inactivity log for davidz

What's that? Chicken?

Monday, June 27, 2011

Writing a C library, part 1

This is part one in a series of blog-posts about best practices for writing C libraries.

Base libraries

Since libc is a fairly low-level set of libraries, there exists higher-level libraries to make C programming a more pleasant experience including libraries in the GLib and and GTK+ stack. Even while the following is going to be somewhat GLib- and GTK+-centric, these notes are written to be useful for any C code whether it's based on libc, GLib or other libraries such as NSPR, APR or some of the Samba libraries.

Most programmers would agree that it's usually a bad idea to implement basic data-types such as string handling, memory allocation, lists, arrays, hash-tables or queues yourself just because you *can* - it only makes code harder to read and harder to maintain by others. This is where C libraries such as GLib and GTK+ come into play - these libraries provides much of this out of the box. Plus, when you end up needing non-trivial utility functions (and chances are you will) for, say, Unicode manipulation, rendering complex scripts, D-Bus support or calculating checksums, ask yourself (or worse: wait until your manager or peers ask you) if the decision to avoid a well-tested and well-maintained library was a good decision.

In particular, for things like cryptography, it is usually a bad idea to implement it yourself (however inventing your own algorithm is worse); instead, it is better to use an existing well-tested library such as NSS (and even if you do, be careful of using the library correctly). Specifically, said library may even be FIPS-140 certified which is a requirement if you want to do business with the US government.

Similarly, while it’s more efficient to use e.g. epoll than poll for event notification, maybe it doesn't matter if your application or library is only handling on the order of ten file descriptors. On the other hand, if you know that you are going to handle thousands of file descriptors, you can still use e.g. GLib for the bulk of your library or application - just use epoll from dedicated threads. Ditto, if you need O(1) removal from a list, maybe don’t use a GList - use an embedded list instead.

There was an error in this gadget

There was an error in this gadget

Blog Archive

- ► 2012 (3)
- ▼ 2011 (12)
  - ► September (1)
  - ► July (3)
  - ▼ June (3)
    - Writing a C library, part 3
    - Writing a C library, part 2
    - Writing a C library, part 1
  - ► April (2)
  - ► February (2)
  - ► January (1)
- ► 2010 (3)
- ► 2009 (4)
- ► 2008 (6)
- ► 2007 (16)
- ► 2006 (18)
- ► 2005 (39)
- ► 2004 (14)
- ► 2003 (8)

About Me

davidz

Boston, MA, United States

Software Engineer at Red Hat, Inc.

View my complete profile

Above all, no matter what libraries or code you end up using, make sure you have at least a rudimentary understanding of the relevant data-types, concepts and implementation details. For example, with GLib it is very easy to use high-level constructs such as GHashTable, g_timeout_add() or g_file_set_contents() without knowing how things are implemented or what a file descriptor really is. For example, when saving data, you want to do so atomically (to avoid data-loss) and just knowing that g_file_set_contents() Does The Right Thing(tm) is often enough (often just reading the API docs will tell you what you need to know). Additionally make sure you understand both the algorithmic complexity of the data-types you end up using and how they work on modern hardware.

Finally, try not to get caught up in religious discussions about “bloated” libraries with random people on the Internet - it’s usually not a good a use of time and resources.

Checklist

- Don’t reinvent basic data-types (unless performance is a concern).
- Don’t avoid standard libraries just because they are portable.
- Be wary of using multiple libraries with overlapping functionality.
- To the extent where it’s possible, keep library usage as a private implementation detail.
- Use the right tool for the right job - don’t waste time on religious discussions.

Library initialization and shutdown

Some libraries requires that an function, typically called foo_init(), is called before other functions in the library is called - this function typically initializes global variables and data structures used by the library. Additionally, libraries may also offer a shutdown function, typically called foo_shutdown() (forms such as foo_cleanup(), foo_fini(), foo_exit() and the grammatically dubious foo_deinit() have also been observed in the wild), to release all resources used by the library. The main reason for having a shutdown() function is to play nicer with Valgrind (for finding memory leaks) or to release all resources when using dlopen() and friends.

In general, library initialization and shutdown routines should be avoided since they might cause interference between two unrelated libraries in the dependency chain of an application; e.g. if you don’t call them from where they are used, you are possible forcing the application to call a init() function in main(), just because some library deep down in the dependency chain is using the library without initializing it.

However, without a library initialization routine, every function in the library would have to call the (internal) initialization routine which is not always practical and may also be a performance concern. In reality, the check only has to be done in a couple of functions since most functions in a library depends on an object or struct obtained from e.g. other functions in the library. So in reality, the check only has to be done in _new() functions and functions not operating on an object.

For example, every program using the GLib type system has to call g_type_init() and this includes libraries based on libgobject-2.0 such as libpolkit-gobject-1 - e.g. if you don't call g_type_init() prior to calling polkit_authority_get_sync() then your program will probably segfault. Naturally this is something most people new to the GLib stack gets wrong and you can't really blame them - if anything, g_type_init() is a great poster-child of why init() functions should be avoided if possible.

One reason for library initialization routine has to do with library configuration, either app-specific configuration (e.g. the application using the library might want to force a specific behavior) or end-user specific (by manipulating argc and argv) - for example, see gtk_init(). The best solution to this problem is of course to avoid configuration, but in the cases where it's not possible it is often better to use e.g. environment variables to control behavior - see e.g. the environment variables supported by libgtk-3.0 and the environment variables supported by libgio-2.0 for examples.

If your library does have an initialization routine, do make sure that it is idempotent and thread-safe, e.g. that it can be called multiple times and from multiple threads at the same time. If your library also has a shutdown routine, make sure that some kind of "initialization count" is used so the library is only shutdown once all users of it have called its shutdown() routine. Also, if possible, ensure that your library init/shutdown routines calls the init/shutdown routines for libraries that it depends on.

Often, a library's init() and shutdown() functions can be removed by introducing a *context object* - this also fixes the problem of global state (which is undesirable and often break multiple library users in the same process), locking (which can then be per context instance) and callbacks / notification (which can call back / post events to separate threads). For example, see libudev's struct udev_monitor.

Checklist

- Avoid init() / shutdown() routines - if you can't avoid them, do make sure they are idempotent, thread-safe and reference-counted.
- Use environment variables for library initialization parameters, not argc and argv.
- You can easily have two unrelated library users in the same process - often without the main application knowing about the library at all. Make sure your library can handle that.
- Avoid unsafe API like atexit(3) and, if portability is a concern, unportable constructs like library constructors and destructors (e.g. gcc's __attribute__ ((constructor)) and __attribute__ ((destructor))).

Memory management

It is good practice to provide a matching free() function or each kind of allocated object that your API returns. If your library uses reference counting, it is often more appropriate to use the suffix _unref instead of _free. An example of this in the GLib/GTK+ stack the functions used are g_object_new(), g_object_ref() and g_object_unref() that operate on instances of the GObject type (including derived types). Similarly, for the GtkTextIter type, the relevant functions are gtk_text_iter_copy() and gtk_text_iter_free(). Also, note that some objects may be stack-allocated (such as GtkTextIter) while others (such as GObject) can only be heap-allocated.

Note that some object-oriented libraries with the concept of derived types may require the app to use the unref() method from a base type - for example, an instance of a GtkButton must be released with g_object_unref() because GtkButton is also a GObject. Additionally, some libraries have the concept of floating references (see e.g. GInitiallyUnowned, GtkWidget and GVariant) - this can make it more more convenient to use the type system from C since it e.g. allows using the g_variant_new() constructor in place of a parameter like in the example code for g_dbus_proxy_call_sync() without leaking any references.

Unless it’s self-evident, all functions should have documentation explaining how parameters are managed. It is often a good idea to try to force some kind of consistency on the API. For example, in the GLib stack the general rule is that the caller owns parameters passed to a function (so the function need to take a reference or make a copy if the parameter is used after the function returns) and that the callee owns the returned parameters (so the caller needs to make a copy or increase the reference count) unless the function can be called from multiple threads (in which case the caller needs to free the returned object).

Note that thread-safety often dictates what the API looks like - for example, for a thread-safe object pool, the lookup() function (returning an object) must return a reference (that the caller must unref()) because the returned object could be removed from another thread just after lookup() returns - one such example is g_dbus_object_manager_get_object().

If you implement reference counting for an object or struct, make sure it is using atomic operations or otherwise protect the reference count from being modified simultaneously by multiple threads.

If a function is returning a pointer to memory that the caller isn’t supposed to free or unref, it is often necessary to document for how long the pointer is valid - for example the documentation for the getenv() C library function says “The string pointed to by the return value of getenv() may be statically allocated, and can be modified by a subsequent call to getenv(), putenv(3), setenv(3), or unsetenv(3).”. This is useful information because it shows that care should be taken if the result from getenv() is used by multiple threads; also this kind of API can never work in a multi-threaded application and the only reason it works is that applications or libraries normally don’t modify the environment.

It is often advantageous for an application to not worry about out-of-memory conditions and instead just call abort() if the underlying allocator signals an out-of-memory condition. This holds true for most libraries as well since it allows a simpler and better API and huge code-footprint reductions. If you do decide to worry about OOM in your library, do make sure that you test all code-paths or your effort will very likely have been in vain. On the other hand, if you know your library is going to be used in e.g. process 1 (the init process) or other critical processes, then not handling OOM is not an option.

Checklist

- Provide a free() or unref() function for each type your library introduces.
- Ensure that memory handling consistent across your library.
- Note that multi-threading may impose certain kinds of API.
- Make sure the documentation is clear on how memory is managed.
- Abort on OOM unless there are very good reasons for handling OOM.

Multiple Threads and Processes

A library should clearly document if and how it can be used from multiple threads. There are often multiple levels of thread-safety involved - if the library has a concept of objects and a pool of objects (as most libraries do), the enumeration and management of the pool might be thread safe while applications are supposed to provide their own locking when operating on a single object from multiple threads, concurrently.

If you are providing a function performing synchronous I/O, it is often a good idea to make it thread-safe so an application can safely use it from a helper thread

If your library is using threads internally, be wary of manipulating process-wide state, such as the current directory, locale, etc. Doing so from your private worker thread will have unexpected consequences for the application using your library.

A library should always use thread-safe functions (e.g. getpwnam_r() rather than getpwnam()) and avoid libraries and code that is not thread-safe. If you can’t do this, clearly state that your library isn’t thread-safe so applications can use it from a dedicated helper process instead if they need thread-safety.

It is also important to document if your library is using threads internally, e.g. for a pool of worker threads. Even though you think of the thread as a private implementation detail, its existence can affect users of your library; e.g. Unix signals might need to be handled differently in the the presence of threads, and there are extra complications when forking a threaded application.

If your library has interfaces involving resources that can be inherited over fork(), such as file descriptors, locks, memory obtained from mmap(), etc, you should try to establish a clear policy for how an application can use your library before/after a fork. Often, the simplest policy is the best: start using nontrivial libraries only after the fork, or offer a way to reinitialize the library in the forked process. For file descriptors, using FD_CLOEXEC is a good idea. In reality most libraries have undefined behavior after the fork() call, so the only safe thing to do is to call the exec() function.

Checklist

Document if and how the library can be used from multiple threads.

Document what steps need to be taken after fork() or if the library is now unusable.

Document if the library is creating private worker threads.

Posted by davidz at 3:49 PM
Labels: C, GLib, Programming

8 comments:

Josh Triplett June 27, 2011 at 6:49 PM

If your library does do proper synchronization for multiple threads, but you don't want to pay the overhead of pthreads for single-threaded programs, or you want to preserve portability to platforms without pthreads, try libpthread-stubs.

Reply

Aater Suleman June 28, 2011 at 12:11 AM

I am think that this is a very tricky requirement. If you make all adds atomic, its a large amount of overhead. I don't see how you can prevent their modification by multiple thread without adding critical section or severely limiting functionality. Perhaps I am misunderstanding your point. Can you please clarify.

Aater (i.e., the futurechips guy)

Reply

davidz June 28, 2011 at 7:51 AM

Hey Aater, just to be clear the suggestion was only to use atomic ops for reference counting, not any kind of add - the typical use is like this

http://git.gnome.org/browse/glib/tree/gio/gfileattribute.c?id=2.29.8#n923

using g_atomic_int_inc() and g_atomic_int_dec_and_test() for atomic operations. See their docs here

http://developer.gnome.org/glib/unstable/glib-Atomic-Operations.html#glib-Atomic-Operations.description

in particular the implementation details. Now, implementation-wise, with recent GLib libraries and recent gcc compilers, these are implemented as macros that expand using gcc built-ins, see http://git.gnome.org/browse/glib/tree/glib/gatomic.h?id=2.29.8#n72

which IIRC results to just a couple of assembler instructions if using a modern CPU (such as x86_64) and falls back to a slower path otherwise. See http://gcc.gnu.org/onlinedocs/gcc/Atomic-Builtins.html#Atomic-Builtins and http://gcc.gnu.org/wiki/Atomic

Reply

j.eng June 28, 2011 at 8:39 AM

>"Don't reinvent basic data-types (unless performance is a concern)."

glib violates this very simple rule right from the start. gpointer is just as bad as PVOID. "Just fuckin write void *."

>Avoid init() / shutdown() routines [...] Avoid unsafe API like atexit(3) and, if portability is a concern, unportable constructs like library constructors and destructors (e.g. gcc's __attribute__ ((constructor)) and __attribute__ ((destructor))).

Explicit destructor seems better. libgcrypt has a track record with all those issues http://comments.gmane.org/gmane.comp.encryption.gpg.libgcrypt.devel/2126 where it seems that explicit init/exit (refcounted of course) is the superior solution.

>if you don't call them from where they are used, you are possible forcing the application to call a init() function in main(), just because some library deep down in the dependency chain is using the library without initializing it.

Fix the intermittent component in question, not main.

Reply

Crazy June 28, 2011 at 9:11 AM

-- Library initialization and shutdown --

"Avoid init() / shutdown() routines - if you can't avoid them, do make sure they are idempotent, thread-safe and reference-counted."

This is not clear enough. Global init and shutdown should be avoided or treated as non-thread/fork safe. A structure that holds all the state information for a library should have corresponding init() and shutdown() calls.

"Use environment variables for library initialization parameters, not argc and argv."

At a low level, a library should have all parameters set with API calls. One level up would include a call such as get_env_opts() to grab environement variable settings and get_cmd_opts(char *) to parse a well defined command line argument list. Then the library user or executable can decide how to handle the library configuration.

"You can easily have two unrelated library users in the same process - often without the main application knowing about the library at all. Make sure your library can handle that."

Uh... this makes little to no sense. But I'll supplement it with... make sure to clearly document the capabilities and behavior of your library in a multi-threaded or multi-process environment. Simply making something "thread-safe" is pointless without context and usage guidance.

"Avoid unsafe API like atexit(3) and, if portability is a concern, unportable constructs like library constructors and destructors (e.g. gcc’s __attribute__ ((constructor)) and __attribute__ ((destructor)))."

OK... so to put this more simple, if portability (of source code) is of high concern, know your dependencies! Highly portable code should by default avoid compiler dependent extensions, and non-POSIX functions. The exception is allowing for tweaked code to be enabled with CPP (pre-processor macros).

Reply

Crazy June 28, 2011 at 9:11 AM

-- Memory management --

"Provide a free() or unref() function for each type your library introduces."

Agreed. But IMHO:
- type_init - sets and allocated type to sane values
- type_new - allocates a type and inits the new allocation
- type_free - deallocates a type
- type_ref, type_getref, type_get - creates a new reference (increments reference count)
- type_unref, type_release, type_put - removes reference (decrements reference count)

Another one I like to include (separate from type_free()) is:
void type_destroy(type_t ** t)
Usually when freeing memory, you should always

free(ptr);
ptr = NULL;

I like to simplify that to a one liner that looks like:

type_destroy(&ptr);

type_destroy exists to decrement reference count, deallocate memory if reference count is zero, and sets pointer to NULL so subsequent "if (ptr)" checks operate as intended.

"Ensure that memory handling consistent across your library."

Memory management should be complete and well defined. As always, it should be clearly described in documentation with trivial and non-trivial examples.

"Note that multi-threading may impose certain kinds of API."

This should be inherit in the programmers skill set, but I agree. OO programming with instanced variables/references and locking mechanisms lend toward a more friendly multi-threaded experience. Static variables, global variables, lend to a less thread safe experience.

My rule of thumb for this is usually:
If there no non-constant variables defined globally, the library is _capable_ of being thread safe. The effort to make the library thread safe depends on the locking or concurrency logic overhead required.

"Make sure the documentation is clear on how memory is managed."

Documentation should always have trivial AND non-trivial examples and a list of potential Pitfalls

"Abort on OOM unless there are very good reasons for handling OOM."

Disagree. There may be valid exceptions to this, and one involves allocations that require _HUGE_ amounts of memory. If this occurs and fails, it is likely something caused by user input and not that the system is low on memory. Prime example is loading a >4GB file into memory on a 32bit system.

Some extra advise (stuff I've struggled with in the past with my own libraries) includes:
- Minimize the usage of "user-defined" void* types.
- When doing memory management, don't forget to think about where the memory your pointers are pointing to is located with respect to the heap or the stack. Nastyness can occur if you free stack memory or don't free heap memory.
- C does not do reference counting, so having a reference counting mechanism for C in a multi-threaded environment should be an absolute requirement. Realistically, you won't be able to always track when to free an allocated type without reference counting. This should especially be considered when using lists or trees to store references to memory.

Reply

Crazy June 28, 2011 at 9:12 AM

-- Multiple Threads and Processes --

"Document if and how the library can be used from multiple threads."

Agreed, documentation should include trivial and non-trivial examples and a list of potential pitfalls.

"Document what steps need to be taken after fork() or if the library is now unusable."

You need to understand that your library is now duplicated but looking at the same file descriptors and streams as another as well as having a different pid. To handle this case gracefully, you'll need advanced locking, or IPC techniques. I agree that it'd be safer to just exec() if possible.

"Document if the library is creating private worker threads."
And don't forget about child processes.

Some extra advised that I've struggled with in the past with my own libraries include:
- Another multi-threading pitfall I've found is knowing where/when to create a new reference to memory. In short, you should always create a reference that a thread will use outside and before the thread execution (if applicable.) The issue is when you have multiple threads executing on a structure, at any time a thread can "unref" the memory potentially causing it to be freed. But as long as your current scope has a valid reference, reference counted memory should prevent it from being freed.

Reply

Crazy June 28, 2011 at 9:12 AM

This article had all types of goodies, but you really have to know what you're looking for to find them. To prevent from being the stereotypical internet pessimist... here are some real comments I have on the article (without peer review.) ;-)

As a foot note:
I'd advise you get a peer review on any follow up parts to this series to make the language flow a little better. Other than the roughness of the article, it did have some good advise.

Thanks,
Chenz

Reply

Enter your comment...

Comment as: Unknown (Goog

Sign out

Publish Preview

Notify me

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

inactivity log for davidz

What's that? Chicken?

Tuesday, June 28, 2011

Writing a C library, part 2

This is part two in a series of blog-posts about best practices for writing C libraries. Previous installments: part one.

Event handling and the main loop

Event-driven applications, especially GUI applications, are often built around the idea of a "main loop" that intercepts and dispatches all sorts of events such as key/button presses, incoming IPC calls, mouse movements, timers and file/socket I/O and so on. The main loop typically "call back" into the application whenever an event happens.

For example, the GLib/Gtk+ library stack is built around the GMainContext, GMainLoop and GSource types and other library stacks provide similar abstractions. Many kernel and systems-level programmers often look funny at GUI programmers when they utter the word "main loop" - much the same way GUI programmers stare confused at kernel programmers when they say put() or get() something. The truth is that a main-loop is really a well-known concept with a different name: it's basically an abstraction of OS primitives such as select(2), poll(2) or equivalent on e.g. Windows.

It is important to note that a multi-threaded application may run different main loops in different threads to ensure that callbacks happen in the right threads - in GLib this is achieved by using the g_main_context_push_thread_default() function which records the main loop for the current thread in thread-local storage. This variable is in turn read when starting an asynchronous operation (such as g_dbus_connection_call() or g_input_stream_read_async()) to ensure that the passed callback function is invoked in a thread running a main loop for the context set with g_main_context_push_thread_default() earlier.

Some main loops, for example the GLib one, allows creating recursive main loops and this is used to implement GtkDialog's run() method. While this indeed appears to block the calling thread, it is important to note that events are *still* being processed (to e.g. process input events and redraw animations). Specifically, this means that the functions (plural since applies to everything in the call stack) that brought up the dialog might end getting called again (from a callback). Thus, when using functions like gtk_dialog_run() you need to ensure that your functions are either

There was an error in this gadget

There was an error in this gadget

Blog Archive

About Me

davidz
Boston, MA, United States

Software Engineer at Red Hat, Inc.

View my complete profile

re-entrant or that they are guaranteed to not get called when the dialog is showing (typically achieved by making the dialog modal so the UI action triggering the display of the dialog can't be accessed). Because of pitfalls like this, you **must** clearly document if a function is using a recursive main loop.

Note that main loops are not a GUI-only concept - a lot of daemons (e.g. background process without any GUI) are built around this concept since it nicely integrates events from any source whether they are file descriptor based or synthetic such as timers or logging events. In fact, a considerable part of the system-level software on a modern Linux system is built on top of GLib and uses its main event loop abstraction to dispatch events - most of the time such daemons sit idle in one or more main loops and wait for D-Bus messages to arrive (to service a client), a timeout to fire (maybe to kick off periodic house-keeping tasks) or a child process to terminate (when using a helper program or process to do work).

Now that we've explained what a main loop is (or rather, what the *idea* of a main loop is), let's look at why this matters if you a writing a C library. First of all, if your library doesn't need to deliver events to users, you don't need to worry about main loops. Most libraries, however, are not that simple - for example, libudev delivers events when devices are plugged or changed, NetworkManager wants to inform of changes in networking and so on.

If your library is using GLib, it is often suitable to just require that the user runs the GLib main loop (if the application is using another main loop, it can either integrate the GLib main loop (like Qt does) or run it in a separate thread) and use g_main_context_get_thread_default() when setting up a callback. This is the way many GObject-based libraries, such as libpolkit-gobject-1, libnm-glib or libgudev work - for example, callbacks connected to the GUdevClient::uevent signal will be called in what was the thread-default main loop when the object was constructed. For a shared resource, such as a message bus connection, a good policy is that callbacks happen in what was the the thread-default main loop when the method was called (see e.g. g_dbus_connection_signal_subscribe() where this is the case) since applications or libraries have no absolute control of when the shared object was created. In any case, functions dealing with callbacks **must** always document in what context the callback happens in.

On the other hand, if your library is not using GLib, a good way to provide notification is simply to **a)** provide a file descriptor that e.g. turns readable when there are events to process; and **b)** provide a function that processes events (and possibly invoke callback functions registered by the user). A good example of this is libudev's udev_monitor_get_fd() and udev_monitor_receive_device() functions. This way the application (or the library using your library) can easily control what thread the event is handled in. As an example of how libudev is integrated into the GLib main loop, see here and here. In the libudev case, the returned file descriptor is the underlying netlink socket used to receive the event from udevd (via the kernel); in the case that there are no natural file descriptor (could be the event is only happening in response to a certain entry in, say, a log file), your library could use pipe(2) (or eventfd(2) if on Linux) and use a private worker thread to signal the other end.

If your library provide callback functions, make sure they take *user_data* arguments so the user can easily associate callbacks with other objects and data. If the scope of your callback is undefined (e.g. may fire more than once or if there is no way to disconnect the callback), also provide a way to free the *user_data* pointer when it is no longer needed - otherwise the application will leak data that needs to be freed later. See g_bus_watch_name() and for an example.

Checklist

- Provide APIs for main loop integration
- Make sure callback functions take *user_data* arguments (possibly with free an accompanying free function)

Synchronous and Asynchronous I/O

It is important for users of a library to know if calling a function involves doing synchronous I/O (also called blocking I/O). For example, an application with an user interface need to be responsive to user input and may even need to update the user interface every frame for smooth animations (e.g. 60 times a second). To avoid unresponsive applications and jerky animations, its UI thread must **never** call any functions that does any synchronous I/O.

Note that even loading a file from local disk may block for a very long amount of time - sometimes tens of seconds. For example the file may not be in the page cache and the hard disk with the file system may be powered down - or the file may be in the users home directory which could be on a network filesystem such as NFS. Other examples of blocking IO includes local IPC such as D-Bus or Unix domain sockets.

If an operation is known to take a long time to complete (synchronous or otherwise), it is often nice if it is possible to easily cancel the it (perhaps from another thread). For example, see the GCancellable type in the GLib stack. Another nicety (although easily implemented via the GCancellable type) is a way to set a timeout for potentially long-running operations - see e.g. g_dbus_connection_send_message_with_reply() and g_dbus_proxy_call() and note how the latter has an object-wide timeout so the timeout only has to be set once.

Some libraries provide both synchronous and asynchronous versions of a function where the former blocks the calling thread and the latter doesn't. Typically asynchronous I/O is implemented using worker threads (where the worker thread is doing synchronous I/O) but it could also involve communicating with another process via IPC (e.g. D-Bus) or even TCP/IP. For example, in the libgio-2.0 case asynchronous file I/O is implemented via synchronous IO (e.g. basically read(2) and write(2) calls) in worker threads (using a GThreadPool) simply because the Linux kernel does not (yet?) provide an adequate way to do asynchronous I/O suitable for libraries (see also: colorful notes about Asynchronous I/O). On the upside, this is mostly an implementation detail and the libgio-2.0 implementation can migrate to a non-threaded approach should such a mechanism be made available in the future.

Asynchronous I/O typically involves callbacks (or at least some kind of event notification) and thus involves a main loop. If a library provides functions for this, it should clearly state what thread the callback will happen in, and whether it requires the application to run a (specific kind of) main loop - see the previous section about main loops for details.

If a library is thread-safe, it is often easier for the application itself to just use the synchronous version of a function in a worker-thread - if using GLib, g_io_scheduler_push_job() is the right way to do that.

In some cases synchronous I/O is implemented by using a recursive main loop (typically by using the asynchronous form of the function) - this should be avoided as it typically causes all kinds of problems because of reentrancy and events being processed while waiting for the supposedly synchronous operation to complete. As always, clearly document what your code is doing.

Some libraries, such as those in the GLib stack, use a consistent pattern for asynchronous I/O for all of its functions involving the GAsyncResult / GSimpleAsyncResult, GAsyncReadyCallback and GCancellable types - this makes it a lot easier both for programmers and for higher-level language bindings especially since important things like life-cycles are part of this model (for example, you are guaranteed that the callback will always happen, even on cancellation, timeout or error).

Checklist

- Clearly document if a function does any synchronous I/O
- Ideally suffix synchronous functions it with _sync() so it's easy to inspect large code-trees using e.g. grep(1)
- Consider if an operation needs to be available in synchronous or asychronous form or both.
- Point to both synchronous and asynchronous functions in your API documentation.
- If possible, use an established model (such as the GIO model) for I/O instead of rolling your own

Posted by davidz at 4:15 PM
Labels: C, GLib, Programming

1 comment:

Amy Terry December 3, 2013 at 8:37 AM

Good one. Thanks.

Reply

Enter your comment...

Comment as: Unknown (Goog

Sign out

Publish Preview

Notify me

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

inactivity log for davidz

What's that? Chicken?

Wednesday, June 29, 2011

Writing a C library, part 3

This is part three in a series of blog-posts about best practices for writing C libraries. Previous installments: part one, part two.

Modularity and namespaces

The C programming language does not support the concept of namespaces (as used in e.g. C++ or Python) so it is usually emulated simply by using naming conventions. The main reason of namespaces is to avoid naming collisions - consider both libwoot and libkool providing a function called get_all_objects() - which one should be used if a program links to both libraries? Namespacing is an important part of a naming strategy and applies to variables, function names, type names (including structs, unions, enums and typedefs) and macros.

The standard convention is to use a short identifier, e.g. for libnm-glib you will see nm_ and NM being used, for Clutter it's clutter and Clutter and for libpolkit-agent-1, it's polkit_agent and PolkitAgent. For libraries that don't use CamelCase for its types, the same prefix is normally used for functions and types - for example, libudev the prefix used is simply udev.

Code that isn't using namespaces properly is not only hard to integrate into other libraries and applications (the chance of symbol collisions is high), there's also a chance that it will collide with future additions to the standard C library or POSIX standards.

One benefit of using namespaces in C (one that ironically is not present in a language with proper support for namespaces), is that it's a lot easier to pinpoint what the code is doing by just looking at a fragment of the source code - e.g. when you see an item being added to a container, you are usually not in doubt whether the programmer meant to invoke GtkContainer's add() method or ClutterContainer's add() method because of how C namespacing forces the programmer to be explicit, for better or worse.

In addition to choosing a good naming strategy, note that the visibility of what symbols (typically variables and functions) a library export can be fine-tuned, see these notes for why this is desirable.

On the topic of naming, it is usually a good idea to avoid C++ keywords (such as "class") for variable names, at least in header

There was an error in this gadget

There was an error in this gadget

Blog Archive

- ► 2012 (3)
- ▼ 2011 (12)
  - ► September (1)
  - ► July (3)
  - ▼ June (3)
    - Writing a C library, part 3
    - Writing a C library, part 2
    - Writing a C library, part 1
  - ► April (2)
  - ► February (2)
  - ► January (1)
- ► 2010 (3)
- ► 2009 (4)
- ► 2008 (6)
- ► 2007 (16)
- ► 2006 (18)
- ► 2005 (39)
- ► 2004 (14)
- ► 2003 (8)

About Me

davidz

Boston, MA, United States

Software Engineer at Red Hat, Inc.

View my complete profile

files that you except C++ code to include using e.g. extern "C". Additionally, generally avoid names of functions in the C standard library / POSIX for variable names such as "interface" or "index" because these functions can (and on Linux, actually is) be defined as macros.

Checklist

Choose a naming convention - and stick to it.

Do not export symbols that are not public API.

Error handling

If there's one statement that adequately describes error handling in the C programming language, it's perhaps that it's something that people rarely agree on. Most programmers, however, would agree that errors can be broken down into two categories **1)** programmer errors; **2)** run-time errors.

A *programmer error* is when the programmer isn't using a function correctly - e.g. passing a non-UTF-8 string to a function expecting a valid UTF-8 string such as g_variant_new_string() (if unsure, validate with g_utf8_validate() before calling the function) or passing an invalid D-Bus name to g_bus_own_name() (if unsure, validate with g_dbus_is_name() and g_dbus_is_unique_name() before calling).

Most libraries have undefined behavior in the presence of being used incorrectly - in the GLib case the macros g_return_if_fail() / g_return_val_if_fail() are used, see e.g. the checks in g_variant_new_string() and the checks in g_dbus_own_name(). Additionally, for performance, these checks can be disabled by defining the macro G_DISABLE_CHECKS when building either GLib itself or applications using GLib (but usually aren't). Not all parameters may be checked, however, and the check might not cover all cases because checks can be expensive. Combined with the G_DEBUG flag, it's even easy to trap this in debugger by running the program in an environment where G_DEBUG=fatal-warnings.

Having g_return_if_fail()-style checks is usually a trade-off - for example, GLib didn't initially have the UTF-8 check in g_variant_new_string() - it was only added when it became apparent that a considerable amount of users passed non-UTF-8 data which caused errors in unrelated code that was extremely hard to track down - see the commit message for details. If this cost is unacceptable, the programmer can easily use the g_variant_new_from_data() function passing TRUE as the *trusted* parameter.

Even with a library doing proper parameter validation (to catch programmer errors early on), if you pass garbage to a function you usually end up with undefined behavior and undefined behavior can mean *anything* including formatting your hard disk or evaporating all booze in a five-mile radius (oh noz). That's why some libraries simply calls abort() instead of carrying on pretending nothing happened. In general, a C library can **never** guarantee that it won't blow up no matter what data is passed - for example the user may pass a pointer to invalid data and, boom, SIGSEGV is raised when the library tries to accesses it. Of course the library *could* try to recover, longjmp(3) style, but since it's a library it can't mess around with process-wide state like signal handlers. Unfortunately, even smart people sometime fail to realize that the caller has a responsibility and instead blames the library instead of its user (for the record, libdbus-1 is fine which is why process 1 is able to use it without any problems). In most cases, problems like these are solved by just throwing documentation at the problem.

To conclude, when it comes to programmer errors, one key take away is that it's a good idea to document exactly what kind of input a function accepts. As the saying goes, "trust is good, control is better", it is also a good idea to verify that the programmer gets it right by using g_return_if_fail() style checks (and possibly provide API that does no such checks). Also, if your code does any kinds of checks, make sure that the functions used for checking (if non-trivial) are public so e.g. language bindings have a chance to validate input before calling the function (see also: notes on errors in libdbus).

A *run-time* error is e.g. if fopen(3) returns NULL (for example the file to be opened does not exist or the calling process is not privileged to open it), g_socket_client_connect() returns FALSE (the network might not be up) or g_try_malloc() returns NULL (might not have enough address space for a 8GiB array). By definition, run-time errors are recoverable although the code you are using might treat some (like malloc(3) failing) as irrecoverable because handling some run-time errors (such as OOM) would complicate the API not only on the function level (possibly by taking an error parameter), but also by requiring transactional semantics (e.g. rollback) on most data types (see also: write-up on why handling OOM is hard and a good explanation of Linux's overcommit feature).

For simple libraries just using libc's errno is often simplest approach to handling run-time errors (since it's thread-safe and every C programmer knows it) but note that some functions including asprintf(3) does not set errno to ENOMEM if e.g. failing to allocate memory. If you are basing your code on a library like GLib, use its native error type, e.g. GError, for run-time errors. An interesting approach to handling errors is the one used by the cairo 2D graphics library where (non-trivial) object instances track the error state (see e.g. cairo_status() and cairo_device_status()). There are many many other ways to convey run-time errors - as always, the important thing when writing a C library is to be consistent.

Checklist

- Document valid and invalid value ranges for parameters (if any) and provide facilities to validate parameters (unless trivial) for programmers and language bindings
- Try to validate incoming parameters at public API boundaries
- Establish a policy on how to deal with programmer errors (e.g. undefined behavior or abort()).

- Establish a policy on how to deal with run-time errors (e.g. use errno or GError)
- Ensure the way you handle run-time errors map to common exception handling systems.

Encapsulation and OO design

While C as programming language does not have built-in support for object-oriented programming lots of C programmers use C that way - in many ways it's almost hard *not* to. In fact, many C programmers regard the simplicity of C (compared to, say, C++) as a feature insofar that you are not bound to any one object model - for example, the kernel uses various OO techniques and the GLib/GTK+ stack has its own dynamic type system called GType on which the GObject base class (that many classes are derived from) is built.

There's of course a price to pay for defining your own object model - it typically involves more typing (identifiers are longer) and, especially for GObject, involves actual function calls to register properties, add private instance data and so on (example). On the other hand, such a dynamic type system often offer some level of type introspection so it's possible to easy link the property for whether a check-button widget is active with whether an text-entry widget should use password mode using the g_object_bind_property() function (screenshot). Polymorphism in GObject is provided by embedding a virtual method table in the class struct (example) and providing a C functions that uses the function pointer (example) - note that derived types can access the class struct to chain up (example).

One important feature of object-oriented design in C is that it usually promotes encapsulation and data hiding through the use of opaque data types - this is desirable as it allows extending the data type (e.g. adding more properties or methods) without breaking or requiring a recompile existing programs using the library (a future installment will discuss API and ABI and what it means wrt. API design). In an opaque data type, fields that would usually be in the C struct are hidden from the user and instead are made available via a getter (example) and/or setter (example) - additionally, if the object model support properties, the member may also be made available as a property (example) - for example, this is useful for notifying when the property changes.

Of course, not every single data structure need to be a full-blown GObject - for example, in some cases data hiding might not be desirable (sometimes it's awkward to use a C getter function) or maybe it's too slow to do from an inner loop (direct struct access is without a doubt faster). Also, for simple data structures it is sometimes desirable to initialize struct instances directly in the code.

Even when a full-blown object model (like GType and GObject) isn't used, it's never a bad idea to use opaque data structures and getters/setters. As an interesting alternative to this, note that some libraries explicitly allows extending a C structure without considering it an ABI change - while there's no easy way to enforce this (the user may allocate the structure on the stack), at least the library author can always tell the programmer that he shouldn't have done so (which may or may not be useful).

Checklist

Establish an object model for your library (if applicable).

Hide as many implementation details as is practical without impacting performance

Ensure that you can extend your library and types without breaking API or ABI.

If possible, build on top of an established and well-understood object system (such as the GLib one)

Posted by davidz at 6:12 PM

Labels: C, GLib, Programming

8 comments:

Robert Ancell June 30, 2011 at 7:25 AM

This is a really nice resource, thanks!

Reply

j.eng June 30, 2011 at 11:21 AM

>[errno] (since it's thread-safe and every C programmer knows it)

It may be safe from other threads due to TLS, but it is still a object with global scope, which results in having to longwindingly save and restore errno across calls to other opaque library functions. The Linux kernel in contrast shows how to do without such a global.

Reply

Philip Fry June 30, 2011 at 5:09 PM

>The Linux kernel in contrast shows how to do without such a global.

Is there an article on this I can read? Please post if so. It sounds interesting.

Reply

Joakim Hove June 30, 2011 at 6:39 PM

Very nice articles - thank you!

When it comes error handling I have made it a principle that _all_ functions calling abort() should have an accompanying _validate_input() function which the timid user can call first.

Reply

Stefan Sauer June 30, 2011 at 10:01 PM

One lesson learned on the topic of namespaces for me was to *not* use abbreviations for library names. In my pet project buzztard I initially named a library libbtcore and that happened to later on clash with a libbtcore from ktorrent. I realized that 'bt' could mean lots of things from buzztard, bittorrent, bluetooth and decided to rename my lib to libbuzztard-core. The ktorrent guy did not rename theirs upon request though.

Reply

rojtberg July 3, 2011 at 2:34 PM

> One benefit of using namespaces in C (one that
> ironically is not present in a language with proper
> support for namespaces), is that it's a lot easier to
> pinpoint what the code is doing by just looking at a
> fragment of the source code

actually it is a feature of the language with namespace support. C programmers maybe never heard of it in their whole life, but for completeness its called "abstraction" ;)

Its useful for instance when you port some code from gtk to clutter. Then all you have to do is change the type of my_container form GtkContainer to ClutterContainer instead of changing all the calls on my_container.
Of course this only works if you got the abstraction of your own code right in the first place.

Reply

davidz July 4, 2011 at 10:05 AM

@rotjberg: the point is that the code is easily grep(1)'able - also mentally grep'able which is what I tried to convey in the snippet you posted. If you've ever worked on e.g. a large c++ or Java codebase you will know what I'm talking about.

Reply

j.eng July 5, 2011 at 7:42 PM

@Phillip Fry (Is there an article on [linux kernel way of errno] I can read?): Not that I know of, but http://jengelh.medozas.de/2011/0705-lkerrno.php for a start.

Reply

Enter your comment...

Comment as: Unknown (Goog Sign out

Publish Preview ☐ Notify me

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

inactivity log for davidz

What's that? Chicken?

Friday, July 1, 2011

Writing a C library, part 4

This is part four in a series of blog-posts about best practices for writing C libraries. Previous installments: part one, part two, part three.

Helpers and daemons

Occasionally it's useful for a program or library to call upon an external process to do its bidding. There are many reasons for doing this - for example, the code you want to use

> might not be easily used from C - it could be written in say, python or, gosh, bash; or
>
> could mess with signal handlers or other global process state; or
>
> is not thread-safe or leaking or just bloated; or
>
> its error handling is incompatible with how your library does things; or
>
> the code needs elevated privileges; or
>
> you have a bad feeling about the library but it's not worth (or (politically) feasible) to re-implement the functionality yourself.

There are three main ways of doing this.

The first one is to just call fork(2) and start using the new library in the child process - this usually doesn't work because chances are that you are already using libraries that cannot be reliably used after the fork() call as discussed in previously (additionally, a lot of unnecessary COW might be happen if the parent process has a lot of writable pages mapped). If portability to Windows is a concern, this is also a non-starter as Windows does not have fork() or any meaningful equivalent that is as efficient.

The second way is to write a small helper program and distribute the helper along with your library. This also uses fork() but the difference is that one of the exec(3) functions is called immediately in the child process so all previous process state is cleaned up when the process image is replaced (except for file descriptors as they are inherited across exec() so be wary of undesired leaks). If using GLib, there's a couple of (portable) useful utility functions to do this (including support for automatically closing file descriptors).

The third way is to have your process communicate with a long-lived helper process (a socalled daemon or background process).

There was an error in this gadget

There was an error in this gadget

Blog Archive

- ► 2012 (3)
- ▼ 2011 (12)
  - ► September (1)
  - ▼ July (3)
    - Writing a C library, intro, conclusion and errata
    - Writing a C library, part 5
    - Writing a C library, part 4
  - ► June (3)
  - ► April (2)
  - ► February (2)
  - ► January (1)
- ► 2010 (3)
- ► 2009 (4)
- ► 2008 (6)
- ► 2007 (16)
- ► 2006 (18)
- ► 2005 (39)
- ► 2004 (14)
- ► 2003 (8)

About Me

davidz

Boston, MA, United States

Software Engineer at Red Hat, Inc.

View my complete profile

The helper daemon can be launched either by dbus-daemon(1) (if you are using D-Bus as the IPC mechanism), systemd if you are using e.g. Unix domain sockets, an init script (uuidd(8) used to do this - wasteful if your library is not going to get used) or by the library itself.

Helper daemons usually serve multiple instances of library users, however it is sometimes desirable to have a helper daemon instance per library user instance. Note that having a library spawn a long-lived process by itself is usually a bad idea because the environment and other inherited process state might be wrong (or even insecure) - see Rethinking PID 1 for more details on why a good, known, minimal and secure working environment is desirable. Another thing that is horribly difficult to get right (or, rather, horribly easy to get wrong) is uniqueness - e.g. you want at most one instance of your helper daemon - see Colin's notes for details and how D-Bus can be used and note that things like GApplication has built-in support for uniqueness. Also, in a system-level daemon, note that you might need to set things like the loginuid (example of how to do this) so things like auditing work when rendering service for a client (this is related to the Windows concept known as impersonation).

As an example, GLib's libproxy-based GProxy implementation uses a helper daemon because dealing with proxy servers involves a interpreting JavaScript (!) and initializing a JS interpreter from every process wanting to make a connection is too much overhead not to mention the pollution caused (source, D-Bus activation file - also note how the helper daemon is activated by simply creating a D-Bus proxy).

If the helper needs to run with elevated privileges, a framework like PolicyKit is convenient to use (for checking whether the process using your library is authorized) since it nicely integrates with the desktop shell (and also console/ssh logins). If your library is just using a short-lived helper program, it's even simpler: just use the pkexec(1) command to launch your helper (example, policy file).

As an aside (since this write-up is about C libraries, not software architecture), many subsystems in today's Linux desktop are implemented as a system-level daemons (often running privileged) with the primary API being a D-Bus API (example) and a C library to access the functionality either not existing at all (applications then use generic D-Bus libraries or tools like gdbus(1) or dbus-send(1)) or mostly generated from the IDL-like D-Bus XML definition files (example). It's useful to contrast this approach to libraries using helpers since one is more or less upside down compared to the other.

Checklist

- Identify when a helper program or helper daemon is needed
- If possible, use D-Bus (or similar) for activation / uniqueness of helper daemons.
- Communicating with a helper via the D-Bus protocol (instead of using a custom binary protocol) adds a layer of safety because message contents are checked.

Using D-Bus through a message bus router (instead of peer-to-peer connections) adds yet another layer of safety since the two processes are connected through an intermediate router process (a dbus-daemon(1) instance) which will also validate messages and disconnects processes sending garbage.

Hence, if the helper is privileged (meaning that it must **a)** treat the unprivileged application/library using it as untrusted and potentially compromised; and **b)** validate all data to it - see Wheeler's Secure Programming notes for details), activating a helper daemon on the D-Bus system bus is often a better idea than using a setuid root helper program spawned yourself.

If possible, in particular if you are writing code that is used on the Linux desktop, use PolicyKit (or similar) in privileged code to check if unprivileged code is authorized to carry out the requested operation.

Testing

A sign of maturity is when a library or application comes with a test suite; a good test suite is also incredible useful for ensuring mostly bug-free releases and, more importantly, ensuring that the maintainer is comfortable putting releases out without loosing too much sleep or sanity. Discussing specifics of testing is out of the scope for a series on writing C libraries, but it's worth pointing to the GLib test framework, how it's used (example, example and example) and how this is used by e.g. the GNOME buildbots.

One metric for measuring how good a test suite is (or at least how *extensive* it is), is determining how much of the code it covers - for this, the gcov tool can be used - see notes on how this is used in D-Bus. Specifically, if the test suite does not cover some edge case, the code paths for handling said edge case will appear as never being executed. Or if the code base handles OOM but the test suite isn't set up to handle it (for example, by failing each allocation) the code-paths for handling OOM should appear as untested.

Innovative approaches to testing can often help - for example, Mozilla employ a technique known as reftests (see also: notes on GTK+ reftests) while the Dracut test suite employs VMs for both client and server to test that booting from iSCSI work.

Checklist

Start writing a test suite as early as possible.

Use tools like gcov to ascertain how good the test suite is.

Run the test suite often - ideally integrate it into the build system ('make check'), release procedures, version control etc.

Posted by davidz at 3:19 PM
Labels: C, GLib, Programming

4 comments:

djcb July 2, 2011 at 6:21 AM

Regarding testing, g_test is very useful, but I think it'd be great to have some better documentation on how to integrate it with your project, e.g. with 'make check'

glib shows how to do it (using Makefile.decl etc.), but it's not trivial, and esp. not something most people would come up with by themselves.

Reply

davidz July 2, 2011 at 4:25 PM

Yeah, the state of build systems (and, as a generalization: IDE, RAD etc.) is kinda of a mess on (at least) Linux. I was thinking about covering it in my series... but since there really is no good guidance except for "copy/paste whatever autofoo other projects are using" it's probably just going to be a bullet-point in the list of items not covered with that guidance. Answers on a post-card....

Reply

RareCactus July 11, 2011 at 7:28 PM

> But since there really is no good guidance
> except for "copy/paste whatever autofoo other
> projects are using" it's probably just going to
> be a bullet-point in the list of items not
> covered with that guidance. Answers on a post-card...

I think the best guidance is to use CMake!

It is cross-platform (to all the platforms that anyone has used in the last 20 years), much MUCH easier than automake, and much faster!

And yes, I have used automake, cons, scons, jam, and even imake. CMake is the best.

Reply

Unknown February 25, 2016 at 1:57 PM

There is security flaw in "Disks". In the "about"-Dialogue, the website's link URL is not shown. It show rather show the complete link to copy and paste into the browser's address field.
No-one should trust and click unsolved links nowhere nowadays, not even in a well-known open source software.

Reply

Enter your comment...

Comment as: Unknown (Goog **Sign out**

Publish **Preview** ☐ Notify me

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

inactivity log for davidz

What's that? Chicken?

Tuesday, July 5, 2011

Writing a C library, part 5

This is part five in a series of blog-posts about best practices for writing C libraries. Previous installments: part one, part two, part three, part four.

API design

A C library is, almost by definition, something that offers an API that is used in applications. Often an API can't be changed in incompatible ways (it can, however, be extended) so it is usually important to get right the first time because if you don't, you and your users will have to live with your mistakes for a long time.

This section is not a a full-blown guide to API design as there's a lot of literature, courses and presentations available on the subject - see e.g. Designing a library that's easy to use - but we will mention the most important principles and a couple of examples of good and bad API design.

The main goals when it comes to API design is, of course, to make the API easy to use - this include choosing good names for types, functions and constants. Be careful of abbreviations - atof might be quick to type but it's not exactly clear that the function parses a C string and returns a double (no, not a float as the name suggests). Typically nouns are used for types and while verbs are used for methods.

Another thing to keep in mind is the number of function arguments - ideally each function should take only a few arguments so it's easy to remember how to use it. For example, no-one probably ever remembers exactly what arguments to pass to g_spawn_async_with_pipes() so programmers end up looking up the docs, breaking the rhythm. A better approach (which is yet to be implemented in GLib), would be to create a new type, let's call it GProcess, with methods to set what you'd otherwise pass as arguments and then a method to spawn the actual program. Not only is this easier to use, it is also extensible as adding a method to a type doesn't break API while adding an argument to an existing function/method does. An example of such an API is libudev's udev_enumerate API - for example, at the time udev starting dealing with device tags, the udev_enumerate type gained the add_match_tag() method.

If using constants, it is often useful to use the C enum type since

There was an error in this gadget

There was an error in this gadget

Blog Archive

About Me

davidz
Boston, MA, United States

Software Engineer at Red Hat, Inc.

View my complete profile

the compiler can warn if a switch statement isn't handling all cases. Generally avoid boolean types in functions and use flag enumerations instead - this has two advantages: first of all, it's sometimes easier to read foo_do_stuff(foo, FOO_FLAGS_FROBNICATOR) than foo_do_stuff(foo, TRUE) since the reader does not have to expend mental energy on remembering if TRUE translates into whether the frobnicator is to be used or not. Second, it means that several booleans arguments can be passed in one parameter so hard-to-use functions like e.g. gtk_box_pack_start() can be avoided (most programmers can't remember if the *expand* or *fill* boolean comes first). Additionally, this technique allows adding new flags without breaking API.

Often the compiler can help - for example, C functions can be annotated with all kinds of gcc-specific annotations that will cause warnings if the user is not using the function correctly. If using, GLib, some of these annotations are available as macros prefixed with G_GNUC, the most important ones being G_GNUC_CONST, G_GNUC_PURE, G_GNUC_MALLOC, G_GNUC_DEPRECATED_FOR, G_GNUC_PRINTF and G_GNUC_NULL_TERMINATED.

Checklist

- Choose good type and function names (favor expressiveness over length).
- Keep the number of arguments to functions down (consider introducing helper types).
- Use the type system / compiler to your advantage instead of fighting it (enums, flags, compiler annotations).

Documentation

If your library is very simple, the best documentation might just be a nicely formatted C header file with inline comments. Often it's not that simple and people using your library might expect richer and cross-referenced documentation complete with code samples.

Many C libraries, including those in GLib and GNOME itself, use inline documentation tags that can be read by tools such as gtk-doc or Doxygen. Note that gtk-doc works just fine even on low-level non-GLib-using libraries - see e.g. libudev and libblkid API documentation.

If used with a GLib library, gtk-doc uses the GLib type system to draw type hierarchies and show type-specific things like properties and signals. gtk-doc can also easily integrate with any tool producing Docbook documentation such as manual pages or e.g. gdbus-codegen(1) when used to generate docs describing D-Bus interfaces (example with C API docs, D-Bus docs and man pages).

Checklist

- Decide what level of documentation is needed (HTML, pdf, man pages, etc.).
- Try to use standard tools such as Doxygen or gtk-doc.

If shipping commands/daemons/helpers (e.g. anything showing up in ps(1) output), consider shipping man pages for those as well.

Language bindings

C libraries are increasingly used from higher-level languages such as Python or JavaScript through a so-called language binding - for example, this is what allows the Desktop Shell in GNOME 3 to be written entirely in JavaScript while still using C libraries such as GLib, Clutter and Mutter underneath.

It's outside the scope of this article to go into detail on language bindings (however a lot of the advice given in this series does apply - see also: Writing Bindable APIs) but it's worth pointing out that the goal of the GObject Introspection project (which is what is used in GNOME's Shell) is aiming for 100% coverage of GLib libraries assuming the library is properly annotated. For example, this applies to the GUdev library (a thin wrapper on top of the libudev library) can be used from any language that supports GObject Introspection (JS example).

GObject Intropspection is interesting because if someone adds GObject Introspection support to a new language, X, then the GNOME platform (and a lot of the underlying Linux plumbing as well cf. GUdev) is now suddenly available from that language without any work.

Checklist

Make sure your API is easily bindable (avoid C-isms such as variadic functions).

If using GLib, set up GObject Introspection and ship GIR/typelibs (notes).

If writing a complicated application, consider writing parts of it in C and parts of it in a higher-level language.

ABI, API and versioning

While the API of a library describes how the programmer use it, the ABI describes how the API is mapped onto the target machine the library is running on. Roughly, a (shared) library is said to be compatible with a previous version if a recompile is not needed. The ABI involves a lot of factors including data alignment rules, calling conventions, file formats and other things that are not suitable to cover in this series; the important thing to know about when writing C libraries is how (and if) the ABI changes when the API changes. Specifically, since some changes (such as adding a new function) are backwards compatible, the interesting question is what kind of API changes result in non-backwards-compatible ABI changes.

Assuming all other factors like calling convention are constant, the rule of thumb about compatibility on the ABI level basically boils down to a very short list of allowed API changes:

- you may add new C functions; and
- you may add parameters to a function only if it doesn't cause a memory/resource leak; and
- you may add a return value to a function returning void only if it doesn't cause a memory leak; and
- modifiers such as const may be added / removed at will since they are not part of the ABI in C

The latter is an example of a change that breaks the API (causing compiler warnings when compiling existing programs that used to compile without warnings) but preserve the ABI (still allowing any previously compiled program to run) - see e.g. this GLib commit for a concrete example (note that this can't be done in C++ because of how name mangling work).

In general, you may not extend C structs that the user can allocate on the stack or embed in another C structure which is why opaque data types are often used since they can be extended without the user knowing. In case the data type is not opaque, an often used technique is to add padding to structs (example) and use it when adding a new virtual method or signal function pointer (example). Other types, such as enumeration types, may normally be extended with new constants but existing constants may not be changed unless explicitly allowed.

The semantics of a function, e.g. its side effect, is usually considered part of the ABI. For example, if the purpose of a function is to print diagnostics on standard output and it stops doing it in a later version of the library, one could argue it's an ABI break even when existing programs are able to call the function and return to the caller just fine possibly even returning the same value.

On Linux, shared libraries (similar to DLLs on Windows) use the so-called soname to maintain and provide backwards-compatibility as well as allowing having multiple incompatible run-time versions installed at the same time. The latter is achieved by increasing the major version number of a library every time a backwards-incompatible change is made. Additionally, other fields of the soname have other (complex) rules associated (more info).

One solution to managing non-backwards-compatible ABI changes without bumping the so-number is symbol versioning - however, apart from being hard to use, it only applies to functions and not e.g. higher-level run-time data structures like e.g. signals, properties and types registered with the GLib type-system.

It is often desirable to have multiple incompatible versions of libraries and their associated development tools installed at the same time (and in the same prefix) - for example, both version 2 and 3 of GTK+. To easily achieve this, many libraries (including GLib and up) include the major version number (which is what is bumped exactly when non-backwards-compatible changes are made) in the library name as well as names of tools and so on - see the Parallel Installation essay for more information.

Some libraries, especially when they are in their early stages of development, specifically gives no ABI guarantees (and thus, does not manage their soname when incompatible changes are made). Often, to better manage expectations, such unstable libraries require that the user defines a macro acknowledging this (example). Once the library is baked, this requirement is then removed and normal ABI stability rules starts applying (example).

Related to versioning, it's important to mention that in order for your library to be easy to use, it is absolutely crucial that it includes pkg-config files along with the header files and other development files (more information).

Checklist

Decide what ABI guarantees to give if any (and when)

Make sure your users understand the ABI guarantees (being explicit is good)

If possible, make it possible to have multiple incompatible versions of your library and tools installed at the same time (e.g. include the major version number in the library name)

Posted by davidz at 3:31 PM

Labels: C, GLib, Programming

6 comments:

fredcadete July 6, 2011 at 4:03 AM

This is a GREAT series of articles. I especially like the many very useful links. Go try and write something like this on a dead-trees book.

Thanks a lot!

Reply

Chris July 6, 2011 at 8:16 PM

How does adding new parameters to an existing function NOT break the function's ABI?

1. If a v2 caller passes unexpected parameters to a v1 library, then the library will safely ignore them. No problem.

2. But if a v1 caller passes TOO FEW parameters to a v2 library, then the library will read garbage values for the new parameters (unless the v2 library is deduce if the new parameters are missing based on the values of the old parameters).

Reply

davecb July 6, 2011 at 9:21 PM

An interesting and somewhat elegant variant of a parallel installation with symbol versioning dates back to Multics, and some early linkers.

This versions the interface or a struct passed via it, and allows asynchronous change: the client and server need not change at the same time.

This is described in Paul Stachour's article in CACM, at http://cacm.acm.org/magazines/2009/11/48444-you-dont-know-jack-about-software-maintenance/fulltext

--dave (who helped on the article) c-b

Reply

paul_moloney July 8, 2011 at 5:28 PM

Structure versioning guidelines in glibc
http://sourceware.org/glibc/wiki/Development/Versioning_A_Structure

Reply

Peter Clay January 25, 2013 at 11:33 AM

Very good overview of libraries. I like the links to outside resources, very handy!

Reply

The Geeks February 24, 2014 at 11:06 PM

hi..Im college student, thanks for sharing :) inspire..!!!

Reply

Enter your comment...

Comment as: Unknown (Goog Sign out

Publish Preview Notify me

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)