Gandalf VMMにおけるShadow Pagingの実装と評価

伊藤 愛[†1] 追川 修一[†1]

これからの組み込みシステムでは，ユーザの選択したアプリケーションを動作させるとともに，安全で効率の良い実行環境を実現する必要がある．その解決策として，VMM は有効な手段である．組み込みシステムで VMM を動作させることによって，資源の効率利用，安全性の向上，信頼性の向上を実現することができる．そのため，マルチコア CPU 指向の軽量 VMM として，Gandalf を設計，実装してきた．本論文では，ゲスト OS 間のメモリ保護を実現するシャドウページングについて述べる．シャドウページングを利用することで，VMM がゲスト OS のメモリ利用を監視することができる．2 方式のシャドウページングを設計し，実装を行った．それぞれの方式について評価実験を行い，ネイティブな Linux や XenLinux との比較した．その結果，Gandalf が Xen より軽量に実現できていること，また，割込み応答性に対して軽微な影響で済んでいることが確認できた．

Implementation and Evaluation of Shadow Paging on Gandalf VMM

Megumi Ito[†1] and Shuichi Oikawa[†1]

While the provision of secure and reliable, yet efficient execution environments is a must for embedded systems, users' desire for using applications of their own choices is rapidly growing. In order to deal with both requirements, VMMs will be an answer. By using VMMs in embedded systems, we can effectively utilize the resources, improve safety and reliability. We designed and implemented a multi-core processor-oriented lightweight VMM, Gandalf. This paper focuses on shadow paging, which enables memory protection among guest OSes. A VMM can monitor the use of memory by guest OSes through shadow paging. We designed and implemented the two models of shadow paging. The results from benchmark experiments show that Gandalf performs better than Xen.

1. はじめに

これからの組み込みシステムでは，ユーザの選択したアプリケーションを動作させるとともに，安全で十分な実行環境を確保する必要がある．その解決策として，資源の効率利用，安全性の向上，信頼性の向上を実現することができる VMM（Virtual Machine Monitor）は有効な手段である．VMM を用いて 1 つのマシンで複数の OS（Operating System）を同時に実行させることによって，計算機資源を有効に活用することができる．VMM によって OS どうしは隔離されているので，アプリケーションの信頼度に応じて動作させる OS を分け，悪意のあるプログラムから情報を保護することで安全性を高めることができる．また，あらかじめ代替 OS とペアで動作させておき，実行中の OS に障害が発生したときにすぐに代替 OS に切り替えて実行することによって，可用性の高いシステムを構築することもできる．

組み込みシステムの特性として，専用化されたシステムであること，リソース制約があること，高い信頼性が求められること，リアルタイム性が求められることがあげられる[13]．このような特性を持つ組み込みシステムで仮想化技術を利用するうえで，重要な要求が大きく 3 つあげられる．まず，VMM 自体の軽量化の要求である．VMM を通した OS の実行にコストがかかりすぎてしまうと，1 つのマシンで複数の OS を同時に実行させることができるというメリットが失われてしまう．また，VMM のサイズが小さいことも重要である．限られた資源を使って動作するため，VMM が多くの資源を使用してしまうと OS の実行に影響してしまう．3 つ目の要求として，VMM 上で動作させるための OS の変更量をできるだけ少なくしたいという要求がある．多くの変更を加えれば，それだけ変更コードによるバグが発生する可能性が高くなる．少ない移植コストで OS を実行できるということは，今まで開発してきたプログラム資源の有効活用につながり，OS のバージョンアップに対応するのも容易になる．

†1 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba

これらの要求を受け，本研究ではこれまで，マルチコア CPU 指向の軽量 VMM として，Gandalf を開発してきた．組み込みシステムでのマルチコア CPU は開発が進みつつあり，将来は幅広く利用されると考えられる[7]．しかし，現在一般的に使用できる組み込みシステム向けのマルチコア CPU は存在しないため，すでにマルチコア CPU 化が進んでいる IA-32 アーキテクチャ[5] を対象として開発を行っている．Gandalf 上で動作させるゲスト OS として Linux を用いている．現在，2 つの CPU 上で 2 つのゲスト OS が動作しており，ゲスト OS は最低限の書き換えを行うだけで動作することができる[6]．

本論文では，ゲスト OS 間のメモリ保護を実現するシャドウページングの設計と実装，そのコストの評価について述べる．シャドウページングでは，ゲスト OS が使用するゲストページテーブルとは別に VMM 内にシャドウページテーブルを確保する．CPU はゲストページテーブルではなく，シャドウページテーブルを参照する．シャドウページテーブルに書き込むことができるのは VMM のみであるので，VMM がゲスト OS のメモリ利用を管理することができる．シャドウページングの方式として Single Shadow 方式と Multiple Shadow 方式の 2 方式を設計し，実装を行った．Multiple Shadow 方式では，シャドウページテーブル用のメモリを再利用するための Reclamation Policy として FIFO 方式と Page Directory LRU 方式を設計し，実装した．

ベンチマークプログラムの lmbench[8] を用いて評価実験を行い，シャドウページングのコストについて計測した．さらに，PMC（PerforMance Counter）による詳細な測定を行い，コストの原因について分析した．また，割込み応答性を評価するための実験を行った．ネイティブな Linux と Xen，Gandalf の No Shadow 方式についても同様の測定を行い，Gandalf のシャドウページングの各方式との比較を行った．その結果，Gandalf が Xen より軽量に実現できていることと，割込み応答性に対して軽微な影響で済んでいることが確認できた．また，Multiple Shadow 方式における各 Reclamation Policy のコストについて測定して比較を行った結果，システムの利用形態によって適切な Reclamation Policy を選択することが重要であることが分かった．

本論文の構成は以下のようになっている．2 章では，Gandalf の概要について述べる．3 章でシャドウページングの設計について述べ，4 章でその実装について述べる．5 章では実装したシャドウページングのコストの測定実験と比較を行い，6 章でその結果をもとにコスト要因と組み込みシステムへの対応について考察する．7 章で関連研究について述べ，8 章で本論文をまとめる．

2. Gandalf

本研究で開発しているマルチコア CPU 指向の軽量 VMM である Gandalf について述べる．まず組み込みシステムをターゲットとして VMM を開発するうえでの設計方針について述べ，続いて Gandalf のアーキテクチャ概要について述べる．

2.1 設計方針

組み込みシステムの特性として，専用化されたシステムであること，リソース制約があること，高い信頼性が求められること，リアルタイム性が求められることがあげられる[13]．組み込みシステムをターゲットとした VMM では，これらの性質を生かし，また制約や要求を満たすことが重要である．

組み込みシステムは，PC のような汎用システムと異なり，システム上で使用する OS をユーザが選択してインストールするようなことは通常ない．そのため，ゲスト OS をまったく変更なしに動作させることができるのは，必ずしも重要ではない．ゲスト OS を変更なしに動作させるためには，ゲスト OS の利用するハードウェアの機能をすべて VMM で仮想化してゲスト OS に提供する必要があり，VMM の実装コストと実行コストは非常に大きくなってしまう．それを避けるため，本研究では，VMM が必要最低限の機能のみ提供すればよいようにゲスト OS に少量の変更を加えることで，少ない実装コストと実行コストで VMM を用いたシステムを実現する．

また，組み込みシステムでは厳しいリソース制約があることが多い．コストを抑えるために，搭載されているプロセッサが低い性能のものであったり，メモリ量が少なかったりすることがある．そのため本研究では，少ない命令数で軽量な VMM を実現し，また，VMM の使用するメモリ量を少なく抑える．

組み込みシステムにおける重要な要求として，高い信頼性の実現がある．高い信頼性を実現するためには，ソフトウェアに含まれるバグの数はできるだけ少なくする必要がある．一定行数あたりのバグの数を 0 にすることはできないことが知られているが，少ないコード量で実装することでバグを少なくすることができる[10] ため，小規模な VMM であることと，ゲスト OS の変更量が少ないことが求められる．本研究では，ゲスト OS が必要とする最低限の機能のみを提供する

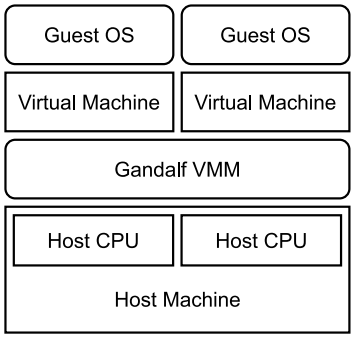


図 1 Gandalf VMM を含むシステムの構造図
Fig. 1 Structure of a Gandalf VMM based system.

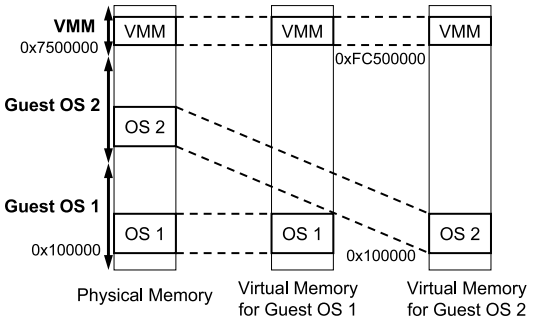


図 2 メモリマップ
Fig. 2 Memory map.

ことで簡素で小さな VMM を実現する．また，実装および実行コスト削減のためにゲスト OS に少量の自明な変更を加える．

さらに，組み込みシステムではリアルタイム性が求められており，定められた時間制約を満たすようにシステムが動作することが重要である．本研究では，少ない命令数で VMM を実現することで VMM による影響を抑える．

このように，組み込みシステムの特性に沿う VMM であるためには，VMM が軽量であること，小さな VMM であること，ゲスト OS の変更量が少ないことが求められている．これらの要求を満たす VMM を実現するため，上記の設計方針に従い，本研究ではこれまで，マルチコア CPU 指向の軽量 VMM として，Gandalf を開発してきた．次節でその概要について述べる．

2.2 Gandalf のアーキテクチャ概要

マルチコア CPU 指向の軽量 VMM である Gandalf のアーキテクチャ概要について述べる．Gandalf は，IA-32 アーキテクチャを対象とし，Gandalf 上で Linux を動作させることができる．**図 1** に，Gandalf を含むシステムの構造図を示す．ホストマシンとなる実機上で VMM である Gandalf が動作し，仮想的な計算機環境として仮想マシンを複数個生成する．仮想マシン上で動作させる OS のことをゲスト OS と呼ぶ．仮想マシンは独立しており，それぞれの仮想マシン上で個別にゲスト OS を動作させることができる．Gandalf では，実 CPU をそのまま 1 つの仮想 CPU として使用する．従来の手法では 1 つの実 CPU を多重化して複数の仮想マシンで共有するモデルが多かったが，この手法では仮想 CPU の管理と切替えにコストがかかってしまう．CPU 単位で資源を割り当てることによって，より軽量な VMM を作ることができる．これ以降，単に VMM といった場合には Gandalf を指す．

Linux は通常，存在する物理メモリをすべて使用して動作している．しかし，同時に VMM を動作させたり，複数の Linux を動作させたりするときには，物理メモリ領域を分け合いながら使用する必要がある．このため，**図 2** の左側に示すように，VMM には物理メモリの上部を割り当て，その残りの部分を分割してそれぞれの Linux に割り当てている．また，仮想メモリについても同様に，Linux は通常仮想アドレス空間のすべてを使用して動作しているが，VMM を同じ仮想アドレス空間で実行させるとき，Linux が VMM の仮想メモリ領域をアクセスできてしまうと問題である．そのため，図 2 の右側に示すように，Linux の使用できる仮想メモリ領域から VMM の使用する部分を除外している．なお，使用可能仮想メモリ領域の制限にはセグメント機構を利用しているが，簡単のため，VMM の使用メモリ領域は仮想アドレス空間の上端付近とした．

IA-32 アーキテクチャには，0 から 3 までの 4 段階の特権レベル（リング）が存在する．数が小さいほうが特権レベルが高く，大きいほうが特権レベルは低い．重要ないくつかの命令は特権命令と呼ばれ，特権レベルが 0 の状態でないと実行できない．**図 3** の左側に示すように，Linux は通常，カーネルをリング 0 で，ユーザプロセスをリング 3 で動作させることで，カーネルが特権命令を使用して CPU を管理し，また，ユーザプロセスからカーネルを保護している．本研究では，VMM が Linux を管理するために VMM を Linux より高い特権レベルで動作させる必要があるため，図 3 の右側に示すように，VMM をリング 0 で動作させ，Linux カーネルは 1 つ特権レベルの低いリング 1 で動作させる．リング 1 に移動させたことによって，Linux カーネルは特権命令を使用できなくなってしまうが，これらの特権命令は VMM で適切にエミュレーションを行う．Gandalf の特権命令エミュレータは，実際に

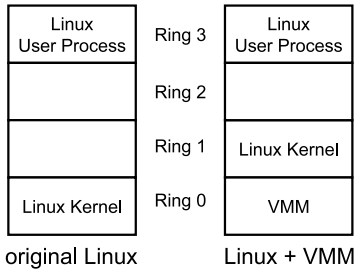


図 3 特権レベルの利用図
Fig. 3 Privilege level usage.

Linux で使用されている特権命令のみを処理するように設計されており，簡素な作りになっている．

3. シャドウページングの設計

ゲスト OS 間のメモリ保護を実現するシャドウページングの設計について述べる．まず，シャドウページングの概要について述べ，続いて本研究で設計したシャドウページングの詳細について述べる．本研究では，シャドウページテーブルを 1 つのみ使用する Single Shadow 方式と，複数のシャドウページテーブルを使用する Multiple Shadow 方式の 2 方式を設計した．また，シャドウページテーブルを使用しない No Shadow 方式についても述べる．

3.1 シャドウページングの概要

シャドウページングでは，ゲスト OS が使用するページテーブルであるゲストページテーブルとは別に，シャドウページテーブルと呼ばれるページテーブルを VMM 内に準備する．ゲスト OS 実行時には，CPU はゲスト OS 内のページテーブルではなく，シャドウページテーブルを参照する．シャドウページングの概要を**図 4** に示す．

ゲスト OS がゲストページテーブルに新しいエントリを追加し，追加したエントリに対応するアドレスにアクセスすると，そのエントリはまだシャドウページテーブル内に存在しないので，ページフォルトが発生する．VMM がページフォルトを処理する際に，ページフォルトが発生したアドレスに対応する新しいエントリがゲストページテーブルに追加されていれば，このエントリの内容を調べ，問題がなければシャドウページテーブルにコピーする．

ゲストページテーブルでのエントリの更新は，INVLPG 命令で検知する．IA-32 アーキテクチャでは，ページテーブルエントリを変更した際は必ず INVLPG 命令で TLB をフラッシュする必要がある[⋆1]．INVLPG

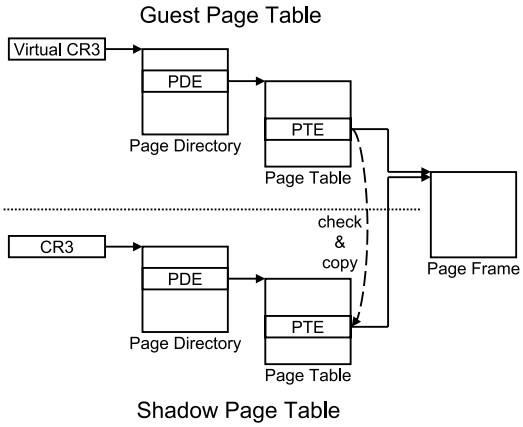


図 4 シャドウページング概要図
Fig. 4 Overview of shadow paging.

命令は特権命令であるため，ゲスト OS がこれを実行すると一般保護例外が発生し，VMM が呼び出される．VMM は INVLPG 命令をエミュレーションするときに，実際に INVLPG 命令を実行するとともに，対応するシャドウページテーブルのエントリを更新する．これにより，シャドウページテーブルの一貫性を維持できる．

ゲスト OS でのページフォルト処理中に新しくページエントリを追加した場合には，ハイパーコールによってこれを VMM に通知し，シャドウページテーブルにも同時にエントリを追加する．これにより，ゲストページエントリを追加した後，シャドウページテーブルを更新するためにもう 1 度ページフォルトが発生してしまうのを防ぐことができる．

シャドウページテーブルに書き込むことができるのは VMM のみであるので，シャドウページングを利用することで，VMM がゲスト OS のメモリ利用を監視することができる．VMM は，ゲストページテーブルのエントリの内容に問題があれば，そのエントリを無効化したり，エントリの内容を書き換えたりすることによって，メモリが正しく利用されていることを保証することができる．

3.2 Single Shadow 方式

Single Shadow 方式では，VMM は各ゲスト OS に対し 1 つのシャドウページテーブルを確保する．**図 5** に Single Shadow 方式の概要図を示す．ゲストページテーブル切替えの際には，シャドウページテーブルをフラッシュすることで，新しいシャドウページテーブルへの切替えを実現する．したがって，プロセス切替え後には再びゲストページテーブルのエントリをコピーする必要がある．ページテーブルの切替えが頻繁

⋆1 したがって，INVLPG 命令を持たない CPU はサポートしない．

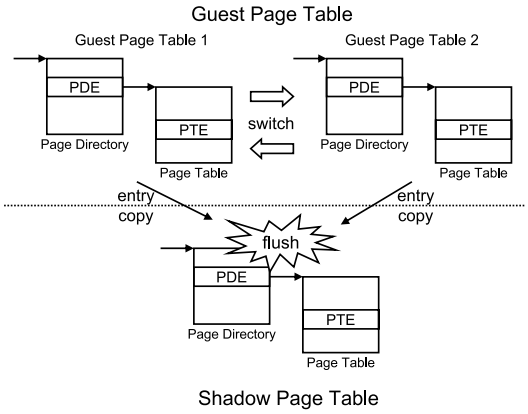


図 5 Single Shadow 方式
Fig. 5 Structure of Single Shadow Model.

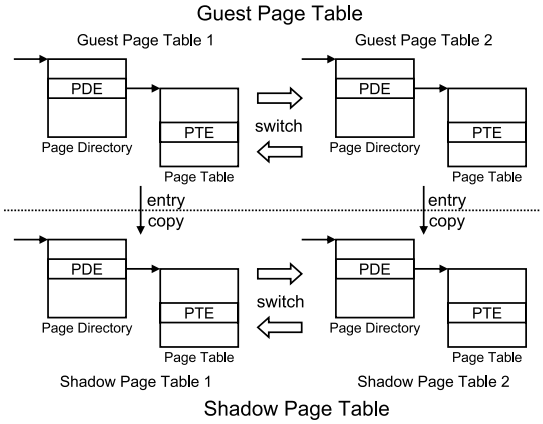


図 6 Multiple Shadow 方式
Fig. 6 Structure of Multiple Shadow Model.

に発生する場合では，このシャドウページテーブルフラッシュとその後のシャドウページテーブル更新のコストが重大な問題となる．しかし，Single Shadow 方式では，ゲスト OS ごとにシャドウページテーブルを 1 つしか用意する必要がないので，シャドウページテーブルの管理が簡単であり，また，実装が容易である．

3.3 Multiple Shadow 方式

Multiple Shadow 方式では，各ゲストページテーブルにそれぞれシャドウページテーブルを割り当てる．Multiple Shadow 方式の概要を**図 6** に示す．ゲストページテーブルごとにシャドウページテーブルが存在するため，ゲストページテーブルの切替えが起こった場合は，シャドウページテーブルの切替えを行うだけでよい．シャドウページテーブルのフラッシュは不要であり，ゲストページテーブルのエントリを再びコピーする必要もないため，ページテーブル切替えのコストを抑えることができる．

シャドウページテーブルに使用するメモリは，あらかじめ VMM 内に静的に確保しておき，このメモリを順にページディレクトリとページテーブルの区別なくシャドウページテーブルに割り当てていく．すべてのメモリが使用されてしまったら，Reclamation Policy によって決定されるページテーブルを解放し，そのメモリを再利用する．ページディレクトリを解放してしまうとシャドウページテーブルを再構築するコストが大きくなってしまうため，ページテーブルのみを再利用の対象としている．

3.3.1 Reclamation Policy

Multiple Shadow 方式で，再利用するシャドウページテーブルを決定する Reclamation Policy として，FIFO，Page Directory LRU の 2 方式を設計した．

FIFO 方式では，ページディレクトリに使用されているメモリを除き，ページテーブル単位で FIFO キューを作成する．新しくページテーブルを割り当てたら，FIFO キューの最後尾にページテーブルを追加する．シャドウページテーブルに使用するメモリが不足したら，この FIFO キューの先頭にあるページテーブルをフラッシュして解放し，新しいシャドウページテーブルに割り当てる．

Page Directory LRU 方式では，ページディレクトリ単位で LRU キューを作成する．シャドウページテーブルの切替えが起こったら，切り替えたシャドウページテーブルのページディレクトリを LRU キューの最後尾に移動させる．新しくシャドウページテーブルが作成された場合には，LRU キューの最後尾に追加する．メモリの再利用が要求されたら，LRU キューの先頭にあるページディレクトリで使用されているページテーブルをすべてフラッシュして解放する．

3.4 No Shadow 方式

比較のため，シャドウページテーブルを使用しない No Shadow 方式を設計した．この方式では，ゲスト OS が使用するゲストページテーブルを CPU が参照するページテーブルとして使用し，VMM 内ではシャドウページテーブルの管理は行わない．VMM のメモリ領域をマップするためのページエントリをゲストページテーブルに貼り付けて使用する．

No Shadow 方式では，CPU が参照するページテーブルをゲスト OS が直接変更できるため，ページテーブルの更新にコストがかからない．また，VMM 内にシャドウページテーブル用のメモリを確保する必要がないので，VMM が使用するメモリ量を抑えることができる．しかし，ゲスト OS が不正なエントリをページテーブルに追加したとしても，VMM はそれを検知

することができない．

4. シャドウページングの実装

3 章で述べたシャドウページングの設計に基づいた実装について述べる．現在の Gandalf の実装はコメントや空行を含めて 8,121 行であり，7,150 行の C プログラムと 971 行のアセンブラプログラムからなる．Linux の変更量は 47 カ所，101 行である．

4.1 ゲストページテーブルの参照

ゲスト OS から VMM に渡されるゲストページテーブルのアドレスは物理アドレスであるため，そのままこのアドレスにアクセスすることができない．そこで，VMM がゲストページテーブルを参照する場合には，VMM の使用する仮想メモリ領域内の固定アドレスに使用中のゲストページテーブルのページディレクトリをマップする．4 KB ページのマッピングのためにページテーブルを使用している場合は，そのページテーブルもマップする．

4.2 Single Shadow 方式

Single Shadow 方式の現在の実装では，シャドウページテーブル用にページディレクトリを 1 個，ページテーブルを 1024 個確保している．ページテーブルは 0 からのテーブル番号で管理されている．4 KB のページマッピングを管理するためにページテーブルが必要な場合は，仮想アドレスを 22 ビット右シフトした数に対応するテーブル番号のページテーブルを使用する．

4.3 Multiple Shadow 方式

Multiple Shadow 方式では，シャドウページテーブル用のメモリに 4 KB のページサイズ単位でテーブル番号を割り当て，この番号によってメモリを管理している．VMM ではこのメモリの使用状況を管理するテーブルを持っており，ページディレクトリとして使用中であるか，ページテーブルとして使用中であるか，あるいは使用されていないかを記録している．新しくページディレクトリやページテーブルとしてメモリを要求されたときにはまずこのテーブルを調べ，使用されていないメモリを探す．すべてのメモリが使用中であれば Reclamation Policy を呼び出し，再利用するページテーブルを探す．

シャドウページテーブルはそれぞれ固有のシャドウ番号によって管理されている．それぞれのシャドウページテーブルを管理するデータ構造体にはページディレクトリのテーブル番号とポインタが保存されている．またページテーブルのテーブル番号も保存されており，ページディレクトリのエントリを調べることなく高速にページテーブルにアクセスすることが可能である．ページテーブルの更新や解放の際はこのデータ構造体を利用してシャドウページテーブルへの書き込みを行う．

各シャドウページテーブルに対応するゲストページテーブルは，シャドウ番号によるテーブルで管理されている．VMM はこのテーブルにゲストページテーブルのページディレクトリのアドレスを保存しておき，ゲストページテーブルの切替え時に対応するシャドウページテーブルを検索するために使用する．

プロセス終了時には，ハイパーコールを利用して明示的にゲストページテーブルの解放を VMM に通知し，シャドウページテーブルを解放する．これにより，対応していたゲストページテーブルが新しく他のプロセスに割り当てられたときに，前のシャドウページテーブルを対応させてしまうことを防ぐことができる．解放されたメモリは，他のシャドウページテーブルに再利用される．

4.4 No Shadow 方式

No Shadow 方式では，シャドウページテーブルは存在しないため，ゲスト OS が引き起こしたページフォルトは，すべてゲスト OS に渡して処理を任せる．ページテーブルの操作に関連して，VMM は INVLPG 命令と CR3 レジスタへの書き込みをエミュレーションする．TLB フラッシュの INVLPG 命令が実行された場合には，単純に INVLPG 命令の代替実行のみを行う．ゲスト OS が CR3 への書き込みを行う命令が実行したときには，新しいゲストページテーブルに VMM をマップするページエントリを追加したうえで実際に CR3 レジスタへの書き込みを行う．

5. 実　　験

実装したシャドウページングのコストについて評価するため，シングルゲスト OS で実験を行った．実験には Dell Precision 490 を用い，CPU は Intel Xeon 5130 2.0 GHz を使用した．ゲスト OS として Linux 2.6.18 に必要な変更を加えたものを使用した．ベンチマークプログラムとして lmbench[8] を用い，シャドウページングのコストについて計測した．さらに，PMC（PerforMance Counter）による詳細な測定を行い，コストの原因について分析した．また，Multiple Shadow 方式における Reclamation Policy のコストについて比較するため，独自にベンチマークプログラムを作成し実験を行った．最後に，割込み応答性を評価するための実験を行った．Reclamation Policy のコスト比較を除き，ネイティブな Linux 2.6.18 と

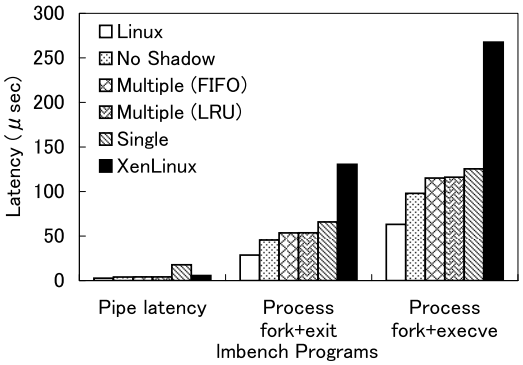


図 7 lmbench の測定結果の比較
Fig. 7 Comparison of results of lmbench.

Xen 3.1 の XenLinux についても同様の測定を行い，Gandalf との比較を行った[⋆1]．

5.1 マイクロベンチマークによる評価

ベンチマークプログラムとして lmbench プログラムの lat_pipe，lat_proc fork，lat_proc exec を用い，シャドウページングによる影響が大きい操作であるパイプレイテンシ，プロセスの fork と exit，プロセスの fork と execve のコストを計測した．Gandalf の Single Shadow 方式，Multiple Shadow 方式，No Shadow 方式の各方式についてコストを計測した．Multiple Shadow 方式では Reclamation Policy が FIFO 方式のものと Page Directory LRU 方式のものそれぞれについて計測した．また，ネイティブな Linux と XenLinux についても同様の測定を行い，Gandalf との比較を行った．**図 7** にその計測結果を示す．

パイプレイテンシの計測では，Single Shadow 方式のコストが突出して大きくなっている．これは，パイプによるプロセス間通信を行うためにプロセスの切替えが起こったときに，シャドウページテーブルのフラッシュとエントリの再コピーが発生しているためである．

プロセスの fork と exit，また，プロセスの fork と execve では，シャドウページングを利用している場合，子プロセスのページテーブルエントリをシャドウページテーブルにコピーするコストが発生する．Single Shadow 方式では，親プロセスが使用していたシャドウページテーブルのフラッシュが行われる分，さらにコストが大きい．No Shadow 方式では，シャドウページテーブルの管理コストがないため，シャドウページングを利用した場合に比べてコストが少ない．プロセスの fork と exit よりプロセスの fork と execve でシャドウページングのコストが増えているのは，後者でアクセスするアドレス範囲がより広いためにゲストページエントリのコピーがより多く発生するためである．Xen では，paravirtualization によって高速化を図っているものの，Gandalf のどの方式よりも大きなコストがかかっており，Gandalf が軽量に実現できていることが分かる．

Gandalf でのシャドウページング方式に着目すると，Single Shadow 方式は，ゲストページテーブルの切替えが発生した際のシャドウページテーブルフラッシュとエントリの再コピーのコストがかかるため，一番コストが大きくなっている．しかし，どの lmbench プログラムでも Multiple Shadow 方式とのコスト差は変わらないため，フラッシュと再コピー以外のコストは発生していないと考えられる．Single Shadow 方式は実装や管理が容易であり，使用するメモリ量も少なく抑えることができるので，実行コストより使用メモリ量が重要な場面では有効な方式であるといえる．

Multiple Shadow 方式では，Reclamation Policy が FIFO 方式のものと Page Directory LRU 方式のものについて測定した．FIFO 方式の方が Page Directory LRU 方式より若干コストが低いものの，ほぼ同じ結果となっている．この測定ではページテーブルの再利用は行われていないため，Reclamation Policy によるコスト差は各方式のキュー操作のコスト差のみに起因する．FIFO 方式ではページテーブルの割当てか解放が発生したときにしか FIFO キューの操作を行わないが，Page Directory LRU 方式ではシャドウページテーブルが切り替わるたびに LRU キューの操作を行う．実行コストが重要な場合は Multiple Shadow 方式の方が有効であると考えられる．特にプロセスの切替えが多く発生してシャドウページテーブルが頻繁に切り替えられる場合は，Reclamation Policy として FIFO 方式を採用した方がよい．

No Shadow 方式とネイティブな Linux とのコスト差は，Gandalf における仮想化のコストであると見ることができる．FIFO 方式の Multiple Shadow 方式と No Shadow 方式とのコスト差は，シャドウページングの実現にかかるコストであるといえる．lmbench プログラムの測定結果から算出したこれらのコストがネイティブな Linux に対してどの程度のオーバヘッドになっているのかを**表 1** に示す．

仮想化のオーバヘッドは，50%前後である．これには特権命令のエミュレーションや，ページフォルトを 1 度 Gandalf で受けているコストが含まれている．

⋆1 Xen 3.1 の XenLinux は Linux 2.6.18 を使用しており，バージョンを揃えるため，Gandalf，ネイティブな Linux でも同じバージョンを用いて実験を行った．

表 1 仮想化とシャドウページングのオーバヘッド（%）
Table 1 Overheads of virtualization and shadow paging.

lmbench プログラム	仮想化	シャドウページング
Pipe latency	46.4	3.6
Process fork+exit	59.2	27.7
Process fork+execve	55.2	27.2

シャドウページングによるオーバヘッドは 30%以下である．プロセスの fork と exit を行うときにはシャドウページテーブルの割当てと解放やシャドウページテーブルの切替えが発生するために Gandalf でのオーバヘッドが大きい．しかし，execve を行うときにはエントリのコピーを行うだけであり，fork と exit に比べてオーバヘッドが少ない．そのため，fork と execve を行う場合は fork と exit を行う場合よりもオーバヘッドが減少している．

特に，シャドウページテーブルの切替えを行うだけのパイプレイテンシ計測では，3.6%と非常に少ないオーバヘッドでシャドウページングを実現することができている．プロセス操作では，操作の違いによらずほぼ一定の値を保っており，一定のオーバヘッドでシャドウページングを実現できていることが分かる．

5.2 PMC による分析

前節で測定した pipe，fork+exit，fork+execve の各 lmbench プログラムと同等のプログラムを作成し，これについて PMC を用いて詳細な分析を行った．ネイティブな Linux，Gandalf 上の Linux，XenLinux についてそれぞれ測定し，比較した．Gandalf は，Multiple Shadow 方式を用い，Reclamation Policy として FIFO 方式を使用しているものについて測定した．TSC（Time Stamp Counter）は計測していた間の時間を表す．これには PMC 計測にかかった時間も一部含まれるため，図 7 とは多少異なる結果となっている．

表 2 に測定した PMC のイベントを示す．なお，PMC には L1D_MISSES というイベントはないため，代わりに L1D_REPL の値を用いた．今回の実験では，プログラムサイズが L2 キャッシュに収まるサイズであり，L2 キャッシュミスは発生しなかったため，その発生回数は測定していない．PMC はリング 0 でのイベント数のみを計測することも可能であるため，Gandalf や Xen のみのイベント数も計測した．これによって，ゲスト OS としての Linux のイベント数も知ることができる．

5.2.1 Pipe Latency

パイプレイテンシ測定のプログラムにおける PMC による測定結果を，ネイティブな Linux の測定結果を 1 として正規化したものを，**図 8** に示す．実行命令数

表 2 PMC のイベント
Table 2 PMC events.

イベント名	内容
INST_RETIRED.ANY.P	実行命令数
RESOURCE_STALLS.ANY	ストール発生回数
L1I_MISSES	L1 命令キャッシュミス数
L1D_MISSES	L1 データキャッシュミス数
ITLB_MISSES	命令 TLB ミス数
DTLB_MISSES	データ TLB ミス数
PAGE_WALKS.COUNT	ページテーブルウォーク数
SEGMENT_REG_LOADS	セグメントレジスタロード回数
CYCLES_INT_MASKED	割込み禁止サイクル数

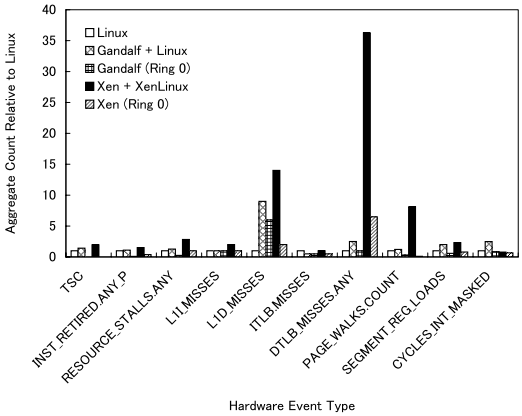


図 8 PMC を用いた測定結果—pipe
Fig. 8 Results of PMC—pipe.

は，レイテンシにほぼ比例した結果となっている．つまり，そもそも実行している命令数に差があるので，コスト差が発生しているということがいえる．また，ストール数も実行命令数と似たような結果となっており，これもコスト差の要因の 1 つであると考えられる．ストール数が増えている原因と考えられるのが，L1 データキャッシュのミスと，データ TLB のミスである．特に XenLinux ではデータ TLB のミスが多く，この処理に多くの時間を費やしてしまっている．これは，Xen が多くのデータ構造体を処理する必要があり，データの局所性が低くなっていることを示している．

ここで，Gandalf と Xen の実行命令数の差について詳細に考察する．lmbench における pipe のコスト測定では，2 プロセス間に 2 つのパイプを作成し，それぞれのプロセスが読み出し用のパイプから 1 バイト読んで書き込み用のパイプに 1 バイト書き込むという作業を繰り返している．このうち，片方のプロセスがパイプに書き込んでから読み込み終わるまでの時間を計測し，パイプを使ったプロセス間通信にかかるレイテンシとしている．

Gandalf 上の Linux でこのようなパイプを利用し

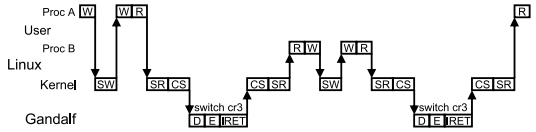


図 9 パイプを利用したプロセス間通信の実行パス
Fig. 9 Execution path of interprocess communication through pipes.

表 3 パイプを利用したプロセス間通信における実行命令数
Table 3 Numbers of executed instructions in interprocess communication through pipes.

	Native Linux	Gandalf	Xen
Linux	4245	4263	4842
VMM	0	400	1589
Total	4245	4663	6431

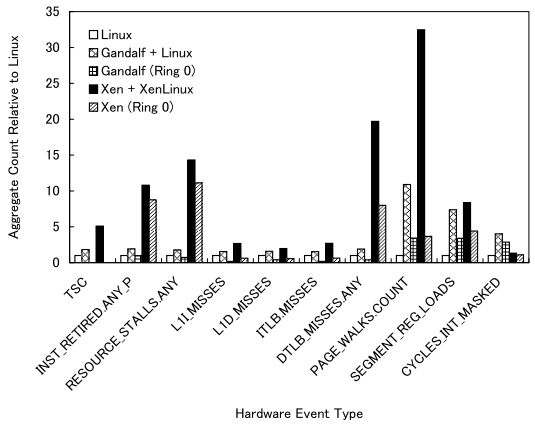


図 10 PMC を用いた測定結果— fork+exit
Fig. 10 Results of PMC–fork+exit.

たプロセス間通信を行ったときの実行パスを**図 9** に示す．各プロセスの W はパイプへの書き込みを実行中であり，R はパイプからの読み出しを実行中であることを示す．W 内ではシステムコール write を，R 内ではシステムコール read をそれぞれ発行する．Linux カーネルの SW，SR はそれぞれシステムコール write，read を処理中であることを示し，CS はコンテクストスイッチを行っていることを示す．Linux カーネルはコンテクストスイッチ処理の一部で CR3 レジスタへの書き込みを行うことによりページテーブルの切替えを行う．CR3 レジスタへの書き込みを行う命令は特権命令であるため，一般保護例外が発生し，Gandalf がエミュレーションを行う．Gandalf は D で一般保護例外を発生させた命令のデコードを行い，E でその命令のエミュレーションを行う．すべての処理が終了したら，IRET によって Linux カーネル内に戻る．Xen 3.1 でも同様のパスを通る．

表 3 に lmbench によるパイプを利用したプロセス間通信を行ったときに実行される命令数を示す．Gandalf の VMM 内での実行命令数について，実際にソースコードの該当部分のアセンブラでの命令数を数えたところ，実験結果とほぼ一致した．VMM 内での実行命令数では，Xen は Gandalf に比べて 4 倍多くの命令を実行している．これは，図 9 における D や E での実行命令数が多いためである．特に，Xen の特権命令エミュレータは，Intel VT のようなハードウェア仮想化支援機構にも対応するために多くの特権命令を処理する必要があり，Gandalf のものより大きく複雑になっている．そのため，D にかかる命令数が多い．Gandalf の特権命令エミュレータは，実際に Linux で使用されている特権命令のみを処理するように設計されており，簡素な作りになっている．また，E で CR3 レジスタへの書き込みをエミュレーションするとき，Xen では実際に CR3 の更新を行うほかに仮想 CPU を管理するデータ構造体の操作を行っており，これもオーバヘッドとなっている．Gandalf では仮想 CPU は管理していないので，対応するシャドウページテーブルが見つかれば，そのアドレスを CR3 に書き込むだけで済む．

さらに，注目すべきなのは，Xen を利用した場合に Linux 自身の実行命令数も増加している点である．Xen では Linux に大量の変更を加えているため，実行命令数が増加してしまっている．Gandalf を利用した場合にも実行命令数が増加しているが，これはごく少量である．Gandalf では Linux への変更量を少なく抑えているため，ネイティブな Linux とほぼ同じ命令数で実行することができる．

5.2.2 プロセス操作

プロセスの fork+exit，fork+execve の各プログラムにおける測定結果を，ネイティブな Linux の測定結果を 1 として正規化したものを，それぞれ**図 10**，**図 11** に示す．

Xen では，総実行命令に対して Xen の VMM 部分が占める割合が大きい．これは，プロセスの生成などを行った際に発生するシャドウページングの処理や，仮想 CPU の管理にコストがかかっているためであると考えられる．XenLinux 自身の命令実行数も多く，ネイティブな Linux の 2 倍程度になっている．一方 Gandalf では，Linux の変更量を少量にとどめているため，Gandalf 上の Linux の実行命令数はネイティブな Linux と同程度に抑えられている．Xen では Linux のメモリ管理部分を多く変更しているため，実行命令数が増加していると考えられる．

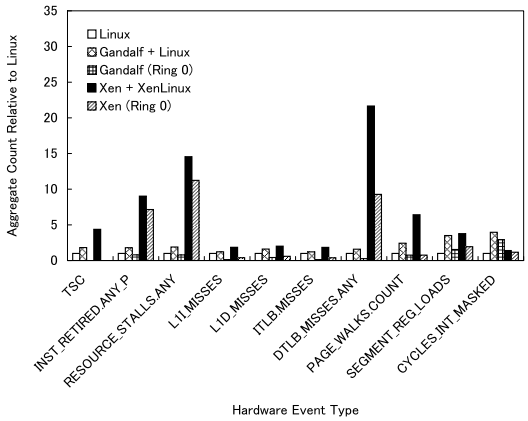


図 11 PMC を用いた測定結果—fork+execve
Fig. 11 Results of PMC—fork+execve.

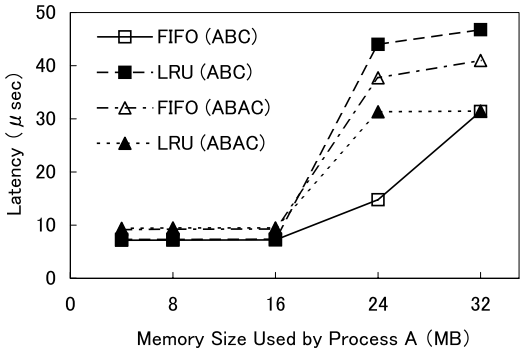


図 12 Reclamation Policy の比較
Fig. 12 Comparison of reclamation policies.

L1 キャッシュミスや TLB ミスにおいては，すべてで Xen が Gandalf を上回る結果となっている．特に Xen ではデータ TLB ミスの多さが顕著に現れている．これは，Xen がより多くのデータ構造体にアクセスし，より多くの処理を行っていることの証拠である．Xen では仮想 CPU の管理などのために Xen 内部で多くのデータ構造体を使用しており，これらの管理に多くのコストがかかっている．TLB ミスが発生するとページテーブルウォークが起こるため，さらに大きなオーバヘッドが発生する．

プロセスの fork+exit の結果では，プロセスの fork+execve の結果に比べて，ページテーブルウォークの回数が Gandalf，Xen ともに非常に多い．これは，lmbench の測定結果を裏付けるものである．

5.3 Reclamation Policy 比較

Multiple Shadow 方式における Reclamation Policy である FIFO 方式と Page Directory LRU 方式のコストについて評価するため，lmbench に準じてベンチマークプログラムを作成し，コストの測定を行った．ベンチマークプログラムでは，3 つのプロセス A，B，C が，それぞれ指定された大きさのメモリ領域に順番にアクセスする．プロセスが実行される順序は，ABC の順に実行が繰り返されるものと，ABAC の順に実行が繰り返されるものについて測定した．ABC の順序は大小のプロセスが混在して実行される場合を模擬したもの，ABAC の順序は 1 つのプロセスが主に実行される場合を模擬したものとなっている．ABC，もしくは ABAC を 1 セットの実行単位とし，1 セット実行するのにかかったレイテンシを計測した．メモリアクセスのオーバヘッドを最小限に抑えつつ確実にページテーブルを消費するため，メモリへのアクセスは 4 MB 間隔で行う[⋆1]．各プロセスがアクセスするメモリ領域のサイズは 4 MB 単位とし，プロセス A には 4，8，16，24，32 MB のメモリサイズを，プロセス B と C にはそれぞれ 4 MB の固定なメモリサイズを指定した．そのため，プロセス A は 1 から 8 個，プロセス B と C はそれぞれ 1 個ずつのページテーブルをこれらのメモリアクセス時に使用することになる．また，ページフォルトハンドラ中で Reclamation Policy が呼び出される場合のページフォルトアドレスを調べたところ，各プロセスは，それぞれのカーネル部分とプログラム部分のために，計 2 個ずつのページテーブルを必要としていた．したがって，3 つのプロセス全体で使用するページテーブルは，9 から 16 個になる．Reclamation Policy が呼び出されるようにするため，使用可能なシャドウページテーブルのページテーブルを 13 個とし，測定を行った結果を**図 12** に示す．

プロセス A のアクセスするメモリサイズが 16 MB までの場合はページテーブルの再利用は発生していない．プロセス A が 16 MB のメモリにアクセスする場合，プロセス A，B，C 全体で必要とするページテーブルは 12 個である．これは使用可能なページテーブル数以内であるため，再利用を行わずにすべてのプロセスを実行することができる．したがって，この場合の Reclamation Policy によるコスト差は，各方式のキューの管理コストのみとなる．このコスト差は，プロセスの実行順序やプロセス A がアクセスするメモリサイズによらずほぼ一定であり，Page Directory LRU 方式の方が FIFO 方式より約 2% 多くキュー管理コストが発生している．FIFO 方式ではページテーブルが新たに使用されたり再利用が発生したりしたと

⋆1 ページサイズが 4 KB，各ページテーブルは 1024 エントリ含まれるため，1 ページテーブルあたり 4 MB の仮想メモリ領域を管理できる．

きにのみキューの操作を行うため，再利用が発生しない場合はキューの管理コストは発生しない．逆に Page Directory LRU 方式では，シャドウページテーブルが切り替わるたびにキューの操作を行うため，ページテーブルの再利用が発生しなくてもキューの管理コストが発生する．

一方，プロセス A が 24 MB 以上のメモリにアクセスする場合には，合計で 14 個以上のページテーブルを必要とするため，ページテーブルの再利用が発生し，Reclamation Policy の違いによるコスト差が大きく出る．プロセスが ABC の順序で実行される場合は，FIFO 方式の方が Page Directory LRU 方式より圧倒的に良い結果となっている．Page Directory LRU 方式では各プロセスの実行が再開されるたびに 1 回ずつの計 3 回 Reclamation Policy が呼び出され，再利用の対象とされたプロセスで使用可能なすべてのページテーブルが再利用される．FIFO 方式について，ページフォルトハンドラ中で Reclamation Policy が呼び出される場合のページフォルトアドレスを調べたところ，その順序が実行のたびに異なるということが分かった．これは，プロセス起動時の実行順序が不定であるためにメモリへのアクセス順序が異なり，ページテーブルが割り当てられてキューに入れられる順序が変化することが原因である．ベンチマークプログラムによる測定が開始されてからは決められた順序でプロセスが実行されるが，キューに入っているページテーブルの順序が異なるため，再利用される順序が異なる．その順序により，1 セット実行中にすべての使用可能なページテーブルが再利用されることもあれば，2 セット実行しないとすべてのページテーブルが再利用されないこともある．後者の場合は 1 セットで再利用されるページテーブル数が少なくなり，1 セットあたりの実行コストは低くなるため，測定結果にばらつきが生じた．そのため，FIFO 方式についてはベンチマーププログラムを 100 回実行し，その平均を測定結果とした．Page Directory LRU 方式ではページテーブル自体の順序は関係ないため，このような結果にはならない．このため，FIFO 方式の方がコストが低い結果となっている．

Page Directory LRU 方式の方がより大きなコストが発生しているもう 1 つの原因として考えられるのが，再利用するページテーブルの探索手法である．現在の実装では，ページテーブルのテーブル番号は 1024 要素を持つ配列に使用されているアドレスに対応して格納されており，この配列をリニアサーチすることによって再利用するページテーブルを探している．このリニアサーチが Page Directory LRU 方式の大きなオーバヘッドとなっていると考えられる．シャドウページテーブルごとに使用しているページテーブルのキューを作ってページテーブルの探索時間を削減することによって，Page Directory LRU 方式全体のコストを削減することができると考えられる．

ABAC の順序でプロセスが実行される場合は，逆に Page Directory LRU 方式の方が FIFO 方式より低い実行コストとなった．プロセス A が 24 MB 以上のメモリにアクセスする場合，ABC の 3 つのプロセスで必要なページテーブルは 14 から 16 個だが，AB もしくは AC の 2 つのプロセスが必要とするページテーブルは 11 から 13 個であり，利用可能なページテーブル数に収まる．そのため，ABAC の順番で実行した場合には，Page Directory LRU 方式ではつねにプロセス B と C の間でページテーブルの再利用が行われ，プロセス A のページテーブルは確保されたままで，再利用されることはない．しかし，FIFO 方式では使用可能なすべてのページテーブルが順番に再利用されるため，Page Directory LRU 方式より多くのコストがかかる．

以上のように，Reclamation Policy とプロセスの実行順序の組合せによって，シャドウページングのコストに差が出るということが分かった．複数のプロセスが順番に実行されるような場合は FIFO 方式を，ある 1 つのプロセスが主に実行されるような場合は Page Directory LRU 方式を用いることで，より少ないコストでシャドウページングを利用することができる．システムの利用形態によって適切な Reclamation Policy を選択することが重要である．

5.4 割込み応答性

組み込みシステムにおいて重要な割込み応答性について評価を行う．Linux の RTC（Real-Time Clock）デバイスを利用し，割込みが発生し Linux カーネルの割込み処理ハンドラが動き出してから，割込み待ちをしていたプロセスの実行が再開されるまでのレイテンシを計測した．**表 4** に測定結果を示す．

Gandalf では，ネイティブな Linux と比べると 4.8% 遅いものの，Xen より早く割込みに対応することができている．Xen ではすべての割込みを 1 度 Xen で受けてから，必要なものについては Linux へ割込みの通

表 4 割込み応答性の測定結果
Table 4 Results of interrupt latency.

	Native Linux	Gandalf	Xen
TSC counts	21868	22908	23902
latency (μsec)	10.93	11.48	11.98

知を行うため，Linux が割込みを処理するまでに時間がかかる．Gandalf では実 CPU を 1 つの仮想 CPU として使用する構造であるため，割込みはすべて直接 Linux へ通知でき，このコストがかからない．

しかし，5.2 節での PMC による計測結果では，割込み禁止サイクル数（CYCLES_INT_MASKED）で唯一 Gandalf が Xen を上回る結果となっている．これは，Gandalf が Linux で発生した例外を処理するために割込み禁止状態で実行されるためである．Gandalf 上の Linux の変更量は非常に少ないため，その割込み禁止サイクル数はネイティブな Linux とほぼ同じである．XenLinux はネイティブな Linux よりも大幅に少ないサイクル数となっている．Xen 自体の割込み禁止サイクル数も，Gandalf より大幅に少ない．割込み禁止サイクル数については，6 章で詳しく考察する．

6. 考　　察

5 章での実験結果をふまえ，Gandalf の実行コストと割込み応答性について考察する．また，割込み禁止区間に着目し，その削減方法について考察する．さらに，本論文で述べたシャドウページングの設計の IA-32 アーキテクチャへの依存性について議論する．

6.1 実行コストと割込み応答性

実験結果より，Gandalf は Xen よりも軽量に実現できていることが分かった．Gandalf は簡素で容易な設計になっており，少ない命令数で動作することができる．Xen では実 CPU を複数の仮想 CPU で利用するため仮想 CPU の管理が複雑であり，多くの処理を必要とする．使用しているデータ構造体も多く，これらの管理に大きなコストがかかる．多くのデータ構造体にアクセスして多くの処理を行うということは，単純に実行命令数が増えるだけではなく，キャッシュミスや TLB ミスを引き起こす要因にもなる．キャッシュや TLB を埋めるためにはさらにコストがかかる．TLB ミスが起こった場合にはページテーブルウォークが行われるので，さらにオーバヘッドが発生する．Gandalf では実 CPU をそのまま 1 つの仮想 CPU として使用するため，複雑なデータ構造体による管理を必要としない．また，Xen では Linux を多く変更することによって Linux 内部での処理も増加してしまっているが，Gandalf では Linux の書き換えを少量に抑えているため，ネイティブな Linux とほぼ同じコストでゲスト OS として Linux を動作させることができる．

Gandalf では，割込みはすべて直接ゲスト OS である Linux が受けることができる．Gandalf では実 CPU を 1 つの仮想 CPU として使用する構造であるため，1 度 Gandalf が割込みを受けて Linux に割込みを割り振る必要がない．IA-32 アーキテクチャでは，非特権モードである特権レベルが 1 の状態でも割込みハンドラを実行することができるので，Linux の割込みハンドラをあらかじめ登録しておくことによって，直接 Linux の割込みハンドラをハードウェアによって実行することができ，軽量に割込みを処理することができる．その結果，割込み応答性に対しても軽微な影響で済んでいる．

6.2 割込み禁止区間の削減

Gandalf は Xen より軽量に実現できているため，より組み込みシステムでの利用に向いた VMM であるといえる．しかし，割込み禁止区間は Xen より Gandalf の方が長くなってしまっている．Xen は割込みを仮想化し，それに対応するように Linux を書き換えることで，XenLinux の割込み禁止区間を削減している．その結果，Xen 自身の割込み禁止区間はネイティブな Linux と同程度であり，また，XenLinux の割込み禁止区間はわずかであることが，PMC の測定結果より分かる．一方，Gandalf では Linux はネイティブな Linux と同程度の割込み禁止区間を持ち，さらに，Gandalf 自身も割込み禁止状態で動作する．これを改善する方法として，Xen のように Linux の割込み禁止区間を減らし，Gandalf を割込み可能とする方法がある．

Gandalf を割込み可能とするためには，Gandalf で例外のネストをサポートできるようにすることが必要である．**図 13** に，ユーザプロセスでページフォルトが発生して Gandalf のページフォルトハンドラが処理を行っている最中に Linux の割込みハンドラが呼び出された場合の実行パスを示す．割込み可能な状態で Gandalf が実行中に割込みが発生して Linux の割込みハンドラが呼び出されると，実行中の特権レベルより低い特権レベルで動作する割込みハンドラが起動されることになるため，一般保護例外が発生する．また，Linux の割込みハンドラから Gandalf に復帰しようとするときも，特権の高いレベルへの復帰になるため，一般保護例外が発生する．これらの例外を Gandalf の

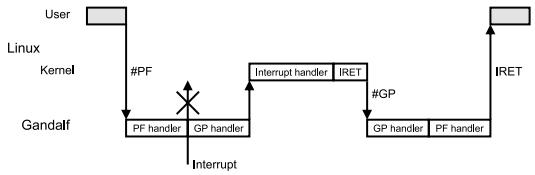


図 13 例外処理のネスト
Fig. 13 Nest of exception handling.

一般保護例外ハンドラでエミュレーションし，Linuxの割込みハンドラの呼び出しとGandalfへの復帰を行うことによって，Gandalf実行中にLinuxの割込みハンドラを起動することができるが，そのためには例外のネストをサポートすることが必要である．

Linux実行中に割込みがかかると一般保護例外が発生せずにLinuxの割込みハンドラが呼び出されるという点については変更がないため，軽量な割込み処理はそのままに，割込み禁止区間を短くすることが可能である．

6.3 IA-32アーキテクチャへの依存性

本研究では，開発対象としているIA-32アーキテクチャに依存した設計や実装も存在するが，シャドウページングを実現する本質的な技法としては，ページテーブルを使用する他のプロセッサにも適用できる．特にプロセッサアーキテクチャに依存する部分として特権レベルがある．本研究ではIA-32アーキテクチャの4つの特権レベルのうちの3つを使用してVMMを実装しているが，特権レベルが2つしか使用できない場合のVMMについても研究しており，実現可能であることを示している[12]．

シャドウページングに関しては，それを実現するために重要な条件は3つある．最も重要な条件として，ページテーブルエントリが存在しなかったり不正であったりする場合，そのエントリが示すアドレスにアクセスしたときに例外が発生する，という条件がある．この例外をVMMで受け，その処理中でゲストページテーブルからシャドウページテーブルへのエントリのコピーを行ったり，エントリの無効化を行ったりする．この機能は，仮想メモリを実現するために必要なものであり，IA-32アーキテクチャ以外のプロセッサでも実現されている機能である．

また，ゲストOSによるページテーブルアドレスの設定や読み出しをVMMが検知し，VMMがその操作をエミュレーションできるという条件がある．シャドウページングでは，システムが参照するページテーブルはゲストページテーブルではなくシャドウページテーブルであるので，ゲストOSがページテーブルアドレスを設定しようとしたときには代わりにシャドウページテーブルアドレスを設定してやる必要がある．また，ページテーブルアドレスを読み出そうとしているときは，設定されているシャドウページテーブルアドレスではなく，ゲストOSが設定しようとしたゲストページテーブルアドレスを返してやる必要がある．IA-32アーキテクチャではCR3レジスタへページテーブルアドレスを設定することでプロセッサがページテーブルを認識するが，CR3レジスタへの書き込みや読み出しを行う命令は特権命令となっている．このため，ゲストOSがCR3レジスタにアクセスすると一般保護例外が発生し，これをVMMでハンドルすることでエミュレーションを行うことができる．このような命令をゲストOSが実行したときに例外が発生しないようなプロセッサでは，ゲストOSがハイパーコールを実行するように書き換えることで同等の機能を実現することができる．

さらに，ゲストページテーブルのエントリの更新を検知して，VMMがシャドウページテーブルの更新を行うことができるという条件がある．ゲストページテーブルエントリが更新された場合には，それに合わせてシャドウページテーブルエントリも更新し，新しいマッピングを使用する必要がある．IA-32アーキテクチャでは，エントリを更新した後にTLBをフラッシュする特権命令のINVLPG命令を実行することになっているため，この命令をエミュレーションするときにシャドウページテーブルの更新も行っている．これも，同等の命令が存在しなかったり，特権命令ではなかったりする場合には，ゲストOSにハイパーコールを追加することによって条件を満たすことができる．

組み込み用途に広く用いられているARMプロセッサを例として，本論文で述べたシャドウページングの適用について考察する．ARMプロセッサ[11]のMMUはページテーブルを使用する．無効エントリへのアクセスはMMU Fault（Translation Fault）を起こし，ページテーブルアドレスを格納するTTB（Translation Table Base）レジスタやTLBフラッシュのためのレジスタ（CP15 Register 8）にアクセスするための命令（MCR，MRC）は特権命令でありユーザモードでの実行は未定義命令例外を起こす．したがって，上記の3つの条件を満たすため，ハイパーコールを用いることなく，本論文で述べたシャドウページングを適用することができる．

7. 関連研究

これまでの研究でもシャドウページングは利用されてきたが，その設計や実装については，詳しく述べられているものは少ない．

Single Shadow方式は，文献5)[⋆1]で紹介されているシャドウページングの実現手法である．シャドウページテーブルを1つだけ使用するため，管理が容易であり，

⋆1 Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide 26.3節 Memory Virtualization

実装コストを低く抑えることができる．しかし，ゲストページテーブルが切り替わるたびにシャドウページテーブルのフラッシュと大量のエントリコピーを行わなければならないため，実行時のオーバヘッドが大きくなる．そのため，実際のシステムで Single Shadow 方式のようなシャドウページングを使用することは少ないと考えられる．

Multiple Shadow 方式の関連研究として，Xen3.0 [9]，Xen3.1 [4]，VMware VMI [1] があげられる．Multiple Shadow 方式のように複数のシャドウページテーブルを使用する方法では，シャドウページテーブルの管理が複雑になり実装コストが大きくなるが，オーバヘッドは少なく抑えることができる．そのため，多くのシステムが複数のシャドウページテーブルを使用してシャドウページングを実現している．

Xen 3.0 でのシャドウページテーブルの更新手法は他と大きく異なる．Xen 3.0 では，ゲストページテーブルにゲスト OS が書き込めないようにあらかじめ read-only に設定しておく．ゲスト OS がゲストページテーブルを書き換えたいときにはページフォルトが発生するので，その時点でのゲストページテーブルの状態を保存しておき，そのテーブルを out of sync list に加えたうえでゲスト OS が書き込めるように変更する．その後，ゲスト OS が TLB をフラッシュするような命令を実行したら，out of sync list のテーブルの内容をシャドウページテーブルに反映し，再びゲストページテーブルを read-only に設定する．Gandalf ではゲストページテーブルを read-only にはせず，ページテーブルエントリの更新時ではなく対応するページでのページフォルトが発生した時点でシャドウページテーブルの更新を行っている点が異なっている．

Xen 3.1 のシャドウページングは Xen 3.0 から大きく変更され，ゲスト OS によるゲストページテーブルへのすべての書き込みをエミュレーションしている．ゲストページテーブルへ書き込むと同時にハイパーコールが呼び出され，シャドウページテーブルへの書き込みを行う．VMware VMI でも，paravirtualization の実現のために Xen 3.1 と同様の機構のシャドウページングを使用している．Gandalf でも，ゲストページエントリが存在しないためにページフォルトが発生した場合は，ゲスト OS がページテーブルへの書き込みを行う際にシャドウページテーブルへも書き込むようにしている．これは，ゲストページテーブルを更新した後に，シャドウページテーブルを更新するためのページフォルトが起こるのを避けるためである．しかし，ページフォルト処理以外でゲストページテーブルへ書き込んでいる場合はシャドウページテーブルの更新を行わないので，アクセスが発生したときにシャドウページテーブルに反映するという方針に変わりはなく，使用しないページエントリをシャドウページテーブルに書き込むコストを削減することができる．

Multiple Shadow 方式では，メモリを再利用するための Reclamation Policy として，FIFO 方式と LRU 方式という異なる 2 つの方式について設計し，実装した．このように複数の Reclamation Policy が存在し，用途に適した方式を選択して使用可能である点は関連研究と異なる．

本論文では，実装したシャドウページングの各方式について，マイクロベンチマークによりそのコストを測定し比較した．これまでこのように複数のシャドウページングの実現方式について比較が行われたような研究はなく，本論文でシャドウページングのコスト要因について明らかにすることができた．

8. まとめと今後の課題

本研究でこれまで開発してきたマルチコア CPU 指向の軽量 VMM である Gandalf において，ゲスト OS 間のメモリ保護を実現するシャドウページングについて述べた．

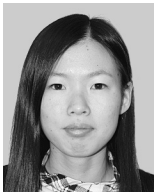
Single Shadow 方式と Multiple Shadow 方式の 2 方式のシャドウページングを設計し，実装を行った．Multiple Shadow 方式では，メモリ再利用のための Reclamation Policy として FIFO 方式と Page Directory LRU 方式を設計し，実装した．評価実験を行い，シャドウページングの各方式とネイティブな Linux，Xen や No Shadow 方式との比較を行った．その結果，Gandalf はゲスト OS への変更量が少なく簡素な構造になっているため，Xen より軽量に実現できていることが分かった．また，割込み応答性に対して軽微な影響で済んでいることが確認できた．さらに，Multiple Shadow 方式における各 Reclamation Policy のコストについて測定して比較を行った結果，システムの利用形態によって適切な Reclamation Policy を選択することが重要であることが分かった．

今後は，Gandalf の割込み禁止区間の削減に努める．割込み禁止区間を削減することで，より高い割込み応答性の実現を目指す．また，マルチコア CPU 対応のゲスト OS が実行できるように Gandalf の拡張を行う．マルチコア CPU 対応のゲスト OS を実行するためには VMM で APIC の仮想化を行う必要があり，その実現方法が大きな課題となる．

参 考 文 献

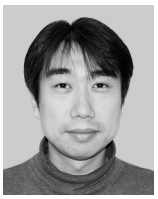
1) Amsden, Z., Arai, D., Hecht, D., Holler, A. and Subrahmanyam, P.: VMI: An Interface for Paravirtualization, *Proc. 2006 Linux Symposium*, Vol.2, pp.363–378 (July 2006).
2) Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the Art of Virtualization, *Proc. 19th ACM Symposium on Operating System Principles*, pp.164–177 (Oct. 2003).
3) Creasy, R.J.: The Origin of the VM/370 Time-Sharing System, *IBM Journal of Research and Development*, Vol.25, No.5 (1981).
4) Deegan, T. and Fetterman, M.: Shadow2, *Xen Summit* (Sep. 2006).
5) Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual.
6) Ito, M. and Oikawa, S.: Mesovirtualization: Lightweight Virtualization Technique for Embedded Systems, *Proc. 5th IFIP Workshop on Software Technologies for Future Embedded & Ubiquitous Systems* (May 2007).
7) Kamei, T.: SH-X3: An Enhanced SuperH Core for Low-power MP Systems, *Fall Microprocessor Forum 2006* (Oct. 2006).
8) McVoy, L. and Staelin, C.: lmbench: Portable Tools for Performance Analysis, *Proc. USENIX Annual Technical Conference*, pp.279–294 (Jan. 1996).
9) Pratt, I., Magenheimer, D., Blanchard, H., Xenidis, J., Nakajima, J. and Liguori, A.: The Ongoing Evolution of Xen, *Proc. 2006 Linux Symposium*, Vol.2, pp.255–266 (July 2006).
10) Gray, J. and Siewiorek, D.P.: High-Availability Computer Systems, *IEEE Computer*, Vol.24, No.9, pp.39–48 (Sep. 1991).
11) Seal, D.: *ARM Architecture Reference Manual, 2nd ed.*, Addison-Wesley (2001).
12) 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，情報処理学会研究報告，2007-OS-105, Vol.2007, No.36, pp.55–62 (2007).
13) 高田広章：組込みシステム開発技術の現状と展望，情報処理学会論文誌，Vol.42, No.4, pp.930–938 (2001).

(平成 19 年 7 月 23 日受付)
(平成 19 年 11 月 7 日採録)



伊藤　愛

昭和 60 年生．平成 18 年筑波大学第三学群情報学類卒業．現在，筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻博士前期課程に在学中．オペレーティングシステム，仮想マシンモニタに関する研究に従事．



追川　修一（正会員）

昭和 40 年生．平成 8 年慶應義塾大学より博士（工学）．平成 16 年筑波大学大学院システム情報工学研究科助教授に着任．現在，筑波大学大学院システム情報工学研究科准教授．オペレーティングシステム，仮想マシンモニタに関する研究に従事．IEEE，USENIX，電子情報通信学会各会員．