Seminarausarbeitung zur Virtualisierungstheorie

im Hauptseminar Konzepte und Techniken virtueller Maschinen und Emulatoren

Johannes Schlumberger
spjsschl@cip.informatik.uni-erlangen.de

Kurzzusammenfassung

Dieses Dokument soll einen Überblick über die theoretischen Grundlagen der Virtualisierung geben und die wichtigsten verschiedenen Arten und Techniken der Virtualisierung kurz vorstellen.

1 Einführung

Virtuelle Maschinen werden aus einer Vielzahl von Gründen eingesetzt; Sandboxing, Simulation anderer Hardware, Sicherheit durch einen höheren Grad an Isolation von Prozessen untereinander und Testen von Betriebssystemcode oder hardwarenahen Anwendungen sind nur einige der zahlreichen Anwendungsbereiche.

1.1 Formen virtueller Maschinen

1.1.1 Hardwarevirtualisierung

Hardwarevirtualisierung zeichnet sich durch mehrere zueinander identische, voneinander unabhängige Ausführungsumgebungen aus, die jeweils mit einem eigenen Betriebssystem versehen sind. Diese virtuellen Maschinen sind auf einer einzigen realen Maschine koexistent.

Erste Erfahrungen mit dem Konzept der Hardwarevirtualisierung wurden bereits 1972 von IBM mit der VM/370 gesammelt. Auf der nackten Hardware lief ein Minibetriebssystem, genannt VM370, das verschiedene virtuelle VM/370 Maschinen multiplexen konnte (Abbildung 1).

Vor allem zwei Vorteile der Virtualisierungstechnik sind sofort erkennbar; erstens wird durch den Ablauf eines Betriebssystems in einer Art Gefängnis (engl. jail) die Sicherheit erhöht. Gelingt es einem Angreifer aus dem Gastbetriebssystem auszubrechen, hat er noch immer keinen direkten Zugriff auf die nackte Hardware und auf Anwendungen, die in anderen Gastbetriebssystemen laufen. Zweitens ist es möglich, Betriebsmittel den verschiedenen Gastbetriebssystemen sehr flexibel zuzuteilen, was sich vor allem anbietet, wenn die einzelnen Systeme starken Schwankungen in ihrer Auslastung unterworfen sind.

Hardwarevirtualisierung wird üblicherweise auf eine von vier Hauptarten durch einen sogenannten Virtual machine monitor (VMM) oder Hypervisor implementiert:

1. Emulation
   Hierbei läuft ein unmodifiziertes Betriebssystem auf einer CPU, für die es eigentlich nicht gebaut wurde (Zum Beispiel Qemu).

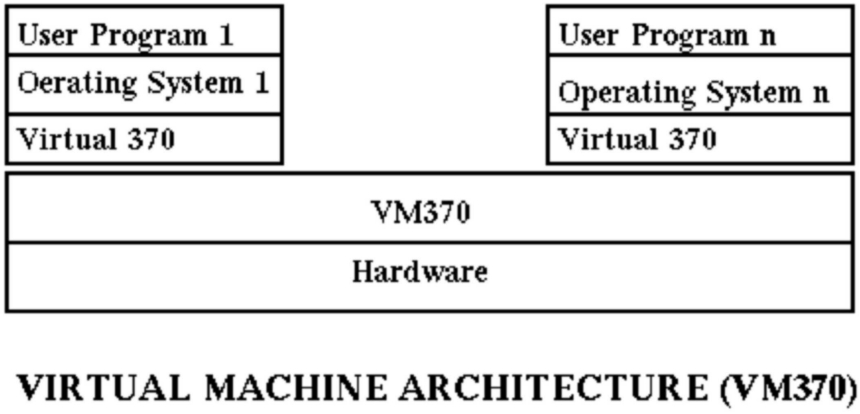

Abbildung 1: Konzept der IBM VM/370

2. Paravirtualisierung
   Hierbei bietet der VMM eine Programmierschnittstelle (API) für virtuelle Geräte an, die durch den VMM auf reale Geräte abgebildet werden; das Gastbetriebssystem wird passend zur API modifiziert. (Xen ist die bekannteste Implementierung)

3. Vollvirtualisierung
   Ein nicht modifiziertes Betriebssystem kommt auf einer CPU zum Einsatz für die es auch entworfen wurde.

4. native Virtualisierung
   Unter nativer Virtualisierung versteht man Vollvirtualisierung mit Hardwareunterstützung zum Beispiel durch Technologien wie Vanderpool oder Pacifica.

1.1.2 Anwendungsvirtualisierung

Manche Anwendungen möchte man nicht für jede Hardwareplattform neu übersetzen müssen; hier bietet sich unter Umständen eine virtuelle Maschine an, die eine abstrakte Architektur für diese Anwendungen virtualisiert. Die virtuelle Maschine bietet einen einheitlichen Befehlssatz auf jeder Plattform an, auf die sie portiert worden ist. Auf dieser Basis wird dann das eigentliche Programm ausgeführt. Somit kann jede für die virtuelle Maschine geschriebene Software unverändert auf jeder Architektur zum Einsatz kommen, für die eine Implementierung der virtuellen Maschine vorliegt. Das bekannteste Beispiel für diese sogenannte Anwendungsvirtualisierung ist sicherlich die Java Virtual Machine von Sun Microsystems.

1.1.3 Virtuelle Umgebungen

Eine weitere Form der Virtualisierung stellen die sogenannten virtuellen Umgebungen (engl. virtual environments) dar, die verschiedene Userlandumgebungen auf demselben Kernel ermöglichen. Im Internet werden Server, die auf diese Art virtualisiert sind, gegen meist monatliches Entgelt als Vserver (kurz für engl. virtual Server) angeboten. Bekannte Implementierungen sind die Free-BSD Jails, Linux-VServer, Solaris containers oder, etwas weniger speziell, die chroot jails.

1.1.4 Maschinenaggregation

Oft möchte man große Cluster ab einem gewissen Abstraktionsniveau gerne wie einen einzigen (sehr mächtigen) Rechner bedienen. Eine Umgebung, die mehrere, homogene oder heterogene reale Machinen umfasst und als Einheit virtualisiert, bietet eine Abstraktion, die als Maschinenaggregation bezeichnet wird. Als bekannte Implementierungen wären hier Parallel Virtual Machine (PVM), Message Passing Interface (MPI) und das nicht mehr aktive Plan9 aus den Bell-Laboratories zu nennen.

1.2 Techniken

Man unterscheidet 3 Haupttechniken die bei Virtualisierung zum Einsatz kommen.

1.2.1 native Ausführung

Die native Ausführung ist die Haupttechnik der Vollvirtualisierung. Es werden Instrukionen aus den virtuellen Maschinen direkt auf der echten Hardware ausgeführt. Umgesetzt wird dies mit einem Hypervisor der entweder direkt auf der realen Hardware (Typ 1) oder in einem Betriebssystem läuft (Typ 2). Bei der VM/370 beispielsweise handelt es sich um eine Implementierung mit Typ1-Hypervisor, das VM370-System ist selbst der komplette VMM. Multitasking und Ressourcenmanagement kann entweder durch diesen Hypervisor oder (inklusiv) in den Gastbetriebssystemen implementiert werden.

1.2.2 Emulation

Die Virtualisierungssoftware übersetzt der Hostarchitektur fremde Anweisungen in ihr bekannte Anweisungen, um eine Ausführung zu ermöglichen. Das kann zum Beispiel durch Interpreterroutinen oder Just-in-time-Compile-Mechanismen (JIT) geschehen. Soll eine neue Architektur unterstützt werden, muss nur der Emulator angepasst werden, was jedoch unter Umständen einen beträchtlichen Aufwand erforden kann.

1.2.3 Virtualisierung auf Betriebssystemebene

Der Name weist schon darauf hin, dass diese Form der Virtualisierung auf Betriebssystemebene stattfindet; insbesondere haben also die virtualisierten Maschinen alle denselben Betriebssystemkern. Wie bei der Partitionierung einer Festplatte wird ein Server hier in Teile gespalten, die für einen Benutzer dieses Teiles aussehen, wie der komplette, reale Server. Der Aufwand ist vergleichsweise gering, da nur ein Kernel für alle virtuellen Umgebungen zum Einsatz kommt; das ist jedoch gleichzeitig ein nicht zu unterschätzender Nachteil, da der Kern eben nicht an abweichende Anforderungen der einzelnen Umgebungen, die er bedienen soll, angepasst werden kann.

2 Virtualisierung nach Popek/Goldberg

Aus den frühen siebziger Jahren stammt ein Papier von J. Popek und Robert P. Goldberg, das Anforderungen an Architekturen und Software untersucht, die erfüllt werden müssen um Virtualisierung zu ermöglichen. Insbesondere wurde auf eine mathematische Notation Wert gelegt, die beweisbare Aussagen über Virtualisierungsverhalten und Virtualisierbarkeit von Architekturen ermöglichen soll. Im folgenden sollen die wichtigsten Ergebnisse und Techniken des Papiers hier kurz vorgestellt werden.

2.1 Rechnermodell

Ausgehend von Rechnern der dritten Generation (DEC PDP-10, IBM360) wird ein Modell für einen modernen Rechner entwickelt, der über einen Prozessor mit privilegiertem und Usermodus sowie einen linearen, einheitlich adressierbaren Speicher verfügt. Ein Teil seiner Instruktionen sind nur im privilegierten Modus verfügbar, rudimentärer Speicherschutz wird durch ein relocation-bounds-Register ermöglicht; Interrupts werden nicht mitmodelliert, auf ihre Wichtigkeit hinsichtlich der Performanz von E/A-Vorgängen wird jedoch wiederholt hingewiesen. Hierbei handelt es sich um ein (natürlich vereinfachtes) aber auch heute noch gültiges Modell für einen Allzweckrechner.

2.1.1 Rechnerzustände

Ein Realer Rechner befindet sich immer in einem Zustand S der durch ein Viertupel $S = (E, M, P, R)$ beschrieben werden kann. Die einzelnen Elemente des Tupels werden wie folgt definiert:

- E
  - bezeichnet den Arbeitsspeicher des Rechners; er besteht aus q vielen Bytes oder Maschinenworten. Man sagt: Der Arbeitsspeicher hat Grösse q.
  - $E[i]$ steht für den Inhalt der i-ten Speicherstelle in E
  - Zwei Speicher E, E' sind gleich, genau dann wenn $E[i] = E'[i] : 0 \leq i < q$
- M
  bezeichnet den Prozessormodus; Werte sind s für den privilegierten Modus, u sonst.
- P
  bezeichnet den Programmzähler; es ist dies ein Index in E, relativ zu R angegeben.
- R
  - bezeichet das relocation-bounds-Register $R = (l, b)$.
  - l ist hierbei eine absolute Speicheradresse, die den Beginn den virtuellen Speichers markiert, b gibt die Grösse des virtuellen Speichers an.
  - Insbesondere ist $R = (0, q - 1)$ genau dann wenn komplett freier Speicherzugriff gewährleistet wird.

Die Adressumsetzung für eine Adresse a läuft dann wie folgt ab:

```
address
get_address(address a)
{
        if (a+l >= q) memorytrap();
        if (a >= b) memorytrap();
        return E[a+l];
}
```

Das Tripel (M, P, R) wird als Programmstatuswort bezeichnet. Es wird davon ausgegangen, dass dieses in einer Speicherzelle darstellbar ist. Da alle Komponenten von S aus endlichen Mengen gewählt werden, ist auch $C = \{S : S \in E \times M \times P \times R\}$ eine endliche Menge.

2.1.2 Instruktionen

Eine Instruktion ist in diesem Modell eine Funktion i mit Urbildmenge C und Bildmenge $C : i(S_1) = S_2$.

Man sagt, dass eine Instruktion $i : i((E_1, M_1, P_1, R_1)) = (E_2, M_2, P_2, R_2)$ genau dann trapt, wenn

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

gilt. Üblicherweise gilt $M_2 = s$ und $R_2 = (0, q-1)$. Für eine trapende Instruktion wird also eine privilegierte Subroutine aufgerufen.

Die ISA wird im Modell in drei Gruppen aufgeteilt, nämlich die der privilegierten, der kontrollsensitiven und der verhaltenssensitiven Instruktionen. Die beiden letzteren werden auch allgemein als sensitive Instruktionen bezeichnet. Privilegierte Instruktionen zeichnen sich dadurch aus, dass sie im Usermode stets trapen, im privilegierten Modus jedoch nicht $(\forall(S_1 = (e, s, p, r), S_2 = (e, u, p, r)) : i(S_2)$ trapt $\wedge\, i(S_1)$ trapt nicht). Kontrollsensitive Instruktionen sind all jene, die Änderungen an der Systemressourcenkonfiguration, also insbesondere dem Speicherregister oder Prozessormodus vornehmen, ohne dabei *immer* zu trapen. $(\exists S_1 = (e_1, m_1, p_1, r_1) \wedge i(S_1) = S_2 = (e_2, m_2, p_2, r_2) : i(S_1)$ trapt nicht $\wedge\, (r_1 \neq r_2 \vee m_1 \neq m_2))$. Verhaltenssensitive Instruktionen schließlich sind all die Instruktionen, deren Verhalten von der aktuellen Ressourcenkonfiguration abhängig ist.

Um die kontrollsensitiven Instruktionen noch näher beschreiben zu können, definiert man zunächst zwei Operationen auf dem relocation-bounds-Register r. Die erste Operation ist $r' = r \oplus x = (l + x, b), x \in \mathbb{N}$; hierbei handelt es sich um eine einfache Basisverschiebung um x. Die andere Operation reduziert den Speicher E auf die augenblicklich sichtbare Speichersektion. Mathematisch ausgedrückt durch $E|r = E[l]$ bis $E[l + b]$. Mit dieser Operation ist es jetzt möglich Zustände als $S = (e|r, m, p, r)$ zu beschreiben. (Anmerkung: Eigentlich handelt es sich hier um eine Äquivalenzklasse von Zuständen, da $E[i], i = 0, 1$ missachtet werden).

Führt man die beiden eben eingeführten Operationen hintereinander aus, erhält man ein Werkzeug mit dessen Hilfe man die Programmverschiebungen im Arbeitsspeicher ausdrücken kann, wie sie beispielsweise von platzierenden Ladern vorgenommen werden. So sind $E|r \oplus x = E[l + x]$ bis $E[l + x + b]$ und $E|r = E'|r \oplus x \Leftrightarrow E[l + i] = E'[l + x + i] : 0 \leq i < b$.

Kontrollsensitive Operationen können jetzt genauer spezifiziert werden. Kontrollsensitive Instruktionen sind Instruktionen, deren Ergebnis vom Inhalt des relocation-bounds-Registers oder vom Prozessormodus abhänigig ist. Formal schreibt sich das: $\exists x \in \mathbb{N}, S_1 = (e|r, m_1, p, r), S_2 = (e|r \oplus x, m_2, p, r \oplus x)$ mit $i(S_1) = (e_1|r, m_1, p_1, r), i(S_2) = (e_2|r \oplus x, m_2, p_2, r \oplus x)$ so, dass $i(S_1)$ nicht trapt $\wedge\, i(S_2)$ nicht trapt, wobei $e_1|r \neq e_2|r \oplus x \vee p_1 \neq p_2$.

Ein Beispiel für eine solche Instruktion ist `LPI` (Load physical Address), was je nach den aktuell geltenden Speichergrenzen zu einem Trap oder zu vermutlich gewünschtem Ergebnis führt.

2.2 VMM

Ein Hypervisor besteht im Allgemeinen aus drei Teilen. Zuerst einem Allokator, der auf Anforderung Systemressourcen für die virtuellen Maschinen zur Verfügung stellt und über die Zuteilungen entscheidet. Er implementiert den Speicherschutz zwischen den virtuellen Maschinen und dem Host. Dann einem Interpreter, der für jede trapende Instrukion eine

Interpreterroutine bereithält, die die Instruktion simuliert. Sowie zuletzt einem Dispatcher der entscheidet ob Interpreter oder Allokator gebraucht werden und diese entsprechend aufruft.

An einen VMM stellen Popek und Goldberg drei Forderungen. Das Ergebnis eines Programmes muss dasselbe sein, egal ob es virtualisiert oder auf der realen Maschine läuft (Äquivalenz). Desweiteren muss der VMM alle Zugriffe auf alle Systemressourcen kontrollieren und validieren können (Ressourcenkontrolle). Zuletzt muss dafür Sorge getragen werden, dass ein statistisch überwiegender Teil der Instruktionen ohne VMM-Intervention ausgeführt wird (Effizienz). Wird die Effizienzforderung nicht erfüllt, spricht man von einer hybriden virtuellen Maschine.

3 Ergebnisse

3.1 Satz

Das Papier mündet in den Satz von Popek und Goldberg:
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.
Der Beweis ist langwierig und meiner persönlichen Meinung nach auch teilweise zu ungenau. Es wird skizziert, wie ein VMM nach den drei obigen Voraussetzungen gebaut werden kann und ein Existenzbeweis für allgemeine Interpreterroutinen geführt. Abschliessend wird diskutiert warum die Emulation von zu vielen Instruktionen zu großen Performanzverlusten führt.

3.2 Interpretation am Beispiel des Pentium

Mit einem "üblichen Rechner", ist im Satz ein Rechner gemeint, der vier Grundanforderungen erfüllt. Er hat einen Prozessor mit mindestens zwei Betriebsmodi, nichtprivilegierte Programme müssen privilegierte Subroutinen aufrufen können, es gibt Speicherschutz (beispielsweise durch Segementierung oder Paging), und asynchrone Interrupts stehen für E-A-Systeme zur Verfügung . Der Intel Pentium erfüllt alle diese Vorraussetzungen, er verfügt über 4 Prozessormodi (sogenannte Ringe), hat ein sogenanntes call gate, stellt Unterstützung für Paging und Segmentierung zur Verfügung und hält 16 vordefinierte sowie 224 weitere Interrupts bereit.

Auf dem Pentium sind jedoch 17 von rund 250 Instruktionen sensitiv nach Popek/Goldberg und nicht privilegiert. Damit ist eine effiziente Virtualisierung nach Popek/Goldberg nicht möglich. Es existieren jedoch eine ganze Reihe von hybriden Virtualisierungslösungen für den Pentium (VMWare,...).

4 Literaturverzeichnis

Literatur

[1] Gerald J. Popek and Robert P. Goldberg (1974). "Formal Requirements for Virtualizable Third Generation Architectures". Communications of the ACM 17 (7): 412 -421.

[2] Robert P. Goldberg (1973). "Architecture of Virtual Machines".

[3] John Scott Robin, Cynthia E. Irvine (o. J.). "Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor".

[4] Jim, Jr. Smith, Ravi Nair, James E. Smith. “Virtual Machines: Versatile Platforms For Systems And Processes”. Morgan Kaufmann Publishers, Mai 2005