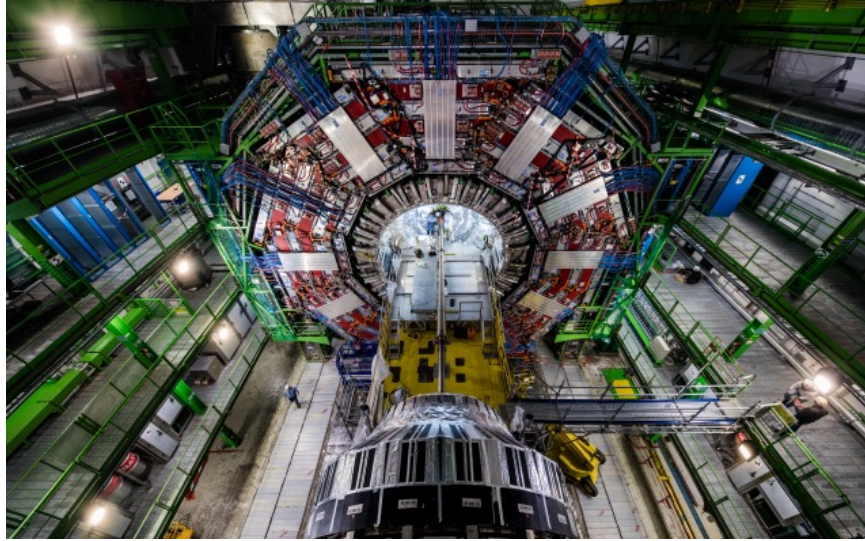


Software Design in the Many-Cores era

A. Gheata, S. Hageböck

CERN, EP-SFT/IT

CERN School of Computing 2022



Lecture II

Base Concepts of Parallel Programming: A Pragmatic Approach

Outline of This Lecture

The Goals:

- 1) *Become familiar with the basic concepts of parallel programming through the discussion of concrete examples in C++*
- 2) *Know what is behind the scenes of a task-based approach*
- 3) *Be able to start developing parallel applications.*

- Concurrency: asynchronous execution and threads
- Synchronisation: design principles, replication, atomics, transactions and locks

C++: A Reminder

- The approach of this lecture is **pragmatic**.
 - “Forward declarations” to concepts treated later will be used!
 - Concepts are illustrated through **concrete examples** involving C++ constructs.
- **C++ is the programming language of HEP** for frameworks, event generators, simulation toolkits, analysis and reconstruction applications (number crunching code!)
 - Python is also widespread for configuration, analysis and scripting
- C++: “The power, elegance and simplicity of a hand grenade”



Object Orientation

C++ allows OO programming.

Now, are objects good?

Object Orientation

C++ allows OO programming.

Now, are objects good?

- Well, yes, it is easier to describe concepts this way
 - Using features like: inheritance, encapsulation, polymorphism

Most of the HEP code moved from FORTRAN to C++ in the early 90s

Object Orientation

C++ allows OO programming.

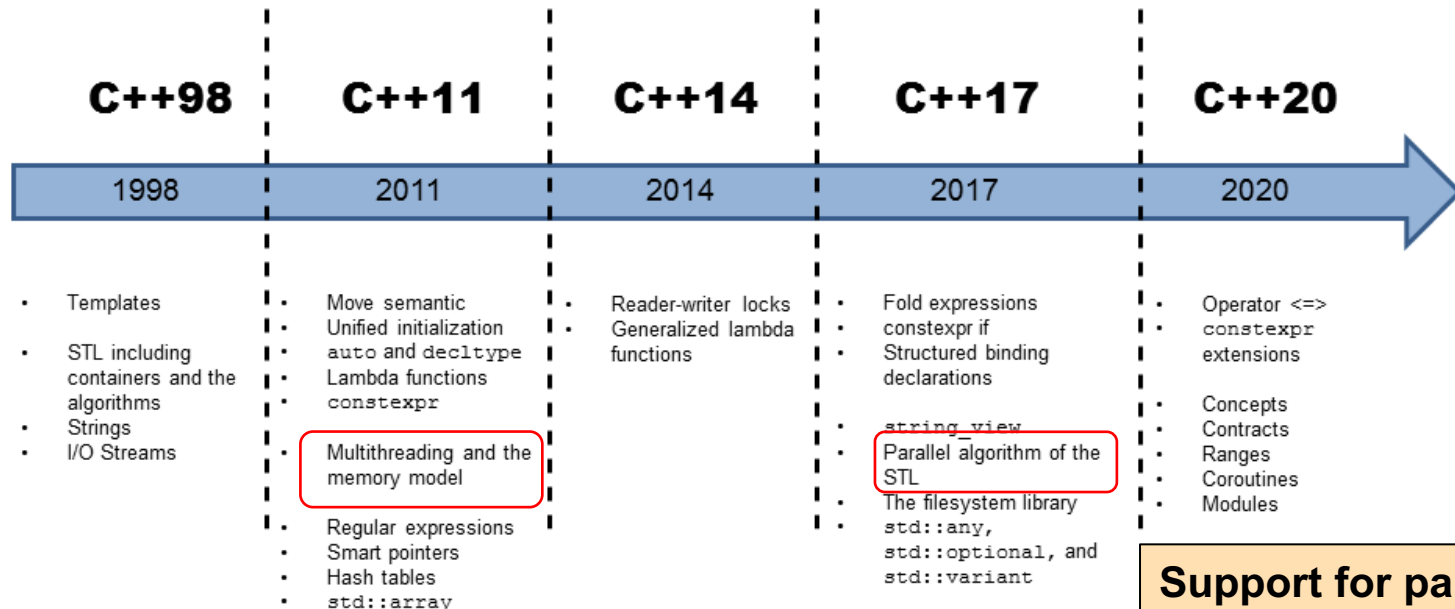
Now, are objects good?

All of the HEP code moved from FORTRAN to C++ in the early 90s

- Well, yes, it is easier to describe concepts this way
 - Using features like: inheritance, encapsulation, polymorphism
- And no, it is very easy to abuse OOP, creating inefficient code
 - Search for: “[OOP best practices](#)” and “[OOP anti-patterns](#)”...
 - CPU/memory price tags for the features above
- The data layout and access pattern are essential for performance
 - Keyword: Data Oriented Design (re-design?)

C++ Evolves!

- A committee reviews the C++ standard
 - CERN is part of it!

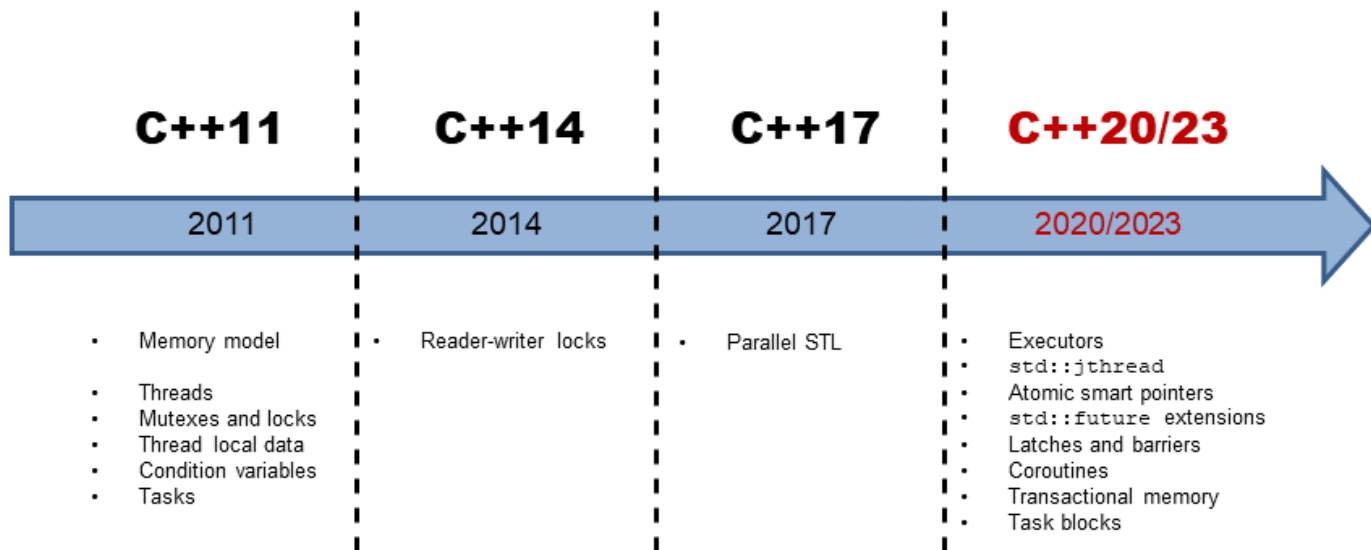


–std=c++NN
switch to activate standard <NN>

Support for parallelism gradually introduced and evolved

C++ concurrency evolves!

- A committee reviews the C++ standard
 - CERN is part of it!

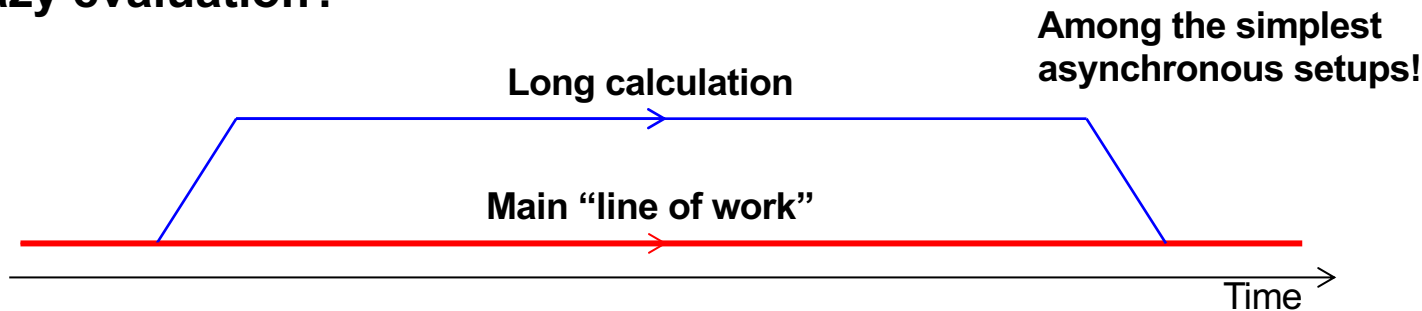


Concurrency



Asynchronous Task Execution

- Problem: a **long calculation**, the result of which **is not immediately needed**
- Possible solution: **asynchronous execution** of the calculation, retrieval of the result at a later stage
- Nuances: result may or may not be **needed later** depending on the control flow steering the application
 - **Lazy evaluation?**



std::async

- A solution is provided by the standard library natively: `std::async`
 - `#include <future>`
- **Execute a function concurrently in a separate thread** or on demand when the result is needed (lazily)
- **Result is a `std::future` : a “bridge”** between the two locations:
 - `std::future` **“transports” results and exceptions from *thread* to *thread***
- In other words, code to be executed is passed around

std::async in Action

```
#include <future> // Header for async and future
#include <iostream>

int lengthyCalculation(){ /* independent calculation */ };
void doOtherStuff(){ /* do work here */ };

int main() {
    std::future<int> myAnswer = std::async(lengthyCalculation);
    doOtherStuff();
    std::cout << "The result is: " << myAnswer.get() << std::endl;
}
```

std::async in Action

```
#include <future> // Header for async and future
#include <iostream>

int lengthyCalculation(){ /* independent calculation */ };
void doOtherStuff(){ /* do work here */ };

int main() {
    std::future<int> myAnswer = std::async(lengthyCalculation);
    doOtherStuff();
    std::cout << "The result is: " << myAnswer.get() << std::endl;
}
```

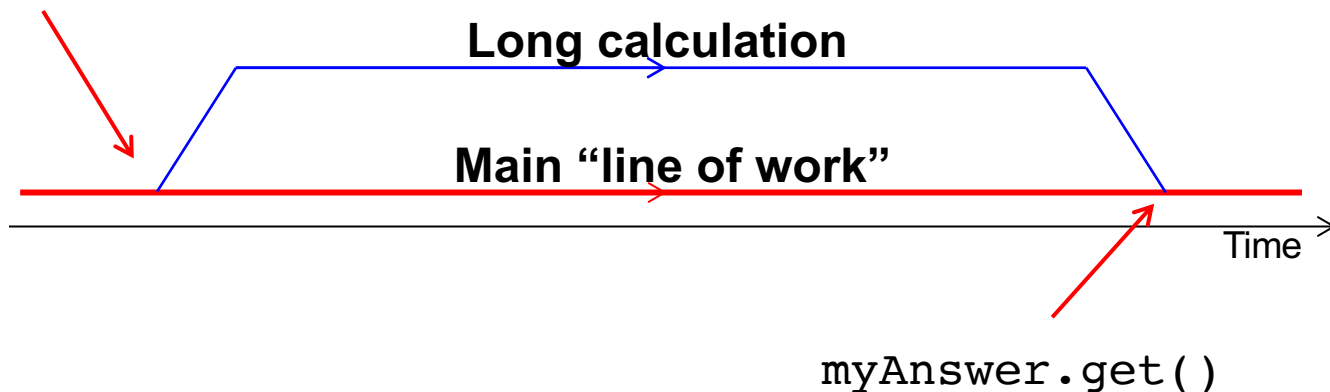
“Launch” the calculation

Retrieve the result

- `std::async` can have a second parameter, the “policy”:
 - `std::launch::async`: execute function in a new separate thread
 - `std::launch::deferred`: defer call until `get()` is called (lazy)
 - Default: “async or deferred”, the implementation chooses!

std::async in Action

```
std::future<int> myAnswer = std::async(lengthyCalculation);
```

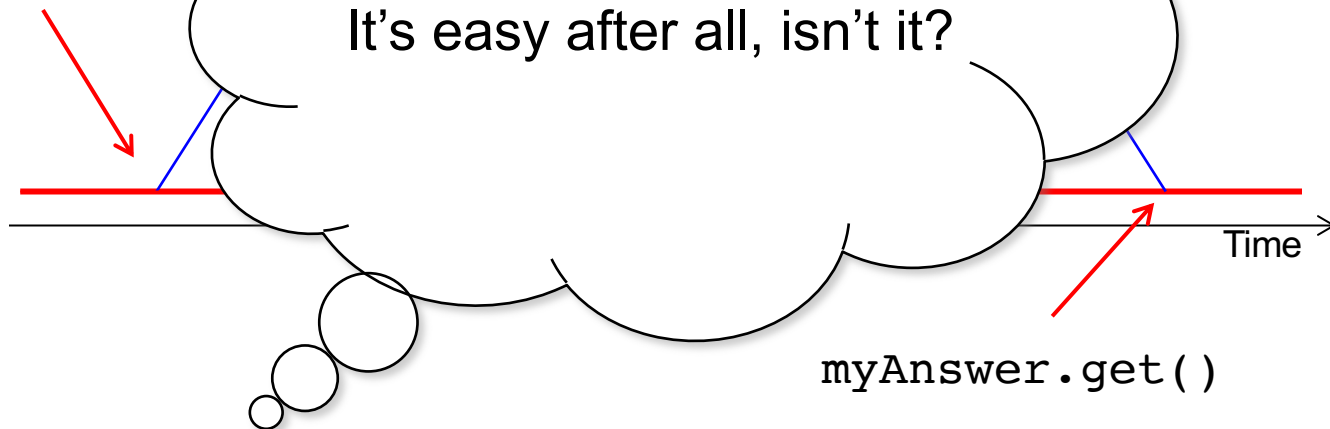


std::async in Action

```
std::future<int>
```

```
calculation);
```

It's easy after all, isn't it?



Well, to be Honest

- Scientifically relevant / potentially lucrative **real life use cases are complex**
 - Cannot be solved simply throwing threads at them ☺
- In addition, many existing high-quality non-parallel large software systems are in production
 - **Starting fresh may not be always possible**
- Example: software stack of an LHC experiment
 - Tens of (large) packages integrated
 - $O(10^2)$ shared libraries
 - Experiment specific code
 - → Millions of nicely working lines of code

Unity of opposites ☺

Need to think parallel

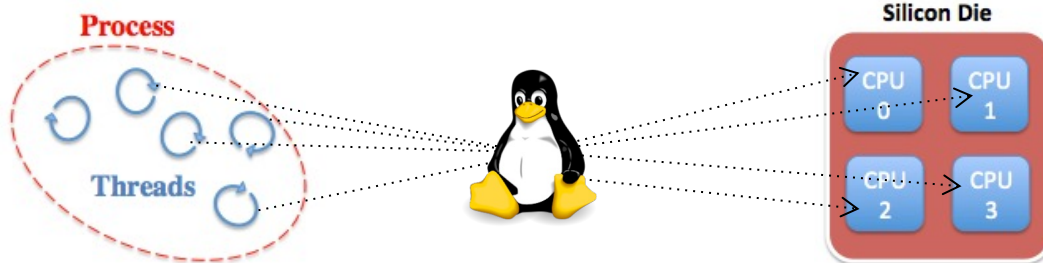
- Evolve the existing systems
- Be disruptive and think to the future

Threads



Let's switch gears: Threads

- From the operating system point of view:
 - Process**: isolated instance of a program, with its own space in (virtual) memory, can have multiple threads
 - Thread**: light-weight process within a process, **sharing the memory** with the other threads living in the same process
- The kernel manages the existing threads, scheduling them to the available resources (CPUs)*
 - There can be more threads in a single process than cores in the machine!

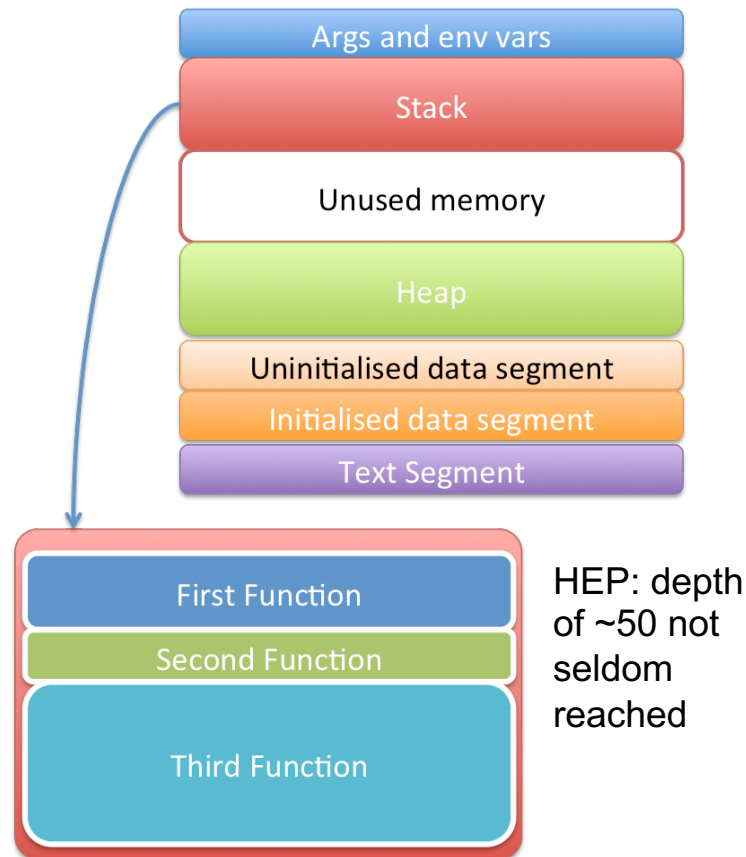


* Actually mapping user threads to kernel threads, but this simplification ok in first order!



Interlude: A Program in Memory

- **Text Segment:** code to be executed.
- **Initialized Data Segment:** global variables initialized by the programmer.
- **Uninitialized Data Segment:** This segment contains uninitialized global variables.
- **The stack:** The stack is a collection of stack frames. It grows whenever a new function is called. “Thread private”.
- **The heap:** Dynamic memory (e.g., requested with “new”).





Interlude: A Program in Memory

- **Text Segment:** code to be executed.
- **Initialized Data Segment:** global variables initialized by the programmer.
- **Uninitialized Data Segment:** global variables.
- **The stack:** stack frames, function is called.
- **The heap:** memory requested w/ malloc.

Example of allocations:

- On the stack:

```
int a=12;
myClass myObject;
```

- On the heap:

```
int* a_pointer = new int;
auto myObjectPtr = new myClass();
```

Args and env vars

Stack

Second Function

Third Function

P: depth
-50 not
seldom
reached

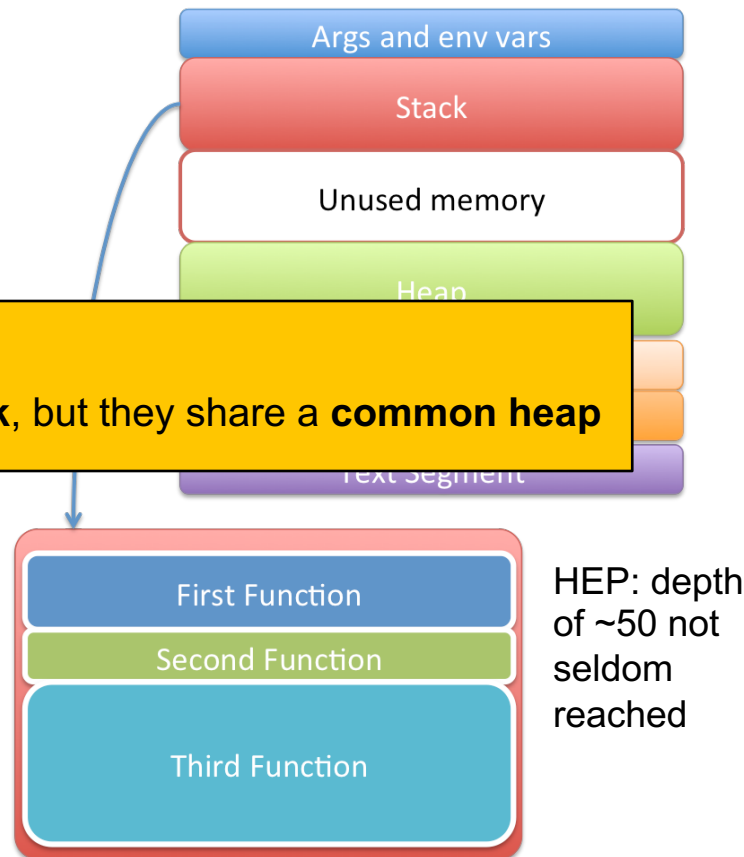


Interlude: A Program in Memory

- **Text Segment:** code to be executed.
- **Initialized Data Segment:** global variables initialized by the programmer.
- **Uninitialized Data Segment:** This segment contains uninitialized global variables.
- **The stack:** The stack grows downwards. Each time a new function is called, “Thread private”.
- **The heap:** Dynamic memory (e.g., requested with “new”).

Terminology:

Threads have their **own stack**, but they share a **common heap**



Processes and Threads: price tags

Process:

- ⊕ Isolated (different address spaces)
- ⊕ Easy to manage
- ⊖ Communication between them possible but pricey
- ⊖ Price to switch among them

Threads:

- ⊕ ⊖ Sharing memory (communication is a memory access)
- ⊕ Lower overhead for creation, lower coding effort
- ⊕ ⊕ Fit well many-cores architectures
- ⊕ ⊕ Ideal for a task-based programming model



Threads in C++

- C++ offers a construct to represent a thread: `std::thread`
- Interfaced to the underlying backend provided by the OS – 100% portable:
 - Linux: pthreads
 - Windows: Windows threads
 - ...
- A function (a *callable* in general) can be executed within a thread asynchronously
- Many more possibilities than the simple `std::async` execution
 - Full control on the thread!



Threads example

```
#include <thread> // header for std::thread
#include <iostream>

void f() { std::cout << "Hello Concurrent World!\n"; }

int main() {
    std::thread t(f);
    t.join();
}
```

Threads example

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

void f() { std::cout << "Hello Concurrent World!\n"; }

int main() {
    std::thread t(f);
    t.join();
}
```

Create and start a thread

Wait for the thread to finish its job

- In general, it is possible that the thread does not need to be joined
 - A “daemon thread”: the method to use is `std::thread::detach()`
 - Once detached, the thread cannot be joined anymore!
 - Possible use cases: I/O, monitor filesystems, clean caches...

A Pitfall with Threads



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

void f(const std::string& s) { std::cout << s; }

void g() {
    std::string s("Hello\n");
    std::thread t(f,s);
    t.detach();
}
```

A Pitfall with Threads



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

Passed by
reference

```
void f(const std::string& s) { std::cout << s; }
```

```
void g() {
    std::string s("Hello\n");
    std::thread t(f,s);
    t.detach();
}
```

String *s* lives in the
scope of function *g*

Parallel programs: variables' lifetime even more important than in sequential world

Typical behavior of the example above:

- Function ***g*** terminates before ***f***: ***s*** is a dangling reference!
- Corruption and seg-faults are guaranteed

A Pitfall with Threads



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

void f(const std::string s) { std::cout << s; }

void g() {
    std::string s("Hello\n");
    std::thread t(f,s);
    t.detach();
}
```

Passed by value

Always carefully consider ownership!

- A possible solution: create a string object and pass it by value
- But it's a copy of a string! **Yes.**
- The phase-space of design and implementation choices significantly expands when introducing concurrency!

A First Abstraction

**A possible prototype backend
behind task oriented programming!**

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

std::mutex myMutex;
void printThreadID(int i) {
    std::lock_guard<std::mutex> myLock(myMutex);
    std::cout << "thread num " << i << " - id "
               << std::this_thread::get_id() << std::endl;
};

int main() {
    std::vector<std::thread> myThreads;
    myThreads.reserve(10);
    for (int i=0; i<10; i++)
        myThreads.emplace_back(printThreadID, i);

    for (auto& t : myThreads)
        t.join();
}
```

A First Abstraction

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

**A possible prototype backend
behind task oriented programming!**

```
std::mutex myMutex;
void printThreadID(int i) {
    std::lock_guard<std::mutex> myLock(myMutex);
    std::cout << "thread num " << i << " - id "
               << std::this_thread::get_id() << std::endl;
};
```

Be patient for the moment! ☺

```
int main() {
    std::vector<std::thread> myThreads;
    myThreads.reserve(10);
    for (int i=0; i<10; i++)
        myThreads.emplace_back(printThreadID, i);

    for (auto& t : myThreads)
        t.join();
}
```

Identify the thread

Limitation: cannot retrieve
the return value.

The first step towards
automating the
management of threads in
the application!

A First Abstraction

A possible prototype backend behind task oriented programming!

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

```
std::mutex myMutex;
void printThreadID() {
    std::lock_guard<std::mutex> lg(myMutex);
    std::cout << "Thread ID: " << std::this_thread::get_id() << std::endl;
};
```

```
int main() {
    std::vector<std::thread> myThreads;
    myThreads.reserve(10);
    for (int i=0; i<10; i++) {
        myThreads.emplace_back(printThreadID);
    }
```

```
for (auto& t : myThreads)
    t.join();
}
```

```
-> g++ -std=c++14 -lpthread -o myTest myTest.cpp
-> ./myTest
thread num 0 - id 139708894000896
thread num 5 - id 139708852037376
thread num 3 - id 139708868822784
thread num 2 - id 139708877215488
thread num 4 - id 139708860430080
thread num 8 - id 139708826859264
thread num 1 - id 139708885608192
thread num 7 - id 139708835251968
thread num 6 - id 139708843644672
thread num 9 - id 139708818466560
```

When dealing with
concurrency,
asynchronous
events are daily
business!

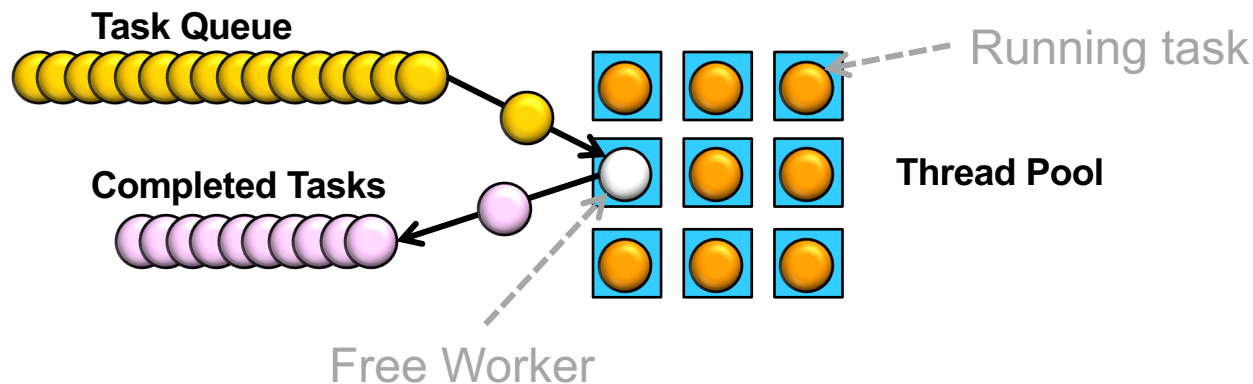
Limitation: cannot retrieve
the return value.

The first step towards
automating the
management of threads in
the application!

ment! ☺

The Thread Pool Model

- Thread pool: ensemble of worker threads which are ...
- Initialised once, consuming work from ...
- .. A work queue ...
- .. to which elements of work (tasks) can be added



Hard to program in an optimised and general way!
(usually provided by 3rd part libraries: TBB, boost, ...)

Modern Syntax: An Interlude

- A nice byproduct of the previous examples - three C++ constructs:
 - `std::vector<T>::emplace_back(T&&)`
 - `auto`
 - `for (auto& element : myCollection)`
- **emplace_back**: do not construct and then copy/move back in the vector (`push_back`) but construct *in place*. One copy less!
- **auto**: do not specify the type, the compiler finds it out at compile time. Useful to avoid tedious typing also detrimental for readability of the code!
- **Range based loops**: build a loop with a concise syntax!





A modern approach to scientific computation cannot avoid the usage of the most modern tools!

Synchronisation: Good Design, Replication, Atomics, Transactions and Locks



The Problem

- Fastest way to share data: access the same shared memory
 - One of the advantages of threads
- Parallel memory access: delicate issue - *race conditions* 
 - I.e. behaviour of the system depends on the sequence of events which are intrinsically asynchronous
- Consequences, in order of increasing severity 
 - Catastrophic terminations: seg-faults, crashes
 - Non-reproducible, intermittent bugs
 - Apparently sane execution but data corruption: e.g. wrong value of a variable or of a result

Operative definition: An entity which cannot run w/o issues linked to parallel execution is said to be thread-unsafe (the contrary is thread-safe)

To Be Precise: Data Race

Standard language rules, § 1.10/4 and /21:

- Two expression evaluations **conflict** if one of them **modifies** a memory location (1.7) and the other one accesses or **modifies** the same memory location.
- The execution of a program contains a **data race** if it contains two conflicting actions in different threads, at least one of which is **not atomic**, and **neither happens before the other**. Any such data race results in undefined behaviour.



MATT GROENING

Simple Example

Concurrency can compromise correctness

- Two threads: A and B, a variable X (44)
- A adds 10 to a variable X
- B subtracts 12 to a variable X

2 threads only
No crash
Bogus results

A then B		
Thread A	Thread B	X Val.
Read X (44)		44
Add 10	Read X (44)	44
Write X (54)	Subtract 12	54
	Write X (32)	32
RACE		

Desired		
Thread A	Thread B	X Val.
	Read X (44)	44
	Subtract 12	44
	Write X (32)	32
Read X (32)		32
Add 10		32
Write X (42)		42

B then A		
Thread A	Thread B	X Val.
	Read X (44)	44
Read X (44)	Subtract 12	44
Add 10	Write X (32)	32
Write X (54)		54
RACE		

Why so many strategies?

- Design, replication, atomics, transactions (DB like) and locks !
- ★ ■ There is **no silver bullet** to solve the issue of “resources protection”
 - Complex problematic
- **Case by case investigation** needed
 - Better to be aware of several strategies
- Best solution: **often a trade-off**
 - The lightest in the serial case?
 - The lightest in presence of high contention?



What is not Thread Safe?

Everything, unless explicitly stated!

In four words: **Shared State Among Threads**

Examples:

- Static non const variables
- STL containers
 - Some operations are thread safe, but useful to assume none is!
 - Very well documented (e.g. <http://www.cplusplus.com/reference>)
- Many random number generators (the stateful ones)
- Calls like: `strtok`, `strerror`, `asctime`, `gmtime`, `ctime` ...
- Some math libraries (statics used as cache for speed in serial execution...)
- Const casts, singletons with state: indication of unsafe policies

It sounds depressing. But there are several ways to protect thread unsafe resources!

Const Means Thread Safe

More a “new convention” rather than a technique.

- True for the STL and all at least C++11 compliant code.

“I do point out that const means immutable and absence of race conditions[...]" B. Stroustrup

(Changed) Fact of Life

C++98: “const \Rightarrow logically const”

[slightly awkward]

C++11: “const \Rightarrow thread safe
(bitwise const or internally synchronized)”

[profound change]

From H. Sutter, “You don’t know const and mutable”



Functional Programming Style

Operative definition: computation as evaluation of functions the result of which depends only on the input values and not the program state.

- Functions: **no side effects, no input modification, return new values**

3 examples of functional languages: [Haskell](#), [Erlang](#), [Lisp](#).

C++: building blocks to implement functional programming.

- STL algorithms: map an operation to a list of values.
- [Decompose operations in functions](#), percolate the information through their arguments

Even without becoming purists, functional programming principles can avoid lots of headaches typical of parallel programming



One copy of the data per Thread

- Sometimes it can be useful to have thread local variables
 - A “private heap” common to all functions executed in one thread
- Thread Local Storage (TLS)
- Replicate per thread some information
 - C++ keyword `thread_local`
- Analogies with multi-process approach but
 - Does not rely on kernel features (copy-on-write)
 - Can have high granularity
- E.g.: build “smart-thread-local pointers”
 - De-reference: provide the right content for the current thread
- Not to “one size fits them all” solution
 - Memory usage
 - Overhead of the implementation, also memory allocation strategy

Example:

`boost::thread_specific_ptr`

TLS in Action

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

Global variable, but one private copy per thread will exist

```
thread_local unsigned int t1Index = 0;
```

```
std::mutex myMutex;
void IncrAndPrint(const char* tName, unsigned int i) {
    t1Index+=i;
    std::lock_guard<std::mutex> myLock(myMutex);
    std::cout << tName << " - Thread loc. Index " << t1Index
              << std::endl;
};
```

```
int main() {
    auto t1 = std::thread(IncrAndPrint, "t1", 1);
    auto t2 = std::thread(IncrAndPrint, "t2", 2);
    IncrAndPrint("main", 0);
    t1.join(); t2.join();
}
```

Thread 1, 2 and main thread
(de facto just “threads” for the OS)

TLS in Action

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

Global variable, but one private copy per thread will exist

```
thread local unsigned int t1Index = 0;
```

Possible output:

```
main - Thread loc. Index 0
t2 - Thread loc. Index 2
t1 - Thread loc. Index 1
```

```
<< std::endl;
};

int main() {
    auto t1 = std::thread(IncrAndPrint, 1);
    auto t2 = std::thread(IncrAndPrint, 2);
    IncrAndPrint("main", 0);
    t1.join(); t2.join();
}
```

Possible output w/o tls (not correct!):

```
main - Thread loc. Index 0
t2 - Thread loc. Index 3
t1 - Thread loc. Index 3
```

Thread 1, 2 and main thread
(de facto just "threads" for the OS)



Atomic Operations

- Building block of thread safety: **an atomic operation is an operation seen as non-splittable by other threads**
 - Other real life examples: database transactions
 - Either entirely successful (subtract from A, add to B) or rolled back
- C++ offers support for atomic types
 - `#include <atomic>`
 - Usage: `std::atomic<T>`
- Operations supported natively vary according to T
 - Subtleties present: e.g. cannot instantiate `atomic<MyClass>` under all circumstances (must be *trivially copyable*)
- Well behaved with:
- boolean, integer types. E.g.** `std::atomic<unsigned long>`
 - Pointer to any type. E.g. `std::atomic<MyClass*>`

A then B		
Thread A	Thread B	X Val.
Read X (44)		44
Add 10	Read X (44)	44
Write X (54)	Subtract 12	54
	Write X (32)	32

RACE

Desired		
Thread A	Thread B	X Val.
	Read X (44)	44
	Subtract 12	44
	Write X (32)	32
Read X (32)		32
Add 10		32
Write X (42)		42

B then A		
Thread A	Thread B	X Val.
	Read X (44)	44
Read X (44)	Subtract 12	44
Add 10	Write X (32)	32
Write X (54)		54

RACE

Atomic Counter

```
#include <atomic> // and others...

std::atomic<int> gACounter;
int gCounter;

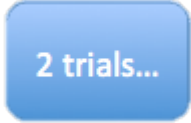
void f(){ // increment both
    gCounter++;gACounter++;}

int main() {
    std::vector<std::thread> v;
    v.reserve(10);

    for (int i=0;i<10;++i)
        v.emplace_back(std::thread(f));
    for (auto& t:v) t.join();

    std::cout << "Atomic Counter: "
               << gACounter << std::endl
               << "Counter: "
               << gCounter << std::endl;
}
```

```
$ g++ -o atomic atomic.cpp -std=c++14 -lpthread
$ ./atomic
Atomic Counter: 10
Counter: 9
$ ./atomic
Atomic Counter: 10
Counter: 10
```



3 observations:

- Atomics allow **highly granular resources protection**.
- Real life example: incorrect reference counting leads to double frees!
- **Bugs in multithreaded code** can have *extremely* subtle effects **and are in general not-reproducible!**

Food
For
Thought

The Cornerstone of Atomics

- **Compare/exchange** operation: fundamental in programming with atomics
- **At the core of implementing lock-free data structures**

```
bool std::atomic<T>::compare_exchange_strong (T& expected, T desired);
```

- Check the value of the atomic

Usable also with pointer types

- 1) If equal to `expected`, store into the atomic the value of `desired`. Return true if successful
- 2) If different from `expected`, load value of the atomic into it and return false

All of these operations are seen as a single step by all threads:
no race conditions are possible

Food
For
Thought

Compare/Exchange Example

- Problem: build cache in an object, many threads can ask the cached value
 - Example: ϕ angle between $x=0$ axis and vector initialised only with x , y and z

```
enum class cacheStates : char { kSet, kSetting, kUnset };

float myVect::phi() {
    if(cacheStates::kSet == m_phiCacheStatus.load()) return m_phi;
    float stackPhi = myMath::phi(m_x,m_y);

    auto expected = kUnset;
    if(m_phiCacheStatus.compare_exchange_strong(expected, cacheStates::kSetting)) {
        m_phi = stackPhi ;
        m_phiCacheStatus.store(cacheStates::kSet);
        return m_phi;
    }
    return stackPhi;
}
```

Food
 For
 Thought

Compare/Exchange Example

- Problem: build cache in an object, many threads can ask the cached value
 - Example: ϕ angle between $x=0$ axis and vector initialised only with x , y and z

```
enum class cacheStates : char { kSet, kSetting, kUnset };

float myVect::phi() {
    if(cacheStates::kSet == m_phiCacheStatus.load()) return m_phi;
    float stackPhi = myMath::phi(m_x,m_y);

    auto expected = kUnset;
    if(m_phiCacheStatus.compare_exchange_strong(expected, cacheStates::kSetting)) {
        m_phi = stackPhi ;
        m_phiCacheStatus.store(cacheStates::kSet);
        return m_phi;
    }
    return stackPhi;
}
```

If already calculated (ask atomically), return it!

Otherwise, calculate it

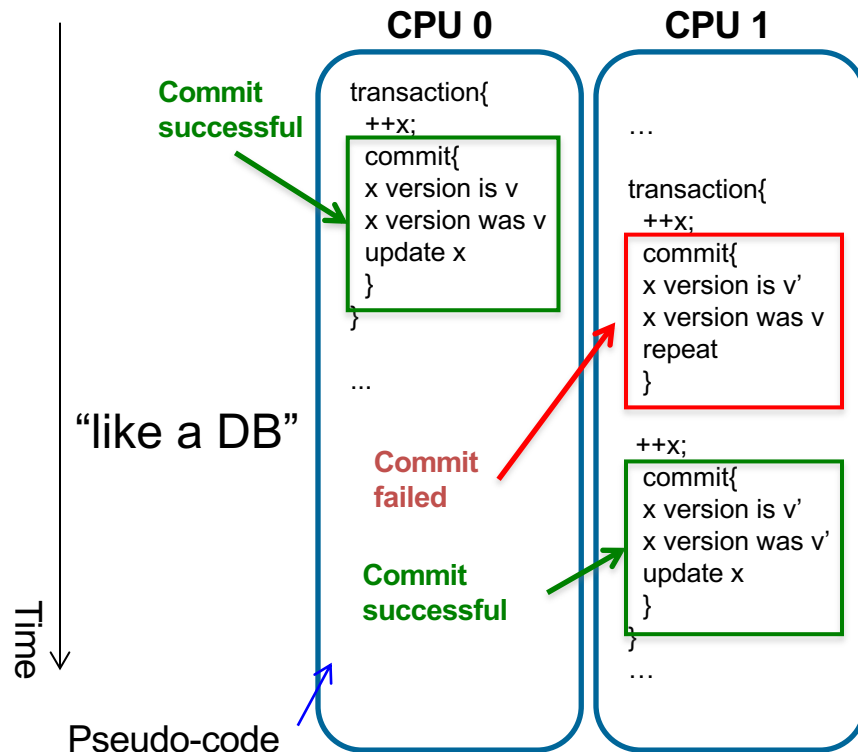
Only 1 thread will make it through this barrier!

Set the state to kSet after changing m_phi and return

Return the calculated cache: you may do the work multiple times (in presence of high contention), but you never block!

Transactional Memory (TM)

Simple example: increment variable x



Steps:

1. Check x "version" and record it
2. Increment x, do not actually *change* the value of x

❖ Is the version of x now the same of the one recorded?

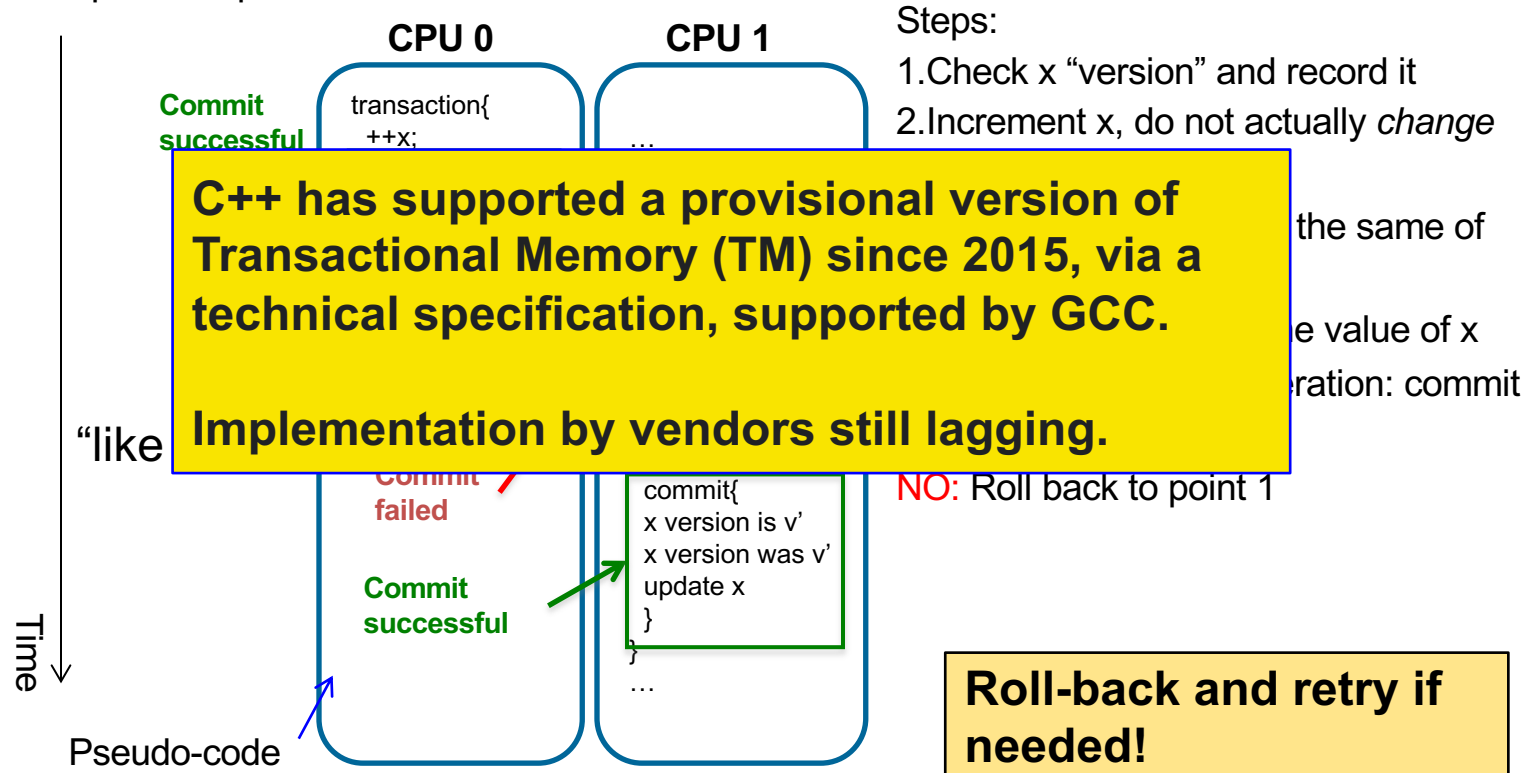
YES: No thread varied the value of x during the increment operation: commit new value

NO: Roll back to point 1

Roll-back and retry if needed!

Transactional Memory (TM)

Simple example: increment variable x





Locks and Mutexes

- Make a section of the code executable by one thread at the time
- **Use case:** detect condition (concurrently) → modify memory (once)
- **Locks should be avoided**, but yet known
 - They are a blocking synchronisation mechanisms
 - They can suffer pathologies
 - ... they could be present in existing code: use your common sense and a grain of salt!

Terminology:

- Before the section, the thread is said to *acquire a lock on a mutex*
- After that, no other thread can acquire the lock
- After the section, the thread is said to *release the lock*



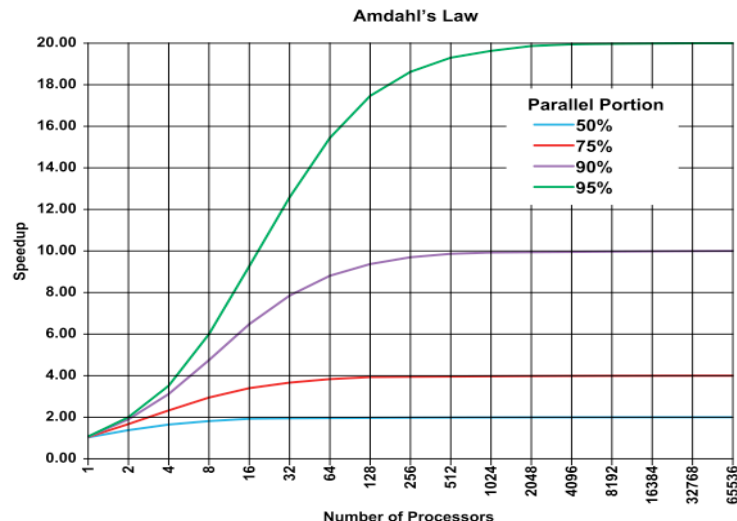
Amdahl's Law

- It predicts the **maximum speedup achievable** given a problem of **fixed size**

$$Speedup = \frac{1}{(1-p) + \frac{p}{n}}$$

n: number of cores

p: parallel portion



“... the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude.” - 1967

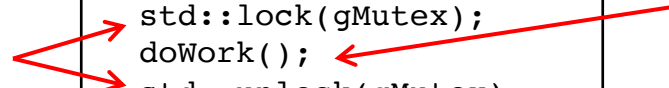


A first Lock Example

Acquire/release
lock on the
mutex

```
[...]  
std::mutex gMutex;  
void g() {  
    std::lock(gMutex);  
    doWork();  
    std::unlock(gMutex);  
}  
[...]
```

Only one thread at the
time can access this
section



A first Lock Example

Acquire/release
lock on the
mutex

```
[...]  
std::mutex gMutex;  
void g() {  
    std::lock(gMutex);  
    doWork();  
    std::unlock(gMutex);  
}  
[...]
```

Only one thread at the
time can access this
section

- Potential issue: `doWork()` **throws an exception**
- The lock is never released: the program will stall forever
- A possible solution: **a *scoped lock*** (seen in the previous slides!)

Scoped Locks: the Proper Way

Food
For
Thought

Instance of a
class, locks the
scope!

```
[...]  
std::mutex gMutex;  
void g(){  
    std::lock_guard<std::mutex> lg(gMutex);  
    doWork();  
}  
[...]
```

- Construct an object which lives in the scope to be locked
- C++ provides a class to ease this: `std::lock_guard<T>(T&)`
- When the scope is left, the object is destroyed and the lock is released
- **Application of the RAII idiom (Resource Acquisition Is Initialisation)**
 - operation lifetime = object lifetime

Pathologic Behaviours of Locks

Deadlock: Two tasks are waiting for each other to finish in order to proceed.

- One task tries to acquire a lock it already acquired and the mutex is not recursive

Convoying: A thread holding a lock is interrupted, delayed (by the OS, to do some I/O). Other threads wait that it resumes and releases the lock.

Priority inversion: A low priority thread holds a lock and makes a high priority one wait.

Lock based entities do not compose: the combination of correct components may be ill behaved.

A deadlock



Good Practices with Locks

- **Don't use them if possible**
- ... Really, don't!
- **Hold locks for the smallest amount of time possible**
- Avoid nested locks
- Avoid calling user/library code you don't control which holds locks
- Acquire locks in a fixed order

Take Away Messages

Concurrency:

- Know the internals behind a task based approach
 - Threads and shared memory
- Asynchronous execution and non-determinism permeate concurrent applications:
 - Paradigm shift needed to understand and design parallel software solutions

Synchronisation:

- Try not to be obliged to synchronise: choose the right design
- Choose atomic types and memory transactions whenever possible
 - Atomic types supported by C++
- Locks are the last resort:
 - Reduce the critical sections to the bare minimum
 - Hold locks for the smallest time possible