修士論文

Master's Thesis

CLI で実装する次世代オペレーティングシステム

A NEW GENERATION OPERATING SYSTEM IMPLEMENTED WITH THE CLI

— 公開版 –

高橋 明生

(957 ◆ KhjEefLZMk)

Akio TAKAHASHI

2006 年 2 月 9 日

はじめに

もっとも長く触れているソフトウェアはオペレーティングシステムである。それにもかかわらず、オペレーティングシステムの動作をユーザが意識することはない。オペレーティングシステムはプログラムと対話する存在であり、基本的にユーザとは関わろうとしないからである。このせいでユーザは自らが最もふれているソフトウェアのことを意識しないようになってしまい、オペレーティングシステムの評価がユーザの視点からなされることが多くなってしまった。これによって、“機能” や “見た目” など本来はオペレーティングシステムが負うべきものではないものが重要視されてしまっている。

この結果、ユーザの要望に応えるかたちでオペレーティングシステムは肥大化し、理解しにくいプログラムの代表格になってしまった。そして、もともとは単なるルーチンの寄せ集めにすぎなかったものが、もっとも開発しにくいソフトウェアになってしまった。いまではオペレーティングシステムを開発することはそれ自体が何か神聖なものであると考えられているようにさえみえる。

筆者はオペレーティングシステムの開発が敬遠されていることは非常に好ましくないと考えている。たとえば、なぜオペレーティングシステムの部品の研究は多々あるのに、それを組み合わせる研究は少ないのか。部品があるだけでソフトウェアが動作するわけではない。たとえば、なぜ組み込み向けの研究はパーソナルコンピュータ向けの研究より少ないのか。今後、組み込み機器に情報処理の負荷が分散することはありそうだが、私たちの目の前からコンピュータが消えるときが来るのだろうか。

本研究では、近代的オペレーティングシステムが備えるべき機能がいくつか欠けているものの Microsoft 社の.NET 技術を取り入れるなど最新技術に対応するという側面も持ち合わせたパーソナルコンピュータ向けオペレーティングシステムを開発するとともに、カーネルの実行に関して新しい手法を提案している。

内容には、ほかの研究者が結果を再現できるように既存技術について説明を追加した。本書だけで PC/AT 互換機用オペレーティングシステムを開発することは難しいと思うが、オペレーティングシステム開発の参考書となりうる構成にした。

本書の構成

本研究ではカーネルを実行する新しい手法を提案しているが、その手法だけでオペレーティングシステムが構築できるわけではなく、オペレーティングシステムの常として多くの既存技術を利用している。また、提案手法はカーネルの実行に関わるものであるが、提案手法を生かせる状態までコンピュータをブートするためにはいくつもの手順を経なければならない。その過程が明らかでないと提案手法を完全に理解できないと思われるため、次のような構成を取ることにした。

第 1 章　背景

本章では本研究の背景を示し、研究動機を述べることで本研究の主眼がどこに置かれているのかを明らかにする。

第 2 章　既存技術

本章では本研究で用いた既存技術について説明する。オペレーティングシステムの開発には文献だけではなく純粋な技術資料も必要であるため、特に重要と思われるものについて概要を示した。

第 3 章　アーキテクチャ

本章では開発したオペレーティングシステムの全体像を示すとともにアーキテクチャについて説明し、システムの構成について明らかにする。

第 4 章　コンポーネント

本章では開発したオペレーティングシステムをどのようなコンポーネントに分けて構築するかを説明し、プログラムとしてどのような構造になっているか明らかにする。

第 5 章　実装

本章では前章で説明したコンポーネントに実装される個々の機能について説明し、実際の動作について明らかにする。

第 6 章　提案手法によるカーネルの実行

本章では本研究で提案するカーネルを動作させるための新しい手法について説明する。

第 7 章　検証と評価

本章では第 5 章の提案手法を中心に、オペレーティングシステムとしての検証と評価を述べている。

第 8 章　結論

本章ではまとめと結論を簡潔に述べている。

本書の表記

数値

数値の本文中での表記は**表 1** に示すようにする。（プログラムコード中の表記とそれを本文へ明示的に引用した部分では、プログラミング言語に従った表記を行う。）10 進数には特別な修飾を行わなず、16 進数に

表 1　数値の表記

基数	表記
10	57
16	0x39 または 39h

は先頭に `0x` を付加するか、慣用的な表現のために末尾に `h` を付けることにする。末尾に `h` を付ける表現は、特にメモリアドレスや I/O ポート番号、CPU 割り込み番号などを示すときに一般的に用いられている。

なお、一部のプログラミング言語では先頭の 0 によって 8 進数を意味することがあるが、本書では 8 進数表記は用いないため先頭の 0 は特別な意味を持たない。

識別子

CIL においては、型の完全名を

`<namespace>.<typename>`

のように表す。`<typename>`は型の名前である。`<namespace>`は名前空間であり、英数字だけではなくドット（`.`）も含むことができる。（たとえば `System.Reflection` などの例がある。）また、ある型のメンバは

`<namespace>.<typename>:<member-name>`

のように表す。`<member-name>`は型が含むメンバ（フィールドやメソッドなど）の名前である。`<namespace>.` の部分は曖昧さがないとき省略してよい。

本書では冗長になる場合は文脈によって名前空間を省略している。また、紙面の都合で完全名を表記できないときは、FieldInfo[*1]のように脚注に示している。

コード

プログラムコードやコンソール出力などテキストを掲示するときは**テキスト 1** のように背景が灰色のボックスの中に行番号を付与して示すことにする。また、コードの長さが不十分などの理由でプログラミング言語が分かりにくいときは、**テキスト 2** のように先頭にコメントとしてプログラミング言語を明示することにする。

テキスト 1　コード例

```
1  using System;
2
3  namespace MyNamespace {
4      public class MyClass {
5          public static void SayHello() {
6              Console.WriteLine("Hello!");
7          }
8      }
9  }
```

テキスト 2　言語を明示したコード例

```
1  // IA-32 Assembler
2  freeze:
3      cli
4  sleep:
5      hlt
6      jmp sleep
```

[*1] System.Reflection.FieldInfo

手続き

コードにするには煩雑な処理は PAD 図による手続きとして表現することにする。このとき、一般的な PAD 表記と異なり、図が表す手続きの名前は上部に分離して示すことにした。手続きには関数とサブルーチンの二種類がある。関数の例を**図 1** に示す。

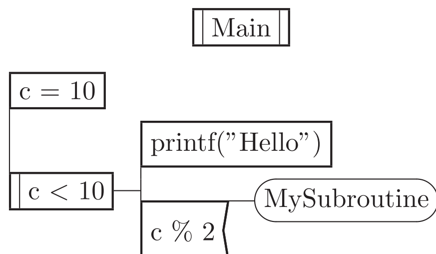


図 1　PAD 関数

サブルーチンは紙面の都合で関数を分割する必要があるときに用いている。サブルーチンの例を**図 2** に示す。関数と異なり、サブルーチンは参照元とスコープを共有するものとした。具体的には、たとえば変数が

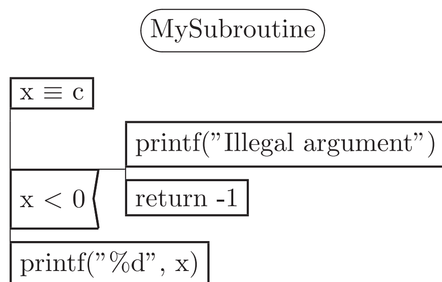


図 2　PAD サブルーチン

共有されることになり、図では変 `c` を共有している。また、`return` 命令は `MySubroutine` ではなく `Main` 関数から制御を戻すことを意図している。

見やすさのために、識別子の別名定義を行う場合がある。図 2 では `x` $\equiv$ `c` という箇所が該当し、これは以降 `c` の別名として `x` という名前も使うことを意味する。代入と異なり両者は同一のインスタンスを参照するため、一方への操作はもう一方へも影響する。

謝辞

大学に提出した論文には書けませんでしたが、私が開発をはじめてから今日まで（主にネットワーク上で）貴重な活動を場をいただき、またそれを支えてくださる方々に、改めて感謝いたします。

目次

はじめに i
本論文の構成 i
表記 ii
謝辞 iv

第 1 章 背景 1
1.1 新しい実行環境 2
1.2 オペレーティングシステムへの適用 2

第 2 章 既存技術 3
2.1 PC/AT 4
2.2 Intel Architecture 32 (IA-32) 10
2.3 Microsoft Windows 13
2.4 Common Language Infrastructure (CLI) 14

第 3 章 アーキテクチャ 19
3.1 コンセプト 20
3.2 ハードウェア要求 21
3.3 ソフトウェア構成 22
3.4 メモリモデル 24
3.5 カーネルブリッジ 25
3.6 パッチ 26

第 4 章 コンポーネント 29
4.1 ブートストラップ・ローダ 30
4.2 レガシーカーネル 31
4.3 マネージカーネル 32
4.4 FreeType 34
4.5 ビルド 38

第 5 章 実装 43
5.1 CIL インタープリタ 44
5.2 メモリ管理機構 47
5.3 基本コレクション 48
5.4 リフレクション 49
5.5 コードベリファイア 52
5.6 CIL コンパイラ 54

第 6 章　提案手法によるカーネルの実行 59
- 6.1 カーネルのブート 60
- 6.2 メタデータの構築 61
- 6.3 コンパイル 65
- 6.4 JIT コンパイルの仕組み 66
- 6.5 CIL 命令の翻訳 67

第 7 章　検証と評価 71
- 7.1 実験環境 72
- 7.2 インタープリタの数値演算 72
- 7.3 インタープリタの動作 73
- 7.4 メタデータの構築 73
- 7.5 コンパイラ 74
- 7.6 JIT コンパイル 75
- 7.7 オペレーティングシステム動作試験 76

第 8 章　結論 79

付録 A　ソースファイル 81
- A.1 bootloader 81
- A.2 kernel 85
- A.3 csbridge 85
- A.4 fdimage 86
- A.5 cdimage 86

参考文献 89

図目次

1 PAD 関数 iv
2 PAD サブルーチン iv

2.1 DMAC の接続 6
2.2 PIC デバイスの接続 7
2.3 PIT の接続 7
2.4 ATAPI コントローラの接続 8
2.5 FDC と FDD の接続 8
2.6 ATAPI 光学ドライブの接続 9
2.7 IA-32 の特権レベル 12

3.1 インタープリタとコンパイラの関係 23
3.2 カーネルと内部コンポーネントの関係 24
3.3 メモリ構造概要 24

4.1 FD 媒体のデータレイアウト 30
4.2 ブートストラップ・ローダの実行シーケンス 31

5.1 インタープリタの主処理 45
5.2 CIL 実行ルーチン 46
5.3 特殊実行ルーチン 46
5.4 間接呼び出し実行ルーチン 46
5.5 基本的なリフレクションクラス 49
5.6 コンパイラが生成する機械語のスタックフレーム 55

6.1 メモリの物理的なレイアウト 60
6.2 初期状態 60
6.3 メタデータ構築手順 61
6.4 コンパイル手順 61
6.5 単純な実装でのメモリ中のデータとコード 61
6.6 提案手法のメモリ中のデータとコード 62
6.7 メモリ中のデータとコード 63
6.8 切り替え前のメタデータ構造 64
6.9 切り替え後のメタデータ構造 65
6.10 JIT コンパイル時のシーケンス 67

7.1 単純ループの実行時間 74
7.2 再帰による総和計算の実行時間 75

7.3 起動画面（テキストモード） 77
7.4 起動画面（グラフィックモード） 77
7.5 フォントレンダラで日本語を表示した画面（普通のフォントサイズ） 78
7.6 フォントレンダラで日本語を表示した画面（大きいフォントサイズ） 78

表目次

1 数値の表記 ii

2.1 PC/AT の構成 4
2.2 PC/AT のメモリマップ 5
2.3 テキストプレーンの文字データ構造 5
2.4 PC/AT の内部デバイス 6
2.5 IA-32 の汎用レジスタ 10
2.6 IA-32 汎用レジスタの重ね合わせ 11
2.7 IA-32 のセグメントレジスタ 11
2.8 IA-32 の制御レジスタ 11
2.9 スロットの要素 17
2.10 MyClass のインターフェイスマップ 17

3.1 アーキテクチャの対応 21
3.2 CooS のハードウェア構成 21
3.3 ブロックの構造 25

4.1 マネージカーネルを構成するアセンブリ 32
4.2 プロジェクトに追加するファイル 34
4.3 ビルド環境 38
4.4 ブートストラップ・ローダの構成 38
4.5 C 言語ソースコードのリンクパラメタ 39
4.6 マネージカーネルを構成するアセンブリのビルド 40

5.1 Stack 44
5.2 Method 44
5.3 Frame 45
5.4 EvalStack 45
5.5 Machine 45
5.6 コードフロー例のスタック状態 53
5.7 コードフローの例 53
5.8 呼び出し規約 54

6.1 補助ルーチン一覧 68

7.1 Double.MaxValue の値 72

テキスト目次

1 コード例 . . . iii
2 言語を明示したコード例 . . . iii
2.1 スロットのサンプルコード 1 . . . 16
2.2 スロットのサンプルコード 2 . . . 16
2.3 スロットのサンプルコード 3 . . . 16
3.1 型オブジェクトの取得 . . . 25
3.2 IKernel の宣言 . . . 25
3.3 IKernel の定義 . . . 26
3.4 IKernel の受け渡し . . . 26
3.5 CooS.Bridge の実装 . . . 26
3.6 画面に文字を出力するメソッドのシグニチャ . . . 27
3.7 パッチされるコード例 . . . 27
3.8 パッチするコード例 . . . 27
4.1 malloc . . . 35
4.2 free . . . 35
4.3 realloc . . . 35
4.4 FreeType を外部に公開するためのクラス . . . 36
5.1 add 命令の実行ルーチン . . . 47
5.2 メモリブロックのヘッダ . . . 47
5.3 メモリブロック確保ルーチン . . . 48
5.4 RuntimeTypeHandle に関するメンバ . . . 50
5.5 RuntimeTypeHandle アクセッサ . . . 50
5.6 スロットに関するメンバ . . . 51
5.7 インターフェイスマップに関するメンバ . . . 51
5.8 コードフローの C#サンプル . . . 52
5.9 コードフローの CIL サンプル . . . 52
5.10 CIL の ld.i4 命令例 . . . 56
5.11 ld.i4 と等価な IA-32 命令 . . . 56
5.12 CIL の add 命令例 . . . 56
5.13 add と等価な IA-32 命令 . . . 56
5.14 isinst . . . 57
5.15 isinst の実装 . . . 57
6.1 メソッド f . . . 67
6.2 f の呼び出し文 . . . 67
6.3 f の変更された呼び出し文 . . . 67

7.1 Double 型検証コード 72
7.2 メタデータ検証コード 73
7.3 メタデータ検証結果 73
7.4 単純ループ 74
7.5 総和計算 74
7.6 JIT コンパイル検証コード 76
7.7 JIT コンパイルの応答 76
A.1 bootldr.h 81
A.2 bpb.asm 81
A.3 1stboot.c 82
A.4 2ndboot.c 82
A.5 3rdboot.asm 84
A.6 makefile 85
A.7 _entrypoint.cpp 85
A.8 コンパイラオプション 85
A.9 リンカオプション 85
A.10 interop.h 85
A.11 rootdirectory.asm 86
A.12 fat.asm 86
A.13 makefile 86
A.14 makefile 86

第 1 章

背景

従来、PC 上でのアプリケーションの開発には C 言語などが使われてきたが、近年急速に C#や Java などの新しい言語を用いるようになっている。これによって、たとえば C 言語の場合に多発していたメモリ関係のバグが減少したり（あるいは無くなったり）、オブジェクト指向による開発手法を取り入れることが可能になるなど、いくつもの有用な利点がある。

1.1 新しい実行環境

C#や Java の重要な利点の多くは仮想マシンを想定した実行環境によって実現されている。C#の場合には中間言語として CIL (Common Intermediate Language) を、仮想マシンとして†CLI (Common Language Infrastructure) を定義している[20]。

† CLI の場合は仮想マシンというよりは仮想環境と表現したほうが的確であるが、一般的な呼称に従い仮想マシンと呼ぶことにする。

しかしこれまで CLI はアプリケーションにしか用いられておらず、システムソフトウェアにはほとんど活用されていない。CLI にはセキュリティなど基盤ソフトウェアに求められている性質がいくつもあるが、特にカーネルに活用するには CLI が中間言語と仮想マシンを想定していることに由来する技術的な課題がある。

中間言語は、逐次解釈や JIT コンパイル (Just-in-time compile)、AOT コンパイル (Ahead-of-time compile) などによってコンピュータ上で実行される。このとき、仮想マシンの一部の命令は単純に処理できないので、そのような処理は実マシンに置かれたランタイムプログラム†によって処理されることが多い。具体的には、型システムに関する機能やコンパイルに関する機能が特にランタイムプログラムへ依存してしまう。

† 実行時に必要な仮想マシンの実行を補助するためのプログラム。たとえば、インタープリタや JIT コンパイラなどがランタイムの中に含まれることになる。

しかし、カーネルに CLI や JVM に由来する技術を適用する場合、通常は実マシンに置かれるランタイムプログラムの存在は期待できないため、ランタイムへと移譲していた処理の実装が困難になる。

1.2 オペレーティングシステムへの適用

JX[3]や JNode[9]、Singularity[4]では、配布イメージ†をビルドする際に AOT コンパイルし、カーネルの機械語と初期データをセットで生成してしまうことで、中間言語から得られたプログラムがランタイムを必要とする実行ステップを事前に終えてしまう。しかし、この手法ではビルド時に生成する実行イメージとカーネルの実行時に扱うメモリイメージの間に直接の相関性がなく、カーネルに関するデータだけが特別扱いされてしまう。このような規格外のデータは一般にポインタを介してアクセスするなど、CLI 非準拠の部分では CLI の利点を生かすことができなくなってしまう。

† ここではユーザが実行可能なバイナリイメージを指している。オペレーティングシステムの場合は、起動可能媒体の生イメージであることが多い。

また、Java（および JVM）をカーネルに適用する研究は行われているのに対し、CLI を適用するものはほとんど行われていない†が、ポインタや機械語の扱い方などにおいて JVM よりも CLI の方がカーネルには適しており、CLI でのカーネル実装に意義があると考えられる。

† 2006 年 1 月時点で本研究の他には Singularity[4] しか見つけることができなかった。

そこで本研究はカーネルに CLI を適用した新しいアーキテクチャを持つ PC/AT 互換コンピュータ向けオペレーティングシステムを開発し、CIL カーネルを動作させるための手法を提案した。提案手法ではインタープリタとコンパイラを併用することで、CIL カーネルが扱うどのようなデータも CLI に準じた形式で扱うことができるとともに、すべての処理を CIL カーネルに記述することができる。

第 2 章

既存技術

本研究の主旨であるオペレーティングシステムはさまざまな既存技術の上に成り立っている。これは、オペレーティングシステムが「そのときある資産を活かす」という他のソフトウェアとは一線を画する特異な性質によるものである。

そこで、本研究をオペレーティングシステムそのものの開発として位置づけるためにも、基礎となる既存技術について本章で紹介する。

2.1　PC/AT

PC/AT とは “The Personal Computer for Advanced Technologies” の頭文字を取ったコンピュータの規格である[27]。互換性のあるアーキテクチャを開発するための仕様が公開されたため、多くのメーカーから PC/AT 互換機が発売された。2006 年現在のパーソナルコンピュータの標準的な規格になっている。

現在では EFI[6] など、PC/AT を一部置き換えるための新しい規格が提案されているが、開発段階であり、十分に普及しているとは言えない。また、それら代替技術によっても PC/AT が全面的に置き換わることは考えにくい。このため、パーソナルコンピュータ向けのオペレーティングシステムを開発する場合には、まず PC/AT に対応することがもっとも良い選択肢となる。

2.1.1　構成

表 2.1 に PC/AT 互換機が持つべき構成を示す。

表 2.1　PC/AT の構成

項目	PC/AT での要求
CPU	i80286 6MHz
メモリ	256KB〜512KB
ハードディスク装置	20〜30MB
フロッピーディスク装置	1.2MB × 2 基
光学ドライブ	（なし）
グラフィックス	EGA
キーボード	84 または 101 キーボード
そのほか	シリアルポート
	パラレルポート
	AT 拡張バス

2.1.2　メモリマップ

PC/AT 互換機のメモリマップは**表 2.2** のようになっている[34][32]。メモリは様々な用途に使われており、オペレーティングシステムも全空間を自由に使うことはできない。特に `0x00100000h` (1MB) 以下の空間は非常に混み合っており、オペレーティングシステムにとって使いにくい領域となっている。

テキストモードでは†、グラフィックスバッファのうち `000B8000h`〜`000B8FA0h` の領域が画面上 80 文字× 25 行のテキストを保持するメモリ領域（テキストバッファ）になる。この領域を読み取ると画面上の文字が得られ、書き換えると画面の文字が即座に書き換わる。

† ディスプレイにテキストのみで出力するグラフィックアダプタのモード。

テキストバッファは**表 2.3** に示す構造体が画面左上から行方向を優先して 80 × 25 個並んだ領域である。

表 2.2　PC/AT のメモリマップ

開始アドレス	終端アドレス	使途
00000000h	000003FFh	リアルモード割り込みベクタ
00000400h	000004FFh	BIOS システム変数領域
00001000h	0000FFFFh	(空き領域)
00010000h	00007BFFh	(空き領域)
00007C00h	00007DFFh	MBR 展開位置
000B8000h	000BBFFFh	グラフィックスバッファ
000BC000h	000FFFFFh	BIOS 予約
00100000h	F9FFFFFFh	(空き領域)
FA000000h	FFFFFFFFh	VGA プレーン

表 2.3　テキストプレーンの文字データ構造

バイト位置	意味
0	ASCII コード
1	文字色と背景色

2.1.3　ブートシーケンス

PC/AT では、ブートシーケンスを次のような段階に分けて行う[34)]。

1. IPL の実行
   IPL (Initial Program Loader) とは、コンピュータが起動してから最初に実行されるプログラムである。その特別な性質故に ROM 相当として作られていることが多く、メモリ領域の最上位（FFFF0000h 以降など）にマップされることが多い。
   IPL は POST (Power-on Self Test) を行いコンピュータの状態を検査したのち、BIOS に制御を渡す。
2. BIOS の処理
   BIOS (Basic Input/Output System) は IPL から制御を渡されたあと、ブート可能デバイス（ハードディスクや光学ドライブなど）を初期化する。そして、起動用のデータが準備されているデバイスの MBR (Master Boot Record) を 7C00h〜7DFFh に読み込み、制御を渡す。
3. ブートストラップ・ローダの実行
   MBR には大抵ブートストラップ・ローダと呼ばれるプログラムが格納されており、このプログラムがオペレーティングシステム依存の初期化処理を開始する。

これは一般的な説明であるが、IPL と BIOS の関係については疑問がある。BIOS は実際にはサービス集合であるため、実際には「IPL が BIOS を利用しながらコンピュータを初期化する」と表現した方が適切であると考えることもできる。また、IPL と POST の実装はマザーボードベンダによるため、これらの順序については曖昧であると考えた方がよく、POST についても数回に分けて実施される場合がある。

結局のところ、IPL+POST+BIOS の動作はブラックボックスであり、オペレーティングシステムにとって重要な点は最終的に MBR が 7C00h に配置されて実行されることである。本研究においても MBR 相当の領域にブートストラップ・ローダ†を用意している。

† 4.1 節参照

2.1.4　内部デバイス

表 2.1 のような目に見えるデバイスに加えて、**表 2.4** のようなデバイスが現在の PC/AT 互換機にはある[31)14)]。

表 2.4　PC/AT の内部デバイス

名前	役割
DMAC (DMA Controller)	CPU を必要としないメモリ転送
PIC (Programmable Interrupt Controller)	周辺機器からの割り込み制御
PIT (Programmable Interrupt Timer)	設定可能な周期での割り込み
KBC (Keyboard Controller)	PS/2 キーボードと PS/2 マウスの制御
FDC (Floppy-disk Controller)	FDD の制御
ATAC (ATA Contoller)	ATA 接続機器の制御

DMA Controller (DMAC)

DMAC は DMA (Direct Memory Access) 方式で周辺機器とデータを交換するために用いられる。PC/AT 互換機には DMAC が二台備わっており、それぞれマスタ DMAC とスレーブ DMAC と呼ばれる。**図 2.1** にそれらの DMAC の接続状況を示す。

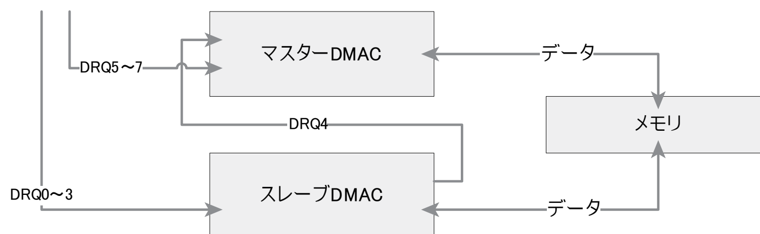


図 2.1　DMAC の接続

図より、スレーブ DMAC の出力がマスタ DMAC の入力に回っていることが分かる。これはもともと一台の DMAC が 8 ビットを転送するように作られた後に、機能拡張で 16 ビットを転送する必要が生じたために為された処置である。二台の DMAC を巧みに組み合わせて動作させることで、全体として 16 ビット幅の DMA 転送を実現する。

Programmable Interrupt Controller (PIC)

PIC は周辺機器からの割り込み要求を受け付け、優先順位に従って調停するデバイスである。PIC コントローラには 8 台までしか周辺機器を接続できないが、2 個の PIC コントローラをカスケード接続することにより合計で 15 台の周辺機器を接続できる。**図 2.2** に PIC の接続状況を示す。

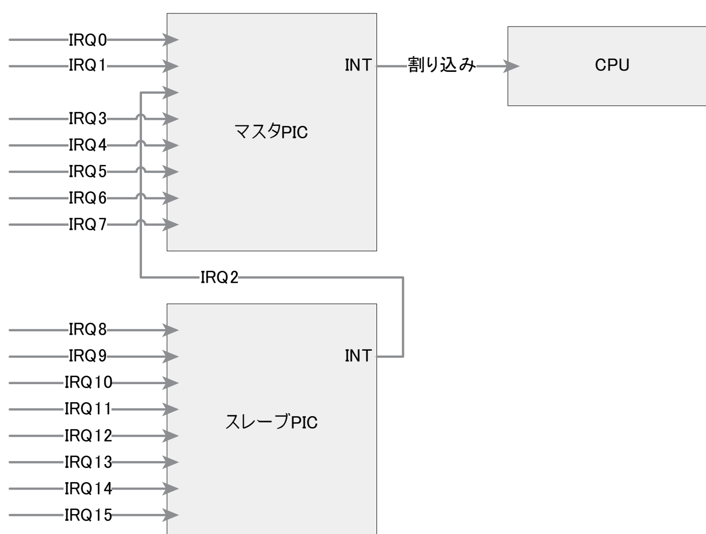


図 2.2　PIC デバイスの接続

図より、スレーブコントローラの INT（割り込み要求線）がマスタコントローラの IRQ2 として入力されている（つまり、カスケード接続されている）ことが分かる。これにより、IRQ8〜15 に接続された周辺機器の要求はスレーブとマスタ両方のコントローラを介して処理される。

Programmable Interrupt Timer (PIT)

PIT は周期的な割り込みを発生させるデバイスである。DRAM のリフレッシュ動作やビープ音に用いられるほか、オペレーティングシステムとしては一定周期の割り込みでプログラムコンテキストを切り替えることでマルチスレッド（マルチプログラミング）を実現するために用いる。**図 2.3** に PIT の接続状況を示す。

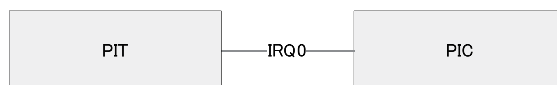


図 2.3　PIT の接続

PIT は内部レジスタとしてカウンタと周期を持っており、周期は 16 ビット整数範囲でプログラムから指定可能である。PIT が初期化されるとまずカウンタが設定値に初期化される。クロックごとにそのカウンタの値がデクリメントされ、ゼロになった瞬間に再び初期値に設定すると同時に割り込みを発生させる。クロック周波数は 1.19318MHz である。

ATA Controller (ATAC)

ATAC は接続された ATA/ATAPI デバイスを制御するために用いられる。PC/AT 互換機には ATAC がたいてい 2 個備わっており、それぞれプライマリ ATAC とセカンダリ ATAC と呼ばれる。**図 2.4** にそれらの ATAC の接続状況を示す。

ATAC には一本のケーブルでドライブを二台まで接続することができる。終端に接続したデバイスをマスタデバイス、中間に接続したデバイスをスレーブデバイスと呼ぶ。マスタデ

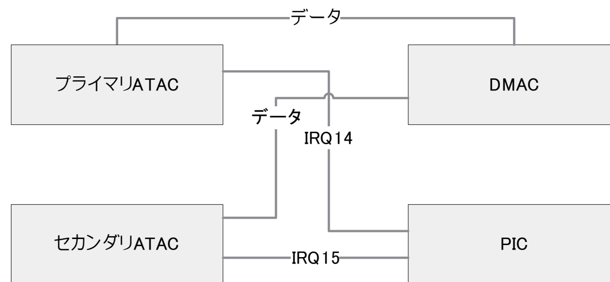


図 2.4　ATAPI コントローラの接続

バイスを接続することなくスレーブデバイスを接続することは規格上許されない[†]。一台の ATAC に 2 台のデバイスを接続できるため、標準的な構成の PC/AT 互換機では計 4 台の ATA/ATAPI デバイスを接続できる。

† 規格上は許されないが、ドライバがそのような状況を受け入れられるように作られていることがほとんどなので、動作してしまうことが多い。

FDC と FDD

FDC と FDD は**図 2.5** のようにコンピュータに接続されている。

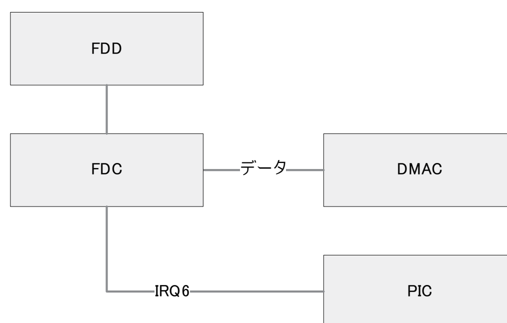


図 2.5　FDC と FDD の接続

2.1.5　フロッピーディスク

フロッピーディスクは――現在はあまり使用されなくなったが――パーソナルコンピュータにおいて最も一般的な外部記憶媒体である。容量は少ないものの、取り扱いが簡便である、対応環境が多い、起動可能であるなど、過不足ない機能を備えている。このうち、オペレーティングシステムとしては安定した対応環境と起動可能であることにより、最初の起動媒体として長く利用されてきた[†]。

フロッピーの記録単位はセクタと呼ばれ、外側から内側に向かって同心円状に配置される。セクタはコンピュータプログラムの慣習と異なり番号が 1 から始まる[†]。片面にある同心円状にある複数のセクタを指してトラックと呼び、放射方向について同じ位置にあるトラックの集合をシリンダと呼ぶ。フロッピーの場合は両面に読み書きするため、シリンダあたり 2 トラックあることになる[†]。一番外側のトラックの番号は 0 であり、それから内側に向かって 1 ずつ増えていく。

フロッピーは両面使うことができるが、表と裏という区別はしない。そのかわり、面に接している磁気ヘッド（以降単にヘッドと呼ぶ）の番号によって区別する。ヘッドは 0 番と 1 番がありそれぞれ表面と裏面に接している。

† 現在では CD-ROM ドライブからの起動環境が普及したために主流は CD-ROM 媒体になっているが、フロッピーディスクによる起動もサポートされている場合が多い。

† 第一セクタは起動プログラムを格納するために使われるため、特に MBR (Master Boot Record) と呼ばれる。

† セクタ集合という意味でシリンダという場合もある。

CHS 指定

ある位置のセクタを読み書きするための位置指定方法は、上記のような CHS (Cylinder/-Head/Sector) 指定方法以外にも方法が考えられる。もっとも簡単な方法は、セクタに順番に番号を割り当ててトラックとヘッドはセクタ番号から計算してしまうことである。しかしこのような方法が採用されず実際には CHS 指定が採られている理由は、フロッピーの物理的な記録方法とフロッピードライブの動作に関係している。

はじめにでセクタ番号が 1 から割り当てられていると述べたが、これは厳密には正確ではない。より厳密には「セクタ番号は一般的なフロッピーでは 1 から順番に割り振られているが、これに限らず、任意の番号を無秩序に割り振ることができる番号」という説明になる†。

† 極論すれば、すべてのセクタに 1 番を割り振ってもかまわない。それでオペレーティングシステムがうまく扱えるかは別問題である。

セクタ番号は、実はそれ自体がフロッピーの記録面に記録されており、その記録部分に数値を書き込んでしまえばセクタ番号はいくらでも書き換えることができる。そのため、一般にはセクタ番号からトラックとヘッドを計算することはできず、フロッピーを操作するときにはセクタ番号だけではなくヘッド番号とトラック番号も同時に指定することになる。

トラック番号は、ヘッドが放射方向のどこに位置すればよいかを指定するためにある。フロッピードライブは、トラック番号に従って（ヘッド番号やセクタ番号とは無関係に）ヘッドを移動させる。

ヘッド番号は、アクセス対象のセクタがどちらのヘッドの下に来るはずなのか指定するためにある。フロッピードライブはセクタ番号 n のセクタの転送を命令されると、命令に含まれているヘッダ番号のヘッダからデータを読み取り続ける。そして、セクタ番号として n を持つセクタを発見すると、そこから読み取りや書き込みを開始する。

2.1.6 光学ドライブ

ATAPI 光学ドライブは**図 2.6** のような順序でコンピュータに接続されている。

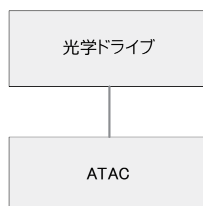


図 2.6　ATAPI 光学ドライブの接続

ATAPI 光学ドライブはデータ転送のモードが大別して 2 つあり、それぞれ PIO 転送モードと DMA 転送モードと呼ばれる。PIO 転送モードでは CPU の I/O 命令を使って、バイト単位ずつプログラムが転送していく。DMA 転送モードでは DMAC を利用して、一定量のデータを CPU を介さずに主記憶メモリに転送する。

PIO 転送モードはプログラムが簡単でどのデバイスも必ずサポートしているという利点があるが、非常に処理効率が悪く実用的ではない。DMA 転送モードはプログラムが複雑になるが、高速に転送することができる。

2.1.7 El Torito

El Torito[15)] は、IBM 社と Phoenix 社が共同で考案した起動可能 CD-ROM の規格である。この規格は、それまでに起動媒体として用いられていたハードディスクやフロッピー

ディスク用のブートストラップ・ローダを CD-ROM でも動作されるために策定された。

もともと、ハードディスクやフロッピーディスクでは MBR のサイズが 512 バイトであり、1 セクタが 2048 バイトある CD-ROM とは合致しない。しかし、El Torito ではハードディスクやフロッピーディスクのイメージをそのまま CD-ROM に記録しておき、BIOS の層で CD-ROM がハードディスクやフロッピーディスクであるかのようにアクセスを切り替える。このため、ハードディスクやフロッピーディスク用のブートストラップ・ローダをそのまま CD-ROM 用として利用することができる。

2.2　Intel Architecture 32 (IA-32)

IA-32[5),7),8)]は Intel 社の Pentium プロセッサなどと互換性のあるプロセッサアーキテクチャの名前である。i80486[30)]、Pentium、Pentium　、Pentium　、Pentium4、Pentium M、Celeron、Xeon などのプロセッサがこの規格に該当する。（SSE[*1]命令など各モデル固有の仕様も存在するが、基本的な動作は IA-32 として総括できる。）

2.2.1　レジスタ

IA-32 のレジスタは汎用レジスタとセグメントレジスタ、および制御レジスタの三種類に分けることが出来る。

表 2.5 に汎用レジスタを示す。これらのレジスタは 32 ビット幅であるが、IA-32 では他

表 2.5　IA-32 の汎用レジスタ

名前	ビットサイズ	目的
EAX	32	汎用
ECX	32	汎用
EDX	32	汎用
EBX	32	汎用
ESP	32	スタックポインタ
EBP	32	フレームポインタ
ESI	32	汎用
EDI	32	汎用

のサイズでこれらのレジスタにアクセスできるように、これらのレジスタの一部分に別名を付けている（**表 2.6**）。この別名によって、たとえば `EAX` は下位 16 ビットには `AX` という名前で、`AX` の下位 8 ビットには `AL` という名前でアクセスすることができる。

表 2.7 にセグメントレジスタを示す。セグメントレジスタはセグメント方式によるメモリ管理に使用する。`CS` と `DS`、`SS` はプログラムの実行中に常に参照されるが、値が変化することは少ない。ES はデスティネーションセグメントを指定するときに使われる場合がある[†]。GS と FS はあまり使われない。

† たとえば、メモリ領域をコピーする命令に対して、DS がソースセグメントとして、ES がデスティネーションセグメントとして使われる。

表 2.8 に、プロセッサの動作を変更するために使う制御レジスタを示す。これらのレジスタに格納される値の各ビットは特別な意味を持っており、特権を持つプログラム（一般にはオペレーティングシステム）しか変更できない。

*1 Streaming SIMD (Single Instruction / Multiple Data) Extensions

表 2.6　IA-32 汎用レジスタの重ね合わせ

<table>
<tr><td colspan="3">EAX</td><td colspan="3">ECX</td></tr>
<tr><td rowspan="2"></td><td colspan="2">AX</td><td rowspan="2"></td><td colspan="2">CX</td></tr>
<tr><td>AH</td><td>AL</td><td>CH</td><td>CL</td></tr>
<tr><td colspan="3">EDX</td><td colspan="3">EBX</td></tr>
<tr><td rowspan="2"></td><td colspan="2">DX</td><td rowspan="2"></td><td colspan="2">BX</td></tr>
<tr><td>DH</td><td>DL</td><td>BH</td><td>BL</td></tr>
<tr><td colspan="3">ESP</td><td colspan="3">EBP</td></tr>
<tr><td></td><td colspan="2">SP</td><td></td><td colspan="2">BP</td></tr>
<tr><td colspan="3">ESI</td><td colspan="3">EDI</td></tr>
<tr><td></td><td colspan="2">SI</td><td></td><td colspan="2">DI</td></tr>
</table>

表 2.7　IA-32 のセグメントレジスタ

名前	ビットサイズ	目的
CS	16	コードセグメント
DS	16	データセグメント
SS	16	スタックセグメント
ES	16	汎用セグメントレジスタ
GS	16	汎用セグメントレジスタ
FS	16	汎用セグメントレジスタ

表 2.8　IA-32 の制御レジスタ

名前	ビットサイズ	目的
CR0	16	制御レジスタ
CR1	16	制御レジスタ
CR2	16	制御レジスタ
CR3	16	制御レジスタ
CR4	16	制御レジスタ

2.2.2　IA-32 プロセッサの動作モード

IA-32 プロセッサはリアルモード、プロテクトモード、仮想 86 モード、システム管理モードと呼ぶ 4 つの動作モードを持っている。このうち特に重要なプロテクトモードについては 2.2.3 項で改めて説明する。

リアルモード

CPU は電源投入直後に必ずリアルモードになる。リアルモードというのは 8086 プロセッサなどと互換性のあるモードで、最新の機能などは無効化されている。現在のコンピューティング環境としてはまったく使えないが、プロセッサの互換性という目的のために用意さ

れている。

リアルモードはアドレス空間が 20 ビット（リニアアクセス可能なのは 16 ビット）であり、セグメント方式のメモリ管理を行うことができる。リアルモードはプロテクトモードに比べて貧弱であまり利点がないが、コンピュータの起動直後はプロセッサはリアルモードで動作しているため、オペレーティングシステムは（一時的にせよ）必ず対応する必要がある。

また、一般的な BIOS のサービスがリアルモード向けに作られているため、BIOS の機能を利用する場合にはリアルモード（または仮想 86 モード）であることが必要になる。BIOS のサービスにはディスクの読み込みや画面操作など便利なものがいくつもあり、それを利用する場合にはリアルモードにせざるを得ない。

仮想 86 モード

仮想 86 モードはプロテクトモードで動作するオペレーティングシステムがリアルモード向けプログラムを安全に動作させることができるように用意されたモードである。Windows95/98 などで MS-DOS プログラムを動かすときなどに使われていたが、現在ではリアルモード向けのプログラムが使われなくなったため、積極的に使われることはない。

システム管理モード

システム管理モードは、コンピュータの動作に関わる重大な事象が起こった場合にそれを処理するための特別なモードである。スタンバイやスリープなど電源管理のイベントなどで使われる。

2.2.3　プロテクトモード

現在のオペレーティングシステムが動作するモードがプロテクトモードである。プロテクトという響きからは保守的なイメージになってしまうが、これは「システムの完全性を保護できる」ことに由来する。

特権レベル

IA-32 プロセッサは実行中のアプリケーションプログラムがオペレーティングシステムに干渉しないように、影響を与える命令を特権命令として分類し、4 段階の特権レベルによって動作を制約する（**図 2.7**）。特権レベルは下位レベルが上位レベルより全ての点で強い制約

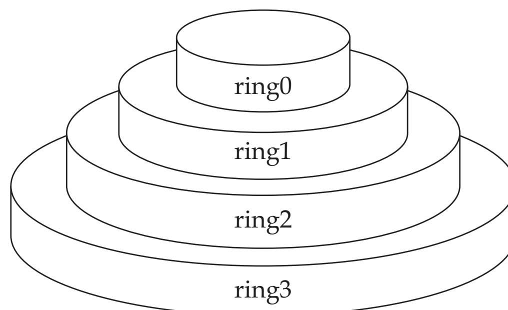


図 2.7　IA-32 の特権レベル

を受けることから “ring” とも呼ばれ、`ring0` がカーネルのための制約のないレベル、`ring3` がユーザプログラムのためのもっとも強い制約のレベルである。

割り込みと例外

CR4 の割り込みフラグがマスクされていないとき、プロセッサの INT ピンに信号が立ち上げると割り込みが受け付けられる。割り込みが受け付けられると割り込み番号に対応する IDT (Interrupt Description Table) 内のエントリが調べられ、エントリに書かれている割り込みハンドラに制御が移される。（この段階で、割り込み直前に実行されていたプログラムは中断する。）

割り込みハンドラは、すべての割り込み処理が終わったあと `IRET` 命令によって制御を戻す†。このとき、ほとんどの割り込みハンドラは割り込み直前の実行コンテキストが復元されるように（つまり元のプログラムが再開するように）復帰するが、周期タイマーハンドラなどではマルチスレッド（マルチプログラミング）のために、割り込み前とは異なる実行コンテキストに復帰する場合がある。

† 通常のプログラムが使用するのは RET または RETF 命令。

2.3 Microsoft Windows

CLI は Microsoft 社が主導して策定された規格であるため、Windows の影響を強く受けている。ここでは、特に CLI にも採り入れられた Windows に由来する技術を説明する。

2.3.1 Portable Executable (PE) 形式

Windows においてプログラムを格納する標準ファイルは `.exe` および `.dll` 形式だが、このどちらも内部的には PE というフォーマットに従っている。PE はローダーにプログラムの展開に関わる情報を提供し、効率的なメモリの利用や動的リンクを可能にする[12)]。

PE 形式はプログラムを納めているためコードが含まれていることは当然だが、それ以外にもプログラムが必要とする情報として

- 命令列
- 静的データ領域（要初期化）
- 静的データ領域（未初期化）
- エクスポートシンボル
- インポートシンボル
- 再配置情報
- デバッグ情報

などがあり、PE はこれらの情報をうまく保存するための構造を持っている。

この構造はプログラムのレイアウトを柔軟にする一方、プログラムをロードするときに複雑な処理が必要になってしまう。たとえば、命令列を納めた領域を 100000h (1MB)〜200000h (2MB) に、データを納めた領域を 10000000h (100MB)〜10200000h (102MB) に配置するような DLL があったとき、実行時のメモリレイアウトをそのままファイルにしてしまうと 102-1=101MB ものファイルになってしまい、98MB が無駄な領域になってしまう。

この問題を解決するために、PE にはセクションという概念がある。上記の例ではセクションが二つ定義され、1MB に配置されるべき 1MB のセクションが一つと、100MB に配置されるべき 2MB のセクションが含まれている。ファイルのサイズは 1+2=3MB 程度になり、効率的であるといえる。

2.4　Common Language Infrastructure (CLI)

ECMA-335[20]では、Partition I: Concepts and Architecture の第 1 章 “Scope” 内で、CLI について次のように述べている。（参考までに日本語訳を示す。）

> This International Standard defines the Common Language Infrastructure (CLI) in which applications written in multiple high-level languages can be executed in different system environments without the need to rewrite those applications to take into consideration the unique characteristics of those environments.
>
> この国際標準は Common Language Infrastructure (CLI) を定義する。CLI では複数の高級言語で書かれたアプリケーションが様々な環境に対しても独特の特徴に影響されることなく実行されうる。

また、同編第 6 章 “Overview of the Common Language Infrastructure” 内で次のように述べている。

> The Common Language Infrastructure (CLI) provides a specification for executable code and the execution environment (the Virtual Execution System) in which it runs. Executable code is presented to the VES as modules.
>
> CLI は実行可能コードとそれを実行する実行環境 (Virtual Execution System) のための仕様を提供する。
>
> （中略）
>
> At the center of the CLI is a unified type system, the Common Type System that is shared by compilers, tools, and the CLI itself. It is the model that defines the rules the CLI follows when declaring, using, and managing types. The CTS establishes a framework that enables cross-language integration, type safety, and high performance code execution.
>
> CLI の核心はコンパイラ、ツール、そして CLI 自身によって共有される統一型システム (CTS) である。CTS は CLI が宣言や利用、および型を管理するときに従うべき規則を定義するモデルである。CTS は異種言語の統合、型安全性、高性能コード実行を可能にするフレームワークを構築する。

また、同編 6.2.3 項 “Summary” 内で次のように述べている。

> The CTS is about integration between languages: using another language ’ s objects as if they were one ’ s own. The objective of the CLI is to make it easier to write components and applications in any language. It does this by defining a standard set of types, by making all components fully self-describing, and by providing a high performance common execution environment. This ensures that all CLI-compliant system services and components will be accessible to all CLI-aware languages and tools.
>
> CTS は言語間の統合—他の言語のオブジェクトを何物によるものかに依らず使用するためのものである。CLI の目的はコンポーネントやアプリケーションをどの言語においても書きやすくすることである。これは標準の型セットを定義し、すべてのコンポーネントを完全に自己記述させ、高性能な実行環境を用意することによって成される。これによって、CLI 互換系のサービスとコンポーネントがすべての CLI 向け言

語やツールへアクセスできるようになることを保証できる。

ECMA-335 では、CLI 以外にも VES、CTS、CLS など多数の用語を定義しているが、説明にあたっては煩雑になってしまうため、本論文ではよほどの理由がない限り CLI としてまとめることにする。

2.4.1 オブジェクト指向

CLI はオブジェクト指向（特にクラス指向）であり、強く型付け†されている。CLI のクラス継承関係は次のような規則を持つ。

† 強く型付けされているとは、ある変数の型がコンパイル時に自明であり、自明である範囲でその変数が扱われているということである。CIL のほかに C 言語や Java が例としてあげられる。

- 親クラスを必ず一つだけ持つ。（ただし Object[*2]だけは持たない。）
- インターフェイスは 0 個以上複数実装できる。

C++ とは違い、（一般のクラスでは）親クラスが必ず一つ必要であり、複数持つことはできない。これによってクラス階層図は Object を頂点とする樹形図になり、C++ よりもクラス情報の扱いが格段に単純される。

クラス階層が樹形に制限されることは CLI にとっては単純化という利点があるが、ソフトウェアのデザインには厳しい制限となってしまう。そこで、インターフェイスはいくつでも実装できるように定め、クラス階層以外でのソフトウェアの設計を可能にしている。

2.4.2 アセンブリ

Windows ではプログラムを PE 形式で記録していたが、CLI では PE 形式を拡張したものに記録している。CLI では従来型のプログラムも制限付きで実行できるようになっているため、従来の PE の機能は従来のプログラム向けに利用し、拡張された CLI 向けの部分は CLI プログラムに利用するようになっている。そのため、拡張された部分と従来との部分で機能的には重複するものがある。

CLI のプログラムファイルをアセンブリファイル（または単にアセンブリ）という。アセンブリは普通はファイルだが、場合によってはメモリ上のデータだったり、プログラムが動的に生成したものであることもある。

アセンブリにはメタデータが格納されているため、CLI プログラムはリフレクションを提供することができる。

2.4.3 スロット

スロットの説明をするために 3 つのコードを用いることにする。**テキスト 2.1** はインターフェイスの定義である。**テキスト 2.2** は基本クラスの定義である。**テキスト 2.3** はスロットの説明に用いるクラスの定義である。

`MyClass` のスロットを**表 2.9** に示す。順序とはスロット内の位置であり、宣言とはそのメソッドがはじめに定義された位置、定義とは実際に使われるメソッドの位置を示す。

あらゆるクラスは Object[*3]から派生しているため、スロットの最初の 3 エントリはかならず `Equals` と `GetHashCode`、`ToString` になる。表 2.9 では、これらのメソッドが最初に定義された位置が Object であることを宣言の列に示している。定義の列はそのメソッドが呼

[*2] System.Object
[*3] System.Object

テキスト 2.1　スロットのサンプルコード 1

```
1  interface MyInterface1 {
2      void MethodA();
3      void MethodB();
4  }
5
6  interface MyInterface2 {
7      void MethodC();
8  }
```

テキスト 2.2　スロットのサンプルコード 2

```
1  abstract class MyBase : MyInterface1, MyInterface2 {
2      public virtual void Virtual1() {
3          Console.WriteLine("MyBase");
4      }
5      public virtual void Virtual2() {
6          Console.WriteLine("MyBase");
7      }
8      public abstract void Abstract1();
9      public void MethodA() {
10         Console.WriteLine("MyBase:MethodA");
11     }
12     public virtual void MethodB() {
13         Console.WriteLine("MyBase:MethodB");
14     }
15     public abstract void MethodC();
16 }
```

テキスト 2.3　スロットのサンプルコード 3

```
1  class MyClass : MyBase {
2      public override string ToString() {
3          return "MyClass:ToString";
4      }
5      public override void Virtual2() {
6          Console.WriteLine("MyClass:AbstractMethod");
7      }
8      public override void Abstract1() {
9          Console.WriteLine("MyClass:AbstractMethod");
10     }
11     public override void MethodC() {
12         Console.WriteLine("MyClass:MethodC");
13     }
14 }
```

表 2.9　スロットの要素

順序	宣言	定義
1	Object:Equals	Object:Equals
2	Object:GetHashCode	Object:GetHashCode
3	Object:ToString	MyClass:ToString
4	MyBase:Virtual1	MyBase:Virtual1
5	MyBase:Virtual2	MyClass:Virtual2
6	MyBase:Abstract1	MyClass:Abstract1
7	MyBase:MethodB	MyBase:MethodB
8	MyBase:MethodC	MyClass:MethodC
9	MyInterface1:MethodA	MyBase:MethodA
10	MyInterface1:MethodB	MyBase:MethodB
11	MyInterface2:MethodC	MyClass:MethodC

び出されたとき、実際に実行されるコードの定義位置を示している。たとえば、`ToString` は `MyClass` でオーバーライドされているため定義が `MyClass` になっている。

4〜8 番目のエントリは MyBase で定義されている仮想関数（virtual）と抽象関数（abstract）である。このうち、7〜8 番目のメソッドはインターフェイスに由来するものであるが、ここではインターフェイスとは特に関係ない。9〜10 番目および 11 番目のエントリについては 2.4.4 項で説明する。

表 2.9 を使用すれば、たとえば `MyBase:Virtual2` を呼び出す処理は、対象のインスタンスの実際の型がなんであれスロット中の 5 番目のエントリに書かれているメソッドを呼び出すことで実現できる。なぜなら、MyBase から派生するすべてのクラスはスロットの最初から 8 個は Object と MyBase に関するエントリだからである。このように、スロットを使うことで仮想関数呼び出しを実行時に実際の型を意識することなく処理できるようになる。

2.4.4　インターフェイスマップ

表 2.9 の 9〜10 番目および 11 番目のエントリはインターフェイスマップで使用されるエントリである。MyClass に関する仮想メソッドと抽象メソッドのエントリは 1〜8 番目で完成しているが、それらはインターフェイスとは無関係に整列されているためインターフェイスマップとしては使用できない。そこで、インターフェイスごとに改めてメソッドを並べておき、インターフェイスによる呼び出しに備えている。

表 2.10 には、表 2.9 において `MyInterface1` のインターフェイスマップが 9 から始まっていることを反映して、基準位置の列に 9 として示されている。

表 2.10　MyClass のインターフェイスマップ

名前	基準位置
MyInterface1	9
MyInterface2	11

2.4.5 Common Intermediate Language (CIL)

CIL は CLI が定義する仮想実行環境のアセンブリ言語としての役割を持つ。CLI に対応した言語は最終的には CIL にコンパイルされることになるため、CLI の実行系は（各々のプログラミング言語に対応するのではなく）単に CIL を実行する方法を持てばよいことになる。

† 2.2 節参照

CLI と CIL が想定する実行環境はスタックマシンである。IA-32[†]と異なり、スタックマシンにはレジスタが一切無いため、すべてのオペランドはスタックを通して受け渡すことになる。ほかにデータを格納する場所としては、ヒープ領域や局所変数、静的変数などがある。

CIL の命令はふたつのグループに区分できる[20]。ひとつは基本命令 (Basic Instructions) で、もう一つはオブジェクト指向命令 (Object-oriented Instructions) である。

基本命令は、単純なスタック操作、算術演算などである。次に代表的な命令の一覧を示す。（分類は筆者の私見による。）

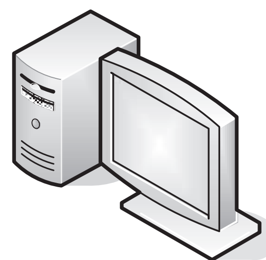
- 算術演算と論理演算
  add, mul, and, or, shl, shr, conv, etc.
- 分岐と比較
  br, beq, bgt, ceq, etc.
- ロードとストア
  ldc, ldarg, starg, ldarga, ldloc, stloc, ldloca, ldind, stind, cpblk.
- 関数呼び出し
  call, calli, ret.
- メタデータ
  ldftn

オブジェクト指向命令には次のようなものがある。（分類は筆者の私見による。）

- ヒープ操作
  box, unbox, newarr, newobj, ldstr, initobj, cpobj, stobj, ldobj.
- 型演算
  castclass, isinst, ldtoken, sizeof. mkrefany, refanytype, refanyval.
- オブジェクト要素操作
  ldelem, ldelema, ldfld, ldflda, ldsfld, ldsflda, stelem, stfld, stsfld, ldlen.
- 仮想呼び出し
  callvirt, ldvirtftn.
- 例外処理
  rethrow, throw.

第 3 章

アーキテクチャ



本章では開発したオペレーティングシステム "CooS" のアーキテクチャを紹介することによって、全体像を明らかにすることを目的とする。アーキテクチャはソフトウェアの骨組みともいえる概念であり、そこにはソフトウェアの目的・思想・限界などが反映される。

本章では具体的なプログラム要素を取り上げた説明はしないが、次章以降に説明する実装は本章のアーキテクチャに則って行われることになる。

3.1　コンセプト

本研究で開発したオペレーティングシステムの名前を CooS（クース）とした。研究目的に鑑み、他のオペレーティングシステムとの互換性や実用目的の機能は重視せず、積極的に CLI を活かすようなアーキテクチャにした。

メモリ保護

従来手法では、実行するプログラムは機械語で書かれていたため、ハードウェアによってメモリ保護が行われていた。IA-32 ではセグメント方式やページング方式にメモリ保護のための機能を持ち、プログラムが不正なアクセスを行おうとしたときにオペレーティングシステムが介入できるようにしている。

しかし、CLI においては、1) 中間言語を用いるためプログラムがどのような命令列を含むのか事前に検査できる、2) 一般の CIL プログラムではメモリアドレスを操作するような記述ができない、という特徴があるため、そもそもメモリ破壊を起こす可能性のある操作は記述できないか、記述した場合には実行前にそれを知ることができる。そのため、ハードウェアによるメモリ保護がなくても、ソフトウェアによってメモリ保護を実現することができる。

メモリ保護は性能を低下させる原因となるため、ハードウェアによるメモリ保護が不要になればその分高速に実行できると考えられる。

特権モードと特権命令

特権命令制限についても、従来手法ではメモリ保護と同様の理由でハードウェアによる制限を行える必要があった。そのために IA-32 では特権モードと呼ばれる仕組みが導入されており、実行中のプログラムを ring 0（無制限）から ring 3（もっとも強い制限）に分けて実行する。

しかし、CIL には CPU 依存命令などは定義されていないため、そもそも特権命令を記述することができない。Managed C++ などを使ってアセンブリに機械語を埋め込んだ場合でも、そのプログラムが機械語を実行する可能性があることを実行前に検査できるため、そのようなプログラムを一般のアプリケーションとして動作させることを制限できる。このため、提案手法においては特権モードなどによって特権命令の実行を制限する必要はない。

最適化

最適化はもっとも性能向上に寄与すると考えられる事項である。一般的にプログラムは解析が難しいバイナリ表現であり、それを実行時に改変することはほぼ不可能である。しかし、CLI ではプログラムが中間言語で表現されておりメタデータも付随しているため、実行時に環境に合った最適化をすることができる。

さらに CooS の場合には、特権命令制限が無いこととを併せて、システムコールをインライン化†し、アプリケーションで特権命令を利用できるようになると考えられる。たとえば、特権命令による周辺機器との入出力を考える。従来はシステムコールを呼び出して特権モードに切り替えていたのが、提案手法では JIT コンパイルなどによってアプリケーションがシステムコールを呼び出す箇所に特権命令を埋め込むことが可能になり、性能が向上すると考えられる†。

† ある手続きを呼び出しているとき、通常の呼び出し手順を省略して、その手続き自体を呼び出し元に埋め込んでしまうこと。

† 埋め込み処理はオペレーティングシステムの管轄であるため、セキュリティが脅かされることはない。

メモリモデル

既存オペレーティングシステムにおいて、メモリは C モデルで管理されることが多かった。C モデルとは C 言語でのメモリ操作を意識し、ある確保されたメモリ領域について、1) アドレスが変化することはなく、2) どのような型としてもアクセスされる可能性があり、3) 明示的に解放されるまでは使用中とみなすというモデルとする。C モデルの管理ではメモリが断片化する可能性が常にあり、確保と解放にコストがかかってしまう。

しかし、CLI が採用する GC (Garbage Collection) モデルは、断片化をごみ収集手順によって解消でき、確保にかかるコストは最小 O(1) であるとされる[2]。CLI には「固定された (pinned) メモリ領域」など GC の動作を妨げる仕様も含まれているため単純に GC の利点を得られるとは言えないが、性能向上できる可能性はあると考えられる。

まとめ

コンセプトについてまとめたものを**表 3.1** に示す。

表 3.1　アーキテクチャの対応

項目	従来手法	提案手法
メモリ保護	必要	不要
特権命令制限	必要	不要
最適化	プログラム単位で事前に	全体を実行時に
メモリモデル	C モデル	GC モデル

3.2　ハードウェア要求

表 3.2 に CooS が要求する構成を示す。

表 3.2　CooS のハードウェア構成

項目	CooS の要求
CPU	IA-32 互換プロセッサ
メモリ	128MB〜4GB
ハードディスク装置	使用しない
フロッピーディスク装置	1.44MB × 1 基
光学ドライブ	ATA/ATAPI[21]接続× 1 基
グラフィックス	VBE[25]2.0 に対応した VGA
キーボード	日本語 109 キーボード
そのほか	（なし）

CPU について、PC/AT では i80286 を想定しているが、本研究では使用している命令が i80286 のものだけではなく PentiumIII 以降にしか含まれていないものも用いているため、PentiumIII 以降の IA-32 互換プロセッサを必要とする。メモリは 7 章において 128MB で動作させ試験を行っているため、128MB あれば動作可能であるとした。ハードディスク、

シリアルポート、パラレルポート、AT 拡張バス、および（PC/AT には含まれていないが）PCI バスなどは使用していない。CooS は CD-ROM 媒体から起動するため、CD-ROM が読み込み可能な ATAPI 接続の光学ドライブを必要とする。グラフィックスについて、本研究では VBE2.0 に対応したグラフィックアダプタが使用可能でなければならない。とくにグラフィックモードで描画する場合には、フラットモデル[*1]に対応している必要がある。キーボードについて、PC/AT は英語キーボードを想定しているが、本研究では日本語キーボードを想定している。両者は一部のキーマップが異なったり欠けているだけで完全に異なるものではないため、CooS でも英語キーボードを使うことはできるが、一部の応答が正常ではなくなる。

全体として、CooS が要求する水準はだいたい 2004 年頃の平均的な PC と同じくらいであり、2006 年現在の PC の水準を下回っている。このため、パーソナルコンピュータ向けオペレーティングシステムとして妥当な水準の要求であるといえる。

3.3　ソフトウェア構成

ここで、オペレーティングシステムとして必要なプログラムを決定するために、つぎのような事柄を考慮した。まず、PC/AT の仕様により次のプログラムが必ず必要になる。

- ブートストラップ・ローダ
- 機械語カーネル

一方、CLI を適用したオペレーティングシステムを作ろうとすれば、次のプログラムが開発される。

- CIL カーネル

この CIL カーネルは、コンピュータは機械語しか実行できないため、次のいずれかの方法によって実行されなければならない。

1. インタープリタによって、CIL のまま実行される。
2. JIT コンパイルによって、実行時に機械語に変換する。
3. AOT コンパイルによって、あらかじめ機械語に変換しておく。

方法 1 では、インタープリタ自体は C++ などを使って記述する必要がある。（機械語で書かれている必要があるため。）しかし、この方法では決して C++ で記述された従来と変わらない性質のプログラムがカーネルに残存することになり、たとえコード量は小さいとしてもカーネル全てを CLI で記述したことにはならない。また、処理速度も機械語実行に比べて決して速くならないため、カーネルとしては実用的ではないと考えられる。

方法 2 では、JIT コンパイラは機械語で書かれている必要がある。しかし、この手法についても (1) と同様に機械語で書かれたコンパイラがカーネルの一部として機能しなければならず、やはりカーネル全体が CLI で記述できるわけではない。

方法 3 では、AOT コンパイラはある程度[†]まで処理されたデータを生成する必要があり、そのデータをオペレーティングシステム起動時に解釈して適切に配置する必要がある。この手順は自明ではないため、上記 (1) や (2) のように既存手法の延長として実現することは出来ない。また、AOT コンパイルによって用意したデータを起動時に利用する方法や、それ

† たとえば、起動時に必要なすべてのデータを満たすデータを用意すれば、ブート時のロード処理は簡単になるが、AOT コンパイルは複雑になる。簡素なデータのみ用意すれば、AOT コンパイルは簡単だが、ロード処理は複雑になる。

*1 画面バッファにリニアアクセス可能なモード。

を CLI 規格内に収める方法も自明ではない。

このように各方法とも長所・短所があるが、方法 2 および方法 3 の機械語を得られるという特徴は、カーネルであることを考えると必須であると考えられる。たとえば、ハードウェアの処理などでは実時間で制約のある場合があり、そのような時は機械語による高速実行が求められる。また単純に考えても、カーネルがインタープリタによって実行されている限り、そのインタープリタを含む部分がカーネルとなってしまい、そもそもの意味が無くなってしまう。

しかし、カーネルは CIL で書かれているため、機械語はコンパイルしないと得ることができない。コンパイルを事前に行うとすれば、その結果をブート時に誰が、どのように利用するかが問題になってしまう。

インタープリタとコンパイラによる CIL カーネルの実行

本手法ではこの問題をインタープリタとコンパイラを組み合わせて用いることによって解決することにした。その基本的な仕組みを**図 3.1** に示す。

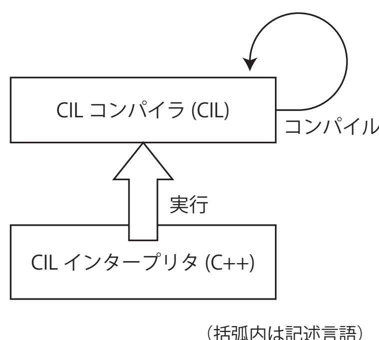


図 3.1　インタープリタとコンパイラの関係

CIL インタープリタで CIL コンパイラを駆動し、CIL コンパイラが自分自身をコンパイルすることで、CIL インタープリタに頼らずに実行可能な CIL コンパイラを手に入れる†。CIL インタープリタは C++ で記述するから CLI カーネルの一部としては不適格だが、CIL コンパイラは CIL で書かれているため CLI カーネルとして適格である。CIL インタープリタは CIL コンパイラのコンパイルが終了した時点で破棄できる。

† 6.3 節参照

以上で、CIL カーネルの実行可能なコードを手に入れることはできるが、実際にコンパイルすると、メタデータに関する命令の処理はランタイムプログラムへ依存してしまう。コンパイルしたコードを動作させるためには、さらに、メタデータの配置についても考慮する必要がある†。

† 6.2 節参照

ここで、インタープリタとコンパイラはそれぞれ単体のプログラムとしてはすでにいくつものソフトウェアが存在し、本提案手法で用いるものもそれらと大きな違いはない。本手法の課題は、外部ランタイムのない状態で、インタープリタとコンパイラをどう組み合わせて動作させるかということである。

二つのカーネル

インタープリタとコンパイラそれぞれについて、器となるカーネルを用意した（**図 3.2**）。C++ による機械語のカーネルはレガシーカーネル、主に C#による CIL カーネルはマネー

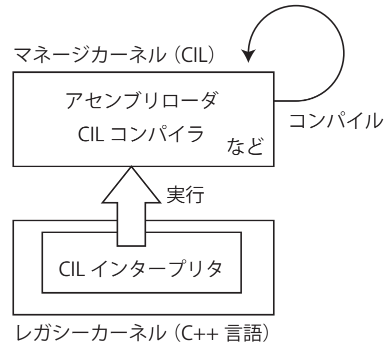


図 3.2　カーネルと内部コンポーネントの関係

ジカーネルとした。そして、インタープリタはレガシーカーネル内に、コンパイラはマネージカーネル内に納めることにした。

3.4　メモリモデル

表 3.1 に示したように、CooS ではハードウェアによるメモリ保護をしない。また、実験的な開発であり、アプリケーションの動作は主目的ではないためアドレス空間の分離も行わないことにした。採用したメモリ構造の概要を**図 3.3** に示す†。ここで P1 と P2 はそれぞれ

† 詳細は 5.2 節や 6.1 節で述べている。

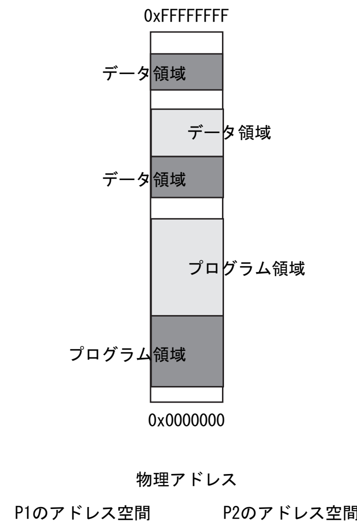


図 3.3　メモリ構造概要

プログラムを表す。P1 と P2 ともにアドレス空間を共有しており、原理的には互いのデータにアクセスできる。(もちろんアプリケーションがそのような操作をすることは許可されない。)

全てのメモリ領域はすべて**ブロック**として確保される。ブロックは**表 3.3** のような構造を持つ。ただし、T はブロックが保持するデータの型であり、`sizeof(T)` は純粋に T を格納するために必要なメモリサイズである。また、**型オブジェクト**とはこのブロックを指す Object 型変数 p に対して、**テキスト 3.1** の実行結果として得られるオブジェクトである†。また、4 のパディングとは、全体のサイズを 4 の倍数にするために付加される意味のないデータである。(以降、N のパディングという言葉は、対象のサイズを N の倍数にするための同様の

† 端的に言えば T を表現する System.Type オブジェクトを指す。

表 3.3　ブロックの構造

バイト位置	型	内容
0	Int32	ブロックのサイズ
4	Int32	型オブジェクト
8	Byte[sizeof(T)]	データ領域
8+sizeof(T)	Byte[]	4 のパディング

テキスト 3.1　型オブジェクトの取得

```
1  p.GetType()
```

データを意味するものとする。）

GC を想定しているため、ブロックの確保はメモリの最初から切り取ることで実現できる。解放は GC によるが、本研究ではメモリ管理アルゴリズムの評価が目的ではないため実装しなかった。

3.5　カーネルブリッジ

本手法では起動時の一部の処理についてマネージカーネルからレガシーカーネルの機能を呼び出す必要が生じる。このために IKernel という通信のためのインターフェイスを定義し、通信のためのコードをカーネルブリッジというコンポーネントに格納することにした。（完全な定義を付録に収録している。）

IKernel は関数ポインタの配列として定義され、配列の i 番目にどのような目的の関数を格納するかは IKernel のコントラクト†としてあらかじめ決められている。この構造は C++ 言語では**テキスト 3.2** に示すような純粋仮想関数のみで構成されたクラスとして表現できる。この表現を発展させたものが COM[18] などにも見られ、言語間の違いを吸収しやすい仕

† ある公開された機能が提供することとしないこと、利用側がしてよいこととしてはいけないことなどを指してコントラクト（契約）という。仕様と似た意味だが、コントラクトは提供側と利用側の規約という点が注目される。

テキスト 3.2　IKernel の宣言

```
1  // This is acceptable as either plain C++ and Managed C++.
2  struct IKernel {
3      virtual int getVersion() = 0;
4      virtual void Write(char ch) = 0;
5  };
```

組みである。

レガシーカーネルは**テキスト 3.3** のように実装を行い、マネージカーネルの起動開始直後にインターフェイスのポインタを**テキスト 3.4** のように通知する。ただし、ExecuteManaged-Code は CIL コードをインタープリタによって実行する関数である。また、CooS.Bridge は IKernel ポインタを受け取って、後のマネージカーネルからレガシーカーネルへの通信を処理するクラスである。

CooS.Bridge クラスの実装を**テキスト 3.5** に示す。CooS.Bridge は Managed C++ で書かれているので、テキスト 3.2 で示した C++ での IKernel の定義をそのまま再利用でき、関数ポインタ経由での関数呼び出しも CLI との不整合なく行うことができる。

テキスト 3.3　IKernel の定義

```
// C++
struct MyKernel : IKernel {
    virtual int getVersion() {
        return 1;
    }
    virtual void Write(char ch) {
        printf("%c", ch);
    }
};
```

テキスト 3.4　IKernel の受け渡し

```
IKernel* kernel = new MyKernel();
ExecuteManagedCode("CooS.Bridge:SetKernel", kernel);
```

テキスト 3.5　CooS.Bridge の実装

```
// Managed C++
__gc class Bridge {
    IKernel* kernel;
private:
    void SetKernel(IKernel* kernel) {
        this->kernel = kernel;
    }
public:
    __property int get_Version() {
        return kernel->getVersion();
    }
    void Write(wchar_t ch) {
        // Don't support non-ASCII characters
        kernel->Write((char)ch);
    }
};
```

3.4 の処理以降は、CooS.Bridge クラスを経由することで、マネージカーネルがレガシーカーネルを呼び出すことができるようになる。ただし、実際には CooS.Bridge クラスが直接利用されることは少なく、3.6 節で述べるパッチを経由して利用されることがほとんどである。

3.6　パッチ

マネージカーネルは最終的に単独でカーネルとして機能するが、カーネルの起動時にのみレガシーカーネルに移譲したい処理がある。たとえば**テキスト 3.6** のような画面に文字を出力するメソッドを考える。このメソッドの本来の実装では、コンピュータの VRAM に出力文字のコードを書き込むことでディスプレイの特定位置に文字を表示することになる。しかし、起動時にはレガシーカーネルも画面への出力を行っており、レガシーカーネルとマネージカーネルが独立して画面バッファを管理すると画面表示が崩れてしまう。そこで出力処理はレガシーカーネルに一元化したほうが好ましいが、移譲するためのコードは起動時に一時

テキスト 3.6　画面に文字を出力するメソッドのシグニチャ

```
1  // C#
2  static void Write(char ch);
```

的に使用されるだけであるので、マネージカーネルの本来のコードに混ぜて記述するのは可読性や性能を損なってしまう。また、CooS はクラスライブラリとして Mono クラスライブラリ[13)]を利用しているが、Mono クラスライブラリの中には Linux 向けに書かれた部分が存在するため、特定の箇所のみを CooS 向けに置き換えたいということがある。

そこで、レガシーカーネルがアセンブリをロードするときに、ロード対象アセンブリの特定箇所を置き換えるための特別な処理を行うことにした。この処理によって、CooS.Wrap 名前空間に含まれる名前の一文字目がアンダースコアであるクラスの各々のメソッドについて、その名前空間の名称から `CooS.Wrap._`を取り除いた名前空間に属しかつ先頭のアンダースコアを取り除いたクラス名に該当するクラスの、戻り値を除くシグネチャが等しいメソッドがパッチ対象として置き換えられる。

例えば、レガシーカーネルのデバッグモードの切り替えメソッドを考える。レガシーカーネルのデバッグモード設定はレガシーカーネルのソースコードで行うのが自然である。しかし、マネージカーネルに含まれるコードからその設定が出来るとデバッグに便利である。このとき、マネージカーネルに**テキスト 3.7** のようなクラスとメソッドを定義する。このメ

テキスト 3.7　パッチされるコード例

```
1  namespace MyNamespace {
2      class MyClass {
3          public void SetDebugMode(bool enabled) {
4              // NO OPERATION
5          }
6      }
7  }
```

ソッドにはレガシーカーネルが消滅した後の状態で実行されるべき処理を書く。この例ではデバッグ用なので何も処理をしていない。

次に**テキスト 3.8** のようなクラスとメソッドを定義する。このメソッドによってパッ

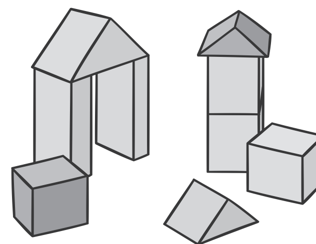
テキスト 3.8　パッチするコード例

```
1  namespace CooS.Wrap._MyNamespace {
2      class _MyClass {
3          public void SetDebugMode(bool enabled) {
4              // ここにはレガシーカーネルと通信するプログラムを書く
5          }
6      }
7  }
```

チされると、`MyNamespace.MyClass:SetDebugMode` への呼び出しがすべて `CooS.Wrap._MyNamespace._MyClass:SetDebugMode` への呼び出しに置き換えられる。

第 4 章

コンポーネント



本章と次章では、前章で説明したアーキテクチャのオペレーティングシステムをどうやって実現するか説明する。本章では特にプログラムとしての実現するためにどのようなサブシステムに分割したかを述べ、オペレーティングシステムのプログラム構造を明らかにする。

4.1　ブートストラップ・ローダ

ブートストラップ・ローダは機械語による処理が相応しい初期化処理の実行と、レガシーカーネルをロードする役割を持つ。FDD の MBR に記録されたときに動作するように作られており、El Torito† によって CD-ROM に対応している。

† 2.1.7 項参照

ブートストラップ・ローダは次のような 3 つのモジュールから成り立つ。なお、括弧内は記述言語を示す。

1. ファースト・ローダ（アセンブリ言語 + C 言語）
2. セカンド・ローダ（C 言語）
3. サード・ローダ（アセンブリ言語）

ファースト・ローダは MBR として第 1 セクタに記録される。セカンド・ローダは起動 FD 媒体の第 3 セクタから 7 セクタ分の領域に記録される。サード・ローダは起動 FD 媒体の第 2 セクタから 1 セクタ分の領域に記録される。（**図 4.1**）このような配置にしたのは、サー

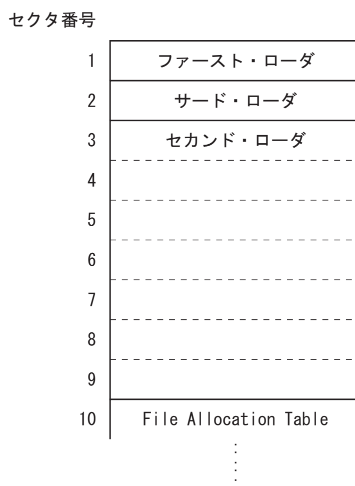


図 4.1　FD 媒体のデータレイアウト

ド・ローダのサイズが小さく固定であるのに対して、セカンド・ローダのサイズが変わりやすいからである。

ファースト・ローダが起動されてからレガシーカーネルに制御が移るまでのシーケンス図を**図 4.2** に示す。

ファースト・ローダは冒頭にアセンブリ言語によるプログラムがあり、C 言語プログラムが直後に結合された構造になっている。アセンブリ言語の部分には BPB (BIOS Parameter Block) と、C 言語プログラムにジャンプするための数個の命令がある。BPB は FAT12[11] フォーマットの FD 媒体に必要なものである。C 言語プログラムは起動 FD 媒体からセカンド・ローダを 00001200h に、サード・ローダを 00001000h に BIOS を利用して読み込み†、セカンド・ローダを呼び出す。

† 実際には、第 2 セクタから 8 セクタ分を 00001000h に一括して読み込んでしまう。

セカンド・ローダは BIOS を利用して起動 FD 媒体にアクセスし、FAT12 構造を解釈してルートディレクトリに配置されているレガシーカーネルのイメージファイルを 00101000h に読み込む。また、コンピュータに搭載されたメモリのサイズを取得し、カーネルのためにある特定のメモリ位置に書き込む。さらに、グラフィックモードで起動しようとする場合には、VBE[25] を利用してモード情報（解像度や色深度など）を取得し適切なグラフィックス

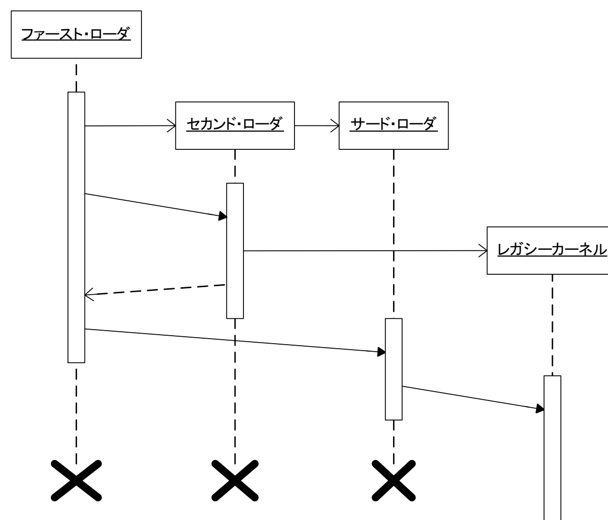


図 4.2　ブートストラップ・ローダの実行シーケンス

モードの切り替えを行う。このとき、カーネルのために、モード情報はある特定のメモリ位置に書き込んでおく。以上の処理を終えると、サード・ローダに制御が移る。

サード・ローダは CPU のモードをプロテクトモードに切り替えようとするが、そのためにまず A20 線と呼ばれるアドレスラインを有効にする。A20 線とはアドレスバスの 21 本目の信号線を指す†が、歴史的な理由により PC/AT 互換機では起動時に A20 線が無効化されているため、そのままプロテクトモードに切り替えても第 21 ビットが 1 になるようなアドレスにアクセスできないからである。(この無効化は CPU モードに関係ないため、無効化されたまま 32 ビットモードに CPU を切り替えても A20 線は有効にならない。)

† アドレス線の第 1 本目が#0 であるから A20 線になる。

A20 線を有効にしたのち、CPU のモードをリアルモードからプロテクトモードに切り替え、`00101000h` に配置されているレガシーカーネルに制御を引き渡す。

4.2　レガシーカーネル

レガシーカーネルはマネージカーネルを動作させて、マネージカーネルが自律動作するまでカーネルとして動作するプログラムである。レガシーカーネルは次のようなコンポーネントを持つ。

- エントリポイント
- メモリ管理機構
- 割り込み管理機構
- コンソール管理機構
- デバイスドライバ
  - PIC (Programmable Interrupt Controller)
  - KBC (Keyboard Controller)
  - PS/2 キーボード
  - DMAC (DMA Controller)
  - PIT (Programmable Interrupt Timer)
  - FDC/FDD (Floppy-disk Controller/Drive)
  - ATAC (ATA Controller)

  - 光学ドライブ
- ファイルシステムドライバ
  - FAT12
  - ISO9660[19)][*1]
- リフレクション
  - PE 形式[*2]パーサ
  - メタデータパーサ
  - シグネチャリーダ
  - リフレクタ
- CIL インタープリタ
  - 仮想マシン実装
  - 評価スタック実装
  - CIL メモリ管理
  - CIL 命令実装

レガシーカーネルの起動手順は本研究独自のものであるため 6 章で説明している。

4.3 マネージカーネル

マネージカーネルは**表 4.1** に示す 4 つのアセンブリから成り立っている。マネージカーネルの実行手順は本研究独自のものであるため 6 章で説明している。

表 4.1　マネージカーネルを構成するアセンブリ

アセンブリ名	記述言語	初期化時	本動作時
cscorlib	C#	必要	必要
cskorlib	Managed C++	必要	必要
csgraphics	C#	必要	必要
csbridge	Managed C++	必要	不要

cscorlib アセンブリ

cscorlib はマネージカーネルの機能の大部分を実装する最も重要なアセンブリである。

- レガシーカーネル補助ルーチン
- メモリ管理機構
- エントリポイント
- アーキテクチャ抽象レイヤ
  - IA-32 アセンブラ
  - CIL アセンブラ実装
  - 割り込みマネージャ

[*1] CD-ROM のファイルシステムの規格。

[*2] Portable Executable Format. 一般的には Microsoft Windows の.EXE や.DLL の形式であり、それを拡張したものが CLI のファイルフォーマットとして用いられている。

  - I/O ポートクラス
- コードモデル
  - PE 形式パーサ
  - メタデータパーサ
  - シグネチャリーダ
  - リフレクション実装
  - アセンブリローダ
  - 評価スタック
  - コードベリファイア
  - 抽象 CIL アセンブラ
  - コンパイラ
- 基本コレクション
- デバイスドライバ
  - PIC (Programmable Interrupt Controller)
  - KBC (Keyboard Controller)
  - PS/2 キーボード
  - PS/2 マウス
  - DMAC (DMA Controller)
  - PIT (Programmable Interrupt Timer)
  - FDC/FDD (Floppy-disk Controller/Drive)
  - ATAC (ATA Controller)
  - 光学ドライブ
  - VGA グラフィックアダプタ
- ファイルシステムドライバ
  - FAT12
  - ISO9660[19)][*3]
- マネジメント
  - コードマネージャ
  - アセンブリマネージャ
  - ファイルシステムマネージャ
- リフレクション
  - 抽象リフレクション
  - CLI リフレクション実装
- シェル抽象レイヤ
- CIL ファイルシステム実装
- パッチフラグメント

cskorlib アセンブリ

cskorlib は、特権命令など C#で表現できない処理を実行して cscorlib を補完する。

- アーキテクチャ依存コード
  - バーストメモリ操作

[*3] CD-ROM のファイルシステムの規格。

  - 浮動小数点数演算
  - 特権命令実行
- 移植プログラム
  - MT19937[10)]乱数生成ルーチン

csgraphics アセンブリ

csgraphics はグラフィックモードで必要なプログラムを提供している。

- フォントマネージャ
- グラフィックス
- グラフィックコンソールシェル

csbridge アセンブリ

† 3.6 節参照

csbridge はパッチ†を利用してマネージカーネルとレガシーカーネルが情報をやり取りする際に経由する中間層を提供する。

- カーネルブリッジ
- パッチフラグメント

4.4　FreeType

FreeType は freetype-2.1.9.tar.gz を元に、.NET ライブラリとして Visual C++ のプロジェクトを次のような手順で作成した。

ファイルの追加

表 4.2 はプロジェクトに追加する必要のあるファイル群である。追加の際に内容を改変したり、ディレクトリ構造を変更する必要はない。

表 4.2　プロジェクトに追加するファイル

autofit.c	autohint.c	bdf.c	cff.c	ftbase.c	ftcache.c
ftdebug.c	ftglyph.c	ftgzip.c	ftinit.c	ftlzw.c	ftmm.c
ftsystem.c	pcf.c	pfr.c	psaux.c	pshinter.c	psmodule.c
raster.c	sfnt.c	smooth.c	truetype.c	type1.c	type1cid.c
type42.c	winfnt.c				

コードの追加

FreeType は C 言語スタイルのメモリモデルを要求するため、そのままでは CLI のメモリアロケーションと適合しない。特に malloc, free, realloc が欠けているためにコンパイルすることができない。そこで、CLI と適合するように作られたそれら 3 つの関数を実装することで、大きな変更をすることなく FreeType を動作させる。

まず、malloc 関数として**テキスト 4.1** を追加する。この関数は、CLI スタイルで byte[] を確保し、さらに GCHandle を使ってそのメモリ領域を固定し、固定した領域のメモリアド

テキスト 4.1　malloc

```
1  extern "C" extern void* malloc(size_t size) {
2      unsigned char buf __gc[] = new unsigned char __gc[size+sizeof(
             __int32)];
3      GCHandle handle = GCHandle::Alloc(buf, GCHandleType::Pinned);
4      void* p = handle.AddrOfPinnedObject().ToPointer();
5      *(void**)p = GCHandle::op_Explicit(handle).ToPointer();
6      return (void*)((char*)p+sizeof(void*));
7  }
```

レスを返している。ただし、領域に関する情報が失われてしまうと困るので、確保領域の先頭にヘッダを用意して、そこに管理用の情報を追加している。（そのために確保サイズがヘッダ分だけ増加する。）また、関数の戻り値として返すアドレスはヘッダをスキップした位置のアドレスになっている。

この malloc 関数はポインタなども使っている上に領域を固定するなどという強引な手法を用いているが、CLI プログラムとして全く正しい†。

次に free 関数の実装として**テキスト 4.2** を追加する。この関数は malloc 関数が埋め込ん

† 実際、この FreeType プログラムは Windows プラットフォームでも動作する。

テキスト 4.2　free

```
1  extern "C" extern void free(void* p) {
2      void* handle = *(void**)((char*)p-sizeof(void*));
3      GCHandle::op_Explicit(handle).Free();
4  }
```

だ管理用の情報から GCHandle を復元し、その Free メソッドを呼び出している。Free メソッドによって malloc 関数が固定したメモリ領域は自由になり、おそらく次のガベージコレクションによって回収されることになる。

さらに realloc 関数の実装のために**テキスト 4.3** を追加する。CLI には確保した領域の大

テキスト 4.3　realloc

```
1  extern "C" extern void* realloc(void* p, size_t size) {
2      void* q = malloc(size);
3      if(p!=NULL) {
4          void* ph = *(void**)((char*)p-sizeof(void*));
5          GCHandle handle = GCHandle::op_Explicit(ph);
6          unsigned char buf __gc[] = (unsigned char __gc[])handle.Target
               ;
7          if((int)size>buf->Length-sizeof(void*)) size=buf->Length-
               sizeof(void*);
8          memcpy(q, p, size);
9          free(p);
10     }
11     return q;
12 }
```

きさを動的に変更する機能がないために、この realloc の実装はもっとも単純な方針に基づいている。つまり、単に要求されたサイズの領域を新しく確保して、そこに内容をコピーし、既存の領域を解放する。

ただし、既存の領域のサイズは malloc 関数が作った `byte[]` が必要であるため、free 関数と同様の手順で GCHandle を復元し、GCHanlde:Target から取得した `byte[]` を使って領域のサイズを得ている。

クラスの公開

以上の作業で FreeType そのもののコンパイルが可能になるので、必要に応じて機能を公開するクラスを作成すればよい。たとえば**テキスト 4.4** はフォントを描画するための簡単な機能を提供する。

テキスト 4.4　FreeType を外部に公開するためのクラス

```
namespace FreeType {

    public __delegate void FontDrawHandler(int x, int y, int level);

    public __value struct FontBitmap {
        IntPtr Buffer;
        int Width;
        int Height;
        int ScanlineSize;
    };

    public __gc class FontData {

        FT_Library freetype; // handle to library
        FT_Face  face;  // handle to face object
        FontDrawHandler* painter;

    public:

        FontData(byte fontdata __gc[]) {
            FT_Library freetype;
            int error = FT_Init_FreeType(&freetype);
            if(error) throw new FreeTypeException(error);
            this->freetype = freetype;
            FT_Face  face;
            byte __pin* pfont = &fontdata[0];
            error = FT_New_Memory_Face(this->freetype,
                pfont, fontdata->Length,
                0 /*face_index*/, &face);
            pfont = NULL;
            this->face = face;
            SetSizeByPixel(0, 16);
        }

    public:

        void SetSizeByPixel(int width, int height) {
            int error = FT_Set_Pixel_Sizes(
                face,  // handle to face object
                width,  // pixel_width
                height); // pixel_height
            if(error) throw new FreeTypeException(error);
        }

        void SetPainter(FontDrawHandler* fp) {
            this->painter = fp;
```

```
47          }
48
49      private:
50
51          void PrepareBitmap() {
52              if(face->glyph->format!=FT_GLYPH_FORMAT_BITMAP) {
53                  int error = FT_Render_Glyph(face->glyph, FT_RENDER_MODE_NORMAL);
54                  if(error) throw new FreeTypeException(error);
55              }
56          }
57
58      public:
59
60          void LoadGlyph(wchar_t ch) {
61              // load glyph image into the slot (erase previous one)
62              int error = FT_Load_Char(face, ch, FT_LOAD_DEFAULT);
63              if(error) throw new FreeTypeException(error);
64          }
65
66          __property Size get_AdvanceSize() {
67              return Size(face->glyph->advance.x, face->glyph->advance.y);
68          }
69
70          __property Size get_BitmapSize() {
71              PrepareBitmap();
72              return Size(face->glyph->bitmap.width, face->glyph->bitmap.rows);
73          }
74
75          __property Size get_BearingSize() {
76              PrepareBitmap();
77              return Size(face->glyph->bitmap_left, face->glyph->bitmap_top);
78          }
79
80          void DrawGlyph() {
81              PrepareBitmap();
82              FT_Bitmap bitmap = face->glyph->bitmap;
83              unsigned char* ppixel = bitmap.buffer;
84              for(int y=0; y<bitmap.rows; ++y) {
85                  for(int x=0; x<bitmap.width; ++x) {
86                      this->painter(x, y, *(ppixel+x));
87                  }
88                  ppixel += bitmap.pitch;
89              }
90          }
91
92      };
93
94      public __gc class FreeTypeException : public SystemException {
95      public:
96          FreeTypeException(int error) : SystemException(String::Format("FreeType returns an error: 0x{0:X}", __box(error))) {
97          }
98      };
```

```
 99
100     }
```

CooS の FreeType ライブラリもこのようにして機能を外部に公開しているが、機能自体は FreeType 依存であるし、機能の改良といったことも行っていない。また、クラスの公開時に C 言語スタイルから CLI スタイルへと API を変えることでユーザビリティの向上は見込まれるが、本論ではないので割愛する。

4.5　ビルド

本節ではこれまでに紹介したモジュールをビルド（コンパイルとリンク、および必要な仕上げ加工）するための手順を説明する。通常のプログラムであればこのような説明は不要だが、ブートストラップ・ローダなど、オペレーティングシステムには特殊な手順を経るものがあるため、詳しく説明することにする。

筆者が CooS をビルドした環境を**表 4.3** に示す。主として Windows 上でビルドを行い、

表 4.3　ビルド環境

項目	環境
プラットフォーム	Microsoft Windows Server 2003 Ent.
	Fedora Core 4
開発環境	Microsoft Visual Studio .NET 2003 Pro.
	Cygwin[16]
	binutils
	nasm[24] (Netwide Assembler)
	LSI C-86 試食版[29]
ツール類	.NET Reflector[17]

Fedora Core は Mono[13]のビルドのときのみ利用した。

4.5.1　ブートストラップ・ローダのビルド

ブートストラップ・ローダは**表 4.4** のソースファイルから成り立つが、その性質上、成果物のイメージを正確に制御してビルドしなければならない。ソースファイルのうち、アセ

表 4.4　ブートストラップ・ローダの構成

モジュール	構成ファイル
ファースト・ローダ	bpb.asm
	first.c
セカンド・ローダ	second.c
サード・ローダ	third.asm

ンブリ言語のプログラム（`.asm` ファイル）は 4.1 節で述べたメモリレイアウトを想定してハードコーディングされている。C 言語のプログラムは LSI C-86[28),29]によってスモールモ

デル†の COM 形式†へビルドされる。このとき、LSI C-86 は MS-DOS での実行を想定したファイルを生成してしまうため、次のようなパラメタを指定して実行する。

† セグメントをまたぐようなメモリ操作を行わないプログラムの規格。C 言語上で far ポインタが利用できなくなる。

† 複雑な構造を持たず、メモリへ直接配置して実行可能な形式。

```
lcc -O -o (object file) -c (source-code file)
lld -M -T <code address> -TDATA <data adress> -o (target file) (object file)
```

ただし、`(source-code file)` は C 言語ソースコードファイル、`(object file)` はソースコードをコンパイルした目的ファイル（.obj ファイル)、`(target file)` はビルドしようとする COM 形式ファイルであり、`<code address>`は実行時にコードが配置されるアドレス、`<data address>`は実行時にデータが配置されるアドレスである。

こうしてビルドされた COM 形式ファイルは `CS` レジスタが`<code address>`、`DS` レジスタが`<data address>`に設定されているときに正しく動作するようにコードが生成されるため、ファースト・ローダの C 言語プログラム部分とセカンド・ローダは、呼び出された直後に、インラインアセンブラによって `CS` レジスタと `DS` レジスタを適切な値に初期化する。

`first.c` および `second.c` は**表 4.5** に示すパラメタを用いてビルドされる。これらのパラメタの値は、純粋にコンパイル結果を観察して設定したものである。

表 4.5　C 言語ソースコードのリンクパラメタ

	code address	data address
first.c	7C60h	7DC0h
second.c	1200h	1C00h

最後に、完成した 3 つのモジュールを図 4.1 に示したようなレイアウトになるように結合し、最終的な CD-ROM 媒体イメージを作成するときに FD 媒体の先頭イメージとして利用する。

4.5.2　レガシーカーネルのビルド

レガシーカーネルはブートストラップ・ローダによって読み込まれるプログラムであり、その実行開始に当たって多様なサポートは受けられない。そのため、ブートストラップ・ローダが単純な処理でレガシーカーネルに制御を渡せるようにすることが重要である。しかし、Visual C++ の出力である DLL 形式はロードと配置に関する多様な要求を満たせる一方、配置時にそれなりの処理が必要になってしまい、ブートストラップ・ローダが処理するには困難である。

そこで、DLL を「そのままメモリにコピーすれば動作する状態」に変換すると都合がよい†。データサイズの増加や再配置できなくなるなど柔軟性は失われるが、ブートストラップ・ローダがするべきことは読み込んだファイルを “適切なアドレス” に配置して制御を引き渡すだけになるという利点がある。

† そのような、メモリ上に展開されたときの様子をそのまま保存したような形式を生 (raw) データとか生イメージという。

そこで、DLL ファイルのメモリ展開されたイメージを得ることができる strip というツールを次のように使うことにした。

```
strip -O binary kernel.dll -o kernel.img
```

こうして得た kernel.img はどこにでも配置できるわけではない。筆者が調べたところ、特に何も指定しない場合は DLL ファイルのベースアドレスが 0x11000000 番地になっており、

そのため kernel.img はそのアドレス以外にはロードできなくなっている。

ベースアドレスの指定

ベースアドレスはリンカのパラメタで指定できるため、0x00100000 番地にロードするように指定し、さらにマップファイル (kernel.map) を作成するようにして再ビルドした。結果のマップファイルの先頭部分を次に示す。

```
Address          Publics by Value             Rva+Base      Lib:Object
0000:00000000    ___safe_se_handler_count     00000000      <absolute>
0000:00000000    ___safe_se_handler_table     00000000      <absolute>
0001:00000000    _entrypoint@12               00101000 f    _entrypoint.obj
```

この中で Lib:Object が<absolute>ではない最初の行の Rva+Base に注目する。この例では

```
0001:00000000    _entrypoint@12               00101000 f    _entrypoint.obj
```

という行について、_entrypoint.obj （ _entrypoint.cpp の目的ファイル）で定義された _entrypoint という関数が DLL ファイルの一番先頭にあり†、00101000h に配置されるように DLL ファイルがリンクされたことを示している。

† リンカに渡される目的ファイルの一番はじめに指定されたものが先頭に配置されるので、ファイル名の頭にアンダースコアを付加している。

strip した kernel.img のファイルの最初のバイトはこの _entrypoint という関数になっているため、kernel.img は 0x00101000 にロードされたときに正しく動作する。よって、この値を元にブートストラップ・ローダがレガシーカーネルを配置するアドレスを調整することになる。

4.5.3　マネージカーネルのビルド

マネージカーネルの各アセンブリについて**表 4.6** のような開発ツールとオプションでビルドする。

表 4.6　マネージカーネルを構成するアセンブリのビルド

アセンブリ名	開発ツール	オプション
cscorlib	Visual C#	特になし
cskorlib	Visual C++	デフォルトライブラリを無視
csgraphics	Visual C#	特になし
csbridge	Visual C++	特になし

4.5.4　mscorlib.dll のビルド

mscorlib.dll は Mono 1.1.5.2 のコンパイル結果として得ることができる。（具体的な手順は Mono プロジェクトの説明を参照のこと。）しかし、配布されているものをそのままコンパイルすると、一部の型が隠蔽されているので開発に利用することができない。そこで、次のような手順に従ってクラスのアクセス修飾子を public へ改変した。

1. コンパイルして mscorlib.dll を生成する。

2. 出来た mscorilb.dll を参照して cscorlib.dll などをビルドする。このとき、mscorlib.dll の型が隠蔽されている（internal になっている）ために外部から参照できないというエラーが起こる。
3. エラーの原因となったクラスのアクセス修飾子を internal から public へと変更する。
4. ビルドエラーが起こらなくなるまで 1〜3 を繰り返す。

こうして出来た mscorlib.dll を参照することで Mono 独自のデータ型などを外部でも使うことができるようになる。

なお、この改変は cscorlib.dll などのビルドのために必要な作業であり、実行時には Mono オリジナルのものに差し替えても動作する。なぜならレガシーカーネルのランタイムはクラスのアクセス修飾子を無視してアセンブリをリンクするからである。

4.5.5 CD-ROM イメージの生成

普通、CD-ROM イメージの作成には必要なファイルを準備すればいいだけだが、CooS では El Torito† を用いるため、ブートストラップ・ローダを格納したフロッピーディスクのイメージを準備する必要がある。

† 2.1.7 項参照

bootldr.img のビルド

bootldr.img は FAT12 でフォーマットされた 1.44MB のフロッピーディスクイメージである。ブートセクタおよびそれに続く計 9 セクタにブートストラップ・ローダが格納されており、FAT12 のファイルシステムデータ続いたあと、さらにルートディレクトリに kernel.img が記録されている。kernel.img は kernel.dll（レガシーカーネルのコンパイル結果）を strip プログラムで処理した結果である。

CD の作成

CD-ROM イメージの作成には mksisofs を使い、ブートイメージとして bootldr.img を指定し、El Torito のフロッピーエミュレーションを有効にする。イメージ作成は、

```
mkisofs.exe -verbose -iso-level 3 -eltorito-boot bootldr.img \
    -o coos.iso <ImageDir>
```

として作成した。ただし、紙面の都合で二行にわたって分かち書きをしてあり、`<ImageDir>` は CD-ROM イメージに含まれるファイルセットが展開されたディレクトリである。

第 5 章

実装

本章では、前章で説明したオペレーティングシステムのコンポーネントが持つ各機能について詳細に説明し、オペレーティングシステムをどのように開発したか明らかにする。

5.1　CIL インタープリタ

インタープリタは仮想マシンの上でメソッドを実行する機能を持つ。

データ構造

インタープリタは仮想マシンを抽象化するが、そのためにまず**表 5.1** に示す汎用スタック型を定義する。`Stack<T>`は T を格納するスタックを提供する。このとき、横線で区切られ

表 5.1　Stack

名前	型	説明
top	`T`	末尾の要素を取得する。
push	`void`	末尾に要素を追加する。
pop	`void`	末尾の要素を削除する。

た上部はフィールドまたはプロパティであり、下部はメソッドである。（以降も同様である。）

次にメソッドを定義する。メソッドはひとつの関数を表し、**表 5.2** のようなデータを持つ。codeptr はこのメソッドの命令列を格納してあるメモリ領域を指すポインタである。このメ

表 5.2　Method

名前	型	説明
codeptr	`void*`	コードを指すポインタ
codesize	`int`	コードのバイトサイズ
codetype	`(enum)`	CIL, Runtime, Native のいずれか
name	`string`	メソッドの名前
getOpcode	`(enum)`	指定位置の CIL 命令

ソッドが黎明列と関連づけられていないときは `null` になる。codesize は codeptr が指す領域のサイズである。codetype は codeptr が指す領域の命令の種類を示す。codetype=CIL のメソッドはコードが CIL で構成されている。codetype=Runtime のメソッドはランタイムプログラムが実装提供するべきメソッドであり、その処理はランタイムプログラムが代わりに行う必要がある。このとき codeptr=null であり命令列は指定されない。codetype=Native のメソッドはコードが機械語である。

次にフレームを定義する。フレームとはひとつの関数を実行コンテキストを表し、**表 5.3** のようなデータを持つ。method、args および vargs は初期化時に設定されて、その後は変化しない。PC は実行するべき命令の、method.codeptr からのバイトオフセットを示す。

表 5.4 に評価スタック（Evaluation Stack: CLI のスタック）を表すデータ構造を示す。評価スタックはここでは単純なバッファであるが、実際の実装ではスタックとして操作するための機能セットを持っている。

最後に仮想マシンを表すデータ構造を Machine として定義する（**表 5.5**）。

表 5.3　Frame

名前	型	説明
method	`Method`	フレームが表すメソッド
args	`void*`	引数が置かれたスタック中の位置
vars	`void*`	局所変数が置かれたスタック中の位置
pc	`int`	プログラムカウンタ
	(none)	

表 5.4　EvalStack

名前	型	説明
data	`byte[]`	スタックのデータ領域
	(none)	

表 5.5　Machine

名前	型	説明
stack	`EvalStack`	マシンのスタック
framelist	`Stack<Frame>`	フレームのリスト
	(none)	

プログラム構造

インタープリタは**図 5.1** に示すようなループを基本としている。このループは関数単位の

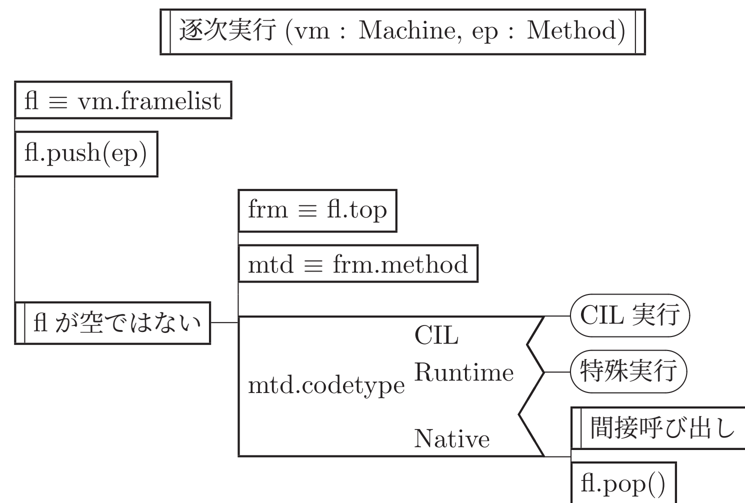


図 5.1　インタープリタの主処理

ループであるため、ループが処理する単位はフレームである。

CIL メソッドを処理するルーチンを**図 5.2** に示す。ただし、*Disp* は個別に命令を処理するためのサブルーチンを呼び出すための関数である。この関数は命令が呼び出しを行う場合

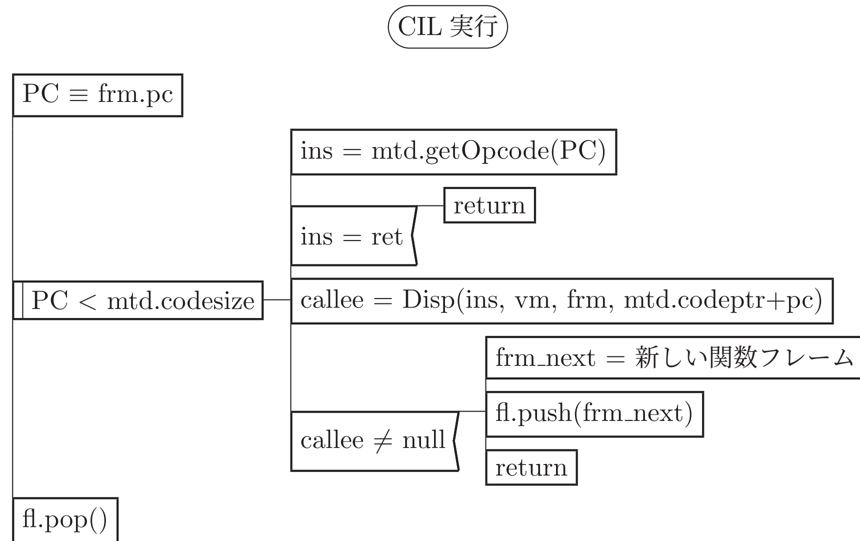


図 5.2　CIL 実行ルーチン

に戻り値として呼び出すべきメソッドを返戻し、呼び出しを行わない（そのまま順次実行する）場合は null を返戻する。

特殊実行はデリゲートのコンストラクタと呼び出しメソッドに対して行われる。特殊実行を行うルーチンを**図 5.3** に示す。

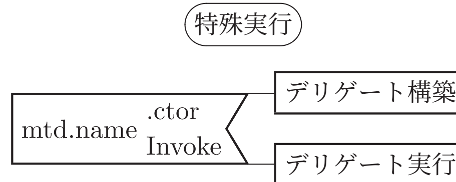


図 5.3　特殊実行ルーチン

間接呼び出しを行うルーチンを**図 5.4** に示す。

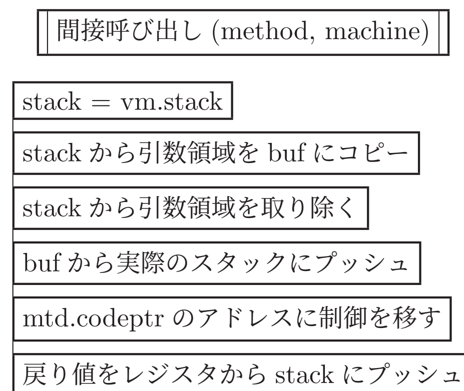


図 5.4　間接呼び出し実行ルーチン

5.1.1　CIL 命令実装の例

CIL 命令の実行ルーチンの例として、add 命令を実行するものを**テキスト 5.1** に示す。ス

テキスト 5.1　add 命令の実行ルーチン

```
1  extern Method* execute_add(ILMachine& machine, Frame& frame,
       const byte* operand) {
2      frame.pc += 1;
3      ILStack& stack = machine.stack;
4      switch(stack.getDataType()) {
5      default: warning("{AOP:UNK}");
6      case STACK_TYPE_N: {
7      case STACK_TYPE_I4: {
8          __int32 v2=(__int32)stack.popi(); __int32 v1=(__int32)stack.popi();
9          stack.pushi(v1 + v2);
10         } break;
11     case STACK_TYPE_I8: {
12         __int64 v2=(__int64)stack.popl(); __int64 v1=(__int64)stack.popl();
13         stack.pushl(v1 + v2);
14         } break;
15     case STACK_TYPE_R4: {
16         float v2=stack.popr4(); float v1=stack.popr4();
17         stack.pushr4(v1 + v2);
18         } break;
19     case STACK_TYPE_R8: {
20         double v2=stack.popr8(); double v1=stack.popr8();
21         stack.pushr8(v1 + v2);
22         } break;
23     case STACK_TYPE_O:
24     case STACK_TYPE_P: {
25         nint v2=stack.popn(); nint v1=stack.popn();
26         stack.pushp((void*)(v1 + v2));
27         } break;
28     }
29     return NULL;
30 }
```

タックから適切な型で値を取り出し、加算を行った後、結果をスタックに積んでいる。add 命令は呼び出しを行わないので戻り値を NULL にする。

5.2　メモリ管理機構

ブロック（3.4 項参照）の先頭はヘッダと呼ばれる。ヘッダは**テキスト 5.2** のような構造を持つ。ここで、2 つのフィールドの型が uint であるのはそのサイズが IA-32 のポインタサ

テキスト 5.2　メモリブロックのヘッダ

```
1  // C#
2  [StructLayout(LayoutKind.Sequential)]
3  struct Header {
4      public uint Size;
5      public uint Type;
6  };
```

イズと等しいからである。一般に CLI ではそのようなときは IntPtr を使用するが、IntPtr を用いると値へのアクセスにメソッドを経由するなど好ましくないので端的に uint を用いて

いる。Type フィールドの内容は型オブジェクトであり本来その値は参照であるが、メンバに参照型を含む構造体は操作が制限されてしまうのでこのようにしている。†

† これによって GC 管理外となるため、内容が指す先が GC によって移動されないようにする処理が将来的には必要になると思われる。

メモリブロックを確保するためのプログラムを**テキスト 5.3** に示す。ただし、引数 `size` は確保しようとする利用可能な（管理区域を含まない）領域サイズであり、`Current` は現在の空き領域の先頭アドレスを格納する変数である。2 行目によって、サイズは 4 の倍数へ

テキスト 5.3　メモリブロック確保ルーチン

```
1   private static unsafe Header* AllocateBlock(int size) {
2       size = (size+3)&~3;
3       Header* p0;
4       while((p0 = (Header*)Current)->Size!=0) {
5           Current = Current+p0->Size;
6       }
7       Header* p1 = (Header*)((byte*)(p0+1)+size);
8       p1->Size = 0;
9       p1->Type = 0;
10      p0->Type = 0;
11      p0->Size = (nint)(size+sizeof(Header));
12      Memory.Clear(p0+1, size);
13      return p0;
14  }
```

と調整される。これは IA-32 プロセッサのメモリアクセスのルール†に従ったものである。4〜6 行目は空きブロックの先頭を探すための処理である。テキスト 5.3 はマネージカーネルのものであるが、レガシーカーネルのインタープリタによって空き領域が消費されていると `Current` が指す領域もすでに消費されている可能性があるため、この処理によって未使用領域まで辿っていく。7〜9 行目は現在の空き領域から必要なサイズだけ後ろの位置（新しい空き領域の先頭）に、先頭であることを示すブロックを作成している。10〜12 行目で、確保したブロックを初期化している。

† たとえば 4 バイトのアクセスなら、アクセスする番地が 4 の倍数のときにもっとも処理時間が短くなる。このような規則は一般的にデータの alignment と呼ばれる。

5.3　基本コレクション

基本コレクションクラスとは特殊化された 2 つの Hashtable[*1]である。（説明のために Hashtable を挙げたが、クラス階層的な関係はなく、表面的に同等という意味である。）それぞれ、キーを MethodInfo[*2]と Int32 に限定している。

これらは次のようなランタイムの動作にとって必要になる情報を格納する。

- ロード済みアセンブリのテーブル。キーが数値 ID で、値がアセンブリオブジェクト。
- 生成済み機械語コードのテーブル。キーが MethodInfo で、値が CodeInfo[*3]。
- ハンドル†に対応するオブジェクトのテーブル。キーがハンドルの値で、値がオブジェクト。

† 5.4.1 項参照

このようなデータの格納と操作は標準の Hashtable でも提供されているが、標準の実装では複雑な処理を含むため動作の予測が困難になってしまう。例えば、単に Int32 をキーとした場合でも、IComparable インターフェイスを経由するため仮想関数呼び出しが関わってく

[*1] System.Collections.Hashtable
[*2] System.Reflection.MethodInfo
[*3] コード情報を格納するクラス。

るなどして好ましくない。基本コレクションクラスではキーの型を限定することで、たとえば Int32 なら単に数値比較だけでキー比較をすませており、数値比較は単純な機械語に置き換えることができる。

5.4 リフレクション

リフレクションの実装にあたって特に重要な要素は型に関する `Type` クラス[*4]、フィールドに関する `FieldInfo` クラス[*5]、メソッドに関する `MethodInfo` クラス[*6]である。これらのクラスは利便性のために多くの機能を公開しているがランタイム依存部分の実装は行っていないため、CLI 実装系は環境に合わせた実装を提供する必要がある。そこで、本研究では**図 5.5** に示すようなこれらのクラスの派生クラスを用意して適切な実装を提供している。図

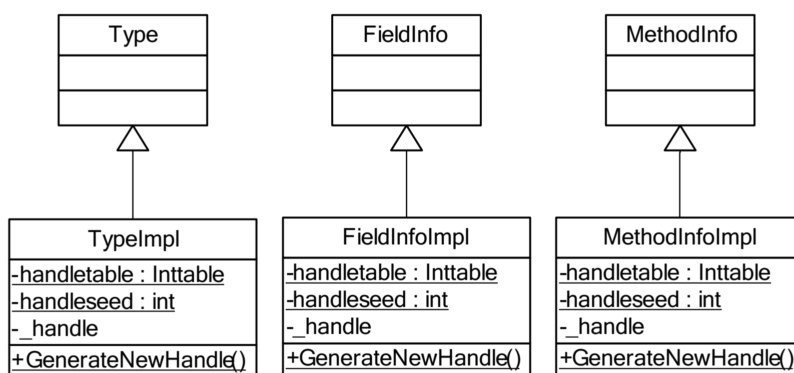


図 5.5　基本的なリフレクションクラス

で示したように、TypeImpl、FieldInfoImpl、MethodInfoImpl がそれぞれ親クラスの実装を提供する。ただし、図において各クラスのフィールドとメソッドに関しては後の説明で重要になるものしか示していない。

前述の 3 つのクラスが実装する機能には、ランタイムにとっては実行に必要な特別なデータに関するものがある。実行に必要なデータとは

- ハンドル
- スロット
- インターフェイスマップ

である。

5.4.1 ハンドル

ハンドルには 3 種類あり、それぞれ `RuntimeTypeHandle`[*7]、`RuntimeFieldHandle`[*8]、`RuntimeMethodHandle`[*9]である。`RuntimeTypeHandle` は型 (Type) と、`RuntimeFieldHandle` はフィールド (FieldInfo) と、`RuntimeMethodHandle` はメソッド (MethodInfo) と関連づ

[*4] System.Type
[*5] System.Reflection.FieldInfo
[*6] System.Reflection.MethodInfo
[*7] System.RuntimeTypeHandle
[*8] System.RuntimeFieldHandle
[*9] System.RuntimeMethodHandle

けられている。

ハンドルはある特定のインスタンスと結びつけられているが、その保持する内容の数値は変化しない。そのため、あるオブジェクトを参照以外の形式で表したいときはハンドルを用いることになる。

ここで代表例として `RuntimeTypeHandle` について説明する。`RuntimeTypeHandle` は主に Type のメソッドを通じて操作されるため、CooS はハンドルを管理するためのメソッドを TypeImpl に定義している。中心的なコードを**テキスト 5.4** に示す。TypeImpl:handletable

テキスト 5.4　RuntimeTypeHandle に関するメンバ

```
1  public abstract class TypeImpl : Type {
2      // Fields
3      static Inttable handletable = new Inttable();
4      static int handleseed = 1;
5      private RuntimeTypeHandle _handle;
6      // Methods
7      private static unsafe RuntimeTypeHandle GenerateNewHandle(
           TypeImpl type) {
8          int handle;
9          for(;;) {
10             handle = handleseed++;
11             if(!handletable.ContainsKey(handle)) {
12                 break;
13             }
14         }
15         handletable[handle] = type;
16         return *(RuntimeTypeHandle*)&handle;
17     }
18 }
```

はハンドルの値をキーとして Type オブジェクトを格納し、両者を結びつけるために用いられる。このコレクションは実行の基礎部分で使用されるため、外部への依存をなくすために標準の Hashtable ではなく基本コレクション†の Inttable を用いる。

† 5.3 節参照

ハンドルにアクセスするためのコードを**テキスト 5.5** に示す。コードより、ハンドルは

テキスト 5.5　RuntimeTypeHandle アクセッサ

```
1  public override sealed RuntimeTypeHandle TypeHandle {
2      get {
3          if(this._handle.Value==IntPtr.Zero) {
4              this._handle = GenerateNewHandle(this);
5          }
6          return this._handle;
7      }
8  }
```

オンデマンドで生成される†ことが分かる。`sealed` 修飾子を指定することによって、コンパイラが仮想関数呼び出しではなく通常の関数呼び出しのコードを生成させることができるため、仮想関数呼び出しに関わる複雑な手続きを省略することができるようになる。

† CooS 内部ではハンドルの値をそれほど重要な値として扱っていない。

5.4.2 スロット

スロット[†]は仮想関数の実行に必要であるため、それを仮想関数を用いて得ることはできない。そのため、スロットの構築手続きは抽象メソッドによって実装されるが、データはTypeImpl のフィールドに直接保持される（**テキスト 5.6**）。

† 2.4.3 項参照

テキスト 5.6 スロットに関するメンバ

```
1  public abstract class TypeImpl : Type {
2      // Fields
3      private MethodInfoImpl[] slots;
4      // Abstract methods
5      public abstract MethodInfoImpl[] ConstructSlots(
6          out InterfaceBase[] ifbases);
7      // Methods
8      public void PrepareSlots() {
9          if(this.slots!=null) return;
10         this.slots = this.ConstructSlots(out this.ifbases);
11     }
12     public MethodInfoImpl GetSlotMethod(int slotIndex) {
13         if(this.slots==null) this.PrepareSlots();
14         return this.slots[slotIndex];
15     }
16 }
```

`TypeImpl:slots` は表 2.9 の定義列を格納した配列である。ある仮想メソッドまたは抽象メソッドがこの配列中のどこに位置するかは静的に決定できるため、実行時に正しいメソッドを簡単に取得できる。

5.4.3 インターフェイスマップ

インターフェイスマップ（2.4.4 項参照）もスロットと同様の理由でデータを `TypeImpl` に直接保持することにした（**テキスト 5.7**）。

テキスト 5.7 インターフェイスマップに関するメンバ

```
1  public abstract class TypeImpl : Type {
2      // Fields
3      private InterfaceBase[] ifbases;
4      // Methods
5      public int GetInterfaceBaseIndex(RuntimeTypeHandle handle) {
6          this.PrepareSlots();
7          for(int i=0; i<this.ifbases.Length; ++i) {
8              if(this.ifbases[i].Handle.Value==handle.Value) {
9                  return this.ifbases[i].BaseIndex;
10             }
11         }
12         throw new NotImplementedException();
13     }
14 }
```

これによって、インターフェイスのメソッドを高速に呼び出すことができる。

5.5　コードベリファイア

コードベリファイアはコードを走査し、整合性を調べたりコンパイルに必要な情報を補完する。ここで、以降の説明に用いる C#のサンプルコードを**テキスト 5.8** に示し、それをコンパイルした結果の CIL コードを**テキスト 5.9** に示す。

テキスト 5.8　コードフローの C#サンプル

```
1  public static void Main() {
2      for(int i=0; i<10; ++i) {
3          Console.WriteLine("Hello!");
4      }
5  }
```

テキスト 5.9　コードフローの CIL サンプル

```
1   L_0000: ldc.i4.0
2   L_0001: stloc.0
3   L_0002: br.s L_0012
4   L_0004: ldstr "Hello!"
5   L_0009: call void [mscorlib]System.Console::WriteLine(string)
6   L_000e: ldloc.0
7   L_000f: ldc.i4.1
8   L_0010: add
9   L_0011: stloc.0
10  L_0012: ldloc.0
11  L_0013: ldc.i4.s 10
12  L_0015: blt.s L_0004
13  L_0017: ret
```

L_0004〜L_0015 までコードベリファイアはまずコードフローを解析する。テキスト 5.9 のスタック状態は**表 5.6** のように決定できる。この図において、たとえば L_0004 の `ldstr` 命令は文字列をロードする命令のためスタックに String を積む。L_0011 の `stloc.0` 命令はスタックから値を取り出し 0 番目の局所変数（var[0]）に格納する。そのため、結果的にスタックからひとつ値が消える。

ここで、CIL の命令のうち実行制御を行うものは

- `br`
- `br.<cond>`
- `switch`
- `throw`
- `leave`

だけであり、これら以外は単純に次の命令に制御が移っていくため、フロー制御を伴わない命令の連なり（コードブロック）のスタック状態は簡単に決定できる。表 5.6 においては、L_0000〜L_0002、L_0004〜L_0015、L_0017 がブロックになる。

表 5.6　コードフロー例のスタック状態

ラベル	命令	スタック状態
L_0000:	ldc.i4.0	
L_0001:	stloc.0	Int32
L_0002:	br.s	
L_0004:	ldstr	
L_0009:	call	String
L_000e:	ldloc.0	
L_000f:	ldc.i4.1	Int32
L_0010:	add	Int32 Int32
L_0011:	stloc.0	Int32
L_0012:	ldloc.0	
L_0013:	ldc.i4.s 10	Int32
L_0015:	blt.s	Int32 Int32
L_0017:	ret	

5.5.1　オペランドの補完

L_0010 の add 命令はスタックから 2 つの数値を取り加算結果をスタックに積む命令だが、演算対象の型はオペランドに示されないためテキスト 5.6 のようにして演算対象の型を調べる必要がある†。

† インタープリタではスタックが型情報も併せ持つことで解決していた。

5.5.2　合流点の検査

フロー制御命令によって分岐したときスタック状態の整合性が取れないと正常に実行できなくなるため、コードベリファイアが分岐前と分岐後のスタック状態を検査し、整合性を確認する。

テキスト 5.9 の例では L_0002 と L_0015 がフロー制御命令になり、これらの分岐先と状態は**表 5.7** のようになる。br.s は無条件ジャンプ命令であるためスタックを消費しない。

表 5.7　コードフローの例

ラベル	命令	分岐先ラベル	分岐元状態	分岐先状態
L_0002:	br.s	L_0012	(empty)	(empty)
L_0015:	blt.s	L_0004	Int32 Int32	(empty)

そのため、分岐元と分岐先のスタック状態は等しい必要がある。対して、blt.s は条件付きジャンプ命令（branch on less than）であるためスタックから 2 つの値を取り出して比較するから、スタックから 2 つ値を取り除いた状態が分岐先のスタック状態と等しくなければならない。

5.6　CIL コンパイラ

コンパイラの処理で特に重要なのは、生成した機械語をどのように呼び出せばよいか（呼び出し規約: calling convention）と、どのようにメモリを使用するのか（スタックフレーム構造）と、命令の翻訳処理の仕方である。

5.6.1　呼び出し規約

呼び出し規約は CLI の規格に準じた形にした。**表 5.8** に本研究の呼び出し規約である `coosdecl` と共に、一般的な規約も併記して示す。cdecl は C 言語プログラムで用いられる

表 5.8　呼び出し規約

規約名	クリーンアップ	引数渡し
cdecl	呼び出し元	スタックに右から左へ
stdcall	呼び出し先	スタックに右から左へ
fastcall	呼び出し先	左から ECX と EDX レジスタへ 残りをスタックに右から左へ
thiscall	呼び出し先	this ポインタは ECX レジスタへ それ以外はスタックに右から左へ
coosdecl	呼び出し先	スタックに左から右へ

規約である。呼び出し元でスタックを保守することと右から左へ積むという二点により、C 言語の可変個引数を可能にしている†。一方、ほとんどの関数は可変個引数ではないのに引数の後始末を行うためのコードがすべての呼び出し元に必要になってしまい、冗長なコードがかなり存在してしまう。そこで引数の後始末を呼び出し先で行うことにしたのが stdcall である。可変個引数は実現できないが、cdecl よりメモリを効率的に使うことができる。

† 呼び出し先はスタックトップに近いものを第一引数として扱えばいいし、引数がいくつ積まれているかは気にしなくてよいため、引数の個数には無関心になれる。

fastcall は cdecl および stdcall より高速な呼び出しのために用意された。左から 2 つの引数（ただしレジスタサイズ以下のもの）についてレジスタを経由して渡すことによって、スタック操作にかかる処理を省いている。引数が右から左へ評価されるとき最新の引数はレジスタに格納されている確率が高いから、コンパイラによって最後の引数が ECX と EDX に格納するように最適化されれば、fastcall によって性能が改善されうる。

thiscall は C++ のメンバ関数呼び出しで用いられる規約である。メンバ関数には this に関わる引数は明示的に宣言されないが、その this ポインタ†を ECX レジスタに格納して渡すものである。

† インスタンスを指すポインタ。

coosdecl はほかの規約と違い、引数を左から右へ積んでいく。これは CIL の評価スタックに準じた形である。C 言語のような可変個引数は使用できないが、CLI では最後の引数を `Object[]` とすることで仮想的に可変個引数を実現できるので実用上問題にはならない†。

† 実際、C#では C 言語と同じように可変個引数を受け取れる。

5.6.2　スタックフレーム

コンパイラは**図 5.6** のようなスタックフレームを用いる。N はこのフレームのメソッドの引数の数であり、arg[N-1] は最後の（一般には右端の）引数である。EIP は当該メソッドの処理が終わったときの戻りアドレスである。Previous EBP は EBP の古い値であり、新しい値は Previous EBP が格納されているアドレスになる。Local Workspace はコンパイ

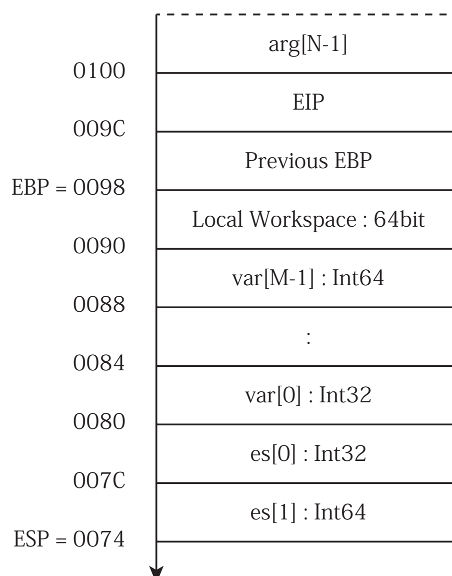


図 5.6　コンパイラが生成する機械語のスタックフレーム

ラが必要に応じて使うための予約領域である†。M はこのメソッドの局所変数の個数であり、var[M-1] は Int64 型の最後の局所変数である。以降 var[0] まで局所変数領域が続き、そのあとは評価スタックのための領域になる。図では、es[0] として Int32 型の値と es[1] として Int64 型の値が積まれている。このとき es[1] がスタックトップに当たる。

† 6.5 節参照

Local Workspace から var[0] までの領域は、スタックフレームとしてメソッドのプロローグコード†で一気に確保される。この領域はデータの記憶域として利用されるため、その内部のレイアウトがスタックの形に縛られることはない。対して、その後のメソッド本体の実行で使用される評価スタック領域は、まさにスタックとして操作される。

† メソッドの開始直後に実行される、コンパイラによって自動的に生成されるようなコードのこと。

IA-32 プロセッサにはスタック操作のための命令が存在するが、その最大の単位はレジスタサイズと等しい 32 ビットである。しかし、図中の Int64 の値のように、CLI ではスタックへの型として 64 ビット整数と 64 ビット浮動小数点数を定義しており、その部分では 64 ビット単位でスタックに置かれることになる。

逆に、32 ビット未満の値が評価スタックに置かれることはない。CIL で扱う、たとえば Boolean や Int16 のような 32 ビット未満の型は、Int32 に符号付き拡張されたうえでスタックに置かれることになり、命令で必要になる型情報はオペランドかコードフロー解析によって、いずれにせよ静的に決定されることになる†。

† 静的に決定できるため、ある箇所のたとえば add 命令が、符号付きと符号なしの両方の数値に対して実行されるという状況は起こらない。

5.6.3　基本命令の翻訳

CIL はスタックマシンを想定するため、IA-32 のレジスタマシンとは本質的な動作形態は異なる。しかし、（速度効率を無視すれば）IA-32 の命令体系にあるスタック操作のための命令を使用することによって、比較的簡単に変換を行うことができる。とくに CLI において “Basic Instructions” と分類されている命令群にこの傾向が強い。

たとえば、**テキスト 5.10** は「スタックに 3 を置く」という動作する。この命令は**テキスト 5.11** と等価であるといえる。この機械語はさらに適切な一語の命令で実現する事もできるが、本研究はコンパイラの最適化を目的にしてはいないため考えないことにする。

もう少し複雑な例として加算命令を取り上げる。例を**テキスト 5.12** に示す。ただし、id.i4

テキスト 5.10　CIL の ld.i4 命令例

```
1  // CIL
2      ld.i4 3
```

テキスト 5.11　ld.i4 と等価な IA-32 命令

```
1  // x86 Assembler
2      mov eax, 3
3      push eax
```

命令は説明のために記述したものである。例のコードにおいて add 命令は「スタックから 3 と 4 を取り出して加算し、結果の 7 をスタックにプッシュ」する。テキスト 5.12 は**テキス**

テキスト 5.12　CIL の add 命令例

```
1  // CIL
2      ld.i4 3
3      ld.i4 4
4      add
```

ト 5.13 の IA-32 機械語と等価であるといえる。

テキスト 5.13　add と等価な IA-32 命令

```
1  // x86 Assembler
2      push 3
3      push 4
4      pop ecx
5      pop eax
6      add eax, ecx
7      push eax
```

総じて、5.6.4 項で述べるような命令以外の、たとえば算術演算命令などは、上記のような単純な変換で実現できるものが多い。

5.6.4　オブジェクト指向命令の翻訳

CIL はオブジェクト指向であるため、IA-32 機械語への単純な変換では実現不可能な命令がいくつか存在する。とくに CLI において “Object Model Instructions” として分類されている命令群にそのような傾向を持つものが多い。

オブジェクト指向命令は、実行に必要な情報がコンパイル時に静的に決定するために処理が簡単なもの（静的命令）と、静的に決定できないもの（動的命令）に分類できる。

静的命令としては「オブジェクト要素操作」の命令が挙げられる。例えば ldfld 命令はインスタンスフィールドをスタックに積む命令であるが、フィールドのオフセットはメタデータに基づき事前決定されるため、命令の実行に動的な要素は含まれない。そのため、ldfld 命令が Int32 型のフィールドを読み出すとき、

; インスタンスアドレスを取り出す

```
pop eax
; オフセット加えて 32bit 整数をロードする
push dword ptr [eax+<offset>]
```

のような機械語列に変換することができる。ただし、`<offset>`はインスタンスを指すアドレスからフィールドまでのオフセットである。

動的命令にはたとえば**テキスト 5.14** の `isinst` 命令が挙げられる。`isinst` 命令は、「スタックからオブジェクト参照を取りだし、そのオブジェクトが ValueType*10かまたはその派生型であるか調べ、結果を真偽値でプッシュする」というものである。（ただし、ValueType の指定は実際には符号化されて 32 ビット整数で表現される。）この命令の動作には、スタックにあるオブジェクト参照の型情報を処理し、ValueType 型との関係を実行時に調べなくてはならない。そのような処理はリフレクションを使う必要があり、命令として実行するよりも補助ルーチンを呼び出した方が現実的である。

テキスト 5.14　isinst

```
1  // CIL
2      isinst System.ValueType
```

そこでコンパイラは isinst 命令を**テキスト 5.15** のようなメソッド呼び出しに置き換える。（次のコードは CIL であるが、結果は例示の CIL をさらに機械語にコンパイルした結果と等しい。）ただし、IsinstImpl は isinst を処理するためのメソッドであり、トークンとは対象を表す 32 ビット整数である。CIL が命令に符号化されるときに “System.ValueType” と文字列を埋め込むのは非現実的なので、トークンとして符号化されて命令のオペランドに記述される。

テキスト 5.15　isinst の実装

```
1  // CIL
2      push <System.ValueType のトークン>
3      call IsinstImpl
```

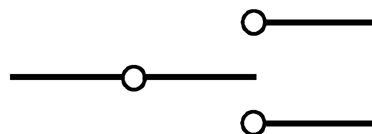
ほかの動的命令、例えば callvirt 命令†も実際に実行対象となるメソッドはコンパイル時には確定できないため、単純な呼び出し命令には翻訳できない。最終的に、これら動的命令の多くは補助ルーチン（6.5.2 項参照）を呼び出すことによってその機能を実装した。

† 仮想メソッド呼び出しを実行する命令。

*10 System.ValueType

第 6 章

提案手法によるカーネルの実行



本研究ではカーネルの実行方法に関する新しい手法を提案し、実際に適用した。その手法によってシステムのメタデータをカーネル自身で構築することができ、システム全体を CLI 準拠にすることができるようになる。本章では、その提案手法について説明する。

6.1　カーネルのブート

コンピュータが起動したあと、レガシーカーネルがメモリに配置され、通常必要とされる初期化処理を終えたとする†。このとき、256MB のメモリが搭載されているコンピュータの場合、物理メモリは**図 6.1** のように使用されている。PC/AT では先頭領域を BIOS などが

† PC/AT においては、ブートストラップローダがレガシーカーネルを読み込み、レガシーカーネルが外部記憶装置への I/O 準備を完了した状態である。

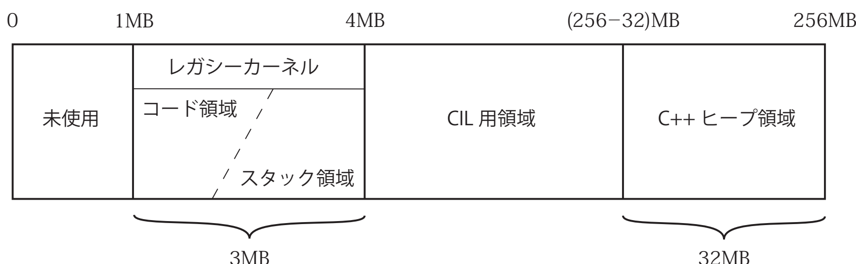


図 6.1　メモリの物理的なレイアウト

利用しているため、1MB 以下の領域は使用しない。1MB〜の領域にはレガシーカーネルのコードと一部の静的データが配置され、〜4MB の領域はスタック領域として利用される。レガシーカーネルのサイズは事前に決定できるため、コード部分を除いた残りの領域でスタックに十分な大きさとして 3MB とした。物理メモリの終端から 32MB の領域は C++ ヒープ領域†として利用される。残りの領域は CIL 用として利用される。

† malloc 関数や new 演算子で確保される領域のこと。

このレイアウトにおいてレガシーカーネルが利用する領域が前後に分断されているのは意図的なものである。レガシーカーネルが不必要になったときは C++ ヒープ領域も不要になるため、こうすることで CIL 領域をそのまま伸張すれば領域を再利用できるようになっている。

ここで、レガシーカーネルのコードおよびスタック領域と C++ ヒープ領域の両方を指して “Unmanaged 空間” と呼ぶことにする。さらに対応のために、CIL 用領域を “Managed 空間” と呼ぶことにする†。

† Managed/Unmanaged は CIL において「従来スタイル」と「CIL スタイル」を区別するためにしばしば付けられる形容詞である。

手順概要

次に、レガシーカーネルはマネージカーネルをロードし、インタープリタによる実行のためにメタデータを構築する。6.2 節で述べるものと違い、このメタデータはレガシーカーネル（のインタープリタ）が利用するものであるから、CLI とは互換性のない表現で構わない。このときの様子を**図 6.2** に示す。（なお、レガシーカーネル自体など冗長な要素は省略している。）メモリ空間には C++ 用の領域しかなく、その中にはロードしたメタデータと、レガ

図 6.2　初期状態

シーカーネルのコンポーネントである CIL インタープリタがある。

次に、レガシーカーネルは CIL インタープリタによってマネージカーネル内のアセンブリローダを動作させ、Managed 空間内にもメタデータを構築する（**図 6.3**）。この手順においてアセンブリローダという “CIL プログラム” によって処理を行うには特別な手法を用いる

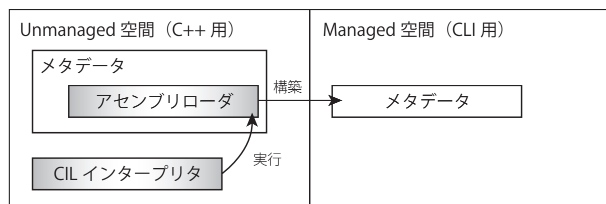


図 6.3　メタデータ構築手順

必要があるため、6.2 節で詳しく説明する。

そして最後に、構築したメタデータをコンパイルすることでマネージカーネルの機械語表現を得ることができる（**図 6.4**）。実行に必要なランタイムプログラムとデータも Managed

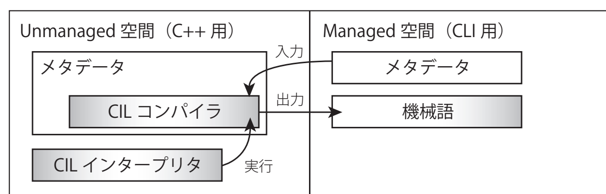


図 6.4　コンパイル手順

空間にあるため、生成した機械語はレガシーカーネルおよび Unmanaged 空間にあるデータへ依存しない。このとき、プログラムを全てコンパイルすることによって容易に機械語がインタープリタに依存しないようにすることが可能であるが、本研究では、必要な箇所のみ明示的にコンパイルし、残りは JIT コンパイルすることを試みている。コンパイル手順については 6.3 節で詳しく説明する。

6.2　メタデータの構築

単純に CLI を実装することを考えたとき、プログラムの実行制御はランタイムプログラムの役割のひとつであるから、メタデータはランタイムプログラムと共に保持する手法が自然である。このように実装した場合のメモリの様子を**図 6.5** に示す。メモリ全域は

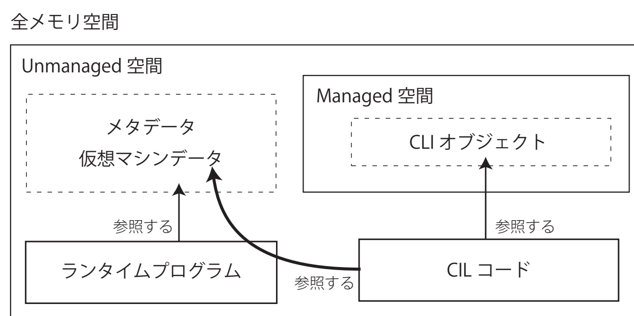


図 6.5　単純な実装でのメモリ中のデータとコード

Unmanaged 空間として使用され、Unmanaged 空間の一部が Managed 空間として利用される。カーネルを含むランタイムプログラムと、実行する CLI プログラムコードおよびメタデータは Unmanaged 空間に格納される。CLI オブジェクトは Managed 空間に格納さ

れる。

ランタイムプログラムは C 言語ファミリで書かれる（ことが多い）ので、Unmanaged 空間のデータには自然な形でアクセスできる。Managed 空間の CLI オブジェクトにアクセスするには、データ構造をプログラムで解釈しながらアクセスする必要がある。

CIL プログラムは、当然 CLI オブジェクトには自然な形でアクセスできるが、Unmanaged 空間のデータには基本的にアクセスできない。基本的にというのは、CLI では（参照ではなく C++ と同じ意味での）ポインタが存在するのでデータを読み書きすることは可能†だからである。しかし、確保や解放の機能は存在しないことや、ランタイムプログラムの存在を考慮する必要があるなど、整合性を保ったままデータ構造の操作することは難しい。また、ポインタを多用することは CLI の利点を失う原因になるため好ましくないとされており、CIL プログラムから Unmanaged 空間へのアクセスはできないと考えるべきである。

† 対照的に、JVM ではポインタがないため規格内ではできない。

CIL プログラムがロードされると、まずランタイムプログラムによって解析され、メタデータと CLI コードが図中の該当位置に配置される。その後、動作中にインスタンスを生成したりすると Managed 空間内にオブジェクトが生成される。

CIL プログラムがメタデータを参照するといったんランタイムプログラムへの呼び出しに置き換えられる。ランタイムプログラムは参照されたメタデータの値を取得し、CLI プログラムへと返戻される。（図 6.5 中央の矢印。）このようにいったんランタイムプログラムを介する理由は、前述のように、メタデータは Unmanaged 空間に存在し、その領域に自然にアクセスできるのはランタイムプログラムだからである。

ここで、このメモリモデルから Unmanaged 空間を無くすことにする。（なぜなら、Unmanaged 空間のデータ構造は CLI にとって自明ではなく、ガーベジコレクタによって管理できないため、CLI の利点を生かすことができないからである。）そのために、メタデータは CLI メモリ空間にも構築することにし、次に、CIL プログラムも CLI メモリ空間内に配置するようにした（**図 6.6**）。こうすることによって、CIL プログラムがメタデータを参照しても

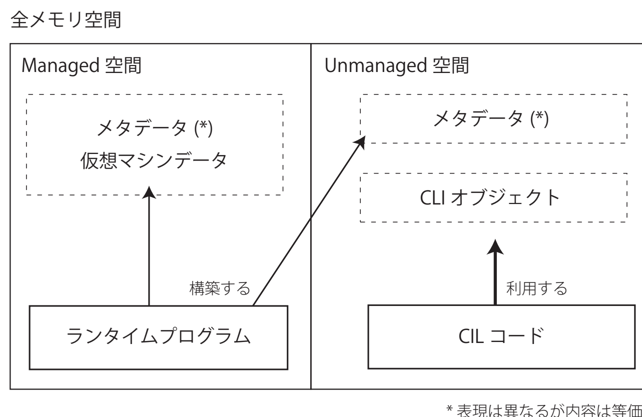


図 6.6　提案手法のメモリ中のデータとコード

ランタイムプログラムを経由することがなくなるが、しかし、メタデータの構築には依然としてランタイムが使われてしまい、やはり C++ メモリ空間をなくすことが出来なくなってしまう。また、C++ プログラムから CLI メモリ空間を操作することは困難であるという問題もある。

そこで、さらにこれを解決するために、メタデータを構築するためのアセンブリローダを用意し、CLI メモリ空間内で動作させることにした（**図 6.7**）。

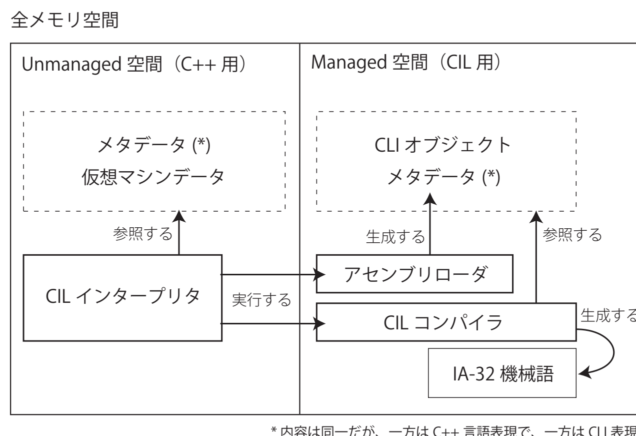


図 6.7　メモリ中のデータとコード

しかし、マネージカーネルはランタイムプログラムに依存することなく単独で動作しなければならないため、メタデータをなんらかの方法で Managed 空間内に構築する必要がある。

このとき、Managed 空間を操作する処理は CIL を使用した方が記述しやすいため、マネージカーネル内にアセンブリをロードしメタデータを構築するプログラムとしてアセンブリローダを用意した。しかしここで、次のような循環する問題が起こってしまう。

- CIL プログラムの動作にはメタデータが必要である。
- メタデータは（CIL プログラムである）アセンブリローダの出力として得られる。

そこで、起動時に必要なアセンブリについて次のようなロード手順を踏むことで上記の問題を回避した。

手順 1: 仮メタクラスの作成

メタクラスは CLI では System.Type クラスとして表される。（Type は抽象クラスであり、実装は派生クラスが行う。）Type の機能の一部はリフレクションと呼ばれ、そのインスタンスが表す型の名前や、フィールド・メソッドの一覧などを得ることができる。しかし、メタデータ構築の初期段階ではアセンブリなどメタデータに関する情報が整っていないため、リフレクションの全ての機能を実現することはできない。そこで、初期段階では完全な実装を諦め、限定的な（ただし初期手順の実行には十分な）機能だけを実装した PseudoType クラスを用意した。

PseudoType を用いて、次のような手順でメタクラスを生成する。

1. **PseudoType を表す PseudoType のインスタンス P を確保**
   「PseudoType を表す PseudoType のインスタンス」とは、リフレクションによって PseudoType に関する情報を得られる PseudoType 型のインスタンスのことである。これのための領域をメモリ上に確保しゼロで初期化する。このときコンストラクタは呼び出さない。これを P とし、P の型オブジェクト[*1]を P に設定する。
   この一連の操作はレガシーカーネルにハードコーディングするため、その処理の内容は CLI の命令に限定されない。そのため、コンストラクタを呼び出さずに P を確保

[*1] あるインスタンス i の型を表す Type オブジェクトを型オブジェクトと呼ぶことにする。CLI では i.GetType() で型オブジェクトを得られる。

するといったことが可能である。

2. **P の親クラスを処理**
   P の親クラスに関する型オブジェクトを作成する。（コンストラクタも呼び出す。）作成した型オブジェクトの型オブジェクトは P になる。
3. **P を構築**
   P のコンストラクタを呼び出す。このとき、リフレクションに必要な名前や名前空間名などが渡される。
4. **その他のメタクラスを作成**
   インタープリタが参照した時点で、必要となった型のメタクラスが作成される。

以上の処理が終わった後のメタクラスに関する様子を**図 6.8** に示す。なお、Int32[*2]は 32 ビット符号付き整数の型である。また、MyClass はユーザ定義のクラス型であるとし、myInstance は MyClass のインスタンスであるとする。

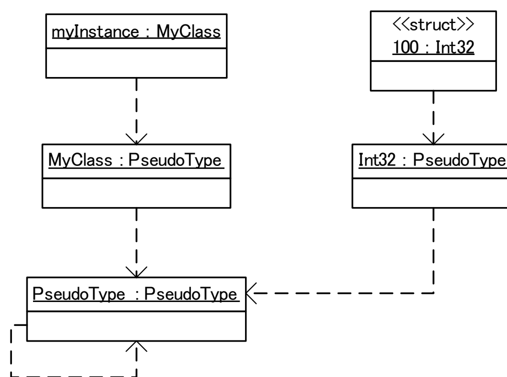


図 6.8　切り替え前のメタデータ構造

PseudoType は PseudoType 自身が定義されているアセンブリが読み込まれていない状態で使用されるため、PseudoType には本来 Type の派生クラスが持つべき機能の一部しか実装できないが、手順 3 までの実行においては未実装の機能は使用されないため問題にはならない。

手順 2: アセンブリのロード

PseudoType を整えたのち、レガシーカーネルはアセンブリファイルのイメージを CD-ROM から読み込み、マネージカーネルのアセンブリローダにイメージを解析・ロードさせる。ロードが終了すると、Int32 や Object などの基本的な型やマネージカーネルに含まれるクラスなどのメタデータに、CLI プログラムがアクセスできるようになる。

手順 3: 真のメタクラスへ切り替え

アセンブリのロードが終わると、PseudoType を用いて生成されたメタデータを正しい実装系へ切り替える。切り替えた後の様子を**図 6.9** に示す。なお、ClassType と PrimitiveType は PseudoType に代わる正しいリフレクション実装を持つクラスであり、ClassType はクラス型、PrimitiveType は Int32 などの基本型に適用されるメタクラスである。mscorlib およ

[*2] System.Int32

び cscorlib はアセンブリの名前である。

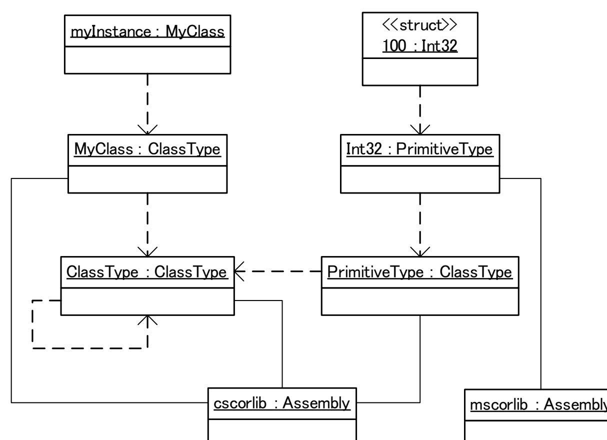


図 6.9　切り替え後のメタデータ構造

ClassType など正しいリフレクション実装のためのクラスは、自分が定義されたアセンブリを知っており（図 6.9 の Assembly オブジェクト）メタデータを参照することができるため、リフレクション機能を完全に実装することができる。

切り替えは、レガシーカーネルによって、メモリ中の参照を書き換えることで行われる。すでにメモリ中にロードされている（つまり使われている）型は、PseudoType を用いてロードされてしまっているため、これらについてもう一度 ClassType などで型情報をロードし直して、PseudoType を参照している箇所をロードし直した ClassType などへの参照に書き換える。これによって、以降、リフレクションを完全な形で利用可能になる。

6.3　コンパイル

メタデータがリフレクションを通して利用可能になると、その情報を基にコンパイルすることができるようになる。コンパイル作業は次のような手順で行う。[†]

† 命令単位の翻訳処理については 5.6 節および 6.5 節参照。

手順 1: 基本コレクションクラスのコンパイル

基本コレクションクラス（5.3 節参照）は仮想関数呼び出しの処理など基本的な命令の実行に必要なので、基本コレクションクラスのメソッドに対して JIT コンパイルが発生してはいけない。そこで、基本コレクションクラスのメソッドを予めコンパイルすることによって JIT コンパイル処理が発生することを防いでいる。これは、基本コレクションクラスの実装にあたり、仮想関数の呼び出しなど動的束縛が起こる実装を避けているために可能である。

ここで、コンパイルしたメソッドが直ちに機械語実行されるわけではない。コンパイルしたメソッドが機械語で実行されるのは、他の機械語から呼び出されたときになる。

手順 2: コンパイルメソッドのコンパイル

基本コレクションクラスに続き、コンパイル処理の開始点となるメソッドをコンパイルして機械語 M を生成する。そして、コンパイル対象メソッド F を、M によってコンパイルする。

このとき、M が呼び出したメソッドも機械語で実行するために JIT コンパイルされる。た

だし、JIT コンパイル処理自体はインタープリタで実行されるようになっているため、コンパイルで実行されるメソッドのコンパイル処理はインタープリタによって行われる。これは JIT コンパイル処理が循環しないようにするためである。

この手順が終わったとき、コンパイルに必要なメソッドの機械語が（JIT コンパイルによって）得られていると考えられる。

手順 3: 再リンク

これまでの手順で生成した機械語は JIT コンパイルのためにインタープリタを呼び出してしまうため、それらの呼び出し部分を機械語の JIT コンパイル手続きへの呼び出しに書き換える。

手順 4: カーネル開始メソッドのコンパイル

最後にマネージカーネルのエントリポイント（4.3 節参照）をコンパイルし、そのメソッドを呼び出す。

6.4　JIT コンパイルの仕組み

JIT コンパイルのために、コンパイラはメソッド `Caller` がメソッド `Callee` を呼び出す次のような call 命令

```
call Callee
```

を次のようなコード

```
call Stub
```

へと置き換え、同時に `Stub` として次のようなコード

```
ldc.i4 <Assembly ID>
ldc.i4 <Row Index>
call PrepareCode
// ここで呼び出し元の書き換え処理
// ここで制御の転送
```

を用意する。ただし、`<Assembly ID>`とは `Callee` が定義されているアセンブリの識別子であり、`Row Index` とは `Callee` のアセンブリ内定義位置である。PrepareCode は指定されたメソッドの機械語を用意するメソッドである。この様子を**図 6.10** に示す。

呼び出し元の書き換え処理で、`Caller` に含まれる `Callee`（実際には `Stub`）への機械語呼び出し命令のオペランド（呼び出しアドレス）を、`Callee` の機械語コードになるように書き換える。これによって次回の `Caller` の実行時はランタイムを介さずに `Callee` が呼び出されることになる。

制御の転送は通常の呼び出し手続きではなく、機械語の `jmp` によって処理される†。これによって、`Stub` が `Callee` の戻り値などに注意を払う必要がなくなるなど、処理を単純化できる。

† つまり、転送先から戻ってくることはなく、転送先から自身の呼び出し元へと直接戻るようにプログラムされている。

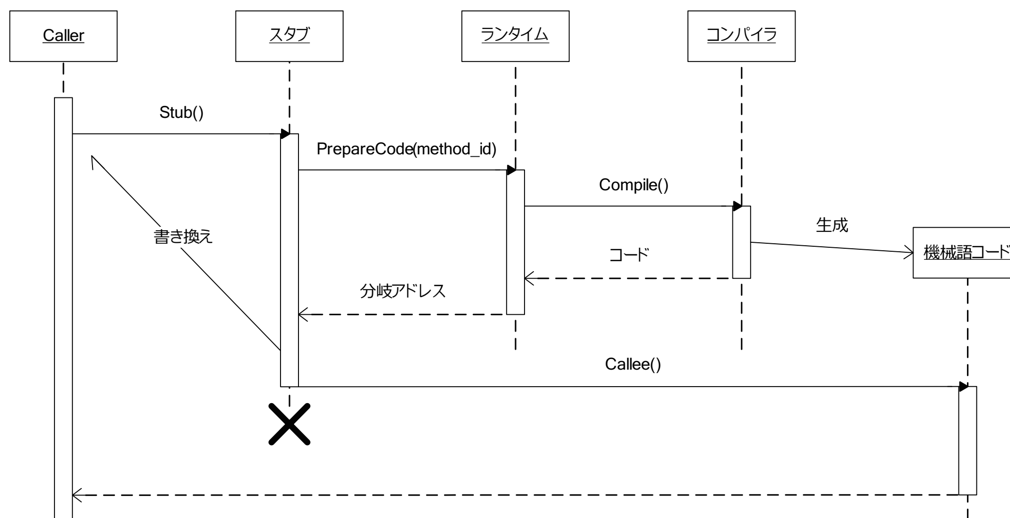


図 6.10 JIT コンパイル時のシーケンス

6.5 CIL 命令の翻訳

スタックマシンを想定した CIL 命令の翻訳についての基本はすでに 5.6 節で述べているが、CIL の仕様では単純に IA-32 プロセッサ機械語に変換しにくい命令がある。ここではそのような命令をどのように翻訳するか説明する。

6.5.1 8 バイトを超える返戻値の受け渡し

IA-32 においては 8 バイトを超えるサイズのデータはレジスタを通じて受け渡ししにくい†ため、引数の最後に戻り値の型へのポインタが渡されると仮定してコンパイルする。つまり、HugeValueType が 8 バイトを超えるサイズの値型としたとき、メソッド `f`（**テキスト 6.1**）への呼び出し文（**テキスト 6.2**）は、実際には**テキスト 6.3** と解釈されて翻訳される。

† 8 バイト以下はレジスタを 2 つまたは 1 つ使って受け渡せる。

テキスト 6.1 メソッド f

```
1  static HugeValueType f() {
2      return new HugeValueType();
3  }
```

テキスト 6.2 f の呼び出し文

```
1  HugeValueType v;
2  v = f();
```

テキスト 6.3 f の変更された呼び出し文

```
1  HugeValueType v;
2  f(&v);
```

6.5.2　補助ルーチン

CIL の命令には単純な機械語への変換が不可能なものがある。そのような命令はコンパイラが備える補助ルーチンの呼び出しに置き換えられ、実行時には命令の機能を補助ルーチンが代行する。補助ルーチンはコンパイラの一部として C#で記述されている。次に、それら補助関数を使用して実装命令の一部について述べる。**表 6.1** に、補助ルーチンを必要とする命令と、対応する補助ルーチンを示す。

表 6.1　補助ルーチン一覧

命令	メソッド
newobj	AllocateObjectImpl
newarr	AllocateSzArrayImpl
ldstr	LoadStringImpl
box	BoxImpl
isinst	IsInstImpl
castclass	CastClassImpl
call	PrepareCode *
callvirt	FindActualMethod *
callvirt	FindActualMethodForInterface *

このうち、現状では*付きのルーチンの実行はインタープリタに頼っている。この問題については 8 章で触れる。

newobj と AllocateObject

newobj は、指定された型のインスタンスを、ヒープまたはスタックに確保する命令である。処理対象の型は命令の一部としてエンコードされているため静的に決定できる。型が参照型の場合はヒープに領域を確保し、確保したインスタンスへの参照をスタックに積む。値型の場合は単にインスタンスをスタックに積む。どちらの場合にも、確保したインスタンスに対して、コンストラクタが呼ばれる。

参照型に対する場合、コンパイラの AllocateObject メソッドとそのあとのコンストラクタへの call で処理される。AllocateObject は現在のメモリの空き領域の先頭を初期化し、その領域のポインタを Object への参照として返戻する。

値型に対する場合、コンパイラはコンパイル中のメソッドが必要とする作業領域[*3]に、値型のサイズ分を追加する。（以降、生成領域と呼ぶ。）これにより、実行時にはこの生成領域をスタック状態と無関係に利用可能になる。

実行時には、生成領域を値型のインスタンスとして初期化し、生成領域へのポインタをコンストラクタの引数として渡す。コンストラクタの実行後に、生成領域からスタックへインスタンスそのものをコピーし、スタックに積む。

callvirt と FindActualMethod

callvirt は仮想メソッド呼び出しのための命令であり、オペランドのインスタンスの実際の型に基づいてメソッドを呼び出す。callvirt は実際には次のような呼び出しに展開される。

[*3] スタックフレームにおかれる、いわゆる局所変数のための領域に相当する。

1. FindActualMethod メソッドへの call
2. 返戻値が指すアドレスへのジャンプ

6.5.3 プリコンパイル

実行中にコンパイラの中でコンパイル処理が再帰してしまわないようにするため、一部のコードについてプリコンパイルを行う。対象となるメソッドを手作業で列挙するのは非合理的なので、明示的に指定したメソッドから（直接的であれ間接的であれ）call によって呼ばれるメソッドもプリコンパイルの対象とした。

明示的な対象となるのは、

1. 全てのメタタイプクラス
2. コンパイラの補助ルーチン

である。（対象がクラスの場合はそれに含まれる全てのメソッドである。）

第 7 章

検証と評価

本研究で開発したオペレーティングシステムが正しく動作しているか調べるための試験を行った。本章ではそれら試験について述べ、結果を示す。

7.1　実験環境

提案手法を実装したカーネルを CooS に搭載し、VMware[26]上でオペレーティングシステムとして動作させた。ホストマシンの CPU は Intel Pentium4 3.40GHz であり、VMware には 128MB のメモリ領域を割り当てた。また、試験はしていないが、VMware のほかにも QEMU[1), 33)]や[22)]でも動作確認を行っている。

なお、検証のためのプログラムコードはすべて C#であり、ソースファイル全体ではなく重要な箇所のみを抜き出している。

7.2　インタープリタの数値演算

† 32 ビット浮動小数点数型。

† 64 ビット浮動小数点数型。

インタープリタについて、Int32 と Int64 と Single†と Double†の算術演算（および整数型には論理演算）を計算させた。この 4 種を選んだ理由は、CIL におけるスタック上の数値が実質的にこの 4 種で表されるからである。**テキスト 7.1** に Double に関する試験のコードの抜粋を示す。ただし、`TestCalc` は引数の四則演算を行い、結果を表示するメソッドである。

テキスト 7.1　Double 型検証コード

```
1  //static void TestCalc(double x, double y);
2  double[] arr = {0, 1, −1, 10, −10, short.MaxValue, short.MinValue
3        , int.MaxValue, int.MinValue, double.MaxValue, double.MinValue
4        , double.PositiveInfinity, double.NegativeInfinity, double.NaN };
5  foreach(double x in arr) {
6        foreach(double y in arr) {
7              TestCalc(x, y);
8        }
9  }
```

この結果、Int32 と Int64 については、このコードを Windows 上の.NET Framework で計算させたときと同じ結果が得られた。Single と Double については、最大の数や最小の数の計算で差違があったものの、計算規則†については等しい結果が得られた。

† たとえば、NaN に対する四則演算結果は NaN になる、など。

違いが見られた Double 最大値の表示結果を**表 7.1** に示す。この違いは、コンソールへの

表 7.1　Double.MaxValue の値

実行系	Double.MaxValue の値
.NET Framework	1.79769313486232E+308
Our Interpreter	1.77008066823353E+308

表示をする際の計算誤差だと思われる。インタープリタでは計算結果を逐一 32 ビットまたは 64 ビットでスタックに格納しているのに対し、.NET Framework は最適化によって可能な限り FPU レジスタ（IA-32 の場合は最大 80 ビット）を利用しており、計算過程か文字列変換過程で誤差が発生していると考えられる。しかし、この誤差は Double.MaxValue という非常に大きな数値付近でのみ発生し、Int.MaxValue までの計算は発生しなかった。このため、インタープリタが利用される段階では浮動小数点数があまり使用されないということもあり、動作には影響しないと考えられる。

7.3 インタープリタの動作

インタープリタによって、たとえばメソッド呼び出しなどが正しく動作するかといった実験は、インタープリタが後続の試験の基礎であることを考え、改めて行うことはしなかった。

後続の試験を行うことができていることから、少なくとも、Mono クラスライブラリ[13)]や FreeType[23)]について、それらの動作結果を確認できる程度には正しく実装されていると言える。

7.4 メタデータの構築

メタデータが正しく構築できていることを確認するために**テキスト 7.2** を実行した。実行

テキスト 7.2　メタデータ検証コード

```
static void ShowMetatype(object o) {
    Console.WriteLine("{0}", o);
    Console.WriteLine(">␣{0}"
        , o.GetType().Name);
    Console.WriteLine(">>␣{0}"
        , o.GetType().GetType().Name);
}
static void TestMetatypes() {
    ShowMetatype(1);
    ShowMetatype("abc");
    ShowMetatype(typeof(int));
    ShowMetatype(typeof(Console));
}
```

結果（**図 7.3**）より、たとえば 1 という 32 ビット符号付き整数の型は Int32 であり、それの型は PremitiveType であり、それの型は ClassType であり、最終的に ClassType の型は ClassType であるといったように、メタデータが正しく構築されている様子が分かる。

テキスト 7.3　メタデータ検証結果

```
1
> Int32
>> PrimitiveType
abc
> String
>> StringType
Type: System.Int32
> PrimitiveType
>> ClassType
Type: System.Console
> ClassType
>> ClassType
```

7.5　コンパイラ

インタープリタとメタデータが利用できることが確認できたので、次にそれらに基づきコンパイラが正しく動作するか確認するための**テキスト 7.4** を実行した。計測は各 x について

テキスト 7.4　単純ループ

```
1  public static int Calculate(int x) {
2      while(x>0) {
3          --x;
4      }
5      return x;
6  }
```

10 回試行し、各計算でもっとも良い数値を採用した[†]。時間の単位はクロックである。引数 x を様々に変えたときの処理時間の結果を**図 7.1** に示す。横軸は引数 x の値であり、縦軸は処理クロック数である。両軸とも対数目盛りである。

† 平均などではなく、最もよい値を採用するのは、それが外部からの影響を除いた純粋な実行時間に近いと考えられるからである。

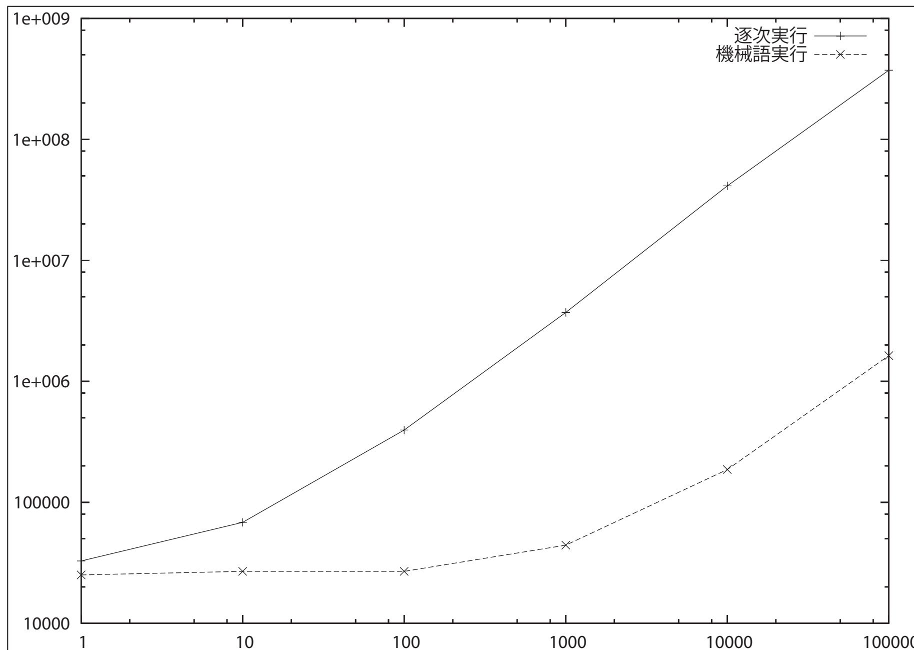


図 7.1　単純ループの実行時間

グラフより、逐次実行より機械語実行の方が 100 倍以上の高速であることが分かる。$x = 1$ で時間が 0 でないのは計測のための定数コストのためであると考えられる。

次に、メソッド呼び出しを多用するコードサンプルとして**テキスト 7.5** を実行した。結果を**図 7.2** に示す。計測は各 x について 10 回試行し、各計算でもっとも良い数値を採用した。

テキスト 7.5　総和計算

```
1  public static int Calculate(int x) {
2      if(x==0) return 0;
3      return x+Calculate(x-1);
4  }
```

時間の単位はクロックである。引数 x を様々に変えたときの処理時間の結果を表 7.2 に示す。横軸は引数 x の値であり、縦軸は処理クロック数である。両軸とも対数目盛りである。グラフより、逐次実行が機械語実行に比べて極端に遅くなっていくことが分かる。この原因

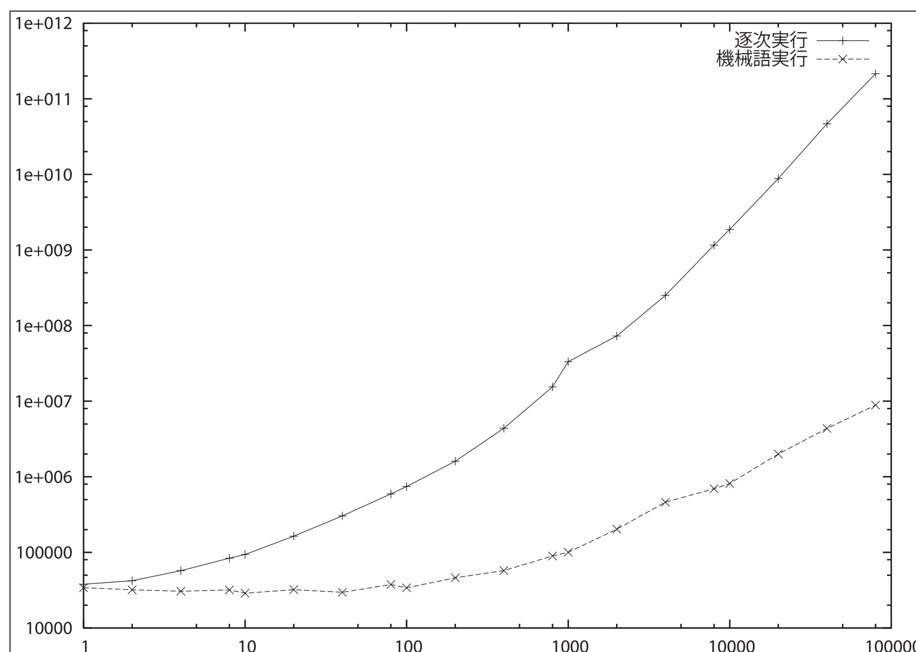


図 7.2 再帰による総和計算の実行時間

は、機械語実行は（原理的には）呼び出しの深さと処理時間に関係ないのに対して、インタープリタの一部のデータ構造は呼び出しの深さにつれて操作に掛かるコストが増加するためだと思われる。問題箇所を最適化することで性能改善は期待できるが、機械語より速くなる見込みは薄く、機械語実行の速度的な優位性が示されたといえる。

7.6 JIT コンパイル

次に JIT コンパイルが動作することを確認するために**テキスト 7.6** に示すようなコードを用意し、`Calculate1` のみをコンパイルしたのち実行した。

実行したときの出力を**図 7.7** に示す。まず Calculate1 がコンパイルされており、`0x0211926Ch` 番地にコードが配置されている（1〜2 行目）。同時に、Calcualte1 が呼び出す `Calculate2` を JIT コンパイルするためのコード（スタブコード）が `0x02119BF4h` 番地に配置されている（3 行目）。

最初の $x = 0$ の計算では Calculate2 を呼び出さないため、計算はそのまま終了していることが分かる（4 行目）。次に $x = 1$ の実行では、スタブコードが呼び出されてコンパイルが行われており、Calculate2 の実行可能コードが `0x0211D744h` 番地に配置されている（5〜6 行目）。このとき、Calculate1 の Calculate2 呼び出し命令の呼び出し先を `0x0211D744h` 番地に書き換えている。

以降の実行では JIT コンパイルは呼び出されることなく計算が行われている（7〜8 行目）。

テキスト 7.6　JIT コンパイル検証コード

```
1  static int Calculate1(int x) {
2      if(x==0) return 0;
3      return x+Calculate2(x−1);
4  }
5  static int Calculate2(int x) {
6      if(x==0) return 0;
7      return x+Calculate1(x−1);
8  }
9  static int Test(int x) {
10     Console.WriteLine("x={0,3},␣y={1,3}"
11         , x, Calculate1(x));
12 }
13 static int Test() {
14     Test(0);
15     Test(1);
16     Test(10);
17 }
```

テキスト 7.7　JIT コンパイルの応答

```
1  COMPILE> sum:Calculate1
2  [0x0211926C:E] sum:Calculate1
3  [0x02119BF4:C] sum:Calculate2
4  x= 0, y= 0
5  COMPILE> sum:Calculate2
6  [0x0211D744:E] sum:Calculate2
7  x= 1, y= 1
8  x= 10, y= 55
```

7.7　オペレーティングシステム動作試験

提案手法がカーネルにおいても有効であることを確認するために、さらに次のようなハードウェアデバイスドライバを C#で開発しシステムに加えた。

- VGA
- ATA/ATAPI
- キーボードコントローラ
- PS/2 キーボードおよび PS/2 マウス

また、アウトラインフォントをレンダリングするための FreeType[23)]を加え、オペレーティングシステムとして必要なシェルなどを組み込んだのち、テキストモードとグラフィックモードそれぞれで CooS を起動した。テキストモードでの起動画面を**図 7.3** に、グラフィックモードでの起動画面を**図 7.4** に示す。

起動したのち、フォントのレンダリングが動作することを確認するために、枕草子の一部を画面に出力した。そのときの様子を**図 7.5** に示す。また、TrueType フォントのサイズ可変という特徴が活かせているか確認するために、フォントサイズを非常に大きくして同じ文章を画面に表示した（**図 7.6**）。このとき、画面への出力に加えて、指示を与えるためのキーボード入力、フォントファイルの CD-ROM 媒体から読み込みなども行っているため、

```
CooS version 0.3 built on Feb  6 2006 14:39:09  Mem: 128960KB                   --\

Load primitive commands
Registered [ls] with ID#9
ls, Version=0.0.0.0
signature: 0x424A5342
version: 0x0001.0x0001
length: 0x0000000C
version: v1.1.4322
Assembly: ls(0x00000028) 0.0.0.0 (0)
OK: ls (Version 0.0.0.0)
Registered [cat] with ID#10
cat, Version=0.0.0.0
signature: 0x424A5342
version: 0x0001.0x0001
length: 0x0000000C
version: v1.1.4322
Assembly: cat(0x00000029) 0.0.0.0 (0)
OK: cat (Version 0.0.0.0)

CooS [Version 0.3.2228.25815]

Type '?' to show operation guide.
cd0a:/>
Press CTRL+ALT+DEL to reset
```

図 7.3　起動画面（テキストモード）

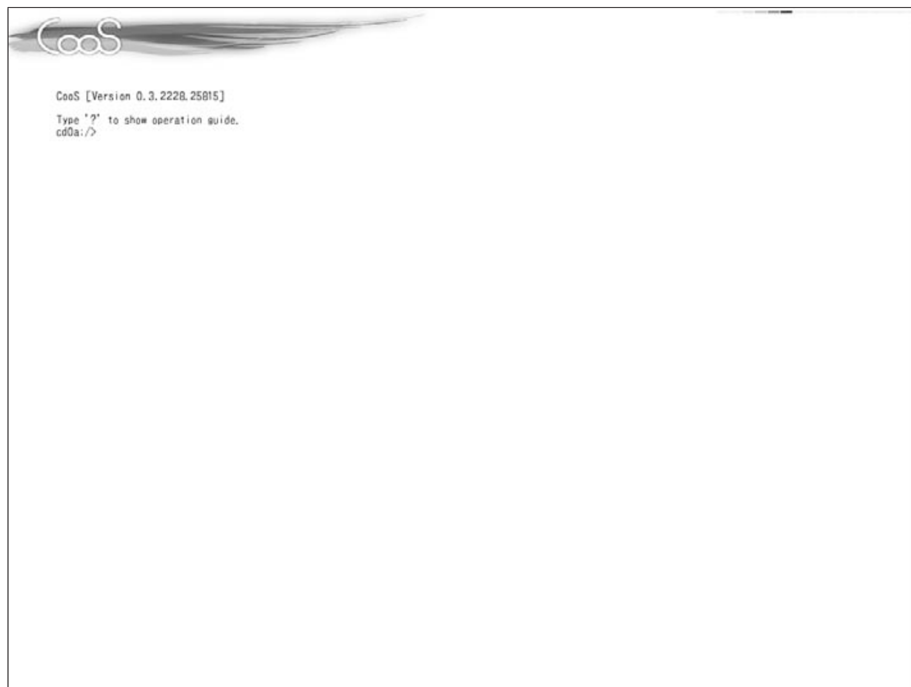


図 7.4　起動画面（グラフィックモード）

それらについても併せて動作を確認できたといえる。

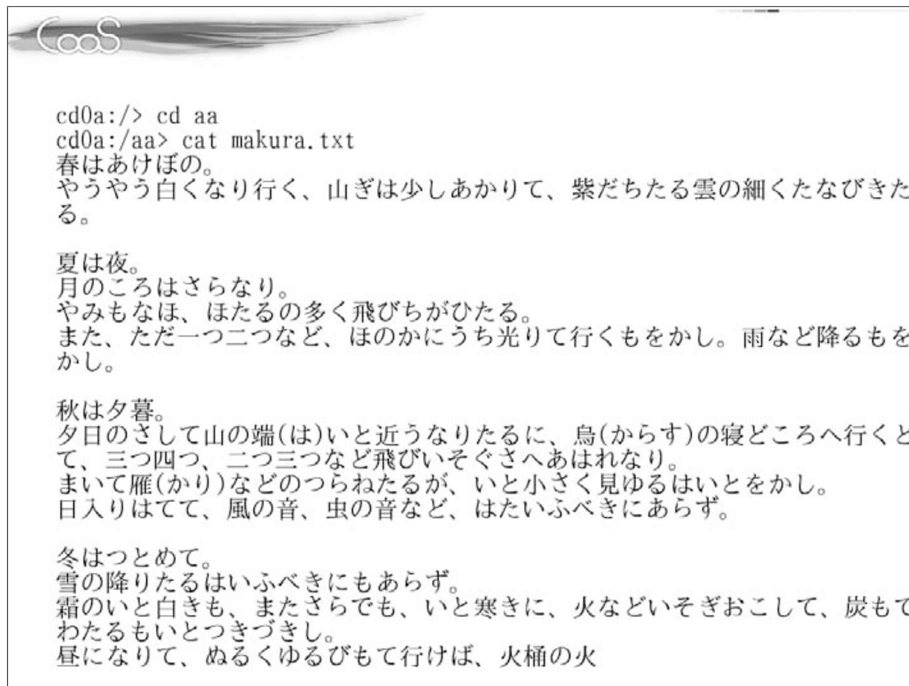


図 7.5　フォントレンダラで日本語を表示した画面（普通のフォントサイズ）

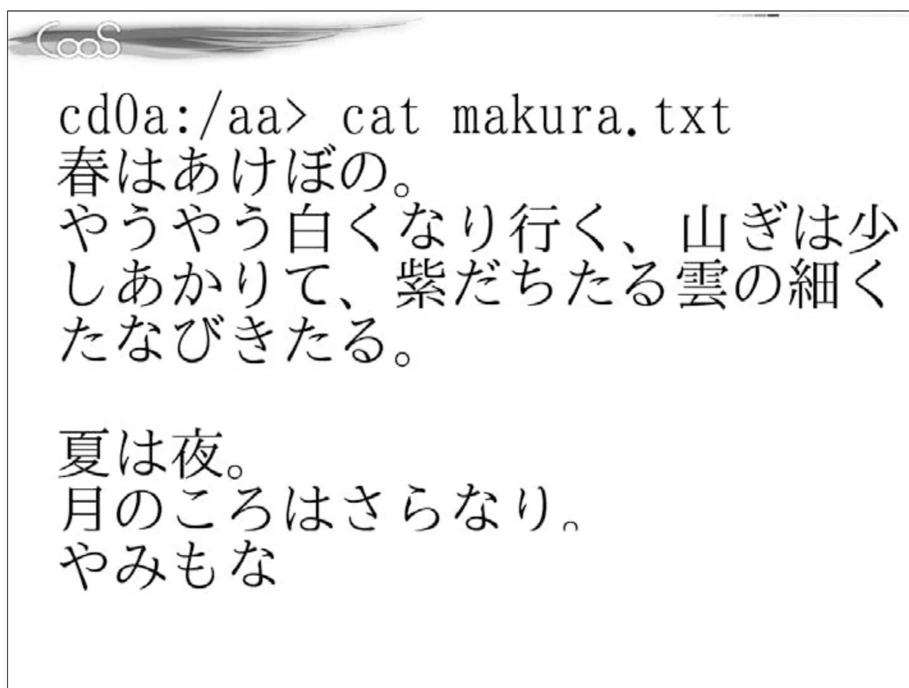
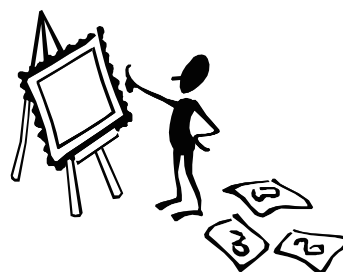


図 7.6　フォントレンダラで日本語を表示した画面（大きいフォントサイズ）

第 8 章

結論



本研究では、CLI のためのオペレーティングシステム CooS を実装し、実際に動作させて評価した。また、レガシーカーネルとマネージカーネルという二つのカーネルを組み合わせて動作させる手法を提案し、起動のための特別なデータを用意することなくシステム全体を CLI に準拠させた状態でカーネルを実行することができた。本論文では今回の開発を俯瞰できるだけの技術について簡単に説明するとともに、提案手法の核心であるメタデータの構築手順とコンパイル手順について詳しく説明し、評価によって提案手法がカーネルの実行手法として有効であることを示した。

現状では、表 6.1 において*付きで示したメソッドの実行を依然としてインタープリタに頼っている。そのため、インタープリタをコンピュータ上から消すまでには至っていない。これらのメソッドへの依存を取り除けない原因は、6.3 項で述べた「コンパイラのコンパイル」が不十分であるためにコンパイル処理が循環するからである。この問題はコンパイル処理を行うメソッドのコールグラフを解析して必要なメソッドだけを予めコンパイルしておくことで解決できると考えられるため、現在はこの依存関係の同定作業を行っている。

付録 A

ソースファイル

A.1 bootloader

bootloader はブートストラップ・ローダをビルドするプロジェクトである。以降に bootloader のすべてのソースファイルを示す。

テキスト A.1 bootldr.h

```
1 int _asm_c();
2 int _asm_int();
3 long _asm_long();
4
5 #define _enstring_(s) # s
6 #define combine(a,b) (((int)(a)<<8)|(b))
7 #define makelong(a,b) (((long)(a)<<16)|(b))
8
9 #define true 1
10 #define false 0
11 #define NULL 0
12
13 int boot();
14 void freeze();
15
16 int main() {
17     _asm_c(
18         " db 0xEA \n"
19         " dw 0x0005 \n"
20         " dw " _enstring_(__eval__(CODE_SEGMENT)) " \n"
21         );
22     _asm_c(
23         " mov ax, " _enstring_(__eval__(DATA_SEGMENT))
              " \n"
24         " mov ds, ax \n"
25         " mov es, ax \n"
26         );
27     boot();
28     _asm_c("retf");
29 }
30
31 void freeze() {
32     for(;;) {
33         _asm_c("hlt");
34     }
35 }
36
37 void printc(char ch) {
38     _asm_c(
39         "mov ah, 0x0E \n"
40         "mov bx, 7 \n"
41         "int 10h \n"
42         , ch);
43 }
44
45 void prints(const char* s) {
46     while(*s) {
47         printc(*(s++));
48     }
49 }
50
51 int read(int segment, int offset, int sector, int count)
     {
52     int track, head, cylinder;
53     --sector;
54     track = sector/18;
55     head = track&1;
56     cylinder = track>>1;
57     sector %= 18;
58     ++sector;
59     _asm_c("mov es, ax", segment); /* Segment of buffer
           */
60     head = _asm_c(
61         "int 13h   \n" /* Diskette services */
62         "mov bx, ds  \n"
63         "mov es, bx  \n"
64         , /*AX*/ 0x0200|count    /* Read diskette sectors
              */ /* Number of sectors (1-15) */
65         , /*BX*/ offset      /* Offset of buffer */
66         , /*CX*/ combine(cylinder,sector) /* Track number
              (0-79) */ /* Sector number (8-36) */
67         , /*DX*/ combine(head,0)   /* Head number (0-1) */
                /* Drive number (0-1) */
68         );
69     return head;
70 }
```

テキスト A.2 bpb.asm

```
1 %define BytesPerSector  512
2 %define SectorsPerFAT  9
3 %define SectorsPerCluster 1
4 %define SectorsPerTrack  18
5 %define ReservedSectorCount 8
6 %define RootEntryCount  32
7 %define NumberOfHeads  2
8 %define NumberOfFAT  2
9 %define NumberOfSectors  0x0b40
10
11 %define RootDirSectors  ((RootEntryCount*32) + (
      BytesPerSector-1)) / BytesPerSector
12 %define FirstDataSector  ReservedSectorCount + (
      NumberOfFAT * SectorsPerFAT) + RootDirSectors
13
14 [bits 16]
15 [org 0]
16     jmp short start
17     nop
18
19 ; boot parameter block(bpb)
20 oem db "CLIOS "
21 bps dw BytesPerSector   ; Bytes per Sector
22 spc db SectorsPerCluster  ; Sectors per Cluster
23 rsc dw ReservedSectorCount  ; Reserved sector count: spec
       dictates this should be 1.
24 nof db NumberOfFAT   ; Number of FAT
25 rec dw RootEntryCount   ; Root Entry Count
26 nos dw NumberOfSectors   ; Total sector count
27 med db 0xf0    ; MediaType
28 spf dw SectorsPerFAT   ; Sectors per FAT
29 spt dw SectorsPerTrack   ; Sectors per Track
30 noh dw NumberOfHeads   ; Number of heads
31     dd 0
32     dd 0
33     db 0
34     db 0
35     db 0x29
36     dd 0x00000000
37     db "clios boot "
38     db "FAT12 "
39
```

```
; ブートローダースタート
start:
    mov sp, 0x4000
    mov ax, 0x0200
    mov ss, ax
    call 0x07C0:firstboot
    call 0x0120:0x0000
    call 0x0100:0x0000

;0で埋める
    times 0x60-($-$$) db 0
firstboot:
    ; COMBINED WITH 1stboot.c
```

テキスト A.3　1stboot.c

```
#define CODE_SEGMENT 0x07C6
#define DATA_SEGMENT 0x07DC
#include "bootldr.h"


int boot() {
    prints("CooS First BOOTLOADER...");
    read(0x0100, 0, 2, 8);
    prints("OK\r\n");
}
```

テキスト A.4　2ndboot.c

```
#define CODE_SEGMENT 0x0120
#define DATA_SEGMENT 0x01C0
#include "bootldr.h"


#define BytesPerSector  512
#define SectorsPerFAT  9
#define SectorsPerCluster 1
#define SectorsPerTrack  18
#define ReservedSectorCount 8
#define RootEntryCount  32
#define NumberOfHeads  2
#define NumberOfFAT   2
#define NumberOfSectors  0x0B40


void printn(unsigned int n, int base);
void set32(long farptr, int offset, long value);

void clearCursor();
void clearA20();
int getMemorySize();
void initVbe(int seg, void* vbeinfo);
int initVbeMode(int seg, void* vbeinfo, int dstseg);
int switchVbeMode(int mode);
long loadKernel(unsigned int dstseg);


int boot() {
    int memsize;
    int vbemode;
    long size;
    prints("CooS Second BOOTLOADER\r\n");
    clearCursor();
    clearA20();
    memsize = getMemorySize();
    set32(0, 0x06FC, memsize*64);
    printn(memsize*64, 10);
    prints(" KB MEMORY AVAILABLE\r\n");
    initVbe(0x0070, 0);
    vbemode = initVbeMode(0x0070, 0, 0x0080);
    size = loadKernel(0x1000);
    set32(0, 0x6F8, size);
    /* 画面切り替え *
    if(vbemode>0) switchVbeMode(vbemode); else/**/
    set32(0, 0x828, NULL);
    _asm_c("db 0xEA \n dw 0x1000, 0 ;=0x0:0x1000", 0x1000);
}

void printn(unsigned int n, int base) {
    if(n==0) {
        printc('0');
    } else {
        char buf[32];
        char* p;
        p = buf;
        while(n>0) {
            *p = (n%base)+'0';
            if(*p>'9') *p=*p-'9'+'A'-1;
            ++p;
            n /= base;
        }
        --p;
        while(p>=buf) {
            printc(*(p--));
        }
    }
}

void printd(int n) {
    if(n<0) {
        printc('-');
        n=-n;
    }
    printn(n, 10);
}

void printh(int n) {
    printc('0');
    printc('x');
    printn((unsigned int)n, 16);
}


char vbeinfo[512];

int get16(long farptr, int offset) {
    int s = ((int*)&farptr)[1];
    int o = (int)farptr+offset;
    return _asm_int("push ds \n mov ds, ax \n mov ax, [bx] \n pop ds", s, o);
}

void set16(long farptr, int offset, int value) {
    int s = ((int*)&farptr)[1];
    int o = (int)farptr+offset;
    _asm_int("push ds \n mov ds, ax \n mov [bx], cx \n pop ds", s, o, value);
}

long get32(long farptr, int offset) {
    int s = ((int*)&farptr)[1];
    int o = (int)farptr+offset;
    return _asm_long("push ds \n mov ds, ax \n mov ax, [bx] \n mov bx, [bx+2] \n pop ds", s, o);
}

void set32(long farptr, int offset, long value) {
    int s = ((int*)&farptr)[1];
    int o = (int)farptr+offset;
    _asm_long("push ds \n mov ds, ax \n mov [bx], cx \n mov [bx+2], dx \n pop ds", s, o, (int)value, ((int*)&value)[1]);
}

int getCodeSeg() {
    return _asm_c("mov ax, cs");
}

int getDataSeg() {
    return _asm_c("mov ax, ds");
}

int getStackSeg() {
    return _asm_c("mov ax, ss");
}

void initVbe(int seg, void* vbeinfo) {
    long l;
    int i;
    prints("GRAPHICS: ");
    getVbeInfo(seg, vbeinfo);
    l = get32(makelong(seg, (int)vbeinfo), 6);
    for(i=0; ; ++i) {
        char ch = (char)get16(l, i);
        if(ch=='\0') break;
        printc(ch);
```

```
133     }
134     prints("\r\n");
135 }
136
137 int searchMode(int seg, void* vbeinfo, int xres, int yres
        , int depth) {
138     long l = get32(makelong(seg,(int)vbeinfo), 14);
139     int i;
140     for(i=0; i<64; ++i) {
141         char buf[256];
142         int mode = get16(l,i*2);
143         if(mode==0xFFFF) break;
144         if(!getVbeModeInfo(mode, getStackSeg(), buf)) {
145             prints("VBE MODE INFO FAILED\r\n");
146         } else {
147             short x = *(short*)&buf[18];
148             short y = *(short*)&buf[20];
149             char d = buf[25];
150             if(x>0 && y>0 && d>0
151             && (xres<=0 || x==xres)
152             && (yres<=0 || y==yres)
153             && (depth<=0 || d>=depth))
154             {
155                 return mode;
156             }
157         }
158     }
159     return -1;
160 }
161
162 int initVbeMode(int seg, void* vbeinfo, int dstseg) {
163     long l;
164     int mode;
165     l = get32(makelong(seg,(int)vbeinfo), 14);
166     mode = -1;
167     if(mode<0) mode=searchMode(seg, vbeinfo, 1152, 864,
            24);
168     if(mode<0) mode=searchMode(seg, vbeinfo, 1280, 1024,
            24);
169     if(mode<0) mode=searchMode(seg, vbeinfo, 1024, 768,
            24);
170     if(mode<0) mode=searchMode(seg, vbeinfo, 800, 600,
            24);
171     if(mode<0) mode=searchMode(seg, vbeinfo, 1152, 864,
            16);
172     if(mode<0) mode=searchMode(seg, vbeinfo, 1280, 1024,
            16);
173     if(mode<0) mode=searchMode(seg, vbeinfo, 1024, 768,
            16);
174     if(mode<0) mode=searchMode(seg, vbeinfo, 800, 600,
            16);
175     if(mode<0) mode=searchMode(seg, vbeinfo, 1152, 0, 24)
            ;
176     if(mode<0) mode=searchMode(seg, vbeinfo, 1280, 0, 24)
            ;
177     if(mode<0) mode=searchMode(seg, vbeinfo, 1024, 0, 24)
            ;
178     if(mode<0) mode=searchMode(seg, vbeinfo, 800, 0, 24);
179     if(mode<0) mode=searchMode(seg, vbeinfo, 1152, 0, 16)
            ;
180     if(mode<0) mode=searchMode(seg, vbeinfo, 1280, 0, 16)
            ;
181     if(mode<0) mode=searchMode(seg, vbeinfo, 1024, 0, 16)
            ;
182     if(mode<0) mode=searchMode(seg, vbeinfo, 800, 0, 16);
183     if(mode<0) mode=searchMode(seg, vbeinfo, 640, 480,
            24);
184     if(mode<0) mode=searchMode(seg, vbeinfo, 640, 480,
            16);
185     if(mode<0) mode=searchMode(seg, vbeinfo, 640, 0, 24);
186     if(mode<0) mode=searchMode(seg, vbeinfo, 640, 0, 16);
187     if(mode==-1) {
188         prints("SUITABLE VBE MODE NOT FOUND");
189         freeze();
190         return -1;
191     } else {
192         char buf[256];
193         if(!getVbeModeInfo(mode, getStackSeg(), buf) || !
                getVbeModeInfo(mode, dstseg, (void*)0)) {
194             prints("VBE MODE INFO FAILED\r\n");
195             freeze();
196         } else {
197             short xres = *(short*)&buf[18];
198             short yres = *(short*)&buf[20];
199             char depth = buf[25];
200             prints("MODE: [");
201             printn(mode, 16);
202             prints("] ");
203             printd(xres);
204             prints("x");
205             printd(yres);
206             prints("x");
207             printd(depth);
208             prints("\r\n");
209         }
210         return mode;
211     }
212 }
213
214 long loadKernel(unsigned int dstseg) {
215     int i;
216     char pfat[SectorsPerFAT*BytesPerSector];
217     char proot[RootEntryCount*32];
218     read(getStackSeg(), (int)pfat, 1+ReservedSectorCount+
            SectorsPerFAT, SectorsPerFAT);
219     read(getStackSeg(), (int)proot, 1+ReservedSectorCount
            +SectorsPerFAT*NumberOfFAT, (RootEntryCount*32)
            /512);
220     for(i=0; i<RootEntryCount*32; i+=32) {
221         int j, flag;
222         if(proot[i]=='\0') continue;
223         flag = 0;
224         for(j=0; j<11; ++j) {
225             if(proot[i+j]!="KERNEL IMG"[j]) {
226                 flag = 1;
227                 break;
228             }
229         }
230         if(flag==0) {
231             short cluster = *(short*)&proot[i+26];
232             long size = *(long*)&proot[i+28];
233             long copyed = 0;
234             unsigned int dstoff;
235             prints("READ ");
236             printh(size>>16);
237             prints(" ");
238             printh((int)size);
239             prints(" BYTES FROM ");
240             printd(cluster);
241             prints("\r\n");
242             dstoff = 0;
243             while(size>copyed) {
244                 unsigned short t;
245                 prints("\rREAD #");
246                 printd(cluster);
247                 prints("...");
248                 read(dstseg, dstoff, 1+ReservedSectorCount+
                        SectorsPerFAT*NumberOfFAT+(
                        RootEntryCount*32)/512+cluster-2, 1);
249                 t = *(unsigned short*)(pfat+cluster*12/8);
250                 if(cluster&1) {
251                     t >>= 4;
252                 } else {
253                     t &= 0x0FFF;
254                 }
255                 cluster = t;
256                 copyed += 512;
257                 if(dstoff>=0xFFFF-512) {
258                     dstseg += 0x1000;
259                     dstoff = 0;
260                 } else {
261                     dstoff += 512;
262                 }
263             }
264             prints("\rLOADING KERNEL IMAGE
                    COMPLETED\r\n");
265             return size;
266         }
267     }
268     prints("KERNEL IMAGE NOT FOUND");
269     freeze();
270 }
271
272 void clearCursor() {
273     /* カーソルを消去 */
274     _asm_c(
275         "mov ah, 0x01 \n"
276         "mov ch, 0x20 \n"
277         "int 0x10  \n"
278         );
279 }
280
```

```
281 void clearA20() {
282     _asm_c("int 0x15", 0x2401);
283     prints("A20 LINE ENABLED\r\n");
284 }
285
286 int getMemorySize() {
287     int l = _asm_c("int 0x15 \n mov ax, cx", 0xE801
          , 0, 0, 0);
288     int h = _asm_c("int 0x15 \n mov ax, dx", 0xE801
          , 0, 0, 0);
289     return h + (l >> 6);
290 }
291
292 int getVbeInfo(int seg, void* p) {
293     int ret = _asm_c(
294         " push di   \n"
295         " mov es, bx  \n"
296         " mov di, ax  \n"
297         " mov ax, 0x4F00 \n"
298         " int 0x10  \n"
299         " pop di   \n"
300         , p, seg);
301     return ret==0x004F;
302 }
303
304 int getVbeModeInfo(int mode, int seg, void* p) {
305     int ret = _asm_c(
306         " push di   \n"
307         " mov es, bx  \n"
308         " mov di, ax  \n"
309         " mov ax, 0x4F01 \n"
310         " int 0x10  \n"
311         " pop di   \n"
312         , p, seg, mode);
313     return ret==0x004F;
314 }
315
316 int switchVbeMode(int mode) {
317     int ret = _asm_c(
318         " or bx, 0x4000 \n"
319         " int 0x10  \n"
320         , 0x4F02, mode);
321     return ret==0x004F;
322 }
```

テキスト A.5　3rdboot.asm

```
1 [ORG 0x00001000]
2
3 ;-----------------------------------------------------------
4 ; プロテクトモードへ移行
5 ;-----------------------------------------------------------
6 [BITS 16]
7 enter_protectmode:
8     mov si, ax   ; AX = segment of kernel.img
9     mov ax, cs
10    mov ds, ax
11    mov es, ax
12    cli
13    lgdt [gdtr]
14    mov eax, cr0
15    or eax, 1
16    mov cr0, eax ; プロテクトモードへ
17    jmp near far_jump
18 far_jump:
19    jmp dword 0x08:on_protectmode
20
21 ;**********************************************************
22 ; 32 bit コード
23 ;**********************************************************
24
25 [bits 32]
26 on_protectmode:
27    mov ax, 0x10  ; ds & es selector
28    mov ds, ax    ; is 0x10
29    mov es, ax    ;
30    mov ax, 0x18  ; ss selector
31    mov ss, ax    ; is 0x18
32    mov ax, 0x00  ; null gdt
33    mov fs, ax
34    mov gs, ax
35    lidt [idtr+0x10000]  ; IDT の設定
36    mov esp, 1024*1024*4 ; sp is 4MB
37    ; カーネルコードのコピー
38    and esi, 0x0000FFFF  ; SI = segment of kernel.img
39    shl esi, 4
40    mov edi, 0x00101000
41    mov ecx, [0x06F8]
42    cld
43    rep movsb
44    ; 1ch
45    mov eax, [lastmsg]
46    mov [0xb8000], eax
47    ;------------------------------------------------
48    ; CR4 の操作をここでする
49    ; これによってSSE/SSE2 が許可される
50    ; 未対応のCPU でこれするとどうなるかは不明
51    mov eax, cr4
52    or eax, 0x200
53    mov cr4, eax
54    ;------------------------------------------------
55    ; カーネルにジャンプ
56 ; jmp stop
57    lss esp, [stack]
58    jmp 0x8:0x101000
59
60 stack dd 0x00400000
61    dw 0x0018
62
63 ;-------------------------------------------------------
64 ; 処理の停止
65 ;-------------------------------------------------------
66 stop:
67    cli
68 forever:
69    hlt
70    jmp short forever
71
72 ;-------------------------------------------------------
73 ; GDT definitions
74 ;-------------------------------------------------------
75 gdtr:
76    dw gdt_end-gdt0-1  ; gdt limit
77    dd gdt0
78
79 gdt0:   ; segment 00
80    dw 0   ; segment limitL
81    dw 0   ; segment baseL
82    db 0   ; segment baseM
83    db 0   ; segment type
84    db 0   ; segment limitH, etc.
85    db 0   ; segment baseH
86 gdt08:   ; segment 08(code segment)
87    dw 0xffff  ; segment limitL
88    dw 0x0000  ; segment baseL
89    db 0   ; segment baseM
90    db 0x9a   ; Type Code
91    db 0xdf   ; segment limitH, etc.
92    db 0   ; segment baseH
93 gdt10:   ; segment 10(data segment)
94    dw 0xffff  ; segment limitL
95    dw 0x0000  ; segment baseL
96    db 0   ; segment baseM
97    db 0x92   ; Type Data
98    db 0xdf   ; segment limitH, etc.
99    db 0   ; segment baseH
100 gdt18:   ; segment 18(stack segment)
101    dw 0x20   ; segment limitL (2MB)
102    dw 0x0000  ; segment baseL
103    db 0   ; segment baseM
104    db 0x96   ; Type Stack
105    db 0xC0   ; segment limitH, etc.
106    db 0   ; segment baseH
107
108 gdt_end:   ; end of gdt
109
110 ;-------------------------------------------------------
111 ; IDT definitions
112 ;-------------------------------------------------------
113 idtr:
114    dw idt_end-idt_begin-1 ; IDT のサイズ
115    dd 0x10000+idt_begin ; IDT のアドレス
116
117 %macro makeidt 1
118    dw ig%1
119    dw 0x08
120    db 0
121    db 2+4+8+0+0x80
122    dw 0x0001
```

```
123 %endmacro
124
125 idt_begin:
126 %assign i 0
127 %rep 2
128     makeidt i
129 %assign i i+1
130 %endrep
131 idt_end:
132
133 ;-------------------------------------------------------
134 ; Interrupt Handlers
135 ;-------------------------------------------------------
136
137 %macro makeig 1
138 ig%1:
139     mov ebx, eax
140     mov eax, %1
141     sti
142 igb%1:
143     ;hlt
144     jmp igb%1
145     iret
146 %endmacro
147
148 %assign i 0
149 %rep 2
150     makeig i
151 %assign i i+1
152 %endrep
153
154 ;-------------------------------------------------------
155 ; メッセージ
156 ;-------------------------------------------------------
157
158 lastmsg  db '>', 0x0F
159        db '-', 0x0F
160
161 times 512 - ($-$$) db 0
```

テキスト A.6 makefile

```
1 #---------------------------------------------------
2 # CLIOS makefile
3 #---------------------------------------------------
4
5 PATH = $(PATH);C:\osdev\tool\lsic\BIN
6
7 #---------------------------------------------------
8 # 生成規則
9 #---------------------------------------------------
10
11 all: bootloader.img
12
13 bootloader.img: makefile 1stboot.img 2ndboot.img 3rdboot.
      bin
14     copy /b /y bpb.bin + 1stboot.img + 3rdboot.bin + 2
          ndboot.img bootloader.img
15
16 1stboot.img: 1stboot.obj
17     lld -M -T 7C60 -TDATA 7DC0 -o 1stboot.com 1stboot.obj
18     copy /y 1stboot.com 1stboot.img > nul
19     fixsize 0x1A0 1stboot.img 55 AA
20
21 2ndboot.img: 2ndboot.obj
22     lld -M -T 1200 -TDATA 1C00 -o 2ndboot.com 2ndboot.obj
23     copy /y 2ndboot.com 2ndboot.img > nul
24     fixsize 0xC00 2ndboot.img
25
26 #---------------------------------------------------
27 # suffix rules
28 #---------------------------------------------------
29
30 .c.obj:
31     lcc -O -o $@ -c $<
32
33 #---------------------------------------------------
34 # クリア
35 #---------------------------------------------------
36
37 clean:
38     -del /f *.img > nul
39     -del /f *.com > nul
40     -del /f *.obj > nul
```

A.2 kernel

kernel はレガシーカーネルをビルドするためのプロジェクトである。以降にはカーネルのエントリポイントを定義するソースファイルと、ビルドツールのオプションを示す。

テキスト A.7 _entrypoint.cpp

```
1 #define busyloop for(;;) { __asm { nop } __asm { nop } }
2
3
4 extern void kernelmain();
5
6 extern "C" extern void __stdcall entrypoint(int argc,
      char* argv[], char* envp[]);
7
8
9 #pragma alloc_text(".entry", entrypoint)
10
11 #pragma optimize("", off)
12
13 extern "C" extern __declspec(naked) void __stdcall
      entrypoint(int argc, char* argv[], char* envp[]) {
14     __asm jmp kernelmain;
15 }
16
17 #pragma optimize("", on)
```

テキスト A.8 コンパイラオプション

```
1 /Ox /Og /Ob2 /Oi /Os /Oy /G6 /I "D:\boost\include\
boost-1_32" /D "NDEBUG" /D "_USRDLL" /D "
_WINDLL" /D "_ATL_MIN_CRT" /GF /Gm /ML /Zc:wchar_t
/Zc:forScope /FAs /Fa"Release/" /Fo"Release/" /Fd"
Release/vc70.pdb" /W3 /nologo /c /Zi /TP
```

テキスト A.9 リンカオプション

```
1 /VERBOSE:LIB /OUT:"kernel.dll" /INCREMENTAL:NO /NOLOGO
 /DLL /MAP /MAPINFO:EXPORTS /OPT:REF /OPT:ICF /ENTRY:"
entrypoint" /NOENTRY /RELEASE /BASE:"0x100000" /
NOASSEMBLY /IMPLIB:"Release/kernel.lib" /MERGE:".
text=.entry" /MACHINE:X86 /FIXED
```

A.3 csbridge

csbridge はカーネルブリッジ（3.5 節参照）を担当するプログラムを作成するプロジェクトである。次に IKernel の定義を示す。

テキスト A.10 interop.h

```
1 #pragma once
2
3
4 struct __declspec(novtable) IKernel {
5     virtual bool __stdcall NotAsSystem() = 0;
6     virtual void* __stdcall alloc(unsigned int size) = 0;
7     virtual void __stdcall free(void* p) = 0;
8     virtual void* __stdcall ReadLine() = 0;
9     virtual void __stdcall Write(wchar_t ch) = 0;
10    virtual void __stdcall SetDebugMode(bool enabled) =
          0;
11    virtual void __stdcall LoadAssembly(void* assembly,
          void* p, int size) = 0;
12    virtual void* __stdcall GetExecutingAssembly(int
          depth) = 0;
```

```
13     virtual void __stdcall ReloadMethodCode(void* handle,
           int rowIndex, const unsigned char* p) = 0;
14     virtual void* __stdcall GetTypeFromHandle(void*
          handle) = 0;
15     virtual void* __stdcall CreateInstance(void* handle)
          = 0;
16     virtual void* __stdcall CloneInstance(void* obj) = 0;
17     virtual void* __stdcall CreateString(unsigned int
          length) = 0;
18     virtual void* __stdcall CreateArray(void* handle,
          unsigned int length) = 0;
19     virtual void* __stdcall NotifyLoadingType(const
          wchar_t* asmname, int len, int typerid, void*
          obj) = 0;
20     virtual const void* __stdcall GenerateProxyCode(const
           wchar_t* asmname, int len, int methodrid) = 0;
21 };
```

A.4　fdimage

fdimage は bootldr および kernel の成果物から、フロッピーディスクイメージを得るためのプロジェクトである。以降に fdimage のすべてのソースファイルを示す。

テキスト A.11　rootdirectory.asm

```
1 %include "define.mac"
2
3 %rep RootEntryCount
4     db 0x00  ; Indicates that this is free.
5     db ' ' ; Dir Name
6     db 0  ; Attributes
7     db 0  ; for WindowsNT
8     db 0  ; Creation Time Tenth
9     dw 0  ; Creation Time
10    dw 0  ; Creation Date
11    dw 0  ; Last Access Date
12    dw 0  ; First Cluster (high)
13    dw 0  ; Write Time
14    dw 0  ; Write Date
15    dw 0  ; First Cluster (low)
16    dd 0  ; File Size
17 %endrep
```

テキスト A.12　fat.asm

```
1 %include "define.mac"
2
3 %rep 2
4     times SectorsPerFAT*BytesPerSector db 0
5 %endrep
```

テキスト A.13　makefile

```
1 #--------------------------------------------------
2 # CLIOS makefile
3 #--------------------------------------------------
4
5 cygwin  = C:\cygwin\bin
6 as  = $(cygwin)\nasm.exe
7 disas  = $(cygwin)\ndisasm.exe
8 strip  = $(cygwin)\strip.exe
9 cat  = $(cygwin)\cat.exe
10
11 # for nasm
12 ASFLAGS  = -f bin
13
14 #--------------------------------------------------
15 # suffix definitions
16 #--------------------------------------------------
17
18 .SUFFIXES: .asm .bin .img
19
20 #--------------------------------------------------
21 # コンパイルするファイルを定義
22 #--------------------------------------------------
23
24 target = fdimage.img
25 kernel = kernel.img
26 knlasm = kernel.asm
27 bin = bootloader.bin bootkernel.bin fat.bin rootdirectory
      .bin
28 drv = ..\driver\Release\driver.dll
29 asm = $(bin:.bin=.asm)
30
31 #--------------------------------------------------
32 # 生成規則
33 #--------------------------------------------------
34
35 all: $(target)
36
37 $(target): $(bin) $(kernel) $(drv) makefile ..\bootloader
      \bootloader.img
38     -vfd.bat close
39     copy /b /y ..\bootloader\bootloader.img + fat.bin +
          rootdirectory.bin $(target)
40     fixsize.exe 1474560 $(target)
41     vfd.bat open fdimage.img /1.44 /w
42     copy $(kernel) b: > nul
43     copy $(drv) b: > nul
44     attrib +h +s +r b:\$(kernel)
45     $(cygwin)\sleep 1s
46     vfd.bat close
47
48 $(kernel): ..\kernel\$(kernel:.img=.dll) makefile
49     $(strip) -O binary ..\kernel\$(kernel:.img=.dll) -o
          $@
50     $(disas) -b 32 $@ > $(@:.img=.asm)
51
52 $(bin): $(@:.bin=.asm)
53
54 #--------------------------------------------------
55 # suffix rules
56 #--------------------------------------------------
57
58 .asm.bin:
59     $(as) $(ASFLAGS) -o $@ $<
60
61 #--------------------------------------------------
62 # クリア
63 #--------------------------------------------------
64
65 clean:
66     -vfd.bat close
67     -del $(bin) > nul
68     -del $(kernel) > nul
69     -del $(target) > nul
```

A.5　cdimage

cdimage は fdimage の成果物を利用して起動可能 CD-ROM イメージを得るプロジェクトである。CD-ROM イメージにはマネージカーネル、mscorlib、freetype などが納められる。

テキスト A.14　makefile

```
1 #--------------------------------------------------
2 # CooS makefile
3 #--------------------------------------------------
4
5 target = $(TargetDir)\..\coos.iso
6 fdimg = ..\fdimage\fdimage.img
7 brg = ..\csbridge\Release\csbridge.dll
8 krl = ..\cskorlib\Release\cskorlib.dll
9 cor = ..\cscorlib\bin\Release\cscorlib.dll
10 grx = ..\csgraphics\bin\Release\csgraphics.dll
11 asm = ..\ia32assembler\bin\Release\ia32assembler.dll
12 ft2 = ..\freetype2\Release\freetype2.dll
13 drv = ..\driver\Release\driver.dll
14 app = ..\application\bin\Release\application.exe
```

```
15 doc = ..\cdimage\README.html
16 lic = ..\cdimage\LICENSE.html
17
18 #---------------------------------------------------
19 # 生成規則
20 #---------------------------------------------------
21
22 $(target): $(fdimg) $(brg) $(krl) $(cor) $(grx) $(asm) $(
       ft2) $(drv) $(app) $(doc) $(lic) makefile
23     -mkdir $(TargetDir)
24     copy /y $(fdimg) $(TargetDir)\bootldr.img > nul
25     copy /y mscorlib.dll $(TargetDir) > nul
26     copy /y System.dll $(TargetDir) > nul
27     copy /y System.Drawing.dll $(TargetDir) > nul
28     copy /y $(brg) $(TargetDir) > nul
29     copy /y $(krl) $(TargetDir) > nul
30     copy /y $(cor) $(TargetDir) > nul
31     copy /y $(grx) $(TargetDir) > nul
32     copy /y $(asm) $(TargetDir) > nul
33     copy /y $(ft2) $(TargetDir) > nul
34     copy /y $(drv) $(TargetDir) > nul
35     copy /y $(app) $(TargetDir) > nul
36     copy /y $(doc) $(TargetDir) > nul
37     copy /y $(lic) $(TargetDir) > nul
38     xcopy /d /e /i /y ..\resource $(TargetDir) > nul
39     xcopy /d /e /i /y licenses $(TargetDir)\licenses >
           nul
40     D:\Miscellaneous\cdrtools-1.11a12-win32-bin\mkisofs.
           exe -verbose -iso-level 3 -eltorito-boot bootldr
           .img -o $(target) $(TargetDir)
41
42 #---------------------------------------------------
43 # クリア
44 #---------------------------------------------------
45
46 clean:
47     -del $(target)
48     -rmdir /S /Q $(TargetDir)
49
50 all: clean $(target)
```

参考文献

[1] Bellard, F.: QEMU, http://fabrice.bellard.free.fr/qemu/.

[2] Chase, D.: GC FAQ, http://www.iecc.com/gclist/GC-faq.html (2006).

[3] Golm, M., Felser, M., Wawersich, C. and Kleinoder, J.: The JX Operating System, *In Proceedings of the USENIX Annual Technical Conference*, pp. 45–58 (2002).

[4] Hunt, G. and Larus, J.: An Overview of the Singularity Project, http://research.microsoft.com/os/singularity/ (2005).

[5] Intel Corporation: IA-32 Intel Architecture Software Developer's Manual, http://www.intel.com/design/pentium4/manuals/index_new.htm (2005).

[6] Intel Corporation: Extensible Firmware Interface (EFI), http://www.intel.com/technology/efi/ (2006).

[7] Intel Corporation: Intel Pentium 4 Processor Family Technical Documents, http://developer.intel.com/design/Pentium4/documentation.htm (2006).

[8] Intel Corporation: 日本語技術資料, http://www.intel.co.jp/jp/developer/download/index.htm (2006).

[9] JNode.org: JNode Operating System, http://www.jnode.com/ (2006).

[10] Matsumoto, M., Nishimura, T. and Saito, M.: Mersenne Twister: A random number generator, http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html (2005).

[11] Microsoft Corporation: *FAT: General Overview of On-Disk Format*, Microsoft Corporation (2000).

[12] Microsoft Corporation: Microsoft Portable Executable and Common Object File Format Specification, http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx (2004).

[13] Mono Project: Mono, http://www.mono-project.net/ (2005).

[14] OS-Wiki: OS Wiki, http://community.osdev.info/ (2006).

[15] Phoenix Technologies: 'El Torito' Bootable CD-ROM Format Specification, http://www.phoenix.com/NR/rdonlyres/98D3219C-9CC9-4DF5-B496-A286D893E36A/0/specscdrom.pdf (1995).

[16] Red Hat: Cygwin, http://www.cygwin.com/.

[17] Roeder, L.: .NET Reflector, http://www.aisto.com/roeder/dotnet/.

[18] Rogerson, D.: Inside COM—Microsoft ' s Component Object Model, 株式会社アスキー (1997).

[19] Standard ECMA-119: Volume and File Structure of CDROM for Information Interchange, http://www.ecma-international.org/ (1998).

[20] Standard ECMA-335: Common Language Infrastructure (CLI), http://www.ecma-international.org/publications/standards/Ecma-335.htm (2005).

[21] T13 Technical Committee : AT Attachment with Packet Interface (ATA/ATAPI), http://www.t13.org/ (2001).

[22] The Bochs Project: bochs: The Open Source IA-32 Emulation Project, http://bochs.

sourceforge.net/.

[23] The FreeType Project: FreeType, http://freetype.sourceforge.net/ (2005).

[24] The Netwide Assembler Project: The Netwide Assembler, http://nasm.sourceforge.net/.

[25] VESA: The Vesa BIOS extension (VBE) 2.0, http://www.vesa.org/public/VBE.

[26] VMware Inc.: VMware Workstation, http://www.vmware.com/.

[27] Wikipedia: PC/AT, http://ja.wikipedia.org/wiki/PC/AT (2006).

[28] エル・エス・アイジャパン株式会社：LSI C-86, http://www.lsi-j.co.jp/product/c86/index.html (2006).

[29] エル・エス・アイジャパン株式会社：LSI C-86 v3.30c 試食版，http://www.vector.co.jp/soft/maker/lsi/se001169.html (2006).

[30] 蒲地輝尚：はじめて読む 486—32 ビットコンピュータをやさしく語る，株式会社アスキー (1994).

[31] 桑野雅彦：パソコンのレガシィ I/O 活用大全，ハードウェアデザインシリーズ，Vol. 12, No. 1, CQ 出版株式会社，1st edition (2000).

[32] 白崎博生：Linux のブートプロセスをみる，株式会社アスキー (2004).

[33] 不明：QEMU on Windows, http://www.h7.dion.ne.jp/~qemu-win/index-ja.html.

[34] 墨染さくら：オリジナル OS を作ろう！，C MAGAZINE, Vol. 6, No. 7 (2004). p.20-51.