Implements BIOS emulation support for BHyVe: A BSD Hypervisor

Abstract

Current BHyVe only supports FreeBSD/amd64 as a GuestOS.
One of the reason why BHyVe cannot support other OSes is lack of BIOS support.
My project is implementing BIOS emulator on BHyVe, to remove these limitations.

1. Background

1.1 History of virtualization on x86 architecture

There's a famous requirements called "Popek & Goldberg Virtualization requirements"[1], which defines a set of conditions sufficient for an architecture to support virtualization efficiently.
Efficient virtualization means virtualize machine without using full CPU emulation, run guest code natively.
Explain the requirements simply, to an architecture virtualizable, all **sensitive instructions** should be privileged instruction.
Sensitive instructions definition is the instruction which can interfere the global status of system.
Which means, all sensitive instructions executed under user mode should be trapped by privileged mode program.
Without this condition, Guest OS affects Host OS system status and causes system crash.
x86 architecture was the architecture which didin't meet the requirement, because It had non-privileged sensitive instructions.
To virtualize this architecture efficiently, hypervisors needed to avoid execute these instructions, and replace instruction with suitable operations.
There were some approach to implement it:
On VMware approach, the hypervisor replaces problematic sensitive instructions on-the-fly, while running guest machine. This approach called **Binary Translation**[2].
It could run most of unmodified OSes, but it had some performance overhead.
On Xen approach, the hypervisor requires to run pre-modified GuestOS which replaced problematic sensitive instructions to dedicated operations called **Hypercall**. This approach called **Para-virtualization**[3].
It has less performance overhead than Binary Translation on some conditions, but requires pre-modified GuestOS.
Due to increasing popularity of virtualization on x86 machines, Intel decided to enhance x86 architecture to **virtualizable**.
The feature called **Intel VT-x**, or **Hardware-Assisted Virtualization** which is vendor neutral term.
AMD also developed hardware-assisted virtualization feature on their own CPU, called **AMD-V**.

1.2 Detail of Intel VT-x

VT-x provides new protection model which isolated with **Ring protection**, for virtualization.
It added two CPU modes, **hypervisor mode** and **guest machine mode**.
Hypervisor mode called **VMX Root Mode**, and guest machine mode called **VMX non Root Mode**(Figure 1).

Gerald J. Popek and Robert P. Goldberg. 1974. Formal requirements for virtualizable third generation architectures. Commun. ACM 17, 7 (July 1974), 412-421. DOI=10.1145/361011.361073 http://doi.acm.org/10.1145/361011.361073

[2] Brian Walters. 1999. VMware Virtual Platform. Linux J. 1999, 63es, Article 6 (July 1999).

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03). ACM, New York, NY, USA, 164-177. DOI=10.1145/945445.945462 http://doi.acm.org/10.1145/945445.945462

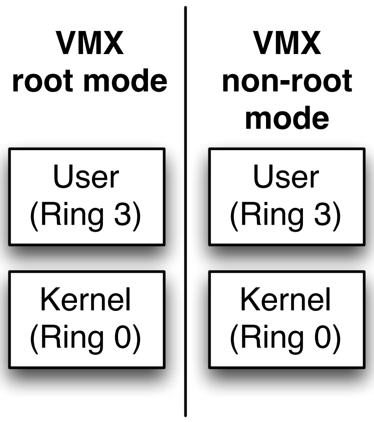

Figure 1. VMX root Mode and VMX non-root Mode

On VT-x, hypervisor can run guest OS on VMX non Root Mode without any modification, including **sensitive instructions**, without affecting Host OS system status.
When sensitive instructions are being executed under VMX non Root Mode, CPU stops execution of VMX non Root Mode, exit to VMX Root Mode.
Then it trapped by hypervisor, hypervisor emulates the instruction which guest tried to execute.
Mode change from VMX Root Mode to VMX non-root Mode called **VMEntry**, from VMX non-root Mode to VMX Root Mode called **VMExit**(Figure 2).

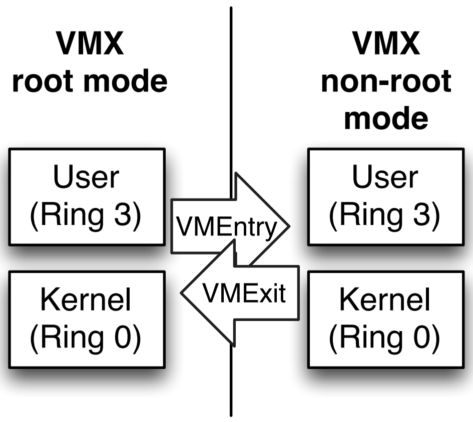

Figure 2. VMEntry and VMExit

Some more events other than sensitive instructions which need to intercept by hypervisor also causes VMExit.
For example, IN/OUT instruction causes VMExit, and hypervisor emulates virtual device access.
VT-x defines number of events which can cause VMExit, and hypervisor needs to configure enable/disable on each VMExit events.
Reasons of VMExit is called **VMExit reason**, it classified by genres of events.

Here are VMExit reason list:

- Exception or NMI
- External interrupt
- Triple fault
- INIT signal received
- SIPI received
- SM received
- Internal interrupt
- Task switch
- CPUID instruction
- Intel SMX instructions
- Cache operation instructions(INVD, WBINVD)
- TLB operation instructions(HNVLPG, INVPCID)
- IO operation instructions(INB, OUTB, etc)
- Performance monitoring conter operation instruction(RDTSC)
- SMM related instruction(RSM)
- VT-x instructions(Can use for implement nested virtualization)
- Accesses to control registers
- Accesses to debug registers
- Accesses to MSR
- MONITOR/MWAIT instructions
- PAUSE instruction
- Accesses to Local APIC
- Accesses to GDTR, IDTR, LDTR, TR
- VMX preemption timer
- RDRAND instruction

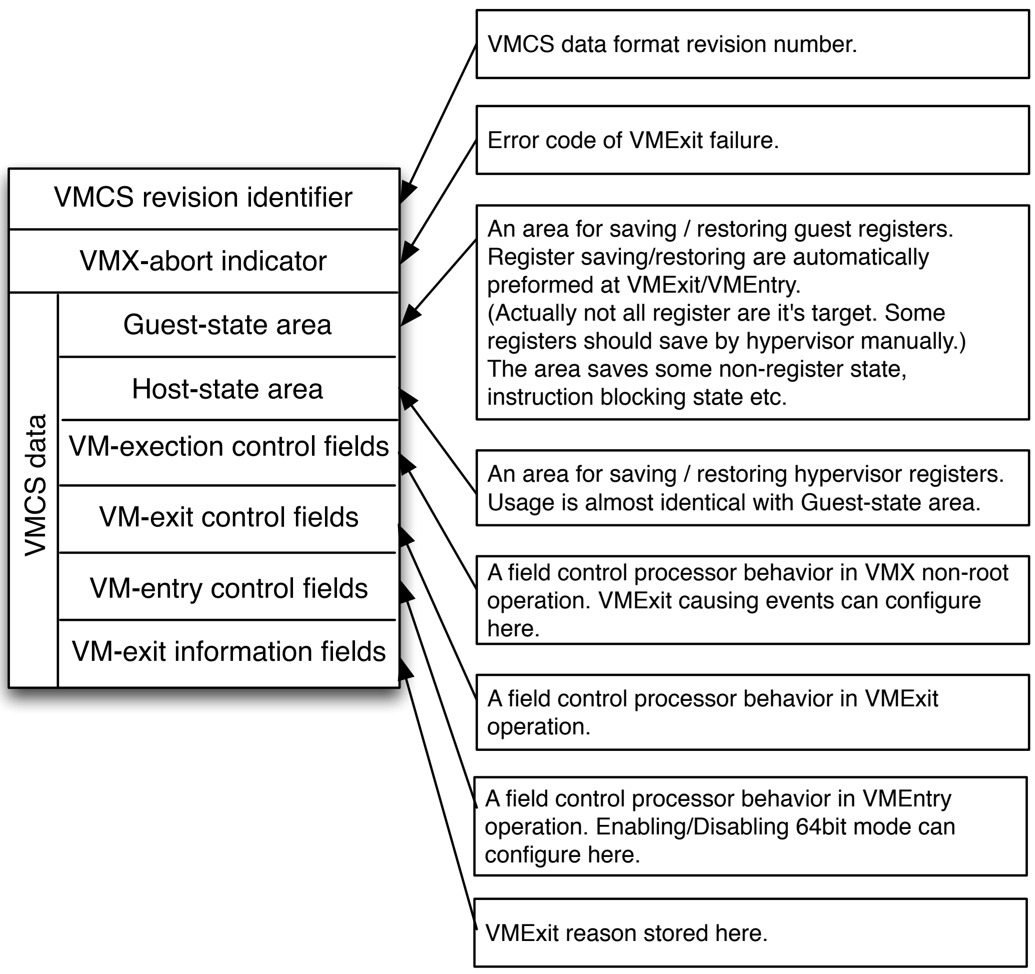
VMCS revision identifier
VMX-abort indicator
VMCS data
Guest-state area
Host-state area
VM-exection control fields
VM-exit control fields
VM-entry control fields
VM-exit information fields
VMCS data format revision number.
Error code of VMExit failure.
An area for saving / restoring guest registers. Register saving/restoring are automatically preformed at VMExit/VMEntry. (Actually not all register are it's target. Some registers should save by hypervisor manually.) The area saves some non-register state, instruction blocking state etc.
An area for saving / restoring hypervisor registers. Usage is almost identical with Guest-state area.
A field control processor behavior in VMX non-root operation. VMExit causing events can configure here.
A field control processor behavior in VMExit operation.
A field control processor behavior in VMEntry operation. Enabling/Disabling 64bit mode can configure here.
VMExit reason stored here.

Figure 3. Structure of VMCS

All configuration data related to VT-x stored to **VMCS(Virtual Machine Control Structure)**, which is on memory data structure for each guest machine[4].
Figure 3 shows VMCS structure.

1.3 VT-x enabled hypervisor lifecycle
Hypervisors for VT-x works as following lifecycle (Figure 4).

1. VT-x enabling
   It requires to enable at first to use VT-x features.
   To enable it, you need set VMXE bit on CR4 register, and invoke VMXON instruction.
2. VMCS initialization
   VMCS is 4KB alined 4KB page.
   You need to notify the page address to CPU by invoking VMPTRLD instruction, then write initial configuration values by VMWRITE instruction.
   You need to write initial register values here, and it done by /usr/sbin/bhyveload.
3. VMEntry to VMX non root mode
   Entry to VMX non root mode by invoking VMLAUNCH or VMRESUME instruction.
   On first launch you need to use VMLAUNCH, after that you need to use VMRESUME.
   Before the entry operation, you need to save Host OS registers and restore Guest OS registers.
   VT-x only offers minimum automatic save/restore features, rest of the registers need to take care manually.
4. Run guest machine
   CPU runs VMX non root mode, guest machine works natively.

[4] If guest system has two or more virtual CPUs, VMCS needs for each vCPUs.

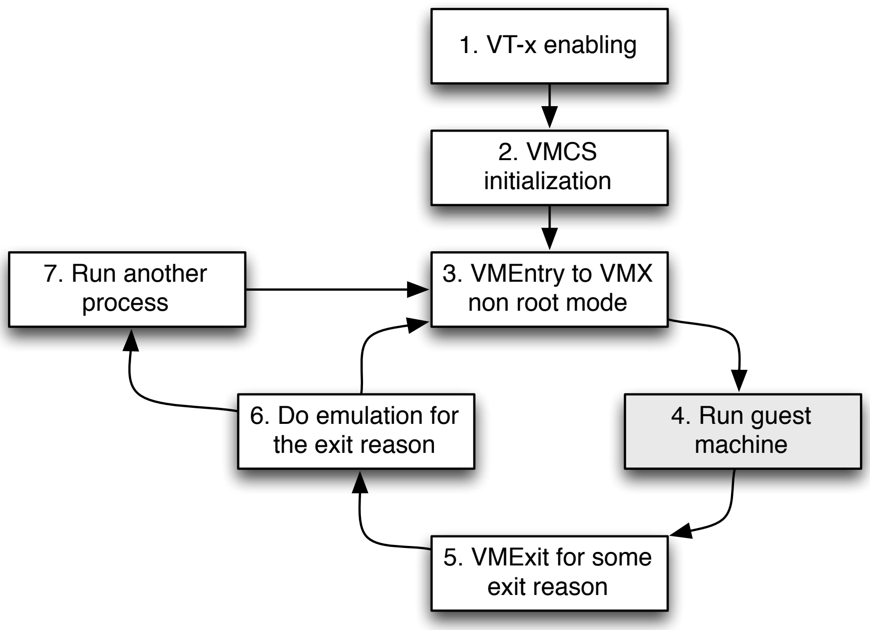
1. VT-x enabling
2. VMCS initialization
3. VMEntry to VMX non root mode
4. Run guest machine
5. VMExit for some exit reason
6. Do emulation for the exit reason
7. Run another process

Figure 4. VT-x enabled hypervisor lifecycle

5. VMExit for some reason
   When some events which causes VMExit, CPU returns to VTX root mode.
   You need to save/restore register at first, then check the VMExit reason.
6. Do emulation for the exit reason
   If VMExit reason was the event which requires some emulation on hypervisor, perform emulation. (Ex: Guest OS wrote data on HDD
   Depending Host OS scheduling, it may resume VM by start again from 3, or task switch to another process.

1.4 Memory Virtualization

Mordan multi-tasking OSes use paging to provide individual memory space for each processes.
To run guest OS program natively, address translation on paging become problematic function.
For example (Figure 5):
You allocate physical page 1- 4 to Guest A, and 5-8 to GuestB.
Both guests map page 1 of Process A to page 1 of guest physical memory.
Then it should point to:

- Page 1 of Process A on Guest A ->
  Page 1 of Guest physical memory ->
  Page 1 of Host physical
- Page 1 of Process B on Guest B ->
  Page 1 of Guest physical memory ->
  Page 5 of Host physical

But, if you run guest OS natively, CPU will translate Page 1 of Process B on Guest B to **Page 1 of Host physical memory**.
Because CPU doesn't know the paging for guests are nested.

There is software technique to solve the problem called **shadow paging** (Figure 6).
Hypervisor creates clone of guest page table, set host physical address on it, traps guest writing CR3 register and set cloned page table to CR3.
Then CPU able to know correct mapping of guest memory.
This technique was used on both Binary translation based VMware, and also early implementation of hypervisors for VT-x.
But it has big overhead, Intel decided to add nested paging support on VT-x from Nehalem micro-architecture.

EPT is the name of nested paging feature (Figure 7),
It simply adds Guest physical address to Host physical address translation table.
Now hypervisor doesn't need to take care guest paging, it become much simpler and faster.

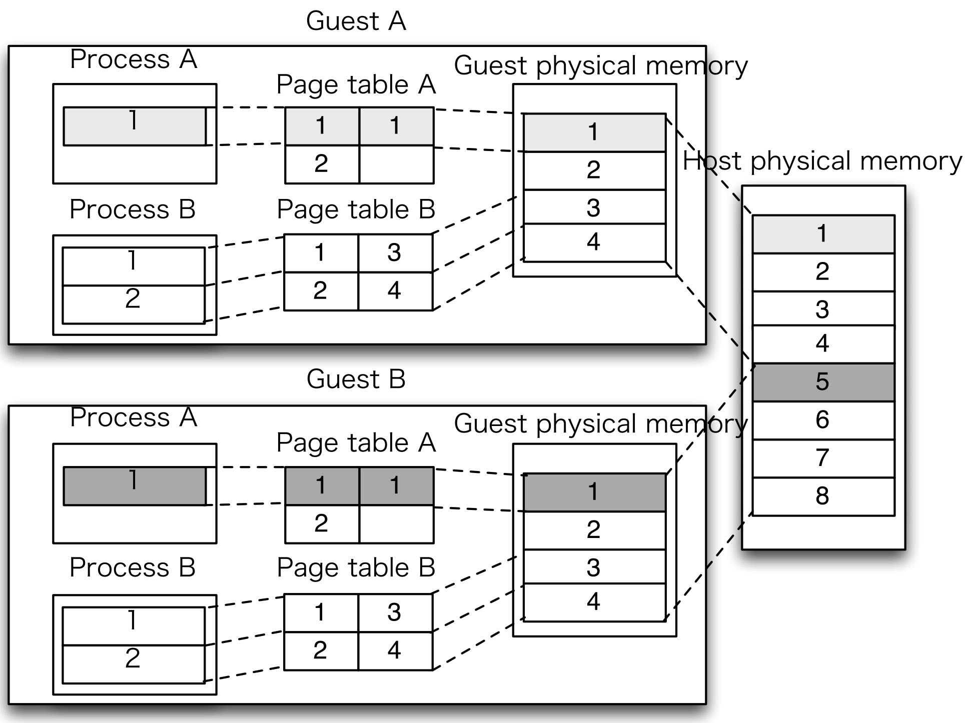

Figure 5. Problem of memory virtualization

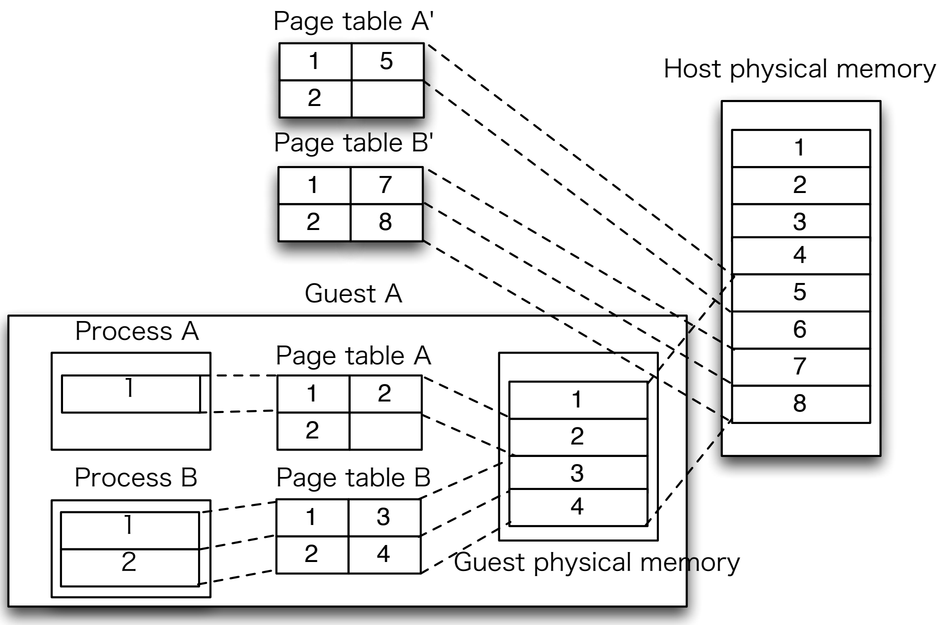

Figure 6. Shadow paging

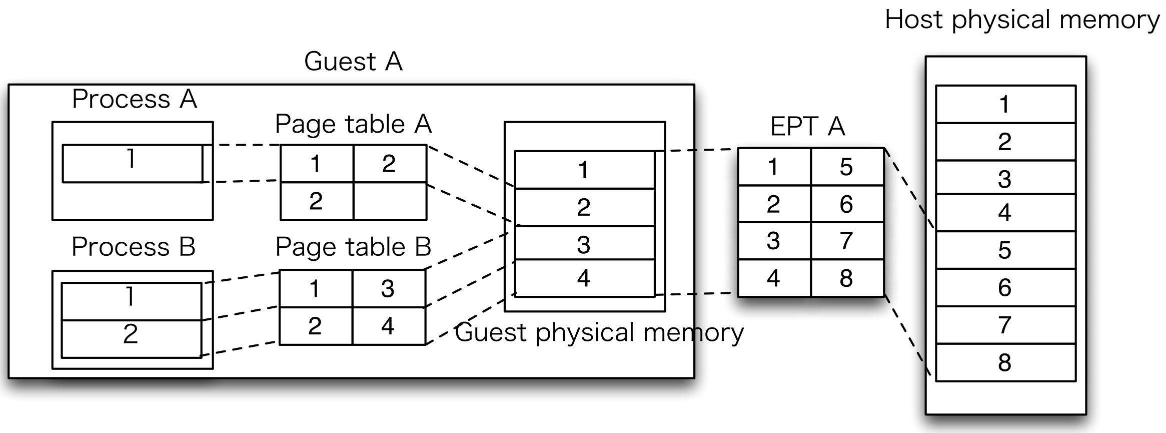

Figure 7. EPT

Actually, not all VT-x supported CPUs supports EPT, on these CPUs hypervisors still need to do shadow paging.

2. BHyVe: BSD Hypervisor

2.1 What is BHyVe?

BHyVe is new project to implement a hypervisor witch will integrate in FreeBSD.
The concept is similar to Linux KVM, it provides “hypervisor driver” to unmodified BSD kernel running on bare-metal machine.
With the driver, the kernel become a hypervisor, able to run GuestOS just like normal process on the kernel.
Both hypervisors are designed for hardware assisted virtualization, unlike Xen’s para-virtualization and VMware’s binary translation.
The kernel module only provides a feature to switch CPU modes between Host mode and Guest mode, almost all device emulation is performed in userland process.

2.2 Difference of approach between Linux KVM and BHyVe

Linux KVM uses modified QEMU[5] as the userland part[6].
It’s good way to support large coverage of Guest OSes, because QEMU is highly developed emulator, many people already confirmed to run variety of OSes on it.
KVM could support almost same features what QEMU has, and it just worked fine.
BHyVe’s approach is different.
BHyVe implements minimum set of device support which required to run FreeBSD guest, from scratch.
In the result, we could have completely GPL-free, BSD licensed, well coded hypervisor, but it only supports FreeBSD/amd64 as a Guest OS at this point.
One of the reason why BHyVe cannot support other OSes is lack of BIOS support.
BHyVe loads and executes FreeBSD kernel directly using custom OS loader runs on Host OS, instead of boot up from disk image.
With this method, we need to implement OS loader for each OSes, and currently we don’t have any loader other than FreeBSD.
Also, it doesn’t support some OSes which calls BIOS function while running.
So I started the project to implementing BIOS emulator on BHyVe, to remove these limitations.

2.3 Hardware requirements

BHyVe requires an Intel CPU which supports Intel VT-x and EPT.
It means you will need Nehalem core or later Intel CPUs, because EPT is only supported on these processors.
Currently, AMD-V is not supported.
Installing on physical machine is best choice, but it also works on recent version of VMware, using **Nested virtualization feature**[7].

2.3 Supported features

BHyVe only supports FreeBSD/amd64 8-10 for guest OS.

---

[5] Original QEMU has full emulation of x86 CPU, but on KVM we want to use VT-x hardware assisted virtualization instead of CPU emulation.
So they replace CPU emulation code to KVM driver call.

[6] Strictly speaking, KVM has another userland implementation called Linux Native KVM Tools, which is built from scratch - same as BHyVe’s userland part.
And it has similar limitation with BHyVe.

[7] The technology which enables **Hypervisor on Hypervisor**. Note that it still requires Nehalem core or later Intel CPUs even on VMware.

It emulates following devices:

- HDD controller: virtio-blk
- NIC controller: virtio-net
- Serial console: 16550 compatible PCI UART
- PCI/PCIe devices passthrough (VT-d)

Boot-up from virtio-blk with PCI UART console is not general hardware configuration on PC architecture, we need to change guest kernel settings on /boot/loader.conf(on guest disk image).
And some older FreeBSD also need to add a virtio drivers[8].
PCI device passthrough is also supported, able to use physical PCI/PCIe devices directly.
Recently ACPI support and IO-APIC support are added, which improves compatibility with existing OSes.

2.4 BHyVe internal
BHyVe built with two parts: kernel module and userland process.
The kernel module is called vmm.ko, it performs actions which requires privileged mode (ex: executes VT-x instructions.
Userland process is named /usr/sbin/bhyve, provides user interface and emulates virtual hardwares.
BHyVe also has OS Loader called /usr/sbin/ bhyveload, loads and initializes guest kernel without BIOS.
/usr/sbin/bhyveload source code is based on FreeBSD bootloader, so it outputs bootloader screen, but VM instance is not yet executing at that stage.
It runs on Host OS, create VM instance and loads kernel onto guest memory area, initializes guest machine registers to prepare direct kernel boot.
To destroy VM instance, VM control utility / usr/sbin/bhyvectl is available.
These userland programs are accesses vmm.ko via VMM control library called libvmmapi.
Figure 8 illustrates overall view of BHyVe.

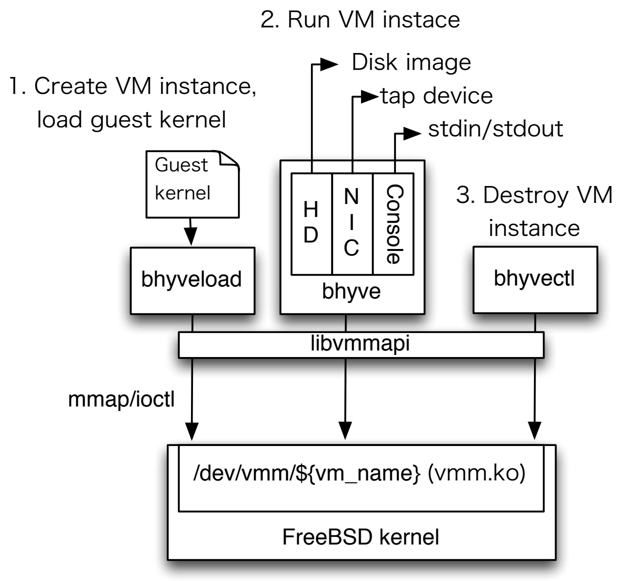

Figure 8. BHyVe overall view

3. Implement BIOS Emulation

3.1 BIOS on real hardware
BIOS interrupt calls are implemented as software interrupt handler on real mode(Figure 9).
CPU executes initialization code on BIOS ROM at the beginning of startup machine, it initializes real mode interrupt vector to handle number of software interrupts reserved for BIOS interrupt calls(Figure 10).
BIOS interrupt calls aren't only for legacy OSes like MS-DOS, almost all boot loaders for mordan OSes are using BIOS interrupt call to access disks, display and keyboard.

3.2 BIOS on Linux KVM
On Linux KVM, QEMU loads **Real BIOS(called SeaBIOS)** on guest memory area at the beginning of QEMU startup.
KVM version of SeaBIOS's BIOS call handler accesses hardware by IO instruction or memory mapped IO, and the behavior is basically same as BIOS for real hardware.
The difference is how the hardware access handled.
On KVM, the hardware access will trapped by KVM hypervisor driver, and QEMU emulates

[8] virtio is **para-virtual driver** which designed for Linux KVM. para-virtual driver needs special driver for guest, but usually much faster than full emulation driver.

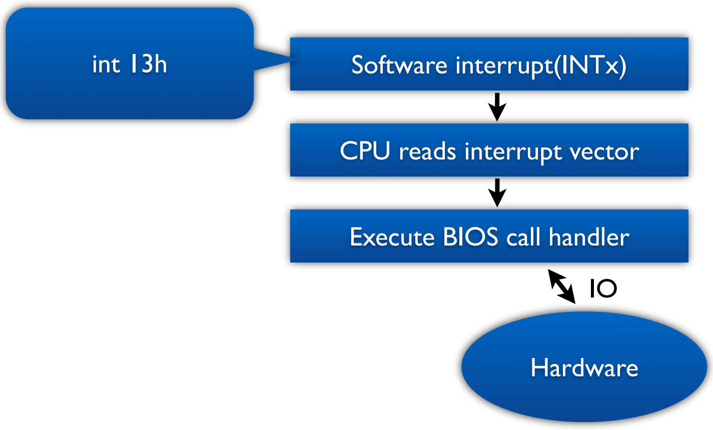

Figure 9. BIOS interrupt call mechanism on real hardware

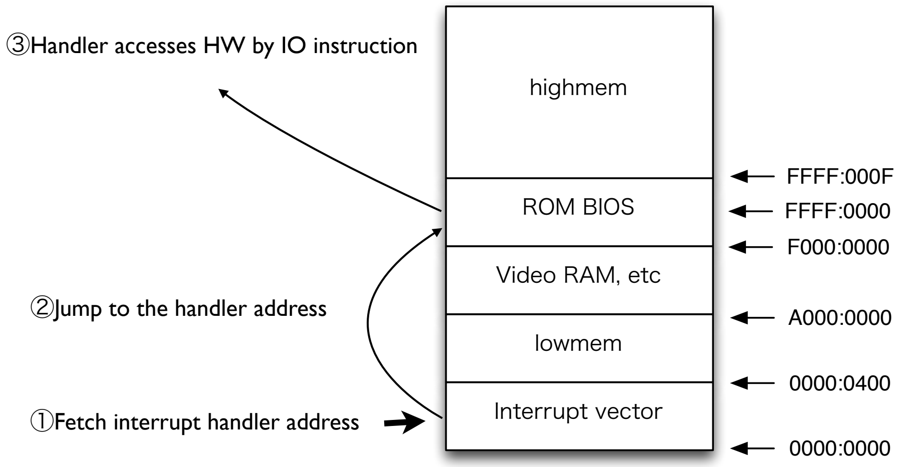

Figure 10. Memory map on real hardware

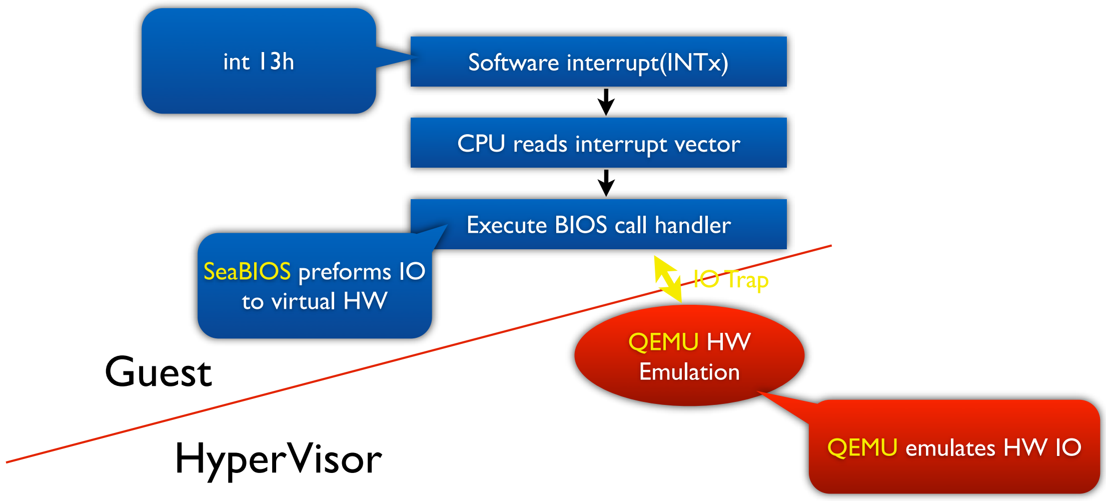

Figure 11. BIOS interrupt call mechanism on KVM

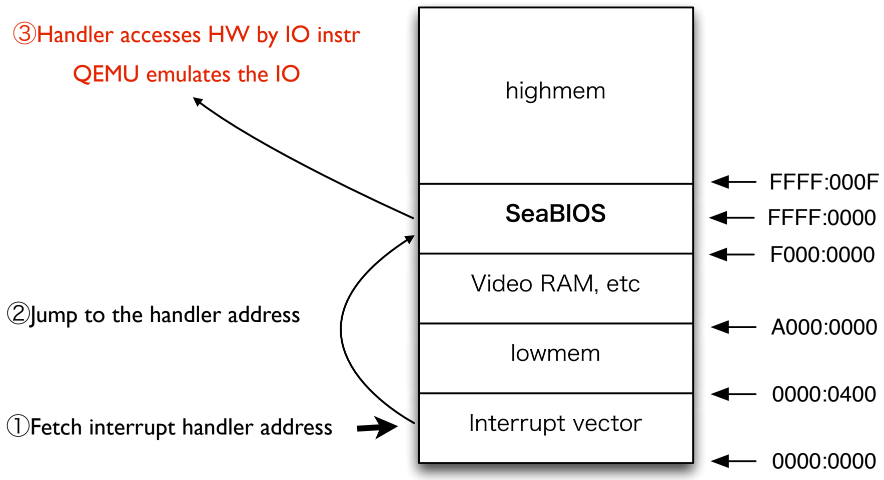

Figure 12. Memory map on KVM

hardware device, then KVM hypervisor driver resume a guest environment(Figure 11).
In this implementation, KVM and QEMU doesn't trap BIOS interrupt calls, it just loads real BIOS on guest memory space(Figure 12) and emulates hardware device.

3.3 Emulating BIOS on BHyVe

3.3.1 doscmd

Port SeaBIOS on BHyVe and implement hardware emulation was an option, and it was probably best way to improve compatibility of legacy code, but SeaBIOS is GPL'd software, it's not comfortable to bring in FreeBSD code tree.
And there's no implementation non-GPL opensourced BIOS.
Instead, there's BSD licensed DOS Emulator called **doscmd**.
It's the software to run old DOS application on FreeBSD using virtual 8086 mode, similar to DOSBox(but DOSBox is GPL'd software).

The emulator mechanism is described as follows:

1. Map pages to lowmem area (begin from 0x0), load the DOS application on the area.
2. Enter virtual 8086 mode, start executing the DOS application.
3. DOS application invokes BIOS interrupt call or DOS API call by INTx instruction.
4. DOS Emulator traps software interrupt, emulate BIOS interrupt call or DOS API call.
5. Resume DOS application.

It traps BIOS interrupt calls and DOS API calls and emulate them on FreeBSD protected mode program.
I decided to port the BIOS interrupt call emulation code to BHyVe and trap BIOS interrupt call on BHyVe, instead of porting real BIOS.

3.3.2 Run real mode program on VT-x

On older implementation of VT-x enabled CPU doesn't allow to VMEnter the guest which doesn't enable paging.
Which means real mode program cannot run on VT-x, and hypervisors needed to virtualize real mode without VT-x.
Linux KVM used full CPU emulation using QEMU to virtualize real mode.
Some other hypervisors are used virtual 8086 mode.
This issue was resolved by extending VT-x features.
Intel added **unrestricted guest mode** on Westmere micro-architecture and later Intel CPUs, it uses EPT to translate guest physical address access to host physical address.
With this mode, VMEnter without enable paging is allowed.
I decided to use this mode for BHyVe BIOS emulation.

3.3.3 Trapping BIOS interrupt call

VT-x has functionality to trap various event on guest mode, it can be done by changing VT-x configuration structure called VMCS.
And BHyVe kernel module can notify these events by IOCTL return.
So all I need to do to trapping BIOS call is changing configuration on VMCS, and notify event by IOCTL return when it trapped.
But the problem is which VMExit event is optimal for the purpose.
It looks like trapping software interrupt is the easiest way, but we may have problem after Guest OS switched protected mode.
Real mode and protected mode has different interrupt vector.
It's possible to re-use BIOS interrupt call vector number for different purpose on protected mode.
Maybe we can detect mode change between real mode/protected mode, and enable/disable software interrupt trapping, but it's bit complicated.

Instead of implement complicated mode change detection, I decided to implement software interrupt handler which cause VMExit.

The handler doesn't contain programs for handling the BIOS interrupt call, just perform VMExit by **VMCALL instruction**.
VMCALL causes unconditional VMExit.
It's for call hypervisor from guest OS, such function is called **Hypercall**.

Following is simplest handler implementation:

```
    VMCALL
    IRET
```

Even program is same, you should have the handler program for each vector.
Because guest EIP can be use for determine handled vector number.

If you place BIOS interrupt call handler start at 0x400, and program length is 4byte for each (VMCALL is 3byte + IRET is 1byte), you can determine vector number from hypervisor with following program:

```
vector = (guest_eip - 0x400) / 0x4;
```

BHyVe need to initialize interrupt vector and set pointer of the handler described above.
In this way, it doesn't take care about mode changes anymore.

Figure 13 shows BIOS interrupt call mechanism on my implementation.
On the implementation, it traps BIOS interrupt call itself, emulates by hypervisor.

4. Implementation

Most of work are rewriting doscmd to fit BHyVe interface, from FreeBSD virtual 8086 API.

- Code was 64bit unsafe

doscmd was designed only for 32bit x86, and BHyVe is only for amd64.
So I need to re-write some codes to 64bit safe.

ex:

```
    u_long
    ↓
    uint32_t
```

- Guest memory area started from 0x0

To use virtual 8086, doscmd places guest memory area from 0x0.
But BHyVe's guest memory area is mapped to non-zero address, we need to move all address to BHyVe's guest memory area.

ex:

```
 *(char *)(0x400) = 0;
         ↓
 *(char *)(0x400 + guest_mem) = 0;
```

- Interface with /usr/sbin/bhyve

I don't wanted to mix doscmd's complicated source code with /usr/sbin/bhyve's code, so I modified doscmd's Makefile to build it as a library.
And named it **libbiosemul**.

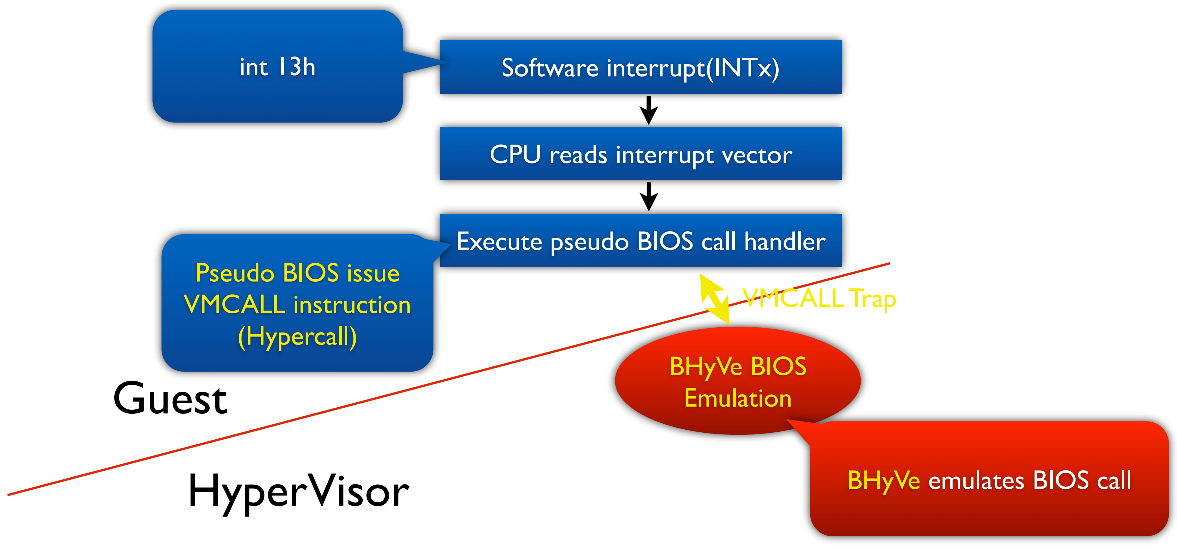
int 13h
Software interrupt(INTx)
CPU reads interrupt vector
Execute pseudo BIOS call handler
Pseudo BIOS issue VMCALL instruction (Hypercall)
VMCALL Trap
BHyVe BIOS Emulation
BHyVe emulates BIOS call
Guest
HyperVisor

Figure 13. BIOS interrupt call mechanism on BHyVe

It exposed only few functions:

```
void biosemul_init(struct vmctx
*ctx, int vcpu, char *lomem, int
trace_mode);

int biosemul_call(struct vmctx
*ctx, int vcpu);
```

biosemul_init is called at initialization.
biosemul_call is main function, which called at every BIOS call.

- Guest register storage

doscmd stored guest register values on their structure, but BHyVe need to call ioctl to get / set register value.
It's hard to re-write all code to call ioctl, so I didn't changed doscmd code.
I just copy all register values to doscmd struct at beginning of BIOS call emulation, and copyback it the end of the emulation.

- Instruction level tracing

I implemented instruction level tracer to debug BIOS emulator.
It's also uses psuedo BIOS interrupt call handler to implement.

5. Development status
It still early stage of development, none of OSes boots up with the BIOS emulator.
I'm focusing to boot-up FreeBSD/amd64, now mbr and boot1 are working correctly.