KVM

Florent Gluck - Florent.Gluck@hesge.ch

March 8, 2023

Kernel modules

Linux kernel modules

Module = kernel code that can be loaded/unloaded at runtime

- Allows to add/remove kernel features while system is running
- Modules have **full privileges** and control of the system → buggy modules may crash the kernel!
- Make it easy to develop drivers without rebooting
- Help keep kernel image size to a minimum
- Help reduce boot time: avoid spending time initializing devices and kernel features that will only be needed later
- Modules installed in `/lib/modules/<kernel_version>/kernel` and have the `.ko` extension

Loading modules

- To load a single module **without** its dependencies:

```
sudo insmod <module_path>.ko
```

- To load a module **with** its dependencies:

```
sudo modprobe <module_name>
```

- `modprobe` reads `/lib/modules/<kernel_version>/modules.dep.bin` to determine:
  - each module's location (path)
  - each module's dependencies

Module utilities

- To get information about a module (parameters, license, description, dependencies, etc.):

```
modinfo <module_name>
modinfo <module_path>.ko
```

- To display all loaded modules (see `/proc/modules`):

```
lsmod
```

- To remove a module (and its depedencies with `-r`):

```
rmmod <module_name>
```

KVM

What is KVM?

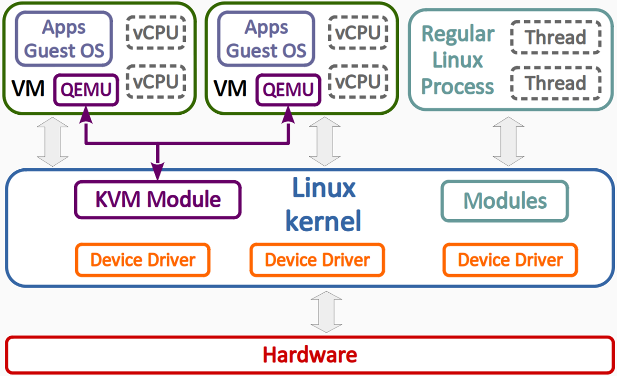
- **KVM** stands for **K**ernel based **V**irtual **M**achine
- Linux kernel module providing hardware-assisted virtualization
- **Provide a virtualization API** for hypervisors
- **Requires** Intel VT-x or AMD-V
- Originally, KVM virtualized only CPU and memory
  - devices (I/O) had to be emulated by QEMU
- Nowadays, KVM supports device virtualization (PIC and probably more)
- Being part of Linux, KVM is open-source software

Evolution of KVM

- Introduced to make VT-x/AMD-V available to user space
  - expose virtualization features securely
  - interface: `/dev/kvm`
- Quickly merged into Linux mainline
  - available since kernel 2.6.20 (2006)
  - from first LKML posting to kernel merge: only 3 months!
  - 7300 lines of C code! (as of Linux 5.8.12)
- Evolved significantly since 2006
  - ported to other architectures: RAM, s390, PowerPC, IA64
  - became recognized & driving part of Linux kernel
  - quick support of latest virtualization features

KVM model

- User processes can create VMs → VMs are just user space processes
- Virtual CPUs (vCPUs) mapped to kernel threads



Architectural benefits of KVM model

- **Proximity of guest and user space hypervisor**
  - Only one address space switch: guest $\leftrightarrow$ host
  - Both run in user space $\rightarrow$ lighter context switch
- **Massive Linux kernel reuse**
  - Memory management
  - Scheduler
  - I/O stacks, power management, host CPU hot-plugging, etc.
- **Massive Linux user space reuse**
  - Network configuration
  - Handling of VM images
  - Logging, tracing, debugging

KSM (Kernel Samepage Merging)

- Linux kernel feature that deduplicates “identical” pages found across user processes
  - **benefit: massive gain in memory usage!**
  - drawback: security (theoretical)
- Spawn `ksmd` daemon which inspects pages for possible merges
- Kernel must be compiled with `CONFIG_KSM=y` (> 2.6.32)
- KSM controled[1] by writing to `/sys/kernel/mm/ksm/run`:
  - 0: stop `ksmd` from running but keep merged pages
  - 1: run `ksmd`
  - 2: stop `ksmd` and unmerge all pages currently merged
- Article on KSM "Increasing memory density by using KSM"[2]

[1] https://www.kernel.org/doc/Documentation/vm/ksm.txt
[2] https://www.kernel.org/doc/ols/2009/ols2009-pages-19-28.pdf

Can a host run KVM?

- Check for hardware virtualization support (Intel or AMD):

```
$ lscpu|grep Flags|grep "vmx\|svm"
Flags:  fpu vme de pse tsc msr pae mce cx8 apic sep
    mtrr pge mca cmov pat pse36 clflush dts acpi mmx
    fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp
     lm constant_tsc arch_perfmon pebs bts rep_good nopl
     xtopology nonstop_tsc cpuid aperfmperf pni
    pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2
    ssse3 sdbg fma cx16 xtpr pdcm ...
```

- Check the `kvm` module is loaded in the kernel:

```
$ lsmod|grep kvm
kvm_intel              282624  0
kvm                    663552  1 kvm_intel
```

Accessing KVM

- To access the KVM device, `/dev/kvm`, one **must** either (depends on the Linux distro):
  - be in the `kvm` group[3]
  - have the proper ACL permissions[4]
- Typical examples of KVM API use:
  - QEMU when launched with `-enable-kvm`
  - Any other Linux-based hypervisor using KVM
  - Any application using `/dev/kvm`, typically a custom VMM

[3] https://linuxize.com/post/how-to-add-user-to-group-in-linux/
[4] https://www.redhat.com/sysadmin/linux-access-control-lists

KVM API

Overview

- Device /dev/kvm provides access to the KVM API
  - kvm module must be loaded!
- Requests performed through ioctl calls
- Provide 3 types of resources, accessed by file descriptors:
  - system (KVM): VM creation, memory mapping, etc.
  - VM: vCPU creation, interrupts, etc.
  - vCPU : access to registers, etc.

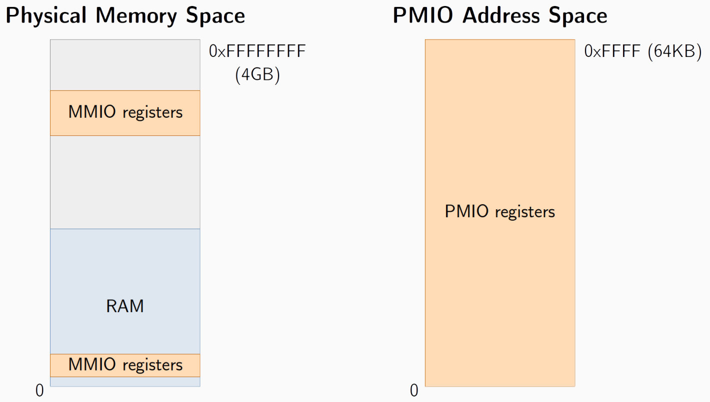
Basic example

```c
#include <linux/kvm.h>

const uint8_t code[] = {  // Write 0x42 at I/O adress 0x80:
  0x66, 0xba, 0x80, 0x00, // mov dx,0x80
  0xb0, 0x42,             // mov al,0x42
  0xee                    // out dx,al
};

int main(int argc, char **argv) {
    // Open KVM
    int kvmfd = open("/dev/kvm", O_RDWR | O_CLOEXEC);
    // Set up a virtual machine
    int vmfd = ioctl(kvmfd, KVM_CREATE_VM, (unsigned long)0);
    // Get some page-aligned memory and copy code to it
    void *m = mmap(0, 0x1000, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS,
         -1, 0);
    memcpy(m, code, sizeof(code));
    // Create a virtual CPU #0
    int vcpufd = ioctl(vmfd, KVM_CREATE_VCPU, (unsigned long)0));
    // Run the VCPU #0
    while (1) {
        ioctl(vcpufd, KVM_RUN, 0);
        ...
    }
}
```

Devices: MMIO vs PMIO

- **Memory-Mapped I/O devices (MMIO)**
  - device registers are mapped into the physical memory space
  - **RAM and devices share the same address space!**
  - read/write to/from these devices happen exactly like memory
  - all instructions dealing with memory operands can interact with these devices
    - e.g. `mov` instruction (x86) → `mov al,[42]`
- **Port-Mapped I/O devices (PMIO)**
  - device registers are mapped into a specific memory space, **distinct** from the physical memory space
  - require **specific** instructions to access these devices
    - e.g. `in` and `out` instructions (x86) → `in al,42`

Device address spaces



- MMIO address space, accessed using regular memory instructions (`mov`)
- Distinct memory space, **only** accessed using specific assembly instructions (`in`, `out`)

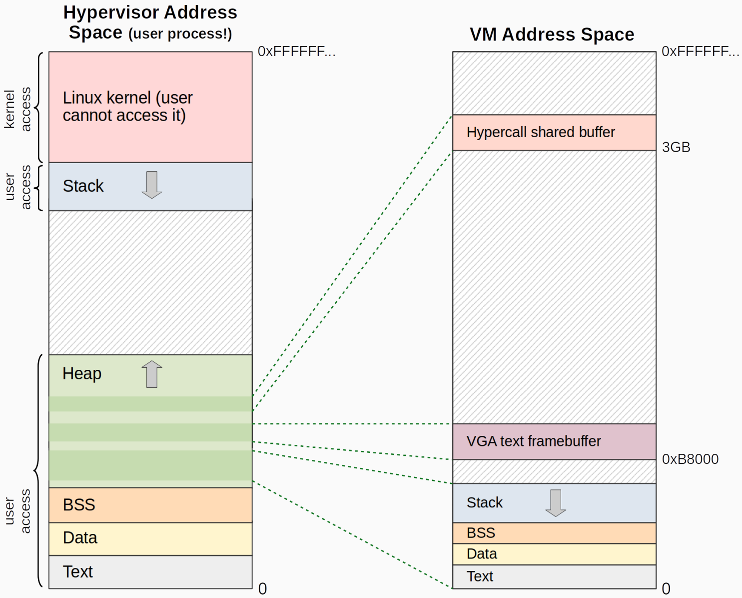
KVM workflow from the VMM's perspective

1. Create a KVM device
2. Create a VM
3. Allocate and map RAM for the VM
4. Allocate and map MMIO memory (devices) for the VM
5. Load guest OS binary blob into the VM's RAM
6. Create a vCPU and setup its registers
7. Run the vCPU on the guest OS' code
8. Handle `VMexits` to emulate the guest OS' expected behavior

vCPU execution and VMexits

- **vCPU execution happens at native speed**, without interruption until guest OS code generates a `VMexit`
- A `VMexit` occurs when guest OS code:
  - reads from or writes to an I/O port
  - writes to a MMIO address
  - triggers some special operation (including errors)
- After a `VMexit` is handled by the VMM code, the vCPU resumes its execution

VM memory mapping example



(1) Create a KVM device

- To obtain a file descriptor on the kvm device and check the stable version of the API is available:

```
int kvmfd = open("/dev/kvm", O_RDWR | O_CLOEXEC);
if (kvmfd < 0) err(1, "%s", "/dev/kvm");

int version = ioctl(kvmfd, KVM_GET_API_VERSION, NULL);
if (version < 0) err(1, "KVM_GET_API_VERSION");
if (version != KVM_API_VERSION) err(1, "Unsupported
    version of the KVM API");
```

(2) Create a VM

- To obtain a file descriptor on a newly created VM:

```
int vmfd = ioctl(kvmfd, KVM_CREATE_VM, 0);
if (vmfd < 0) err(1, "KVM_CREATE_VM");
```

(3)(4) Allocate RAM or MMIO for the guest

- Memory allocated for the guest is made out of pages
- Page are 4KB in size and aligned to 4KB
- Allocated memory for guest **must** be pages
  - must be aligned to 4KB and multiple of 4KB in size
  - `mmap` must be used as `malloc`'s memory not aligned to 4KB

```
// Alloc 256KB for the guest
u_int page_count = 64;
u_int size = 4096*page_count;
uint8_t *mem = mmap(NULL, size, PROT_READ|PROT_WRITE,
    MAP_SHARED|MAP_ANONYMOUS, -1, 0);
if (!mem) err(1, "Allocating guest memory");
```

(3) Map RAM into the guest address space

- Define where in the guest, the memory is physically mapped
- Below code maps `mem` at physical address 0 in the guest
- Each memory mapping (*region* in KVM lingo) must be associated to a different slot, here 0

```
struct kvm_userspace_memory_region memreg = {
    .slot = 0,
    .guest_phys_addr = 0,
    .memory_size = size,
    .userspace_addr = (uint64_t)mem,
    .flags = 0
};
if (ioctl(vmfd, KVM_SET_USER_MEMORY_REGION, &memreg)
    < 0) err(1, "KVM_SET_USER_MEMORY_REGION");
```

(4) Map MMIO into the guest address space

- Allow hypervisor to be notified when guest tries to write to the MMIO area
- Same as RAM mapping, except must be **marked as read-only**
- Below code maps `mmio` at physical address 32MB in the guest

```
struct kvm_userspace_memory_region mmioreg = {
    .slot = 1,
    .guest_phys_addr = 32*1024*1024,
    .memory_size = mmio_size,
    .userspace_addr = (uint64_t)mmio,
    .flags = KVM_MEM_READONLY  // mandatory for MMIO!
};
if (ioctl(vmfd, KVM_SET_USER_MEMORY_REGION, &mmioreg)
     < 0) err(1, "KVM_SET_USER_MEMORY_REGION");
```

(5) Load guest OS into VM's RAM

- Guest OS must be "loaded" into the guest address space
- Simplest guest OS = mini bare-metal OS
- Simply perform the following operations:
    1. read the binary file generated from compiling/linking the guest OS' code + data
    2. read the file into the pages allocated for the guest RAM
    3. easiest is to load it at address 0 in guest physical memory
        - later on, we'll set the CPU instruction pointer to 0 as well

(6) Create a vCPU

- A vCPU is referenced through a file descriptor, `vcpufd` below
- The vCPU is represented as a memory-mapped file
  - the memory-mapped area is a `kvm_run` structure, `run` below

```
int vcpufd = ioctl(vmfd, KVM_CREATE_VCPU, 0);
if (vcpufd < 0) err(1, "KVM_CREATE_VCPU");

int vcpu_mmap_sz = ioctl(kvmfd, KVM_GET_VCPU_MMAP_SIZE, NULL);
if (vcpu_mmap_sz < 0) err(1, "KVM_GET_VCPU_MMAP_SIZE");
if (vcpu_mmap_sz < sizeof(struct kvm_run)) err(1, "KVM_GET_VCPU_MMAP_SIZE
     unexpectedly small");

struct kvm_run *run = mmap(NULL, vcpu_mmap_sz, PROT_READ | PROT_WRITE,
     MAP_SHARED, vcpufd, 0);
if (!run) err(1, "mmap vcpu");
```

(6) Setup the vCPU registers (1/2)

- Initialize code segment register `cs`:

```
struct kvm_sregs sregs;
if (ioctl(vm.vcpufd, KVM_GET_SREGS, &sregs) < 0) err(1,
     "KVM_GET_SREGS");
sregs.cs.base = 0;
sregs.cs.selector = 0;
if (ioctl(vm.vcpufd, KVM_SET_SREGS, &sregs) < 0) err(1,
     "KVM_SET_SREGS");
```

(6) Setup the vCPU registers (2/2)

- Initialize instruction pointer `rip` to point to the beginning of the OS' code
- Initialize flags register `rflags` (bit 1 is reserved: must be 1)
- Initialize stack pointer `rsp` to point to top of the RAM

```
struct kvm_regs regs;
memset(&regs, 0, sizeof(regs));
regs.rsp = size;
regs.rip = 0;
regs.rflags = 0x2;
if (ioctl(vcpufd, KVM_SET_REGS, &regs) < 0) err(1, "
    KVM_SET_REGS");
```

(7) Run the vCPU

Use the `KVM_RUN` `ioctl` on the vCPU file descriptor to run it:

```
while (1)) {
  int status = ioctl(vcpufd, KVM_RUN, NULL);
  if (status < 0) {
      err(1, "VMM: KVM_RUN");
  }
  switch (run->exit_reason) {
      case KVM_EXIT_IO:
          ...
      case KVM_EXIT_MMIO:
          ...
      case KVM_EXIT_HLT:
          ...
      case KVM_EXIT_SHUTDOWN:
          ...
      case KVM_EXIT_FAIL_ENTRY:
      case KVM_EXIT_INTERNAL_ERROR:
      default:
          fprintf(stderr, "KVM error");
          ...
  }
}
```

(8) Handle VMexits to emulate desired behavior

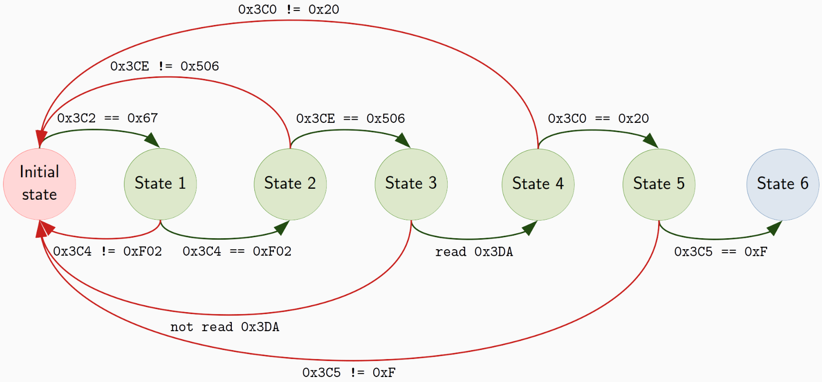
- There are many types of `VMexits`, but these 2 are especially interesting:
  - `KVM_EXIT_IO`: encountered when guest issues an I/O (`in` or `out`) machine instruction
  - `KVM_EXIT_MMIO`: encountered when guest writes to a read-only memory slot (see slide “Mapping MMIO into the guest address space”)
- Note that `KVM_EXIT_IO` is **significantly faster** than `KVM_EXIT_MMIO` (according to KVM documentation)
- VMM must then emulate the desired behavior expected by the guest OS

Device emulation

- Emulating a real device allows a guest OS implementing a driver for the real hardware to use it
- OS implements drivers to support various physical devices, for instance:
  - VGA display, mouse, keyboard, SATA drive, etc.
- Device drivers write and read to specific device registers
  - either MMIO or PMIO or both
- How does an hypervisor emulate a device?

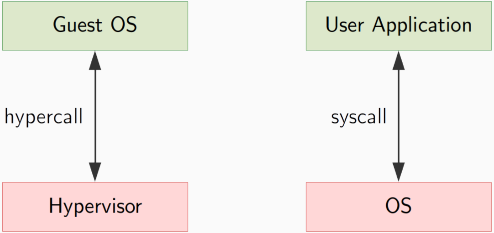
Device emulation: state machine

```
// Code excerpt of init. sequence for VGA X-mode 400x300
outb(0x3C2, 0x67);
outw(0x3C4, 0x0F02);  // enable writing to all planes
outw(0x3CE, 0x0506);  // graphic mode
inb(0x3DA);
outb(0x3C0, 0x20);   // enable video
outb(0x3C5, 0x0F);
```



Hypercalls

- Mechanism for the guest to request the help of the hypervisor
- Similar to a system call between an application and an OS

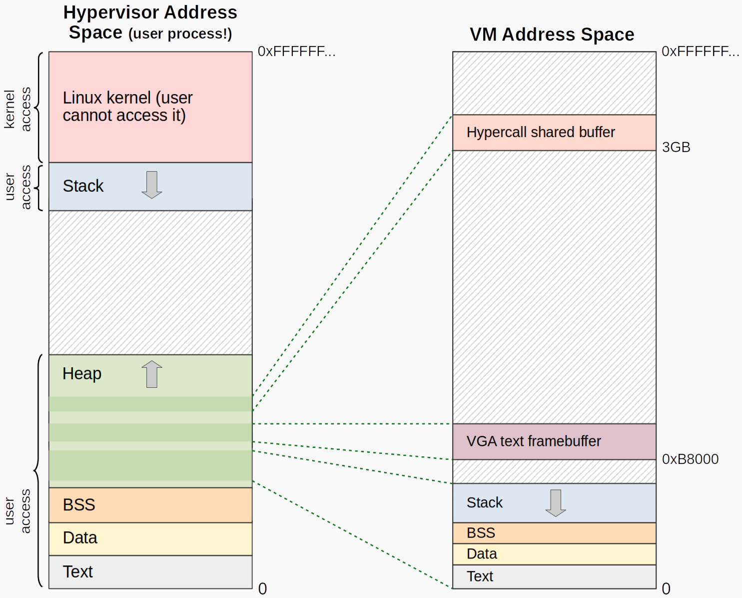


- Benefits
  - **much simpler drivers** → no need to emulate the real hardware!
  - **much better performance!**

Hypercalls: simple implementation example

- **Guest**
  - write hypercall arguments in shared memory (buffer) between guest $\leftrightarrow$ hypervisor
  - write hypercall request to an unused port, e.g. `0xABBA`
- **Hypervisor**
  - when `KVM_EXIT_IO` encountered, check whether it's an hypercall
  - if hypercall, then extract its arguments and emulate the expected behavior
    - possibly write an output to the shared buffer

Hypercalls: shared buffer



Gracefully exit the VM (1/2)

Case 1: guest OS explicitely stops the CPU

- Guest: executes the `hlt` machine instruction to stop the CPU
- Hypervisor: `hlt` triggers the `KVM_EXIT_HLT` VMexit
- Do not forget to deallocate the VM's resources!

Gracefully exit the VM (2/2)

Case 2: execution of the guest OS never stops (infinite loop)

→ `KVM_RUN ioctl` never stops either!

- How to handle a graceful exit initiated on the host?
  - run the vCPU in a dedicated thread
  - configure the thread to be **asynchronously interruptible** with:

```
pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
```

  - when the VM must be stopped, cancel the vCPU thread with:

```
pthread_cancel(vcpu_thread);
```

- Do not forget to deallocate the VM's resources!

Cleanup the VM's data structures

- Using `munmap`, free all memory regions allocated with `mmap`
- Close the KVM device using the file descriptor previously returned by `open(/dev/kvm, ...)`

Resources

- KVM
  http://linux-kvm.org
- KVM API reference
  https://www.kernel.org/doc/html/latest/virt/kvm/api.html
- Using the KVM API
  https://lwn.net/Articles/658511/
- Kernel module (Arch Linux documentation)
  https://wiki.archlinux.org/index.php/Kernel_module