

C++ Programming II

STL - Concurrent Programming I

C++ Programming II
October 8, 2018

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

Agenda

► Intro

Lecture 5

Prof. Dr. P. Arnold



Bern University
of Applied Sciences

Intro

STL Threads

Data Races and Mutex

Agenda

▶ Intro

▶ STL Threads

Lecture 5

Prof. Dr. P. Arnold



Intro

STL Threads

Data Races and Mutex

Agenda

▶ Intro

▶ STL Threads

▶ Data Races and Mutex



Intro

STL Threads

Data Races and Mutex

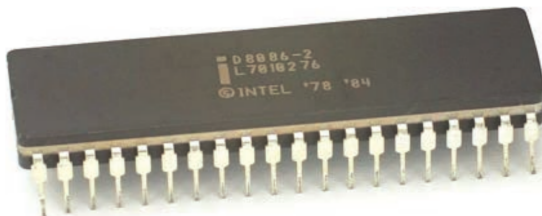


Intro

Intro

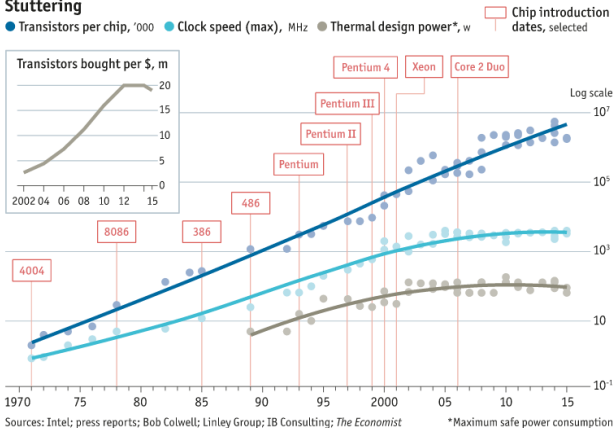
STL Threads

Data Races and Mutex

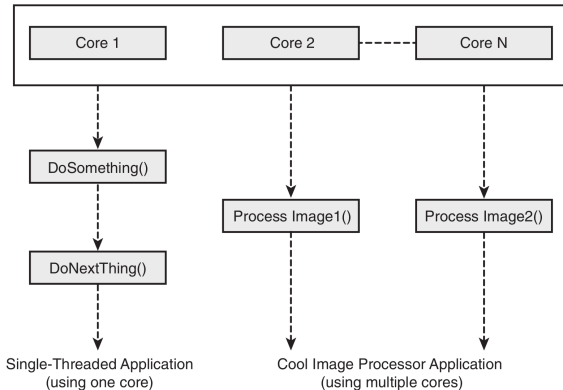


- ▶ 1978: Intel 8086, 16-bit, 10MHz
- ▶ 2017: Intel Core i7-7700K, 64-bit, 4.2 GHz (4 cores (8 threads))

Stuttering



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



- ▶ 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

Intro

STL Threads

Data Races and Mutex

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})
```

```
1  #include <iostream>
2
3  using namespace std;
4
5  void counting()
6  {
7      for (size_t i = 0; i < 1000; ++i)
8          cout << i << endl;
9  }
10
11 int main()
12 {
13     counting();
14     return 0;
15 }
```

- ▶ Single threaded program which counts to 1000



```
1 #include <iostream>
2 #include <thread>
3
4 using namespace std;
5
6 void counting()
7 {
8     for (size_t i = 0; i < 1000; ++i)
9         cout << i << endl;
10 }
11
12 int main()
13 {
14     thread t(counting);
15     t.join();
16
17     return 0;
18 }
```

- ▶ 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

```
1 #include <iostream>
2 #include <thread>
3
4 using namespace std;
5
6 void counting(size_t count)
7 {
8     for (size_t i = 0; i < count; ++i)
9         cout << i << endl;
10 }
11
12 int main()
13 {
14     thread t(counting, 1000);
15     t.join();
16
17     return 0;
18 }
```

- We can also provide function parameters to the thread constructor

Thread Synchronisation

An Example

```
1 #include <iostream>
2 #include <thread>
3 using namespace std;
4 using namespace chrono_literals;
5
6 void threadWithParam(int threadNbr)
7 {
8     this_thread::sleep_for(1ms * threadNbr);
9     cout << "Hello from thread " << threadNbr << '\n';
10
11     this_thread::sleep_for(1s * threadNbr);
12     cout << "Bye from thread " << threadNbr << '\n';
13 }
```

- ▶ 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

```
1 int main()
2 {
3     cout << thread::hardware_concurrency()
4         << " concurrent threads are supported.\n";
5     thread t1 {threadWithParam, 1};
6     thread t2 {threadWithParam, 2};
7     thread t3 {threadWithParam, 3};
8
9     t1.join();
10    t2.join();
11    t3.detach();
12
13    cout << "Threads joined.\n";
14 }
```

- ▶ 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

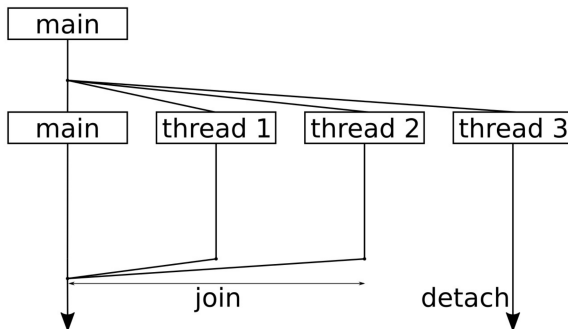
- ▶ The output of the program can look like this:

```
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 parameter by reference
    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

Intro

STL Threads

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`

```
#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

```
From child: -29
From child: -30
From child: -31
From child: -32
From main: From child: -33
28From child: -34
From child: -35

From main: 29
From main: 30
From main: 31
From main: 32
From child: From main: -36
From child: -37
From child: -38
From child: -39
33From child: -40
From child: -41
```



Data Race Example 2

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

```
1 #include <iostream>
2 #include <string>
3 #include <thread>
4 #include <mutex>
5
6 using namespace std;
7
8 mutex mu;
9
10 void sharedPrint(string msg, int value)
11 {
12     mu.lock(); // only one thread can enter!
13     cout << msg << value << endl;
14     mu.unlock();
15 }
```

- ▶ 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`

```
1  #include <iostream>
2  #include <string>
3  #include <thread>
4  #include <mutex>
5
6  using namespace std;
7
8  mutex mu;
9
10 void sharedPrint(string msg, int value)
11 {
12     mu.lock(); // only one thread can enter!
13     cout << msg << value << endl;
14     mu.unlock();
15 }
16
17 void print()
18 {
19     for (int i = 0; i > -100; --i)
20         sharedPrint("From child: ", i);
21 }
22
23 int main()
24 {
25     thread t{print};
26
27     for (int i = 0; i < 100; ++i)
28         sharedPrint("From main: ", i);
29
30     t.join();
31
32     return 0;
33 }
```

- ▶ 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

Data Race Example 2

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

```
1 #include <iostream>
2 #include <string>
3 #include <thread>
4 #include <mutex>
5
6 using namespace std;
7
8 mutex mu;
9
10 void sharedPrint(string msg, int value)
11 {
12     lock_guard<mutex> guard(mu); // RAII
13     cout << msg << value << endl;
14 }
```

- ▶ Therefore don't use `lock` and `unlock` directly
- ▶ Use RAII technique: `std::lock_guard`
- ▶ Mutex is freed automatically when going out of scope

Thank You

Questions

???

