Vol.2010-OS-113 No.6
2010/1/27

スクリプティング言語によるカーネル拡張

井 出 真 広[†1] 中 田 晋 平[†2] 倉 光 君 郎[†2,†3]

近年、スクリプティング言語を用いたソフトウェア開発が広がっている．しかし、実行パフォーマンスがクリティカルな分野への応用は限定的であり、特にカーネル開発の分野においての利用はなされてこなかった．我々はスクリプティング言語によるカーネルの拡張を目指している．本研究では静的型付けをもったオブジェクト指向のスクリプティング言語 Konoha を用い、言語エンジンをカーネルへ移植し、カーネルの拡張をスクリプティング言語で行った．実験では簡単なキャラクタデバイスをスクリプトで記述し、呼び出しオーバーヘッドを測定した．

A Kernel Extension with Scripting Language

Masahiro Ide,[†1] Shinpei Nakata[†2]
and Kimio Kuramitsu[†2,†3]

Recently, scripting language becomes more likely to use for developing software rapidly, efficiently. For instance, most of famous web applications are implemented with scripting language that is Python, Perl, Ruby, etc. However, system software such as hardware driver is not implemented with scripting language because of its poor performance, poor type strictness. Our goal is to provide an environment for kernel driver scripting. In this paper, we ported a Konoha scripting language runtime inside the kernel and integrated some kernel API binds into Konoha language. We evaluated a runtime performance, and our results showed that it has an achievable performance.

†1 横浜国立大学
Yokohama National University
†2 横浜国立大学大学院
Graduate School of Yokohama National University
†3 科学技術振興機構/CREST
Japan Science and Technology Agency/CREST

1. はじめに

近年、スクリプティング言語を用いたソフトウェア開発が急速に広がっている．代表的な例として、Web アプリケーション開発における PHP や JavaScript などの利用が挙げられる．また近年ではゲーム開発の分野においても Lua 言語などの適用が急速に進んでいる．

スクリプティング言語は、初心者にとってもプログラミングしやすく、型安全性からメモリアクセス違反など深刻なバグをもたらさず、一般に生産性が高いとされている．[9] しかし、実行パフォーマンスがネイティブコードに比べると遅くなるという課題も知られ、パフォーマンスがクリティカルな分野への応用は限定的な状況といえる．

オペレーティングシステム (OS) のカーネルは、実行パフォーマンスが強く要求される代表的なソフトウェアであり、Linux カーネルは C 言語とアセンブラで開発されている．本研究では、我々が開発を行っている新しいスクリプティング言語 Konoha を用いて、Konoha 言語による Linux カーネル拡張、つまりカーネルドライバ開発に挑戦する．カーネル拡張をスクリプティング言語で行うことが可能になれば、スクリプティング言語の新たな応用領域が広がるのみならず、より柔軟に再構成可能な Linux OS の実現が可能になると期待している．

本稿では、最初の試みとして、Konoha 言語エンジンを Loadable Kernel Module(LKM) としてカーネル空間内で動作させたことを報告する．あわせて、簡単なキャラクタデバイスを Konoha 言語によるスクリプトとして記述し、ドライバとして実行する検証を行った．

本稿の構成は、以下の通りである．まず、第 2 節では LKM について述べ、カーネル開発とユーザプログラムの開発環境の違いについて述べる．第 3 節では、スクリプトによるカーネル拡張の全体設計について述べる．第 4 節では、スクリプティング言語 Konoha とカーネルへの言語エンジンの移植について述べる．また、カーネル拡張の手順と実装についても述べる．そして、第 5 節で動作性能の評価結果を述べ、第 6 節では関連研究について述べる．最後に、第 7 節にて本稿を総括する．

2. 問題提起

我々はスクリプトによるカーネル拡張を実現するため、まずのその実行環境となる言語エンジンを LKM としてカーネル空間に移植する．本節では、LKM の仕組みについて概説し、次にカーネル空間とユーザ空間のプログラミングの違いによる問題点をまとめる．

© 2010 Information Processing Society of Japan

2.1 Loadable Kernel Module

Linux はモノリシックカーネルを採用しているが、デバイスドライバなどの一部の機能をモジュールとしてあとから動的に拡張するために Loadable Kernel Module (LKM) を提供している．LKM は insmod など、コマンドベースで簡単に追加登録、削除が行えるため、ファイルシステムなど広くカーネル拡張に利用されている．LKM はカーネル内の関数を利用することができる．特に、カーネルのシンボルテーブルへ明示的に export された関数をカーネル API と呼ぶ．図 1 は簡単なキャラクタデバイスを C 言語で記述した例である．(我々の提案する Konoha 言語による代替コードは後述する．) 図 1 のように、open, read など、デバイスドライバへのシステムコールへのハンドラを記述していくことで、デバイスドライバを記述することができる．

```
int device_open (struct inode* inode, struct file *filp) {
    return 0;
}
ssize_t device_read (struct file* filp, char *user_buf,
        size_t count, loff_t *offset) {
struct device *device = filp->private_data;
    return 0;
}
```

図 1 キャラクタデバイスの疑似コード

2.2 ユーザ空間とカーネル空間の違い

移植するスクリプティング言語エンジンはユーザ空間のアプリケーションである．LKM として実装されたソフトウェアはカーネルで動作するので、カーネル API を用いてプログラミングされる必要がある．カーネル空間におけるプログラミングはいくつかの点でユーザ空間のプログラミングとは大きく異なる．以下は代表的なカーネル空間における制約事項である．

- 浮動小数点演算
- スタックサイズ制限
- ユーザライブラリの利用制限

まず、Linux カーネルは様々なアーキテクチャで動作することを前提としたソフトウェアである．一部のアーキテクチャにおいて、浮動小数点演算命令が用意されていないため、カーネルはオプションとしてソフトウェアエミュレーションによる解決を用意している．しかし、ソフトウェアエミュレーションによる浮動小数点演算はハードウェアによる演算にくらべてコストが高いため、広く配布されるカーネルイメージはソフトウェアエミュレーション機能を標準では利用しない．このため、一般的なカーネル内では浮動小数点演算を行うことができない．次に、Linux の仮想メモリのレイアウトは、ユーザ空間の 3GiB に対し、カーネル空間は 1GiB である．カーネル空間の仮想メモリでは大部分をコード領域が占める上、多様な割り込み処理で、スタック消費も激しい．そのため、使用できるカーネルスタックは 4KiB に抑えなければならないという制限が設けられている．最後に、libc などのライブラリをカーネル空間では用いることができない．このため、printf, assert など、多くの標準ライブラリ関数は使えないため、代替の手段を用意する必要がある．

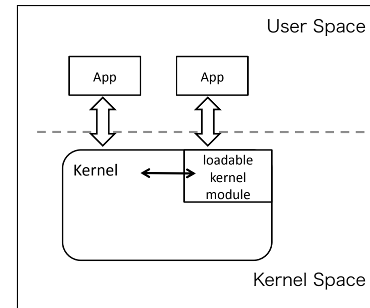

図 2 LKM のアーキテクチャ図

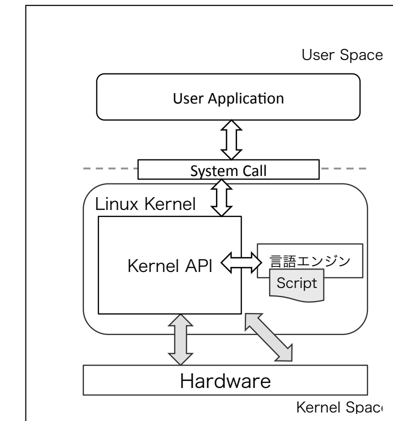

図 3 スクリプトによるカーネル拡張のアーキテクチャ図

3. Konoha 言語

Konoha 言語[7] は、我々の研究チームが言語設計から実行処理系まで開発を行っているスクリプティング言語である．本研究では、Konoha 言語のスクリプティング言語エンジンを Linux Kernel Module に移植した Konoha LKM を開発した．本節では、Konoha 言語について紹介を行う．

© 2010 Information Processing Society of Japan

3.1 概 要

Konoha は、Java 言語スタイルの文法をもったオブジェクト指向スクリプティング言語である．言語的には次のような特徴をもつ．

- Java スタイルの名前ベースのクラスシステム
- VM によるマルチプラットフォーム動作
- 単一継承と簡単な総称型
- 委譲による関数オブジェクト化
- 対話的なプログラミング実行環境

Konoha 言語は、現在、バージョン 0.7 が公開されている．C 言語によって実装が行われ、Linux や Mac OS X など、UNIX 系 GCC 開発環境のみならず、Windows 環境 (Visual C++) や TRON 系の OS (T-Kernel) など、様々なプラットホームによる実行が報告されている．

3.2 型システムの特徴

スクリプティング言語は、ダイナミック言語ともよばれるとおり、代表的な言語はほとんどすべて動的な型付けを採用している．これは、コンパイルエラーによってコーディング＆テストランのサイクルが中断されないメリットがあるが、反面、些細なケアレスミスであっても実行するまで機械検出できないことを意味する．そのため、カーネルドライバのような、たとえクラッシュしなくても実行停止すると困る用途には適用しにくい．

これらに対し、Konoha は静的な型付けを特徴としており、バイトコンパイル時に型検査を行うことができる．そのため、実行前に型エラーを検出し、単純なケアレスミスによって実行が停止することを防ぐことができる．

また、Konoha 言語では、静的な型付けであっても、スクリプティング言語らしくプログラミングが可能なように Any (dynamic) 型、run anytime コンパイル技法、さらに型推論などの機構を備えている．これらの詳細は、[6] に詳しい．

3.3 Konoha VM: 実行処理系

Konoha スクリプトは、一旦、バイトコードにコンパイルされ、Konoha 言語の実行処理系である Konoha Virtual Machine (VM) 上で実行される．Konoha VM は、専用の命令セットをもったレジスタ型の VM であり、静的な型付けによるコード生成の効率化とあわせて、スクリプティング言語としては高速な実行処理を実現している．表 1 は、代表的なスクリプティング言語との fibo(36) ベンチマークによる実行速度の比較をまとめたものである．

Konoha VM の実装上の特徴は次のとおりである．

言語 (バージョン)	fibo(36)
perl(5.8.9)	48.1s
python(2.6.4)	15.4s
ruby(1.9.1)	6.6s
konoha(0.7)	2.2s

表 1 Fibonacci 数 (36) を計算したときの実行時間 (s) の比較．
実験環境:CPU Intel Core2 2.00GHz、メモリ 2GB,OS Mac OS X 10.5.8

- 専用オブジェクトアロケータ
- 複数 VM のサポート
- リファレンスカウンティング方式のガベージコレクタ
- setjmp/longjmp による例外処理

Konoha VM の大きな特徴は、スクリプティング言語として eval による動的な実行をサポートするため、パーサからバイトコンパイラの機能がバーチャルマシンと不可分になっている点である．そのため、最新の Konoha VM のソースコードサイズは、60,000 行程度の大きさになる．

3.4 言語バインド

Konoha は、C 言語の構造体へのポインタをクラスインスタンス、それを用いた関数をメソッドとして、簡単にスクリプトから呼び出すことができる．これらの機能を言語バインドと呼ぶ．

以下は、Konoha 言語のバインドによる C 言語の sin 関数をメソッド化した例である．

```
METHOD Math_sin (Ctx *ctx, knh_sfp_t *sfp ) {
float arg = sfp[1].fvalue; //  スタックから引数を取得
float ret = sin(arg); // sin 関数から結果を格納
KNH_RETURN(ctx, sfp, ret); // Konoha へ計算結果を Float として返す
}
```

4. 実 装

本節では、Konoha LKM の実装を述べる．

4.1 Linux カーネルへの移植

2.2 節で述べた問題について以下のように対応した．まず浮動小数点演算の使用について述べる．Konoha 言語にはプリミティブ型として浮動小数点 (Float) 型が用意されているが、カーネル内では原則使用することができないため、Konoha 言語から取り除くことで対応し

© 2010 Information Processing Society of Japan

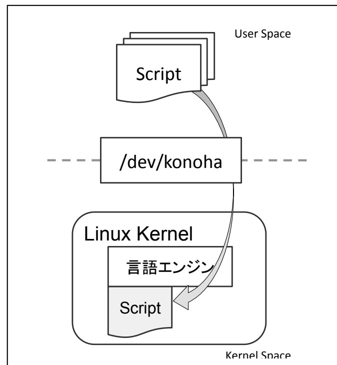

図 4 カーネルスクリプティングのアーキテクチャ図

	KernelAPI	libc API
文字出力	printk()	printf()
文字列比較	strncmp()	strncmp()
メモリ割当	kmalloc()	malloc()
メモリ解放	kfree()	free()
lock	mutex_lock()	mutex_lock()
assertion	BUG_ON()	assert()

表 2 KernelAPI と libc API との対応表

た．次にスタックサイズの制限については、カーネルに付属する `checkstack.pl` などを用いてスタックサイズを調べたところ、特に問題となるようなメモリ使用箇所はなかった．最後に、ライブラリ API が使用できない問題については表 2 に示す通り、カーネル API として libc に対応する関数がカーネルから提供されている．そのため、カーネル API と libc 関数との対応を作ることで解決した．また、対応する API が存在しない以下の関数については libc から移植を行って対応した．

- qsort
- strerror

4.2 カーネル空間とユーザ空間とのインタフェース

カーネルの拡張を記述したスクリプトをカーネル内で実行するにあたって、ユーザ空間に存在するスクリプトをカーネル空間に送る必要がある．本稿ではカーネルのインタフェースとしてキャラクタデバイスを作成し、カーネル空間とユーザ空間の通信を行うことにした．カーネルスクリプトはすべてキャラクタデバイスである `/dev/konoha` を通してカーネルに送られる．この様子を図 4 に示す．

以下に `/dev/konoha` を用いたカーネルの機能をスクリプトにて追加する過程を示す．まず、カーネル拡張スクリプトを `/dev/konoha` へ書き込む．

```
$ cat device_script_file > /dev/konoha
```

これによりデバイスを通じてスクリプトがカーネル空間へ移動される．カーネル空間側では受け取ったスクリプトは、Konoha コンパイラによって VM コードへとコンパイルが行われ、実行可能な状態になる．その後、カーネルスクリプトで記述された機能がカーネルより呼び出されたとき、言語エンジンがコードの実行を行う．

5. 評　価

本節では Linux カーネルへ Konoha 言語の言語エンジンを組み込み、キャラクタデバイスを Konoha 言語で実装した．また C 言語と Konoha 言語で実装したキャラクタデバイスについて呼び出しのオーバーヘッドについて計測した．

5.1 評価環境

評価は、 Intel Core2 Duo CPU E8400 3.00GHz のプロセッサ上で行った．利用したソフトウェアは次の通りである．使用した Linux カーネルは GNU/Linux 2.6.32-rc6 i686、コンパイラは gcc version 4.3.4 を利用した．また、システムコールのベンチマークには lmbench-3.0-a9[10] を用いた．

5.2 スクリプトによるカーネル拡張記述

Konoha 言語で実装したデバイスはカーネルからのシステムコールの処理を行うために C 言語で実装した関数を用意する必要がある．open システムコールは図 1 に示す device_open 関数で実装される．open システムコールが発行された場合の実行の流れを以下に示す．open システムコールが発行されるとまず、device_open 関数がカーネルから呼び出される．次に、file 構造体に割り付けられた Konoha 言語のキャラクタデバイスのオブジェクトを取得する．そして、キャラクタデバイスに対する open メソッドを探し、Konoha 言語で実装されたメソッドを実行する．

```
struct device {
struct cdev cdev;
konoha_object object;
}
```

図 5 Device クラスの構造体

また、図 6 に Konoha で記述されたデバイスドライバのソースコードを示し、図 5 に Device クラスの構造体を示す．カーネルのキャラクタデバイスドライバは open, read など、デバイスに対するシステムコールへのハンドラとなる関数ポインタを要素として持つ．この cdev 構造体を定義し、カーネルへ登録することでデバイスは動作可能となる．キャラ

© 2010 Information Processing Society of Japan

クタデバイスは Konoha 言語において Device クラスとして扱われる．Konoha はコンパイルが終わると、read システムコールのハンドラとして Device.read メソッドが呼び出されるように、cdev 構造体の確保、カーネルへのキャラクタデバイスの登録などデバイスを使用するために必要な操作を行う．

```
void Device.open() {
 /* open システムコールが呼ばれた際の処理 */
}
void Device.read(OutputStream out) {
 /* read システムコールが呼ばれた際の処理 */
}
```

図 6 スクリプトによるデバイスドライバの記述

5.3 オーバーヘッドの測定

Konoha 言語のメソッドの呼び出しオーバーヘッドを測定するため、C 言語、Konoha 言語それぞれで実装したキャラクタデバイスへの open システムコールの呼び出しのオーバーヘッドを測定した．

	C 言語	Konoha	Overhead
open	1.66msec	2.55msec	153.6%

表 3 open システムコールのオーバーヘッド
Table 3 Overhead of open system call

測定結果を表 3 に示す．C 言語で実装されたものと比べ、Konoha 言語で実装した場合の open システムコールの呼び出しオーバーヘッドは 53%程度であった．本実装ではスクリプトのメソッドを動的に検索し、メソッドの呼び出しを行っている．動的にメソッドの探索を行っているため、オーバーヘッドが生じたものと考えられる．

6. 関 連 研 究

本稿ではプログラミング言語の言語エンジンをカーネル内に移植することでカーネル拡張を行っているが、同様の手法を取っている研究として奥村ら[2] の研究がある．奥村らはカーネルに組込むための軽量な Java 言語の VM として NVM を開発している．NVM は Just In Time コンパイラを搭載しており、実行パフォーマンスも考慮している．しかし、奥村らの研究ではネットワークを抽象化したフレームワークを用いてカーネル拡張を行うため、C 言語などで実装された既存のコードを再利用することが難しい．しかし、本稿ではスクリプティング言語でのカーネル拡張を可能にしているので、既存のコードをスクリプティング言語にバインドすれば利用でき、さらに拡張することも可能である．

Linux のメインラインに統合されているカーネルの拡張機能として FUSE（Filesystem in Userspace）[5] がある．FUSE は独自のファイルシステムをユーザ空間で開発できる機能として Linux バージョン 2.6.14 から利用可能である．FUSE ではカーネルからユーザ空間へインタフェースを提供しており、これによってユーザ空間でのファイルシステム開発を可能としている．また、この FUSE を拡張する形で、キャラクタデバイスをユーザー空間で実装できる CUSE（Character devices in Userspace）が Linux カーネル バージョン 2.6.31 から利用できる．これらの手法の利点として、ユーザ空間で利用されている既存のライブラリやデバッガなどを使用できることがあげられる．しかし、FUSE,CUSE を用いて拡張されたカーネルの機能はユーザ空間で動作するため、これらの機能を使用し、ファイル等にアクセスを行う場合、通常のカーネル内部にあるファイルシステム、キャラクタデバイスの機能を利用する場合に比べて 2 倍のコンテキストスイッチが発生するため、パフォーマンスの面で不利である．我々の手法で作成したカーネル拡張はカーネル空間で動作するため、ユーザ空間、カーネル空間の遷移が不要である．また、CPU などのハードウェアにて特権命令を発効することができるなどユーザ空間での開発では行えないような拡張を開発することも可能である．

7. 結　　論

本稿では、スクリプティング言語 Konoha による柔軟なカーネル拡張を目標に、言語エンジンをカーネルへ移植し、スクリプトによるカーネル拡張方法を提示した．キャラクタデバイスを作成し、スクリプティング言語でカーネルの拡張を行うことが可能であることを示した．我々の行った実験ではスクリプトで記述した機能の呼び出しオーバーヘッドは 50%以上となった．今後の予定として、呼び出しのオーバーヘッド削減を含むスクリプティング言語自体の高速化と、カーネル API のバインドの自動化を目指して開発を進めていく．

© 2010 Information Processing Society of Japan

7.1 スクリプティング言語の高速化

スクリプティング言語でカーネルの拡張を記述していく場合は言語の実行速度が大きな問題となる．そこで我々は Context Threading[4] をベースにした Just In Time(JIT) コンパイラを作成し、スクリプティング言語 Konoha の実行速度の改善を行なっている．今後はカーネル内にこの JIT コンパイラを移植し、カーネル空間での実行速度の向上を目指す．

7.2 カーネル API のバインド

現在、カーネル API を Konoha 言語から用いる場合、カーネル API とスクリプトとのインタフェースは手作業で作成している．しかし、カーネル API は膨大な数があり、またカーネルのバージョンごとに API のインタフェースは変わっていくため、現状では全てに対応することが難しい．そこで言語間のインタフェースの自動生成[11] などの考えを取り入れ、機能の効率的な拡張を可能とすることを目指す．

謝辞 本研究は、JST/CREST「実用化を目指した組込みシステム用ディペンタブル・オペレーティングシステム」領域の研究課題「実行時の安全性を確保する SecurityWeaver と P-SCRIPT」の一部として行われた．

参 考 文 献

1) Daniel Plerre Bovet, Marco Cesati, Bovet Daniel, Cesati Marco, *Understanding the Linux Kernel, Third Edition* O'Reilly Media, 2005
2) Takashi Okumura , Bruce Childers , Daniel Mosse, *Running a Java VM Inside an Operating System Kernel*, Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, Pages 161-170, 2008
3) Richard P. Spillane, Charles P. Wright , Gopalan Sivathanu, Erez Zadok *Rapid file system development using ptrace*, Proceedings of the 2007 workshop on Experimental computer science
4) Marc Berndl , Benjamin Vitale , Mathew Zaleski , Angela Demke Brown, *Context Threading: A Flexible and Efficient Dispatch Technique for Virtual Machine Interpreters*, Proceedings of the international symposium on Code generation and optimization, p.15-26, March 20-23, 2005
5) FUSE: Filesystem in Userspace "http://fuse.sourceforge.net/"
6) 倉光君郎 , Konoha: ハイブリッドな型検査システムを備えたスクリプティング言語 第 10 回プログラミングおよびプログラミング言語ワークショップ (PPL2008) ,March 2008
7) Konoha scripting language "http://konoha.sourceforge.jp/"
8) Feng Zhou , Jeremy Condit , Zachary Anderson , Ilya Bagrak , Rob Ennals , Matthew Harren , George Necula , Eric Brewer, *SafeDrive: safe and recoverable extensions using language-based techniques* , Proceedings of the 7th symposium on Operating systems design and implementation, November 06-08, 2006, Seattle, Washington
9) John K. Ousterhout, Scripting: Higher-Level Programming for the 21st Century, Computer, IEEE, 1998
10) LM Bench Project, "http://sourceforge.net/projects/lmbench"
11) Tristan Ravitch , Steve Jackson , Eric Aderhold , Ben Liblit, *Automatic generation of library bindings using static analysis* ,Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation, Pages 352-362

© 2010 Information Processing Society of Japan