Tasks

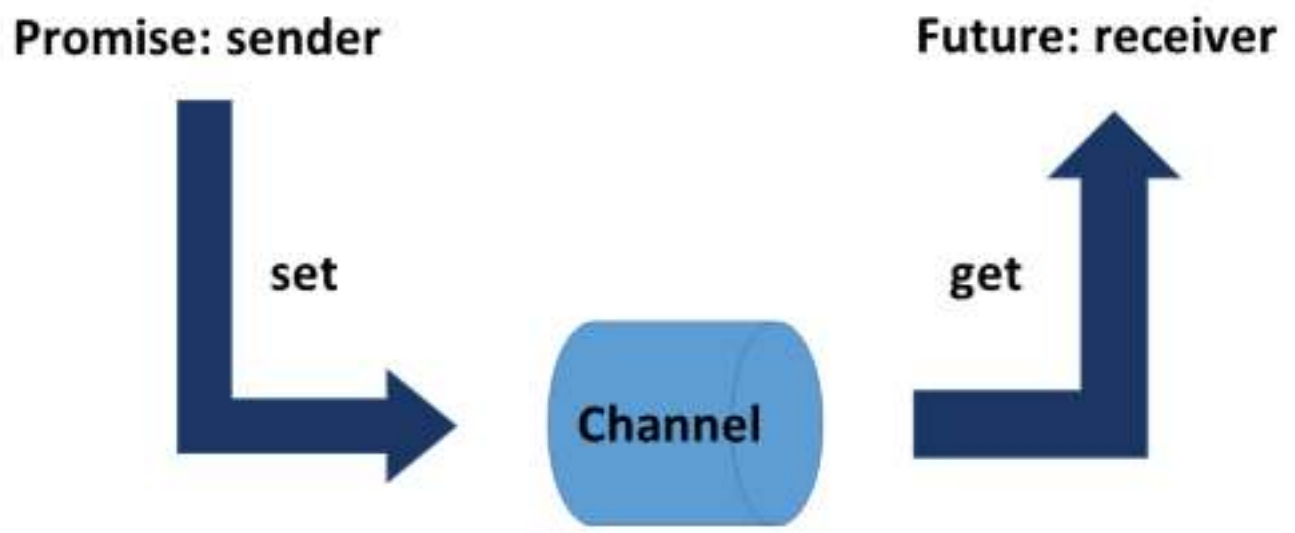
Now that we've learned about threads, let's discuss tasks for asynchronous programming.

WE'LL COVER THE FOLLOWING

- Threads versus Tasks
- std::async
- std::packaged_task
- std::promise and std::future

In addition to threads, C++ has tasks to perform work asynchronously. Tasks need the header . A task is parameterized with a work package and consists of the two associated components, a promise, and a future. Both are connected via a data channel. The promise executes the work packages and puts the result in the data channel; the associated future picks up the result. Both communication endpoints can run in separate threads. What's special is that the future can pick up the result at a later time. Therefore the calculation of the result by the promise is independent of the query of the result by the associated future.

> 🔑 **Regard tasks as data channels**
> Tasks behave like data channels. The promise puts its result in the data channel. The future waits for it and picks it up.



Threads versus Tasks

Threads are very different from tasks. For the communication between the creator thread and the created thread, you have to use a shared variable. The task communicates via its data channel, which is implicitly protected. Therefore a task must not use a protection mechanism like a mutex. The creator thread is waiting for its child with the `join` call. The future `fut` is using the `fut.get()` call which is blocking if no result is there.

If an exception happens in the created thread, the created thread terminates and therefore the creator and the whole process. On the contrary, the promise can send the exceptions to the future, which has to handle the exception.

A promise can serve one or many futures. It can send a value, an exception or only a notification. You can use a task as a safe replacement for a condition variable.

```
#include <future>
#include <thread>
//...
int res;
std::thread t([&]{ res= 2000+11;});
t.join();
std::cout << res << std::endl;
// 2011
auto fut= std::async([]{ return 2000+11; });
std::cout << fut.get() << std::endl; // 2011
```

The child thread `t` and the asynchronous function call `std::async` calculates

the sum of 2000 and 11. The creator thread gets the result from its child thread `t` via the shared variable `res`. The call `std::async` creates the data channel between the sender (promise) and the receiver (future). The future asks the data channel with `fut.get()` for the result of the calculation. The `fut.get` call is blocking.

std::async

`std::async` behaves like an asynchronous function call. This function call takes a callable and its arguments. `std::async` is a variadic template and can, therefore, take an arbitrary number of arguments. The call of `std::async` returns a future object `fut`. That's your handle for getting the result via `fut.get()`. Optionally you can specify a start policy for `std::async`. You can explicitly determine with the start policy if the asynchronous call should be executed in the same thread (`std::launch::deferred`) or in another thread (`std::launch::async`). What's special about the call auto `fut = std::async(std::launch::deferred, ... )` is that the promise will not immediately be executed. The call `fut.get()` lazy starts the promise.

```
//...
#include <future>
//...
using std::chrono::duration;
using std::chrono::system_clock;
using std::launch;
auto begin= system_clock::now();
auto asyncLazy= std::async(launch::deferred, []{ return system_clock::now(); });
auto asyncEager= std::async(launch::async, []{ return system_clock::now(); });
std::this_thread::sleep_for(std::chrono::seconds(1));
auto lazyStart= asyncLazy.get() - begin;
auto eagerStart= asyncEager.get() - begin;
auto lazyDuration= duration<double>(lazyStart).count();
auto eagerDuration= duration<double>(eagerStart).count();
std::cout << lazyDuration << " sec"; // 1.00018 sec.
std::cout << eagerDuration << " sec"; // 0.00015489 sec.
```

The output of the program shows that the promise associated with the future `asyncLazy` is executed one second later than the promise associated with the future `asyncEager`. One second is exactly the time duration the creator is sleeping before the future `asyncLazy` asks for its result.

> 🔑 **std::async should be your first choice**
> The C++ runtime decides if `std::async` is executed in a separated thread. The decision of the C++ runtime may be dependent on the number of

> The decision of the C++ runtime may be dependent on the number of
>
> your cores, the utilization of your system or the size of your work package.

std::packaged_task

`std::packaged_task` enables you to build a simple wrapper for a callable, which can later be executed on a separate thread. Therefore four steps are necessary.

1. **Wrap your work:**

```
std::packaged_task<int(int, int)> sumTask([](int a, int b){ return a+b; });
```

2. **Create a future:**

```
std::future<int> sumResult= sumTask.get_future();
```

3. **Perform the calculation:**

```
sumTask(2000, 11);
```

4. **Query the result:**

```
sumResult.get();
```

You can move either the `std::package_task` or the `std::future` in a separate thread.

```
//...
#include <future>
//...
using namespace std;
struct SumUp{
  int operator()(int beg, int end){
    for (int i= beg; i < end; ++i )
      sum += i;
    return sum;
  }
private:
  int beg;
  int end;
  int sum{0};
```

```
};

SumUp sumUp1, sumUp2;

packaged_task<int(int, int)> sumTask1(sumUp1);
packaged_task<int(int, int)> sumTask2(sumUp2);

future<int> sum1= sumTask1.get_future();
future<int> sum2= sumTask2.get_future();

deque< packaged_task<int(int, int)>> allTasks;

allTasks.push_back(move(sumTask1));
allTasks.push_back(move(sumTask2));

int begin{1};
int increment{5000};
int end= begin + increment;

while (not allTasks.empty()){
  packaged_task<int(int, int)> myTask= move(allTasks.front());
  allTasks.pop_front();
  thread sumThread(move(myTask), begin, end);
  begin= end;
  end += increment;
  sumThread.detach();
}
auto sum= sum1.get() + sum2.get();
cout << sum;
```

The promises `(std::packaged_task)` are moved into the `std::deque allTasks`. The program iterates in the while loop through all promises. Each promise runs in its thread and performs its addition in the background `(sumThread.detach())`. The result is the sum of all numbers from 0 to 100000.

std::promise and std::future

The pair `std::promise` and `std::future` gives the full control over the task.

Method	Description
`prom.swap(prom2)` and `std::swap(prom, prom2)`	Swaps the promises.
`prom.get_future()`	Returns the future.
`prom.set_value(val)`	Sets the value.
`prom.set_exception(ex)`	Sets the exception.

`prom.set_value_at_thread_exit(val)`	Stores the value and makes it ready if the promise exits.
`prom.set_exception_at_thread_exit(ex)`	Stores the exception and makes it ready if the promise exits.

If the promise sets the value or the exception more then once an `std::future_error` exception is thrown.

Method	Description
`fut.share()`	Returns a `std::shared_future`.
`fut.get()`	Returns the result which can be a value or an exception.
`fut.valid()`	Checks if the result is available. Returns after the call `fut.get()` false.
`fut.wait()`	Waits for the result.
`fut.wait_for(relTime)`	Waits for a time duration for the result.
`fut.wait_until(absTime)`	Waits until a time for the result.

If a future `fut` asks more than once for the result, an `std::future_error` exception is thrown. The future creates a shared future by the call `fut.share()`. Shared future are associated with their promise and can independently ask for the result. A shared future has the same interface as a future. Here is the usage of promise and future.

```
//...
```

```
#include <future>
//...
void product(std::promise<int>&& intPromise, int a, int b){

  intPromise.set_value(a*b);
}
int a= 20;
int b= 10;
std::promise<int> prodPromise;
std::future<int> prodResult= prodPromise.get_future();
std::thread prodThread(product, std::move(prodPromise), a, b);
std::cout << "20*10= " << prodResult.get(); // 20*10= 200
```

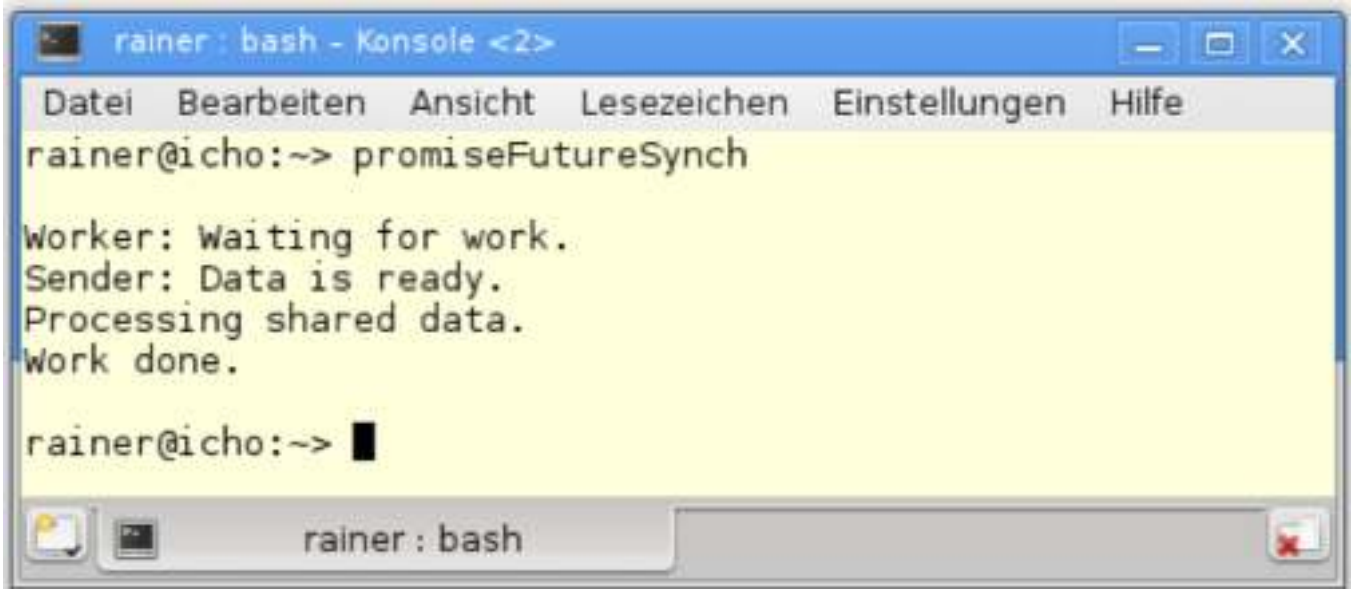
The promise `prodPromise` is moved into a separate thread and performs its calculation. The future gets the result by `prodResult.get()`.

> 🔑 **Use promise and future for the synchronisation of threads**
> A future `fut` can be synchronized with its associated promise by the call `fut.wait()`. Contrary to condition variables, you need no locks and mutexes, and spurious and lost wakeups are not possible.

```
//...
#include <future>
//...
void doTheWork(){
  std::cout << "Processing shared data." << std::endl;
}
void waitingForWork(std::future<void>&& fut){
  std::cout << "Worker: Waiting for work." << std::endl;
  fut.wait();
  doTheWork();
  std::cout << "Work done." << std::endl;
}
void setDataReady(std::promise<void>&& prom){
  std::cout << "Sender: Data is ready." << std::endl;
  prom.set_value();
}
std::promise<void> sendReady;
auto fut= sendReady.get_future();

std::thread t1(waitingForWork, std::move(fut));
std::thread t2(setDataReady, std::move(sendReady));
```

The call of the promise `prom.set_value()` wakes up the future which then can perform its work.


rainer : bash - Konsole <2>
Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe
rainer@icho:~> promiseFutureSynch
Worker: Waiting for work.
Sender: Data is ready.
Processing shared data.
Work done.
rainer@icho:~>
rainer : bash