第 1 章 はじめに

1.1 背景

ネイティブコードからなるユーザーモードアプリケーションの異なる OS、アーキテクチャ間での可搬的な利用が困難な理由として、コンパイル済みバイナリの機械語命令列と OS のシステムコールが挙げられる。このうち OS のシステムコールについては、機械語命令に互換性のあるアーキテクチャ上であっても、OS が提供する機能に大きく依存するため、ファイルシステムなどの周辺機能も含めてハードウェア機構全体を仮想化するフルシステムエミュレーションによって解決する手法が試みられている。しかし、エミュレーションする OS が利用する特権命令や入出力デバイスを含めてエミュレーションするこの方式では、アプリケーションが必要とする環境を提供するために大きな計算機資源を必要とする。

他方、ユーザーモードにおける機械語命令のエミュレーションのみを行い、システムコールは、エミュレータの下で動作している実機の OS に対し処理を依頼するユーザーモードエミュレータは、エミュレーションする対象のハードウェアが少ないことによる必要な計算機資源の低減だけでなく、エミュレータの動作環境に存在する実機のハードウェア資源をアプリケーションから直接利用出来るという点において有用である。しかし、この方式はエミュレータの下で動作している OS のシステムコールライブラリに大きく依存し、異種 OS 間における可搬的な利用を妨げている。このため、既存のユーザーモードエミュレータの多くは、エミュレーションする対象のプログラムが本来動作する環境と同一の OS 上でしか動作しない。

また、既存のユーザーモードエミュレータの多くは、エミュレータ自体がネイティブコードで構成されるアプリケーションとして実現されており、エミュレータ自身の可搬性を妨げている。結果、多くのユーザーモードエミュレータは、エミュレーションする対象のアプリケーションを利用するユーザーに対し、単に新しいプラットフォームを提案しているに過ぎない。

これらのことから、ネイティブコードからなる汎用的なユーザーモードアプリケーションに対し、異種 OS・アーキテクチャ間での可搬性を付与し、エミュレータ自身も可搬性を持つユーザーモードエミュレータはまだ存在していない。

本研究では GUI、マルチスレッド、共有ライブラリの動的リンクといった機能を利用する今日の実用アプリケーションが動作するユーザーモードエミュレータの実現を目指した。

1.2 目的と本論文の構成

本研究では、ユーザーモードエミュレーションにおける異種 OS 間での可搬性問題を解決し、今日の実用的なユーザーモードアプリケーションを異種 OS・アーキテクチャ間で利用可能なユーザーモードエミュレーションを実現するため、JavaVM 上で動作する x86 ユーザーモードエミュレータの実現を目指した。具体的には、GUI や共有ライブラリの動的リンク、マルチスレッドといった機能を利用する、今日の一般的な x86/Linux 用のアプリケーションが、JavaVM が動作する様々な環境上で動作することを目標とした。

本論文では、まず既存のエミュレータを構成する機械語命令エミュレーションの方式や、エミ

ュレータ上でアプリケーションが利用するシステムコールの取り扱いについて調べた。これを第 2 章で述べる。続いて、これら既存のエミュレータの中でも特にユーザーモードエミュレータが抱える問題を定義し、本研究において解決すべき事柄を第 3 章で述べる。第 4, 5, 6 章では本研究において実装するユーザーモードエミュレータの設計方針について述べる。第 7 章では実際にエミュレータを実装した結果、実現した機能やテスト項目について述べる。第 8 章では本エミュレータの性能に基づく定量的な評価と、どのようなアプリケーションが本エミュレータ上で動作するかといった定性的な評価について述べる。第 9 章では本研究の成果と今後の課題についてまとめた。

第 2 章　既存の研究・技術

2.1 Full System Emulation

Full System Emulation は CPU エミュレーションのみならず、ディスクやネットワークといった周辺 I/O も含めた、計算機システム全体をエミュレーションする方式であり、QEMU[1]のユーザーマニュアル内で用語の定義がなされている。フルシステムエミュレーションは、ユーザーモードアプリケーションの可搬的利用を目的とした場合に、エミュレーションに要する計算機資源が後述するユーザーモードエミュレーションに対し過大であることから、本研究ではユーザーモードエミュレーションを目指した。一方で、エミュレータの実装や最適化において参考とする部分が多いため、以下に代表的なものを記す。

Bochs[2]は C++で書かれた PC/AT 互換機のエミュレータである。Bochs は CPU の命令セットを全てソフトウェアでエミュレーションする。このため、Bochs 自体の再コンパイルを行えば、複数の異なるハードウェア上でエミュレーションを行うことが可能である。Bochs は命令列の逐次エミュレーションを行うため、動的な命令変換等を施した場合に比べ性能は低いが、一方で再コンパイルに伴う特定アーキテクチャに対する依存性を下げている。このため、エミュレータ自体の可搬性は後述する JPC ほどないが、再コンパイルによる移植の容易性は JIT を用いるエミュレータよりも高い。

JPC[3]は Java で書かれた PC/AT 互換機のエミュレータである。JPC は JPC 自体の再コンパイルを行わずに、JavaVM が動作する様々な環境上で実行することが可能である。また、Java アプレットとしてブラウザ上で実行することも可能である。

JPC の機械語命令エミュレーションは、x86 命令をより小さな JPC 独自の命令の組み合わせからなるマイクロコードとして扱う。これによってオペランドの表現形式を簡略化し、命令エミュレーションの最適化を容易にしている。また、エミュレーションする対象の命令列をもとに動的な Java バイトコードの生成を行う。このような最適化手法は、ユーザーモードエミュレータの実装においても有用であると推測される。

Xen[4]は準仮想化を特徴とするハイパバイザ型の仮想マシンであり、実機のハードウェア資源を利用することでエミュレーションのオーバーヘッドを低減している。ただし、完全仮想化の実行モードでは、x86 の特権命令を非特権モード上でエミュレーションする機能も有している。

準仮想化と完全仮想化のいずれのモードで Xen を利用した場合も、非特権命令のエミュレーションは行われず、実機の命令セットを使用する。このため Xen を実行可能な環境は x86 アーキテクチャ上に限定される。

Xen は x86 アーキテクチャに完全に依存する設計方針を採っており、エミュレータの可搬性では JPC と対極にあるが、フルシステムエミュレーションに要するオーバーヘッドをある程度削減しているため、比較のために記述した。

2.2 User Mode Emulation

User Mode Emulation は CPU の命令セットのうち、ユーザーモードのプログラムが使用す

る非特権命令のエミュレーションのみを行い、システムコールはエミュレータの外部で動作する実機の OS のシステムコールライブラリに処理を依頼する方式である。

QEMU[1]は様々なアーキテクチャのエミュレーションを行うことを目的とした C++言語で実装された汎用エミュレータである。ディスク装置などを含めた計算機資源全体のエミュレーションを行うフルシステムエミュレーションと、CPU の命令セットのエミュレーションのみを行うユーザーモードエミュレーションの 2 種類の動作モードを持つが、ここではユーザーモードエミュレーションについて述べる。

QEMU のユーザーモードエミュレーション機能は、幾つかのアーキテクチャ上で動作する Linux 用のユーザーモードプログラムを、複数の異種アーキテクチャ上で動作する Linux 環境において相互に利用可能とするものである。したがって、QEMU のユーザーモードエミュレーション機能は異種アーキテクチャ間での可搬性を提供しても、異種 OS 間での可搬性は提供していない。

QEMU の演算性能は実機に対し 3 倍から 50 倍程度の性能低下で実現している。QEMU は、エミュレーションする対象のプログラムの命令列を、基本ブロックと呼ばれる単位で分割し、各基本ブロックをエミュレータの実行環境の命令列に動的に変換する。基本ブロックを構成する命令列中には分岐命令は一つしかなく、また分岐命令は命令列の終端にのみ存在する。したがって、基本ブロックの入り口と出口はそれぞれ一つしか存在しない。QEMU に実装された命令列の動的変換によるエミュレーション手法は、後述する NestedVM の静的変換手法と対照的である。

図 2.2.1　QEMU のエミュレーション方式

NestedVM[5]は、MIPS R2000 用バイナリを対象とする、Java で書かれたユーザーモードエミュレータである。ここでいう MIPS R2000 用バイナリは、gcc のクロスコンパイルオプション”mips-unknown-elf”が適用され、NestedVM 独自のシステムコール番号に基づくシステムコールを使用し、NestedVM が定義するエントリポイントから開始される命令列からなる、静的リンクされた ELF バイナリである。したがって、MIPS/Linux 等の汎用 OS 上で一般的に使用されるアプリケーションバイナリは直接実行できない。既存のアプリケーションを NestedVM 上で利用する場合には、この特異な環境をターゲットとして再コンパイルする必要がある。

NestedVM は POSIX 互換システムコールの一部を Java 言語のランタイムライブラリの組み

合わせでエミュレーションする機能を有しており、gcc や TeX といったコンソールベースのアプリケーションを、pure-Java な環境上で実行可能である。

性能面では実機に対し、演算性能で 8 倍程度、I/O 性能で 2 倍程度の性能低下を実現している。NestedVM はエミュレーションする対象の MIPS 命令列を分解し、エミュレーションするプログラムの関数単位で、Java 言語の switch-case 文に静的に変換する。各 case 節のラベルは、エミュレーションする対象プログラム内でのアドレスを示し、case 節内には当該関数をさらに分解した命令単位のエミュレーションコードがインラインで展開される。プログラムの命令列を事前に静的変換するには、メモリ上に配置される各命令のアライメントが固定されていることが前提となる。これは、間接分岐命令による分岐先アドレスが事前に予測できないことによる。間接分岐命令による分岐先アドレスのアライメントが確実に予測できれば、予め全ての命令列を静的に解析することが可能である。このため、命令列の静的解析による最適化は、MIPS アーキテクチャのエミュレーションに特化し、かつ静的リンクされた ELF バイナリしかサポートしない NestedVM に特有の機能である。

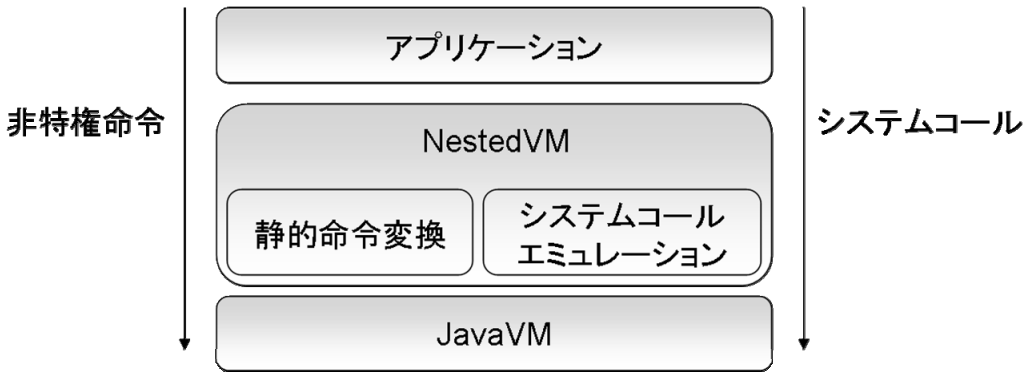

図 2.2.2　NestedVM のエミュレーション方式

Emulin[6]は x86/Linux 用の ELF バイナリを対象とする、Java で書かれたユーザーモードエミュレータである。Emulin 上におけるシステムコールは、JNI を用いてエミュレータの下で動作する OS またはライブラリの機能を呼び出して実現している。したがって、Emulin が動作する環境は、POSIX 互換システムコールを有する OS 上に限定される。

Emulin がサポートする命令セットは、x86 の非特権命令のうち一部に限られており、FPU 命令などはサポートされない。またロード可能な ELF バイナリも静的リンクされたものに限られる。

Emulin は、x86 命令列を 1 命令ずつ逐次解釈し実行する。静的・動的を問わず x86 命令列の Java バイトコードへの変換を試みない。Emlin は cat や wc といった Linux 用コマンドプログラムの実行をサポートしている。Emulin の性能を示す資料は公開されておらず、また筆者は今日の x86/Linux(カーネル 2.2.25)用の ELF バイナリを Emulin 上で使用することはできなかった。

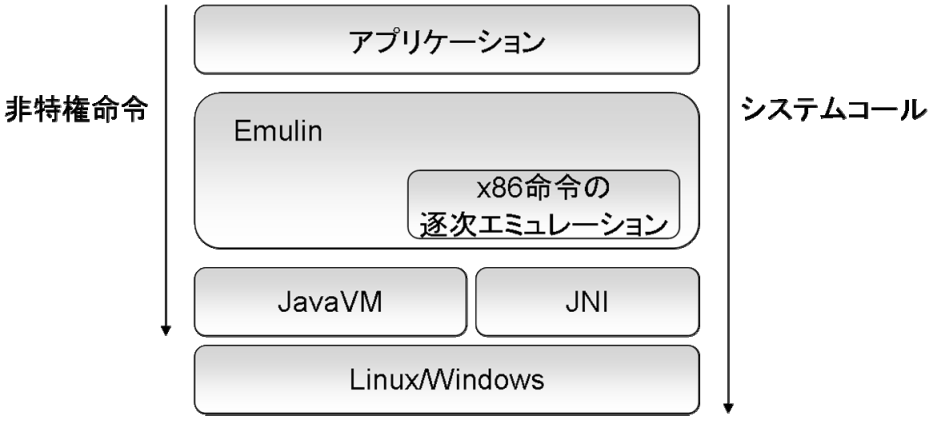

図 2.2.3　Emulin のエミュレーション方式

EM86[7]は Linux/Alpha 環境上で、Linux/x86 アプリケーションを実行するためのユーザーモードエミュレータである。QEMU のユーザーモードエミュレーション機能と同様に、EM86 の動作環境も Linux 上に限定される。また、Alpha アーキテクチャの命令セットに特化した動的コード変換による最適化を行うため、Alpha アーキテクチャ上でしか動作しない。

EM86 は、x86/Linux 用の Netscape Navigator や Adobe Acrobat Reader 等をエミュレーション可能である。また、共有ライブラリの動的リンクにも対応している。

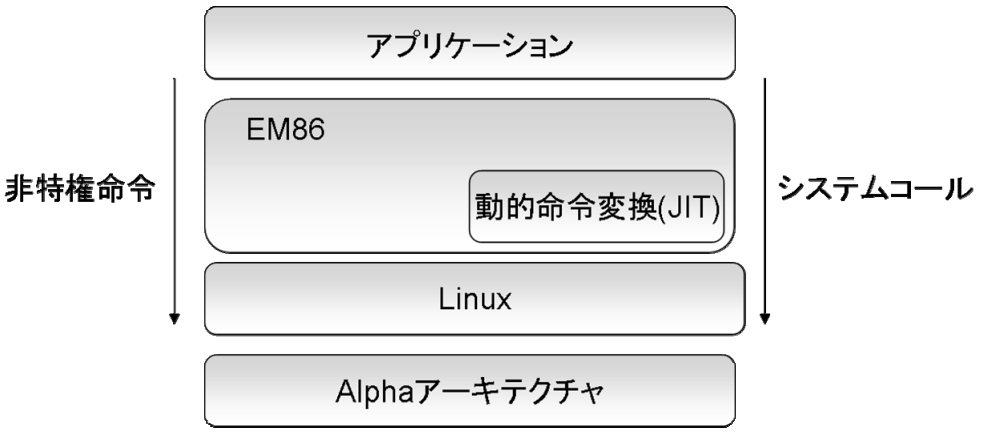

図 2.2.4　EM86 のエミュレーション方式

VEELS[8]は PowerPC と ARM 用の Linux プログラムを、JavaVM 上でエミュレーション可能なエミュレータである。エミュレーションする対象のアーキテクチャに対する簡易な記述を元に、様々なプラットフォームを対象とするエミュレーションを行うことを目標としている。

VEELS は Java 言語のみで記述されており、Linux 用のプログラムを Java アプレットとして Web ブラウザ上で実行することが可能である。

VEELS の実機に対する性能比率は示されていないが、QEMU に対しては gzip や bzip2 でほぼ同じ性能を実現し、gcc では 20 倍程度の性能低下で実現している。VEELS は JPC と同じく、エミュレーションする対象の命令列を基本ブロックと呼ばれる単位に分割し、動的に Java バイ

トコードに変換している。各基本ブロックは単一の Java クラスに動的に変換される。このため、エミュレーションする対象のプログラムが大きくなり、生成される Java クラスの個数が増えると性能が低下することが示されている。

VEELS の最終的な目標は、MMU をエミュレーションし VEELS 上でゲスト OS が動作する Full System Emulation の実現にあるが、現在は PowerPC と ARM 用にコンパイルされた Linux 用のユーザーモードプログラムのエミュレーションにとどまっている。このため、User Mode Emulation に分類した。VEELS 上でのシステムコールの利用がどの程度可能であるかは示されていない。また、VEELS 上で動作するプログラムに対し、どのようにシステムコール機能を提供するかも示されていない。しかし、PowerPC/Linux 用のほとんどのバイナリが動作すると VEELS の作者らは主張している。

2.3 既存技術の比較

フルシステムエミュレーションは I/O デバイス等も含めた計算機システム全体をエミュレーションするため、全般的にエミュレーションに必要な計算機資源が大きい。個々の技術については、JavaVM 上で動作しエミュレータ自身の可搬性を保証する JPC、再コンパイルによって異種アーキテクチャ間での可搬性を提供する Bochs、x86 ハードウェアに完全に依存する Xen の順にエミュレータが提供する可搬性が低くなる。一方、エミュレータの性能は概ねこの逆順である。

ユーザーモードエミュレーションについては、JavaVM 上で動作する NestedVM、VEELS、Emulin が提供する可搬性が高く、再コンパイルによるエミュレータの可搬性を提供する QEMU、Alpha アーキテクチャの命令セットに完全に依存する EM86 の順となる。また、QEMU と EM86 はアーキテクチャだけでなく、エミュレータの実行環境の OS にも依存する。

性能面では、命令列の静的解析を行う NestedVM が最も高く、次いで JIT 機能を有する QEMU と EM86、伝統的なエミュレーションを行う VEELS や Emulin の順となる。各エミュレータがアプリケーションに対し提供する機能としては、QEMU が共有ライブラリの動的リンクやマルチスレッド機能を提供し、その他のエミュレータは静的リンクの強制などの形で制約が存在する。

表 2.3.1 ユーザーモードエミュレータの比較

	アーキテクチャ可搬性	OS可搬性	性能向上のためのバイナリ変換	スレッド	共有ライブラリ
QEMU	△（再コンパイル）	×	JIT	○	○
NestedVM	○	○	静的変換	×	×
Emulin	○	△(JNI)	×	×	×
EM86	×	×	JIT	不明	○
VEELS	○	○	JIT	×	×

第 3 章　目標と位置付け

3.1 既存研究の問題点

1.2 節で述べた目的である、今日一般的に使用されているユーザーモードアプリケーションの異なるアーキテクチャ・OS 間での可搬的な利用とユーザーモードエミュレータ自身のバイナリレベルでの可搬性という観点から、筆者は 2.2 節で述べた既存のユーザーモードエミュレータがこの目的を充足していないと考えた。以下にその理由を示す。

ⅰ） エミュレーションするプログラムに対し異種 OS 間での可搬性を提供していない。

ⅱ） ユーザーモードエミュレータ自身に可搬性がない。または不足している。

ⅲ） エミュレーション可能なプログラムが限定されている。または特異である。

表 3.1.1 既存研究の問題点

	異種 OS 間での 可搬性	エミュレータ自身 の可搬性	エミュレーション可能 なプログラム
QEMU	×	×	○
NestedVM	○	○	×(専用バイナリ)
Emulin	○	△ (JNI)	×
EM86	×	×	△(スレッドは不明)
VEELS	○	○	×

ⅰ）は QEMU, EM86 が該当する。これらのエミュレータは、Linux 上でしか動作せず、エミュレーションする対象のプログラムに対し、異種アーキテクチャ間での可搬性は提供しても、異種 OS 間における可搬性を提供しない。これらのユーザーモードエミュレータは、エミュレーションする対象のプログラムが発行したシステムコールに手を加えず、透過的にエミュレータの下部で動作する実機の OS に渡している。このため、エミュレーションする対象のプログラムが Linux 独自のシステムコールを利用する場合、エミュレータの動作環境も Linux に限定されてしまう。今日の Linux 用プログラムの多くがリンクする glibc は、プログラムのロード時に Linux 独自のシステムコールを幾度も使用する。このため、アプリケーションの作成者が、ソースコードレベルでの可搬性を考慮して、POSIX 互換システムコールのみを用いてプログラムを書いても、生成された ELF バイナリは Linux 環境に大きく依存している。

また、各種システムコールに渡される定数値は、その意味するところは同じであっても、値は OS ごとに異なることが多い。例として、open()システムコールの引数である、O_CREATE などが挙げられる。このため、同一のシステムコールライブラリが実装された OS 間であっても、異種 OS 間での可搬性を提供するユーザーモードエミュレータは、シス

テムコールの引数を解析し、定数値の違いを吸収する必要がある。

ii）は QEMU, Emulin, EM86 が該当する。これらのエミュレータは、エミュレータの全部または一部がネイティブコードで構成され、エミュレータ自身にバイナリレベルでの可搬性がない。このため、エミュレータのユーザーに対し、単に一時的に利用可能な新しいプラットフォームを提案しているに過ぎない。したがって、アプリケーションを可搬的に利用可能なプラットフォームとしてのユーザーモードエミュレータという観点から機能が不足している。

QEMU と EM86 は特定のアーキテクチャの命令セットに特化した、動的なコード変換を行うため、移植も容易ではない。Emulin は大部分のコードが JavaVM 上で動作するため、ある程度この問題を緩和しているが、一方で Linux のコンソールを模した機能をネイティブコードで構成し、可搬性を下げている。更にユーザーは本コンソールを介してしか、エミュレーションするアプリケーションを操作できない。このため、Emulin 上で動作するプログラムと、Emulin の外部で動作するプログラムを、シェルのパイプ機能等を利用して連携させることは難しい。

iii）は NestedVM,VEEL,Emulin が該当する。これらのエミュレータが提供する機能は、今日一般的に利用されているアプリケーションをエミュレーションするという観点から不足している。

NestedVM は、MIPS 用の ELF バイナリをサポートするが、プログラムのエントリポイントもシステムコール番号も NestedVM 専用のものを使用する。これらの条件を満たす ELF バイナリを生成するには、基本的に NestedVM 専用のコンパイラと libc を用いるしか方法がない。このため、一般的な OS 上で動作する既存の MIPS 用プログラムのバイナリを、直接 NestedVM 上で実行することはできない。また、NestedVM は、共有ライブラリの動的リンクや、マルチスレッドといった機能を提供しない。

VEEL は PowerPC/Linux 用のプログラムを直接ロード可能であることが示されているものの、どのようなシステムコールをサポートするかは不明であり、またマルチスレッドや GUI、共有ライブラリの動的リンクをサポートするかも不明であったため、これに分類した。

Emulin は今日の x86/Linux 用 ELF バイナリがロードできず、また Emulin の作者が意図した機能限界として、共有ライブラリの動的リンク機能をサポートしないことが挙げられているため分類した。

3.2 本研究の目標

先に述べた既存研究の問題点を解決し、エミュレーションするアプリケーションが利用可能な機能とエミュレータ自身の可搬性を両立した、ユーザーモードエミュレータを実現するため、

本研究では次の点を目標として定めた。

1）x86/Linux 用のアプリケーションバイナリが直接動作すること
2）マルチスレッドや GUI、共有ライブラリといった機能をサポートすること
3）エミュレータ自身が JavaVM 上で動作し可搬性を持つこと
4）GUI プログラムが Java アプレットして Web ブラウザ上で動作すること

1）では今日最もよく使われているプロセッサアーキテクチャである x86 と、その上で動作する汎用 OS としての Linux に注目した。Linux には POSIX 規格に基づいた UNIX-OS 間で互換性のあるシステムコール以外にも sys_clone などの Linux 独自のシステムコールが存在する。ユーザーモードエミュレータではシステムコール機能の可搬性を保証するため、その実現上、システムコールの仕様がオープンであることが望ましい。また独自のシステムコールを持った Linux 用のプログラムを、他の OS 上で可搬的に利用することは、ソースコードレベルでの移植であっても容易ではないため、本研究の有用性を明確に示すことが出来ると考えた。

2）では一般的な x86/Linux 用アプリケーションの定義として、共有ライブラリを実行時にリンクし、GUI やマルチスレッドといった複雑なシステムコール機能を利用するプログラムを想定した。GIMP や OpenOffice といったアプリケーションがこれに該当する。

3）ではエミュレータ自身の可搬性を保証するため、エミュレータが JavaVM 上で動作することを想定した。既存研究では、NestedVM のようにシステムコールまでを含めて完全に JavaVM の上でサポートするものもあれば、Emulin のようにシステムコールは JNI で提供する種類のものもある。筆者は、Linux 用システムコールの JavaVM 上での提供は、サポートできるシステムコール数とトレードオフの関係にあると考え、システムコール機能は少なくとも Emulin のように JNI で提供できる状態を維持し、可能であれば一部を JavaVM 上で提供する方式が有用であると考えた。

4）では JavaVM 上で動作するユーザーモードエミュレータの利点の一つとして、既存の x86/Linux 用の GUI プログラムが、Web ブラウザ上で動作するアプレットとして配信可能であることを示した。Web アプリケーションを構成するための言語・ライブラリ・実行環境等は数多くあるが、古くからある既存のデスクトップアプリケーションに変更を加えることなく Web ブラウザ上で実行することは難しい。またデスクトップアプリケーションと Web アプリケーションを共通の開発言語・ライブラリで実現することは有用である。

3.3 本研究の有用性

本研究は以下の点において新規性または有用性を持つ。

１）汎用的かつ可搬性の高いユーザーモードエミュレーション技術の実現

GUI やマルチスレッド、共有ライブラリの動的リンクといった今日のユーザーモードアプリケーションが一般的に利用する機能を提供し、かつ異種 OS・アーキテクチャ間で動作し、エミュレータ自身も可搬性を持つユーザーモードエミュレータは今日まだ存在しない。

エミュレーションするアプリケーションに対し、様々なプラットフォーム上において、バイナリレベルで、長期的に利用可能であることを保証することは重要である。特に Linux 用のプログラムは、gcc に施された GNU 拡張や、POSIX に完全に準拠していないシステムコールライブラリによって、ソースコードレベルであっても、異種環境間において可搬性を保証することが困難になりつつある。結果、今日の Linux アプリケーションの多くが、コンパイラや、glibc のバージョンに大きく依存する傾向にある。Java や LLVM のように、中間コードと VM によってこうした問題を解決する立場もあるが、実装言語やライブラリを新しく置き換えるこれらの手法は、既存のアプリケーションに直接適用することは難しい。

また、エミュレータの動作環境を増やし可搬性を高めるため、QEMU のように JIT コードの生成作業を簡略化することを目的に、マシン記述によって実装する立場もある。しかし、既存の JavaVM を利用し、エミュレータの移植に伴う作業を低減することには有用性があると考える。本研究ではこれを示すため、多数の複雑な命令を持つ x86 アーキテクチャを対象とするエミュレータが、JavaVM を用いることで様々な環境上で容易に実現できることを示す。

２）既存のデスクトップアプリケーションを Web 配信する新しい手法の提案

本研究では、アプリケーションの可搬的利用のためのユーザーモードエミュレータを提案すると同時に、既存のデスクトップアプリケーションを Web 配信するための手法の一つとして、ユーザーモードエミュレータを用いることを提唱している。先に述べた汎用性と可搬性を兼ね備えたユーザーモードエミュレータを、Web アプリケーションの配信基盤に用いることで、既存の多くの x86/Linux 用のアプリケーションを、様々なプラットフォーム上の Web ブラウザの上で実行できる。これにより、従来 Web アプリケーションを開発する上で、独自の言語・開発環境を用いてきた実装上の問題を解決可能である。

今日の Web アプリケーションを構成する言語・開発環境は複雑かつ多彩である。特にクライアントサイドでは、ユーザーインターフェース(U/I)を構成することに特化した環境として、JavaScript[9]、Microsoft Silverlight[10]などがある。一方で、既存のデスクトップアプリケーションの一部、または全部を Web ブラウザと同一のプロセス内で実行しようとする立場もある。ActiveX[11]や Java アプレット、Google Native Client[12]がこれに該当する。後者は汎用言語で書かれ、システムコールを利用するデスクトッププログラムを Web ブラウザ上で実行可能であるという点で有用であるが、U/I を構成するための API はデスクトップアプリケーションで使用するものと異なるなど、Web アプリケーションを構成するために特別のコーディングが必要

であるという点では、JavaScript などと同様である。

本研究では、既存のデスクトップアプリケーションを構成するソースコードに一切の変更を加えず、Web ブラウザ上でシステムコールの利用と描画処理が可能な、Web アプリケーションの配信基盤を目指した。類似のものとしては、Adobe Air[13]があるが、ネイティブコードで構成される旧来のデスクトップアプリケーションを対象としてはいない。本研究では、xlib 以外の、特定のランタイム環境や描画ライブラリに依存しない、あらゆる x86/Linux 用の GUI プログラムが、Web ブラウザ上で動作することを目指した。Linux 用のネイティブコードからなる GUI プログラムに対し、こうした機能を提供するシステムは、今日まだ存在しない。

第 4 章　本エミュレータの概念設計

4.1 本エミュレータの適用と特徴

図 4.1.1 に本エミュレータの適用概念を示す。本エミュレータのユーザーは、x86/Linux 用のアプリケーションが、あたかも任意の OS・アーキテクチャ上で動作しているものとして利用できる。その一例として、エミュレータの外側で動作している通常のアプリケーションと、エミュレーションする x86/Linux 用のアプリケーションが連携して処理を行うことが考えられる。本エミュレータは、ディスク装置やネットワーク I/O を仮想化することなく、アプリケーションに対し、実機のデバイスにアクセスするための手段を提供する。このため、エミュレータの動作環境によっては、シェルのパイプ機能やクリップボードなどを利用して、エミュレーションされるアプリケーションが、エミュレータの外側のアプリケーションとデータをやり取りして連携することが想定される。

本エミュレータがエミュレーションの対象とする x86/Linux プログラムは、GUI、マルチスレッド、共有ライブラリの動的リンクなどを利用する今日の一般的なアプリケーションプログラムである。OpenOffice、GIMP などがこれに該当する。筆者は、これらの x86/Linux アプリケーションを構成する ELF バイナリに何らの変更を加えることなく、エミュレータ上で利用可能となることを目指した。

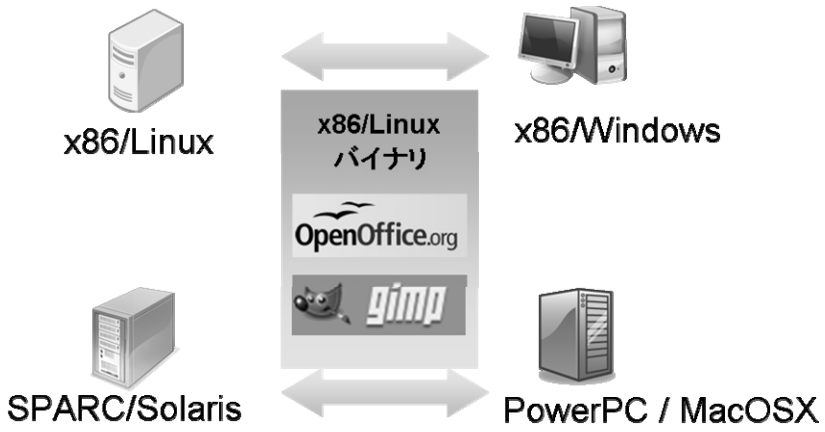

図 4.1.1　本エミュレータの適用

本エミュレータの技術的な特徴として、マルチスレッドプログラムのエミュレーション時にエミュレータ自身がマルチスレッド動作を行うこと、mmap システムコールのエミュレーション時に、エミュレーションするメモリ空間とエミュレータの下部で動作する実機のメモリ空間の一部が、単一のメモリイメージとしてアプリケーションから認識されること、ローダーモジュールの実現において、x86/Linux 用の共有ライブラリローダーをエミュレータ上でエミュレーションすることにより、ローダーの実装過程を大幅に簡略化したこと、などが挙げられる。

第 5 章　本エミュレータの外部設計

5.1 エミュレータの外部設計の概要

本エミュレータの基本的な入出力を図 5.1.1 に示す。エミュレータの起動時に、x86/Linux 用の ELF バイナリが入力され、そのプログラムが実行中に行ったシステムコール操作の一部が本エミュレータの出力となり、エミュレータの下層で動作している OS に適宜渡される。このシステムコールをエミュレータの外部に出力する処理を、説明のため”システムコールの実体化”と呼ぶ。

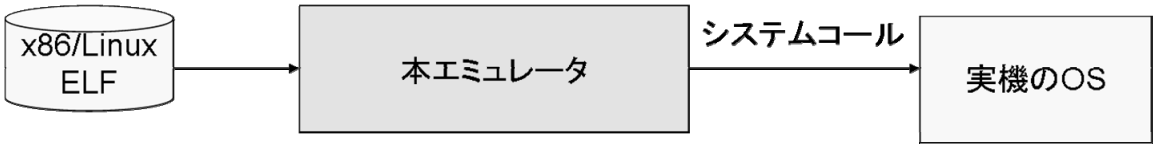

図 5.1.1 本エミュレータの入出力

エミュレータがシステムコールをどの程度実体化するかは、エミュレータの可搬性に大きく関わってくる。本エミュレータは Linux 以外の環境上でも動作することを目標として掲げているため、Linux 独自のシステムコールを実体化しない。Linux 独自のシステムコールは実体化せずに本エミュレータによってエミュレーションされるか、様々な OS 上でサポートされている POSIX 互換システムコールのような、実体化しても問題が生じないシステムコールの組み合わせとして実行する。

さらに、POSIX 互換システムコールのように、異種 OS 間で互換性のあるシステムコールであっても、システムコールの引数を直接渡すことは出来ない場合がある。例えば、mmap()システムコールでのマップ先アドレス指定のように、エミュレータの下層で動作している JavaVM のプロセスメモリ空間を破壊する可能性のある操作は、マップ先アドレスをずらすなどの処理が必要となる。

本エミュレータがシステムコールを実体化する単位として、POSIX システムコールにするのか、JavaVM が提供するランタイムライブラリの組み合わせにするかという問題がある。前者はその実現上、pure-Java なエミュレータの実現と言う観点から外れ、エミュレータ自身の可搬性の向上という本研究の目的に対し矛盾を伴う。一方、後者はシステムコールを JavaVM 上で提供することから、エミュレータの可搬性は大幅に高まる。しかし、UNIX ドメインソケットのような Java 言語のランタイムライブラリでサポートできないシステムコールも存在するため、エミュレータの汎用性という観点からは後退する。本エミュレータはこの 2 つの手法をユーザーが任意に設定可能なものとして共存させている。

図 5.1.2 に本エミュレータの全体象を示す。本エミュレータは x86/Linux 用のユーザーモードアプリケーションが使用する可能性のある、ほとんどの非特権命令をエミュレーションする。また本エミュレータ上では、基本的な入出力機能のほか、スレッド、メモリ管理、ネットワークなどに関連する Linux 用のシステムコールを利用可能である。これらのシステムコールは、GUI や共有ライブラリの動的リンクといった今日の一般的な Linux アプリケーションが利用する機能のシステムコールレベルでのインターフェースに該当する。

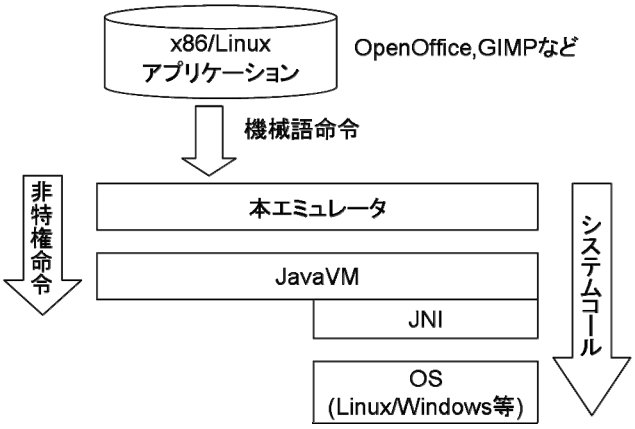

図 5.1.2 本エミュレータの全体象

本エミュレータ上においてアプリケーションが利用可能な機能として、x86 の非特権命令や Linux 独自のシステムコールが挙げられる。さらにこうした機能を組み合わせて、アプリケーションは、GUI やマルチスレッド、共有ライブラリの動的リンクといった機能を本エミュレータ上で利用可能である。また、本エミュレータはユーザーに対し、このようなアプリケーションを Java アプレットとして Web ブラウザ上で利用することを可能とする。

本エミュレータが提供しない機能として、x86 の特権命令エミュレーションや I/O デバイスのエミュレーションが挙げられる。こうした機能を利用するプログラムとして、Linux のカーネルモードモジュールや、デーモンが存在する。また、本エミュレータは基本的に POSIX システムコールの利用に際し、エミュレータの外部に存在する OS のシステムコールライブラリに依存するため、POSIX 互換システムコールを持たない OS 上で本エミュレータを使用することはできない。ただし、後述するように NestedVM と本エミュレータを組み合わせた場合は、この限りではない。

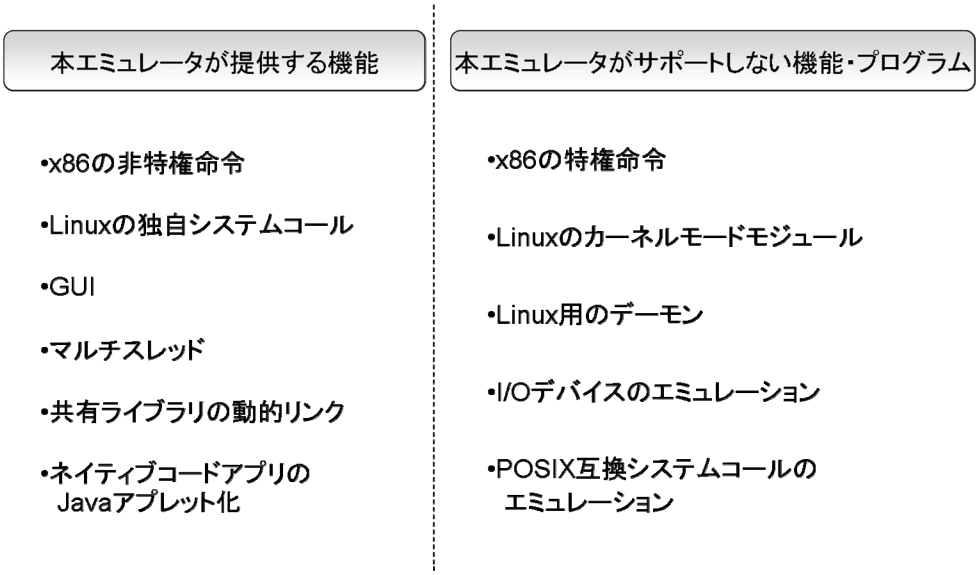

図 5.3.1　本エミュレータが提供する機能としない機能

5.2 命令列の逐次解釈によるエミュレーション

本エミュレータは、メモリ空間に展開した x86/Linux プログラムの命令列を逐次解釈しながら、エミュレーションを行う。本エミュレータは、エミュレーションするプログラムを 1 命令ずつエミュレーションしながら、逐次命令列の解析とエミュレーションコードの実行を行う。

一方、NestedVM は、アプリケーションのバイナリ内に存在する命令列は全て静的解析可能であることを前提としており、事前に全てのエミュレーションコードを生成し、Java クラスファイルとして出力する。アプリケーションの開発者から見た場合、NestedVM はコンパイルとリンクが完了したアプリケーションのバイナリファイルに対し、さらにもう一段階の操作を加えるだけで、単体実行が可能な Java クラスを生成可能であるため、より有用性が高い。

x86/Linux 用のアプリケーションを構成する命令列は静的解析が困難である。これは命令列のアライメントが揃っている必要が無く、間接分岐命令によって分岐する対象のアドレスに無数の組み合わせが存在するためである。また、共有ライブラリの利用は、プログラムの実行を開始する直前まで命令列を予測することを妨げている。以上の理由により、本エミュレータは命令列の逐次解釈によってエミュレーションを行う立場を採った。

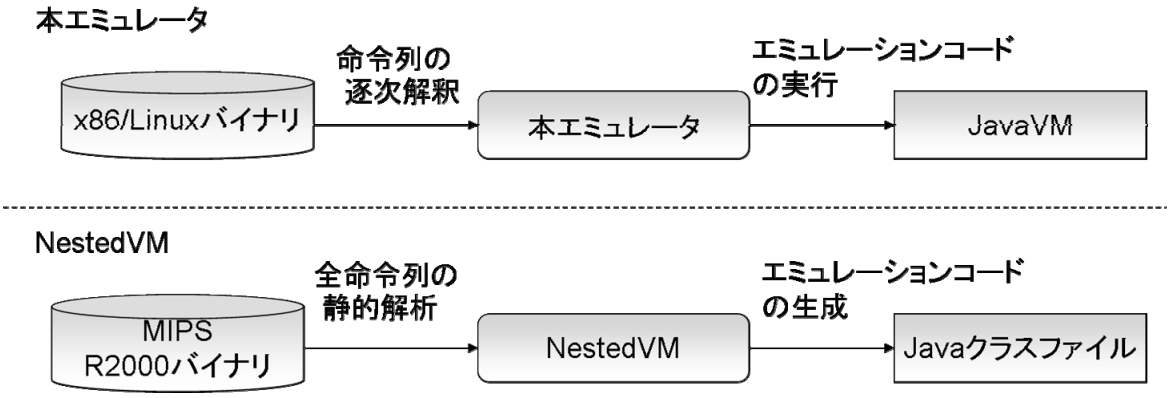

図 5.6.1　命令列の逐次解釈と静的変換

5.3 エミュレータがアプリケーションに提供する可搬性

本エミュレータがアプリケーションに提供する可搬性を図 5.2.1 に示す。本エミュレータは、C/C++といった言語で書かれ、libc や xlib といった既存のライブラリをリンクし、x86/Linux 環境に特化したアプリケーションプログラムのバイナリを様々な環境で利用可能とする。ここでいう x86/Linux アプリケーションは、x86 の命令セットや Linux 独自のシステムコールを利用するほか、共有ライブラリの動的リンクや、GUI、マルチスレッドといった機能を利用する。本エミュレータは、こうしたアプリケーションを Linux 以外の OS 上や、x86 以外のアーキテクチャ上、Java アプレットとして任意の Web ブラウザ上において利用可能とする。

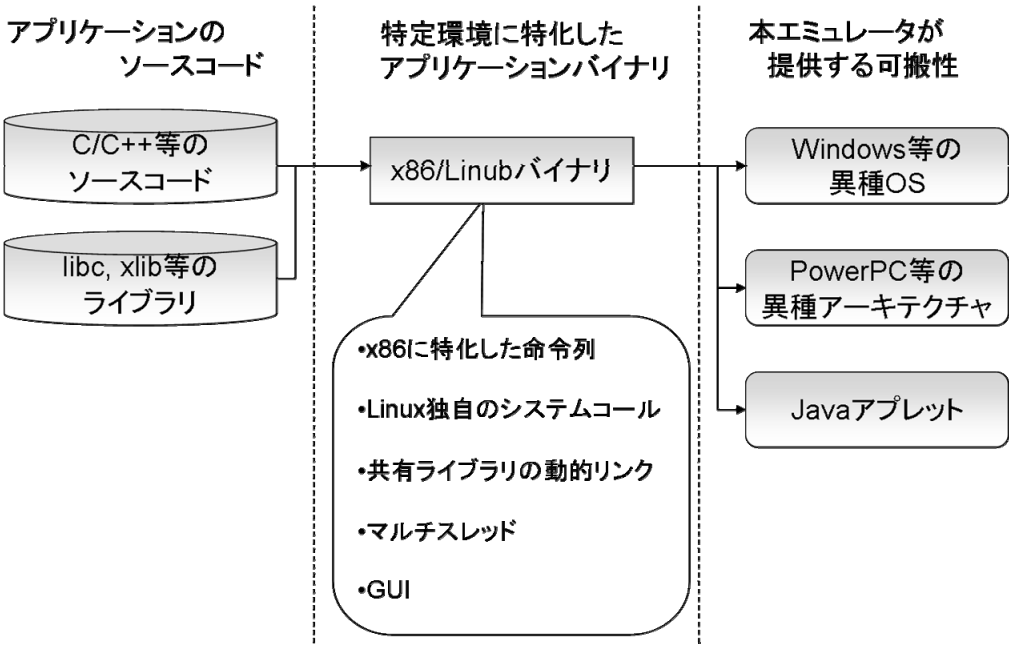

図 5.2.1　本エミュレータがアプリケーションに提供する機能

本エミュレータは既にコンパイル・リンク済みの x86/Linux 用アプリケーションバイナリを対象とする。このため、アプリケーションがどのような言語で書かれ、どのようなコンパイラやリンカで作成されたかといったことに本エミュレータは影響されない。

LLVM のように独自のコンパイラを用いて、C/C++で書かれたプログラムを対象としながら、複数の環境間で可搬的に利用可能な中間コードを生成する立場もある。しかし、今日の x86/Linux アプリケーションは、gcc に施された gnu 拡張などによって、ソースコードレベルであっても、異種環境間で可搬性を得ることが難しくなっている。このため筆者は、アプリケーションの作成者がどのようなコンパイラや開発環境を用いているかに依存せず、バイナリレベルで可搬性を提供することには合理性があると考えた。

本エミュレータがアプリケーションに対し提供するシステムコールのインターフェースは、内部割込みを契機として発動するレイヤに属している。一方、NestedVM のように、アプリケーションに対し、NestedVM 独自の newlib をリンクすることを強制し、高級言語のレイヤでシステムコールを提供する立場もある。

本エミュレータは、アプリケーションが利用する割込み命令によってシステムコールを検出するため、アプリケーションがリンクする標準ライブラリに依存しない。アプリケーションの開発者がデバッグ等の目的で改変した libc を用いていても、本エミュレータは改変部分も含めてエミュレーション可能である。

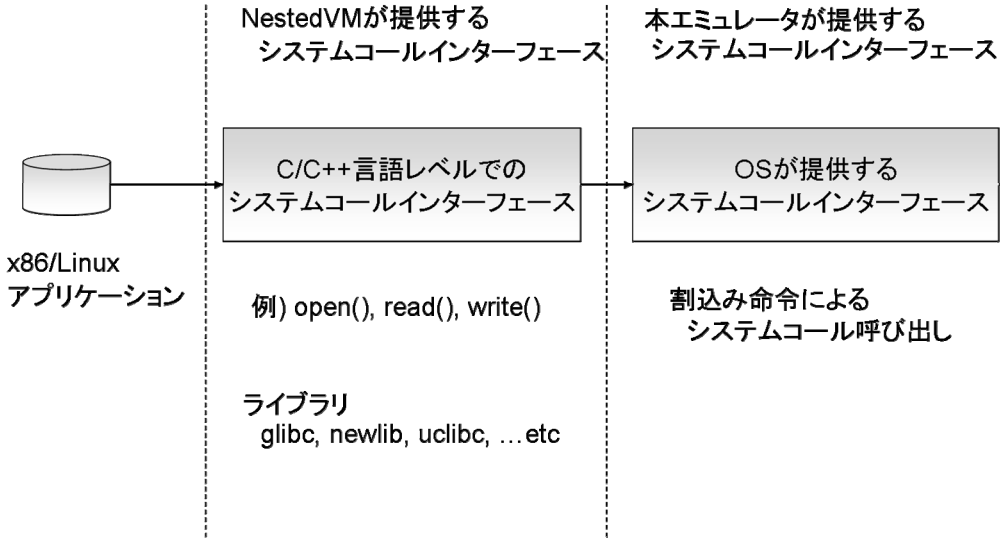

図 5.5.1　システムコールのインターフェースが所属するレイヤ

5.4 アプリケーションが利用可能なメモリ空間の制約

本エミュレータがアプリケーションに対し提供するメモリ空間は、本エミュレータを実行する JavaVM プロセスに所属する単一のプロセスメモリ空間の一部である。今日の多くの環境では、プロセスメモリ空間の最大サイズは 4GB である。一方、本エミュレータが x86/Linux 対し提供する必要のあるメモリ空間の最大サイズも 4GB である。

現在のところ、本エミュレータは、エミュレーションする対象のプログラムが、それほど巨大なメモリ空間を利用しないことを前提とし、この問題に対する対策を特に行わない。一方、JPC はエミュレーションするメモリ空間の一部をファイルとしてディスクに格納し、4GB 以下のメモリ領域で 4GB のメモリ空間を再現している。

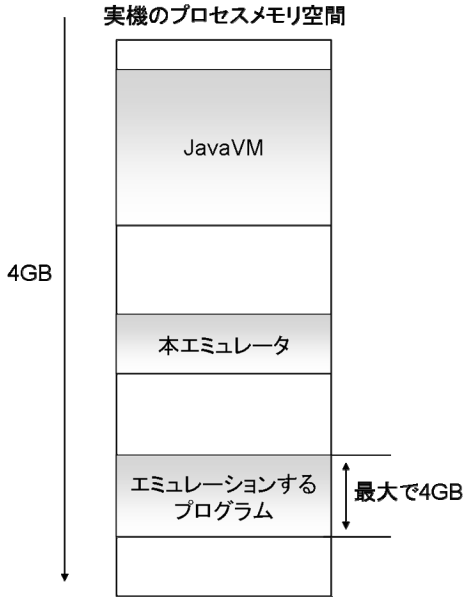

図 5.7.1　エミュレーションするプログラムと実機のプロセスメモリ空間の関係

第 6 章　本エミュレータの内部設計

6.1 本エミュレータの内部設計の概要

図 6.1.1 に本エミュレータの内部構成を示す。本エミュレータは、命令セットや主記憶といった、x86/Linux 用のユーザーモードアプリケーションが必要とするハードウェアをエミュレーションする。また、システムコールの一部を直接実体化せずに、エミュレーションしている。エミュレーションされるアプリケーションからは自身があたかも、x86/Linux 上で動作しているかのように見え、エミュレータの下層で動作している OS からは、単一の JavaVM プロセスのように見える。

本エミュレータを含め、一部のユーザーモードエミュレータの特徴の一つとして、x86 ハードウェアのエミュレーションを行う部分と、Linux-OS に相当する機能を提供する部分が、同一のエミュレータプログラムの中に混在していることが挙げられる。本研究における目標の一つである、マルチスレッドや共有ライブラリを使用するアプリケーションのサポートは、この OS に相当する部分の機能に依存している。

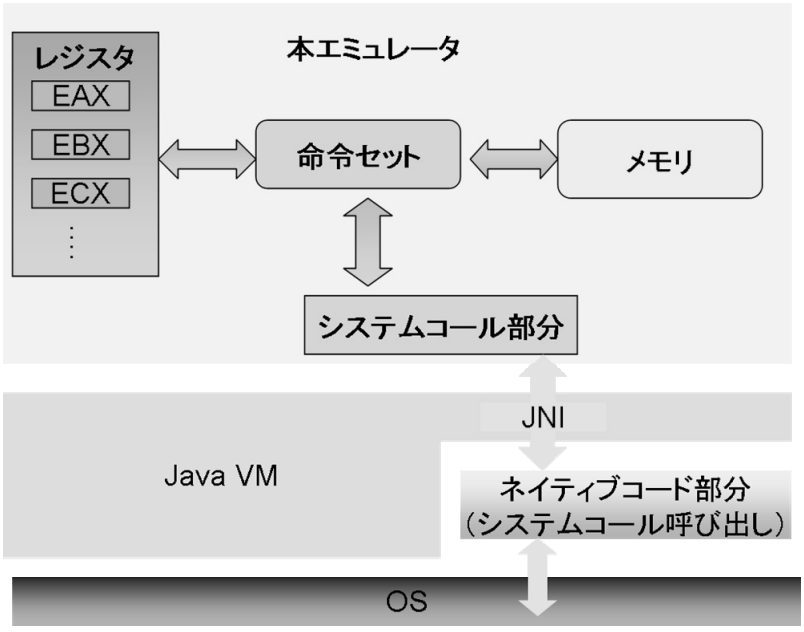

図 6.1.1　本エミュレータの内部構成

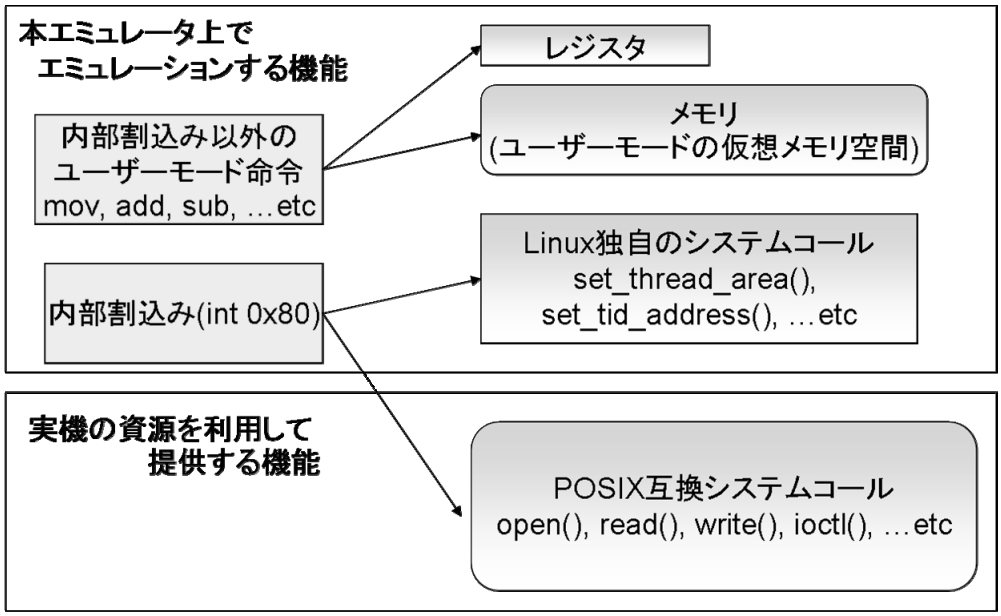

図 6.1.2　本エミュレータの担当範囲

6.2 x86 ハードウェアの仮想化

図 6.2.1 に本エミュレータによって仮想化された x86 ハードウェアを示す。本エミュレータは x86/Linux 用のユーザーモードのアプリケーションを実行する上で必要となるハードウェアのみをエミュレートする。現実の x86 ハードウェアに搭載されているが、本エミュレータが搭載していない機能として TLB やリアルモードのメモリアドレス空間、周辺機器を接続するバスなどが存在するが、いずれもユーザーモードエミュレータには不要な機能である。

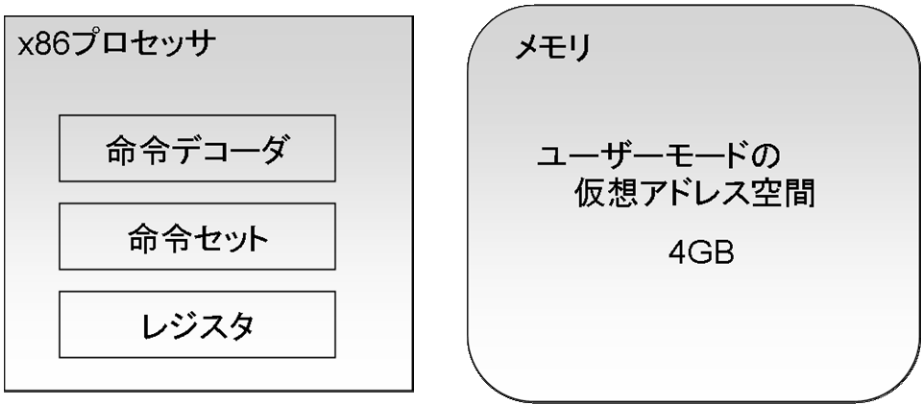

図 6.2.1　仮想化対象のハードウェア

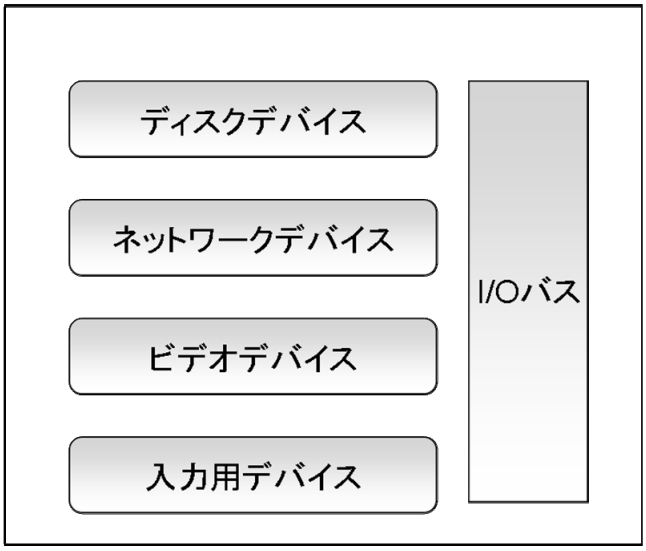

図 6.2.2 本エミュレータが提供しないハードウェア

(1) 命令デコーダ

命令デコーダは、本エミュレータが提供するメモリ空間に展開された x86/Linux 用の ELF バイナリから、一命令ずつバイナリコードを読み取り、本エミュレータが認識可能な内部表現に変換している。

実装上の問題から、本エミュレータでは二種類の命令デコーダを併用している。一方は低速だが x86 の全ての命令セットをデコード可能な汎用ディスアセンブラを改変したものであり、一方は x86 の命令セットのうち命令語長が 4 バイトまでで、かつ浮動小数演算命令以外の命令のデコードのみが可能な比較的高速な命令デコーダである。二つの命令デコーダの平均デコード時間には約 2 倍の差がある。

x86 命令のデコードには、命令語長が可変長であることと、多用なメモリアドレッシングを取りうることにより、複雑な処理を要する。デコーダが特に苦手とする命令として、" mov eax,dword [ebx+ecx*3]" のような命令がある。この mov 命令では第 2 オペランドのメモリアドレスの指定に、積和算で表現される間接アドレスが用いられている。

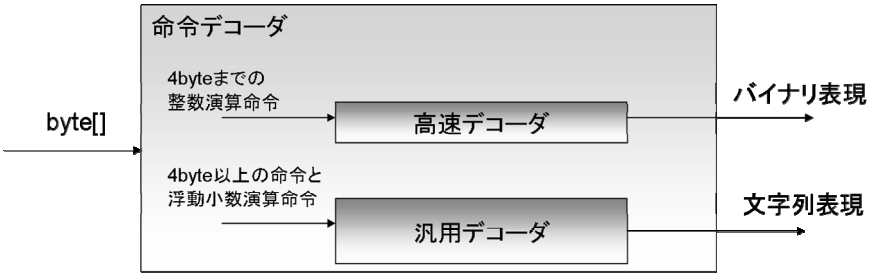

図 6.2.3　2 種類の命令デコーダ

(2) 命令デコード結果のキャッシュ

命令デコードの結果は、ある一定数までキャッシュされる。このキャッシュは LRU を用いて

おり、最近実行したメモリ番地の命令が格納される。最適なキャッシュサイズはエミュレーションするプログラムによって異なるが、筆者は経験上 32K 個に固定化した。命令デコード結果をキャッシュすることにより、プログラムによっては、キャッシュを施さなかった場合に比べ 20 倍程度の高速化を実現した。

本エミュレータには実装していないが、より効果的な手法として、エミュレーションを行う前にエミュレーションするプログラムの全命令列を静的に解析する立場もある。NestedVM は、エミュレーションするプログラムの命令列は、事前に全て静的解析可能であることを前提としているため、エミュレーション中に命令デコードを一切行わない。筆者は、本エミュレータが共有ライブラリの動的リンク等をサポートするため、命令列の完全な静的解析は難しいが、一方でプログラムのシンボルテーブルを解析する機能を有しているため、ある程度までは類似の手法を採用することは可能であると考えている。

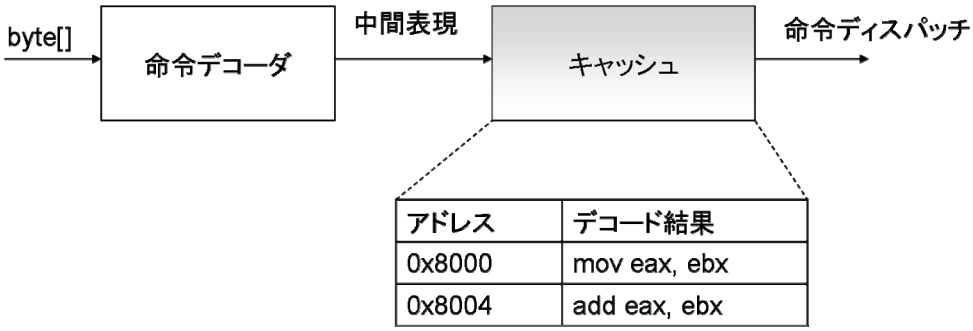

図 6.2.4 命令デコード結果のキャッシュ

(3) 命令ディスパッチ

命令ディスパッチは、命令デコーダが出力した表現形式を元に、エミュレーションする対象の命令を特定し、本エミュレータの処理過程を遷移する処理を指す。VM における命令ディスパッチの最適化手法として、threaded-code や super-instruction の有効性が指摘されている[14]が、本エミュレータはまだいずれも実装していない。その理由として、Java 言語、又は JavaVM 上において関数ポインタに相当する高速な機能を提供することが困難であることが挙げられる。

Java 言語レベルでの関数ポインタに相当する機能として、リフレクション機能があるが一般的に低速である。一方、Java 言語レベルではサポートされていないが、JavaVM に実装されたバイトコード命令として goto 命令があり、バイトコードを直接操作することで関数ポインタと同等の機能を提供できる。しかし、Java バイトコードレベルでの直接操作は実装上敷居が高いため、本エミュレータでは javassist や Java Compiler API を用いた Java 言語ソースコードの動的コンパイルを行っている。この動的コンパイルは、JavaVM の invoke_virtual 命令で呼び出される関数を動的に生成し、生成関数内に目的のコードを配置することで、JavaVM 上での動的処理を実現

している。

動的コンパイル手法は、実行時コンパイルのオーバーヘッドが極端に大きいため、threaded-code には不適である。一方、複数の命令を組み合わせて単一命令とする super-instruction では、組み合わせる命令数によって損益分岐点が存在するため有用である。

(4) 命令セット

本エミュレータは、x86 の命令セットのうち、ユーザーモードのプログラムが直接実行する可能性のある命令について、サポートする必要がある。mov, add といった非特権命令がこれに該当する。一方、Linux のカーネルモードでのみ実行される特権命令については実装する必要はない。in,out などの命令がこれに該当する。

また x86 では、非特権命令のうち、浮動小数演算のためのコプロセッサ命令として FPU,MMX,SSE の各命令セットが定義されている。このうち、FPU 命令については gcc を含む今日の一般的なコンパイラの多くがデフォルトで、コンパイラが出力するコード中で使用することを前提としている。MMX,SSE 命令については、コンパイラのオプションとしてユーザーが指定した場合にのみ適用される。したがって、本エミュレータでは浮動小数演算命令のうち、FPU 命令のみをサポートする。

本エミュレータが実装すべき x86 の非特権命令のうち、内部割込みで用いられる int 命令以外の全命令は、後述するエミュレータ上のレジスタ、メモリ空間に対する操作として定義される。Java 言語の数値表現形式はビッグエンディアンであり、リトルエンディアンで動作する x86/Linux 環境とは異なる。このため、命令セットをエミュレーションするうえで、演算の入出力先としてメモリ空間が指定された場合、演算前にメモリから読み込んだデータのエンディアン表現をビッグエンディアン形式に変換し、演算後に再びリトルエンディアン形式に直してからメモリに出力する処理を施している。入出力先がレジスタの場合は、エンディアンの変換を行わず、直接入出力を行っている。

内部割込みで用いられる int 命令は、エミュレーションする対象のプログラムが利用するシステムコールを検出し、その一部をエミュレータの下で動作している OS のシステムコールライブラリに処理を転送する。

表 6.2.1 に本エミュレータに実装された命令セットの一覧を示す。本エミュレータは x86 の命令セットのうち、ユーザーモードで使用される非特権命令のエミュレーションを行うため、特権命令は実装対象から除外している。また、浮動小数演算命令の FPU,MMX,SSE 命令のうち、gcc がデフォルトで出力しない MMX,SSE の命令セットについても除外している。実装予定の非特権命令のうち、FPU 命令を除くほとんど全ての非特権命令を本エミュレータでは実装した。

FPU 命令は四則演算などの一般的な命令のみを実装し、BCD 符号化などの命令は未実装である。また、x86 の仕様として全 FPU 命令は内部で 80bit に拡張された数値データに対する演算として定義されている。しかし、Java 言語レベルでサポートされている浮動小数演算の最大桁数は 64bit までであるため、本エミュレータでは 64bit の演算としてエミュレーションを行っている。このため、一部の演算において x86 の本来の仕様では起こりえない桁落ちや情報落ちが発生する。

BigDecimal などのソフトウェアフローティングライブラリを使用することで FPU 命令の精度を向上させることも考慮したが、実装の容易性を最優先に考え今回は除外した。

各命令はエミュレータ内部で次のような関数で実現されている。ここでは add 命令の一部を例に取った。

```
public static int add_r32_imm32(Register reg, Memory mem, String[] operands );
```

この関数は x86 の”add r32, imm32”命令に相当し、32 ビットの汎用レジスタのいずれかに 32 ビットの定数値を加えるものである。オペランドの表現が String 型文字列であるのは、本エミュレータの実装当初に文字列ベースの命令デコーダを採用したからである。一部の僅かな命令においては Java 言語の数値データの集合としてオペランドを表現しているが、性能向上に与える影響が僅差であったため、全命令に対しては適用していない。オペランドの内部表現の改善による性能向上が見込まれるのは命令デコーダの初期段階から、命令の実行時まで一貫した改善を施した場合だと考えられる。

表 6.2.1 命令セットの実装状況

命令種別	例	備考
整数演算	ADD,SUB, MUL など	全て実装済み
論理演算	AND,OR,NOT など	全て実装済み
比較	CMP など	全て実装済み
シフト/ローテート	SAR,ROL など	全て実装済み
10 進算術演算	AAA,AAS など	全て実装済み
制御転送	CALL,RET, JMP など	セグメント間ジャンプ などは不要なため未実装
システム	LIDT,STR, ARPL など	ユーザーモードでは不要
ストリング	MOVS,CMPS など	全て実装済み
FPU	FLDCW,FILD, FADD など	64 ビットまでの演算のみ対応
MMX	MOVQ,PADDB, PAND など	未実装
SSE	MOVAPS,CMPPS など	未実装

(4) レジスタ

本エミュレータでは各汎用レジスタを Java 言語の 64bit 符号付変数である long 型として定義した。32bit の汎用レジスタを 64bit で表現したのは、Java 言語に符号無し変数が用意されていないためである。また、浮動小数演算で用いられる FPU レジスタは、Java 言語の 64bit 符号付浮動小数点数である double 型で定義した。x86 の FPU レジスタは 80bit の浮動小数点数のための領域として確保されているが、移植性を考慮して書かれた一般的な C 言語のプログラムが使用する浮動小数点数は 64bit までであるため、筆者は 64bit の double 型で十分だと考えた。また、前節で述べた理由により、MMX,SSE レジスタは実装しない。

後述するメモリ空間と違い、レジスタでは特にエンディアン形式を意識する必要が無いため、全てのレジスタは Java 言語で使用されるビッグエンディアン形式で、数値を格納している。

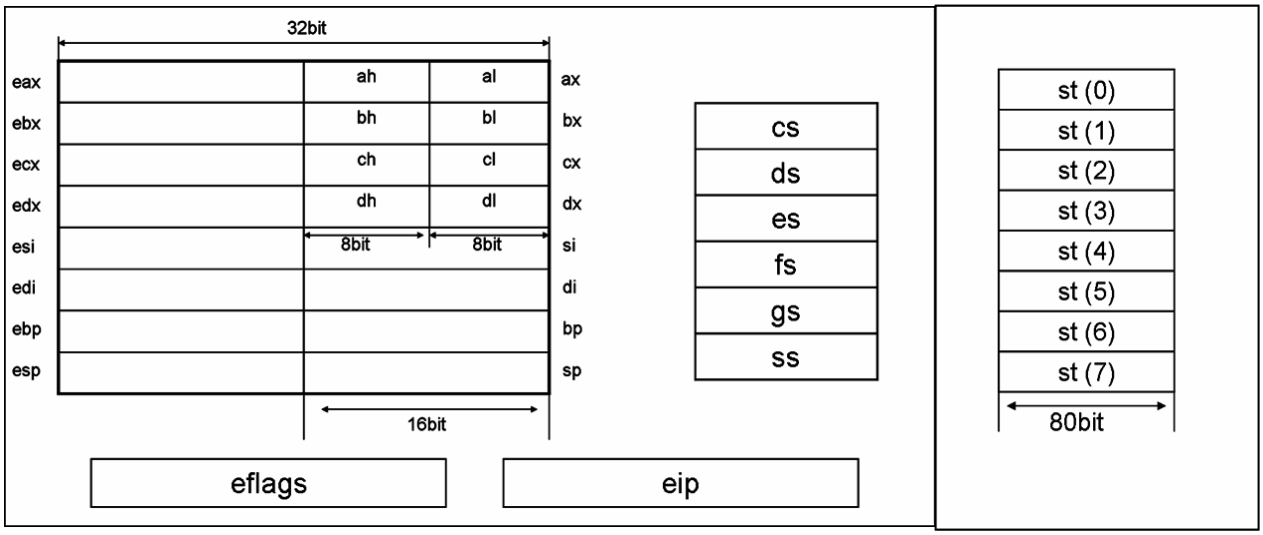

図 6.2.5　本エミュレータがエミュレーションする x86 レジスタ

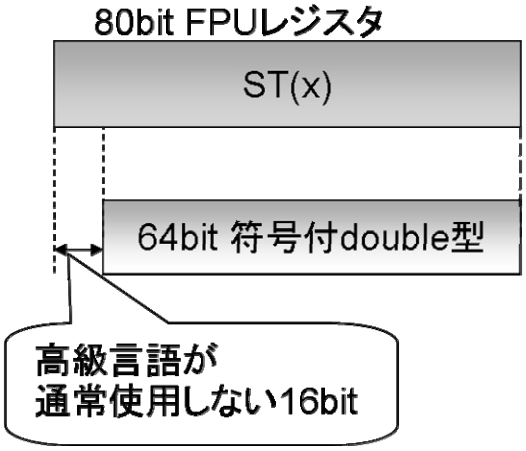

図 6.2.6　本エミュレータ上での FPU レジスタ

(5) メモリ空間

本エミュレータが提供するメモリ空間は、x86/Linux 環境におけるユーザーモードの仮想アドレス空間に相当するものである。

本エミュレータ上におけるメモリ空間は基本的に Java 言語の byte 型配列の集合と、各 byte 型配列を集約して操作するための関数群からなる。x86/Linux の仮想アドレス空間は最大で 4GB のメモリ空間をアプリケーションに対し提供している。Java 言語の仕様上、単一の byte 型配列として 4GB の空間を表現することは不可能であるため、本エミュレータでは複数の byte 型配列を集約している。また、各 byte 型配列は可変長である。これは、アプリケーションが mmap()システムコールを用いて、任意のファイルディスクリプタとメモリ空間を関連付けた際に、本エミュレータ上のメモリ空間の一部を切り離して、本エミュレータの下で動作する OS が管理するメモリ空間に結合するためである。

以下に本エミュレータのメモリ空間を操作するための内部インターフェースを示す。

・void memcpy(long address, byte[] set_array);
・byte[] memcpy(long address, long size);
・void mmap(long address, long size, long native_address);
・void munmap(long address, long size);

二つの memcpy()関数は、それぞれメモリ空間への値の設定と取得に用いられる。mmap()と munmap()は、アプリケーションが発行した mmap(),munmap()システムコールと連動し、メモリ空間のうち、指定した一部の空間を、エミュレータの外側で動作する OS の管理下におくことを意味している。

本エミュレータは、JavaVM 上の byte 型配列として確保したヒープ領域と、JavaVM の外部に存在する mmap()によって確保したネイティブなメモリ空間からなる、2 種類の異なるメモリ空間を集約し、アプリケーションに対し、単一な線形空間として提供する。

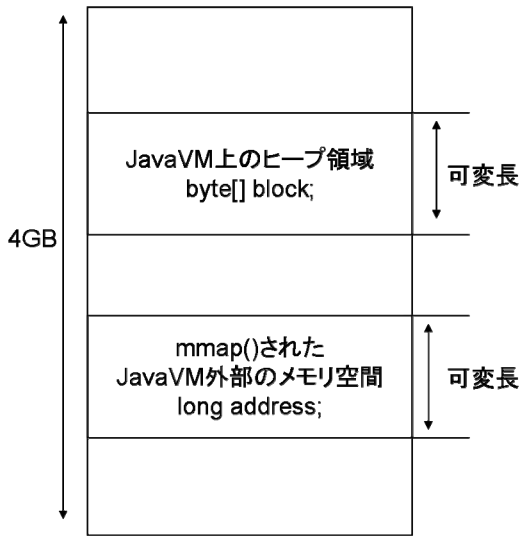

図 6.2.7　2 種類のメモリ空間の線形配置

本エミュレータのメモリを構成する一つの byte 型配列、または一度の mmap()で確保したネイティブメモリ空間を説明のためメモリブロックと称する。メモリブロックは、主に共有ライブラリのサポートを目的とした mmap()による実機のメモリ空間との連携が必要になるため、切り離しや連結が可能な仕様になっている。このため、あるメモリアドレスが所属するメモリブロックを特定するためには、検索処理が必要になる。二分探索によってある程度オーバーヘッドを緩和しているが、メモリアクセスの度に検索を行うコストは大きいため、最近使用したメモリブロックを LRU アルゴリズムに基づいてキャッシュしている。

本エミュレータの性能に寄与する効果的なキャッシュサイズを特定することは難しい。メモリブロックのサイズは可変長であり、キャッシュがカバーするメモリ領域のサイズもエミュレーション中に動的に変化するからである。筆者は経験的に、メモリブロックのデフォルトのサイズが 512KB のときに、キャッシュサイズを 5 程度にすると、特定のプログラムのエミュレーション時に良好な性能を示すことを理解し、固定した。より効果的に性能に寄与するキャッシュサイズは、エミュレーションするプログラムによって大きく異なることが予想され、動的に変更されることが望ましい。

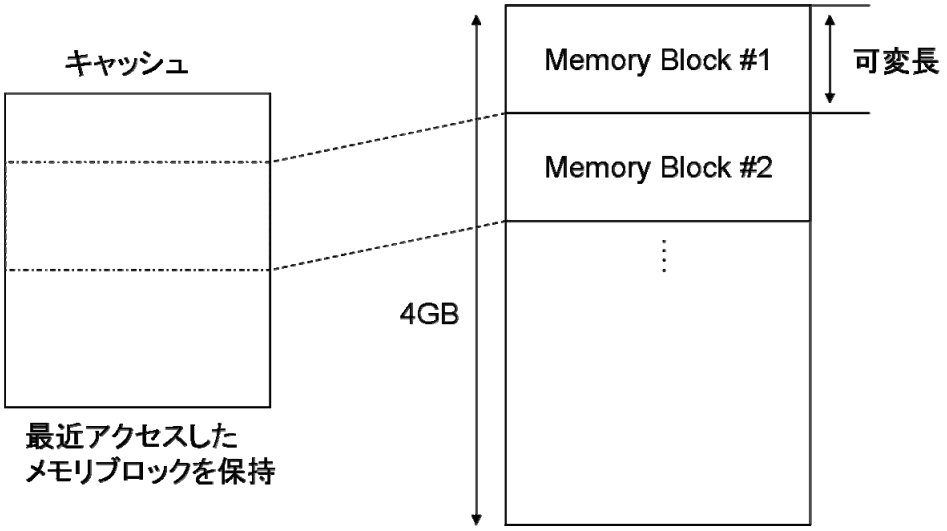

図 6.2.8　メモリブロックのキャッシュ

6.3 プロセスメモリ空間の初期化

本エミュレータに実装された ELF ローダーは、静的リンクされた ELF バイナリのヘッダを参照し、ELF バイナリ中の命令列、初期化データ等を、エミュレータ上のメモリ空間に展開する。共有ライブラリを使用するプログラムのロードは、本エミュレータの ELF ローダー単体ではサポートしない。共有ライブラリを使用するプログラムのロードには、Linux 環境における標準的なローダープログラムである ld.so をエミュレータ上で動作させ、間接的に対応している。

プロセスメモリ空間を初期化する作業は ELF バイナリをメモリ上に展開する作業に留まらない。適切な番地をスタックポインタに設定し、スタック上には環境変数、コマンド変数等を配置する。

以下では本エミュレータ上におけるプロセスメモリ空間の初期化作業について詳細に述べる。

(1)静的リンクされたプログラムのロード。

図 6.3.1 に ELF バイナリのフォーマットを示す。ELF バイナリをメモリに展開する際に、プログラムヘッダによって参照される空間をセグメントと呼び、セクションヘッダによって参照される空間をセクションと呼ぶ。通常セグメントは複数のセクションを包含している。プログラムヘッダとセクションヘッダは、ELF バイナリファイルの先頭から目的のデータが格納されている箇所までのオフセットと、そのデータをメモリ上に展開する先の仮想アドレス、サイズを保持している。

本エミュレータの ELF ローダーはセクション単位で、ELF バイナリ内のデータをメモリ上に展開する。セグメントを単位として展開しないのは、セグメントに所属する各セクションが非連続なメモリ空間に展開される可能性があるためである。

bss などの幾つかのセクションは、ELF バイナリ内に対応するデータを持たない。このセクシ

ョンが指す空間は明示的にゼロ初期化される。

symtab、strtab 等のセクションにはプログラムを構成する関数名などのシンボル情報が格納されている。本エミュレータは、ロードするプログラムのシンボル情報を解析し、エミュレータのデバッグ時の参考にした。本エミュレータには実装していないが、NestedVM ではこのシンボル情報をもとに、エミュレータの外部から任意の関数を呼び出すことを可能としている。

これらの作業が終わると本エミュレータは、ELF ヘッダが保持しているプログラムのエントリポイントをプログラムカウンタに設定する。

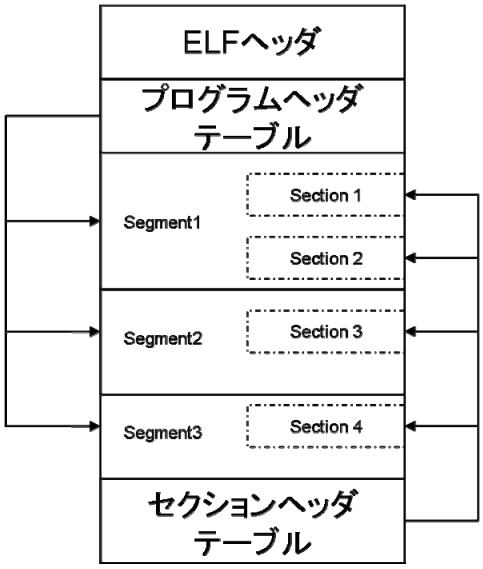

図 6.3.1 ELF バイナリの構造

(2)スタックの初期化

今日の一般的な Linux 環境においてユーザーモードプロセスのスタックの初期アドレスは一意に決まらない。これはセキュリティ上の問題を解決するため、Exec Shield と呼ばれる機能によって実現されている。本エミュレータでは簡単化のため、スタックの初期アドレスを十分に大きなアドレスで固定化している。

スタックポインタの値が決まると、本エミュレータはスタック上に環境変数と、コマンド引数を展開する。環境変数は本エミュレータの下部で動作する OS に設定された値を用いる。ただし、GUI アプリケーションが動作する上で必要となる DISPLAY 環境変数は、Windows 環境では設定されていない場合が多いため、適切な値が存在しなければ、本エミュレータが設定する。コマンド引数は、本エミュレータの起動時にユーザーから渡された値を用いる。その他、スタック上にはエミュレーションするプログラムのエントリポイントなど、Linux 環境において特有の情報が配置される。

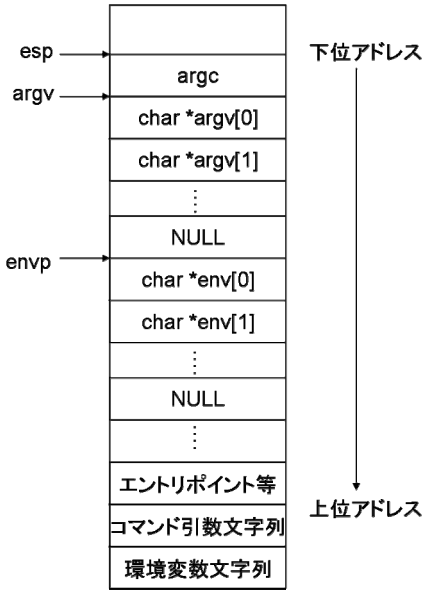

図 6.3.2　スタックの初期化

(3)共有ライブラリを動的リンクするプログラムのロード

Linux 環境における標準的なローダープログラムである ld.so は、静的リンクされ、単体での実行が可能な ELF バイナリである。本エミュレータ上において、共有ライブラリを動的リンクするプログラムをロードする場合、この ld.so をエミュレータがロードし実行することで間接的に対応している。

ld.so は mmap()を大量に行うプログラムである。mmap()の第一引数ではマップ先のアドレスを指定する場合がある。ld.so はこのアドレス指定機能により、共有ライブラリ内の任意の命令列を適切な番地に配置している。本エミュレータでは、システムコールの多くが後述する POSIX ラッパーを介して、エミュレータの外側で動作している OS のライブラリに依存しており、mmap()も例外ではない。しかし、アドレス指定が有効なまま mmap()を行えば、JavaVM のプロセスメモリ空間を破壊する可能性がある。また、OS ごとにマップ先アドレスとして指定可能な空間に差異があり、x86/Linux 用の共有ライブラリを、ld.so が指定したアドレスに直接ロード可能であるとは限らない。

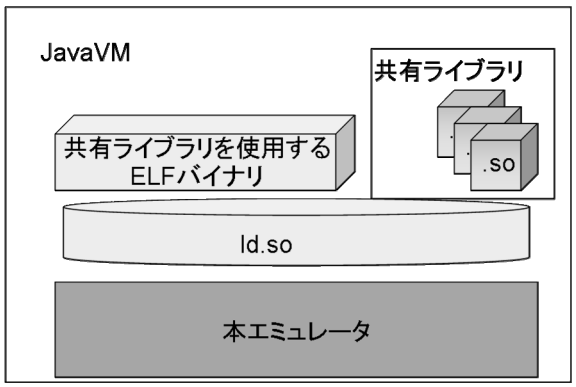

図 6.3.3 ld.so を用いた共有ライブラリロード

この問題を解決するため、本エミュレータでは、エミュレータ上で発行された mmap()のアドレス指定機能は全て無効化し、エミュレータの外部で動作している OS にとって都合のよいアドレスに共有ライブラリをロードしている。エミュレータが提供している x86/Linux の仮想メモリ空間と mmap()によって確保したネイティブコード中のアドレス空間は別個のものとして管理しているため、どの番地にロードされても問題はない。エミュレーションされるプログラムからは、あたかも連続したアドレス空間に共有ライブラリや ELF バイナリがロードされているように見えて、実際は Java 言語の byte 型配列で構成される空間と、ネイティブコード中の mmap()で確保された任意のアドレス空間が混在している。

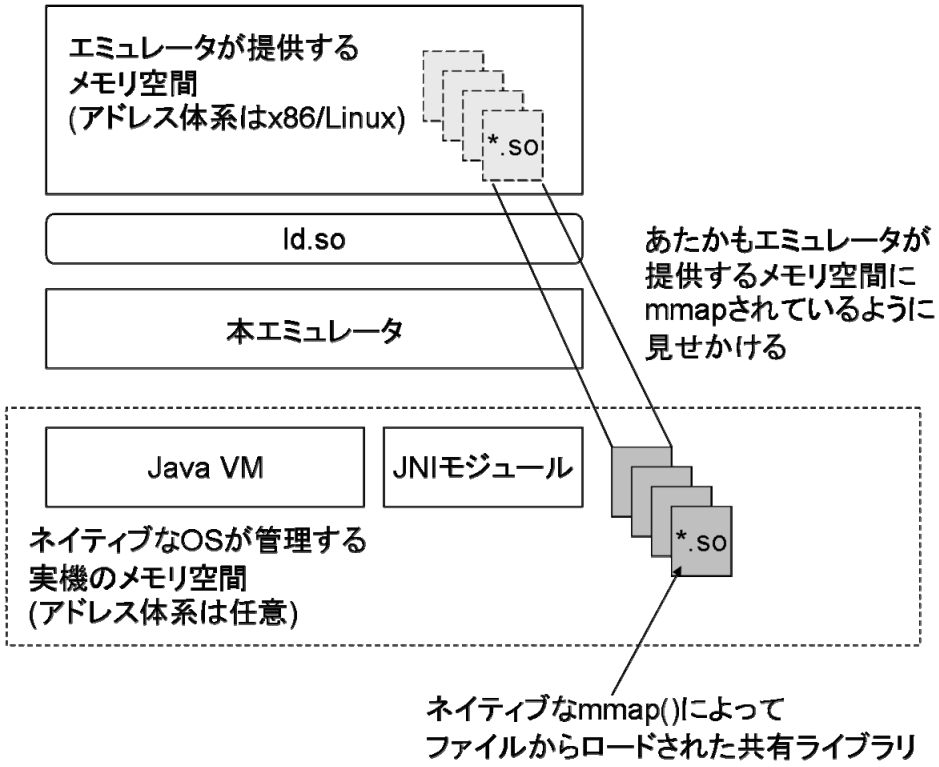

図 6.3.4 共有ライブラリの写像

6.4 システムコールの仮想化

Linux が提供するシステムコールは、POSIX 互換システムコールと、Linux 独自のシステムコールの 2 種類からなる。本エミュレータでは、エミュレータの動作環境として Linux 以外の OS も想定している。その第一段階として、POSIX 互換システムコールを有する他の UNIX-OS、第二段階として JavaVM が動作する全ての OS を想定した。

本節で述べるシステムコールの仮想化は、エミュレータの動作環境の OS が提供していないシステムコールを、本エミュレータがエミュレーションによって提供することを指す。第一段階におけるシステムコールの仮想化は、Linux の独自システムコールを POSIX 互換システムコールの組み合わせによって提供することを指し、第二段階では POSIX 互換システムコールを Java 言語のランタイムライブラリでエミュレーションすることを意味する。

システムコールの仮想化はその性質上、全機能を完全にサポートすることは困難であり、本エミュレータでは仮想化する対象を実現の難易度に応じて、Linux の独自システムコール、POSIX 互換システムコールの 2 種類に分類し、それぞれの仮想化機能の間に依存関係がなるべく生じないよう留意した。

(1) システムコール番号

Linux が提供するシステムコールは一意な番号によって管理される。この番号はアプリケーションからは、unistd.h で定義された値をもとに利用される。x86/Linux 環境では、アプリケーションは"int 0x80"命令を利用する際に eax レジスタに、所定のシステムコール番号を書き込んでおく。Linux カーネルはこの番号をもとに、システムコールを特定し処理を開始する。

本エミュレータがアプリケーションに対し提供するシステムコール番号は、エミュレータの起動時に外部のファイルから読み取って設定する。このファイルの記述は、unistd.h とほぼ同等な書式によってなされており、エミュレータのユーザーはいつでも任意の値にシステムコール番号を変更することが可能である。したがって、本エミュレータではシステムコール番号の非互換性による可搬性の低下を、設定ファイルを用いてある程度回避している。

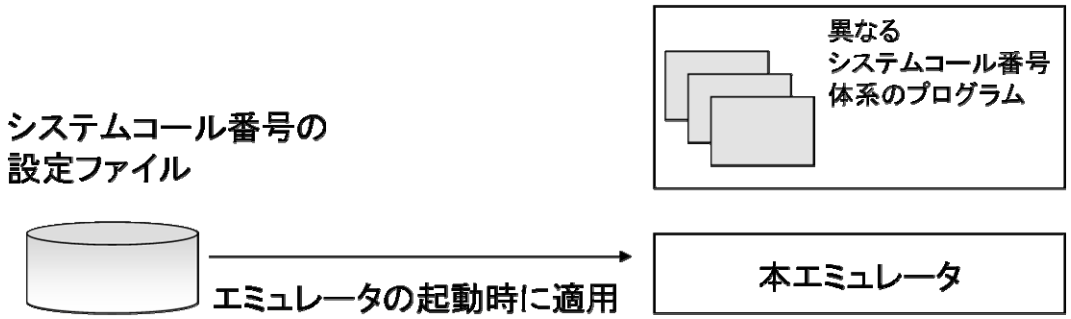

図 6.4.1　システムコール番号の設定

(2) システムコールの引数

Linux 環境で利用可能なシステムコールの引数は、値として渡されることもあれば、ポインタを介した参照として渡されることもある。本エミュレータがアプリケーションに提供するメモリ空間は、エミュレータの外部で動作する OS から隠蔽されている。このため、ポインタ参照によってシステムコールの引数が渡される場合、本エミュレータはポインタの参照先のデータをコピーして、エミュレータの外部のメモリ空間に展開する。システムコールの呼び出しが終了すると、本エミュレータは、エミュレータの外部のメモリ空間上にあるデータを、エミュレータ上のメモリ空間に上書きする。

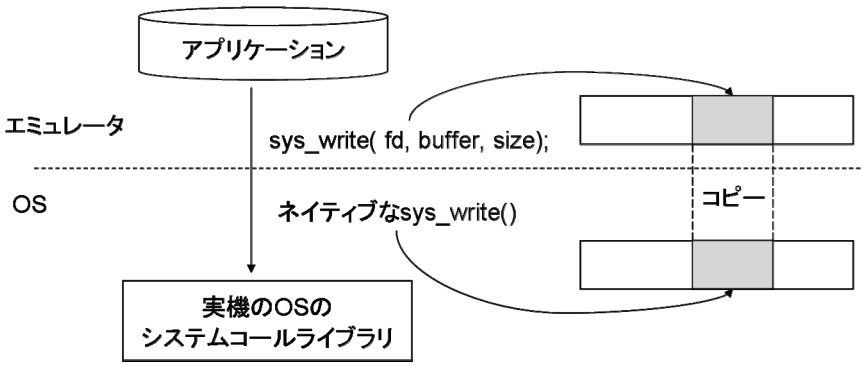

図 6.4.2　システムコール引数のコピー

(3) POSIX ラッパー

システムコール仮想化の第一段階では、POSIX システムコールは JNI(Java Native Interface)を用いて、JavaVM の下層で動作する OS のシステムコールライブラリを利用する方針を採った。このため、POSIX システムコールを JavaVM 上から呼び出すための JNI モジュールを実装し、これを POSIX ラッパーと称した。

多くのシステムコールでは、その引数に OS 環境に特化した定数値を取ることがある。例えば、socket()システムコールにおける、AF_INET などである。これらの定数値は、その意味するところは同じであっても、値は環境ごとに異なるため、POSIX ラッパーには Linux 環境の定数値と、エミュレータの動作環境の定数値を変換するための処理が組み込まれている。さらに、エミュレータの動作環境によって、システムコールが提供する機能に差異があり、一概に定数値を変換できない場合がある。例えば、socket システムコールにおいて UNIX ドメインソケットを表わす PF_UNIX が、Windows 環境で指定される場合が想定される。このような場合、POSIX システムコールは処理を中断しエラーを返している。

POSIX ラッパーが提供するシステムコールと、Java 言語が提供するランタイムライブラリを組み合わせる場合は、十分に注意が必要である。例えば、スレッド ID の取得に Java 言語の Thread#getID()を使用し、取得したスレッド ID に対し、pthread_join()を呼び出す場合などである。この組み合わせが正常に機能するかどうかは、JavaVM の実装に依存する。安全な回避策として、POSIX ラッパーが提供するシステムコールを使用する場合は、ある特定のシステムコール機能の集合全てに、POSIX ラッパーを用いるという解決策が考えられ、本エミュレータでは、可能な限り全てのシステムコールが POSIX ラッパーに依存するよう設計されている。

(4) NestedVM を用いた POSIX エミュレーション

NestedVM は POSIX システムコールのエミュレーションに対し、強力な機能を提供している。アプリケーションが使用する基本的な POSIX システムコールに的を絞り、Java 言語のランタイムライブラリでサポートできるものを取捨選択している。結果、スレッドやシグナルなどの機能をサポートしていないが、gcc や TEX の動作をサポートしている。

本エミュレータでは、NestedVM を利用し、POSIX システムコールを JavaVM 上でエミュレーションするための機能を間接的に実現しようと試みた。この際、UNIX ドメインソケットやシ

グナルなど、JNI を使用した POSIX ラッパーでは実現可能であっても、Java 言語のランタイムライブラリの組み合わせでは実現不可能な機能を平行してサポートすることを目指した。このため、対象とするアプリケーションの種類に応じて、NestedVM の使用の可否をユーザーが選択する方式を採った。

筆者は実装の都合上、JNI を使用した既存の POSIX ラッパーが直接 NestedVM の上で動作することを期待した。このため、JNI を実現するための幾つかの機能を NestedVM 上に追加する作業を行った。NestedVM 上のメモリ空間に、JavaVM 上のオブジェクトの写像を展開する処理や、NestedVM 上から JavaVM 上の資源にアクセスするための JNIEnv クラスの整備などである。

これらを実装することにより、JNI を使用するネイティブコードの POSIX ラッパーを改変することなく、直接 NestedVM 上に置くことが可能となる。本エミュレータのユーザーはシステムコールの処理層に NestedVM を含めるか否かを、エミュレータの起動時に設定する。

(5) Linux 独自のシステムコール

Linux の独自システムコールとして、set_thread_area()などがある。set_thread_area()はアプリケーションから指定されたメモリ領域を、[gs:0x00]で参照可能なセグメントのベースアドレスに設定する。この操作は特権命令を伴うため、Linux ではシステムコールとして実装している。これら Linux 独自のシステムコールはアプリケーションのソースコード中から直接呼び出されていなくても、glibc などの低レベルライブラリから呼ばれることが多い。

本エミュレータでは、Linux 独自のシステムコールを、POSIX 互換システムコールの組み合わせで再現している。これはシステムコール資源の仮想化における第一段階である、POSIX 互換 OS 間でのエミュレータの可搬的利用のためである。更に、第二段階として、NestedVM を用いた POSIX 互換システムコールの仮想化を行った場合でも、これらの Linux 独自システムコールが機能している。

(6) シグナル

シグナルの利用は UNIX 環境における標準的なプログラミングテクニックの一つである。sigwait()のように逐次処理を前提としてシグナルを待機することもあれば、sigaction()のように特定のシグナルを検出すると、実行中の処理を一時的に中断して、シグナルハンドラを呼び出す場合もある。

sigwait()のような待機手法の実現はエミュレータ上でも容易に実現可能であったが、sigaction()のようにシグナルハンドラを呼び出す処理の実現は、JavaVM の動作を外部から一時停止するための処理を組み込むという点で複雑である。本エミュレータでは、POSIX ラッパーが提供する sigaction()に渡すシグナルハンドラとして、ネイティブコード側のメソッドを定義し、ハンドラ内から JavaVM 上の任意の命令列を実行する仕様とした。図 4.3.14 にこの概念図を示す。

ネイティブコードから JavaVM 上の資源にアクセスするためのパラメータは、sigaction()の実行時に配置されるものとし、そのパラメータには、エミュレーションする対象のアプリケーショ

ンが利用するシグナルハンドラのアドレスなどが含まれている。エミュレーションする対象のシグナルハンドラが呼ばれるには、エミュレータの外部にあるネイティブコードのシグナルハンドラが起動し、JavaVM 上の本エミュレータのメソッドを呼び出し、本エミュレータ上のプログラムカウンタの値を変更する必要がある。シグナルハンドラの実行中に更にシグナルが配送された場合、再び同様の手順で入れ子構造の呼び出しが開始される。

JavaVM の外部から、エミュレータの動作を動的に制御するための機能は複雑であり、またシグナルに相当する機能を Java 言語のランタイムライブラリがサポートしていないこともあり、本エミュレータでは NestedVM を用いた pure-Java なエミュレーションにおいて、シグナルのサポートは実現できていない。

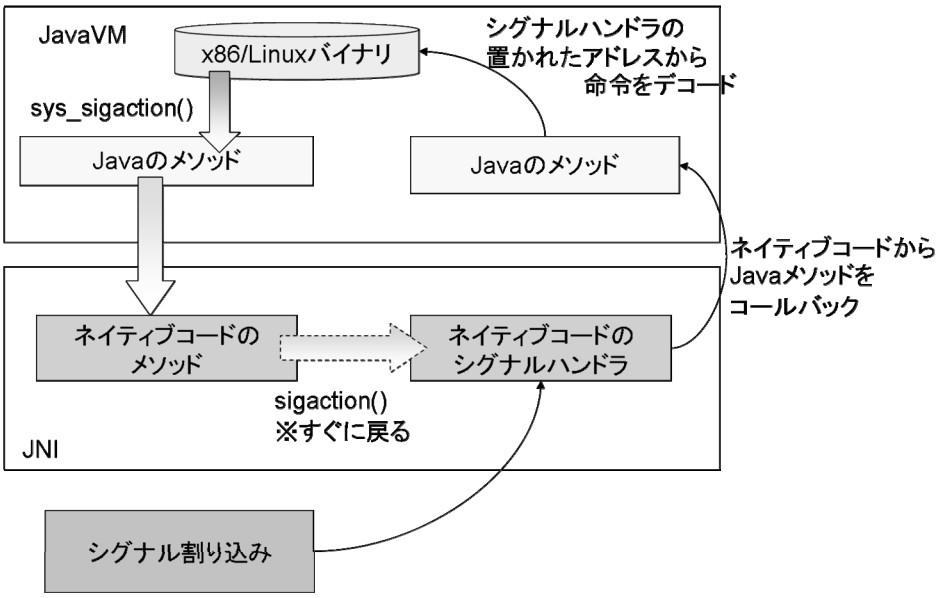

図 6.4.3 シグナルハンドラへの対応

(7) スレッド

スレッドもシグナル同様に、JavaVM の外部からエミュレータの動作を制御するための仕組みが必要になる。本エミュレータ上でのスレッドの生成方式を図 6.4.4 に示す。Linux アプリケーションがスレッドの生成のために clone()システムコールを呼び出すと、本エミュレータは pthread_create()を用いてエミュレーションを行う。JNI で記述された pthread_create()が呼ばれると、JavaVM の外側に新しいネイティブスレッドが誕生する。このネイティブスレッドは JavaVM 上のスレッドとは関連付けられていないため、JavaVM の外部インターフェースを用いて、JavaVM 上のスレッドと対応付けを行う。JavaVM 上のスレッドとの対応付けが完了すると、ネイティブスレッドはエミュレータの資源のうち、レジスタのみを新たに生成する。x86/Linux 環境のマルチスレッドプログラムは、スレッド間でメモリを同時に共有することはあっても、レジスタは排他的に利用する。本エミュレータでは、1 つのレジスタを複数のスレッドで排他的に利用するのではなく、レジスタはスレッドの数だけ生成されるものとして扱う。レジスタの生成

後、ネイティブスレッドはJavaVM上に処理を移行し、エミュレーションされるスレッドの実行が開始される。

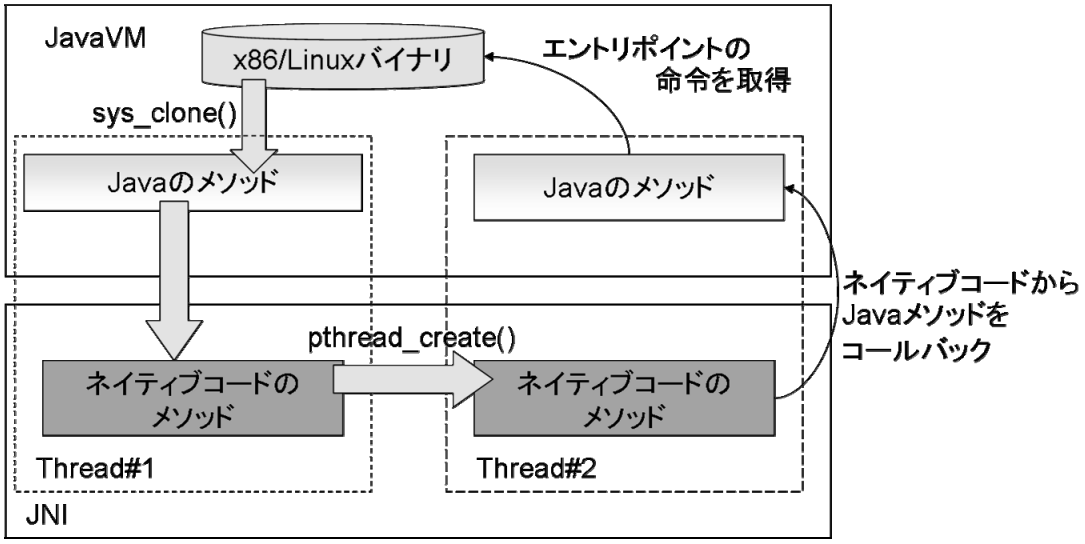

図 6.4.4　スレッドの生成

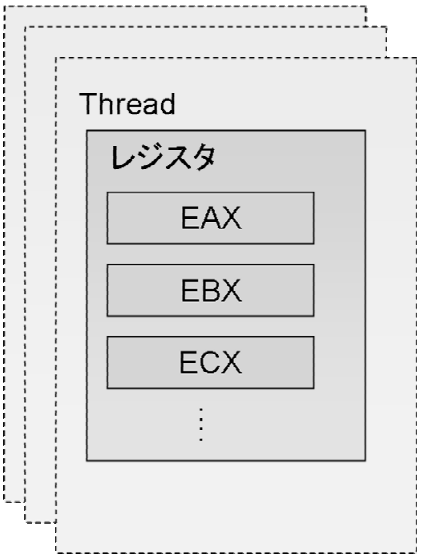

図 6.4.5 スレッドごとに独立なレジスタ

エミュレーションされるスレッドが pthread_exit()の呼び出しを行うと、エミュレータは終了コードをメモリ上に出力し、エミュレータ自身が動作していたスレッドの廃棄を試みる。この際、POSIXラッパーに定義されたpthread_exit()を直接呼びにいくのではなく、エミュレータ上において、あたかも pthread_exit()に相当する処理が行われたかのように振舞う。JavaVM の外部から、ネイティブスレッドと、JavaVM 上のスレッドの対応付けを解除し、不要な資源を解放後、

このネイティブスレッド自身が pthread_exit()の呼び出しを行う。

(8)利用可能なシステムコールの一覧

以上で述べた機能を実装した結果、本エミュレータ上で利用可能なシステムコールをまとめる。POSIX システムコールうち、本エミュレータの POSIX ラッパーがサポートしているものを表 6.4.1 に示す。システムコールの実装基準は Linux の ls コマンドや GUI 機能など実際のプログラムが本エミュレータ上で動作するために必要となる機能を優先的に扱って策定した。

NestedVM との連携によって POSIX システムコールのエミュレーションを行う場合は、NestedVM がサポートするシステムコールに依存するため、ここに示した POSIX ラッパーのサブセットが使用可能である。

表 6.4.1　本エミュレータ上で利用可能な POSIX 互換システムコール

exit	mkdir	sigprocmask	pause	getpeername
read	rmdir	nanosleep	utime	setsockopt
write	dup	sched_yield	access	
open	pipe	getcwd	nice	計 66 個
close	times	lseek	sync	
creat	setgid	poll	kill	
link	getgid	waitpid	rename	
unlink	geteuid	select	stat	
chdir	getegid	socket	fstat	
time	writev	bind	lstat	
mknod	readv	connect	ioctl	
chmod	uname	listen	gettimeofday	
lseek	mmap	accept	fcntl	
getpid	munmap	send	sigaction	
stime	mprotect	recv	recvfrom	
alarm	brk	sendto	getsockname	

Linux 独自のシステムコールのうち、本エミュレータがサポートするシステムコールの一覧を表 6.4.2 に示す。

表 6.4.2　本エミュレータ上で利用可能な Linux 独自のシステムコール

set_thread_area	futex	getdents
set_tid_address	set_robust_list	
getrlimit	statfs	計 9 個
clone	fstatfs	

6.5 GUI とアプレット

X ウィンドウシステムは多くの Linux 環境において GUI を実現するために用いられており、X サーバーと X クライアントからなる。このうち、X クライアントは、Linux 用の GUI アプリケーションがリンクして使用するライブラリである xlib に、その機能が集約されており、アプリケーションからの描画要求を X サーバーに転送している。X サーバーは X クライアントからの要求に応じて描画処理を行っている。

本エミュレータ上における GUI アプリケーションのサポートは、xlib をリンクした GUI アプリケーションがエミュレータ上で動作することを想定し、X サーバーについてはエミュレータの外部で動作する汎用のシステムを利用することを想定している。

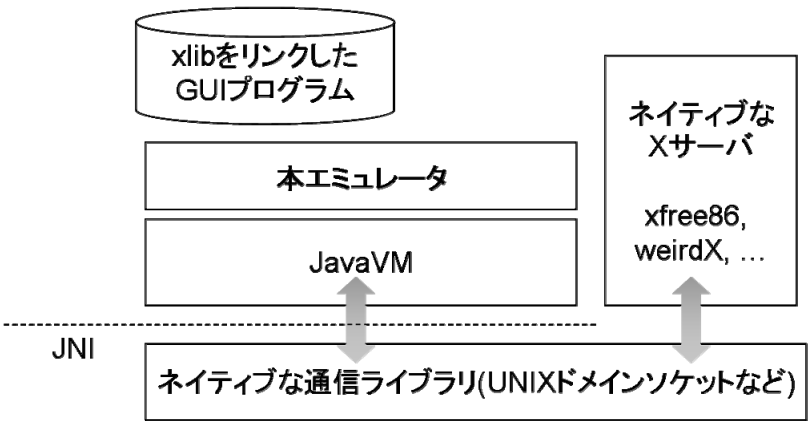

図 6.5.1 GUI アプリケーションの想定図

(1) GUI アプリケーション

xlib をリンクした x86/Linux 用 GUI アプリケーションは、X サーバーに対する描画要求を UNIX ドメインソケット、又は TCP/IP によって送信する。本エミュレータではシステムコール機能を JNI で提供した場合に限り、UNIX ドメインソケットの使用をサポートし、システムコールを Java 言語のランタイムライブラリでエミュレーションした場合は、TCP/IP のみをサポートする。X サーバーは本エミュレータの外部で汎用のシステムが動作していることを想定しているため、本エミュレータが GUI プログラムをサポートするために必要な機能は、xlib が動作するために必要なネットワーク機能に関するシステムコールのみである。

(2) X サーバーとの連携

本エミュレータにおける GUI 機能のサポートは、エミュレータの外側で汎用的な X サーバーが動作することを前提としており、エミュレータ上で Linux 用の X サーバープログラムが動作することは想定していない。一方で、Windows 環境のように、本来 X サーバーが動作していない環境においても、GUI アプリケーションを可搬的に実行するうえで、Java 言語で書かれた X サーバープログラムである WeirdX と本エミュレータを組み合わせることは有用である。

WeirdX は Java 言語と Swing ライブラリで書かれた X サーバーであり、動作環境の GUI ライブラリやハードウェアに依存しない。さらにウィンドウシステムを構成する上で必要な、ウィンドウマネージャに相当する機能が、Swing ライブラリによってサポートされるため、ユーザーは WeirdX 上が提供する GUI ウィンドウと、その他の通常のウィンドウを一元的に扱うことが出来る。

本エミュレータを WeirdX と連携させることにより、JavaVM が動作する様々な環境において、x86/Linux 用の GUI アプリケーションを実行することが可能になる。

(3) Java アプレット

本エミュレータと WeirdX を組み合わせ、Java アプレットとして展開することにより、既存の x86/Linux 用 GUI アプリケーションを、Web ブラウザ上で実行することが可能になる。Web アプリケーションを構成するための言語・ライブラリ・実行環境等は数多くあるが、古くからある既存のデスクトップアプリケーションに変更を加えることなく Web ブラウザ上で実行することは難しく、またデスクトップアプリケーションと Web アプリケーションを共通の開発手法で実現することは有用である。

Web ブラウザ上で動作する x86/Linux アプリケーションが使用するシステムコールの発行先は、Web ブラウザが動作している環境の OS、又は JavaVM であり、先に述べたシステムコール資源の仮想化オプションによって決まる。Web アプリケーションがクライアント環境のシステムコール資源を使用することはセキュリティ上好ましくないと言われているが、一方で ActiveX や Google Native Client といった技術が普及していることを考えれば、利便性とのトレードオフであると言える。本エミュレータが提供する POSIX システムコールの Java ランタイムライブラリによるエミュレーション機能を用いれば、クライアント環境のシステムコール利用における危険性は、サンドボックス化された JavaVM 上での Java アプレット実行時の問題と同義となるため、ユーザーに対し一定レベルの解決策を提供すると考えている。

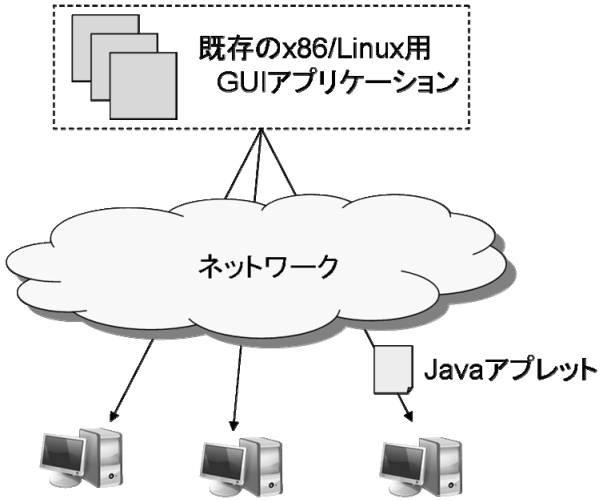

図 6.5.2 アプリケーションの Web 配信

6.6 命令レベル統計情報

本エミュレータには、どのような命令を何回実行したかを記録する機能が備わっている。また、記録した情報を元に命令の使用頻度や全体実行時間に占める割合などを解析する機能も有している。本エミュレータの実装を進める上で、この統計情報をもとに使用頻度の高い命令から最適化を進めた。また、まだ実現はできていないものの、統計情報をもとに命令ディスパッチの最適化を行うことを考えている。

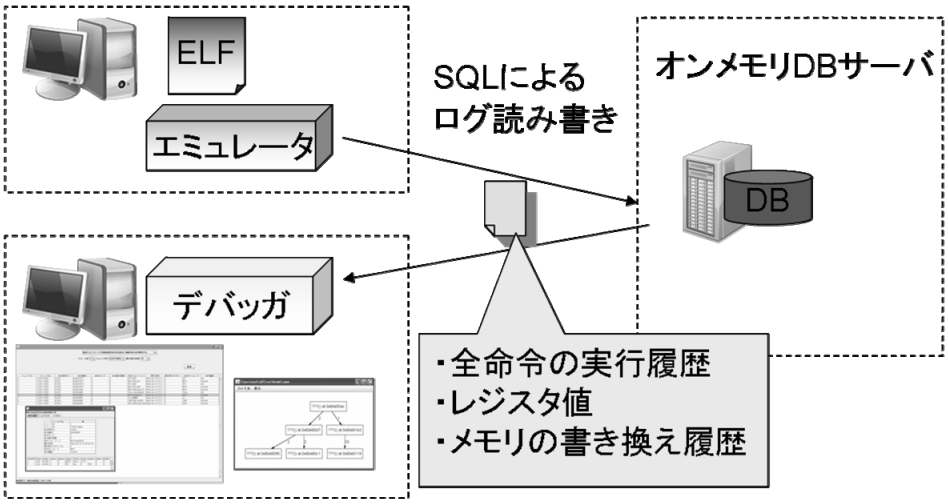

図 6.6.1　命令レベルでの統計機能

6.7　NestedVM との連携機構

NestedVM に実装された POSIX 互換システムコールのエミュレーション機能を利用するため、筆者は JNI 機能を利用する POSIX ラッパーが NestedVM 上で動作することを期待した。これは本エミュレータの実装を簡略化し、POSIX ラッパーに施した変更が、NestedVM との連携動作においても即座に反映されることを意図したものである。

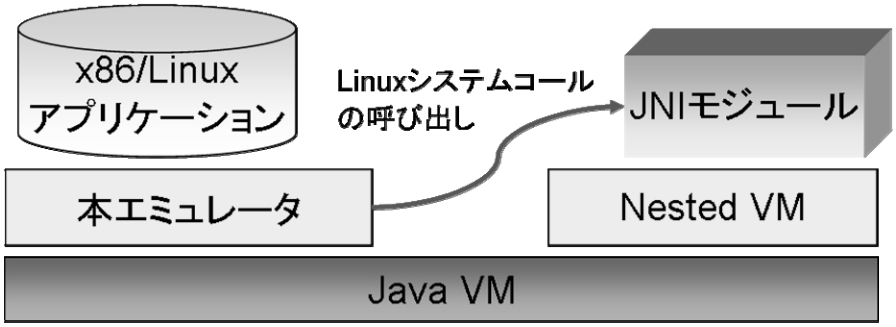

図 6.7.1　NestedVM を用いた JNI エミュレーション

表 6.7.1 には NestedVM がサポートするシステムの一覧を示す。

表 6.7.1 NestedVM がサポートするシステムコール

exit	pipe	lchown	recvfrom	fchmod
pause	dup2	ftruncate	select	alarm
open	fork	usleep	umask	getgroups
close	waitpid	getppid	getuid	setsid
read	__getcwd_r	mkfifo	geteuid	__resolve_ip_r
write	__execve_r	klogctl	getgid	fsync
sbrk	fcntl	realpath	getegid	call_java
fstat	rmdir	__sysctl_r	send	計 87 個
lseek	sysconf	getpriority	recv	
kill	readlink	setpriority	getsockname	
getpid	lstat	socket	getpeername	
stat	symlink	connect	setuid	
gettimeofday	link	__resolve_hostname_r	seteuid	
sleep	getdents	accept	setgid	
times	memcpy	setsockopt	setegid	
mkdir	memset	getsockopt	setgroups	
getpagesize	dup	listen	access	
unlink	vfork	bind	chown	
utime	chroot	shutdown	fchown	
chdir	mknod	sendto	chmod	

(1) NestedVM の外部インターフェース

NestedVM の外側にある Java クラスから、NestedVM 上の資源にアクセスするインターフェースとして以下の API が提供されている。このうち、64bit 値を返す call64()については今回独自に実装した。

表 6.7.2　NestedVM の外部 API

void memWrite(int addr, int value)	4 バイトメモリ書き込み
int memRead(int addr)	4 バイトメモリ読み込み
int call(String sym, int[] args)	戻り値が 32bit までの C 言語関数の呼び出し
long call64(String sym, int[] args)	戻り値が 64bit までの C 言語関数の呼び出し
void setCallJavaCB(CallJavaCB callJavaCB)	SYS_calljava で呼ばれるクラスの登録

NestedVM では MIPS 用 ELF バイナリのシンボルテーブルを解析し、ELF バイナリ内の任意の関数を、関数名から呼び出すための機能を提供している。したがって、malloc(),free()といった関数を NestedVM の外側から呼び出すことで、高機能なメモリ操作も実現できる。

(2) NestedVM の内部インターフェース

NestedVM によってエミュレーションされる MIPS コードの中から、NestedVM の外側に位置する JavaVM の資源にアクセスするための機能も提供されている。表 6.7.3 に示す_call_java 関数は、NestedVM が提供する独自のシステムコールである SYS_call_java システムコールのラッパーであり、NestedVM 専用の crt0.o のオブジェクトファイルに含まれている。

SYS_call_java システムコールは、事前に設定された NestedVM の外側に位置する Java クラスの call メソッドを呼び出す。

表 6.7.3　NestedVM の内部 API

int _call_java(int a0, int a1, int a2, int a3)	Java クラスの呼び出し(C 言語)
#define SYS_calljava 13	_call_java()の実体となるシステムコール

(3) JNIEnv に相当する機能

JNI では、ネイティブコードから JavaVM 上の資源を操作するために C++で実装された JNIEnv クラスを提供している。Java クラスからネイティブコードを呼び出した際に、ネイティブコード側では、この JNIEnv クラスへのポインタが JavaVM から渡される。

今回、JNIEnv クラスを継承した JNIEnvEx クラスを新たに定義し、NestedVM 上の JNI モジュールから Java 言語の資源を操作するための機能を提供することとした。通常の JNIEnv クラスがネイティブコードから JavaVM 上の資源を操作するのに対し、この JNIEnvEx クラスは JavaVM 上で全ての操作が完結する。

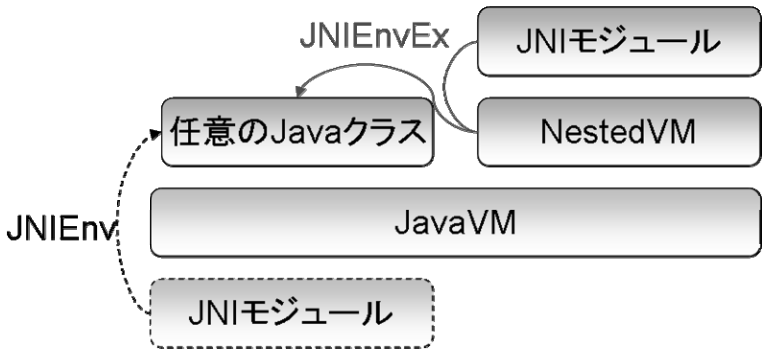

図 6.7.2　JNIEnvEx の概要

JNIEnvEx には、JNI1.1 で定義された API の大半を実装した。表 6.7.4 に API の実装状況を示す。実装できなかった API として、例外機能、特殊なオブジェクトの管理、JavaVM のモニター機能が挙げられる。例外機能は NestedVM 内部で例外を発生させた場合に、NestedVM の実装上、途中でハンドリングされる可能性が高い。特殊なオブジェクト管理では、Java オブジェクトをインスタンス化せずに必要なメモリ空間の確保だけを行う AllocObject()が該当する。JavaVM のモニター機能は、JavaVM に関する情報を JavaVM の内部から収集することが困難であるため、実装不可である。なお、これら 3 種類の実装困難な API は、今回対象としている POSIX ラッパーでは使用していない。

表 6.7.4　JNIEnvEx に実装された API の一覧

JNI1.1 の API と実装状況　○:実装済み X:実装不可					
文字列操作		インスタンスメソッド		バージョン情報	
NewString	○	GetMethodID	○	GetVersion	○
GetStringLength	○	Call<type>Method	○	static フィールドアクセス	
GetStringChars	○	Call<type>MethodA	○	GetStaticFieldID	○
ReleaseStringChars	○	Call<type>MethodV	○	GetStatic<type>Field	○
NewStringUTF	○	フィールドアクセス		SetStatic<type>Field	○
GetStringUTFLength	○	GetFieldID	○	クラス操作	
GetStringUTFChars	○	Get<type>Field	○	DefineClass	○
ReleaseStringUTFChars	○	Set<type>Field	○	FindClass	○
配列操作		オブジェクトオペレーション		GetSuperclass	○
GetArrayLength	○	AllocObject	×	GetSuperclass	○
NewObjectArray	○	NewObject	○	static メソッドの呼び出し	
GetObjectArrayElement	○	NewObjectA	○	GetStaticMethodID	○
SetObjectArrayElement	○	NewObjectV	○	CallStatic<type>Method	○
New<PrimitiveType>Array	○	GetObjectClass	○	CallStatic<type>MethodA	○
Get<PrimitiveType>ArrayElements	○	IsInstanceOf	○	CallStatic<type>MethodV	○
Release<PrimitiveType>ArrayElements	○	IsSameObject	○		
Get<PrimitiveType>ArrayRegion	○	グローバル参照とローカル参照			
Set<PrimitiveType>ArrayRegion	○	NewGlobalRef	○		
ネイティブメソッドの登録		DeleteGlobalRef	○		
RegisterNatives	○	DeleteLocalRef	○		
UnregisterNatives	○	例外			
モニターオペレーション		Throw	×		
MonitorEnter	×	ThrowNew	×		
MonitorExit	×	ExceptionOccurred	×		
Java VM インタフェース		ExceptionDescribe	×		
GetJavaVM	○	ExceptionClear	×		
		FatalError	×		
		IsAssignableFrom	○		

(4) JNI コードの自動生成

NestedVM 上での JNI コードのエミュレーションに伴う変更作業を軽減する目的で、以下に示す 2 種類の Java コードの自動生成を行った。

・NestedVM を継承しネイティブコードをエミュレーションするエミュレーションクラス

・JNI 関数の基底クラスを継承し、NestedVM に対する呼び出しを行う派生クラス

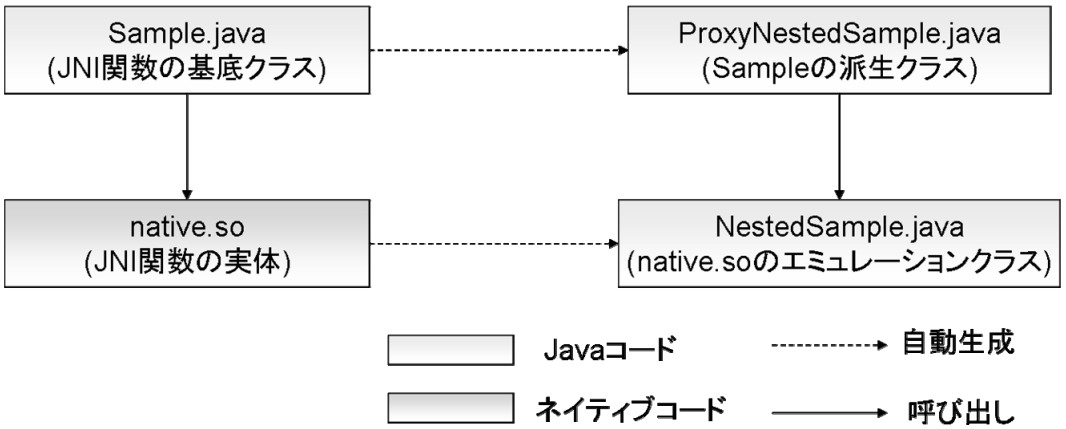

図 6.7.3 自動生成コードの対応関係

エミュレーションクラスは NestedVM を継承し、ネイティブコードモジュール内の命令列も含まれた単体でロード可能な Java クラスである。

派生クラスは、JNI 関数の呼び出しを担当していた基底クラスを継承し、JNI 関数に対する呼び出しをエミュレーションクラスへの呼び出しへと変更したものである。

基底クラスの外部から JNI 関数を呼び出している場合、JNI 関数がインスタンスメソッド（非 static メソッド）として定義されている場合には、Java 言語の多態性により、本作業の影響をほとんど受けない。基底クラスをインスタンス化していた部分を派生クラスのインスタンス化を行うよう変更するだけで済む。

一方で、静的メソッド(static メソッド)として JNI 関数が定義されている場合、継承による多態性の効果は見込めない。

第 7 章　本エミュレータの実現

7.1 エミュレーション機能の実現

本エミュレータを実現するうえで、最も重要視したのはエミュレータが提供する異種 OS 間における x86/Linux 環境との互換性であり、その実現度合いは簡易な GUI プログラムや、共有ライブラリ、マルチスレッド機能のサポートいった点に現れている。特に Windows 環境において、このような機能を使用する x86/Linux プログラムがユーザーモードエミュレーションによって動作した例は無く、本エミュレータの有用性を示すものであると考えている。本章ではエミュレータの機能レベルでの実現度合いと、実装上の工夫を示す。

本エミュレータの命令セットエミュレーションの基本的な流れを図 7.1.1 に示す。

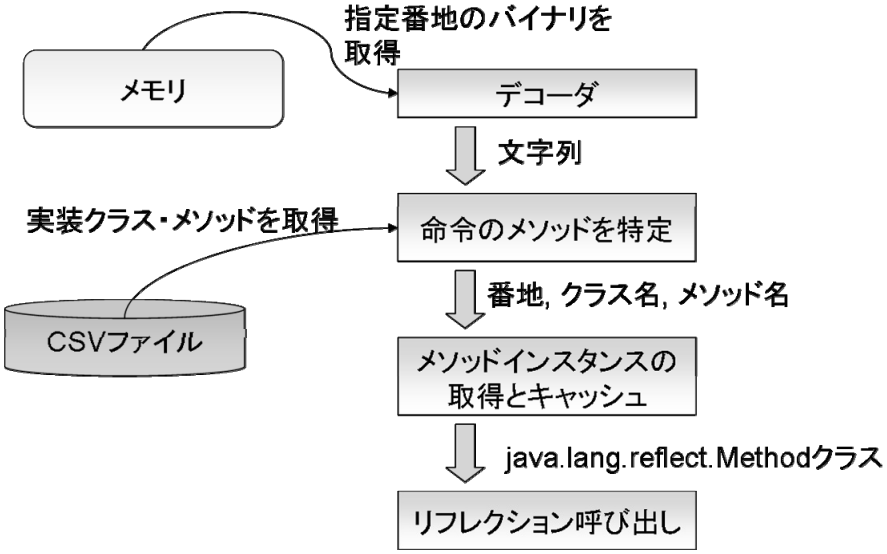

図 7.1.1　命令セットエミュレーション

7.2 本エミュレータを構成する Java クラス

本エミュレータの主要部分を構成する Java 言語で書かれたソースコードは、112 個のファイルに分散した約 6 万行のコードで構成されている。この中には x86 の非特権命令やメモリ空間のエミュレーションコード、ELF ローダー、Linux が提供するシステムコールのエントリなどが含まれている。

また、上記のソースコードとは別に POSIX システムコールを JNI 経由で呼び出す部分は Java で書かれたインターフェース部分が約 3 千行、C++で書かれたラッパールーチンが 7 千行である。NestedVM とのインターフェースを構成し、JNIEnv に相当する処理を行う部分は 3 千行である。この中にはオリジナルの NestedVM を構成するコードは含まれていない。

そのほか、一部をオープンソースソフトウェアの改良によって済ませた命令デコーダや、命令レベルテストの生成系、エミュレーションした命令列をデータベースに格納し解析するための機能などがある

7.3 エミュレータのテスト

エミュレータの命令セットに潜むバグを検出するには手間を要する。本エミュレータでは、命令レベルでのテスト検証を自動化するため、幾つかのツールを用意した。エミュレーションする対象のプログラムが実行した全命令をログに書き出し、オペランド間の依存関係を自動解析するツールや、命令の種類と、オペランドの初期値の集合から、取りうる命令列を全てアセンブラコードとして書き出して、命令実行後のレジスタ値を出力するようなコードの生成系などである。これらのテストツールが自動で検出した命令のバグは、数十個に上る。本エミュレータの実現過程において、エミュレータのデバッグに費やした時間はエミュレータのコードを記述していた時間の数十倍に上るため、テストを自動化する意義は大きい。

以下に命令レベルテスト生成系の入力フォーマットの一例を示す。検査項目を指定する CSV ファイルの一部を、ここでは抜粋した。一行の入力が一種類の命令のテストプログラム生成に対応している。”|”記号で区切られたパラメータはテスト前に設定する初期状態が複数存在することを示しており、項目ごとのパラメータの個数を全て掛け合わせた回数のテストが実行される。また、検査対象オペランドタイプでは、この命令が取りうるオペランドの組み合わせのうち、指定した種類のオペランド全てを検査する。したがって、ここに示した 1 行の入力パターンだけで、mov 命令に対し数百通りのテストを生成する。また、ここには示さなかったが、命令コードの入力データも存在する。これは x86 命令セットの全命令と全オペランドの組み合わせからなるデータであり、検査項目で指定した命令、オペランド数、オペランドタイプに該当する x86 命令を本テスト生成系が識別するのに使用している。

テスト名	命令	オペランド数	op1 初期値	op2 初期値	op3 初期値	フラグレジスタ検査項目	検査対象オペランドタイプ	汎用初期化コード
mov2	mov	2	0x00|0xff	0x00|0x11	N/A	cf|sf	reg|mem	eax=0x01|0x02

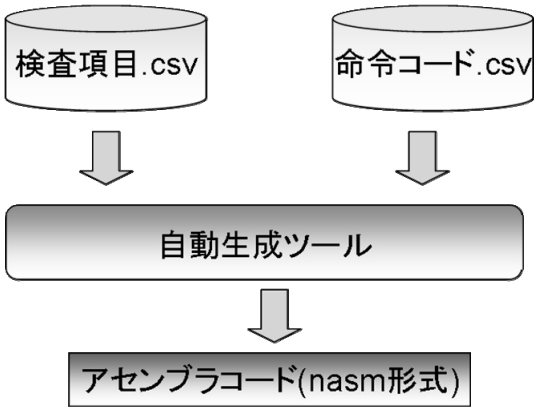

図 7.3.1 命令レベルテストの自動生成ツール

高級言語のコンパイラが出力するような複雑な制御構造をもった命令列の実行は、命令レベルテストでは検証しきれなかったエミュレータのバグに遭遇する場合がある。本エミュレータでは、

汎用コンパイラのテストケースを用いて、switch 文や構造体メンバ変数の型キャストといった操作が適切に実行可能であるかを調べた。

coins コンパイラに付属する約 800 個のテストプログラムを実行し、8 件のエミュレータのバグに起因する不具合を発見した。以下にその内訳を示す。

未実装命令	“fdivrp st(i)”	:	3
変数の型キャスト(movzx)		:	3
SUB 命令の SF フラグ設定		:	1
負数の剰余演算(idiv)		:	1

また、汎用のコンパイラテスト生成系である testgen が提供する約 8000 個のテストプログラムでは、59 件のエラーを確認した。以下にその内訳を示す。

- 構造体メンバ変数の型キャスト	:	7
- 変数の型キャスト	:	15
- 大域変数の型キャスト	:	1
- 静的変数の型キャスト	:	1
- NOT 演算	:	8
- float 型の除算	:	5
- 関数ポインタ	:	5
- short 型変数のポインタによる代入	:	15
- #define による定数値とその使用	:	2

第 8 章 評価

本研究における評価として、エミュレータの性能に基づいた定量的な評価と、エミュレータが実際にサポートするアプリケーションに基づく定性的な評価を示す。性能評価の項目では、先に示した本エミュレータの最適化機能のうち、命令デコードキャッシュ、メモリブロックキャッシュが有効になっている。

8.1 性能評価

本エミュレータの演算性能と、read/write といったアプリケーションが使用する I/O 性能の評価を行った。

8.1.1 演算性能

本エミュレータの基本的な演算性能を測るため、以下のベンチマークプログラムを用意した。

・フィボナッチ数列
・空のfor ループ
・繰り返しの無いコード

ベンチマークの選定では、C言語で書かれた一般的なx86/Linuxアプリケーションの制御構造が顕著に反映されることを意図した。したがって、汎用的なベンチマークではなく、ループや関数呼び出しにおける特性が顕著に反映される、これらのプログラムを選定した。

測定環境は、VMware 上のFedora8 であり、各ベンチマークプログラムはC言語、又はインラインアセンブラで記述され、GCC4.1.2を用いてコンパイルされている。各ベンチマークプログラムは、その内部でx86プロセッサのサイクルカウンタの値を取得する。本エミュレータ上でのサイクル数の取得は、JNIを用いて実機のサイクル数を取得することとした。ベンチマークプログラムが実行に要したサイクル数の、本エミュレータと実機環境との比率を性能指標とする。

(1) フィボナッチ数列

表8.1.1 にフィボナッチ数列を計算するプログラムの実行結果を示す。本システム上での実行に要したサイクル数は、ネイティブな環境で実行した場合の2万～7 万倍程度であり、性能が大幅に低下している。サイクル比は25 要素まで増加傾向にあるが、それ以降は減少に転じている。これは、25 要素目以降において、本システムのサイクル数の増加率がネイティブ環境の増加率を大きく下回るためである。

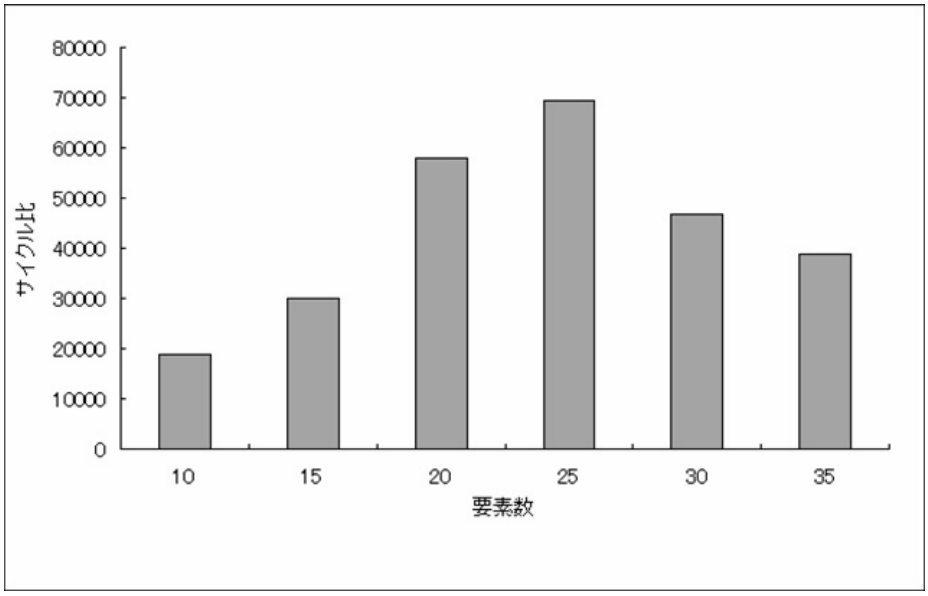

図8.1.1　フィボナッチ数列の性能比

表8.1.1 フィボナッチ数列の性能

要素数	10	15	20	25	30	35
本システム (サイクル数)	4.47×10^{9}	9.98×10^{9}	6.43×10^{10}	6.66×10^{11}	7.38×10^{12}	8.08×10^{13}
ネイティブ (サイクル数)	2.38×10^{5}	3.31×10^{5}	1.11×10^{6}	9.61×10^{6}	1.58×108	2.08×10^{9}
サイクル比	**1.87×10^{4}**	**3.01×10^{4}**	**5.79×10^{4}**	**6.93×10^{4}**	**4.67×10^{4}**	**3.87×10^{4}**

(2) 空のfor ループ

表8.1.2に空のfor ループの実行結果を示す。本プログラムは、C 言語によって生成されるforループであり、ループ文内部では何も行わない。サイクル比は10 万～13 万倍程度に安定して分布しており、次節で示す分岐の無いコードよりも良い結果を示している。これは命令のデコード結果をキャッシュし再利用する機能によるものである。

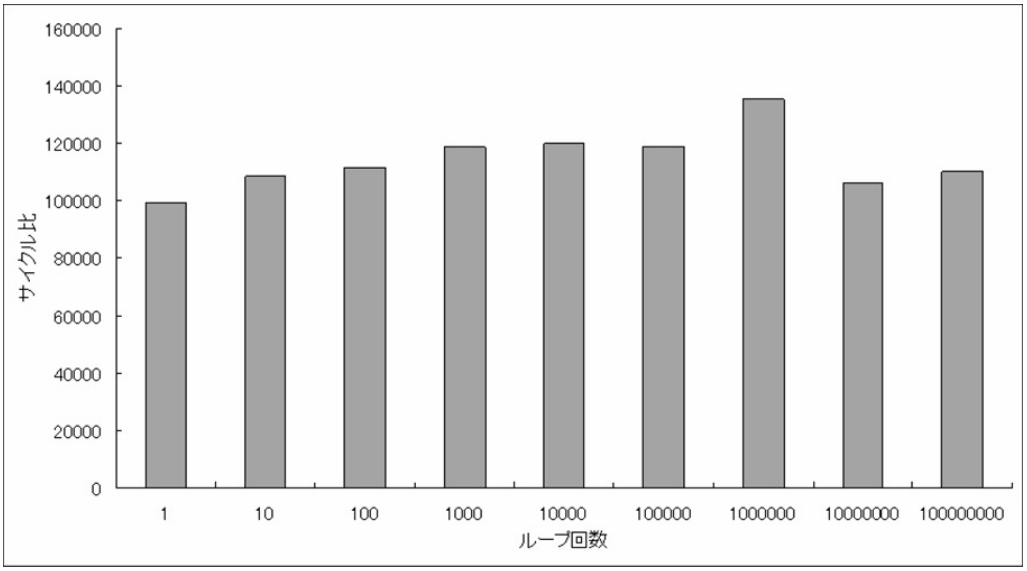

図8.1.2　空のforループの性能比

表8.1.2 空のforループの性能

ループ回数	1	10	100	1000	1万	10万	100万	1000万	1億
本システム(サイクル数)	1.6×10^8	1.9×10^8	3.4×10^8	1.8×10^9	1.6×10^{10}	1.6×10^{11}	1.6×10^{12}	1.6×10^{13}	1.6×10^{14}
ネイティブ(サイクル数)	1.6×10^3	1.8×10^3	3.1×10^3	1.5×10^4	1.4×10^5	1.3×10^6	1.2×10^7	1.5×10^8	1.4×10^9
サイクル比	**9.9×10^4**	**1.0×10^5**	**1.1×10^5**	**1.1×10^5**	**1.2×10^5**	**1.1×10^5**	**1.3×10^5**	**1.0×10^5**	**1.1×10^5**

(3) 繰り返しの無いコード

このプログラムは、分岐命令を一切使用せず、ADD命令を線形に並べたコードである。ADD命令を並べる個数を変動して測定した。表8.1.3に実行結果を示す。サイクル比に基づく性能は、今回試したベンチマークの中で最も悪く、14 万～42 万倍程度で増加傾向にある。これは本システムが命令デコードの結果をキャッシュすることで、ループや再帰処理における性能を上げていたのに対し、分岐の無いコードを1 回だけ実行する本プログラムでは、この機能が効果を成さないためである。先に示したフィボナッチ数列や空のfor ループと本プログラムとの比較から、命令デコードの結果をキャッシュしない場合、本システムは最大で20 倍程度の性能の低下が見られることが分かった。

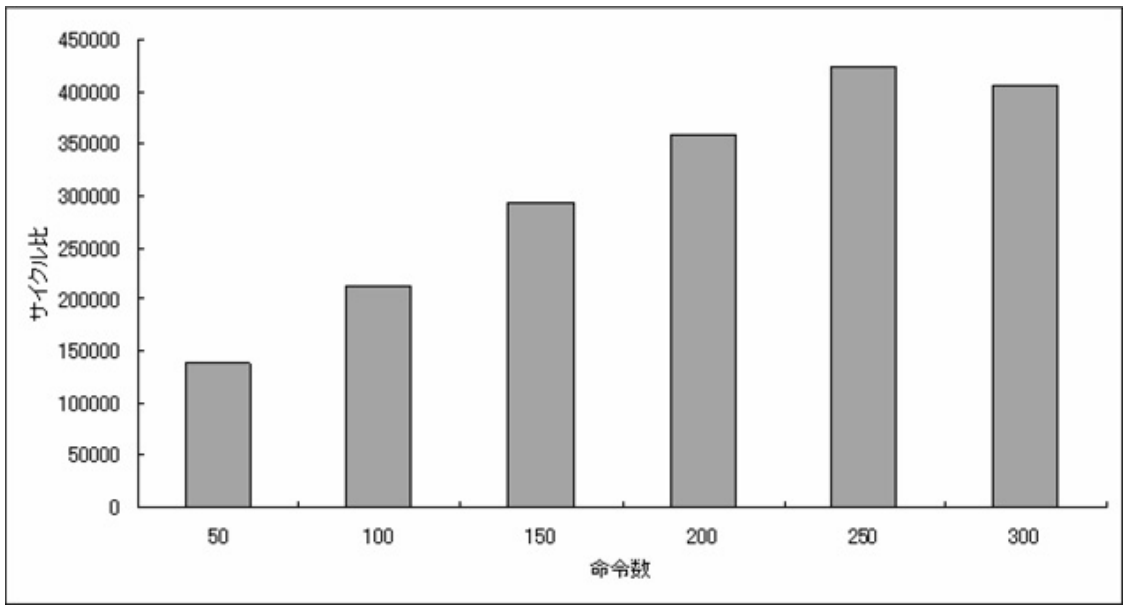

図8.1.3　繰り返しのないコードの性能比

表8.1.3　繰り返しのないコードの性能

命令数	50	100	150	200	250	300
本システム(サイクル数)	2.0×10^{8}	3.4×10^{8}	4.9×10^{8}	6.3×10^{8}	7.8×10^{8}	9.2×10^{8}
ネイティブ(サイクル数)	1.4×10^{3}	1.6×10^{3}	1.6×10^{3}	1.7×10^{3}	1.8×10^{3}	2.2×10^{3}
サイクル比	**1.3×10^{5}**	**2.1×10^{5}**	**2.9×10^{5}**	**3.5×10^{5}**	**4.2×10^{5}**	**4.0×10^{5}**

8.1.2　I/O 性能

本エミュレータが想定する一般的な x86/Linux アプリケーションの実行では、演算性能以外にもシステムコールを介した I/O 処理に伴う性能を考慮する必要がある。このため以下の内容の入出力を行うベンチマークプログラムを用意した。

1)/dev/zero, /dev/null に対する read/write

2)ディスクに対する read/write

3)dd コマンド

1)ではディスクへの読み書きを伴わない read/write システムコールに対し、本エミュレータの POSIX ラッパーの処理性能を示す。2)ではディスク上のファイルに対し、読み書きを行う。3)では dd コマンドを用いてディスクへの読み書きを伴わない入出力を行った。

演算性能での評価と同様に、本エミュレータ上、ならびに x86/Linux 実機上でのベンチマークの実行に要したサイクル数の比率から性能を論じる。

1)-1 /dev/zero からの read

/dev/zero に対し read システムコールを 1000 回実行し、読み込むバイト数を変動した。実行に要した CPU サイクル数から本エミュレータと実機との性能比率を表 8.1.4 に示す。

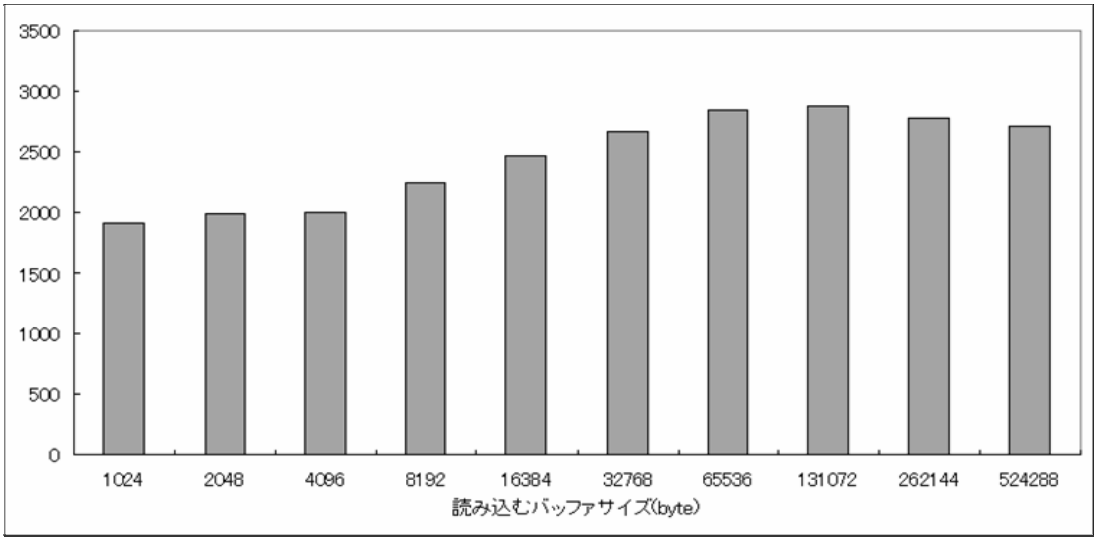

図 8.1.4 /dev/zero に対する read

表 8.1.4 /dev/zero に対する read システムコール

バッファサイズ(byte)	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288
サイクル比	**1919**	**1998**	**2008**	**2250**	**2471**	**2671**	**2843**	**2876**	**2781**	**2710**

1)-2 /dev/null に対する write

/dev/null に対し write システムコールを 1000 回実行し、書き込むバイト数を変動した。

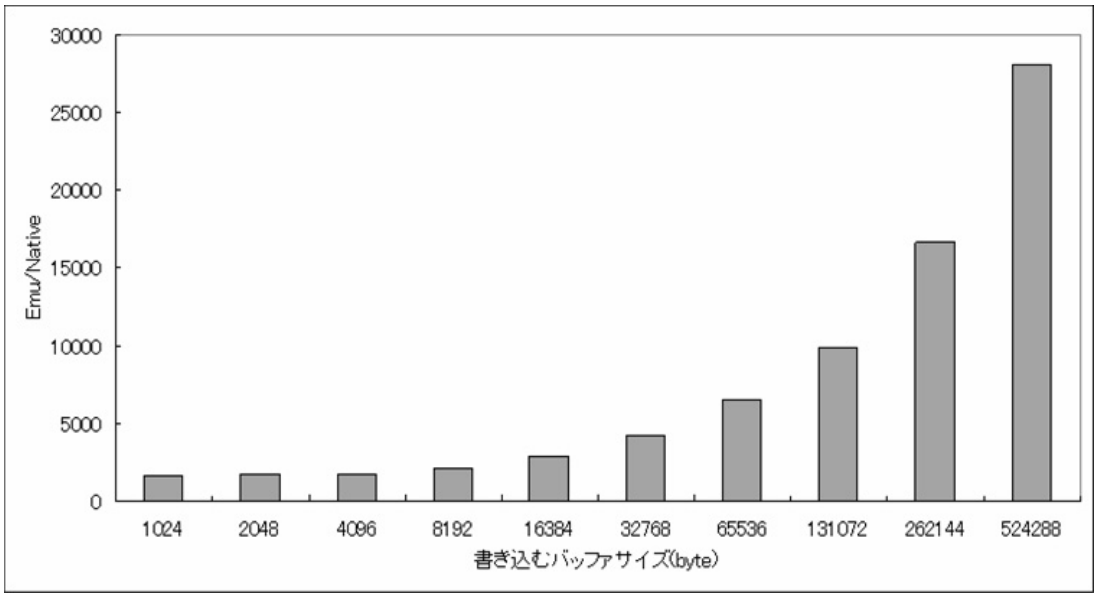

図 8.1.5 /dev/null に対する write

表 8.1.5 /dev/null に対する write システムコール

バッファサイズ(byte)	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288
サイクル比	**1618**	**1688**	**1726**	**2128**	**2928**	**4246**	**6576**	**9930**	**16663**	**28042**

2)-1ファイル読み込み

プログラムは、open(),read(),close()の一連の操作を1000 回繰り返すものであり、読み込むデータのサイズを変化させて測定を行った。表8.1.6に実行結果を示す。サイクル比に基づく性能低下は600～1300 倍程度となっている。他の評価に比べ、比較的よい性能を示しているが、これはディスクアクセスのオーバーヘッドが大きいため、実機と本エミュレータで処理時間の差異が小さいことによる。しかし、実際のアプリケーションを想定した場合、必ずしも演算性能だけが重要でないことを、本ベンチマークの結果は示唆している。

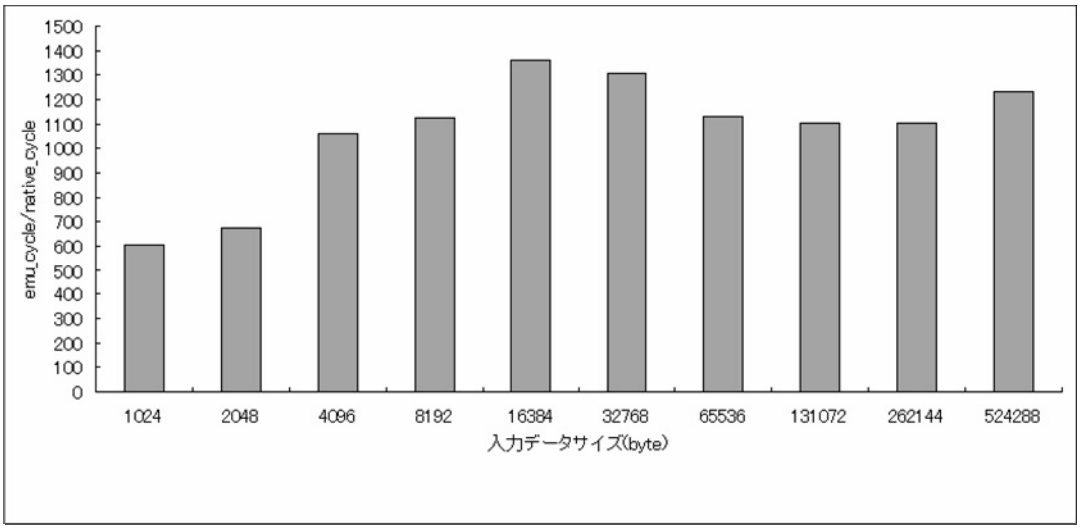

図8.1.6 ファイル入力の性能

表8.1.6 ファイル入力の性能

バッファサイズ(バイト)	1K	2K	4K	8K	16K	32K	64K	128K	256K	512K
本システム(サイクル数)	7.335 ×10^9	8.007 ×10^9	9.167 ×10^9	1.160 ×10^{10}	1.628 ×10^{10}	2.527 ×10^{10}	4.312 ×10^{10}	8.002 ×10^{10}	1.551 ×10^{11}	3.007 ×10^{11}
ネイティブ(サイクル数)	1.209 ×10^7	1.191 ×10^7	8.646 ×10^6	1.031 ×10^7	1.197 ×10^7	1.934 ×10^7	3.822 ×10^7	7.241 ×10^7	1.404 ×10^8	2.441 ×10^8
サイクル比	**606.8**	**672.4**	**1060**	**1126**	**1360**	**1306**	**1128**	**1105**	**1105**	**1232**

2)-2 ファイル書き込み

ファイル読み込みと同様に,open(),write(),close()の一連の操作を1000 回繰り返し、書き込むデータのサイズを変化させた。表8.1.7 に実行結果を示す。サイクル比は350～630倍程度と、前節で示した読み込み性能と比べても、さらに2 倍近く向上している。

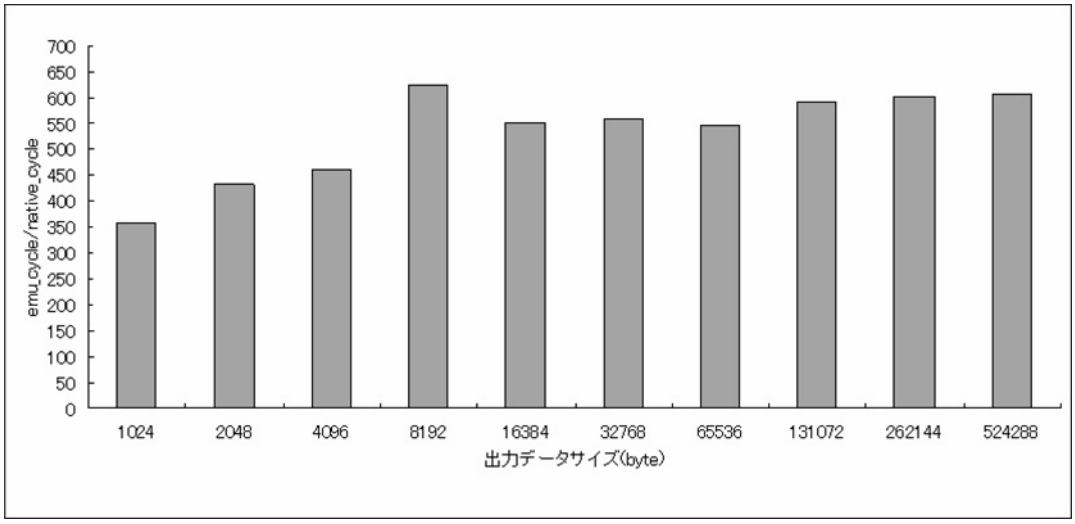

図 8.1.7 ファイル出力の性能

表 8.1.7 ファイル出力の性能

バッファサイズ（バイト）	1K	2K	4K	8K	16K	32K	64K	128K	256K	512K
本システム（サイクル数）	6.178 ×10^9	6.722 ×10^9	8.024 ×10^9	1.029 ×10^{10}	1.515 ×10^{10}	2.399 ×10^{10}	4.224 ×10^{10}	7.775 ×10^{10}	1.557 ×10^{11}	3.068 ×10^{11}
ネイティブ（サイクル数）	1.724 ×10^7	1.551 ×10^7	1.745 ×10^7	1.646 ×10^7	2.747 ×10^7	4.287 ×10^7	7.764 ×10^7	1.310 ×10^8	2.584 ×10^8	5.058 ×10^8
サイクル比	**358.4**	**433.3**	**460**	**625**	**551.7**	**559.5**	**544**	**593.5**	**602.7**	**606.5**

3)dd コマンド

dd コマンドによる I/O 入出力の性能を計測した。/dev/zero からの入力を/dev/null に出力するものとし、読み書きするサイズを変動した。ここでは更に静的リンクされた busybox の dd コマンドと、共有ライブラリを用いる coreutils の dd コマンドをそれぞれ実行し比較する。

表 8.1.8 に測定結果を、図 30 に実機に対する本エミュレータの性能低下を示す。本エミュレータ上で共有ライブラリを使用する dd コマンドを実行した場合、実機に比べ、約 2500 倍から 8500 倍程度の性能低下を確認した。また静的リンクされた dd コマンドの場合、約 1300 倍から 3800 倍程度の性能低下を確認した。この性能は先に示した read/write の基本性能が、実際のアプリケーションプログラムにおいても同様であることを示唆している。また、エミュレータ上で共有ライブラリをロードする場合、静的リンクされたプログラムをロードする場合に比べ 2 倍程度の性能低下が伴うことを示している。

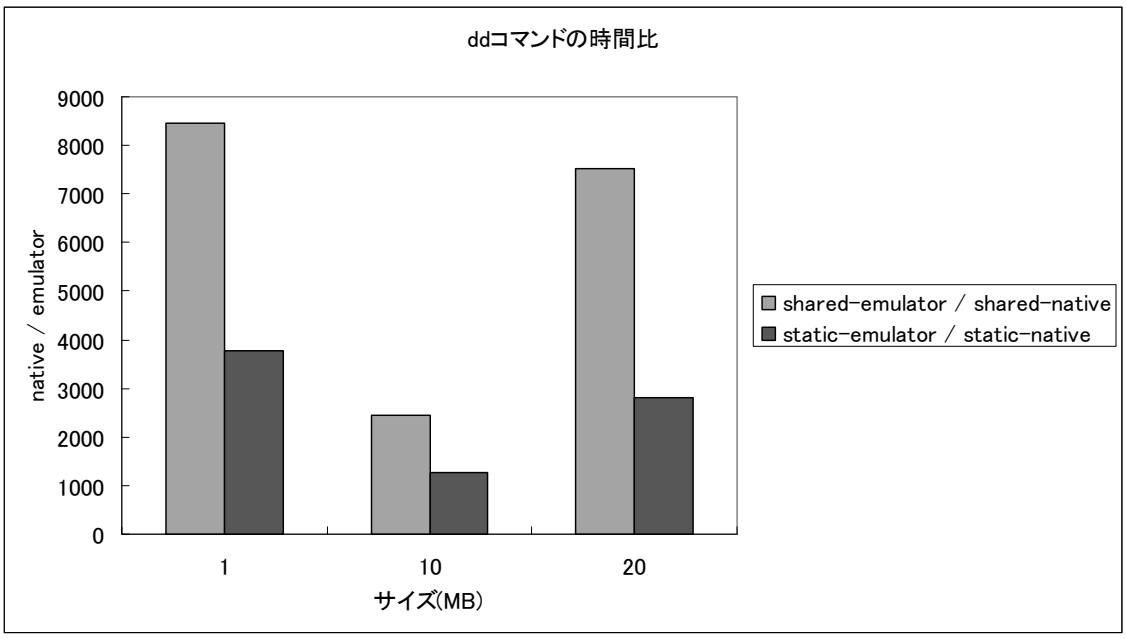

図 8.1.8 dd コマンドの性能

表 8.1.8 dd コマンドの性能

バイトサイズ(MB)	1	10	20
shared-native (s)	0.005	0.021	0.046
shared-emulator (s)	42.298	51.454	346.059
shared-emulator / shared-native	8459.6	2450.19	7523.022
static-native (s)	0.004	0.016	0.046
static-emulator (s)	15.086	20.492	128.93
static-emulator / static-native	3771.5	1280.75	2802.826

8.2 サポートするアプリケーション

本エミュレータがサポートするアプリケーションの具体例を示す。これらのプログラムは、x86/Linux 環境で実行可能な、共有ライブラリを使用するプログラムであり、エミュレータ上において実機と同様の実行結果が得られることを確認している。

(1)Linux コマンド

Linux に標準で添付されている 3 つのコマンドプログラムの実行時性能を測定した．本エミュレータ上での実行に要した時間を実機上での実行時間で割ったものを表 8.2.1 に示す．ls は 15 個のファイルをオプションなしで表示した．wc は 112 個のファイルに分散した Java 言語のソースファイル 6 万行程度の行数を取得した．gzip は約 7kb のファイルを圧縮した．

大幅な性能低下が見られるが，これらの実際的なプログラムではシステムコールを多用するた

め，先に示した基本性能と比べ良い性能を示した.

表 8.2.1 Linux コマンドの性能

	ls	wc	gzip
本エミュレータ(s)	5.460×10^{1}	3.262×10^{4}	7.736×10^{1}
実機(s)	5.0×10^{-3}	5.920×10^{-1}	6.000×10^{-3}
時間比	1.092×10^{4}	5.511×10^{4}	1.289×10^{4}

(2) シグナルとマルチスレッド

シグナル機能の一例としてキーボードから Ctrl+C で SIG_INT を入力されるとメッセージを表示するプログラムを実行した。JavaVM のプロセスに対し SIG_INT を発行した場合、通常はプロセス自体が終了するが、適切にマスクして処理することにより問題がないことを確認した。

スレッド機能の例として、複数のスレッドを起動後、join する操作を繰り返すプログラムを作成し、本エミュレータ上で実行した。図 8.2.1 に JavaVM 上でのスレッドの状態遷移を示す。

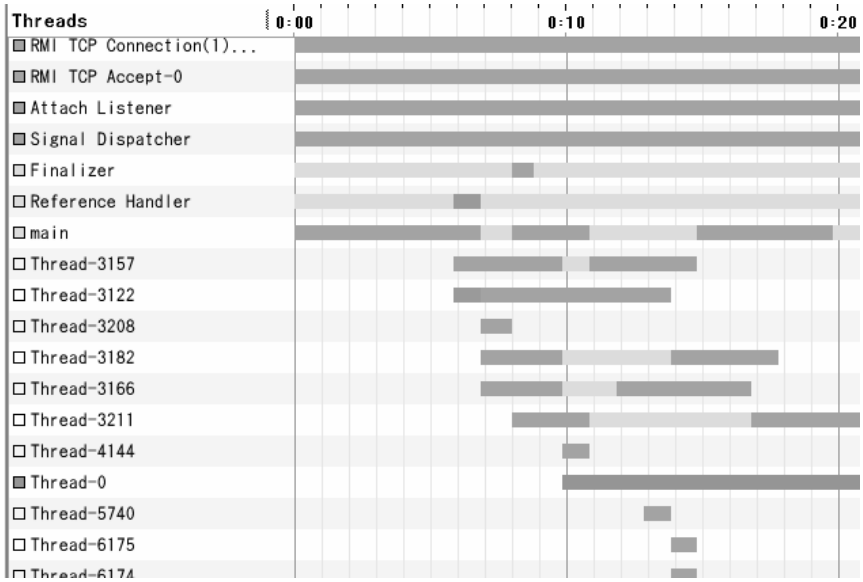

図 8.2.1　マルチスレッドプログラムエミュレーション時の JavaVM 上での状態遷移

(3) GUI とアプレット

以下では、GUI プログラムをエミュレートした際の 3 種類の実行形態について示す。図 8.2.2 は Windows 上で x86/Linux 用の GUI プログラムをエミュレートしたものである。図 8.2.3 は Web ブラウザ内の画面でアプレットとして実行したもの、図 8.2.4 は同じくアプレットだが、ブラウザ外にウィンドウを取り出して実行したものを示した。ブラウザ内に描画する場合、ウィンドウマネージャを付加していないため、ウィンドウの拡縮といった基本操作を実現できていない。

JavaVM 上で動作するウィンドウマネージャは looking glass や puppet があるが、weirdX との X プロトコルのバージョン整合が取れず併用は難しい。

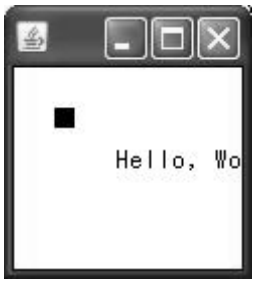

図 8.2.2 デスクトップ上での描画

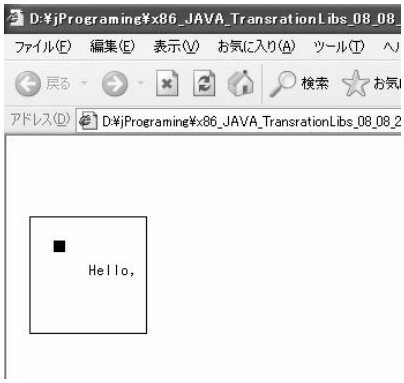

図 8.2.3 ブラウザ内描画

図 8.2.4　ブラウザ外描画

図 8.2.5 には簡易な GUI プログラムをブラウザ上で実行した際の画面を示す。本プログラムはマウス入力をもとに描画を更新するものであるが、Web ブラウザ上においてもユーザーからの入力を適切に処理できることを確認した。

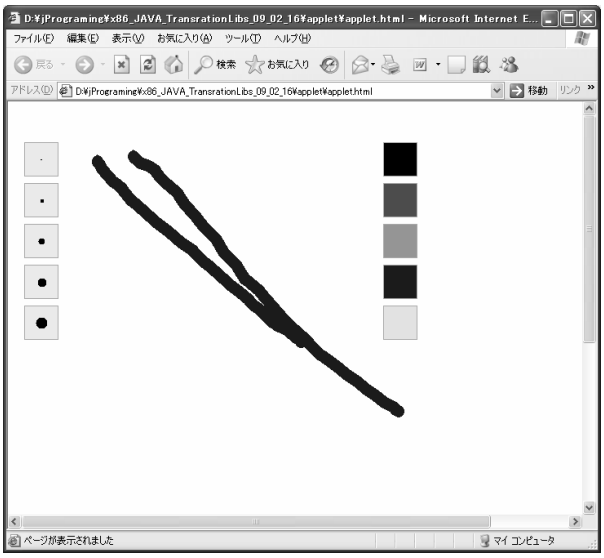

図 8.2.5 アプレット化した GUI プログラムへのマウス入力

8.3 その他の評価

8.3.1 命令レベル統計情報

本エミュレータには、どのような命令を何回実行したかを記録する機能が備わっている。表 8.3.1 に、先に使用したフィボナッチのプログラムの統計情報の一部を示す。命令の使用頻度が高い上位 5 件までを抜粋した。

表 8.3.1　フィボナッチの命令レベル統計情報

命令フォーマット	実行回数	使用頻度(%)	実行時間の割合
MOV reg32 mem32	8053	15.801	22.916
ADD reg32 imm8	3839	10.347	6.477
JZ imm32	4572	9.356	2.246
JNZ imm32	3829	9.297	3.493
MOV mem32 reg32	7413	8.163	10.362

8.3.2 命令デコーダの処理性能

本エミュレータでは二種類の命令デコーダを併用している。一方は低速だが x86 の全ての命令セットをデコード可能な汎用ディスアセンブラを改変したものであり、一方は x86 の命令セットのうち命令語長が 4 バイトまでで、かつ浮動小数演算命令以外の命令のデコードのみが可能な比較的高速な命令デコーダである。二つの命令デコーダの平均デコード時間には約 2 倍の差がある。表 8.3.2 に命令語長が 3 バイトまでの命令デコードにおける、二つの命令デコーダの性能を示す。命令語長が 1 バイトまでの命令デコードでは高速デコーダが汎用デコーダに比べ約 3 倍低速であるが、これは、たかだか 256 回の命令デコードでは JavaVM の JIT 機能が十分に作用しないためだと考えられる。参考までに JIT 機能を無効にした結果を示したが、命令語長が 2 バイトより大きい場合、JIT 機能が有効であれば平均デコード時間が大幅に短縮されているのに対し、JIT 機能を無効にした場合平均デコード時間は段階的に大きくなっていく。また、汎用デコーダの平均デコード時間は JIT の有無に関わらずほぼ一定であるが、これは汎用デコーダの実際の処理部分が JNI によって呼び出されるネイティブコードで構成されているためである。

表 8.3.2 命令デコーダの平均デコード時間(ナノ秒)

デコーダタイプ	1byte	2byte	3byte
高速デコーダ	37704	5798	4240
汎用デコーダ	10586	8501	8355
高速デコーダ（no-JIT）	46386	61888	67749
汎用デコーダ（no-JIT）	11113	9467	9304

8.3.3 命令ディスパッチの性能

本エミュレータの命令ディスパッチは、リフレクションや javassist、Java Compiler API を用いた動的コンパイルによって実現され、ディスパッチ部分がハードコードされていない。これは将来、threaded code や super instruction などの最適化手法を適用することを前提としたものである。簡単化のため、これらの最適化手法はまだ組み込んでいない。

図 8.3.1 では、javassit を用いた動的コンパイルによるメソッド呼び出しのコストをリフレクションと比較した。リフレクションのデータはメソッドインスタンスのキャッシュ有無により 2 種

類の状態を計測した。引数の種類が異なる静的・動的メソッドを 1 回から 100 回まで呼び出して時間を計った。結果、javassist を用いた動的コンパイルは、メソッドの初回呼び出しの際に、リフレクションに対し 20 倍から 300 倍程度低速になる一方で、2 回目以降の呼び出しではリフレクションに対し 10%程度呼び出し性能が高速になることが分かっている。このため、本エミュレータでは使用頻度の高い MOV 命令などのディスパッチに動的コンパイルを用い、使用頻度の比較的低い命令にはリフレクションを適用することとした。

x86 の各命令と、Java 言語で実装した本エミュレータが提供する命令セットとの関連付けは CSV ファイルに記載してあり、エミュレータの起動時に CSV ファイルを読み取って設定される。これもエミュレータの実行時プロファイリングをもとに、CSV ファイルの内容を書き換えて最適化を図ることを目指した仕様であるが、まだ実現できていない。

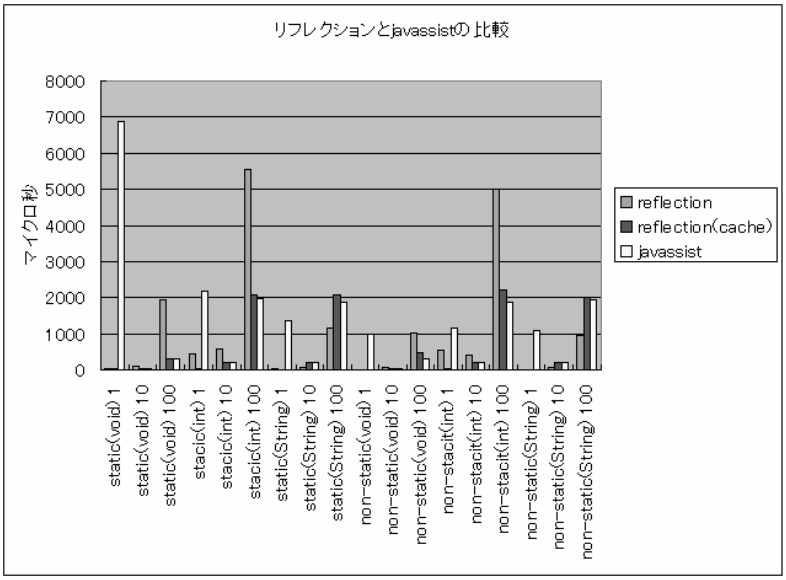

図 8.3.1 リフレクションと javassist による動的コンパイルの比較

表 8.3.3　静的メソッド呼び出しにおける
リフレクションと javassist による動的コンパイルの比較(マイクロ秒)

	static(void) 1	static(void) 10	static(void) 100	stacic(int) 1	stacic(int) 10	stacic(int) 100	static(String) 1	static(String) 10	static(String) 100
Reflector.ReflectorReflection	48	101	1955	423	562	5531	23	84	1137
Reflector.ReflectorReflectionWithCache	19	49	324	48	202	2021	14	216	2043
Reflector.ReflectorJavassist	6724	44	296	2236	200	1833	1210	210	1879
Reflector.ReflectorJavaCompilerAPI	410476	51	203	29818	204	1823	54927	186	1628

表 8.3.4　動的メソッド呼び出しにおける
リフレクションと javassist よる動的コンパイルの比較(マイクロ秒)

	non-static(void) 1	non-static(void) 10	non-static(void) 100	non-stacit(int) 1	non-stacit(int) 10	non-stacit(int) 100	non-static(String) 1	non-static(String) 10	non-static(String) 100
Reflector.ReflectorReflection	11	78	1008	556	407	4949	17	80	955
Reflector.ReflectorReflectionWithCache	15	49	474	48	211	2105	15	207	1987
Reflector.ReflectorJavassist	951	35	300	1122	215	1885	1100	201	1926
Reflector.ReflectorJavaCompilerAPI	20538	23	149	17664	190	1660	19832	197	1680

8.3.4 JNI エミュレーションによる性能低下

NestedVM 上で JNI コードをエミュレーションした場合の基本性能を計測した。測定は Java 言語から C 言語の関数を呼び出した場合と、C 言語から JavaVM 上の資源を読み書きした場合を対象とした。

表 8.3.5 に JIT 機能が有効な JavaVM 上で、値を返すだけの C 言語の関数を Java 言語から呼び出した場合の性能を示す。また表 8.3.6 では同じプログラムを、JIT 機能を無効にして測定した結果を示す。

表 8.3.5　Java から C 言語のメソッド呼び出し(JIT 有効)

引数タイプ	void	Int	int*8	String	String*8
Linux (nano sec)	513683	3551648	336310	201675	320947
NestedVM (nano sec)	58059298	65460812	89676278	94035286	343185035
性能比: NestedVM/Linux	113.026	18.431	266.648	466.271	1069.289

表 8.3.6 Java から C 言語のメソッド呼び出し(JIT 無効)

引数タイプ	void	int	int*8	String	String*8
Linux no-JIT (nano sec)	544143	593864	721240	541629	894428
NestedVM no-JIT (nano sec)	625092285	691765406	990720412	998012427	3589148365
性能比: NestedVM /Linux	1148.765	1164.855	1373.635	1842.613	4012.786

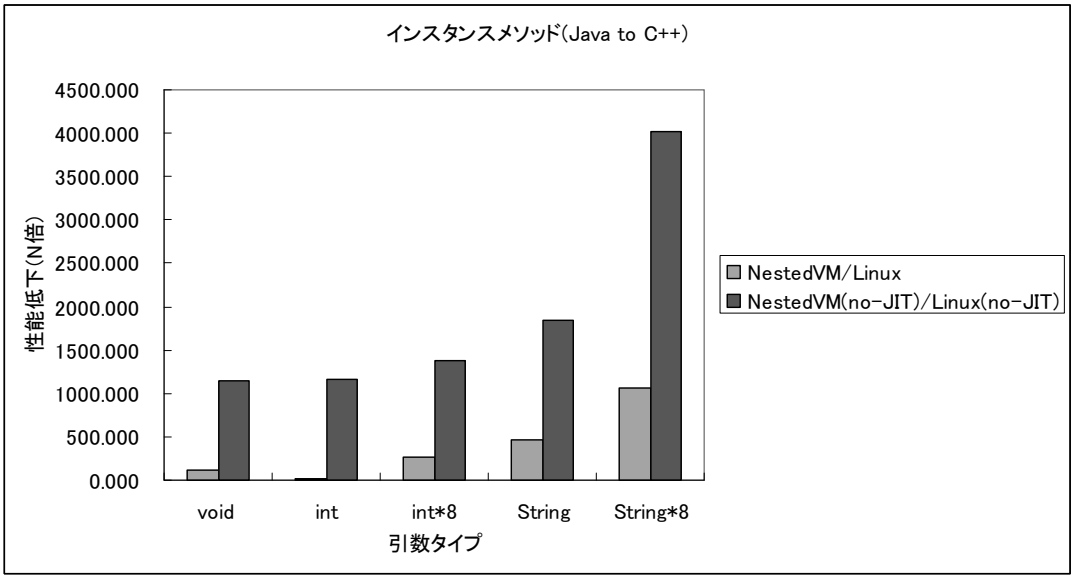

図 8.3.2 Java 言語から C 言語の関数呼び出し性能

表 8.3.7 には C 言語から Java 言語の静的フィールドを読み書きした際の性能を示す。同様に、表 8.3.8 には動的変数を読み書きした際の性能を示す。図 8.3.3 ではこれらの結果をグラフ化した。

表 8.3.7 C 言語から JavaVM 上の静的フィールドアクセス

タイプ	static int(read)	static String(read)	static int(write)	static String(write)
Linux	2771766	4045499	2729588	3621759
NestedVM	211074007	218193394	157811713	184485598
NestedVM/Linux	76.151	53.935	57.815	50.938

表 8.3.8　C 言語から JavaVM 上の動的フィールドアクセス

タイプ	int(read)	String(read)	int(write)	String(write)
Linux	2577913	4067566	2750258	4001086
NestedVM	217168517	292648482	219468554	246375962
NestedVM/Linux	84.242	71.947	79.799	61.577

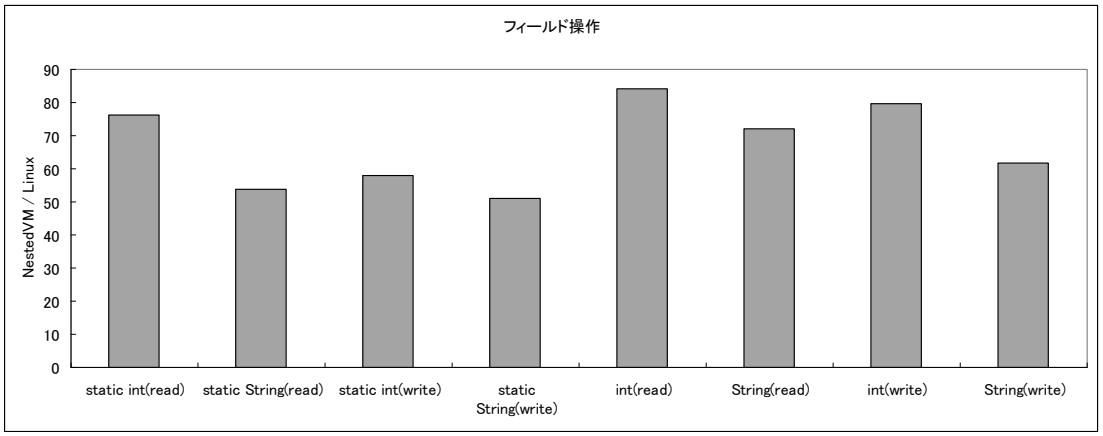

図 8.3.3　C 言語から JavaVM 上のフィールドアクセス性能

Java 言語から C 言語のメソッドを呼び出した際の実機に対する JNI エミュレーションの性能低下は、JavaVM の JIT が有効な場合、概ね 20 倍から 1000 倍程度で実現している。しかし、20 倍の性能低下を示した、int 型引数 1 個のメソッド呼び出しでは、実機の性能が著しく低く、同じく実機で実行した int 型引数 8 個のメソッド呼び出しと比べても約 10 倍の時間が掛かっているため評価の対象として適切ではない。したがって、Java 言語から C 言語のメソッド呼び出しにおける現実的な性能低下は 100 倍～1000 倍であり、性能低下の度合いは引数の種類(プリミティブ/参照型)と個数に比例して大きくなる。

int 型引数 1 個の場合、実機の性能が極端に低下した原因として JIT コンパイラのオーバーヘッドが挙げられる。JavaVM の-Xint オプションを用いて JIT 機能を無効化したうえで、実行時間を計測したところ、int 型引数 1 個の静的メソッドの呼び出し時間は 0.5 ミリ秒程度であり、JIT コンパイラが有効であった際の 3 ミリ秒を大きく下回っている。逆に JNI エミュレーションの性能は大幅に低下しており、JIT が有効であった際の 78 ミリ秒に対し、732 ミリ秒を要している。このため、JIT コンパイラが無効な状態での、Java 言語から C 言語のメソッド呼び出しにおける、JNI エミュレーションの実機に対する性能低下は、1100 倍から 4000 倍程度となっている。ネイティブコードを JavaVM 上でエミュレーションする今回の方式では、JIT コンパイラが有効であったほうが性能的に有利であるが、一方で実機の JNI の性能が極端に低く反映されることもあり、単純に比較することは難しい。

ネイティブコードから JavaVM 上のクラスメンバ変数に対し読み書きを行う場合は、60 倍

から 85 倍程度の性能低下で実現している。

本エミュレータが POSIX ラッパーを使用しシステムコールを呼び出す頻度は、通常の命令エミュレーションに比べ大幅に低いため、ここで述べた性能低下は許容できると考えている。

8.4 考察

基本性能の評価より、本エミュレータの演算性能は x86/Linux の実機の環境に対し 2 万倍から 40 万倍程度低速である。このうち、フィボナッチや for ループのように繰り返し処理のあるプログラムにおいては 2 万倍から 13 万倍程度低速であり、繰り返しのない処理では 14 万倍から 42 万倍程度低速である。繰り返し処理のあるプログラムが比較的高速なのは、本エミュレータが命令デコード結果をキャッシュし、またメモリに対する読み書き処理においてメモリブロックの検索結果をキャッシュすることが影響している

本エミュレータの I/O 性能は演算性能に比べ大幅に改善し、実機に対し 1600 倍から 2.8 万倍程度で実現している。これはシステムコール処理をエミュレータの外部で動作する OS に対し依頼するユーザーモードエミュレータの性質を表している。したがって、本エミュレータは、I/O 処理の性能が重要となる GIMP のような GUI アプリケーションを利用するうえで有用であると推測される。さらに、上記の I/O 性能は/dev/null や/dev/zero といったデバイスに対する読み書きに基づいており、ディスク装置上に実在するファイルに対する読み書きは、実記に対し 350 倍から 1300 倍程度の性能低下で実現している。ディスク上のファイルに対する読み書き性能は計測環境に大きく依存するものの、実際的な環境を想定した場合の本エミュレータの有用性をある程度示唆している。

dd コマンドの評価では共有ライブラリを利用するプログラムバイナリと、静的リンクされたプログラムバイナリを本エミュレータ上で実行した。結果、本エミュレータ上で共有ライブラリを利用した場合、静的リンクしたプログラムを実行する場合に比べ、2 倍程度性能が低下することが分かった。これは、本エミュレータが mmap システムコールをサポートする過程で、メモリ空間の一部をネイティブなメモリ空間に対応づけているため、一部のメモリアドレスに対する読み書きが、JNI のオーバーヘッドに影響されることを示している。

第 9 章 まとめ

本研究では、x86/Linux 用ユーザーモードアプリケーションの異種 OS・アーキテクチャ間における可搬的な利用を目的とした、JavaVM 上で動作するユーザーモードエミュレータを実装し評価を行った。

本エミュレータの実現に必要であった x86 命令セットのうち、浮動小数演算命令以外の全ての非特権命令と、FPU 命令の一部を実装した。またシステムコールは、POSIX 互換システムコール 66 個と、Linux 独自のシステムコール 9 個を利用をサポートしている。

本エミュレータの特徴として、マルチスレッドプログラムのエミュレーション時にエミュレータ自身がマルチスレッド動作を行うこと、mmap()システムコールのエミュレーション時に、エミュレーションするメモリ空間とエミュレータの下部で動作する実機のメモリ空間の一部が、単一のメモリイメージとしてアプリケーションから認識されること、ローダーモジュールの実装において、x86/Linux 用の共有ライブラリローダー自体をエミュレーションすることによりローダーの実装過程を大幅に簡略化したこと、などが挙げられる。

機能面では、GUI、共有ライブラリ、マルチスレッド等を使用する x86/Linux 用プログラムを、異種 OS 間において可搬的に利用可能な機能を実際に有している。Linux 用プログラムが使用するこれらの機能を、他の OS 上においてもサポートするユーザーモードエミュレータはまだ存在しない。また GUI プログラムを Web ブラウザ上でアプレットとして実行することも可能であり、既存の X ウィンドウライブラリを使用する GUI プログラムに変更を加えることなく Web ブラウザ上に配信することを可能とした。一方、機能面で本エミュレータが実現できなかったものとして、OpenOffice や GIMP といった大規模かつ、システムコールの複雑な組み合わせを伴う実用的なアプリケーションのサポートが挙げられる。

性能面では、x86/Linux の実機の環境に対し、演算性能で 2 万倍から 40 万倍程度、I/O 性能で 1600 倍から 28000 倍程度の性能低下が見られた。これは類似研究である NestedVM が MIPS 用プログラムを実機に対し演算性能で 8 倍程度、I/O 性能で 2 倍の性能低下で実現しており、さらに GUI プログラムのレスポンス性能が極端に低下したことなどから見ても、実用上の問題を有している。

今後の課題として、性能面では threaded code や super instruction といった命令ディスパッチの最適化手法をユーザーモードエミュレータの実装に取り入れることや、動的プロファイリングによる統計情報を元に Java バイトコードの動的生成を行うことが考えられる。機能面では、未実装の FPU 命令を実装することによる高機能なウィジェットツールキットのサポートや、利用可能システムコールの拡充によるエミュレーション可能なアプリケーションの増加が挙げられる。

参考文献

[1] “Bochs project”, http://bochs.sourceforge.net/

[2] Rhys Newman et al, “JPC project”, http://www-jpc.physics.ox.ac.uk/index.html

[3] “Xen 3.0 user’s manual”, Xen project, http://www.xen.org/

[4] “QEMU Emulator User Documentation”, QEMU Project, http://fablice.bellard.free.fr/qemu/qemu-doc.html

[5] ALLIET, B., AND MEGACZ, A. Complete translation of unsafe native code to safe bytecode. In Proceedings of the 2004 Workshop on Interpreters, Virtual Machines and Emulators, pp. 32–41,2004.

[6] 西山 清香, Emulin, http://www.netfort.gr.jp/~kiyoka/emulin/

[7] Linux/Alpha project, “EM86: How To Run Linux/x86 Apps on Linux/Alpha”, http://www.alphalinux.org/faq/FAQ-16.html

[8] A. Yermolovich et al, ” Portable Execution of Legacy Binaries on the Java Virtual Machine”, ACM International Conference Proceeding Series; Volume 347, pp.63-72, 2008

[9] ECMA International. “ECMAScript language specification. Stardard ECMA-262,5th Edition”, http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf, December 2009.

[10]Microsoft Corp, "Silverlight", http://silverlight.net.

[11] Microsoft Corp, "ActiveX Controls", http://msdn.microsoft.com/en-us/library/aa268985(VS.60).aspx

[12]Adobe Systems Incorporated, "Adobe AIR SDK", http://get.adobe.com/jp/air/

[13] Bennet Yee, et al, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," 2009 30th IEEE Symposium on Security and Privacy, pp.79-93, May. 2009.

[14] M. Anton Ertl , David Gregg .”Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters”, ACM SIGPLAN Notices, Volume 38 , pp.278-288(2003)

[15] D.Ung, C.Cifuentes. “Machine-adaptable dynamic binary translation”, *proc. of the ACM SIGPLAN workshop on Dynamic and adaptive compliation and optimization*, pp.41-51,2000.

[16] R.Gordon 著,林 秀幸訳, “Java Native Interface プログラミング”, *ピアソン・エデュケーション*, 1998

[17] J.Meyer 著,鷲見 豊訳, ”Java Virtual Machine”, *オライリージャパン*, 1997

[18] R.Sites et al., ”Binary Translation”, *Communications of the ACM volume 36*,

pp69-81, 1993

[19] Intel, “IA32 Intel アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル”, Intel Corporation, 2004

[20] 千葉滋, 他, "Javaバイトコード変換による構造リフレクションの実現", 情報処理学会論文誌 42(11), pp.2752-2760, Nov. 2001.

[21] Simon Kagstrom, et al, "Cibyl: an environment for language diversity on mobile devices", ACM/Usenix International Conference On Virtual Execution Environments, Proceedings of the 3rd international conference on Virtual execution environments, pp75-82, June.2007.

謝辞

本研究を進めるにあたり、筆者が所属する並木研究室の方々には日ごろから研究にあたって多くの助言やご指導を賜りましたことを厚く御礼申し上げます。

修士課程 2 年の磯部泰徳氏、大田篤志氏、砂田徹也氏、竹川和孝氏、林和宏氏には、2 年間にわたる研究生活で大変お世話になりました。日々の研究や、対外発表等での助言のみならず、学生生活全般において力を貸していただきました。深く感謝いたします。

修士課程 1 年の木村一樹氏、高畠和幸氏、盛合智紀氏には、ゼミを通して興味深い発表を聞かせていただき、筆者の研究生活を有意義なものにしていただきました。深く感謝いたします。

学部 4 年の池内嵩俊氏、仁科圭介氏、村田勇次郎氏、茂木勇氏、吉原陽香氏には、ゼミを通して興味深い発表を聞かせていただき、筆者の研究においても大いに参考になりました。深く感謝いたします。

特別研究生の佐藤未来子氏には、ゼミで貴重な発表を聞かせていただいたほか、人生の先輩としての貴重なご意見を聞かせていただきました。深く感謝いたします。

並木美太郎教授には、修士課程の 2 年間お世話になりました。他学からの進学者である筆者の指導に際しては、常々大変な思いをされたことと存じます。筆を降ろすにあたり、先生の温かいご指導に敬意を表すとともに心より御礼申し上げます。

付録

川口直也,並木美太郎,”JavaVM による x86 ユーザーモードエミュレーション機構の実装と評価”, 情報処理学会「システムソフトウェアとオペレーティング・システム」研究会第 109 回研究報告, Vol.2008-OS-109, pp.77-84, 2008.