GiantVM 이해를 위한

QEMU-KVM 가상화 기초 자료

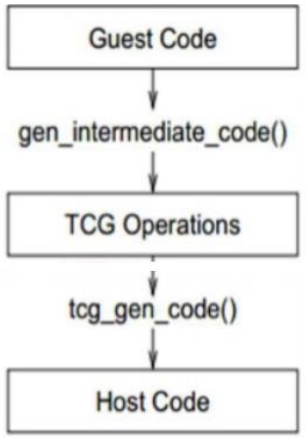
2021-08-09

인용 및 참고 문헌:

- (주 인용/참고 문헌) Mastering KVM Virtualization (*Humble Devassy Chirammal, Prasad Mukhedkar, Anil Vettathu* 저, Packt Publishing)
- (주 인용/참고문헌 번역판) KVM 가상화 완전 가이드 클라우드 컴퓨팅을 위한 리눅스 가상화 시스템 (*공용준* 역, 에이콘출판사)
- Xen으로 배우는 가상화 기술의 이해 (*박은병, 김태훈, 이상철, 문대혁* 저, 한빛미디어)
- QEMU internals: Big picture overview (*Stefan Hajnoczi*, *http://blog.vmsplice.net/2011/03/qemu-internals-big-picture-overview.html*)

QEMU: intro

- QEMU는 "에뮬레이터" 또는 "가상화 장치"로 작동 가능
  - 어떤 머신에서 QEMU가 에뮬레이터로 사용될 때는 다른 머신용으로 만든 OS 및 프로그램을 실행 가능함
  - Binary translation을 사용



Ex. ARM 기반 코드 → x86 코드로 실행하는 경우
1) ARM 바이너리 이미지를 스캔해 역어셈블 과정으로 ARM 명령어 추출
2) 추출된 ARM 명령어를 x86에서 동작할 수 있도록 translation

➔ 에뮬레이터에는 translation 과정이 포함되므로 실제 머신에서 실행하는 것보다 필연적으로 성능 저하 발생함

QEMU has a dynamic **binary** translator (DBT) which converts **binary** code from the guest CPU architecture to the host architecture. For example, the DBT can take x86 guest code and turn it into an equivalent sequence of instructions natively executable on a SPARC host.

QEMU: 작동방식 1 - 에뮬레이터

- Dynamic binary translation 을 통해 CPU를 에뮬레이트하고 장치 모델의 세트를 제공
  ➔ 다른 아키텍처의 수정되지 않은 guest OS 실행 가능

- guest OS가 호스트 CPU에서 실행되어야 하기 때문에 binary translation 이 필요

- 이 작업을 수행하는 바이너리 변환기는 JIT 컴파일러로 알려진 TCG (Tiny Code Generator) → 주어진 프로세서용으로 작성된 바이너리 코드를 다른 프로세서용으로 변환함 (ex. ARM → X86)

참고: TCG

- TCG는 런타임에 QEMU가 수행하는 작업에 컴파일러(코드 생성기)를 동시에 수행

- 전체 변환 작업은 두 부분으로 구성됨
  - TCG 연산에서 다시 쓰여지는 대상 코드 블록(기계에 독립적인 중간 표기법의 일종) 생성 부분
  - TCG에 의해 호스트 아키텍처용으로 컴파일되는 부분

QEMU: 작동방식 2 - 가상화 장치

- QEMU가 호스트 CPU에서 guest 코드를 직접 실행하여 본래의 성능을 얻는 작동 방식
  - Host CPU에서 target architecture를 지원하는 경우에만 가능

- For example, Xen/KVM 에서 동작할 때 QEMU는 이러한 작동 방식을 사용 가능

- QEMU는 binary translation 을 사용해 KVM 없이 실행할 수 있으나, KVM에 의해 활성화 된 하드웨어 가속 가상화보다 느림

QEMU: 작동방식 2 - 가상화 장치 (계속)

- KVM으로 작업할 때 QEMU-KVM은 가상 머신을 생성하고 초기화
- Guest의 각 가상 CPU에 대해 서로 다른 POSIX 쓰레드를 초기화
- QEMU-KVM의 사용자 모드 주소 공간 내에 가상 머신의 물리 주소 공간을 에뮬레이트하는 프레임워크를 제공
- 물리적 CPU에서 guest 코드를 실행하려면, QEMU는 POSIX 쓰레드 사용. 즉, guest 가상 CPU는 호스트에서 posix 쓰레드로 실행됨
- QEMU는 KVM 커널 모듈을 통해 guest 코드를 실행함
- QEMU는 KVM 커널 모듈에 의해 노출된 장치 파일(/dev/kvm)을 열고 ioctls()를 실행
- 결론적으로, KVM은 QEMU를 사용해 완전한 하이퍼바이저가 됨
- KVM은 프로세서가 제공하는 하드웨어 가상화 확장(VMX 또는 SVM)의 가속기임. 일단, 활성화되면 binary translation 류의 기술보다 성능이 우월

QEMU: main data structure

- KVMState 에는 QEMU에서 VM을 대표하는 중요한 파일 디스크립터가 포함됨
- 아래 그림의 예: 가상 머신 파일 디스크립터가 포함

```
struct KVMState
{
    AccelState parent_obj;

    int nr_slots;
    int fd;
    int vmfd;
    int coalesced_mmio;
    struct kvm_coalesced_mmio_ring *coalesced_mmio_ring;
```

QEMU: main data structure (계속)

- QEMU-KVM은 각 가상 CPU의 CPUX86State 구조 목록을 유지
- 범용 레지스터(RSP, RIP 등)의 내용은 CPUX86State의 일부임

```
struct CPUState {
    /*< private >*/
    DeviceState parent_obj;
    /*< public >*/

    int nr_cores;
    int nr_threads;
    int numa_node;
```

```
    int kvm_fd;
    bool kvm_vcpu_dirty;
    struct KVMState *kvm_state;
    struct kvm_run *kvm_run;
```

```
typedef struct CPUX86State {
    /* standard registers */
    target_ulong regs[CPU_NB_REGS];
    target_ulong eip;
    target_ulong eflags; /* eflags register. During CPU emulation, CC
                            flags and DF are set to zero because they are
                            stored elsewhere */
```

```
    uint64_t system_time_msr;
    uint64_t wall_clock_msr;
```

```
    /* exception/interrupt handling */
    int error_code;
    int exception_is_int;
```

QEMU: main data structure (계속)

- 다양한 ioctls(), 즉 kvm_ioctl, kvm_vm_ioctl, kvm_vcpu_ioctl, kvm_device_ioctl 등이 있음 (함수 정의는 **kvm-all.c** 참조)

- ioctls()는 기본적으로 시스템 KVM 수준, VM 수준, vCPU 수준에 매핑됨

- KVM 커널 모듈에 의해 노출된 ioctls()를 액세스하려면 QEMU-KVM이 /dev/kvm을 open 해야 하고, 그 결과 파일 디스크립터가 KVMState->fd에 저장됨

QEMU: main data structure (계속)

- kvm_ioctl(): ioctl()은 주로 KVMState->fd를 매개변수로 사용하고, 이 매개변수는 /dev/kvm을 open한 결과값에서 가져옴
  - Ex. kvm_ioctl(s, KVM_CREATE_VM, type);

- kvm_vm_ioctl(): 이 ioctl()은 주로 KVMState->vmfd를 매개변수로 사용함
  - Ex. kvm_ioctl(s, KVM_CREATE_VCPU, (void *)kvm_arch_vcpu_id(cpu));

- kvm_vcpu_ioctl(): 주로 KVM에 대한 vCPU한 파일 디스크립터인 CPUState->kvm_fd를 매개변수로 자용
  - Ex. kvm_vcpu_ioctl(cpu, KVM_RUN, 0)

- kvm_device_ioctl(): 이 ioctl()은 주로 장치 fd 값을 매개변수로 사용
  - Ex. kvm_device_ioctl(dev_fd, KVM_HAS_DEVICE_ATTR, &attribute)

QEMU: *KVM 기반으로* 가상머신 및 vCPU 생성

- kvm_init() → KVM 장치 파일을 열고 KVMState의 fd, vmfd를 채움

```
static int kvm_init(MachineState *ms)
{
    ....
    KVMState *s;
    s = KVM_STATE(ms->accelerator);
    ...
    s->vmfd = -1;
    s->fd = qemu_open("/dev/kvm", O_RDWR);
     ....
    do {
        ret = kvm_ioctl(s, KVM_CREATE_VM, type);
    } while (ret == -EINTER);
    s->vmfd = ret;

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

(참고) QEMU: kvm-all.c

QEMU: *KVM 기반으로* 가상머신 및 vCPU 생성 (계속)

- QEMU가 fd와 vmfd를 가지면, kvm_fd 또는 vcpu_fd가 채워져야 함

qemu_init_vcpu
→ qemu_kvm_start_vcpu()
→ qemu_thread_create
→ qemu_kvm_cpu_thread_fn()
→ kvm_init_vcpu(CPUState *cpu)

```
int kvm_init_vcpu(CPUState *cpu)
{
   KVMState *s = kvm_state;
   ret = kvm_vm_ioctl(s, KVM_CREATE_VCPU, (void *)kvm_arch_vcpu_id(cpu));
   cpu->kvm_fd = ret;                    → vCPU fd
   …
   mmap_size = kvm_ioctl(s, KVM_GET_VCPU_MMAP_SIZE, 0);
   cpu->kvm_run = mmap(NULL, mmap_size, PROT_READ|PROT_WRITE,
                                   MAP_SHARED, cpu->kvm_fd, 0);
    …
   ret = kvm_arch_init_vcpu(cpu);    → target-i386/kvm.c
   …
}
```

(참고) QEMU: kvm-all.c

- 메모리 페이지의 일부는 QEMU-KVM 프로세스 및 KVM 커널 모듈 사이에서 공유됨
- kvm_init_vcpu()에서 이런 매핑을 볼 수 있음. 즉, vCPU 당 두개의 호스트메모리 페이지가 QEMU 사용자 공간 프로세스와 KVM 커널 모듈 간 통신 채널을 만듦. 이를 kvm_run 및 pio_data라 함.
- 또한, 앞서 fd 값들을 반환하는 ioctls()를 실행하는 동안 리눅스 커널은 파일 구조를 할당하고 이것을 익명 노드와 연관지음

QEMU: *KVM 기반으로* vCPU 실행

- vCPU 쓰레드는 다음 코드와 같이 KVM_RUN을 인수로 사용해 ioctl()을 실행

```
int kvm_cpu_exec(CPUState *cpu)
{
    struct kvm_run  *run = cpu->kvm_run;
    run_ret = kvm_vcpu_ioctl(cpu, KVM_RUN, 0);
    …

    switch(run->exit_reason) {
      case KVM_EXIT_IO:
                DPRINTF("handle_io₩n");
                kvm_handle_io(…);
                break;

      case KVM_EXIT_MMIO:
                DPRINTF("handle_mmio₩n");
                 ….; break;
       …
      case KVM_EXIT_SYSTEM_EVENT:
              switch(run->system_event.type) {
                  case KVM_SYSTEM_EVENT_SHUTDOWN: …; break;
                  case KVM_SYSTEM_EVENT_RESET: …; break;
                  case KVM_SYSTEM_EVENT_CRASH: …; break;
```

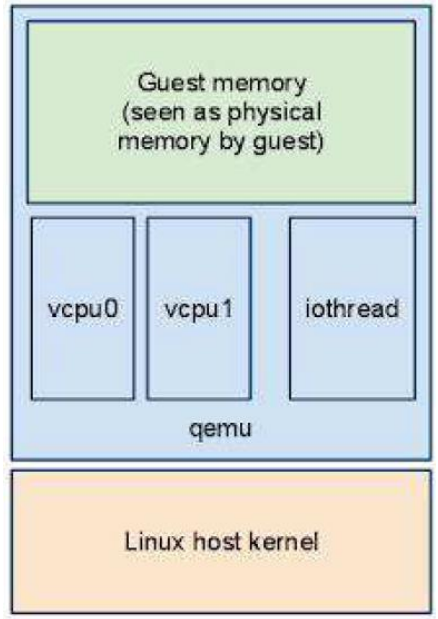
(참고) QEMU: kvm-all.c

QEMU: *KVM 기반으로* vCPU 실행 (계속)

- kvm_cpu_exec()는 KVM이 VM을 종료시키고 제어를 QEMU-KVM 사용자 공간으로 넘길 때 수행해야할 작업을 정의함
- KVM은 인텔, AMD같은 공급업체가 VMX 및 SVM 같은 가상화 확장을 제공하는 하드웨어에서 확장 명령을 사용할 수 있게함
- 확장 기능은 KVM이 호스트 CPU에서 guest 코드를 직접 실행하는데 사용됨
- 하지만, 운영중인 guest 커널 코드가 QEMU에 의해 emulation되는 하드웨어 장치 레지스터에 접근하는 등의 이벤트가 있는 경우 KVM은 종료되고 제어를 QEMU로 넘겨야 함
- 이어서 QEMU는 제어 이벤트의 결과를 에뮬레이트할 수 있음

QEMU: 쓰레드 모델

[그림] 호스트에서 보이는 qemu process
(출처: http://blog.vmsplice.net/2011/03/qemu-internals-big-picture-overview.html)



- 주요 쓰레드
  - Main thread → 각종 이벤트를 event-driven으로 처리하는 전용 iothread
  - 각 vCPU의 thread → guest 코드 실행
  - 가상 디스크 I/O 백엔드의 worker thread

- 각각의 VM은 호스트 시스템에서 실행되는 QEMU 프로세스에 매치. guest 시스템이 종료될 경우 이 프로세스도 삭제/종료됨
  - 참고로, guest의 물리적 RAM은 QEMU 프로세스의 주소 공간 안에 있음

- vCPU 쓰레드와 별도로 네트워크 패킷 및 디스크 I/O 완료와 같이 I/O 처리를 위한 select 이벤트 루프를 실행하는 전용 iothread가 있음 (이벤트 루프 쓰레드)

QEMU: 쓰레드 모델 (계속)

- 이벤트 루프는 타이머, 파일 디스크립터 모니터링 등에 사용됨

- main_loop_wait()는 다음과 같이 정의된 QEMU 메인 이벤트 루프 쓰레드임

```
static void main_loop(void)   ← vl.c
{
    bool nonblocking;
    int last_io = 0;
...
    do {
        nonblocking = !kvm_enabled() && !xen_enabled() && last_io > 0;
        ...
        last_io = main_loop_wait(nonblocking);
        ...
     } while (!main_loop_should_exit());
}
```

- 이러한 메인 이벤트 루프 쓰레드는 파일 디스크립터 콜백, bottom half, 타이머를 포함하는 메인 루프 서비스를 담당
- Bottom half는 즉시 실행되지만 오버헤드가 적은 타이머와 비슷하며 대기 시간, 쓰레드 안전성 및 신호 안전성을 고려한 스케쥴링이 가능

KVM 세부

- KVM은 /dev/kvm이라는 장치 파일을 노출해 응용이 ioctls()을 사용할 수 있게함

- QEMU는 이 장치 파일을 사용해 KVM과 통신하고 가상 머신의 커널 모드 작업을 생성, 초기화, 관리함

- 기타 도움이 될 수 있는 부분들
  - 특정 I/O 장치의 emulation, 예를 들어 CPU 당 로컬 APIC 및 시스템 전체 IOAPIC("mmio"를 통해)
  - 특정 '특권'(시스템 레지스터 CR0, CR3, CR4의 R/W 등) 명령어의 emulation
  - VMENTRY를 통한 guest 코드 실행 및 VMEXIT에서의 '인터셉트 이벤트' 처리
  - 가상머신의 실행 흐름에 대한 가상 인터럽트 및 페이지 오류 같은 이벤트의 '주입 ' 은 KVM의 도움으로 처리

KVM 세부 (계속)

- KVM은 QEMU 같은 프로그램이 호스트 CPU에서 직접 guest 코드를 안전하게 실행할 수 있게 해주는 리눅스 커널의 가상화 기능임

- 타겟 아키텍처가 호스트 CPU에 의해 지원되는 경우에만 가능

- 그러나, KVM은 guest 모드라는 또 하나의 모드를 도입 ➔ guest 모드는 guest 시스템 코드의 실행임. 이 것은 guest의 사용자 또는 커널 코드를 실행할 수 있음

KVM 세부 (계속)

- 가상화 기술 프로세서 도입으로 VMX라는 확장 명령어 세트가 도입됨

- 인텔 VT-x에서 VMM은 “VMX 루트 작동 모드 " 로 실행되고 guest(수정되지 않은 OS)는 “VMX 비루트 작동 모드 " 로 실행됨

- 이 VMX는 VMREAD, VMCALL, VMLAUNCH, VMRESUME, VMXOFF, VMXON 등과 같은 가상화 명령을 CPU에 제공. 가상화 모드(VMX)는 VMXON에 의해 활성화되고 VMXOFF에 의해 비활성화

[참고] 인텔 VT-x의 핵심 부분 도시

(출처: Xen으로 배우는 가상화 기술의 이해(박은병 외 3인 공저))

기존 특권모드와 비특권 모드 외에 추가로 하이퍼바이저를 위한 root 모드와 guestOS를 위한 non-root 모드 출현

- 응용: non-root 모드의 비특권 모드에서 동작
- guestOS: non-root 모드의 특권 모드로 동작
- 하이퍼바이저: root 모드에서 동작

➔ 문제의 소지가 있는 명령어가 non-root 모드에서 실행되면 트랩이 발생하고 root 모드로 변경되어 하이퍼바이저로 제어권이 넘어가도록 하드웨어적으로 구현됨

KVM 세부 (계속)

- guest 코드를 실행하려면 VMLAUNCH/VMRESUME 명령어를 사용하고 VMEXIT를 남겨야 함
  - 이 전환을 하고 나서 나중에 가져올 수 있도록 일부 정보를 저장
  - 인텔은 VMCS라는 이 전환을 용이하게 하는 structure를 제공. 예를 들어, VMEXIT의 종료 이유는 이 structure에 기록. VMREAD나 VMWRITE 명령어가 VMCS 필드를 읽거나 기록하는데 사용됨

- 최신 인텔 프로세서에서 제공되는 기능을 사용하면 각 guest가 메모리 주소를 추적할 수 있는 자체 페이지 테이블을 가질 수 있음. EPT가 없으면 하이퍼바이저는 주소 변환을 수행하기 위해 가상머신을 종료해야 하며 이렇게 하면 성능이 저하됨

KVM: API

- KVM의 API는 가상머신의 다양한 측면을 제어하는 ioctl()의 집합

- ioctl()의 세가지 클래스
  - System ioctls: 전체 KVM 하위 시스템에 영향을 주는 전역 속성을 쿼리하고 설정. 또한, 시스템 ioctl은 가상머신을 만드는데 사용
  - VM ioctls: 이런 쿼리 및 속성은 전체 가상 머신(ex. 메모리 레이아웃)에 영향을 줌. 또한 VM ioctls는 vCPU를 만드는데 사용됨
  - vCPU ioctls: 단일 가상 CPU의 작동을 제어하는 쿼리 및 속성을 지정. 이들은 vCPU를 작성하는 데 사용된 것과 동일한 쓰레드에서 vCPU ioctls을 실행

KVM: main data structure

- kvm-main.c

```
static struct file_operations kvm_chardev_ops = {
        .unlocked_ioctl = kvm_dev_ioctl,
        .compat_ioctl = kvm_dev_ioctl,
        .llseek = noop_llseek,
};

kvm_dev_ioctl(struct file *filp, unsigned int ioctl, unsigned long arg) {
    switch(ioctl) {
        case KVM_GET_API_VERSION:
                ...
        case KVM_CREATE_VM:
                r = kvm_dev_ioctl_create_vm(arg);
                break;
         ....
}
```

```
static struct file_operations kvm_vm_fops = {
        .release = kvm_vm_release,
        .unlocked_ioctl = kvm_vm_ioctl,
         ....
        .llseek = noop_llseek,
};

static struct file_operations kvm_vcpu_fops = {
        .release = kvm_vcpu_release,
        .unlocked_ioctl = kvm_vcpu_ioctl,
         ....
        .mmap = kvm_vcpu_mmap,
        .llseek = noop_llseek,
};
```

- inode 할당

```
static int kvm_dev_ioctl_create_vm(unsigned long type)
{
      ...
      file = anon_inode_getfile("kvm-vm", &kvm_vm_fops, kvm, O_RDWR);
      ...
}
```

```
static int create_vcpu_fd(struct kvm_vcpu *vcpu)
{
        char name[8 + 1 + ITOA_MAX_LEN + 1];

        snprintf(name, sizeof(name), "kvm-vcpu:%d", vcpu->vcpu_id);
        return anon_inode_getfd(name, &kvm_vcpu_fops, vcpu, O_RDWR | O_CLOEXEC);
}
```

KVM: main data structure

- KVM 커널 모듈의 관점에서 각각의 가상 머신은 KVM structure로 표시

include/linux/kvm_host.h:

```
struct kvm {
    …
    struct mm_struct *mm;

    struct kvm_vcpu *vcpus[KVM_MAX_VCPUS];
    …
}
```

- kvm structure 는 QEMU-KVM 사용자 공간의 CPUX86State 에 해당하는 kvm_vcpu의 포인터 배열 포함

KVM: main data structure (계속)

- kvm_vcpu는 공통 부분과 레지스터 내용을 포함하는 x86 아키텍처 부분으로 구성됨

```
struct kvm_vcpu {
    …
    struct kvm *kvm;
    int cpu;
    …
    int vpcu_id;
    …
    struct kvm_run *run;
    …
    struct kvm_vcpu_arch arch;
    …
}
```

(참고) Linux: include/linux/kvm_host.h

```
struct kvm_vcpu_arch {
    …
    unsigned long regs[NR_VCPU_REGS];
    unsigned long cr0;
    unsigned long cr0_guest_owned_bits;
    …
    struct kvm_lapic *apic;    → kernel irqchip context
    …
    struct kvm_mmu mmu;

    …
}
```

(참고) Linux: arch/x86/include/asm/kvm_host.h

KVM: main data structure (계속)

- kvm_vcpu에는 연관된 **kvm_run** structure가 있고, 이 것은 QEMU 사용자 공간과 KVM 커널 모듈 간의 통신에 사용됨

- 예를 들어, VMEXIT의 컨텍스트에서 가상 하드웨어 액세스의 emulation을 만족하려면 KVM이 QEMU 사용자 공간 프로세스로 돌아가야 하며, KVM은 QEMU가 정보를 가져올 수 있도록 kvm_run structure에 정보를 저장함

QEMU-KVM: vCPU 실행 과정

- vCPU 쓰레드는 ioctl(.., KVM_RUN, ..)을 실행해 guest 코드를 실행함

```
QEMU-KVM User Space:
kvm_init_vcpu()
    kvm_arch_init_vcpu()
        qemu_init_vcpu()
            qemu_kvm_start_vcpu()
                qemu_kvm_cpu_thread_fn()
                    while(!
                        if(cpu_can_run(cpu)) {
                         r = kvm_cpu_exec(CPU);
                        }


                            kvm_cpu_exec(CPUState *cpu)
                                --> run_ret = kvm_vcpu_ioctl(cpu, KVM_RUN, 0);
```

QEMU-KVM: vCPU 실행 과정 (계속)

- KVM은 kvm_x86_ops 벡터를 사용해 하드웨어용으로 로드된 KVM 모듈(kvm-intel 또는 kvm-amd)에 따른 벡터 중 하나를 가리킴

- run 포인터는 guest vCPU가 실행 중일 때 실행해야 할 기능을 정의하고, handle_exit는 VMEXIT 실행시 수행해야할 작업을 정의

vcpu_vmx structure:

```
static struct kvm_x86_ops vmx_x86_ops = {
        ...
          .vcpu_create = vmx_create_vcpu,
          .run = vmx_vcpu_run,
          .handle_exit = vmx_handle_exit,
        ...
}
```

vcpu_svm structure:

```
static struct kvm_x86_ops svm_x86_ops = {
        ...
          .vcpu_create = svm_create_vcpu,
          .run = svm_vcpu_run,
          .handle_exit = handle_exit,
        ...
}
```

QEMU-KVM: vCPU 실행 과정 (계속)

- vcpu_enter_guest()에서 VMENTRY와 VMEXIT의 개괄적인 흐름 파악 가능

```
static long kvm_vcpu_ioctl(struct file *filp, unsigned int ioctl, unsigned long arg) {
        switch(ioctl) {
            case KVM_RUN:
                ....
                kvm_arch_vcpu_ioctl_run(vcpu, vcpu->run);
                      -> vcpu_load
                      -> vmx_vcpu_load
                      -> vcpu_run(vcpu);
                      -> vcpu_enter_guest
                      -> vmx_vcpu_run
```

```
static int vcpu_run(struct kvm_vcpu *vcpu) {
        ...
        for(;;) {
            if (kvm_vcpu_running(vcpu)) {
                r = vcpu_enter_guest(vcpu);
            } else {
                r = vcpu_block(kvm, vcpu);
            }
```

```
static int vcpu_enter_guest(struct kvm_vcpu *vcpu) {
        ...
        kvm_x86_ops->prepare_guest_switch(vcpu);
        vcpu->mode = IN_GUEST_MODE;
        // enter guest
        ...
        kvm_x86_ops->run(vcpu);
                        [vmx_vcpu_run or svm_vcpu_run]
        ...
        vcpu->mode = OUTSIDE_GUEST_MODE;
         // exit guest
        r = kvm_x86_ops->handle_exit(vcpu);
                        [vmx_handle_exit or handle_exit]
```

- VMENTRY([vmx_vcpu_run 또는 svm_vcpu_run])하여 guest 코드 실행. 이 단계에서 여러 이벤트로 인해 VMEXIT가 발생함. 이 경우 vmx_handle_exit 또는 handle_exit가 종료 원인을 처리