# ハイパーバイザの作り方〜ちゃんと理解する仮想化技術〜 第1回 x86 アーキテクチャにおける仮想化の歴史と Intel VT-x

## はじめに

初めまして、浅田拓也（@syuu1228）です。本号より「ハイパーバイザの作り方」と題して、ハイパーバイザの内部の実装やその土台となるハードウェア側の仮想化支援技術の詳細について解説を行なっていきます。よろしくお付き合いお願い致します。

## x86 アーキテクチャにおける仮想化の歴史と仮想化手法

近年、x86 アーキテクチャのコンピュータの性能が劇的に向上したことにより、デスクトップ用途だけでなくサーバ用途にも積極的に用いられるようになりました。

さらに、サーバとしてもユースケースによってはハードウェア性能に余裕が出てきたことにより、ここに仮想化を導入して複数のサーバインスタンスを 1 つの物理サーバで実行することが現実的な選択肢になってきました。

しかしながら、x86 アーキテクチャは仮想化が困難なアーキテクチャとして知られていました。(“Formal Requirements for Virtualizable Third Generation Architectures”) という今から 40 年ほど前に発表された論文で、ハイパーバイザを実装するのに必要な命令セットアーキテクチャ上の要件として「Popek と Goldberg の仮想化要件」というものが定義されています。

内容を簡単に説明すると、システム資源の構成を変えようとする命令やシステム資源の構成に動作が依存している命令（「センシティブな命令」と呼ばれる）がユーザモードで実行されるときには、これがトラップされなければならない、ということを言っています（図 1)。

なぜかということをもう少し説明します。この論文では、ハイパーバイザの構成法として、ゲストマシン上のユーザランドプログラム・カーネルプログラムをユーザモードで動作させ、センシティブ命令をトラップしてハイパーバイザで適切なエミュレーション処理を行うことにより、ゲストマシンへ仮想的なコンピュータの状態を提供することを意図しています。

しかし、x86 アーキテクチャではユーザ権限で実行可能である（＝トラップされない）にもかかわらず、センシティブな命令というものが複数あります。このため、このアーキテクチャ上でハイパーバイザを実装するに

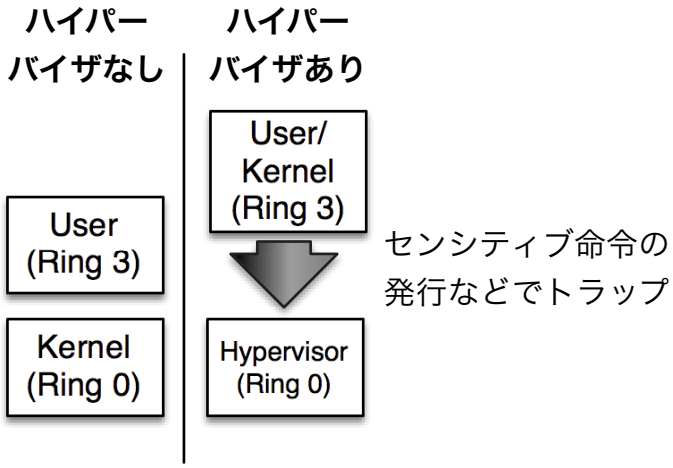


図 1　センシティブな命令

は効率性をある程度犠牲にしてセンシティブな命令の実行を回避し、適切な処理に置き換える必要がありました[*1]。

では、どのような方法が取られたのでしょうか。

## VMware での実装

VMware では、Binary Translation と呼ばれる手法が採られました。これは、ゲストマシンで実行したい命令のうち、実行して問題ないもの（ユーザモードの命令のうち、センシティブではない命令）は直接 CPU で実行し、実行されては問題がある命令は実行される前にハイパーバイザが動的に書き換えていく、というしくみを採っています。

これによって、OS をハイパーバイザ向けに変更することなく、ある程度高速に実行できるようになりました（オーバーヘッドはあるのですが、ゲストマシンで実行されるすべてての命令をエミュレーションする方式に比べるとかなり速くなりました）。

## Xen での実装

一方、Xen では準仮想化と呼ばれる手法が採られました。これは、ハイパーバイザ向けに書き換えた専用のゲスト OS を用意することにより、ゲスト実行の効率を上げようという考え方です。Xen 向けに書き換えられた

[*1] 効率を気にしなければ、ゲストマシン上で実行されるすべての命令をエミュレートすればどのような命令セットでも問題なくゲストマシンを実行できます。これは「エミュレータ」と呼ばれ、異なるアーキテクチャのコンピュータをゲストとして実行するのには必要ですが、同じアーキテクチャのコンピュータをゲストとして実行する場合には効率が非常に悪いです。x86 アーキテクチャのエミュレータとして QEMU が有名ですが、QEMU と VT-x を用いるハイパーバイザの LinuxKVM の実行速度を比較してみてください。かなり差があるはずです。

ゲスト OS ではハイパーバイザの介在が必要な処理で「ハイパーバイザコール」を発行し、処理をハイパーバイザへ依頼します。

前述の動的に書き換えるという手法に対して、この手法では事前に OS を書き換えているため、静的に書き換えている、とも言えます。これにより、OS を修正するというコストがかかる代わりに、仮想化環境でより実機に近い性能が得られるようになりました。

## 仮想化技術の普及

これらの技術を用いて、VMware 社は 2001 年にサーバ向けのハイパーバイザである VMware GSX/ESX を、ケンブリッジ大学の Computer Laboratory は 2003 年に Xen 1.0 をリリースしています。これらのプロダクトの出現により、PC サーバでの仮想化が普及し始めました。ここで、Intel はそもそも x86 を「仮想化可能」なアーキテクチャへ拡張しようという取り組みを開始しました。Intel Developers ForumFall 2003 で Vanderpool Technology（VT）として紹介され（その後 VT-x と改名し）、2005 年に最初の VT-x 対応 Pentium 4 が出荷されました[*2]。

これにより、前述のような工夫を行う場合に比べて、より単純なハイパーバイザ実装で仮想化が実現できるようになりました。しかもより実機に近い性能が得られるように発展しました[*3]。VT-x がリリースされたことにより、上述のソフトウェアによる仮想化サポートの手法を採っていた初期の VT-x では VT-x よりも仮想化方式のほうが性能が高いケースがあったようですが、そのあとの CPU の改良により VT-x のパフォーマンスや機能が向上したため、今では VT-x を使用することが多くなっています。

また、ごくローエンドの PC を除き、ラップトップ PC をも含めほぼすべてのレンジの PC に VT-x が載ることとなり、サーバマシンだけでなくデスクトップ用途でも VT-x の恩恵を受けることができるようになりました。

さらに、最初からハードウェア仮想化支援機能をハイパーバイザ実装に用いることを前提に開発された Linux KVM などの新しいハイパーバイザも登場してきました。

## Intel VT-x の概要

それではこの、PC の仮想化に必須の技術となっている VT-x の中身について見て行きましょう。VT-x では、既存のソフトウェアとの互換性を保ちつつ「仮想化可能」にするため、x86 アーキテクチャにすでに存在するプログラムの権限管理に用いられる Ring プロテクションのしくみとは別に、ハイパーバイザ向けのプロテクションモデルを導入しています。このために「ハイパーバイザのモード」と「ゲストマシンのモード」が追加されました。この「ハイパーバイザのモード」を VMX Root Mode、「ゲストマシンのモード」を VMX non Root Mode と呼びます（図 2）。

---

[*2] その後、AMD も同様の機能を AMD-V として出してきており、そのような技術の総称としてこれらは「ハードウェア仮想化支援機能」と呼ばれています。本記事では、解説がややこしくなることを避けるために、敢えて「Intel VT」に絞って解説を行いますが、基本的に AMD-V も同じようなしくみを提供しています。

[*3] とくに Xen の場合は、Xen サポートを実装した Windows がリリースされなかったため、VT － x に対応して未書き換えな OS を動作可能にさせる意義がありました。

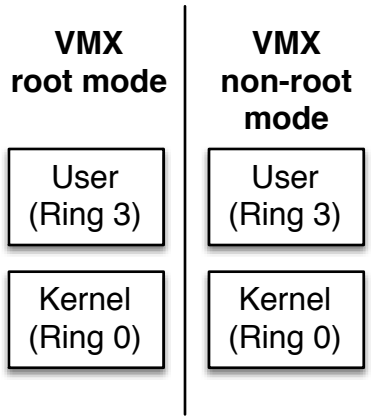


図 2　VMX non Root Mode

先ほど紹介した VMware や Xen のアプローチとは異なり、VT-x ではゲスト OS に VMX non Root Mode 上でセンシティブな命令をそのまま実行させます。実行された命令がセンシティブな命令だったときは、VMX non Root Mode でのプログラムの実行が中断されて VMX Root Mode へモードが遷移し、ハイパーバイザがゲストマシンで実行された命令に対して適切な処理を行う、というしくみになっています。

つまり、従来「Popek と Goldberg の仮想化要件」を満たさない命令（＝そのまま CPU で実行したときにトラップできないセンシティブ命令）が存在していた x86 アーキテクチャに上述の 2 モードを導入することにより、VMX non Root Mode 上で実行されたセンシティブな命令を VMX Root Mode でトラップ可能にして「Popek と Goldberg の仮想化要件」を満たすようアーキテクチャを拡張しています。

このとき、VMX Root Mode から VMX non RootMode へ切り替えることを VMEntry、VMX non RootMode が中断され VMX Root Mode へ戻ってくることを VMExit と呼びます（図 3）。

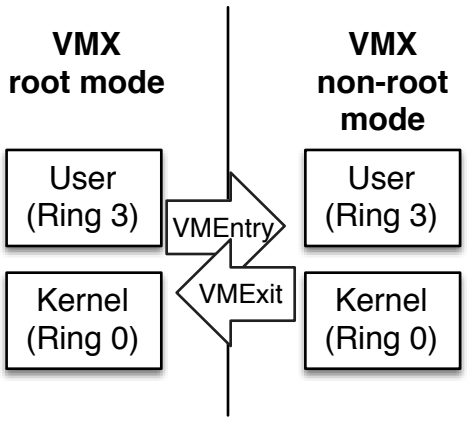


図 3　VMEntry、VMExit

センシティブな命令以外でも、一般的なハイパーバイザでは CPU 外部のハードウェアから割り込みがかかっ

たときにも、ハイパーバイザ側で割り込みをハンドルするために VMX non Root Mode を中断して VMExit する必要が生じます。ですが、どのようなイベントが発生したときに VMExit を起こすべきなのかは、ハイパーバイザの設計に依存するところがあります。

このため、VT-x ではあらかじめ用意されたイベントの中から「これは VMExit する、これは VMExit しない」というようにハイパーバイザ側でコンフィギュレーション可能になっています。このコンフィギュレーション情報は VMCS と呼ばれ、ハイパーバイザのメモリ空間上にゲストマシンごとに別々に用意する必要があります。

ハイパーバイザは VMCS（Virtual Machine Control Structure）に対してゲストマシンの設定を行い、VMEntry 前に CPU へアドレスを通知し、CPU は VMCS の値に基づいてゲストマシンの制御を行います。また、コンフィギュレーション情報だけでなく、VMExit されたときのゲストマシンのレジスタ値や VMExit 要因などゲストマシンのステート情報も CPU によって VMCS へ記録されます。

## VT-x を用いたハイパーバイザのライフサイクル

それでは、これまでに説明した Intel VT-x を用いたハイパーバイザ上でゲストが実行される一連のサイクルをより詳しく見ていきましょう。

### (1)「VT-x を有効化」

VT-x の機能を使うには、まず VT-x を有効にする必要があります。VT-x を有効にするには、CR4 レジスタの VMXE ビットに 1 をセットした上で VMXON 命令を発行します。この作業は Ring0（特権モード）で行われる必要があります。これは、以降の作業についても基本的に同様です（ですので、ユーザモードのプログラムで VT-x を用いたハイパーバイザを実装することはできません)。

また、厳密にはプロテクトモードでなければならない、A20 モードが無効化されていなければならないなどの制約があり、VT-x を有効化するための条件を満たしていない場合は VMXON 命令発行時に例外が発生します。

### (2)「VMCS を初期化、ゲストマシンの設定をロード」

ゲストマシンを実行する為には、前述した VMCS がゲストマシンの CPU((通常、仮想 CPU(vCPU) と呼ばれます))1 つに対して 1 つ用意され、適切に初期化されていなければなりません。この VMCS はメインメモリ上の 4KB の領域でページ境界（4KB）でアラインされている必要があります。メモリ上の領域なので直接読み書きできるのですが、VMCS の内部構造は公開しておらず、専用の命令を介して間接的にアクセスすることになっています。

ここでは、VMCS のアドレスを CPU へ設定する為に VMPTRLD 命令を実行し、続いて VMCS 上のフィールドへ書き込みを行う為に VMWRITE 命令を実行します。ここで初期化する情報としては、ゲストレジスタ

の初期値や VMX non-root mode の挙動の制御などがあります。詳しくは次号でこの「VMCS の構造」を解説しますので、しばしお待ち（ご期待）ください。

### (3)「CPU に VMCS をセット」

(2) ですでに設定されていれば必要ありませんが、(4) の VMEntry に先立って VMPTRLD 命令で VMCS のアドレスを CPU へ設定する必要があります。

### (4)「ハイパーバイザのレジスタを退避」

VMEntry に先立って、VMExit 時にハイパーバイザの状態を復元できるようにするため、レジスタの値を退避しておく必要があります。一部のレジスタは VMExit 時に自動的に復帰するために、VMCS へ退避します。このための領域が “Host state area” として VMCS に定義されており、VMWRITE 命令で書き込みます。それ以外のレジスタは PUSHA、PUSHF 命令などで手動で退避します。

### (5)「ゲストのレジスタを復帰」

ゲストのレジスタも同様、一部のレジスタは VMEntry 時に自動的に復帰されます。この為の領域として VMCS に “Guest state area” が定義されています。ただし、これも一部のレジスタのみが対象であるため、それ以外のレジスタは VMCS とは別にレジスタを保存するためのメモリ領域を確保しておき、ここから MOV 命令などを用い手動で復帰します。

### (6)「VMX non Root Mode へ VMEntry」

CPU のモードを切り替えて (3) で指定された VMCS が指すゲストマシンを実行します。この時、(2) で VMCS を初期化してから初めての VMEntry の時のみ VMLAUNCH 命令で VMEntry を行います。以降の VMEntry は VMRESUME 命令で行います。

### (7)「ゲストマシン実行」

CPU は何らかの理由によって VMExit されるまでの間、VMX non Root Mode でゲストマシンを実行します。

### (8)「何らかのトラップ要因が発生、VMExit する」

センシティブ命令の実行や外部割り込みなどの要因により VMExit が発生すると、CPU は VMC の “VM-exit information fields” へ VMExit 要因を書き込み、“Guest state area” へゲストのレジスタを退避し、“Host state area” からホストのレジスタを復帰し、CPU のモードを VMX Root mode へ切り替えます。

### (9)「ゲストのレジスタを退避」

VMExit で退避対象になっていないレジスタを (5) で使用しているメモリ領域へ MOV 命令などを用いて手動で退避します。

### (10)「ハイパーバイザのレジスタを復帰」

VMExit で復帰対象になっていないレジスタを (4) で退避された領域から POPA、POPF 命令などを用いて手動で復帰します。

### (11)「VMExit 要因を調べ、要因に合わせたエミュレーション処理を行う」

VMCS の “VM-exit information fields” に書き込まれた VMExit 要因を調べ、要因に合わせたエミュレーション処理を行います。特権レジスタへの読み書きであればレジスタの挙動をエミュレーションする必要がありますし、ハードウェアへの I/O 命令であれば対象のハードウェアの挙動をエミュレーションする必要があります。どのようなイベントで VMExit を発生させることができるのかについては (“Intel(R) 64 and IA-32 Architectures Software Developer Manuals”) を参照してください。

また、この時、次回の VMEntry が異なる CPU から実行される可能性がある場合は、必ず VMCLEAR 命令を実行して CPU から VMCS のアドレスをクリアしなければなりません。これは、VMCLEAR 命令を実行するまでは VMCS のデータがメモリへライトバックされていることが保証されないためです。

### (12)「(3) に戻る」

以上のように、VMX non Root Mode でのゲストマシン実行と VMExit 要因ごとのエミュレーション処理を繰り返すことにより、仮想化環境が実現されます（図 4)。

## まとめ

いかがでしたでしょうか。今回は初回ということで、まず x86 アーキテクチャにおける仮想化の歴史を振り返り、次に Intel VT-x によるハードウェア仮想化支援機構の概要について見てきました。

お気づきの方もいらっしゃると思いますが、今回の Intel VT-x の解説では、主に「CPU の仮想化」についてのみ解説を行なっています。実際にハイパーバイザを動かすには、大きな括りとしてもう 2 つ、「メモリの仮想化」と「I/O の仮想化」を考えなければなりません。

ですので次回は、今回紹介しきれなかった「CPU の仮想化」の詳細と「メモリの仮想化」について解説していこうと思います。

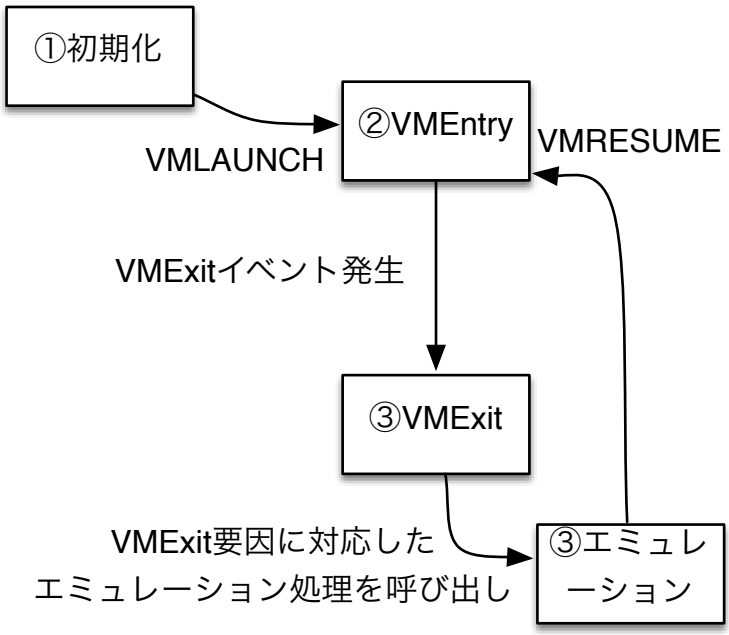


図 4　VT-x のライフサイクル

## ライセンス



## 参考文献

“Formal Requirements for Virtualizable Third Generation Architectures.” http://www.dc.uba.ar/materias/so/2010/veran

“Intel(R) 64 and IA-32 Architectures Software Developer Manuals.” http://www.intel.com/content/www/us/en/processor