test

目次

第 1 章

はじめに

前回 OpenStack に関する本を書いたのだが、これが意外にも完売してしまった。今回は KVM と QEMU とコンテナについて書くことにした。前回からスタッフは総入れ替えとなってしまった。みんな忙しいのだ。僕らと違って。

前から KVM について書きたいと思ってはいたが、成就していなかった。理由は簡単だ。書けるほど詳しくないし、勉強もしていないからだ。これはまぁ、今でもそうかもしれない。

KVM や QEMU の開発動向を見ていると、開発者達はいかにハードウェアとゲストを近づけるか、に苦心しているように思われる。いかにコンテクストスイッチングを減らすか、いかにユーザー・カーネル空間の遷移を減らすか、以下にハードウェアの占有を安全に行うか、云々。それだったらブレード使えば、と言った感じである。

第 2 章

QEMU/KVM 超入門

KVM について知りたい人がいたとして、その人が知りたいのは恐らく KVM ではなく QEMU だろう。QEMU のシステムエミュレーターの事を言っているのだろう。

この章では、実は KVM は、特に何もしていない、というのを解説しようと思う。メモリの割り当て？ネットワーク接続？ USB?ディスク？それは KVM ではなく QEMU で語るものだ。皆が何となく「ハイパーバイザ」と呼ぶ物、それは正確には「仮想マシン」と呼び、それは QEMU によって実現される。KVM は QEMU の仮想 CPU 高速化モジュールとしてとらえてしまっても、特に誤解はない。そして、これを正確には「ハイパーバイザ」と呼ぶ。

KVM と QEMU の違いをはっきりさせないと、有らぬ誤解を生むばかりか、適切な情報を検索することもままならない。この 2 つは似て非なるものではなく、はっきり言って、特に依存はない。KVM が無くても QEMU は動く。実は KVM なんて影も形もない頃から、QEMU は比較的安定して動作していた。あえて言ってしまうと、むしろ KVM の方が、存在意義という観点から、QEMU に依存している。

勉強し始めた時、僕はまず KVM から勉強を始めた。そこから vhost や virtio をいったものを調べ始めた。が、どうしてもよく分からない。最近ようやく理解のとっかかりをつかみ始めたが、ここに至ってようやく、最初から QEMU を勉強しておけば迷いにくいことに気付いた。というわけで、この本では QEMU の解説からしていくことにする。

2.1 QEMU

フォン・ノイマンという人が提唱したノイマン型コンピュータの仕組みは、何の改善もないまま、現在のコンピュータに使われている。基本的には、こうだ。コンピュータは以下の要素で成り立つ

- 演算装置（CPU）
- 記憶装置（RAM, Disk）
- 出力装置（Tape, Display）
- 入力装置（マウス・キーボード）

記憶装置に書き込まれたプログラム通りに、記憶装置に一時的な計算結果を書き込みつつ、演算装置が計算を行う。可変パラメータは入力装置から得られるし、計算結果は出力装置に送られる。言ってしまえば、コンピュータはこれだけのことしかしない。CPU にはレジスタもある、という反論もあるが、それは要するに記憶装置だ。ネットは？それはよく訓練された入出力装置に過ぎない。まとめてしまえばたったこれだけでコンピュータは動作する。

そして、驚くべきことに、QEMU を知る上での前提知識は以上だ。QEMU とは、つまりこの各装置をソフトウェアでエミュレートしているだけなのだ。

2.1.1 CPU のエミュレート

KVM の話をする時に必要になるのが、CPU の仮想化だ。CPU が解釈できる様々な演算命令をすべてエミュレートすれば、CPU の仮想化ができる。QEMU は様々な CPU の命令をエミュレートできる。ARM も x86 も認識できる。

中でやっていることは単純だ。実行するべき命令をホストの CPU 命令に書き換えて実行する。x86 の CPU で動作するホストで ARM の命令をエミュレートするには、ARM の命令を x86 の命令に変換すればよい。CPU でできることなどたかがしれているから、ほぼ一対一で対応がある。対応がない場合は、複数の命令を組み合わせて代替する。

で、問題は x86 の命令をエミュレートして x86 の命令に書き換える時だ。これはエミュレートする必要がない。直接ホストの x86 の命令を実行すればよい。双方同じ CPU モデルなのだから、同じ命令がそのまま実行できるはずだ。そっちの方が変換の手間がなくなるから実行が高速になるが、それ以前に、そもそも無駄だ。

のだが、ここで問題が起きる。実は、そのまま実行できない命令があるのだ。これは CPU が悪いのでも、QEMU のエミュレートが悪いわけでもない。実行できないのは、OS が実行を制限してしまうからだ。

2.1.2 リングプロテクション

実はすべての CPU 命令をユーザーが実行できるわけではない。OS しか実行できないような強力なコマンドは、OS のユーザーは実行することができない。これを OS の「リングプロテクション」と言い、これら特別な命令群を「Ring 0」とか「センシティブな命令」と読んだりするが、要するに OS のような特別な奴にしか実行が許されていない命令群があるのだ。Linux で言うならば、カーネル空間でしか実行できない命令群である。

ここで常に念頭において置かねばならないが、当たり前で忘れがちなことがある。それは、エミュレーターである QEMU は「ユーザー空間のプログラムである」ということである。QEMU がユーザー空間で動作するソフトウェアであることが、脈々と発展し続ける KVM 開発の原動力となる。すべての周辺技術はこの当たり前の事実との戦いの歴史である。

つまり、確かに QEMU は x86 の命令をすべて解釈し実行できる能力はあるが、センシティブな命令は QEMU が動作している OS により実行が禁止される。つまり、実行できない。実行しようとした場合はどうなるか分からない。普通なら、強制的に Kill されるだろう。ホストから見れば、完全に「悪意のあるプログラム」にしか見えないからだ。

センシティブな命令を実行するには、システムコールを使うか別のセンシティブでない命令群で置き換えるかしなければならない。さて、今 QEMU は命令をホストの CPU で直接実行している。センシティブな命令が実行されようとしていることを知るにはどうすれば？この瞬間を QEMU がキャッチできなければ、QEMU はプロセスごと Kill されることになる。

長かったが、ここで KVM が現れる準備が整ったことになる。

2.2 KVM

さて、前章をもってすれば KVM の解説は単純を極める。KVM とは、センシティブな命令が実行されようとすしていることを、ユーザー空間のプログラムに教える機能を持つ、カーネルモジュールである。

知りたいユーザー空間のプログラムは/dev/kvm をポーリングしていると、KVM が教えてくれる。KVM はそれしかしない。今の KVM はいかに「本当にそれだけのことをする」ように機能を削り、磨かれているところである。

virtio

一般にはパラバーチャルドライバとか準仮想化ドライバとか言われたりする。何かそう聞くとマニアックっぽいんだけど、シンプルに言うなら仮想環境用のデバイスドライバ群である。正確に言うと、仮想化環境用のデバイスドライバを書く際に使う、フレームワークである。ここで言うフレームワークとは、ウェブアプリケーションフレームワークのフレームワークと同じ意味である。つまり、便利なライブラリ群、のようなものである。

原理はとても簡単である。ゲストのデバイスドライバと QEMU の間にキューを作り、データの受け渡しを一旦キューで持つ。キューはリングだ。教科書を開くと出てくる。

なんでこんなことをすると幸せになるか。たとえばネットワークのパケットをゲストのデバイスドライバが放出したとする。これを QEMU が一旦キャッチして、システムコールを用いてホストに送るのだが、一パケットごとに毎回システムコールなんぞしていたらたまらない。さすがに遅すぎる。というわけで、一旦キューに貯めて、まとめて送受信する。

ネットワークだけではなく、メモリやディスクアクセスも virtio 経由にできる。lsmodvirtio を探してみると、virtio_scsi や virtio_balloon 何かが見えると思う。virtio とはフレームワークのことである、というのが分かってもらえると思う。

vhost

前の話に続くが、virtio とは言え、毎回 QEMU が介在するのも意味のない話だ。いや、意味はあって、カーネルがちゃんとスケジューリングできるとか、ユーザー空間だからセキュリティか堅牢とかがある。が、それでも QEMU を介在させないようにしたい人達がいて、その人達が作ったのが vhost である。話し始めると、KVM とか irqfd とか eventfd とかいろいろ出てくるんだけど、それはこの際置いておくとする。

vhost とは、ゲストとホストカーネルとの間の共有メモリ上に virtio のキューをつくり、ホストカーネルが直接キューを処理するカーネルモジュールだ。うん、まぁそっちの方が早いだろうね、という感じ。キューへの出入りイベントをゲストとホストで伝達しあうのに、irqfd と eventfd を使う。

2.3 QEMU/KVM

libvirt 経由で起動した QEMU による仮想マシンの引数をみてみよう。

```
/usr/bin/qemu-system-x86_64
-machine accel=kvm
-name instance-00000001
-S
-machine pc-i440fx-2.3,accel=kvm,usb=off
-m 512
-realtime mlock=off
-smp 1,sockets=1,cores=1,threads=1
-uuid c4a02e50-30de-4162-b847-71d65a1942d3
-smbios type=1,manufacturer=OpenStack Foundation,product=OpenStack Nova,
        version=12.0.0,serial=1f04f737-4d3f-42df-ac64-29be9553015d,
        uuid=c4a02e50-30de-4162-b847-71d65a1942d3,family=Virtual Machine
-no-user-config
```

```
-nodefaults
-chardev socket,id=charmonitor,
         path=/var/lib/libvirt/qemu/instance-00000001.monitor,
         server,nowait
-mon chardev=charmonitor,id=monitor,mode=control
-rtc base=utc,driftfix=slew
-global kvm-pit.lost_tick_policy=discard
-no-hpet
-no-shutdown
-boot strict=on
-kernel /opt/stack/data/nova/instances/c4a02e50-.../kernel
-initrd /opt/stack/data/nova/instances/c4a02e50-.../ramdisk
-append root=/dev/vda console=tty0 console=ttyS0
-device piix3-usb-uhci,id=usb,bus=pci.0,addr=0x1.0x2
-drive file=/opt/stack/data/nova/instances/c4a02e50-.../disk,
       if=none,id=drive-virtio-disk0,format=qcow2,cache=none
-device virtio-blk-pci,csi=off,bus=pci.0,addr=0x4,drive=drive-virtio-disk0,
        id=virtio-disk0,bootindex=1
-drive file=/opt/stack/data/nova/instances/c4a02e50-.../disk.config,
       if=none,id=drive-ide0-1-1,readonly=on,format=raw,cache=none
-device ide-cd,bus=ide.1,unit=1,drive=drive-ide0-1-1,id=ide0-1-1
-netdev tap,fd=24,id=hostnet0,vhost=on,vhostfd=25
-device virtio-net-pci,netdev=hostnet0,
        id=net0,mac=fa:16:3e:51:76:0c,bus=pci.0,addr=0x3
-chardev file,id=charserial0,
         path=/opt/stack/data/nova/instances/c4a02e50-.../console.log
-device isa-serial,chardev=charserial0,id=serial0
-chardev pty,id=charserial1
-device isa-serial,chardev=charserial1,id=serial1
-vnc 127.0.0.1:0
-k en-us
-device cirrus-vga,id=video0,bus=pci.0,addr=0x2
-device virtio-balloon-pci,id=balloon0,bus=pci.0,addr=0x5
-msg timestamp=on
```

長い！「そんな書き方できるんだ……」というほど複雑なオプションの組み方をしている。

まずは KVM によるアクセラレーションの有効化である。

```
-machine accel=kvm
```

デバイスは PCI として接続される

ディスクに関してのオプションは

```
-drive file=/opt/stack/data/nova/instances/c4a02e50-.../disk,
       if=none,id=drive-virtio-disk0,format=qcow2,cache=none
-device virtio-blk-pci,csi=off,bus=pci.0,addr=0x4,drive=drive-virtio-disk0,
        id=virtio-disk0,bootindex=1
```

である。QCOW2 フォーマットのディスクを virtio のブロックデバイスとして使う、という指定がなんとなく分かると思う。

ネットワークインターフェースは

```
-netdev tap,fd=24,id=hostnet0,vhost=on,vhostfd=25
-device virtio-net-pci,netdev=hostnet0,
```

```
id=net0,mac=fa:16:3e:51:76:0c,bus=pci.0,addr=0x3
```

vhost を使用しているのが分かる。

ディスクもネットワークインターフェースも、デバイスは PCI として接続されるので、PCI のアドレスも指定されている。また、USB や CD ドライブも接続されている。

メモリに関しては

```
-device virtio-balloon-pci,id=balloon0,bus=pci.0,addr=0x5
```

となっている。つまりバルーニングができるように、virtio_balloon ドライバを使用するオプションが付いている。

かなり見にくいオプション列だが、ちゃんと見ていくと、前述したようなコンピュータが動作するために必要な装置が 1 つずつ指定されていることが分かる。

2.3.1 libvirt

これらのオプションをすべて手で指定するのはかなり骨が折れる。そこで、libvirt のような汎用ライブラリを用いる。上記のオプションは、OpenStack が Libvirt 経由で起動した仮想マシンの引数である。

2.4 QEMU Monitor

QEMU にはユーザーモードとシステムエミュレーターモードがある。前者はバイナリ変換をするだけなのだが、KVM の事を話すのであれば後者のシステムエミュレーターモードについて考えねばなるまい。ネットワークやディスクなどをエミュレートしてくれるのは後者のモードである。

QEMU がどの様に動いているのかを見るには、QEMU Monitor を使う。これを使うと外部、つまりホストマシンから、仮想マシンの様々な情報を取得することができる。QEMU は仮想マシンの全てを知っているはずだから、取得できる情報はかなり多い。ホストとゲスト間のメモリのマッピングを当然のこと、ゲストの CPU のレジスタの内容まで見えてしまう。

この QEMU モニター、できることが多くて意外に楽しいのだが、いまいち日の目を浴びていないように思われる。いや、浴びているのかもしれないけど、コンテナ界隈の人とか OpenStack の人とか、なんかそういう華やかな舞台にどうにも現れないきらいがある。ということで、ドキュメントがあまりない。使い方自体は簡単なので、是非いじって欲しい機能である。

2.4.1 qemu-monitor-command –hmp

この QEMU モニター、どう使うのかというと、QEMU の画面で F2（または Alt-F2）を押す。すると、画面が変わって QEMU モニターを操作するコマンドを受け付けるプロンプトが出現する。あとは help とか打ってみよう。

なんのこっちゃって？そりゃそうだ。QEMU に画面があるって意外に知られていないのではないかと思う。百聞は一見にしかず。ためしにその端末で qemu-system-x86 とか打ってみよう。X Window System を使っている人なら何か黒い画面が出てくると思う。これが QEMU の GUI である。新しいディストリビューションを使っている人はメニューバーなんかも出てくると思う。なんて言っている間に、QEMU は一旦 iPXE での起動を試し、それに失敗して今、起動できないという旨のメッセージが表示されているのではないか。普段 libvirt 経由で仮想マシンが動作しているときは、GUI を使わずにバックグラウンドで起動するようになっている。なので画面は出ない。

この QEMU モニターはソケット経由で操作できる。GUI がなくてもそのソケットに接続すれば QEMU モニターが使える。どうせ libvirt で仮想マシンを建てているだろうから、QEMU のモニターを直接使わずに、libvirt を経由することにしよう。実はそちらの方が見やすいし、大抵の場合は簡単だ。virsh に qemu-monitor-command というコマンドがあるので、使ってみよう。

```
virsh qemu-monitor-command --hmp instance-00000001 help
```

2.5 QEMU/KVM Internal

QEMU が KVM アクセラレーションを有効にして仮想マシンを作成する部分である。#ifdef やエラーチェックなどを除いてシンプルにしてある。

```
static int kvm_init(MachineState *ms)
{
    MachineClass *mc = MACHINE_GET_CLASS(ms);
    static const char upgrade_note[] =
        "Please upgrade to at least kernel 2.6.29 or recent kvm-kmod\n"
        "(see http://sourceforge.net/projects/kvm).\n";
    struct {
        const char *name;
        int num;
    } num_cpus[] = {
        { "SMP",          smp_cpus },
        { "hotpluggable", max_cpus },
        { NULL, }
    }, *nc = num_cpus;
    int soft_vcpus_limit, hard_vcpus_limit;
    KVMState *s;
    const KVMCapabilityInfo *missing_cap;
    int ret;
    int i, type = 0;
    const char *kvm_type;

    s = KVM_STATE(ms->accelerator);

    /*
     * On systems where the kernel can support different base page
     * sizes, host page size may be different from TARGET_PAGE_SIZE,
     * even with KVM.  TARGET_PAGE_SIZE is assumed to be the minimum
     * page size for the system though.
     */
    assert(TARGET_PAGE_SIZE <= getpagesize());
    page_size_init();

    s->sigmask_len = 8;

    s->vmfd = -1;
    s->fd = qemu_open("/dev/kvm", O_RDWR);
```

ここで/dev/kvm がオープンされる。

```
    ret = kvm_ioctl(s, KVM_GET_API_VERSION, 0);
```

```
s->nr_slots = kvm_check_extension(s, KVM_CAP_NR_MEMSLOTS);

/* If unspecified, use the default value */
if (!s->nr_slots) {
    s->nr_slots = 32;
}

s->slots = g_malloc0(s->nr_slots * sizeof(KVMSlot));

for (i = 0; i < s->nr_slots; i++) {
    s->slots[i].slot = i;
}

/* check the vcpu limits */
soft_vcpus_limit = kvm_recommended_vcpus(s);
hard_vcpus_limit = kvm_max_vcpus(s);

kvm_type = qemu_opt_get(qemu_get_machine_opts(), "kvm-type");
if (mc->kvm_type) {
    type = mc->kvm_type(kvm_type);
} else if (kvm_type) {
    ret = -EINVAL;
    fprintf(stderr, "Invalid argument kvm-type=%s\n", kvm_type);
    goto err;
}
```

ここまでで KVM のバージョン、メモリのスロット数、CPU 数が確認される。最小限の情報がそろったところで仮想マシンの作成に入る。

```
do {
    ret = kvm_ioctl(s, KVM_CREATE_VM, type);
} while (ret == -EINTR);

s->vmfd = ret;
```

以下はその他の Capability が確認されていく。

```
missing_cap = kvm_check_extension_list(s, kvm_required_capabilites);
if (!missing_cap) {
    missing_cap =
        kvm_check_extension_list(s, kvm_arch_required_capabilities);
}

s->coalesced_mmio = kvm_check_extension(s, KVM_CAP_COALESCED_MMIO);

s->broken_set_mem_region = 1;
ret = kvm_check_extension(s, KVM_CAP_JOIN_MEMORY_REGIONS_WORKS);
if (ret > 0) {
    s->broken_set_mem_region = 0;
}

s->robust_singlestep =
    kvm_check_extension(s, KVM_CAP_X86_ROBUST_SINGLESTEP);
```

```
    s->intx_set_mask = kvm_check_extension(s, KVM_CAP_PCI_2_3);

    s->irq_set_ioctl = KVM_IRQ_LINE;
    if (kvm_check_extension(s, KVM_CAP_IRQ_INJECT_STATUS)) {
        s->irq_set_ioctl = KVM_IRQ_LINE_STATUS;
    }

    kvm_eventfds_allowed =
        (kvm_check_extension(s, KVM_CAP_IOEVENTFD) > 0);

    kvm_irqfds_allowed =
        (kvm_check_extension(s, KVM_CAP_IRQFD) > 0);

    kvm_resamplefds_allowed =
        (kvm_check_extension(s, KVM_CAP_IRQFD_RESAMPLE) > 0);

    kvm_vm_attributes_allowed =
        (kvm_check_extension(s, KVM_CAP_VM_ATTRIBUTES) > 0);
```

eventfd、irqfd、MMIO、メモリ領域のチェックが行われる。ここに至ってようやく、KVM 仮想マシンの初期化が行われる。

```
    ret = kvm_arch_init(ms, s);

    ret = kvm_irqchip_create(ms, s);

    kvm_state = s;
    memory_listener_register(&kvm_memory_listener, &address_space_memory);
    memory_listener_register(&kvm_io_listener, &address_space_io);

    s->many_ioeventfds = kvm_check_many_ioeventfds();

    cpu_interrupt_handler = kvm_handle_interrupt;

    return 0;
}
```

VMEXIT のハンドリング部分

```
int kvm_cpu_exec(CPUState *cpu)
{
    struct kvm_run *run = cpu->kvm_run;
    int ret, run_ret;

    DPRINTF("kvm_cpu_exec()\n");

    if (kvm_arch_process_async_events(cpu)) {
        cpu->exit_request = 0;
        return EXCP_HLT;
    }

    qemu_mutex_unlock_iothread();
```

ここまでが初期化で、以降 VMEXIT のたびに EXIT 理由に応じてハンドリングがなされる。

```
do {
    MemTxAttrs attrs;

    if (cpu->kvm_vcpu_dirty) {
        kvm_arch_put_registers(cpu, KVM_PUT_RUNTIME_STATE);
        cpu->kvm_vcpu_dirty = false;
    }

    kvm_arch_pre_run(cpu, run);
    if (cpu->exit_request) {
        DPRINTF("interrupt exit requested\n");
        /*
         * KVM requires us to reenter the kernel after IO exits to complete
         * instruction emulation. This self-signal will ensure that we
         * leave ASAP again.
         */
        qemu_cpu_kick_self();
    }

    run_ret = kvm_vcpu_ioctl(cpu, KVM_RUN, 0);

    attrs = kvm_arch_post_run(cpu, run);
```

めでたく RUN される。VMEXIT するまでブロック。

```
    trace_kvm_run_exit(cpu->cpu_index, run->exit_reason);
    switch (run->exit_reason) {
    case KVM_EXIT_IO:
        DPRINTF("handle_io\n");
        /* Called outside BQL */
        kvm_handle_io(run->io.port, attrs,
                      (uint8_t *)run + run->io.data_offset,
                      run->io.direction,
                      run->io.size,
                      run->io.count);
        ret = 0;
        break;
    case KVM_EXIT_MMIO:
        DPRINTF("handle_mmio\n");
        /* Called outside BQL */
        address_space_rw(&address_space_memory,
                         run->mmio.phys_addr, attrs,
                         run->mmio.data,
                         run->mmio.len,
                         run->mmio.is_write);
        ret = 0;
        break;
    case KVM_EXIT_IRQ_WINDOW_OPEN:
        DPRINTF("irq_window_open\n");
        ret = EXCP_INTERRUPT;
        break;
    case KVM_EXIT_SHUTDOWN:
```

```
            DPRINTF("shutdown\n");
            qemu_system_reset_request();
            ret = EXCP_INTERRUPT;
            break;
        case KVM_EXIT_UNKNOWN:
            fprintf(stderr, "KVM: unknown exit, hardware reason %" PRIx64 "\n",
                    (uint64_t)run->hw.hardware_exit_reason);
            ret = -1;
            break;
        case KVM_EXIT_INTERNAL_ERROR:
            ret = kvm_handle_internal_error(cpu, run);
            break;
        case KVM_EXIT_SYSTEM_EVENT:
            switch (run->system_event.type) {
            case KVM_SYSTEM_EVENT_SHUTDOWN:
                qemu_system_shutdown_request();
                ret = EXCP_INTERRUPT;
                break;
            case KVM_SYSTEM_EVENT_RESET:
                qemu_system_reset_request();
                ret = EXCP_INTERRUPT;
                break;
            default:
                DPRINTF("kvm_arch_handle_exit\n");
                ret = kvm_arch_handle_exit(cpu, run);
                break;
            }
            break;
        default:
            DPRINTF("kvm_arch_handle_exit\n");
            ret = kvm_arch_handle_exit(cpu, run);
            break;
        }
    } while (ret == 0);

    qemu_mutex_lock_iothread();

    if (ret < 0) {
        cpu_dump_state(cpu, stderr, fprintf, CPU_DUMP_CODE);
        vm_stop(RUN_STATE_INTERNAL_ERROR);
    }

    cpu->exit_request = 0;
    return ret;
}
```

KVM から EXIT が出力される部分は散らばっていて困るが、例えば KVM_EXIT_IO を放出する部分は arch/x86/kvm/x86.c にある

```
static int emulator_pio_in_out(struct kvm_vcpu *vcpu, int size,
                               unsigned short port, void *val,
                               unsigned int count, bool in)
{
```

```
    vcpu->arch.pio.port = port;
    vcpu->arch.pio.in = in;
    vcpu->arch.pio.count  = count;
    vcpu->arch.pio.size = size;

    if (!kernel_pio(vcpu, vcpu->arch.pio_data)) {
        vcpu->arch.pio.count = 0;
        return 1;
    }

    vcpu->run->exit_reason = KVM_EXIT_IO;
    vcpu->run->io.direction = in ? KVM_EXIT_IO_IN : KVM_EXIT_IO_OUT;
    vcpu->run->io.size = size;
    vcpu->run->io.data_offset = KVM_PIO_PAGE_OFFSET * PAGE_SIZE;
    vcpu->run->io.count = count;
    vcpu->run->io.port = port;

    return 0;
}
```

第 3 章

Containers 超入門

3.1 Containers の世界と LXC、そして Docker

3.1.1 昔からあるコンテナ技術

コンテナ技術を取り囲む現在の状況と、それを踏まえた上での LXC と Docker の根本的な違いについて説明したいと思う。Linux Containers(LXC) は、どうやって我々がアプリケーションを動かしスケールさせるかという問題を変化させる可能性を持っている。コンテナ技術は新しいものではない。そして、LXC に関して言うと、追加パッチを Linux Kernel に適用させることなく、vanilla Linux Kernel 上で稼働させることができる。なお、LXC の Version1 は、長期サポートバージョンであり、5 年間サポートされることになる。話が逸れるが、vanilla Linux Kernel とは、Linux 作者の Linus Torvalds 氏がリリースするプレーンな Kernel のことである。それをベースに様々なベンダーが追加で拡張していくのである。また、vanilla という言葉には「普通の、ありきたりな、おもしろみのない」という意味がある。
話を戻そう。

コンテナ技術は、最近登場した新技術ではない。昔から存在し色んな所で採用されている。FreeBSD には Jail があり、Solaris には Zone がある。それに加えて、OpenVZ や Linux VServer のような Containers も存在する。その歴史は、chroot に始まり、FreeBSD Jail を経て、Linux Containers(LXC) に至る。chroot では、大雑把に言って、ディレクトリーツリーの分離を行っていた。プロセスリスト自体は共有するようなモデルである。chroot のユースケースとしては、開発者向けのテスト/ビルド用環境である。FreeBSD Jail では、chroot の機能に加えて、プロセスリストとネットワークスタックも分離 (というか隔離) された。ユースケースとしては、root 権限の一般ユーザへの委譲、またそれに頼る形でのホスティングサービスである。LXC では、リソース管理テーブルを隔離し、cgroups によるシステムリソース (CPU、メモリ、ディスク etc) の制御を行えるようになった。これにより、LXC は、軽量な仮想環境と見なすことができるようになった。

3.1.2 なぜ皆コンテナに騒いでいるのか

コンテナは、ホストシステムからアプリケーションのワークロードを隔離、あるいはカプセル化する。コンテナを、ホスト OS 内にあるアプリケーションが実行されている OS と見なすことができ、かつ、それは Virtual Machine のように振る舞うのである。このエミュレーションは、Linux Kernel それ自体と、様々なディストリビューションとコンテナを使ってアプリケーションを動かすユーザのためにコンテナ用 OS のテンプレートを提供する LXC Project によって、実現されている。このように、Containers 技術が仮想マシンのように振る舞うことが可能になったことが、一気に注目を浴びる原因となったのである。

3.1.3 コンテナの価値は?

コンテナは、アプリケーションをホスト OS から分離、抽象化することで、LXC をまたぐシステム間でのポータビリティをもたらす。また、コンテナは、ハードウェアをエミュレートすることなく、cgroups と namespaces を駆使して Linux Kernel 内でベアメタルに近いスピードの軽量な OS 環境を実現している。シンプルで高速、かつハードウェアの仮想化よりもよりポータビリティーがありスケールしやすいという構造により、コンテナは、根本的なユーザのワークロードやアプリケーションの仮想化の方法を変えるものなのである。なお、ここでいうポータビリティーとは Docker によりもたらされる、どこでも同一のアプリケーションを稼働することができるという意味ではない。

3.1.4 LXC

LXC プロジェクトは、コンテナ用の OS テンプレートとライフサイクル管理のための幅広いツールセットを提供しています。現在、Canonical のサポートのもと、Stephane Graber と Serge Hallyn により開発は主導されています。

LXC は、活発に開発されているがその割にはドキュメントが少ない。特に Ubuntu 以外のディストリビューションで利用する際のドキュメントが欠如しており、多くの機能はまず Ubuntu 上で実装される。他のディストリビューションを利用しているユーザーからしたら、とてもフラストレーションのたまることである。また、ネット上には数多くの誤解を招くような情報があふれ、混乱を招いている状況も少なからずある。広くマーケットで存在感を示している Docker と混同されたり、そもそもの情報の多さが混乱の元となっていたりする。

LXC は、Docker のようなフロントエンドのアプリケーションのためのローレイヤー層なのか、はたまた、Docker が LXC 上に構築されたユーザーフレンドリーなフロントエンドなのか？こういった不確かな情報が広く出回っている。コンテナ技術のメリットを享受するために必ずしも Docker を使う必要はない。Docker はコンテナ利用の一つの選択でしかないのである。

3.1.5 Docker と LXC では何が違うのか

Docker 視点から見た LXC との違いについて説明する。そもそも LXC に対して Docker が提供している機能とは何なのか。まず第一に言われることは、どのようなホスト OS であってもポータブルなデプロイが可能である点である。Docker はアプリケーションをビルドするためのフォーマットが定義されている。この定義をまとめて記述するために Dockerfile ファイルを利用する。この Dockerfile ファイルはビルドでよく使われる makefile ファイル同様に Docker Containers の構成情報をまとめて記述するテキストファイルである。このファイルに記述する定義情報が全ての依存関係をカプセル化しているため、それはどこで実行してもアプリケー

ション実行環境が同一になるのです。LXC のプロセスのサンドボックスもポータビリティーを持っているが、もし LXC のコンフィグをカスタマイズしているとしたら、ネットワークやストレージ、ディストリビューションの違いにより、それは別環境で稼働しない可能性が高くなります。Docker はその全てを抽象化するためどんな環境でも稼働させることができるのである。

Docker について言及するエンジニアは、総じてアプリケーション寄りのエンジニアである。Docker は、軽量な仮想マシンとしての利用というよりもアプリケーションのデプロイに最適化されている。これは Docker 自体の API やデザインの設計思想に反映されている。それとは対照的に、LXC は軽量な仮想マシンとしての利用に注力している。Docker には、git に似たバージョン管理機能が含まれている。バージョン間の diff の取得や Commit、ロールバックが可能となっている。それによって、Containers の変更を誰がどのように行ったのかについての全てのログを追うことができる。

他にも Docker の利点は多く存在するが、主にこのような点により、Docker は、コンテナそのもの対する見方を変えるきっかけを作った。今まで軽量な仮想マシンとして見られていたコンテナ技術をアプリケーションとしてのコンテナとしてエンジニアに再認識させることに成功したのである。

3.2 LXC を使い軽量仮想環境を手に入れよう

LXC の基本的なコマンドを使ったコンテナ操作を，Ubuntu14.04 をベースにした環境を使って説明していきたい。

3.2.1 LXC のインストール

Ubuntu の最新版である Ubuntu 14.04 LTS では、LXC 1.0.7 が lxc というパッケージ名で提供されている。また、Debian 8 (Jessie) では、LXC 1.0.6 のパッケージが提供されている。
インストールは以下のコマンドを叩くだけである。

```
$ sudo apt-get install lxc
```

3.2.2 LXC で仮想環境を立ち上げる

LXC による仮想環境を立ち上げるためには、まずテンプレートと呼ばれる設定ファイルを用いる。デフォルトでメジャーディストリビューションのテンプレートはすでに同梱されているためこちらを利用する。テンプレートは/usr/share/lxc/templates に配置されている。

```
$ ls /usr/share/lxc/templates/
lxc-alpine    lxc-archlinux  lxc-centos  lxc-debian    lxc-fedora  lxc-openmandriva
lxc-oracle  lxc-sshd    lxc-ubuntu-cloud
lxc-altlinux  lxc-busybox    lxc-cirros  lxc-download  lxc-gentoo  lxc-opensuse
lxc-plamo   lxc-ubuntu
```

Ubuntu のテンプレートを用いて test-container-101 という名前の Ubuntu のコンテナを立ち上げる。

```
$ sudo lxc-create -t ubuntu -n test-container-101
```

これでコンテナの root ディレクトリに相当するディレクトリに必要なものがインストールされる。コンテナの場所は以下のディレクトリである。

```
/var/lib/lxcコンテナ名/<>/
```

そこで、test-container-101 の rootfs の中を覗いてみると以下の通りとなる。

```
$ sudo ls -F /var/lib/lxc/test-container-101/rootfs/
bin/  boot/  dev/  etc/  home/  lib/  lib64/  media/  mnt/
opt/  proc/  root/      run/  sbin/  srv/  sys/  tmp/  usr/  var/
```

インストールしたコンテナの起動には以下のコマンドを実行する。

```
$ sudo lxc-start -n test-container-101 -d
```

-d オプションでデーモンとしてコンテナを起動する。この状態で lxc-console コマンドを用いてコンソール接続することでコンテナの内部にログインすることができる。

```
$ sudo lxc-console -n test-container-101
```

コンテナから抜ける際には、Ctrl+A を入力してその後 Q を押す。また、デフォルトのユーザは ubuntu で、パスワードも ubuntu である。インストール時に以下のメッセージが表示されているはずである。

```
# The default user is 'ubuntu' with password 'ubuntu'!
# Use the 'sudo' command to run tasks as root in the container.
```

コンテナの終了は、lxc-shutdown コマンドを実行すればよい。

```
$ sudo lxc-shutdown -n test-container-101
```

3.2.3 コンテナの情報を見る

コンテナに関する情報を見てみよう。lxc-ls というコマンドでホスト上にあるコンテナの情報を確認することができる。–fancy オプションを付けることで、コンテナ名、状態、IPv4 のアドレス、IPv6 のアドレス、自動起動の有無を確認できる。

```
$ sudo lxc-ls --fancy
NAME                STATE    IPV4  IPV6  AUTOSTART
----------------------------------------------------
test-container-101  STOPPED  -     -     NO
```

コンテナ単体の詳細情報については lxc-info というコマンドが提供されている。コンテナが STOPPED した状態のときにはこう表示される。

```
$ sudo lxc-info -n test-container-101
Name:           test-container-101
State:          STOPPED
```

コンテナを起動すると詳細情報が表示される。

```
$ sudo lxc-info -n test-container-101
Name:           test-container-101
State:          RUNNING
PID:            20434
CPU use:        0.77 seconds
BlkIO use:      7.16 MiB
Memory use:     13.53 MiB
KMem use:       0 bytes
Link:           vethABI04E
 TX bytes:      940 bytes
 RX bytes:      592 bytes
 Total bytes:   1.50 KiB
```

このように特定の情報のみを取得することも可能である。

```
$ sudo lxc-info -n test-container-101 -c lxc.utsname -c lxc.rootfs
lxc.utsname = test-container-101
lxc.rootfs = /var/lib/lxc/test-container-101/rootfs
```

3.2.4 LXD とは何か

LXD について本家ページを元に少し説明する。LXD とは Linux Container Daemon の略である。Canonical 主導で開発が進められているコンテナ技術であり、コンテナに今どきのハイパーバイザーの機能を追加するサーバプログラムである。このデーモンは REST API を提供しているのでローカルからだけでなくネットワーク経由でのコンテナの操作が可能である。主要機能は以下の通りである。

- 非特権コンテナ、リソース制限を用いたセキュアなデザイン
- 直感的なコマンドラインと REST API
- イメージベースのコンテナ構築
- ライブマイグレーション

特に、Docker Hub にあるイメージを利用可能になるということがアナウンスされている点が期待できる。

3.2.5 LXD インストール

Ubuntu ユーザは PPA を使って以下の通りインストール可能である。なお、他のディストリビューションのユーザは、最新のリリースの tarball か git リポジトリから直接 LXD をダウンロードしてビルドできる。

```
$ sudo add-apt-repository ppa:ubuntu-lxc/lxd-git-master
$ sudo apt-get update && sudo apt-get -y install lxd
```

3.2.6 LXD イメージのインポート

イメージベースなので、ダウンロードする。なお、LXD のコマンドラインは lxc というコマンドである。何ともややこしい。コンテナイメージのインポートは lxc-images というコマンドを利用する。以下では、Ubuntu14.04 をインポートしている。

```
$ sudo lxd-images import lxc ubuntu trusty amd64 --alias ubuntu
```

以下のコマンドでイメージ一覧を取得できる。

```
$ sudo lxc image list
+--------+--------------+--------+-------------+--------+-------------------------------+
| ALIAS  | FINGERPRINT  | PUBLIC | DESCRIPTION |  ARCH  |          UPLOAD DATE          |
+--------+--------------+--------+-------------+--------+-------------------------------+
| ubuntu | 04aac4257341 | no     |             | x86_64 | Jul 15, 2015 at 1:16pm (UTC) |
+--------+--------------+--------+-------------+--------+-------------------------------+
```

3.2.7 LXD コンテナの起動

lxc launch コマンドで起動できる。

```
$ sudo lxc launch ubuntu test-container-102
Creating container...done
Starting container...done
error: saving config file for the container failed
```

できなかった。。。。どうやらこのバグにヒットしたらしい。(https://github.com/lxc/lxd/issues/739)
改めて起動するとこのようになる。

```
$ lxc list
+--------------------+---------+------------+------+-----------+-----------+
|        NAME        |  STATE  |    IPV4    | IPV6 | EPHEMERAL | SNAPSHOTS |
+--------------------+---------+------------+------+-----------+-----------+
| test-container-103 | RUNNING | 10.0.3.138 |      | NO        | 0         |
+--------------------+---------+------------+------+-----------+-----------+
```

このような感じで LXD を扱えるが、まだまだバグもあり不安定だという印象が強い。もし、興味があれば使ってみてほしい。

第 4 章

あとがき

4.1 こじろー

本体は表紙です。ついでに、中身も流し読みしていただけると嬉しいです。

4.2 まっきー

hogehoge

4.3 だーまり

hogehoge