KVM HOST IN A FEW LINES OF CODE

KVM is a virtualization technology that comes with the Linux kernel. In other words, it allows you to run multiple virtual machines (VMs) on a single Linux VM host. VMs in this case are known as guests. If you ever used QEMU or VirtualBox on Linux - you know what KVM is capable of.

But how does it work under the hood?

IOCTL

KVM provides an API via a special device node - `/dev/kvm`. By opening a device you obtain a handle to the KVM subsystem and later make ioctl syscalls to allocate resources and launch VMs. Some ioctls return file descriptors that can also be controlled by ioctls. Turtles all the way down. But not too deep. There are only a few layers of APIs in KVM:

- /dev/kvm layer, the one used to control the whole KVM subsystem and to create new VMs,
- VM layer, the one used to control an individual virtual machine,
- VCPU layer, the one used to control operation of a single virtual CPU (one VM can run on a multiple VCPUs)

Additionally, there are APIs for I/O devices.

Let’s see how it looks in practice.

```
// KVM layer
int kvm_fd = open("/dev/kvm", O_RDWR);
int version = ioctl(kvm_fd, KVM_GET_API_VERSION, 0);
printf("KVM version: %d\n", version);

// Create VM
int vm_fd = ioctl(kvm_fd, KVM_CREATE_VM, 0);

// Create VM Memory
#define RAM_SIZE 0x10000
void *mem = mmap(NULL, RAM_SIZE, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS |
struct kvm_userspace_memory_region mem = {
        .slot = 0,
        .guest_phys_addr = 0,
        .memory_size = RAM_SIZE,
        .userspace_addr = (uintptr_t) mem,
};
ioctl(vm_fd, KVM_SET_USER_MEMORY_REGION, &mem);

// Create VCPU
int vcpu_fd = ioctl(vm_fd, KVM_CREATE_VCPU, 0);
```

At this point, we’ve created a new VM, allocated its memory and assigned one VCPU for it. To make our VM actually run something we need to load VM image and configure CPU registers properly.

LOADING VM

This one is simple. Just read a file and copy its contents into the VM memory. Of course, `mmap` might be a good option here, too.

```
int bin_fd = open("guest.bin", O_RDONLY);
if (bin_fd < 0) {
        fprintf(stderr, "can not open binary file: %d\n", errno);
        return 1;
}
char *p = (char *)ram_start;
for (;;) {
        int r = read(bin_fd, p, 4096);
        if (r <= 0) {
                break;
        }
        p += r;
}
close(bin_fd);
```

It is assumed that `guest.bin` contains valid bytecode for the current CPU architecture, because KVM does not interpret CPU instructions one by one, like old-school VM hosts did. It lets the real CPU do the calculations and only intercepts the I/O. That's why modern VMs run at a very decent performance, close to bare metal, unless you do I/O heavy operations.

Here's a tiny guest VM "kernel", that we will try to run first:

```
#
# Build it:
#
# as -32 guest.S -o guest.o
#       ld -m elf_i386 --oformat binary -N -e _start -Ttext 0x10000 -o guest guest.o
#
.globl _start
.code16
_start:
  xorw %ax, %ax
loop:
        out %ax, $0x10
        inc %ax
        jmp loop
```

If assembly is out of your interest, it’s a tiny 16-bit executable that increments a register in a loop and outputs the value into the I/O port 0x10.

It was deliberate that we compiled it as an archaic 16-bit app, because the KVM VCPU starts can run in multiple modes, much like the real x86 processor. The simplest mode is “real” mode, which has been used to run 16-bit code from the last century. Real mode is notable for memory addressing, it’s direct instead of using descriptor tables - it would be simpler to initialize our register for real mode:

```
struct kvm_sregs sregs;
ioctl(vcpu_fd, KVM_GET_SREGS, &sregs);
// Initialize selector and base with zeros
sregs.cs.selector = sregs.cs.base = sregs.ss.selector = sregs.ss.base = sregs.ds.selec
```

```
// Save special registers
ioctl(vcpu_fd, KVM_SET_SREGS, &sregs);

// Initialize and save normal registers
struct kvm_regs regs;
regs.rflags = 2; // bit 1 must always be set to 1 in EFLAGS and RFLAGS
regs.rip = 0; // our code runs from address 0
ioctl(vcpu_fd, KVM_SET_REGS, &regs);
```

RUNNING

Code is loaded, registers are ready. Shall we start? To run the VM we need to get a pointer to the “run state” for each VCPU and then enter a loop where the VM is run until it’s interrupted by I/O or other operations, where it passes the control back to the host.

```
int runsz = ioctl(kvm_fd, KVM_GET_VCPU_MMAP_SIZE, 0);
struct kvm_run *run = (struct kvm_run *) mmap(NULL, runsz, PROT_READ | PROT_WRITE, MAP

for (;;) {
	ioctl(vcpu_fd, KVM_RUN, 0);
	switch (run->exit_reason) {
	case KVM_EXIT_IO:
		printf("IO port: %x, data: %x\n", run->io.port, *(int *)((char *)(run)
		break;
	case KVM_EXIT_SHUTDOWN:
		return;
	}
}
```

Now, if we run the app we will see:

```
IO port: 10, data: 0
IO port: 10, data: 1
IO port: 10, data: 2
IO port: 10, data: 3
IO port: 10, data: 4
...
```

It works! The complete sources are available in this gist: https://gist.github.com/zserge/d68683f17c68709818f8baab0ded2d15 (if you spot a mistake - comments are welcome!)

YOU CALL IT A KERNEL?

Obviously, it's not that impressive. How about running a Linux kernel instead?

The beginning would be the same - open /dev/kvm, create a VM etc. However, we will need a few more ioctls in the VM layer to add a periodic interval timer, to initialize TSS (required for Intel chips), to add interrupt controller:

```
ioctl(vm_fd, KVM_SET_TSS_ADDR, 0xffffd000);
uint64_t map_addr = 0xffffc000;
ioctl(vm_fd, KVM_SET_IDENTITY_MAP_ADDR, &map_addr);
ioctl(vm_fd, KVM_CREATE_IRQCHIP, 0);
```

```c
struct kvm_pit_config pit = { .flags = 0 };
ioctl(vm_fd, KVM_CREATE_PIT2, &pit);
```

Also, we will need to change the way we initialize the registers. Linux kernel requires a protected mode, so we enable that in register flags and initialize base, selector, granularity for each special register:

```c
sregs.cs.base = 0;
sregs.cs.limit = ~0;
sregs.cs.g = 1;

sregs.ds.base = 0;
sregs.ds.limit = ~0;
sregs.ds.g = 1;

sregs.fs.base = 0;
sregs.fs.limit = ~0;
sregs.fs.g = 1;

sregs.gs.base = 0;
sregs.gs.limit = ~0;
sregs.gs.g = 1;

sregs.es.base = 0;
sregs.es.limit = ~0;
sregs.es.g = 1;

sregs.ss.base = 0;
sregs.ss.limit = ~0;
sregs.ss.g = 1;
sregs.cs.db = 1;
sregs.ss.db = 1;
sregs.cr0 |= 1; // enable protected mode

regs.rflags = 2;
regs.rip = 0x100000; // This is where our kernel code starts
regs.rsi = 0x10000; // This is where our boot parameters start
```

What are the boot parameters and why can't we just load kernel at address zero? Time to learn more about the bzImage format.

The kernel image follows a special "boot protocol" and there is a fixed header with boot parameters, followed by the actual kernel bytecode. The format of the boot header is described here.

LOADING KERNEL IMAGE

To properly load kernel image into our VM we need to read the whole bzImage file first. The we look at the offset 0x1f1 and get the number of setup sectors from there. This is what we shall skip to find where the kernel code starts. Additionally, we will copy boot parameters from the beginning of the bzImage into the boot parameters offset in VM RAM (0x10000).

But even doing so is not enough. We will have to patch the boot parameters for our VM, to force the VGA mode, and to initialize the command line pointer.

We want our kernel to print logs to ttyS0, so that we could intercept the I/O and our VM host would print it to stdout. To achieve this we need to append “console=ttyS0” to the kernel command line.

But even after doing so, we won’t get any result. I had to set a fake CPU ID to our kernel to start (https://www.kernel.org/doc/Documentation/virtual/kvm/cpuid.txt). Most likely the kernel I’ve built was relying on this information to tell if it’s running inside a hypervisor, or on bare metal.

I was using a kernel compiled with “tiny” config, and adjusted a few configuration flags to support serial console and virtio.

The full code of the modified KVM host and test kernel image is available as a gist: https://gist.github.com/zserge/ae9098a75b2b83a1299d19b79b5fe488

If we compile it and run, we will get the following output:

```
Linux version 5.4.39 (serge@melete) (gcc version 7.4.0 (Ubuntu 7.4.0-1ubuntu1~16.04~pp
Command line: console=ttyS0
Intel Spectre v2 broken microcode detected; disabling Speculation Control
Disabled fast string operations
x86/fpu: Supporting XSAVE feature 0x001: 'x87 floating point registers'
x86/fpu: Supporting XSAVE feature 0x002: 'SSE registers'
x86/fpu: Supporting XSAVE feature 0x004: 'AVX registers'
x86/fpu: xstate_offset[2]:  576, xstate_sizes[2]:  256
x86/fpu: Enabled xstate features 0x7, context size is 832 bytes, using 'standard' form
BIOS-provided physical RAM map:
BIOS-88: [mem 0x0000000000000000-0x000000000009efff] usable
```

```
BIOS-88: [mem 0x0000000000100000-0x00000000030fffff] usable
NX (Execute Disable) protection: active
tsc: Fast TSC calibration using PIT
tsc: Detected 2594.055 MHz processor
last_pfn = 0x3100 max_arch_pfn = 0x400000000
x86/PAT: Configuration [0-7]: WB  WT  UC- UC  WB  WT  UC- UC
Using GB pages for direct mapping
Zone ranges:
  DMA32    [mem 0x0000000000001000-0x00000000030fffff]
  Normal   empty
Movable zone start for each node
Early memory node ranges
  node   0: [mem 0x0000000000001000-0x000000000009efff]
  node   0: [mem 0x0000000000100000-0x00000000030fffff]
Zeroed struct page in unavailable ranges: 20322 pages
Initmem setup node 0 [mem 0x0000000000001000-0x00000000030fffff]
[mem 0x03100000-0xffffffff] available for PCI devices
clocksource: refined-jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 76
Built 1 zonelists, mobility grouping on.  Total pages: 12253
Kernel command line: console=ttyS0
Dentry cache hash table entries: 8192 (order: 4, 65536 bytes, linear)
Inode-cache hash table entries: 4096 (order: 3, 32768 bytes, linear)
mem auto-init: stack:off, heap alloc:off, heap free:off
Memory: 37216K/49784K available (4097K kernel code, 292K rwdata, 244K rodata, 832K ini
Kernel/User page tables isolation: enabled
NR_IRQS: 4352, nr_irqs: 24, preallocated irqs: 16
Console: colour VGA+ 142x228
printk: console [ttyS0] enabled
APIC: ACPI MADT or MP tables are not detected
APIC: Switch to virtual wire mode setup with no configuration
Not enabling interrupt remapping due to skipped IO-APIC setup
clocksource: tsc-early: mask: 0xffffffffffffffff max_cycles: 0x25644bd94a2, max_idle_n
Calibrating delay loop (skipped), value calculated using timer frequency.. 5188.11 Bog
pid_max: default: 4096 minimum: 301
```

```
Mount-cache hash table entries: 512 (order: 0, 4096 bytes, linear)
Mountpoint-cache hash table entries: 512 (order: 0, 4096 bytes, linear)
Disabled fast string operations
Last level iTLB entries: 4KB 64, 2MB 8, 4MB 8
Last level dTLB entries: 4KB 64, 2MB 0, 4MB 0, 1GB 4
CPU: Intel 06/3d (family: 0x6, model: 0x3d, stepping: 0x4)
Spectre V1 : Mitigation: usercopy/swapgs barriers and __user pointer sanitization
Spectre V2 : Spectre mitigation: kernel not compiled with retpoline; no mitigation ava
Speculative Store Bypass: Vulnerable
TAA: Mitigation: Clear CPU buffers
MDS: Mitigation: Clear CPU buffers
Performance Events: Broadwell events, 16-deep LBR, Intel PMU driver.
...
```

Obviously, it’s still a fairly useless result - there is no initrd or root partition, no actual apps that could run in this kernel, but still it proves that KVM is not that scary and a rather powerful tool.

SUMMARY

To make it run a proper Linux, the VM host has to be much more advanced - we need to simulate multiple I/O drivers for disks, keyboard, graphics. But the general approach would remain the same, for example for initrd we would map is similarly to the command line options. For disks we would have to intercept I/O and respond properly.

However, no one forces you to use KVM directly. There is libvirt, a nice friendly wrapper for low-level virtualization techniques such as KVM or BHyve.

If you are interested to learn more about KVM, I would suggest to look at the kvmtool sources. They are much easier to read than QEMU, and the whole project is much smaller and simpler.

I hope you've enjoyed this article. You can follow – and contribute to – on Github, Twitter or subscribe via rss.

May 10, 2020

See also: Linux containers in a few lines of code and more.