Using the Unicorn Emulator

This package supplies a Julia wrapper for the **Unicorn Emulation Library**. To get started with the library, simply import it with

```
Pkg.add(url="https://github.com/oblivia-simplex/unicorn-jl")
```

and then, in the REPL, enter

```
using Unicorn
```

An emulator can be initialized by passing an `Arch.t` and a `Mode.t` variant to the `Emulator` constructor, like so:

```
emu = Emulator(X86::t = 4, MODE_64::t = 8, Base.RefValue{Ptr{Nothing}}(Ptr{Nothing}
```

```
emu = Emulator(Arch.X86, Mode.MODE_64)
```

Mapping and Preparing Memory

The next step is typically to map a region of memory to the emulator. This can be done in one of two ways: we may provide the emulator with `address`, `perms` and `size` parameters, and let it allocate the memory itself, or we may pre-allocate an array, and pass it to the emulator. This second option allows us to reuse the same memory across numerous emulators (which we can do safely enough so long as the memory is marked as read-only), and retain direct access to that region of memory.

Note that emulator mapped memory *must* be page-aligned (i.e., evenly divisible by 0x1000).

```
(UInt8[0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

```
text_memory, stack_memory = fill(0x00, 0x2000), fill(0x00, 0x1000)
```

```
UInt8[0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

```
try
    Unicorn.mem_map_array(emu, address = 0x1000, size = 0x2000, perms = Perm.READ |
Perm.WRITE | Perm.EXEC, array = text_memory)
catch e
    if e == Unicorn.UcException(Unicorn.UcError.MAP)
        md"This method will throw an $e exception if run more than once in a row
with the same parameters. This is because the emulator will refuse to map a region
that has already been mapped."
```

```
    else
        throw(e)
    end
end
```

```
UInt8[0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

```
try
    Unicorn.mem_map_array(emu, address = 0x40_000, size = 0x1000, perms = Perm.WRITE
| Perm.READ, array = stack_memory)
catch e
    md"Here, we can expect to see a '$(e)' if the cell is run more than once."
end
```

Now let's load some code into the emulator.

```
49 c7 c6 08 00 04 00      mov $0x00040000, %r14
4c 89 f4                  mov %r14, %rsp
ba ef be ad de            mov $0xcafebeef, %edx
52                        push %rdx
```

```
code = UInt8[0x49, 0xc7, 0xc6, 0x08, 0x00, 0x04, 0x00, 0x4c, 0x89, 0xf4, 0xba,
```

```
code = [
    0x49, 0xc7, 0xc6, 0x08, 0x00, 0x04, 0x00,
    0x4c, 0x89, 0xf4,
    0xba, 0xef, 0xbe, 0xfe, 0xca,
    0x52
]
```

```
mem_write(emu, address = 0x1000, bytes = code)
```

Let's check to see if `text_memory` has been written to.

```
UInt8[0x49, 0xc7, 0xc6, 0x08, 0x00, 0x04, 0x00, 0x4c, 0x89, 0xf4, 0xba, 0xef,
```

```
text_memory
```

We can check the mapped memory regions at any time with `mem_regions()`.

```
Unicorn.MemRegion[Unicorn.MemRegion(0x0000000000001000, 0x0000000000002fff, EXEC|WRI
```

```
mem_regions(emu)
```

Hooking Callbacks into Emulation Events

Much of the power of the Unicorn library comes from its ability to hook specific emulation events and call user-defined callbacks. Let's set a callback to disassemble instructions as they're executed. For this, we'll use the Capstone disassembly library, via its Python bindings, using `PyCall`.

```
using PyCall
```

```
using Printf
```

capstone = PyObject <module 'capstone' from '/home/lucca/.conda/lib/python3.7/site-packages/capstone/__init__.py'>

```
capstone = pyimport("capstone")
```

cs = PyObject <capstone.Cs object at 0x7f74e0d56a58>

```
cs = capstone.Cs(capstone.CS_ARCH_X86, capstone.CS_MODE_64)
```

The best way to get information back out of the emulator is by using closures as callbacks. Here, we're going to use a closure to push disassembly results back into the environment from which the emulation is dispatched.

disassembly = String[]

```
disassembly = Vector{String}()
```

addresses = UInt64[]

```
addresses = Vector{UInt64}()

```

callback = (::Main.workspace140.var"#closure#1"{Array{String,1},Array{UInt64,1}}) (generic function with 1 method)

```
callback =
    let disassembly::Vector{String} = disassembly
    let addresses::Vector{UInt64} = addresses
    sizehint!(disassembly, 1024)
    sizehint!(addresses, 1024)
    function closure(handle::UcHandle, address::UInt64, size::UInt32)
        push!(addresses, address)
        bytes = mem_read(handle, address = address, size = size)
        for inst in cs.disasm(bytes, address)
            dis = @sprintf "0x%x: %s, %s\n" inst.address inst.mnemonic inst.op_str
            push!(disassembly, dis)
        end
    end
end
end
```

Note that we need to be *extremely* careful here, or else we risk memory corruption. The size and memory layout of any data structures that will be mutated by the callback functions should be fixed before execution begins.

hook_handle = 0x0000000003e2c3d0

```
hook_handle = code_hook_add(emu, begin_addr=0x1000, until_addr=0x2000,
callback=callback)
```

Now we're ready to launch the emulation.

OK::t = 0

```
start(emu, begin_addr=0x1000, until_addr=0x2000, steps=4)
```

The `disassembly` vector should now contain the output of the capstone disassembler that we ran in our code hook callback. Let's take a look.

Results of the Disassembly Trace

```
0x1000: mov, r14, 0x40008
0x1007: mov, rsp, r14
0x100a: mov, edx, 0xcafebeef
0x100f: push, rdx
0x1010: add, byte ptr [rax], al
```

```
String["0x1000: mov, r14, 0x40008\n", "0x1007: mov, rsp, r14\n", "0x100a: mov, edx,
```

And the `addresses` vector should contain all of the addresses executed by the emulator.

```
UInt64[0x0000000000001000, 0x0000000000001007, 0x000000000000100a, 0x00000000000010(
```

```
addresses
```

Finally, the `stack_memory` array should contain the word `0xdeadbeef`, which our emulated x86_64 code pushed to the stack.

```
0xcafebeef
```

```
reinterpret(UInt32, stack_memory[1:4])[1]
```

Let's look at the stack pointer in our emulated CPU.

```
0x0000000000040000
```

```
reg_read(emu, X86.Register.RSP)
```

We can also read emulation memory through the Unicorn API, with the `mem_read()` method.

```
0xcafebeef
```

```
reinterpret(UInt32, mem_read(emu, address=0x40_000, size=4))[1]
```

Finally, we can removed the hooked callback with `hook_del()`:

```
hook_del(emu, hook_handle)
```