

C++20 Thread Improvements

Lieven de Cock

www.codeblocks.org

lieven.de.cock@telenet.be

std::thread

- 2 important things

std::thread

- **JOIN**

std::thread

- **JOIN** only **ONCE**

Allocate memory → delete/free

- **Delete** the memory

Allocate memory → delete/free

- **Delete** the memory only **ONCE**

Solution

- RAII
- `std::unique_ptr`

OOPS

- No RAII foreseen in c++11 for `std::thread`
- :-(

C++20

- Throw away all your home made solutions

.....

std::jthread

jthread

- Automatically joins in its destructor
 - In case it is still joinable
- Can signal stop toward the thread callable in its destructor

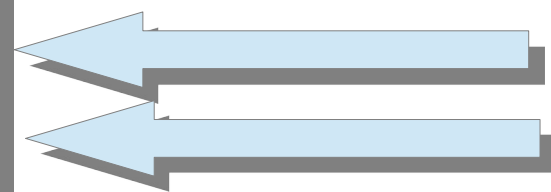
jthread

```
1  #include <iostream>
2  #include <thread>
3
4  namespace
5  {
6
7  void freeMethod()
8  {
9      std::this_thread::sleep_for(std::chrono::seconds(1));
10     std::cout << "freeMethod in the thread\n";
11     std::this_thread::sleep_for(std::chrono::seconds(1));
12 }
13
14 } // namespace
15
16
17 int main()
18 {
19     std::jthread t{freeMethod};
20
21     std::cout << "main done, waiting till (j)thread ready\n";
22     return 0;
23 }
24
```



jthread

```
5 namespace
6 {
7
8 void takingArguments(std::string& name)
9 {
10     name = "Lost";
11     std::cout << "called\n";
12     std::this_thread::sleep_for(std::chrono::seconds(2));
13 }
14
15 } // namespace
16
17 int main()
18 {
19     std::string name = "Fringe";
20     std::jthread t1{takingArguments, std::ref(name)};
21
22     t1.join(); /// we can still join if we want
23
24     std::cout << " name is now : " << name << '\n';
25
26     std::jthread t3{takingArguments, std::ref(name)};
27
28     return 0;
29 }
30
```



Start / stop

- How to stop a thread ?
- Again home made solutions
 - Atomic bool ...

C++20 : stop source + token

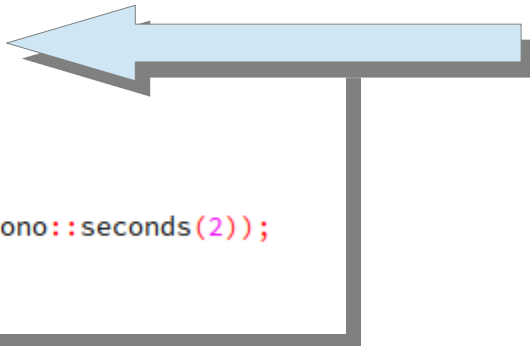
- Stop source → the thing you want to stop
 - Hands out tokens
 - Mechanism to listen to the token of the associated stop source for stop requests
- Jthread is a stop source
 - You can request a stop source from it
 - You can request a stop token from it
 - Or request it directly to stop
 - ==> shared stop state of the thread

C++20 : stop source + token

- Jthread :
 - Will request in its destructor its source to stop
- Stop token is cheap to copy

Callable entities : optional first argument => the stop token


```
14 void doSomething(std::stop_token token)
15 {
16     while(!token.stop_requested())
17     {
18         std::cout << "sleepy sleepy\n";
19         std::this_thread::sleep_for(std::chrono::seconds(2));
20     }
21 }
```



Collaborative cancellation

- If the thread function never checks if a stop request came in, all is lost
- So the thread function should check/collaborate

```
14 void doSomething(std::stop_token token)
15 {
16     while(!token.stop_requested())
17     {
18         std::cout << "sleepy sleepy\n";
19         std::this_thread::sleep_for(std::chrono::seconds(2));
20     }
21 }
```



Many threads can listen to the same token

- When creating a `jthread` you can pass in a token
- When creating an old fashioned `std::thread` you can pass in a token
- When launching `std::async` you can pass in a token

One stop token for many

```
11 void doSomething(std::stop_token stoken, int instance)
12 {
13     while(!stoken.stop_requested())
14     {
15         std::cout << instance << " sleepy sleepy\n";
16         std::this_thread::sleep_for(std::chrono::seconds(2));
17     }
18     std::cout << instance << " done\n";
19 }
```

One stop token for many

```
24 int main()
25 {
26     std::stop_source ssource;
27     auto token{ssource.get_token()};
28
29     std::jthread t1{doSomething, token, 1};
30     std::jthread t2{doSomething, token, 2};
31     std::thread t3{doSomething, token, 3};
32
33     auto fut = std::async(std::launch::async, doSomething, token, 4);
34
35
36     /// we do some stuff, and suddenly we decide the threads should stop
37     std::this_thread::sleep_for(std::chrono::seconds(5));
38     std::cout << "Main is asking the threads to stop\n";
39     ssource.request_stop();    /// we request the source to stop
40
41     std::cout << "Main still has some work to do, while the threads have already been asked to stop.\n";
42     std::this_thread::sleep_for(std::chrono::seconds(4));
43     std::cout << "Main also done\n";
44
45     t3.join();
46
47     return 0;
48 }
```

C++20 : stop callback

- Multiple stop callbacks can be registered on a stop token
 - The order of their execution is undetermined
- RAII:
 - Registers in constructor on the token (constructor argument)
 - Unregisters in destructor

```
45     std::stop_callback cb{token, [](){  
46         std::cout << "callback for MAIN \n";  
47     } };  
48
```

C++20 : stop callback

- On which thread is the callback executed
 - Stop not yet requested
 - Callback run on the thread requesting the stop
 - Stop already requested
 - Callback immediately run on the thread registering the callback

Before C++20 : stopping a condition variable

- Is blocked till the condition is fulfilled
- How to stop ?
 - Hand made, extend the condition to check on
 - The original condition
 - OR
 - A boolean variable 'stopRequested'
 - Notify the CV when putting stopRequested to true
 - CV needs to check which of the 2 subconditions are met

C++20 : stopping a condition variable

- **`std::condition_variable_any`**
- Stop token as second argument for the wait method
- Return value of wait:
 - True : the real condition was met
 - False : stop was received

C++20 : stopping a condition variable

- **std::condition_variable_any**

```
31 void popItem(std::stop_token stoken, QueueItem& item)
32 {
33     std::unique_lock lock{mMutex};
34
35     const bool ret = mCondVar.wait(lock, stoken, [&]{return !mQueue.empty();});
36     if(ret)
37     {
38         std::cout << "real notification\n";
39     }
40     else /// wait returned due to 'stop'
41     {
42         std::cout << "stop request received\n";
43         return;
44     }
```

C++20 : stopping a condition variable

- **`std::condition_variable_any`**
- Stop token as second argument for the `wait_for` method
- Return value of `wait`:
 - True : the real condition was met
 - False : stop was received or timeout occurred → check needed

C++20 : stopping a condition variable

```
50 bool popItem(std::stop_token token, QueueItem& item, uint32_t waitDurationMilliseconds)
51 {
52     std::unique_lock lock{mMutex};
53
54     if (mCondVar.wait_for(lock, token, std::chrono::milliseconds(waitDurationMilliseconds),
55         [&]{return !mQueue.empty();}) == true)
56     {
57         item = mQueue.front();
58         mQueue.pop();
59
60         return true;
61     }
62     /// due to time out or due to stop request
63     if(token.stop_requested())
64     {
65         std::cout << "due to stop request\n";
66     }
67     else
68     {
69         std::cout << "due to timeout\n";
70     }
71
72     return false;
73 }
```

On your marks go ==> Latch

- When you create threads, they start
- So they are not nicely started together
- The best approximation
 - Have those threads start of and **arrive and wait** at “the **latch**”
 - And they all continue once all threads have arrived there
- Latch : **1 time** synchronization point
- Another use case, wait till a bunch of threads are ready (**count_down** by the thread, **wait** by the other party waiting)
- Constructor argument : the number of ‘threads’
- <latch>

Latch

- Any thread can wait for it
- Any thread can countdown (by any amount, which obviously should be lower than actual count value)
- Once count reaches 0, latch is signaled and stays signaled
- Can not copy or move (assign) a barrier

Latch : silly example

- Running contest
- A runner is modeled by a thread
- We create threads ==> runner
 - The race can only start when all are at the starting blocks (this is modeled by our first latch)
- Main will wait till all runners have reached the finish line
 - Runners have different speed (modeled by sleeps)
 - Modeled by a second latch

Latch : silly example

```
22  const auto nrRunners{3};
23  std::latch atStartLine{nrRunners};
24  std::latch arrived{nrRunners};
25
26  void running(int runner)
27  {
28      std::cout << runner << " Preparing\n";
29      atStartLine.arrive_and_wait(); // all threads/runners arrive here and wait till everyone is here; and then the thread can move on
30
31      std::cout << runner << " Running\n";
32      std::this_thread::sleep_for(std::chrono::seconds(2 + 3 * runner));
33      std::cout << runner << " reached the finish line\n";
34
35      arrived.count_down();
36  }
```

Latch : silly example

```
41  int main()
42  {
43      std::vector<std::jthread> runnings;
44      for(auto runner{0}; runner < nrRunners; ++runner)
45      {
46          runnings.emplace_back(running, runner);
47          /// let's exaggerate the creation/launching part
48          std::this_thread::sleep_for(std::chrono::seconds(1));
49      }
50
51      std::this_thread::sleep_for(std::chrono::seconds(1));
52
53      std::cout << "waiting till all runners arrive at the finish line\n";
54      arrived.wait();
55
56      std::cout << "let's all hit the showers\n";
57      return 0;
58  }
```

Latch : other use case

- Say we have work we divide over threads
 - Eg generating data
- Once threads have generated their part of the data, they can continue with their (other) work
 - **count_down()**
- Some party will wait for all the threads to have delivered their data
 - **wait()**

Barrier : synchronization point → use multiple times

- `<barrier>`
- There is an (optional) completion function / callback which is called when the synchronization point is achieved (every body arrived, count became 0)
 - This is the template parameter
 - Must be **noexcept**
- Threads **arrive** and **wait**, and when all have arrived we move on
- Threads can **arrive** and **drop**, meaning they no longer take part in the next synchronization
- Completion function is run on 1 of the threads
 - The one which finally decremented the counter to 0
- When completion function has finished, counter is set back to maximum
- Can not copy or move (assign) a barrier

Barrier : silly example

- Biathlon contest (langlaufing + shooting)
- 6 contenders (threads)
 - 3 of them will not be allowed to do the shooting (constructor argument will determine that)
- We have multiple synchronization points
 - When done skiing
 - When done shooting
 - Both modeled by the single barrier

Barrier : silly example

- So all 6 have to arrive, but
 - 3 will arrive and wait (in order to shoot)
 - 3 will arrive and drop (not allowed to shoot, no longer part for the next synchronization)
 - The drop outs their thread will already run to completion
 - The 3 shooters, shoot, and synchronize again on the barrier after the shooting, and only then the threads can finish and go home

Barrier : silly example

```
29 Contender(int id, bool shootAllowed) :  
30     mId{id},  
31     mShootAllowed{shootAllowed}  
32 {}
```

```
20 const auto nrRunners{6};  
21  
22 std::barrier round2{nrRunners,  
23     [round = 0]() mutable noexcept {std::cout << " round " << ++round << " finished\n"; }  
24 };
```

```
61 int main()  
62 {  
63     std::vector<std::jthread> runnings;  
64     for(auto runner{0}; runner < nrRunners; ++runner)  
65     {  
66         runnings.emplace_back(Contender(runner, runner % 2));  
67     }  
68  
69     std::this_thread::sleep_for(std::chrono::seconds(1));  
70  
71     return 0;  
72 }
```

Barrier : silly example

```
34 void operator()()
35 {
36     std::cout << mId << " skiing \n";
37     std::this_thread::sleep_for(std::chrono::seconds(1 + mId * 1));
38     if(mShootAllowed)
39     {
40         round2.arrive_and_wait();
41     }
42     else
43     {
44         round2.arrive_and_drop();
45         return;
46     }
47     std::cout << mId << " shooting \n";
48     std::this_thread::sleep_for(std::chrono::seconds(1 + mId * 1));
49
50     round2.arrive_and_wait();
51 }
```

Barrier : another use case

- Say image rendering
- Image divided in several parts
 - Each part dealt by 1 thread
- Barrier for synchronizing
 - When each part ready ==> arrive and wait
 - When all ready ==> completion function does something
 - When that finishes, all threads go back to work to render the next image (so no thread is already trying to do some stuff and potentially create race conditions)
- So typically tasks that loop

Semaphores

- Light weight synchronization primitives
- Can be used as mutexes
- Can be used to limit availability of resources
- **std::counting_semaphore<N>** : up to a maximum amount N
- **std::binary_semaphore** : up to a maximum of 1
 - == std::counting_semaphore<1>
- Models N (available) slots, starting with an initial value between 0 and N

Semaphores

- `acquire()`
 - `try_acquire()`
- `release()`
- Both actions don't need to be done by the same thread

Semaphores : silly example

- Players (16 of them)
 - 1 player → 1 thread
- Allowed players is modeled by the counting_semaphore
 - Max 16
 - We will never allow the maximum of 16, we play with a certain current maximum amount
 - Actual players : the ones who acquired
 - Player done
 - release()
 - Another player can now acquire and join the game

Semaphores

```
15 namespace
16 {
17     const auto maxPlayers{16};
18     std::counting_semaphore<maxPlayers> sema{0}; // at 0 --> so no players allowed yet
19 }
```

Semaphores

```
20  class Player
21  {
22  public:
23      Player(int id) :
24          mId{id}
25      {}
26
27      void operator()(std::stop_token token)
28      {
29          std::osyncstream sout{std::cout};    /// <==== synchronize cout
30          while(!token.stop_requested())
31          {
32              sout << mId << " waiting to join the game \n" << std::flush_emit;
33
34              sema.acquire();
35              sout << mId << " in the game \n" << std::flush_emit;
36              std::this_thread::sleep_for(std::chrono::seconds(1));
37              sout << mId << " stepping out of the game \n" << std::flush_emit;
38              sema.release();
39          }
40          sout << mId << " stopping\n" << std::flush_emit;
41      }
42
43  private:
44      int mId;
45  };
```

Semaphores


```
50 int main()
51 {
52     std::vector<std::jthread> threads;
53     for(auto player{0}; player < maxPlayers; ++player)
54     {
55         threads.emplace_back(Player{player});
56     }
57
58     std::this_thread::sleep_for(std::chrono::seconds(3));
59
60     sema.release(2); /// 2 gamers allowed
61
62     std::this_thread::sleep_for(std::chrono::seconds(5));
63     sema.release(1); /// 1 more gamers allowed
64
65     std::this_thread::sleep_for(std::chrono::seconds(5));
66
67     std::osyncstream sout{std::cout}; /// <==== synchronize cout
68     sout << "main says, time to go home\n\n\n\n" << std::flush_emit;
69
70     return 0;
71 }
```

Observations

- No fair scheduling
 - Keep them hot and avoid context switch
- Why does it take so long to stop ?
- Let's do 2 improvements

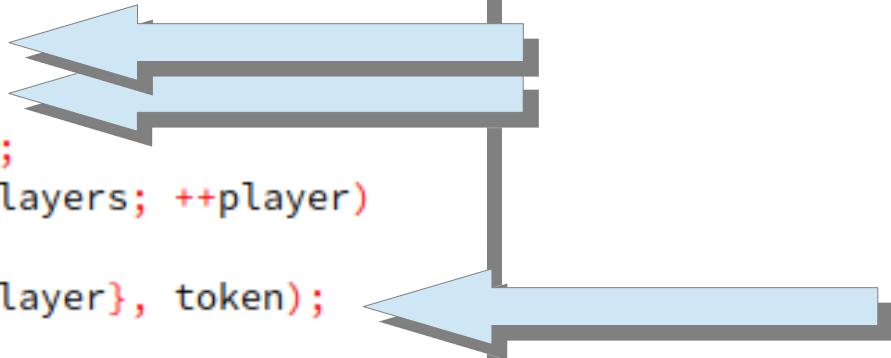
Improvement 1 : try_acquire

```
20 void operator()(std::stop_token token)
21 {
22     std::ostream sout{std::cout};    /// <==== synchronize cout
23     while(!token.stop_requested())
24     {
25         //sout << mId << " waiting to join the game \n" << std::flush_emit;
26
27         if(sema.try_acquire_for(std::chrono::milliseconds(200)))
28         {
29             sout << mId << " in the game \n" << std::flush_emit;
30             std::this_thread::sleep_for(std::chrono::seconds(1));
31             sout << mId << " stepping out of the game \n" << std::flush_emit;
32             sema.release();
33         }
34     }
35     sout << mId << " stopping\n" << std::flush_emit;
36 }
```



Improvement 2 : all threads same stop source

```
45  int main()
46  {
47      std::stop_source ssource;
48      auto token{ssource.get_token()};
49      std::vector<std::jthread> threads;
50      for(auto player{0}; player < maxPlayers; ++player)
51      {
52          threads.emplace_back(Player{player}, token);
53      }
54  }
```



QUESTIONS