2007－OS－106 (12)
2007／8／3

PerlOSの試作と評価

浅野 一成 [†] 並木 美太郎 [††]

[†] 東京農工大学工学府情報工学科 [††] 東京農工大学共生科学技術研究院システム情報科学部門

本報告では，Microperlを用いたPerlOSの試作とその評価について述べる．Microperlを拡張し，I/Oポートや実メモリへアクセスを行う機構を取り入れ，Perlメソッドコールの API を使うことで，Perlで割込み処理やデバイスドライバが記述できることを示す．今回は試作段階として，デバイスドライバと割込み処理を扱った．ドライバと割込み処理の実装は，キーボードについて行った．評価では，I/Oポートの入出力時間と割込みハンドラの起動時間について，CとPerlでの実行時間を比較した．その結果，Cに対するPerlのI/Oポート入出力時間は，最大で85%，割込みハンドラの起動時間は17%の性能比となった．各種ドライバやファイルシステムの実装・評価が，今後の課題である．

A Prototype and Evaluation of PerlOS

Asano Kazunari [†] Namiki Mitarou [††]

[†] Faculty of Engineering, Tokyo University of Agriculture and Technology

[††] Division of Systems and Information Technology, Institute of Symbiotic Science and Technology, Tokyo University of Agriculture and Technology

This paper describes a design and evaluation for perl interpreter running on a bare machine. We propose that Extented-Microperl can access to I/O port and physical memory, and that perl program provieds device drivers and interrupt handlers using Perl method call API. This time, the updated interpreter provides device drivers and interrupt handlers. Keyboard driver and handler has been implemented. In evaluation, we have compared access time to I/O port and uptime to interrupt handler with C. As a result, each performance is 85 percent and 17 percent as fast as C code. Other device drivers and file systems are prospected.

1 はじめに

プログラミング言語Perl(以下Perl)は元来，テキスト処理のために作成された言語であるが，現在ではその枠組みを超えて，Webアプリケーション・管理スクリプト・フィルタプログラム等，多くの場面で利用されている．その理由としては，多くの標準ライブラリ，言語仕様の柔軟さが，ソースコードの記述速度を上げ，生産性の向上につながっているからであると考えられる．多くの人々によって利用されることにより，言語仕様やライブラリは，充実しており，動作の安定性も保証されている．その結果，数多くのテキストプログラム，WebサーバのCGIプログラムなどが開発されてきた．

本報告では，OSが搭載されていないハードウェア上でPerl処理系を動作させ，元来CやC++で実装が行われていた資源管理をPerlで行う．以下，このOSをPerlOSと呼ぶ．これまで多くの人々に利用され，安定したPerlの標準ライブラリや言語仕様を用いることで，OSコードの記述に柔軟性を持たせる．また，OS呼び出しのオーバヘッドがないPerlの性能を検証する．

本報告では，オペレーティングシステム(以下OS)をPerlプログラムで記述するための方法を提案する．PC/AT互換機上にPerl処理系で動作するPerlプログラムによるOS，PerlOSを実現させる．実現には，これまで多くの人々に利用され，安定したPerlの標準ライブラリや，Perl5から導入されたオブジェクト指向による記述を用いる．今回は試作段階として，OS内部のデバイスドライバ部分について扱う．デバイスドライバ内でCPUの周辺機器を制御するための機構，機器からの割込み処理について設計を示し，実装・評価を行う．

2 既存の構成と処理系の問題点

OS内部に言語処理系を搭載し，OSを動作させるものがいくつか存在する．以下にそれらの構成と処理系の問題点を示す．

2.1 JavaOS

JavaOS[1] は，組込み機器のプラットフォームとして作られたものである．JavaVM を搭載し，プログラミング言語 Java(以下 Java) で OS のほとんど部分 (デバイスドライバ・ウィンドウシステム・プロトコルスタック) が書かれている．ハードウェア層に近い部分 (ページング・スレッド・割込み) では，C で書かれたマイクロカーネルが動作する．商用システムであることから，x86 以外に 5 つの主要な CPU に対応している．しかし，現在 Sun は，この OS をレガシーシステムとして使わないように勧告し，代わりに API の集合である JavaME を使うように推奨している．

2.2 Movitz

Movitz[2] は，組込み機器に Lisp プログラミング環境を構築するために作られたものである．Lisp で書かれた Common Lisp コンパイラと実行環境を搭載し，OS が Lisp で書かれている．対象とするアーキテクチャは，x86 となっている．このシステムの特徴としては，ホスト環境上に Lisp コードから x86 のバイナリを生成するコンパイラを持ち，すべての OS コードを Lisp で記述できるという点にある．また，作成された OS 上でも Lisp の実行環境を持ち，エディタ emacs で Lisp を使ったプログラミングができる．しかし，Movitz が実装した Lisp コンパイラは，アセンブラが x86 の機械語のみに対応しているなど，x86 以外のアーキテクチャへの変更が考慮されておらず，OS コードの移植性に欠ける．

Perl の場合，Microperl をコンパイルするのに用いるコンパイラを別のターゲットアーキテクチャ用に変えればよい．Microperl は C 言語で実装されており，x86 以外のアーキテクチャをターゲットとする C コンパイラは多数存在する．また，Perl で書かれたコード自体は，アーキテクチャ独立であるため，移植性は保たれる．

2.3 CooS

CooS[3] は，カーネルレベルでセキュリティ保護を実現するために作られたものである．Microsoft 社により開発され，仕様が公開されている CLI の実装したものを搭載し，C#で書かれた OS が動作する．このシステムの特徴としては，CLI を OS の実行環境に持つことにより，カーネルレベルでの動作不良に対して，体系的な検査が行うことにより，カーネルのセキュリティ保護ができるという点にある．しかし，現段階ではガベージコレクタが未実装であり，メモリ解放を意識しなければならず，プログラマに負担がかかる．

PerlOS では，Microperl[4] を採用することにより，これまで実用的に用いられてきたガベージコレクタを利用する．PerlOS のプログラマはメモリ解放を意識しなくてよい．参照されなくなった変数はリファレンスカウンタ方式によりメモリ回収される．Perl インタプリタが終了する際には，Mark and Sweep 方式で，パッケージ内の変数群がメモリ回収される．また，先に述べたように Perl の仕様は柔軟であり，プログラマが明示的にメモリ回収を行う関数で，その変数のメモリ回収を強制的に行うこともできる．

3 本研究の目標

本研究では，Perl で OS を記述し，PC/AT 互換機上のハードウェア上で直接動作させることを目標とする．最終的な目標として，以下に挙げる資源管理を Perl を用いて記述し，PerlOS 上に実装する．

- デバイスドライバ
- 割込み処理
- スレッド
- ファイルシステム
- TCP/IP プロトコルスタック
- ウィンドウシステム

今回は試作段階として，デバイスドライバと割込み処理の設計・実装に取り組み，Perl でデバイスドライバと割込み処理が記述可能であることを示す．コンテキスト退避・復帰処理を除いて，可能な限り Perl で記述する．Perl でデバイスドライバの実装する際には，Perl の標準ライブラリを用いて，実装の効率化をはかる．

Perl の処理系には Microperl を用いる．Microperl は，Perl をビルドする目的で作成された．ビルド段階で用いるシェルスクリプト・awk・sed・grep の代わりに Microperl を用いることで，Perl のビルドをいくつかの CPU に移植可能にした．そのような目的で作られたためため，Microperl は，WindowsCE や PalmOS が搭載されている多くの組込み機器で動作する．また，移植性を考えたつくりであるため，ソースコードの規模が小さく，修正を行いやすい．Windows や Linux で配布されるフルバージョンの Perl のソースコードが，1700 以上のファイルから構成されるのに対し，Microperl は，約 80 のファイル

から構成される．コード量が減っても，Perl のコア機能 (正規表現・Unicode サポート等) は一部のシステム依存の組込み関数を除いて，すべて利用可能である．本研究では，開発の効率を考え，Perl の処理系に Microperl を採用する．また，Microperl を採用することにより，移植性の向上を目指す．本報告では PC/AT 互換機をターゲットアーキテクチャとしているが，他アーキテクチャにも移植可能にする．

PerlOS が CPU の周辺機器の制御や割込み処理を扱う際，実メモリ・I/O アドレス空間にアクセスが必要となる．しかし，Microperl の処理系には，それらの資源にアクセスする機能がないため，実メモリ・I/O アドレス空間へのアクセス機構を追加する必要がある．追加するにあたって，Microperl の仕様拡張を行う．追加した機能は，パッケージ化して，Perl プログラムで利用できるようにする．パッケージから利用できる API は，手続き型・オブジェクト指向等，状況に応じて柔軟に設計する．

4 全体構成

本研究が目標とするシステムの全体構成を図 1 に示す．大きく分けて (1) から (3) の 3 つの部分から構成される．

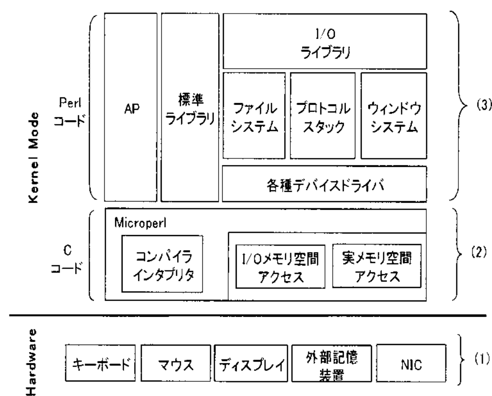

図 1　システムの全体構成

(1) ハードウェア

CPU の周辺機器として，キーボード・マウス・ディスプレイ・外部記憶装置・NIC を扱う．キーボード・マウスはユーザからの入出力インタフェースとなる．ディスプレイは，PerlOS からユーザへの出力インタフェースとなる．ディスプレイの出力方式としては，キャラクタベースの CUI と，ウィンドウシステムを用いた GUI に対応する．外部記憶装置は，PerlOS が扱うファイルの内容を保存するために用いる．NIC は，PerlOS が TCP/IP プロトコルを使って，ネットワーク上の他のマシンとデータをやりとりするために用いる．

(2) Perl 処理系

Perl の処理系には Microperl を利用する．Microperl は C 言語で書かれ，上位層にあるカーネルコードである Perl プログラムを解釈・実行する．Windows や Linux 用に配布されている Perl 処理系は，様々な拡張機能が搭載しており，ソースコードの規模も，バイナリサイズも大きい．研究開発のしやすさを考慮し，最低限の機能を搭載した Microperl を Perl の処理系として採用した．

Microperl には，実メモリ空間・I/O アドレス空間にアクセスするための拡張が施されている．実メモリアクセス空間にアクセスすることにより，メモリマップドされたハードウェアを操作する．また，ハードウェアからデータが DMA で実メモリ上に転送された際に，実メモリから，Perl メソッドコール API を使って Perl プログラム内にその内容をコピーする．I/O アドレス空間は，Perl からレガシーデバイスを操作する場合に利用する．利用方法は，メモリマップドされたハードウェアと同様となる．

将来的には，複数の Microperl のインスタンスが図 1 に示した各ブロックごとに動作するようにする．複数のインタプリタのインスタンスが動作している様子を図 2 に示す．

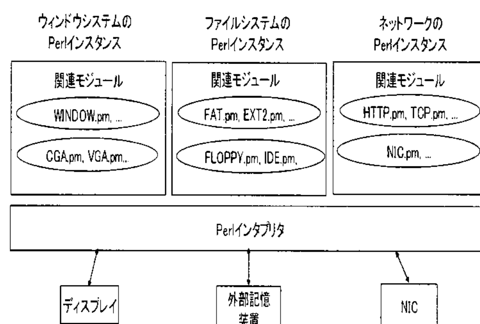

図 2　複数インタプリタのインスタンスの動作

(3) Perl プログラム

この層からはすべてPerlプログラムとなる．Perlプログラムは，カーネルプログラムとしてMicroperl上で動作する．この層で動作するプログラムは，標準ライブラリ・アプリケーションプログラム・デバイスドライバ・ウィンドウシステム・プロトコルスタック・ファイルシステム・IOライブラリとなる．

標準ライブラリは，WindowsやLinuxで配布されているPerlに付属しているPerl標準ライブラリのサブセットを用意した．用意した標準モジュールは他の標準モジュールや標準モジュール以外のコードに多用されており，必要最低限なものとなる．表1に標準ライブラリの一覧と機能を示す．

表1 標準ライブラリ一覧

パッケージファイル名	機能
strict.pm	コンパイル時に変数宣言やベアワードの使用等のチェック
warnings.pm	コンパイル時に変数の未定義値の代入や数値型のチェック
vars.pm	グローバル変数の宣言
base.pm	クラス継承
overload.pm	演算子のオーバーロード
constant.pm	定数の宣言
Carp.pm	各種メッセージ出力
Exporter.pm	変数の名前輸出

デバイスドライバは，Perlで書かれる．CPUの周辺機器を操作し，ウィンドウシステム・プロトコルスタック・ファイルシステムにターゲットデバイスの制御コードを提供する．それら3つについて，デバイスドライバと対応するハードウェアを以下に示す．

- ウィンドウシステム：I/Oポート経由でディスプレイを制御し，CUIまたはGUI環境の実現する
- プロトコルスタック：I/Oポートと実メモリアクセスでNICを制御し，ネットワークを介したデータのやりとりを行う
- ファイルシステム：各種外部記憶装置のディスクドライバを制御し，ファイルやディレクトリの読み書きを管理する

I/Oライブラリは，ウィンドウシステム・プロトコルスタック・ファイルシステムへopen・read・write・seek等のAPIを提供する．この層を通じることにより，入出力装置の仮想化を行う．アプリケーションで入出力を行うプログラムを書く際は，Perlの組込み関数を使わず，それらのAPIを使うことを推奨する．

5 Microperl仕様拡張

5.1 拡張の必要性

一般に，デバイスドライバを作成する際には，メモリマップド形式やI/O命令で周辺機器の制御を行う．またDMAによる実メモリ上へのデータ転送を扱うことになる．Perlでデバイスドライバを作成する場合も，それらの手段や手続きを取ることになるが，既存のPerl処理系には，I/O命令が組み込み関数で用意されていない．また，Perlにおけるアドレス空間は，実アドレスとは異なるため，Perlにおいて実アドレスとデータをやり取りするための仕組みがない．そこでMicroperlを拡張し，(1) I/O空間へのアクセス，(2) 実メモリ空間へのアクセスを実現させる．

5.2 拡張方式の検討

これまで述べたように，Microperlには拡張が必要である．Perlを拡張する方法としてXS[5]が考えるが，Microperlでは使えない．それ以外の拡張方式は2つ考えられる．1つ目は，Microperlの組込み関数に新たに，前節の(1)と(2)で示した機能を付け加えるやり方である．この方式の場合，Microperlのコンパイラと，インタプリタの両方に手を加えなければならない．2つ目は，既存の組込み関数を拡張する方法である．前節で述べたように，Microperlでは，システム依存の組込み関数が無効になっており，この関数の仮想マシン中の動作を，前節の(1)と(2)に書き換えてしまうことができる．本報告では，試作段階ということで，実装効率を重視し，コンパイラに手を加えない2つ目のやり方でMicroperlを拡張することにした．

Microperlでは，移植性を考慮して，システム依存の組込み関数がいくつか無効になっている．無効になっているものには，ディレクトリ操作関数(opendir・readdir等)やプロセス操作関数(fork・wait・pipe等)やシステムコール関数がある．システム依存のために無効となっている関数のうち，今後も使われないであろうものを検討する．本研究ではMicroperlのコードは特権モードで動作する．そのため，ユーザからカーネルへのシステムコールがない．無効になっている関数のうち，システムコールを行うsyscall関数がある．この関数を使って，I/O空間と実メモリ空間へのアクセス機構の拡張を行う．syscall関数で拡張されたMicroperlを図3に示す．

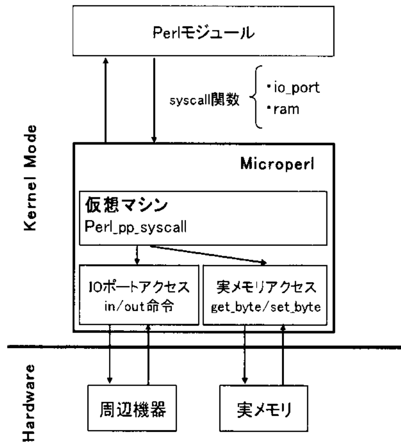

図 3 拡張された Microperl

5.3 拡張機能以外の API

本報告で利用する Microperl が C 言語で提供している API を示す．これらの API は,

(1) Perl インタプリタインスタンス生成・破壊

(2) Perl プログラム内の変数操作

(3) Perl プログラム内のサブルーチン・メソッドの呼出し

の 3 つに分類される．以下に，(1) から (3) について示す．

(1) インタプリタインスタンスに対する操作

Perl インタプリタインスタンスに関する操作 API を表 6 に示す．

表 2 インタプリタのインスタンス操作 API

関数名	機能
perl_alloc	インタプリタの生成・メモリ確保
perl_construct	perl_alloc 確保したメモリ内部情報の初期化
perl_parse	Perl プログラムのコンパイル
perl_run	Perl インタプリタの起動・実行
perl_destruct	メモリ内部情報の破壊
perl_free	メモリ解放

これらの API は，マシンがブートした後に，Perl のインタプリタインスタンスを生成する際に利用する．

(2) プログラム内の変数操作

Perl プログラム内の変数操作に関する API を表 3 に示す．これらの API は，(1) の perl_alloc で生成されたインタプリタインスタンスに作用する．

表 3 プログラム内の変数操作の API

関数名	機能
newSViv	Perl プログラム内の整数の生成
newSVpv	Perl プログラム内の文字列の生成
sv_setiv	newSViv で生成した値の変更
sv_setpv	newSVpv で生成した値の変更
get_sv	Perl プログラム内のスカラ変数の取得
newAV	Perl プログラム内の配列の生成
av_push	newAV で生成した配列に値の追加

これらの API は，インタプリタインスタンス内に存在する Perl プログラムの変数操作を行うときに用いる．

(3) プログラム内のサブルーチン・メソッドの呼出し

Perl プログラム内のサブルーチン呼出しに関する API を表 4 に示す．これらの API は，(1) の perl_alloc で生成されたインタプリタインスタンスに作用する．

表 4 サブルーチン・メソッド呼出しの API

関数名	機能
call_argv	Perl プログラム内のサブルーチンの実行
call_method	Perl プログラム内のメソッドの実行

これらの API は，インタプリタインスタンス内に存在する Perl プログラムのサブルーチン・メソッドを呼び出すときに用いる．

5.4 拡張機能の API

Perl でデバイスドライバを作成するにあたり，(1)I/O 空間へのアクセス，(2) 実メモリ空間へのアクセスを実現するために，Microperl に拡張を施した．拡張された機能は，モジュール化され，パッケージとして提供される．以下，(1) と (2) について，拡張した機能を利用するための API の仕様と，使用例を示していく．

(1) I/O メモリ空間へのアクセス

組込み関数 syscall を改変し，バイト単位で I/O ポートにアクセスする inb・outb を実装した．syscall

関数のままでは使い勝手が悪いため，IO_PORT モジュールとしてパッケージ化した．IO_PORT モジュールでは，関数名が Export モジュールで輸出されており，組込み関数のように利用できる．in・out 命令の API を表 5 に示す．

表 5　I/O ポートへのアクセス API 一覧

機能	関数名	引数	返り値
ポート番号から値を受信する	inb	ポート番号	値
ポート番号に値を送信する	outb	値，ポート番号	なし

この API を利用したプログラミング例を図 4 に示す．

```
use IO_PORT;

# I/O ポート 0x123 からデータの受信
my $value = inb (0x123);
# I/O ポート 0x345 に 0x10 の転送
outb (0x10, 0x345);
```

図 4　IO_PORT パッケージによるプログラミング例

wait 時間は，in/out 命令に含まれていないので，必要ならば別途用意する．

(2) 実メモリへのアクセス

組込み関数 syscall を改変し，バイト単位で実メモリにアクセスする get_byte・set_byte を実装した．これらの関数は，RAM モジュールとしてパッケージ化した．パッケージ RAM の API を以下に示す．

この API を利用した例を図 5 に示す．以下の例では，テキスト VRAM への描画を行っている．

6　割込み処理

6.1　処理の流れ

割込み処理は，アセンブラ言語で書かれた処理から開始する．まずアセンブリ言語のコードにより，コンテキストスイッチのためのレジスタ退避処理を完了させる．それより先の処理は Perl により行う．Perl のメソッドコール API(call_method) により，Perl コードが実行を開始する．Perl インタプリタ上で，Perl コードで記述された割込み要因となったデバイスへ

```
use RAM;

# オブジェクト生成
my $ram = RAM->new(start_addr => 0xB8000);
# 文字コード
$ram->set_byte('a');
# 色コード
$ram->set_byte(0x0F);
```

図 5　RAM パッケージによるプログラミング例

の ack・割込み要求のマスク・割込みハンドラの実行が行われる．Perl インタプリタの実行が終わると，アセンブリ言語によりレジスタ復帰処理と，iret 命令の発行を行う．

6.2　割込み処理の API

Perl で割込み処理を行う際に，割込みコントローラを制御する必要がある．割込みコントローラを制御するモジュールは，PIC モジュールとしてパッケージ化した．PIC モジュールの API を表 6 に示す．

表 6　割込み処理 API

機能	関数名	引数	返り値
割込みコントローラへの ack と割込み要求マスク	mask_and_ack	マスクする信号線番号	なし
割込み要求のマスクを解除	unmask	マスク解除する信号線番号	なし
割込みハンドラの登録	regist_handler	ハンドラとして実行する Perl プログラムメソッドのアドレス	なし

7　デバイスドライバの実装

現在までにキーボードのデバイスドライバを実装した．実装したキーボードドライバにおける割込みハンドラを図 6 に示す．

このハンドラは，Perl のメソッドコール API で，アセンブリ言語で書かれた割込みエントリルーチンから呼び出される．割込みエントリルーチンは，IDT の IRQ1 ハンドラの部分に登録されており，キーボード割込みが発生すると，Perl のキーボードハンドラ

```
package DEV::KEYBOARD;

use IO; # I/O モジュール
use IO_PORT; # in/out 命令
use PIC; # 割込みコントローラ

our $map =
{ # キーマップは省略
};

sub handler
{
    my $self = shift;
    # ir1 信号のマスクと ack
    PIC::mask_and_ack (IR1);
    # スキャンコードの取得
    my $scan_code = inb (SCAN_CODE);
    # スキャンコードからアスキー文字への変換
    my $string = $map{$scan_code};
    # テキスト VRAM に文字出力
    IO::print ($string):
    # ir1 信号のマスク解除
    PIC::umask (IR1);
}
```

図 6　キーボード割込みハンドラのコード

が実行される．

今回は試作段階ということで，Perl インタプリタのインスタンスは，メモリ上に 1 つだけ生成し，実行させる．インスタンス内には，インスタンス生成後に走るメインプログラムと，デバイスドライバモジュールが内蔵されている．このインスタンスは，ブート後，ハードウェアのセットアップが終了した後に生成され，メインプログラム部分が実行される．外部機器からの割込みが発生すると，IDT に登録されたハンドラにより，割込み要因に対応したデバイスドライバモジュールの割込みハンドラが実行される．キーボード割込み処理の流れを図 7 に示す．

割込み前の状態が (1) である．この状態では，インスタンス内では，メインプログラムが実行されている．キーボード割込みが発生すると，CPU により割込み禁止状態となり，他の機器からの割込みは保留状態となる．その後，IDT にあるハンドラがコンテキストの退避を行い，インスタンス内のキーボードデバイスドライバの割込みハンドラを呼び出し，(2) の状態になる．この状態では，インスタンス内で，キーボードドライバの割込みハンドラが実行されている．ドライバの処理が終わり，コンテキストが復帰すると，割込み禁止状態が解除され，(1) の

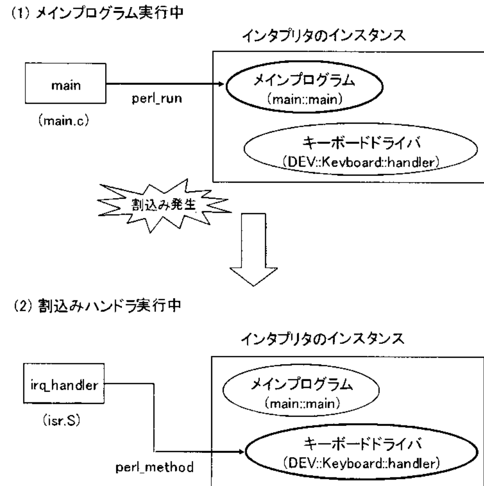

図 7　キーボード割込み処理

状態に戻る．インスタンス内では，割込み直後の状態からメインプログラムが再開される．

8　評価

実装したキーボードドライバについて，割込みハンドラ起動までの時間と，in/out 命令の実行時間を測定し，C と Perl で比較評価を行った．評価にはクロックカウント数を取得する rdtsc 命令を用いて，Celeron の 1.7GHz の実機で実行した．

I/O ポートの入出力時間と割込みハンドラの起動時間の結果を，C と Perl についてそれぞれ表 7 に示す．

表 7　各種測定結果

測定対象	C	Perl
inb	2182	6523
outb	1912	5136
割込みハンドラの起動時間	4721	29997

各表の数値は，処理を行う前後のクロックカウント数となる．ただし，各表の数値には測定用の関数のオーバヘッドが含まれている．測定用の関数のオーバヘッドは，C と Perl についてそれぞれ表 8 に示す値となった．

測定関数のオーバヘッドを除いた状態の，Perl の C に対する性能比は表 9 に示す．

I/O ポートの入出力は最大で 85%の性能が出てい

表 8 測定関数のオーバヘッド

言語	クロック数
C	84
Perl	3002

表 9 Perl と C の性能比

測定対象	性能比 (%)
inb	59.5
outb	85.7
割込みハンドラの起動時間	17.2

る．一方で，割込みハンドラの起動時間は17%の性能となっている．この性能差が出た理由としては，Perl モジュールのメソッド呼び出しのオーバヘッドが考えられる．I/O ポートの入出力関数は，組込み関数として定義して利用する一方，割込みハンドラは，キーボードドライバモジュールのメソッドとして呼び出す．Perl メソッドを呼び出すときに用いる call_method には，キーボードドライバクラスのインスタンスを格納した変数を必要とする．この変数を Perl 内部のスタックに積み，call_method を呼び出す．このため，call_method を呼び出すまでの手続きに時間がかかると推測される．さらに詳しくコストを見積もるためには，割込みハンドラを手続き呼び出しにしたときとの比較を取ることが必要となるが，今回は試作段階であるため，次回の設計の際に詳しく調査する．

9 おわりに

本報告では，PerlOS の試作とその評価について述べた．Microperl への拡張を施すことにより，Perl によるデバイスドライバの開発が可能となった．今回の実装はキーボードドライバのみであったが，さらに実装を進めていくことにより，各種ディスクドライバやファイルシステムを Perl で実現させていく．

ディスクドライバ・ファイルシステム・I/O ライブラリの未実装部分を完了させ，評価を行う．ディスクドライバ・ファイルシステムについては，C ではすでに実装・動作確認済みであり，それらを Perl に書き換える．また，現在，Perl プログラムは 1 つのインタプリタのインスタンスで動作している．インスタンスには，メインプログラムとデバイスドライバのコードが混在している．そのため，割込みが発生した際のメインプログラムの実行内容によっては，割込み処理を行った後，メインプログラムの状態が復帰できない可能性がある．複数のインタプリタのインスタンスを動作させ，ドライバコードの処理を分離して，誤動作を防止する．

参考文献

1) トム・サルポー, チャールズ・ミロ著, 油井 尊 訳 : インサイド JavaOS オペレーティングシステム, ピアソン (1999)

2) Frode V. Fjeld : Movitz, `http://common-lisp.net/project/movitz/movitz.html`

3) 高橋 明生, 加藤 和彦 : CIL で表現された OS カーネルの実行方法. 情報処理学会研究報告 2006-OS-101

4) Simon Cozens : Microperl, The Perl Journal Volume 5, Number 3 (#19), Fall 2000

5) perldoc : perlxs, `http://perldoc.perl.org/perlxs.html`