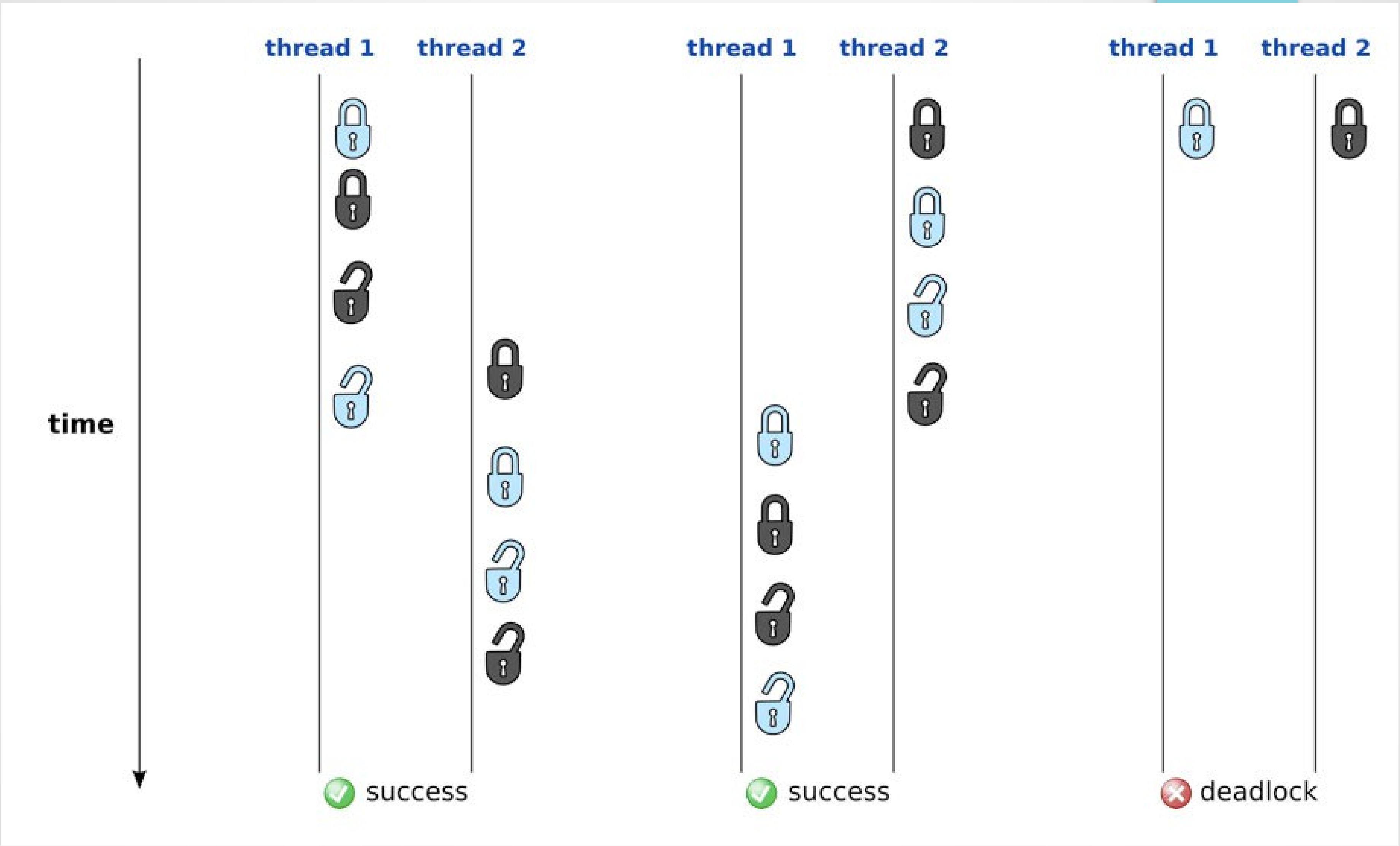
# C++ Threads, Atomic & Mutex




Krishna Kumar

# C++11 (Threads)

- Perhaps one of the biggest change to the language is the addition of multithreading support. Before C++11 – OpenMP, MPI to target multicore systems.
- **std::threads**
  - The class thread represents a single thread of execution. Threads allow multiple pieces of code to run asynchronously and simultaneously.
- Starting a new thread is very easy. When you create an instance of a std::thread, it will automatically be started.
- When you create a thread you have to give it the code it will execute. The first choice for that, is to pass it a function pointer.
- Calling join() function forces the current thread to wait for the other one (in this case, the main thread has to wait for the thread t1 to finish). If you omit this call, the result is undefined.

# Synchronisation

```
class Counter {
  int value_;
public:
  Counter() : value_(0){}
  void increment() { ++value_; }
  int value() { return value_; }
  void value(int& newvalue)
{ value_ = newvalue; }
};
```

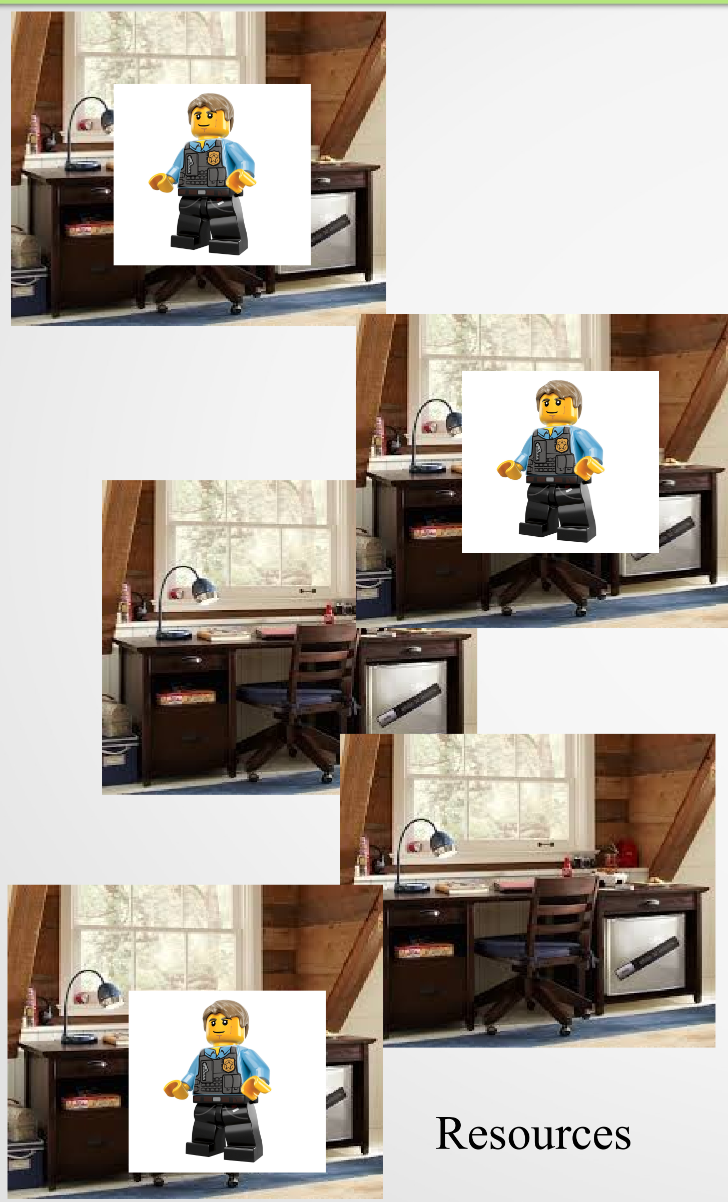
```
for (int i = 0; i < 500; ++i) {

threads.push_back(std::thread([
&counter]() {
    for (int i = 0; i < 100; ++i)
      counter.increment();
  }));
}
// After thread.join()
 std::cout << counter.value() ;
```

# Synchronisation issues

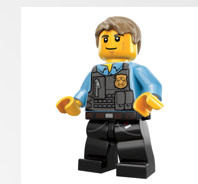
- The problem is that the incrementation is not an atomic operation. As a matter of fact, an incrementation is made of three operations:
  - Read the current value of value
  - Add one to the current value
  - Write that new value to value
- When you run that code using a single thread, there are no problems. But on several threads:
  - Thread 1 : read the value, get 0, add 1, so value = 1
  - Thread 2 : read the value, get 0, add 1, so value = 1
  - Thread 1 : write 1 to the field value and return 1
  - Thread 2 : write 1 to the field value and return 1
- These situations come from what we call interleaving.

# Semaphores



Resources


Semaphore



Processes

# C++ Mutex

- The C+11 standard provides std::mutex primitive. A mutex object also provides member functions – lock() and unlock( ) - to explicitly lock or unlock a mutex.

```
void increment(){
    mutex.lock();
    ++value;
    mutex.unlock();
}
```

- The first one enable a thread to obtain the lock and the second releases the lock. The lock() method is blocking. The thread will only return from the lock() method when the lock has been obtained.

# Locks and Exceptions

- Imagine that the Counter has a decrement operation that throws an exception if the value is 0:

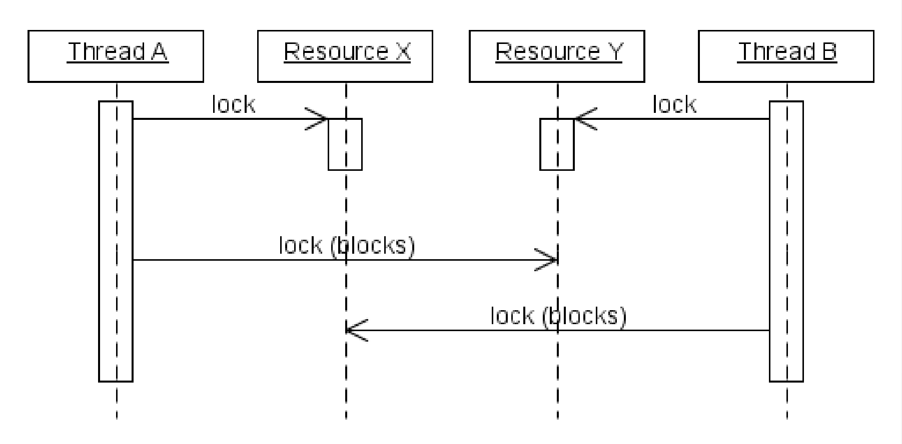
```
void decrement(){
    if(value == 0){
        throw "Value cannot be less than 0";
    }
    --value;
}
```

- This wrapper works well in most of the cases, but when an exception occurs in the decrement method, you have a big problem. Indeed, if an exception occurs, the unlock() function is not called and so the lock is not released. As a consequence, you program is completely blocked.

# Atomic

- The C++11 Concurrency Library introduces Atomic Types as a template class: std::atomic. You can use any Type you want with that template and the operations on that variable will be atomic and so thread-safe.
- If you want to make a big type (let's saw 2MB storage), you can use std::atomic as well, but mutexes will be used. In this case, there will be no performance advantage.
- The main functions that std::atomic provide are the store and load functions that atomically set and get the contents of the std::atomic.

# Recursive Lock – Deadlock!



# References


- http://baptiste-wicht.com/posts/2012/03/cpp11-concurrency-part1-start-threads.html
- http://baptiste-wicht.com/posts/2012/03/cp11-concurrency-tutorial-part-2-protect-shared-data.html
- http://baptiste-wicht.com/posts/2012/04/c11-concurrency-tutorial-advanced-locking-and-condition-variables.html