
多线程
Multithreading

现代C++基础 Modern C++ Basics

Jiaming Liang, undergraduate from Peking University

- **Thread**
- **Synchronization utilities**
- **High-level Abstraction of Asynchronous Operation**

Multithreading

Thread

Multithreading

- Thread
 - Abstract thread model
 - thread
 - jthread
 - Miscellaneous topics

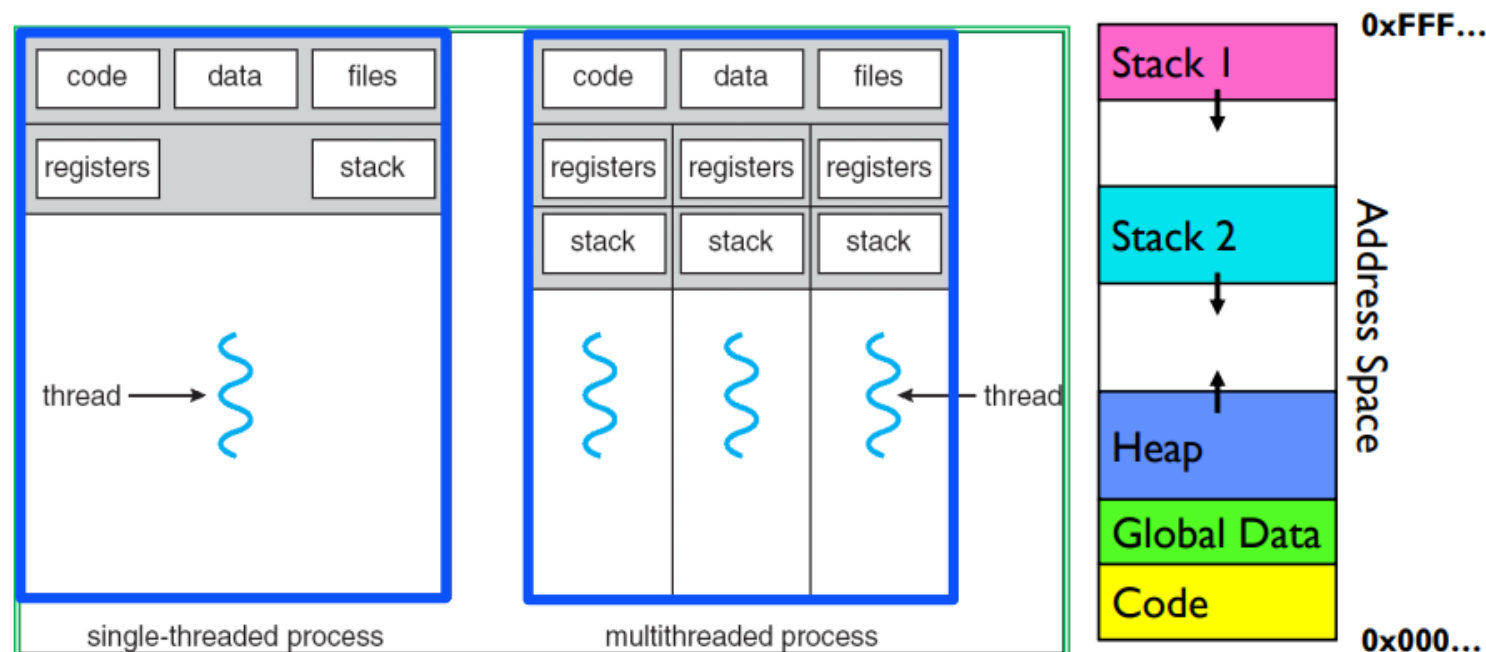
Thread model

- We first briefly review what thread is...
- We've learnt in ICS that each program is a **process**;
 - It has independent address space, and possibly other status like file descriptors (depend on OS).
 - Good isolation, good protection, really limited ability to access memory of another process.
- Threads: less protection, better data sharing!
 - They still partially keep their own set of resources (like registers)...
 - But lie in the same virtual address space, so easy to access memory of other threads!
 - Usually, OS will schedule threads instead of processes;
 - So to some extent, we can say threads are the smallest units to utilize multi-core parallelism.

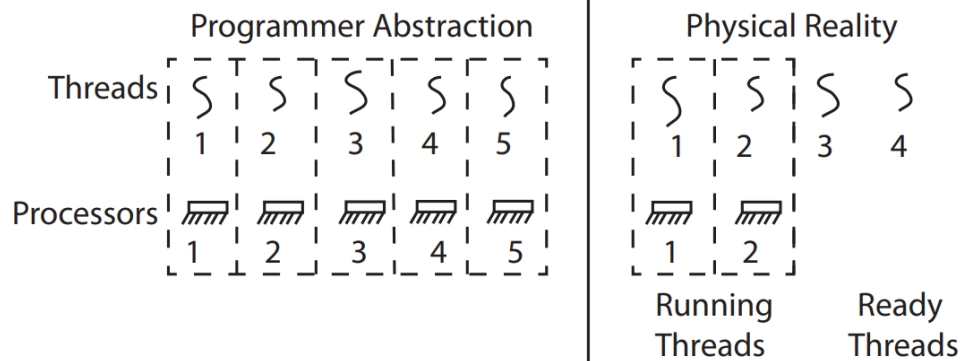
BTW: C++ standard in fact doesn't have a concept of "process", since all things it regulates happen in a single program; you need OS-dependent APIs to manipulate processes, like [fork/exec](#) in Linux, or external libraries like Boost.Process.

Thread model

- In a nutshell:



Roughly speaking, if there are two physical cores, there are actually only two threads executing simultaneously; but OS gives an illusion that more than two threads run concurrently by scheduling.



Credit: Prof. Jin Xin @ PKU OS.

We only give a very rough understanding of thread; you need to learn it comprehensively in OS course.

Thread model

- Scheduling is in fact pausing a running thread, then running a ready thread.
 - And scheduling algorithms determine which to pause and which to run.
 - Registers will be saved and restored during context switch.
- Threads compete with each other for executing themselves!
 - Thus, you may think statements may execute in any order, which leads to data race and synchronization problems.^[1]
- Threads need to be **joined** or **detached** after creation; the former will **wait** until the thread function exits, and the latter will make it execute separately and freely.

[1]: We'll give a rigorous definition for data races in *Advanced Concurrency*.

Multithreading

- Thread
 - Thread model in computers
 - thread
 - jthread
 - Miscellaneous topics

Thread

- We've learnt in ICS how to use **pthread** in POSIX system.

- **pthread_create/join/...**, like this:

- Obscure C interface...

1. The thread function should be **void* func(void*)**;
2. Parameters are packed in a **void*** to be passed in.
 - You need to unpack it inside the thread function, like **(int)** here.
3. Return value is accepted by **void***.
 - Here we **pthread_join(..., NULL)**, i.e. not need return value.
 - If you need it, unpack again...

- Very strange... can we improve it in C++?

```
// create the function to be executed as a thread
void *thread(void *ptr)
{
    int type = (int) ptr;
    fprintf(stderr, "Thread - %d\n", type);
    return ptr;
}

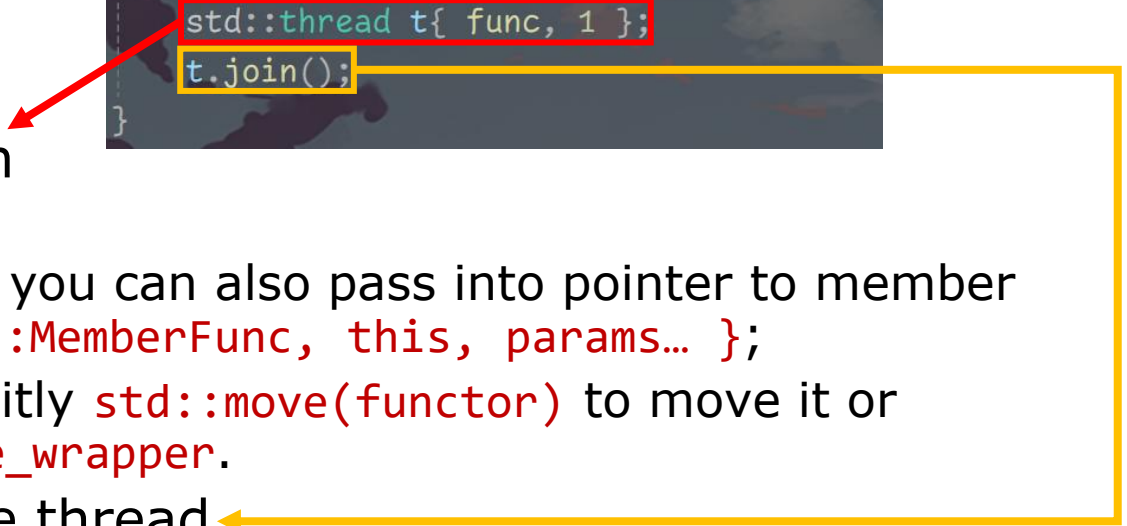
int main(int argc, char **argv)
{
    // create the thread objs
    pthread_t thread1, thread2;
    int thr = 1;
    int thr2 = 2;
    // start the threads
    pthread_create(&thread1, NULL, thread, (void *) thr);
    pthread_create(&thread2, NULL, thread, (void *) thr2);
    // wait for threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    return 0;
}
```

Thread

- Of course! We may code like:
- Very intuitive, very simple, by `std::thread` defined in `<thread>`.
 1. It isn't limited to a function, but can use any functor.
 - It's equivalent to call `std::invoke`, so you can also pass into pointer to member function with `this`, like `{ &SomeClass::MemberFunc, this, params... }`;
 - Functor will be copied; you can explicitly `std::move(funcutor)` to move it or `std::ref` to make it a `std::reference_wrapper`.
 2. You can call `.detach()` to detach the thread.
 - After join/detach, the underlying thread is not associated with `std::thread` object. Unless you move a new object to it, this `std::thread` object is in an empty state (just like default-constructed/moved).
 3. Return value of `func` will be omitted; it should be passed by ref. param.

```
void func(int num)
{
    std::println("Thread info: {}", num);
}

int main()
{
    std::thread t{ func, 1 };
    t.join();
}
```



Thread

- And some other APIs:
 1. Move ctor, move assignment, and swappable (by `.swap()` or `std::swap`).
 2. `.joinable() -> bool`: whether the thread is in an empty state.
 - E.g. before calling `.join()/.detach()`, it returns `true`; afterwards `false`.
 3. Dtor: **`std::terminate` if `.joinable()`**, otherwise do nothing;
 - That is, every running `std::thread` must call `.join()/.detach()` before destruction.
 4. `.get_id()`: get thread id;
 - Class `std::thread::id` instead of simply an integer.
 - Restricted integer: only comparable, hashable and printable (by `<<` since C++11 or `std::formatter` since C++23).
 - Particularly, it only supports `fill-align-width` format.
 - `std::thread` in empty state will get default-constructed `id`.

Thread

```
INVOKE( decay-copy( std::forward<F>(f) ),  
        decay-copy( std::forward<Args>(args) ) ... ) (until C++23)
```

```
std::invoke( auto( std::forward<F>(f) ),  
            auto( std::forward<Args>(args) ) ... ) (since C++23)
```

- **Note:** parameters are decay-copied to the thread functions.
 - And since C++23 it can be explained by **auto(...)** + materialization, i.e. generate a prvalue that is materialized in the current thread.
 - So **auto** will decay type, e.g. **Object& -> Object**.
 - When forwarded type is **Object&**, then copy;
 - When **Object&&**, then move;
- Let's break it down step by step...

```
void func(Object object)  
{  
    std::cout << "Thread id: " << std::this_thread::get_id() << "\n";  
}  
  
int main()  
{  
    std::cout << "Main id: " << std::this_thread::get_id() << "\n";  
    std::thread t{ func, Object{} };  
    t.join();  
}
```

```
class Object  
{  
public:  
    Object() { std::cout << "Construct at " << std::this_thread::get_id() << "\n"; }  
    ~Object() { std::cout << "Destruct at " << std::this_thread::get_id() << "\n"; }  
    Object(const Object&) {  
        std::cout << "Const Copy at " << std::this_thread::get_id() << "\n";  
    };  
    Object(Object&&) { std::cout << "Move at " << std::this_thread::get_id() << "\n"; }  
    Object& operator=(const Object&) {  
        std::cout << "Const Copy Assignment at " << std::this_thread::get_id() << "\n";  
        return *this;  
    };  
    Object& operator=(Object&&) {  
        std::cout << "Move Assignment at " << std::this_thread::get_id() << "\n";  
        return *this;  
    };  
};
```

Thread

```
std::thread t{ func, Object{} };
```

```
template< class F, class... Args >  
explicit thread( F&& f, Args&&... args );
```

1. Parameters are passed into ctor of `std::thread`.
 - Encountering reference, prvalue `Object{}` is materialized and thus constructed as `arg0`.

2. `arg0` is then decay-copied.

- Here it's prvalue `Object{ std::move(arg0) }`, and materialized (and thus move-constructed).

```
INVOKE(decay-copy(std::forward<F>(f)),  
        decay-copy(std::forward<Args>(args))...) (until C++23)
```

```
std::invoke(auto(std::forward<F>(f)),  
            auto(std::forward<Args>(args))...) (since C++23)
```

3. Then thread executes `std::invoke`;

- Materialized parameters are passed to new thread.

```
template< class F, class... Args >  
std::invoke_result_t<F, Args...>  
invoke( F&& f, Args&&... args )
```

4. And finally `arg1` is forwarded to parameters of `func`.

- And thus move-constructed as `object`.

1) Invoke the *Callable* object `f` with the parameters `args` as by

```
INVOKE(std::forward<F>(f), std::forward<Args>(args)...) (since C++23)
```

```
void func(Object object)  
{  
    ...  
    std::cout << "Thread id: " << std::this_thread::get_id() << "\n";  
}
```

Thread

You'll implement `std::thread` yourself in our homework to know how these really happen.

- The first three steps happen at the current thread, and the final step happens at the new thread.
- So the output is like:
 - Any exception thrown in step 1 & 2 can then be caught in old thread.
- Exercise: is this piece of code right?
 - No, since we forward materialized temporary to `func`, i.e. `func(std::move(...))`;
 - And you cannot bind rvalue to lvalue reference...
 - Even if you use `const int&`, it in fact refers to a temporary, not the parameter you passed to thread ctor.

```
Main id: 21244
Construct at 21244
Move at 21244
Destruct at 21244
Move at 8504
Thread id: 8504
Destruct at 8504
Destruct at 8504
```

Step 1
Step 2; new thread executes Step 3.
Param of thread ctor destructed.
Step 4

Param of func destructed.
Materialized temporary destructed.

```
#include <thread>
#include <iostream>

void func(int& type) {
    type = 2;
}

int main()
{
    int type = 1;
    std::thread t{ func, type };
    t.join();
    return 0;
}
```

```
/opt/compiler-explorer/gcc-13.2.0/include/c++/13.2.0/bits/std_thread.h:157:72: error: static
assertion failed: std::thread arguments must be invocable after conversion to rvalues
157 |                                     typename decay<_Args>::type...>::value,
    |                                     ^~~~~~
```


Thread

- Reason: decay-copy instead of reference is safer.
 - We've learnt in ICS that you may pass a pointer to another thread, so that another thread can access memory of current thread.
 - If the referred object goes out of its lifetime, you're accessing invalid memory!
 - Simultaneous access in different threads may lead to data races too.
- You need to use `std::(c)ref()` explicitly to pass the (const) reference.
 - You've seen similar way in `std::bind_xx`, which also warns you about lifetime problem.

```
std::thread t{ func, std::ref(type) };  
t.join();  
std::cout << type; 2请按任意键继续. . .
```

Thread

```
native_handle_type native_handle();
```

(since C++11)
(not always present)

Returns the implementation defined underlying thread handle.

- Note 1: APIs provided in `std::thread` are high-level; sometimes you may want fine-grained control.
 - For example, you may want to change the priority of some threads.
 - It's platform-dependent, so C++ provides a `.native_handle()` for it.
 - The return type is platform-dependent (e.g. `pthread_t` in POSIX system).
- Note 2: `static constexpr std::thread::hardware_concurrency()` can be used to check number of real parallel threads.
 - Roughly speaking, how many physical cores.
 - This is only a hint to the possible maximum parallelism; you need profiling to get the best thread number for your program's performance.
 - When the system cannot give a hint, return 0.

Thread

- Note 3: some rare but possible exceptions, listed here.

- All exceptions are `std::system_error` with some error code.
- Ctor: *Throws:* `system_error` if unable to start the new thread.

Error conditions: i.e. error code

— `resource_unavailable_try_again` — the system lacked the necessary resources to create another thread, or the system-imposed limit on the number of threads in a process would be exceeded.

- `.join()/.detach()`:

Error conditions:

— `resource_deadlock_would_occur` — if deadlock is detected or `get_id() == this_thread::get_id()`. A thread waiting for itself; `.detach()` doesn't have this case.

— `no_such_process` — if the thread is not valid.

— `invalid_argument` — if the thread is not joinable.

Thread

- Note 4: `namespace std::this_thread` has many methods for the current thread.
 - `get_id()`: get id of current thread.
 - `sleep_for()/sleep_until()`: pause the current thread.
 - `yield()`: request scheduling.
 - We've said that threads compete with each other; OS will schedule a thread when it has executing for a period of time.
 - That is, a thread will execute eagerly, and OS forces it to pause.
 - `yield` means the thread gives up execution right voluntarily, and OS re-schedules it.
 - However, OS may still choose the original thread to run, if the priority of this thread is high enough so that scheduling algorithms still choose it.
 - i.e. pause the thread, save its state, and reload the same state, and continue to run.

Multithreading

- Thread
 - Thread model in computers
 - thread
 - jthread
 - Miscellaneous topics

jthread

- C++ encourages RAII, which means dtor will release resource acquired by ctor.
 - But `std::thread` seems to violate it, because if you forget to join/detach a thread, then the whole program is terminated.
 - `std::jthread` in `<thread>` since C++20 is used to solve that; it will automatically join the thread if its `joinable()` is still `true` in dtor.
 - It has all APIs of `std::thread`, i.e. you can use `join/detach/swap/move/native_handle/joinable/get_id/hardware_concurrency`.
 - But move will try to join thread it holds, and then move another to `*this`.
 - **Self-move will also join itself!**
- In C++11, some argue that termination is better than silent wait, which makes it not default behavior for `std::thread`.

From the start (pre-C++11), many (incl. me) had wanted `thread` to have what is now `jthread`'s behavior, but people grounded in traditional operating systems threads insisted that terminating a program was far preferable to a deadlock. In 2012 and 2013, Herb Sutter proposed a joining

jthread

- Besides, `std::jthread` also adds **stop token handling**.
 - Also sometimes called *cooperative cancellation*.
 - We know that threads compete and execute eagerly.
 - You can seldom *force* a thread to do something, but **request** it to do.
 - For stopping a thread, you may set some shared data, and the thread checks it periodically; when check succeeds, it **returns voluntarily**.
-
- Note that you can NOT kill a thread (though you can kill a process), since data dependence in threads is too common.
 - For example, what if a thread is still holding a lock, but it's forced to exit? Then the waiting thread will go into deadlock!
 - All in all, you can hardly ever guarantee a thread to be in a consistent state when you kill it; that's why we need stop token as a hint.

Stop token

- So the requester holds a ***stop source***, and the thread holds a ***stop token*** that associates with the stop source.
 - To prevent use-after-free, they share an underlying *stop state* with reference counts;
 - The state records related information and will be freed when counts goes to 0.
 - The stop source can only request once, which sets some flag in state;
 - Future requests have no actual effects.
 - And the stop token can check regularly whether the flag is set.
- So the state should expose interface below:
 - Setter: `request_stop()`, set the flag;
 - Getter: `stop_requested()`, check whether the flag is set;
 - Share/Detach: increment/decrement reference count.

Stop token

- And accordingly, `std::stop_source` and `std::stop_token` in `<stop_token>` wrap and expose them in a thread-safe way.
- `std::stop_source`:
 - Setter: `.request_stop()`;
 - Getter: `.stop_requested()`;
 - Share & Detach:
 - Default ctor: create a `stop_source` with newly created state.
 - Copy ctor & assignment: share the current state with others;
 - Move ctor & assignment: transfer the ownership;
 - Dtor: detach.
 - `.get_token()` -> `std::stop_token`: get a stop token that shares the same state.

Stop token

- `std::stop_token`:
 - No setter;
 - Getter: `.stop_requested()`;
 - Share & Detach:
 - Copy ctor & assignment: share the current state with others;
 - Move ctor & assignment: transfer the ownership;
 - Dtor: detach.
- For example:

```
void func(std::stop_token token, int& cnt)
{
    while (!token.stop_requested())
    {
        cnt++;
    }
}
```

```
int main()
{
    int cnt = 0;
    using namespace std::literals; // To use literal suffix 'ms'.

    std::stop_source source;
    std::thread t{ func, source.get_token(), std::ref(cnt) };
    std::this_thread::sleep_for(1ms);
    source.request_stop();
    t.join();

    return 0;
}
```


Stop token

- They can also attach to an empty state, then every operation does nothing.
 - `std::stop_token`: default construct it;
 - `std::stop_source`: add a placeholder tag:

```
explicit stop_source( std::nostopstate_t nss ) noexcept;    (2)  std::stop_source source{ std::nostopstate };
```

- Since default ctor will create a new state.
 - When they're e.g. moved-from, then the state will be empty too.
 - To check whether the current state is empty, you can use method `.stop_possible()`; it returns `false` when empty.
 - And `.stop_requested()` also returns `false` when empty.
- Particularly, when only stop tokens associate with a state that hasn't been requested (i.e. no stop source, so no possible request), `token.stop_possible()` also returns `false`.

- To conclude:

std::stop_token

Member functions

(constructor)	constructs new stop_token object (public member function)
(destructor)	destructs the stop_token object (public member function)
operator=	assigns the stop_token object (public member function)

Modifiers

swap	swaps two stop_token objects (public member function)
------	--

Observers

stop_requested	checks whether the associated stop-state has been requested to stop (public member function)
stop_possible	checks whether associated stop-state can be requested to stop (public member function)

Non-member functions

operator== (C++20)	compares two std::stop_token objects (function)
swap(std::stop_token) (C++20)	specializes the std::swap algorithm (function)

std::stop_source

Member functions

(constructor)	constructs new stop_source object (public member function)
(destructor)	destructs the stop_source object (public member function)
operator=	assigns the stop_source object (public member function)

Modifiers

request_stop	makes a stop request for the associated stop-state, if any (public member function)
swap	swaps two stop_source objects (public member function)

Observers

get_token	returns a stop_token for the associated stop-state (public member function)
stop_requested	checks whether the associated stop-state has been requested to stop (public member function)
stop_possible	checks whether associated stop-state can be requested to stop (public member function)

Non-member functions

operator== (C++20)	compares two std::stop_source objects (function)
swap(std::stop_source) (C++20)	specializes the std::swap algorithm (function)

Stop token

- So how does `std::jthread` cooperate with stop token?
 1. It contains a default-constructed `std::stop_source` directly.
 - `.get_stop_source()` to get a copy;
 - `.get_stop_token()` to get a token associated with underlying source;
 - Equivalent to `underlying_source.get_token()`.
 - And `.request_stop()`, equivalent to `underlying_source.request_stop()`.
 2. In dtor of `std::jthread`, if the source hasn't issued a request, call `.request_stop()`;
 - RAII to some extent.
 3. When possible, it will pass `get_token()` to its functor.

The new thread of execution starts executing:

```
std::invoke(auto(std::forward<F>(f)), get_stop_token(),  
            auto(std::forward<Args>(args))...) (since C++23)
```

if the expression above is well-formed, otherwise starts executing:

```
std::invoke(auto(std::forward<F>(f)),  
            auto(std::forward<Args>(args))...) (since C++23)
```

Stop token

- For example:

`get_stop_token()` is provided automatically.

```
using namespace std::literals;
std::jthread t{ [](std::stop_token token) {
    while (!token.stop_requested())
    {
        std::cout << "PKU No.1!";
    }
} };
std::this_thread::sleep_for(1s);
t.request_stop();
```

This may be omitted since
dtor of `std::jthread` will
automatically `.request_stop()`.

Note that `std::cout` is safe to be used in multiple threads; other streams should use `std::osyncstream` in C++20.

Stop token

- Another example:

```
using namespace std::literals;
std::jthread t{ [](std::stop_token token) {
    while (!token.stop_requested())
    {
        std::cout << "PKU No.1!\n";
    }
} };

std::jthread t2{ [](std::stop_token token) {
    while (!token.stop_requested())
    {
        std::cout << "THU No.2!\n";
    }
}, t.get_stop_token() };

std::this_thread::sleep_for(1s);
t.request_stop();
```

Question: can we omit `t.request_stop()` here?

No! Since `t2` is destructed first, so `t2.join()` is before `t.request_stop()` in dtor of `t`.

Thus infinite loop in `t2`...

`t2` doesn't use its own `.get_token()` in functor; the functor shares the same state with `t`.

The new thread of execution starts executing:

```
std::invoke(auto(std::forward<F>(f)), get_stop_token(),
            auto(std::forward<Args>(args))...) (since C++23)
```

if the expression above is well-formed, otherwise starts executing:

```
std::invoke(auto(std::forward<F>(f)),
            auto(std::forward<Args>(args))...). (since C++23)
```

Stop token

- Finally, stop request can be associated with callbacks.

- By `std::stop_callback` with `std::stop_token`:

- Ctor registers callback on the state;

```
template< class C >
explicit stop_callback( const std::stop_token& st, C&& cb ) noexcept(/*see below*/);
```

- Dtor deregisters the callback.

- For example:

Callbacks will be executed here.

```
using namespace std::literals;
std::jthread t{ [](std::stop_token token) {
    while (!token.stop_requested())
    {
        std::cout << "PKU No.1!\n";
    }
} };

std::stop_callback callback{
    t.get_stop_token(),
    []() { std::cout << "THU No.2!\n"; }
};

std::this_thread::sleep_for(1s);
t.request_stop();
```

Stop token

- Note 1: it's quite like doing callback in a thread-safe way.
 - Callbacks will be executed exactly once for multiple requests;
 - Register and deregister are thread-safe; Deregister in a thread will wait for invocation in another thread if they happen in parallel.
- Note 2: the thread that first calls `.request_stop()` will execute all callbacks;
 - If there are multiple callbacks, the execution order is not regulated.
- Note 3: when request has been issued before registering...
 - i.e. in ctor of `std::stop_callback`, `token.stop_requested() == true`;
 - Then callback will be executed immediately in ctor in the current thread.
- Note 4: callback is not allowed to throw; `std::terminate()` if exception is thrown out of callback (treated as if `noexcept`).

Multithreading

- Thread
 - Thread model in computers
 - thread
 - jthread
 - Miscellaneous topics

Exception in threads

- We know that if we don't catch an exception in a single-threaded program, then `std::terminate`.
- Generally speaking, any thread that doesn't catch its exception when exiting will lead to `std::terminate`.
- So code right doesn't work:

```
void func()
{
    throw std::runtime_error{ "Not implemented" };
}

int main()
{
    try {
        std::thread t{ func }; // std::terminate!
        t.join();
    }
    catch (const std::runtime_error& err)
    {
        // ...
    }
}
```

Exception in threads

- So how can we pass the exception out of the thread?
- By `std::exception_ptr` defined in `<exception>`!
 - Roughly speaking, it's a shared pointer to exception.
 - Only when all pointers to the exception object destruct will the object destruct.
 - The actual type is implementation-defined... `using exception_ptr = /*unspecified*/` (since C++11)
 - It's regulated to expose these interfaces:
 - Default construct: as if it's a `nullptr`;
 - `std::make_exception_ptr(Exception)`: make a pointer that copies `Exception`;
 - And can be converted to `bool`, like a pointer.
 - `std::current_exception()`: used in catch block, as if `make_exception_ptr` to the current caught exception;
 - `std::rethrow_exception(std::exception_ptr)`: rethrow the exception object.

Exception in threads

- For example:

Pass exception
out of thread
by parameter.

```
void THUStudent() {  
    throw std::runtime_error("THU is not best!");  
    std::cout << "THU is best.\n";  
}  
  
void PKUStudent() {  
    std::cout << "PKU is best.\n";  
}  
  
void Work(std::exception_ptr& ptr)  
{  
    try {  
        PKUStudent();  
        THUStudent();  
    }  
    catch (const std::runtime_error&) {  
        ptr = std::current_exception();  
    }  
    return;  
}  
  
void Watch()  
{  
    std::exception_ptr ptr;  
    // join immediately.  
    {std::jthread _{ Work, std::ref(ptr) }; }  
    if (ptr)  
        std::rethrow_exception(ptr);  
    std::cout << "All students over.\n";  
}  
  
int main()  
{  
    try {  
        Watch();  
    }  
    catch (const std::runtime_error& error) {  
        std::cout << error.what();  
    }  
    return 0;  
}
```

Continue to throw
the exception in
the main thread.

PKU is best.
THU is not best!

Static block variable

- Previously we may use static variables in function to share it across calls.
 - But is it safe to use in multiple threads?
- Yes and no...
 - Yes: only one thread will execute initialization and other threads will wait (since C++11).
 - No: to modify it across multiple threads, you still need lock.

```
void Foo(int id) {  
    // Thread-safe: initialization will be executed exactly once.  
    static std::map<int, int> lookupTable{};  
    // Not thread-safe, need lock protection.  
    lookupTable.emplace(1, 2);  
}
```

Once-for-all

- More generally, if we want to execute some segment of code only once across all threads...

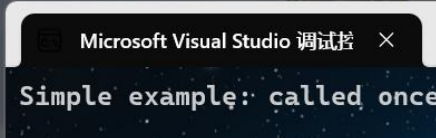
1. Tricks by static variable:

- This trick is often used in single-thread program too...

```
[[maybe_unused]] static int _ = []() {  
    HostUtils::CheckOptixError(optixInit());  
    return 0;  
}();
```

2. By `std::call_once` and `std::once_flag` defined in `<mutex>`:

```
std::once_flag flag1;  
  
void simple_do_once()  
{  
    std::call_once(flag1, []() { std::cout << "Simple example: called once\n"; });  
}  
  
int main()  
{  
    std::jthread st1{ simple_do_once }, st2{ simple_do_once },  
        st3{ simple_do_once }, st4{ simple_do_once };  
}
```



Once-for-all

Defined in header `<mutex>`

```
template< class Callable, class... Args >  
void call_once( std::once_flag& flag, Callable&& f, Args&&... args );
```

- `std::once_flag` only has a default ctor, meaning “not already called”.
 - And `std::call_once` will set the flag; exactly one thread will execute the callable and others will wait until it has completed.
- What’s the difference?
 1. `std::call_once` is slightly more flexible and intuitive;
 - But return value is ignored.
 2. Static variable trick may have slightly better performance.
 - See [stackoverflow](https://stackoverflow.com/questions/1649190/recursive-initialization-of-static-variables-is-undefined-behavior) for details.
 3. Recursive initialization for static variable is UB;
 - While `std::call_once` will lead to deadlock.

```
int Foo(int a, int b)  
{  
    static int m = Foo(a + 1, b + 1);  
    return m + 1;  
}
```

Once-for-all

- Sometimes we only want to share variables in calls of the current thread;
 - E.g. each thread has its own “static block variable”.
- We can use `thread_local` to specify thread storage duration!

This `static` can be omitted; `thread_local` block variables imply `static` if not specified.
[See C++ Standard.](#)

```
void Foo(int id) {  
    ....  
    // Each thread has its own lookupTable.  
    thread_local static std::map<int, int> lookupTable{};  
    lookupTable.emplace(1, 2);  
    lookupTable.find(id);  
}
```

- Of course, you can use in “global” variables...

```
thread_local int m = 0; // external linkage  
static thread_local int n = 0; // internal linkage  
class A  
{  
    static thread_local int k;  
};  
thread_local int A::k = 0;
```

Note that it's not allowed to write `static` here.

Once-for-all

- `thread_local` global variables are created after a thread starts, and destructed when it exits.
- Final word: if an exception throws out of:
 - Initialization of static block variables;
 - Function of `std::call_once`;
- Then it's seen as execution failure, and it will be initialized / executed again the next time.
 - So pay attention if there are other side effects that cannot be executed twice.

Multithreading

Synchronization Utilities

Multithreading

- Synchronization Utilities
 - semaphore
 - mutex & lock
 - condition variable
 - latch & barrier

Semaphore

```
void Inc(int& a) { for (int i = 0; i < 100000; i++) a++; }

int main()
{
    int a = 0;
    {std::jthread t1{ Inc, std::ref(a) }, t2{ Inc, std::ref(a) }; }
    std::cout << a;
    return 0;
}
```



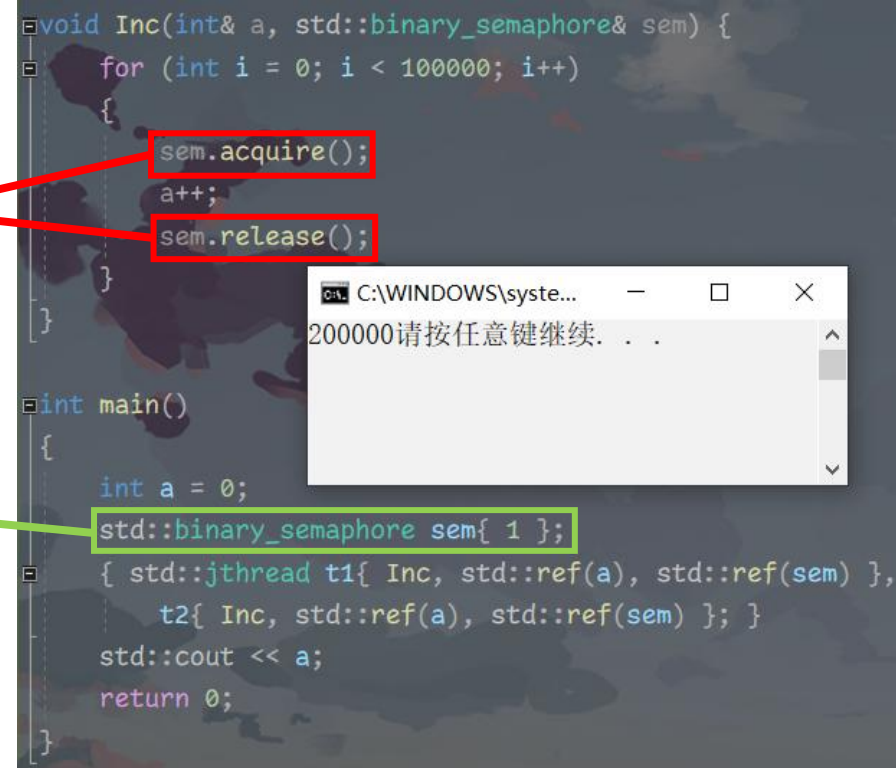
C:\WINDOWS\system32\cmd.exe
174568请按任意键继续. . .

- We've learnt data races in ICS.
 - When more than one thread access the same memory location concurrently, with at least one writing the memory, the final result is unexpected and unpredictable.
 - E.g. Two threads executing `a++` for 100000 times may not result in `a += 200000`.
 - We call region that may cause data races "critical section" (CS).
 - The most basic way is to guarantee only one thread will operate data!
- Semaphore, proposed by Dijkstra in 1962, is for that.
 - It has an initial integer state and two operations.
 - V, or up, or release, means that the state is increased by 1.
 - P, or down, or acquire, means that the state is decreased by 1.
 - The state cannot be negative; if it has reached 0, then wait until it's increased.
 - Two operations are exclusive and safe to execute concurrently.

Semaphore

Since the initial state is 1, only one thread can enter CS and `a++`.

- `<semaphore>` is introduced since C++20:
 - Ctor is just the initial integer state.
- More generally, you can use `std::counting_semaphore<LeastMaxValue>`.
 - And `std::binary_semaphore` is just `std::counting_semaphore<1>`.
 - Library may do optimization based on `LeastMaxValue`, e.g. when `LeastMaxValue <= 255`, use `std::uint8_t`; otherwise `std::uint32_t`; etc.
 - So the actual max possible value that can be represented by the semaphore is not necessarily `LeastMaxValue`;
 - You can use `static constexpr std::ptrdiff_t max()` to check the actual max possible value.



```
void Inc(int& a, std::binary_semaphore& sem) {  
    for (int i = 0; i < 100000; i++)  
    {  
        sem.acquire();  
        a++;  
        sem.release();  
    }  
}  
  
int main()  
{  
    int a = 0;  
    std::binary_semaphore sem{ 1 };  
    { std::jthread t1{ Inc, std::ref(a), std::ref(sem) },  
      t2{ Inc, std::ref(a), std::ref(sem) }; }  
    std::cout << a;  
    return 0;  
}
```

C:\WINDOWS\system...
200000请按任意键继续. . .

Semaphore

- Quite easy, so just list methods below:
 - Ctor: assign the initial integer state;
 - Not copyable, moveable, copy-assignable and move-assignable.
 - Acquire / Decrement:
 - `.acquire()`: decrease the semaphore; if it can't (i.e. current state is 0), wait.
 - `.try_acquire()`: decrease the semaphore and return `true`; if it can't, return `false`.
 - It's allowed to fail spuriously, i.e. the current state is not 0 but `try_acquire` still returns `false`.
 - `.try_acquire_for/until(...)`: try to decrease the semaphore for at most a period of time; if it fails, return `false`.
 - Release / Increment:
 - `.release(std::ptrdiff_t n = 1)`: increase the semaphore by n.
 - Dtor: when destructing, no thread should wait on the semaphore (UB).

Multithreading

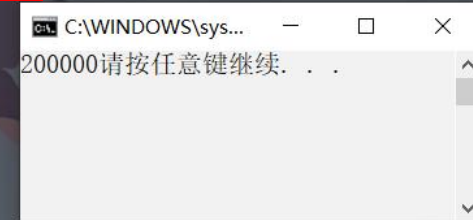
- Synchronization Utilities
 - semaphore
 - mutex & lock
 - condition variable
 - latch & barrier

Mutex

- **Mutual exclusion** means to make threads do some operations exclusively.
 - Kind of like binary semaphore.
 - For our ++ example, you can also write like this:
- Again, very easy APIs, just list here.
 - Default ctor, meaning “unlock” state.
 - Not copyable, moveable, assignable.
 - `.lock()/unlock()`;
 - `.try_lock()`;
 - No time-related APIs, e.g. `.try_lock_for`.
 - Dtor: when destructing, no thread should wait on the mutex.
 - `.native_handle()`: optional and platform-dependent, e.g. return `pthread_mutex_t` in POSIX system.

```
void Inc(int& a, std::mutex& sem) {  
    for (int i = 0; i < 100000; i++)  
    {  
        sem.lock();  
        a++;  
        sem.unlock();  
    }  
}  
  
int main()  
{  
    int a = 0;  
    std::mutex sem;  
    { std::jthread t1{ Inc, std::ref(a), std::ref(sem) },  
      t2{ Inc, std::ref(a), std::ref(sem) }; }  
    std::cout << a;  
    return 0;  
}
```

In <mutex>



Mutex

- So what's the difference between mutex and semaphore?
 1. Semaphore can be initialized with an integer state that can be 0 or more than 1, but mutex is always like binary semaphore with initial state as 1.
 - The up-down parameter of semaphore can also be inconsistent, e.g. you can down 1 but up 3.
 2. Semaphore can be released by a thread that doesn't acquire it, but **mutex must be unlocked by the thread that locks it.**
 - Thus, mutex is **exactly used for mutual exclusion**;
 - But semaphores can be used in other ways, like act as an efficient condition variable sometimes.

You'll learn how to implement semaphore and mutex in OS course.

Mutex

- When a mutex is locked twice in the same thread, then UB.
 - A typical behavior is deadlock.
 - The implementation is encouraged to throw exception `std::system_error` with error code `resource_deadlock_would_occur`.
- If you want to lock for multiple times in the locking thread, you can use `std::recursive_mutex`.
 - You need to unlock for a matching time to make it really unlocks.
 - APIs are completely same as `std::mutex`, not listed again.
 - For example:

```
class Bar
{
public:
    void Work1()
    {
        m.lock();
        /* some work */
        m.unlock();
    }

    void Work2()
    {
        m.lock();
        Work1();
        /* other work */
        m.unlock();
    }
private:
    std::recursive_mutex m;
};
```

*Normally code that uses recursive mutex can be transformed to only use simple mutex in some way.

**Note that usually mutex is `mutable` data member to lock & unlock in `const` methods.

Lock

- Mutex doesn't obey RAII...
 - If you return in many paths (including possible exceptions), it's miserable and dangerous to write `unlock` yourself.
 - A better way is to lock mutex in ctor, and unlock it in dtor.
- Lock in C++ is for that!
 - The most basic one is `std::lock_guard<MutexType>` in `<mutex>`:
 - For provided `MutexType` in ctor, it will call `.lock()`;
 - And `.unlock()` it in dtor.

Note:

1. Here CTAD so `MutexType` can be omitted.
2. `_` is necessary, otherwise it's a temporary and will be destructed (i.e. unlocked) immediately after this statement.

```
void Inc(int& a, std::mutex& mut) {  
    for (int i = 0; i < 100000; i++)  
    {  
        std::lock_guard _{ mut };  
        a++;  
    }  
}
```

Lock

- Note 1: `if` initializer can be used if you want to lock the mutex in an `if` clause:
 - It will lock for the whole clause, e.g. including `else` here.

```
if (std::lock_guard_{ mut }; someCond)
    a++;
else
    a--;
```

- Note 2: `std::lock_guard` doesn't have any other APIs; you cannot unlock by `lock_guard` until destruction.
- Note 3: if you want to use `std::lock_guard` to manage a mutex that's already locked, add a tag parameter `std::adopt_lock`.
 - A naïve example:

```
std::recursive_mutex m;
m.lock();
std::lock_guard_{ m, std::adopt_lock };
```

Lock

- A more general RAII type is `std::unique_lock`.
 - It “owns” some mutex, and can also give up this ownership.
 - Specifically, this ownership has two aspects:
 - A. Have (associate with) a mutex or not;
 - B. Lock a mutex or not.
- Take ctor as example:
 - Default ctor: not have a mutex, no lock → !A & !B
 - `(MutexType& m)`: takes a mutex, and lock it (call `m.lock()`) → A & B
 - `(MutexType& m, std::adopt_lock)`: takes a locked mutex → A & B
 - `(MutexType& m, std::defer_lock)`: takes a mutex, but not lock it → A & !B
 - `(MutexType& m, std::try_to_lock)`: call `m.try_lock()`;
 - If succeed → A & B; otherwise → A & !B.

Lock

- Dtor: A & B → `.unlock()` in dtor; otherwise do nothing.
- When `std::unique_lock` have a mutex...
 - If you have locked it, you can call...
 - `.unlock()`.
 - If you haven't locked it, you can call...
 - `.lock()`.
 - `.try_lock()`.
 - `.release()`: disassociate with the mutex (without calling `.unlock()`).
- When `std::unique_lock` doesn't have a mutex...
 - All locking operations, e.g. `.unlock/lock/try_lock()`, will throw exception.
 - You can use move assignment / swap to associate with a new mutex;
 - Move assignment equiv. to call dtor + steal the new state (move ctor).

Lock

- You can also observe the ownership:
 - `.mutex()`: return pointer to the mutex; if no associated mutex, `nullptr`;
 - `.owns_lock()` / `operator bool()`: check whether it has a mutex and has locked it (A & B).
- For example:

```
auto worker_task = [&](int id)
{
    std::unique_lock<std::mutex> lock(counter_mutex);
    ++counter;
    std::cout << id << ", initial counter: " << counter << '\n';
    lock.unlock();

    // don't hold the lock while we simulate an expensive operation
    std::this_thread::sleep_for(std::chrono::seconds(1));

    lock.lock();
    ++counter;
    std::cout << id << ", final counter: " << counter << '\n';
};

for (int i = 0; i < 10; ++i)
    threads.emplace_back(worker_task, i);

for (auto& thread : threads)
    thread.join();
```

Lock

- Another example:

We'll cover
`std::lock` later; it
locks two locks here.

```
void transfer(Box& from, Box& to, int num)
{
    // don't actually take the locks yet
    std::unique_lock lock1{from.m, std::defer_lock};
    std::unique_lock lock2{to.m, std::defer_lock};

    // lock both unique_locks without deadlock
    std::lock(lock1, lock2);

    from.num_things -= num;
    to.num_things += num;

    // "from.m" and "to.m" mutexes unlocked in unique_lock dtors
}
```

Lock

	!A & !B (No mutex)	A & !B (Have mutex, not locked)	A & B (Have mutex, locked)
.lock/try_lock()	<code>std::system_error{ operation_not_permi tted }</code>	→ A & B	<code>std::system_error{ resource_deadlock_w ould_occur}</code>
.unlock()	<code>std::system_error{ operation_not_permi tted }</code>	<code>std::system_error{ operation_not_permi tted }</code>	→ A & !B
.release()	→ !A & !B	→ !A & !B	→ !A & !B
.mutex()	<code>nullptr</code>	Pointer to mutex	Pointer to mutex
.owns_lock/operator bool()	<code>false</code>	<code>false</code>	<code>true</code>

We omit possible exceptions thrown by e.g. `mutex.lock()` here.

Shared mutex

- All mutexes before lead to exclusive access in CS.
 - But in fact only write is needed to be exclusive; read can be performed simultaneously.
 - Also known as “Readers-Writers Problem”.
- Shared mutex since C++17 is for that...
 - `std::shared_mutex` in `<shared_mutex>`;
 - Two modes – exclusive mode and shared mode.
 - For exclusive mode, APIs and functionalities are same as `std::mutex`.
 - Mutex can only be held by one thread; other threads will wait until unlock.
 - For shared mode, `.lock_shared()`, `.unlock_shared()` and `.try_lock_shared()` are provided.
 - Mutex can be held by multiple threads; threads that want to hold mutex in exclusive mode will be blocked until all sharing threads unlock.

Unmatched lock & unlock (e.g. lock in exclusive mode but unlock in shared mode) is UB.

Shared lock

- Correspondingly, `std::shared_lock` in `<shared_mutex>` is provided for RAII (since C++14, before `std::shared_mutex` is accepted).
 - All APIs are completely same as `std::unique_lock`; the only difference is that functions that call `mutex.lock()/unlock()/try_lock()` will call `mutex.lock_shared()/unlock_shared()/try_lock_shared()` instead.
- So, to lock `std::shared_mutex` in exclusive mode with RAII...
 - Just use `std::unique_lock _{ mut }` or `std::lock_guard _{ mut }`;
- And, to lock `std::shared_mutex` in shared mode with RAII...
 - Just use `std::shared_lock _{ mut }`.

Shared lock

- A naïve example:

```
class Database
{
public:
    int Read()
    {
        std::shared_lock sharedLock{ mutex_ };
        std::this_thread::sleep_for(100ms); // Assuming we need 100ms to read
        return data_;
    }

    void Write(int d)
    {
        std::lock_guard exclusiveLock{ mutex_ };
        std::this_thread::sleep_for(300ms); // Assuming we need 300ms to write.
        data_ = d;
    }

private:
    std::shared_mutex mutex_;
    int data_ = 0;
};
```



Shared mutex implementation*

- This part is **optional**.
- In ICS, we've learnt starvation problem...
 - Writer starvation: if we prioritize read, then when there are lots of readers, mutex will be always held in shared mode.
 - Before a reader finishes its read, new readers will come and thus writers block forever.
 - Reader starvation: similarly, if we prioritize write, then when there are lots of writers, mutex will be always held in exclusive mode.
- So in some algorithms, you can specify "priority" for lock.
 - But it seems that C++ standard library doesn't do so!
- A reference implementation is proposed by [Howard E. Hinnant](#), which is invented by Alexander Terekhov.

Sorry, but I can't find the exact blog / paper /... of Alexander Terekhov.
It seems to be invented during pthread-win32 development around 2003.

Shared mutex implementation*

- This algorithm makes it “fair” for reader and writer.
- To put it simply, two gates:
 - Gate 1: reader and writer compete fairly;
 - When readers pass through gate 1, they can get mutex in shared mode.
 - When a writer pass through gate 1, it needs to wait for gate 2.
 - Gate 2: when a writer waits on gate 2, gate 1 should block all requests;
 - And the writer will wait until all readers that have already passed through gate 1 to unlock.
 - Then the writer will get mutex in exclusive mode.
 - After the writer unlocks, gate 1 will accept requests again.
- libcpp uses this algo.; libstdc++ has a macro to control (either this algo., or native `pthread_rwlock_t`); MS-STL uses native [SRW](#).

A single SRW lock can be acquired in either mode; reader threads can acquire it in shared mode whereas writer threads can acquire it in exclusive mode. There is no guarantee about the order in which threads that request ownership will be granted ownership; SRW locks are neither fair nor FIFO.

[But SRW has OS bug](#), which may grant exclusive ownership in shared mode. It's fixed by in 2024.5.9 but which Windows update includes this fix is unknown.

Safe Reclamation*

- Shared mutex makes it possible to read concurrently, but still grant write with mutual exclusion.
 - Is it possible to make read and write happen “concurrently”?
- By Read-Copy-Update (RCU) or hazard pointer!
 - To put it simply, RCU allows concurrent write by allocating a new version, so previous read can still go on.
 - Old versions will be freed when no readers are on it.
 - And hazard pointer registers “it’s hazard to delete some version” by other techniques, making it protected from use-after-free problem.
- C++26 introduces them by `<rcu>` and `<hazard_pointer>`.
 - Quite complex so not discussed here.

Deadlock

- Finally, let's talk about deadlock.
 - Of course, what we talk about is not deadlock caused by locking for multiple times in the same thread.
 - You can use `std::recursive_mutex` to solve that.
 - A typical example: lock in different orders in different threads.

```
void Worker1(std::mutex& mut1, std::mutex& mut2)
{
    std::lock_guard _1{ mut1 };
    std::this_thread::sleep_for(1s);
    std::lock_guard _2{ mut2 };
}

void Worker2(std::mutex &mut1, std::mutex &mut2)
{
    std::lock_guard _2{ mut2 };
    std::this_thread::sleep_for(1s);
    std::lock_guard _1{ mut1 };
}
```

```
int main()
{
    std::mutex mut1, mut2;
    std::jthread t1{ Worker1, std::ref(mut1), std::ref(mut2) },
        t2{ Worker2, std::ref(mut1), std::ref(mut2) };
}
```

Worker1 holds mut1, waits for mut2;
But Worker2 holds mut2, waits for mut1.

Deadlock

- Deadlock happens only when these four conditions are true:
 1. Mutual exclusion, i.e. only limited threads can use the resource.
 2. Hold and wait, i.e. a thread will hold some resource, while requesting other resources.
 3. No preemption, i.e. resources are only given up voluntarily, but a thread never does that; and other threads cannot force that thread to give up.
 4. Circular wait, i.e. hold & wait sequence has a circle.
- To prevent deadlock, we need to break one of these conditions.
 - 1. is usually not easy to break, because it's determined by the property of shared data.
 - 2. is also necessary when using multiple locks; if you can use only one lock, then that's fine.

Deadlock

- Generally, there are three practical ways to solve deadlock:
 - Deadlock prevention: pre-design so that deadlock will not happen.
 - Like locking all mutex in the same order;
 - Deadlock recovery: just let deadlock happen; but when a resource is occupied for a too long time, the owner will be forced to give up.
 - This breaks “3. No preemption”.
 - Deadlock avoidance: pre-check whether subsequent resource requests will possibly cause deadlock and dynamically delay them if necessary.
 - This breaks “4. Circular wait”, i.e. if circular wait is possible, then it will delay following requests.
 - There are many delicate algorithms for deadlock avoidance.
- C++ wraps deadlock avoidance as function `std::lock`.

Deadlock Avoidance

- For example:

```
class MyMutex : public std::mutex
{
public:
    void lock()
    {
        std::this_thread::sleep_for(1s); // To make deadlock definitely happen.
        std::mutex::lock();
    }
};

void WorkerDeadlock(int id, MyMutex& mut1, MyMutex& mut2)
{
    std::println("Begin: {}", id);
    std::lock_guard _1{ mut1 }, _2{ mut2 };
    std::println("End: {}", id);
}
```

Deadlock for reversed lock order.

```
MyMutex mut1, mut2;
// Pass them in reversed order
std::jthread t1{ WorkerDeadlock, 0, std::ref(mut1), std::ref(mut2) },
               t2{ WorkerDeadlock, 1, std::ref(mut2), std::ref(mut1) };
```

```
Begin: 1
Begin: 0
```

Deadlock Avoidance

- If we use `std::lock(...)`:
 - Notice that mutexes are still passed into `std::lock` in reversed order.

```
void WorkerNoDeadlock(int id, MyMutex& mut1, MyMutex& mut2)
{
    std::println("Begin: {}", id);
    std::lock(mut1, mut2);
    std::lock_guard _1{ mut1, std::adopt_lock }, _2{ mut2, std::adopt_lock };
    std::println("End: {}", id);
}
```

```
Begin: 0
Begin: 1
End: 1
End: 0
```

- So now you can understand this piece of code...
 - Mutex or `unique_lock` are fine since both of them provides `.lock/unlock/try_lock()`; `lock_guard` cannot be passed in.

```
void transfer(Box& from, Box& to, int num)
{
    // don't actually take the locks yet
    std::unique_lock lock1{from.m, std::defer_lock};
    std::unique_lock lock2{to.m, std::defer_lock};

    // lock both unique_locks without deadlock
    std::lock(lock1, lock2);

    from.num_things -= num;
    to.num_things += num;

    // "from.m" and "to.m" mutexes unlocked in unique_lock dtors
}
```

Deadlock Avoidance

- Since C++17, `std::scoped_lock` is also provided to do deadlock avoidance with RAII.

```
void WorkerNoDeadlock(int id, MyMutex& mut1, MyMutex& mut2)
{
    ....
    std::println("Begin: {}", id);
    std::scoped_lock _{ mut1, mut2 };
    std::println("End: {}", id);
}
```

- Similar to `std::lock_guard`, only ctor and dtor. Ctor actually calls `std::lock`.
 - But overload with `std::adopt_lock` is the first param, since variadic arguments can only be the last.

`std::scoped_lock<MutexTypes...>::scoped_lock`

`explicit scoped_lock(MutexTypes&... m);` (1) (since C++17)

`scoped_lock(std::adopt_lock_t, MutexTypes&... m);` (2) (since C++17)

Deadlock Avoidance

- Note 1: multiple `std::lock/scoped_lock` can still lead to deadlock; only when all locks are provided in one instance can the avoidance algorithm work.

```
std::scoped_lock _{ mut1 };  
std::scoped_lock _{ mut2 };
```



- Note 2: there also exists function `std::try_lock`, which isn't deadlock avoidance but just an all-or-nothing wrapper.
 - Provided locks are locked according to parameter order exactly;
 - Any failure to `.try_lock()` (including thrown exception) means overall failure, which will unlock previous locks.
 - Only when all locks are locked will it be seen as success.
 - Return the index of mutex that fails to lock, or -1 if success.
 - Equivalent to (ignoring exception):

```
// for each mutex:  
//     if(!mut.try_lock())  
//         unlocks previous locks  
//     return index  
// return -1
```

Timed mutex

- Back to deadlock recovery...
 - We can also let owners give up their resources voluntarily when waiting for a too long time.
 - That is, we need mechanisms like semaphore `.try_acquire_for/until()`...
- Timed mutex is provided for that!
 - All mutex types have a timed version: `std::timed_mutex`, `std::recursive_timed_mutex` and `std::shared_timed_mutex`.
 - Normally timed mutex has slightly higher cost than normal ones.
 - Note that `std::shared_timed_mutex` is introduced in C++14, and [for performance](#) `std::shared_mutex` is introduced in C++17.
 - Beyond original APIs, they also add timed ones.
 - i.e. `.try_lock_for/until()`; `.try_lock_shared_for/until()`.

Timed mutex

- And `std::unique_lock` & `std::shared_lock` will also have `.try_lock_for/until()` when provided mutex type is timed.

- And additional ctor overload:

```
template< class Rep, class Period >  
shared_lock( mutex_type& m,  
             const std::chrono::duration<Rep,Period>& timeout_duration );  
  
template< class Clock, class Duration >  
shared_lock( mutex_type& m,  
             const std::chrono::time_point<Clock,Duration>& timeout_time );
```

- Note again `std::shared_lock::try_lock_for()` calls `MutexType::try_lock_shared_for()`.
- For example, to do deadlock recovery:

```
class MyMutex : public std::timed_mutex  
{  
public:  
    void lock()  
    {  
        std::timed_mutex::lock();  
        std::this_thread::sleep_for(1s);  
    }  
};
```


Deadlock Recovery

```
Begin: 1
Begin: 0
Release resources and try again: 0.
Release resources and try again: 1.
End: 0
End: 1
```

Two segments are basically equivalent here.

```
void WorkerDeadlockRecovery(int id, MyMutex& mut1, MyMutex& mut2)
{
    std::println("Begin: {}", id);
    while (true)
    {
        std::unique_lock lock1{ mut1 };
        // Wait for a random time between 0 ~ 1s.
        std::unique_lock lock2{ mut2, GetRandomNumber() * 1s };
        if (lock2.owns_lock())
        {
            std::println("End: {}", id);
            break;
        }
        else {
            std::println("Release resources and try again: {}.", id);
            lock1.unlock();
            std::this_thread::sleep_for(GetRandomNumber() * 1s);
        }
    }
}
```

```
void WorkerDeadlockRecovery2(int id, MyMutex& mut1, MyMutex& mut2)
{
    std::println("Begin: {}", id);
    std::unique_lock lock1{ mut1, std::defer_lock },
        lock2{ mut2, std::defer_lock };

    while (true)
    {
        lock1.lock();
        // Wait for a random time between 0 ~ 1s.
        if (lock2.try_lock_for(GetRandomNumber() * 1s))
        {
            std::println("End: {}", id);
            break;
        }
        else {
            std::println("Release resources and try again: {}.", id);
            lock1.unlock();
            std::this_thread::sleep_for(GetRandomNumber() * 1s);
        }
    }
}
```


Multithreading

- Synchronization Utilities
 - semaphore
 - mutex & lock
 - condition variable
 - latch & barrier

Condition Variable

- Besides mutual exclusion, concurrent programming also needs synchronization.
 - That is, ***after some condition has met***, I will do something...
- If only mutual exclusion is provided, then we have to busy-wait:
 - Low efficiency, high CPU consumption.
- Or we may do “not-that-busy-wait”:
 - But what’s the ideal time for sleep?
 - Too short: ~ busy-wait
 - Too long: low throughput.
 - It’s better to be determined by OS, or noticed by the condition modifier...

```
while (true)
{
    std::lock_guard _{ mutex };
    if (cond)
        break;
}
```

```
while (true)
{
    // sleep after a try
    std::this_thread::sleep_for(100ms);
    std::lock_guard _{ mutex };
    if (cond)
        break;
}
```

Condition Variable

- Condition variable is a primitive for lazy-wait synchronization.
 - It needs a predicate to judge condition;
 - And a lock to ensure mutual exclusion when calling predicate.
- And it has two operations:
 - **Wait**, i.e. wait on some condition to satisfy.
 1. Before waiting, the current thread should hold the lock.
 2. When beginning to wait, it will release the lock;
 3. Once it's unblocked, the lock will be acquired to check condition safely;
 4. If the condition is not satisfied, back to step 2; otherwise end wait.
 - So condition variable will always hold the mutex, except for blocking in step 2.
 - **Notify** (or *Signal*), i.e. notify the waiting threads that the condition may have been satisfied.
 - So the waiting condition variable will enter step 3 and check it.

Condition variable is also called Monitor (管程) in some languages / materials.

Condition Variable

- In C++, we may just use `std::condition_variable` defined in `<condition_variable>`.
 - Wait:
 - `.wait(std::unique_lock<std::mutex>& lock);`
 - `.wait_for/until(lock, time) -> std::cv_status;`
 - Either `std::cv_status::no_timeout` or `std::cv_status::timeout`.
 - Notify:
 - `.notify_one()`: wake up one of waiting threads;
 - `.notify_all()`: wake up all waiting threads (also called **Broadcast**).
- For example, to start task after initialization completes:

```
std::condition_variable condVar;  
std::mutex mutex;  
bool complete = false;
```

You'll learn how to implement condition variable in your OS course.

- For the waiting thread:

```
void work()
{
    DoPreparation();
    // wait until initialization ends.
    std::unique_lock lock{ mutex };
    while (!complete)
        condVar.wait(lock);
    // Now we still holds mutex in lock!
}
```

1. First hold the mutex to make it safe to check condition;
2. Blocking will call `.unlock()`, to let modifiers change condition;
3. Unblocking will `.lock()`, then check condition again;
4. When condition has met, break loop (Now mutex is still locked).

- For the notifying thread:

```
int main()
{
    std::jthread thread{ work };
    DoInitialization();
    {
        std::lock_guard _{ mutex };
        complete = true;
    }
    condVar.notify_one();
    return 0;
}
```

1. Hold the mutex and modify condition;
2. Notify the waiting thread to check the condition.

Condition access is always mutual exclusive, protected by mutex.

Condition Variable

```
void Work()
{
    DoPreparation();
    // wait until initialization ends.
    std::unique_lock lock{ mutex };
    while (!complete)
        condVar.wait(lock);
    // Now we still holds mutex in lock!
}
```

- So why do we need a loop instead of `if (!complete)`?
 - It seems that there are only two outcomes:
 - `complete`, then initialization has been done and just continue;
 - `!complete`, then
 - After variable is notified (and then unblocked), condition has been met so we just continue too;
 - It seems enough to check condition once.
- Two Reasons:
 1. In more complex scenarios, condition may be modified by many.
 - Waiting thread will not be scheduled immediately^[1], so before it's scheduled, condition may not be true again (also called "ABA problem").
 2. C++ allows condition variable to wake up spuriously.
 - Even when it's not notified, condition variable may be unblocked.
 - Here it means when `complete` is still `false`, `.wait()` may return.

[1]: To be exact, such behavior is called Mesa monitor; Immediate schedule is called Hoare monitor, which is usually not implemented by current OS since it's hard and expensive.

Condition Variable

- So in C++, condition variable should almost always cooperate with a loop to ensure some condition has been met.

- Thus they are wrapped in overloads of `wait` directly:

- `.wait(lock, condPred)`, wait until `condPred` is satisfied;

- Equiv. to:

```
while (!pred())  
    wait(lock);
```

 So in `condPred`, lock is already locked.

- `.wait_for/until(lock, time, condPred) -> bool`;

- Equiv. to:

```
while (!pred())  
    if (wait_until(lock, abs_time) == std::cv_status::timeout)  
        return pred();  
return true;
```

- For example:

```
void work()  
{  
    DoPreparation();  
    std::unique_lock lock{ mutex };  
    condVar.wait(lock, [](){ return complete; });  
    // Now we still holds mutex in lock!  
}
```

Condition Variable

- Note 1: you should not rely on spurious wakeup; always notify if condition has met.
 - My experience on an embedded device (Raspberry Pi) shows it never wakes up unless it's notified; but my Windows PC will do so.
- Note 2: it's normally better to unlock before notification.
 - Reason: if we notify before unlocking, then it may be:
 1. Waiting thread is waken up (for OS scheduling);
 2. It tries to lock the mutex, but fails and thus blocks again.
 3. Notifying thread unlocks the mutex;
 4. Waiting thread is waken up, and finally can check condition.
 - But if we unlock before notifying, it gives waiting thread a chance to wake up and hold the mutex directly, without blocking again.

```
{  
    std::lock_guard _{ mutex };  
    complete = true;  
    // Instead of .notify_one() here.  
}  
condVar.notify_one();
```


Condition Variable

- Note 3: condition **must** be modified with locked mutex, even if not doing so is data-race-free (e.g. atomic variable).
 - For example: `std::atomic<bool> complete{ false };`
 - Then the process may be:
 1. `!complete` has been checked by waiting thread;
 2. `complete` is set and `condVar` is notified by notifying thread;
 3. `condVar` waits (thus misses notification and possibly never wakes up!).
 - When the mutex is held, step 1 and step 3 are indivisible (and thus step 2 cannot kick in), making it definitely correct.

The diagram illustrates the execution flow of a condition variable. It features two code snippets. The first snippet, enclosed in a blue box, contains the lines `complete = true;` and `condVar.notify_one();`. The second snippet, enclosed in a grey box, contains `std::unique_lock lock{ mutex };`, `while (!complete)` (highlighted with a red box), and `condVar.wait(lock);` (highlighted with a green box). A red arrow points from the `while (!complete)` line to the first step of the process list. A blue arrow points from the `condVar.notify_one();` line to the second step. A green arrow points from the `condVar.wait(lock);` line to the third step.

```
complete = true;
condVar.notify_one();
```

```
std::unique_lock lock{ mutex };
while (!complete)
    condVar.wait(lock);
```

Condition Variable

- Note 4: finally list omitted APIs:
 - Only default ctor, not copyable, moveable and assignable.

Error conditions:

— `resource_unavailable_try_again` — if some non-memory resource limitation prevents initialization.

- `.native_handle()`;
- Dtor: no one should wait on destructing cv.

`~condition_variable()`;

Preconditions: There is no thread blocked on `*this`.

[*Note 1:* That is, all threads have been notified; they can subsequently block on the lock specified in the wait. This relaxes the usual rules, which would have required all wait calls to happen before destruction. Only the notification to unblock the wait needs to happen before destruction. Undefined behavior ensues if a thread waits on `*this` once the destructor has been started, especially when the waiting threads are calling the wait functions in a loop or using the overloads of `wait`, `wait_for`, or `wait_until` that take a predicate. — *end note*]

Condition Variable

- `std::condition_variable` can only wait on `std::unique_lock<std::mutex>` to maximize efficiency.
 - Sometimes we may need a more general condition variable to wait on any lockable object...
 - Then `std::condition_variable_any` can be used!
 - The only difference in API is that wait is template function to accept any possible lock:

```
template< class Lock >  
void wait( Lock& lock );
```
- For example, to write a simple system tick simulator...
 - That is, tick is increased periodically and globally;
 - For threads that call `Sleep(tickNum)`, they will be blocked unless `tickNum` has passed.

Condition Variable

- It seems better to use `std::shared_mutex`, since then all threads can read the tick simultaneously.

```
std::condition_variable_any condVar;  
std::shared_mutex mutex;  
std::uint64_t tick = 0;
```

```
void Sleep(std::uint64_t tickNum)  
{  
    std::shared_lock lock{ mutex };  
    auto wakeupTick = tick + tickNum;  
    condVar.wait(lock, [wakeupTick]() {  
        // Sleep until current tick has exceeded limit.  
        return tick >= wakeupTick;  
    });  
}  
  
void Work(int id, int loopNum, std::uint64_t sleepTick)  
{  
    for (int i = 0; i < loopNum; i++)  
    {  
        Sleep(sleepTick);  
        std::println("Hello from {}", id);  
    }  
}
```

```
for (int i = 0; i < 50; i++)  
{  
    {  
        std::lock_guard lock{ mutex };  
        ++tick;  
        std::println("Now it's tick {}", tick);  
    }  
    condVar.notify_all();  
    std::this_thread::sleep_for(100ms);  
}
```

Condition Variable

- Since C++20, `std::condition_variable_any` also adds stop token handling.
 - `.wait/wait_for/wait_until(lock, stop_token, condPred, ...) -> bool`.
 - They also check whether stop is requested;
 - And it will also register its notification on the stop state, so when stop is requested, it will be notified automatically (and then exits `.wait`).
- Note: `std::condition_variable` doesn't introduce stop token handling since it's impossible (with the same optimization).
 - See [answer of Anthony Williams](#) for details.

```
while (!token.stop_requested())  
{  
    if (pred())  
        return true;  
    wait(lock);  
}  
return pred();
```

Condition Variable

We'll talk about more `xx_at_thread_exit` methods later.

- Finally, if you want to notify other threads that the current thread has exited, you can use `std::notify_all_at_thread_exit(std::condition_variable& cond, std::unique_lock<std::mutex> lock)`.
 - `xx_at_thread_exit` is called after all other things end in the current thread, even **after destruction of thread_local variables**.
 - Then `lock.unlock()`, and then `cond.notify_all()`.

```
void thread_func()
{
    thread_local std::string thread_local_data = "42";

    std::unique_lock<std::mutex> lk(m);

    // assign a value to result using thread_local data
    result = thread_local_data;
    ready = true;

    std::notify_all_at_thread_exit(cv, std::move(lk));
} // 1. destroy thread_locals;
  // 2. unlock mutex;
  // 3. notify cv.
```

```
int main()
{
    std::thread t(thread_func);
    t.detach();

    // do other work
    // ...

    // wait for the detached thread
    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk, []{ return ready; });

    // result is ready and thread_local destructors have finished
    assert(result == "42");
}
```

Multithreading

- Synchronization Utilities
 - semaphore
 - mutex & lock
 - condition variable
 - latch & barrier

Latch

In `<latch>`,
since C++20.

- Latch is a ***one-shot*** synchronization mechanism.
 - It has an initial integer state, and supports two operations:
 - Arrive, which will decrease the integer state;
 - `.count_down(std::ptrdiff n = 1);`
 - Wait, which will wait until the integer state reaches 0.
 - `.wait/try_wait();` `try_wait` is allowed to return `false` spuriously.
 - You can also combine these two operations by `.arrive_and_wait(n = 1)`, which first `count_down` by `n` and then waits.
- For example, we call multiple threads to do initialization...
 - Only when all initializations are done can these threads go ahead safely.
 - And main thread should also wait on initializations.
 - So every thread can arrive and wait, and the main thread should only wait.

Latch

- So we may code like:

```
int main()
{
    int threadNum = 4;
    std::vector<std::jthread> threads;

    std::latch initializationDone{ threadNum };
    for (int i = 0; i < threadNum; ++i)
        threads.emplace_back(Work, i, std::ref(initializationDone));
    initializationDone.wait();
    DoMoreStuff();

    return 0;
}
```

If working thread only relies on its own initialization, then you can call `.count_down()` so only main thread will wait all initializations.

```
void Work(int id, std::latch& latch)
{
    ....
    Initialize(id);
    latch.arrive_and_wait();
    DoMoreStuff(id);
}
```

Only when all of four threads arrive will the state goes to 0, so that all threads are unblocked.

Barrier

- But sometimes we may repeat actions in multiple epochs, which then needs to “reuse a latch”.
- Barrier is a **reusable** synchronization mechanism.
 - Similarly, it has an initial integer state, and supports:
 - Arrive, which will decrease the integer state;
 - `.arrive(std::ptrdiff n = 1);`
 - Wait, which will wait until the integer state reaches 0.
 - `.wait()` (no `try_wait`);
 - You can also combine these two operations by `.arrive_and_wait(n = 1)`.
- When the integer state reaches 0, we say a *phase* has completed;
 - All waiting threads will be unblocked, and the integer state will be restored so synchronization of the next phase can happen.

In `<barrier>`,
since C++20.

Barrier

- When a phase has ended, barrier can also do some callback.
 - Passing by value in ctor; the default callback is doing nothing.
 - And callback must be labeled as **noexcept**.
 - Callback happens before unblocking (i.e. beginning of the next phase).

So `std::barrier` is actually a template class.

```
std::barrier<CompletionFunction>::barrier
```

```
constexpr explicit barrier( std::ptrdiff_t expected,  
                           CompletionFunction f = CompletionFunction());
```

- And threads can also choose to quit the next phase:
 - **.arrive_and_drop()**: decrement both the state and its initial value by 1.
- A toy example (just to show usage): print multiplication table.

Barrier

```
int main()
{
    std::println("\t{}", FormatLine(std::views::iota(1, 10)));
    int threadNum = 9;
    std::vector<std::jthread> threads;

    for (int i = 1; i <= threadNum; ++i)
        threads.emplace_back(Work, i);
    return 0;
}
```

	1	2	3	4	5	6	7	8	9
9	9	18	27	36	45	54	63	72	81
8	8	16	24	32	40	48	56	64	
7	7	14	21	28	35	42	49		
6	6	12	18	24	30	36			
5	5	10	15	20	25				
4	4	8	12	16					
3	3	6	9						
2	2	4							
1	1								

Barrier

- And finally list some omitted APIs (for both `std::latch` and `std::barrier`):
 1. Only default ctor, not copyable/moveable/assignable.
 2. Dtor: no thread should wait.
 3. `static constexpr std::ptrdiff_t max()` can be used to check the supported maximum integer state value.
 4. It's UB to set the state to be `< 0` (including decreased to `< 0`), or `> max()`.

Multithreading

High-Level Abstraction of Asynchronous Operations

Multithreading

- High-level abstraction of asynchronous operations
 - Future-Promise model
 - `packaged_task`
 - `async`

Future-promise model

- Sometimes we may find it too low-level to use `std::thread...`
 - No return value, complex exception management, only by reference parameter;
 - And if return value is set early (e.g. threads need to do some additional cleanup, but the return value has been prepared), the main thread requires to do synchronization to get it correctly.
- Instead, we want to screen these details and prefer a task-based view...
 - That is, we designate a task to the thread, and hope to get the result in the future, without caring when the task finishes or whether we need to wait for it.

Future-promise model

- This is what future-promise model provides.
 - “future” represents task producer, meaning that it expects some result “in the **future**”.
 - “promise” represents task worker, meaning that it **promises** to provide some result.
- To be specific, future-promise model is a **one-shot** channel.
 - The promise end works and transports the result to the future end;
 - Result can be set and got only once before the channel is destroyed.

• For example:

```
std::mutex mutex;  
std::condition_variable condVar;  
bool ready = false;
```

```
void work(int arg, int& result)  
{  
    {  
        std::lock_guard _{ mutex };  
        result = calculate(arg);  
        ready = true;  
    }  
    condVar.notify_one();  
    DoCleanup();  
}
```

```
void Launch()  
{  
    int result = 0;  
    std::jthread t{ work, 1, std::ref(result) };  
    std::unique_lock lock{ mutex };  
    condVar.wait(mutex, []{ return ready; });  
    // Then we can use result safely...  
}
```

Very complex
to implement
with basic
primitives...

Future-promise model

- If we use `std::future<T>` and `std::promise<T>` in `<future>`:

```
void Launch()  
{
```

```
    std::promise<int> promise;  
    auto future = promise.get_future();
```

```
    std::jthread t{ work, 1, std::move(promise) };  
    auto result = future.get();  
    // Then we can use result safely...
```

```
}
```

```
void work(int arg, std::promise<int> promise)
```

```
{
```

```
    promise.set_value(Calculate(arg));  
    DoCleanup();  
}
```

1. Create the channel;

2. Pass promise to task worker;

3. Get result from the channel asynchronously (may or may not wait, depending on worker).

4. Set result to channel in worker (future can get result early, in parallel with `DoCleanup()`).

Future-promise model

- Note 1: for exception management, you can use `.set_exception(std::exception_ptr)`.
 - For example:
 - For `future.get()`, it's then as if `std::rethrow_exception(ptr)`.
- Note 2: `std::future` and `std::promise` are move-only.
 - Only at most one future and one promise occupy the channel.
 - And the channel is shared between two sides, so to prevent use-after-free problem, it also introduces a reference count.
 - Similar to stop token handling, this channel is also called shared state, with both future and promise owning a pointer to it.
- So what does “one-shot channel” really mean in future-promise?

```
void Work(int arg, std::promise<int> promise)
{
    try
    {
        int result = calculateMayThrow(arg);
        promise.set_value(result);
    }
    catch(...)
    {
        promise.set_exception(std::current_exception());
    }
    DoCleanup();
}
```

Future-promise model

- For `std::future`,
 - `.get()` can only be called once, then it cuts the relation with the shared state.
 - And afterwards you need to manipulate retrieved result directly, instead of calling `.get()` again and again.
 - It essentially sets the pointer to the shared state `nullptr`.
- For `std::promise`,
 - `.get_future()` can only be called once, to share the channel uniquely;
 - Otherwise `std::future_error` is thrown, with error code (`.code()`) `std::future_errc::future_already_retrieved`.
 - `.set_xx()` can only be called once, meaning that every task has only one result (either some value or some exception).
 - Otherwise `std::future_error` is thrown, with error code `std::future_errc::promise_already_satisfied`.

Future in detail

- To be specific, `std::future` either associates with a shared state or not (i.e. no state).
 - You can use `.valid()` to check it; when it's valid, you can call:
 - `.get()` -> `T`, which gets the result and then **turns to no state** (invalid).
 - The result `std::move` from the shared state since it's only retrieved once.
 - `.wait()`, which waits until result is ready (i.e. `.get()` will return immediately then instead of waiting).
 - `.wait_for/until(time)` -> `std::future_status`;
 - The return value is scoped enumeration, either `std::future_status::ready` or `std::future_status::timeout`.
 - If it's not valid, all operations above will cause UB.
 - But the standard encourages to throw `std::future_error` with error code `std::future_errc::no_state`.
 - Default ctor or moved-from objects are all invalid.

Shared future

- If you really want to retrieve the result multiple times, you can use `std::shared_future`.
 - For example:

```
void work2(std::shared_future<int> future)
{
    int result = future.get();
    // Do more work on result...
}

void Launch()
{
    std::promise<int> promise;
    std::future<int> future = promise.get_future();
    std::shared_future<int> newFuture = future.share();
    std::jthread t{ work, 1, std::move(promise) },
                t2{ work2, newFuture }, t3{ work2, newFuture };
    int result = newFuture.get();
}
```

Shared future can be copied and `.get()` by multiple times.

Transfer state to shared future (original future is then invalid);



Shared future

- So you can use `.share()` or `std::shared_future(std::future&&)` to get a `std::shared_future`.
 - They both transfer the shared state from the unique future to the shared future, making the unique future invalid;
 - All APIs are same as `std::future`, except that:
 - Copyable, i.e. can share the channel with other `std::shared_future`;
 - `.get()` -> `const T&`, since the result is not one-shot and thus cannot be moved.
 - Can be called multiple times without making itself invalid.
- Essentially, every `std::shared_future` object increases reference count of the shared state and has a pointer to it.

Promise in detail

- For `std::promise`, it creates the channel in default ctor.
 - And it can use `.get_future()` to share the channel with a `std::future`.
 - Besides `.set_value/exception(...)`, you can also call `.set_value/exception_at_thread_exit(...)`.
 - The result is set immediately in this call, but it can only be got after the thread exits (and after thread local objects have been destructed).
 - A naïve example:

```
void work(int arg, std::promise<int>& promise)
{
    promise.set_value_at_thread_exit(Calculate(arg));
}

void Launch()
{
    std::promise<int> promise;
    auto future = promise.get_future();
    std::thread t{ work, 1, std::ref(promise) };
    t.detach();
    // result is got after detached thread exits.
    int result = future.get();
}
```

Here:
reference parameter + `std::ref`;

Previously:
value parameter + `std::move`!

Why?

Promise in detail

- Formally, the channel (shared state) has three cases:
 1. Neither result is set, nor ready: promise hasn't call any `.set_xx()`.
 2. Result is set, but not ready: promise has called `.xx_at_thread_exit()`, but the thread hasn't exited.
 3. Result is set, and ready: promise has called `.set_value/exception()`, or the thread exits with promise previously calling `.xx_at_thread_exit()`.
- Promise must be **ready** before destruction.
 - Otherwise, the task is not finished and the shared state is *abandoned*.
 - Dtor of promise will then automatically set an exception (`std::future_error` with error code `std::future_errc::broken_promise`), and make it ready.
 - Unlike future, promise will only cut off its relation with the shared state in dtor.

Promise in detail

libstdc++ and libc++ will make promise cut off its relation with shared state as soon as result is set, making it have different behaviors in the problem.

See [my stackoverflow answer](#) for complete analysis!

- So if we move the `std::promise` to the thread, and calls `.xx_at_thread_exit()`...
 - Then it will be destructed before ready, since local variables are destructed before the thread exits.
 - And thus exception is tried to set; but we've set a result, causing a conflict (e.g. in MS-STL, `std::terminate`).
- Exercise: is it correct?

```
void Func(std::promise<int>& promise) {  
    promise.set_value_at_thread_exit(10086);  
}  
  
int main()  
{  
    std::future<int> future;  
    {  
        std::promise<int> promise;  
        future = promise.get_future();  
        std::jthread th{ Func, std::ref(promise) };  
    }  
    std::cout << future.get();  
    return 0;  
}
```

Yes, since promise is destructed after the thread exits (thus ready).

And future still refers to the shared state (i.e. reference count is 1), so `.get()` is safe.

After `.get()`, the shared state will be destroyed.

Future-promise model

- Finally, `std::(shared_)future<T>` and `std::promise<T>` have specializations for `T&` and `void`.
 - For `T&`, it's just as if storing `T*`; so differences are:
 - `std::promise::set_value(_at_thread_exit)` accepts `T&` and stores address;
 - For `T` it accepts `const T&` or `T&&` and forwards.
 - `std::future::get()` returns `T&` instead of `std::move` to `T`.
 - For `void`, `get` returns nothing and `set` accepts nothing.
 - It's sometimes used as a wrapper of mutex + condition variable + flag.
 - For example:

```
std::jthread t{ work, 1 };
std::unique_lock lock{ mutex };
condVar.wait(mutex, []{ return ready; });
// Now global work is done
```

```
void Work(int arg)
{
    {
        std::lock_guard _{ mutex };
        SomeGlobalWork(arg);
        ready = true;
    }
    condVar.notify_one();
    DoCleanup();
}
```

```
void Work(int arg, std::promise<void> promise)
{
    SomeGlobalWork(arg);
    promise.set_value();
    DoCleanup();
}

std::promise<void> promise;
auto sync = promise.get_future();
std::jthread t{ work, 1, std::move(promise) };
sync.wait();
// Now global work is done
```

Multithreading

- High-level abstraction of asynchronous operations
 - Future-Promise model
 - `packaged_task`
 - `async`

packaged_task

- Sometimes you may find it still inconvenient to use future-promise...
 - After all, the worker has to accept a promise as parameter, and you still need to manually set results.
 - Why can't we **return** or throw exception directly, just like normal functions?
 - This is what **std::packaged_task** for!

Just like a normal function,
return or throw exception.

1. Create the task with the function
(like **std::move_only_function**);

2. Get the asynchronous result.

3. Pass the function to worker;

Note that [currently MS-STL has a bug](#): move-only functor cannot be used to construct **std::packaged_task**; but to maintain ABI, it's unsolved for a long time.

```
int Work(int arg)
{
    return calculateMayThrow(arg);
}

void Launch()
{
    std::packaged_task<int(int)> task{ Work };
    auto future = task.get_future();
    // Or std::ref(task); packaged_task is not copyable.
    std::jthread t{ std::move(task), 1 };
    auto result = future.get();
    // Then we can use result safely...
}
```

packaged_task

- Essentially, it wraps a `std::promise` and a functor.
 - It defines `operator()`, so it can be used as the first parameter of the thread as a functor.
 - Calling it is equivalent to set the underlying promise with the return value (or thrown exception).
- If you want to `DoCleanUp()` in parallel:

```
std::packaged_task<int(int)> task{ work };
auto future = task.get_future();
std::jthread t{ [task = std::move(task)]() mutable
{
    task(1);
    // Now shared state is already ready.
    DoCleanUp();
}};
auto result = future.get();
// Then we can use result safely...
```

Since `operator()` is not `const`, as it will modify the underlying promise.

packaged_task

```
std::jthread t{ [&task]() mutable  
{  
    task.make_ready_at_thread_exit(1);  
}};
```

- Of course, you can `xx_at_thread_exit`:
 - `operator()` is equiv. to `.set_value/exception()`;
 - `.make_ready_at_thread_exit()` is equiv. to `.set_value/exception_at_thread_exit()`;
 - Note that dtor of `packaged_task` will also abandon the shared state (though here all standard library implementations check “whether result is set” instead of “ready” in `packaged_task`).
- Note: calling `std::packaged_task` won’t release the shared state.
 - Only default ctor / moved-from will make it associate with no state.
 - You can use `.valid()` to check it.
 - Any function call without shared state leads to exception (`no_state`).
 - But you cannot call `packaged_task` twice (`promise_already_satisfied`).
 - And you cannot call `.get_future()` twice (`future_already_retrieved`).

packaged_task

- More conveniently, you can restore the shared state by `.reset()` after the task is called.
 - To be exact, it will first abandon the original state, and then establish a new state. So:
 1. If it doesn't have an initial state, `no_state`.
 - So here you cannot `std::move(task)`, since moved-from task has no state and thus cannot `.reset()`.
 2. Reset a possibly unfinished task is **wrong**.
 - `.reset` will abandon the state (i.e. for promise that's not ready, `set_exception()` with `broken_promise`); but sooner when the task has finished, another result will be set and thus `promise_already_satisfied`.
 - Also, if you don't have a future to refer to the original state, abandoning will free it and may cause illegal memory access.
 3. `.get_future()` should be called again to get future of the new state; the original future is associated with the old state.

```
std::packaged_task<int(int)> task{ Work };
auto future = task.get_future();
std::jthread t{ std::ref(task), 1 };
auto result = future.get();

task.reset();
auto future2 = task.get_future();
std::jthread t2{ std::ref(task), 1 };
auto result2 = future2.get();
```


Multithreading

- High-level abstraction of asynchronous operations
 - Future-Promise model
 - `packaged_task`
 - `async`

async

- Sometimes, you may still find `std::packaged_task` inconvenient...
 - After all, we need to dispatch the task to a thread.
 - We may want a really high-level task-based view: we have a task, and we want to get the result in the future. We don't care how and when it's executed, but just get the result when we need it.
 - That's what `std::async()` does!
 - Just two lines:
- It almost screens all the details we care in future-promise or `std::packaged_task`; it completely represents the concept of task.
 - The best you can do is to set a policy:

```
auto fut = std::async(Work, 1);  
std::cout << fut.get() << '\n';
```

`std::async`

Defined in header `<future>`

```
template< class F, class... Args >  
std::future< /* see below */> async( F&& f, Args&&... args ); (1)
```

```
template< class F, class... Args >  
std::future< /* see below */> async( std::launch policy, (2)  
                                   F&& f, Args&&... args );
```

async

```
enum class launch : /* unspecified */ {  
    async = /* unspecified */,  
    deferred = /* unspecified */,  
    /* implementation-defined */  
};
```

- It is just a scoped enumeration: `};`
 - `std::launch::async`: force the task to be executed on a new thread;
 - If a new thread cannot be started, throw `std::system_error` with error code `std::errc::resource_unavailable_try_again`.
 - `std::launch::deferred`: defer it and still execute on the same thread.
 - That is, execute only when you call `.get/wait()` of the returned future.
 - **Note**: since deferred one will never execute until `.get/wait()`, `.wait_for/until()` will return immediately and return `std::future_status::deferred`.
 - For other future sources (`std::promise` or `std::packaged_task`), it can only return `std::future_status::ready/timeout`.
 - By default (overload (1)), it's `std::launch::async | std::launch::deferred`.
 - Meaning that the system will determine to use `async` or `deferred`.
 - For other implementation-defined flags, the behavior is also implementation-defined.

async

- Moreover, an **async** task has only two cases: all or nothing.
 - That is, once the task has been executed, it will go until done.
 - For **deferred** policy, this is quite natural:
 - If you never call `.get/wait()`, then it's "nothing";
 - If you call them, then it's "all".
 - For **async** policy, since the task will be executed immediately...
 - Destruction of its **future** will be **delayed until the task has completed**.

- Exercise: will these two **Work** execute in parallel?

```
std::async(std::launch::async, work, 1);  
std::async(std::launch::async, work, 2);
```

- No! The returned **future** is ignored, and as a temporary, it's destructed immediately after a statement ends.
- So **Work(2)** begins only when **Work(1)** has completed.
- Instead:

```
// Ensure async futures don't destruct immediately.  
auto future1 = std::async(std::launch::async, work, 1);  
auto future2 = std::async(std::launch::async, work, 2);  
auto result = future1.get() + future2.get();
```

async

- Note 1: in implementation, different behaviors of futures are due to shared state.
 - Though they are all “shared state”, it’s actually polymorphic; and future allocates the state (normally **new** on heap) and stores a pointer to the base class.
- Note 2: **std::async** with **async** policy is better to be used in toy examples, since all details are screened.
 - In MS-STL, tasks are dispatched to an internal thread pool efficiently;
 - But in libstdc++, every new task will launch a new thread, so a large bunch of tasks will drag performance significantly.

Summary

- Threads
 - Abstract thread model.
 - `std::thread`
 - `std::jthread` and stop token handling.
 - `std::exception_ptr`, `thread_local`, `static/std::call_once`
- Synchronization Utilities
 - Semaphore
 - Mutex and lock
 - Shared, recursive, timed
 - Deadlock avoidance – `std::scoped_lock/std::lock`
 - Condition variable
 - Latch & barrier
- High-level abstraction of asynchronous operations
 - Future-promise model.
 - `std::packaged_task`
 - `std::async`

Next lecture...

- We'll talk about advanced concurrency.
- Memory model (memory order)
 - Quite difficult for self-learner.
- Atomic variables
- Coroutine