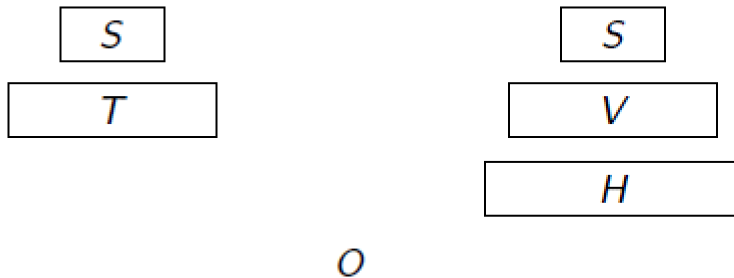
Introduction

Friday, August 25, 2017 15:57



- S: software
- T: target or **guest** (the emulated machine) (hardware or software, like Windows)
- V: virtual machine
- H: **host** (the machine that emulates)
- O: Observer
  - The same as S, since O's interaction with T is often limited to S

Why

- We don't want to change S
  - S can be in machine language
  - When S is changed, it's called *paravirtualization*
- T is not available
  - Historic emulation
  - Doesn't exist yet (new processors)
  - It never existed (JVM)
  - Something else is already used (like an OS, e.g. Wine)

<table>
<tr>
<td>• V is less expensive than T
• V is more flexible than T
• V offers a good protection model for S
• Replacing hardware with software, creates new functionalities
○ Live migration
○ State checkpoints
○ Apps on different OSs on the same desktop</td>
<td>Also useful when T = H, e.g. when the available hardware already is the target machine.

V adds a "new layer of indirection" between S and T.</td>
</tr>
</table>

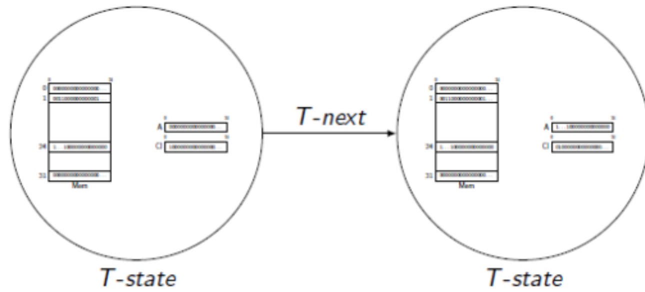
Virtualization techniques

- Emulation
- Binary translation (QEMU, JVM)
- Hardware-assisted (Virtualbox)
- Paravirtualization

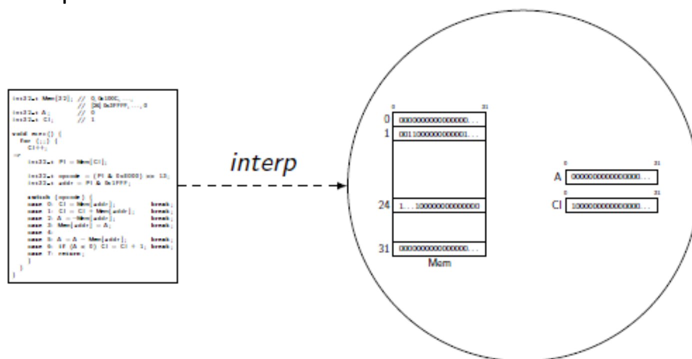
Abstraction

- A description $T-state$ is a snapshot containing all <u>relevant</u> information describing a system's state
  - What's the step to consider for capturing different $T-states$?

        - Program Counter increment
        - Full single instruction execution
- An abstraction is a sequence of these descriptions
- Deterministic state machines $< T - state, T - next >$
    - $T - next$ is a function, $T - next$: $T - state \rightarrow T - state$



    - Usually there are more $V - states$ than $T - states$.
      This means that a $V - next$ transition generally involves **several** transition in the emulation program: we olny consider a new $V - state$ when the program state is again interpretable as a $T - state$.
    - The observer should only be able to see $T - states$, and not $V - states$ (that could be meaningless)
- Interpretation: $V - state \rightarrow T - state$



    - Each $T - state$ might be equivalente to several $V - state$

Emulation

Monday, August 28, 2017 16:55

Emulation program

```
// C doesn't support exceptions
#include <setjmp.h>
```

```
// V-state
memory mem;
cpu_state cpu; // all CPU registers
```

```
// Interrupt variables
bool interrupt; // flag for the interrupt request
uint8_t int_vector; // interrupt type
```

```
// Interrupt variables
jmp_buf exc_jbuf;
exception exc_type; /* this is global because the stack
                       is unwinded during the jump, so
                       it cannot be stored in it */
```

```
// I/O interfaces
iomap io; // data structure mapping I/O addresses
mem_map mm; // data structure mapping memory mapped (mm) interfaces
```

```
void cpu_loop() {
    raw_instr ri;
    decoded_instr di;

    for(;;) {
        try { // exceptions (not supported in C: see next column)
            ri = fetch(cpu->ip);
            di = decode(ri);
            exec(di);

            if(interrupt) {
                /* obtain the new instruction pointer from
                   the interrupt descriptor table */
                cpu->ip = read_idt(int_vector);
            }
        } catch(exception e) {
            // …
            cpu->ip = read_idt(e->type); // as an interrupt
        }
    }
}
```

```
void cpu_loop() {
    raw_instr ri;
    decoded_instr di;

    for(;;) {
        /* this function stores in exc_jbuf all
           the information needed to jump at the
           current program point and returns 0, so
           this if-branch is always skipped */
        if(setjmp(&exc_jbuf)) {
            // …
            cpu->ip = read_idt(exc_type);
        }

        ri = fetch(cpu->ip);
        di = decode(ri);
        exec(di);

        if(interrupt) {
            /* obtain the new instruction pointer from
               the interrupt descriptor table */
            cpu->ip = read_idt(int_vector);
        }
    }
}
```

```
void exec(decoded_instr di) {
    switch(ri->opcode) {
        case . . .
        case . . .

        case IN:
            // …
            a = get_io_addr(di);
            v = io_input(a);

            // …
           break;

        case OUT:
            // …
            v = get_1st_operand(di);
            a = get_io_addr(di);
            io_output(v, a);

            // …
           break;

        // Exception example: division by zero exception
        case DIV:
            so = get_2nd_operand(ri);
            if(so == 0)
                throw exception(DIVISION_BY_ZERO);

            // …
            break;

        // Protection example: the privileged SIDT instruction
        case SIDT:
            if(cpu->privilege_level < SYSTEM)
                throw exception(GENERAL_PROTECTION);
```

```
void exec(decoded_instr di) {
    switch(ri->opcode) {
        case . . .
        case . . .

        case IN:
            // …
            a = get_io_addr(di);
            v = io_input(a);

            // …
           break;

        case OUT:
            // …
            v = get_1st_operand(di);
            a = get_io_addr(di);
            io_output(v, a);

            // …
           break;

        // Exception example: division by zero exception
        case DIV:
            so = get_2nd_operand(ri);
            if(so == 0) {
                exc_type = DIVISION_BY_ZERO;

                /* if this will be called, the program
                   will jump to this point */
                longjmp(&exc_buf);
            }

            // …
            break;
```

```
            /* otherwise, the interrupt descriptor table (IDT)
               pointer is changed, because SIDT stands for
               "Store Interrupt Descriptor Table" */
            cpu->idtptr = …;

            // …
            break;
    }
}
```

```
        // Protection example: the privileged SIDT instruction
        case SIDT:

            // …
            break;

    }
}
```

```
void io_output(operand v, ioaddr a) {
    // Static mapping: not flexible
    switch(a) {
        case 0x40: /* timer */ break;
        case 0x60: /* keyboard */ break;
    }

    iodevice *iodev = io.search(a);

    if(iodev != NULL)
        iodev->set_register(v, a);
}
```

```
void mem_output(operand v, addr a) {
    /* Checks whether the given address corresponds
       to an I/O interface */
    iodevice *iodev = mm.search(a); // can be restricted
                                    // to some mem. region

    if(iodev != NULL)
        // writes in the I/O interface's register
        iodev->set_register(v, a);
    else
        mem[a] = v; // normal write
}
```

```
// I/O device interface (CPU-frontend interface)
class IOdev {
    public:
        // get contents from register at address a
        virtual uint8_t get_register(address a) = 0;

        // write v into register at address a
        virtual void set_register(address a, uint8_t v) = 0;
}
```

```
// Frontend HDD class
class HDFrontend: public IOdev {
    enum { iSN1, iSN2, … };
    static const uint8_t WRITE = …;
    static const uint8_t READ = …;

    uint8_t = SN1, SN2, SN3, SN4;
    uint8_t CMD;
    uint8_t buf[512]; // interface internal buffer
    int next; // next r/w position in the buffer

    HDBackend *be; // frontend/backend split

public:
    // constructor
    HDFrontend(HDBackend *be_): next(0), be(be_) {}

    void set_register(address a, uint8_t v) {
        /* the interface might be mounted at several addresses,
           but the lower bits (acting like an offset) always
           identify the same register */
        int index = a & ADDR_MASK; // like an offset

        switch(index) {
            case iSN1:
                SN1 = v;
                break;

            case iSN2:
                SN2 = v;
                break;

            ...

            case CMD:
                CMD = v;
                break;
```

```
            // the interface internal buffer has been written
            case iBR:
                if(CMD != WRITE) {
                    // error: not consistent
                }

                // writing in the interface internal buffer
                buf[next++] = v;

                /* the interface starts writing in the HDD only
                   when the internal buffer buf is full */
                if(next == 512) {

                    // computing the Sector Number
                    uint32_t sn = SN1 | SN2 << 8 | ...;

                    // real write in the HDD
                    be->write_sector(sn, buf);

                    // array index reset
                    next = 0;
                 }
                break;

            ...
        }
    }

    ...
}
```

```
// Frontend-backend interface for block devices
class BlockDeviceBackend {
    public:
        virtual void write_sector(int sn, const uint8_t *buf) = 0;
        virtual uint8_t read_sector(int sn) = 0;
}
```

```
// Backend HDD class that makes use of files to emulate the HDD
class FileBackend: public BlockDeviceBackend {
    int fd; // file descriptor
    size_t hdsize;
    ...

public:
    // constructor
    FileBackend(const char *filename, size_t hdsize_):
        fd(0), hdsize(hdsize_)
    {
        // creates the file if it doesn't exist
        if((fd = open(filename, O_RDWR|O_CREAT, 0660)) < 0) {
            // host-side error: the emulator cannot continue
            throw ...;
        }
    }

    ~FileBackend() { close(fd); }

    void write_sector(int sn, const uint8_t *buf) {
        if(outsideHD(sn)) { // SN outside the HDD
            // guest-side error: emulating error bit setting
            return;
        }

        off_t offset = sn * 512;

        /* changes the location of the read/write pointer
           of a file descriptor */
        lseek(fd, offset, SEEK_SET);

        write(fd, buf, 512);
    }
    ...
}
```

```
// The TLB is an array of 1024 of these entries
struct TLB_entry {
    uint32_t guest_virtual; /* mapping is b/w pages!
                               the offset must be added */
    uint8_t* host_virtual; // can be NULL
};
```

General strategy

1. Write the emulator as a non-privileged program in the host system
    - The emulator can only interact with the host HW through the OS libraries and primitives
2. Define a data structure for each device (CPU, memory, MMU, I/O devices, …)
3. Write a CPU loop code

CPU emulation

- **All** CPU registers need to be considered, even those that are only accessible to privileged software
    - IDTR: Interrupt Descriptor Table Register
    - CR0: Status control register
    - CR2: Page Fault Linear Address.
      When a page fault occurs, the address the program attempted to access is stored in the CR2 register, so it can be restored in the Program Counter later.
    - CR3: it scontains the Page directory address

  This is because the emulator also has to emulate the **system software**, including interrupts, exceptions and protection.

Interruption

- Flag for the interrupt request
- Variable for the interrupt type
- The CPU loop must check the interrupt flag after every instruction execution
    - If set, the CPU must load the interrupt handler address in the emulated Instruction Pointer, and start fetching again

Exceptions

- Examples
    - Division by 0
    - General protection
    - Page fault
    - By the `read_idt`
        - Gate not present
        - Protection
        - Page fault
- They may occur during fetch, decode and execution
- Handled by the try…catch construct

Protection

- Example: the `SIDT` x86 instruction (Store Interrupt Descriptor Table) is a privileged instruction
- Handled in the `exec()` function

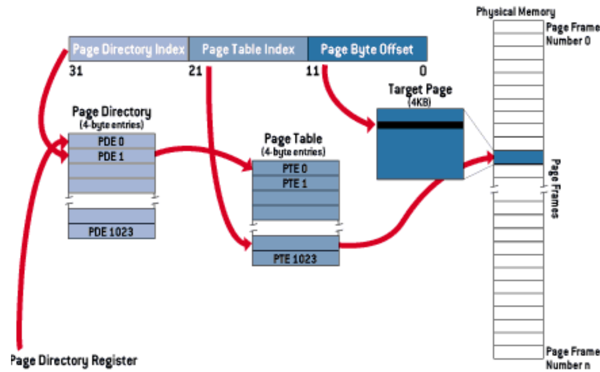
I/O devices

- How they interrupt
    - I/O devices do not send interrupts directly to the CPU
    - The CPU only has one interrupt request pin for all I/O devices
    - The target system may have an APIC (Advance Programmable **Interrupt Controller**)
        1. It collects all interrupt requests
        2. It prioritizes them
        3. It sends one interrupt at the time to the CPU, with an interrupt source ID
        4. If interrupts are enable, the CPU jumps to an interrupt handler routine
        5. Upon finishing the routine, the CPU writes an EOI word in an APIC register
        6. The APIC is then enable to pass the CPU any possible lower priority interrupts
- Interaction
    1. I/O devices are connected to the rest of the system via **interfaces**
    2. Interfaces have a set of registers mapped in I/O or memory space
    3. I/O registers look like normal memory locations
        - Difference: **reads and writes involving I/O registers correspond to real actions**
        - That's why they need to be mapped to function (which do some action) instead of simple memory locations
        - In x86, I/O space can only be accessed via the `in` and `out` instructions
- `io` is a data structure mapping I/O addresses
- Memory mapped I/O: "*Memory-mapped I/O uses the **same address space to address both memory and I/O devices**. The memory and registers of the I/O devices are mapped to (associated with) values. So **when an address is accessed by the CPU, it may refer to a portion of physical RAM, but it can also refer to memory of the I/O device**. Thus, the CPU instructions used to access the memory can also be used for accessing devices*"
    - The instruction in charge of writing in memory has to check whether the given address corresponds to an I/O interface
    - If it does, the instruction needs to write in the interface's register
- I/O asynchronous events
    - Intermediate I/O interface's registers values can be **skipped** during emulation if the action is implemented by a (mostly) non-blocking operation in the host system, i.e. showing characters on terminal
    - They can be implemented with *non blocking* I/O in the host system.
        - This option can be set on a file descriptor (even those connected to a terminal, the first three)
        - Any `read` system call **will return an error** if no input is available, **instead of blocking**.
            - If it returns error, the emulated input instuction can return the previously read character.
    - Using interrupts
        - The CPU can periodically check a new keyboard value with a `read` system call
        - One thread for a CPU loop and **one** or ~~more~~ thread for the I/O devices
            - The keyboard thread can be blocked in the `read` system call, ad then we can set the interrupt flag (in shared memory) whenever the system call returns
            - Most emulators have just one thread for all the I/O devices because it's block by the `select` system call, that check all file descriptors (the **I/O `select` loop**)
            - The interference problem is still present
                - "*Performance degradation observed by an application in contention with other applications for the access to a shared resource*"
- I/O devices
    - Implemented as objects
    - Recall: interface's registers can be addressed
        - In a separate address space

    - In memory space (memory mapped I/O)
  - Common programming interface (just read/write functions)
    - Object representing the I/O devices will implement these functions
    - These objects have to interact with the host OS
      - There will be another interface b/w the I/O object and the host OS
    - Splitting a device emulation code
      - *Frontend*
        - It only depends on the device (its registers and what they do)
        - It implements the interface b/w the emulated CPU and the emulated device (the *CPU-frontend* interface)
      - *Backend*
        - It depends on how we are using the host to emulate the device (using a file, a terminal, …)
        - It implements the interface b/w the emulated device and the host (the *host-backend* interface)
      - In some emulators these two objects interact through the *frontend-backend* interface
        - This allows any equivalent emulated device to any possible host resource
        - It depends on the kind of emulated device
      - ! Both frontend and backend can be accessed by both threads (the CPU one and the I/O `select` one)
        Typical interaction:
        - CPU thread
          1. It executes an I/O instruction by calling the corresponding `set_register()`/`get_register()` method on the device
          2. This method will update the I/O object state and will call into the backend (using the *frontend-backend* interface) to **complete** the I/O operation
        - I/O thread
          1. It may exit from the `select()` and find out that **the same** device needs to be updated (e.g.: a new key has been pressed on the keyboard)
          2. It will call some backend function (from the *host-backend* interface) which will in turn call some function in the frontend (from the *frontend-backend* interface)
  - Assumption: each interface function is protected by per-object mutual exclusion
- Hard disk
  - Registers
    - SN1, SN2, SN3, SN4: Sector Number
    - CMD: operation type
    - BR: interface internal buffer
  - Write
    1. Write the Sector Number in SN1, …, SN4
    2. Write the "write sector" code in CMD
    3. Sequence of writes in BR
       - When it's full, the interface starts writing in the HDD
- VGA-compatible video adapter
  - Two modes
    - Text mode: display organized in rows and columns
      - Each byte takes 2B
        - 1B for ASCII
        - 1B for the background color and blinking
    - Graphic mode: pixel matrix, colors must be set
  - The video adapter scan the entire **video memory** in 60fps and displays the output
    - Video memory emulated by a plain buffer: writing in it causes no immediate side effect
    - The I/O thread periodically reads it and draw the corrisponding content in a window
- APIC (interrupt controller)
  - `IOdev` object
    - The CPU must be able to read and write into its registers (like EOI)
    - I/O devices (the I/O thread) must send their interrupts to it, calling a specific method in its interface
      - If there are no other interrupts, the APIC interrupts the CPU
      - Otherwise, it queues the interrupt request
    - `set_register()`
      - Upon writing in EOI (by the CPU [thread]), other possible interrupts should pass

Virtual memory

- The MMU (Memory Management Unit) translates virtual memory addresses into physical ones
  - Pagination can be disabled
- Physical memory is represented by a byte array called mem
- Names
  - Example: `Mem[4100]` will be translated in `Mem[8196]` by the guest MMU
    - `Mem[4100]`: *guest virtual address*
    - `Mem[8196]`: *guest physical address*
    - `&Mem[8196]`: *host virtual address*
      - Address of the location the emulator will actually read
    - The host MMU will translate `&Mem[8196]` into the *host physical address*
- Translations
  - *guest virtual address* → *guest physical address*
    - The guest MMU needs to be emulated
    - `translate_address()` whenever the CPU needs to access guest memory
      - During loads and stores
      - During fetches, since instructions are in memory
    - MMU steps
      - MIT's translation diagram
      - Address is divived in
        - Directory (10 bits)
        - Table (10 bits)
        - Offset (12 bits)



1. Obtaining the page table descriptor

```
CPU->cr3 + 4(Bytes) * ((virtual_address & 0xFFC00000) >> 22)
```

   - Each page is 4KB
   - The `0xFFC00000` mask gets the first 10 most significant bits (*Page Directory Index*)
   - It is then shifted to the right to 22 bit locations in order to add it to `CPU->cr3` (the *Page Directory Address*)
2. Obtaining the target page
   - By using the page table previously retrieved
3. Using the *Page Byte Offset* as it is to get the right word within the physical page

- TLB
  - It caches recently used translations
  - In SW it can be faster than it would be in HW because it could require less instructions
    - Array of

```
// The TLB is an array of 1024 of these entries
struct TLB_entry {
    uint32_t guest_virtual; /* mapping is b/w pages!
                               the offset must be added */
    uint8_t* host_virtual; // can be NULL
};
```

    - In this way, it'll translate `4100` into `&Mem[8196]`
  - Usage
    - An **hash** is applied to the *guest virtual address* in order to find its entry in the TLB
      - 10 lower order bits of the *guest virtual page address* is ok
    - If the `host_virtual` entry is not null, the TLB works
    - Otherwise, ordinary translation, and then the translation result is stored in the TLB

- *guest physical address → host virtual address*
  - The *guest physical address* is used as an index in the `mem` array
    - Example: 8196 in `&Mem[8196]`
- *host virtual address → host physical address*
  - Translated by the host MMU

The IA-32 or i386 instruction set architecture (ISA)

- IA-32 stands for "Intel Architecture, 32-bit"
- Emulation is hard because of
  - Complex instruction fomat
    - Recall: `disp(base, index, scale) = base + (index * scale) + disp`
    - ! They have variable length
    - Structure

	[Prefix]	Opcode	[Mod R/M]	[SIB]	[Displacement]	[Immediate operand]
Length	$[1 \div 4\ B]$	$1 \div 3\ B$	$[1\ B]$	$[1\ B]$	$[1, 2, 4\ B]$	$[1, 2, 4\ B]$
Notes	• REP, REPE, REPNE for string instructions • LOCK, to lock the bus for atomic operations • "large" operands default size	Might include up to 3 bits that encode a register operand • `pushl %eax` encoded in just 1B	Encodes the mode of one of the two operands • Register • Immediate • Memory ○ ...	It encodes: • Scale • Index • Base of memory operand expressions like: `16(%ebx, %ecx, 4)`	The "16" of `16(%ebx, %ecx, 4)`	Encodes the value of operands, such as `$1000`

  - `EFLAGS` register
    - ! Almost every instruction update it
      - Arithmetic, compare, logic, shift and rotate instructions...
      - Its updates can be delayed since `EFLAGS` is read only by conditional jump instruction
    - Some flags are complex to compute
      - Parity flag: set if the result is even
      - Adjust flag: set if there's benn a borrow/carry out of the four LSBs in the result

Binary translation

Wednesday, August 30, 2017 16:37

What is it

Translating guest code in host code

Translation cache (caching host codes)

- Avoid translating frequent guest codes
- Translated code can be optimized since the emulator knows the whole code
- The CPU loop is improved
  - I/O, virtual memory and multi-threading code remains the same

Translation

- DBB (*Dynamic Basic Blocks*): guest code block
  - It starts with an instruction which is the target of a jump
    - In this way, the corresponding bytes pointed by the jump instruction are certainly instructions
    - It's certainly followed by other instrucitons, until another branch or jump
  - It ends after the first branch or jump instruction
    - A DBB stops even after **unconditional** jumps, because it's impossible to determine wheter the following bytes are instructions or data
  - Thanks to DDBs it's possible to only translate code that's actually going to execute
  - They contain target instructions in the target memory
- TB (*Translated Block*): translation of a DBB
  - They contain host instructions
  - Identified by the guest address of the first instruction in the corresponding DBB
    - In this way, after the execution of a TB, the current value of the guest instruction pointer can be used to find the next TB to execute
- CPU loop modification

```
for(;;) {
    tb = find_in_cache(CPU->ip); // pointer to a TB descriptor
    if(!tb) {
        tb = translate(CPU->ip);
        add_to_cache(tb);
    }
    exec(tb, env); // env contains the guest environment (CPU, memory, ...)
}
```

- State snapshots are taken just before the execution of each DBB
  - During the execution of each TB, instruction rearrangements, omissions and optimizations are allowed, since intermediate $V - states$ must not necessarily have their corresponding $T - states$

Translation optimizations

1. Constant propagation

- Replacing a register or memory operand with an immediate operand
  - It can be done when the register content or memory operand is known to the optimizer
- Example

```
movl $0, %eax
... // no other updates involving %eax
incl %eax
movl %eax, %ebx ⟹ movl $1, %ebx
```

(In the last line, `movl %eax, %ebx` is struck through.)

2. Dead code elimination
    - Removing host code that cannot affect the state
    - Examples
        - Avoiding updating EFLAGS

```
addl %eax, %ebx /* flags not computed: they would be overwritten by the
                   next operation */
subl $1, %ebx
```

        - Avoiding updating guest IP register (contained in the cpu_state structure in the emulation code)
            - It can be updated at the end of a DBB
                - Relative jump case: the IP can be updated only before a relative jump instruction
                - Subroutine call case: the IP is usually placed on the stack when calling a subroutine. This value will come back in the IP register after a RET instruction, so it's useless to update the IP **during** the subroutine code
3. Register allocation
    - During a DBB, a guest register can be allocated in a host registers (faster than memory)
    - Mapping can be different for each DBB
        - Frequently accessed guest registers may be mapped always in the same host register, like the guest ESP (stack pointer) register
4. Lazy condition code computation
    - Instead of updating the EFLAGS register at the end of each DBB, the operands and the result of its last instruciton could are saved, in case a following guest instruction needs the EFLAGS value
    - Most of the times, the next DBB's first instruction will overwrite the EFLAGS register
5. Translated block chaining

```
for(;;) {
    tb = find_in_cache(CPU->ip); // pointer to a TB descriptor
    if(!tb) {
        tb = translate(CPU->ip);
        add_to_cache(tb);
    }

    // The old TB can be patched in order to avoid returning to the
    // CPU loop, and start executing the next TB instead.
    // If the last instruction is a conditional jump, two possible TBs
    // will then be linked

    exec(tb, env); // env contains the guest environment (CPU, memory, ...)
}
```

Bynar translator problems

1. Handling interrupts
    - Interrupts could be handled after a DBB's translation
    - However, if chaining is used, the `exec()` function could take a lot of time
        - **Chaining must be disabled** if there is a **pending interrupt**
            - Equivalent to putting the interrupt checking code at the end of each TB, before jumping to the next one
2. Handling faults
    - Faults can happen within a DBB
    - Unlike interrupts, they cannot be delayed
    - They can be handled with a `longjmp()` invocation that jump to the main loop
        - The corresponding `setjmp()` will then read the guest interrupt table and change the guest IP accordingly

- The guest fault handler may need the contents of all guest registers
  - Registers content reconstruction depends on used optimizations
    - Register allocation: the table that maps guest registers into host registers can be used to update all guest registers
    - Dead code elimination: guest state updated before any instruction that could cause a fault
    - Most complicated cases: a consistent state should be restored and DBB translation should be disabled (single instructions should be fetched and executed instead: this is called ***switching to simulation***)

3. Virtual memory

- Previous examples omitted virtual addresses translation
- Addresses must be translated from virtual to physical during each operation involving memory
  - Code is the same
  - Software TLB can still be used

4. Self modifying code

- Translated block containing modified code must be invalidated
- Since code is just data in the guest memory, **any** guest write to memory must be checked
  - SW only solution
    1. Mantain in some data structure the range of guest addresses containing translated code
    2. Check it before each guest memory write
  - SW + host HD solution
    - Steps
      1. Write-protect guest memory parts containing already-translated code
      2. Upon a page fault involving a page containing modified code, invalidate the corresponding translation cache entry so the CPU will be forced to re-translate the DBB containing code
    - Problem
      - The binary translator is an unprivileged programing running on a OS
      - It doesn't have direct access to page tables
      - It cannot directly intercept page faults
    - Solution
      1. Allocate guest memory with `mmap()` (so `mprotect()` can be used later on)
      2. Set a function to handle the `SIGSEGV` signal with `signal()`
         - Normally, the SIGSEGV signal is generated by the kernel whenever the current process tries to access memory in a manner that violates the protections (e.g.: program trying to read/write outside the memory it is allocated for it)

           From <http://man7.org/linux/man-pages/man2/mprotect.2.html>
         - In this case, `SIGSEGV` will be generated when a process tries to write into a protected TB page containing code
           - Normally, this would cause the termination (with a "`Segmentation fault`" message)
           - The specified handler **invalidates the translation cache** so the CPU will be forced to re-translate the DBB containing code
      3. After a DBB translation, use `mprotect()` to remove the write permission for the corresponding pages in the guest memory
         - `void*` *`addr`*`, size_t` *`len`*: to specify the corresponding pages
         - `int` *`prot`* `= PROT_READ | PROT_EXEC` for read-only permission
    - Code block tries to modify itself
      - This TB cannot invalidated in the translation cache since it is currently being executed
      - Switch to simulation (emulating one instruction at a time) until the execution of this block is over

HW assisted virtualization

Sunday, September 17, 2017 17:38

What is it

- Executing target machine instructions directly on the host processor, as much as possible
- "*Hardware is needed to maintain the correspondence b/w the virtual machine state and the target state*"

Why

- It obtains a large speedup w.r.t. to classic emulation

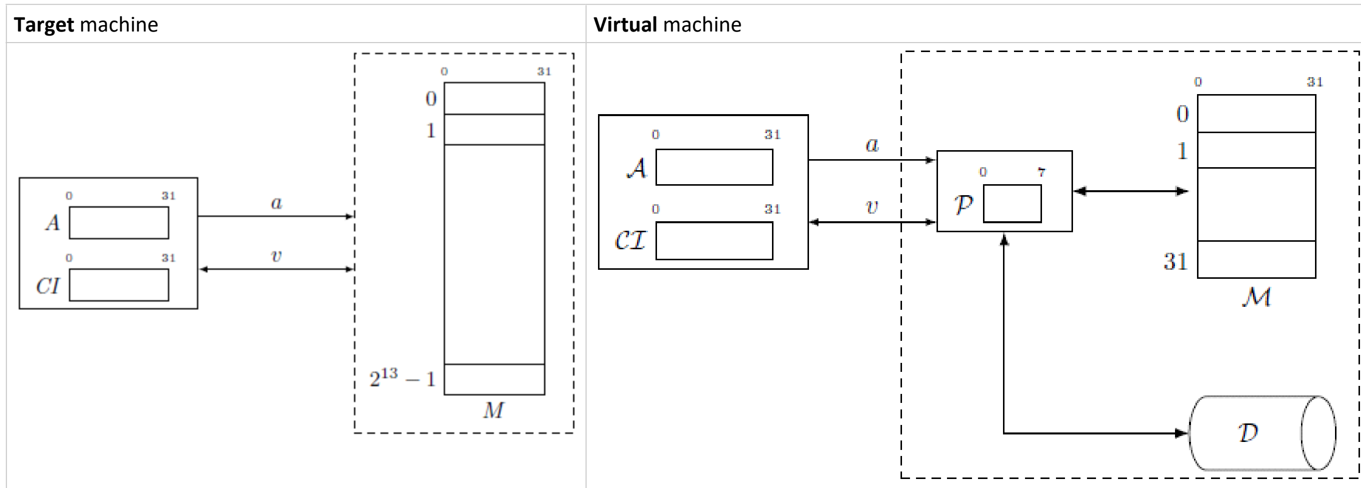
When it can be done

- When the host machine understands a superset of the target machine instructions
  - There can happen that they both have the same architecture
- When flexibility and security are the real virtualization motivations
  - If architectures are similar, it means that the target machine hardware is not so unavailable

Simple HW assisted virtualization examples

1. Making full use of the Manchester Baby's *addr* register

- The Machester Baby has 32 words of memory, but the *addr* register is 13 bits wide
- To use the *addr* register completely, we can emulate a machine with $2^{13} = 8196$ addressable words
- Instead of buying expensive additional tubes to display all the 8196 memory contents at once, we can add an MMU and another cheaper storage in which we can store everything
- The MMU will have a register P containing the word that is currently in the Machester Baby machine, and it will swap pages from the cheaper memory storage D (acting like an HDD) to the Manchester Baby machine, that has only 32 words of memory (hence it can just contain one page at a time, acting like a sort of cache)
- The MMU and the additional cheaper memory storage are not part of the original Manchester Baby machine, but all of these 3 components belong to the *target machine*
- In short:



- Formalization
  - Snapshots are still taken before fetching a new instruction
  - Machines
  - $T - state = \langle A, CI, M \rangle$
    - A: accumulator. The only register that the machine can use temporarily
    - CI: instruction pointer
    - M: memory (8192 words)
  - $V - state = \langle A, CI, M, P, D \rangle$
  - $interp_M()$ function that maps **virtual memory into the target memory**
    - $X = interp_M(M, P, D)$ is a 8192-elements vector representing the target memory
      - Its elements:

$$X_a = \begin{cases} |M_a|_{32} & if\ P = \lfloor a/32 \rfloor \\ D_a & otherwise \end{cases}, \quad for\ 0 \le a < 8192$$

Where M is the real 32-words memory.
If a is such that $P = \lfloor a/32 \rfloor$, it means that the location at address a is actually in the real 32-words memory.

  - Interpretation function $interp(V - state) = T - state$
    - Defined as $interp(\langle A, CI, M, P, D \rangle) = \langle A, CI, interp_M(M, P, D) \rangle$

- $Target \neq Host$
  - Target machine has a bigger tube memory
  - Host machine has a cheaper storage memory D
- In this case it's the MMU that does all the translation job.
  - If real memory would have had more than 1 word in its memory, part of the virtual machine logic would have been implemented in software
    - The MMU would only translate addresses for pages already loaded in memory
    - It would raise exceptions for those which are not in it
    - Exception would cause the execution of virtual machine software that implements swapping
  - Additional hardware would still be needed
    - In fact, the exception is raised via HW, and the swapping part is made by SW

2. The virtual processor: multiprogramming

- **Multiprogramming**: emulating a target machine which has more processors than the host machine
  - Host *phyisical* processor = *host* processor
  - *Virtual* processors are multiplexed on *host* processors
- ! Assumption: target machine has shared memory
  - Otherwise each target process' private memory should be be virtualized

2.1 Simple case: one host processor

- $T - state$: register contents of all virtual processors
- $V - state$
  - Register contents of the *host* processor
  - One data structure for each virtual processor, containing a copy of virtual processors registers
  - Current virtual processor ID
- Virtual processor change consists in loading all processor's registers and let it continue its execution
  - Virtual processor changing could be timed by a timer
  - This *context switching* (un/loading processors' registers) is made in SW
- **Hardware assistance**
  - HW could invoke the context switching SW (e.g.: **timer**)
  - The data structure containing all virtual processors' registers contents should be only accessed by SW that implements the virtual machine, and not by the target SW running on virtual processors
    - Can be done by introducing ***privilege levels*** on the *host* processor: "*system*" and "*user*"
      - In the "*user*" level, the processor cannot access to that data structure
      1. The timer changes the privilege to the "*system*" level
      2. This causes jump to the virtual machine SW that performs the context switch
      3. A special system-instruction can then return at the "*user*" mode

Virtual machines

Sunday, September 17, 2017 23:07

What are they

- A **complete** virtualization of a computing system
  - Full processors
  - Memory
  - I/O peripherals
- Virtual machines **emulate OSs**
  - The OS has access to privileged registers (`%cr3`, `%idtr`, ...) and instructions that manipulates them
    - They cannot be executed directly as they would affect the state of the entire host machine. Just virtual register copies should be affected
    - At the same time, instruction should be executed directly on the host HW to improve performance

Trap and emulate

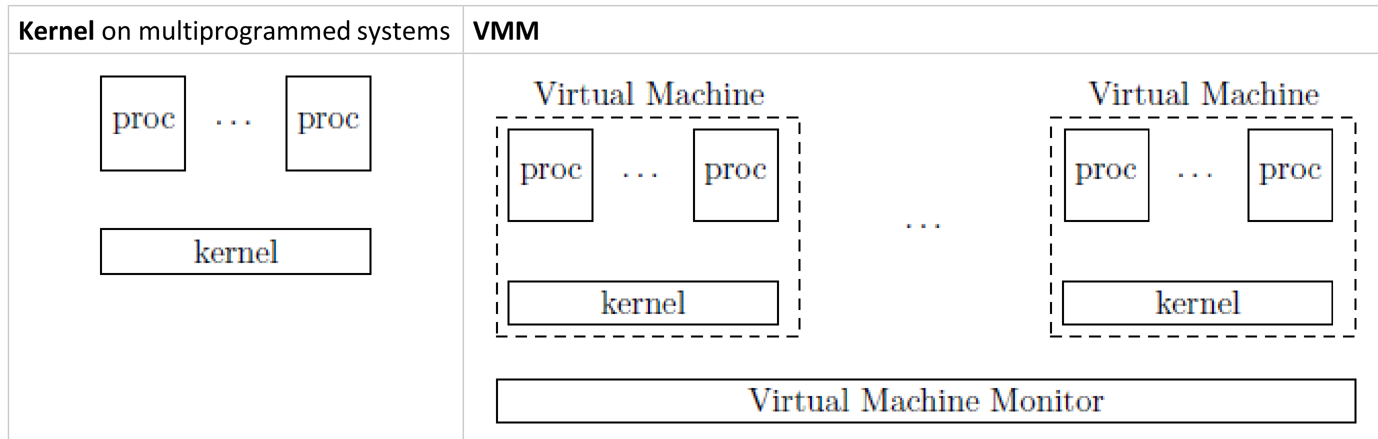
- Target and host architecture is the same
- Two levels of privilege
  - System: virtual machine monitor (implementing the virtual machine)
  - User: software running inside the virtual machine (in this case, originally written for the target one)
- Result: target machine system SW will **only** run at user privilege
- Handling target machine's privileged instructions
  - The host processor raises exception for privileged instruction at user level (hence by the target SW)
  - The virtual machine monitor intercepts (handles) them and **emulate their effect on the virtual state** (this is why it's called *trap and emulate*)
- Trap-and-emulate virtual machine monitors cannot be implemented on Intel x86 processors
  - Why
    - Some privileged instructions do not raise exceptions when executed at user level
      - `popf` instruction: it might be used to disable host interrupts.
        This instruction "pops" 2 words from the stack and stores them in the `EFLAGS` register. Since `EFLAGS` contains the interrupt flag `IF`, if a target SW tries to execute this instruction, it may think that `IF` has changed, while it's not changed both on the virtual machine and on the host machine (that doesn't raise exceptions, but it won't execute it either)
    - Exceptions are raised upon writing in special registers, but reading is allowed
      - The SW running inside the VM may detect it's not running on the host machine by comparing the actual content contained in the `%cr3` register with the value that the SW is trying to write in it
  - VMware's solution
    - Hardware-assisted virtualization for all target **<u>userspace</u>** SW
    - Switching to binary **translation** (one instruction at a time) for target **<u>system</u>** SW
- Intel and AMD have virtualization extensions on their processors

The virtual machine monitor

Monday, September 18, 2017 16:06

- "Kernel" of VMs
- VMM/kernel analogy

Kernel on multiprogrammed systems	**VMM**



- *Guest*: SW running inside a VM
- The VMM supervises the execution of the *guests*, so they cannot interfere with each other and with the host
  - As kernel does it with processes
  - Each *guest* may be a multiprogrammed system with its own kernel that manages processes

Intel VMX

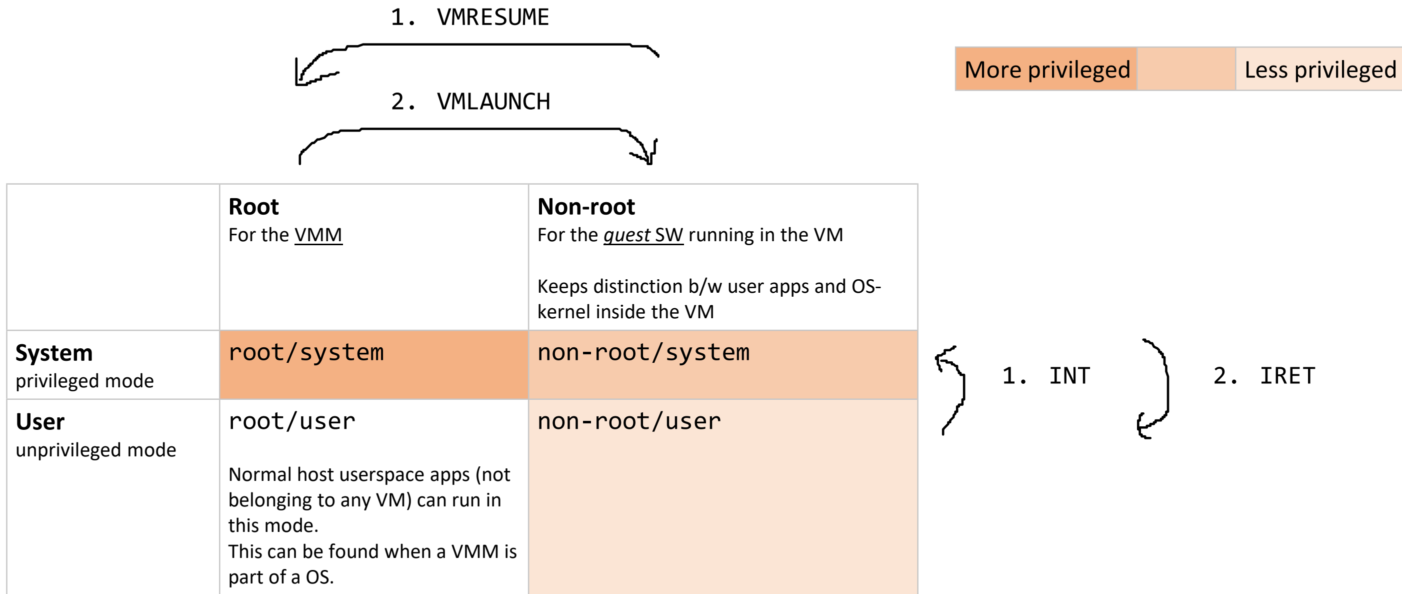
Monday, September 18, 2017 16:15

What is it

- Intel's CPU extensions designed to support VMMs
  - Also AMD has done this, but it's not compatible with Intel VMX

Root and non-root modes

- Intel CPUs have now 4 modes



	Root For the VMM	**Non-root** For the *guest* SW running in the VM Keeps distinction b/w user apps and OS-kernel inside the VM
System privileged mode	`root/system`	`non-root/system`
User unprivileged mode	`root/user` Normal host userspace apps (not belonging to any VM) can run in this mode. This can be found when a VMM is part of a OS.	`non-root/user`

- They've been introduced to put HW-controlled limitations to the guest SW
- Mode-switching instructions
  - Typical cycle
    1. Normally the guest SW is running in the VM (**non-root** mode)
    2. Whenever the system code tries to execute an instruction that
       - would violate the isolation of the VMM
       - must be emulated via SW

       The HW can trap it and switch back to the VMM via the **VMRESUME** instruction
       - So the VMRESUME instruction goes in root mode (returning control to the VMM)
       - This is a *VM exit*
    3. The VMM will complete the emulation of the action initially executed by the guest code
    4. The VMM gives control back to the guest via the **VMLAUNCH** instruction
       - So the VMLAUNCH instruction goes in non-root mode (returning control to the guest SW running in the VM)
  - The VMRESUME and the VMLAUNCH instructions are only allowed in `root/system` mode

The virtual machine control structure (VMCS)

- The VMCS contains all the infos needed to manage the new non-root mode
- One VMCS for each processor of each VM
  - Only one VMCS at the time is the *current* one on the physical processor
  - The processor has a register pointing to the current VMCS
    - VMPTRLD ("*VM pointer load*") instruction to load a new VMCS address, making it the new current one
  - All VM instructions (VMRESUME, VMLAUNCH, ...) use the current VMCS
- Fields

? `Don't know`	• Posted Interrupt Descriptor (PID) pointer
`Guest state`	Virtual processor's state (registers) • Loaded from here during a VM enter • Stored back here during a VM exit

	Interesting registers: • The `%ip`'s content determines ○ The first instruction that the VM will execute ○ The last instruction that caused a *VM exit*
`Host state`	Physical processor's state (registers) • Loaded from here during a VM exit 💡 Thanks to this, the guest can manipulate real processor's registers without affecting the host state. Interesting registers: • The `%ip`'s content determines the point from which the host machine will keep executing. This should be the entry point of a VMM routine that 1. Will examine the *exit reason* 2. Will perform the necessary emulation 3. Will re-enter the non-root mode (VMLAUNCH)
`VM execution control`	What's allowed and what's not during **non-root** (VM) mode • Unallowed action will cause a *VM exit* Interesting flags: • CPU behaviour upon receiving an external interrupt while running in non-root mode ○ CPU may serve the interrupt using the guest's `IDT` w/o leaving `non-root` mode ○ VM exit: the VMM regains control regardless of the guest's `IF` value • Should some critical instructions cause a *VM exit* or not? ○ Flags for `htl`, `invlpf`, reading/writing `%cr3`, … • Should some exceptions cause a *VM exit* or not? ○ Page faults, … • Should I/O operations cause a *VM exit* or not? ○ Generic flag for **any** in/out instructions ○ Bitmap with a flag for each of the 65536 possible I/O registers (8 KiB)
`VM enter control`	Behaviour during the root → non-root transition • Fields to **ingect an event** during a VM enter ○ Fake external interrupt ○ Exceptions ○ Faults
`VM exit control`	**Behaviour** during the non-root → root transition • External interrupt: should the CPU obtain the interrupt vector? Recall: 1. The interrupt controller sends the request 2. The processor has to reply and obtain the interrupt vector from the interrupt controller ○ Yes 1. It obtains the interrupt vector during the *VM exit* ○ For external I/O interrupts coming from passed-through devices, it obtains it from the Notification Vector NV from the Posted Interrupt Descriptor PID 2. It stores the interrput vector it in the VMCS 3. The *VM exit* jumps to the address stored in `Host state` 4. The VMM reads the vector in the VMCS and jump to the interrupt handler via SW ○ No ▪ The *VM exit* jumps to the address stored in `Host state` ▪ Since external interrupts are disabled during the *VM exit*, the VMM re-enables them via the `STI` instruction ▪ The processor will complete the protocol with the interrupt controller □ It obtains the vector □ It jumps to the proper interrupt handler
`VM exit reason`	• Code identifying the reason • Additional info • I/O register's address • Exception type

Examples

- Some instructions like `popf` and reading from `%cr3` were difficult to virtualize on the Intel x86

1. `popf`

- ❗ Recall: it might be used to disable host interrupts, since it pops 2 words from the stack and stores them into `EFLAGS`, that contains the interrupt flag (`IF`) bit
  - It cannot be executed at host user privilege since the CPU won't raise any exception and it won't even change the guest `EFLAGS` content, that should be modified
- With VMX, the `popf` instruction can be executed in any non-root mode
  - The guest system software will change its `IF` value
  - Host interrupts won't be disabled if it doesn't want to
    - By setting the proper flag in the `VM execution control section`, the CPU will decide wheter to receive interrupts or not while running in non-roote mode

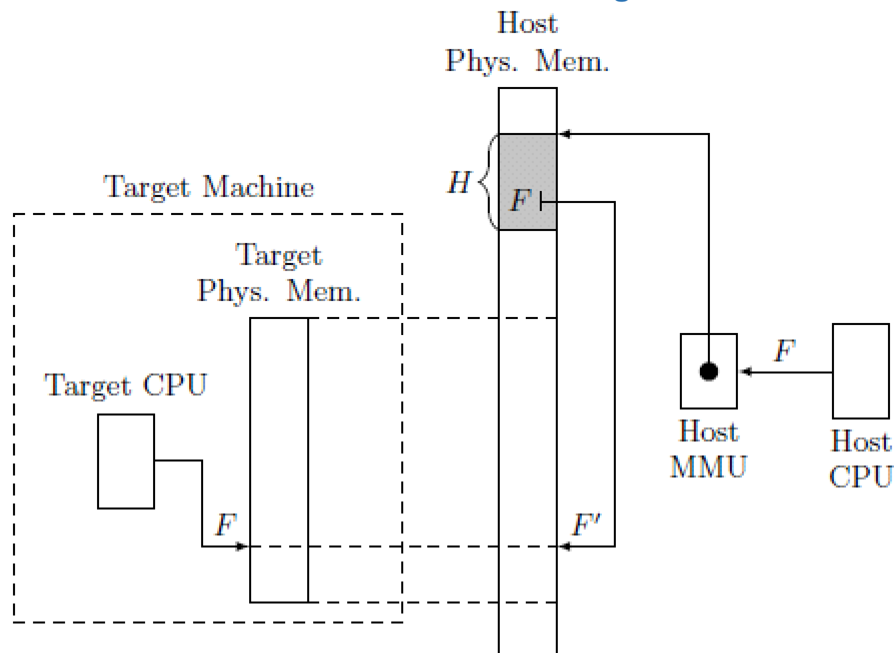
2. `mov %cr3, %eax`

- Recall: this instruction doesn't case a fault
  - The guest is not allowed to write in `%cr3` since it would give him full access to the host memory
  - ❗ The guest might realize it's virtualized by comparing what it has written in `%cr3` and what it can read from it
- With VMX, any read to `%cr3` can cause a *VM exit*.
  The VMM will then:
  1. Put in `%eax` the guest's expected value
  2. Skip the `mov %cr3, %eax` instruction by incrementing the guest's `IP` in its VMCS's `Guest section`
  3. Return to guest mode with the VMLAUNCH instruction

Virtual memory virtualization

Wednesday, September 20, 2017 10:19

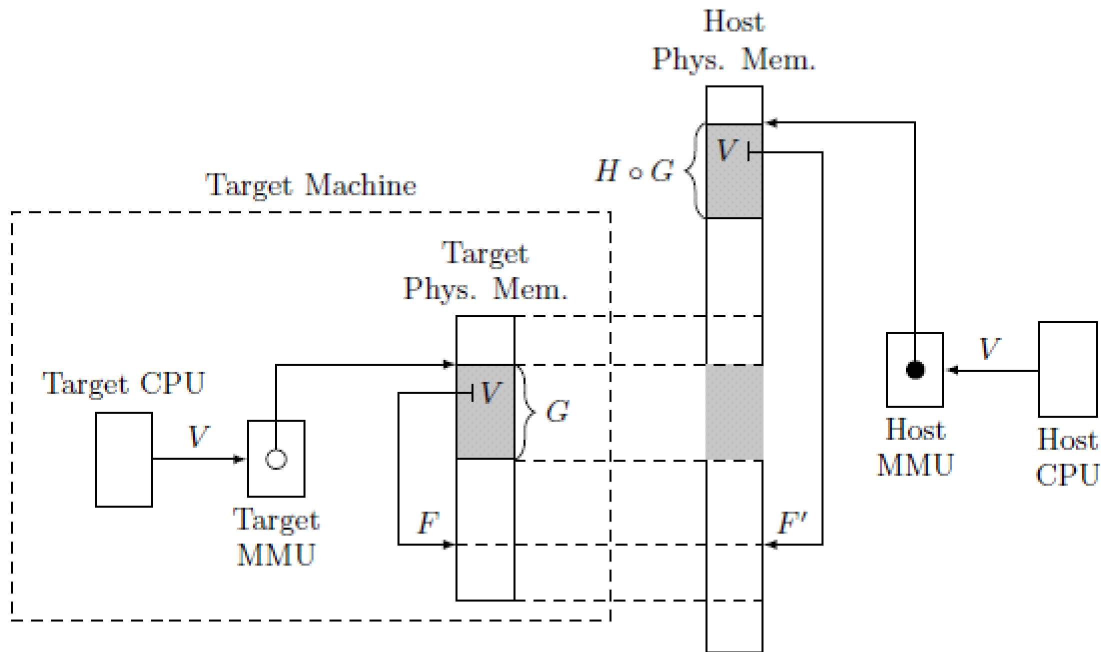
HW-assisted virtualization: when the target machine **had no virtual memory**



- Target (*guest*) physical memory is a subset of host physical memory
  - Guest system SW must
    - Only access to its dedicated host physical memory
      - Otherwise it'll be able to interfere with other VMs and with the VMM
    - Think that it has access to all physical memory, starting from address 0
- *Guest*-host phyiscal memory mapping is done by the host MMU
  - H is the translation data structure (page directory and present page tables for a guest) that translates guest physical addresses into host physical addresses
  a. Assume the target machine is accessing address F
  b. In HW-assisted virtualization, also the host CPU will be accessing address F, since both CPUs will be executing the same instructions for most of the time
  c. This will be translated in F' by the host MMU using H
- There's a similary with the virtual-memory translation that happens on a classical computer with no VMs on it

Virtual MMU

This doesn't exist in real life	This is the only part that exists



- When target machines have virtual memory, they have their own target MMU
- Maps
  - The guest has its own G map where any virtual *guest* address V is mapped into physical *guest* address F
  - The VMM has a mapping H that maps any physical *guest* address F into a host physical address F'
- Combined effect: whenever the guest SW wants to access V, the host CPU must access F'
  - Host's complete translation

$$V \xrightarrow{\boldsymbol{G}\,(guest's\ MMU)} F \xrightarrow{\boldsymbol{H}\,(physical\ guest-to-host\ translation)} F'$$

  - The VMM must be able to
    - Build the $H \circ G$ map that implements both translations
    - Let the host MMU point at the page tables implementing $H \circ G$ while the guest is running

- Building the host page tables $(H \circ G)$ and let the guest modify the G mapping
  - The brute force method
    - "Brute force": updating host page tables as soon as the guest modifies the G mapping (the guest page tables)
    - The guest can modify the G mapping in two ways:
      - Writing in `%cr3` (`mov %eax, %cr3`)
        1. The VMM may set up the VMCS so that each write to `%cr3` from non-root/system mode causes a *VM exit*
        2. When the VMM re-gains control, it can read `%eax` (V) and learn the guest physical address F of the page directory that the guest was trying to install (by using the guest/target 's MMU G, I guess, via the VMM software)
        3. The VMM can use the H map to translate this guest physical address into a host physical address F'
           - Actually done at once with the $H \circ G$ map
        4. The VMM is now able to read the guest's page directory
        5. The VMM can then **load** all guest's page tables, and so on
           - The VMM uses all this information to prepare its host page directory and page tables via $H \circ G$
        6. The VMM writes into `%cr3` the host physical address of the page directory

it has created
  - Does he mean the actual `%cr3` host's register?
    - If so, will this write be propagated to the VMCS?
7. The VMM modifies the `%eip` field in the `Guest state` part of the VMCS so the guest will skip this instruction
8. The VMM returns control to the guest via the VMLAUNCH instruction

- Changing some entries in the page tables in memory (in the G mapping)
  - Could be done with any write in memory with an address that involves a page directory/table
  - Steps
    1. The VMM may write-protect, in the host page tables, the pages that contain the active guest page directory/tables
    2. The VMM sets up the VMCS so that any write to a write-protected page causes a *VM exit*
    3. Upon examining the VM exit reason, the VMM must check if the address that the guest was trying to write into falls within any active page directory/table or not
    4. If it does, the VMM must decode the instruciton and update:
       - The host page tables
       - The gust page directory/table
    5. The VMM then re-enter the VM skipping the trapped instruction
- The VM must be set up to trap both actions
- Cons
  - Complex to implement
  - Too many *VM exits*

The virtual TLB method

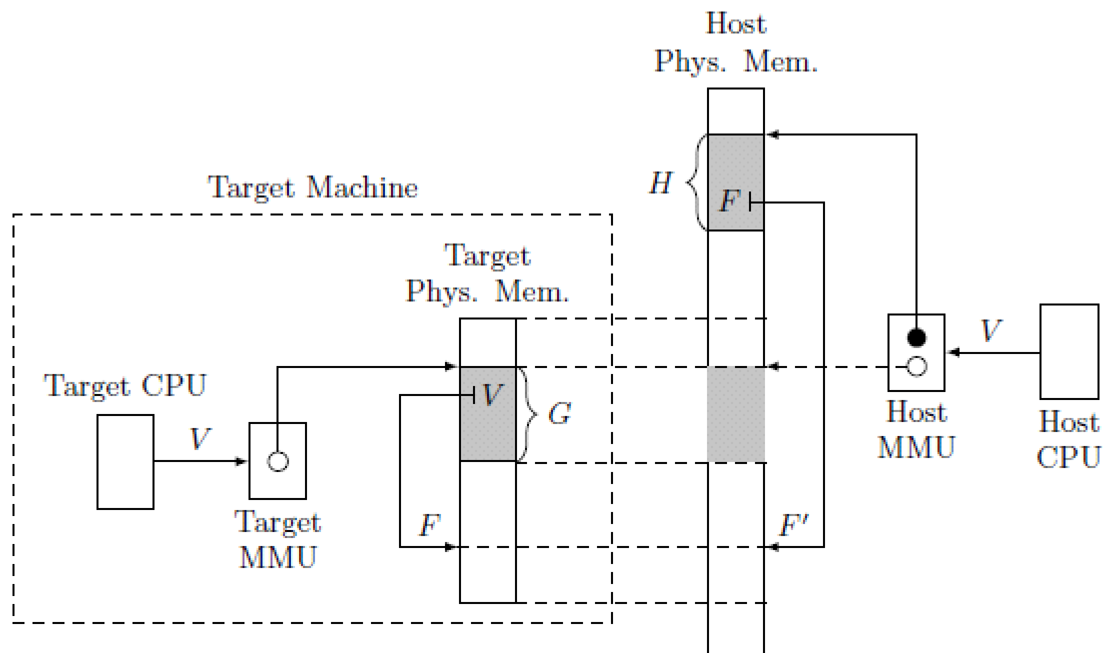
- Host page tables "like the MMU's TLB", as a "virtual TLB"
- There must be a *VM exit* on each `invlpg` (invalidate page instruction)
- There must be a *VM exit* on each page fault
- The guest can modify the G mapping in two ways:
  - Writing in `%cr3`: there must be a *VM exit*
    - So the VMM may operate as in the brute force method
  - Changing some entries in the page tables in memory (in the G mapping)
    1. Guest page directory/tables are not write protected
    2. This won't cause a *VM exit*
    3. Therefore, host page table **will be out of sync** w/ guest page tables
    4. Presence-bit change example
       - Guest had made present a new guest virtual page
         a. The host will cause a page fault, hence a *VM exit*
         b. The VMM examies the guest page tables and finds out that the page was actually present, and updates the host page tables accordingly
         c. The VMM can re-enter the VM
       - Guest had made non-present an old guest virtual page
         a. The only ways to invalidate the TLB are
            - Writing in `%cr3`
            - The `invlpg` instruction
         b. Both of these are trapped by the VMM
         c. The VMM can invalidate the entire virtual TLB (host page tables) and update the host page tables

Extended page tables

Wednesday, September 20, 2017 15:56

- Page table extensions to simplify the task of virtualizing guest virtual memory
- Host MMU holds 2 pointers, pointing at

Host page tables	Guest page tables
• The **only** contain the H mapping • Manipulated by the VMM	• They contain the G mapping • Manipulated **only** by the guest



- $H \circ G$ is perfomed in HW by the host MMU
  1. $V \xrightarrow{G\ (guest's\ MMU)} F$
     guest virtual → guest physical
  2. $F \xrightarrow{H\ (physical\ guest-to-host\ translation)} F'$
     guest physical → host physical
- No need for *VM exits*
- More difficult binary translation
  - During translation, each address is a guest physical one, and must be translated into a host phyisical one by using H
- Memory accesses needed to translate $V \rightarrow F'$
  1. Finding the host physical address of the guest page table containing the translation for V
     i. Guest page directory access (through the second MMU pointer) to get the guest physical address of the guest page directory entry PDE_F
     ii. Host page directory access (through the first MMU pointer) to get the host physical address of the host page table containing the translation of the guest page directory entry
     iii. Host page table access to get the host physical address of the guest page directory entry PDE_F'
  2. Finding the host final physical address
     i. Guest page table access (through PDE_F') to get the guest physical address F
     ii. Host page directory access (through the first MMU pointer) to get the host physical

address of the host page table containing the translation of the guest phyiscal address
    iii. Host page table access to get the host physical address F'
- 6 additional memory accesses
    - 20 accesses on 64-bit CPUs, but there are several TLBs there to help

HW passthrough

Thursday, September 21, 2017 14:31

What is it

- A VM can access directly a I/O peripheral
    - Target and host machine have the same peripheral
- The VMM is not involved
    - Hence, help from hardware is needed

VM/peripheral interactions

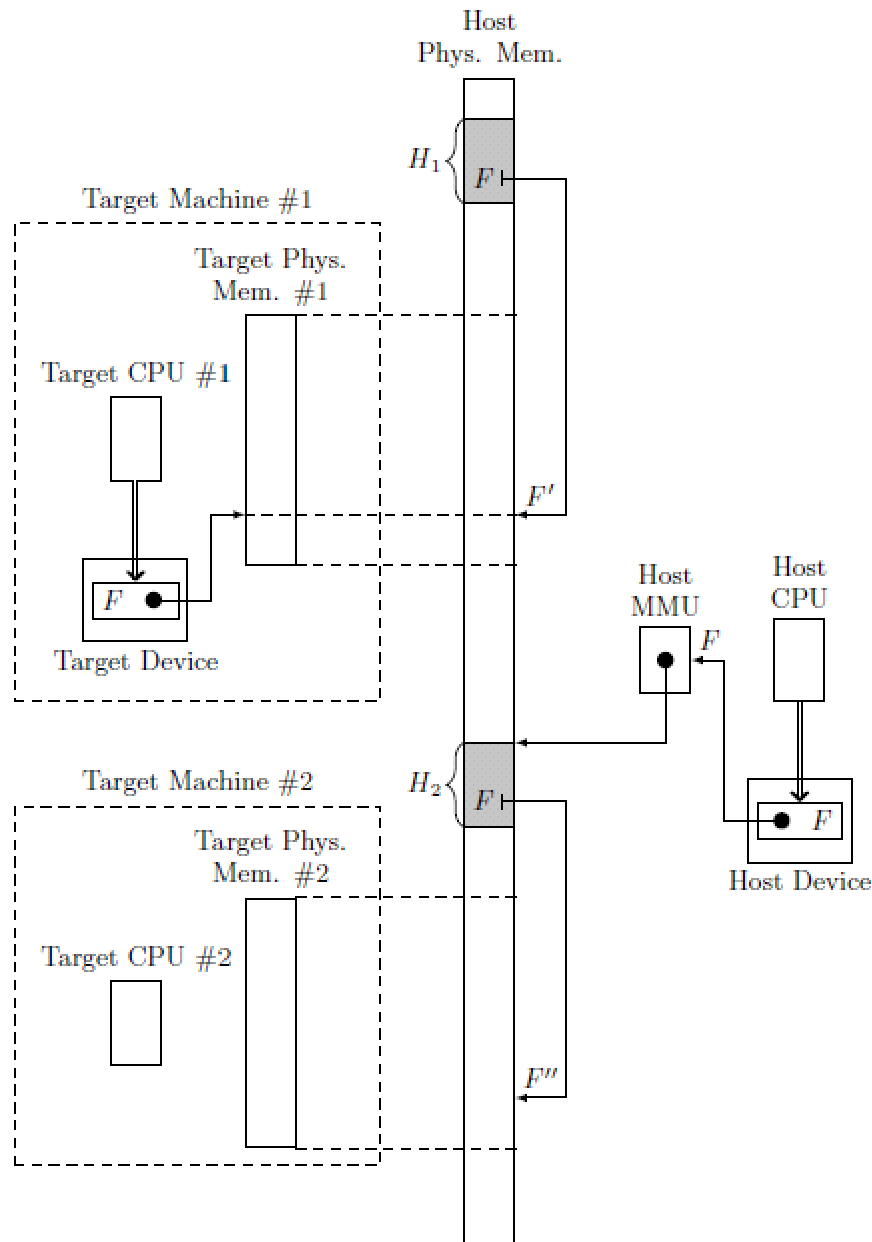
1. Read and writes to I/O registers

- The VMCS contains a pointer to a bitmap containing 1 bit for each one of the possible 65536 I/O registers
    - A bit indicates whether the VMM should cause a *VM exit* or not (letting the HW complete the operation without the VMM intervention, accessing real I/O registers) when it accesses that address
        - 0: *VM exit*
        - 1: passthrough
    - The VMM checks this bitmap for all `in` and `out` instructions
- Memory mapped I/O registers
    - The VMM must use the host MMU to map some guest physical addresses to the host physical register addresses
        - Usually they're the same
    - Registers in a page must belong to **one** device
        - Because mapping is per-page

2. DMA

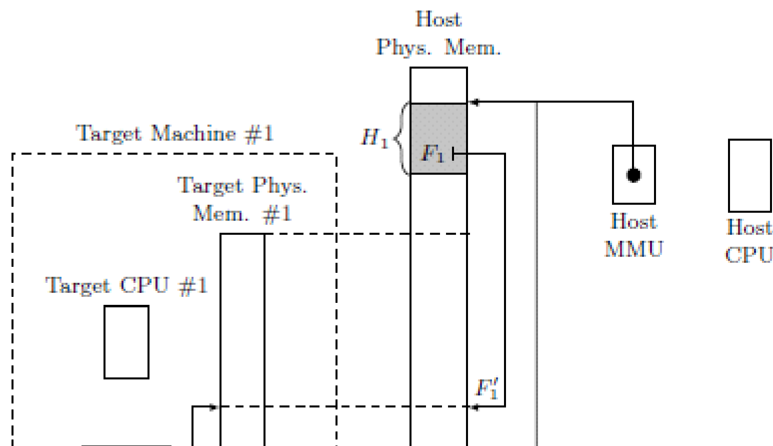
- Recall
    - A DMA-capable device is able to read/write on the system memory by itself
    - The device reads/writes from memory using the address contained in one of its registers
    - Steps
        1. This register is written by SW by the CPU
        2. Later, the device will read from memory using this register
- Problem: the guest CPU will write the **guest** physical address F into the host device, not the **host** physical address F'

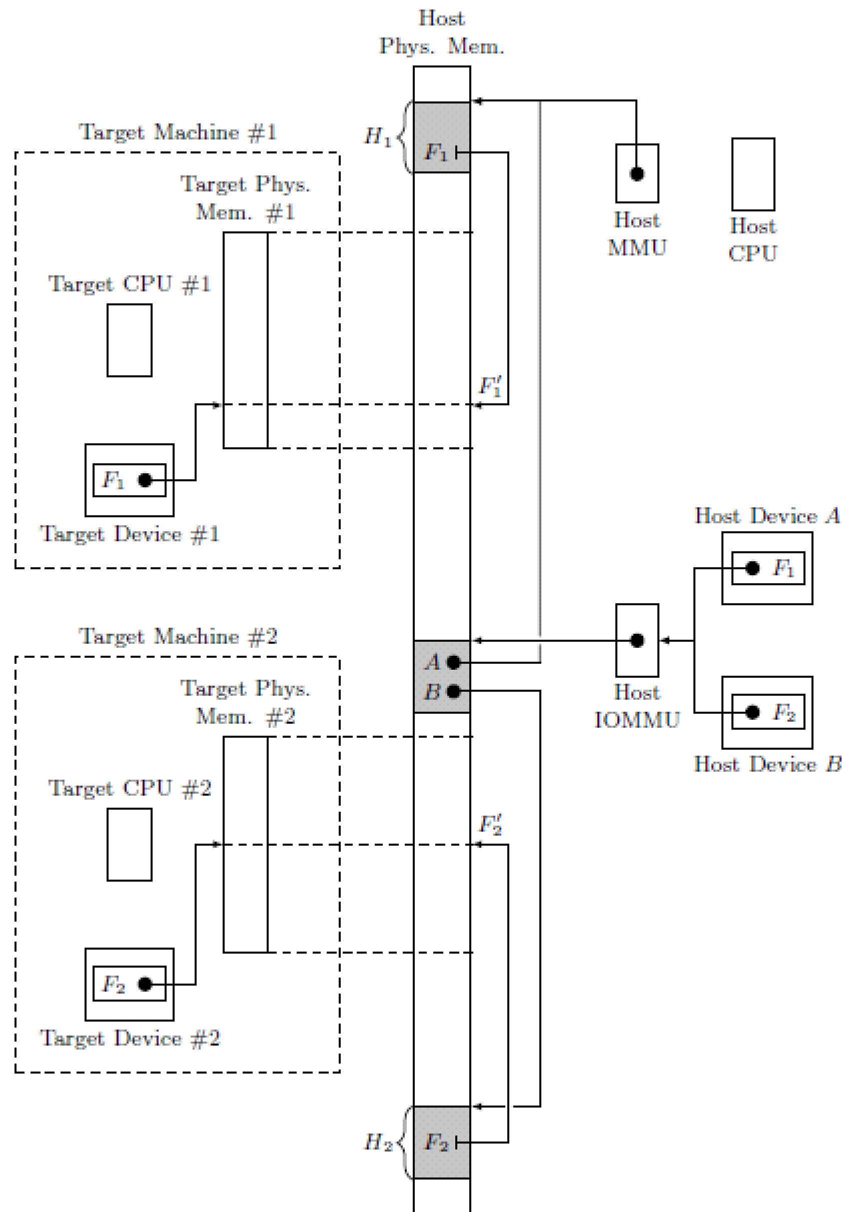
- *Guest* physical addresses **were** usually translated into *host* physical addresses by the host MMU, using the H mapping
  - If F won't be translated, the guest SW could maliciously access any part of the host memory
  - This H-translation cannot be done as it requires a *VM exit* for every read and write operation to device registers to let the VMM translate the address
- Wrong solution: let **all** addresses go through the host MMU, not only those coming from the CPU, but also the ones comming from I/O peripherals
  - Scheme



- It may happen that when the device is read to access the memory, the VMM has scheduled VM #2 and, therefore, the address will be translated by using the H_2 mapping (and not H_1)
  - This will let the VM #1 access the VM #2 physical memory at address F''

- Right solution: IOMMU (special additional MMU used just for I/O devices)
  - Scheme



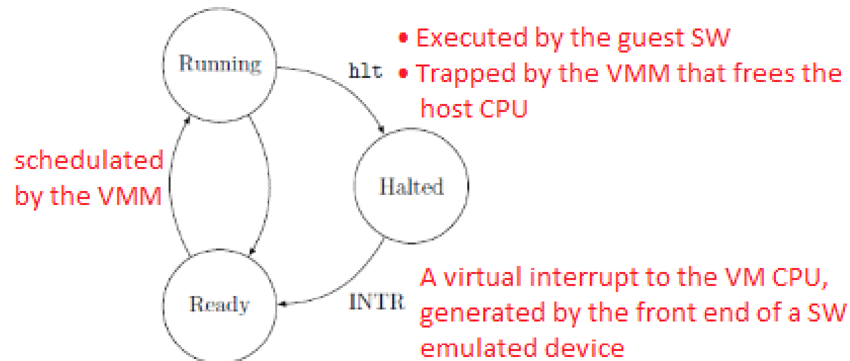


- *Host Device A* has been assigned to VM #1, *Host Device B* to VM #2
- To allow the IOMMU to choose the right translation, **host devices must identify themselves** in **each memory operation** they issue on the **bus**
  - Remember that all transaction happen on the **shared bus**
  - This is automatically done in PCI express systems because each transaction contain the $[bus, device, function]$ triple
- The IOMMU uses a dedicated translation data structure in the host physical memory
  - This maps **every host device** to the corresponding **owner's H map**
  - When a device is passed-through, the VMM must add an entry in the IOMMU datastructure
  - Devices belonging to no host belong to the host, so no translation will be done
- Intel and AMD's IOMMUs
  - They can issue page faults (by interrupting the host CPU) if they don't find the desired memory page in which the devices wants to write into
    - So there's no need for the guest physical memory to be always resident in host physical memory
  - Translations can be cached inside the IOMMU

- There are also TLBs inside the devices
- IOMMUs can be also used by ordinary multiprogrammed systems
  - To program DMA accesses b/w devices and swappable userspace buffers

3. Interrupts

- Until now, during non-root mode
  - All interruptions cause a *VM exit* (handled by the VMM)
  - None of them does (handled by the VM)
- Who handles what with passthrough
  - Interrupts **not** coming from passed-through devices: handled by the VMM
  - Interrupts coming from passed-through devices: handled by the VM
    - These interrupts may arrive while the interested VM is not running
    - A VM CPU has running states



      - Running: VM using the host CPU
      - Ready: host CPU currently used by another VM
      - Halted: guest SW halted itself
- Handling a passed-through (to the VM M) device's interrupt to the host
  - Basic functioning
    - If the M CPU is **running**
      - Interrupt handled by the CPU
      - Jumping to guest interrupt handler
      - No *VM exit*, no VMM involved
    - If the M CPU is **ready**
      - Interrupt request stored somewhere
      - The M VM will handle it later
    - If the M CPU is **halted**
      - Interrupt request stored somewhere
      - The M VM will handle it later
  - Implementation: **Posted interrupts**
    - Must be supported by
      - CPU
      - Interrupt controller
        - It has an ***Interrupt Remapping Table*** (**IRT**)
    - ***Posted Interrupt Descriptor*** (**PID**)
      - Stored in system memory
      - One for each VM
      - 64B
      - Only used by the CPU in non-root mode
      - Accessible from the VMCS
      - It contains:
        - ***Posted Interrupt Request*** (**PIR**): it stores pending interrupts, one bit for each of the 256 possible interrupt vectors
          - An interrupt vector is an address of an interrupt handler
          - The PIR just stores incoming **requests**, not handlers
        - The VM running status, econded with the ***Suppress Notification***

(**SN**) bit
- 0: the controller must deliver the interrupt (VM CPU running)
- 1: the controller must post the interrupt in the PID (VM CPU ready or halted)

◇ ***Notification Vector*** (**NV**): it contains the interrupt vector (address of the interrupt handler) of the incoming request

- Passed-through devices' interrupts handling steps
  1. The interrupt controller sets the proper bit in the PIR
  2. If SN = 0, it does nothing else
  3. If SN = 1, it interrupts the CPU using the interrupt vector in NV
- VMCS configuration
  - The `VM execution control` section must specify that external interrupts shuld cause *VM exits*
  - The `VM exit control` section must specify that, upon receiving an external interrupt, the CPU must obtain the interrup vectore before exiting
- How the CPU is able to distinguish normal interrupt from those coming from passed-through devices
  - Passed-through devices interrupt with a special interrupt vector, called the ***Active Notification Vector*** (**ANV**)
    - Otherwise there would be a *VM exit*
  - Steps
    1. The CPU receives an ANV
    2. Via a microprogram, the CPU looks at the PIR and handles all interrupt vectors that it finds set, w/o leaving non-root mode
- PID configuration
  (Interrupt I to be passed through to VM M with vector V)
  1. The VMM creates a PID
  2. The VMM writes the PID address in the VMCS of M
  3. The VMM fills the IRT's entry for I with
     - The vector V
     - The pointer to the PID
  4. The VMM prepares a handler for a vector **WNV** (***Wakeup Notification Vector***), different from ANV
  5. The VMM updates the PID as follows:
     - $M \xrightarrow{goes} Running$: SN = 0, NV = ANV
       - The VM M will received the passed-through interrupt
       - No intervention from the VMM
       - ? The VMM must look at the PIR
         - ? If there is any bit set, it must inhect the ANV when entering the VM
         - ? In this way the processor will look at the PIR and process the interrupts that were posted while the VM was not running
     - $M \xrightarrow{goes} Ready$: SN = 1
       - Interrupts will be posted in the PIR
       - No intervention from the VMM
     - $M \xrightarrow{goes} Halted$: SN = 0, NV = WNV
       - ? The VMM must be notified when the interrupt comes
       - ? The VMM needs to know that M is now eligible for execution and has to update its own data structure accordingly
       - ? The VMM must move the VM back into the list of the

Ready VMs

- ? – That's why NV is changed to WNV
- ? – Since WNV != ANV, interrupts delivered though the PID won't trigger the posted interrupt processing, but a *VM exit* to the VMM that can then process the event

? ◻ In order to setup the IRT, the VMM must know the vector V, but this vector is determined by the guest. The VMM may discover V by intercepting guest writes to the interrupt controller registers

Distributed file systems

Friday, September 22, 2017 18:15

- Centralized file server
  - Users may access their files from any client
  - Client file system management is simplified (OS installed and upgraded only once on the server)
- Cloud data centers
  - VMs are clients
  - File systems and VMs can be stored in different servers, so VM migration won't require large files transfers

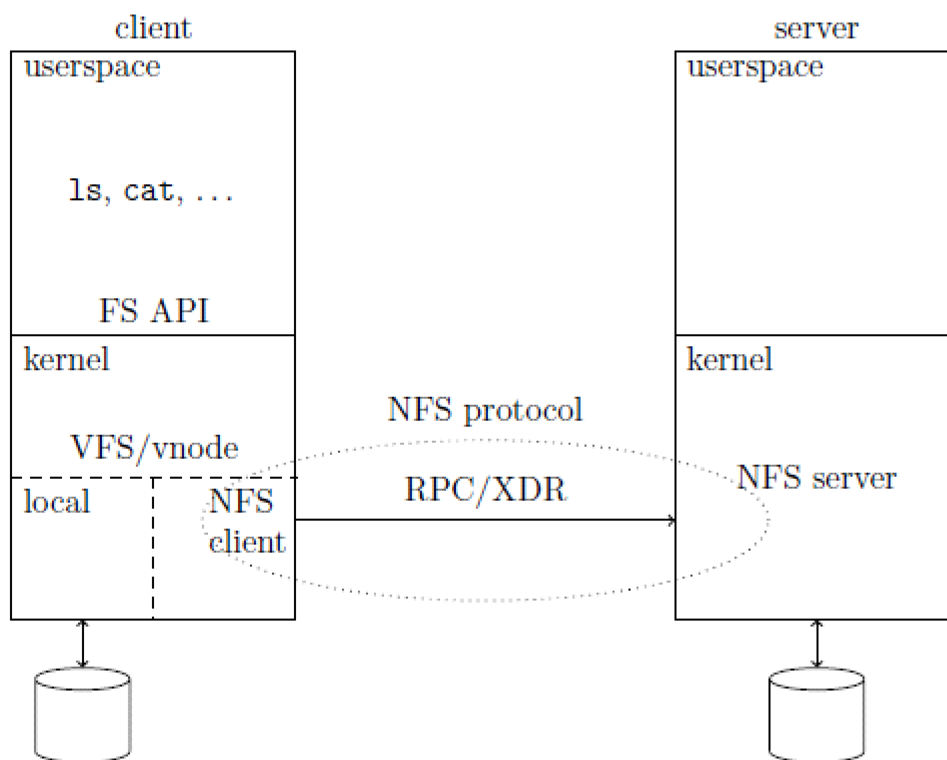
NFS

Friday, September 22, 2017 18:18

What is it

- *Network File System*
- It's a protocol b/w a client and a server for distributed file systems
- Developed by Sun Microsystems
- Platform-independent: exchanged mesages use the XDR format (eXternal Data Representation) to be architecture-indipendent (little-endiand, big-endian)
- OS-independent: users can mount a remote file system on a directory and then use remote files almost exactly like the local ones
    - No userspace program (like `ls`, `cat`, ...) needs to be modified in order to use remote files
    - Made possible by an abstract interface inside the kernel through all file operations are performed
        - One local access implementation
        - One remote access implementation with RPCs

Architecture



- The client issues Remote Procedure Calls (RPCs)
- NFS defines a set of RPCs that the server must implement
    - With parameters and replies
- Synchronous (blocking) RPCs
- The NFS servers runs in a kernel thread
- Crash recovery techniques
    - Server is stateless
        - It doesn't store any information about clients
        - Crashed server won't have to recover any state after a reboot
    - RCPs are idempotent
        - The effect of issuing the saonceme RPC several times in a row is the same as issuing it only

- Requests can be re-sent after a timeout: duplicate requests cause no problems

- Unix filesystem's APIs are stateful
  - `open()`
    - It returns a file descriptor
    - A file descriptor is an index in small per-process table stored in the kernel
    - The stateless NFS server has no such table
    - The `open()` RPC returns a **file handle**
      - It contains all the information needed to find the file
      - It must be sent by the client at each request
      - Structure
        - File system ID (identifies the file system on the server)
        - `inode` number (identifies the file in the file system)
        - Generation number (because `inode` numbers are reused, and it's incremented when it happens)
  - `read()`
    - The kernel moves the read pointer, so it's not idempotent
    - The read pointer must be remembered by the client and passed at each read request
  - `write()`
    - Same problem with the write pointer, which will also be mantained by the client and passed at each write request
  - `unlink()`
    - It removes the file name from the file system, but it doesn't delete the file until
      - The `close()` RPC is called
      - It's no longer open in any process
        - Temporary files cleanup made by the kernel, even after a crash
        - Not possible on a stateless server which doesn't know which files are opened by clients
    - The `unlink()` RPC only renames the file, and it'll be deleted just with the `delete()` RPC
      - Deletion of open files is stil possible

Performance

- UDP/IP is used
  - Packets loss, duplications and reordering is not a problem since clients resend a request after a timeout
- Clients' buffer caches are used to improve performance
  - Read RPCs' results are cached locally (so future reads won't call a RPC)
    - Cached data is valid for a limited time, after which becomes *stale*
  - Local writes are first accumulated in the buffer: this reduces the number of write RPCs

Limitations

- Unix UID and GID belong to processes within a client, but they must be checked on the server
  - Solution: global namespace for UIDs and GIDs
- The root user on the client is mapped to user nobody on the server
  - Not an effective solution
- No authentication
- File handles can be forged

Paravirtualization

Friday, September 22, 2017 22:58

Introduction

- Typically, each VM only runs just one application
- Should the OS really be unmodified? Should it be the same OS that runs on real HW?
    - Most of the costs of virtualization come from the need to intercept and emulate actions perfomed by the guest OS kernel that assumes it has direct access to the HW
    - Can be the guest OS changes as long as the OS API remains the same?

The Xen hypervisor

- Hypervisor = VMM
- Kernels are ported to different machines (having a different architecture) just by modifying the short-assembly code related to the underlying machine
- At Xen they observed that a VM can be seen as just another (virtual) **architecture** to which a kernel may by ported
    - For example, in the "VM architecture", the MMU can be accesses by issuing calls to the hypervisor, instead of writing into registers
    - Kernel may be optimized for the virtual architecture (paravirtualization)
- Xen does no longer user paravirtualization for virtualizing the CPU
    - Instead, they use hardware extensions
    - Paravirtualization is still used in I/O
        - It avoids inefficient emulation of hardware I/O devices
        - ? New virtual-only devices are defines, and just drivers need to be defined for them
- Architecture
    - Very small kernel that is loaded first on the machine and **gains direct control to the hardware**
    - On top of it (lower privilege level) there are ***domains***
        - Domains are used to implements VMs
            - So a domain can contain an entire OS
            - Kernels running in each domain can be
                - Standard
                - Unmodified (HW-assisted virtualization)
                - Using Xen APIs to improve performance
        - Dom 0
            - Special domain
            - It has access to the Xen API to create and destroy other domains
            - It usualy contains Debian w/ Xen management tools
        - Other domains
            - Can be given direct access to some I/O devices
            - Can use fully virtualized devices
            - Can use paravirtual devices
                - Front-end
                - Back-end
                    - Runned in a domain which has direct access to HW devices (like Dom 0)
                    - It gives indirect access to several front-ends

Virtual memory paravirtualization example

- **Unmodified kernels** use a set of page tables that or not the ones actually used by the MMU
    - G is the translation made by the guest kernel

- maps guest-physical addresses into host-physical addresses. It's used to:
  - Create the illusion of contiguous memory in the guest kernel
  - Limit access from the guest kernel to the pages that have been assigned to it
    - Writes to host page tables are denied
    - Reads to host page tables could be allowed for performance...
      - The guest kernel can read A and D bits updated by the host MMU, so there won't be a *VM exit* to let the hypervisor (VMM) sync host page tables with guest page tables
      - ! Reads can't be allowed because otherwise the guest kernel would know that its memory it's not contiguous

- **Paravirtual kernels** (they know they're runned in a VM)
  - They know the distiction b/w guest and host physical addresses
  - They can read host page tables
  - Writes to host page tables are still denided
    1. A paravirtual kernel must call an hypervisor protected routine (hypercall) when it wants to update **its** page tables
    2. The hypervisor will then check if the paravirtual kernel can or not

Unikernels

Saturday, September 23, 2017 11:16

What are they

- Kernels directly linked to an applicatoin
- They're like a normal library that can be loaded and run directly on the HW machine
  - Standard function calls
  - They don't involve a privilege level change
- They usually provide a single process

Unikenels vs standard library

- Unikernels contain low level access code to the HW
- Standard libraries don't

Unikernel with VMs

- ? VMs can provide a well defined set of I/O interfaces.
  Maintaining a unikernel is simple becuase it won't have to provide drivers for the large variety of HW devices
- ? Hypervisors already provide the isolation needed to run different applications: no need to do the same thing in VMs

Unikernels vs full OS

- Memory needed from VMs is much less
- Lighter: faster bootstrap, improved performance
- Unikernels run just one application: no need for protection

Containers

Saturday, September 23, 2017 11:29

What are they

- OS feature
- A way to
  - Isolate a set of processes
  - Make them think they're the only ones running on the machine
- Containers generally have less resources
  - Less memory, less disk space, less CPUs, less network bandwith, ...

Containers vs VMs

- Containers are **not** VMs
  - Processes running inside a container are normal processes running on the host kernel
  - There's no guest kernel running inside a container
  - There can't be any OS in a container, since the kernel is shared with the host
- Performance: **no penalty** in running an application inside a container compared to running it on the host
- Security: VMs have a smaller *attack surface* than containers

Kernel features used by containers

1. Namespaces

- Used to segregate system resources so they can be hidden from selected processes
- Unix `chroot()` system call has a similar purpose:
  - The kernel remembers, for each process
    - The `inode`
    - The *root directory*
  - Initially, every process' *root directory* is the filesystem root directory
  - Only `root` can call `chroot()`
  - By calling this syscall, we can make a subset of the filesystem look like it was the full filesystem for a set of processes (namespace)
  - Chroot environments are not full containers
    - System entities such as network ports, process IDs, ... are still not isolated
      - Namespaces also isolate all these other identifiers
    - Processes within a namespace can still send signals to other processes outside of it
- System entities are grouped in ***namespaces***
  - E.g.: two equals port numbers in different namespaces create no ambiguity
    - Same for processes and users
  - Normally processes share the same namespace
  - A process can start a new namespace with the `clone()` syscall and its ascendants will inherit it
    - It generalize the `fork()` and `pthread_create()` calls
    - Both processes and thread share something with the creator and have something private

2. Control groups

- Namespaces isolate all system entities but do not limit resources (so processes can indirectly interfere b/w them)
- Control groups define a group of processes that will have **limited resources**
  - A process cannot escape a control group
- A group of limited resources is called a **subsystem**
  - Control groups can then be **linked** so subsystems