Atomics, Locks, and Tasks

Rainer Grimm

Training, Coaching, and Technology Consulting

www.ModernesCpp.de

Concurrency in C++20

C++11	C++14	C++17	C++20	C++23
2011	2014	2017	2020	2023
• Memory model • Threads • Mutexes and locks • Thread local data • Condition variables • Tasks	• Reader-writer locks	• Parallel STL	• `std::jthread` • Atomic smart pointers • Latches and barriers • Semaphores • Coroutines	• Executors • `std::future` extensions • Transactional memory • Task blocks • Data-parallel vector library

Challenge – Shared, Mutable State

Data race:

- At least two threads access shared state concurrently. At least one thread tries to modify the shared state.

⇒ The program has undefined behaviour.

The program has catch-fire semantic.

Requirement for a data race:

- shared, mutable state

Challenge – Shared, Mutable State

Critical Section:

- Shared state which can only be accessed concurrently by at most one thread.

⇒ Exclusive access has to be guaranteed.

Critical sections are the enemies of performance.

Challenge – Shared, Mutable State

Deadlock:

- A deadlock is a state in which at least one thread is blocked forever because it waits for the release of a resource it does never get.

⇒ The thread and, therefore, the program stands still.
The program has to be interrupted.

Requirement:

- Waiting for a resource (critical section, file, socket, memory, …)

Safe Concurrency

- Atomics
- Locks
- Tasks
- Parallel STL
- Patterns

Safe Concurrency

- Atomics
- Locks
- Tasks
- Parallel STL
- Patterns

Challenge – Shared Counter

An counter is concurrently incremented by different threads.

- Typical use-case:
  - Reference counter of a `std::shared_ptr`

Shared Counter – `int`

```
auto constexpr num = 1000;
vector<thread> vecThread(num);
int i{};

for (auto& t: vecThread){ t = thread([&i]{
    for(int n = 0; n < num; ++n) cout << ++i << " ";
  });
}

for (auto& t: vecThread){ t.join(); }

cout << "num * num:  " << num * num << endl;
cout << "i:          " << i << endl;
cout << "num * num - i: " << num * num - i << endl;
```

Shared Counter – `int`

File Edit View Bookmarks Settings Help

```
98431 998432 998433 998434 998435 998436 998437  998439998438  998440 998441998442  998443998444  998445
 998446998446  998447 998448 998448 998449998450 998451 998452 998453 998454 998455 998456 998457 998458
 998459 998460 998461 998462 998463 998464 998465 998466 998467 998468 998469 998470 998471 998472 99847
3 998474 998475 998476 998477 998478 998479 998480 998481 998482 998483 998484 998485 998486 998487 9984
88 998489 998490 998491 998492 998493 998494 998495 998496 998497 998498 998499 998500 998501 998502 998
503 998504 998505 998506 998507 998508 998509 998510 998511 998512 998513 998514 998515 998516 998517 99
8518 998519 998520 998521 998522 998523 998524 998525 998526 998527 998528 998529 998530 998531 998532 9
98533 998534 998535 998536 998537 998538 998539 998540 998541 998542 998543 998544 998545 998546 998547
998548 998549 998550 998551 998552 998553 998554  998555 998556 998557 998558 998559 998560 998561 99856
2 998563 998564 998565 998566 998567 998568 998569 998570 998571 998572 998573 998574 998575 998576 9985
77 998578 998579 998580 998581 998582 998583 998584 998585 998586 998587 998588 998589 998590 998591 998
592 998593 998594 998595 998596 998597 998598 998599 998600 998601 998602 998603 998604 998605 998606 99
8607 998608 998609 998610 998611 998612 998613 998614 998615 998616 998617 998618 998619 998620 998621 9
98622 998623 998624 998625 998626 998627 998628 998629 998630 998631 998632 998633 998634 998635 998636
998637 998638 998639 998640 998641 998642 998643 998644 998645 998646 998647 998648 998649 998650 998651
 998652 998653 998654 998655 998656 998657 998658 998659 998660 998661 998662 998663 998664 998665 99866
6 998667 998668 998669 998670 998671 998672 998673 998674 998675 998676 998677 998678 998679 998680 9986
81  998682 998683 998684 998685 998686 998687 998688 998689 998690 998691 998692 998693 998694 998695 99
8696 998697 998698 998699 998700 998701 998702 998703 998704 998705 998706 998707 998708 998709 998710 9
98711 998712 998713 998714 998715 998716 998717 998718 998719 998720 998721 998722 998723 998724 998725
998726 998727 998728 998729 998730 998731 998732 998733 998734 998735 998736 998737 998738 998739 998740
 998741 998742 998743 998744 998745 998746 998747 998748 998749 998750 998751 998752 998753 998754 99875
5 998756 998757 998758 998759 998760 998761 998762 998763 998764 998765 998766 998767 998768

num * num:  1000000
i:          998768
num * num - i: 1232

rainer@linux:~>
```

rainer : bash

Shared Counter – `int`

```
rainer@linux:~> g++-6 -O3 -fsanitize=thread -g sharedCounterInt.cpp -o sharedCounterInt
rainer@linux:~> sharedCounterInt

==================
WARNING: ThreadSanitizer: data race (pid=2838)
  Read of size 4 at 0x7ffc66d1959c by thread T2:
```

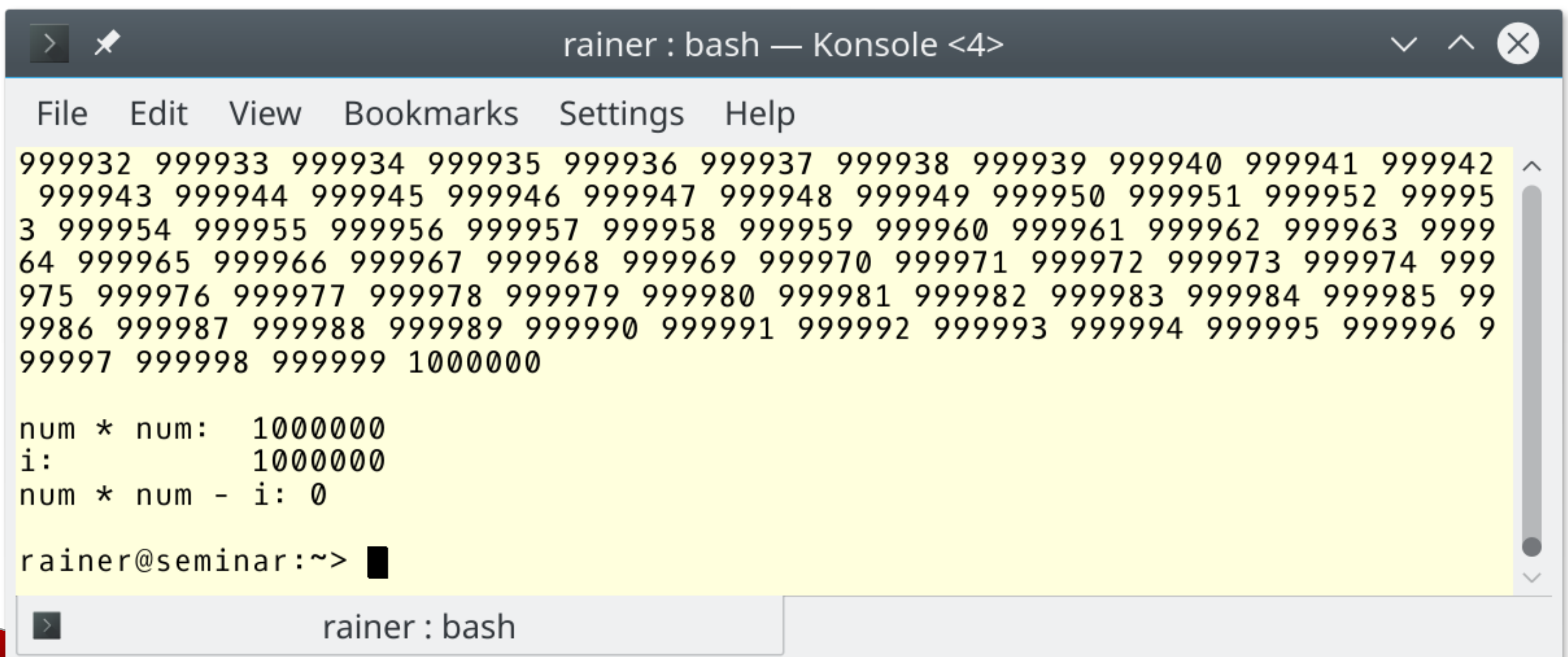
```
SUMMARY: ThreadSanitizer: data race /home/rainer/sharedCounterInt.cpp:18 in operator()
==================
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 3
51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 8
98 99 100

num * num:  100
i:          100
num * num - i: 0

ThreadSanitizer: reported 1 warnings
rainer@linux:~>
```

Shared Counter - Atomic

The shared counter has to be protected.

Use an atomic shared counter.

```
int i{};   std::atomic<int> i{};
```



Atomics

- Examples:
  - `sharedCounterInt.cpp`
  - `sharedCounterAtomicInt.cpp`
- Issues:
  - Expensive synchronisation
- Further Information:
  - Memory model

Safe Concurrency

- Atomics
- **Locks**
- Tasks
- Parallel STL
- Patterns

Challenge – Shared State

A variable is concurrently modified by different threads.

- The variable is the criticial section.
- To protected the criticial, a exclusion mechanism such as `std::mutex` is used.
- `std::mutex` guarantees that at most one thread can access the critical section.

Challenge – Shared State

```
std::mutex m;
m.lock();
sharVar = getVar();
m.unlock();
```

- Assumptions:
  - `sharVar` is concurrently used
  - `getVar()` is a unknown function
  - `getVar()` may be modified in the future

➡ ISSUES ?

Shared State – Lock

First Improvement

```
auto res = getVar();
m.lock();
sharVar = res;
m.unlock();
```

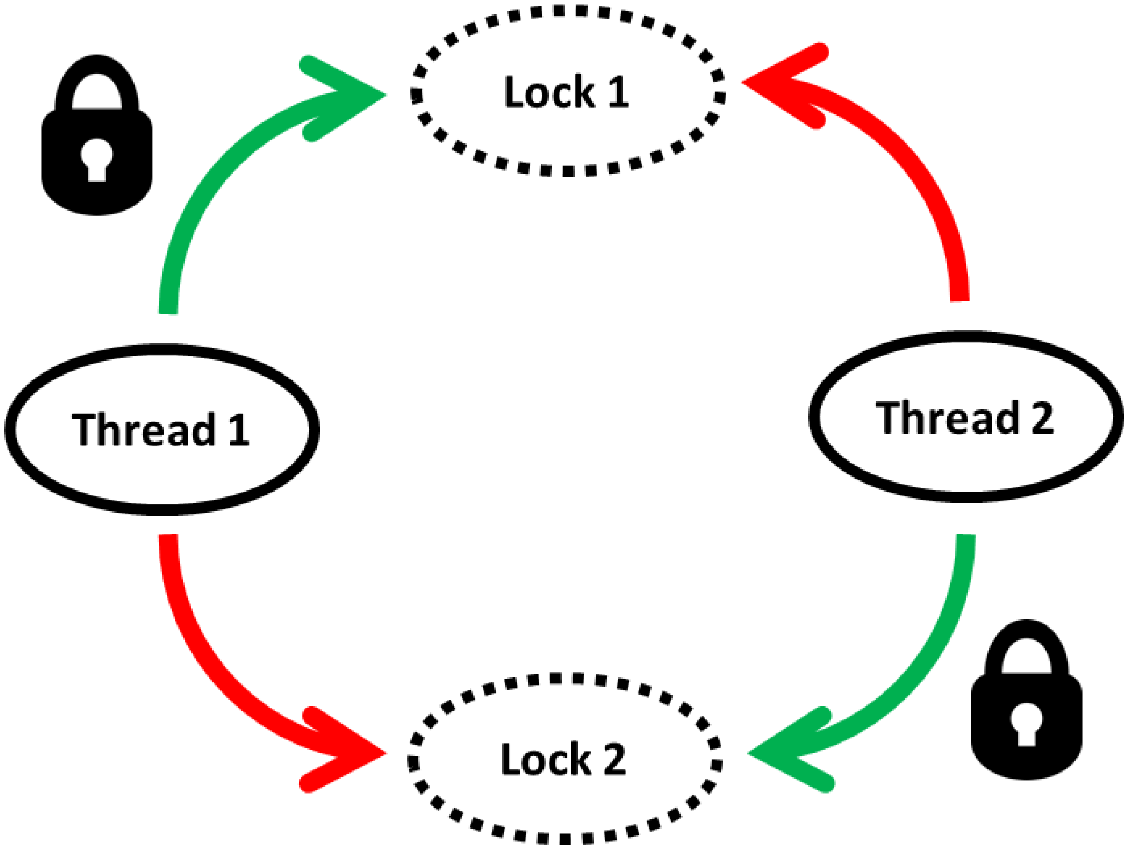
Second Improvement

```
auto res = getVar();
{
  std::lock_guard<std::mutex> lo(m);
  sharVar = res;
}
```

Put a mutex into a lock.
NNM: No Nacked Mutex.

Challenge – More Shared State

A thread needs more than one critical section at one point in time.



Challenge – More Shared State

A thread needs more than one critical section at one point in time.

```
struct CriticalData{
  mutex mut;
};

void deadLock(CriticalData& a, CriticalData& b){
  lock_guard<mutex>guard1(a.mut);
  this_thread::sleep_for(chrono::milliseconds(1));
  lock_guard<mutex>guard2(b.mut);
  // do something with a and b
}
...
thread t1([&]{ deadLock(c1,c2);} );
thread t2([&]{ deadLock(c2,c1);} );
```

More Shared State – Locks

- Delayed Locking

```
void deadLock(CriticalData& a, CriticalData& b){
  unique_lock<mutex>guard1(a.mut, defer_lock);
  this_thread::sleep_for(chrono::milliseconds(1));
  unique_lock<mutex>guard2(b.mut, defer_lock);
  lock(guard1, guard2);
  // do something with a and b
}
```

- Scoped Locking

```
void deadLock(CriticalData& a, CriticalData& b){
  scoped_lock(a.mut, b.mut);
  // do something with a and b
}
```

More Shared State – Locks

- Deadlock

rainer : deadLock — Konsole

File Edit View Bookmarks Settings Help

```
rainer@seminar:~> deadLock

Thread: 140449451566848 locking of the first mutex
Thread: 140449443174144 locking of the first mutex
```

rainer : deadLock

- Deadlock Resolved

rainer : bash — Konsole <4>

File Edit View Bookmarks Settings Help

```
rainer@seminar:~> deadLockResolved

Thread: 139966140966656 defer the locking of the first mutex
Thread: 139966132573952 defer the locking of the first mutex
Thread: 139966140966656 defer the locking of the second mutex
Thread: 139966140966656 locking them both atomically
Thread: 139966132573952 defer the locking of the second mutex
Thread: 139966132573952 locking them both atomically

rainer@seminar:~>
```

rainer : bash

Challenge – Expensive Synchronisation

A shared counter can be synchronised in various ways.

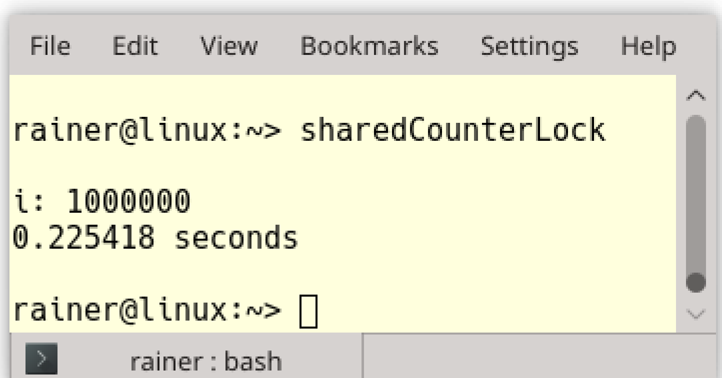
- Locks
- Atomics
- Local variables with atomics

Most important rule for concurrency:

⇒ Don’t share!

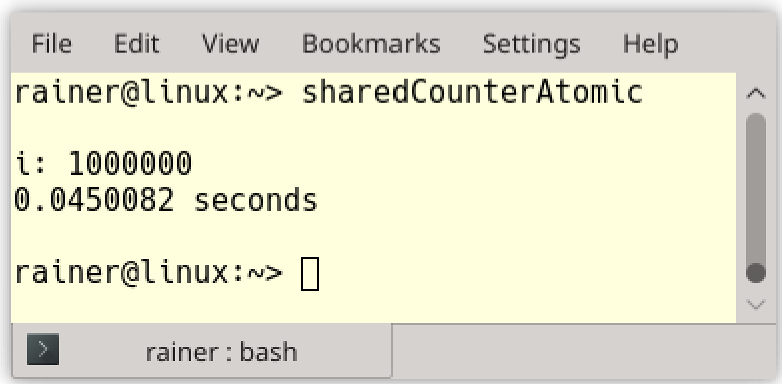
Expensive Synchronisation – Lock

```
mutex mut;

auto start = chrono::steady_clock::now();
for (auto& t: vecThread){
  t = thread([&i, &mut, &num]{ for(int n = 0; n < num; ++n){
          lock_guard<mutex> lo(mut);
          ++i;
        }
    });
}
chrono::duration<double> dur = chrono::steady_clock::now() - start;
```



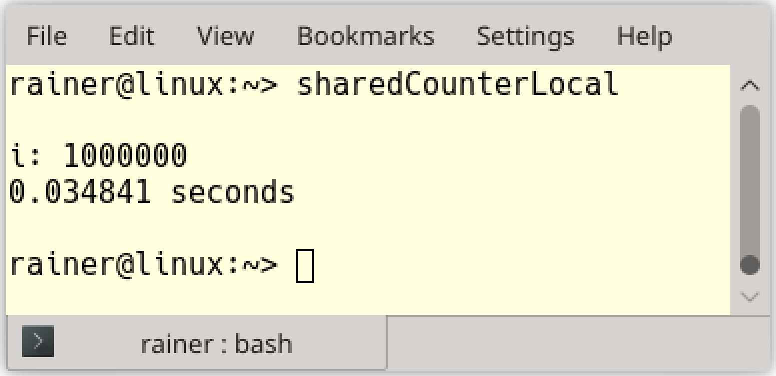
Expensive Synchronisation – Atomic

```
atomic<int> i{};

auto start = chrono::steady_clock::now();
for (auto& t: vecThread){
  t = thread([&i, &num]{ for(int n = 0; n < num; ++n){ ++i; } });
}

chrono::duration<double> dur = chrono::steady_clock::now() - start;
```



Expensive Synchronisation – Local

```
atomic<int> i{};

auto start = chrono::steady_clock::now();
for (auto& t: vecThread){
  t = thread([&i, &num]{
    int local{};
    for(int n = 0; n < num; ++n){
      ++local;
    }
    i += local;
  });
}
```




```
chrono::duration<double> dur = chrono::steady_clock::now() - start;
```

Locks

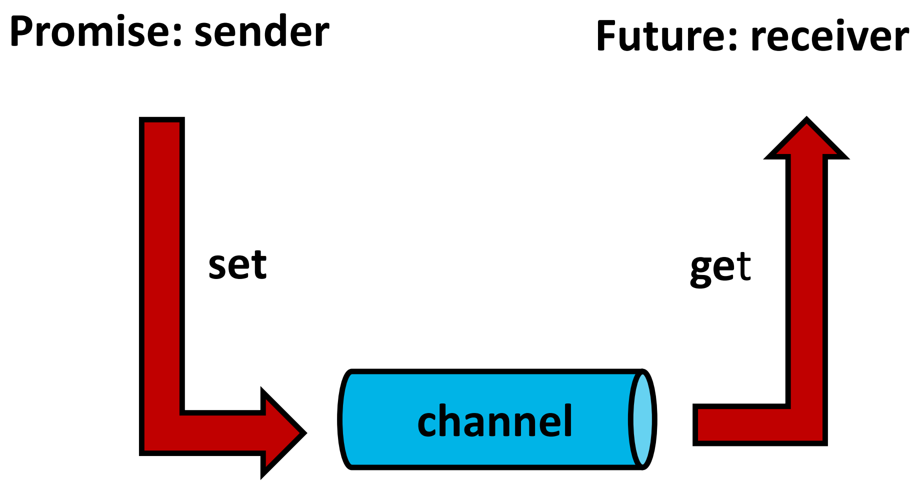
- Examples:
  - `deadLock.cpp`
  - `deadLockResolved.cpp`
  - `sharedCounterLock.cpp`
  - `sharedCounterAtomic.cpp`
  - `sharedCounterLocal.cpp`

Safe Concurrency

- Atomics
- Locks
- **Tasks**
- Parallel STL
- Patterns

Tasks as Data Channels

A task is like a data channel.



The promise

- is the data sender.
- can serve many futures.
- can send values, exceptions and notifications.

The future

- is the data receiver.
- calls get and is eventually blocked.

Threads versus Tasks

Thread

```
int res;
thread t([&]{ res= 3+4; });
t.join();
cout << res << endl;
```

Task

```
auto fut=async([]{ return 3+4; });
cout << fut.get() << endl;
```

Characteristic	Thread	Task
Header	`<thread>`	`<future>`
Participants	Creator and child thread	Promise and future
Communication	Shared variable	Communication channel
Thread creation	Obligatory	Optional
Synchronisation	join call waits	get call waits
Exception in the child thread	Child and creator thread terminate	Exception can be send to the future

Challenge – Wait for the Child

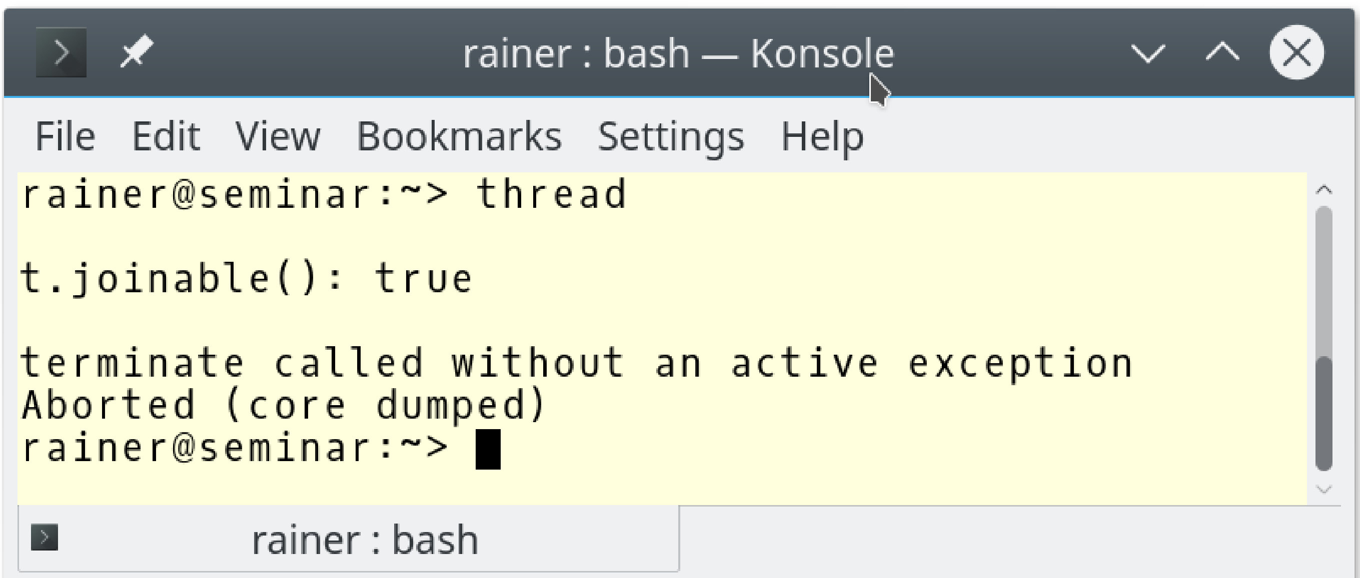
The creator of a thread must take care of its child.

- The creator
  - Waits for its child `t`: `t.join();`
  - Detaches itself from its child `t`: `t.detach();`
- A thread `t` is joinable if `t.join()` or `t.detach()` was not performed.

A joinable thread `t` calls in its destructor the exception `std::terminate()`. ⇒ Program termination

Wait for the Child – Joinable

```
int main(){
  std::thread t{[]{ std::cout << "New thread"; }};
  std::cout << "t.joinable(): " << t.joinable();
}
```



Wait for the Child – `scoped_thread`

```
class scoped_thread{
  std::thread t;

public:
  explicit scoped_thread(std::thread t_): t(std::move(t_)){
    if (!t.joinable()) throw std::logic_error("No thread");
  }

  ~scoped_thread(){
    t.join();
  }

  scoped_thread(scoped_thread&)= delete;
  scoped_thread& operator=(scoped_thread const &)= delete;
};
```

Wait for the Child – `jthread`

```
#include <iostream>
#include "jthread.hpp"

int main(){

  std::jthread thr{[]{
    std::cout << "Joinable std::thread" << std::endl; }
  };

  std::cout << "thr.joinable(): "
            << thr.joinable() << std::endl;

}
```

Challenge – Shared State

A compute intensive job has to be performed.

```
long long getDotProduct(std::vector<int>& v,
                        std::vector<int>& w){
  return std::inner_product(v.begin(), v.end(),
                            w.begin(),
                            0LL);
}
```

⇒ There are no data dependencies.

Shared State – Lock

```
long long getDotProduct(vector<int>& v, vector<int>& w){

  long long res{};
  mutex mut;

  thread t1{[&]{
      auto prod = inner_product(&v[0], &v[v.size()/4], &w[0], 0LL);
      lock_guard<mutex> lockRes(mut);
      res += prod;
    }
  };
  ...
  t1.join(), t2.join(), t3.join(), t4.join();

  return res;
}
```

Shared State – `std::async`

```
long long getDotProduct(vector<int>& v, vector<int>& w){

  auto future1= async([&]{
    return inner_product(&v[0], &v[v.size()/4], &w[0], 0LL); });

  auto future2= async([&]{
    return inner_product(&v[v.size()/4], &v[v.size()/2],
                                   &w[v.size()/4], 0LL);});

  auto future3= async([&]{
    return inner_product(&v[v.size()/2], &v[v.size()*3/4],
                                  &w[v.size()/2], 0LL);});

  auto future4= async([&]{
    return inner_product(&v[v.size()*3/4], &v[v.size()],
                                  &w[v.size()*3/4], 0LL);});

  return future1.get() + future2.get() + future3.get() + future4.get();
}
```

Shared State – Parallel STL

```
long long getDotProduct(std::vector<int>& v,
                        std::vector<int>& w){
  return std::transform_reduce(std::execution::par,
                               v.begin(), v.end(),
                               w.begin(),
                               0LL);
}
```

Challenge – Delayed Execution

`std::packaged_task` wraps a callable so that it can be invoked later.

```
while (! allTasks.empty()){
  packaged_task<workPackage> myTask = move(allTasks.front());
  allTasks.pop_front();
  thread sumThread(move(myTask), v, w, begin, end);
  begin = end;
  end += increment;
  sumThread.detach();
}
```

Challenge – Notifications

Condition variables enable you to synchronise threads.

- Typical use-cases
  - Sender - Receiver
  - Producer – Consumer

- `condition_var`
  - Needs the header `<condition_var>`.
  - Can play the role of the sender and of the receiver.

Challenge – Notifications

- Sender sends a notification.

Method	Description
`cv.notify_one()`	Notifies one waiting thread
`cv.notify_all()`	Notifies all waiting threads

- Receiver is waiting for the notification while holding the mutex.

Method	Description
`cv.wait(lock, ... )`	Waits for the notification
`cv.wait_for(lock, relTime, ... )`	Waits for the notification for a time period
`cv.wait_until(lock, absTime, ... )`	Waits for the notification until a time point

In order to protect against a spurious wakeup or a lost wakeup, the `wait` method should be used with an optional predicate.

Challenge – Notifications

Thread 1: Sender

- Does its work
- Notifies the receiver

```
// do the work
{
  lock_guard<mutex> lck(mut);
  ready= true;
}
condVar.notify_one();
```

Thread 2: Receiver

- Waits for its notification while holding the lock
  - Gets the lock
  - Checks and continues to sleep
- Does its work
- Releases the lock

```
{
  unique_lock<mutex>lck(mut);
  condVar.wait(lck,[]{return ready;});
  // do the work
}
```

Notifications – Atomics

When the sender uses an atomic `dataReady` without a critical section, a *lost wakeup* may happen.

```
std::unique_lock<std::mutex> lck(mutex_);
condVar.wait(lck, []{ return dataReady.load(); });
```

Implementation

```
std::unique_lock<std::mutex> lck(mutex_);
while ( ![]{ return dataReady.load(); }() ) {
    // time window (1)
    condVar.wait(lck);
}
```

Notifications – Tasks

```
void waitingForWork(std::future<void>&& fut){
    std::cout << "Worker: Waiting for work." << std::endl;
    fut.wait();
    std::cout << "Preparation done." << std::endl;
}
void setDataReady(std::promise<void>&& prom){
    std::cout << "Sender: Data is ready."  << std::endl;
    prom.set_value();
}
...
std::promise<void> sendReady;
auto fut= sendReady.get_future();

std::thread t1(waitingForWork, std::move(fut));
std::thread t2(setDataReady, std::move(sendReady));

t1.join(), t2.join();
```

Challenge – Exceptions

The “thread of execution” throws an exception:

Any return value from the function is ignored. If the function throws an exception, std::terminate is called. In order to pass return values or exceptions back to the calling thread, std::promise or std::async may be used. (cppreference.com).

Exceptions – Tasks

```
struct Div{
  void operator()(promise<int>&& intPromise, int a, int b){
    try{
      if (b == 0) throw runtime_error("illegal division by zero");
      intPromise.set_value(a/b);
    }
    catch ( ... ){ intPromise.set_exception(current_exception()); }
  }
};
...
Div div;
thread divThread(div, move(divPromise),20, 0);
try{
  cout << "20 / 0 = " << divResult.get() << endl;
}
catch (runtime_error& e){ cout << e.what() << endl; }
```

Tasks

- Examples:
  - `thread.cpp`
  - `scoped_thread.cpp`
  - `jthreadJoinable.cpp`
  - `dotProduct.cpp`
  - `dotProductThread.cpp`
  - `dotProductAsync.cpp`
  - `dotProductCpp17.cpp`
  - `dotProductPackagedTask.cpp`
  - `conditionVariable.cpp`
  - `conditionVariableAtomic.cpp`
  - `promiseFutureException.cpp`
  - `promiseFutureSynchronize.cpp`

Safe Concurrency

Atomics

Locks

Tasks

Parallel STL

Patterns

Challenge – Parametrised Algorithms

You want to perform an algorithm in a sequential, parallel, or parallel and vectorised version.

⇒ Use the Parallel STL of C++17

Parametrised Algorithms – Execution Policy

You can choose the execution policy of an algorithm.

- Execution policies

  `std::execution::seq`
  - Sequential in one thread

  `std::execution::par`
  - Parallel

  `std::execution::par_unseq`
  - Parallel and vectorised → SIMD

Parametrised Algorithms - Vectorisation

```
const int SIZE = 8;
int vec[]={1, 2 , 3, 4, 5, 6, 7, 8};
int res[SIZE] = {0,};

int main(){
    for (int i= 0; i < SIZE; ++i){
        res[i] = vec[i] + 5;
    }
}
```

Not vectorised

```
movslq  -8(%rbp), %rax
movl    vec(,%rax,4), %ecx
addl    $5, %ecx
movslq  -8(%rbp), %rax
movl    %ecx, res(,%rax,4)
```

Vectorised

```
movdqa  .LCPI0_0(%rip), %xmm0    # xmm0 = [5,5,5,5]
movdqa  vec(%rip), %xmm1
paddd   %xmm0, %xmm1
movdqa  %xmm1, res(%rip)
paddd   vec+16(%rip), %xmm0
movdqa  %xmm0, res+16(%rip)
xorl    %eax, %eax
```

Parametrised Algorithms – `std::sort`

```
using namespace std;
vector<int> vec = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};


sort(vec.begin(), vec.end());      // sequential as ever

sort(execution::seq, vec.begin(), vec.end());        // sequential
sort(execution::par, vec.begin(), vec.end());        // parallel
sort(execution::par_unseq, vec.begin(), vec.end()); // par + vec
```

Parametrised Algorithms – All Algorithms

adjacent_difference, adjacent_find, all_of any_of, copy, copy_if, copy_n, count, count_if, equal, **exclusive_scan**, fill, fill_n, find, find_end, find_first_of, find_if, find_if_not, **for_each**, **for_each_n**, generate, generate_n, includes, **inclusive_scan**, inner_product, inplace_merge, is_heap, is_heap_until, is_partitioned, is_sorted, is_sorted_until, lexicographical_compare, max_element, merge, min_element, minmax_element, mismatch, move, none_of, nth_element, partial_sort, partial_sort_copy, partition, partition_copy, **reduce**, remove, remove_copy, remove_copy_if, remove_if, replace, replace_copy, replace_copy_if, replace_if, reverse, reverse_copy, rotate, rotate_copy, search, search_n, set_difference, set_intersection, set_symmetric_difference, set_union, sort, stable_partition, stable_sort, swap_ranges, transform, **transform_exclusive_scan**, **transform_inclusive_scan**, **transform_reduce**, uninitialized_copy, uninitialized_copy_n, uninitialized_fill, uninitialized_fill_n, unique, unique_copy

Parametrised Algorithms – map_reduce

`std::transform_reduce`

- Haskells function `map` is called `std::transform` in C++
- `std::transform_reduce` ➡ `std::map_reduce`

```
std::vector<std::string> strVec{"Only", "for", "testing", "purpose"};

std::size_t res = std::transform_reduce(std::execution::par,
                         strVec.begin(), strVec.end(), 0,
                         [](std::size_t a, std::size_t b){ return a + b; },
                         [](std::string s){ return s.length(); });

std::cout << res;     // 21
```

Parametrised Algorithms – Data Races

- Danger of data races or deadlocks

```
int numComp = 0;
std::vector<int> vec = {1, 3, 8, 9, 10};
std::sort(std::execution::par, vec.begin(), vec.end(),
       [&numComp](int fir, int sec){ numComp++; return fir < sec; }
);
```

➡ The access to **`numComp`** has to be atomic.

Parametrised Algorithms – Availability

- Support for `std::execution::par`
  - GCC and Clang:
    - ~~GCC 9.1 and~~ Intel TBB (Solarian Programmer)
  - MSVC with Visual Studio 2017 15.8 (`/std=c++latest`)
    - sequential execution with `std::execution::par`

      `copy, copy_n, fill, fill_n, move, reverse, reverse_copy, rotate, rotate_copy, swap_ranges`

    - parallel execution

      `adjacent_difference, adjacent_find, all_of, any_of, count, count_if, equal, exclusive_scan, find, find_end, find_first_of, find_if, for_each, for_each_n, inclusive_scan, mismatch, none_of, partition, reduce, remove, remove_if, search, search_n, sort, stable_sort, transform, transform_exclusive_scan, transform_inclusive_scan, transform_reduce`

Parametrised Algorithms – Execution Time

```
getExecutionTime("execution::seq", [workVec]() mutable {
  transform(execution::seq, workVec.begin(), workVec.end(),
                            workVec.begin(),
                           [](double arg){ return tan(arg); });
    });

getExecutionTime("execution::par", [workVec]() mutable {
  transform(execution::par, workVec.begin(), workVec.end(),
                            workVec.begin(),
                            [](double arg){ return tan(arg); });
    });

getExecutionTime("execution::par_unseq", [workVec]() mutable {
  transform(execution::par_unseq, workVec.begin(), workVec.end(),
                                  workVec.begin(),
                                 [](double arg){ return tan(arg); });
    });
```

Parametrised Algorithms – Execution Time

x64 Native Tools-Eingabeaufforderung für VS 2017

```
C:\Users\rainer>cl.exe /EHsc /W4 /WX /std:c++latest /MD /O2 parallelSTLPerformance.cpp
Microsoft (R) C/C++-Optimierungscompiler Version 19.16.27025.1 für x64
Copyright (C) Microsoft Corporation. Alle Rechte vorbehalten.

parallelSTLPerformance.cpp
Microsoft (R) Incremental Linker Version 14.16.27025.1
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:parallelSTLPerformance.exe
parallelSTLPerformance.obj

C:\Users\rainer>parallelSTLPerformance.exe

std::execution::seq: 5.44017 sec.
std::execution::par: 0.455092 sec.
std::execution::par_unseq: 0.458994 sec.

C:\Users\rainer>
```

Parallel STL

- Examples:
  - `newAlgorithm.cpp`
  - `parallelSTLPerformance.cpp`

Safe Concurrency

Atomics

Locks

Tasks

Parallel STL

Patterns

Challenge – Use a Method Thread-Save

Thread-Safe Interface:

- The thread-save interface extend the critical region to the interface of an object.

- Antipattern: Each method uses internally a lock.
  - The performance is the objects goes down.
  - Deadlocks appear, when two methods call each other.

Thread-Save Interface

A deadlock due to entangled calls.

```
struct Critical{
    void method1(){
        lock(mut);
        method2();
        . . .
    }
    void method2(){
        lock(mut);
        . . .
    }
    mutex mut;
}
```

```
int main(){
    Critical crit;
    crit.method1();
}
```

Thread-Save Interface

- Solution:
  - All interface-methods (`public`) use a lock.
  - All implementation-methods (`protected` and `private`) must not use a lock.
  - The interface-methods call only implementation-methods.

Thread-Save Interface

```
class Critical{
public:
  void interface1() const {
    lock_guard<mutex> lockGuard(mut);
    implementation1();
  }
  void interface2(){
    lock_guard<mutex> lockGuard(mut);
    implementation2();
    implementation3();
    implementation1();
  }
private:
  void implementation1() const {
    cout << "implementation1: "
    }
  void implementation2(){
    cout << "implementation2: ";
  }
  void implementation3(){
    cout << "implementation3: ";
  }
mutable mutex mut;
};
```

Thread-Save Interface

- Challenges:
  - Virtual interface-methods, which are overwritten, need also a lock, if they are `private`.
  - `static`, mutable members of a class needs synchronisation on the class and not on the object level.

Thread-Save Interface

- Examples:
  - `threadSafeInterface.cpp`
  - `threadSafeInterfaceVirtual.cpp`

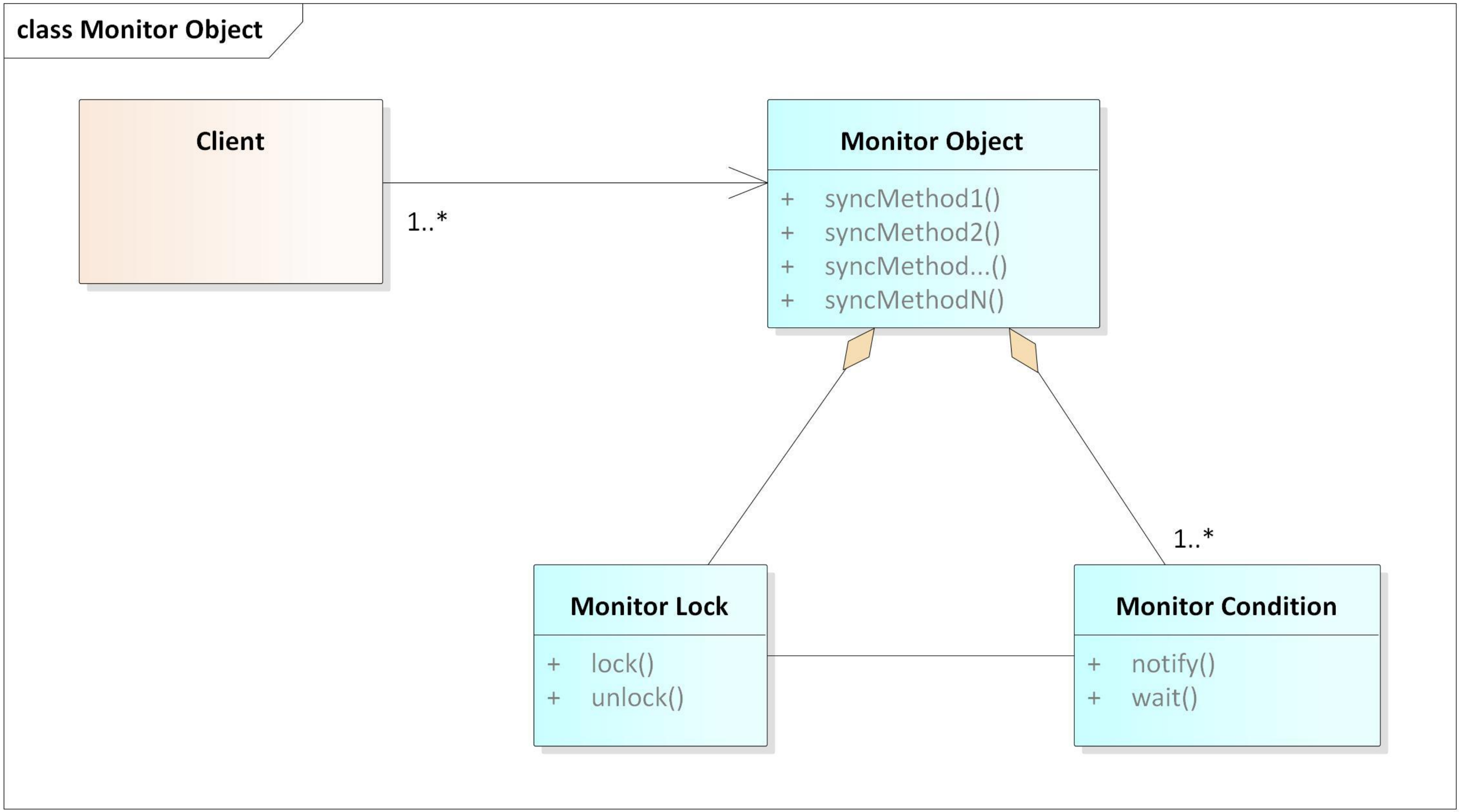
Challenge: Use an Object Thread-Safe

Monitor Object:

- The monitor object synchronises the access to an object in such way that at most one method can run at any moment in time.

- Each object has a monitor lock and a monitor condition.
- The monitor lock guarantees that only one client can execute a method of the object.
- The monitor condition notifies eventually waiting clients.

Monitor Object



Monitor Object

Monitor Object:

- support methods, which can run in the thread of the client

Synchronised Methods:

- Interface methods of the **monitor object**
- at most one method can run at any point in time
- the methods should implement the thread-save interface pattern

Monitor Lock:

- each **monitor object** has a **monitor lock**
- guarantees the exclusive access on the methods

Monitor Condition:

- allows various thread to store there method invocation
- when the current thread is done with its method execution, the next thread is woken up

Monitor Object

```
template <typename T>
class Monitor{
public:
  void lock() const { monitMutex.lock(); }
  void unlock() const { monitMutex.unlock(); }

  void notify_one() const noexcept { monitCond.notify_one(); }
  void wait() const {
    std::unique_lock<std::recursive_mutex> monitLock(monitMutex);
    monitCond.wait(monitLock);
  }

private:
  mutable std::recursive_mutex monitMutex;
  mutable std::condition_variable_any monitCond;
};
```

Monitor Object

```
template <typename T>
struct ThreadSafeQueue: public Monitor<ThreadSafeQueue<T>>{
  void add(T val){
    derived.lock();
    myQueue.push(val);
    derived.unlock();
    derived.notify_one();
  }
  T get(){
     derived.lock();
     while (myQueue.empty()) derived.wait();
     auto val = myQueue.front();
     myQueue.pop();
     derived.unlock();
     return val;
  }
private:
  std::queue<T> myQueue;
  ThreadSafeQueue<T>& derived = static_cast<ThreadSafeQueue<T>&>(*this);
};
```

Monitor Object

Advantages:

- The synchronisation is encapsulated in the implementation.
- The method execution is automatically stored and performed.
- The monitor object is a simple scheduler.

Disadvantages:

- The synchronisation mechanism and the functionality are strongly coupled and can, therefore, not so easily be changed.
- When the synchronised methods invoke a additional method of the monitor object, a deadlock may happen.

Patterns

- Examples:
  - `threadSafeInterface.cpp`
  - `threadSafeInterfaceVirtual.cpp`
  - `monitorObject.cpp`

- Further information:
  - CRTP

Blogs

www.grimm-jaud.de [De]

www.ModernesCpp.com [En]

Rainer Grimm

Training, Coaching, and Technology Consulting

www.ModernesCpp.de