Virtualisierungstheorie

Vortrag im Hauptseminar Konzepte und Techniken virtueller Maschinen und Emulatoren

Johannes Schlumberger
asso@0xbadc0ffee.de

Friedrich-Alexander-Universität Erlangen/Nürnberg

20. November 2007

Inhalt

Motivation

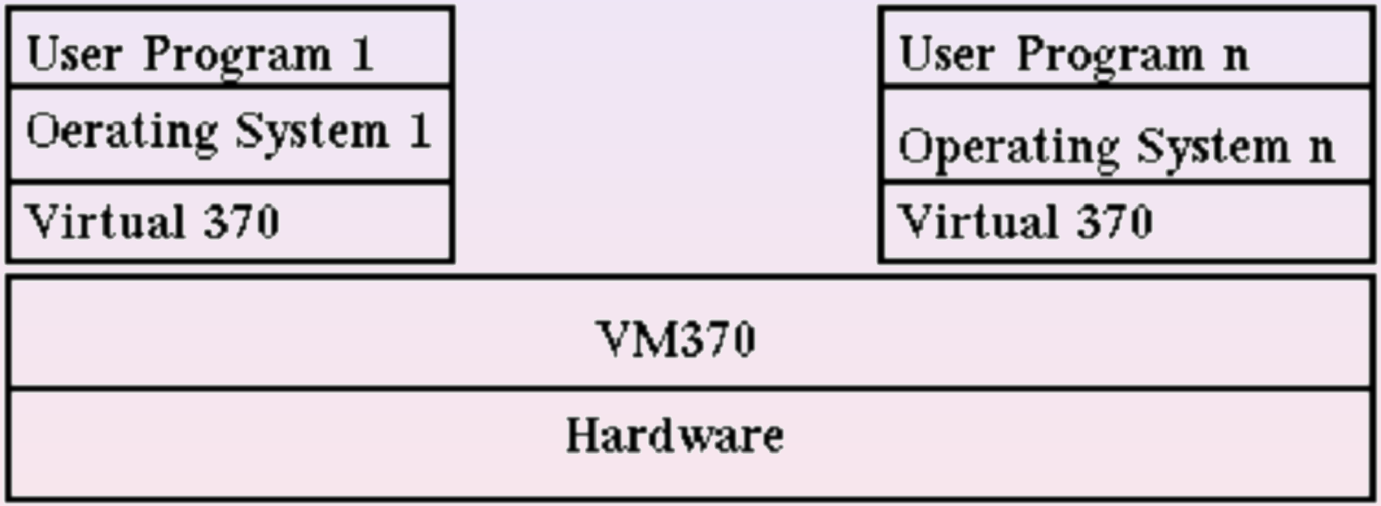
- stärkere Isolation von Prozessen (Security, Privacy)
- Sandboxen
- Anwendungen für ein spezifisches OS sollen auf einem anderen OS laufen
- Benutzer bekommen jeweils “eigene” Maschinen
- schnellerer Bootvorgang, Hardwareinitialisierung entfällt
- Simulation anderer Hardware (mehr Prozessoren, mehr Speicher, . . .)
- Testzwecke (Kernelpatches, etc.)
- usw.

Definitionen

Hardwarevirtualisierung

- mehrere Ausführungsumgebungen
- zueinander identisch
- voneinander unabhängig
- mit jeweils einem Betriebssystem
- eine einzige reale Maschine

IBM/VM-370



VIRTUAL MACHINE ARCHITECTURE (VM370)

Formen der Hardwarevirtualisierung

Software die Hardwarevirtualisierung implementiert nennt man Virtual machine monitor (VMM) oder Hypervisor.
Hauptsächlich wird auf 4 Arten virtualisiert

1. Emulation (unmodifiziertes OS, fremde CPU)
2. Paravirtualisierung (VMM bietet API (virtuelle Geräte), modifiziertes OS)
3. Vollvirtualisierung (unmodifiziertes OS, eigene CPU)
4. native Virtualisierung (Vollvirtualisierung mit Hardwareunterstützung (Vanderpool, etc.))

Anwendungsvirtualisierung

- Problem: Man möchte Anwendungen nicht für jede Hardwareplattform neu portieren müssen.
- Lösung: Eine virtuelle Maschine mit einheitlichem Befehlssatz wird für jede Plattform übersetzt, in dieser wird dann das eigentlich Programm ausgeführt.
- Jede für die VM geschrieben Software kann auf jeder von der VM unterstützten Plattform unverändert laufen
- bekanntestes Beispiel: Java Virtual Machine von Sun Microsystems

Virtuelle Umgebungen

- Ermöglicht verschiedene Userlandumgebungen auf demselben Kernel
- bekannt als Vserver
- Virtualisierung auf OS-Ebene
- bekannte Implementierungen:
  - FreeBSD Jails
  - Linux-VServer
  - Solaris Containers
  - chroot jails

Maschinenaggregation

- Problem: Man möchte große Cluster gerne wie einen einzigen Rechner bedienen
- Umgebung die mehrere heterogene, reale Maschinen umfasst
- bekannte Implementierungen:
  - Parallel Virtual Machine (PVM)
  - Message Passing Interface (MPI)
  - Plan9 (nicht mehr aktiv)

Techniken

Native Ausführung

- Vollvirtualisierung
- Anweisungen werden direkt auf der echten Hardware ausgeführt
- implementiert mit Hypervisor, der
  - Typ1: direkt auf der Hardware läuft
  - Typ2: in einem Betriebssystem läuft
- Multitasking und Ressourcenmanagement kann durch VMM oder in den Gastbetriebssystemen implementiert werden

Emulation

- Virtualisierungssoftware übersetzt der Hostarchitektur fremde Anweisungen in native Anweisungen (JIT, Interpreter)
- nur der Emulator muss an neue Architekturen angepasst werden

Virtualisierung auf Betriebssystemebene

- Implementiert auf OS-Ebene
- wie Partitionierung: ein Server wird in Teile gespalten (zones, virtual environments, virtual private servers)
- jeder Teil sieht für den Benutzer aus wie der komplette, reale Server
- geringer Aufwand, da gemeinsamer Kernel für alle VMs
- andererseits aber eben auch *nur* ein Kernel

Virtualisierung formal

Geht das wirklich?

Für fundierten Einsatz in Wirtschaft oder Militär?
Frühe Untersuchungen am Klassiker (der Vollvirtualisierung) durch Popek und Goldberg.
Ziele:

- Anforderungen an Architekturen
- Anforderungen an Software
- Beweisbarkeit der Anforderungen

Rechnermodell

- ausgehend von Rechnern der dritten Generation (DEC PDP-10, IBM360)
- Prozessor mit privilegiertem und usermode
- Teil der Instruktionen nur im privilegierten Modus verfügbar
- linearer, einheitlich adressierbarer Speicher
- Speicherzugriffe beschränkt und relativ durch relocation-bounds-register
- IO/Interrupts werden nicht modelliert

Aussagekräftiges Modell auch für heutige Rechner

Rechnerzustand

Realer Rechner ist immer in einem Zustand $S = (E, M, P, R)$

Speicher E

- Arbeitsspeicher mit Grösse q in bytes oder Maschinenworten
- $E[i]$ Inhalt i-ter Speicherstelle in E
- $E = E' \Leftrightarrow E[i] = E'[i] : 0 \leq i < q$

Prozessormodi M

- s Superusermode
- u Usermode

Program counter P

- Index in E
- relativ zu R

Speicher

Begrenzungsregister R

- $R = (l, b)$, relocation-bounds-register
- l ist absolute Speicheradresse, b gibt absolute Grösse des virtuellen Speichers
- freier Speicherzugriff $R = (0, q - 1)$

Adressumsetzung für Adresse a

$$a = \begin{cases} memorytrap & \text{falls } a + l \geq q \\ memorytrap & \text{falls } a \geq b \\ E[a + l] & \text{sonst.} \end{cases}$$

Instruktionen und Programmstatuswort

Instruktion

Komponenten von S endlich $\Rightarrow$ endliche Zustandsmenge C

$i : C \rightarrow C$

$i(S1) = S2$ oder

$i((E_1, M_1, P_1, R_1)) = (E_2, M_2, P_2, R_2)$

Programmstatuswort

Tripel (M, P, R)

in einer Speicherzelle speicherbar und wieder ladbar

Traps

i trapt genau dann wenn

$i((E_1, M_1, P_1, R_1)) = (E_2, M_2, P_2, R_2)$

- $E_2[j] = E_1[j] : 1 < j < q$ (Speicher beinahe unverändert)
- $E_2[0] = (M_1, P_1, R_1)$ (altes PSW in $E_2[0]$)
- $(M_2, P_2, R_2) = E_1[1]$ (neues PSW aus $E_1[1]$)

Üblicherweise $M_2 = s$ und $R_2 = (0, q - 1)$

ISA

Aufteilung in 3 Gruppen

Privilegierte Instruktionen

$\forall(S_1 = (e, s, p, r), S_2 = (e, u, p, r)) : i(S_2)$ trapt $\wedge\ i(S_1)$ trapt nicht

Sensitive Instruktionen:

Kontrollsensitive Instruktionen

$\exists S_1 = (e_1, m_1, p_1, r_1) \wedge i(S_1) = S_2 = (e_2, m_2, p_2, r_2) :$
$i(S_1)$ trapt nicht $\wedge (r_1 \neq r_2 \vee m_1 \neq m_2)$

Änderungen an Systemressourcen (Speicherregister oder Prozessormodus) ohne trap

Verhaltenssensitive Instruktionen

Verhalten abhängig von Ressourcenkonfiguration (r-b-register, etc.)

Zwei Operationen auf r-b-Register $r = (l, b)$

$\oplus$

$r' = r \oplus x = (l + x, b), x \in \mathbb{N}$(Basisverschiebung)

$|$

$E|r = E[l]$ bis $E[l + b]$ ($E|R$ die augenblicklich zugreifbare Speichersektion)

Zustände jetzt beschreibbar mit $S = (e|r, m, p, r)$ (eigentlich Äquivalenzklasse, da $E[i], i = 0, 1$ missachtet werden)

$| \circ \oplus$

$E|r \oplus x = E[l + x]$ bis $E[l + x + b]$
$E|r = E'|r \oplus x \Leftrightarrow E[l + i] = E'[l + x + i] : 0 \leq i < b$
(Programmverschiebungen im Arbeitsspeicher)

Verhaltenssensitive Instruktionen

Verhaltenssensitive Instruktionen II

$\exists x \in \mathbb{N}, S_1 = (e|r, m_1, p, r), S_2 = (e|r \oplus x, m_2, p, r \oplus x)$ mit $i(S_1) = (e_1|r, m_1, p_1, r), i(S_2) = (e_2|r \oplus x, m_2, p_2, r \oplus x), i(S_1)$ trapt nicht $\wedge\, i(S_2)$ trapt nicht; so, dass $e_1|r \neq e_2|r \oplus x \vee p_1 \neq p_2$

Ausführungsergebnis von i ist abhängig vom Inhalt des r-b-Registers oder vom Prozessormodus.
Beispiele:

- `LPI` Load physical adress (klappt je nach r-b-Inhalt)
- `MFPI` Move from previous instruction (effektive Adresse aus Modusabhängiger Information)

Drei Module des VMM

Allokator

- stellt Systemressourcen auf Anforderung zur Verfügung
- entscheidet über Ressourcenzuteilung
- implementiert Speicherschutz zwischen VM's und Host (r-b-Änderungen, etc)

Interpreter

- pro trapender Instruktion eine Interpreterroutine
- simuliert Effekt der getrapten Instruktion

Dispatcher

- entscheidet bei Trap zwischen Interpreter und Allokator
- ruft diese entsprechend auf

Drei Forderungen an den VMM

Äquivalenz

Programmergebnis darf durch Virtualisierung nicht verändert werden

Ressourcenkontrolle

VMM kontrolliert Zugriff auf alle Systemressourcen

Effizienz

statistisch überwiegender Teil der Instruktionen wird ohne VMM-Intervention ausgeführt

Wird die Effizienzforderung nicht erfüllt, sog. hybride VM

Satz von Popek/Goldberg

Virtualisierung ist möglich

Für jeden üblichen Rechner der dritten Generation kann genau dann ein VMM gebaut werden, wenn für diesen Rechner die Menge der sensitiven Instruktionen eine Untermenge der privilegierten Instruktionen ist.

Beweisskizze:

- VMM nach den drei Voraussetzungen kann wie skizziert gebaut werden
- Existenzbeweis für allgemeine Interpreterroutinen
- Effizienz nicht möglich wenn zu viele Instruktionen interpretiert werden

Bedeutung des Satzes

Was bedeutet "üblicher Rechner"?

Grundlegende Anforderungen an die Architektur

1. Zwei Prozessormodi
2. nichtprivilegierte Programme müssen privilegierte Subroutinen aufrufen können
3. Speicherschutz (Segmentierung, Paging)
4. asynchrone Interrupts für I/O-Systeme (nicht modelliert)

diese sind i. A. für moderne Rechner erfüllt

Beispiel Pentium

1. hat 4 Prozessormodi (Ringe)
2. “call gate”
3. Unterstützung für Paging und Segementierung
4. 16 vordefinierte Interrupts, 224 maskierbare

Aber: 17 von 250 Instruktionen sind sensitiv und nicht privilegiert.
`MOV, LAR, VERR, VERW, LSL, POP, PUSH, SMSW, PUSHF, POPF, ...`
$\Rightarrow$ trap-and-interpret verletzt
$\Rightarrow$ nach P/G nicht virtualisierbar.

Literatur

- Gerald J. Popek and Robert P. Goldberg (1974). “Formal Requirements for Virtualizable Third Generation Architectures”. Communications of the ACM 17 (7): 412 -421.
- Robert P. Goldberg (1973). “Architecture of Virtual Machines”.
- John Scott Robin, Cynthia E. Irvine (o. J.). “Analysis of the Intel Pentium’s Ability to Support a Secure Virtual Machine Monitor”.
- Jim, Jr. Smith, Ravi Nair, James E. Smith. “Virtual Machines: Versatile Platforms For Systems And Processes”. Morgan Kaufmann Publishers, Mai 2005