Concurrent Programming in C++

Venkat Subramaniam
venkats@agiledeveloper.com
@venkat_s

Platform Neutral

- The standard concurrency model makes it possible to write portable concurrent code

Level of Concurrency

```
cout << std::thread::hardware_concurrency() << endl;
```

- Number of threads that can be run concurrently by the hardware
- May be number of CPU cores available on the hardware

Creating Thread

```
#include <iostream>
#include <string>
#include <thread>
using namespace std;

void printThreadInfo(string msg) {
    cout << "In " << msg << " " << std::this_thread::get_id() << endl;
}

int main()
{
    printThreadInfo("main");

    std::thread thread(printThreadInfo, "in another thread");

    thread.join();

    return 0;
}
```

```
In main 8700
In in another thread 9236
```

Starting Thread Is Easy

- But, we immediately have to worry about concurrency issues

```
printThreadInfo("main");

std::thread thread1(printThreadInfo, "in thread1");
std::thread thread2(printThreadInfo, "in thread2");

thread1.join();
thread2.join();
```

```
In main 6380
In In in thread1in thread2  31445524
```

Manage Shared Resources

```cpp
#include <mutex>
using namespace std;

std::mutex cout_mutex;

void printThreadInfo(string msg) {
    cout_mutex.lock();
    cout << "In " << msg << " " << std::this_thread::get_id() << endl;
    cout_mutex.unlock();
}
```

```
In main 7960
In in thread1 7604
In in thread2 932
```

- Careful, unlock may not happen on exception
- We'll solve this later

Thread Function

- To launch a thread pass it a callable
- It can be a function like we saw
- It can be a lambda expression

```
std::thread thread1([]() { cout << "Hi from another thread" << endl; });
```

- It can be an object with operator() overloaded

```
class Sample {
public:
    void operator()() { cout << "howdy" << endl; }
};

int main()
{
    Sample sample;
    std::thread thread1(sample);
```

Passing Object Gotcha

```
std::thread thread1(Sample()); //DOES NOT WORK
```

- Thinks thread1 is a function declaration taking a pointer to a function

```
std::thread thread1((Sample() )); // OK
```

```
std::thread thread1{ Sample() }; // OK
```

Passing Object

- When a callable object is passed, a copy of the object is passed. It is safe to destroy the object after passing.

```cpp
class Sample {
public:
    void operator()() { cout << this << endl; }
};

int main()
{
    Sample sample;
    sample();

    std::thread thread1{ sample };
```

```
004FF8CF
00EFF658
```

Think Abstraction

- Keep in mine the difference between the thread object and the thread of execution
- The thread object may die long before the thread of execution is finished
- The thread of execution may finish long before the thread object dies

join or detach

- Once you start a thread, either join or detach from it
- Join will wait for the thread to finish

- detaching a thread makes it a daemon thread
- You mean to fire and forget these threads
- No longer attached to thread object

- Not doing either may result in a runtime error

detaching

```
void func() {
    cout << "finishing task..." << endl;
}

int main()
{
    std::thread thread(func);
    thread.detach();

    std::this_thread::sleep_for(2s);
```

joinable?

```
std::thread thread1(func);
cout << thread1.joinable() << endl; // true
thread1.detach();

cout << thread1.joinable() << endl; // false

std::thread thread2(func);
cout << thread2.joinable() << endl; // true
thread2.detach();

cout << thread2.joinable() << endl; // false
```

joining

- If you want to wait for the thread to finish before moving on
- Use caution in join
- What if an exception is thrown?

Consider Exceptions

```cpp
void slowFunction() { std::this_thread::sleep_for(2s); }

void troublesomeMethod() { throw "oops"; }

void caller() {
    std::thread thread1(slowFunction);
    troublesomeMethod();
    thread1.join();
}

int main()
{
    try { caller(); }
    catch (...) { cout << "now what?" << endl; }
    //Runtime error since thread was not joined or detached
```

join properly…

```
void caller() {
    std::thread thread1(slowFunction);
    try {
        troublesomeMethod();
    }
    catch (...) {
        thread1.join();
        throw;
    }
    thread1.join();
}
```

- verbose
- Mundane
- Error prone

Use RAII pattern

- Resource Acquisition Is Initialization pattern

```
enum JoinOrDetach { join, detach };

class Thread {
    std::thread thread;
    JoinOrDetach joinOrDetach;
public:
    Thread(std::thread&& thread, JoinOrDetach joinOrDetach)
        : thread(std::move(thread)), joinOrDetach(joinOrDetach) {}
    Thread(const Thread&) = delete;
    Thread(Thread&&) = delete;

    ~Thread() {
        if (thread.joinable()) {
            if (JoinOrDetach::join == joinOrDetach)
                thread.join();
            else
                thread.detach();
        }
    }
};
```

Use RAII pattern

```
void caller() {
    Thread thread(std::thread(slowFunction), JoinOrDetach::join);
    troublesomeMethod();
}
```

Thread Argument Gotcha

```
class Person {
private:
    int age;
public:
    Person(int age) : age(age) {}
    int getAge() const { return age; }
    void growOlder() { age++; }
};

void grow(Person& person) {
    cout << "increasing age..." << endl;
    person.growOlder();
}
```

```
Person sam(2);

std::thread thread(grow, sam);
thread.join();

cout << sam.getAge() << endl; // 2 and not 3!
```

Thread Argument Gotcha

```
Person sam(2);

std::thread thread(grow, std::ref(sam));
thread.join();

cout << sam.getAge() << endl; // 3
```

- Decide to pass a value or a reference

Concurrency & Mutability

- Read-only data are the safest from the currency point of view
- Mutability is not very pleasant
- shared mutability is purely evil
    - This is the source of many concurrency issues

Rule for Concurrency

- A concurrent code should not break an invariants from the point of view of observing threads

- It’s our responsibility to avoid race conditions

Race Condition

```
int a_count = 0;

void change(int by) {
    int value = a_count;
    a_count = value + by;
}

int main()
{
    vector<thread> threads;

    for (auto i = 0; i < 20000; i++) {
        threads.push_back(std::thread(change, (i % 2 == 0) ? 1 : -1));
    }

    for (auto thread = threads.begin(); thread != threads.end(); thread++) {
        thread->join();
    }

    cout << a_count << endl; // should be 0 but result is unpredictable
```

Avoiding Race Condition

```
int a_count = 0;
std::mutex a_count_mutex;

void change(int by) {
    a_count_mutex.lock();
    int value = a_count;
    a_count = value + by;
    a_count_mutex.unlock();
}
```

- Risky, however
- What if there was an exception, we forget to unlock, or a path misses call to unlock?

Avoiding Race Condition

```
int a_count = 0;
std::mutex a_count_mutex;

void change(int by) {
    std::lock_guard<std::mutex> guard(a_count_mutex);
    int value = a_count;
    a_count = value + by;
}
```

- Resource Acquisition Is Initialization Pattern here again

lock_guard not a Panacea

- We can’t get confident just because we see mutex or lock_guard in code
- Encapsulating the data within an object will not totally cure our issues either

Don’t Let the data Escape

- Escaping is one of the common issues that leads to concurrency bugs
- Not only should methods of an object encapsulate it, it should also not allow data to escape
- Anywhere a pointer or reference is returned from a method or passed to another method is a source of potential trouble

Avoiding Deadlock

- Deadlocks can happen if we lock multiple mutex one at a time
- To avoid we often aim for an ordered lock, but that can be hard to implement
- std::lock comes to help

Deadlock

```
class Account {
private:
    int balance;
    std::mutex mutex;
public:
    Account(int balance) : balance(balance) {}

    int getBalance() const { return balance;  }

    void transferFrom(Account& other, int amount) {
        std::lock_guard<std::mutex> guard1(mutex);
        std::this_thread::sleep_for(1s); //simulate delay
        cout << "acquired one... waiting for the other..." << endl;
        std::lock_guard<std::mutex> guard2(other.mutex);

        //assume there is enough fund...
        balance += amount;
        other.balance -= amount;
    }
};
```

Deadlock

```
Account account1(1000);
Account account2(1000);

std::thread thread1(&Account::transferFrom, &account1, std::ref(account2), 100);
std::thread thread2(&Account::transferFrom, &account2, std::ref(account1), 100);

thread1.join();
thread2.join();

cout << account1.getBalance() << endl;
cout << account2.getBalance() << endl;
```

Fixing Deadlock

```
void transferFrom(Account& other, int amount) {
    std::lock(mutex, other.mutex);
    std::lock_guard<std::mutex> guard1(mutex, std::adopt_lock);
    std::this_thread::sleep_for(1s); //simulate delay
    cout << "acquired one... waiting for the other..." << endl;
    std::lock_guard<std::mutex> guard2(other.mutex, std::adopt_lock);

    //assume there is enough fund...
    balance += amount;
    other.balance -= amount;
}
```

Multiple Locks

- Never acquire multiple locks one at a time
- Always ask for them in one shot
- Never lock on an already acquired mutex
- Avoid Nested Locks

unique_lock

- Unlike lock_guard, these are movable (but not copyable)
- They can be locked later - in deferred mode, if desired
- You can unlock and lock again on this one as needed

Using unique_lock

```
void transferFrom(Account& other, int amount) {
    std::unique_lock<std::mutex> lock1(mutex, std::defer_lock);
    std::this_thread::sleep_for(1s); //simulate delay
    std::unique_lock<std::mutex> lock2(other.mutex, std::defer_lock);

    cout << "request lock" << endl;
    std::lock(lock1, lock2);
    cout << "one thread working..." << endl;
    //assume there is enough fund...
    balance += amount;
    other.balance -= amount;
    cout << "done working..." << endl;
}
```

```
request lockrequest lock

one thread working...
done working...
one thread working...
done working...
1000
1000
```

Another Race Condition

```
class Resource {
private:
    bool used = false;
    std::mutex mutex;

public:
    bool isAvailable() {
        std::lock_guard<std::mutex> guard(mutex);
        return !used;
    }
    string use() {
        std::lock_guard<std::mutex> guard(mutex);
        if (!used) {
            used = true;
            return "it's for you";
        }
        else {
            return "it's gone";
        }
    }
};
```

Another Race Condition

```
void useResource(Resource& resource) {
    if (resource.isAvailable()) {
        cout << "is available!" << endl;
        std::this_thread::sleep_for(1s); //simuate delay
        cout << resource.use() << endl;
    }
}
int main()
{
    Resource resource;

    std::thread thread1(useResource, std::ref(resource));
    std::thread thread2(useResource, std::ref(resource));

    thread1.join();
    thread2.join();
```

Another Race Condition

- Need to lock around the entire operation

- But, how…

Another use of unique_lock

```
    std::mutex external;

public:
    auto getLock() { return std::unique_lock<std::mutex>(external); }

    bool isAvailable() {
        std::lock_guard<std::mutex> guard(mutex);
```

```
void useResource(Resource& resource) {
    auto lock = resource.getLock();

    if (resource.isAvailable()) {
        cout << "is available!" << endl;
```

Another Approach

```
    std::mutex external;

public:
    void operateOn(function<void(Resource&)> func) {
        std::unique_lock<std::mutex> guard(external);

        func(*this);
    }

    bool isAvailable() {
        std::lock_guard<std::mutex> guard(mutex);
```

```
void useResource(Resource& resource) {
    resource.operateOn([](Resource& resource) {
        if (resource.isAvailable()) {
            cout << "is available!" << endl;
            std::this_thread::sleep_for(1s); //simuate delay
```

Need for call_once

```
class Singleton {
private:
    static std::shared_ptr<Singleton> ptr;
    static std::mutex mutex;

    Singleton() { cout << "created..." << endl; }
public:
    static shared_ptr<Singleton> getInstance() { //convoluted, error prone, slow
        if (!ptr) {
            std::lock_guard<std::mutex> guard(mutex);
            if (!ptr) {
                ptr.reset(new Singleton());
            }
        }

        return ptr;
    }

};
```

Using call_once & once_flag

```
class Singleton {
private:
    static std::shared_ptr<Singleton> ptr;
    static std::once_flag ptr_once;

    Singleton() { cout << "created..." << endl; }
public:
    static shared_ptr<Singleton> getInstance() {
        std::call_once(ptr_once, []() { return std::shared_ptr<Singleton>(new Singleton()); });

        return ptr;
    }
};
```

Synchronizing

- Shared variables are not the smartest way to communicate and synchronize between threads

- Need to avoid busy waits and polling

conditional_variable

```cpp
int product;
std::mutex product_mutex;
std::condition_variable signal;

void producer() {
    while (true) {
        {
            std::lock_guard<std::mutex> guard(product_mutex);
            product++;
            signal.notify_one();
        }

        std::this_thread::sleep_for(300ms);
    }
}

void consumer(string name) {
    while (true) {
        {
            std::unique_lock<std::mutex> guard(product_mutex);
            signal.wait(guard, [] {return product > 0; });
            cout << name << " " << product << endl;
            product--;
        }
        std::this_thread::sleep_for(1s);
    }
}
```

conditional_variable

```
int main()
{
    int numberOfConsumers = 3;
    std::thread producerThread(producer);
    producerThread.detach();

    vector<thread> threads;

    for (int i = 0; i < numberOfConsumers; i++) {
        string name = "consumer" + to_string(i);
        threads.push_back(std::thread(consumer, name));
    }

    for (auto& thread : threads) {
        thread.join();
    }


    return 0;
}
```

condition_variable

- When wait is called, it checks the condition
- If condition is true, proceeds
- If condition is false, it will release the lock and wait
- Once notified, it will acquire the lock, check the condition
- If condition satisfied, moves forward
- Otherwise, releases lock and waits

Always Timeout

- Anytime you call a function that will wait for some thread or task to complete, always specify a timeout

- Look for variations of wait that take a timeout
  - duration
  - until a particular time

An Awkward Use

```cpp
std::condition_variable done;
long result;

long fib(int position) {
    if (position < 2)
        return 1;
    else
        return fib(position - 1) + fib(position - 2);
}

void computeFib(int position) {
    result = fib(position);
    done.notify_one();
}

void printResult() {
    std::mutex mutex;
    std::unique_lock<std::mutex> guard(mutex);

    done.wait(guard, [] { return result > 0; });
    cout << result << endl;
}

int main()
{
    std::thread compute(computeFib, 40);
    std::thread print(printResult);

    compute.join();
    print.join();
```

- Though conditional_variable may be used here, it is not the right fit

Future

- Future is useful for one-off event
- It may accompany some data with it

Using Future

```cpp
#include <future>

long fib(int position) {
    if (position < 2)
        return 1;
    else
        return fib(position - 1) + fib(position - 2);
}

void printResult(future<long> result) {
    cout << result.get() << endl;
}

int main()
{
    std::thread print(printResult, std::async(fib, 40));
    print.join();
```

async launch options

- deferred - postpone until get or wait called
- May run in the callers thread
- Lazy and may never run—efficient

- async- run in a new thread
- std::launch::deferred
- std::launch::async
- std::launch::deferred | std::launch::async

async launch options

```
#include <future>

long fib(int position) {
    if (position < 2)
        return 1;
    else
        return fib(position - 1) + fib(position - 2);
}

long compute(int position) {
    cout << "computing in thread..." << std::this_thread::get_id() << endl;
    return fib(position);
}

int main()
{
    cout << "in main..." << std::this_thread::get_id() << endl;
    auto result = std::async(std::launch::deferred, compute, 40);

    cout << result.get() << endl;
```

```
in main...49916
computing in thread...49916
165580141
```

async launch options

```
auto result = std::async(std::launch::async, compute, 40);

cout << result.get() << endl;
```

```
in main...51392
computing in thread...51400
165580141
```

Future & Thread Safety

- Future is thread safe for access by worker thread and calling thread

- Future is *not* thread safe for multiple threads to access the same instance

Future & Thread Safety

```
future<long> badIdea;

void print1() {
    cout << badIdea.get() << endl;
}

void print2() {
    cout << boolalpha << (badIdea.get() > 1) << endl;
}

int main()
{
    badIdea = std::async(fib, 40);
                                            // Will FAIL
    std::thread thread1(print1);
    std::thread thread2(print2);

    thread1.join();
    thread2.join();
```

Future & Thread Safety

```
void print1(std::shared_future<long> future) {
    cout << future.get() << endl;
}

void print2(std::shared_future<long> future) {
    cout << boolalpha << (future.get() > 1) << endl;
}

int main()
{
    auto future = std::async(fib, 40);

    std::shared_future<long> sharedFuture(std::move(future));

    std::thread thread1(print1, sharedFuture);
    std::thread thread2(print2, sharedFuture);
```

packaged_task

- Think of this as a connector between a function and a future of the result of that function

- Useful to schedule a set of functions for execution on a thread pool

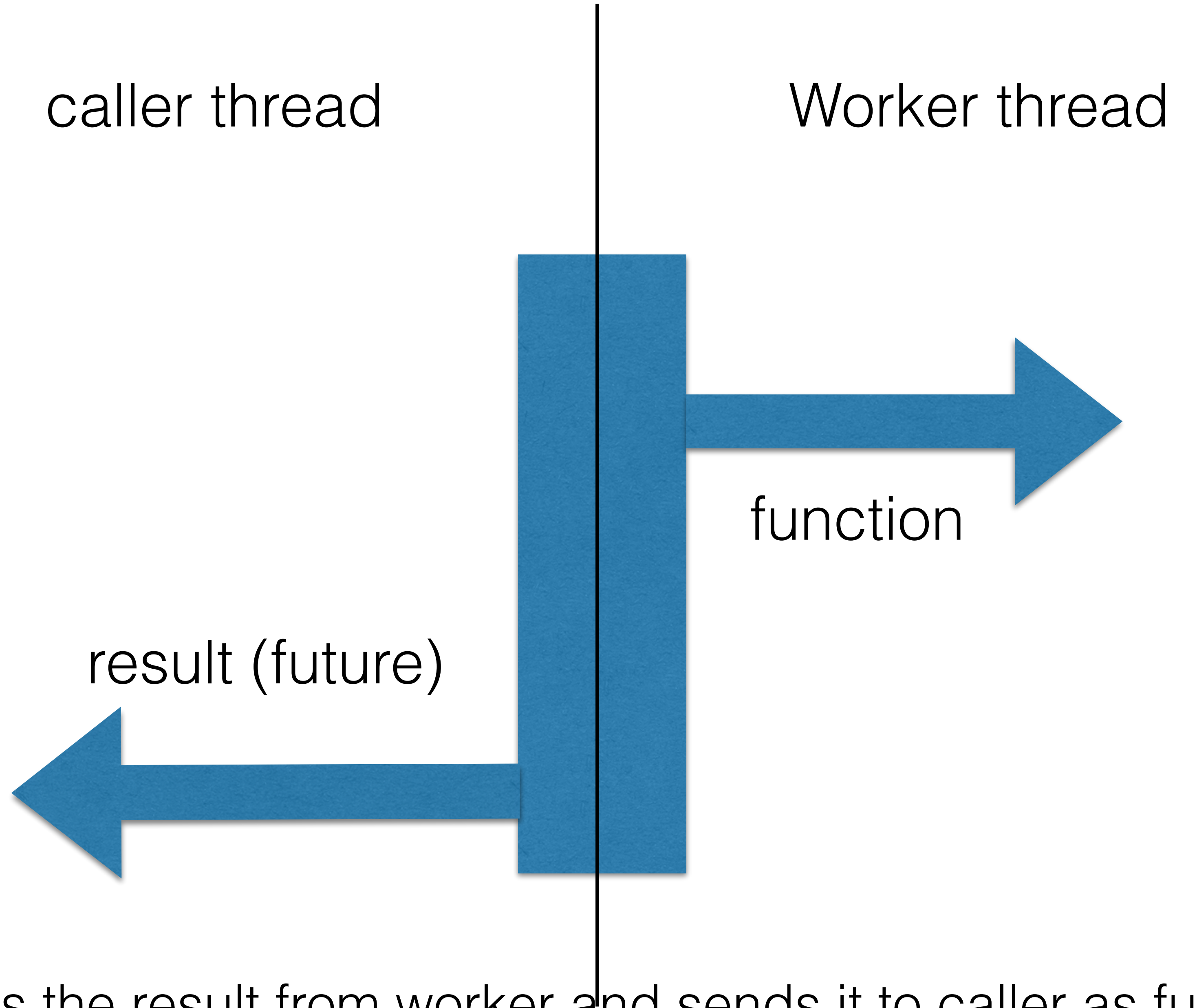
packaged_task

- General purpose function
- It is a callable
- It’s operator() is a void function that takes some parameters
- It’s get_future function eventually returns a future of computed result
- Can be passed to thread
- get the future and then send of task to thread

packaged_task

```
#include <future>

long fib(int position) {
    if (position < 2)
        return 1;
    else
        return fib(position - 1) + fib(position - 2);
}

long compute(int position) {
    cout << "computing in thread..." << std::this_thread::get_id() << endl;
    return fib(position);
}

int main()
{
    std::packaged_task<long(int)> task(compute);

    auto result = task.get_future();

    cout << "in main..." << std::this_thread::get_id() << endl;

    std::thread thread(std::move(task), 40);
    thread.detach();

    cout << "wait for result" << endl;
    cout << result.get() << endl;
```

What's really doing on?



- Takes the result from worker and sends it to caller as future

What if something goes wrong?

- Promise is like a packaged_task in that you can get a future from it
- The user of a Promise can either set a value or set an exception
- Use Promise as a mechanism to communicate between the worker thread and the caller

Using Promise

```
#include <future>

long fib(int position) {
    if (position < 0)
        throw string("invalid position");

    if (position < 2)
        return 1;
    else
        return fib(position - 1) + fib(position - 2);
}

void compute(int position, std::promise<long> resultPromise) {
    cout << "computing in thread..." << std::this_thread::get_id() << endl;
    try {
        long result = fib(position);
        resultPromise.set_value(result);
    }
    catch (...) {
        resultPromise.set_exception(std::current_exception());
    }
}
```

Using Promise

```
int main()
{
    vector<int> positions = { 10, -40 };

    for (auto position : positions) {
        try {
            std::promise<long> resultPromise;
            auto result = resultPromise.get_future();
            std::thread thread(compute, position, std::move(resultPromise));
            thread.detach();
            cout << result.get() << endl;
        }
        catch (const string& ex) {
            cout << ex << endl;
        }
    }
```

Thank you

venkats@agiledeveloper.com

@venkat_s