Vol.2010-ARC-189 No.4
Vol.2010-OS-114 No.4
2010/4/21

RubyによるOS構成法の提案とその実行基盤の試作

吉原陽香[†1]　笹田耕一[†2]　並木美太郎[†3]

本研究では，Ruby で記述された OS である RubyOS の構成法の提案と，その実行基盤の試作を行った．また RubyOS の試作として，ポーリング方式でキーボードの入力を受け取り，画面にその文字を出力するプログラムを Ruby にて記述した．実行基盤については，既存の Ruby 処理系を OS を搭載していないハードウェア上で直接実行できるように移植した．OS を記述するために必要となる，実メモリアクセス・I/O ポートアクセスといった Ruby の言語仕様にない機能は，拡張ライブラリを自作して実装した．評価としていくつかのプログラムの実行時間の計測を行い，Ruby が OS を記述するのに十分な機能を持っているか検討した．

Basic Design of Operating System Construction with Programming Language Ruby and Prototype of its Basic Execution Environment

Haruka YOSHIHARA,[†1] Koichi SASADA[†2]
and Mitaro NAMIKI[†3]

This paper describes basic design of the 'RubyOS' operating system with programming language Ruby and its VM(Virtual Machine) for the RubyOS. For building RubyOS,the CRuby interpreter was revised and executed directly on a PC/AT compatible machine without OS. The Ruby extension libraries were implemented for physical memory and I/O port access in Ruby. In evalution, The CRuby interpreter performances were mesured on PC/AT compatible machine without OS and with Ubuntu 9.04. As the result, this Ruby environment has performance enough to build operating system.

1. はじめに

従来，ベアマシン上で直接実行される OS は C 言語（以下，C）やターゲット CPU のアセンブリ言語で記述されてきた．その理由には，それら言語によってポインタ演算を用いてアドレス操作を行うことが可能であり，またインラインアセンブラを利用してアセンブリ言語をソース中に埋め込むことができるなどといった利点があることが挙げられる．これらによって，C で書かれた OS は計算機の資源管理を行いやすくなっている．その反面，不正なメモリアクセスやメモリリークが起きやすいなど，プログラマから発見しにくいバグの原因を生む可能性も含んでいる．これらのバグは，OS の実行停止や，場合によってはデータの破壊などといった計算機の保護を脅かすものにもなりかねない．

一方，スクリプト言語はインタプリタや仮想マシン（以下，VM）で中間コードを実行し，動的型付け，テキスト処理や不要になったメモリの管理，C にはない便利なデータ構造・制御構造を言語仕様としてもっている．この際，インタプリタや VM でプログラムを実行するためにランタイムコンパイル時のオーバーヘッドが生じる．しかし，メモリリークやデータ構造の作成といった，OS のプログラミングの本質からは遠い部分にプログラマが手を煩わせることがなくなるため，生産性，安全性の向上のメリットは大きいと考えられる．

そのスクリプト言語の一つである Ruby[3] はまつもとゆきひろによって開発された．Ruby はすべての値，また手続きなどもオブジェクトとして扱い，統一的に扱う．その一方，手続き型のプログラミングも可能であり，どんな局面でも二つのプログラミングスタイルをプログラマが自由に選ぶことができる．また変数宣言が必要なく，さらに変数に型がない動的型付けの仕様のため，変数の型についてプログラマが考慮する必要がない．

さらに，C により拡張ライブラリを記述すると，Ruby 処理系の機能を拡張することができる．拡張ライブラリは既存のものも存在し，その種類の豊富さ，また Ruby の言語仕様の柔軟さなどから多くの人々に利用される．また UNIX，Windows など多くの OS 上で動

†1 東京農工大学 工学部 情報工学科
Faculty of Engineering, Tokyo University of Agriculture and Technology.
†2 東京大学大学院 情報理工学系研究科 創造情報学専攻
Department of Creative Informatics, Graduate School of Information Science and Technology, The University of Tokyo
†3 東京農工大学大学院 共生科学技術研究院
Institute of Symbiotic Science and Technology, The Graduate School at Tokyo University of Agriculture and Technology.

く移植性の高さも特徴である．バージョン 1.9.0 からは，インタプリタ YARV[1] が正式に組み込まれた．現在，Ruby は Ruby on Rails を始めとした Web アプリケーションなどの様々なアプリケーションの開発に用いられている．

本報告では，上記で述べた従来の C による OS 実装の欠点をできる限り排除して OS を構成する，Ruby で記述された OS（以下，RubyOS）の設計について述べる．また，その RubyOS 実装のために必要となったハードウェア上で直接実行される Ruby 処理系の試作について述べる．PC/AT 互換機上に Ruby の実行基盤を作成し，その上で動作する OS の設計，試作を行う．Ruby のオブジェクト指向の記述法や手続き型の記述法を用いて，適所には既存の拡張ライブラリを利用することで効率よく RubyOS の開発を行うことを目標とする．

第 2 章では本研究の関連事項から既存の問題点を検討し，第 3 章では本研究の目標を述べる．第 4 章では目標となる RubyOS の設計について示す．第 5 章にて現在までに実装できた RubyOS について述べた後，第 6 章ではその実装した部分についての評価，考察を行った．最後に，第 7 章にて現在までの成果と今後の課題を挙げる．

2. 関 連 研 究

ここでは，OS を搭載していない計算機（以下，ベアマシン）上で C 以外の言語を直接実行した例を示し，それらの問題点から RubyOS の実行基盤や設計について考察する．

M3L マシン[2] は，まず LEM と呼ばれるマイクロプログラミング言語に対応するハードウェアを利用し，このハードウェア上に LISP のインタプリタを LEM にて実装する．そしてこのインタプリタ上で直接 LISP を実行する．また LEM でのデータの実体はセルで表され，このセルをガーベージコレクションするための機構はハードウェアにて実装されている．この方法は，一部の資源管理をハードウェアで，またあるマイクロプログラミング言語によって処理系を実装し，その上であるプログラミング言語を動作させるという方式である．ただし，Ruby 処理系をこの方法で実装する場合，あるハードウェアに対応するマイクロコードにて処理系を新たに実装する手間が生じる．現在，Ruby の処理系としては C で書かれた処理系などがあるため，それらを利用することで大幅に実行基盤の作成の手間を省くことができると考えられる．

Sun Microsystems が開発した JavaOS[4] は，JavaVM をベアマシン上に搭載し，Java バイトコードを実行することで実装されている OS である．開発マシンのバイトコードコンパイラにて JavaOS のソースコードをバイトコードにコンパイルし，それをベアマシン上の JavaVM にて実行させる．Java のソースコードや Java バイトコードの規格は OS に非依存のため，ターゲットアーキテクチャによらず開発が可能である．しかし，開発マシンにて Java ソースコードをコンパイルし，生成したバイトコードをベアマシン上に搭載する必要がある．RubyOS の場合，OS コードはテキスト形式のまま処理系に実行させる形となるため，コンパイルの必要がない．

OCOS[8] とは，関数型言語 ML の方言である OCaml にて記述した OS である．OCaml は関数型言語であるがループ文も用意されており，同時にオブジェクト指向でもあり，ガーベージコレクションなどの機能や型検査の機能ももつ．ハードウェア上にて C やアセンブラで記述されたランタイムシステムと I/O アクセス層が動作し，OCOS はそれらを利用して動作する．ランタイムシステムは OCaml を動作させるため，I/O アクセス層は OCaml では記述できない I/O ポートへのアクセスやメモリアクセスを OCOS が行うために用意されている．現時点では，割り込み処理とプロセス管理が実装されている．メモリ管理は物理メモリの確保・解放が行えるようになっている．OCOS はユーザプログラムとして a.out 形式のものが実行でき，記述言語は問わない．このため，OCaml にて a.out 形式の解釈を行う必要がある．物理メモリや I/O ポートへのアクセスができない点は Ruby も OCaml と同様である．C で記述された Ruby 処理系を利用する場合，C にて拡張ライブラリを実装することでこれらアクセスが Ruby にて記述可能になると考えられる．

PerlOS[5],[6] は，インタプリタ言語の Perl を用いて浅野一成が開発した OS である．Perl 処理系の構築途中で作られる処理系の microperl を用いて Perl プログラムの解釈，実行を行い，プロセス管理・ファイルシステム・デバイスドライバ・外部割り込み処理を実装した．Perl の言語仕様にない実メモリアクセスや I/O ポートアクセスは，XSUB を利用して Perl 処理系を拡張した．この拡張機能は PerlOS 上のユーザプログラムは利用することができない．そのため，PerlOS のプロセスは他プロセスのメモリにアクセスすることができないことから，メモリの仮想化は行わず，実メモリ空間を割り当てる．またプロセスに関しては，Perl インタプリタのインスタンスをカーネルが生成し，一つのインスタンスにつき一つのプロセスを割り当てることで実現している．このプロセス上で，ユーザプログラムとして Perl のプログラムを実行することができる．この方式の場合，Perl で書かれたプログラムそのものはアーキテクチャに依存しない点は Ruby と同様である．しかし，実アドレス空間を使うことで，プロセスが増大した場合にメモリ不足が起こる可能性がある．この場合，仮想アドレス空間を用いることでその欠点が回避できると考えられる．

Vol.2010-ARC-189 No.4
Vol.2010-OS-114 No.4
2010/4/21

3. 目　　標

本節では，本研究の目標，RubyOS の構成について述べる．

3.1 本研究の目標

RubyOS は，C にて実装されていた OS が行う計算機資源管理を Ruby を用いて行う．C で記述されるプログラムに見られるようなメモリリークなどの問題を回避し，また Ruby の言語仕様にすでに備えられている機能を利用することで，安全性や生産性，拡張性に優れた OS を目指す．また，Ruby の言語仕様や拡張機能を利用し，見通しのよい構成をもつ OS を目標とする．この RubyOS には，以下に挙げる機能を実装することを目指す．

- ファイルシステム
- プロセス・スレッド管理
- デバイスドライバ
- メモリ管理
- 割り込み処理

ここで，RubyOS を搭載した x86 の PC/AT 互換機を RubyMachine と呼ぶ．RubyOS が実行するユーザアプリケーションプログラム (以下，AP）は，すべて Ruby で記述されたものになる．つまり，Ruby で記述された OS 上にて Ruby で書かれた AP を実行することが最終的な目標となる．ただし，ハードウェアに強く依存する割り込み例外発生時のコンテキストの退避や復元，MMU の操作，I/O レジスタの操作には C のライブラリを利用して記述する．他の資源管理は可能な限り Ruby で記述し，その実行は VM で行う．

3.2 システムの全体構成

RubyMachine の全体設計を次の**図 1** に示す．

次にこれらの構成要素について概略を述べる．

3.2.1 ターゲットハードウェア

ターゲットアーキテクチャは PC/AT 互換機の x86 とする．また CPU の周辺機器としてキーボード，マウス，ディスク，ディスプレイを扱う．キーボードとマウスはユーザからの入力インタフェース，ディスプレイはユーザへの出力インタフェースとなる．外部記憶装置はハードディスクに対応し，RubyOS が扱うデータの情報を保存するために利用する．

3.2.2 Ruby 処理系

Ruby 処理系は，Ruby プログラムの解釈・実行を行う．今回，Ruby の処理系には既存の C 言語で記述された言語処理系を用いる．この処理系は，www.ruby-lang.org で配布されている公式な処理系である．

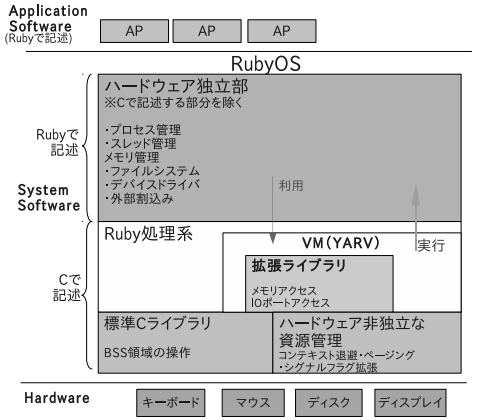

図 1　RubyMachine の全体構成

本研究では，上記の Ruby 処理系をコンパイルする際に用いられる miniruby を利用する．miniruby とは，完全な Ruby 処理系のビルド途中で構築される最小単位の Ruby 処理系である．miniruby は，一部のエンコーディング以外と拡張ライブラリのロード機能は含まないものの，それらを利用しない Ruby のプログラムなら満足に実行できる．さらにプラットフォームに非依存である点は完全な処理系と同様である．以上から，研究開発のしやすさを考え，本研究では miniruby を Ruby の処理系として採用した．

また，ベアマシン上で miniruby が独立して動作するため，OS を持たない計算機用の標準 C ライブラリをリンクする．この標準 C ライブラリのうち，例えば物理メモリアドレスの操作といった計算機の資源を操作する関数をベアマシン向けの資源操作に書き換えることで，既存のプラットフォームで記述された Ruby プログラムをそのまま RubyMachine に搭載した Ruby 処理系でも実行できる．

3.2.3 拡張ライブラリ

Ruby の言語仕様にない機能を用いたい場合，ハードウェア操作のための拡張ライブラリを別途作成して利用できるようにする．この拡張ライブラリを利用すると Ruby プログラム内で使用できるクラスやメソッドの追加を行うことができ，また Ruby 処理系そのものの内部仕様の改変を行わない．そのため自分で記述した拡張ライブラリのほかに，既存の Ruby 拡張ライブラリを利用することができる．

Vol.2010-ARC-189 No.4
Vol.2010-OS-114 No.4
2010/4/21

表 1 RubyOS による計算機資源管理の概要

機能	説明
プロセス管理	RubyVM インスタンス一つにプロセスを割り当てる
スレッド管理	Ruby のスレッドを利用する
メモリ管理	仮想メモリ空間で BSS 領域をプロセスごとに割り当てる
ファイル管理	PStore ライブラリを利用しオブジェクト単位で扱う
デバイスドライバ	ドライバごとにオブジェクトとして記述する
外部割込み	Ruby 処理系のシグナルフラグを拡張して対応する

作成した拡張ライブラリは，miniruby を構成するソースコードと同様にコンパイルした後，静的リンクして利用する．

3.2.4 Ruby 処理系で動作させるプログラム

前節の Ruby 処理系上にて，計算機の資源管理を行う OS，すなわち RubyOS を実行する．この RubyOS は Ruby で記述されたものとなる．RubyOS がもつ機能について概要を示す．

- プロセス・スレッド管理：Ruby 処理系の VM を複数ベアマシン上で実行し，AP を平行に実行する
- メモリ管理：VM を複数動作させるのに適した物理メモリ空間の制御やオブジェクトのメモリ管理を提供する
- ファイルシステム：オブジェクトを適切に管理し，AP にデータを提供する
- デバイスドライバ：CPU の周辺機器を Ruby プログラムにて操作する
- 割り込み処理：Ruby 処理系の外側から入った割込みを，Ruby プログラムにて制御する

それぞれの機能の設計に関しては第 4 章にて示す．これら機能によって計算機の資源が適切に操作され，Ruby で記述された AP を実行することを目指す．

4. RubyOS の設計

本節では，RubyOS の各機能の設計，その設計を実現するための記述言語の検討，そして拡張ライブラリの設計について述べる．各資源管理の概要は次の**表 1** のようになっている． 次にそれぞれについて述べる．

4.1 プロセス管理

プロセスは,「CPU が割り当てられて Ruby の AP を実行する」ものと定義する．Ruby で記述された AP は Ruby の VM のインスタンスにて実行される．また RubyOS そのものも VM のインスタンスにて実行される．RubyOS を動かす VM のインスタンスを OSVM, プロセスに割り当てられた VM のインスタンスをプロセス VM と呼ぶ．この OS やプロセスと VM インスタンスの対応関係は，一対一の対応とする．この方式の場合，プロセスの数に応じてメモリ使用量が増大する．しかし，アドレス空間が独立するため，一つのプロセスのエラーが他のプロセスの実行を妨げることはなくなる．

4.2 スレッド管理

このプロセス内にて用いるスレッドは，Ruby の Thread クラスを利用する．Ruby の Thread クラスは，バージョン 1.9 からはカーネルレベルでスレッドの管理を行うネイティブスレッドをサポートしている．この Ruby のスレッドを利用するには，スレッドライブラリを Ruby 処理系にて扱う必要がある．RubyOS では，開聞[7] という本研究室で開発されている組み込み用 OS のタスクの機構を利用する．

4.3 メモリ管理

メモリ管理はプロセスに割り当てる物理メモリの割り当てや解放を行う．前節で述べたように，RubyOS では VM インスタンス一つを一つのプロセスに割り当て，プロセス VM とする．このプロセス VM がそれぞれ別個にプログラムの実行を行う．プロセスや OS のメモリ使用状況が競合しないように，OS と全プロセスそれぞれに異なるアドレス空間を割り当てる．ただし，Ruby などのスクリプト言語ではヒープ領域が大きくなる傾向があるため，メモリレイアウトは仮想アドレス空間としページング機構を用いる．この割り当ての概要を**図 2** に示す．

OSVM やプロセス VM の BSS 領域の操作については，C ライブラリ関数によって操作される．さらに，不要になったオブジェクトは Ruby 処理系によるガーベージコレクションにて収集される．Ruby 処理系にあるガーベージコレクションは，マークアンドスイープ方式によるものであり，この機能により不要になったオブジェクトを回収する．

4.4 ファイルシステム

RubyOS のファイルシステムは，オブジェクトをディスク上に保存し適切な管理を行い，ユーザプログラムから扱えるようにする．そこで，オブジェクトをバイナリ化し，ディスク上に保存する方法を取る．オブジェクトのバイナリ化やファイルシステムの構築には，Ruby の拡張ライブラリである PStore を用いる．PStore ライブラリは，Ruby 処理系に標準で添付されるライブラリの一つであり，Ruby のオブジェクトをバイナリデータに変換し外部に保存するためのものである．PStore ではある一つのキーに対し一つのオブジェクトを割り当て操作を行う．このライブラリは Ruby で書かれており，内部で Ruby のクラスである Marshal クラスを用いてオブジェクトのバイナリ化を行っている．PStore ライブラリは既

Vol.2010-ARC-189 No.4
Vol.2010-OS-114 No.4
2010/4/21

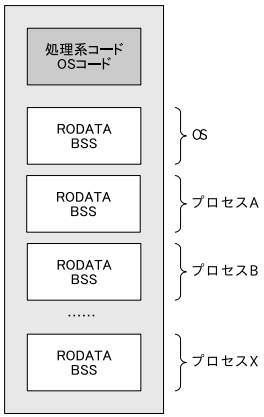

図 2 メモリ割り当ての概要

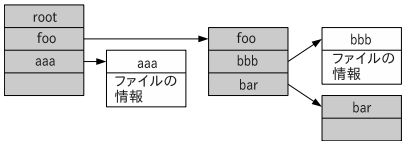

図 3 ファイルシステムの構成

存の OS がもつファイルシステムに依存しているため，その依存部分をベアマシン上でも動作するように修正することで利用する．この PStore ライブラリを利用しディレクトリやファイルを表現することで，階層的なファイルシステムを構築する．その様子を**図 3** に示す．なお，この図 3 のブロックそれぞれが PStore オブジェクトを表し，それぞれのブロックの一番上の文字列がキーとなる．

また，複数プロセスから無秩序にファイル操作が行われないように，RubyOS によってファイルに対する排他制御が行われる．排他制御には，Ruby のクラスである File の flock メソッドにて行う．

4.5 デバイスドライバ・割り込み制御

デバイスドライバは，キーボードなどの CPU の周辺機器を制御するための Ruby プログラムである．デバイスドライバはそのデバイスごとにクラスとして Ruby で記述し，メモリアクセスや I/O ポートアクセスは拡張ライブラリにて実装した機能を用いる．デバイスドライバのクラスの初期化は RubyOS が起動した際に行われる．

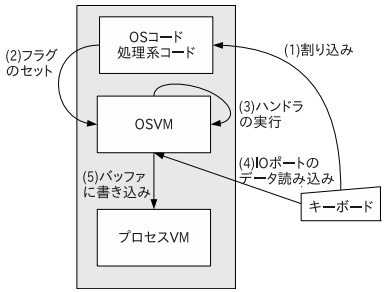

図 4 キーボードによる割込み発生時の流れの例

外部割り込みが入った際には，メモリに設定された割り込み関数テーブルより割り当てられた関数が実行される．この関数はコンテキスト退避を行った後，Ruby 処理系に拡張されたシグナルフラグを設定し，処理系がどの割込みが入ったかを判断できるようにする．シグナルフラグは Ruby の処理系によりスレッドのタイマーなどでチェックされ，フラグが設定されていた場合はそのフラグに対応するシグナルに設定されたハンドラを実行する．ハンドラの設定は Ruby にて行い，外部機器に対応するデバイスドライバが処理を行うように設定する．**図 4** にキーボードからの割り込みが入った際の処理の流れの例について示す．

4.6 RubyOS の記述言語の検討

第 3 章でも述べたとおり，RubyOS では可能な限りすべての資源管理機能を Ruby にて記述することを目的としている．しかし，これまでに見てきたように，メモリアクセスなど，資源管理に必要なもので Ruby の言語仕様に含まれていない機能が必要になるため，すべてを Ruby だけで記述することは現実的には不可能である．また，レジスタの退避など，Ruby 処理系の動作を中断させるような動作も RubyOS には必要となるため，その部分についても Ruby を用いて記述することはできない．これらの動作については C かターゲットアーキテクチャのアセンブラを利用して記述する．また，メモリアクセスや I/O ポートアクセスに関しては，C を利用して拡張ライブラリを記述し，Ruby 処理系を拡張する．以上のことをまとめると**表 2** のようになる．

この表 2 以外の部分についてはすべて Ruby にて記述する．

4.7 拡張ライブラリ

RubyOS では，Ruby の言語仕様では提供されないメモリアクセス，I/O ポートアクセス

Vol.2010-ARC-189 No.4
Vol.2010-OS-114 No.4
2010/4/21

表 2 C で記述する箇所

内容	具体例
Ruby 処理系の動作を妨げる	・コンテキスト退避 ・BSS 領域の操作 ・仮想アドレス空間実装のためのページング
Ruby の処理系への操作	・シグナルフラグの拡張・セット ・メモリ・I/O ポートアクセス（拡張ライブラリ） ・Ruby 処理系内部で扱うスレッドライブラリ

表 3 クラス Memio がもつクラス

機能	引数	メソッド名
メモリからの読み取り	メモリアドレス	readb, readw, readl
メモリへの書き込み	メモリアドレス, 値	writeb, writew, writel
I/O ポートからの受信	I/O ポートアドレス	inb, inw, readl
I/O ポートへの送信	I/O ポートアドレス, 値	outb, outw, outl

```
#アドレス 0x12345 から 256 バイト分を tmp に保存
256.times{|x|
  tmp += Memio.readb(0x12345 + x)
}

#VRAM メモリへの書き込み
Memio.writew((0x0F << 8 ) | 'a'.bytes.to_a[0], 0xB8000)
```

図 5 メソッド readb,writew の使用例

の機能を拡張ライブラリを用いて提供する．作成する拡張ライブラリの名前は Memio とする．Memio は OSVM の特異クラスとし，またインスタンスを作成できないようにしておく．クラス Memio の持つメソッドを次の**表 3** にまとめる．

また，それぞれの使用例を次の**表 5** と**表 6** に示す．

5. RubyOS の実装

本章では，本研究で実現した RubyOS の実行基盤と，その実行基盤上での Ruby スクリプトの実行確認のために作成した Ruby プログラムについて述べる．現時点で，第 3 章で示した図 1 のうち，実装できた部分は次のようになる．

```
#キーコードを読み取り対応する文字を表示
key_map ={...}
key_data = Memio.inb(0x60)
print key_map[key_data]

#VRAM の (x,y) の位置にカーソルを表示
Memio.outb(0x0e, 0x03d4)
Memio.outb((cursor >> 8), 0x03d5)
Memio.outb(0x0f, 0x03d4)
Memio.outb(cursor, 0x03d5)
```

図 6 メソッド inb,outb の使用例

- 標準 C ライブラリによる BSS 領域の操作
- キーボードの割り込み関数テーブルの設定
- miniruby のベアマシン上への移植
- メモリアクセス・I/O ポートアクセスを行うための拡張ライブラリの実装
- Ruby による，キーボードドライバ・テキスト VRAM ドライバの実装

5.1 実現環境

RubyOS を搭載する計算機の仕様について次の**表 4** に示す．

5.2 現時点で実現された RubyMachine の構成

現時点までに実現された RubyMachine の全体構成を次の**図 7** に示す．

現時点では，ベアマシン上での Ruby 処理系の移植を行い，RubyOS の実行基盤の作成を行った．さらに，RubyOS の試作としてキーボードドライバ・VRAM ドライバの実現を行った．それぞれの詳細については次節以降で述べる．

5.3 Ruby 処理系

本節では，Ruby 処理系として採用した miniruby の実装について述べる．

Vol.2010-ARC-189 No.4
Vol.2010-OS-114 No.4
2010/4/21

表 4 RubyMachine の実現環境

アーキテクチャ	PC/AT 互換機
CPU，メモリ	Intel celeron 2.66GHz，1.25GB
エミュレータ	QEMU
Ruby 処理系	ruby-1.9.1-p129
C 標準ライブラリ	newlib 1.6
ブートローダ	GNU GRUB
フロッピーディスク（ブートデバイス）	3.5 インチ 1.4MB
キーボード	PS/2 接続

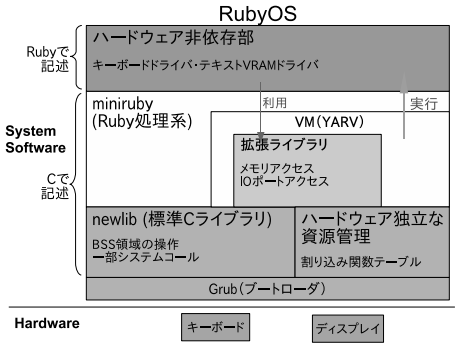

図 7 試作した RubyMachine の全体構成

5.3.1 ベアマシン上への移植

Ruby 処理系をベアマシン上で動作させるためには，標準 C ライブラリも実装する必要がある．RubyOS では，組み込み用標準ライブラリの newlib を用い，Ruby 処理系とのリンクを行った．また，miniruby のソースコードに newlib と関数のインタフェースが異なる部分があったため，関数 open の引数などの該当箇所を newlib の側に合わせるように修正した．newlib のシステムコール呼び出し部分に関しては，BSS 領域の管理や C レベルでのテキスト VRAM の書き込みといった資源管理の操作に対応させた．

5.3.2 拡張ライブラリ

本節では，Ruby 処理系の拡張のために作成した拡張ライブラリの実装について述べる．第 4.7 節で述べたように，Ruby 処理系の拡張ライブラリとしてクラス Memio を作成した．メモリアクセスを行う read・write メソッドについてはポインタ演算，I/O ポートアクセスを行う in・out メソッドはインラインアセンブラを用いて記述した．

また，実装した機能の評価のため，CPU クロック数を計測する x86 のアセンブラ命令である rdtsc を用いて，拡張ライブラリ Clock を作成した．

5.4 RubyOS の実装

本節では，現在の RubyOS にて実装された部分について述べる．

5.4.1 ブートプロセス

Ruby 処理系と，その Ruby 処理系を実行するカーネル部分は GRUB によってブートデバイスからメモリ上にロードされる．その後，ブート用のデバイス上に置かれた RubyOS をメモリ上のヒープ・スタック領域以下にロードする．ロードし終わると，カーネル部分がハードウェアの初期化を行い，Ruby 処理系の初期化を行った後，RubyOS を Ruby 処理系にて実行する．

5.4.2 Ruby 以外で書かれた箇所

現在までに実装された RubyOS の機能のうち，次に挙げるものは Ruby でなく C やアセンブラ言語を利用して記述している．

- キーボードからの割り込みを検知する割り込み関数テーブルの設定
- Ruby 処理系が用いる C レベルでの VRAM メモリの操作
- newlib 内に実装されたメモリ管理などを行うシステムコール
- スレッドライブラリに属する関数のダミー
- Ruby 処理系を起動するカーネル部分
- メモリアクセス，I/O ポートアクセス，時間計測を行う拡張ライブラリ

5.4.3 Ruby で記述した部分

Ruby では，デバイスドライバとしてキーボードドライバと，テキスト VRAM のドライバを実装した．それぞれ Keyboard クラス，Vram クラスと定義し，キーボードとテキスト VRAM に対する操作をそれらクラスのメソッドとして作成した．キーボードドライバは I/O ポートアクセス，テキスト VRAM のドライバはメモリアクセスを必要としたため，上記の拡張ライブラリの機能を利用した．それ以外の部分についてはすべて Ruby の言語仕様内で記述した．キーボードドライバは，ポーリング形式にてキーボードからの入力を受け取り，表示可能な文字に変換する．テキスト VRAM のドライバはキーボードドライバが変換した文字を受け取り，画面上に表示する．これらデバイスドライバのクラスを利用して，キーボードからの入力を受け取り画面に表示するプログラムを Ruby で記述した．この Ruby プログラムのうち，クラスを利用している部分を**図 8**，これを RubyOS の実行基盤にて実行した様子を QEMU でのエミュレート画面にて**図 9** に示す．

Vol.2010-ARC-189 No.4
Vol.2010-OS-114 No.4
2010/4/21

```
loop do
  tmp = keyboard.get_keychar #キーボードクラスが受け取ったデータを格納
  if tmp != nil              #何らかの文字を取得した場合
    vram.put_char(tmp)       #その文字を vram クラスがテキスト VRAM に表示
  end
end
```

図 8 作成した Ruby プログラムの一部

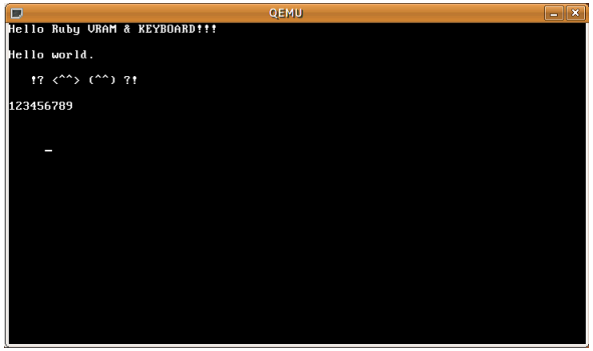

図 9 QEMU でエミュレートした RubyOS の試作プログラム

6. 評価・考察

本節では，本研究で実装された処理系に対して行った評価と考察について示す．

6.1 評　　価

今回製作したベアマシン上で動作する Ruby 処理系の機能を評価するため，いくつかのプログラムを実行させた．また比較対象として，同アーキテクチャ上で動作する次の二つにも同じ動作を行うプログラムを実行させ，その実行時間を計測した．

- Ubuntu9.04 で動作する Ruby 処理系
- またベアマシン上で動作させる C

動作させたプログラムの説明と，動作し測定した結果をそれぞれ**表 5** と**表 6** に示す．

また，Ruby で記述したデバイスドライバを利用して，次の動作に関して測定を行った．その結果を**表 7** に示す．

次節にて，この結果と実装された機能からベアマシン上の Ruby 処理系に関する評価を行う．

表 5 評価した機能

プログラム名称	説明
while	while 構文を用いて 10000 回空ループを回す
fib	10 を引数にしてフィボナッチ数列を計算する
times	times 構文を用いて 10000 回空ループを回す
ary	要素数 10 個の文字型配列に 1 つずつ計 10 個要素を増やす
stack	配列を利用してスタックを実装し，値を 10 個格納した後 5 個取り出す
inb	拡張ライブラリを用いて I/O ポート 0x80 から値を読み込む
outb	拡張ライブラリを用いて I/O ポート 0x80 への書き込みを行う

表 6 Ruby と C での実行時間の比較 [マイクロ秒]

評価した機能	ベアマシン上の Ruby	Ubuntu 上の Ruby	ベアマシン上の C
while	812	876	22.7
fib	33.6	36.4	1.05
times	1600	1632	-
ary	8.11	53.7	6.67
stack	7.99	19.6	0.561
inb	2.21	-	0.153
outb	2.15	-	0.153

表 7 評価した機能

動作	測定結果 [マイクロ秒]
キーボードから 1 文字の入力を受けとる	3.54
テスト VRAM に 10 文字の文字列を書き込む	71.2

6.2 ベアマシン上の Ruby 処理系の機能評価

今回試作的に実装された機能については，ベアマシン上の Ruby 処理系においてキーボードドライバとテキスト VRAM ドライバを実装できた．このとき，必要となったメモリアクセスや I/O ポートアクセスについては，自作した拡張ライブラリにおいて Ruby 処理系に拡張を行い，実行することが可能となった．これにより，ベアマシン上の Ruby に，OS の実装に必要となる実メモリへのアクセスや I/O ポートアクセスの機能が実装されたことが確認できた．さらに，表 6 の結果から，ベアマシン上の Ruby 処理系は Ubuntu9.04 上の Ruby 処理系よりも速度が向上している．特に，配列への要素の追加を行う ary はおよそ

Vol.2010-ARC-189 No.4
Vol.2010-OS-114 No.4
2010/4/21

6.6 倍，スタックの操作を行う stack は 2.5 倍となっている．この理由として，既存の OS 上ではプロセス切り替えや，仮想メモリ空間上におけるメモリ割り当てなどのオーバーヘッドがあったため，ベアマシンでの動作よりも実行時間が大きくなったものと思われる．

ただし，C とベアマシン上の Ruby 処理系にて比較を行うと，Ruby 処理系が C の平均しておよそ 20 倍の処理時間がかかっている．これは，Ruby が C で記述された処理系によって実行されるスクリプト言語となっているからである．しかし，第 3 章でも述べた通り，本研究の目標は，安全性や生産性，拡張性に優れた OS を製作することである．そのため既存の OS との速度の差はあまり有用ではなく，C の処理時間に差があったとしても，動作が問題なく行えれば目的は果たしていると考えられる．さらに，Ruby では C にはないような例外処理といった Ruby プログラムのエラーを捕捉するような機能をもつ．また OS のコード自体がテキストファイルであることから，C のように実行の度にあらかじめコンパイルする必要がなく，またベアマシン上でなくとも既存の OS 上の Ruby 処理系でも実行できる移植性をもつ．そのため，C にくらべて OS コードの修正・拡張もより行いやすいと考えられる．よって今回作成した Ruby 処理系の挙動は本研究の目的に即していると判断した．ただし，今回はキーボードからの入力やテキスト VRAM への出力を行うデバイス操作のみを実現したが，速度性能が問題となる入出力制御について VM を含めた実行系の性能向上を含めて検討したい．

7. おわりに

本章では，本研究で得られた成果と今後の課題について述べる．

7.1 本研究の成果

本論文では，Ruby を用いて記述する OS の構成法について述べた．その実行基盤として，ベアマシン上で既存の Ruby 処理系を実行する方式を提案した．その上で，すでに Ruby の言語仕様や処理系に用意されている機能を利用することで，本研究の目的である RubyOS の開発をより効率よく行えることを示した．

Ruby 処理系には最小単位の処理系である miniruby を利用し，組み込み用標準 C ライブラリをリンクさせることでベアマシン上での実行を可能にした．Ruby の言語仕様に含まれないメモリアクセスと I/O ポートアクセスについては，Ruby の拡張ライブラリとして記述し，それをリンクすることで Ruby プログラムからこれらアクセスが実行できるようにした．これらにより，RubyOS の実行基盤を試作することができた．

その RubyOS の試作として，ポーリング方式でキーボードからの入力を受け取り，それを画面に表示する Ruby のプログラムをベアマシン上の Ruby 処理系にて実装した．

最後に，既存 OS 上の Ruby 処理系やベアマシン上の C との実行時間の比較を行い，今回作成した Ruby 処理系が OS を記述するにあたり十分な機能をもっていることを考察した．

7.2 今後の課題

今後の課題としてはスレッドライブラリの実装が挙げられる．Ruby 処理系がシグナルフラグをチェックするのはスレッドのタイマーチェックなど，スレッドが動いていることが前提であるため，スレッドライブラリを実装しない限り外部割り込みが実装不可能となる．これを解決するため，スレッド API として本研究室で開発された組み込み機器用 OS の開聞を利用する．スレッドライブラリの実装後，外部割り込みの実装や PStore ライブラリの組み込みなど，RubyOS の実装を進める予定である．

参考文献

1) 笹田耕一，松本行弘，前田敦司，並木美太郎：Ruby 用仮想マシン YARV の実装と評価，情報処理学会論文誌 (PRO)， Vol.47, No.SIG 2(PRO28), pp.57–73 (2006).
2) Sansonnet, J.P., Castan, M., Percebois, C. and D.Botella, J.P.: Direct Execution of Lisp on a List_directed Architecture, *Architectural Support for Programming Languages and Operating Systems(Proc. ASPLOS-I)*, ACM, pp.132–139 (1982).
3) まつもとゆきひろ，石塚圭樹：オブジェクト指向スクリプト言語 Ruby，株式会社アスキー (1999).
4) 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).
5) 浅野一成，並木美太郎：PerlOS の試作と評価，情報処理学会「システムソフトウェアとオペレーティング・システム」研究会第 106 回研究報告， Vol.2007-OS-106(12), pp. 87–94 (2007).
6) 浅野一成，並木美太郎：プログラミング言語 Perl によるオペレーティングシステム構成法の研究，1k-7, 情報処理学会第 71 回全国大会 (2009).
7) 萱嶋志門，並木美太郎：組込み用 OS『開聞』の割込み管理機構，情報処理学会研究報告. [システムソフトウェアとオペレーティング・システム]， Vol.2000, No.43, pp. 47–54 (20000525).
8) 井上翔大，大山恵弘：OCaml による OS の実装，情報処理学会「システムソフトウェアとオペレーティング・システム」研究会第 113 回研究報告， Vol.2010-OS-113, No.4 (2010).