

# VT-X

Dr.-Ing. Volkmar Sieh

Department Informatik 4  
Verteilte Systeme und Betriebssysteme  
Friedrich-Alexander-Universität Erlangen-Nürnberg

WS 2016/2017



- Nachbilden der CPU in Software
- Register als Variablen
- Befehl holen, und Semantik nachprogrammieren
- Performance: 1 Opcode des Gasts benötigt viele Opcodes des Hosts (ganze Funktion(en) um Opcode zu emulieren)
- Zusätzlicher Overhead durch Adressumrechnung (Segmentierung, Paging) bei Operanden im Speicher (und Befehlsadressen)
- Zugriffe auf Hardware durch Simulation der Hardware emulieren

⇒ **Langsam**



Möglichkeiten zur Optimierung für reine Emulation?



- Mehrere Opcodes auf einmal Holen („Prefetch Queue“), physikalische Adresse somit nur einmal berechnen
- Speichern von Adressbereichen und zugeordneter Hardware
- gesonderte Simulation des Speichers, um Buszugriffsfunktionen komplett zu vermeiden
- nicht benötigte Teile der Opcodes nicht simulieren, sofern die Ergebnisse nicht verwendet werden (Write-After-Write-Abhängigkeiten), zum Beispiel:
  - Flags-Berechnung, falls Flags danach überschrieben werden
  - Registerzustand Zwischenspeichern, falls keine Exception möglich
  - Inkrementieren des Program Counters nach jedem Befehl
  - Rückschreiben des Ergebnisses, falls danach Überschrieben *und* Ergebnis nicht in MMIO-Bereichen

Aber: Write-After-Write meist nicht einfach festzustellen!

...



## Idee

Befehlssequenzen werden oft mehr als nur einmal durchlaufen

Opcodes können in statischen und dynamischen Anteil aufgeteilt werden:

- statischer Anteil:
  - *MOD/RM* auswerten
  - somit Operanden bestimmen
  - Auswählen, welche Flags berechnet werden sollen
  - Operation dekodieren
- dynamische Anteil:
  - Speicheroperanden laden
  - Operation ausführen
  - Speicheroperanden zurückschreiben
  - Ausnahmen erzeugen
  - ...



- Opcodesequenzen zur Laufzeit übersetzen (entspricht Berechnung des statischen Anteils)
- übersetzten Block ausführen (dynamischer Anteil)

⇒ Siehe Vorlesung, Kapitel JIT.



## Idee

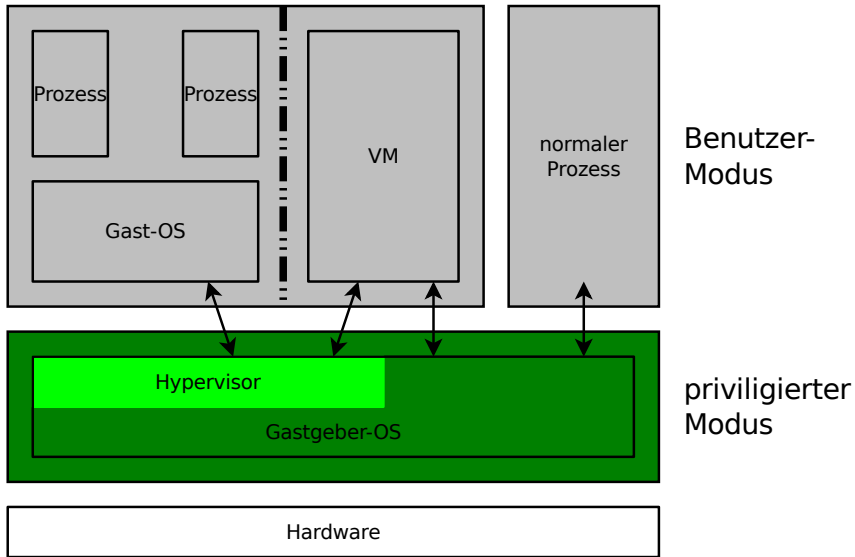
Unkritischer Gast-Code kann auf Host-CPU ausgeführt werden, nur kritische Instruktionen werden emuliert. Gast-System stellt somit einen Prozess des Host-Systems dar.

Ideale Instruktionssatzarchitektur:

- Gast-System wird „als Prozess“ im Benutzermodus ausgeführt
- zugeordneter Speicher entspricht Speicher des Gast-Systems
- kritische Instruktionen erzeugen Exception
- Hypervisor emuliert kritische Instruktionen
- kritische Instruktionen:
  - Zugriff auf Hardware `in, out`: Sind privilegiert
  - Zugriff auf memory-mapped Hardware `mov`: relevante Speicherbereiche nicht mappen
  - Wechsel des Betriebsmodus: privilegiert
  - ...



# Virtualisierung - Übersicht





Probleme der x86-ISA:

- Befehle, die keine Exception erzeugen, aber im User-Modus andere Semantik besitzen als im privilegierten Modus: `popf`
- Befehle, die keine Exception erzeugen, aber Daten der Hardware abfragen: `sgdt`, `sidt`



## Lösung

Neue Befehle, welche Probleme der ISA umgehen und zusätzliche Funktionalität zur Virtualisierung bereitstellen:

- Intel: VM-X (Vanderpool)  
<http://download.intel.com/design/processor/manuals/253669.pdf>
- AMD: AMD-V (Pacifica, „Secure Virtual Machine“)  
[http://www.amd.com/us-en/assets/content\\_type/white\\_papers\\_and\\_tech\\_docs/24593.pdf](http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf),  
Kapitel 15



## Grundidee

Ein neuer Befehl, welcher

- den Prozessorzustand sichert
- den Betriebsmodus wechselt:
  - spezielle Exception, welche ein `exit` bewirkt und den Prozessorzustand wiederherstellt
  - Interrupts, die diese Exception erzeugen (unabhängig vom Interrupt-Enable Flag!)
  - kritische Opcodes, die diese Exception erzeugen
- und Schattenregister zum Lesen von Kontrollregistern bereitstellt, z.B. `cr0`, `cr3`, ...

- Der spezielle Aufruf wird als *VMENTER* bezeichnet,
- die Rückkehr als *VMEXIT*.



Neue Opcodes für den Gastgeber:

- VMXON: Aktiviert die VT-X Erweiterungen
- VMXOFF: Deaktiviert die VT-X Erweiterungen
- VMLAUNCH: Startet eine Gast-Instanz und führt dabei ein *VMENTER* durch.
- VMRESUME: Setzt eine Gast-Instanz fort und führt dabei ein *VMENTER* durch
- und ein paar Verwaltungsopcodes, z.B. *VMPTRLOAD*, *VMREAD*, usw.

Neuer Opcode für den Gast:

- VMCALL: Erzwingt ein VMEXIT mit Möglichkeiten zur Parameterübergabe



Aufgeteilt in:

- Zustand des Gasts
- Zustand des Hosts
- Einstellungen für Ausführung des Gasts
- Kontrollfelder für *VMEXIT*
- Kontrollfelder für *VMENTER*
- Informationsfelder von *VMEXIT* (Grund des *VMEXIT*)

Setzen des aktiven *VMCS* mittels *VMPTRLD*.



- cr0, cr3, cr4
- esp, eip, eflags
- Segmentregister **mit** nicht-sichtbaren Anteilen (Limit, Typ, Rechte)
- VMX-Preemption Timer (*VMEXIT* wird erzwungen, falls Timer abgelaufen)
- ...



- cr0, cr3, cr4
- esp, eip
- Segmentregister **ohne** nicht-sichtbare Anteile
- ...



- sollen externe Interrupts ein *VMEXIT* erzwingen? (falls ja → Interrupt-Enable Flag unterbindet keine Interrupts)
- Wird VMX Preemption Timer verwendet?
- Welche Befehle sollen ein *VMEXIT* erzwingen?
- Welche Interrupts sollen ein *VMEXIT* erzwingen?
- Welche Ports erzwingen bei `in`, `out`-Befehlen ein *VMEXIT*?
- TSC-Offset
- ...





- *VMEXIT*:
  - Interrupt-Acknowledge-Zyklus während *VMEXIT* durchführen und Vektor in *VMCS* speichern?
  - VMX-Preemption Timer sichern?
  - ...
- *VMENTER*:
  - Soll ein Interrupt/Exception/Trap injiziert werden?
  - Vektornummer des Interrupts
  - Interruptquelle
  - Error Code
  - ...



```
VMXON;
VMLDPTR vmcs;
initialize_vmcs();
...
while (can_run_guest()) {
    save_host_regs();
    load_guest_regs();
    if (guest_started_already()) {
        VMRESUME;
    } else {
        VMLAUNCH;
    }
    save_guest_regs();
    load_host_regs();
}
```



```
switch (exit_reason) {
case REASON_CRITICAL_OPCODE:
    emulate_critical_opcode();
    /* length of opcode is stored in
     * exit reason fields */
    advance_eip();
    break;
case REASON_INTERRUPT:
    break;
case REASON_HLT:
    sched_yield();
    break;
/* ... */
}
}
VMXOFF;
```



- Probleme der x86-Instruktionsatzarchitektur behoben
- Beschleunigung durch Schattenregister (Lesen erzwingt kein *VMEXIT*)
- Hilfe zur Emulation kritischer Sequenzen:
  - Interrupt-Injektion
  - Länge des zu emulierenden Opcodes bekannt
  - Teilweise Zustandssicherung/-wiederherstellung durch Prozessor
  - Grund des *VMEXIT* bekannt
- Sehr fein granulare Kontrolle durch *VMCS*
- Paravirtualisierung auch ein Ziel:
  - Interruptbehandlung durch vertrauenswürdigen Gast
  - vertrauenswürdiger Gast kann Hardware kontrollieren
  - vertrauenswürdiger Gast kann Prozessorzustand verändern



Die ersten Versionen von VT-X erlaubten lediglich die Ausführung, wenn sich sowohl das Gastsystem als auch das Gastgebersystem im *Protected Mode* befanden und dabei Paging aktiviert war. Warum?



## Problem

Das Gastsystem benutzt Paging, und definiert sich eigene Page-Tabellen. Das Hostsystem benutzt ebenfalls Paging. Wie kann das Gastsystem

1. daran gehindert werden, etwas anderes als seinen simulierten Speicher zu sehen?
2. seinen Speicher an den richtigen Adressen sehen?



## Lösung

Für jeden Eintrag in der Seitentabelle des Gasts definiert sich der Host einen Eintrag in einer sogenannten **Shadow Page Table**, welcher die virtuellen Adressen des Gasts auf den Speicherbereich des simulierten Arbeitsspeichers abbildet.



# Beispiel Shadow Page Table

9	15
8	3
7	27
6	8
5	-
4	-
3	-
2	-
1	-
0	-

ursprüngliche  
Seitentabelle  
Host,  
Speicher für Gast

3	4
-	3
2	2
1	1
0	0

Seitentabelle  
eines Gast-Prozesses





# Beispiel Shadow Page Table

9	15
8	3
7	27
6	8
5	-
4	-
3	-
2	-
1	-
0	-

ursprüngliche  
Seitentabelle  
Host,  
Speicher für Gast

3	4
-	3
2	2
1	1
0	0

Seitentabelle  
eines Gast-Prozesses

9	15
8	3
7	27
6	8
5	-
4	15
3	-
2	3
1	27
0	8

Seitentabelle  
Host mit  
Einträgen für  
Schattenseitentabelle



Zwei gebräuchliche Verfahren hierzu:

1. Erzwingen eines *VMEXIT* bei

- `invlpg`
- Schreiben von `cr3`

Emulation des Opcodes aktualisiert dann Shadow-Page-Table.

2. Bereich der Seitentabelle des Gasts wird nur zum Lesen gemappt, *VMEXIT* also bei:

- Schreibzugriffen des Gasts auf die Seitentabelle
- Schreiben von `cr3`

Page-Fault-Handler aktualisiert ggf. Shadow Page Table, emuliertes Schreiben von `cr3` ebenfalls



## Probleme Shadow Page Table

1. Seiten des Hypervisor sichtbar für Gast?
2. Kollisionen mit Seiten für Hypervisor?



1. Seiten des Hypervisor sichtbar für Gast?
2. Kollisionen mit Seiten für Hypervisor?

### Lösung

- Seiten des Hypervisor sind System-Seiten!
- Gast läuft nur im User-Modus, kritische Befehle müssen sowieso emuliert werden!
- Kollisionen verringern: Hypervisor in unbenutzte Address-Bereiche „verstecken“
- restliche Kollisionen emulieren

Besser: ...



## Hardwareunterstützung für Paging des Gasts wünschenswert

- Intel: Extended Page Tables
- AMD: Nested Page Tables  
<http://developer.amd.com/assets/NPT-WP-1%201-final-TM.pdf>

- Im Prinzip gleiche Funktionalität bei AMD und Intel
- Physikalische Adressen in der virtuellen Instanz werden nicht direkt verwendet, sondern mittels der Seitentabellen des Hosts zu tatsächlichen physikalischen Adressen übersetzt
- Achtung: Auch Adressen der Page-Tabellen des Gasts werden übersetzt
- Problem: Ein Befehl muss emuliert werden, welcher ein Datum an Adresse 0x12345678 schreibt. Weder *Extended Page Tables* noch *Nested Page Tables* liefern hierbei die lineare Adresse des Hosts!



## Hands on VT-X

- VT-X Verwendung sehr viel Feinarbeit
- nicht gerade trivial
- Testen erweist sich als schwierig
- da Ring 0 benötigt wird, also
- entweder als Kernel-Modul (OOPS!)
- oder auf nackter Hardware

**Zum Glück hat das schon wer getan!**

## Nächste Woche

KVM (Kernel-Modul) und libkvm

