Concurrency in C++

Table of Contents

[C++11 2](#__RefHeading___Toc259_808808649)

[Parallelizing code that uses standard algorithms (C++17) 2](#__RefHeading___Toc261_808808649)

[Execution Policies 3](#__RefHeading___Toc263_808808649)

[Supported Algorithms 4](#__RefHeading___Toc1319_2896344822)

[Vectorization 4](#__RefHeading___Toc265_808808649)

[Putting a Thread to Sleep 5](#__RefHeading___Toc267_808808649)

[Starting/Stopping Threads 6](#__RefHeading___Toc269_808808649)

[Thread Synchronization 7](#__RefHeading___Toc271_808808649)

[Mutex 9](#__RefHeading___Toc273_808808649)

[Lock 10](#__RefHeading___Toc275_808808649)

[Avoiding Deadlocks using std::scoped\_lock 11](#__RefHeading___Toc1321_2896344822)

[Scoped Lock 12](#__RefHeading___Toc1323_2896344822)

[Synchronizing concurrent std::cout use 12](#__RefHeading___Toc1325_2896344822)

[Safely postponing initialization with std::call\_once 13](#__RefHeading___Toc1327_2896344822)

[Pushing the execution of tasks into the background using std::async 14](#__RefHeading___Toc1329_2896344822)

[Launch Policy 16](#__RefHeading___Toc1331_2896344822)

[Implementing the producer/consumer idiom with std::condition\_variable 16](#__RefHeading___Toc1333_2896344822)

# C++11

* Introduced std::thread
  + enables basic portable thread control across all operating systems
* introduced mutex classes and comfortable RAII-style lock wrappers
* std::condition\_variable allows for flexible event notification between threads
* std::async: execute functions asynchronously in the background
  + std::future object contains result if this async function call

# Parallelizing code that uses standard algorithms (C++17)

* execution policies: can be added to existing STL (69) algorithms
  + run in parallel on multiple cores, and even with enabled vectorization.
* operating system-independent

#include <iostream>

#include <vector>

#include <random>

#include <algorithm>

#include <execution> // C++17

static bool odd(int n) { return n % 2; }

int main()

{

std::vector<int> d (50000000);

std::mt19937 gen //Mersenne Twister

std::uniform\_int\_distribution<int> dis(0, 100000);

auto rand\_num ([=] () mutable { return dis(gen); }); // why mutable: <https://stackoverflow.com/questions/5501959/why-does-c11s-lambda-require-mutable-keyword-for-capture-by-value-by-defau>

std::generate(std::execution::par, std::begin(d), std::end(d), rand\_num);// std::execution::par is an instance of the class std::execution::parallel\_policy

std::sort(std::execution::par, std::begin(d), std::end(d));

std::reverse(std::execution::par, std::begin(d), std::end(d));

auto odds (std::count\_if(std::execution::par, std::begin(d), std::end(d), std::odd));

std::cout << (100.0 \* odds / d.size()) << "% of the numbers are odd.\n";

}

*\* When using parallel execution policy, it is the programmer's responsibility to avoid data races and deadlocks, eg of data race:\**  
int a[] = {0,1};

[std::vector](http://en.cppreference.com/w/cpp/container/vector)<int> v;

[std::for\_each](http://en.cppreference.com/w/cpp/algorithm/for_each)([std::execution::par](http://en.cppreference.com/w/cpp/algorithm/execution_policy_tag), [std::begin](http://en.cppreference.com/w/cpp/iterator/begin)(a), [std::end](http://en.cppreference.com/w/cpp/iterator/end)(a), [&](int i) {

v.push\_back(i\*2+1); // Error: data race

});

Other examples in notes section of <https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t>

note use of std::begin and std::end (C++11) instead of std::vector::begin/end. Why? Better with template classes, also works with c-style arrays: <https://stackoverflow.com/questions/7593086/why-use-non-member-begin-and-end-functions-in-c11>

\*/

Other execution policies:

std::execution::seq, std::execution::par, std::execution::par\_unseq, and std::execution::unseq – instances of [std::execution::sequenced\_policy](https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t), [std::execution::parallel\_policy](https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t), [std::execution::parallel\_unsequenced\_policy](https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t), and [std::execution::unsequenced\_policy](https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t)

## Execution Policies

1. sequenced\_policy: sequential form similar to the original algorithm without an execution policy. The globally available instance has the name std::execution::seq
2. parallel\_policy: may be executed with multiple threads that share the work in a parallel fashion. The globally available instance has the name std::execution::par.
3. parallel\_unsequenced\_policy: may be executed with multiple threads sharing the work. In addition to that, it is permissible to vectorize the code. In this case, container access can be interleaved between threads and also within the same thread due to vectorization. The globally available instance has the name std::execution::par\_unseq.

All element access functions used by the parallelized algorithm must not cause

deadlocks or data races

In the case of parallelism and vectorization, all the access functions must not use any kind of blocking synchronization

Note that just using parallel STL algorithms correctly does not always lead to guaranteed speedup.

### Supported Algorithms

|  |  |
| --- | --- |
| std::adjacent\_difference std::inplace\_merge  std::replace\_if  std::adjacent\_find  std::is\_heap  std::reverse  std::all\_of  std::is\_heap\_until  std::reverse\_copy  std::any\_of  std::is\_partitioned  std::rotate  std::copy  std::is\_sorted  std::rotate\_copy  std::copy\_if  std::is\_sorted\_until  std::search  std::copy\_n  std::lexicographical\_compare std::search\_n  std::count  std::max\_element  std::set\_difference  std::count\_if  std::merge  std::set\_intersection  std::equal  std::min\_element  std::set\_symmetric\_difference  std::exclusive\_scan  std::minmax\_element  std::set\_union  std::fill  std::mismatch  std::sort  std::fill\_n  std::move | std::stable\_partition  std::find  std::none\_of  std::stable\_sort  std::find\_end  std::nth\_element  std::swap\_ranges  std::find\_first\_of  std::partial\_sort  std::transform  std::find\_if  std::partial\_sort\_copy  std::transform\_exclusive\_scan  std::find\_if\_not  std::partition  std::transform\_inclusive\_scan  std::for\_each  std::partition\_copy  std::transform\_reduce  std::for\_each\_n  std::remove  std::uninitialized\_copy  std::generate  std::remove\_copy  std::uninitialized\_copy\_n  std::generate\_n  std::remove\_copy\_if  std::uninitialized\_fill  std::includes  std::remove\_if  std::uninitialized\_fill\_n  std::inclusive\_scan  std::replace  std::unique  std::inner\_product  std::replace\_copy  std::unique\_copy  std::replace\_copy\_if |

## Vectorization

* both the CPU and the compiler need to support
* eg:   
  int sum {std::accumulate(v.begin(), v.end(), 0)};  
    
  loop generated :  
  int sum {0};  
  for (size\_t i {0}; i < v.size(); ++i) {  
  sum += v[i];  
  }  
    
  With vectorization compiler could generate the following:  
  int sum {0};  
  for (size\_t i {0}; i < v.size() / 4; i += 4) {  
   sum += v[i] + v[i+1] + v[i + 2] + v[i + 3];  
  }  
    
  Why? CPU may be able to perform sum += v[i] + v[i+1] + v[i + 2] + v[i + 3]; in 1 step, reducing the number of instructions

# Putting a Thread to Sleep

* do not need to use any external or operating system-dependent libraries
* Std::this\_thread (C++11)

#include <iostream>  
#include <chrono>  
#include <thread>

using namespace chrono\_literals;

int main()

{

std:: cout << "Going to sleep for 5 seconds and 300 milli seconds.\n";

std::this\_thread::sleep\_for(5s + 300ms); //relative

std::cout << "Going to sleep for another 3 seconds.\n";

std::this\_thread::sleep\_until( std::chrono::high\_resolution\_clock::now() + 3s); //absolute

std::cout<<” Done.\n”;

}

Running the program with time command:

$ time ./sleep

Going to sleep for 5 seconds and 300 milli seconds.

Going to sleep for another 3 seconds.

Done.

real 0m8.320s

user 0m0.005s

sys 0m0.003s

* A thread does not consume CPU time while it is blocked, just becomes inactive
* OS independent
* sleep\_for accepts std::chrono::duration, sleep\_until accepts std::chrono::time\_point
* Timing accuracy depends on OS
* nano second level accuracy/granularity may not be possible
* std::this\_thread::yield:
  + no args
  + tells the OS that another thread of any other process can be scheduled ahead of this one
  + thread is blocked (sleeping) until that other thread is done
  + if no other thread, then the thread doesn’t sleep

# Starting/Stopping Threads

#include <iostream>

#include <thread>

using namespace chrono\_literals;

static void thread\_with\_param(int i)

{

std::this\_thread::sleep\_for(1ms \* i);

std::cout << "Hello from thread " << i << '\n';

std::this\_thread::sleep\_for(1s \* i);

std::cout << "Bye from thread " << i << '\n';

}

int main()

{

std::cout << std::thread::hardware\_concurrency() << " concurrent threads are supported.\n";

std::thread t1 {thread\_with\_param, 1}; // call thread\_with\_param with argument = 1

std::thread t2 {thread\_with\_param, 2};

std::thread t3 {thread\_with\_param, 3};

t1.join(); // blocks current thread until thread 1 returns

t2.join();  
t3.detach(); // let thread 3 running even after t3 is desctructed

cout << "Threads joined.\n";

}

* If we don’t call join or detach, application will be terminated (thread object calls std::terminate()) when the std::thread objects get destructed

Output:

$ ./threads

8 concurrent threads are supported.

Hello from thread 1

Hello from thread 2

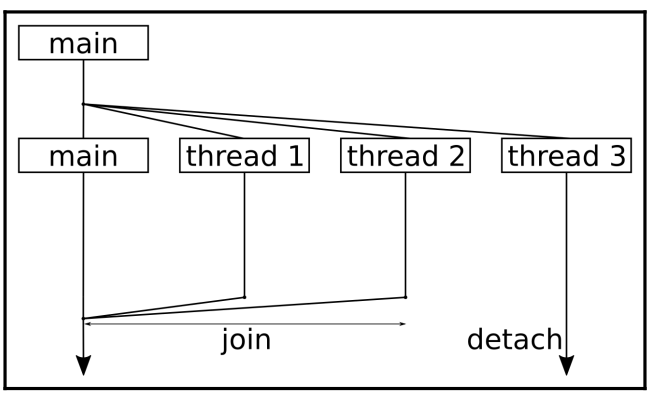
Hello from thread 3

Bye from thread 1

Bye from thread 2

Threads joined.

No bye message from thread 3 as we didn’t join it, and program gets terminated while it was in its wait period of 3 seconds.



* The constructor call of std::thread accepts a function pointer or a callable object, followed by arguments that should be used with the function call
* Parallel vs Concurrent
  + If the computer has only one CPU core, then there can be a lot of threads that run in parallel but never concurrently because one CPU core can only run one thread at a time. They are interleaved.

Currently, no control over:

* The order in which the threads are interleaved when sharing a CPU core.
* The priority of a thread, or which one is more important than the other.
* The fact that threads are really distributed among all the CPU cores or if the operating system just pins them to the same core. It is indeed possible that all our threads run on only a single core, although the machine has more than 100 cores.

Most OS’s provide above capabililties, STL doesn’t.

# Thread Synchronization

#include <iostream>

#include <shared\_mutex>

#include <thread>

#include <vector>

using namespace chrono\_literals;

std::shared\_mutex shared\_mut;

using shrd\_lck = std::shared\_lock<std::shared\_mutex>;

using uniq\_lck = std::unique\_lock<std::shared\_mutex>;  
  
// using vs typedef: <https://stackoverflow.com/questions/10747810/what-is-the-difference-between-typedef-and-using-in-c11>

static void print\_exclusive()

{

uniq\_lck l {shared\_mut, std::defer\_lock};

//”instantiate a unique\_lock instance on the shared mutex”

// defer\_lock tells ctor to keep mutex unlocked (by default ctor locks the mutex)

if (l.try\_lock()) {

cout << "Got exclusive lock.\n";

} else {

cout << "Unable to lock exclusively.\n";

}

}

static void exclusive\_throw()

{

uniq\_lck l {shared\_mut};

throw 123; // unique\_lock guarantees exception safety, so still will be destructed, unlocking the shared\_mut

}

int main()

{

{

shrd\_lck sl1 {shared\_mut}; // immediately lock the mutex in shared mode

std::cout << "shared lock once.\n";

{

shrd\_lck sl2 {shared\_mut}; // sl1 and sl2 both hold a shared lock on the mutex

std::cout << "shared lock twice.\n";

print\_exclusive(); // aquiring exclusive lock will fail as mutex is locked in shared mode

} //dtor of sl2 releases shared lock

std::cout << "shared lock once again.\n";

print\_exclusive(); // exclusive lock will still fail as sl1 has locked the mutex in shared mode

} // dtor of sl1 releases shared lock on shared\_mut

std::cout << "lock is free.\n";

try {

exclusive\_throw();

} catch (int e) {

std::cout << "Got exception " << e << '\n';

}

print\_exclusive(); // able to lock the mutex as unique\_lock unlocks it even on exception

}

Output:

$ ./shared\_lock

shared lock once.

shared lock twice.

Unable to lock exclusively.

shared lock once again.

Unable to lock exclusively.

lock is free.

Got exception 123

Got exclusive lock.

## Mutex

* mutual exclusion
* prevent concurrently running threads from altering the same object in a non-orchestrated way that might lead to data corruption
* different mutex classes in STL have different specialities
* all have lock and unlock methods
* when a thread calls lock on a mutex that is not locked, it owns the mutex
  + other threads will block on their lock calls until locking thread calls unlock
* Mutex classes:
  + mutex: standard with lock and unlock. Provides nonblocking try\_lock method
  + timed\_mutex: standard mutex + try\_lock\_for and try\_lock\_until (allow timing out instead of blocking forever)
  + recursive\_mutex: if a thread locks an instance of this, it can call lock on it multiple times without blocking. Released when owning thread calls unlock equal number of times.
  + recursive\_timed\_mutex: timed\_mutex + recursive\_mutex
  + shared\_mutex
    - can be locked in exclusive or shared mode
    - exclusive mode: same as standard mutex
    - shared mode:
      * shared ownership by multiple threads for non-exclusive access
      * analogous to shared\_ptr
      * if a thread locks the mutex in shared mode, other threads can also lock it in shared mode (but not in exclusive mode)
      * unlocked as soon as the last shared mode lock owner releases it
      * useful when shared data can be safely read by any number of threads simultaneously, but a thread may only write the same data when no other thread is reading or writing at the same time.
  + shared\_timed\_mutex: shared\_mutex + timed\_mutex

## Lock

* If we miss to unlock a mutex somewhere after locking it, or an exception is thrown while a mutex is still locked, difficult to debug as behavior may not be well defined (similar to memory leak issues when delete call is missing – raii lock helpers are analogous to smart pointers)
* std::lock\_guard eg:  
  void critical\_function()  
  {  
   std::lock\_guard<std::mutex> l {some\_mutex};  
   // critical section  
  }
  + The ctor of lock\_guard accepts a mutex and calls lock on it immediately
  + dtor calls unlock
* RAII Lock helper types:
  + C++17 onwards compiler can deduce the type of the template argument automatically
  + std::lock\_guard: ctor and dtor with lock and unlock respectively
  + std::scoped\_lock: supports arbitrarily many mutexes in ctor, which are all released in dtor
  + std::unique\_lock:
    - lock mutex in exclusive mode
    - ctor also accepts arguments to timeout instead of blocking forever on lock
  + std::shared\_lock: like unique lock but in shared mode
  + Locking strategies (supplied to constructor):

|  |  |
| --- | --- |
| Type | Effect(s) |
| defer\_lock\_t | do not acquire ownership of the mutex (only available for unique\_lock and shared\_lock) |
| try\_to\_lock\_t | try to acquire ownership of the mutex without blocking (only available for unique\_lock and shared\_lock) |
| adopt\_lock\_t | assume the calling thread already has ownership of the mutex |

## Avoiding Deadlocks using std::scoped\_lock

#include <iostream>

#include <thread>

#include <mutex>

using namespace chrono\_literals;

std::mutex mut\_a;

std::mutex mut\_b;

static void deadlock\_func\_1()

{

std::cout << "bad f1 acquiring mutex A..." << std::endl;

std::lock\_guard<std::mutex> la {mut\_a};

std::this\_thread::sleep\_for(100ms);

std::cout << "bad f1 acquiring mutex B..." << std::endl;

std::lock\_guard<std::mutex> lb {mut\_b};

std::cout << "bad f1 got both mutexes." << std::endl;

}

static void deadlock\_func\_2()

{

std::cout << "bad f2 acquiring mutex B..." << std::endl;

std::lock\_guard<std::mutex> la {mut\_b};

std::this\_thread::sleep\_for(100ms);

std::cout << "bad f2 acquiring mutex A..." << std::endl;

std::lock\_guard<std::mutex> lb {mut\_b};

std::cout << "bad f2 got both mutexes." << std::endl;

}

static void sane\_func\_1()

{

scoped\_lock l {mut\_a, mut\_b}; // acquires both locks, in the given order. Dtor releases.

std::cout << "sane f1 got both mutexes." << std::endl;

}

static void sane\_func\_2()

{

scoped\_lock l {mut\_b, mut\_a};

std::cout << "sane f2 got both mutexes." << std::endl;

}

int main()

{

{

std::thread t1 {sane\_func\_1};

std::thread t2 {sane\_func\_2};

t1.join();

t2.join();

}

{

std::thread t1 {deadlock\_func\_1};

std::thread t2 {deadlock\_func\_2};

t1.join();

t2.join();

}

}

Output:

$ ./avoid\_deadlock

sane f1 got both mutexes

sane f2 got both mutexes

bad f2 acquiring mutex B...

bad f1 acquiring mutex A...

bad f1 acquiring mutex B...

bad f2 acquiring mutex A…

* This problem is called lock order inversion (when different pieces of codes lock the mutexes in a different order, causing a deadlock)

### Scoped Lock

* Takes multiple mutexes as arguments to ctor
* Uses std::lock function – performs a series of try\_lock calls on mutexes provided to avoid deadlocks

## Synchronizing concurrent std::cout use

#include <iostream>

#include <thread>

#include <mutex>

#include <sstream>

#include <vector>

using namespace std;

struct pcout : public stringstream {

static inline mutex cout\_mutex;

~pcout() {

lock\_guard<mutex> l {cout\_mutex};

cout << rdbuf();

cout.flush();

}

};

static void print\_cout(int id)

{

cout << "cout hello from " << id << '\n';

}

static void print\_pcout(int id)

{

pcout{} << "pcout hello from " << id << '\n'; //produces non garbled output

}

int main()

{

vector<thread> v;

for (size\_t i {0}; i < 10; ++i) {

v.emplace\_back(print\_cout, i);

}

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

cout << "=====================\n";

v.clear();

for (size\_t i {0}; i < 10; ++i) {

v.emplace\_back(print\_pcout, i);

}

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

}

# Safely postponing initialization with std::call\_once

* Sometimes we need a setup/init routine to be called once before executing some parallel code
* Simple solution: execute the setup function before the parallel code
  + Drawbacks:
    - if the parallel function comes from a library, the user must not forget to call the setup function.
    - If the setup function is expensive, or it doesn’t need to be executed every time, then we need code that decides when/if to run it.

#include <iostream>

#include <thread>

#include <mutex>

#include <vector>

std::once\_flag callflag; // needed for the synchronization of all threads that use call\_once on a specific function

static void once\_print() // must be executed once

{

cout << '!';

}

static void print(size\_t x) // called by all threads

{

std::call\_once(callflag, once\_print);

std::cout << x;

}

int main()

{

vector<thread> v;

for (size\_t i {0}; i < 10; ++i) {

v.emplace\_back(print, i);

}

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

cout << '\n';

}

Output:

$ ./call\_once

!1239406758

* First thread to reach the callable provided to std::call\_once gets to execute it
* until this thread has finished executing it, other threads that reach the call\_once line are blocked
* The std::once\_flag is used by the threads to determine if they need to execute the call\_once callable or wait
* If the first thread to execute the call\_once function fails because of some exception, next thread is allowed to execute the function again

# Pushing the execution of tasks into the background using std::async

Usual flow:

std::thread t {my\_function, arg1, arg2, ...};

// do something else

t.join(); // wait for thread to finish

In the above code we don’t have the return value of my\_function: std::async solves this problem

#include <iostream>

#include <iomanip>

#include <map>

#include <string>

#include <algorithm>

#include <iterator>

#include <future>

using namespace std;

static map<char, size\_t> histogram(const string &s)

{

map<char, size\_t> m;

for (char c : s) { m[c] += 1; }

return m;

}

static string sorted(string s)

{

sort(begin(s), end(s));

return s;

}

static bool is\_vowel(char c)

{

char vowels[] {"aeiou"};

return end(vowels) !=

find(begin(vowels), end(vowels), c);

}

static size\_t vowels(const string &s)

{

return count\_if(begin(s), end(s), is\_vowel);

}

int main()

{

cin.unsetf(ios::skipws); // don’t segment input into words

string input {istream\_iterator<char>{cin}, {}};

input.pop\_back();

auto hist (async(launch::async, histogram, input)); // launch::async is the policy

auto sorted\_str (async(launch::async, sorted, input));

auto vowel\_count (async(launch::async, vowels, input));

/\* *The async calls return immediately because they do not actually execute our*

*functions. Instead, they set up synchronization structures which will obtain the*

*results of the function calls later. The results are now being calculated*

*concurrently by additional threads.* \*/

/\* *The return values hist, sorted\_str and vowel\_count are of the types the functions histogram, sorted, and vowels return, but they were wrapped in a* ***future*** *type by*

*std::async. Objects of this type express that they will contain their values at*

*some point in time. By using* ***.get()*** *on all of them, we can make the main*

*function block until the values arrive, and then use them for printing* \*/

for (const auto &[c, count] : hist.get()) {

cout << c << ": " << count << '\n';

}

cout << "Sorted string: "

<< quoted(sorted\_str.get()) << '\n'

<< "Total vowels: "

<< vowel\_count.get()

<< '\n';

}

Output:  
$ echo "foo bar baz foobazinga" | ./async

: 3

a: 4

b: 3

f: 2

g: 1

i: 1

n: 1

o: 4

r: 1

z: 2

Sorted string: "

aaaabbbffginoooorzz"

Total vowels: 9

* async starts a new thread an executes the given function concurrently
* it accepts: launch policy, function, arguments to function
* it returns: a future type which contains the return value of the called function

## Launch Policy

launch::async : The function is guaranteed to be executed by another thread.

launch::deferred : The function is executed by the same thread, but later (lazy evaluation). Execution then happens when get or wait is called on the future. If none of both happens, the function is not called at all.

launch::async | launch::deferred : Having both flags set, the STL's async implementation is free to choose which policy shall be followed. This is the default choice if no policy is provided. This means that we cannot be sure that another thread is started at all, or if the execution is just deferred in the current thread.

# Implementing the producer/consumer idiom with std::condition\_variable

* One thread produces items and puts into a queue.
* Another thread consumes the items.
* If there is nothing to produce, the producer thread sleeps.
* If there is no item in the queue to consume, the consumer sleeps.
* Queue needs to be protected by a mutex since both producer and consumer update it
* Consumer can:
  + poll the queue for new items, or
  + wait for wakeup **events** triggered by the producer

#include <iostream>

#include <queue>

#include <tuple>

#include <condition\_variable>

#include <thread>

using namespace std;

using namespace chrono\_literals;

queue<size\_t> q;

mutex mut; // protects the queue and finished flag

**condition\_variable cv;**

bool finished {false};

static void producer(size\_t items) {

for (size\_t i {0}; i < items; ++i) {

this\_thread::sleep\_for(100ms);

{

lock\_guard<mutex> lk {mut};

q.push(i);

}

**cv.notify\_all();**

}

{

lock\_guard<mutex> lk {mut};

finished = true;

}

cv.notify\_all();

}

static void consumer() {

while (!finished) {

unique\_lock<mutex> l {mut};

cv.wait(l, [] { return !q.empty() || finished; });

/\* The cv.wait call unlocks the lock and waits until the condition described by the

predicate function holds. Then, it locks the mutex again and consumes everything

from the queue until it appears empty. The predicate is checked when another (producer) thread calls notify\_one() or notify\_all() on the same condition\_variable object \*/

while (!q.empty()) {

cout << "Got " << q.front() << " from queue.\n";

q.pop();

}

}

}

int main() {

thread t1 {producer, 10};

thread t2 {consumer};

t1.join();

t2.join();

cout << "finished!\n";

}

Output:

$ ./producer\_consumer

Got 0 from queue.

Got 1 from queue.

Got 2 from queue.

Got 3 from queue.

Got 4 from queue.

Got 5 from queue.

Got 6 from queue.

Got 7 from queue.

Got 8 from queue.

Got 9 from queue.

Finished!

# Implementing the multiple producers/consumers idiom with std::condition\_variable

#include <iostream>

#include <iomanip>

#include <sstream>

#include <vector>

#include <queue>

#include <thread>

#include <mutex>

#include <condition\_variable>

#include <chrono>

using namespace std;

using namespace chrono\_literals;

struct pcout : public stringstream {

static inline mutex cout\_mutex;

~pcout() {

lock\_guard<mutex> l {cout\_mutex};

cout << rdbuf();

}

};

queue<size\_t> q;

mutex

q\_mutex;

bool

production\_stopped {false};

condition\_variable go\_produce;

condition\_variable go\_consume;

static void producer(size\_t id, size\_t items, size\_t stock)

{

for (size\_t i = 0; i < items; ++i) {

unique\_lock<mutex> lock(q\_mutex);

go\_produce.wait(lock, [&] { return q.size() < stock; }); // producer waits till queue size is below the stock threshold

q.push(id \* 100 + i);

pcout{} << "Producer " << id << " --> item " << setw(3) << q.back() << '\n';

go\_consume.notify\_all();

this\_thread::sleep\_for(90ms);

}

pcout{} << "EXIT: Producer " << id << '\n';

}

static void consumer(size\_t id)

{

while (!production\_stopped || !q.empty()) {

unique\_lock<mutex> lock(q\_mutex);

if (go\_consume.wait\_for(lock, 1s, [] { return !q.empty(); })) {

pcout{} << "item "<< setw(3) << q.front()<< " --> Consumer "<< id << '\n';

q.pop();

go\_produce.notify\_all();

this\_thread::sleep\_for(130ms);

}

}

pcout{} << "EXIT: Producer " << id << '\n';

}

int main()

{

vector<thread> workers;

vector<thread> consumers;

for (size\_t i = 0; i < 3; ++i) {

workers.emplace\_back(producer, i, 15, 5);

}

for (size\_t i = 0; i < 5; ++i) {

consumers.emplace\_back(consumer, i);

}

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

production\_stopped = true;

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

}

Output snippet:

$ ./multi\_producer\_consumer

Producer 0 --> item

0

Producer 1 --> item 100

item

0 --> Consumer 0

Producer 2 --> item 200

item 100 --> Consumer 1

item 200 --> Consumer 2

Producer 0 --> item

1

Producer 1 --> item 101

item

1 --> Consumer 0

...

Producer 0 --> item 14

EXIT: Producer 0

Producer 1 --> item 114

EXIT: Producer 1

item 14 --> Consumer 0

Producer 2 --> item 214

EXIT: Producer 2

item 114 --> Consumer 1

item 214 --> Consumer 2

EXIT: Consumer 2

EXIT: Consumer 3

EXIT: Consumer 4

EXIT: Consumer 0

EXIT: Consumer 1

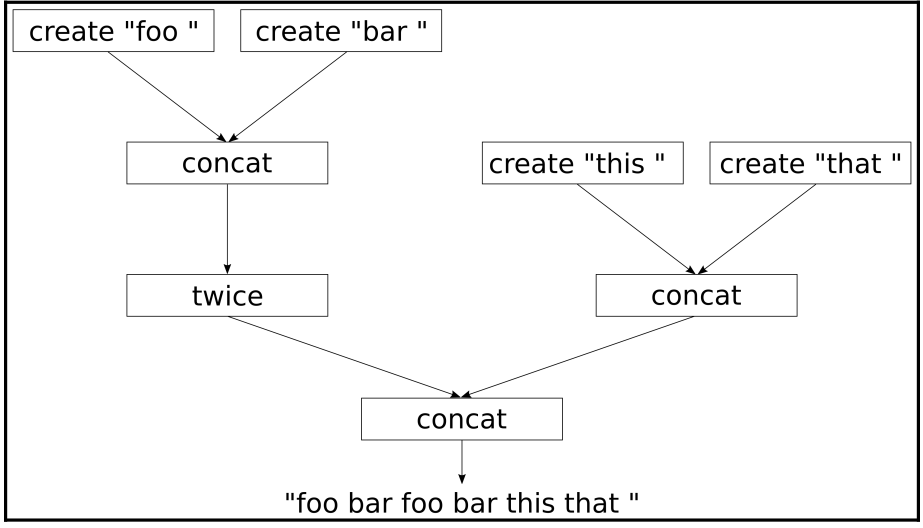
* The go\_produce signals the event that the queue is not completely filled to the maximum and the producers may fill it up again.
* The go\_consume signals the event that the queue reached its maximum length and consumers are free to consume items again.
* In the go\_consume.wait\_for call, we additionally added a timeout argument of 1 second. This is the exit mechanism for consumers: if the queue is empty for longer than a second, maybe there are no active producers any longer.
* If we have (max - 1) items in the queue and want one new item produced so that the queue is filled up again, we have the guarantee that only one producer thread will exit the go\_produce.wait call, because, for all other producer threads, the q.size() < stock wait condition doesn't hold any longer as soon as they get the mutex after being woken up

# Implementing a tiny automatic parallelization library with std::future

Aim: Produce the string “foo bar foo bar this that” using the following available routines:

* create single word
* concat 2 words
* twice (concat with itself)

A directed acyclic graph (DAG) can be drawn from the subtasks:



We want to execute independent subtasks concurrently

#include <iostream>

#include <iomanip>

#include <thread>

#include <string>

#include <sstream>

#include <future>

using namespace std;

using namespace chrono\_literals;

struct pcout : public stringstream {

static inline mutex cout\_mutex;

~pcout() {

lock\_guard<mutex> l {cout\_mutex};

cout << rdbuf();

cout.flush();

}

};

static string create(const char \*s)

{

pcout{} << "3s CREATE " << quoted(s) << '\n';

this\_thread::sleep\_for(3s);

return {s};

}

static string concat(const string &a, const string &b)

{

pcout{} << "5s CONCAT " << quoted(a) << " " << quoted(b) << '\n';

this\_thread::sleep\_for(5s);

return a + b;

}

static string twice(const string &s)

{

pcout{} << "3s TWICE " << quoted(s) << '\n';

this\_thread::sleep\_for(3s);

return s + s;

}

template <typename F>

static auto asynchronize(F f)

{

return [f](auto ... xs) {

return [=] () {

return async(launch::async, f, xs...);

};

};

}

template <typename F>

static auto fut\_unwrap(F f)

{

return [f](auto ... xs) {

return f(xs.get()...);

};

}

template <typename F>

static auto async\_adapter(F f)

{

return [f](auto ... xs) {

return [=] () {

return async(launch::async,fut\_unwrap(f), xs()...);

};

};

}

int main()

{

auto pcreate (asynchronize(create));

auto pconcat (async\_adapter(concat));

auto ptwice (async\_adapter(twice));

auto result (

pconcat(

ptwice(

pconcat(

pcreate("foo "),

pcreate("bar "))),

pconcat(

pcreate("this "),

pcreate("that "))));

cout << "Setup done. Nothing executed yet.\n";

cout << result().get() << '\n';

}

Output:  
$ ./chains

Setup done. Nothing executed yet.

3s CREATE "foo "

3s CREATE "bar "

3s CREATE "this "

3s CREATE "that "

5s CONCAT "this " "that "

5s CONCAT "foo " "bar "

3s TWICE "foo bar "

5s CONCAT "foo bar foo bar " "this that "

foo bar foo bar this that

sss