社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2007－ARC－171 (5)
2007－EMB－ 3 (5)
2007／1／22

2レベル保護リング環境でのVMM構築

青柳 信吾　追川 修一
筑波大学

小規模組込み機器用のCPUの多くはカーネル用とユーザプロセス用の2レベルの保護リングしか持たない．しかし，現在研究されているVMMは3レベルの保護リング環境用のものが多く，組込み機器に移植することが困難である．本論文では2レベルの保護リング環境における，ページング機構を利用した仮想保護リングによる，仮想マシン環境の構築について述べる．

Building VMM on 2-Level Ring Architecture

Shingo Aoyagi　Shuichi Oikawa
University of Tsukuba

Most of the currently available embedded processors provide only two protection levels, privileged and unprivileged, at most. On the other hand, the researches and developments of VMMs are currently conducted mainly using the Intel IA-32 processsor architecture, which provides four protection levels; thus, porting such VMMs to embedded processors requires significant effort and costs. This paper describes our VMM that requires only two protection levels but does not sacrifice the protection among the VMM, the guest OS kernel, and its user processes by "virtual protection ring".

1　はじめに

今日，組込み機器のハードウェアは非常に複雑化し，これまでワークステーションなど高機能な環境でしか実行できなかった複雑なプログラムなどが組込み機器上で実行できるようになった．しかし一方で，組込み機器は生活，社会の中に存在するため，このようなシステムがソフトウェアのバグなどにより異常動作を起こした場合にはワークステーションの暴走以上に脅威となる．そこで近年では，これまで行ってきたハードウェア上で直接OSを実行する環境からハードウェア上に仮想マシンを構築し仮想マシン内でOSを実行する環境へ変化している．このような環境ではアプリケーションやOSなどのソフトウェア資源が暴走した場合でもハードウェアに危機的な影響が出る前に仮想マシンにより処理を中断することができ，安全性を高める手法として広い分野での利用が求められている．

現在実用化されている仮想マシンモニタ (VMM) は，Intel IA-32など複雑なプロセッサ上で実行するものが多く，このようなVMMは，プロセッサがVMMを実行するために十分な保護リング機能が提供しているものとして設計，実装されている．しかし，通常，小規模な組込み機器用のプロセッサでは，OSカーネルとユーザプロセスを実行するのに必要最低限の保護リングのみしか提供しておらず，2レベルの保護リングであることが多い．そのためIA-32などの上に構築された多数の保護リングを使用する仮想マシンは，小規模な組込み機器環境への移植は困難であり，移植するためにプロセッサの保護リング機能を変更するなど多額の開発コストがかかる．

そこで本研究は，2レベルの保護リング環境に仮想マシン環境を構築し，保護リングレベルが限定されている場合でのVMMのソフトウェアアーキテクチャを提案する．本研究のVMMアーキテクチャにより，これまでにPC用として設計された仮想マシンを小規模な組込み機器環境へ移植することを可能にする．

本研究では実験環境としてIA-32アーキテクチャのCPUの環境を用意した．IA-32アーキテクチャでは4レベルの保護リングを提供する．本研究は2レベルの保護リング環境でのVMM実行を目的としているため，IA-32が提供する保護リングのうち2個のリングのみを使う．またこの環境では，Xenなど多くの3レベル保護リングを用いるVMMが実現されており，これらと性能を比較することでリング数の違いによる性能への影響を考察することも目的とする．ゲストOSとしては，組込み機器で広く使われているLinuxを選択した．

本研究では2レベルの保護リング環境用のVMMを構築する手法として，ページング機能を利用した仮想保護リング機構を作成した．仮想保護リングは非特権モードで実行するゲストOSカーネルとユーザプロセス間の保護を提供する機能である．仮想保護リング機構により，本研究では2レベルの保護リング環境内でVMMを構築し，VMM上にてLinux Kernel 2.6.17とユーザプロセスとしてBusyboxを実行できることを確認した．

2　アーキテクチャ

2.1　IA-32アーキテクチャ

これまでに研究されているVMMは，IA-32環境用に設計されたものが多い．今後組込み機器にVMM

を移植する場合にはIA-32上で実行されているVMMに変更を加えて実現することが予想される．

IA-32アーキテクチャとは，Intel Pentiumシリーズ用のCPUアーキテクチャである[4]．IA-32 CPUはシステム上に複雑なOSを実行させることを目標としており，4レベルの保護リング，ページング機構，セグメント機構，割込み機構などを提供している．

IA-32ではCPUが発行する命令のうち，システム全体の挙動に影響を及ぼす命令を特権命令と呼び，その他の命令を非特権命令と呼ぶ．CPUの実行時には，特権処理を行えるかどうかの指定が可能で，この状態をモードと呼ぶ．保護リング機構はモードと関連し，最も特権の高いリング(リング0)ではCPUの特権状態やハードウェア全体の操作が可能である．これ以外の状態を非特権モードと呼ぶ．非特権モードでは特権命令の発行ができないものの，システム全体に影響するレジスタの一部を読み出すことが可能である．

IA-32 CPUで割込みが発生すると，発生要因ごとにあらかじめ設定した処理が呼び出される．割込み要因と呼び出される関数の対応を指定する配列を割込みベクタと呼ぶ．割込みベクタには予約済みのベクタ(0〜19)とユーザ定義ベクタ(32〜255)がある．予約済みのベクタには，0による除算やページフォルト，一般保護例外などマップされており，ユーザ定義ベクタにはタイマ割込みやソフトウェア割込みを使ったシステムコールなどを割り込みコントローラ(PIC)によって配置することができる．

またIA-32ではCPUによってCPU独自の命令を提供しているものもある．最近のLinuxではこれらCPU依存の命令を利用することも多く，VMM設計の際にはこれらの命令を仮想化する必要がある．

2.2 Linux

本研究では，構築したVMM環境の実用性評価のため，VMM上にゲストOSを実行することでテストを行う．ゲストOSには組込み機器やPCなど幅広い分野で使われているLinuxを選択した．LinuxはIA-32環境をはじめ，PowerPC，ARMなど多くのアーキテクチャ上に移植されている．

IA-32環境で実行するLinuxは，IA-32の持つページング機構を用いて4GBの仮想メモリ空間を作り，この中で実行する．また，LinuxはIA-32が提供する保護リング機構のうち2つのレベルを使いカーネルとユーザプロセス間の保護を確立する(図1-(a))．通常はカーネルをレベル0，ユーザプロセスをレベル3で実行する．これにより，カーネルがハードウェア全体を管理，各プロセスに対して適切にリソースを配分することができる

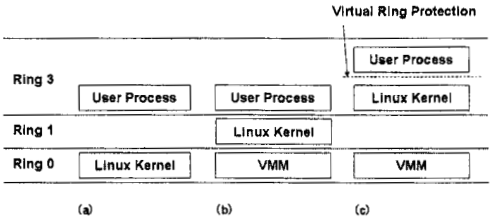

図1: 保護リング上のOSとVMM

2.3 3レベルリングでのVMM

IA-32上にVMMを構築する場合，OSが使用していない保護リングを利用する方法が一般的である．この場合，VMMとゲストOSはCPUの保護リング内に図1-(b)のように配置される．ゲストOSカーネルのソースコードの一部に変更を加え，ゲストOSカーネルをリング1で実行する．ハードウェアの電源をいれると，まずVMMがリング0で起動する．次にカーネルを呼び出してゲストOSの初期化を行い，最後にユーザプロセスが生成される．ゲストOSカーネルはリング1で実行しており特権命令を発行することができないため，これをVMMに依頼し処理する．また，ゲストOSカーネルはユーザよりも特権の高いリングで実行しているため，VMMが無い状態で実行している場合と同様にユーザプロセスを管理することができる．このようなVM内で実行するOSをゲストOSと呼ぶ．

本論文では，ゲストOSに対して，実機上で実行されるOSをネイティブOSと呼ぶ．

3 2レベルリング内でのVMMの実現方法

組込み機器用のCPUでは2レベルのみの保護リングを提供するものが多く，2.3で紹介した3レベルの保護リングを使う手法ではVMMを構築することができない．そこで，本研究ではゲストOSカーネルとユーザプロセスをリング3，VMMをリング0で配置することで，2レベルの保護リング環境で実行できるVMMを実現する．この場合，ゲストOSカーネルとユーザプロセスは，同じ特権のリングで実行されるため，3レベルの保護リング環境のようにVMMが特にゲストOSカーネル，ユーザプロセス間の保護を行わない方法ではユーザプロセスからカーネル空間へのアクセスが可能となってしまい，ネイティブLinuxでのカーネル，ユーザプロセスの階層構造を保つことができない．

そのため，このようなVMM環境ではVMMがゲストOSカーネルとユーザプロセス間に何らかの方法を使って保護を構築しなければならない．本研究

では VMM がゲスト OS のカーネルとユーザプロセス間に仮想保護リングを提供する (図 1-c). 仮想保護リングは VMM によって管理され，ユーザプロセスからカーネルが持つメモリ領域へのアクセスを禁止する．またこの機構はゲスト OS カーネルからユーザプロセスのメモリ領域へのアクセスを可能にし，ネイティブ Linux 内での階層構造を保つことができる．

本研究では仮想保護リングの実現方法として IA-32 が提供するページング機構を用いる．ページング機構は仮想メモリを提供するための機能であり，保護リングが 2 レベルのみの CPU であってもページング機構が提供されていることが多い．例えばこのようなプロセッサとしては SH-4 がある．SH-4 は組込み機器用のプロセッサであり，ページング機構とソフトウェア制御の TLB により柔軟な仮想メモリ管理が行える．しかし，保護リングは 2 レベルであり，カーネルとユーザプロセスが実行される [5].

VMM 環境において，ページング機構はシステム全体の実行に関連する機能であるため VMM が管理する．3 レベルの保護リング環境では，ゲスト OS カーネルが用意したページテーブルを VMM が受け取り，VMM がチェックした後，ページテーブルを CPU に設定する．本研究では仮想保護リングを提供するため，VMM は VMM 内にページテーブルを 2 種類管理する．VMM はゲスト OS カーネルから指定されたページテーブルを VMM へコピーする際に，特権モードで参照できる範囲のカーネルモードページテーブルと，非特権モードで参照できる範囲のユーザモードページテーブルを生成する．ゲスト OS カーネル実行中にはカーネルモードページテーブルを，ユーザプロセスが実行中にはユーザモードページテーブルを使用する設定にすることで，ゲスト OS カーネルからはゲスト OS 全体が，ユーザプロセスからはユーザプロセスのみの領域が参照できる．(図 2)

また仮想保護リングの環境では，ユーザプロセスとゲスト OS カーネルが切り替わる際にページテーブルを切り替える必要がある．これはネイティブ Linux には無い処理のため VMM によって処理される必要がある．そのため VMM はカーネルモードとユーザモードの切り替えを検出しなければならない．検出方法については次章で詳細を述べる．

4 実装

3 章では，2 レベルの保護リング環境上に VMM を構築する手法の概要について述べた．本章では，本研究で構築した VMM の実装の詳細と，2 レベルの保護リング環境に VMM を実現するためのテクニックを紹介する．

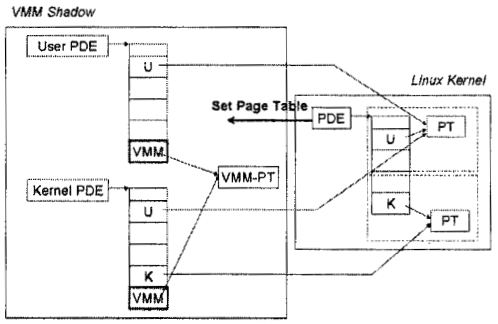

図 2: シャドウページテーブル

4.1 仮想アドレスマップ

VMM 環境において VMM からゲスト OS の仮想メモリの内容を参照したい場合には，VMM の仮想メモリ内にゲスト OS をすべてマップしてしまう方法が便利である．ネイティブ Linux は仮想メモリ 4GB 内にカーネルとユーザプロセスを配置するが，今回は VMM からゲスト OS を簡単に参照するためにゲスト OS の Linux カーネルに修正を加え，Linux の仮想メモリを 3.95GB とし，残りの領域を VMM 用メモリ空間として確保した．

4.2 セグメント機構の仮想化

IA-32 のセグメント機構は実行時に参照できる仮想アドレスの範囲を設定する機能である．ページング機構は，各ページごとに U/S ビットを持ち，U/S ビットが 0 のページは特権モード時にのみアクセスが可能となる機能を持っている．ネイティブ Linux ではページング機構の各ページの U/S ビットによってカーネルとユーザプロセス間のメモリ保護を提供する．そのため，ネイティブ Linux のセグメントは保護機構としては利用されず，仮想アドレス全体にアクセスできるように設定される．

本システムでは仮想アドレス空間内に VMM とゲスト OS が混在しているが，ゲスト OS から VMM のアドレス領域へのメモリの読み書きは VMM 保護の観点から禁止されるべきである．そのため，本システムではセグメント機構を利用する．VMM はゲスト OS 用の VMM のアドレス空間へのアクセスを制限するセグメントと VMM 用のすべてのアドレス空間へアクセスが可能な 2 種類のセグメントを用意する．これを実行中に切り替えることで，VMM 実行中には仮想アドレス空間内の任意の場所へのアクセスが可能とし，ゲスト OS からは VMM のアドレスへのアクセスが禁止することができる．

本システムでは，VMM をリング 0，ゲスト OS をすべてリング 3 で実行する．そのため，ページテー

ブルでVMMが使用するページのU/Sビットにより特権ページとすることでVMMとゲストOS間のメモリ保護を実現することもできる．しかしページング機構を用いた場合，メモリ保護に関するデバッグの際にページテーブル内の多数のページ設定を確認，テストする必要があるために大変な苦労を要する．そこで，本システムではVMMとゲストOS間のメモリ保護に，単純で設定量の少ないセグメント機構を採用した．

4.3 ページング機構の仮想化

本システムでは，仮想保護リングを提供するためにページング機構を利用する．そのため，CPUのページング機構をVMMが管理し，ゲストOSに仮想ページング機構を提供する．仮想ページング機構は，ゲストOSがページテーブルを構築し，CPUのページテーブル設定用レジスタ（CR3）にアクセスした場合に呼び出される．VMMの仮想ページング機構は，ゲストOSカーネルがCR3に書き込もうとした値を取得し，ゲストOSのページテーブルを取得する．

その後，VMMはページテーブルの内容から仮想保護リング用の2種類のシャドウページテーブルを作成する．ゲストOSからのページテーブルにはU/Sビットが0である特権ページとU/Sビットが1である非特権ページが含まれている．VMMはU/Sビットが0のページを含むページテーブルをカーネルモードページテーブルとして，U/Sビットが0のページを含まないページテーブルをユーザモードページテーブルとして作成する．また，両方のページテーブルの終端にVMM用のページを設定することで図2のように，アドレスの一部にVMMをマップすることが可能となる．以上のように作成したページテーブルをカーネルモード用ページテーブル，ユーザモード用ページテーブルそれぞれに設定し，ユーザモードとカーネルモードの切り替えの際にこれらのページテーブルを切り替えることでユーザプロセスとカーネル間の保護を実現する．例えば，ユーザモードで実行中にカーネルの持つアドレスへアクセスを起こした場合，カーネルのページはユーザモードページテーブル上には情報が無いためページフォルトが発生しVMMにユーザモード内でカーネル空間へのアクセスが発生したことが通知される．

4.4 割込み機構の仮想化

VMMはシステム全体を管理するために発生した割込みをすべてハンドルする．今回は割り込みコントローラ(PIC)とハードウェアの仮想化を省略するため，ゲストOSはPICとハードウェアを操作できるようになっている．そのためVMMはユーザ定義の割込みベクタをすべてハンドルし，必要な場合には適切なカーネルの割込みハンドラを呼び出す．この節では本研究で作成したVMMの割込み処理について述べる．

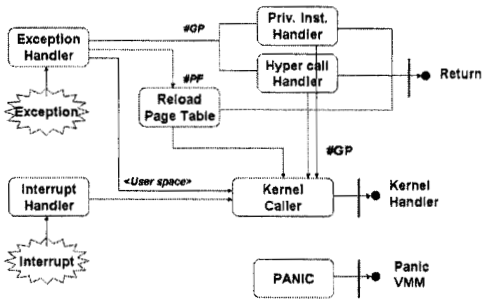

図3: VMMの割込み処理

4.4.1 ユーザ定義ベクタの仮想化

まず，ユーザ定義ベクタ割込みについて述べる(図3)．ユーザ定義割込みが発生すると，まず発生した割り込みに対応するVMMのハンドラが呼び出される．次に，VMMハンドラ内で発生した割込みに対応するゲストOSのハンドラを検索する．最後にVMMはゲストOSが割込みのハンドルに必要な情報を用意し，ゲストOSのハンドラを呼び出す．また，もし割込みがユーザモード中に発生していた場合には，仮想保護リングでの特権モード変更が発生するため，ページテーブルをカーネルモードページテーブルに変更する．

IA-32では割込みハンドラにも特権を指定することが可能であり，カーネルモードからのみ呼び出すことのできるハンドラを作成することが可能である．しかし，現在はこの機能の仮想化は行っていない．

4.4.2 一般保護例外のハンドリング

一般保護例外(GP)は，予約済みベクタで発生する例外である．GPの発生要因としては，(1)ゲストOSカーネル内の特権命令発行，(2)ユーザプロセス内の特権命令発行，(3)IRET検出用のGP(詳細は4.5)，(4)ゲストOS内からのVMMのメモリの仮想アドレスへのアクセス時などである．

VMMのGPハンドラはまずGPを発生させた命令をデコードする．この命令が特権命令の場合，要因は(1)，(2)のいずれかであり，(1)の場合はGP発生時のレジスタ情報を元に特権命令をエミュレーションする．(2)の場合はネイティブLinuxでも一般保護例外を発生させる要因であるため，この場合にはゲストOSの割り込みハンドラへ処理を委譲する．(4)の場合はカーネルが不正な領域にアクセスしようと

した場合であるため，VMM がなにかしらの対応を行う必要がある．しかし，現在は適切な対応策が決定していないため，(4) の要因による GP が発生すると VMM は発生時のレジスタ情報をダンプして停止する．

4.4.3 ページフォルトのハンドリング

ページフォルト（PF）もまた予約済みベクタで発生する例外である．PF の要因としては，3 つの要因が考えられ，(1) ユーザモード中のカーネルメモリへのアクセス，(2) ゲスト OS ページテーブルがシャドウページテーブルへ反映されていない場合，(3) ゲスト OS が準備していないページでのページフォルトの場合，がある．(1) はユーザプロセスからゲスト OS のカーネル領域へアクセスのため GP であり，この場合には VMM はゲスト OS のカーネルへ GP を通知する．(2) は VMM がシャドウページテーブルを持つために発生する例外である．ゲスト OS から VMM へのページテーブル登録後，ゲスト OS カーネルがゲスト OS 内のページテーブルを変更した場合，シャドウページテーブルとゲスト OS ページテーブル間に矛盾が生じる．そのため，CPU に設定されているページテーブルはゲスト OS が指定した適切なページングを行うことができず PF が発生する．この例外の場合，ゲスト OS のページテーブルを読み込み，シャドウページテーブルを再構築し，ゲスト OS ページテーブルをシャドウページテーブルへ反映させた後，PF を発生させた命令へ戻し，実行を再開させる．ここで，この方法によるシャドウページテーブルの更新はゲスト OS の書き換えたページがシャドウページテーブル内に登録されていない場合のみ有効であり，ページがシャドウページテーブルに登録されている場合には PF が発生しないためシャドウページテーブルの更新が発生しない．これについては VMM がゲスト OS によるページテーブルの変更を検出する必要があるが，現在は未対策である．(3) はゲスト OS でハンドルされる例外のため，VMM はゲスト OS のハンドラを呼び出す．

現在，以上に述べなかった予約済みベクタ例外が発生した場合，VMM はゲスト OS ハンドラを呼び出し例外処理を依託する．

4.4.4 ハイパーコール

VMM 環境ではゲスト OS から VMM を呼び出すためのハイパーコール (ハイパーバイザーコール) 機構を提供することが多い．ハイパーコール機構により，CPU には無い機能を VMM がゲスト OS へ仮想的に提供したり，1 回の VMM 呼出で複数の特権処理を行うといった柔軟な仮想マシン環境を構築することができる．本システムもハイパーコール機構を用意しており，ゲスト OS からのソフトウェア割込みにより発行することができる．ゲスト OS がソフトウェア割込みを特定のベクタに対して行った場合，VMM がこれをハイパーコールとして検出し，このときのレジスタの値を元に処理を決定し実行する．今回は 2 レベル保護リング環境でのハイパーコールのテストとして，IA-32 の特権命令 4 個（lgdt，lldt，lidt，ltr）をハイパーコールによって実装した．また，これらの動作を確認するためにゲスト OS カーネルの該当する命令をハイパーコールに置き換えた．

4.5 IRET の検出

4.3 では 2 種類のページテーブルによる仮想保護リングを作成した．本節ではユーザモードとカーネルモードの切り替え時に行う際の仮想保護リングの特権移行と IRET のハンドリングについて紹介する．

本研究では，ユーザモードとカーネルモードの切り替えの際にページテーブルを切り替えることで仮想保護リングを提供する．ユーザモードからカーネルモードへの切り替えは通常割込みによって起こり，これは本システムにおいても同様である．この場合必ず VMM の割込みハンドラが呼び出されるため，モードの切り替えを簡単に検出することができる．しかし，IA-32 環境ではカーネルモードからユーザモードへの切り替えを VMM が検出するのは困難である．ネイティブ Linux では，このモード移行の際に IRET 命令を発行する．IRET 命令はある特権状態から特権の低い，もしくは特権の同じコードセグメントへジャンプするための命令である．現在の特権と移行後の特権を比較し，条件を満たす場合にはセグメントの切り替え，スタックの変更が自動で行われジャンプする．このような IRET 命令はカーネルからユーザプロセスへの移行や，割り込みハンドリング完了後，割り込み発生位置へのリターン処理に用いられる．ネイティブ Linux ではこの方法により特権モードの切り替えを行っているが，本システムでは特権モードを変更を VMM によるページテーブルの切り替えで実現しているため，ゲスト OS カーネルの IRET 発行時に必ず VMM が呼び出されなければならない．

そこで，本研究では VMM を呼び出す方法として，ゲスト OS カーネルのマクロ定義に変更を加えることで IRET 命令の発行時に例外を起こすことでこれを実現した．変更点としては，カーネルモード，ユーザモードとしてマクロ定義されたコードセグメントの値をそれぞれ VMM があらかじめ予約した不正なセグメント番号を含む，特権，非特権セグメントに変更する．これによりゲスト OS 実行中にゲスト OS カーネルがこのマクロ定義を元にセグメントの値を準備し IRET 命令を発行すると，宛先セグメントは不正なセグメントとなるため GP が発生する．GP は VMM によりハンドルされ，命令のデコード，ス

タックの解析を行い，原因となったセグメント値から宛先のコードセグメントがユーザセグメントなのかカーネルセグメントなのかを判別することができる．もし，宛先の特権モードがユーザモードの場合には，ユーザモードページテーブルを設定し仮想保護リングの特権を変更する．最後に，修正した宛先へジャンプすることでIRETのエミュレーションは完了となる．

5 結果

本研究では，IA-32アーキテクチャのPentium 4 XEマシン上でVMware Workstation 5.5.1を実行し実験環境を構築した．VMwareは仮想マシンとして，ホスト環境と同種類のCPUと512MBのメモリを提供する．以上の仮想マシン内で，本研究で構築したVMMとゲストOSのLinux 2.6.17，シェル環境を提供するBusyboxを実行する．Linux 2.6.17には4章に示した変更点を加えた．

仮想マシン上でgrubを使いネットワーク経由でブートした結果，VMMの起動，ゲストのLinuxカーネルの起動，シェルの起動が確認できた．また，シェル内でコマンドを実行したところネイティブLinux同様の結果が得られ，正常に処理されていることが確認できた．現在はゲストOSのカーネル，ユーザプロセス間の保護に関するテスト及び速度性能に関する評価は行っていないため，今後の研究課題とする．

6 関連研究

DiscoはMIPS用のVMMである[1]．MIPSは3レベルの保護リング機構，ページング機構とソフトウェアTLB制御を持ち，SH4とのアーキテクチャ的な差は主に保護リングレベルのみである．そのため本論文の手法を用いることで簡単に移植することができ，本手法は大変有用であると考える．

ソフトウェアによる2レベルの保護リング環境内でのVMM構築方法としては，x86-64環境用のXenがある[2]．この手法も本研究同様の2種類のページテーブルを使う手法が提案されているが，詳細，実行結果等は不明である．

また，2レベルの保護リング環境で複数のOS環境を実行する方法としては，マイクロカーネルによる方法がある[3]．しかし，この方法の場合はネイティブOSを参考にしOSサーバやエミュレーションライブラリを構築する必要があり，ソフトウェアの設計実装コストが非常に高い．本研究では，ゲストOSへの変更はほとんど無いため移植コストは低く，OSレベルのバグを増やすことなくシステムを構築することができる．

7 おわりに

本研究では，2レベルの保護リング環境にページング機能を利用した仮想保護リングによりVMMを実行する方法を提案した．本研究は，特権リング上でVMMを実行し，1つの非特権リング内でカーネルとユーザプロセスを実行する．これには，ゲストOS 1つに対しVMMが仮想保護リング用にシャドウページテーブルを複数用意することで，カーネルとユーザプロセス間に保護機能を実現した．現在はシステムではゲストOS 1つに対してページテーブルを2種類作成しているが，さらにVMM内で複数のページテーブルを用意することで複数レベルの仮想保護リングを提供でき，今後もさまざまな用途に利用可能な手法であると考える．

また，本研究では仮想保護リングを実現するために同じ特権モード内で実行されるカーネルとユーザプロセス間の切り替えを行うIRETの検出を行った．これはLinuxのマクロ定義の変更とVMMの割込み処理部のみの変更によって実現することができた．

以上の方法で構築したVMMは2レベルの保護リングのアーキテクチャ内で動作し，VMMが構築した仮想マシン内でLinuxおよびBusyboxが実行できることを確認した．

参考文献

[1] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum : "Disco: Running Commodity Operating Systems on Scalable Multiprocessors" , ACM TOCS, 15(4) 1997.

[2] Ian Pratt, Keir Fraser, Steven Hand, Christian Limpach, Andrew Warfield : "Xen 3.0 and the Art of Virtualization" , Ottawa Linux Symposium, 2005.

[3] Jochen Liedtke : "Toward Real μ-kernels" , Communications of the ACM, 39 (9), pp. 70-77, 1996.

[4] Intel Corporation: "Intel 64 and IA-32 Architectures Software Developer's Manuals" , Intel Corporation, 2006.

[5] ルネサステクノロジ: "SH-4ソフトウェアマニュアル Rev.6.00" , ルネサステクノロジ, 2006.