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