

## 2 レベル保護リング環境でのVMM構築と評価

青柳 信吾 迫川 修一

筑波大学 システム情報工学研究科

小規模組み込み機器用の 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、組み込み機器に移植することが困難である。[1] では、ページング機構を利用した仮想保護リングにより、2 レベルの保護リング環境で実行する仮想マシンモニタ (VMM) を構築した。本論文ではこれの性能を測定、性能改善について述べる。また性能改善の結果、ネイティブ Linux のレイテンシより 1.5~4 倍程度でゲスト OS 実行に成功した。

## Implementation and Evaluation of VMM on 2-Level Ring Architecture

Shingo Aoyagi Shuichi Oikawa

University of Tsukuba, Graduate School of Systems and Information Engineering

Most of the currently available embedded processors provide only two protection levels, privileged and unprivileged, at most. The researches and developments of VMMs are, however, currently conducted mainly using the Intel IA-32 processor architecture, which provides four protection levels: porting such VMMs to embedded processors requires significant effort and costs. In [1], we introduced the VMM that has “virtual protection ring”, which provides isolation between the guest OS kernel and processes. This paper presents the performance of this VMM, and the techniques we applied to improve the performance. As a result, we successfully reduced the latencies and kept them 1.5 to 4 times as slow as the native Linux.

### 1 はじめに

今日、組み込み機器環境で実行されるプログラムの複雑化に伴い、これらのプログラムに潜在的に含まれるバグへの対策が重要な課題となっている。その解決方法の一つとして、近年では、仮想マシンモニタ (VMM) を利用し、ハードウェア上に仮想マシン (VM) を構築し VM 内でサービス用ソフトウェアを実行する方法が提案されている。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にもハードウェアに危機的な影響が出る前に VM 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 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、組み込み機器環境で実行可能な VMM はほとんど無い。また IA-32 環境用の 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、小規模な組み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いない場合があり、そのようなプロセッサへの移植には VMM 設計の見直しを行う必要がある。

そこで、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み込み機器環境へ移植することを可能にするため、2 レベルの保護リング環境に VM

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した [1]。2 レベルの保護リング環境内に VMM 環境を構築する手法としては、ページング機能を利用した仮想保護リング機構を利用した。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。これにより、[1] 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 を実行できることを確認した。

本論文では [1] で構築した VM 環境の性能について述べる。本研究では構築した VMM を実行し、この環境内で lmbench を実行することによりマイクロベンチマークを行った。また、これにより得られた性能を改善するため、VMM に対してグローバルページ機構の利用、分岐除去、ページテーブルキャッシュを利用した速度向上を試み、結果としてマイクロベンチマークにおいてネイティブ Linux から 1.5~4 倍の速度低下のみで IA-32 環境に VMM 環境を構築することに成功した。

### 2 これまでのアーキテクチャ

これまでに研究されている VMM は、IA-32 アーキテクチャ環境用に設計されたものが多い。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 [5]。IA-32 CPU はシステム

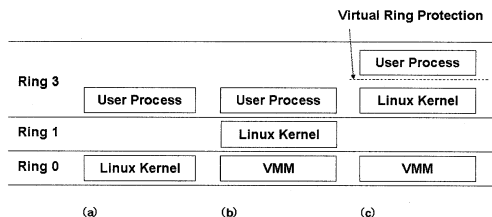


図 1: 保護リング上の OS と VMM

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、割り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 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り、この状態を特権モードと呼ぶ。また、これ以外の状態を非特権モードと呼ぶ。

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することができる。

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。このシステムでは、まず VMM がリング 0 で起動し、ゲスト OS カーネルはリング 1 で実行される。ここで、ゲスト OS カーネルは特権を持たないため、これを VMM に依頼し処理する。

### 3 2 レベル保護リング環境での VMM アーキテクチャ

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 節で紹介した手法では VMM を構築することができない。

そこで [1] 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 で配置し、ゲスト OS カーネルとユーザプロセス間に仮想保護リングを構築することにより、これらの問題を解決し組

込み環境等の 2 レベル保護リング環境での VMM 構築方法について提案した (図 1 - (c))。本章では仮想保護リングを利用した VMM の構築方法の概要について述べる。

#### 3.1 メモリ管理機構

本 VMM は、メモリ管理機構と割り込み管理機構から成る。まずメモリ管理機構について述べる。

メモリ管理機構は、仮想保護リングにおける保護を実現し、現在実行中のプログラムの特権にあわせたメモリアクセスを提供する。メモリ管理機構では IA-32 の提供するセグメント機構とページング機構の管理を行う。

まず、VMM とゲスト OS 間のメモリ保護について述べる。本環境では VMM からゲスト OS を簡単に参照するため、ゲスト OS 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、VMM とゲスト OS が同じアドレス空間内で実行される仕様とした。そのため、これらの間に保護を実現する必要があるが、本環境では VMM、ゲスト OS それぞれのセグメントを設定することで、VMM 実行中にはアドレス空間全体へアクセスを許可し、ゲスト OS 実行中にはゲスト OS 内のみへのアクセスを許可する保護を実現した。

次に仮想保護リング機構が提供する、ゲスト OS カーネルとユーザプロセス間の保護について述べる。仮想保護リングの提供する保護は、ゲスト OS カーネルはゲスト OS 内すべてのアドレス空間へのアクセスを許可し、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ものである。この実現方法としては、本システムでは IA-32 の提供するページング機構を利用した。

3 レベル保護リング環境の VMM は通常、ゲスト OS が生成したページテーブルを、VMM が管理する 1 つのページテーブルへ追記することでゲスト OS のメモリ管理を行う。ここで VMM が管理するページテーブルをシャドウページテーブル (シャドウ) と呼ぶ。本研究では、仮想保護リングが提供するゲスト OS カーネルとユーザプロセス間の保護関係を実現するため、VMM 内にゲスト OS 内の権限に応じた、カーネルモードページテーブルとユーザモードページテーブルの 2 個のシャドウを用意する。VMM はゲスト OS カーネルから指定されたページテーブルからシャドウを生成する際に、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へ、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へ書き込む。その後、ゲスト OS の実行の際に、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

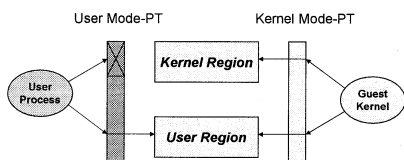


図 2: ページング機構を利用した保護

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可能とし、仮想保護リングを実現する (図 2)。

仮想保護リング実現の際には、ゲスト OS 実行中にこれらのページテーブルの切替えを行う必要がある。これについては次節にて述べる。

## 3.2 割込み管理機構

続いて割込み管理機構について述べる。割込み管理機構はシステム内で発生するすべての割込みを検出し、これらを管理する。これにより、(1) 特権命令の検出、(2) メモリ保護違反の検出、(3) ゲスト OS 内イベント検出、(4) 割込みの管理を行うことができる。

### 3.2.1 特権命令の検出

VMM 環境では、ゲスト OS カーネルは非特権モード内で実行されるため、一部の特権命令を発行した際に一般保護例外 (GP) が発生する。GP は特権違反や CPU への不正な設定を行った際に発生する例外である。

本システムでは、ネイティブ Linux からほとんど変更の無いゲスト OS のプログラムを VMM により非特権モード内で実行する。そのため、プログラムにはネイティブ Linux においてハードウェアの管理や CPU の特権機構の操作などに利用していた特権命令が含まれており、これにより GP が発生する。

本環境では、VMM によって GP が発生した場合に VMM 内の特定の GP ハンドラが起動するよう設定を行う。GP ハンドラでは、発生要因を取得、解析を行い VMM が代行処理を行うことによって、ゲスト OS が必要とする処理を提供する。これにより、例えば、ゲスト OS 内で発行されるページテーブルの設定を行うための CR3 レジスタ操作命令などを検出し、登録しようとしたページテーブルを VMM 内へ取得、シャドウの生成が可能となる。

### 3.2.2 メモリ保護違反の検出

本システムではページング機構を利用して仮想保護リングを実現する。ゲスト OS カーネルとユーザプロセス間保護の実現方法としては、2 種類のシャドウを用意し実行プログラムに応じてこれらを切替えることで、ユーザプロセスからカーネルへのメモリアクセスを禁止する。これにより、ゲスト OS 実行中にユーザプロセスがカーネル領域へアクセスを行うと、ページング機構によりページフォルト (PF) が発生する。本システムではこの PF を検出、管理することで、ゲスト OS 内の仮想保護リング違反を検出しゲスト OS カーネルに対して保護例外が発生したことを通知する。

### 3.2.3 ゲスト OS 内のイベント検出

本節ではゲスト OS 内で発生するイベントの検出方法について述べる。

3.1 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。本システムでは、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ため、これらのモード切替は、VMM が検出する必要のある重要なイベントである。

ユーザプログラムからカーネル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。本システムでは割込みを VMM が管理するため、このモード切替は VMM の割込みハンドラが呼び出しにより容易に検出することができる。

カーネルからユーザプログラムへの切り替えは IRET 命令により行われる。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の変更を 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 が呼び出される必要がある。

しかし、ゲスト OS として変更の無いネイティブ Linux を実行した場合、ゲスト OS カーネル内で実行される IRET 命令は非特権モードから非特権モードへのジャンプ処理として発行されるため、特権命令として判定されず CPU によって直接処理され、VMM はゲスト OS カーネルからユーザプロセスへの移行を検出することができない。

そこで本システムではモード切替が発生する IRET 命令の検出方法として、ゲスト OS カーネル内のユーザプログラムのセグメント番号を設定するマクロ定義に変更を加え、IRET 命令の発行時に例外を起こすことでイベント検出を実現した (図 3)。変更点としては、ユーザーモード用に定義されたセグメント番号を、VMM が確保した不正なセグメント番号を含むセグメントに変更する。

図 3 は例外を利用した IRET 命令の検出及び仮想特権切替のためのページテーブル切替を示したものである。まず、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使用して IRET 命令を発行すると (a)、宛先セグメントは不正なセグメントとなるため、GP が発生し (b)、VMM に対してモード切替イベントが発生したことが通知される。次に、VMM はこのイベントにあわせてシャドウの切替を行い、仮想保護リングの提供する仮想特権を変更する (c)。最後に VMM からユーザプロセスへジャンプすることで、適切な仮想特権を持つプロセスが実行される。ユーザプロセスからゲスト OS カーネルへの移行は割込みによって行われるため、VMM が割込みを管理し、割込み発生時にページテーブルを切替えることにより仮想保護リングの仮想特権を変更する。

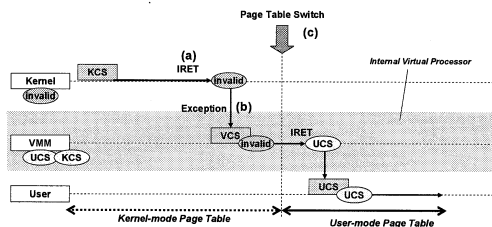


図 3: 特権移行命令の検出

### 3.2.4 その他の割込み管理

今回のシステムでは、VMM を簡単に構築するために CPU 以外のハードウェアの仮想化は行わない。しかし割込みの発生はゲスト OS のカーネル、ユーザプロセス間のモード移行を発生させる要因であるため、VMM は割込みをすべて管理する。

まず、割込みが発生し VMM の割込みハンドラが呼び出された場合には、VMM は割込み要因を解析し VMM が処理すべき要因かどうかを判定する。発生した割込みが GP など VMM がハンドルする必要があるものである場合には、VMM はこれに応じた処理を行った後に割込み元へ処理を戻す。もし、発生した割込みが VMM がタイマ割込みなど、ゲスト OS の実行のみに関連するものの場合には、VMM の割込みハンドラはゲスト OS の割込みハンドラを呼び出すことで、間接的に割込みハンドラを起動させ、

ゲスト OS に割込みのハンドルを委任する。もしユーザプロセスからカーネルの呼び出しなど、特権モードの移行が発生する場合には、割込みハンドラ内にてページテーブルの切替えを行う。

### 3.2.5 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。そこで、本システムもハイパーコール機構を提供する。本システムではゲスト OS は特定のソフトウェア割込みを起こすことで VMM を呼び出すことができる。今回は IA-32 の特権命令 4 個 (lgdt, lldt, lidt, ltr) を実装した。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 結果

### 4.1 実験環境

本研究では、実験環境として IA-32 の Pentium Xeon マシンである、DELL Precision 490, Xeon 5130 (2GHz, 4MB L2 Cache, 1333MHz FSB), DDR2-SDRAM 2GB を用意した。この環境内で、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。

### 4.2 ゲスト OS への変更

ゲスト OS Linux 2.6.17 へ、表 1 に示す 10 箇所 23 行へ変更を加えた。これらの変更のうち、(a) は現在 MWAIT 仮想化の実装が終わっていないために発生している変更点であり、(b) は VMM に実装されているハイパーコール機構が正常に機能しているかどうかを試すための変更である。そのため、変更点は表中の無印の項目 2 点のみに削減することが可能である。またこれらの変更項目は複雑な変更ではなく機械的に変更できる内容のため、Linux のバージョンアップへの対応や、また Linux に類似したオープンソースの OS も本研究で構築した VMM 上で実行することも可能であると考えられる。

テスト環境内で grub を使いネットワークブートを行ったところ、VMM の起動、ゲストの Linux カーネルの起動、シェルの起動を確認した。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、正常に実行されていることを確認した。

表 1: ゲスト OS への変更点

| 変更内容                      | 変更数 | 行数 |
|---------------------------|-----|----|
| (b) lgdt を hypercall に変更  | 2   | 6  |
| (b) lidt を hypercall に変更  | 2   | 6  |
| (b) lldt を hypercall に変更  | 2   | 6  |
| (b) ltr を hypercall により実装 | 1   | 2  |
| ユーザセグメントの値を変更             | 1   | 1  |
| 仮想アドレス上限の変更               | 1   | 1  |
| (a) MWAIT 命令の無効化          | 1   | 1  |
| 合計                        | 10  | 23 |

### 4.3 性能測定

構築したシステムのゲスト OS 上でベンチマークスイートを実行することで、2 レベル保護リング環境での VMM がゲスト OS に与える影響を測定する。今回はベンチマークスイートとして lmbench を利用した。lmbench は OS のマイクロベンチマークを行うベンチマークスイートであり、システムコールのレイテンシやパイプのレイテンシなどを測定することができる。今回の測定では、4 種類についてマイクロベンチマークを行った。ベンチマークの内容については表 2 に示す。また、ベンチマーク対象としては、ネイティブ Linux 及び VMM 環境内の Linux の 2 種類について行う。

表 2: lmbench の測定内容

| 項目           | レイテンシの測定対象          |
|--------------|---------------------|
| null syscall | null システムコール        |
| pipe         | pipe システムコール        |
| fork+exit    | fork+exit システムコール   |
| fork+exec    | fork+execve システムコール |

これにより得られた結果を表 3 に示す。表中の“x86”がネイティブ Linux での lmbench, “VMM”が VMM 環境内 Linux での lmbench の結果である。これより、ネイティブ Linux のレイテンシを 1 とした場合のレイテンシ比を図 4 - “VMM”に示す。

結果より、VMM による性能への影響はプロセス切り替えの際に最も影響が現れ 11 倍程度の性能低下となった。Linux ではプロセス切り替えの際に、各プロセスにあわせたページテーブルへ切り替えることでプロセス間のメモリ保護を実現しているため、VMM の提供する仮想ページング機構が速度性能上問題であることが推測される。

## 5 性能改善

前章にて、VMM の性能評価と VMM が性能低下を起こしている原因についての解析を行った。本章ではこの結果に基づいて性能改善を行い、性能への影響を測定する。

### 5.1 Global Page の利用

まずページテーブル更新による TLB (Table Lookaside Buffer) への影響について考察する。

IA-32 ではページテーブルにより仮想アドレス空間を実現しているが、ページテーブルウォーク高速化のため仮想アドレスから物理アドレスへの変換情報を CPU 内に用意された TLB に保存する。また、ページテーブル登録のための CR3 レジスタへの代入が行われると TLB による仮想アドレス変換とページテーブルによる仮想アドレス変換の処理が異なる可能性があるため、TLB の内容をフラッシュすることになっている。本システムではページング機構を利用し仮想保護リングを実装するため、ゲスト OS によるページテーブルの更新処理や、割り込み発生時などに TLB フラッシュが頻繁に発生する。そのため、TLB フラッシュによる影響は無視できないと考えた。

そこで、TLB フラッシュの対策を行うことで性能改善を試みる。方法としては、IA-32 のグローバルページ機構を使う。グローバルページ機構はページテーブルのうち、ページテーブル切り替え時に TLB フラッシュされないページを設定できる機能である。この機構を利用し、仮想アドレス空間へ VMM をマップするページをグローバルページとすることにより、CR3 変更時にも VMM のページ変換情報が TLB 上に維持されるため、VMM を高速化することが可能である。

この手法により得られた高速化結果を図 4 に “+G”として示す。+G では、全体的にレイテンシが減少しているが、Pipe のレイテンシのレイテンシが増加している。この原因としては、ゲスト Linux がこれらの処理の際にグローバルページ機構を操作している事が原因であると考えられる。IA-32 ではグローバルページの管理は CR4 により行い、CR4 は特権レジスタであるが、現在の VMM 実装ではゲスト OS が CR4 の操作命令を発行すると、この命令の内容が直接 CPU に設定されるようになっていたため、ゲスト OS はグローバルページ機構を操作できる環境になっていた。そのため今後は、CR4 操作のエミュレーションへグローバルページの管理を適切に処理するように VMM を変更する必要がある。

## 5.2 分岐命令の削除

本研究では、ゲスト OS の生成したページテーブルを元に、VMM がシャドウページテーブルを構築する。このページテーブル構築には、ゲスト OS ページテーブルの内容取得とシャドウへの書き込みの他に、特権ページの振分けが必要となる。シャドウ生成の際、ゲスト OS が登録しようとしたページが特権ページである場合には、そのページを VMM 内のカーネルモードページテーブルへ非特権ページとして複製し、ユーザページテーブルへは無効なページとして登録する。通常この処理を書く場合には、ページの特権を判定し結果を書き込むプログラムとなるが、このような比較には多くの演算が必要となるため、ページテーブル生成の実行時間が増加する結果となる。また、IA-32 はパイプライン方式の CPU であり、分岐後の命令を予測することでプログラムを高速に実行するが、分岐予測失敗の場合、先読みしたプログラムが破棄されてしまうため、分岐処理を回避することが高速なプログラムの構築に必要となる。

そこで本実装では、シャドウページテーブル生成の高速化を計るため、ループ判定以外の分岐命令を削除した。方法としては、分岐命令を IA-32 のページング機構の持つ、ページテーブルエントリ Present (P) ビットを利用した論理演算へと変更し、分岐命令を削除する。ページテーブルエントリ中の P ビットはそのページテーブルの有効性を示すものであり、これが 0 に設定されたページにアクセスした場合、PF が発生する。そこで、ユーザモードページテーブル生成の際に、ゲスト OS ページテーブルの特権ビットを論理演算により P ビットの位置へと移動し、このビットによりマスク処理を行うことで、ゲスト OS 内の特権ページをユーザページテーブル内で無効なページとすることができる。ゲスト OS の非特権ページは、このマスクによる影響を受けないため有効なページとなり、仮想保護リング機構の求める 2 つのページテーブルを構築することができる。

## 5.3 Page-Table Cache の導入

VMM の高速化のため、TSC (Time Stamp Counter) レジスタを用いて VMM 内の関数それぞれのレイテンシを測定した。これにより、VMM 内の処理のうち CR3 レジスタへの代入が最も処理時間がかかることが判明した。この処理はゲスト OS がゲスト OS が生成したページテーブルを VMM へ登録するための処理であり、仮想保護リング構築のためページテーブルの複製処理が必要となる。IA-32 の最上位のページテーブルは 4KB であり、これを複製するには最低 8000 命令が必要となるため、膨大な時間がかかると予想できる。そこで、本研究ではページテーブルの生成回数を削減することによる、VMM の高速化を考案した。

ページテーブル生成回数削減方法として、VMM 内で生成するページテーブルのキャッシュ機構を構築した。一度 CR3 に設定されたページテーブルは VMM 内でキャッシュすることで、再度同じ CR3 の値が設定された際にキャッシュを利用し高速化を図る。これを本システムではページテーブルキャッシュと呼ぶ。ページテーブルキャッシュは、VMM がゲスト OS から CR3 の値を取得した際に、過去に登録された CR3 の値と比較し、利用可能な場合は過去に生成したページテーブルキャッシュ利用し高速化を図る。もし、ゲスト OS から登録されたページテーブルアドレスが過去に登録されていない場合にはこのページテーブルのキャッシュが存在しないために、VMM は領域を確保しページテーブルキャッシュの生成を行う。これにより、過去に登録されたページテーブルの場合には、シャドウの生成処理が発生せず VMM 実行の高速化を期待することができる。

本研究ではページテーブルキャッシュを 8 個保存し、FIFO (First-In First-Out) で更新する。キャッシュが限界量となった場合には、最も古いキャッシュが廃棄され、領域が確保される。キャッシュ更新方式として FIFO を選んだ理由は、実装が簡単なためであり、今後更新アルゴリズムについて再度検討する必要がある。

VMM 内にページテーブルキャッシュを設けることの問題点として、ゲスト OS はゲスト OS 内で実行するプロセス分のページテーブルを確保するが、プロセスの終了などにより不要となったページテーブルを VMM がキャッシュとして用意してしまった場合、そのページテーブルを再利用したプロセスが過去のプロセス用のページテーブルを利用できてしまう場合がある。そのため、ページテーブルの再利用が発生するゲスト OS の fork システムコール (プロセスの生成を行う) が発生した際に VMM 内のページテーブルを消去する仕様とした。これにより、プロセスの切替が行われるコンテキストスイッチの際にはページテーブルが再利用されるが、新しく生成されるプロセスは新規のページテーブルを利用することができる。

この手法により得られた lmbench 結果を図 4 - “+F” として示す。結果より、+F では pipe のレイテンシ削減に成功している。pipe ベンチマークは特定のプロセス間でデータを交換するベンチマークであり、頻繁にコンテキストスイッチが発生し、ゲスト OS カーネルによりページテーブルの登録が行われる。ページテーブルキャッシュが有効な VMM の場合、ベンチマーク開始時に VMM に作られたページテーブルキャッシュはベンチマーク実行中常に有効となり、シャドウページテーブルを更新することなくプロセスを切り替えることが可能となるため、ページテーブルキャッシュを利用しない VMM から大幅なレイテンシ削減が可能になったと考えられる。

これより本研究では、2 レベル保護リング環境上で

実行する VMM に性能改善手法を複数適応することにより、マイクロベンチマーク測定においてネイティブ Linux から 1.5～4 倍程度の速度低下により VMM 環境を実現することができた。

## 6 Xen (x86\_64 版) との比較

最後に本システムと既存の VMM の性能を比較することで、VMM の性能について考察する。本システムとの比較対象として、Xen3.0.3 を x86\_64 (64bit) 環境用に構築したもの（これを以降 “Xen64” と表記し、32bit 環境で実行する Xen に関しては “Xen” と表記する）を利用する [3]。Xen64 が実行される x86\_64 環境は 2 レベルの保護リングのアーキテクチャであり、Xen64 は 2 レベル保護リング環境用の VMM である。OS 環境としては、Xen64 上でゲスト OS の Linux Knoppix が実行される Xenoppix (x86\_64 版: 以降 Xenoppix64) を利用し、これを Dell Precision 490 上で実行する。また 32bit 環境と条件を合わせるため Linux の SMP 機能を無効にし CPU は 1 個のみを認識させる。

また、Xenoppix64 は x86\_64 環境で実行されるため、ゲスト OS としては x86\_64 Knoppix (以降 Knoppix64) が実行される。そのため、Xenoppix64 とネイティブ Linux とのレイテンシの比較には、ネイティブとして x86\_64 環境用に構築された Knoppix64 を用いる。Xenoppix64、Knoppix64 を実行しレイテンシを測定したところ表 3 に示す “Xen64”、“x86\_64” の結果を得ることができた。これらの比較により得られたレイテンシ比を図 4 - Xen (x86\_64) に示す。

表 3: lmbench の測定結果

| env.   | sys  | pipe  | fork  | exec  |
|--------|------|-------|-------|-------|
| x86    | 0.29 | 2.71  | 27.3  | 60.1  |
| VMM    | 1.13 | 6.89  | 84.8  | 164.9 |
| x86_64 | 0.13 | 4.21  | 117.1 | 131.5 |
| Xen64  | 0.69 | 10.28 | 334.2 | 359.6 |

システムコールのレイテンシでは本システムの VMM が相対的に早いことが確認できる。しかしシステムコールの呼び出しには本システム、Xen64 共にページテーブル切り替えが必要であるため、実装差はほぼ無いものと考えられる。本システムにおけるシステムコール発行の際の処理流れを図 5 に示す。今後はさらに Xen64 の実装を調査し、レイテンシ比の差の原因についての調査を行う必要がある。

pipe については VMM 環境、Xen64 環境共に処理の流れは同じものであると考えられる。これを図 6 に示す。しかし lmbench により測定を行ったところ、Xen64 のほうが少ないオーバーヘッドで実装されて

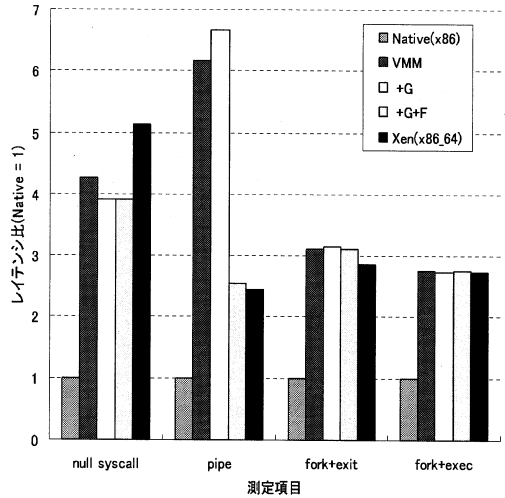


図 4: lmbench 結果

いることが判った。これについての詳細は現時点では得られていないが、Xen64 は para-virtualization 方式の OS でありゲスト OS に対して多数の変更を加えることで低レイテンシの環境を目指すものであるため、Linux において頻繁に使われる pipe 処理が最適になるように設計されている可能性が考えられる。

fork+exit レイテンシ測定では Xen64 が相対的に早く、fork+exec のレイテンシ測定ではほぼ同じレイテンシ比が得られた。これは本システムと Xen64 の内部処理の差から発生する考える。まず fork についての本システムの挙動を図 7 に、Xen64 の挙動を図 8 に示す。fork システムコールが発行されると、本システムも Xen64 もアドレス空間をゲスト OS カーネル用のものへと切り替える。次に、ゲスト OS カーネルが子プロセスのアドレス空間を生成し、VMM へ登録を行う。この際、本システムでは取得したアドレス空間を構成するためのページテーブルをすべて複製し、2 つのページテーブルを生成するが、Xen64 の場合はほとんど空のページテーブルを生成し設定を行う。そのため、高速にページテーブルの生成をすることが可能となる。ここで Xen64 により生成されるページテーブルは不完全なものであるが、Xen64 環境ではこれらのページが利用された際にページフォルトを起こし、この中で必要となったページテーブルを追記する仕様となっている。

execve システムコールの場合、処理の流れとしては fork とほぼ同様であるが、execve システムコール内ではプロセスの内容が大きく変更されるため、最低限のページテーブルを準備する Xen64 の場合、プロセス開始時にページフォルトが多発する。これより Xen64 環境での fork+exec は、fork+exit ほど速度性能を得ることができない。

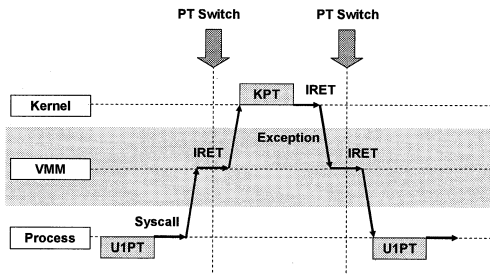


図 5: VMM 環境での syscall の実行

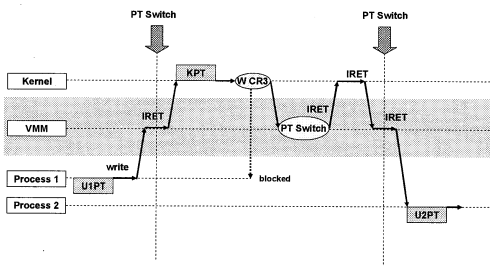


図 6: VMM 環境での pipe の実行

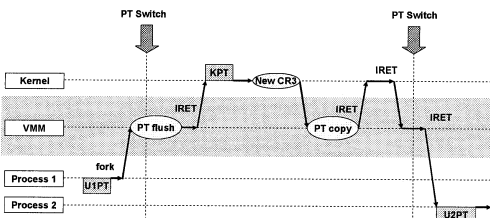


図 7: VMM 環境での fork, execve の実行

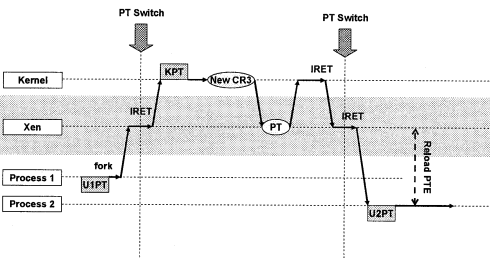


図 8: Xen64 環境での fork, execve の実行

## 7 関連研究

本研究では VMM 環境の速度性能向上のために 3 種類の性能向上手法を適応した. Xen [2] では VMM 環境の性能向上のため, ゲスト OS に対して大幅に変更を加えることでゲスト OS と VMM 間の切替コストを削減するパラバーチャライゼーションが述べられている. これに対して, 本研究ではゲスト OS の変更を最小限にとどめる代わりに VMM をゲスト OS に合わせて改善することで, Xen64 同等の性能を得ることに成功した.

$\mu$ -Kernel 環境での性能改善手法の研究としては [4] がある. これは  $\mu$ -Kernel 環境において重要となる IPC の性能を向上させることによって  $\mu$ -Kernel の性能を向上するものであり, 本研究が利用した分岐命令の除去もこの研究では使われている.

## 8 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構築した VMM 環境の評価と性能改善を行った.

[1] で構築した VMM の性能をマイクロベンチマークである lmbench により測定したところネイティブ Linux から 3~11 倍の性能低下が確認できた. 本研究ではこの性能低下を改善するために, IA-32 のグローバルページ機構による性能改善手法とページテーブルシャドウ生成の際の分岐命令の除去, ページテーブルシャドウ生成回数を削減するためのページテーブルキャッシュを導入した. これにより, マイクロベンチマークにおいてレイテンシ 1.5~4 倍程度の速度性能で 2 レベル保護リング環境で VMM を構築することに成功した.

## 参考文献

- [1] 青柳信吾, 追川修一: “2 レベル保護リング環境での VMM 構築と評価”, ARC-171 EMB-3, 2007
- [2] Paul Barham, et. al.: “Xen and the Art of Virtualization”, SOSP 2003.
- [3] Ian Pratt, Keir Fraser, Steven Hand, Christian Limpach, Andrew Warfield: “Xen 3.0 and the Art of Virtualization”, Ottawa Linux Symposium, 2005.
- [4] Jochen Liedtke: “Improving IPC by Kernel Design”, 14th SOSP, 1993.
- [5] Intel Corporation: “Intel 64 and IA-32 Architectures Software Developer’s Manuals”, Intel Corporation, 2006.