Implementing a Linux Sysfs Interface to the VMCS

By
Ian Kronquist

A THESIS

submitted to

Oregon State University

University Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science
(Honors Scholar)

Presented May 16, 2017
Commencement June 2017

AN ABSTRACT OF THE THESIS OF

Ian Kronquist for the degree of Honors Baccalaureate of Science in Computer Science presented on May 16, 2017. Title: Implementing a Linux Sysfs Interface to the VMCS

Abstract approved:

D. Kevin McGrath

The Virtual Machine Control Structure is an x86 hardware structure available on Intel platforms which support the VMX instruction set extensions. Developers working with Linux kernel virtualization technologies may need to alter fields in the VMCS, but the few existing tools for updating the structure are unstable or difficult to use. Fortunately the Kernel has a way to expose internal structures using SysFS, a virtual file system which exposes a common interface for Kernel implementation details, internal structures, and layout. To expose VMCS fields to SysFS we wrote a patch to the Linux kernel. We unsuccessfully attempted to get this patch merged upstream into the Linux Kernel.

Key Words: Linux, Virtual Machine, KVM

Corresponding e-mail address: kronquii@oregonstate.edu

Implementing a Linux Sysfs Interface to the VMCS

By
Ian Kronquist

A THESIS

submitted to

Oregon State University

University Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science
(Honors Scholar)

Presented May 16, 2017
Commencement June 2017

Honors Baccalaureate of Science in Computer Science project of Ian Kronquist presented on May 16, 2017

APPROVED:

D. Kevin McGrath, Mentor, representing Department of Electrical Engineering and Computer Science

---

Lance Albertson, Committee Member, representing Oregon State University Open Source Lab

---

Dr. Bobba Rakesh, Committee Member, representing Department of Electrical Engineering and Computer Science

---

Toni Doolen, Dean, Oregon State University Honors College

I understand that my project will become part of the permanent collection of Oregon State University Honors College. My signature below authorizes release of my project to any reader upon request.

Ian Kronquist, Author

Contents

1 Introduction

The goal of this project is to create a Linux kernel patch which will expose the Virtual Machine Control Structure used by Intel's VT-x virtual machine instruction set extensions to the Linux kernel SysFS Virtual File System. This feature will help researchers at Oregon State, and potentially researchers at Intel who are working on Intel virtualization related technologies in the Linux kernel to better control the behavior of virtual machines. The module would be useful in many situations, such as injecting a fault into a running VM during testing, and inspecting the state of guest CPU features.

Similar patches have been submitted to the Linux Kernel Mailing List before. They have been thoroughly reviewed, sometimes through multiple rounds of examination, but all have been rejected. As part of preparation for this project we examined a series of these patch sets.

This thesis is organized as follows: the first section is devoted to providing a background in the relevant technologies, such as the VMCS, the SysFS virtual file system, and Linux kernel modules. In the second section we will enumerate the design goals and requirements for the project. Next we will explain the current organization of the data exposed via SysFS. In the following section we will describe the current implementation, and then move on to contrast it with alternative implementation approaches. Finally, we will provide a high level summary of the different approaches and how they accomplish the design goals and conclude the thesis.

2 Background

2.1 The Virtual Machine Control Structure

In 2006, Intel released a series of extensions to its x86 and x86_64 instruction set architectures to enable hardware accelerated virtual machines. Intel introduced a special data structure used by the hardware to store the processor state when starting and running a VM called the Virtual Machine Control Structure, or VMCS. The VMCS may use up to a page worth of memory and currently has over 172 fields describing various parts of the host and guest OS state. Each processor has a pointer to the currently loaded VMCS.

Before initializing a VMCS structure, its contents should be erased with the `VMCLEAR` instruction. Afterwards, the manual mandates that fields should only be updated using the `VMREAD` and `VMWRITE` instructions. Interestingly, the current generation of Intel processors allows you to read and write fields from the structure with the `MOV` instruction [1]. This feature was used during early development of the patch.

Once the VMCS is prepared it can be loaded onto the current CPU with the `VMPTRLD` instruction, and the VM can be started with the `VMLAUNCH` instruction and resumed with the `VMRESUME` instruction. If the guest VM attempts to issue a privileged

instruction, such as `HLT` or `VMLAUNCH`, it will cause the VM to exit and control will transfer to the host. The host can then inspect the VMCS to learn what the VM was trying to do when it exited and choose to emulate the privileged instruction inside the VM's memory space or to let the VM fail and clean up the VMCS. For instance, if a program in the VM attempts to learn what features are supported by the CPU it's running on with the `CPUID` instruction, the VM will exit, and the host can choose to update the VM to indicate what features the host would like the guest CPU to support.

Some of the fields in the VMCS represent important guest state such as the current values in the VM's registers, or the pointer to the Interrupt Descriptor Table. However, on 32-bit CPUs the registers are shorter than on 64-bit CPUs, so the fields in the VMCS may have different sizes on different architecture variants. Also, future revisions of the VMCS may change the order of the fields or even add entirely new fields [2].

2.2 The SysFS Virtual Filesystem

SysFS is the name for all of the files under `/sys/`. It is a virtual file system, which means files on that file system do not exist on the disk or any other physical media – rather they are representations of configurable kernel variables and the state of certain kernel data structures. For instance, on most systems the file `/sys/block/sda/queue/scheduler` has a list of available I/O schedulers for the system's hard drive. The current I/O scheduler can be changed by writing the name of one of the schedulers to the file.

Listing 1: 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 example of 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.

Listing 2: 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
```

Users cannot 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. `kobject`s 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 attribute's 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(2)` 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(2)` 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 `kobject`s, `kset`s are represented as directories and have parents, however they do not have attributes [4].

2.3 Linux Kernel Modules

In order to add new drivers or functionality to the kernel, developers can write kernel modules. Modules can be used to create a driver for specialized hardware and expose the device to the `/dev` file system, or expose kernel settings and data structures to the `/sys` file system. Kernel modules must have an initialization function and a exit function, and should have meta data such as the module author and license. Kernel module symbols which can be used by other modules should be explicitly decorated with the macro `EXPORT_SYMBOL`.

Kernel modules can be loaded when the Linux starts or even after it is already running. Kernel modules can also be baked into the kernel at compilation time [3].

3 Design Goals

1. The kernel patch should be stable under a variety of conditions. If it is a removable module, it should be able to be inserted and removed cleanly.

2. The patch should never cause a kernel panic.

3. The patched kernel should display all of the VMCS structures controlled by the kernel at any point in time. If new structures are added or removed the proper files in the SysFS directory should be updated. When a new VM is started and new VMCSs are created they should be displayed in SysFS. In some cases if the VM uses symmetric multiprocessing there will be multiple VMCSs assigned to the VM. Each of the VMCSs should show up under the SysFS directory individually.

4. The patch should work on both 32-bit and 64-bit machines.

5. The patch should work on the latest version of Linux. We are currently working off of the tip of Linus' tree, however the patch seems to apply cleanly to versions of the kernel as old as 3.14.

6. The patched kernel should work on CentOS on the Intel Minnowboard.

7. The patch should require minimal changes to the current VMX implementation.

8. The patch must be GPLv2 compliant so it can be merged into the kernel. We have chosen to dual license it under MIT and GPLv2.

4 The VMCSCTL SysFS Interface

The VMCSCTL module creates a `kset` known as `vmcsctl`. This `kset` uses the `/sys/kernel` `kobject` as its parent. Under `vmcsctl` are kobjects representing VMCSs. Each one is named `vmcsN`, where "N" is the PID of the process which created the VMCS. Each VMCS has 172 attributes, named after the symbols in the `vmx.h` source code. These represent fields in the VMCS, as specified in Appendix B.18 of the Intel manual.

Listing 3: Example of VMCSCTL SysFS Interface Usage

```
# ls /sys/kernel/vmcsctl/

vmcs1555  vmcs1556

# ls /sys/kernel/vmcsctl/vmcs1555/

APIC_ACCESS_ADDR                    GUEST_SYSENTER_CS
APIC_ACCESS_ADDR_HIGH               GUEST_SYSENTER_EIP
CPU_BASED_VM_EXEC_CONTROL           GUEST_SYSENTER_ESP
CR0_GUEST_HOST_MASK                 GUEST_TR_AR_BYTES
CR0_READ_SHADOW                     GUEST_TR_BASE
CR3_TARGET_COUNT                    GUEST_TR_LIMIT
CR3_TARGET_VALUE0                   GUEST_TR_SELECTOR
CR3_TARGET_VALUE1                   HOST_CR0
```

```
CR3_TARGET_VALUE2
CR3_TARGET_VALUE3
CR4_GUEST_HOST_MASK
CR4_READ_SHADOW
EOI_EXIT_BITMAP0
EOI_EXIT_BITMAP0_HIGH
EOI_EXIT_BITMAP1
EOI_EXIT_BITMAP1_HIGH
EOI_EXIT_BITMAP2
EOI_EXIT_BITMAP2_HIGH
EOI_EXIT_BITMAP3
EOI_EXIT_BITMAP3_HIGH
EPT_POINTER
EPT_POINTER_HIGH
EXCEPTION_BITMAP
EXIT_QUALIFICATION
GUEST_ACTIVITY_STATE
GUEST_BNDCFGS
GUEST_BNDCFGS_HIGH
GUEST_CR0
GUEST_CR3
GUEST_CR4
GUEST_CS_AR_BYTES
GUEST_CS_BASE
GUEST_CS_LIMIT
GUEST_CS_SELECTOR
GUEST_DR7
GUEST_DS_AR_BYTES
GUEST_DS_BASE
GUEST_DS_LIMIT
GUEST_DS_SELECTOR
GUEST_ES_AR_BYTES
GUEST_ES_BASE
GUEST_ES_LIMIT
GUEST_ES_SELECTOR
GUEST_FS_AR_BYTES
GUEST_FS_BASE
GUEST_FS_LIMIT
GUEST_FS_SELECTOR
GUEST_GDTR_BASE
GUEST_GDTR_LIMIT
GUEST_GS_AR_BYTES
GUEST_GS_BASE
GUEST_GS_LIMIT
GUEST_GS_SELECTOR
GUEST_IA32_DEBUGCTL
GUEST_IA32_DEBUGCTL_HIGH
GUEST_IA32_EFER
GUEST_IA32_EFER_HIGH
GUEST_IA32_PAT
GUEST_IA32_PAT_HIGH
HOST_CR3
HOST_CR4
HOST_CS_SELECTOR
HOST_DS_SELECTOR
HOST_ES_SELECTOR
HOST_FS_BASE
HOST_FS_SELECTOR
HOST_GDTR_BASE
HOST_GS_BASE
HOST_GS_SELECTOR
HOST_IA32_EFER
HOST_IA32_EFER_HIGH
HOST_IA32_PAT
HOST_IA32_PAT_HIGH
HOST_IA32_PERF_GLOBAL_CTRL
HOST_IA32_PERF_GLOBAL_CTRL_HIGH
HOST_IA32_SYSENTER_CS
HOST_IA32_SYSENTER_EIP
HOST_IA32_SYSENTER_ESP
HOST_IDTR_BASE
HOST_RIP
HOST_RSP
HOST_SS_SELECTOR
HOST_TR_BASE
HOST_TR_SELECTOR
IDT_VECTORING_ERROR_CODE
IDT_VECTORING_INFO_FIELD
IO_BITMAP_A
IO_BITMAP_A_HIGH
IO_BITMAP_B
IO_BITMAP_B_HIGH
MSR_BITMAP
MSR_BITMAP_HIGH
PAGE_FAULT_ERROR_CODE_MASK
PAGE_FAULT_ERROR_CODE_MATCH
PIN_BASED_VM_EXEC_CONTROL
PLE_GAP
PLE_WINDOW
PML_ADDRESS
PML_ADDRESS_HIGH
POSTED_INTR_DESC_ADDR
POSTED_INTR_DESC_ADDR_HIGH
POSTED_INTR_NV
SECONDARY_VM_EXEC_CONTROL
TPR_THRESHOLD
TSC_MULTIPLIER
TSC_MULTIPLIER_HIGH
TSC_OFFSET
TSC_OFFSET_HIGH
VIRTUAL_APIC_PAGE_ADDR
VIRTUAL_APIC_PAGE_ADDR_HIGH
```

```
GUEST_IA32_PERF_GLOBAL_CTRL        VIRTUAL_PROCESSOR_ID
GUEST_IA32_PERF_GLOBAL_CTRL_HIGH   VMCS_LINK_POINTER
GUEST_IDTR_BASE                    VMCS_LINK_POINTER_HIGH
GUEST_IDTR_LIMIT                   VMREAD_BITMAP
GUEST_INTERRUPTIBILITY_INFO        VMWRITE_BITMAP
GUEST_INTR_STATUS                  VMX_INSTRUCTION_INFO
GUEST_LDTR_AR_BYTES                VMX_PREEMPTION_TIMER_VALUE
GUEST_LDTR_BASE                    VM_ENTRY_CONTROLS
GUEST_LDTR_LIMIT                   VM_ENTRY_EXCEPTION_ERROR_CODE
GUEST_LDTR_SELECTOR                VM_ENTRY_INSTRUCTION_LEN
GUEST_LINEAR_ADDRESS               VM_ENTRY_INTR_INFO_FIELD
GUEST_PDPTR0                       VM_ENTRY_MSR_LOAD_ADDR
GUEST_PDPTR0_HIGH                  VM_ENTRY_MSR_LOAD_ADDR_HIGH
GUEST_PDPTR1                       VM_ENTRY_MSR_LOAD_COUNT
GUEST_PDPTR1_HIGH                  VM_EXIT_CONTROLS
GUEST_PDPTR2                       VM_EXIT_INSTRUCTION_LEN
GUEST_PDPTR2_HIGH                  VM_EXIT_INTR_ERROR_CODE
GUEST_PDPTR3                       VM_EXIT_INTR_INFO
GUEST_PDPTR3_HIGH                  VM_EXIT_MSR_LOAD_ADDR
GUEST_PENDING_DBG_EXCEPTIONS       VM_EXIT_MSR_LOAD_ADDR_HIGH
GUEST_PHYSICAL_ADDRESS             VM_EXIT_MSR_LOAD_COUNT
GUEST_PHYSICAL_ADDRESS_HIGH        VM_EXIT_MSR_STORE_ADDR
GUEST_PML_INDEX                    VM_EXIT_MSR_STORE_ADDR_HIGH
GUEST_RFLAGS                       VM_EXIT_MSR_STORE_COUNT
GUEST_RIP                          VM_EXIT_REASON
GUEST_RSP                          VM_INSTRUCTION_ERROR
GUEST_SS_AR_BYTES                  XSS_EXIT_BITMAP
GUEST_SS_BASE                      XSS_EXIT_BITMAP_HIGH
GUEST_SS_LIMIT                     abort
GUEST_SS_SELECTOR                  revision_id

# cat /sys/kernel/vmcsctl/vmcs1555/revision\_id

300252880
```

5 The Patch Implementation

In order to expose a data structure to SysFS, the data structure must have an associated `kobject`. Since the VMCS does not have its own `kobject` field we must create a special structure which has a `kobject` and a pointer to the VMCS. We call this object the `vmcsctl` structure. This structure also has a TID field which represents the ID of the thread which owns the associated VMCS. A new `vmcsctl` structure must be created for each new VMCS, and it must not outlive the VMCS after the VMCS has been freed. The obvious way to satisfy these requirements is to have the kvm_intel kernel module, which controls the VMCS, pass a pointer to the VMCS to our kernel module every time a new VMCS is created or destroyed. This means that our module must be loaded before the first VMCS is created. Due to the high degree

of coupling with the kvm_intel kernel module we declare our module to be dependent on it, and also declared that our module could not be runtime loadable.

Listing 4: `vmcsctl` Structure

```
struct vmcsctl {
    int pid;
    struct kobject kobj;
    struct vmcs *vmcs;
};
```

In order to manipulate the fields of the VMCS we must represent each field as a file in the VMCS's associated SysFS directory. The recommended way to create a large number of attributes is with an `attribute_group` structure, which has a NULL terminated vector of individual `kobject` attributes. Each attribute requires its own show and store functions which are called when the file is written to and read from respectively. Since there are over 172 relevant fields in the current VMCS revision we created a series of macros to build these show and store functions, as well as the `kobject` attribute. Additionally, different fields in the VMCS have different widths. There are 16-bit, 32-bit, and 64-bit fields. The macro takes the field name and its width as an argument and builds the associated show and store functions and the `kobject` attribute. Certain fields are read only and are declared with restricted permissions as one of the macro's arguments.

To complicate matters, there are certain fields in the VMCS which have a "natural width", that is 64 bits on x86_64 and only 32 bits on x86. The kvm_intel module handles this with a conditional new type definition. This approach would work for our module as well, however, we would need to write a `natural_width` family of functions to read from and write to natural width data. If we instead define these as macros, they will be automatically expanded into the right data type allowing us to reuse the 32 and 64 bit store and show functions.

Listing 5: `vmcsctl` Structure

```
#ifdef x86_64
#define natural_width u64
#else
#define natural_width u32
#endif

#define VMCS_ATTR_PERMS(attr_field, type, perms) \
static ssize_t vmcs_##attr_field##_show(struct kobject *kobj, \
                struct kobj_attribute *attr, char *buf) \
{ \
        return vmcs_field_show_##type(kobj, attr, buf, attr_field)
           ; \
}\
static ssize_t vmcs_##attr_field##_store(struct kobject *kobj, \
```

```
                struct kobj_attribute *attr, const char *buf,
                   size_t count) \
{ \
        return vmcs_field_store_##type(kobj, attr, buf, count,
           attr_field); \
}\
static struct kobj_attribute vmcs_field_##attr_field = \
        __ATTR(attr_field, perms, vmcs_##attr_field##_show, \
                vmcs_##attr_field##_store);

#define VMCS_ATTR(attr_field, type) VMCS_ATTR_PERMS(attr_field,
   type, 0644)
#define VMCS_ATTR_RO(attr_field, type) VMCS_ATTR_PERMS(attr_field,
    type, 0444)
```

The Intel manual requires reads and writes to the VMCS structure to happen with the `VMREAD` and `VMWRITE` assembly instructions. In order to use these instructions we chose to move several static inline functions from `vmx.c` to the `vmx.h` header file. Notably, the `VMREAD` and `VMWRITE` instructions do not take a target VMCS. They affect the current VMCS on the core they are issued on. In order to read from or write to the right VMCS we need to save the current VMCS pointer, load the VMCS pointer of the VMCS we are interested in, perform the read or write, and restore the original VMCS pointer.

One of the issues with these instructions is that they can only be used when the processor has VMX mode enabled. If the processor is not currently in VMX mode these instructions will cause a General Protection Fault, and possibly a kernel panic. To prevent the instructions from being run when not in VMX mode, we track when `kvm_cpu_vmxon` and `kvm_cpu_vmxoff` functions are called. If we wind up in a state where the VMCSs have not been destroyed but the processor is not in VMX mode we will cause the read or write to fail with an I/O error.

One advantage of the current approach is that it requires minimal changes to the kvm_intel module. The definition of the `vmcs` structure must be moved to the associated header file, and simple hooks are added to the `alloc_vmcs_cpu`, `free_vmcs` `kvm_cpu_vmxon` and `kvm_cpu_vmxoff` functions. No other structures or functions are changed or moved to the header file.

The biggest disadvantage of this approach is the tight coupling to the kvm_intel kernel module. The nature of this problem requires us to track the creation and destruction of the VMCS as well as whether the processor is in VMX mode. After experimenting with several approaches we concluded that direct function calls from the kvm_intel module into the vmcsctl module was the best way to learn this information in spite of the tight coupling it introduces.

6 Alternative Implementation Approaches

One way to improve the current approach would be to keep new VMCS structures in a queue. When they are removed they would be added to a deletion queue. Unfortunately, if a VMCS is allocated, freed, and another VMCS is allocated in the same location, they might show up as being added and deleted multiple times. The queues might also become very large. What's more, we would have to store the pid of the process which owns the VMCS along with the structure. Additionally, the kernel likely will not accept such major changes which have an incredibly specific use case. We attempted an implementation of this strategy and decided that the drawbacks of this approach outweighed the benefits.

Another approach would be to register a callback which is called every time a VMCS is allocated or freed. Unfortunately, VMCSs which exist before the callback is registered will not be exposed to SysFS using this approach alone. This would require adding a series of new functions to manage the callback to the kvm_intel module interface, which may face some resistance. However, if this approach is properly engineered it may be useful for other module authors in the future. This approach could be used in conjunction with an approach which will display the VMCS structures running on the CPU at load time to display all of the VMCSs. This holds more promise than the queue approach, but was discarded because it is somewhat complex and adding a mechanism for registering callbacks in this code seems niche.

We have devised an approach to examine the list of available VMCS at module load time, but it is fairly invasive. Each VMCS is associated with a `vcpu_vmx` structure, which is private to the `vmx.c` file. Embedded in this structure is a `kvm_vcpu` structure. It is relatively easy to get a hold of all of the `kvm_vcpu` structures by inspecting the `kvm` structure, and fortunately the definitions of both of them are public. If we were to expose the `vcpu_vmx` function and the `to_vmx`, which is a trivial wrapper around `container_of` for retrieving a `vcpu_vmx` structure from a pointer to its embedded `kvm_vcpu`, we could list all of the running VMCSs. Unfortunately, this requires moving a fairly large structure to a header file, and thus a public API, which may meet some push back. Additionally, this will only allow us to probe for running VMCSs. It will not allow us to detect newly started VMCSs after the module is loaded. However, it could be used in conjunction with the callback approach described earlier.

Another approach would be to ditch the separate module design altogether and move the code into `vmx.c`. The SysFS code could be surrounded by `#ifdef` guards controlled by a kernel configuration parameter. One criticism of this approach is that it violates separation of concerns. We would be mixing UI code which exposes the VMCS to SysFS with code concerned only with the core business logic which manipulates VMCS structures and controls virtual CPUs. Additionally, it would be a fairly large change to the `vmx.c` file and would likely face considerable scrutiny.

7 Submission and Review

On April 27^{th} 2017 we submitted the first version of this patch to the Linux kernel KVM mailing list. Radim Krčmář pointed out that this first version did not load the proper VMCS before issuing the `VMREAD` or `VMWRITE` instructions, which would cause the processor to possibly read from or write to the wrong VMCS. He also expressed concern with the purpose of the patch and suggested that the current approach of implementing KVM IOCtls for manipulating VMCS fields was superior. Paolo Bonzini suggested that we should not allow write access to the VMCS, likely because an errant write could easily crash the guest or the host [5].

On May 9^{th} an updated version of the patch was submitted which addressed Mr. Krčmář's technical comments. Ultimately, Bonzini suggested that the current custom IOCtl approach already fulfilled this sort of functionality [6].

8 Conclusion

In the end, we chose the current design because it managed to maintain separation of UI and functionality concerns with minimal changes to the core VMX code. We believed that more substantial changes were more likely to be rejected by the kernel community and it is best to keep the changes short and simple. However, after careful review by the core KVM developers our changes were rejected, as they were unwilling to allow raw write access to the VMCS and could already manipulate the VMCS with custom IOCtls when it is truly necessary.

References

[1] Muli Ben-Yehuda et al. "The Turtles Project: Design and Implementation of Nested Virtualization." In: *OSDI*. Vol. 10. 2010, pp. 423–436.

[2] *Intel 64 and IA-32 Architectures Software Developer's Manual 2014*. Intel Corporation, 2014.

[3] Robert Love. *Linux Kernel Development*. 3rd ed. Novell Press, 2010.

[4] Patrick Mochel. "The sysfs Filesystem". In: *Linux Symposium*. 2005, pp. 313–326.

[5] Ian Kronquist Paolo Bonzini Radim Krčmář. *[PATCH] KVM: Expose Virtual Machine Control Structure to SYSFS*. kvm@vger.kernel.org public mailing list discussion. Apr. 27, 2017. URL: `https://patchwork.kernel.org/patch/9702245/`.

[6] Ian Kronquist Paolo Bonzini Radim Krčmář. *[PATCH v2] KVM: Expose Virtual Machine Control Structure to SYSFS*. kvm@vger.kernel.org public mailing list discussion. May 9, 2017. URL: `https://patchwork.kernel.org/patch/9719097/`.

Appendices

A Patch to Linux Kernel

Listing 6: `vmcsctl` Structure

```
From 84d0b67840ecc5fd5e36f0b774240002bf15501f Mon Sep 17
   00:00:00 2001
From: Ian Kronquist <iankronquist@gmail.com>
Date: Thu, 6 Apr 2017 20:46:36 -0400
Subject: [PATCH] KVM: Expose Virtual Machine Control
   Structure to SYSFS

Users doing research or development on KVM and the Intel VMX
    virtualization
extensions may desire fine grained control of Virtual
   Machine host and
guest state by reading or writing fields in the Virtual
   Machine Control
Structure (VMCS).
```

```
In order to allow users to manipulate VMCSs, we expose
   individual VMCSs via
SYSFS. VMCS fields are under the directory `/sys/kernel/
   vmcsctl/vmcsN`
where N is the PID of the process which controls the VMCS.
   Each attribute
under such a directory is named after the corresponding
   symbol in `vmx.h`,
which closely follow the field names in the Intel manual.

By default, this feature is disabled. It can be enabled via
   the KVM_VMCSCTL
Kconfig option.

Version 2:
* Load and restore the VMCS pointer when reading and writing
   .
* Use kstrtouint instead of kstrtoint.

Signed-off-by: Ian Kronquist <iankronquist@gmail.com>
---
 arch/x86/include/asm/vmx.h | 149 ++++++++++
 arch/x86/kvm/Kconfig       |   9 +
 arch/x86/kvm/Makefile      |   1 +
 arch/x86/kvm/vmcsctl.c     | 649
    ++++++++++++++++++++++++++++++++++++++++++
 arch/x86/kvm/vmcsctl.h     |  27 ++
 arch/x86/kvm/vmx.c         | 167 +++++----------
 6 files changed, 871 insertions(+), 131 deletions(-)
 create mode 100644 arch/x86/kvm/vmcsctl.c
 create mode 100644 arch/x86/kvm/vmcsctl.h

diff --git a/arch/x86/include/asm/vmx.h b/arch/x86/include/
   asm/vmx.h
index cc54b7026567..fe8708d61d74 100644
--- a/arch/x86/include/asm/vmx.h
+++ b/arch/x86/include/asm/vmx.h
@@ -25,10 +25,15 @@
 #define VMX_H


+#include <linux/kvm_host.h>
 #include <linux/bitops.h>
 #include <linux/types.h>
 #include <uapi/asm/vmx.h>
```

```
+#define __ex(x) __kvm_handle_fault_on_reboot(x)
+#define __ex_clear(x, reg) \
+       ____kvm_handle_fault_on_reboot(x, "xor " reg " , "
   reg)
+
 /*
  * Definitions of Primary Processor-Based VM-Execution
    Controls.
  */
@@ -112,6 +117,8 @@
 #define VMX_MISC_SAVE_EFER_LMA                  0x00000020
 #define VMX_MISC_ACTIVITY_HLT                   0x00000040

+noinline void vmwrite_error(unsigned long field, unsigned
   long value);
+
 static inline u32 vmx_basic_vmcs_revision_id(u64 vmx_basic)
 {
        return vmx_basic & GENMASK_ULL(30, 0);
@@ -485,6 +492,7 @@ enum vmcs_field {
 #define ASM_VMX_VMLAUNCH           ".byte 0x0f, 0x01, 0xc2"
 #define ASM_VMX_VMRESUME           ".byte 0x0f, 0x01, 0xc3"
 #define ASM_VMX_VMPTRLD_RAX        ".byte 0x0f, 0xc7, 0x30"
+#define ASM_VMX_VMPTRST_RAX        ".byte 0x0f, 0xc7, 0x38"
 #define ASM_VMX_VMREAD_RDX_RAX     ".byte 0x0f, 0x78, 0xd0"
 #define ASM_VMX_VMWRITE_RAX_RDX    ".byte 0x0f, 0x79, 0xd0"
 #define ASM_VMX_VMWRITE_RSP_RDX    ".byte 0x0f, 0x79, 0xd4"
@@ -499,6 +507,12 @@ struct vmx_msr_entry {
        u64 value;
 } __aligned(16);

+struct vmcs {
+       u32 revision_id;
+       u32 abort;
+       char data[0];
+};
+
 /*
  * Exit Qualifications for entry failure during or after
    loading guest state
  */
@@ -554,4 +568,139 @@ enum vm_instruction_error_number {
        VMXERR_INVALID_OPERAND_TO_INVEPT_INVVPID = 28,
 };
```

```
+static __always_inline void vmcs_check16(unsigned long
   field)
+{
+       BUILD_BUG_ON_MSG(__builtin_constant_p(field) &&
+                        ((field) & 0x6001) == 0x2000,
+                        "16-bit accessor invalid for 64-bit
   field");
+       BUILD_BUG_ON_MSG(__builtin_constant_p(field) &&
+                        ((field) & 0x6001) == 0x2001,
+                        "16-bit accessor invalid for 64-bit
   high field");
+       BUILD_BUG_ON_MSG(__builtin_constant_p(field) &&
+                        ((field) & 0x6000) == 0x4000,
+                        "16-bit accessor invalid for 32-bit
   high field");
+       BUILD_BUG_ON_MSG(__builtin_constant_p(field) &&
+                        ((field) & 0x6000) == 0x6000,
+                        "16-bit accessor invalid for natural
    width field");
+}
+
+static __always_inline void vmcs_check32(unsigned long
   field)
+{
+       BUILD_BUG_ON_MSG(__builtin_constant_p(field) &&
+                        ((field) & 0x6000) == 0,
+                        "32-bit accessor invalid for 16-bit
   field");
+       BUILD_BUG_ON_MSG(__builtin_constant_p(field) &&
+                        ((field) & 0x6000) == 0x6000,
+                        "32-bit accessor invalid for natural
    width field");
+}
+
+static __always_inline void vmcs_check64(unsigned long
   field)
+{
+       BUILD_BUG_ON_MSG(__builtin_constant_p(field) && ((
   field) & 0x6000) == 0,
+                        "64-bit accessor invalid for 16-bit
   field");
+       BUILD_BUG_ON_MSG(__builtin_constant_p(field) &&
+                        ((field) & 0x6001) == 0x2001,
```

```
+                       "64-bit accessor invalid for 64-bit
   high field");
+       BUILD_BUG_ON_MSG(__builtin_constant_p(field) &&
+                       ((field) & 0x6000) == 0x4000,
+                       "64-bit accessor invalid for 32-bit
   field");
+       BUILD_BUG_ON_MSG(__builtin_constant_p(field) &&
+                       ((field) & 0x6000) == 0x6000,
+                       "64-bit accessor invalid for natural
    width field");
+}
+
+static __always_inline void vmcs_checkl(unsigned long field
   )
+{
+       BUILD_BUG_ON_MSG(__builtin_constant_p(field) && ((
   field) & 0x6000) == 0,
+                        "Natural width accessor invalid for
    16-bit field");
+       BUILD_BUG_ON_MSG(__builtin_constant_p(field) &&
+                       ((field) & 0x6001) == 0x2000,
+                       "Natural width accessor invalid for
   64-bit field");
+       BUILD_BUG_ON_MSG(__builtin_constant_p(field) &&
+                       ((field) & 0x6001) == 0x2001,
+                       "Natural width accessor invalid for
   64-bit high field");
+       BUILD_BUG_ON_MSG(__builtin_constant_p(field) &&
+                       ((field) & 0x6000) == 0x4000,
+                       "Natural width accessor invalid for
   32-bit field");
+}
+
+static __always_inline unsigned long __vmcs_readl(unsigned
   long field)
+{
+       unsigned long value;
+
+       asm volatile (__ex_clear(ASM_VMX_VMREAD_RDX_RAX,
   "%0")
+                     : "=a"(value) : "d"(field) : "cc");
+       return value;
+}
+
+static __always_inline u16 vmcs_read16(unsigned long field)
```

```
+{
+       vmcs_check16(field);
+       return __vmcs_readl(field);
+}
+
+static __always_inline u32 vmcs_read32(unsigned long field)
+{
+       vmcs_check32(field);
+       return __vmcs_readl(field);
+}
+
+static __always_inline u64 vmcs_read64(unsigned long field)
+{
+       vmcs_check64(field);
+#ifdef CONFIG_X86_64
+       return __vmcs_readl(field);
+#else
+       return __vmcs_readl(field) | ((u64)__vmcs_readl(
   field+1) << 32);
+#endif
+}
+
+static __always_inline unsigned long vmcs_readl(unsigned
   long field)
+{
+       vmcs_checkl(field);
+       return __vmcs_readl(field);
+}
+
+static __always_inline void __vmcs_writel(unsigned long
   field,
+                unsigned long value)
+{
+       u8 error;
+
+       asm volatile (__ex(ASM_VMX_VMWRITE_RAX_RDX) "; setna
    %0"
+                     : "=q"(error) : "a"(value), "d"(field
   ) : "cc");
+       if (unlikely(error))
+                vmwrite_error(field, value);
+}
+
+static __always_inline void vmcs_writel(unsigned long field
   ,
```

```
+                unsigned long value)
+{
+       vmcs_checkl(field);
+       __vmcs_writel(field, value);
+}
+
+static __always_inline void vmcs_write16(unsigned long
   field, u16 value)
+{
+       vmcs_check16(field);
+       __vmcs_writel(field, value);
+}
+
+static __always_inline void vmcs_write32(unsigned long
   field, u32 value)
+{
+       vmcs_check32(field);
+       __vmcs_writel(field, value);
+}
+
+static __always_inline void vmcs_write64(unsigned long
   field, u64 value)
+{
+       vmcs_check64(field);
+       __vmcs_writel(field, value);
+#ifndef CONFIG_X86_64
+       asm volatile ("");
+       __vmcs_writel(field+1, value >> 32);
+#endif
+}
+
+struct vmcs *vmcs_store(void);
+void vmcs_load(struct vmcs *vmcs);
 #endif
diff --git a/arch/x86/kvm/Kconfig b/arch/x86/kvm/Kconfig
index ab8e32f7b9a8..30137fe8ae13 100644
--- a/arch/x86/kvm/Kconfig
+++ b/arch/x86/kvm/Kconfig
@@ -69,6 +69,15 @@ config KVM_INTEL
          To compile this as a module, choose M here: the
             module
          will be called kvm-intel.

+config KVM_VMCSCTL
+       bool "Expose the VMCS to sysfs"
```

```
+       depends on KVM
+       depends on KVM_INTEL
+       default n
+       ---help---
+         Adds entries to sysfs which allow fine grained
   virtual machine state to
+         be manipulated from userland.
+
 config KVM_AMD
        tristate "KVM for AMD processors support"
        depends on KVM
diff --git a/arch/x86/kvm/Makefile b/arch/x86/kvm/Makefile
index 3bff20710471..4d1c5a321a8e 100644
--- a/arch/x86/kvm/Makefile
+++ b/arch/x86/kvm/Makefile
@@ -22,4 +22,5 @@ kvm-amd-y             += svm.o pmu_amd.o

 obj-$(CONFIG_KVM)      += kvm.o
 obj-$(CONFIG_KVM_INTEL)        += kvm-intel.o
+obj-$(CONFIG_KVM_VMCSCTL)      += vmcsctl.o
 obj-$(CONFIG_KVM_AMD)  += kvm-amd.o
diff --git a/arch/x86/kvm/vmcsctl.c b/arch/x86/kvm/vmcsctl.c
new file mode 100644
index 000000000000..6d5db494003b
--- /dev/null
+++ b/arch/x86/kvm/vmcsctl.c
@@ -0,0 +1,649 @@
+#include "vmcsctl.h"
+
+static struct kset *vmcsctl_set;
+static bool vmxon;
+
+static inline struct vmcsctl *vmcsctl_container_of(struct
   kobject *kobj)
+{
+       return container_of(kobj, struct vmcsctl, kobj);
+}
+
+static void vmcsctl_release(struct kobject *kobj)
+{
+       struct vmcsctl *vmcsctl = vmcsctl_container_of(kobj)
   ;
+
+       kfree(vmcsctl);
+}
```

```
+
+static struct kobj_type vmcsctl_kobj_ktype = {
+       .release        = vmcsctl_release,
+       .sysfs_ops      = &kobj_sysfs_ops,
+};
+
+static ssize_t revision_id_show(struct kobject *kobj,
+       struct kobj_attribute *attr, char *buf)
+{
+       struct vmcsctl *vmcsctl = vmcsctl_container_of(kobj)
   ;
+
+       WARN_ON(vmcsctl->vmcs == NULL);
+       return sprintf(buf, "%d\n", vmcsctl->vmcs->
   revision_id);
+}
+
+static ssize_t abort_show(struct kobject *kobj,
+       struct kobj_attribute *attr, char *buf)
+{
+       struct vmcsctl *vmcsctl = vmcsctl_container_of(kobj)
   ;
+
+       WARN_ON(vmcsctl->vmcs == NULL);
+       return sprintf(buf, "%d\n", vmcsctl->vmcs->abort);
+}
+
+static ssize_t vmcs_field_show_u16(struct kobject *kobj,
+       struct kobj_attribute *attr, char *buf, enum
   vmcs_field field)
+{
+       u16 value;
+       struct vmcsctl *vmcsctl = vmcsctl_container_of(kobj)
   ;
+       struct vmcs *original_vmcs = vmcs_store();
+
+       WARN_ON(vmcsctl->vmcs == NULL);
+       vmcs_load(vmcsctl->vmcs);
+
+       value = vmcs_read16(field);
+       vmcs_load(original_vmcs);
+       return sprintf(buf, "%hu\n", value);
+}
+
+static ssize_t vmcs_field_store_u16(struct kobject *kobj,
```

```
+       struct kobj_attribute *attr, const char *buf, size_t
    count,
+       enum vmcs_field field)
+{
+       int ret;
+       u16 value;
+       struct vmcsctl *vmcsctl = vmcsctl_container_of(kobj)
   ;
+       struct vmcs *original_vmcs = vmcs_store();
+
+       WARN_ON(vmcsctl->vmcs == NULL);
+       vmcs_load(vmcsctl->vmcs);
+
+       ret = kstrtou16(buf, 10, &value);
+       if (ret < 0)
+               return ret;
+
+       vmcs_write16(field, value);
+       vmcs_load(original_vmcs);
+       return count;
+}
+
+static ssize_t vmcs_field_show_u32(struct kobject *kobj,
+       struct kobj_attribute *attr, char *buf, enum
   vmcs_field field)
+{
+       u32 value;
+       struct vmcsctl *vmcsctl = vmcsctl_container_of(kobj)
   ;
+       struct vmcs *original_vmcs = vmcs_store();
+
+       WARN_ON(vmcsctl->vmcs == NULL);
+       vmcs_load(vmcsctl->vmcs);
+       value = vmcs_read32(field);
+       vmcs_load(original_vmcs);
+       return sprintf(buf, "%u\n", value);
+}
+
+static ssize_t vmcs_field_store_u32(struct kobject *kobj,
+       struct kobj_attribute *attr, const char *buf, size_t
    count,
+       enum vmcs_field field)
+{
+       int ret;
+       u32 value;
```

```
+       struct vmcsctl *vmcsctl = vmcsctl_container_of(kobj)
   ;
+       struct vmcs *original_vmcs = vmcs_store();
+
+       WARN_ON(vmcsctl->vmcs == NULL);
+       vmcs_load(vmcsctl->vmcs);
+       ret = kstrtouint(buf, 10, &value);
+       if (ret < 0)
+               return ret;
+
+       vmcs_write32(field, value);
+       vmcs_load(original_vmcs);
+       return count;
+}
+
+static ssize_t vmcs_field_show_u64(struct kobject *kobj,
+       struct kobj_attribute *attr, char *buf, enum
   vmcs_field field)
+{
+       u64 value;
+       struct vmcsctl *vmcsctl = vmcsctl_container_of(kobj)
   ;
+       struct vmcs *original_vmcs = vmcs_store();
+
+       WARN_ON(vmcsctl->vmcs == NULL);
+       vmcs_load(vmcsctl->vmcs);
+       value = vmcs_read64(field);
+
+       vmcs_load(original_vmcs);
+       return sprintf(buf, "%llu\n", value);
+}
+
+static ssize_t vmcs_field_store_u64(struct kobject *kobj,
+       struct kobj_attribute *attr, const char *buf, size_t
    count,
+       enum vmcs_field field)
+{
+       int ret;
+       u64 value;
+       struct vmcsctl *vmcsctl = vmcsctl_container_of(kobj)
   ;
+       struct vmcs *original_vmcs = vmcs_store();
+
+       vmcs_load(vmcsctl->vmcs);
+
```

```
+       WARN_ON(vmcsctl->vmcs == NULL);
+       ret = kstrtoull(buf, 10, &value);
+       if (ret < 0)
+               return ret;
+
+       vmcs_write64(field, value);
+
+       vmcs_load(original_vmcs);
+       return count;
+}
+
+
+#ifdef x86_64
+#define natural_width u64
+#else
+#define natural_width u32
+#endif
+
+#define VMCS_ATTR_SHOW(attr_field, type) \
+static ssize_t vmcs_##attr_field##_show(struct kobject *
   kobj, \
+               struct kobj_attribute *attr, char *buf) \
+{ \
+       if (vmxon) { \
+               return vmcs_field_show_##type(kobj, attr,
   buf, attr_field); \
+       } else { \
+               return -1; \
+       } \
+}
+
+#define VMCS_ATTR_STORE(attr_field, type) \
+static ssize_t vmcs_##attr_field##_store(struct kobject *
   kobj, \
+               struct kobj_attribute *attr, const char *buf
   , size_t count) \
+{ \
+       if (vmxon) { \
+               return vmcs_field_store_##type(kobj, attr,
   buf, count, \
+                                       attr_field); \
+       } else { \
+               return -1; \
+       } \
+}
```

```
+
+#define VMCS_ATTR(attr_field, type) \
+       VMCS_ATTR_SHOW(attr_field, type) \
+       VMCS_ATTR_STORE(attr_field, type) \
+       static struct kobj_attribute vmcs_field_##attr_field
    = \
+               __ATTR(attr_field, 0644, vmcs_##attr_field##
   _show, \
+                      vmcs_##attr_field##_store)
+
+#define VMCS_ATTR_RO(attr_field, type) \
+       VMCS_ATTR_SHOW(attr_field, type) \
+       static struct kobj_attribute vmcs_field_##attr_field
    = \
+               __ATTR(attr_field, 0444, vmcs_##attr_field##
   _show, \
+                      NULL)
+
+VMCS_ATTR(VIRTUAL_PROCESSOR_ID, u16);
+VMCS_ATTR(POSTED_INTR_NV, u16);
+VMCS_ATTR(GUEST_ES_SELECTOR, u16);
+VMCS_ATTR(GUEST_CS_SELECTOR, u16);
+VMCS_ATTR(GUEST_SS_SELECTOR, u16);
+VMCS_ATTR(GUEST_DS_SELECTOR, u16);
+VMCS_ATTR(GUEST_FS_SELECTOR, u16);
+VMCS_ATTR(GUEST_GS_SELECTOR, u16);
+VMCS_ATTR(GUEST_LDTR_SELECTOR, u16);
+VMCS_ATTR(GUEST_TR_SELECTOR, u16);
+VMCS_ATTR(GUEST_INTR_STATUS, u16);
+VMCS_ATTR(GUEST_PML_INDEX, u16);
+VMCS_ATTR(HOST_ES_SELECTOR, u16);
+VMCS_ATTR(HOST_CS_SELECTOR, u16);
+VMCS_ATTR(HOST_SS_SELECTOR, u16);
+VMCS_ATTR(HOST_DS_SELECTOR, u16);
+VMCS_ATTR(HOST_FS_SELECTOR, u16);
+VMCS_ATTR(HOST_GS_SELECTOR, u16);
+VMCS_ATTR(HOST_TR_SELECTOR, u16);
+VMCS_ATTR(IO_BITMAP_A, u64);
+VMCS_ATTR(IO_BITMAP_A_HIGH, u64);
+VMCS_ATTR(IO_BITMAP_B, u64);
+VMCS_ATTR(IO_BITMAP_B_HIGH, u64);
+VMCS_ATTR(MSR_BITMAP, u64);
+VMCS_ATTR(MSR_BITMAP_HIGH, u64);
+VMCS_ATTR(VM_EXIT_MSR_STORE_ADDR, u64);
+VMCS_ATTR(VM_EXIT_MSR_STORE_ADDR_HIGH, u64);
```

```
+VMCS_ATTR(VM_EXIT_MSR_LOAD_ADDR, u64);
+VMCS_ATTR(VM_EXIT_MSR_LOAD_ADDR_HIGH, u64);
+VMCS_ATTR(VM_ENTRY_MSR_LOAD_ADDR, u64);
+VMCS_ATTR(VM_ENTRY_MSR_LOAD_ADDR_HIGH, u64);
+VMCS_ATTR(PML_ADDRESS, u64);
+VMCS_ATTR(PML_ADDRESS_HIGH, u64);
+VMCS_ATTR(TSC_OFFSET, u64);
+VMCS_ATTR(TSC_OFFSET_HIGH, u64);
+VMCS_ATTR(VIRTUAL_APIC_PAGE_ADDR, u64);
+VMCS_ATTR(VIRTUAL_APIC_PAGE_ADDR_HIGH, u64);
+VMCS_ATTR(APIC_ACCESS_ADDR, u64);
+VMCS_ATTR(APIC_ACCESS_ADDR_HIGH, u64);
+VMCS_ATTR(POSTED_INTR_DESC_ADDR, u64);
+VMCS_ATTR(POSTED_INTR_DESC_ADDR_HIGH, u64);
+VMCS_ATTR(EPT_POINTER, u64);
+VMCS_ATTR(EPT_POINTER_HIGH, u64);
+VMCS_ATTR(EOI_EXIT_BITMAP0, u64);
+VMCS_ATTR(EOI_EXIT_BITMAP0_HIGH, u64);
+VMCS_ATTR(EOI_EXIT_BITMAP1, u64);
+VMCS_ATTR(EOI_EXIT_BITMAP1_HIGH, u64);
+VMCS_ATTR(EOI_EXIT_BITMAP2, u64);
+VMCS_ATTR(EOI_EXIT_BITMAP2_HIGH, u64);
+VMCS_ATTR(EOI_EXIT_BITMAP3, u64);
+VMCS_ATTR(EOI_EXIT_BITMAP3_HIGH, u64);
+VMCS_ATTR(VMREAD_BITMAP, u64);
+VMCS_ATTR(VMWRITE_BITMAP, u64);
+VMCS_ATTR(XSS_EXIT_BITMAP, u64);
+VMCS_ATTR(XSS_EXIT_BITMAP_HIGH, u64);
+VMCS_ATTR(TSC_MULTIPLIER, u64);
+VMCS_ATTR(TSC_MULTIPLIER_HIGH, u64);
+VMCS_ATTR_RO(GUEST_PHYSICAL_ADDRESS, u64);
+VMCS_ATTR_RO(GUEST_PHYSICAL_ADDRESS_HIGH, u64);
+VMCS_ATTR(VMCS_LINK_POINTER, u64);
+VMCS_ATTR(VMCS_LINK_POINTER_HIGH, u64);
+VMCS_ATTR(GUEST_IA32_DEBUGCTL, u64);
+VMCS_ATTR(GUEST_IA32_DEBUGCTL_HIGH, u64);
+VMCS_ATTR(GUEST_IA32_PAT, u64);
+VMCS_ATTR(GUEST_IA32_PAT_HIGH, u64);
+VMCS_ATTR(GUEST_IA32_EFER, u64);
+VMCS_ATTR(GUEST_IA32_EFER_HIGH, u64);
+VMCS_ATTR(GUEST_IA32_PERF_GLOBAL_CTRL, u64);
+VMCS_ATTR(GUEST_IA32_PERF_GLOBAL_CTRL_HIGH, u64);
+VMCS_ATTR(GUEST_PDPTR0, u64);
+VMCS_ATTR(GUEST_PDPTR0_HIGH, u64);
+VMCS_ATTR(GUEST_PDPTR1, u64);
```

```
+VMCS_ATTR(GUEST_PDPTR1_HIGH, u64);
+VMCS_ATTR(GUEST_PDPTR2, u64);
+VMCS_ATTR(GUEST_PDPTR2_HIGH, u64);
+VMCS_ATTR(GUEST_PDPTR3, u64);
+VMCS_ATTR(GUEST_PDPTR3_HIGH, u64);
+VMCS_ATTR(GUEST_BNDCFGS, u64);
+VMCS_ATTR(GUEST_BNDCFGS_HIGH, u64);
+VMCS_ATTR(HOST_IA32_PAT, u64);
+VMCS_ATTR(HOST_IA32_PAT_HIGH, u64);
+VMCS_ATTR(HOST_IA32_EFER, u64);
+VMCS_ATTR(HOST_IA32_EFER_HIGH, u64);
+VMCS_ATTR(HOST_IA32_PERF_GLOBAL_CTRL, u64);
+VMCS_ATTR(HOST_IA32_PERF_GLOBAL_CTRL_HIGH, u64);
+VMCS_ATTR(PIN_BASED_VM_EXEC_CONTROL, u32);
+VMCS_ATTR(CPU_BASED_VM_EXEC_CONTROL, u32);
+VMCS_ATTR(EXCEPTION_BITMAP, u32);
+VMCS_ATTR(PAGE_FAULT_ERROR_CODE_MASK, u32);
+VMCS_ATTR(PAGE_FAULT_ERROR_CODE_MATCH, u32);
+VMCS_ATTR(CR3_TARGET_COUNT, u32);
+VMCS_ATTR(VM_EXIT_CONTROLS, u32);
+VMCS_ATTR(VM_EXIT_MSR_STORE_COUNT, u32);
+VMCS_ATTR(VM_EXIT_MSR_LOAD_COUNT, u32);
+VMCS_ATTR(VM_ENTRY_CONTROLS, u32);
+VMCS_ATTR(VM_ENTRY_MSR_LOAD_COUNT, u32);
+VMCS_ATTR(VM_ENTRY_INTR_INFO_FIELD, u32);
+VMCS_ATTR(VM_ENTRY_EXCEPTION_ERROR_CODE, u32);
+VMCS_ATTR(VM_ENTRY_INSTRUCTION_LEN, u32);
+VMCS_ATTR(TPR_THRESHOLD, u32);
+VMCS_ATTR(SECONDARY_VM_EXEC_CONTROL, u32);
+VMCS_ATTR(PLE_GAP, u32);
+VMCS_ATTR(PLE_WINDOW, u32);
+VMCS_ATTR_RO(VM_INSTRUCTION_ERROR, u32);
+VMCS_ATTR_RO(VM_EXIT_REASON, u32);
+VMCS_ATTR_RO(VM_EXIT_INTR_INFO, u32);
+VMCS_ATTR_RO(VM_EXIT_INTR_ERROR_CODE, u32);
+VMCS_ATTR_RO(IDT_VECTORING_INFO_FIELD, u32);
+VMCS_ATTR_RO(IDT_VECTORING_ERROR_CODE, u32);
+VMCS_ATTR_RO(VM_EXIT_INSTRUCTION_LEN, u32);
+VMCS_ATTR_RO(VMX_INSTRUCTION_INFO, u32);
+VMCS_ATTR(GUEST_ES_LIMIT, u32);
+VMCS_ATTR(GUEST_CS_LIMIT, u32);
+VMCS_ATTR(GUEST_SS_LIMIT, u32);
+VMCS_ATTR(GUEST_DS_LIMIT, u32);
+VMCS_ATTR(GUEST_FS_LIMIT, u32);
+VMCS_ATTR(GUEST_GS_LIMIT, u32);
```

```
+VMCS_ATTR(GUEST_LDTR_LIMIT, u32);
+VMCS_ATTR(GUEST_TR_LIMIT, u32);
+VMCS_ATTR(GUEST_GDTR_LIMIT, u32);
+VMCS_ATTR(GUEST_IDTR_LIMIT, u32);
+VMCS_ATTR(GUEST_ES_AR_BYTES, u32);
+VMCS_ATTR(GUEST_CS_AR_BYTES, u32);
+VMCS_ATTR(GUEST_SS_AR_BYTES, u32);
+VMCS_ATTR(GUEST_DS_AR_BYTES, u32);
+VMCS_ATTR(GUEST_FS_AR_BYTES, u32);
+VMCS_ATTR(GUEST_GS_AR_BYTES, u32);
+VMCS_ATTR(GUEST_LDTR_AR_BYTES, u32);
+VMCS_ATTR(GUEST_TR_AR_BYTES, u32);
+VMCS_ATTR(GUEST_INTERRUPTIBILITY_INFO, u32);
+VMCS_ATTR(GUEST_ACTIVITY_STATE, u32);
+VMCS_ATTR(GUEST_SYSENTER_CS, u32);
+VMCS_ATTR(VMX_PREEMPTION_TIMER_VALUE, u32);
+VMCS_ATTR(HOST_IA32_SYSENTER_CS, u32);
+VMCS_ATTR(CR0_GUEST_HOST_MASK, natural_width);
+VMCS_ATTR(CR4_GUEST_HOST_MASK, natural_width);
+VMCS_ATTR(CR0_READ_SHADOW, natural_width);
+VMCS_ATTR(CR4_READ_SHADOW, natural_width);
+VMCS_ATTR(CR3_TARGET_VALUE0, natural_width);
+VMCS_ATTR(CR3_TARGET_VALUE1, natural_width);
+VMCS_ATTR(CR3_TARGET_VALUE2, natural_width);
+VMCS_ATTR(CR3_TARGET_VALUE3, natural_width);
+VMCS_ATTR_RO(EXIT_QUALIFICATION, natural_width);
+VMCS_ATTR_RO(GUEST_LINEAR_ADDRESS, natural_width);
+VMCS_ATTR(GUEST_CR0, natural_width);
+VMCS_ATTR(GUEST_CR3, natural_width);
+VMCS_ATTR(GUEST_CR4, natural_width);
+VMCS_ATTR(GUEST_ES_BASE, natural_width);
+VMCS_ATTR(GUEST_CS_BASE, natural_width);
+VMCS_ATTR(GUEST_SS_BASE, natural_width);
+VMCS_ATTR(GUEST_DS_BASE, natural_width);
+VMCS_ATTR(GUEST_FS_BASE, natural_width);
+VMCS_ATTR(GUEST_GS_BASE, natural_width);
+VMCS_ATTR(GUEST_LDTR_BASE, natural_width);
+VMCS_ATTR(GUEST_TR_BASE, natural_width);
+VMCS_ATTR(GUEST_GDTR_BASE, natural_width);
+VMCS_ATTR(GUEST_IDTR_BASE, natural_width);
+VMCS_ATTR(GUEST_DR7, natural_width);
+VMCS_ATTR(GUEST_RSP, natural_width);
+VMCS_ATTR(GUEST_RIP, natural_width);
+VMCS_ATTR(GUEST_RFLAGS, natural_width);
+VMCS_ATTR(GUEST_PENDING_DBG_EXCEPTIONS, natural_width);
```

```
+VMCS_ATTR(GUEST_SYSENTER_ESP, natural_width);
+VMCS_ATTR(GUEST_SYSENTER_EIP, natural_width);
+VMCS_ATTR(HOST_CR0, natural_width);
+VMCS_ATTR(HOST_CR3, natural_width);
+VMCS_ATTR(HOST_CR4, natural_width);
+VMCS_ATTR(HOST_FS_BASE, natural_width);
+VMCS_ATTR(HOST_GS_BASE, natural_width);
+VMCS_ATTR(HOST_TR_BASE, natural_width);
+VMCS_ATTR(HOST_GDTR_BASE, natural_width);
+VMCS_ATTR(HOST_IDTR_BASE, natural_width);
+VMCS_ATTR(HOST_IA32_SYSENTER_ESP, natural_width);
+VMCS_ATTR(HOST_IA32_SYSENTER_EIP, natural_width);
+VMCS_ATTR(HOST_RSP, natural_width);
+VMCS_ATTR(HOST_RIP, natural_width);
+
+static struct kobj_attribute revision_id_attribute =
+       __ATTR(revision_id, 0444, revision_id_show, NULL);
+
+static struct kobj_attribute abort_attribute =
+       __ATTR(abort, 0444, abort_show, NULL);
+
+static struct attribute *vmcsctl_attrs[] = {
+       &revision_id_attribute.attr,
+       &abort_attribute.attr,
+       &vmcs_field_VIRTUAL_PROCESSOR_ID.attr,
+       &vmcs_field_POSTED_INTR_NV.attr,
+       &vmcs_field_GUEST_ES_SELECTOR.attr,
+       &vmcs_field_GUEST_CS_SELECTOR.attr,
+       &vmcs_field_GUEST_SS_SELECTOR.attr,
+       &vmcs_field_GUEST_DS_SELECTOR.attr,
+       &vmcs_field_GUEST_FS_SELECTOR.attr,
+       &vmcs_field_GUEST_GS_SELECTOR.attr,
+       &vmcs_field_GUEST_LDTR_SELECTOR.attr,
+       &vmcs_field_GUEST_TR_SELECTOR.attr,
+       &vmcs_field_GUEST_INTR_STATUS.attr,
+       &vmcs_field_GUEST_PML_INDEX.attr,
+       &vmcs_field_HOST_ES_SELECTOR.attr,
+       &vmcs_field_HOST_CS_SELECTOR.attr,
+       &vmcs_field_HOST_SS_SELECTOR.attr,
+       &vmcs_field_HOST_DS_SELECTOR.attr,
+       &vmcs_field_HOST_FS_SELECTOR.attr,
+       &vmcs_field_HOST_GS_SELECTOR.attr,
+       &vmcs_field_HOST_TR_SELECTOR.attr,
+       &vmcs_field_IO_BITMAP_A.attr,
+       &vmcs_field_IO_BITMAP_A_HIGH.attr,
```

```
+       &vmcs_field_IO_BITMAP_B.attr,
+       &vmcs_field_IO_BITMAP_B_HIGH.attr,
+       &vmcs_field_MSR_BITMAP.attr,
+       &vmcs_field_MSR_BITMAP_HIGH.attr,
+       &vmcs_field_VM_EXIT_MSR_STORE_ADDR.attr,
+       &vmcs_field_VM_EXIT_MSR_STORE_ADDR_HIGH.attr,
+       &vmcs_field_VM_EXIT_MSR_LOAD_ADDR.attr,
+       &vmcs_field_VM_EXIT_MSR_LOAD_ADDR_HIGH.attr,
+       &vmcs_field_VM_ENTRY_MSR_LOAD_ADDR.attr,
+       &vmcs_field_VM_ENTRY_MSR_LOAD_ADDR_HIGH.attr,
+       &vmcs_field_PML_ADDRESS.attr,
+       &vmcs_field_PML_ADDRESS_HIGH.attr,
+       &vmcs_field_TSC_OFFSET.attr,
+       &vmcs_field_TSC_OFFSET_HIGH.attr,
+       &vmcs_field_VIRTUAL_APIC_PAGE_ADDR.attr,
+       &vmcs_field_VIRTUAL_APIC_PAGE_ADDR_HIGH.attr,
+       &vmcs_field_APIC_ACCESS_ADDR.attr,
+       &vmcs_field_APIC_ACCESS_ADDR_HIGH.attr,
+       &vmcs_field_POSTED_INTR_DESC_ADDR.attr,
+       &vmcs_field_POSTED_INTR_DESC_ADDR_HIGH.attr,
+       &vmcs_field_EPT_POINTER.attr,
+       &vmcs_field_EPT_POINTER_HIGH.attr,
+       &vmcs_field_EOI_EXIT_BITMAP0.attr,
+       &vmcs_field_EOI_EXIT_BITMAP0_HIGH.attr,
+       &vmcs_field_EOI_EXIT_BITMAP1.attr,
+       &vmcs_field_EOI_EXIT_BITMAP1_HIGH.attr,
+       &vmcs_field_EOI_EXIT_BITMAP2.attr,
+       &vmcs_field_EOI_EXIT_BITMAP2_HIGH.attr,
+       &vmcs_field_EOI_EXIT_BITMAP3.attr,
+       &vmcs_field_EOI_EXIT_BITMAP3_HIGH.attr,
+       &vmcs_field_VMREAD_BITMAP.attr,
+       &vmcs_field_VMWRITE_BITMAP.attr,
+       &vmcs_field_XSS_EXIT_BITMAP.attr,
+       &vmcs_field_XSS_EXIT_BITMAP_HIGH.attr,
+       &vmcs_field_TSC_MULTIPLIER.attr,
+       &vmcs_field_TSC_MULTIPLIER_HIGH.attr,
+       &vmcs_field_GUEST_PHYSICAL_ADDRESS.attr,
+       &vmcs_field_GUEST_PHYSICAL_ADDRESS_HIGH.attr,
+       &vmcs_field_VMCS_LINK_POINTER.attr,
+       &vmcs_field_VMCS_LINK_POINTER_HIGH.attr,
+       &vmcs_field_GUEST_IA32_DEBUGCTL.attr,
+       &vmcs_field_GUEST_IA32_DEBUGCTL_HIGH.attr,
+       &vmcs_field_GUEST_IA32_PAT.attr,
+       &vmcs_field_GUEST_IA32_PAT_HIGH.attr,
+       &vmcs_field_GUEST_IA32_EFER.attr,
```

```
+       &vmcs_field_GUEST_IA32_EFER_HIGH.attr,
+       &vmcs_field_GUEST_IA32_PERF_GLOBAL_CTRL.attr,
+       &vmcs_field_GUEST_IA32_PERF_GLOBAL_CTRL_HIGH.attr,
+       &vmcs_field_GUEST_PDPTR0.attr,
+       &vmcs_field_GUEST_PDPTR0_HIGH.attr,
+       &vmcs_field_GUEST_PDPTR1.attr,
+       &vmcs_field_GUEST_PDPTR1_HIGH.attr,
+       &vmcs_field_GUEST_PDPTR2.attr,
+       &vmcs_field_GUEST_PDPTR2_HIGH.attr,
+       &vmcs_field_GUEST_PDPTR3.attr,
+       &vmcs_field_GUEST_PDPTR3_HIGH.attr,
+       &vmcs_field_GUEST_BNDCFGS.attr,
+       &vmcs_field_GUEST_BNDCFGS_HIGH.attr,
+       &vmcs_field_HOST_IA32_PAT.attr,
+       &vmcs_field_HOST_IA32_PAT_HIGH.attr,
+       &vmcs_field_HOST_IA32_EFER.attr,
+       &vmcs_field_HOST_IA32_EFER_HIGH.attr,
+       &vmcs_field_HOST_IA32_PERF_GLOBAL_CTRL.attr,
+       &vmcs_field_HOST_IA32_PERF_GLOBAL_CTRL_HIGH.attr,
+       &vmcs_field_PIN_BASED_VM_EXEC_CONTROL.attr,
+       &vmcs_field_CPU_BASED_VM_EXEC_CONTROL.attr,
+       &vmcs_field_EXCEPTION_BITMAP.attr,
+       &vmcs_field_PAGE_FAULT_ERROR_CODE_MASK.attr,
+       &vmcs_field_PAGE_FAULT_ERROR_CODE_MATCH.attr,
+       &vmcs_field_CR3_TARGET_COUNT.attr,
+       &vmcs_field_VM_EXIT_CONTROLS.attr,
+       &vmcs_field_VM_EXIT_MSR_STORE_COUNT.attr,
+       &vmcs_field_VM_EXIT_MSR_LOAD_COUNT.attr,
+       &vmcs_field_VM_ENTRY_CONTROLS.attr,
+       &vmcs_field_VM_ENTRY_MSR_LOAD_COUNT.attr,
+       &vmcs_field_VM_ENTRY_INTR_INFO_FIELD.attr,
+       &vmcs_field_VM_ENTRY_EXCEPTION_ERROR_CODE.attr,
+       &vmcs_field_VM_ENTRY_INSTRUCTION_LEN.attr,
+       &vmcs_field_TPR_THRESHOLD.attr,
+       &vmcs_field_SECONDARY_VM_EXEC_CONTROL.attr,
+       &vmcs_field_PLE_GAP.attr,
+       &vmcs_field_PLE_WINDOW.attr,
+       &vmcs_field_VM_INSTRUCTION_ERROR.attr,
+       &vmcs_field_VM_EXIT_REASON.attr,
+       &vmcs_field_VM_EXIT_INTR_INFO.attr,
+       &vmcs_field_VM_EXIT_INTR_ERROR_CODE.attr,
+       &vmcs_field_IDT_VECTORING_INFO_FIELD.attr,
+       &vmcs_field_IDT_VECTORING_ERROR_CODE.attr,
+       &vmcs_field_VM_EXIT_INSTRUCTION_LEN.attr,
+       &vmcs_field_VMX_INSTRUCTION_INFO.attr,
```

```
+       &vmcs_field_GUEST_ES_LIMIT.attr,
+       &vmcs_field_GUEST_CS_LIMIT.attr,
+       &vmcs_field_GUEST_SS_LIMIT.attr,
+       &vmcs_field_GUEST_DS_LIMIT.attr,
+       &vmcs_field_GUEST_FS_LIMIT.attr,
+       &vmcs_field_GUEST_GS_LIMIT.attr,
+       &vmcs_field_GUEST_LDTR_LIMIT.attr,
+       &vmcs_field_GUEST_TR_LIMIT.attr,
+       &vmcs_field_GUEST_GDTR_LIMIT.attr,
+       &vmcs_field_GUEST_IDTR_LIMIT.attr,
+       &vmcs_field_GUEST_ES_AR_BYTES.attr,
+       &vmcs_field_GUEST_CS_AR_BYTES.attr,
+       &vmcs_field_GUEST_SS_AR_BYTES.attr,
+       &vmcs_field_GUEST_DS_AR_BYTES.attr,
+       &vmcs_field_GUEST_FS_AR_BYTES.attr,
+       &vmcs_field_GUEST_GS_AR_BYTES.attr,
+       &vmcs_field_GUEST_LDTR_AR_BYTES.attr,
+       &vmcs_field_GUEST_TR_AR_BYTES.attr,
+       &vmcs_field_GUEST_INTERRUPTIBILITY_INFO.attr,
+       &vmcs_field_GUEST_ACTIVITY_STATE.attr,
+       &vmcs_field_GUEST_SYSENTER_CS.attr,
+       &vmcs_field_VMX_PREEMPTION_TIMER_VALUE.attr,
+       &vmcs_field_HOST_IA32_SYSENTER_CS.attr,
+       &vmcs_field_CR0_GUEST_HOST_MASK.attr,
+       &vmcs_field_CR4_GUEST_HOST_MASK.attr,
+       &vmcs_field_CR0_READ_SHADOW.attr,
+       &vmcs_field_CR4_READ_SHADOW.attr,
+       &vmcs_field_CR3_TARGET_VALUE0.attr,
+       &vmcs_field_CR3_TARGET_VALUE1.attr,
+       &vmcs_field_CR3_TARGET_VALUE2.attr,
+       &vmcs_field_CR3_TARGET_VALUE3.attr,
+       &vmcs_field_EXIT_QUALIFICATION.attr,
+       &vmcs_field_GUEST_LINEAR_ADDRESS.attr,
+       &vmcs_field_GUEST_CR0.attr,
+       &vmcs_field_GUEST_CR3.attr,
+       &vmcs_field_GUEST_CR4.attr,
+       &vmcs_field_GUEST_ES_BASE.attr,
+       &vmcs_field_GUEST_CS_BASE.attr,
+       &vmcs_field_GUEST_SS_BASE.attr,
+       &vmcs_field_GUEST_DS_BASE.attr,
+       &vmcs_field_GUEST_FS_BASE.attr,
+       &vmcs_field_GUEST_GS_BASE.attr,
+       &vmcs_field_GUEST_LDTR_BASE.attr,
+       &vmcs_field_GUEST_TR_BASE.attr,
+       &vmcs_field_GUEST_GDTR_BASE.attr,
```

```
+       &vmcs_field_GUEST_IDTR_BASE.attr,
+       &vmcs_field_GUEST_DR7.attr,
+       &vmcs_field_GUEST_RSP.attr,
+       &vmcs_field_GUEST_RIP.attr,
+       &vmcs_field_GUEST_RFLAGS.attr,
+       &vmcs_field_GUEST_PENDING_DBG_EXCEPTIONS.attr,
+       &vmcs_field_GUEST_SYSENTER_ESP.attr,
+       &vmcs_field_GUEST_SYSENTER_EIP.attr,
+       &vmcs_field_HOST_CR0.attr,
+       &vmcs_field_HOST_CR3.attr,
+       &vmcs_field_HOST_CR4.attr,
+       &vmcs_field_HOST_FS_BASE.attr,
+       &vmcs_field_HOST_GS_BASE.attr,
+       &vmcs_field_HOST_TR_BASE.attr,
+       &vmcs_field_HOST_GDTR_BASE.attr,
+       &vmcs_field_HOST_IDTR_BASE.attr,
+       &vmcs_field_HOST_IA32_SYSENTER_ESP.attr,
+       &vmcs_field_HOST_IA32_SYSENTER_EIP.attr,
+       &vmcs_field_HOST_RSP.attr,
+       &vmcs_field_HOST_RIP.attr,
+       NULL,   /* need to NULL terminate the list of
   attributes */
+};
+
+static struct attribute_group attr_group = {
+       .attrs = vmcsctl_attrs,
+};
+
+static struct vmcsctl *vmcsctl_create(struct vmcs *vmcs)
+{
+       struct vmcsctl *new;
+
+       new = kzalloc(sizeof(*new), GFP_KERNEL);
+       if (new == NULL)
+               return NULL;
+       kobject_init(&new->kobj, &vmcsctl_kobj_ktype);
+       new->vmcs = vmcs;
+       new->pid = task_pid_nr(current);
+       return new;
+}
+
+static void vmcsctl_del(struct vmcsctl *vmcsctl)
+{
+       kobject_del(&vmcsctl->kobj);
+       kfree(vmcsctl);
```

```
+}
+
+int vmcsctl_register(struct vmcs *vmcs)
+{
+       int err;
+       struct vmcsctl *vmcsctl;
+
+       WARN_ON(vmcs == NULL);
+       vmcsctl = vmcsctl_create(vmcs);
+       if (vmcsctl == NULL)
+               return -1;
+       vmcsctl->kobj.kset = vmcsctl_set;
+       err = kobject_add(&vmcsctl->kobj, NULL, "vmcs%d",
+                         vmcsctl->pid);
+       if (err != 0)
+               goto out;
+       err = sysfs_create_group(&vmcsctl->kobj, &attr_group
   );
+       if (err != 0)
+               goto out;
+       return 0;
+out:
+       vmcsctl_del(vmcsctl);
+       return err;
+}
+
+void vmcsctl_unregister(struct vmcs *vmcs)
+{
+       struct kobject *kobj;
+       struct vmcsctl *vmcsctl;
+
+       list_for_each_entry(kobj, &vmcsctl_set->list, entry)
    {
+               vmcsctl = vmcsctl_container_of(kobj);
+               if (vmcsctl->vmcs == vmcs) {
+                       vmcsctl_del(vmcsctl);
+                       return;
+               }
+       }
+}
+
+void vmcsctl_vmxon(void)
+{
+       vmxon = true;
+}
```

```
+
+void vmcsctl_vmxoff(void)
+{
+       vmxon = false;
+}
+
+static int __init vmcsctl_init(void)
+{
+       int err;
+
+       vmcsctl_set = kset_create_and_add("vmcsctl", NULL,
   kernel_kobj);
+       if (vmcsctl_set == NULL)
+               return -ENOMEM;
+       err = kset_register(vmcsctl_set);
+       if (err != 0)
+               return err;
+       return 0;
+}
+
+static void __exit vmcsctl_exit(void)
+{
+       kset_unregister(vmcsctl_set);
+       kset_put(vmcsctl_set);
+}
+
+module_init(vmcsctl_init);
+module_exit(vmcsctl_exit);
+
+MODULE_AUTHOR("Ian Kronquist <iankronquist@gmail.com>");
+MODULE_LICENSE("Dual MIT/GPL");
diff --git a/arch/x86/kvm/vmcsctl.h b/arch/x86/kvm/vmcsctl.h
new file mode 100644
index 000000000000..022d4878f2b8
--- /dev/null
+++ b/arch/x86/kvm/vmcsctl.h
@@ -0,0 +1,27 @@
+#ifndef __VMCSCTL_H
+#define __VMCSCTL_H
+
+#include <linux/kobject.h>
+#include <linux/module.h>
+#include <linux/sched.h>
+#include <linux/slab.h>
+#include <linux/sysfs.h>
```

```
+
+#include <asm/vmx.h>
+
+struct vmcs;
+
+struct vmcsctl {
+       int pid;
+       struct kobject kobj;
+       struct vmcs *vmcs;
+};
+
+int vmcsctl_register(struct vmcs *vmcs);
+
+void vmcsctl_unregister(struct vmcs *vmcs);
+
+void vmcsctl_vmxon(void);
+
+void vmcsctl_vmxoff(void);
+#endif
diff --git a/arch/x86/kvm/vmx.c b/arch/x86/kvm/vmx.c
index 259e9b28ccf8..262f09007ca3 100644
--- a/arch/x86/kvm/vmx.c
+++ b/arch/x86/kvm/vmx.c
@@ -36,6 +36,10 @@
 #include "kvm_cache_regs.h"
 #include "x86.h"

+#ifdef CONFIG_KVM_VMCSCTL
+#include "vmcsctl.h"
+#endif
+
 #include <asm/cpu.h>
 #include <asm/io.h>
 #include <asm/desc.h>
@@ -52,10 +56,6 @@
 #include "trace.h"
 #include "pmu.h"

-#define __ex(x) __kvm_handle_fault_on_reboot(x)
-#define __ex_clear(x, reg) \
-       ____kvm_handle_fault_on_reboot(x, "xor " reg " , "
   reg)
-
 MODULE_AUTHOR("Qumranet");
 MODULE_LICENSE("GPL");
```

```
@@ -184,12 +184,6 @@ extern const ulong vmx_return;
 #define NR_AUTOLOAD_MSRS 8
 #define VMCS02_POOL_SIZE 1

-struct vmcs {
-       u32 revision_id;
-       u32 abort;
-       char data[0];
-};
-
 /*
  * Track a VMCS that may be loaded on a certain CPU. If it
    is (cpu!=-1), also
  * remember whether it was VMLAUNCHed, and maintain a
    linked list of all VMCSs
@@ -1469,7 +1463,7 @@ static inline void loaded_vmcs_init(
   struct loaded_vmcs *loaded_vmcs)
        loaded_vmcs->launched = 0;
 }

-static void vmcs_load(struct vmcs *vmcs)
+void vmcs_load(struct vmcs *vmcs)
 {
        u64 phys_addr = __pa(vmcs);
        u8 error;
@@ -1482,6 +1476,19 @@ static void vmcs_load(struct vmcs *
   vmcs)
                        vmcs, phys_addr);
 }

+struct vmcs *vmcs_store(void)
+{
+       struct vmcs *vmcs;
+       u64 phys_addr;
+
+       asm volatile (__ex(ASM_VMX_VMPTRST_RAX)
+                     : : "a"(&phys_addr)
+                     : "memory");
+
+       vmcs = __va(phys_addr);
+       return vmcs;
+}
+
 #ifdef CONFIG_KEXEC_CORE
```

```
 /*
  * This bitmap is used to indicate whether the vmclear
@@ -1594,132 +1601,13 @@ static inline void ept_sync_context
   (u64 eptp)
        }
 }

-static __always_inline void vmcs_check16(unsigned long
   field)
-{
-        BUILD_BUG_ON_MSG(__builtin_constant_p(field) && ((
   field) & 0x6001) == 0x2000,
-                         "16-bit accessor invalid for 64-bit
    field");
-        BUILD_BUG_ON_MSG(__builtin_constant_p(field) && ((
   field) & 0x6001) == 0x2001,
-                         "16-bit accessor invalid for 64-bit
    high field");
-        BUILD_BUG_ON_MSG(__builtin_constant_p(field) && ((
   field) & 0x6000) == 0x4000,
-                         "16-bit accessor invalid for 32-bit
    high field");
-        BUILD_BUG_ON_MSG(__builtin_constant_p(field) && ((
   field) & 0x6000) == 0x6000,
-                         "16-bit accessor invalid for
   natural width field");
-}
-
-static __always_inline void vmcs_check32(unsigned long
   field)
-{
-        BUILD_BUG_ON_MSG(__builtin_constant_p(field) && ((
   field) & 0x6000) == 0,
-                         "32-bit accessor invalid for 16-bit
    field");
-        BUILD_BUG_ON_MSG(__builtin_constant_p(field) && ((
   field) & 0x6000) == 0x6000,
-                         "32-bit accessor invalid for
   natural width field");
-}
-
-static __always_inline void vmcs_check64(unsigned long
   field)
-{
```

```
-        BUILD_BUG_ON_MSG(__builtin_constant_p(field) && ((
   field) & 0x6000) == 0,
-                         "64-bit accessor invalid for 16-bit
    field");
-        BUILD_BUG_ON_MSG(__builtin_constant_p(field) && ((
   field) & 0x6001) == 0x2001,
-                         "64-bit accessor invalid for 64-bit
    high field");
-        BUILD_BUG_ON_MSG(__builtin_constant_p(field) && ((
   field) & 0x6000) == 0x4000,
-                         "64-bit accessor invalid for 32-bit
    field");
-        BUILD_BUG_ON_MSG(__builtin_constant_p(field) && ((
   field) & 0x6000) == 0x6000,
-                         "64-bit accessor invalid for
   natural width field");
-}
-
-static __always_inline void vmcs_checkl(unsigned long field
   )
-{
-        BUILD_BUG_ON_MSG(__builtin_constant_p(field) && ((
   field) & 0x6000) == 0,
-                         "Natural width accessor invalid for
    16-bit field");
-        BUILD_BUG_ON_MSG(__builtin_constant_p(field) && ((
   field) & 0x6001) == 0x2000,
-                         "Natural width accessor invalid for
    64-bit field");
-        BUILD_BUG_ON_MSG(__builtin_constant_p(field) && ((
   field) & 0x6001) == 0x2001,
-                         "Natural width accessor invalid for
    64-bit high field");
-        BUILD_BUG_ON_MSG(__builtin_constant_p(field) && ((
   field) & 0x6000) == 0x4000,
-                         "Natural width accessor invalid for
    32-bit field");
-}
-
-static __always_inline unsigned long __vmcs_readl(unsigned
   long field)
-{
-       unsigned long value;
-
```

```
-       asm volatile (__ex_clear(ASM_VMX_VMREAD_RDX_RAX,
   "%0")
-                     : "=a"(value) : "d"(field) : "cc");
-       return value;
-}
-
-static __always_inline u16 vmcs_read16(unsigned long field)
-{
-       vmcs_check16(field);
-       return __vmcs_readl(field);
-}
-
-static __always_inline u32 vmcs_read32(unsigned long field)
-{
-       vmcs_check32(field);
-       return __vmcs_readl(field);
-}
-
-static __always_inline u64 vmcs_read64(unsigned long field)
-{
-       vmcs_check64(field);
-#ifdef CONFIG_X86_64
-       return __vmcs_readl(field);
-#else
-       return __vmcs_readl(field) | ((u64)__vmcs_readl(
   field+1) << 32);
-#endif
-}
-
-static __always_inline unsigned long vmcs_readl(unsigned
   long field)
-{
-       vmcs_checkl(field);
-       return __vmcs_readl(field);
-}
-
-static noinline void vmwrite_error(unsigned long field,
   unsigned long value)
+noinline void vmwrite_error(unsigned long field, unsigned
   long value)
 {
        printk(KERN_ERR "vmwrite error: reg %lx value %lx (
           err %d)\n",
               field, value, vmcs_read32(
                  VM_INSTRUCTION_ERROR));
```

```
        dump_stack();
 }

-static __always_inline void __vmcs_writel(unsigned long
   field, unsigned long value)
-{
-       u8 error;
-
-       asm volatile (__ex(ASM_VMX_VMWRITE_RAX_RDX) "; setna
    %0"
-                     : "=q"(error) : "a"(value), "d"(field
   ) : "cc");
-       if (unlikely(error))
-               vmwrite_error(field, value);
-}
-
-static __always_inline void vmcs_write16(unsigned long
   field, u16 value)
-{
-       vmcs_check16(field);
-       __vmcs_writel(field, value);
-}
-
-static __always_inline void vmcs_write32(unsigned long
   field, u32 value)
-{
-       vmcs_check32(field);
-       __vmcs_writel(field, value);
-}
-
-static __always_inline void vmcs_write64(unsigned long
   field, u64 value)
-{
-       vmcs_check64(field);
-       __vmcs_writel(field, value);
-#ifndef CONFIG_X86_64
-       asm volatile ("");
-       __vmcs_writel(field+1, value >> 32);
-#endif
-}
-
-static __always_inline void vmcs_writel(unsigned long field
   , unsigned long value)
-{
-       vmcs_checkl(field);
```

```
-       __vmcs_writel(field, value);
-}
-
 static __always_inline void vmcs_clear_bits(unsigned long
    field, u32 mask)
 {
         BUILD_BUG_ON_MSG(__builtin_constant_p(field) && ((
            field) & 0x6000) == 0x2000,
@@ -3427,6 +3315,10 @@ static void kvm_cpu_vmxon(u64 addr)
         asm volatile (ASM_VMX_VMXON_RAX
                       : : "a"(&addr), "m"(addr)
                       : "memory", "cc");
+
+#ifdef CONFIG_KVM_VMCSCTL
+       vmcsctl_vmxon();
+#endif
 }

 static int hardware_enable(void)
@@ -3495,6 +3387,10 @@ static void kvm_cpu_vmxoff(void)
         asm volatile (__ex(ASM_VMX_VMXOFF) : : : "cc");

         intel_pt_handle_vmx(0);
+
+#ifdef CONFIG_KVM_VMCSCTL
+       vmcsctl_vmxoff();
+#endif
 }

 static void hardware_disable(void)
@@ -3729,6 +3625,11 @@ static struct vmcs *alloc_vmcs_cpu(
   int cpu)
         vmcs = page_address(pages);
         memset(vmcs, 0, vmcs_config.size);
         vmcs->revision_id = vmcs_config.revision_id; /* vmcs
             revision id */
+
+#ifdef CONFIG_KVM_VMCSCTL
+       vmcsctl_register(vmcs);
+#endif
+
         return vmcs;
 }

@@ -3739,6 +3640,9 @@ static struct vmcs *alloc_vmcs(void)
```

```
 static void free_vmcs(struct vmcs *vmcs)
 {
+#ifdef CONFIG_KVM_VMCSCTL
+       vmcsctl_unregister(vmcs);
+#endif
        free_pages((unsigned long)vmcs, vmcs_config.order);
 }

@@ -3800,6 +3704,7 @@ static void init_vmcs_shadow_fields(
    void)
                            vmx_vmread_bitmap);
 }

+
 static __init int alloc_kvm_area(void)
 {
        int cpu;
--
2.12.2
```