# 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１０回 Intel VT-x を用いたハイパーバイザの実装その５「ユーザランドでの I/O エミュレーション」

## はじめに

前回は、VMX non root mode から vmm.ko へ VMExit してきたときの処理を解説しました。今回は 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 (VMEntry)。

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 (VMExit)。

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。/usr/sbin/bhyve はイベントの種類やレジスタの値などを参照し、デバイスエミュレーションなどの処理を行います。

今回は、この/usr/sbin/bhyve でのデバイスエミュレーション処理の部分を見ていきます。

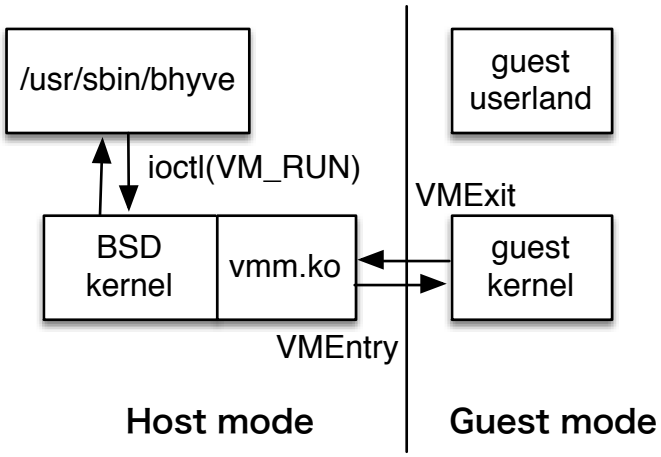


図 1　VM_RUN ioctl による仮想 CPU の実行イメージ

## /usr/sbin/bhyve での I/O 命令ハンドリング

前回の記事に引き続き、I/O 命令で VMExit した場合について見ていきます。VMExit に関する情報は VM_RUN ioctl の引数である struct vm_run の vm_exit メンバ (struct vm_exit) に書き込まれ、ioctl return 時にユーザランドへコピーされます。/usr/sbin/bhyve はこれを受け取り、vmexit->exitcode を参照してどのような VMExit 要因だったか判定し、VMExit 要因ごとの処理を呼び出します。I/O 命令で VMExit した場合の exitcode は VM_EXIT_INOUT です。

VM_EXIT_INOUT の場合、I/O の命令のエミュレーションに必要な情報 (ポート番号、アクセス幅、書き込み値 (読み込み時は不要)、I/O 方向 (in/out)) が struct vm_exit を介して vmm.ko から渡されます。

/usr/sbin/bhyve はこの値を I/O ポートエミュレーションハンドラに渡し、I/O ポート番号からどのデバイスへのアクセスなのかを判定し、デバイスのハンドラを呼び出します。

ハンドラの実行が終わったら、/usr/sbin/bhyve はふたたび VM_RUN ioctl を発行して、ゲストマシンの実行を再開します。

では、以上のことを踏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、リスト 4 にソースコードを示します。キャプションの丸数字で読む順番を示しています。

### vmmapi.c と bhyverun.c の解説

libvmmapi は vmm.ko への ioctl, sysctl を抽象化したライブラリで、/usr/sbin/bhyve, /usr/sbin/bhyvectl はこれを呼び出すことにより vmm.ko へアクセスします (リスト 1)。

リスト 2 bhyverun.c は/usr/sbin/bhyve の中心になるコードです。

**リスト** 1 lib/libvmmapi/vmmapi.c

```
.....(省略).....
 280:  int
 281:  vm_run(struct vmctx *ctx, int vcpu, uint64_t rip, struct vm_exit *vmexit)
 282:  {
 283:   int error;
 284:   struct vm_run vmrun;
 285:
 286:   bzero(&vmrun, sizeof(vmrun));
 287:   vmrun.cpuid = vcpu;
 288:   vmrun.rip = rip;
 289:
 290:   error = ioctl(ctx->fd, VM_RUN, &vmrun);                    (1)
 291:   bcopy(&vmrun.vm_exit, vmexit, sizeof(struct vm_exit));     (2)
 292:   return (error);
 293:  }
```

- (1) 前回の記事の最後でユーザランドへ return された ioctl はここに戻ってくる。
- (2) vmm.ko から渡された vmexit 情報をコピーしてコール元へ渡す。

**リスト** 2 usr.sbin/bhyve/bhyverun.c

```
.....(省略).....
 294:  static int
 295:  vmexit_inout(struct vmctx *ctx, struct vm_exit *vme, int *pvcpu)
 296:  {
 297:   int error;
 298:   int bytes, port, in, out;
 299:   uint32_t eax;
 300:   int vcpu;
 301:
 302:   vcpu = *pvcpu;
 303:
 304:   port = vme->u.inout.port;                                  (6)
 305:   bytes = vme->u.inout.bytes;
 306:   eax = vme->u.inout.eax;
 307:   in = vme->u.inout.in;
 308:   out = !in;
```

```
 309:
......(省略).....
 322:   error = emulate_inout(ctx, vcpu, in, port, bytes, &eax, strictio);  (7)
 323:   if (error == 0 && in)                                               (16)
 324:       error = vm_set_register(ctx, vcpu, VM_REG_GUEST_RAX, eax);
 325:
 326:   if (error == 0)
 327:       return (VMEXIT_CONTINUE);                                       (17)
 328:   else {
 329:       fprintf(stderr, "Unhandled %s%c 0x%04x\n",
 330:           in ? "in" : "out",
 331:           bytes == 1 ? 'b' : (bytes == 2 ? 'w' : 'l'), port);
 332:       return (vmexit_catch_inout());
 333:   }
 334:  }
......(省略).....
 508:  static vmexit_handler_t handler[VM_EXITCODE_MAX] = {
 509:   [VM_EXITCODE_INOUT]  = vmexit_inout,                               (5)
 510:   [VM_EXITCODE_VMX]    = vmexit_vmx,
 511:   [VM_EXITCODE_BOGUS]  = vmexit_bogus,
 512:   [VM_EXITCODE_RDMSR]  = vmexit_rdmsr,
 513:   [VM_EXITCODE_WRMSR]  = vmexit_wrmsr,
 514:   [VM_EXITCODE_MTRAP]  = vmexit_mtrap,
 515:   [VM_EXITCODE_PAGING] = vmexit_paging,
 516:   [VM_EXITCODE_SPINUP_AP] = vmexit_spinup_ap,
 517:  };
 518:
 519:  static void
 520:  vm_loop(struct vmctx *ctx, int vcpu, uint64_t rip)
 521:  {
......(省略).....
 532:   while (1) {                                                         (19)
 533:       error = vm_run(ctx, vcpu, rip, &vmexit[vcpu]);                  (3)
 534:       if (error != 0) {
 535:           /*
 536:            * It is possible that 'vmmctl' or some other process
 537:            * has transitioned the vcpu to CANNOT_RUN state right
 538:            * before we tried to transition it to RUNNING.
 539:            *
 540:            * This is expected to be temporary so just retry.
```

```
541:             */
542:            if (errno == EBUSY)
543:                continue;
544:            else
545:                break;
546:        }
547:
548:        prevcpu = vcpu;
549:                    rc = (*handler[vmexit[vcpu].exitcode])(ctx, &vmexit[vcpu],
550:                                                           &vcpu);       (4)
551:        switch (rc) {
552:                    case VMEXIT_SWITCH:
553:            assert(guest_vcpu_mux);
554:            if (vcpu == -1) {
555:                stats.cpu_switch_rotate++;
556:                vcpu = fbsdrun_get_next_cpu(prevcpu);
557:            } else {
558:                stats.cpu_switch_direct++;
559:            }
560:            /* fall through */
561:        case VMEXIT_CONTINUE:
562:                            rip = vmexit[vcpu].rip + vmexit[vcpu].inst_length;   (18)
563:            break;
564:        case VMEXIT_RESTART:
565:                            rip = vmexit[vcpu].rip;
566:            break;
567:        case VMEXIT_RESET:
568:            exit(0);
569:        default:
570:            exit(1);
571:        }
572:    }
573:    fprintf(stderr, "vm_run error %d, errno %d\n", error, errno);
574:  }
```

- (6) VMExit 時に vmm.ko が取得した、in/out 命令のエミュレーションに必要な情報
  （ポート番号、アクセス幅、書き込み値（読み込み時は不要)、IO 方向（in ／ out））を展開する。
- (7) デバイスエミュレータを呼び出す。
- (16) in 命令だった場合は読み込んだ結果がゲストの rax レジスタにセットされる。
  今回は out なのでここを通らない。

- (17) VMEXIT_CONTINUE が return される。
- (5) VM_EXITCODE_INOUT で VMExit してきているので vmexit_inout() が呼ばれる。
- (19) while ループで再び vm_run() が実行され、ゲストマシンが再開される。
- (3) ioctl から抜け、ここに戻ってくる。
- (4) EXITCODE に対応したハンドラーを呼び出す。
  ここでは in/out 命令の実行で VMExit してきたものとして解説を進める。
- (18) ゲストの rip を 1 命令先に進める。

### inout.c

inout.c は I/O 命令エミュレーションを行うコードです。実際には I/O ポートごとの各デバイスエミュレータのハンドラを管理する役割を担っており、要求を受けるとデバイスエミュレータのハンドラを呼び出します。呼び出されたハンドラが実際のエミュレーション処理を行います。

**リスト** 3 usr.sbin/bhyve/inout.c

```
......(省略)......
  72:  int
  73:  emulate_inout(struct vmctx *ctx, int vcpu, int in, int port, int bytes,
  74:         uint32_t *eax, int strict)
  75:  {
  76:   int flags;
  77:   uint32_t mask;
  78:   inout_func_t handler;
  79:   void *arg;
  80:
  81:   assert(port < MAX_IOPORTS);
  82:
  83:   handler = inout_handlers[port].handler;                    (8)
  84:
  85:   if (strict && handler == default_inout)
  86:       return (-1);
  87:
  88:   if (!in) {
  89:       switch (bytes) {
  90:       case 1:
  91:           mask = 0xff;
  92:           break;
  93:       case 2:
```

```
 94:            mask = 0xffff;
 95:            break;
 96:        default:
 97:            mask = 0xffffffff;
 98:            break;
 99:        }
100:        *eax = *eax & mask;
101:    }
102:
103:    flags = inout_handlers[port].flags;
104:    arg = inout_handlers[port].arg;
105:
106:    if ((in && (flags & IOPORT_F_IN)) || (!in && (flags & IOPORT_F_OUT)))
107:        return ((*handler)(ctx, vcpu, in, port, bytes, eax, arg));  (9)
108:    else
109:        return (-1);
110:  }
.....(省略).....
141:  int
142:  register_inout(struct inout_port *iop)                          (10)
143:  {
144:    assert(iop->port < MAX_IOPORTS);
145:    inout_handlers[iop->port].name = iop->name;
146:    inout_handlers[iop->port].flags = iop->flags;
147:    inout_handlers[iop->port].handler = iop->handler;
148:    inout_handlers[iop->port].arg = iop->arg;
149:
150:    return (0);
151:  }
```

- (8) ポート番号ごとに登録されている IO ポートハンドラを取り出す。
- (9) ポート番号ごとに登録されているハンドラを取り出す。
- (10) IO ポートハンドラは register_inout() で登録されている。

### consport.c

consport.c は BHyVe 専用の準仮想化コンソールドライバです。現在は UART(Universal Asynchronous Receiver Transmitter) エミュレータが導入されたので必ずしも使う必要がなくなったのですが、デバイスエミュレータとしては最も単純な構造をしているので、デバイスエミュレータの例として取り上げました。

**リスト** 4 usr.sbin/bhyve/inout.c

```
.....(省略).....
 95:  static void
 96:  ttywrite(unsigned char wb)
 97:  {
 98:   (void) write(STDOUT_FILENO, &wb, 1);                          (15)
 99:  }
100:
101:  static int
102:  console_handler(struct vmctx *ctx, int vcpu, int in, int port, int bytes,
103:       uint32_t *eax, void *arg)
104:  {
105:   static int opened;
106:
107:   if (bytes == 2 && in) {
108:       *eax = BVM_CONS_SIG;
109:       return (0);
110:   }
111:
112:   if (bytes != 4)
113:       return (-1);
114:
115:   if (!opened) {
116:       ttyopen();
117:       opened = 1;
118:   }
119:
120:   if (in)                                                        (13)
121:       *eax = ttyread();
122:   else
123:       ttywrite(*eax);                                            (14)
124:
125:   return (0);
126:  }
127:
128:  static struct inout_port consport = {
129:   "bvmcons",
130:   BVM_CONSOLE_PORT,
```

```
131:   IOPORT_F_INOUT,
132:   console_handler                                          (12)
133:  };
134:
135:  void
136:  init_bvmcons(void)
137:  {
138:
139:   register_inout(&consport);                               (11)
140:  }
```

- (15) ttywrite() は write() で標準出力に文字を書き込む。
- (13) console_handler() では IO 方向が in なら ttyread()、out なら ttywrite() を実行し、標準入出力に対して IO を行う。
- (14) 今回は out が実行された場合を見ていく。
  eax で指定された書き込み値を ttywrite() に渡している。
- (12) 登録するハンドラ関数として console_handler() が指定されている。
- (11) consport デバイスは起動時にここでハンドラを登録している。

## まとめ

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、ソースコードを解説しました。今回までで、ハイパーバイザの実行サイクルに関するソースコードの解説を一通り行ったので、次回は virtio のしくみについて見ていきます。

## ライセンス