GROMACS - Feature #950

Path/directory/filename handling in Gromacs

05/31/2012 05:55 AM - Teemu Murtola

Status:	New
Priority:	Normal
Assignee:	
Category:	core library
Target version:	
Difficulty:	uncategorized

Description

Currently, futil.h has two defines, PATH_SEPARATOR and DIR_SEPARATOR, which are set to ';' and '\\' on Windows, and ':' and '/' otherwise. PATH_SEPARATOR is used in futil.c and fflibutil.c, DIR_SEPARATOR ~45 times in various contexts. This would need some clean-up, since it doesn't really work on Windows. Also, it would be better to not have such an #ifdef in an installed header. Note that, e.g., fopen() accepts '/' also on Windows, so there is no need to use '\\' when constructing our own filenames. Further, it is possible to have '/' as directory separator in argv[0] even on Windows (at least CTest does that), which breaks several things in Gromacs.

The last is currently causing tests in https://gerrit.gromacs.org/#/c/1000/ to fail on Windows.

I think there are two reasonable options:

- Use an external library that does cross-platform filename/path manipulations (Roland has suggested boost).
- Write our own wrappers, like the currently very simple ones in src/gromacs/utility/path.*.

Issues to consider:

- Is it necessary for the GMXLIB environment variable to be interpreted differently on Windows than on other systems, like it is now?
- If we implement our own, how should it work? Should it normalize everything to use '/' internally, even on Windows? Should it accept both '/' and '\\' as directory separators (either always, or just on Windows), or have more complex logic?

Associated revisions

Revision ea8524ad - 07/04/2012 01:25 AM - Teemu Murtola

Improve ProgramInfo and use it everywhere.

- The ProgramInfo class now stores the full command line.
Moved functionality to add quotes to arguments with spaces from
gmx::test::CommandLine to ProgramInfo.
- ProgramInfo initialization is now protected by a mutex for
completeness.
- oenv.c and statutil.c now use ProgramInfo internally to store/return
the binary name and command line. Removes duplicate implementations
of this functionality. Required changing the return values of
ProgramInfo methods from std::string values to const references.
- Removed many unnecessary #include directives from these two files.
- Fixed warnings that were produced from these files when switching them
to C++ compilation (one cppcheck warning suppressed for now).
- Add a temporary hack for Windows to try to make get_libdir() work for
tests that are run through CTest.

Helps with #950.

Change-Id: I1bfd4231b8b7055d0a014b41be67a7c1c99e36b0

Revision d873b7de - 03/29/2014 02:52 PM - Teemu Murtola

Optimize intermodule dependencies

- Move genconf from gmxana to gmxpreprocess. sortwater.* is used by
genconf and grompp, so moving them to the same place allows moving
these files as well, removing all dependencies between gmxana and
gmxpreprocess.
- Move utility/path.* to fileio/. This removes a cyclic dependency

between utility and fileio modules. Can be moved back if #950 gets resolved such that nothing in path.h no longer depends on futil.h.

Change-Id: Ia81173d4845143dc3cc94e0c200cab96c87308a5

Revision 68fc3130 - 01/20/2015 06:45 PM - Mark Abraham

Removed gmx_header_config.h

Now we no longer install a header just to get platform independence for functionality that is not actually part of the the installed API.

Moved DIR_SEPARATOR into its own non-installed header.

Moved snprintf MSVC-workaround define into its own non-installed header.

Moved GMX_NATIVE_WINDOWS definition to config.h

Removed checks related to the use of gmx_header_config.h

Refs #950, #1454

Change-Id: I6aebcddd6772dfa82bf1214c6ed42f9da6ac22e0

History

#1 - 06/01/2012 08:14 AM - Roland Schulz

I don't think path handling by itself is complex enough to make it worth using an external library. Only when looking at all of futil.c, it is a lot of code we have to maintain ourselves and I would think having boost::filesystem could do that.

Regarding the two issues you raised:
- I'm not sure what you mean with GMXLIB different interpreted on Windows. Do you mean that the PATH_SEPARATOR is used for it instead of always using the same character?
- It seems the easiest if we always use / to catenate paths and always allow both / and \ when separating. I don't think this can go wrong unless someone tries to have a file with a \ on Unix or tries to run it on OpenVMS.

#2 - 06/01/2012 09:15 AM - Teemu Murtola

Roland Schulz wrote:

> I don't think path handling by itself is complex enough to make it worth using an external library. Only when looking at all of futil.c, it is a lot of code we have to maintain ourselves and I would think having boost::filesystem could do that.

There's perhaps half of that file (~500 lines) that is not Gromacs-specific and would benefit from an external library. So that's the amount of code one would save. More, if you consider also the path manipulations etc. Less, if one considers that we probably wouldn't want most of our code to have a direct dependence on boost::filesystem (since, among other things, I think it says it brings boost::mpl in, and probably also windows.h on Windows), but instead write a futil.c-like wrapper for it. The wrapping code will be easier to maintain, though. But I don't really have any strong opinion on this one. An external library is likely to work better than on our implementation, in particular if it transparently handles stuff like long UNC paths (\\?\) on Windows.

> - I'm not sure what you mean with GMXLIB different interpreted on Windows. Do you mean that the PATH_SEPARATOR is used for it instead of always using the same character?

Yes, and also that it uses DIR_SEPARATOR.

#3 - 06/01/2012 09:39 AM - Roland Schulz

> I think it says it brings boost::mpl in, and probably also windows.h on Windows

While source depends on windows.h, the headers don't. So windows.h isn't an issue (I suppose you meant the namespace issue with e.g. MIN/MAX). mpl is an issue if we judge dependencies by their line-numbers (which I think doesn't make sense - because it is a poor proxy for potential problems with dependencies). But I don't see how having a wrapper would help with that.

> Is it necessary for the GMXLIB environment variable to be interpreted differently on Windows than on other systems, like it is now?

I don't think it is necessary or high priority. But I think it would be nicer. As an environment variable is it public and as a user I wouldn't expect : as PATH_SEPARATOR.

#4 - 06/01/2012 10:19 AM - Teemu Murtola

Roland Schulz wrote:

> I think it says it brings boost::mpl in, and probably also windows.h on Windows

While source depends on windows.h, the headers don't. So windows.h isn't an issue (I suppose you meant the namespace issue with e.g. MIN/MAX).

Partly that, but mostly the size of the header (=compilation time). I was assuming that it was header-only, sorry for that.

> mpl is an issue if we judge dependencies by their line-numbers (which I think doesn't make sense - because it is a poor proxy for potential problems with dependencies). But I don't see how having a wrapper would help with that.

I think there are two potential problems with mpl:

- If there are compilation problems with boost on more exotic platforms, I would assume that they mainly come from the use of some advanced mpl construct.
- Compilation time: a lot of lines and heavy use of templates both make compilation slower. Even if it just a fraction of a second for a single file, if the headers now get included in most of our files (right now in the order of a thousand), it quickly adds up...

But that said, I don't have any strong opinion against it (but Erik might...).

> Is it necessary for the GMXLIB environment variable to be interpreted differently on Windows than on other systems, like it is now?

I don't think it is necessary or high priority. But I think it would be nicer. As an environment variable is it public and as a user I wouldn't expect : as PATH_SEPARATOR.

On the other hand, as a user coming from Unix (and Gromacs is still essentially Unix-like on Windows), I would expect it's behavior not to change. How about making it accept both ';' and ':' (and both '/' and '\\', if done for other paths as well), and perhaps warn/error out if both are used simultaneously? But I agree that this is not of a very high priority.

#5 - 06/01/2012 10:43 AM - Roland Schulz

Teemu Murtola wrote:

> Roland Schulz wrote:
>
> > mpl is an issue if we judge dependencies by their line-numbers (which I think doesn't make sense - because it is a poor proxy for potential problems with dependencies). But I don't see how having a wrapper would help with that.
>
> I think there are two potential problems with mpl:
>
> - If there are compilation problems with boost on more exotic platforms, I would assume that they mainly come from the use of some advanced mpl construct.
> - Compilation time: a lot of lines and heavy use of templates both make compilation slower. Even if it just a fraction of a second for a single file, if the headers now get included in most of our files (right now in the order of a thousand), it quickly adds up...
>
> But that said, I don't have any strong opinion against it (but Erik might...).

All dependencies on mpl I could find easily where from the source too (not easy to do because bcp doesn't support dependencies of parts of a library). The size of mpl is actually not as large as it might seem. Large part is conditional (compiler fallbacks) and preprocessed headers. Another reason why size doesn't make sense as a criteria. The preprocessing doesn't add potential problems but adds large percentage of the code.

#6 - 06/01/2012 10:48 AM - Roland Schulz

Actually this is a case where using an external library can increase portability. Our own implementation would be only supported on some platforms (either because we are not aware of issue or ignore very rare platforms by design (e.g. assuming slash or back-slash are the only choices for path-separator). A well supported external library has the potential of supporting more platforms.

#7 - 06/04/2012 01:27 PM - Teemu Murtola

Roland Schulz wrote:

> Actually this is a case where using an external library can increase portability.

I agree with you there (see also my comment 2 on how it may make things work better on Windows). But if the external library does not compile on

some platform, the effort to make things work is potentially much bigger. With proper wrappers, it's still not that much, though.

As for conditional compilation, it may not directly add to the complexity, but it does indicate that these constructs have had some problems with compiler support. So a new/exotic platform may also have issues that are not solved by the existing #ifdefs.

There are other reasons to wrap external libraries than compilation time or portability, and since the wrapping is not really that hard to do when done initially/on-the-need basis, I would suggest that we do it. Because if a library is first taken into extensive use, and then we realize that we would like to have a wrapper, the amount of work is much larger. In the current code base, there are already at least two layers of wrapping on top of the basic file APIs, so it's likely we will want to add some custom functionality that isn't included in boost::filesystem...

For the compilation time issue, this is just not idle talk: I just tried, and removing the dependency on boost::exception from gromacs/utility/exceptions.h speeds up compilation of files that depend on it (that is, nearly all C++ files) by ~15%. And I think that that mostly comes from just the number of lines in the headers and their dependencies, since we really don't use much of the code from the headers.

But these are just concerns that need to be taken into account in the decision. As I said, I don't have any strong opinion either way. So probably this needs to wait until 4.6 is out to have more resources for the discussion...

#8 - 06/04/2012 04:35 PM - Roland Schulz

I agree mostly. Just two minor comments
- As far as I understand a lot of the MPL code size is preprocessed automatically for several compilers (to reduce compile time and not require the preprocessor). Thus it doesn't automatically imply that it would not work for others.
- Maybe it might be worth to test whether precompiled headers might help with the exceptions compile time.

#9 - 10/20/2012 04:08 AM - Roland Schulz

I looked at this issue again and I can't really find a good library which would make sense in our context. I can't find a tiny C++ library which has portable OS functions like file/path classes. The ones which have good portable classes are all huge (QT, Poco, or Boost) and none let one easily take a small subset. The boost dependencies would be small if we needed the MPL anyhow. But currently that doesn't seem to be the case.
What would be possible is to take the respective classes from either QT or Poco and remove the dependencies on the rest of the library (for Boost the file classes probably depend too much on MPL to make this viable). This would nullify the advantage of a library that maintenance is done by others (too much changes are probably required to make an update to a newer version easy), but it might make it still easier to get a good well tested initial version so that hopefully little maintenance is needed. It still might be faster than basing it on the Gromacs C code.

#10 - 02/25/2014 02:37 AM - Erik Lindahl

- Affected version set to future

I just figured I should chip in my $0.02 to (maybe) save Mark from having to address this ;-) However, this is an opinion, not an imperial command.

Given the experience from the last 15 months of development for 5.0 (and that we still want to see all C++ parts tested for real on lots of esoteric installs), I think it would be good to move to a system where we have a discussion about potential new optional dependencies or external parts to include at the beginning of the development cycle for each new major version. There is also the problem that a developer who wants something will keep bugging and keep a discussion alive (while others might be less interested), so I also think it is good to combine this with a deadline relatively early in the cycle, after which we no longer consider new additions for that release.

I am not an expert on the path details in particular (so don't take this as specific criticism for the file stuff), but there is also the more philosophical discussion: If we always end up including a file from boost whenever there is something we need that boost can do, we will eventually end up having most of boost distributed with Gromacs, although large parts will not be relevant.

Not having boost as a dependency was one of the things that we agreed on as a boundary condition when we decided to allow C++, but occasionally it feels as if we repeatedly get stuck in the discussion that because boost has some useful functionality (which it obviously does), we should add more files from boost unless somebody else comes up with a specific proposal for how to solve it another way?

Again, I'm happy to have a discussion after 5.0 about what parts of boost in particular we want (and maybe we have been too conservative), but I think one important aspect of that discussion also needs to be how to define a small set of modules and stick to those long-term, rather than gradually nominating more parts from boost in every release?

#11 - 02/25/2014 03:16 AM - Roland Schulz

Something which might work is having boost as an optional dependency. The implementation of path (https://gerrit.gromacs.org/#/c/2975/8/src/gromacs/utility/path.cpp,cm) would stay as is without a full boost (and thus e.g. not work correctly on a case-insensitive FS on Linux or if symlinks are involved) and would use boost if available (and thus also work correctly in the corner cases. In general the default implementation would be simpler and restricted for corner cases. I'm not sure that would be a good solution. I'm just adding it as an option.

#12 - 02/28/2014 09:17 AM - Teemu Murtola

- Category set to core library

- Target version changed from 5.0 to 5.x

I generally agree with Erik, but there's one point that would need more consideration:

Erik Lindahl wrote:

> Given the experience from the last 15 months of development for 5.0 (and that we still want to see all C++ parts tested for real on lots of esoteric installs), I think it would be good to move to a system where we have a discussion about potential new optional dependencies or external parts to include at the beginning of the development cycle for each new major version. There is also the problem that a developer who wants something will keep bugging and keep a discussion alive (while others might be less interested), so I also think it is good to combine this with a deadline relatively early in the cycle, after which we no longer consider new additions for that release.

If the problem is that people keep bugging about an issue until it is solved, it doesn't really help to just forbid that or ignore that bugging on a pretense that "now is not the time for this discussion". ;) Also, it doesn't help if there is just discussion, since that easily ends up with nothing changing because no one wants to say the final work. That is only going to make other people annoyed.

What would help is that someone would *make a decision* after one or a few rounds of comments, and *justify* that decision so that it is clear that all concerns have been considered, even if all of them could not be solved by the decision. The most important thing would be that these decisions would hold for more than a few months, everyone would be aware of them, and they would not be made retrospectively (i.e., it would be clear *before* starting a full-blown implementation that people will not complain about the external dependencies or such). Otherwise, it's a very unwelcome environment for people outside of Stockholm to contribute anything...

For this particular issue, I don't think it would have taken me more than maybe a few effective working days (probably longer in calendar time) to write something that for sure works better than the current approach. But I have not, since there has not been a clear decision that everyone would agree to, and I have seen enough of cases where people get complained at several months after they have implemented something and the implementation has even passed review in Gerrit.

I also want to mention it that there may be parts of boost that we can bring in without such heavyweight discussion. I think that things like boost::scoped_ptr make much more sense to include from there, than to reimplement ourselves; our implementation would simply duplicate all the code, but could very easily work incorrectly in corner cases (if trying to make shortcuts to make it simpler), or at least lose the benefit that people reading/writing the code need to look at Gromacs-specific constructs instead of being able to rely on a well-known implementation.

For this particular issue, I don't see a strong need for Boost. There is currently a single piece of code in https://gerrit.gromacs.org/#/c/2975/ where Roland brought up the point of it "not working" on a case-sensitive file system. But even this depends on the definition of "not working": even with boost::filesystem, I would prefer to implement the functionality that this piece of code supports using path::compare() instead of boost::filesystem::equivalent(). The cases where the user manually runs things from the build directory and provides a path that does not match in case with the one used by CMake during compilation time should be sufficiently rare to not needing full support (I think it is unlikely that, e.g., CMake or CTest would trigger such corner cases).

#13 - 03/28/2014 05:22 AM - Gerrit Code Review Bot

Gerrit received a related patchset '1' for Issue #950.
Uploader: Teemu Murtola (teemu.murtola@gmail.com)
Change-Id: Ia81173d4845143dc3cc94e0c200cab96c87308a5
Gerrit URL: https://gerrit.gromacs.org/3297

#14 - 06/11/2014 11:39 PM - Erik Lindahl

- Tracker changed from Bug to Feature

#15 - 01/08/2015 04:42 PM - Gerrit Code Review Bot

Gerrit received a related patchset '1' for Issue #950.
Uploader: Mark Abraham (mark.j.abraham@gmail.com)
Change-Id: I6aebcddd6772dfa82bf1214c6ed42f9da6ac22e0
Gerrit URL: https://gerrit.gromacs.org/4356

#16 - 01/08/2015 04:50 PM - Erik Lindahl

It was a very long time ago I wrote this, but since we don't use it for much functionality I think it's easier to use our own simple code instead of introducing a large external dependency.

The only thing to consider on windows is that the GMXLIB variable might have been set through a dialog box selecting a folder, and in that case I think it will be the windows standard using semicolon and backslash.

I don't have any strong feelings about how things work under windows. The only concern might be that those users are likely to be the least experienced ones we have, so they might not understand differences that are obvious to normal Linux users.

#17 - 01/08/2015 04:52 PM - Erik Lindahl

Forget about my comments - my brain isn't engaged yet, so I kept commenting on Teemu's original ideas rather than Mark's actual patch. I'm sure the patch is fine :-)

#18 - 01/08/2015 07:28 PM - Mark Abraham

Erik Lindahl wrote:

> It was a very long time ago I wrote this, but since we don't use it for much functionality I think it's easier to use our own simple code instead of introducing a large external dependency.

Writing our own platform-independence code is not free, either. Somee good news about Boost::Filesystem is that it has just been accepted as an ISO Technical Specification, which puts it on the path to being part of C++17. We still wouldn't want to import all of it (particularly if MPL is involved) for a fairly niche problem. Localizing DIR_SEPARATOR to a single source file shouldn't be too big a deal once we get more of the Gromacs hinterland compiling as C++.

> Forget about my comments - my brain isn't engaged yet, so I kept commenting on Teemu's original ideas rather than Mark's actual patch. I'm sure the patch is fine :-)

I'm only sealing steerage hatches on the Titanic, not even shifting deckchairs...

#19 - 07/11/2016 08:24 PM - Mark Abraham

- Target version deleted (5.x)