

Simple Hypervisor

2026 – 2027



Mathieu Bacou¹
mathieu.bacou@telecom-sudparis.eu
Télécom SudParis
Institut Polytechnique de Paris

I. Introduction

In this lab, you will familiarize yourself with  Linux's KVM API by working on a minimal hypervisor.

I.1. Objectives

First, you will implement the bare minimum of working with KVM and its abstractions to hardware-assisted virtualization. Then, you will add paravirtualization features to an existing codebase of a hypervisor that can boot a real Intel x86_64 executable.

The purpose of this lab is to showcase the virtualization techniques of different computing resources, to get used to the double level of memory virtualization, and to practice paravirtualization.

Info and warnings, questions and code questions.

In this lab subject, you will find:

- info boxes (such as this one) that will give you additional information about what you are doing;
- **warning boxes** that you **must** read to avoid pitfalls and common problems;
- **questions** that you answer with human-readable text;
- **code questions** that you complete by writing code in your source files, or by providing the right command.

Turning in the lab.

Your turn-in should include:

- for **questions**, a report with their answers and other relevant information;
- for **code questions**, the source files that you created or modified **without the compiled content**.

Please upload to Moodle a `.tar.gz` archive.

¹Acknowledgements to Jean-François Dumollard for its beta-testing!

I.2. Prerequisites

- **An Intel x86_64 machine** (sorry no ARM-based Macs);
- An up-to-date Linux kernel (WSL2 may work, but is untested);
- Common OS development tools (git, make, gcc...);
-  [NASM — The Netwide Assembler](#).

This lab is going to make you start an Intel x86_64 virtual machine. Although KVM, like the Linux kernel, supports multiple architectures, some parts of the lab are written for x86_64 (especially the guest boot code used in Section III and forward).

Obviously, you need a real Linux kernel for this lab, that is compiled with support for KVM (which is probably the default for any distribution).

There is some assembly code, used to bootstrap the virtualized Intel x86_64 guest in Section III, that is compiled using NASM. Install it using your package manager; its package name is most probably just `nasm`.

II. Minimum KVM-based hypervisor

To begin with, you will discover KVM APIs to work with elements of a virtual machine (VM): vCPUs, memory mappings, etc. The goal is to make the minimum required work to effectively run code inside a VM.

The guest code that will run inside the VM is given in Listing 1. Copy-paste it as-is to your new C source file.

```
/* Guest code.
 *
 * - add registers al and bl (initialized by the SHV code) into al
 * - convert the result digit into ASCII by adding '0'
 * - output it to serial port 0x3f8, followed by a new line
 * - halt
 */
static const uint8_t guest_code[] = {
    0xba, 0xf8, 0x03, /* mov $0x3f8, %dx */
    0x00, 0xd8,      /* add %bl, %al */
    0x04, '0',       /* add '$0', %al */
    0xee,            /* out %al, (%dx) */
    0xb0, '\n',      /* mov $'\n', %al */
    0xee,            /* out %al, (%dx) */
    0xf4,            /* hlt */
};
```

Listing 1: Guest code.

About the guest code.

As you can see, this guest code in Listing 1 is provided as a C array of bytes. Its content are **instructions** for an Intel x86 machine. In this part, you will not compile any guest code: you will directly load these instruction bytes into the VM's memory and make it run them.

II.1. KVM virtual machine setup

Code question C1

First, we will check you can use KVM. In the `main` function, open the KVM device at `/dev/kvm` (read-write); we'll call the resulting file descriptor the "KVM FD".

Then, issue on it the `ioctl` `KVM_GET_API_VERSION` (with `NULL` as a parameter), and check the result is 12. If this is not the case, error out.

How do I do all of this?

This subject will give you all the KVM-specific information you need, but having  [The Definitive KVM \(Kernel-based Virtual Machine\) API Documentation](#) at reach would be a good idea.

To get help on other topics, such as how to use the `ioctl` system call, call the `man` command! Most KVM-related operations go through `ioctls`, either on the file descriptor opened on `/dev/kvm`, or on the file descriptors opened by the next `ioctls`.

For now, at least do not forget to `#include <linux/kvm.h>!`

Handling errors.

If you are not yet familiar with the facilities provided by `err.h`, now is the time! They are used to interact with the global error number variable `errno` (see `man 3 errno`).

When you just want to exit your program in error if some system call failed², you can use `err(int status, char *fmt, ...)`: your program will print the string formatted with the remaining arguments (like `printf`), followed by the error message of the failed system call, and then exit with the provided status code. There is also `errx(int eval, char *fmt, ...)`, that behaves similarly but without trying to print the system call error message, which is useful when you need to handle another kind of error (for instance an unexpected value in your program).

Example:

```
1 int fd;
2 char *fn = "toto";
3 if ((fd = open(fn, O_RDWR)) == -1)
4     err(EXIT_FAILURE, "failed opening file %s", fn);
5 if (fd != 3)
6     errx(42, "I don't like file descriptor 3");
```

Listing 2: Example usage of `err` and `errx`.

Code question C2

Create a new KVM virtual machine by issuing the `ioctl` `KVM_CREATE_VM` on KVM's FD, with the parameter 0 (it asks for a "default" VM). Keep the return value: this is your "virtual machine file descriptor" (VM FD).

²Technically, if exactly the last function call that sets `errno` fails.

Next we will set a memory region for the guest's content: its code. To achieve this, you have to allocate memory, copy the code to it, then tell KVM to map it in your guest. Because our guest is so simple, without any stack nor data, you only need to allocate a single memory page.

Code question C3

Allocate a page worth of memory (size of 4096 B, or 0x1000) using `mmap` (make the mapping shared): this will be the guest's memory. Then, copy the guest code to it.

Finally, give this memory to the guest as a *memory region* by issuing the `ioctl` `KVM_SET_USER_MEMORY_REGION` on the VM FD. This `ioctl` takes as parameter the address to a `struct kvm_userspace_memory_region` that is given in Listing 3. In this structure, `guest_mem` should be the memory you allocated to the guest just before.

```
/* slot is just an ID for memory regions managed by KVM.
 * guest_phys_addr is the address where the memory is mapped in the
 * guest's memory space.
 * guest_mem is the address in the hypervisor's memory space, of the
 * allocated memory.
 */
struct kvm_userspace_memory_region region = {
    .slot = 0,
    .guest_phys_addr = 0x1000,
    .memory_size = 0x1000,
    .userspace_addr = (uint64_t)guest_mem,
};
```

Listing 3: `struct kvm_userspace_memory_region` to map memory in the guest.

Question Q1

Explain the memory mapping that you set in C3, in terms of GVA, GPA, HVA and/or HPA, for any relevant level. *Hint*: KVM uses SLAT (if available).

Code question C4

Create a virtual CPU (vCPU) for the VM, by calling the `ioctl` `KVM_CREATE_VCPU` on the VM FD. The parameter is the ID of the vCPU; this is the first one, so give 0. Keep the return value: this is the vCPU file descriptor (vCPU FD) to get its running state.

To interact with the vCPU, the interface is based on mapping the vCPU FD in memory. This is similar to mapping a regular file, except the content is dynamically provided by the kernel under the shape of a `struct kvm_run`.

Code question C5

First, get the size of the `struct kvm_run` mapping: this is the return value of the `ioctl` `KVM_GET_VCPU_MMAP_SIZE` (NULL parameter) on **the KVM FD**. You should check that it is at least as big as the size of a `struct kvm_run`, otherwise error out.

Then, map the vCPU FD in memory.

Now we need to set the initial state of the vCPU's registers.

Code question C6

Get the current state of the vCPU's special registers by issuing the `ioctl KVM_GET_SREGS` on its file descriptor; its parameter is the address to a `struct kvm_sregs` that will be filled with the current values.

Then, modify this structure to set its fields `cs.base` and `cs.selector` to 0.

Finally, set the vCPU's special registers by issuing the `ioctl KVM_SET_SREGS` on its file descriptor, which also takes as parameter the address of your `struct kvm_sregs`.

About Intel x86_64 initialization rituals.

Here, we are setting x86_64-specific registers that deal with segments. [Segmentation](#) is “an obsolete memory protection technique [...] it is no longer supported.” However in this part, your guest will execute code in [real mode](#), so segmentation isn't really obsolete for you.

Anyways, Intel x86_64 initialization rituals necessitate correctly initializing the CS register, while keeping other values set by KVM intact (thus the read-modify-write operation in C6). As you will see in question C7, there is another ritual: `rflags`'s second bit must be set. How Intel expects you to discover that, as [the bit is marked as “reserved”](#), is unknown. Do it, don't ask.

Code question C7

Initialize the general purpose registers of your vCPU. Start from a `struct kvm_regs` that is initialized to `{0}`. In it, set the following fields:

- `rip`: this is the initial value of the instruction pointer, or in other words, the address in your VM's memory of the first instruction that it will execute;
- `rax` and `rbx`: the registers that your guest code will add together before outputting the result (see Listing 1), the sum of their values should stay **between 0 and 9** because the guest code converts to ASCII, and outputs a single digit;
- `rflags`: **must be set to 0x2**.

Finally, set the VM's registers with a call to the `ioctl KVM_SET_REGS` on the vCPU FD, that takes the address to the `struct kvm_regs` as parameter.

II.2. Running the KVM virtual machine

When using KVM, the hypervisor implements a *control loop*. It starts with asking KVM to run the VM (or rather, one of its vCPUs) with the `ioctl KVM_RUN` (no parameters) on a vCPU FD. This call is blocking. When it returns, it either means:

- the VM executed an operation that KVM could not fulfill itself (e.g., I/O operations);
- or it encountered a triple fault or a hardware exception;
- or it simply halted;
- or KVM itself had an error.

So when the `ioctl` returns, the job of your hypervisor is to handle these cases, and to decide whether the VM should continue running.

To help your hypervisor do its task, the `struct kvm_run` you mapped in C5 is updated with relevant information. It can determine the exit reason by looking at the value of its field `exit_reason`. Here are a few reasons that you must handle:

- `KVM_EXIT_HLT`: the VM halted (this is the last instruction of your guest code, see Listing 1);
- `KVM_EXIT_IO`: the VM tried to do an I/O operation using the `in` or `out` instructions,
 - the direction (in or out) is given by the field `io.direction` of the `struct kvm_run` which can take the value `KVM_EXIT_IO_IN` or `KVM_EXIT_IO_OUT`,
 - the destination port is given by the field `io.port` (your guest code outputs to port `0x3f8`, which is the standard [COM1 serial port](#)),
 - the address of the exchange buffer (data to output, or where input data must be stored) is given by `io.data_offset`, which is **an offset from the address of the `struct kvm_run`**,
 - the guest has provided (for output) or is expecting (for input) `io.count` elements of `io.size` bytes each;
- `KVM_EXIT_FAIL_ENTRY`: KVM failed entering the VM (this is a fatal error);
- `KVM_EXIT_INTERNAL_ERROR`: KVM encountered a fatal error.

Code question C8: The KVM control loop

Write the KVM control loop: in a loop, call the `ioctl KVM_RUN` on the vCPU FD, then properly handle the exit reasons listed above.

For `KVM_EXIT_IO`, an easy way to handle `out` is to call the function `fwrite`.

Congratulations!
You successfully implemented a
(very) minimal KVM hypervisor!

Question Q2

How could you run a VM with multiple vCPUs?

III. Towards a real guest

While your code does implement the basics of a hypervisor, it remains limited in that it only runs a fixed code embedded in its source code. In this section, you will familiarize yourself with an more advanced hypervisor that can load an arbitrary ELF program. Then in Section IV, you will handle more operations from your guest by implementing *hypercalls*.

III.1. Preparation

Fetch the provided code base as instructed in Listing 4.

```
curl https://www-public.imtbs-tsp.eu/~mbacou/cours/practicals/\
simple-hv/simple-hv.tgz | tar --extract --gzip
```

Listing 4: Get the code base.

You should get a directory named `hypercalls` with the following content:

- `Makefile`: you will type `make run` to run your hypervisor;
- `guest/`: for content that runs inside the VM
 - `guest/app/`: the guest application that will run inside the VM,
 - `guest/app/main.c`: the guest application's code, **initially with an empty `main`**;
 - `guest/boot/`: for content that bootstraps the VM up to the guest application,
 - `guest/boot/boot.asm`: the assembly code that boots the VM and runs the guest application;
- `shv/`: for the hypervisor content
 - `shv/hypercall.c`: where **you will write code to handle hypercalls**,
 - `shv/main.c`: the start function of the hypervisor, where **you will handle passing arguments to the guest application**,
 - `shv/vmm.c`: the code code of the hypervisor, **where you will redirect hypercalls handling to code in `hypercall.c`**,
 - `shv/*.h`: headers that define potentially interesting macros.

You should not have to interact with other source files, but *feel free to explore!*

III.2. Testing

Let's check you can work on this codebase: run `make run` to compile everything and run the hypervisor! This first try should end with:

```
1 shv: unhandled exit_reason 0x2: @rip=0x8097
2 make: *** [Makefile:24: run] Error 1
```

What you are seeing, is the unhandled operation from your guest, that tries to issue a system call. In the next section, you will handle such system calls as hypercalls!

Compiling the guest application

The guest application is C code that is compiled against the  `musl libc`, an alternative to the classic  `GNU C library`. More precisely, the first invocation of `make run` downloads and compiles a custom version, modified to work in a VM of your hypervisor (it simplifies the program exit epilogue and fixes returning from a syscall).

Next compilations will be much faster.

Question Q3

Look at `create_vm` and `launch_vm` in `shv/vmm.c`, and confirm that their content mostly matches your simple hypervisor from Section II.

Also observe `shv/main.c`. It starts with loading ELF executables into guest memory using `load_sections` (you do not have to look at the code of the ELF loader), and eventually launches the VM with `launch_vm`. But before that, it calls `memory_map_range`, which is just a macro over `memory_map`. What does `memory_map` do, and at what memory virtualization level(s) does it map memory?

Finally, explain the memory mapping of the guest by describing what lies at which address; be explicit about the memory spaces. `shv/memory_map.h` will help you.

IV. Hypercalls

In this new hypervisor, you will:

1. handle system calls issued by the guest by executing them on the host, and emulating the expected result in the guest;
2. pass arguments to the guest application;
3. implement a hypercall for paravirtualized memory allocation, via a custom guest system call.

System calls are hypercalls?

In this lab, your guest does not have any kernel running, and it sends its system calls to the hypervisor. This paravirtualization of system calls makes them hypercalls, and the term “hypercalls” will be used in the rest of the lab.

IV.1. Handling `exit`

We will start by implementing the basic infrastructure for handling system calls, starting with `exit`.

In this lab, there is not a real kernel in the guest. However, an important piece of privileged code that *is* set, is the system call handler. Instead of implementing system calls in the guest, the handler executes an `out` instruction on port `0x118` (given to you by `shv/vmm.h` as `HC_IO_PORT`), that will trigger a VM exit to your hypervisor. No other instruction will ever `out` to this port, so you can check for these properties and be sure you received a hypercall from the guest.

The paravirtualized system call handler.

This custom system call handler is defined in `guest/boot/boot.asm`, and installed during bootstrap just like a normal kernel would for its regular system call handler.

What about KVM_EXIT_HYPERCALL?

KVM's API includes an exit reason named `KVM_EXIT_HYPERCALL`. However, as “strongly recommended” by [KVM documentation](#), userspace VMMs should use `KVM_EXIT_IO`, which is the case here (triggered by `out`). KVM rather considers hypercalls to be triggered by [vmcall and vmmcall instructions](#).

So, do not try to handle the wrong exit reason!

Code question C9

In `shv/vmm.c`, handle the exit reason for a hypercall by calling `hypercall` (from `shv/hypercall.c`) with the required arguments. You can identify this exit reason as explained above (with the `struct kvm_run` named `vcpu_run`); refer to C8 for details.

Ignored hypercalls.

From this point on, running `make run` will succeed. Messages about ignored hypercalls 158 and 218 are safe to ignore; they belong to the threading API that the `musl` libc always uses, even though your own code does not.

The system call `exit` is issued at the end of the guest application (even if not present in its code, because the `libc` always adds it as an epilogue to the `main` function).

Code question C10

In `hypercall`, handle `exit` (and `exit_group`) where you are instructed to. You just have to exit with the exit code provided by the guest, which is the first argument of the system call. You can access arguments 1 through 6 of a hypercall using `ARG1` to `ARG6` (defined in `shv/hypercall.h`).

Your guest application exits with the return code `0x42` (66 in decimal), so exiting with the same code will show as an error for `make run`.

Question Q4

Explain what happens in terms of privilege rings when the guest application issues a system call in your hypervisor. You do not have to read any code to answer.

IV.2. Handling file operations

Let's implement paravirtualized file operations support! The goal is to handle `read`, `write`, `open` and `close`, by redirecting them to a dedicated folder on your host.

Code question C11

Complete the guest application's `main` function with calls to test the system calls listed in Table 1. For instance, you could open and read from a file, then open and write the same content to another file, before closing them.

System call	Number	Purpose
<code>read</code>	<code>0x0</code>	Read bytes from a file descriptor into a buffer.
<code>write</code>	<code>0x1</code>	Write bytes from a buffer into a file descriptor.
<code>open</code>	<code>0x2</code>	Open a file, returning a file descriptor.
<code>close</code>	<code>0x3</code>	Close a file descriptor.

Table 1: File operations to handle as hypercalls.

Code question C12

Implement support for paravirtualized file operations listed in Table 1. Refer to their manual pages for details on parameters.

You should also know that the argument `memory` of `hypercall` is a pointer to the memory of the guest. Moreover, the return value of the hypercall is set by assigning to `HC_RET`.

IV.3. Passing arguments to the guest application

The guest application behaves like an ELF program normally loaded by a kernel: through the `musl` C library, its `main` function expects two arguments, `argc` and `argv`, where `argv` is an array of the strings of arguments, and `argc` is the number of arguments in `argv`. Your goal is to pass arguments received by your hypervisor's `main` to the guest application.

Code question C13

Complete the guest application's `main` function to print the number of arguments `argc`, and the arguments themselves listed in `argv`.

Classic printing routines are not available!

You cannot use the classic printing routines (`printf`, etc.) in your guest application, because they use the Thread Local Storage (TLS) which is not properly initialized by your hypervisor nor by the guest.³ You are provided with the function `writeln` to write a line to the console, and `itoa` to convert an integer into a string.

Following the Application Binary Interface (ABI) of Linux on Intel x86_64:⁴

- `argc` (a 64bit integer) is put at the address of the base stack pointer, i.e., just above the stack (given by `APP_STACK_GPA` in `shv/main.c`);
- `argv` (an array of string pointers) is put just after `argc`.

³Do you remember the ignored system calls in C9?

⁴Technically, the  [System V ABI for Intel386 Architecture](#).

Code question C14

In `shv/main.c`, pass arguments received by `main` to the application guest, but **start with the 4th one** (index 3), because the first ones are used by your hypervisor itself (the first one is just its name, and the second and third ones are paths to the bootloader and guest application, respectively).

Note there is a **free mapped page** above the guest application's base stack pointer, to let you achieve this, including copying the value of the arguments!

You can test your work by running `make run arg1 arg2...`

IV.4. Paravirtualized memory allocation

The goal is to implement `void *pvmmmap(void *addr, size_t len, int fd, off_t offset)`, a simplified, paravirtualized `mmap`. In other words, `pvmmmap` must create a new mapping in the guest application's memory space, backed by anonymous memory from the host, or by content taken from the given file descriptor.

Its arguments are:

- `void *addr`: address of the new mapping, it may be `NULL` in which case the implementation must find a free memory area;
- `size_t len`: length of the mapping;
- `int fd`: file descriptor backing the mapping, or `-1` for anonymous memory;
- `off_t offset`: offset in file content backing the mapping.

And it returns the address of the new mapping, which may not be the same as `addr`.

First for convenience, in the `guest/app/main.c`, add the hypercall wrapper given in Listing 5.

```
#define SYS_PVMMAP 511

void *pvmmmap(void *addr, size_t len, int fd, off_t offset) {
    return (void *)syscall(SYS_PVMMAP, addr, len, fd, offset);
}
```

Listing 5: `pvmmmap` hypercall wrapper.

Code question C15

Add test code in the guest application for an anonymous mapping, i.e., calling `pvmmmap` with its 3rd argument set to `-1`. Make sure to actually use the mapping in the guest application (write to, and read from it).

Code question C16

Implement `pvmmmap` (in `hypercall` in `shv/hypercall.c`) for an anonymous mapping. You can use `get_pages` (from `shv/memory.c`) to get new guest physical pages, and `memory_map` or `memory_map_range` to map them in the guest application.

Code question C17

Add test code in the guest application for a file-backed mapping, i.e., calling `pvmmap` with a valid 3rd argument like the one returned by `open`.

Code question C18

Complete your `pvmmap` implementation for a file-backed mapping. Here, you should use the `mmap` flag `MAP_FIXED` when setting the mapping in the host.

Congratulations!
You completed a more complex
hypervisor with paravirtualized
capabilities!

IV.5. Bonus: dynamic memory allocation to the guest

The function `get_pages` gives pages that are already allocated in the host to the guest (and already mapped in the Extended Page Table). If it reaches the end of the preallocated area (given by the first upper invalid page `MEMORY_TOP_GPA`), it crashes the hypervisor with an “OOM” error. But you could also allocate more memory in the hypervisor, then map it in the VM by setting a new memory region, like in C3.

Code question C19: Bonus question

Fix `get_pages` to allocate more memory when it reaches the end of the memory area currently allocated to the VM.

V. Conclusion

In this lab, you worked on a hypervisor that uses the KVM API to run a simple application in a virtual machine. Your hypervisor provides a few paravirtualized facilities.

To go further, you could try implementing support for multiple vCPUs in your hypervisor. However supporting multiple CPUs in the guest application — threading — is more complicated, and would necessitate properly initializing the hardware.

To make your hypervisor more featureful, you could implement checkpointing and restoring a virtual machine: upon receiving a signal, your hypervisor must dump any state needed to represent the VM to a file. Then, providing this file to your hypervisor must allow it to start back the VM in this exact same state. Furthermore, if the VM state is sent to another instance of your hypervisor, you can implement live migration!

Finally, a very high-end goal would be to achieve booting a real Linux kernel — at least with the minimum required feature set.