Java向けヒープに対する不正参照のページ保護による検出

千 葉 雄 司†

Java 仮想機械にまつわる障害の原因の 1 つに，Java 仮想機械が管理するヒープに対する不正な書き換えがある．不正な書き換えの原因は大きく分けて 2 つある．1 つは Java 仮想機械の実現が含むバグであり，もう 1 つは Java で記述したプログラムから呼び出す，C 言語で記述したネイティブライブラリが含むバグである．Java 仮想機械を対象とした障害解析では，障害の原因が不正な書き換えである場合，まず最初に，書き換えの原因が Java 仮想機械の実現にあるのか，あるいはネイティブライブラリにあるのか判断し，バグを含むと推定したプログラムの開発部署に障害解析を依頼する必要がある．この判断は必ずしも容易なものではないが，判断を誤ると，障害解析の長期化といった問題が生じてしまう．そこで我々は，この問題の解決を目的として，ページ保護を活用して，スレッドがネイティブライブラリを実行している際には，Java 仮想機械が管理するヒープを参照不能にする機能を実装した．実装にあたっては，個々のスレッドごとに参照可能なメモリを設定するために，Java 仮想機械だけでなく，オペレーティングシステム（具体的には Linux）も改変した．実装した機能によれば，バグがネイティブライブラリ中にあるのか否か容易に推定可能になる．なぜなら，実装した機能を使って Java 仮想機械が管理するヒープを保護すれば，ネイティブライブラリ中にあるバグからの不正参照に際して，即座にページトラップが発生し，バグの所在が明かになるからである．実装した機能の実用性を検討するために，機能の利用にともなって生じるオーバヘッドの大きさを，実用的なアプリケーションを模したベンチマークである SPECjvm98 や SPECjbb2000 を使って評価した結果，相乗平均で 3.8%になることが分かった．

Detection of Invalid Reference to the Heap for Java Using Page Protection

Yuji Chiba†

Crashes of Java virtual machines (JVMs) often come from invalid memory reference to the heap for the JVMs. Because the bugs that cause the invalid memory reference exist in one of the following two places (1) the implementation of the JVMs, (2) the native library written in C language invoked from Java applications, system engineers who analyze the crashed JVM must first guess in which the bug exists and then ask the developer of the software guessed to contain the bug for debugging, but it is often not easy for system engineers to guess in which the bug exists. To solve this problem, we implemented a feature that use page protection to prohibit threads executing the native library from reference to the heap for the JVM. The feature helps system engineers to find where the bug in the native library is, because page trap happens on the invalid reference by the bug. The result of SPECjvm98 and SPECjbb2000 showed runtime overhead for using the feature is 3.8% in average.

1. はじめに

近年，ソフトウェアシステムは肥大化，複雑化する傾向にあり，1 つのプロセス内で動作するプログラムが，複数のプログラム製品から構成されていることも珍しくない．1 つのプロセス内で複数のプログラム製品が動作している状況においては，障害が発生した際に，どのプログラム製品が障害の原因となったか判別する作業が必要になるが，この作業は必ずしも容易でない．特に，障害の症状がメモリの不正な書き換えとしか分からず，しかも，破壊が発生してから長い時間がたった状況では，一体，どのプログラム製品の，どの部分が破壊の原因となったか推定するのが困難である場合が多い．このような状況では，往々にして破壊されたメモリ領域を管理するプログラム製品の開発元に原因調査の依頼が出されるが，この依頼が適切であるとは限らない．なぜなら破壊の原因となったプログラム製品が，破壊されたメモリ領域を管理するプログラム製品とは限らないからである．この問題を解決す

† 日立製作所システム開発研究所
Systems Development Laboratory, Hitachi, Ltd.

る手段の 1 つに，メモリ保護の機能を使う方法がある．すなわち，あるプログラム製品を実行中のスレッドが，別のプログラム製品の関数を呼び出すにあたり，呼び出しの前に，呼び出し元のプログラム製品に固有なメモリ領域への書き込みを禁止し，呼び出しから戻った時点で，書き込みを許可する．このとき，呼び出し先のバグが，呼び出し元に固有なメモリ領域を不正に書き換えようとすると，即座に割込みが発生する．この割込みを捕捉して，割込みを発生した命令がどこにあるか調べれば，バグの所在を突き止めることができる．

ただし，プログラム製品によっては，この方法を使ってデバッグできない場合もある．具体的には，Windows®☆や UNIX®☆☆系オペレーティングシステム（Operating System，OS と略記する）など，現在一般的な OS 向けのマルチスレッドアプリケーションには，この方法を適用しにくい．なぜなら，これらの OS はメモリに対する読み書き権限の設定をプロセス単位で実施し，プロセス内で動作する全スレッドに均一な読み書き権限を与えるので，特定のスレッドに特別な読み書き権限を与えることができないからである．たとえば，これらの OS 上で動作しているプロセスの内部で 2 つスレッドが動作しており，それらがともに，プログラム製品 A を実行している状況について考える．この場合，片方のスレッドが別のプログラム製品の関数を呼び出すからといって，プログラム製品 A 向けのメモリ領域を書き込み禁止にすることはできない．なぜなら，もう片方のスレッドが，プログラム製品 A を実行中であり，その実行の過程でプログラム製品 A 向けのメモリ領域へ書き込みうるからである．

この問題の解決策の 1 つに，メモリへの参照権限をプロセスごとではなく，スレッドごとに与え，スレッドが参照可能なメモリ領域を，実行中のプログラム製品に応じて制限する機能を使う方法がある．本論文では，この機能を，プログラム別メモリ保護と呼ぶことにする．プログラム別メモリ保護を使うと，マルチスレッドアプリケーションにおいても，ページ保護の機能を使って，プログラム製品にまたがるメモリ破壊を検出可能になる．

プログラム別メモリ保護は，これまでに，いくつかの OS で提供されてきた．しかしながら，現行の OS である Windows や UNIX 系 OS に，プログラム別メモリ保護を提供するものはない．したがって現状では，メモリ破壊の原因を究明する手段として，プログラム別メモリ保護を即座に活用しにくい．しかしながら現在コンピュータシステムがおかれた次の状況を考慮すると，プログラム別メモリ保護を実装する価値は十分にあるように考える．

- サーバが搭載するメモリ量の増大にともない，サーバ上で動作する個々のプロセスが占有するメモリ容量も数ギガバイトに達している．結果として，プロセスが 32 bit のアドレス空間を利用している場合，そのほとんどが有効なメモリ領域となり，結果として，バグによる不正なメモリの書き換えを，メモリ保護によって検知できる確率が低くなる．
  この問題を解決する手段として，プロセスが利用するアドレス空間を 64 bit に拡張することもできるが，64 bit のアドレシングを採用するとポインタのサイズが大きくなるなどオーバヘッドが生じてしまう．
- ソフトウェアシステムが複雑化し，1 つのプロセス内で動くプログラムを複数のプログラム製品が構成することが多くなった．結果として，メモリ破壊の発生時に，原因となったプログラム製品を特定する技術が強く求められるようになった．
  たとえばアプリケーションサーバのように規模の大きなプログラム製品では，製品としては 1 つとして販売されていても，製品を構成する個々の部品となるプログラムの開発元は別々に分かれている場合が多い．
  このようなプログラム製品が障害を起こした場合，まず，障害の原因となった「部品」のプログラムがどれか推定し，その開発元に障害原因の調査依頼を出すことになるが，メモリ破壊の障害では原因となったプログラムの推定が簡単でない場合が多い．往々にして破壊されたメモリ領域を管理するプログラム製品の開発元に調査依頼が出されるが，この依頼が適切であるとは限らない．
  アプリケーションサーバの例について考えると，アプリケーションサーバが占有するメモリ領域の多くを管理するのは Java™ ☆☆☆仮想機械（Java Virtual Machine，JavaVM と略記）であることが多く，したがって破壊されるメモリ領域が Java のヒープになる確率は小さくないが，Java のヒープの破壊が発生すると，その原因として，まずは

☆ Windows は，米国およびその他の国における米国 Microsoft Corporation の登録商標です．

☆☆ UNIX は，The Open Group がライセンスしている米国およびその他の国における登録商標です．

☆☆☆ Java および HotSpot は米国およびその他の国における米国 Sun Microsystems, Inc. の商標です．

JavaVM に疑いがかかり，JavaVM の開発元に調査依頼がゆくことがある☆．しかしながら真の原因は Java アプリケーションから呼び出した，C 言語で記述した関数（C 関数と略記する）の中にあるバグというケースも多く，このとき，原因調査の最初の依頼先を誤ったことから，障害解析に遅れが生じてしまう．

アプリケーションサーバは銀行や保険などミッションクリティカル分野にも適用されているプログラム製品である．そういったプログラム製品の障害解析では，迅速さが強く要求されることから，最初の調査依頼先を誤ることを原因とする初動の遅れはぜひとも回避したい．

- プログラム別メモリ保護を利用できる状況では，セキュリティ的により望ましいプログラミングが可能になる．ここでセキュリティ的により望ましいプログラミングとは，個々のスレッドに必要最低限の実行権限を与えることを表す．

  プログラム別メモリ保護を利用できる状況において，より望ましいプログラミングが可能になるプログラムの例に，JavaVM の実装の 1 つである HotSpotVM [21] がある．現在の HotSpotVM では，実行中に動的コンパイラのスレッド☆☆と，Java アプリケーションの実行を担当するスレッドが現れる．ここで HotSpotVM が管理するメモリ領域のうち，動的コンパイル済みコードを格納する部分は，動的コンパイラからの書き込みを受け付けるために，双方のスレッドから読み書き可能な状態になっているが，この状態は望ましくない．なぜなら Java アプリケーションの実行を担当するスレッドは，動的コンパイルコードを格納するメモリ領域に対する書き込み権限を保持すべきでないからである．Java アプリケーションの実行を担当するスレッドは，その実行中に動的コンパイルコードを読み込む必要はあっても，基本的に書き換える必要はなく，したがって書き込み権限を必要としない．セキュリティを考慮すれば不必要な権限は極力保持するべきでなく，さもないと Java アプリケーションの実行を担当するスレッドが攻撃にあった際に，動的コンパイル済みコードを書き換えてしまい，結果として攻撃者に制御を奪われるといった障害が発生しうる．この問題を回避する手段として，プログラム別メモリ保護を利用できる．具体的には，HotSpotVM の実装において，プログラム別メモリ保護を利用し，動的コンパイラのスレッドのみに，動的コンパイル済みコードを格納するメモリ領域に対する読み書き双方の権限を与え，Java アプリケーションの実行を担当するスレッドには，読み込み権限のみを与えるようにすればよい．

プログラム別メモリ保護は有用な機能だが，万能ではなく，その利用にあたっては注意を要する．なぜならプログラム別メモリ保護を利用すると，実行時オーバヘッドが発生し，プログラムの実行効率が劣化するからである．劣化の度合いによってはプログラム別メモリ保護を実用に供することが困難な場合もありえ，したがってプログラム別メモリ保護の利用を検討する際には，実行時オーバヘッドがどれだけ生じるかという見積りが重要になる．実行時オーバヘッドの主な発生源は，スレッドから参照可能なメモリ領域を切り替えるための処理にある．この処理はプログラム製品にまたがる関数呼び出しの際に実行する必要があるもので，したがってプログラム製品にまたがる関数呼び出しが頻発する状況ではプログラム別メモリ保護の利用が困難になりやすい．

本論文では HotSpotVM を対象として，プログラム別メモリ保護を使って次の機能を追加することが，実用に支障のない範囲の実行時オーバヘッドで実現できるか否か評価した結果を示す．

動的コンパイル済みコードの保護 動的コンパイル以外の処理を担当しているスレッドが，誤って，動的コンパイル済みコードを破壊することを防ぐ．

Java のヒープの保護 C 関数の実行を担当しているスレッドが，誤って，Java のヒープを破壊することを防ぐ☆☆☆．

これらの領域を保護する際にプログラム別メモリ保護が必要になる理由は，HotSpotVM がマルチスレッドアプリケーションであることから，一般的な OS が広く提供するメモリ保護機能（`mprotect()` など）のみを

☆ JavaVM 上で動作する Java アプリケーションには疑いがかからない．なぜなら，Java では言語仕様上，Java で記述したアプリケーションの実行によってヒープを破壊することは不可能だからである．たとえば Java アプリケーションの中で配列の大きさを超えた不正なメモリ参照を試みると，メモリ参照は行われずに，代わりに例外が発生する仕様になっている．

☆☆ バイトコード形式で表現されている Java アプリケーションを機種依存の機械コードにコンパイルするスレッド．

☆☆☆ C 関数の実行時に Java のヒープを保護しても，C 関数の実行に支障が生じることはない．なぜなら Java の言語仕様が，C 関数による Java のヒープへの直接参照を禁じているからである．C 関数と Java アプリケーションのやりとりは，Java が定義するインタフェース（Java Native Interface [17]，JNI と略記する）を通じて行う必要がある．

使って，これらの領域を保護することが困難だからである．評価にあたっては現行の OS である Linux™ ☆上にプログラム別メモリ保護の機能を実装した．実装はカーネルの変更をともなうものであり，したがって，このプログラム別メモリ保護の機能を利用して開発した HotSpotVM 向けヒープ保護機能は，即座に商用サーバで動作する HotSpotVM の障害対策に応用できるものではない．なぜなら Linux 向け商用アプリケーションは商用 Linux に対して動作保証を行うのが通例であり，改変したカーネルでは動作保証が得られないからである．ただし，開発した HotSpotVM 向けヒープ保護機能は，現状でも，Linux カーネルを改変する権限を有する HotSpotVM の開発者向けのデバッグツールとして活用できる．また，プログラム別メモリ保護の設計にあたっては，この機能を多くの OS に実装できるよう，特殊なハードウェアに依存せず，多くのプロセッサが提供する一般的なページ保護の機能のみあれば実装できるよう配慮した☆☆．もし将来，OS が公式にプログラム別メモリ保護に類する機能を提供すれば，HotSpotVM 向けヒープ保護機能の適用先を拡大し，たとえば商用サーバで動作する HotSpotVM の障害対策などに応用可能になる．HotSpotVM は多くのプラットフォームをサポートする一方で，不正メモリ参照はどのプラットフォームでも発生しうる．したがって HotSpotVM 向けヒープ保護機能をデバッグに活用するためには，より多くのプラットフォーム向けにこの機能を実現する必要があり，その意味からも，プログラム別メモリ保護を多くのプラットフォームに実現できるよう設計することには意義がある．なお，プログラム別メモリ保護は JavaVM のヒープの保護に特化した機能ではなく，プログラム製品に固有なヒープを保護する汎用的な機能であり，OS にこの機能を実装すれば，広範なプログラム製品のデバッグや障害解析を支援できる．ただし，プログラム別メモリ保護は，サンドボックスのようなセキュリティ機能の実装を支援するものではない．サンドボックスの実現においては，悪意のあるソフトウェアの活動を封じ込めるために，メモリ保護の機能を使う．サンドボックスのようなセキュリティ機能の実現に配慮したメモリ保護の実現[2)~4),9),13)] は，メモリ参照権限の変更を，適切な権限を保持するものにのみ許可する仕組みを備えるが，本論文で示すプログラム別メモリ保護は，この仕組みを持たない．したがって悪意のあるソフトウェアは，プログラム別メモリ保護による参照の制限を解除できる．

本論文ではまず，2 章で関連研究を示し，続く 3 章では我々によるプログラム別メモリ保護の実装について詳述する．4 章では HotSpotVM に施した，プログラム別メモリ保護を利用する機能拡張の実装について述べる．5 章ではプログラム別メモリ保護の実装および利用にともなう実行時オーバヘッドを評価した結果を示す．6 章は結論である．

2. 関 連 研 究

不正なメモリの書き換えは古くから問題視されており，その防止に向けて，数多くの対策が提案されてきた．過去に提案された対策は大きく次の 4 つに分類できる．

(1) プログラミング言語による対策
(2) 仮想マシンによる対策
(3) ハードウェアによる対策
(4) OS による対策

これら 4 つの対策について順次詳述し，プログラム別メモリ保護との比較を行う．

2.1 プログラミング言語による対策

アセンブリ言語や C 言語によるプログラミングでは，ポインタに任意のアドレスを指示させることが可能だが，その反面，ポインタの使い方を誤ると，メモリを不正に書き換えてしまう．これに対し，Java[1)] や C#[19)] といったプログラミング言語は，メモリを不正に書き換えるバグの撲滅を目的として，ポインタを提供せず，代わりに参照と，参照を通じたメモリの不正な読み書きを防ぐ機能を提供する．

これらの対策はもちろん有効だが，アセンブリ言語や C 言語などでプログラミングを行う必要がある場合向けの対策にはならない．一方でプログラム別メモリ保護は，特定のプログラミング言語に依存する機能ではなく，アセンブリ言語や C 言語で開発したプログラムにも適用できる．

2.2 仮想マシンによる対策

仮想マシン[7),22),29)] を使った不正メモリ参照の検出では，仮想マシンを使ってプログラムを解釈実行し，実行中に生じるメモリ参照が不正か否か監視する．仮想マシンを使う技法は，主にプログラムのデバッグの分野で利用されているが，実際に運用されているプログラム向けの監視技術としては必ずしも適切でない．なぜなら，仮想マシンを使わずに実行する場合に比べ，大きな実行時オーバヘッドが発生するからである．発

☆ Linux は Linus Torvalds の米国およびその他の国における登録商標または商標です．
☆☆ 5 章で示すように，特殊なハードウェアを利用可能な状況では，より効率的な実現が可能になる．

生する実行時オーバヘッドの大きさは仮想マシンの実装によって大きく異なるが，孝壽らの研究[29]によれば，プログラムが行う全ストアの動作を監視すると，プログラムの実行にかかる時間が，仮想マシンをインタプリタとして実装した場合には 231.2 倍，ダイナミックトランスレーションの技術を利用した場合でも 11.7 倍になる．

これに対し，プログラム別メモリ保護の利用にともなって生じる実行時間の増加は，5 章で示すように，実用上，最大でも 20%程度で済む．

2.3 ハードウェアによる対策

Dennis らの指摘にあるように[8]，マルチプログラミング環境にとって，あるプログラムの実行が，不正なメモリ参照などを通じて，他のプログラムの実行を妨げることがないようにする機能は必須のものである．この機能をどう実現すべきかという点について，ハードウェアおよびオペレーティングシステムの分野で長く議論されてきた．

現行のハードウェアは，不正なメモリ参照を防ぐ手段として，アーキテクチャによらず，おおむね次に示す 2 つの機能を提供する．

アドレス変換 アドレス変換機能は論理アドレスを物理アドレスに変換する役割を果たす．アドレス変換を使うと，OS 上で動作する個々のプログラムに，それぞれ固有の論理アドレス空間を与えることが可能になり，結果として，あるプログラムの実行により，別のプログラムから参照可能なメモリを破壊するといった障害を防ぐことができる．

ページ保護 ページ保護では，メモリを固定長の領域（ページ）に切り分け，それぞれの領域に対する読み書きの可否を制御可能にする．プログラム別メモリ保護は，ページ保護の機能によって実現できる．

過去には様々なハードウェアが提案された．たとえば IBM の System/38™ ☆では，ハードウェアレベルで，整数など，ポインタでない値を通じたメモリ参照を防ぐために，メモリ上の個々の値に，ポインタか否かを表すタグビットを付与した[10]．

2.4 OS による対策

オペレーティングシステムが提供する不正メモリ参照対策には，仮想メモリやページ保護がある．仮想メモリはプログラムの個々の実行に独立した論理アドレス空間を与えるもので，ハードウェアが提供するアドレス変換の機能を使って実現する．ページ保護は論理アドレス空間上の個々のページに対して読み込み可，書き込み不可といった権限の指定を可能にする．

Windows や UNIX 系 OS など，現在一般的な OS では，プログラムの実行をプロセスという単位で切り分け，プロセスごとに独立した論理アドレス空間を与え，論理アドレス空間に対する読み書き権限も，プロセス単位で管理する．結果として，プロセス内で動作しているスレッドは皆，論理アドレス空間に対して，同じ読み書き権限を保持することになる．これらの OS は，プログラム別メモリ保護，つまり，スレッドが参照可能なメモリ領域を，実行中のプログラム製品に応じて制限する機能を提供しない．

プログラム別メモリ保護に相当する機能は，過去には HYDRA[4),16),28)] や Opal[2)] といった OS で提供されてきた．現行の OS では i5/OS®が提供し[13)]，i5/OS 向け JavaVM の実装[14)] では，プログラム別メモリ保護を使ってヒープを保護できるはずだが，我々が調査した限りでは，本論文と同様に，保護にともなう実行時オーバヘッドの大きさを評価した論文は過去に存在しない．

Windows や UNIX 系 OS の上で，OS を改変することなく，プログラム別メモリ保護に相当する機能を実現するには，個々のプログラム製品の実行を個別のプロセスで行い，プログラム製品にまたがる関数呼び出しを，プロセス間通信を介して実現する必要がある．しかしながら，この実現方法は効率的とはいい難い．実際，Czajkowski らは，この実現方法によって MVM という JavaVM を実装し[5),6)]，Java アプリケーションとネイティブライブラリの実行を切り分けたが，Czajkowski らの試算によれば，すべての JNI 呼び出しをプロセス間通信を介して実行すると，ベンチマーク（SPECjvm98）の実行速度が最悪で 10 倍弱遅くなる[6)]．

3. プログラム別メモリ保護の実装

プログラム別メモリ保護の機能は，実装対象の OS である Linux に保護ドメインを導入することで実現した．本論文において，保護ドメインとは，スレッドが論理メモリ空間中の個々のアドレスに対して保持する読み書き権限を規定するものを意味する．OS はスレッドを必ず 1 つの保護ドメインに所属させ，スレッドが保持するメモリへの参照権限を，所属先の保護ドメインに応じて制御する．

3.1 IA32 における保護ドメインの実装

IA32[11)] 向け Linux においては，保護ドメインを，

☆ PowerPC，System/38 および i5/OS は，米国およびその他の国における International Business Machines Corporation の商標または登録商標です．

ページグローバルディレクトリとして実装できる．ページグローバルディレクトリは，仮想メモリの実現において，論理アドレスを物理アドレスに変換する過程で使用するアドレス変換テーブルの一種であり，ページの一種であるラージページへの参照権限を制御する手段としても利用できる．ここでは，まず，ページグローバルディレクトリの役割を明らかにするために，IA32 における仮想メモリの実現について詳述し☆，次に，ページグローバルディレクトリを使った保護ドメインの実装方法を示す．

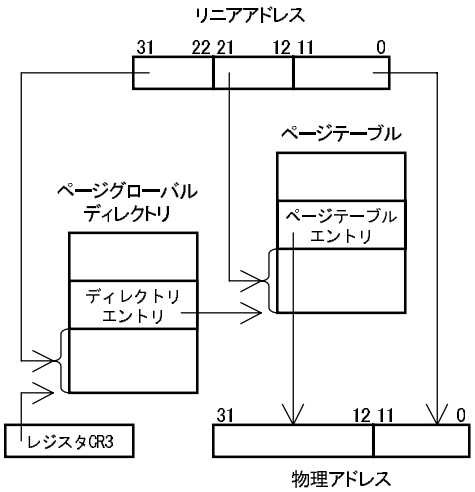
3.1.1 IA32 における仮想メモリの実現

IA32 では，論理アドレスから物理アドレスへの変換を，2 つの段階に分けて行う．具体的には，まず最初にセグメンテーションの機能によって物理アドレスをリニアアドレスに変換し，次にページングの機能によってリニアアドレスを物理アドレスに変換する．リニアアドレスから物理アドレスへの変換の手順を**図 1** に示す．

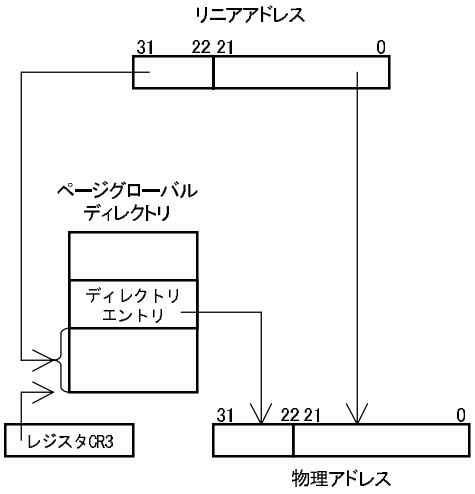
図 1 には，変換の手順を 2 種類記載してあるが，これはリニアアドレスに対応するページの大きさによって変換の手順が変化するためである．IA32 がサポートするページには，大きさが 4 KByte のものと，4 MByte のものがあり，後者を特にラージページと呼ぶ．

ページの大きさがどちらの場合であっても，リニアアドレスから物理アドレスへの変換の最初の手順は同じである．まず，レジスタ CR3 がページグローバルディレクトリというテーブルのアドレスを保持しているので，このアドレスに，リニアアドレスの上位 10 ビットをオフセットとして加え，得られたアドレスにあるエントリ（ディレクトリエントリ）を参照する．ディレクトリエントリは，リニアアドレスに対応するページの大きさを表すビットを含み，このビットが 0 の場合には，大きさが 4 KByte なので図 1 (a) の手順によって変換を実施し，1 の場合は 4 MByte なので図 1 (b) の手順によって変換を実施する．

図 1 (a) の手順では，ディレクトリエントリがページテーブルというテーブルのアドレスを保持しているので，このアドレスにオフセットとして，リニアアドレスの第 12〜21 ビットの値を加え，得られたアドレスにあるエントリ（ページテーブルエントリ）に，リニアアドレスに対応する物理ページのアドレスの上位 20 ビットが記録してあるので，これにリニアアドレスの下位 12 ビットを加えることで，リニアアドレスに対応する物理アドレスを求める．



(a) ページサイズが 4 KByte の場合



(b) ページサイズが 4 MByte の場合

図 1 リニアアドレスから物理アドレスへの変換
Fig. 1 Translation from linear address to physical address.

図 1 (b) の手順では，ディレクトリエントリに，リニアアドレスに対応する物理ページのアドレスの上位 10 ビットが記録してあるので，これにリニアアドレスの下位 22 ビットを加えることで，リニアアドレスに対応する物理アドレスを求める．

3.1.2 ページグローバルディレクトリによる保護ドメインの実装

ラージページに対応するディレクトリエントリには，物理ページのアドレスの上位 10 bit のほかにも，ページに対する読み書き権限を制御するためのビット列が含まれており，プロセッサはこのビット列を参照してページに対する読み書きの可否を判断し，不正な参照

☆ IA32 におけるページングの実装には，アドレス変換に使うテーブルが 2 段階のものと 3 段階のものがあるが，本論文では，評価の際に利用した 2 段階のものについて述べる．

表 1 保護ドメインの実装にあたって新規導入/機能追加したシステムコール
Table 1 System calls created or modified to implement protection domain.

新規導入	概要
`int createpd()`	保護ドメインを新規生成する.
`int setpd(int pd)`	スレッドの所属先保護ドメインを pd に変更する．返戻値は変更前まで所属していた保護ドメイン.
`int deletepd(int pd)`	保護ドメインを削除する.

機能追加	概要
`mprotect()`	フラグ PROT_PD_READ，PROT_PD_WRITE による保護ドメインに固有な読み書き権限の登録.

に対してページトラップを発生する．この機能を使うと，保護ドメインの実現が可能になる．具体的には，次のようにすればよい．

(1) 保護ドメインを生成する際に，生成する保護ドメインに対応するページグローバルディレクトリを 1 つ生成する☆.

(2) 保護ドメインが保持する読み書き権限を，ページグローバルディレクトリ中の，個々のラージページ向けディレクトリエントリが含む，読み書き権限を制御するためのビット列の値に反映する.

(3) タスクスイッチの際に，レジスタ CR3 の指示先を，次に実行するスレッドの所属先保護ドメインに対応するページグローバルディレクトリにする．また，システムコールを通じてスレッドの所属先保護ドメインを変更するよう要求された際にも，レジスタ CR3 の指示先を変更する.

3.2 導入したシステムコール

保護ドメインの導入にともなって新規導入あるいは改変したシステムコールの一覧を**表 1** に示す．表 1 に示したシステムコールを使って固有のヒープ領域を保護するプログラムのソースコードの例を**図 2** に示す．簡潔のため，図 2 のソースコードではエラー処理の記述を省略した．図 2 のソースコードを使って，表 1 のシステムコールの用途の具体例を示す.

図 2 のソースコードは 3 つの関数 `load_product()`，`public_service()`，`unload_product()` からなる．関数 `load_product()` はプログラムの初期化処理を担当し，まず 9 行目でシステムコール `createpd()` を呼んで保護ドメインを新規生成し，生成した保護ドメインに対応する番号を変数 `mypd` に格納する．保護ドメインの新規生成にともない，カーネル空間には，保護ドメインに対応するページグローバルディレクトリが割り付けられる．続く 11 行目では，ラージページを使うヒープ `myheap` を割り付け，13〜15 行目で `mypd` に対応する保護ドメイン（ページグローバルディレクトリ）にヒープ `myheap` に対する読み書き権限を登録する.

```
 1: // このプログラム向け保護ドメイン
 2: static int mypd;
 3: // このプログラムに固有なヒープ
 4: static void* myheap;
 5:
 6: // 初期化処理
 7: void load_product(){
 8:   // 保護ドメインの生成
 9:   mypd = createpd();
10:   // ラージページを使うヒープの割り付け
11:   myheap = large_page_heap_alloc(SIZE);
12:   // ヒープへの読み書き権限を保護ドメインに登録
13:   setpd(mypd);
14:   mprotect(myheap, SIZE,
15:            PROT_PD_READ | PROT_PD_WRITE);
16: }
17:
18: // プログラムが提供するサービス
19: void public_service(){
20:   // スレッドの所属先保護ドメインを mypd に変更
21:   int oldpd = setpd(mypd);
22:   {
23:     ···myheap を参照する処理 ···
24:   }
25:   // 所属先保護ドメインを oldpd に戻す
26:   setpd(oldpd);
27: }
28:
29: // 終了処理
30: void unload_product(){
31:   deletepd(mypd);
32: }
```

図 2 保護ドメインを利用するプログラム
Fig. 2 Program that use protection domain.

関数 `public_service()` はプログラムが提供するサービスに対応するもので，処理の過程でヒープ `myheap` を参照する．この関数の呼び出し元は同一のプログラムとは限らないことから，この関数では，まず入口の 21 行目で，この関数を実行するスレッドの

☆ ページテーブルについても，保護ドメインごとに作成すれば，保護ドメインによる読み書きの制御を 4 KByte のページごとに実施可能になるが，一方でページテーブルが消費するメモリが増大するといった問題が生じる．そこで本論文で評価に用いた保護ドメインの実装では，簡単のため，保護ドメインごとに作成する対象をページグローバルディレクトリのみとし，ページテーブルについては保護ドメイン間で共有するものとした.

所属先保護ドメインを mypd に変更するために，システムコール setpd() を呼び出す．システムコール setpd() は，レジスタ CR3 の指示先を，引数 mypd に対応する保護ドメインのページグローバルディレクトリに変更する．続く 22〜24 行目ではヒープ myheap を参照する処理を行い，最後に出口の 26 行目で再びシステムコール setpd() を呼び出して，スレッドの所属先保護ドメインを元に戻す．

なお，保護ドメインの変更を行う場所は，関数の呼び出し元であっても，呼び出し先であってもかまわない．関数 public_service() については，その出入口に保護ドメインを変更する処理がついており，したがって関数 public_service() の呼び出し元で保護ドメインを変更する必要はないが，出入口に保護ドメインを変更する処理がついていない関数を呼び出す場合などについては，呼び出し元で変更してもよい．

関数 unload_product() は，プログラムのアンロード時に呼び出される処理であり，システムコール deletepd() を呼び出して mypd に対応する保護ドメインを削除する．

3.3 システムコール setpd() の実装

表 1 に示したシステムコールのうち，setpd() はスレッドの所属先保護ドメインを変更するたびに呼び出す必要があることから，比較的，呼び出し頻度が高くなると予想でき，したがって効率的に実装する必要がある．

setpd() の教条的な実装を**図 3** に示す☆．図 3 のソースコードにおいて，setpd() は，まず 33 行目で，引数として受け取った新規所属先の保護ドメイン new_pd がカレントスレッド current の所属先保護ドメインと一致するか調べ，一致しないならば 36 行目に進んで，メモリ保護ドメインを表すデータ構造を検索する際に必要なロックを確保し，続く 37 行目で検索を行って，new_pd に対応する保護ドメインがあるか調べ，あるならば 39 行目で新規所属先保護ドメインに所属するスレッドの数に 1 を加え，さらに，旧所属先保護ドメインに所属するスレッドの数から 1 を減じ☆☆，47 行目でレジスタ CR3 に新規所属先保護ドメインのページグローバルディレクトリのアドレスを読み込み，48 行目でカレントスレッド current の所属先保護ドメインを記録するメンバ変数 pdid の値を new_pid に更新する．最後に 53 行目において，36 行

```
 1:  // 保護ドメインを表すデータ構造
 2:  struct pd_struct{
 3:    // ページグローバルディレクトリへの参照
 4:    pgd_t* pgd;
 5:    atomic_t users; // 所属するスレッド数
 6:  };
 7:  // 保護ドメインのクラスタ（二分木の節）
 8:  struct pd_cluster_struct{
 9:    struct pd_struct pd[PD_PER_CLUSTER];
10:    struct pd_cluster_struct* left;
11:    struct pd_cluster_struct* right;
12:  }
13:  // プロセスのメモリを管理するデータ構造
14:  struct mm_struct{
15:    ...
16:    // 保護ドメインクラスタ木の根
17:    struct pd_cluster_struct* pd_tree;
18:    // 保護ドメインクラスタ木参照時用ロック
19:    rw_lock_t pd_lock;
20:  };
21:  // スレッドを表すデータ構造
22:  struct task_struct{
23:    // 所属先プロセスの mm_struct への参照
24:    struct mm_struct* mm;
25:    ...
26:    int pdid; // 所属先保護ドメインの番号
27:  };
28:  // カレントスレッド
29:  extern struct task_struct* current;
30:
31:  long setpd(unsigned int new_pdid){
32:    long result = current->pdid;
33:    if (new_pdid != current->pdid){
34:      struct pd_struct *new_pd;
35:      pgd_t *new_pgd;
36:      read_lock(&(current->mm->pd_lock));
37:      if (new_pd = find_pd(new_pdid)){
38:        new_pgd = new_pd->pgd;
39:        if (new_pdid){
40:          atomic_inc(&(new_pd->users));
41:        }
42:        if (current->pdid){
43:          struct pd_struct *old_pd =
44:             find_pd(current->pdid);
45:          atomic_dec(&(old_pd->users));
46:        }
47:        load_cr3(new_pgd);
48:        current->pdid = new_pdid;
49:      }
50:      else{
51:        result = -EINVAL;
52:      }
53:      read_unlock(&(current->mm->pd_lock));
54:    }
55:    return (result);
56:  }
```

図 3 setpd() の教条的な実装
Fig. 3 Canonical implementation of setpd().

☆ 煩雑を避けるため，若干簡略化した．
☆☆ デフォルトの保護ドメイン（0 番）については，決して削除されないものと見なし，所属するスレッドの数を加減する処理を省略する．

目で確保したロックを解放する．

図 3 よりも高速に setpd() を実装する方法として，一度生成した保護ドメインの解放を，プロセスの終了まで諦める☆こともできる．この方法を採用すると，図 3 の setpd() の実装から，次の処理を省略できる．

(1) 36, 53 行目の処理：これらの処理では，setpd() の実行と並行して deletepd() が実行され，不正に保護ドメインが解放されることを防ぐために，排他制御を行うが，deletepd() のサポートをやめれば不要になる．
なお，createpd() については，その実装に配慮すれば，排他制御なしで setpd() と並行して実行可能になる．具体的には，setpd() ではプロセスのメモリを管理するデータ構造 mm_struct のメンバ変数 pd_tree が指示する木に，保護ドメインのクラスタにあたる構造体 pd_cluster_struct やページグローバルディレクトリといったノードを追加するが，この追加に先立って，ノードの初期化処理を行えばよい．初期化より先に追加を行うと，createpd() の実行と並行して setpd() を実行するプロセッサが，未初期化のページグローバルディレクトリなどを参照し，不正メモリ参照などの障害を起こしてしまう．

(2) 39〜46 行目の処理：これらの処理では保護ドメインを表すデータ構造 pd_struct のメンバ変数 users の値，つまり保護ドメインに所属するスレッドの数を管理している．管理を行う目的は，システムコール deletepd() において，削除の対象として指定された保護ドメインに，まだ所属しているスレッドがある（メンバ変数 users の値が 0 より大きい）際に，エラーを返すためである．まだスレッドが所属している保護ドメインを解放し，そのページグローバルディレクトリとして使用していたページを別の用途に使ってしまうと，不正に解放された保護ドメインに所属するスレッドが，不正なメモリの書き換えといった障害を起こす．しかしながら，deletepd() のサポートをやめれば，この障害は発生しなくなり，したがってメンバ変数 users の値を管理する処理も不要になる．

保護ドメインの解放を諦めると，メモリを無駄に消費する可能性はあるが，この無駄を軽減する方法はある．具体的には，ユーザプログラム側で，不要になった保護ドメインを管理，再利用すればよい．ユーザプログラム側では，より効率的に保護ドメインを管理できる．たとえば，4 章で述べる HotSpotVM 向け機能拡張の実装では，HotSpotVM が使う保護ドメイン（Java アプリケーションを実行する際に使う保護ドメインと，動的コンパイラが使う保護ドメイン）を再利用可能にするために，それぞれの保護ドメインに所属するスレッドの数を，setpd() のたびに更新する（図 3 の 39〜46 行目にある処理を HotSpotVM 側で実施する）必要はない．なぜなら，HotSpotVM が使う保護ドメインは，HotSpotVM の終了時に不要になると，あらかじめ分かっているからである．

☆ もしくは，Read-Copy Update [18] によるリストの更新と同様に，保護ドメインの解放を遅延する技法を使う．

4. HotSpotVM 向け機能拡張の実装

HotSpotVM には，Java のヒープおよび動的コンパイル済みコードの保護のため，次の機能拡張を施した．

保護ドメインの確保：HotSpotVM の起動処理に，新たに 2 つの保護ドメインを確保する処理を加えた．ここで確保する保護ドメインの一方を動的コンパイラ用，もう一方を Java アプリケーション用として使用する．また，C 関数を呼び出す際には，HotSpotVM が所属するプロセスが起動時に確保したデフォルトの保護ドメインを使用することにする．それぞれの保護ドメインを $PD_{compiler}$，PD_{java}，$PD_{default}$ と略記する．

メモリ参照権限の設定：HotSpotVM が管理するメモリ領域，つまり Java のヒープと動的コンパイル済みコード用メモリ領域に対して，個々の保護ドメインが保持する参照権限を**表 2** に示すとおりに設定する処理を加えた．

保護ドメインの切替え：個々のスレッドの所属先保護ドメインを，実行する処理（動的コンパイル，Java アプリケーション，C 関数）に応じて適切に設定するために，次に示す修正を加えた．

- HotSpotVM の実装のうち，Java アプリケーションと C 関数のやりとりを仲立ちする部分（JNI の実装）に次の処理を追加した．
  - Java アプリケーションから JNI を通じて C 関数を呼び出す際には，呼び出し前に，所属先保護ドメインを $PD_{default}$ に切り替え，呼び出し後に，所属先保護ドメインを PD_{java} に切り替える．
  - C 関数から JNI を通じて HotSpotVM に処理を依頼する際には，HotSpotVM が処理を受け付けた際に，呼び出し元に

表 2 個々の保護ドメインに与える参照権限
Table 2 Access rights for each protection domain.

保護ドメイン	Java のヒープ	動的コンパイル済みコード
$PD_{compiler}$	読み書き可能	読み書き可能
PD_{java}	読み書き可能	読み込み可能
$PD_{default}$	読み込み可能	読み込み可能

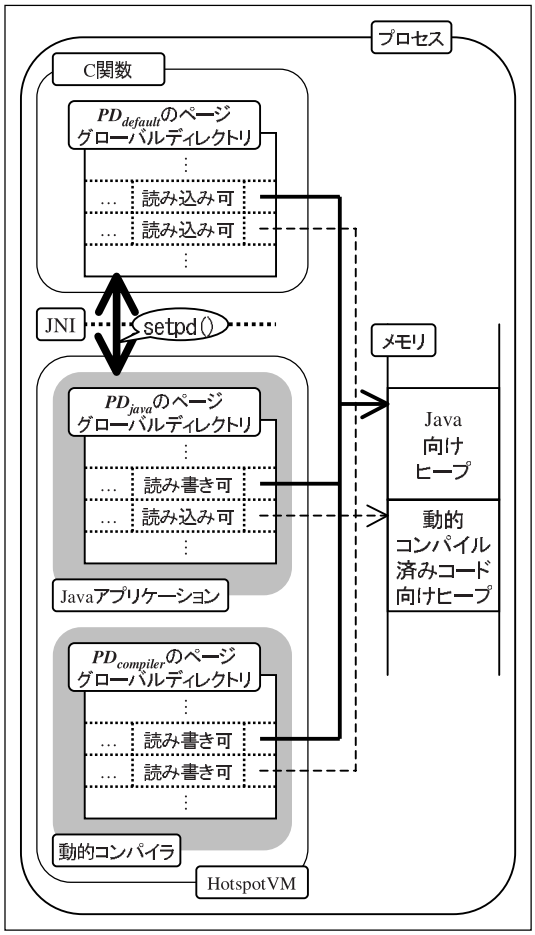


図 4 ヒープを保護する HotSpotVM
Fig. 4 HotSpotVM with protection for heap.

おける所属先保護ドメイン PD_{caller} を記録したうえで，所属先保護ドメインを PD_{java} に切り替える．依頼された処理が終了して HotSpotVM から戻る際には，所属先保護ドメインを PD_{caller} に戻す．

ただし，依頼内容がフィールドからの値の読み込みである場合には，実行時オーバヘッドを軽減するために，切替えを省略する．フィールドから値を読み込むだけならば，Java のヒープへの書き込みが発生しないので，所属先保護ドメインを切り替えずに実行できる．

- HotSpotVM の実装のうち，動的コンパイラのスレッドが起動時に実行する部分に，所属先保護ドメインを $PD_{compiler}$ に切り替える処理を追加した．

これらの機能拡張を通じてヒープを保護する HotSpotVM の実行イメージを**図 4** に示す．図 4 では，保護ドメイン $PD_{compiler}$，PD_{java}，$PD_{default}$ に対応して，それぞれのページグローバルディレクトリを確保しており，さらに，それぞれの Java のヒープや動的コンパイル済みコードに対応するディレクトリエントリの読み書き権限を表 2 のとおりに設定し，このことによって，Java のヒープや動的コンパイル済みコードの保護を実現している．

5. 評　　価

プログラム別メモリ保護の利用にともなう実行時オーバヘッドの計測を目的として，プログラム別メモリ保護の機能を提供する Linux と，プログラム別メモリ保護の機能を利用する HotSpotVM を実装し，評価を行った．本章では，まず，5.1 節で評価方法を示し，次に，5.2 節で評価結果を示す．最後に，5.3 節で実行時オーバヘッドの発生源について考察する．

5.1 評 価 方 法

評価は，プログラム別保護ドメインを実装する前後の Linux/HotSpotVM の上でベンチマークを実行し，それぞれの実行速度を比較することで行った．使用したベンチマークは SPECjvm98[24] および SPECjbb2000[25] である．SPECjvm98 は**表 3** に示す 7 つの実用的アプリケーションからなるベンチマークであり，SPECjbb2000 はサーバサイドにおけるビジネスロジックの処理能力を計測する．

実装のベースとして，また，実行速度の比較対象として使った Linux は，カーネルのバージョンが 2.6.14 のものであり，HotSpotVM は Sun Microsystems が jdk1.5.0_06 として配布しているものに独自のパッチをあてたものである．HotSpotVM にあてたパッチとは，動的コンパイル済みコードをラージページに配置するためのものである．Sun Microsystems が配布している Linux 向け jdk1.5.0_06 では，オプション `-XX:+UseLargePages` を指定すると Java ヒープをラージページ上に確保するが，パッチの適用により，動的コンパイル済みコード向けのヒープもラージページ上に確保するようにした．パッチの適用により，ベンチマークのスコアは若干向上する．

表 3 SPECjvm98 が含むベンチマーク項目
Table 3 Benchmark items in SPECjvm98.

ベンチマーク項目名	内容
_201_compress	テキストファイルの圧縮伸張
_202_jess	エキスパートシステムによる推論
_209_db	データベース処理
_213_javac	Java ソースコード向けコンパイラ
_222_mpegaudio	MP3 データの伸張
_227_mtrt	マルチスレッドによるレイトレース
_228_jack	構文解析器生成系

ベンチマークの実行にあたっては，ハードウェアとして PC（CPU: Intel Xeon®☆1.6 GHz × 2，主記憶 2 GByte）を利用した．また，HotSpotVM に対して実行時オプションを通じてヒープサイズなどに関する指示を与えた．ヒープサイズについては，初期サイズ，最大サイズともに SPECjvm98 では 512 MByte，SPECjbb2000 では 1 GByte とした．他には次の実行時オプションを指定した．

`-server` サーバ VM を使用する．HotSpotVM は 2 種類の VM からなり，一方をクライアント VM，もう一方をサーバ VM と呼ぶ．前者と後者の違いは利用する動的コンパイラにあり，後者向けの動的コンパイラの方が，コンパイルにより長い時間をかける代わりに，多くの最適化を適用し，より実行性能に優れたコードの生成を試みる．このオプションを指定した理由は，より実行性能に優れたコードを使って評価を行うためである．

`-XX:-BackgroundCompilation` 動的コンパイル処理のバックグラウンドでの実行を禁止する．このオプションを指定により，動的コンパイル処理が完了するタイミングの違いがベンチマークの実行に与える影響を小さくした．

`-XX:+UseLargePages` Java のヒープと動的コンパイル済みコード用ヒープをラージページ上に確保する．このオプションを指定した理由は，実装したプログラム別メモリ保護機能がラージページ向けのためである．プログラム別メモリ保護を利用するか否かによらず，このオプションを指定することで，Java のヒープや動的コンパイル済みコードの配置先の違いがベンチマークの実行に影響を与えないようにした．

各ベンチマークのスコアの測定は，動的コンパイル処理が終わり，アプリケーションの実行が定常状態になってから行った．

☆ Intel Xeon は米国およびその他の国における米国 Intel Corporation の登録商標です．

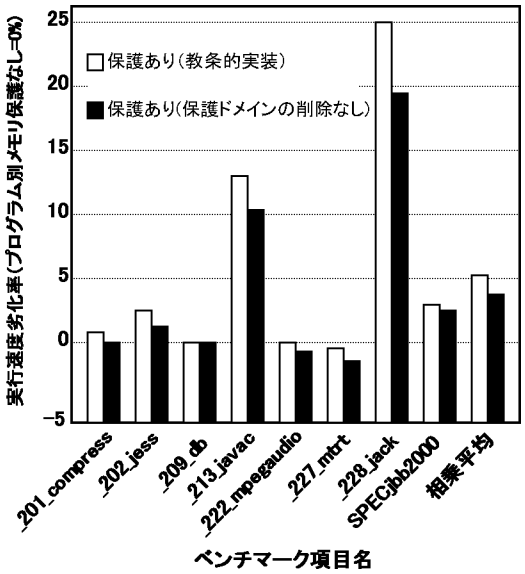


図 5 プログラム別メモリ保護の利用にともなう実行時オーバヘッド
Fig. 5 Runtime overhead to protect local heap.

5.2 評価結果

評価結果を**図 5** に示す．図 5 の縦軸はプログラム別メモリ保護の利用にともなって生じる実行速度の劣化率を表し，横軸はベンチマーク項目名を表す．図 5 から，実行速度の劣化率は，システムコール `setpd()` を教条的に実装した場合（図 3）には相乗平均で 5.2%，最大で 25.1%になり，保護ドメインの削除をプロセスの終了まで諦めると，それぞれ 3.8%，19.4%になることが分かる．

最大で 20%前後になる速度の劣化率は無視できるものではないが，MVM における Java のヒープの保護[6]，すなわち Java アプリケーションとネイティブライブラリを別々のプロセスで実行する技法に比べれば，被るオーバヘッドは小さく，少なくとも，プログラム別メモリ保護をネイティブライブラリのデバッグといった用途に利用する分には実用に耐える水準にあるのではないかと考える．

なお，`_228_jack` で実行速度の劣化が最大になる理由は，JNI 呼び出しのうち，`setpd()` を必要とするもの☆☆を頻繁に実行するためである．SPECjvm98 を対象に，プログラム別メモリ保護を利用しない場合について，`setpd()` を必要とする JNI 呼び出しの実行頻度を測定した結果を**表 4** に示す．表 4 から分かるように，`_228_jack` が，`setpd()` を必要とする JNI 呼び出しを最も頻繁に実行する．

☆☆ 4 章で述べたように，C 関数からの JNI 呼び出しのうち，Java のヒープへの書き込みをともなわないものは `setpd()` を必要としない．

表 4 setpd() を必要とする JNI 呼び出しの実行頻度
Table 4 Frequency of JNI calls that require setpd().

ベンチマーク項目名	実行頻度 (times/sec)
_201_compress	473
_202_jess	7,491
_209_db	1,440
_213_javac	34,429
_222_mpegaudio	754
_227_mtrt	3,930
_228_jack	45,434

5.3 実行時オーバヘッドの発生源の分析

プログラム別メモリ保護の利用にともなう実行時オーバヘッドを軽減する余地の有無を求めるために，大きな実行時オーバヘッドが生じている 2 つのベンチマーク項目（_213_javac と _228_jack）を対象として，実行時オーバヘッドの発生源がどこか分析した．他のベンチマーク項目を分析の対象外とした理由は，発生する実行時オーバヘッドが小さく，分析が困難なためである．

分析対象のプログラム別メモリ保護は，システムコール setpd() を図 3 のとおり，教条的に実装したものとした．分析では，次に示す 2 つの要因 users，CR3 から生じる実行時オーバヘッドが，発生する実行時オーバヘッド全体に対して，どれだけの割合を占めるか調査した．

DEL プロセスの実行中に保護ドメインを削除可能にするための処理．この処理は図 3 の 36，39〜46，53 行目にある．

CR3 プロセッサが保持する，ページグローバルディレクトリを参照するレジスタ CR3 の書き換え．

レジスタ CR3 を書き換えると，Translation Look-aside Buffer（TLB と略記する）の内容がフラッシュされるため，レジスタ CR3 を書き換えるためのコストより大きな性能劣化が生じる．ここで TLB とは，プロセッサがアドレス変換を高速化するために使うバッファのことである．

プログラム別メモリ保護を利用すると，次の 2 つの要因により，OS がレジスタ CR3 を書き換える頻度が増える．

(1) システムコール setpd() の実施．図 3 の 47 行目にレジスタ CR3 を書き換える処理がある．
(2) 所属先保護ドメインの異なるスレッドにまたがるタスクスイッチ．

これらのうち，実行時オーバヘッドに大きな影響を与える要因は前者である．たとえば _228_jack の実行時には，システムコール setpd() が毎秒 7 万回程度呼び出されるが，タスクスイッチの頻度は通常，それよりはるかに少ない．

表 5 実行時オーバヘッドの発生源と setpd() の実行コスト
Table 5 Source of runtime overheads and cost for setpd().

ベンチマーク項目名	実行時オーバヘッド			実行コスト
	O_{DEL}	O_{CR3}	その他	
_213_javac	19.8%	53.2%	27.0%	1.9 μs
_228_jack	22.8%	60.8%	16.4%	2.8 μs

5.3.1 分析の手順

分析は次の手順で行った．

(1) 次に示す 4 つの場合についてベンチマークの実行を行い，それぞれの実行にかかる時間 $T_{original}$，$T_{protect}$，T_{noDEL}，T_{noCR3} を測定した．
- 保護ドメインの利用なし
- 保護ドメインの利用あり
- 保護ドメインの利用あり（ただし，DEL に関係する処理を省略）
- 保護ドメインの利用あり（ただし，CR3 に関係する処理を省略）

(2) 保護ドメインの利用にともなうオーバヘッドの全体に対して，DEL および CR3 から生じるオーバヘッドが占める割合 O_{DEL}，O_{CR3} を次の式によって算出した．

$$O_{DEL} = \frac{T_{protect} - T_{noDEL}}{T_{protect} - T_{original}} \quad (1)$$

$$O_{CR3} = \frac{T_{protect} - T_{noCR3}}{T_{protect} - T_{original}} \quad (2)$$

この分析方法では，ベンチマークの実行時間が実行ごとに異なることの影響を受けるため，分析の精度を高くすることはできないが，おおむねの傾向は理解できる．

5.3.2 分析結果

分析結果を**表 5** に示す．表 5 から分かるように，O_{CR3} や O_{DEL} の値はベンチマーク項目によって多少異なる．差異の原因としては，測定誤差のほか，メモリアクセスパターンや setpd() を呼び出すタイミングの違いなどが考えられる．CR3 から生じるオーバヘッドはこれらの要因によって変化する．表 5 には，setpd() の平均実行コストも併記したが，この値もベンチマーク項目ごとに異なっている．ここで，平均実行コストは，保護ドメインの利用にともなって生じる全オーバヘッドを，setpd() の実行回数で除して求めた．この計算では，実行時オーバヘッドの多くが setpd() から発生し，他の要因は無視できると仮定した．なお，setpd() の実行コストを，setpd() を連続的に呼び出すベンチマークを使って評価した結果は 1.17 μs だが，表 5 記載のコストはそれより大きな値

になっている．その理由は，実アプリケーションの方が CR3 から大きな影響受けるためだと考える．

表 5 から，実行時オーバヘッドの主な発生源はレジスタ CR3 の書き換えだといえるが，この実行時オーバヘッドは容易に削減できない．レジスタ CR3 の書き換えを回避するために，品川ら[30] および Chiueh ら[3] は，IA32 が提供するセグメンテーションの機能を利用する方法を提案している．

ただし品川らの技法は，サーバサイドにおける JavaVM など，ギガバイト単位のヒープを使うアプリケーションには適用しにくい．なぜなら，この技法では，使用する保護ドメインの数に応じてプロセスが使用可能な論理アドレス空間の広さに制限がかかってしまうため，たとえば JavaVM から使用可能なヒープが狭くなり，ギガバイト単位のヒープの確保が困難になるからである．ただし品川らの技法は PowerPC™ [15] 向けの保護ドメインの実装では実用的になりうる．PowerPC は IA32 と同様に，仮想メモリの実現にあたってセグメンテーションとページングを併用するが，論理アドレスをセグメンテーション機構によって変換したあとのアドレス（IA32 におけるリニアアドレス，PowerPC における仮想アドレス）は，IA32 が 32 bit であるのに対し，32 bit の PowerPC では 52 bit であり，PowerPC の方が広いアドレス空間を利用できる．品川らの技法が，使用可能な論理アドレス空間の広さを制限してしまう原因は，32 bit しかない IA32 のリニアアドレス空間を個々の保護ドメインに分割して与えることにある．一方で PowerPC では，仮想アドレス空間が広いため，個々の保護ドメインに仮想アドレス空間を 32 bit 分ずつ分け与え，個々の保護ドメインから 32 bit の論理アドレス空間の全域を使用可能にできる．

Chiueh らの技法は，品川らの技法とは異なり，利用可能な論理アドレス空間を制限しない．Chiueh らの技法は，プログラムを，核となる部分とその拡張部分に分割し，拡張部分の実行中に，核となる部分向けのメモリ領域を破壊できなくする機能を提供する．Chiueh らは，核となる部分と拡張部分にまたがる関数呼び出しを，システムコールやレジスタ CR3 の書き換えなしで実現する技法を示しており，この技法を使えば，効率的にメモリを保護できる．ただし Chiueh らの技法では拡張部分のプログラムが別の拡張部分向けのメモリ領域を破壊することを防ぐことはできず，また，IA32 プロセッサに固有なセグメンテーションの機能を利用したものであることから，他のプロセッサに応用しにくい．本論文で示したプログラム別メモリ保護は，これらの欠点をともなわない，より汎用的なメモリ保護の枠組みである．

レジスタ CR3 の書き換えは，ハードウェアの支援を通じて削減することも可能である．たとえば 1 つのディレクトリエントリに，複数の保護ドメインの読み書き権限を記録可能にすれば，複数の保護ドメインでページグローバルディレクトリを共有可能になる．このとき，保護ドメインの切替えにおいて，切替え前後で使用するページグローバルディレクトリが同一のケースについて，レジスタ CR3 の書き換えが不要になる．IA32 のマニュアルを参照すると，ディレクトリエントリのいくつかのビットは未使用のまま残っていることが分かり，この機能を実現する余地はあるといえるが，この機能を提供するハードウェアはむろん存在しない．IA32 以外のアーキテクチャには，TLB のフラッシュによるオーバヘッドを回避するために，タグ付き TLB を実装しているものがある[12),20),23),27]．タグ付き TLB とは，複数のアドレス空間向けのアドレス変換情報を保持できる TLB である．本論文で示した保護ドメインの実装では，保護ドメインを実質的に 1 つのアドレス空間として実現しているが，ここで TLB が複数のアドレス空間に対するアドレス変換情報を保持できるのであれば，保護ドメインの切替え（setpd()）にあたって TLB をフラッシュする必要がなくなり，プログラム別メモリ保護を効率的に実装可能になる．Takahashi らはタグ付き TLB を使ったプログラム別メモリ保護に相当する機能の効率的な実現を示している[26]．

表 5 から，プロセスの実行中に保護ドメインを削除可能にするための実行時オーバヘッドが約 20%に達することが分かるが，これを削減するには，3.3 節で述べたように，プロセスの終了まで保護ドメインの削除をあきらめればよい．表 5 で「その他」に分類した実行時オーバヘッドの主な発生原因は，割込み命令など，setpd() などのシステムコールの発行にともなって必ず実行すべき処理である．したがって，この部分にはあまり高速化の余地がない．

ここまでの考察から，実行時オーバヘッドを軽減する手段として，現状のハードウェアを使ったプログラム別メモリ保護の実装では，OS 側の処理を高速化するために，プロセスの終了まで保護ドメインの削除をあきらめることが有効であることが分かる．さらなる実行時オーバヘッドの軽減を目指す場合には，ユーザプログラム（Java VM）側に手を加え，保護ドメイン切替の瀬度を減らす方法も有効である．たとえば今回の実験ではすべての JNI 呼び出しにおいてメモリ

保護ドメインの切替えを実施したが，必ずしもすべての JNI 呼び出しを監視の対象とする必要はない．Java VM と標準クラスライブラリからなる Java 実行環境の全体を 1 つのプログラム製品と見なせば，標準クラスライブラリ中のネイティブメソッドを呼び出す際にはメモリ保護ドメインの切替えが不要になり，オーバヘッドの軽減を図ることができる．この技法を適用すると，SPECjvm98 および SPECjbb2000 の実行における実行速度の劣化をほぼ解消できる．なぜならこれらのベンチマークは Java のみで実現されており，標準ライブラリ以外のネイティブメソッドを呼び出さないからである．今回の実験ですべての JNI 呼び出しにおいてメモリ保護ドメインの切替えを実施した主な理由は，MVM との比較のためである．もちろん MVM は JavaVM に固有なヒープの保護のほかにも様々な機能を提供するものであり，単純に比較すべきものでは決してないが，ヒープの保護のみに限れば，プログラム別メモリ保護の方が，被る実行時オーバヘッドは小さい．

6. 結　　論

JavaVM に固有なヒープ領域を保護するための技術として，プログラム別メモリ保護を利用した場合に発生する実行時オーバヘッドが，どれだけの大きさになるか，評価を行った．評価にあたっては，まず，Linux カーネルを改造してプログラム別メモリ保護の機能を実現し，次に，JavaVM を改造してプログラム別メモリ保護を使ってヒープを保護する機能を実現した．実用的アプリケーションを模したベンチマーク SPECjvm98 および SPECjbb2000 を使って評価した結果，実行時オーバヘッドは相乗平均で 3.8%，最大 19.4%になることが分かった．

参 考 文 献

1) Arnold, K., Gosling, J. and Holmes, D.: *The Java Programming Language*, 4th Edition, Addison-Wesley, Reading, Mass. (2005).
2) 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. Computer System*, Vol.12, No.4, pp.271–307 (1994).
3) Chiueh, T., Venkitachalam, G. and Pradhan, P.: Integrating Segmentation and Paging Protection for Safe, Efficient and Transparent Software Extensions, *Proc. 17th ACM Symposium on Operating System Principles*, pp.140–153 (1999).
4) Cohen, E. and Jefferson, D.: Protection in the Hydra Operating System, *Proc. 5th ACM Symposium on Operating System Principles*, pp.141–160 (1975).
5) Czajkowski, G., Danyés, L. and Nystrom, N.: Code Sharing among Virtual Machines, *Proc. 16th European Conference on Object-Oriented Programming*, pp.155–177 (2002).
6) Czajkowski, G., Danyés, L. and Wolczko, M.: Automated and Portable Native Code Isolation, *Proc. 12th IEEE International Symposium on Software Reliability Engineering*, pp.298–307 (2001).
7) Demetrescu, C. and Finocchi, I.: A Portable Virtual Machine for Program Debugging and Directing, *Proc. 2004 ACM Symposium on Applied Computing*, pp.1524–1530 (2004).
8) Dennis, J.B. and Horn, E.C.V.: Programming Semantics for Multiprogrammed Computations, *Comm. ACM*, Vol.9, No.3, pp.143–155 (1966).
9) Dike, J.: *User Mode Linux*, Prentice Hall, Indianapolis, Ind. (2006).
10) Houdek, M.E., Soltis, F.G. and Hoffman, R.L.: IBM System/38 Support for Capability-Based Addressing, *Proc. 8th Annual Symposium on Computer Architecture*, pp.341–348 (1981).
11) Intel Corporation: *IA-32 Intel Architecture Software Developer's Manual* (2006). http://www.intel.com/design/Pentium4/documentation.htm
12) Intel Corporation: *Intel Itanium Architecture Software Developer's Manual* (2006). http://www.intel.com/design/itanium/manuals/iiasdmanual.htm
13) International Bussiness Machines Corporation: *ILE Concepts* (2002). http://publib.boulder.ibm.com/iseries/v5r2/ic2924/books/c4156066.pdf
14) International Bussiness Machines Corporation: *Java and WebSphere Performance on IBM iSeries Servers* (2002). http://www.redbooks.ibm.com/redbooks/pdfs/sg246256.pdf
15) International Bussiness Machines Corporation: *PowerPC Architecture Book* (2003). http://www-128.ibm.com/developerworks/eserver/articles/archguide.html
16) Levin, R., Cohen, E., Corwin, W., Pollack, F. and Wulf, W.: Policy/mechanism Separation in Hydra, *Proc. 5th ACM Symposium on Operating System Principles*, pp.132–140 (1975).
17) Liang, S.: *The Java Native Interface: Pro-*

grammer's Guide and Specification, Addison-Wesley, Reading, Mass. (1999).
18) McKenney, P.E. and Slingwine, J.D.: Read-Copy Update: Using Execution History to Solve Concurrency Problems, *Proc. IASTED International Conference on Parallel and Distributed Computing and Systems*, pp.509–518 (1998).
19) Microsoft Corporation: *Microsoft C# Language Specifications*, Microsoft Press, Redmond, Wash. (2001).
20) MIPS Technologies, Incorporated: *MIPS32 Architecture for Programmers Volume III: The MIPS Privileged Resource Architecture* (2005). http://www.mips.com/content/Documentation/MIPSDocumentation/ProcessorArchitecture/doclibrary
21) Paleczny, M., Vick, C. and Click, C.: The Java HotSpot Server Compiler, *Proc. Java Virtual Machine Research and Technology Symposium*, pp.1–12 (2001).
22) Sosič, R.: Dynascope: A tool for program directing, *Proc. ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pp.12–21 (1992).
23) SPARC International, Incorporated: *SPARC Architecture Manual, Version 9* (2000). http://developer.sun.com/solaris/articles/sparcv9.html
24) Standard Performance Evaluation Corporation: SPEC JVM98 Benchmarks (1998). http://www.spec.org/osg/jvm98/
25) Standard Performance Evaluation Corporation: SPEC JBB2000 (2000). http://www.spec.org/osg/jbb2000/
26) Takahashi, M., Kono, K. and Masuda, T.: Efficient kernel support of fine-grained protection domains for mobile code, *Proc. 19th IEEE International Conference on Distributed Computing Systems*, pp.64–73 (1999).
27) The Alpha Architecture Committee: *The Alpha AXP Architecture Reference Manual, Third Edition*, Digital Press, Boston, Mass. (1998).
28) Wulf, W., Levin, R. and Pierson, C.: Overview of the Hydra Operating System Development, *Proc. 5th ACM Symposium on Operating System Principles*, pp.122–131 (1975).
29) 孝壽俊彦，高田眞吾，土居範久：既存の開発環境との互換性と高速な実行を実現したプログラム実行制御・監視環境，情報処理学会論文誌，Vol.46, No.12, pp.3040–3053 (2005).
30) 品川高廣，河野健二，高橋雅彦，益田隆司：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，情報処理学会論文誌，Vol.40, No.6, pp.2596–2606 (1999).

(平成 18 年 5 月 1 日受付)
(平成 18 年 8 月 10 日採録)

千葉 雄司（正会員）

1972 年生．1997 年慶應義塾大学大学院理工学研究科計算機科学専攻修士課程修了．同年（株）日立製作所入社．システム開発研究所にてコンパイラの研究開発に従事．