Hypervisor from Scratch in Linux

Mehul Varma
mehulvarma@tamu.edu
Texas A&M University

Reid O'Boyle
oboylere@tamu.edu
Texas A&M University

KEYWORDS

Hypervisor, Virtualization, Virtual Machine

ACM Reference Format:
Mehul Varma and Reid O'Boyle. 2022. Hypervisor from Scratch in Linux. In *N/A, May 03, 2022, College Station, TX.*, **??** . https://doi.org/

1 INTRODUCTION

A hypervisor is software used to run virtual machines on a host. The hypervisor software isolates resources from the operating system for exclusive use on the virtual machine. The hypervisor is critical for the management, creation, and destruction of virtual machines. Hypervisors manage the resources it has available and distribute the resources to guests. Guests are users running a virtual machine on a hypervisor. Hypervisors use the idea of virtualization to run a second operating system concurrently with your system. Virtual machines give users the perception that they are running their own isolated machine. Hypervisors isolate virtual machines and the resources used from the underlying system. Virtual machines are useful for many reasons such as running a Linux operating system on Windows or vice versa. They are also useful in cloud computing because they can isolate portions of a server's operating system for a user based on a user's needs. This also allows for multiple users to utilize a virtual machine on the same server each having their own isolated resources.

In the next section, we will explain how we implemented virtualization and a hypervisor on a Linux system.

ACM ISBN …$
https://doi.org/

2 BACKGROUND

There are a few virtualization terminologies that are used throughout the development.

- `Virtual Machine Monitor (VMM)`: This is the hypervisor software itself. The VMM is the host for the guest software that intends to run on it. It acts as a host for them and has control over the processes and the platform hardware to initialize the guest software. To provide for the guest software, it abstracts the hardware as a virtual processor so it directly executes on a logical processor.
- `Virtual Machine Extensions (VMX)` : This refers to Intel's instruction set that is used by the guest software to simultaneously share the x86 processor.
- `VMX Root / VMX Non-root operation` : The VMM runs using VMX root operations and the guest software that runs on VMM uses the VMX non-root operations. In the case when the guest software needs to perform any root operations, a VMX transaction takes place; this is similar to a system call in a user and kernel environment.
- `VIRTUAL-MACHINE CONTROL STRUCTURE (VMCS)`: This is a structure in the memory that the processor uses to keep track of the different VMX transactions taking place. It stores the guest and host eip. VMCS is managed using the VMCS pointer which is part of the processor state.

Some Common Virtualization Instructions are as follows

- VMXON - Enable VMX
- VMXOFF - Disable VMX
- VMLAUNCH - Start/enter VM
- VMRESUME - Re-enter VM
- VMCLEAR - Null out/reinitialize VMCS
- VMPTRLD - Load the current VMCS
- VMPTRST - Store the current VMCS
- VMREAD - Read values from VMCS

- VMWRITE - Write values to VMCS
- VMCALL - Exit virtual machine to VMM
- VMFUNC - Invoke a VM function in VMM without exiting guest operation

3 IMPLEMENTATION OF HYPERVISOR IN LINUX

In this section, we will summarize the steps we took to implement a hypervisor in Linux.

3.1 VMX Operation Flow

- To start the VMM we first have to execute the VMXON operation which enters the VMX mode. Once that is done, the VMM software is running.
- VMCLEAR is then used to clear existing guest's VMCS and make it ready for new guest software.
- To start guest software, VMLAUNCH and VMRESUME are used.
- VMWRITE is used to write the parameters of the new guest to the VMCS
- VMCALL is used to exit back to the VMM.
- To shut down the VMM, VMXOFF is used.

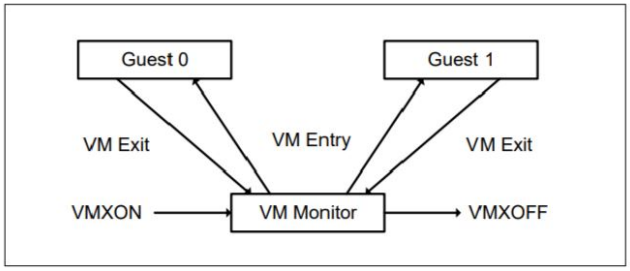


Figure 1: Interaction of a Virtual Machine and Guests, via Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 3 (https://www.intel.in/content/www/in/en/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html).

3.2 Creating a Kernel Module

To begin, the hypervisor development includes a kernel module. At the bare bones, the hypervisor includes a c file that runs module_init() and module_exit(), and a Makefile to build the project. module_init() and module_exit() handle the loading and unloading of the kernel module. To start we just create methods my_init() and my_exit() which will handle all the operations needed to start the hypervisor and shut it down when loading/unloading the kernel module. We then use module_init(my_init() to run my_init() during the kernel module initialization. Later on, we will use my_init() to execute all the steps in starting up the hypervisor. For now, we will just print "Hello world" and "Goodbye world" to show that our my_init() and my_exit() are run during the kernel module load and unload. This process is shown below.

```
mehul@mehul-XPS-15-9560:~/Desktop/TAMU/Spring2022/CSCE613/Hypervisor$ sudo rmmod hypervisor
mehul@mehul-XPS-15-9560:~/Desktop/TAMU/Spring2022/CSCE613/Hypervisor$ sudo insmod hypervisor.ko
mehul@mehul-XPS-15-9560:~/Desktop/TAMU/Spring2022/CSCE613/Hypervisor$ sudo rmmod hypervisor
mehul@mehul-XPS-15-9560:~/Desktop/TAMU/Spring2022/CSCE613/Hypervisor$ sudo insmod hypervisor.ko
mehul@mehul-XPS-15-9560:~/Desktop/TAMU/Spring2022/CSCE613/Hypervisor$ dmesg | tail
[11830.261015] Goodbye world.
[11831.257609] Hello world.
[11832.187724] Goodbye world.
[11833.044833] Hello world.
```

Figure 2: Basic kernel Module Execution

3.3 Setting up VMXON

The first step the hypervisor does when creating a VM is to enter the VMXON operation. VMX (Virtual Machine Extensions) is the CPU flag that defines support for virtualization in the processor. Before running VMXON, we have to check if VMX is supported on the machine. This is done by running the cpuid assembly instruction. The cpuid [2] instruction returns data containing the details of the processor and what features it supports such as VMX. There are more steps before VMXON can be run. The first is setting the VMXE bit in control register # 4 The first step the hypervisor does when creating

13	VMXE	Virtual Machine Extensions Enable	see Intel VT-x x86 virtualization.

Figure 3: Control Register 4 Bit 13(https://en.wikipedia.org/wiki/Control_register#CR4)

a VM is to enter the VMXON operation. VMX (Virtual Machine Extensions) is the CPU flag that defines support for virtualization in the processor. Before running VMXON, we have to check if VMX is supported on the machine. This is done by running the cpuid assembly instruction. The cpuid [2] instruction returns data containing the details of the processor and what features it supports such as VMX. There are more steps before VMXON can be run. The first is setting the VMXE bit

in control register 4 [3].
VMX can also be blocked by bits in the IA32_FEATURE _CONTROL in the module-specific registers (MSR). Specifically, we have to set bits 0 and 2. Bit 0 is a lock bit that needs to be set for VMX, and bit 2 needs to be set for VMX operations outside safer mode extensions (SMX). To set these bits we check if bit 0,1,2 is equal to 101, and if not we set them manually. Next, certain bits in control registers 0 and 4 will cause VMX to fail if they are set to the undesired values. So we have to set/clear those bits based on their current status. Lastly, we need to allocate 4 Kb of memory used by the processor to support VMX operations. We will pass the address of this memory to VMXON when we call it. We use kzalloc() to allocate kernel memory for the regions and set the values to zero. We have to set the first 31 bits in this buffer to the virtual machine control structure (VMCS) revision identifier. Different processors use different revision identifiers to specify how the data is stored in the buffer. The revision identifier ensures that data in the VMCS region is read correctly. Now we are finally able to call VMXON and upon success, VMX is enabled.

3.4 Setting up VMCS

About VMCS
A separate VMCS can be used for each virtual machine by the VMM. For machines with multiple processors which map to multiple logical processors, multiple VMCS can be used for those processors. VMWRITE and VMREAD are used to interact with a VMCS.

There are various states to the VMCS which can be summarized using the state machine. In the state machine, a cleared active VMCS can launch a new VM using VMLAUNCH, which can be loaded, unloaded, or cleared. The following are functionalities of the different commands used in the state machine

- VMPTRLD - Makes the VMCS Active and current.
- VMCLEAR - Changes the state from current to non-current. This is used before every VMPTLDR to make previous current VMCS non-current.
- VMLAUNCH - To launch the VMCS or VM defined by VMCS.
- VMRESUME - Used to launch again the previously launched VMCS. Used to launch the VM which is exit (VMEXIT) due to some reason.

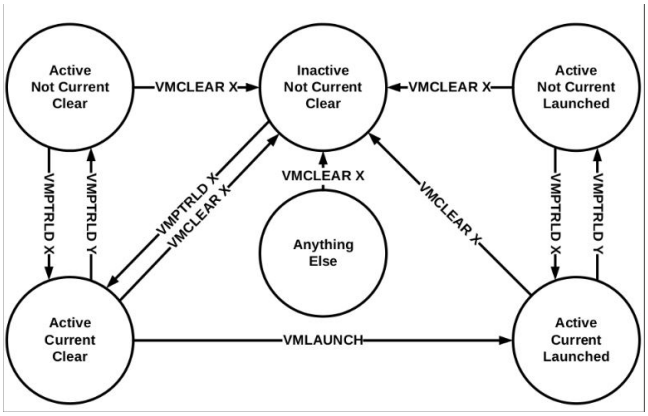


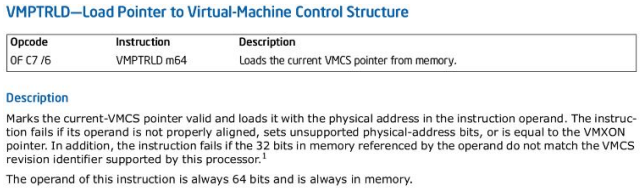
Figure 4: States of VMCS, via Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 3 (https://www.intel.in/content/www/in/en/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html).

Implementation
To set up the VMCS, we allocate a 4kb aligned and zeroed out memory using kzalloc() similar to the previous section. The VMCS is formatted using the byte offsets shown in the image. We see that the first 30 bits are the VMCS revision identifier which is initialized using the vmcs_revision_id() function. The 31st bit is the shadow VMCS indicator. The next byte is the VMX instruction abort indicator which is written by a logical processor in the case that a VMX abort takes place. The 3rd byte onward is the VMCS data that actually has everything else in it.

Byte Offset	Contents
0	Bits 30:0: VMCS revision identifier Bit 31: shadow-VMCS indicator (see Section 24.10)
4	VMX-abort indicator
8	VMCS data (implementation-specific format)

Figure 5: Byte offsets of VMCS, via Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 3 (https://www.intel.in/content/www/in/en/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html).

VMCS data is setup using the VMPTRLD function which is defined in the figure from the Intel manual. After finally setting up the VMCS, the VMCS data can



VMPTRLD—Load Pointer to Virtual-Machine Control Structure

Opcode	Instruction	Description
0F C7 /6	VMPTRLD m64	Loads the current VMCS pointer from memory.

Description

Marks the current-VMCS pointer valid and loads it with the physical address in the instruction operand. The instruction fails if its operand is not properly aligned, sets unsupported physical-address bits, or is equal to the VMXON pointer. In addition, the instruction fails if the 32 bits in memory referenced by the operand do not match the VMCS revision identifier supported by this processor.[1]

The operand of this instruction is always 64 bits and is always in memory.

Figure 6: The VMPTRLD function, via Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 3 (https://www.intel.in/content/www/in/en/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html).

be initialized starting from the 3rd byte. The data is set up into 6 parts.

1 Guest-state area: This has the VMX transition information. It consists of the guest and host eip along with the Registers state for the next VM entry.
2 Host-state area: This is a buffer that consists of the state of the processor after the root operations have been executed and exited using the VMexit.
3 VM-exit control fields: This is the information on how to execute VMexit. VM Exit control fields consist of the 32-bit vector for performing VMexit. VM-Exit Controls for MSRs have store count and store address for the MSR registers.
4 VM-execution control fields: This is the control for the processor behavior for the VMX non-root operation. Pin-based (asynchronous) controls, Processor-based (synchronous) controls, Exception bitmap, I/O bitmap addresses, Timestamp Counter offset, CR0/CR4 guest/host masks, CR3 targets, and MSR Bitmaps are initialized using MSR registers and field definitions.
5 VM-entry control fields: These are the fields to control the VM entries. Information includes what registers need to be loaded. VM entry controls include standard entry controls like the guest mode, debug controls and other Intel-specific registers. VM Entry controls for MSR include the MSR load count and the load address. VM Entry Controls for Even Injection allows for interrupts, traps, and exceptions to the guest VM.
6 VM-exit information fields: These are the fields that control the VMexit. Mainly used to store debugging information.

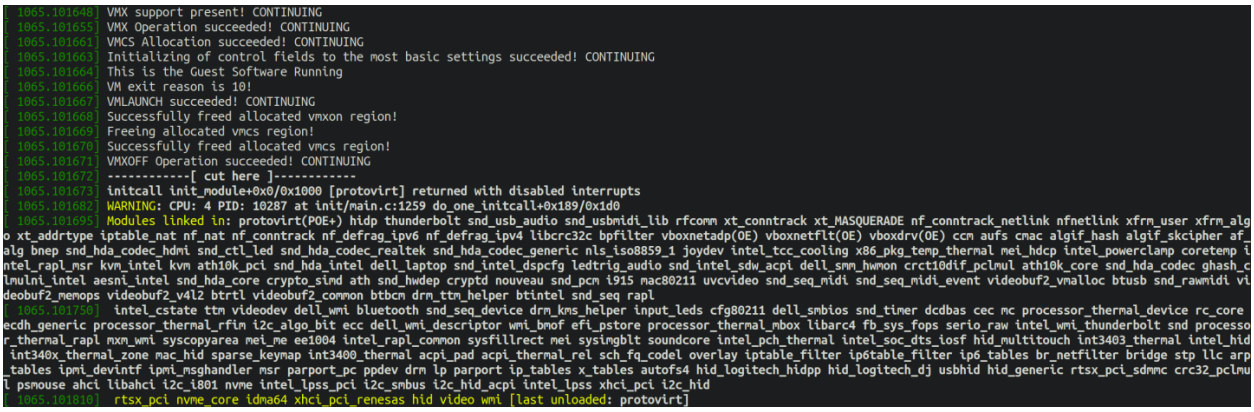
3.5 Setting up Host and Guest State

The first thing we will set up is the host state. The host handles the hardware for the guests running on the virtual machine. When a VM is exited, it returns back to the host state. The state of the processor is saved into the guest state when the VM exits and loaded from the guest state when the VM starts up. When the VM exits, the processor state is loaded from the host state. We have to set up the appropriate methods in our code to load/save the guest and host state. When the hypervisor starts up, we will load control registers, selector fields, base-address fields, and module-specific registers. Lastly, we need to save the stack pointer and instruction pointer to the host state, but we won't do that until we are ready to launch the guest state. This ensures that we return to the state of the stack just before the guest was launched. The guest state is split into the register and non-register states. The register state holds the processor state from registers similar to those in the host state. The non-register state holds information that characterizes the guest state but is not held in processor registers. The guest register state holds the control register, selector fields, base-address fields, stack pointer, instruction pointer, debug register, access rights, and module-specific registers. The control register and most of the fields are copied directly from the host state. For the non-register state, we only set the necessary components in our hypervisor. This includes the activity state, preemption timer, VMCS link pointer, guest interrupt status, and PML index. The activity state describes whether the guest state is executing instruction normally (active) or not executing instructions (inactive). The guest state can be inactive because it executed a halt instruction (state 1), incurred a fault or serious error (state 2), or it is waiting for the startup IPI to tell it where to start (state 3). The preemption timer is used to invoke VMEXIT based on the timer value. This can be saved in the guest state using the preemption timer value. The VMCS link pointer is used the save the address that VMWRITE/VMREAD use to access the VMCS. The PML index holds the index of the next value in the page modification log (PML). The PML holds logging information pertaining to writes

to guest physical memory. After launching the guest VM, the guest stack pointer corresponds to a buffer for that guest. The instruction pointer corresponds to the instructions specified inside the guest_code() method. Upon exit, the guest saves its state and deallocates all the memory used to store its state.

4 RESULTS

In the first figure you can see the printed debug statements from the startup and shutdown of the hypervisor. You can see that it first checks for VMX support which was explained in 3.3. The hypervisor runs functions to check for support for VMX. After that, the VMX operation that consists of the setup for VMXON and calling for the VMXON is executed. This is explained in section 3.4 which includes the details on setup for the VMXON. This is followed by the setup and allocation for VMCS which is done according to the details specified in section 3.4. Lastly, the guest state is setup in the steps laid out in section 3.5. You can then see the message that the guest software is running. The guest software calls a cpuid assembly operation which causes the exit reason 10 to be CPUID which is caused when "Guest software attempted to execute CPUID". After that a message is shown to notify the user that the VM successfully launched. The following lines show the shutdown operations after module is unloaded. The VMXON and VMCS regions are deallocated and VMXOFF closes the hypervisor.



The second picture shows the change in registers and instruction pointer when going back from the Guest software to our VMM. The switch happens when various syscalls are performed. The trace ends when this has concluded.

Because we could not get support for arguments to guest_code() to work, we decided to show the use of the hypervisor by running fibonacci. We implemented a simple for loop to run calcuate fibonacci numbers up to 10 using c code in the guest_code() function. The results of the output are shown in the second figure below.

```
[ 1065.101810]  rtsx_pci nvme_core idma64 xhci_pci_renesas hid video wmi [last unloaded: protovirt]
[ 1065.101819] CPU: 4 PID: 10287 Comm: insmod Tainted: P        W  OE     5.13.0-40-generic #45~20.04.1-Ubuntu
[ 1065.101822] Hardware name: Dell Inc. XPS 15 9560/05FFDN, BIOS 1.21.0 10/19/2020
[ 1065.101824] RIP: 0010:do_one_initcall+0x189/0x1d0
[ 1065.101828] Code: fa a2 48 8d 7d a0 e8 a6 6a 5e 00 fb 66 0f 1f 44 00 00 e9 49 ff ff ff 48 8d 55 a0 4c 89 e6 48 c7 c7 a8 2c fa a2 e8 5f b3 b9 00 <0f> 0b e9 39 ff ff ff 41 bd ff f
f ff ff e9 2e ff ff ff e8 e0 72 bf
[ 1065.101830] RSP: 0018:ffffaa3d839bfbb8 EFLAGS: 00010286
[ 1065.101833] RAX: 0000000000000000 RBX: 0000000000000000 RCX: 0000000000000027
[ 1065.101834] RDX: 0000000000000027 RSI: 00000000ffffdfff RDI: ffff91926e420988
[ 1065.101836] RBP: ffffaa3d839bfc18 R08: ffff91926e420980 R09: ffffaa3d839bf990
[ 1065.101837] R10: 0000000000000001 R11: 0000000000000001 R12: ffffffffc12a9000
[ 1065.101839] R13: 0000000000000000 R14: ffffffffc16c3000 R15: 0000000000000000
[ 1065.101840] FS:  00007f39b4f09540(0000) GS:ffff91926e400000(0000) knlGS:0000000000000000
[ 1065.101842] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[ 1065.101844] CR2: 000055d9b65e06b8 CR3: 000000030defe004 CR4: 0000000003726e0
[ 1065.101846] Call Trace:
[ 1065.101849]  <TASK>
[ 1065.101852]  ? kmem_cache_alloc_trace+0x37c/0x440
[ 1065.101864]  do_init_module+0x62/0x260
[ 1065.101868]  load_module+0x125d/0x1440
[ 1065.101872]  __do_sys_finit_module+0xc2/0x120
[ 1065.101874]  ? __do_sys_finit_module+0xc2/0x120
[ 1065.101876]  __x64_sys_finit_module+0x1a/0x20
[ 1065.101878]  do_syscall_64+0x61/0xb0
[ 1065.101883]  ? syscall_exit_to_user_mode+0x27/0x50
[ 1065.101888]  ? __x64_sys_mmap+0x33/0x40
[ 1065.101892]  ? do_syscall_64+0x6e/0xb0
[ 1065.101893]  ? do_syscall_64+0x6e/0xb0
[ 1065.101895]  ? do_syscall_64+0x6e/0xb0
[ 1065.101896]  ? syscall_exit_to_user_mode+0x27/0x50
[ 1065.101898]  ? __x64_sys_newstat+0x16/0x20
[ 1065.101905]  ? do_syscall_64+0x6e/0xb0
[ 1065.101906]  ? irqentry_exit_to_user_mode+0x9/0x20
[ 1065.101909]  ? irqentry_exit+0x19/0x30
[ 1065.101912]  ? exc_page_fault+0x8f/0x170
[ 1065.101914]  ? asm_exc_page_fault+0x8/0x30
[ 1065.101922]  entry_SYSCALL_64_after_hwframe+0x44/0xae
[ 1065.101924] RIP: 0033:0x7f39b504b76d
[ 1065.101927] Code: 00 c3 66 2e 0f 1f 84 00 00 00 00 00 90 f3 0f 1e fa 48 89 f8 48 89 f7 48 89 d6 48 89 ca 4d 89 c2 4d 89 c8 4c 8b 4c 24 08 0f 05 <48> 3d 01 f0 ff ff 73 01 c3 48 8
b 0d f3 36 0d 00 f7 d8 64 89 01 48
[ 1065.101930] RSP: 002b:00007fff942a0768 EFLAGS: 00000246 ORIG_RAX: 0000000000000139
[ 1065.101932] RAX: ffffffffffffffda RBX: 000055d9b65dc780 RCX: 00007f39b504b76d
[ 1065.101934] RDX: 0000000000000000 RSI: 000055d9b615d358 RDI: 0000000000000003
[ 1065.101935] RBP: 0000000000000000 R08: 0000000000000000 R09: 00007f39b5122580
[ 1065.101936] R10: 0000000000000003 R11: 0000000000000246 R12: 000055d9b615d358
[ 1065.101937] R13: 0000000000000000 R14: 000055d9b65df6b0 R15: 0000000000000000
[ 1065.101940]  </TASK>
[ 1065.101940] ---[ end trace 1a910bb75a1a373c ]---
```

Figure 7: Running Fibonacci in Hypervisor

```
[  635.480687] protovirt: module license 'GPL V3' taints kernel.
[  635.480701] Disabling lock debugging due to kernel taint
[  635.481501] VMX support present! CONTINUING
[  635.481506] VMX Operation succeeded! CONTINUING
[  635.481510] VMCS Allocation succeeded! CONTINUING
[  635.481512] Initializing of control fields to the most basic settings succeeded! CONTINUING
[  635.481513] This is the Guest Software Running
[  635.481514] Here are 10 Fibonacci number
[  635.481514] 0;
[  635.481515] 1;
[  635.481516] 1;
[  635.481516] 2;
[  635.481517] 3;
[  635.481517] 5;
[  635.481518] 8;
[  635.481518] 13;
[  635.481519] 21;
[  635.481520] 34;
[  635.481520] Guest Finishing
[  635.481537] general protection fault, maybe for address 0xf: 0000 [#1] SMP PTI
[  635.481545] CPU: 1 PID: 7066 Comm: insmod Tainted: P           OE     5.13.0-40-generic #45~20.04.1-Ubuntu
[  635.481548] Hardware name: Dell Inc. XPS 15 9560/05FFDN, BIOS 1.21.0 10/19/2020
[  635.481551] RIP: 0010:guest_code+0x61/0x62 [protovirt]
[  635.481558] Code: c7 c7 ba a3 4d c1 e8 8a b1 2c ea 44 89 e0 41 01 dc 89 c3 41 ff cd 75 e5 48 c7 c7 bf a3 4d c1 e8 71 b1 2c ea 5b 41 5c 41
5d 5d <c3> 48 c7 c7 a0 a0 4d c1 e8 5e b1 2c ea 31 c0 e9 a0 f5 ff ff 48 c7
[  635.481560] RSP: 0018:ffffb33102937c20 EFLAGS: 00010046
[  635.481563] RAX: 000000000000000f RBX: 0000000002100800 RCX: 0000000000000027
[  635.481565] RDX: 0000000000000000 RSI: 00000000ffffdfff RDI: ffff9701ae2a0988
[  635.481567] RBP: ffffb33102937c88 R08: ffff9701ae2a0980 R09: ffffb331029379e0
[  635.481568] R10: 0000000000000001 R11: 0000000000000001 R12: ffffffffc14e8000
[  635.481570] R13: ffff96ff5dc190f0 R14: ffffffffc14db000 R15: 0000000000000000
[  635.481571] VM exit reason is 31!
[  635.481572] VMLAUNCH succeeded! CONTINUING
[  635.481574] Successfully freed allocated vmxon region!
[  635.481575] Freeing allocated vmcs region!
[  635.481578] Successfully freed allocated vmcs region!
[  635.481579] VMXOFF Operation succeeded! CONTINUING
```

5 CONCLUSION

In this project we were able to successfully implement a hypervisor in a Linux system that can run specified code given from the user. We started out by giving readers a quick explanation on what a hypervisor is and how it is used to run virtual machines. We laid out the terminology and explained each part and how it relates to our implementation. Next we showed how to set up a basic kernel module in linux and how it can be used to start a hypervisor. We then explained the steps taken to setup the project before starting a VMXON operation. We then explained the Virtual Machine Control Structure and how to add it to the project. Next, we explained the registers and data needed for the guest and host state and how to implement them in the project. Lastly, we show the results from a quick startup and shutdown of our hypervisor and prove that it runs the guest code given.

We tried to support arguments for the guest function so we can increase functionality, but the extended inline assembly code needed support for some standard library files that wouldn't work on the current implementation of the VMM created. The *stdio.h* would have been used to support custom registers which would help print and extract the register values transferred from the VMM. It was also difficult to test register functionality of register operations as it would cause the program and the OS that our machine was running on to completely become unresponsive. In future works, we would implement this as well as I/O support to make our hypervisor more competitive with those used in industry.

6 CITATIONS

1 http://linasm.sourceforge.net/docs/instructions/vmx.php#:~:text=VM%20extension%20allows
2 https://en.wikipedia.org/wiki/CPUID
3 https://en.wikipedia.org/wiki/Control_register#CR4
4 https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
5 https://nixhacker.com/developing-hypervisior-from-scratch-part-1/