

# **CS100**

# **Introduction to Programming**

## **Lecture 24. Concurrency**

# Today's learning objectives

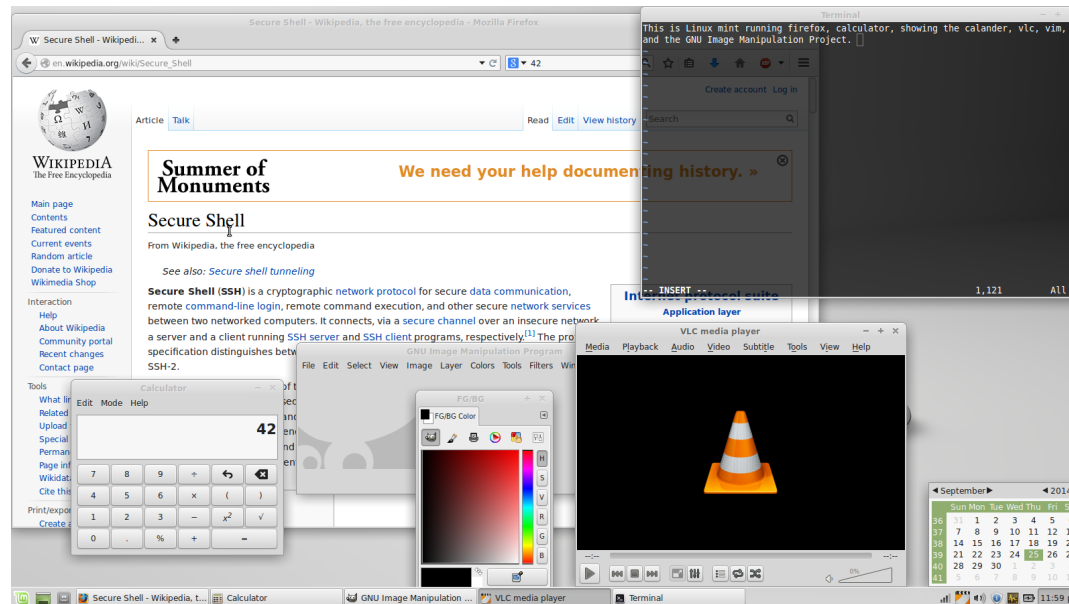
- Understand the need for/meaning of multi-tasking, concurrency, and parallel computation
- Understand the difference between processes and threads
- Learn how to implement them
- Learn about thread safety and how to implement it
- Learn about thread synchronization

# Outline

- Multi-tasking, concurrency, and parallel processing
- `std::thread`
- `std::mutex`
- `std::lock_guard`
- `std::atomic`
- Thread synchronization

# Multi-tasking, Concurrency, and Parallel Computing

- What is multi-tasking?
  - Multi-tasking is one of the main functionalities supported by an operating system. It allows the quasi-parallel execution of multiple processes



Source:  
Wikipedia

# Multi-tasking, Concurrency, and Parallel Computing

- What is concurrency?
  - More general
  - Execution of several computations at overlapping times
  - Concurrency can happen at the level of:
    - Network (cloud computing)
    - Computer / OS (multi-tasking, multiple processes)
    - Program (multiple threads)

# Multi-tasking, Concurrency, and Parallel Computing

- What is parallel computing?
  - Strictly parallel execution of (possibly same) computations
  - Requires parallel computing hardware
    - Multi-core processor
    - Graphics Processing Unit (GPU)
    - Field Programmable Gate Array (FPGA)
    - Derived ASICs
    - Specialized software-programmable SoCs (Ambarella etc.)

# A process

- A process is ...
  - ... started at the operating system level
  - ... assigned a space of individual memory that is typically not shared with other processes
  - ... communicating with other processes via other interfaces (network, disk space, etc.)

# A thread

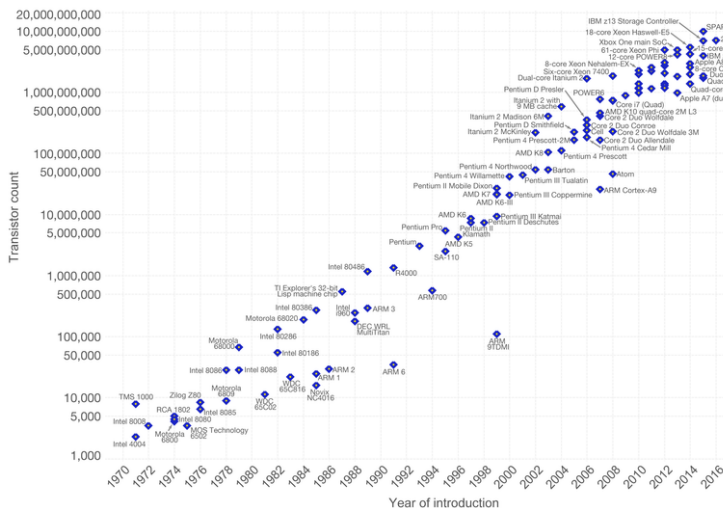
- A thread is ...
  - ... started at the process/program level
  - ... granted access to the memory space that has been allocated to the process
  - ... possibly sharing that memory space with other threads from the same process
  - ... able to communicate directly with other threads through the assigned memory
  - ... a means to realize parallel computing



# Why is concurrency necessary?

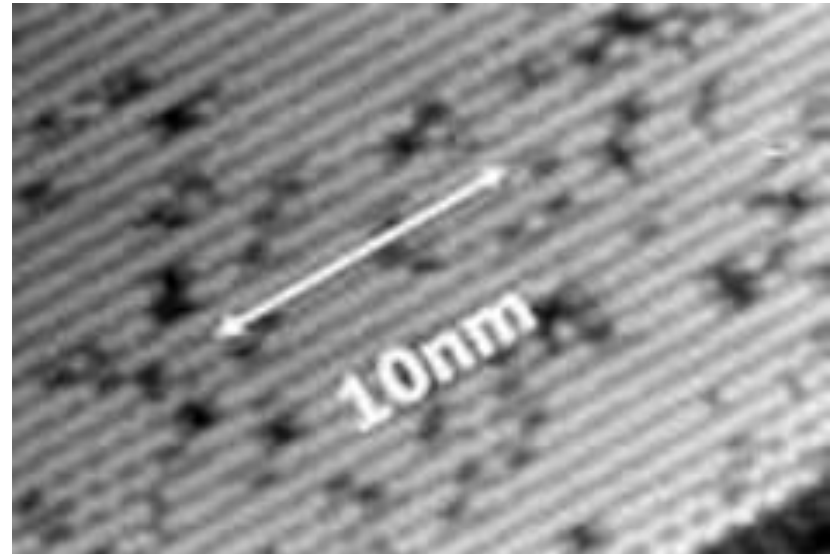
- Moore's law is dying
  - Currently ~10nm process technology (distance refers to half the distance between identical features in array)
  - That's about 20 silicon atoms!

**Moore's Law – The number of transistors on integrated circuit chips (1971-2016)**   
Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.



Data source: Wikipedia ([https://en.wikipedia.org/wiki/Transistor\\_count](https://en.wikipedia.org/wiki/Transistor_count))  
The data visualization is available at OurWorldInData.org. There you find more visualizations and research on this topic.

Licensed under CC-BY-SA by the author Max Roser.



# Why is concurrency necessary?

- Size of gate/transistor limits the processor clock!
  - Higher frequency -> More dissipated heat!
  - Smaller transistors "can't take it"
- → We want ever more powerful computing resources
- → Concurrency!

# Why is concurrency possible?

- Tasks can often be naturally divided into multiple (often simpler) sub-tasks
  - Divide and Conquer
- Many problems are embarrassingly parallel in their nature
  - Matrix manipulations
  - Image processing
  - ...

# Outline

- Multi-tasking, concurrency, and parallel processing
- **std::thread**
- std::mutex
- std::lock\_guard
- std::atomic
- Thread synchronization

# Concurrency in C++

- Original C++ Standard (1998) only supported single thread programming
  - Requirement for other libraries (e.g. pthread) to access POSIX threads functionality
- Since C++ 11:
  - Acknowledges the existence of multi-threaded programs
  - Provides interface to create/synchronize threads
  - Introduces memory models for concurrency

# Requirements

Start of  
main  
process

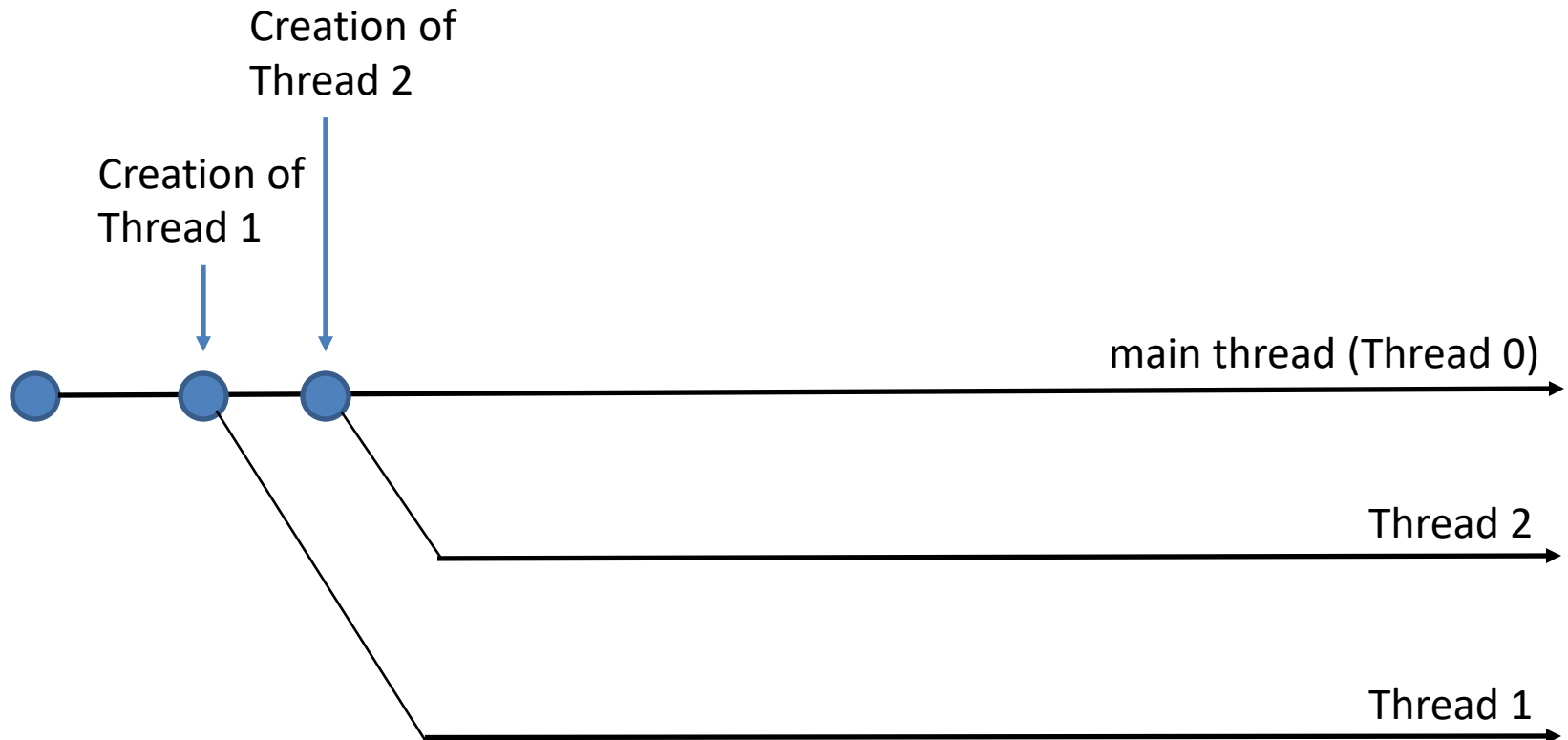


main thread (Thread 0)



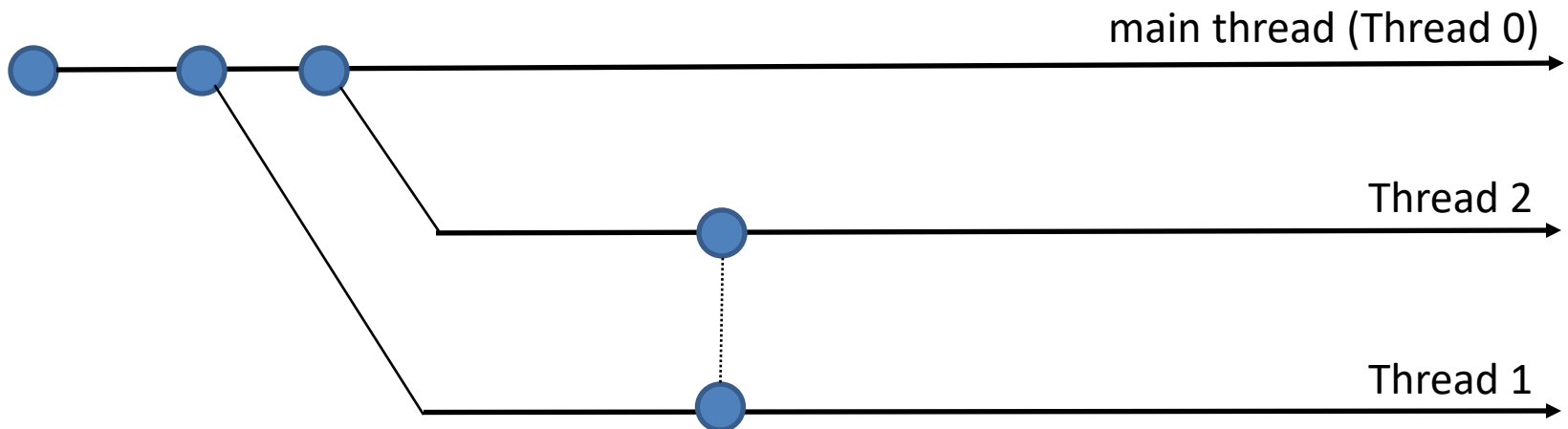
# Requirements

- Thread creation
  - Ability to start a new thread (i.e. from the main thread)



# Requirements

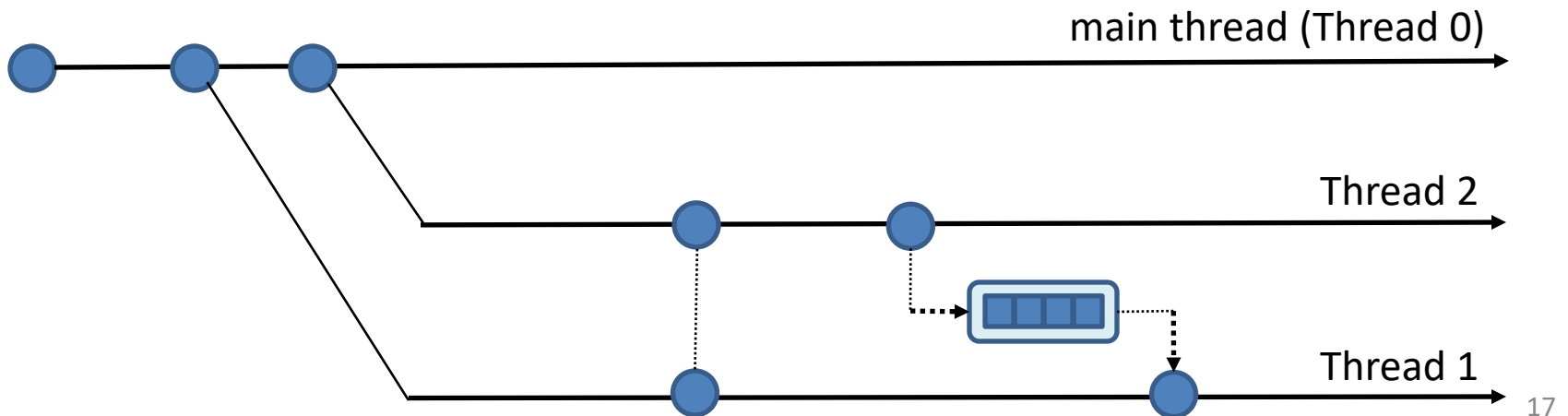
- Thread synchronization
  - Ability to establish timing relationships between threads
    - Example: One thread waits until another thread has reached a certain point in its code
    - Example: One thread is ready to transmit information while the other is ready to receive the message, simultaneously





# Requirements

- Thread communication
  - Ability to correctly transmit data among different threads



# Thread creation

- Use C++ 11!
- Use thread library

```
#include <thread>
```

- Creating an instance of `std::thread` automatically starts a new thread

```
std::thread th( threadFunction );
```

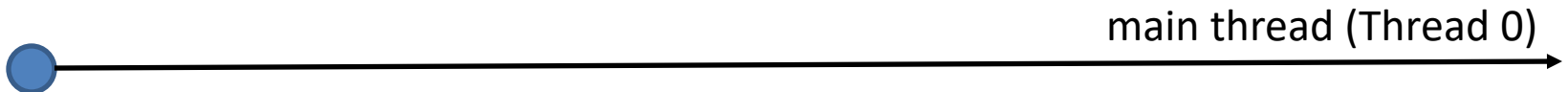
# Thread creation

- Example:

```
#include <thread>
#include <iostream>

void threadFunction() {
    std::cout << "Hello from thread 1\n";
}

int main() {
    std::thread th(threadFunction);
    std::cout << "Hello from thread 0\n";
    th.join();
}
```



# Thread creation

- Example:

```
#include <thread>
#include <iostream>
```

```
void threadFunction() {
    std::cout << "Hello from thread 1\n";
}
```

```
int main() {
    std::thread th(threadFunction);
    std::cout << "Hello from thread 0\n";
    th.join();
}
```

Thread creation



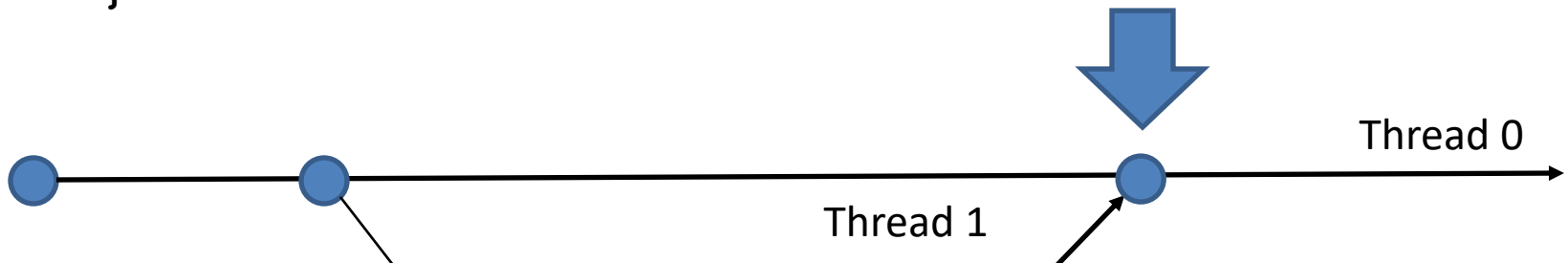
# Thread joining

- Main thread waits for other threads to finish!

```
#include <thread>
#include <iostream>

void threadFunction() {
    std::cout << "Hello from thread 1\n";
}

int main() {
    std::thread th(threadFunction);
    std::cout << "Hello from thread 0\n";
    th.join();
}
```



# Outline

- Multi-tasking, concurrency, and parallel processing
- `std::thread`
- **`std::mutex`**
- `std::lock_guard`
- `std::atomic`
- Thread synchronization

# What are critical sections?

- Data is usually shared between threads
- Problem:
  - Multiple threads access the same object at the same time
  - If operation is atomic (i.e. not divisible)
    - No other thread could read/operate on a partial result
    - It is safe!
  - If operation is not atomic (i.e. divisible into several steps)
    - Other threads could read/operate on partial result if switching happens in between
    - It is not safe!

# Example

- 5 threads increase the same counter 5000 times

```
#include <vector>
#include <thread>
#include <iostream>
```

```
class Counter {
public:
    Counter() {
        m_value = 0;
    };

    int getValue() {
        return m_value;
    };
    void increment() {
        ++m_value;
    };
private:
    int m_value;
};
```

```
void incrementCounterManyTimes(
    Counter & counter ) {
    for( int i = 0; i < 5000; i++ ) {
        counter.increment();
    }
}

int main() {
    Counter counter;
    std::vector<std::thread> threads;
    for( int i = 0; i < 5; i++ ) {
        threads.push_back( std::thread(
            incrementCounterManyTimes,
            std::ref(counter) ) );
    }
    for( int i = 0; i < 5; i++ ) {
        threads[i].join();
    }
    std::cout << counter.getValue() << "\n";
    return 0;
}
```

Reference  
needed!



# Example

- Result
    - Program has synchronization issues!
    - Possible outputs
      - 9840, 6102, 11952, 8740, 10515, 11635, 8490, 15170, 7202, 6218
      - The output is different every time!
    - What has happened?
      - "increment" is not an atomic operation!
        - It first reads the value
        - It adds one
        - Then copies the result back
- ← Thread switching at either of these points will cause a **data race!**

# What are critical sections?

- Critical section
  - A piece of code that accesses/modifies a shared resource, and the access/modification is non-atomic
  - → Access must not be concurrent!
  - → Simultaneous access by multiple threads must be prevented
  - → Access needs to be mutually exclusive!

# How to protect shared data?

- Solutions
  - Semaphores
  - Mutexes (binary semaphores)
  - Monitors (guarantees only one time can be active within a monitor at a time (supported in Java))
  - Condition variables
  - Compare-and-wrap: The idea is to compare the contents of a memory location to a given value and, only if they are the same, modifies the contents of that memory location to a given new value
  - etc.

# Mutex

- Mutexes (named after mutual exclusion) enable us to
  - mark critical sections as mutually exclusive
  - if any thread enters that critical section, any other thread that tries to enter a critical section that is marked by the same mutex has to wait!

# Mutex

- How does it work?
  - Create a mutex by creating an instance of `std::mutex`
  - Lock it with a call to the member function `lock()`
  - Unlock it with a call to the member function `unlock()`

```
class Counter {  
public:  
    Counter() { m_value = 0; };  
  
    int getValue() { return m_value; };  
    void increment() {  
        m_mutex.lock();  
  
        ++m_value;  
        m_mutex.unlock();  
    };  
};
```

```
private:  
    int m_value;  
  
    std::mutex m_mutex;  
};
```

Output:  
**25000 every time!**

# Mutex

- How does it work?
  - A mutex does not directly lock a part of the code
  - A mutex is a resource (i.e. a lock), and we use it passively to protect a critical section
  - **lock()** is blocking until it “has the lock”
  - **unlock()** “releases the lock”
  - Only one at a time can have the lock
  - → Bound all critical sections (w.r.t. to the same data) by the same mutex

# Outline

- Multi-tasking, concurrency, and parallel processing
- `std::thread`
- `std::mutex`
- `std::lock_guard`
- `std::atomic`
- Thread synchronization

# Problems with Mutex

- It is not wise to call **lock()** directly
  - You have to remember to **unlock()** on every code path out of a critical section (including those due to exceptions)
- Use **std::lock\_guard**



# std::lock\_guard

- Example:

```
void increment() {  
    std::lock_guard<std::mutex> lock(m_mutex);  
    ++m_value;  
};
```

- `m_mutex.lock()` is called when the instance is constructed
- `m_mutex.unlock()` is called when the instance is destructed
- → `lock_guard` locks the mutex for the duration of the scope in which `lock_guard` is defined

# **std::lock\_guard**

- Also called the RAII idiom  
(resource acquisition is initialization)

# Advanced locking with mutexes

- Recursive locking:
  - `std::recursive_mutex`
  - Enables the same thread to lock the mutex twice
  - Useful in the context of recursive functions

# Advanced locking with mutexes

- Timed locking:
  - `std::timed_mutex`
  - `std::timed_recursive_mutex`
  - Similar to `std::mutex`, but enables a thread to do something else while waiting for another thread to **unlock**
  - Additional functionality:

`try_lock` : tries to lock, returns false if failed

`try_lock_for` : tries to lock for specified time

`try_lock_until` : tries to lock until specified time

# std::call\_once

- It is possible that some operations are to be done only once
- Use

```
std::call_once( std::once_flag, function );
```

# std::call\_once Example

```
#include <iostream>
#include <thread>
#include <mutex>
```

```
std::once_flag flag1;
void printHello() { std::cout << "Hello\n"; }
void threadFunction() {
    std::call_once(flag1, printHello);
}
```



Conditional variable



Unique call

```
int main() {
    std::thread st1(threadFunction);
    std::thread st2(threadFunction);
    std::thread st3(threadFunction);
    st1.join();
    st2.join();
    st3.join();
    return 0;
}
```

# Outline

- Multi-tasking, concurrency, and parallel processing
- `std::thread`
- `std::mutex`
- `std::lock_guard`
- **`std::atomic`**
- Thread synchronization

# **std::atomic**

- C++11 introduces atomic types as a generic template class that can be wrapped around any type

```
std::atomic<Type> object;
```

- Can be used with any type
- Makes the operations on that type atomic
- Locking technique depends on type, and can be very fast for small objects (faster than mutex!)



# Example atomic

- Back to our counter example

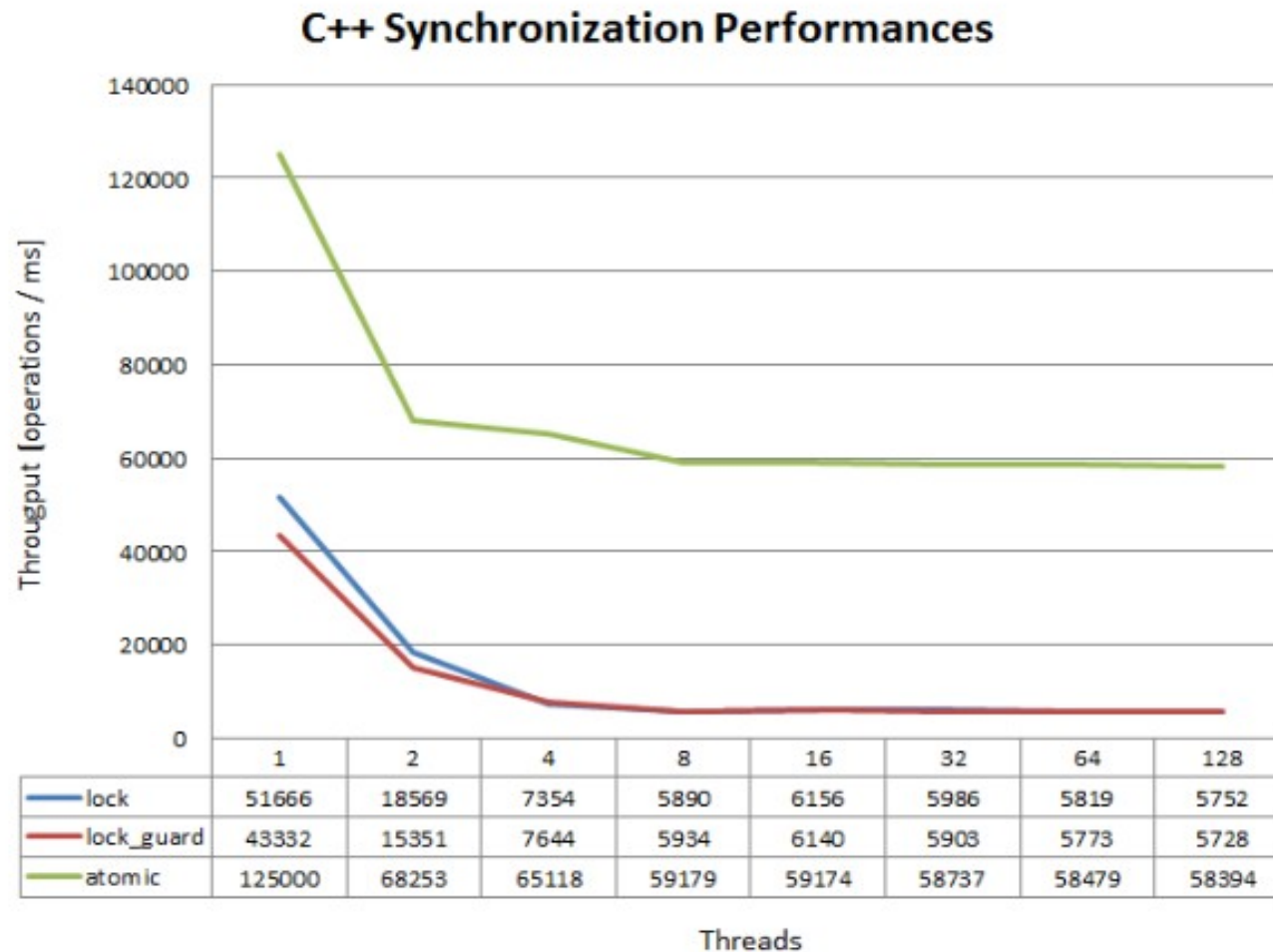
```
class Counter {  
public:  
    Counter() { m_value = 0; };  
  
    int getValue() { return m_value; };  
    void increment() {  
        m_mutex.lock();  
  
        ++m_value;  
        m_mutex.unlock();  
    };  
  
private:  
    int m_value;  
  
    std::mutex m_mutex;  
};
```

# Example atomic

- Back to our counter example

```
class Counter {  
public:  
    Counter() { m_value = 0; };  
  
    int getValue() { return m_value; };  
    void increment() {  
        ++m_value;  
    };  
  
private:  
    std::atomic<int> m_value;  
};
```

# Speed comparison



# Outline

- Multi-tasking, concurrency, and parallel processing
- `std::thread`
- `std::mutex`
- `std::lock_guard`
- `std::atomic`
- Thread synchronization

# Synchronization between threads

- Apart from just protecting data, sometimes we may wish for one thread to wait until another thread has something done
- In C++:
  - Conditional variables
  - Futures

# **std::condition\_variable**

- A synchronization primitive that can be used to block a thread or multiple threads at the same time, until
  - A notification is received from another thread
  - A time-out expires

# `std::condition_variable`

- A thread that intends to wait on `std::condition_variable` has to acquire a `std::unique_lock` first
- The wait operations atomically release the mutex and suspend the execution of the thread
- When the condition variable is notified, the thread is awakened, and the mutex is reacquired

# Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;
```



Mutex to protect resource

```
void data_preparation_thread() {
    while( more_data_to_prepare() ) {
        data_chunk data = prepare_data();
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);
        data_cond.notify_one();
    }
}
```

```
void data_processing_thread() {
    while(true) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, []{return !data_queue.empty();});
        data_chunk data = data_queue.front();
        data_queue.pop();
        lk.unlock();
        process(data);
        if(is_last_chunk(data))
            break;
    }
}
```



# Example

```
std::mutex mut;  
std::queue<data_chunk> data_queue;  
std::condition_variable data_cond;
```

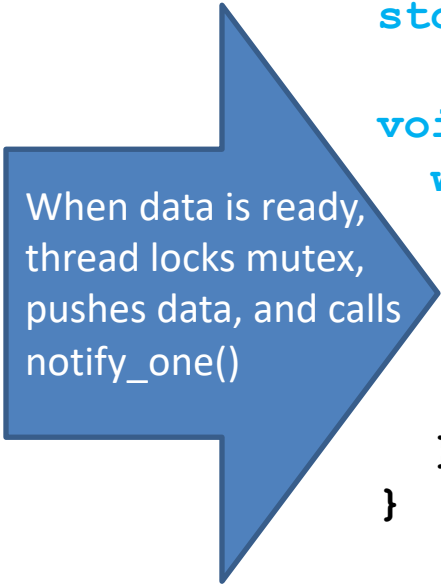


Queue used to pass data

```
void data_preparation_thread() {  
    while( more_data_to_prepare() ) {  
        data_chunk data = prepare_data();  
        std::lock_guard<std::mutex> lk(mut);  
        data_queue.push(data);  
        data_cond.notify_one();  
    }  
}
```

```
void data_processing_thread() {  
    while(true) {  
        std::unique_lock<std::mutex> lk(mut);  
        data_cond.wait(lk, []{return !data_queue.empty();});  
        data_chunk data = data_queue.front();  
        data_queue.pop();  
        lk.unlock();  
        process(data);  
        if(is_last_chunk(data))  
            break;  
    }  
}
```

# Example



When data is ready,  
thread locks mutex,  
pushes data, and calls  
notify\_one()

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

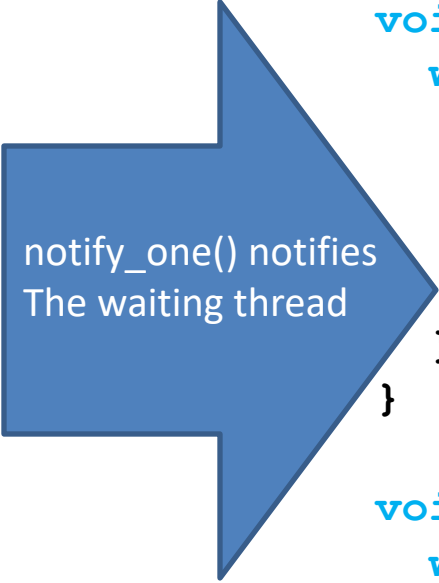
void data_preparation_thread() {
    while( more_data_to_prepare() ) {
        data_chunk data = prepare_data();
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);
        data_cond.notify_one();
    }
}

void data_processing_thread() {
    while(true) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, []{return !data_queue.empty();});
        data_chunk data = data_queue.front();
        data_queue.pop();
        lk.unlock();
        process(data);
        if(is_last_chunk(data))
            break;
    }
}
```

# Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
    while( more_data_to_prepare() ) {
        data_chunk data = prepare_data();
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);
        data_cond.notify_one();
    }
}
```



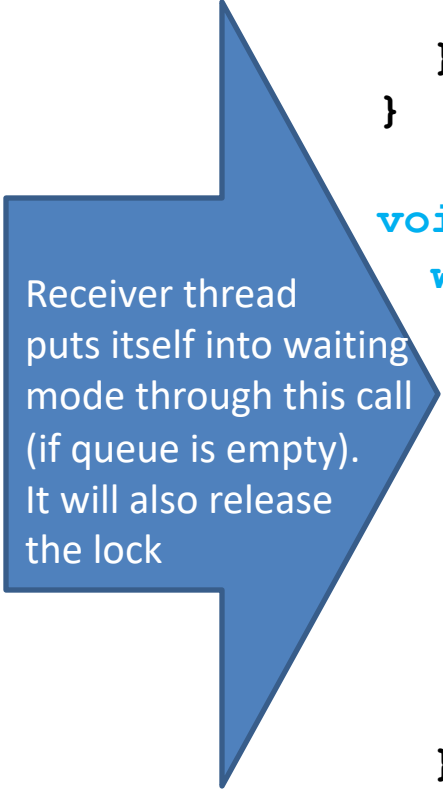
notify\_one() notifies  
The waiting thread

```
void data_processing_thread() {
    while(true) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, []{return !data_queue.empty();});
        data_chunk data = data_queue.front();
        data_queue.pop();
        lk.unlock();
        process(data);
        if(is_last_chunk(data))
            break;
    }
}
```

# Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;
```

```
void data_preparation_thread() {
    while( more_data_to_prepare() ) {
        data_chunk data = prepare_data();
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);
        data_cond.notify_one();
    }
}
```



Receiver thread  
puts itself into waiting  
mode through this call  
(if queue is empty).  
It will also release  
the lock

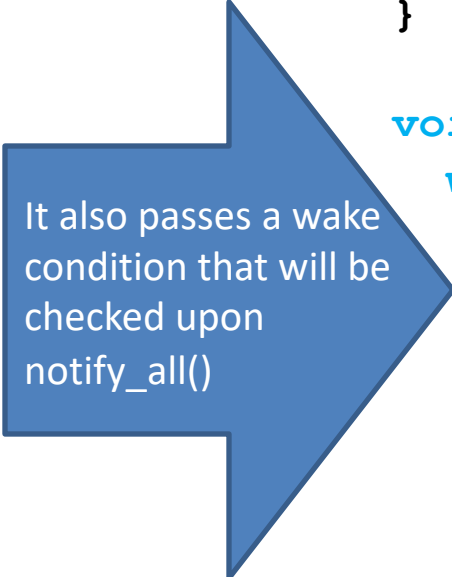
```
void data_processing_thread() {
    while(true) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, []{return !data_queue.empty();});
        data_chunk data = data_queue.front();
        data_queue.pop();
        lk.unlock();
        process(data);
        if(is_last_chunk(data))
            break;
    }
}
```

# Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;
```

```
void data_preparation_thread() {
    while( more_data_to_prepare() ) {
        data_chunk data = prepare_data();
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);
        data_cond.notify_one();
    }
}
```

```
void data_processing_thread() {
    while(true) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, []{return !data_queue.empty();});
        data_chunk data = data_queue.front();
        data_queue.pop();
        lk.unlock();
        process(data);
        if(is_last_chunk(data))
            break;
    }
}
```




It also passes a wake condition that will be checked upon notify\_all()

# Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;
```

```
void data_preparation_thread() {
    while( more_data_to_prepare() ) {
        data_chunk data = prepare_data();
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);
        data_cond.notify_one();
    }
}
```

```
void data_processing_thread() {
    while(true) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, []{return !data_queue.empty();});
        data_chunk data = data_queue.front();
        data_queue.pop();
        lk.unlock();
        process(data);
        if(is_last_chunk(data))
            break;
    }
}
```



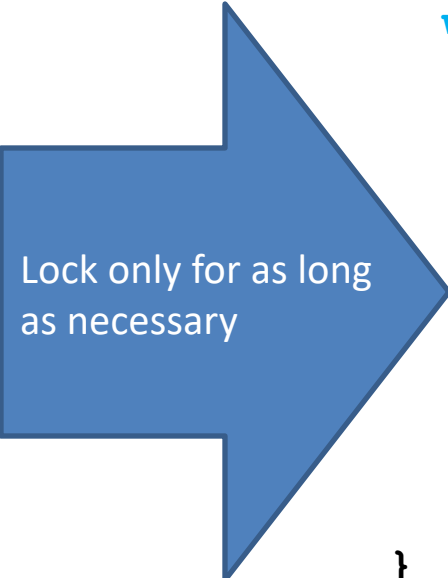
The mutex will be automatically locked once the wait terminates

# Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;
```

```
void data_preparation_thread() {
    while( more_data_to_prepare() ) {
        data_chunk data = prepare_data();
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(data);
        data_cond.notify_one();
    }
}
```

```
void data_processing_thread() {
    while(true) {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, []{return !data_queue.empty();});
        data_chunk data = data_queue.front();
        data_queue.pop();
        lk.unlock();
        process(data);
        if(is_last_chunk(data))
            break;
    }
}
```



Lock only for as long  
as necessary