Exercises week 8 - Multi-threading II

Klaas Isaac Bijlsma s2394480

David Vroom s2309939

February 1, 2018

Exercise 57

Learn to design and implement a Semaphore class

We used the following code,

semaphore/semaphore.h

```
1  #ifndef INCLUDED_SEMAPHORE_H
2  #define INCLUDED_SEMAPHORE_H
3
4  #include <mutex>
5  #include <condition_variable>
6
7  class Semaphore
8  {
9      mutable std::mutex d_mutex;
10     std::condition_variable d_condition;
11     size_t d_nAvailable;
12
13     public:
14         Semaphore(size_t nAvailable);
15         void notify();
16         void notify_all();
17         size_t size() const;
18         void wait();
19 };
20
```

```
21 inline Semaphore::Semaphore(size_t nAvailable)
22 :
23     d_nAvailable(nAvailable)
24 {}
25
26 inline size_t Semaphore::size() const
27 {
28     return d_nAvailable;
29 }
30
31 #endif
```

semaphore/semaphore.ih

```
1 #include "semaphore.h"
2
3 using namespace std;
```

semaphore/notify.cc

```
1 #include "semaphore.ih"
2
3 void Semaphore::notify()
4 {
5     lock_guard<mutex> lg(d_mutex);
6
7     if (d_nAvailable++ == 0)
8         d_condition.notify_one();
9 }
```

semaphore/notifyall.cc

```
1 #include "semaphore.ih"
2
3 void Semaphore::notify_all()
4 {
5     lock_guard<mutex> lg(d_mutex);
6
```

```
7       if (d_nAvailable++ == 0)
8           d_condition.notify_all();
9   }
```

semaphore/wait.cc

```
1   #include "semaphore.ih"
2
3   void Semaphore::wait()
4   {
5       unique_lock<mutex> ul(d_mutex);
6       while (d_nAvailable == 0)
7           d_condition.wait(ul);
8
9       --d_nAvailable;
10  }
```

Exercise 58

Become familiar with ***packaged_task***

We used the following code,

main.cc

```
1  #include <iostream>
2  #include <future>
3  #include <thread>
4  #include <iomanip>
5
6  using namespace std;
7
8  double lhs[4][5] =
9  {
10     {1, 2, 3, 4, 1},
11     {3, 4, 5, 7, 4},
12     {2, 4, 5, 9, 3},
13     {21, 8, 9, 42, 4}
14 };
15
16 double rhsT[6][5] =
17 {
18     {1, 2, 3, 4, 2},
19     {3, 4, 5, 7, 2},
20     {2, 4, 5, 90, 3},
21     {21, 8, 9, 42, 4},
22     {1, 2, 3, 4, 8},
23     {3, 4, 5, 7, 4}
24 };
25
26 enum
27 {
28     ROWS = 4,
29     COLS = 6,
30     COMMON = 5,
31 };
32
33 future<double> fut[4][6];
```

```
34
35 double innerProduct(size_t row, size_t col)
36 {
37     double sum = 0;
38     for (size_t idx = 0; idx != COMMON; ++idx)
39         sum += lhs[row][idx] * rhsT[col][idx];
40     return sum;
41 }
42
43 void computeElement(size_t row, size_t col)
44 {
45     packaged_task<double (size_t, size_t)> task(innerProduct);
46     fut[row][col] = task.get_future();
47     thread(move(task), row, col).detach();
48 }
49
50 int main()
51 {
52     for (size_t row = 0; row != ROWS; ++row)
53         for (size_t col = 0; col != COLS; ++col)
54             computeElement(row, col);
55
56     for (size_t row = 0; row != ROWS; ++row)
57     {
58         for (size_t col = 0; col != COLS; ++col)
59         {
60             try
61             {
62                 cout << setw(5) << fut[row][col].get();
63             }
64             catch (exception &msg)
65             {
66                 cout << "Exception: " << msg.what() << '\n';
67             }
68         }
69         cout << '\n';
70     }
71 }
```

Exercise 59

Become familiar with ***packaged_task (2)***

We used the following code,

main.ih

```
1  #include <iostream>
2  #include <future>
3  #include <thread>
4  #include <iomanip>
5  #include <mutex>
6  #include <queue>
7  #include "semaphore/semaphore.h"    // from ex57
8
9  using namespace std;
10
11
12 enum
13 {
14     ROWS = 4,
15     COLS = 6,
16     COMMON = 5,
17
18     NTHREADS = 8,
19     NBUSYWORKERS = 0
20 };
21
22 struct RC
23 {
24     size_t row;
25     size_t col;
26 };
27
28 typedef packaged_task<double (RC)> PTask;
29
30 extern PTask pTask[ROWS][COLS];
31 extern double lhs[4][5];
32 extern double rhsT[6][5];
33 extern queue<RC> todoQueue;
```

```
34 extern mutex queueMutex;
35 extern Semaphore producer;
36 extern Semaphore worker;
37
38
39 double innerProduct(RC rc);
40 RC getSpecs();
41 void client();
42 void produce();
```

main.cc

```
1  #include "main.ih"
2
3  double lhs[4][5] =
4  {
5      {1, 2, 3, 4, 1},
6      {3, 4, 5, 7, 4},
7      {2, 4, 5, 9, 3},
8      {21, 8, 9, 42, 4}
9  };
10
11 double rhsT[6][5] =
12 {
13     {1, 2, 3, 4, 2},
14     {3, 4, 5, 7, 2},
15     {2, 4, 5, 90, 3},
16     {21, 8, 9, 42, 4},
17     {1, 2, 3, 4, 8},
18     {3, 4, 5, 7, 4}
19 };
20
21 PTask pTask[ROWS][COLS];
22
23 queue<RC> todoQueue;
24 mutex queueMutex;
25
26 Semaphore producer(NTHREADS);
27 Semaphore worker(NBUSYWORKERS);
28
```

```
29 int main()
30 {
31     for (size_t idx = 0; idx != NTHREADS; ++idx)
32         thread(client).detach();
33 
34     produce();
35 
36     for (size_t row = 0; row != ROWS; ++row)
37     {
38         for (size_t col = 0; col != COLS; ++col)
39         {
40             try
41             {
42                 cout << setw(5) << pTask[row][col].get_future().get();
43             }
44             catch (exception const &msg)
45             {
46                 cout << "Exception: " << msg.what() << '\n';
47             }
48         }
49         cout << '\n';
50     }
51 }
```

client.cc

```
1  #include "main.ih"
2  
3  void client()
4  {
5      while (true)
6      {
7          worker.wait();
8  
9          RC rc = getSpecs();
10         if (rc.row == ROWS)
11             return;
12 
13         pTask[rc.row][rc.col](rc);
14 
```

```
15          producer.notify_all();
16      }
17  }
```

getspecs.cc

```
1   #include "main.ih"
2
3   RC getSpecs()
4   {
5       lock_guard<mutex> lg(queueMutex);
6       RC ret = todoQueue.front();
7
8       if (ret.row == ROWS)
9       {
10          worker.notify_all();
11          return ret;
12      }
13
14      todoQueue.pop();
15      return ret;
16  }
```

innerproduct.cc

```
1   #include "main.ih"
2
3   double innerProduct(RC rc)
4   {
5       double sum = 0;
6       for (size_t idx = 0; idx != COMMON; ++idx)
7           sum += lhs[rc.row][idx] * rhsT[rc.col][idx];
8       return sum;
9   }
```

produce.cc

```
1   #include "main.ih"
```

```
2
3  void produce()
4  {
5      for (size_t row = 0; row != ROWS; ++row)
6      {
7          for (size_t col = 0; col != COLS; ++col)
8          {
9              producer.wait();
10             pTask[row][col] = PTask(innerProduct);
11             {
12                 lock_guard<mutex> lg(queueMutex);
13                 todoQueue.push(RC{ row, col });
14             }
15             worker.notify_all();
16         }
17     }
18     todoQueue.push(RC{ ROWS, COLS });
19     worker.notify_all(); // notify threads to stop
20 }
```

Exercise 60

Learn to implement a multi-threaded algorithm (2)

We used the following code,

main.cc

```
1  #include <iostream>
2  #include <algorithm>
3  #include <future>
4
5  using namespace std;
6
7  void quickSort(int *beg, int *end)
8  {
9      if (end - beg <= 1)
10         return;
11
12     int lhs = *beg;
13     int *mid = partition(beg + 1, end,
14         [&](int arg)
15         {
16             return arg < lhs;
17         }
18     );
19
20     swap(*beg, *(mid - 1));
21
22     async(launch::async, quickSort, beg, mid);
23     async(launch::async, quickSort, mid, end);
24 }
25
26 int main()
27 {
28     int ia[] = {16, 2, 77, 40, 12071, 12, 3134, 42,
29                 5, 2453, 45, 3456, 35, 6, 56, 546, 2};
30
31     size_t iaSize = 17;
32
33     quickSort(ia, ia + iaSize);
```

```
34
35      for (int el: ia)
36          cout << el << '\n';
37  }
```

Exercise 62

Learn to inspect one or more ***futures*** *from inside a repeat-statement, even if the future has not yet been made ready*

For the case of one thread, we used the code below.
In the case of multiple threads, some modifications are required. We propose to store the wanted number of threads in an enum. Then define an array of this size holding `future` objects. Then in a for loop start all the threads using `async`, and store the returned `future` object in the array. Then in the eternal while loop the main task is done and at regular times an inspection, just as is done in the code below. This inspection consists of a for loop over the future-array, checking if any is ready in the same way as in the code below. If this is the case, you should break from the while loop. In order to retrieve the message, the index of the specific `future` object is needed. Therefore, the `idx` parameter of the last for loop should be defined outside this loop.

main.cc

```
1  #include <iostream>
2  #include <string>
3  #include <chrono>
4  #include <thread>
5  #include <future>
6
7  using namespace std;
8
9  string threadFun()
10 {
11     cerr << "entry\n";
12
13     this_thread::sleep_for(chrono::seconds(5));
14     cerr << "first cerr \n";
15
16     this_thread::sleep_for(chrono::seconds(5));
17     cerr << "second cerr\n";
18
19     return "done\n";
20 }
21
22 int main()
```

```
23 {
24     future<string> fut = async(launch::async, threadFun);
25 
26     size_t count = 0;
27 
28     while (true)
29     {
30         this_thread::sleep_for(chrono::seconds(1));
31         cerr << "inspecting: " << ++count << '\n';
32 
33         future_status status = fut.wait_for(0ms);
34         if (status == future_status::ready)
35             break;
36     }
37 
38     try
39     {
40         cout << fut.get();
41     }
42     catch (exception const &msg)
43     {
44         cout << msg.what() << '\n';
45     }
46 }
```