Concurrency in C++11/14 series

pabloxrl@gmail.com

Index

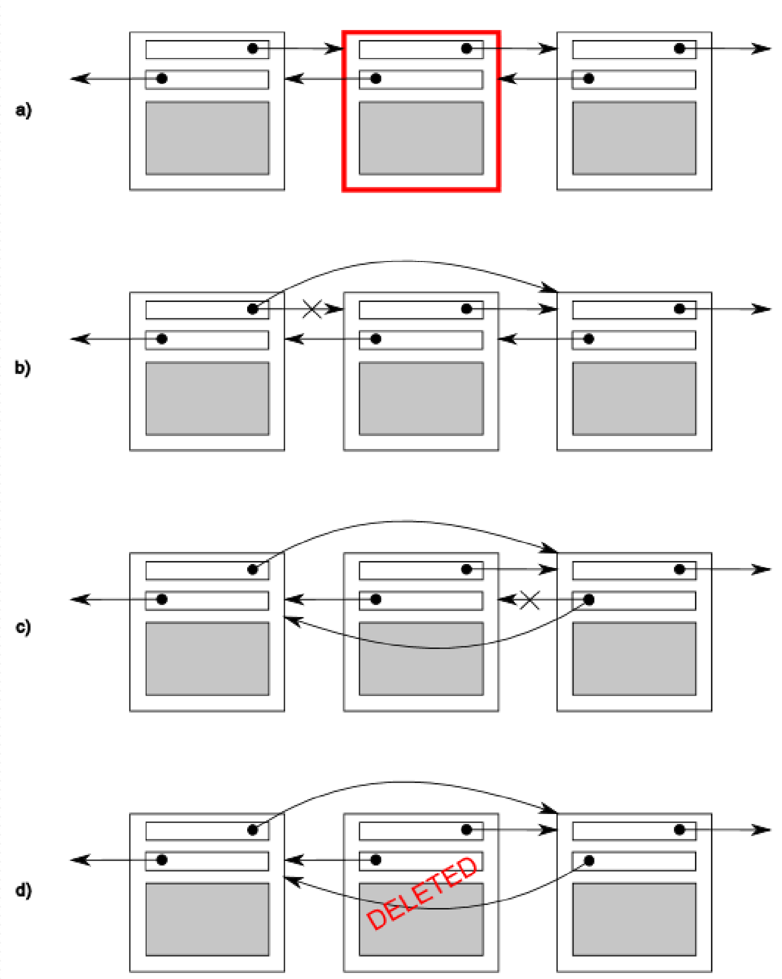
Problems with sharing data between threads

- Problems arise due to the consequences of modifying data shared between threads (read-only is ok)
- Handy concept: *Invariants*
- Example of invariant: For doubly linked list
  - If you follow a “next” pointer from one node (A) to another (B), the “previous” pointer from that node (B) points back to the first node (A).

Deleting element from doubly linked list

- Steps for deletion:
  - Identify node to delete (N)
  - Update link from N-1 to N+1
  - Update link from N+1 to N-1
  - Delete N

- Unless something is done other threads could see the list in an inconsistent state



Problems with sharing data between threads

- Race conditions are the biggest threat
  - Outcome depends on the relative ordering of execution
  - May lead to broken invariants
  - Ultimately can cause *undefined behavior*
- Ways to avoid race conditions
  - Wrap data structures with protection mechanisms
  - Modifications are done as a series of indivisible changes preserving invariants (lock free programming)
  - Handle updates as transactions

Protecting shared data with mutexes

- Using a mutex → *MUTually Exclusive*
  - 1- Lock the mutex
  - 2- Do stuff
  - 3- Unlock the mutex
- Using an instance of `std::mutex`
  - 1- Call `std::mutex::lock()`
  - 2- Do stuff
  - 3- Call `std::mutex::unlock()`

Protecting shared data with mutexes

- Using a mutex → *MUTually Ex*clusive
  - 1- Lock the mutex
  - 2- Do stuff
  - 3- Unlock the mutex

- ~~Using an instance of `std::mutex`~~
  - ~~1- Call `std::mutex::lock()`~~ Not recommended
  - ~~2- Do stuff~~
  - ~~3- Call `std::mutex::unlock()`~~

Protecting shared data with mutexes

- Using mutexes in C++14
  - Standard provides `std::lock_guard<T>` class template which implements RAII (scoped locking/unlocking on supplied mutex) (s2t01)
  - `std::lock_guard` and `std::mutex` are both declared in `<mutex>` header
  - Potential danger: Returning references to protected data or passing functions that access protected data. (s2t02, s2t03)

Protecting shared data with mutexes

- Neat tricks for avoiding deadlocks
  - Locking more than one object at once (s2t04)
  - Flexible locking with `std::unique_lock` (s2t05)
  - Trasferring ownership of locks (s2t06)

- Bonus:
  - Thread-safe lazy initialization of a class member using `std::call_once()` vs `std::mutex` (s2t07)
  - Example of actual usage of `std::call_once` (s2t08)

Protecting shared data with mutexes

- General guidelines for avoiding deadlocks
  - Avoid nested locks
  - Avoid calling user-supplied code while holding a lock
  - Acquire locks in a fixed order

- Extend these guidelines beyond locks

Protecting shared data with mutexes

- Structuring code for protecting shared data
  - Not as easy as slapping `std::lock_guard` everywhere
  - Spotting race conditions inherent in interfaces (Example: stack)
    - Top()
    - Pop()
    - Push()
    - Empty()
    - Size()

  - Example of definition of thread safe stack (s2t09)

Protecting shared data with mutexes

- Top():
  - Returning the object: copy or reference?
  - What about the results of empty() and size()
  - What about this code in a shared stack?

```
std::stack<int> s;
if(!s.empty())
{
   int const value = s.top(); // race #1
   s.pop(); // race #2
   do_something(value);
}
```

Protecting shared data with mutexes

- Possible solution: Combine top() + pop()
  - Option 1: Passing a reference:

```
std::vector<T> result;
some_stack.pop(result);
```

  - Disadvantages:

Protecting shared data with mutexes

- Possible solution: Combine top() + pop()
  - Option 1: Passing a reference:

```
std::vector<T> result;
some_stack.pop(result);
```

  - Disadvantages:
    - Having to construct an instance of the stack's value prior to the call
    - For certain types it can be too expensive
    - Some types may not be assignable

Protecting shared data with mutexes

- Possible solution: Combine top() + pop()
  - Option 2: Return a pointer to the popped item
  - Advantages:
    - Relatively cheap
  - Disadvantages:

Protecting shared data with mutexes

- Possible solution: Combine top() + pop()
  - <u>Option 2: Return a pointer to the popped item</u>

  - Advantages:
    - Relatively cheap

  - Disadvantages:
    - Having to manage the allocated memory (can be dealt with through std::shared_ptr)

Protecting shared data with mutexes

- Summary
  - How to use `std::mutex` and `std::lock_guard`
  - Avoiding deadlock with `std::lock`
  - Alternative data protection facilities like locking hierarchy or `std::call_once`
  - Example of a broken-by-design thread-(un)safe interface
- Next meeting:
  - Waiting for events and using atomics
  - `std::future`

The end

- Any questions?