2-mutex

June 2, 2024

1 Sharing data between threads

1.1 Potential problem

The threads launched by the same program see the same memory area. They can possibly interact at the same time on the same data and produce undesirable effects. In the following example the functions `compute` and `print` are executed within two different threads. We see that `print` can interfere during the execution of `compute` between the increment of the number and the calculation of the square, which can produce an erroneous display.

[1]:
```
%%file tmp.mutex.h

#include <iostream>
#include <chrono>
#include <thread>
#include <cstdlib>
#include <cassert>
#include <mutex>

using namespace std::chrono_literals ;
```

```
Writing tmp.mutex.h
```

[20]:
```
%%file tmp.mutex-calculator.h

class Calculator
{
 public :
   void compute()
    {
     m_number++ ;
     std::this_thread::sleep_for(900us) ;
     m_square = m_number*m_number ;
    }
   void print()
    {  std::cout<<m_number<<" squared "<<m_square<<std::endl ; }
 private :
   int m_number{}, m_square{} ;
} ;
```

Overwriting tmp.mutex-calculator.h

[21]:
```
%%file tmp.mutex-repeaters.h

void repeat_print( int nb, Calculator & c )
{
 for ( int i=0 ; i<nb ; ++i )
  {
   c.print() ;
   std::this_thread::sleep_for(1ms) ;
  }
}
void repeat_compute( int nb, Calculator & c )
{
 for ( int i=0 ; i<nb ; ++i )
  {
   c.compute() ;
   std::this_thread::sleep_for(100us) ;
  }
}
```

Overwriting tmp.mutex-repeaters.h

[22]:
```
%%file tmp.mutex.cpp

#include "tmp.mutex.h"
#include "tmp.mutex-calculator.h"
#include "tmp.mutex-repeaters.h"

int main( int argc, char * argv[] )
{
  assert(argc==2) ;
  int nb = atoi(argv[1]) ;
  Calculator c ;
  std::thread t1(repeat_print,nb,std::ref(c)) ;
  std::thread t2(repeat_compute,nb,std::ref(c)) ;
  t1.join() ; t2.join() ;
 }
```

Overwriting tmp.mutex.cpp

[23]:
```
%%file tmp.mutex.sh
echo

rm -f tmp.mutex.exe \
&& g++ -std=c++17 -lpthread tmp.mutex.cpp -o tmp.mutex.exe\
&& ./tmp.mutex.exe $*
```

```
echo
```

```
Overwriting tmp.mutex.sh
```

[28]:
```
! bash -l tmp.mutex.sh 10
```

```
0 squared 0
1 squared 1
2 squared 4
3 squared 9
4 squared 9
5 squared 16
6 squared 25
7 squared 36
8 squared 49
9 squared 64
```

1.2 Locking a portion of code

To avoid the above situation, C ++ 11 offers the notion of a lock via the `std::mutex` class, which guarantees that a certain portion of code is only executed by one process at a time. Below, a correction for the class `Calculator`.

[29]:
```
%%file tmp.mutex-calculator.h

class Calculator
{
 public :
   void compute()
    {
     m_mtx.lock() ; // lock acquisition
     m_number++ ;
     std::this_thread::sleep_for(900us) ;
     m_square = m_number*m_number ;
     m_mtx.unlock() ; // lock release
    }
   void print()
    {
     m_mtx.lock() ; // lock acquisition
     std::cout<<m_number<<" squared "<<m_square<<std::endl ;
     m_mtx.unlock() ; // lock release
    }
 private :
   int m_number{} ,m_square{} ;
   std::mutex m_mtx ; // can only be acquired by one thread at a time
} ;
```

```
Overwriting tmp.mutex-calculator.h
```

[34]:
```
! bash -l tmp.mutex.sh 10
```

```
0 squared 0
1 squared 1
2 squared 4
3 squared 9
4 squared 16
5 squared 25
6 squared 36
7 squared 49
8 squared 64
9 squared 81
```

BEWARE: the mutex used by multiple threads must be the same single object. It can be passed by reference to processes and their functions via an `std::ref`, or it can be enclosed in an object passed in this way. It can also be a global, static, or heap-allocated object with a `new`. In all cases, for the locking to be effective, the different processes involved must view the same object.

1.3 Protection against exceptions

This code is not completely protected against the occurrence of an exception, which could leave locks blocked indefinitely. As with all resources, we can improve this code with an object responsible for acquiring a lock on construction and releasing it on destruction. To manage `std::mutex` in this way, C++11 introduced `std::lock_guard`, and C++17 improved it with `std::scoped_lock`.

[35]:
```
%%file tmp.mutex-calculator.h

class Calculator
{
 public :
   void compute()
    {
     std::scoped_lock<std::mutex> lg(m_mtx) ;
     m_number++ ;
     std::this_thread::sleep_for(500us) ;
     m_square = m_number*m_number ;
    }
   void print()
    {
     std::scoped_lock<std::mutex> lg(m_mtx) ;
     std::cout<<m_number<<" squared "<<m_square<<std::endl ;
    }
 private :
   int m_number{} ,m_square{} ;
   std::mutex m_mtx ; // can only be acquired by one thread at a time
```

```
} ;
```

```
Overwriting tmp.mutex-calculator.h
```

[36]:
```
! bash -l tmp.mutex.sh 10
```

```
0 squared 0
2 squared 4
4 squared 16
6 squared 36
8 squared 64
10 squared 100
10 squared 100
10 squared 100
10 squared 100
10 squared 100
```

1.3.1 Additional remarks

- `mutex::try_lock()` allows to check whether or not the mutex is locked.
- `unique_lock` is a "movable" variant of `lock_guard`.
- `recursive_mutex`: can be acquired more than once (incrementation of a counter) and freed as many times (decrementation of the counter) for recursive approaches.
- `timed_mutex`: the `try_lock_for` and `try_lock_until` methods allow to set a time limit for lock acquisition.
- `recursive_timed_mutex`: combines the properties of `recursive_mutex` and `timed_mutex`.

2 Questions ?

3 Exercise

In the program below, we added a display within the `complexes_pow` function, which is executed by threads. Normally, when running the program several times, you will notice anomalies in the display. Indeed, like any variable seen and shared by several threads, `std::cout` can malfunction when it is used at the same time by several threads. To solve this, use an `std::mutex` to lock the portion of code that is displaying.

[37]:
```
%%file tmp.mutex.cpp

#include <complex>
#include <vector>
#include <iostream>
#include <cassert>
#include <cmath>
#include <thread>
#include <mutex>
```

```
using Real = double ;
using Complex = std::complex<Real> ;
using Complexes = std::vector<Complex> ;

// random unitary complexes
void generate( Complexes & cs )
 {
  srand(1) ;
  for ( auto & c : cs )
   {
    Real angle {rand()/(Real(RAND_MAX)+1)*2.0*M_PI} ;
    c = Complex{std::cos(angle),std::sin(angle)} ;
   }
 }

// compute a slice of xs^degree and store it into ys
// xs.size() must be a multiple of nb_slices
void complexes_pow
 ( std::size_t num_slice, std::size_t nb_slices,
   Complexes const & xs, int degree, Complexes & ys )
 {
  assert((xs.size()%nb_slices)==0) ;
  auto slice_size {xs.size()/nb_slices} ;
  auto min {num_slice*slice_size} ;
  auto max {(num_slice+1)*slice_size} ;
  std::cout<<"complexes_pow "<<num_slice<<" min : "<<min<<std::endl ;
  std::cout<<"complexes_pow "<<num_slice<<" max : "<<max<<std::endl ;
  for ( auto i {min} ; i<max ; ++i )
   {
    ys[i] = Complex{1.,0.} ;
    for ( int d=0 ; d<degree ; ++d )
     { ys[i] *= xs[i] ; }
   }
 }

// display the angle of the complexes global product
void postprocess( Complexes const & cs )
 {
  Complex prod(1.,0.) ;
  for( auto c : cs ) { prod *= c ; }
  double angle {atan2(prod.imag(),prod.real())} ;
  std::cout<<"result = "<<static_cast<int>(angle/2./M_PI*360.)<<"\n" ;
 }

// main program
int main ( int argc, char * argv[] )
```

```
{
 assert(argc==4) ;
 std::size_t nbtasks {std::stoul(argv[1])} ;
 std::size_t dim {std::stoul(argv[2])} ;
 int degree {std::stoi(argv[3])} ;

 // prepare input
 Complexes input(dim) ;
 generate(input) ;

 // compute
 Complexes output(dim) ;
 std::size_t numtask ;
 std::vector<std::thread> workers ;
 for ( numtask = 0 ; numtask<nbtasks ; ++numtask )
  { workers.emplace_back(complexes_pow,numtask,nbtasks,std::
↪ref(input),degree,std::ref(output)) ; }
 for ( auto & worker : workers )
  { worker.join() ; }

 // post-process
 postprocess(output) ;
}
```

Overwriting tmp.mutex.cpp

[38]:
```
%%file tmp.mutex.sh
echo

rm -f tmp.mutex.exe \
&& g++ -std=c++17 -lpthread tmp.mutex.cpp -o tmp.mutex.exe\
&& ./tmp.mutex.exe $*

echo
```

Overwriting tmp.mutex.sh

[39]:
```
! bash -l tmp.mutex.sh 2 2 3
```

```
complexes_pow 0 min : 0complexes_pow
complexes_pow 0 max : 1
1 min : 1
complexes_pow 1 max : 2
result = -106
```

[41]:
```
! bash -l tmp.mutex.sh 4 1024 100000
```

```
complexes_pow 0 min : 0
complexes_pow complexes_pow 0 max : 256
3 min : 768
complexes_pow 3 max : 1024complexes_pow 1 min : 256

complexes_pow 1 max : 512
complexes_pow 2 min : 512
complexes_pow 2 max : 768
result = -77
```