Grad Project: Implementing a sysfs interface to the VMCS

Chao-Ting Wen Chih-Hsiang Wang .Suwadi

May 25th, 2018

CS544

Operating Systems II

(Spring 2018)

Abstract

Researching the contents of the 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 and the implementation of sysfs to VMCS.

CONTENTS

1 INTRODUCTION

VMCS stands for virtual machine control structure and is used to manage and track the virtual CPU in the VMX/KVM layer of the kernel. In the project, 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. Once the research and design stages are finished, we will illustrate how to install the two systems, CentOS 7 64-bit and 4.1.5 linux kernel, into the Minnowboard step by step. Finally, the test result and the conclusion will be given to sum up the project.

2 BACKGROUND OF VMCS

2.1 Overview

VMCS is a data structure in memory that exists exactly once per VM while it is managed by the VMM. VMCS is defined for VMX(Virtual Machine Extensions) operation that it manages transitions in and out of VMX non-root operation and processor behavior in VMX non-root operation. The main purpose of VMX operation is to support virtualization in the system. There are four instructions being used to access the VMCS from the kernel, which are VMCLEAR, VMPTRLD, VMREAD, and VMWRITE. VMCS region is a region in memory with a logical processor associating with each VMCS a 4KB region. To access the VMCS by instructions, VMCS pointer is utilized. VMCS pointer is a 64-bit physical address referenced with a VMCS in VMCS region and every VMCS pointer is 4KB-aligned. To be noticed, the VMCS pointer should not set bits beyond the processors physical address width.

There may be any number of active VMCSs in a logical processor while there can only be one current VMCS at most. Executing VMPTLRD with the address of the VMCS can active the VMCS while VMCLEAR with the address of the VMCS can inactive it. If the VMCS is inactive, it's status will disappear from the logical processor. VMCS region is undefined if executing VMXOFF when VMCS is active, so the situation should be avoided by the software. By executing VMPTLRD with the address of the VMCS, the VMCS can be made current with the address loaded into the current VMCS pointer. There are some VMX instructions to operate on the current VMCS, which are VMLAUNCH, VMPTRST, VMREAD, VMRESUME, and VMWRITE. The detailed of the instructions will not be illustrated in the paper.

2.2 Region of VMCS

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 is for VMCS revision identifier that different formats of VMCS data in the processors come with different VMCS revision identifiers. The 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.

The next 4 bytes are responsible for the VMX-abort indicator. The VMX-abort indicator is used when a VMX abort happens. If the value of bytes in the VMX-abort indicator is not zero, it means there is a VMX abort and the software will write into the field.

The last 4 bytes are for the use of data of VMCS which is in charge of the the VMX non-root operation and the VMX transitions. The format of VMCS data is implementation-specific that it will be discussed later.

2.3 Data of VMCS

The data of VMCS can be categorized into six logical groups:

- Guest-state area. It is used to store the state of processor into the area on VM exits and load the state from the area on VM entries.
- Host-state area. It is the area on VM exits where process state is loaded from.
- VM-execution control fields. The fields can control processor behavior in VMX non-root operation and are related to VM exists.
- VM-exit control fields. It is for the control of VM exists.
- VM-entry control fields. It is for the control of VM entries.
- VM-exit information fields. The fields are read-only for describing the cause of VM exists after receiving information on VM exists.

2.4 Guest-state and Host-state Area

There are fields in the guest-state area and host-state area of the VMCS. These fields are used for loading processor state on VM entry and storing into the fields on VM exit. Some fields in the guest-state area correspond to processor registers while some are not. In the contrast, all fields in the host-state area correspond to processor registers.

The following fields are in guest register state:

- Control registers CR0, CR3, and CR4 (64 bits each).
- Debug register DR7 (64 bits).
- RSP, RIP, and RFLAGS (64 bits each).
- Registers CS, SS, DS, ES, FS, GS, LDTR, and TR.
  — Selector (16 bits).
  — Base address (64 bits).
  — Segment limit (32 bits).
  — Access rights (32 bits).
- Registers GDTR and IDTR.
  — Base address (64 bits). — Limit (32 bits).
- MSRs IA32_DEBUGCTL (64 bits), IA32_SYSENTER_CS (32 bits), IA32_SYSENTER_ESP (64 bits), and IA32_SYSENTER_EIP (64 bits).

The following fields are in guest non-register state:

- Activity state (32 bits). When a logical processor is executing instructions normally, the field is in active state. There are four active states as following:

— 0: Active. The logical processor is executing instructions normally.

— 1: HLT. The logical processor is inactive.

— 2: Shutdown. The logical processor is inactive because it incurred a triple fault3 or some other serious error.

— 3: Wait-for-SIPI. The logical processor is inactive because it is waiting for a startup- IPI (SIPI).

- Interruptibility state (32 bits). The field contains information to block some events for a certain period.
- Pending debug exceptions (64 bits). The field contains information to recognize one or more debug exceptions without immediately delivering them.
- VMCS link pointer (64 bits). The field is used for future expansion that it should be set to FFFFFFFF_FFFFFFFFH to avoid VM-entry failures.

The following fields are in host register state:

- 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 (64 bits each) for FS, GS, TR, GDTR, and IDTR.
- MSRs IA32_SYSENTER_CS (32 bits), IA32_SYSENTER_ESP (64 bits), and IA32_SYSENTER_EIP (64 bits).

2.5 VM Control Fields

The fields of VM can be roughly categorized into five groups based on the different functions. These fields will be briefly explained in the following sections.

2.5.1 VM-execution Control Fields

The VM-execution control fields govern VMX non-root operation. There are two types of VM-execution control fields, pin-based VM-execution controls and processor-based VM-execution controls. While the pin-based VM-execution controls govern the handling of asynchronous events (interrupts), the processor-based VM-execution controls govern the handling of synchronous events.

If the activate I/O bitmaps control is 1, a logical processor uses bitmaps A and B which are 64-bit physical addresses in the VM-execution control fields. These bitmaps are used for the execution of a VM exit while any bit in the I/O bitmaps matching to a port is 1. If the RDTSC exiting control is 0 and the use TSC offsetting control is 1, the 64-bit TSC-offset field in VM-execution control fields controls the executions of the RDTSC instruction.

VM-execution control fields include guest/host masks and read shadows for the CR0 and CR4 registers. Generally, bits are set to 1 to indicate bits owned by the host while set to 0 to indicate bits owned by the guest. Guest attempts to modify bits and guest reads return the the values from corresponding read shadow. VM-execution control fields also include a set of 4 64-bit CR3 target values and a 32-bit CR3 target count. Besides, CR8 register is included to access the task-priority register of the logical processors local APIC.

2.5.2 VM-exit Control Fields

The VM-exit control fields govern the behavior of VM exits. There are two VM-exit controls used currently. Bit 9, host address-space size, is the control determines whether a logical processor is in 64-bit mode after the next VM exit. Bit 15, acknowledge interrupt on exit, is the control affects VM exits because of external interrupts.

While VMM may specify lists of MSRs to be stored and loaded on VM exits, there are four VM-exit control fields for the storing and loading.

- VM-exit MSR-store count (32 bits).
- VM-exit MSR-store address (64 bits).
- VM-exit MSR-load count (32 bits).
- VM-exit MSR-load address (64 bits).

2.5.3 VM-entry Control Fields

The VM-entry control fields govern the behavior of VM entries. There is one VM-entry control, bit 9, IA-32e mode guest, used currently. The bit determines whether the logical processor is in IA-32e mode after VM entry.

While VMM may specify a list of MSRs to be loaded on VM entries, there are two VM-entry control fields for the function.

- VM-entry MSR-load count (32 bits).
- VM-entry MSR-load address (64 bits).

There are three VM-entry control fields as following to deal with event injection.

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

2.5.4 VM-exit Information Fields

The VMCS contains a section of read-only fields that contain information about the most recent VM exit. Attempts to write to these fields with VMWRITE fail. The following fields provide the information while a VM exit happens. Different fields may be produced by different reasons, users can look up the manual to find out the mapped information.

- Exit reason (32 bits).
- Exit qualification (64 bits).
- VM-exit interruption information (32 bits).
- VM-exit interruption error code (32 bits).
- IDT-vectoring information (32 bits)
- IDT-vectoring error code (32 bits).
- VM-exit instruction length (32 bits).
- Guest linear address (64 bits).
- VMX-instruction information (32 bits).

2.5.5 VM-instruction Error Fields

Instead of providing information about the exit of VM, the VM-instruction error fields provide the information about any error encountered by a non-faulting execution of one of the VMX instructions.

3 BACKGROUND KNOWLEDGE FOR DESIGNING

In order to show the data to SYSFS, the structure should use kernel object(Kobject). The structure of the Kobject should be created and point to the VMCS due to the fact that VMCS doesnt have its own kobject. Before the structure is created, the kernel config should be enable/disable.

3.1 Kconfig

The Kconfig should be added some content in order to enable and disable the VMCS sysfs. The content should be added with the follow.

- Config VMCS SYSFS
- tristate Enable_VMCS_SYSFS
- depends on KVM && TRACEPOINTS

3.2 Kobject

A Kobject is an object of type struct Kobject. Kobjects have a name and a reference count. It also has a parent pointer, a specific type, and usually, a representation in the sysfs virtual filesystem. Kobjects are generally not interesting on their own; instead, they are usually embedded within some other structure. No structure should EVER have more than one Kobject embedded within it. If it has more than Kobject embedded within it, the reference counting for the object will be messed up and incorrect. The code below is the structure of Kobject.

```
1  #define KOBJ_NAME_LEN    20
2
3  struct kobject {
4          char                    *k_name;
5          char                    name[KOBJ_NAME_LEN];
6          struct kref             kref;
7          struct list_head        entry;
8          struct kobject          *parent;
9          struct kset             *kset;
10         struct kobj_type        *ktype;
11         struct dentry           *dentry;
12 };
```

Every *struct Kobject* has a name.

- The name is $kobj \rightarrow k\_name$, and this pointer points either to the internal array or to an external string obtained from *kmalloc()* and to be *kfree()* when the kobject dies.
- A *struct kref* is an object that handles reference counting.
- The *entry* field is either empty or part of the circularly linked list containing members of the kset.
- The parent pointer points to this Kobjects parent that Kobjects build an object hierarchy in the kernel and enable the expression of the relationship between multiple objects.

- *kset* is associated with a Kobject, then the parent for the Kobject can be set to NULL in the call to *Kobject_add()* and then the Kobject's parent will be the kset itself.

As we mention before, *Kobject* are usually embedded within some other structure.

3.2.1 Control and manage Kobject

First, a *struct Kobject* must be initialized by using this function *Kobject_init()*. Most fields are not touched by *Kobject_init()*. Before calling *Kobject_init()*, Kobject should memset to zero and assign kset. Which does the *kref_init* that sets the refcount to 1, initializes the entry field to an empty circular list, and does $kobj \rightarrow kset = kset\_get(kobj \rightarrow kset)$; which does not change $kobj \rightarrow kset$ but increments its refcount.

3.2.2 Reference Counts

To serve as a reference counter for the object in which it is embedded is one of the key functions of a Kobject. Once references to the object exist, the object and the code must continue to exist. The routines *kobject_get()* and *kobject_put()* do get/put on the *kref* field. When the reference count drops to zero, a *kobject_cleanup()* is done.

When a reference is released, the call to *kobject_put()* will decrement the reference count and possibly and free the object. Note that *kobject_init()* sets the reference count to one, so the code which sets up the kobject will need to do a *kobject_put()* eventually to release that reference.

3.2.3 Kref

A *struct kref* is just an *atomic_t*. Still, it is a useful abstraction. The struct kref defined as the following below and the avaliable methods *struct kref *k* and *void (*release)(struct kref *k))*:

```
struct kref {
        atomic_t refcount;
};
```

The *struct kref* had a release field, with this appropriate release function that it could be a parameter of kref_put() to save memory.

```
void kref_init(k)              k->refcount = 1;
void kref_get(k)               k->refcount++;
void kref_put(k,release)       if (!--k->refcount) release(k);
```

The *kref get()* function gets the reference of kref which is declared in linux/kref.h. The reference count increased by calling this function but it do not return value. The *kref put()* decrease the reference of kref. If the count is down to zero, then the *release()* function should be called.

3.3 Kset

The Kset is a special Kobject which means that it will show up in "sys" file system. It usually used to union the similar Kobject. The property of these Kobject can be the same or not.

- The *list/list_lock* is to store all the link lists of the Kobject
- *Kobject* is the Kset's own kobject
- *uevent_ops* is operating function set of Kset. When any Kobject need to use uevent, it must call uevent_ops under its own Kset. Which means that if a Kobject didn't belong to any Kset, the uevent is not allow to be sent.

```
1  /* include/linux/kobject.h, line 159 */
2   struct kset {
3       struct list_head list;
4       spinlock_t list_lock;
5       struct kobject kobj;
6       const struct kset_uevent_ops *uevent_ops;
7   };
```

3.4 Ktype

Ktype reprsents the different Kobject type. Every Kobject will have its' own Ktype or the kernel will run error. Ktype(kernel object type) is a special type which relate to Kobject. Ktype is represented by struct kobj type, it defined in linux/kobject.h. Kobj type data structure contains three fields:A release method for releasing resources that kobject occupied; A sysfs ops pointer point to the sysfs operating table; A default list of attributes of sysfs file system. Sysfs operating table includes two functions store () and show (). When a user reads the state property, show () function is called, the function encoding specified property values are stored in the buffer and returned to the user mode. The store () function is used tostore property values that come from user mode.The detail structure is below:

```
1 struct kobj type {
2 void (* release )( struct kobject * ) ;
3 const struct sysfs ops * sysfs ops;
4 struct attribute ** default attrs ;
5 } ;
```

```
1 struct kobj_type {
2         void (*release)(struct kobject *);
3         struct sysfs_ops        * sysfs_ops;
4         struct attribute        ** default_attrs;
5 };
```

```
1 static struct kobj_type vmcsctl_kobj_ktype = {
2                 release = vmcsctl_release ,
3                 sysfs_ops = & kobj_sysfs_ops ,
4 };
```

Here is the detail introduce of the element in kobj type:

- Release pointer point to the deconstructor that called when the kobject reference count reach zero. This function is responsible for free memory and other clean up.

- Sysfs ops variable points to a sysfs ops structure. This structure describe the behavior of sysfs files on read and write.
- Default attrs points to a attribute structure array. This structs define the default attribute of this kobject. attribute describe the feature of the given object. If this kobject export to sysfs, this attribute will be export as a relative file. The last element in the array must ne NULL. Ktype is for describe the behavior of a group of kobject, instead of each kobject define its own behavior, the normal kobject behavior defined in ktype structure once, then all the similar kobject can sharing this same feature.

3.5 Sysfs

SysFS is a ram-based virtual file system. It is the name for all of the files which are under /sys/. So it means files on that file system do not exist on the disk or any other physical media, rather than the representations of configurable kernel variables and the state of certain kernel data structures. SysFS provide a function to export kernel data structures, attributes, and then linkages which between them with userspace. Sysfs file system is a special file system which is similar to proc file system. It was used for organizing the device in the system into a hierarchy and providing particular information of kernel data structures for user-mode programs. For instance, on most systems the file /sys/block/sda/queue/scheduler has a list of available I/O schedulers for the systems hard drive. The current I/O scheduler can be changed by writing the name of one of the schedulers to the file.

3.5.1 Directories in Sysfs

- Block directory: contains all block devices.
- Devices directory: Contains all of the device of system, and organized into a hierarchy according to the type of device attached bus.
- Bus directory: Contains all system bus types
- Drivers directory: Includes all registered kernel device driver
- Class directory: system device type.
- Kernel directory: kernel include the kernel configuration option and state information.

3.5.2 Directory and Devices

- Add and delete kobject from sysfs: Only initialize the kobject cannot export into sysfs. If want to export kobject to sysfs. We need function kobject add(). In general, one or both of parent and kset should be set appropriately before kobject add() is called. The help function kobject create and add() combines the work of kobject create() and kobject add() into one function.
- Add file into sysfs: Kobjects map to directories, and the complete object hierarchy maps to the complete sysfs structure. Sysfs is a tree without file to provide actual data. Here we have some controls for attributes.
- Default attributes: the default files class is provide through the field of kobject and kset. Therefore, all the kobject that have same types have same default file class under the corresponding sysfs directory. kobj type include a field, which is default attrs, it is an attribute structure array. This attribute responsible map the kernel data into a file that in sysfs.
- Create new attribute: In special situation, some kobject instance need to have its own attribute. Therefore, kernel provide a function sysfs create file() interface to create new attribute.
- Remove new attribute: Removing a attribute need the function sysfs remove file().

3.5.3 Changing the sda hard drive I/O scheduler with SysFS

```
$ cat / sys/ block / sda/ queue / scheduler
noop deadline [ cfq]
$ echo noop > / sys/ block / sda/ queue / scheduler
$ cat / sys/ block / sda/ queue / scheduler
[ noop ] deadline cfq
```

Another functionality which can be changed with SysFS is selinux. selinux is a kernel module which adds additional access control policies to Linux to help secure the kernel. selinux can be easily disabled by writing a 0 to /sys/fs/selinux/enforce, and reenabled by writing a 1.

3.5.4 Disabling selinux with SysFS

```

$ cat / sys/ fs/ selinux/ enforce
1
$ echo 0 > / sys/ fs / selinux/ enforce
$ cat / sys/ fs/ selinux/ enforce
0
$ echo 1 > / sys/ fs / selinux/ enforce
$ cat / sys/ fs/ selinux/ enforce
1
```

As we know, user has no authority to create arbitrary files in SysFS. Instead, only a certain kind of reference counted kernel structure known as a kobject can be added to this file system. kobjects added to the SysFS file system are organized in a tree hierarchy. Each kobject exposed to SysFS should have a parent kobject, a name, and a series of attributes. The kobject is exposed as a directory, and its attributes are exposed as files. As a consequence, each kobject and attributes name must be unique among its siblings since it is being exposed as a file name. In the example above, the selinux object had an attribute named enforce. Its parent was the fs object.

Each kobject attribute has a show function, which displays information about the kobject, and optionally a store function, which updates information in the kobject. Whenever an attribute is read from using the read system call, the function fills a provided buffer with a human readable ASCII representation of its value. Similarly, when an attribute is written to using the write system call, it unmarshals the human readable data in the buffer into a machine readable value, and then updates its value. Since they are represented as files, attributes have permissions indicating whether they can be read from or written to by various users. Files and directories under SysFS are regarded as owned by the root user.

kobjects with many attributes should have an attribute group structure. Large groups of kobjects can be encapsulated in a special type of object known as a kset. Like normal kobjects, ksets are represented as directories and have parents, however they do not have attributes.

4 IMPLEMENT SYSFS TO VMCS

We modified five files, which are Makefile, Kconfig, vmx.c, vmcs-sysfs.h and vmcs-sysfs.c for the design. The notion and method related to the design on each file is illustrated.

4.1 Makefile

In order to compile all the files, we need to add a line in Makefile for compiling vmcs-sysfs.c

```
obj \$ (CONFIG\_VMCS\_SYSFS) += vmcs\_sysfs.o
```

4.2 Kconfig

We added config VMCS_SYSFS to control the VMCS_SYSFS. We also used bool, depends, KVM, and KVM_INTEL to control and compile the v,cs-sysfs.c file.

```
config VMCS_SYSFS
        bool "Sysfs interface to VMCS"
        depends on KVM && KVM_INTEL
        default n
```

4.3 vmx.c

We follow the paper by Ian Kronquist to modify the vmx.c. The method of hardware_disable was used to start code of vmcs-sysfs.

4.4 vmcs-sysfs.c & and vmcs-sysfs.h

Based on paper by Ian Kronquist, we created some special functions as below.

- vmcs-sysfs.c & vmcs-sysfs.h : sort kobject
- reg_vmcs_sysfs & unreg_vmcs_sysfs : feature of sysfs
- kset_create_and_add : create VMCS fields
- sysfs_create_group : add the attribute
- kset_unregister : delete the folder created

5 Setting up the environment

To do the project, we need to install CentOS 7 and 4.1.5 linux kernel into MinnowBoard first. MinnowBoard is a developer program offering performance, flexibility, openness, and support of standards for the smallest of devices. In the project, we made use of the USB port, Ethernet port, micro HDMI port, SD card socket, and power input to set up the environment.

5.1 Install CentOS 7 64-bit

The CentOS (Community Enterprise Operating System) is a community-driven free software focused on delivering a robust open source ecosystem. It offers a consistent and manageable platform for a wide variety of deployments. For open source communities, it offers a solid, predictable base to build upon, along with extensive resources to build, test, release, and maintain the code.

First, we have to make a USB flash disk from Cent OS 7 and then follow steps below to install Cent OS 7.

- Connect to monitor with Minnowboard with HDMI to micro HDMI.

- Insert SD card into Minnowbroad.
- Plug cable into Minnowbroad.
- Insert USB flash which contains Cent OS 7.
- Plug power into Minnowbroad.
- Boot the Minnowbroad.
- Press F2 or Delete to enter to the BIOS menu.
- Choose EFI USB boot mode to boot Minnowbroad.
- Choose Cent OS 7 to install the operating system.
- Select English(United State) for language and press continue.
- Select Day&Time and choose current location.
- Click on install destination and choose I will configure partitioning.
- Click on Network&Hostname and click auto detection of Ethernet.
- Click Begin to installation
- Setup Root password and create User.
- After complete installation of Cent OS 7, reboot Minnowbroad.
- Reboot Cent OS 7 and then start installing 4.1.5 Linux kernel.

5.2 Install 4.1.5 Linux kernel

The purpose of using Linux kernel is testing the implementation of sysfs to VMCS. The installation steps are as follow.

- Key in the ID and password to enter the root.
- Key in the the following command line.
  - yum install gcc ncurses ncurses-devel update bzip2 wget bzip2 bzip2-libs bc perl
  - wget git.yoctoproject.org/cgit/cgit.cgi/linux-yocto-4.1/snapshot/linux-yocto-4.1-4.1.5.tar.bz2 tar xvjf linux4.1-4.1.5.tar.bz2
  - cd /usr/src/linux-yocto-4.1-4.1.5
  - wget http://www.elinux.org/images/e/e2/Minnowmax-3.18.txt
  - make defconfig
  - scripts/kconfig/merge_config.sh .config Minnowmax-3.18.txt
- Key in the command: make menuconfig, and change the configs as steps below.
  - Device Drivers –
    Multi-device support (RAID and LVM) –
    ⟨∗⟩ Multiple devices driver support (RAID and LVM)
    ⟨⟩ Device mapper support
  - Power management and ACPI options –
    ⟨∗⟩ ACPI (Advanced Configuration and Power Interface)Support
  - Processor type and features –
    ⟨∗⟩ EFI runtime service support
    ⟨∗⟩ EFI stub support

  - Firmware Drivers –
    EFI (Extensible Firmware Interface)Support –
    ⟨∗⟩ EFI Variable Support via sysfs

- Make files and modules by following commands.
  - make -j4 && make -j4 modules
  - make modules_install && make install

6 Conclusion

The result of the project is that we created a vmcs_sysfs file under the /sys/kernel/. This is the first step of implementing sysfs to VMCS. There are many other potential to add more features to the VMCS. Though it is only the beginning of the project, we still spent a lot of time figuring the method. Sometimes we found it hard to get to the next step, while sometimes we needed to ask other groups to make sure our method is correct. However, it was a hard but meaningful period that we learned a lot from this project.

By using the mininal config file (wget http://classes.engr.oregonstate.edu/eecs/fall2015/cs444/examples/minimal.config) given by the instruction, we bumped into the problem that we could not successfully boot the system. We tried many times but still failed. As an alternative way, we found another config file on the Internet, which help us successfully boot the system. We learned how to debug while there are some errors and how to get the solution to reach our goal.

When we test the MinnowBoard, it took us a long time to test because the compile time and make file time is long. As a result, we need to do our best to make sure our code is correct before running. However, things didn't go well. Even though we thought the code was right, we still spent a lot of time on retesting the code, compiling the code and rewriting the code. Moreover, in order to let MinnowBroad work, we have to use mouse, keyboard, screen and HDMI; however, we didn't have these equipments in the first place. Therefore, we ask the library to borrow these equipments. But these equipments only viable for borrowing for six hours, which is not efficient enough for us to test our MinnowBroad, meaning that we have to buy these equipments at Beaver store. In the future, we think we can use VM for testing the kernel code first, and then run the code on the MinnowBoard.

7 Patch File

```
1  Binary files Original/.DS_Store and New/.DS_Store differ
2  diff -u Original/Kconfig.octet-stream New/Kconfig.octet-stream
3  --- Original/Kconfig.octet-stream       2018-06-12 17:46:54.000000000 -0700
4  +++ New/Kconfig.octet-stream    2018-06-12 19:28:12.000000000 -0700
5  @@ -1,104 +1,113 @@
6  -#
7  -# KVM configuration
8  -#
9  -
10 -source "virt/kvm/Kconfig"
11 -
```

```
12 -menuconfig VIRTUALIZATION
13 -       bool "Virtualization"
14 -       depends on HAVE_KVM || X86
15 -       default y
16 -       ---help---
17 -         Say Y here to get to see options for using your Linux host to run other
18 -         operating systems inside virtual machines (guests).
19 -         This option alone does not add any kernel code.
20 -
21 -         If you say N, all options in this submenu will be skipped and disabled.
22 -
23 -if VIRTUALIZATION
24 -
25 -config KVM
26 -       tristate "Kernel-based Virtual Machine (KVM) support"
27 -       depends on HAVE_KVM
28 -       depends on HIGH_RES_TIMERS
29 -       # for TASKSTATS/TASK_DELAY_ACCT:
30 -       depends on NET
31 -       select PREEMPT_NOTIFIERS
32 -       select MMU_NOTIFIER
33 -       select ANON_INODES
34 -       select HAVE_KVM_IRQCHIP
35 -       select HAVE_KVM_IRQFD
36 -       select HAVE_KVM_IRQ_ROUTING
37 -       select HAVE_KVM_EVENTFD
38 -       select KVM_APIC_ARCHITECTURE
39 -       select KVM_ASYNC_PF
40 -       select USER_RETURN_NOTIFIER
41 -       select KVM_MMIO
42 -       select TASKSTATS
43 -       select TASK_DELAY_ACCT
44 -       select PERF_EVENTS
45 -       select HAVE_KVM_MSI
46 -       select HAVE_KVM_CPU_RELAX_INTERCEPT
47 -       select KVM_GENERIC_DIRTYLOG_READ_PROTECT
48 -       select KVM_VFIO
49 -       select SRCU
50 -       ---help---
```

```
51 –          Support hosting fully virtualized guest machines using hardware
52 –          virtualization extensions.  You will need a fairly recent
53 –          processor equipped with virtualization extensions. You will also
54 –          need to select one or more of the processor modules below.
55 –
56 –          This module provides access to the hardware capabilities through
57 –          a character device node named /dev/kvm.
58 –
59 –          To compile this as a module, choose M here: the module
60 –          will be called kvm.
61 –
62 –          If unsure, say N.
63 –
64 –config KVM_INTEL
65 –        tristate "KVM for Intel processors support"
66 –        depends on KVM
67 –        # for perf_guest_get_msrs():
68 –        depends on CPU_SUP_INTEL
69 –        ---help---
70 –          Provides support for KVM on Intel processors equipped with the VT
71 –          extensions.
72 –
73 –          To compile this as a module, choose M here: the module
74 –          will be called kvm-intel.
75 –
76 –config KVM_AMD
77 –        tristate "KVM for AMD processors support"
78 –        depends on KVM
79 –        ---help---
80 –          Provides support for KVM on AMD processors equipped with the AMD-V
81 –          (SVM) extensions.
82 –
83 –          To compile this as a module, choose M here: the module
84 –          will be called kvm-amd.
85 –
86 –config KVM_MMU_AUDIT
87 –        bool "Audit KVM MMU"
88 –        depends on KVM && TRACEPOINTS
89 –        ---help---
```

```
90 -          This option adds a R/W kVM module parameter 'mmu_audit', which allows
91 -          auditing of KVM MMU events at runtime.
92 -
93 -config KVM_DEVICE_ASSIGNMENT
94 -         bool "KVM legacy PCI device assignment support"
95 -         depends on KVM && PCI && IOMMU_API
96 -         default y
97 -         ---help---
98 -           Provide support for legacy PCI device assignment through KVM.  The
99 -           kernel now also supports a full featured userspace device driver
100 -          framework through VFIO, which supersedes much of this support.
101 -
102 -          If unsure, say Y.
103 -
104 -# OK, it's a little counter-intuitive to do this, but it puts it neatly under
105 -# the virtualization menu.
106 -source drivers/vhost/Kconfig
107 -source drivers/lguest/Kconfig
108 -
109 -endif # VIRTUALIZATION
110 +# Group 6
111 +
112 +#
113 +# KVM configuration
114 +#
115 +
116 +source "virt/kvm/Kconfig"
117 +
118 +menuconfig VIRTUALIZATION
119 +         bool "Virtualization"
120 +         depends on HAVE_KVM || X86
121 +         default y
122 +         ---help---
123 +           Say Y here to get to see options for using your Linux host to run other
124 +           operating systems inside virtual machines (guests).
125 +           This option alone does not add any kernel code.
126 +
127 +           If you say N, all options in this submenu will be skipped and disabled.
128 +
```

```
129 +if VIRTUALIZATION
130 +
131 +config KVM
132 +       tristate "Kernel-based Virtual Machine (KVM) support"
133 +       depends on HAVE_KVM
134 +       depends on HIGH_RES_TIMERS
135 +       # for TASKSTATS/TASK_DELAY_ACCT:
136 +       depends on NET
137 +       select PREEMPT_NOTIFIERS
138 +       select MMU_NOTIFIER
139 +       select ANON_INODES
140 +       select HAVE_KVM_IRQCHIP
141 +       select HAVE_KVM_IRQFD
142 +       select HAVE_KVM_IRQ_ROUTING
143 +       select HAVE_KVM_EVENTFD
144 +       select KVM_APIC_ARCHITECTURE
145 +       select KVM_ASYNC_PF
146 +       select USER_RETURN_NOTIFIER
147 +       select KVM_MMIO
148 +       select TASKSTATS
149 +       select TASK_DELAY_ACCT
150 +       select PERF_EVENTS
151 +       select HAVE_KVM_MSI
152 +       select HAVE_KVM_CPU_RELAX_INTERCEPT
153 +       select KVM_GENERIC_DIRTYLOG_READ_PROTECT
154 +       select KVM_VFIO
155 +       select SRCU
156 +       ---help---
157 +         Support hosting fully virtualized guest machines using hardware
158 +         virtualization extensions.  You will need a fairly recent
159 +         processor equipped with virtualization extensions. You will also
160 +         need to select one or more of the processor modules below.
161 +
162 +         This module provides access to the hardware capabilities through
163 +         a character device node named /dev/kvm.
164 +
165 +         To compile this as a module, choose M here: the module
166 +         will be called kvm.
167 +
```

```
168 +          If unsure, say N.
169 +
170 +config KVM_INTEL
171 +        tristate "KVM for Intel processors support"
172 +        depends on KVM
173 +        # for perf_guest_get_msrs():
174 +        depends on CPU_SUP_INTEL
175 +        ---help---
176 +          Provides support for KVM on Intel processors equipped with the VT
177 +          extensions.
178 +
179 +          To compile this as a module, choose M here: the module
180 +          will be called kvm-intel.
181 +
182 +config KVM_AMD
183 +        tristate "KVM for AMD processors support"
184 +        depends on KVM
185 +        ---help---
186 +          Provides support for KVM on AMD processors equipped with the AMD-V
187 +          (SVM) extensions.
188 +
189 +          To compile this as a module, choose M here: the module
190 +          will be called kvm-amd.
191 +
192 +config KVM_MMU_AUDIT
193 +        bool "Audit KVM MMU"
194 +        depends on KVM && TRACEPOINTS
195 +        ---help---
196 +         This option adds a R/W kVM module parameter 'mmu_audit', which allows
197 +         auditing of KVM MMU events at runtime.
198 +
199 +config KVM_DEVICE_ASSIGNMENT
200 +        bool "KVM legacy PCI device assignment support"
201 +        depends on KVM && PCI && IOMMU_API
202 +        default y
203 +        ---help---
204 +          Provide support for legacy PCI device assignment through KVM.  The
205 +          kernel now also supports a full featured userspace device driver
206 +          framework through VFIO, which supersedes much of this support.
```

```
+
+          If unsure, say Y.
+
+config VMCS_SYSFS
+        bool "Sysfs interface to VMCS"
+        depends on KVM && KVM_INTEL
+        default n
+        ---help---
+                Provides a sysfs interface to control the VMCS. Also dumps the contents of the VMC
+
+# 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 -u Original/Makefile.octet-stream New/Makefile.octet-stream
--- Original/Makefile.octet-stream      2018-06-12 17:46:56.000000000 -0700
+++ New/Makefile.octet-stream   2018-06-12 19:28:17.000000000 -0700
@@ -1,22 +1,25 @@
-
-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
-
```

```
246 -obj-$(CONFIG_KVM)          += kvm.o
247 -obj-$(CONFIG_KVM_INTEL)          += kvm-intel.o
248 -obj-$(CONFIG_KVM_AMD)    += kvm-amd.o
249 +# Group 6
250 +
251 +ccflags-y += -Iarch/x86/kvm
252 +
253 +CFLAGS_x86.o := -I.
254 +CFLAGS_svm.o := -I.
255 +CFLAGS_vmx.o := -I.
256 +
257 +KVM := ../../../virt/kvm
258 +
259 +kvm-y                   += $(KVM)/kvm_main.o $(KVM)/coalesced_mmio.o \
260 +                               $(KVM)/eventfd.o $(KVM)/irqchip.o $(KVM)/vfio.o
261 +kvm-$(CONFIG_KVM_ASYNC_PF)      += $(KVM)/async_pf.o
262 +
263 +kvm-y                   += x86.o mmu.o emulate.o i8259.o irq.o lapic.o \
264 +                           i8254.o ioapic.o irq_comm.o cpuid.o pmu.o
265 +kvm-$(CONFIG_KVM_DEVICE_ASSIGNMENT)     += assigned-dev.o iommu.o
266 +kvm-intel-y             += vmx.o
267 +kvm-amd-y               += svm.o
268 +
269 +obj-$(CONFIG_KVM)          += kvm.o
270 +obj-$(CONFIG_KVM_INTEL)          += kvm-intel.o
271 +obj-$(CONFIG_KVM_AMD)    += kvm-amd.o
272 +
273 +obj-$(CONFIG_VMCS_SYSFS) += vmcs-sysfs.o
274  diff -u Original/vmcs-sysfs.c New/vmcs-sysfs.c
275 --- Original/vmcs-sysfs.c       2018-06-12 18:00:32.000000000 -0700
276 +++ New/vmcs-sysfs.c    2018-06-12 19:29:13.000000000 -0700
277 @@ -0,0 +1,101 @@
278 +// Group 6
279 +
280 +/*Ref */
281 +#include "vmcs-sysfs.h"
282 +
283 +struct attribute rip_a = {
284 +        .name = "g_rip_a",
```

```
285 +       .mode = S_IRWXUGO,
286 +};
287 +struct attribute rip_b = {
288 +       .name = "g_rip_b",
289 +       .mode = S_IRWXUGO,
290 +};
291 +struct attribute rip_c = {
292 +       .name = "g_rip_c",
293 +       .mode = S_IRWXUGO,
294 +};
295 +
296 +
297 +/*static unsigned long guest_activity_state;
298 +module_param(guest_activity_state, ulong, 0644);
299 +*/
300 +struct kobject kobj;
301 +static struct kset *vmcs_sysfs_set;
302 +
303 +static struct attribute *vmcs_sysfs_attrs[] = {
304 +      NULL,
305 +};
306 +
307 +static inline struct vmcs_sysfs *vmcs_sysfs_container_of(struct kobject *kobj)
308 +{
309 +       return container_of(kobj, struct vmcs_sysfs, kobj);
310 +}
311 +
312 +static void vmcsctl_release(struct kobject *kobj)
313 +{
314 +       struct vmcs_sysfs *vmcs_sysfs = container_of(kobj, struct vmcs_sysfs, kobj);
315 +
316 +       kfree(vmcs_sysfs);
317 +}
318 +
319 +static struct kobj_type kt = {
320 +       .release = vmcsctl_release,
321 +       .sysfs_ops = &kobj_sysfs_ops,
322 +};
323 +
```

```
+static struct attribute_group attr_group = {
+       .attrs = vmcs_sysfs_attrs,
+};
+
+int reg_vmcs_sysfs(struct vmcs *vmcs)
+{
+       int err;
+       struct vmcs_sysfs *vmcs_sysfs = kzalloc(sizeof(*vmcs_sysfs), GFP_KERNEL);
+
+       kobject_init(&vmcs_sysfs->kobj, &kt);
+       vmcs_sysfs->vmcs = vmcs;
+       vmcs_sysfs->pid = task_pid_nr(current);
+       vmcs_sysfs->kobj.kset = vmcs_sysfs_set;
+
+       err = kobject_add(&vmcs_sysfs->kobj, NULL, "vmcs%d", vmcs_sysfs->pid);
+       if (err != 0)
+               goto err_flag;
+
+       err = sysfs_create_group(&vmcs_sysfs->kobj, &attr_group);
+       if (err != 0)
+               goto err_flag;
+
+       return 0;
+
+err_flag:
+
+       return err;
+}
+
+void unreg_vmcs_sysfs(struct vmcs *vmcs)
+{
+}
+
+static int vmcs_sysfs_init(void)
+{
+       /*create a struct kset dynamically and add it to sysfs*/
+       vmcs_sysfs_set = kset_create_and_add("vmcs_sysfs", NULL, kernel_kobj);
+       printk(KERN_INFO "VMCS vmcs_sysfs_init \n");
+
```

```
363 +       return 0;
364 +}
365 +
366 +static void vmcs_sysfs_exit(void)
367 +{
368 +       printk(KERN_INFO "VMCS vmcs_sysfs_exit\n");
369 +
370 +       kset_unregister(vmcs_sysfs_set);
371 +}
372 +
373 +module_init(vmcs_sysfs_init);
374 +module_exit(vmcs_sysfs_exit);
375 +
376 +MODULE_LICENSE("GPL");
377 +MODULE_AUTHOR("Author");
378 +MODULE_DESCRIPTION("Implementing a sysfs interface to the VMCS");
379 diff -u Original/vmcs-sysfs.h New/vmcs-sysfs.h
380 --- Original/vmcs-sysfs.h       2018-06-12 18:00:36.000000000 -0700
381 +++ New/vmcs-sysfs.h    2018-06-12 19:28:24.000000000 -0700
382 @@ -0,0 +1,27 @@
383 +// Group 6
384 +
385 +#include <linux/fsnotify_backend.h>
386 +#include <linux/init.h>
387 +#include <linux/inotify.h>
388 +#include <linux/kernel.h>
389 +#include <linux/module.h>
390 +#include <linux/stat.h>
391 +#include <linux/slab.h>
392 +#include <linux/string.h>
393 +#include <linux/sysfs.h>
394 +#include <linux/sched.h>
395 +#include <linux/syscalls.h>
396 +#include <asm/vmx.h>
397 +
398 +
399 +struct vmcs;
400 +
401 +struct vmcs_sysfs {
```

```
+       int pid;
+       struct kobject kobj;
+       struct vmcs *vmcs;
+};
+
+int reg_vmcs_sysfs(struct vmcs *vmcs);
+
+void unreg_vmcs_sysfs(struct vmcs *vmcs);
diff -u Original/vmx.c New/vmx.c
--- Original/vmx.c      2018-06-12 17:46:59.000000000 -0700
+++ New/vmx.c   2018-06-12 19:29:05.000000000 -0700
@@ -1,20 +1,4 @@
-/*
- * Kernel-based Virtual Machine driver for Linux
- *
- * This module enables machines with Intel VT-x extensions to run virtual
- * machines without emulation or binary translation.
- *
- * Copyright (C) 2006 Qumranet, Inc.
- * Copyright 2010 Red Hat, Inc. and/or its affiliates.
- *
- * Authors:
- *   Avi Kivity   <avi@qumranet.com>
- *   Yaniv Kamay  <yaniv@qumranet.com>
- *
- * This work is licensed under the terms of the GNU GPL, version 2.  See
- * the COPYING file in the top-level directory.
- *
- */
+// Group 6

 #include "irq.h"
 #include "mmu.h"
@@ -49,6 +33,12 @@

 #include "trace.h"

+/*HW3*/
+# ifdef VMCS_SYSFS
```

```
441 +# include " vmcs-sysfs.h"
442 +# endif
443 +
444 +
445  #define __ex(x) __kvm_handle_fault_on_reboot(x)
446  #define __ex_clear(x, reg) \
447         ____kvm_handle_fault_on_reboot(x, "xor " reg " , " reg)
448 @@ -162,11 +152,15 @@
449  #define NR_AUTOLOAD_MSRS 8
450  #define VMCS02_POOL_SIZE 1
451
452 +/*
453 +HW3
454  struct vmcs {
455         u32 revision_id;
456         u32 abort;
457         char data[0];
458  };
459 +*/
460 +
461
462  /*
463   * Track a VMCS that may be loaded on a certain CPU. If it is (cpu!=-1), also
464 @@ -2943,6 +2937,12 @@
465                 kvm_cpu_vmxoff();
466         }
467         cr4_clear_bits(X86_CR4_VMXE);
468 +
469 +       /*HW3*/
470 +       # ifdef VMCS_SYSFS
471 +       reg_vmcs_sysfs(vmcs);
472 +       # endif
473 +
474  }
475
476  static __init int adjust_vmx_controls(u32 ctl_min, u32 ctl_opt,
477 @@ -9271,7 +9271,7 @@
478         /* vmcs12's VM_ENTRY_LOAD_IA32_EFER and VM_ENTRY_IA32E_MODE are
479          * emulated by vmx_set_efer(), below.
```

```
480            */
481 -          vm_entry_controls_init(vmx,
482 +          vm_entry_controls_init(vmx,
483                    (vmcs12->vm_entry_controls & ~VM_ENTRY_LOAD_IA32_EFER &
484                            ~VM_ENTRY_IA32E_MODE) |
485                    (vmcs_config.vmentry_ctrl & ~VM_ENTRY_IA32E_MODE));
486  diff -u Original/vmx.h New/vmx.h
487 --- Original/vmx.h      2018-06-12 17:47:01.000000000 -0700
488 +++ New/vmx.h   2018-06-12 19:28:58.000000000 -0700
489 @@ -1,26 +1,5 @@
490 -/*
491 - * vmx.h: VMX Architecture related definitions
492 - * Copyright (c) 2004, Intel Corporation.
493 - *
494 - * This program is free software; you can redistribute it and/or modify it
495 - * under the terms and conditions of the GNU General Public License,
496 - * version 2, as published by the Free Software Foundation.
497 - *
498 - * This program is distributed in the hope it will be useful, but WITHOUT
499 - * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
500 - * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
501 - * more details.
502 - *
503 - * You should have received a copy of the GNU General Public License along with
504 - * this program; if not, write to the Free Software Foundation, Inc., 59 Temple
505 - * Place - Suite 330, Boston, MA 02111-1307 USA.
506 - *
507 - * A few random additions are:
508 - * Copyright (C) 2006 Qumranet
509 - *    Avi Kivity <avi@qumranet.com>
510 - *    Yaniv Kamay <yaniv@qumranet.com>
511 - *
512 - */
513 +// Group 6
514 +
515  #ifndef VMX_H
516  #define VMX_H
```

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 22 May. 2018].

[2] Ian Kronquist. *Implementing a Linux Sysfs Interface to the VMCS*. Oregon State University. 2017. [online] Available at: https://ir.library.oregonstate.edu/concern/honors_college_theses/rb68xd807. [Accessed 22 May. 2018].

[3] Andries Brouwer. (2003). *The Linux kernel: Sysfs and kobjects.* [online] Available at: https://www.win.tue.nl/ aeb/linux/lk/lk-13.html [Accessed 25 May 2018].

[4] Wowo (2014). *Linux_Kobject*. [online] Available at: http://www.wowotech.net/linux_kenrel/kobject.html [Accessed 25 May 2018]

[5] Kroah-Hartman, G. (2018). *Everything you never wanted to know about kobjects, ksets, and ktypes*. [online] Kernel.org. Available at: https://www.kernel.org/doc/Documentation/kobject.txt [Accessed 25 May 2018].

[6] minnowboard.org. 2018. Available at: https://minnowboard.org [Accessed 12 Jun 2018].

[7] CentOS. 2018. Available at: https://www.centos.org [Accessed 12 Jun 2018].

[8] Cezar, M. (2014). *Installation of "CentOS 7.0*. [online] Tecmint.com. Available at: https://www.tecmint.com/centos-7-installation/ [Accessed 12 Jun. 2018].