GSoC 2013 Boost Async file i/o intern test

Niall Douglas http://www.nedprod.com <nialldouglas14@gmail.com>

Dear Interested Interns,

Firstly thank you for submitting yourselves as GSoC candidates for converting my async file i/o library based on Boost.ASIO into a standalone Boost.AFIO library. GSoC is a lot of work not well paid for what it is, but you will find it benefits you immeasurably not just in your career, but usually as a person too.

Here is your programming test. It consists of wiring a bit of functionality from std::filesystem into the asynchronous batch execution nature of this library, and writing a Boost.Test unit test to test your new code. You will be doing a lot this during this GSoC project. It ought to be a good test of your suitability for this GSoC project.

Submission deadline: **Monday 6th May 11.59pm GMT** (i.e. the time in London Greenwich).

Some notes before we begin:

1. Failure to complete this test is *not a failure*! In fact we are far more interested in **how** you go about failing. For example, did you panic, did you cut corners to deliver on time, if so how did you panic and which corners did you cut. That sort of thing. We are looking more for personality traits and how you deal with pressure than coding ability or technical knowledge.

2. Therefore you must absolutely 100% send us whatever your attempt is whether it works or even compiles or not. As I said, how you go about failing is far more important that whether you fail.

3. You **are** allowed to ask questions, just as in real GSoC. Please use nialldouglas14@gmail.com. Do bear in mind I'll be away from email contact for several hours at a time this weekend so there will be delays in replies. Remember also the timezone difference. All these things mirror real GSoC.

4. I wrote, debugged, valgrinded and finished this assignment in exactly **58 minutes** testing on Windows 7 x64 with VS2012 using the experimental Nov 2012 CTP compiler, Ubuntu 12.04 on ARM using GCC 4.8 and clang 3.3, and Ubuntu 12.04 on x64 using GCC 4.6. I expect it will take you rather longer.

Preparation:

1. Go clone https://github.com/ned14/TripleGit.git using git. You'll be interested in the GSoC2013_test branch, so checkout that. It needs to pull in submodules, specifically NiallsCPP11Utilities.
2. For Linux, BSD et al, get at least GCC 4.6. GCC 4.8 and clang 3.2 also work. If you choose clang you may see an error about a missing std::current_exception, if so I'd recommend switching to GCC (it's easier). For Linux BSD et al you'll also need a build tool called "scons". On Debian/Ubuntu, 'apt-get install scons' is sufficient.
3. For Windows, get Visual Studio Express 2012 for Windows (it's free). You'll need the experimental C++11 Microsoft compiler called "Nov 2012 CTP" which you can find at http://www.microsoft.com/en-us/download/details.aspx?id=35515.

4. You'll need a copy of Boost v1.53 or later on your system (earlier won't work). Easiest is to get a copy from http://sourceforge.net/projects/boost/files/boost/1.53.0/ and unpack into a "boost" directory in the TripleGit directory i.e. TripleGit/boost. Follow the boost instructions to build it. Note that on Windows x64, Boost needs to have the "address-model=64" specified to b2 in the Developer Tools Command Line, and the documentation doesn't mention this (failure to do this means you'll get lots of link errors later).
5. On Linux, typing scons in the TripleGit directory should throw any errors for missing dependencies, and if none it ought to build a full set of release binaries. Do apt-get install X as appropriate until all dev library dependencies are satisfied. On Windows, load the supplied VS2012 project files and hit compile.
6. In the GSoCTest directory you'll find a main.cpp. It simply contains main() and is otherwise empty. This is going to receive your submission.
7. Note that doxygen generated documentation lives in html. You will find this useful.
8. Before you change any lines of code, make **sure** everything compiles. Otherwise something is wrong. Note that "scons –debugbuild" builds debug binaries, and is far faster to compile. Also "scons –j 4" will do a parallel build.

Assignment:

Your assignment is as follows:

Part 1:

Write an implementation modeling a new member function in triplegit::async_io::async_file_io_dispatcher_base:

```
std::pair<std::vector<future<std::vector<std::filesystem::path>>>, std::vector<async_io_op>>
enumerate(const std::vector<async_io_op> &ops, const std::vector<std::filesystem::path> &paths);
```

This function will enumerate the contents of each of the directories specified by *paths* using std::filesystem::directory_iterator (my code maps the Boost implementation of filesystem in as if it were std::filesystem, so see http://www.boost.org/doc/libs/1_53_0/libs/filesystem/doc/reference.html#Class-directory_iterator). For each path in *paths*, there can be a corresponding optional precondition in *ops* (the precondition, because async_io_op is typedefed as a shared_ptr<>, can be null which means there is no precondition and therefore can be executed immediately). Your asynchronously called closure (callback function) which you pass to call() which schedules the asynchronous enumeration will, for each directory enumeration, return that enumeration as a vector<filesystem::path> which should consist of leafnames only.

Because this is a batch call, this implies many such directory enumerations, and thus takes in two vectors, one of dependencies *ops* (when to execute) and one of what to enumerate *paths*. It therefore returns a pair of two vectors, the first vector being a vector of futures (http://en.cppreference.com/w/cpp/thread/future) each returning a vector of filenames, the second vector the completion ops such that other items may be chained to complete when these complete. As call() returns a vector of futures, your implementation will simply return the vector of directory contents as a vector of futures.

As much as this task may seem daunting, the implementation code is approximately *ten* lines of C++. Tips:

1. triplegit::async_io::async_file_io_dispatcher_base::call() does almost all the work for you, including taking care of checking preconditions and scheduling the work and assembling the vector of futures to return.
2. Examine async_file_io.hpp for example patterns to clone.
3. Don't bother modifying async_file.io.hpp as every time you change it you'll have to recompile the library, which is very, very slow. Place your new implementation in the unit test main.cpp file below while it is being tested. Obviously you'll have to adjust its API above to match.

Part 2:

Write a unit test testing your new code above using the Boost Testing framework. As I mentioned earlier, use GSoCTest/main.cpp for this. You'll probably want to start with:

```
using namespace triplegit::async_io;
using namespace std;
using triplegit::async_io::future;
```

For convenience, we'll be using the lightweight Boost.Test framework – I've already included the header in the main.cpp for you. Your main() function will need to test your new code by doing this:

1. Asynchronously create a test directory called "testdir" using dispatcher->dir(… file_flags::Create).
2. Asynchronously create 100 directories "testdir/N" where N is 0…99 using dispatcher->dir(… file_flags::Create). Obviously these must only be invoked when testdir has completed.
3. Asynchronously create inside each of those 100 directories ten files "testdir/N/M" where M is 0…9. Obviously these must only be created when their containing directories from step 2 have completed. To create a file, simply do a dispatcher->file(… file_flags::Create|file_flags::Write), and then dispatcher->close() on each of the file creation completions.
4. Asynchronously execute your directory enumeration routine on each of the 100 directories, obviously only once the files have been closed and therefore created.
5. Wait for all outstanding operations to complete.
6. Test each of the future vectors returned by your routine to ensure that each contains the expected list of ten filename entries named 0…9. Use BOOST_TEST() as the test macro.
7. Clean up the test by asynchronously deleting all the files you created earlier.
8. Clean up the test by asynchronously deleting all the directories you created earlier.
9. Clean up the test by asynchronously deleting "testdir".
10. Return the success or failure of the unit test to Boost's automated framework by returning boost::report_errors() from main().

As much as the above seems "huge", almost all of it is a copy & paste job from unittests/main.cpp. Feel free to copy & paste as much as is needed from there (I did). For your interest, my example implementation came in at about 120 lines of C++, almost all of which was copy & paste. Tips:

1. There is an interesting cause of unit test failure which only presents itself on Linux and not Windows due to differences in how directory enumeration works.
2. I haven't finished my async_file_io_dispatcher_base::barrier() implementation yet, so you may get stuck on how to get your directory enumeration to wait until all the files you created have closed because you won't see how to reduce many completions into fewer (that's what barrier() will be for). You can work around this temporarily by chaining your directory enumeration to complete only when the very last file has been closed i.e. use manyclosedfiles.back() as the precondition dependency.

Completion instructions:

I'd suggest you clone my github repo above to your own account, and push your submission there before the closing date and time. A github gist of main.cpp is also fine. Post a link to your repo on your Google Melange public review, and **write 100-200 words on what you did, why you did it and your experience of the test**.

I wish you all the very best of luck!

Niall Douglas