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