Proposal for Mixxx's project "Non-Blocking Database Access"

Nazar "tr0" Gerasymchuk,

Nazar.Gerasymchuk@gmail.com

May 1, 2013

Let me introduce myself. I'm getting Master degree at Cybernetics Department of Taras Shevchenko National University of Kyiv, Ukraine (http://univ.kiev.ua/en).

I'm Qt programmer for about 3 years. I have some experience with programming databases (MySQL, SQLite), threads etc.

I still have burning desire to take a part in development of Mixxx. Also, I'm interested in music production. You can listen some of my own tracks here – http://soundcloud.com/tr0. Sometimes I write programming notes at my blog – http://neval8.wordpress.com. Hope, there will be more on related to development of Mixxx themes soon.

Contents

1 Intro

"Non-Blocking Database Access" is one of the clearest projects for me.

I have some skills and practice with programming databases using Qt. Also I have a theoretical base I've not used yet. I'm sure the Mixxx can be a right place to improve my database programming skills.

2 What do I have now?

- My working OS is Debian GNU/Linux now.
- I'm registered user at Launchpad and I'm able to check out Mixxx sources.
- I'm studying the `bzr` usage. I see that `bzr` is nice distributed revision control system.
- I accomplished connecting `scons` to QtCreator, so I got my favorite IDE working with `scons`. And I compiled it (it took about 10 minutes on 4 cores). Here is draft article about how I do that:
  `http://neval8.wordpress.com/2013/04/30/using-scons-with-qtcreator`.

3 What'll I do next?

- According to the fact that Mixxx is written in C++, we can make some static analysis by `cppcheck`[1], as I did and got a bunch of warnings. Here, I can learn Mixxx sources deeper.
- Fix some issues found by `cppcheck`. Maybe, it is not critical, but it helps to learn sources and become more familiar with it.
- Fix some issues and bugs from bugtracker.
- Work on my ideas and test them.
- Think out on my roadmap at GSoC 2013.

Some time ago, Ryan have described precisely the problem with databases on mailing list: *Today's approach of doing some operations on the GUI thread blocks Qt from processing events. This has implications on Mixxx's responsiveness because things like waveform rendering cannot do work while the Qt main thread is blocked attempting to read/write from the database. As Daniel mentioned on the waveform thread, sometimes*

[1]Cppcheck (`http://cppcheck.sourceforge.net`) is a static analysis tool for C/C++ code. Unlike C/C++ compilers and many other analysis tools, it does not detect syntax errors in the code. Cppcheck primarily detects the types of bugs that the compilers normally do not detect. The goal is to detect only real errors in the code (i.e. have zero false positives).

normal, small library operations hog the main thread for up to 20 ms. This is enough to cause a dropped frame when rendering the waveform at a reasonable FPS. It also increases the overall latency of the `ControlObject` system when the control events are proxied through the Qt event queue. So database queries on the main thread can add to the latency of pressing a button / slider / knob on the GUI. These are all motivating factors for moving database queries to a thread.

4 How I see the problem?

After exploring Mixxx sources I've found out that we have concrete DAO in GUI thread to access database. That concrete DAO applies query to database. Maybe, I missed something, but I haven't seen any errors processing (because a lot of things can happen while applying query) `// that is another story`.

So, we need to keep all business logic the same, but bring all database queries beyond GUI thread. All what is needed is usage of bare lambdas (introduces in new C++11 standard) and Qt's `QtConcurrent::run`.

4.1 How to use lambdas here?

Lambdas[2] can help not to mess the code in places where "fix" will be applied. We just move the code that is called after response arrives to the lambda and pass it to the `DAO` as callback parameter. In this case we do not hang the application, but can gently show "wait" message.

4.2 QtConcurrent

`QtConcurrent`[3] is simple mechanism to get programs multi-threaded with minimal overhead and also with minimal control on respective threads that is enough for us. Also, we can pass lambda to `QtCoucurrent::run` as parameter.

To implement user interaction while database is applying query, we should do next:

- Avoid applying database queries from GUI thread. *It should be `QtConcurrent` with lambda which I propose.*
- We must agree on how UI will react while applying queries:
  - What to do in case of "quick" query (for example, $< 200ms$)?
  - What to do in case of "long" query (for example, $\approx 3s$)?

[2] *"Lambdas in C++"* at Wiki – `http://en.wikipedia.org/wiki/Anonymous_function#C.2B.2B`, *"What is a lambda expression in C++11?"* at StackOverflow – `http://stackoverflow.com/questions/7627098/what-is-a-lambda-expression-in-c11`.

[3] *"QtConcurrent Namespace"* at QtProject – `http://qt-project.org/doc/qt-4.8/qtconcurrent.html`.

  - Who, when, where and how will inform user (for example, show `ProgressBar`, show `MessageBox` or so on)?
  - Is button "Cancel" planned?

5 How to solve problem?

I propose to rewrite code of calling `DAO`-objects. I created minimal project to show what I recommend to do. Here, you can see it: `http://github.com/troyane/lambdaConcurrent`.

Main scheme is next:

- In GUI:
  1. Prepare query string (as it was before).
  2. Prepare lambda *"how to freeze GUI"*
     (Inform user. Show some progress bar etc.).
  3. Prepare lambda *"how to unfreeze GUI"*
     (Inform user. Hide some progress bar etc.).
  4. Call concrete `DAO`s `applyQuery` function with (2) and (3) parameters (lambdas *"how to freeze GUI"*, *"how to unfreeze GUI"*).
  5. Release GUI thread – let it flow as it is.
- In Concrete `DAO`s `applyQuery`:
  1. Apply own event filter to block user input.
  2. Wrap all code originally placed in concrete `DAO` into lambda.
  3. Send lambda work to other thread (using `QtConcurrent::run`).
  4. Control time of thread working (using `QFuture`).
     - If thread is working longer than some constant limit time, we must apply received lambda as a parameter *"how to freeze GUI"*.
     - If thread overtimed, than we need to apply received lambda as a parameter *"how to unfreeze GUI"*.
  5. Remove own event filter to unblock user input.

We'll wrap all code of `DAO`'s with the next construction (see `DAO::applyQuery` in file `http://github.com/troyane/lambdaConcurrent/blob/master/dao.cpp)`.

I tried to comment as clear as I can, but if you have questions – you are welcome.

6 Approximate roadmap

1. Learn Mixxx sources (to understand how does it work) – *continuous process.*
   a) Play with code.

   b) Create own brunch.
   c) Solve my initial problems with code on IRC channel.

2. Fix bugs (to get involved into development process) – *continuous process.*
3. Learn more about Qt, multi-threading, databases etc related to Mixxx development – *continuous process.*
4. Implement my idea on concrete example.
   a) Discuss idea on IRC, on mail list.
   b) Write code.
   c) Write documentation.
   d) Publish general scheme on how to implement code for database access.
5. Rewrite all database access entries in Mixxx source to comply main scheme.
   - Find all entries.
   - Discuss entries.
   - Make changes.
   - Test applied changes.
6. Discuss all changes.
7. Perform general test.
8. Work on general documentation.
9. Code revise.