ハイパーバイザの作り方〜ちゃんと理解する仮想化技術〜 第９回 Intel VT-x を用いたハイパーバイザの実装その４「vmm.ko への VMExit」

はじめに

前回は、vmm.ko が VM_RUN ioctl を受け取ってから VMEntry するまでの処理を解説しました。 今回は VMX non root mode から vmm.ko へ VMExit してきたときの処理を解説します。

解説対象のソースコードについて

本連載では、FreeBSD-CURRENT に実装されている BHyVe のソースコードを解説しています。

このソースコードは、FreeBSD の Subversion リポジトリから取得できます。リビジョンは r245673 を用いています。お手持ちの PC に Subversion をインストールし、次のようなコマンドでソースコードを取得してください。

svn co -r245673 svn://svn.freebsd.org/base/head src

/usr/sbin/bhyve による仮想 CPU の実行処理のおさらい

/usr/sbin/bhyve は仮想 CPU の数だけスレッドを起動し、それぞれのスレッドが/dev/vmm/${name}に対して VM_RUN ioctl を発行します (図 1)。 vmm.ko は ioctl を受けて CPU を VMX non root mode へ切り替えゲスト OS を実行します (これが VMEntry です)。

VMX non root mode でハイパーバイザの介入が必要な何らかのイベントが発生すると制御が vmm.ko へ戻され、イベントがトラップされます (これが VMExit です)。

イベントの種類が/usr/sbin/bhyve でハンドルされる必要のあるものだった場合、ioctl はリターンされ、制御が/usr/sbin/bhyve へ移ります。イベントの種類が/usr/sbin/bhyve でハンドルされる必要のないものだった場合、ioctl はリターンされないままゲスト CPU の実行が再開されます。

今回は、VMX non root mode から vmm.ko へ VMExit してきたときの処理を見ていきます。

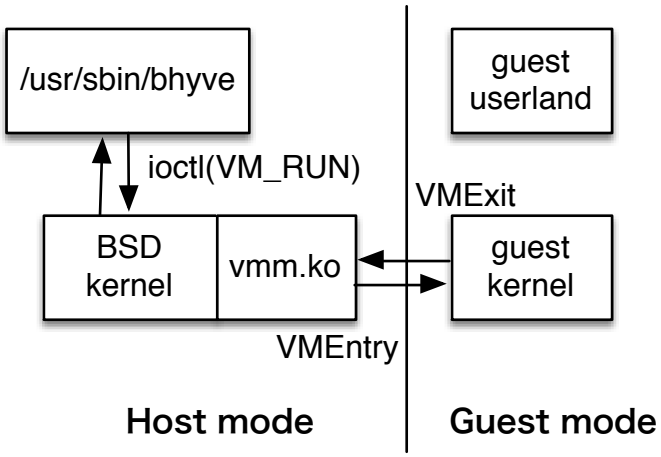

図 1　VM_RUN ioctl による仮想 CPU の実行イメージ

vmm.ko が VM_RUN ioctl を受け取ってから VMEntry するまで

vmm.ko が VM_RUN ioctl を受け取ってから VMEntry するまでの処理について、順を追って見ていきます。今回は、I/O 命令で VMExit したと仮定します。

前回解説のとおり、VMExit 時の VMX root mode の再開アドレス (RIP[*1]) は VMCS の HOST_RIP で指定された vmx_longjmp に設定されています。vmx_longjmp は vmx_setjmp と対になっている関数で、POSIX API の setjmp/longjmp に近い動作を行います。つまり、vmx_longjmp は vmx_setjmp が呼ばれた直後のアドレスへジャンプします。結果として、VMExit されると vmx_longjmp を経由し vmx_run の while ループへ戻ってくることになります。

また、vmx_setjmp はどこから return してきたかを示すために戻り値を用いています。ここでは vmx_longjmp から戻ってきたことを表す VMX_RETURN_LONGJMP を返します。

では、以上のことをふまえてソースコードの詳細を見ていきましょう。リスト 1、リスト 2、リスト 3 に示します。解説キャプションの番号は、注目すべき処理の順番を示します。

sys/amd64/vmm/intel/vmx.c

intel/ディレクトリには Intel VT-x に依存したコード群が置かれています。今回はゲストマシン実行ループの中心となる vmx_run と、VMExit のハンドラ関数である vmx_exit_process を解説します。

[*1] すでに前回までの記事でも「RIP」と表記していますが、なんのことだろうと思った方もいらっしゃるかもしれません。
これは、x86_64 アーキテクチャでの 64bit 幅の EIP レジスタ (インストラクションポインタ) の名前です。
ほかにも EAX、EBX レジスタが RAX、RBX のような名前になっています。

リスト 1 sys/amd64/vmm/intel/vmx.c

```
.....(省略).....
1197:  static int
1198:  vmx_exit_process(struct vmx *vmx, int vcpu, struct vm_exit *vmexit)
1199:  {
1200:   int error, handled;
1201:   struct vmcs *vmcs;
1202:   struct vmxctx *vmxctx;
1203:   uint32_t eax, ecx, edx;
1204:   uint64_t qual, gla, gpa, cr3, intr_info;
1205:
1206:   handled = 0;
1207:   vmcs = &vmx->vmcs[vcpu];
1208:   vmxctx = &vmx->ctx[vcpu];
1209:   qual = vmexit->u.vmx.exit_qualification;                    (20)
1210:   vmexit->exitcode = VM_EXITCODE_BOGUS;
1211:
1212:   switch (vmexit->u.vmx.exit_reason) {
.....(省略).....
1286:   case EXIT_REASON_INOUT:                                     (21)
1287:       vmexit->exitcode = VM_EXITCODE_INOUT;                   (22)
1288:       vmexit->u.inout.bytes = (qual & 0x7) + 1;               (23)
1289:       vmexit->u.inout.in = (qual & 0x8) ? 1 : 0;              (24)
1290:       vmexit->u.inout.string = (qual & 0x10) ? 1 : 0;         (25)
1291:       vmexit->u.inout.rep = (qual & 0x20) ? 1 : 0;            (26)
1292:       vmexit->u.inout.port = (uint16_t)(qual >> 16);          (27)
1293:       vmexit->u.inout.eax = (uint32_t)(vmxctx->guest_rax);    (28)
1294:       break;
.....(省略).....
1310:   default:
1311:       break;
1312:   }
.....(省略).....
1351:   return (handled);                                          (29)
1352:  }
1353:
1354:  static int
1355:  vmx_run(void *arg, int vcpu, register_t rip)
```

```
1356:  {
......(省略).....
1394:   do {
......(省略).....
1398:       rc = vmx_setjmp(vmxctx);                                    (13)
......(省略).....
1402:       switch (rc) {
......(省略).....
1411:       case VMX_RETURN_LONGJMP:                                    (14)
1412:           break;          /* vm exit */
......(省略).....
1436:       }
1437:
1438:       /* enable interrupts */
1439:       enable_intr();
1440:
1441:       /* collect some basic information for VM exit processing */
1442:       vmexit->rip = rip = vmcs_guest_rip();                       (15)
1443:       vmexit->inst_length = vmexit_instruction_length();          (16)
1444:       vmexit->u.vmx.exit_reason = exit_reason = vmcs_exit_reason();   (17)
1445:       vmexit->u.vmx.exit_qualification = vmcs_exit_qualification();   (18)
......(省略).....
1455:       handled = vmx_exit_process(vmx, vcpu, vmexit);              (19)
1456:       vmx_exit_trace(vmx, vcpu, rip, exit_reason, handled);
1457:
1458:   } while (handled);                                              (30)
......(省略).....
1481:   return (0);
......(省略).....
1490:  }
```

- (20) Exit Qualification を取り出し。
- (21) IO 命令で VMExit した場合、Exit Reason 30（EXIT_REASON_INOUT）となる。
- (22) Exit Reason を代入。
- (23) Exit Qualification からアクセス幅を代入。
- (24) Exit Qualification からアクセス方向を代入。
- (25) Exit Qualification から String 命令かどうかのフラグを代入。
- (26) Exit Qualification から rep prefix 付きかどうかのフラグを代入。
- (27) Exit Qualification からポート番号を代入。

- (28) rax レジスタの値を代入。
- (29) EXIT_REASON_INOUT では、ユーザランドでのエミュレーション処理を要求するため handled = 0 を返す。
- (13) vmx_return からここへリターンされてくる。
  返り値として VMX_RETURN_LONGJMP を返す。
- (14) rc は VMX_RETURN_LONGJMP。
- (15) VMCS からゲスト OS の RIP を取得して vm_exit 構造体にセット。
- (16) VMCS から RIP が指している命令の命令長を取得して vm_exit 構造体にセット。
- (17) VMCS から Exit reason を取得して vm_exit 構造体にセット。
- (18) VMCS から Exit qualification を取得して vm_exit 構造体にセット。
- (19) vmx_exit_process で Exit reason に応じた処理を実行。
- (30) handled = 0 が返ったため、ループを抜け vmx_run から抜ける。

sys/amd64/vmm/intel/vmx_support.S

vmx_support.S は C 言語で記述できない、コンテキストの退避/復帰や VT-x 拡張命令の発行などのコードを提供しています。今回は、vmx_setjmp・vmx_longjmp を解説します。

リスト 2 sys/amd64/vmm/intel/vmx_support.S

```
......(省略).....
 100:  /*
 101:   * int vmx_setjmp(ctxp)
 102:   * %rdi = ctxp
 103:   *
 104:   * Return value is '0' when it returns directly from here.
 105:   * Return value is '1' when it returns after a vm exit through vmx_longjmp.
 106:   */
 107:  ENTRY(vmx_setjmp)
 108:   movq    (%rsp),%rax         /* return address */            (1)
 109:   movq    %r15,VMXCTX_HOST_R15(%rdi)                          (2)
 110:   movq    %r14,VMXCTX_HOST_R14(%rdi)
 111:   movq    %r13,VMXCTX_HOST_R13(%rdi)
 112:   movq    %r12,VMXCTX_HOST_R12(%rdi)
 113:   movq    %rbp,VMXCTX_HOST_RBP(%rdi)
 114:   movq    %rsp,VMXCTX_HOST_RSP(%rdi)
 115:   movq    %rbx,VMXCTX_HOST_RBX(%rdi)
 116:   movq    %rax,VMXCTX_HOST_RIP(%rdi)                          (3)
 117:
```

```
118:   /*
119:    * XXX save host debug registers
120:    */
121:   movl    $VMX_RETURN_DIRECT,%eax
122:   ret
123:  END(vmx_setjmp)
124:
125:  /*
126:   * void vmx_return(struct vmxctx *ctxp, int retval)
127:   * %rdi = ctxp
128:   * %rsi = retval
129:   * Return to vmm context through vmx_setjmp() with a value of 'retval'.
130:   */
131:  ENTRY(vmx_return)
132:   /* Restore host context. */
133:   movq    VMXCTX_HOST_R15(%rdi),%r15                          (8)
134:   movq    VMXCTX_HOST_R14(%rdi),%r14
135:   movq    VMXCTX_HOST_R13(%rdi),%r13
136:   movq    VMXCTX_HOST_R12(%rdi),%r12
137:   movq    VMXCTX_HOST_RBP(%rdi),%rbp
138:   movq    VMXCTX_HOST_RSP(%rdi),%rsp
139:   movq    VMXCTX_HOST_RBX(%rdi),%rbx
140:   movq    VMXCTX_HOST_RIP(%rdi),%rax                          (10)
141:   movq    %rax,(%rsp)         /* return address */            (11)
142:
143:   /*
144:    * XXX restore host debug registers
145:    */
146:   movl    %esi,%eax
147:   ret                                                         (12)
148:  END(vmx_return)
149:
150:  /*
151:   * void vmx_longjmp(void)
152:   * %rsp points to the struct vmxctx
153:   */
154:  ENTRY(vmx_longjmp)                                          (4)
155:   /*
156:    * Save guest state that is not automatically saved in the vmcs.
157:    */
```

```
158:   movq    %rdi,VMXCTX_GUEST_RDI(%rsp)                    (5)
159:   movq    %rsi,VMXCTX_GUEST_RSI(%rsp)
160:   movq    %rdx,VMXCTX_GUEST_RDX(%rsp)
161:   movq    %rcx,VMXCTX_GUEST_RCX(%rsp)
162:   movq    %r8,VMXCTX_GUEST_R8(%rsp)
163:   movq    %r9,VMXCTX_GUEST_R9(%rsp)
164:   movq    %rax,VMXCTX_GUEST_RAX(%rsp)
165:   movq    %rbx,VMXCTX_GUEST_RBX(%rsp)
166:   movq    %rbp,VMXCTX_GUEST_RBP(%rsp)
167:   movq    %r10,VMXCTX_GUEST_R10(%rsp)
168:   movq    %r11,VMXCTX_GUEST_R11(%rsp)
169:   movq    %r12,VMXCTX_GUEST_R12(%rsp)
170:   movq    %r13,VMXCTX_GUEST_R13(%rsp)
171:   movq    %r14,VMXCTX_GUEST_R14(%rsp)
172:   movq    %r15,VMXCTX_GUEST_R15(%rsp)
173:
174:   movq    %cr2,%rdi
175:   movq    %rdi,VMXCTX_GUEST_CR2(%rsp)
176:
177:   movq    %rsp,%rdi
178:   movq    $VMX_RETURN_LONGJMP,%rsi                       (6)
179:
180:   addq    $VMXCTX_TMPSTKTOP,%rsp
181:   callq   vmx_return                                     (7)
182:  END(vmx_longjmp)
```

- (1) スタック上のリターンアドレスを %rax に取り出す。
- (2) 以下の行では、VMEntry 時に VMCS へ自動保存されないホスト OS のレジスタを vmxctx 構造体へ保存している。
- (3) %rax に取り出したリターンアドレスを vmxctx 構造体の host_rip メンバに保存。
  ここまでが VMExit 前に行われている処理。
- (8) 以下の行では、VMExit 時に VMCS から自動復帰されなかったホスト OS のレジスタを復帰している。
- (10) vmxctx 構造体の host_rip メンバから %rax へリターンアドレスをコピー。
- (11) リターンアドレスをスタックにセット。
- (12) 11 でセットしたアドレスへリターン。
- (4) VMExit 時にはここから実行が再開される。
  以降の行で参照されている %rsp は VMEntry 時に自動保存され、VMExit 時に自動復帰されている。
- (5) 以下の行では、VMExit 時に VMCS へ自動保存されなかったゲスト OS のレジスタを保存して

いる。

- (6) 返り値として VMX_RETURN_LONGJMP を指定。
- (7) vmx_return を呼び出してホスト OS のレジスタを復帰する。

sys/amd64/vmm/vmm.c

vmm.c は、Intel VT-x と AMD-V の 2 つの異なるハードウェア仮想化支援機能のラッパー関数を提供しています。今回は vmx_run のラッパー関数の vm_run を解説します。

リスト 3 sys/amd64/vmm/vmm.c

```
......(省略)......
 672:  int
 673:  vm_run(struct vm *vm, struct vm_run *vmrun)
 674:  {
......(省略)......
 681:   vcpuid = vmrun->cpuid;
......(省略)......
 686:   vcpu = &vm->vcpu[vcpuid];
 687:   vme = &vmrun->vm_exit;
 688:   rip = vmrun->rip;
......(省略)......
 701:   error = VMRUN(vm->cookie, vcpuid, rip);                    (31)
......(省略)......
 709:   /* copy the exit information */
 710:   bcopy(&vcpu->exitinfo, vme, sizeof(struct vm_exit));       (32)
......(省略)......
 757:  }
```

- (31) vmx_run は EXIT_REASON_INOUT をハンドルしてここへ抜けてくる。
- (32) vm_exit 構造体の値はユーザランドへの返り値としてここでコピーされる。

まとめ

VMX non root mode から vmm.ko へ VMExit してきたときの処理について、ソースコードを解説しました。次回は、I/O 命令による VMExit を受けて行われるユーザランドでのエミュレーション処理について見ていきます。

ライセンス