Real-Time Systems (RTSYST)

Week 3

DE HAAGSE HOGESCHOOL

C++ concurrent programmeren

- C++ heeft sinds C++11 een standaard library voor concurrent programmeren.
- Alternatieve libraries:
  - Boost Thread library http://www.boost.org/
  - Intel Threading Building Blocks (TBB) http://www.threadingbuildingblocks.org/
  - Microsoft Parallel Pattern Library (PPL) http://msdn.microsoft.com/en-us/library/dd492418.aspx
  - Open Multi-Processing (OpenMP) http://openmp.org
- Er zijn ook uitbreidingen van C++ die concurrency aan de taal toevoegen (met taalconstructies).
  - µC++ voegt o.a. taalconstructies task en monitor toe. http://plg.uwaterloo.ca/~usystem/uC++.html

Beschikbaar in MS VC++ ≥2012 en in GCC ≥4.7

Real-Time?

DE HAAGSE HOGESCHOOL 60

C++11 concurrency

- Threads
- Synchronisatie
  - Mutexen
  - Locks
  - Conditionele variabelen
  - Call once
- Asynchrone taken en Futures
- Atomics

DE HAAGSE HOGESCHOOL

C++11 Thread voorbeeld 1 (1)

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

void print1() {
    for (int i = 0; i < 10; ++i) {
        this_thread::sleep_for(chrono::milliseconds(10));
        cout << "print1" << endl;
    }
}

void print2() {
    for (int i = 0; i < 10; ++i) {
        this_thread::sleep_for(chrono::milliseconds(20));
        cout << "print2" << endl;
    }
}
```

DE HAAGSE HOGESCHOOL 62

C++11 Thread voorbeeld 1 (2)

```
int main() {
    thread t1(&print1);
    thread t2(&print2);
    t1.join();
    t2.join();
    return 0;
}
```


C++11 Thread voorbeeld 2

```
#include <thread>
// ...

void print(int delay, const string& msg) {
    for (int i = 0; i < 10; ++i) {
        this_thread::sleep_for(
                        chrono::milliseconds(delay));
        cout << msg << endl;
    }
}

int main() {
    thread t1(&print, 10, "print1");
    thread t2(&print, 20, "print2");
    t1.join();
    t2.join();
    return 0;
}
```

SAFETY

Wel type-safe (in tegenstelling tot pthreads).

```
print1
print2
print1
print1
print2
print1
print1
print2
print1
print1
print2
print1
print1
print2
print1
print2
print2
print2
print2
print2
```

C++11 Thread

- `thread::hardware_concurrency()`
  - Geeft het aantal hardware threads wat op het huidige systeem beschikbaar is (= aantal CPU's of cores of hyperthreading units), of 0 als deze informatie niet beschikbaar is.
- `t.native_handle()`
  - Geeft een instantie van `native_handle_type` wat gebruikt kan worden met de platform-specific API om de onderliggende implementatie te programmeren.
  - Kan in een real-time pthread omgeving gebruikt worden om de prioriteit in te stellen.

C++11 std: The presence of native_handle() and its semantics is implementation-defined. Actual use of this member is inherently non-portable.

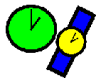
65

C++11 Synchronization

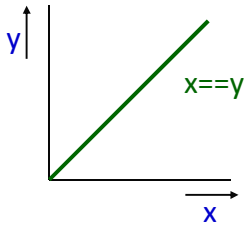
- Mutexen
  - `mutex`
    - `void lock();`
    - `bool try_lock();`
    - `void unlock();`
  - `recursive_mutex`
    - Idem maar telt aantal locks (en aantal unlocks).
- Locks
- Conditionele variabelen

DE HAAGSE HOGESCHOOL 66

C++11 Mutex voorbeeld



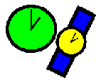
```
class Point {
public:
    Point(): x(0), y(0) {
    }
    void stepNorthEast() {
        ++x;
        ++y;
    }
    bool isNorthEast() const {
        return x == y;
    }
private:
    int x, y;
};
```



Wat gebeurt er als deze class in een multi-threaded omgeving gebruikt wordt?

DE HAAGSE HOGESCHOOL 67

C++11 Mutex voorbeeld



```
class TestPoint {
private:
    Point p;
    void thread1() {
        for (int i = 0; i < 20000000; ++i)
            p.stepNorthEast();
    }
    void thread2() {
        for (int i = 0; i < 20000000; ++i)
            if (!p.isNorthEast()) cout << "Probleem!" << endl;
    }
public:
    void test() {
        thread t1(&TestPoint::thread1, this);
        thread t2(&TestPoint::thread2, this);
        t1.join(); t2.join(); cout << "Einde." << endl;
    }
};
```

```
Probleem!
…
Probleem!
Einde.
```

DE HAAGSE HOGESCHOOL 68

C++11 Mutex voorbeeld

```
class Point {
public:
    Point(): x(0), y(0) { }
    void stepNorthEast() {
        m.lock();
        ++x; ++y;
        m.unlock();
    }
    bool isNorthEast() const {
        m.lock();
        bool res = x == y;
        m.unlock();
        return res;
    }
private:
    mutable mutex m;
    int x, y;
};
```

Einde.

	Run-time
Zonder mutex	0.265 s
Met mutex	103.482 s

Kan **aangepast** worden in **const** memberfunctie

69

C++11 Mutex voorbeeld

```
class Point {
public:
    Point(): x(0), y(0) { }
    void stepNorthEast() {
        m.lock(); ++x; ++y; m.unlock();
    }
    bool isNorthEast() const {
        m.lock(); bool res = x == y;
        m.unlock(); return res;
    }
    bool isWithinLimits() const {
        m.lock(); bool res = x >= 0 && x <= 1000000 &&
                                    y >= 0 && y <= 1000000;
        m.unlock(); return res;
    }
private:
    mutable mutex m; int x, y;
};
```

Je wilt niet onnodig wachten!

DE HAAGSE HOGESCHOOL 70

boost::shared_mutex

isNorthEast() en isWithinLimits() kunnen nu parallel draaien!

```
class Point {
public:
    Point(): x(0), y(0) { }
    void stepNorthEast() {
        m.lock(); ++x; ++y; m.unlock();
    }
    bool isNorthEast() const {
        m.lock_shared(); bool res = x == y;
        m.unlock_shared(); return res;
    }
    bool isWithinLimits() const {
        m.lock_shared(); bool res = x >= 0 && x <= 1000000 &&
                                    && y >= 0 && y <= 1000000;
        m.unlock_shared(); return res;
    }
private:
    mutable boost::shared_mutex m; int x, y;
};
```

DE HAAGSE HOGESCHOOL 71

C++11 Mutex

- C++11 kent **geen** shared_mutex
- Je kunt zelf met behulp van conditionele variabelen en mutexen een multiple readers / single-writer mutex maken.

- **Huiswerk:**
  - Implementeer zelf een *shared_mutex* class met de in C++11 beschikbare mutex en condition_variable.

DE HAAGSE HOGESCHOOL 72

C++11 Synchronization

- Mutexen
- Locks
  - `lock_guard`
    - Gebruikt RAII (Resource Acquisition Is Initialization):
      - **`lock()`** mutex in **constructor**,
      - **`unlock()`** mutex in **destructor**
  - `unique_lock`
    - Idem als `lock_guard` +
    - `lock()`, `try_lock()` en `unlock()` memberfuncties
- Conditionele variabelen

Een lock maakt het gebruik van een mutex **eenvoudiger** en **exception safe**.
Zie C++ exceptions (OGOPRG dictaat par.6.8).

DE HAAGSE HOGESCHOOL 73

C++11 Lock voorbeeld

```
class Point {
public:
    Point(): x(0), y(0) {
    }
    void stepNorthEast() {
        lock_guard<mutex> lock(m);
        ++x;
        ++y;
    }
    bool isNorthEast() const {
        lock_guard<mutex> lock(m);
        return x == y;
    }
private:
    mutable mutex m;
    int x, y;
};
```

DE HAAGSE HOGESCHOOL 74

C++11 Lock voorbeeld

```
vector<int> v;
mutex m;
//  ...
m.lock();
v.at(10) = 27;
m.unlock();
```

```
m.lock();
try {
    v.at(10) = 27;
    m.unlock();
catch(...) {
    m.unlock();
    throw;
}
```

Niet handig!

Wat gaat er **mis** als `at` een exception gooit?

Oplossing !

```
{
    lock_guard<mutex> l(m);
    v.at(10) = 27;
}
```

DE HAAGSE HOGESCHOOL 75

C++11 Synchronization

- Mutexen
- Locks
- Conditionele variabelen
  - `conditional_variable`
    - `void notify_one();`
    - `void notify_all();`
    - `void wait(unique_lock<mutex>& lock);`
    - `void wait(unique_lock<mutex>& lock, pred_type pred);`
  - `conditional_variable_any`
    - Idem maar werkt met elk type lock

DE HAAGSE HOGESCHOOL 76

C++11 Monitor voorbeeld

```
#include <thread>
#include <mutex>
#include <condition_variable>

int ei_teller = 0;
mutex m;
condition_variable c;

//  consumer thread(s)
//  ...
    unique_lock<mutex> lock(m);
    while (ei_teller < 12)
        c.wait(lock);
    ei_teller -= 12;
//  ...
```

DE HAAGSE HOGESCHOOL 77

C++ Monitor voorbeeld

```
//  producer thread(s)
//  ...
    lock_guard<mutex> lock(m);
    ei_teller += n;
    c.notify_all();
//  ...
```

Waarom moeten we `notify_all` gebruiken in plaats van `notify_one`?

Omdat er meerdere consumers kunnen zijn en die moeten **allemaal** wakker gemaakt worden!

DE HAAGSE HOGESCHOOL 78

Monitor voorbeeld

```
    while (ei_teller < 12)
        c.wait(lock);

// alternatief met predicate functie:
bool predicate() {
    return ei_teller >= 12;
}
//  ...
    c.wait(lock, &predicate);
```

DE HAAGSE HOGESCHOOL 79

Lambda functie C++11

- Een lambda functie is een anonieme functie die eenmalig gebruikt wordt als een functie object (functor).
- De lambda functie wordt gedefinieerd op de plaats waar het functie object nodig is.
- Een lambda functie kan dus als predicate gebruikt worden.

DE HAAGSE HOGESCHOOL

Monitor voorbeeld

```
    while (ei_teller < 12)
        c.wait(lock);

// alternatief met predicate functie:
bool predicate() {
    return ei_teller >= 12;
}
//  ...
    c.wait(lock, &predicate);

// alternatief met lambda functie
    c.wait(lock, []() {
        return ei_teller >= 12;
    });
```

Voordeel?	"eenvoudige" syntax
Nadeel?	geen hergebruik mogelijk

DE HAAGSE HOGESCHOOL 81

Boost Synchronization

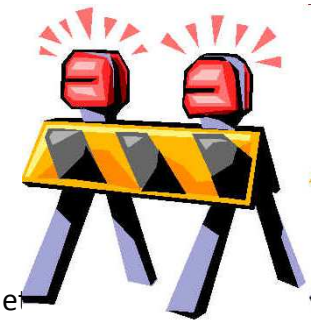
- Mutexen
- Locks
- Conditionele variabelen
- Barrier
  - `boost::barrier`
    - `barrier(unsigned count);`
    - `bool wait();`

Een barrier is een ontmoetingspunt (rendezvous). Pas nadat count threads een wait hebben gedaan mogen ze samen verder.

DE HAAGSE HOGESCHOOL 82

C++11 Synchronization

- Mutexen
- Locks
- Conditionele variabelen
- C++11 kent geen barrier
- **Huiswerk:**
  - Implementeer zelf een *barrier* class met beschikbare `mutex` en `condition_variable`.
    - `barrier(unsigned count);`
    - `bool wait();`



Een barrier is een ontmoetingspunt (rendezvous). Pas nadat count threads een wait hebben gedaan mogen ze samen verder.

83

C++11 concurrency

- Memory model
- Threads
- Synchronisatie
  - Mutexen
  - Locks
  - Conditionele variabelen
  - Call once
- **Asynchrone taken en Futures**
- Atomics

DE HAAGSE HOGESCHOOL

async en future

- Bedoeld om eenvoudig gebruik van concurrency mogelijk te maken.
- Met behulp van `async` kunnen we een functie *f* als een asynchrone taak starten. We krijgen dan een future object terug (van hetzelfde type als de functie *f*).
- We kunnen de memberfunctie get() van dit future object gebruiken om het resultaat van de (asynchrone) functie *f* op te vragen (indien nodig wordt er gewacht).
- De asynchrone taak kan in een aparte thread uitgevoerd worden (dat bepaald de implementatie / de programmeur).

DE HAAGSE HOGESCHOOL 85

async voorbeeld

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

int som(const vector<int>& v) {
    int s = 0;
    for (auto e: v) s += e;
    return s;
}

int main() {
    vector<int> v1(10000000, 1);
    vector<int> v2( 5000000, 2);
    vector<int> v3( 5000000, 4);
    cout << "som=" << som(v1) + som(v2) + som(v3) << endl;
    cin.get(); return 0;
}
```

```
som=40000000
Tijdsduur: 7.97 sec
```

som(v1), som(v2) en som(v3) kunnen **parallel** worden uitgevoerd.

Hoe vertellen we dat aan de compiler?

DE HAAGSE HOGESCHOOL 86

async voorbeeld

```
#include <future>
#include <vector>
#include <iostream>
using namespace std;

int som(const vector<int>& v) {
    /* idem */
}
int main() {
    vector<int> v1(10000000, 1);
    vector<int> v2( 5000000, 2);
    vector<int> v3( 5000000, 4);
    future<int> s1 = async(&som, ref(v1));
    future<int> s2 = async(&som, ref(v2));
    future<int> s3 = async(&som, ref(v3));
    cout << "som=" << s1.get() + s2.get() + s3.get() << endl;
    cin.get(); return 0;
}
```

De implementatie bepaald of er **aparte theads** worden gestart!

```
som=40000000
Tijdsduur: 3.99 sec
```

Waarom niet **3x** zo snel?

DE HAAGSE HOGESCHOOL 87

async voorbeeld (met auto)

```
#include <future>
#include <vector>
#include <iostream>
using namespace std;

int som(const vector<int>& v) {
    /* idem */
}
int main() {
    vector<int> v1(10000000, 1);
    vector<int> v2( 5000000, 2);
    vector<int> v3( 5000000, 4);
    auto s1 = async(&som, ref(v1));
    auto s2 = async(&som, ref(v2));
    auto s3 = async(&som, ref(v3));
    cout << "som=" << s1.get() + s2.get() + s3.get() << endl;
    cin.get(); return 0;
}
```

De implementatie bepaald of er **aparte theads** worden gestart!

DE HAAGSE HOGESCHOOL 88

async voorbeeld

```
#include <future>
#include <vector>
#include <iostream>
using namespace std;

int som(const vector<int>& v) {
    /* idem */
}
int main() {
    vector<int> v1(1000000, 1);
    vector<int> v2( 500000, 2);
    vector<int> v3( 500000, 4);
    auto s1 = async(launch::async, &som, ref(v1));
    auto s2 = async(launch::deferred, &som, ref(v2));
    auto s3 = async(launch::async, &som, ref(v3));
    cout << "som=" << s1.get() + s2.get() + s3.get() << endl;
    cin.get(); return 0;
}
```

De programmeur bepaald of er **aparte theads** worden gestart!

Start aparte thread

Start geen aparte thread

DE HAAGSE HOGESCHOOL 89

async voorbeeld (deferred)

```
#include <future>
#include <vector>
#include <iostream>
using namespace std;

int som(const vector<int>& v) {
    /* idem */
}
int main() {
    vector<int> v1(10000000, 1);
    vector<int> v2( 5000000, 2);
    vector<int> v3( 5000000, 4);
    auto s1 = async(launch::deferred, &som, ref(v1));
    auto s2 = async(launch::deferred, &som, ref(v2));
    auto s3 = async(launch::deferred, &som, ref(v3));
    cout << "som=" << s1.get() + s2.get() + s3.get() << endl;
    cin.get(); return 0;
}
```

De implementatie start **geen** aparte theads!

```
som=40000000
Tijdsduur: 7.92 sec
```

DE HAAGSE HOGESCHOOL 90

C++11 concurrency

- Memory model
- Threads
- Synchronisatie
  - Mutexen
  - Locks
  - Conditionele variabelen
  - Call once
- Asynchrone taken en Futures
- **Atomics**

DE HAAGSE HOGESCHOOL

Atomics

- Een variabele van het type `atomic<T>` kan (zonder problemen) gedeeld worden door meerdere threads.
- T kan alleen een POD (Plain Old Datatype) zijn.
- De implementatie bepaald of busy-waiting (lock-free) of locking wordt gebruikt. **Lock-free** is vaak sneller.

DE HAAGSE HOGESCHOOL 92

Mutex probleem

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

volatile int aantal = 0;

void teller() {
    for (int i = 0; i < 100000; ++i) {
        ++aantal;
    }
}
int main(void) {
    thread t1(&teller), t2(&teller), t3(&teller);
    t1.join(); t2.join(); t3.join();
    cout << "aantal = " << aantal << endl;
    return 0;
}
```

```
$ ./a.exe
aantal = 177395
$ ./a.exe
aantal = 190237
$ ./a.exe
aantal = 151421
$ time ./a.exe
aantal = 199184
    0.065 s real
```

DE HAAGSE HOGESCHOOL 93

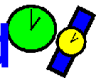
Mutex oplossing mutex

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

int aantal = 0;
mutex m;

void teller() {
    for (int i = 0; i < 100000; ++i) {
        m.lock();
        ++aantal;
        m.unlock();
    }
}
int main(void) {
    /* idem */
```

```
$ ./a.exe
aantal = 300000
$ ./a.exe
aantal = 300000
$ ./a.exe
aantal = 300000
$ time ./a.exe
aantal = 300000
    2.397 s real
```

DE HAAGSE HOGESCHOOL 94

Mutex oplossing lock_guard



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

int aantal = 0;
mutex m;

void teller() {
    for (int i = 0; i < 100000; ++i) {
        lock_guard<mutex> l(m);
        ++aantal;
    }
}
int main(void) {
    /* idem */
```

```
$ ./a.exe
aantal = 300000
$ ./a.exe
aantal = 300000
$ ./a.exe
aantal = 300000
$ time ./a.exe
aantal = 300000
    2.496 s real
```

DE HAAGSE HOGESCHOOL 95

Mutex oplossing atomic

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

atomic<int> aantal(0);

void teller() {
    for (int i = 0; i < 100000; ++i) {
        ++aantal;
    }
}
int main(void) {
    /* idem */
```

```
$ ./a.exe
aantal = 300000
$ ./a.exe
aantal = 300000
$ ./a.exe
aantal = 300000
$ time ./a.exe
aantal = 300000
    0.610 s real
```

DE HAAGSE HOGESCHOOL 96

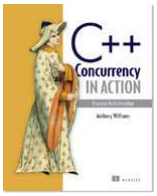
Vergelijk mutex oplossingen

Programma	Correct?	Executietijd
mutex_problem	Nee	0.065 s
mutex_solution_**mutex**	Ja	2.397 s
mutex_soution_**guard_lock**	Ja	2.496 s
mutex_solution_**atomic**	Ja	0.610 s

DE HAAGSE HOGESCHOOL 97

Meer over C++11 concurrency...

- Memory model
- Threads
- Synchronisatie
  - Mutexen
  - Locks
  - Conditionele variabelen
  - Call once
- Asynchrone taken en Futures
- Atomics



C++ Concurrency in Action
Practical Multithreading
Anthony Williams

February, 2012 | 528 pages
ISBN: 9781933988771

DE HAAGSE HOGESCHOOL