15 Jan, 2022

Investigating Memory Allocations in Rust

I recently stumbled upon a situation where I needed to parse strings containing arithmetic expressions such as `"2 + 3"`. The first thing I've done, of course, was to check crates.io for "arithmetic eval", "expression eval", etc. I found some crates like eval, evalexpr, and meval but they all do more than what I need. I generally like to avoid overhead and introducing too many or too big dependencies if I don't absolutely have to, but in this case I thought it would be really nice to just have a simple, minimal crate that was capable of doing this kind of simple arithmetic evaluation (containing only numbers, the 4 basic operations, and parentheses). I also found other crates that were more aligned to this idea, such as pmalmgren/rust-calculator and adriaN/simple_rust_parser, but they are binaries instead of libraries. Long story short, the Rust ecosystem seems to lack a dominant crate for parsing very simple arithmetic expressions as fast as possible. So I decided to start my own.

The problem we have at hand should have an `O(n)` complexity (i.e. linear). In one pass we can tokenise the input, and with one more pass we can process the list of tokens (which has size at most `n`). So this is a problem we should be able to solve very fast. At this point it becomes clear that the performance bottleneck of this library would not be in the loops and comparisons of the parser, but in the much slower memory allocations used to implement this. That is why my goal for mexe is to minimise allocations. You see, we know that the list of tokens has size at most `n`, but if we allow the user to supply large arithmetic expressions, this `n` can be very large and it could be a problem (and a waste) to pre-allocate a fixed space for this on the stack (such as using a fixed size array). Not to mention that that would create an arbitrary limitation on the size of inputs that we accept. Here is where allocations come into play: when we want things to have a size that is unknown at compile-time, we usually have to allocate space for those things at runtime, which is what the heap is for.

Tracing Memory Allocation Syscalls

Before we can think of minimising memory allocations though, it is good to have some way to know how this works in Rust. In Linux, as far as I know, there is no way to obtain memory without (usually indirectly) invoking the system calls `brk`, `mmap` and `mmap2` (something I learned when making an assembly tutorial for my abandoned compiler project). We can trace communications (e.g. via syscalls) between programs and the kernel with the Linux utility `strace`.

So, first, let's find out how many memory allocation related syscalls an empty Rust program (just `fn main() {}`) does. I built it in release and then ran `strace` on it:

```
$ strace -f -e brk,mmap,mmap2 -- ./target/release/alloc-test
brk(NULL)                               = 0x5592b7a70000
mmap(NULL, 142634, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f8bc9777000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f8bc
.
.
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0
brk(NULL)                               = 0x5592b7a70000
brk(0x5592b7a91000)                     = 0x5592b7a91000
+++ exited with 0 +++
```

I omitted some lines in the output above because it was a bit large. Using some `sed` magic, we can just count the allocations:

```
$ strace -f -e brk,mmap,mmap2 -- ./target/release/alloc-test 2>&1 | grep 'brk\|mma
  3 brk
 19 mmap
```

Hmm, 22 memory allocation syscalls for a program that does nothing. What about C? We can write `int main() { return 0; }` in a file `success.c`, build with `gcc success.c -o success_c`, then run `strace`:

```
$ strace -f -e brk,mmap,mmap2 -- ./success_c 2>&1 | grep 'brk\|mmap' | sed -e 's/^
  1 brk
  5 mmap
```

Not surprisingly, C is a bit more economic. What *is* surprising for me

is that C (which has "no runtime") makes these 6 calls. Are these system calls inevitable whenever we execute any program? Do they happen simply because the kernel is loading a program in memory? To verify, let's go even lower and write a "do-nothing" program in assembly and see what happens. The code below (`success.asm`) just returns a 0 ("success"):

```
global _start      ; make the symbol `_start` visible from outside

section .text      ; here begins the code

_start:
  mov rax, 60      ; exit(
  mov rdi, 0       ;   EXIT_SUCCESS
  syscall          ; );
```

We can assemble and link this with:

```
$ nasm -f elf64 -o success_asm.o success.asm && ld -o success_asm success_asm.o
$ chmod +x success_asm
```

Now to `strace` again:

```
$ strace -f -e brk,mmap,mmap2 -- ./success_asm
+++ exited with 0 +++
```

Okay, so executing the assembly program does not make any memory syscalls by default. Our C program clearly does not ask for any allocations, so I will assume that these allocations come from the C runtime. It is worth noting that our C executable has some dynamically linked dependencies:

```
$ ldd success_c
        linux-vdso.so.1 (0x00007ffce8c82000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f8972d4e000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f8973341000)
```

As does the Rust program:

```
$ ldd target/release/alloc-test
        linux-vdso.so.1 (0x00007ffd7ca8b000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f1e96cb0000)
        librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1 (0x00007f1e96aa8000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f1e968890
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f1e964eb000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f1e962e7000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f1e95ef6000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f1e97110000)
```

But here there is a difference when looking at the assembly program:

```
$ ldd success_asm
            not a dynamic executable
```

And if we use `file` on our executables:

```
$ file success_asm
success_asm: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linke

$ file success_c
success_c: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically lin

$ file target/release/alloc-test
target/release/alloc-test: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
```

The assembly program is considered an LSB executable, but the Rust and C programs are LSB “shared objects”, which is a term in Linux used for libraries. Not only that, we also find that the C and Rust programs are *interpreted*. This is a topic I definitely want to explore in a future post, but let’s leave this aside for now.

The hidden allocations might be coming either from these dynamic dependencies or from the “runtime”, which is a bunch of code that is inserted in your binary by default. It is the reason why the Rust do-nothing release binary has 3.5MB in size (actually most of it might be due to object symbols metadata). Symbols are names of variables and functions that are used in the binary (but may be defined somewhere else), and a non-stripped binary may contain metadata

about those symbols. They can be seen with the `nm` utility. For example our C binary has a few of them:

```
$ nm success_c
0000000000201010 B __bss_start
.
.
00000000000005fa T main
0000000000000560 t register_tm_clones
00000000000004f0 T _start
0000000000201010 D __TMC_END__
```

(I deleted most lines of the output.) In total the binary has 32 symbols. Our Rust binary in comparison has 677:

```
$ nm target/release/alloc-test | wc -l
677
```

Make Binary Small Again

Let's follow this nice guide on how to reduce the size of our Rust binary and see what's the actual code size.

```
$ strip target/release/alloc-test
```

Woah, okay, so this simple `strip` reduced our binary to only 287KB. That is still massive compared to our C binary though, which only has 8168 bytes. After stripping, the C binary goes to 6056 bytes, almost 50 times smaller! Anyways, with the following changes to `Cargo.toml` (and without resorting to nightly) we can reduce the binary size to 239KB:

```
[profile.release]
opt-level = "z"
lto = true
codegen-units = 1
panic = "abort"
```

The funny thing is that, after this, we not only get a leaner binary but

also fewer memory allocation syscalls:

```
$ strace -f -e brk,mmap,mmap2 -- ./target/release/alloc-test 2>&1 | grep 'brk\|mma
  3 brk
 14 mmap
```

I was going to guess that these syscalls were related to pre-allocating memory for panic, but removing the `panic = "abort"` did not change those numbers.

We want to understand where these 17 allocations are coming from, and if they are not in these 240KB they have to be in the dependencies we found with `ldd`. I suspect the number of allocations could also be machine-dependent.

A Failed Attempt with `ltrace`

Now so far we have only looked at memory allocation on the OS level, through system calls. But many programming languages don't allocate memory by directly invoking `brk` or `mmap`. In Linux they usually delegate this job to `libc`. Rust used to statically link `jemalloc` in all binaries until version 1.32, when it started to use the default system allocator, which is `glibc`'s `malloc` in Linux. And indeed, we saw earlier that our Rust binary links to `glibc`. If we do `nm target/release/alloc-test | grep -i malloc` we can see that indeed there is a symbol for `malloc` coming from `glibc`, with type `U`, which stands for undefined – in this case because it is defined outside of our binary, namely in `libc`. And if we `grep` for `glibc` we see that there are also a bunch of other symbols coming from that library that are dynamically linked to our binary.

To my surprise, running `ltrace` (similar to `strace`, but for library calls) on our Rust program shows that there are no external calls:

```
$ ltrace target/release/alloc-test
+++ exited (status 0) +++
```

So, are those 17 memory allocations coming from our own binary? Inspecting it with `objdump` we can see that there are some interrupts, but checking this website they don't seem to be system calls (which would be either the instruction `int 0x80` or `syscall`):

```
$ objdump -D -Mintel target/release/alloc-test | grep 'int\|sys'
.
.
   3f765:   cd a0                 int    0xa0
   40305:   cc                    int3
   40c24:   cd ab                 int    0xab
   4224c:   cc                    int3
   429e1:   cc                    int3
   43dfd:   cd fd                 int    0xfd
.
.
```

Let's try to force some memory allocations to see if we can get some `malloc`s:

```
fn main() {
    let size = 50 * 1024 * 1024; // 8 (bytes in usize) * 50M ~ 420MB
    let mut x: Vec<usize> = Vec::new();

    for i in 0..size {
        x.push(i);
    }
    println!("{:?}", x.iter().sum::<usize>());
}
```

Then `ltrace`:

```
$ ltrace ./target/release/alloc-test
1374389508505600
+++ exited (status 0) +++
```

Still nothing. What is happening here?

With my current toolbelt I am getting out of ideas here. My last resort will be to check Rust's sources on Box to see if we can find out what exactly is done during allocations. Right off the bat, I learn a new Rust keyword: `box`:

```
impl<T> Box<T> {
    /// Allocates memory on the heap and then places `x` into it.
```

```
.
.
pub fn new(x: T) -> Self {
    box x
}
```

It seems that this `box` keyword is unstable, but has more or less the expected functionality of allocating memory on the heap. The problem is that it is implemented internally by the compiler, so we reached a dead-end here (it would be interesting to take a look inside `rustc` sometime, though).

`gdb` to our Rescue

Let's go ahead and try to see more or less what is happening with the help of `gdb`. Since we saw earlier that there is a symbol called `malloc` in our Rust binary, we can build in debug mode, run `gdb target/debug/alloc-test` and put a breakpoint on it with the command `b malloc`:

```
(gdb) b malloc
Function "malloc" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (malloc) pending.
```

The function will only be loaded dynamically, so for now `gdb` reports it as undefined. Now we use the command `r` (run). The program stops indeed at `malloc`. If we now use the command `bt` (backtrace), we can see where the invocation of `malloc` came from:

```
Breakpoint 1, malloc (n=1425) at dl-minimal.c:50
.
.
#7  0x00007ffff7dd4098 in _start () from /lib64/ld-linux-x86-64.so.2
```

This is the Linux dynamic linker/loader, which is responsible for loading libraries dynamically and linking them to the user program – and which is also the *interpreter* of our programs according to `file`. What this means, I think, is that similarly to when you put a `#!/bin/sh` in the beginning of a script to tell the shell that your script

should be interpreted by `sh`, this metadata tells the kernel to invoke `ld-linux` with our program instead of simply "executing it directly" (whatever that means) like our non-dynamically linked assembly program. A quick online search for the file `dl-minimal.c` and we can see in the first lines:

```
/* Minimal replacements for basic facilities used in the dynamic linker.
   Copyright (C) 1995-2016 Free Software Foundation, Inc.
   This file is part of the GNU C Library.
```

If we type `c` (continue), the program will continue running until it stops at the breakpoint again. I repeated this process tens of times and the result was similar, stopping at `dl-minimal.c` which we can assume is the dynamic loader loading the libraries we saw with `ldd`. So that might explain the mysterious 17 `brk`/`mmap` syscalls that we saw earlier, but as expected `malloc` was called way more times than the syscalls. That is because `malloc` don't need to ask the kernel for memory every single time the user makes an allocation – it just gets a larger buffer that it manages for the user program as it allocates. New syscalls will only be made if `malloc` needs more space (for example if it does not have any space large enough to satisfy what the user program requested).

So after perhaps 40 or so repetitions, `gdb` stops at `malloc` but the backtrace shows a different source:

```
Continuing.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, __GI___libc_malloc (bytes=552) at malloc.c:3038
.
.
#6  std::sys::unix::thread::guard::init () at library/std/src/sys/unix/thread.rs:4
#7  std::rt::init () at library/std/src/rt.rs:80
#8  0x0000555555570c04 in std::rt::lang_start_internal:: () at library/std/src/rt.
#9  std::panicking::try::do_call () at library/std/src/panicking.rs:403
#10 std::panicking::try () at library/std/src/panicking.rs:367
#11 std::panic::catch_unwind () at library/std/src/panic.rs:133
#12 std::rt::lang_start_internal () at library/std/src/rt.rs:127
#13 0x000055555555bc32 in main () at library/std/src/panicking.rs:543
```

I check `rustc --version`, in this case `rustc 1.57.0 (f1edd0429 2021-11-29)`, so if we go to that commit, in line 543 of `library/std/src/panicking.rs` we might see exactly what kind of panic-related thing is allocating memory. At first I land at `begin_panic`, which seems strange (are we panicking?). But looking at the calls higher on the backtrace we see `rt.rs`. This module sets up the Rust runtime. In the code for `lang_start_internal` we see the comment:

```
// SAFETY: Only called once during runtime initialization.
```

So this is indeed initialization code and panic setup as we suspected. In summary, there were 43 breakpoint stops at `dl-minimal.c` (dynamic library loading) and 8 stops at `malloc.c` (panicking setup). We can get more clues by changing our program a bit:

```
fn main() {
    actual_code();
}

#[inline(never)]
fn actual_code() {
    let size = 50 * 1024 * 1024;
    let mut x: Vec<usize> = Vec::new();

    for i in 0..size {
        x.push(i);
    }
    println!("{:?}", x.iter().sum::<usize>());
}
```

We use `#[inline(never)]` because this function was being inlined by the compiler (I could not set a breakpoint at `actual_code` and `nm` did not show any symbol for it in the binary). Now we can set breakpoints at this new function and at `malloc`. When we stop at the breakpoint at `actual_code` we can use the command `f` to execute until the end of the function. When we do that, we see that indeed during the execution of the actual code there were no stops at `malloc`. In the end, 6 calls to `malloc` happened before the call `actual_code();` and 2 after. We can safely assume that this is initialization and termination code that happens before and after `main`, and it is all related to panic setup.

Going back to the syscalls, there is a way to stop at them using `gdb` with the commands `catch syscall brk` and `catch syscall mmap`. These work as breakpoints, so we can use `bt` when we stop at them to see the backtrace. Again we see that they are invoked from the dynamic loader and from the panic setup steps. If you are curious, the 8 calls to `malloc` tracing back to `panicking.rs` ask for 552, 120, 1024, 32, 5, 48, 32, and 1024 bytes, respectively, for a total of 2837 bytes.

Making `ltrace` Work

But I still don't get why we don't see these calls via `ltrace`. It definitely works, for example with this C code:

```
#include <stdlib.h>

int main() {
    malloc(1); // allocates enough for 1 byte
    return 0;
}
```

Compile and run with `ltrace`:

```
$ gcc alloc.c -o alloc_c
$ ltrace ./alloc_c
malloc(1)                                                = 0x556b3d00c260
+++ exited (status 0) +++
```

At this point I was almost giving up on writing this post, but after a lot of pain and frustration (and web searching), I found that I needed to specify the `-x` CLI option:

```
$ ltrace -x "malloc" ./target/release/alloc-test
malloc@libc.so.6(552)                                    = 0x55948e7f5260
malloc@libc.so.6(120)                                    = 0x55948e7f5490
malloc@libc.so.6(1024)                                   = 0x55948e7f5510
malloc@libc.so.6(32)                                     = 0x55948e7f5a20
malloc@libc.so.6(5)                                      = 0x55948e7f5a50
malloc@libc.so.6(48)                                     = 0x55948e7f5a70
malloc@libc.so.6(32)                                     = 0x55948e7f5a20
malloc@libc.so.6(1024)                                   = 0x55948e7f5510
1374389508505600
```

```
+++ exited (status 0) +++
```

Finally! Interestingly, these are the same calls I mentioned above (look at the arguments, i.e. the number of bytes requested). Going back to the empty Rust program (`fn main() {}`), the output of `ltrace` is almost the same as above, except for the program output (the big sum) and the last 2 `malloc`s. We saw with `gdb` that in reality our program triggered about 50 `malloc`s, but `ltrace` only reports 6 of them. Maybe that's because the others were done by the dynamic loader so that "doesn't count", I guess. However, even the C program has a different output now:

```
$ ltrace -x "malloc+free@*" ./alloc_c
malloc(1 <unfinished ...>
malloc@libc.so.6(1)                                      = 0x55e35f795260
<... malloc resumed> )                                   = 0x55e35f795260
+++ exited (status 0) +++
```

Back to What is Relevant

So, this might have bored the readers a lot, but bear with me. All this was pretty irrelevant to the goals I stated in the beginning: understand allocations to make my parser fast. Everything so far was about allocations that happen *regardless* of what we do. So this knowledge won't help us to write better Rust code. But for the attentive reader, our futile journey already had some interesting clues on memory allocation *during* execution. For example, did you notice that our program that allocates a large `Vec` only made 2 calls to `malloc` (for a grand total of measly 1056 bytes) during actual execution? And note that we made a point of not being efficient by not doing `Vec::with_capacity(5 * 1024 * 1024)`. That would have allocated all necessary memory at once. Usually list types like `Vec` start with a small capacity (a memory buffer in the heap) and is reallocated whenever this capacity is not enough. So I expected that this program would make lots of allocations by starting `Vec` with a capacity of say 4 elements or so which would double when we tried to `push` the 5th element, and so on until all the 50 million elements are pushed.

Did `rustc` and/or `llvm` somehow optimise our program enough that it only needed to allocate 1056 bytes instead of 420MB? Indeed, if you

think about what this program is doing, we don't even need a `Vec`. But that is not the explanation. We have been looking so far at `malloc`, but reallocation is done with `realloc`:

```
$ ltrace -x "malloc+realloc" ./target/release/alloc-test
malloc@libc.so.6(552)                                           = 0x5576e0844260
malloc@libc.so.6(120)                                           = 0x5576e0844490
malloc@libc.so.6(1024)                                          = 0x5576e0844510
realloc@libc.so.6(0x5576e0844490, 240)                          = 0x5576e0844920
realloc@libc.so.6(0, 32 <unfinished ...>
malloc@libc.so.6(32)                                            = 0x5576e0844a20
<... realloc resumed> )                                         = 0x5576e0844a20
malloc@libc.so.6(5)                                             = 0x5576e0844a50
malloc@libc.so.6(48)                                            = 0x5576e0844a70
malloc@libc.so.6(32)                                            = 0x5576e0844a20
realloc@libc.so.6(0x5576e0844a20, 64)                           = 0x5576e0844ae0
realloc@libc.so.6(0x5576e0844ae0, 128)                          = 0x5576e0844ae0
realloc@libc.so.6(0x5576e0844ae0, 256)                          = 0x5576e0844ae0
realloc@libc.so.6(0x5576e0844ae0, 512)                          = 0x5576e0844ae0
realloc@libc.so.6(0x5576e0844ae0, 1024)                         = 0x5576e0844ae0
realloc@libc.so.6(0x5576e0844ae0, 2048)                         = 0x5576e0844ae0
realloc@libc.so.6(0x5576e0844ae0, 4096)                         = 0x5576e0844ae0
realloc@libc.so.6(0x5576e0844ae0, 8192)                         = 0x5576e0844ae0
realloc@libc.so.6(0x5576e0844ae0, 16384)                        = 0x5576e0844ae0
realloc@libc.so.6(0x5576e0844ae0, 32768)                        = 0x5576e0844ae0
realloc@libc.so.6(0x5576e0844ae0, 65536)                        = 0x5576e0844ae0
realloc@libc.so.6(0x5576e0844ae0, 131072)                       = 0x5576e0844ae0
realloc@libc.so.6(0x5576e0844ae0, 262144)                       = 0x7fee0c62d010
realloc@libc.so.6(0x7fee0c62d010, 524288)                       = 0x7fee0c5ac010
realloc@libc.so.6(0x7fee0c5ac010, 1048576)                      = 0x7fee0c4ab010
realloc@libc.so.6(0x7fee0c4ab010, 2097152)                      = 0x7fee0b83f010
realloc@libc.so.6(0x7fee0b83f010, 4194304)                      = 0x7fee0b43e010
realloc@libc.so.6(0x7fee0b43e010, 8388608)                      = 0x7fee0ac3d010
realloc@libc.so.6(0x7fee0ac3d010, 16777216)                     = 0x7fee09c3c010
realloc@libc.so.6(0x7fee09c3c010, 33554432)                     = 0x7fee07c3b010
realloc@libc.so.6(0x7fee07c3b010, 67108864)                     = 0x7fee03c3a010
realloc@libc.so.6(0x7fee03c3a010, 134217728)                    = 0x7fedfbc39010
realloc@libc.so.6(0x7fedfbc39010, 268435456)                    = 0x7fedebc38010
realloc@libc.so.6(0x7fedebc38010, 536870912)                    = 0x7fedcbc37010
malloc@libc.so.6(1024)                                          = 0x5576e0844510
1374389508505600
+++ exited (status 0) +++
```

No compiler magic. But now we can nicely see what I just explained about `Vec`. It indeed start with a small size, and doubles when

necessary. Also, notice the behaviour of `malloc` (the allocator, not the function): it gives us the same pointer (`0x5576e0844ae0`) from size `64` until `131072`, and then it changes in the next doubling. This is probably because when `malloc` is called by the Rust program, it finds a space in its buffer that can accommodate for future growth without having to move the data around (which is expensive/slow because the bytes have to be copied). This is the kind of management that an allocator has to do. As you can imagine, for different tasks and use cases, different allocators can be better suited.

Wrapping Up

This post is getting way too long, and certainly much longer than I expected when I started. I intend to continue on this topic in a "Part 2" another time.

I have learned some things while doing this experiment. Some of my conclusions:

- The footprint of an assembly program is smaller than that of a C program, which in turn is smaller than that of a Rust program; that is reflected not only in binary size but also in the amount of memory occupied by the code of the program being executed (the *process*), which in turn affects the time it takes to load the program in memory and the amount of syscalls used for that;
- Only building in `--release` mode won't make your binary small; but binary size can be drastically reduced with `strip`; there seems to be some talk of an option for that on `Cargo.toml` but it doesn't seem to be available yet;
- Both C and Rust have a *runtime*, the latter's is considerably larger than the former's; that, among other things, is the cost of Rust's abstractions (and a good builtin `std` library);
- Rust's runtime does, among other things, "panic setup", which involves some allocations; using `panic="abort"` does not seem to completely eliminate this;
- At least at a first glance, allocations seem to behave in a more or less predictable/unsurprising way, which is good, in my opinion; that means we can reason about memory allocation as we normally would, without thinking that `rustc` is doing a lot of magic behind the scenes.

This was a very exploratory post that I wrote during a few days as I made my investigation on this, hopefully it turned out somewhat

readable and perhaps even useful to someone.