Synchronization

Task Programming in C++

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

License Information

Introduction

- Multi-threading in C++ has two main limitations
  1. The number of software threads may be higher than the number of hardware threads
     - **Over-subscription** occurs every time the number of software threads ready to start is higher than the number of hardware threads available in the system
     - Over-subscription implies system overhead and some performance penalty
     - One of the possibilities is to use thread pools (which is a generic methodology)

- To solve the problem, C++11 introduced **task-based** parallel programming

Introduction

- Multi-threading in C++ has two main limitations

  2. Threads (the std::thread library) do not offer any direct way to return a value to the caller
     - In POSIX
       - A simple strategy is return a value with pthread_exit
       - More general strategies must be user-implemented (through global or local objects)
     - In native C++ only the general strategy is available (you must manipulate objects explicitly)

➢ To solve this problem, C++11 introduced **futures** and **promises**

Task processing

- A task is an entity that runs asynchronously producing output data that will become available (and useful) at a later time
  - The operating system associate a **thread** to a **task** in an automatic way
  - Balancing tasks is automatic, through work-stealing features
    - The process is often implemented using a thread pool
  - Tasks have the possibility of handling return values

Task processing

- In C++
  - Thread-based parallel programming relies on **std::thread** objects

```
std::thread t(thread_function);
```

  - Task-based parallel programming relies on **std::asynch** objects

```
auto fut = std::asynch(thread_function);
```

From now on, we ignore std::

Task processing

```
#include <future>

future<T> async(policy, function, args...);
```

Parameters for the thread function

<T> is the type of the future

Asynchronous policy

"Thread" function

- Function async (namespace std)
  - Is an alternative to std::thread to execute functions in parallel
  - Has and extra parameter, i.e., the policy
  - Returns a future of type T

For now, ignore it

Task processing

- The user may decide the running policy
  - There are three different types of policies
    - The **deferred** policy is motivated exactly by the **oversubscription**, as the thread is run when the OS wants

Policy	Description
launch::async	Asynchronous launch, i.e., a new thread is generated to run the new function.
launch::deferred (Lazy threading)	The call to the new function is deferred. The OS may never run it. The new function will be run when we **wait for** it or **get** its future.
launch::async | launch::deferred (Default policy)	The policy to run the new thread is selected by the system accordingly to the availability of concurrency in the system. It is implementation dependent.

Examples

Running async tasks

```
auto f1 = std::async(std::launch::async, my_f, 10);
// Thread function my_f is run in a new thread

auto f2 = std::async(
  std::launch::deferred, my_f, 20);
// Thread function my_f is not run until we get
// its results or wait for it


f2.wait();
// Invoke deferred function f2 (i.e., run it)

auto f3 = std::async(
   std::launch::async | std::launch::deferred,
   my_f, 30);
// The system decides when running my_f.
// Possibly, it never runs my_f.
```

For now, we do not know what a future is

Force task f2 to be associated and run it within a thread

Example

Running policy
(part A)

```
auto f = std::async (my_f);

while (f.wait_for(100ms)!=std::future_status::ready) {

   ...

}
```

If it is async or deferred;
it may never run

Wait for 100 milliseconds.
Then, check the status

If it is not ready
f.wait_for returns
future_status::deferred
and the program cycles

The problem is that the program **can cycle forever,** as the policy is deferred

Example

Running policy (part B)

```
auto f = std::async (my_f);

if (f.wait_for(0s)==std::future_status::deferred) {

  f.wait();

} else {

  while (f.wait_for(100ms)!=
    std::future_status::ready) {

    ... do something ...

  }
  ...
}
```

If it is async or deferred; it may never run

Wait for 0 seconds, i.e., do not wait, check the status

It is deferred: Use wait to force the execution.
It is asynch, it is already running.

Check status every 100 milli-seconds

If it is not ready f.wait_for returns future_status::deferred

In this case, do something in parallel

Here the future f is ready

To wait for 0s or 100ms use std::literals

Futures

- An async object will eventually hold the return value of the thread function in a future
  - A **future** is an object that can represent a value generated by some provider
  - Function **<future>::get** applied to a valid future
    - Blocks the thread until the object is ready
    - Get the object (returned with "return" by the thread setting it) once the future is ready

<T> is the type of the future

```
future<T> async(policy, function, args...);
```

Example

Get a future at the end of a task

This library allows the passage of values from the thread and the caller

For example: Check if num is prime

Run a new task

Wait for function is_prime to return and make the Boolean value available

```
#include <future>
...
bool is_prime (int n) {
  if (num <= 1) return false;
  if (num <= 3) return true;
  ...
  return false;
}

int main () {
  std::future<bool> fut = std::async(
    std::launch::async, is_prime, 117);

  // ... do other work ...

  bool ret = fut.get();
  cout << ret;
  return 0;
}
```

Example

Get a future at the end of a task

Where the task is a lambda function

Run a new function thread

lambda expressions

Wait for the future to be ready and **get** the return value

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

...
auto fut = std::async (
  std::launch::async,
  [](){
    std::vector<int> v;
    for (int i=0; i<100; i++)
      v.push_back(i);
    return v;
  }
);

...

auto ret = fut.get();
for (auto e: ret)
  std::cout << e << std::endl;
```

Shared futures

- In C++ there are two types of futures
  - **Unique** future, i.e., std::future<T>
    - There is only one instance referring to the event
  - **Shared** future, i.e., std::shared_future<T>
    - A shared_future object behaves like a future object, except that it can be copied
    - Multiple instances **may refer to the same event**
    - All instances will become ready at the same time and can be retrieved
    - **May be used to signal multiple threads simultaneously**, similarly to std::condition_variable::notify_all

Example (buggy)

Unique future

```
int sum(int a, int b) {
  std::this_thread::sleep_for(std::chrono::seconds(2));
  return a + b;
}

int main() {
  std::future<int> fut =
    std::async(std::launch::async, sum, 10, 20);
  ...

  int result1 = fut.get();
  std::cout << "Result: " << result1 << endl;

  int result2 = fut.get();
  std::cout << "Result: " << result2 << endl;

  return 0;
}
```

Wait and then get the future

Result: 30

terminate called after throwing an instance of 'std::future_error'
what(): std::future_error: No associated state
Aborted (core dumped)

Example (correct)

Shared future

```
int sum(int a, int b) {
  std::this_thread::sleep_for(std::chrono::seconds(2));
  return a + b;
}

int main() {
  std::shared_future<int> fut =
    std::async(std::launch::async, sum, 10, 20);
  ...

  int result1 = fut.get();
  std::cout << "Result: " << result1 << endl;

  int result2 = fut.get();
  std::cout << "Result: " << result2 << endl;

  return 0;
}
```

Wait and then get the future

Result: 30

Result: 30

Promises

- At the highest level, you encounter a future when an async end
  - An async returns the future
  - The future represents a value that you do not yet have but will have eventually
- In other words, in all previous cases, the future is set when the task ends and issues a “return”
  - The std::future is associated with the return value of the function you launched
  - The future becomes "ready" when that function completes and returns its value (or throws an exception)

Promises

- At the lowest level, a future comes from an associated promise
  - A **promise** is an **object** that can store a value to be retrieved by a **future object**
    - A promise is an object that you will eventually set
    - When the value is set, it will make it available to its corresponding future
- A promise explicitly decouples the setting of the value (exception) from the end of the task's execution
  - The thread holding the promise **can set the future at any point during its execution**

Promises

- To summarize
  - The library std::async creates tasks that automatically return a future with the return statements
  - If we need a result **before** the task ends, we need to use promises and futures
    - The task set the value in the promise and client obain it in the corresponding future
  - Promises and futures can be used for **asynchronous communication** even with manually managed **thread objects** (not only with tasks)

Promises: Overall logic

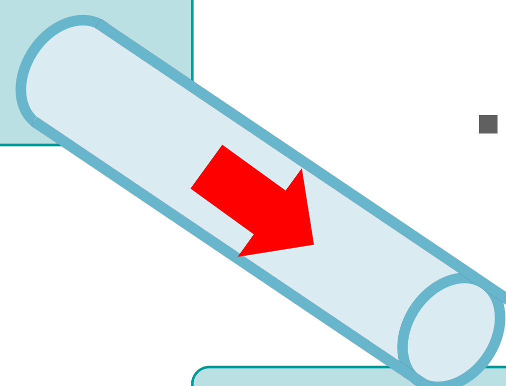
Promise and future "create" a one-shot channel for data

- A promise
  - Creates the channel
  - Writes data in the channel

```
std::promise<type> pn;

pn.set_value(...);
```

Promise Name



- A future
  - Connects to the other end of the channel
  - Waits and reads the data once it has been written

Future Name

```
auto fn = pn.get_future();

fn.get();
```

Promises: Implementation

The principal steps are

The main thread

- Defines a promise
- Associate a future to the promise

```
std::promise<type> pn;
auto fn = pn.get_future();
```

- Retrieves the result

```
fn.get();
```

The working thread

- Receives the promise
- Execute the function and fulfill the promise

```
pn.set_value(...);
```

Example

One thread (lambda)
Promise (thread) and future (main).

```
int f(int x) { return x + 1; }

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

  std::thread thread([&promise] (int x) {
    int result = f(x);
    promise.set_value(result);
    },
    5
  );

  std::cout << future.get() << std::endl;
}
```

Thread function

Define the promise
This is the producer-write end

Define the future from the promise
This is the consumer-read end

Launch f asyncronously

Get the promise in the capture list

Set the value (to be communicated) into the promise

Pass the argument 5 to the lambda

Get the value

Example

Two threads (lambda)
Promise (thread) and future (thread).

Define the promise and the future from the promise

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

auto producer = std::thread([&] {
  promise.set_value("Hello World");
});

auto consumer = std::thread([&] {
  std::cout << future.get();
});

producer.join();
consumer.join();
```

Run thread 1
It sets the promise

Run thread 2
It gets the future

Example

One thread (function)
Promise (thread) and future (main).

One-way communication:
The thread set the promise
and get the future

```
#include <future>
using namespace std;

void factorial (const int &N, promise<int>& pr) {
  int res = 1;
  for (int i=N; i> 1; i--)
  res *=i;
  pr.set_value(res);
}

int main () {
  promise<int> p;
  future<int> f = p.get_future();
  thread t = thread(factorial, 4, ref(p));
  // here we have the data
  int x = f.get();
  t.join();
}
```

Define the promise and the
future from the promise

ref generates an object of
type promise<int> to
hold a reference to p

Example

One task (function)
Two way synchronization

```
#include <future>

using namespace std;

int factorial (std::future<int>& f ) {
  int res = 1;
  int N = f.get();
  for ( int i=N; i> 1; i-- )
  res *=i;
  return res;
}

int main () {
  std::promise<int> p;
  std::future<int> f = p.get_future();
  std::future<int> fu =
    async(std::launch::async,factorial,std::ref(f));
  p.set_value(4);
  int x = fu.get();
}
```

Two-ways communication:
The caller set the promise and get the future

The future must be passed by reference, since it doesn't support copy semantics

Define the promise and the future from the promise

Example

Two tasks (functions)
Two way synchronization

```
void func1 (promise<int> p) {
  int res = 18;
  p.set_value(res);
}

int func2 (future<int> f) {
  int res=f.get();
  return res;
}

int main () {
  promise<int> p;
  future<int> f = p.get_future();
  future<void> fu1 = async(func1, move(p) );
  future<int> fu2 = async(func2, move(f) );
  int x = fu2.get();
  return 0;
}
```

The move semantics is achieved by std::move

Example

Shared future

```
usign namespace std;
int factorial (shared_future<int> f) {
  int res = 1;
  int N = f.get();
  for (int i=N; i> 1; i--) res *=i;
  return res;
}

int main () {
  promise<int> p;
  future<int> f = p.get_future();
  shared_future<int> sf = f.share();

  future<int> fu1 = async(std::launch::async, factorial, sf);
  future<int> fu2 = async(std::launch::async, factorial, sf);
  future<int> fu3 = async(std::launch::async, factorial, sf);


  p.set_value(4);
  int r1=fu1.get(); int r2=fu2.get(); int r3=fu3.get();
  return 0;
}
```

Observations

- If we destroy the promise without setting a value
  - The object will store an exception
    - Function **get** will return
  - The object associated to a promise **is usually stored in the heap** as it cannot be stored
    - In the setter of the promise, as the setter can die
    - In the getter of the future, as we futures can be shares among several getters

Observations

- A single promise-future pair is designed to transmit **one value, exactly once**
  - It's a single-use communication channel for one result
  - You cannot use one promise to send multiple, independent values sequentially to the same future
  - Calling set_value more than once on the same promise results in an exception

Observations

- How do you "return" multiple values?
  - If a task needs to produce multiple pieces of data as its single result, you can set the promise with a type that contains multiple values, such as

```
std::promise<std::vector<int>>
std::promise<std::pair<double, std::string>>
```

Observations

- How do you "return" multiple results at different times?
  - If a task must communicate multiple results in different moments, promise and future are not the right tool for that specific pattern
  - You would typically use other concurrency mechanisms like
    - Multiple promise/future pairs (one for each distinct result)
    - A thread-safe queue (protected by a std::mutex, or a custom concurrent queue)
    - Condition variables for more complex signaling

Conclusions

- The task-based approach
  - Makes the OS in charge of the parallelism
  - Makes the return value of a thread/task accessible
  - Run threads with a smart policy
    - CPU load balancing
      - The C++ library can run the function without spawning a thread
    - Avoid the raising of **std::system_error** in case the thread number reaches the system limit
  - Allows futures to catch exceptions thrown by the function
    - With **std::thread** the program terminates

Conclusions

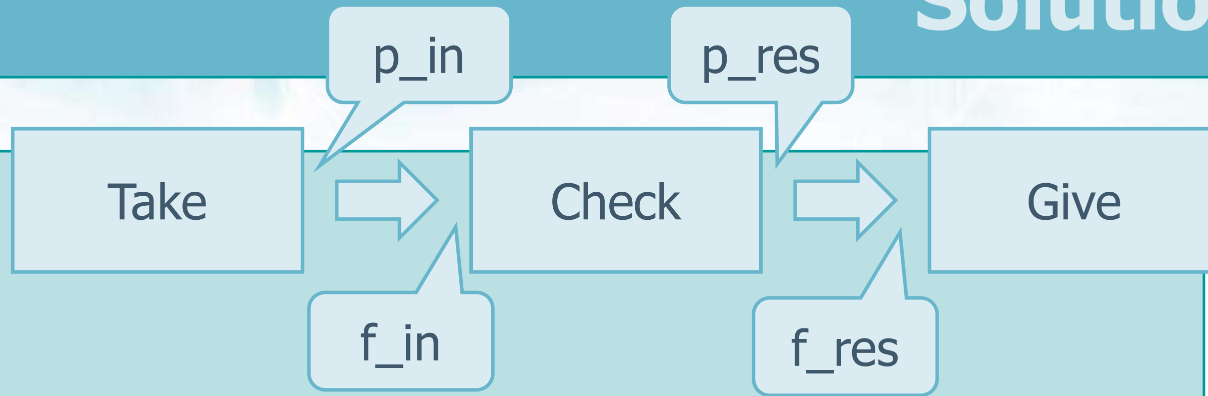
- Thread-based approach
  - Is used to execute tasks that do not terminate till the end of the application
    - A thread entry point function is like a second, concurrent **main**
  - It is a more general concurrency model
    - Can be used for thread-based design patterns
  - Allows us to access to the pthread native handle
    - Makes the programmer in charge of the parallelism
    - Useful for advanced management (priority, affinity, scheduling policies, etc.)

Exercise 01

Exam 5 July 2021

- Write a C++ program with three tasks
  - Thread **take** reads a number from command line
  - Thread **check** checks whether the number is prime
  - Thread **give** displays the answer to standard output
- Thread communication should be made using promises and futures
  - All functions are acyclic

Solution

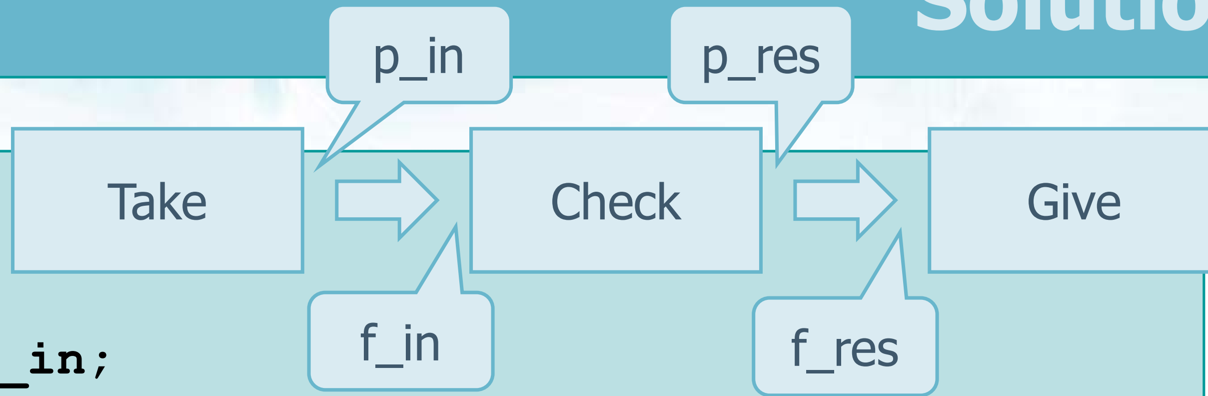



```
#include <iostream>
#include <thread>
#include <vector>
#include <future>

void take (std::promise<int>&);
void check (std::future<int>&, std::promise<bool>&);
void give (std::future<bool>&);
```

Thread functions

Solution




```
int main(){
  std::promise<int> p_in;
  std::future<int> f_in = p_in.get_future();

  std::promise<bool> p_res;
  std::future<bool> f_res = p_res.get_future();

  std::thread t1(take, std::ref(p_in));
  std::thread t2(check, std::ref(f_in), std::ref(p_res));
  std::thread t3(give, std::ref(f_res));

  t1.join();
  t2.join();
  t3.join();
  return 0;
}
```

Solution

Reading thread

```
void take (std::promise<int> &p_in) {
  int in;
  std::cout << “Insert a number" << std::endl;
  std::cin >> inp;
  p_in.set_value (in);
}
```

Set promise “in”

Writing thread

```
void give (std::future<bool>& f_res) {
  bool answer = f_res.get();
  std::string s0 (" ");
  if(!answer)
    s0=" NOT";
  std::cout << "Number is" << s0 << " prime";
}
```

Get future “ref”

Solution

Computation thread

```
void check (
  std::future<int> &f_in, std::promise<bool>& p_res)
{
    int n = f_in.get();
    bool prime=true;
    if (n <= 1){
      prime = false;
    }
    // Check from 2 to n-1
    for (int j=2; j<n; j++) {
      if (n % j == 0) {
        prime = false;
        break;
      }
    }
    p_res.set_value(prime);
}
```

Get future "in"

Set promise "res"