Chapter 7: The Concurrency API
std::future
https://en.cppreference.com/w/cpp/thread/future
template< class T > class future;
template< class T > class future<T&>;
template<>   class future<void>;
The class template std::future provides a mechanism to access the result of asynchronous operations:

- An asynchronous operation (created via std::async, std::packaged_task, or std::promise) can provide a std::future object to the creator of that asynchronous operation.
- The creator of the asynchronous operation can then use a variety of methods to query, wait for, or extract a value from the std::future.

Item 35: Prefer task-based programming to thread-based
int doAsyncWork();
1.thread-based:
std::thread t(doAsyncWork);

- If you program directly with std::threads, you assume the burden of dealing with thread exhaustion, oversubscription, and load balancing yourself.

2.task-based:
auto fut = std::async(doAsyncWork);

- doAsyncWork produces a return value, which we can reasonably assume the code invoking doAsyncWork is interested in.
  - With the task-based approach, it's easy, because the future returned from std::async offers the get function.
- This call shifts the thread management responsibility to the implementer of the C++ Standard Library.
  - std::async doesn't guarantee that it will create a new software thread.
  - it permits the scheduler to arrange for the specified function (doAsyncWork) to be run on the thread requesting doAsyncWork's result.

3つ種類thread

- Hardware threads are the threads that actually perform computation.
- Software threads (OS threads or system threads) are the threads that the operating system manages across all processes and schedules for execution on hardware threads.
  - Software threads are a limited resource.
  - If you try to create more than the system can provide, a std::system_error exception is thrown.
  - oversubscription: when there are more ready-to-run software threads than hardware threads.
- std::threads are objects in a C++ process that act as handles to underlying software threads.

std::threadsを使うべき場合

- You need access to the API of the underlying threading implementation.
  - The C++ concurrency API is typically implemented using a lower-level platform-specific API, usually pthreads or Windows' Threads.
  - Those APIs are currently richer than what C++ offers.

・The std::thread API offers no direct way to get return values from asynchronously run functions, and if those functions throw, the program is terminated.

・Thread-based programming calls for manual management of thread exhaustion, oversubscription, load balancing, and adaptation to new platforms.

・Task-based programming via std::async with the default launch policy handles most of these issues for you.

Item 36: Specify std::launch::async if asynchronicity is essential

run asynchronouslyの意味：run on a different thread.

std::asyncの2つlaunch policy（std::asyncのdefault launch policyはneither of these）

- std::launch::async, f must be run asynchronously, on a different thread.
- std::launch::deferred, f may run only when get or wait is called on the future returned by std::async.
  - When get or wait is invoked, f will execute synchronously, i.e., the caller will block until f finishes running.

std::asyncのdefault launch policy

```
auto fut1 = std::async(f);
auto fut2 = std::async(std::launch::async |
                       std::launch::deferred,
                       f);
```

- This flexibility permits std::async and the thread-management components of the Standard Library to assume responsibility for thread creation and destruction, avoidance of oversubscription, and load balancing.
- This flexibility leads to uncertainty when accessing thread_locals, implies that the task may never execute, and affects program logic for timeout-based wait calls.

Item 37: Make std::threads unjoinable on all paths

基礎

1. std::thread::joinable

https://en.cppreference.com/w/cpp/thread/thread/joinable

- Checks if the thread object identifies an active thread of execution.
- Checks whether the thread is joinable, i.e. potentially running in parallel context.

2. std::thread::detach

https://en.cppreference.com/w/cpp/thread/thread/detach

- Separates the thread of execution from the thread object, allowing execution to continue independently.
  - Any allocated resources will be freed once the thread exits.
- Permits the thread to execute independently from the thread handle.

僕の理解：detachを呼ぶと、thread object (std::thread, thread handle)もthreadから離された。このthreadは自分で終わるまで実行するだけ。thread objectはもうdangleした。

3. std::thread::join

- Waits for a thread to finish its execution.
- Block the current thread until the thread identified by *this finishes its execution.
  - The completion of the thread identified by *this synchronizes with the corresponding successful return from join(). つまりjoin()がreturnするタイミングも、thread of executionの完了するタイミング。

僕の理解：joinはstd::threadが指しているthread of executionが実行されるまで待つ。detachはstd::threadが指しているthread of executionをstd::threadから切って独自で実行させる。つまりjoinはserial, detachはparallel?

A joinable std::thread corresponds to an underlying asynchronous thread of execution that is or could be running.

- 例えば、a std::thread corresponding to an underlying thread that’s blocked or waiting to be scheduled is joinable.
- 例えば、… that have run to completion is joinable.

Unjoinable std::threads

僕の理解：std::threadとunderlying thread of executionが分けられたら、unjoinable.

underlying thread of executionはsoftware threads (Item 35, p242)?

つまりthread handleとthreadが分けられたら、unjoinable. dangle referenceみたい。

threadとthread of executionは同じ?

- Default-constructed std::threads: have no function to execute, hence don't correspond to an underlying thread of execution.
- std::thread objects that have been moved from.
- std::threads that have been joined.
- std::threads that have been detached.

std::threadのjoinabilityが重要の原因

If the destructor for a joinable thread is invoked, execution of the program (i.e., all threads) is terminated.

なぜall threads?? underlying threadだけじゃない?

僕の理解：他のthreadは多分このjoinable threadの実行を依頼している

だからensure that if you use a std::thread object, it's made unjoinable on every path out of the scope in which it's defined.

やり方：Any time you want to perform some action along every path out of a block, the normal approach is to put that action in the destructor of a local project.

- Such objects are known as RAII objects, and the classes they come from are know as RAII classes.
- RAII: Resource Acquisition Is Initialization.
- RAII classes are common in the Standard Library.
- Examples:
  - STL containers: each container's destructor destroys the container's contents and releases its memory
  - the standard smart pointers
    - std::unique_ptr's destructor invokes its deleter on the object it points to.
    - destructors in std::shared_ptr and std::weak_ptr decrement reference counts.
  - std::fstream objects: their destructors close the files they correspond to.

ThreadのRAII classの例：

```
class ThreadRAII {
public:
	enum class DtorAction { join, detach };
	ThreadRAII(std::thread&& t, DtorAction a) : action(a), t(std::move(t)) {}
	~ThreadRAII() {
		if (t.joinable()) {
			if (action == DtorAction::join) {
				t.join();
			} else {
				t.detach();
			}
		}
	}
	ThreadRAII(ThreadRAII&&) = default;		// support moving
	ThreadRAII& operator=(ThreadRAII&&) = default;
	std::thread& get() { return t; }
private:
	DtorAction action;
	std::thread t;
};
```

- std::thread objects aren't copyable.
- 突然の質問：std::thread tにすればどうなる? まずは、だめ、std::threadはcopyできない。でも、copyableオブジェクトだったら? std::thread tでも、std::move()がtをrvalue（T&&）に変換するから大丈夫でしょう。Item 41に答えがあるかも。copyが発生するのが確かに避ける価値がある。でもT&にすれば?？

- 関数ポインタはもう使わないので、overloaded AddSidePointsAtもstaticにする必要がなくなる。enumを利用するoverloadってできる? 良い? 探そう! ROS source codeのoverload策に学ぼう! (ros::NodeHandle) https://docs.ros.org/api/roscpp/html/
- because std::thread objects may start running a function immediately after they are initialized, it's a good habit to declare them last in a class.
- get() is analogous to the get functions offered by the standard smart pointer classes that give access to their underlying raw pointers.
  - Providing get avoids the need for ThreadRAII to replicate the full std::thread interface, and it also means that ThreadRAII objects can be used in contexts where std::thread objects are required.
- The "proper" solution to std::thread destruction would be to communicate to the asynchronously running lambda that we no longer need its work and that it should return early, but there's no support in C++11 for interruptible threads.

Item 38: Be aware of varying thread handle destructor behavior

The destructor for a future sometimes behaves as if it did an implicit join, sometimes as if it did an implicit detach, and sometimes neither.
- It never causes program termination.

Caller←Shared State (Callee's Result)←Callee
- The destructor for the last future referring to a shared state for a non-deferred task launched via std::async blocks until the task completes. (exception) shared stateを壊す条件:
  - It refers to a shared state that was created due to a call to std::async
  - The task's launch policy is std::launch::async
  - The future is the last future referring to the shared state
- The destructor for all other futures simply destroys the future object. (normal)

Item 39: Consider void futures for one-shot event communication

inter-thread communication: a task tells a second, asynchronously running task that a particular event has occurred, because the second task can't proceed until the event has taken place.

方法1: condition variable、良くなさそう。

1. mutexを使うのがcode smellだ。Mutexes are used to control access to shared data, but it's entirely possible that the detecting and reacting tasks have no need for such mediation.
2. If the detecting task happens to execute the notification before the reacting task executes the wait, the reacting task will miss the notification, and it will wait forever.
3. The wait statement fails to account for spurious wake's. つまり以下の再度チェックが必要

cv.wait(lk, []{ return whether the event has occurred; });でもinter-threadなので、reacting taskはチェックできない。That's why it's waiting on a condition variable!
- Only condition variables are susceptible to spurious wakeups.

用語:

code smell: コードの臭い

https://ja.wikipedia.org/wiki/コードの臭い

In computer programming, a code smell is any characteristic in the source code of a program that possibly indicates a deeper problem.
Determining what is and is not a code smell is subjective, and varies by language, developer, and development methodology.

方法2: shared boolean flag

std::atomic<bool> flag(false);
flag = true; // tell reacting task

```
while(!flag); // wait for event
```
問題：the cost of polling in the reacting task.

- During the time the task is waiting for the flag to be set, the task is essentially blocked, yet it's still running. つまりtruly blocked taskではない！
- As such, it occupies a hardware thread that another task might be able to make use of, it incurs the cost of a context switch each time it starts or completes its time-slice, and it could keep a core running that might otherwise be shut down to save power. truly blocked taskだったら、これら一切しない。
- That's an advantage of the condvar-based approach, because a task in a wait call is truly blocked.

方法３：combine condvar and shared boolean flag

```
std::condition_variable cv;
std::mutex m;
bool flag(false); // not std::atomic
{
        std::lock_guard<std::mutex> g(m);
        flag = true; // tell reacting task (part 1)
} // unlock m via g's dtor
cv.notify_one(); // tell reacting task (part 2)
reacting task:
{
        std::unique_lock<std::mutex> lk(m);
        cv.wait(lk, []{ return flag; }); // use lambda to avoid spurious wakeups
        … // react to event (m is locked)
}
… // continue reacting (m now unlocked)
```
まだ良くなさそうなところ：同じことを二度やっている

- Notifying the condition variable tells the reacting task that the event it's been waiting for has probably occurred, but the reacting task must check the flag to be sure.
- Setting the flag tells the reacting task that the event has definitely occurred, but the detecting task still has to notify the condition variable so that the reacting task will awaken and check the flag.

unique_lock: C++11
https://en.cppreference.com/w/cpp/thread/unique_lock

- The class unique_lock is a general-purpose mutex ownership wrapper allowing deferred locking, time-constrained attempts at locking, recursive locking, transfer of lock ownership, and use with condition variables.
- The class unique_lock is movable, but not copyable.
- The class unique_lock meets the BasicLockable requirements. If Mutex meets the Lockable requirements, unique_lock also meets the Lockable requirements; if Mutex meets the TimedLockable requirements, unique_lock also meets the TimedLockable requirements.
- implements movable mutex ownership wrapper

lock_guard: C++11

- The class lock_guard is a mutex wrapper that provides a convenient RAII-style mechanism for owning a mutex for the duration of a scoped block.
- When a lock_guard object is created, it attempts to take ownership of the mutex it is given. When control leaves the scope in which the lock_guard object was created, the lock_guard is destructed and the mutex is released.
- The lock_guard class is non-copyable.
- implements a strictly scope-based mutex ownership wrapper.

scoped_lock: C++17

- deadlock-avoiding RAII wrapper for multiple mutexes.

方法４：having the reacting task wait on a future that's set by the detecting task

Such a communications channel can be used in any situation where you need to transmit information from one place in your program to another.
std::promise<void> p;
voidについて説明:
1. The only thing of interest to the reacting task is that its future has been set.
2. What we need for the std::promise and future templates is a type that indicates that no data is to be conveyed across the communications channel.

detecting task: p.set_value();
reacting task: p.get_future().wait(); // truly blocked after making the wait call.

・Designs employing a flag avoid those problems, but are based on polling, not blocking.

・A condvar and flag can be used together, but the resulting communications mechanism is somewhat stilted.

・Using std::promise and futures dodges these issues, but the approach uses heap memory for shared states, and it's limited to one-shot communication.

Item 40: Use std::atomic for concurrency, volatile for special memory

std::atomic:
Once a std::atomic object has been constructed, operations on it behave more or less as if they were inside a mutex-protected critical section, but the operations are generally implemented using special machine instructions that are more efficient than would be the case if a mutex were employed.
std::atomic<int> ai(0);
++ai;
- read-modify-write (RMW) operations, yet they execute atomically.
- This is one of the nicest characteristics of the std::atomic types: once a std::atomic object has been constructed, all member functions on it, including those comprising RMW operations, are guaranteed to be seen by other threads as atomic.

volatile: guarantee virtually nothing in a multithreaded context
volatile int vi(0);
vi = 10;
std::cout << vi;
++vi;

std::atomicのもう1つ成功する例

背景: As a general rule, compilers are permitted to reorder such unrelated assignments.

でもstd::atomicが存在する場合、std::atomic imposes restrictions on how code can be reordered.

例えば: No code that precedes a write of a std::atomic variable may take place afterwards.

- volatileはこういう効果はない。

ではvolatileの存在する意味は: telling compilers that they're dealing with memory that doesn't behave normally.

Normal memory: if you write a value to a memory location, the value remains there until something overwrites it.
Special memory: Locations in such memory actually communicate with peripherals, e.g., external sensors or displays, printers, network ports, etc. rather than reading or writing normal memory (i.e., RAM).
- 例えばmemory used for memory-mapped I/O.
- volatile is the way we tell compilers that we're dealing with special memory.
  - Its meaning to compilers is "Don't perform any optimizations on operations on this memory."

なぜstd::atomicがcopy constructionできない、copy assignmentできない?

- In order for the copy construction of y from x to be atomic, compilers would have to generate code to read x and write y in a single atomic operation.
- Hardware generally can’t do that.

結論：
- std::atomic is useful for concurrent programming, but not for accessing special memory.
- volatile is useful for accessing special memory, but not for concurrent programming.
  - volatile is for memory where reads and writes should not be optimized away.