

TDDD38/726G82: Adv. Programming in C++

Christoffer Holm

Department of Computer and information science

- 1 Threads
- 2 Synchronization
- 3 Sharing data
- 4 Thread communication
- 5 Parallel algorithms
- 6 Thinking about parallelism

- 1 **Threads**
- 2 Synchronization
- 3 Sharing data
- 4 Thread communication
- 5 Parallel algorithms
- 6 Thinking about parallelism

Threads

Introduction

- When writing performance critical software there are many techniques for improving performance.
- The single most important factor when it comes to performance is to efficiently use the hardware available to us. We have previously discussed how to write code that utilizes CPU caching and pre-fetching, but there is one aspect of the hardware we haven't really discussed: how to utilize the fact that modern computers have multiple CPUs.
- In this lecture we will introduce threading, specifically how to use the threading library from STL.

Threads

What are threads?

- A *thread* is a sequence of instructions within a process or program that can run independently.
- Threads *can* be run in *parallel* (if the hardware supports it). Otherwise threads will take turns executing on the CPU.
- Each program starts with *one* thread, if we need more we have to *dynamically* create threads from within the initial thread. This is called *thread creation*.

Threads

Thread creation

examples/1.cc

```
1  #include <iostream>
2  #include <thread>
3
4  void task()
5  {
6      auto id = std::this_thread::get_id();
7      std::cout << "thread " << id << " created!\n";
8  }
9
10 int main()
11 {
12     std::thread t1 { task };
13 }
```

Threads

Threading library

- A thread is represented by an `std::thread` object
- When we create a thread we have to pass it a *function* (or other *callable object*), which begins executing *as soon as possible* after creation.
- Each thread has its own execution context which we can access through the `std::this_thread` namespace. In the previous example we saw how to get the *ID* of a thread using the `get_id()` function.
- Let's see what happens if we run the example.

Threads

Thread creation – Problem

```
$ g++ 1.cc -lpthread  
$ ./a.out  
terminate called without an active exception  
Aborted (core dumped)
```

Threads

Thread creation – Problem

- What happened?
- Our main thread simply starts the thread `t1` and then immediately exits.
- Whenever the main thread exits all the other threads will be forced to exit as well.
- This causes an exception to be raised, since we haven't specified what should happen if the main thread exits.
- In order to fix this we have to, from the main thread, specify the relationship between our thread and the main thread.

Threads

Thread creation – Fix

examples/1-fixed.cc

```
1  #include <iostream>
2  #include <thread>
3
4  void task()
5  {
6      auto id = std::this_thread::get_id();
7      std::cout << "thread " << id << " created!\n";
8  }
9
10 int main()
11 {
12     std::thread t1 { task };
13     t1.join();
14 }
```

Threads

Thread creation – Fixed

```
$ g++ 1-fixed.cc -lpthread  
$ ./a.out  
thread 140072753923840 created!
```

Threads

Thread creation – Fixed

- There are two ways a thread can be related to its parent thread (i.e. the thread that created it).
- A thread can be *joined* with its parent thread. Then the parent thread is guaranteed to not exit until all of its joined threads have finished execution. This is done by calling `.join()` on the created thread from the parent thread. If the created thread hasn't finished its execution the parent thread will pause and wait for the created thread when `.join()` is called.
- A thread can also be *detached* from its parent. This means that the created thread should immediately exit once the parent thread exits. When detaching a thread from its parent we specify that it is perfectly OK and safe to prematurely exit the created thread. This relationship is established by calling `.detach()` on the created thread *anywhere* in the parent thread.
- Note that joining a thread and dispatching a thread is mutually exclusive. Once you have established a relationship between parent and child you cannot change it.

Threads

Multiple threads

- We can of course create more than one thread, which will be demonstrated on the next slide.
- Note that we have to specify a relationship with *each* of the created threads.
- Another interesting thing that will be demonstrated on the next slide is how to give the thread access to data through the use of parameters.

Threads

Multiple threads

examples/2.cc

```
1 void task(std::string const& name)
2 {
3     std::cout << name << std::endl;
4 }
5
6 int main()
7 {
8     std::string a { "A" };
9     std::string b { "B" };
10
11     std::thread t1 { task, std::cref(a) };
12     std::thread t2 { task, std::cref(b) };
13
14     t1.join();
15     t2.join();
16 }
```

Threads

Multiple threads – Parameters

- On the previous slide we passed additional parameters to the `std::thread` constructor.
- The first parameter is the function that the thread will execute, and any subsequent parameters are passed along to the function, *by-value* as default.
- We wrap our parameters in `std::cref()` to ensure that they instead are passed *by-reference* thus making data shared between the main-thread and the child-thread.

Threads

Multiple threads – Different outputs

```
$ ./a.out
```

```
A
```

```
B
```

```
$ ./a.out
```

```
B
```

```
A
```

```
$ ./a.out
```

```
AB
```

```
$ ./a.out
```

```
BA
```

Threads

Multiple threads – Why different outputs?

- Both threads are executed in parallel, independent of each other. It is only guarantees that operations *within* a thread is executed in order. However, the order of operations *between* threads is non-deterministic since the threads execute at their respective pace completely oblivious to the other threads existence. Meaning that the operations from the two threads get “mixed” in a non-deterministic order. One such “mix” of operations is called an *interleaving*. When writing multithreaded code you have to assume that *any* interleaving between threads can occur, so you must make sure that the result will be correct for *each* possible interleaving.
- `std::cout` is *shared* between all threads. Calling `operator<<` on `std::cout` is *thread-safe*, meaning it is completely safe to have multiple threads call it at the same time. However, the interleaving between chained `operator<<` calls is not deterministic, which means that the order of prints can occur in different order between different interleavings.
- Because of this we get four potential outputs.

Threads

Mutliple threads – Possible interleavings

```
std::cout << "A";  
std::cout << std::endl;
```

```
std::cout << "B";  
std::cout << std::endl;
```

Threads

Multiple threads – Possible interleavings

Interleaving #1

```
std::cout << "A";  
std::cout << std::endl;  
std::cout << "B";  
std::cout << std::endl;
```

Threads

Multiple threads – Possible interleavings

Interleaving #2

```
std::cout << "A";  
std::cout << std::endl;  
std::cout << "B";  
std::cout << std::endl;
```

Threads

Multiple threads – Possible interleavings

Interleaving #3

```
std::cout << "A";  
std::cout << std::endl;  
std::cout << "B";  
std::cout << std::endl;
```

Threads

Multiple threads – Possible interleavings

Interleaving #4

```
std::cout << "A";  
std::cout << std::endl;  
  
std::cout << "B";  
std::cout << std::endl;
```

Threads

Multiple threads – Possible interleavings

Interleaving #5

```
std::cout << "A";  
std::cout << std::endl;  
std::cout << "B";  
std::cout << std::endl;
```

Threads

Multiple threads – Possible interleavings

Interleaving #6

```
std::cout << "A";  
std::cout << std::endl;  
std::cout << "B";  
std::cout << std::endl;
```

Threads

An important detail

- Note that if we perform an operation from two threads simultaneously that is *not* thread-safe, we can no longer assume that threads are interleaved.
- This is because simultaneously performing non-thread-safe operations will result in *race conditions*, where the behaviour of the program is dependent on the order it is called from different threads, which cannot be deterministic.
- If the operation is thread-safe however, then the order of threads performing it should not affect the behaviour. Therefore we can *think* of operations as happening at distinct time steps (i.e. as interleavings) because them occurring at the same time will have the same effect as them happening in sequence.
- But, if this assumption is broken your program lands in undefined behaviour which can have bad consequences.
- For more information you can look at resources related to *sequential consistency* and *data races*.

Threads

Notes about threading

- Threading isn't free
- Hardware support required
- Fallbacks to sharing the CPU

Threads

Notes about threading

- Threading isn't free
 - Creating a thread has a steep cost
 - Managing and synchronizing threads is expensive
- Hardware support required
- Fallbacks to sharing the CPU

Threads

Notes about threading

- Threading isn't free
- Hardware support required
 - Threading is generally only an improvement *if* your hardware has multiple CPU:s.
 - To make multi-threading efficient one should usually only use the same number of threads as your hardware supports.
 - Use `std::thread::hardware_concurrency()` to see how many threads the current system supports.
- Fallbacks to sharing the CPU

Threads

Notes about threading

- Threading isn't free
- Hardware support required
- Fallbacks to sharing the CPU
 - If you are using more threads than your system supports, the excess threads will be scheduled on the same CPU as other threads.
 - So you are still executing code sequentially, but concurrently. I.e. threads are sporadically pausing and starting on CPUs.
 - This will generally not improve the execution time and is therefore not an efficient solution (in terms of performance).

- 1 Threads
- 2 **Synchronization**
- 3 Sharing data
- 4 Thread communication
- 5 Parallel algorithms
- 6 Thinking about parallelism

Synchronization

Solving part of the problem

- The issue with these interleavings is that in some of them there are print statements from one thread *between* two chained `operator<<` calls within another thread, which will mess up the message printed from both threads.
- To solve this we need to ensure that only *one* thread at a time is allowed to print a message.
- I.e. we want to give a thread exclusive access to `std::cout`. We do this by utilizing a concept called *mutual exclusion*.

Synchronization

Mutual exclusion

examples/3.cc

```
1 void task(std::string const& name, std::mutex& mutex)
2 {
3     mutex.lock();
4     std::cout << name << std::endl;
5     mutex.unlock();
6 }
7
8 int main()
9 {
10     std::string a { "A" };
11     std::string b { "B" };
12
13     std::mutex mutex { };
14     std::thread t1 { task, std::cref(a), std::ref(mutex) };
15     std::thread t2 { task, std::cref(b), std::ref(mutex) };
16
17     t1.join();
18     t2.join();
19 }
```

Synchronization

Mutual exclusion – Different outputs

```
$ ./a.out  
A  
B
```

```
$ ./a.out  
B  
A
```

Synchronization

Mutual exclusion – Explanation

- Mutual exclusion is done in C++ using a `std::mutex` object.
- A `std::mutex` acts as a *barrier*. The first thread to reach the barrier (by calling `.lock()`) is allowed entry, but any other threads that arrive after that (i.e. by also calling `.lock()`) must wait until the first thread no longer need exclusive access (by calling `.unlock()`).
- In the example on the previous slide we ensure that the message printed from each thread is guaranteed to be uninterrupted by the other thread, since the `mutex` guarantees that only one thread can execute the print-statement in `task()` at a time.
- However, it is still non-deterministic which thread locks the mutex first, so we still have two different interleavings.
- Note that the synchronization of `std::mutex` is entirely dependent on the `.lock()` and `.unlock()` which means that it can be used to synchronize threads that have completely different tasks.

Synchronization

examples/4.cc

```
1  std::mutex cout_mutex { };
2
3  void task_A()
4  {
5      auto id = std::this_thread::get_id();
6      cout_mutex.lock();
7      std::cout << "My ID: " << id << std::endl;
8      cout_mutex.unlock();
9  }
10
11 void task_B(std::string const& message)
12 {
13     cout_mutex.lock();
14     std::cout << message << std::endl;
15     cout_mutex.unlock();
16 }
17
18 int main()
19 {
20     std::thread t1 { task_A };
21     std::thread t2 { task_B, "A message from t2" };
22
23     cout_mutex.lock();
24     std::cout << "Starting!" << std::endl;
25     cout_mutex.unlock();
26
27     t1.join();
28     t2.join();
29 }
```

Synchronization

Multiple critical sections

- Generally we strive after associating one mutex with each *shared* resource, so that whenever a thread uses that resource it locks the mutex and unlocks once it is finished. Thus ensuring that no two-threads try to use the resource at the same time. Note that a resource can be a group of variables or similar constructs, it isn't necessarily a single variable.
- However, note that one can have multiple different mutexes associated with different resources that are independent of each other.
- A piece of code that needs to be locked with *at least* one mutex, due to it having to use some resource(s), we generally call a *critical section*.
- Critical section A can never be executed at the same time as critical section B **if and only if** A and B have overlapping resources they both use.
- This is why we need one mutex *per* resource, so that critical sections can gain exclusive access to *all* resources it needs, no more and no less.
- Next, we show a situation where we might need to lock multiple mutexes for the same critical section and some problems that can arise from it.

Synchronization

examples/5.cc

```
1 void task(std::mutex& mutex_A, std::mutex& mutex_B)
2 {
3     mutex_A.lock();
4     mutex_B.lock();
5
6     auto id = std::this_thread::get_id();
7     std::cout << id << std::endl;
8
9     mutex_B.unlock();
10    mutex_A.unlock();
11 }
12
13 int main()
14 {
15     std::mutex mutex_A { };
16     std::mutex mutex_B { };
17
18     std::thread t1 { task, std::ref(mutex_A), std::ref(mutex_B) };
19     std::thread t2 { task, std::ref(mutex_B), std::ref(mutex_A) };
20
21     t1.join();
22     t2.join();
23 }
```

Synchronization

examples/5.cc

```
1 void task(std::mutex& mutex_A, std::mutex& mutex_B)
2 {
3     mutex_A.lock();
4     mutex_B.lock();
5
6     auto id = std::this_thread::get_id();
7     std::cout << id << std::endl;
8
9     mutex_B.unlock();
10    mutex_A.unlock();
11 }
12
13 int main()
14 {
15     std::mutex mutex_A { };
16     std::mutex mutex_B { };
17
18     std::thread t1 { task, std::ref(mutex_A), std::ref(mutex_B) };
19     std::thread t2 { task, std::ref(mutex_B), std::ref(mutex_A) };
20
21     t1.join();
22     t2.join();
23 }
```

Synchronization

Deadlocks

Mutex A

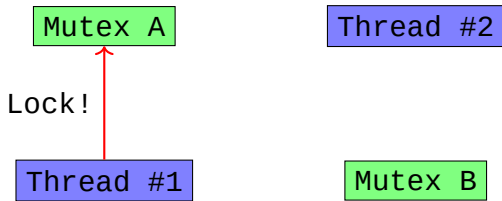
Thread #2

Thread #1

Mutex B

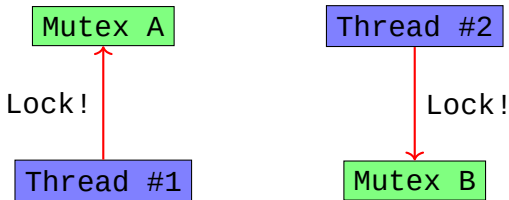
Synchronization

Deadlocks



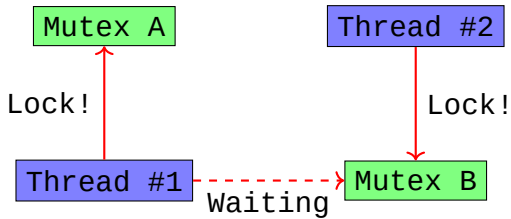
Synchronization

Deadlocks



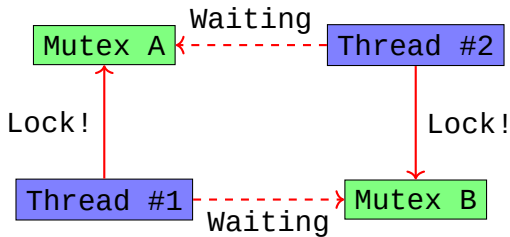
Synchronization

Deadlocks



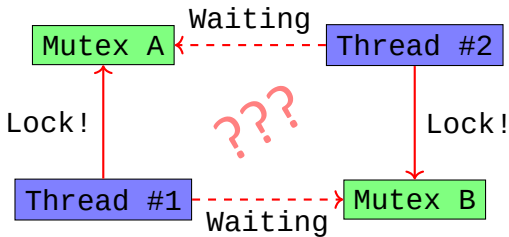
Synchronization

Deadlocks



Synchronization

Deadlocks



Synchronization

Avoid deadlocks

- A solution for avoiding deadlocks is to ensure that mutexes are locked in a *consistent* order.
- What this means is that if mutex A is locked before mutex B in one critical section in one thread, then it has to be locked in the same order in *every* thread and critical section that needs to use A and B.
- This can be solved by using the `std::lock()` function from C++11, or even better: the `std::scoped_lock` class from C++17. Both of these ensure that all passed mutexes are locked in a consistent order.
- The difference between `std::lock()` and `std::scoped_lock` is that the latter also unlocks all mutex once the end of the surrounding scope is reached.
- The reason why `std::scoped_lock` is preferred is that its abstraction removes the need of explicitly unlocking mutexes, we instead express critical sections using scope, which is more in line with how modern C++ is used. It also avoids problems where we forget to unlock a mutex.

Synchronization

Avoid deadlocks

examples/5-fixed.cc

```
1 void task(std::mutex& mutex_A, std::mutex& mutex_B)
2 {
3     std::scoped_lock lock(mutex_A, mutex_B);
4
5     auto id = std::this_thread::get_id();
6     std::cout << id << std::endl;
7 }
8
9 int main()
10 {
11     std::mutex mutex_A { };
12     std::mutex mutex_B { };
13
14     std::thread t1 { task, std::ref(mutex_A), std::ref(mutex_B) };
15     std::thread t2 { task, std::ref(mutex_B), std::ref(mutex_A) };
16
17     t1.join();
18     t2.join();
19 }
```

- 1 Threads
- 2 Synchronization
- 3 **Sharing data**
- 4 Thread communication
- 5 Parallel algorithms
- 6 Thinking about parallelism

Sharing data

Example #1

```
1 int sum(std::vector<int>& values)
2 {
3     int result { 0 };
4
5     for (int value : values)
6         result += value;
7
8     return result;
9 }
```

Sharing data

Example #1

- Let us take this simple sum-function and try to make it multithreaded.
- There are several issues we need to solve to make this happen.
- Firstly we need to determine how we split up the work among different threads. For this particular function it is quite easy to do. Given N threads, we split up the passed in vector into N equally-sized blocks, and let each thread sum its own block.
- The second thing we must figure out is how to merge the results from each thread into the main thread again. This is actually more delicate than it first seems, because here we have to take into account that we don't actually know the exact order work will be done.
- Let's try with the simplest solution we can come up with: make each thread add their total sum to a shared variable containing the result.

Sharing data

Example #1 – Problem

examples/6.cc

```
1 void task(It begin, It end, int& total)
2 {
3     int result { 0 };
4     for (It it { begin }; it != end; ++it)
5         result += *it;
6     total += result;
7 }
8
9 int sum(std::size_t thread_count, std::vector<int>& values)
10 {
11     std::size_t const size { values.size() / thread_count };
12     std::vector<std::thread> pool { };
13     int result { };
14
15     for (std::size_t i { 0 }; i < thread_count; ++i)
16     {
17         auto begin { values.begin() + size * i };
18         auto end { values.begin() + size * (i + 1) };
19         pool.emplace_back(task, begin, end, std::ref(result));
20     }
21
22     for (std::thread& thread : pool)
23         thread.join();
24
25     return result;
26 }
```

Sharing data

Example #1 – Running the example

```
1 std::vector<int> v(1000000, 1);  
2 int answer { sum(20000, v) };
```

What should the answer be?

Sharing data

Example #1 – Running the example

```
1 std::vector<int> v(1000000, 1);  
2 int answer { sum(20000, v) };
```

What should the answer be? 1000000

Sharing data

Example #1 – Running the example

But what do we get?

Sharing data

Example #1 – Running the example

But what do we get?

```
$ ./a.out  
999850  
$ ./a.out  
999600
```

Sharing data

Example #1 – Why?

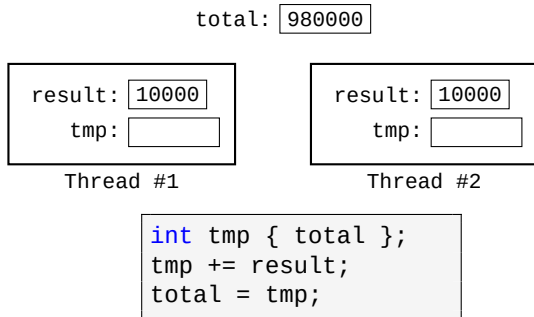
- We do not get consistent results, why?
- The problem is that multiple threads are potentially trying to do `total += result` at the same time.
- Note that `total += result` is equivalent to:

```
1 int tmp { total };  
2 total = tmp + result;
```

Since it first have to read the current value of `total`, then add `result` to it, and then finally overwrite `total` with the new value. Meaning it is actually *three* operations, that can interleave each other from different threads...

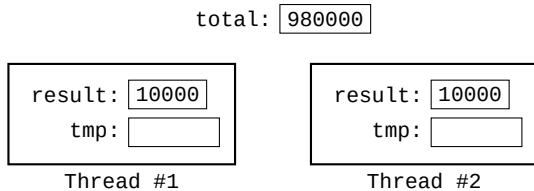
Sharing data

Example #1 – Problem



Sharing data

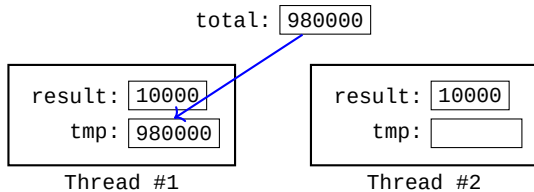
Example #1 – Problem



```
#1> int tmp { total };  
    tmp += result;  
    total = tmp;
```

Sharing data

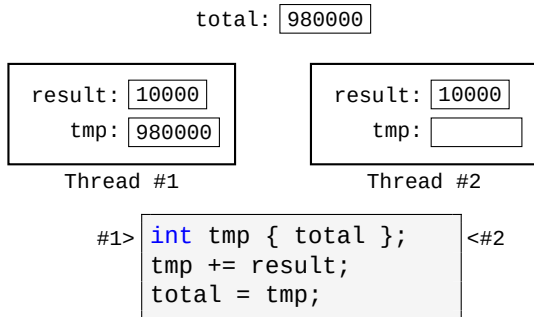
Example #1 – Problem



```
#1> int tmp { total };  
    tmp += result;  
    total = tmp;
```

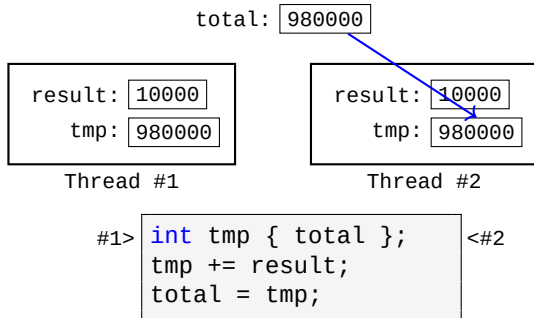
Sharing data

Example #1 – Problem



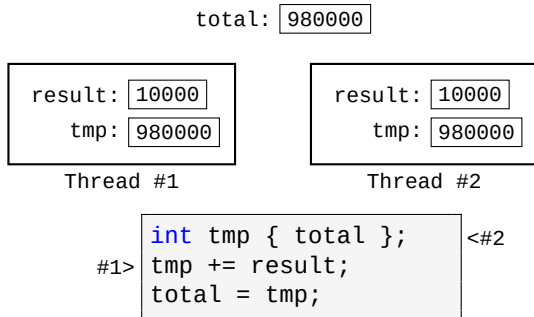
Sharing data

Example #1 – Problem



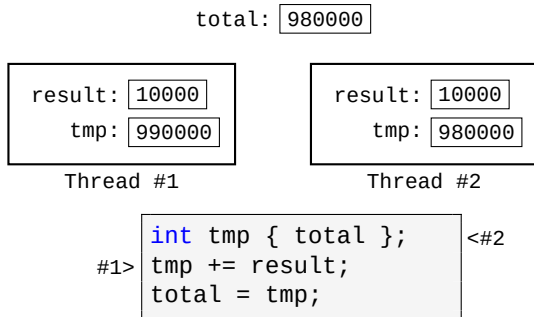
Sharing data

Example #1 – Problem



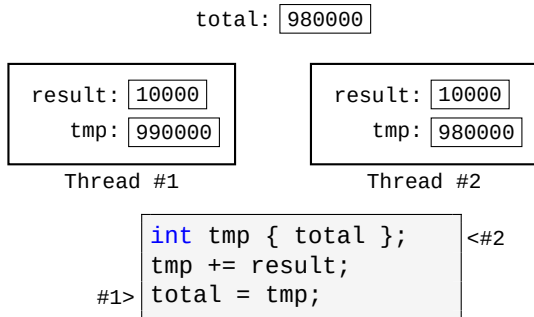
Sharing data

Example #1 – Problem



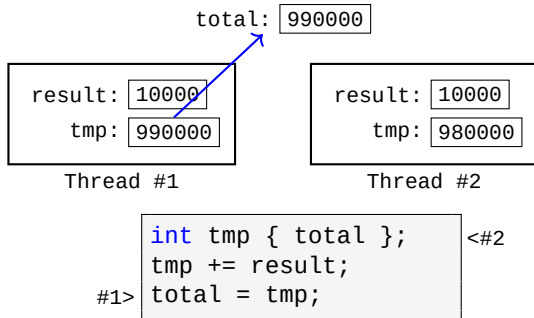
Sharing data

Example #1 – Problem



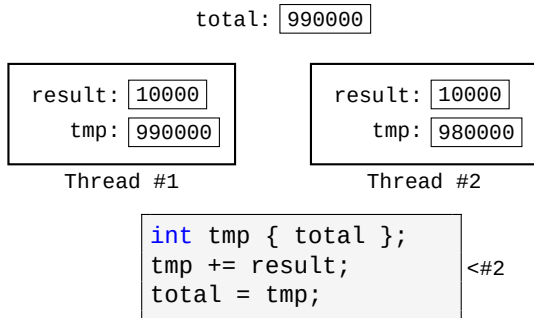
Sharing data

Example #1 – Problem



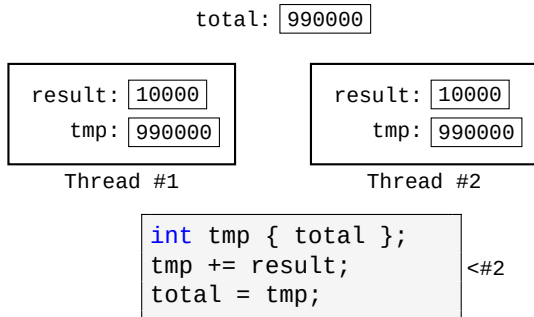
Sharing data

Example #1 – Problem



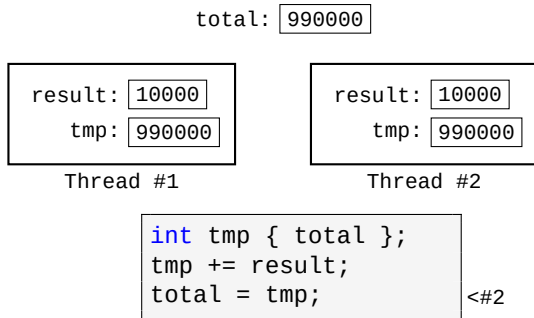
Sharing data

Example #1 – Problem



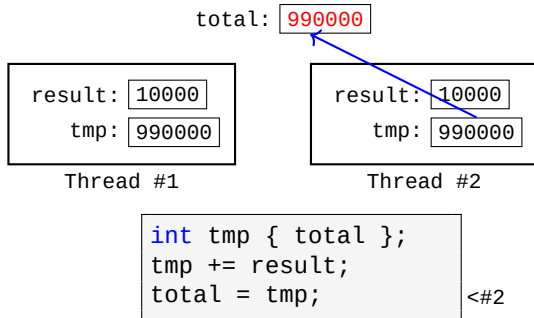
Sharing data

Example #1 – Problem



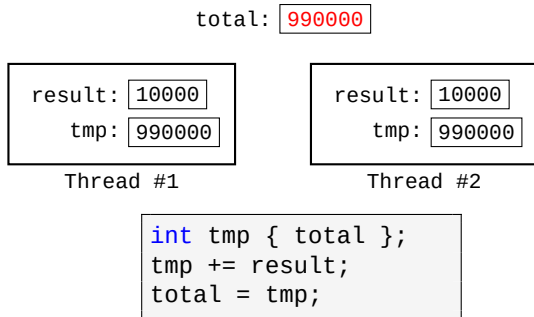
Sharing data

Example #1 – Problem



Sharing data

Example #1 – Problem



Sharing data

Example #1 – Problem

- As is hopefully clear from the animation on the previous slides, the problem arises when two threads happen to read the same initial value from `total`, because then one of them will not take the other into account when calculating the total sum.
- The big problem here is that we have multiple threads that read and write to the same value without any synchronization.
- The easiest solution to this is to make sure that updating the total is a *critical section*, meaning no threads may update the `total` simultaneously.

Sharing data

Example #1 – Fixed with mutex

examples/6-fixed.cc

```
1 void task(It begin, It end, int& total, std::mutex& mutex)
2 {
3     int result { 0 };
4     for (It it { begin }; it != end; ++it)
5         result += *it;
6
7     {
8         std::lock_guard<std::mutex> lock { mutex };
9         total += result;
10    }
11 }
```

Sharing data

Example #1 – Fixed with mutex

- On the previous slide we introduce a `std::mutex` parameter to our `task()` function.
- This mutex is used to lock the update of `total` (think about why it is a bad idea to lock the whole function).
- We are using `std::lock_guard` which is an object that locks the passed in mutex in the constructor, and unlocks it in the destructor. This allows us to use scopes to more clearly mark our critical sections.

Sharing data

Example #1 – Fixed with `std::atomic`

examples/6-atomic.cc

```
1 void task(It begin, It end, std::atomic<int>& total)
2 {
3     int result { 0 };
4     for (It it { begin }; it != end; ++it)
5         result += *it;
6     total += result;
7 }
```

Sharing data

Example #1 – Fixed with `std::atomic`

- We can also achieve the same thing by using an `std::atomic<int>`
- An *atomic type* is a data type where it is well defined what happens if multiple threads attempt to read or write from/to the variable. It specifically ensures that a write is visible to everyone directly, without any intermediate steps.
- Another way to view it is that each operation on an `std::atomic` is *thread-safe*.

- 1 Threads
- 2 Synchronization
- 3 Sharing data
- 4 **Thread communication**
- 5 Parallel algorithms
- 6 Thinking about parallelism

Thread communication

`std::jthread`

```
1 std::thread t {  
2     []()  
3     {  
4         std::cout << "called!" << std::endl;  
5     }  
6 };  
7  
8 t.join();
```

Thread communication

`std::jthread`

```
1 std::jthread t {  
2     []()  
3     {  
4         std::cout << "called!" << std::endl;  
5     }  
6 };
```

Thread communication

`std::jthread`

- `std::jthread` is a version of `std::thread` that automatically joins the thread with the parent in the destructor.
- There are features for prematurely stopping or joining the thread, but in those circumstances it is usually easier to just use `std::thread`.
- `std::jthread` is especially useful to ensure that the thread is handled properly without any actions from the programmer. It is recommended to use `std::jthread` as a default.

Thread communication

Communication between threads

- A common problem in multithreaded software is how to properly and safely communicate between threads.
- We are now going to focus on how to delegate work from one thread to another. The idea is that the parent thread creates a thread and assigns it a task, and then continues doing something else while waiting for the result from its child thread.
- Once the work is ready from the child thread it will be made available to the parent.
- We are going to implement this using a concept called *promise + future*.

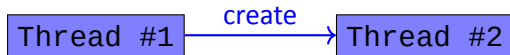
Thread communication

Communication between threads

Thread #1

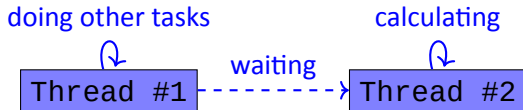
Thread communication

Communication between threads



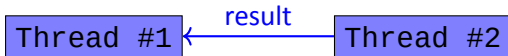
Thread communication

Communication between threads



Thread communication

Communication between threads



Thread communication

Communication between threads

Done!

Thread #1

Thread communication

std::promise + std::future

```
1  std::promise<int> promise { };
2  std::future<int> future { promise.get_future() };
3
4  std::jthread thread {
5      [](std::promise<int> promise)
6      {
7          promise.set_value(5);
8      },
9      std::move(promise)
10 };
11
12 // do other things
13
14 std::cout << future.get() << std::endl;
```

Thread communication

std::promise + std::future

```
1  std::promise<int> promise { };
2  std::future<int> future { promise.get_future() };
3
4  std::jthread thread { Get associated future
5      [](std::promise<int> promise)
6      {
7          promise.set_value(5);
8      },
9      std::move(promise)
10 };
11
12 // do other things
13
14 std::cout << future.get() << std::endl;
```

Thread communication

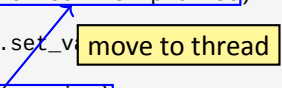
std::promise + std::future

```
1  std::promise<int> promise { };
2  std::future<int> future { promise.get_future() };
3
4  std::jthread thread {
5      [](std::promise<int> promise)
6      {
7          promise.set_value(5);
8      },
9      std::move(promise)
10 };
11
12 // do other things
13
14 std::cout << future.get() << std::endl;
```

Thread communication

std::promise + std::future

```
1  std::promise<int> promise { };
2  std::future<int> future { promise.get_future() };
3
4  std::jthread thread {
5      [(std::promise<int> promise)]
6      {
7          promise.set_value(1);
8      },
9      std::move(promise)
10 };
11
12 // do other things
13
14 std::cout << future.get() << std::endl;
```



Thread communication

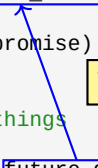
std::promise + std::future

```
1  std::promise<int> promise { };
2  std::future<int> future { promise.get_future() };
3
4  std::jthread thread {
5      [](std::promise<int> promise)
6      {
7          promise.set_value(5);
8      },
9      std::move(promise)
10 };
11
12 // do other things
13
14 std::cout << future.get() << std::endl;
```

Thread communication

std::promise + std::future

```
1  std::promise<int> promise { };
2  std::future<int> future { promise.get_future() };
3
4  std::jthread thread {
5      [](std::promise<int> promise)
6      {
7          promise.set_value(5);
8      },
9      std::move(promise)
10 };
11
12 // do other things
13
14 std::cout << future.get() << std::endl;
```



A blue arrow originates from the `future.get()` call on line 14 and points to the `promise.set_value(5);` call on line 7. This illustrates the flow of data from the promise to the future.

Wait for result from promise

Thread communication

`std::promise + std::future`

- The idea with promise and future is that the `std::promise` is an object that “promises” to deliver a value some time in the future.
- The `std::future` is an object that will, at some point in the “future”, receive the promised value.
- Each `std::promise` can produce *one* `std::future` object (using `.get_future()`) which is where the value will be delivered. Trying to call `.get_future()` more than once is an error.
- If more than one thread need the value, one can construct a `std::shared_future` and pass it to all threads that need it.
- Once the receiving thread needs the value they call the `.get()` function on the `std::future` object. If the value isn’t delivered yet, the thread will at that point wait for it to be ready.
- A `std::future` object can only retrieve the value *once*. Trying to retrieve it more than once is an error.
- Let us now look at a more complicated example.

Thread communication

Example #2 – `std::promise` + `std::future`

examples/7.cc

```
1 void task(std::vector<int>& v, std::promise<int> promise)
2 {
3     int count { std::count_if(v.begin(), v.end(), is_prime) };
4     promise.set_value(count);
5 }
6
7 int main()
8 {
9     std::vector<int> v(1000);
10    std::iota(v.begin(), v.end(), 1);
11
12    std::promise<int> promise { };
13    auto future = promise.get_future();
14
15    std::jthread thread { task, std::ref(v), std::move(promise) };
16
17    std::cout << "The number of primes is: ";
18    std::cout << future.get() << std::endl;
19 }
```

Thread communication

Let's look at the `sum()` example again

Thread communication

Example #1 – Using future + promise and `std::jthread`

examples/8.cc

```
1 void task(It begin, It end, std::promise<int> promise)
2 { promise.set_value(std::reduce(begin, end)); }
3
4 int sum(std::size_t thread_count, std::vector<int> const& values)
5 {
6     std::size_t const size { values.size() / thread_count };
7     std::vector<std::jthread> pool { };
8     std::vector<std::future<int>> futures { };
9
10    for (std::size_t i { 0 }; i < thread_count; ++i)
11    {
12        auto begin { values.begin() + size * i };
13        auto end { values.begin() + size * (i + 1) };
14        std::promise<int> promise { };
15
16        futures.push_back(std::move(promise.get_future()));
17        pool.emplace_back(task, begin, end, std::move(promise));
18    }
19
20    return std::accumulate(futures.begin(), futures.end(), 0,
21                           [](int result, std::future<int>& future)
22                           { return result + future.get(); });
23 }
```

- 1 Threads
- 2 Synchronization
- 3 Sharing data
- 4 Thread communication
- 5 **Parallel algorithms**
- 6 Thinking about parallelism

Parallel algorithms

STL algorithms

- You might have seen that for many of the STL algorithms there are alternative overloads using something called *execution policies*.
- These are algorithms that can be multi-threaded *automatically*.
- The idea is that execution policies communicate to the algorithm certain models for parallelism.

Parallel algorithms

Execution policies

- `std::execution::seq`
- `std::execution::par`
- `std::execution::par_unseq`
- `std::execution::unseq` (C++20)

Parallel algorithms

Execution policies

- `std::execution::seq`
the algorithm is executed on a single thread, in sequence.
- `std::execution::par`
- `std::execution::par_unseq`
- `std::execution::unseq` (C++20)

Parallel algorithms

Execution policies

- `std::execution::seq`
- `std::execution::par`
the algorithm is executed in parallel, but internally in each thread the work is executed in sequence.
- `std::execution::par_unseq`
- `std::execution::unseq` (C++20)

Parallel algorithms

Execution policies

- `std::execution::seq`
- `std::execution::par`
- `std::execution::par_unseq`
the algorithm is executed in parallel, and there are no guarantees what-so-ever regarding which order the operations are performed.
- `std::execution::unseq` (C++20)

Parallel algorithms

Execution policies

- `std::execution::seq`
- `std::execution::par`
- `std::execution::par_unseq`
- `std::execution::unseq` (C++20)
the algorithm is executed on a single thread, but there are no guarantees regarding the order.

Parallel algorithms

Execution policies

- Note that `par_unseq` is generally the most efficient parallelization, but it has weaker guarantees when it comes to the result. If operations being performed are *associative*, i.e. if $(a \oplus b) \oplus c = a \oplus (b \oplus c)$ for some operation \oplus , then the operation can safely be performed unsequenced.
- A typical example of an operation that is **not** associative, is subtraction, since $(1 - 2) - 3 = -4 \neq 1 - (2 - 3) = 2$, so if we are to calculate a difference in an algorithm we have stronger guarantees regarding the result by using `std::execution::par`.

Parallel algorithms

Example #1 – Using execution policy

examples/9.cc

```
1 int sum(std::vector<int> const& values)
2 {
3     return std::reduce(std::execution::par_unseq,
4                       values.begin(), values.end());
5 }
6
7 int main()
8 {
9     std::vector<int> v(1000000, 1);
10    int answer { sum(v) };
11    std::cout << answer << std::endl;
```

Parallel algorithms

Example #2 – Using execution policy

examples/10.cc

```
1  int main()
2  {
3      std::vector<int> v(1000);
4      std::iota(v.begin(), v.end(), 1);
5
6      auto count { std::count_if(std::execution::par_unseq,
7                                v.begin(), v.end(), is_prime) };
8
9      std::cout << "The number of primes is: ";
10     std::cout << count << std::endl;
11 }
```

- 1 Threads
- 2 Synchronization
- 3 Sharing data
- 4 Thread communication
- 5 Parallel algorithms
- 6 Thinking about parallelism

Thinking about parallelism

Amdahl's law

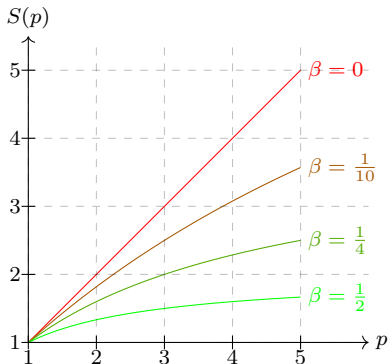
$$S(p) = \frac{p}{\beta p + (1 - \beta)}$$

Where:

- $S(p)$ is the factor speedup of the program, given p threads.
- β is the fraction of the program that *cannot* be parallelized.

Thinking about parallelism

Consequences



Thinking about parallelism

Consequences of Amdahl's law

- What Amdahl basically tells us is that we do not solve efficiency by simply throwing threads at our problem.
- Instead we have to make sure that β is as small as possible. I.e. focus on rewriting your programs so that you minimize the amount of work that *must* be single-threaded, which requires a whole lot of work.
- Some problems *cannot* be parallelized, and in those cases you have to focus on aspects other than parallelism.

Thinking about parallelism

Recommended course

To learn more about this: TDDE65

www.liu.se