Grad Project: Implementing a sysfs interface to the VMCS

Deval Kaku Jing Huang William Maxwell

May 24th, 2018

CS544: Operating Systems II
Spring 2018

Abstract

The VMCS is the virtual machine control structure. This paper includes the research done on VMCS. The document includes all fields, what they are used for, what they control and how to access them from the kernel. This document also includes the design for the patch.

Contents

1 Introduction

Sysfs, a virtual file-system which can be used for bidirectional communication with the kernel, will be integrated with VMCS to make it easy to access VMCS data in user-space. This paper will first make a brief introduction to VMCS based on the research to make the following progress easier. Then, the modified patch is designed to implement the VMCS Sysfs interface. The modified code will also be included in this part. This project requires Minnowboard hardware, CentOS 7 64-bit and 4.1.5 linux kernel.

2 VMCS

2.1 Introduction

The virtual-machine control data structure (VMCS) is defined for the virtual machine extensions (VMX) which manages transitions in and out of VMX non-root operation which are VM entries and VM exits as well as processor behavior in VMX non-root operation. VMCLEAR, VMPTRLD, VMREAD, and VMWRITE are the four new instructions which can be used to manipulate the VMCS.

A logical processor uses virtual-machine control data structures (VMCSs) for its VMX operation. It is manipulated by the instructions of VMCLEAR, VMPTRLD, VMREAD, and VMWRITE. A VMM can use a different VMCS for each machine that it supports. The VMM uses a different VMCS for every virtual processor of the virtual machine which has multiple processors.

There is a memory region in the VMCS called as VMCS Region which has a logical processor associated to it. The 64-bit VMCS address of a region (VMCS pointer) is referenced by a software. The logical processor also keeps track of how many VMCS are active. At any given time, at most one of the active VMCSs is the current VMCS. The VMLAUNCH, VMREAD, VMRESUME, and VMWRITE instructions operate only on the current VMCS. The VMCS pointer is read and written using the instructions VMPTRST and VMPTRLD. The VMM configures a VMCS using the VMREAD, VMWRITE, and VMCLEAR instructions.

2.2 VMCS Region

In order to make sure the proper behavior of VMX operation, the VMCS region and the related structures should be maintained by the software. The total size of a VMCS region is 4KB. There are three major contents with 12 bytes in total in the format of VMCS, which are VMCS revision identifier, VMX-abort indicator, and VMCS data (implementation-specific format).

The first 4 bytes of the VMCS region is used for the VMCS revision identifier. he function of VMCS revision identifier is to avoid the wrong usage of processors with different format. By reading the VMX capability MSR VMX_BASIC, the software can get the VMCS revision identifier used by specific processor. Every time before using the region of a VMCS, it is necessary to write the VMCS revision identifier to the VMCS region by the software.

VMX-Abort indicator uses the next 4 bytes of the VMCS region. The contents of these bytes do not control processor operation in any way. but only when VMX-abort happens. A logical processor writes a non-zero value into these bytes if a VMX abort occurs and also the software writes into this field

The last 4 bytes are used for VMCS Data. It is mainly responsible of the VMX transitions and VMX non-root operations.

2.3 VMCS Data

The VMCS data are organized into following logical groups:

2.3.1 Guest-state area

Processor state is saved into the guest-state area on VM exits and loaded from there on VM entries. This section includes fields both with and without processor registers.

There are fields in guest-state area:

- Control registers CR0, CR3, and CR4 (64 bits each)
- Debug register DR7 (64 bits).
- RSP, RIP, and RFLAGS (64 bits each).1
- Registers CS, SS, DS, ES, FS, GS, LDTR, and TR
- Registers GDTR and IDTR
- MSRs IA32_DEBUGCTL (64 bits), IA32_SYSENTER_CS (32 bits), IA32_SYSENTER_ESP (64 bits), and IA32_SYSENTER_EIP (64 bits).

There are fields in guest non-register state:

- Activity state (32 bits)
- Interruptibility state (32 bits)
- VMCS link pointer (64 bits)
- Pending debug exceptions (64 bits)

2.3.2 Host-state area

Processor state is loaded from the host-state area on VM exits. All fields below is in the host-state area correspond to processor registers:

- CR0, CR3, and CR4 (64 bits each)
- RSP and RIP (64 bits each)
- Selector fields (16 bits each) for the segment registers CS, SS, DS, ES, FS, GS, and TR
- Base-address fields for FS, GS, TR, GDTR, and IDTR (64 bits each)
- MSRs IA32_SYSENTER_CS (32 bits), IA32_SYSENTER_ESP (64 bits), and IA32_SYSENTER_EIP (64 bits).

2.3.3 VM-execution control fields

The VM-execution control fields govern VMX non-root operation.

- Pin-Based VM-Execution Controls: Pin-based VM execution controls form a 32-bit vector that manages the processing of asynchronous events.
- Processor-Based VM-Execution Controls: The processor-based virtual machine execution controls constitute two 32-bit vectors that are used to manage the processing of synchronous events, primarily caused by the execution of specific instructions. These two vectors are based on the main processor's VM execution control and the auxiliary processor's VM execution control.
- Exception Bitmap: The exception bitmap is a 32-bit field, and each exception contains one bit. When an exception occurs, its vector is used to select a bit in this field. If this bit is 1, an exception will cause the VM to quit. If this bit is 0, the exception is normally transmitted through the IDT and the descriptor corresponding to the exception vector is used.
- Executive-VMCS Pointer: The execution VMCS pointer is a 64-bit field that is used for dual-monitor processing in System Management Interrupt (SMIs) and System Management Mode (SMM).
- Virtual Processor Identifier: The virtual processor identifier (VPID) is a 16-bit field. It only exists on processors that support the 1-setting of "Enable VPID" VM Execution Control.

2.3.4 VM-Exit Control

The virtual machine exit control forms a 32-bit vector that is used to manage the basic operations of the virtual machine exit. There is a field related to the size of the host address space (whether the host should wake up in 64-bit mode after the VM exits) and a field indicating whether the logical processor acknowledges the interrupt controller, obtains the interrupt vector, and the VM is due to an external interrupt.

2.3.5 VM-Exit Information Fields

These control fields contains read only information about the most recent VM exit. It can be divided in five parts.

- Basic VM-Exit Information
  - Information for VM Exits Due to Vectored Events
  - Information for VM Exits That Occur During Event Delivery
  - Information for VM Exits Due to Instruction Execution
  - VM-Instruction Error Field

2.3.6 VM-Entry Control Fields

These fields control the entries of VM. There are three parts in VM-entry Control Fields.

- VM-entry interruption-information field (32 bits).
- VM-entry exception error code (32 bits).
- VM-entry instruction length (32 bits).

2.4 VMX operations

Virtual-machine extensions (VMX) allow for processor support for virtual machines. This can either be achieved as a Virtual-machine monitor (VMM) which acts as the host and has full processor control, or as guest software which has reduced privileges. VMX operations provide processor support for virtualization. This can be done as either a root or non-root operation. Typically, VMMs run in VMX root mode while guest software will run in VMX non-root mode. VMX can transition between root and non-root mode using an operation called a VMX transition. A transition into a non-root operation is called a VM entry while a transition from a non-root operation is VM exit.

A typical VMM will follow the process:

- Begin VMX operation by executing the VMXON instruction.
- VMM places guest software into virtual machines by using the VM entry operation. This is done with the VMX instructions VMLAUNCH and VMRESUME. Control is given back to VMM by using VM exit.
- VMM leaves the VMX operation by executing the VMXOFF instruction.

The purpose of this project is to implement the virtual-machine control system (VMCS). This is a data structure used to manage VMX non-root operations and VMX transitions. The processor state maintains a VMCS pointers that manages access to the VMCS. Read and writes through the pointer are performed with the instructions VMPTRST and VMPTRLD. VMM has instructures VMREAD, VMWRITE, and VMCLEAR used to configure the VMCS. The VMM can use one VMCS for each virtual machine that it manages, or if one virtual machine contains multiple logical processors the VMM could use one VMCS for each logical processor.

2.5 Virtual-Machine Control Structure (VMCS)

Each logical processor stores its VMCS into a 4KB section of memory known as the VMCS region. The VMCS region is configured via the VMCS pointer mentioned in the previous section. There are three pieces of data associated with the VMCS region.

- VMCS revision identifier - Specifies the format of the VMCS. Processors require a specific VMCS format to run.
- VMX-abort indicator - These four bytes do not control any of the processor operations. Instead, a non-zero value is written here by a logical processor if a VMX abort occurs.
- VMCS data - The remainder of the VMCS region stores the operations that control the VMX operations.

The description of the VMCS data section above is rather vague. VMCS data is actually organized into six logical sections each controlling a different part of the VMX operations.

- Guest-state area - Processor state is saved here on VM exits and loaded from here on VM entries.
- Host-state area - Processor state is loaded from here on VM exits.
- VM-exit control fields - Control VM exits.
- VM-entry control fields - Control VM entries.
- VM-exit information fields - Read-only fields that receive information describing VM exits.

2.6 Kobjects

A kobject is an object of type `struct kobject` and is used as the abstraction for the Linux driver model. They are defined in `linux/kobject.h`. They contain a name, reference count, parent pointer, a type, and a representation in the sysfs virtual file system. They are typically embedded into some other object rather than used on their own. Objects that embed kobjects are known as ktypes. The ktype controls the creation and destruction of the kobject. A group of kobjects is called a kset, kobjects within a kset can belong to one or more different ktypes. Kobjects are initialized with the function

```
void kobject_init(struct kobject *kobj, struct kobj_type *ktype);
```

and are added with

```
int kobject_add(struct kobject *kobj, struct kobject *parent, const char *fmt, ...);
```

These two operations can be done simultaneously with

```
int kobject_init_and_add(struct kobject *kobj, struct kobj_type *ktype, ...);
```

3 Installation

3.1 CentOS 7 Installation

For our installation we chose to use the CentOS 7 minimal installation. The minimal installation does not contain a GUI but instead only ships with core functionality. We downloaded the .iso file from the CentOS 7 downloads page and used the `dd` command to write it do a USB flash drive.

After plugging the microHDMI cable into the Minnowboard we power up, exit the UEFI shell, and select to boot from USB. We boot into the CentOS installer and install to the SD card. Upon rebooting we can boot directly from the SD card.

3.2 Linux 4.1.5 kernel install

We start by downloading the Linux 4.1.5 source from kernel.org. In the top level directory weuse `mv` to make the `minimal.config` file provided by the course into the `.config` file required by the Kernel. We need to get the Minnowboard specific configurations, and do so with the following command

```
wget http://www.elinux.org/images/e/e2/Minnowmax-3.18.txt
```

Next, we need to merge the Minnowboard configurations with our current configuration.

```
scripts/kconfig/merge_config.sh .config Minnowmax-3.18.txt
```

Now we compile the kernel

```
make -j4 && make -j4 modules
```

To install the kernel we issue

```
make install && make install_modules
```

Now, we can choose the 4.1.5 kernel from the Grub menu upon booting the Minnowboard.

4 Configuration

- Use the configure file that from class website of the minimal.config and use command mv minimal.config to change the minimal.config for the current configuration.
- Download the config files for Minnowboard, then we need to merge the two kernel config files.
- Compile and install kernel.
  - make -j4 && make -j4 modules
  - make install && make modules install
- Use "make menuconfig" command to configure the Linux 4.1.5

5 Modify VMCS sysfs

sysfs is a ram-based file system used by the Linux kernel to export kernel data to userspace. Sysfs can be accessed via the `/sys` directory. Some important subdirectories include:

- `block/` - Contains block devices
- `bus/` - Contains one subdirectory for each bus type in the kernel
- `class/` - Contains subdirectories for each device class
- `dev/` - Contains subdirectories `block/` and `char/` corresponding to block and character devices on the system
- `devices/` - Contains kernel device tree
- `kernels/` - Contains information about the running kernel

Sysfs acts as a representation for the kobjects defined in the Linux kernel. To export a kobject to sysfs we call the function `kobject_add()`. We can do this at the same time as creation by instead using the function `kobject_create_and_add()`. Individual kobjects appear as directories in `/sys` and the hierarchical structure defined by kobjects, ktypes, ksets, and the parent pointer define the directory trees found in `/sys`.

Vmx.c is under the directory /arch/x86/kvm. VMCS sets is in this file. In order to access the VMCS, we should add code that how to read and write VMCS systs.

- We define serval attribute to represent the files in vmcs directory. This attribute structures have different name and same mode. They store in d_abt[]. After define the attributes, we define Ktype. Its default attrs is d_abt[]. When doing the kobject init and add() function we use the Ktype.

- After define the attribute and ktype, we need define sysfs ops structure. It have two functions to operate the kobject. Kobj_show() and kobj_store().
- The function kobj_show() and kobj_store() let the kernel read and write vmcs sysfs. The vmcs_readl() in kobj_show and vmcs_writel() in kobj_store() are used for getting the vmcs value from kernel and change the vmcs value.

6 Patch Design

6.1 Overview

Sysfs is the virtual filesystem, its content under /sys directory when system startup.

6.2 Design

The code we would modify is in fs/sysfs. Here are the files that may be modified.

- include/linux/sysfs.h
- fs/sysfs/sysfs.h
- fs/sysfs/mount.c
- fs/sysfs/inode.c
- fs/sysfs/dir.c
- fs/sysfs/file.c
- fs/sysfs/group.c
- fs/sysfs/symlink.c
- fs/sysfs/bin.c

At last, we need to use "sysfs_init" to initialize the sysfs.

7 Implementation

7.1 Design

We start by initializing a new git repository for the kernel by issuing the `git init` command at the root directory of the kernel source. We will implement our VMCS sysfs interface as a kernel module. The files we are interested in are located in the `arch/x86/kvm` directory.

First we create the file `arch/x86/kvm/vmcs-sysfs.c` which will serve as the file defining our kernel module. Second, we modify the Makefile located at `arch/x86/kvm/Makefile`. We add the line

```
obj-$(CONFIG_VMCS__SYSFS) += vmcs-sysfs.o
```

to let the Makefile know we wish to compile `vmcs-sysfs.c`. To allow the VMCS interface to be selected as a compilation configuration we need to modify the Kconfig file at `arch/x86/kvm/Kconfig`. We add the following lines

```
config VMCS_SYSFS
        tristate "VMCS sysfs interface"
        default y
```

7.2 Testing

If everything is working properly there should be a `vmcs/` subdirectory in the `/sys` directory. This is the result of the defined kobject. The attributes defined in the module should appear in the directory tree. To test the read functionality simply `cat` the `/sys/vmcs/g_rsp_b` path to check for a response. Similarly, to test the write functionality just `echo` some value to `/sys/vmcs/g_rsp_b` and run `cat` again to see if the value has changed.

8 Patch source code

```
        commit ea68cb2e8b6fd40f802a51a21ac5324c2532b376
Author: will <wmaxwell90@gmail.com>
Date:   Tue Jun 12 12:01:59 2018 -0700

    final

diff --git a/arch/x86/kvm/Kconfig b/arch/x86/kvm/Kconfig
new file mode 100644
index 0000000..bd7465b
--- /dev/null
+++ b/arch/x86/kvm/Kconfig
@@ -0,0 +1,108 @@
+#
+# KVM configuration
+#
+
+source "virt/kvm/Kconfig"
+
+menuconfig VIRTUALIZATION
+	bool "Virtualization"
+	depends on HAVE_KVM || X86
+	default y
+	---help---
+	  Say Y here to get to see options for using your Linux host to run other
+	  operating systems inside virtual machines (guests).
+	  This option alone does not add any kernel code.
+
+	  If you say N, all options in this submenu will be skipped and disabled.
+
+if VIRTUALIZATION
+
+config KVM
+	tristate "Kernel-based Virtual Machine (KVM) support"
+	depends on HAVE_KVM
+	depends on HIGH_RES_TIMERS
+	# for TASKSTATS/TASK_DELAY_ACCT:
+	depends on NET && MULTIUSER
+	select PREEMPT_NOTIFIERS
+	select MMU_NOTIFIER
+	select ANON_INODES
+	select HAVE_KVM_IRQCHIP
+	select HAVE_KVM_IRQFD
+	select HAVE_KVM_IRQ_ROUTING
+	select HAVE_KVM_EVENTFD
+	select KVM_APIC_ARCHITECTURE
+	select KVM_ASYNC_PF
+	select USER_RETURN_NOTIFIER
+	select KVM_MMIO
+	select TASKSTATS
+	select TASK_DELAY_ACCT
+	select PERF_EVENTS
+	select HAVE_KVM_MSI
+	select HAVE_KVM_CPU_RELAX_INTERCEPT
+	select KVM_GENERIC_DIRTYLOG_READ_PROTECT
+	select KVM_VFIO
+	select SRCU
+	---help---
```

```
+         Support hosting fully virtualized guest machines using hardware
+         virtualization extensions.  You will need a fairly recent
+         processor equipped with virtualization extensions. You will also
+         need to select one or more of the processor modules below.
+
+         This module provides access to the hardware capabilities through
+         a character device node named /dev/kvm.
+
+         To compile this as a module, choose M here: the module
+         will be called kvm.
+
+         If unsure, say N.
+
+config KVM_INTEL
+       tristate "KVM for Intel processors support"
+       depends on KVM
+       # for perf_guest_get_msrs():
+       depends on CPU_SUP_INTEL
+       ---help---
+         Provides support for KVM on Intel processors equipped with the VT
+         extensions.
+
+         To compile this as a module, choose M here: the module
+         will be called kvm-intel.
+
+config KVM_AMD
+       tristate "KVM for AMD processors support"
+       depends on KVM
+       ---help---
+         Provides support for KVM on AMD processors equipped with the AMD-V
+         (SVM) extensions.
+
+         To compile this as a module, choose M here: the module
+         will be called kvm-amd.
+
+config KVM_MMU_AUDIT
+       bool "Audit KVM MMU"
+       depends on KVM && TRACEPOINTS
+       ---help---
+        This option adds a R/W kVM module parameter 'mmu_audit', which allows
+        auditing of KVM MMU events at runtime.
+
+config KVM_DEVICE_ASSIGNMENT
+       bool "KVM legacy PCI device assignment support"
+       depends on KVM && PCI && IOMMU_API
+       default y
+       ---help---
+
+config VMCS_SYSFS
+    tristate "VMCS sysfs interface"
+    default y
+         Provide support for legacy PCI device assignment through KVM.  The
+         kernel now also supports a full featured userspace device driver
+         framework through VFIO, which supersedes much of this support.
+
+         If unsure, say Y.
+
+# OK, it's a little counter-intuitive to do this, but it puts it neatly under
+# the virtualization menu.
```

```
+source drivers/vhost/Kconfig
+source drivers/lguest/Kconfig
+
+endif # VIRTUALIZATION
diff --git a/arch/x86/kvm/Makefile b/arch/x86/kvm/Makefile
new file mode 100644
index 0000000..67dc3fe
--- /dev/null
+++ b/arch/x86/kvm/Makefile
@@ -0,0 +1,23 @@
+
+ccflags-y += -Iarch/x86/kvm
+
+CFLAGS_x86.o := -I.
+CFLAGS_svm.o := -I.
+CFLAGS_vmx.o := -I.
+
+KVM := ../../../virt/kvm
+
+kvm-y                   += $(KVM)/kvm_main.o $(KVM)/coalesced_mmio.o \
+                                $(KVM)/eventfd.o $(KVM)/irqchip.o $(KVM)/vfio.o
+kvm-$(CONFIG_KVM_ASYNC_PF)      += $(KVM)/async_pf.o
+
+kvm-y                   += x86.o mmu.o emulate.o i8259.o irq.o lapic.o \
+                           i8254.o ioapic.o irq_comm.o cpuid.o pmu.o
+kvm-$(CONFIG_KVM_DEVICE_ASSIGNMENT)     += assigned-dev.o iommu.o
+kvm-intel-y             += vmx.o
+kvm-amd-y               += svm.o
+
+obj-$(CONFIG_KVM)       += kvm.o
+obj-$(CONFIG_KVM_INTEL)         += kvm-intel.o
+obj-$(CONFIG_KVM_AMD)   += kvm-amd.o
+obj-$(CONFIG_VMCS_SYSFS) += vmcs-sysfs.o
diff --git a/arch/x86/kvm/vmcs-sysfs.c b/arch/x86/kvm/vmcs-sysfs.c
new file mode 100644
index 0000000..5198361
--- /dev/null
+++ b/arch/x86/kvm/vmcs-sysfs.c
@@ -0,0 +1,30 @@
+#include <linux/init.h>
+#include <linux/module.h>
+#include <linux/kernel.h>
+#include <asm/vmx.h>
+
+static unsigned long guest_activity state;
+
+static int vmcs_sysfs_init(void) {
+    struct vmcs *current_vmcs;
+
+    asm volatile("VMPTRST %0" : "=m"(current_vmcs));
+    asm volatile("VMREAD %0, %1"
+                : "=r"(guest_activity_state)
+                : "r"(enc_guest_activity_state)
+            );
+
+    printk(KERN_INFO "initializing VMCS, guest activity state is %lu\n", guest_activity_state);
+
+}
+
+static void vmcs_sysfs_exit(void) {
+	printk(KERN_INFO "exiting VMCS, guest activity state is %lu\n" guest_activity_state);
+}
+
+module_init(vmcs_sysfs_init);
+moduel_exit(vmcs_sysfs_exit);
+
+MODULE_LICENSE("GPL");
+MODULE_AUTHOR("Group 12");
+MODULE_DESCRIPTION("An implementation of a VMCS sysfs interface");
```

References

[1] Intel. *Virtualization Technology Specification for the IA-32 Intel Architecture*. Intel. Apr, 2005. [online] Available at: http://dforeman.cs.binghamton.edu/ foreman/550pages/Readings/intel05virtualization.pdf. [Accessed 24 May. 2018].

[2] Stephen J. B. *The 'what, where and why' of VMCS shadowing*. Search Server Virtualization. NOV 2013. [online] Available at:https://searchservervirtualization.techtarget.com/feature/The-what-where-and-why-of-VMCS-shadowing.[Accessed 24 May. 2018].

[3] Intel. *4th Generation Intel© Core^{TM} vPro^{TM} Processors with Intel© VMCS Shadowing*. Intel. 2013. [online] Available at: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-vmcs-shadowing-paper.pdf. [Accessed 24 May. 2018].