The parent of a thread has to take care of its child. The parent can wait, until its child is done or detach itself from its child. But that is not really new. But that will not hold for std::async. The big charm of std::async is, that the parent has not take to care of its child.

Fire and forget

std::async creates special futures. These futures wait in its destructor, until the work of the associated promise is done. That is the reason, why the creator has not to take care of its child. But it gets even better. You can execute a std::future as a fire and forget job. The by std::async created future will be executed just in place. Because the std::future fut is in this case not bound to a variable, it's not possible to invoke fut.get() or fut.wait() on the future to get the result of the promise.

Maybe, my last sentences were a bit too confusing. So I'll compare an ordinary future with a fire and forget future. It is necessary for a fire and forget future, that the promise runs in a separate thread to start immediately with its work. This is done by the std::launch::async policy. You can read the details to the launch policy in the post [asynchronous function calls](http://modernescpp.com/index.php/asynchronous-function-calls).

auto fut= std::async([]{return 2011;});

std::cout << fut.get() << std::endl; /// 2011

std::async(std::launch::async,[]{std::cout << "fire and forget" << std::endl;}); // fire and forget

The fire and forget futures have a big charm. They will run in place and execute there work package, without the creator taking care of them. The simple example shows the described behaviour.

// async.cpp

#include <iostream>

#include <future>

int main() {

std::cout << std::endl;

std::async([](){std::cout << "fire and forget" << std::endl;});

std::cout << "main done " << std::endl;

}

One after another

The future, that is created by std::async, waits in its destructor, until its work is done. An other word for waiting is blocking. The future blocks the progress of the program in its destructor. The becomes obvious, in case you use fire and forget futures.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | // blocking.cpp  #include <chrono>  #include <future>  #include <iostream>  #include <thread>  int main(){  std::cout << std::endl;  std::async(std::launch::async,[]{  std::this\_thread::sleep\_for(std::chrono::seconds(2));  std::cout << "first thread" << std::endl;  });    std::async(std::launch::async,[]{  std::this\_thread::sleep\_for(std::chrono::seconds(1));  std::cout << "second thread" << std::endl;}  );    std::cout << "main thread" << std::endl;  } |

The program executes two promises in their own thread. The resulting futures are fire and forget futures. These futures block in their destructor until the associated promise is done. The result is, that the promise will be executed with high probability in that sequence, in which you find them in the source code. That is exactly what you see in the output of the program.

I want to stress this point once more. Although I create in the main-thread two promises, which are executed in separate threads, the threads run in sequence one after the other. That is the reason, why the thread with the more time consuming work package (line 12) finishes first. Wow, that was disappointing. Instead of three threads running concurrently, each thread will be executed after another.

The key issue is, that the by std::async created thread is waiting in its destructor until the associated promise is done, can not  be solved. The problem can only be mitigated. In case you bind the future to a variable, the blocking will take place ‎在时间点‎, when the variable goes out of scope. That is the behaviour, you can observe in the next example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | // notBlocking.cpp  #include <chrono>  #include <future>  #include <iostream>  #include <thread>  int main(){  std::cout << std::endl;  auto first= std::async(std::launch::async,[]{  std::this\_thread::sleep\_for(std::chrono::seconds(2));  std::cout << "first thread" << std::endl;  });    auto second= std::async(std::launch::async,[]{  std::this\_thread::sleep\_for(std::chrono::seconds(1));  std::cout << "second thread" << std::endl;}  );    std::cout << "main thread" << std::endl;  } |

Now, the output of the program matches our intuition, because the three threads are executed in parallel. The future first (line 12) and second (line 17) are valid until the end of the main-function (line 24). So, the destructor will perhaps blocks at this time point. The result is, that the threads with the smallest work package is the fastest one.

## It's not so bad

I have to admit, my usage of std::async creates futures very contrived. At first, the futures were not bound to a variable. At second, I didn't use the future to pick up the result from the promise by a get or wait call. Exactly in that situation, we can observe the strange behaviour, that the future blocks in its destructor.

The key reason for these posts was it, to show, that a fire and forget future, that is not bound to a variable, must be handled with great care. But this point doesn't hold for futures, which are created by std::packaged\_task or std::promise.