Week 8

Exercise 63

../63/semaphore/semaphore.h

```
#ifndef INCLUDED_SEMAPHORE_
#define INCLUDED_SEMAPHORE_

#include <mutex>
#include <condition_variable>

class Semaphore
{
  size_t d_nAvailable;
  std::mutex d_Mutex;
  std::condition_variable d_condition;

  public:
    Semaphore(size_t nAvailable);

    void notify();
    void notify_all();

    size_t size() const;
    void wait();

  private:
};

#endif
```

../63/semaphore/semaphore.ih

```
#include "semaphore.h"

#include "chrono"


using namespace std;
```

../63/semaphore/c_semaphore.cc

```
#include "semaphore.ih"

Semaphore::Semaphore(size_t nAvailable)
:
d_nAvailable(nAvailable)
{
}
```

../63/semaphore/notify_all.cc

```
#include "semaphore.ih"

void Semaphore::notify_all()
{
  //all waiting thread(wait member) is notified reactivating that thread
  //only one waiting thread will be able to obtain the semaphore;s lock
  //and to reduce available and that particular thread is thereupon reactivated.
  lock_guard<mutex> lk(d_Mutex);
  if (d_nAvailable++ == 0)
    d_condition.notify_all();
}
```

../63/semaphore/notify.cc

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

../63/semaphore/size.cc

```
1  #include "semaphore.ih"
2
3  size_t Semaphore::size() const
4  {
5    return d_nAvailable;
6  }
```

../63/semaphore/wait.cc

```
1  #include "semaphore.ih"
2
3  void Semaphore::wait()
4  {
5    unique_lock<mutex> lk(d_Mutex);
6
7    while(d_nAvailable == 0)
8      d_condition.wait(lk);
9
10   --d_nAvailable;
11 }
```

../63/main.ih

```
1  #define ERR(msg) printf("%s : %d", (msg), __LINE__)
2
3  #include "semaphore/semaphore.h"
4  #include <vector>
5  #include <thread>
6  #include <queue>
7  #include <iostream>
8
9  using namespace std;
10
11 void process(size_t item);
12
13 void consumer(Semaphore &filled, Semaphore &available,
14   queue<size_t> &itemQueue, bool &finished);
15 void producer(Semaphore &filled, Semaphore &available,
16   queue<size_t> &itemQueue, bool &finished);
```

../63/main.cc

```
1  #include "main.ih"
2
3  int main(int argc, char const **argv)
4  {
5    bool finished = 0;
6
7    Semaphore available(10);
8    Semaphore filled(0);
9
```

```
10    std::queue<size_t> itemQueue;
11
12    thread consume(consumer, ref(filled), ref(available), ref(itemQueue),
13                   ref(finished));
14    thread produce(producer, ref(filled), ref(available), ref(itemQueue),
15                   ref(finished));
16
17    consume.join();
18    produce.join();
19  }
```

../63/consumer.cc

```
1   #include "main.ih"
2
3   void consumer(Semaphore &filled, Semaphore &available,
4     queue<size_t> &itemQueue, bool &finished)
5   {
6     while (!finished || !itemQueue.empty())
7     {
8       filled.wait();
9       size_t item = itemQueue.front();
10      itemQueue.pop();
11      available.notify_all();
12      process(item);
13    }
14  }
```

../63/process.cc

```
1   #include "main.ih"
2
3   void process(size_t item)
4   {
5     cout << item << '\n';
6   }
```

../63/producer.cc

```
1   #include "main.ih"
2
3   void producer(Semaphore &filled, Semaphore &available,
4     queue<size_t> &itemQueue, bool &finished)
5   {
6     size_t item = 0;
7     while (item < 10)
8     {
9       ++item;
10      available.wait();
11      itemQueue.push(item);
12      filled.notify_all();
13    }
14    finished = true;
15  }
```

Exercise 64

../64–2/main.ih

```
#define ERR(msg) printf("%s : %d", (msg), __LINE__)

#include <iostream>
#include <array>
#include <numeric>

#include <future>
#include <thread>
#include <iomanip>

using namespace std;
```

../64–2/main.cc

```
#include "main.ih"

double lhs[4][5] = {
  {0, 1, 2, 3, 4},
  {5, 6, 7, 8, 9},
  {0, 1, 2, 3, 4},
  {5, 6, 7, 8, 9}
};

double rhsT[6][5] = {
  {0, 1, 2, 3, 4},
  {5, 6, 7, 8, 9},
  {0, 1, 2, 3, 4},
  {5, 6, 7, 8, 9},
  {5, 6, 7, 8, 9},
  {0, 1, 2, 3, 4}
};

void innerProduct(promise<double> &ref, int row, int col)
{
  int sum = 0;
  for (int idx = 0; idx != 5; ++idx)
    sum += lhs[row][idx] * rhsT[col][idx]; //here for one would expect
                          //rhsT[idx][col] but we instead transposed
  ref.set_value(sum);           //the matrix first thus rhsT[col][idx]
}

int main(int argc, char const **argv)
{

  promise<double> result[4][6];

  for (int row = 0; row != 4; ++row) //running threads
  {
    for (int col = 0; col != 6; ++col)
      thread(innerProduct, ref(result[row][col]), row, col).detach();
  }

  for (int row = 0; row != 4; ++row) //printing the new matrix
  {
    for (int col = 0; col != 6; ++col)
      cout << setw(3) << result[row][col].get_future().get();
    cout << '\n';
  }


}
```

Exercise 65

../65/main.ih

```
#define ERR(msg) printf("%s : %d", (msg), __LINE__)


#include <algorithm>
#include <iostream>
#include <future>
#include <chrono>

void qqsort(int *beg, int *end);
```

../65/main.cc

```
#include "main.ih"

int main(int argc, char const **argv)
try
{
  int ia[5] = {3,2,1,7,4};

  qqsort(ia, ia + sizeof(ia)/sizeof(int));

  for (auto &i: ia)
  {
    std::cout << i << '\n';
  }

}
catch(...)
{
  std::cout << "haHA caught one!\n";
}
```

../65/qqsort.cc

```
#include "main.ih"

void qqsort(int *beg, int *end)
 {
     if (end - beg <= 1)
         return;

     int lhs = *beg;
     int *mid = std::partition(beg + 1, end,
         [&](int arg)
         {
             return arg < lhs;
         }
       );

     std::swap(*beg, *(mid - 1));

     auto fut1 = async(std::launch::async, qqsort, beg, mid);
     auto fut2 = async(std::launch::async, qqsort, mid, end);

     std::this_thread::sleep_for(std::chrono::seconds(2));

     fut1.get();
     fut2.get();

 }
```

Exercise 68

First, the files that are to be compiled should be collected into a container of some sort. The assumption is, to keep it simple, that all `.cc` files in or under the current working directory should be compiled, and thereafter linked. This can be accomplished using the `fs::recursive_directory_iterator` function, which searches a folder and its subfolders for any files, returning path iterators that can then be converted to strings for convenience.
Another possibility is for the command line argument to specify the folder to compile the files therein. This argument should then be removed from the array, ensuring that any modifiers that are specified thereafter (i.e. `-pthread`) are the only elements left in the array.
Of course, only the relevant files should be kept (i.e. the `.cc` files). A simple `remove_if` algorithm solves this.
Second, the relevant files should be compiled. A queue-like structure could be implemented here, but another possibility is to simply successively create vectors of threads, sized to the number that is specified by the user (which is, again, thereafter deleted from the `argv` array), and then waiting until these threads have finished before starting the next group of threads. Specifically, `pipe` and `popen` can be used to execute a system command and catch its output. Here, it is especially important to catch any error (specifically run-time errors) that may result from using these commands. Now, there are two options. First, as the exercise specifies, any time there is any output at all, that would mean there has occurred some kind of error or warning, which should be displayed, and the loop of creating successive threads should halt. Second, and I think preferably, compilation could simply continue, outputting any kind of error that other files may exhibit, which would allow the user to correct more errors than simply the first one(s). Of course, the step thereafter, linking, is pointless if there are any errors and should be skipped entirely.
If, however, a class structure is implemented, this is how it could look:

Class	**Collector**	**Compilers**
Data	**d_Folder (string)** Specifies folder to search for .cc files	**d_vFilepathsSub (vector)** Contains strings representing a subset of .cc files
	d_vFilepaths (vector) Contains strings representing .cc files	**d_mErrors (map)** Stores the output from the compile commands
Functions	**iterateFolder** Finds the files in the specified folder	**compile** Compiles the files in d_vFilepathsSub and catches any output (esp. run-time errors, but also g++ errors)
	findCC Deletes any files from d_vFilepaths that are not .cc files	

Once a specified client (i.e. `Compiler`) is done, then it acquires a mutex and displays its d_mErrors to cerr, with associated filenames.
The actual command used to compile the `.cc` files is as folows:

```
string const command = "g++ -fdiagnostics-color=always --std=c++17 -Wall -O2 -c -o ./tmp/o/117"
+ filename + string(".o ") + filepath;
```

N.B.: The `-fdiagnostics-color=always` switch forces the colors that `g++` errors typically output to also be present in this case.
The same `pipe` and `popen` process is then used to link the files together. Considering the fact that all `.cc` files were (ideally) compiled, the following command can be used to link the resulting `.o` files in a subfolder of the current working directory:

```
g++ -o ./tmp/bin/binary ./tmp/o/*.o
```

N.B.: That folder is chosen to ensure some compatibility with Icmake. If, however, another folder was chosen instead of the current working directory, the `tmp` folder of that location is used here instead.
Of course, since sometimes static libraries must be specified (e.g. in the case of using the filesystem or pthread libraries), this is where any optional argv arguments should be appended.
Lastly, the folder containing the temporary files is deleted (i.e. `./tmp/bin/o`).