Lecture 4

Resource Protection and Thread Safety
This Lecture

The Goals:
1) Understand the problem of contention of resources within a parallel application
2) Become familiar with the design principles and techniques to cope with it
3) Appreciate the advantage of non-blocking techniques

The outline:
- Threads and data races: synchronisation issues
- Useful design principles
- Replication, atomics, transactions and locks
- Higher level concrete solutions
Threads and Data Races: Synchronisation Issues
The Problem

- Fastest way to share data: access the same memory region
  - 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: segfaults, 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.
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

<table>
<thead>
<tr>
<th>A then B</th>
<th>Desired</th>
<th>B then A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread A</td>
<td>Thread B</td>
<td>X Val.</td>
</tr>
<tr>
<td>Read X (44)</td>
<td>Read X (44)</td>
<td>44</td>
</tr>
<tr>
<td>Add 10</td>
<td>Read X (44)</td>
<td>44</td>
</tr>
<tr>
<td>Write X (54)</td>
<td>Subtract 12</td>
<td>54</td>
</tr>
<tr>
<td>Write X (32)</td>
<td>Add 10</td>
<td>32</td>
</tr>
<tr>
<td>Write X (42)</td>
<td>Write X (32)</td>
<td>42</td>
</tr>
</tbody>
</table>

2 Threads only
No crash
Bogus results!
What is not Thread Safe?

Everything, unless explicitly stated!

In four words: Shared States 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](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!
Useful Design Principles
Minimise Contention

- Designing and implementing software for the serial case to make it parallel afterwards
  - Not exactly a winning strategy

- Rather **think parallel right from the start**
  - Advice **not straightforward** to put in place
  - Needs careful **planning and thinking**

- Depends on the problem being studied
  - Understand what you are doing!
OUR FIELD HAS BEEN STRUGGLING WITH THIS PROBLEM FOR YEARS.

STRUGGLE NO MORE! I'M HERE TO SOLVE IT WITH ALGORITHMS!

SIX MONTHS LATER:
WOW, THIS PROBLEM IS REALLY HARD.
YOU DON'T SAY.
Ex.1: 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

Example of 3 functional languages: Haskell, Erlang, Lisp.

C++: building blocks to implement functional programming. E.g.

- Move semantics: can return entities w/o overhead
- Lambdas & algorithms: map a list of values to another list of values.
- Decompose operations in functions, percolate the information through their arguments

Without becoming purists, functional programming principles can avoid lots of headaches typical of parallel programming.
Ex. 2: No Contention a Priori

The problem:

- Task based parallel application
- Decomposition implies a chunk of work involving a thread unsafe resource (e.g. I/O on the same file)

A possible solution:

- Only one instance at the time of such task existing in the program
- No task is blocked to impose synchronisation
Ex. 3: Const Means Thread Safe

More a “new convention” rather than a technique.
- True for the STL and all 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 ⇒ logically const”
[slightly awkward]

C++11: “const ⇒ thread safe
(bitwise const or internally synchronized)”
[profound change]
Ex. 3: Const Means Thread Safe

**std::map::size**

```cpp
size_type size() const; (until C++11)
size_type size() const noexcept; (since C++11)
```

Returns the number of elements in the container, i.e. `std::distance(begin(), end())`.

**Parameters**

(none)

**Return value**

The number of elements in the container.

**Complexity**

```cpp
T& operator[]( const Key& key ); (1)
T& operator[]( Key&& key ); (2) (since C++11)
```
Replication, Atomics, Transactions and Locks
Why so many strategies?

- There is no silver bullet to solve the issue of “resource protection”
  - Complex problem

- Case by case investigation needed
  - Better to be aware of many strategies

- Best solution: often a trade-off
  - The lightest in the serial case?
  - The lightest in presence of high contention?
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`
  - Example: `boost::thread_specific_ptr`

- 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”
  - Deference: 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
  - Cannot clutter the code with `thread_local` storage specifiers
# TLS in Action

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

thread_local unsigned int tlIndex(0);

std::mutex myMutex;

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

One private copy per thread will exist

Be patient for a moment ;-) Thread 1, 2 and main thread (de facto just “threads” for the OS)
#include <thread>
#include <mutex>
#include <vector>
#include <iostream>

thread_local unsigned int tlIndex=0;

std::mutex myMutex;

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

Possible output:
main - Thread loc. Index 0  
t2 - Thread loc. Index 2  
t1 - Thread loc. Index 1  

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-splitable by other threads
  - Other real life examples: finance, 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*>`
# Atomic Counter

```cpp
#include <atomic> ...

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;
}
```

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**!

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

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

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

- Check the value of the atomic
  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`

- **Usable also with pointer types**

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

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

```cpp
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;
}
```
Compare/Exchange Example

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

Typical way of building TS caches

```cpp
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 = cachestates::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 and return

Return the calculated cache: you may do the work multiple times (in presence of high contention), but you never block!
Concurrent 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 as the one recorded?
     - YES: No thread varied the value of x during the increment operation: commit new value
     - NO: Roll back to point 1

Pseudo-code:
```
CPU 0
transaction{
  ++x;
  commit{
    x version is v
    x version was v
    update x
  }
}
...
Commit successful
```

```
CPU 1
transaction{
  ++x;
  commit{
    x version is v
    x version was v
    repeat
  }
  ++x;
  commit{
    x version is v'
    x version was v'
    update x
  }
}
...
Commit successful
```
Transactional Memory (TM)

- Software Transactional Memory (STM) already widely available
- STM can be slow – Useful tool to learn how to protect resources!
- TM supported by modern CPUs (e.g. Intel Haswell – TSX extensions)

- Concept not part of C++ (yet), but supported at least by Clang and GCC compilers. Two types of transactions:
  - __transaction_atomic: isolated, may contain only safe code (i.e. possible to roll back, no I/O, no volatile variables…)
  - __transaction_relaxed: isolated only from other transactions. Used where atomic transaction is not suited (e.g. IO, volatile, atomic memory access).

Atomic transactions can be nested into atomic and relaxed transactions, relaxed transactions only in relaxed transactions.

We will focus on atomic transactions only!
An STM Concurrent Stack

bool stmQueue::push(const float f){
    bool statusCode = false;
    __transaction_atomic {
        if (m_size<m_MaxQueueSize){
            m_v[m_size]=f;
            m_size++;
            statusCode=true;
        }
    }
    return statusCode;
}

bool stmQueue::pop(float& f){
    bool statusCode = false;
    __transaction_atomic {
        if (m_size>0){
            m_size--;
            f=m_v[m_size];
            statusCode=true;
        }
    }
    return statusCode;
}

One single, non splittable block of operations.

Transactions are a powerful way to implement synchronisation
- Code simple to understand and maintain
- Lock pathologies automatically avoided
Locks and Mutexes

- Make a section of the code executable by one thread at the time

- **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 tells us something about parallel execution:

It states the maximum speedup achievable given a certain problem of FIXED size and serial portion of the program.

\[
\Delta t = \Delta t_0 \cdot \left[ (1 - P) + \frac{P}{N} \right]
\]

\[
\text{Speedup} = \frac{\Delta t_0}{\Delta t} = \frac{1}{(1 - P) + \frac{P}{N}}
\]

N: number of cores
P: parallel portion
\(\Delta t_0\): serial exec. time

“...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 lock can be …

- **a spin lock**: if it makes a task spin while waiting (“busy wait”)
  - Short tasks: spin is better (putting a thread to sleep costs cycles)
  - Big implications also in terms of power consumption

- **Scalable**: cannot perform worse than serial execution
- **Fair**: it lets threads through in the order they arrive
- **Recursive**: it can be acquired multiple times by the same thread

Each attribute comes with a pricetag: an unfair, non-scalable, non-reentrant lock might be ideal in some situations if faster than others!

Example:
* tbb::spin_mutex class
A first Lock Example

```cpp
 [...] std::mutex gMutex;
 void g(){
   std::lock(gMutex);
   doWork();
   std::unlock(gMutex);
 }
 [...] Only one thread at the time can access this section

Acquire/release lock on the mutex
```
A first Lock Example

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

- 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

Instance of a class, locks the scope!

```cpp
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 destroyed and the lock released
- **Application of the RAII idiom (Resource Acquisition Is Initialisation)**
  - RAII: “bread and butter” in modern and performant C++
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 city 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
Higher Level Solutions
Thread Safe Containers

- Safety is an attribute of the methods, of the operations they perform.
- STL containers: reasonable thread safety in reading
  - In doubt, look for const.
- Prime example of thread unsafety in writing
  - E.g. `std::vector<T>::push_back()`
- Specific usages: design your own TS container …
- … Or rely on existing ones, e.g. the ones provided by TBB
  - `tbb::concurrent_vector`
  - `tbb::concurrent_hash_map`
  - `tbb::concurrent_bounded_queue`
**tbb::concurrent_bounded_queue**

- Example of a fundamental concept: the queue
- `tbb::concurrent_queue<T>` (simplified, drop default template arg)
- Similar to the `std::queue` but
  - Supports concurrent push and pop of items

### Important methods:

- `push, pop`: push and pop items in/from the queue. Blocking.
- `try_push, try_pop`: as above, non-blocking, return true if succeeded
The Queue of Closures

- Closure: a function + its referencing environment
  - E.g: `[](() { doWork(); })` or `std::bind(MyFunc, arg1, arg2)`

- `tbb::concurrent_queue<T>`: T can be a closure
  - Queue of work items

- Example: `serialise operations` such as output on a terminal

- **No need for locking!**

Worker Threads need to print messages on screen.

Queue of "actions"

"Printer" thread

A "service thread", can be obliged to always perform the same operation
#include "tbb/concurrent_queue.h"
#include <iostream>
#include <list>
#include <thread>

using wItem=std::function<bool(void)>;
tbb::concurrent_bounded_queue<wItem> gActionQueue;

void listen(){
    wItem work;
    bool continueListening=true;
    while(continueListening){
        gActionQueue.pop(work);
        continueListening = work();
    }
}

bool print(std::string msg){
    std::cout << msg << std::endl; return true;
}

void doWork(int s){
    for (int i=0;i<10000;++i){
        gActionQueue.push(std::bind(print, "Thread "+std::to_string(s)+" msg "+std::to_string(i)));
    }
}

int main(){
    std::thread listenerThread(listen);
    std::list<std::thread> thrList;
    for (int i=0;i<100;++i)
        thrList.push_back(std::thread(doWork,i));
    for (auto& thr:thrList) thr.join();
    gActionQueue.push([]{return false;});
    listenerThread.join();
    return 0;
}
Take Away Messages

- Choose from the start a **design which helps avoiding data races**:
  - Understand your problem: no silver bullet
  - Prefer approaches w/o global states (e.g. functional)
  - From C++11: const == thread safe

- Choose **non blocking mechanisms** if possible
  - E.g. atomics and transactions
  - Locks present in existing software
  - Use a grain of salt

- Third party libraries offer sometimes good solutions
  - E.g. concurrent containers