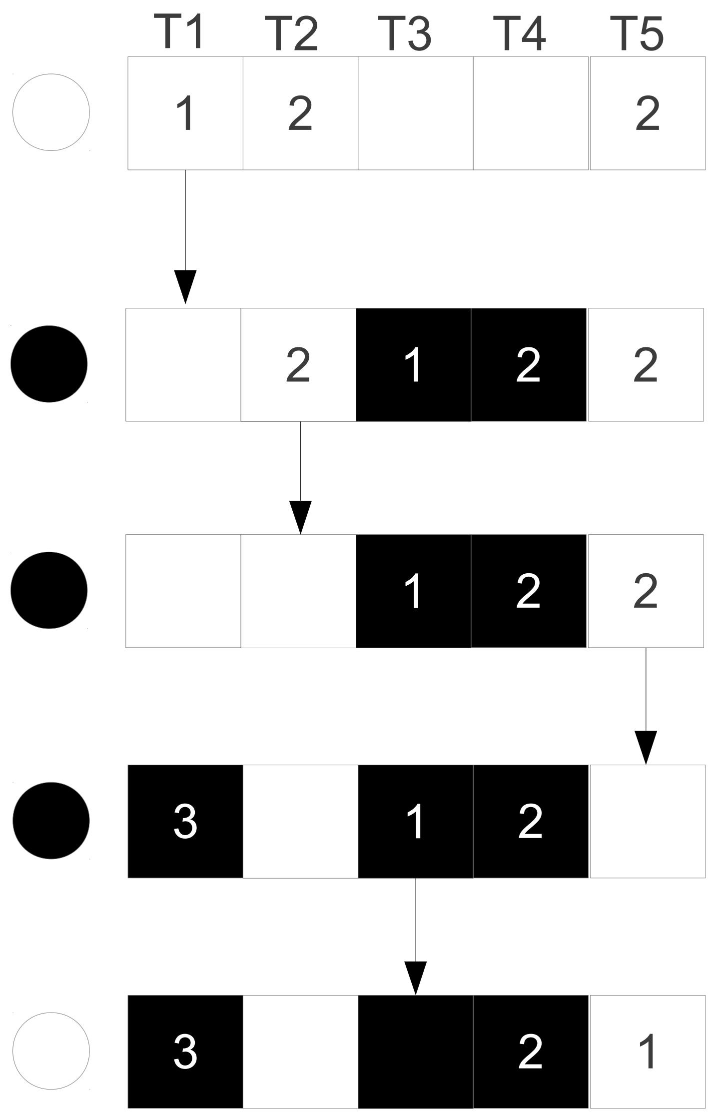
Waiting conditions

- Other thread has same color
  - Similar waiting conditions to original algorithm
  - If other thread changes color, we can also proceed

```
...
if(mycolor[me] == mycolor[i]) {
  // Same color case
  while(label[i] != 0 &&
      (label[i] < label[me] ||
      (label[i] == label[me] &&
        i < me)) &&
       mycolor[me] == mycolor[i]);
}
else {
  // Colors differ
  while(label[i] != 0 &&
       mycolor[me] == color &&
       mycolor[i] != mycolor[me]);
}
...
```

- Different color
  - Wait if my color equals shared color
  - Wait until
    - shared color changes
    - other thread unlocks
    - other thread changes colors

Black-white bakery lock



Still, the Bakery lock isn't practical

- O(n) execution time
- O(n) memory

- Can this be improved based on reading and writing memory?
  - No!
  - A lock for n threads needs at least n locations

    (Recall proof in second lecture)

- Peterson and Bakery lock are thus optimal!

  (but not practical)

Consensus

- Wait-free N-thread consensus protocol
  - Agree on a common value between n threads
  - Has to be wait free
    - This means the result may need to be decided even before some threads enter the protocol!
- Consensus number
  - A construction with consensus number n can be used to implement wait-free consensus for n threads

Consensus numbers

(Recall third lecture)

- Consensus number 1
  - Atomic registers
- Consensus number 2
  - Wait-free FIFO Queues
  - get-and-set
  - test-and-set
  - fetch-and-add
- Consensus number ∞
  - compare-and-swap (CAS)
  - load-linked + store conditional

Putting higher consensus operations to use

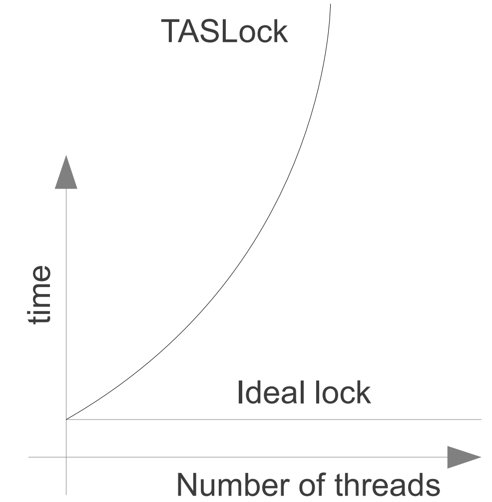
- Test-And-Set Lock
  - Single flag field
  - atomically set field to true if it isn't already
    - On success we have the lock
  - For unlock just reset the field

```
bool locked = false;

void lock() {
  // test_and_set checks whether
  // variable is set to false
  // and atomically sets it to
  // true
  while(!test_and_set(locked));
}

void unlock() {
  locked = false;
}
```

- Performance far from ideal
  - Each test-and-set call invalidates cached copies for all threads
  - High contention of interconnect



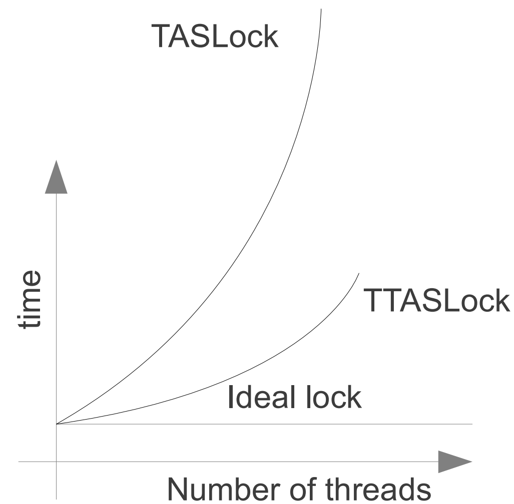
Test-And-Test-And-Set lock

- Only perform test-and-set if there is a chance of success
- Cache invalidated less often
- Still some contention with more threads

```
bool locked = false;

void lock() {
  while(true) {
     while(locked);
     if(test_and_set(locked))
       return true;
  }
}

void unlock() {
  locked = false;
}
```



Exponential Backoff

- On failure to acquire lock:

  Backoff for a random amount of time

- Time to wait increases exponentially
- Reduces contention
  - On high contention threads try less often
  - Randomization makes sure threads wake up at different times
- Thread might wait longer than necessary!
- C++ note: Don't use the standard rand()

  It uses locks inside!

```
class Backoff {
  int limit = MIN_DELAY;
  void backoff() {
     int delay = rand() % limit;
     limit = min(MAX_DELAY,
          limit*2);
     sleep(delay);
  }
}
```

```
bool locked = false;

void lock() {
  Backoff bo;
  while(true) {
     while(locked);
     if(test_and_set(locked))
       return true;
     bo.backoff();
  }
}

void unlock() {
  locked = false;
}
```

Test-And-Set Lock summary

- Space complexity O(1) for ∞ threads
  - Made 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

Queue lock

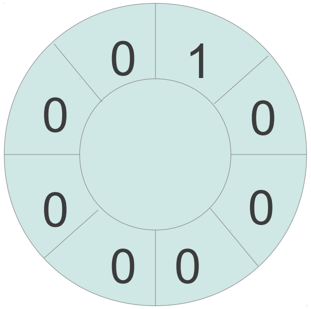
- First come – first served
- Less contention
  - Each thread spins on a local copy of a variable
  - (false sharing might occur, can be resolved with padding)
- Not space efficient!

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

void lock() {
  mySlot =
      fetch_and_add(tail) % N;

  while(!flags[mySlot]) {};
}

void unlock() {
  flags[mySlot] = false;
  flags[(mySlot + 1) % N] = true;
}
```



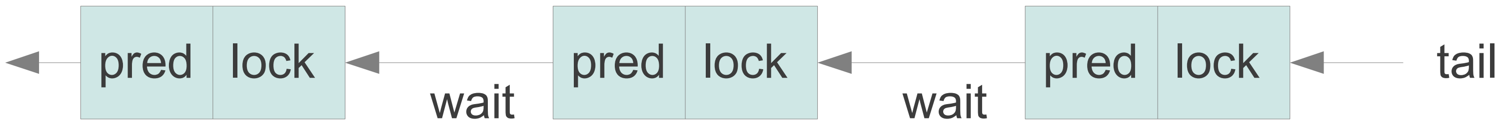
CLH lock

- Linked list
- Single Sentinel node
- Spin on locked flag of previous node

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

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

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



CLH 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

- Differs slightly from Herlihy/Shavit
  - Predecessor stored in node
  - Manual memory management
    - We may safely delete predecessor after it is unlocked

      (no other thread accessing it)

CLH lock properties

- First-come-first-served
- O(L) space, where L is the number of threads currently accessing the lock
  - Some more space depending on implementation of thread_local data
  - Herlihy/Shavit implementation requires O(p)
- Each thread spins on a separate location
  - Allocated locally by each thread, reduces false sharing
- Problem on some architectures:
  - locked field is in remote location

MCS 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* n = node;
  pred = get_and_set(&tail, n);
  if(pred != nullptr) {
     n->locked = true;
     pred->next = n;
     while(n->locked);
  }
}

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

MCS lock properties

- First-come-first-served
- O(p) space
  - Some more space depending on implementation of thread_local data
- Each thread spins on its own memory location
  - Updated by other thread

- Requires compare-and-swap

  (consensus number ∞)
- Unlock is not wait-free any more!
  - We might have to wait for the next lock owner to set the next pointer

Locks with timeouts

- 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

```
Node* tail = nullptr;
thread_local Node* node;

void lock() {
  node = new Node();
  node->locked = true;
  node->pred =
     get_and_set(&tail, node);
  while(node->pred != nullptr &&
     node->pred->locked) {

     // Remove abandoned nodes
     if(node->pred->abandoned) {
       Node* tmp =
          node->pred->pred;
       delete node->pred;
       node->pred = tmp;
     }
  }
}

void timeout() {
  node->abandoned = true;
}

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

Composite lock

```
Node nodes[n];

void lock() {
  Backoff bo;
  Node* node;
  while(!(n =
     acquireNode(rand() % n)));

  enqueueNode(n);

  wait(n);
}
```

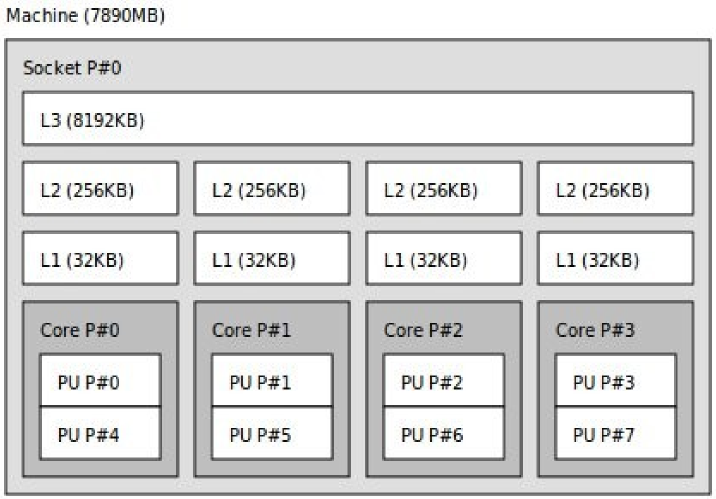
- Combines Backoff lock and Queue lock
- Preallocate fixed number of nodes < p
- Acquire a random node
  - Only one thread may use a certain node
  - On failure back off
- As soon as a node is acquired, enqueue it

- Can be augmented with a fast path for low contention
  - If queue is empty, try fast path, on failure use normal path

Hierarchical locks

- Most modern architectures can be modeled in a hierarchy
  - Some processors are “near” to each other

    → Smaller memory access times

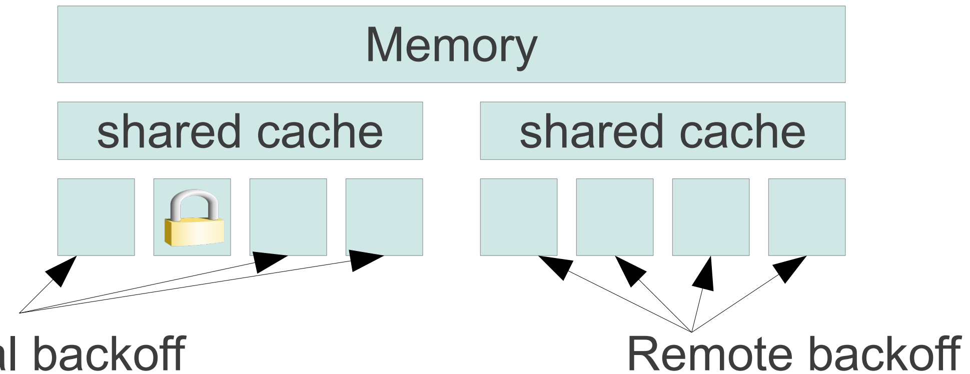


- We can improve efficiency by taking advantage of this

- Look at the architecture of your own machine
  - lstopo tool (part of hwloc)
  - Try it on saturn as well

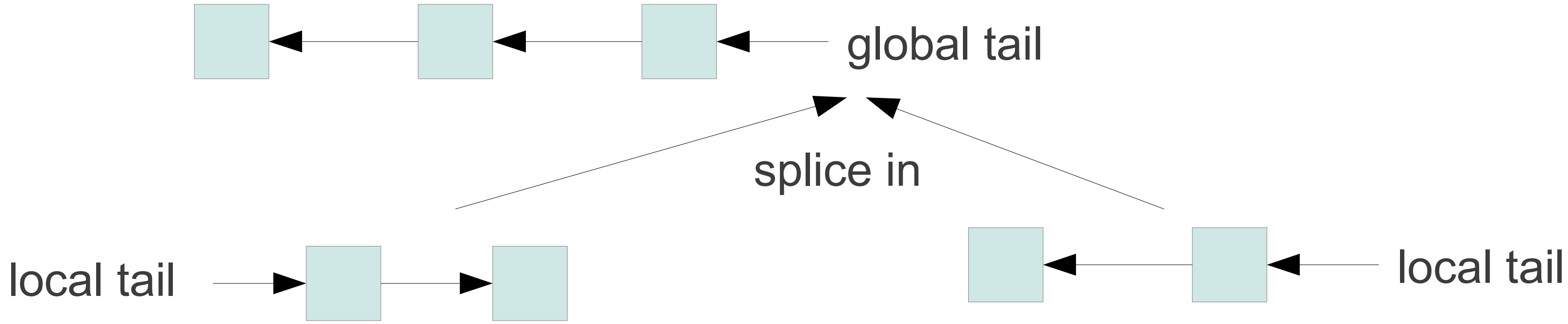
Hierarchical backoff lock

- Make distinction between local and remote accesses
- Use shorter backoff for local accesses
  - Local threads are more likely to acquire lock
- May starve remote threads
- There may be more than two levels in the hierarchy!



Hierarchical queue lock

- Build queues locally
- In addition a global queue exists
- First thread in local queue is cluster master
  - tries to splice in local queue into global queue
- First element added to local queue after splicing in is new cluster master



What we have learned so far

- To implement efficient locks, higher consensus operations are needed
  - At least consensus number 2
- Difficulties
  - Fairness
  - Congestion
  - Memory hierarchy
  - Space considerations
  - Integer overflow

What we ignored so far

- Memory consistency (coming next)
  - We assumed sequential consistency
  - Sequential consistency is not very efficient on current architectures
  - Programming with other consistency models is more difficult

Project topic

- Implement various locks
  - Test-And-Set lock
  - Test-And-Test-And-Set lock
  - Backoff lock
  - Black-White Bakery lock
    - Try different consistency models of C++11 (sequential consistency, acquire/release)
    - Also implement with volatile keyword and fences for comparison
  - CLH lock
  - Hierarchical Backoff lock
    - For n levels
    - distance function provided by framework
- Implementation as single classes in C++
  - Should conform to a certain interface (tbd – C++11 compatible)
  - We will provide test program in Pheet framework where locks can be plugged in
  - Run performance measurements with test program on saturn
- Report experiences & analyze performance
  - Short written report (what you learned, difficulties, pitfalls, performance analysis) + plots
  - Probably short presentation at end of semester (share experience with other groups)