Vol.2009-ARC-183 No.12
Vol.2009-OS-111 No.12
2009/4/23

ARMアーキテクチャ用仮想マシンモニタの開発

鈴 木 章 浩[†1]　　追 川 修 一[†1]

今日，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 を開発する．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．本研究は現在ゲスト OS のユーザプロセスの生成まで実装が完了している．

Development 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 develop the 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. The current status is that the guest OS can create user process.

†1 筑波大学
University of Tsukuba

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してしまう恐れがある．

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 を利用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，例として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 アーキテクチャに主眼を置いた 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であろうと考えられる．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．

本論文では ARM アーキテクチャ上で動作する VMM の基本設計と実装について述べる．提供する VM 環境は 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 ボートをエミュレートする QEMU[2] を利用する．本研究では特権モードで VMM，非特権モードでゲスト OS を動作させ，ゲスト OS の実行するセンシティブ命令を例外によって VMM で検知する．そのためゲスト OS である Linux はカーネルモードとユーザモード間のアクセス制御を非特権モード上で実装しなければならない．そこで本研究では ARM アーキテクチャのプロセッサモードを仮想化し，アクセス制御にドメインを利用することにより，Linux を非特権モード上で動作させることを可能にした．また，センシティブ命令であるのにも関わらず特権命令でない命令については故意に例外を発生させる代理特権命令によってその命令を静的に書き換え，VMM で検知することができるようにした．

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

© 2009 Information Processing Society of Japan

Vol.2009-ARC-183 No.12
Vol.2009-OS-111 No.12
2009/4/23

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 章では本研究の現時点での進捗状況について述べる．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ない．そのため，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．

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表 1 に示すような 6 つの特権モードと 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．ユーザモードは非特権であるため，実行中のプログラムが保護されているシステムリソースにアクセスしたり，現在のプロセッサモードを変更したりするためには 3.3 節で述べる例外を発生させる必要がある．ユーザモード以外のモードは特権を持ち，システムリソースにアクセスしたり現在のプロセッサモードを自由に変更することができる．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．

表 1　プロセッサモード

プロセッサモード	略称	特権	説明
ユーザ	usr	なし	通常のプログラム実行モード
FIQ	fiq	あり	高速なデータ転送又はチャネルプロセスに対応
IRQ	irq	あり	汎用割り込み処理に使用
スーパバイザ	svc	あり	オペレーティングシステム用の保護モード
アボート	abt	あり	仮想メモリ，メモリ保護，又は両方を実装
未定義	und	あり	ハードウェアコプロセッサのソフトウェアエミュレーションに使用
システム	sys	あり	特権オペレーティングシステムタスクを実行

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 個のプログラムステータスレジスタ (psr) 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るが，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，関数が破壊できる汎用スクラッチレジスタとなる．レジスタ r13 はスタックポインタ (sp) として使用され，スタックの先頭を指す．レジスタ r14 はリンクレジスタ (lr) として使用され，サブ

© 2009 Information Processing Society of Japan

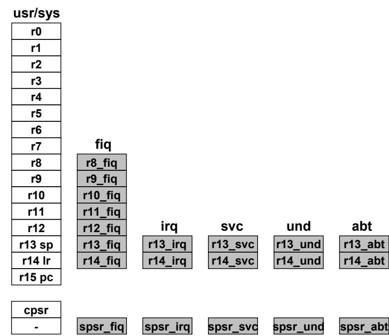

図 1 ARM アーキテクチャのレジスタ群

ルーチンを呼び出す際に復帰アドレスを格納する．レジスタ r15 はプログラムカウンタ (pc) として使用され，プロセッサが次にフェッチする命令のアドレスを保持する．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．

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う．表 2 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，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 に移動され，復帰アドレスを格納している r14 の値が pc にコピーされる．

表 2 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 (命令フェッチ中に発生するメモリアボート)	アボート	0xFFFF000C
データアボート (データアクセス中に発生するメモリアボート)	アボート	0xFFFF0010
IRQ (割り込み)	IRQ	0xFFFF0018
FIQ (高速割り込み)	FIQ	0xFFFF001C

3.4 ページング

ARM アーキテクチャは他のアーキテクチャと同様に MMU (Memory Management Unit) を用いることで仮想メモリ機能を提供する．また，ARM MMU のページテーブル (PT) にはレベル 1 (L1) とレベル 2 (L2) が用意されている．これにより ARM MMU は複数のページサイズを提供することができる．L1PT のページテーブルエントリ (PTE) にはドメインビットフィールドと呼ばれるビットフィールドが存在し，各ページの PTE には AP (Access Permission) ビットフィールドと呼ばれるビットフィールドが存在する．ARM MMU はこれらのビットフィールドを使用することで二つの異なる制御方式での，ページに属するメモリ領域へのアクセス制御を行っている．最初にドメインによるアクセス制御を行い，次に AP ビットフィールドを使ってのアクセス制御を行う．前者のドメインによるアクセス制御に関しては 3.5 節で述べる．後者のアクセス制御に関しては表 3 のように行われる．

表 3 アクセス許可と制御ビット

特権モード	ユーザモード	AP ビットフィールド
読み出し/書き込み	読み出し/書き込み	11
読み出し/書き込み	読み出し専用	10
読み出し/書き込み	アクセス不可	01

3.5 ド メ イ ン

ARM アーキテクチャにはドメインと呼ばれる，仮想メモリ上のメモリ領域の集合を表すことのできる機能が存在する．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) によって管理される．DACR に

© 2009 Information Processing Society of Japan

は 16 個のドメイン用にビットフィールドが用意されており，このビットフィールドに表 4 に示す値を設定することで，各ドメイン毎に固有のアクセス制御を提供することができる．

表 4 ドメインのアクセスタイプ

アクセスタイプ	値	説明
アクセス不可	00	全てのアクセスがドメインフォルトを生成
クライアント	01	TLB エントリのアクセス制御ビットに対してアクセスがチェックされる
マネージャ	11	アクセスチェックは行われず，アクセス許可フォルトは発生しない

3.4 節で述べたページは必ずいずれかのドメインに属しており，ページに属するメモリ領域へのアクセスに対して最初にドメインによるアクセス制御が行われ，ドメインがクライアントに設定されている場合には 3.4 節で述べた AP ビットフィールドによるアクセス制御が行われる．

3.6 センシティブ命令

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 を構築する際にこの命令の扱いが非常に重要になる．そのため本節では 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 (ARMv5TEJ) の命令から抽出している．

3.6.1 psr へのアクセス命令

psr はプロセッサの内部状態を管理するレジスタであり，このレジスタにアクセスすることで cpsr を変更して現在のプロセッサモードを変更したり，割り込みを禁止したりすることができる等，システムリソースに関する重要な情報を扱っている．

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 5 に示す．

表 5 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

3.6.2 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 等，システムリソースに関する重要な情報を扱っている．

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する特権命令である．

3.6.3 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

・LDM (2) : 指定された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，指定されたアドレスからレジスタにロードする

・STM (2) : 指定された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．

3.7 デバイス

特権モード時に本研究で使用するボートである Integrator/CP のデバイスが配置されているアドレスにメモリ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．

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

本節では第 3 章で述べた ARM アーキテクチャの機能を利用して 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 (3.6 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．

3.6 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

© 2009 Information Processing Society of Japan

よって検知し，VMM で適切にエミュレーションする．3.6.1 項で述べた特権命令でない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 節でゲスト 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．しかしこの方法には新たな問題が生じる．それは本研究でゲスト OS として使用する Linux のカーネルモードとユーザモードに関するものである．

本研究では 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．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．

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．

4.2.1 特権命令であるセンシティブ命令のエミュレーション

4.1.1 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，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，例としてレジスタ r0 の値に従ってキャッシュ全体をフラッシュする命令

```
mcr     p15, 0, r0, c7, c7, 0
```

と，ARM プロセッサの ID をレジスタ r3 に取得する命令

```
mrc     p15, 0, r3, c0, c0, 0
```

の 2 種類の命令について詳述する．

ゲスト OS でこれらの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 4.2.3 項によりあらかじめ VMM の未定義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，まずそこに pc が移動する．VMM の未定義例外ハンドラのスタブでは例外発生前のゲスト OS の汎用スクラッチレジスタやゲスト OS への復帰アドレスを保存し，VMM の例外ハンドラでの処理が完了した後に，保存したこれらのレジスタを復帰してゲスト OS の例外発生箇所へ戻る．3.2 節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，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する．このスタックフレーム構造体へのポインタを VMM の例外ハンドラへ引数として渡す．こうすることで 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 の値が引数となるため，レジスタ r0 にスタックフレーム構造体のアドレスを代入し，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，保存したレジスタを復帰した後，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は以下の C 言語のプログラムソースから構成される．なお，ここでは本項で詳述する命令のエミュレーション箇所のみをソースコードから抜粋して記述する．

```
 1  void vmm_und_handler(stack_frame *registers)
 2  {
 3          unsigned int address = registers->r14;
 4          unsigned int *inst = 0;
 5          unsigned int cause_inst;
 6
 7          address -= 4;
 8          inst = (unsigned int *)address;
 9          cause_inst = (unsigned int)*inst;
10
11          switch (cause_inst) {
12          case 0xEE070F17:
13                  __asm__ __volatile__(
14                          "ldr r0, [sp, #4]\n\t"
15                          "ldr r0, [r0]\n\t"
16                          "mcr p15, 0, r0, c7, c7, 0\n\t"
17                  );
18                  break;
```


© 2009 Information Processing Society of Japan

```
19          case 0xEE103F10:
20                  __asm__ __volatile__(
21                          "mrc p15, 0, r3, c0, c0, 0\n\t"
22                          "ldr r0, [sp, #4]\n\t"
23                          "str r3, [r0, #12]\n\t"
24                  );
25                  break;
26          }
27  }
```

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状態で再度実行することでエミュレーションする．

まず 7 行目で未定義例外を発生させた命令がどのアドレスに配置されているかを例外時のゲスト OS への復帰アドレスから計算し，9 行目でそのアドレスから未定義例外を発生させたセンシティブ命令を取得する．

11〜26 行目で取得したセンシティブ命令毎に処理を分ける．12〜18 行目がレジスタ r0 の値に従ってキャッシュ全体をフラッシュする命令，19〜25 行目が ARM プロセッサの ID をレジスタ r3 に取得する命令のエミュレーションを担当する．

まず前者の命令のエミュレーションについて詳述する．この命令はレジスタ r0 の値を使用するが，r0 は汎用スクラッチレジスタであるため，保護のためにスタックフレーム構造体として保存され，VMM の未定義例外ハンドラに引数として渡されている．レジスタ r0 を例外発生前の状態に復元するために 14 行目で sp+4 の位置に保存されている引数を取得してスタックフレーム構造体を指すポインタを取得し，これを元に 15 行目でレジスタ r0 を復元する．以上の操作により例外を発生させた時点と同じ状態にレジスタを復元することができ，その命令の実行に影響するコンテキストが復元されたことになる．そのため，この状態で 16 行目より例外を発生させたセンシティブ命令を実行することでエミュレーションを行うことができる．

次に後者の命令のエミュレーションについて詳述する．この命令は値を読み出す命令であるため，前述した命令実行前のレジスタの復元については行う必要がない．そのため 21 行目で例外を発生させたセンシティブ命令をまず実行する．この命令によりレジスタ r3 に変更が加えられたため，これを 22，23 行目の操作によってスタックフレーム構造体中の保存した r3 のレジスタ値にも反映させる．この操作を施さないと VMM の未定義例外ハンドラを抜けてスタックフレーム構造体中に保存したレジスタを復元する際に更新前の値を復元することになってしまい，適切にエミュレーションすることができなくなってしまう．

以上により 2 種類のセンシティブ命令が適切にエミュレーションされた．

4.2.2 特権命令でない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，3.6.1 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．

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 項と同様である．

4.2.3 例外ベクタへの分岐命令の設定

ゲスト OS で発生するセンシティブ命令を VMM で例外という形で検知するには，4.2.1 項で述べたように，例外ベクタに VMM の例外ハンドラのスタブへの分岐命令を設定しなければならない．システム起動後，ゲスト OS の起動処理を開始する前に VMM はこの設定を行わなければならない．

VMM にはあらかじめ例外の種類ごとに例外ハンドラとそのスタブを用意する．そして，以下に示すアセンブリ言語によって VMM の例外ハンドラのスタブへの分岐命令を設定する．例として VMM の未定義例外ハンドラのスタブへの分岐命令を例外ベクタに設定する方法について述べる．

```
1  __asm__ __volatile__(
2          "ldr r0, =_vmm_und_handler_stub\n\t"
3          "ldr r1, =0xFFFF0004\n\t"
4          "ldr r2, =0xEAFFFFFE\n\t"
5          "mov r3, r1\n\t"
6          "sub r3, r0\n\t"
7          "sub r2, r3, lsr #2\n\t"
8          "str r2, [r1]\n\t"
9  );
```

2 行目で VMM の未定義例外ハンドラのスタブのアドレス，3 行目で未定義例外ベクタア

© 2009 Information Processing Society of Japan

ドレス，4 行目で現在のアドレスに分岐する命令を格納する．6 行目で未定義例外ベクタアドレスから VMM の未定義例外ハンドラのスタブへのアドレスを減算することで 2 つのアドレス間の相対アドレスを求め，7 行目で現在のアドレスに分岐する命令から 6 行目で求めた相対アドレスを引き，相対アドレス分低いアドレスに分岐する命令が作られる．この命令を 8 行目で未定義例外ベクタアドレスに書き込むことで，未定義例外ベクタから VMM の未定義例外ハンドラのスタブへ分岐する命令を設定することができる．これを残り 6 つの例外にも同様に行うことで，例外ベクタへ VMM の例外ハンドラのスタブへの分岐命令を設定することができる．

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．

そこで本研究では仮想バンクレジスタ用の領域として VMM のメモリ領域を利用する．この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 アクセス制御

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 3.5 節で述べたドメインを表 6 のように使用することで，このアクセス制御を実現する．

表 6 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	マネージャ
カーネルモード	クライアント	マネージャ	マネージャ

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．

表 6 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 を付加しない通常の 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．

以上によりドメインを利用することでユーザモードで動作するゲスト OS としての Linux にカーネルモードとユーザモードの 2 つのモードのアクセス制御を提供することが可能である．

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，

© 2009 Information Processing Society of Japan

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．

4.5 物理アドレスでの VMM の保護

QEMU のダイレクトブート時にカーネルオプションとして “mem=112M” を指定することにより，ゲスト OS は物理メモリを 112MB までしか使用することができなくなる．本研究では物理メモリの容量として 128MB を使用するため，112〜128MB の領域はゲスト OS から保護されることとなる．この領域に VMM を配置することで物理メモリ上で VMM をゲスト OS から保護することを可能にする．

5. 実　験

本章では前章に示した設計実装方針に基づいて構築した VMM を実行し，現時点で実装が完了している箇所までの結果を示す．

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

5.2 実装量評価

前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 4000 行の実装量となった．なおそのうち 2500 行はインラインアセンブラによるコードである．また，4.2.2 項で述べた代理特権命令は MRS 命令 18 箇所，MSR 命令 42 箇所であり，Linux カーネルへの変更は 60 行である．

5.3 現　状

開発の現状は，最初のユーザプロセスである init プロセスの生成し，ユーザレベルで実行中にエラーが発生してしまっている状態である．

6. おわりに

本研究では ARM アーキテクチャ用 VMM の実装の方法を提案し，その開発を行った．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で，ゲスト OS による特権命令でない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．仮想プロセッサモード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．

以上の方法で実装した VMM は現在ユーザプロセスの生成を行っている段階まで動作している．今後の課題として，ゲスト OS がログインプロンプトを表示してユーザの入力を待つことのできる状態まで正常に動作させ，ゲスト OS 内でベンチマークスイートである lmbench を用いてネイティブ Linux 環境やその他の ARM アーキテクチャ用 VMM との性能の比較を実施することを目標に開発を続ける．また，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．

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)

ⓒ 2009 Information Processing Society of Japan