Notice

I'm leaving Github. The main official location for this project is now:
https://codeberg.org/RafaGago/mini-async-log

Minimal Asynchronous Logger (MAL)

A performant asynchronous data logger with acceptable feature-bloat.

Credit

- To my former employer **Diadrom AB.** for allowing me to share this code with a BSD license. They funded most of the development of this project.
- To Dmitry Vjukov for all the knowledge he has spread on the internets, including the algorithms for the two queues on this project.
- To my girlfriend for coexisting with me when I become temporarily autistic after having been "in the zone" for too long.

Motivation

This started with the intention to just develop an asynchronous logger that could be used from different dynamically loaded libraries on the same binary without doing link-time hacks like being forced to link static, hiding symbols and some other "niceties".

Then at some point way after the requirements were met I just improved it for fun.

Design rationale

To be:

- Simple. Not over abstracted and feature bloated, explicit, easy to figure out what the code is doing, easy to modify (2017 self-comment: not that easy after having switched to raw C coding for a while :D).
- Very low latency. Fast for the caller.
- Asynchronous (synchronous calls can be made on top for special messages, but they are way slower than using a synchronous logger in the first place).
- Have minimum string formatting on the calling thread for the most common use cases.
- Don't use thread-local-storage, the user threads are assumed as external and no extra info is attached to them.
- Have termination functions to allow blocking until all the logs have been written to disk in program exit (or signal/fault) scenarios.

Various features

- Targeting g++4.7 and VS 2010(can use incomplete or broken C+11 features from boost).

- Nanosecond performance.
- No singleton by design, usable from dynamically loaded libraries. The user provides the logger instance either explicitly or by a global function (Koenig lookup).
- Suitable for soft-realtime work. Once it's initialized the fast-path can be clear from heap allocations (if properly configured to).
- File rotation-slicing.
- One conditional call overhead for inactive logging levels.
- Able to strip log levels at compile time (for Release builds).
- Lazy parameter evaluation (as usual with most logging libraries).
- No ostreams(*) (a very ugly part of C++: dynamically allocated, verbose and stateful), just format strings checked at compile time (if the compiler supports it) with type safe values.
- The logger severity threshold can be externally changed outside of the process. The IPC mechanism is the simplest, the log worker periodically polls some files when idling (if configured to).
- Fair blocking behaviour (configurable by severity) when the bounded queue is full and the heap queue is disabled. The logger smoothly starts to act as a synchronous logger. If non-blocking behavior is desired an error is returned instead.
- Small, you can actually compile it as a part of your application.

(*)An on-stack ostream adapter is available as a last resort, but its use is more verbose and has more overhead than the format literals.

> see this example
> that more or less showcases all available features.

How does it work

It just borrows ideas from many of the loggers out there.

As an asynchronous logger its main objetive is to be as fast and to have as low latency as possible for the caller thread.

When the user is to write a log message, the producer task is to serialize the data to a memory chunk which then the logging backend (consumer) can decode, format and write. No expensive operations occur on the consumer, and if they do it's when using secondary features.

The format string is required to be a literal (compile time constant), so when encoding something like the entry below...

> log_error ("File:a.cpp line:8 This is a string that just displays the next number {}, int32_val);

...the memory requirements are just a pointer to the format string and a deep copy of the integer value. The job of the caller is just to serialize some bytes to a memory chunk and to insert the chunk into a queue.

The queue is a mix of two famous lockfree queues of Dmitry Vyukov (kudos to this genious) for this particular MPSC case. The queue is a blend of a fixed capacity and fixed element size array based preallocated queue and an intrusive node based dynamically allocated queue. The resulting queue is still linearizable.

The format string is type-safe and validated at compile time for compilers that support "constexpr" and "variadic template parameters". Otherwise the errors are caught at run time on the logged output (Visual Studio 2010 mostly).

There are other features: you can block the caller thread until some message has been dequeued by the logger thread, to do C++ stream formatting on the caller thread, etc.

File rotation

The library can rotate log files.

Using the current C++11 standard files can just be created, modified and deleted. There is no way to list a directory, so the user is required to pass the list of files generated by previous runs.at startup time.

> There is an example
> here.

Initialization

The library isn't a singleton, so the user should provide a reference to the logger instance on each call.

There are two methods to pass the instance to the logging macros, one is to provide it explicitly and the other one is by providing it on a global function.

If no instance is provided, the global function "get_mal_logger_instance()" will be called without being namespace qualified, so you can use Koenig lookup/ADL. This happens when the user calls the macros with no explicit instance suffix, as e.g. "log_error(fmt string, ...)".

To provide the instance explictly the macros with the "_i" suffix need to be called, e.g. "log_error_i(instance, fmt_string, ...)"

The name of the function can be changed at compile time, by defining MAL_GET_LOGGER_INSTANCE_FUNCNAME.

Termination

The worker blocks on its destructor until its work queue is empty when normally exiting a program.

When a signal is caught you can call the frontend function on termination in your signal handler. This will flush the logger queue and early abort any synchronous calls.

Errors

As of now, every log call returns a boolean to indicate success.

The only possible failures are either to be unable to allocate memory for a log entry or an asynchronous call that was interrupted by "on_termination". A filtered out call returns true".

The logging functions never throw.

Compiler macros

Those that are self-explanatory won't be explained.

- *MAL_GET_LOGGER_INSTANCE_FUNC*: See the "Initialization" chapter above.
- *MAL_STRIP_LOG_SEVERITY*: Removes the entries of this severity and below at compile time, so they are not evaluated and don't take code space. 0 is the "debug" severity, 5 is the "critical" severity.
  Stripping at level 5 leaves no log entries at all. Yo can define e.g. MAL_STRIP_LOG_DEBUG, MAL_STRIP_LOG_TRACE, etc. instead. If you define MAL_STRIP_LOG_TRACE all the severities below will be automatically defined for you (in this case MAL_STRIP_LOG_DEBUG).
- *MAL_DYNLIB_COMPILE*: Define it when **compiling** MAL as a dynamic library/shared object.
- *MAL_DYNLIB*: Define it when **using** MAL as a dynamic library. Don't define it if you are static linking or compiling the library with your project.
- *MAL_CACHE_LINE_SIZE*: The cache line size of the machine you are compiling for. This is just used for data structure padding. 64 is defaulted when undefined.
- *MAL_USE_BOOST_CSTDINT*: If your compiler doesn't have use boost.
- *MAL_USE_BOOST_ATOMIC*
- *MAL_USE_BOOST_CHRONO*
- *MAL_USE_BOOST_THREAD*
- *MAL_NO_VARIABLE_INTEGER_WIDTH*: Integers are encoded ignoring the number trailing bytes set to zero, not based on its data type size. So when this isn't defined e.g. encoding an uint64 with a value up to 255 takes one byte (plus 1 byte header). Otherwise all uint64 values will take 8 bytes (plus header), so encoding is less space efficient in this way but it frees the CPU and allows the compiler to inline more.

compilation

You can compile the files in the "src" folder and make a library or just compile everything under /src in your project.

Otherwise you can use cmake.

On Linux there are Legacy GNU makefiles in the "/build/linux" folder too. They respect the GNU makefile conventions. "DESTDIR", "prefix", "includedir" and "libdir" can be used. These are mainly left there because the examples were not ported to CMake.

REMEMBER (for legacy users that use boost): If the library is compiled with e.g. the "MAL_USE_BOOST_THREAD" and "MAL_USE_BOOST_CHRONO" preprocessor variables the client code must define them too. TODO: config.h.in

Windows compilation

There is a Visual Studio 2010 Solution in the "/build/windows" folder, but you need to do a step before opening:

If you don't need the Boost libraries you should run the "build\windows\mal-log\props\from_empty.bat" script. If you need them you should run the "build\windows\mal-log\props\from_non_empty.bat" script.

If you don't need the Boost libraries you can open and compile the solution, otherwise you need to edit (with a text editor) the newly generated file ""build\windows\mal-log\props\mal_dependencies.props" before and to update the paths in the file. You can do this through the Visual Studio Property Manager too.

Performace

It used to be some benchmark code here but the results were reported as being off compared with what some users where getting (out of date versions?).

The tests also had the problem that they were assuming a normal distribution for the latencies.

It was not very nice to add a lot of submodules to this project just for building the benchmarks.

Because all of these reasons I have created a separate repository with some benchmark code. It uses CMake to compile everything and build a static linked executable, so it's very easy to build and run.

If you are writing a logger and want to add it or have some suggestions about how to improve the bencmark code Pull Requests are welcome.

> Here is the benchmark project..

> Written with StackEdit.

我的修复:

```
C:\Program Files (x86)\Microsoft Visual Studio\2019\Enterprise\VC\Tools\MSVC\14.29.30133\include\type_traits:1003: error: C2338: You've instantiated std::aligned_storage<Len, Align> with an extended alignment (in other words, Align > alignof(max_align_t)). Before VS 2017 15.8, the member "type" would non-conformingly have an alignment of only alignof(max_align_t). VS 2017 15.8 was fixed to handle this correctly, but the fix inherently changes layout and breaks binary compatibility (*only* for uses of aligned_storage with extended alignments). Please define either (1) _ENABLE_EXTENDED_ALIGNED_STORAGE to acknowledge that you understand this message and that you actually want a type with an extended alignment, or (2) _DISABLE_EXTENDED_ALIGNED_STORAGE to silence this message and get the old non-conforming behavior.
```

_ENABLE_EXTENDED_ALIGNED_STORAGE

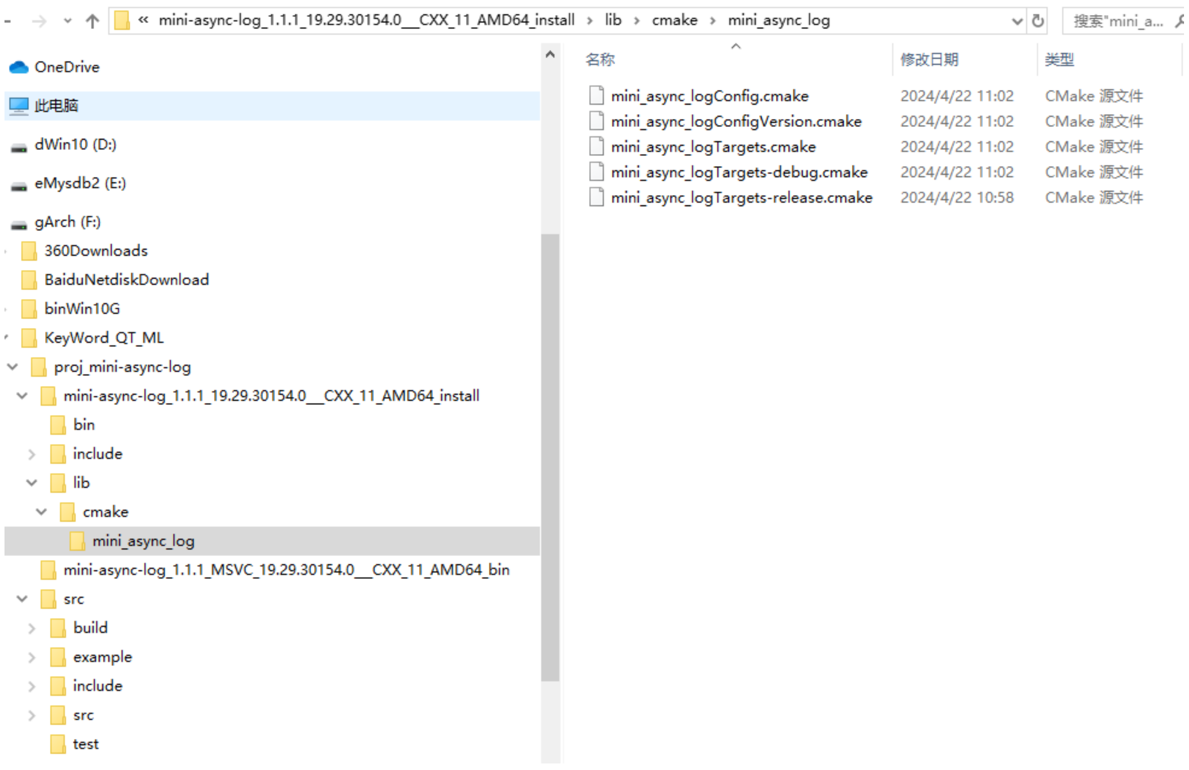
_DISABLE_EXTENDED_ALIGNED_STORAGE

```
add_definitions(-D_ENABLE_EXTENDED_ALIGNED_STORAGE) # added by  2024-4-22
```

compile ok.

add test example.

add cmake configure files support.



mini-async-log_1.1.1_19.29.30154.0__CXX_11_AMD64_install › lib › cmake › mini_async_log

- OneDrive
- 此电脑
- dWin10 (D:)
- eMysdb2 (E:)
- gArch (F:)
  - 360Downloads
  - BaiduNetdiskDownload
  - binWin10G
  - KeyWord_QT_ML
  - proj_mini-async-log
    - mini-async-log_1.1.1_19.29.30154.0__CXX_11_AMD64_install
      - bin
      - include
      - lib
        - cmake
          - mini_async_log
    - mini-async-log_1.1.1_MSVC_19.29.30154.0__CXX_11_AMD64_bin
    - src
      - build
      - example
      - include
      - src
      - test

名称	修改日期	类型
mini_async_logConfig.cmake	2024/4/22 11:02	CMake 源文件
mini_async_logConfigVersion.cmake	2024/4/22 11:02	CMake 源文件
mini_async_logTargets.cmake	2024/4/22 11:02	CMake 源文件
mini_async_logTargets-debug.cmake	2024/4/22 11:02	CMake 源文件
mini_async_logTargets-release.cmake	2024/4/22 10:58	CMake 源文件