Amdahl's law

**n N the number of threads**

**B [0, 1]** the fraction of the algorithm that is strictly serial

The time an algorithm takes to finish when being executed on thread(s) of execution corresponds to

**T(n) = T(1) ( B + 1/n \* (1 - B) )**

Therefore, the theoretical speedup that can be had by executing a given algorithm on a system capable of executing threads of execution is:

S(n) = T(1) / T(n) = T(1) / ( T(1) \* (B + 1/n \* ( 1 – B ) ) =

1 / ( B + 1/n \* ( 1 – B ) )

Пусть необходимо решить некоторую вычислительную задачу. Предположим, что её [алгоритм](/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC) таков, что доля A от общего объёма вычислений может быть получена только последовательными расчётами, а, соответственно, доля 1 - A может быть распараллелена идеально (то есть время вычисления будет обратно пропорционально числу задействованных узлов p). Тогда ускорение, которое может быть получено на вычислительной системе из p процессоров, по сравнению с однопроцессорным решением не будет превышать величины

**Sp = 1 / ( A + ( ( 1 – A ) / p ) )**

**Закон Густафсона** (иногда **Густавсона**) **— Барсиса** ([англ.](/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Gustafson – Barsis's law*) — оценка максимально достижимого ускорения выполнения [параллельной](/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F) программы, в зависимости от количества одновременно выполняемых [потоков вычислений](/wiki/%D0%9F%D0%BE%D1%82%D0%BE%D0%BA_%D0%B2%D1%8B%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D1%8F) («процессоров») и доли последовательных расчётов. Аналог [закона Амдала](/wiki/%D0%97%D0%B0%D0%BA%D0%BE%D0%BD_%D0%90%D0%BC%D0%B4%D0%B0%D0%BB%D0%B0).

Закон Густафсона — Барсиса выражается формулой:

Sp = g + ( 1 – g ) \* p = p + ( 1 – p ) \* g

, где

g — доля последовательных расчётов в программе,

p — количество процессоров (одновременных потоков).

Данную оценку ускорения называют **ускорением масштабирования** ([англ.](/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *scaled speedup*), так как данная характеристика показывает, насколько эффективно могут быть организованы параллельные вычисления при увеличении сложности решаемых задач.

# class std::thread

Member types

[id](/reference/thread/thread/id/) Thread id (public member type )

[native\_handle\_type](/reference/thread/thread/native_handle_type/) Native handle type (public member type )

Member functions[(constructor)](/reference/thread/thread/thread/) Construct thread (public member function )

[(destructor)](/reference/thread/thread/~thread/)Thread destructor (public member function )

[operator=](/reference/thread/thread/operator=/)Move-assign thread (public member function )

[get\_id](/reference/thread/thread/get_id/) Get thread id (public member function )

[joinable](/reference/thread/thread/joinable/) Check if joinable (public member function )

[join](/reference/thread/thread/join/) Join thread (public member function )

[detach](/reference/thread/thread/detach/) Detach thread (public member function )

[swap](/reference/thread/thread/swap/) Swap threads (public member function )

[native\_handle](/reference/thread/thread/native_handle/) Get native handle (public member function )

[hardware\_concurrency [static]](/reference/thread/thread/hardware_concurrency/) Detect hardware concurrency (public static member function )

Non-member overloads [swap (thread)](/reference/thread/thread/swap-free/)Swap threads (function )

// thread example

#include <iostream> // std::cout

#include <thread> // std::thread

void foo()

{

// do stuff...

}

void bar(int x)

{

// do stuff...

}

int main()

{

std::thread first (foo); // spawn new thread that calls foo()

std::thread second (bar,0); // spawn new thread that calls bar(0)

std::cout << "main, foo and bar now execute concurrently...\n";

// synchronize threads:

first.join(); // pauses until first finishes

second.join(); // pauses until second finishes

std::cout << "foo and bar completed.\n";

return 0;

}   
  
Output:

main, foo and bar now execute concurrently...

foo and bar completed.

native\_handle()

Although the C++ Thread Library provides reasonably comprehensive facilities for

multithreading and concurrency, on any given platform there will be platform-specific

facilities that go beyond what’s offered. In order to gain easy access to those facilities

without giving up the benefits of using the Standard C++ Thread Library, the types in

the C++ Thread Library may offer a native\_handle()

join()

joinable()

detach()

you can call

join() only once for a given thread; once you’ve called join(), the std::thread

object is no longer joinable, and joinable() will return false.

Daemon threads - detached threads

**Race condition** a race condition is anything where the outcome depends on the

relative ordering of execution of operations on two or more threads

**Data race** to mean the specific type of race condition that arises because of concurrent

modification to a single object

std::unique\_lock() // adopt\_lock, deffered\_lock

std::call\_once()

std::once\_flag;

std::shared\_ptr<some\_resource> resource\_ptr;

std::once\_flag resource\_flag;

void init\_resource()

{

resource\_ptr.reset(new some\_resource);

}

void foo()

{

std::call\_once(resource\_flag,init\_resource);

resource\_ptr->do\_something();

}

class my\_class;

my\_class& get\_my\_class\_instance()

{

static my\_class instance;

return instance;

}

boost::shared\_mutex;

lock\_guard<boost::shared\_mutex> // for read write problem

<std::recursive\_mutex> and std::unique\_lock<std::recursive\_mutex>

# class condition\_variable;

Condition variable

A condition variable is an object able to block the calling thread until notified to resume.

It uses a **unique\_lock** (over a mutex) to lock the thread when one of its wait functions is called. The thread remains blocked until woken up by another thread that calls a notification function on the same condition\_variable object.

Objects of type condition\_variable always use **unique\_lock<mutex>** to wait: for an alternative that works with any kind of lockable type, see **condition\_variable\_any**

Member functions

(constructor)Construct condition\_variable (public member function )(destructor)Destroy condition\_variable (public member function )

Wait functions

wait Wait until notified (public member function )

wait\_for Wait for timeout or until notified (public member function )

wait\_untilWait until notified or time point (public member function )

Notify functions

notify\_one Notify one (public member function )

notify\_all Notify all (public member function )

Example

// condition\_variable example

#include <iostream> // std::cout

#include <thread> // std::thread

#include <mutex> // std::mutex, std::unique\_lock

#include <condition\_variable> // std::condition\_variable

std::mutex mtx;

std::condition\_variable cv;

bool ready = false;

void print\_id (int id) {

std::unique\_lock<std::mutex> lck(mtx);

while (!ready) cv.wait(lck);

// ...

std::cout << "thread " << id << '\n';

}

void go() {

std::unique\_lock<std::mutex> lck(mtx);

ready = true;

cv.notify\_all();

}

int main ()

{

std::thread threads[10];

// spawn 10 threads:

for (int i=0; i<10; ++i)

threads[i] = std::thread(print\_id,i);

std::cout << "10 threads ready to race...\n";

go(); // go!

for (auto& th : threads) th.join();

return 0;

}

Possible output (thread order may vary):

10 threads ready to race...

thread 2

thread 0

thread 9

thread 4

thread 6

thread 8

thread 7

thread 5

thread 3

thread 1

// Thread run

void do\_some\_work()

std::thread thr( do\_some\_work )

// with functor

class DoSomeWork

{

public:

void operator()() const

{

do\_something();

do\_something\_else();

}

};

DoSomeWork do;

std::thread thr( do ); // copy to memory

C++ most vexing parse

**Std::thread thr( do() ); compiler interpret this construction as a function declaration with no parameters which returns DoSomeWork object and NO RUN of thread**

There are two methods to solve this problem

std::thread thr( ( DoSomeWork() ) );

Or for C++11

std::thread thr{ DoSomeWork() };

Or with lambda function

std::thread thr( []{ do\_something(); do\_something\_else(); });

After creation detach or join. Remember dangling references. If not when then destructor std::thread call std::terminate() and program will be terminated.

It is better to copy data to thread function not share.

Joining

thr.join() The act of calling join() cleans up any storage associated with the thread, so the std::thread object is no longer associated with the now finished thread; it isn’t associated with any thread. This means that you can call join() only once for a given thread; once you’ve called join(), the std::thread object is no longer joinable, and joinable() will return false.

Detaching

thr.detach();

call thr.join() in catch block too

struct func;

void f()

{

int some\_local\_state = 0;

func my\_func( some\_local\_state );

std::thread t( my\_func );

try

{

do\_something\_in\_current\_thread();

}

catch( ... )

{

t.join();

throw;

}

t.join();

}

Or employ RAII

class thread\_guard

{

std::thread& t;

public:

explicit thread\_guard( std::thread& t\_ ): t( t\_ ){ }

~thread\_guard()

{

// object is joinable() before calling join().

// This is important, because join()

// can be called only once for a given thread of execution,

// so it would therefore be a mistake

// to do so if the thread had already been joined.

if( t.joinable() ) //< tests to see if the std::thread

{

t.join();

}

}

thread\_guard(thread\_guard const&)=delete;

thread\_guard& operator=(thread\_guard const&)=delete;

};

struct func;

void f()

{

int some\_local\_state = 0;

func my\_func( some\_local\_state );

std::thread t( my\_func );

thread\_guard g( t );

do\_something\_in\_current\_thread( );

}

If thread is detached we don’t worry about exceptions. Detached threads work in background no means to communicate with. They are thread *daemons like process daemons.*

std::thread t( do\_background\_work );

t.detach( );

assert( !t.joinable() );

// when arguments passed to thread function

// they copy even if they passed by ref

void f(int i,std::string const& s);

std::thread t(f,3,”hello”);

This creates a new thread of execution associated with t, which calls f(3,”hello”). Note that even though f takes a std::string as the second parameter, the string literal is passed as a char const\* and converted to a std::string only in the context of

the new thread. This is particularly important when the argument supplied is a

pointer to an automatic variable, as follows:

void f(int i,std::string const& s);

void oops(int some\_param)

{

char buffer[1024];

sprintf(buffer, "%i",some\_param);

**std::thread t(f,3,buffer);**

t.detach();

}

In this case, it’s the pointer to the local variable buffer B that’s passed through to the

new thread c, and there’s a significant chance that the function oops will exit before

the buffer has been converted to a std::string on the new thread, thus leading to

undefined behavior. The solution is to cast to std::string *before* passing the buffer

to the std::thread constructor:

void f(int i,std::string const& s);

void not\_oops(int some\_param)

{

char buffer[1024];

sprintf(buffer,"%i",some\_param);

**std::thread t(f,3,std::string(buffer));**

t.detach();

}

It’s also possible to get the reverse scenario: the object is copied, and what you

wanted was a reference. This might happen if the thread is updating a data structure

that’s passed in by reference, for example:

void update\_data\_for\_widget(widget\_id w,widget\_data& data);

void oops\_again(widget\_id w)

{

widget\_data data;

std::thread t(update\_data\_for\_widget,w,data);

display\_status();

t.join();

process\_widget\_data(data);

}

Although update\_data\_for\_widget B expects the second parameter to be passed by

reference, the std::thread constructor c doesn’t know that; it’s oblivious to the

types of the arguments expected by the function and blindly copies the supplied values.

When it calls update\_data\_for\_widget, it will end up passing a reference to

the internal copy of data and not a reference to data itself. Consequently, when the

thread finishes, these updates will be discarded as the internal copies of the supplied

arguments are destroyed, and process\_widget\_data will be passed an unchanged

data d rather than a correctly updated version. For those of you familiar with

std::bind, the solution will be readily apparent: you need to wrap the arguments that

really need to be references in std::ref. In this case, if you change the thread invocation

to

std::thread t(update\_data\_for\_widget,w,**std::ref(data)**);

you can pass a member function pointer as the function, provided you supply

a suitable object pointer as the first argument:

class X

{

public:

void do\_lengthy\_work();

};

X my\_x;

std::thread t(&X::do\_lengthy\_work,&my\_x);

The *move*

*constructor* and *move assignment operator* allow the ownership of an object to be transferred

around between std::unique\_ptr instances (see appendix A, section A.1.1, for

more on move semantics). Such a transfer leaves the source object with a NULL

pointer. This moving of values allows objects of this type to be accepted as function

parameters or returned from functions. Where the source object is a temporary, the

move is automatic, but where the source is a named value, the transfer must be

requested directly by invoking std::move(). The following example shows the use of

std::move to transfer ownership of a dynamic object into a thread:

void process\_big\_object(std::unique\_ptr<big\_object>);

std::unique\_ptr<big\_object> p(new big\_object);

p->prepare\_data(42);

std::thread t(process\_big\_object,std::move(p));

By specifying std::move(p) in the std::thread constructor, the ownership of the

big\_object is transferred first into internal storage for the newly created thread and

then into process\_big\_object.

std::thread

are *movable*, even though they aren’t *copyable*. This ensures that only one object is associated

with a particular thread of execution at any one time while allowing programmers

the option of transferring that ownership between objects.

The example

**26** CHAPTER 2 ***Managing threads***

shows the creation of two threads of execution and the transfer of ownership of those

threads among three std::thread instances, t1, t2, and t3:

void some\_function();

void some\_other\_function();

std::thread t1(some\_function);

std::thread t2=std::move(t1);

t1=std::thread(some\_other\_function);

std::thread t3;

t3=std::move(t2);

t1=std::move(t3); // transfers ownership of the thread running some\_function back

to t1 where it started. But in this case t1 already had an associated thread (which was

running some\_other\_function), so std::terminate() is called to terminate the

program.

if ownership should be transferred into a function, it can just accept an

instance of std::thread by value as one of the parameters, as shown here:

void f(std::thread t);

One benefit of the move support of std::thread is that you can build on the

thread\_guard class from listing 2.3 and have it actually take ownership of the thread.

This avoids any unpleasant consequences should the thread\_guard object outlive the

thread it was referencing, and it also means that no one else can join or detach

the thread once ownership has been transferred into the object. Because this would

primarily be aimed at ensuring threads are completed before a scope is exited, I named

this class scoped\_thread. The implementation is shown in the following listing, along

with a simple example.

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 const&)=delete;

scoped\_thread& operator=(scoped\_thread const&)=delete;

};

struct func;

void f()

{

int some\_local\_state;

scoped\_thread t(std::thread(func(some\_local\_state)));

do\_something\_in\_current\_thread();

}

The move support in std::thread also allows for containers of std::thread

objects, if those containers are move aware (like the updated std::vector<>). This

means that you can write code like that in the following listing, which spawns a number

of threads and then waits for them to finish.

void do\_work(unsigned id);

void f()

{

std::vector<std::thread> threads;

for(unsigned i=0;i<20;++i)

{

threads.push\_back(std::thread(do\_work,i));

}

std::for\_each(threads.begin(),threads.end(),

std::mem\_fn(&std::thread::join));

}

One feature of the C++ Standard Library that helps here is std::thread::hardware\_

concurrency(). This function returns an indication of the number of threads that can

truly run concurrently for a given execution of a program. On a multicore system it

might be the number of CPU cores, for example.

unsigned long const num\_threads=

std::min(hardware\_threads!=0?hardware\_threads:2,max\_threads);

the identifier for a thread can be obtained from its associated std::thread

object by calling the get\_id() member function. If the std::thread object doesn’t

have an associated thread of execution, the call to get\_id() returns a defaultconstructed

std::thread::id object, which indicates “not any thread.”

This allows them

to be used as keys in associative containers, or sorted, or compared in any other way

that you as a programmer may see fit.

The Standard Library also provides

std::hash<std::thread::id> so that values of type std::thread::id can be used as

keys in the new unordered associative containers too.

You can

even write out an instance of std::thread::id to an output stream such as std::cout:

std::cout<<std::this\_thread::get\_id();

Share data between threads

Break invarianat (e.g. double linked list)

In concurrency, a race condition is anything where the outcome depends on the

relative ordering of execution of operations on two or more threads

Another way of dealing with race conditions is to handle the updates to the data

structure as a *transaction*, just as updates to a database are done within a transaction.

The required series of data modifications and reads is stored in a transaction log and

then committed in a single step. If the commit can’t proceed because the data structure

has been modified by another thread, the transaction is restarted. This is termed

*software transactional memory (STM)*,

Mutex lock unlock lock\_quard

#include <list>

#include <mutex>

#include <algorithm>

std::list<int> some\_list;

std::mutex some\_mutex;

void add\_to\_list(int new\_value)

{

std::lock\_guard<std::mutex> guard(some\_mutex);

some\_list.push\_back(new\_value);

}

bool list\_contains(int value\_to\_find)

{

std::lock\_guard<std::mutex> guard(some\_mutex);

return std::find(some\_list.begin(),some\_list.end(),value\_to\_find)

!= some\_list.end();

}

if one of the

member functions returns a pointer or reference to the protected data, then it

doesn’t matter that the member functions all lock the mutex in a nice orderly fashion,

because you’ve just blown a big hole in the protection. *Any code that has access to that*

*pointer or reference can now access (and potentially modify) the protected data without locking the*

*mutex.*

As you’ve just seen, protecting data with a mutex is not quite as easy as just slapping a

std::lock\_guard object in every member function; one stray pointer or reference, and

all that protection is for nothing. At one level, checking for stray pointers or references is

easy; as long as none of the member functions return a pointer or reference to the protected

data to their caller either via their return value or via an out parameter, the data is

safe. If you dig a little deeper, it’s not that straightforward—nothing ever is. As well as

checking that the member functions don’t pass out pointers or references to their callers,

it’s also important to check that they don’t pass such pointers or references *in* to functions

they call that aren’t under your control. This is just as dangerous: those functions

might store the pointer or reference in a place where it can later be used without the protection

of the mutex. Particularly dangerous in this regard are functions that are supplied

at runtime via a function argument or other means, as in the next listing.

class some\_data

{

int a;

std::string b;

public:

void do\_something();

};

class data\_wrapper

{

private:

some\_data data;

std::mutex m;

public:

template<typename Function>

void process\_data(Function func)

{

std::lock\_guard<std::mutex> l(m);

func(data);

}

};

Listing 3.2 Accidentally passing out a reference to protected data

some\_data\* unprotected;

void malicious\_function(some\_data& protected\_data)

{

unprotected=&protected\_data;

}

data\_wrapper x;

void foo()

{

x.process\_data(malicious\_function); // Pass “protected” data to

//user-supplied function

unprotected->do\_something(); // Unprotected access to protected data

}

***Don’t pass pointers and references to protected data outside the scope of the lock, whether by***

***returning them from a function, storing them in externally visible memory, or passing them as***

***arguments to user-supplied functions.***

*As*

Even though you can detect at compile time

the existence of a copy or move constructor that doesn’t throw an exception using the

std::is\_nothrow\_copy\_constructible and std::is\_nothrow\_move\_constructible

type traits, it’s quite limiting.

*Threadsafe stack*

#include <exception>

#include <memory>

struct empty\_stack: std::exception

{

const char\* what() const throw();

};

template<typename T>

class threadsafe\_stack

{

public:

threadsafe\_stack();

threadsafe\_stack(const threadsafe\_stack&);

threadsafe\_stack& operator=(const threadsafe\_stack&) = delete;

void push(T new\_value);

std::shared\_ptr<T> pop();

void pop(T& value);

bool empty() const;

};

following listing shows a simple implementation that’s a wrapper around

std::stack<>.

#include <exception>

#include <memory>

#include <mutex>

#include <stack>

struct empty\_stack: std::exception

{

const char\* what() const throw();

};

A fleshed-out class definition for a thread-safe stack

Implementation on std::stack<>

#include <exception>

#include <memory>

#include <mutex>

#include <stack>

struct empty\_stack: std::exception

{

const char\* what() const throw();

};

template<typename T>

class threadsafe\_stack

{

private:

std::stack<T> data;

mutable std::mutex m;

public:

threadsafe\_stack(){}

threadsafe\_stack(const threadsafe\_stack& other)

{

std::lock\_guard<std::mutex> lock(other.m);

data=other.data; // **Copy performed in**

**constructor body**

}

threadsafe\_stack& operator=(const threadsafe\_stack&) = delete;

void push(T new\_value)

{

std::lock\_guard<std::mutex> lock(m);

data.push(new\_value);

}

std::shared\_ptr<T> pop()

{

std::lock\_guard<std::mutex> lock(m);

if(data.empty()) throw empty\_stack(); // Check for empty before

trying to pop value

std::shared\_ptr<T> const res(std::make\_shared<T>(data.top())); // Allocate return

value before

modifying stack

data.pop();

return res;

}

void pop(T& value)

{

std::lock\_guard<std::mutex> lock(m);

if(data.empty()) throw empty\_stack();

value=data.top();

data.pop();

}

bool empty() const

{

std::lock\_guard<std::mutex> lock(m);

return d

Deadlock

threads arguing

over locks on mutexes: each of a pair of threads needs to lock both of a pair of

mutexes to perform some operation, and each thread has one mutex and is waiting

for the other. Neither thread can proceed, because each is waiting for the other to

release its mutex. This scenario is called *deadlock*, and it’s the biggest problem with

having to lock two or more mutexes in order to perform an operation.

The common advice for avoiding deadlock is to always lock the two mutexes in the

same order: if you always lock mutex A before mutex B, then you’ll never deadlock.

Thankfully, the C++ Standard Library has a cure for this in the form of std::lock—

a function that can lock two or more mutexes at once without risk of deadlock.

Using **std::lock()** and **std::lock\_guard** in a swap operation

class some\_big\_object;

void swap(some\_big\_object& lhs,some\_big\_object& rhs);

class X

{

private:

some\_big\_object some\_detail;

std::mutex m;

public:

X(some\_big\_object const& sd):some\_detail(sd){}

friend void swap(X& lhs, X& rhs)

{

if(&lhs==&rhs)

return;

std::lock(lhs.m,rhs.m);

std::lock\_guard<std::mutex> lock\_a(lhs.m,std::adopt\_lock);

std::lock\_guard<std::mutex> lock\_b(rhs.m,std::adopt\_lock);

swap(lhs.some\_detail,rhs.some\_detail);

}

};

The std::adopt\_lock parameter is supplied in

addition to the mutex to indicate to the std::lock\_guard objects that the mutexes

are already locked, and they should just adopt the ownership of the existing lock on

the mutex rather than attempt to lock the mutex in the constructor. This ensures that the mutexes are correctly unlocked on function exit in the general

case where the protected operation might throw an exception; it also allows for a

simple return.

Although std::lock can help you avoid deadlock in those cases where you need to

acquire two or more locks together, it doesn’t help if they’re acquired separately. In

that case you have to rely on your discipline as developers to ensure you don’t get

deadlock. This isn’t easy: deadlocks are one of the nastiest problems to encounter in

multithreaded code and are often unpredictable, with everything working fine the

majority of the time. There are, however, some relatively simple rules that can help

you to write deadlock-free code.

AVOID NESTED LOCKS

AVOID CALLING USER-SUPPLIED CODE WHILE HOLDING A LOCK

ACQUIRE LOCKS IN A FIXED ORDER

USE A LOCK HIERARCHY

Using a lock hierarchy to prevent deadlock

hierarchical\_mutex high\_level\_mutex(10000);

hierarchical\_mutex low\_level\_mutex(5000);

int do\_low\_level\_stuff();

int low\_level\_func()

{

std::lock\_guard<hierarchical\_mutex> lk(low\_level\_mutex);

return do\_low\_level\_stuff();

}

void high\_level\_stuff(int some\_param);

void high\_level\_func()

{

std::lock\_guard<hierarchical\_mutex> lk(high\_level\_mutex);

high\_level\_stuff(low\_level\_func());

}

void thread\_a()

{

high\_level\_func();

}

hierarchical\_mutex other\_mutex(100);

void do\_other\_stuff();

void other\_stuff()

{

high\_level\_func();

do\_other\_stuff();

}

void thread\_b()

{

std::lock\_guard<hierarchical\_mutex> lk(other\_mutex);

other\_stuff();

}

std::unique\_lock

provides a bit more flexibility than std::lock\_guard by relaxing

the invariants; a std::unique\_lock instance doesn’t always own the mutex that it’s

associated with.

std::adopt\_lock

std::defer\_lock

The lock can then be acquired later by calling

lock() on the std::unique\_lock object (*not* the mutex) or by passing the std::

unique\_lock object itself to std::lock().

std::unique\_lock

takes more space and is a fraction slower to use than std::lock\_guard

class some\_big\_object;

void swap(some\_big\_object& lhs,some\_big\_object& rhs);

Listing 3.9 Using **std::lock()** and **std::unique\_lock** in a swap operation

class X

{

private:

some\_big\_object some\_detail;

std::mutex m;

public:

X(some\_big\_object const& sd):some\_detail(sd){}

friend void swap(X& lhs, X& rhs)

{

if(&lhs==&rhs)

return;

**// std::defer\_lock leaves mutexes unlocked**

**std::unique\_lock<std::mutex> lock\_a(lhs.m,std::defer\_lock);**

**std::unique\_lock<std::mutex> lock\_b(rhs.m,std::defer\_lock);**

**std::lock(lock\_a,lock\_b);**

swap(lhs.some\_detail,rhs.some\_detail);

}

};

***Transferring mutex ownership between scopes***

One possible use is to allow a function to lock a mutex and transfer ownership of

that lock to the caller, so the caller can then perform additional actions under the protection

of the same lock. The following code snippet shows an example of this: the

function get\_lock() locks the mutex and then prepares the data before returning

the lock to the caller:

std::unique\_lock<std::mutex> get\_lock()

{

extern std::mutex some\_mutex;

std::unique\_lock<std::mutex> lk(some\_mutex);

prepare\_data();

return lk;

}

void process\_data()

{

std::unique\_lock<std::mutex> lk(get\_lock());

do\_something();

}

The flexibility of std::unique\_lock also allows instances to relinquish their locks

before they’re destroyed. You can do this with the unlock() member function, just

like for a mutex: std::unique\_lock supports the same basic set of member functions

for locking and unlocking as a mutex does, in order that it can be used with generic

functions such as std::lock. The ability to release a lock before the std::unique\_

lock instance is destroyed means that you can optionally release it in a specific code

branch if it’s apparent that the lock is no longer required.

***Locking at an appropriate granularity***

std::unique\_lock works well in situation, because you can call unlock()

when the code no longer needs access to the shared data and then call lock() again if

access is required later in the code:

void get\_and\_process\_data()

{

std::unique\_lock<std::mutex> my\_lock(the\_mutex);

some\_class data\_to\_process=get\_next\_data\_chunk();

// You don’t need the mutex locked across the call to process(), so you manually

unlock it before the call and then lock it again afterward

my\_lock.unlock(); // **Don’t need mutex locked across call to process()**

result\_type result=process(data\_to\_process); // Very long operation e.g. file input output

my\_lock.lock();**Relock mutex**

c **to write result**

***Protecting shared data during initialization***

Infamous *Double-Checked Locking* pattern: the pointer is first read without acquiring

the lock B (in the code below), and the lock is acquired only if the pointer is NULL.

The pointer is then checked *again* once the lock has been acquired c (hence the *doublechecked*

part) in case another thread has done the initialization between the first check

and this thread acquiring the lock:

void undefined\_behaviour\_with\_double\_checked\_locking()

{

if(!resource\_ptr)

{

std::lock\_guard<std::mutex> lk(resource\_mutex);

if(!resource\_ptr)

{

resource\_ptr.reset(new some\_resource);

}

}

resource\_ptr->do\_something();

}

Unfortunately, this pattern is infamous for a reason: it has the potential for nasty race

conditions, because the read outside the lock B isn’t synchronized with the write

done by another thread inside the lock d. This therefore creates a race condition

that covers not just the pointer itself but also the object pointed to; even if a thread

sees the pointer written by another thread, it might not see the newly created instance

Listing 3.11 Thread-safe lazy initialization using a mutex

b

**All threads are**

**serialized here**

**Only the initialization**

**needs protection**

b

c

d

e

***Alternative facilities for protecting shared data* 61**

of some\_resource, resulting in the call to do\_something() e operating on incorrect

values. This is an example of the type of race condition defined as a *data race* by the

C++ Standard and thus specified as *undefined behavior*. It’s is therefore quite definitely

something to avoid. See chapter 5 for a detailed discussion of the memory model,

including what constitutes a *data race*.

The C++ Standards Committee also saw that this was an important scenario, and so

the C++ Standard Library provides std::once\_flag and std::call\_once to handle

this situation. Rather than locking a mutex and explicitly checking the pointer, every

thread can just use std::call\_once, safe in the knowledge that the pointer will have

been initialized by some thread (in a properly synchronized fashion) by the time

std::call\_once returns. Use of std::call\_once will typically have a lower overhead

than using a mutex explicitly, especially when the initialization has already been

done, so should be used in preference where it matches the required functionality

The following example shows the same operation as listing 3.11, rewritten to use

std::call\_once. In this case, the initialization is done by calling a function, but it

could just as easily have been done with an instance of a class with a function call operator.

Like most of the functions in the standard library that take functions or predicates

as arguments, std::call\_once works with any function or callable object.

std::shared\_ptr<some\_resource> resource\_ptr;

std::once\_flag resource\_flag;

void init\_resource()

{

resource\_ptr.reset(new some\_resource);

}

void foo()

{

std::call\_once(resource\_flag,init\_resource); // **Initialization is**

**called exactly once**

resource\_ptr->do\_something();

}

Thread-safe lazy initialization of a class member using **std::call\_once**

class X

{

private:

connection\_info connection\_details;

connection\_handle connection;

std::once\_flag connection\_init\_flag;

void open\_connection()

{

connection=connection\_manager.open(connection\_details);

}

public:

X(connection\_info const& connection\_details\_):

connection\_details(connection\_details\_)

{}

void send\_data(data\_packet const& data)

{

std::call\_once(connection\_init\_flag,&X::open\_connection,this); // Initialization is

called exactly once

connection.send\_data(data);

}

data\_packet receive\_data()

{

std::call\_once(connection\_init\_flag,&X::open\_connection,this);

return connection.receive\_data();

Static variable inititialization

class my\_class;

my\_class& get\_my\_class\_instance()

{

static my\_class instance; // **Initialization guaranteed**

**to be thread-safe in C++11x**

return instance;

}

Multiple threads can then call get\_my\_class\_instance() safely B, without having to

worry about race conditions on the initialization.

# Futures

*Once tie UNIQUE events – cant be reverted.*

*unique futures* (std::future<>) – seems like std::unique\_ptr

and *shared futures* (std::shared\_future<>) - seems like std::shared\_ptr

These are modeled after std::unique\_ptr

and std::shared\_ptr. An instance of std::future is the one and only instance that

refers to its associated event, whereas multiple instances of std::shared\_future may

refer to the same event. In the latter case, all the instances will become *ready* at the

same time, and they may all access any data associated with the event. This associated

data is the reason these are templates; just like std::unique\_ptr and std::shared\_ptr,

the template parameter is the type of the associated data. The std:future<void>,

std::shared\_future<void> template specializations should be used where there’s no

associated data.

Although futures are used to communicate between threads, the

future objects themselves don’t provide synchronized accesses. If multiple threads need

to access a single future object, they must protect access via a mutex or other synchronization

mechanism

# std::async

You use std::async to start an *asynchronous task* for which you don’t need the

result right away. Rather than giving you back a std::thread object to wait on,

std::async returns a std::future object, which will eventually hold the return value

of the function. When you need the value, you just call get() on the future, and the

thread blocks until the future is *ready* and then returns the value.

#include <future>

#include <iostream>

int find\_the\_answer\_to\_ltuae();

void do\_other\_stuff();

int main()

{

std::future<int> the\_answer=std::async(find\_the\_answer\_to\_ltuae);

do\_other\_stuff();

std::cout<<"The answer is "<<the\_answer.get()<<std::endl;

}

By default, it’s up to the implementation whether std::async starts a new thread, or

whether the task runs synchronously when the future is waited for. In most cases this is

what you want, but you can specify which to use with an additional parameter to

std::async before the function to call. This parameter is of the type std::launch, and

can either be std::launch::deferred to indicate that the function call is to be

deferred until either wait() or get() is called on the future, std::launch::async to

indicate that the function must be run on its own thread, or std::launch::deferred |

std::launch::async to indicate that the implementation may choose. This last option

is the default. If the function call is deferred, it may never actually run. For example:

auto f6=std::async(std::launch::async,Y(),1.2); **Run in new thread**

auto f7=std::async(std::launch::deferred,baz,std::ref(x)); **Run in**

**wait()**

**or get()**

auto f8=std::async(

std::launch::deferred | std::launch::async,

baz,std::ref(x)); **Implementation or get()**

**chooses**

auto f9=std::async(baz,std::ref(x)); )); **Implementation or get()**

**chooses**

f7.wait();**Invoke deferred function**

# std::packaged\_task

std::packaged\_task<> ties a future to a function or callable object. When the std::

packaged\_task<> object is invoked, it calls the associated function or callable object

and makes the future *ready*, with the return value stored as the associated data.