Subject: Job package: A lightweight worker thread for non-blocking operations.
Posted by Oblivion on Sun, 10 Sep 2017 10:29:46 GMT
View Forum Message <> Reply to Message

Hello,

Below you will find a worker thread implementation which -hopefully- can simplify creating non-blocking or asynchronous applications,
or allow you to easily port your applications to MT-era. I also supplied an example code demonstrating its basic usage pattern.

Job package for Ultimate++

This template class implements a scope bound, single worker thread based on RAII principle. It provides a return semantics for result gathering, functionally similar to promise/future pattern (including void type specialization). Also it provides a convenient error management and exception propagation mechanisms for worker threads, and it is compatible with U++ single-threaded mode.

Note that while Job is a general purpose multithreading tool, for high performance loop parallelization scenarios CoWork would be a more suitable option. This class is mainly designed to allow applications and libraries to gain an easily managable, optional non-blocking behavior where high latency is expected such as network operations and file I/O, and a safe, container-style access to the data processed by the worker threads is preferred.

Features and Highlights

- A safe way to gather results from worker threads.
- Simple and easy-to-use thread halting, and error reporting mechanism.
- Exception propagation.
- External blocking is possible.
- Optional constant reference access to job results.
- Compatible with U++ single-threaded environment.
- All Job instances are scope bound and will forced to finish job when they get out of scope.

Known Issues

- Currently none.

History

- 2017-10-07: Compatibility with U++ single-threaded mode is added.

- 2017-10-01: Global variables moved into JobGlobal namespace in order to avoid multiple definitions error. Accordingly, global functions are defined in Job.cpp.

- 2017-09-22: Exception propagation mechanism for job is properly added. From now on worker threads will pass exceptions to their caller.
Void template specialization is re-implemented (without using future/promise).
Constant reference access operator is added. This is especially useful where the data is a container with iterators (such as Vector or Array).

- 2017-09-19: std::exception class exceptions are handled, and treaated as errors. (For the time being.)
void instantiation is now possible.
Jobs will notify their workers on shutdown.
Clean up & cosmetics...

- 2017-09-18: Clear() method is added. Worker id generator is using int64. Documentation updated.

- 2017-09-17: Future/promise mechanism, and std template library code completely removed. From now on Job has its own result gathering mechanism with zero copy/move overhead.

- 2017-09-16: Job is redesigned. It is now a proper worker thread.

- 2017-09-10: Initial public beta version is released.

Hope you'll find it useful.

Best Regards,
Oblivion

File Attachments

1) Job Package and Examples.zip, downloaded 507 times

Subject: Re: Job package: A lightweight multithreading tool, using promise/future mechanism
Posted by mirek on Sun, 10 Sep 2017 13:08:26 GMT
View Forum Message <> Reply to Message

Trying to figure out what is relative advantage of using Job vs CoWork vs C++11 future/promise....

For now, it looks like Job is not using worker threads, creating thread for each run. Which actually can have advantage if there are blocking operations (file I/O etc), but that is meant to be dealt with in CoWork by increasing the number of threads.

Other than that, your examples can be rewritten with CoWork easily. "WaitForJobs" is outright ugly, being single global function for all Jobs; I think CoWork::Finish has a clear upper hand here...

```
{
 // Print a message to stdout.
 // Returns void.
 CoWork job;
 job & [=] { Cout() << "Hello world!\n"; };
 job.Finish();
}

{
 // Print all the divisors of random numbers.
 // Returns a String.
 CoWork jobs;
 Vector<String> results;
 for(int i = 0; i < 5; i++)
  jobs & [=, &results] { String h = GetDivisors2(); CoWork::FinLock(); results.At(i) = GetDivisors();
};
 jobs.Finish();
 for(auto r : results)
  Cout() << r << '\n';
}
```

Subject: Re: Job package: A lightweight multithreading tool, using promise/future mechanism
Posted by mirek on Sun, 10 Sep 2017 13:56:16 GMT
View Forum Message <> Reply to Message

BTW, as exercise:

```
#include <CtrlLib/CtrlLib.h>

#include <future>

using namespace Upp;

template< class Function, class... Args>
std::future<std::result_of_t<std::decay_t<Function>(std::decay_t<Args>...)>>
Async(Function&& f, Args&&... args )
{
```

```
 std::promise<std::result_of_t<std::decay_t<Function>(std::decay_t<Args>...)>> p;
 auto ftr = p.get_future();
 CoWork::Schedule([=, p = pick(p)]() mutable {
  p.set_value(f(args...));
 });
 return ftr;
}

GUI_APP_MAIN
{
 DDUMP(Async([] { return "Hello world"; }).get());
}
```

Still not sure about real world scenarion where I would prefer using future/promise over CoWork.

Maybe my problem with future/promise really is that fact that usually the "result" of async operation as a change in some data that gets into it as reference. future forces me to do a copy to store the function result.

Subject: Re: Job package: A lightweight multithreading tool, using promise/future mechanism
Posted by Oblivion on Sun, 10 Sep 2017 15:05:51 GMT
View Forum Message <> Reply to Message

Hello Mirek,

Thank you very much for your comments.

Quote:For now, it looks like Job is not using worker threads, creating thread for each run.

Yes, this is because Job is not a queue or pool. Maybe I have given you the wrong impression: It is not even meant to be an alternative or rival to CoWork which works just well. (I used a similar interface, because CoWork is already part of core U++ with a suitable interce design.)

Job is designed to be an interface to plain Thread, with a simplified error and result handling. (I admit the Job/Worker metaphor can be somewhat misleading.)

Quote:Which actually can have advantage if there are blocking operations (file I/O etc), but that is meant to be dealt with in CoWork by increasing the number of threads.

This is why I designed it for in the first place: For file and network operations. (That's why I supplied SocketClients example).

As for the CoWork's capabilities: Sure, CoWork can deal with such operations as well.

Quote:"WaitForJobs" is outright ugly, being single global function for all Jobs

I can turn it into Job::FinishAll(). However, Jobs, by default, wait for their workers to finish when they go out of scope.(Unless their workers are explicitly detached.)
And Finish & IsFinished will wait for each job to be done. (Though they can be improved.)

Consider this one:

```
{
 Job<String> filejob([=]{
  FileIn fi("/home/test_session/Very_Large_Test_File.txt");
  if(fi.IsError())
   JobError(fi.GetError(), fi.GetErrorText());
  Cout() << "Reading file...";
  return LoadStream(fi);
 });
 while(!filejob.IsFinished()) Cout() << "....";
 Cout() << (filejob.IsError() ? filejob.GetErrorDesc() : filejob.GetResult()) << '\n';
}
```

Now, of course this can be written in CoWork, but for such operations as above, I believe the interface of Job is simpler for individual thread operations, and would be less error prone. (No locks, no sharing, individual asynchronous operations, and their results...)

As a side note I run a simple test (I'm not sure if it's legitimate, but was curious.):

Below is the timing results for the same operations carried out by Job and CoWork (I see them as somewhat different, and complementary tools, but wanted to see how well they perform.)
When I reduce Sleep value in WaitForJosb() (I know it's ugly) to 1, I get this for 1500 computations (Under latest GCC):

```
TIMING Job          : 137.00 ms - 137.00 ms (137.00 ms / 1 ), min: 137.00 ms, max: 137.00 ms, nesting: 1 - 1
TIMING CoWork       : 206.00 ms - 206.00 ms (206.00 ms / 1 ), min: 206.00 ms, max: 206.00 ms, nesting: 1 - 1
```

Code is:

```
String GetDivisors()
{
 String s;
 int number = (int) 1000;
 Vector<int> divisors;
 for(auto i = 1, j = 0; i < number + 1; i++) {
  auto d = number % i;
  if(d == 0){
   divisors.Add(i);
   j++;
  }
  if(i == number)
   s = Format("Worker Id: %d, Number: %d, Divisors (count: %d): %s",
      GetWorkerId(),
      number,
      j,
      divisors.ToString());
 }
 return pick(s);
}

CONSOLE_APP_MAIN
{
 {
  CoWork jobs;
  jobs.SetPoolSize(1500);
  Vector<String> results;
  TIMING("CoWork");
  for(int i = 0; i < 1500; i++)
   jobs & [=, &results] { String h = GetDivisors(); CoWork::FinLock(); results.At(i) = h; };
  jobs.Finish();
  for(auto r : results)
   Cout() << r << '\n';


 }

 {
  Array<Job<String>> jobs;
  jobs.SetCount(1500);
  TIMING("Job");
  for(int i = 0; i < 1500; i++)
   jobs[i].Start([=]{ return GetDivisors(); });
  WaitForJobs();
```

```
  for(auto& job : jobs)
    Cout() << job.GetResult()    << '\n';

 }

}
```

Best regards,
Oblivion

Subject: Re: Job package: A lightweight multithreading tool, using promise/future mechanism
Posted by Oblivion on Sun, 10 Sep 2017 17:08:28 GMT
View Forum Message <> Reply to Message

And below is where CoWork starts to outperform Job (As you pointed out, because of the additional copying involved, I guess.):

I changed the code to see other options:

```
{
  CoWork jobs;
  Vector<String> results;
  TIMING("CoWork");
  jobs & [=, &results] {
   Vector<String> s;
   for(int i = 0; i < 50000; i++)
     s.Add() = GetDivisors();
    CoWork::FinLock();
    results = pick(s);
  };
  jobs.Finish();
 }

 {
  Job<Vector<String>> job;
  TIMING("Job");
  job.Start([=]{
   Vector<String> s;
   for(int i = 0; i < 50000; i++)
    s.Add() = GetDivisors();
   return pick(s);
  });
  job.Finish();
  auto s = job.GetResult();
```

```
}
```

```
TIMING Job          :  1.42 s  -  1.42 s  ( 1.42 s  / 1 ), min:  1.42 s , max:  1.42 s , nesting: 1 - 1
TIMING CoWork       :  1.39 s  -  1.39 s  ( 1.39 s  / 1 ), min:  1.39 s , max:  1.39 s , nesting: 1 - 1
```

Job is not an alternative to CoWork, but it's not a bad tool either. It does simplify writing high performance MT code in a convenient way, thanks to U++.
It is suitable for such asynchronous operations mainly where a high latency is expected (IO/sockets, etc.) and where the code needs to be easily managable (errors, and results should be easily and immediately dealt with.)

Best regards,
Oblivion

Subject: Re: Job package: A lightweight multithreading tool, using promise/future mechanism
Posted by mirek on Sun, 10 Sep 2017 18:09:50 GMT
View Forum Message <> Reply to Message

Oblivion wrote on Sun, 10 September 2017 17:05
```
  jobs.SetPoolSize(1500);
```

I think above is potentionally performance killer.

Subject: Re: Job package: A lightweight multithreading tool, using promise/future mechanism
Posted by mirek on Sun, 10 Sep 2017 18:13:29 GMT
View Forum Message <> Reply to Message

OK, so the difference is

- "return" semantics (implemented using promise/future)
- error states

Right?

If true, what about using future / promise directly? IMO the only problem is to have them

interfaced with Thread and/or CoWork (for different kinds of usage).

Subject: Re: Job package: A lightweight multithreading tool, using promise/future mechanism
Posted by Oblivion on Sun, 10 Sep 2017 19:16:13 GMT
View Forum Message <> Reply to Message

Hello Mirek, and thank you for your patience.

Quote:mirek
OK, so the difference is

- "return" semantics (implemented using promise/future)
- error states

Right?

If true, what about using future / promise directly? IMO the only problem is to have them interfaced with Thread and/or CoWork (for different kinds of usage).

Well, Job is one way of having them interfaced with U++ Thread. Using f/p pair directly is somewhat cumbersome.
However, TBH, what I really see lacking in multithreading tools in general is a simple error handling mechanism, and a per-thread shutdown mechanism.
This is the one of the reasons why I've come up with the Job class. It is also an attempt to solve these problems.

E.g.
Thread.IsError()
Thread.GetError()
Thread.GetErrorDesc()
Thread.ShutDown()
and
Thread::Error (exception)

As you can see in the Job.hpp these are not very hard to implement. (I don't see why it should be difficult for Thread at least (performance?). Cowork, being a job queue, is naturally a more complex beast.)

Best regards,
Oblivion

Subject: Re: Job package: A lightweight multithreading tool, using promise/future

mechanism
Posted by mirek on Sun, 10 Sep 2017 22:06:15 GMT
View Forum Message <> Reply to Message

Oblivion wrote on Sun, 10 September 2017 21:16and a per-thread shutdown mechanism.

That one is a borderline to impossible.

The problem are all those destructors that need to called. It is hard to abort thread that is does not know about it...

Mirek

Subject: Re: Job package: A lightweight multithreading tool, using promise/future mechanism
Posted by Oblivion on Sun, 10 Sep 2017 22:25:12 GMT
View Forum Message <> Reply to Message

Quote:

That one is a borderline to impossible.

The problem are all those destructors that need to called. It is hard to abort thread that is does not know about it...

Ah no, what I mean is setting a unique flag for each thread (maybe using ThreadId), so that it can be checked from within like IsShutdownThreads.
(I do this with Job:Cancel() method and IsJobCancelled() global function, by keeping an index of their IDs, which are integer numbers, incremented with each new thread, using a thread local variable as the unique id.)
I was just talking about signalling the shutdown to specific targets so that user can design his/her MT code and handle its shutdown conditions easily.

And this is exactly where Job comes handy. It is a higher level interface, a convenience wrapper, if yow will, with a simpler return semantics, error management, and a more refined shutdown mechanism. With a reasonably small overhead.

Best regards,
Oblivion

Subject: RE: Job package: A scope-bound worker thread for non-blocking

operations.
Posted by Oblivion on Sun, 17 Sep 2017 21:20:44 GMT
View Forum Message <> Reply to Message

Hello Mirek (and all U++ community),

After your review and criticism of Job class. I went back to design board and come up with a new version, mostly re-written.
I updated the description and the package in the first post, as usual, but allow me to copy/paste it's description here:

This package contains a lightweight and easy-to-use multithreading tool for U++ framework: Job.
Job template class implements a scope bound, single worker thread based on RAII principle.
It provides a return semantics for result gathering functionally similar to promise/future pattern but with three major differences:

1) future/promise pair requires at least moving of the resulted data, which can be relatively expensive depending on the object type. On the other hand, Job acts as a simple container and uses a reference based result gathering method. This makes it possible to reduce move/copy overhead involved (nearly down to zero).

2) Job does not allow the T to be of plain void type (of course, void pointer is allowed).

3) Trying to access the resulting data while it is still invalid will not throw. Resources are allocated during construction (including the job data).

Note that for higher performance loop parallelization scenarios, CoWork would be a more suitable option. This class is mainly designed to allow the applications and libraries to gain an easily managable, optional non-blocking behaviour where high latency is expected (Such as network operations and file I/O), and a safe "referential access" to the objects processed by the worker threads is preferred.

- It is now a proper single worker thread. (performance gain, and memory reduction is visible.) By design Job has no work scheduling (it is not meant to be a queue, not directly at least.)

- It is re-designed around the RAII principle: A scope-bound single worker thread that only gets destroyed when it is out-of-its scope.

- Most importantly, thanks to your criticisim, I ditched the future/promise mechanism completely, in favour of "Upp-native" way: Job instance are from now on basically simple data containers with referential acces to their data (result). Yet I've kept the alternative return semantics. It is really useful.

Granted, none of these are impossible to implement with CoWork or Thread. AFAIK CoWork is scope bound too. But Job's purpose is different. Although it can be used as a general parallelization tool, it is really meant to simplify writing non-blocking applications, or porting exisiting ones to them, providing a simple yet convenient interface.

For example, with this new design it took around 2 hours for me to port my own FTP class fully into MT environment, using a simple switch (Ftp::Blocking(false)) (News: Upcoming version (2.0) will support MT internally.).
Again, I've begun porting (an experiment for now) the SSH package to MT using Job, and it solves nearly every problem that I ran against wrapping SSH (non-blocking), and also both source code and interface is reduced drastically. It is very clean now. (all those Startxxx() and xxx() method pairs are gone, there are now only xxx ones. E.g. Ssh:Connect()) You can see this uniform programming/porting pattern emerging int the new SocketClients example I provided:

```
class Client : public Job<String> {
public:
 Client& Blocking(bool b = true) { blocking = b; return *this; }
 String Request(const String& host, int port);

private:
 String Run(Event<>&& cmd);
 bool blocking = true;
};

String Client::Run(Event<>&& cmd)
{
 Start(pick(cmd));
 if(blocking) Finish();
 return blocking && !IsError() ? GetResult() : GetErrorDesc();
}

String Client::Request(const String& host, int port)
{
 auto cmd = [=]{
  TcpSocket socket;
  auto& output = Job<String>::Data(); // This method allows referential access to the data of
respective job.
  output = Format("Client #%d: ", GetWorkerId());

  INTERLOCKED { Cout() << output << "Starting...\n"; }

  if(socket.Timeout(10000).Connect(host, port))
   output.Cat(socket.GetLine());
  if(socket.IsError())
   throw JobError(socket.GetError(), socket.GetErrorDesc());
 };
 return Run(cmd);
}

CONSOLE_APP_MAIN
{
```

```
	//.....

	// Requesting in a simple, blocking way.
{
 Cout() << "----- Processing individual blocking requests...\n";
 Cout() << c1.Request(host1, 21) << '\n';
 Cout() << c2.Request(host2, 21) << '\n';
}

// Reuse workers and make requests in a simple, non-blocking way.
{
 Cout() << "----- Processing individual non-blocking requests...\n";
 // We can "clear" the data (String):
 c1.GetResult().Clear();
 c2.GetResult().Clear();

 c1.Blocking(false).Request(host1, 21);
 c2.Blocking(false).Request(host2, 21);

 while(!c1.IsFinished() || !c2.IsFinished())
  ;
 if(c1.IsError()) Cerr() << c1.GetErrorDesc() << '\n';
 else Cout() << ~c1 << '\n';

 if(c2.IsError()) Cerr() << c2.GetErrorDesc() << '\n';
 else Cout() << ~c2 << '\n';

}

	//....

}
```

Please also take a look into the full code.

As always, review, bug reports, criticism, feedback are greatly appreciated.

Best regards,
Oblivion

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.

Posted by mirek on Mon, 18 Sep 2017 06:23:02 GMT
View Forum Message <> Reply to Message

Oblivion wrote on Sun, 17 September 2017 23:20

- Most importantly, thanks to your criticisim, I ditched the future/promise mechanism completely, in favour of "Upp-native" way: Job instance are from now on basically simple data containers with referential acces to their data (result). Yet I've kept the alternative return semantics. It is really useful.

That's intereseting, because I actually suggested adding future/promise to U++ facilities...

Anyway, I might check it, but I do not see any archive / link posted.

Mirek

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by Oblivion on Mon, 18 Sep 2017 06:29:46 GMT
View Forum Message <> Reply to Message

Quote:That's intereseting, because I actually suggested adding future/promise to U++ facilities... Smile

Anyway, I might check it, but I do not see any archive / link posted.

Mirek

That can also be done, I can work on it separately, or re-introduce it as an option if you think it is worth it. (In it's current state Job package does not cover a shared_future alternative. Maybe I should consider adding a SharedJob class, but 1) there is already CoWork for paralellization, 2) I don't quite know a real life scenario where it would be more useful than CoWork.)

Nevertheless, recently I've found my "reference based" approach much more performant and convenient than future/promise.

Consider this (pseudo) code:

```
double sum_total = 0;
Job<Vector<double>> job;
job.Start(DoSomeHeavyCalculationAndFillTheVector);
```

```
    //....

    if(job.IsFinishe() && !job.IsError()) {
       for(auto& e : ~job)         // Reference access to Vector. Data is not copied or moved. Vector is filled via a direct -yet safe- access.
             sum_total += e;       // And retrieved in same fashion as well. Note also that there is no requirement for the data type to be contained.
                                   // In fact, using One or Any, or Value, we can even differentiate between the job
                                   // results for given operations in a single Job easily. :)
    }
```

Package's link is on the bottom of the first post of this topic.

Best regards,
Oblivion

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by Oblivion on Mon, 18 Sep 2017 22:01:55 GMT
View Forum Message <> Reply to Message

Here is the code:

File Attachments
1) Job Package and Example (ClientSockets).zip, downloaded 434 times

---

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by Oblivion on Tue, 19 Sep 2017 06:12:31 GMT
View Forum Message <> Reply to Message

Hello Mirek,

(Thank you for your comments. I deeply appreciate it.)

I was curious about if the new design of Job class will pay off, whether it is also a resonable general paralellization tool, and did some benchmarking with the Divisors example.

I assumed Job and CoWork to be functionally and effectively identical in this example (In the sense that, regardless of their internals, they are both doing the same thing: Calculating divisiors for the number 1000, 10.000 times in available worker threads, then printing the results to the screen.)

I simply changed the jobs loop to take advantage of new return semantics (I don't know if CoWork can be put into a similar loop, so I am taking this with a grain of salt):
The loop for the job is simply a very crude slot manager for 8 Job workers. (Tested on AMD FX 6100, six core processor.)

```
 Array<Job<String>> jobs;
 jobs.SetCount(CPU_Cores() + 2);

 CoWork cowork;
// cowork.SetPoolSize(CPU_Cores() + 2);

 Vector<String> results;
 DUMP(CPU_Cores());
 {

  TIMING("CoWork -- With stdout output");
  for(int i = 0; i < 10000; i++)
   cowork & [=, &results] { String h = GetDivisors(); CoWork::FinLock(); results.At(i) = h; };
  cowork.Finish();
  // Stdout output section.
  for(auto& r : results)
   Cout() << r << '\n';

 }
 {
  TIMING("Job -- With stdout output");

  int i = 0;
  while(i < 10000) {
   for(auto& job : jobs) {
    if(!job.IsFinished()) {
     continue;
    }
    job & [=]{ Job<String>::Data() = GetDivisors(); };
    if(!(~job).IsEmpty()) {
     Cout() << ~job << '\n';
     if(++i == 10000) break;
    }
   }

  }
```

}

Results (consistent):

For 10000 computiation.

CPU_Cores() = 6
TIMING Job -- With stdout output: 370.00 ms - 370.00 ms (370.00 ms / 1), min: 370.00 ms, max: 370.00 ms, nesting: 1 - 1
TIMING CoWork -- With stdout output: 461.00 ms - 461.00 ms (461.00 ms / 1), min: 461.00 ms, max: 461.00 ms, nesting: 1 - 1

CPU_Cores() = 6
TIMING Job -- Without stdout output: 228.00 ms - 228.00 ms (228.00 ms / 1), min: 228.00 ms, max: 228.00 ms, nesting: 1 - 1
TIMING CoWork -- Without stdout output: 234.00 ms - 234.00 ms (234.00 ms / 1), min: 234.00 ms, max: 234.00 ms, nesting: 1 - 1

for 1000 compuitation.

CPU_Cores() = 6
TIMING Job -- With stdout output: 34.00 ms - 34.00 ms (34.00 ms / 1), min: 34.00 ms, max: 34.00 ms, nesting: 1 - 1
TIMING CoWork -- With stdout output: 53.00 ms - 53.00 ms (53.00 ms / 1), min: 53.00 ms, max: 53.00 ms, nesting: 1 - 1

CPU_Cores() = 6
TIMING Job -- Without stdout output: 24.00 ms - 24.00 ms (24.00 ms / 1), min: 24.00 ms, max: 24.00 ms, nesting: 1 - 1
TIMING CoWork -- Without stdout output: 31.00 ms - 31.00 ms (31.00 ms / 1), min: 31.00 ms, max: 31.00 ms, nesting: 1 - 1

What do you think?

Best regards,
Oblivion

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.

Posted by mirek on Tue, 19 Sep 2017 06:37:02 GMT
View Forum Message <> Reply to Message

This is ugly:

```
 // Reuse workers and make requests in a simple, non-blocking way.
 {
  Cout() << "----- Processing individual non-blocking requests...\n";
  // We can "clear" the data. Useful especially if the data is "picked".
  c1.Clear();
  c2.Clear();

  c1.Blocking(false).Request(host1, 21);
  c2.Blocking(false).Request(host2, 21);

  while(!c1.IsFinished() || !c2.IsFinished())
   ;
  if(c1.IsError()) Cerr() << c1.GetErrorDesc() << '\n';
  else Cout() << ~c1 << '\n';

  if(c2.IsError()) Cerr() << c2.GetErrorDesc() << '\n';
  else Cout() << ~c2 << '\n';

 }
```

I think this should work without IsFinished loop.

Anyway, all thing still feels very much like future, with implicit promise connected to Thread. In that regard, full future/promise still sounds more powerful.

Mirek

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by Oblivion on Tue, 19 Sep 2017 06:53:19 GMT
View Forum Message <> Reply to Message

Yes that does work withoud IsFinished loop.
I put that simply to show that it is possible to do someting else in that while() loop (some checks, calculations, whatever is needed.) while the requests are being processed.
It gives per-job control over the async works.

Example:

```
while(!c1.IsFinished() || !c2.IsFinished()) {
```

```
if(c2.IsFinished())
    Cout() << second request is complete before the first\n";
   // Etc...

}
```

Best regards,

Oblivion

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by mirek on Tue, 19 Sep 2017 08:12:17 GMT
View Forum Message <> Reply to Message

Small issues with the code:

sData template does not compile with MSC (I believe it is not legit C++ code, probably GCC eats it but it is not OK). Replaced by

```
template<class T>
One<T>*& sData()
{
 static thread_local One<T>* sData = NULL;
 return sData;
}
```

GetNewWorkerId has int64 internal counter (which IMO is reasonable), but the rest of the code is using int.

Anyway, as you are using "TIMING", it appears you are benchmarking in debug mode. I have done some changes:

```
#include <Core/Core.h>
#include <Job/Job.h>

using namespace Upp;

String GetDivisors()
{
 String s;
 int number = (int) 1000;
```

```
 Vector<int> divisors;
 for(auto i = 1, j = 0; i < number + 1; i++) {
  auto d = number % i;
  if(d == 0){
   divisors.Add(i);
   j++;
  }
  if(i == number)
   s = Format("Worker Id: %d, Number: %d, Divisors (count: %d): %s",
      GetWorkerId(),
      number,
      j,
      divisors.ToString());
 }
 return pick(s);
}

// #define OUTPUT
#define N 100000

CONSOLE_APP_MAIN
{
 Array<Job<String>> jobs;
 jobs.SetCount(CPU_Cores() + 2);

 CoWork cowork;
// cowork.SetPoolSize(CPU_Cores() + 2);

 Vector<String> results;
 RDUMP(CPU_Cores());
 {

  RTIMING("CoWork -- With stdout output");
  for(int i = 0; i < N; i++)
   cowork & [=, &results] { String h = GetDivisors(); CoWork::FinLock(); results.At(i) = h; };
  cowork.Finish();
  // Stdout output section.
 #ifdef OUTPUT
  for(auto& r : results)
   Cout() << r << '\n';
 #endif
 }
 {
  RTIMING("Job -- With stdout output");

  int i = 0;
  while(i < N) {
   for(auto& job : jobs) {
```

```
    if(!job.IsFinished()) {
     continue;
    }
    job & [=]{ Job<String>::Data() = GetDivisors(); };
    if(!(~job).IsEmpty()) {
    #ifdef OUTPUT
     Cout() << ~job << '\n';
    #endif
     if(++i == N) break;
    }
   }

  }
 }
}
```

with results in _RELEASE_ mode:

* d:\upp.out\MyApps\MSC15\JobBenchmark.exe 19.09.2017 10:01:27, user: cxl

CPU_Cores() = 8
TIMING Job -- With stdout output: 123.00 ms - 123.00 ms (123.00 ms / 1), min: 123.00 ms, max: 123.00 ms, nesting: 1 - 1
TIMING CoWork -- With stdout output: 103.00 ms - 103.00 ms (103.00 ms / 1), min: 103.00 ms, max: 103.00 ms, nesting: 1 - 1

Now I fully understand that performance is not the reason for Job, but as I have seen no technical reasons why CoWork should be that much slower, I had to check....

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by mirek on Tue, 19 Sep 2017 08:15:33 GMT
View Forum Message <> Reply to Message

Oblivion wrote on Sun, 17 September 2017 23:20
[code]
1) future/promise pair requires at least moving of the resulted data, which can be relatively expensive depending on the object type. On the other hand, Job acts as a simple container and uses a reference based result gathering method. This makes it possible to reduce move/copy overhead involved (nearly down to zero).

This is not quite true (unlike Job, which in fact requires you to move the data if you need it outside the Job). E.g.:

```
#include <Core/Core.h>
#include <future>

using namespace Upp;

template <class D, class P, class F, class... Args>
void set_future_value(D *dummy, P&& p, F&& f, Args&&... args)
{
 p.set_value(f(args...));
}

template <class P, class F, class... Args>
void set_future_value(void *dummy, P&& p, F&& f, Args&&... args)
{
 f(args...);
 p.set_value();
}

template< class Function, class... Args>
std::future<std::result_of_t<std::decay_t<Function>(std::decay_t<Args>...)>>
Async(Function&& f, Args&&... args )
{
 std::promise<std::result_of_t<std::decay_t<Function>(std::decay_t<Args>...)>> p;
 auto ftr = p.get_future();
 CoWork::Schedule([=, p = pick(p)]() mutable {
  std::result_of_t<std::decay_t<Function>(std::decay_t<Args>...)> *dummy = NULL;
  set_future_value(dummy, pick(p), pick(f));
 });
 return ftr;
}

CONSOLE_APP_MAIN
{
// with reference
 String h;
 auto f = Async([&h] { h = "Hello world"; });
 f.get();
 DDUMP(h);

// using return value
 DDUMP(Async([] { return "Hello world"; }).get());
}
```

BTW, I am mostly arguing there because it is a good way for me to investigate these issues....

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by mirek on Tue, 19 Sep 2017 08:24:30 GMT
View Forum Message <> Reply to Message

future / promise also seems to have superior error handling facilities (exceptions can easily be passed from thread to caller).

The one thing that seems to be missing is future -> promise abort.

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by Oblivion on Tue, 19 Sep 2017 09:12:03 GMT
View Forum Message <> Reply to Message

Hello Mirek,
Thanks for all the corrections. I find your opinions really valuable and useful. I was suspicios there was something wrong with that benchmark too.

Quote:This is not quite true (unlike Job, which in fact requires you to move the data if you need it outside the Job).

Not necessarily. I can always use a pointer or Ref(), or std::ref. (In it's current state it wouldn't be as clean as capturing, I admit, but can be improved)
Or I can simply pass reference, using lambda capture. That is not forbidden. But then I have to take into account the object's life time too.
Job solves some of those head aches. (Of course, tt may introduce its own).

For the sake of discussion (ugly):
In Job.h

```
template<class V = T>
Job(V v) : Job()       { *data = v; }
```

then, in Example.cpp:

```
int i;

Job<int*> job(&i);
job.Start([]{ auto& n = *Job<int*>::Data() = 100;});
job.Finish();
Cout() << "Number is " << i << '\n';
```

Quote:future / promise also seems to have superior error handling facilities (exceptions can easily be passed from thread to caller).

Not really hard to add. I can: process JobErrors, and re-throw plain Upp::Exc, or std::exception category.

Quote:
The one thing that seems to be missing is future -> promise abort.

Job has a per-job cancel mechanism.

So what do you suggest?

Your Async wrapper is really simple and nice. Are you going to add it to U++?

Or should I improve Job? (add exception forwarding, easier way to pass references and pointers...)

My favourite : Or should I re-introduce Future/promise pair as was before? After all Job was inially a future/promise wrapper. But it lacked RAII and worker thead model.
Now they are in place. All I need to do is to incorporate it to the current model using the Async example you provided (except cowork).
(Without changing the public api of Job much. IMO It looks familiar and acts fine.)

Or should I simply abandon it? This tool is born out of need, which plain future/promise mechanism couldn't satisfy (i.e. a fine-grained control over threads for non-blocking behaviour.)

I don't see a reson to hang onto it if there will be a better solution.

Again, thank you very much for taking time to discuss the issue.

Job currently lacks shared states but it is not what Job is meant for anyway. IMO CoWork is there for it (and for bunch of other things).

Best regards,
Oblivion

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by Oblivion on Tue, 19 Sep 2017 10:31:23 GMT
View Forum Message <> Reply to Message

How about adding An Async/Result pair, the way you suggested.

ASync will be an improved version of the example you provided with (using CoWork), addition of result id generator (for cancellation), exception handling:
Result will be a wrapper for future where we can do such things as follows:

```
auto result1 = Async([=]{ return GetDivisors(); })
while(!result.IsReady())
	; // Do something;
Cout() << result.IsValid() ? result.Get() : result.GetErrorDesc();

// and of course...
Cout() << Async([=]{ return "Hello World\n"; }).Get();
```

Result's Interface may be consisted of:

Result::Get();

Result::IsReady();

Result::IsValid();

Result::GetError();

Result::GetErrorDesc():

Result::GetId();

Result::Cancel();

If you'd like, I can implement it this way. (IMO Result is a more meaningful name, but naming can change...)

Best regards,
Oblivion

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking

operations.
Posted by mirek on Tue, 19 Sep 2017 11:01:00 GMT
View Forum Message <> Reply to Message

Oblivion wrote on Tue, 19 September 2017 12:31How about adding An Async/Result pair, the way you suggested.

ASync will be an improved version of the example you provided with (using CoWork), addition of result id generator (for cancellation), exception handling:
Result will be a wrapper for future where we can do such things as follows:

```
auto result1 = Async([=]{ return GetDivisors(); })
while(!result.IsReady())
	; // Do something;
Cout() << result.IsValid() ? result.Get() : result.GetErrorDesc();

// and of course...
Cout() << Async([=]{ return "Hello World\n"; }).Get();
```

Result's Interface may be consisted of:

Result::Get();

Result::IsReady();

Result::IsValid();

Result::GetError();

Result::GetErrorDesc():

Result::GetId();

Result::Cancel();

If you'd like, I can implement it this way. (IMO Result is a more meaningful name, but naming can change...)

Best regards,

Oblivion

Not sure what is best...

Mirek

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by Oblivion on Sat, 23 Sep 2017 12:17:54 GMT
View Forum Message <> Reply to Message

Hello Mirek (and Upp community),

Job package is updated in accordance with the above discussion.

To sum up, it is heavily improved (tested on GCC 7.2.0, MinGW (supplied with U++), and latest MinGWx64, and MSC 2017):

- Void type instantiation is re-introduced for good (does not rely on anything other than plain template specializaton rules.)

- Value return semantics is re-introduced for good.

- Constant reference access operator is added. (This is especially useful when using Job with containers such as Vector, or array (e.g. Job<Vector<int>>) in a loop. See JobExample test app provided)

- Exception propagation mechanism is added. Workers will propagate raised exceptions to their caller. (when one of the GeResult(), operator~(), or IsError() methods is called.)

- JobExample test application is added to the package. This example demonstrates the above mentioned traits.

I believe with the recent update Job is now slightly superior to async/future/promise trio (not shared_future), in that it has full control over its thread (which is a proper worker thread) worker cancelleation/abortion mechanics,
and optional container-like reference access to its result which can reduce copying/moving where unnecessary. (Except Job doesn'T support reference specialization. Job<T&> is not possible for now. But that is not a big deal, really)

Bug reports, criticism and feedback is deeply appreciated.

[See below message for new version]

Best regards.
Oblivion

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by Oblivion on Sat, 07 Oct 2017 13:03:44 GMT
View Forum Message <> Reply to Message

Hello,

Job package is now compatible with single-threaded U++ environment. (Yet -almost- all methods and global functions retain their functionality under ST env.)
Documentation and provided example is updated accordingly.

The code below is a part of JobExample.cpp and demonstrates both non-blocking behaviour and Job cancelling feature in a single-threaded environment:

```
void CancelJob()
{
 // Singlethreaded version. (non-blocking operation)
 // Below example is one of the simplest ways to achive non-blocking operations with Job in a
 // single-threaded environment. A finer-grained operation would involve handling of return
 // codes. (e.g. using Job<int>)

 Job<void> job;
 auto work = [=] {
  if(IsJobCancelled()) {
   Cout() << "Worker #" << GetWorkerId() << " received cancel signal. Cancelling job...\n";
   throw JobError("Operation cancelled.");
  }
 };

 Cout() << "Worker #" << GetWorkerId() << " started. (Waiting the cancellation signal...)\n";
 const int timeout = 5000;
 auto start_time = msecs();
 while(!job.IsError()) {
  job.Start(work);
  if(msecs(start_time) >= timeout)
   job.Cancel(); // Or if you have more than one non-blocking operation in progress,
                 // you can call CancelJobs() global function to cancel all running jobs.
 }
```

}

As always, reviews, criticism, bug reports, etc. are deeply appreciated.

Best regards,
Oblivion

File Attachments

1) Job Package and Examples.zip, downloaded 452 times

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by mirek on Sat, 07 Oct 2017 13:51:44 GMT
View Forum Message <> Reply to Message

Oblivion wrote on Sat, 07 October 2017 15:03Hello,

Job package is now compatible with single-threaded U++ environment. (Yet -almost- all methods and global functions retain their functionality under ST env.)

ST is now deprecated...

BTW, have you noticed that, inspired by this discussion thread, CoWork now has cancelation and exception propagation?

IMO, that makes it very similar to job, the only part that is really missing is the return value.

Actually, the main reason I have not implemented that yet is that I do not like the explicit specification of return value, as in Job. Which requires to implement it as function, which in turn requires some form of 'future'.

Mirek

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by Oblivion on Sat, 07 Oct 2017 14:04:50 GMT
View Forum Message <> Reply to Message

Hello Mirek,

Quote:ST is now deprecated...

Sure, The reason I added ST support is that I have some code that needs to be backward (ST) compatible. And for those who want't to keep their code as such. (It wasn't very difficult to add, anyway).

Quote:
BTW, have you noticed that, inspired by this discussion thread, CoWork now has cancelation and exception propagation?

IMO, that makes it very similar to job, the only part that is really missing is the return value.

Actually, the main reason I have not implemented that yet is that I do not like the explicit specification of return value, as in Job. Which requires to implement it as function, which in turn requires some form of 'future'.

Mirek

No, I didn't notice it till now, that's great, one less reason for me to use Job, thanks!
I'll try and see if I can use CoWork instead. (It shouldn't be very difficult)
Job's return semantics proved really useful, so for the time being I intend to maintain it.

And in the meantime let's see if I can come up with a suitable alternative to std::future (to use it with CoWork).

Best regards,
Oblivion.

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by Oblivion on Sun, 08 Oct 2017 13:01:15 GMT
View Forum Message <> Reply to Message

Hello Mirek,

I wrote a prototype for U++ implementation of promise/future mechanism. It is 104 LOCs, and it took an hour for me to to design and write it (It is in its current state not perfect but it works.)

It is consisted of three classes (please do not dwell on naming, I didn't think them through, they'll change):

- WorkEntry -> std:promise (This class is not explicitly called. It is allocated on the heap (using new), and its ownership is passed to WorkResult.)

- WorkResult -> std: future (With pick semantics, void type specialization, error handling, exception propagation, and optional constant reference access to the result.)
- Work (Contains a CoWork instance, and exposes Do() and Finish() methods.)

I re-implemented your Async() code in Work class, using CoWork::Do() instead of CoWork::Schedule(). The rationale behind this decision is that sometimes asynchronous operations need to be externally blocked, all at once ("wait for all works to finish"). And I found no other reliable way. Also, works use a single semaphore to wait and run.

Example code:

```
// Error handling, picking/moving.
 auto r1 = work.Do([=] { throw Work::Error("Worker failed."); });
 work.Finish();
 auto r2 = pick(r1); // r1 is picked.
 if(r2.IsError()) {
  Cout() << r2.GetErrorDesc() << '\n';
 }
```

There is room for improvements, and probably some issues I didn't encounter yet or I overlooked. Feel free to comment on it. (One area that needs improving and to be made more error resistant is moving/picking behaviour. I'll improve them)

Below you can find the Work, and WorkExample, which is basically JobBencmark adapted to Work.

Best regards,
Oblivion

File Attachments

1) Work.zip, downloaded 445 times

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by Oblivion on Tue, 10 Oct 2017 09:51:00 GMT
View Forum Message <> Reply to Message

A further refinement would be to explicity relate CoWork to promise/future pattern:

My suggested namings:

-

- WorkResult (former WorkEntry): Unlike std::promise, this is called privately. I don't really see any use in calling it explicitly
- Work (former WorkResult): This can be the U++ std::future counterpart (see the first prototype packageI provided above) This can be a helper class to CoWork, representing a single, isolated (semantically) thread.

- CoWork::Work(): This is the thread-starter method which will return, well, Work :)

I propose adding CoWork::Work() as a (non-static) method to CoWork, because from my experience with Job,Thread, future/promise and CoWork, and as I noted on my previous message, keeping workers contained in CoWork instances, using a CoWork::Do() call, have some advantages over using a static method such as CoWork::Schedule(): Such as the ability to use CoWork::Finish() on demand, and waiting the workers to be finished automatically on class instance destruction.

As for the error management mechanism using a specific exception type such as Work::Error, along with IsError(), GetError() and GetErrorDesc() methods, IMO they would be a very useful addition, but they are not necessary, can be removed.

What do you think?

Best regards,
Oblivion

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by mirek on Tue, 10 Oct 2017 14:10:09 GMT
View Forum Message <> Reply to Message

My try.

```
#include <Core/Core.h>

using namespace Upp;

template <class Ret>
class AsyncWork {
 template <class Ret>
 struct Imp {
  CoWork co;
  Ret    ret;
```

```
 template< class Function, class... Args>
 void       Do(Function&& f, Args&&... args) { co.Do([=]() { ret = f(args...); }); }
 const Ret& Get()                            { return ret; }
};

template <>
struct Imp<void> {
 CoWork co;

 template< class Function, class... Args>
 void       Do(Function&& f, Args&&... args) { co.Do([=]() { f(args...); }); }
 void       Get()                            {}
};

One<Imp<Ret>> imp;

public:
 template< class Function, class... Args>
 void  Do(Function&& f, Args&&... args)    { imp.Create().Do(f, args...); }

 void  Cancel()                          { if(imp) imp->co.Cancel(); }
 Ret   Get()                           { ASSERT(imp); imp->co.Finish(); return imp->Get(); }
 Ret   operator~()                       { return Get(); }

 AsyncWork(AsyncWork&&) = default;
 AsyncWork()                             {}
 ~AsyncWork()                             { if(imp) imp->co.Cancel(); }
};

template< class Function, class... Args>
AsyncWork<std::result_of_t<std::decay_t<Function>(std::decay_t<Args>...)>>
Async(Function&& f, Args&&... args)
{
 AsyncWork<std::result_of_t<std::decay_t<Function>(std::decay_t<Args>...)>> h;
 h.Do(f, args...);
 return pick(h);
}

CONSOLE_APP_MAIN
{
 auto h = Async([] { return "Hello world"; });
 DDUMP(h.Get());

 DDUMP(~Async([](int x) { return x * x; }, 9));

 auto x = Async([] { throw "error"; });
 try {
  x.Get();
```

```
 }
 catch(...) {
  DLOG("Exception caught");
 }
}
```

I guess it is quite similar now.

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by Oblivion on Tue, 10 Oct 2017 15:02:53 GMT
View Forum Message <> Reply to Message

Hello Mirek,

I haven't tested it yet, but as usual, yours is much more clean, and concise.

IMO the only missing part here is an IsFinished() method. A non-blocking wait is in some cases a must. std::future provides this with future::wait_for(),
As far as I can see it usually translates to IsFinished() in U++ terminology

```
bool IsFinished() { ASSERT(imp); return imp->co.IsFinished(); }
```

Other than that, JMO AsyncWork and Async would be a worthy addition to U++.

Best regards,

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by mirek on Tue, 10 Oct 2017 18:51:14 GMT
View Forum Message <> Reply to Message

There is a tricky catch with IsFinished:

```
template <class Range>
ValueTypeOf<Range> ASum(const Range& rng, const ValueTypeOf<Range>& zero)
{
 int n = rng.GetCount();
 if(n == 1)
```

```
  return rng[0];
 if(n == 0)
  return 0;
 auto l = Async([&] { return ASum(SubRange(rng, 0, n / 2)); });
 auto r = Async([&] { return ASum(SubRange(rng, n / 2, n - n / 2)); });
 while(!l.IsFinished() || !r.IsFinished())
  Sleep(1);
 return l.Get() + r.Get();
}
```

What do you think is wrong with this code?

Mirek

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by Oblivion on Tue, 10 Oct 2017 21:48:05 GMT
View Forum Message <> Reply to Message

Hello Mirek,

tested Async/AsyncWork with MSC 2017, MinGW on windows, and with GCC on linux.

- It compiles on MSC without any hiccup.
- It does not compile on GCC (7.2) or MinGW unless the nested classes are moved out, (That's why I wrote my prototype that way.) and "Ret" is changed to some other parameter name. Here are the error codes I get:

```
\AsyncTest.cpp (8): error: declaration of template parameter 'Ret' shadows template parameter
\AsyncTest.cpp (18): error: explicit specialization in non-namespace scope 'class
AsyncWork<Ret>'
\AsyncTest.cpp (19): error: template parameters not deducible in partial specialization:
\AsyncTest.cpp (31): error: too many template-parameter-lists
\AsyncTest.cpp (47): error: 'class AsyncWork<const char*>' has no member named 'Do' ():
 h. D o (f, args...);
\AsyncTest.cpp: In instantiation of 'AsyncWork<typename std::result_of<typename
std::decay<_Tp>::type(std::decay_t<Args>...)>::type> Async(Function&&, Args&& ...) [wit
h Function = ConsoleMainFn_()::<lambda(int)>; Args = {int}; typename std::result_of<typename
std::decay<_Tp>::type(std::decay_t<Args>...)>::type = int]':
\AsyncTest.cpp (56): required from here
\AsyncTest.cpp (47): error: 'class AsyncWork<int>' has no member named 'Do'
\AsyncTest.cpp (47): error: 'class AsyncWork<void>' has no member named 'Do'
\AsyncTest.cpp (11): error: 'AsyncWork<Ret>::Imp<Ret>::ret' has incomplete type
\AsyncTest.cpp (11): error: invalid use of 'void'
\AsyncTest.cpp (15): error: forming reference to void
```

\AsyncTest.cpp (34): error: 'struct AsyncWork<void>::Imp<void>' has no member named 'Get'; did you mean 'ret'?
\AsyncTest.cpp (34): error: return-statement with a value, in function returning 'void' [-fpermissive]

- More importantly there seems to be something wrong with the exception propagation mechanism. For,

  1) Sometimes it fails to catch the exception, and the application crashes with that exception.
  2) When it catches the exception the application hangs at the end (after the "exception caught" message is printed.)
  3) Sometimes the application simply hangs.

I got this erratic behaviour both on windows and on linux, on a single machine, so it maybe a local hardware problem, I need to investigate it further...

Quote:s a tricky catch with IsFinished:

```
template <class Range>
ValueTypeOf<Range> ASum(const Range& rng, const ValueTypeOf<Range>& zero)
{
 int n = rng.GetCount();
 if(n == 1)
  return rng[0];
 if(n == 0)
  return 0;
 auto l = Async([&] { return ASum(SubRange(rng, 0, n / 2)); });
 auto r = Async([&] { return ASum(SubRange(rng, n / 2, n - n / 2)); });
 while(!l.IsFinished() || !r.IsFinished())
  Sleep(1);
 return l.Get() + r.Get();
}
```

What do you think is wrong with this code? Smile

Mirek

Sure, but can this really be attributed to a design flaw?
I mean, f I'm not really missing anything else, it seems that here we simply have a careless programming.
Recursion is potentially tricky by nature, and requires the developer to be extra cautious with his/her assumptions.

A proper use of IsFinished() can be found in my JobBenchmark example , where it is simply used

to check the worker, and move on to others if the job is not finished... (at least, that's what I have in my mind in the first place)

Best regards.
Oblivion

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by mirek on Wed, 11 Oct 2017 07:23:30 GMT
View Forum Message <> Reply to Message

Oblivion wrote on Tue, 10 October 2017 23:48Hello Mirek,

tested Async/AsyncWork with MSC 2017, MinGW on windows, and with GCC on linux.

- It compiles on MSC without any hiccup.
- It does not compile on GCC (7.2) or MinGW unless the nested classes are moved out, (That's why I wrote my prototype that way.) and "Ret" is changed to some other parameter name. Here are the error codes I get:

After a bit of wrestling with C++11, it now compiles.

Quote:
- More importantly there seems to be something wrong with the exception propagation mechanism. For,

1) Sometimes it fails to catch the exception, and the application crashes with that exception.
2) When it catches the exception the application hangs at the end (after the "exception caught" message is printed.)
3) Sometimes the application simply hangs.

I got this erratic behaviour both on windows and on linux, on a single machine, so it maybe a local hardware problem, I need to investigate it further...

I would like to investigate, but need a testcase.

Quote:
Sure, but can this really be attributed to a design flaw?
Recursion is potentially tricky by nature, and requires the developer to be extra cautious with his/her assumptions.

Well, recursion aside, I think that in general, we want the mechanism reentrant. I mean, it should not be a part of contract/function documentation whether is it using IsFinished internally.

I can actually fix the issue, but then IsFinished will not be doing the same thing.

BTW, the very same issue is true for CoWork. So something to think about..

Mirek

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by Oblivion on Wed, 11 Oct 2017 07:42:56 GMT
View Forum Message <> Reply to Message

Quote:
Well, recursion aside, I think that in general, we want the mechanism reentrant. I mean, it should not be a part of contract/function documentation whether is it using IsFinished internally.

Ah, I see. Apparently I'm still not thinking out of my Job-mindset (It was theoretically reentrant, although I did not actually test it.).
CoWork is a different beast internally. I forgot that.

As for the erratic exception handling, I didn't use any code other than you provided above. Simply put, it sometimes works and sometimes doesn't. I didn't notice any regular pattern.

Best regards,
Oblivion

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by mirek on Wed, 11 Oct 2017 07:47:52 GMT
View Forum Message <> Reply to Message

Oblivion wrote on Wed, 11 October 2017 09:42Quote:
Well, recursion aside, I think that in general, we want the mechanism reentrant. I mean, it should not be a part of contract/function documentation whether is it using IsFinished internally.

As for the erratic exception handling, I didn't use any code other than you provided above. Simply put, it sometimes works and sometimes doesn't. I didn't notice any regular pattern.

Now I am confused. There is no 'throw' in my ASum example...

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by Oblivion on Wed, 11 Oct 2017 07:50:53 GMT
View Forum Message <> Reply to Message

Quote:
Now I am confused. There is no 'throw' in my ASum example...

No, not that one. The basic test case above, withthe AsyncWork code above, where you throw "error":

```
CONSOLE_APP_MAIN
{
 auto h = Async([] { return "Hello world"; });
 DDUMP(h.Get());

 DDUMP(~Async([](int x) { return x * x; }, 9));

 auto x = Async([] { throw "error"; });
 try {
  x.Get();
 }
 catch(...) {
  DLOG("Exception caught");
 }
}
```

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by mirek on Wed, 11 Oct 2017 17:30:58 GMT
View Forum Message <> Reply to Message

Weird. The only reason for erratic behaviour would be some form of race condition, so I have tried this:

```
#include <Core/Core.h>

using namespace Upp;

Atomic h;

void Wait(int c)
{
```

```
 while(c--)
  h++;
}

CONSOLE_APP_MAIN
{
 StdLogSetup(LOG_COUT|LOG_FILE);

 for(int i = 0; i < 100000; i++) {
  if(i % 1000 == 0)
   LOG(i);
  bool b = false;
  auto x = Async([] { Wait(Random(3000)); throw "error"; });
  try {
   Wait(Random(3000));
   x.Get();
  }
  catch(...) {
   b = true;
  }
  ASSERT(b);
 }

 LOG("============ OK");
}
```

... and it passed without a problem. Added that test to autotest just to be sure...

Anyway, (maybe I forgot to mention to AsyncWork is now in U++): Are you trying with current trunk?

Subject: Re: RE: Job package: A scope-bound worker thread for non-blocking operations.
Posted by Oblivion on Wed, 11 Oct 2017 18:14:24 GMT
View Forum Message <> Reply to Message

Hello Mirek,

Quote:Anyway, (maybe I forgot to mention to AsyncWork is now in U++): Are you trying with current trunk?

This is great news! Thank you very much for your patience and efforts. This is a really handy tool which obsoletes Job.

As for the weird error I get, it seems to be stopped after I purged UPPOUT.

IF I encounter it again, I'll report it.

By the way, I noticed that there are missing topic titles in Core/src.tpp : (Huge, Function, Unicode Support). Also, Huge class title is missing in it's doc.

Best regards.
Oblivion
