Build your own hypervisor using the KVM API

This assignment is borrowed from the Virtualization and Cloud Computing course of IIT Bombay

[Instructor : Prof. Mythilli Vutukuru]

1. Go through the following link : https://lwn.net/Articles/658511/
and build your own VM as mentioned in the tutorial.

2. To be able to run any of these programs, you must have QEMU/libvirt/KVM installed on your system, and your system hardware must also have support for hardware virtualization. This link tells you how to do this for Ubuntu; you should find several such links online.

3. Extend the KVM "Hello, world!" code available on GitHub. You may download the code directly from GitHub, or use our local copy of the code.

4. You should begin reading the code at the main function of `kvm-hello-world.c`, which begins by calling `vm_init` and `vcpu_init`. The code within these functions allocates memory for the guest and its VCPU.

Questions to be answered :

Q: What is the size of the guest memory (that the guest perceives as its physical memory) that is setup in the function `vm_init`? How and where in the hyprevisor code is this guest memory allocated from the host OS? At what virtual address is this memory mapped into the virtual address space of this simple hypervisor? (Note that the address returned by mmap is a host virtual address.)

Q: Besides the guest memory, every VCPU also allocates a small portion of VCPU runtime memory from the host OS in the function , to store the information it has to exchange with KVM. In which lines of the program is this memory allocated, what is its size, and where is it located in the virutal address space of the hypervisor?

After allocating the guest and VCPU memory, the program proceeds to format the guest memory area and CPU registers, by configuring these values via the KVM API. Let us understand how the guest is setup in the long mode.

Q: The guest memory area is formatted to contain the guest code (which is made available as an `extern char` array in the executable), the guest page table, and a kernel stack. Can you identify where in the code each of these is setup? What range of addresses do each of these occupy, in the guest physical address space, and the host virtual address space? That is, can you visualize the physical address space of the guest VM, as well as the virtual address space of the host user process that is setting up the VM?

Q: A more detailed study of the code will help you understand the structure of the guest page table, and how the guest virtual address space is mapped to its physical address space using its page table. How many levels does the guest page table have in long mode? How many pages does it occupy? What are the (guest) virutal-to-physical mappings setup in the guest page table? What parts of the

guest virtual address space is mapped by this page table? Can you visualize the page table structure and the address translation in the guest?

In the end, the guest's fancy page table simply computes the physical address as exactly equal to the virtual address, so that the guest's entire physical address space can be addressed directly without any concept of virtual addressing. But you will need to understand the page table structure to convince yourself of this fact. Some hints to help you understand the 64-bit page table setup are given below.

- A brief overview of page table structure in 64-bit systems: In typical 64-bit systems, the least significant 48 bits are used for virtual addresses today. Assuming 4 KB pages, the virtual address space has 2^36 pages. Assuming each page table entry is 8 bytes, each page can store 2^9 page table entries. Therefore, the page table has 4 levels. The CR3 register stores the physical address of the outermost page table, called the PML4 (page map level 4). The most significant 9 (out of 48) bits are used to index into the pml4 table to obtain the physical address of the PDP (page directory pointer) table. The next 9 bits are used to index into the PDP to obtain the address of the page directory (PD) table. The next 9 bits index into page directory to obtain the address of the (innermost) page table. The next 9 bits are used to index into the page table to obtain the physical frame number. The last 12 bits form the offset within the page.
- However, when phsical address extension is enabled via the CR4_PAE flag, the page size is treated as 2MB, and the last 21 bits are used to index into the pgdir. That is, there are only 3 levels of page table, the most significant 27 bits are used to index into these 3 levels, and the lookup in the innermost page table is ommitted.
- If a page table entry has no physical frame number and only a set of flags, then the physical frame number stored will be zero.

Q: At what (guest virtual) address does the guest start execution when it runs? Where is this address configured?

After configuring the guest memory and registers, the hypervisor proceeds to run the guest in the function `run_vm`.

Q: At which line in the hypervisor program does the control switch from running the hypervisor to running the guest? At which line does the control switch back to the hypervisor from the guest?

It is now time to start reading the guest code in `guest.c`, whose binary was copied into the guest memory area by the hypervisor. Note that this guest code is minimalistic, and doesn't use a lot of fancy C libraries, in order to keep the guest memory footprint small. The guest does not have multiple processes. It just has one executable, a simple page table and a basic stack, all of which are already setup. There is no switching across processes, page tables, or stacks. Real-life guest OSes are of course much more full-fledged, but the mechanism of switching between guest and host that we wish to understand remains the same.

Our simple guest first prints out the "Hello, world!" string to the screen. The guest uses the `outb` CPU instruction to write a byte of data to a certain I/O port number. This instruction causes the guest to exit to our hypervisor program (since guests cannot perform privileged operations such as accessing I/O ports), and the hypervisor then prints out the character on behalf of the guest.

Note that the actual I/O instruction of `outb` is written in assembly language and embedded into the C code, a technique called "Inline Assembly". You can read more about inline assembly here, here, here and at many such links online. While we encourage you to understand inline assembly fully, below is a short explanation that suffices for now.

Consider the following code snippet in `guest.c`

```
static void outb(uint16_t port, uint8_t value) {
        asm("outb %0,%1" : /* empty */ : "a" (value), "Nd" (port) : "memory");
}
```

This C function `outb` takes two arguments, a port number and a value. Note that the x86 EAX and EDX registers are conventionally used to hold arguments for the `outb` instruction. Therefore, the assembly code in this function loads the value to put out on the port into the EAX register, the port number into the EDX register, and invokes the `outb` instruction with suitable arguments.

Once this instruction is executed by the guest, it causes an exit into the hypervisor code. The hypervisor checks the reason for the exit and handles it accordingly.

Q: Can you fully understand how the "Hello, world!" string is printed out by the guest via the hypervisor? What port number is used for this communication? How can you read the port number and the value written to the port within the hypervisor? Which memory buffer is used to communicate the value written by the guest to the hypervisor? How many exits (from guest to hypervisor) are required to print out the complete string?

After saying hello to the world, the guest writes the number 42 into a memory location and into the EAX register, before invoking the halt instruction to quit.

Q: Can you figure out what's happening with the number 42? Where is it written in the guest and where is it read out in the hypervisor?

That's it we're done! Hopefully, you should have fully understood this KVM "Hello, world!" example, and you should be all set to extend this code.