How to JIT - an introduction (https://eli.thegreenplace.net/2013/11/05/how-to-jit-an-introduction)

November 05, 2013 at 05:59 **Tags** C & C++ (https://eli.thegreenplace.net/tag/c-c) , Code generation (https://eli.thegreenplace.net/tag/code-generation) , Compilation (https://eli.thegreenplace.net/tag/compilation)

When I wrote the introductory article for libjit (https://eli.thegreenplace.net/2013/10/17/getting-started-with-libjit-part-1/), I aimed it at programmers who know what JITs are, at least to some extent. I did mention what a JIT is, but only very briefly. The purpose of this article is to provide a better introductory overview of JITing, with code samples that don't rely on any libraries.

Defining JIT

JIT is simply an acronym for "Just In Time". That, in itself, doesn't help much - the term is quite cryptic and seems to have little to do with programming. First, let's define what "a JIT" actually refers to. I find the following way to think about this useful:

> Whenever a program, while running, creates and runs some new executable code which was not part of the program when it was stored on disk, it’s a JIT.

What about the historical usage of the term "JIT", though? Luckily, John Aycock from the University of Calgary has written a very interesting paper named "A Brief History of Just-In-Time" (google it, PDFs are available online) looking at JIT techniques from a historical point of view. According to Aycock's paper,

the first mention of code generation and execution during program runtime is apparent as early as McCarthy's LISP paper from 1960. In later work, such as Thompson's 1968 regex paper, it was even more apparent (regexes are compiled into machine code and executed on the fly).

The term JIT was first brought into use in computing literature by James Gosling for Java. Aycock mentions that Gosling has borrowed the term from the domain of manufacturing (http://en.wikipedia.org/wiki/Just_in_time_%28business%29) and started using it in the early 1990s.

This is as far as I'll go into history here. Read the Aycock paper if you're interested in more details. Let's now see what the definition quoted above means in practice.

JIT - create machine code, then run it

I think that JIT technology is easier to explain when divided into two distinct phases:

- Phase 1: create machine code (http://en.wikipedia.org/wiki/Machine_code) at program run-time.
- Phase 2: execute that machine code, also at program run-time.

Phase 1 is where 99% of the challenges of JITing are. But it's also the less mystical part of the process, because this is exactly what a compiler does. Well known compilers like `gcc` and `clang` translate C/C++ source code into machine code. The machine code is emitted into an output stream, but it could very well be just kept in memory (and in fact, both `gcc` and `clang/llvm` have building blocks for keeping the code in memory for JIT execution). Phase 2 is what I want to focus on in this article.

Running dynamically-generated code

Modern operating systems are picky about what they allow a program to do at runtime. The wild-west days of the past came to an end with the advent of protected mode (http://en.wikipedia.org/wiki/Protected_mode), which allows an OS to restrict chunks of virtual memory with various permissions. So in "normal" code, you can create new data dynamically on the heap, but you can't just run stuff from the heap (http://en.wikipedia.org/wiki/Executable_space_protection) without asking the OS to explicitly allow it.

At this point I hope it's obvious that machine code is just data - a stream of bytes. So, this:

```
unsigned char[] code = {0x48, 0x89, 0xf8};
```

Really depends on the eye of the beholder. To some, it's just some data that could represent anything. To others, it's the binary encoding of real, valid x86-64 machine code:

```
mov %rdi, %rax
```

So getting machine code into memory is easy. But how to make it runnable, and then run it?

Let's see some code

The rest of this article contains code samples for a POSIX-compliant Unix OS (specifically Linux). On other OSes (like Windows) the code would be different in the details, but not in spirit. All modern OSes have convenient APIs to implement the same thing.

Without further ado, here's how we dynamically create a function in memory and execute it. The function is intentionally very simple, implementing this C code:

```
long add4(long num) {
  return num + 4;
}
```

Here's a first try (the full code with a Makefile is available in this repo (https://github.com/eliben/libjit-samples)):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>


// Allocates RWX memory of given size and returns a pointer to it. On failure,
// prints out the error and returns NULL.
void* alloc_executable_memory(size_t size) {
  void* ptr = mmap(0, size,
                   PROT_READ | PROT_WRITE | PROT_EXEC,
                   MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
  if (ptr == (void*)-1) {
    perror("mmap");
    return NULL;
  }
  return ptr;
}

void emit_code_into_memory(unsigned char* m) {
  unsigned char code[] = {
    0x48, 0x89, 0xf8,                   // mov %rdi, %rax
    0x48, 0x83, 0xc0, 0x04,             // add $4, %rax
    0xc3                                // ret
  };
  memcpy(m, code, sizeof(code));
}

const size_t SIZE = 1024;
typedef long (*JittedFunc)(long);

// Allocates RWX memory directly.
void run_from_rwx() {
  void* m = alloc_executable_memory(SIZE);
  emit_code_into_memory(m);

  JittedFunc func = m;
  int result = func(2);
  printf("result = %d\n", result);
}
```

The main 3 steps performed by this code are:

1. Use `mmap` to allocate a readable, writable and executable chunk of memory on the heap.
2. Copy the machine code implementing `add4` into this chunk.

3. Execute code from this chunk by casting it to a function pointer and calling through it.

Note that step 3 can only happen because the memory chunk containing the machine code is *executable*. Without setting the right permission, that call would result in a runtime error from the OS (most likely a segmentation fault). This would happen if, for example, we allocated `m` with a regular call to `malloc`, which allocates readable and writable, but not executable memory.

Digression - heap, malloc and mmap

Diligent readers may have noticed a half-slip I made in the previous section, by referring to memory returned from `mmap` as "heap memory". Very strictly speaking, "heap" is a name that designates the memory used by `malloc`, `free` et. al. to manage runtime-allocated memory, as opposed to "stack" which is managed implicitly by the compiler.

That said, it's not so simple :-) While traditionally (i.e. a long time ago) `malloc` only used one source for its memory (the `sbrk` system call), these days most malloc implementations use `mmap` in many cases. The details differ between OSes and implementations, but often `mmap` is used for the large chunks and `sbrk` for the small chunks. The tradeoffs have to do with the relative efficiency of the two methods of requesting more memory from the OS.

So calling memory provided by `mmap` "heap memory" is not a mistake, IMHO, and that's what I intend to keep on doing.

Caring more about security

The code shown above has a problem - it's a security hole. The reason is the RWX (Readable, Writable, eXecutable) chunk of memory it allocates - a paradise for attacks and exploits. So let's be a bit more responsible about it. Here's some slightly modified code:

```
// Allocates RW memory of given size and returns a pointer to it. On failure,
// prints out the error and returns NULL. Unlike malloc, the memory is allocated
// on a page boundary so it's suitable for calling mprotect.
void* alloc_writable_memory(size_t size) {
  void* ptr = mmap(0, size,
                   PROT_READ | PROT_WRITE,
                   MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
  if (ptr == (void*)-1) {
    perror("mmap");
    return NULL;
  }
  return ptr;
}

// Sets a RX permission on the given memory, which must be page-aligned. Returns
// 0 on success. On failure, prints out the error and returns -1.
int make_memory_executable(void* m, size_t size) {
  if (mprotect(m, size, PROT_READ | PROT_EXEC) == -1) {
    perror("mprotect");
    return -1;
  }
  return 0;
}

// Allocates RW memory, emits the code into it and sets it to RX before
// executing.
void emit_to_rw_run_from_rx() {
  void* m = alloc_writable_memory(SIZE);
  emit_code_into_memory(m);
  make_memory_executable(m, SIZE);

  JittedFunc func = m;
  int result = func(2);
  printf("result = %d\n", result);
}
```

It's equivalent to the earlier snippet in all respects except one: the memory is first allocated with RW permissions (just like a normal `malloc` would do). This is all we really need to write our machine code into it. When the code is there, we use `mprotect` to change the chunk's permission from RW to RX, making it executable but *no longer writable*. So the effect is the same, but at no point in the execution of our program the chunk is both writable and executable, which is good from a security point of view.

What about malloc?

Could we use `malloc` instead of `mmap` for allocating the chunk in the previous snippet? After all, RW memory is exactly what `malloc` provides. Yes, we could. However, it's more trouble than it's worth, really. The reason is that protection bits can only be set on virtual memory page boundaries. Therefore, had we used `malloc` we'd have to manually ensure that the allocation is aligned at a page boundary. Otherwise, `mprotect` could have unwanted effects from failing to enabling/disabling more than actually required. `mmap` takes care of this for us by only allocating at page boundaries (because `mmap`, by design, maps whole pages).

Tying loose ends

This article started with a high-level overview of what we mean when we say JIT, and ended with hands-on code snippets that show how to dynamically emit machine code into memory and execute it.

The technique shown here is pretty much how real JIT engines (e.g. LLVM and libjit) emit and run executable machine code from memory. What remains is just a "simple" matter of synthesizing that machine code from something else.

LLVM has a full compiler available, so it can actually translate C and C++ code (through LLVM IR) to machine code at runtime, and then execute it. libjit picks the ball up at a much lower level - it can serve as a backend for a compiler. In fact, my introductory article on libjit (https://eli.thegreenplace.net/2013/10/17/getting-started-with-libjit-part-1/) already demonstrates how to emit and run non-trivial code with libjit. But JITing is a more general concept. Emitting code at run-time can be done for data structures (http://pyevolve.sourceforge.net/wordpress/?p=914), regular expressions (http://sljit.sourceforge.net/) and even accessing C from language VMs (http://luajit.org/ext_ffi.html). Digging in my blog's archives helped me find a mention of some JITing I did 8 years ago (https://eli.thegreenplace.net/2005/09/04/cool-hack-creating-custom-subroutines-on-the-fly-in-perl/). That was Perl code generating more Perl code at run-time (from a XML description of a serialization format), but the idea is the same.

This is why I felt that splitting the JITing concept into two phases is important. For phase 2 (which was explained in this article), the implementation is relatively obvious and uses well defined OS APIs. For phase

1, the possibilites are endless and what you do ultimately depends on the application you're developing.

For comments, please send me an email (mailto:eliben@gmail.com).

Back to top