Vol.2009-ARC-183 No.33
Vol.2009-OS-111 No.33
2009/4/24

仮想マシン間プロセススケジューリングの実環境への適用にむけて

田所 秀和[†1]　光来 健一[†2,†3]　千葉 滋[†1]

我々は仮想マシンモニタが仮想マシン間にまたがるプロセススケジューリングを行う Monarch Scheduler を提案してきた．Monarch Scheduler では仮想マシンモニタから仮想マシン上のゲスト OS のランキューを操作することで，ゲスト OS のプロセススケジューリングを調整する．しかし，Monarch Scheduler を実環境へ適用する上でいくつかの問題があった．本論文では従来の Monarch Scheduler では制御するのが難しかった I/O バウンドプロセスを制御するための手法を提案する．また，Monarch Scheduler はゲスト OS に強く依存するため Linux にしか対応できていなかったが，Windows にも対応できるようにした．実験により，これらのプロセスをうまく制御できることを確認した．

Toward Applying Inter-Virtual-Machine Process Scheduling to Realistic Execution Environments

Hidekazu TADOKORO,[†1] Kenichi KOURAI[†2,†3]
and Shigeru CHIBA[†1]

We have been proposed Monarch Scheduler, which schedules processes among virtual machines. In Monarch Scheduler, the virtual machine monitor directly manipulates the run queues in guest operating systems running on virtual machines to adjust their process scheduling. However, there were several problems when we apply Monarch Scheduler to realistic execution environments. This paper proposes a method for controlling I/O-bound processes in Monarch Scheduler. Also, we have implemented the support for the Windows guest operating system. The previous Monarch Scheduler supported only Linux because it strongly depends on guest operating systems. We experimentally confirmed that we can control I/O-bound processes and Windows processes.

1. はじめに

近年，サーバの利用率を向上させるために，仮想マシンを用いて複数のサーバを一台のマシンに統合することが増えてきている．このようなシステムでは複数の仮想マシンが一台の物理マシンを共有することになるため，システム全体で優先度の高い処理と低い処理を考えることが必要になる．しかし，このようなシステム全体でプロセスを意識したスケジューリングを行うのは難しい．仮想マシン上で動くゲスト OS 内のスケジューリングだけでは，異なる仮想マシン上で動くプロセスに優先度をつけることができない．仮想マシン自体のスケジューリングを併用すれば仮想マシン間でプロセスの優先度をある程度制御することができるが，ゲスト OS のスケジューリングが少し変わるだけでうまく制御できなくなる．

このような問題を解決するために，我々は仮想マシンモニタが仮想マシン間にまたがるプロセススケジューリングを行う *Monarch Scheduler* を提案してきた[1)]．Monarch Scheduler では仮想マシンモニタから仮想マシン上のゲスト OS のランキューを操作することで，ゲスト OS のプロセススケジューリングを調整する．さらに，仮想マシンのスケジューリングおよびゲスト OS の既存のスケジューリングと連携することにより，システム全体でプロセスに優先度をつけることができる．Monarch Scheduler ではゲスト OS のカーネルを変更する必要がなく，スケジューリングポリシにおいてゲスト OS 上で動くどのプロセスも特別扱いする必要がない．

しかし，Monarch Scheduler を実環境へ適用する上でいくつかの問題があった．一つは I/O バウンドプロセスをうまく制御できないことである．I/O バウンドプロセスは I/O 待ちをしていることが多いため，ランキューで CPU を待っていることは少ない．Monarch Scheduler では定期的にランキューを操作することでプロセスの実行を制御するため，このようなプロセスを制御するのは難しかった．もう一つの問題は，Monarch Scheduler はゲスト OS に強く依存するため，現状では Linux にしか対応できていないことである．実環境では Windows も多くの場面で使われているため，Linux だけへの対応では不十分である．

†1 東京工業大学 情報理工学研究科 数理・計算科学専攻
Department of Mathematical and Computing Sciences, Graduate School of Information Science and Engineering, Tokyo Institute of Technology
†2 九州工業大学 情報工学研究院 情報創成工学研究系
Department of Creative Informatics, Kyushu Institute of Technology
†3 独立行政法人科学技術振興機構, CREST
JST, CREST

© 2009 Information Processing Society of Japan

Vol.2009-ARC-183 No.33
Vol.2009-OS-111 No.33
2009/4/24

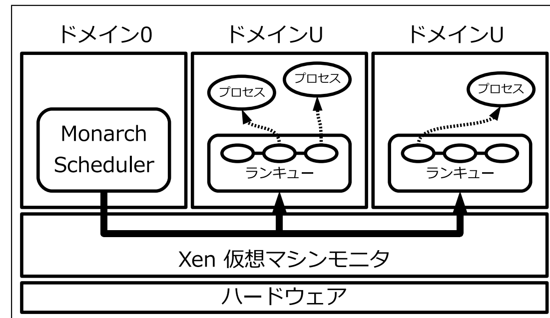

図 1 Monarch Scheduler の構成

しかし，Windows に対応させるには，Windows 内部の情報が非常に少ないことが障害であった．

本論文ではこれまでに提案してきた Monarch Scheduler を拡張して，I/O バウンドプロセスを制御できるようにし，Windows ゲスト OS への対応を行った．I/O バウンドプロセスを制御するために，CPU スケジューラのウェイトキューを操作する方法や I/O スケジューラのリクエストキューを操作する方法などを検討したが，これらの実現は難しいことが分かった．そこで，仮想マシンモニタが I/O 待ちしているプロセスの状態を強制的に変更する手法を提案する．Windows ゲスト OS においてランキューの位置を特定し，一貫性を保ってランキューを操作する方法は Linux のような直接的な方法が使えなかったため，様々な知識を用いて間接的に実現した．

以下，2 章ではこれまでに提案してきた Monarch Scheduler とその問題点について述べる．3 章では Monarch Scheduler が I/O バウンドプロセスを制御するための手法について述べる．4 章では Monarch Scheduler を Windows ゲスト OS に対応させるための手法について述べる．5 章では I/O バウンドプロセスおよび Windows ゲスト OS 上のプロセスを実際に制御できていることを確認した実験について述べる．6 章で関連研究を述べ，7 章で本稿をまとめる．

2. Monarch Scheduler

2.1 Monarch Scheduler の概要

Monarch Scheduler は仮想マシンモニタ上で動作し，システム全体のプロセスを意識して複数の仮想マシンにまたがったスケジューリングを行うスケジューラである．Monarch Scheduler では，**図 1** のように，仮想マシンモニタから仮想マシン上で動作しているゲスト OS の CPU スケジューラのランキューを操作することで，ゲスト OS のプロセススケジューリングを調整する．さらに，仮想マシンのスケジューリングおよびゲスト OS の既存のスケジューリングと連携することにより，システム全体でプロセスに優先度をつけることができる．

Monarch Scheduler の利点は，このようなシステム全体の協調スケジューリングを行う際に各ゲスト OS のカーネルを変更する必要がないことである．Monarch Scheduler では仮想マシンモニタがゲスト OS のカーネルメモリを直接書き換えることでランキューの操作を行なう．そのため，スケジューリングのための機能をゲスト OS に追加する必要がない．さらに，Monarch Scheduler のスケジューリングポリシではゲスト OS 上のどのプロセスも特別扱いする必要がない．協調スケジューリングにおいて情報をやりとりするためのプロセスをゲスト OS 上で動かすと，意図しない挙動を防ぐためにそれらのプロセスを特別扱いしなければならなくなる．Monarch Scheduler では仮想マシンモニタが各ゲスト OS の情報を直接取得するため，このようなスケジューリングのためのプロセスを動かす必要がない．

Monarch Scheduler は Xen[2] 上に実装されており，ゲスト OS として Linux に対応している．ドメイン 0 と呼ばれる特権仮想マシン内でスケジューラプロセスが動作し，ドメイン U と呼ばれる一般の仮想マシンのメモリをスケジューラプロセス内にマップすることでゲスト OS のランキューを操作する．Monarch Scheduler は一定時間ごとに全てのドメイン U を一時停止し，スケジューリングポリシに応じてゲスト OS のランキューを操作し，ドメイン U の実行を再開する．プロセスを一時的にランキューから取り除くことで停止させ，ランキューに挿入することで再開させる．このようなランキューの操作によって優先度に応じた CPU 割り当てを実現している．

Monarch Scheduler はゲスト OS 内のランキューを操作するために，ゲスト OS 内のランキューの位置を特定する．`x86_64` アーキテクチャにおける Linux のランキューの位置は CPU の `GS` レジスタによって指されているデータ構造からたどることができる．その際に必要なゲスト OS 内のデータ構造を取得するために，Monarch Scheduler はゲスト OS 内で使われている構造体の型情報を利用する．型情報はデバッグオプションつきでコンパイルした OS カーネルから GDB を使って取得する．

一貫性を保ってゲスト OS を操作するために，Monarch Scheduler では仮想マシンモニタからゲスト OS のカーネルがランキューを操作中でないかどうかをチェックする．カーネ

© 2009 Information Processing Society of Japan

Vol.2009-ARC-183 No.33
Vol.2009-OS-111 No.33
2009/4/24

ルがランキューをつなぎかえている間は一貫性のとれたデータ構造になっておらず，ランキューを操作するとゲスト OS のデータ構造を破壊してしまう．Linux カーネルではスピンロックを使ってランキューの排他制御を行っている．ゲスト OS のカーネルがランキューを操作していないことを保証するために，仮想マシンモニタはゲスト OS によってランキューのロックが取得されているかどうかを調べる．もしロックが取得されていたら，ゲスト OS のカーネルがランキューを操作していると見なす．この場合には，少しの間ドメイン U を再開した後で再度チェックする．

2.2 実環境に適用する上での問題

Monarch Scheduler を実環境に適応する上で，いくつかの問題があった．一つは，I/O をよく使う I/O バウンドプロセスをうまく制御できないことである．現実の多くのアプリケーション，特にサーバアプリケーションは CPU を使う処理のみを行っていることは少なく，多くの場合は I/O を行う．I/O バウンドプロセスは相対的に CPU 待ちをしていることが少なく，I/O 待ちをしていることが多い．そのため，プロセスがランキューで待っていることが少なく，Monarch Scheduler が定期的にランキューを調べても制御したいプロセスがランキューで CPU を待っている確率が低い．これまでに提案してきた Monarch Scheduler ではランキューを操作することでプロセスの実行を制御するので，あまりランキューで待たないプロセスを制御するのは難しかった．

もう一つの問題は，Monarch Scheduler がゲスト OS に強く依存するため，現状では Linux にしか対応できていないことである．実環境では Windows も多くの場面で使われており，Linux だけへの対応では不十分である．Windows に対応させるにあたって，Windows 内部の情報が非常に少ないことが問題になる．内部構造についての情報はある程度公開されているが，詳細については多くの部分が不明である．また，Windows リサーチカーネルのソースコードは公開されているものの，商用の Windows カーネルのソースコードを参照することはできない．そのため，ランキューの位置や操作方法などが不明である．また，Windows カーネル内のグローバル変数のアドレスはカーネルをロードするたびに変わるため，データ構造をたどるための始点を見つけるのも困難である．

3. I/O バウンドプロセスの制御

Xen において，ゲスト OS はドメイン 0 を経由して I/O デバイスにアクセスする．ゲスト OS からの I/O の流れは**図 2** のようになる．ゲスト OS のプロセスが発行した I/O はゲスト OS 内の I/O サブシステムによって処理される．I/O サブシステムがフロントエンドドライバに I/O リクエストを送ると，そのリクエストはドメイン 0 の OS 内のバックエンドドライバに渡される．バックエンドドライバは実ドライバを使うことでデバイスにアクセスする．I/O を発行したプロセスが I/O の結果を必要とする場合や同期的に I/O を発行する場合，プロセスは I/O 待ちでブロックし，I/O が完了した時に再開する．

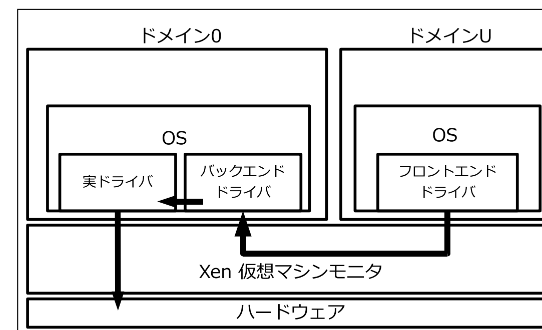

図 2 Xen での I/O の流れ

3.1 I/O 待ちプロセスを制御する難しさ

I/O 待ちでブロックしているプロセスを I/O 完了後も停止させ続けるために，CPU スケジューラのウェイトキューからプロセスを取り除く方法が考えられる．CPU スケジューラはプロセスが CPU を待つランキューとともに，プロセスが I/O の完了を待つウェイトキューを持つ．ウェイトキューからプロセスを取り除けば，I/O が完了した時にプロセスが実行を再開するのを防ぐことができる．しかし，Linux ではウェイトキューはランキューのように一箇所に作られるのではなく，プロセスが I/O 待ちを行う関数で個々に作られている．これらの関数ではウェイトキューに実行中のプロセスを登録してから I/O 待ちを行う．そのため，プロセスをウェイトキューから取り除くには無数にあるウェイトキューをすべて調べなければならない．また，I/O が完了した時にはその I/O 待ちを行っていたプロセスが起こされるが，ウェイトキューからプロセスを取り除いてしまうと，そのプロセスを起こすことができなくなってしまう．

I/O リクエストに着目すると，I/O スケジューラのキューからリクエストを取り除く方法が考えられる．Linux では，ディスクアクセス等を効率よく行えるようにするために，I/O リクエストを適切に並べ換える I/O スケジューラが使われている．I/O リクエストをキューから取り除けばその I/O は発行されないため，I/O の完了を待っているプロセスを間接的

ⓒ 2009 Information Processing Society of Japan

Vol.2009-ARC-183 No.33
Vol.2009-OS-111 No.33
2009/4/24

に停止させることができる．また，取り除いたI/Oリクエストをキューに戻せば，I/Oが完了した後でそのプロセスを再開させることができる．しかし，I/Oリクエストがこのキューで待っている時間はI/O処理を行っている時間と比較して非常に短いため，仮想マシンモニタが定期的にキューをチェックすることによってI/Oリクエストを取り除ける確率は低い．

その上，I/Oリクエストをプロセスに対応づけるのも難しい．あるプロセスを停止させるには，そのプロセスが発行したI/Oリクエストをキューから取り除く必要がある．そのためには，I/Oリクエストからそのリクエストを発行したプロセスを見つけなければならないが，そのような情報は一般的にはI/Oリクエストには含まれていない．I/Oスケジューラとして，Linuxのcfqスケジューラやanticipatoryスケジューラを選択した場合には，プロセス毎にI/Oの統計情報を管理しているため，I/Oリクエストからプロセスを見つけることが可能である．しかし，特定のI/Oスケジューラを使わなければならないことはシステムの汎用性を低下させ，ゲストOSの性能向上を阻害する可能性もあるため望ましくない．また，I/Oスケジューラは性能向上のために複数のI/Oリクエストを1つのリクエストにまとめる場合がある．異なるプロセスが発行したI/Oリクエストを1つにまとめられた場合にはI/Oリクエストを1つのプロセスに対応づけることができない．

I/Oスケジューラでの制御に似た方法として，ドメイン0のバックエンドドライバに渡されたI/Oリクエストの処理を停止させる方法も考えられる．カーネルの変更を避けたいゲストOS内で制御する場合は仮想マシンモニタから間接的に行わなければならないが，ドメイン0のOSでの制御はカーネルを直接書き換えられるため，はるかに容易に実装することができる．バックエンドドライバでI/Oリクエストの処理を行わないようにすれば，そのリクエストを発行したプロセスを間接的に停止させることができる．しかし，I/Oスケジューラのキューからリクエストを取り除く場合と同様に，I/Oリクエストからそのリクエストを発行したプロセスを見つけるのが困難な場合がある．さらに，ドメイン0に渡されたI/Oリクエストとゲスト OS内でのI/Oリクエストの関連づけも行わなければならない．

3.2 提案手法

そこで，我々は仮想マシンモニタからゲストOS内のI/O待ちプロセスの状態を強制的に変更することで，I/Oが完了した後もプロセスを停止させ続ける手法を提案する．Linuxではプロセスの状態は`task_struct`構造体の`state`メンバで管理されている．プロセスがI/O待ちしている間，プロセスの状態は`TASK_UNINTERRUPTIBLE`か`TASK_INTERRUPTIBLE`のどちらかになっている．提案手法では仮想マシンモニタからこのプロセスの状態を`TASK_STOPPED`に変更することにより，プロセスを停止状態にする．この状態はプロセスを一時停止させるためのシグナルである`SIGSTOP`シグナルをプロセスに送った時と同じ状態である．I/Oが完了してこのプロセスが起こされる際にプロセスの状態が`TASK_STOPPED`になっていると，CPUスケジューラはそのプロセスをランキューに入れずに停止状態のままにする．これにより，プロセスは停止し続けることになる．他方，停止状態のプロセスを再開させるには，仮想マシンモニタからプロセスの状態を`TASK_RUNNING`に変更し，ランキューに挿入する．

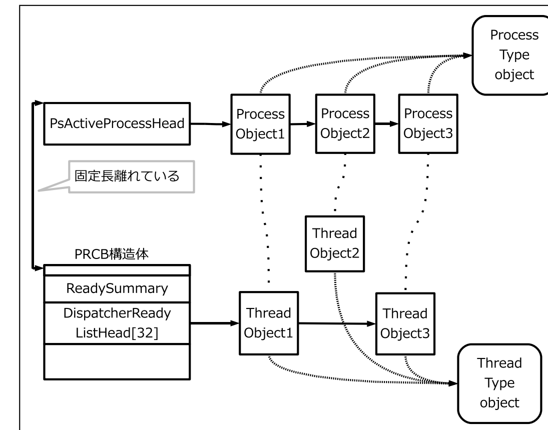

図3 Windowsカーネルの内部

4. Windowsゲスト OSへの対応

4.1 型情報の取得

WinDbgカーネルデバッガ[3]を使うことで，Windowsカーネル内の構造体について型情報を取得することができる．ただし，Linuxカーネルとは違い，カーネル内で使われているすべての構造体の型情報を取得できるわけではない．Windowsカーネルに対してWinDbgを動かすには，カーネルをデバッグモードで起動する必要がある．そのため，事前にWinDbgを使って必要な構造体の型情報を取得しておく．Linuxの場合には，デバッグオプションつきでコンパイルしたカーネルを用意しておけば，GDBを使ってそのバイナリを解析することで型情報を取得することができた．

4.2 ランキューの位置の特定

ランキューを操作するためにはランキューの構造体が置かれているアドレスを見つける必

© 2009 Information Processing Society of Japan

要があるが，Windows においても Linux と同様に起動するたびにランキューの位置が変わるため，事前に知ることはできない．そのため，実行時にランキューの位置を特定する．Windows ではランキューは PRCB と呼ばれる構造体に含まれている．この PRCB はグローバル変数の PsActiveProcessHead から固定長離れた位置に存在する．Linux ではこのようなグローバル変数のアドレスはシンボルテーブルから容易に取得することができる．しかし，Windows の場合はグローバル変数のアドレスはカーネルのロード時まで決まらないため，カーネルをデバッグモードで動かして WinDbg を使わなければ，グローバル変数のアドレスを取得するのは難しい．

そこで，PsActiveProcessHead が全プロセスからなる循環リストの起点を表わすグローバル変数であることに着目し，Monarch Scheduler はまずプロセスを見つける．プロセスを見つけることができれば，循環リストをたどることで PsActiveProcessHead を見つけることができる．ただし，ドメイン U のメモリ中からプロセスを直接見つけるのは難しいため，プロセスを表すビット列をメモリ中から探す．**図 3** のように，Windows カーネル内ではプロセスやスレッドなどはオブジェクトとして表現されている．各オブジェクトは型オブジェクトを指すヘッダを持っている[4]．この型オブジェクトを指すヘッダの値が型ごとに特徴的な値であることを利用し，プロセスオブジェクトの候補を見つけることができる[5]．

プロセスに特徴的な値が見つかったとしても必ずしもプロセスオブジェクトであるとは限らないが，以下の知識を用いることで，ほぼ確実にプロセスオブジェクトを見つけることができる．

- ほとんどの場合，実行ファイル名はアスキー文字である
- Windows のプロセス ID は 4 の倍数である[6]
  プロセスオブジェクトのプロセス ID を表わすフィールドの下位 2 ビットが 0 である．
- プロセスオブジェクトは 1 つの循環リストにつながっている
  すべてのプロセスオブジェクトは ActiveProcessLinks というメンバを使って 1 つの循環リストにつながっているため，プロセスらしきオブジェクトから ActiveProcessLinks フィールドをたどり，最初のオブジェクトまで戻ることができれば，それらのオブジェクトはプロセスオブジェクトである可能性が高い．

プロセスオブジェクトの循環リストの中から PsActiveProcessHead を見つけ出すには，このグローバル変数のアドレスがプロセスオブジェクトとは異なるという知識を利用する．x86_64 アーキテクチャの場合，プロセスオブジェクトのアドレスは上位 32 ビットが 0xfffffa80 であるのに対して，PsActiveProcessHead のアドレスの上位 32 ビットは 0xfffff800 である．OS の起動中，PsActiveProcessHead のアドレスは不変であるため，一旦，このアドレスを見つければ簡単にすべてのプロセスを見つけることができる．同様に，この変数から固定長離れた位置にある PRCB のアドレスも不変である．

4.3 ランキュー操作

Linux の場合と同様に，Windows でもランキューからスレッドを取り除いたり，再度挿入したりすることでスレッドの実行を制御する．Linux との違いは，Linux ではプロセスが制御の対象だったのに対して，Windows ではスレッドが制御対象になる点である．Windows ではプロセスを表す EPROCESS 構造体がスレッドを表す ETHREAD 構造体のリストを持っており，ETHREAD がランキューのリストにつながれている．

Windows の場合，ランキューのリスト操作を行うだけではなく，特定の優先度のスレッドが存在するかどうかを表す ReadySummary と呼ばれるビットマップも更新する必要がある．ランキューからある優先度のスレッドを 1 つ取り除いた時，他に同じ優先度のスレッドが存在しなければ ReadySummary 内のその優先度を表すビットを 0 にする．逆に，ランキューにある優先度のスレッドを挿入した時，ReadySummary 内のその優先度を表すビットを 1 にする．ReadySummary においてどのビットがどの優先度に対応するかは，ReactOS[7] のソースコードを参考にした．

4.4 一貫性を保ったランキュー操作

Linux の場合と同様に，仮想マシンモニタからゲスト OS のランキューを操作するには，ゲスト OS がランキューを操作していないことを保証しなければならない．Linux の場合は，ソースコードを参照することによって，ランキューを操作する前にはどのスピンロックを獲得していることが分かった．そのため，ゲスト OS がそのスピンロックを獲得しているかどうかをチェックすることで，ランキューの操作中かどうかを判別することができた．しかし，Windows の場合にはソースコードを参照できないため，スピンロックのためのデータ構造がカーネル内のどこに置かれているかが不明である．

そこで，Monarch Scheduler では割り込み要求レベル（IRQL）を調べることで，Windows ゲスト OS がスケジューリング中かどうかを判定する．IRQL の値は PRCB 構造体のフィールドから読むことができる．Windows ではスケジューリングを行う際には IRQL が SYNCH_LEVEL になっている．そのため，IRQL の値が SYNCH_LEVEL 未満ならスケジューリング中でないことを保証できる．IRQL の値が SYNCH_LEVEL 以上の場合，必ずしもスケジューリング中とは限らないが，安全のためにスケジューリング中とみなす．このことから，スケジューラが獲得しているスピンロックをピンポイントにチェックできる Linux と比べると，

© 2009 Information Processing Society of Japan

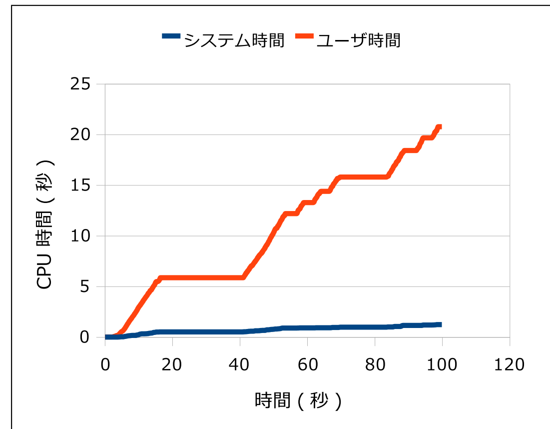

図 4 iozone の CPU 時間の変化

ランキューを操作できない期間が長くなる可能性がある．

4.5 スレッド優先度の変更

仮想マシンモニタから ETHREAD 構造体の BasePriority フィールドを書き換えることで OS 内でのスレッドの優先度を変更する．スレッド優先度は 0 から 31 までの値を取り，値が大きくなるほど優先度が高いことを表す．

5. 実　　験

実験には，Core2Duo E6600 2.4GHz，メモリ 6Gbyte のマシンを用い，Xen 3.3.0，ドメイン 0 に Linux 2.6.18.8，ドメイン U には Linux 2.6.16.33 と Windows Vista Service Pack 1 を用いた．アーキテクチャは，すべて x86_64 である．メモリ割り当ては，特に指定しない限り，ドメイン 0 に 2Gbyte，ドメイン U に 1Gbyte である．

5.1 I/O バウンドプロセス制御の挙動

Monarch Scheduler が I/O バウンドプロセスを制御できていることをを確かめるために，ドメイン U 内で Linux ゲスト OS を動かし，I/O バウンドプロセスとして iozone[8] を実行した．Monarch Scheduler は実験開始から 16.4 秒のところで目的のプロセスを停止させようとし，41.0 秒のところで再開させようとした．さらに，70.0 秒のところでもプロセスを停止させようとし，83.6 秒のところで再開させようとした．iozone が実行中か停止しているかは，/proc/PID/stat から取得できる CPU 時間から調べた．

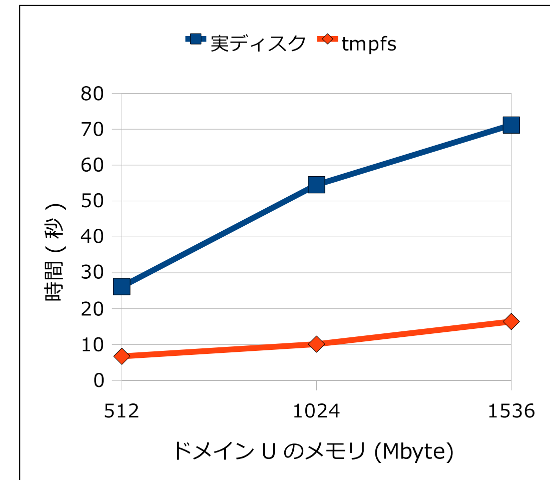

図 5 Windows のランキューの仮想アドレスを取得するのにかかる時間

図 4 が実験結果である．Monarch Scheduler が iozone を停止させている間は，iozone の CPU 時間の増加が見られなかった．このことから，iozone の実行を停止させることができたと考えられる．

5.2 Windows のランキューの位置特定にかかる時間

Windows ゲスト OS のランキューの位置を特定するのにかかる時間を測定した．現在の実装では，Xen が提供するスナップショット機能を用いて Windows が動いているドメイン U のメモリをファイルに保存し，そのメモリイメージを使ってランキューの位置を特定している．そのため，ランキューの位置特定にかかる時間は，ドメイン U のメモリをファイルに書き出す時間と，そのファイルからランキューのアドレスを見つける時間の合計になる．ドメイン U のメモリを書き出す場所として，実ディスク上とメモリ上にファイルシステムを構築する tmpfs 上の 2 通りについて調べた．また，ドメイン U に割り当てるメモリのサイズを変えて調べた．

図 5 が実験結果である．ランキューのアドレスを特定する時間は，ドメイン U のメモリサイズにほぼ比例することがわかる．また，ドメイン U のメモリを実ディスクに書き出した場合のほうが遅いことから，ディスク I/O がボトルネックとなっていることがわかる．ド

© 2009 Information Processing Society of Japan

Vol.2009-ARC-183 No.33
Vol.2009-OS-111 No.33
2009/4/24

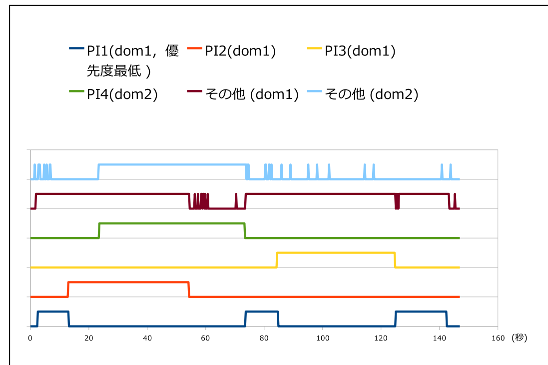

図 6　特定のプロセスの優先度を最低にするスケジューリングの挙動

表 1　Windows スレッドの優先度の変更
Table 1　Changing priority of threads of Windows.

内容	PI 単独	優先度同じ		PI1 を優先	
		PI1	PI2	PI1	PI2
計算時間 (秒)	63.25	120.67	120.42	60.76	119.89

メイン U のメモリをドメイン 0 にマップして調べる方法に変更することで，常に tmpfs と使った場合と同等の時間でランキューの位置を特定することができると考えられる．

5.3　Windows におけるプロセススケジューリングの挙動

Monarch Scheduler を使って 2 つのドメイン U の中で動かした円周率を計算するプロセス[9]4 つをスケジューリングする際のプロセスの挙動を詳しく調べた．ドメイン U1 ではプロセス PI1，PI2，PI3 を動かし，ドメイン U2 ではプロセス PI4 を動かした．そして，PI1 の優先度を最低にして，円周率を計算する他のプロセスが動いていない時だけ動くというスケジューリングポリシを適用した．

それぞれのプロセスの挙動を**図 6** に示す．グラフの各線が下のときはそのプロセスが止まっていることを表し，上のときは動いていることを表す．このグラフから，優先度最低のプロセスはほとんどの場合，すべてのドメインで他に円周率を計算するプロセスが動いていないときのみ動いていることが分かる．

5.4　Windows でのスレッド優先度の変更

Monarch Scheduler が Windows ゲスト OS 上のスレッドの優先度を変更できることを確認する実験を行った．単一のドメイン上 U で円周率を計算するプロセス PI1 と PI2 を同時に動かし，PI1 を優先するというポリシを適用した場合と，同じ優先度にするというポリシを適用した場合を比較した．優先するスレッドの優先度は 24（Real-time）にし，それ以外はデフォルトである 8（Normal）にした．

実験結果は**表 1** のようになった．同じ優先度にするというポリシを適用した場合，PI1 と PI2 を単独で動かした場合のそれぞれ約 2 倍の時間がかかっている．これは優先度が同じスレッドが同時に CPU を使おうとしたためと考えられる．PI1 を優先するというポリシを適用した場合には，PI1 の実行は単独で動かした場合と同等の時間で終了している．これは PI1 を優先できたことを示している．

6. 関 連 研 究

Virtual Machine Introspection[10)11)] は，仮想マシンモニタからゲスト OS の状態を調べる技術である．実装対象のシステムは異なるが，仮想マシンモニタからゲスト OS のメモリを読み，型情報を使ってカーネルの状態を取得するという手法は Monarch Scheduler と同じである．Livewire[10)] は，仮想マシンモニタを通して監視対象のゲスト OS の状態を調べて侵入されたかを判断し，侵入検知システムを監視対象の仮想マシンの外で実行する．IntroVirt[11)] では，さらにゲスト OS のコードを実行することもできる．それに対して，Monarch Scheduler ではゲスト OS の状態を操作してスケジューリングを変更する．

VMwatcher[12)] では，ゲスト OS の中から取得した情報と，仮想マシンモニタを使ってゲスト OS の外から取得した情報を比較することで，マルウェアの存在を検知する．ゲスト OS として Linux だけでなく，Windows にも対応している．Windows カーネル内部のプロセスオブジェクトを推定する点は Monarch Scheduler と同じであるが，Monarch Scheduler ではプロセスオブジェクトからさらにランキューの位置を特定している．また，ゲスト OS を観察するだけでなく，変更している点も異なる．

Lares[13)] はゲスト OS にフックを挿入し，そこから別の仮想マシンで動作させたセキュリティアプリケーションを安全に呼び出せるシステムである．このシステムでは，XenAccess[14)] と呼ばれるライブラリを使ってゲスト OS の内部を調べており，Windows にも対応している．しかし，ゲスト OS の内部状態を変更する API は提供されていない．

タスクを意識した仮想マシンスケジューリング[15)] では，I/O バウンドプロセスを意識して仮想マシンのスケジューリングを行う．Antfarm[16)] の手法やグレーボックス知識を用いて，仮想マシンモニタから仮想マシン上の I/O バウンドプロセスを推定している．推定し

ⓒ 2009 Information Processing Society of Japan

たI/Oバウンドプロセスの情報を基に，パケットが到着した時にそのパケットを受け取るべきプロセスがある仮想マシンを即座にスケジュールすることができる．Monarch Scheduler では仮想マシンを意識したプロセススケジューリングを行っている点が異なる．

ゲストを意識した優先度に基づくスケジューリング[17] はゲスト OS 上で動いているプロセスの優先度を仮想マシンのスケジューリングに反映している．このシステムではゲスト OS を変更して最も高優先度で動いているプロセスの優先度を仮想マシンモニタに通知させ，それを仮想マシンの優先度としている．この際に，実行可能状態のプロセスだけでなく，I/O 待ちしているプロセスも考慮に入れている．Monarch Scheduler ではゲスト OS を変更することなく，プロセスの優先度情報を利用したスケジューリングを行うことが可能である．

7. まとめ

本論文では，仮想マシンモニタによるプロセススケジューリングを実現する Monarch Scheduler において，I/O バウンドプロセスを制御し，Windows ゲスト OS に対応するための手法について述べた．定期的にランキューを操作するこれまでの手法ではランキューで CPU を待つことが少ない I/O バウンドプロセスを制御するのは難しかったため，プロセスの状態を強制的に書き換える手法を提案した．一方，Linux と同様の方法では Windows カーネル中のランキューの位置を特定できなかったため，様々な知識からプロセスの位置を推定し，そこからランキューを特定した．実験により，Linux 上の I/O バウンドプロセスと Windows 上のプロセスをうまく制御できていることを示した．今後の課題は，Windows 上の I/O バウンドプロセスも制御できるようにし，Monarch Scheduler を現実的なアプリケーションに適用することである．

参考文献

1) 田所秀和，光来健一，千葉　滋：仮想マシン間にまたがるプロセススケジューリング，情報処理学会論文誌:コンピューティングシステム（ACS）， Vol.1, No.2, pp.124–135 (2008).
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. Symp. Operating Systems Principles*, pp.164–177 (2003).
3) Microsoft Corporation.: Debugging Tools for Windows, `http://www.microsoft.com/whdc/devtools/debugging/default.mspx`.
4) Russinovich, M. and Solomon, D.: Microsoft Windows Internals, Fourth Edition: Windows 2000, Windows XP, and Windows Server 2003, `http://www.microsoft.com/learning/en/us/Books/6710.aspx`.
5) bugcheck: GREPEXEC: Grepping Executive Objects from Pool Memory, `http://uninformed.org/?v=4&a=2&t=pdf`.
6) oldnewthing: Why are process and thread IDs multiples of four?, `http://blogs.msdn.com/oldnewthing/archive/2008/02/28/7925962.aspx`.
7) ReactOS Foundation: ReactOS, `http://www.reactos.org/`.
8) iozone.org: IOzone Filesystem Benchmark, `http://www.iozone.org/`.
9) Free Software Foundation, Inc.: Computing Billions of *PI* Digits Using GMP, `http://gmplib.org/pi-with-gmp.html`.
10) Garfinkel, T. and Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection, *Proc. Network and Distributed Systems Security Symp.*, pp.191–206 (2003).
11) Joshi, A., King, S., Dunlap, G. and Chen, P.: Detecting Past and Present Intrusions through Vulnerability-specific Predicates, *Proc. Symp. Operating Systems Principles*, pp.91–104 (2005).
12) Jiang, X., Wang, X. and Xu, D.: Stealthy malware detection through vmm-based ”out-of-the-box” semantic view reconstruction, *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pp.128–138 (2007).
13) Payne, B.D., Carbone, M., Sharif, M. and Lee, W.: Lares: An Architecture for Secure Active Monitoring Using Virtualization, *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pp.233–247 (2008).
14) Payne, B.D., Carbone, M. and Lee, W.: Secure and Flexible Monitoring of Virtual Machines, *Computer Security Applications Conference, Annual*, pp.385–397 (2007).
15) Kim, H., Lim, H., Jeong, J., Jo, H. and Lee, J.: Task-aware virtual machine scheduling for I/O performance., *Proc. Int'l Conf. Virtual Execution Environments*, pp. 101–110 (2009).
16) Jones, S., Arpaci-Dusseau, A. and Arpaci-Dusseau, R.: Antfarm: Tracking Processes in a Virtual Machine Environment, *Proc. USENIX 2006 Annual Technical Conf.*, pp.1–14 (2006).
17) Kim, D., Kim, H., Jeon, M., Seo, E. and Lee, J.: Guest-Aware Priority-Based Virtual Machine Scheduling for Highly Consolidated Server, *Proceedings of the 14th international Euro-Par conference on Parallel Processing*, pp.285–294 (2008).

© 2009 Information Processing Society of Japan