Lecture 5 – Concurrent Data Structures (in modern C++)

CS3211 Parallel and Concurrent Programming

Outline

- Design concurrent (thread-safe) data structures
  - Fine-grained vs. coarse-grained synchronization
  - Lock-based vs. lock-free
- Examples:
  - Stack
  - Queue

Design data structures for concurrency

- Goal
  - Multiple threads can access the data structure concurrently
    - Performing the same or distinct operations
  - Each thread sees a self-consistent view of the data structure
- Designing *thread-safe* data structures
  - No data is lost or corrupted
  - All invariants are upheld
  - No problematic race condition

Protect the data structure with a mutex

- Prevents *true* concurrent access to the data it protects
- **Serialization**: threads take turns accessing the data protected by the mutex
- We want concurrent data structures that enable true concurrency
  - Some data structures have more scope for true concurrency than others

Invariants

- Invariants – statements that are always true about a particular data structure
  - Often broken during an update on the data structure
  - Example: this variable `no_of_items` contains the number of items in the list
- Use invariants to reason about program correctness

Delete a node from a doubly linked list

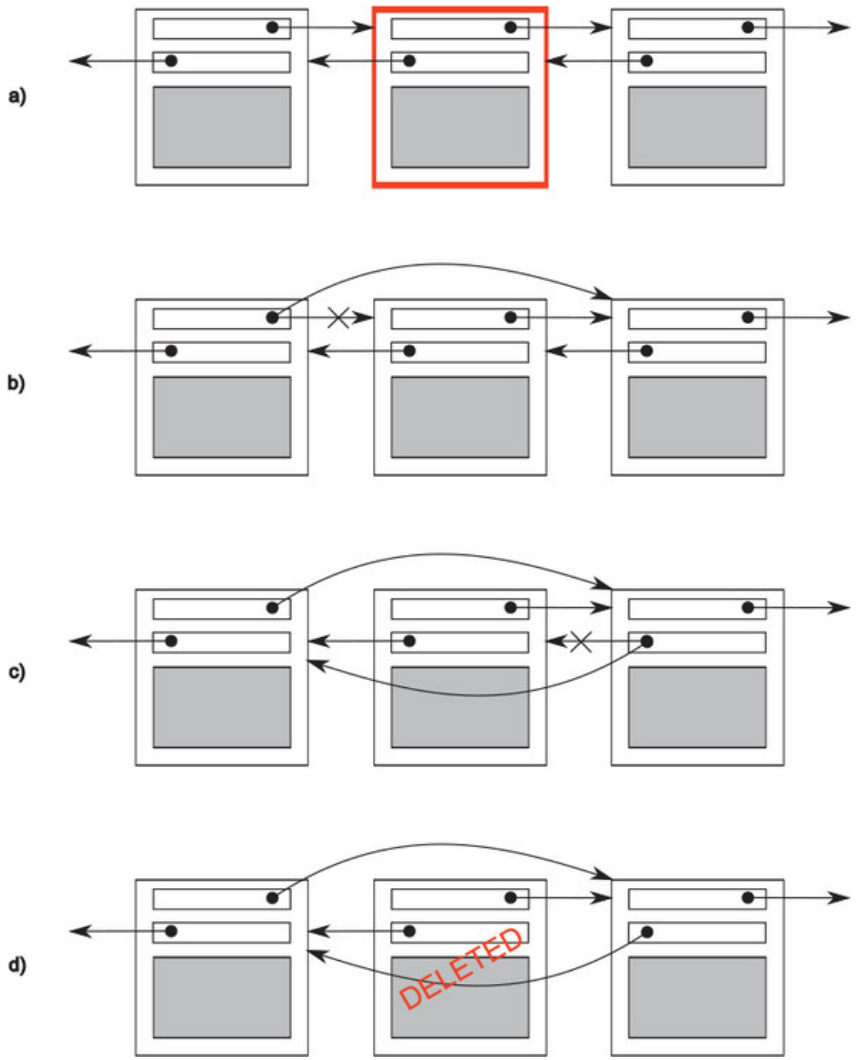
*Invariant is broken during the delete

a) Identify the node to delete: N.

b) Update the link from the node prior to N to point to the node after N.

c) Update the link from the node after N to point to the node prior to N.

d) Delete node N.



Building a thread-safe data structure

- Ensure that no thread can see a state where the invariants of the data structure have been broken by the actions of another thread
- Take care to avoid race conditions inherent in the interface to the data structure by providing functions for complete operations rather than for operation steps
- Pay attention to how the data structure behaves in the presence of exceptions to ensure that the invariants are not broken
- Minimize the opportunities for deadlock when using the data structure by restricting the scope of locks and avoiding nested locks where possible

Concurrency while calling functions

- Constructors and destructors require exclusive access to the data structure
  - It's up to the users to ensure that data structures not accessed before construction is complete or after destruction has started
- Data structure that support assignment, swap(), or copy construction:
  - Decide whether these operations are safe to call concurrently with other operations or whether they require the user to ensure exclusive access even though the majority of functions for manipulating the data structure may be called from multiple threads concurrently without any problems

Truly designing for concurrency

- The smaller the protected region, the fewer operations are serialized, and the greater the potential for concurrency
- Provide the *opportunity for concurrency* to threads accessing a thread-safe data structure
  - If one thread is accessing the data structure through a particular function, which functions are safe to call from other threads?

Thread-safe data structures

- Minimize the amount of serialization that must occur
- Lock at an appropriate granularity
  - A lock should be held for only the minimum possible time needed to perform the required operations
- Alternatives
  - Multiple threads might perform one type of operation on the data structure concurrently, whereas another operation requires exclusive access by a single thread
    - Use `std::shared_mutex` to allow concurrent access from threads that read the data structure, but exclusive access for threads that modify the data structure
  - Safe for multiple threads to access a data structure concurrently if they're performing *different* actions, whereas multiple threads performing the *same* action would be problematic

Enable genuine concurrent access

- Can the scope of locks be restricted to allow some parts of an operation to be performed outside the lock?
- Can different parts of the data structure be protected with different mutexes?
- Do all operations require the same level of protection?
  - How about operations on *const* objects?
  - *mutable* keyword
- Can a simple change to the data structure improve the opportunities for concurrency without affecting the operational semantics?

Lock-based concurrent data structures

- Ensure that the right mutex is locked when accessing the data and that the lock is held for the minimum amount of time
  - Data can’t be accessed outside the protection of the mutex lock
  - There are no race conditions inherent in the interface
- Using multiple mutexes to protect separate parts of the structure
  - Deadlocks are possible

S1: A thread-safe stack with one global mutex

- Safe for multiple threads to call the member functions concurrently

```
#include <exception>
struct empty_stack: std::exception
{
    const char* what() const throw();
};
template<typename T>
class threadsafe_stack
{
private:
    std::stack<T> data;
    mutable std::mutex m;
public:
    threadsafe_stack(){}
    threadsafe_stack(const threadsafe_stack& other)
    {
        std::lock_guard<std::mutex> lock(other.m);
        data=other.data;
    }
    threadsafe_stack& operator=(const threadsafe_stack&) = delete;
    void push(T new_value)
    {
        std::lock_guard<std::mutex> lock(m);
        data.push(std::move(new_value));
    }
    std::shared_ptr<T> pop()
    {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw empty_stack();
        std::shared_ptr<T> const res(
           std::make_shared<T>(std::move(data.top())));
        data.pop();
        return res;
    }
    void pop(T& value)
    {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw empty_stack();
        value=std::move(data.top());
        data.pop();
    }
    bool empty() const
    {
        std::lock_guard<std::mutex> lock(m);
        return data.empty();
    }
};
```

S1: Concurrency in the thread-safe stack

- Safe for multiple threads to call the member functions concurrently
- BUT the work is *serialized* for the stack data structure
  - only one thread is ever doing any work in the stack data structure at a time
- Exception safe
- Serialization limits the performance of an application that exhibits significant contention on the stack
- No means of waiting for an item to be added
  - A thread must periodically call `empty()`, or call `pop()` and catch the `empty_stack` exceptions
  - Consume precious resources checking for data or the user must write an external wait and notification code

Q1: A thread-safe queue with notification (one mutex)

```
template<typename T>
class threadsafe_queue
{
private:
    mutable std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue()
    {}
    void push(T new_value)
    {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(std::move(new_value));
        data_cond.notify_one();
    }
    void wait_and_pop(T& value)
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        value=std::move(data_queue.front());
        data_queue.pop();
    }
    std::shared_ptr<T> wait_and_pop()
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        std::shared_ptr<T> res(
            std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }
    bool try_pop(T& value)
    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return false;
        value=std::move(data_queue.front());
        data_queue.pop();
        return true;
    }
    std::shared_ptr<T> try_pop()
    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return std::shared_ptr<T>();
        std::shared_ptr<T> res(
            std::make_shared<T>(std::move(data_queue.front())));
        data_queue.pop();
        return res;
    }
    bool empty() const
    {
        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }
};
```

Q1: A thread-safe queue with notification (one mutex)

- The analysis for the stack applies here as well
- The `wait_and_pop()` functions are a solution to the problem of waiting for a queue entry
- Exception safety
  - If more than one thread is waiting when an entry is pushed onto the queue (`data_cond.wait`), only one thread will be woken by the call to `data_cond.notify_one()`
  - But if that thread then throws an exception in `wait_and_pop()` (when the new `std::shared_ptr<>` is constructed), none of the other threads will be woken
  - Solutions:
    - Replaced with `data_cond.notify_all()`, which will wake all the threads but at the cost of most of them then going back to sleep when they find that the queue is empty after all, or
    - `wait_and_pop()` call `notify_one()` if an exception is thrown, or
    - Move the `std::shared_ptr<>` initialization to the `push()` call and store `std::shared_ptr<>` instances rather than direct data values (see next slide)

Q2: Use shared-pointers

```
template<typename T>
class threadsafe_queue
{
private:
    mutable std::mutex mut;
    std::queue<std::shared_ptr<T> > data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue()
    {}
    void wait_and_pop(T& value)
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        value=std::move(*data_queue.front());
        data_queue.pop();
    }
    bool try_pop(T& value)
    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return false;
        value=std::move(*data_queue.front());
        data_queue.pop();
        return true;
    }
    std::shared_ptr<T> wait_and_pop()
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk,[this]{return !data_queue.empty();});
        std::shared_ptr<T> res=data_queue.front();
        data_queue.pop();
        return res;
    }
    std::shared_ptr<T> try_pop()
    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return std::shared_ptr<T>();
        std::shared_ptr<T> res=data_queue.front();
        data_queue.pop();
        return res;
    }
    void push(T new_value)
    {
        std::shared_ptr<T> data(
            std::make_shared<T>(std::move(new_value)));
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);
        data_cond.notify_one();
    }
    bool empty() const
    {
        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }
};
```

Q2: Use shared-pointers

- Exception safe
- The allocation of the new instance can now be done outside the lock in push()
  - Beneficial for the performance of the queue
- Usage of one mutex limits the concurrency supported by this queue
  - By using the standard container (std::queue<>) you now have one data item that’s either protected or not
  - For fine-grained locking you need to take control of the detailed implementation of the data structure

A thread-safe queue using fine-grained locks

- Analyze the constituent parts and **associate one mutex with each distinct item**
  - Insert mutexes into the data structure itself, and thus we cannot simply make use of the STL library anymore

Q3: Single-threaded queue implementation

```
template<typename T>
class queue
{
private:
    struct node
    {
        T data;
        std::unique_ptr<node> next;
        node(T data_):
            data(std::move(data_))
        {}
    };
    std::unique_ptr<node> front;
    node* back;
```

```
public:
    queue(): back(nullptr)
    {}
    queue(const queue& other)=delete;
    queue& operator=(const queue& other)=delete;
    std::shared_ptr<T> try_pop()
    {
        if(!front)
        {
            return std::shared_ptr<T>();
        }
        std::shared_ptr<T> const res(
            std::make_shared<T>(std::move(front->data)));
        std::unique_ptr<node> const old_front=std::move(front);
        front=std::move(old_front->next);
        if(!front)
            back=nullptr;
        return res;
    }
    void push(T new_value)
    {
        std::unique_ptr<node> p(new node(std::move(new_value)));
        node* const new_back=p.get();
        if(back)
        {
            back->next=std::move(p);
        }
        else
        {
            front=std::move(p);
        }
        back=new_back;
    }
};
```

Q3: Single-threaded queue implementation

- Uses `std::unique_ptr<node>` to manage the nodes
  - Ensures that they (and the data they refer to) get deleted when they're no longer needed, without having to write an explicit `delete`
- Problems when adding a mutex for front and back
  - `push()` can modify both `front` and `back`
  - `push()` and `pop()` access the `next` pointer of a node. If there's a single item in the queue:
    - `push()` updates `back->next`, and `try_pop()` reads `front->next`
    - `front==back`, so both `front->next` and `back->next` are the same object ➔ requires protection
    - You can't tell if it's the same object without reading both `front` and `back`, you now have to lock the same mutex in both `push()` and `try_pop()`
      - Same serialization as before

Q4:Enabling concurrency by separating data

- Pre-allocate a dummy node with no data
  - Ensure that there's always at least one node in the queue to separate the node being accessed at the `front` from that being accessed at the `back`
  - Empty queue: `front` and `back` point to the dummy node
  - No race on front->next and back->next
  - Downside: add an extra level of indirection to store the data by pointer in order to allow the dummy nodes

Q4:Enabling concurrency by separating data

```
template<typename T>
class queue
{
private:
    struct node
    {
        std::shared_ptr<T> data;
        std::unique_ptr<node> next;
    };
    std::unique_ptr<node> front;
    node* back;
public:
    queue():
        front(new node),back(front.get())
    {}
    queue(const queue& other)=delete;
    queue& operator=(const queue& other)=delete;
    std::shared_ptr<T> try_pop()
    {
        if(front.get()==back)
        {
            return std::shared_ptr<T>();
        }
        std::shared_ptr<T> const res(front->data);
        std::unique_ptr<node> old_front=std::move(front);
        front=std::move(old_front->next);
        return res;
    }
    void push(T new_value)
    {
        std::shared_ptr<T> new_data(
            std::make_shared<T>(std::move(new_value)));
        std::unique_ptr<node> p(new node);
        back->data=new_data;
        node* const new_back=p.get();
        back->next=std::move(p);
        back=new_back;
    }
};
```

Q4:Enabling concurrency by separating data

Q5: Adding locks (fine-grained thread-safe queue)

```
1  template<typename T>
2  class threadsafe_queue
3  {
4  private:
5      struct node
6      {
7          std::shared_ptr<T> data;
8          std::unique_ptr<node> next;
9      };
10     std::mutex front_mutex;
11     std::unique_ptr<node> front;
12     std::mutex back_mutex;
13     node* back;
14     node* get_back()
15     {
16         std::lock_guard<std::mutex> back_lock(back_mutex);
17         return back;
18     }
19     std::unique_ptr<node> pop_front()
20     {
21         std::lock_guard<std::mutex> front_lock(front_mutex);
22
23         if(front.get()==get_back())
24         {
25             return nullptr;
26         }
27         std::unique_ptr<node> old_front=std::move(front);
28         front=std::move(old_front->next);
29         return old_front;
30     }
```

```
31 public:
32     threadsafe_queue():
33         front(new node),back(front.get())
34     {}
35     threadsafe_queue(const threadsafe_queue& other)=delete;
36     threadsafe_queue& operator=(const threadsafe_queue& other)=delete;
37     std::shared_ptr<T> try_pop()
38     {
39         std::unique_ptr<node> old_front=pop_front();
40         return old_front?old_front->data:std::shared_ptr<T>();
41     }
42     void push(T new_value)
43     {
44         std::shared_ptr<T> new_data(
45             std::make_shared<T>(std::move(new_value)));
46         std::unique_ptr<node> p(new node);
47         node* const new_back=p.get();
48         std::lock_guard<std::mutex> back_lock(back_mutex);
49         back->data=new_data;
50         back->next=std::move(p);
51         back=new_back;
52     }
53 };
```

Queue invariants

- `back->next==nullptr`
- `back->data==nullptr`
- `front==back` implies an empty list
- A single element list has `front->next==back`
- For each node `x` in the list, where `x!=back`, `x->data` points to an instance of T and x->next points to the next node in the list. `x->next==back` implies `x` is the last node in the list
- Following the next nodes from front will eventually yield back

Q6: An even more fine-grained lock-based queue

```
1  class FineQueue {
2    struct Node {
3      std::mutex mut;
4      Node *next = nullptr;
5      Data data{};
6    };
7
8    std::mutex mut_back;
9    Node *back;
10
11   std::mutex mut_front;
12   Node *front;
13
14 public:
15   FineQueue()
16       : mut_back{},
17         back{new Node{}},
18         mut_front{},
19         front{back} {}
20
21   ~FineQueue() {
22     while (front != nullptr) {
23       Node *next = front->next;
24       delete front;
25       front = next;
26     }
27   }
```

```
28   void enqueue(Data data) {
29     Node *new_node = new Node{};
30     std::scoped_lock lock{mut_back};
31     std::scoped_lock node_lock{back->mut};
32     back->data = data;
33     back->next = new_node;
34     back = new_node;
35   }
36   std::optional<Data> try_dequeue() {
37     Node *old_node;
38     {
39       std::scoped_lock lock{mut_front};
40       old_node = front;
41       std::scoped_lock node_lock{old_node->mut};
42       if (old_node->next == nullptr) {
43         return std::nullopt;
44       }
45       front = front->next;
46     }
47     Data data = old_node->data;
48     delete old_node;
49     return data;
50   }
51 };
```

Q6: An even more fine-grained lock-based queue

- Line 3: Per-node mutex
  - Synchronizes-with relationship between push and pop threads
- Line 38: Additional scope introduced in `try_pop()`
  - Avoid UAF: node_lock unlocks early, BEFORE we proceed to call delete old_node
  - Delete is expensive → release `mut_front` early

Lock-based data structures

- Mutexes are powerful mechanisms for ensuring that multiple threads can safely access a data structure without encountering race conditions or broken invariants
- The granularity of locking can affect the potential for true concurrency

Lock-free concurrent data structures

Blocking data structures

- Algorithms and data structures that use mutexes, condition variables, and futures to synchronize the data are called *blocking* data structures and algorithms
- The application calls library functions that will *suspend* the execution of a thread until another thread performs an action
- These library calls are termed *blocking* calls
  - The thread can't progress past this point until the block is removed
  - Typically, the OS will suspend a blocked thread completely (and allocate its time slices to another thread) until it's unblocked by the appropriate action of another thread, such as
    - unlocking a mutex,
    - notifying a condition variable, or
    - making a future ready

Nonblocking data structures

- Data structures and algorithms that do not use blocking library functions are said to be *nonblocking*
- Types of nonblocking data structures
    - Example: a spin lock is nonblocking, as it spins until the `test_and_set` is successful
  - **Obstruction-free**: if all other threads are paused, then any given thread will complete its operation in a bounded number of steps.
  - **Lock-free**: if multiple threads are operating on a data structure, then after a bounded number of steps one of them will complete its operation.
  - **Wait-free**: every thread operating on a data structure will complete its operation in a bounded number of steps, even if other threads are also operating on the data structure.

Lock-free data structures

- More than one thread must be able to access the data structure concurrently
  - They do not have to be able to do the same operations
  - If one of the threads accessing the data structure is suspended, the other threads must still be able to complete their operations without waiting for the suspended thread
  - Example: a lock-free queue might allow one thread to push and one to pop but break if two threads try to push new items at the same time

Wait-free data structures

- Data structures that avoid the following problem are wait-free
  - Lock-free algorithms with loops (using compare/exchange operations) can result in one thread being subject to *starvation*
    - If another thread performs operations with the “wrong” timing, the other thread might make progress but the first thread continually has to retry its operation
- Algorithms that can involve an unbounded number of retries because of clashes with other threads are not wait-free
- Writing wait-free data structures correctly is extremely hard
  - It’s all too easy to end up writing what’s essentially a spin lock

Pros of lock-free data structures

- Enable maximum concurrency
  - Some thread makes progress with every step
- Robustness
  - If a thread dies partway through an operation on a lock-free data structure, nothing is lost except that thread's data; other threads can proceed normally
  - You can not exclude threads from accessing the data structure
    - Ensure that the invariants are upheld or choose alternative invariants that can be upheld
    - Pay attention to the ordering constraints you impose on the operations
    - Ensure that changes become visible to other threads in the correct order

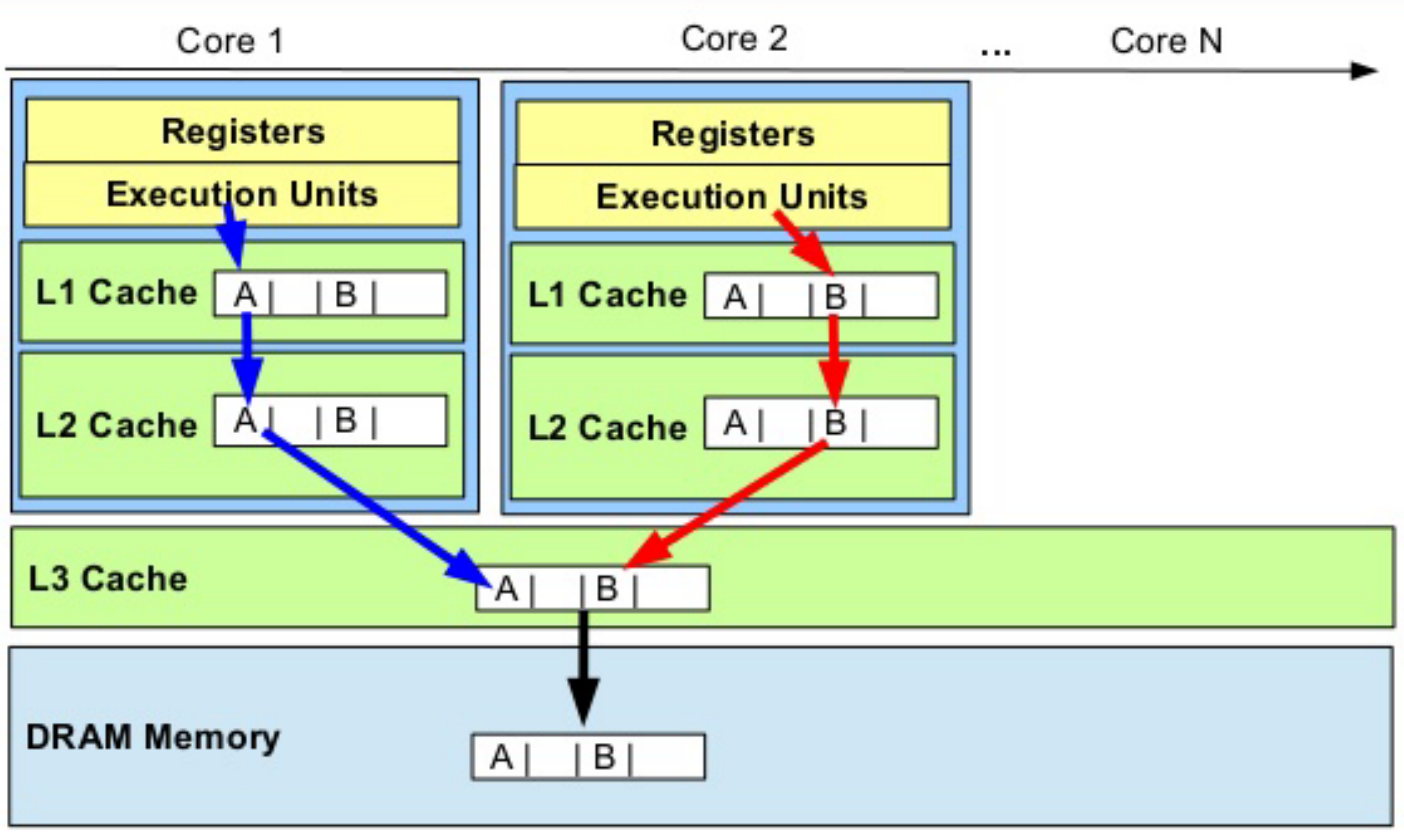
Cons of lock-free data structures

- Livelocks are possible
  - Two threads each try to change the data structure, but for each thread, the changes made by the other require the operation to be restarted, so both threads loop and try again
- Decrease overall performance, even though they reduce the time an individual thread spends waiting
  - Atomic operations used for lock-free code can be much slower than non-atomic operations
  - The hardware must synchronize data between threads that access the same atomic variables
    - Memory contention and write propagation
    - Cache ping-pong with multiple threads accessing the same atomic variables

Contention and cache ping pong

- If one of the threads modifies the data, this change then has to propagate to the cache on the other core, which takes time
- Depending on the memory orderings used for the operations, this modification may cause the second core to stop and wait for the change to propagate through the memory hardware
  - Extremely slow
  - Memory contention increases with the increase in number of threads
- Accessing data from the same cache line within multiple threads
  - Example: a mutex used by many threads
  - False-sharing can produce cache ping-pong as well, and it is more difficult to identify.

Problem: false sharing




```
struct foo {
    int x;
    int y;
};

static struct foo f;
```

```
int sum_a(void)
{
    int s = 0;
    for (int i = 0; i < 1000000; ++i)
        s += f.x;
    return s;
}
```

```
void inc_b(void)
{
    for (int i = 0; i < 1000000; ++i)
        ++f.y;
}
```

Guidelines for writing lock-free code

- Use std::memory_order_seq_cst for prototyping
- Use a lock-free memory reclamation scheme
  - Use some method to keep track of how many threads are accessing a particular object and delete each object when it is no longer referenced from anywhere
  - Recycle nodes
- Watch out for the ABA problem
  - Include an ABA counter alongside the variable
  - Prevalent when using free lists
- Identify busy-wait loops and help the other thread

Summary

- Safely and efficiently using synchronization primitives and atomics to implement thread-safe data structures
  - Lock-based
    - Granularity level in synchronization
  - Lock-free
    - Difficult to get right
  - Principles for design for concurrent data structures

References

- C++ Concurrency in Action, Second Edition
  - Chapters 6.1, 6.2, 7.1, 7.3