illumos / illumos-gate Public

Code   Pull requests 7   Actions   Security   Insights

84971882a9

illumos-gate / usr / src / uts / i86pc / io / vmm / intel / vmcs.c

pfmooney 12996 bhyve kernel should be wscheck clean ...   History

2 contributors

562 lines (489 sloc) | 12.9 KB

```c

#ifdef  __FreeBSD__
#include "opt_ddb.h"
#endif

#include <sys/cdefs.h>
__FBSDID("$FreeBSD$");

#include <sys/param.h>
#include <sys/sysctl.h>
#include <sys/systm.h>
#include <sys/pcpu.h>

#include <vm/vm.h>
#include <vm/pmap.h>

#include <machine/segments.h>
#include <machine/vmm.h>
#include "vmm_host.h"
#include "vmx_cpufunc.h"
#include "vmcs.h"
#include "ept.h"
#include "vmx.h"

#ifdef DDB
#include <ddb/ddb.h>
#endif

SYSCTL_DECL(_hw_vmm_vmx);

static int no_flush_rsb;
SYSCTL_INT(_hw_vmm_vmx, OID_AUTO, no_flush_rsb, CTLFLAG_RW,
    &no_flush_rsb, 0, "Do not flush RSB upon vmexit");

static uint64_t
vmcs_fix_regval(uint32_t encoding, uint64_t val)
```

```c
{

	switch (encoding) {
	case VMCS_GUEST_CR0:
		val = vmx_fix_cr0(val);
		break;
	case VMCS_GUEST_CR4:
		val = vmx_fix_cr4(val);
		break;
	default:
		break;
	}
	return (val);
}

static uint32_t
vmcs_field_encoding(int ident)
{
	switch (ident) {
	case VM_REG_GUEST_CR0:
		return (VMCS_GUEST_CR0);
	case VM_REG_GUEST_CR3:
		return (VMCS_GUEST_CR3);
	case VM_REG_GUEST_CR4:
		return (VMCS_GUEST_CR4);
	case VM_REG_GUEST_DR7:
		return (VMCS_GUEST_DR7);
	case VM_REG_GUEST_RSP:
		return (VMCS_GUEST_RSP);
	case VM_REG_GUEST_RIP:
		return (VMCS_GUEST_RIP);
	case VM_REG_GUEST_RFLAGS:
		return (VMCS_GUEST_RFLAGS);
	case VM_REG_GUEST_ES:
		return (VMCS_GUEST_ES_SELECTOR);
	case VM_REG_GUEST_CS:
		return (VMCS_GUEST_CS_SELECTOR);
	case VM_REG_GUEST_SS:
		return (VMCS_GUEST_SS_SELECTOR);
	case VM_REG_GUEST_DS:
		return (VMCS_GUEST_DS_SELECTOR);
	case VM_REG_GUEST_FS:
		return (VMCS_GUEST_FS_SELECTOR);
	case VM_REG_GUEST_GS:
		return (VMCS_GUEST_GS_SELECTOR);
	case VM_REG_GUEST_TR:
		return (VMCS_GUEST_TR_SELECTOR);
	case VM_REG_GUEST_LDTR:
		return (VMCS_GUEST_LDTR_SELECTOR);
	case VM_REG_GUEST_EFER:
		return (VMCS_GUEST_IA32_EFER);
	case VM_REG_GUEST_PDPTE0:
		return (VMCS_GUEST_PDPTE0);
	case VM_REG_GUEST_PDPTE1:
		return (VMCS_GUEST_PDPTE1);
	case VM_REG_GUEST_PDPTE2:
		return (VMCS_GUEST_PDPTE2);
	case VM_REG_GUEST_PDPTE3:
		return (VMCS_GUEST_PDPTE3);
	case VM_REG_GUEST_ENTRY_INST_LENGTH:
		return (VMCS_ENTRY_INST_LENGTH);
	default:
		return (-1);
	}

}

static int
vmcs_seg_desc_encoding(int seg, uint32_t *base, uint32_t *lim, uint32_t *acc)
{

	switch (seg) {
	case VM_REG_GUEST_ES:
		*base = VMCS_GUEST_ES_BASE;
		*lim = VMCS_GUEST_ES_LIMIT;
		*acc = VMCS_GUEST_ES_ACCESS_RIGHTS;
		break;
	case VM_REG_GUEST_CS:
		*base = VMCS_GUEST_CS_BASE;
		*lim = VMCS_GUEST_CS_LIMIT;
		*acc = VMCS_GUEST_CS_ACCESS_RIGHTS;
		break;
	case VM_REG_GUEST_SS:
		*base = VMCS_GUEST_SS_BASE;
		*lim = VMCS_GUEST_SS_LIMIT;
		*acc = VMCS_GUEST_SS_ACCESS_RIGHTS;
		break;
	case VM_REG_GUEST_DS:
		*base = VMCS_GUEST_DS_BASE;
		*lim = VMCS_GUEST_DS_LIMIT;
		*acc = VMCS_GUEST_DS_ACCESS_RIGHTS;
		break;
	case VM_REG_GUEST_FS:
		*base = VMCS_GUEST_FS_BASE;
		*lim = VMCS_GUEST_FS_LIMIT;
		*acc = VMCS_GUEST_FS_ACCESS_RIGHTS;
		break;
	case VM_REG_GUEST_GS:
		*base = VMCS_GUEST_GS_BASE;
		*lim = VMCS_GUEST_GS_LIMIT;
		*acc = VMCS_GUEST_GS_ACCESS_RIGHTS;
		break;
	case VM_REG_GUEST_TR:
		*base = VMCS_GUEST_TR_BASE;
		*lim = VMCS_GUEST_TR_LIMIT;
		*acc = VMCS_GUEST_TR_ACCESS_RIGHTS;
		break;
	case VM_REG_GUEST_LDTR:
		*base = VMCS_GUEST_LDTR_BASE;
		*lim = VMCS_GUEST_LDTR_LIMIT;
		*acc = VMCS_GUEST_LDTR_ACCESS_RIGHTS;
		break;
	case VM_REG_GUEST_IDTR:
		*base = VMCS_GUEST_IDTR_BASE;
		*lim = VMCS_GUEST_IDTR_LIMIT;
		*acc = VMCS_INVALID_ENCODING;
		break;
	case VM_REG_GUEST_GDTR:
		*base = VMCS_GUEST_GDTR_BASE;
		*lim = VMCS_GUEST_GDTR_LIMIT;
		*acc = VMCS_INVALID_ENCODING;
		break;
	default:
		return (EINVAL);
	}

	return (0);
}

int
vmcs_getreg(struct vmcs *vmcs, int running, int ident, uint64_t *retval)
{
	int error;
	uint32_t encoding;

	/*
	 * If we need to get at vmx-specific state in the VMCS we can bypass
	 * the translation of 'ident' to 'encoding' by simply setting the
	 * sign bit. As it so happens the upper 16 bits are reserved (i.e
	 * set to 0) in the encodings for the VMCS so we are free to use the
	 * sign bit.
	 */
	if (ident < 0)
		encoding = ident & 0x7fffffff;
	else
		encoding = vmcs_field_encoding(ident);

	if (encoding == (uint32_t)-1)
		return (EINVAL);

	if (!running)
		VMPTRLD(vmcs);

	error = vmread(encoding, retval);

	if (!running)
		VMCLEAR(vmcs);

	return (error);
}

int
vmcs_setreg(struct vmcs *vmcs, int running, int ident, uint64_t val)
{
	int error;
	uint32_t encoding;

	if (ident < 0)
		encoding = ident & 0x7fffffff;
	else
		encoding = vmcs_field_encoding(ident);

	if (encoding == (uint32_t)-1)
		return (EINVAL);

	val = vmcs_fix_regval(encoding, val);

	if (!running)
		VMPTRLD(vmcs);

	error = vmwrite(encoding, val);

	if (!running)
		VMCLEAR(vmcs);

	return (error);
}

int
vmcs_setdesc(struct vmcs *vmcs, int running, int seg, struct seg_desc *desc)
{
	int error;
	uint32_t base, limit, access;

	error = vmcs_seg_desc_encoding(seg, &base, &limit, &access);
	if (error != 0)
```

```c
		panic("vmcs_setdesc: invalid segment register %d", seg);

	if (!running)
		VMPTRLD(vmcs);
	if ((error = vmwrite(base, desc->base)) != 0)
		goto done;

	if ((error = vmwrite(limit, desc->limit)) != 0)
		goto done;

	if (access != VMCS_INVALID_ENCODING) {
		if ((error = vmwrite(access, desc->access)) != 0)
			goto done;
	}
done:
	if (!running)
		VMCLEAR(vmcs);
	return (error);
}

int
vmcs_getdesc(struct vmcs *vmcs, int running, int seg, struct seg_desc *desc)
{
	int error;
	uint32_t base, limit, access;
	uint64_t u64;

	error = vmcs_seg_desc_encoding(seg, &base, &limit, &access);
	if (error != 0)
		panic("vmcs_getdesc: invalid segment register %d", seg);

	if (!running)
		VMPTRLD(vmcs);
	if ((error = vmread(base, &u64)) != 0)
		goto done;
	desc->base = u64;

	if ((error = vmread(limit, &u64)) != 0)
		goto done;
	desc->limit = u64;

	if (access != VMCS_INVALID_ENCODING) {
		if ((error = vmread(access, &u64)) != 0)
			goto done;
		desc->access = u64;
	}
done:
	if (!running)
		VMCLEAR(vmcs);
	return (error);
}

int
vmcs_set_msr_save(struct vmcs *vmcs, u_long g_area, u_int g_count)
{
	int error;

	VMPTRLD(vmcs);

	/*
	 * Guest MSRs are saved in the VM-exit MSR-store area.
	 * Guest MSRs are loaded from the VM-entry MSR-load area.
	 * Both areas point to the same location in memory.
	 */
	if ((error = vmwrite(VMCS_EXIT_MSR_STORE, g_area)) != 0)
		goto done;
	if ((error = vmwrite(VMCS_EXIT_MSR_STORE_COUNT, g_count)) != 0)
		goto done;

	if ((error = vmwrite(VMCS_ENTRY_MSR_LOAD, g_area)) != 0)
		goto done;
	if ((error = vmwrite(VMCS_ENTRY_MSR_LOAD_COUNT, g_count)) != 0)
		goto done;

	error = 0;
done:
	VMCLEAR(vmcs);
	return (error);
}

int
vmcs_init(struct vmcs *vmcs)
{
	int error, codesel, datasel, tsssel;
	u_long cr0, cr4, efer;
	uint64_t pat;
#ifdef	__FreeBSD__
	uint64_t fsbase, idtrbase;
#endif

	codesel = vmm_get_host_codesel();
	datasel = vmm_get_host_datasel();
	tsssel = vmm_get_host_tsssel();

	/*
	 * Make sure we have a "current" VMCS to work with.
	 */
	VMPTRLD(vmcs);

	/* Host state */

	/* Initialize host IA32_PAT MSR */
	pat = vmm_get_host_pat();
	if ((error = vmwrite(VMCS_HOST_IA32_PAT, pat)) != 0)
		goto done;

	/* Load the IA32_EFER MSR */
	efer = vmm_get_host_efer();
	if ((error = vmwrite(VMCS_HOST_IA32_EFER, efer)) != 0)
		goto done;

	/* Load the control registers */

	cr0 = vmm_get_host_cr0();
	if ((error = vmwrite(VMCS_HOST_CR0, cr0)) != 0)
		goto done;

	cr4 = vmm_get_host_cr4() | CR4_VMXE;
	if ((error = vmwrite(VMCS_HOST_CR4, cr4)) != 0)
		goto done;

	/* Load the segment selectors */
	if ((error = vmwrite(VMCS_HOST_ES_SELECTOR, datasel)) != 0)
		goto done;

	if ((error = vmwrite(VMCS_HOST_CS_SELECTOR, codesel)) != 0)
		goto done;

	if ((error = vmwrite(VMCS_HOST_SS_SELECTOR, datasel)) != 0)
		goto done;

	if ((error = vmwrite(VMCS_HOST_DS_SELECTOR, datasel)) != 0)
		goto done;

#ifdef	__FreeBSD__
	if ((error = vmwrite(VMCS_HOST_FS_SELECTOR, datasel)) != 0)
		goto done;

	if ((error = vmwrite(VMCS_HOST_GS_SELECTOR, datasel)) != 0)
		goto done;
#else
	if ((error = vmwrite(VMCS_HOST_FS_SELECTOR, vmm_get_host_fssel())) != 0)
		goto done;

	if ((error = vmwrite(VMCS_HOST_GS_SELECTOR, vmm_get_host_gssel())) != 0)
		goto done;
#endif

	if ((error = vmwrite(VMCS_HOST_TR_SELECTOR, tsssel)) != 0)
		goto done;

#ifdef	__FreeBSD__
	/*
	 * Load the Base-Address for %fs and idtr.
	 *
	 * Note that we exclude %gs, tss and gdtr here because their base
	 * address is pcpu specific.
	 */
	fsbase = vmm_get_host_fsbase();
	if ((error = vmwrite(VMCS_HOST_FS_BASE, fsbase)) != 0)
		goto done;

	idtrbase = vmm_get_host_idtrbase();
	if ((error = vmwrite(VMCS_HOST_IDTR_BASE, idtrbase)) != 0)
		goto done;

#else /* __FreeBSD__ */
	/*
	 * Configure host sysenter MSRs to be restored on VM exit.
	 * The thread-specific MSR_INTC_SEP_ESP value is loaded in vmx_run.
	 */
	if ((error = vmwrite(VMCS_HOST_IA32_SYSENTER_CS, KCS_SEL)) != 0)
		goto done;
	/* Natively defined as MSR_INTC_SEP_EIP */
	if ((error = vmwrite(VMCS_HOST_IA32_SYSENTER_EIP,
	    rdmsr(MSR_SYSENTER_EIP_MSR))) != 0)
		goto done;

#endif /* __FreeBSD__ */

	/* instruction pointer */
	if (no_flush_rsb) {
		if ((error = vmwrite(VMCS_HOST_RIP,
		    (u_long)vmx_exit_guest)) != 0)
			goto done;
	} else {
		if ((error = vmwrite(VMCS_HOST_RIP,
		    (u_long)vmx_exit_guest_flush_rsb)) != 0)
			goto done;
	}

	/* link pointer */
	if ((error = vmwrite(VMCS_LINK_POINTER, ~0)) != 0)
		goto done;
done:
	VMCLEAR(vmcs);
	return (error);
}

#ifdef DDB
extern int vmxon_enabled[];

DB_SHOW_COMMAND(vmcs, db_show_vmcs)
{
	uint64_t cur_vmcs, val;
	uint32_t exit;

	if (!vmxon_enabled[curcpu]) {
		db_printf("VMX not enabled\n");
		return;
	}

	if (have_addr) {
		db_printf("Only current VMCS supported\n");
		return;
	}

	vmptrst(&cur_vmcs);
	if (cur_vmcs == VMCS_INITIAL) {
		db_printf("No current VM context\n");
		return;
	}
	db_printf("VMCS: %jx\n", cur_vmcs);
	db_printf("VPID: %lu\n", vmcs_read(VMCS_VPID));
	db_printf("Activity: ");
	val = vmcs_read(VMCS_GUEST_ACTIVITY);
	switch (val) {
	case 0:
		db_printf("Active");
		break;
	case 1:
		db_printf("HLT");
		break;
	case 2:
		db_printf("Shutdown");
		break;
	case 3:
		db_printf("Wait for SIPI");
		break;
	default:
		db_printf("Unknown: %#lx", val);
	}
	db_printf("\n");
	exit = vmcs_read(VMCS_EXIT_REASON);
	if (exit & 0x80000000)
		db_printf("Entry Failure Reason: %u\n", exit & 0xffff);
	else
		db_printf("Exit Reason: %u\n", exit & 0xffff);
	db_printf("Qualification: %#lx\n", vmcs_exit_qualification());
	db_printf("Guest Linear Address: %#lx\n",
	    vmcs_read(VMCS_GUEST_LINEAR_ADDRESS));
	switch (exit & 0x8000ffff) {
	case EXIT_REASON_EXCEPTION:
	case EXIT_REASON_EXT_INTR:
		val = vmcs_read(VMCS_EXIT_INTR_INFO);
		db_printf("Interrupt Type: ");
		switch (val >> 8 & 0x7) {
		case 0:
			db_printf("external");
			break;
		case 2:
			db_printf("NMI");
			break;
		case 3:
			db_printf("HW exception");
			break;
		case 4:
			db_printf("SW exception");
			break;
		default:
			db_printf("?? %lu", val >> 8 & 0x7);
			break;
		}
		db_printf("  Vector: %lu", val & 0xff);
		if (val & 0x800)
			db_printf("  Error Code: %lx",
			    vmcs_read(VMCS_EXIT_INTR_ERRCODE));
		db_printf("\n");
		break;
	case EXIT_REASON_EPT_FAULT:
	case EXIT_REASON_EPT_MISCONFIG:
		db_printf("Guest Physical Address: %#lx\n",
		    vmcs_read(VMCS_GUEST_PHYSICAL_ADDRESS));
		break;
	}
	db_printf("VM-instruction error: %#lx\n", vmcs_instruction_error());
}
#endif
```