Rust Heap Profiling with Jemalloc

August 31 2023

by Marc-Andre Giroux

I recently had to investigate a memory leak in apollo/router, a federated GraphQL gateway written in Rust. The first question I had was how to get a memory profile out of our running instances. For several reasons I decided to use Jemalloc's Heap Profiling tooling to do so:

- apollo/router uses jemalloc on linux by default
- jemalloc 's has a relatively low performance overhead compared to other tools.
- ByteHound sounded amazing but could not get it to work with apollo/router , something about it using V8 under the hood.

Took me a bit to get all the pieces working, so hopefully this post saves someone some time.

Getting Heap Dumps from Jemalloc

To use jemalloc in Rust these days we must do it explicitely using the #[global_allocator] attribute. rustc used to link against jemalloc on some platforms, but that's not the case anymore.

```
# Cargo.toml
[dependencies]

[target.'cfg(not(target_env = "msvc"))'.dependencies]
tikv-jemallocator = "0.5"
```

Then in your main.rs , start using jemallocator as your #[global_allocator] :

```
// main.rs
#[cfg(not(target_env = "msvc"))]
use tikv_jemallocator::Jemalloc;
```

```
#[cfg(not(target_env = "msvc"))]
#[global_allocator]
static GLOBAL: Jemalloc = Jemalloc;
```

One more thing, `jemalloc` needs to be configured with `--enable-prof` for heap profiling to work. The `tikv-jemallocator` crate can do that in its build script by adding the `profiling` feature:

```
[target.'cfg(not(target_env = "msvc"))'.dependencies]
tikv-jemallocator = { version = "0.5", features = ["profiling"] }
```

Activating Heap Dumps

If you read jemalloc's docs, the heap profiles are configured through the `MALLOC_CONF` environment variable.

```
export MALLOC_CONF="prof:true,prof_prefix:jeprof.out"
```

For my specific use case, I was using this config:

```
export MALLOC_CONF=prof:true,lg_prof_interval:30,lg_prof_sample:21,p
```

- `prof:true` : enables profiling
- `lg_prof_interval` : 2MB interval between allocation samples, in bytes of allocation acitvity (2^21)
- `lg_prof_sample:21` : will dump a heap profile after 1GB allocations (2^30)
- `prof_prefix` : a prefix for the heap dump files.

I deployed all this and waited... and... **nothing**.

After a while I discovered that `tikv-jemallocator` configures `jemalloc` with a **prefix** (`--with-jemalloc-prefix=_rjem_`). This also influences the env var. The actual config should look like this:

```
export _RJEM_MALLOC_CONF=prof:true,lg_prof_interval:30,lg_prof_sampl
```

Analysis

Once that was deployed I started getting a bunch of heap dumps appearing in `/tmp/jeprof*`. To get those sweet SVGs I first got the `jeprof` utility on the machine, but also needed `dot` from `graphviz`:

```
$ apt install libjemalloc-dev graphviz
```

Once I got those on the machine I was profiling, I ran this command to generate a graph like this one:

```
$ jeprof --svg   > heapdump.svg
```

Dynamic Profiling

As you can see, getting those heapdumps requires running your executable with `prof:true`, which by default will start profiling right away. This definitely has an overhead, and we should probably not run this on all our instances serving real production traffic.

At the same time, it's annoying to have to start up a new executable to get a profile. There are situations where we'd love to get insights into a long running process, or getting profiles during very specific moments only.

Thankfully, jemalloc's got our back with the `prof.active` option:

```
  opt.prof_active (bool) r- [--enable-prof]

      Profiling activated/deactivated. This is a secondary control mec
      that makes it possible to start the application with profiling e
      (see the opt.prof option) but inactive, then toggle profiling at
      time during program execution with the prof.active mallctl.
      This option is enabled by default.
```

So the idea is that:

- `opt.prof` generally enables or disables profiling memory allocation activity
- `prof_active` activates or deactivates the profiling.

You may wonder if having `prof:true` but `prof_active:false` still has

a considerable overhead. I was wondering the same and found this insightful comment in an email thread from 2014:

> *Yes, you can use jemalloc's heap profiling as you describe, with essentially no performance impact while heap profiling is inactive. You may even be able to leave heap profiling active all the time with little performance impact, depending on how heavily your application uses malloc. At Facebook we leave heap profiling active all the time for a wide variety of server applications; there are only a couple of exceptions I'm aware of for which the performance impact is unacceptable (heavy malloc use, ~2% slowdown when heap profiling is active).*

So depending on your use-case, this may be totally fine.

mallctl

Ok, moving on, how do we enable/disable `prof_active` from our rust app? `jemalloc` gives us the API to control some of its options. The main thing we'll use here is the `mallctl()` function:

```
int mallctl(const char *name,
    void *oldp,
    size_t *oldlenp,
    void *newp,
    size_t newlen);
```

`mallctl` takes the option name as a parameter as a period-separated name, in our case, that's `prof.active`. We can use `mallctl` to both read and write to options.

- To **read**, we'll pass a pointer `oldp` to memory that will contain the value we're reading, and set `oldlenp` to a pointer to the length of `oldp`. We'll pass a null ptr and 0 to the rest of the arguments.
- To **write**, we'll pass null pointers to both `oldp` and `oldlenp`, a pointer to the new value in `newp`, and a pointer to its size in `newlen`.

In Rust, the `tikv-jemallocator-sys` crate already gives us the binding to `mallctl()`.

```
// cargo.toml
tikv-jemalloc-sys  = { version = "0.5", features = ["profiling"] }
```

We already had a dynamic config system that receives config updates through a tokio watch::Receiver . I decided to use this system to dynamically enable/disable jemalloc profiles. Something like this:

```
pub(crate) async fn profiling_task(mut config_updates: watch::Receiv
    while config_updates.changed().await.is_ok() {
        let new_conf = config_updates.borrow();

        // call mallctl with the new value!
        let result = set_prof_active(new_conf.prof_active);

        // ...
    }
```

And here's how we finally make the call to mallctl()

```
#[derive(Debug, Clone)]
struct MallctlError { code: i32 };

fn set_prof_active(new_value: bool) -> Result<(), MallctlError> {
    let option_name = CString::new("prof_active").unwrap();

    let result = unsafe {
        tikv_jemalloc_sys::mallctl(
            option_name.as_ptr(), // const char *name
            str::ptr::null_mut(), // void *oldp
            str::ptr::null_mut(), // size_t *oldlenp
            &new_value as *const _ as *mut, // void *newp
            std::mem::size_of_val(&new_value) // size_t newlen
        )
    }

    if result != 0 {
        return Err(MallctlError { code: result });
    }

    Ok(())
```

```
}
```

Fun Debugging Story

Locally at first this code always resulted in a `MallctlError`. The error code was `2`, which is `ENOENT`. According to the docs that means:

> *name or mib specifies an unknown/invalid value.*

I was pretty sure I had the name right (`prof.active`). I used `lldb` to debug the `mallctl` call and eventually tracked it down to this:

```
Process 89566 stopped
* thread #12, name = 'tokio-runtime-worker', stop reason = step over
    frame #0: 0x0000000100dc9084 router`prof_active_ctl(tsd=0x000000
   3224                 bool val = *(bool *)newp;
   3225                 if (!opt_prof) {
   3226                         if (val) {
-> 3227                                 ret = ENOENT;
   3228                                 goto label_return;
   3229                         } else {
   3230                                 /* No change needed (already
Target 0: (router) stopped.
(lldb)
```

If the `prof` option is disabled and we try to set `prof.active`, it results in `ENOENT`. I guess that would be considered an "invalid value" in that case. This is actually indicated in the docs. Locally I wasn't running with `prof:true` initially, which caused this issue!

There you have it, now we have on-demand `jemalloc` heap profiling! Hope this helps. If you're curious about that particular instance of the memory leak, you can find the investigation here, and the fix here. TL;DR: careful with `Arc`!

- xuorig