Programming Assignment 1: Build your own hypervisor using the KVM API

Back to CS695 home page

Introduction

The goal of this assignment is to understand how hypervisors such as QEMU utilize the Linux KVM API to provide efficient hardware-assisted virtualization.

In this assignment, we will understand and 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.

Before you begin this assignment, you must familiarize yourself with the basics of the KVM API, from resources such as this LWN.net article. 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.

Part A: Understanding the "Hello, world!" example

In this part, you will first understand the KVM "Hello, world!" example, by reading through the code and answering the various questions posed below.

You need not submit the answers to any of the questions asked in this part. You will, however, be asked to answer some of these questions during the viva. You may need to place some `printf` statements in the code in order to answer some of the questions. Please leave these statements commented out in your final submission, so that you can uncomment and execute them to answer your viva questions.

For this assignment, it is enough for you to understand the code within the files **`kvm-hello-world.c`** and **`guest.c`**. The C program **`kvm-hello-world.c`** acts as a simple hypervisor, and runs the code within the file **`guest.c`** in a sandbox using the KVM API, much like how a real hypervisor like QEMU runs a full-fledged guest OS. You can compile the hypervisor **`kvm-hello-world.c`** using the command **`make kvm-hello-world`** to produce an executable of the hypervisor, with the guest code embedded within it. You can then provide different commandline arguments to run the executable in various CPU modes. For example, **`./kvm-hello-world -l`** runs the executable in long (64-bit compatibility) mode. For this assignment, it is enough for you to focus on the long mode. You may want to read up online to get a high-level understanding of what these CPU modes mean.

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.

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 in the next part.

Part B: Adding new hypercalls

In this part, you will add new hypercalls from the guest to the hypervisor. You will use the **`in/out`** I/O instructions cause guest exits to the hypervisor, and communicate information across the guest and the hypervisor by reading/writing values into I/O ports. You will then leverage this hypercall mechanism to add some useful functionality to your simple guest.

We will first provide you with some help in writing assembly code at the guest to correctly invoke the privileged **`in/out`** I/O instructions. The following inline assembly code is a modification of what exists in the current guest: it can pass across a 32-bit unsigned integer value (instead of an 8-bit value) over a certain I/O port. You can use this code snippet in your solution as required.

```
static inline void outb(uint16_t port, uint32_t value) {
  asm("out %0,%1" : /* empty */ : "a" (value), "Nd" (port) : "memory");
}
```

Note that you can pass any 32-bit value in this manner between the guest and hypervisor. This 32-bit value can be an integer, the address of an array (i.e., a **`char *`** pointer), or anything you want it to be. You have already seen that the **`out`** instruction will trigger a KVM_EXIT_IO in the hypervisor, with the I/O direction set to KVM_EXIT_IO_OUT. The 32-bit value you have written will be available within the VCPU runtime buffer shared between the hypervisor and KVM, at a certain specified offset, which you have seen as well. You must carefully dereference the value within the buffer and interpret it correctly, based on what you have written at the guest.

Now, how do you pass information back from the hypervisor to the guest? You can use the **`in`** instruction for this purpose. This instruction reads a 32-bit value from a specified port number. The following C function takes a port number as an argument, invokes the **`in`** instruction on the specified port, and returns a 32-bit value fetched on the input port. The inline assembly code within this function stores the port number argument in the EDX register, invokes the **`in`** instruction, and stores its return value in the EAX register. The C compiler then uses the value in the EAX register as its return value when returning from the C function, as per GCC conventions. You may use this code snippet in your solution as required.

```
static inline uint32_t inb(uint16_t port) {
  uint32_t ret;
  asm("in %1, %0" : "=a"(ret) : "Nd"(port) : "memory" );
  return ret;
}
```

The **`in`** instruction triggers a KVM_EXIT_IO exit in the hypervisor, with the I/O direction set to KVM_EXIT_IO_IN. Now, the hypervisor can write whatever 32-bit value it wishes to send to the guest into the VCPU runtime buffer shared between the hypervisor and KVM, at the suitable I/O offset, and this value will be automatically available to the guest as the return value of the **`in`** instruction. That is, the same memory buffer that was used to pass values from the guest to the hypervisor can also be used to send information back to the guest when the exit is caused by the **`in`** instruction. Note that you must be careful with your pointer arithmetic when reading and writing values into these memory locations.

After you understand how to pass 32-bit values back and forth between the guest and the hypervisor, you must implement the following functions within your guest, invoke them suitably, and demonstrate their correctness. (You are free to choose different I/O port numbers for the various hypercalls below.)

- The function **`printVal(uint32_t val)`** should print to the screen the 32-bit value given as argument.
- The function **`getNumExits()`** should return the number of exits incurred by the guest since it started. This count of exits should be maintained at the hypervisor, i.e., you should count the number of exits seen in your userspace KVM-based program. When the guest invokes this function, it should result in a hypercall to the hypervisor. Note that this function should not print the number of exits directly to the screen from the hypervisor, but should return the value back to the guest. How will you inspect the value returned by this function in the guest, given that you do not have access to C library functions like `printf`? Well, you can use the `printVal` function developed above to print the return value via a guest exit again. That is, the code in your guest should look as follows.

  ```
  uint32_t numExits = getNumExits();
  printVal(numExits);
  ```

  Your count of exits can include the exit that was required to process this particular hypercall as well.
- The function **`display(const char *str)`** should print the argument string to the screen via a hypercall. However, you should not incur one exit per character, as done in the code seen in part A. Instead, you must print the entire string in just one guest exit. How can you do this? Instead of passing the string character-by-character via **`out`**, you may pass the virtual address of the character buffer via the **`out`** instruction. Once you get this address in the hypervisor, you must access the guest memory at this guest virtual address, and print out the entire string in one shot. To do all this right, you must clearly understand how to access the guest memory area from the hypervisor correctly. You can demonstrate the correctness of this function by calling `getNumExits` before and after this system call, and verifying that you only incurred one exit to print the entire string.

Part C: Filesystem hypercalls

In this part of the assignment, you must implement hypercalls to emulate some of the file system functionality, e.g., opening, reading, and writing files. That is, the filesystem related functions in the guest must cause an exit to the hypervisor. The hypervisor must then use regular system calls on files to emulate the operations

requested by the guest, returning suitable values back to the guest. For example, when the guest asks to open a file, the hypervisor must open the file and return a file descriptor back to the guest. When the guest asks to read a file, the hypervisor must read the file and return the read data back to the guest. You are expected to support open/read/write functionality at the bare minimum, though you may choose to support more file operations such as seek. While you have some freedom in choosing the design and implementation of the hypercalls, it is desirable that the semantics of the hypercalls be as close to that in a real system. A few points to keep in mind.

- You may decide whether you will have only a single file **open**, or whether you wish to support multiple open files. If you wish to support multiple open files, you must return some sort of a file descriptor from the hypervisor to the guest, so that the guest may pass the file descriptor to identify the file in future read/write hypercalls.
- In the **read** hypercall, you must return the buffer of read data back to the guest. You may then use a hypercall like `display` developed in part B to print it out to screen and verify that you correctly read the file data.
- In the **write** system call, you may flush the file in the hypervisor and actually verify that the data you have written from the guest is correctly seen on the disk.
- You need to carefully think about how you will pass multiple arguments in these hypercalls. For example, if you are aiming for Linux-like semantics for the read hypercall, you may need to pass the file descriptor, the address of the buffer into which read data must be stored, and the number of bytes to be read. Since you can pass only one value to the hypervisor, you will need to think about how to pass multiple arguments. Do you wish to pass arguments one after the other, and execute the read once all arguments have arrived at the hypervisor? Or maybe you can place the arguments in a structure and pass the address of the structure in a single hypercall? There are many ways to solve this problem, and you can come up with your own clever solution.
- As with any implementation, you may want to start simple (single open file, fixed number of bytes to read/write, and so on), and slowly add complexity to your implementation to make the hypercalls more closer to real life. You may also make some reasonable assumptions, e.g., a sensible limit on the number of open files or the number of bytes read/written.

This code on GitHub, along with its explanation, provide a sample example of how you would implement filesystem-related system calls/hypercalls in KVM. Feel free to explore this code on your own to get some ideas.

Your grade in this assignment will depend on how complex and realistic your implementation is. For example, we will watch out for whether you support multiple open files, whether you can correctly pass multiple arguments in your hypercalls (to make them close to real life), and other such features.

Submission and grading

You must solve this assignment individually. You must submit your modified files `guest.c` and `kvm-hello-world.c`. You should also submit the `Makefile` if you needed to change it for some reason. Create a tar-gzipped file of your submission, with the filename being your roll number, and upload it on Moodle. Grading this assignment

will involve a demo and viva with the instructor/TA.

Good luck!

Back to CS695 home page