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

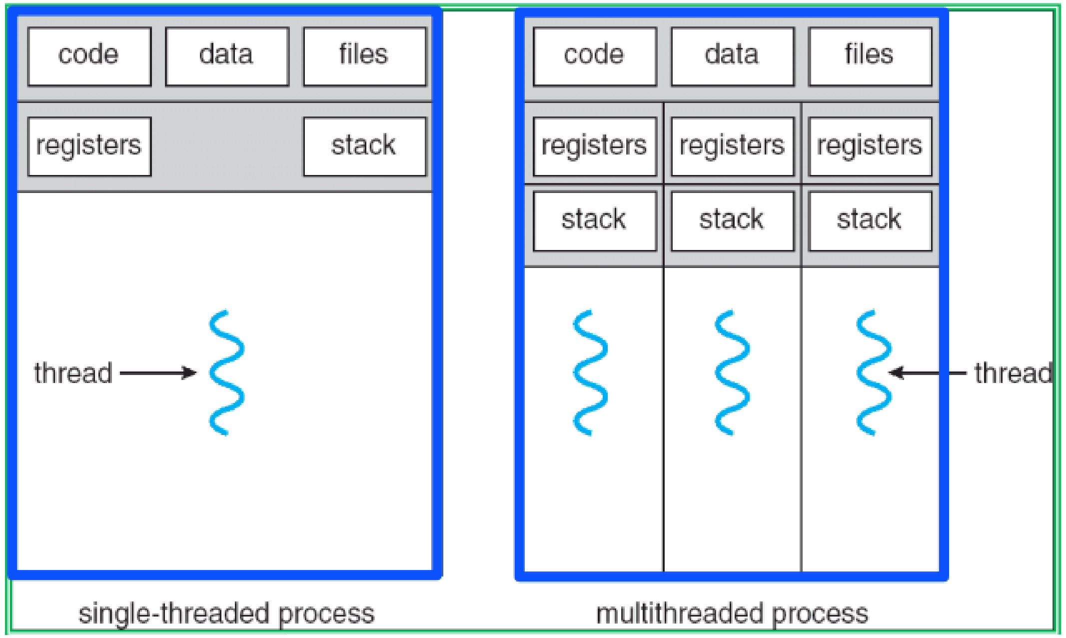
Thread model

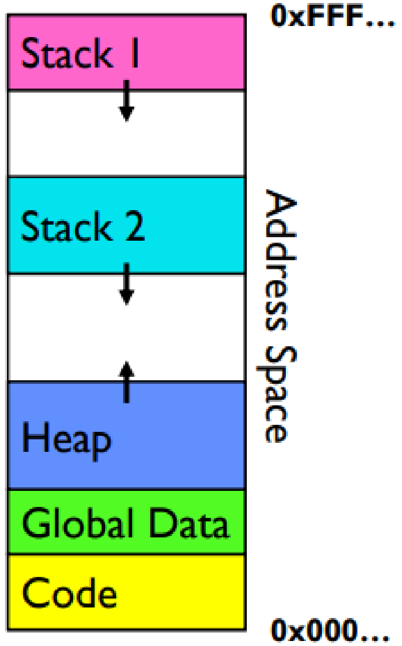
- We first briefly review what thread is…
- We've learnt in ICS that each program is a **process**;
  - It has independent address space, and possibly other status like file descriptors (depend on OS).
  - Good isolation, good protection, really limited ability to access memory of another process.
- 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.

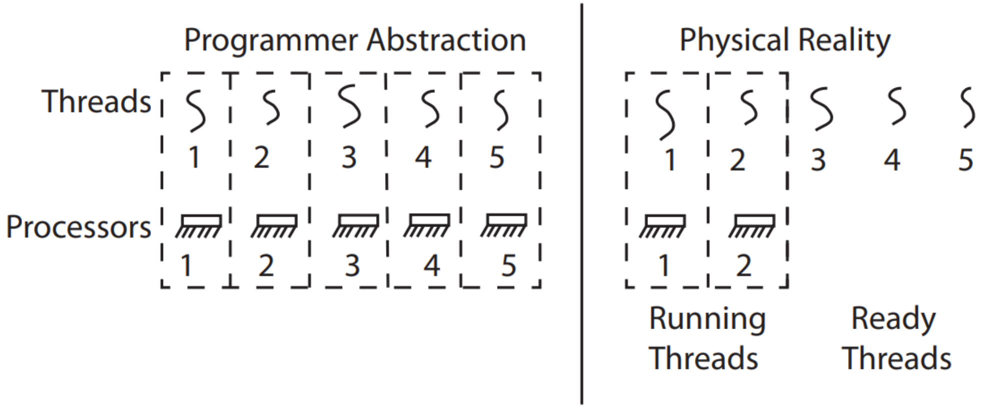
Thread model

- 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++?

```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

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

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

- 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(functor) 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.

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…

```cpp
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();
}
```

```cpp
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`.

```
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)
```

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

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

3. Then thread executes `std::invoke`;
   - Materialized parameters are passed to new thread.

1) Invoke the *Callable* object `f` with the parameters `args` as by `INVOKE(std::forward<F>(f), std::forward<Args>(args)...)`

4. And finally `arg1` is forwarded to parameters of `func`.
   - And thus move-constructed as `object`.

```
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.

Output	
Main id: 21244	
Construct at 21244	Step 1
Move at 21244	Step 2; new thread executes Step 3.
Destruct at 21244	Param of thread ctor destructed.
Move at 8504	Step 4
Thread id: 8504	
Destruct at 8504	Param of func destructed.
Destruct at 8504	Materialized temporary destructed.

- 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.

```cpp
#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.

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:

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

```cpp
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:

```cpp
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();
```

`get_stop_token()` is provided automatically.

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:

```cpp
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:

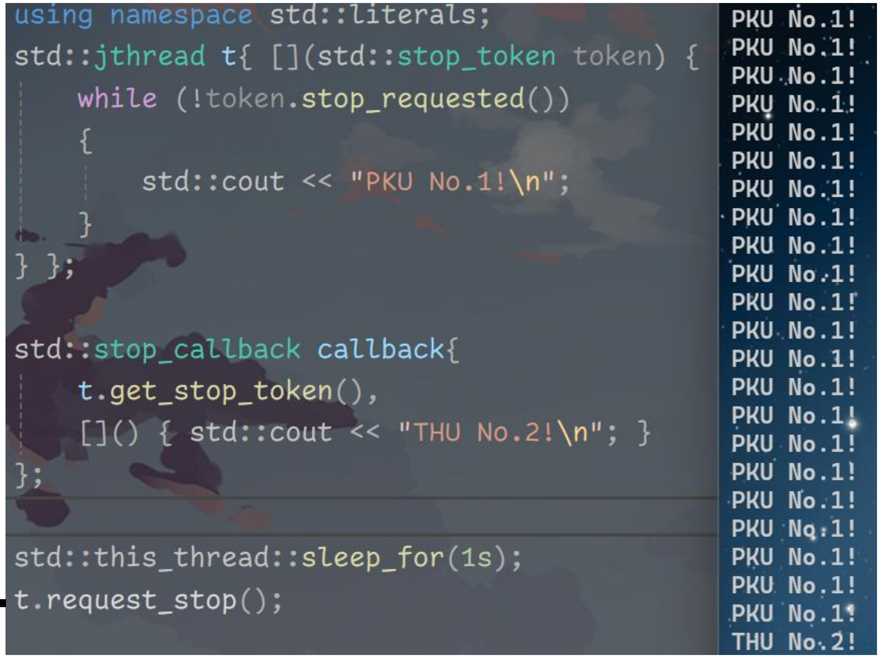
`std::invoke(auto(std::forward<F>(f)), auto(std::forward<Args>(args))...)`.	(since C++23)

Stop token

- Finally, stop request can be associated with callbacks.
  - By `std::stop_callback` with `std::stop_token`:
    - Ctor registers callback on the state;

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

    - Dtor deregisters the callback.
- For example:



```cpp
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();
```

```
PKU No.1!
PKU No.1!
PKU No.1!
PKU No.1!
PKU No.1!
PKU No.1!
PKU No.1!
PKU No.1!
PKU No.1!
PKU No.1!
PKU No.1!
PKU No.1!
PKU No.1!
PKU No.1!
PKU No.1!
PKU No.1!
PKU No.1!
PKU No.1!
PKU No.1!
PKU No.1!
PKU No.1!
PKU No.1!
THU No.2!
```

Callbacks will be executed here.

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:

```cpp
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.

```cpp
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;
}
```

```cpp
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.

```cpp
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 };
}
```

```
Microsoft Visual Studio 调试控制台
Simple example: called 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 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 loopupTable.
    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.