25 minutes allocated. 20 minutes for talk, 5 minutes for questions.
Hello! In this talk I'm going to cover some changes we've been making to the threading model in gnome-software over the last couple of cycles. Work on this has been done by Endless and Red Hat. The aim is both to provide a bit of a progress update for those who are interested in gnome-software, and to provide some more general ideas about what works, and what doesn't work so well, when using threads in a program. I'll also briefly cover some thoughts about trying to minimise pain while landing big changes to a project like this.

History of gnome-software

- Project started in 2012
- Plugins added in 2013
- Architecture has always been entirely plugin based
- AppStream support from 2013

The gnome-software project started ten years ago. I wasn't around then, so please correct me if you were there any I get any of this wrong! Right from the beginning, the architecture has been plugin based. I believe this was because the idea was to have one or more plugins per distribution, so that the UI could be shared between distros but it could still function with different package managers. A lot of the original API design was done with traditional package managers in mind, as OSTree was only a year old at the time, and it would be another 2 years before flatpak was started.

History of gnome-software

- Project started in 2012
- Plugins added in 2013
- Architecture has always been entirely plugin based
- AppStream support from 2013

More than just being plugin based, the architecture was entirely plugin based: gnome-software didn't do anything without its plugins. AppStream support, for example, was a plugin. Over time, the decision to standardise on AppStream turned out to be a better and better one, but the support for it in gnome-software remained inside a plugin just like everything else. The plugin is mandatory now.

History of gnome-software

- Project started in 2012
- Plugins added in 2013
- Architecture has always been entirely plugin based
- AppStream support from 2013

(AppStream is a format for providing information about software, such as its description, screenshots, installation requirements, package format, etc. It is distribution-agnostic. There's plenty of documentation available about it online: `https://www.freedesktop.org/software/appstream/docs/`)

Reworking threading in GNOME Software

Previous architecture

Previous architecture

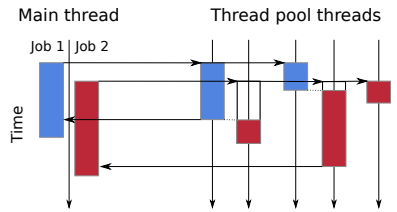

Figure: Previous architecture of gnome-software

In the old architecture of gnome-software, each job (such as refreshing metadata, getting more details for applications, or installing an app) calls a function in every plugin. All those functions are run in separate thread pool threads — so each job requests about 10 thread pool threads, one for each plugin.

Previous architecture

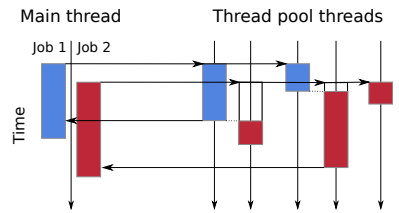

Figure: Previous architecture of gnome-software

Plugins are run in a fixed order, defined when gnome-software starts. This allows them to modify the results returned by earlier plugins. For example, some plugins, such as the flatpak plugin, will add apps to the list of results for the job. Later in the job, the icons plugin might go through the in-progress list of results and download an icon from the internet for each app, modifying the results list in-place. (This isn't shown in the diagram for reasons of space.)

Reworking threading in GNOME Software

Previous architecture

Each plugin function does its work synchronously (not yielding control of the thread until it's finished), and then returns. When all of the plugin functions have returned, the main thread marks the job as complete and returns the results to the caller.

Previous architecture

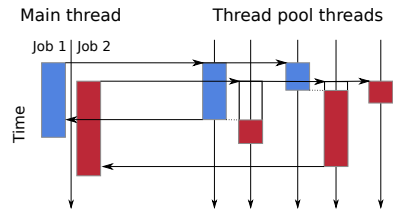

Figure: Previous architecture of gnome-software

Previous architecture

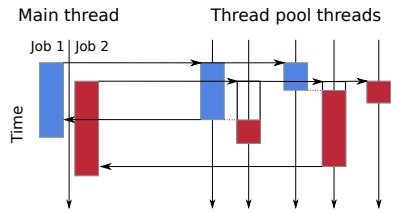

Figure: Previous architecture of gnome-software

The diagram shows this as a graph of time against threads. Two jobs are run, and cause 5 threads to be requested from the thread pool.

Reworking threading in GNOME Software

Previous architecture

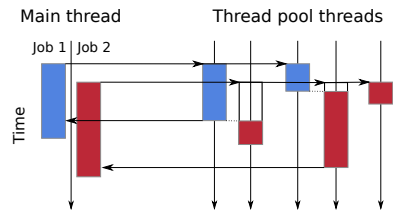

Figure: Previous architecture of gnome-software

I've not shown the sequential ordering of plugins in the diagram, as otherwise it would be too tall. However, I have shown serialisation of jobs in separate threads due to locking inside each plugin (as outline boxes). As functions can be called on each plugin from multiple thread pool threads, its internal data structures have to be locked when accessed. This serialises multiple jobs on the same plugin, delaying subsequent ones until the first has finished.

Issues

Issues

1. Limited number of threads in thread pool leads to exhaustion (hangs)
2. Large number of threads uses lots of resources (memory)
3. Locking required everywhere causes serialisation (slow)
4. Threading overhead is large for simple jobs (slow)

As the complexity of gnome-software, and the number of plugins, grew, it became apparent that the architecture didn't scale well enough. The most immediate problem was that the number of threads in the thread pool is limited, and it's possible to hit a deadlock if the limit is reached and all the pending threads are waiting on an operation which requires an additional thread pool thread to complete it. This can happen with OSTree and flatpak.

One solution to that would be to increase the thread pool size limit. But where do you stop? Every time you increase it, it's possible to hit the deadlock after a few more pending operations. The long-term fix is to decouple the number of threads from the number of pending operations.

Another problem with using a lot of threads is that they consume significant resources. While each thread theoretically shares the full address space with the rest of the process, glibc actually reserves 64MB of heap pages privately for each new thread (a per-thread arena `https://www.gnu.org/software/libc/manual/html_node/Memory-Allocation-Tunables.html`, `https://siddhesh.in/posts/malloc-per-thread-arenas-in-glibc.html`. It uses these to speed up heap allocations (`malloc()` calls) for that thread, by avoiding locking. This is a good optimisation, but means that the non-shareable memory use of the process increases by 64MB for each new thread pool thread. 10 thread pool threads is not uncommon with gnome-software, which leads to 640MB of additional non-shareable memory use. People complain about this, which is reasonable, although not much of this memory ever has to be physically backed.

Issues

1. Limited number of threads in thread pool leads to exhaustion (hangs)
2. Large number of threads uses lots of resources (memory)
3. Locking required everywhere causes serialisation (slow)
4. Threading overhead is large for simple jobs (slow)

The third and fourth problems relate to how work is scheduled on threads. As it turns out, a lot of the work each gnome-software plugin ends up doing is quite simple and doesn't require much CPU time. In particular, several plugins just talk to a daemon over D-Bus, and don't do any CPU intensive work themselves.

GDBus threading

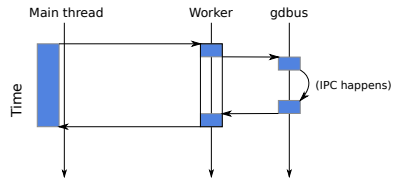

Figure: Threading when making a GDBus call

In order to do this, they use GDBus. Whenever you make a method call in GDBus, the calling thread puts together a D-Bus message packet, and then queues it to a GDBus worker thread provided by GLib. That worker thread handles communicating with the daemon, and processing replies from it. So when a gnome-software plugin was making a D-Bus call, a thread pool thread was being picked, sent a small message about the ongoing job, which was then doing a small amount of work to turn that into a D-Bus call, and sending that message on to the GDBus worker thread. Until the daemon and GDBus worker thread replied, the thread pool thread then slept and didn't do any other work.

GDBus threading

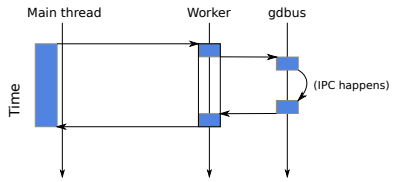

Figure: Threading when making a GDBus call

This occupied thread pool threads for a relatively long time (the time taken for the daemon to do whatever it was asked to do, plus the D-Bus round trip time of on the order of 1ms), while actually spending very little time using the CPU or other resources. When thread pool threads are a scant resource, this is not an efficient use of them.

Reworking threading in GNOME Software

Previous architecture

Recap of the old architecture.

Previous architecture

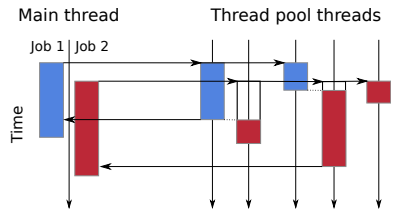

Figure: Previous architecture of gnome-software

Reworking threading in GNOME Software

New architecture

New architecture

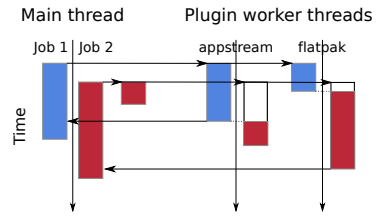

Figure: New architecture of gnome-software

So this brings us to the new architecture. The key change is that plugins are now in charge of their own threading model rather than being forced to use a thread pool thread for each operation.

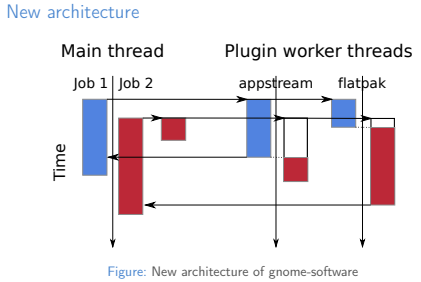

Figure: New architecture of gnome-software

This means that some plugins can work entirely in the main thread, without using any separate threads, as the work they do is light and doesn't block. You can see that with this short function call for job 2, now run in the main thread. Other plugins, which do need to do blocking operations, can explicitly run their own worker thread (or threads). This also allows them to set appropriate scheduling priorities for the thread. The `appstream` and `flatpak` plugins do this.

Reworking threading in GNOME Software

New architecture

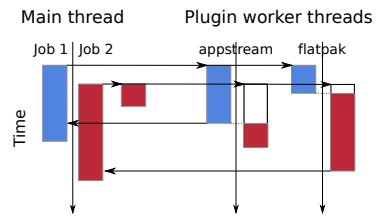

Figure: New architecture of gnome-software

By using a fixed number of threads per plugin, the problem of exhausting the thread pool is eliminated — the width of this graph is now bounded.

Reworking threading in GNOME Software

New architecture

New architecture

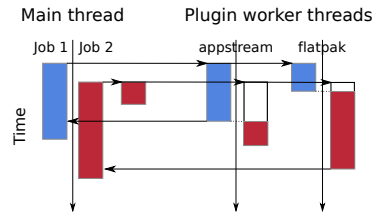

Figure: New architecture of gnome-software

The plugins which now operate without a thread no longer need to do any locking, which means that concurrent jobs on them won't be serialised and block each other. Locking and serialisation are still required for the plugins which have threads, though. That's unavoidable.

New architecture

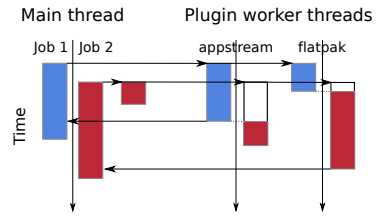

Figure: New architecture of gnome-software

Using fewer threads also means a reduction in the non-shared memory footprint of gnome-software. There is still more work to do in this area, though, as further memory reductions are possible through other changes to the code unrelated to threading.

New architecture

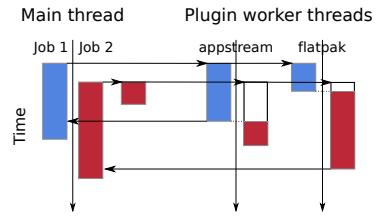

Figure: New architecture of gnome-software

The new way for plugins to make D-Bus calls is to just make them asychronously from the main thread. This means the D-Bus message is assembled in the main thread (which doesn't take long), enqueued to the GDBus worker thread (which always exists, provided by GLib), and then the main thread gets on with other work until there's a callback from the GDBus worker with a reply.

Reworking threading in GNOME Software

New architecture

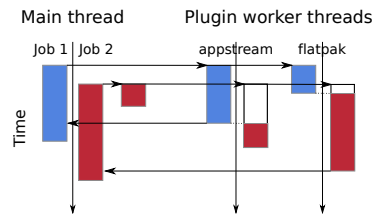

Figure: New architecture of gnome-software

As well as reducing the use of thread pool threads, this makes it very easy for a plugin to have multiple D-Bus calls in flight at once. Previously, the easiest way to code a plugin was to serialise D-Bus calls. Parallelising them may speed up some plugin operations (if the daemon the plugin is talking to can handle the calls faster in parallel). This is certainly the case in the PackageKit plugin.

New architecture

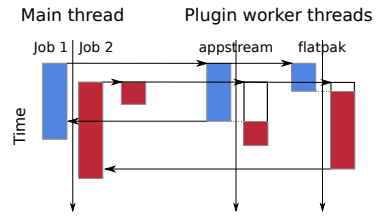

Figure: New architecture of gnome-software

Finally, changing the architecture has allowed us to split plugin jobs up into one class per job, rather than one generic job class used for all jobs. For example, there's now `GsPluginJobListApps` for querying for apps, and `GsPluginJobRefreshMetadata` for refreshing repo metadata. This makes the code a lot more type safe and hence understandable: rather than having to remember which job properties apply to each job, the API specifies it. It also allows expanding the API to expose, for example, job-specific and more detailed progress reporting in future.

Different approaches to threading in C

- Synchronous code always run in a worker thread
- Asynchronous code run somewhere
- Threading model determined at a high level vs locally

This is the bit of the talk which is going to be a bit more like a general software engineering textbook. What are the trade-offs when writing threading in C in the GNOME environment? The trade-offs with higher level languages or with other toolkits might be different, as they provide different primitives and language constructs which help you in different ways. Using a higher level or more modern language than C is the best way to go if you have the choice. But there are various places in GNOME where, due to the rest of the ecosystem being in C, it is pragmatic to still write new code in C. In these cases, you basically have the choice between writing synchronous code to run in a separate thread, or writing asynchronous code to run somewhere — in the main thread or in a separate thread.

Different approaches to threading in C

- Synchronous code always run in a worker thread
- Asynchronous code run somewhere
- Threading model determined at a high level vs locally

Writing synchronous code is easy, and it's easy to read sequentially. As a result, it's easier to maintain. Writing asynchronous code requires more boilerplate in C, and is thus a bit harder to maintain. The main benefit that asynchronous code provides, however, is that it defines the yield points in code: where the code needs to block on I/O or other synchronisation, and hence where it can yield control of the thread to something else for a while. It's not possible to define these in synchronous code in C — there is no `yield` keyword.

Different approaches to threading in C

- Synchronous code always run in a worker thread
- Asynchronous code run somewhere
- Threading model determined at a high level vs locally

The fact that its yield points are defined means that asynchronous code can be run in a worker thread or in the main thread fairly easily. It can be run synchronously using a trivial async-to-sync wrapper. For that reason, I think it's better to define all APIs as asynchronous, even if the initial implementation of them is written synchronously for ease of development. (Or: just use a higher level language.)

Different approaches to threading in C

- Synchronous code always run in a worker thread
- Asynchronous code run somewhere
- Threading model determined at a high level vs locally

gnome-software didn't do that: its plugin job API was synchronous, and hence plugins didn't have the choice of running in the main thread. They were all bound to always being run in thread pool threads. And while that allowed for fast development and linear code, it committed the code to high thread use. It's worth noting that a lot of these asynchronous coding patterns were refined since gnome-software was written, so gnome-software is basically just a victim of its age.

Different approaches to threading in C

- Synchronous code always run in a worker thread
- Asynchronous code run somewhere
- Threading model determined at a high level vs locally

Further reading about threads and, in particular, their interactions with main contexts, is here: `https://developer.gnome.org/ExtractShell/documentation/tutorials/threading.html`.

Approaches for landing big changesets

- Land early
- Keep things working
- Keep adapter wrappers around old code and drop it eventually

So how have we gone about landing these changes? From my point of view, development has worked very well. Others might give you a different perspective though. This series of changes to gnome-software is quite large — we're now up to about 30 merge requests. One thing that seems to have worked well is to land smallish chunks of work fairly often, and keep close to `main`.

Approaches for landing big changesets

Approaches for landing big changesets

- Land early
- Keep things working
- Keep adapter wrappers around old code and drop it eventually

By keeping merge requests small, they've been easier to review, and have had minimal merge conflicts. The code which has landed early has been tested in `main` for longer, and bugs which have been found in it have informed subsequent development.

The downside to this is that old code – which we're trying to replace – hangs around for longer and gets gnarlier: 30 merge requests into the refactor, the `GsPluginLoader` code has gained a load of new exceptions to interface it with the new code, and only a few parts of the old code have been dropped at this point. You don't get the pleasure of seeing a huge diffstat for all the changes.

Approaches for landing big changesets

Approaches for landing big changesets

- Land early
- Keep things working
- Keep adapter wrappers around old code and drop it eventually

The other advantage of landing smaller chunks of work often is that it allows the work to be split up, with multiple people contributing different parts of the refactor in parallel at times.

Approaches for landing big changesets

Approaches for landing big changesets

- Land early
- Keep things working
- Keep adapter wrappers around old code and drop it eventually

A big thank you to Milan Crha, Georges Neto and everyone else who's been involved in doing and reviewing the threading rework in gnome-software. It's very much been a collaborative project.