拡張コンポーネントのためのカーネルによる細粒度軽量保護ドメインの実現

品川高廣[†] 河野健二[†]
高橋雅彦[†] 益田隆司[†]

アプリケーションへの要求が多様化するにつれて，インターネットなどの開放性の高いネットワークから拡張コンポーネントを入手し，アプリケーションに組み込んで利用する形態が一般的になっている．本論文では，悪意のある拡張コンポーネントからローカルな計算機資源を保護するための細粒度保護ドメインの機構として，マルチプロテクションページテーブルという機構を提案する．マルチプロテクションページテーブルとは，従来のページテーブルを拡張し，同じ仮想アドレス空間内にある細粒度保護ドメインであっても互いに異なるページ保護モードを持たせることを可能にしたものである．マルチプロテクションページテーブルが既存のCPU上で実装できることを示すために，Intel x86アーキテクチャ上での実装を行った．また，この実装を用いた実験によって，1秒あたり10万回の保護ドメイン切替えを行っても，そのオーバヘッドが4.5～14.5%程度に抑えられることを確認した．

Kernel Support of Fine-grained Protection Domains for Extention Components

TAKAHIRO SHINAGAWA,[†] KENJI KONO,[†] MASAHIKO TAKAHASHI[†] and TAKASHI MASUDA[†]

It is commonplace to download an extension component from an open network such as Internet in order to supplement the functionality of an application. This paper proposes a protection mechanism, called a multi-protection page table, that protects the local computing resources from the attacks by malicious components. A multi-protection page table provides fine-grained protection domains, thereby enabling efficient cross-domain calls. To prove that a multi-protection page table can be implemented on top of stock hardware, this paper shows the implementation on the Intel x86 architectures. Experimental results show that our implementation incurs only a 4.5% to 14.5% execution overhead even if cross-domain calls occur 100,000 times per second.

1. はじめに

アプリケーションに対する要求が多様化するにつれて，拡張コンポーネントを利用してユーザが必要な機能を選んで自由に組み込める機構が提供されるようになってきた．ユーザはインターネットに代表される開放性の高いネットワークを通じて様々な拡張コンポーネントを入手し，必要に応じてアプリケーションの機能を拡張することができる．たとえば，Netscape社のWebブラウザであるNavigatorは，Plug-Inと呼ばれる拡張コンポーネントを組み込むことができ，実際に多くのユーザがインターネットからPlug-Inを入手して利用している．

しかし，開放性の高いネットワークはその匿名性が高く，入手した拡張コンポーネントの中にはユーザのパスワードを盗んだりするような悪意を持つものが存在している可能性がある．したがって，アプリケーションに組み込んで実行する際には，拡張コンポーネントの不正なアクセスからローカルの計算機資源を守るための保護ドメインが必要になる．アプリケーションと拡張コンポーネントは同じプロセス内で実行されるので，拡張コンポーネントからの不正アクセスを防ぐためにはプロセスの中により粒度の細かい保護ドメインを形成しなければならない．しかし，従来のオペレーティングシステムでカーネルが提供してきた保護ドメインは異なるプロセス間の保護を目的としたものであり，その粒度も比較的粗いものであった．

† 東京大学大学院理学系研究科情報科学専攻
Department of Information Science, Faculty of Science, University of Tokyo

本論文では拡張コンポーネントのためのカーネルによる細粒度保護ドメインの機構を提案する．この細粒度保護ドメインでは拡張コンポーネントによる（1）不正なメモリアクセス，（2）不正なシステムコールの発行を防ぐことができる．

不正なメモリアクセスを防止するために，本論文では**マルチプロテクションページテーブル**という機構を提案する．マルチプロテクションページテーブルとは，ページテーブルのそれぞれのエントリに対して従来は1つしか設定できなかった保護モードを複数同時に設定できるように拡張したものであり，ある時点では複数の保護モードのうち1つが有効である．マルチプロテクションページテーブルを利用すると，同じ仮想アドレス空間を共有しながらも細粒度保護ドメインごとに異なるページ保護モードを持たせることが可能になり，拡張コンポーネントによる不正なメモリアクセスを防止できる．また保護ドメインの切替え時にページテーブルを切り替える必要がなくなり，TLB フラッシュなどを避けて軽量な保護ドメイン切替えを提供できる．マルチプロテクションページテーブルが一般的なプロセッサ上で実装できることを示すため，本論文では Intel x86 アーキテクチャ上での実装例を示す．

また拡張コンポーネントが不正にシステムコールを発行してローカルの計算機資源にアクセスすることを防ぐため，保護ポリシーに違反したシステムコールの発行を検出できるようにしている．保護のポリシーはアプリケーションごとに決めることができる．

以下，2章で関連研究について述べる．3章でマルチプロテクションページテーブルに基づいた保護のモデルを説明し，4章でその実装を述べる．5章で性能分析を行い，6章で本論文をまとめる．

2. 関連研究

従来の保護ドメイン実現方式には，大きく分けるとオペレーティングシステムが提供する方式と，言語処理系で提供する方式の2通りがある．

2.1 オペレーティングシステムが提供する方式

オペレーティングシステムで提供する保護ドメインでは，CPU のメモリ管理機構を利用して粗粒度の保護ドメインを実現している．オペレーティングシステムが提供する保護ドメインの例として，仮想アドレス空間，リングプロテクション，単一仮想記憶におけるマルチプロセスの3種類の保護について述べる．

2.1.1 仮想アドレス空間による保護

UNIX などの従来のオペレーティングシステムでは，プロセスごとに private な仮想アドレス空間を割り当てている．通常，この仮想アドレス空間が保護ドメインとしての役割を果たしており，その目的はプロセスの誤動作が他の関係ないプロセスに影響しないようにすることである．

仮想アドレス空間による保護ドメインは，プロセス単位という比較的粒度の粗いものであり，保護ドメイン切替えにともなうオーバヘッドは小さいとはいえない．そのため，アプリケーションと拡張コンポーネント間のように，頻繁に保護ドメインを切り替えて通信を行うモジュール間での保護には利用しにくいとされている[1),2)]．

2.1.2 リングプロテクションによる保護

Multics[3)] などでは，リング状の階層的な保護を提供している．それぞれのリングが1つの保護ドメインに対応しており，内側のリングに対応する保護ドメインほど高い特権レベルを持っている．ある保護ドメインで実行中のプロセスは，外側のリングに対応する保護ドメインにはアクセスできるが，内側のリングに対応する保護ドメインにはゲートと呼ばれるエントリ・ポイントを介してのみアクセスできる．

リングプロテクションによって提供できる保護は階層的な保護に限られており，同じリングに属する複数のコンポーネントに対して保護を与えるものではない．拡張コンポーネントの保護では，アプリケーションと拡張コンポーネント間での保護だけでなく，拡張コンポーネント間での保護も行う必要があり，階層的な保護だけでは不十分である．そのため，拡張コンポーネントのための細粒度保護ドメインにはそのままでは利用しにくい．

2.1.3 単一仮想記憶におけるマルチプロセスの保護

Opal[4)] などの単一仮想記憶オペレーティングシステムでは，単一の仮想記憶内に複数のプロセスを配置し，それらのプロセス間でのデータ共有を容易にしつつ互いに保護を行っている．単一仮想記憶におけるプロセスの保護の特徴は，プロセスの保護ドメインと仮想アドレス空間とを分離し，保護とアドレッシングとを分離した点にある．

しかしながら，資源割当ての単位であるプロセスと保護ドメインとは分離されておらず，1つのプロセス内にさらに粒度の小さい保護ドメインを提供することは想定していない．

2.2 言語処理系が提供する方式

言語処理系が提供する保護ドメインでは，処理系が様々な手法を用いることによって言語レベルで細粒度保護ドメインを実現している．言語処理系による手法としては，仮想マシン方式，実行コード修正方式，証

明添付方式の3つの方式が主にあげられる．

2.2.1 仮想マシン方式

仮想マシン方式はコンパイラがソースコードをいったん仮想的な命令列である中間コードに変換し，仮想マシンを用いてその中間コードを1命令ずつ解釈しながら実行する方式である．実行時に仮想マシンがすべての命令をチェックすることができるので，かなり粒度の細かい保護ドメインを実現することが可能である．

たとえば，Sun Microsystems社のJava[5]は，Java言語をバイトコードと呼ばれる中間コードに変換してクラスファイルというファイルに格納する．クラスファイルには安全性を検証するのに必要な情報が含まれており，JVM（Java Virtual Machine）はクラスファイルからバイトコードを読み込む際に，未定義命令やメソッド外へのジャンプ命令などの不正な命令を検出することができる．実行時にはクラスへのアクセス権限や配列の境界チェックなどを行うことによって不正なアクセスを防止することができる．またローカルの計算機資源に対するアクセスは，セキュリティマネージャというクラスを経由することによって，APIの単位で可否を決定することができる．

しかし，仮想マシン方式は中間コードを1命令ずつ解釈しながら実行するので，その実行速度はネイティブコード（CPUが直接実行できるコード）に比べると劣る．JIT（just-in-time）コンパイラを用いてネイティブコードに変換しながら実行する技術も使われはじめているが，最近の研究でもネイティブコードに匹敵する性能は報告されていない[6]．

2.2.2 コード修正方式

コード修正方式は，実行コードの中に安全性を保証するためのコードを挿入する方式である．挿入されたコードは次に実行されるコードをチェックして，不正なアクセスをアプリケーションに通知したり，不正なアクセスをしたりしないように修正したりすることができる．

たとえば，SFI[7),8)]はメモリアクセスやジャンプを実行する命令の前に，アクセスしようとしているアドレスが一定の範囲に収まるよう強制的に修正する命令を挿入し，不正な領域へのアクセスを防ぐ方式である．この方式を効率よく実現するためには，ソースコードをもとにフロー解析を行って不要なチェックを取り除くなどの最適化が必要であり，SFIの利用を想定して改造したコンパイラを使用することが望ましい．

2.2.3 証明添付方式

証明添付方式は，実行コードがあらかじめ決められた保護ポリシーに従っていることを保証する証明を添付する方式である．コードを組み込む際に添付された証明が正しいかどうか検証することによって，実行時に不正なアクセスが行われないことを保証できる．

たとえば，proof-carrying code（PCC）[9),10)]はカーネルを拡張するコードを安全に実行するためのメカニズムで，カーネルはあらかじめ保護ポリシーを決めて公開しておき，実行コードを作成する側がその保護ポリシーに従っていることの証明を添付する方式である．カーネルは実行コードを組み込む際に添付された証明が正しいことを検証し，いったん安全性が確認されたらそのコードの実行時にチェックを行う必要がない．

2.3 まとめ

オペレーティングシステムが提供する保護ドメインは，CPUのメモリ管理機構を利用してハードウェアで不正なメモリアクセスを検出するため，実行時のオーバーヘッドが低く，特定の言語処理系に依存しないといった利点がある．一方でこの方式は，保護ドメイン間呼び出しに必要なコストが大きく，頻繁に通信を行うモジュール間の保護には適さない．言語処理系で提供する方式は，単一の仮想アドレス空間内に保護ドメインを形成するため，従来のオペレーティングシステムが提供する方式よりも，保護ドメイン間呼び出しを効率よく行うことができるといわれている[1),2)]．

しかし，これは従来のカーネルが提供してきた保護ドメインが細粒度保護ドメインとして利用されることを想定していなかったためであり，カーネルが細粒度保護ドメインのための抽象化を提供すれば十分な実行時性能が得られると考えられる．実際最近の研究によれば，従来の粗粒度保護ドメイン間の保護ドメイン切替えであっても，比較的効率よく実装できることが報告されている（LRPC[11)]，Spring[12),13)]，L4[14),15)]）．本論文では細粒度保護ドメインのための抽象化とその実装方式を提案し，カーネルでも細粒度保護ドメインを軽量に実現可能であることを示す．

3. マルチプロテクションページテーブルによる細粒度保護ドメイン

カーネルの提供する細粒度保護ドメインの機構として，我々はマルチプロテクションページテーブルという機構を提案している[16),17)]．本章ではマルチプロテクションページテーブルを用いた保護のモデルについて述べる．

3.1 メモリ保護の実現

マルチプロテクションページテーブルとは，従来のページテーブルを拡張して，ページテーブルの各エントリに複数の保護モードを持たせることを可能にした

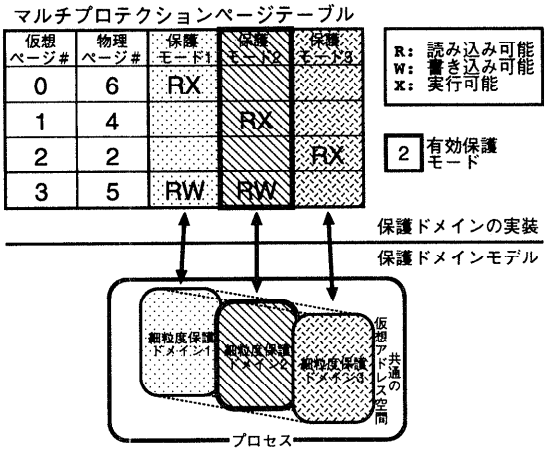


図1 マルチプロテクションページテーブル
Fig. 1 Multi-protection page table.

ものである．図1に示すように，それぞれのエントリには仮想アドレスから物理アドレスへのマッピングに加え，従来は1つであったページ保護モードが複数個記述できるようになっている．ページ保護モードの列はそれぞれ1つの細粒度保護ドメインに対応しており，細粒度保護ドメインごとにそれぞれ異なるページ保護モードを持たせることができる．複数の保護モードの列のうち，ある時点で有効な保護モードは1つの列である．細粒度保護ドメインを切り替えるには有効な保護モードの切替えを行う．図1では，保護ドメイン2に対応する保護モードが有効になっている．

この機構を用いると，アプリケーションと拡張コンポーネントとを別々の細粒度保護ドメインに入れ，拡張コンポーネントの不正メモリアクセスからアプリケーションを保護できる．たとえば図1では，仮想ページ0は保護ドメイン1からは読み込みと実行が可能であるが，それ以外の保護ドメインからは読み込み，書き込み，実行のいずれも不可に設定されており保護されている．

3.2 保護ドメインの切替え

マルチプロテクションページテーブルによる保護モデルの利点は，同じプロセスに属する保護ドメインでは，プロセス単位で割り当てられた計算機資源が共有できるという点にある．したがって，保護ドメイン切替えの際に，プロセス単位で割り当てられた計算機資源を切り替える必要がない．たとえば，それぞれの細粒度保護ドメインが同じ仮想アドレス空間を共有しているため，保護ドメイン切替えの際にページテーブルを切り替える必要はなく，TLBフラッシュにともなうオーバヘッドが回避できる．また，プロセスに割り当てられたタイムスライスも共有できるので，保護ドメイン切替えの際にスケジューリングを行う必要もない．この点に着目すると，細粒度保護ドメインの切替えは，従来のプロセス間切替えよりも軽量に実現することができ，保護ドメイン間呼び出しのコストを低減できることが期待できる．

3.3 システムコールの発行の規制

拡張コンポーネントがローカルの計算機資源に不正にアクセスできないようにするため，あらかじめ決めておいた保護ポリシーに反するシステムコールの発行を禁止する必要がある．本論文で述べる方式では，アプリケーションごとに保護ポリシーを設定できるようにし，保護ポリシーに照らしてシステムコール発行の可否を決定できるようにしている．

このような機構を提供するために，拡張コンポーネントがシステムコールを発行するとアプリケーションにアップコールがかかるようにしておく．アプリケーションでは，アップコールの際に呼び出されるコールバックルーチンを設定しておけば，そのルーチンの中でシステムコール発行の可否を決定できる．コールバックルーチンはアプリケーション内で実行される通常の手続きであり，引数としてシステムコールの種類やその引数が受け渡される．したがって，コールバックルーチンさえ記述すれば，拡張コンポーネントにはある特定のファイルのアクセスだけを許可するとか，ネットワーク上の特定のマシンへの接続のみ許可するといった保護ポリシーを実現できる．

4. 細粒度保護ドメインの実装

我々の提案するマルチプロテクションページテーブルは，最近のプロセッサであれば特別なハードウェアを必要とせずに実装できる手法であり，実際にSPARCプロセッサでの実装をすでに行った[17]．この実装は，SPARCを含めAlpha，MIPSなどの多くのプロセッサに搭載されているtag付TLBを活用した実装であり，原理的にはそれらのプロセッサにも同じ手法が適用できる．本論文ではマルチプロテクションページテーブルの汎用性を実証するために，tag付TLBを持たないプロセッサであるIntel x86アーキテクチャ☆での実装を行った．Intel x86は広く実用的に利用されているアーキテクチャであり，マルチプロテクションページテーブルのIntel x86上での実装方法を示すことの実用上の意義は大きい．

☆ 本論文ではIntel x86アーキテクチャとは386プロセッサ以降のプロテクトモードがサポートされたプロセッサを指す．

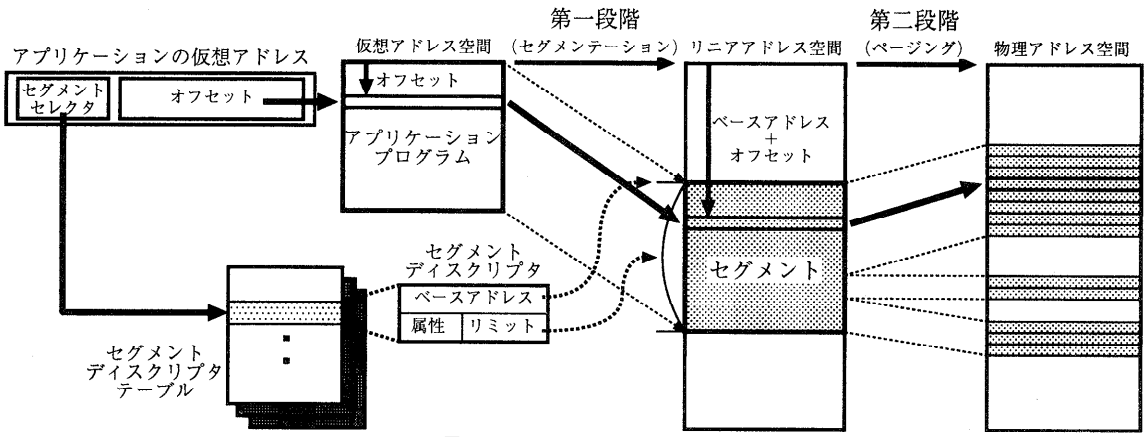


図2 セグメンテーション機構
Fig. 2 Segmentaion mechanism.

我々はIntel x86アーキテクチャのセグメンテーション機構を利用してマルチプロテクションページテーブルの実装を行った．さらにその最適化として，リングプロテクションを利用した実装も行った．以下でそれぞれの実装について説明する．また，3.3節で述べたシステムコールのチェック機構の実装についても説明する．

4.1 マルチプロテクションページテーブルの実装

4.1.1 セグメンテーション機構の概要

Intel x86アーキテクチャでは，プログラムから見える仮想アドレスは，16 bitのセグメントセレクタとセグメントに対する32 bitのオフセットという2つの数値の組から構成される．また，仮想アドレスから物理アドレスへのマッピングは2段階で行われる．この様子を**図2**に示す．

第一段階はセグメンテーション機構による**仮想アドレス**から**リニアアドレス**への変換である．セグメントセレクタはセグメントを指定する整数で，上位13 bitがセグメントディスクリプタテーブルのインデックスとなっている．セグメントディスクリプタには，セグメントのリニアアドレス空間での先頭アドレスを指定するベースアドレス，セグメントの大きさを指定するリミット，セグメントのアクセス権や存在bitなどの属性が記述される．仮想アドレスからリニアアドレスへの変換は，セグメントセレクタで指定するセグメントのベースアドレスにオフセットを加算することによって行われる（図2の第一段階のマッピング参照のこと）．

第二段階のマッピングは通常のページングによる**リニアアドレス**から**物理アドレス**への変換である．この変換では，ページテーブルに基づいてリニアアドレス空間のページを物理ページにマッピングする．リニアアドレス空間は仮想アドレス空間とは異なる概念で，ユーザプログラムからは見ることができない．

セグメントディスクリプタの属性にある存在bitによって，セグメントの有効/無効を指定できる．有効なセグメントはセグメントセレクタを指定することで，ユーザ権限であってもアクセスできる．一方，無効なセグメントにアクセスすると，セグメント不在例外が発生する．存在bitの切替えは特権モードでなければできないようになっており，ユーザ権限でセグメントの有効/無効を切り替えることはできない．

セグメント間の保護はハードウェアによって実現されており，セグメントリミットを越えてアクセスしたりアクセス許可のないセグメントにアクセスすると一般保護例外が発生する．

4.1.2 セグメンテーション機構による実装

セグメント間の保護機構を利用して細粒度保護ドメインを実現するために，保護ドメインごとに別々のセグメントを割り当てる．この様子を**図3**で示す．セグメントが異なると通常の仮想アドレスでは他の保護ドメインのメモリにアクセスすることはできない．しかし4.1.1項で述べたように，他のセグメントへのアクセスはユーザ権限でも可能なので，悪意のある拡張コンポーネントが他のセグメントのセグメントセレクタを指定して他の保護ドメインに不正にアクセスできるという問題がある．

この問題を回避するために，有効な保護ドメインに対応するセグメント以外のセグメントは無効にしておき，他の保護ドメインのセグメントにはアクセスできないようにしておく．4.1.1項で述べたように，ユーザ権限では他の保護ドメインのセグメントを有効にす

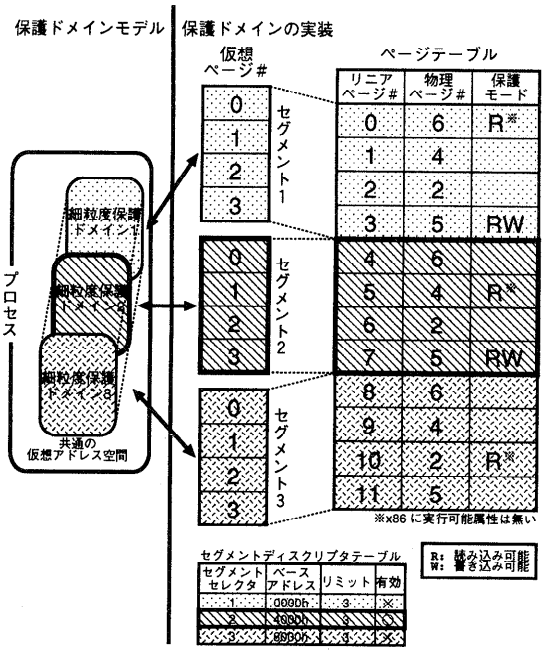


図3 マルチプロテクションページテーブルの実装
Fig. 3 Implementation of a multi-protection page table.

ることはできないので，これによってユーザ権限で保護ドメインの切替えはできなくなる．

保護ドメイン切替えを実行できるようにするために，保護ドメイン切替えを行うためのソフトウェア割込みを提供する．このソフトウェア割込みでは，セグメントディスクリプタの存在 bit を書き換えて，新しい保護ドメインに対応するセグメントを有効に，元の保護ドメインに対応するセグメントを無効に設定する．このときプロセス単位で割り当てられた計算機資源はいっさい切り替える必要はないため，軽量な保護ドメイン切替えが実現する．

マルチプロテクションページテーブルを実現するためには，同じ仮想ページであっても，セグメントごとに異なる保護モードを持てるようにする必要がある．これを可能にするために，それぞれのセグメントをリニアアドレス空間上の別々の領域にマップする．図3では，セグメント1から3がそれぞれリニアアドレス空間上の別の領域にマップされている．これによって，セグメントが違えば同じ仮想ページであってもそれぞれ別の保護モードを持つことができる．図3の仮想ページ0は，セグメント1では読み込み可能であるが，セグメント2では読み書き不可に設定されている．この実装の利点は，セグメントが違っても同じページテーブルが利用できるので，保護ドメインを切替えてもページテーブルを切り替える必要はなく，TLB フラッシュやキャッシュミスによる実行効率の低下を避けることができることである．

また，ポインタを含んだ構造を保護ドメイン間で共有できるように，仮想ページと物理ページのマッピングはすべてのセグメントで同じになるようにしておく．図3では，仮想ページ3上のデータをセグメント1とセグメント2で共有している．セグメントが違っても仮想アドレスと物理アドレスのマッピングは変わらないので，ポインタを含んだ構造でも自由に共有できる．

ただし，この方式では1つのプロセスあたりに組み込むことができる拡張コンポーネントの数が，利用可能なセグメントの数によって制限される．x86 アーキテクチャでは，1プロセスあたりに利用可能なセグメントの数は最大 8192 ($=2^{13}$) であり，実用上の制約にはならない数だといえる．しかし，セグメントをリニアアドレス空間内に配置するためには，セグメントの大きさに個数を掛け合わせた分の大きさのアドレス空間が必要になる．したがって，4 GB の大きさしかないリニアアドレス空間が不足しないように配慮する必要がある．

我々の実装では，拡張コンポーネントが占めるセグメントの大きさを小さくするために，アプリケーションの仮想ページのうちアクセス許可のないページは，拡張コンポーネントのセグメントには含めないようにしている．また，拡張コンポーネントと共有する仮想ページをアプリケーションのセグメントの後方に置くと，それにあわせて拡張コンポーネントのセグメントも大きくする必要があるので，共有する仮想ページはセグメントの先頭付近に置くようにしている．

我々の提案方式を含め，CPU のメモリ管理機構を利用した保護ドメインの粒度は仮想ページ単位である．オブジェクト単位などの仮想ページより粒度の小さい保護を行うには，強く型付けされた言語を用いるなど，コンパイラや実行時システムのサポートを併用して実現したほうが良い場合が多い．しかしながら，信頼できないコンポーネントどうしを分離するには，ハードウェアによる保護境界を設ける必要があり，CPU のメモリ管理機構を用いた保護ドメインが必要とされる．

4.2 リングプロテクションを利用した保護ドメイン間呼び出しの最適化

拡張コンポーネントどうしでの保護ドメイン間呼び出しに比べると，アプリケーションと拡張コンポーネント間での保護ドメイン間呼び出しの方がより頻繁に行われる．本節では，x86 アーキテクチャのリングプロテクション機構を利用して，アプリケーションと拡張コンポーネント間での保護ドメイン間呼び出しを最

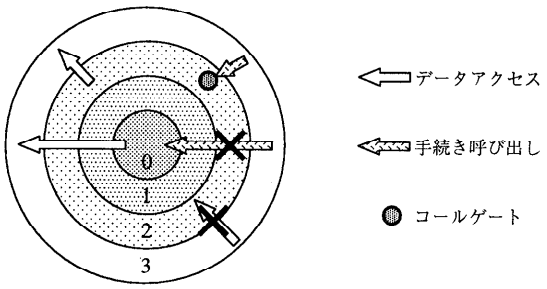


図4 リングプロテクション
Fig. 4 Ring protection.

適化する手法を述べる．

4.2.1 リングプロテクション機構の概要

Intel x86 アーキテクチャはリングプロテクションという保護機構を持っており，0，1，2，3の4段階の特権レベルが図4のような保護リングを構成している．セグメントごとに特権レベルが指定されており，実行中のプログラムの特権レベルはそのコードを含むセグメントの特権レベルとなる．特権レベルは0が最も高く，3が最も低い．実行中のプログラムは自分の特権レベルと同じか低い特権レベルのセグメントにはアクセスできるが，より高い特権レベルのセグメントにはアクセスできない．特権レベル0のプログラムは特権命令を実行することができる．通常は特権レベル0にはカーネルが置かれ，ユーザプログラムは特権レベル3で動作する．

システムコールを発行する場合など高い特権レベルのルーチンを呼び出すためにゲートという機構が用意されている．ゲートには呼び出すルーチンのセグメントとアドレスが記述されており，ゲートを呼び出すための命令を発行すれば，特権レベルを遷移してゲートで指定されたルーチンを呼び出すことができる．このときスタックの切替えも同時に行われ，必要であれば指定した数の引数を新しいスタックにコピーすることもできる．

4.2.2 保護ドメイン間呼び出しの最適化

リングプロテクションを用いた最適化の着眼点は，拡張コンポーネントの不正アクセスからアプリケーションを保護する必要はあるが，アプリケーションの不正アクセスから拡張コンポーネントを保護する必要はない，という点である．この点に着目すると，アプリケーションから拡張コンポーネントを呼び出す際に，4.1.2 項で述べたようにアプリケーションのセグメントを無効にする必要はなく，アプリケーションからの拡張コンポーネントの呼び出しを最適化できる．

アプリケーションから拡張コンポーネントのセグメントにアクセスできるのはかまわないが，拡張コンポーネントからはアプリケーションのセグメントにアクセスできないようにするため，拡張コンポーネントのセグメントは従来どおり特権レベル3に設定するが，アプリケーションの特権レベルを拡張コンポーネントより高い特権レベル2に設定する．これによって，拡張コンポーネントの不正アクセスからアプリケーションを保護しつつ，アプリケーションからの拡張コンポーネント呼び出しはセグメント間呼び出しの1命令で実現できる．特権レベル3から2に戻るにはゲートを呼び出す必要があるため，拡張コンポーネントからはアプリケーションの正しい領域に戻るようにできる．

なお，この最適化手法はアプリケーションと拡張コンポーネント間にのみ適用できるものである．したがって，この方法を用いた場合であっても，拡張コンポーネント間では，4.1.2 項で説明したようにソフトウェア割込みを利用して有効なセグメントの切替えを行う必要がある．

この手法の場合，アプリケーションの特権レベルを2にあげたことによって，カーネルが提供する従来どおりの保護が損なわれることがない点を確認しておく必要がある．特権レベル2と特権レベル3の違いは，I/O ポートの操作とページの書き込み保護に関する扱いにある．いわゆる特権命令は特権レベル0でのみ実行可能であり，特に問題とはならない．

I/O ポートの操作に関しては，I/O ポートを操作できる特権レベル（IOPL）を任意の特権レベルに設定できるため，IOPL が2に設定されているとアプリケーションが I/O ポートを操作できてしまう．しかし通常 IOPL は0に設定されており，IOPL を変更する命令は特権命令なので，これは問題とはならない．

ページの書き込み保護に関しては，Intel x86 アーキテクチャの初期の実装である 386 プロセッサでは，特権レベル 0，1，2 で動作するプログラムではページの書き込み保護ができなかった．しかし，486 プロセッサや Pentium プロセッサなどのそれ以降のプロセッサでは書き込み保護も行えるように変更されたので，これも問題とはならない．

以上から，アプリケーションを特権レベル2で動作させることに実用上の問題は生じないといえる．

4.3 システムコールのチェック機構の実装

3.3 節で述べたように，アプリケーションごとに定めた保護ポリシーに従って，システムコールの発行の可否を決定できる機構が必要である．

それぞれのアプリケーションは保護ポリシーを記述したコールバックルーチンをカーネルに登録しておく．

拡張コンポーネントがシステムコールを発行すると，カーネルは登録されたコールバックルーチンをアップコールする．コールバックルーチンには，拡張コンポーネントのセグメントセレクタ値とシステムコールの種類，およびその引数を受け渡す．コールバックルーチンはあらかじめ決められた保護ポリシーに照らし合わせて，そのシステムコールの発行を許可したり却下したり，あるいは引数を書き換えたうえで許可するなどの処理を行う．コールバックルーチンは通常の手続きであり，ケーパビリティやアクセスコントロールリストなどを利用した様々な保護ポリシーを実装できる．また，コールバックルーチンは拡張コンポーネントと同じ仮想アドレス空間で動作しているので，システムコールの引数がポインタを含むような場合でもアドレス空間を切り替えずにチェックできる．

Intel x86 アーキテクチャでは，システムコールのためにソフトウェア割込みを行うと呼び出し元のプログラムのセグメントセレクタ値がカーネル・スタックにコピーされる．拡張コンポーネントはアプリケーションとは異なるセグメントで動作しているので，カーネルは拡張コンポーネントからの呼び出しを識別できる．

この手法の特徴はシステムコールのインタフェースを変更していないため，バイナリレベルの互換性を提供している点である．これによって既存のライブラリを変更せずに，そのまま利用することができる．

この手法では拡張コンポーネントからシステムコールが発行されるたびにコールバックルーチンを呼び出すため，ユーザ空間とカーネル空間の切替えが増え，システムコールのオーバヘッドが増加する．しかし，コールバックルーチンの呼び出しは，マルチプロテクションページテーブルによる保護ドメイン間呼び出しになっており，システムコールの実行時間全体に占める割合は比較的小さい．またシステムコールのオーバヘッドを避けるため，システムコールを頻発しないように作成されたプログラムが多い．結果として，この手法によって生じたオーバヘッドは，アプリケーション全体の実行時間から比べればきわめて少ないといえる．

5. 性能分析

一般にカーネルで細粒度保護ドメインを提供すると，保護ドメイン切替えのオーバヘッドがきわめて大きくなるといわれている[1),2),7)]．しかし，本論文で我々が提案した方式では言語処理系による方式と比べ遜色のないオーバヘッドで細粒度保護ドメインを実現できる．このことを示すために，本章では我々の方式におけるオーバヘッドについての分析と実験を行い，言語処理系による方式との比較を行う．

ここでいうオーバヘッドとは，保護を行うことによって増加した実行時間の割合である．したがって，保護を行っていないプログラムの実行時間を T [秒]，保護を行った場合の実行時間の増加分を dT [秒] とすると，プログラム全体でのオーバヘッドは，次式のように表せる．

$$overhead \equiv \frac{dT}{T} \tag{1}$$

まず，我々の方式において保護を行った場合のオーバヘッドを表す式を導出する．保護を行っていないプログラムでは，拡張コンポーネントは通常の手続き呼び出しで呼び出されるのに対し，保護を行った場合のプログラムでは保護ドメイン間呼び出しによって呼び出される．保護ドメイン間呼び出しにかかる時間を t_d [秒/回]，通常の手続き呼び出しにかかる時間を t_p [秒/回] とすると，拡張コンポーネント呼び出し 1 回あたりの実行時間の増加分は $(t_d - t_p)$ [秒/回] である．したがって，拡張コンポーネント呼び出しの頻度を β [回/秒] とすれば，プログラム全体でのオーバヘッドは以下の式のようになる．

$$overhead = \frac{(t_d - t_p) \times \beta \times T}{T} \tag{2}$$

$$= (t_d - t_p) \times \beta \tag{3}$$

次に，保護を行ったことによるオーバヘッドを，CPU のクロックサイクル数から求められるようにするために，t_d および t_p をクロックサイクル数に置き換える．CPU のクロック周波数を $Clock$ [サイクル/秒]，保護ドメイン間呼び出しにかかるクロックサイクル数を c_d [サイクル]，通常の手続き呼び出しにかかるクロックサイクル数を c_p [サイクル] とすると，

$$t_d = \frac{c_d}{Clock},\ t_p = \frac{c_p}{Clock} \tag{4}$$

であるから，式 (3) は次式のように書き換えることができる．

$$overhead = (t_d - t_p) \times \beta$$

$$= \frac{c_d - c_p}{Clock} \times \beta \tag{5}$$

式 (5) の c_d および c_p を実測するために，保護ドメイン間呼び出しにかかるサイクル数と通常の手続き呼び出しにかかるサイクル数とを計測する実験を行った．ヌル手続きを保護ドメイン間呼び出しで呼び出した場合のサイクル数と，通常の手続き呼び出しを用いて呼び出した場合のサイクル数とを，Pentium のパフォーマンスカウンタ（Time-Stamp カウンタ）を用

表1 手続き呼び出しにかかるクロックサイクル数
Table 1 Clock cycles of the procedure call.

変数	方式	CPU	Cycle
c_d	セグメント方式	PentiumII	584
		Pentium	230
	特権レベル遷移方式	PentiumII	190
		Pentium	101
	Context Switch	Pentium	約5600
c_p	通常の方式	Pentium	3

いて計測した．表1に実測結果を示す．

実験に用いたマシンは，それぞれCPUとしてPentiumII 400 MHzおよびPentium 133 MHzを搭載した2台のPCである．OSはLinux 2.0.35を改造して我々の方式を組み込んだものである．4章で述べたように，我々はセグメンテーション機構のみを利用したセグメント方式と，リングプロテクションを利用した特権レベル遷移方式の2種類の方式を実装した．したがって，それぞれの方式について，PentiumおよびPentiumIIの2種類のCPU上で実験を行い，計4通りの組合せで保護ドメイン間呼び出しのクロックサイクル数を計測した．比較のために，従来のOSにおけるContext Switchを利用した場合の平均クロックサイクル数も計測した．

この測定結果を式(5)に代入し，保護ドメイン間呼び出しの頻度（β[回/秒]）を0～500,000まで変化させたときのオーバヘッドを求めたものを図5に示す．ただし“Segment”はセグメント方式，“Level2”は特権レベル遷移方式をそれぞれ示している．従来のOSにおけるContext Switchを利用した場合の平均オーバヘッドは“ContextSwitch”で示している．比較のために，言語処理系による方法の1つであるSFIを使用した場合のオーバヘッドを図中に“SFI”で示す．文献7)によれば，SFIではjump，store，loadの3種類の命令の前にチェックコードを挿入した場合で，平均21.8%のオーバヘッドがかかるという実験結果があり，ここではそれに従った．ただし，厳密には保護ドメイン間呼び出しの頻度に比例して，SFIのオーバヘッドも大きくなると予想される．

図5のグラフから，我々の方式では保護ドメイン間呼び出しの頻度に比例してオーバヘッドが大きくなっているが，保護ドメイン間呼び出しの頻度が1秒間に10万回になっても，保護のオーバヘッドは4.5～14.5%程度であり，SFIと比べても遜色のないパフォーマンスが出ていることが分かる．なお，参考データとして，Javaにおける保護ドメイン間呼び出しの頻度がおよそ1秒間に3万回という実験データがある[2]．

以上の分析では，CPUのクロック周波数が高くなると1秒間に実行できる命令数も増えるために，保護ドメイン切替えの頻度が同じであれば，クロック周波数が高いほど相対的に保護のオーバヘッドが小さく見える．クロック周波数に依存しない比較を行うために，式(5)の$\beta/Clock$を以下のように置き換える．

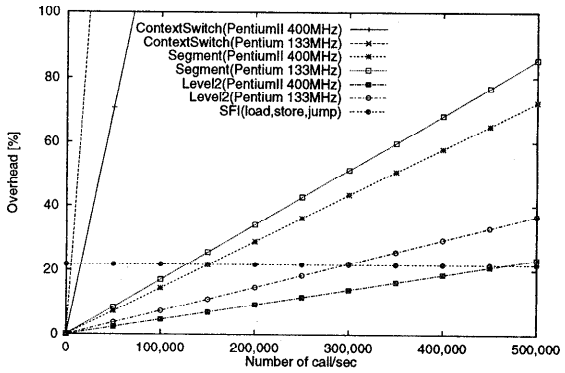


図5 保護ドメイン間呼び出しの頻度に対するオーバヘッド
Fig. 5 Overhead to frequency of the call between protection domain.

$$\frac{1}{Clock} \times \beta = \frac{t_p}{c_p} \times \beta \tag{6}$$

$$= \frac{\gamma}{c_p} \quad (\text{ただし } \gamma = t_p \times \beta) \tag{7}$$

ここでγという変数が持つ意味は，保護を行わないプログラム全体の実行時間のうち，拡張コンポーネントを呼び出す手続き呼び出しに費やす時間の割合（保護ドメイン間呼び出しの割合）[%]である．式(7)を式(5)に代入すると，次式のようになる．

$$overhead = \frac{c_d - c_p}{Clock} \times \beta$$

$$= \frac{c_d - c_p}{c_p} \times \gamma \tag{8}$$

式(8)のγを0～1[%]まで変化させたときのオーバヘッドを求めたものを図6に示す．クロック周波数に依存しない比較でも，保護ドメイン間呼び出しの割合が，セグメント方式で約0.1～0.3%程度，特権レベル遷移方式で0.35～0.7%程度までなら，SFIと同程度のオーバヘッドで実行できることが分かる．これは直感的にいえば，1命令を1サイクルで実行できると仮定して，セグメント方式で2000～670命令，特権レベル遷移方式で570～280命令に1回の頻度で保護ドメイン間呼び出しを行うことに相当する．

以上の結果から，本論文で提案した方式であれば，言語処理系による方式に比べ遜色のないオーバヘッドで細粒度保護ドメインを実現できるといえる．

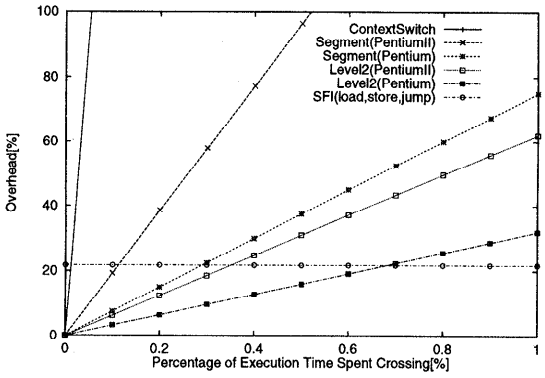


図6 保護ドメイン間呼び出しの割合に対するオーバヘッド
Fig. 6 Overhead to percentage of the call between protection domain.

6. まとめ

拡張コンポーネントのためのカーネルによる細粒度保護ドメインの機構として，マルチプロテクションページテーブルを提案し，Intel x86 アーキテクチャ上での実装方法を示した．マルチプロテクションページテーブルとは，従来のページテーブルを拡張し，各エントリに複数の保護モードを設定できるようにしたものである．マルチプロテクションページテーブルを用いると，1つのプロセス内に複数の細粒度保護ドメインを提供できるだけでなく，それらの保護ドメイン間で計算機資源を共有することにより，軽量な保護ドメイン間呼び出しを提供できる．この方法を用いると，1秒あたり10万回の保護ドメイン間呼び出しを行っても4.5〜14.5%程度にオーバヘッドを抑えることができる．

マルチプロテクションページテーブルは汎用的な保護機構であり，拡張コンポーネントの不正アクセスからの保護以外の目的にも応用できると考えられる．すでに我々の研究グループでは，分散環境での移動オブジェクトのための保護機構への応用を進めている．マルチプロテクションページテーブルの応用性および汎用性を示していくことが今後の課題である．

参考文献

1) Thron, T.: Programing languages for mobile code, *ACM Computing Surveys*, pp.29(3):213–239 (1997).
2) Wallanch, D.S., Balfanz, D., Dean, D. and Felten, E.W.: Extensible Security Architecture for Java, *Proc. 16th ACM Symposium on Operating System Principles*, pp.116–128 (1997).
3) Corbato, F.J. and Vyssotsky, V.A.: Introduction and Overview of the MULTICS System, *Proc. AFIPS Fall Joint Computer Conference*, pp.185–196 (1965).
4) Chase, J.S., Levy, H.M., Feeley, M.J. and Lazowska, E.D.: Sharing and Protection in a Single-Address-Space Operating System, *ACM Trans. Comuputer Systems*, Vol.12, No.4, pp.271–307 (1994).
5) Team, J., Gosling, J., Joy, B. and Steele, G.: *The Java [tm] Language Spectication*, Addison Wesley Longman (1996).
6) Adl-Tabatabai, A.-R., Cierniak, M., Lueh, G.-Y., Parikh, V.M. and Stichnoth, J.M.: Fast, Effective Code Generation in a Just-In-Time Java Compiler, *Proc. ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pp.280–290 (1998).
7) Wahbe, R., Lucco, S., Anderson, T.E. and Graham, S.L.: Efficient Software-Based Fault Isolation, *Proc. 14th ACM Symposium on Operating System Principles*, pp.203–216 (1993).
8) Adl-Tabatabai, A.-R., Langdale, G., Lucco, S. and Wahbe, R.: Efficient and Language-Independent Mobile Programs, *Proc. ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pp.127–136 (1996).
9) Necula, G.C. and Lee, P.: Safe Kernel Extensions without Runtime Checking, *Proc. 2nd USENIX symposium on Operating System Design and Implementation*, pp.229–243 (1996).
10) Necula, G.C.: Proof-Carrying Code, *Proc. 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.106–119 (1997).
11) Bershad, B.N., Anderson, T.E., Lanzowska, E.D. and Levy, H.M.: Lightweight Remote Procedure Call, *ACM Trans. Comuputer Systems*, Vol.8, No.1, pp.37–55 (1990).
12) Hamilton, G., Powell, M.L. and Mitchell, J.G.: A flexible base for distributed programing, *Proc. 14th ACM Symposium on Operating Systems Principles*, pp.69–79 (1993).
13) Hamilton, G. and Kougiouris, P.: The Spring nucleus: A microkernel for objects, *Proc. Summer 1993 USENIX Conference*, pp.147–159 (1993).
14) Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S. and Wolter, J.: The Performance of μ-Kernel-Based Systems, *Proc. 16th ACM Symposium on Operating System Principles*, pp.66–77 (1997).
15) Liedtke, J., Elphinstone, K., Schönberg, S., Härtig, H., Heiser, G., Islam, N. and Jaeger,

T.: Achieved IPC Performance, *6th Workshop on Hot Topics in Operating Systems*, pp.28–31 (1997).
16) 高橋雅彦，河野健二，益田隆司：OS とアプリケーションの連携による軽量保護ドメインの実現方式，情報処理学会研究会報告書，98-OS-78, pp.153–160 (1998).
17) Takahashi, M., Kono, K. and Masuda, T.: Efficient Kernel Support of Fine-Grained Protection Domains for Mobile Code, *Proc. IEEE 19th International Conference on Distributed Computing Systems* (1999). To appear.

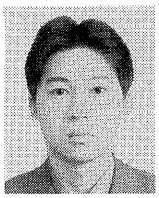
(平成 10 年 12 月 4 日受付)
(平成 11 年 4 月 1 日採録)

品川 高廣（学生会員）

1974 年生．1998 年東京大学工学部電子工学科卒業．現在，同大学大学院理学系研究科情報科学専攻修士課程在学中．オペレーティングシステム等のシステムソフトウェアに興味を持つ．

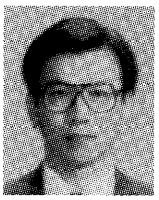
河野 健二（正会員）

1970 年生．1993 年東京大学理学部情報科学科卒業．1997 年同大学大学院理学系研究科情報科学専攻博士課程中退，同専攻助手に就任，現在に至る．オペレーティングおよびミドルウェア等のシステムソフトウェア，分散および並列処理に興味を持つ．IEEE/CS，ACM 各会員．



高橋 雅彦

1973 年生．1997 年京都大学工学部情報工学科卒業．1999 年東京大学大学院理学系研究科情報科学専攻修士課程修了．同年日本電気（株）に入社し，現在に至る．オペレーティング・システムに関する研究に従事．



益田 隆司（正会員）

1939 年生．1963 年東京大学工学部応用物理学科卒業．1965 年同大学大学院修士課程修了．同年（株）日立製作所入社．1977 年から筑波大学，1988 年 3 月から東京大学に勤務．現在，東京大学大学院理学系研究科情報科学専攻教授．専門はオペレーティングシステム．