情報処理学会論文誌　Vol. 0　1–11 (??? 1959)

仮想マシン間にまたがるプロセススケジューリング

田所秀和[†1]　光来健一[†1]　千葉滋[†1]

サーバ統合により複数の仮想マシンを用いる場合，重要なサービスを阻害しないようにシステム全体でプロセスに優先度をつけられることが望ましい．しかし，従来では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かった．本論文で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システムを提案する．このシステムでは，仮想マシンモニタから仮想マシン上のゲスト OS のランキューを直接操作することで，ゲスト OS のプロセススケジューリングを調整する．我々はこのようなプロセススケジューリングを Xen を用いて実装し，他のすべてのプロセスが停止した時だけ特定のプロセスを動かすというポリシーを実現できることを実験により確かめた．

Process Scheduling among Virtual Machines

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

Under server consolidation with virtual machines, it is hopeful to prioritize processes in the whole system so that important services are not disturbed by other tasks. However, scheduling that is aware of processes in the whole system was difficult in the previous virtualized systems. This paper proposes process scheduling among virtual machines, which is performed by a virtual machine monitor. In our system, the virtual machine monitor directly manipulates the run queues of guest operating systems running on virtual machines to adjust their process scheduling. We have implemented this system using Xen and experimentally confirmed that we could implement the policy that ran a specified process only while the other processes stopped.

†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

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．例えば，バックアップやウィルススキャン等はサーバの提供するサービスを阻害しないように優先度を低くしておき，システム全体の負荷が低い時に動かすべきである．それらのプロセスを動かす仮想マシンの負荷が低いというだけでは不十分で，他に負荷の高い仮想マシンがある場合にはそのサービスを阻害してしまう．

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．例えば，仮想マシン内のプロセスの 1 つが停止するだけで，その仮想マシン内の他のプロセスに割り当てられる CPU 時間が増え，システム全体の中での優先度が上がってしまう．

このような問題を解決するために，本論文で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システムを提案する．このシステムでは，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．

我々はこのようなプロセススケジューリングを Xen を用いて実装した．スケジューラプロセスをドメイン 0 と呼ばれる特権仮想マシン内で動かし，ドメイン U と呼ばれるその他の仮想マシンのメモリをマップすることでゲスト OS のランキューを操作する．その際にランキューのロックをチェックし，ゲスト OS がロックを取得していないことを確認する．そして，プロセスを一時的にランキューから取り除いて停止させ，しばらくしてからランキューに挿入して再開させることで，優先度に応じた CPU 割り当てを実現する．

本システムを用いて，他の仮想マシンのすべてのプロセスが停止した時だけ特定のプロセスを動かすというスケジューリングポリシーを実装し，実験を行った．実験の結果，ポリシー通りにプロセスがスケジューリングされることを確認した．また，スケジューリングの

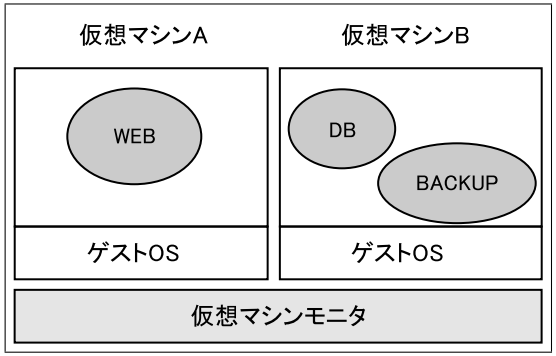


図 1　仮想マシンモニタを使ったサーバ統合
Fig. 1　Server consolidation with a virtual machine monitor.

オーバヘッドを測定したところ，0.8% 程度のオーバヘッドに抑えられることが分かった．

以下 2 章では，既存の仮想マシン環境のスケジューリングとその問題点について述べ，3 章では提案するシステムについて述べる．4 章では実装の詳細について述べる．5 章では実装したスケジューリングポリシーを用いた実験について述べる．6 章では，本システムのゲスト OS へ依存性と実現可能なスケジューリングについて議論する．7 章で関連研究を述べ，8 章で本稿をまとめる．

2. 仮想マシン環境のスケジューリング

近年では，設置スペースや維持費の削減，管理対象マシンの減少による管理負荷軽減などを目的として，仮想マシン技術を用いて複数のサーバを統合することが行われている．例えば，ウェブサーバとデータベースサーバを仮想マシンを使って，一つの物理マシン上に統合する場合を考える．**図 1** のように 2 つの仮想マシンを作り，仮想マシン A をウェブサーバとし，仮想マシン B をデータベースサーバとする．ウェブサーバにはウェブサーバプロセス WEB があり，データベースサーバにはデータベースプロセス DB とデータベースをバックアップするプロセス BACKUP があるとする．

ウェブサーバはデータベースにアクセスすることがあり，DB プロセスの速度低下は，ウェブサーバの速度低下の原因になる．よって，WEB プロセスよりも DB プロセスにより多くの CPU 時間を割当てたい．また，BACKUP プロセスは他のサービスを妨害しないよう，他のどのプロセスよりも優先度を下げたい．

しかし，仮想マシンを用いる場合には，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 B のゲスト OS のスケジューリングを使って，BACKUP プロセスの優先度を下げるか DB プロセスの優先度を上げることで，DB プロセスと BACKUP プロセスの優先順位をつけることはできる．一方，仮想マシンにまたがる DB プロセスと WEB プロセスの優先順位や，BACKUP プロセスと WEB プロセスの優先順位はつけられない．これらの優先度は仮想マシンモニタが行う仮想マシンのスケジューリングに依存する．

これらの仮想マシンには以下の 2 通りの優先順位のつけ方が考えられるが，ゲスト OS のスケジューリングと独立しているため，どちらもうまくいかない．

- 仮想マシン A の優先度が高い場合
  BACKUP プロセスより WEB プロセスを優先することができる．しかし，WEB プロセスは仮想マシン A の CPU 時間をすべて使うので，DB プロセスよりも WEB プロセスの優先度が高くなってしまう．
- 仮想マシン B の優先度が高い場合
  WEB プロセスより DB プロセスを優先することができる．しかし，DB プロセスが止まった場合，BACKUP プロセスが仮想マシン B の CPU をすべて使ってしまう．このとき WEB プロセスよりも BACKUP プロセスの優先度が高くなってしまう．

また，WEB プロセス内の DB にアクセスするスレッドの優先度のみを高くするようなチューニングでも不十分である．このスレッドの優先度だけを高くしても，DB プロセスの優先度が低ければウェブサーバの性能は上がらない．よって，DB プロセスも優先する必要があるが，上記したようにうまく優先順位をつけることができない．

異なる仮想マシン内のプロセスに優先度を付けられるようにするには，仮想マシンのスケジューリングと全てのゲスト OS のプロセススケジューリングを連携させる必要がある．しかし，仮想マシン内で連携のためのコードを実装すると様々なことを考慮しなければならなくなる．仮想マシン間で協調してスケジューリングを行うためには，各仮想マシン内のゲスト OS のスケジューリングの情報を共有し，その情報に基づいて各仮想マシン内のゲスト OS のスケジューリングを調整する必要がある．これらを仮想マシン内で行う際には専用のスレッドが必要になるが，このスレッドはスケジューリングにおいて特別扱いしなければならない．もし，このスレッドを止めてしまうと，スケジューリングを協調させられなくなるからである．そのため，スケジューリングを調整するスレッドを考慮したスケジューラを実装する必要がある．

さらに，スケジューリングスレッドに伴って動くスレッドも特別扱いする必要がある．例

えば，スケジューリングスレッドがスケジューリングの様子をログにデバッグ出力している場合，syslogd などのログを書き出すプロセスも同時に動いてしまう．また，ssh 経由でスケジューリングスレッドをデバッグしている場合，デバッグ出力が端末に出力されるたびに，sshd プロセスが起動する．このように，実際にはあまり動かないはずのプロセスが，スケジューリングスレッドが動くときには必ず動いてしまう．そのため，仮想マシン内のスケジューリングスレッドからはゲスト OS のスケジューリングの情報を正しく得ることができず，例えば，アイドル状態の判定に常に失敗してしまう．

3. 提案システム

3.1 仮想マシンモニタによるプロセススケジューリング

2 章のような問題を解決するために，仮想マシンモニタがシステム全体のプロセスを意識したスケジューリングを行うシステムを提案する．本システムでは，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．ランキューとはプロセスが実行を待つキューなので，これを操作することでスケジューリングが変更可能である．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でシステム全体でプロセスに優先度をつける．

これにより仮想マシンの優先度に依存してその中の優先したくないプロセスまで優先してしまうことを防ぐことができる．2 章の例では，まず仮想マシンのスケジューリングを用いて，仮想マシン A よりも仮想マシン B を優先する．そして，仮想マシンモニタが定期的に仮想マシン A と仮想マシン B それぞれのゲスト OS のランキューを調べ，WEB プロセスと DB プロセスどちらかが動いているときは，仮想マシン B のゲスト OS のランキューから BACKUP プロセスを外し BACKUP プロセスを止める．これによって BACKUP プロセスの優先度を最低にすることができる．さらに，WEB プロセスと DB プロセスが動いているときは，BACKUP プロセスが止まっているので，仮想マシンの優先度により DB の優先度を最も高くすることができる．

本システムでは，仮想マシン内でスケジューリングの協調を実装する場合と比較して，次の 2 つの利点がある．一つは，スケジューリングにおいて特別扱いするプロセスが存在しない点である．本システムは，仮想マシン上のゲスト OS を経由せずに仮想マシンモニタから直接プロセスの実行を調節するので，仮想マシン上のプロセスをすべて平等に扱える．特別扱いが必要なプロセスを考慮してスケジューラを実装する必要がないので，2 章で述べたような意図しない挙動を防ぐことができる．

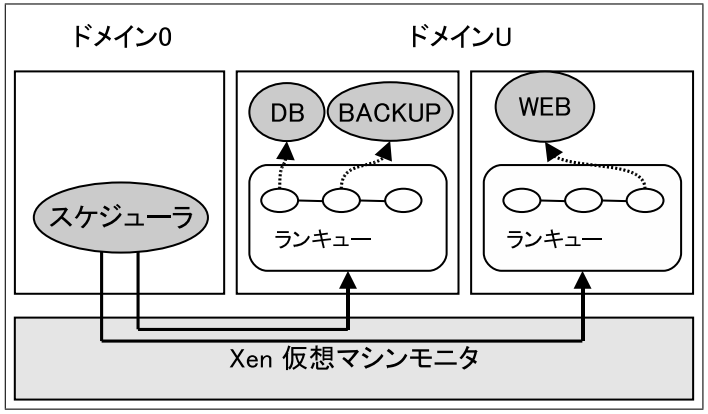


図 2　システムの構成
Fig. 2　An overview of our system.

2 つ目の利点は，ゲスト OS のカーネルを変更する必要がないことである．本システムでは，ゲスト OS のカーネルのメモリを仮想マシンモニタから直接書き換えることでランキューの操作を行なっている．これにより，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．

3.2 システム全体でのプロセス優先度の実現

本システムでは，仮想マシンモニタによる仮想マシンのスケジューリングとゲスト OS によるプロセススケジューリングを最大限利用する．まず，制御したいプロセスにシステム全体での優先度を設定する．それぞれの仮想マシン毎に最大の優先度を持つプロセスを選び，その優先度順に仮想マシンの優先度を設定する．各仮想マシンでは，プロセスに設定されたシステム全体の優先度に応じて，ゲスト OS 内のスケジューラにおける優先度を設定する．仮想マシン内のプロセスに優先度をつけるには，仮想マシンモニタから OS が管理するプロセスの優先度を直接操作する．

仮想マシンモニタによる仮想マシンのスケジューリングとゲスト OS によるプロセススケジューリングだけでは不十分な場合に，ゲスト OS のランキューを操作してプロセスの実行を調整する．例えば，仮想マシンモニタのスケジューラと OS のプロセススケジューラだけでは，自分より優先度の高いプロセスが全て止まったときのみ動くというポリシーは実現

できない．このような場合には，プロセスをランキューから抜くことで実行を制御する．また，プロセスの実行時間を見て優先度順に実行されていなければ，実行されすぎているプロセスを一定時間ランキューから取り除き，実行を遅らせることもできる．

実行中に状況に応じてスケジューリングを変える時は，仮想マシンのスケジューリング，ゲスト OS のプロセススケジューリング，ランキュー操作を再度行う．例えば，ある条件で特定のプロセスの優先度を上げる場合，そのプロセスの属する仮想マシンの優先度とゲスト OS 内でのそのプロセスの優先度を上げる．相対的に他のプロセスの優先度が下がるように他の仮想マシンの優先度と，ゲスト OS 内のプロセスの優先度を変更し，必要ならランキューの操作を行いプロセスの実行を調整する．

4. 実　　装

4.1 システムの概要

本システムでは，仮想マシンモニタとして Xen 仮想マシンモニタ[1] を用いる．Xen 上の仮想マシンはドメインと呼ばれ，ドメインには特権仮想マシンであるドメイン 0 とそれ以外のドメイン U がある．またゲスト OS としては Linux を用い，アーキテクチャは x86_64 を対象としている．

本システムでは，**図 2** のようにスケジューラを Xen 仮想マシンモニタのドメイン 0 上のプロセスとして実装する．このプロセスは仮想マシンのスケジューリングを設定し，さらに一定時間ごとに全てのドメイン U を一時停止し，ドメイン U のゲスト OS のメモリを調べる．ゲスト OS のランキューが一貫性を保って操作可能ならスケジューリングポリシーに応じてランキューを操作してから，ドメイン U を再び動かす．

本システムではスケジューラプロセス自体が常にスケジューリングされるように，ドメイン 0 はスケジューリングの調査の対象とはしない．ドメイン 0 には十分な CPU 時間を割り当てるようにし，ドメイン 0 内のプロセスはランキューから外さない．ドメイン 0 は管理用の仮想マシンであり，通常のアプリケーションは動かさないため，問題となることは少ないと考えられる．

4.2 ドメイン U のカーネルメモリへのアクセス

Xen は各ドメインにメモリを割り当てるために，マシンメモリと疑似物理メモリを管理する．マシンメモリは物理メモリであり，マシンフレームと呼ばれるページサイズのかたまりに分割される．マシンフレームにはマシンフレーム番号と呼ばれる 0 から始まる番号がつけられる．一方，疑似物理メモリは，ドメインが仮想的に物理メモリとして扱うメモリであり，疑似物理フレームと呼ばれるページサイズのかたまりに分割される．疑似物理フレームには疑似物理フレーム番号と呼ばれる 0 から始まる番号がつけられる．Xen はマシンフレームと疑似物理フレームの対応を管理することで，ドメインのメモリを管理する．

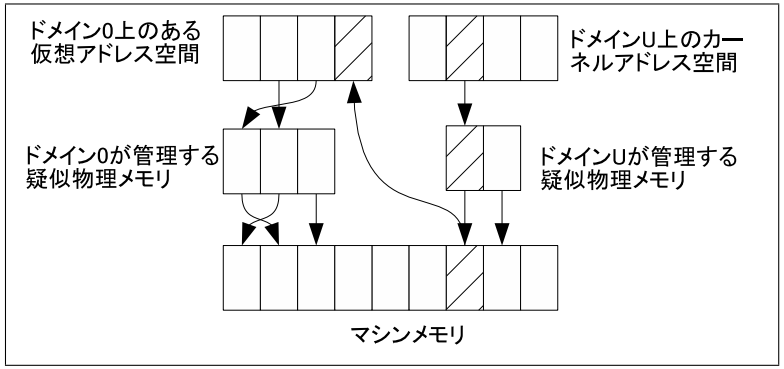


図 3　ドメイン 0 のプロセスにドメイン U のカーネルメモリをマップ
Fig. 3　Mapping DomainU's kernel memory into the address space of a Domain0's process.

ドメイン 0 のプロセスからドメイン U のカーネルメモリを読み書きするために，Xen 仮想マシンモニタの機能を用いる．Xen では**図 3** のように，任意のマシンフレームをドメイン 0 のプロセスのアドレス空間にマップすることが可能である．まず，Xen の API を使って，ドメイン U のカーネル空間の仮想アドレスからそのアドレスに対応する疑似物理フレーム番号を求める．次に，その疑似物理フレーム番号に対応するマシンフレーム番号を求め，そのフレームをマップすることで，任意のドメイン U の任意の仮想アドレスを読み書きすることが可能である．

ページフレームのマップの指定はページ単位で行うため，ページ境界をまたぐ可能性のあるメモリアクセスを工夫する必要がある．例えば，一つの構造体がページ境界をまたぐ可能性がある．そのため，構造体のメンバ変数へのアクセスは，構造体全体をマップしてアクセスするのではなく，メンバ変数のアドレスを計算して，メンバ変数が存在するページをマップしてアクセスする．

4.3 ドメイン U のメモリ操作のための型情報取得

ドメイン 0 のプロセスから，ドメイン U のランキューを操作するためには，構造体の型情報が必要である．本システムでは，構造体の型情報をデバッグオプション付きでコンパ

イルしたカーネルから取得する．デバッグオプション付きでコンパイルされたカーネルは，標準的なデバッグ情報用のフォーマットである DWARF2 フォーマット[2] でデバッグ情報を持っており，DWARF2 フォーマットに対応したデバッガ GDB を使うことで型情報が取得可能である．本システムでは GDB を使い，必要な型情報を事前にすべて取得しておき，プログラムに埋めこみ利用する．GDB を使って型情報を取得するには，`ptype` コマンドを用いる．

メモリ操作のための型情報をソースコードからではなくデバッグ情報から取得した理由は，簡単のためである．Linux カーネルはコンフィグファイルに応じてマクロを定義することで，さまざまな機能の有効無効を設定する．そのため，定義されるマクロによって，構造体の要素が変化してしまう．例えば，`CONFIG_SMP` マクロが定義されているかどうかで，`runqueue` 構造体の定義が変わる．Linux カーネルのソースコードを利用しようとした場合，コンフィグファイルを解析して，マクロを生成し，そのマクロに応じた構造体を利用しなければならない．

4.4　ランキューのアドレスの取得

Xen 仮想マシンモニタからドメイン U のカーネルのランキューにアクセスするためには，ランキューが存在する仮想アドレスが分からなければならない．しかし，SMP 用にコンパイルされた Linux カーネルでは，ランキューのメモリ上の位置は，カーネルが起動するまで不明である．なぜならランキューは CPU 毎に存在し，CPU の数は起動するまで決まらないからである．

カーネルの実行中にランキューのアドレスを調べるために，**図 4** のように CPU の `GS` レジスタからたどる．`GS` レジスタは各 CPU 固有のデータ構造を指すレジスタであり，指す先には `x8664_pda` 構造体が存在する．この構造体にはその CPU のランキューへのポインタや，今動いているタスク構造体へのポインタ，割り込み回数などの情報がある．ドメイン U の `GS` レジスタの値は，Xen の機能を使って調べることができる．

4.5　一貫性を保ったドメイン U のカーネルの操作

Xen 仮想マシンモニタからゲスト OS のカーネルのランキューを操作するときには，ゲスト OS のカーネルがランキューを操作していないかどうかを確認する必要がある．カーネルがランキューを継ぎ換えている間は一貫性のとれたキューになっておらず，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．

本システムでは，ランキューのロックをチェックすることで一貫性を保ってドメイン U のランキューを操作する．ランキューはスピンロックを使って排他制御を行っており，ランキューのスピンロックのためのデータ構造はそのランキュー構造体の中にある．Xen 仮想マシンモニタからゲスト OS のランキューをアクセスする時に，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，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．もしロックが取られていたら，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，少しドメイン U を動かした後で再度チェックする．

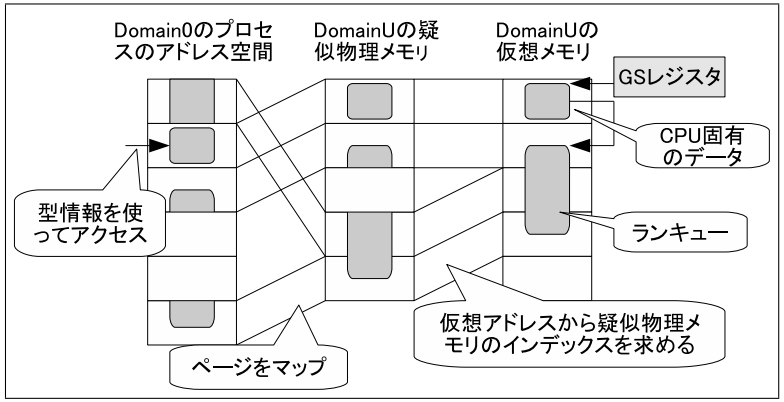


図 4　ドメイン 0 からドメイン U のランキューへのアクセス
Fig. 4　Accessing the runqueue in DomainU from Domain0.

本システムでは，ドメイン U のゲスト OS のランキューのロックをチェックできるようにするために，ドメイン U のカーネルとして SMP 用カーネルを用いる．Linux カーネルは SMP 用カーネルとしてコンパイルされている場合，複数の CPU がカーネル内の同じデータを同時にアクセスしないようにするために排他制御を行っている．

4.6　ドメイン U のランキューの操作

Xen 仮想マシンモニタからドメイン U 内のスケジューリングを調節する手段として，特定のプロセスの実行を一時停止させることと，実行を再開させることを行う．Linux のランキューは，**図 5** のように優先度の配列になっており，実行可能状態のタスク構造体がリストでつながっている．特定のプロセスの実行を一時停止させる方法は，このランキューからそのプロセスが表しているタスク構造体を取り除くことで実現している．実行を再開する方法は，すでに取りのぞいていたタスク構造体を再びランキューに挿入することで実現している．ランキューからプロセスを出し入れする際には，ランキューの操作だけでなく，ランキューにつながっているプロセスの数も更新する．

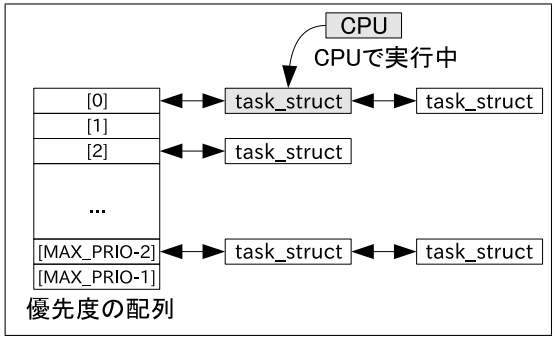


図 5　Linux のランキュー
Fig. 5　The runqueue in the Linux kernel.

SMP 環境においては，Linux 2.6 では CPU 毎にランキューが存在するため，ゲスト OS の一つ以上のランキューそれぞれについて，存在するプロセスの実行を中断または再開させる．例えば，ドメインがアイドル状態かどうかを判定するにはゲスト OS のすべてのランキューを確認する．

ドメイン U のあるプロセスの実行を一時停止する場合，目的のプロセスに対応するタスク構造体をランキューの中から見つけなければならない．本システムではランキューにつながれているタスク構造体をすべて調べ，タスク構造体内のプロセスの名前やプロセス ID を用いて目的のプロセスかどうかを判断する．

実行中でない実行可能状態のプロセスはランキューにつながっているだけであり，ランキューから取り除いても問題ない．ランキューから取り除かれたプロセスは，待ち状態のプロセスと同じ意味になり，実行の順番がまわってこないので実行されないだけである．一方，プロセスが実行中の場合，ランキューから取り除くだけでは不十分である．そのプロセスが実行中であるという情報が Linux カーネルの様々な場所に格納されるためである．もし実行中のプロセスをランキューから取り除いてしまうと，コンテキストスイッチの際に問題が生じる．実行中のプロセスをランキューから取り除くためには，コンテキストスイッチを行うのと同じ処理を行わなければならない．これを仮想マシンモニタから実装するのは容易ではないため，今後の課題である．

4.7　プロセスの nice 値の操作

ゲスト OS 内のプロセスの優先度を変更するには，Xen 仮想マシンモニタからそのプロセスの nice 値を変更する．そのために，ゲスト OS のカーネルメモリを直接参照し，対象プロセスのタスク構造体内の nice 値に対応するメンバ変数を書き換える．Linux ではプロセスの nice 値はタスク構造体の `static_prio` メンバ変数の値と一対一に対応するので，この値を書き換えることで nice 値を変更できる．

4.8　仮想マシンのスケジューリング

本システムでは仮想マシンのスケジューラとして，Xen が提供している Credit Scheduler[3] を用いる．Credit Scheduler は，SMP 環境を効率よく利用でき，各ドメインにパラメータ `WEIGHT` と `CAP` を割当て，使う CPU 時間を制御する．`WEIGHT` は相対的な CPU の割り当て時間を表す．例えばドメイン A の `WEIGHT` の値が 512 で，ドメイン B の `WEIGHT` の値が 256 である時，ドメイン A はドメイン B の 2 倍の CPU 時間を割り当てられる．`CAP` は最大で割り当てる CPU の上限を表す．ドメインはその `CAP` 以上の CPU は使えない．例えば，`CAP` の値が 100 ならそのドメインの使える CPU は最大で物理 CPU 1 つ分であり，50 なら物理 CPU 1 つの半分である．

基本的には `WEIGHT` を使ってドメインが使う CPU 時間を制御する．しかし，各ドメインが別々の物理 CPU を使える場合は `WEIGHT` は効果がないため，`CAP` を使う．例えば，上記のように `WEIGHT` を割り当てたとしても，物理 CPU が 2 つありドメイン A とドメイン B にそれぞれ CPU を一つずつ割り当てている場合には，それぞれのドメインは割り当てられた CPU をすべて使える．このような場合には，ドメイン B の `CAP` の値を 50 にすることで，CPU 時間が無駄になるが，相対的に CPU 時間を制御することができる．

5. 実　　験

実験には，デュアルコア CPU である PentiumD 3.4GHz，メモリ 2Gbyte のマシンを用い，Xen 3.0.4，Linux 2.6.16.33 を用いた．仮想 CPU はドメインに一つずつ割り当て，メモリは，ドメイン 0 に 1Gbyte，ドメイン U に 512Mbyte を割り当てた．また，5.3 節を除いて，ドメイン 0 のスケジューラが動く間隔を 0.5 秒に設定した．

5.1　ドメイン U のメモリアクセスのオーバーヘッド

ドメイン 0 のプロセスにドメイン U のメモリをマップしてアクセスするコストを測定するための実験を行なった．この実験では以下のコードを 100 万回実行した．まず，ドメイン U を止め，ドメイン U で現在動いているプロセスの仮想アドレスから対応するマシンフレーム番号を求める．そして，マシンフレーム番号に対応するページをドメイン 0 のプロセスにマップし，マップしたメモリから 1 ワード読む．さらに，マップしたページをアンマッ

表 1　ドメイン 0 からドメイン U のメモリをアクセスするのにかかる時間の内訳
Table 1　The breakdown of the time needed for accessing memory of DomainU from Domain0.

処理	時間（μs）
ドメインの一時停止と再開	14.84
仮想アドレスに対応するフレーム番号の取得	68.95
ページをドメイン 0 のプロセスにマップ・アンマップ	13.72
1 ワードのメモリアクセス	0.00

プし，ドメイン U を再び動かす．

表 1 にそれぞれの処理にかかった平均時間を示す．結果から，この処理時間の大部分が，ドメインの仮想アドレスからマシンフレーム番号を求める処理に占められていることが分かる．これは，ゲスト OS 内のページテーブルを引くのに時間がかかるためであると考えられる．

5.2　優先度による性能の変化

優先度による性能の変化を測定するために，ウェブサーバの lighttpd と侵入検知システムである tripwire を同時に動かし，lighttpd の性能を測定した．性能の測定には，ウェブサーバの性能を測定するソフトウェアである apache bench[4] を使った．この実験では

- lighttpd を単独で動かしたときの性能
- lighttpd と tripwire を同じ優先度で同時に動かしたときの性能
- lighttpd と tripwire を同時に動かし，かつ本システムを用いて tripwire の優先度を最低にしたときの性能

を測定した．lighttpd と tripwire を同時に動かす場合には，同じドメインと別のドメインで動かす 2 通りを調べた．優先度を最低にしたプロセスは，他のプロセスが動いている間はランキューから外した．

図 6 に lighttpd が 100 万アクセスを処理するのに要した時間を示す．lighttpd は tripwire と同時に動かすと，単独で動かす場合に比べて大幅に性能が低下する．しかし，本システムを用いて tripwire の優先度を下げると，lighttpd 単独の場合と比べて 0.8%の性能低下に抑えられた．

5.3　スケジューリングを行う間隔の影響

スケジューラプロセスがスケジューリングを行う間隔を変化させ，オーバーヘッドを調べた．同じドメインで円周率を計算するプロセス[5]pi1 と pi2 を同時に動かし，pi1 が動いている間は pi2 を停止するというスケジューリングを実行した．そして，スケジューリングを

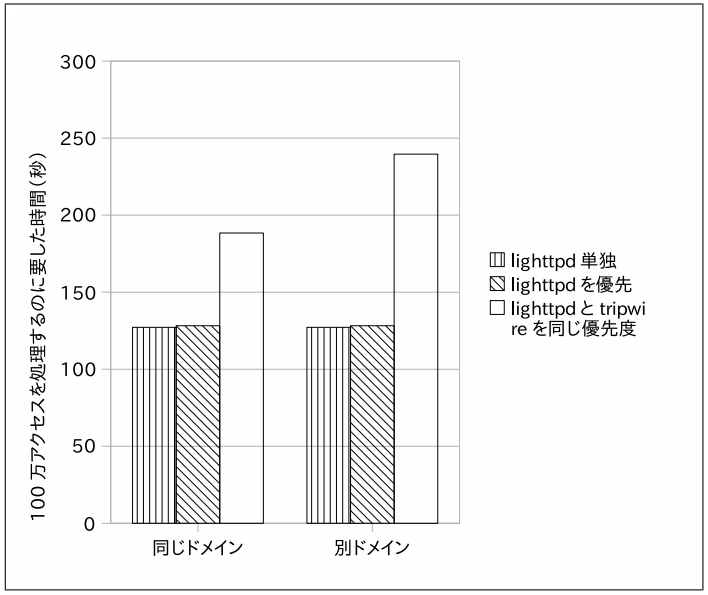


図 6　優先度による性能の変化
Fig. 6　The difference of performance according to the priority of a process.

行う間隔を変更して，実行時間がどう変わるかを測定した．

図 7 に結果を示す．スケジューリングの間隔が極端に短いと，優先度を低くした pi2 の実行時間が長くなることが分かった．これは，本システムのスケジューリングのオーバーヘッドだと考えられる．スケジューリングの間隔を長くすると pi1 の実行時間が長くなるが pi2 の実行時間はあまり長くなっていない．これは，スケジューリングの間隔を長くしたことによりスケジューリングの精度が落ち，pi1 と p2 が同時に動き始めた後，しばらくは pi2 が止まらなかったためと考えられる．

5.4　プロセススケジューリングの挙動

2 つのドメインの中で動く 4 つのプロセス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 3 つのプロセス pi1，pi2，pi3 を動かし，pi1 の優先度を最低にして，他のプロセスが動いていない時だけ動くようにした．ドメイン U2 ではプロセス pi4 を動かした．

プロセスの挙動を**図 8** 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

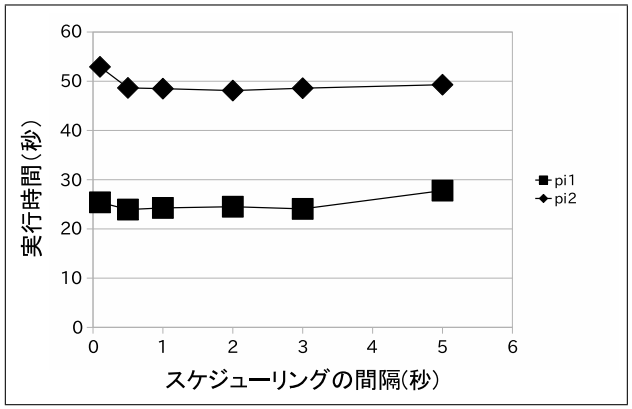


図 7　スケジューリングの間隔の性能への影響
Fig. 7　The impact of changing scheduling interval.

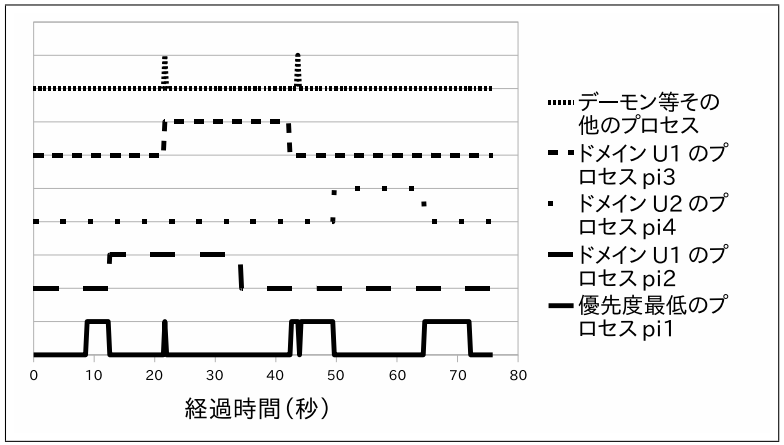


図 8　特定のプロセスの優先度を最低にするスケジューリングの挙動
Fig. 8　The process scheduling for making the priority of a process the lowest.

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．プロセス pi1 が 20 秒付近で動いているのは，スケジューラが一時的に正しく判断できなかったためと考えられる．

5.5　実行時間の短いプロセスの優先

実行時間が短いプロセスを優先できるかどうかを調べるため，実行時間が短いプロセスを優先するスケジューリングポリシーを実装して実験を行った．このようなプロセスを優先するために，そのプロセスが動くドメインの優先度を上げ，さらにゲスト OS 内でのプロセスの優先度を上げた．ドメインの優先度は Xen の Credit Cscheduler の `WEIGHT` によって設定し，優先するプロセスが存在するドメインの `WEIGHT` を 256 とし，別のドメインの `WEIGHT` を 64 とした．ゲスト OS 内の優先度はゲスト OS 内のプロセス構造体の nice 値を直接変更し，優先するプロセスの nice 値は -20 とし，それ以外のプロセスは 19 とした．

実行時間の短いプロセスとして，一定時間スリープして少しだけ CPU を使う処理を繰り返すプログラムを使用した．ドメイン A でこのプロセスを動かし，ドメイン A とドメイン B で円周率を計算するプロセスを動かした．実行時間の短いプロセスを単独で実行した場合，同時に円周率を計算するプロセスを動かした場合，円周率を計算するプロセスを動かしてかつ実行時間の短いプロセスを優先した場合について，実行時間の短いプロセスが CPU を使う処理を一回行うのにかかる時間を比較した．また，`WEIGHT` の効果を確かめるために，物理 CPU を 1 つに制限して実験を行った．

図 9 が結果である．CPU を使う処理にかかる時間を変化させながら，単独実行時を 1 とした実行時間比をプロットしている．CPU を使う処理の実行時間が 20 ミリ秒よりも長い場合には，本システムを使って実行時間の短いプロセスを優先できていることが分かった．スケジューリングポリシーを仮想マシンモニタのスケジューラとゲスト OS のスケジューラのみで実現しているので，ランキュー操作によるオーバーヘッドはない．CPU を使う処理の実行時間がおよそ 10 ミリ秒よりも短い場合，円周率を計算するプロセスの影響を受けなかった．これは，スリープ直後には Linux のスケジューラにより最優先でプロセスが実行されるためである．一方，スリープしてから CPU を使う処理を開始するまでの時間は負荷の有無に関わらずほぼ一定であった．

5.6　仮想マシン間で協調するスケジューラ

仮想マシン内でコードを動かし仮想マシン間で協調するスケジューラを実装し，5.4 節の実験を行った．この実験の目的は，仮想マシン間で協調するスケジューラの実装の際に必要となる作業について明らかにすること，および，提案システムとのスケジューリングの挙動の違いを調べることである．実装したスケジューリングポリシーは，特定のプロセスの優先

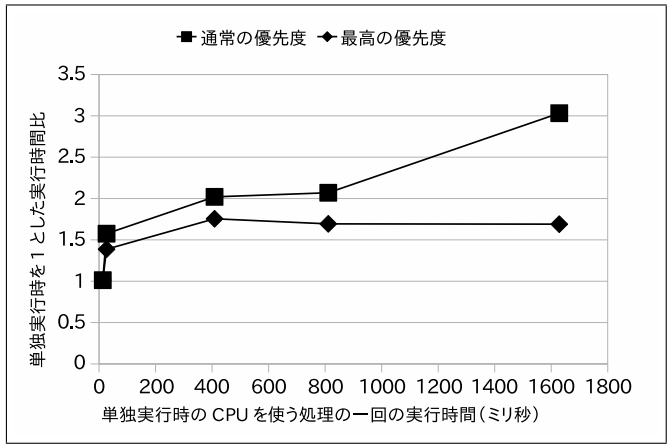


図 9　実行時間の短いプロセスの優先
Fig. 9　The precedence of a process that is executed for a short period.

度を最低にするものである．

この協調スケジューラでは，ドメイン 0 とすべてのドメイン U にスケジューラプロセスを配置する．ドメイン 0 のスケジューラプロセスは TCP/IP 通信により，ドメイン U のスケジューラプロセスから動いているプロセスのリストを取得したり，ドメイン U の特定のプロセスの一時停止，再開を指示する．これらの機能を実現するために，ドメイン U のカーネルに，動いているプロセスのリストを取得するシステムコール，特定のプロセスを一時停止するシステムコール，止めたプロセスを再開するシステムコールを追加する必要があった．

ドメイン U のスケジューラプロセスを実装する際，そのプロセス自身を特別扱いする必要があった．これは，プロセスのリストを取得するシステムコールを発行したときは，必ずそのシステムコールを発行したプロセスが動いているからである．また，システムコールが正しく呼ばれているかをチェックするために，カーネル内で `printk` 関数によってログを出力していたため，ログを書き出す `klogd` プロセスも除外する必要があった．さらに，ssh 経由でドメイン U にログインしてスケジューラプロセスを起動していたため，スケジューラプロセスがログを端末に出力した時に動く `sshd` も除外する必要があった．

このスケジューラを用いて 5.4 節の実験を行ったところ，ほぼ同じ結果が得られた．

6. 議　　論

6.1 ゲスト OS への依存性

本システムは，ゲスト OS のスケジューラの実装に強く依存する．ゲスト OS のメモリを操作するためには，ゲスト OS に合わせた構造体の型情報が必要である．違うバージョンの OS カーネルだけでなく，同じバージョンの OS カーネルでもコンフィグレーションが違えば，構造体の定義が変わる可能性がある．また，ゲスト OS によって構造体をどのように操作してスケジューリングを行うかが異なる．同じ OS であればバージョンが大きく異ならない限り，操作方法は大きく変わらないと考えられるが，スケジューラそのものが変わったり，OS が違ったりすると，操作方法も大きく変わる．

様々なゲスト OS に対応するためには，スケジューリング対象のゲスト OS のすべての種類について型情報を取得し，操作方法について調べる必要がある．本システムでは，デバッグ情報から構造体の定義を得ており，ランキューの操作は OS カーネルのソースコードを見て同じ意味になるように操作を行なっている．提案システムが利用している準仮想化を用いる Xen の場合，ゲスト OS として使える OS カーネルはあらかじめドメイン 0 に置かれているため，その種類だけ対応すればよい．一方，完全仮想化を用いるシステムの場合は，実行時までどのゲスト OS が実行されるか分からないため，ゲスト OS の種類を特定するのは難しい．また，仮想マシンモニタからゲスト OS のメモリを一貫性をもって操作できるように，ゲスト OS のスケジューラが適切に排他制御を行っている必要がある．一貫性をもった操作のためには，ゲスト OS が操作していないタイミングで，データ構造が正しい意味になるように操作する必要がある．例えば，カーネルが行なうランキューの操作が排他制御されている必要がある．Linux の場合 SMP 用カーネルでなければならない．SMP 用カーネルでない場合には，Linux カーネルによる排他制御が不要になり，カーネルはスピンロックを取らずにランキューを操作する．そのため，仮想マシンモニタからゲスト OS がランキューを操作をしているタイミングが分からない．

本システムでは，ゲスト OS のランキューを操作することによりスケジューリングを行うため，ゲスト OS のスケジューラがランキューを使っていない場合，そのまま適用できない．同様に，ゲスト OS が優先度スケジューラを使っていることを想定しているため，他のスケジューラが使われている場合は，ゲスト OS のスケジューラの利用方法を変更する必要がある．

6.2　実現可能なスケジューリング

本システムでは，ある程度時間かけて平均的にポリシーを実現するスケジューリングは実現可能である．例えば，複数のプロセスに優先度を設定するような場合に，一時的には優先度が逆転してしまう場合もあるが，ランキューの操作によって調整することで設定した優先度を平均的に実現することが可能である．また，対象とする全てのプロセスについてランキューを操作して実行を制御することで，様々なスケジューラを実現することが可能である．例えば，ランキューを操作してすべてのプロセスの実行を調整することで，すべてのプロセス間に優先順位をつけることができる．単純に優先度を付けるだけでなく，フェアシェアスケジューリングやプロポーショナルシェアスケジューリングなども実現可能であると考えられる．

一方，リアルタイムスケジューリングのように，短い時間に対して正確に行うスケジューリングには不向きである．これは，本システムが定期的に仮想マシンのスケジューリングとOS のプロセススケジューリング，OS のランキュー操作でスケジューリングを実現しているためである．スケジューリングの間隔を短くすればより正確なスケジューリングが可能になるが，それによってオーバーヘッドが大きくなる．また，ゲスト OS のスケジューラによっては実現できないスケジューリングも存在する．例えば，ゲスト OS がバッチスケジューラを使っている場合，優先度の変更によって CPU の時間を調整できない．

7. 関 連 研 究

Virtual Machine Introspection[6][7] 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提案手法と同じである．Livewire[6] 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[7] 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，提案システムではゲスト OS の状態を操作してスケジューリングを変更する．

Antfarm[8] は，仮想マシンモニタ上から仮想マシン上のゲスト OS に手を加えずにプロセスの状態を取得する技術である．取得したプロセスの状態を用いて仮想マシンモニタ上でゲスト OS 用の I/O スケジューラを実装している．仮想マシン上の OS のソースコードに手を加えずに，OS の情報を取得する点は本研究と同じである．しかし，取得できる情報はプロセスの状態の変化だけであり，プロセスの種類までは分からない．OS の内部構造までは理解しないので，取得できる情報は限られている．同様の技術にはゲスト OS のバッファキャッシュの状態を推測する Geiger[9] もある．

FoxyTechnique[10] では，仮想マシンモニタを用いてゲスト OS を変更せずに新たな資源管理ポリシーを利用可能にする．仮想マシンモニタからドメインの仮想デバイスの振舞いを変更して，ゲスト OS の振舞いを間接的に変更する技術である．しかし，ゲスト OS の挙動を間接的にしか変更できず，スケジューリングの変更は難しい．本研究ではゲスト OS のメモリを操作し直接スケジューリングを変更することができる．

Xen が行うドメインのスケジューリングには，Borrowed Virtual Time（BVT）[11] と Simple Earliest Deadline First（sEDF）[12]，Credit Scheduler がある．BVT では，ドメイン毎に最小時間が決められていて，その間はドメインは割り込まれずに実行する事が保証される．sEDF は，ドメイン毎に周期とタイムスライスが割り当てられ，この周期毎にタイムスライスだけ CPU を使うことができる．これらのスケジューリングはドメインのみを対象としているので，プロセススケジューリングを行うことはできない．

8. まとめと今後の課題

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提案した．Xen 仮想マシンモニタを用い，ドメイン 0 からドメイン U 上のゲスト OS のメモリを操作する．メモリ操作によって，ゲスト OS のスケジューラのランキューを操作してゲスト OS のプロセススケジューリングを変更する．また，この手法を用いて優先度を変更するスケジューラを実装し，実際にスケジューリングが行えることを確認した．スケジューラはドメイン 0 のプロセスとして実装したので，仮想マシンモニタやドメイン U のゲスト OS のカーネルは変更していない．

今後の課題としては，プロセスの実行時間を監視して，実行時間に応じてプロセスの実行を調整するスケジューリングの実現が挙げられる．プロセスの実行時間の監視や実行の制御は，すでに実装した機構の組み合わせで実装可能である．実際にこのようなスケジューラを実装し，実験を行う必要があると考えている．

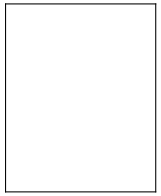
また，I/O バウンドなプロセスの調整も行えるようにする必要がある．本システムではプロセスを停止させる方法として，ランキューからプロセスに対応するタスク構造体を取り除くという方法を用いたため，プロセスを止めるにはタスク構造体がランキュー内にある必要がある．I/O バウンドなプロセスはランキューに存在することが少なく，ランキューを操作する方法での制御が難しい．このようなプロセスは，I/O リクエストをドメイン 0 で調整

することにより制御することが考えられる．

参 考 文 献

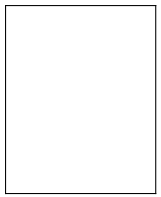
1) Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the Art of Virtualization, *Proc. Symp. Operating Systems Principles*, pp.164–177 (2003).
2) DWARF Standards Committee: The DWARF Debugging Standard, `http://dwarfstd.org/`.
3) XenSource, Inc.: Credit-Based CPU Scheduler, `http://wiki.xensource.com/xenwiki/CreditScheduler`.
4) The Apache Software Foundation: Apache HTTP Server Benchmarking Tool, `http://httpd.apache.org/docs/2.2/programs/ab.html`.
5) Free Software Foundation, Inc.: Computing Billions of *PI* Digits Using GMP, `http://gmplib.org/pi-with-gmp.html`.
6) 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).
7) 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).
8) 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).
9) Jones, S., Arpaci-Dusseau, A. and Arpaci-Dusseau, R.: Geiger: Monitoring the Buffer Cache in a Virtual Machine Environment, *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp.14–24 (2006).
10) Yamada, H. and Kono, K.: FoxyTechnique: Tricking Operating System Policies with a Virtual Machine Monitor, *Proc. Int'l Conf. Virtual Execution Environments*, pp.55–64 (2007).
11) Duda, K. and Cheriton, D.: Borrowed-virtual-time (BVT) Scheduling: Supporting Latency-sensitive Threads in a General-purpose Scheduler, *Proc. Symp. on Operating System Principles*, pp.261–276 (1999).
12) Leslie, I., McAuley, D., Black, R., Roscoe, T., Barham, P., Evers, D., Fairbairns, R. and Hyden, E.: The Design and Implementation of an Operating System to Support Distributed Multimedia Applications, *IEEE Journal of Selected Areas in Communications*, Vol.14, No.7, pp.1280–1297 (1996).

(平成？年？月？日受付)
(平成？年？月？日採録)



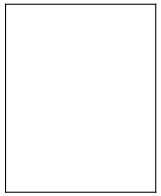
田所　秀和（学生会員）

2007 年東京工業大学理学部情報科学科卒業．現在，同大学大学院情報理工学研究科数理・計算科学専攻在学中．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．現在，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．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 年東京大学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．