RubyによるOSの構築を目指して

吉原陽香[†1] 笹田耕一[†2]
佐藤未来子[†3] 並木美太郎[†4]

本稿では，RubyOS という Ruby にて記述される OS の構成法と，現在までの実装について述べる．RubyOS では資源管理を可能な限り Ruby を用いて記述することを目標としており，Ruby の言語仕様にない，例えば実メモリや I/O ポートへのアクセスなどの処理を行う際には拡張ライブラリを用いて処理系を拡張して利用する．Ruby で OS を記述することにより，例外処理や動的型付け，オブジェクト指向による記述によって開発効率や安全性に優れた OS を実装することを目標としている．また Ruby 処理系には VM が組み込まれており，この VM による実行を行うことでさらに安全性の高い OS を記述できると考えている．この RubyOS の試作として，キーボードとテキスト VRAM のドライバを Ruby で記述し，それらを利用して入力された文字を Ruby スクリプトとして実行するプログラムを 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．

Toward to RubyOS, The OS constructed with Programming Language Ruby

Haruka YOSHIHARA,[†1] Koichi SASADA,[†2] Mikiko SATO[†3] and Mitaro NAMIKI[†4]

This presentation describes a design of the 'RubyOS' operating system with programming language Ruby. The goal of this study is to construct resource management with Ruby, but some functions which can't be executed by this language are implemeted with Ruby extension libraries with C. Some feature of Ruby, such as exception handling, dynamic typing, object-oriented language and RubyVM contribute to safely execution of 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

†1 東京農工大学 大学院 工学府 情報工学専攻
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 工学府
Faculty of Engineering, Tokyo University of Agriculture and Technology.
†4 東京農工大学大学院 工学研究院
Institute of Symbiotic Science and Technology, The Graduate School at Tokyo University of Agriculture and Technology.

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実行時における解釈処理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．

そこで本研究は，スクリプト言語の 1 つである Ruby[4] を用いて OS を構成することを目的とした．

Ruby はまつもとゆきひろらによって開発されたオブジェクト指向のプログラミング言語である．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，変数宣言が必要なく，さらに動的型付けにより型を決定するため，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．さらに，組み込み機器向けの Ruby 処理系の開発も検討されている[3]．

本発表では，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つつ Ruby で記述を行った OS（以下，RubyOS）の設計と，現在までの実装について述べる．また PC/AT 互換機上に Ruby の実行基盤を作成するために試作した，ハードウェア上で直接実行する Ruby 処理系について述べる．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 Ruby 用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 Ruby で OS を記述する妥当性と，実際にベアマシン上で Ruby をどのように動作させるべきかをいくつか案を提示しながら考察する．第 3 章では本研究の関連事項から既存の問題点を検討し，それらをふまえて第 4 章では本研究の目標を述べる．第 5 章では目標となる RubyOS の設計について示す．第 6 章にて現在までに実装できた RubyOS について述べた後，第 7 章ではその実装した部分についての評価，考察を行う．最後に，第 8 章にて現在までの成果と今後の課題を挙げる．

2. Ruby に対する検討

本研究では，Ruby にて資源管理を行うことを目的としている．つまり具体的には，プロセス管理，メモリ管理，ファイルシステム，デバイスドライバや割り込み処理をできる限り Ruby にて実装することが本研究の目的である．そこで本章では，Ruby でこれら資源管理を記述する際の利点や，RubyOS における Ruby スクリプトの実行方式を検討する．

2.1 OS 記述言語としての Ruby の妥当性

本節では Ruby で OS を書くことの利点を示し，その妥当性を検討する．

オブジェクト指向によるプログラミング

第一の利点としては Ruby がオブジェクト指向であることが挙げられる．Ruby ではすべてのデータをオブジェクトとして扱うため，例えばデバイスなどをオブジェクトとして仮想化できれば，資源をすべてオブジェクトとして統一的に扱うことができる．このことは OS の開発効率を向上させると考えられる．

動的型付けによる型決定

Ruby は動的型付けにより型を決定する．これと先のオブジェクト指向により，データの型を意識することなく柔軟に処理を記述することが可能となる．

例外処理の記述が可能

エラーが起きた際にも，Ruby では例外処理の記述が可能であるため，あらかじめ処理を記述しておけばエラーを安全に処理することができる．

直接メモリ・ハードウェアを操作できない Ruby の仕様

Ruby にはポインタ演算などの処理を行う仕様がないため，従来の C などで起こるメモリリークや不正アクセスなどといったエラーが本来は起こることはない．これは一方で，OS を記述する際にもポインタ演算によるメモリへのアクセスや，I/O ポートへのアクセスは Ruby そのままの言語仕様では記述できないということになる．この問題に対しては，後述する拡張ライブラリによって Ruby からそれら機能が扱えるようにして解決を図ればよい．この時，ユーザからはそれら処理を担うクラスは見えないように設計すれば，少なくともユーザプログラムからメモリに不正にアク

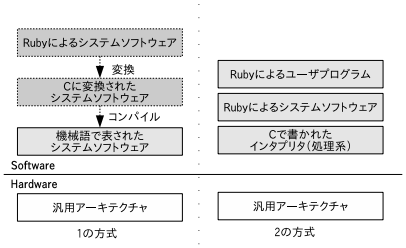

図 1 ベアマシン上での Ruby の実行方式の概略

セスされるといった可能性は限りなく低くなり，安全性に与える影響を小さくできると考えられる．

以上により，Ruby の言語仕様および言語処理系の観点から，Ruby で OS を書くことには妥当性があると考えた．

2.2 ベアマシン上での Ruby の実行

本研究では Ruby スクリプトをベアマシン上にて実行し，資源管理を行うことが目的となる．しかし，Ruby は従来の汎用 OS 上で動くように設計されており，したがって通常はベアマシン上では Ruby は実行できない．ただ，前節で述べたとおり，Ruby で OS を記述することには十分な妥当性があると考えている．よって，Ruby で資源管理を行うには，まずベアマシン上で Ruby を実行できるようにしなければならない．これを解決する方法として，大まかに分けて次の二つが考えられる．

(1) Ruby スクリプトを C に変換し，コンパイルする
(2) ベアマシン上に処理系を移植し，その上で Ruby スクリプトを実行

この二つの方式を図にすると次の**図 1** になる．

次節からそれぞれの方式について考えられる利点や欠点を述べる．

2.2.1 C に変換しコンパイルする方式

Ruby で書いた OS コードを C に変換し，それをコンパイルして実行する方式をとる場合，利点として Ruby をベアマシン上で直接処理系に実行させるよりも実行速度が出せるという点がある．Ruby はインタプリタ言語であるため，事前にコンパイルされバイナリ化されたプログラムよりも速度が出ないという欠点がある．しかし C のコードに変換してコンパイルする場合，たとえ元々のソースコードが Ruby で書かれたものであっても，事前にコンパイルされるため速度に関しては問題はなくなる．

一方，欠点としては，Ruby スクリプトを C に変換する何らかのソフトウェアが必要となる点である．実行前に Ruby スクリプトを C に変換し，コンパイルする AOT コンパイラは現在研究されている[9] ものの，このコンパイラによって生成されたバイナリは Ruby 処理系のランタイムライブラリである libruby に依存したものとなるため，まずこの libruby を汎用 OS に依存せずベアマシン上で利用できるように作成しなおす必要がある．またこの問題を解決したとしても，Ruby で記述したソースコードを修正したとき，実行するためにはその都度そのコード全体を C に変換してからコンパイルするというように開発上での手間が生じる．

2.2.2 ベアマシン上へ処理系を移植する方式

ベアマシン上にて Ruby 処理系を実行し，その上で Ruby で記述したスクリプトを実行する方式の利点は，C で記述された既存の Ruby 処理系を用いることができる点である．これをコンパイルすることでベアマシン上でも Ruby を動作させることができる．ただし，通常 Ruby 処理系のコンパイルに使う C のライブラリは OS に依存するため，今回の目的に即するには組み込み用などのライブラリを別途用意してリンクする必要がある．

またこの方式の場合，Ruby 処理系に組み込まれた VM にて Ruby スクリプトを実行するため，仮に何らかの重大なエラーが発生したとしてもその VM の停止だけで済み，全体に影響を与えないという利点がある．さらに前述の通り例外処理の機能にてエラーを事前に補足する処理を記述できるため，さらに安全性を高めることができる．

また，Ruby のソースコードをテキスト形式のまま実行できるため，前者の方式のように修正の度にコンパイルする手間が省け，開発効率にも貢献するものと思われる．

なお，前節でも述べたように Ruby スクリプトを直接実行する場合，コンパイルされたバイナリよりも実行速度は基本的に落ちる．

2.2.3 実行方式の比較

上記二つの方式を比較・検討し，本研究で用いる方法を決定する．

まず前者の Ruby を C に変換しコンパイルする方式では，Ruby を直接実行するよりも速度は向上できるものの，まず前段階として Ruby を C に変換するソフトウェアを作成する必要がある．一方，ベアマシン上に処理系を移植する方法では，前者よりも実行速度は落ちるが，VM による実行によってより安全性の高い OS が実装できると考えている．本研究では，OS による資源管理をできる限り Ruby で書くことが目的

であり，速度の差は検討材料としてあまり有用ではないと考えている．また既存の Ruby 処理系には VM が組み込まれたことから速度の面でも改善され多くの人に利用されており，速度に関しては過剰に意識する必要はないと考える．

以上の点から，今回は Ruby 処理系をベアマシン上に移植し，その VM にて Ruby で記述した OS コードを実行する方式を採用した．

3. 関連研究

ここでは，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 の処理系としてすでに 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[5] 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．しかしこの場合，開発マシンにて Java ソースコードをコンパイルする必要がある．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[10] 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．本研究の場合，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[6),7] は，インタプリタ言語の Perl を用いて浅野が開発した 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，使用メモリ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．

4. 目　　標

本節では，第 2 章での考察を元に，本研究の目標，RubyOS の構成について述べる．

4.1 本研究の目標

RubyOS は，従来 C にて実装されていた OS が行う計算機資源管理をできる限り 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，例えば計算機資源を仮想化して Ruby のオブジェクトに対応させ操作するといった，統一的で見通しのよい構成をもつ OS を目標とする．

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

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か，後述する拡張ライブラリにて Ruby 処理系を拡張し，Ruby からそのライブラリに実装された機能を利用することで操作できるようにする．他の資源管理は可能な限り Ruby で記述し，その実行は Ruby 処理系の VM で行う．

つまり，次の要素にて RubyOS を構成する．

- Ruby で記述された部分（ハードウェア独立部）
- Ruby 処理系や標準 C ライブラリなどのハードウェアに依存する部分

ここで，RubyOS を搭載した x86 の PC/AT 互換機を RubyMachine と呼ぶ．RubyOS で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．また，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．

4.2 システムの全体構成

本研究における RubyMachine の全体構成を次の**図 2** に示す．

次に各構成要素について概略を述べる．

4.2.1 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 が扱うデータを保存するために利用する．

4.2.2 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 C 言語で記述された既存の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．この既存の Ruby 処理系を利用することにより，処理系内に組み込まれた VM を利用してより安全に Ruby スクリプトを実行することができると考えている．また処理系が用いる標準 C ライブラリは OS 依存のものではなく，組み込み用のライブラリなどを用いる．

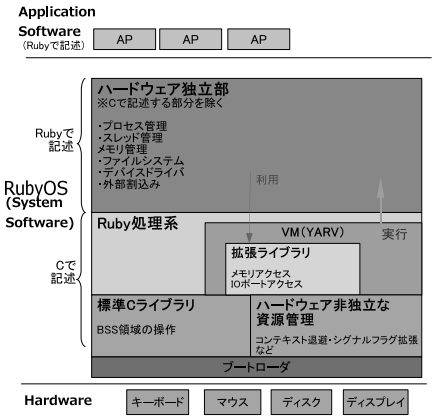

図 2 RubyMachine の全体構成

4.2.3 拡張ライブラリ

Ruby の言語仕様にない機能を用いる場合，拡張ライブラリを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，もとの Ruby 処理系そのものの内部の改変を行うことなく，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る．また現在，処理系には組み込まれていないものの，すでに一般に使われている拡張ライブラリが多数存在する．これらに関しても，既存 OS に依存する処理さえ含まれていなければ，今回実装するベアマシン上の Ruby 処理系でも利用することができると考えられる．

4.2.4 RubyOS で行う資源管理

RubyOS がもつ機能について概要を下に示す．

- プロセス・スレッド管理 : Ruby 処理系の VM を複数起動して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 5 章にて示す．なるべく多くの資源管理処理を Ruby にて実装することを目的とするが，前に述べたとおりハードウェアに依存する部分に関しては C やアセンブラを利用し，Ruby 処理系の拡張ライブラリを作成するなどして作成する．それ以外の Ruby で書かれた部分については，前節の Ruby 処理系にて実行させる．

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．

5. 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ようになっている．次節以降にてそれぞれについて詳細を述べる．

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 の軽量スレッドの Fiber 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

5.1 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，一つのメモリ空間を複数のプロセスで共有する場合と異なり，プロセス数が増えるほどそれぞれに固有にメモリを割り当てるため，全体のメモリ使用量が増大する．しかし，アドレス空間がプロセスごとに独立するため，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．

図 3 メモリ割り当ての概要

5.2 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 Fiber[8] クラスを利用する．Fiber とは Ruby1.9 から導入された軽量スレッドで，自動で処理が切り替わらないノンプリエンプティブなユーザレベルのスレッドである．RubyOS ではこの Fiber を利用し，プリエンプティブなスレッドを実装する．具体的にはスレッド一つに Fiber 一つを割り当て，それらスレッドのコンテキストを切り替えるスケジューラを Ruby にて記述する．

5.3 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，ヒープ領域が増大した場合を考え，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 3** に示す．

オブジェクトはページングによる仮想メモリ上に読み込まれ，VM（プロセス）ごとに一つの仮想アドレス空間しか操作できないようにする． 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．

5.4 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．ユーザは Store クラスというクラスを用いてファイルシステムを利用する．ファイルの指定には Unix ライクなパスを用いる．この Store クラスを使用したコードの例を**図 4** に示す．

```
#ファイル bar に対応付けた Store オブジェクトを作成
file = Store.new("foo/bar", 1)
# ディスクからメモリ上のオブジェクト temp にデータの内容を読み込む
temp = file.data
# データの中身を変更（この時点でディスクに書き込まれる）
file.data = "TESTTEST"
# ファイルのクローズ（オブジェクト file からファイルへのアクセスを終了）
file.close
```

図 4 Store の使用例

この Store クラスは RubyOS のファイルシステムのために構成されたものであり，PStore という Ruby ライブラリを中心に構成する．PStore については次節にて説明する．この PStore を用いてファイルやディレクトリを表現する．また PStore からディスクにアクセスするには Block クラスというクラスをデバイスドライバとして使用する．上記の関係を図にすると次の**図 5** のようになる．

5.4.1 ディスクレイアウトと Block クラスについて

ディスクは一定の大きさのセクタに分けられ，その空き管理はビットマップを使って管理する．ディスクの最初にはそのビットマップのサイズが格納される．またルートディレクトリのデータは表の直後に配置する．Block クラスは，セクタ番号からアドレスを割り出し，対応するディスクの中身の読み込みや書き込み，またセクタの空き領域の探索を行う．

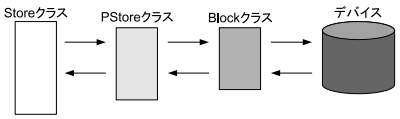

図 5 ファイルシステムの構成

5.4.2 PStore

PStore とは Ruby 処理系に付属する Ruby のライブラリであり，オブジェクトを外部記憶（ファイル）へと保存することができる．PStore はハッシュと似た機構を持っており，PStore にて保存したオブジェクトにはハッシュと同様にキーを割り当ててアクセスすることができる．なお，オブジェクトはバイナリダンプされた状態で保存され，このダンプには Marshal というクラスの dump メソッドが用いられている．この PStore を利用したコードの例を**図 6** に示す．

```
require 'PStore'

test = PStore.new("dumpfile")

test.transaction do
  test["foo"] = "text"
  test["bar"] = [12,34]
end

test.transaction do
  p test["foo"] #=>"text"
  p test["bar"] #=>[12, 34]
end
```

図 6 PStore の利用例

5.5 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．

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

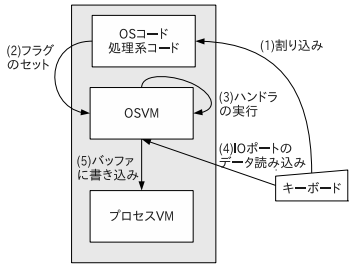

図 7 キーボードによる割込み発生時の流れの例

る．このフラグは Ruby 処理系内でシグナルのハンドラを実行しても安全な箇所にてチェックされ，フラグが立っていた場合にはそのシグナルに対応するハンドラが実行される．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 7** 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．

5.6 記述言語の検討

第 4 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．以上のことに関して具体例も含めてまとめると次のようになる．

- Ruby 処理系の動作を妨げる処理
  - プロセスコンテキスト退避
  - 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 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）

これ以外の部分についてはすべて Ruby にて記述する．

5.7 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 1** にまとめる．

また，それぞれの使用例を次の**表 8** と**表 9** に示す．

表 1 クラス Memio がもつクラス

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

```
# 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 8 メソッド readb,writew の使用例

```
# 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 9 メソッド inb,outb の使用例

6. 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．

6.1 現時点での RubyOS の構成

現時点で，第 4 章で示した図 2 のうち，次の部分を実装した．

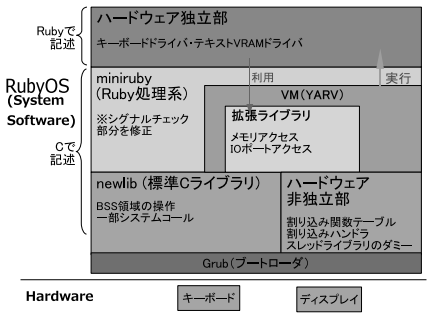

図 10 試作した RubyMachine の全体構成

- ハードウェア独立部
  - Ruby のシグナルトラップ方式によるキーボードドライバの実装
  - テキスト VRAM ドライバの実装
- Ruby 処理系
  - Ruby 処理系である 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
  - キーボードドライバ用シグナル SIGKBD の拡張
- ハードウェア非独立部（C ライブラリ含）
  - 標準 C ライブラリによる BSS 領域の操作
  - キーボードとタイマの割り込み関数テーブルの設定
  - タイマ割り込みによる clock_gettime() の実装
  - ネイティブスレッドライブラリのダミー
  - キーボードドライバ用シグナル SIGKBD の拡張
- ブートローダ
  - grub によるブートと Ruby スクリプトのロードの実装

これらを図にまとめたものを次の**図 10** に示す．

これらの詳細については次節以降で述べる．なお，現時点では RubyOS として一つのプロセスだけを実メモリ上で実行しており，ファイルシステムに当たるものは未実装である．

また，RubyOS を搭載する計算機の仕様を次の**表 2** に示す．

6.2 Ruby 処理系

本節では，Ruby 処理系のベアマシン上での実装について述べる．

表 2 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
キーボード	PS/2 接続

6.2.1 ベアマシン上への移植

今回，Ruby 処理系として，最小単位の Ruby 処理系である miniruby を利用する．この miniruby は完全な Ruby 処理系のビルド途中で構築され，その完全な Ruby 処理系をコンパイルする際に用いられる．この miniruby には一部のエンコーディング以外と付属の拡張ライブラリは処理系に組み込まれないが，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．今回 RubyOS では，組み込み用標準ライブラリの newlib を用い，miniruby とのリンクを行った．この標準 C ライブラリのうち，例えば BSS 領域の拡大・縮小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 miniruby でも実行できる．

ほか，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．

6.2.2 拡張ライブラリ

第 5.7 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．これにより，Ruby から実メモリや I/O ポートに直接アクセスができるようになった．

6.3 ブート手順

移植した 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する．その後，ブート用のデバイス上に置かれた Ruby スクリプト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，Ruby スクリプトを Ruby 処理系にて実行する．

6.4 ハードウェア非独立部

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．

- キーボードやタイマ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
- タイマ割り込みによる clock_gettime() の実装
- Ruby 処理系の内部で用いる C レベルでの VRAM メモリの操作
- newlib 内に実装された BSS 領域の管理などを行うシステムコール
- キーボード用シグナル SIGKBD の追加
- 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

Ruby 処理系はネイティブスレッドを内部で利用しているため，そのネイティブスレッドのライブラリに属する関数をダミー関数，つまり呼び出しても何も処理を行わず，成功した場合の値を返すような関数として実装した．ほか，後述するキーボードドライバに利用するため，シグナル SIGKBD を処理系や newlib に追加した．

6.5 ハードウェア独立部

上記で移植した Ruby 処理系で動作させる Ruby スクリプトとして，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，先ほど述べた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，まずポーリング形式にて I/O ポートからキーコードを取得し，それを文字に変換するような処理を実装した．そのドライバを実行し，Ruby にてキーボードからの入力が取得できていることを確認した．その後，Ruby によるシグナルハンドラの機能を用いて，キーが押された際に処理系内のフラグが立ち，それをチェックした処理系が対応するハンドラを実行してキーコードを取得するように修正した．

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

図 11 作成した Ruby プログラムの一部

具体的には，C レベルにてキーボードの割り込み関数テーブルを設定し，C ライブラリと Ruby 処理系に SIGKBD というシグナルを追加した．そして Ruby にてそのシグナル SIGKBD に対応するシグナルハンドラを記述した．そのシグナルハンドラ内では，入力されたキーのキーコードをキューに追加する処理を行うようにした．この記述により，キーが押されると割り込み関数テーブルに設定された関数が実行され，その関数が Ruby 処理系に SIGKBD に対応したフラグを立てる．そのフラグを処理系がチェックし，Ruby のシグナルハンドラを実行することで，キューにキーコードが追加される．このキューから一つキーコードを取り出して文字に変換するメソッド Keyboard#get_char も実装した．この Keyboard#get_char メソッドを利用してテキスト VRAM のドライバは文字を受け取り，画面上に表示する．これらデバイスドライバを用いたコードの例を**図 11** に示す．

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決められた文字を入力するとそれまでの文字列を Ruby スクリプトとして実行するプログラムを Ruby で記述した．このプログラム内では上記デバイスドライバのインスタンスを作成してメソッドを利用することで，処理の流れを簡潔に表現することができた．この Ruby プログラムを 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 12** に示す．

7. 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．

図 12 QEMU でエミュレートした RubyOS の試作プログラム

表 3 評価した機能

プログラム名称	説明
times	times 構文にて 10000 回空ループを回す
ary	2 つの配列からハッシュを作成する
matrix	100 × 100 の行列同士の乗算を行う

表 4 Ruby と C での実行時間の比較 [秒]

評価した機能	ベアマシン	Ubuntu
times	0.01	0.002
ary	0.01	0.004
matrix	0.76	0.74

7.1 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 Ubuntu9.04 上の Ruby 処理系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．この時間の計測には，内部で標準 C ライブラリの clock_gettime() を利用する Ruby の Time クラスを使用した．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 3** と**表 4** に示す． なお matrix の計算には Ruby で書かれたライブラリ matrix を用いた．

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．

7.2 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 4 の結果から，ベアマシン上の Ruby 処理系は Ubuntu9.04 上の Ruby 処理系と比べても数倍の処理速度の差しかないことがわかる．なお実行速度に差が出た原因として，Ubuntu 上の Ruby 処理系では実行の際に OS による何らかの最適化が行われたためと考えている．また matrix に関してだが，ファイルシステムのないベアマシン上では，パスを検索してファイルをロードする Ruby の require 文は利用できないため，ライブラリの中身を直接同じファイル内に記述し，grub にロードさせている．一方の Ubuntu 上では require 文にてライブラリのロードを行っているため，この読み込みにかかるオーバーヘッドにより速度差が縮まり，約 1.02 倍ほどの速度差になっているものと思われる．

ただし，既存の OS の実装に使用されることが多い C などと比べると，Ruby は 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であるため，Ruby の処理速度は基本的に遅くなる．しかし，第 4 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，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．つまり，Ruby 処理系が利用する C ライブラリなどのハードウェアに依存する部分さえ実装が行われていれば，あるアーキテクチャにて動作する Ruby スクリプトはそのまま別アーキテクチャでも利用できる移植性があるということである．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．

8. 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．

8.1 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 の実行基盤を試作した．

その RubyOS の試作として，キーボードからの入力を受け取り，それを Ruby スクリプトとして実行する 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．これにより，RubyOS の設計部分のうち，キーボードドライバと VRAM ドライバに関して実装が確認できた．キーボードドライバに関しては Ruby のシグナルトラップの機能を利用した．またドライバをクラスとして定義することで，実際にそれらドライバを利用して外部機器を操作する際にはメソッドを呼び出せばよいため，メイン部分の表現が見通しの良いものになった．

最後に，既存 OS 上の Ruby 処理系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．

8.2 今後の課題

今後の課題としてはファイルシステムの実装が挙げられる．まず Ruby レベルでのハードディスクドライバとなる Block クラスの実装を行い，並行して PStore ライブラリのベアマシン上 Ruby 処理系への移植，メモリのページング機構の実装などを進めていく予定である．

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：RubyConf 2010, keynote, http://www.rubyconf.org/ (2010).
4) 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).
5) 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).
6) 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 (2007).
7) 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).
8) 芝　哲史，笹田耕一：Ruby1.9 での高速な Fiber の実装，第 51 回プログラミングシンポジウム予稿集，pp.21–28 (2010).
9) 芝　哲史，笹田耕一，卜部昌平，松本行弘，稲葉真理，平木　敬：実用的な Ruby 用 AOT コンパイラ，情報処理学会プログラミング研究会 SWoPP, Vol.2010-2-(7) (2010).
10) 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).