Kernel Mode Linux:
ハードウェアに頼らずにオペレーティングシステムを保護する方法

前田俊行
東京大学大学院情報理工学系研究科

1 はじめに

従来の OS (オペレーティングシステム) は今日のほとんどの CPU が持つ「特権レベル機構」によって保護されてきた．特権レベル機構とは，実行するプログラム毎に特権レベルを割り当て，特権レベルの低いプログラムが特権レベルの高いプログラムへアクセスすることを防ぐ仕組みである．通常，OS は最も高い特権レベル (カーネルモード) で実行され，ユーザプログラムは最も低い特権レベル (ユーザモード) で実行される．このため，もしユーザプログラムが OS を攻撃しようとしても，CPU の特権レベル機構が働き，OS はその攻撃から保護される．

この特権レベル機構を用いた保護の問題の一つは，システムコール (ユーザプログラムから OS のサービスを呼び出すこと) が遅くなる点である．これは，システムコールを実行するためには CPU の特権レベルを安全に切り替えなければならず，その処理にかかるコストが大きいためである．例えば，現在広く使われている Pentium 4 という CPU 上では通常の関数呼び出しに比べてシステムコールは 100 倍以上遅くなっている．このシステムコールのオーバーヘッドはシステムコール呼び出しを頻繁に行うようなプログラム (ウェブサーバやデータベース) では性能に大きく影響する．

この問題を解決する最も単純な方法は，特権レベル機構を使わないこと，つまり，OS だけではなく，ユーザプログラムもカーネルモードで実行してしまうことである．するとユーザプログラムは OS に直接アクセスできるようになるため，システムコールは単なる関数呼び出しと全く同等となり，システムコールのオーバーヘッドは完全に解消できる．しかしこの方法では，ユーザプログラムは OS，そしてシステムのハードウェアさえも完全に自由にアクセスできてしまうため，システムの安全性が損なわれてしまう．

そこで本研究では，ユーザプログラムをカーネルモードで実行し，かつ，システムの安全性を損なわない OS の保護機構を提案する．我々の手法では，プログラマはユーザプログラムを型付きアセンブリ言語 (TAL) を用いて記述する．型付きアセンブリ言語とは Morrisett ら [4, 3] によって提案された型安全性を保証できるアセンブリ言語である．OS はユーザプログラムを実行する前に型検査を行い，そのユーザプログラムが型安全であるかを検証する．型安全であると検証されたプログラムは実行時に不正なメモリアクセスやコード実行を行わないので，安全にカーネルモードで実行できる．

この手法にもとづいて我々は，ユーザプログラムをカーネルモードで実行できる Linux カーネル，Kernel Mode Linux (KML) [1] を実装した．この KML 上でいくつかの性能測定実験を行ったところ，KML がシステムコールのオーバーヘッドを解消し，ファイル I/O 操作の性能を改善できることが分かった．

2 手法

まずプログラマはユーザプログラムを型付きアセンブリ言語 (TAL) を用いて作成する．TAL は，レジスタやメモリ，コードラベルが型付けされたアセンブリ言語であり，型検査をパスした TAL プログラムは，実行時に不正なメモリアクセスを行わないこと (メモリ安全性)，不正なコード実行を行わないこと (制御フロー安全性) が保証される．(詳細は [4, 3, 2] 等を参照されたい．)

プログラムの作成にあたって，プログラマは直接

TAL でプログラムを記述してもよいし，他の高級プログラミング言語で記述し，それをコンパイラで TAL に変換してもよい．TAL は，型付けされていることを除けば通常のアセンブリ言語なので，型安全な言語から TAL への変換が理論的には可能である．このためプログラマは，TAL へのコンパイラが存在すれば，様々なプログラミング言語を利用できる．

次に，カーネルはプログラムを実行する前に TAL の型検査を行いユーザプログラムの安全性を検証する．安全性が確認されたプログラムはカーネルモードで実行される．TAL のアセンブラは機械語コードと同時に型情報も生成するので，この情報を用いることで，機械語コードのレベルでプログラムの安全性を検証することが可能である．このため外部のコンパイラやアセンブラ等を信用する必要がなく，TCB (Trusted Computing Base) を小さくできる．また TAL の型検査はプログラムの実行前に行われ，実行時の安全性検査はほとんど不要なため，プログラムの実行性能が低下しない．

上述したように，TAL の型システムが保証するのは，メモリ安全性と制御フロー安全性という比較的単純なものであるが，従来のハードウェアの特権レベル機構が保証している安全性も実はこれら 2 つの安全性とほぼ等価であるため，特権レベル機構を置き換えるには TAL の型システムで十分である．(より詳細な議論は [2] を参照されたい．)

3 実装: Kernel Mode Linux

我々は前節で述べた手法にもとづき，Intel IA-32 アーキテクチャ CPU 用の Linux カーネルを改造し，ユーザプロセスをカーネルモードで実行できるカーネル，Kernel Mode Linux (KML) を実装した．

ユーザプロセスをカーネルモードで実行するために KML は，CPU の特権レベルを表す特別なビットを，通常のユーザプロセス実行ではユーザモードに設定するところをカーネルモードに設定する．基本的にはこれだけでユーザプロセスをカーネルモードで実行できる．(実際にはこの操作に伴う副作用としてメモリスタックの扱いを変更しなければならない．詳細については [2] を参照されたい．)

このように，KML においてカーネルモードで実行されるユーザプロセスは，特権レベルを除いて，通常のユーザプロセスと全く変わらないため，例えばメモリページングやプロセススケジューリングなども通常どおり行われる．このためメモリを大量に消費したり無限ループに入ってしまうようなプログラムでもカーネルモードで安全に実行できる．

ただし現時点の KML は，まだ型検査器がカーネルに組込まれていない．そのためカーネルはユーザプログラムの安全性を直接検証することはできない．型検査器をカーネルに組込むこと自体になんら理論的問題は存在しないので，我々は容易に型検査器をカーネルに組込めると考えている．

また付加機能として我々は，既存のバイナリプログラムを全く変更することなく，システムコールの呼び出しを単なる関数呼び出しに置き換え，高速化する仕組みを実装した．具体的には，最近の Linux カーネルが持つ，システムコール呼び出しの方法を切り替える仕組みを流用した．これは，システムコール呼び出しの方法として，従来のソフトウェア割込み命令 (*int*) ではなく，比較的新しい，システムコールのための高速なソフトウェア割込み命令 (*sysenter*/*sysexit*) を，もし可能であれば自動的に検出して用いるための仕組みである．KML は，この仕組みに第 3 の呼び出し方法として，直接関数呼び出しとしてシステムコールを実現するという選択肢を追加した．この機能は特にプログラムの性能比較を行う場合に便利である．またプログラムの安全性よりも性能を重視するような場合においても有用と考えられる．

4 性能比較実験

前節で述べた実装 (Kernel Mode Linux) を用いていくつかの性能比較実験を行った．実験では全く同一のバイナリプログラムをオリジナルの Linux と KML 上でそれぞれ実行し結果を比較した．システムコールの実行にあたっては，オリジナルの Linux 上では *sysenter*/*sysexit* 命令を利用し，KML 上では前節で述べた既存のバイナリプログラムを変更することなくシステムコールを高速化する仕組みを利用した．実験環境は表 1 の通りである．

表 1: 実験環境

CPU	Pentium 4 3.000GHz
メモリ	1GB (PC3200 DDR SDRAM)
ハードディスク	120GB
OS	Linux kernel 2.5.72

表 2: システムコールのレイテンシ (“sysenter” は *sysetner/sysexit* を用いたオリジナルの Linux カーネル)

	`getpid`	`read`	`write`	`fstat`
sysenter	135.1	201.1	164.9	383.5
KML	16.9	91.0	53.4	204.1

(単位: ナノ秒)

4.1 システムコールのレイテンシ測定

まずベンチマークソフト lmbench を用いてシステムコールのレイテンシを計測した．実験では 4 つのシステムコール (getpid，read，write，fstat) を対象とした．

結果は表 2 の通りである．実験結果によりシステムコールに伴うオーバーヘッドが約 100〜200 ナノ秒であり，KML がそのオーバーヘッドを解消できることが確かめられた．

4.2 ファイル I/O 操作のスループット

次にベンチマークソフト IOzone を用いてファイル I/O 操作のスループットがどの程度改善されるかを計測した．実験では 4 種類のファイル I/O 操作 (Read，Write，Re-read，Re-write) を対象とした．Read と Write はファイルの読み書きの操作，また Re-read と Re-write はそれぞれ一度読み書きしたファイルを更に繰り返して読み書きする操作である．これら 4 種類のファイル I/O 操作についてファイルサイズを変えてそれぞれスループットを計測した．なお実験で用いたファイル I/O 操作のためのメモリバッファは 16 キロバイトである．

結果は図 1 の通りである．それぞれ，Read では最大 25.9 %，Write では最大 8 %，Re-read では最大 14.7 %，Re-write では最大 15.9 %，スループットが改善した．なお，KML 上で実行した方がスループットがわずかに悪化している場合 (特に Write において) があるが，これはハードディスクへのアクセスなどの要因により，システムコールのオーバーヘッド解消による性能改善が相対的に小さくなってしまっているためと考えられる．

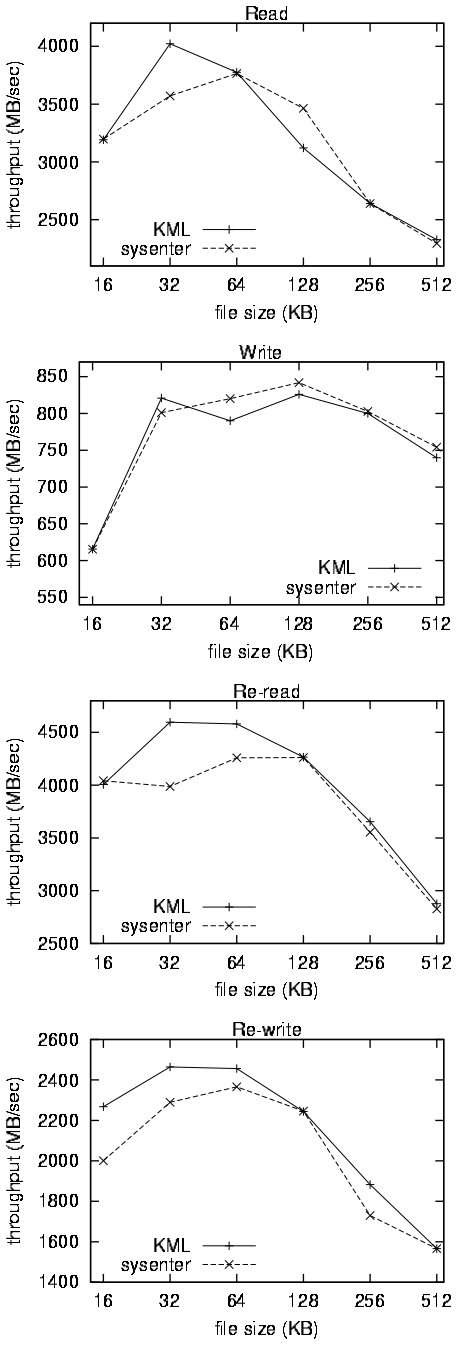

図 1: ファイル I/O 操作のスループット (“sysenter” は *sysetner/sysexit* を用いたオリジナルの Linux カーネル)

5 おわりに

我々はシステムコール呼び出しの性能を改善するために，ユーザプログラムを安全にカーネルモードで実行する手法を提案した．この手法にもとづきユーザプログラムをカーネルモードで実行する OS カーネル，Kernel Mode Linux (KML) を実装し，KML 上で性能比較実験を行ったところ，実際にシステムコール呼び出しが高速化し，ファイル I/O 操作の性能が改善されることが確かめられた．

6 今後

現在の KML は，単にシステムコールのオーバーヘッドを解消しているだけだが，TAL の型システムをより積極的に利用することにより，更なるプログラムの性能改善が期待できる．例えば，TAL の型安全性により，OS カーネル内のデータをユーザプログラムに安全に直接公開することができる．このことを利用すれば，例えばネットワーク I/O 操作で用いられる OS カーネル内のメモリバッファを直接ユーザプログラムからアクセス可能にすることでゼロコピー通信を実装し，安全かつ高性能なネットワーク通信を実現できる．

また我々は OS 自身を型安全な言語，特に TAL で構築することも考えている．TAL を用いれば，型安全性を検証するのが困難に見えるコード，例えばハードウェアを直接操作するようなコードでも多くの部分が記述可能となる．型安全な言語で OS を記述することの利点は，OS 自身のメモリ安全性と制御フロー安全性を保証できる点である．もちろん，より複雑な安全性 (デッドロックが生じないという安全性，ファイルシステムが壊れないという安全性など) は TAL のみでは保証できないが，現在広く使われている OS がそもそもメモリ安全性や制御フロー安全性すら形式的に検証されていない点を考えれば，大きな意味がある．

参考文献

[1] Kernel Mode Linux. http://www.yl.is.s.u-tokyo.ac.jp/~tosh/kml.

[2] Toshiyuki Maeda and Akinori Yonezawa. Kernel Mode Linux: Toward an operating system protected by a type theory. In *Proceedings of the 8th Asian Computing Science Conference (ASIAN'03)*, Vol. 2896 of *Lecture Notes in Computer Science*, pp. 3–17, Mumbai, India, December 2003. Springer.

[3] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *Proc. of ACM SIGPLAN Workshop on Compiler Support for System Software*, pp. 25–35, 1999.

[4] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, Vol. 21, No. 3, pp. 527–568, 1999.