Linux Technology Center

Kernel/VM 探検隊
QEMUとkvmとvt-x

2013年4月13日

日立製作所 横浜研究所
PCP (Private Cloud Platform Project) 兼
LTC (Linux Technology Center)
林　佳寛

HITACHI
Inspire the Next

自己紹介 (1/2)

- 名前:　　林　佳寛
- 職場:　　日立製作所　横浜研究所
- これまでの仕事:
  - OSのいろいろ　特にDISK I/O周りに携わる期間が長い

自己紹介 (2/2)

Linux Technology Center

- 趣味:
  - アニメ鑑賞
  - DTM・DJ
  - プログラミング
    - エゴサーチ自動化
      - twitterは人間が読むものじゃない。自動的に検索するものだ。
    - kconfig増分自動投稿blog
      - とりあえずkernelの新機能を一通りチェックしたい。

この発表/資料について

- 仮想化がらみの仕事を始めるにあたり
勉強したことを横展開するため
部署内勉強会向けに作った資料を修正したもの
- なので、ターゲットはqemuやkvmに興味があってこれから
取り組もうという人向け
- kernel/VM勉強会常連の方にとってはいつも聞く話題で
少し退屈かも

QEMU、kvm、vt-xとは？

Linux Technology Center

- qemuとは （QEMUの公式ページ http://wiki.qemu.org より）
  - 「QEMU is a generic and open source machine emulator and virtualizer.」
- kvmとは （KVMの公式ページ http://www.linux-kvm.org より）
  - 「KVM (for Kernel-based Virtual Machine) is a full virtualization solution for Linux on x86 hardware containing virtualization extensions (Intel VT or AMD-V). It consists of a loadable kernel module, kvm.ko, that provides the core virtualization infrastructure and a processor specific module, kvm-intel.ko or kvm-amd.ko. KVM also requires a modified QEMU although work is underway to get the required changes upstream. 」
- vt-xとは （Intel® 64, IA-32 SDMより）
  - 「Virtual-machine extensions define processor-level support for virtual machines on IA-32 processors.」

→よくわからない

とすると、よく起きること。

- 書籍がない・・・
  - 新しい分野だし中身のコードレベルまで興味がある人は少ないからしかたないか・・・
  - 英語の書籍があると思って買おうと思ったらドイツ語だった。あの本もこの本もなぜかドイツ語ばかり・・・

    → 昔からドイツにはエミュレーションをやってた人が多いためらしいです。
- 仕方ないからウェブ上の資料を探そう

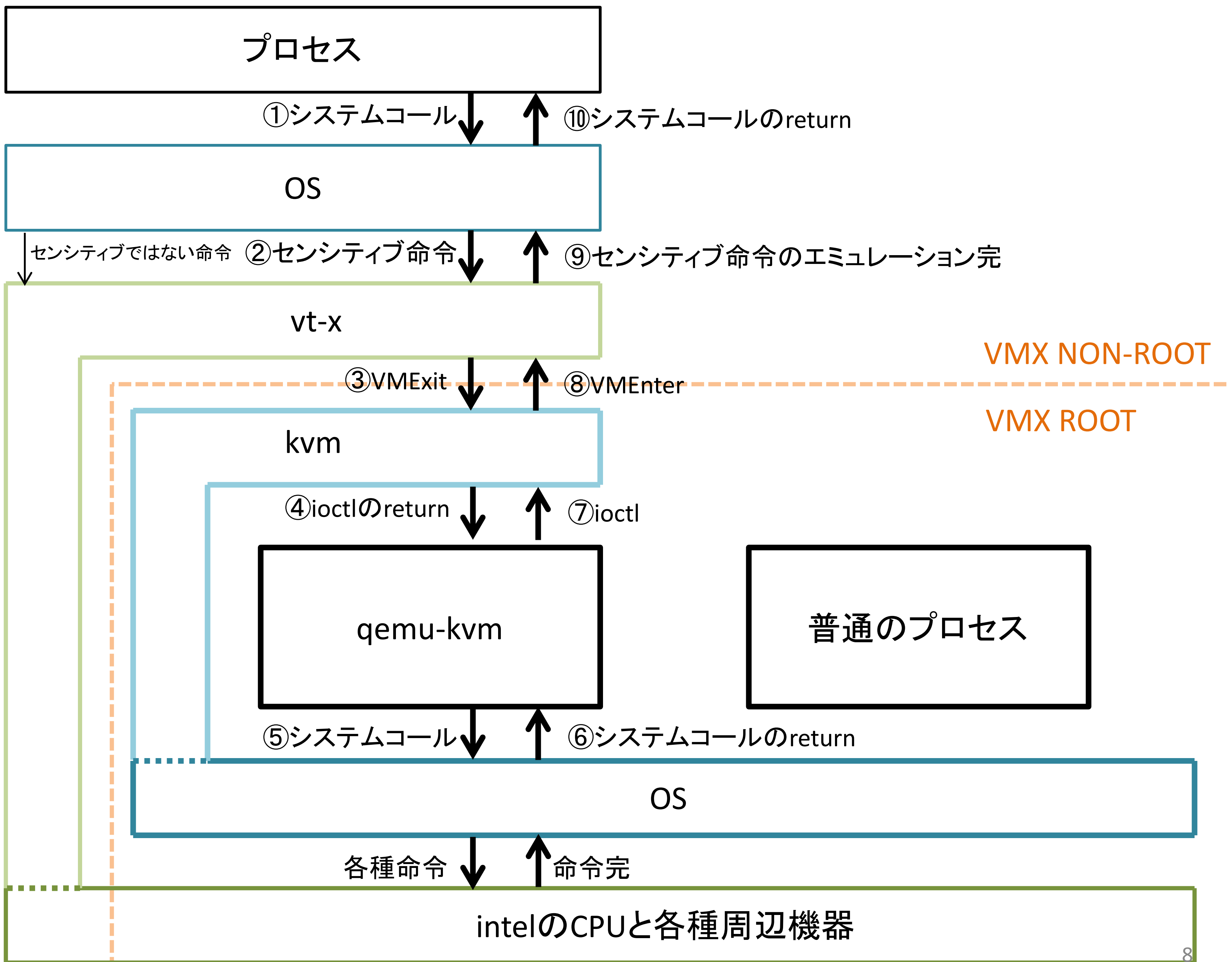
    → なんだかよくわからなかった

    仕方ないから自分でソースコードを読むか・・・という

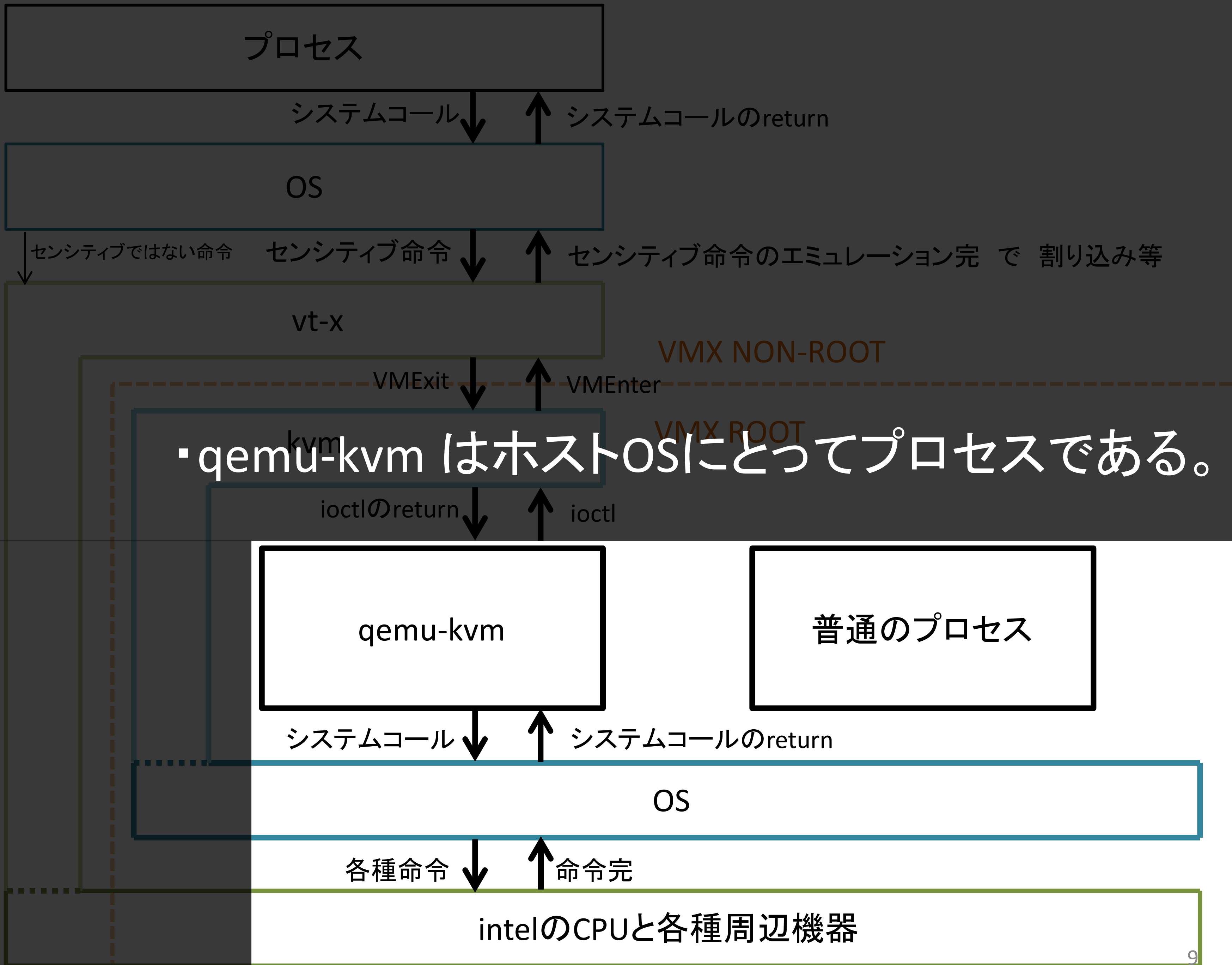
    そんなあなたの役に立ちたい

qemuとvt-xとkvmの概要

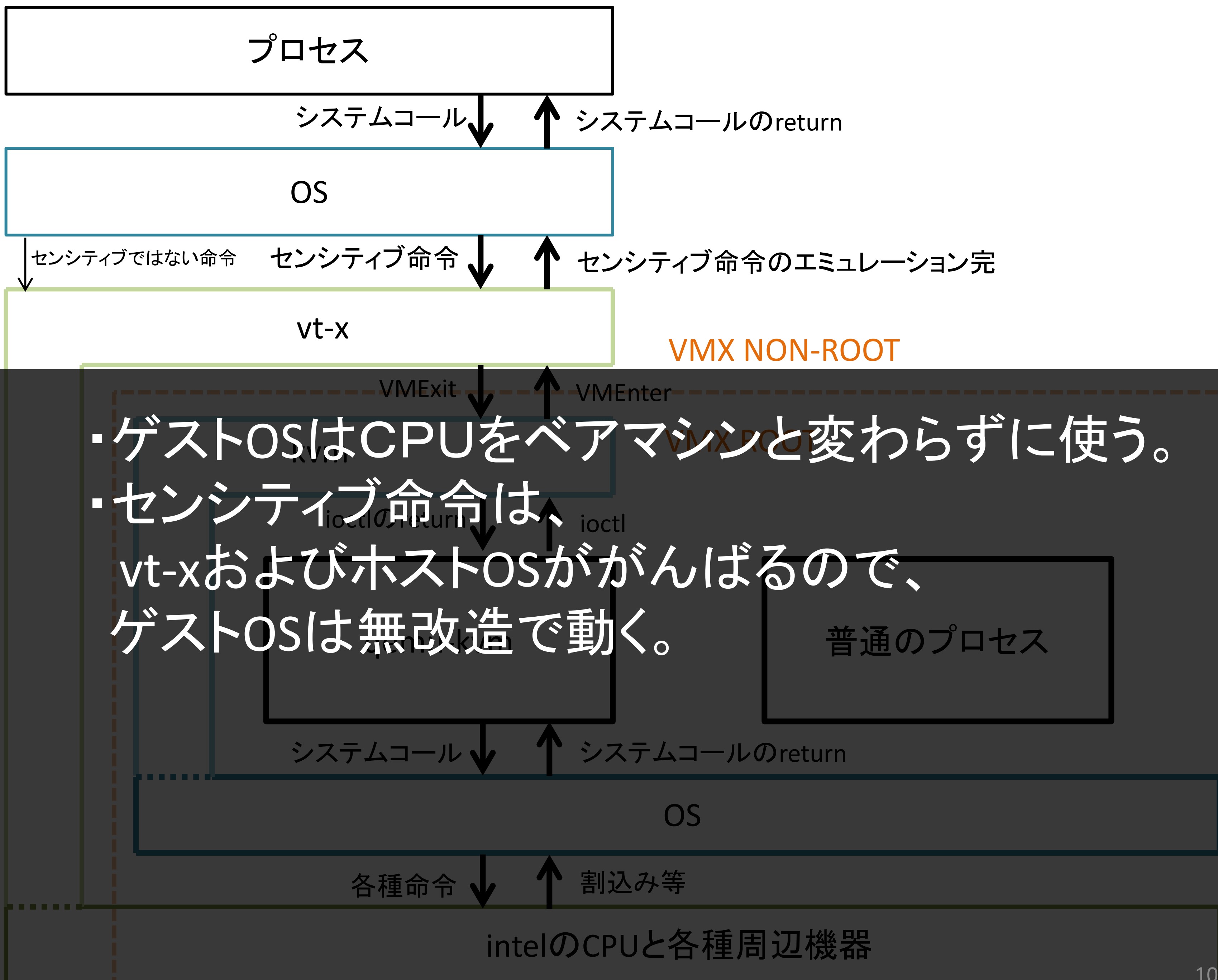
- 仮想化されたOS上のアプリケーションの命令が実行されるまでの流れを上から下まで書いてみた。

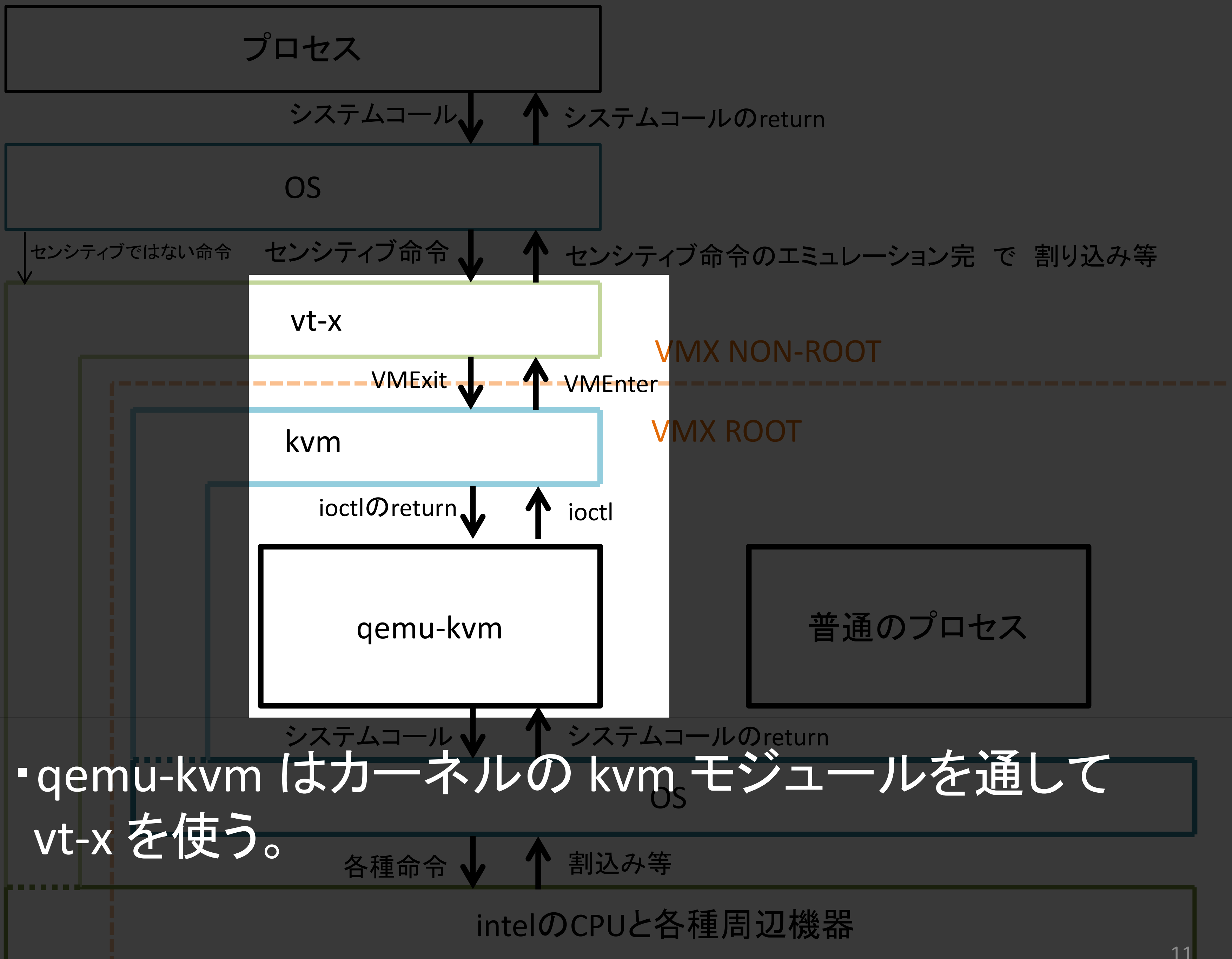


プロセス
①システムコール
⑩システムコールのreturn
OS
センシティブではない命令
②センシティブ命令
⑨センシティブ命令のエミュレーション完
vt-x
VMX NON-ROOT
③VMExit
⑧VMEnter
VMX ROOT
kvm
④ioctlのreturn
⑦ioctl
qemu-kvm
普通のプロセス
⑤システムコール
⑥システムコールのreturn
OS
各種命令
命令完
intelのCPUと各種周辺機器

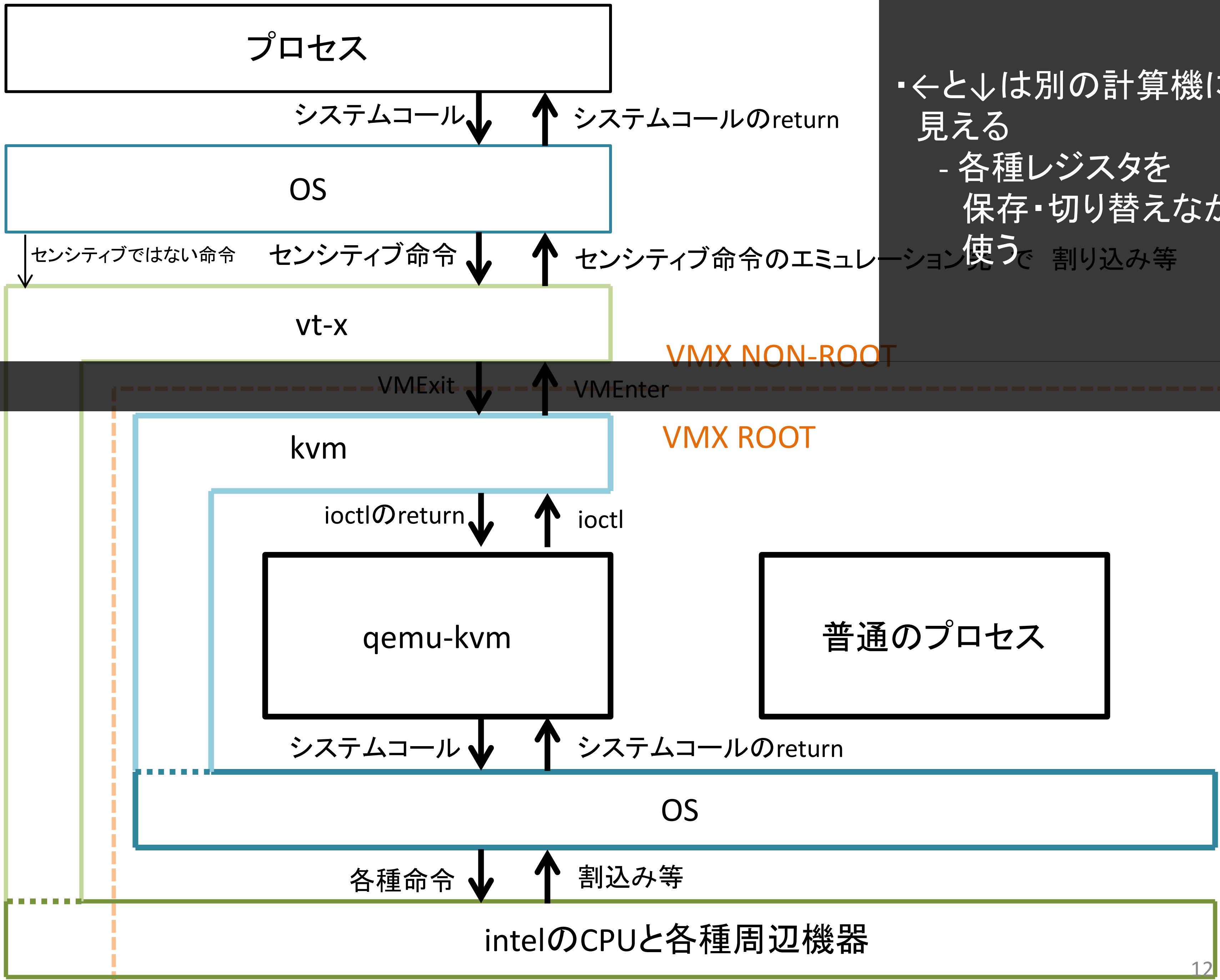


プロセス
システムコール
システムコールのreturn
OS
センシティブではない命令
センシティブ命令
センシティブ命令のエミュレーション完 で 割り込み等
vt-x
VMX NON-ROOT
VMExit
VMEnter
VMX ROOT
kvm
ioctlのreturn
ioctl
・qemu-kvm はホストOSにとってプロセスである。
qemu-kvm
普通のプロセス
システムコール
システムコールのreturn
OS
各種命令
命令完
intelのCPUと各種周辺機器


プロセス
システムコール
システムコールのreturn
OS
センシティブではない命令
センシティブ命令
センシティブ命令のエミュレーション完
vt-x
VMX NON-ROOT
VMExit
VMEnter
ioctl
普通のプロセス
システムコール
システムコールのreturn
OS
各種命令
割込み等
intelのCPUと各種周辺機器
・ゲストOSはCPUをベアマシンと変わらずに使う。
・センシティブ命令は、
vt-xおよびホストOSががんばるので、
ゲストOSは無改造で動く。



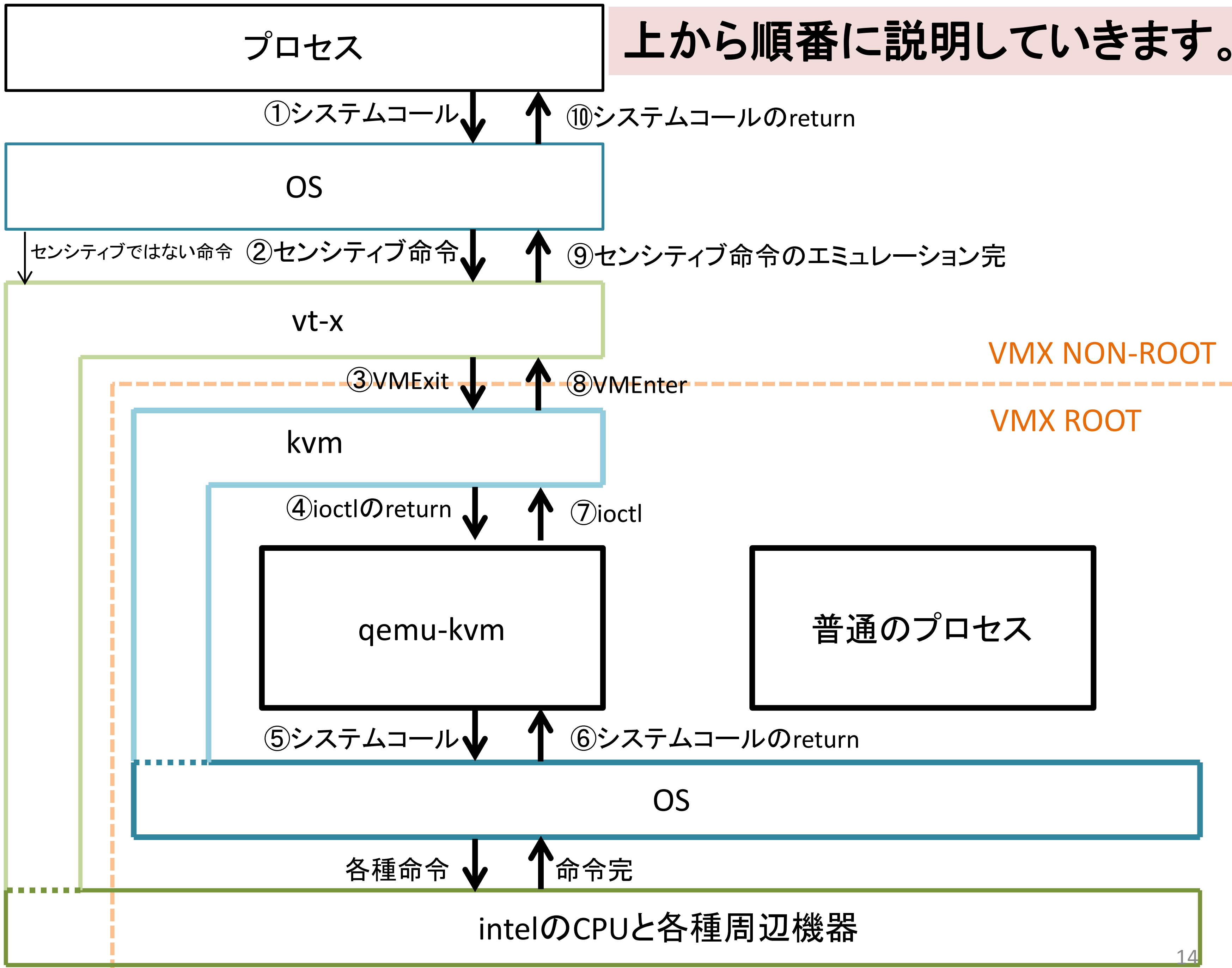
- qemu-kvm はカーネルの kvm モジュールを通して vt-x を使う。


プロセス
システムコール
システムコールのreturn
OS
センシティブではない命令
センシティブ命令
センシティブ命令のエミュレーション後で　割り込み等
vt-x
VMX NON-ROOT
VMExit
VMEnter
kvm
VMX ROOT
ioctlのreturn
ioctl
qemu-kvm
普通のプロセス
システムコール
システムコールのreturn
OS
各種命令
割込み等
intelのCPUと各種周辺機器
・←と↓は別の計算機に
見える
- 各種レジスタを
保存・切り替えながら
使う

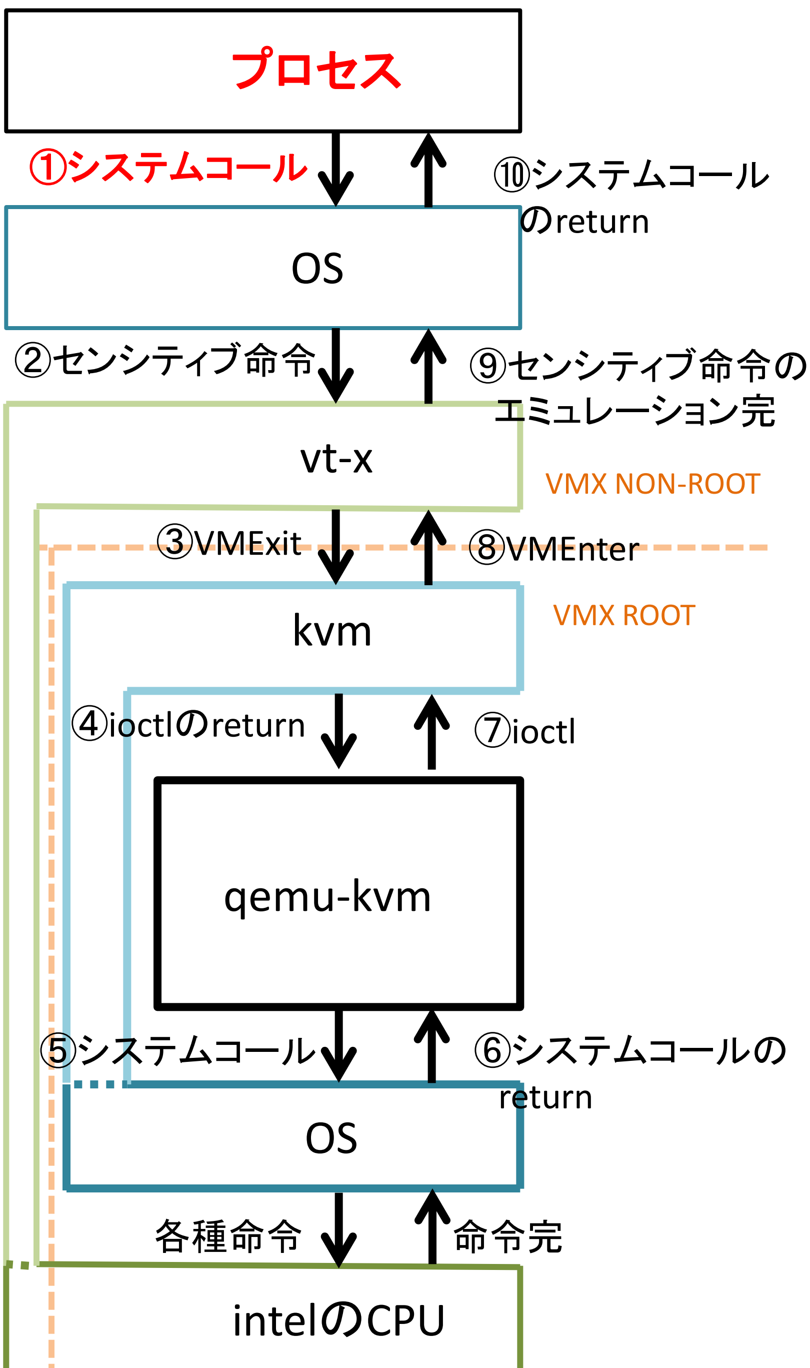
実例: **virtio-blk**上の ファイルへの**write()**

Linux Technology Center

- 目的:
  - qemuとkvmとvt-xで どのように処理が流れているか知る
- 環境(参照しているコード):
  - Fedora 15 x86_64 （kernelは2.6.42）
- シチュエーション:
  - ゲストOSは起動済み
  - ゲストOSのプロセスが virtio-blkデバイス上のファイルに writeしたとき

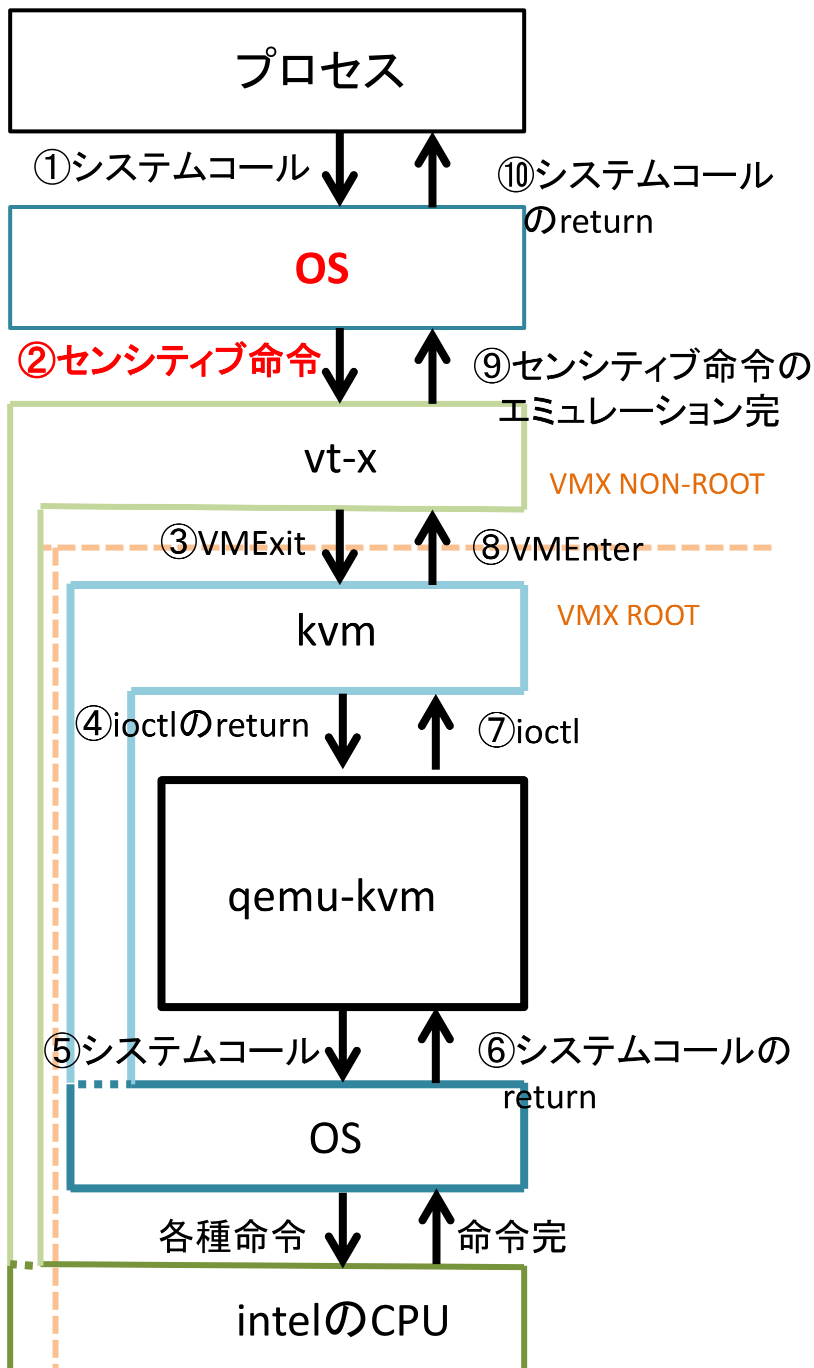


プロセス
上から順番に説明していきます。
①システムコール
⑩システムコールのreturn
OS
センシティブではない命令
②センシティブ命令
⑨センシティブ命令のエミュレーション完
vt-x
VMX NON-ROOT
③VMExit
⑧VMEnter
VMX ROOT
kvm
④ioctlのreturn
⑦ioctl
qemu-kvm
普通のプロセス
⑤システムコール
⑥システムコールのreturn
OS
各種命令
命令完
intelのCPUと各種周辺機器



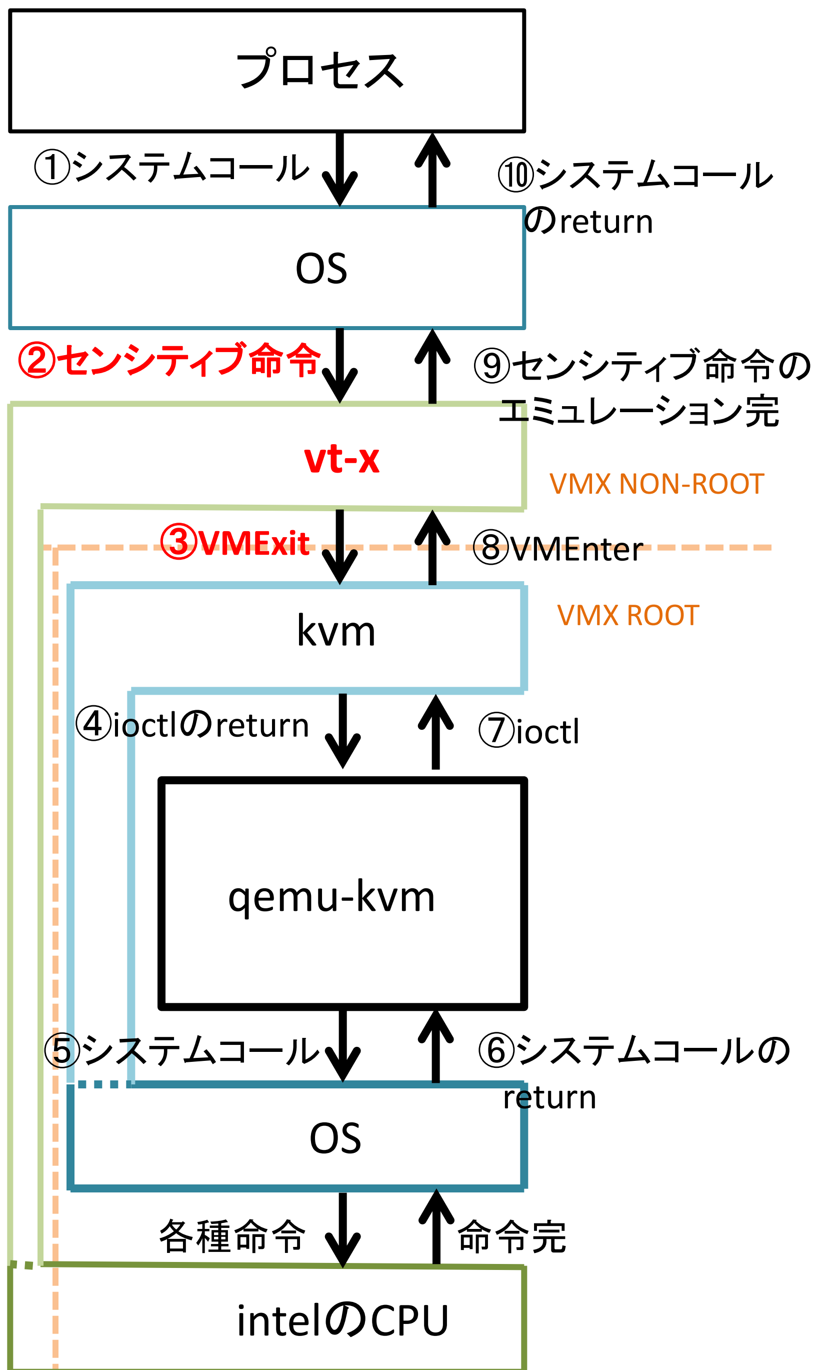
①システムコール

- プロセスがファイルを開いて、書き込む
  - write(fd, buf, count);
- write()はシステムコールなので、OSに処理が移る



②センシティブ命令

- writeシステムコールから処理を受け取ったゲストOSは、
  - ファイルシステムを通して、ファイルへのI/OをブロックデバイスへのI/Oに変換
  - ブロックデバイスに書き込む
  - 今回はvirtio-blk(準仮想化デバイス)を使っている
    - virtioは、ゲストOSがゲストとホストが共有するメモリに用意したqueueに要求を入れて、「今queueの中にある要求を処理しろ！」とホストOSに伝える。
    - virtqueue_kick()→
      vp_notify()→
      io_write16() →
      outw(value,port)
      PORT I/Oを発行

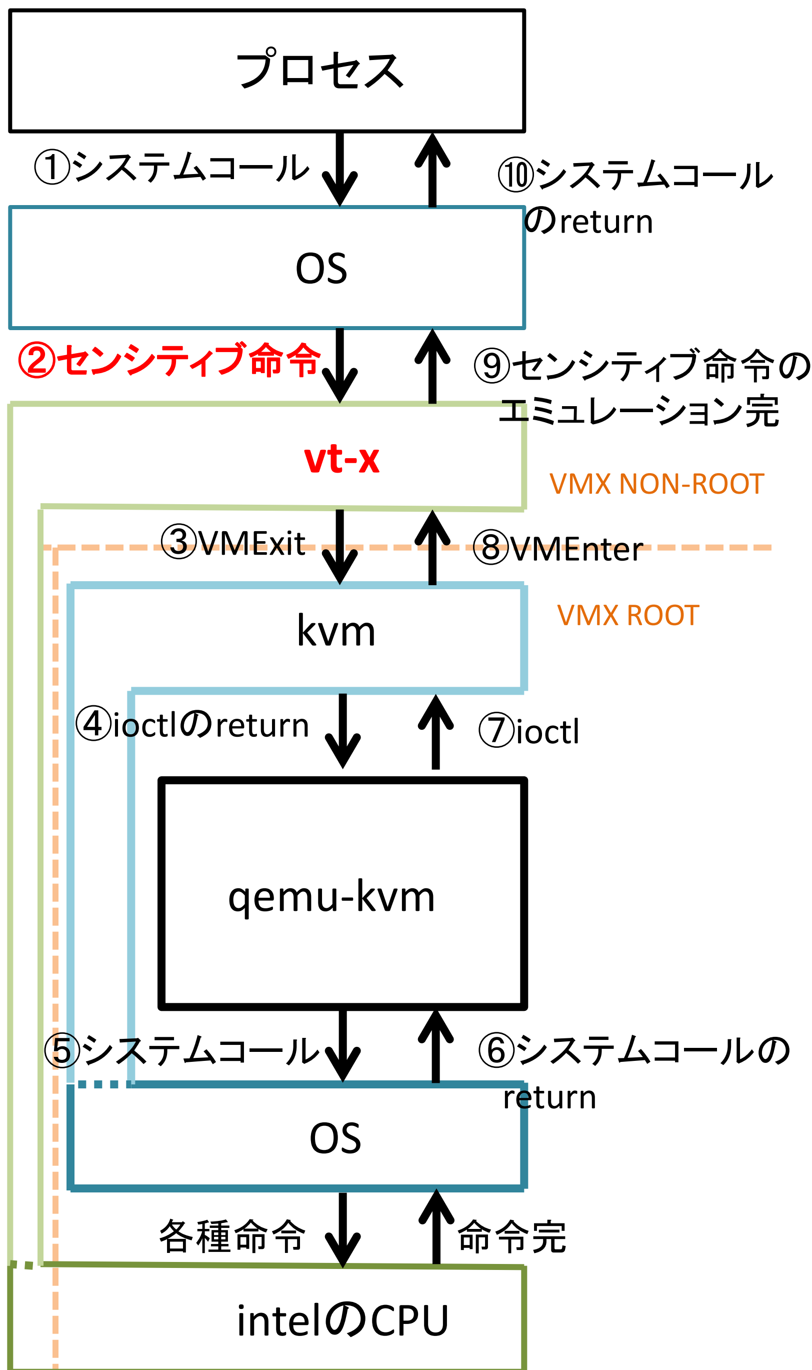


その時 vt-x が動いた

- 普段、CPUは、ゲストOSやプロセスの命令をを普通に動かしている
  - 足し算とか、掛け算とか、そういう命令は、別にゲストOSであろうとホストOSであろうとやることは変わらない
- たまに、そのままCPUで実行するとまずい命令をゲストOSやプロセスが動かそうとする

まずい命令＝センシティブ命令

- **vt-xはセンシティブ命令を検出するハードウェアの仕組み**
  - センシティブ命令でなければホストOSのときと同様に実行
  - センシティブ命令ならVMExitし、kvmに処理を移す
    - kvmやqemuがエミュレーションしてくれることを期待している。

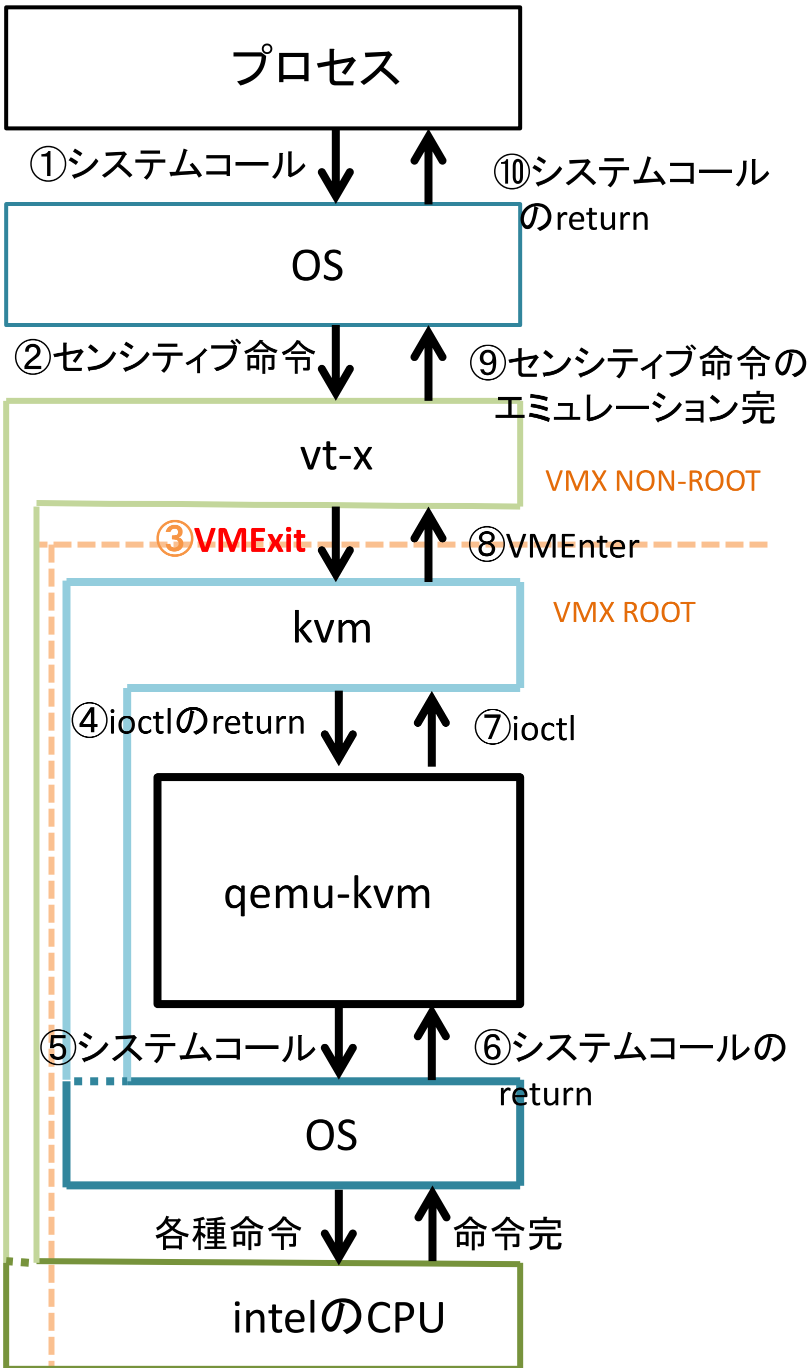


まずい命令＝センシティブ命令

- 例えば、ゲストOSが発行したPORT I/Oをホストで実行するのと同じように実行
  - ゲストOSにとって、そのPORTはvirtioデバイスのポートだけど、
  - ホストOSにとってはそのPORTには何もないかもしれないし、別のデバイスかもしれない
  - ホストOSと同じように実行するとまずい

  →VMExitしてホストOSでエミュレートする

- センシティブ命令
  - **External interrupt.:** An external interrupt arrived and the “external-interruptexiting” VM-execution control was 1.
    - 割込み
  - **EPT violation.:** An attempt to access memory with a guest-physical address was disallowed by the configuration of the EPT paging structures.
    - EPT使っていて仮想物理にページがないとき
  - 他にもいろいろ intel SDMの後ろの方に



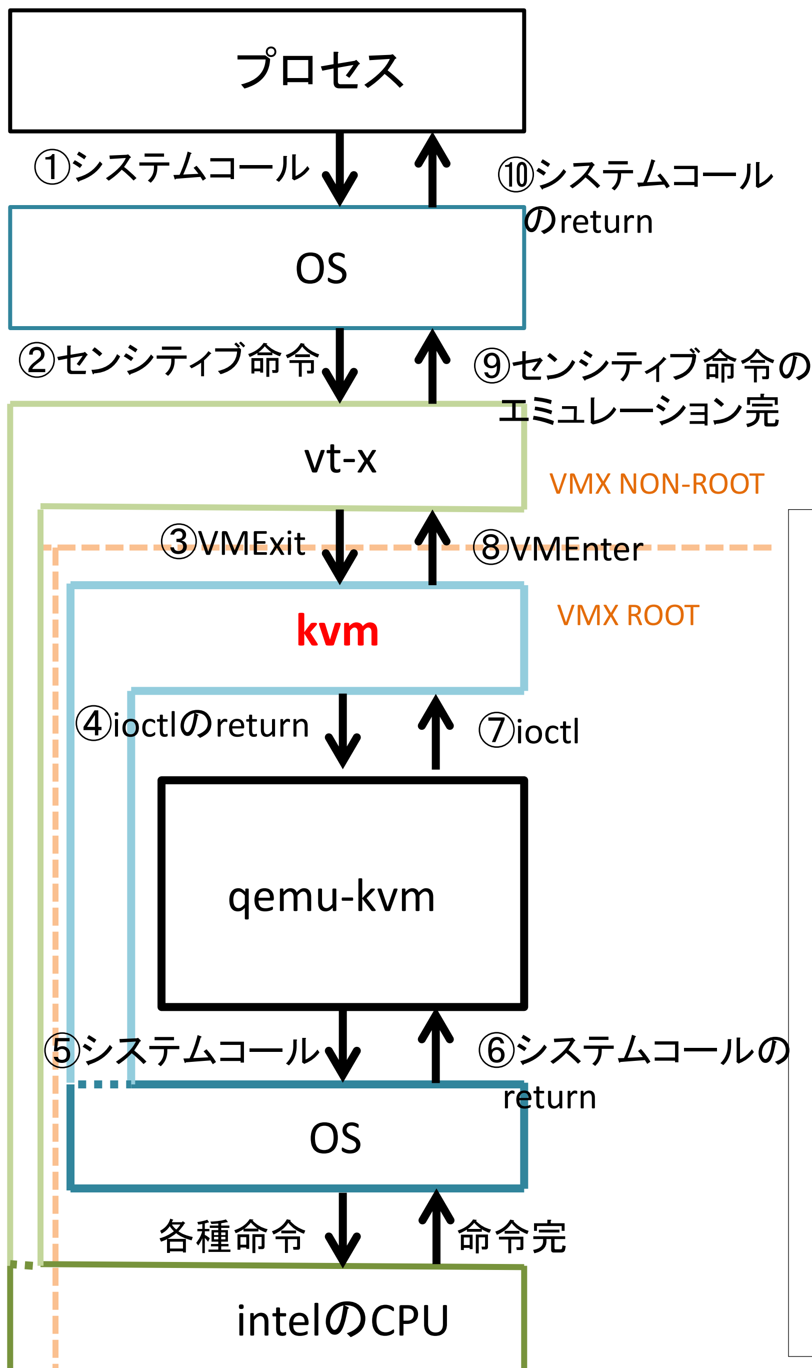
③VMExit

- VM Exitとは
  - センシティブ命令の実行を契機として、
  - VM Enterした命令の次の命令に処理が戻ってくること
    - VM Enterについては後で説明する

- センシティブ命令はそのまま実行できないのでホストOSにエミュレートしてもらうために処理がホストOSに移ってきました。

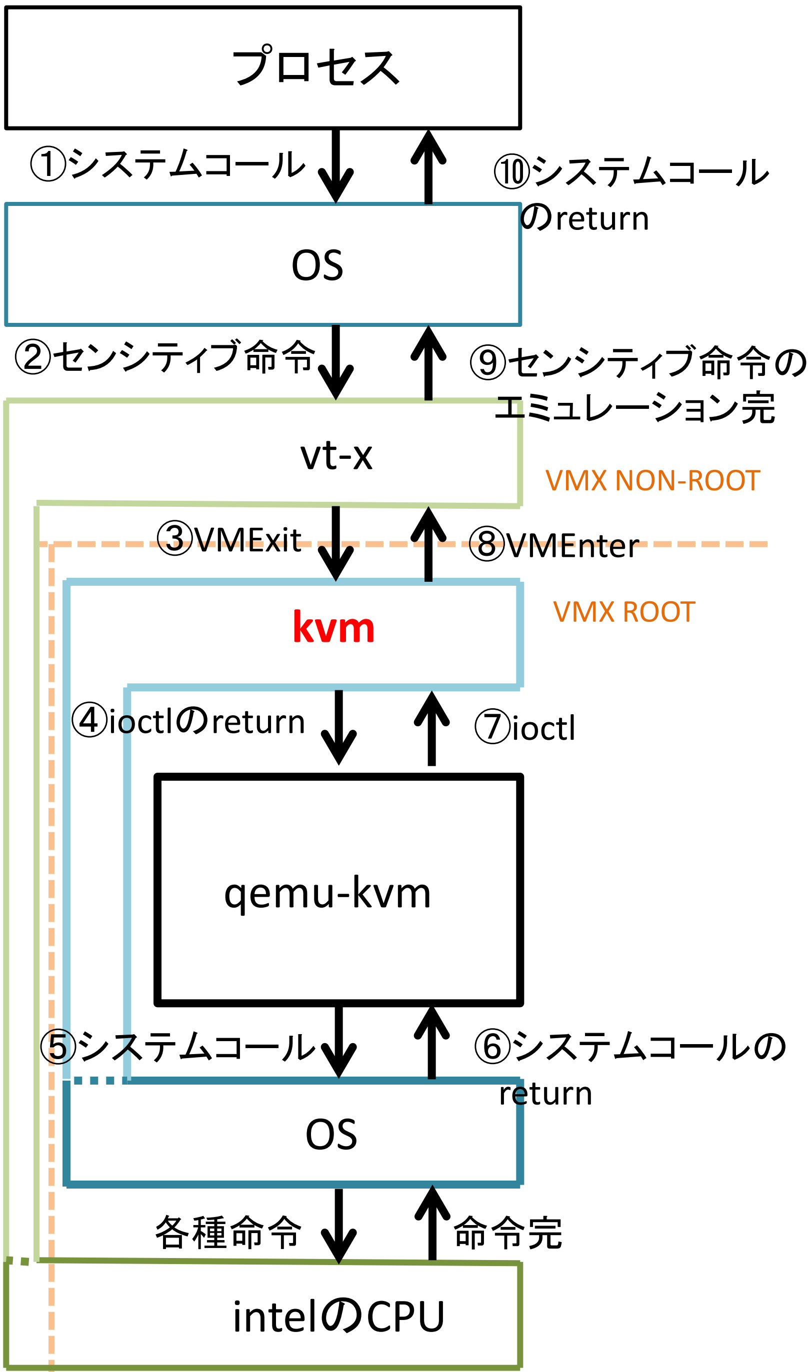
kvm

- 過去にVX Enterをしたのはkvmなのでkvmのコードに戻ってくる



vmx_vcpu_run(struct kvm_vcpu *vcpu) 抜粋

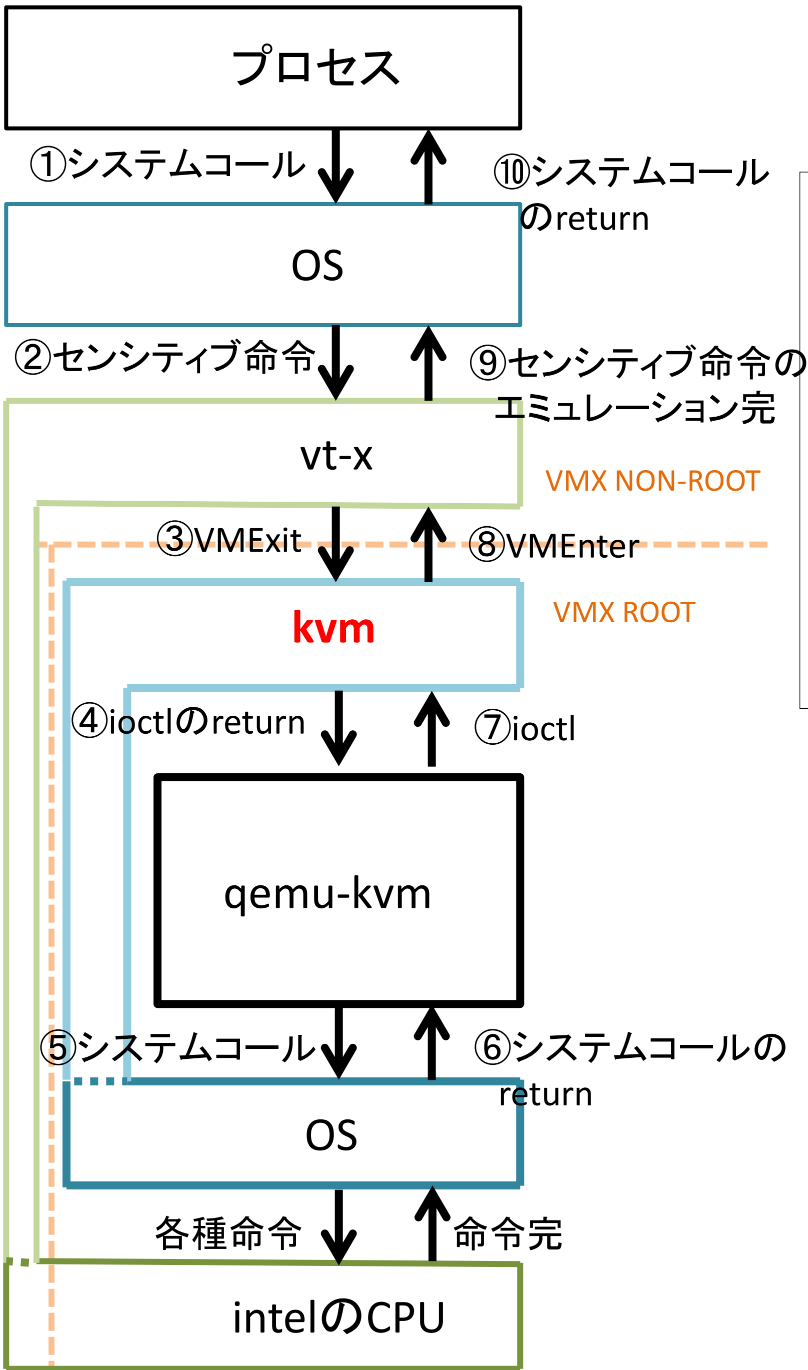
```
/* Enter guest mode */
        "jne .Llaunched ¥n¥t"
        __ex(ASM_VMX_VMLAUNCH) "¥n¥t"
        "jmp .Lkvm_vmx_return ¥n¥t"
        ".Llaunched: " __ex(ASM_VMX_VMRESUME) "¥n¥t"
        ".Lkvm_vmx_return: "    ← ここに戻ってくる
        /* Save guest registers, load host registers, keep flags */
        "mov %0, %c[wordsize](%%"R"sp) ¥n¥t"
        "pop %0 ¥n¥t"
        "mov %%"R"ax, %c[rax](%0) ¥n¥t"
        "mov %%"R"bx, %c[rbx](%0) ¥n¥t"
        "pop"Q" %c[rcx](%0) ¥n¥t"
        "mov %%"R"dx, %c[rdx](%0) ¥n¥t"
        "mov %%"R"si, %c[rsi](%0) ¥n¥t"
        "mov %%"R"di, %c[rdi](%0) ¥n¥t"
        "mov %%"R"bp, %c[rbp](%0) ¥n¥t"
```



kvm

vcpu_enter_guest(struct kvm_vcpu *vcpu) 抜粋

```
(前略)
 kvm_x86_ops->run(vcpu);   /* 実体はvmx_vcpu_run() */
（中略）
r = kvm_x86_ops->handle_exit(vcpu);
  /* 実体は vmx_handle_exit() */
（後略）
```

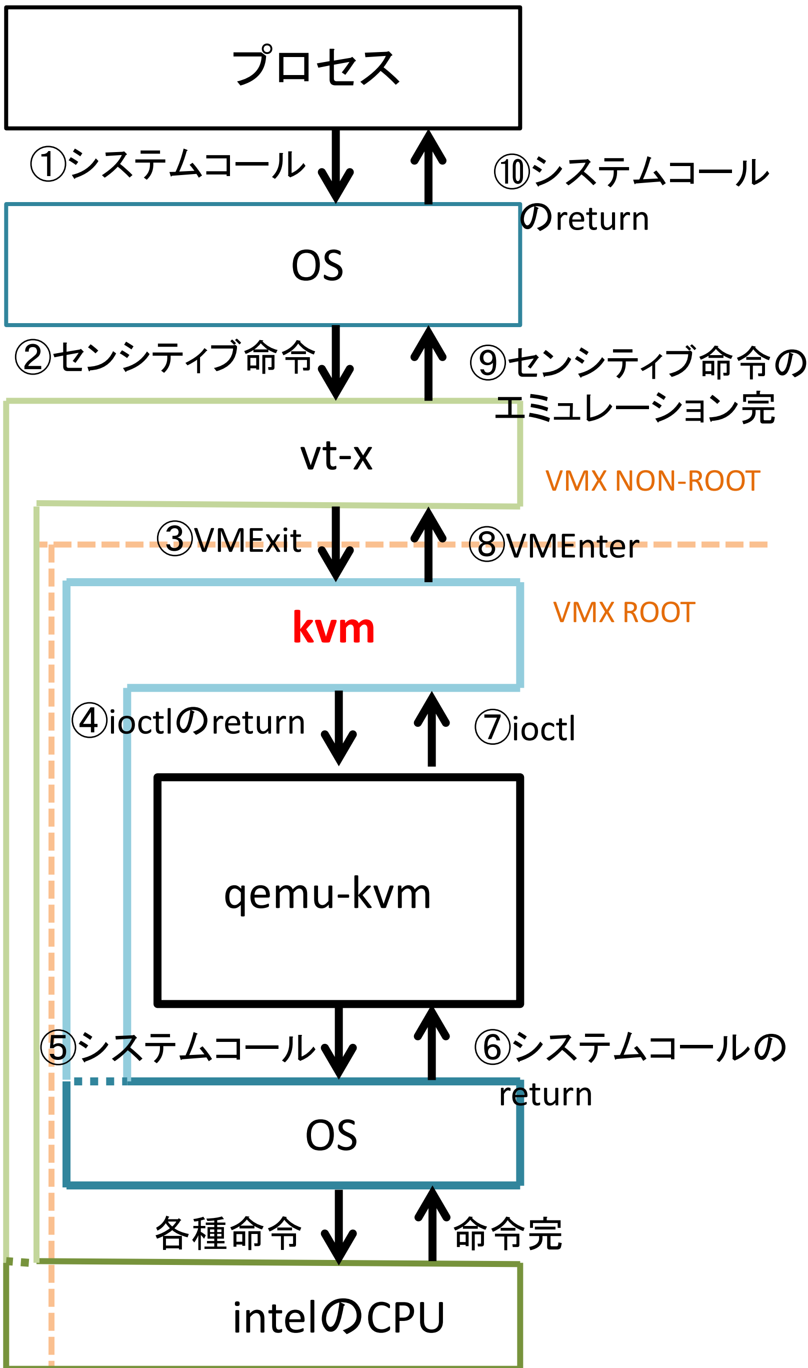


kvm

vmx_handle_exit(struct kvm_vcpu *vcpu)抜粋

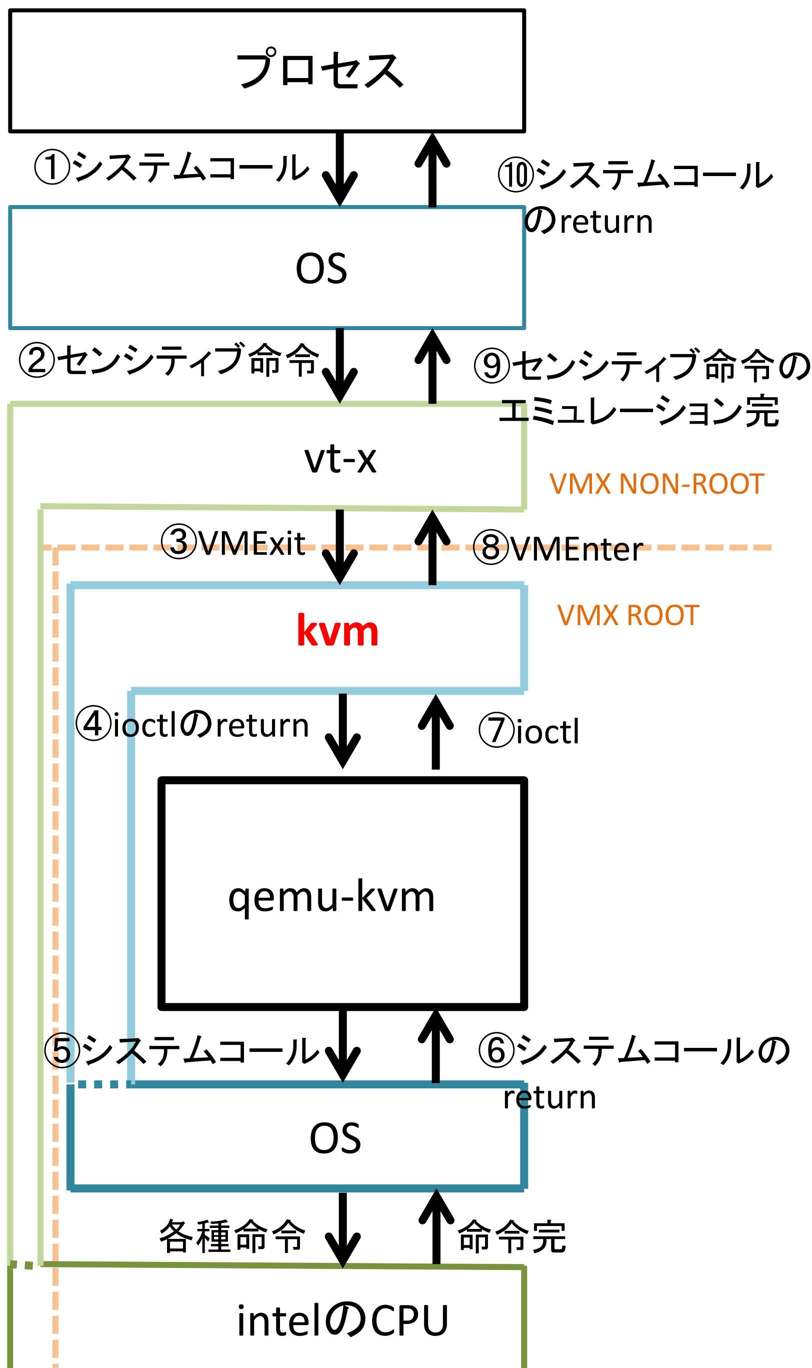
```
(前略)
 if (exit_reason < kvm_vmx_max_exit_handlers
        && kvm_vmx_exit_handlers[exit_reason])
        return kvm_vmx_exit_handlers[exit_reason](vcpu);
   else {
        vcpu->run->exit_reason = KVM_EXIT_UNKNOWN;
        vcpu->run->hw.hardware_exit_reason = exit_reason;
   }
   return 0;
}
```

- VM Exitの要因に応じたハンドラを呼び出す



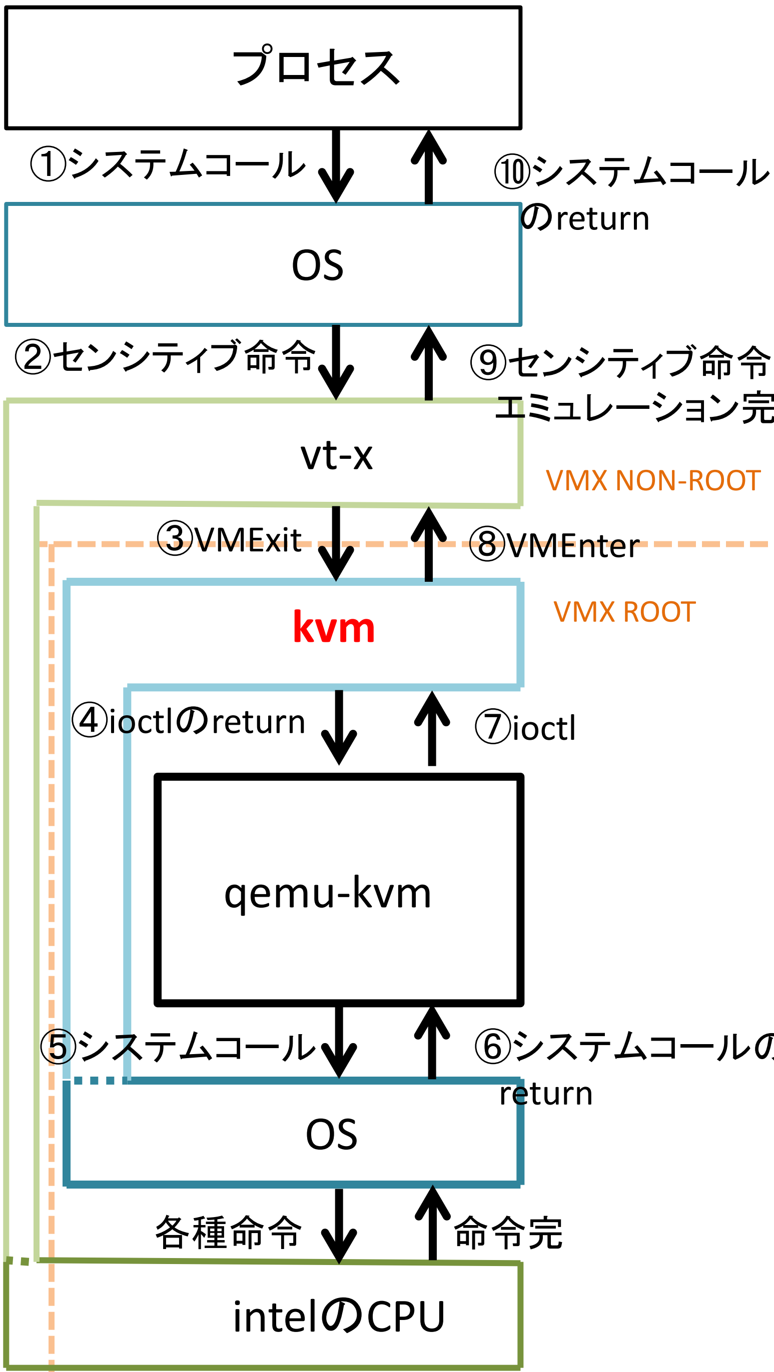
kvm

```
static int (*kvm_vmx_exit_handlers[])(struct kvm_vcpu *vcpu) = {
    [EXIT_REASON_EXCEPTION_NMI]          = handle_exception,
    [EXIT_REASON_EXTERNAL_INTERRUPT]     = handle_external_interrupt,
    [EXIT_REASON_TRIPLE_FAULT]           = handle_triple_fault,
    [EXIT_REASON_NMI_WINDOW]             = handle_nmi_window,
    [EXIT_REASON_IO_INSTRUCTION]         = handle_io,
    [EXIT_REASON_CR_ACCESS]              = handle_cr,
    [EXIT_REASON_DR_ACCESS]              = handle_dr,
    [EXIT_REASON_CPUID]                  = handle_cpuid,
    [EXIT_REASON_MSR_READ]               = handle_rdmsr,
    [EXIT_REASON_MSR_WRITE]              = handle_wrmsr,
    [EXIT_REASON_PENDING_INTERRUPT]      = handle_interrupt_window,
    [EXIT_REASON_HLT]                    = handle_halt,
    [EXIT_REASON_INVD]                   = handle_invd,
    [EXIT_REASON_INVLPG]                 = handle_invlpg,
    [EXIT_REASON_VMCALL]                 = handle_vmcall,
    [EXIT_REASON_VMCLEAR]                = handle_vmx_insn,
    [EXIT_REASON_VMLAUNCH]               = handle_vmx_insn,
    [EXIT_REASON_VMPTRLD]                = handle_vmx_insn,
    [EXIT_REASON_VMPTRST]                = handle_vmx_insn,
    [EXIT_REASON_VMREAD]                 = handle_vmx_insn,
    [EXIT_REASON_VMRESUME]               = handle_vmx_insn,
    [EXIT_REASON_VMWRITE]                = handle_vmx_insn,
    [EXIT_REASON_VMOFF]                  = handle_vmx_insn,
    [EXIT_REASON_VMON]                   = handle_vmx_insn,
    [EXIT_REASON_TPR_BELOW_THRESHOLD]    = handle_tpr_below_threshold,
    [EXIT_REASON_APIC_ACCESS]            = handle_apic_access,
    [EXIT_REASON_WBINVD]                 = handle_wbinvd,
    [EXIT_REASON_XSETBV]                 = handle_xsetbv,
    [EXIT_REASON_TASK_SWITCH]            = handle_task_switch,
    [EXIT_REASON_MCE_DURING_VMENTRY]     = handle_machine_check,
    [EXIT_REASON_EPT_VIOLATION]          = handle_ept_violation,
    [EXIT_REASON_EPT_MISCONFIG]          = handle_ept_misconfig,
    [EXIT_REASON_PAUSE_INSTRUCTION]      = handle_pause,
    [EXIT_REASON_MWAIT_INSTRUCTION]      = handle_invalid_op,
    [EXIT_REASON_MONITOR_INSTRUCTION]    = handle_invalid_op,
};
```



kvmの handle_io() の概要

- 引数に指定されたポート番号を処理するための関数が設定されているかを確認する
- が、virtio-blk用の関数は設定されていない
- 関数が設定されていて処理がうまくいけば 1を返す
- 関数が設定されていないと 0を返す
  - virtio-blkは関数ないのでこっちを通る



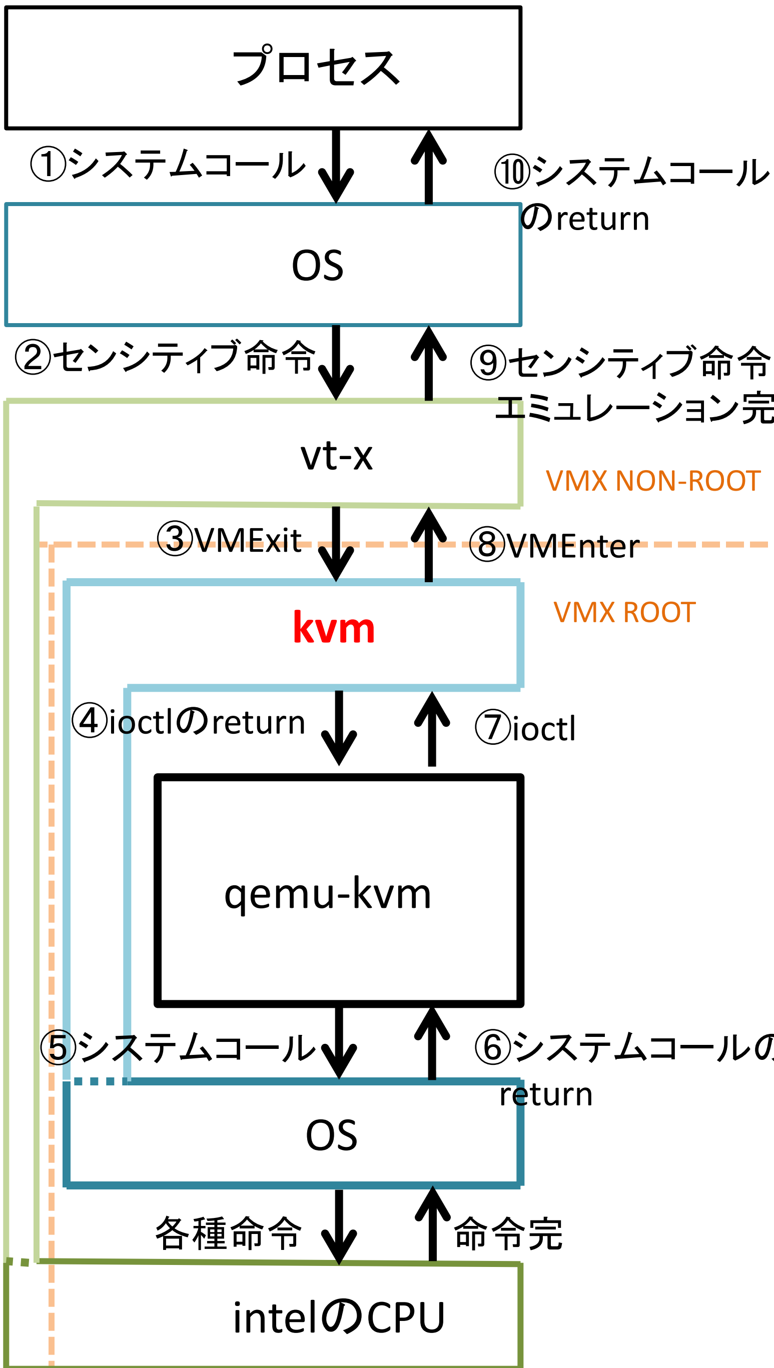
kvm

vcpu_enter_guest(struct kvm_vcpu *vcpu) 抜粋

```
(前略)
 kvm_x86_ops->run(vcpu);    /* 実体はvmx_vcpu_run() */
（中略）
r = kvm_x86_ops->handle_exit(vcpu);   ←これが終わって
                                        帰ってきたところ
  /* 実体は vmx_handle_exit() */

out:
   return r;     ←  0 が入ってreturnする。
}
```

- 呼び出し元は　__vcpu_run()

kvm



__vcpu_run() 抜粋

```
(前略)

while( r > 0 ){
（中略）
        r = vcpu_enter_guest(vcpu);
（中略）

        if (r <= 0)
                break;
（中略）
}

return  r;
```

- kvmの中でエミュレーションできなければreturnする。
- エミュレーションできてればreturnせずにすぐVMEnterする。
- 呼び出し元は kvm_arch_vcpu_ioctl_run()

④ ioctlのreturn

kvm_arch_vcpu_ioctl_run() 抜粋

```
    r = __vcpu_run(vcpu);

out:
    post_kvm_run_save(vcpu);
    if (vcpu->sigset_active)
        sigprocmask(SIG_SETMASK, &sigsaved, NULL);

    return r;
```

- /dev/kvmに対するioctlがreturnした。

プロセス

①システムコール

⑩システムコールのreturn

OS

②センシティブ命令

⑨センシティブ命令のエミュレーション完

vt-x

VMX NON-ROOT

③VMExit

⑧VMEnter

kvm

VMX ROOT

④ioctlのreturn

⑦ioctl

qemu-kvm

⑤システムコール

⑥システムコールのreturn

OS

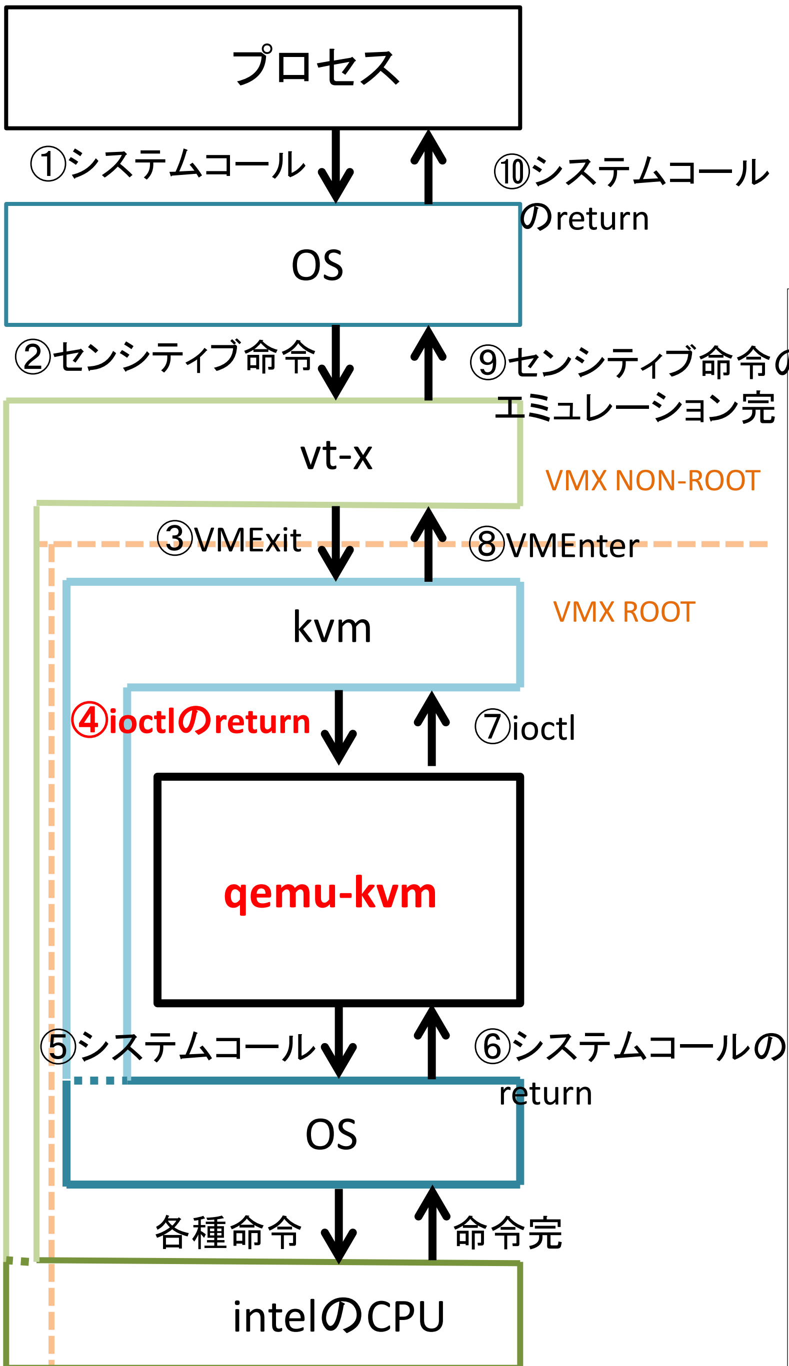
各種命令

命令完

intelのCPU

qemu-kvmがしていること

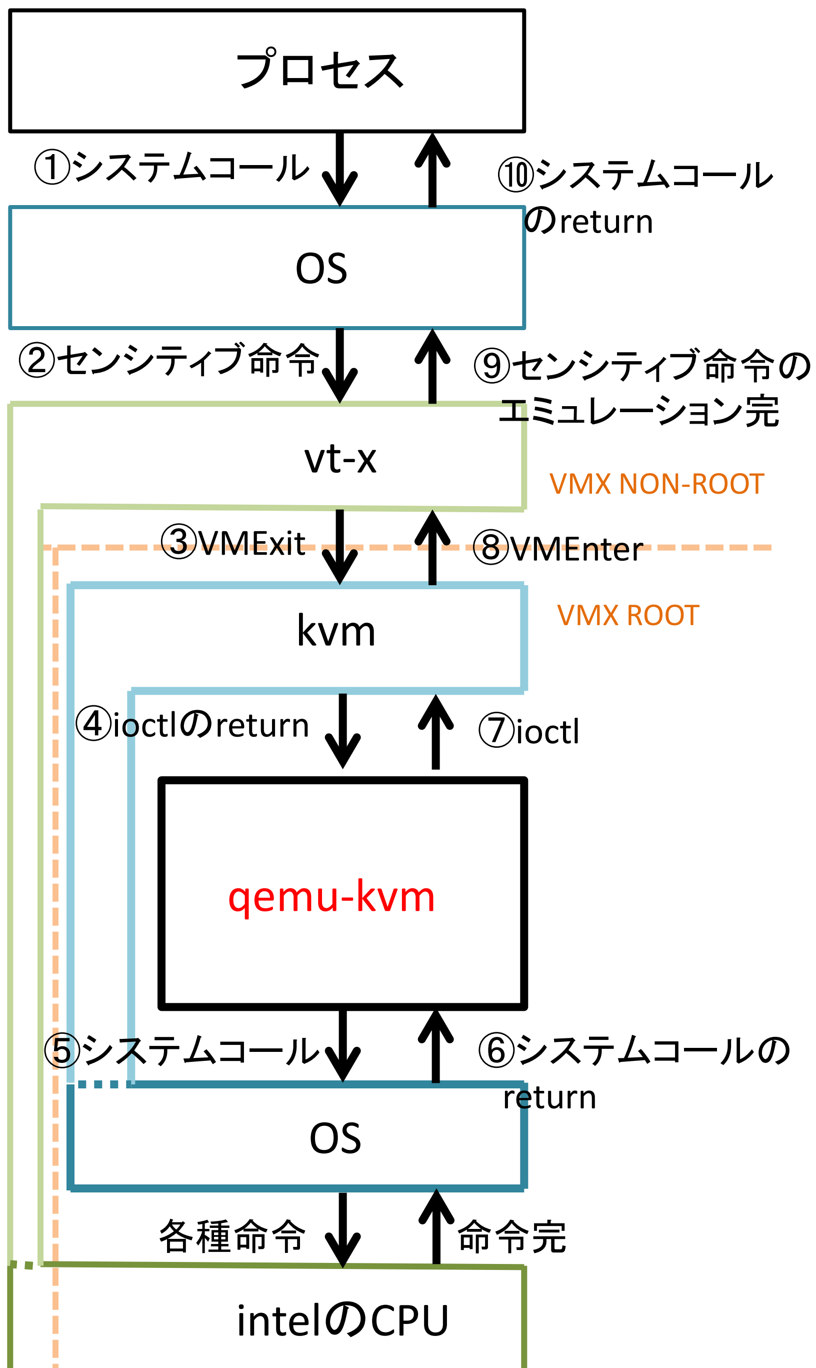
- ioctlが返ってきた理由を確認



kvm_run() 抜粋

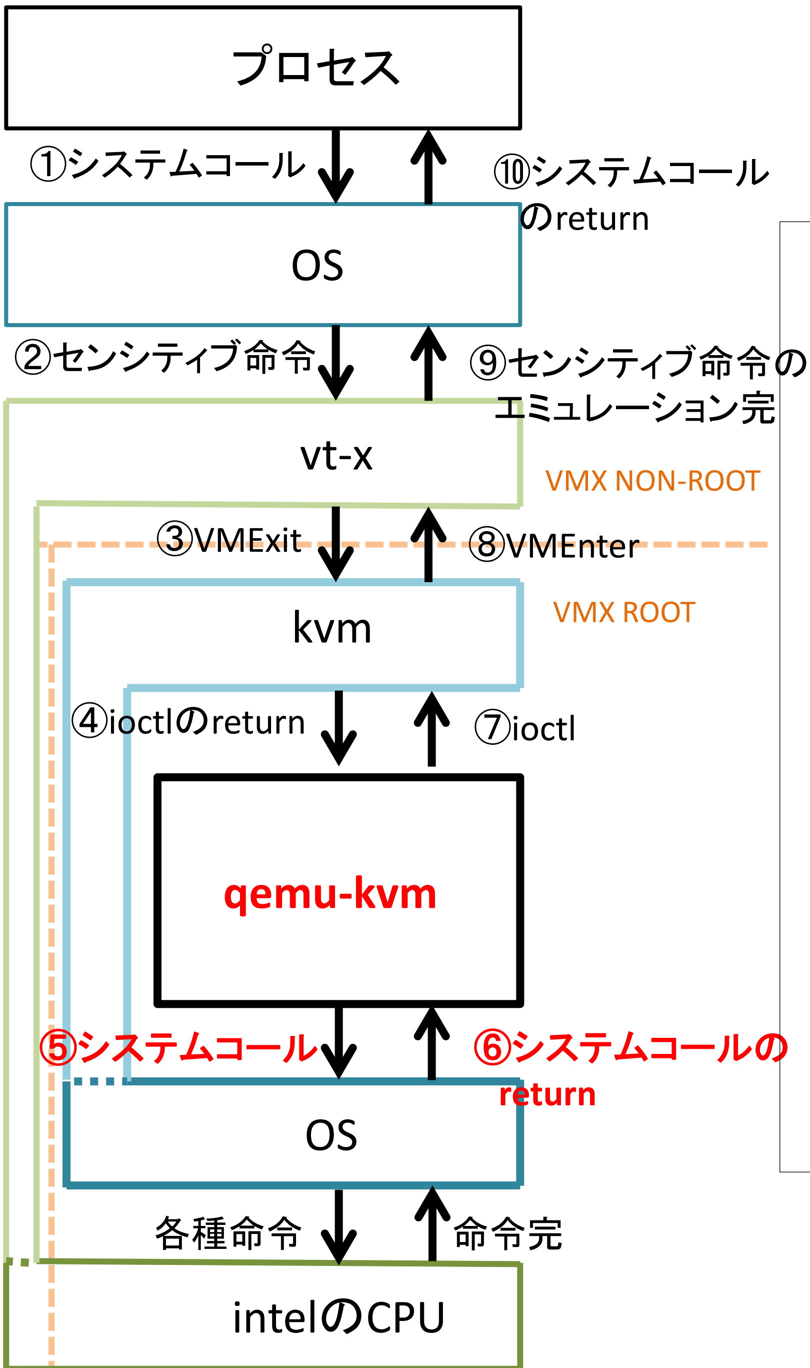
```
struct kvm_run *run = env->kvm_run;
(中略)
post_kvm_run(kvm, env);   /* envにreturn要因等を入れる */

switch (run->exit_reason) {
    case KVM_EXIT_UNKNOWN:
      r = handle_unhandled(run->hw.hardware_exit_reason);
      break;
    case KVM_EXIT_FAIL_ENTRY:
      r = handle_failed_vmentry(run->fail_entry.hardware_entry_failure_reason);
      break;
    case KVM_EXIT_EXCEPTION:
      fprintf(stderr, "exception %d (%x)¥n", run->ex.exception,
          run->ex.error_code);
      kvm_show_regs(env);
      kvm_show_code(env);
      abort();
      break;
    case KVM_EXIT_IO:
      r = kvm_handle_io(run->io.port,
                (uint8_t *)run + run->io.data_offset,
                run->io.direction,
                run->io.size,
                run->io.count);
      r = 0;
      break;
    case KVM_EXIT_DEBUG:
      r = handle_debug(env);
      break;
    case KVM_EXIT_MMIO:
      r = handle_mmio(env);
      break;
```

kvm_handle_io()



- ポート番号からデバイスを検索
  - 今回はvirtio-blkのデバイスが見つかる
- デバイスごとにエミュレーション処理
  - エミュレーションの結果、イメージファイルに対する書き込み

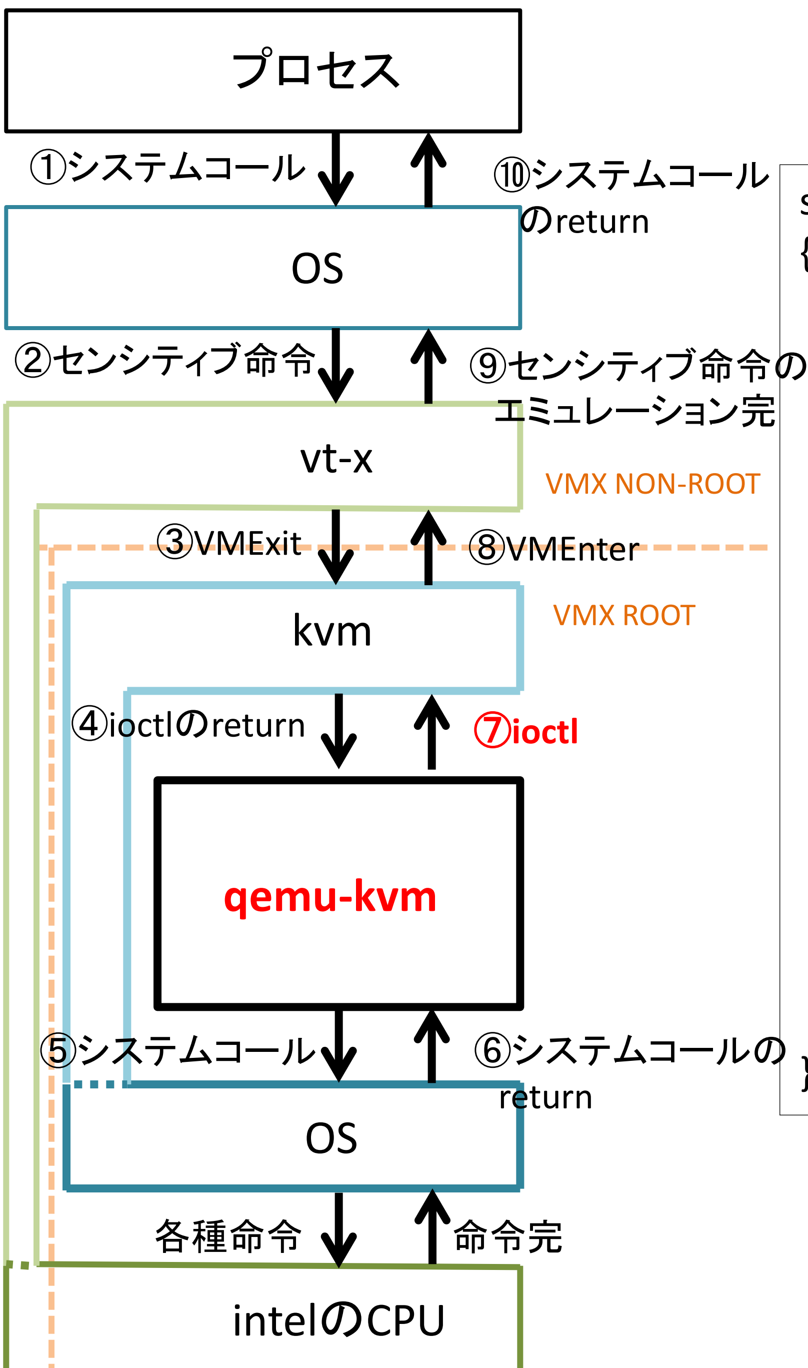
kvm_handle_io()



laio_submit() 抜粋

```
switch (type) {
  case QEMU_AIO_WRITE:
    io_prep_pwritev(iocbs, fd, qiov->iov, qiov->niov, offset);
  break;
  case QEMU_AIO_READ:
    io_prep_preadv(iocbs, fd, qiov->iov, qiov->niov, offset);
  break;
  default:
    fprintf(stderr, "%s: invalid AIO request type 0x%x.¥n",
                    __func__, type);
    goto out_free_aiocb;
  }
  io_set_eventfd(&laiocb->iocb, s->efd);
  s->count++;

  if (io_submit(s->ctx, 1, &iocbs) < 0)
    goto out_dec_count;
  return &laiocb->common;
```

- ライブラリ関数だが、いずれシステムコールにたどり着く

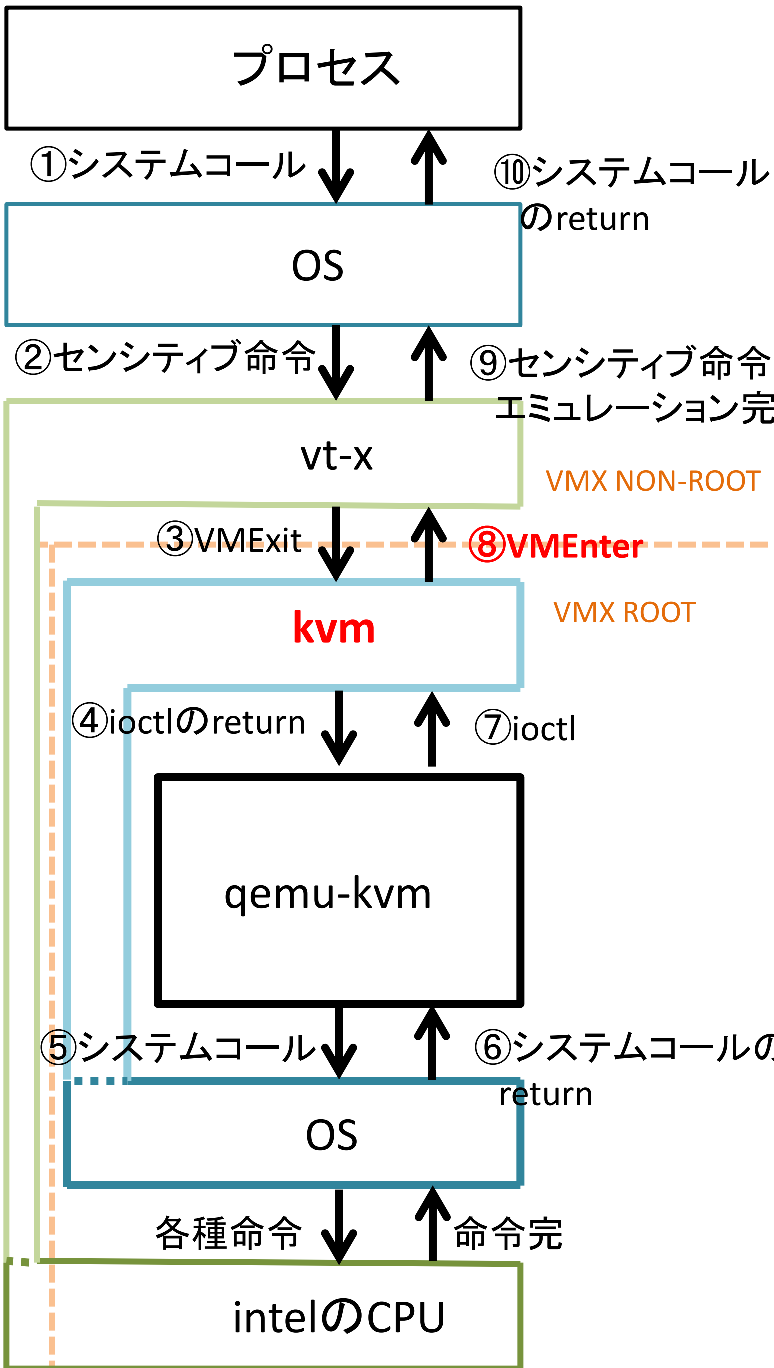


⑦ioctl

kvm_main_loop_cpu() 抜粋

```
static int kvm_main_loop_cpu(CPUState *env)
{
    while (1) {
        int run_cpu = !kvm_cpu_is_stopped(env);
        if (run_cpu && !kvm_irqchip_in_kernel()) {
            process_irqchip_events(env);
            run_cpu = !env->halted;
        }
        if (run_cpu) {
            kvm_cpu_exec(env);
            kvm_main_loop_wait(env, 0);
        } else {
            kvm_main_loop_wait(env, 1000);
        }
    }
    pthread_mutex_unlock(&qemu_mutex);
    return 0;
}
```

- kvm_cpu_exec();
  - kvm_run();
    - **ioctl(fd, KVM_RUN, 0);**



⑧VMEnter

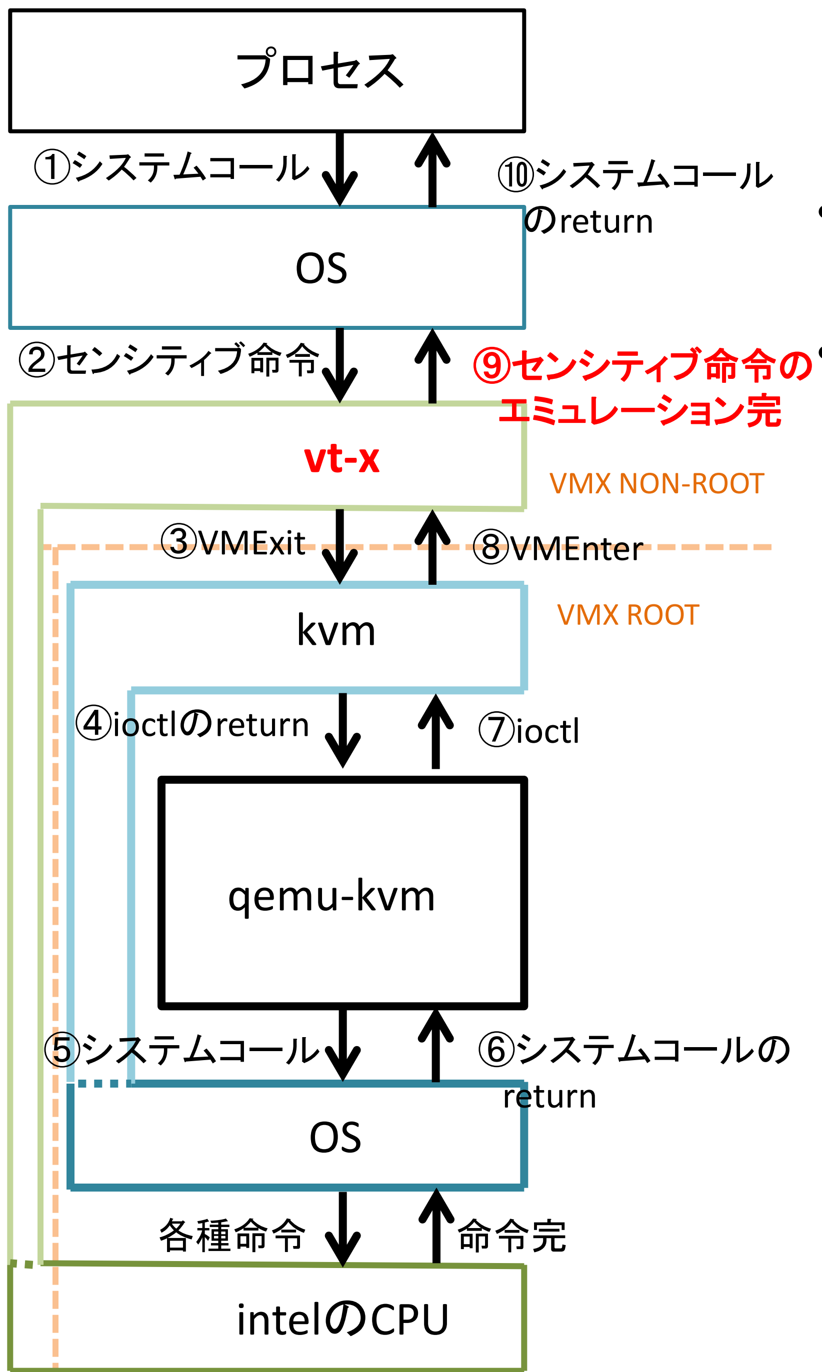
- ioctl されたkvmはVM Enterする。

vmx_vcpu_run(struct kvm_vcpu *vcpu) 抜粋

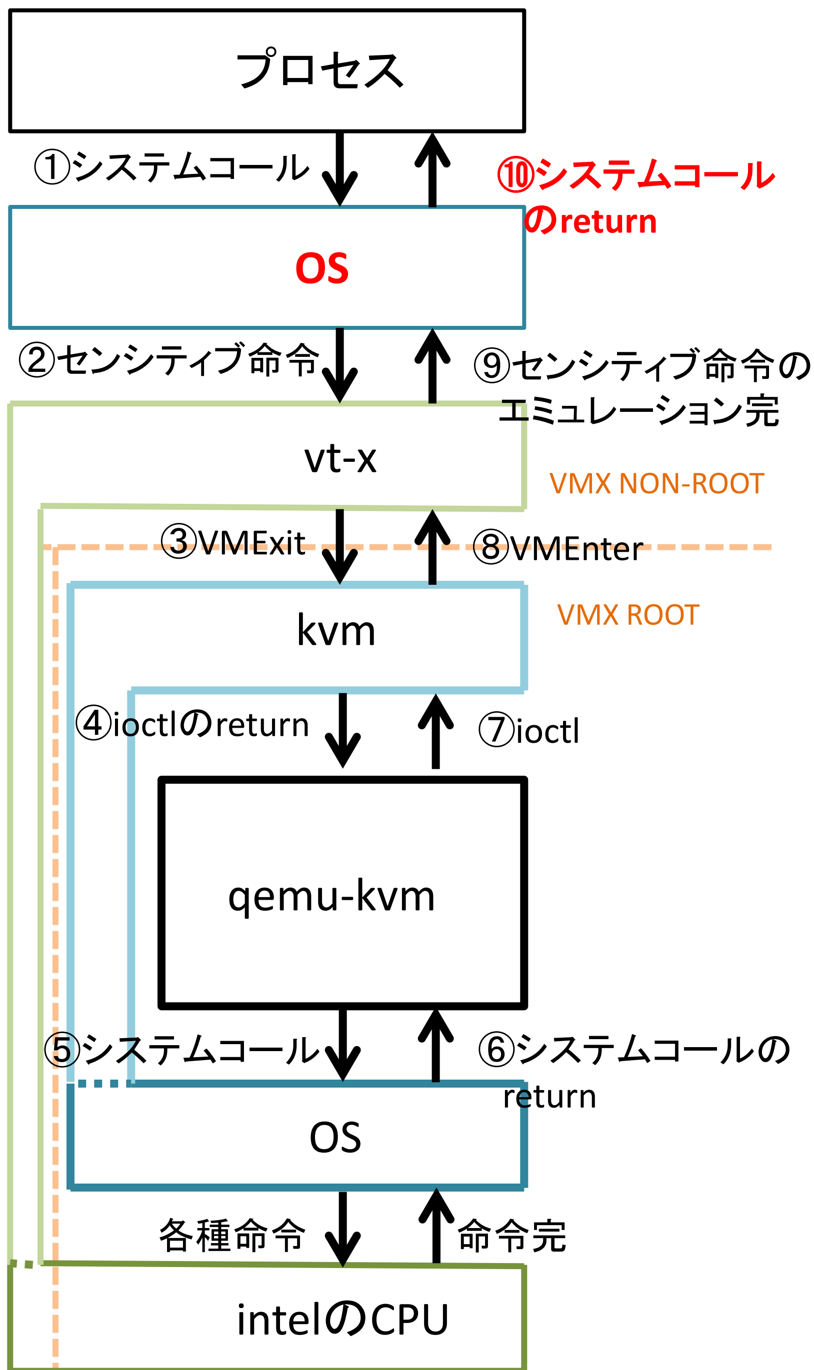
```
/* Enter guest mode */
    "jne .Llaunched ¥n¥t"
    __ex(ASM_VMX_VMLAUNCH) "¥n¥t"
    "jmp .Lkvm_vmx_return ¥n¥t"
    ".Llaunched: " __ex(ASM_VMX_VMRESUME) "¥n¥t"
    ".Lkvm_vmx_return: "
```

- VMLAUNCHとVMRESUME合わせてVM Enterと呼ぶ
  - 最初の起動はVMLAUNCHで
  - 2回目以降はVMRESUMEで

⑨センシティブ命令のエミュレーション完



- ようやく②のI/O PORTへのoutが終わったことになる。
- OSは次の命令を実行することができるようになる。



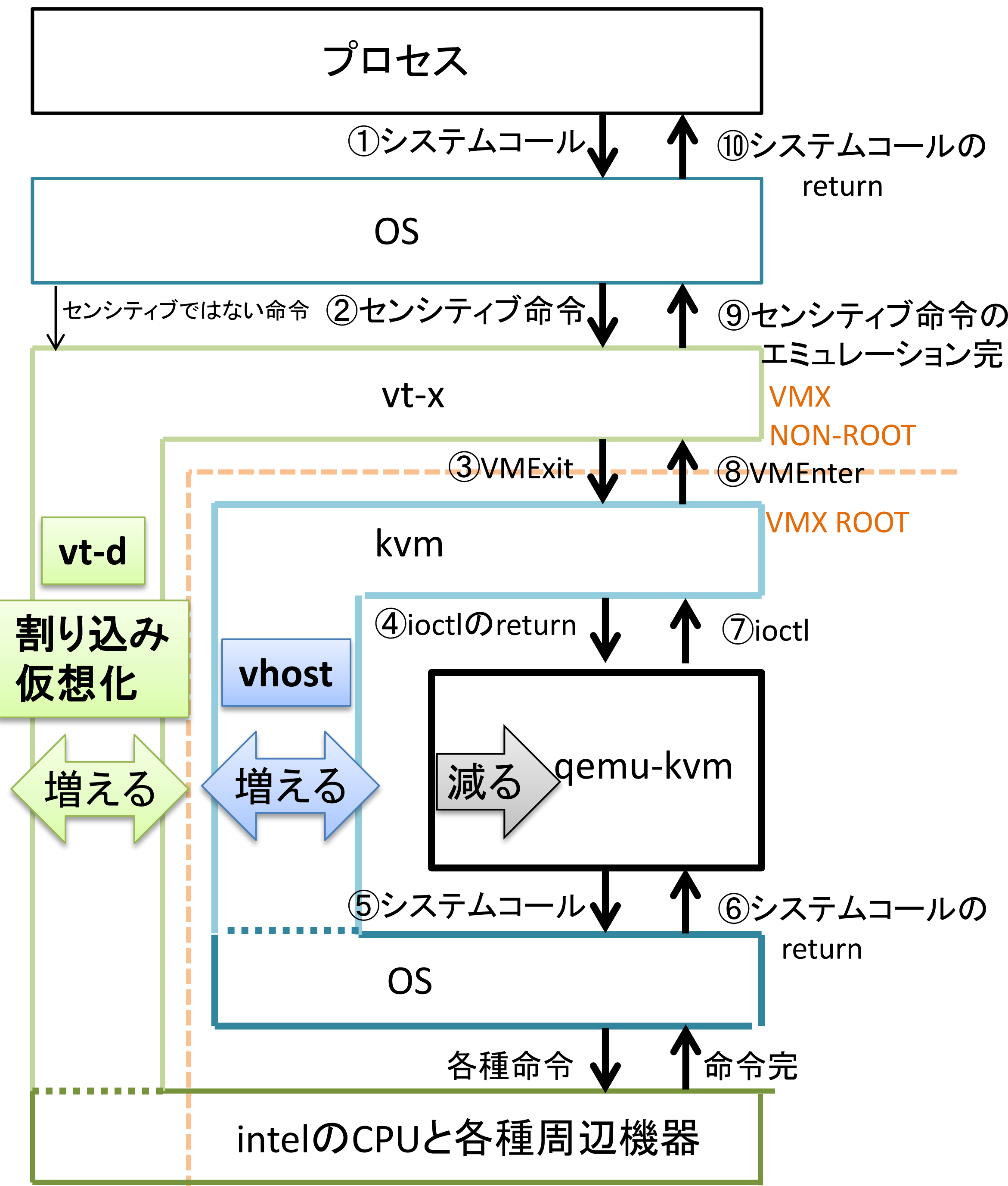
⑩システムコールのreturn

- おつかれさまでした。

qemuとkvmとvt-x

- qemu-kvm
  - エミュレータ。ただし、命令を全部エミュレーションするのはオーバヘッドが大きいため、vt-xとkvmの力を借りて、なるべくエミュレーションしないように改造されているQEMU。
  - kvmが対処できない命令をエミュレートする。
- kvm
  - カーネルモジュール。qemuが、vt-xを使うようにいろいろな準備をしたり、VMがCPUで直接実行できない命令を使ったのをvt-xが教えてくれた時に頑張ってエミュレーションする。エミュレーションできなければqemu-kvmにお願いする。
- vt-x
  - qemuが作ったVMが、CPUで直接実行するとおかしくなる命令を発行したか監視するハードウェアの仕組み。

最近の仮想化潮流



- qemu-kvmの仕事が減る方向
- 箱を超えるためにはメモリのコピーやringの切り替えなどCPUをたくさん使う
- ハードウェアやkvmで仕事をすると箱を超える回数が減少するので高速化できる　はず
  - vhost
  - vt-d
  - 割り込み仮想化　など

商標など

- Intelは，アメリカ合衆国およびその他の国におけるIntel Corporationの商標です。
- Linuxは，Linus Torvalds氏の日本およびその他の国における登録商標または商標です。