情報処理学会論文誌　コンピューティングシステム　Vol.5　No.1　27–40 (Jan. 2012)

WinKVM：Hybrid Hypervisor移植の一例

高橋 一志[1,a)]　笹田 耕一[1,b)]

受付日 2011年5月11日, 採録日 2011年8月6日

概要：我々は, Linux/KVM の Windows 移植版である WinKVM（Windows Kernel-based Virtual Machine）を構築した．WinKVM は KVM のすべての機能を移植できており，WinKVM と KVM 間での VM ライブマイグレーションすら可能である．現在のクラウドコンピューティング環境では Windows を使用するユーザが，Linux サーバ上で動作する VM をリモートで使用することは一般的である．しかし，このような環境では，ユーザが使用する VM がサーバに固定されるため，VM をローカルの Windows マシン上にすばやく転送することは困難である．OS の違いの壁を乗り越えて，リモートにある VM をローカルにすばやく転送する技術は有用である．WinKVM を開発するにあたって，我々は，Windows 上で Linux カーネルを模倣するエミュレーションレイヤを構築し，KVM をその上で動作させることで，KVM を Windows に移植した．これにより，Windows と Linux 間でのライブマイグレーションを達成することができた．いくつかのベンチマークプログラムで WinKVM の性能を測定したところ，オリジナルの KVM との性能差はみられず，エミュレーションレイヤによる性能劣化はないことが分かった．

キーワード：仮想化，VMM（Virtual Machine Monitor），KVM，移植

WinKVM: An Alternative Example of Porting Hypervisor

Kazushi Takahashi[1,a)]　Koichi Sasada[1,b)]

Received: May 11, 2011, Accepted: August 6, 2011

Abstract: We achieve VM live-migration between Windows and Linux by developing *WinKVM*, which is a port of Linux/KVM to Windows. WinKVM has all functions of KVM and even it has ability to migrate VM between KVM and WinKVM. Nowadays, in the cloud-computing environment, Windows users are able to use remotely the VM on Linux servers via the Internet. In such environments, however, VMs are fixed in server-side and we could not quickly transmit them to client-side. We believe that it is important to develop the technique that enables users to migrate VMs from server-side to client-side whenever they want. Therefore, we develop the live migration between WinKVM and Linux KVM. To implement WinKVM, we develop an emulation layer that allows us to emulate Linux kernel functions on Windows kernel. Accordingly, we achieve the VM live migration between Linux and Windows. In terms of benchmark, WinKVM comes very close to original KVM performance and our emulation layer gives no overhead.

Keywords: virtualization, VMM (Virtual Machine Monitor), KVM, porting

1. はじめに

仮想マシンモニタ（VMM: Virtual Machine Monitor）の成熟にともない，計算機の利用者に対し，データセンタ内にある VMM が管理する仮想マシン（VM: Virtual Machine）を割り当てるシステムが普及している [1]．このような基盤システムでは，VM はデータセンタ内に固定されるため，ユーザはリモートデスクトップや SSH を使って，ネットワーク越しに VM を操作する必要がある．しかし，計算機の利用形態は多種多様であるため，リモートデスクトップや SSH を用いたネットワーク越しでの利用でそれらすべての要求をかなえることは不可能である．たとえば，高い

[1] 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo, Chiyoda, Tokyo 113–8656, Japan
[a)] kazushi@rvm.jp
[b)] ko1@rvm.jp

対話応答性が求められる CAD ソフトウェアや動画編集ソフトウェアを利用するために一時的に手元の計算機資源を活用したいという要求や，ネットワークが使えない環境下でも VM を使用したいという要求に対して対応することができない．

そこで，我々は必要なときにはリモートにある VM 自体をシームレスに手元のコンピュータに移動させることができればより便利になると考えた．このような技術には，VM ライブマイグレーションという技術がすでに存在するが，主に管理者用の技術として使われており，エンドユーザ向けの技術としては使われていない．そのため，エンドユーザが使用する OS（Windows）とサーバ用途向けの OS（Linux）間のライブマイグレーションを実現するソフトウェアは有用であると考えられる．

この目標を実現するために，我々は，Windows と Linux 間のライブマイグレーションを実現する，Hybrid 型 Hypervisor である Linux/KVM を Windows 上に移植した．移植した Windows 版 KVM である WinKVM は KVM と WinKVM 間でライブマイグレーションが可能である．移植は，Windows 上に Linux カーネルを高速にエミュレーションするエミュレーションレイヤを開発し，その上で KVM を動作させることで行った．

なお，ライブマイグレーションは，NFS や Samba 等のネットワークファイルシステムを用いて，VM のディスクイメージが転送先と転送元で共有されていることを前提とする．もちろん，エンドユーザが NFS や Samba 環境を整え，ライブマイグレーションを行うことは困難であると考えるが，現行の VMM では，NFS や Samba を使用しなくても，VM のディスクイメージごと VM をマイグレーションする技術の開発も進んでいる [2], [3], [4], [5]．そのため，本研究では VM のディスクイメージの共有については言及せず，従来どおり，NFS や Samba を用いた VM ライブマイグレーションを想定する．

本論文の主眼は，Linux 上で動作する KVM を Windows 上で動作させる一手法を述べることにある．我々の開発したエミュレーションレイヤは Linux カーネルを高速にエミュレーションすることが可能であり，アーキテクチャがまったく異なる Windows と Linux 間での Hypervisor 移植を実現可能であることを示す．また，Linux エミュレーションレイヤを作る過程で直面した，開発環境の違いや，両 OS のメモリアーキテクチャの根本的な相違等，いくつかの困難な課題をどのように解決にしたかについて論じ，それらの課題を解決する過程で得た，Windows と Linux で動作する移植性のある Hybrid 型 Hypervisor を設計するうえでの知見について述べる．Windows と Linux の両 OS 上で動作する Hybrid 型 Hypervisor はすでに存在しているものの，こういった Hypervisor を設計するための知見はあまり文章化されていない．さらに，いくつかのベンチマークを使用し，開発したエミュレーションレイヤ上で動作する WinKVM がオリジナルの KVM と比べてほとんど性能劣化を引き起こさないことを示す．

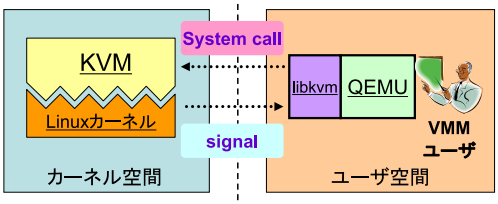


図 1　KVM のアーキテクチャ
Fig. 1　The overview of KVM architecture.

2. KVM のアーキテクチャ

WinKVM の設計と実装を理解するためには，まず，KVM の構造を理解する必要がある．本章では，KVM のアーキテクチャのうち WinKVM と関連性の強い箇所について重点的に解説する．

図 1 に示すように KVM はカーネル空間で動作する KVM ドライバとユーザ空間で動作する QEMU [6] から構成されている．KVM ドライバは特権命令である VT-x [7] や AMD-SVM [8] を操作しゲスト OS を動作させる．ユーザ空間にある QEMU は主に VM のデバイス（VGA や NIC）のエミュレーションを行う．ゲスト OS を実行するときには QEMU 側から KVM ドライバ側にシステムコールを発行し，そのシステムコールを受けて KVM ドライバが VT-x や AMD-SVM を操作し，ゲスト OS を実行する．

ゲスト OS が動作するためにはゲスト OS が使用する実メモリが必要である．これは KVM のドライバ側で確保する．ドライバ側で確保する理由は最終的にゲスト OS 用の実メモリを物理アドレスに変換する必要があるためである．これは VT-x や AMD-SVM の仕様上必要なことである．Linux では，メモリアドレスを物理アドレスに変換するためには OS がドライバのみにアクセスを許す API を使用する必要がある．そのため，KVM ドライバ内のカーネル空間で，ゲスト OS 用のメモリを確保する必要がある．

KVM ドライバが確保したゲスト OS 用の実メモリは，KVM ドライバと QEMU 側で共有される．QEMU はこのドライバに対して `mmap()` システムコールを発行することでゲスト用の仮想メモリ領域を確保する．KVM ドライバは Linux のメモリ管理機構の機能を使用して QEMU 側で確保された仮想メモリ領域と KVM ドライバ内で確保された実メモリ領域を対応付ける．これにより，QEMU 側からゲスト OS の仮想メモリ領域へと書き込まれた内容はただちに KVM ドライバ側にも反映される．

3. WinKVM の移植手法

図 2 に WinKVM のアーキテクチャを示す．KVM を移

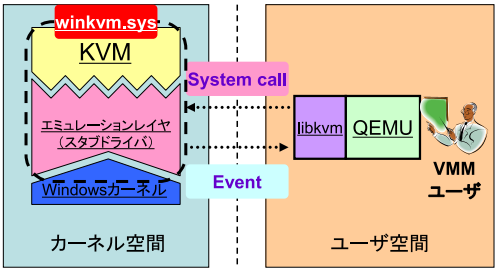


図 2　WinKVM のアーキテクチャ
Fig. 2　The overview of WinKVM architecture.

植するために，我々は，Windows 上に Linux カーネルをエミュレーションするレイヤを構築しその上で KVM を動作させた．開発したエミュレーションレイヤと KVM ドライバをあわせてリンクして最終的に Windows のドライバとして動作する WinKVM を得た．

なお，KVM を構成する要素にはそれぞれ若干の修正が必要である．本手法は Windows 上に Linux カーネルを模倣するエミュレーションレイヤを構築することで KVM ドライバを修正することなく Windows 上で動作させることを目標としている．しかし，エミュレーションレイヤではどうしても KVM を動作させることができない部分もあるため，KVM ドライバに数行の修正を加える必要があった．これについては 3.2 節と 3.4 節で述べる．また，KVM ドライバ以外にも，ユーザ側で動作する QEMU と libkvm が存在しており，こちらも Windows 上で動作させる必要がある．QEMU は Windows 上で動作させるために改良を施した QEMU が存在するため移植する必要はない．しかし，libkvm に関しては，KVM ドライバと通信する部分であるため，システムコールを呼び出す部分を変更する必要がある．具体的には，`ioctl()` を呼び出している部分を，Windows 用の `ioctl()` である `DeviceIoControl()` 関数に置き換える必要がある．

WinKVM を開発するにあたって KVM-17 を使用した．KVM-17 は VM のライブマイグレーションを実装した初めてのバージョンであり，最新の KVM バージョンに比べて比較的構造が単純である．そのため，プロトタイプの実装としては最適である．

Linux カーネルをエミュレーションするレイヤを開発するために，まず，我々は KVM ドライバがどのような Linux カーネル内関数を使用しているかを調査した．次に，調査した Linux カーネル内関数を Windows 上でエミュレーションする関数を記述して，エミュレーションレイヤを開発した．

エミュレーションレイヤは Linux カーネル関数を Windows カーネル関数に変換して実行する．ほとんどの関数は対応する Windows カーネル関数へと 1 対 1 で変換可能である．そのため，エミュレーションのための処理をほとんど加えずに，Windows 上で KVM を実行することが可能である．**表 1** に Linux カーネル関数と Windows カーネル関数の変換表を示す．

カーネル内でのメモリ割当て　Linux のすべてのメモリアロケータは `ExAllocatePoolWithTag()` に変換される．Linux カーネルでは `kmalloc()`, `vmalloc()` といった 2 つのメモリアロケータが存在するが，これら 2 つメモリアロケータは `ExAllocatePoolWithTag()` に変換して問題はない．

カーネル内でのメモリ開放　メモリ開放のための関数はすべて `ExFreePoolWithTag()` に変換される．上述したとおり，Linux のメモリアロケータはすべて `ExAllocatePoolWithTag()` に変換されるため，Windows ではこの関数に対応するメモリ開放関数である `ExFreePoolWithTag()` を使用する．

ミューテックスとスピンロック　ミューテック関数やスピンロックのための関数は表に示すとおりに変換される．特に難しい点はない．

ページアロケータ　Linux カーネルとは異なり，Windows には `alloc_pages()` のような，ページを割り当てるための専用アロケータは存在しない．その代わり，`ExAllocatePoolWithTag()` の第 1 引数に PAGE_SIZE を指定することで空ページを得ることができる．そのため，ページアロケータにも同様に `ExAllocatePoolWithTag()` を使用する．ページを開放するときには `ExFreePoolWithTag()` を使用する．

アドレス変換　`__va()` と `__pa()` は C マクロである．`__va()` は物理アドレスを仮想アドレスに変換するためのマクロであり，`__pa()` は仮想アドレスを物理アドレスに変換するためのマクロである．マクロと関数の差はあるものの，表記のとおりに変換して問題ない．

CPU 数取得　SMP 環境では，CPU 数取得のための関数は表記の関数に変換される．

メモリバリア　メモリバリアとは今まで行われたすべてのメモリに対する書き込みを実メモリに確定させるためのものである．これらの関数は表記のとおり変換される．

3.1　エミュレーションレイヤを構築する際の課題

エミュレーションレイヤを構築するためには，次の 3 つの課題がある．

ドライバ動作モードにおける RPL の違い　KVM はハードウェア仮想化支援機構である VT-x や AMD-SVM を使用している．VT-x にはゲスト OS からホスト OS への復帰ポイントを設定するためのフィールドが存在する．その中の，ホスト OS の ES レジスタと DS レジスタの値の RPL フィールドと呼ばれる部分が 0 でなければ，VT-x はゲスト OS を開始することができない．Linux の場合は ES レジスタと DS レジスタの RPL の

表 1　Linux のカーネル内関数と Windows のカーネル API 対応表
Table 1　The list of Windows kernel APIs that are associated with emulating Linux kernel functions.

API の説明	Linux のカーネル関数	変換に利用される Windows の API
カーネル内のメモリ割当て	kmalloc() kzalloc() vmalloc()	ExAllocatePoolWithTag()
カーネル内のメモリ開放	kfree() vfree()	ExFreePoolWithTag()
排他制御 （ミューテックス/スピンロック）	mutex_init() mutex_lock() mutex_unlock() mutex_trylock()	ExInitializeFastMutex() ExAcquireFastMutex() ExReleaseFastMutex() ExTryToAcquireFastMutex()
ページアロケータ	alloc_pages() free_pages()	ExAllocatePoolWithTag() ExFreePoolWithTag()
アドレス変換	__va() __pa()	MmGetVirtualForPhysical() MmGetPhysicalAddress()
アクティブな CPU の個数を得る	get_cpu() get_nr_cpus()	KeGetCurrentProcessorNumber() KeQueryActiveProcessorCountCompatible()
メモリバリア	smp_wmb() smp_mb()	KeMemoryBarrier()

値は 0 であるため問題は発生しないが，Windows の場合，これらの値が 3 になっている．そのため，このままでは VT-x での仮想化を使用することができない．

KVM コードの gcc 依存　KVM のソースコードは Linux のコンパイラである GNU gcc に強く依存している．一方で，Windows のデバイスドライバを開発するためには Microsoft 製のコンパイラ（VisualC++等）を使用しなければならない．KVM コードを Microsoft 製のコンパイラでコンパイルすることはできず，また，gcc を使って Windows のドライバを開発することもできない．この開発環境のギャップを埋める必要がある．

ドライバのメモリアーキテクチャの違い　Windows と Linux はドライバのアーキテクチャが根本的に異なる．特に，両 OS がドライバに対して提供するカーネルとユーザ間で同一メモリ空間を共有するメモリマッピング機構の性質の違いがエミュレーションレイヤを構築するにあたっての障害となる．

本章では，上記の 3 つの課題に対する解決策について詳しく論じていく．

3.2　ドライバ動作モードにおける RPL の違いの解決手法

我々はこの問題を，RPL を 3 から 0 に低下させた DS 値と ES 値をゲスト OS からホスト OS への復帰ポイントに設定することで解決した．KVM のホスト復帰ポイントを設定する部分を改造し，ホスト OS の ES と DS の RPL を 3 から 0 に書き換えた値を，ホスト復帰ポイントとして設定するようにした．つまり，ゲスト OS 実行前の DS と CS の RPL 値は 3 であるが，ゲスト OS が実行された後では，その値が 0 になることになる．

この手法では致命的な問題点は発生しない．引き下げられた RPL の値は，Windows の割込み時には 0 から 3 に戻るため，永続的に 0 にはならない．RPL は Requester's Privilege Level の略で不適切なシステムコール呼び出しを回避するためのものである．たとえば，特権レベル 0 のセグメンテーション A と，特権レベルが 3 のセグメンテーション B，C，合計で 3 つ（A，B，C）のセグメンテーションを使用するオペレーティングシステムが存在するとして．特権レベルが 3（B，C）のプログラムが特権レベル 0 上（A）のセグメンテーションにあるメモリデータを直接盗み出すことはできない．しかし，特権レベル 0（A）にシステムコール相当（たとえばコールゲート）が存在すると，そのコールゲートを経由することで，特権レベルが 3（C）で動作しているプログラムが，特権レベルが 3 である（B）に特権レベル 0（A）のメモリデータを書き込み，そのうえで，特権レベルが 3（C）のプログラムが特権レベル（B）のデータを読むことができる．つまり，特権レベルが 3 であるプログラムがシステムコール相当（コールゲート）を経由して特権レベル 0 のメモリデータを読めてしまうのである．RPL はこれを防止するためのものであり，システムコールの発行元の特権レベルを OS がチェックできるようにするためのものである．

しかし，現在のオペレーティングシステムは，セグメンテーションによる保護機構は使用しないのが一般的であり，主に，ページ単位による保護機構を利用している．RPL はセグメンテーションを活用するオペレーティングシステム上で利用されるためのものであるため，ページ単位による

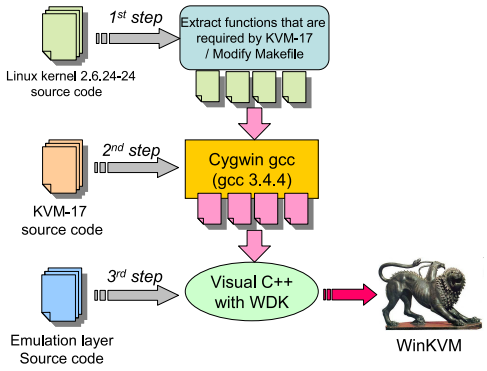


図 3　KVM コードの gcc 依存に対する解決手法
Fig. 3　How to solve compiler gap.

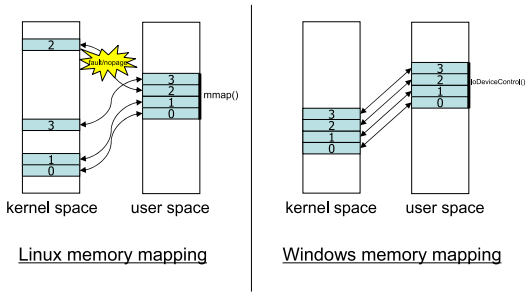


図 4　Linux と Windows のメモリマッピングの違い
Fig. 4　The difference between Linux and Windows memory mapping mechanisms.

保護機構を使用している Windows では特に大きな問題は発生しないと考える．

なお，Windows 用の VMM である VirtualBox も我々の手法と同様のことを行って PRL の問題点を解決している．ゲスト OS からホスト OS への復帰ポイントには RPL が 0 になったホスト OS のセグメント値が代入されるため，ゲスト OS からホスト OS に VMexit が発生したときには，一時的に VirtualBox 内の ES，DS が 0 である復帰ポイントに戻ることになる．この復帰ポイントが実行される間は割込みを禁止しておき，VirtualBox 以外のプログラムがいっさい実行できない状態にする．そのあとで，退避させておいた Windows の元の値である ES 値と DS の値を復帰しホスト OS に戻るという形をとっている．

仮に，RPL が 0 に修正されたことによるセキュリティリスクが発生するとしても，WinKVM は VirtualBox と同様の手法でこれを回避することが可能である．そのため，今回のようにプロトタイプの実装においては致命的な問題は発生しない．

3.3　KVM コードの gcc 依存に対する解決手法

我々は，Visual C++と Cygwin gcc を併用することでこの問題を解決した．初めに KVM のソースコードを Cygwin gcc でコンパイルする．すると，COFF バイナリで KVM のバイナリフォーマットが出力される．我々のエミュレーションレイヤは Visual C++で書かれているが，COFF バイナリ形式でコンパイルされた KVM は Visual C++でリンクすることができる．このように，gcc 依存である KVM のソースコードに修正を加えることなく，Visual C++を使って KVM を Windows 用のドライバとしてビルドすることが可能である．

図 3 に WinKVM のビルドプロセスを示す．ビルドプロセスは 3 ステップに分けて行われる．

第 1 ステップ，Linux ドライバのコンパイルにはカーネルのヘッダファイルが必要である．そのため，KVM-17 で使用されているカーネル関数のヘッダファイルを Linux カーネルのソースコードから抽出した．また，カーネルドライバのコンパイル時には `genksym` とよばれるプログラムが実行され，これはドライバのソースファイルがコンパイルされた後に特殊なシンボルを付与する後処理である．WinKVM のビルドにあたってこの後処理は不要であるため，カーネルの Makefile から `genksym` を走らせる処理を削除した．

第 2 ステップとして，Linux カーネルから抽出したヘッダファイルと改造した Makefile とをあわせて，Cygwin gcc を用いてコンパイルする．コンパイルすると，COFF 形式でコンパイルされた KVM バイナリが手に入る．COFF バイナリは VisualC++で解釈可能なバイナリであるため，次のステップにつなげることができる．

第 3 ステップは，我々の書いたエミュレーションレイヤのプログラムと先ほど COFF 形式で吐き出された KVM のバイナリを VisualC++であわせてコンパイルとリンクを行う．

以上の処理を行うことで，KVM のコンパイル済みバイナリとエミュレーションレイヤをリンクし，WinKVM を得る．

3.4　メモリアーキテクチャの違いの解決法

上述したように，KVM はカーネル側で動作する KVM ドライバと，ユーザ側で動作する QEMU 側の 2 つに分けることができる．この 2 つのコンポーネントはゲストメモリ空間を共有している．しかし，Windows と Linux がドライバに対して提供するカーネルとユーザ間で同一メモリ空間を共有するメモリマッピング機構の性質の違いがエミュレーションレイヤを構築するにあたっての障害となる．

本節では，我々はこのメモリ共有メカニズムの差異をエミュレーションレイヤでどのように吸収するかついて論じる．

Linux のカーネルとユーザ間のメモリ共有メカニズムの概念図を**図 4** の左に示す．図にあるように，Linux ではカーネル側で確保した非連続のページ領域を，ユーザ側の連続メモリ領域（`mmap()` で確保した領域）にマッピング

することが可能である．

プログラマはドライバ内の fault/nopage ハンドラを使って，カーネル内ページがどのユーザメモリ領域に関連付けられるかを指定することができる．たとえば，図 4 の左図にもあるように，mmap() で確保した 4 ページ分（0 番〜3 番ページ）のメモリがあるとして，このとき，ユーザ側のメモリ領域から 2 番ページにアクセスがあったとき，Linux カーネルは fault ハンドラ/nopage ハンドラを呼ぶ．このときプログラマは，2 番ページがどのカーネル内ページに割り当てられるかを指定することができる．結果的に，連続したユーザメモリ領域を非連続のカーネルメモリ領域に割り当てることができる．KVM-17 はこの fault/nopage ハンドラを KVM ドライバと QEMU 間のゲストメモリ共有に使用している．そのため，エミュレーションレイヤを構築するためには，この fault/nopage ハンドラをエミュレーションする必要がある．

一方，Windows のカーネルユーザ間のメモリ共有メカニズムの概念図を図 4 の右図に示す．Windows の場合，Linux では可能だった，非連続のカーネルメモリと，連続領域のユーザメモリのマッピングが不可能である．Windows では連続したカーネルメモリ領域を，同じく連続したユーザメモリ領域に割り当てることしかできない．どのユーザメモリ領域に，どのカーネルメモリ領域が割り当てられるのかは，Windows カーネルが暗黙的に決めてしまうのである．つまり，Linux ではあった fault ハンドラ/nopage ハンドラに相当するものが Windows には存在せず，エミュレーションレイヤ上でこのハンドラを高速にエミュレーションすることが難しい．さらに，1 度にマッピングできるメモリ共有領域にも制限があり，たとえば，2 GBytes の物理メモリを搭載するマシンでは 1 度に 300 MB のメモリ共有領域しか作ることができない．すなわち，WinKVM では VM に割り当てられるメモリ領域が制限されてしまう．

このように，Windows と Linux 間ではユーザ空間とカーネル空間とのメモリ共有に大きな違いがあるため，2 つの問題が発生する．1 つ目は，KVM が使っている fault/nopage ハンドラを高速にエミュレーションすることができないという問題点である．このため，ゲスト領域とホスト領域のメモリマッピングが行われず，ゲスト OS を正しくエミュレーションすることが不可能になる．2 つ目は，Windows 上でのメモリマッピングが 1 度に 300 MB 分しか作れないため，WinKVM の VM に割り当てられるメモリ領域が制限されてしまうという問題点である．

1 つ目の問題点である fault/nopage ハンドラはエミュレーションレイヤを使用して高速にエミュレーションすることが不可能であるという結論に至った．そのため，KVM 側のソースコードを修正し，KVM 側のコードが fault/nopage ハンドラを使用しなくてもカーネル側とユーザ側のメモリマッピングが可能になるように KVM のソースコードを修正することでこの問題を解決した．むろん，ゲスト OS をソフトウェア的にエミュレーションする VMM である以上，低速な手法でこの fault/nopage ハンドラをエミュレーションすることは可能である．たとえば，ゲスト OS のすべてのメモリアクセスをトレースし，エミュレータ上ではページが割り当てられていないと見なされたメモリ領域にアクセスが起こったとき，fault/nopage ハンドラを呼び出すといった手法が考えられる．しかし，この手法は著しいパフォーマンスの低下を引き起こすことが予想され，現実的な手法ではない．我々の今回の手法は，エミュレーションレイヤにより，Linux カーネルのカーネル関数と Windows のカーネル関数を 1 対 1 で変換することにより，Windows カーネル上で高速な Linux カーネルのエミュレーションを行うことである．そのため，fault/nopage ハンドラを使わなくてもエミュレーションレイヤ上で KVM が動作できるように KVM を改造する手法をとった．

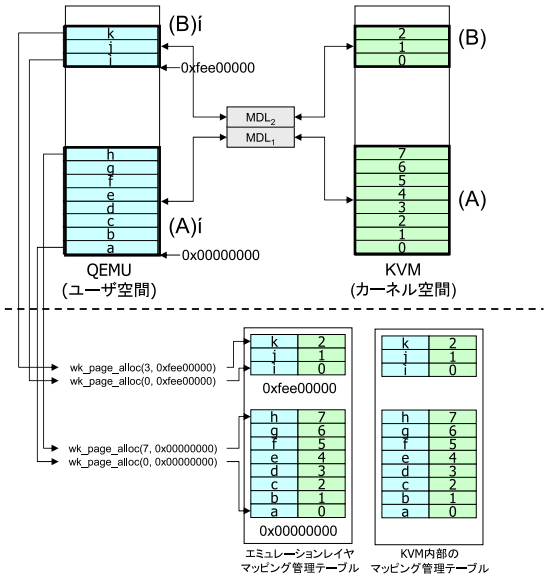


図 5　エミュレーションレイヤの機構
Fig. 5　The mechanism of our emulation layer.

2 つ目の問題点を解決するためには，QEMU 側のソースコードと KVM のソースコードを根本的に書き直す必要がある．そのため，今回我々が行ったエミュレーションレイヤを用いた WinKVM 実装ではこの問題を解決できなかった．しかし，この 2 つ目の問題点は Windows と Linux 間で動作する Hypervisor を構築する際に留意すべき点についての知見を我々にもたらしてくれる．これに関する詳細については 4 章で述べる．

はじめに，1 つ目の問題を解決するために KVM 側にあてたソースコードの修正（パッチ）について詳解する．**図 5** にエミュレーションレイヤと KVM に適用するパッチがどのように協調動作するかを図示する．

パッチは，KVM が内部に持つマッピング管理テーブルをエミュレーションレイヤ側の持つマッピング管理テーブ

ルに置き換えてしまう作用を持つ．KVM は内部に独自のマッピング管理用のデータ構造を持っている．図中では，QEMU の 0 番仮想ページを a 番の物理ページに対応付けるというテーブルが KVM 内に存在している．fault/nopage ハンドラが呼ばれたときには KVM は内部的にこのテーブルを参照してどの QEMU の仮想ページにどの物理ページを対応付けるかを決定している．一方で，エミュレーションレイヤのほうでもマッピングの管理テーブルを持っている．図中では，QEMU の 0 番仮想アドレスに a 番の物理ページが対応付けられている．

エミュレーションレイヤとパッチの実際の動作を順を追って解説する．これから記す一連の流れは，ユーザによってゲスト OS が起動されたときの初期化時に 1 度だけ行われるものである．

初めに，エミュレーションレイヤはカーネル空間とユーザ空間との間にメモリマッピング領域（図中の (A)(B)(A)'(B)'）を作成する．領域のサイズはゲスト OS に割り当てられる全メモリの量で決まる．たとえば，ユーザがゲスト OS に 300 MB のメモリ領域を割り当てた場合，300 MB のメモリマッピング領域が作られる．MDL [*1] と呼ばれる Windows の機構を使うことでメモリマッピングを行う．マッピングが行われると，エミュレーションレイヤ内のマッピングの管理テーブルに，QEMU の仮想アドレスのどのオフセットアドレスから何ページ分割り当てられているか，それらがどのカーネル内物理アドレスに割り当てられているかが記録される．図中では，2 つのマッピング (A)(B) が作られており，(A) はゲスト OS のメモリの 0x0 番地から 8 ページ分のメモリが割り当てられている．(B) には 0xfee00000 番地から 3 ページ分のメモリが割り当てられている．(A) と (B) に存在するページはカーネル内の物理アドレスに対応付けられている．そして，(A) 領域の 0 番仮想ページはカーネル内の a 番物理ページに割り当てられている．(B) 領域の 0 番仮想ページはカーネル内の i 番物理ページに割り当てられている．これらの割当ては上述したとおり Windows カーネルが暗黙的に決定する．これらの割当てはすべてエミュレーションレイヤのマッピング管理テーブルがすべて把握している．

次に KVM に修正を加える．これは KVM が内部でゲスト OS 用の物理メモリを割り当てるときに，QEMU のどの仮想アドレスを割り当てようとしているのかをエミュレーションレイヤに対して通知できるように改良を加えるものである．具体的には，KVM 内でゲスト OS 用のメモリ領域を割り当てる関数である `kvm_main.c` 内の `kvm_vm_ioctl_set_memory_region()` にパッチを適用した．図を使って説明する．KVM は QEMU の仮想アドレスに対して物理ページを割り当てるべく `alloc_pages()` と呼ばれる関数を複数回呼び出す．図では (A) 領域と (B) 領域であわせて 11 ページ分のメモリがゲスト OS の領域として割り当てられているため，11 回 `alloc_pages()` 関数が呼ばれて KVM は 11 個の物理ページを得る．KVM が `alloc_pages()` を 1 回呼び出すごとに，どの仮想ページがどの物理ページに対応付けられるかが決定され，これは KVM 内部のマッピング管理テーブルに登録される．パッチはこの `alloc_pages()` 関数を `wk_alloc_pages(pnum, base)` に置き換える．この関数は 1 つの空ページを返す点では `alloc_page()` と同じであるが `pnum`，`basefn` の 2 つの引数をとる点で異なっている．これは，KVM が `wk_alloc_page()` を呼び出したとき，どの QEMU の仮想ページ（$basefn + pnum$）にどの物理ページを割り当てようとしているのかをエミュレーションレイヤに伝えるためである．エミュレーションレイヤはこの 2 つの引数を受け取り，すでに存在するマッピングと整合がとれるように適切な物理ページを返す．図を使って説明すると，`wk_page_alloc(0, 0x00000000)` が呼ばれたときには，エミュレーションレイヤは a 番の物理ページを返す．なぜなら，すでに，エミュレーションレイヤがメモリマッピングを作っているため，0 番仮想ページに割り当てる物理ページは a 番でなければならないからである．つまり，`wk_alloc_page()` 関数は，KVM がゲスト OS 用のメモリを割り当てるために使用する特別仕様の `alloc_page()` 関数であるといえる．

結果的に，エミュレーションレイヤのマッピング管理テーブルと KVM 内部のマッピングテーブルが完全に同一のものとなる．これで，KVM の fault/nopage ハンドラを使用しなくても Windows 上で QEMU と WinKVM ドライバ間でメモリマッピングを行うことができる．なお，KVM 内の fault/nopage ハンドラはエミュレーションレイヤが呼び出すことはない．つまり，WinKVM 上で fault/nopage ハンドラが使用されることはない．

しかしながら，上述したように，改造を施した KVM でもゲスト OS に 300 MB のメモリしか割り当てられないという問題は解決しない．この問題を解決するためには，QEMU のメモリ割当ての手法を根本的に変更する必要がある．これについては，4 章で解説する．

4. 移植で得た知見

我々は，WinKVM を開発する過程で，Windows と Linux の双方で動作する移植性のある Hybrid Hypervisor を設計するうえで守らなければならない重要な知見を得ることができた．

最も重要な点は，ゲスト用のメモリをホスト OS のユーザ空間上で大きな 1 枚の連続したメモリ領域（ポインタ）として管理するべきではないという点にある．QEMU で

[*1] セクションを使ってマッピングする方法もあるが，今回は使用できない．なぜなら，カーネル内で物理ページのアドレスを得ることができなくなってしまうからである．

はゲスト用のメモリをホスト OS から見たユーザ空間上に 1 つの連続したメモリ領域（ポインタ）として確保しており，それを前提としてコードが書かれている．すなわち，一部の QEMU デバイスがゲスト OS のメモリ領域を読み書きするとき，通常のポインタ操作でこれを実現している．このメモリ確保の手法では Windows と Linux 間で互換性のある HybridHypervisor を実装することができない．つまり，ゲスト OS 用のメモリとして 512 MB の領域を割り当てること考えたとき，ホスト OS のユーザ側からこれを `mmap()` や `VirtualAlloc()` 等の関数で 512 MB の 1 つの連続したメモリ領域として確保するべきではない．Linux だけで動作する Hybrid 型 Hypervisor を考えるのであれば，この設計で特に問題はない．しかし，Windows と Linux で動作する移植性のある Hybrid 型 Hypervisor を考えるとこのアーキテクチャは大きな問題を引き起こす．Windows では 1 度にマッピングできるメモリの領域が限られているため，ゲスト OS のメモリの割当て制限を引き起こすのである．

移植性の高いメモリ管理手法を**図 6** に示す．たとえば，512 MB のメモリをゲスト用として割り当てる状況を考える．このとき，512 MB の領域を複数のメモリチャンクに分割して，たとえば，4 つに分割した 128 MB のメモリ領域としてホスト OS のユーザ側にそれぞれ独立してマッピングできるように設計する必要がある．つまり，4 つの独立した非連続のメモリチャンクを用意し，それぞれ独立してユーザ空間にマッピングする必要がある．Windows では 1 つのメモリチャンクがマッピングできるメモリの容量は限られている（2 GByte の物理マシンを搭載したマシンで 1 度に 300 MB のメモリチャンクしかマッピングできない）．しかし，4 つメモリチャンクを独立してそれぞれマッピングすることで，512 MB のメモリ領域をユーザ空間に割り当てることができるようになる．なお，WinKVM にこの手法を適用するためには，上述したとおり，ゲスト OS 用のメモリを分割して確保できるように QEMU と KVM のコードを大幅に書き直す必要がある．上述したとおり，ユーザ空間にある QEMU ではゲスト OS を 1 つの大きなメモリ領域として確保しており，ポインタを通した通常のメモリ操作でゲスト OS へのアクセスを行っている．そのため，本論文で述べたエミュレーションレイヤではメモリ割当て制限を回避することはできない．

上記の点が守られない場合, WinKVM のように, 2 GBytes の物理メモリを搭載していても，ゲスト OS に 300 MB のメモリしか割り当てられないという事象が発生する．この知見は KVM の移植のためだけでなく，VT-x や AMD-SVM を使用するすべての Hybrid 型 Hypervisor に関係する話である．なぜなら，これらの拡張機能を使う Hypervisor を設計するにあたって，ユーザ空間のコンポーネントとカーネル空間のコンポーネント間でゲストのメモリ空間を共有

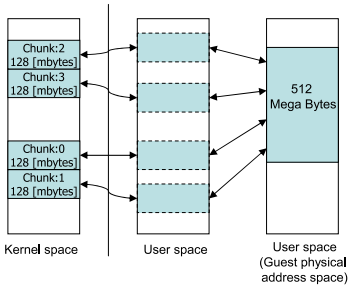


図 6　移植性のある Hypervisor のためのメモリマッピング手法
Fig. 6　The best method to memory mapping for the compatible hypervisor.

（マッピング）するアーキテクチャを採用することははほぼ必須だからである．

Linux と Windows で動作する Hybrid 型の Hypervisor である VirtualBox はゲスト用のメモリを単一の連続したメモリ領域としてユーザ側にマッピングしていない．VirtualBox では，ホスト OS のユーザ側で動作するコードがゲスト OS のメモリを読み書きする際には，すべて `PDMDevHlpPhysRead()`, `PDMDevHlpPhysWrite()` 関数を経由して読み書きしている．これらの関数は，確保されたゲスト OS のメモリチャンクが独立してマッピングされていても，正確にゲスト OS のメモリを読み書きできるようにするためのものである．VirtualBox はゲスト OS のメモリを確保するとき，複数の非連続なメモリチャンクを別々に確保している．たとえば，900 MB のゲストメモリ確保するときには，300 MB ごとのメモリチャンクを 3 個別々にマッピングすることで，結果的にゲスト OS には 900 MB のメモリが割り当てられるような設計になっている．そのため，WinKVM で発生しているような，2 GByte の物理メモリを確保しているにもかかわらず，ゲスト用のメモリ 300 MB のメモリしか割り当てられないといった不適切な制限は発生しない．

5. WinKVM と KVM のライブマイグレーション

ここまで，我々は WinKVM の開発手法について述べてきた．WinKVM の完成度は高く，KVM と WinKVM 間のライブマイグレーション機能を動作させることも可能である．ライブマイグレーションの処理は，ほぼユーザ側で動作する QEMU 側から行われるため，カーネル側のエミュレーションレイヤに特に工夫をする必要はない．しかし，後述するライブマイグレーションの評価を理解するためには，KVM のライブマイグレーションのプロトコルについて詳細に理解する必要がある．そのため，ライブマイグレーションプロトコルについて詳解する．

5.1 KVM のライブマイグレーションプロトコル

KVM のライブマイグレーションのシーケンス図を**図 7**

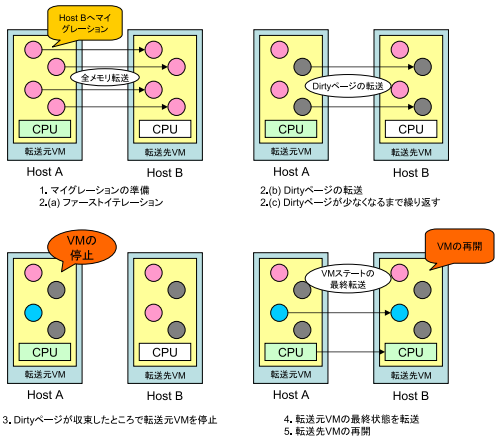


図 7　KVM の VM ライブマイグレーションメカニズムの全容図
Fig. 7　The overview of the VM-migration machanism of KVM.

に示す．基本的な流れは VMware ESX Server の VMotion [9] や Xen のマイグレーション [10] と同様に Pre-copy と呼ばれる手法を用いている．

VM ライブマイグレーションを考えるうえで重要な点は VM のダウンタイムを可能な限り縮小することである．VM を構成するペイロードの中で最も支配的な要素は VM のメモリデータである．これは巨大なデータであり，数ギガバイトになることもめずらしくはない．VM のダウンタイムを小さくするためには，このメモリ領域の転送手法を工夫する必要がある．

そのため，KVM のライブマイグレーション機構は VM を動作させつつメモリの転送（Pre-copy）を行うことで，VM のダウンタイムを削減している．図 7 を用いて，ホスト B とホスト A 間でライブマイグレーションを行う例を考える．ライブマイグレーションのプロトコルは以下の 5 つのフェーズに分割して考えることができる．

(1) 転送元ホストから転送先ホストへマイグレーション要求の送信．転送先ホストの資源の予約と初期化．
(2) メモリの転送
　(a) はじめに転送元 VM のメモリすべてを転送する．（ファーストイテレーション）．
　(b) さらにイテレーションを行う．このときは，前回のイテレーション時から更新されたページ（Dirty ページ）のみを転送する．
　(c) Dirty ページが少なくなるまで（収束するまで）イテレーションを繰り返す．
(3) 転送元 VM の停止．ある程度 Dirty ページが少なくなったところで転送元 VM を完全に停止する．
(4) 転送元 VM ステートの最終転送．CPU の状態や I/O デバイスの内部レジスタの状態等を転送する．転送元 VM が完全停止の直前に行ったイテレーションで見つかった Dirty ページもこのとき転送される．
(5) VM の再開
　(a) 転送先 VM へのライブマイグレーションが成功すれば．転送元 VM を停止する．ネットワークがあれば，ARP ブロードキャストを使用して IP アドレスと MAC アドレスの対応付けを更新する．
　(b) 転送先 VM へのライブマイグレーションが失敗すれば．転送元 VM を再開する．

(2) (c) の収束条件を変えることによりライブマイグレーションの性能を変更することができる．Dirty ページの残りを少なく設定すれば (3) の実行時間が短くなるため，VM のダウンタイムが短くなる．しかし，少なく設定することにより (2) から (3) への遷移がいつまでたっても起こらない可能性がある．逆に大きく設定すれば (2) から (3) への遷移はすばやく完了するが，今度は (4) のフェーズに時間がかかってしまい，結果 VM のダウンタイムが長くなってしまう．

KVM は Dirty ページの数が 50 前後になった時点で (3) から (4) への遷移を行っているようである．この値は変更可能であり，ユーザがライブマイグレーションに求める性能に応じて調整可能である．以上が KVM のライブマイグレーションプロトコルの概要である．

5.2　WinKVM でのライブマイグレーション実装

ライブマイグレーションの処理はほぼユーザ側で動作する QEMU 側から行われるため，カーネル側のエミュレーションレイヤに工夫をする必要はない．WinKVM でライブマイグレーションを達成するためには，エミュレーションレイヤと，図 1 に示した QEMU 側から WinKVM を制御するための `libkvm` を正確に移植する必要がある．たとえば，KVM ドライバ側から VM の MSR を取得するための `kvm_get_msrs()` 関数の移植も正確に行う必要がある．この関数は，WinKVM 上で CentOS 5.4 を起動させるためには不要であるが，ライブマイグレーションを実現するためには必要不可欠である．

また，KVM-17 と WinKVM が使用する QEMU のバージョンを正確にそろえる必要がある．KVM-17 はデフォルトで QEMU 0.9.0 を使用しているが，一方で，WinKVM は Windows の移植版が存在する QEMU 0.9.1 を使用していた．なぜなら，Windows に移植されている QEMU に 0.9.0 は存在しなかったためである．QEMU のバージョンが違うと，送信受信間での QEMU デバイスのレジスタセットの整合がとれなくなる．これは，両者の QEMU 間でレジスタセットをそのまま送信するライブマイグレーションにとっては致命的である．ライブマイグレーションを行うためには KVM が使用する QEMU を 0.9.0 から 0.9.1 に変更する必要がある．KVM-17 が使用する QEMU 0.9.0 には，KVM と連携を行うための修正が入っているが，QEMU 0.9.1 にはこのような修正が入っていない．そこで，QEMU 0.9.1 でも KVM-17 との連携を行うための修正を入れた．

6. 評価

評価は大きく分けて 2 つの項目について行った．WinKVM の自体の性能評価と，VM ライブマイグレーションの性能評価の 2 つである．初めに，WinKVM 単体での比較を行い，我々のエミュレーションレイヤが KVM に対して与えるオーバヘッドを調査する．次に，エミュレーションレイヤが VM ライブマイグレーションが与える影響について調査する．これから述べてゆくベンチマーク結果を使って，エミュレーションレイヤによる移植手法は，性能劣化を引き起こすことなくまったく異なる OS 間での移植を実現することが可能であることを示す．また，KVM と KVM 間のライブマイグレーションに比べて，エミュレーションレイヤが WinKVM と KVM のライブマイグレーションに致命的な影響を及ぼさないこと示す．

6.1 WinKVM の性能評価

初めに，WinKVM 自体の性能測定を行った．評価環境は以下のとおり，DELL LATITUDE D630 ラップトップコンピュータを SingleCore モードでブートして使用した．CPU は Intel Core2Duo T7300 2.0 GHz で，物理メモリは 2 GB である．HDD は WDC 製の 7200 rpm HDD で 8 MB のキャッシュメモリが搭載されている．ホスト OS は CentOS 5.4 で，カーネルのバージョンは 2.6.18-164.6.1.el5 (32 bit，100 Hz の割込みレート) と，Windows XP (Service Pack 3) を使用した．ゲスト OS はすべて CentOS 5.4 で，カーネルのバージョンは 2.6.18-164.6.1.el5 である．ゲスト OS に割り当てられたメモリは 300 MB である．

6.1.1 Linux/UNIX nbench による評価

WinKVM の性能測定では，初めに，computation-intensive なベンチマークとして Linux/UNIX nbench プログラム [11] を使用する．このプログラムは Mayer 氏によって書かれたベンチマークプログラムであり，NUMERICSORT, STRINGSORT, NeuralNetwork シミュレーション，LU DECOMPOSITION などといったいくつかのプログラムを含んでいる．nbench によるすべての評価の値は 30 回行ったうちの平均値である．

測定にあたって，Linux/UNIX nbench プログラムに 2 つの改良を加えた．これには 2 つの理由がある．

1 つ目に，nbench は対象コンピュータの性能を事前に測定し，最適なベンチマークのループ回数を決定する．また，結果として出るスコアは nbench オリジナルのスコアである．我々は，この事前測定はベンチマーク結果に不明瞭な要素を与え，さらに，nbench のオリジナルスコアは分かりにくいと考えた．そのため，各ベンチマークプログラムが最低でも 20 秒以上の時間を消費するようにループ回数を固定し，スコアはベンチマークプログラムが費やした秒数を表示するように nbench を修正した．

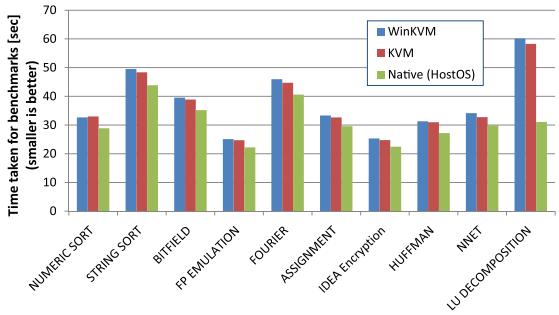


図 8　Linux/UNIX nbenchmark プログラムの測定結果
Fig. 8　Linux/UNIX nbenchmark results, compiling with gcc 4.1.2 CFLAGS = -O3 -Wall -s.

2 つ目に，一般的に Hypervisor 上での時計は不正確であるため，時計に依存しているベンチマーク結果も同様に不正確である．改造していない nbench を WinKVM 上と KVM 上で走らせてベンチマーク結果を得たところ，WinKVM がオリジナルの KVM よりも 10 倍以上高速であるという結果が出た．この結果はまったく信頼できない値である．正しい測定結果を得るため，我々は，物理的なマシンを 1 台用意しそのマシン上で時間測定を行った．より具体的にいうと，nbench プログラムが使用している `gettimeofday()` 関数を，ネットワーク経由の外部マシン上で実行できる仕組みを構築し，それを使用してベンチマーク結果をとった．このとき，ネットワーク遅延による時間測定のずれを考慮する必要がある．この遅延をなるべく減らすため，以下に示す 2 つの条件下での実験を行った．1 つ目は，Nagle アルゴリズムを OFF にした状態で測定を行った．2 つ目は，実験で使用する LAN 環境上でベンチマーク測定のためのパケット以外はいっさい流れていない状態で測定を行った．むろん，以上の 2 つを考慮してもネットワーク経由の時間測定である以上，ある程度の遅延は避けられない．しかし，今回行った nbench によるすべての測定結果はまったく同一 LAN 環境下での実験結果である．ping コマンドによる測定では，ネットワーク遅延の大きな乱れは見られなかったため[*2]，3 種類の値（ホスト OS 上，KVM 上，WinKVM 上）に対してかかるネットワークの遅延は安定しているため，相対的にみると公平な値であり，比較して論じることが可能である．

図 8 に nbench の測定結果を示す．なお，参考までにホスト OS（CentOS 5.4）で実行したときのベンチマーク結果もグラフに加えた．全体的にみて，ベンチマーク結果からエミュレーションレイヤが WinKVM に致命的な影響を与えないことが分かる．最も性能低下を引き起こしている NNET ベンチマークでも，約 3.9%の性能低下にとどまっている．また，BITFIELD や FP EMULATION，HUFFMAN ベンチマークの性能低下は 2%以内である．

[*2] 測定機器へと 30 回 ping を送ったときの平均値は 0.283 ms，標準偏差は 0.0013 である．

```
int main(int argc, char *argv[])
{
  int i;
  int status;
  for (i = 0 ; i < 40000; i++) {
    int pid = fork();
    if (pid < 0) return -1;
    if (pid == 0) return 0;
    waitpid(pid, &status, 0);
  }
  return 0;
}
```

図 9　FORKWAIT ベンチマーク
Fig. 9　FORKWAIT benchmark.

さらに，NUMERICSORT ベンチマークでは WinKVM と KVM の間にほとんど性能差がみられないという結果が出ている．つまり，nbencn のような computation-intensive なベンチマークでは，エミュレーションレイヤのオーバヘッドは無視できるレベルであることが分かった．

6.1.2　forkwait ベンチマークによる評価

次に，virtualization-sensitive なベンチマークとして，**図 9** に示す FORKWAIT マイクロベンチマークを使用する．これはプロセスの生成と破棄を 40,000 回繰り返すプログラムである．

評価の結果，ホスト OS での実行時間は 3.296 秒，KVM では 53.436 秒，WinKVM では 54.304 秒であった．WinKVM は 1.6%の性能低下しか引き起こしていないことが分かる．つまり，FORKWAIT のような virtualization-sensitive なベンチマークでもエミュレーションのオーバヘッドは無視できるレベルであることが分かった．

6.1.3　ApacheBench による評価

次に，より現実的なプログラムを用いて WinKVM の性能を評価するため，Apache server benchmark を使用した．サーバに Apache/2.3.3 を使用し，10,000 回の HTTP リクエストを同時接続数 1 として WinKVM と KVM-17 に対して処理させた．LAN 上にある計算機で AB（Apache server benchmark）を実行して測定した．**図 10** に評価結果を示す．10,000 回のリクエストを処理する時間が，WinKVM が 102.33871 秒であるのに対して，KVM-17（100 Hz）は 39.831594 秒であることが判明した．つまり，WinKVM のほうが約 62.507 秒低速であるという結果が出た．

既存研究で，OS の割込みレートの違いによってベンチマークの測定値が変わるという現象が報告されている [12]．我々は，ApacheBench において，WinKVM が低速である理由が Linux の割込みレートと Windows の割込みレートの違いにあると考えた．

この仮説を検証するために 2 つの実験を行った．1 つ目の実験は，異なる割込みレートを持つ Linux カーネルを 3 つ

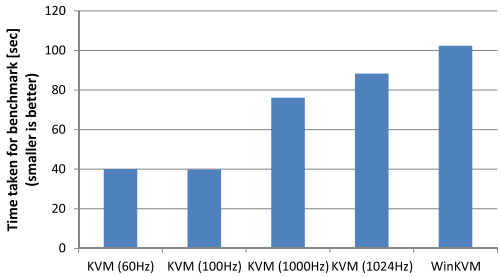


図 10　ApachBench で 10,000 リクエストにかかる時間の平均値
Fig. 10　Consumued time for 10,000 requests by ApachBench.

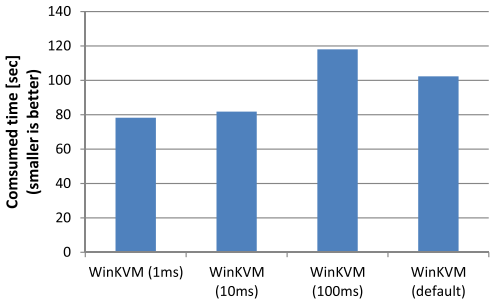


図 11　ApachBench で，WinKVM 上で異なる割込みレートにおいて，10,000 リクエストにかかる時間の平均値
Fig. 11　Consumued time for 10,000 requests by ApachBench with different interrupt rates on WinKVM.

（60 Hz，1,000 Hz，1,024 Hz）用意して ApacheBench による計測を行った．これらの割込みレートは代表的な OS で使用されている値になっている．2 つ目の実験は，Windows カーネルの割込みレートを `timeBeginPeriod()` API によって変更して，同様に ApacheBench による計測を行った．

1 つ目の実験結果を図 10 に示す．Linux の割込みレートによって Apache Bench の結果が大きく変わることが判明した．全体的に，割込みレートが増加すると，速度が低下する．KVM（1,000 Hz，1,024 Hz）と KVM（60 Hz，100 Hz）では約 2 倍の性能差がみられる．Linux の割込みレートが 1,024 Hz の場合，差はあるものの，WinKVM の速度とほぼ同等になる．

2 つ目の実験結果を**図 11** に示す．Windows 側の割込みレートを変更したうえで ApacheBench による計測を行った．全体的に，タイマの分解精度を 100 ms から 1 ms へ変更してゆくと（割込みレートを高くすると）ApacheBench の結果は KVM（割込みレート 1,000 Hz）の値に近づくことが分かった．

上記の 2 つの実験結果より，WinKVM が低速である理由が Linux の割込みと Windows の割込みレートの違いにあることが証明された．

6.2　ライブマイグレーションの性能評価

我々は次に，エミュレーションレイヤが，ライブマイグレーションに対して致命的なパフォーマンス低下を与えていないかを調査するための評価を行った．

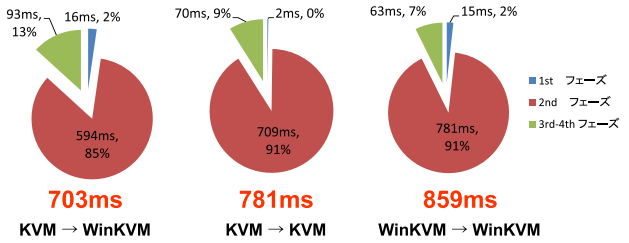


図 12　ライブマイグレーションの測定結果
Fig. 12　The measurement of live migration.

5 章で述べたように，ライブマイグレーションのプロトコルは 5 つのフェーズに分けて考えることができる．評価は 2 つ行った．初めに，KVM-17 から KVM-17 に向けて VM ライブマイグレーションを行い，4 つのフェーズがそれぞれどの程度の時間を消費しているかを測定した．次に，KVM-17 側から WinKVM 側に VM ライブマイグレーションを行い，同様に 4 つのフェーズがどの程度時間を消費しているか測定を行った．最後の 1 フェーズは VM を再開するだけの処理なのでベンチマーク結果には含めない．なお，すべてのベンチマークの値は 3 回行ったうちの最良値である．

測定環境には 2 台のコンピュータをネットワークで接続して行った．1 台目 VM 送信元のコンピュータは IBM ThinkPad X60 を SingleCore モードでブートして使った，CPU は Inte Core DUO T2400 @ 1.83 GHz．物理メモリ 2 GB である．ホスト OS は CentOS 5.4 で，カーネルのバージョンは 2.6.18-164.6.1.el5（32 bit）である．2 台目のコンピュータは 6.1 節の評価で用いた DELLLATITUDE D630 ラップトップコンピュータである．ホスト OS はホスト OS は CentOS 5.4 で，カーネルのバージョンは 2.6.18-164.6.1.el5（32 bit）と WindowsXP（Service Pack 3）である．そして，これら 2 つのコンピュータは GIGABIT ハブで接続されている．ライブマイグレーションに使用した VM には 300 MB のメモリが割り当てられており，ゲスト OS は RedHat 9（カーネルバージョン 2.6.20）をベースにした QEMU のテスト用 Linux である．これは QEMU の Web ページからダウンロードすることができる．なお，それぞれ 2 つの VM には `/dev/zero` を 100 MB 分 `mmap()` した後，確保した 100 MB の領域を 4 KB ページごとに線形に書き込んで行くという負荷をかけたうえでライブマイグレーションを行った．

図 12 の中央に KVM-17 から KVM-17 へ VM ライブマイグレーションの結果を示す．VM ライブマイグレーションに費やした時間は 781 ミリ秒である．すべての処理の中で最も支配的な要素は 2nd フェーズの VM メモリ転送であり，全体時間の 91%を占めていることが分かる．次に支配的な要素が 3rd-4th フェーズである．これは VM の最終ステート転送であり，9%を占めていることが分かる．最後に，1st フェーズが 1%未満でありほとんど時間を使用していないことが分かる．

次に，図 12 の左に KVM-17 から WinKVM への転送時間を示す．VM ライブマイグレーションに費やした時間は 703 ミリ秒である．すべての処理の中で最も支配的な要素は，同様に 2nd フェーズの VM メモリ転送であり 85%を占めている．次に支配的な要素も同様に，3rd-4th フェーズであり 13%を占める．最後に，1st フェーズが 2%の時間を占める．KVM-17 から KVM-17 のマイグレーションを基準にして，マイグレーションの全体時間では 11%の性能変化，2nd フェーズでは 19%の性能変化，3rd-4th フェーズでは 24%，1st フェーズでは 87%の性能変化がみられる．

最後に，図 12 の右に WinKVM から WinKVM へのライブマイグレーションの結果を示す．VM ライブマイグレーションに費やした時間は 859 ミリ秒である．すべての処理の中で最も支配的な要素は，同様に 2nd フェーズであり全体時間の 91%を占めている．次に支配的な要素も同様に 3rd-4th フェーズでありこれは全体の 7%を占めている．最後に，1st フェーズが 2%の時間を占める．KVM-17 から KVM-17 へのマイグレーションを基準にして，マイグレーションの全体時間では 9%の性能変化，2nd フェーズでは 9%の性能変化，3rd-4th フェーズでは 11%の性能変化，1st フェーズでは 86%の性能変化がみられる．

3 つのライブマイグレーションの結果を比較する．エミュレーションレイヤによる性能変化は 1st フェーズで最大 87%，2nd フェーズで最大 19%，3rd-4th フェーズでは 24%である．しかし，マイグレーション全体の性能変化は最大でも 11%の性能変化にとどまっている．また，実測値を比較すると，それぞれの差は数十秒ミリ秒とほとんど変わらず，エミュレーションレイヤはライブマイグレーションに対して致命的な性能低下を与えるものではないということが分かった．

6.3　評価のまとめ

本章で行った評価をまとめる．Linux/UNIX nbench のような computation-intensive なベンチマークでは，エミュレーションレイヤが WinKVM に対して致命的なパフォーマンスの低下を引き起こしていないことが判明した．最も性能低下を引き起こしている NNET ベンチマークでも，約 3.9%の性能低下にとどまっている．次に，virtualization-sensitive なベンチマークとして，FORKWAIT ベンチマークを走らせた結果，WinKVM は 1.6%の性能低下しか引き起こさない．また，実用的なベンチマークとして ApacheBench を使用した測定を行ったところ，WinKVM のほうが約 60%低速であるという結果を得た．この速度差はホスト OS の割込みレートに起因するものであるということが分かった．全体的にみて，エミュレーションのオーバヘッドは無視できるレベルであることが判明した．

また，KVM-KVM，KVM-WinKVM，WinKVM-

WinKVM のライブマイグレーションの評価結果では，それぞれ数十ミリ秒程度の変化しかみられず，同様に，エミュレーションレイヤがライブマイグレーションに与えるオーバヘッドは無視できるレベルのものである．

7. 関連研究

1 つ目の研究として，VirtualBox があげられる．VirtualBox は Windows や Linux をはじめとするさまざまな OS 上で動作する高い移植性を持った Hybrid 型 VMM である．VirtualBox は初めから高い移植性を持つように設計されており，ゲスト OS のメモリ管理については入念な設計がなされたうえで実装が行われている．これについては 4 章で述べたとおりである．これに対して本研究は，Linux に強く依存している KVM をほとんど修正することなく，エミュレーションレイヤを用いて Windows 上で動作させることを目的としている．本研究の新規性は Windows カーネル上でパフォーマンスの低下をほとんど引き起こすことなく Linux カーネルをエミュレーションし Hypervisor の移植手法を提示したことにある．そのため，VirtualBox とは前提条件の違いによりメモリ割当て等の移植手法に関するアプローチが異なっている．

2 つ目の研究としては Porting Linux KVM to FreeBSD [13] があげられる．これは linux-kmod-compat とスタブドライバを利用して，KVM-17 を FreeBSD 上に移植したものである．linux-kmod-compat とは，Linux のデバイスドライバを FreeBSD 上でもコンパイルを可能にする互換レイヤである．このプロジェクトは Linux の互換レイヤを作るという点では本提案に似ている．しかし，KVM の移植を考えるうえでは FreeBSD と Linux のメモリ管理機構にそれほどの違いはないため，本論文のようにメモリ共有に関して考慮する必要はない．そのため，KVM 移植という観点では同じであるが，本質的にあまり違いのない Linux から FreeBSD へのドライバ移植の例であるため，3 章で述べたような本質的に異なるメモリ管理機構をまたいだ移植の知見は存在しない．

3 つ目の研究としては，Linux と FreeBSD 上で，Windows の無線 LAN ドライバ（NDIS ドライバ）を動作させるプロジェクトがあげられる．これらの研究は，それぞれ，Ndiswrapper [14]，NDISulator（Project Evil）[15] と呼ばれている．これらのプロジェクトは，Windows の NDIS ドライバをそのまま動作させることができる点で興味深い．しかし，本研究が Linux から Windows 上へのドライバ移植（KVM）を目指しているのに対してこれらのプロジェクトはその逆を行う．

4 つ目の研究としては，Linux ドライバを Solaris 上で動作させる PITS Library という研究がある [16]．この研究は，Linux のデバイスドライバのソースコードのコンパイル時に，PITS Library と呼ばれるスタブドライバをリンクさせ，Linux の互換レイヤを移植対象の Linux のデバイスドライバに提供している．しかし，この研究にも 3 章で述べたようなメモリマッピングの違いを吸収するための機構に関する説明は見当たらない．

8. おわりに

我々は，Linux/KVM の Windows 移植版である WinKVM（Windows Kernel-based Virtual Machine）を開発した．開発した WinKVM は Windows と Linux 間で VM ライブマイグレーションが可能であるほど完成度が高い．WinKVM を構築するために，Windows 上で Linux カーネルを模倣するエミュレーションレイヤを構築し KVM をその上で動作させた．

いくつかのベンチマークプログラムで WinKVM の性能を測定したところ，オリジナルの KVM との性能差はみられず，エミュレーションレイヤによる性能劣化はないことが分かった．また，KVM-KVM，WinKVM-KVM，WinKVM-WinKVM 間のライブマイグレーションをそれぞれ比較したところ，性能劣化は数十ミリ秒程度に収まっている．つまり，本手法により，致命的なパフォーマンスの低下を引き起こすことなく，まったく異なる OS 間での Hypervisor 移植の実現が可能であることが示された．

また，Linux エミュレーションレイヤを作るうえで直面した，開発環境の違いや，メモリアーキテクチャの根本的な相違等，いくつかの困難か課題をどのように解決したかについて論じた．さらに，その問題を解決する過程で我々は Windows と Linux 間で互換性のある Hybrid 型 Hypervisor を設計するうえで留意すべき重要な知見を得ることができた．最も重要なことは，ゲスト用のメモリをユーザ側で大きな 1 枚の連続したメモリ領域として扱ってはいけないという点である．つまり，ゲスト OS 用のメモリとして 512 MB の領域を割り当てること考えたとき，ユーザ側からこれを `mmap()` や `VirtualAlloc()` 等の関数で 512 MB の 1 つの連続したメモリ領域として確保するべきではない．Linux だけで動作する Hybrid 型 Hypervisor を考えるのであれば，この設計で特に問題はない．しかし，Windows と Linux で動作する移植性のある Hybrid 型 Hypervisor を考える際，このメモリアーキテクチャを採用すると，不当な VM へのメモリ割当て制限を引き起こしてしまう．

今後の課題には，エミュレーションレイヤの完成度を向上し，最新版 KVM への対応を行うことが考えられる．現在のエミュレーションレイヤは KVM-17 が使用している Linux カーネル関数のみをエミュレーションしている．WinKVM がさらなる発展をとげるためには，最新の KVM がエミュレーションレイヤ上で動作できるようになることが望ましい．そのため，より包括的な Linux カーネルのエミュレーションが可能になるようにエミュレーションレイ

ヤに機能追加を行うことを考えている．さらに，最新版に対応したWinKVMを使用して本論文と同様の評価をとるべきであると考えている．

謝辞　本研究を遂行するにあたって多くのサポートをしていただいた，後輩である芝哲史氏に感謝します．

WinKVMの公開Webページ

`https://github.com/ddk50/winkvm`

参考文献

[1] Amazon.com, Inc.: Amazon EC2 (2009), available from ⟨http://aws.amazon.com/ec2/⟩.

[2] Luo, Y., Zhang, B., Wang, X., Wang, Z., Sun, Y. and Chen, H.: Live and incremental whole-system migration of virtual machines using block-bitmap., *CLUSTER'08*, pp.99–106 (2008).

[3] Bradford, R., Kotsovinos, E., Feldmann, A. and Schiöberg, H.: Live wide-area migration of virtual machines including local persistent state, *Proc. 3rd international conference on Virtual execution environments*, *VEE '07*, pp.169–179, ACM, New York, NY, USA (2007).

[4] Hirofuchi, T., Ogawa, H., Nakada, H., Itoh, S. and Sekiguchi, S.: A Live Storage Migration Mechanism over WAN for Relocatable Virtual Machine Services on Clouds, *IEEE International Symposium on Cluster Computing and the Grid*, Vol.0, pp.460–465 (2009).

[5] VMware, Inc.: VMware Storage VMotion: Non-disruptive, live migration of virtual machine storage, available from ⟨http://www.vmware.com/products/storage-vmotion/⟩.

[6] Bellard, F.: QEMU, a fast and portable dynamic translator, *Proc. USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, *ATEC'05*, p.41, USENIX Association, Berkeley, CA, USA (2005).

[7] Uhlig, R., Neiger, G., Rodgers, D., Santoni, A.L., Martins, F.C.M., Anderson, A.V., Bennett, S.M., Kagi, A., Leung, F.H. and Smith, L.: Intel Virtualization Technology, *Computer*, Vol.38, pp.48–56 (2005).

[8] AMD: Secure Virtual Machine Architecture Reference Manual, Technical report, Advanced Micro Devices (2005).

[9] Nelson, M., Lim, B.-H. and Hutchins, G.: Fast transparent migration for virtual machines, *Proc. USENIX Annual Technical Conference*, *ATEC '05*, p.25, USENIX Association, Berkeley, CA, USA (2005).

[10] Clark, C., Fraser, K., Hand, S., Hansen, J.G., Jul, E., Limpach, C., Pratt, I. and Warfield, A.: Live migration of virtual machines, *Proc. 2nd Conference on Symposium on Networked Systems Design & Implementation — Volume 2*, *NSDI'05*, pp.273–286, USENIX Association, Berkeley, CA, USA (2005).

[11] Mayer, U.F.: Linux/Unix nbench, available from ⟨http://www.tux.org/ mayer/linux/bmark.html⟩.

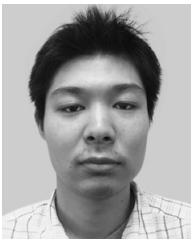
[12] Adams, K. and Agesen, O.: A comparison of software and hardware techniques for x86 virtualization, *Proc. 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, *ASPLOS-XII*, pp.2–13, ACM, New York, NY, USA (2006).

[13] Fabio, C.: Linux KVM on FreeBSD, available from ⟨http://retis.sssup.it/ fabio/freebsd/lkvm/⟩.

[14] Fuchs P, Pemmasani G.: Ndiswrapper (2005), available from ⟨http://ndiswrapper.sf.net/⟩.

[15] Paul B.: NDISulator, available from ⟨http://www.freebsd.org/doc/en/books/handbook/config-network-setup.html⟩ (2003).

[16] McIlwain, S. and Miller, B.P.: A tool for converting Linux device drivers into Solaris compatible binaries, *Softw. Pract. Exper.*, Vol.36, No.7, pp.689–710 (2006).



高橋 一志　（正会員）

1986年生まれ．2008年金沢工業大学工学部情報工学科卒業．東京大学大学院情報理工学系研究科創造情報学専攻博士前期課程修了．現在，同大学院博士後期課程2年所属．

笹田 耕一　（正会員）

2004年東京農工大学大学院工学研究科博士前期課程情報コミュニケーション工学専攻修了．2006年同大学院工学教育部博士後期課程電子情報工学専攻退学．博士（情報理工学）（東京大学情報理工学系研究科2007年）．2006年東京大学情報理工学系研究科助手，2008年同講師（現職）．システムソフトウェア，特に並列処理システム，言語処理系に関する研究に興味を持つ．