CS100 Introduction to Programming

Recitation 9
llk89

async 异步 ☆ 推荐

thread 多线程 — 实时性

区别:

① 难以拿到值

② 资源紧张.

mutex:

lock

unlock = cost-guard

NO PLAGIARISM!!!

- The most likely cause for failing this course.
- You WILL be caught!
- We WILL punish!
- They WILL know!
  - Parents
  - University
  - School
  - Fellows

Today's learning objectives

- Asynchronous tasks and threads
- Promises and tasks
- More on mutexes and condition variables
- More on std::call_once
- Example: Ping-Pong threads

Spawning asynchronous tasks

- Two ways: **`std::async`** and **`std::thread`**
- It's all about things that are callable:
  - Functions and Member functions.
  - Objects with **`operator()`** and Lambda functions

Hello World with std::async

Output:
hellow ... from main
---- async.

```
#include <future> // for std::async
#include <iostream>

void write_message(std::string const& message) {
    std::cout<<message;
}

int main() {
    auto f = std::async(write_message,
            “hello world from std::async\n");
    write_message("hello world from main\n");
    f.wait();
}
```

形参

报错：terminate called without an active expection

Hello World with `std::thread`

```
#include <thread> // for std::thread
#include <iostream>
void write_message(std::string const& message) {
    std::cout<<message;
}

int main() {
    std::thread t(write_message,
                  "hello world from std::thread\n");
    write_message("hello world from main\n");
    t.join();
}
```

output: hello world from std:thread.

Missing join with std::thread

```
#include <thread>
#include <iostream>
void write_message(std::string const& message) {
    std::cout<<message;
}
int main() {
    std::thread t(write_message,
                  "hello world from std::thread\n");
    write_message("hello world from main\n");
    // oops no join
}
```

Missing `wait` with `std::async`

```
#include <future>
#include <iostream>
void write_message(std::string const& message) {
    std::cout<<message;
}
int main() {
    auto f = std::async(write_message,
                  "hello world from std::async\n");
    write_message("hello world from main\n");
    // oops no wait
}
```

Async Launch Policies

- The standard launch policies are the members of the `std::launch` scoped enum.
- They can be used individually or together.

Async Launch Policies

- `std::launch::async` => "as if" in a new thread.
- `std::launch::deferred` => executed on demand.
- `std::launch::async | std::launch::deferred` => implementation chooses (default).

std::launch::async

```
#include <future>
#include <iostream>
#include <stdio.h>
void write_message(std::string const& message) {
    std::cout<<message;
}

int main() {
    auto f=std::async(
        std::launch::async, write_message,
         "hello world from std::async\n");
    write_message("hello world from main\n");
    getchar();
    f.wait();
}
```

std::launch::deferred

```
#include <future>
#include <iostream>
#include <stdio.h>
void write_message(std::string const& message) {
    std::cout<<message;
}

int main() {
    auto f=std::async(
        std::launch::deferred, write_message,
        "hello world from std::async\n");
    write_message("hello world from main\n");
    getchar();
    f.wait();
}
```

Returning values with `std::async`

```
#include <future>
#include <iostream>
int find_the_answer() {
    return 42;
}

int main() {
    auto f = std::async(find_the_answer);
    std::cout<<"the answer is "<<f.get()<<"\n";
}
```

42!

Passing parameters

```
#include <future>
#include <iostream>
std::string copy_string(std::string const&s) {
    return s;
}

int main() {
    std::string s="hello";
    auto f=std::async(std::launch::deferred,
        copy_string,s);
    s="goodbye";
    std::cout<<f.get()<<" world!\n";
}
```

Output: hello world.

Passing parameters with `std::ref`

```
#include <future>
#include <iostream>
std::string copy_string(std::string const&s) {
    return s;
}

int main() {
    std::string s="hello";
    auto f=std::async(std::launch::deferred,
        copy_string,std::ref(s));
    s="goodbye";
    std::cout<<f.get()<<" world!\n";
}
```

引用传递.

Output: goodbye world.

std::ref() → 普通版

std::cref() → const 版

Passing parameters with a lambda

```
std::string copy_string(std::string const&s) {
    return s;
}

int main() {
    std::string s="hello";
    auto f=std::async(std::launch::deferred,
        [&s](){return copy_string(s);});
    s="goodbye";
    std::cout<<f.get()<<" world!\n";
}
```

[ ]( )

lambda → C++11 老特性

std::async passes exceptions

```
#include <future>
#include <iostream>
int find_the_answer() {
  throw std::runtime_error("Unable to find the answer”);
}
```

判断运行时间错误

```
int main() {
  auto f=std::async(find_the_answer);
  try {
    std::cout<<"the answer is "<<f.get()<<"\n";
  }
  catch(std::runtime_error const& e) {
    std::cout<<"\nCaught exception: "<<e.what();
  }
}
```

Today’s learning objectives

- Asynchronous tasks and threads
- Promises and tasks
- More on mutexes and condition variables
- More on std::call_once
- Example: Ping-Pong threads

Manually setting futures

- Two ways:
  - **`std::promise`** 拿值
  - **`std::packaged_task`** 拿函数.
- **`std::promise`** allows you to explicitly set the value
- **`std::packaged_task`** is for manual task invocation, e.g. thread pools.

std::promise

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

void find_the_answer(std::promise<int>* p){
  p->set_value(42);
}

int main() {
  std::promise<int> p;
  auto f = p.get_future();
  std::thread t(find_the_answer,&p);
  std::cout<<"the answer is "<<f.get()<<"\n";
  t.join();
}
```

Output : the answer is 42

std::packaged_task

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

int find_the_answer() {
  return 42;
}

int main() {
  std::packaged_task<int()> task(find_the_answer);
  auto f=task.get_future();
  std::thread t(std::move(task));  不用赋值，直接做值的传递
  std::cout<<"the answer is "<<f.get()<<"\n";
  t.join();
}
```

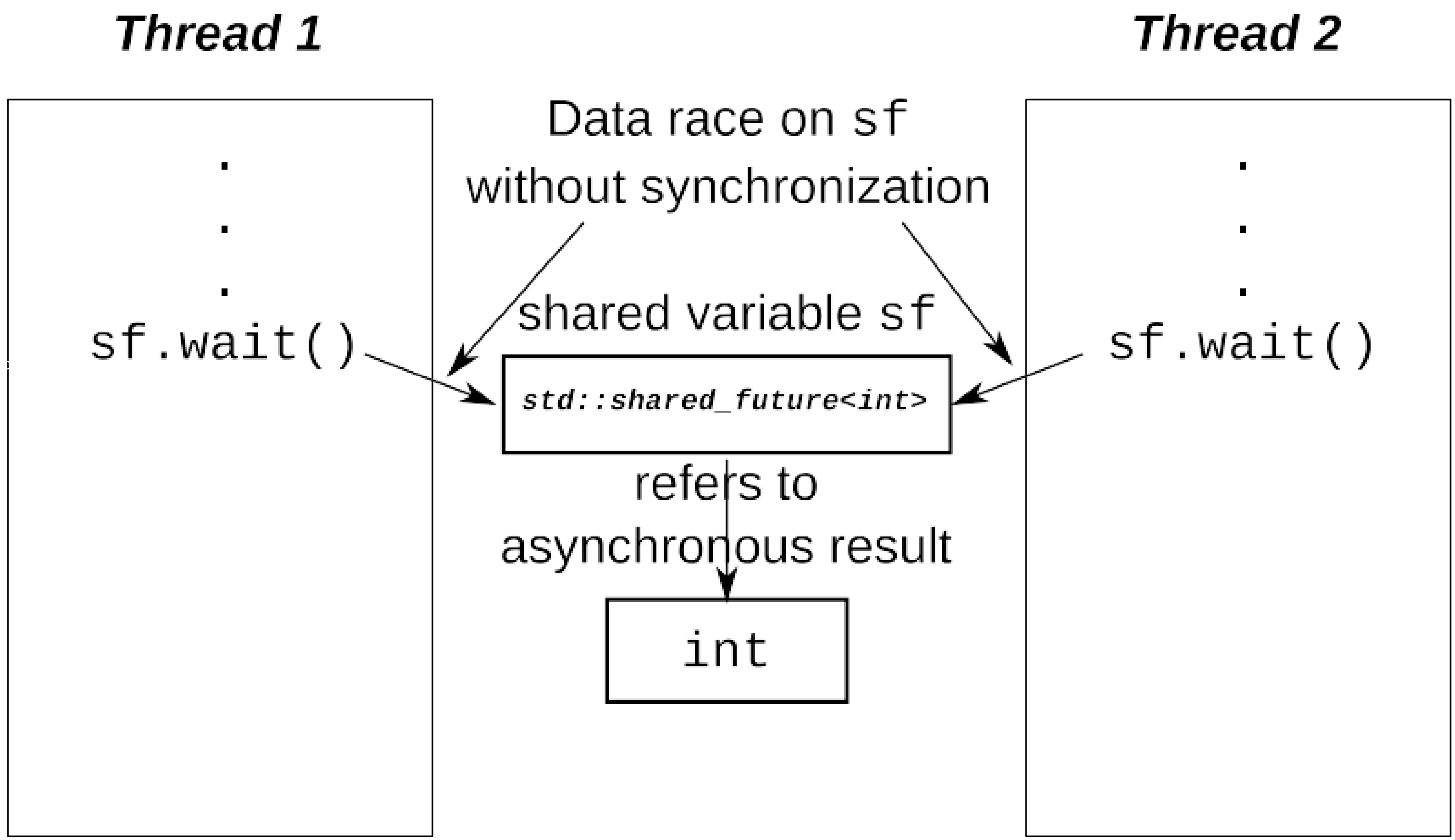
Waiting for futures from multiple threads

- **Use** `std::shared_future<T>` **rather than** `std::future<T>`

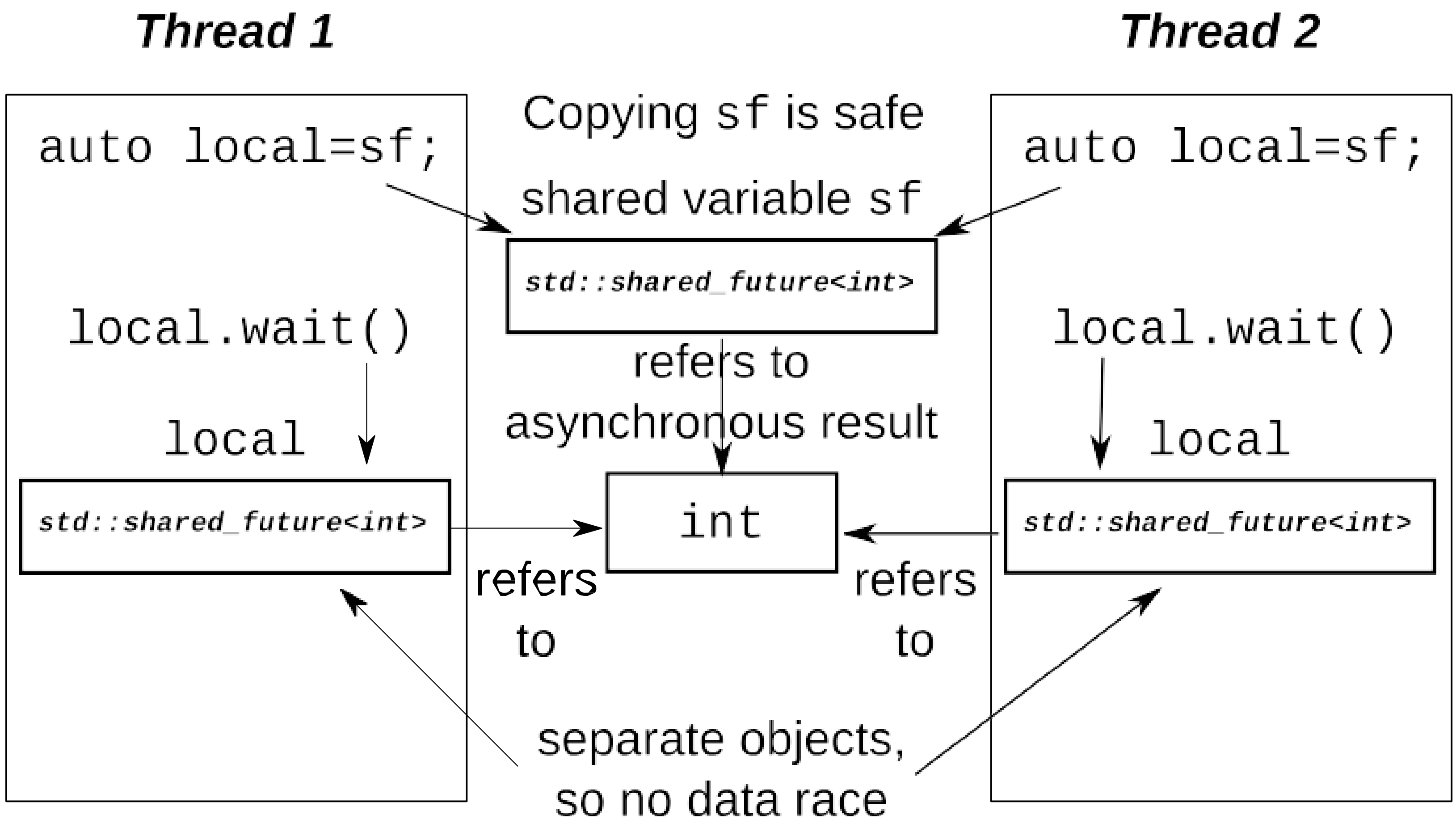
```
std::future<int> f=/*...*/;
std::shared_future<int> sf(std::move(f));

std::future<int> f2=/*...*/;
std::shared_future<int> sf2(f.share());

std::promise<int> p;
std::shared_future<int> sf3(p.get_future());
```

```cpp
#include <future>
#include <thread>
#include <iostream>
#include <sstream>

void wait_for_notify(int id, std::shared_future<int> sf)
   std::ostringstream os;
   os << "Thread " << id << " waiting\n";
   std::cout << os.str();
   os.str("");
   os << "Thread "<< id << " woken, val=" << sf.get() << "\n";
   std::cout << os.str();
}
int main() {
   std::promise<int> p;
   auto sf = p.get_future().share();
   std::thread t1(wait_for_notify,1,sf);
   std::thread t2(wait_for_notify,2,sf);
   std::cout << "Waiting\n";
   std::cin.get();
   p.set_value(42);
   t2.join(); t1.join();
}
```

`std::shared_future<T>` objects cannot be shared



Separate `std::shared_future<T>` objects can share state



Today's learning objectives

- Asynchronous tasks and threads
- Promises and tasks
- More on mutexes and condition variables
- More on std::call_once
- Example: Ping-Pong threads

Locking multiple mutexes

mutex:

lock

unlock

e.g. vecto

```
class account {
    std::mutex m;
    currency_value balance;
public:
    friend void transfer( account& from,
                          account& to,
                          currency_value amount ) {
        std::lock_guard<std::mutex> lock_from(from.m);
        std::lock_guard<std::mutex> lock_to(to.m);
        from.balance -= amount;
        to.balance += amount;
    }
};
```

Locking multiple mutexes (II)

```
void transfer( account& from,
               account& to,
               currency_value amount) {
  std::lock(from.m,to.m);
  std::lock_guard<std::mutex> lock_from(
      from.m, std::adopt_lock);
  std::lock_guard<std::mutex> lock_to(
      to.m, std::adopt_lock);
  from.balance -= amount;
  to.balance += amount;
}
```

Waiting for events without futures

- Repeatedly poll in a loop (busy-wait)
- Wait using a condition variable

Synchronization between threads

- Apart from just protecting data, sometimes we may wish for one thread to **wait** until another thread has something done
- In C++:
  - Conditional variables
  - Futures

Example: Waiting for an item

- If all we've got is `try_pop()`, the only way to wait is to poll:

```
std::queue<my_class> the_queue;
std::mutex the_mutex;
void wait_and_pop(my_class& data){
  for(;;){
    std::lock_guard<std::mutex> guard(the_mutex);
    if(!the_queue.empty()) {
      data=the_queue.front();
      the_queue.pop();
      return;
    }
  }
}
```

- This is not ideal.

std::condition_variable

- A synchronization primitive that can be used to block a thread or multiple threads at the same time, until
  - A notification is received from another thread
  - A time-out expires

std::condition_variable

std:: future :

A.get( )

A.future_get( )

provider :

thread.

- A thread that intends to wait on std::condition_variable has to acquire a std::unique_lock first
- The wait operations atomically release the mutex and suspend the execution of the thread
- When the condition variable is notified, the thread is awakened, and the mutex is reacquired

Performing a blocking wait

- We want to wait for a particular condition to be true (there is an item in the queue).
- This is a job for **`std::condition_variable`**:

```
std::condition_variable the_cv;

void wait_and_pop(my_class& data) {
     std::unique_lock<std::mutex> lk(the_mutex);
     the_cv.wait(lk,
                 []()
                 {return !the_queue.empty();});
     data = the_queue.front();
     the_queue.pop();
}
```

Signalling a waiting thread

- To signal a waiting thread, we need to notify the condition variable when we push an item on the queue:

```
void push(Data const& data)
{
    {
        std::lock_guard<std::mutex> lk(the_mutex);
        the_queue.push(data);
    }
    the_cv.notify_one();
}
```

Example

Mutex to protect resource

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

Example

Queue used to pass data

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

Example

When data is ready, thread locks mutex, pushes data, and calls notify_one()

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

notify_one() notifies
The waiting thread

Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

Receiver thread puts itself into waiting mode through this call (if queue is empty). It will also release the lock

Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

It also passes a wake condition that will be checked upon notify_all()

Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

The mutex will be automatically locked once the wait terminates

Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

Lock only for as long as necessary

Today's learning objectives

- Asynchronous tasks and threads
- Promises and tasks
- More on mutexes and condition variables
- More on std::call_once
- Example: Ping-Pong threads

std::call_once

- It is possible that some operations are to be done only once
- Use

```
std::call_once( std::once_flag, function);
```

One-time initialization with `std::call_once`

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

std::once_flag flag1;
void printHello() {std::cout << "Hello\n"; }
void threadFunction(){
    std::call_once(flag1, printHello);
}

int main(){
    std::thread st1(threadFunction);
    std::thread st2(threadFunction);
    std::thread st3(threadFunction);
    st1.join();
    st2.join();
    st3.join();
    return 0;
}
```

Conditional variable

Unique call

只调用一次

One-time initialization with `std::call_once`

- Example use for resource allocation

```
std::unique_ptr<some_resource> resource_ptr;
std::once_flag resource_flag;

void foo() {
  std::call_once(
      resource_flag,
      []{resource_ptr.reset(new some_resource);});
  resource_ptr->do_something();
}
```

One-time initialization with local statics

```
void foo() {
  static some_resource resource;
  resource.do_something();
}
```

Today's learning objectives

- Asynchronous tasks and threads
- Promises and tasks
- More on mutexes and condition variables
- More on std::call_once
- Example: Ping-Pong threads

Example: Ping-Pong threads

```
#include "stdlib.h"
#include <string>
#include <thread>
#include <mutex>
#include <iostream>
#include <unistd.h>

bool onRightSide;
std::mutex mut;
std::condition_variable data_cond;
void player( bool isRightSidePlayer, std::string message ) {
  while(1) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[&isRightSidePlayer]{
       return isRightSidePlayer == onRightSide;});
    std::cout << message << "\n";
    usleep(1000000);
    onRightSide = !onRightSide;
    lk.unlock();
    data_cond.notify_one();
  }
}
```

Example: Ping-Pong threads

```
int main() {
  onRightSide = true;
  std::thread leftPlayer( player, false, std::string("Pong") );
  std::thread rightPlayer( player, true, std::string("Ping") );
  leftPlayer.join();
  rightPlayer.join();
  return 0;
}
```

QA Time

- If you have any problems with…
  - last week's lecture
  - Recitation 9
- Ask now