C++ Programming II

STL - Concurrent Programming I

C++ Programming II
October 22, 2018

Prof. Dr. P. Arnold
Bern University of Applied Sciences

Agenda

- ▶ **Intro**

Agenda

- Intro
- STL Threads

Agenda

- Intro
- STL Threads
- Data Races and Mutex

Agenda

- Intro
- STL Threads
- Data Races and Mutex
- Mutexes

Agenda

- Intro
- STL Threads
- Data Races and Mutex
- Mutexes
- Locks
  - Deadlock

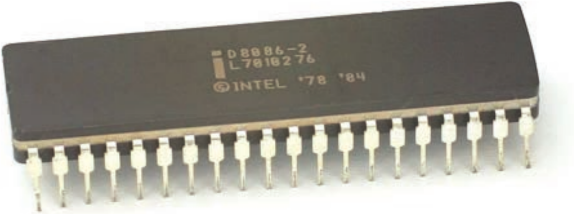
Agenda

- Intro
- STL Threads
- Data Races and Mutex
- Mutexes
- Locks
  - Deadlock
- Call Once

Intro

Intro

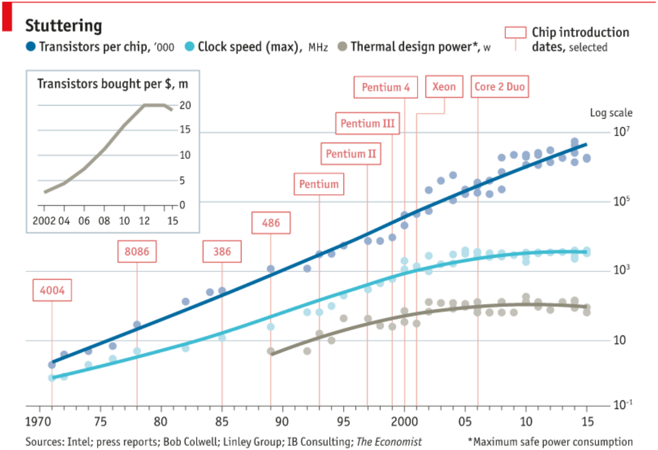
History & Moore's Law



- 1978: Intel 8086, 16-bit, 10MHz
- 2017: Intel Core i7-7700K, 64-bit, 4.2 GHz (4 cores (8 threads))

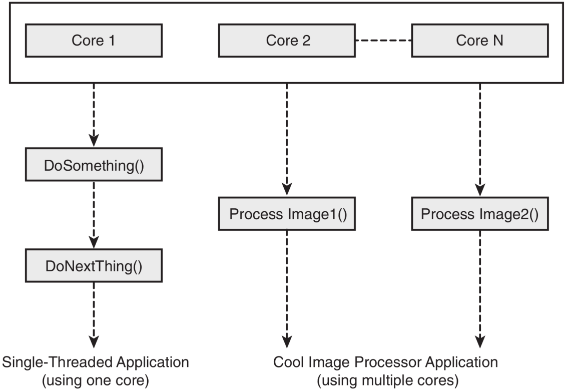
Intro

History & Moore's Law



- Moore's Law: Double clock speed each 2 years
- But clock speed stabilized around 2000 (heat dissipation)
- Number of Cores did not!

Intro
Single vs. Multi Core



- Single Core: Sequential Program Flow
- Multi Core: Parallel Program Flow
- Since C++ 11, STL provides a clean and simple way to start and stop threads without using external libraries

STL Threads

Enable Threads in CMake

`find_package` and `target_link_libraries`

- In order to use STL threads we have to:
  1. locate them with `find_package` and
  2. link the OS specific thread libraries with `target_link_libraries`

Enable Threads in CMake

`find_package` and `target_link_libraries`

- In order to use STL threads we have to:
  1. locate them with `find_package` and
  2. link the OS specific thread libraries with `target_link_libraries`
- Fortunately, `CMake` is doing that for us:

```
...
...
find_package(Threads)
...
...
...
target_link_libraries (${PROJECT_NAME} ${CMAKE_THREAD_LIBS_INIT})
```

- **Note:** Linking might not be necessary on Windows!

STL Thread

Using and Creating Threads

```
#include <iostream>

using namespace std;

void counting()
{
    for (size_t i = 0; i < 1000; ++i)
        cout << i << endl;
}

int main()
{
    counting();
    return 0;
}
```

- Single threaded program which counts to 1000

STL Thread

Using and Creating Threads

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

using namespace std;

void counting()
{
    for (size_t i = 0; i < 1000; ++i)
        cout << i << endl;
}

int main()
{
    thread t(counting);
    t.join();

    return 0;
}
```

- In the main thread we start a second thread doing the work.
- The main thread waits for the “worker-thread” to finish by calling the `join()` function! Otherwise the application might crash.
- Either the function `join()` or `detach()` have to be used.
- `join()`: blocks the main thread until the “worker-thread” is finished!
- `detach()`: detaches the “worker-thread” from the main thread, no longer external control over the thread

STL Thread

Using and Creating Threads

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

using namespace std;

void counting(size_t count)
{
    for (size_t i = 0; i < count; ++i)
        cout << i << endl;
}

int main()
{
    thread t(counting, 1000);
    t.join();

    return 0;
}
```

- We can also provide function parameters to the thread constructor

Thread Synchronisation

An Example

```cpp
#include <iostream>
#include <thread>
using namespace std;
using namespace chrono_literals;

void threadWithParam(int threadNbr)
{
    this_thread::sleep_for(1ms * threadNbr);
    cout << "Hello from thread " << threadNbr << '\n';

    this_thread::sleep_for(1s * threadNbr);
    cout << "Bye from thread " << threadNbr << '\n';
}
```

- Simple function taking an argument as thread ID to identify the thread output later
- The threads wait a different amount of time to not write to `cout` at the same time

Thread Synchronisation

An Example

```
int main()
{
    cout << thread::hardware_concurrency()
         << " concurrent threads are supported.\n";
    thread t1 {threadWithParam, 1};
    thread t2 {threadWithParam, 2};
    thread t3 {threadWithParam, 3};

    t1.join();
    t2.join();
    t3.detach();

    cout << "Threads joined.\n";
}
```

- In the main function we can print how many threads can be run at the same time using `hardware_concurrency()`
- We start 3 threads with different thread ID. `Thread t{f, x}` leads to a call of `f(x)`
- Since these threads are freely running, we need to stop them again when they are done with their work (`join()`, `detach()`)

Thread Synchronisation

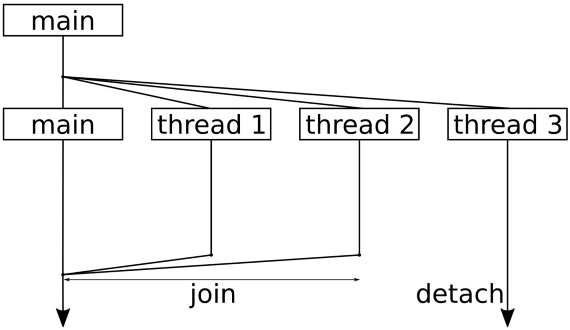
An Example

- The output of the program can look like this:

```
8 concurrent threads are supported.
Hello from thread 1
Hello from thread 2
Hello from thread 3
Bye from thread 1
Bye from thread 2
Threads joined.
```

Thread Synchronisation

`join()` and `detach()`



- Reading the diagram from top to the bottom, it shows one point in time where we split the program workflow to four threads in total
- After starting the threads, the main thread executing the main function remained without work
- The main thread waits for thread 1 and thread 2, but not for detached thread 3

Thread Parameters

Passing Value by Reference using `std::ref`

- By default, parameters are passed by copy
- To pass by reference use `std::ref`

```
#include <iostream>
#include <thread>
using namespace std;

void addToVal(int& val, int addent)
{
    val +=addent;
}

int main()
{
    int val = 23;
    // Pass paramter by refernce
    thread t{addToVal, std::ref(val), 19};
    // Main and child thread share memory!

    if(t.joinable())
        t.join();

    cout << " Result is: " << val << endl; // 42
    return 0;
}
```

Data Races and Mutex

Data Race Example

Two Threads try to set the same variable x

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

using namespace std;

void setX(int& x, int value)
{
    x = value;
}

int main()
{
    int x = 1;
    thread setX1{setX, ref(x), 100};
    thread setX2{setX, ref(x), 200};

    setX1.join();
    setX2.join();

    cout << x << " ";
    return 0;
}
```

- What is the output?
- Demo...

Data Race Example

Two Threads try to set the same variable `x`

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

using namespace std;

void setX(int& x, int value)
{
    x = value;
}

int main()
{
    int x = 1;
    thread setX1{setX, ref(x), 100};
    thread setX2{setX, ref(x), 200};

    setX1.join();
    setX2.join();

    cout << x << " ";
    return 0;
}
```

- What is the output?
- Demo...
- The two threads are racing for the same resource

Data Race Example 2

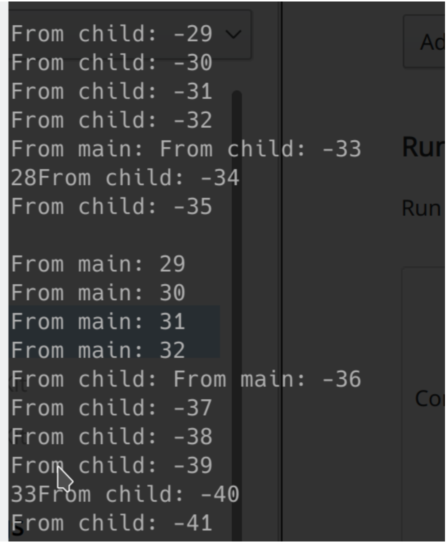
Two Threads race for the common resource `cout`

```cpp
#include <iostream>
#include <string>
#include <thread>

using namespace std;

void print()
{
    for (int i = 0; i > -100; --i)
        cout << "From child: " << i << endl;
}

int main()
{
    thread t{print};

    for (int i = 0; i < 100; ++i)
        cout << "From main: " << i << endl;

    t.join();

    return 0;
}
```

- The output is cluttered!
- We have to synchronise the access of the common resource `cout`!

Data Race Example 2

Two Threads race for the common resource `cout`



Data Race Example 2

Synchronise the access of the common resource with `std::mutex`

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

using namespace std;

mutex mu;

void sharedPrint(string msg, int value)
{
    mu.lock(); // only one thread can enter!
    cout << msg << value << endl;
    mu.unlock();
}
```

- The term mutex stands for **mut**ual **ex**clusion
- We can restrict access to one thread using `std::mutex`
- `lock` locks the mutex
- `unlock` frees the mutex

Data Race Example 2

Synchronise the access of the common resource with `std::mutex`

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

using namespace std;

mutex mu;

void sharedPrint(string msg, int value)
{
    mu.lock(); // only one thread can enter!
    cout << msg << value << endl;
    mu.unlock();
}

void print()
{
    for (int i = 0; i > -100; --i)
        sharedPrint("From child: " , i);
}

int main()
{
    thread t{print};

    for (int i = 0; i < 100; ++i)
        sharedPrint("From main: ", i);

    t.join();

    return 0;
}
```

- Do you see a problem? What happens if the code between the lock fails, *i.e.* throws an exception
- Mutex `mu` is locked for ever, called a **deadlock**

Mutex helper classes

Control `std::mutex` with `lock_guard` following RAII

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

using namespace std;

mutex mu;

void sharedPrint(string msg, int value)
{
    lock_guard<mutex> guard(mu); // RAII
    cout << msg << value << endl;
}
```

- **Don't** use `lock` and `unlock` directly
- Use RAII technique: `lock_guard`
- Mutex is freed automatically when going out of scope
- Risk of deadlock reduced

Mutexes

Overview Mutex Classes

Many different mutex classes in STL

Besides `mutex`, STL provides many different kinds of mutexes:

1. `mutex`:
   provides `lock`, `unlock` and non-blocking `try_lock` method

Overview Mutex Classes

Many different mutex classes in STL

Besides `mutex`, STL provides many different kinds of mutexes:

1. `mutex`:
   provides `lock`, `unlock` and non-blocking `try_lock` method
2. `timed_mutex`:
   Same as mutex, but provides *timing out* methods `try_lock_for` and `try_lock_until`

Overview Mutex Classes

Many different mutex classes in STL

Besides `mutex`, STL provides many different kinds of mutexes:

1. `mutex`:
   provides `lock`, `unlock` and non-blocking `try_lock` method
2. `timed_mutex`:
   Same as mutex, but provides *timing out* methods `try_lock_for` and `try_lock_until`
3. `recursive_mutex`:
   - Same as mutex, but mutex can be locked multiple times by the same thread without blocking.
   - It is released after the owning thread called `unlock` as often as it called `lock`.

Overview Mutex Classes

Many different mutex classes in STL

Besides `mutex`, STL provides many different kinds of mutexes:

1. `mutex`:
   provides `lock`, `unlock` and non-blocking `try_lock` method
2. `timed_mutex`:
   Same as mutex, but provides *timing out* methods `try_lock_for` and `try_lock_until`
3. `recursive_mutex`:
   - Same as mutex, but mutex can be locked multiple times by the same thread without blocking.
   - It is released after the owning thread called `unlock` as often as it called `lock`.
4. `recursive_timed_mutex`:
   Provides the features of both `timed_mutex` and `recursive_mutex`

Overview Mutex Classes

Many different mutex classes in STL

Besides `mutex`, STL provides many different kinds of mutexes:

1. `mutex`:
   provides `lock`, `unlock` and non-blocking `try_lock` method
2. `timed_mutex`:
   Same as mutex, but provides *timing out* methods `try_lock_for` and `try_lock_until`
3. `recursive_mutex`:
   - Same as mutex, but mutex can be locked multiple times by the same thread without blocking.
   - It is released after the owning thread called `unlock` as often as it called `lock`.
4. `recursive_timed_mutex`:
   Provides the features of both `timed_mutex` and `recursive_mutex`
5. `shared_mutex`:
   - It can be locked in *exclusive* mode and in *shared* mode.
   - If a thread locks it in shared mode, it is possible for other threads to lock it in shared mode, too!
   - While a lock is locked in shared mode, it is not possible to obtain exclusive ownership.
   - This is very similar to the behavior of `shared_ptr` , only that it does not manage memory, but lock ownership.

Locks

Overview RAII Lock-helpers

std::lock_guard

For memory management, we have `unique_ptr`, `shared_ptr` and `weak_ptr`. Those helpers provide very convenient ways to avoid memory leaks. Such helpers exist for mutexes, too.

Overview RAII Lock-helpers

std::lock_guard

For memory management, we have `unique_ptr`, `shared_ptr` and `weak_ptr`. Those helpers provide very convenient ways to avoid memory leaks. Such helpers exist for mutexes, too.

- C++ 11 provides two flavors of locks
    1. `std::lock_guard` for the simple use cases
    2. `std::unique_lock` for the advanced use cases

Overview RAII Lock-helpers

std::lock_guard

For memory management, we have `unique_ptr`, `shared_ptr` and `weak_ptr`. Those helpers provide very convenient ways to avoid memory leaks. Such helpers exist for mutexes, too.

- C++ 11 provides two flavors of locks
  1. `std::lock_guard` for the simple use cases
  2. `std::unique_lock` for the advanced use cases
- C++ 17 provides two more locks
  1. `std::scoped_lock` supporting multiple mutexes
  2. `std::shared_lock` locks mutex in shared mode (not discussed)

Overview RAII Lock-helpers

std::lock_guard

For memory management, we have `unique_ptr`, `shared_ptr` and `weak_ptr`. Those helpers provide very convenient ways to avoid memory leaks. Such helpers exist for mutexes, too.

- C++ 11 provides two flavors of locks
  1. `std::lock_guard` for the simple use cases
  2. `std::unique_lock` for the advanced use cases
- C++ 17 provides two more locks
  1. `std::scoped_lock` supporting multiple mutexes
  2. `std::shared_lock` locks mutex in shared mode (not discussed)

RAII Lock-helpers

`std::lock_guard`

- The simples one is `std::lock_guard`

```
{
  std::mutex m;
  std::lock_guard<std::mutex> lockGuard{m};
  sharedVariable= getVar();
}
```

- `std::lock_guard` element's constructor accepts a mutex, on which it calls lock immediately.
- The whole constructor call will block until it obtains the lock on the mutex

RAII Lock-helpers

std::unique_lock

- The more advanced is `std::unique_lock`
- Allows fine-grained locking

```
std::mutex m;
std::unique_lock<std::mutex> locker(m, std::defer_lock);
// Locker owns mutex, but m is not locked
```

RAII Lock-helpers

std::unique_lock

- The more advanced is `std::unique_lock`
- Allows fine-grained locking

```
std::mutex m;
std::unique_lock<std::mutex> locker(m, std::defer_lock);
// Locker owns mutex, but m is not locked

// Do something

locker.lock();  // Now the mutex is locked
// Work on shared resource
```

RAII Lock-helpers

std::unique_lock

- The more advanced is `std::unique_lock`
- Allows fine-grained locking

```
std::mutex m;
std::unique_lock<std::mutex> locker(m, std::defer_lock);
// Locker owns mutex, but m is not locked

// Do something

locker.lock();  // Now the mutex is locked
// Work on shared resource

locker.unlock();  // Now the mutex is unlocked again
// Do something else
```

RAII Lock-helpers

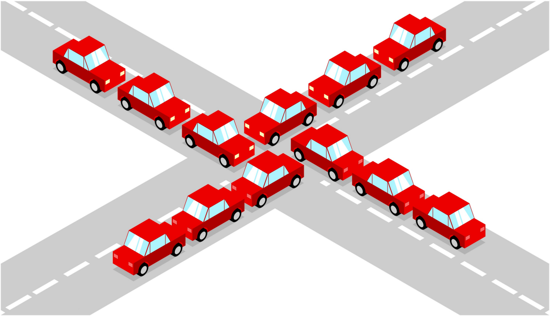
`std::unique_lock`

- The more advanced is `std::unique_lock`
- Allows fine-grained locking

```
std::mutex m;
std::unique_lock<std::mutex> locker(m, std::defer_lock);
// Locker owns mutex, but m is not locked

// Do something

locker.lock();  // Now the mutex is locked
// Work on shared resource

locker.unlock();  // Now the mutex is unlocked again
// Do something else

locker.lock();  // Now the mutex is locked again
// etc.
```

Deadlock

Avoiding deadlocks with `std::scoped_lock`



- If multiple mutexes are locked in the wrong sequence by multiple threads, a deadlock can happen
- Thread1 gets mutex1 and trys to get mutex2, Thread2 gets mutex2 and trys to get mutex1, which is locked by thread1 $\rightarrow$ deadlock

Deadlock

Avoiding deadlocks with `std::scoped_lock`

```
using namespace std;
using namespace chrono_literals;

mutex mutA;
mutex mutB;
```

Deadlock

Avoiding deadlocks with `std::scoped_lock`

```
using namespace std;
using namespace chrono_literals;

mutex mutA;
mutex mutB;

void deadlockFunc1()
{
    cout << "bad f1 acquiring mutex A..." << endl;
    lock_guard<mutex> la{mutA};
    this_thread::sleep_for(100ms);
    cout << "bad f1 acquiring mutex B..." << endl;
    lock_guard<mutex> lb{mutB};
    cout << "bad f1 got both mutexes." << endl;
}

void deadlockFunc2()
{
    cout << "bad f2 acquiring mutex B..." << endl;
    lock_guard<mutex> lb{mutB};
    this_thread::sleep_for(100ms);
    cout << "bad f2 acquiring mutex A..." << endl;
    lock_guard<mutex> la{mutA};
    cout << "bad f2 got both mutexes." << endl;
}
```

Deadlock

Avoiding deadlocks with `std::scoped_lock`

```
using namespace std;
using namespace chrono_literals;

mutex mutA;
mutex mutB;

void deadlockFunc1()
{
    cout << "bad f1 acquiring mutex A..." << endl;
    lock_guard<mutex> la{mutA};
    this_thread::sleep_for(100ms);
    cout << "bad f1 acquiring mutex B..." << endl;
    lock_guard<mutex> lb{mutB};
    cout << "bad f1 got both mutexes." << endl;
}

void deadlockFunc2()
{
    cout << "bad f2 acquiring mutex B..." << endl;
    lock_guard<mutex> lb{mutB};
    this_thread::sleep_for(100ms);
    cout << "bad f2 acquiring mutex A..." << endl;
    lock_guard<mutex> la{mutA};
    cout << "bad f2 got both mutexes." << endl;
}
```

- In main we create two threads:

```
        thread t1{deadlockFunc1};
        thread t2{deadlockFunc2};
        t1.join();
        t2.join();
```

- A typical deadlock - the program will hang forever!

Deadlock

Avoiding deadlocks with `std::scoped_lock`

- With `scoped_lock` STL provides a solution

```
void saneFunc1()
{
    scoped_lock l{mutA, mutB};
    cout << "sane f1 got both mutexes." << endl;
}
void saneFunc2()
{
    scoped_lock l{mutB, mutA};
    cout << "sane f2 got both mutexes." << endl;
}
```

```
        thread t1{saneFunc1};
        thread t2{saneFunc2};
        t1.join();
        t2.join();
```

- `scoped_lock` uses the `std::lock` function, which applies a special algorithm that performs a series of `try_lock` calls on all the mutexes provided, in order to prevent deadlocking.
- It's perfectly safe to use `scoped_lock` or call `std::lock` on the same set of locks, but in different orders.

Call Once

Call Once Flag

If a task has to be done only once

- Imagine you want to add a header to your output only once, but you don't know which thread will be first
- We introduce a variable to make sure the header is only written once

```cpp
bool hdrWritten{false};

void sharedPrint(string msg, int value)
{
    if(!hdrWritten)
    {
        std::cout <<  std::string(4, '-')
                  << " HEADER "
                  <<  std::string(4, '-') << endl;

        hdrWritten = true;
    }

    lock_guard<mutex> coutGuard(mu);
    cout << msg << value << endl;
}
```

- This is not thread safe!
- We have to put the guard above!

Call Once Flag

If a task has to be done only once

- STL provides an elegant solution for exactly this kind of issue
- `once_flag` and `call_once`

```
once_flag flag;
void sharedPrint(string msg, int value)
{
    call_once(flag,[]{std::cout <<  std::string(4, '-')
                     << " HEADER " <<  std::string(4, '-')
                     << endl;});

    lock_guard<mutex> coutGuard(mu);
    cout << msg << value << endl;
}
```

- This way the program is safe and efficient

Thank You

Questions

???