With std::promise and std::future, you have the full control over the task.

Full control over the task

A std::promise permits

* to set a value, a notification or an exception. That result can in addition delayed be provided by the promise.

A std::future permits to

* pick up the value from the promise.
* asks the promise, if the value is available.
* wait for the notification of the promise. That waiting can be done with a relative time duration or an absolute time point. => Replacement for [condition variables.](http://modernescpp.com/index.php/condition-variables)
* create a shared future (std::shared\_future).

Both [communication endpoints](http://modernescpp.com/index.php/tasks) promise and future can be moved in a separate thread. So the communication is taking place between threads.

// promiseFuture.cpp

#include <future>

#include <iostream>

#include <thread>

#include <utility>

void product(std::promise<int>&& intPromise, int a, int b){

intPromise.set\_value(a\*b);

}

struct Div{

void operator() (std::promise<int>&& intPromise, int a, int b) const {

intPromise.set\_value(a/b);

}

};

int main(){

int a= 20;

int b= 10;

std::cout << std::endl;

// define the promises

std::promise<int> prodPromise;

std::promise<int> divPromise;

// get the futures

std::future<int> prodResult= prodPromise.get\_future();

std::future<int> divResult= divPromise.get\_future();

// calculate the result in a separat thread

std::thread prodThread(product,std::move(prodPromise),a,b);

Div div;

std::thread divThread(div,std::move(divPromise),a,b);

// get the result

std::cout << "20\*10= " << prodResult.get() << std::endl;

std::cout << "20/10= " << divResult.get() << std::endl;

prodThread.join();

divThread.join();

std::cout << std::endl;

}

Per default, there is a one-to-one relation between the promise and the future. But std::shared\_future supports a one-to-many relations between a promise and many futures.

std::shared\_future

A std::shared\_future 

* permits you to ask the promise independent of the other associated futures.
* has the same interface as a std::future.
* can be created by a std::future fut with the call fut.share().
* can be created by a std::promise divPromise with the call std::shared\_future<int> divResult= divPromise.get\_future().

The managing of std::shared\_future is special.

// sharedFuture.cpp

#include <exception>

#include <future>

#include <iostream>

#include <thread>

#include <utility>

std::mutex coutMutex;

struct Div{

void operator()(std::promise<int>&& intPromise, int a, int b){

try{

if ( b==0 ) throw std::runtime\_error("illegal division by zero");

intPromise.set\_value(a/b);

}

catch (...){

intPromise.set\_exception(std::current\_exception());

}

}

};

struct Requestor{

void operator ()(std::shared\_future<int> shaFut){

// lock std::cout

std::lock\_guard<std::mutex> coutGuard(coutMutex);

// get the thread id

std::cout << "threadId(" << std::this\_thread::get\_id() << "): " ;

// get the result

try{

std::cout << "20/10= " << shaFut.get() << std::endl;

}

catch (std::runtime\_error& e){

std::cout << e.what() << std::endl;

}

}

};

int main(){

std::cout << std::endl;

// define the promises

std::promise<int> divPromise;

// get the futures

std::shared\_future<int> divResult= divPromise.get\_future();

// calculate the result in a separat thread

Div div;

std::thread divThread(div,std::move(divPromise),20,10);

Requestor req;

std::thread sharedThread1(req,divResult);

std::thread sharedThread2(req,divResult);

std::thread sharedThread3(req,divResult);

std::thread sharedThread4(req,divResult);

std::thread sharedThread5(req,divResult);

divThread.join();

sharedThread1.join();

sharedThread2.join();

sharedThread3.join();

sharedThread4.join();

sharedThread5.join();

std::cout << std::endl;

}

Both work packages of the promise and the future are in this current example function objects. In case you divide to numbers, you have to take care of the denominator. It must not be 0. If is is 0, you get an exception. The promise deals with this issue by catching the exception (line 18 - 20) and rethrowing it to the future. The std::future catches the exception and displays it in line 40. In line 58, divPromise will be moved and executed in divThread. Accordingly, the std::shared\_future's are **copied** in the five threads. I will emphasis this once more. In opposite to a std::future object, which can only be moved, you can copy a std::shared\_future object.

The main thread waits In the line 69 to 73 for its child´s and shows the results.