Final Project:Implementing a Sysfs interface to the VMCS

Author : Yi-Jung Chiang

CS 544

Spring 2017

June 14, 2017

Abstract

The purpose of this project is to complete a Sysfs interface to the VMCS. VMCS is the virtual machine control structure. It is responsible for organizing and tracking the virtual CPU in the VMX/KVM layer of the kernel. Sysfs is a pseudo file system provided by the kernel that exports information about various kernel subsystems, hardware devices, and associated device drivers. This report will show how to use CentOS7 64-bit with Minnowboard and install the 4.1.5 linux kernel.

CONTENTS

I. Introduction

In this project, I understand how to build a sysfs interface to the VMCS. Sysfs is a psedudo file system that is provided by the Linux kernel. It also can export data to user space through virtual files. I use Minnowboard as hardware, CentOS, and Linux 4.1.5. In this report, I will introduce concept of VMCS and how to install the operating system.

II. VMCS

A. Introduction of VMCS

VT-x(Virtualization Techonology for x86 CPUs) includes some parts. There are two new processor execution-modes. One is VMX root mode(for VM Managers), another one is non-root mode(for VM Guests). Also, it contains 10 new hardware instructions and six parts of VMCS data-structure. Moreover, there is a variety of control-options for VMs. Between VMs(guest) and VMM(host), there are VM Exit, VM Entry, VMXON and VM OFF.

VMM(Virtual-Machine Monitors) is a host. It has right to control the processors and hardware of platform. Also, it allows the guests directly run in logical processor. VMM can control information of processors, physical storage, interrupt and choices of I/O. VM(Virtual-Machine) is a guest. Actually, it supplies a kind of software environment which can maintain hosts that includes OS and application software. Each execution is independent and uses same physical platform. Furthermore, these hosts do not know extinction of VMM. The software is running on VM is limited.

Access to the VMCS is managed through a component of processor state called the VMCS pointer (one per logical processor). The value of the VMCS pointer is the 64-bit address of the 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. A VMM could use a different VMCS for each virtual machine that it supports. For a virtual machine with multiple logical processors (virtual processors), the VMM could use a different VMCS for each virtual processor. Software must first initialize each VMCS.

B. VMCS Region

VMCS region is a storage region that associates a logical processor with each VMCS. Software refers to a specific VMCS using the 64-bit physical address of the region (a VMCS pointer). VMCS pointers must be aligned to a 4K limit (bits 11: 0 must be zero). These pointers must not set bits beyond the physical address width of the processor.
VMCS revision identifiers use the first 32 bits of the VMCS region. Processors that maintain VMCS data in different formats use different VMCS revision identifiers.
VMX abort indicator use the next 32 bits of the VMCS region. It does not control the processor operation. A virtual processor writes a non-zero value to these bits when a VMX is aborted. Software can also write in this field.

VMCS data will remain the VMCS region that it uses to control VMX non-root operation and the VMX transitions. The format of this data is implementation-specific.

C. Types of VMCS

VMCS includes two types which are ordinary and shadow. The shadow-VMCS indicator in the VMCS region can decide the type. Shadow VMCS is just supported in CPU of 1-setting of the VMCS shadowing VM-execution. The difference of these two types is ordinary VMCS can be used in VM-Entry but shadow VMCS cannot be used.

D. VMCS Data

There are six parts which are included in VMCS data:

- Guest-state area: Processor state saves into the guest-state area when VM exits and it loads from guest-state area when VM entries.
- Host-state area: Processor state is loaded from the host-state area when VM exits.
- VM-execution control fields: Control processor in VMX non-root operation.
- VM-exit control fields: Control VM exits
- VM-entry control fields: Control VM entries
- VM-exit information fields: Receive information on VM exits and describe the reason of VM-exit. These fields are read-only

E. Guest-state area

The fields contained in the guest-state area of the VMCS. Processor state is loaded from the fields in every VM entry and stored into the fields in every VM exit.

F. Host-state area

The fields contained in the host-state area of the VMCS. Processor state is loaded from the fields in every VM exit and stored into the fields in every VM entry.

G. VM-execution control fields

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

H. VM-Exit Control Fields

The VM-exit control fields manage the VM exits.

I. VM-Entry Control Fields

The VM-exit control fields manage the VM entries.

J. VM-Exit Information Fields

The VMCS contains a section of read-only fields which includes information about the most recent VM exit.

K. VM-Instruction Error Field

The 32-bit VM-instruction error field does not provide information about the most recent VM exit. In fact, it is not modified on VM exits. Instead, it provides information about errors encountered by a non-faulting execution of one of the VMX instructions.

III. CentOS7 install

CentOS (CommunityenterpriseOperatingSystem), released in May 2004, is a totally free operating system distribution based upon the Linux kernel. It is derived entirely from the Red Hat Enterprise Linux (RHEL) distribution. CentOS exists to provide a free enterprise class computing platform and strives to maintain totally binary compatibility with its upstream source, Red Hat. It will be used in this project and install it on the Minnowboard Turbot.
Minnowboard Turbot is a MinnowBoard MAX compatible board developed and produced by ADI Engineering. It includes the processor of the 64-bit Intel Atom E38xx Series SoC and an integrated Intel HD graphics developed for Linux OS. Therefore, we can install CentOS 7 on it, amd rim the Linux 4.1.5 drauf. There is a port that is important on the Minnowboard Turbot.

- HDMI: The MinnowBoard uses a Type D micro HDMI connector. This is a default port. Cables and adapters can be easily picked up from most electronics stores.
- Ethernet: The MinnowMax uses a Realtek RTL8111GS-CG PCIe-based chipset to provide 10/100/1000 Ethernet connectivity.
- Low Speed Expansion (Top): The low speed connector uses 0.1 ”(2.54 mm) male circlips in a 2 x 13 array for a total of 26 pins. Pin 1 is closest to the row closest to the power connector and the edge of the vehicle.
- High Speed Expansion (Bottom): The high-speed connector uses a TE Connectivity-compatible 60-pin header. The generally recommended header is the 3-5177986-2 or the 60POS .8MM FH 8H GOLD part that rises 7.85mm so that 3/8 ”spacers can be used at the corners to attach the bait to the minnowboard.

A. Installation

Using CentOS:

1) Make USB flash disk: Go to https://www.centos.org/download/ and get the CentOS 7 ISO Image. After downloading the ISO image. Then boot the USB system. There is a lot of software can create a bootable USB flash disk. After USB is established, we can use the USB to install the system.

2) Install CentOS:

- Connect HDMI.
- Insert SD card.
- Insert USB into the Minnowboard
- power on the Minnowboard.
- press F2 go to the BIOS menu, choose EFI USB to boot it.
- After boot the system, choose the install CentOS7 option to install OS
- Select the language:English(United States).
- Choose the install location and enter the disk partition interface
- Begin installation.
- Set the password of the root.
- After install the system, restart the operating system, and use root(account) to log in.
- Set network: 1. cd /etc/sysconfig/network-scripts/ enter the network configuration directory. 2. vi ifcfg-eno16777736
- Add and modify some content below, and save the the document.
  - TYPE=Ethernet
  - BOOTPROTO=static
  - DEFROUTE=yes
  - PEERDNS=yes
  - PEERROUTES=yes
  - IPV4_FAILURE_FATAL=no
  - IPV6INIT=yes
  - IPV6_AUTOCONF=yes
  - IPV6_DEFROUTE=yes
  - IPV6_PEERDNS=yes
  - IPV6_PEERROUTES=yes
  - IPV6_FAILURE_FATAL=no
  - NAME=eno16777736
  - UUID=ae0965e7-22b9-45aa-8ec9-3f0a20a85d11
  - ONBOOT=yes
- Enter: service network restart

IV. Install Linux 4.1.5

A. Download Linux 4.1.5

- Install Linux 4.1.5 in CentOS in Minnowboard
- Activate the Ethernet interface first. If enpos3 Command is enabled Ethernet, to enable Ethernet after booting, you must change the system interface configuration file. Change the ONBOOT value to yes.
- Log in with root account.
- Install the dependency to compile Linux kernel: 1.yum install gcc ncurses ncurses devel
  2.yum update
- Go to the directory: /tmp and download the Linux 4.1.5 from : http://www.kernel.org/pub/linux/kernel/v4.x/linux-4.1.5.tar.xz 1.cd /tmp
  2. wget http:// www.kernel.org/pub/linux/kernel/v4.x/linux-4.1.5.tar.xz
- extract the Linux 4.1.5 in /usr/src/ 1. tar xvjf linux-4.1.5.tar.xz -C /usr/src
  2. cd /ssr/src/linux-4.1.5

B. Configuration

- Use the configure file that is from the minimum site .config, and use the command mv minimal.config .config to change the minimimal.config for our current configuration.
- Download the config files for Minnowboard. After downloading from http://www.elinux.org/Images/e/e2/M, we need to merge two kernel config files.
- Compile and install kernel, use command make j4 make j4 modules
- Install the new kernel and modules and use command make install make modulesinstall
- Use make menuconfig command to configure the Linux 4.1.5make menuconfig
- After making menuconfig command, we will enter the graphic interface to configure the core functions. Below is some useful option of the kernel configuration option. (In Menuconfig)

C. Menuconfig

After entering the graphical interface, they will configure havce several option. In addition, choose the core functionality, one should note a point: If you are sure that the core function will be used in the future, entered directly into the kernel. If the function may be used in the future, the module is entered as possible. In short, try to keep the core small and nice, the rest of the function is compiled into a module.Below is some detail information that is important.
After entering the graphical interface, they will configure havce several option. In addition, choose the core functionality, one should note a point: If you are sure that the core function will be used in the future, entered directly into the kernel. If the function may be used in the future, the module is entered as possible. In short, try to keep the core small and nice, the rest of the function is compiled into a module.Below is some detail information that is important.

1) Setup: Most revelant program interact with Linux, the core version description, use the development code and some information are set here. The program here is design for the correlation between core and program. In general, keep the default value well and do not cancel any project under it because it can cause problems with some programs that can not run at the same time. If you are in doubt, press Help and find help. You can select the features based on Help. Here are some useful information:

- Networking support: Internet support, this option must be choosed.
- PCI support: PCI Support, if you use the PCI card, must be choosed.
- PCI access mode: This has BIOS, Direct and Any option. Choosing the any option.
- Support for paging of anonymous memory (swap): This is the use of swap or swap file for virtual memory.
- System V IPC: It used to synchronize the processor between the program and exchange the information.
- POSIX Message Queues: This is the POSIX message queue.
- Auditing support: Use for the kernel and certain submodules simultaneously.
- Kernel .config support The .config configuration information stored in the kernel.
- Optimize for size: This option cause gcc use -Os parameters instead of -O2 to optimization parameters to get smaller kernel.

2) Enable Loadable module support: If your core needs to support dynamic kernel modules, you should be enable loadable module option. Below is some useful information of the option:

- Forced module unloading: Allow to unload a module that is used (more dangerous). This option allows you to forcibly suspend the module, even if the kernel considers it insecure. Kernel will remove the modules immediately, regardless of whether someone is using it. This is especially for developers and users who are impulsive. If you are unsure, choose N.

- Forced module loading: Allow Forced load module.

- Module unloading: Allow unloading already loaded module.

- Module versioning support: Sometimes, you need to compile the module. This option will add some version information to the compiled modules to provide independent features. Therefore, different kernels can distinguish it from the original module when use the same module.

3) Enable the block layer: Block device support, hard disk/USB/SCSI devies required use this option makes the block device can be removed from the kernel. If not select, the blockdev file will not available ,some file system such as ext3 will be unavailable. The block layer is enable by default, you can also enter the breakdown set in this project, select the function you need. Below is some useful information of the option: 1.Block layer SG support v4: Scsi generic block device the fourth Version support
2.Block layer data integrity support: Data integrity support for Block devices

4) I/O schedulers: IO scheduler I/O is input and output bandwidth control, mainly for the hard disk. Also, it is the core for kernel. It is used to decide in which order the block I/O operations will submitted to storage volumes.

- Deadline I/O scheduler: Polling scheduler, simple and compact, providing a minimum read latency and throughput well, especially adapted to the environment that have more read(such as a database). Deadline I/O scheduler is simple and close, its performance is as well as the preemptive scheduler and work better when some of the data in.

- CFQ I/O scheduler: Use QoS policies and assigned the same amount of bandwidth for all tasks, avoid process starvation and to achieve low latency. CFQ scheduler attempts to provide the same bandwidth for all processes. It will provide equal working environment, very suitable for a desktop system.

5) Types and Features of Processor: Enter the Processor type and features, choose the form of your actual CPU. Below is some useful information.

- High memory support: Large memory support, it can support up to 4G, 64G, generally can not choose it.

6) Power management and ACPI options: If you select ”Power management and ACPI options”, After that, it will enter the power management mechanism system. In fact, the power management mechanism also need to match the motherboard and the associated power-saving features of the CPU, then can actually achieve power-saving.

7) Device Drivers: Enter ”Device Drivers” This is a driver library for all hardware devices. This interface lets you select features and parameters for the build. Features can either be built-in, modularized, or ignored. Parameters must be entered in as decimal or hexadecimal numbers or text

8) Networking support: Network options, it is mainly about a number of options for network protocols. Linux is a network operating system, the most powerful feature of Linux is that the flexible support for network features

9) File system: The file in the Linux file system is a collection of data, the file system contains not only the data of the files but also the structure of the file system. The files, directories, soft links and file protection information that can see by All Linux users and programs are all stored in the file system.

V. Implementing a sysfs interface to the VMCS

This part introduces how to modify the Kconfig file, and how to modify the vmx.c. Moreover, we will introduce some functions of Kconfig and Kset. Kconfig is a file. When users is compiling linux kernel and inputs ”make menuconfig. The screen will show the lists which are about installation.

A. Kconfig

To enable and disable the VMCS SYSFS interface, we need add some contents to the /arch/x86/kvm/Kconfig. We need add information below:

```
config VMCS SYSFS
tristate "Enable VMCS SYSFS"
depends on KVM \&\& TRACEPOINTS
```

B. Kobject

Kobject is the heart of the device model. It is represented by structing kobject and defining in linux/kobject.h. It also provides basic facilities, such as reference counting, a name, and a parent pointer, enabling the creation of a hierarchy of object. Below is the structure.

```
struct kobject {
        const char *name;
        struct list_head entry;
        struct kobject *parent;
        struct kset *kset;
        struct kobj_type *ktype;
        struct sysfs_dirent *sd;
        struct kref kref;
        unsigned int state_initialized:1;
        unsigned int state_in_sysfs:1;
        unsigned int state_add_uevent_sent:1;
        unsigned int state_remove_uevent_sent:1;
        unsigned int uevent_suppress:1;
};
```

There are some information of details which are about the elements of the structure.

1. The name pointer point to the name of this Kobject.
2. The parent pointer points to this kobject’s parent. In this behavior, kobjects build an object hierarchy in the kernel

and enable the expression of the relationship between multiple objects. This is the Sysfs, which is a user-space filesystem representation of the kobject object hierarchy inside the kernel.

3. The sd pointer points to a sysfs_direct structure that represents this kobject in sysfs. Inside this structure, it is a n inode structure representing the kobject in the sysfs filesystem.

4. The kref structure provides reference counting.

5. the ktype and kset structures describe and group kobject.

Kobjects are usually embedded in other structures.

1) How to manage and control Kobject: First step is declaring and initializing a Kobject. Kobject is initialized via the function kobject_init. It is declared in linux/kobject.h.

The first parameter in the function kobject_init is the kobject that need to initialize. Before calling this function, the kobject must be cleaned. This might normally done during the initilization of the higher function which the kobject is embedded in. If it is not clean, we must call memset() to do this trick.

After cleaning, we can initialize the parent and kset safely.

We also can use kobject_create() to tackle it automatic, it return a new allocate kobject. Most time, we should be use kobject_create() or a related helper function rather than directly manipulate the structure.

2) Reference Counts: One of the primary feature is provided by kobject which is a unified reference counting system. After it is initialized, the kobject's reference count is set to one. Thus, it is long as the reference count without nonzero, the object continues to exist from memory and it is pinned. Below is some useful functions:

1. Increase reference count: Increment the reference count is done via kobject_get(), declared in linux/kobject.h

2. Decrease reference count: Decrease reference count is done via kobject_put(), declared in linux/kobject.h

3) kref: The kobject reference counting is supported by the kref structure, it is defined in linux/kref.h. before we use kref. Then, you must initialize it through kref_init(). If you want to get the reference of kref, we need to call kref_get() function, it is declared in linux/kref.h. This function increase the reference count and it does not have to return value. kref_put() decrease the reference fo kref, it declared in linux/kref.h. If the count is decreasing to zero, then we need call release() function.

C. Ktypes

Kobject is relative to a special type, it is Ktype(kernel object type). Ktype is represented by struct kobj_type, it is 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. Then, the

function encoding specified property values that are stored in the buffer and returned to the user mode. The store () function is used to store property values that come from user mode.The detail structure is below:

```
struct kobj_type {
void (*release)(struct kobject *);
const struct sysfs_ops *sysfs_ops;
struct attribute **default_attrs;
};
```

There are the details about instructions of the elements in kobj_typ:

1. 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.
2. sysfs_ops variable points to a sysfs_ops structure. This structure describe the behavior of sysfs files on read and write.
3. 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 to describe the behavior of a group of kobject, instead of each kobject. Also, Ktype defines its own behavior and the behavior normal kobject is defined in ktype structure once. Last, all the similar kobjects can share the same feature.

D. Sysfs

Sysfs is a ram-based filesystem which initially bases on ramfs. It provides export kernel data structures, attributes, and the linkages between them to userspace. Sysfs file system is a special file system that is similar to a proc file system. It uses for organizing the device in the system into a hierarchy and provides detailed information of kernel data structures for user-mode programs. Under /sys is some directory about sysfs:

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

The most important directory is device. This directory is exporting the device module into user-space.

1) Add and Delete kobject from sysfs: Only initializing the kobject cannot export into sysfs. If we want to export kobject to sysfs, We need function kobject_add(). In general, one or both of parents and kset should be set appropriately before kobject_add() is called. The helpful function kobject_create_and_add() combines the work of kobject_create() and kobject_add() into one function.

2) 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.

1. 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.
2. 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.
3. Remove new attribute: Removing a attribute need the function sysfs_remove_file().

E. Implementation

1) Makefile: We need to change the Makefile file of /arch/x86/kvm/Makefile for compiling the file.

```
obj?$(CONFIG VMCS SYSFS) += vmcs?sysfs.o
```

VI. How to modify the code

Vmx.c is in the directory /arch/x86/kvm. VMCS sets are in this file. In order to access the VMCS, we should add and modify code that is about how to read and write VMCS sysfs. Below is the function that is about how to reach the function which kernel read and write VMCS SYSFS.

We define several attributes to represent the files in vmcs directory. This structures of attribute have many different names and same modes. They storage in d_abt[]. After defining the attributes, we define Ktype. Its default_attrs is d_abt[]. When doing the kobject_init_and_add() function, we use this Ktype.

Here are some of the functions.

```
struct attribute rip_a = {
        .name = "g_rip_a",
        .mode = S_IRWXUGO,
};

struct attribute rsp_b = {
        .name = "g_rsp_b",
        .mode = S_IRWXUGO,
};

struct attribute cr0_c = {
        .name = "g_cr0_c",
        .mode = S_IRWXUGO,
};
static struct attribute * d_abt[] = {
        &rip_a,
        &rsp_b,
        &cro_c,
        &cr3_c,
        &cr4_c,
        &G_interuptibility_info,
        &G_rflags,
        NULL,
}
struct kobj_type kt = {
        .release = vmcs_sysfs_release,
        .sysfs_ops = &obj_sys_ops,
        .default_attrs = d_abt,
};
```

After defining the attribute and ktype, we need to define sysfs ops structure. It has two functions to operate the kobject. ".show" is using kobj_show() function, another is ".store" that is using kobj_store() function. obj_sys_ops function is use in Kt structure.
Here is the function:

```
static struct sysfs_ops obj_sys_ops = {
        .show = kobj_show,
        .store = kobj_store,
};
struct kobj_type kt = {
        .release = vmcs_sysfs_release,
        .sysfs_ops = &obj_sys_ops,
        .default_attrs = d_abt,
};
```

The function kobj_show and kobj_store tell provide the function that let the kernel read and write vmcs sysfs. The vmcs_readl() in kobj_show and vmcs_writel() in kobj_store are used for get the vmcs value from kernel and change the vmcs value. In the show two function, we have a switch that use for make sure which value in vmcs to read or write. This two function return the amount of the bit that the function read or write.

```
ssize_t kobj_show ( struct kobject * kobject , struct attribute * attr , char * buf )
{
        ssize_t count;
        unsigned long value;
        unsigned long field;
        field = get_field_addr(attr->name);
        printk("vmcs: read %ld\n", field);
        value = vmcs_readl(field);
        count = sprintf(buf, "%ld\n", value);
        return count;
}

ssize_t kobj_store( struct kobject * kobject , struct attribute * attr , const char *
{
unsigned long value;
unsigned long field;
field = get_field_addr(attr->name);
value = simple_strtoul(buf, NULL, 2);
printk("vmcs: write %ld\n", field);
```

```
vmcs_writel(field, value);
return count;
}
```

VII. Source Code

```
#include <linux/device.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/stat.h>
#include <linux/string.h>
#include <linux/sysfs.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Author");

struct kobject kobj;

extern unsigned long vmcs_readl(unsigned long field);
extern void vmcs_writel(unsigned long field, unsigned long value);
extern unsigned long get_field_addr(const char *field);

struct attribute rip_a = {
        .name = "g_rip_a",
        .mode = S_IRWXUGO,
};

struct attribute rsp_b = {
        .name = "g_rsp_b",
        .mode = S_IRWXUGO,
};

struct attribute cr0_c = {
        .name = "g_cr0_c",
        .mode = S_IRWXUGO,
};
```

```
struct attribute cr3_c = {
        .name = "g_cr3_c",
        .mode = S_IRWXUGO,
};

struct attribute cr4_c = {
        .name = "g_cr4_c",
        .mode = S_IRWXUGO,
};

struct attribute G_interuptibility_info = {
        .name = "guest_interuptibility_info",
        .mode = S_IRWXUGO,
};

struct attribute G_rflags = {
        .name = "guest_rflags",
        .mode = S_IRWXUGO,
};

static ssize_t kobj_show (struct kobject *kobject, struct attribute *attr, char *buf)
{
        ssize_t count;
        unsigned long value;
        unsigned long field;
        field = get_field_addr(attr->name);
        printk("vmcs: read %ld\n", field);
        value = vmcs_readl(field);
        count = sprintf(buf, "%ld\n", value);
        return count;
}

static ssize_t kobj_store(struct kobject *kobject, struct attribute *attr, const char
{
        unsigned long value;
        unsigned long field;
        field = get_field_addr(attr->name);
        value = simple_strtoul(buf, NULL, 2);
```

```
        printk("vmcs: write %ld\n", field);
        vmcs_writel(field, value);
        return count;
}

static void vmcs_sysfs_release(struct kobject *kobject)
{
        printk("vmcs: release\n");
}

static struct attribute * d_abt[] = {
        &rip_a,
        &rsp_b,
        &cro_c,
        &cr3_c,
        &cr4_c,
        &G_interuptibility_info,
        &G_rflags,
        NULL,
};

static struct sysfs_ops obj_sys_ops = {
        .show = kobj_show,
        .store = kobj_store,
};

struct kobj_type kt =
{
        .release = vmcs_sysfs_release,
        .sysfs_ops = &obj_sys_ops,
        .default_attrs = d_abt,
};

static int __init vmcs_sysfs_module_init(void)
{
        int ret;
        printk("vmcs: init\n");
        ret = kobject_init_and_add(&kobj, &ktype, NULL, "vmcs");
```

```
        return ret;
}

static void __exit vmcs_sysfs_module_exit(void)
{
        printk("vmcs: exit\n");
        kobject_del(&kobj);
}

module_init(vmcs_sysfs_module_init);
module_exit(vmcs_sysfs_module_exit);
```

VIII. Test

To test if we can successfully terminate the vmcs sysfs interface, we need to go to the / sys directory and find there is a vmcs directory. If there are the vmcs, we need a command to check if the system can read or write the value into the value.

A. Test Read

Use the command ”cat /sys/vmcs/g_rsp_b” to check whether there are values response. If there is value response, it means that we have to read the sysfs file.

B. Test Write

Use command ”echo 2 /sys/vmcs/g_rsp_b” and ”cat /sys/vmcs/g_rsp_b”. If the value appears 2, it means that we need to change the sysfs file.

REFERENCES

[1] Intel. (2011). Intel 64 and IA-32 Architectures Software Developer's Manual. Intel.

[2] Rachamalla, S. (2011). Kernel Virtual Machine.

[3] Aires, B. (2008). Hardware Assisted Virtualization Intel Virtualization Technology.

[4] Love, R. (2010). Linux Kernel Development