情報処理学会論文誌　コンピューティングシステム　Vol. 4　No. 3　100–114 (May 2011)

実用性を考慮した仮想マシン間プロセススケジューラ

田所秀和[†1]　光来健一[†2,†3]　千葉滋[†1]

我々は仮想マシン（VM）をまたがって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 は VM の外側からゲスト OS のランキューを操作することで，プロセススケジューリングを調整する．しかし，Monarch Scheduler にはスケジューリングの精度や汎用性，セキュリティなどの実用上の問題があった．本稿では従来の Monarch Scheduler を拡張した *Monarch Scheduler 2.0* を提案する．Monarch Scheduler 2.0 は仮想マシンモニタ（VMM）内で動作し，より良い精度でスケジューリングを行うことができる．ゲスト OS として Windows にも対応し，ゲスト OS に依存せずにスケジューリングポリシを記述するための高レベル API を提供する．また，VM 間の性能分離を活かしたハイブリッドスケジューリングを行うことで，VM をまたがったサービス妨害攻撃の影響を緩和する．

A Practical Process Scheduler across Virtual Machines

Hidekazu Tadokoro,[†1] Kenichi Kourai[†2]
and Shigeru Chiba[†1]

We have been proposed Monarch scheduler, which schedules processes across virtual machines (VMs). Monarch scheduler directly manipulates the run queues in guest operating systems running on VMs to adjust their process scheduling. However, it had several practical problems of accuracy, generality, and security. This paper proposes *Monarch scheduler 2.0*, which is extended from the previous Monarch scheduler. Monarch scheduler 2.0 runs inside the virtual machine monitor (VMM), and achieves accurate process scheduling. It also supports Windows as guest OSes and provides a high-level API to schedule processes independently from guest OSes. Its hybrid scheduling mitigates DoS attacks by leveraging performance isolation among VMs.

1. はじめに

仮想マシン（VM）を用いたサーバ統合において，管理者が意図したようにプロセスをスケジューリングするのは難しい．各 VM のゲスト OS のプロセススケジューラは他の VM のプロセスを認識しないため，ある VM で動くプロセスが他の VM で動いている優先すべきプロセスの実行を阻害することがある．たとえば，アンチウィルスはしばしば低い優先度で実行される[1] が，同じ VM 上で優先度の高いプロセスが動いていない場合でも，他の VM 上では動いていることがある．また，サーチエンジンのファイルインデキシングはしばしばアイドル時に実行される[2),3)] が，サーチエンジンが動く VM がアイドルだとしても他の VM ではプロセスが動いているかもしれない．

このような問題を解決するために，我々は VM 間にまたがってプロセススケジューリングを行う *Monarch Scheduler* を提案してきた[4]．Monarch Scheduler はすべての VM 上のプロセスを監視し，スケジューリングポリシに基づいて，各ゲスト OS のランキューを操作することでプロセススケジューリングを調整する．Monarch Scheduler は VM の外側からゲスト OS のメモリを直接アクセスするため，ゲスト OS を変更する必要がない．このようにシステム全体でプロセススケジューリングを行うことで，すべての VM 上のプロセスを考慮することができるようになる．

しかし，従来の Monarch Scheduler には以下に示す 3 つの実用上の問題があった．1 つ目は，スケジューリングの精度が低いことである．VM のメモリにアクセスするオーバヘッドのためにスケジューリング間隔をあまり短くすることができず，ゲスト OS から取得するプロセスの実行時間が不正確な場合があった．また，ランキュー上のプロセスしか制御することができず，I/O 待ちプロセスや実行中のプロセスは制御できなかった．2 つ目の問題は，Monarch Scheduler が Linux ゲスト OS のみに対応していたため，スケジューリングポリシがゲスト OS に強く依存しており，汎用性がないことである．3 つ目の問題は，システム全体でプロセススケジューリングを行うと，攻撃者がある VM でプロセスを動かすだけでサービス妨害（DoS）攻撃を行える可能性があることである．

†1 東京工業大学
Tokyo Institute of Technology
†2 九州工業大学
Kyushu Institute of Technology
†3 独立行政法人科学技術振興機構，CREST
Japan Science and Technology Agency, CREST

© 2011 Information Processing Society of Japan

本稿ではこれらの実用上の問題を解決した *Monarch Scheduler 2.0* を提案する．Monarch Scheduler 2.0 はスケジューラを仮想マシンモニタ（VMM）内に実装することで VM のメモリにアクセスするオーバヘッドを抑え，プロセスの正確な実行時間を測定することができる．また，プロセスの情報を書き換えることで，I/O 待ちプロセスや実行中のプロセスも制御する．ゲスト OS として Windows にも対応し，どのゲスト OS に対しても同一のスケジューリングポリシを記述できるように高レベル API を提供する．さらに，ハイブリッドスケジューリングによって VM 間プロセススケジューリングと VM による性能分離を両立させることで，VM をまたがる DoS 攻撃の影響を緩和する．

以下，2 章で従来の Monarch Scheduler の実用上の問題について述べる．3 章でこの問題を解決する Monarch Scheduler 2.0 について述べ，4 章でその実装の詳細について述べる．5 章で Monarch Scheduler 2.0 に対して行った実験について述べる．6 章で関連研究に触れ，7 章で本稿をまとめる．

2. 従来の Monarch Scheduler の問題

2.1 従来の Monarch Scheduler

これまでに提案してきた Monarch Scheduler [4] は，**図 1** のように Xen [5] のドメイン 0 と呼ばれる特権 VM のプロセスとして動作し，ドメイン U と呼ばれる一般の VM 上のプロセスの実行を制御する．Monarch Scheduler はドメイン U のゲスト OS 内のプロセス構造体に記録された実行時間を参照することでプロセスの実行を監視する．その情報を基に，ゲスト OS 内のスケジューラのランキューからプロセスを取り除いたり挿入したりすることでプロセスの実行を調整する．このような制御をすべてのドメイン U 上のプロセスを対象として行うことで，VM 間にまたがるプロセススケジューリングを実現する．

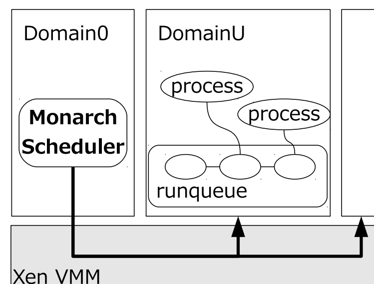


図 1　従来の Monarch Scheduler の構成
Fig. 1　The previous Monarch Scheduler running in domain 0.

ドメイン 0 のスケジューラプロセスは一定時間ごとにすべてのドメイン U を一時停止し，ドメイン U のゲスト OS のメモリを解析する．ゲスト OS のメモリには，ドメイン U のメモリをドメイン 0 のプロセスのアドレス空間にマップすることでアクセスする．この際に，事前にゲスト OS のカーネルのデバッグ情報から取得しておいた型情報を用いる．一貫性を保ってゲスト OS のランキューを操作するために，スケジューラはランキューのロックを調べ，ゲスト OS がランキューを操作中でないことを確認する．このように，Monarch Scheduler はゲスト OS に依存しており，Linux ゲスト OS にのみ対応していた．

Monarch Scheduler は同一組織のサーバを集約して CPU を有効活用しつつ，各プロセスを意図したようにスケジューリングする場合に有効である．一般的には，VM に物理 CPU を排他的に割り当てる方式が用いられていることが多い．排他的な CPU 割当てを行えば，特定の VM の CPU 利用が別の VM に影響を与えることはない．このように VM の性能を分離する方式は，異なるユーザのサーバを集約してユーザごとに使える CPU を保証する場合に有効である．しかし，余った CPU 時間を別の VM が使えないため，CPU を有効活用できない．Monarch Scheduler では，CPU は VM 間で共有し，プロセスへの CPU 時間の割当てを制御するため，CPU を有効に活用することができる．

一方，Xen のクレジットスケジューラでは CPU 割当ての上限や重みを設定することはできるが，CPU を有効活用し，かつ，意図したプロセススケジューリングを行うことは難しい．VM に割り当てる CPU 時間の上限を設定すると，物理 CPU を排他的に割り当てる方式と同様の問題が生じる．VM に CPU 時間を割り当てる比率を重みで設定すれば，ある VM で CPU が余っている場合には別の VM が使うことができる．しかし，重みは VM 単位でしか設定できないため，システム全体で優先度の低いプロセスだけが動いている状況に合わせて VM の重みを小さくすると，その VM の他のプロセスにも十分な CPU 時間が割り当てられなくなる．

2.2 スケジューリングの精度

2.2.1 スケジューリング間隔

従来の Monarch Scheduler はドメイン 0 上で動作していたため，ドメイン U のゲスト OS のメモリにアクセスするオーバヘッドが大きかった．ドメイン 0 からゲスト OS のメモリにアクセスするには，まず，ドメイン U 内のページテーブルを参照して，ゲスト OS の仮想アドレスを疑似物理アドレスに変換する．疑似物理アドレスはドメイン U に割り当てられたメモリに 0 から順番につけられたアドレスである．次に，VMM 内の P2M テーブル

© 2011 Information Processing Society of Japan

を参照して，疑似物理アドレスをマシンアドレスに変換する．マシンアドレスはシステムの物理メモリ全体に 0 から順番につけられたアドレスである．そして，このマシンアドレスを含むメモリページをマップしてアクセスする．この間に，ドメイン U のメモリページをドメイン 0 の Monarch Scheduler のアドレス空間に何度もマップしなければならず，ハイパーコールを使って VMM を呼び出したり，TLB のフラッシュを行ったりする必要もある．

このため，スケジューリング間隔を数百ミリ秒程度以上に設定する必要があった．Monarch Scheduler はすべての VM のすべてのプロセスを調べるため，プロセス数に比例した時間がかかる．大規模なシステムになるとメモリマップのオーバヘッドのためにスケジューリングに時間がかかり，スケジューリング間隔をあまり短くすることができない．その結果，き め細かくプロセスの実行を制御することが難しかった．

また，従来の Monarch Scheduler はドメイン 0 のプロセスとして実装されているため，Monarch Scheduler の実行が VM スケジューリングとドメイン 0 内のプロセススケジューリングに依存していた．Monarch Scheduler が実行されるためには，ドメイン 0 がスケジューリングされて，さらにスケジューラプロセスがスケジューリングされなければならない．負荷の高いシステムでは，スケジューラプロセスの実行が定期的に行われない可能性があった．

2.2.2　プロセスの実行時間の監視

従来の Monarch Scheduler はゲスト OS 内のプロセス構造体に記録されたプロセスの実行時間を基にスケジューリングを行うが，この実行時間は不正確な場合がある．複数の VM が動いている場合，VMM は VM を切り替えながら実行する．VM がスケジュールされていなかった期間に発生したタイマ割込みはその VM がスケジュールされたときに一括で送られる．タイマ割込みの発生時に実行中のプロセスの実行時間を一定時間だけ増やす Linux の O(1) スケジューラなどの場合，VM がスケジュールされていなかった時間もプロセスが動いていたことになってしまう．また，プロセスのコンテキストスイッチの間の時間をプロセスの実行時間として記録する Linux の CFS など場合，VM がスケジュールされていなかった時間もプロセスの実行時間に含まれてしまうことがある．

2.2.3　制御可能なプロセス

従来の Monarch Scheduler はランキュー上で CPU を待っているプロセスのみ停止することが可能であり，I/O の完了を待っているプロセスや実行中のプロセスを停止することはできなかった．多くのアプリケーションは I/O を行うため，I/O 待ちのプロセスを停止できないと停止させたいプロセスをなかなか停止させられず，スケジューリングの精度が低下する．また，CPU を使うプロセスが CPU の数と同じか少ない場合，プロセスはランキューで CPU を待つ必要がなく，つねに実行されている状態になる．そのため，実行中のプロセスを停止することができないと，停止させたいプロセスをいつまでも停止させることができない場合がある．

しかし，I/O 待ちしているプロセスを制御するのは容易ではない．I/O 待ちのプロセスはウェイトキューにつながれているため，ランキューからプロセスを取り除く従来の手法を用いることはできない．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が，Linux ではウェイトキューはプロセスが I/O 待ちを行う箇所で個々に作られている．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．I/O リクエストに着目して 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も考えられる．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に停止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に存在する時間は I/O 処理を行っている時間と比較して非常に短いため，I/O リクエストを取り除ける確率は低い．

また，実行中のプロセスを停止させるのも容易ではない．Linux では実行中のプロセスもランキューにつながれているが，そのプロセスを単純にランキューから取り除くだけでは実行を停止させることができない．プロセスを取り除いたとしても，コンテキトスイッチの際にゲスト OS のプロセススケジューラがそのプロセスをランキューの後ろにつなぎ直してしまうためである．

2.3　汎　用　性

従来の Monarch Scheduler のスケジューリングポリシの記述はゲスト OS の内部構造に強く依存しており，汎用性がなかった．たとえば，特定の名前のプロセスを停止したい場合には，ゲスト OS 内のデータ構造を意識して目的のプロセスを探すコードを書く必要がある．そのため，Monarch Scheduler を Linux 以外のゲスト OS にも対応させたとしても，Linux 用に書いたポリシを Windows 用に使うといったことはできなかった．様々なゲスト OS が動いているシステムの場合，ゲスト OS ごとにポリシを書かなければならず煩雑である．ゲスト OS の違いを意識させずに透過的にポリシを書けるようにする必要がある．

2.4　DoS 攻撃の危険性

VM 間にまたがったプロセススケジューリングを行うと，いずれかの VM に侵入できた攻撃者が新しい種類の DoS 攻撃を行える可能性がある．スケジューリングポリシによっては，攻撃者が特定のプロセスを実行するだけで，他の VM のプロセスに対する DoS 攻撃が可能である．たとえば，システム全体がアイドル状態のときにだけファイルのインデキシン

© 2011 Information Processing Society of Japan

グを行うというポリシの場合，無限ループするプロセスを動かすだけでインデキシングの実行を妨害することができる．Monarch Scheduler は攻撃者が動かしたプロセスを認識して，ファイルのインデキシングを停止させてしまうためである．

従来，VM による性能分離はこのような VM をまたがる DoS 攻撃を防ぐことができていた．ある VM 内に実行を待っているプロセスが存在すれば，その VM には必ず一定の CPU 時間が与えられる．しかし，VM 間プロセススケジューリングを行うと，VM による性能分離がうまく機能しなくなる場合がある．あらかじめこのような攻撃を考慮してスケジューリングポリシを記述することも考えられるが，そのようなポリシは複雑になりエラーの温床になる．

3. Monarch Scheduler 2.0

上記の実用上の問題を解決するために，本稿では *Monarch Scheduler 2.0* を提案する．Monarch Scheduler 2.0 では，従来の Monarch Scheduler のスケジューリング機構を改良することでスケジューリング精度を向上させている．さらに，従来の設計を拡張して，複数のゲスト OS 上のプロセスを統一的に扱えるようにし，DoS 攻撃に対する脆弱性を改善している．

3.1 改良されたスケジューリング機構

Monarch Scheduler 2.0 は**図 2** のように VMM 内で動くプロセススケジューラである．VMM から各ゲスト OS のプロセスの実行を監視し，ゲスト OS の内部データを操作することでプロセススケジューリングを変更する．VMM からドメイン U のメモリにアクセスする際にはオーバヘッドが生じないため，スケジューリング間隔を短くすることができる．これにより，スケジューリングの機会を増やし，精度を向上させる．また，VMM はプロセスの仮想アドレス空間の切替えを監視することができるため，特定のアドレス空間での実行時間を測定することで，プロセスの実行時間を正確に取得することができる．アドレス空間とゲスト OS のプロセスの対応づけはゲスト OS の内部データを解析することで行う．

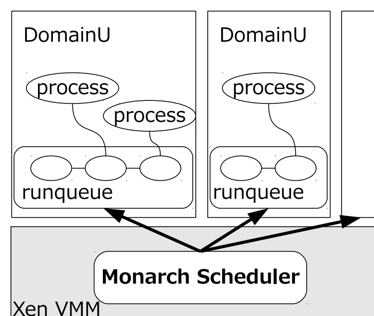


図 2 Monarch Scheduler 2.0 の構成
Fig. 2 The Monarch Scheduler 2.0 running in the VMM.

さらに，ゲスト OS のランキュー操作およびプロセスの状態書き換えによって，様々な状態のプロセスの実行制御を行う．CPU を待っているプロセスに関しては，ランキューからプロセスを取り除くことで停止させる．I/O 待ちプロセスや実行中のプロセスに関しては，プロセスの状態を書き換えることで，ゲスト OS が自発的にプロセスを停止するようにさせる．停止したプロセスを再開する際には，プロセスをランキューに挿入する．このような制御を行うことにより，プロセスの状態にかかわらず，プロセスの実行を停止させることができる．

3.2 ポリシ記述用の高レベル API

Monarch Scheduler 2.0 はゲスト OS の内部構造に依存せずにスケジューリングポリシを記述できるようにするために，高レベル API を提供している．高レベル API を使うことで，どのゲスト OS にも適用できるポリシを書くことができ，様々な OS が動作している環境でもプロセスを統一的に制御することができる．

この API は以下の 2 つのオブジェクトを提供している．

ドメインオブジェクト ドメインオブジェクトは VM の集合を表現し，VM 名を引数に指定して `get_domain_by_name` 関数を呼び出すことで取得することができる．VM 名は正規表現で指定することができ，複数の VM にマッチすればドメインオブジェクトはマッチするすべての VM を表現する．これは複数の VM を透過的に扱うことを可能にする．

タスクオブジェクト タスクオブジェクトはプロセスの集合を表現し，ドメインオブジェクトとプロセス名を引数に指定して `get_task_by_name` 関数を呼び出すことで取得することができる．同名のプロセスが複数あれば，タスクオブジェクトはそのすべてのプロセスを表現する．プロセス名も正規表現で指定可能である．さらに，指定したドメインオブジェクトが複数の VM を表現していれば，タスクオブジェクトは複数の VM にまたがった複数のプロセスを表現する．

これらのオブジェクトが表現する VM やプロセスの集合は，スケジューリングの実行時に動的に決定される．そのため，途中で作成および終了された VM やプロセスもオブジェクトに含めることができる．これらのオブジェクトを用いて，以下の操作を行うことがで

© 2011 Information Processing Society of Japan

```
#define P 10
task_t p_all, p_si;

void init() {
  dom_t d_all = get_domain_by_name(".*");
  p_all = get_task_by_name(d_all, ".*");
  p_si = get_task_by_name(d_all, "SearchIndexer");
  set_period(P);
}
void schedule() {
  time_t t_all = get_time(p_all, P);
  time_t t_si = get_time(p_si, P);
  if (t_all - t_si > 0)
    suspend(p_si);
  else
    resume(p_si);
}
```

図 3 アイドル時スケジューリングのためのポリシ例
Fig. 3 An example policy for idle-time scheduling.

きる．

スケジューリング関数の登録 sched_loop 関数を使って，スケジューリングポリシを実装した関数を Monarch Scheduler に登録する．この関数は引数で指定した関数を定期的に実行する．スケジューリング間隔はクォンタムと呼ばれ，set_quantum 関数を使って設定することができる．デフォルトのクォンタムは 10 ms である．

タスクの監視 set_period 関数を使って，プロセスが使った CPU 時間を記録しておく期間をクォンタム単位で指定する．get_time 関数はタスクオブジェクトとクォンタム単位の期間を引数にとり，指定したタスクオブジェクトが表現しているプロセス群が指定した期間に使った CPU 時間の合計を返す．

タスクの制御 suspend 関数は引数で指定したタスクオブジェクトが表現しているプロセス群を停止させる．resume 関数は引数で指定したタスクオブジェクトが表現しているプロセス群を再開させる．

図 3 はシステム全体がアイドルのときだけインデキシングサービスが動くようにする簡単なアイドル時スケジューリング[6] を実現するポリシである．最初に実行される init 関数で，すべての VM を表現するドメインオブジェクトを作る．次に，すべての VM 内のすべてのプロセスを表現するタスクオブジェクトと，SearchIndexer という名前のプロセスを表現するタスクオブジェクトを作る．パラメータ P でシステム全体がアイドルになってからインデキシングサービスが動き出すまでの時間を指定できる．schedule 関数は sched_loop 関数で Monarch Scheduler に登録され，すべてのプロセスが使った CPU 時間の和とインデキシングサービスが使った CPU 時間を定期的に収集する．インデキシングサービス以外のプロセスが CPU を使っていたら，インデキシングサービスを停止する．

このほかにも，プロセスの実行時間を監視してスケジューリングを行うポリシは記述可能である．たとえば，プロセスに優先度を与える優先度スケジューリングやプロセスに一定の割合で CPU 時間を割り当てるプロポーショナルスケジューリングなどが可能である．これらのスケジューリングでは，get_time 関数で CPU 時間を取得し，目的の CPU 時間になるように suspend 関数および resume 関数を用いて，プロセスの実行を調整すればよい．また，必要に応じて API の拡張が必要となるが，プロセス名だけでなくプロセス構造体内の他の情報を使ったスケジューリングポリシを記述することも可能である．プロセスのユーザ情報を使うことで，ユーザ単位でプロセスをグループ化し，CPU 時間を制限することができる．また，プロセスを親子関係に基づいてグループ化することもできる．一方，それ以外の情報を使ったスケジューリングポリシは記述できない．たとえば，ウェブサーバプロセスを特定のリクエストの場合だけ優先するといったことはできない．これを記述可能にすると API がアプリケーションに依存してしまうからである．

この高レベル API は Monarch Scheduler のゲスト OS 依存の実装を隠蔽することができるが，さらには Monarch Scheduler に限らず，VM 間にまたがるプロセススケジューラ全般で利用可能である．たとえば，ゲスト OS を改変してプロセスの操作を行えるようにする実装であっても同一の API を用いることができる．その点で，提案する高レベル API は Monarch Scheduler に依存したものではなく，非常に汎用性が高い．

3.3 ハイブリッドスケジューリング

Monarch Scheduler 2.0 は，制御モードと自律モードの 2 つのモードを定期的に切り替えるハイブリッドスケジューリングを行う．制御モードでは VM 間プロセススケジューリングを行い，自律モードでは VM 間プロセススケジューリングを行うのをやめ，VMM とゲスト OS の本来のスケジューリングを実行する．2.4 節で述べたように，ある VM に侵入した攻撃者がその VM でプロセスを動かすことで他の VM のプロセスを止めようとしたとしても，そのような DoS 攻撃は自律モードによって緩和される．攻撃対象となったプロセスはスケジューリングポリシによっては制御モードではまったく動かなくなるが，自律モードによって一定の実行時間が保証される．

© 2011 Information Processing Society of Japan

ハイブリッドスケジューリングは，VM をまたがったスケジューリングと VM 間の性能分離を両立させる．制御モードでは VM 間の分離を意識せずに VM にまたがってプロセスの制御を行うことができる．自律モードでは VM によって実現されている性能分離を有効にすることで他の VM からの DoS 攻撃を防ぐ．これら 2 つのモードはトレードオフの関係にあるため，Monarch Scheduler 2.0 では 2 つのモードを実行する時間の比率を管理者が指定できる．制御モードの比率を高くすれば，プロセスをより正確に制御することができるようになる．逆に，自律モードの比率を高くすれば，より厳密に VM による性能の保証を行うことができるようになる．ハイブリッドスケジューラも Monarch Scheduler に依存した手法ではなく，VM 間にまたがるプロセススケジューラ全般に対して DoS 攻撃を緩和するために使うことができる．

4. 実　　装

我々は Monarch Scheduler 2.0 を Xen 3.4.2（x86_64）上に実装した．対応ゲスト OS は Linux 2.6.16.33 と Windows Vista SP1 であり，Xen に追加する必要があったコードの行数は約 6200 行であった．追加コードの内訳は，Linux への対応 1900 行，Windows への対応 1200 行，API の解釈部 900 行，それ以外が 2200 行であった．

4.1 プロセスの状態書き換え

Monarch Scheduler 2.0 は，プロセスの状態が blocked または running である場合，プロセス構造体の中のプロセスの状態を書き換えることで，プロセス自身に自発的に実行を停止させる．blocked 状態は I/O などのイベント待ちをしているプロセスの状態であり，running 状態は現在実行中のプロセスの状態である．一方，状態を書き換えて停止させたプロセスを再開させる際には，プロセスの状態を ready に書き換えてから，プロセスをランキューに挿入する．ready 状態はランキューで CPU を待っているプロセスの状態である．

blocked 状態のプロセスを停止させるために，Monarch Scheduler 2.0 はプロセスの状態を stopped に書き換える．stopped 状態とは SIGSTOP シグナルによってプロセスを停止したのと同じ状態である．通常，I/O 待ちしているプロセスは I/O が完了するとランキューに挿入される．しかし，I/O が完了した時点でプロセスの状態が stopped になっていると，そのプロセスはランキューに挿入されず停止し続ける．

running 状態のプロセスを停止させるために，Monarch Scheduler 2.0 はプロセスの状態を blocked に書き換える．通常，実行中のプロセスはコンテキストスイッチが起こると実行を停止させられ，ランキューに挿入される．しかし，コンテキストスイッチが起こった時点でプロセスの状態が blocked になっていると，そのプロセスはランキューに挿入されず，I/O 待ちと同じ状態になって停止する．

この手法は現在のところ，Linux に対してのみ有効である．Windows では stopped 状態が存在しないため blocked 状態のプロセスを同じ手法で停止させることはできない．

4.2 VMM でのプロセススケジューリング

Monarch Scheduler 2.0 は VMM の中で動作するため，ゲスト OS のメモリに非常に小さなオーバヘッドでアクセスすることができる．2.2.1 項で述べたように，ゲスト OS の特定の仮想アドレスにアクセスするには，ドメイン U のページテーブルを参照し，VMM 内の P2M テーブルを参照してから，ドメイン U のメモリにアクセスする必要がある．VMM はドメイン U のメモリに直接アクセスすることができるため，メモリマップを行うオーバヘッドを削減することができる．ゲスト OS のメモリへのアクセスの高速化によって，スケジューリング間隔を 1 ミリ秒程度まで短くすることも可能になった．

Monarch Scheduler 2.0 は，VMM 内のタイマ割込みによって定期的に呼び出されるため，呼び出されるタイミングが正確である．タイマ割込みによって呼び出されたスケジューラは，スケジューラ自身が使っている CPU 以外のすべての物理 CPU を停止させることで，すべてのドメイン U を一時停止させる．従来の Monarch Scheduler ではハイパーコールを呼び出してドメイン U を一時停止していた．スケジューリングが完了したら，停止させた物理 CPU を再開する．

4.3 VMM でのプロセス実行の監視

ゲスト OS のプロセスの実行時間を正確に記録できるようにするために，Monarch Scheduler 2.0 は VMM 内でプロセスの実行を監視する．プロセスは OS の概念であるため，プロセスに対応する仮想アドレス空間の変化を見ることで CPU 時間を測定する．仮想アドレス空間はページディレクトリテーブルの物理アドレスによって一意に特定できる．ゲスト OS がプロセスをコンテキストスイッチする際には，x86 アーキテクチャでは仮想 CPU の CR3 レジスタにページディレクトリテーブルのアドレスを書き込む．CR3 レジスタの値を書き換える命令は特権命令であるため，VMM が命令の実行をトラップすることができる．Monarch Scheduler 2.0 は CR3 レジスタに書き込まれるアドレスを調べることで，どのプロセスが実行されるかを知ることができる．CR3 レジスタにある値が書き込まれてからその値が変化するまでの時間が，対応するプロセスが CPU を使用した時間となる．

仮想アドレス空間とプロセスを対応づけるために，Monarch Scheduler 2.0 はゲスト OS 内の情報を利用する．Linux ではページディレクトリテーブルのアドレスは task_struct 構

© 2011 Information Processing Society of Japan

造体からたどることができる mm_struct 内に格納されている．Windows では EPROCESS 構造体の中の DirectoryTableBase フィールドに格納されている．Monarch Scheduler 2.0 はスケジューリングを行う際に，ゲスト OS 内のプロセスリストを調べることにより，測定した CPU 時間と実行されたプロセスを結び付ける．

CR3 レジスタを使ったプロセスの実行時間の測定は，スケジューリング間隔より短い時間しか生存しないプロセスについても網羅的に行うことができる．ただし，このようなプロセスはスケジューリング時にはすでにゲスト OS 内に存在しないため，測定した実行時間を考慮することができない．この問題に対処するには，CR3 レジスタに新しい値が書き込まれた時点でゲスト OS 内の対応するプロセスを探すようにする必要がある．Monarch Scheduler 2.0 ではある程度以上動き続けるプロセスをスケジューリング対象にしており，このような非常に短時間しか生存しないプロセスは対象外である．たとえば，Xen 3.4.2 のハイパーバイザ部を make したとき，Monarch Scheduler 2.0 のデフォルトスケジューリング間隔である 10 ms 未満しか生存しないプロセスは 21%あったが，make 全体の実行時間に占める割合は 0.9%であった．このことから，短時間しか生存しないプロセスを考慮しなくても，一般的にはスケジューリングに大きな影響はないと考えられる．

現在の実装では，Intel EPT や AMD NPT などのメモリ仮想化支援機能には対応していない．このような支援機能を利用すると，CR3 の更新が VMM でトラップされないためである．VMM でトラップできるように設定することで対応可能だと考えられるが，オーバヘッドが大きくなることが予想される．

4.4　Windows ゲスト OS への対応

提案したポリシ記述用の高レベル API が複数のゲスト OS に共通で使えることを示すために，Monarch Scheduler 2.0 が Windows ゲスト OS 上のプロセスを操作できるように実装を行った．まず，WinDbg カーネルデバッガ[7] を使うことで，Windows カーネル内の構造体の型情報を取得する．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，Windows をデバッグモードで起動する必要がある．そのため，オフラインで WinDbg を使って必要な構造体の型情報を取得しておく．デバッグモードで Windows を動かすとシリアルポートに大量のデバッグメッセージを出力するため，型情報を取得するときだけデバッグモードで動かす．

4.4.1　ランキューの探索

ランキューを操作するにはランキューが置かれているアドレスを見つける必要があるが，ランキューのアドレスは Windows を起動するたびに変わるため，事前に知ることができない．Windows ではランキューは PRCB 構造体に含まれている．我々の調査によると，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ことが分かっている．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た．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，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 が全プロセスの循環リストの起点を表すグローバル変数であることに着目し，Monarch Scheduler 2.0 はまずプロセスオブジェクトを探す．プロセスオブジェクト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 のメモリの中からプロセスオブジェクトを直接見つけるのは難しいため，プロセスオブジェクトを推測することで見つけ出す．Windows カーネル内ではプロセスなどの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[8]．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[9]．

メモリ中から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ため，以下の知識を用いてプロセスオブジェクトだけを抽出する．

- 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[10]．
- 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 というフィールドを使って 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 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大きく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．Windows の実行中は，PsActiveProcessHead のアドレスは不変であるため，いったん，このアドレスを見つければつね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 のアドレスを知ることができる．

© 2011 Information Processing Society of Japan

4.4.2　ランキュー操作

Windows でも Linux の場合と同様に，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．Linux との違いは，Linux ではプロセスがスケジューリングの単位になっているのに対して，Windows ではスレッドがスケジューリングの単位になっている点である．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．プロセスを停止させる際にはプロセスが持つすべてのスレッドをランキューから取り除き，再開する際にはそれらをランキューに挿入する．

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同時に更新する．ある優先度のスレッドをランキューから取り除いたとき，他に同じ優先度のスレッドが存在しなければ ReadySummary 内のその優先度に対応するビットを 0 にする．逆に，ある優先度のスレッドをランキューに挿入したとき，ReadySummary 内のその優先度に対応するビットを 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については，ReactOS [11] のソースコードを参照した．

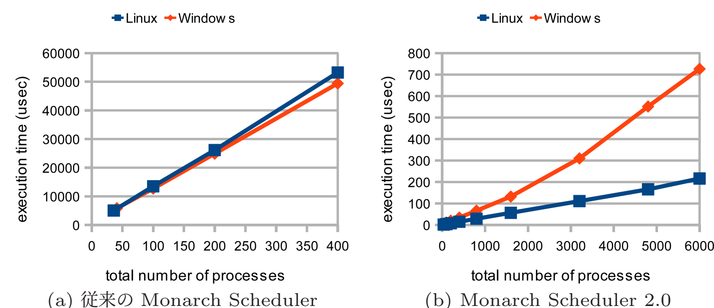
4.4.3　一貫性の保証

一貫性を保ってランキューを操作するために，Monarch Scheduler 2.0 は Linux の場合と同様に，Windows 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．PRCB 構造体のロックが Windows カーネルによって取得されていなければ，Monarch Scheduler 2.0 は安全にランキューを操作することができる．

一方，プロセスオブジェクトの循環リストを参照する際には，リストの次の要素を指すポインタが NULL かどうかをチェックする．Linux の場合にはプロセスリストへのアクセスを保護するために使われているロックを調べていたが，Windows ではそのようなロックを見つけることができなかったためである．NULL チェックの手法は XenAccess [12] でも用いられており，タイミングによってはすべてのプロセスやスレッドをたどれない場合があるが，安全にアクセスを行うことは保証される．

5. 実　　験

Monarch Scheduler 2.0 の実用性を調べるための実験を行った．実験には，Core 2 Duo 2.4 GHz，メモリ 6 GB のマシンを用い，ドメイン U のメモリには Linux の場合は 1 GB，Windows の場合は 2 GB を割り当てた．ドメイン 0 にはメモリサイズを指定せず，メモリが動的に割り当てられるようにした．クライアントマシンとして Core 2 Quad 2.83 GHz，メモリ 8 GB のマシンを用い，2 つのマシンをギガビットスイッチで接続した．



(a) 従来の Monarch Scheduler　(b) Monarch Scheduler 2.0

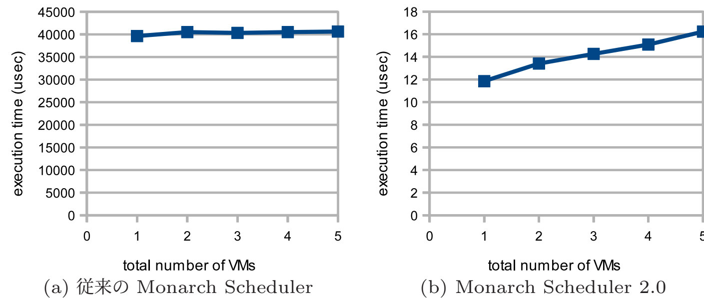
図 4　プロセス数を変化させた場合のプロセスリストをたどる時間
Fig. 4　The time for traversing the processes list for various numbers of processes.

5.1　メモリアクセスのオーバヘッド

従来の Monarch Scheduler と Monarch Scheduler 2.0 がゲスト OS のプロセスリストをたどるのにかかる時間を比較した．Monarch Scheduler は，スケジューリング対象のプロセスを見つけるために，スケジューリングのたびにゲスト OS のプロセスリストをたどる必要がある．まずは，1 つの VM を対象にプロセス数を変化させた場合のプロセスリストをたどる時間を，Linux と Windows ゲスト OS についてそれぞれ測定した．Linux でのデフォルトのプロセス数は 36 であり，Windows では 41 であった．**図 4** (a) はドメイン 0 からプロセスリストをたどる時間であり，図 4 (b) は VMM からたどる時間である．VMM からたどることにより，400 プロセスをたどる時間が Linux で 15 μs になり 3500 倍高速になった．Windows では 34 μs になり 1400 倍高速になった．

さらに，VM の数を 1 から 5 まで変化させて，すべての VM のプロセスリストをたどる時間を Linux ゲスト OS について測定した．VM の数を変化させた場合のオーバヘッドを測定するため，システム全体のプロセス数を 300 に固定した．**図 5** が測定結果である．従来の Monarch Scheduler では，VM の数に依存せずにほぼ一定の時間がかかった．これは，メモリアクセスに時間がかかるため，VM の停止や再開にかかる時間が相対的に小さくなるためだと考えられる．一方，Monarch Scheduler 2.0 では VM の数が増えるとプロセスリストをたどる時間が長くなった．

© 2011 Information Processing Society of Japan



(a) 従来の Monarch Scheduler　(b) Monarch Scheduler 2.0

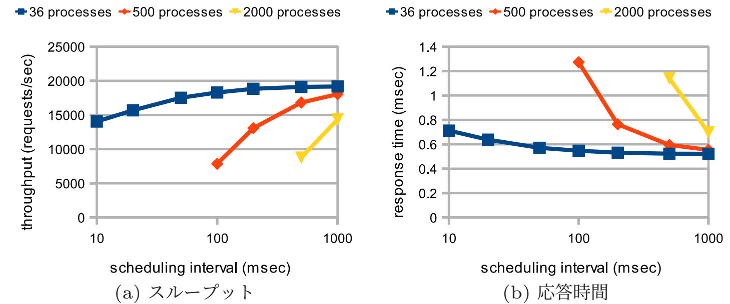
図 5　VM の数を変化させた場合のプロセスリストをたどる時間
Fig. 5　The time for traversing the process list for various numbers of VMs.

5.2　スケジューリング間隔の性能への影響

Monarch Scheduler による性能の劣化を調べるため，lighttpd [13] ウェブサーバのスループットと応答時間を測定した．1 つの VM 上で lighttpd と，プロセス数を調整するためにつねにスリープしているプロセスを必要な数だけ動かした．MonarchScheduler はスケジューリングのたびにプロセスリストをたどり，Linux ゲスト OS のプロセススケジューリングは変更しなかった．性能を測定するために，ベンチマークツールの ApacheBench [14] を使い，10 並列でリクエストを送った．スケジューリング間隔を 0.1 ms から 100 ms まで変化させ，プロセス数を 36，500，2000 と変化させてスループットと応答時間を測定した．このときの性能低下を，オリジナルの Xen 上での lighttpd の性能と比較した．

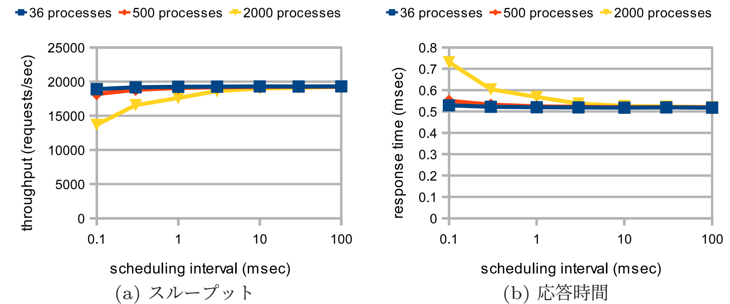
従来の Monarch Scheduler での結果が**図 6** である．スケジューリング間隔が 10 ms，プロセス数が 36 の場合でもスループットが 27%，応答時間が 27%劣化した．プロセス数が 500 の場合では，スケジューリング間隔を 100 ms にしてもスループットと応答時間ともに性能が 50%以上劣化した．これより，プロセス数が増加すると間隔を短くできないことが分かる．

一方，Monarch Scheduler 2.0 での結果は**図 7** である．スケジューリング間隔が 10 ms，プロセス数が 500 の場合には，スループットと応答時間ともに 0.3%以下の劣化に抑えられた．一方，間隔が 0.1 ms，プロセス数が 2000 の場合にはパフォーマンスが大きく劣化した．しかし，間隔が 10 ms ならばプロセス数が 2000 になってもスループットは 1.5%，応答時間は 1.3%の劣化に抑えられた．



(a) スループット　(b) 応答時間

図 6　従来の Monarch Scheduler でのウェブサーバの性能劣化
Fig. 6　The performance degradation of a web server with the previous Monarch scheduler.



(a) スループット　(b) 応答時間

図 7　Monarch Scheduler 2.0 でのウェブサーバの性能劣化
Fig. 7　The performance degradation of a web server with the Monarch scheduler 2.0.

5.3　スケジューリング間隔の精度

VMM でスケジューリングするほうが，正確な間隔でスケジューリングできることを示す実験を行った．10 ms ごとに定期的に動くよう指定したスケジューラが実際にどれくらいの間隔で実行されるかを，ドメイン 0 上のプロセスとして実装した場合と，VMM 内に実装した場合とで比較した．ドメイン 0 に負荷をかけない状態でも，スケジューラプロセスを使った場合にはスケジューリング間隔は 20 ms になった．これはドメイン 0 のプロセススケジューリングの影響だと考えられる．一方，VMM 内で行った場合は正確に 10 ms 間隔でスケジューリングされた．

次に，ドメイン U 上で Bonnie++ [15] を動かし，ドメイン 0 とドメイン U が同じ物理

© 2011 Information Processing Society of Japan

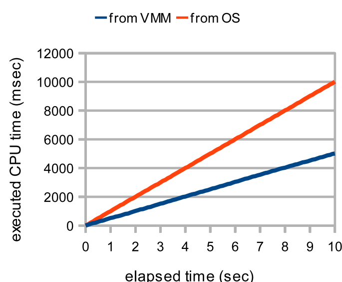
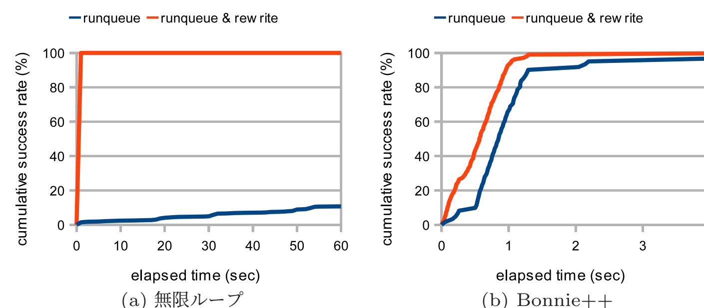


図 8　VMM から測定した CPU 時間とゲスト OS 内の CPU 時間との比較
Fig. 8　The difference between the process times obtained from a guest OS and that tracked by the VMM.



(a) 無限ループ　(b) Bonnie++

図 9　プロセスを停止させるのにかかる時間の分布
Fig. 9　The distribution of times elapsed for stopping a process.

CPU を使うようにして同様の実験を行った．スケジューラプロセスを用いた場合にはスケジューリング間隔が 32 ms に増加したが，VMM 内で行った場合には 10 ms のままであった．これは，負荷のためにドメイン 0 のスケジューラプロセスに十分な CPU 時間が与えられなかったためだと考えられる．VMM 内で実装すれば負荷をかけた状態でも正確な間隔でスケジューリングを行うことができる．

5.4　VMM で測定したプロセス時間の精度

VMM からプロセス時間を測定したほうが正確に測定できることを示す実験を行った．ゲスト OS に Linux 2.6.16.33 を用いて VM1 と VM2 を動かし，それぞれ無限ループするプロセスを 1 つずつ動かした．これらの VM は 1 つの物理 CPU を共有し，同じ優先度に設定した．VM1 で動くプロセスが使った CPU 時間について，VMM で測定した値とゲスト OS が記録した値を比較した．1 つの物理 CPU を同じ優先度の 2 つのプロセスが共有しているため，VM1 上のプロセスが実際に使用した CPU 時間は経過時間の半分になるはずである．

図 8 に経過時間と測定したプロセス時間を示す．Monarch Scheduler 2.0 が行っているように，VMM から測定した場合は正しく経過時間の半分になっていることが分かる．一方，従来の Monarch Scheduler が使っていたゲスト OS 内の情報は経過時間と同じになっており正しくない．これはゲスト OS が物理 CPU を使っていない時間を考慮できていないからである．

5.5　プロセスの状態書き換えの効果

状態書き換えにより，より正確にプロセスを制御できることを示す実験を行った．従来のランキュー操作のみの場合と状態書き換えを併用した場合について，10 ms の間隔でプロセスを停止させる操作を繰り返し，そのプロセスを一定時間以内に止められた割合を調べた．対象としたプロセスは無限ループプロセスと Bonnie++であり，それぞれ単独で動かし実験を行った．無限ループプロセスはつねに CPU を使うため，単独で動かすとつねに running 状態であった．Bonnie++は I/O 待ちの時間が多く，単独で動かすと 53%の時間が blocked 状態であった．

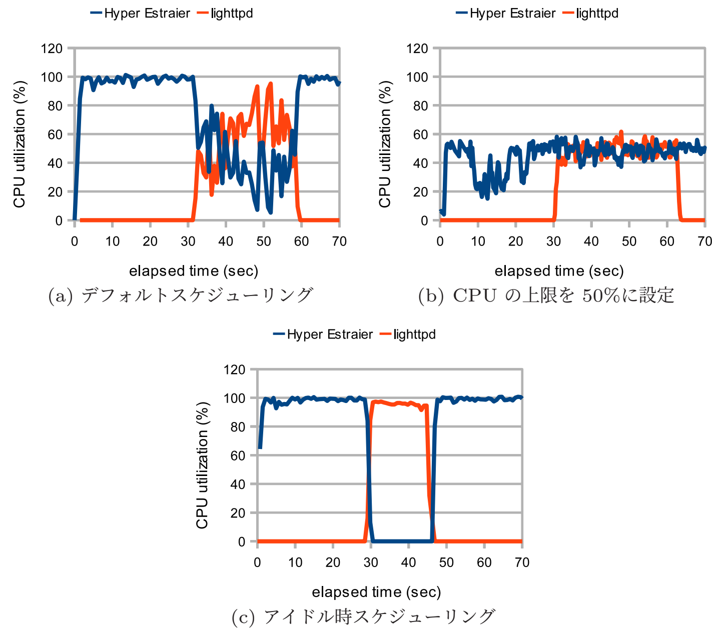
図 9 (a) から，無限ループプロセスの場合，ランキュー操作のみでは 60 秒たっても 11%しか停止させることができなかったが，プロセスの状態書き換えを併用することで正確に停止させられるようになったことが分かる．これはランキュー操作では制御できなかった実行中のプロセスを，状態書き換えによって制御できるようになったためである．図 9 (b) から，Bonnie++の場合，1 秒後までに停止させられた割合がランキュー操作のみでは 66%だったのに対し，状態書き換えを併用することで 93%になったことが分かる．これは I/O 待ち状態のプロセスを制御できていることを示している．

5.6　VM 間プロセススケジューリングの効果

5.6.1　アイドル時スケジューリング

Monarch Scheduler 2.0 を用いて，VM 間にまたがるアイドル時スケジューリングの効果を調べた．VM 1 で lighttpd を動かし，VM 2 で Hyper Estraier [16] のファイルインデキシングを動かした．Hyper Estraier は全文検索エンジンである．Hyper Estraier が lighttpd の性能に影響を与えないように，システム全体がアイドルのときだけインデキシングする図 3 のようなポリシを適用した．

© 2011 Information Processing Society of Japan

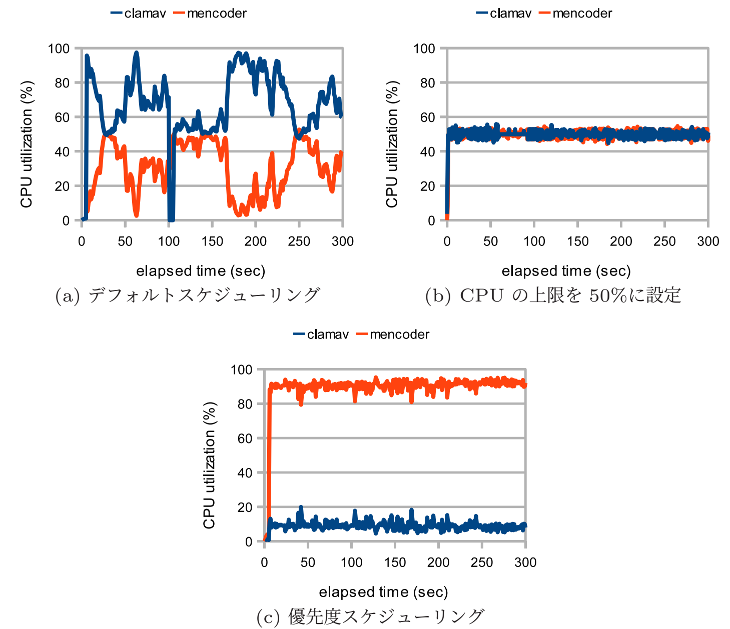


(a) デフォルトスケジューリング　(b) CPU の上限を 50%に設定

(c) アイドル時スケジューリング

図 10　Hyper Estraier を対象としたアイドル時スケジューリング

Fig. 10　System-wide idle-time scheduling for Hyper Estraier.



(a) デフォルトスケジューリング　(b) CPU の上限を 50%に設定

(c) 優先度スケジューリング

図 11　ClamAV を対象とした優先度スケジューリング

Fig. 11　System-wide priority scheduling for ClamAV.

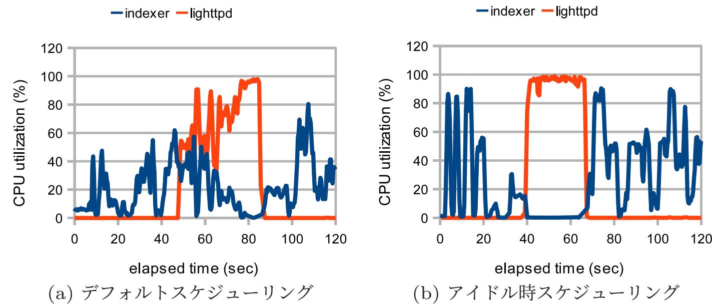
最初に，Monarch Scheduler 2.0 を使わず，Xen のクレジットスケジューラのみを用いた場合の 2 つのプロセスの動きを調べた．**図 10** (a) は 2 つの VM に 1 : 1 の重みを設定した Xen のデフォルトスケジューリングを行った場合での，2 つのプロセスの CPU 使用率の変化である．VM2 でも検索結果を提供するサーバを動かす必要があるため，2 つの VM には同じ重みを与えた．VM 1 で lighttpd が動いている間，VM 2 に他のプロセスが存在しないためファイルインデキシングが動いてしまった．その結果，ファイルインデキシングは lighttpd の性能に大きく影響を与えた．このとき lighttpd のスループットは 24%低下し，応答時間は 32%増加した．図 10 (b) は 2 つの VM に割り当てる CPU の上限をそれぞれ 50%に設定した場合の結果である．このとき lighttpd のスループットは 52%低下し，応答時間は 52%増加した．これより，VM2 がアイドル状態であっても VM1 がその CPU 時間を使えていないことが分かる．

次に，Monarch Scheduler 2.0 を使い，システム全体がアイドルのときだけファイルインデキシングを動かした．図 10 (c) より，VM1 で lighttpd が動くと VM2 のファイルインデキシングがすぐに止まっていることが分かる．このときの lighttpd のスループットは 2.4%低下し，応答時間は 2.5%増加しただけであった．

5.6.2　優先度スケジューリング

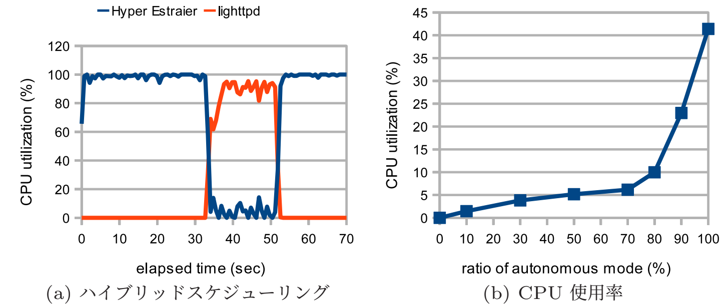
Monarch Scheduler 2.0 による優先度スケジューリングの効果を調べた．VM 1 で MEncoder[17] を 4 つ，VM 2 で ClamAV[18] のウイルススキャンを動かし，MEncoder を優先して動かすポリシを適用した．最初に，Monarch Scheduler 2.0 を使わない場合の 5 つのプロセスの動きを調べた．**図 11** (a) は 2 つの VM の重みを 1 : 1 に設定したときのプロセスの CPU 使用率の変化である．MEncoder は 4 プロセスの CPU 使用率の合計値をプロットした．このとき，ClamAV の平均 CPU 使用率は 67%であり，MEncoder より優先されてし

© 2011 Information Processing Society of Japan



(a) デフォルトスケジューリング　(b) アイドル時スケジューリング

図 12　複数 OS を対象としたアイドル時スケジューリング
Fig. 12　System-wide idle-time scheduling across multiple OSes.



(a) ハイブリッドスケジューリング　(b) CPU 使用率

図 13　アイドル時スケジューリングを用いたハイブリッドスケジューリングの効果
Fig. 13　The effects of hybrid scheduling with idle-time scheduling.

まっている．一方，それぞれの VM の CPU の上限を 50%に設定した場合の結果が図 11 (b) である．VM1 が 50%の CPU 時間しか使えなくなるため，ClamAV が使った CPU 時間は各 MEncoder の 4 倍になった．

次に，Monarch Scheduler 2.0 を使い，MEncoder と ClamAV の使う CPU の割合が 8 : 1 になるように ClamAV の優先度を低くした．図 11 (c) より，ClamAV の CPU 使用率の平均は 12%となり，優先度を低くできていることが分かる．

5.7　複数 OS にまたがるプロセススケジューリング

Windows ゲスト OS が混在した場合でも 1 つのスケジューリングポリシを用いてうまくスケジューリングできることを示すために，Windows と Linux を対象にアイドル時スケジューリングを行った．VM 1 では Linux 上で lighttpd を動かし，VM 2 では Windows 上で SearchIndexer を動かした．SearchIndexer は，ファイルを検索するためのインデックスを作るプロセスである．スケジューリングポリシは 5.6.1 項のものと同じであり，Hyper Estraier のプロセス名だけを SearchIndexer に書き換えた．

Monarch Scheduler 2.0 を使わない場合に，2 つのプロセスの動きを調べたものが**図 12** (a) である．lighttpd と SearchIndexer が同時に動いてしまい，lighttpd のスループットが 25%低下し，応答時間は 34%増加した．次に，Monarch Scheduler 2.0 を使いシステム全体がアイドルのときだけ SearchIndexer を動かした．図 12 (b) より，VM1 で lighttpd が動くと SearchIndexer がすぐに止まり，SearchIndexer を制御できていることが分かる．これより，ゲスト OS が変わっても同一のポリシを使えることが示された．このときの lighttpd のスループットは 4.3%低下し，応答時間は 4.5%増加した．

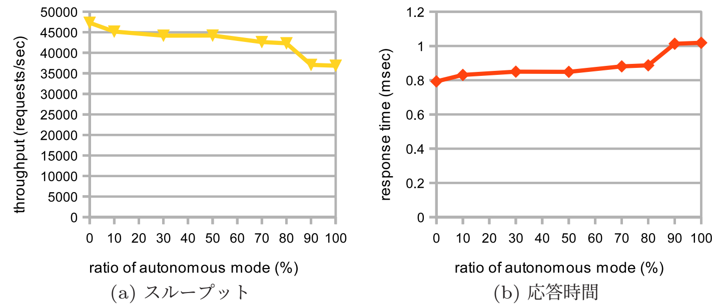
5.8　ハイブリッドスケジューリングの効果

5.6.1 項の実験について，ハイブリッドスケジューリングを有効にして実験を行った．**図 13** (a) は自律モードの割合が 50%のときの 2 つのプロセスの CPU 使用率の変化である．ハイブリッドスケジューリングを行うと，lighttpd が動いている間もファイルインデキシングが完全には止まらなくなり，DoS 攻撃の緩和ができていることが分かる．次に，自律モードの割合を変化させた場合のファイルインデキシングの CPU 使用率を測定した．図 13 (b) より，自律モードの割合を増やすとファイルインデキシングがより長く動くことが分かる．自律モードの割合が 80%を超えると，CPU 使用率が急激に増えている．これは，ゲスト OS の操作の効果が出る前に，制御モードから自律モードに切り替わってしまうためである．

一方，ハイブリッドスケジューリングは lighttpd のみが動く期間でも，lighttpd の性能を劣化させる．性能の劣化は**図 14** のようになった．自律モードの割合が増えると，スループットが低下し応答時間が増加している．自律モードの割合が 50%の場合，スループットは 9.7%低下し，応答時間は 8.8%増加した．

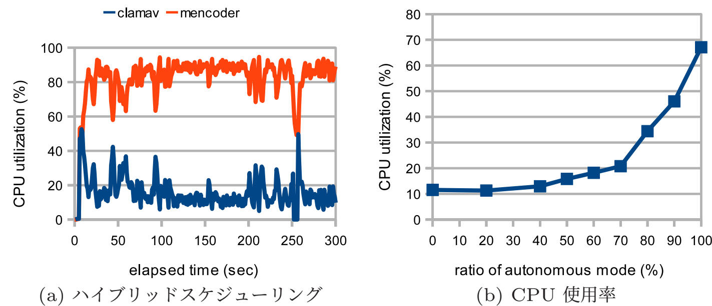
さらに，5.6.2 項の実験についてハイブリッドスケジューリングを有効にして実験を行った．**図 15** (a) は，自律モードの割合が 50%のときの 2 つのプロセスの CPU 使用率の変化である．このときの ClamAV の平均 CPU 使用率は 16%であり，ハイブリッドスケジューリングが無効の場合より多くの CPU 時間を使えていることが分かる．図 15 (b) は自律モードの割合を変化させた場合の CPU 使用率を表す．自律モードの割合を増やすと，ClamAV がより長く動くことが分かる．

© 2011 Information Processing Society of Japan



(a) スループット　(b) 応答時間

図 14　ハイブリッドスケジューリングによるウェブサーバの性能劣化
Fig. 14　The performance degradation of a web server by hybrid scheduling.



(a) ハイブリッドスケジューリング　(b) CPU 使用率

図 15　優先度スケジューリングを用いたハイブリッドスケジューリングの効果
Fig. 15　The effects of hybrid scheduling with priority scheduling.

6. 関 連 研 究

VMM 内で動き，ゲスト OS のプロセスを考慮することができるスケジューラはいくつか提案されている．ゲストを意識した優先度に基づくスケジューリング[19]では，各ゲスト OS は VM 内で動いているプロセスの中で一番高い優先度を VMM に通知する．VMM 内のスケジューラは，通知された優先度に応じて VM の優先度を調節する．この VM スケジューラは各 VM の中の一番高い優先度のプロセスのみを考慮するため，同じ VM 内の優先度が低いプロセスが相対的に多くの CPU 時間を得る可能性がある．Monarch Scheduler では各プロセスを個別にスケジューリングすることができる．

タスク粒度スケジューリング[20]では，各ゲスト OS がすべてのプロセスの優先度を L4 マイクロカーネルに通知する．スケジューリングに関して，L4 マイクロカーネルは VMM と同等の役割を果たす．プロセスのコンテキストスイッチが起こるたびに，L4 マイクロカーネルは通知されたプロセスの優先度に従って，適切なゲスト OS をスケジュールする．しかし，Xen のような通常の VMM で頻繁に VM のコンテキストスイッチを行うとオーバヘッドが大きい．また，これらのスケジューラはゲスト OS から VMM に情報を通知するために，ゲスト OS の修正が必要である．Monarch Scheduler は VM のコンテキストスイッチの回数を増大させず，ゲスト OS の修正も不要である．

一方，タスクを意識した VM スケジューリング[21]では，VMM 内のスケジューラが I/O バウンドプロセスを実行する VM を優先してスケジュールする．このスケジューラは Antfarm[22]やゲスト OS に関するグレイボックス知識を用いて VM 内の I/O バウンドプロセスを検知するため，ゲスト OS の変更は不要である．しかし，このスケジューラは I/O を行わないプロセスの優先度を上げることはできない．

VM 環境におけるシステム全体でのプロセススケジューリングは，分散システムで使われているような協調スケジューリングと同様の手法[23]を使っても実装できる．各 VM でローカルスケジューラを動かし，ローカルスケジューラどうしで通信することによってシステム全体のプロセス情報を取得する．各ローカルスケジューラはその VM 内のプロセスの実行を制御する．ゲスト OS を変更できない場合には，ローカルスケジューラはユーザレベルスケジューリング[24),25]のようにプロセスとして実装される．この場合，もし攻撃者がローカルスケジューラを攻撃することができると，偽の情報を他の VM に伝えることで簡単に DoS 攻撃を行うことができる．さらに，2.2.2 項で述べたように，VM 内で記録したプロセスの実行時間は不正確な場合がある．

Windows ゲスト OS の内部情報を取得することができるライブラリとして，XenAccess[12),26]や EaglEye[27]，VIX[28]が開発されている．これらのライブラリは Xen のドメイン 0 で使われることを想定しているため，VMM 内で動作する Monarch Scheduler 2.0 では利用するのが難しい．ドメイン 0 からはドメイン U のメモリ上の情報を容易に取得できるため，多くの研究でドメイン 0 からの参照が行われている[27)–32]．しかし，ドメイン 0 からドメイン U のメモリにアクセスするのは 2.2.1 項で述べたようにオーバヘッドが大きく，ゲスト OS のメモリを大量に参照することがある Monarch Scheduler には適していない．また，XenAccess はゲスト OS のメモリの変更をサポートしていない．

© 2011 Information Processing Society of Japan

7. まとめ

本稿では，従来の Monarch Scheduler を拡張した Monarch Scheduler 2.0 を提案した．Monarch Scheduler 2.0 ではスケジューリング精度や汎用性，セキュリティの問題を解決し，実用性を向上させた．スケジューラを VMM 内で動作させることで，VM のメモリアクセスのオーバヘッドを削減し，より正確なプロセス時間を取得できるようにした．ゲスト OS として Windows にも対応し，ゲスト OS に依存せずにスケジューリングポリシを記述するための高レベル API を提供することで汎用性を高めた．また，ハイブリッドスケジューリングを行うことで，VM をまたがった DoS 攻撃を緩和し，セキュリティを向上させた．実験により，よく使われるアプリケーションに対して 2 種類のスケジューリングポリシを適用し，プロセスをうまく制御できていることを示した．

参考文献

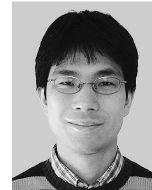
1) ClamWin Team: ClamWin Free Antivirus. http://www.clamwin.com/
2) Google, Inc.: Google Desktop. http://desktop.google.com/
3) Microsoft Corp.: SQL Server.
4) 田所秀和，光来健一，千葉　滋：仮想マシン間にまたがるプロセススケジューリング，情報情報処理学会論文誌：コンピューティングシステム（ACS），Vol.1, No.2, pp.124–135 (2008).
5) Barham, P., Dragovic, B., Fraser, K., 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).
6) Eggert, L. and Touch, J.: Idletime Scheduling with Preemption Interval, *Proc. Symp. Operating System Principles*, pp.249–262 (2005).
7) Microsoft Corp.: Debugging Tools for Windows. http://www.microsoft.com/whdc/devtools/debugging/default.mspx
8) Russinovich, M. and Solomon, D.: Microsoft Windows Internals, Fifth Edition: Covering Windows Server 2008 and Windows Vista.
9) bugcheck: GREPEXEC: Grepping Executive Objects from Pool Memory (2006). http://uninformed.org/?v=4&a=2&t=pdf
10) oldnewthing: Why are process and thread IDs multiples of four?. http://blogs.msdn.com/oldnewthing/archive/2008/02/28/7925962.aspx
11) ReactOS Foundation: ReactOS. http://www.reactos.org/
12) Bryan D. Payne: XenAccess Library. http://code.google.com/p/xenaccess/
13) J. Kneschke: lighttpd. http://www.lighttpd.net/
14) The Apache Software Foundation: Apache HTTP Server Benchmarking Tool. http://httpd.apache.org/
15) R. Coker: Bonnie++. http://www.coker.com.au/bonnie++/
16) Hirabayashi, M.: Hyper Estraier: a Full-text Search System for Communities. http://hyperestraier.sourceforge.net/
17) MPlayer Team: MPlayer – The Movie Player. http://www.mplayerhq.hu/
18) Sourcefire, Inc.: Clam AntiVirus. http://www.clamav.net/
19) Kim, D., Kim, H., Jeon, M., Seo, E. and Lee, J.: Guest-aware Priority-based Virtual Machine Scheduling for Highly Consolidated Server, *Proc. Int. Euro-Par Conf. Parallel Processing*, pp.285–294 (2008).
20) Kinebuchi, Y., Sugaya, M., Oikawa, S. and Nakajima, T.: Task Grain Scheduling for Hypervisor-Based Embedded System, *Proc. Int. Conf. High Performance Computing and Communications* (2008).
21) Kim, H., Lim, H., Jeong, J., Jo, H. and Lee, J.: Task-aware Virtual Machine Scheduling for I/O Performance, *Proc. Int. Conf. Virtual Execution Environments*, pp.101–110 (2009).
22) Jones, S., Arpaci-Dusseau, A. and Arpaci-Dusseau, R.: Antfarm: Tracking Processes in a Virtual Machine Environment, *Proc. USENIX Annual Technical Conf.*, pp.1–14 (2006).
23) Ousterhout, J.: Scheduling Techniques for Concurrent Systems, *Proc. Int. Conf. Distributed Computing Systems*, pp.22–30 (1982).
24) Newhouse, T. and Pasquale, J.: A User-Level Framework for Scheduling within Service Execution Environments, *Proc. Int. Conf. Services Computing*, pp.311–318 (2004).
25) Newhouse, T. and Pasquale, J.: ALPS: An Application-Level Proportional-share Scheduler, *Proc. Int. Symp. High Performance Distributed Computing*, pp.279–290 (2006).
26) Payne, B., Carbone, M. and Lee, W.: Secure and Flexible Monitoring of Virtual Machines, *Proc. Annual Conf. Computer Security Applications*, pp.385–397 (2007).
27) Quynh, N., Suzaki, K. and Ando, R.: eKimono: A Malware Scanner for Virtual Machines, *HITB SecConf 2009* (2009).
28) Brian, H. and Kara, N.: Forensics Examination of Volatile System Data Using Virtual Introspection, *SIGOPS Oper. Syst. Rev.*, Vol.42, No.3, pp.74–82 (2008).
29) Jiang, X., Wang, X. and Xu, D.: Stealthy Malware Detection through VMM-based “Out-of-the-box” Semantic View Reconstruction, *Proc. Conf. Computer and Communications Security*, pp.128–138 (2007).
30) Quynh, N. and Takefuji, Y.: Towards a tamper-resistant kernel rootkit detector, *Proc. Symp. Applied Computing*, pp.276–283 (2007).

© 2011 Information Processing Society of Japan

31) Petroni, N. Jr. and Hicks, M.: Automated Detection of Persistent Kernel Control-flow Attacks, *Proc. Conf. Computer and Communications Security* (2007).
32) Baiardi, F., Maggiari, D., Sgandurra, D. and Tamberi, F.: PsycoTrace: Virtual and Transparent Monitoring of a Process Self, *Proc. Euromicro Int. Conf. Parallel, Distributed and network-based Processing*, pp.393–397 (2009).

(平成 22 年 9 月 29 日受付)
(平成 23 年 2 月 5 日採録)

田所　秀和（学生会員）
2007 年東京工業大学理学部情報科学科卒業．2009 年同大学院情報理工学研究科修士課程修了．現在，同大学院情報理工学研究科博士課程在学中．オペレーティングシステムの研究に従事．



光来　健一（正会員）
2002 年東京大学大学院理学系研究科情報科学専攻博士課程修了．同年日本電信電話株式会社入社．東京工業大学大学院情報理工学研究科数理・計算科学専攻助教を経て，現在，九州工業大学大学院情報工学研究院准教授．博士（理学）．オペレーティングシステムの研究に従事．

千葉　　滋（正会員）
1991 年東京大学理学部情報科学科卒業．1996 年同大学院理学系研究科情報科学専攻博士課程退学．東京大学助手，筑波大学講師を経て，現在，東京工業大学大学院情報理工学研究科教授．博士（理学）．システムソフトウェアの研究に従事．

© 2011 Information Processing Society of Japan