Vol.2011-OS-119 No.9
Vol.2011-EMB-23 No.9
2011/11/29

ホスト型仮想計算機における メモリ管理のアウトソーシングの提案

戸祭 要[†1] 新城 靖[†1] 齊藤 剛[†1]
豊岡 拓[†1] 板野 肯三[†1]

ホスト型仮想計算機においては、ホストOS（Operating System）とゲストOSでメモリ管理が重複することで様々な問題が発生する。そのような問題には、大域的なメモリ割り当ての最適化が難しいこと、同じページを2重にページアウトしようとしてしまうこと、および、ゲストのページをメインメモリにピン止めができないことを含む。この問題を解決するためにこの論文は、ゲストOSのメモリ管理をホストOSにアウトソースすることで、メモリ管理の重複を無くすことを提案している。アウトソーシングとは、従来の準仮想化とは異なりゲストOSの高水準のモジュールを置き換え、ゲストOSからホストOSの機能を利用可能にする手法である。従来、アウトソーシングはホスト型仮想計算機の入出力性能を改善するために用いられる手法であったが、この論文では、アウトソーシングを非入出力であるメモリ管理に適用している。提案方式は、ホストOSとゲストOSともにLinuxで実装を行っている。現在までに、ゲストOSからホストOSの管理するメモリをシステムコールにより利用すること、および、ゲストのページをメインメモリにピン止めすることが動作している。

Proposal for Memory-Management-Oustsourcing in Hosted Virtual Machines

Kaname Tomatsuri,[†1] Yasushi Shinjo,[†1] Go Saito,[†1]
Hiraku Toyooka[†1] and Kozo Itano[†1]

In hosted virtual machines, memory management duplication between the host operating system (OS) and guest OSes causes various problems. These problems include the difficulty of global memory allocation optimization, duplicated paging out of same pages, and inability to pin guest pages in a main memory. To address these problems, this paper proposes outsourcing the memory management of a guest OS to the host OS and eliminating some memory management duplication. Unlike conventional paravirtualization, outsourcing replaces high-level guest OS modules and enables a guest OS to use host OS functions. While the previous work about outsourcing tackles improving I/O performance, this paper applies outsourcing to memory management, which does not perform I/O. The proposed method is being implemented in Linux as a host and a guest OS. The current implementation allows a guest OS using a host memory through a system call, and pining guest pages in a main memory.

1. はじめに

1台の計算機を論理的に分割し、同時に複数のOSを動作させる技術に、仮想計算機モニタ[4]がある。仮想計算機モニタは、仮想計算機を構築・管理・動作させるソフトウェアである。仮想計算機モニタを用いる利点として、複数の計算機を集約することによる設置・管理コストの削減や、計算機を論理的に分割することによる計算資源の柔軟な運用、異種OSの同時使用が可能になることが挙げられる。

仮想計算機モニタのアーキテクチャの一つに、ホスト型がある。ホスト型の仮想計算機モニタでは、実際の計算機上でOSが動作し、そのOSの上で仮想計算機が動作する。一般的に、ホスト型仮想計算機では、ホストOSとゲストOS、それぞれ別々にメモリ管理を行っている。メモリ管理が重複して行われることにより、様々な問題が生じる。例えば、システム全体で最適なメモリ割り当てを行うことができない問題やホストOSとゲストOSで重複してメモリが回収される問題、メモリのピン止めなどのメモリの属性変更がホストOSに通知されない問題がある。詳しくは、3章で述べる。

これらの問題を解決するために本研究では、アウトソーシングという手法を用いてメモリ管理の重複を無くすことを提案する。アウトソーシングとは、仮想計算機上で動作するゲストOSの処理を高いレベルでホストOSに移譲する手法である[3),6),9),10]。従来アウトソーシングは、仮想計算機の入出力を高速化するための手法として用いられていた。この論文では、入出力ではないメモリ管理に対してアウトソーシングを適用し、効率化することについて述べる。これによりホスト型仮想計算機モニタにおいて、ホストOSとゲストOSでメモリ管理が重複して行われることにより生じた様々な問題を解決する。さらに、ファイルシステムのアウトソーシングと組み合わせることにより、ゲストOSにおけるメモリマップトファイルのホストOSによる高速な処理を実現する。また、提案方式をゲストOSとホス

†1 筑波大学システム情報工学研究科コンピュータサイエンス専攻
Department of Computer Science, University of Tsukuba

© 2011 Information Processing Society of Japan

Vol.2011-OS-119 No.9
Vol.2011-EMB-23 No.9
2011/11/29

ト OS として Linux を対象として実装を行っている。現在までに、ゲスト OS からホスト OS の管理するメモリをシステムコールにより利用すること、および、ゲストのページをメインメモリにピン止めすることが動作している。

本論文は次のように構成される。2 章では、アウトソーシングについて述べる。3 章では、仮想計算機におけるメモリ管理の問題点を明らかにする。4 章では、提案方式について述べる。5 章では、Linux と Linux KVM における実装について述べる。6 章では、実験結果を示す。7 章では、この論文のまとめを行う。

2. アウトソーシングとは

アウトソーシングは、ゲスト OS の高水準な処理をホスト OS に移譲することで仮想計算機の入出力を高速化するために用いられる[3),6),9),10)]。仮想計算機の入出力を高速化する手法として、準仮想化（paravirtualization）がある[2),7)]。準仮想化では、ゲスト OS のデバイスドライバという低いレイヤーのモジュールを置き換えることにより、デバイスのエミュレーションのオーバヘッドを削減している。一方、アウトソーシングでは、高水準なモジュールを置き換える。例えば、ファイルシステムのアウトソーシングでは、ゲスト OS の VFS（Virtual File System）層のモジュールを置き換え、ホスト OS の VFS 層のモジュールを呼び出している[10)]。これにより、準仮想化と同様にエミュレーションのオーバヘッドを避けることが出来る。

ファイルシステムのアウトソーシングでは、ホスト OS の資源を効率よく共有したり、ホスト OS やゲスト OS の優れた機能を利用することもできる。例えば、ファイルシステムのアウトソーシングの Linux 実装である WFS では、sendfile() のアウトソーシングを実現している。sendfile() は、ファイルディスクリプタ間のデータ転送をカーネル内で行うシステムコールである。準仮想化では、sendfile() はゲスト OS を介して行われるが、WFS はホスト OS で処理を完結させることにより高速化を実現している。また、ソケットアウトソーシングと組み合わせることにより、Web サーバが発行するようなファイルからソケットへの sendfile() でも、ホスト OS 内で処理を完結させることで高い性能を得ている。

アウトソーシングでは、ゲスト OS の高水準モジュールを置き換え、ホスト OS の機能を利用可能にする。ゲスト OS とホスト OS 間の通信は、仮想計算機に特化した RPC（Remote Procedure Call）である VMRPC(Virtual Machine RPC）を利用する[3),9)]。VMRPC は RPC の他に、ゲスト OS・ホスト OS 間の共有メモリ、キュー、および、ホスト OS からゲスト OS への割り込み注入の機能がある。

本研究では、入出力ではなくメモリ管理にアウトソーシングの手法を適用する。

3. ホスト型仮想計算機におけるメモリ管理の問題点

この章では、従来のホスト型仮想計算機におけるメモリ管理の概要を示し、その問題点を示す。

3.1 従 来 方 式

多くのホスト型仮想計算機では、仮想計算機モニタが仮想計算機の起動時に使用するメモリをホスト OS から確保する。仮想計算機上のゲスト OS は、仮想計算機モニタから割り当てられたメモリを物理メモリとして扱い、実計算機上で動作している時と同様に管理を行う。

メモリ管理には、次のようなものがある。

- ユーザプロセスにアドレス空間を割り当てる。
- ユーザプロセスやカーネル内のモジュールに物理メモリを割り当てる。
- 利用可能なメモリが不足した時に、ユーザプロセスの利用頻度の低いメモリを回収する。
- ユーザプロセスのメモリ属性を変更する。例えば、ページアウトされないようにピン止め（pinning）する。

通常、ホスト型仮想計算機では、ホスト OS は、仮想計算機に割り当てたメモリを回収することが難しい。これは、仮想計算機モニタが管理している仮想計算機が使用するメモリのアドレス変換表と、OS のアドレス変換表との間に齟齬が発生してしまうからである。

Linux には、この問題を解決する機能として MMU Notifier[1)] がある。MMU Notifier は、メモリ回収などの特定の操作が行われた際に、登録された関数を呼び出す機構である。Linux KVM[5)] では、この機構を利用し、ホスト OS がゲスト OS の使用するメモリを回収した際、Linux KVM が管理しているアドレス変換表のうち、回収されたアドレスの変換表を無効化する関数を呼び出すことで、ホスト OS によるゲスト OS の使用しているメモリの回収を可能にしている。

本研究の目的は、ゲスト OS とホスト OS で重複したメモリ管理が行われることにより発生する問題を解決することである。

3.2 ホスト型仮想計算機におけるメモリ管理の課題

ホスト型仮想計算機モニタでは、ゲスト OS とホスト OS で重複したメモリ管理が行われる。これにより、次のような問題が生じる（図 1）。

© 2011 Information Processing Society of Japan

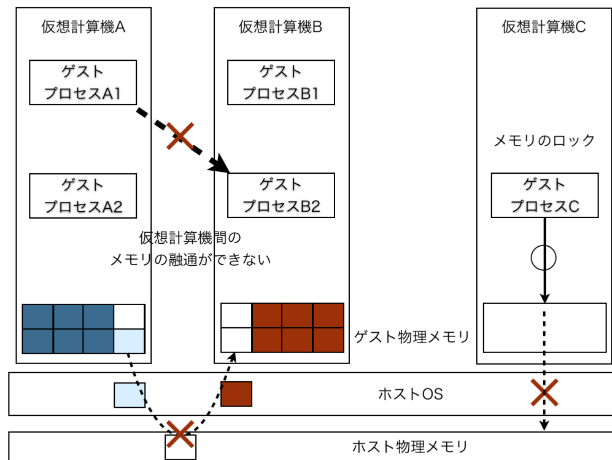

図 1 ホスト型仮想計算機におけるメモリ管理の課題

3.2.1 大域的なメモリ割り当ての最適化

複数のゲスト OS が動作している時、各ゲスト OS は非仮想環境で動作していると思い、自身に割り当てられた物理メモリを全て有効活用しようと試みる。例えば、仮想計算機 A のゲスト OS A で、プロセス A1 とプロセス A2 が動作しており、仮想計算機 B のゲスト OS B で、プロセス B1 とプロセス B2 が動作していたとする。ゲスト OS A は、その中で LRU（Least Recently Used）等のアルゴリズムを動作させ、メモリの割り当ての最適化を行う。その結果、プロセス A1 のメモリを減らしてプロセス A2 のメモリを増やしたとする。ゲスト OS B でも、同様に、プロセス B1 のメモリを減らして、プロセス B2 のメモリを増やしたとする。このような動きは、システム全体では必ずしも最適になってるとは言えない。システム全体で見ると、プロセス A1 のメモリを減らしてプロセス B2 のメモリを増やした方が良いかもしれない。しかしながら、既存の手法ではこのような最適化は行われていない。

仮想計算機間でメモリを融通しあう仕組みとして、バルーニング[8] がある。バルーニングでは、ゲスト OS 内でバルーンドライバと呼ばれる特殊なプログラムを動作させる。ある仮想計算機に割り当てたメモリを減らすには、その仮想計算機で動作しているバルーンドライバでメモリを確保する。仮想計算機モニタは、バルーンドライバが確保したメモリから物理メモリを奪い取る。

バルーニングというメカニズムを使うと、仮想計算機に割り当てたメモリを回収することは可能になる。しかしながら、システム全体で見たとき、どの仮想計算機にどのくらいのメモリを割り当てれば良いかというポリシーについては問題が残されている。

複数の仮想計算機が使用するメモリを最適化する仕組みに、Kernel Samepage Merging や Transparent Page Sharing[8] がある。これらの機能は、重複する内容のメモリを探し出し、そのメモリを参照するページテーブルエントリを同じメモリを参照するように書き換え、不要なメモリを回収する。しかし、回収できるメモリが内容が重複するメモリに限られており、ページが不足した時に、利用頻度が低いページを回収することができない。

VMware の仮想計算機間のメモリ融通技術として、Elastic Memory for Java がある。これは、複数の仮想計算機上で動作する JVM（Java Virtual Machine）のヒープメモリ領域を融通しあう技術である。しかし、融通しあえる範囲がゲスト OS 上の JVM に限定されている。

本研究では、ホスト OS が持っているメモリ割り当ての最適化を行う機能を有効に活用することでこの問題を解決する。

3.2.2 ダブルページング

OS はメモリ資源を有効活用し、OS やプロセスからのメモリ確保要求に応えるためにメモリの回収を行う。ホスト型仮想計算機においては、ゲスト OS とホスト OS で別々にメモリ回収を行っている。例えば、ゲスト OS とホスト OS が同じメモリ領域を不要と判断し、ホスト OS が先にメモリを回収しディスクに書き出し（ページアウト）したとする。その後、ゲスト OS がページアウトを試みると、ホスト OS は回収したメモリを再度物理メモリ上に配置（ページイン）する必要がある。ホスト OS のページインが完了した後に、ゲスト OS がページアウトを行うので、システム全体で見ると余分なページアウト・ページインが発生してしまう。

また、ページアウトの際、回収したメモリをディスクに書き出す必要があるが、ホスト型仮想計算機は入出力が遅い問題がある。そのため、ゲスト OS のページアウト・ページインに時間がかかり性能が低下する問題がある。

本研究では、ゲスト OS のメモリ回収の一部を停止し、ホスト OS のメモリ回収の機能を利用することでこの問題を解決する。

3.2.3 メモリの属性変更

アプリケーションはメモリの属性変更の要求を発行することがある。例えば、アプリケーションがメモリのロックを要求した時、OS は要求されたメモリのロックを行う。一方、ホ

© 2011 Information Processing Society of Japan

スト型仮想計算機では、ホスト OS とゲスト OS で別々にメモリ管理を行っている。そのため、例えばゲスト OS 上のアプリケーションがメモリのロックを要求した場合、ゲスト OS が管理しているメモリには要求が適用されるが、ホスト OS が管理しているメモリには適用されない。そのため、ホスト OS が仮想計算機に割り当てたメモリの一部を回収してしまう可能性がある。ゲスト OS 上のアプリケーションがメモリがロックされていることを前提にして動作している場合、動作に問題が生じる可能性がある。

本研究では、ゲスト OS がホスト OS にメモリのロック要求を通知し、ホスト OS でもロックすることでこの問題を解決する。

3.2.4 メモリマップトファイル

プロセスが自身のメモリ空間を操作する方法の一つにメモリマップトファイルがある。メモリマップトファイルは、ファイルの一部あるいは全部をプロセスのアドレス空間にマッピングし、通常のメモリアクセスのようにファイルにアクセスすることを可能にする。

ファイル入出力性能を改善するファイルシステムのアウトソーシング[10) においては、メモリマップトファイルはホスト OS とゲスト OS の行き来が発生し、オーバヘッドの大きい問題がある。例えば、ファイルがマッピングされたメモリ領域へ初めてアクセスされ、ページフォルトが発生したとする。まず、ページフォルトハンドラで、ゲスト OS 内のページキャッシュを確認し、キャッシュが存在すればそれを利用する。存在しなければ、VMRPC を用いてファイルの読み込み命令を発効する。ホスト OS でも、同様にページキャッシュを確認し、キャッシュが存在すればそれを利用し、存在しなければファイルからの読み込みを行う。このように、ゲスト OS に処理が移りオーバヘッドが大きくなってしまう問題がある。

本研究では、ファイルシステムのアウトソーシングと組み合わせ、メモリマップトファイルをホスト OS で一括して処理を行うことでこの問題を解決する。

4. 提案手法

この章では、第 3 章で述べた問題を解決するため、メモリ管理のアウトソーシングを提案する。

4.1 メモリ管理のアウトソーシングの概要

本手法では、ゲスト OS のメモリ管理機能の一部を無効化し、ゲスト OS から VMRPC を用いてホスト OS のメモリ管理モジュールに処理を渡す。具体的に、次の機能をゲスト OS からホスト OS へ移譲させる（図 2）。

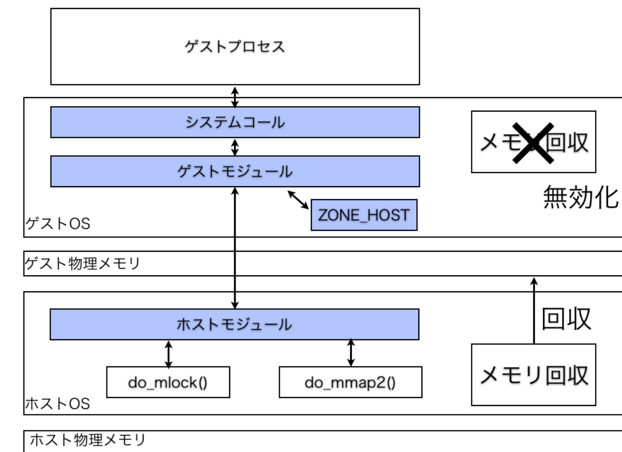

図 2 メモリ管理のアウトソーシング概要

ホスト OS によるメモリ割り当ての集中管理　ゲスト OS で行われていたメモリ割り当ての最適化をホスト OS で集中的に行う。

メモリの属性変更　ゲスト OS 内やゲスト OS 上のプロセスからメモリの属性変更要求を受け取る。それを、VMRPC を用いてホスト OS 内で処理し、メモリの属性を変更する。

メモリマップトファイル　ファイルシステムのアウトソーシング[10] と組み合わせ、メモリマップトファイル時の処理をホスト OS で完結させることで高速化を実現する。

4.2 ホスト OS によるメモリ割り当ての集中管理

メモリ管理のアウトソーシングでは、ゲスト OS で行われていたメモリ割り当ての最適化をホスト OS で集中的に行うようにする。これにより、ホスト OS のレベルで見たときに最適なメモリ管理を可能にする。それには、以下のようなことを行う。

- 従来、ゲスト OS で必要なメモリの割り当ては、そのゲスト OS に割り当てていた物理メモリから行われていた。その部分を変更し、ホスト OS のメモリから割り当てるようにする。ホスト OS から割り当てられたメモリが不要になったらホスト OS に返却する。
- ゲスト OS で行っているメモリ割り当ての最適化に関わる処理を停止する。具体的には、LRU 等のアルゴリズムによるページの回収を行わないようにする。その結果、ページアウトの処理は、ホスト OS のみで行われることになる。ページアウトされたメモリにゲスト OS のプロセスがアクセスした時、ホスト OS はページインの処理を行う。最終

© 2011 Information Processing Society of Japan

的に、ホスト OS においてシステム全体のメモリ割り当ての最適化がなされる。また、ゲスト OS ではページアウトの処理が行われなくなる。したがって、3.2.2 で述べたダブルページングの問題も発生しなくなる。

4.3 メモリの属性変更

メモリ管理のアウトソーシングでは、ゲスト OS とホスト OS で別々に行われたメモリの属性管理を統一する。これにより、ゲスト OS のアプリケーションが意図するメモリの属性変更を可能にする。具体的には、ゲスト OS 内でメモリの属性変更の要求を受け取り、VMRPC を用いてホスト OS 内でメモリの属性変更の処理を行うようにする。

4.4 メモリマップトファイル

メモリ管理のアウトソーシングでは、ファイルシステムのアウトソーシングと組み合わせてメモリマップトファイルの処理をホスト OS で行うようにする。これにより、メモリマップトファイルの高速化を可能にする。それには、以下のようなことを行う。

従来、ファイルシステムのアウトソーシングでは、メモリマップトファイルの処理をゲスト OS で行っていた。その部分を変更し、ホスト OS で行うことで高速化を実現する。具体的には、ゲスト OS 内のモジュールがプロセスから発行されるメモリマップトファイルの要求を受け取る。ゲスト OS 内のモジュールは、マッピングするメモリ空間を確保し、VMRPC を用いてホスト OS 内のモジュールに処理を渡す。ホスト OS 内のモジュールは要求に従い、ファイルを適切なアドレスにマッピングする。そのマップされたメモリをゲスト OS のプロセスがアクセスしてページフォルトが起きた場合、ホスト OS が処理を行う。すなわち、メモリを割り当て、ディスク入力を要求し、それが完了したら処理を再開させる。

5. Linux における実装

現在、提案方式を仮想計算機モニタとして Linux KVM[5]、ホスト OS・ゲスト OS として Linux を用いて実装を行っている。

5.1 ホスト OS によるメモリ割り当ての集中管理

ここでは、ホスト OS によるメモリ割り当ての集中管理のために必要なメモリの割り当て・解放とメモリの回収の 2 つについて述べる。

5.1.1 メモリの割り当て・解放

Linux では、メモリ確保には、alloc_pages() 関数などが用いられる。alloc_pages() 関数は、OS 内のゾーンアロケータからメモリを確保する。ゾーンアロケータは、メモリをゾーンと呼ばれる単位ごとに管理している。よく使われるゾーンとしては、ZONE_DMA や

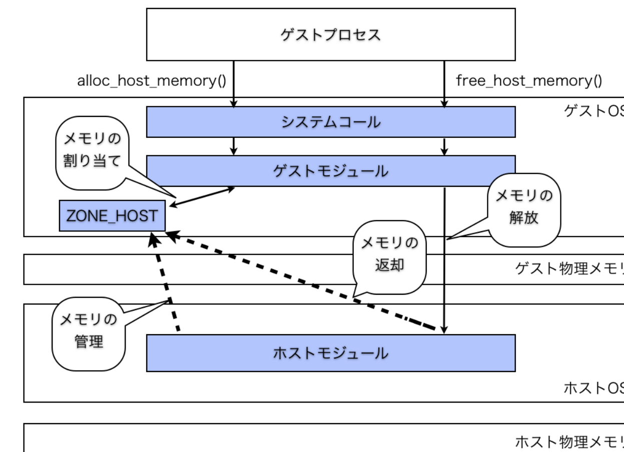

図 3 メモリの割り当てと解放

ZONE_DMA32、ZONE_NORMAL、ZONE_HIGHMEM がある。通常、ユーザプロセスのメモリは、ZONE_HIGHMEM や ZONE_NORMAL からメモリを確保する。

このメモリゾーンに新たに ZONE_HOST を追加する。ZONE_HOST 内のメモリの物理アドレスは、ゲスト OS の起動完了後にホスト OS 内のメモリ管理モジュールに通知される。ホスト OS は、通知されたアドレスのメモリをゲスト OS から奪い取り、ホスト OS 主導で割り当て・解放を行う。加えて、ホスト OS からメモリを取得するための新たなシステムコールを追加する（図 3、表 1）。このシステムコールでは、VMRPC を用いてホスト OS にメモリの割り当て要求を発行する。ホスト OS 内のメモリ管理モジュールは、要求に従いゲスト OS から管理を移譲されたメモリの中から割り当てを行い、ゲスト OS が使用しても良いメモリを返す。メモリの解放も、同様に解放のための新たなシステムコールを追加し、メモリの解放要求をホスト OS に通知し、ホスト OS で解放を行う。

5.1.2 メモリの回収

3 章で述べたように、Linux KVM は MMU Notifier を利用することで、ホスト OS によるゲスト OS のメモリ回収が可能になる。Linux KVM は、仮想計算機を作成する際、そのメモリ空間に対して MMU Notifier 用のコールバック関数を登録する。例えば、ゲスト OS が使用中のメモリがホスト OS に回収された時には、invalidate_page() 関数が呼び出され

© 2011 Information Processing Society of Japan

る。この invalidate_page() 関数の中で Linux KVM は、ゲスト OS が回収されたメモリのアドレスを使用できないようにする。ゲスト OS が回収されたアドレスに対してアクセスを行った場合は、ホスト OS によるページフォルト処理が行われる。このようにして、ゲスト OS が使用中のメモリをホスト OS が回収できる。

加えて、ゲスト OS のメモリ回収を一部無効化する。Linux では、いくつかのメモリ回収アルゴリズムを組み合わせて、メモリ回収を実現している。Linux のメモリ回収は、メモリの確保時など不定期に行われるものと定期的に行われるものがある。本提案方式では、メモリの確保をホスト OS で行うため不定期のメモリ回収はゲスト OS では行われない。Linux では、メモリの定期的な回収を行っている kswapd カーネルスレッドがある。この kswapd カーネルスレッドの機能を制限、無効化することでゲストでのメモリ回収を無効化する。

このように、ゲスト OS のメモリ回収を無効化することで、ホスト OS にシステム全体の最適化を実現する。また、ダブルページングの問題を解消する。

5.2 メモリの属性変更

ここでは、メモリの属性変更要求として、メモリのロックを例に説明する。Linux では、プロセスがメモリのロックを行う場合、mlock() システムコールが発行される。mlock() システムコールは、ゲスト OS 内の do_mlock() 関数を呼び出す。その do_mlock() 関数内で VMRPC を発行し、メモリのロック要求をホスト OS に通知する。ホスト OS 内のメモリ管理モジュールは、通知を受け取るとアドレス変換機能を利用し、ゲスト OS から渡されたロック対象のアドレスをホスト OS のアドレスに変換する。そして、変換したアドレスに対してホスト OS 内で do_mlock() 関数を呼び出し、メモリのロックを行う。

5.3 メモリマップトファイル

メモリマップトファイルは、ファイルシステムのアウトソーシング[10] と組み合わせ実装を行う。ゲスト OS のプロセスが mmap() システムコールを発行した時、対象のファイルがファイルシステムのアウトソーシングを利用して開かれたファイルかどうかを確認する。ファイルシステムのアウトソーシングを利用して開かれたファイルである場合、get_unmapped_area() 関数でマッピングを行うメモリアドレスを決める。その後、VMRPC を通じて、マッピングを行うメモリアドレスとマッピングを行うメモリ領域をホスト OS に通知する。ホスト OS は、ゲスト OS のアドレスからホスト OS のアドレス変換を行い、対象のアドレスに対して do_mmap2() 関数でメモリマップトファイルの処理を行う。

表 1 追加するシステムコール

システムコール API	機能
unsigned long alloc_host_memory(unsigned long size)	ホスト OS からのメモリ確保
void free_host_memory(unsigend long addr, unsigned long size)	ホスト OS から確保したメモリの解放

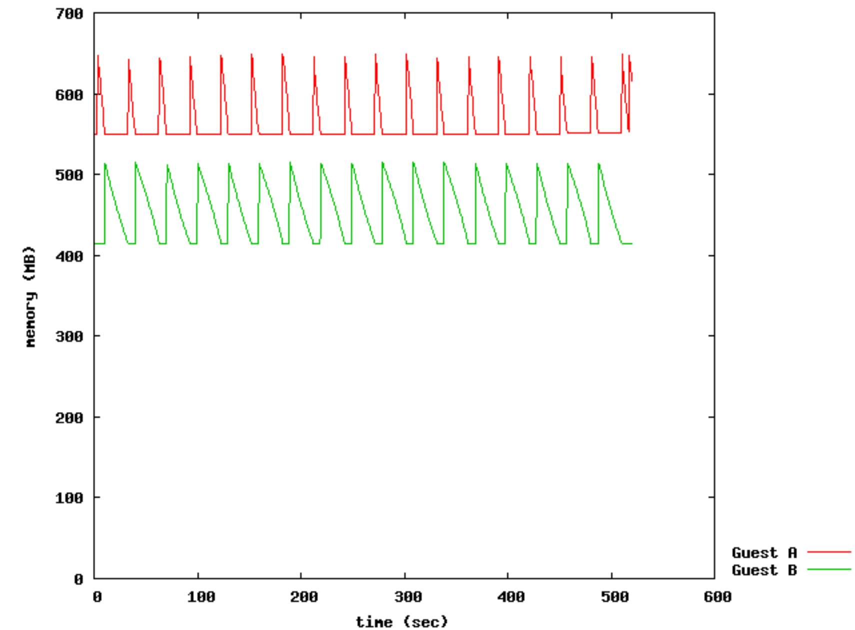

図 4 メモリ割り当てと解放の実験

6. 実　験

メモリの割り当てと解放について実装を行い、その動作の確認のため実験を行った。

6.1 実 験 環 境

実験を行う計算機は、Intel Core i7 2.67GHz の CPU、6GB のメインメモリを備えたものを用いた。ホスト OS・ゲスト OS には Linux 2.6.32 を用いた。また、仮想計算機モニタは Linux KVM を用いた。

6.2 メモリの割り当てと解放

現時点での実装を用いて、2 つの仮想計算機間でメモリの割り当てと解放を行う実験を行っ

© 2011 Information Processing Society of Japan

た。実験では、1GB のメモリを割り当てて起動させた 2 つの仮想計算機で、交互に 100MB のメモリ割り当てと解放を行った。ホスト OS で 2 つの仮想計算機が使用しているメモリ量とホスト OS 全体のメモリ使用量の変化を図 4 に示す。このメモリ使用量は、ホスト OS の cgroup のメモリ統計情報を用いて計測を行った。

図 4 で示すように、メモリの割り当て、解放に伴い、それぞれの仮想計算機のメモリ使用量が増減している。また、それぞれの仮想計算機のメモリ使用量の増加、減少とホスト OS 全体のメモリ使用量の増加が減少が同時に起こっており、ホスト OS からのメモリ割り当てと解放と仮想計算機間のメモリの融通が行われていることが分かる。なお、2 つの仮想計算機で使用しているメモリ量のベースが異なるのは、バッファキャッシュが利用しているメモリ量の違いによるものである。

7. おわりに

この論文では、既存のホスト型仮想計算機におけるメモリ管理の問題点について述べ、その問題に対する解決策として、アウトソーシング方式を提案した。この方式では、ゲスト OS におけるメモリ管理機能の一部を、ホスト OS に処理を移譲することにより問題を解決する。大域的なメモリ割り当ての最適化とダブルページングの問題は、ホスト OS でメモリ割り当ての処理を行うこととゲスト OS でメモリの回収を行わずホスト OS が行うことで解決する。メモリの属性変更の問題は、ゲスト OS で発行されたメモリの属性変更の要求をホスト OS に通知し、ホスト OS で適用することで解決する。メモリマップトファイルの問題は、メモリを割り当て、ディスク入力を要求をホスト OS で行うことで解決する。

提案方式をゲスト OS とホスト OS として Linux を対象に実装している。仮想計算機モニタとしては、Linux KVM を用いた。現在までに、ゲスト OS からホスト OS の管理するメモリをシステムコールにより利用すること、および、ゲストのページをメインメモリにピン止めすることが動作している。

今後の課題として、本論文で提案したアウトソーシング方式の実装を引き続き行い、その評価を行うことである。

参考文献

1) Andrea Arcangeli. Integrating KVM with the Linux memory management. In *KVM Forum*, June 2008.
2) Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp. 164–177, New York, NY, USA, 2003. ACM.
3) Hideki Eiraku, Yasushi Shinjo, Calton Pu, Younggyun Koh, and Kazuhiko Kato. Fast networking with socket-outsourcing in hosted virtual machine environments. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pp. 310–317, New York, NY, USA, 2009. ACM.
4) RobertP. Goldberg. A survey of virtual machine research. *IEEE Computer*, Vol.7, No.6, June 1974.
5) Avi Kivity. KVM : the Linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pp. 225–230, July 2007.
6) Younggyun Koh, Calton Pu, Yasushi Shinjo, Hideki Eiraku, GoSaito, and Daiyuu Nobori. Improving Virtualized Windows Network Performance by Delegating Network Processing. *Network Computing and Applications, IEEE International Symposium on*, Vol.0, pp. 203–210, 2009.
7) Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, Vol.42, pp. 95–103, July 2008.
8) Carl A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, Vol.36, No.SI, pp. 181–194, 2002.
9) 齊藤剛, 新城靖, 榮樂英樹, 佐藤聡, 中井央, 板野肯三. 仮想計算機におけるアウトソーシングのためのゲスト-ホスト間 RPC. 情報処理学会　第 20 回コンピュータシステム・シンポジウムポスターデモセッション, 2008.
10) 豊岡拓, 新城靖, 齊藤剛. ホスト型仮想計算機環境におけるファイル入出力の vfs アウトソーシングによる高速化. 情報処理学会 第 21 回コンピュータシステムシンポジウム, November 2009.

© 2011 Information Processing Society of Japan