C++ Programming II

STL - Concurrent Programming III
Async & Parallel STL

C++ Programming II
November 5, 2018

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

Agenda

- **Async**

Agenda

- Async
- Future and Promise

Agenda

- Async
- Future and Promise
- Parallel STL

Agenda

- Async
- Future and Promise
- Parallel STL
- Exercise

Agenda

- Async
- Future and Promise
- Parallel STL
- Exercise
- Exam

Async

STL-Thread

Running a thread with no return value

- So far, we can easily start a thread to execute a function in an other thread

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

using namespace std;

// For threads to return values:
void factorial(int N)
{
    int res = 1;
    for (int i=N; i>1; i--)
        res *= i;

    cout << "Factorial of " << N << " is " << res << endl;
}

int main()
{
    thread t{factorial,4};
    t.join();
    return 0;
}
// Output:
```

STL-Thread

Running a thread with no return value

- So far, we can easily start a thread to execute a function in an other thread

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

using namespace std;

// For threads to return values:
void factorial(int N)
{
    int res = 1;
    for (int i=N; i>1; i--)
        res *= i;

    cout << "Factorial of " << N << " is " << res << endl;
}

int main()
{
    thread t{factorial,4};
    t.join();
    return 0;
}
// Output:
```

- But how can we get a return value from a thread? `std::ref`?

STL-Thread

Passing the return value by `std::ref`

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

using namespace std;

// Return value with shared resources
void factorial(int N, int& result)
{
    int res = 1;
    for (int i=N; i>1; i--)
        res *= i;

    result = res;
    cout << "Child - Result is: " << res << endl;
}

int main()
{
    int result{0};
    thread t{factorial, 4, ref(result)};
    t.join();
    cout << "Main - Result is: " << result << endl;
    return 0;
}
// Output:
// Child — Result is: 24
// Main — Result is: 24
```

STL-Thread

Passing the return value by `std::ref`

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

using namespace std;

// Return value with shared resources
void factorial(int N, int& result)
{
    int res = 1;
    for (int i=N; i>1; i--)
        res *= i;

    result = res;
    cout << "Child - Result is: " << res << endl;
}

int main()
{
    int result{0};
    thread t{factorial, 4, ref(result)};
    t.join();
    cout << "Main - Result is: " << result << endl;
    return 0;
}
// Output:
// Child — Result is: 24
// Main — Result is: 24
```

- Is this code safe?

STL-Thread

Passing the return value by `std::ref`

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

using namespace std;

// Return value with shared resources
void factorial(int N, int& result)
{
    int res = 1;
    for (int i=N; i>1; i--)
        res *= i;

    result = res;
    cout << "Child - Result is: " << res << endl;
}

int main()
{
    int result{0};
    thread t{factorial, 4, ref(result)};
    t.join();
    cout << "Main - Result is: " << result << endl;
    return 0;
}
// Output:
// Child — Result is: 24
// Main — Result is: 24
```

- Is this code safe?
- First, we have to protect the *shared resources* by a **mutex**

STL-Thread

Passing the return value by `std::ref`

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

using namespace std;

// Return value with shared resources
void factorial(int N, int& result)
{
    int res = 1;
    for (int i=N; i>1; i--)
        res *= i;

    result = res;
    cout << "Child - Result is: " << res << endl;
}

int main()
{
    int result{0};
    thread t{factorial, 4, ref(result)};
    t.join();
    cout << "Main - Result is: " << result << endl;
    return 0;
}
// Output:
// Child — Result is: 24
// Main — Result is: 24
```

- Is this code safe?
- First, we have to protect the *shared resources* by a **mutex**
- Second, we want to make sure, that the child thread sets the `result` first and then the parent thread continuous and fetches the variable!

STL-Thread

Passing the return value by `std::ref`

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

using namespace std;

// Return value with shared resources
void factorial(int N, int& result)
{
    int res = 1;
    for (int i=N; i>1; i--)
        res *= i;

    result = res;
    cout << "Child - Result is: " << res << endl;
}

int main()
{
    int result{0};
    thread t{factorial, 4, ref(result)};
    t.join();
    cout << "Main - Result is: " << result << endl;
    return 0;
}
// Output:
// Child — Result is: 24
// Main — Result is: 24
```

- Is this code safe?
- First, we have to protect the *shared resources* by a **mutex**
- Second, we want to make sure, that the child thread sets the `result` first and then the parent thread continuous and fetches the variable!
- We need a **condition variable**

STL-Thread

Passing the return value by `std::ref`

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

using namespace std;

// Return value with shared resources
void factorial(int N, int& result)
{
    int res = 1;
    for (int i=N; i>1; i--)
        res *= i;

    result = res;
    cout << "Child - Result is: " << res << endl;
}

int main()
{
    int result{0};
    thread t{factorial, 4, ref(result)};
    t.join();
    cout << "Main - Result is: " << result << endl;
    return 0;
}
// Output:
// Child — Result is: 24
// Main — Result is: 24
```

- Is this code safe?
- First, we have to protect the *shared resources* by a **mutex**
- Second, we want to make sure, that the child thread sets the `result` first and then the parent thread continuous and fetches the variable!
- We need a **condition variable** → **The code gets blown up**

STL-Async Function

STL provides an easy solution for that kind of job: `std::async`

```
#include <thread>
#include <future>

using namespace std;

int factorial(int N)
{
    int res = 1;
    for (int i=N; i>1; i--)
        res *= i;

    return res;
}

int main()
{
    future<int> fu = async(factorial, 4);
    // Do something else
    this_thread::sleep_for(chrono::seconds(2));
    cout << "Got from child thread: " << fu.get() << endl;
//    fu.get();  // crash!
    return 0;
}
// Output:
// Got from child thread: 24
```

STL-Async Function

STL provides an easy solution for that kind of job: `std::async`

```cpp
#include <thread>
#include <future>

using namespace std;

int factorial(int N)
{
    int res = 1;
    for (int i=N; i>1; i--)
        res *= i;

    return res;
}

int main()
{
    future<int> fu = async(factorial, 4);
    // Do something else
    this_thread::sleep_for(chrono::seconds(2));
    cout << "Got from child thread: " << fu.get() << endl;
//    fu.get();  // crash!
    return 0;
}
// Output:
// Got from child thread: 24
```

- `std::async` returns a `std::future`
- Call `get` on the future object `fu` to obtain the result
- The factorial function doesn't need a second parameter, but a return value (int).

STL-Async Launch Policy

Control Method of Execution

```
future<int> fu = async(factorial, 4);

future<int> fu = async(launch::deferred, factorial, 4);

future<int> fu = async(launch::async, factorial, 4);

future<int> fu = async(launch::async | launch::deferred,
                       factorial, 4); // Same as first line
```

STL-Async Launch Policy
Control Method of Execution

```
future<int> fu = async(factorial, 4);

future<int> fu = async(launch::deferred, factorial, 4);

future<int> fu = async(launch::async, factorial, 4);

future<int> fu = async(launch::async | launch::deferred,
                       factorial, 4); // Same as first line
```

- `std::launch::deferred`: The function is executed by the same thread, but later (lazy evaluation). Execution then happens when `get` or `wait` is called on the future. If none of both happens, the function is not called at all
- `std::launch::async`: The function is guaranteed to be executed by another thread.
- `std::launch::async | std::launch::deferred`: Default value, chooses policy automatically, depends on the system and library implementation

STL-Async

Example of blocking call

```cpp
int main()
{
    // Record start time
    auto start = std::chrono::high_resolution_clock::now();

    // Blocking call!
    async(factorial,4);
    async(factorial,5);

    // Record end time
    auto finish = std::chrono::high_resolution_clock::now();
    cout << "Elapsed time: " << (finish-start).count()*1e-9 << endl;

    return 0;
}
// Output:
// Elapsed time: 4.00082
```

STL-Async

Example of blocking call

```
int main()
{
    // Record start time
    auto start = std::chrono::high_resolution_clock::now();

    // Blocking call!
    async(factorial,4);
    async(factorial,5);

    // Record end time
    auto finish = std::chrono::high_resolution_clock::now();
    cout << "Elapsed time: " << (finish-start).count()*1e-9 <<
        endl;

    return 0;
}
// Output:
// Elapsed time: 4.00082
```

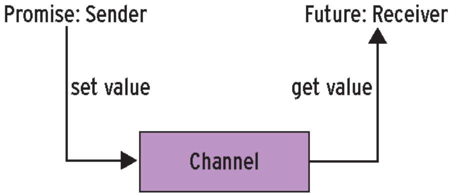
- The lifetime of the futures ends in the same line!
- This means that both the async calls from this short example are blocking
- Fix this by capturing their return values in variables with a longer lifetime

Future and Promise

Future / Promise

Channels between threads

- `std::future` and `std::promise` are a kind of communication channel between the parent and child thread where we can get the result from the child thread.
- We can get a value from the parent thread
- We can also pass a value from the parent thread in the child thread
- This can be done at some time point in the future!
- Therefore, we need a so called `std::promise`



Future / Promise

Set and get values between threads

- We create a `promise` and another `future`:

```
promise<int> p;
future<int> f = p.get_future();
```

Future / Promise

Set and get values between threads

- We create a `promise` and another `future`:

```
promise<int> p;
future<int> f = p.get_future();
```

- We pass the `future` by reference to the `async` function

```
future<int> fu = async(launch::async, factorial, ref(f));
```

Future / Promise

Set and get values between threads

- We create a `promise` and another `future`:

```
promise<int> p;
future<int> f = p.get_future();
```

- We pass the `future` by reference to the `async` function

```
future<int> fu = async(launch::async, factorial, ref(f));
```

- We set the promised value in parent thread:

```
p.set_value(4);
```

Future / Promise

Set and get values between threads

- We create a `promise` and another `future`:

```
promise<int> p;
future<int> f = p.get_future();
```

- We pass the `future` by reference to the `async` function

```
future<int> fu = async(launch::async, factorial, ref(f));
```

- We set the promised value in parent thread:

```
p.set_value(4);
```

- Finally we adapt the `factorial` function:

```
int factorial(future<int>& f)
{
    // do something else
    cout << "waiting for promised data...\n";
    this_thread::sleep_for(chrono::seconds(2));

    int N = f.get();
    cout << "Got from main thread: " << N << endl;
    int res = 1;
    for (int i=N; i>1; i--)
        res *= i;

    return res;
}
```

Parallel STL

Parallel STL

Parallelizing Code that uses Standard Algorithms

- C++ 17 came with one really major extension for parallelism: **execution policies for standard algorithms**!
- **69 algorithms** were extended to accept execution policies in order to run parallel on multiple cores, and even with enabled vectorization (SIMD).
- If we already use STL algorithms everywhere, we get a nice parallelization bonus for free.
- Simply add a single execution policy argument to our existing STL algorithm calls!

Parallel STL

Execution Policy

- `sequenced_policy`: Sequential execution form, similar to the original algorithm without an execution policy.
- `parallel_policy`: The algorithm may be executed with multiple threads.
- `parallel_unsequenced_policy`: The algorithm may be executed with multiple threads sharing the work. In addition to that, it is permissible to vectorize the code.

Parallel STL

Execution Policy

- `sequenced_policy`: Sequential execution form, similar to the original algorithm without an execution policy.
- `parallel_policy`: The algorithm may be executed with multiple threads.
- `parallel_unsequenced_policy`: The algorithm may be executed with multiple threads sharing the work. In addition to that, it is permissible to vectorize the code.

The only specific constraints are:

- All element access functions used by the parallelized algorithm must not cause *deadlocks* or data *races*
- In the case of parallelism and vectorization, all the access functions must not use any kind of blocking synchronization

Note

As long as we comply with these rules, we should be free from bugs introduced by using the parallel versions of the STL algorithms.

Parallel STL
Example

```
#include <iostream>
#include <vector>
#include <random>
#include <algorithm>

using namespace std;

bool odd(int n) { return n % 2; }

int main()
{
    vector<int> d(50000000);

    mt19937 gen;
    uniform_int_distribution<int> dis(0, 100000);
    auto randNum ([=] () mutable { return dis(gen); });
    generate(begin(d), end(d), randNum);

    sort(begin(d), end(d));
    reverse(begin(d), end(d));

    auto odds(count_if(begin(d), end(d), odd));

    cout << 100.0*odds/d.size() << "% of the numbers are odd.\n";
    // --> 50.4% of the numbers are odd.
}
```

Parallel STL

Example

```
#include <iostream>
#include <vector>
#include <random>
#include <algorithm>
#include <execution>
using namespace std;

bool odd(int n) { return n % 2; }

int main()
{
    vector<int> d(50000000);

    mt19937 gen;
    uniform_int_distribution<int> dis(0, 100000);
    auto randNum ([=] () mutable { return dis(gen); });
    generate(execution::par, begin(d), end(d), randNum);

    sort(execution::par, begin(d), end(d));
    reverse(execution::par, begin(d), end(d));

    auto odds(count_if(execution::par, begin(d), end(d), odd));

    cout << 100.0*odds/d.size() << "% of the numbers are odd.\n";
    // --> 50.4% of the numbers are odd.
}
```

- Demo

Exercise

In Class Exercise
Async

- Copy the function `calcStats` which generates `nbrElements` random samples between 1 & 1000 and returns the number of elements > 500

```
int calcStats(int nbrElelements)
{
    vector<int> d(nbrElelements);
    random_device r;
    mt19937 gen{r()};
    uniform_int_distribution<int> dis(1, nbrElelements);
    auto rand_num ([=] () mutable { return dis(gen); });
    generate(begin(d), end(d), rand_num);
    return count_if(begin(d), end(d), [&nbrElelements](int val){return val > nbrElelements/2;});
}
```

- In the `main`-function launch `calcStats` with `async` on all available thread and collect the the futures in a vector.
- In a second for-loop `get` and print the the results to `cout`.

In Class Exercise

Async

- Vary the execution policy, `nbrElements` and measure the execution time.

```cpp
int main()
{
    // Record start time
    auto start = std::chrono::high_resolution_clock::now();

    int nbrElements = 50000000;
    int nbrThreads = thread::hardware_concurrency();
```

- Your code:

```cpp
    // Record end time
    auto finish = std::chrono::high_resolution_clock::now();
    cout << "Elapsed time: " << (finish-start).count()*1e-9 << endl;
}
```

- Possible output:

```
// Run 0: 49.9905% of the numbers are larger than 25000000.
// Run 1: 49.9973% of the numbers are larger than 25000000.
// .
// .
// Run 6: 50.0071% of the numbers are larger than 25000000.
// Run 7: 50.0045% of the numbers are larger than 25000000.
// Elapsed time: 5.30622
```

Exam

Exam

Contents, Style & Material

Content:

1. STL Containers, Algorithms & Iterators
   - Lambda Functions
   - Knowing STL
   - Write faster, better and more readable code
2. STL Concurrent Programming
   - `mutex, lock & lock_guard`
   - `condition_variable`
   - `thread & asynch`
   - `future & promise`

Style:

- 90 minutes
- Hand written exam (paper & pen)
- Write code, interpret code and fix code
- Skill questions

Material:

- C++ Reference Card
- STL-Quick Reference

Thank You

Questions

???