Vol.2010-OS-113 No.1
2010/1/27

ARMアーキテクチャ用仮想マシンモニタの実装

鈴 木 章 浩[†1] 追 川 修 一[†1]

近年，ハードウェアの性能向上によって組み込み機器における仮想マシンモニタ (VMM) の利用が議論されるようになってきている．そこで本研究では組み込み機器用の CPU である ARM アーキテクチャ用の VMM を設計し，実装した．本研究では VMM を特権モード，ゲスト OS を非特権モードで動作させることでセンシティブ命令を例外という形で検知し，VMM で適切にエミュレーションを行う．また，仮想バンクレジスタとドメインを利用した仮想プロセッサモードを提供することでゲスト OS を非特権モード内で動作させることを可能にする．

Implementation of Virtual Machine Monitor for ARM Architecture

Akihiro Suzuki[†1] and Shuichi Oikawa[†1]

In these days, the performance gain of hardware promotes the use of Virtual Machine Monitors (VMMs) in embedded systems. Therefore, we implemented a VMM for the ARM architecture that is the most widely used CPU for embedded systems. Since the VMM executes in privileged mode and the guest OS executes in non-privileged mode, the VMM can catch the execution of sensitive instructions as an exception and emulate them appropriately. The guest OS can execute in non-privileged mode thanks to virtual bank registers and virtual processor mode using domain provided by the VMM.

1. はじめに

今日，組み込み機器は広く社会に普及し社会インフラを構築する重要な要素となっているが，その組み込み機器に要求される機能が，近年のハードウェアの目覚しい性能向上によって非常に高度なものになってきている．また，高速なユビキタス社会化に伴う企業への新製品開発サイクル短縮への要求も同様に増加しており，これらが原因となって組み込み機器製品の不具合や脆弱なセキュリティを誘発する恐れがある．

このような問題を解決する手段として仮想マシンモニタ (Virtual Machine Monitor：VMM)[1)] を導入することが考えられる．VMM を導入することでハードウェア上に複数の仮想マシン (Virtual Machine：VM) を構築することができるため，1 つのハードウェア上で論理的に分離された複数のゲスト OS を実行することができる．そのため，例えばセキュリティのレベルによって個別の VM とゲスト OS を提供した場合，組み込み機器の安定性の向上と強固なセキュリティの実現が可能となる．

VMM は現在主にサーバ用途で利用され，IA-32 アーキテクチャ専用の VMM が多数を占めているが，ハードウェアの性能向上によって将来的には組み込み機器にも VMM の技術が適用されるものと考えられる．そこで本研究では組み込み機器で広く用いられている ARM アーキテクチャ用の VMM を開発することで組み込み機器向けの CPU への VMM の導入を提案する．

本研究では問題を単純化するために VMM が提供する VM 環境を 1 つとし，その上で動作するゲスト OS として Linux を使用する．また，動作環境として ARM926EJ-S プロセッサ搭載の Integrator/CP ボードをエミュレートする QEMU[2)] を利用する．

本論文の構成は以下の通りである．まず第 2 章で関連研究について述べる．第 3 章では VMM 構築の際に必要となる ARM アーキテクチャの機能について述べる．第 4 章では第 3 章で述べた ARM アーキテクチャの機能を用いて VMM を設計し，実装する方法について述べる．第 5 章では第 4 章に基づいて実装した VMM を評価した結果について述べる．そして最後に第 6 章で本論文をまとめる．

2. 関連研究

本章では本研究についての関連研究について述べる．

2.1 Xen

Xen[3)] は Intel IA-32，IA-64，PowerPC 上で動作する Type I VMM である．また，ARM アーキテクチャへの対応も研究されている[4)]．Xen はドメインと呼ばれる仮想マシンの実行単位を用いる．ドメイン 0 と呼ばれるドメインでは，そこで動作する Linux がデバイスドライバを持ち，このドメインが実ハードウェアへのアクセスやその他のドメインを管理する

†1 筑波大学
University of Tsukuba

© 2010 Information Processing Society of Japan

特権的なドメインとなる．ドメイン 0 以外のドメインはドメイン U と呼ばれ，ゲスト OS が動作する．

Xen は仮想化のモデルとして準仮想化 (Paravirtualization) と完全仮想化 (Fullvirtualization) を提供している．準仮想化はゲスト OS を Xen が提供する VM 環境上で動作させるが，提供される VM 環境は Xen が動作する実ハードウェアとは異なる．そのため，VM 環境の操作をするためにはハイパーコールと呼ばれる命令を用いらなくてはならず，ゲスト OS に変更を加える必要がある．一方完全仮想化は Windows のように OS に変更を加えられない場合に対処するために用いられるが，これを利用するには CPU による仮想化支援機能を用いる必要がある．

本研究では CPU による仮想化支援機能を持たない ARM アーキテクチャを対象とした VMM 開発を行った．

2.2 TrustZone

ARM アーキテクチャには ARM1176JZF-S から採用された TrustZone[5] と呼ばれる機能が備わっている．これは ARM アーキテクチャの持つ 2 段階の保護レベル（特権・非特権）に直交する状態として Trust 状態と Non-Trust 状態を持つ．Trust 状態では既存の保護レベルと同様の振る舞いを行い，Non-Trust 状態では特権レベルであっても Trust 状態からのみアクセス可能と設定されたメモリ空間へのアクセスが制限される．また，TrustZone を制御する目的で Secure Monitor モードと呼ばれるモードが追加されている．Trust 状態と Non-Trust 状態の切り替えは Secure Monitor モードを通して行われる．

TrustZone を利用することで容易に VMM 構築ができる可能性があるが，本研究では TrustZone を備えていない ARM アーキテクチャを対象とした VMM 開発を行った．

3. ARM アーキテクチャ

本章では本研究で VMM を構築するうえで必要になる ARM アーキテクチャの機能について述べる．

3.1 プロセッサモード

ARM アーキテクチャのプロセッサモードには FIQ，IRQ，スーパーバイザ，アボート，未定義，システムの 6 つの特権モードとユーザから成る 1 つの非特権モードが存在する．一般的なアプリケーションプログラムはユーザモードで実行される．特権モードのうちシステム以外の 5 つのプロセッサモードは例外モードと呼ばれ，特定の例外が発生すると対応する例外モードにプロセッサモードが自動的に変更される．これらの例外モードには 3.2 節で述べるいくつかの追加レジスタが用意されている．

本研究でゲスト OS として利用する Linux はスーパバイザモードでカーネルプロセスが動作し，ユーザモードでユーザプロセスが動作する．また，例外発生時は 3.3.1 項で述べるスタブ内で各例外モードが利用される．

3.2 レジスタ構成

ARM アーキテクチャには図 1 に示す 37 個のレジスタが存在する．これらは 31 個の汎用レジスタと 6 個の psr (Program Status Register) から構成される．

レジスタ r0〜r12 は汎用的なレジスタとして使用され，ARM-Thumb 手続き呼び出し標準 (ATPCS)[6] に従った場合はレジスタ r0〜r3，r12 の 5 つのレジスタが汎用スクラッチレジスタとなる．レジスタ r13，r14，r15 はそれぞれスタックポインタ (sp)，リンクレジスタ (lr)，プログラムカウンタ (pc) として使用される．cpsr (Current Program Status Register) は現在の psr を保持するレジスタであり，spsr (Saved Program Status Register) は例外が発生した際に以前のプロセッサモードでの psr を保持するレジスタである，psr は条件コードフラグ，割り込みディセーブルビット，プロセッサモードビットフィールドを保持し，ARM アーキテクチャは cpsr を使用することで内部動作をモニタし制御する．

特定のレジスタは各プロセッサモード毎に図 1 の網掛けで示した専用の追加レジスタが用意されており，これをバンクレジスタと呼ぶ．バンクレジスタは各例外モードに固有のレジスタであり，各例外モードにプロセッサモードが遷移した場合はバンクレジスタが用意されているレジスタについてはバンクレジスタを使用する．

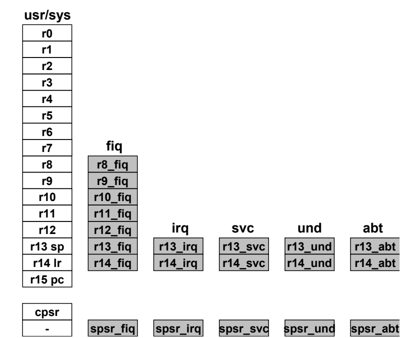

図 1 ARM アーキテクチャのレジスタ群

© 2010 Information Processing Society of Japan

表 1 例外の種類とプロセッサモード，例外ベクタアドレス

例外の種類	モード	例外ベクタアドレス
リセット	スーパーバイザ	0xFFFF0000
未定義命令	未定義	0xFFFF0004
ソフトウェア割り込み (SWI)	スーパーバイザ	0xFFFF0008
プリフェッチアボート	アボート	0xFFFF000C
データアボート	アボート	0xFFFF0010
IRQ (割り込み)	IRQ	0xFFFF0018
FIQ (高速割り込み)	FIQ	0xFFFF001C

3.3 例 外

ARM アーキテクチャは 7 種類の例外に対応している．例外が発生すると，発生した例外の種類に従ってメモリの固定アドレスから実行が開始される．この固定アドレスを例外ベクタといい，通常例外ベクタと高位例外ベクタのどちらかを使用する．表 1 に ARM アーキテクチャで対応している例外の種類と各例外時に使用されるプロセッサモード，高位例外ベクタのアドレスを示す．

例外が発生すると自動的にプロセッサモードが変更され，発生した例外モードの spsr に発生前の cpsr が保存される．また，例外発生時は割り込みが禁止される．例外の復帰時には spsr が cpsr に復帰され，復帰アドレスを格納している lr の値が pc にコピーされる．

3.3.1 Linux による例外の処理

3.3 節の例外機構が本研究で用いるゲスト OS である Linux でどのように利用されているかについて述べる．

Linux では多くの例外を処理する必要があるため，例外ベクタに例外ハンドラのスタブへのブランチ命令を設定する．このスタブは例外の種類毎に用意されているが，その中身はほぼ共通しており，以下のようなコードで構成される．

```
 1  vector_\name:
 2          .if \correction
 3          sub     lr, lr, #\correction
 4          .endif
 5          stmia   sp, {r0, lr}
 6          mrs     lr, spsr
 7          str     lr, [sp, #8]
 8          mrs     r0, cpsr
 9          eor     r0, r0, #(\mode ^ SVC_MODE)
10          msr     spsr_cxsf, r0
11          and     lr, lr, #0x0f
12          mov     r0, sp
13          ldr     lr, [pc, lr, lsl #2]
14          movs    pc, lr
```

6 行目で spsr から値を読み出すことで例外が発生した際のプロセッサモードを判断し，13 行目でその値を元に発生した例外を処理する例外ハンドラを決定する．8〜10 行目で cpsr からスーパーバイザモードに変更した spsr を設定しておくことで，14 行目で例外ハンドラへ移動すると共にプロセッサを例外モードからスーパーバイザモードに変更する．この movs 命令は S ビットがセットされていてかつ pc をディスティネーションとしているため，10 行目で設定した spsr が cpsr にコピーされる．なお，Linux において例外モードは例外発生時から例外ハンドラへ移動する間のこのスタブのみに適用される．

3.4 アクセス制御

ARM アーキテクチャはドメインと AP (Access Permission) ビットによる 2 段階のアクセス制御を行っている．

ドメインとは仮想メモリ上のメモリ領域の集合を表すことの出来る機能であり，この機能を利用することにより，共通の仮想メモリマップを共有しているときにメモリの一部分を他の部分から隔離することができる．

ドメインは DACR (Domain Access Control Register) によって管理されている．DACR は各ドメインのドメインタイプを保持しており，それらはアクセス不可・クライアント・マネージャの 3 つに分けられる．アクセス不可はそのドメインへのアクセスを全て遮断し，逆にマネージャはアクセスを全て許可する．一方クライアントはアクセス制御をドメインで行わず，AP ビットに任せる．

AP ビットによるアクセス制御はドメインよりも細かいページサイズでアクセス制御ができる．このアクセス制御は現在のプロセッサモードの特権状態に基づいて行われる．

3.5 センシティブ命令

ARM アーキテクチャに限らずどのアーキテクチャにおいても VMM を構成する際にセンシティブ命令の扱いが非常に重要となる．センシティブ命令とはシステムリソースの状態や動作モードなどに依存する命令のことであり，VMM を構築する際には全てのセンシティブ命令が同時に特権命令でもなければならない[1)]．

本節では ARM アーキテクチャにおいてセンシティブ命令に当たる命令を列挙する．なお，ここで列挙するセンシティブ命令は本研究で用いる ARM アーキテクチャである ARM926EJ-S

© 2010 Information Processing Society of Japan

表 2 psr へのアクセス命令とその内容，ユーザモード時の psr へのアクセス命令の振る舞い

	内容	spsr	cpsr
MRS	psr の値を汎用レジスタに読み出す	未定義例外発生	実行可能
MSR	汎用レジスタの値を psr に書き込む	未定義例外発生	命令を無視

(ARMv5TEJ) の命令から抽出している．

3.5.1 psr へのアクセス命令

psr へのアクセス命令とその内容，ユーザモード時に対象となる psr の種類によって行う振る舞いを表 2 に示す．これらはセンシティブ命令であるにも関わらず，cpsr へのアクセスに関してはユーザモード時に特権命令とは異なる振る舞いをする．そのため，本研究ではこれらの命令を非特権センシティブ命令と呼ぶ．

3.5.2 コプロセッサレジスタへのアクセス命令

コプロセッサはキャッシュやメモリ管理についての制御を行う MMU など，システムリソースに関する重要な情報を管理している．

コプロセッサのレジスタへのアクセス命令として以下の命令が存在する．

- MRC：コプロセッサのレジスタの値を CPU の汎用レジスタに読み出す
- MCR：CPU の汎用レジスタにコプロセッサのレジスタの値を書き込む

これらの命令はユーザモード時に未定義例外を発生させる特権命令である．

3.5.3 ユーザモードレジスタへのアクセス命令

ARM アーキテクチャには特権モード時にユーザモードのレジスタへアクセスすることのできる命令が存在する．これらは特権モード時の使用を想定している命令であり，動作モードに依存した命令である．

ユーザモードレジスタへのアクセス命令として以下の命令が存在する．なお，命令の後の括弧内の数字は同じ命令でも実行時のプロセッサモードやオペランドによってその振る舞いが複数存在していることを示す[7]．

- LDM (2)：指定アドレスからユーザモードのレジスタにロードする
- LDM (3)：プロセッサモードを変更し，指定アドレスからレジスタにロードする
- STM (2)：指定アドレスにユーザモードのレジスタをストアする

これらの命令はユーザモード時に未定義例外を発生する特権命令である．

3.6 デバイス

ARM アーキテクチャはメモリマップド I/O 方式を採用しているため，デバイスが配置されているアドレスに一般的なメモリアクセス命令でアクセスすることで，そのデバイスを使用することができる．ただしデバイス領域は MMU で保護されているために特権モード時でないとアクセスすることができず，非特権モード時にアクセスするとデータアボート例外を発生する．

4. 設計と実装

本章では ARM アーキテクチャ上で動作する VMM の設計内容とその実装方法について述べる．

4.1 設計実装方針

本節では 3 章で述べた ARM アーキテクチャの機能に基づいて ARM アーキテクチャ上で動作する VMM を開発するに当たっての設計実装方針について述べる．

4.1.1 センシティブ命令の検知

VMM がゲスト OS を自身の提供した VM 上で制御するためには，ゲスト OS が実行するセンシティブ命令 (3.5 節) を検知し，それを適切にエミュレーションすることで VM の状態を更新しなければならない．もしゲスト OS のセンシティブ命令を VMM が検知できなければゲスト OS を正常に動作させることができなくなる．

3.5 節より多くのセンシティブ命令は特権命令であるため，本研究では VMM を特権モード，ゲスト OS を非特権モードで動作させることでゲスト OS のセンシティブ命令を例外によって検知し，VMM で適切にエミュレーションする．3.5.1 項で述べた非特権センシティブ命令については 4.2.2 項で述べる代理特権命令で対処する．

4.1.2 仮想プロセッサモードの導入

4.1.1 項でゲスト OS のセンシティブ命令を検知するために特権モードで VMM，非特権モードでゲスト OS を動作させることについて述べた．しかしこの方法では新たな問題が生じてしまう．それは Linux を非特権モードで動作させるため，Linux のカーネルモードとユーザモードを非特権モード上のみで実装させなければならないというものである．

そこで本研究では各プロセッサモードを非特権モード上に仮想的に構築することでこの問題を解決する．本研究ではこれを仮想プロセッサモードと呼ぶ．仮想的なプロセッサモードは実際のプロセッサモードと同様に 7 つ用意し，Linux のカーネルモードは仮想特権モード，ユーザモードは仮想非特権モード上で動作させる．

4.2 センシティブ命令のエミュレーション

本節ではゲスト OS で実行されるセンシティブ命令を VMM で検知して適切にエミュレーションする方法について述べる．

© 2010 Information Processing Society of Japan

4.2.1 特権センシティブ命令のエミュレーション

3.5 節で述べたように，センシティブ命令はその多くが特権命令である．そのため，ゲスト OS を非特権モードで動作させると多くのセンシティブ命令が未定義例外という例外を発生する．

ここで，例として未定義例外を起こしたセンシティブ命令がどのように VMM によってエミュレーションされるかについて具体的に説明する．ゲスト OS で特権センシティブ命令が実行されると未定義例外が発生し，自動的にプロセッサモードが未定義例外モードに遷移，割り込みが禁止された後，未定義例外ベクタに設定されている命令を実行する．未定義例外ベクタには**??**項で示すように VMM 起動時に VMM の未定義例外ハンドラのスタブへの分岐命令あらかじめ設定しておくことで，pc をスタブへ移動させる．

未定義例外ハンドラのスタブを以下に示す．

```
1   _vmm_und_handler_stub:
2           push    {r0, r1, r2, r3, r11, r12, r14}
3           mov     r0, sp
4           bl      vmm_und_handler
5           pop     {r0, r1, r2, r3, r11, r12, r14}
6           movs    pc, r14
```

3.2 項で述べたようにレジスタ r0〜r3，r12 は汎用スクラッチレジスタであるため，C 言語の関数で実装されている VMM の例外ハンドラではレジスタの値が破壊される恐れがある．そのため，2 行目で push 命令を使ってそれらのレジスタの値をスタックに保存する．この保存した汎用スクラッチレジスタは VMM 内でセンシティブ命令のエミュレーションを行う際に参照したり，エミュレーションした結果を反映させるために更新される可能性がある．そのため，スタックに保存した複数のレジスタを VMM の例外ハンドラでスタックフレーム構造体として管理し，そのポインタを VMM の例外ハンドラへ引数として渡す．これにより VMM の例外ハンドラ内でスタックに保存したレジスタに容易にアクセスすることが可能となる．ATPCS で定められた引数の受け渡し方法によると関数に移動したときの r0 の値がその関数の引数となるため，3 行目で sp の値，すなわちスタックフレーム構造体のポインタを代入し，4 行目で VMM の未定義例外ハンドラへ分岐する．VMM の未定義例外ハンドラでの処理が完了すると再びスタブへ戻り，5 行目で保存したレジスタを復帰した後，6 行目で例外を発生させたゲスト OS のアドレスに復帰する．

VMM の未定義例外ハンドラでは，まず未定義例外を発生させた命令のアドレスとその命令を取得する．次に 4.3.3 項で述べるゲスト OS の例外処理との分岐や 4.3.1 項で述べる仮想バンクレジスタ操作との分岐，4.2.2 項で述べる代理特権命令との分岐を経て，最終的に未定義例外を発生させたセンシティブ命令が VMM の特権内で実行され，エミュレーションが完了する．

本研究において VMM は 1 つの VM 環境を提供することを前提とするため，VMM の未定義例外ハンドラではゲスト OS が権限不足で実行できなかったセンシティブ命令を実行権限を持った VMM の特権内で再度実行することでエミュレーションすることができる．

VMM 内での未定義例外を発生させたセンシティブ命令の実行は以下のコードを通して行われる．なお，コード実行時に r0 には未定義例外を発生させたセンシティブ命令が格納されている．

```
1   __asm__ __volatile__(
2           "ldr r2, =0xFFF90000\n\t"
3           "str r0, [r2], #4\n\t"
4           "ldr r0, =0xE1A0F00E\n\t"
5           "str r0, [r2], #-4\n\t"
6           "push {r14}\n\t"
7   );
8   __asm__ __volatile__(
9           "blx r2\n\t"
10          "pop {r14}\n\t"
11  );
```

まず 2 行目で VMM 内の領域に連続した 32 ビットのワード領域を 2 つ (0xFFF90000, 0xFFF90004) 用意し，3 行目で先頭のワードに未定義例外を発生させたセンシティブ命令，5 行目で次のワードに

```
mov     pc, lr
```

という命令を配置する．この命令をビット列で表すと 4 行目の 0xE1A0F00E にあたる．これはリターンアドレスを格納する lr レジスタをプログラムカウンタである pc に代入することで lr のアドレスへ移動するという mov 命令で，主に関数呼び出しや例外からの復帰に利用される．エミュレーションの際は以上のように命令を配置したアドレスへ 9 行目の blx 命令で分岐する．blx 命令は lr に復帰アドレスをセットして分岐を開始する命令であるため，センシティブ命令を実行した後の mov 命令で分岐先から復帰することができる．しかしその際に今までの lr が上書きされてしまうため，6 行目で lr を push することで一時的に保存し，10 行目で pop により復帰している．これにより VMM 内で未定義例外を発生させたセンシティブ命令を実行させることができる．

© 2010 Information Processing Society of Japan

なお，未定義例外を発生させたセンシティブ命令が汎用スクラッチレジスタをオペランドとして利用していた場合，レジスタの値を書き込むセンシティブ命令の場合は VMM の特権でその命令を実行する前にスタックフレーム構造体から使用するレジスタを復帰させ，命令の実行に影響するコンテキストを復元する必要がある．そのような処理はセンシティブ命令を実行する直前に行わなくてはならないため，7 行目と 8 行目の間で実行する．逆に，レジスタに値を読み込むセンシティブ命令の場合は VMM の特権でその命令を実行した後に使用したレジスタをスタックフレーム構造体の該当するレジスタの領域に上書きする必要がある．これは 11 行目以降で実行する．

以上により，特権センシティブ命令を適切にエミュレーションすることができる．

4.2.2 非特権センシティブ命令のエミュレーション

4.2.1 項ではセンシティブ命令が特権命令である場合のエミュレーションについて述べたが，3.5.1 項で述べた MRS，MSR 命令はユーザモード時に例外を発生させない非特権命令である．このようなセンシティブ命令はその実行を VMM が検知する手段がないため，ゲスト OS を正常に動作させることができない．

そこで本研究ではゲスト OS である Linux のソースコード中の MRS，MSR 命令を適当な特権命令に静的に書き換えることで故意に例外を発生させてこれらの命令のゲスト OS による実行を VMM が検知できるようにした．故意に例外を発生させるこのような命令を本研究では代理特権命令と呼ぶ．代理特権命令には，以下のように Linux 中で使用されていないオペランドを持つ特権命令を用いる．

```
mrc     p15, 2, r2, c2, c2, 2
```

VMM ではこれらの代理特権命令と非特権センシティブ命令を関連付けておくことが必要である．

この代理特権命令によりゲスト OS による MRS，MSR 命令の実行が未定義例外によって検知され，VMM の未定義例外ハンドラ内でエミュレーションを行うことができる．エミュレーションの方法は 4.2.1 項と VMM の未定義例外ハンドラへ処理が移行するところまでは同様である．VMM の未定義例外ハンドラではあらかじめ代理特権命令の処理を個々に記述しておき，以下のように分岐させている．

```
 1   if (inst == 0xEE522F52) { // mrc p15,2,r2,c2,c2,2
 2           /*
 3            * arch/arm/kernel/head.S
 4            * msr cpsr_c, #PSR_F_BIT | PSR_I_BIT | SVC_MODE
 5            */
 6           __asm__ __volatile__(
 7                   "mrs r0, spsr\n\t"
 8                   "orr r0, #0xC0\n\t" // PSR_F_BIT | PSR_I_BIT
 9                   "msr spsr_c, r0\n\t"
10           );
11   }
```

この代理特権命令は 4 行目の msr 命令を置き換えたものであり，この msr 命令は現在のプロセッサモードをスーパーバイザモードにしようとしている．しかし，本研究においてゲスト OS は非特権モードで動作するため，特権を持つスーパーバイザモードになることは許されない．そこで 8 行目でスーパーバイザモードにならないように SVC_MODE を OFF にした値を r0 に代入し，それを msr 命令で書き込んでいる．

以上により非特権センシティブ命令を代理特権命令により適切にエミュレーションすることができた．

4.3 仮想プロセッサモード

本節では 4.1.2 項で述べた仮想プロセッサモードの実装方法について述べる．

4.3.1 仮想バンクレジスタ

仮想プロセッサモードは実際のプロセッサモードと同様にバンクレジスタを持たなければならない．しかしレジスタは限られた数しかないため，仮想プロセッサ用に使用できる余剰レジスタは存在しない．

そこで本研究ではバンクレジスタ用の領域として VMM のメモリ領域を利用し，これを仮想バンクレジスタとする．この仮想バンクレジスタは仮想ユーザ及び仮想システムモードが使用するレジスタについても切り替わる可能性のある仮想例外モードに仮想バンクレジスタが存在するレジスタについては仮想レジスタという呼称で同様に領域を用意する．そしてゲスト OS がプロセッサモードの変更をする命令を実行したときに仮想レジスタに現在のレジスタを保存し，現在のレジスタに仮想バンクレジスタの値を代入する．これによって現在のレジスタをバンクレジスタのように使用することが可能になる．

また，実際のプロセッサモードでは例外発生時に例外発生前の cpsr の値が発生した例外モードの spsr に自動的に保存されるため，仮想プロセッサモードでも例外発生時に例外発生前の cpsr の値を発生した例外に対応する仮想例外モードの spsr に保存する．

4.3.2 ドメインによる仮想保護レベル

実際のプロセッサモードと同じように仮想プロセッサモードでも仮想特権モード時は全メモリにアクセスできるようにし，仮想非特権モード時は仮想特権モード上で動作するカーネ

© 2010 Information Processing Society of Japan

表 3 ドメインを利用した仮想プロセッサモードのアクセス制御

	D15	D1	D0
VMM	クライアント	マネージャ	マネージャ
ユーザモード	クライアント	クライアント	アクセス不可
カーネルモード	クライアント	マネージャ	マネージャ

ルプロセスにアクセスできないようにしなければならない．更に VMM は実際のプロセッサモードの特権モード上で動作しているため，非特権モードで動作するゲスト OS からのアクセスを受けないという状況も維持しなければならない．

そこで本研究では 3.4 節で述べたドメインを表 3 のように使用することで，非特権モードに仮想保護レベルを構築する．ここで D0，D1，D15 はそれぞれカーネルプロセスが属するドメイン，ユーザプロセスが属するドメイン，VMM が属するドメインを表している．なお，VMM の属する PTE の AP ビットフィールドはあらかじめ “01” に設定され，ユーザモードからのアクセスができないようにしておかなければならないものとする．そのようにしなければゲスト OS で例外が発生したときに VMM にアクセスすることができなくなってしまう．したがって VMM が属する D15 は常にクライアントに設定しておき，アクセス制御は常に AP ビットフィールドを使用するようにすることで，非特権モードで動作するゲスト OS から VMM を保護する．

表 3 が正確にアクセス制御を実現しているかどうか確認する．まず VMM 時は全てのドメインに対してアクセスができるように D0，D1 ともにマネージャとした．次にユーザモード時はカーネルプロセスの属する領域である D0 に対してアクセス不可とした．ユーザプロセスの属する領域である D1 に対してはクライアントに設定することで AP ビットフィールドによるアクセス制御を使用する．これは VMM を導入しない通常の Linux でも同様のアクセス制御方法になっている．最後にカーネルモード時は D0 と D1 を両方ともマネージャに設定することでカーネルプロセスの属する領域にもユーザプロセスの属する領域にもアクセスができるようにした．

以上のようにドメインを利用することにより非特権モード上に仮想保護レベルを構築し，ゲスト OS として Linux を動作させることが可能となる．

4.3.3 ゲスト OS の例外処理

本研究でゲスト OS として用いる Linux は例外処理を行うが，ARM アーキテクチャで利用することのできる例外ベクタは 1 つだけである．例外ベクタには既に**??**項で示したように VMM の例外ハンドラのスタブへの分岐命令が設定されているため，ゲスト OS で例外が発生した場合は VMM の例外ハンドラへ移動する．

VMM の例外ハンドラでは，ゲスト OS のエミュレーション処理とゲスト OS の例外処理が重なってしまう例外が存在する．例えば未定義例外は 4.2 節で述べたようにゲスト OS のセンシティブ命令をエミュレーションするために未定義例外を利用し，またデータアボート例外は 4.4 節で述べるようにゲスト OS のデバイスをエミュレーションするために用いられる．このような場合，VMM の例外ハンドラでは現在の仮想プロセッサモードの状態や例外発生アドレスを参考にして発生した例外がゲスト OS の例外処理によるものかどうかを判断する．

VMM ではゲスト OS の例外処理を 3.3.1 項で述べた Linux の例外ハンドラのスタブへ任せる．スタブでは 4.3.1 項で述べた仮想バンクレジスタを使用する．

4.4 デバイスのエミュレーション

本研究ではゲスト OS を非特権モードで動作させるため，デバイスへのアクセスの際に権限不足によるデータアボート例外が発生する．VMM はこの例外を VMM のデータアボート例外ハンドラで適切に処理し，デバイスをエミュレーションする必要がある．本研究では 1 つの VM 環境を提供することを前提としているため，ゲスト OS が実行したデバイスへのアクセスを VMM で再度実行することでデバイスのエミュレーションを行うものとする．

デバイスのエミュレーションは 4.2.1 項で述べたエミュレーションの方法とほぼ同じであり，その差異は例外の種類が未定義例外でなくデータアボート例外であることのみであり，データアボート例外用に用意された VMM の例外ハンドラとそのスタブを使用する．

5. 実験・評価

本章では 4 章で示した設計実装方針に基づいて構築した VMM 上で行った実験と，その評価について示す．

5.1 実 験 環 境

本システムの実験環境は以下の通りである．なお，これは QEMU のフルシステムエミュレーションによってエミュレートされた環境である．

- OS ：Linux 2.6.25.4
- ボード ：Integrator/CP
- CPU ：ARM926EJ-S（ARMv5TEJ）
- メモリ ：128MB

ⓒ 2010 Information Processing Society of Japan

表 4 各ベンチマークの実行結果

Benchmark	NativeLinux (usec)	VMM (usec)	速度比
fork+exit	3,255.97	86,488.48	26.56
fork+exec	14,917.65	274,880.32	18.43
pipe	210.02	1,001.09	4.77
syscall	0.87	7.21	8.29

5.2 実装量評価

4 章で述べた設計実装方針に基づいて VMM を構築した結果，C 言語で 2662 行，インラインアセンブラを含めたアセンブリ言語で 792 行の実装量となった．また，4.2.2 項で述べた代理特権命令は MRS 命令 20 箇所，MSR 命令 24 箇所の合計 44 行であり，Linux カーネルへの変更は非常に少ないものとなっている．

なお，本研究では zImage に VMM を付与してブートするという方法を採っているため，これとは別に zImage の解凍処理に 70 行追加している．

5.3 性能評価

QEMU の icount オプションを用いて各命令の実行時間を一定にした状態で，本研究で開発した VMM と仮想マシンモニタを導入していない Native な Linux 上で簡単なベンチマークプログラムを実行した．その実行速度を比較した結果を表 4 に示す．なお速度比は Native Linux の実行速度を 1 としたときの VMM の実行速度を表している．

表 4 によると，fork+exit や fork+exec のように fork を使用して新しいプロセスを生成するベンチマークの速度比が他のベンチマークに比べて大きくなっている．これはプロセス生成時に PT を更新する際，PTE に対して多くの特権命令が実行され，VMM が行わなければならないエミュレーション回数が増加してしまったためであると思われる．

6. おわりに

本研究では ARM アーキテクチャ用 VMM の設計・実装方法を提案し，それに基づいて開発した VMM の評価を行った．ARM アーキテクチャの特権モードで VMM，非特権モードでゲスト OS を動作させることでゲスト OS の実行するセンシティブ命令を VMM により例外として検知し，それを適切にエミュレーションすることができた．一部のセンシティブ命令は特権命令でないために，ゲスト OS によるそれらの命令の実行を VMM で検知することができないという問題が生じたが，本研究ではそれらの命令を故意に例外を発生させる代理特権命令で静的に置き換えることにより，ゲスト OS による非特権センシティブ命令の実行を検知することを可能にした．また，本研究でゲスト OS として用いた Linux を非特権モードで動作させることにより生じるカーネルモードとユーザモード間のアクセス制御の欠如に関しては，非特権モード時に仮想的なプロセッサモードを提供することで解決した．バンクレジスタは VMM 内にそれらの領域を用意して仮想バンクレジスタを実現し，またゲスト OS である Linux のアクセス制御はドメインを用いることで実現することができた．

今後の課題として，現在 QEMU 上で動作している本システムを実機上に移植し，Native Linux 環境や XenARM との性能の比較を実施することを目標に開発を続ける．更に，その比較によって本研究で開発した VMM の問題箇所を抽出し，それを元に性能の改善を目指すと共に，複数 VM を提供するといった VMM の拡張を施していく．また，現在非特権センシティブ命令の代理特権命令への書き換えを手動で行っているが，修正箇所が多岐にわたるため静的なコード解析による代理特権命令の置き換え[8] により自動化するといった方法も導入していく．

参考文献

1) R. P. Goldberg: Survey of Virtual Machine Research, IEEE Computer, Vol.7, No.6, pp. 34-45 (1974)
2) Febrice Bellard: QEMU, a Fast and Portable Dynamic Translator, USENIX 2005, pp. 41-46 (2005)
3) Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield: Xen and the Art of Virtualization. SOSP 2003, pp. 164-177 (2003)
4) Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, Chul-Ryun Kim: Xen on ARM: System Virtualization using Xen Hypervisor for ARM-based Secure Mobile Phones, CCNC 2008, pp. 257-261 (2008)
5) Alves, T. and Felton, D. TrustZone: Integrated Hardware and Software Security, ARM (2004)
6) Andrew N. Sloss, Dominic Symes, Chris Wright: Arm System Developer's Guide Designed and Optimizing System Software, Morgan Kaufmann Pub (2004)
7) David Seal: Architecture Reference Manual Second Edition, Addison-Wesley Professional (2000)
8) Eiraku, H. and Shinjo, Y. Running BSD Kernels as User Processes by Partial Emulation and Rewriting of Machine Instructions, USENIX BSDCon 2003 Conference (BSDCon'03) (September 2003).

© 2010 Information Processing Society of Japan