2005－OS －98（5）
2005／2／22

Java を用いた OS 構成法

山本 茂樹†　早川 栄一‡

†拓殖大学大学院工学研究科　‡拓殖大学工学部

†yuulu@os.cs.takushoku-u.ac.jp, ‡hayakawa@cs.takushoku-u.ac.jp

本稿は Java を用いた OS 構成を提案する。JavaVM を Java 言語で記述し、直接ネイティブコードに変換することで Java 言語を使用した OS 記述を可能とする。動作環境を JavaVM にすることで、従来のハードウェアによる動作の状態の違いによる保護を使用せずソフトウェアによる保護を実現する。システムの資源管理ポリシを JavaVM 上で動作させることで、JavaVM のサンドボックス環境での動的な追加、削除を可能とする。また、アプリケーションと OS との記述言語を統一することにより、言語のインタフェースギャップをなくすことができる。システムの特徴として仮想マシンと高水準言語によるシステム記述、高水準言語からのハードウェアアクセス、メモリ･レジスタのソフトウェアによる保護法、言語の機能を用いたシステムの拡張方式が挙げられる。

An OS Architecture with Java Virtual Machine

SHIGEKI Yamamoto†, EIICHI Hayakawa‡

†Graduate School of Engineering, Takushoku University
‡Faculty of Engineering, Takushoku University

†yuulu@os.cs.takushoku-u.ac.jp, ‡hayakawa@cs.takushoku-u.ac.jp

This paper proposes an OS architecture which uses Java Virtual Machine. We write an OS using Java language that translates Java byte code to native code. This OS uses software protection model that use type safe language and virtual machine. It is possible to add OS module on Java virtual machine's sand box. To close interface gap of OS and applications, layers of middleware reduce. This system's features are Java virtual machine in OS, software protection model and secure system extension.

１．はじめに

近年計算機の高性能化、小型化が進み、計算機の利用形態が多様化している。組込みシステムにおいてもネットワークに接続することが行なわれてきている。従来の組込み機器は静的に構成が決まっていたが、組込み機器にも動的にハードウェア、ソフトウェアを追加、削除するようなことが起こり始めている。

組込みシステムは PC とは違い、プログラムの入換えがハードウェアの構成から困難である。そのため、PC よりもコンピュータウィルスやプログラムのバグに関しては注意をはらう必要がある。しかし、動的にプログラムやハードウェアを追加するような環境ではデバイスドライバやアプリケーションを動的に追加しなければならなくなってしまう。

現在の携帯電話では、命令セットを提供する仮想マシンを使用して、安全なサンドボックスの環境を構築し、その上で動的に追加されたアプリケーションを動作させる構成となっている。しかし、この方法では仮想マシン上で動作していないプログラムの安全性が確保することができない。また、デバイスドライバなどは命令セットを提供する仮想マシンでは記述することができないため、安全に動的に任意のデバイスを繋ぎかえることが行なえない。

そのため、ハードウェアへのアクセスを含めた安全な環境をシステムが提供することで、動的なデバイスの追加、削除が行なわれる環境下において、システムの安全性を確保しながらシステムの拡張が行なえる環境が必要である。

２．要求分析

組込みシステムではアプリケーションがターゲットとする処理に対して特化される。そのため、OS もそのアプリケーションに対して特化した環境を提供することで、高い実行効率を提供することができるようにする必要がある。このようなアプリケーションに特化した環境を提供する OS として Exokernel[1]などがある。

しかし、組込みシステムに動的なハードウェア、ソフトウェアの追加、削除が行なわれるようになると Exokernel のようなネイティブコードを中心とした環境では問題が起こる。動的なソフトウェアの追加機能により、追加したソフトウェアのバグで OS に影響を与えてしまっては困る。また、追加したアプリケーションのバグによりセキュリティの問題が起きる可能性がある。

組込みシステムは機器の特性上プログラムを入れ直すことが難しい。そのため、ソフトウェアのバグが起きると PC よりも直すのに手間がかかる。動的な環境を与えるとさらに信頼性のないプログラムを動作させることになるので、バグがあったプログラムを簡単に直せる環境が必要である。

このようなことを考慮し、本研究では OS に対して次のような要求が満たされることが望まれていると考える。

1. 動的な追加に対する安全な環境をアプリケーションだけではなく、システムサービスに対しても提供する

   アプリケーションに対してその用途に合った環境を提供することにより、高い実行性能を与えることが求められるが、それはアプリケーションだけではなく、システムサービスに対しても高い実行効率と安全性を与えることが望まれる。

2. OS への安全な実行環境の提供

   OS による安全性の高い環境を提供することで、OS によるシステムの乗っ取りなどを防ぐことができる。

3. アプリケーションへのハードウェアに依存しないリソース制限

   汎用的な処理と専用の処理を同時に行う機器に対しては、汎用処理を行うソフトウェアにはハードウェア依存のリソースを渡す必要がなく、また、組込みシステムではハードウェアアーキテクチャが多種多様であるため、そのリソース制限の方式はデバイスに依存しないことが望まれる

上記のことを解決することができる OS が望まれる。

３．設計方針

先に述べた要求より、次の設計方針を立てた。

1. ソフトウェアを制限の強い命令セット上で動

作させる

ソフトウェアの実行を自由度の高い実 CPU 上で行うのではなく、制限の強い命令セットを提供する仮想マシンを使用することにより、ソフトウェアの実行可能な範囲を絞ることで安全性の高い環境を作る。

2. リソースの制限を行うことにより、ソフトウェアに対して強い制限を与えることができる

アプリケーションの用途により使用することができるデバイスなどを制限し、安全性とデバイスへの低い抽象化による高い実行性能を実現する。

4. 設計

4.1 全体構成

本システムの全体構成を図 1 に示す。本システムは JavaVM を用いて、従来の OS の役割を JavaVM 内で行う。アプリケーションは OS 上で動作するバイトコードのアプリケーションであり、JavaVM が持つコンパイラによりネイティブコードに変換されることにより実行する。

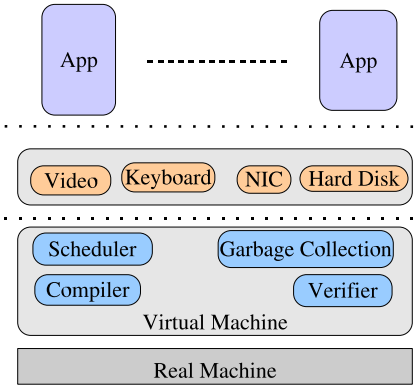

図 1　全体構成

4.2 Java 言語による JavaVM

従来のシステムはアプリケーションに対して、仮想マシンを使用した安全性の高い環境を構築していた。しかし、本システムではアプリケーションだけでなく、OS に対しても安全性の高い実行環境を与えたい。そのため仮想マシンを仮想マシンが解釈する命令セットを使用して実現することにより、安全性の高い仮想マシン環境をシステムに適用する。

仮想マシンを仮想マシンの命令セットで実現するには次の方法がある。

1. ネイティブコードで記述された仮想マシン上にバイトコードの仮想マシンを動作させる

再帰的な仮想マシンの適用方法であり、仮想マシンを多重化する方法である。この方法によりバイトコードで記述された仮想マシンに、ネイティブコードで動作する仮想マシンが構築するセキュアな機能が適用される。このため、安全性の高い仮想マシンを構築することができる。

2. バイトコードで記述された仮想マシンを、実 CPU が理解するネイティブコードに変換するコンパイラを使用することにより実 CPU 上で直接動作させる

バイトコードで記述された仮想マシンを、仮想マシンが提供する機能を保ったままにネイティブコードに変換する。これによりバイトコードで記述され、ネイティブコードで動作しながら、仮想マシン上で動作しているのと同様なセキュアな仮想マシンとなる。

再帰的な仮想マシンの適用法では実行環境が 2 段階となり実行速度のコストが高い。

２つ目の方式ではネイティブコードに変換されるので、実行速度のコストは低いが、仮想マシンを JavaVM が提供する命令セットを使用するので、実 CPU が行えるメモリやレジスタへのアクセスが行うことができない。

本システムでは JavaVM の実装法に 2 つ目の方法を使用する。しかし、本システムでは仮想マシンを OS として使用するため、実 CPU へのアクセス方法を提供する必要があり、これを解決しなければならない。

4.3 Java 命令セットから任意の実 CPU 命令セットの呼出し

仮想マシンをコンパイラに使用した仮想的に再帰的な仮想マシンの構成にするため、OS として使用するためには JavaVM の命令セットから実 CPU の命令セットを呼び出す機構が必要である。JavaVM の命令セットはメモリアドレスを前提とした命令セットではないので、実 CPU の命令セットにあるメモリやアーキテクチャ依存のレジスタなど

のアクセス命令が行えない。

この実ハードウェア依存の命令を仮想マシンから呼び出すことができないと、メモリ管理のコードなど OS が提供すべき機能が実装することができないため、OS に仮想マシンを適用することができない。そのため、この解決策として、本システムではコンパイラに機能を追加する。

コンパイラへの機能追加方法を次に列挙する。

1. 宣言をし、定義のないメソッドを用意する
2. コンパイラにその定義のないメソッドを登録する
3. メソッドに対応した実 CPU の命令列を用意する
4. コンパイル時に定義のないメソッドを探す
5. 定義のないメソッドが来たら、用意しておいた命令列を定義として出力する
6. 定義のある通常のメソッドが来たら、通常のバイトコードをネイティブコードへの変換をする

この方法により仮想マシンの命令セットを使用していても実 CPU の命令セットを使用することができる。この方法ではコンパイラが任意のネイティブコードを出力するので、この Java の命令セットに依存しない命令列はコンパイル時に生成することになる。よって、実行時にこの方法を使用し、任意の命令を使用して新しい命令列を生成することはできないので、セキュリティの問題にはならない。以下この命令列をマジックメソッドと呼ぶ。

しかし、この方法ではマジックメソッドの公開が問題になる。メモリ保護をしない構成では OS のメモリをすべて書き換えることが可能だからである。そのため、本システムではタスクにはマジックメソッドは公開せず、OS だけがこのメソッドにアクセス可能とする。ハードウェアへの直接のアクセスは OS が行い、それをオブジェクトとしてタスクに公開することで、タスクからのアクセスを可能とする。

コンパイラによる命令生成を図 2 に示す。

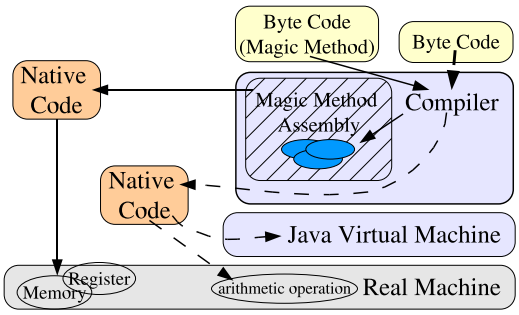

図 2　コンパイラによる非 Java 命令の生成

4.4 アドレス空間

本システムではすべてのアプリケーションは仮想マシン上で動作する。そのため、アプリケーションは JavaVM が提供する制限された命令セットにより、実ハードウェアへのアクセスが制限される。マジックメソッドを使用しない限り、アプリケーションはメモリアドレスを意識した実ハードウェアではなく、名前でアクセス先を決める JavaVM の環境で動作することになる。

これにより、従来の OS が提供していた多重アドレス空間を使用する必要性がなくなる。これはメモリの保護をしなくてもアプリケーションは他のアプリケーションに対してアクセスすることができないからである。

そのため、本システムでは多重アドレス空間よりアドレス切替えの変換コストがかからないシングルアドレス空間を使用する。

4.5 タスク

JavaVM はシングルタスクでマルチスレッドの環境である。OS に JavaVM を適用するにあたり、この JavaVM のタスクモデルを拡張する必要がある。

一つのコンテキストでアプリケーションを動作させてしまうと、ヒープを全スレッドで共有してしまう。この状態では、Garbage Collection（以下 GC）が起きた場合に、タスクを止めるか、タスクと GC を平行して動作させなければならない。タスクを止める方法では一つのタスク処理に他のタスク全体が影響されてしまう。タスクと GC を同時に動作させオブジェクトを回収する場合は、複雑さと回収率の問題がでてくる。

よって、本システムのアプリケーションごとに高い実行効率の環境を提供するという目的から、タスクごとにスタック、ヒープ領域を分け、GC 処理はそのタスクのコンテキストで行う。これにより VM のコンテキストで GC を行う場合に起こる、複数のタスクからの GC 要求に対しても問題がなくなる。そしてタスク間に親子関係を持たせることにより、親がこのヒープ量を決めることができるようにする。

タスクのコンテキストで行う場合には GC を GC 専用のコンテキストで行わないので、JavaVM をバイトコードで記述することにより、GC 中に GC を行うためのオブジェクトを作成するということが起

きてしまう。このため、本システムでは GC を行うために予約領域をあらかじめ取っておくことによってこれを解決する。

4.6 システムコール

従来の実 CPU での動作では任意のアドレスへの関数コールが行えた。本システムは仮想マシン上ですべてのアプリケーションが動作するため、JavaVM 上では名前を使用したメソッドコールなので、リンクが可能であれば任意のメソッドの先頭を呼び出すことができる。これら不正な呼出しから OS を保護するために、OS とアプリケーションは直接呼び出せないようにしなければならない。

従来はソフトウェア割込みを使用した呼出し方法であったが、これは実 CPU のメモリアドレスを意識した命令セットでは効率がよかった。しかし、JavaVM のような名前で呼び出す構成では効率がいいとはいえない。本システムでは、仮想マシンの構成を考慮し、名前を用いたベリファイアによる方法を提案する。

これは次の方法によって実現する。

1. OS がシステムコールをベリファイアに登録する
2. クラスを読み込む
3. クラスファイルから呼び出すメソッド一覧を調べる
   - メソッド一覧に OS が登録したメソッドがあれば、そのメソッドにリンクする
   - メソッド一覧に OS が登録していないメソッドがあれば、例外を起こし動作を止める
   - メソッド一覧に OS が使用しているメソッドがなければなにもしない
4. クラスファイルをコンパイルする

この方法は JavaVM がメソッドを名前でリンクする構成だから可能となり、この方法によりハードウェアの機能を使用せずに JavaVM はシステムコールとして公開するメソッドを選択することができるようになる。

この方法では実行前の検査時にシステムコールを呼出すかどうかを判断しているので、実行時にはシステムコールはメソッド呼出しと等価となり、実行時のコストが少なくなる。

また、アプリケーションと仮想マシンが同一の環境であるので、システムコールのインタフェースが同一となる。従来の Java 言語を C 言語に変換する必要性がないため、スタブを経由する必要がなくなりインタフェースのギャップがなくなる。

4.7 拡張方式

システムの拡張はアプリケーションと同様の層で行う。アプリケーションに対して高い実行効率を実現するため、次の機能をアプリケーション側で制御することができるようにする。

1. デバイスへのアクセス
2. GC のアルゴリズム
3. 割り込み
4. タスク間通信

これらを使用してアプリケーションがアプリケーションの実行にあった環境を構築する。

4.8 デバイスへのアクセス

本システムでは実行性能のためにシステムによるデバイスの抽象化をしない。デバイスはアプリケーションが直接デバイスにアクセスする。

デバイスを操作するためには、デバイスがマップされているアドレスにアクセスする必要がある。このために仮想マシンがデバイスへのアクセス法を公開する。デバイスがマップされているアドレスをオブジェクトとしてタスクに公開し、そのオブジェクトを使用してデバイスへのアクセスを行う。このオブジェクトのメソッドはシステムコールであり、このオブジェクト内でメモリへのアクセスが行われることになる。しかし、デバイスがマップされていないアドレスをアクセスするように指定された場合、メモリの書換えが可能になってしまう。デバイスがマップされているアドレスと他のアドレス空間を切り分けないと、デバイスドライバを記述するときに、セキュアな環境を保てなくなる。

これを解決するために本システムでは SFI[2]のように、オブジェクト内でアクセス先のメモリアドレスを判断し、マップされたアドレスだけをアクセス可能にさせる。ソフトウェアによるアドレスの比較を行い、アクセス先のアドレスにデバイスがマップされているかを検証し、他のアドレスへのアクセスを防ぐ。この方法によりタスクからの安全なデバイスアクセスが可能となる。この方法ではアドレスの比較を行うので、比較のコストが掛かってしまう。しかし、マップされたアドレスは固定であるため、効率的なアドレス比較を用いれば、比較回数を抑えることが可能である。そのため、抽象化によるデバイスの判定や特権状態への遷移を行うのと比較して

もコストの増加はないと考える。

図 3 にデバイスへのアクセス法を示す。

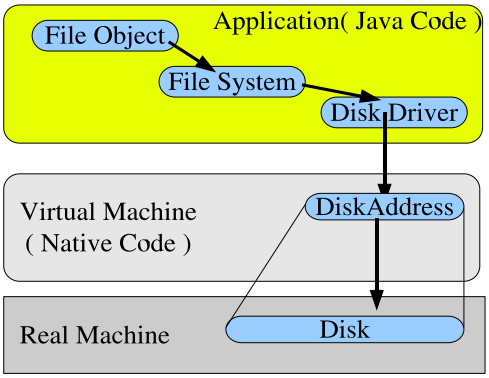

図 3 デバイスのアクセス

4.9 GC アルゴリズムの選択

タスクごとに処理する問題が違うため、その処理にあった GC の処理を行うことが望まれる。メモリを解放することを優先するか、実行速度を優先するかをタスクが選択することができるべきである。

そのために、仮想マシンはタスクを生成するときに仮想マシンが提供する GC のアルゴリズムを選ぶことができる構成とする。この構成により、タスクごとに GC 処理を最適化することができる。

4.10 割込み

仮想マシンは割込みを抽象化せず、アプリケーションに対して直接割込みコントローラに設定をする方式をとる。これは高い割込み応答性が必要なアプリケーションを配慮したものである。

しかし、Java 言語のようにオブジェクト指向の構成では直接メソッドを呼び出すように割込みコントローラに設定するわけにはいかない。これはメソッドを呼び出すにはオブジェクトの参照が必要だからである。そのため、メソッドの登録はシステムコールを介して登録することにより、実 CPU の関数呼出しからオブジェクトのメソッド呼出しの変換をするスタブを介す。

割込みが起きたときにメソッドが存在していない場合問題になる。JavaVM による保護を使用しているので、もしも間違った関数コールをした場合、そのままシステムを壊してしまう。そのため、スタブにそのオブジェクトが存在しているかを確認してから呼び出すことにし、保護を実現する。

これを図 4 に示す。

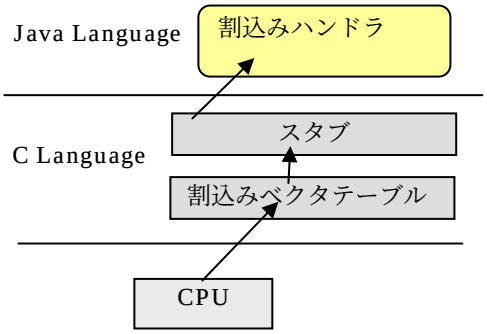

図 4 割込みの呼出し

4.11 タスク間通信

JavaVM 上でシングルアドレス空間をとっていたとしても、JavaVM 上では他のタスクをアクセスすることはできない。そのため、多重アドレス空間同様にタスク間通信の問題がでる。

タスク間通信を実現するために、本システムでは仮想マシンにシステムコールでオブジェクトを登録し、通信を行いたいタスクはシステムコールでその参照を得る方法をとる。これによりオブジェクトの共有が行える。

この方式は RPC と同様の方式であり、一方通行の呼出しである。RPC との相違は関数呼出しであるのと、ネットワークを越えた呼出しはできないことである。

参照している相手のオブジェクトがなくなった場合、不正な呼出しになってしまう。これによりシステムを壊してしまうことが可能なので、本システムでは仮想マシンを介した呼出しにすることで、解決する。仮想マシンが相手のオブジェクトが存在しているかを判断してから呼び出すことにより、不正な呼出しを防ぐ。

これを図 5 に示す。

4.12 アクセスコントロール

アプリケーションごとに使用するリソースは違う。デバイスを使用しないアプリケーションに対して、デバイスを公開するのはセキュアな環境を実現するためには好ましくない。そのため、本システムでは名前を用いたアクセスコントロールを使用する。

仮想マシンの命令セットは名前を使用した呼出しである。また、呼び出す名前はクラスファイルにある。そのため、事前にどのメソッドを呼び出すのかがわかる。この特徴を使用して、仮想マシンはア

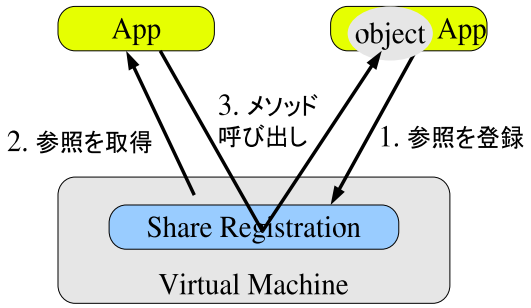

図 5 タスク間通信

クセスすることができる名前空間を切り分けることにより、制限を行う。

タスクの親子関係から、子は親のアクセス不可能な制限空間を引き継ぐ。親は子の生成に対して自分の制限にさらに制限を付加して、子のタスクを生成する。

これにより、あるタスク以下のタスクはデバイスには触ることができないなど、アクセスコントロールが可能となる。また、子は親の制限を必ず引き継ぐために、子が親よりもアクセス権の高いタスクを作り、システムを壊すことを防ぐことができる。

このアクセスコントロールの図を図 6 に示す。

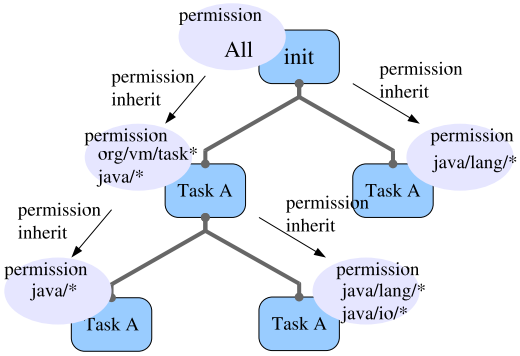

図 6 アクセスコントロール

5．実装

本システムはシステムの実装に JikesRVM[3]を使用している。JikesRVM は Java 言語で書かれた JavaVM であり、Java 言語を直接ネイティブコードに変換するコンパイラを持っている。本システムではバイトコードのコンパイラ、JavaVM として使用している。

JikesRVM はスレッドやメモリ管理に OS が提供する機能を利用しており、直接 C 言語のコードを呼出している。そのため Java 言語によるメモリ管理のコードを記述し、現在は JavaVM のイメージが起動し、シングルタスクの状態で動作している。

5.1 システムコールコスト

Java によるオーバーヘッドを比較するために、Linux とのシステムコールと速度を比較した。

実験に使用したマシンは、Celeron 2.4GHz、メモリ 512M バイトである。Linux のバージョンは 2.4.27 を使用した。

比較した処理は次のものである。

1．Linux で getpid システムコールの呼出し
2．JavaVM 上でメソッド呼出しによるシステムコール
3．Linux 上での C 言語の関数呼出し

この比較を表 1 に示す。

表 1 システムコールの呼出し時間

	1000 万回呼出し(sec)	Linux システムコールとの比(%)
getpid	4.66	100.00
メソッドコール	0.20	4.29
関数呼出し	0.07	1.50

表 1 より、JavaVM のメソッドコールを用いたシステムコールの呼出しを用いることで、getpid の処理時間に比べ約 4.3%の時間でシステムコールを行うことができる。C 言語の関数呼出しとの比較では約 3 倍の速度差となった。

システムコールをメソッドと等価の呼出しとすることで、JavaVM 上のシステムでは従来の OS と比べシステムコールのコストを下げることができる。C 言語の関数呼出しと JavaVM 上のメソッド呼出しでは、Java のオブジェクトを使用することによるコストが約 3 倍のコストが掛かる結果となった。この結果から通常の呼出しにおいては、従来の OS よりもコストが高いことが予想されるが、従来のミドルウェアなどの抽象化層を直接 OS が提供しており、層をなくすことができるため、速度の改善が可能だと考える。

6．関連研究

Java を使用した OS としては JX[4]と JavaOS[5]がある。どちらの OS もマイクロカーネル上に

JavaVMを載せることでセキュアな実行環境を提供するOSとしている。アプリケーションに対してデバイスのアドレス空間を公開し、アプリケーションからのデバイスの制御が可能となっており、割込み、例外、デバイスのバスなどをオブジェクトとして公開することにより、Java言語によるアプリケーションによるシステムのポリシ、メカニズム記述が可能となっている。アプリケーションをJavaに絞ることで、アプリケーションとOSとのインタフェースのギャップをなくしている。しかし、仮想マシン自体はネイティブコードあり、また、マイクロカーネルを使用しているので、システム全体に対して仮想マシンの利点を適用できてはいない。

再帰的な仮想マシンを使用して、リソース管理が行えるようにしたOSとしてFluke[6]がある。仮想マシンを再帰的に割り当てることでリソース管理をアプリケーションに対して行うことができる。しかし、従来のネイティブコードを中心とした実装方法であり、バッファのオーバフローなどの問題を解決することはできていない。

立野[7]らはJavaによるTCP/IPプロトコルスタックの実装を行っている。Java上でシステムモジュールを実装することで、モジュールの追加、削除における高い柔軟性を持たせることを可能にするために、Java上で記述するときに問題になるハードウェアアクセスの手段の解決策を提案している。バッファ管理の改善により実用的な速度で実現することが可能だと主張しており、本研究においても効率的な構成をとることにより、実用的な速度が得られることが考えられる。

7．おわりに

本稿では仮想マシン環境を指向したOSの構成法について述べた。ソフトウェアをすべて仮想マシン上で動作させる構成にすることで、実CPUを用いることで起きていたメモリ関連のバグをなくすことができる。また、名前を使用した命令セットをシステムに用いることで、従来ハードウェアに頼っていた機構をソフトウェアで実装することができる。これによりアプリケーションに対して拡張性の高さを提供しながら、OSの安全な動作環境を実現することができる。

今後の課題としては、OSの各ポリシをJavaVM上での実装、評価をし、アプリケーションレベルも含めた性能、有効性について評価を行うことが挙げられる。

参 考 文 献

[1] Dawson R. Engler, M. Frans Kaashoek, James O'Toole Jr, Exokernel: an Operating System architecture for application-levelresource managemen, Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)

[2] Robert Wahbe, Steven Lucco, Thomas Anderson, Susan Graham, Efficient Software-Based Fault Isolation, Proc. Fourteenth ACM Symposium on Operating Systems Principles (SOSP), December 1993, pages 203 – 216

[3] B. Alpern , V. Litvinov , C. R. Attanasio , M. F. Mergen , J. J. Barton , T. Ngo , M. G. Burke , J. R. Russell , P. Cheng , V. Sarkar , J.-D. Choi , M. J. Serrano , A. Cocchi , J. C. Shepherd , S. J. Fink , S. E. Smith , D. Grove , V. C. Sreedhar , M. Hind , H. Srinivasan , S. F. Hummel , J. Whaley , D. Lieber, The Jalapeño virtual machine, IBM SYSTEMS JOURNAL, VOL 39, 2000

[4] Michael Golm and Meik Felser and Christian Wawersich and Juergen Kleinoeder, The JX Operating System, 2002 USENIX Annual Technical Conference, 2002, pages 45 – 58

[5] Saulpaugh Tom and Mirho Charles, 訳 油井尊, インサイドJava0Sオペレーティングシステム, ピアソンエデュケーション, 1998

[6] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, Stephen Clawson, Microkernels Meet Recursive Virtual Machines, Second Symposium on Operating Systems Design and Implementation, 1996

[7] 立野広樹, 萩野達也, JavaによるTCP/IPプロトコルスタックの設計と実装, 情報処理学会OS研究会報告, 2000-OS-84, pp.99-106 (2000)