OSv on bhyve
Takuya ASADA / @syuu1228
What is OSv?

- OSv is open source OS which designed to execute a single application on top of a hypervisor
- Better performance, easy to manage
- Developing by Cloudius Systems, Israeli startup
  Core member are come from Qumranet
  CTO: Avi Kivity (‘Father’ of KVM)
A Historical Anomaly

Your App

Application Server

JVM

Operating System

Hypervisor

Hardware

provides protection and abstraction

provides protection and abstraction

provides protection and abstraction
<table>
<thead>
<tr>
<th>Property/Component</th>
<th>VMM</th>
<th>OS</th>
<th>runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware abstraction</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>Isolation</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>Resource virtualization</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>Backward compatibility</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>Security</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>Memory management</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>I/O stack</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>Configuration</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
</tbody>
</table>
The new Cloud Stack - OS^v

Your App
Application Server
JVM
Core
Hypervisor
Hardware

Linked to existing JVMs
App sees no change

Single Process
Kernel space only
OSv application deployment

Deploy your Java applications into OSv by following these steps:

- Upload your application zip file (see example project).
- Activate the uploaded application by starting it.
Use cases

● Rent-an-instance on a public cloud
● Internal instance on a private cloud
● Pre-packaged apps
  ○ MemCache, NoSQL
● OEM virtual appliance
Design of OSv

- Single process + thread support
- **Single memory space**, no switch page table
- **Application runs on kernel mode**, no switch privilege mode
- Binary compatibility with Linux app (with some limitation)
OSv internal structure

- Thin OS core + FreeBSD ZFS, TCP/IP and musl-libc
- Able to load & run Linux binary of OpenJDK

Diagram showing components of OpenJDK and OSv kernel, with labels for OpenJDK, libjvm.so, java.so, sched, libc, ELF loader, ZFS, TCP/IP, VFS, virtio-blk, virtio-net, CRuSH, JRuby, Puma, Sinatra, COM port, MM, RAMFS, ACPI, clock, syscall emu, HTTP, SSH, port, virtio-blk, virtio-net, original code (C++11), porting, Linux binary, Java byte code.
C app on OSv

- All application should be compiled with:
  CFLAGS+=-fPIC
  LDFLAGS=-shared
  → Shared library, but with main()

- You can load Linux shared library, but need to recompile executables

- Linux compatibility is implemented on libc layer

- No syscall!
Available apps?

- OpenJDK
- Cassandra
- Tomcat
- haproxy
- memcached
- rogue
- mruby
- sqlite
- benchmarks (netperf, iperf, specjvm)
Supported Hypervisor

- Linux KVM
- Xen
- VirtualBox (work in progress)
- VMware (work in progress)
Device drivers

- virtio-blk, virtio-net
- Xen PV drivers
- VMware PV drivers
- SATA
- IDE
- COM port
- VGA & keyboard
- Clock (HPET)
Let’s support bhyve!

- Device drivers should work on bhyve (COM port, virtio-net, virtio-blk)
- Main problem is bootloader
- bhyve does not have BIOS, need to implement OSLoader for OSv
bhyveosvload

- Implemented, but still work-in-progress:
  - [https://github.com/syuu1228/bhyveosvload](https://github.com/syuu1228/bhyveosvload)
What OSLoader do?

- It executes boot procedure on HostOS side
- VM launch from 64bit entry point of guest kernel
Traditional boot procedure with BIOS

- BIOS loads boot sector from MBR on a disk
- Boot sector loads and jumps to boot loader (BIOS call used for IO)
- Boot loader initializes page table, GDT and special registers
- Boot loader locates and loads kernel (BIOS call used for IO)
- Boot loader switches to 64bit mode, jumps to kernel entry point
Direct boot

- BIOS loads boot sector from MBR on a disk
- Boot sector loads and jumps to boot loader (BIOS call used for IO)
- Boot loader initializes page table, GDT and special registers
- Boot loader locates and loads kernel (BIOS call used for IO)
- Boot loader switches to 64bit mode, jumps to kernel entry point
How to implement it?

- Read assembly code in boot loader, translate it to C code on HostOS, manually
code example(1)

- Print string on console (BIOS INT10h)
  
  `printf()`

- Read disk (BIOS INT13h)
  
  `fd = open(disk_image)`
  
  `read(fd, buf, len)`

- Memory access
  
  `ctx = vm_open(vm_name)`
  
  `ptr = vm_map_gpa(ctx, offset, len)`
  
  `memcpy(ptr, data, len)`
code example (2)

- Register write (normal registers)
  ```c
  ctx = vm_open(vm_name)
  vm_set_register(ctx, cpuno, VM_REG_GUEST_RFLAGS, val)
  ```

- Register write (Segment registers)
  ```c
  ctx = vm_open(vm_name)
  vm_set_desc(ctx, cpuno, VM_REG_GUEST_CS, base, limit, access)
  vm_set_register(ctx, cpuno, VM_REG_GUEST_CS, selector)
  ```
Let’s begin to translate boot16.S


- It’s on MBR boot sector
  - Load kernel argument from disk
  - Load kernel ELF binary from disk
  - Get memory map from BIOS(e820)
  - Entry to kernel 32bit protected mode code
disk image layout of OSv

- Does not use standard boot loader (Ex: GRUB) to boot faster
- kernel argument, kernel ELF binary are placed on fixed sector
Translate bootloader code(1): cmdline load

```
 cmdline = 0x7e00
 mb_info = 0x1000
 mb_cmdline = (mb_info + 16)

 int1342_boot_struct: # for command line ← DAP
    .byte 0x10 ← size of DAP
    .byte 0 ← unused
    .short 0x3f # 31.5k ← number of sectors to be read
    .short cmdline ← segment:offset pointer to the memory buffer (offset側)
    .short 0 ← (segment側)
    .quad 1 ← absolute number of the start of the sectors to be read

 init:
 xor %ax, %ax
 mov %ax, %ds ← DS = 0

 lea int1342_boot_struct, %si ← DS:SIでDAPを指定
 mov $0x42, %ah
 mov $0x80, %dl
 int $0x13 ← INT 13h AH=42h: Extended Read Sectors From Drive
 movl $cmdline, mb_cmdline ← mb_info->mb_cmdlineに0x7e00を代入
```
INT 13h AH=42h: Extended Read Sectors From Drive

Parameters:

<table>
<thead>
<tr>
<th>Registers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH</td>
<td>42h = function number for extended read</td>
</tr>
<tr>
<td>DL</td>
<td>drive index (e.g. 1st HDD = 80h)</td>
</tr>
<tr>
<td>DS:SI</td>
<td>segment:offset pointer to the DAP, see below</td>
</tr>
</tbody>
</table>

DAP: Disk Address Packet

<table>
<thead>
<tr>
<th>Offset Range</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>1 byte</td>
<td>size of DAP = 16 = 10h</td>
</tr>
<tr>
<td>01h</td>
<td>1 byte</td>
<td>unused, should be zero</td>
</tr>
<tr>
<td>02h..03h</td>
<td>2 bytes</td>
<td>number of sectors to be read, (some Phoenix BIOSes are limited to a maximum of 127 sectors)</td>
</tr>
<tr>
<td>04h..07h</td>
<td>4 bytes</td>
<td>segment:offset pointer to the memory buffer to which sectors will be transferred (note that x86 is little-endian: if declaring the segment and offset separately, the offset must be declared before the segment)</td>
</tr>
<tr>
<td>08h..0Fh</td>
<td>8 bytes</td>
<td>absolute number of the start of the sectors to be read (1st sector of drive has number 0)</td>
</tr>
</tbody>
</table>

Results:

<table>
<thead>
<tr>
<th>CF</th>
<th>Set On Error, Clear If No Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH</td>
<td>Return Code</td>
</tr>
</tbody>
</table>
Translate bootcode(1): cmdline load

cchar *cmdline;
struct multiboot_info_type *mb_info;

cmdline = vm_map_gpa(ctx, 0x7e00, 1 * 512);
pread(disk_fd, cmdline, 0x3f * 512, 1 * 512);

mb_info = vm_map_gpa(ctx, 0x1000, sizeof(*mb_info));
mb_info->cmdline = 0x7e00;
tmp = 0x80000

count32: .short 4096 # in 32k units, 4096=128MB

int1342_struct:
  .byte 0x10
  .byte 0
  .short 0x40 # 32k
  .short 0
  .short tmp / 16

lba:
  .quad 128

read_disk:
  lea int1342_struct, %si
  mov $0x42, %ah
  mov $0x80, %dl
  int $0x13
  jc done_disk
  cli
  lgdtw gdt
  mov $0x11, %ax
  lmsw %ax
  ljmp $8, $1f

1:
  .code32
  mov $0x10, %ax
  mov %eax, %ds
  mov %eax, %es
  ljmpw $0x18, $1f

1:
  .code16
  xor %ax, %ax
  mov %eax, %ds
  mov %eax, %es
  sti
  addl $(0x8000 / 0x200), lba
  decw count32
  ljmpw $0, $1f

1:
  .code16
  xor %ax, %ax
  mov %eax, %ds
  mov %eax, %es
  sti
  addl $(0x8000 / 0x200), lba
  decw count32
  ljmpw $0, $1f

done_disk:
char *target;

target = vm_map_gpa(ctx, 0x2000000, 1 * 512);
pread(disk_fd, target, 0x40 * 4096 * 512, 128 * 512);
Translate bootloader code (3): memory map (e820)

mb_info = 0x1000
mb_mmap_len = (mb_info + 44)
mb_mmap_addr = (mb_info + 48)
e820data = 0x2000

mov $e820data, %edi ← ES:DI Buffer Pointer
mov %edi, mb_mmap_addr ← mb_info->mb_mmap_addrに0x2000を代入
xor %ebx, %ebx ← Continuation

more_e820:
mov $100, %ecx ← Buffer Size
mov $0x534d4150, %edx ← Signature 'SMAP'
mov $0xe820, %ax
add $4, %edi
int $0x15 ← INT 15h, AX=E820h - Query System Address Map
jc done_e820
mov %ecx, -4(%edi)
add %ecx, %edi
test %ebx, %ebx
jnz more_e820
done_e820:
sub $e820data, %edi
mov %edi, mb_mmap_len ← mb_info->mb_mmap_lenにe820dataのサイズを代入
Translate bootcode(3): memory map(e820)

```c
struct e820ent *e820data;

e820data = vm_map_gpa(ctx, 0x1100, sizeof(struct e820ent) * 3);
e820data[0].ent_size = 20;
e820data[0].addr = 0x0;
e820data[0].size = 654336;
e820data[0].type = 1;
e820data[1].ent_size = 20;
e820data[1].addr = 0x100000;
e820data[1].size = mem_size - 0x100000;
e820data[1].type = 1;
e820data[2].ent_size = 20;
e820data[2].addr = 0;
e820data[2].size = 0;
e820data[2].type = 0;

mb_info->mmap_addr = 0x1100;
mb_info->mmap_length = sizeof(struct e820ent) * 3;
```
Translate bootcode(4): entry to protected mode

cmdline = 0x7e00
target = 0x200000
tenry = 24+target
mb_info = 0x1000

ljmp $8, $1f

1:
.
code32
mov $0x10, %ax
mov %eax, %ds
mov %eax, %es
mov %eax, %gs
mov %eax, %fs
mov %eax, %ss
mov $target, %eax ← 0x200000をeaxに設定
mov $mb_info, %ebx ← 0x1000をebxに設定
jmp *entry ← 32bit protected modeのコードを動かすつもりはないので無視
Translate bootcode(4): entry to protected mode

```c
vm_set_register(ctx, 0, VM_REG_GUEST_EAX, 0x200000);
vm_set_register(ctx, 0, VM_REG_GUEST_EBX, 0x1000);
```
Translate boot.S

- 32bit entry point on kernel
  - Initialize GDT and Page Table, switch to 64bit mode
Translate bootloader:
Initialize GDT

gdt_desc:
  .short gdt_end - gdt - 1
  .long gdt

.align 8
.gdt = . - 8
  .quad 0x00af9b00000000000000000000000000 # 64-bit code segment
  .quad 0x00cf9300000000000000000000000000 # 64-bit data segment
  .quad 0x00cf9b00000000000000000000000000 # 32-bit code segment

gdt_end = .

lgdt gdt_desc
Translate bootcode(5):
Initialize GDT

/* gdtrは空いてそうでな適当な領域に置く */

uint64_t *gdtr = vm_map_gpa(ctx, 0x5000,
sizeof(struct uint64_t) * 4);
gdtr[0] = 0x0;
gdtr[1] = 0x00af9b000000ffff;
gdtr[2] = 0x00cf93000000ffff;
gdtr[3] = 0x00cf9b000000ffff;

vm_set_desc(ctx, 0, VM_REG_GUEST_GDTR, gdtr,
sizeof(struct uint64_t) * 4 - 1, 0);
Translate bootcode(6): Initialize Page Table

.data
.align 4096
ident_pt_l4:
    .quad ident_pt_l3 + 0x67
    .rept 511
    .quad 0
    .endr
ident_pt_l3:
    .quad ident_pt_l2 + 0x67
    .rept 511
    .quad 0
    .endr
ident_pt_l2:
    index = 0
    .rept 512
    .quad (index << 21) + 0x1e7
    index = index + 1
    .endr

lea ident_pt_l4, %eax
mov %eax, %cr3
Translate bootcode(6):
Initialize Page Table

```c
uint64_t *PT4;
uint64_t *PT3;
uint64_t *PT2;

/* PT4-2は空いてそうな適当な領域に置く */
PT4 = vm_map_gpa(ctx, 0x4000, sizeof(uint64_t) * 512);
PT3 = vm_map_gpa(ctx, 0x3000, sizeof(uint64_t) * 512);
PT2 = vm_map_gpa(ctx, 0x2000, sizeof(uint64_t) * 512);

for (i = 0; i < 512; i++) {
    PT4[i] = (uint64_t) ADDR_PT3;
    PT4[i] |= PG_V | PG_RW | PG_U;
    PT3[i] = (uint64_t) ADDR_PT2;
    PT3[i] |= PG_V | PG_RW | PG_U;
    PT2[i] = i * (2 * 1024 * 1024);
    PT2[i] |= PG_V | PG_RW | PG_PS | PG_U;
}

vm_set_register(ctx, 0, VM_REG_GUEST_CR3, 0x4000);
```
Translate bootcode(7):
Initialize special registers for 64bit mode

```c
#define BOOT_CR0 ( X86_CR0_PE \
| X86_CR0_WP \
| X86_CR0_PG )

#define BOOT_CR4 ( X86_CR4_DE \n| X86_CR4_PSE \n| X86_CR4_PAE \n| X86_CR4_PGE \n| X86_CR4_PCE \n| X86_CR4_OSFXSR \n| X86_CR4_OSEXMMEXCPT )

and $~7, %esp
mov $BOOT_CR4, %eax
mov %eax, %cr4 ← PAE有効など
mov $0xc0000080, %ecx
mov $0x00000900, %eax
xor %edx, %edx
wrmsr ← EFERのLMEフラグを立てている
mov $BOOT_CR0, %eax
mov %eax, %cr0 ← PE,PG有効など
ljmpl $8, $start64

.code64
.global start64
start64:
```
Translate bootcode(7):
Initialize special registers for 64bit mode

vm_set_register(ctx, 0, VM_REG_GUEST_RSP, ADDR_STACK);
vm_set_register(ctx, 0, VM_REG_GUEST_EFER, 0x00000d00);
vm_set_register(ctx, 0, VM_REG_GUEST_CR4, 0x000007b8);
vm_set_register(ctx, 0, VM_REG_GUEST_CR0, 0x80010001);
Translate bootloader code(8):
64-bit entry point

```c
#define BOOT_CR0 ( X86_CR0_PE \ 
  | X86_CR0_WP \ 
  | X86_CR0_PG )

#define BOOT_CR4 ( X86_CR4_DE \ 
  | X86_CR4_PSE \ 
  | X86_CR4_PAE \ 
  | X86_CR4_PGE \ 
  | X86_CR4_PCE \ 
  | X86_CR4_OSFXSR \ 
  | X86_CR4_OSXMMEXCPT )

and $~7, %esp
mov $BOOT_CR4, %eax
mov %eax, %cr4
mov $0xc0000080, %ecx
mov $0x00000900, %eax
xor %edx, %edx
wrmsr
mov $BOOT_CR0, %eax
mov %eax, %cr0
ljmpl $8, $start64

.code64
.global start64 ← Want to set RIP here
start64:
```
Ouch…

- This function is NOT have fixed address
- Address may changed on recompiling
Let’s parse ELF header

Implement symbol name to address function using elf(3) and gelf(3)

```c
int elfparse_open_memory(char *image, size_t size, struct elfparse *ep);

int elfparse_close(struct elfparse *ep);

uintmax_t elfparse_resolve_symbol(struct elfparse *ep, char *name);
```
Translate bootcode(8): 64bit entry point

```c
struct elfparse ep;
uint64_t start64;

if (elfparse_open_memory(target, 0x40 * 4096 * 512, &ep));
start64 = elfparse_resolve_symbol(&ep, "start64");
vm_set_register(ctx, 0, VM_REG_GUEST_RIP, start64);
```
Completed implementation!

```
# /usr/local/sbin/bhyveosvload -m 1024 -d ../loader.img osv0
sizeof e820data=48
cmdline=java.so -jar /usr/mgmt/web-1.0.0.jar app prod
start64:0x208f13
ident_pt_l4:0x8d5000
gdt_desc:0x8d8000

# /usr/sbin/bhyve -c 1 -m 1024 -AI -H -P -g 0 -s 0:0,hostbridge -s 1:0,virtio-net,tap0 -s 2:0,virtio-blk,../loader.img -S 31,uart,stdio osv0
ACPI: RSDP 0xf0400 00024 (v02 BHYVE )
ACPI: XSDT 0xf0480 00034 (v01 BHYVE BVXSDT 00000001 INTL 20130823)
ACPI: APIC 0xf0500 0004A (v01 BHYVE BVMADT 00000001 INTL 20130823)
ACPI: FACP 0xf0600 0010C (v05 BHYVE BVFACP 00000001 INTL 20130823)
ACPI: DSDT 0xf0800 000F2 (v02 BHYVE BVDSDT 00000001 INTL 20130823)
ACPI: FACS 0xf0780 00040
Assertion failed: st == AE_OK (../../drivers/hpet.cc: hpet_init: 171)
Aborted
```
Development Status

- 0SLoader implementation is completed
- Still have some problem to boot OSv, because of lack of device driver