Linux/TAL: 型付きアセンブリプログラムのカーネルモード実行方式

前田 俊行 [1]　　住井 英二郎 [2]　　米澤 明憲 [2]

[1] 東京大学 大学院理学系研究科情報科学専攻
[2] 東京大学 大学院情報理工学系研究科コンピュータ科学専攻

E-mail: {tosh, sumii, yonezawa}@yl.is.s.u-tokyo.ac.jp

要旨

古典的なオペレーティングシステムのシステムコールには、ユーザモードからカーネルモードへ保護ドメインを遷移させるためのオーバーヘッドが存在する。しかし、もしユーザプログラムをカーネルモードで安全に実行できれば、このオーバーヘッドを除去することが可能なはずである。これを実現するために我々は、Morrisett らの型付きアセンブリ言語 (TAL) を用いて、ユーザプログラムをカーネルモードで実行する方式を提案する。TAL とは、型システムによってプログラムのメモリ安全性と制御フロー安全性を保証するアセンブリ言語である。我々の方式では、ユーザプログラムは TAL で提供され、型検査の後にカーネルモードで実行される。これにより、通常のオペレーティングシステムと同等の安全性を保持しつつ、単なる関数呼び出しと同じコストで効率的にシステムコールを実行することができる。この方式を Linux カーネルの改造により実装し、簡単なベンチマークにより性能を評価したところ、ファイルシステムのディレクトリツリーを探索するプログラム find では 14%の高速化が、ネットワークから受信したデータをそのまま送信するサーバプログラム echod では end-to-end のレイテンシで 4 マイクロ秒の改善が観察された。

1　序論

古典的なオペレーティングシステム (OS) では、中央演算装置 (CPU) のメモリ管理ユニット (MMU) が持つメモリアクセス制御機構を用いて、ユーザモードとカーネルモードに保護ドメインを分離することにより、ユーザプログラムから他のユーザプログラムやカーネルを保護している。このため、システムコールを実行するには、ユーザモードからカーネルモードに保護ドメインを切り替える必要があるが、その切り替えにはソフトウェア割り込みやコンテキスト切り替えといったコストの大きい処理が必要なので、結果としてシステムコールのコストも大きくなっている。このようなシステムコールのオーバヘッドは、高性能を要求される Web サーバ等において性能改善の障害となっている。もしユーザプログラムをカーネルモードで実行できれば、システムコールは単なる関数呼び出しとなり、このようなオーバーヘッドは解消するが、カーネルモードではシステムに対してあらゆる操作が可能なので、単純にユーザプログラムをカーネルモードで実行することはもちろん危険である。

そこで、本研究では Morrisett らの型付きアセンブリ言語 (TAL) [1, 2] を利用して、ユーザプログラムをカーネルモードで安全に実行する方式を提案する。TAL とは、型システムによってプログラムのメモリ安全性と制御フロー安全性を保証するアセンブリ言語である。ここで、メモリ安全性とは「プログラムが、読み書きを許可されたメモリのみを読み書きする」性質をいい、制御フロー安全性とは「プログラムが、実行を許可された命令列のみを実行する」

性質をいう。古典的な OS の保護方式では、これらの性質は MMU のアクセス制御によって保証されてきた。我々の提案する方式では、ユーザプログラムは TAL で提供され、型検査により上述の性質が検証されてから、カーネルモードで実行される。このため、実行時の保護ドメイン切り替えや安全性検査は不要となるので (配列の境界検査など一部を除く)、ユーザプログラムをカーネルモードで安全かつ効率的に実行することができる。

我々はこの方式にもとづくプロトタイプシステムを、Linux カーネルを改造することにより実装した。このシステムではユーザプログラムに対するカーネルのインターフェースとして、Linux カーネル内部の通常のシステムコール関数を利用しており、通常の Linux カーネルとまったく同等の機能 (ファイルのアクセス制御など) を提供しつつ、システムコールにともなうオーバーヘッドを解消している。実際にいくつかの単純なプログラムを作成・実行して性能を評価したところ、ファイルシステムのディレクトリツリーを探索するプログラム find が 14%高速化され、受信したデータをそのまま送信するサーバプログラム echod のレイテンシが 4 マイクロ秒改善された。

本論文の以降の構成は以下のとおりである。まず、第 2 節では関連研究について述べる。第 3 節では、Morrisett らによる TAL について説明する。第 4 節では、プロトタイプシステムについて述べ、第 5 節では、第 4 節で述べたプロトタイプシステムを用いた性能測定実験の結果を示す。最後に、第 6 節では本研究の今後について述べる。

2 関連研究

プログラムをカーネル内で実行することにより、システムコールなどのオーバヘッドを解消して性能を改善しようとする試みは既に行われている。例えば kHTTPd [3] や TUX [4] は、Linux カーネル内で高速に実行される Web サーバである。その他にも、カーネル内で実行される NFS サーバなどが存在する。しかし、これらは各プログラムを別々にカーネル内で実装しただけで、安全性を保証する一般的な方式ではない。

SPIN [5] は高級言語 Modula-3 で作成されたカーネル拡張を、カーネル内部に挿入して実行することができる OS である。SPIN の方式では、カーネルが Modula-3 コンパイラを信頼しなければならないので Trusted Computing Base (TCB) が大きく、またユーザプログラムを記述する言語が Modula-3 という一つの高級言語に限定される。これに対して我々の方式では、安全性検査を TAL の型検査により機械語レベルで行うため、コンパイラを信用する必要がなく SPIN に比べて TCB を小さくできる。また、TAL というアセンブリ言語を用いているため、TAL へのコンパイラがあれば様々な高級言語でユーザプログラムを作成することが可能である。実際に、C 言語に似た高級言語である Popcorn や Scheme のサブセットから TAL へのコンパイラが存在する [6]。Scheme のようないわゆる型無し言語でも、universal type にもとづく encoding や soft typing [7] によって、相応の効率で型付き言語である TAL へコンパイルできることに注意されたい。

Software Fault Isolation (SFI) [8] は、プログラムが不正なメモリアクセスやコード実行を行わないように、メモリアクセス命令や分岐命令ごとに検査コードを挿入する方式である。SFI では、プログラムの実行中に多くの安全性検査が行われるため、実行時のオーバーヘッドが大きい。これに対して我々の手法では、型検査にもとづく安全性検査が実行時ではなくプログラムのロード時に行われるので、実行速度の低下はほとんどない。

Proof Carrying Code (PCC) [9] は、プログラムに安全性の証明を付加し、それが正しいことを実行前に検証することにより、そのプログラムを安全に実行する方式である。PCC はメモリ使用量や CPU 時間の制限といった、メモリ安全性や制御フロー安全性よりも高度な安全性を保証することができる。しかし、そのような性質の証明は生成が困難なことが多く、コンパイラやプログラマの負担が大きい。

TAL は、対象とする性質をメモリ安全性と制御フロー安全性に限定することにより、証明の生成を容易にした PCC の一種とみなすことができる。(たとえば ML でユーザプログラムを作成して TAL にコンパイルするならば、標準的な型推論により全自動で証明を生成することが可能である。) そもそも通常の MMU ではメモリに対する読み書きと実行しかアクセス制御を行わないので、古典的な OS が保護ドメインの切り替えによって保証している安全性と同等の安全性を保証するだけならば、TAL のメモリ安全性と制御フロー安全性だけでも十分である。

Janos [10] は、Java 仮想機械をカーネルに組み込み、Java のバイトコードをカーネル内で安全に実行できるようにした active network 用の OS である。このような Java を用いた手法の問題点は、Java のバイトコードの実行性能が悪い点である。また、実行性能を改善しようとして、JIT コンパイラに複雑な最適化処理を行わせると、今度は TCB が巨大化するという問題点が生じてしまう。これに対し、我々の手法ではネイティブの機械語プログラムをそのまま実行するので、Java のバイトコード実行よりも実行性能は良い。また、JIT コンパイラによる複雑な最適化も必要ないため、TCB が巨大化する恐れはない。

また、より広範な観点から比較すると、上述の既存研究はどちらかといえばカーネルの機能を安全に拡張することを主眼としていたが、我々の研究は通常のユーザプログラムをカーネルモードで安全に実行することを目標としている。

3 Typed Assembly Language

Typed Assembly Language (TAL) とはレジスタやメモリに型を付けることでメモリ安全性と制御フロー安全性を保証するアセンブリ言語であり、Morrisett らによって考案された [1, 2]。本節ではまず、型を用いてプログラムのメモリと制御フローの安全性を保証する基本的な手法について説明し、その後、TAL の構文と意味論を簡単に説明する。詳細については Morrisett らの論文 [1, 2] を参照されたい。

3.1 型によるメモリ安全性の保証

図 1 のアセンブリプログラムを考える。このプログラムは、7 行目でレジスタ EBX に有効なメモリアドレスでない値を勝手に代入し、8 行目でラベル sum にジャンプしてしまっている。このため、2 行目で不正なメモリにアクセスしてしまうかもしれない。よってこのプログラムはメモリ安全でない。

```
1:  sum :
2:        add %EAX, (%EBX)
3:        jmp %EDX
4:
5:  illegal :
6:        mov $0x1, %EAX
7:        mov $0x01234567, %EBX
8:        jmp sum
```

図 1: 不正なメモリをアクセスしようとするプログラム

これに対し、図 1 のプログラムに型を付けたものが図 2 である。プログラム中の “{” と “}” で囲まれた部分が付けられた型である。これらの型は、プログラマが直接 TAL でプログラムを書く場合にはそのプログラマが、その他の言語で書いて TAL へのコンパイラを用いる場合にはそのコンパイラが与える。(なお、TAL プログラムに付けられた型は、TAL アセンブラによってプログラムが機械語コードに変換されるときに、同時に型情報として生成される。この生成された型情報を用いることで、プログラムが機械語コードに変換された後でも、型検査を行うことができる。)

図 2 において、“{” と “}” は、プログラムのラベルに付けられる型を意味しており、この “{” と “}” で囲まれた条件を満たしているときのみ、そのラベルへジャンプすることができることを表している。例えば、図 2 の 1 行目と 2 行目は、以下の 3 つの条件:

1. レジスタ EAX が整数 int 型の値を持つ

2. レジスタ EBX が整数 int 型の値を保存するメモリのアドレスを持つ

3. レジスタ EDX が次の 2 つの条件

   (a) レジスタ EAX が整数 int 型の値を持つ

   (b) レジスタ EBX が整数 int 型の値を保存するメモリのアドレスを持つ

   を満たしているときにジャンプすることができるコードラベルのアドレスを持つ

を満たしているときのみ、コードラベル sum にジャンプすることができることをあらわしている。

```
1: { EAX : int, EBX : <int>,
2:   EDX : {EAX : int, EBX : <int>} }
3:   sum :
4:         add %EAX, (%EBX)
5:         jmp %EDX
6:
7: { EDX : {EAX : int, EBX : <int>} }
8:   illegal :
9:         mov $0x1, %EAX
10:        mov $0x01234567, %EBX
11:        jmp sum
```

図 2: 不正なメモリをアクセスしようとする TAL のプログラム

この図 2 のプログラムは型検査をパスできない。なぜなら、このプログラムは 11 行目でラベル sum にジャンプしようとしているが、10 行目でレジスタ EBX の型が整数 int 型となっており、ラベル sum にジャンプするための条件のうち、2 番目の条件が満たされないからである。このような型検査によってプログラムのメモリ安全性を保証することができる。

3.2 型による制御フロー安全性の保証

図 3 のアセンブリプログラムを考える。このプログラムは、7 行目でレジスタ EDX に有効なプログラムアドレスでない値を勝手に代入し、8 行目でラベル sum にジャンプしてしまっている。このため、3 行目で不正なアドレスにジャンプしてしまい、不正なコードを実行してしまうかもしれない。よって、このプログラムは制御フロー安全ではない。

```
1:   sum :
2:         add %EAX, (%EBX)
3:         jmp %EDX
4:
5:   illegal :
6:         mov $0x1, %EAX
7:         mov $0x01234567, %EDX
8:         jmp sum
```

図 3: 不正なコードを実行しようとするプログラム

これに対し、図 3 のプログラムに型を付けたものが図 4 である。

```
1: { EAX : int, EBX : <int>,
2:   EDX : {EAX : int, EBX : <int>} }
3:   sum :
4:         add %EAX, (%EBX)
5:         jmp %EDX
6:
7: { EBX : <int> }
8:   illegal :
9:         mov $0x1, %EAX
10:        mov $0x01234567, %EDX
11:        jmp sum
```

図 4: 不正なコードを実行しようとする TAL のプログラム

この図 4 のプログラムは型検査をパスできない。なぜなら、このプログラムは 11 行目でラベル sum にジャンプしようとしているが、10 行目でレジスタ EDX の型が整数 int 型となっており、ラベル sum にジャンプするための条件のうち、3 番目の条件が満たされないからである。このような型検査によってプログラムの制御フロー安全性を保証することができる。

3.3 TAL の基本的構文とその直観的意味

図 5 は TAL の基本的な構文である。

型	τ	::=	$\alpha \mid int \mid \forall[\vec{\alpha}].\Gamma \mid \langle \tau_1, \ldots, \tau_n \rangle \mid \exists \alpha.\tau \mid S(n) \mid array(n, \tau)$
ヒープ型	Ψ	::=	$\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$
レジスタファイル型	Γ	::=	$\{r_1 : \tau_1, \ldots, r_n : \tau_n\}$
レジスタ	r	$\in$	$\{EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP\}$
ワード値	w	::=	$l \mid i \mid w[\tau] \mid pack[\tau, w]\ as\ \tau'$
ヒープ値	h	::=	$\langle w_1, \ldots, w_n \rangle \mid (w_1, \ldots, w_n) \mid \texttt{code}[\vec{\alpha}]\Gamma.I$
スモール値	v	::=	$r \mid w \mid v[\tau] \mid pack[\tau, v]\ as\ \tau'$
ヒープ	H	::=	$\{l_1 \mapsto h_1, \ldots, l_n \mapsto h_n\}$
レジスタファイル	R	::=	$\{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\}$
命令	ι	::=	$\texttt{mov}\ r_s, r_d \mid \texttt{add}\ r_s, r_d \mid \texttt{sub}\ r_s, r_d \mid \texttt{cmp}\ r_s, r_d \mid \texttt{jcc}\ r_s \mid \ldots$
			$\texttt{malloc}\ r_d\ [\vec{\tau}] \mid \texttt{asub}\ r_{array}, r_{size}, r_{index}, r_d \mid$
			$\texttt{aupd}\ r_s, r_{array}, r_{size}, r_{index} \mid \texttt{unpack}\ [\alpha, r_d], v$
命令列	I	::=	$\iota; I \mid \texttt{jmp}\ v \mid \texttt{halt}[\tau]$
プログラム	P	::=	(H, R, S)

図 5: TAL の構文

TAL の型 τ は、型変数 α、整数型 int、コードラベル型 $\forall[\vec{\alpha}].\Gamma$、タプル型 $\langle \tau_1, \ldots, \tau_n \rangle$、existential 型 $\exists \alpha.\tau$、シングルトン型 $S(n)$、配列型 $array(n, \tau)$ からなる。ここで、コードラベル型 $\forall[\vec{\alpha}].\{r_1 : \tau_1, \ldots, r_n : \tau_n\}$ は、型付けされたラベルにジャンプするためには、型変数集合 α に具体的な型を適用したとき、レジスタ r_1 から r_n の型がそれぞれ τ_1 から τ_n でなければならないことを表している。また、シングルトン型 $S(n)$ は、整数 n のみからなる型をあらわす。例えば、レジスタ EAX が型 $S(15)$ であるときには、レジスタ EAX は整数値 15 を保存している。また、配列型 $array(n, \tau)$ は、長さが n で各要素の型が τ であるような配列をあらわす。例えば型 $array(5, int)$ は、長さが 5 の整数値の配列をあらわす。

TAL のプログラム (または実行状態) は、ヒープ H、レジスタファイル R、命令列 I からなる。ヒープ H はラベル l からヒープ値 h へのマップであり、レジスタファイル R はレジスタ r からワード値 w へのマップである。ここでヒープ値 h はヒープに保存できる値をあらわしており、ワード値のタプル $\langle w_1, \ldots, w_n \rangle$、配列 $(w_1, \ldots, w_n)$、または命令列 $\texttt{code}[\alpha]\Gamma.I$ のどれかである。またワード値 w は、レジスタに保存できる値をあらわしており、ラベル (ヒープ値へのポインタ)、整数値、型を適用したワード値 $w[\tau]$、そして existential package $pack[\tau, w]\ as\ \tau'$ のどれかである。なおスモール値 v は命令の即値オペランドとして指定できる値をあらわしている。

命令列 I は、文字通り命令 ι の列である。命令列の最後は jmp 命令か halt 命令のどちらかでなくてはならない。命令 ι は TAL で利用できるアセンブリ命令であり、よく使われる IA-32 の機械語命令はほとんど全て利用できる。(なお、図 5 では、ごく一部の命令だけを示しており、また命令のオペランドは、直接レジスタを取る場合だけを示している。)

命令 ι のうち、malloc、asub、aupd、unpack の 4 つは特殊なマクロ命令である。

malloc 命令は、オペランドに指定された型 $\vec{\tau}$ を持つタプル (または配列) をヒープに確保し、そのアド

レスをレジスタ r_d に保存する命令である。当然、機械語にはそのような命令は存在しないので、機械語に変換されるときには、必要なサイズのメモリをヒープから確保するような特定の命令列に展開される。

asub 命令と aupd 命令は配列アクセスのためのマクロ命令である。asub 命令は配列の要素を読む命令であり、aupd 命令は配列の要素を更新する命令である。オペランドには、配列 r_{array}、配列の長さ r_{size}、アクセスする位置 r_{index} を指定し、これに加え asub 命令には読んだデータを保存するレジスタ r_d を、aupd 命令には更新するデータを持つレジスタ r_s を指定する。これらのマクロ命令は、TAL アセンブラによって機械語に変換されるときに、オペランドに指定された配列の長さ r_{size} とアクセス位置 r_{index} にもとづいて境界検査を行ってから配列アクセスを行う特定の命令列に展開される。なお、これら配列アクセス命令の型検査では、オペランドに指定された配列 r_{array} の型 $array(n, \tau)$ と、指定された配列の長さ r_{size} の型 $S(n)$ とが同じ n を持つことを検証する。これにより、境界検査が正しい配列の長さに対して行われることを保証できるため、配列アクセスの安全性を保証することができる。

unpack 命令は、existential package をアンパックする命令であり、機械語に変換されるときには単なる mov 命令となる。

Morrisett らによる実際の TAL の実装 [2] では、TAL の型は、variant 型、再帰型、関数のコールスタックの型などによって拡張されており、ML のような強く型付けされた言語から TAL への変換を自動的に行うことが可能である。

3.4 現在の TAL の制限

実は、現在の TAL には実用上、多少問題となると考えられる 2 つの制限がある。しかし、既にそれらに対する研究が行われており、これらの制限は将来解消されると考えられる。以下では現在の TAL の制限とそれに対して行われている研究について述べる。

まず、現在の TAL ではメモリ管理をガーベッジコレクション (GC) に頼っていて、さらに GC が安全であることを仮定しているという制限がある。このため、プログラムの安全性が GC に依存してしまうという問題点がある。この問題に対して、Crary ら [11]、Smith ら [12]、Walker ら [13] は、linear 型や region-based メモリ管理を利用して、直接メモリの確保と解放が行えるように TAL を改善しようとしている。

また、現在の TAL では、配列の境界検査は実行前に行うことはできず、実行時に検査しなければならないという制限がある。このため、プログラムの実行効率が低下する可能性がある。この問題については Xi ら [14] が提案している、dependent 型を追加した TAL である DTAL を用いることにより、不要な配列の境界検査をある程度削除することができる。

4 プロトタイプシステム

4.1 概要

我々は、前節で紹介した Morrisett らによる TAL を用いて、ユーザプログラムをカーネルモードで安全に実行することができるプロトタイプシステムを、IA-32 [15] (x86) 上で Linux カーネルを改造し実装している。

我々のプロトタイプシステムでは、プログラムをカーネルモードで実行させたいプログラマは、まず、プログラムを TAL で直接書くか、もしくは他のプログラミング言語で書きコンパイラを用いて TAL に変換する。次に TAL で書かれたプログラムを TAL アセンブラによって機械語コードに変換する。このとき、TAL アセンブラはプログラムの型情報も同時に生成する。

このようにして作成されたプログラムは、通常のプログラムと同様システムコール execve によって、カーネルモードで実行を開始される。我々のプロトタイプシステムのシステムコール execve はカーネルモードでプログラムを実行する前に、まず TAL の型チェッカを別のプロセスで実行する。この型チェッカ

は、プログラムの機械語コードに対し、TAL アセンブラが生成した型情報を用いて型検査を行い、安全性を検証できた場合のみプログラムをカーネルモードで実行する。

また、我々のプロトタイプシステムのシステムコール execve の行う処理は、プログラムをカーネルモードで実行しようとする点以外は、基本的に元々の Linux と同じである。このため我々のシステムでは、プログラムは特権モードが異なる以外は全く普通のユーザプロセスとして実行される。これにより、メモリのページングやプロセスのスケジューリングなども全く通常どおり行われるため、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することができる。

4.2 実装

我々のプロトタイプシステムはユーザプログラムをカーネルモードで実行する仕組みとカーネルモードで実行しているユーザプログラムからシステムコールを高速かつ安全に呼び出す仕組みの 2 つの機能を実現している。本節では、まずこれら 2 つの仕組みの実装について説明し、それに続いてプロトタイプシステムを実装するにあたって最も大きな障害となった「割り込みスタックの問題」とその解決方法について説明する。

4.2.1 プログラムをカーネルモードで実行する仕組み

まず IA-32 においてカーネルモードとユーザモードの区別がどのように実現されているかについて説明する。なお実際の IA-32 にはカーネルモードとユーザモードという 2 種類ではなく、特権レベル 0 から 3 までの 4 種類の実行モードが存在するが、簡単のため以降では、特権レベル 0 をカーネルモード、特権レベル 3 をユーザモードとして、この 2 つのみを考えることにする。

IA-32 のプログラムカウンタは、レジスタ CS とレジスタ EIP の組からなる。レジスタ CS は、プログラムがどのセグメント中を実行しているのかを示すレジスタである。レジスタ EIP は、レジスタ CS の示すセグメント内のオフセットを格納するレジスタである。IA-32 では、このレジスタ CS が指定するセグメントの設定によって、実行中のプログラムがカーネルモードであるのかユーザモードであるのかが決定される。

よって、プログラムをカーネルモードで実行するためには、単にレジスタ CS が指すセグメントをカーネルモード用のセグメントにするだけでよい。現在の我々のプロトタイプシステムのシステムコール execve が元の Linux のそれと異なるのは、主にこのセグメントを指定する部分である。

4.2.2 カーネルモードで実行中のプログラムからシステムコールを呼ぶ仕組み

まずカーネルはシステムコールごとに、その内部関数のアドレスと、そのアドレスにジャンプする際に満たしていなければならない条件 (引数や返り値) をあらわす TAL の型を、組にして公開しておく。プログラマはこのシステムコールの型と内部関数のアドレスの情報にしたがって、システムコールの内部関数を単なる関数として直接呼び出すプログラムを作成する。カーネルは、プログラムをカーネルモードで実行する前に TAL の型検査を行い、カーネルが公開しているシステムコールの型情報にプログラムが正しくしたがっているかどうかを検証する。以上の方法により、プログラムは単なる関数呼び出しとして高速にシステムコールを実行でき、さらに、安全性も保証できる。

また、現在のプロトタイプシステムでは、システムコールのインタフェイス (引数や返り値の型など) とシステムコールの内部関数は既存の Linux のものを用いている。これにより我々のシステムは、従来の OS が保証している安全性と同レベルの安全性を保証できている。なぜなら、これは結局従来の OS

がカーネルモードとユーザモードの切り替えによって実現してきた安全性の検査を TAL による型検査によって置き換えたのと同じことになるからである。例えば、ファイルアクセスにはシステムコール open、read、write などの内部関数を利用するようにしており、これらの内部関数で正しいアクセス制御が行われているならば (つまり元の Linux カーネルの実装にバグがなければ) ファイルのアクセス制御の安全性を保証できる。

なお、ユーザプログラムへのインタフェイスとして、システムコールの内部関数を用いると述べたが、実際には、スタックの切り替えを行うラッパ関数を用意し、これをユーザプログラムへのインタフェイスとしている。これは、元の Linux のシステムコールの内部関数が、ページフォルトを起こさない特殊なスタックを必要とするからである。また、このラッパ関数は、C 言語と TAL の間の配列や文字列などの表現方式の違いを吸収する役割も果たしている。なお、現在の我々の実装では、ラッパ関数によるオーバヘッドは無視できるくらい小さい。

4.2.3 割り込みスタックの問題

本節では、我々のプロトタイプシステムを実装するにあたって大きな障害となった、「割り込みスタックの問題」について説明する。そのためにまず、IA-32 の割り込み処理のハードウェア仕様について説明する。

IA-32 のハードウェアは、割り込みや例外が発生すると、その時点でのプログラムカウンタであるレジスタ CS とレジスタ EIP、そしてフラグレジスタ EFLAGS を、メモリ上のスタックに保存して、割り込みハンドラを実行する。

ただし、割り込み発生時の実行モードがユーザモードであり、さらに割り込みハンドラをカーネルモードで実行するように指定していた場合には、IA-32 のハードウェアは、まずスタックをカーネルモード実行用のスタックに切り替え、その後でプログラムカウンタであるレジスタ CS とレジスタ EIP、フラグレジスタ EFLAGS、そして元のユーザモードでのスタックを指すレジスタ SS とレジスタ ESP を新しいスタックに保存する。割り込み発生時の実行モードがカーネルモードであった場合には、このようなスタックの切り替えは行われない。

ところが、この通常の割り込み処理の方法では、我々のプロトタイプシステムにおいて問題が生じる。なぜなら、我々のプロトタイプシステムでは、ユーザプログラムはカーネルモードで実行しているので、例えばスタックでページフォルトが起きると、IA-32 のハードウェアはレジスタ EIP、CS、EFLAGS をスタックに保存しようとするが、スタックが切り替わらないために保存できず、ダブルフォルトという例外が発生する。さらにそのダブルフォルトを処理するためのスタックがないために、トリプルフォルト例外となり、CPU が強制的にリセットされてしまう。この問題を、我々は割り込みスタックの問題と名づけた。

この割り込みスタックの問題に対処するために、我々は、IA-32 のハードウェアが提供するタスク機能を用いることにした。IA-32 のタスク機能とは、通常の OS ではソフトウェアで実現しているいわゆるタスク (プロセス) 切り替えをハードウェアが行うという機能である。このタスク機能を使ってページフォルトを処理すれば、たとえスタックでページフォルトが起こっても、IA-32 のタスク機能によって全く新しい別のタスクに切り替えられ、その新しいタスクのスタックを用いてページフォルトを処理できるので、一応割り込みスタックの問題を解決することができる。

ところが、このタスク機能には、実行効率が悪いという大きな問題がある。そのため、全てのページフォルトをタスク機能を使って処理すると、システム全体の効率が悪くなる可能性がある。(このことが、現在の IA-32 上の主流の OS で、割り込み処理にタスク機能が使われない理由の一つである。)

そこで我々のプロトタイプシステムでは、ページフォルトではなく、ダブルフォルトをタスク機能を使って処理することにした。この方法は「ユーザプ

ログラムをカーネルモードで実行しているときにスタックを拡張しようとして生じたページフォルト」のみ、タスク機能で処理するのとほとんど同じであり、システム全体の効率の悪化を小さく抑えられる。

5 実験

我々は、前節のプロトタイプシステムを用いて、実際にユーザプログラムがカーネルモードで安全に実行され、かつ高速にシステムコールを呼び出せることを確かめる実験を行った。具体的には、同一のプログラムをユーザモードとカーネルモードでそれぞれ実行し、その性能を比較した。また、IA-32 では PentiumII 以降、sysenter/sysexit [15] という、システムコールを高速に実行するための特殊命令が存在するので、それらを使った方式とも比較した。sysenter/sysexit は、ユーザモードとカーネルモードの切り替えを、システムコールに特化した方法で行うことにより高速化をはかるものである。

5.1 比較対象プログラム

性能を比較したプログラムは **getpid**、**pipe**、**find**、**echod** の 4 つである。

getpid は、単にシステムコール getpid を呼び出すだけのプログラムである。この **getpid** は、従来の OS におけるシステムコールのオーバヘッドが極端に大きく現れるプログラムである。なぜならシステムコール getpid の行う処理が非常に短いからである。実験ではシステムコール getpid が呼び出されてから終了するまでの時間を測定し比較した。

pipe は、パイプを 1 つ作成し、片方の口からデータを 8 バイト書き込み、もう片方の口からデータを読み出すプログラムである。この **pipe** は簡単なプロセス間通信の性能をどの程度改善できるかを調べるために比較することにした。実験では、データを書きはじめてから読み終わるまでのレイテンシを比較した。

find は、UNIX の find プログラムと同じように、ディレクトリツリーを探索するプログラムである。この **find** は、**getpid** よりも現実的なプログラムの例として比較することにした。実際に探索したファイル数は 35819、ディレクトリ数は 2227 である。なお、実行にあたっては、ディレクトリ情報がメモリ上にキャッシュされている状態で実験を行った。これはディスクアクセスのオーバヘッドによって性能の差が見えなくなってしまうのを防ぐためである。実験ではプログラム全体の実行時間を比較した。

echod は、TCP/IP 上でクライアントから受信したデータをそのままそのクライアントに送信するサーバプログラムである。実験では、クライアントプログラムが **echod** サーバにデータを送信し始めてから返信データを受信し終えるまでの時間を測定し、これをレイテンシとした。また送受信するデータのサイズは 8 バイトとした。**echod** サーバと通信するクライアントは、従来の Linux 上で実行した。

実験を行った環境は表 1 のとおりである。プログラム **find** の実行時間の測定には、bash の time コマンドを利用し、実時間 (real) をプログラムの実行時間とした。一方、プログラム **getpid** の実行時間と、プログラム **pipe** とプログラム **echod** のレイテンシは、rdtsc 命令 [15] を用いて CPU のタイムスタンプカウンタを比較することで測定した。なお、比較したプログラムの作成には、C 言語に似た高級言語である Popcorn [6] を利用した。

5.2 実験結果

表 2 のとおり、**getpid** は 30 倍、**find** は 14%の高速化が実現された。また、**pipe** のレイテンシは 28%改善され、**echod** のレイテンシは 4 マイクロ秒改善された。また、システムコールを高速に呼び出すための特殊命令 sysenter/sysexit を使うよりも我々

	getpid、pipe、find、echod サーバ	echod クライアント
CPU	PentiumII 350MHz	PentiumIII 1GHz
メモリ	256 Mbytes	512 Mbytes
OS	Linux Kernel 2.4.17	Linux Kernel 2.4.17

(ネットワーク : 100Mbps イーサネット)

表 1: 実験環境

の方式の方が高速化の度合いが大きいことがわかる。なお、**echod** のレイテンシの改善の割合が小さいのは、イーサネット上で TCP/IP 通信をする場合には、システムコールだけではなく、割り込み処理やコンテキスト切替え、プロトコル処理なども大きなオーバヘッドになっているためと思わる。(ただし、上述の実験環境では 4 マイクロ秒といえども 1400 CPU サイクルに相当するので、絶対値としては決して小さくない。)

なお、現在の TAL では、配列アクセス時には必ず境界検査が行われるので、プログラムの実行性能が悪化する可能性がある。この可能性を検証するために、プログラム **find** を C 言語で作成し、同様の環境で実行してみた (コンパイラは gcc -O2 を利用)。すると、その実行時間は 148 ミリ秒であり、TAL でプログラムを作成したことによる性能悪化よりも、システムコールのオーバヘッド削減による性能改善の方が大きいことが分かった。これは、プログラム **find** が、システムコールを頻繁に呼び出すプログラムであるためと考えられる。(逆に、配列アクセスが頻繁に行われ、システムコールをあまり呼び出さないようなプログラムは、TAL で作成することによって性能が悪化するかもしれない。そのようなプログラムは、従来どおりに作成し、ユーザモードで実行したほうがよいと現状では考えられる。)

6 研究の今後

6.1 さらなる高速化

現在の我々のプロトタイプシステムでは、Linux のシステムコールの内部関数をそのまま利用しているため、冗長な安全性検査が行われている。例えば、Linux のシステムコールの内部関数は、引数として渡されたメモリのアドレスが、正しいユーザ空間を指しているかどうかの検査を行うが、TAL の型検査を用いれば、引数として渡されたメモリのアドレスが、不正なアドレスを指していないことは保証できるので、我々の実装においては、この検査は冗長である。このように、TAL の型検査を積極的に利用して、実行時の無駄な安全性検査を省くことで、システムを高速化できる。

さらに、TAL では抽象型が利用できるので、これを用いてカーネルの内部データをユーザプログラムから直接アクセスできるように公開することにより、システムの高速化が可能である。例えば、現在の Linux では、ファイル操作を行うシステムコールを実行するたびに、プロセスごとのファイル構造体の配列の検索が行われている。これは、ファイルを指定するためのファイル記述子として、ファイル構造体の配列のインデックスが用いられているためである。TAL の抽象型を用いれば、安全性を損なうことなく、直接カーネル内部のファイル構造体へのポインタをユーザプログラムに渡すことができるため、このようなファイル構造体の配列の検索が不要となり、多少の高速化が実現できる。

もっとも極端な高速化が可能な例としては、プロ

	getpid(ns)	**pipe**(ms)	**find**(ms)	**echod**(μs)
従来の方式	942	4.12	153	200
sysenter	411	3.53	141	200
我々の方式	31	2.96	131	196

表 2: 実験結果 : **getpid** と **find** の実行時間の比較と **pipe** と **echod** のレイテンシ (データサイズ 8 バイト) の比較

セス ID を得るためのシステムコール getpid があげられる。TAL を用いれば、システムコール getpid は、関数呼び出しですらない単なるメモリ参照で実現することができる。その方法は以下のとおりである。まずカーネルはプロセスのプロセス情報をそのプロセスのアドレス空間の固定アドレスにマップし、プロセス ID を保存しているメモリのアドレスだけ、読み込み専用の整数 int 型を保存するメモリのアドレスをあらわす TAL の型を付けてユーザプログラムに公開する。するとユーザプログラムはプロセス ID を単なるメモリ参照で得ることができる。この方法により、システムコール getpid を大幅に高速化することができる。

6.2 ユーザプログラム同士の保護

現在のプロトタイプシステムでは、ユーザプログラム間の保護は従来どおり CPU の MMU を利用して実現している。これを、ユーザとカーネル間の保護と同様に、TAL の型検査で置き換えることも可能である。これにより、ユーザプログラム間の通信の高速化が期待できる。

この場合、問題となるのは、プログラム間のポインタの受け渡しである。例えば、あるプログラム A がメモリを確保し、そのポインタを他のプログラム B に渡したとする。この場合、もし、A が終了した後に、A から渡されたポインタを B が利用すると、不正なメモリアクセスが起きる可能性がある。なぜなら、A の終了とともに A によって確保されたメモリが解放されてしまうはずだからである。この問題に対する最も簡単な解決方法は、1 つアドレス空間を共有している全てのプログラム間でヒープメモリを共有することである。これにより、たとえプログラムが終了しても、ポインタが存在する限り、確保されたメモリ領域が解放されることはないので、不正なメモリアクセスが起きることを防ぐことができる。

6.3 その他

今回我々は、我々の提案方式を Linux というモノリシックなカーネルの OS に適用したが、マイクロカーネル方式の OS に応用することも可能である。例えば、従来のマイクロカーネル方式の OS では、ユーザサーバとマイクロカーネルとの通信のオーバヘッドが問題となってきたが、我々の提案方式を応用すれば、このオーバヘッドを大幅に削減することができるはずである。

さらに、我々の提案方式を IA-32 だけではなく他の CPU に適用することも可能である。現時点では利用できる TAL が IA-32 用のものだけであるが、他の CPU 用の TAL を作成することはそれほど困難ではないと思われる。また、我々の提案方式は MMU によるメモリ保護機構に依存せず同様の安全性を実現することができるので、PDA や携帯電話などの組み込み機器向け CPU への応用も考えられる。

参考文献

[1] Morrisett, G., Walker, D., Crary, K. and Glew, N.: From System F to Typed Assembly Language, *25th ACM SIGPLAN-SIGACT Symposium*

on Principles of Programming Languages, pp. 85–97 (1998).

[2] Morrisett, G., Crary, K., Glew, N., Grossman, D., Samuels, R., Smith, F., Walker, D., Weirich, S. and Zdancewic, S.: TALx86: A Realistic Typed Assembly Language, *ACM SIGPLAN Workshop on Compiler Support for System Software*, pp. 25–35 (1999).

[3] van de Ven, A.: *kHTTPd : Linux HTTP Accelerator*. http://www.fenrus.demon.nl/.

[4] Red Hat, Inc.: *TUX Web server*. http://www.redhat.com/nl/docs/manuals/tux/.

[5] Bershad, B., Savage, S., Pardyak, P., Sirer, E. G., Becker, D., Fiuczynski, M., Chambers, C., and Eggers, S.: Extensibility, safety and performance in the SPIN operating system, *Proceedings of the 15th ACM Symposium on Operating System Principles*, pp. 267–284 (1995).

[6] Cornell University: *Typed Assembly Language software distribution (Popcorn and Scheme--)*. http://www.cs.cornell.edu/talc/releases.html.

[7] Cartwright, R. and Fagan, M.: Soft Typing, *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI)*, pp. 278–292 (1991).

[8] Wahbe, R., Lucco, S., Anderson, T. E. and Graham, S. L.: Efficient Software-Based Fault Isolation, *ACM SIGOPS Operating Systems Review*, Vol. 27, No. 5, pp. 203–216 (1993).

[9] Necula, G. C. and Lee, P.: Safe Kernel Extensions Without Run-Time Checking, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA* (USENIX(ed.)), Berkeley, CA, USA, USENIX, pp. 229–243 (1996).

[10] Tullmann, P., Hibler, M. and Lepreau, J.: Janos: A Java-oriented OS for Active Networks, *IEEE Journal on Selected Areas of Communication*, Vol. 19, No. 3 (2001).

[11] Crary, K., Walker, D. and Morrisett, G.: Typed Memory Management in a Calculus of Capabilities, *26th ACM SIGPLAN-SIGACT Symposium on Priciples of Programming Languages*, pp. 262–275 (1999).

[12] Smith, F., Walker, D. and Morrisett, G.: Alias Types, *European Symposium on Programming*, pp. 366–381 (2000).

[13] Walker, D. and Morrisett, G.: Alias Types for Recursive Data Structures, Technical report, Cornell University (2000).

[14] Xi, H. and Harper, R.: A Dependently Typed Assembly Language, Technical Report CSE-99-008, University of Cincinnati (1999).

[15] Intel Corporation: *IA-32 Intel Architecture Software Developer's Manual*. http://developer.intel.com/.