平成21年度 修士論文

複数OSに対応した仮想マシン間プロセススケジューリング

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

学籍番号 07M54026

田所 秀和

指導教員

千葉 滋 教授

平成21年7月23日

概要

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 Monarch Scheduler を提案する．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．

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．

Monarch Scheduler 用いて，プロポーショナルシェアスケジューリングを実装し，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．

謝辞

みんなありがとう！

目 次

第 1 章　はじめに　10

第 2 章　問題意識と関連研究　12

2.1　仮想マシンモニタ 12
2.1.1　Xen Virtual Machine Monitor 12
2.2　仮想マシン環境のスケジューリング 13
2.3　ゲスト OS 内部の情報を VMM から取得する技術と応用 . . 14
2.3.1　Virtual Machine Introspection 14
2.3.2　VMWatcher 15
2.3.3　Lares 15
2.3.4　XenFIT 15
2.3.5　XenKIMONO 16
2.3.6　gray-box knowledge 16
2.3.7　FoxyTechnique 16
2.3.8　Antfarm 17
2.3.9　Geiger 17
2.4　OS によるプロセススケジューリング 17
2.4.1　ALPS 17
2.5　Xen による VM スケジューリングポリシー 18
2.5.1　credit scheduler 18
2.5.2　Borrowed-Vritual-Time scheduling 19
2.5.3　Atropos 19
2.6　VM スケジューリングとプロセススケジューリングの組合せ 19
2.6.1　guest aware VM scheduling 19
2.6.2　task aware VM scheduling 19

第 3 章　Monarch Scheduler　21

3.1　Monarch Scheduler の概要 21
3.2　議論 22
3.2.1　ゲスト OS への依存性 22
3.2.2　システム全体でのプロセス優先度の実現 23
3.2.3　実現可能なスケジューリング 24

第 4 章　実装 **26**
4.1 ドメインのメモリへのアクセス 26
4.1.1 Xen のメモリ管理 26
4.1.2 ドメインのメモリのプロセスへのプロセスへのマッピング 27
4.2 Linux ゲスト OS への対応 29
4.2.1 デバッグ情報を使う方法 29
4.2.2 ソースを使う方法 34
4.2.3 一貫性を保ったカーネルの操作 36
4.2.4 ランキューのロック 37
4.2.5 Symmetric Multi Processor 用カーネルの必要性 . . 37
4.2.6 カーネルデータへのアクセス 37
4.2.7 ユニプロセッサ用カーネル 38
4.2.8 SMP 用カーネル 39
4.2.9 プロセスの名前の取得 39
4.2.10 スケジューリングプリミティブ: プロセスの停止と再開 42
4.2.11 プロセスを止める事が可能な条件 43
4.2.12 CPU で実行中のプロセスの制御 43
4.2.13 I/O バウンドプロセスの制御 44
4.3 Windows ゲスト OS への対応 46
4.3.1 型情報の取得 46
4.3.2 ランキューの位置の特定 47
4.3.3 ランキュー操作 50
4.3.4 一貫性を保ったランキュー操作 50
4.3.5 スレッド優先度の変更 51
4.3.6 プロセスの統計情報を利用したスケジューリング . 51
4.3.7 ゲスト OS 内の情報の不正確さ 52

第 5 章　実験 **54**
5.1 ドメイン U のメモリアクセスのオーバーヘッド 54
5.2 優先度による性能の変化 55
5.3 スケジューリングを行う間隔の影響 56
5.4 実行時間の短いプロセスの優先 56
5.5 仮想マシン間で協調するスケジューラ 58
5.6 Windows のランキューの位置特定にかかる時間 58
5.7 Windows でのスレッド優先度の変更 59
5.8 Windows におけるプロセススケジューリングの挙動 60
5.9 I/O バウンドプロセス制御の挙動 61

5.10 プロポーショナルシェアスケジューリング 61
5.10.1 一つのドメイン上で一つのプロセスを制御 61
5.10.2 一つのドメイン上で複数のプロセスを制御 62
5.10.3 二つのドメインで複数のプロセスを制御 68
5.10.4 一つのドメイン上で一つのプロセスを制御 68
5.10.5 Windows 74
5.10.6 一つのドメイン上で複数のプロセスを制御 77
5.10.7 プロセスの CPU 使用率を割合で指定 80
5.11 web サーバを優先するための実用的なスケジューリング . . 82

第 6 章　まとめ **84**

図 目 次

2.1 仮想マシンモニタを使ったサーバ統合 12

3.1 Monarch Scheduler の構成 21

4.1 Xen のメモリ管理 27
4.2 マシンフレームのプロセスへの割り当て 30
4.3 Linux kernel menuconfig での debug 情報付きでコンパイルする指定 32
4.4 attribute でアラインメントを指定してあるソースの例 . . . 34
4.5 linux カーネルでのマクロの例 35
4.6 Linux のランキューの構造 36
4.7 あるユニプロセッサカーネルのシンボルのアドレス 38
4.8 SMP カーネルでのランキューの辿り方 40
4.9 top コマンドで出力される名前 41
4.10 Xen での I/O の流れ 44
4.11 Windows カーネルデバッガによる型情報の取得 47
4.12 Windows カーネルの内部 48
4.13 Windows カーネル内のプロセス候補の取得 49

5.1 優先度による性能の変化 55
5.2 スケジューリングの間隔の性能への影響 56
5.3 実行時間の短いプロセスの優先 57
5.4 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 59
5.5 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 60
5.6 iozone の CPU 時間の変化 61
5.7 単独のプロセスを 10%に制御 62
5.8 単独のプロセスを 20%に制御 63
5.9 単独のプロセスを 30%に制御 63
5.10 単独のプロセスを 40%に制御 64
5.11 単独のプロセスを 80%に制御 64

5.12 単独のプロセスを 100%に制御 65
5.13 40%に制御 66
5.14 60%に制御 66
5.15 80%に制御 67
5.16 100%に制御 67
5.17 200 ミリ秒間隔で制御 69
5.18 400 ミリ秒間隔で制御 70
5.19 600 ミリ秒間隔で制御 71
5.20 1000 ミリ秒間隔で制御 72
5.21 2000 ミリ秒間隔で制御 73
5.22 Linux: 単独のプロセスを制限せずに動かす 74
5.23 Windows: 単独のプロセスを 10%で動かす 75
5.24 Windows: 単独のプロセスを 20%で動かす 75
5.25 Windows: 単独のプロセスを 30%で動かす 76
5.26 Windows: 単独のプロセスを 40%で動かす 76
5.27 Windows: 制限の合計を 40%で動かす 77
5.28 Windows: 制限の合計を 60%で動かす 78
5.29 Windows: 制限の合計を 80%で動かす 78
5.30 Windows: 制限の合計を 100%で動かす 79
5.31 Windows: 制限せずに動かす 79
5.32 割合で指定 81
5.33 VM の VCPU を同じ物理 CPU に固定：割合で指定 81
5.34 VM の VCPU を同じ物理 CPU に固定：割合で指定 82
5.35 lighttpd を優先した場合の CPU 使用率の変化 83

表 目 次

5.1 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 54
5.2 Windows スレッドの優先度の変更 59
5.3 CPU 利用率の平均 62
5.4 制約の合計が 40 パーセント 63
5.5 制約の合計が 60 パーセント 64
5.6 制約の合計が 80 パーセント 65
5.7 制約の合計が 100 パーセント 65
5.8 200 ミリ秒での平均 68
5.9 400 ミリ秒での平均 68
5.10 600 ミリ秒での平均 68
5.11 1000 ミリ秒での平均 69
5.12 2000 ミリ秒での平均 70
5.13 Linux: 単独のプロセスを制限 74
5.14 Windows: 単独のプロセスを制限 74
5.15 Windows: 制約の合計が 40 パーセント 78
5.16 Windows: 制約の合計が 60 パーセント 79
5.17 Windows: 制約の合計が 80 パーセント 80
5.18 Windows: 制約の合計が 100 パーセント 80
5.19 Windows: 制約せずに動かす 80
5.20 lighttpd を優先した場合の性能 83

第1章　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*[32] を提案する．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．

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．

Monarch Scheduler は Linux ゲスト OS と Windows ゲスト OS への対応を行った．Linux ゲスト OS では，デバッグ情報とソースコードからランキュー操作に必要な情報を取得した．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，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 章では予備実験と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，6 章で本稿をまとめる．

第2章　問題意識と関連研究

2.1　仮想マシンモニタ

2.1.1　Xen Virtual Machine Monitor

Xen[2] は仮想マシンモニタであり，仮想マシンの実行単位をドメインとして管理する．Xen 上では複数のドメインを同時に動作させる事ができる．ドメイン上ではオペレーティングシステムが動作する．

Xen は準仮想化と呼ばれる実装手法を使っている．実際のハードウェアを完全にエミュレートする代わりに，仮想マシン環境を実現するのに都合の良い仮想的なハードウェアを再定義する．ドメイン上のオペレーティングシステムのカーネルは，仮想的なハードウェアで動作するように修正しなければならない．しかし，そのように変更されたオペレーティングシステム上で動くソフトウェアに関しては変更は不要である．

ドメインにはドメイン 0 と呼ばれる制御用の特殊なドメインと，ドメイン U と呼ばれるドメインが存在する．ドメイン 0 はドメイン U の動作を制御したり，実際のハードウェアへのアクセスのエントリポイントになっている．

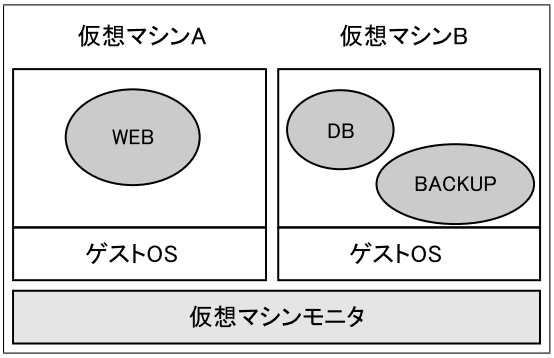

図 2.1: 仮想マシンモニタを使ったサーバ統合

2.2　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 2.1 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．

2.3　ゲスト OS 内部の情報を VMM から取得する技術と応用

2.3.1　Virtual Machine Introspection

Virtual Machine Introspection[8][15] 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[8] 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[15] 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．

2.3.2　VMWatcher

VMwatcher[11] 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．

2.3.3　Lares

Lares[24] 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[23] 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．

2.3.4　XenFIT

XenFIT[25] とは，リアルタイムにファイルの一貫性をチェックするシステムである．旧来の，ファイルの一貫性をチェックするシステムとは違い，ファイルのハッシュ値等のデータベースを作ったり，更新したりする必要が無い．

実装には Xen を用い，ドメイン 0 から対象となるゲスト OS のシステムコールを監視する．システムコールの監視により，特定のディレクトリ配下のファイル内容の変更や，属性の変更を検知することが，可能である．

システムコールの監視方法として，カーネル内のシステムコールに対応するコードにブレイクポイントを設定し，トラップによって検知する方法を用いている．ドメイン 0 から，ゲスト OS カーネル内のシステムコールに対応するコードは，カーネルのデバッグ情報を用いて見付ける．

XenFIT はセキュリティソフトであり，プロセススケジューラである Monarch Scheduler とは異なるシステムである．デバッグ情報を用いて，ゲスト OS カーネル内の情報を取得する点では同じであるが，Monarch Scheduler はゲスト OS を観察するだけでなく，変更している点も異なる．

2.3.5 XenKIMONO

XenKIMONO[26] とは，OS カーネルに対する侵入検知システムである．検知対象となる OS カーネルの外から，カーネルの一貫性のチェックと，カーネル内での異なるレイヤーでの情報のクロスチェックをおこなう．また，プロセスのテキスト領域の一貫性チェックやプロセスの権限の変化も検知可能である．

実装は Xen を用いて，VM 上で検知対象の OS を動かし，VM の外からメモリを観察することで，侵入検知をおこなう．ゲスト OS カーネル内の情報を取得するために，シンボルのアドレスである `System.map` とデバッグ情報を用いている．

XenKIMONO は侵入検知システムであり，プロセススケジューラである Monarch Scheduler とは異なるシステムである．デバッグ情報を用いて，ゲスト OS カーネル内の情報を取得する点では同じであるが，Monarch Scheduler はゲスト OS を観察するだけでなく，変更している点も異なる．

2.3.6 gray-box knowledge

Gray-Box 技術 [1] とは，オペレーティングシステムの内部状態の推測を可能にし，そのオペレーティングシステムの振舞を調整する技術である．特定の I/O が速く終れば，その I/O の対象となるデータがキャッシュから読まれていて，遅ければ，その I/O の対象となるデータがキャッシュに乗っていない，と推定できる．また，オペーティングシステムがタイマ割り込みをカウントする事でタイムスライスを測定しているという Gray-Box 知識を用いて，タイマ割り込みの配送率をを調整する事でタイムスライスの長さを調整する事ができる．

2.3.7 FoxyTechnique

FoxyTechnique[31] 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．

2.3.8 Antfarm

Antfarm[13] とは，仮想マシンモニタ上からドメインに手をくわえずにプロセスの状態を取得する技術である．さらに，取得したプロセスの状態を使って仮想マシンモニタ上で anticipatory I/O scheduling を実装している．

プロセスとアドレス空間は一対一に対応するので，アドレス空間を観察すればプロセスの様子がわかる．例えば i386 では，CR3 レジスタの値が変化したら context switch したと分かり，CR3 レジスタの値が今迄に無い新しい値だったら新しいプロセスが生成されたと分かり，CR3 レジスタの値に対応するページマッピングが存在せず TLB が flush されたら，の CR3 の値に対応するプロセスが終了したと分かる．

ドメイン上のオペレーティングシステムのソースコードに手を加えずに，オペレーティングシステム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，何のプロセスかまでは分らない．オペレーティングシステム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．ドメイン上のオペレーティングシステムに依存せず，Linux 以外でも通用する技術であるという点で，本研究よりも優れている．仮想マシンモニタ上から，ドメイン上のオペレーティングシステムのカーネルの状態を変更する技術ではなく本研究とは方向性が違う技術である．

2.3.9 Geiger

Geiger[14] とは，仮想マシンモニタ上からドメインに手をくわえずにバッファキャッシュの状態を取得する技術である．

Antfarm[13] と同様に，ドメイン上のオペレーティングシステムのソースコードに手を加えずに，オペレーティングシステムの情報を取得する技術である．仮想マシンモニタ上から，ドメイン上のオペレーティングシステムのカーネルの状態を変更する技術ではなく本研究とは方向性が違う技術である．

2.4 OS によるプロセススケジューリング

2.4.1 ALPS

ALPS[21] とは，ユーザレベルでプロポーショナルシェア CPU スケジューリングを実現するライブラリである．このシステムでは，カーネルに特定の機能を要求していない．

ALPS は，cycle という単位でプロポーショナルシェアスケジューリングを達成しようとする．cycle は複数の Quantum という単位からなる．Quantum とは，カーネルが提供するプロセススケジューリングの間隔を一つにまとめたものである．

このシステムでは，この cycle 毎にプロセスの CPU 時間をモニターし，そのプロセスを実行するかを決定している．必然的に，カーネルが提供するスケジューリングの間隔よりも大きな単位でのスケジューリングになる．

この技術はユーザレベルで新しいプロセススケジューリングのポリシーを提供するものであり，Monarch Scheduler のように仮想マシンモニタからスケジューリングを行う技術とは異なる．また，ALSP はプロセスの CPU 使用量を取得するのに OS の API を利用しているが，Monarch Scheduler はゲスト OS 内部の情報を直接見る点も異なる．

2.5　Xen による VM スケジューリングポリシー

2.5.1　credit scheduler

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[30] 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．

2.5.2 Borrowed-Vritual-Time scheduling

Borrowed-Vritual-Time[5] とは，Xen で使われているドメインのスケジューリングポリシーである．Xen 2.0/3.0 ではデフォルトでこのスケジューリングポリシーとして採用されている．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 preempt されずに実行する事が保証される．

このスケジューリングポリシーの特徴として，システムポリシーによって CPU 時間を重みを付けて分配しつつ，リアルタイム性と応答性を重視するアプリケーションに低レイテンシィを提供できるスケジューリングポリシーである．さらに，マルチプロセッサ環境でも低オーバーヘッドで実装可能である．

2.5.3 Atropos

Atropos とは，Xen 2.0 で使う事のできるドメインのスケジューリングポリシーである．

ケンブリッジ大学で開発された，リアルタイムシステム用スケジューラである．Earliest Deadline First を元にしている．このスケジューリングアルゴリズムは，マルチメディア OS で使われている [19].

2.6 VM スケジューリングとプロセススケジューリングの組合せ

2.6.1 guest aware VM scheduling

[16] 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．

2.6.2 task aware VM scheduling

タスクを意識した仮想マシンスケジューリング [18] 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[12] 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．

第3章　Monarch Scheduler

3.1　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 3.1 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．

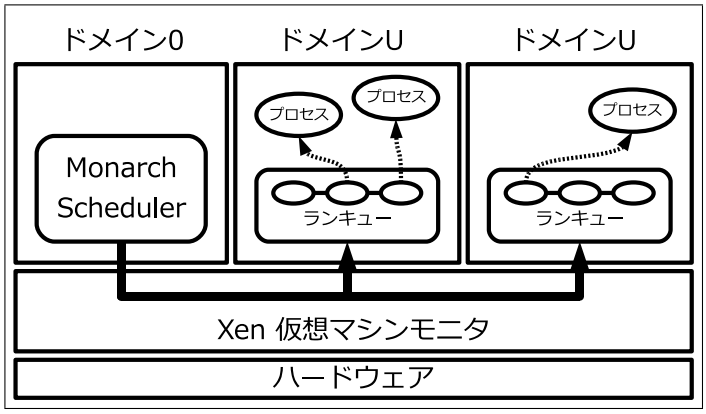

図 3.1: Monarch Scheduler の構成

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[3] 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．

3.2　議論

3.2.1　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．

3.2.2　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．

3.2.3　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．

第4章　実装

4.1　ドメインのメモリへのアクセス

4.1.1　Xen のメモリ管理

Xen はマシンメモリをページサイズ毎に管理している．このページサイズ毎のまとまりをマシンフレーム (machine frame) と呼ぶ．マシンメモリをページサイズ毎に区切った，各マシンフレームに machine frame number(mfn) と呼ばれる 0 から始まる連続した番号を付いている．machine frame number は物理アドレスの順序にしたがって 0 から付けられるので，あるマシンアドレスが，どのマシンフレームに存在するかは，`machine frame number = マシンアドレス >> log2(PAGE_SIZE)` という式で計算できる．`PAGE_SIZE` は `xen/include/ams-x86/page.h` で定義されている．

Xen は各ドメインに対して，いつくかのマシンフレームを割り当てる．ドメインに割り当てられたマシンフレームの machine frame number は連続しているとは限らない．ドメインに割り当てられたマシンフレームを仮想物理フレーム (pseudo physical frame) と呼ぶ．ドメインに割り当てられたマシンフレームに，physical frame number(pfn) と呼ばれる番号が付いている．physical frame number は machine frame number の順序に従って順番に番号が付いている．ドメイン上のオペレーティングシステムは，この仮想物理フレームを physical frame number の順序に連続した本物のメモリだと思って管理している．一つの仮想物理フレームに対応するマシンフレームは常に同じであるとは限らず，変化する場合もある．仮想物理フレームは必ず対応するマシンフレームが存在する．つまり，各ドメインに割り当てたメモリの合計は，物理マシンに積んだメモリより多くはならない．

図 4.1 は，Xen でのメモリ管理を表わしている．一つの箱はページ一つ分を表わしている．machine frame number が 4 のマシンフレームは，domainA の physical machine number が 2 の仮想物理フレームに対応している．

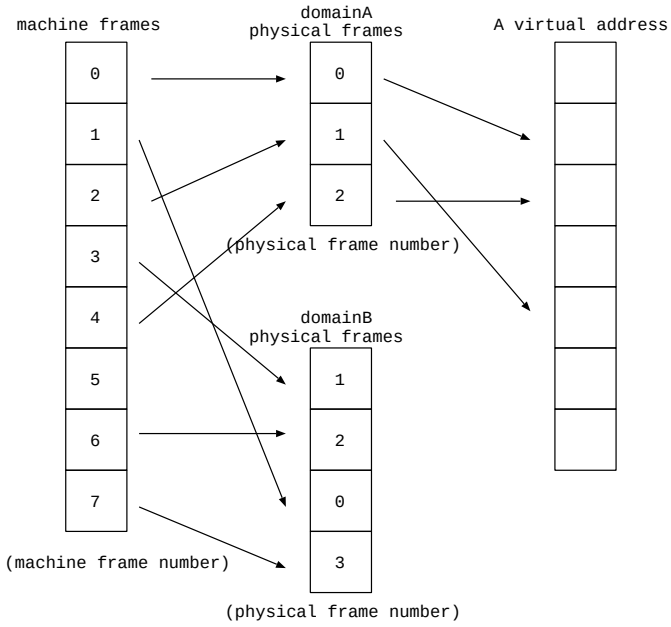

図 4.1: Xen のメモリ管理

4.1.2　ドメインのメモリのプロセスへのプロセスへのマッピング

Xen の機能によって，domain0 のプロセスのアドレス空間に，任意のマシンフレームを割り当てる事ができる．図 4.2 は machine frame number が 7 のマシンフレームを，Domain0 のプロセスの `0x10000` から `0x11000` までのアドレスに貼り付けた様子である．machine frame number が 7 のマシンフレームは，domainC の physical frame number が 3 の仮想物理フレームに対応している．

この場合，domain0 上のプロセスの `0x10000` から `0x11000` までのページへの書き込みは，マシンフレームを通して domainC の仮想物理フレームを変更できる．さらに，domainC の physical frame number が 3 の仮想物理フレームが domainC 上のプロセスの `0x1000` から `0x2000` までのアドレスに割りあてられているとすると，domainC 上のプロセスのアドレス空間を変更する事ができる．

ドメインのメモリを操作する Xen の提供する関数群

Domain0 のプロセス内から呼び，ドメインのメモリを操作する事ができる関数である．すべて xenctrl.h で宣言されている．

- `xc_translate_foreign_address(handle, domID, cpu, addr)`
  プロセスの仮想アドレスから，そのアドレスが存在する pfn を求める関数である．handle は xen へのハンドラを表わす．domID は仮想アドレスがあるプロセスのドメインの ID を表わす．CPU は domID のドメインに割りあてられた CPU の中の一つを表わす．この関数を実行した辞典での，その CPU で走っているプロセスのアドレスを求める事になる．

  図 4.2 で，domainC の CPU 1 で図にあるプロセスが走っているとする．この時
  `xc_translate_foreign_address(handle, domainC, 0, 0x1023)`
  はその `0x1023` が存在するマシンフレームである， 7 を返す．

- `xc_get_pfn_list(handle, domID, *pfn_buf, pfn_buf_size)`
  ドメインに割り当てられた pfn のリストを返す関数である．呼出側で十分なサイズの配列を取っておき，その配列の先頭へのポインタ `pfn_buf` とその配列のサイズ `pfn_buf_size` を渡す．`pfn_buf` に入る pfn の順番は，ドメインから見て連続しているように見える順序である．

  図 4.2 で，`xc_get_pfn_list(handle, domainC, pfn_buf, 4)` と呼ぶと，1，2，0，3 という順番で配列 `pfn_buf` に入る．

- `xc_map_foreign_range(handle, domID, size, prot, mfn)`
  domain0 のプロセスのアドレス空間にマシンフレームを割り当てる関数である．返り値として，プロセスのアドレス空間に割り当てたアドレスが返ってくる．mfn は割り当てたい machine frame number である．prot は割り当てたページのメモリ保護をどのように行うかを表わす．proto は mmap システムコールの第三引数と同じであり，以下の論理和になっている．

  PROT_EXEC ページは実行可能である．

  PROT_READ ページは読み込み可能である．

  PROT_WRITE ページは書き込み可能である．

  PROT_NONE ページはアクセスできない

- `xc_map_foreign_batch(handle, domID, prot, pfn_buf, pfn_buf_size)`
  あるドメインの pfn を連続して domain0 のプロセスのアドレス空間に割り当てる関数である．`pfn_buf` は割り当てたい pfn を要素に持つ配列であり，`pfn_buf_size` はその要素数である．返り値は割り当てた，プロセス内のアドレスである．`proto` は `xc_get_pfn_list` と同じく割り当てたページのメモリ保護を表わす．

図 4.2 で，pfn_buf に 1，2，0，3 と入れて xc_map_foreign_batch(handle, domainC, pfn_buf, 4) と呼ぶと，返り値として domainC 全体のメモリをマップしてその先頭のアドレスが返ってくる．

ドメインのメモリをマップする例

domainC 内のランキューを，Domain0 のプロセスに読み込みのみでマップする例を示す．マップするサイズはページサイズ (PAGE_SIZE) とする．例外処理は省略している．マップしたい domainC のランキューのアドレスを 0x1234 とすると以下のようになる．

```
/* domainC のランキューが存在する フレームを求める */
unsigned long fn =
  xc_translate_foreign_address(xc_handle, domainC, 0, 0x1234);
/* そのフレームをマップする */
void* runqueue =
  xc_map_foreign_range(xc_handle, domainC, PAGE_SIZE,
                       PROT_READ, fn);
/* ランキューへアクセス */
printf("%d\n", (runqueue_t*)runqueu->nr_running);
```

4.2　Linux ゲスト OS への対応

domain0 のプロセスからドメインのメモリにアクセス出来ても，型が分らなければどうアクセスしていいかわからない．C 言語のソースコード上では，正しい型にキャストすれば良い．ここでは，必要な型をどのように取得するかを説明する．

4.2.1　デバッグ情報を使う方法

debug information with gdb

プログラムをデバッグオプションを付けてコンパイルするとデバッグ情報の付いた実行ファイルができる．このデバッグ情報の中に変数の型情報も含まれる．

たとえば，

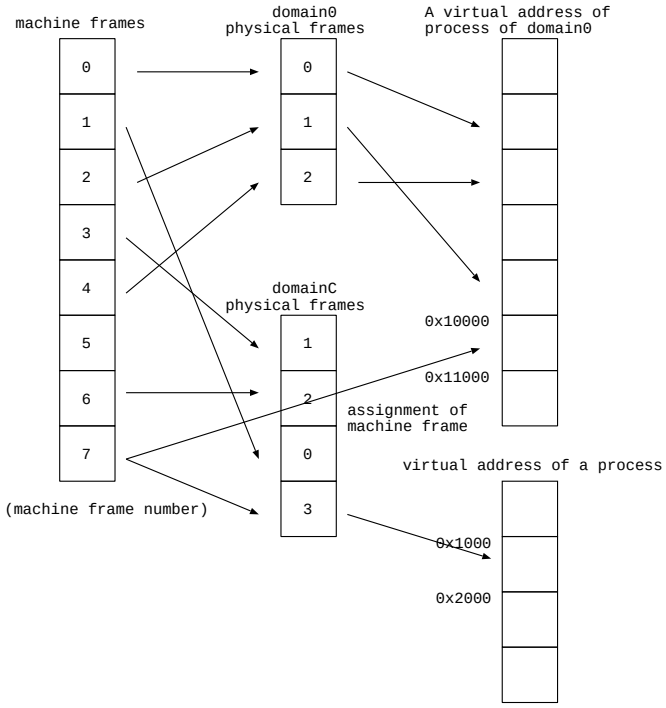

図 4.2: マシンフレームのプロセスへの割り当て

```
#include <stdio.h>

struct HelloMessage {
  char msg[BUFSIZ];
  int length;
  struct HelloMessage* next_msg;
};

struct HelloMessage msg;

int
main(void)
{
  return 0;
}
```

というソースコードがあるとする．

これを GNU C Compiler (gcc) を使い デバッグ情報を付けてコンパイルするオプション -g を付けてコンパイルする．

```
gcc -g hello.c -o hello_dwarf
```

すると，デバッグ情報の付いた実行バイナリ hello_dwarf ができる．デバッグ情報のフォーマットは Debugging With Attibute Record Format (DWARF) である．型情報以外にも DWARF にはさまざまな情報があるが，今回必要なのは型情報だけである．

簡単のために DWARF ファイルから型情報を取り出すのに，gdb (The GNU Project Debugger) を用いた．gdb の組み込みコマンド ptype を以下のように使えば型情報が得られる．

```
% gdb hello_dwarf
GNU gdb 6.6-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu"...
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) ptype msg
```

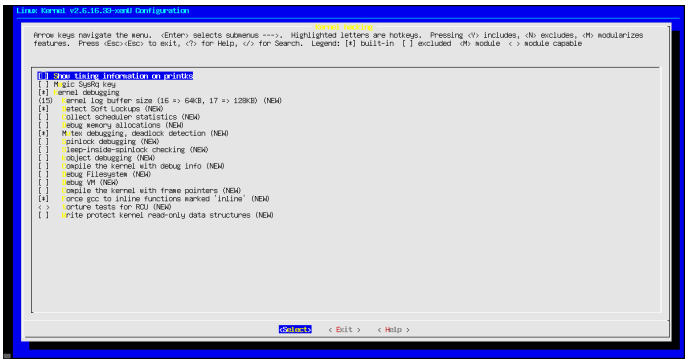

図 4.3: Linux kernel menuconfig での debug 情報付きでコンパイルする指定

```
type = struct HelloMessage {
    char msg[8192];
    int length;
    struct HelloMessage *next_msg;
}
(gdb)
```

Linux カーネルでのデバッグ情報

Linux カーネルをデバッグ情報込みでコンパイルするには，カーネルコンフィグで CONFIG_DEBUG_KERNEL=y とすればいい．menuconfig では

```
Kernel hakcing ->
  [*] Kernel debugging
```

に存在する．

実際のカーネルでのランキュー構造体の型を取得するのは以下のようになる．

```
% gdb vmlinux-syms-2.6.16.33-xenU
GNU gdb 6.6-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu"...
```

```
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) ptype runqueue_t
type = struct runqueue {
    spinlock_t lock;
    long unsigned int nr_running;
    long unsigned int cpu_load[3];
    long long unsigned int nr_switches;
    long unsigned int nr_uninterruptible;
    long unsigned int expired_timestamp;
    long long unsigned int timestamp_last_tick;
    task_t *curr;
    task_t *idle;
    struct mm_struct *prev_mm;
    prio_array_t *active;
    prio_array_t *expired;
    prio_array_t arrays[2];
    int best_expired_prio;
    atomic_t nr_iowait;
    struct sched_domain *sd;
    int active_balance;
    int push_cpu;
    task_t *migration_thread;
    struct list_head migration_queue;
    int cpu;
}
(gdb)
```

デバッグ情報では不十分な場合

図 4.4 のように，ソースコードに `__attribute__` でアラインメントを指定してある場合，デバッグ情報のみではアラインメントを含めた型情報は取得できない．この場合に，デバッグ情報から型情報を取得した場合，アラインメントの情報が無く，構造体の要素がずれてしまう．このような，アラインメントを含んだ構造体に関しては，すべて手動で見付けた．

`__attribute__`とはさまざまな属性を付けられる gcc 拡張である．図 4.4 では属性としてアラインメントを指定している．

図 4.4: attribute でアラインメントを指定してあるソースの例

```
struct thread_struct {
  long unsigned int rsp0;
  long unsigned int rsp;
  long unsigned int userrsp;
  long unsigned int fs;
  long unsigned int gs;
  /* other members... */
  union i387_union i387 __attribute__((aligned(16)));
  int ioperm;
  long unsigned int *io_bitmap_ptr;
  unsigned int io_bitmap_max;
  u64 tls_array[3];
  unsigned int iopl;
};
```

4.2.2　ソースを使う方法

型情報を取得するための自然な方法はソースそのものを使う事である．Linux のソースでも型情報はヘッダに分れているので，型情報に必要なヘッダをインクルードすればよさそうである．しかし，本研究で実装したのはユーザランドのプログラムであり，カーネルソースをそのまま使うのは難しかった．難しい理由として二つある．

ライブラリと名前が衝突

カーネルソースはユーザランドのライブラリと一緒に使う事は想定されていない．よって，ライブラリと名前が衝突する事がある．

本研究実装したプログラムで，Xen のドメインの操作をするのに，Xen のドメインを操作するライブラリを使っている．このドメインを操作するライブラリは libc を使っている．そして，libc とカーネル内の名前が衝突しており，カーネルのソースをそのままインクルードして使う事はできなかった．

図 4.5: linux カーネルでのマクロの例

```
struct runqueue {
       /* other members... */

#ifdef CONFIG_SMP
        struct sched_domain *sd;

        /* For active balancing */
        int active_balance;
        int push_cpu;

        task_t *migration_thread;
        struct list_head migration_queue;
        int cpu;
#endif
};
```

コンフィルファイルを解析する必要がある

Linux カーネルはコンパイル時に，コンフィグファイルを利用してさまざまなカーネルの設定を行なっている．コンフィグファイルに応じてマクロを生成し，そのマクロを使ってカーネルソースをコンパイルしている．生成されるマクロによって，構造体の要素や，どのヘッダをインクルードするかが変化してしまう．

たとえば，コンフィルファイルに CONFIG_SMP=y というエントリが場合，CONFIG_SMP というマクロが定義される．CONFIG_SMP マクロが定義されているかどうかで，図 4.5 にある struct runqueue の型が変化してしまう．

つまり，ユーザランドのプログラムから Linux カーネルのソースツリー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，そのマクロに応じてソースをインクルードする必要がある．

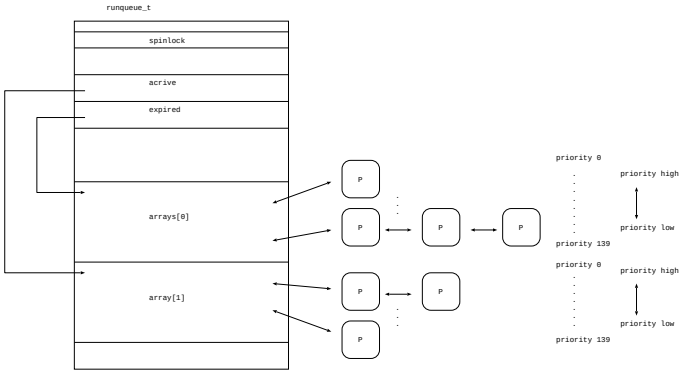

図 4.6: Linux のランキューの構造

4.2.3　一貫性を保ったカーネルの操作

Linux のランキューの構造

Linux のランキュー構造体内には，図 4.6 にあるように，active フィールドと expired フィールドが存在する．active フィールドと expired フィールドはポインタになっていて，それぞれランキュー内の配列 array[0], array[1] を指している．

array[0] と array[1] は サイズ 140 のプロセス構造体へのポインタの配列になっている．そのプロセス構造体へのポインタから，プロセス構造体がリストとして繋がっている．また，プロセス構造体のポインタ配列の順番は，そのプロセスが存在する優先度を表わしている．配列内での順序が若いほど優先度が高い．例えば array[0][130] からたどれるプロセスは，130 の優先度を持っている．

active フィールドの指す先にあるプロセスのリストと expired フィールドの指す先にあるプロセスのリストには，そのリストに繋がっているプロセスに CPU 時間が残っているかどうかの違いがある．active フィールドの指す先にあるプロセスは，CPU 時間が残っている．expired フィールドの指す先にあるプロセスには，CPU 時間が残っていない．

active フィールドの指す先にあるすべてのプロセスが CPU 時間を使い果すと，active フィールドの指す先と expired フィールドの指す先を入れかえる．そしてまた，active フィールドの指す先にあるすべてのプロセスが CPU 時間を使い果すとまた入れかえる，という事を繰替えす．

4.2.4　ランキューのロック

カーネルが Symmetric Multi Processing (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 が同じランキューを同時にアクセスしようとする可能性がある．同時に同じランキューアクセスさせないために，カーネル内ではアクセスする前にスピンロックを取り，操作しおわったらロックを離すという事をやっている．

ランキューのスピンロックはランキュー構造体の中にある．スピンロックの更新はハードウェアの機能を使い必ずアトミック更新するようになっている．

カーネルがシングルプロセッサ用にコンパイルされている場合，ランキューが同時にアクセスされる事はない．よってランキューの操作のために，カーネルがロックを取る事はない．

4.2.5　Symmetric Multi Processor 用カーネルの必要性

外からランキューを操作するには，カーネルがランキューを操作していない時に，操作する必要がある．カーネルがランキューのリストを継ぎ換えている間は，正しいリストになっていない．正しくないリストを操作しようとしても正しくない操作になってしまう．

本研究では，一貫性をもってカーネルを操作するため，ドメインのカーネルとして SMP 用のカーネルを用いた．外部からランキューをアクセスする時に，カーネルがランキューを操作していない事を保証するために，外部からランキュー内のスピンロックを調べ，ロックが取得されているかどうかを調べる．外部からロックを調べて，もしロックが取られていたらカーネルがランキューを操作していると見なす．

シングルプロセッサ用のカーネルだとロックを取らないため，外部から現在カーネルがランキューを操作しているか判断できない．

4.2.6　カーネルデータへのアクセス

外部からランキューを操作するためには，メモリ上のどのアドレスにランキュー構造体が存在するかを知らなければならない．外部からアクセスできるエントリポイントから，どのようにランキュー構造体までたどるか説明する．

図 4.7: あるユニプロセッサカーネルのシンボルのアドレス

```
% nm -n vmlinux-syms-2.6.16.33-xenU | grep --context=2 runqueues
ffffffff803872a0 D per_cpu__nr_multicall_ents
ffffffff803872c0 D per_cpu__mmu_gathers
ffffffff80388300 d per_cpu__runqueues
ffffffff80389580 d per_cpu__phys_domains
ffffffff80389600 D per_cpu__kstat
```

4.2.7　ユニプロセッサ用カーネル

ランキュー構造体は実行バイナリ内のシンボル per_cpu__runqueues として見える．シンボルのアドレスは nm コマンドを用いれば分る．

図 4.7 のカーネルの場合は，リンクアドレス 0xffffffff80388300 に per_cpu__runqueues が存在する．

ユニプロセッサカーネルのメモリイメージはメモリ上にリンクアドレスに対してリニアに配置される．つまり，実際のメモリ上のアドレスは，リンクアドレスからオフセットだけずらすと求める事ができる．

テキスト領域領域と bss 領域のオフセットは -0x100000 (0x100 * PAGE_SIZE) である．ヒープ領域のオフセットは -0xffff880000000000 (__PAGE_OFFSET) である．ランキュー構造体はテキスト領域にあるのでオフセットは -0x100000 である．よって，図 4.7 のカーネルの場合，ランキュー構造体のアドレスは 0xffffffff80388300 - 0x100000 にあるので，この値をマップすればランキュー構造体にアクセスできる．

```
/* domainC のランキューが存在する フレームを求める */
unsigned long fn =
  xc_translate_foreign_address(xc_handle, domainC, 0,
                               0xffffffff80388300 - 0x100000);
/* そのフレームをマップする */
void* runqueue =
  xc_map_foreign_range(xc_handle, domainC, PAGE_SIZE,
                       PROT_READ, fn);
/* ランキューへアクセス */
printf("%d\n", (runqueue_t*)runqueu->nr_running);
```

4.2.8　SMP 用カーネル

CPU 毎のランキューの場所

SMP カーネルの場合，カーネルの実行ファイル内のシンボル `per_cpu__runqueues` の指す先には何もなく，シンボルを利用してランキューにアクセスする事はできない．ランキュー等の CPU 毎に存在するデータは，その CPU に固有の領域に保存されている．CPU に固有の領域はその CPU の gs レジスタから辿る事ができる．

図 4.8 にあるように，gs レジスタが指しているアドレスに `x8664_pda` 構造体 (per cpu data structore) が存在する．その CPU のランキューは `per_cpu__runqueues` のリンクアドレスと，この `x8664_pda` 構造体のメンバ `data_offset` の和でアクセスできる．

ページ毎のマップ

SMP 用カーネルでは，メモリ上のレイアウトがリンクアドレスとリニアに対応していない．よって，シングルプロセッサ用カーネルのように，一括でマップしてオフセットだけずらしてアクセスする事はできない．ただし，ページ単位では必ずリニアに対応する．

本研究の実装では，ページ単位で毎回マップするようにした．マップしたページとマップされたアドレスを管理している．

あるアドレスにアクセスしたいとする．そのアドレスが存在する machine frame number を 関数 `xc_translate_foreign_address` で取得する．まだその machine frame number がマップしてなかったらそのページを実際にマップする．マップした先頭のアドレスが返ってくるので，アクセスしたいアドレスのそのページ内のオフセットにアクセスすれば良い．

ページ単位でマップしているので，ページを跨ぐデータへのアクセスは工夫しなければならない．構造体の先頭と途中のメンバ変数が別のページにあった場合，構造体の先頭のアドレスをマップしても，別のページにあるメンバ変数にはアクセスできない．この場合マップする前に，メンバ変数がどのアドレスに存在するかを計算し，メンバ変数の存在するアドレスを直接マップする必要がある．

4.2.9　プロセスの名前の取得

ドメイン 0 上のプロセスから，ランキューを操作されるドメインのプロセスを区別しなければならない．本研究の実装では，ドメイン上のオペレーティングシステム内で使われる名前を用いた．これは，図 4.9 のように `top` コマンドの出力に用いられる名前と同じである．

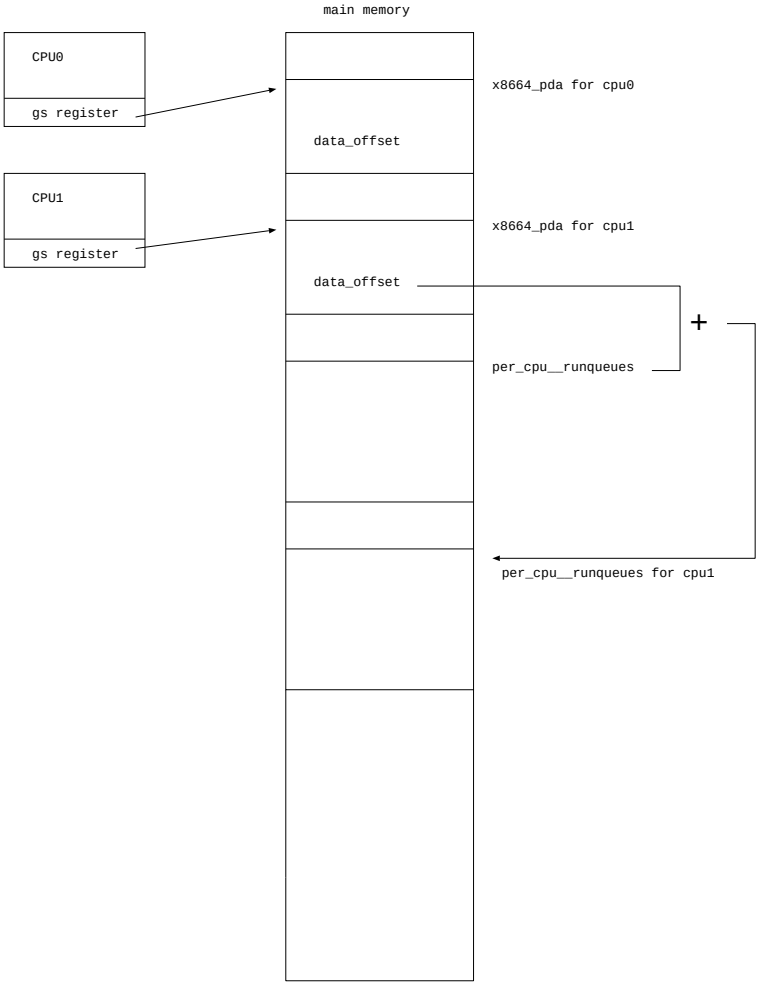

図 4.8: SMP カーネルでのランキューの辿り方

図 4.9: top コマンドで出力される名前

```
top - 13:22:57 up 6 days, 18:27,  1 user,  load average: 0.16, 0.03, 0.01
Tasks:  31 total,   3 running,  28 sleeping,   0 stopped,   0 zombie
Cpu(s): 35.7%us, 23.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  2.4%hi, 38.9%si,  0.0%st
Mem:    524288k total,   425340k used,    98948k free,    58616k buffers
Swap:        0k total,        0k used,        0k free,   282208k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 3480 www-data  25   0 28484 8136  720 R 99.6  1.6   0:10.47 lighttpd
 5761 root      16   0 10596 1240  952 R  0.8  0.2   0:00.02 top
    1 root      16   0  6120  672  556 S  0.0  0.1   0:00.01 init
    2 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/0
    3 root      34  19     0    0    0 R  0.0  0.0   0:00.00 ksoftirqd/0
    4 root      10  -5     0    0    0 S  0.0  0.0   0:00.00 events/0
    5 root      10  -5     0    0    0 S  0.0  0.0   0:00.00 khelper
    6 root      10  -5     0    0    0 S  0.0  0.0   0:00.00 kthread
    7 root      17  -5     0    0    0 S  0.0  0.0   0:00.00 xenwatch
    8 root      17  -5     0    0    0 S  0.0  0.0   0:00.00 xenbus
   12 root      10  -5     0    0    0 S  0.0  0.0   0:00.00 kblockd/0
   47 root      20   0     0    0    0 S  0.0  0.0   0:00.00 pdflush
   48 root      15   0     0    0    0 S  0.0  0.0   0:00.00 pdflush
   50 root      20  -5     0    0    0 S  0.0  0.0   0:00.00 aio/0
   49 root      25   0     0    0    0 S  0.0  0.0   0:00.00 kswapd0
  566 root      19  -5     0    0    0 S  0.0  0.0   0:00.00 kseriod
  682 root      10  -5     0    0    0 S  0.0  0.0   0:00.00 reiserfs/0
  973 root      15   0  4692  964  480 S  0.0  0.2   0:00.01 dhclient3
```

プロセスは Linux カーネル内ではタスク構造体 (struct task_struct) によって表わされている．タスク構造体の定義は linux/sched.h に存在する．プロセスの名前は，以下のようにタスク構造体のメンバ変数 comm に格納されている．

```
#define TASK_COMM_LEN 16
struct task_struct {
        /* other members.. */
        char comm[TASK_COMM_LEN]; /* executable name excluding path
                                     - access with [gs]et_task_comm (which lock
                                       it with task_lock())
                                     - initialized normally by flush_old_exec */

        /* other members.. */
};
```

メンバ変数 comm のサイズは 16 で固定されているので NUL 文字を含めれば，15 文字までしか表わせない．

comm 変数とは別に，プロセスの名前は，タスク構造体の comm 以外に，プロセスの仮想アドレス内に存在する．タスク構造体内の仮想メモリを表わす構造体 struct mm_struct 内に arg_start 変数と arg_end 変数が存在する．プロセスの名前が存在するアドレスは，arg_start 変数の指すアドレスと arg_end 変数の指すアドレスの間に存在する．名前は，arg_start から arg_end までの間に NUL 区切で，名前と引数と環境変数が格納されている．これは execve(2) システムコールの第二引数そのものである．

仮想アドレス内に存在する名前は，そのプロセスのアドレス空間内ではないと意味を持たない．つまり，そのプロセスが実行中ではないとアクセスできない．以上の理由により Xen 側から操作する上で使うのは難しかった．

4.2.10　スケジューリングプリミティブ: プロセスの停止と再開

外部からのゲスト OS を操作する手段として，特定のプロセスを止めるという事を実装した．ドメインを止め，ランキュー内を一通り走査する．走査する時に，止めたいプロセスをタスク構造体の名前まで調べて目的のプロセスかどうか調べる．もし目的のプロセスなら，そのプロセスのタスク構造体のドメイン内でのアドレスを取得する．さらに，そのプロセスをランキューの中から取り除く．ランキューの中から取り除く事により，そ

のプロセスは実行されなくなる．そして再び，ドメインを実行してやれば良い．プロセスを再び実行させたい時は，そのプロセスをランキューの中に再び入れてあげれば良い．

4.2.11　プロセスを止める事が可能な条件

本研究の実装では，プロセスを止める方法として，ランキューからプロセスに対応するタスク構造体取り除く，という方法を用いた．この方法を用いた制限により，Xen 側からドメイン内の目的のプロセスを止める事は限定的しか可能ではない．それは目的のプロセスが次の条件を満すときのみ可能である．

- ランキュー内にある (実行待ちである)
- CPU で実行中ではない

プロセスがランキュー内に存在する

プロセスのタスク構造体が存在するアドレスにアクセスする方法として，ランキューから辿るという実装を採用した．その実装上の制限により，ランキュー内にプロセスが存在する時だけそのプロセスを止める事ができる．

CPU で実行中でない

プロセスが CPU 実行中の場合，その CPU で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．もし実行中のプロセスをランキューから除いてしまうと，レジスタにそのプロセスの情報が残っておりコンテキストスイッチの際に問題が生じる．CPU で実行中のプロセスをランキューから取り除く事は，コンテキストスイッチを行うのと同じ事をしなくてはならず，実装するのは難しい．

4.2.12　CPU で実行中のプロセスの制御

CPU で実行中のプロセスを制御する方法として，ランキュー操作とは別に，プロセスの状態書き換えを用いた．CPU で実行中のプロセスの状態は，`running` である．このとき，タイムスライスを使いきった場合，プロセスの状態に応じて挙動が決まっている．もし `running` の場合は，ま

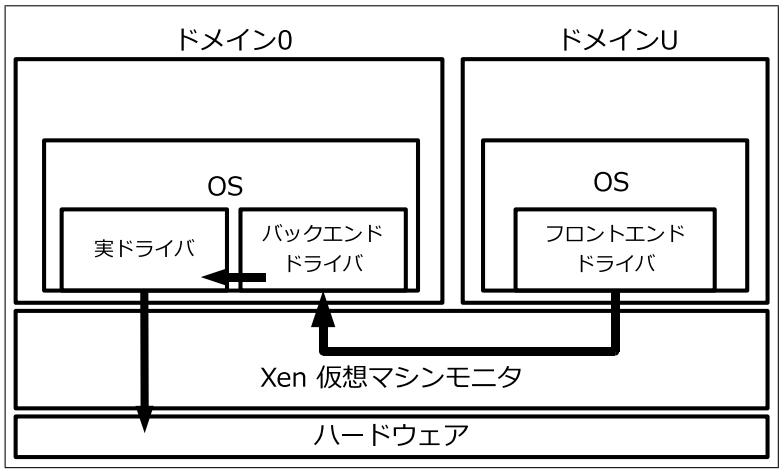

図 4.10: Xen での I/O の流れ

たランキューに挿入される．stopped の場合，ランキューから外されて実行が中断される．

この性質を利用し，CPU で実行中のプロセスの状態を stopped に書き換えることで，CPU で実行中のプロセスを止めることができる．実行を再開させるばあいは，状態を running に書き換えてランキューに挿入する．

これにより，CPU で実行中のプロセスを制御することができる．

4.2.13　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 4.10 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．

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．

I/O バウンドプロセスの制御

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．

4.3　Windows ゲスト OS への対応

4.3.1　型情報の取得

図 4.11 のように，WinDbg カーネルデバッガ [20] 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．

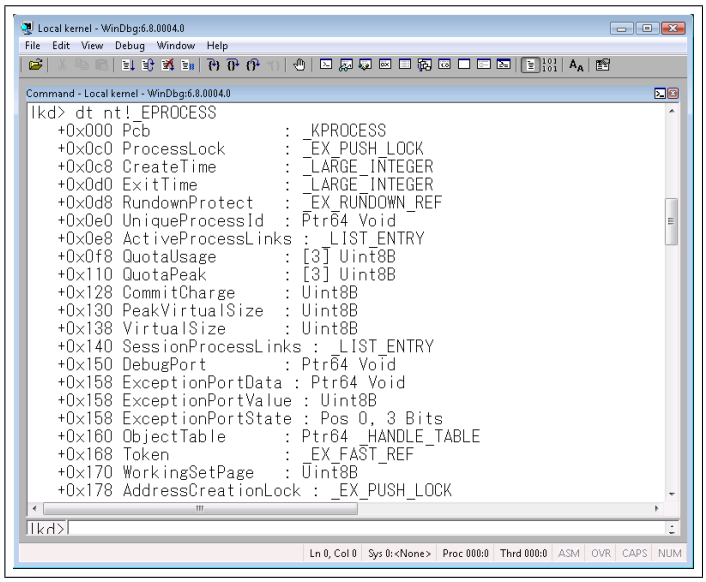

図 4.11: Windows カーネルデバッガによる型情報の取得

4.3.2　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 4.12 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 [28]．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

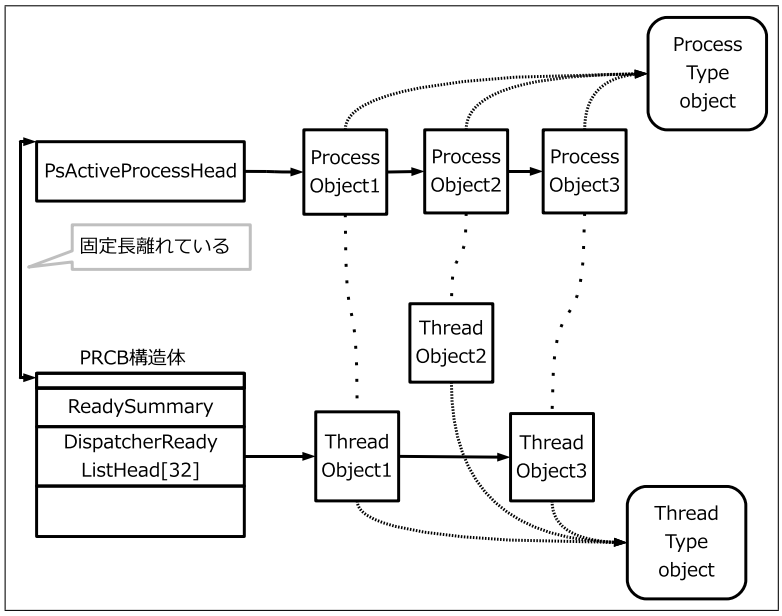

図 4.12: Windows カーネルの内部

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 [4].

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 [22]
  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．

これらの知識を用いて，Windows のメモリイメージからプロセスオブジェクトの候補を取得している様子が図 4.13 である．

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

図 4.13: Windows カーネル内のプロセス候補の取得

```
   0 Idle            priority: 0  next: 0xffffffffffffff18
   4 System          priority: 8  next: 0xfffffa80033af850
1540 svchost.exe     priority: 8  next: 0xfffffa80032aa450
1620 svchost.exe     priority: 8  next: 0xfffffa80032afc10
1584 taskeng.exe     priority: 6  next: 0xfffffa800327cc10
1684 SearchIndexer.e priority: 8  next: 0xfffff800019b6338
 340 smss.exe        priority: 11 next: 0xfffffa8002de5920
 968 svchost.exe     priority: 8  next: 0xfffffa800301c0e0
 116 svchost.exe     priority: 8  next: 0xfffffa800301f570
 244 svchost.exe     priority: 8  next: 0xfffffa800304d040
 416 SLsvc.exe       priority: 8  next: 0xfffffa8003084880
 388 audiodg.exe     priority: 8  next: 0xfffffa800305b0e0
 408 svchost.exe     priority: 8  next: 0xfffffa8003036c10
 768 svchost.exe     priority: 8  next: 0xfffffa80030b5c10
1100 svchost.exe     priority: 8  next: 0xfffffa8003100ae0
1252 spoolsv.exe     priority: 8  next: 0xfffffa800315dc10
1276 svchost.exe     priority: 8  next: 0xfffffa8003277860
 540 winlogon.exe    priority: 13 next: 0xfffffa8002e8fab0
 496 wininit.exe     priority: 13 next: 0xfffffa8002e2d8f0
 504 csrss.exe       priority: 13 next: 0xfffffa8002e16040
 600 lsass.exe       priority: 9  next: 0xfffffa8002eaa040
 584 services.exe    priority: 9  next: 0xfffffa8002e79760
 608 lsm.exe         priority: 8  next: 0xfffffa8002f4c040
 760 svchost.exe     priority: 8  next: 0xfffffa8002f7c9b0
 824 svchost.exe     priority: 8  next: 0xfffffa8002fb9880
 896 LogonUI.exe     priority: 13 next: 0xfffffa800300dc10
 856 svchost.exe     priority: 8  next: 0xfffffa8002fb0460
 452 csrss.exe       priority: 13 next: 0xfffffa8002e21880
```

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．

4.3.3　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[27] のソースコードを参考にした．

4.3.4　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，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．

4.3.5　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．

4.3.6　プロセスの統計情報を利用したスケジューリング

Monarch Scheduler では、プロセスの統計情報に応じてプロセススケジューリングを行うことができる。現在対応しているプロセスの統計情報は、使用した CPU 時間と、発行した I/O のバイト数である。

プロセスの使用した CPU 使用率に応じてプロセススケジューリングを行う場合、プロポーショナルシェアスケジューリングなどが可能になる。例えば、特定のプロセスの CPU 使用率を 50%に制限したり、別のプロセスの CPU 使用率以下に制限する等、CPU 使用率を利用したスケジューリングができる。

Monarch Scheduler では、CPU 使用率を計算するために、定期的にスケジューリング対象のプロセスの使用した CPU 時間を取得する。前回からの CPU 時間の増加量を、スケジューリング間隔の時間で割ることで CPU 使用率とする。

Linux でのプロセスの統計情報の取得

プロセスが利用した CPU 時間として、タスク構造体内の utime と stime メンバ変数を利用する。utime は、タスク構造体に対応するプロセスが利用した、ユーザ時間であり、stime はカーネル時間である。

また、プロセスが発行した I/O のバイト数として、タスク構造体内の rchar と wchar メンバ変数を利用する。rchar は、タスク構造体に対応するプロセスが read した byte 数であり、wchar は write した byte 数である。

Windows でのプロセスの統計情報の取得

Windows でスケジューリングの単位がプロセスではなく、プロセスに 0 以上含まれるスレッドである。CPU 時間や発行した I/O はスレッド毎に管理されている。よって、プロセスに対応する全てのスレッドから、統計情報を取得し、その合計をプロセスの統計情報としている。

スレッドの情報は、`ETHREAD` 構造体内の、`KernelTime` と `UserTime` メンバ変数を利用する。`UserTime` は、タスク構造体に対応するプロセスが利用した、ユーザ時間であり、`KernelTime` はカーネル時間である。

発行した I/O バイト数の情報については、未実装である。

4.3.7　ゲスト OS 内の情報の不正確さ

仮想マシン環境において，CPU が足りていない状況では，ゲスト OS 内の統計情報が不正確になる [29]．CPU が足りていない状況では，仮想マシンモニタは，ゲスト OS に CPU を常に割り当てておくことができず，定期的に別のゲスト OS に CPU を割り当てる．このとき，止められたゲスト OS では，タイマー割り込みの間隔が通常とは異なり，時間に関する統計情報が不正確になる．

Linux や Windows を含む多くの OS カーネルは，以下の 2 つの機能を，タイマー割り込みが一定間隔で発生するという仮定に依存している．

- 起動から経った時間などのために，実時間が進んでいることを知る．
- プロセスが動いていた時間など，統計情報の計算．

仮想マシンモニタが，仮想マシンを一定時間止めてから動かす場合，実時間を合わせるために，止まっていた時間に対応する仮想タイマー割り込みを，高速に発生させる．例えば，10 ミリ秒毎にタイマー割り込みが発生する状況で，仮想マシンを 50 ミリ秒止めた場合，再開させる時に，5 つのタイマー割り込みを高速に発生させる．

このとき，そのゲスト OS で `running` 状態のプロセスに関する統計情報が歪んでしまう．上の例では，仮想マシンを 50 ミリ秒止まっていたにもかかわらず，タイマー割り込みは 5 回発生しているので，その間に動いているプロセスは，50 ミリ秒動いていたと，OS カーネルは判断してしまう．

VMWare ESX Server 3.x では，ゲスト OS 内に時間の歪みを引き受けるプロセスを動かすことで，この問題を解決している．ゲスト OS 内で事前に，CPU を使うだけの VMDesched というプロセスを走らせておく．仮想マシンモニタは，仮想マシンを止めるときに，このプロセスが必ず動

くようにする．仮想マシンが再開するときに，発生するタイマー割り込みはすべて VMDesched が引き受けるので，それ以外のプロセスの統計情報は狂わない．

この制約のため，ゲスト OS の時間に関する統計情報を利用してプロセススケジューリングする場合，Monarch Scheduler では，物理 CPU が十分にあるという仮定をおいている．

第5章　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　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プし，ドメイン U を再び動かす．

表 5.3 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．

表 5.1: 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

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

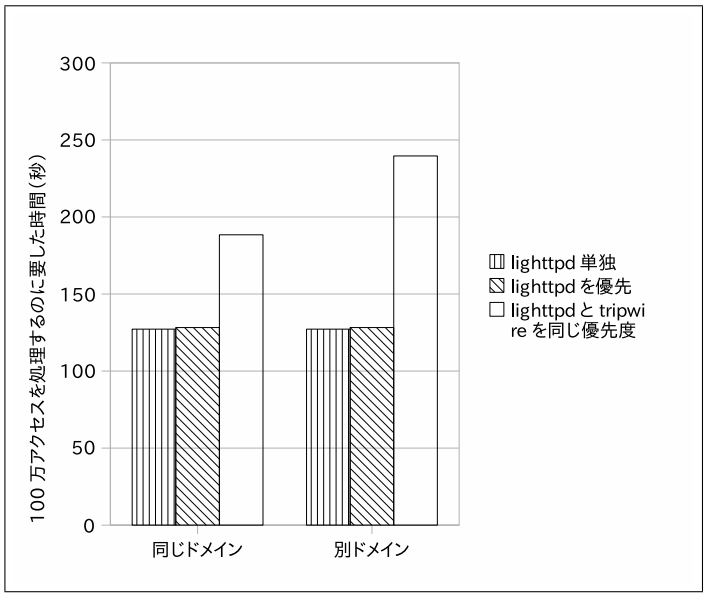

図 5.1: 優先度による性能の変化

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[10] と侵入検知システムである tripwire[17] 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[6] 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 5.1 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．

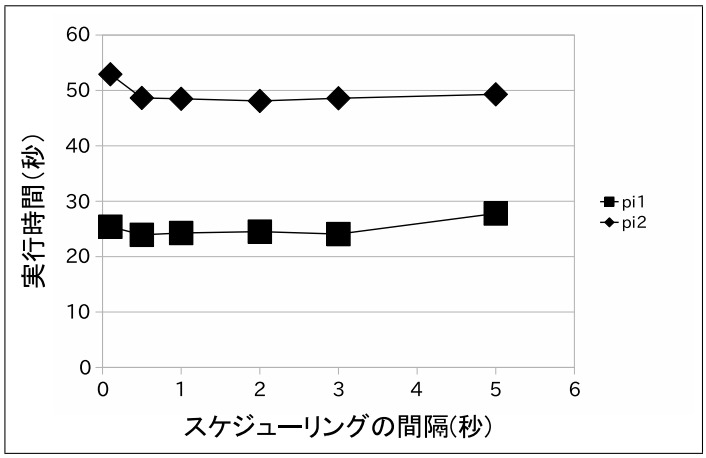

図 5.2: スケジューリングの間隔の性能への影響

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 [7]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 5.2 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　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 Ccheduler 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

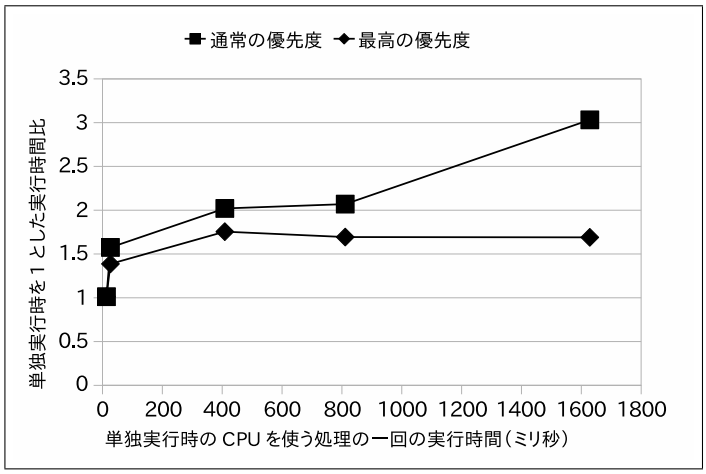

図 5.3: 実行時間の短いプロセスの優先

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 5.3 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．

5.5　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，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．

5.6　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 5.4 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

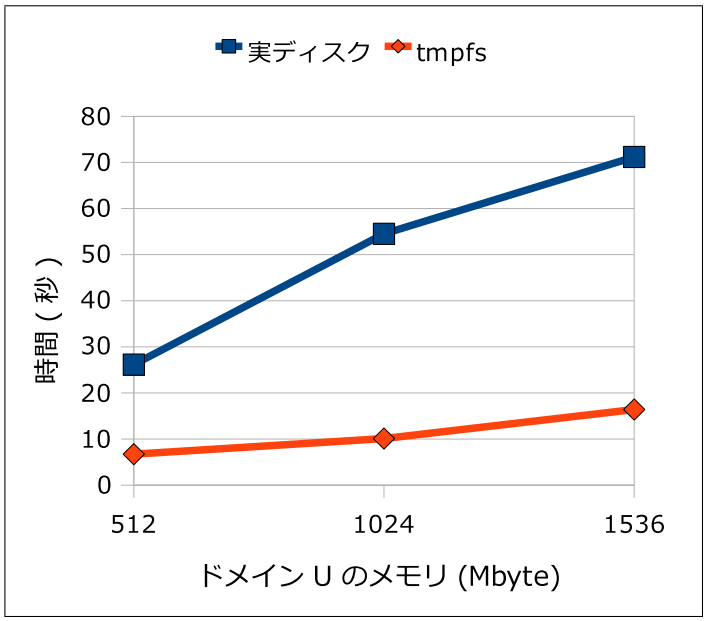

図 5.4: Windows のランキューの仮想アドレスを取得するのにかかる時間

表 5.2: Windows スレッドの優先度の変更

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

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．

5.7　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．

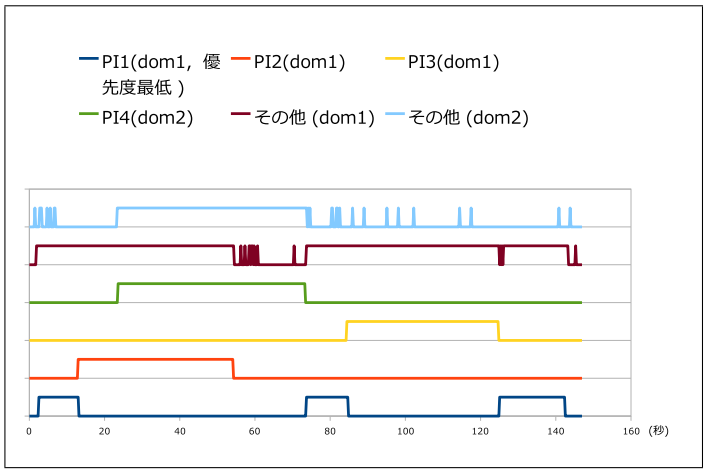

図 5.5: 特定のプロセスの優先度を最低にするスケジューリングの挙動

実験結果は表 5.2 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．

5.8　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 [7]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 5.5 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．

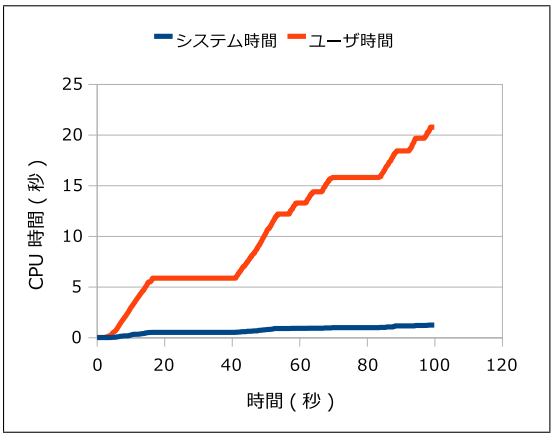

図 5.6: iozone の CPU 時間の変化

5.9　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[9] 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 5.6 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．

5.10　プロポーショナルシェアスケジューリング

Monarch Scheduler を用いて，プロポーショナルスケジューリングを実装し，挙動を詳しく調べた．

5.10.1　一つのドメイン上で一つのプロセスを制御

一つのドメイン上で一つのプロセスの CPU 使用率を制御した．目標とした CPU 使用率は，ぞれぞれ 10,20,30,40%である．

図 5.7 から図 5.10 までが，CPU 使用率の変化であり，表 5.3 が，CPU 利用率の変化の平均である．

表 5.3: CPU 利用率の平均

CPU 使用率の制約 (%)	10	20	30	40	80	100
結果 (%)	11.07	20.91	30.43	39.52	74.2	89.95

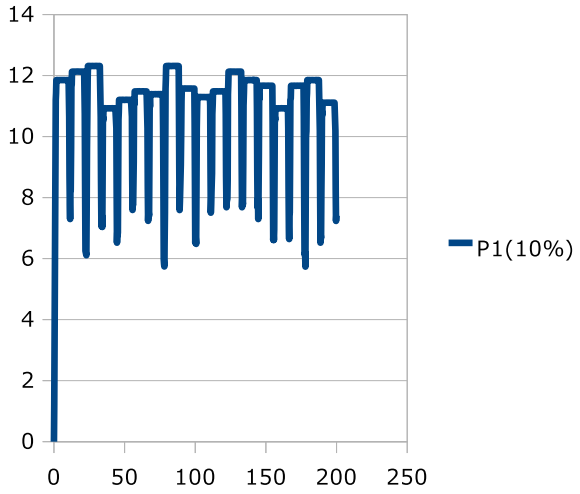

図 5.7: 単独のプロセスを 10%に制御

CPU 利用率の平均は，目標の約 10%以内になっている．特に 30 と 40%では，1.2%以内に抑えられている．

また，100%では 10%低い値になった．100%とは，CPU 利用率を制限しないことを表わすので，これがオーバーヘッドと考えられる．

5.10.2　一つのドメイン上で複数のプロセスを制御

一つのドメイン上で 4 つのプロセスを動かし，それぞれのプロセスに CPU 使用率の制限をかけた．CPU 使用率の合計をそれぞれ 40,60,80,100%の 4 通り調べた．

表 5.4 から表 5.7 が，それぞれのプロセスの CPU 使用率の平均であり，図 5.13 から図 5.16 が，グラフで表わしたものである．

制約の値が大きいときと，小さいときに精度が悪くなることがわかった．特に，制約の値が大きい場合には，オーバーヘッドが原因だと思われる．

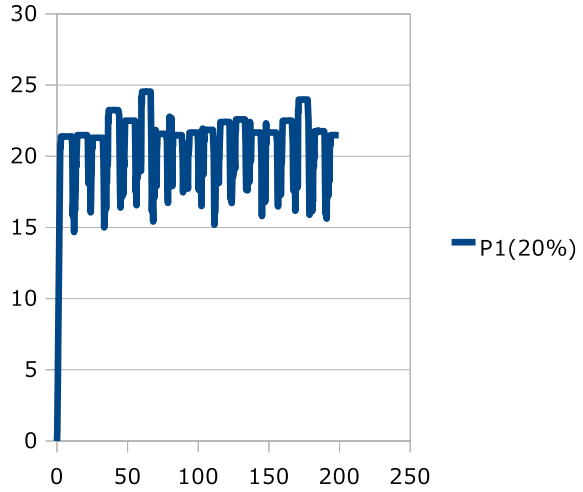

図 5.8: 単独のプロセスを 20%に制御

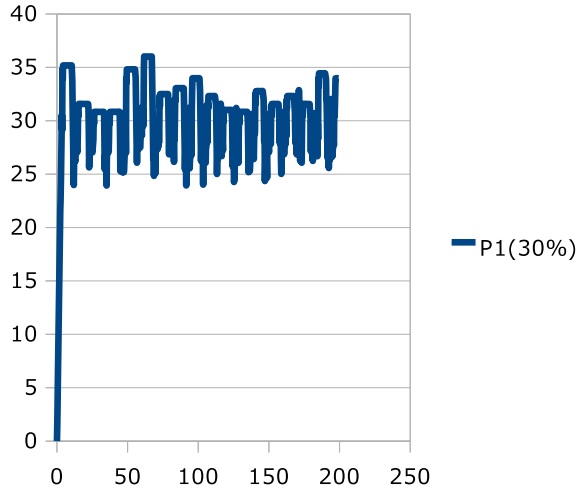

図 5.9: 単独のプロセスを 30%に制御

表 5.4: 制約の合計が 40 パーセント

制約 (%)	4	8	12	16
CPU 使用率の平均 (%)	5.17	8.78	12.85	16.43

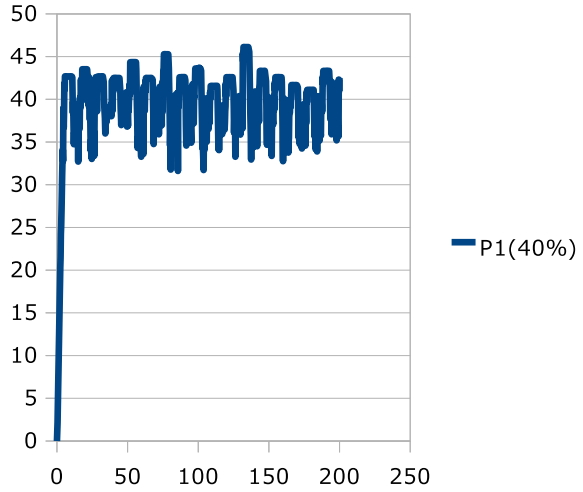

図 5.10: 単独のプロセスを 40%に制御

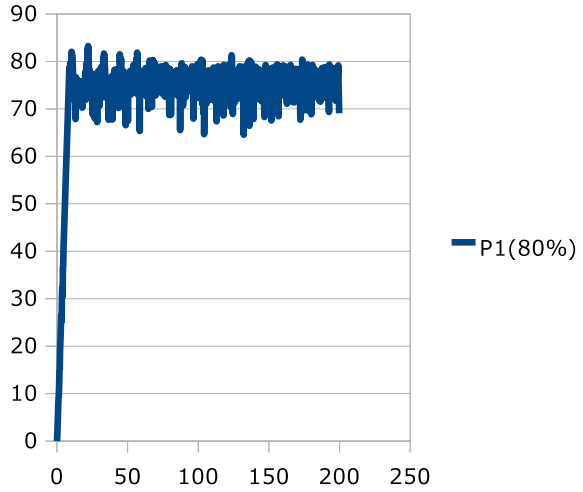

図 5.11: 単独のプロセスを 80%に制御

表 5.5: 制約の合計が 60 パーセント

制約 (%)	6	12	18	24
CPU 使用率の平均 (%)	7.49	12.16	17.85	22.0

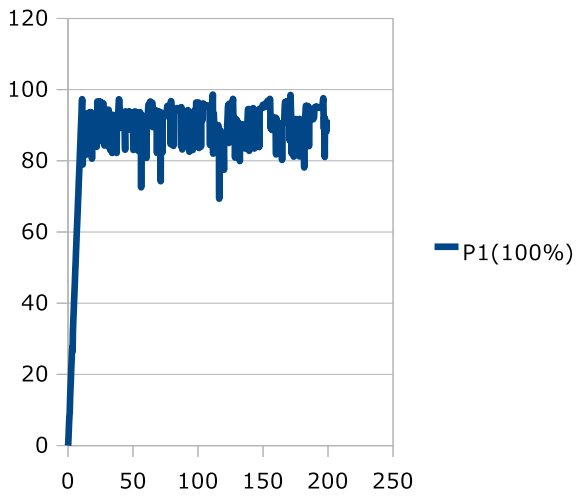

図 5.12: 単独のプロセスを 100%に制御

表 5.6: 制約の合計が 80 パーセント

制約 (%)	8	16	24	32
CPU 使用率の平均 (%)	8.92	15.43	21.57	26.46

表 5.7: 制約の合計が 100 パーセント

制約 (%)	10	20	30	40
CPU 使用率の平均 (%)	10.7	17.77	24.13	25.26

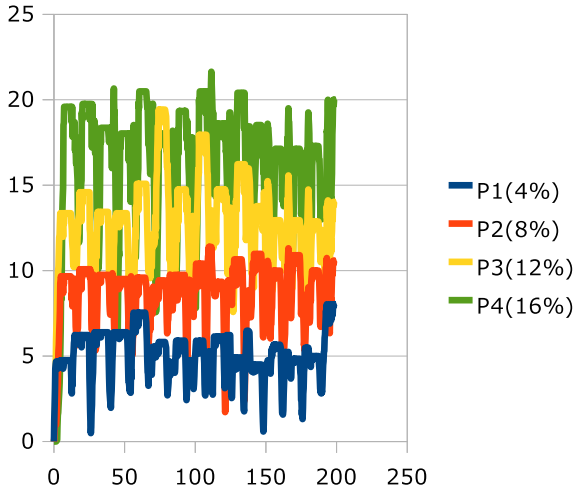

図 5.13: 40%に制御

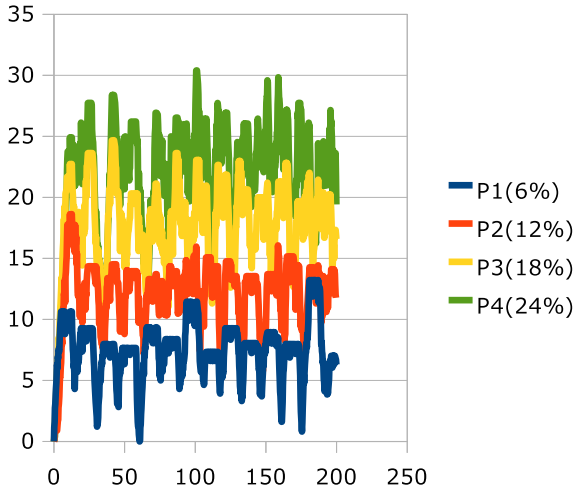

図 5.14: 60%に制御

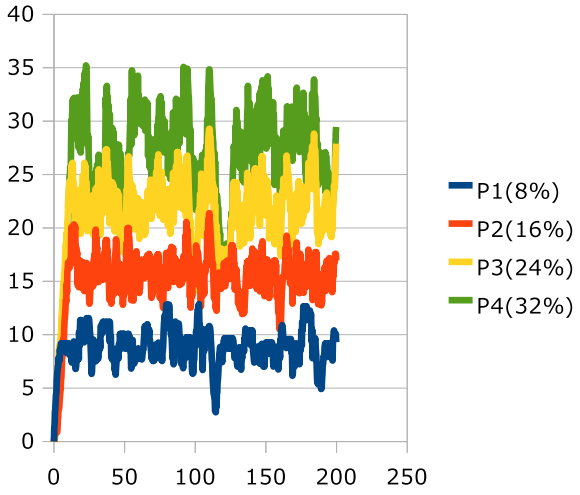

図 5.15: 80%に制御

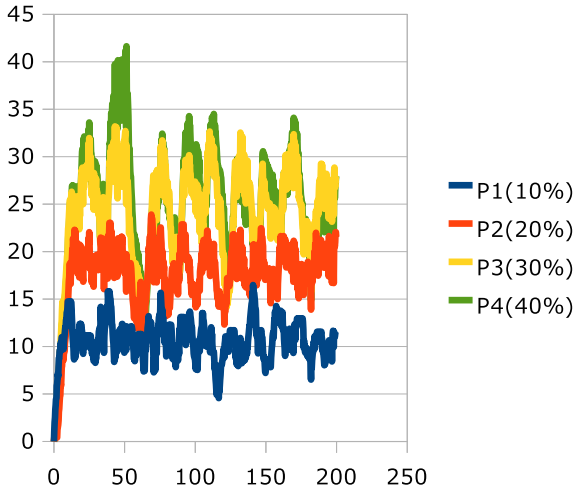

図 5.16: 100%に制御

表 5.8: 200 ミリ秒での平均

ドメイン	dom1				dom2		
制約 (%)	10	20	30	40	10	10	20
結果 (%)	10.97	17.69	23.72	26.96	11.57	10.99	18.23

表 5.9: 400 ミリ秒での平均

ドメイン	dom1				dom2		
制約 (%)	10	20	30	40	10	10	20
結果 (%)	10.92	17.77	23.63	25.16	11.5	11.76	19.81

5.10.3　二つのドメインで複数のプロセスを制御

二つのドメイン上で 7 つのプロセスを動かし，それぞれのプロセスに CPU 使用率の制限をかけた．制御の間隔を変化させ，制御の精度を調べた．

CPU 使用率の変化は，図 5.17 から図 5.21 になり，CPU 使用率の平均は，表 5.8 から表 5.12 のようになった．

制御の間隔が長くなるほど，制約が小さいプロセスの制御精度が悪くなった．

5.10.4　一つのドメイン上で一つのプロセスを制御

Linux

CPU の制限を付けずに，一つのプロセスを制御した．図 5.22 がその様子であり，表 5.13 が平均である．

制限せずに動かしたプロセスが 90.17% 使っているので，約 10%がオーバーヘッドだと思われる．

表 5.10: 600 ミリ秒での平均

ドメイン	dom1				dom2		
割合	10	20	30	40	10	10	20
平均	11.55	18.95	23.88	25.95	12.67	13.24	20.12

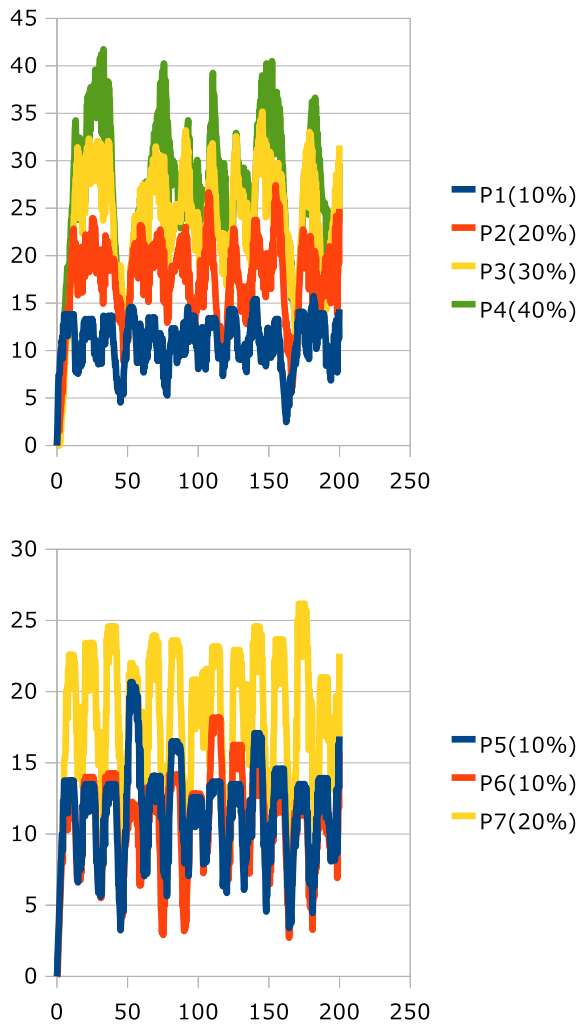

図 5.17: 200 ミリ秒間隔で制御

表 5.11: 1000 ミリ秒での平均

ドメイン	dom1				dom2		
割合	10	20	30	40	10	10	20
平均	13.14	19.05	23.79	24.67	13.17	14.5	21.62

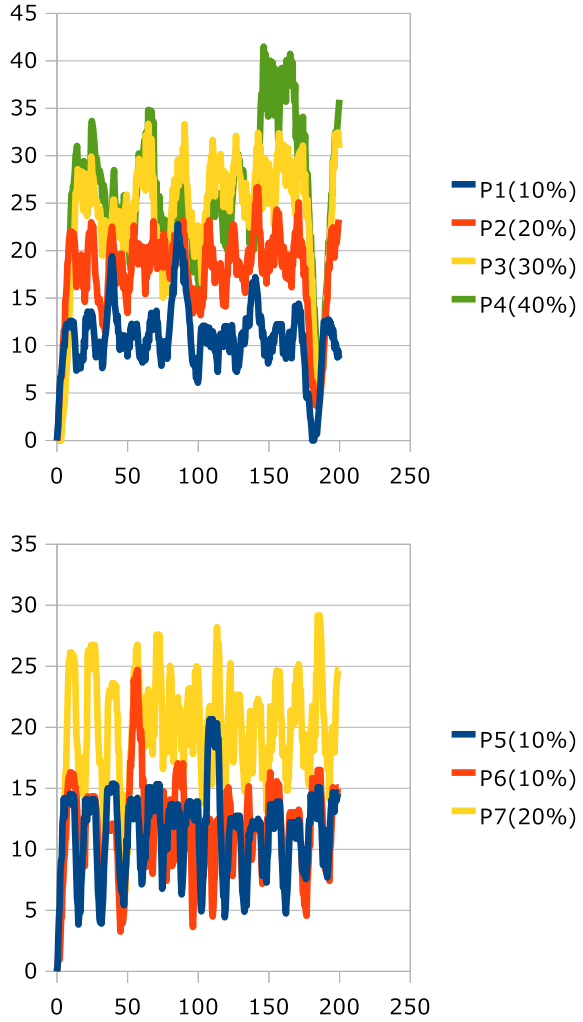

図 5.18: 400 ミリ秒間隔で制御

表 5.12: 2000 ミリ秒での平均

ドメイン	dom1				dom2		
割合	10	20	30	40	10	10	20
平均	14.16	17.71	21.18	20.75	14.15	17.23	23.19

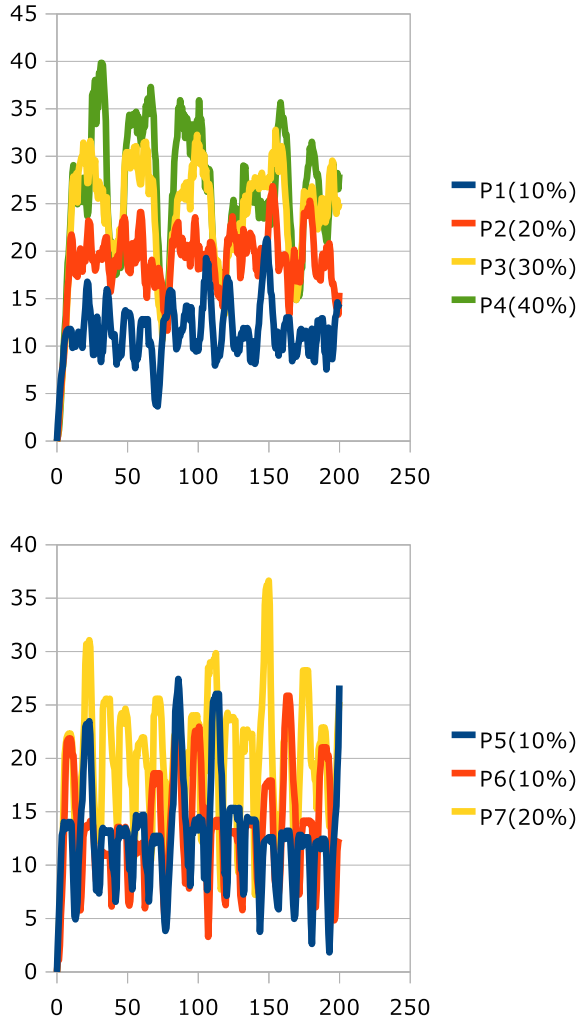

図 5.19: 600 ミリ秒間隔で制御

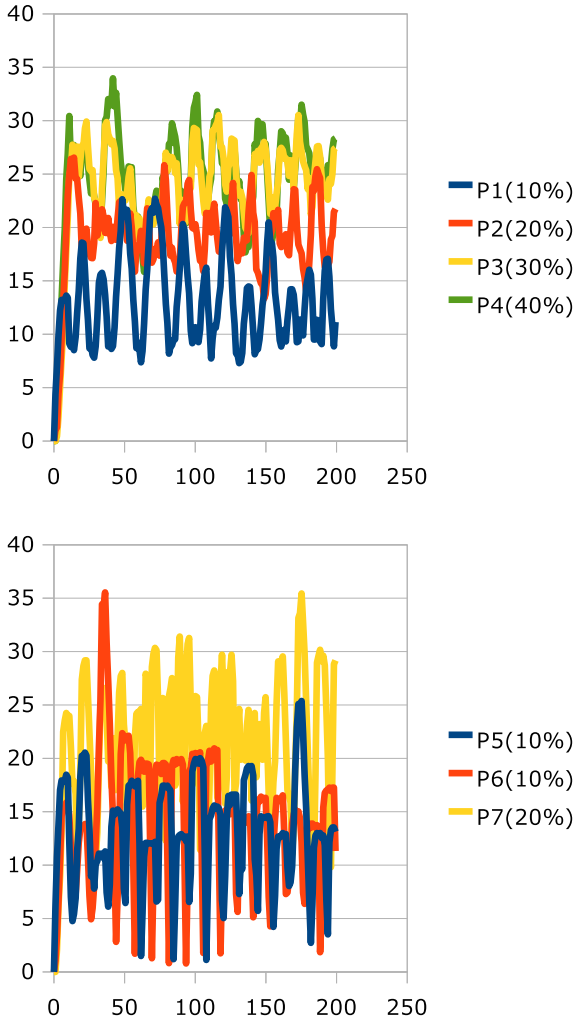

図 5.20: 1000 ミリ秒間隔で制御

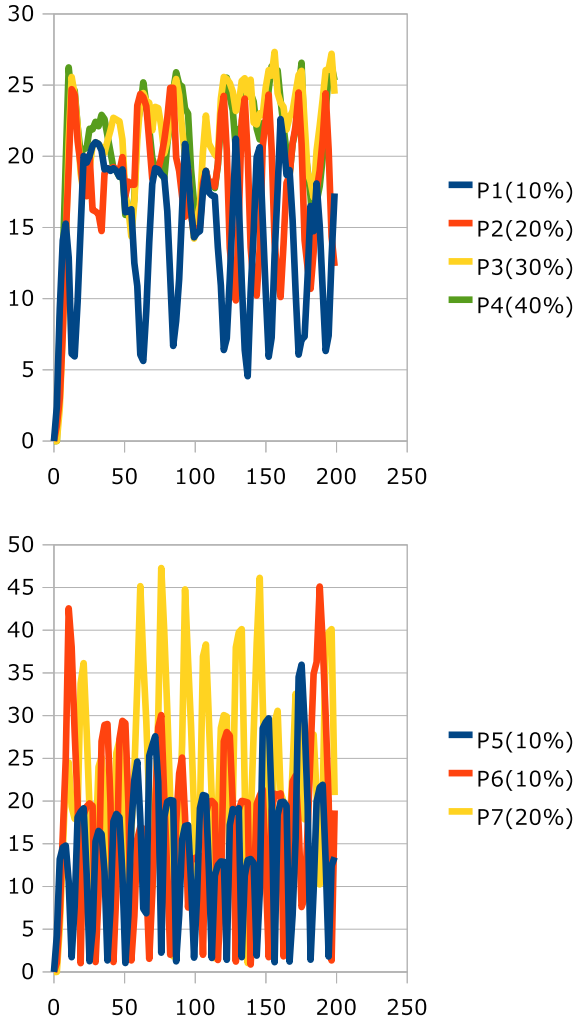

図 5.21: 2000 ミリ秒間隔で制御

表 5.13: Linux: 単独のプロセスを制限

CPU 使用率の制約 (%)	10	20	30	40	80	100	制限しない
結果 (%)	11.07	20.91	30.43	39.52	74.2	89.95	90.17

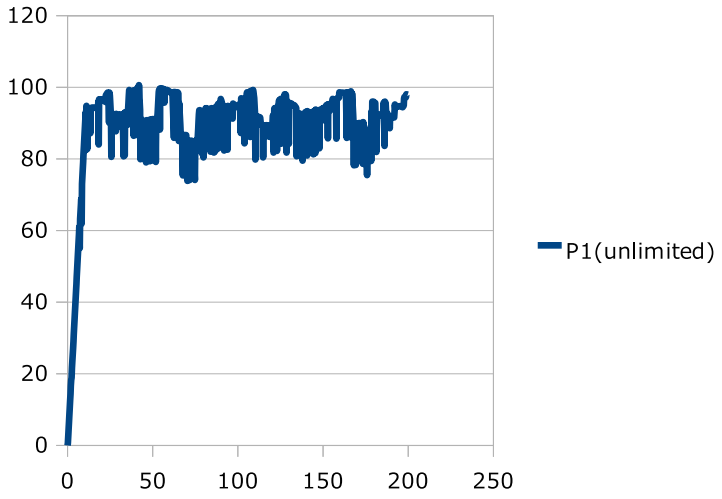

図 5.22: Linux: 単独のプロセスを制限せずに動かす

5.10.5　Windows

Windows ゲスト OS 上でプロセスを一つ動かし，CPU 使用率を制限するスケジューリングを行った．図 5.23 から図 5.26 がその様子である．それぞれの CPU 使用率の平均が，表 5.14 である．

Windows ゲスト OS 上では，単独で動かしたプロセスを制御できなかった．これは，カレントプロセスの制御ができないためだと思われる．プロセスを一つしか動かさない場合，そのプロセスがカレントプロセスになる可能性が高いからである．

表 5.14: Windows: 単独のプロセスを制限

CPU 使用率の制約 (%)	10	20	30	40
結果 (%)	36.19	42.73	36.62	48.21

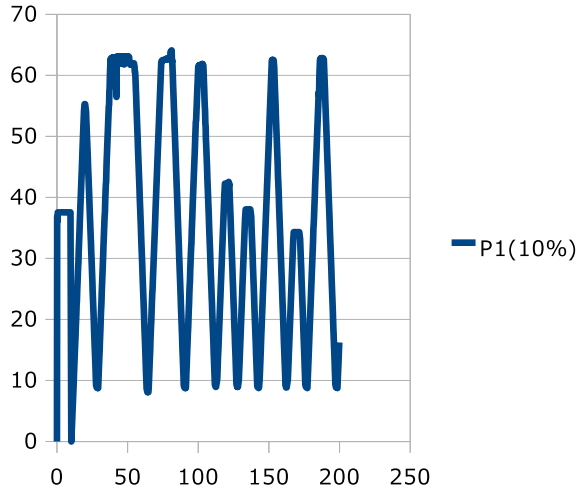

図 5.23: Windows: 単独のプロセスを 10%で動かす

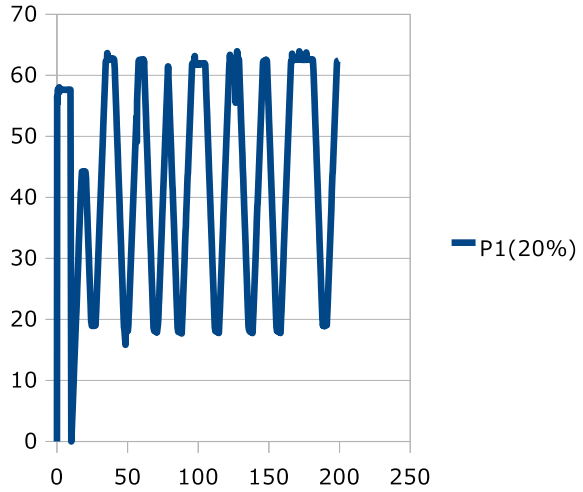

図 5.24: Windows: 単独のプロセスを 20%で動かす

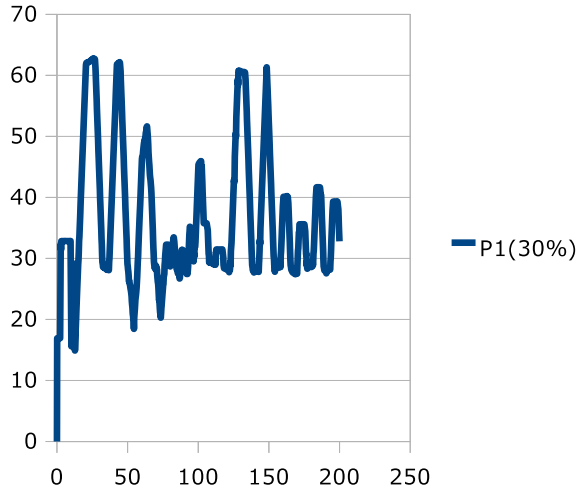

図 5.25: Windows: 単独のプロセスを 30%で動かす

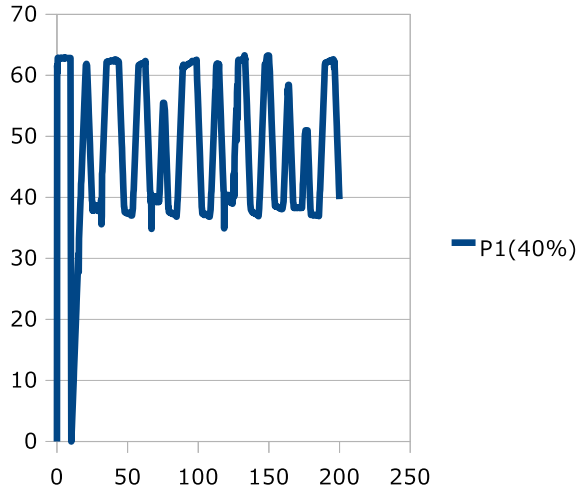

図 5.26: Windows: 単独のプロセスを 40%で動かす

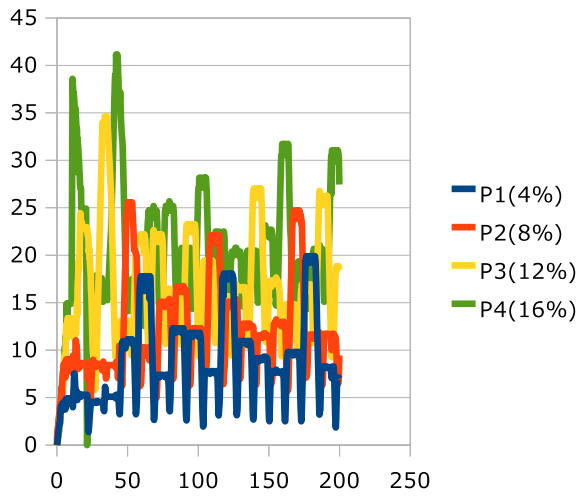

図 5.27: Windows: 制限の合計を 40%で動かす

5.10.6　一つのドメイン上で複数のプロセスを制御

Windows ゲスト OS 上で複数のプロセスを動かし，プロセスの CPU 使用率を制限するスケジューリングをおこなった．

プロセスの CPU 率の制限の合計を 40,60,80,100%で，スケジューリングをおこない，挙動を調べた．また，プロセスの CPU 率を制限しない場合も調べた．

制限した場合の結果は，図 5.27 から 図 5.30 である．CPU 使用率の平均は，表 5.15 から，表 5.18 である．制限しない場合は，図 5.31，表 5.19 のようになった．

すべてのプロセスの実際の CPU 使用率の合計が，60%で限界に達していた．指定した CPU 使用率の合計が 60%を越えた場合，CPU 使用率の割合が大きいプロセスでは，制御精度が悪くなった．

指定した CPU 使用率の合計が 60%を未満の場合，プロセスの制御の精度が悪くなっている．これは，カレントプロセスの制御ができないことが理由だと考えられる．

制限せずに動かした場合，170 秒付近ですべてのプロセスの CPU 使用率が減っている．原因は不明であるが，ドメインスケジューラが影響している可能性がある．

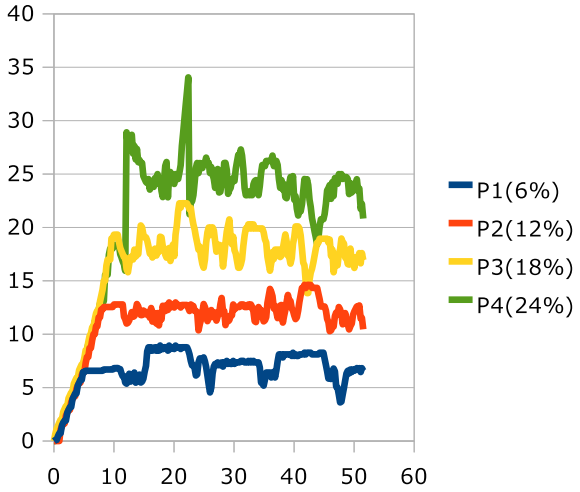

図 5.28: Windows: 制限の合計を 60%で動かす

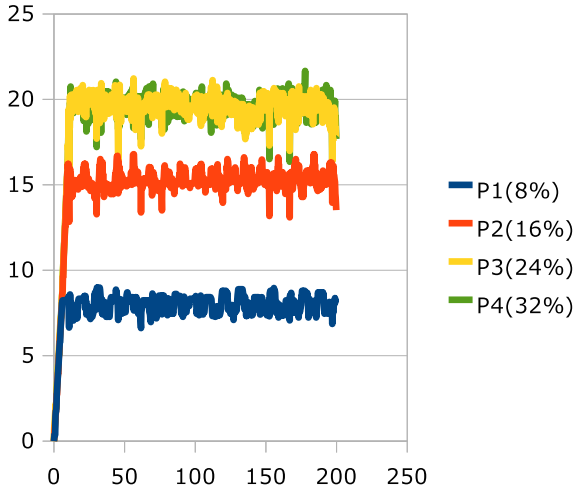

図 5.29: Windows: 制限の合計を 80%で動かす

表 5.15: Windows: 制約の合計が 40 パーセント

制約 (%)	4	8	12	16
CPU 使用率の平均 (%)	8.88	12.17	16.83	20.3

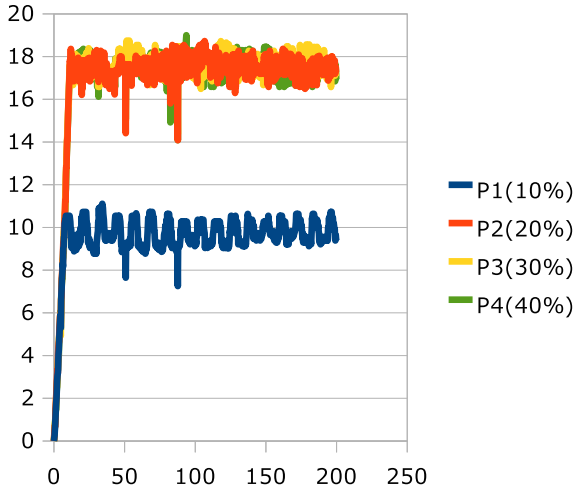

図 5.30: Windows: 制限の合計を 100%で動かす

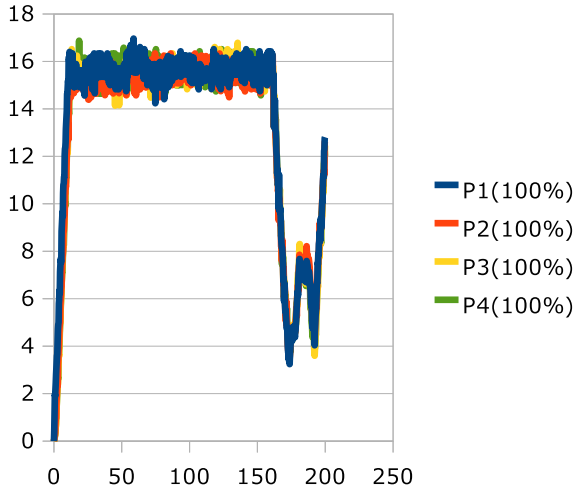

図 5.31: Windows: 制限せずに動かす

表 5.16: Windows: 制約の合計が 60 パーセント

制約 (%)	6	12	18	24
CPU 使用率の平均 (%)	6.74	11.24	16.45	21.08

表 5.17: Windows: 制約の合計が 80 パーセント

制約 (%)	8	16	24	32
CPU 使用率の平均 (%)	7.9	14.9	18.98	19.0

表 5.18: Windows: 制約の合計が 100 パーセント

制約 (%)	10	20	30	40
CPU 使用率の平均 (%)	9.57	16.99	17.12	17.0

5.10.7　プロセスの CPU 使用率を割合で指定

ドメインを二つ動かし，プロセス P1, P2 をドメイン U1 で動かし，プロセス P3, P4 をドメイン U2 で動かした．制約として，`4 * P1 <= P3`，`4 * P4 <= P2` とした．図 5.32 が，結果である．

ドメインを二つ動かし，プロセス P1, P2 をドメイン U1 で動かし，プロセス P3, P4 をドメイン U2 で動かした．さらに，ドメインの仮想 CPU を同じ物理 CPU に固定した．制約として，`4 * P1 <= P3`，`4 * P4 <= P2` とした．図 5.33 が，結果である．

ドメインを二つ動かし，プロセス P1, をドメイン U1 で動かし，プロセス P2 をドメイン U2 で動かした．さらに，ドメインの仮想 CPU を同じ物理 CPU に固定した．制約として，`2 * P1 <= P2` とした．図 5.34 が，結果である．

図 5.33 と図 5.34 において，一つの物理 CPU を共有しているにもかかわらず，CPU 使用率の合計が物理 CPU 一つ分をこえている．特に，CPU 使用率を制限していないプロセスはほぼ 100%動いていることになってしまう．

これは，4.3.7 節で述べた問題であり，一つの物理 CPU を複数の仮想マシンモニタが利用しようとする状況では，ゲスト OS 内の統計情報が不正確になると，考えられる．特に CPU 使用率を制限していないプロセスは，

表 5.19: Windows: 制約せずに動かす

制約 (%)	無し	無し	無し	無し
CPU 使用率の平均 (%)	13.66	13.42	13.5	13.49
150 秒付近までの CPU 使用率の平均 (%)	15.12	14.81	14.91	14.93

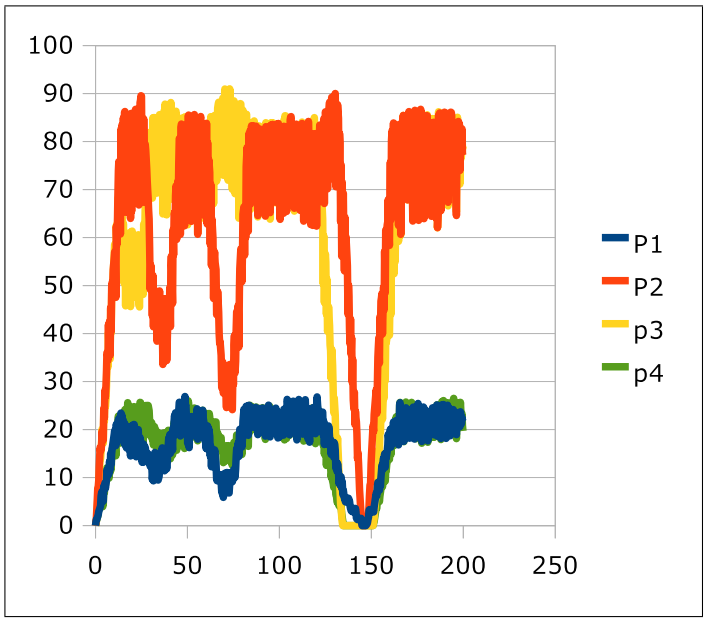

図 5.32: 割合で指定

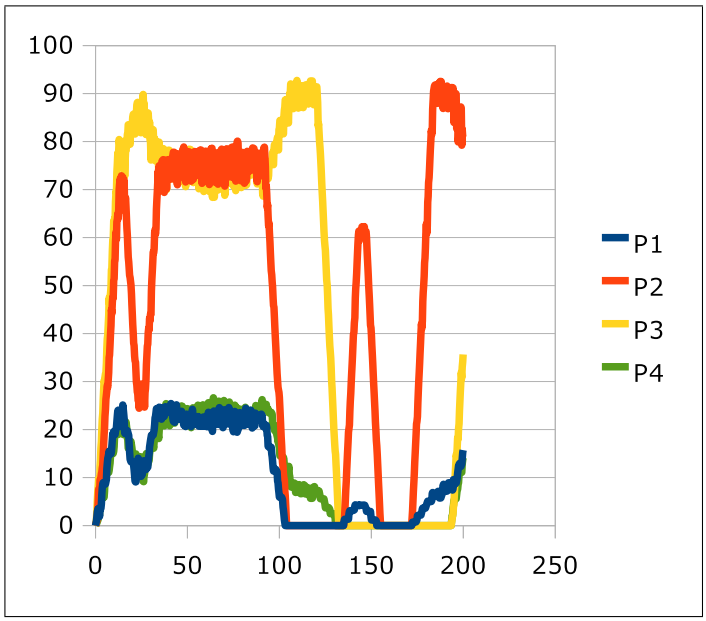

図 5.33: VM の VCPU を同じ物理 CPU に固定：割合で指定

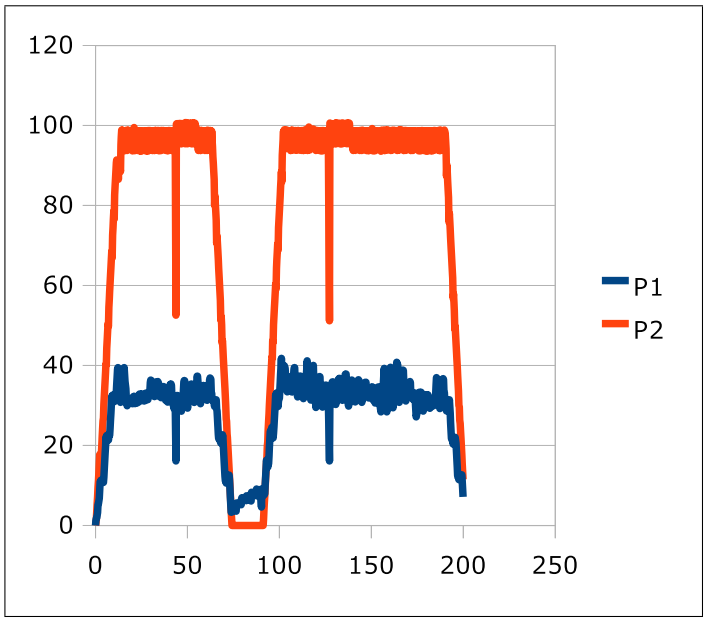

図 5.34: VM の VCPU を同じ物理 CPU に固定：割合で指定

そのゲスト OS で `running` 状態である可能性が高く，そのプロセスに関する統計情報が歪みやすい．

この制約のため，ゲスト OS の時間に関する統計情報を利用してプロセススケジューリングする場合，Monarch Scheduler では，一つの物理 CPU を複数の仮想マシンが利用する状況では，あまりうまくプロセススケジューリングできない．

5.11　web サーバを優先するための実用的なスケジューリング

優先度の低いプロセスを完全に止めてしまうと，ユーザにとって不便である．例えば，定期的にログを取るプログラムが止り続けてしまうのは問題であるし，ユーザがそのプロセスがハングしてしまったと勘違いする可能性がある．

そこで，特定のプロセスの優先度を下げつつも，そのプロセスを完全には止めないスケジューリングポリシーを実装し，実験をおこなった．ドメイン U1 で lighttpd と tripwire を動かし，ドメイン U2 で tripwire を動かた．このとき，lighttpd が動いているときには，tripwire があまり動かないようなスケジューリングをおこなった．

それぞれのドメインで tripwire が動いている間に，lighttpd に対してア

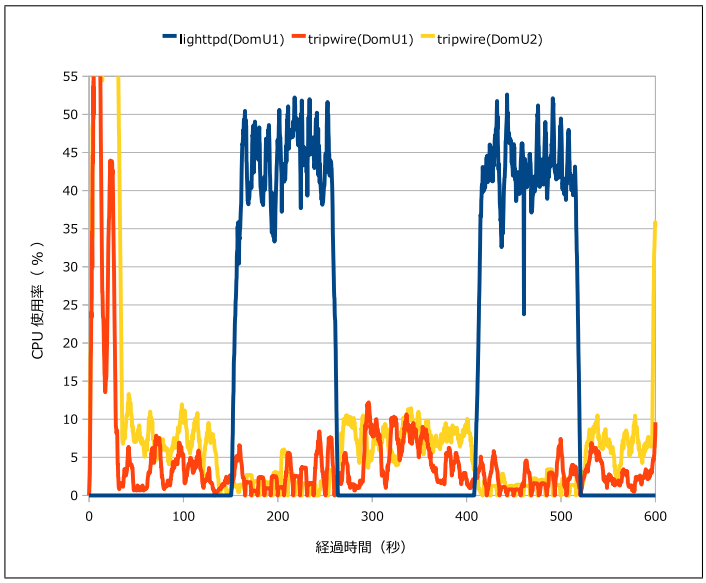

図 5.35: lighttpd を優先した場合の CPU 使用率の変化

表 5.20: lighttpd を優先した場合の性能

	lighttpd のスループット (#/sec)
lighttpd と tripwire を同時に動かす	8047.6
lighttpd を優先	8466.0

クセスし，lighttpd と tripwire の CPU 使用率と，lighttpd の性能を測定した．lighttpd へのアクセスは apache benchmark tool を用い， 900000 リクエストを 30 並列のアクセスを 150 秒の間隔を開けて 2 回繰替えした．

図 5.35 が CPU 使用率の結果である．lighttpd が CPU を使っているときは，tripwire は CPU を 5%以上使うことがほとんどなく，lighttpd が CPU を使っていないときは，tripwire は CPU を 5%から 10%の CPU を利用してる．このことから，lighttpd が動いているときは，tripwire の動きを制限できたことが確認できる．

また，lighttpd がリクエストを処理する性能は，表 5.20 のようになった．lighttpd と tripwire を同時に動かした場合より，提案システムを用いて lighttpd を優先したほうが，約 5%性能が向上したことがわかる．

第6章　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 を提案した．このシステムでは，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．

実験では，Monarch Scheduler を使い，仮想マシン間にまたがったスケジューリングを実装し，効果を確認した．具体的には，使用する CPU の割合を指定するプロポーショナルシェアスケジューリングや，web サーバを優先するスケジューリングを実装した．

参考文献

[1] Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H.: Information and control in gray-box systems, *Proceedings of the eighteenth ACM symposium on Operating systems principles*, Vol. 35 (2001).

[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, *Proceedings of the nineteenth ACM symposium on Operating systems principles*, Vol. 37 (2003).

[3] 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, *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, New York, NY, USA, ACM, pp. 164–177 (2003).

[4] bugcheck: GREPEXEC: Grepping Executive Objects from Pool Memory, `http://uninformed.org/?v=4&a=2&t=pdf`.

[5] Duda, K. J. and Cheriton, D. R.: Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler, *Proceedings of the seventeenth ACM symposium on Operating systems principles*, No. 33 (1999).

[6] Foundation, A. S.: ab - Apache HTTP server benchmarking tool, `http://httpd.apache.org/`.

[7] Free Software Foundation, Inc.: Computing billions of PI digits using GMP, `http://gmplib.org/pi-with-gmp.html`.

[8] Garfinkel, T. and Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection, *Proc. Network and Distributed Systems Security Symposium* (2003).

[9] iozone.org: IOzone Filesystem Benchmark, `http://www.iozone.org/`.

[10] Jan Kneschke: lighttpd, `http://www.lighttpd.net/`.

[11] 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*, New York, NY, USA, ACM, pp. 128–138 (2007).

[12] 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).

[13] Jones, S. T., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H.: Antfarm: Tracking processes in a virtual machine environment, *Proceedings of USENIX Annual Technical Conference* (2006).

[14] Jones, S. T., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H.: Geiger: monitoring the buffer cache in a virtual machine environment, *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (2006).

[15] 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).

[16] 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).

[17] Kim, G. H. and Spafford, E. H.: The design and implementation of tripwire: a file system integrity checker, *CCS '94: Proceedings of the 2nd ACM Conference on Computer and communications security*, New York, NY, USA, ACM, pp. 18–29 (1994).

[18] 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).

[19] Leslie, I. M., McAuley, D., Black, R., Roscoe, T., Barham, P. T., 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).

[20] Microsoft Corporation.: Debugging Tools for Windows, `http://www.microsoft.com/whdc/devtools/debugging/default.mspx`.

[21] Newhouse, T. and Pasquale, J.: ALPS: An Application-Level Proportional-Share Scheduler, *In Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing*, pp. 279–290 (2006).

[22] oldnewthing: Why are process and thread IDs multiples of four?, `http://blogs.msdn.com/oldnewthing/archive/2008/02/28/7925962.aspx`.

[23] Payne, B. D., Carbone, M. and Lee, W.: Secure and Flexible Monitoring of Virtual Machines, *Computer Security Applications Conference, Annual*, pp. 385–397 (2007).

[24] 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).

[25] Quynh, N. A. and Takefuji, Y.: A novel approach for a file-system integrity monitor tool of Xen virtual machine, *ASIACCS '07: Proceedings of the 2nd ACM symposium on Information, computer and communications security*, New York, NY, USA, ACM, pp. 194–202 (2007).

[26] Quynh, N. A. and Takefuji, Y.: Towards a tamper-resistant kernel rootkit detector, *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, New York, NY, USA, ACM, pp. 276–283 (2007).

[27] ReactOS Foundation: ReactOS, `http://www.reactos.org/`.

[28] 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`.

[29] VMware, Inc.: Improving Guest Operating System Accounting for Descheduled Virtual Machines in ESX Server 3.x Systems, `http://www.vmware.com/pdf/vi3_esx_vmdesched.pdf`.

[30] XenSource, Inc.: Credit-Based CPU Scheduler, `http://wiki.xensource.com/xenwiki/CreditScheduler`.

[31] 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).

[32] 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).