




🔍 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

...

```
1  /*-
2   * SPDX-License-Identifier: BSD-2-Clause-FreeBSD
3   *
4   * Copyright (c) 2011 NetApp, Inc.
5   * All rights reserved.
6   *
7   * Redistribution and use in source and binary forms, with or without
8   * modification, are permitted provided that the following conditions
9   * are met:
10  * 1. Redistributions of source code must retain the above copyright
11  *   notice, this list of conditions and the following disclaimer.
12  * 2. Redistributions in binary form must reproduce the above copyright
13  *   notice, this list of conditions and the following disclaimer in the
14  *   documentation and/or other materials provided with the distribution.
15  *
16  * THIS SOFTWARE IS PROVIDED BY NETAPP, INC ``AS IS'' AND
17  * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
18  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
19  * ARE DISCLAIMED. IN NO EVENT SHALL NETAPP, INC OR CONTRIBUTORS BE LIABLE
20  * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
21  * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
22  * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
23  * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
24  * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
25  * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
26  * SUCH DAMAGE.
27  *
28  * $FreeBSD$
29  */
30  /*
31  * This file and its contents are supplied under the terms of the
32  * Common Development and Distribution License ("CDDL"), version 1.0.
33  * You may only use this file in accordance with the terms of version
34  * 1.0 of the CDDL.
35  *
36  * A full copy of the text of the CDDL should have accompanied this
37  * source. A copy of the CDDL is also available via the Internet at
38  * http://www.illumos.org/license/CDDL.
39  *
40  * Copyright 2014 Pluribus Networks Inc.
41  * Copyright 2017 Joyent, Inc.
42  */
43
44  #ifdef __FreeBSD__
45  #include "opt_ddb.h"
46  #endif
47
48  #include <sys/cdefs.h>
49  __FBSDID("$FreeBSD$");
50
51  #include <sys/param.h>
52  #include <sys/sysctl.h>
53  #include <sys/system.h>
54  #include <sys/pcpu.h>
55
56  #include <vm/vm.h>
57  #include <vm/pmap.h>
58
59  #include <machine/segments.h>
60  #include <machine/vmm.h>
61  #include "vmm_host.h"
62  #include "vmx_cpufunc.h"
63  #include "vmcs.h"
64  #include "ept.h"
65  #include "vmx.h"
66
67  #ifdef DDB
68  #include <ddb/ddb.h>
69  #endif
70
71  SYSCTL_DECL(_hw_vmm_vmx);
72
73  static int no_flush_rsb;
74  SYSCTL_INT(_hw_vmm_vmx, OID_AUTO, no_flush_rsb, CTLFLAG_RW,
75  &no_flush_rsb, 0, "Do not flush RSB upon vmexit");
76
77  static uint64_t
78  vmcs_fix_regval(uint32_t encoding, uint64_t val)
```

```

79 {
80
81     switch (encoding) {
82     case VMCS_GUEST_CR0:
83         val = vmx_fix_cr0(val);
84         break;
85     case VMCS_GUEST_CR4:
86         val = vmx_fix_cr4(val);
87         break;
88     default:
89         break;
90     }
91     return (val);
92 }
93
94 static uint32_t
95 vmcs_field_encoding(int ident)
96 {
97     switch (ident) {
98     case VM_REG_GUEST_CR0:
99         return (VMCS_GUEST_CR0);
100    case VM_REG_GUEST_CR3:
101        return (VMCS_GUEST_CR3);
102    case VM_REG_GUEST_CR4:
103        return (VMCS_GUEST_CR4);
104    case VM_REG_GUEST_DR7:
105        return (VMCS_GUEST_DR7);
106    case VM_REG_GUEST_RSP:
107        return (VMCS_GUEST_RSP);
108    case VM_REG_GUEST_RIP:
109        return (VMCS_GUEST_RIP);
110    case VM_REG_GUEST_RFLAGS:
111        return (VMCS_GUEST_RFLAGS);
112    case VM_REG_GUEST_ES:
113        return (VMCS_GUEST_ES_SELECTOR);
114    case VM_REG_GUEST_CS:
115        return (VMCS_GUEST_CS_SELECTOR);
116    case VM_REG_GUEST_SS:
117        return (VMCS_GUEST_SS_SELECTOR);
118    case VM_REG_GUEST_DS:
119        return (VMCS_GUEST_DS_SELECTOR);
120    case VM_REG_GUEST_FS:
121        return (VMCS_GUEST_FS_SELECTOR);
122    case VM_REG_GUEST_GS:
123        return (VMCS_GUEST_GS_SELECTOR);
124    case VM_REG_GUEST_TR:
125        return (VMCS_GUEST_TR_SELECTOR);
126    case VM_REG_GUEST_LDTR:
127        return (VMCS_GUEST_LDTR_SELECTOR);
128    case VM_REG_GUEST_EFER:
129        return (VMCS_GUEST_IA32_EFER);
130    case VM_REG_GUEST_PDPTE0:
131        return (VMCS_GUEST_PDPTE0);
132    case VM_REG_GUEST_PDPTE1:
133        return (VMCS_GUEST_PDPTE1);
134    case VM_REG_GUEST_PDPTE2:
135        return (VMCS_GUEST_PDPTE2);
136    case VM_REG_GUEST_PDPTE3:
137        return (VMCS_GUEST_PDPTE3);
138    case VM_REG_GUEST_ENTRY_INST_LENGTH:
139        return (VMCS_ENTRY_INST_LENGTH);
140    default:
141        return (-1);
142    }
143 }
144 }
145
146 static int
147 vmcs_seg_desc_encoding(int seg, uint32_t *base, uint32_t *lim, uint32_t *acc)
148 {
149
150     switch (seg) {
151     case VM_REG_GUEST_ES:
152         *base = VMCS_GUEST_ES_BASE;
153         *lim = VMCS_GUEST_ES_LIMIT;
154         *acc = VMCS_GUEST_ES_ACCESS_RIGHTS;
155         break;
156     case VM_REG_GUEST_CS:
157         *base = VMCS_GUEST_CS_BASE;
158         *lim = VMCS_GUEST_CS_LIMIT;
159         *acc = VMCS_GUEST_CS_ACCESS_RIGHTS;
160         break;
161     case VM_REG_GUEST_SS:
162         *base = VMCS_GUEST_SS_BASE;
163         *lim = VMCS_GUEST_SS_LIMIT;
164         *acc = VMCS_GUEST_SS_ACCESS_RIGHTS;
165         break;
166     case VM_REG_GUEST_DS:
167         *base = VMCS_GUEST_DS_BASE;
168         *lim = VMCS_GUEST_DS_LIMIT;
169         *acc = VMCS_GUEST_DS_ACCESS_RIGHTS;
170         break;
171     case VM_REG_GUEST_FS:
172         *base = VMCS_GUEST_FS_BASE;
173         *lim = VMCS_GUEST_FS_LIMIT;
174         *acc = VMCS_GUEST_FS_ACCESS_RIGHTS;
175         break;
176     case VM_REG_GUEST_GS:

```

```

177         *base = VMCS_GUEST_GS_BASE;
178         *lim = VMCS_GUEST_GS_LIMIT;
179         *acc = VMCS_GUEST_GS_ACCESS_RIGHTS;
180         break;
181     case VM_REG_GUEST_TR:
182         *base = VMCS_GUEST_TR_BASE;
183         *lim = VMCS_GUEST_TR_LIMIT;
184         *acc = VMCS_GUEST_TR_ACCESS_RIGHTS;
185         break;
186     case VM_REG_GUEST_LDTR:
187         *base = VMCS_GUEST_LDTR_BASE;
188         *lim = VMCS_GUEST_LDTR_LIMIT;
189         *acc = VMCS_GUEST_LDTR_ACCESS_RIGHTS;
190         break;
191     case VM_REG_GUEST_IDTR:
192         *base = VMCS_GUEST_IDTR_BASE;
193         *lim = VMCS_GUEST_IDTR_LIMIT;
194         *acc = VMCS_INVALID_ENCODING;
195         break;
196     case VM_REG_GUEST_GDTR:
197         *base = VMCS_GUEST_GDTR_BASE;
198         *lim = VMCS_GUEST_GDTR_LIMIT;
199         *acc = VMCS_INVALID_ENCODING;
200         break;
201     default:
202         return (EINVAL);
203     }
204
205     return (0);
206 }
207
208 int
209 vmcs_getreg(struct vmcs *vmcs, int running, int ident, uint64_t *retval)
210 {
211     int error;
212     uint32_t encoding;
213
214     /*
215      * If we need to get at vmx-specific state in the VMCS we can bypass
216      * the translation of 'ident' to 'encoding' by simply setting the
217      * sign bit. As it so happens the upper 16 bits are reserved (i.e
218      * set to 0) in the encodings for the VMCS so we are free to use the
219      * sign bit.
220      */
221     if (ident < 0)
222         encoding = ident & 0x7fffffff;
223     else
224         encoding = vmcs_field_encoding(ident);
225
226     if (encoding == (uint32_t)-1)
227         return (EINVAL);
228
229     if (!running)
230         VMPTRLD(vmcs);
231
232     error = vmread(encoding, retval);
233
234     if (!running)
235         VMCLEAR(vmcs);
236
237     return (error);
238 }
239
240 int
241 vmcs_setreg(struct vmcs *vmcs, int running, int ident, uint64_t val)
242 {
243     int error;
244     uint32_t encoding;
245
246     if (ident < 0)
247         encoding = ident & 0x7fffffff;
248     else
249         encoding = vmcs_field_encoding(ident);
250
251     if (encoding == (uint32_t)-1)
252         return (EINVAL);
253
254     val = vmcs_fix_regval(encoding, val);
255
256     if (!running)
257         VMPTRLD(vmcs);
258
259     error = vmwrite(encoding, val);
260
261     if (!running)
262         VMCLEAR(vmcs);
263
264     return (error);
265 }
266
267 int
268 vmcs_setdesc(struct vmcs *vmcs, int running, int seg, struct seg_desc *desc)
269 {
270     int error;
271     uint32_t base, limit, access;
272
273     error = vmcs_seg_desc_encoding(seg, &base, &limit, &access);
274     if (error != 0)

```

```

275         panic("vmcs_setdesc: invalid segment register %d", seg);
276
277     if (!running)
278         VMPTRLD(vmcs);
279     if ((error = vmwrite(base, desc->base)) != 0)
280         goto done;
281
282     if ((error = vmwrite(limit, desc->limit)) != 0)
283         goto done;
284
285     if (access != VMCS_INVALID_ENCODING) {
286         if ((error = vmwrite(access, desc->access)) != 0)
287             goto done;
288     }
289 done:
290     if (!running)
291         VMCLEAR(vmcs);
292     return (error);
293 }
294
295 int
296 vmcs_getdesc(struct vmcs *vmcs, int running, int seg, struct seg_desc *desc)
297 {
298     int error;
299     uint32_t base, limit, access;
300     uint64_t u64;
301
302     error = vmcs_seg_desc_encoding(seg, &base, &limit, &access);
303     if (error != 0)
304         panic("vmcs_getdesc: invalid segment register %d", seg);
305
306     if (!running)
307         VMPTRLD(vmcs);
308     if ((error = vmread(base, &u64)) != 0)
309         goto done;
310     desc->base = u64;
311
312     if ((error = vmread(limit, &u64)) != 0)
313         goto done;
314     desc->limit = u64;
315
316     if (access != VMCS_INVALID_ENCODING) {
317         if ((error = vmread(access, &u64)) != 0)
318             goto done;
319         desc->access = u64;
320     }
321 done:
322     if (!running)
323         VMCLEAR(vmcs);
324     return (error);
325 }
326
327 int
328 vmcs_set_msr_save(struct vmcs *vmcs, u_long g_area, u_int g_count)
329 {
330     int error;
331
332     VMPTRLD(vmcs);
333
334     /*
335      * Guest MSRs are saved in the VM-exit MSR-store area.
336      * Guest MSRs are loaded from the VM-entry MSR-load area.
337      * Both areas point to the same location in memory.
338      */
339     if ((error = vmwrite(VMCS_EXIT_MSR_STORE, g_area)) != 0)
340         goto done;
341     if ((error = vmwrite(VMCS_EXIT_MSR_STORE_COUNT, g_count)) != 0)
342         goto done;
343
344     if ((error = vmwrite(VMCS_ENTRY_MSR_LOAD, g_area)) != 0)
345         goto done;
346     if ((error = vmwrite(VMCS_ENTRY_MSR_LOAD_COUNT, g_count)) != 0)
347         goto done;
348
349     error = 0;
350 done:
351     VMCLEAR(vmcs);
352     return (error);
353 }
354
355 int
356 vmcs_init(struct vmcs *vmcs)
357 {
358     int error, codesel, datasel, tsssel;
359     u_long cr0, cr4, efer;
360     uint64_t pat;
361 #ifdef __FreeBSD__
362     uint64_t fsbase, idtrbase;
363 #endif
364
365     codesel = vmm_get_host_codesel();
366     datasel = vmm_get_host_datasel();
367     tsssel = vmm_get_host_tsssel();
368
369     /*
370      * Make sure we have a "current" VMCS to work with.
371      */
372     VMPTRLD(vmcs);

```

```

373
374     /* Host state */
375
376     /* Initialize host IA32_PAT MSR */
377     pat = vmm_get_host_pat();
378     if ((error = vmwrite(VMCS_HOST_IA32_PAT, pat)) != 0)
379         goto done;
380
381     /* Load the IA32_EFER MSR */
382     efer = vmm_get_host_efer();
383     if ((error = vmwrite(VMCS_HOST_IA32_EFER, efer)) != 0)
384         goto done;
385
386     /* Load the control registers */
387
388     cr0 = vmm_get_host_cr0();
389     if ((error = vmwrite(VMCS_HOST_CR0, cr0)) != 0)
390         goto done;
391
392     cr4 = vmm_get_host_cr4() | CR4_VMXE;
393     if ((error = vmwrite(VMCS_HOST_CR4, cr4)) != 0)
394         goto done;
395
396     /* Load the segment selectors */
397     if ((error = vmwrite(VMCS_HOST_ES_SELECTOR, dataset1)) != 0)
398         goto done;
399
400     if ((error = vmwrite(VMCS_HOST_CS_SELECTOR, codesel)) != 0)
401         goto done;
402
403     if ((error = vmwrite(VMCS_HOST_SS_SELECTOR, dataset1)) != 0)
404         goto done;
405
406     if ((error = vmwrite(VMCS_HOST_DS_SELECTOR, dataset1)) != 0)
407         goto done;
408
409 #ifdef __FreeBSD__
410     if ((error = vmwrite(VMCS_HOST_FS_SELECTOR, dataset1)) != 0)
411         goto done;
412
413     if ((error = vmwrite(VMCS_HOST_GS_SELECTOR, dataset1)) != 0)
414         goto done;
415 #else
416     if ((error = vmwrite(VMCS_HOST_FS_SELECTOR, vmm_get_host_fsbase())) != 0)
417         goto done;
418
419     if ((error = vmwrite(VMCS_HOST_GS_SELECTOR, vmm_get_host_gssbase())) != 0)
420         goto done;
421 #endif
422
423     if ((error = vmwrite(VMCS_HOST_TR_SELECTOR, tssbase)) != 0)
424         goto done;
425
426 #ifdef __FreeBSD__
427     /*
428      * Load the Base-Address for %fs and idtr.
429      *
430      * Note that we exclude %gs, tss and gdtr here because their base
431      * address is pcpu specific.
432      */
433     fsbase = vmm_get_host_fsbase();
434     if ((error = vmwrite(VMCS_HOST_FS_BASE, fsbase)) != 0)
435         goto done;
436
437     idtrbase = vmm_get_host_idtrbase();
438     if ((error = vmwrite(VMCS_HOST_IDTR_BASE, idtrbase)) != 0)
439         goto done;
440
441 #else /* __FreeBSD__ */
442     /*
443      * Configure host sysenter MSRs to be restored on VM exit.
444      * The thread-specific MSR_INT32_SEP_ESP value is loaded in vmx_run.
445      */
446     if ((error = vmwrite(VMCS_HOST_IA32_SYSENTER_CS, KCS_SEL)) != 0)
447         goto done;
448
449     /* Natively defined as MSR_INT32_SEP_EIP */
450     if ((error = vmwrite(VMCS_HOST_IA32_SYSENTER_EIP,
451         rdmsr(MSR_SYSENTER_EIP_MSR))) != 0)
452         goto done;
453 #endif /* __FreeBSD__ */
454
455     /* instruction pointer */
456     if (no_flush_rsb) {
457         if ((error = vmwrite(VMCS_HOST_RIP,
458             (u_long)vmx_exit_guest)) != 0)
459             goto done;
460     } else {
461         if ((error = vmwrite(VMCS_HOST_RIP,
462             (u_long)vmx_exit_guest_flush_rsb)) != 0)
463             goto done;
464     }
465
466     /* link pointer */
467     if ((error = vmwrite(VMCS_LINK_POINTER, ~0)) != 0)
468         goto done;
469 done:
470     VMCLEAR(vmc);

```

```

471         return (error);
472     }
473
474 #ifdef DOB
475 extern int vmxon_enabled[];
476
477 DB_SHOW_COMMAND(vmcs, db_show_vmcs)
478 {
479     uint64_t cur_vmcs, val;
480     uint32_t exit;
481
482     if (!vmxon_enabled[curcpu]) {
483         db_printf("VMX not enabled\n");
484         return;
485     }
486
487     if (have_addr) {
488         db_printf("Only current VMCS supported\n");
489         return;
490     }
491
492     vmprst(&cur_vmcs);
493     if (cur_vmcs == VMCS_INITIAL) {
494         db_printf("No current VM context\n");
495         return;
496     }
497     db_printf("VMCS: %jx\n", cur_vmcs);
498     db_printf("VPID: %lu\n", vmcs_read(VMCS_VPID));
499     db_printf("Activity: ");
500     val = vmcs_read(VMCS_GUEST_ACTIVITY);
501     switch (val) {
502     case 0:
503         db_printf("Active");
504         break;
505     case 1:
506         db_printf("HLT");
507         break;
508     case 2:
509         db_printf("Shutdown");
510         break;
511     case 3:
512         db_printf("Wait for SIPI");
513         break;
514     default:
515         db_printf("Unknown: %lx", val);
516     }
517     db_printf("\n");
518     exit = vmcs_read(VMCS_EXIT_REASON);
519     if (exit & 0x80000000)
520         db_printf("Entry Failure Reason: %u\n", exit & 0xffff);
521     else
522         db_printf("Exit Reason: %u\n", exit & 0xffff);
523     db_printf("Qualification: %lx\n", vmcs_exit_qualification());
524     db_printf("Guest Linear Address: %lx\n",
525             vmcs_read(VMCS_GUEST_LINEAR_ADDRESS));
526     switch (exit & 0x8000ffff) {
527     case EXIT_REASON_EXCEPTION:
528     case EXIT_REASON_EXT_INTR:
529         val = vmcs_read(VMCS_EXIT_INTR_INFO);
530         db_printf("Interrupt Type: ");
531         switch (val >> 8 & 0x7) {
532         case 0:
533             db_printf("external");
534             break;
535         case 2:
536             db_printf("NMI");
537             break;
538         case 3:
539             db_printf("HW exception");
540             break;
541         case 4:
542             db_printf("SW exception");
543             break;
544         default:
545             db_printf("?? %lu", val >> 8 & 0x7);
546             break;
547         }
548         db_printf(" Vector: %lu", val & 0xff);
549         if (val & 0x800)
550             db_printf(" Error Code: %lx",
551                     vmcs_read(VMCS_EXIT_INTR_ERRCODE));
552         db_printf("\n");
553         break;
554     case EXIT_REASON_EPT_FAULT:
555     case EXIT_REASON_EPT_MISCONFIG:
556         db_printf("Guest Physical Address: %lx\n",
557                 vmcs_read(VMCS_GUEST_PHYSICAL_ADDRESS));
558         break;
559     }
560     db_printf("VM-instruction error: %lx\n", vmcs_instruction_error());
561 }
562 #endif

```