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
     - With threads, the global load must be managed manually

- To solve this 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
  - 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);
```

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

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 runt it.The new function will be run when we **wait** 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

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.


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

For now, we do not know what a future is

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

Example

Running policy

```
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 msecons

If it is not ready, 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
    - Returns the object (return with "return") once it is ready

<T> is the type of the future

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

Example

```
#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;
}
```

Check if num is prime

Run a new function in a thread

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

Example

```
#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;
```

Run a new function thread

Work with **lambda** expressions

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

Shared futures

- In C++ there are two types of future
  - 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 intances 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

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

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

- The most common situation where you encounter a future is with a call to an async
  - An async returns a future
  - A future represents a value that you do not yet have but will have eventually
- 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 be made available through its corresponding future

Promises

- Think of promise and future as creating a single-use channel for data
  - A promise
    - Creates the channel
    - Writes data in the channel

Promise Name

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

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

  - 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

- 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();
```

  - The working thread
    - Receive the promise
    - Executes the function and fulfills the promise

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

  - The main thread retrieves the result

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

Promises

- Further considerations
  - If we destroy the promise without setting a value, an exception is stored in the object
    - 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

Example

Promise and future.
One thread (lambda)

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

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

// Launch f asynchronously
std::thread thread([&promise] (int x) {
    promise.set_value(f(x));
  }, 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

Get the promise in the capture list

Set the value (to be communicated) into the promise

Get the value

Example

Promise and future.
Two threads (lambdas)

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 the thread setting the promise

Run the thread getting the future

Example

Promise and future.
One thread (function)

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 async task.
Two way sync

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

```
#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();
}
```

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 async tasks.
Two way sync

```
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;
}
```

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 of thread number reached 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

- Resorting only to asynchrnous tasks, write a C++ program to perform the matrix multiplication
  - C = A x B
- Constraints
  - The number of columns in A must equal the number of rows in B
  - Use C++ containers
    - Implement the matrices as vector of vectors
      - std::vector<std::vector<int>> a, b;
  - To compute a sum of products, it is possible to use the function **std::inner_product**

Solution

```
#include ...

void generateRandomMatrix (
  vector<vector<int>>& m, int nRow, int nCol){
  for (int i = 0; i < nRow; i++){
    vector<int> row;
    for (int j = 0; j < nCol; j++)
      row.push_back(rand()%3);
    m.push_back(row);
  }
}

void printMatrix(const std::vector<std::vector<int>>& m){
  for (auto & row : m){
    for (int element : row)
      cout << element << " ";
    cout << endl;
  }
}
```

Utility function:
Generate a random matrix

Utility function:
Display a matrix

Solution

Function threads

```
int computeSumOfProducts (
  const std::vector<int>& v1, const std::vector<int>& v2){

  return std::inner_product (
    v1.begin(), v1.end(), v2.begin(), 0
  );
}
```

Return result

Until C++11
Computes the sum of products on the range [v1.begin, v1.end] and the range beginning at v2.begin, initializing the accumulator at 0

Solution

Main thread

```
int main() {
  int nRowA = 2, nColA = 3,  nRowB = 3, nColB = 2;
  vector<std::vector<int>> a, b;
  vector<std::vector<std::future<int>>> futures;
  ofstream outputFile;

  cout << "Insert size of matrix A" << endl;
  cin >> nRowA >> nColA;
  cout << "Insert size of matrix B" << endl;
  cin >> nRowB >> nColB;

  if(nColA != nRowB) {
    cout << "Wrong size colA != rowB" << endl;
    return -1;
  }
```

Matrices

Matrix of futures

Read the size of A and B

Solution

```
generateRandomMatrix(a, nRowA, nColA);
// generate the transpose of B (columns -> rows)
// so that we can easily access the columns of B
// for the multiplication
generateRandomMatrix(b, nColB, nRowB);

// Generate the futures to compute the products
for (int i = 0; i < nRowA; i++){
  vector<std::future<int>> futureRow;
  for (int j = 0; j < nColB; j++) {
    future<int> f = std::async(
      std::launch::async | std::launch::deferred,
      computeSumOfProducts, a[i], b[j]);
    // Futures are not copyable so we have to use move
    // to store them in a vector
    futureRow.push_back(std::move(f));
  }
  futures.push_back(std::move(futureRow));
}
```

Generate matrix A

Generate matrix B

Create a future for each element

Insert futures in the matrix of futures

Solution

```
  cout << "A" << endl;
  printMatrix(a);
  cout << "B" << std::endl;
  printMatrix(b);

  outputFile.open("./output-matrix.txt");
  for (auto & row : futures){
    for (std::future<int> & f : row) {
       outputFile << f.get() << " ";
    }
    outputFile << std::endl;
  }
  outputFile.close();

  return 0;
}
```

Display input matrices

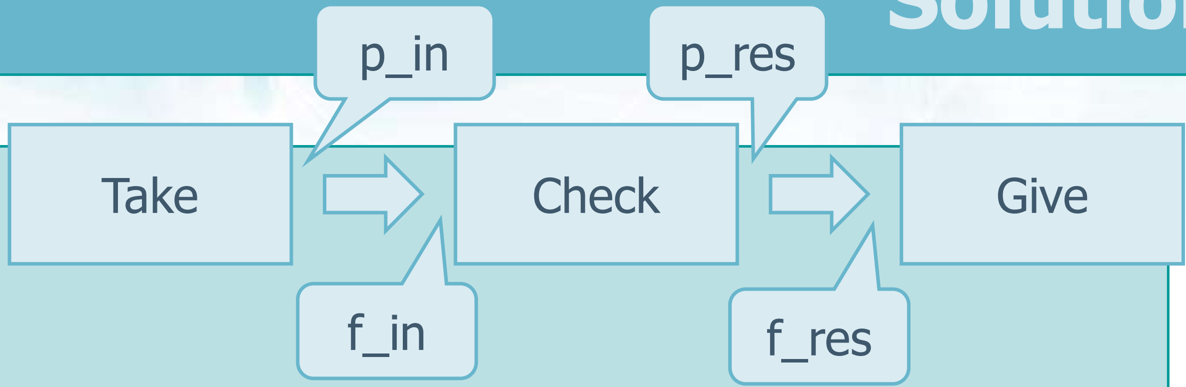
Wait futures to be ready and display result

Exercise

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

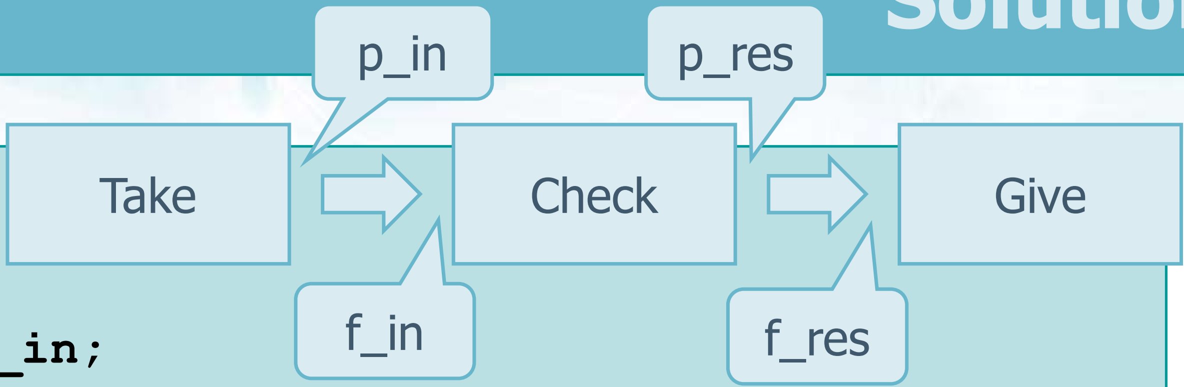


Thread functions

```
#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>&);
```

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"