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１１回 virtio による準仮想化デバイス その１「virtio の概要と Virtio PCI」

はじめに

前回までに、ハイパーバイザでの I/O 仮想化の実装を、BHyVe のソースコードを例に挙げ解説してきました。今回は、ゲスト OS の I/O パフォーマンスを大きく改善する「virtio」準仮想化ドライバの概要と、virtio のコンポーネントの１つである「Virtio PCI」について解説します。

完全仮想化デバイスと準仮想化デバイス

x86 アーキテクチャを仮想化する手法として、「準仮想化」と呼ばれる方式があります。これは、Xen によって実装された方式です (“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”)。

準仮想化では、仮想化に適した改変をゲスト OS に加えます。これにより、改変を加えずゲスト OS を仮想化する完全仮想化と比較し、高いパフォーマンスが得られるようになります。この準仮想化では、ハードウェア仮想化支援も必要としていませんでした。

現在では、先に説明した準仮想化よりも完全仮想化が広く使われるようになっています。これは、ハードウェア仮想化支援機能を持つ CPU が普及し、多くの環境で高速なハードウェア支援機能が利用できるようになったためです。

しかし、完全仮想化を採用したハイパーバイザにおいても、部分的に準仮想化の概念を取り入れています。その部分としては、システム全体のパフォーマンスに大きく影響を及ぼす仮想デバイスが挙げられます。

このような仮想デバイスのことを「準仮想化デバイス」と呼びます。これに対して、実ハードウェアと同じデバイスをエミュレーションしている仮想デバイスのことを「完全仮想化デバイス」と呼びます。

完全仮想化デバイスでは、実ハードウェア向けの OS に付属しているデバイスドライバをそのまま使用できま。しかし、準仮想化デバイスではゲスト環境向けに、準仮想化デバイス用のデバイスドライバをインストールする必要があります。

準仮想化デバイスのフレームワークとして「virtio」があります。これは、特定のハイパーバイザやゲスト OS に依存しないフレームワークです。その仕様やソースコードは公開されているため、ハイパーバイザ側で

は KVM・VirtualBox・lguest・BHyVe など、ゲスト OS 側では Linux・FreeBSD・NetBSD・OpenBSD・Windows・MonaOS など多くの実装が存在しています。

また、Xen や VMware、Hyper-V などのハイパーバイザでも、同様の考え方を採用した準仮想化ドライバが採用されています[*1]。

完全仮想化デバイスが遅い理由

実機上でネットワークインターフェースやブロックデバイスなどに対して I/O を行う場合、OS はデバイスドライバを介して各デバイスのハードウェアレジスタに対して読み書きを行います。

前回までの記事で解説したとおり、Intel VT-x ではこのハードウェアレジスタに対するアクセスを検知するたびに VMExit を行います。ハイパーバイザは VMExit を受けてデバイスエミュレーション処理を行います。

この一連の処理は仮想化を行うときだけに発生するオーバーヘッドであり、この部分の処理の重さが実機と比較した時の I/O 性能の差に現れてきます。

より詳細には次のようなコストが発生し、実機上の I/O と比較してレイテンシが大きくなる可能性があります。

VMX non-root mode・VMX root mode 間のモード遷移にかかるコスト

ハードウェアレジスタアクセス時の VMExit とゲスト再開時の VMEntry では、それぞれ VMX non-root mode と VMX root mode の間でモード遷移が発生します。この遷移のコストは CPU の進化に伴い小さくなってきているものの、VMExit・VMEntry にそれぞれ 1000 サイクルほど消費します。

デバイスエミュレータの呼び出しにかかるコスト

多くの場合、ハイパーバイザのデバイスエミュレータはユーザプロセス上で動作しています。このため、ハードウェアレジスタアクセスをエミュレートするにはカーネルモードからユーザモードへ遷移し、エミュレーションを行ってからカーネルモードへ戻ってくる必要があります。

また、ユーザプロセスはプロセススケジューラが適切と判断したタイミングで実行されるため、VMExit 直後にデバイスエミュレータのプロセスが実行される保証はありません。

同様に、ゲスト再開の VMEntry についてもデバイスエミュレーション終了直後に行われる保証はなく、スケジューリング待ちになる可能性もあります。

また、たいていの完全仮想化デバイスでは一度の I/O に複数回レジスタアクセスを行う必要があります (たとえば、ある NIC の受信処理では 5〜6 回のレジスタアクセスが必要になります)。レジスタアクセスを行うたびに、上述の処理が発生し、大きなコストがかかります。高速な I/O が求められるデバイスの場合には、ここが性能上のボトルネックになります。

[*1] virtio とは異なる独自方式のドライバが用いられています。

virtio の概要

virtio は前述のようなデバイスの完全仮想化にかかるコストを減らすため、ホスト・ゲスト間で共有されたメモリ領域上に置いたキューを通じてデータの入出力を行います。

VMExit はキューへデータを送り出したときに、ハイパーバイザへ通知を行う目的でのみ行われ、なおかつハイパーバイザ側がキュー上のデータを処理中であれば通知を抑制することも可能す。このため、完全仮想化デバイスと比較して大幅にモード遷移回数が削減されています。

virtio は、大きく分けて Virtio PCI と Virtqueue の 2 つのコンポーネントからなります。

Virtio PCI はゲストマシンに対して PCI デバイスとして振る舞い、以下のような機能を提供します。

- デバイス初期化時のホスト<->ゲスト間ネゴシエーションや設定情報通知に使うコンフィギュレーションレジスタ
- 割り込み (ホスト->ゲスト)、I/O ポートアクセス (ゲスト->ホスト) によるホスト<->ゲスト間イベント通知機構
- 標準的な PCI デバイスの DMA 機構を用いたデータ転送機能

Virtqueue はデータ転送に使われるゲストメモリ空間上のキュー構造です。デバイスごとに 1 つまたは複数のキューを持つことができます。たとえば、virtio-net は送信用キュー・受信用キュー・コントロール用キューの 3 つを必要とします。

ゲスト OS は、PCI デバイスとして virtio デバイスを検出して初期化し、Virtqueue をデータの入出力に、割り込みと I/O ポートアクセスをイベント通知に用いてホストに対して I/O を依頼します。

今回の記事では、このうち Virtio PCI についてより詳しく見ていきましょう。

PCI のおさらい

Virtio PCI の解説を行う前に、まずは簡単に PCI についておさらいしましょう。

PCI デバイスは Bus Number・Device Number で一意に識別され、1 つのデバイスが複数の機能を有する場合は Function Number で個々の機能が一意に識別されます。

これらのデバイスは PCI Configuration Space、PCI I/O Space、PCI Memory Space の 3 つのメモリ空間を持ちます。

PCI Configuration Space はデバイスがどこのメーカーのどの機種であるかを示す Vendor ID・Device ID や、PCI I/O Space・PCI Memory Space のマップ先アドレスを示す Base Address Register、MSI 割り込みの設定情報など、デバイスの初期化とドライバのロードに必要な情報を多数含んでいます。

PCI Configuration Space にアクセスするには、次のような手順を実施する必要があります。 1. デバイスの Bus Number・Device Number・Function Number とアクセスしたい領域のオフセット値を Enable Bit と

ともに CONFIG_ADDRESS レジスタ[*2]にセットする。CONFIG_ADDRESS レジスタのビット配置は表 1 のとおり 2. CONFIG_DATA レジスタ[*3]に対して読み込みまたは書き込みを行う

OS は PCI デバイス初期化時に、Bus Number・Device Number をイテレートして順に PCI Configuration Space を参照することで、コンピュータに接続されている PCI デバイスを検出できます。

PCI I/O Space は I/O 空間にマップされており、おもにデバイスのハードウェアレジスタをマップするなどの用途に使われているようです[*4]。

図 1 のように PCI Memory Space は物理アドレス空間にマップされており、ビデオメモリなど大きなメモリ領域を必要とする用途に使われているようです。どちらの領域もマップ先は PCI Configuration Space の Base Address Register を参照して取得する必要があります。

ビットポジション	内容
31	Enable Bit
30-24	Reserved
23-18	Bus Number
15-11	Device Number
10-8	Function Number
7-2	Register offset
1-0	0

表 1: CONFIG_ADDRESS register

Virtio PCI デバイスの検出方法

virtio デバイスはゲストマシンに接続されている PCI デバイスとしてゲスト OS から認識されます。

この際、PCI Configuration Space の Vendor ID は 0x1AF4、Device ID は 0x1000 - 0x1040 の値が渡されます。さらに、virtio デバイスの種類を判別するための追加情報として、表 2 のような Subsystem Device ID が渡されます。

ゲスト OS はこれらの ID を見て適切な virtio 用ドライバをロードします。

[*2] PC では CONFIG_ADDRESS レジスタは I/O 空間の 0xCF8 にマップされています。
[*3] PC では CONFIG_DATA レジスタは I/O 空間の 0xCFC にマップされています。
[*4] PC では I/O 空間にマップされますが、他のアーキテクチャではメモリマップされる場合もあります。

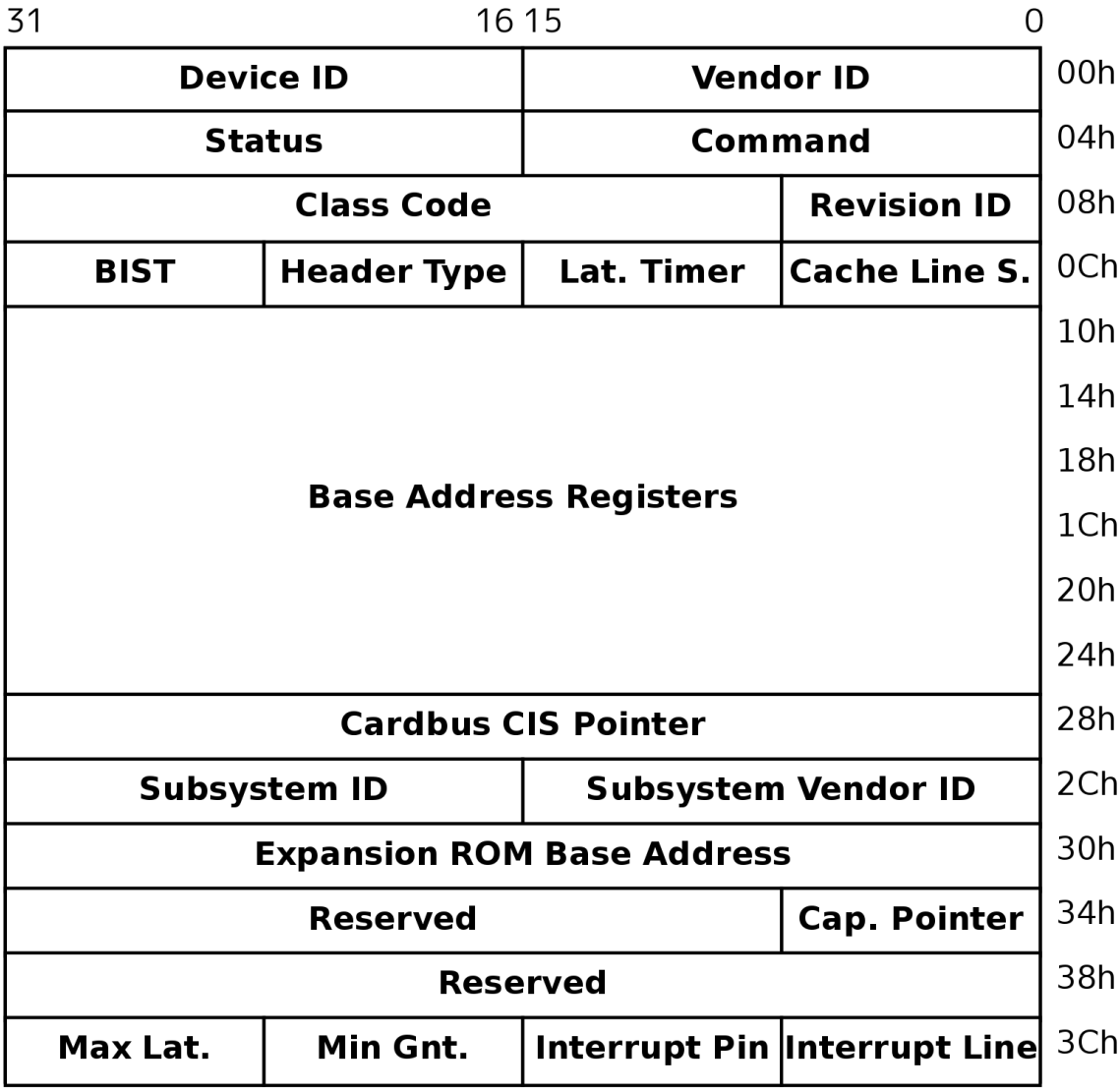

図 1　PCI Configuration Space

Subsystem Device ID	device type
1	network card
2	block device
3	console
4	entropy source
5	memory balooning
8	SCSI host
9	9P transport

Subsystem Device ID	device type

表 2: Subsystem Device ID

Virtio Header

Virtio Header は PCI I/O Space の先頭に置かれた virtio デバイスの設定用のフィールドで、ゲスト OS が virtio デバイスドライバを初期化するときに利用されます (表 3)。Virtio Header の終端部分 (MSI が無効の場合は 20byte、有効の場合は 24byte) からは device specific header が続きます。

表 4 に virtio-net の device specific header を示します。

offset	field name	bytes	direction	description
0	HOST_FEATURES	4	RO	ホストが対応する機能のビットフィールド
4	GUEST_FEATURES	4	RW	ゲストが有効にしたい機能のビットフィールド
8	QUEUE_PFN	4	RW	QUEUE_SEL で指定されたキューに割り当てる
				メモリ領域の物理ページ番号（PFN）
12	QUEUE_NUM	2	RO	QUEUE_SEL で指定されたキューのサイズ
14	QUEUE_SEL	2	RW	キュー番号
16	QUEUE_NOTIFY	2	RW	QUEUE_SEL で指定されたキューにデータがある事を通知
18	STATUS	1	RW	デバイスのステータス
19	ISR	1	RO	割り込みステータス
20	MSI_CONFIG_VECTOR	2	RW	コントロール用キューの MSI ベクタ番号
			（	MSI 有効時のみ存在）
22	MSI_QUEUE_VECTOR	2	RW	QUEUE_SEL で指定されたキューの MSI ベクタ番号
			（	MSI 有効時のみ存在）

表 3: Virtio header

offset	field name	bytes	direction	description
0	MAC	6	RW	MAC アドレス

offset	field name	bytes	direction	description
6	STATUS	2	RO リ	ンクアップ状態などの virtio-net 固有ステータス
8	MAX_VIRTQUEUE_PAIRS	2	RO 最	大 RX ／ TX キュー数（マルチキュー用）

表 4: virtio-net specific header

Virtio PCI デバイスの初期化処理

ゲスト OS における Virtio PCI を用いた virtio デバイスの初期化処理は次のようになります。

■Step 1　通常の PCI デバイスの初期化ルーチンを実行し、Vendor ID・Device ID が virtio のものを発見します。

■Step 2　デバイスの PCI I/O Space のマップ先アドレスを取得し、Virtio Header の STATUS フィールドに ACKNOWLEDGE ビットをセットします (STATUS フィールドが用いるビット値は表 5 に記載)。

■Step 3　Subsystem Device ID に一致するドライバをロードします。たとえば ID が 1 の場合は virtio-net をロードします。

■Step 4　ドライバがロードできたら Virtio Header の STATUS フィールドに DRIVER ビットをセットします。

■Step 5　デバイス固有の初期化処理を実行します。virtio-net の場合、virtio-net specific header から MAC アドレスをコピーする、NIC としてネットワークサブシステムに登録するなどの処理が行われます。この時、Virtio Header の HOST_FEATURES フィールドで示されているデバイスで使える機能のうち、ドライバで使用したい機能のビットを GUEST_FEATURES へ書き込みます。FEATURES の全ビットの紹介は省略しますが、たとえば virtio-net で Checksum Offloading を使いたい場合はビット 0、TSOv4 を使いたい時はビット 11 を有効にする必要があります。

■Step 6　デバイスに必要な数のキューをアロケート、Virtio Header を通じてホストへアドレスを通知します (「キューのアロケート処理」に詳述)。

■Step 7　ドライバの初期化処理がすべて成功したら Virtio Header の STATUS フィールドに DRIVER_OK ビット、途中で失敗したら FAILED ビットをセットします。

bit	name	description
1	ACKNOWLEDGE	ゲスト OS はデバイスを発見
2	DRIVER	ゲスト OS はデバイス向けのドライバを保有
3	DRIVER_OK	ゲスト OS はドライバ初期化を完了
7	FAILED	初期化失敗

表 5: device status

キューのアロケート処理

「Virtio PCI デバイスの初期化処理」の第 6 段階で言及したキューのアロケート処理は、次のような手順をキューごとに実施する必要があります。たとえば virtio-net の場合は 3 つのキューが必要なので、3 回繰り返します。

■Step 1　設定を行うキューの番号を Virtio Header の QUEUE_SEL フィールドに書き込みます。

■Step 2　Virtio Header の QUEUE_NUM フィールドを読み込みます。この値がこれから設定を行うキューのキュー長になります。値が 0 だった場合、ホスト側はこの番号のキューを使うことを認めていないため使用できません。

■Step 3　キューに使うメモリ領域をアロケートします。アロケートするサイズはキュー長に合わせたサイズで、先頭アドレスはページサイズにアラインされている必要があります。

■Step 4　メモリ領域の先頭アドレスの物理ページ番号を Virtio Header の QUEUE_PFN にセットします。

■Step 5　MSI 割り込みが有効な場合、Virtio Header の MSI_CONFIG_VECTOR フィールドまたは MSI_QUEUE_VECTOR フィールドに割り込みベクタ番号を書き込みます。どちらのフィールドに書き込むかはキューがコントロール用キューか否かによって異なります。

まとめ

virtio の概要と Virtio PCI の実装について解説しました。次回はいよいよ Virtqueue とこれを用いた NIC(virtio-net) の実現方法について見ていきます。

ライセンス

参考文献

“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.” http://syuu1228.github.io/howto_implement_hyperviso