Programmation Parallèle (PRPA)

E. Carlinet {edwin.carlinet@lrde.epita.fr}

June 2021

EPITA Research & Development Laboratory (LRDE)

Agenda

Applied synchronization: from big fat lock to lock-free data-structures

Dining Philosophers

Adding fairness to locks

From lock to lock-free programming

Stretch up: Double checked locking

Study case: Lock-based and lock-free lists

Getting lock freedom

Agenda

Agenda

1. Introduction to parallelism
2. Instruction and data-level parallelism
3. Thread level parallism
4. Parallel Design Patterns (with TBB)
5. C++ Memory model
6. Data structure for concurrent programming

Applied synchronization: from big fat lock to lock-free data-structures

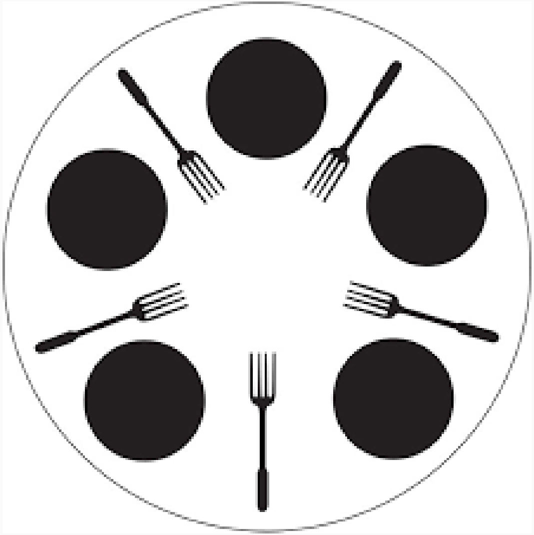
Roadmap

- Diner time
- Non-lock-free vs Lock-free vs wait free
- Case study 1: The Double-checked locking
- Case study 2: A lock-free linked list
- Case study 3: comsumer/producer

Dining Philosophers

Dining Philosophers

- A group of philosophers seat at a round table
- When they want to eat, they must take their left and right forks
- Each fork is shared between two philosophers



First strategy

A philosopher takes its left fork and then, its right fork.

A consumer waits and takes a first resource and then, waits and takes a second resource

First strategy

```cpp
std::mutex forks[N];
```

```cpp
void philo(int id) // from 0 to N-1
{
   while (1)
   {
     think();
     forks[id].lock();
     forks[(id+1) % N].lock();
     eat();    // Critical section
     forks[id].unlock();
     forks[(id+1) % N].unlock();
   }
}
```

Dining Philosophers

What we expect from the previous code:

Mutual exclusion a fork can be used by a single philosopher only

Progression A philosophers waits only if its left *or* right fork are busy

Bounded wait An hungry philosopher eventually eats some time

Dining Philosophers

What we expect from the previous code:

Mutual exclusion a fork can be used by a single philosopher only
Progression A philosophers waits only if its left *or* right fork are busy
Bounded wait An hungry philosopher eventually eats some time

In the case N = 2, there is a single critical section

- Mutual exclusion = only one thread is in the critical section
- Progression = a thread waits for the critical section only is the other is not executing it
- Bounding wait = when waiting for the critical section, a thread sees the other thread passed in the critical section a finite number of time

First strategy

A philosopher takes its left fork and then, its right fork.

```
void philo(int id) // from 0 to N-1
{
  while (1)
  {
    think();
    forks[id].lock();
    forks[(id+1) % N].lock();
    eat();    // Critical section
    forks[id].unlock();
    forks[(id+1) % N].unlock();
  }
}
```

Any problem ?

They may all die of starvation

A philosopher takes its left fork ***and then****, its right fork.*

- Each thread must acquire two shared resources
- Shared resources are acquired in two steps (**and then**)
- **Deadlock**

Deadlocks

Four conditions for deadlocks:

- Mutual exclusion (one resource in non-sharable mode)
- Hold and wait (a process holds a resource and waits for another one)
- No preemption (a resource cannot be preempted)
- Circular wait

Be aware! Deadlock may appear easily!

Deadlocks / Cause #1 : Order acquisition

```
std::lock ma, mb;
```

```
ma.lock();
mb.lock();
CS
ma.unlock();
mb.unlock();
```

```
mb.lock();
ma.lock();
CS
ma.unlock();
mb.unlock();
```

Deadlocks / Cause #1 : Order acquisition

```
std::lock ma, mb;
```

```
ma.lock();
mb.lock();
CS
ma.unlock();
mb.unlock();
```

```
mb.lock();
ma.lock();
CS
ma.unlock();
mb.unlock();
```

Solution

- Always acquire in the same order
- If multiple mutexes required, *acquire all or none* pattern with `std::lock`

Deadlocks / Cause #2 : Recursive lock

```
void foo() { m.lock(); ...; bar(); }
void bar() { m.lock(); ...; }
```

Deadlocks / Cause #2 : Recursive lock

```
void foo() { m.lock(); ...; bar(); }
void bar() { m.lock(); ...; }
```

Or much more common with client-side code:

```
class Widget
{
public:
  void setBorder() { m.lock(); ...; update(); }
  void setWidth() { m.lock(); ...; update(); }
  void onClick(void (*)(Widget*) callback) { m.lock(); callback(this); }
private:
  void update() { ... }
  std::mutex m;
}
```

Deadlocks / Cause #2 : Recursive lock

```
void foo() { m.lock(); ...; bar(); }
void bar() { m.lock(); ...; }
```

Or much more common with client-side code:

```
class Widget
{
public:
  void setBorder() { m.lock(); ...; update(); }
  void setWidth() { m.lock(); ...; update(); }
  void onClick(void (*)(Widget*) callback) { m.lock(); callback(this); }
private:
  void update() { ... }
  std::mutex m;
}
```

Solution

- Avoid calling client-side code while holding a mutex

Dining Philosophers: Second strategy

- Philosophers put back their fork, if the other one is not available
- Time before retry (can be random)

```
void philo(int id) // from 0 to N-1
{
   while (1)
   {
     think();
     std::lock(forks[id], forks[(id+1) % N]);
     eat();    // Critical section
     forks[id].unlock();
     forks[(id+1) % N].unlock();
   }
}
```

Deadlock? Problems?

Dining Philosophers: Second strategy

```
template< class Lockable1, class Lockable2, class... LockableN >
void lock( Lockable1& lock1, Lockable2& lock2, LockableN&... lockn );
```

Locks the given Lockable objects lock1, lock2, ..., lockn using a deadlock avoidance algorithm to avoid deadlock.

Dining Philosophers: Second strategy

```
template< class Lockable1, class Lockable2, class... LockableN >
void lock( Lockable1& lock1, Lockable2& lock2, LockableN&... lockn );
```

Locks the given Lockable objects lock1, lock2, . . . , lockn using a deadlock avoidance algorithm to avoid deadlock.

So. . .

- Deadlock: No!
- Starvation: Still possible! (one philosopher could never get the two forks)

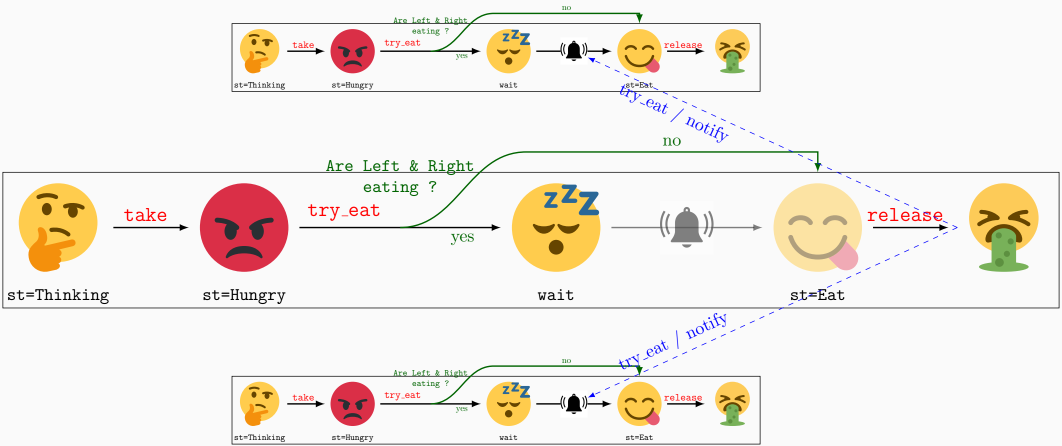
Dining Philosophers: Second strategy (RAII style)

```
void philo(int id) // from 0 to N-1
{
   while (1)
   {
     think();
     {
       std::scoped_lock(forks[id], forks[(id+1) % N]);
       eat();    // Critical section
     }
   }
}
```

Dining Philosophers (without `std::lock`) : Tannenbaum's Solution

- Local two-phase prioritization scheme (status = {THINK, HUNGRY, EAT}) before taking forks
- One global lock + One lock by philosopher

- `take`: Set status = HUNGRY and waits to be notified when status = EAT
- `try_eat`: If left/right philos are not eating, set status = EAT and notify
- `release`: Set status to THINK and free left / right neighbors.



Condition variable C++11 API

`cv.notify_one()`	Notifies one waiting thread
`cv.notify_all()`	Botifies all waiting threads
`cv.wait(l, [pred])`	Blocks until CV is woken up
`cv.wait_for(l, duration, [,pred])`	Blocks until CV is woken up or a timeout

```cpp
std::mutex glock, pl[N];
std::condition_variable cv[N];
int status[N];
```

```cpp
void take_forks(int id)
{
  glock.lock();
  status[id] = HUNGRY;
  try_eat(id);
  glock.unlock();
  std::unique_lock<mutex> l(pl[id]);
  cv[id].wait(l, [id]() {
    return status[id] == EAT;
  });
}
```

```cpp
void try_eat(int id)
{
  if (status[id]== HUNGRY &&
      status[(id-1)%N] != EAT &&
      status[(id+1)%N] != EAT)
  {
    std::lock_guard l(pl[id])
    status[id] = EAT;
    cv[id].notify_one();
  }
}
```

```cpp
void release_forks(int id)
{
  std::lock_guard g(glock);
  status[id] = THINK;
  try_eat( (id-1)%N );
  try_eat( (id+1)%N );
}
```

```cpp
void philo(int id)
{
  while (1) { think(); take_forks(); eat(); release_forks(); }
}
```

Dining Philosophers: Tannenbaum's Solution (without the magic `std::lock`)

Any problems ?

Dining Philosophers: Tannenbaum's Solution (without the magic `std::lock`)

Any problems ?

- Deadlock: No!
- Possible starvation of one philosopher

Dining Philosophers: Tannenbaum's Solution (without the magic `std::lock`)

Any problems ?

- Deadlock: No!
- Possible starvation of one philosopher

Other non-*fair* solutions:

- Only N-1 philosopher can ask for lunch (uses semaphores/condition variable)
- One (or every other) philosopher picks its right fork before its left fork
- Philosopher picks the left/right fork first at random

The idea is to break the cycle.

Dining Philosophers

About fairness:

- starvation freedom is desirable but not essential
- practical locks: many permit starvation but unlikely to happen (may happen when there is high-contention on the shared variable)

Dining Philosophers

About fairness:

- starvation freedom is desirable but not essential
- practical locks: many permit starvation but unlikely to happen (may happen when there is high-contention on the shared variable)

Some ideas to make it starvation-free:

- protocol such that every thread after using a resource can not obtain it right after releasing it
- priority queue such a threads priority increases the longer they have been waiting

Adding fairness to locks

Properties of good lock algorithm

- Mutual exclusion == *safety*
- Progression == *always 1 thread makes progress*
- Bounded wait == *no starvation*

Properties of good lock algorithm

TBB doc adds:

- Scalable: A scalable mutex is one that does not do worse than *limiting execution to one thread at a time*
- Fair: A fair mutex lets threads through in the order they arrived. Fair mutexes avoid starving threads. Each thread gets its turn.
- Recursive: A thread can call `lock()` on a mutex already *locked*
- Yied or Block = busy (active) vs passive wait.

Mutex	Scalable	Fair	Recursive	Busy wait	Size
`mutex`	✓(OS)	✓			> 2 words
`recursive_mutex`	✓(OS)	✓	✓		> 2 words
`spin_mutex`	✗	✗		✓	1 byte
`queuing_mutex`	✓	✓		✓	1 word

Spin lock implementation

```
class spin_lock
{
  void lock()
  {
    while (m_flag.test_and_set(acq_rel))
      ;
  }

  void unlock() { m_flag.clear(); }

  private:
    std::atomic_flag m_flag = ATOMIC_FLAG_INIT;
}
```

Discuss about Safety, Fairness, Recursivity…

Spin lock implementation

```
class spin_lock
{
  void lock()
  {
    while (m_flag.test_and_set(acq_rel))
      ;
  }

  void unlock() { m_flag.clear(); }

  private:
    std::atomic_flag m_flag = ATOMIC_FLAG_INIT;
}
```

Discuss about Safety, Fairness, Recursivity…

Just not *fair*, not *recursive*, may be not scalable (depends).

Adding fairness

1. The peterson’s algorithm: a two-thread solution
2. Filter lock: generalized Peterson

First try: Turn-based solution

```
std::atomic<int> turn = 0;
```

```
// ME = thread id
// OTHER = (ME + 1) % 2
void lock() {
   while (turn.load(std::memory_order_acquire) != ME)
     ;
}

void unlock() {
   turn.store(OTHER, std::memory_order_release);
}
```

Comment & Destroy!

First try: Turn-based solution

```
std::atomic<int> turn = 0;
```

```
// ME = thread id
// OTHER = (ME + 1) % 2
void lock() {
   while (turn.load(std::memory_order_acquire) != ME)
      ;
}

void unlock() {
   turn.store(OTHER, std::memory_order_release);
}
```

Comment & Destroy!

- Mutual exclusion: ✓
- Bounded wait: ✓(if they don't stop asking)
- Progress: ✗
- It supposes in-order exec

Next try: getting *progress* back

```
std::atomic<bool> tickets[2] = {false, false};
```

```
void lock() {
   tickets[ME].store(true);
   while (tickets[OTHER].load())
     ;
}

void unlock() { tickets[ME].store(false); }
```

Comment & Destroy!

Next try: getting *progress* back

```
std::atomic<bool> tickets[2] = {false, false};
```

```
void lock() {
   tickets[ME].store(true);
   while (tickets[OTHER].load())
     ;
}

void unlock() { tickets[ME].store(false); }
```

Comment & Destroy!

- Progress: ✓
- Mutual exclusion: ✓
- Bounded wait: ✗(possible deadlock)

Problem is:

I take a ticket ***and then***

If the other has a ticket, I wait till the other releases it and I enter the CS

ReNext try

```
std::atomic<bool> tickets[2] = {false, false};
```

```
void lock() {
   while (tickets[OTHER].load())
     ;
   tickets[ME].store(true);
}

void unlock() { tickets[ME].store(false); }
```

Comment & Destroy!

ReNext try

```
std::atomic<bool> tickets[2] = {false, false};
```

```
void lock() {
   while (tickets[OTHER].load())
     ;
   tickets[ME].store(true);
}

void unlock() { tickets[ME].store(false); }
```

Comment & Destroy!

- Progress: ✓
- Mutual exclusion: ✗(possible race condition)

Problem is:

If the other has a ticket, I wait till it releases the ticket ***and then***
I enter the CS and take the ticket

ReReReNext try

```
std::atomic<bool> tickets[2] = {false, false};
std::atomic<int> turn = 0;
```

```
void lock() {
  tickets[ME].store(true); // I want to pass
  turn.store(OTHER);       // But go first if you want
  while (tickets[OTHER].load() && turn.load() != ME)
    ;
}
void unlock() { tickets[ME].store(false); }
```

Comment & Destroy!

ReReReNext try

```
std::atomic<bool> tickets[2] = {false, false};
std::atomic<int> turn = 0;
```

```
void lock() {
  tickets[ME].store(true); // I want to pass
  turn.store(OTHER);       // But go first if you want
  while (tickets[OTHER].load() && turn.load() != ME)
    ;
}
void unlock() { tickets[ME].store(false); }
```

Comment & Destroy!

- The order is important *reserve, **then**, give way to the other*
- No race condition: ✓(`turn != ME` is `true` in one thread at least)
- No deadlock: ✓(`turn != ME` cannot be `true` in both threads)
- Progression: ✓(no `tickets[OTHER]` = no wait)
- Bounded wait: ✓(turn based)

BTW, do weed need SC ?

Peterson algorithm

```
std::atomic<bool> tickets[2] = {false, false};
std::atomic<int> turn = 0;
```

```
void lock() {
   tickets[ME].store(true, relaxed);   // I want to pass
   turn.store(OTHER, relaxed);         // But go first if you want
   while (! (turn.load(relaxed) == ME || tickets[OTHER].load(acquire) == false) )
     ;
}

void unlock() {
   tickets[ME].store(false, release);
}
```

Correct ?

Peterson algorithm

```
std::atomic<bool> tickets[2] = {false, false};
std::atomic<int> turn = 0;
```

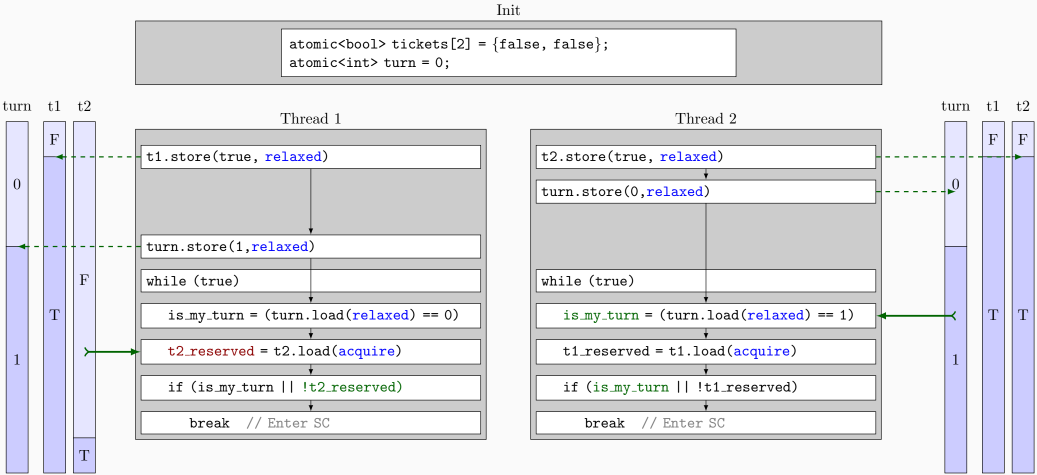
```
void lock() {
   tickets[ME].store(true, relaxed);   // I want to pass
   turn.store(OTHER, relaxed);         // But go first if you want
   while (! (turn.load(relaxed) == ME || tickets[OTHER].load(acquire) == false) )
     ;
}

void unlock() {
   tickets[ME].store(false, release);
}
```

Correct ?

- Unlock() – lock() OK with acquire/release on tickets
- But race condition possible

Peterson algorithm



We do not see the *reservation* of the thread 2.

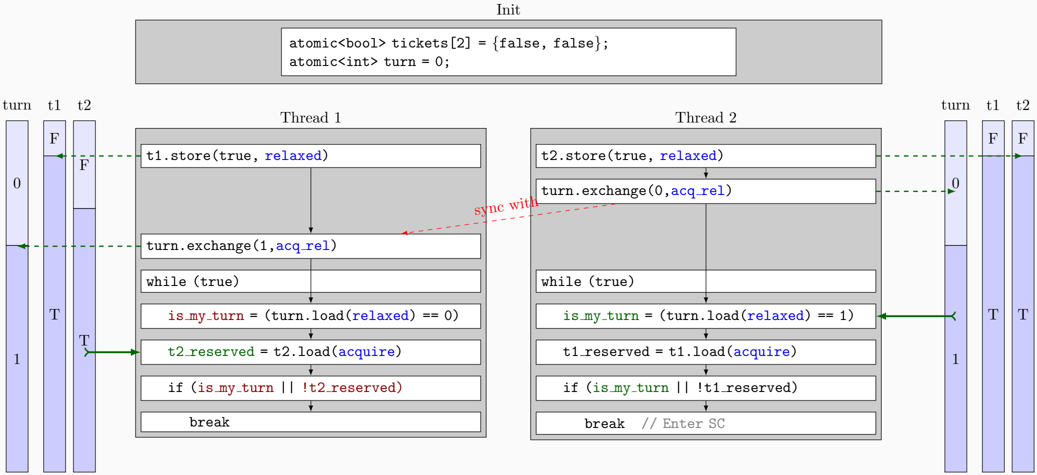
Peterson algorithm

```
std::atomic<bool> tickets[2] = {false, false};
std::atomic<int> turn = 0;
```

```
void lock() {
   tickets[ME].store(true, relaxed);  // I want to pass
   turn.exchange(OTHER, acq_rel);     // But go first if you want
   while (! (turn.load(acquire) == ME || tickets[OTHER].load(acquire)) == false))
     ;
}

void unlock() {
   tickets[ME].store(false, release);
}
```

Peterson algorithm

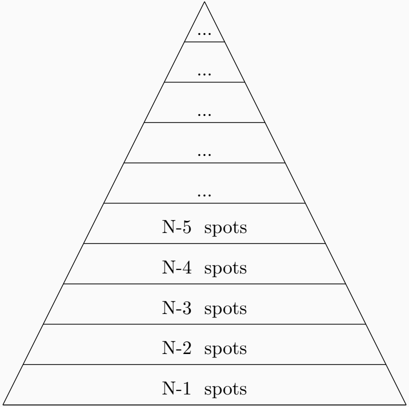


From 2 to N-way mutual exclusion

- Peterson's lock provides 2-way mutual exclusion
- Filter lock: direct generalization of Peterson's lock

Filter lock

- There are N-1 “waiting rooms”
- At each level:
  - A least one enters
  - A least one is blocked if many try
- It will remain only **the one**



Filter lock

```
std::atomic<int> priority[N] = {-1, -1, -1, ...};
std::atomic<int> victim[N] = {-1, -1, -1, ...};
```

```
void lock(){
  for (int j = 0; j < N-1; j++)
  {
    priority[ME] = j; // Take ticket in queue
    victim[j] = ME;
    while (victim[j] == ME &&
           !ImTheONE_TheOnlyONE())
      ;
  }
}
void unlock() { priority[ME] = -1; }
```

```
bool ImTheONE_TheOnlyONE()
{
  int l = priority[ME];
  for (int k = 0; k < N; ++k)
    if (k != ME && priority[k] >= l)
      return false;
  return true;
}
```

Take home message

- Peterson algorithm is a classical lock algorithm with atomic *loads* and *stores* only
- Not used in practice (locks based on stronger atomic primitives are more efficient)

- Mutexes are not free, they may use expensive algorithms to enable some features (fairness, scalability…)
- You must use the right lock for your need

From lock to lock-free programming

Why lock-free code

Single Thread[1]

Lock

Lock-free

[1]Images from Herb Sutter - Lock free programming

Concurrency and scalability

Eliminate/reduce blocking/waiting in algorithm and data structures

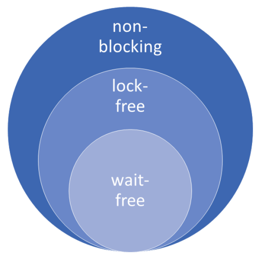
Three levels of lock-freedom

Blocking

Unable to progress in its execution until some other thread releases a resource.
Example: Mutex / A simple CAS in a loop for a two state variable

```
while (!var.test_and_set()) { std::this_thread::yield(); }
```

Non-blocking



- Obstruction-free = *progress if no interferance*
  If a thread is executed in isolation (all the others suspended), it will complete.
- Lock-free = *someone makes progress*
  Every step taken achieves global progress (starvation rare in practice)
- Wait-free = "no one ever waits*
  Every one will complete in #steps whatever what else is going on

Three levels of lock-freedom

```
std::atomic<int> turn = 0;
```

In people's mind: *lock-free = no mutex* (but not necessary)

Compare:

```
while (turn.exchange(1) == 1) {};
```

And:

```
int val = turn.load();
while (!turn.compare_exchange_weak(val, val+1)) {};
```

Remark?

Three levels of lock-freedom

```
std::atomic<int> turn = 0;
```

In people's mind: *lock-free = no mutex* (but not necessary)

Compare:

```
while (turn.exchange(1) == 1) {};
```

And:

```
int val = turn.load();
while (!turn.compare_exchange_weak(val, val+1)) {};
```

Remark?

- First is blocking (waits the thread #0 to finish)
- Second is lock-free (increment the turn counter)

Lock free fundamental #1: transactional model

Think transactional (ACID):

- Atomicity: *all or nothing* (no intermediate state)
- Consistency: one consistent state to another
- Isolation: two transactions never operate simultaneously on the same data
- Durability: once committed, a transaction is not overwritten by a second one that ignores the first one (lost update)

For lock-free:

- Publish each change using one atomic write
- Make sure concurrent updates do not interfere with each other or concurrent readers

When accessing concurrently a shared resource, ask yourself about:

- 1 reader + 1 writer
- 2 writers

Lock free fundamental #2: the atomic weapons

Your key tool is the atomic variable

Semantics and operations:

- read/write are atomic, no locking required
- read/write are guarenteed not to be reordered
- `T exchange(T new)` for a *load* and *store*
- compare-and-swap loop (CAS-loop)

```
bool compare_exhange_weak(T& expected, T desired) {
if (value == expected) { value = desired; return true}
else { expected = value; return false; }
}
```

Stretch up: Double checked locking

Lazy-initialization problem

- You need to initialize some auxiliary data for computing `void foo(args...)`
- `foo` can be called by many threads
- You don't want to initialize the aux data too early (program startup) (if `foo` is not called for example)

```
data_t CreateAuxData();
```

```
void foo()
{
    data_t x = CreateAuxData();
    // Use x
}
```

```
void foo()
{
    data_t x = CreateAuxData();
    // Use x
}
```

Problem:

Lazy-initialization problem

- You need to initialize some auxiliary data for computing `void foo(args...)`
- `foo` can be called by many threads
- You don't want to initialize the aux data too early (program startup) (if `foo` is not called for example)

```
data_t CreateAuxData();
```

```
void foo()
{
    data_t x = CreateAuxData();
    // Use x
}
```

```
void foo()
{
    data_t x = CreateAuxData();
    // Use x
}
```

Problem:

created and initialized twice.

Lazy-initialization problem

```
void foo()
{
  static std::mutex m;
  static std::unique_ptr<data_t>* x = nullptr;

  {
    std::lock_guard l(m);
    if (x == nullptr)
      x = std::make_unique<data_t>(CreateAuxData());
  }
}
```

Problem?

Lazy-initialization problem

```
void foo()
{
  static std::mutex m;
  static std::unique_ptr<data_t>* x = nullptr;

  {
    std::lock_guard l(m);
    if (x == nullptr)
      x = std::make_unique<data_t>(CreateAuxData());
  }
}
```

Problem?

- always block to test initialization (even when the data is initialized)

Lazy-initialization problem

```
void foo()
{
  static std::mutex m;
  static std::unique_ptr<data_t>* x = nullptr;

  if (x == nullptr)
  {
    std::lock_guard l(m);
    x = std::make_unique<data_t>(CreateAuxData());
  }
}
```

OK?

Lazy-initialization problem

```
void foo()
{
  static std::mutex m;
  static std::unique_ptr<data_t>* x = nullptr;

  if (x == nullptr)
  {
    std::lock_guard l(m);
    x = std::make_unique<data_t>(CreateAuxData());
  }
}
```

OK?

- No: there is a data race (concurrent read/write of x)

Lazy-initialization problem

```
void foo()
{
  static std::mutex m;
  static std::atomic<data_t*> x = nullptr;

  if (x.load() == nullptr)
  {
    std::lock_guard l(m);
    x.store(new data_t(CreateAuxData()));
  }
}
```

OK?

Lazy-initialization problem

```
void foo()
{
  static std::mutex m;
  static std::atomic<data_t*> x = nullptr;

  if (x.load() == nullptr)
  {
    std::lock_guard l(m);
    x.store(new data_t(CreateAuxData()));
  }
}
```

OK?

- No data race, but two threads can see nullptr and initialize twice

Lazy-initialization problem

```
void foo()
{
  static std::mutex m;
  static std::atomic<data_t*> x = nullptr;

  if (x.load() == nullptr)
  {
    std::lock_guard l(m);
    if (x.load() == nullptr)
      x.store(new data_t(CreateAuxData()));
  }
}
```

OK?

Lazy-initialization problem

```
void foo()
{
  static std::mutex m;
  static std::atomic<data_t*> x = nullptr;

  if (x.load() == nullptr)
  {
    std::lock_guard l(m);
    if (x.load() == nullptr)
      x.store(new data_t(CreateAuxData()));
  }
}
```

OK?

✓

- Because of the mutual exclusion, `x.load() == nullptr` is true once
- Do we need SC ?

Lazy-initialization problem

```
void foo()
{
  static std::mutex m;
  static std::atomic<data_t*> x = nullptr;

  if (x.load(acquire) == nullptr)
  {
    std::lock_guard l(m);
    if (x.load(relaxed) == nullptr)
      x.store(new data_t(CreateAuxData()), release);
  }
}
```

- When the first `x.load()` is non-null, we need to ensure that memory writes in x are all visible => acquire-release
- The second `x.load()` can be relaxed because already synchronized by the acquire/release semantic of the mutex

BTW: there are tools in the Standard Libtary

std::call_once

```
void foo()
{
  static std::unique_ptr<data_t> x = nullptr;
  static std::once_flag x_flag;

  std::call_once(x_flag, [&]() { x = std::make_unique<data_t>(CreateAuxData()); })

}
```

And BTW, C++ rocks

```
void foo()
{
  static data_t x = CreateAuxData(); // Thread safe
}
```

So what DCLP solve?

- We have an exceptional situation that happens rarely
- Handling the exception is not thread-safe (mutex)
- The test for exception must be atomic (may be under the same mutex)
- There is a **fast** non-locking test
- There is few chances that the exception reoccurs again

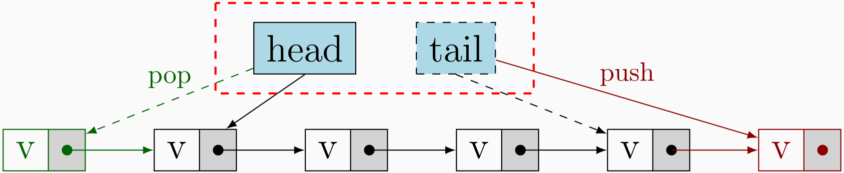
Study case: Lock-based and lock-free lists

Linked lists

Used to implement: * Stacks (one entry linked-list) * Queues (double entry linked-list) (Producer-Consumer problems) * Sets (Sorted linked-items)

Single threaded queue

- Only three operations:`find`, `push`, and `pop`
- Challenge: make it concurrent



```cpp
struct Node { T value; Node* next; };

class queue
{
  Node* m_head = nullptr;
  Node* m_tail = nullptr;

  queue() = default;
  ~queue() {
    while (m_head) { Node* tmp = m_head; m_head = m_head->next; delete tmp; }
  }
  T pop() {
    T v = std::move(m_head->value);
    Node* tmp = m_head; m_head = m_head->next; delete tmp;
    return v;
  }
  void push(T val) {
    Node* tail = new Node{std::move(val), nullptr};
    if (m_tail) { m_tail->next = tail; }
    else { m_head = tail; }
    m_tail = tail;
  }
  bool find(T val) { // trivial }
};
```


We will suppose that T's move constructor/assignement is no-throw

First approach: a big fat lock

One lock to rule them all

Which methods need a special care:

First approach: a big fat lock

One lock to rule them all

Which methods need a special care:

Method	Special care
Constructor	
Destructor	
pop()	✓
push()	✓
find()	✓

First approach: a big fat lock

Lock the whole structure: * Everything gets serialized * Do not scale well (poor with contention)

```
class queue
{
  std::mutex m;
  Node* m_head = nullptr;
  Node* m_tail = nullptr;


  T pop() { std::lock_guard l(m); ... }
  void push(T val) { std::lock_guard l(m); ... }
  bool find(T val) { std::lock_guard l(m); ... }
};
```

Second approach: RW locks

What differs between `pop` / `push` and `find`?

Second approach: RW locks

What differs between `pop` / `push` and `find`?

- `Find` is a R-only operation
- `pop` / `push` are RMW operations

We can allow concurrent R-only operations as long as there is no RMW operations

You can have multiple RW policies w.r.t. to the problem:

- *Read preferring*: writer does not acquire lock while there is one reader in the queue (possible writer starvation)
- *Write preferring*: new readers do not acquire lock while there is a writer queued

Possible read-preferring implementation

- One mutex and one condition variable
- One counter r: number of readers

```
std::mutex g;
std::condition_variable cv;
int r = 0;
```

For reader

```
{
 // Block if active writer
 std::lock_guard l(g);
 r++;
}
// Reader stuff
{
 std::lock_guard l(g)
 r--;
}
cv.notify_one();
```

For writer

```
std::unique_lock l(g);
cv.wait(l, []() { r == 0; });
```

Second approach: RW locks

C++17 has name for this: `shared_mutex`

- Exclusive locking: `lock`, `try_lock`, `unlock`
- Shared locking: `lock_shared`, `try_lock_shared`, `unlock_shared`

```
class queue
{
  std::shared_mutex m;
  Node* m_head = nullptr;
  Node* m_tail = nullptr;


  T pop() { std::lock_guard<std::shared_mutex> l(m); ... }
  void push(T val) { std::lock_guard<std::shared_mutex> l(m); ... }
  T* find(T val) { std::shared_lock<std::shared_mutex> l(m); ... }
};
```

Third approach: fine grained locking

Do we need to lock the whole stuff ?

- Per element locking
- Multiple threads can operate concurrently
- Serialized progression

If we have just `pop` and `push`, what's need to be guarded:

Third approach: fine grained locking

Do we need to lock the whole stuff ?

- Per element locking
- Multiple threads can operate concurrently
- Serialized progression

If we have just `pop` and `push`, what's need to be guarded:

- `m_head` / `m_tail`

If we have `insert` and `delete` in any position, what's need to be guarded:

Third approach: fine grained locking

Do we need to lock the whole stuff ?

- Per element locking
- Multiple threads can operate concurrently
- Serialized progression

If we have just `pop` and `push`, what's need to be guarded:

- `m_head` / `m_tail`

If we have `insert` and `delete` in any position, what's need to be guarded:

- every single element of the list

Third approach: fine grained locking

If we just have `push` **&** `pop`

Problem:

- `push` may modify both `m_head` and `m_tail`
- `pop` may modify both `m_head` and `m_tail`
- They access the `next` pointer of a node

Solution:

Third approach: fine grained locking

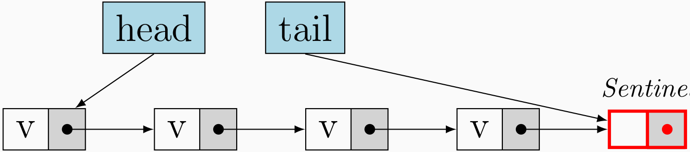
If we just have `push` & `pop`

Problem:

- `push` may modify both `m_head` and `m_tail`
- `pop` may modify both `m_head` and `m_tail`
- They access the `next` pointer of a node

Solution:

- Seperate data to enable concurrency: a sentinel node so that `m_head != m_tail`



- The `empty` condition is `m_head == m_tail`
- `pop` as previously
- `push` = write dummy tail node and add a new dummy one

```
class queue
{
  std::mutex hm, tm;
  Node* m_head = nullptr, m_tail = nullptr;

  queue() : m_head(new node), m_tail(m_head) {}

  T pop() {
    std::lock_guard l(hm);
    auto b = std::move(m_head->val());
    auto tmp = m_head; m_head = m_head->next; delete m_head;
    return v;
   }

  void push(T val) {
    std::lock_guard l(tm);
    m_tail->value = std::move(val);
    m_tail->next = new node();
    m_tail = m_tail->next;
  }
}
```

Third approach: fine grained locking

If we add `find`

Third approach: fine grained locking

If we add `find`

- Find need to lock both `tail` and `head`
- May be combine with RW mutexes for better concurrency

If we add `insert()` and `delete` in any position

Third approach: fine grained locking

If we add `find`

- Find need to lock both `tail` and `head`
- May be combine with RW mutexes for better concurrency

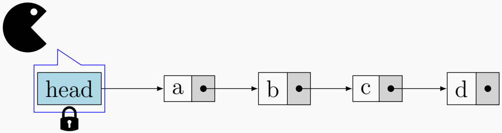
If we add `insert()` and `delete` in any position

- One lock by element or block of element
- Methods that work on disjoint pieces need to exclude each other

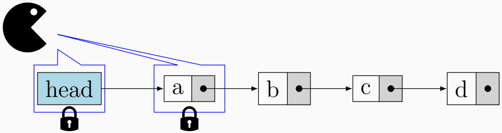
Hand-over-hand / chain locking

- You can't treat each element separately
- You must *not* unlock the current element before locking the next
- Chain locking guaranties progression and safety

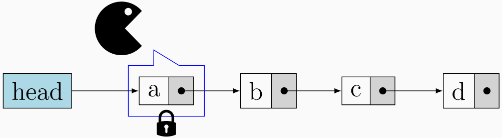
Step 1



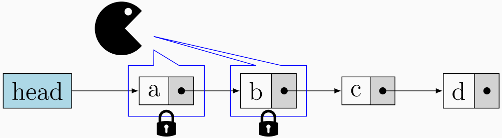
Step 2



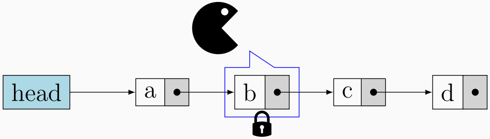
Step 3



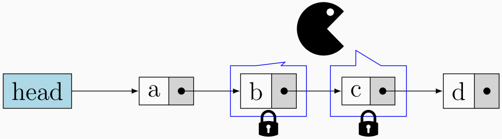
Step 4



Step 5



Step 6



Hand-over-hand / chain locking

- You must *not* unlock the current element before locking the next

```
class forward_lock_guard
{
  std::mutex* m;

  forward_lock_guard(std::mutex& mu) : m(&mu) { m->lock(); }
  ~forward_lock_guard() { m->unlock(); }

  void reset(std::mutex& next)
  {
    next.lock();
    m->unlock();
    m = &next;
  }
};
```

Hand-over-hand - traversal

```
struct Node {
  T value;
  Node* next;
  std::mutex lock;
};

class linked_list
{
  Node* m_head; std::mutex g;

  bool find(T val);
  void delete(T val);
};
```

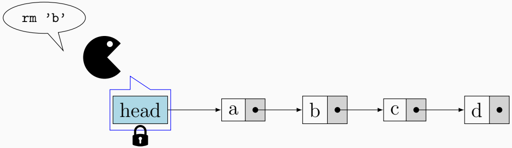
```
bool find(T val)
{
  forward_lock_guard l(g);
  Node* current = m_head;
  while (current != null)
  {
    if (current->value == val)
      return true;
    current = current->next;
    if (current) l.reset(current->lock);
  }
  return false;
}
```

Hand-over-hand - insertion/deletion

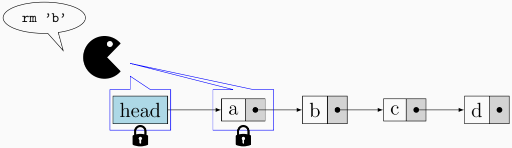
Deletion:

- Find (traverse) node
- lock `current` and `prec`,
- update `prec->next`
- Unlock

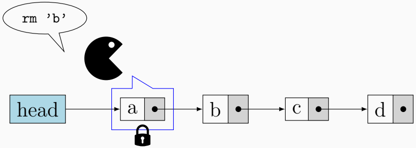
Step 1



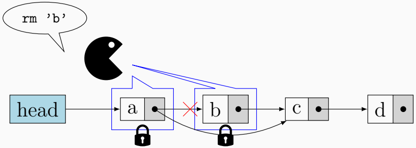
Step 2



Step 3



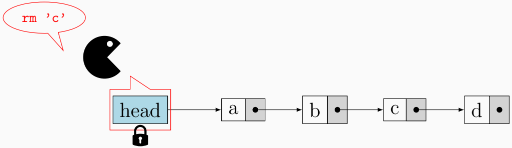
Step 4



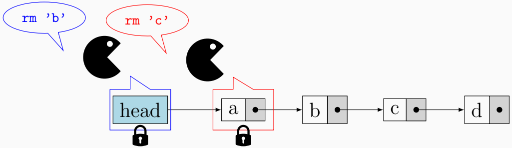
Why do we need to lock the victim ?

Fine grained locking - Hand-over-hand - insertion/deletion

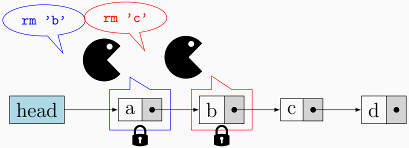
Step 1 (T1 is in the place)



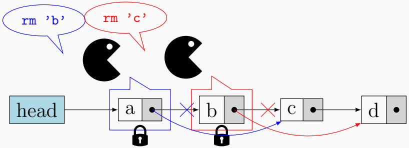
Step 2 (T2 enters the game)



Step 3 (Go go go)



Step 4 (found 'b', found 'c' => update)



WTF ?? (**Lost update** !)

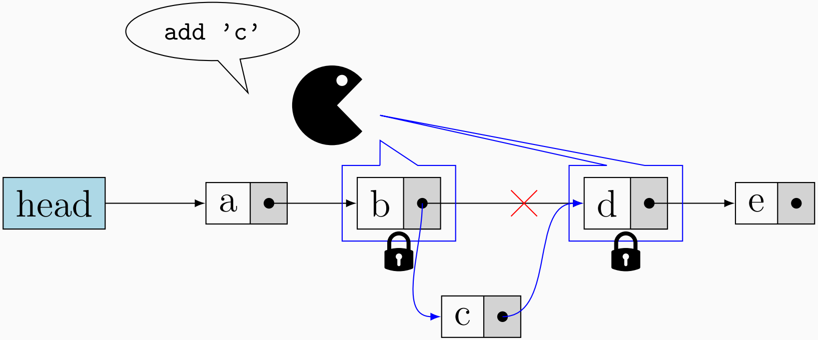
Fine grained locking - Hand-over-hand - insertion/deletion

Deletion:

- Find (traverse) node
- lock `current` and `prec`,
- update `prec->next`
- Unlock

Insertion:

- Find (traverse) node
- lock `succ` and `prec`,
- update `prec->next`
- Unlock



Why do we lock `prec` and `succ` even in the insertion?

- Acutally, locking `prec` is enough (for insert)
- Because *delete* needs 2 locks, if you lock an entry:
  - It cannot be removed
  - Neither its successor

Third approach: fine grained locking

Hand-over-hand / Discussion

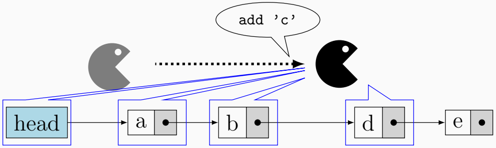
- More concurrency: an operation working at the end of the list does not obstruct those at the beginning
- But operations on "low" nodes may obstruct those on high nodes
- Long chain of acquire/release -> *Optimistic locking*

Optimisitic locking

- No locks on the traverse path
- Try with **no** synchronization
  - if you **win**, you win
  - if you **loose**, **retry** with synchronization
- Less locking and operation can pass working area
- Require a validation step (**win** or **loose**) (expensive?)
- Retry is cheaper than waiting lock

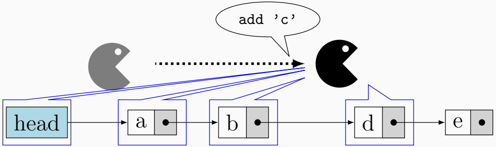
Optimisitic locking (pb 1)

Go go go…

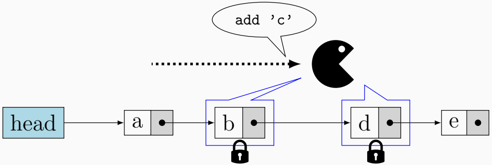


Optimisitic locking (pb 1)

Go go go…

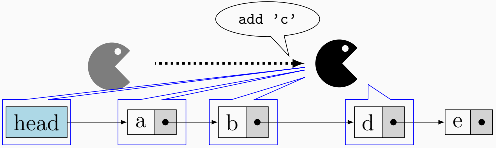


Lock & update (ok… but…)

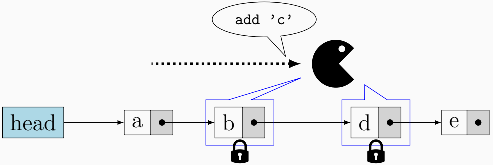


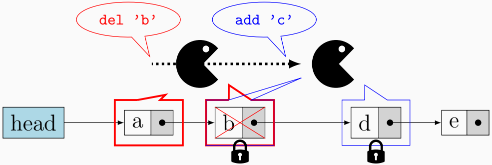
Optimisitic locking (pb 1)

Go go go…



Lock & update (ok… but…)



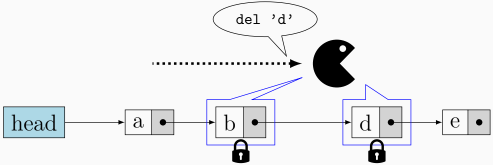


This happened… T2 passed before T1 locked the nodes…

We need to check that *b* is accessible from the head

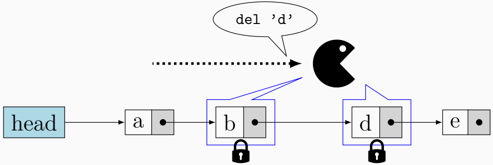
Optimisitic locking (pb 2)

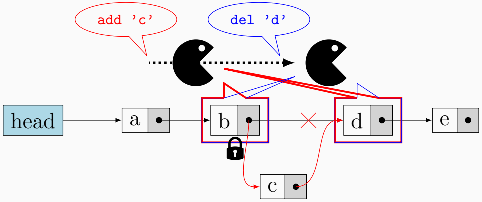
Go, find and lock... but...

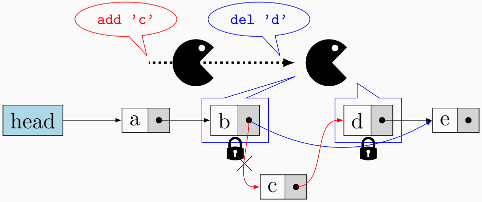


Optimisitic locking (pb 2)

Go, find and lock... but...







This happened. . .

- T2 passed before T1 locks nodes and insert before the *victim node*
- The insertion of 'c' is lost (overwritten the deletion update)

We need to check that `b.next` has not changed

Optimistic locking

Deletion:

- find entries
- lock `current` and `prec`
- check validity
- update `prec`
- release `lock`

Insertion:

- find entries
- lock `current` and `prec`
- check validity
- update `current`
- release `lock`

Validation = while holding lock:

- Check **accessibility** of the node
- Check that the `next` pointer has not changed

Optimisitic locking

Problem:

- What about concurrent traversing / deletion ?
  We need a smart GC for reclamation.
- Validation needs to traverse list twice (to detect deleted items)
- `contains` still requires locks

Solution:

Optimisitic locking

Problem:

- What about concurrent traversing / deletion ?
  We need a smart GC for reclamation.
- Validation needs to traverse list twice (to detect deleted items)
- `contains` still requires locks

Solution:

lazy approach:

- Do not delete the node: *mark* it as deleted
- `contains` is now wait-free
- **accessibility** is constant time

Still need memory reclaim to free deleted nodes some day

Getting lock freedom

Lock-free list

- Simplify first (stack instead of queue, no need to maintain two pointers)
- No mutex
- Raw pointers replaced by atomics
- We forget pop for a while

```
struct Node { T value; Node* next; };

class stack
{
  std::atomic<Node*> m_head = nullptr;

  stack();
  ~stack();

  T pop() {FIXME}
  void push(T val) {FIXME}
  bool find(T val) {FIXME};
};
```

Find

- Concurrency issues: none (none with find operations… and should be safe to run concurrently with insert operations)

```
bool find(T val)
{
  auto p = m_head.load();
  while (p)
  {
    if (p->value == val) return true;
    p = p->next.load();
  }
  return false;
}
```

Push

- Create a new node
- Set next pointer to the current head
- Publish as the new head

```
void push(T val)
{
    auto p = new Node;
    p->value = val;
    p->next = m_head.load();
    m_head.store(p);
}
```

Is it ok ?

Push

- Create a new node
- Set next pointer to the current head
- Publish as the new head

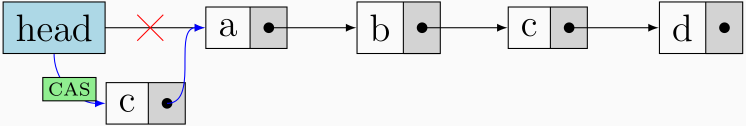
```
void push(T val)
{
   auto p = new Node;
   p->value = val;
   p->next = m_head.load();
   m_head.store(p);
}
```

Is it ok ?

Concurrency issues:

- None for readers: the insertion is atomic
- Problem for writers: if two threads inserts in the same time (lost update problem)

Push - solution

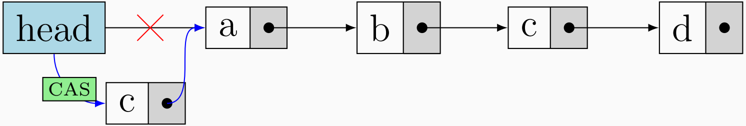



```
void push(T val)
{
    auto p = new Node;
    p->value = val;
    p->next = m_head.load();
    while (!m_head.compare_exchange_weak(p->next, p))
     ;
}
```

Semantics is *loop until the head hasn't changed and we are the head*

Issues?

Push - solution




```
void push(T val)
{
    auto p = new Node;
    p->value = val;
    p->next = m_head.load();
    while (!m_head.compare_exchange_weak(p->next, p))
      ;
}
```

Semantics is *loop until the head hasn't changed and we are the head*

Issues?

- OK for readers
- OK for writers
- That was that easy?

Now POP

- pop comes into the game

```
T pop()
{
  auto p = m_head.load();
  m_head.store(p->next.load());
  T val = std::move(m_head->value);
  delete p;
  return val;
}
```

Problems?

Now POP

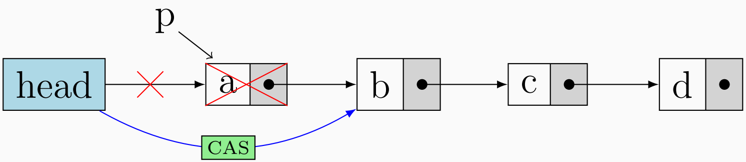
- pop comes into the game

```
T pop()
{
  auto p = m_head.load();
  m_head.store(p->next.load());
  T val = std::move(m_head->value);
  delete p;
  return val;
}
```

Problems?

- For readers: problem with simultaneous traversal + pop
- For writers: problem with two pop or pop + push

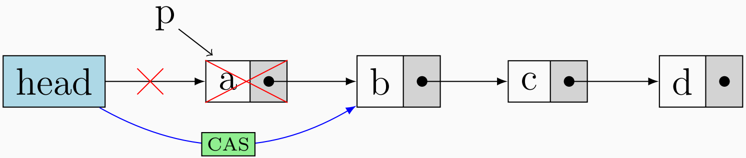
Now POP




```
void pop()
{
  auto p = m_head.load();
  while (p && !m_head.compare_exchange_weak(p, p->next))
    ;
  T val = std::move(p->value);
  delete p;
}
```

Problems?

Now POP




```
void pop()
{
  auto p = m_head.load();
  while (p && !m_head.compare_exchange_weak(p, p->next))
    ;
  T val = std::move(p->value);
  delete p;
}
```

Problems?

- Same problems for readers: `find` is pointing to the first node (*p*) and then read `next`
- Sublte problem for writers: ABA problem. Two nodes with the same address, but different identities (existing at different times).

The ABA problem

- Step 1 of delete:

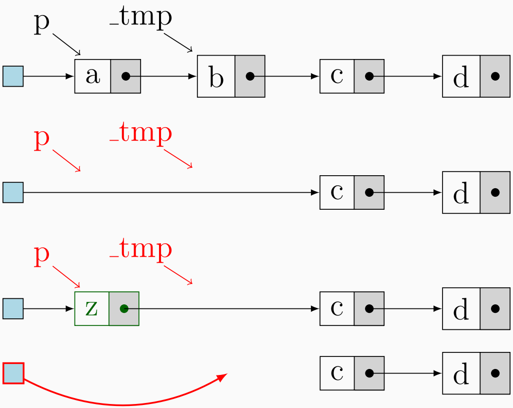
```
p = head; _tmp = p->next;
```

Another thread deletes 2 nodes

Another insert a new node (in the same memory location)

- Step 2: CAS succeeds

```
head.compare_exchange_weak(p, _tmp)
delete p;
```



ABA solutions

- Lazy garbage collection (with reclamation list for example)
- Ref counting

- Lock-free is hard for deletion
- But easy for read/insertions

What I would have liked to talk about

- Read Copy Update (RCU)

CppCon 2017 Read, Copy, Update, then what? RCU for non kernel programmers

- Hazard Pointers

The Landscape and Exciting New Future of Safe Reclamation for High Performance

Sources

- CppCon 2014: Herb Sutter "Lock-Free Programming (or, Juggling Razor Blades)
- Concurrency with Modern C++
- The Art of Multiprocessor Programming