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3回 I/O 仮想化「デバイス I/O 編」

ハイパーバイザによる I/O デバイスエミュレーション方法

OS の主要な機能の 1 つに、コンピュータ上の各種デバイスに対するアクセス (デバイスへの I/O) の抽象化が挙げられます。OS 上で動作するアプリケーションは、ファイルシステムやソケットインターフェースなどの形に抽象化されたものを用いて、デバイスへアクセスを行います。今回は、仮想化環境でゲスト OS のデバイスアクセスをどのように仮想化するか、これに焦点を当て解説します。

I/O デバイスへのアクセスとハードウェアレジスタ

OS のうち、各種デバイスに対するアクセス機能を司るプログラムをとくに「デバイスドライバ」と呼びます。では、デバイスドライバとデバイスとのやりとりはどのように行われるのでしょうか。それぞれのデバイスは、デバイスをソフトウェア (デバイスドライバ) から制御するためのハードウェアレジスタを持っています。OS はデバイスドライバを用いてこのハードウェアレジスタを読み書きすることによって「HDD のセクタを読み取る」「LAN へパケットを送出する」「画面を描画する」などの目的を果たしています。また、デバイスによっては割り込み機能や DMA 機能を持っています[*1] 。デバイスアクセスの例として、シリアルポートの受信処理を見てみましょう。

リスト１. シリアルポートの受信処理

```
unsigned char read_com1(void) {
  while ((read_reg_byte(COM1_LSR) & 1) == 0);
  return read_reg_byte(COM1_RBR);
}
```

リスト 1 は、最も単純に実装した場合のシリアルポートの受信処理のコードです。解説を単純化するため、初期化処理や割り込みの処理などは割愛しています。シリアルポートからデータを受信するには、Line Status

[*1] 割り込みは OS へ何らかのメッセージを通知するために用いられます。 また、 DMA は CPU 負荷を下げるためにデバイスから CPU を介さず直接データをメモリへ転送する機能です。

Register の Data Available ビットをチェックしてデータが着信するまで待ちます。そして、データが着信したら Receiver Buffer Register から読み取ります。

複数バイトのデータを読み出すには、バイト数分この処理を繰り返す必要があります。ここでは Line Status Register のチェック処理をビジーウェイトで実装していますが、実際のドライバではビジーウェイトは CPU 時間を無駄に消費し、他の処理の実行を阻害するのでほとんどの場合使いません。代わりに割り込みを用います。シリアルポートの受信割り込みの場合、Data Available ビットが立つタイミングで割り込みを発生することができます。このため、ドライバは割り込みハンドラでリスト 1 と同様のコードを実行すれば、ビジーウェイトを避けて受信処理を行うことができます。

I/O マップド I/O とメモリマップド I/O

各デバイスのハードウェアレジスタの読み書き (デバイスへの I/O) はどのように実現されているのでしょうか。ここの方法として、I/O マップド I/O とメモリマップド I/O の二種類の方式があり、通常アーキテクチャごとにどちらかの方式をとります。しかし、x86 アーキテクチャでは、歴史的な事情により両方式を併用しています。

I/O マップド I/O では、メモリ空間とは独立したデバイス専用のアドレス空間 (I/O 空間) が存在しており、ここに各デバイスのハードウェアレジスタを割り付けます。なお、どのデバイスをどの番地にするかは、固定的に決まっているアーキテクチャと動的に決まるアーキテクチャとがあります。 I/O 空間へは専用の命令 (IN 命令、OUT 命令) を用いてアクセス を行います。前述のシリアルポートの受信処理の read_reg_byte() 関数及びレジスタの宣言はリスト 2 のようになります。

リスト 2 I/O マップド I/O での read_reg_byte() 関数およびレジスタの宣言

```
#define COM1_PORT (0x3f8)
#define COM1_LSR (COM1_PORT + 0)
#define COM1_RBR (COM1_PORT + 5)
unsigned char read_reg_byte(unsigned short port) {
  unsigned char val;
  asm volatile("inb %1, %0" : "=a"(val) : "Nd"(port));
  return val;
}
```

一方、メモリマップド I/O では、各デバイスのハードウェアレジスタをメモリ空間の一部に割り付けます。こちらも、どのデバイスをどの番地にするかは、固定的に決まっているものと動的に決まるものとがあります。ハードウェアレジスタへのアクセスには MOV 命令など通常のメモリアクセスと同様の手順を用います。前述のシリアルポートの受信処理の read_reg_byte() 関数及びレジスタの宣言はリスト 3 のようになります。

リスト 3 マップド I/O での read_reg_byte() 関数およびレジスタの宣言

```
#define COM1_PORT (0x40100000)
#define COM1_LSR ((void *)(COM1_PORT + 0))
#define COM1_RBR ((void *)(COM1_PORT + 5))
unsigned char read_reg_byte(void *addr) {
  return *((unsigned char *)addr);
}
```

ゲストマシン上で各種デバイスをサポートするには、この 2 つの種類の I/O を仮想化し、アクセスされたデバイスに応じたエミュレーション処理を行う必要があります。

VT-x における I/O エミュレーションの基本的な考え方

連載 1 回目の「VT-x を用いたハイパーバイザのライフサイクル」で解説したとおり、次のようなループの繰り返しによりデバイス I/O のエミュレーション を行います。

1. [ゲスト] ゲスト環境上でデバイスへの I/O が実行される
2. [ゲスト]I/O の実行を契機に VMExit 発生
3. [ハイパーバイザ] アクセス先のデバイス、アクセス幅、アクセス方向、書き込み先・読み込み元などを特定
4. [ハイパーバイザ] デバイス I/O のエミュレーション処理を行う
5. [ハイパーバイザ]VMEnter してゲストを再開させる
6. [ゲスト]I/O 実行の次の命令から実行再開

この処理は、ハードウェアレジスタへの読み書きが行われるごとに繰り返されます。したがって、1 回のハードウェアアクセス処理に必要なハードウェアレジスタへのアクセス回数が多ければ多いほどオーバーヘッドが大きくなります。

VT-x における I/O マップド I/O のハンドリング方法

VT-x において I/O マップド I/O をハンドリングするには、まず VMCS へ設定を行って I/O ポートへのアクセス時に VMExit を発生させる必要があります (VMCS については、連載 1 回目と 2 回目を参照してください)。

設定には 2 とおりあり、すべての I/O ポートへのアクセスで VMExit を発生させる設定と、特定の I/O ポートへのアクセスにのみ VMExit を発生させる設定です。すべての I/O ポートへのアクセスで VMExit を発生させるには、VMCS の VM-Execution Control Fields の Unconditional I/O exiting を 1 にします。ま た、

特定の I/O ポートへのアクセスにだけ VMExit を発生させるには、VMCS の VM-Execution Control Fields の Use I/O bitmaps を 1 にして、VM-Execution Control Fields の I/O-Bitmap Address A と I/O-Bitmap Address B に I/O-bitmap A と I/O-bitmap B のアドレスを設定します。I/O-bitmap B は I/O ポート番号 8000H から FFFFH までを表す同様のテーブルです (図 1)。

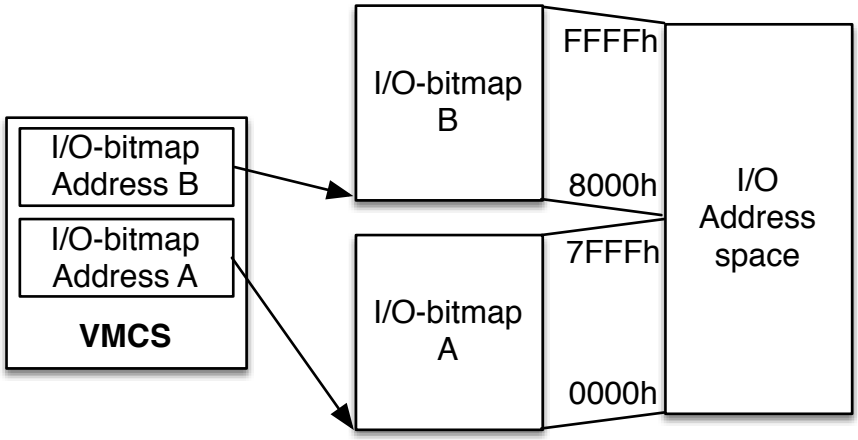

図 1　I/O-bitmap と I/O アドレス空間

これらの設定で I/O アクセス時の VMExit を有効にし、ゲスト OS がデバイスドライバ経由で I/O ポートへアクセスを行うと、VMExit Reason 30 (I/O Instruction) の VMExit が発生します。Exit 要因は VMCS の VM-Exit Information Fields の Exit reason フィールドに書かれており、ハイパーバイザはこれをもとに Exit 要因に合わせた処理を行います。今回の例の場合はデバイスへの I/O アクセスをエミュレーションしますが、Exit 要因だけでは何番の I/ O ポートへアクセスされているのか、アクセス方向が読み込みだったのか、あるいは書き込みだったのかがわかりません。これらの情報は、VM-Exit Information Fields の Exit qualification フィールドで提供されます (表 1)。

ビットポジション	内容
2:00	アクセスサイズ (0 = 1byte、1 = 2byte、3 = 4byte)
3	アクセス方向 (0 = 書き込み、1 = 読み取り)
4	String 命令 (0 = 非 string、1 = string)
5	REP プレフィックス (0 = REP あり、1 = REP)
6	ポート番号指定方法 (0 = DX 間接、1 = 即値)
15:07	Reserved
31:16:00	ポート番号
63:32:00	Reserved

ビットポジション	内容

表 1: Exit Reason 30 のときの Exit qualification

このフィールドは Exit 要因ごとに異なる追加情報 を提供しており、VMExit Reason 48 の場合は表 3 のような情報を提供します。ハイパーバイザは Exit qualification フィールドからポート番号などの情報を読み込み、ポート番号に合わせたデバイスエミュレーション処理を実行します。

デバイスエミュレー ション処理を行う際、アクセス方向が読み込み方向ならば読んだデータの書き込み先、書き込み方向ならば書き込むデータの読み込み元を把握する必要があります。しかし、I/O ポートアクセスの場合は非 string 命令 (IN/OUT) ならば EAX レジスタを使うこと、string 命令 (INS/INSB/INSW/INSD/OUTS/OUTSB/OUTSW/OUTSD) ならば ES:ESI で指定されたメモリアドレスを使うこと、と固定的に決めら れています。

このため、Exit qualification のビット 4 を見て string 命令か否かを判別すれば、ハイパーバイザのエミュレーション処理において、どこから書き込み先/読み込み元を取得すれば良いのかがわかります。

VT-x におけるメモリマップド I/O のハンドリング方法 (シャドーページングの場合)

VT-x におけるメモリマップド I/O は、メモリ仮想化がソフトウェアで行われている (シャドーページング) か、ハードウェアで行われている (EPT) かで 2 とおりあります。まず、シャドーページングの場合から説明します。

シャドーページング環境においてメモリマップド I/O をハンドリングするには、デバイスがマップされたアドレスへのアクセスが発生した時に、ページフォルトを発生させる必要があります。そのために、シャドーページテーブル上のデバイスがマップされたアドレスに対応するページテーブルエントリのプレゼントビットを 0 にします。

前回の記事で説明したとおり、シャドーページン グ時には VMCS の VM-Execution Control Fields の Exception Bitmap の 14bit 目 (page fault exception) にビットを設定して、ページフォルトで VM ExitReason 0 (Exception or non-maskable interrupt) を発生させます。VMExit が発生した時、ハイパーバイザは Exit 要因が 0 であることを確認したあと、 VMCS の VM-Exit Information Fields に ある VM-exit interruption information を参照します (表 2)。

ビットポジション	内容
7:00	割り込みベクタ番号
10:08	割り込みタイプ (0 =外部割り込み、2 = NMI、3=ハードウェア例外、6 =ソフトウェア例外)
11	Error code が正常

ビットポジション	内容
12	IRET による NMI ブロック解除
30:13:00	Reserved
31	VM-exit interruption information が正常

表 2: VM-exit interruption information

この場合、ハイパーバイザは割り込みベクタ番号 が 14 (#PF 例外) で、割り込みタイプがビット 3 (ハードウェア例外) で、VM-exit interruption information が正常であることを確認します。ページフォルトで VMExit したことが確認されたら、前述の Exit qualification フィールドを読み込みます。ページフォルト例外による VMExit の場合、このフィールドには CR2 レジスタの値 (ページフォルト例外が発生したリニアアドレス) になっています。

さて、これでメモリマップド I/O 対象のアドレス はわかりました。しかし、I/O ポートアクセスのとき には Exit qualification フィールドから取得できていたアクセスサイズ、アクセス方向、データの書き込み先・読み込み元がわかりません。実は VT-x ではこれらの情報を提供していないのです。これらの情報を得るため、ハイパーバイザは次のような処理を実行する必要があります。

1. ページフォルト例外発生時の RIP[*2] を VMCS の Guest-State Area の RIP フィールドから取得
2. ゲストマシンのメモリ空間へアクセスして命令のバイト列を読み込み
3. 命令をデコードしてアクセスサイズ、アクセス方向、データの書き込み先・読み込み元を取得
4. 3 の情報を元にしてデバイスアクセスのエミュレーションを実行

つまり、メモリマップド I/O を実施した 1 命令に限りソフトウェアエミュレーション処理を行うことになります。

当然ながら、この処理の分 I/O マップド I/O と比較してオーバーヘッドが高くなります。

VT-x におけるメモリマップド I/O のハンドリング方法 (EPT の場合)

EPT 環境においてメモリマップド I/O をハンドリングする場合は、デバイスがマップされたアドレスへのアクセスが発生した時に、VMExit reason 48 (EPT violation) で VMExit させる必要があります。そのために、EPT 上のデバイスがマップされたアドレスに対応するページテーブルエントリの Read access ビットと Write access ビットをどちらも 0 にします。これによってゲストマシンがデバイスがマップされたアドレスへアクセスした時に VMExit が発生するようになります。VMExit が発生したとき、ハイパーバイザは Exit 要因が 48 であることを確認したあと、 VMCS の VM-Exit Information Fields に ある Guest-physical

[*2] RIP は実行中の命令のアドレスを持つレジスタ。 32bit モードでは EIP と呼ばれます。

address を読み込みます。このアドレスが、VMExit Reason 48 を発生させたアクセス (デバイスへの I/O) になります。さらに、VM Exit qualification フィールドを参照するとアクセス方向 (read または write) を得ることができます。しかし、I/O をエミュレーションするにはアクセスサイズ、データの書き込み先・読み込み元などの情報が足りません (表 3)。

ビットポジション	内容
0	EPT violation の原因は data read
1	EPT violation の原因は data write
2	EPT violation の原因は instruction fetch
3	アクセスされたページに対応する EPT エントリの read access と、 この Exit qualification の 0 ビット目との AND
4	アクセスされたページに対応する EPT エントリの write access と、 この Exit qualification の 1 ビット目との AND
5	アクセスされたページに対応する EPT エントリの execute access と、 この Exit qualification の 2 ビット目との AND
6	Reserved
7	VMCS の VM-Exit Information Fields の Guest-linear address が有効
8	1 = EPT violation の原因がゲストフィジカルアドレスへのアクセス 0 = EPT violation の原因が EPT のページウォーク中や EPT のページテーブルエントリの更新
11:09	Reserved
12	IRET による NMI ブロック解除
63:13:00	Reserved

表 3: Exit Reason 48 のときの Exit qualification

シャドーページングの場合と同様に、VT-x はこれらの情報を提供していません。このため、ハイパーバイザはシャドーページングの場合と同様に Exit 要因になった命令をソフトウェアエミュレーションする必要があります。このため、メモリマップド I/O のハンドリングにおいては、EPT でもシャドーページングの場合と同様にオーバーヘッドが発生します。[*3]

[*3] EPT が一般的にシャドーページングより高い性能が出せないという意味ではなく、メモリーマップド I/O に限っては性能が変わらないという意味です。

Local APIC 仮想化

Pentium Pro 以降の Intel の CPU には Local APIC という割り込みコントローラが内蔵されており、これがすべての割り込みの管理を行っています。Local APIC へのアクセス頻度は非常に高く、割り込みが発生するたびに割り込みハンドル終了を通知するため EOI レジスタへの書き込みを行います。さらに一 部の OS (おもに Windows XP) では割り込み優先度を 変更するために TPR レジスタの値を頻繁に書き換えます。また、クロックも Local APIC へ統合されているので、クロック割り込みごとにレジスタアクセスが行われます。

ゲスト OS からこのような高頻度のアクセスが行われると、頻繁に VMExit が発生し毎回レジスタアクセスのハンドリング処理を行わなければなりません。これらのオーバーヘッドが積み重なりゲストマシンのパフォーマンスが低下してしまうので、これを避けるため Local APIC の仮想化に関する機能がいくつか導入されています。

APIC access VMExit

VT-x には APIC access VMExit と呼ばれる Local APIC へのレジスタアクセス専用の Exit 要因が用意されています。これは、アクセス頻度の高い Local APIC へのレジスタアクセスのハンドリング処理を最適化しやすくするためのものだと思われます。

APIC access VMExit を使うには、VMCS の VM- Execution fields で Virtualize APIC accesses を有効 化し、シャドーページや EPT がゲスト環境の Local APIC アドレスの範囲に割り当てている物理ページ (APIC access page と呼ぶ) のアドレスを VMCS の VM-Execution fields へ設定します。これにより、APIC access page へアクセスが発生した際に、VMExit reason 44 (APIC access) が発生するようになります。このとき、VM Exit qualification を参照すると (表 4)、アクセスのあったレジスタとアクセス方向 (read だったのか write だったのか) がわかります。

ビットポジション	内容
11:00	アクセスのあったレジスタ (APIC page からのオフセット)
15:12	アクセスタイプ (read、write、execute など)
63:16:00	Reserved

表 4: Exit Reason 44 のときの Exit qualification

これだけの情報では命令エミュレーションが避けられないレジスタもありますが、 EOI レジスタに関しては「write only・0 を書くこと」と使い方がきわめて限定的に決まっているので命令エミュレーションをスキップしてハンドリングを完了させることができます。前述のとおりアクセス頻度が高いレジスタであるため、これだけでもそれなりのオーバーヘッド軽減が見込めるようです。

なお、APIC access page に対してメモリマップド I/O ハンドリングのためにページフォールトや EPT Violation が発生する設定をページテーブルエントリへ行なっている場合、ページフォールトや EPT Violation が優先されるため注意が必要です。

TPR shadow

通常のメモリマップド I/O のハンドリング方式では TPR レジスタへのアクセスは無条件に VMExit を引き起こします。TPR レジスタのしくみ上、 VMExit を用いたハイパーバイザからの介入が必要なのは、ある値より優先度を下げる方向の書き込みだけです。それ以外のケースでは、ゲスト OS から読み書きが正常に行えさえすればよく、VMExit を発生させる必要がありません。

このような挙動を実現させるため、 VT-x では TPR shadow という機能を用意しています。 TPR shadow を使うには、VMCS の VM-Execution fields で TPR shadow を有効にし、 Virtual APIC Page と呼ばれる物 理ページを TPR の値を格納する場所として用意し、VMExit を発生させるしきい値を TPR threshold というパラメータで指定します。

ゲストから TPR へのアクセスが発生した時、TPR の値が TPR threshold を下回ると VMExit Reason 43 TPR below threshold で VMExit します。下回らなかった場合は VMExit せずに Virtual APIC Page を使って TPR のアクセスが仮想化されます。

APIC-Register virtualization

TPR shadow で用いられている Virtual APIC Page を用いた Local APIC レジスタ仮想化のしくみは、 最新の Intel CPU では他の割り込み関連レジスタ群へも範囲が広げられています。これにより、VMExit 回数をより減らすことができるようになりました。

このように、同じ「VT-x」と呼ばれている機能でも CPU の世代によって少しずつ改良が加えられており、そのつど CPU 側でできることが増えてきています。

まとめ

いかがでしたでしょうか。今回は Intel VT-x におけるデバイス I/O エミュレーションの実装方法を中心に解説してきました。次回は「割り込みの仮想化」を中心に解説します。

ライセンス