Vol.2010-OS-115 No.13
2010/8/4

KVMを利用した異種ホストOS間上でのライブマイグレーション

高橋一志[†1]　笹田耕一[†1]

われわれは，WinKVM (Windows Kernel-based Virtual Machine) と呼ばれる Linux KVM の Windows 移植版を構築することで，Windows と Linux 間で VM ライブマイグレーションを行うことを目指している．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る手法を開発した．しかし，既存研究で述べたエミュレーションレイヤは Windows 上で Linux カーネルのメモリ管理機構を正確に再現できなかった．そのため，QEMU と KVM ドライバとの間でゲスト OS のメモリ内容に不一致がおこり，VM ライブマイグレーションが実現が不可能であった．エミュレーションレイヤ上で KVM を動作させるため，KVM 内のメモリ管理情報をエミュレーションレイヤ側に通知できる機構を KVM に組み込むこみ，最終的に，われわれは WinKVM 上で Linux kernel 2.6.20 を起動させることに最終的に成功した．これで，VM ライブマイグレーションをおこなう土台が整ったことになる．

Inter-OS live-migration using our porting KVM

Kazushi Takahashi[†1] and Koichi Sasada

We aim at providing the VM-live migration between Windows and Linux by building *WinKVM*, a port of Linux KVM to Windows. To construct *WinKVM*, we developed a method to emulate Linux kernel functions on Windows kernel. However, our previous work could not emulate the memory management system of Linux on Windows correctly. There was an inconsistency of memory between the QEMU and the KVM driver. Therefore, we could not achieve VM-live migration. To achieve correct emulation of Linux memory management system, we have developed a channel sharing KVM internal memory management information with our emulation layer. We succeeded in executing the Linux kernel 2.6.20 on WinKVM eventually. Now, we could build the fundamental of VM-live migration between WinKVM and KVM.

1. はじめに

仮想化ソフトウェアである VMM (VMM:Virtual Machine Monitor) の成熟に伴い，計算機の利用者に対して計算機資源として物理的な計算機を割り当てる代わりに，VMM が管理する VM (VM:Virtual Machine) を割り当てるシステムが普及している．

このような基盤システム上では，利用者は遠隔地にある VM にネットワークを通じてアクセスする必要がある．VM はサーバ側に固定的であり，ユーザはシンクライアントや SSH を通してネットワーク越しに VM を操作する．そのため，ユーザの意思で VM の配置を動的に変更することにはあまり注意が払われていないのが現状である．

ユーザの意思による VM 再配置は数多くの利点がある．例えば，高い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 を使用したいといった要求に対して柔軟に対応することが可能となる．このようにコンシューマ向けの VM の再配置機構には大きなメリットがある．

そのため，コンシューマ向けの 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ことは今後の仮想計算機環境の発展に大きく寄与できる．

われわれは，Linux と Windows 間の VM ライブマイグレーションを目標として WinKVM[1)] を開発している．既存研究にてエミュレーションレイヤによる KVM[2)] の移植手法について述べた．これは，Windows カーネル上で Linux カーネルのエミュレーションレイヤを構築し，その上で KVM を動作させることで，Windows 版の KVM である WinKVM を得る手法である．なお，このエミュレーションレイヤの構築手法は KVM だけのものではなく，Windows から Linux への一般的なドライバ移植手法としても使用できる．しかし，以前で述べた手法にはまだ解決できていない問題点があった．

KVM は QEMU と KVM ドライバの 2 つのコンポーネントで構成されている．QEMU はユーザ空間で動作する Linux アプリケーションであり，KVM ドライバはカーネル空間で動作する Linux ドライバである．この 2 つのコンポーネントはゲスト OS 用のメモリ空間を互いに共有している．このため，QEMU 側から書き込まれたゲスト OS へのメモリ変更は KVM ドライバ側にも直ちに反映され，その逆も同様である．VM ライブマイグレー

†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

© 2010 Information Processing Society of Japan

ションを行うためには，KVM ドライバ側のダーティーページトラッキング機能と QEMU 側で動作するメモリ転送機構の両方が正しく動作する必要があるが，その前提としてこのメモリ共有が正確に行われている必要がある．

しかし，既存研究で述べたエミュレーションレイヤ[1] ではこの KVM ドライバと QEMU 側との間で正確なメモリ共有のエミュレーションを行うことが出来なかった．以前のエミュレーションレイヤでは Windows 上で Linux のメモリ管理機構を正確に再現することができず．そのため，QEMU と KVM ドライバ間でメモリデータの不一致が生じることがあり，QEMU 側から書き込んだデータが KVM ドライバ側に反映されなかったり，KVM ドライバ側からのメモリ書き込みが QEMU 側に反映されないといった問題点があった．そのため，このままでは，VM ライブマイグレーションを行うことが不可能である．このような問題が生じたのは，Windows と Linux のメモリ管理機構の性質が大きく異なり，この違いを吸収するエミュレーションレイヤの構築が困難であったためである．

本稿ではこの問題点を解決し，Windows 上で Linux ドライバを移植する手法について論じる．本稿の構成は以下のとおりである．はじめに，2 章にて KVM ライブマイグレーションの大まかな仕組みについて解説する．次に，3 章にて，エミュレーションレイヤ構築手法の概要について述べる．そして，4 章にて，エミュレーションレイヤを構築する上で問題となる点を挙げる．さらに，5 章にてこの問題点の解決手法について詳しく検討した上で，6 章にて最終的にわれわれがとった解決手法を述べる．さらに，7 章にて今回の解決法が正しく動作することを示し．8 章で関連研究を述べ，最後に 9 章にてまとめを述べる．

2. KVM ライブマイグレーション

われわれの目標は WinKVM と KVM 間とでライブマイグレーションを動作させることである．そのため本章では KVM のライブマイグレーションについて解説する．ライブマイグレーションの機構を述べるためには KVM のアーキテクチャについて理解する必要がある．そのため，はじめに 2.1 節にて KVM のアーキテクチャについて簡単に述べる．その後，2.2 節にて KVM ライブマイグレーションの動作原理について説明する．

2.1 KVM のアーキテクチャ

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 から構成されている．KVM ドライバは特権命令である VT-x[3] や AMD-SVM[4] を操作することでゲスト OS を動作する．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

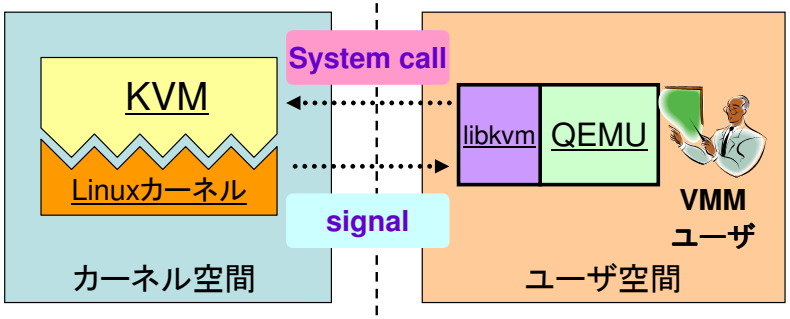

図 1 KVM のアーキテクチャ
Fig. 1 The overview of KVM architecture

から KVM ドライバ側にはシステムコールが発行され，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される．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からである．これは VT-x や AMD-SVM の仕様上必要なことである．一般的に，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．これは Linux も同様である，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．これで，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直ちに KVM ドライバ側にも反映される．なお，WinKVM を開発するにあたってはこのメモリ共有がもっとも障害となる点であり，それについては後の章で詳しく述べてゆく．以上が KVM アーキテクチャの概要である．

2.2 KVM ライブマイグレーションのプロトコル

KVM のライブマイグレーションのシーケンス図を**図 2** に示す．基本的な流れは VMware ESX Server の VMotion[5] や Xen のマイグレーション[6] と同様に Pre-copy と呼ばれる手法を用いている．なお，KVM は完全仮想化の OS をライブマイグレーションすることが出

ⓒ 2010 Information Processing Society of Japan

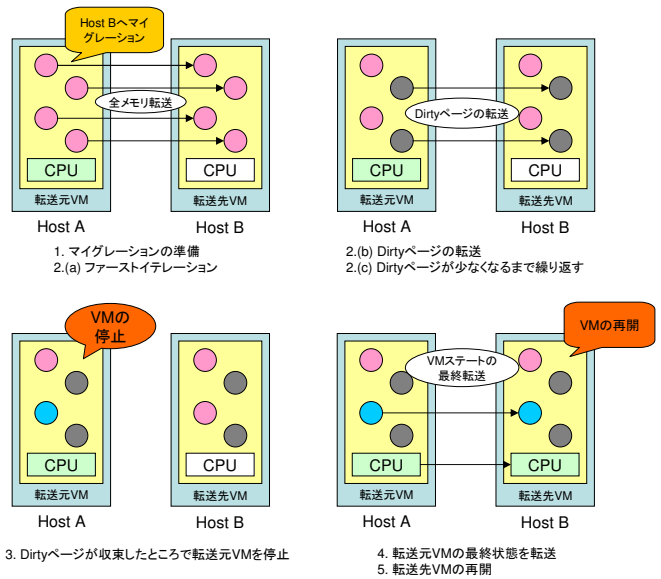

図 2 KVM の VM ライブマイグレーションメカニズムの全容図
Fig. 2 The overview of the VM-migration machanism of KVM

来る．これは VMotion も同様である．

VM ライブマイグレーションを考える上で重要な点は 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もっとも支配的な要素は 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 2 を参照しつつ，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出来る．

(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などを転送する．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 ページもこの時転送される
(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
  (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という可能性がある．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．

3. 移植手法の概要

KVM を移植するために，われわれは 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 を動作させることで，WinKVM として動作させるという手法を開発した.[1] この手法の大きな利点は KVM 自体に改良が加えられても，われわれの開発する Linux カーネルのエミュレーションが正確であれば，どのバージョンの KVM でも WinKVM として動作させることが出来るという利点がある．

© 2010 Information Processing Society of Japan

表 1 Linux のカーネル内函数と 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 のカーネル函数	変換に利用される Windows の API
カーネル内のメモリ割り当て	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)	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()
メモリバリアメモリへの変更を確実に RAM に書き込む	smp_wmb() smp_mb()	KeMemoryBarrier()

2.1 節で説明したように，KVM は KVM ドライバと QEMU から構成されている．はじめに，われわれは KVM ドライバにどのような Linux カーネル内函数を使用しているかを調査しリストアップした．

次に，このリストアップされた Linux カーネル内函数を Windows 上でエミュレーションする函数を記述していった．KVM 内で使用されている Linux カーネル内函数がどのような Windows カーネル API に翻訳されるのかを**表 1** に示す．なおこれはほんの一部であり，残りの函数は表 1 を組み合わせることで実装できる．

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ものを得る．まず，Cygwin 環境で KVM ドライバのソースコードをビルドする．すると，Linux カーネル函数を未解決シンボルとして含む KVM のコンパイル済みバイナリが得られる．KVM のコンパイル済みバイナリと Windows のエミュレーションレイヤをリンクし，KVM 内の未解決シンボルを Windows のエミュレーションレイヤの函数で代替する．こうすることで，正常に動作する Windows のドライバを得ることが出来る．

なお，ユーザ側で動作する QEMU に関しては移植する必要はない．なぜならすでに Windows 上で動作させるために改良を施した QEMU が存在するからである．これも Cygwin 環境でビルドできる．

以上が移植手法の概要である．この方法は KVM のソースコードを最大限に利用しつつ WinKVM を構築することを可能とする手法である．

4. エミュレーションレイヤの課題

3 章で述べたエミュレーションレイヤを構築するためには乗り越えなければならない技術的課題が 2 つある：

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．WinKVM を構築する際には，特に，両 OS がドライバに対して提供するメモリ管理機構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．

ドライバのバイナリフォーマットの違い KVM のコア部分は Linux ドライバとして実装されており，Linux ドライバのフォーマットは ELF と呼ばれるバイナリ形式である．一方，Windows は COFF-i386 形式と呼ばれるバイナリフォーマットである．そのため，

ⓒ 2010 Information Processing Society of Japan

最終的に WinKVM として Windows 上で動作するドライバを得るためには，KVM のソースコードを COFF バイナリとしてコンパイルする方法を開発する必要がある．

上記 2 つの問題点のうち，後者の問題点とそれに対する解決法については既存研究にて解決済み[1] であるため本稿では述べない．今回は以前の原稿で解決できなかった問題点である**ドライバのメモリアーキテクチャの違いについて詳しく述べる．**

Windows と Linux の大きな違いはそのメモリ管理機構の性質の違いにある．KVM を移植する上では特にこの違いが大きな問題となる．本節ではそれぞれのメモリ管理機構について詳しく述べた後，エミュレーションレイヤを実装する上での問題点を明らかにする．

多くの OS ではドライバとアプリケーション空間で同一メモリ空間を共有する機構を備えている．一般的に，この機構は，ドライバとアプリケーション間で大量のデータを送受信する必要あるドライバ (NIC や VGA デバイスのドライバ) で使用される．

このメモリ共有の機構を Windows や Linux 両 OS もそれぞれ備えている．Linux 上で実装されている KVM はこの機構を使用しており，ユーザ空間側で VM のデバイスをエミュレーションする QEMU の部分と，VT-x や AMD-SVM を利用してゲスト OS を実行する KVM ドライバとの両者間でゲスト OS のメモリ空間を共有するのに使用している．そのため，QEMU 側からのゲスト OS メモリ空間への変更が直ちにカーネル空間側の KVM ドライバにも伝わる．

しかし，Windows と Linux 両 OS のメモリ空間の共有機構の性質は大きく異なる．そのため，Windows カーネル上で Linux のメモリ管理機構を効率よくエミュレーションすることは困難である．

Linux のメモリ共有機構の概念図を**図 3** 示す．Linux には，`mmap()` システムコールによって予約した**連続するユーザアドレス空間を，カーネル空間で確保した不連続なカーネル領域のメモリを対応付けることが出来る．**具体的には，Linux ドライバの fault ハンドラ (nopage ハンドラ) 内にて，デバイスドライバのプログラマが指定することが出来る．例えば，ユーザアドレス空間側で図中の 0 番アドレスにデータのアクセスがあったとき，カーネル空間の 0 番アドレスメモリを対応付けるよう，fault ハンドラ内でプログラミングすることが可能である．KVM はアプリケーション (QEMU) と KVM ドライバ間でゲスト OS のメモリ空間を共有するためにこの機能を使用している．

次に，Windows でサポートされる，カーネル空間とユーザ空間のメモリ共有機構の概念図を**図 4** に示す．Linux が連続するユーザアドレス空間をカーネル空間で確保した不連続なカーネル領域のメモリを対応付けることが出来たのに対して，Windows では**連続したユーザアドレス空間を，カーネル空間で確保した連続領域にマップすることしか出来ない．**つまり，Linux カーネルの fault ハンドラ (nopage ハンドラ) に相当するものが Windows には存在しない．Linux カーネルにおいてデバイスドライバのプログラマが fault ハンドラ内で明示的に指定できたことは Windows カーネルが暗黙的に決めてしまう．

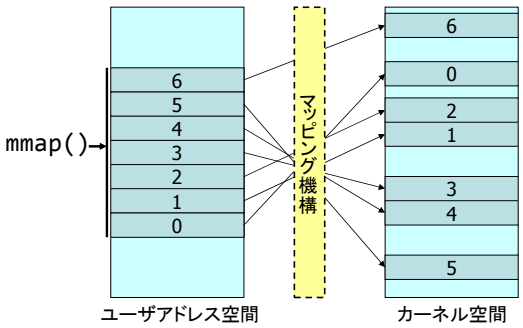

図 3 Linux のメモリ共有機構
Fig. 3 The memory sharing mechanism of Linux

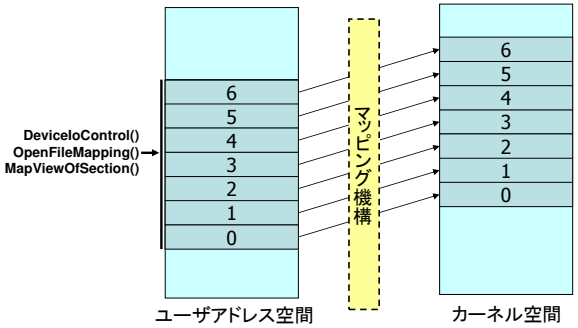

図 4 Windows のメモリ共有機構
Fig. 4 The memory sharing mechanism of Windows

このように，Linux と Windows のメモリ共有機構の性質はそれぞれ大きく異なっており，**これは Windows 上で Linux のメモリ管理機構を正確にエミュレーションする上で大きな障害となる．**このままでは QEMU 側と WinKVM ドライバ側との間でゲスト OS のメモリデータの不一致が起こり，通常のゲスト OS を動作させることも出来ず，もちろん，VM

ⓒ 2010 Information Processing Society of Japan

のライブマイグレーションを行うことも出来ない．

5. 解決手法の検討

4 章で述べた技術的課題に対する解決手法はいくつか考えられる．本章では，考えられる手法をすべて挙げ，最終的にわれわれが取った手法がもっとも妥当であることを示す．

問題点をまとめると，QEMU と KVM ドライバの間でどのようにメモリ同期を行うかが問題である．オリジナルの KVM は Linux の fault (nopage) ハンドラを使ってアプリケーション側 (QEMU) とカーネル側 (KVM ドライバ) のメモリマッピングを行うことでメモリの同期を行っている．KVM の内部にはゲスト OS 用メモリとカーネル内ページの対応表が存在している．しかし，Windows には fault ハンドラ相当の機能が存在しないため，エミュレーションレイヤで KVM ドライバと QEMU 間のメモリ共有をどのように行うかという問題点がある．考えられうる解決策は以下の 2 つである：

(A) システムコールによるメモリ同期 特定のタイミングで QEMU と KVM でメモリコピーを行うことでメモリを同期する手法である．コピーを行うタイミングは 2 つ考えられる：

(1) QEMU 側がゲスト OS のメモリ領域の読み書きをする際に

(2) ゲスト OS 実行直前と直後に QEMU と KVM ドライバ間で変更点のメモリコピーを行う．KVM ドライバがゲスト OS を実行し始めてからも，タイマー割り込みといった外部イベントが発生したときには，ゲスト OS の実行が中断され，KVM ドライバの実行が再開される．このタイミングでメモリのコピーによるメモリの同期を行う．

以上の 2 つである．これらの手法はメモリマッピングを行わずにメモリコピーで同期を取ることで，メモリマッピングと同様の効果を得ようとする手法である．

(B) KVM 自体を変更する KVM 自体に手を加え，Windows のメモリマッピング機構に適応するように変更する手法である．KVM は fault ハンドラを使ってメモリマッピングを行っているが．このハンドラを使用する変わりにエミュレーションレイヤが提供する独自機能を使用して，KVM が Windows のメモリマッピング機能を利用できるように変更してしまう．ちなみに，必要となる変更点はわずかに一行である．

われわれは最終的に (B) の手法を選択した．この手法の選択が最適であることを示すために議論を行う．(A).1 の場合，QEMU 側が 1 バイト書き込んだだけでも 1 つのシステムコールが発行されるためエミュレーションのパフォーマンスに悪影響を及ぼすことが容易に予想できる．また，QEMU は `mmap()` を利用してゲスト OS のメモリ領域をポインタとして扱っており，VM デバイスのエミュレーションを担当する多くの部分がこのポインタに対して読み書きを行っている．これらのメモリ読み書きをすべてフックし KVM ドライバ側に伝えるためには QEMU 自身を大幅に変更する必要がある．これは，(B) の手法で KVM に適応しなければならない変更と比べると多い．(A).2 の場合，ゲスト OS 実行する直前と直後にゲスト OS のメモリ領域をコピーして同期するするため (A).1 と比べてメモリ同期のためのシステムコール発行の回数を減らすことはできる．しかし，一般に巨大な VM メモリの全領域コピーが頻発するため，(A).2 と同様にパフォーマンスに悪影響を及ぼす．(B) は KVM 自身に改造を加えなければならないものの，Windows のメモリマッピング機構が利用してメモリ共有を行うため，(A) の手法と比べるとパフォーマンスへの悪影響も作業コストも小さいと考えられる．そのため，(B) の手法を選択した．

なお，既存研究では[1] (A.1) の手法を使用していたが完成度が低かった．そのため，DMA の挙動を正しくエミュレーションできず，HDD にアクセスを行わない RAM ディスク版の Linux イメージしか起動することができなかった．しかし今回 (B) の手法を採用したことで (A.1) と比べて低い作業コストで完成度の高い WinKVM を実現することができた．

6. 解 決 手 法

本章では 5 章にて決定した解決手法を実際にどのようにエミュレーションレイヤに実装したかについて述べる．結論から言うと，この問題を解決するためには KVM のソースコードにパッチを当てる必要がある．KVM に当てたパッチとエミュレーションレイヤが協調動作することで，KVM が Windows のメモリマッピング機構を使用できるようになる．

図 5 にエミュレーションレイヤと KVM に適応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 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

ⓒ 2010 Information Processing Society of Japan

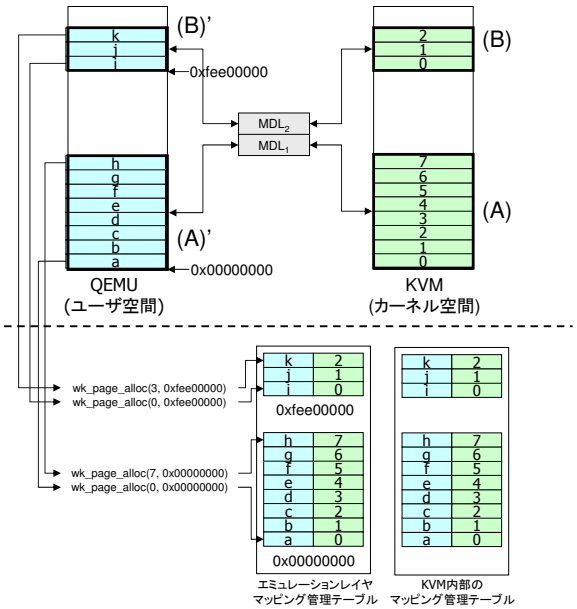

図 5 エミュレーションレイヤの機構
Fig. 5 The mechanism of our emulation layer

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処理時に一度だけ行われるものである．

始めに，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．例えば，ユーザがゲスト OS に 300MB のメモリ領域を割り当てた場合，300MB 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．これらの割り当ては 4 章でも述べたように Windows カーネルが暗黙的に決定する．これらの割り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．

次にパッチの出番である．パッチは，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函数である `kvm_main.c` 内の `kvm_vm_ioctl_set_memory_region()` にパッチを適応した．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()` と呼ばれる函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 が割り当てられているため，11 回 `alloc_pages()` 函数が呼ばれて 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づ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()` 函数を `wk_alloc_pages(pnum, base)` に置き換える．この函数は 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二つの引数を受け取り，すでに存在するマッピングと整合性が取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()` 函数は，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()` 函数であるといえる．

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

*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．

ⓒ 2010 Information Processing Society of Japan

グテーブルが完全に同一のものとなる．これで，KVM が Windows のメモリマッピング機構を使えるようになる．なお，KVM 内の fault 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．初期化以外のタイミングで KVM 内部のマッピングテーブルが変更されることはないので，同期は初期化時に一度行うだけでよい．

7. 評　　価

大きく分けて 2 つの評価を行った．

開発環境とテストに使った動作環境を以下のとおりである．開発環境は，Visual Studio 2008，Windows Server 2003 Service Pack 1(SP1) DDK と Cygwin gcc 3.4.4 を使用した．動作環境は，OS が Windows XPSP3 と CentOS 5.4 (Kernel 2.6.18-165)．CPU が Core 2 Duo T7300 @ 2.00Ghz．HDD は WD2500BEVS (5400rpm, キャッシュ8MB)．RAM は 2GB である．評価に使用したゲスト OS は Linux でカーネルのバージョンは 2.6.20 である．なお，移植に使用した KVM は KVM-17 である．

7.1 Linux カーネル起動による評価

本稿で述べた移植手法が正しく動作することを確認するために，WinKVM 上でゲスト OS を動作させた．ゲスト OS としては QEMU に付属しているテスト用の Linux イメージを使用した．カーネルのバージョンは 2.6.20 である．

評価の結果として Linux 2.6.20 を起動させ bash シェルが起動することを確認することができた．既存研究の手法では，ブートデバイスとしてハードディスクの変わりに RAM ディスクを指定した Linux イメージしか起動させることができなかった．しかし，今回のテストでは起動デバイスとしてハードディスクを指定した Linux イメージをゲスト OS として起動させることに成功した．これは，本稿で述べたメモリ共有の手法が正しく動作しているため，結果として，QEMU 側の DMA エミュレーションが正しく動作することで Linux カーネルがハードディスクにアクセスできるようになったためである．

7.2 各種ベンチマークによる評価

移植した KVM がエミュレーションレイヤから受ける影響を調べるため，オリジナルの KVM とエミュレーションレイヤ上で動作する KVM (WinKVM) 上でベンチマークプログラムを走らせて比較を行った．

一般的に，VMM 上のベンチマークでは時計が不正確になり正確な処理時間を測定することが難しい．そのため，今回では KVM-17 上に独自のハイパーコールを実装してホスト OS 側で時間の測定を行うことでこの問題を解決した．具体的には，ゲスト OS 内から I/O ポート (0x340 番) を読むと，ホスト OS の現在時刻が返るようにした．WinKVM 側にも同様のことを行おうとしたが，現時点ではエミュレーションレイヤの完成度が低く I/O ポートエミュレーションが不正確であり，ハイパーコールを実装することができなかった．そのため，WinKVM 上での測定値は著者自身がストップウォッチを使用して測定した．

大きく分けて 2 つのタイプのベンチマークを用意した．1 つ目のベンチマークは整数計算やビット演算が主な処理内容である．これらのプログラムはページフォルトや I/O 操作といったエミュレーションレイヤと KVM が介入する処理をほとんど引き起こさないプログラムである．そのため，エミュレーションレイヤによる影響をより詳細に調べるために，2 つ目のベンチマークは大量のページフォルトや I/O 操作を引き起こしエミュレーションレイヤや KVM の処理が大量に引き起こされる可能性のあるプログラムを用意した．

すべてのベンチマークの値は 3 回測定した内の最小値である．参考として物理マシン上で走らせたベンチマークの結果も掲載している．なお，測定した数値は小数点第一位で切り捨てた．

7.2.1 整数計算やビット演算のベンチマーク

表 2 に結果を示す．ベンチマークプログラムには nbench[7] と呼ばれるプログラムに手を加えたものを利用した．nbench は一秒間に何回のループを実行できたかを結果として出力する．しかし，今回は，処理に要した時間を測定するために，すべてのベンチマークが 20 秒以上かかるようにパラメータを調整するという改造を行った．また，KVM-17 上で動作させる nbench にはハイパーコールを用いて時刻を計測する改造も行った．

表の右端の「KVM-17 との差」は KVM-17 の処理時間から WinKVM の処理時間を引いたものである．負数が性能低下を示し，正数が性能向上を示す．いくつかのベンチマークプログラムで WinKVM のほうが速いという結果が出ているが，これは十分に予測できる結果である．WinKVM が動作する Windows と KVM-17 が動作する Linux では，リアルタイムクロックの割り込み値が異なる．Windows の割り込み値は Linux よりも遅いので，Windows のほうがタイマー割り込み発生による VM-exit の発生回数が少ない．単位時間当たりのゲスト OS 実行時間は Windows のほうが長いため，このような結果が出ると考えられる．

ベンチマーク結果をまとめると，最大で-4.1 秒の性能低下が見られ，最大で+0.5 秒の性能向上が見られた．先に述べたように，この性能向上は Windows カーネルの性質によるものである．

ⓒ 2010 Information Processing Society of Japan

表 2 WinKVM，KVM と実マシン上での各種ベンチマークの結果
Table 2 The results of our benchmarks with WinKVM, KVM and native

ベンチマーク名	Native (sec.)	KVM-17 (sec.)	WinKVM (sec.)	KVM-17 との差 (sec.)
NUMERIC SORT	29.1	30.3	30.5	-0.2
STRING SORT	37.1	38.2	38.1	+0.1
BITFIELD	35.9	36.6	36.6	0
FP EMULATION	22.7	22.8	22.6	+0.2
FOURIER	41.2	42.1	42.6	+0.2
ASSIGNMENT	30.8	30.8	31.1	0
IDEA Encryption	22.9	23.0	23.6	-0.1
HUFFMAN	27.3	28.0	28.6	-0.3
NEURAL NET	30.8	31.8	31.6	+0.5
LU DECOMPOSITION	31.8	48.9	53.9	-4.1

7.2.2 ページフォルトや I/O 操作を大量に引き起こすベンチマーク

表 3 に結果を示す．これらのベンチマークプログラムはわれわれが独自開発したものである．MMAP ベンチマークは `mmap()` で 200MB 分のメモリ領域を割り当て，繰り返し線形に読み書きを行うベンチマークである．繰り返し回数は 1000 回である．FILE COPY ベンチマークは `system()` 函数を用いて dd を呼び出し，`/dev/zero` から 200MB のファイル生成を繰り返すベンチマークである．繰り返し回数は 20 回である．参考までに物理マシン上のベンチマークも掲載した．同様に数値は小数点第一位で切り捨ててある．ただし，Native 環境の FILE COPY の測定結果だけは 1 秒以下の値なので測定数値をそのまま掲載する．

KVM-17 と比べて，WinKVM は MMAP ベンチマークの結果で-0.3 秒の性能低下が見られた．次に，FILE COPY ベンチマークにおいて，WinKVM は-124.3 秒の性能低下を引き起こしている．これは，7.2 節でも述べたエミュレーションレイヤの完成度の低さが引き起こす結果である．エミュレーションレイヤのバグにより不必要な I/O 操作が大量に引き起こされ，その結果としてこのような性能低下が起こっている．そのため，これは，本稿で述べた手法の本質的な限界を示しているものではない．

8. 関 連 研 究

Porting Linux KVM to FreeBSD[8] 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 の移植を考える上では FreeBSD と Linux のメモリ管理機構にそれほどの違いはないため，本稿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，本研究のように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．

表 3 ベンチマーク 1 の実行結果
Table 3 The results of the benchmark 1

実行環境	MMAP (sec.)	FILE COPY (sec.)
Native	55.0	0.067
KVM-17	54.0	43.2
WinKVM	54.5	167.5

2 つ目の研究としては，KVM のライブマイグレーション機構を高速化するという研究[9] が挙げられる．これは，プレコピーよりも高速に VM ライブマイグレーションが行えるポストコピーと呼ばれる手法を KVM 上に持ち込んだ研究である．この研究では VMEM と呼ばれるデバイスドライバを使用して，KVM 自体には手を加えずにプレコピーライブマイグレーションを実現している．この VMEM ドライバが Windows 上でも使用できれば，WinKVM 上でもプレコピーマイグレーションが導入できる．そして，VMEM ドライバはわれわれのエミュレーションレイヤ上で動作できるのではないかと考えている．そのため今後も注目していきたい．

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) を動作させるというプロジェクトが挙げられる．これらの研究は，それぞれ，

ⓒ 2010 Information Processing Society of Japan

Ndiswrapper[10),NDISulator (Project Evil)[11) 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 上で動作させるという研究がある[12)．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，この研究にも本稿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．

9. おわりに

われわれは，Linux と Windows 間の VM ライブマイグレーションを目標としている．VM ライブマイグレーションを動作させるためには，ダーティページトラッキング機能やネットワークを経由したメモリ転送機能が正しく動作する必要がある．これらの機能が正しく動作するためには，エミュレーションレイヤ上上で動作する KVM ドライバと QEMU との間でメモリ共有が正しく行われていることが前提である．

本稿ではこの問題点を解決し，KVM 側のゲスト OS のメモリアロケータを書き換え，KVM のメモリマッピングを管理する構造体と，われわれが開発したエミュレーションレイヤのマッピングを同期させることで，KVM がエミュレーションレイヤ上で Windows のメモリマッピング機構を使用できるようにした．なお KVM へのパッチは 1 行ですむ．

WinKVM に対していくつかの評価を行った．本稿で述べたメモリ共有の手法が正しく動作することを確認するために，WinKVM 上で Linux をゲスト OS として動作させたところ正常に動作した．また，エミュレーションレイヤが KVM 上で動作するゲスト OS への実行速度の影響を調べるために，KVM-17 と WinKVM 上で数値計算やビット演算のベンチマークプログラムを走らせた．その結果，WinKVM は KVM-17 と比較して最大で-4.1 秒の性能低下が見られ，また，最大で+0.5 秒の性能向上が見られた．

次に今後の課題について述べる．今回述べたメモリ共有の手法で WinKVM は VM ライブマイグレーションを行うことができるようになる．しかし，われわれは WinKVM の VM ライブマイグレーションをテストしておらず，この作業を進めることがさしあたっての目標である．

参 考 文 献

1) 笹田耕一高橋一志: WinKVM: 異なるホスト OS 間の VM ライブマイグレーション実現に向けて, 情報処理学会コンピュータシステム・シンポジウム論文集 (ComSys 2009), IPSJ, pp.59–66 (2009).
2) Kivity, A., Kamay, Y., Laor, D., Lublin, U. and Liguori, A.: kvm: the Linux Virtual Machine Monitor, *Proceedings of the Linux Symposium*, pp.225–230 (2007).
3) Neiger, G., Santoni, A., Leung, F., Rodgers, D. and Uhlig, R.: Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization, *j-INTEL-TECH-J*, Vol.10, No.3, pp.167–177 (2006).
4) AMD, Inc.: AMD Secure Virtual Machine Architecture Reference Manual.
5) Nelson, M., Lim, B.-H. and Hutchins, G.: Fast transparent migration for virtual machines, *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, Berkeley, CA, USA, USENIX Association, pp.25–25 (2005).
6) Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I. and Warfield, A.: Live migration of virtual machines, *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, Berkeley, CA, USA, USENIX Association, pp.273–286 (2005).
7) Uwe F. Mayer.: nbench, `http://www.tux.org/~mayer/linux/bmark.html` (2009).
8) Checconi, F.: LKVM, `http://feanor.sssup.it/~fabio/freebsd/lkvm/` (2007).
9) 広渕崇宏, 中田秀基, 伊藤智, 関口智嗣: 瞬間的な実行ホスト切り替えを可能とする仮想マシンの高速ライブマイグレーション機構, インターネットカンファレンス, 日本ソフトウェア科学会, pp.57–66 (2009).
10) Fuchs P, Pemmasani G.: Ndiswrapper, `http://ndiswrapper.sf.net/` (2005).
11) Paul B.: NDISulator, `http://www.freebsd.org/doc/en/books/handbook/config-network-setup.html` (2003).
12) 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).

ⓒ 2010 Information Processing Society of Japan