(19)日本国特許庁(JP)　(12)特　許　公　報(B2)　(11)特許番号
特許第4880121号
(P4880121)
(45)発行日　平成24年2月22日(2012.2.22)　(24)登録日　平成23年12月9日(2011.12.9)

(51)Int.Cl.　F I
G06F 9/45 (2006.01)　G06F 9/44 320F

請求項の数 2 (全 20 頁)

(21)出願番号	特願2000-565460 (P2000-565460)
(86)(22)出願日	平成11年8月10日 (1999.8.10)
(65)公表番号	特表2002-522844 (P2002-522844A)
(43)公表日	平成14年7月23日 (2002.7.23)
(86)国際出願番号	PCT/US1999/018158
(87)国際公開番号	WO2000/010081
(87)国際公開日	平成12年2月24日 (2000.2.24)
審査請求日	平成18年8月10日 (2006.8.10)
(31)優先権主張番号	09/134,073
(32)優先日	平成10年8月13日 (1998.8.13)
(33)優先権主張国	米国(US)

(73)特許権者　597004720
オラクル・アメリカ・インコーポレイテッド
アメリカ合衆国、９４０６５　カリフォルニア州、レッドウッド・ショアーズ、オラクル・パークウェイ、５００
(74)代理人　110001195
特許業務法人深見特許事務所
(72)発明者　アンガー，デイビッド
アメリカ合衆国・９４３０３・カリフォルニア州・パロ　アルト・ドラフトウッド　ドライブ・８４４

審査官　稲垣　良一

最終頁に続く

(54)【発明の名称】バーチャル・マシン環境でネイティブ・コードを変換し、実行する方法および装置

(57)【特許請求の範囲】
【請求項１】
コンピュータ・システムのバーチャル・マシンにより実行される方法であって、
リンクしたライブラリのネイティブ・コードを取得するステップと、
中間フォームへの前記ネイティブ・コードのバイナリ変換を実行するステップと、
前記中間フォームを処理して、メモリ・アクセス・コールおよびブロッキング・システム・コールが発生しているか否かを判断するステップと、
メモリ・アクセス・コールの発生が識別されたとき、メモリ・アクセス・バグをデバッグするための１つまたは複数のメモリ妥当性チェックを前記ネイティブ・コードの前記中間フォームに挿入するステップと、
ブロッキング・システム・コールの発生が識別されたとき、前記ブロッキング・システム・コールを非ブロッキング・システム・コール・バリアントで置き換えるステップと、
そして
前記修正された中間フォームをコンパイルおよびインタプリタして前記ネイティブ・コードにより定められたルーチンを実行するステップと
を有する方法。
【請求項２】
中間フォームへのネイティブ・コードのバイナリ変換を実行するステップと、
前記中間フォームを処理して、メモリ・アクセス・コールおよびブロッキング・システム・コールが発生しているか否かを判断するステップと、

メモリ・アクセス・コールの発生が識別されたとき、メモリ・アクセス・バグをデバッグするための１つまたは複数のメモリ妥当性チェックを前記ネイティブ・コードの前記中間フォームに挿入するステップと、
ブロッキング・システム・コールの発生が識別されたとき、前記ブロッキング・システム・コールを非ブロッキング・システム・コール・バリアントで置き換えるステップと、そして
前記修正された中間フォームをコンパイルおよびインタプリタして前記ネイティブ・コードにより定められたルーチンを実行するステップと
を有する方法を、コンピュータシステム上で実行されたバーチャル・マシンに実行させるためのコンピュータ読み取り可能なコードを記録したコンピュータで使用可能な記憶媒体。

【発明の詳細な説明】
【０００１】
（発明の背景）
１．発明の分野
本発明は、コンピュータ・システムの分野に関するものであり、より詳細には、バーチャル・マシン実行環境に関する。
【０００２】
Ｓｏｌａｒｉｓ、Ｓｕｎ、Ｓｕｎ Ｍｉｃｒｏｓｙｓｔｅｍｓ、Ｓｕｎロゴ、Ｊａｖａ、およびＪａｖａベースの商標およびロゴは、米国および他国のＳｕｎ Ｍｉｃｒｏｓｙｓｔｅｍｓ，Ｉｎｃ．の商標または登録商標である。
【０００３】
２．背景技術
Ｓｕｎ Ｍｉｃｒｏｓｙｓｔｅｍｓ（登録商標）によって開発されたＪａｖａ™プログラミング言語は、「ｗｒｉｔｅ ｏｎｃｅ，ｒｕｎ ａｎｙｗｈｅｒｅ」™言語であるという、他のプログラミング言語に優る利点を有している。Ｊａｖａプログラミング言語は、バイトコード・クラス・ファイルの形式で設計され、配布され、実行される、実質的にアプリケーションのためのプラットフォーム非依存な機構、つまり「アプレット」を用意している。Ｊａｖａバーチャル・マシンは、Ｊａｖａバーチャル・マシンを含んでいるすべてのコンピュータ・プラットフォームが同じバイトコード・クラス・ファイルを実行することができるように、バイトコードを必要なプラットフォーム依存命令セットに還元する処理を行う。Ｊａｖａプログラミング言語によってサポートされていない機能が必要なとき、バーチャル・マシン内で実行中のＪａｖａアプリケーションは、リンクされたライブラリ内で実装されるネイティブ・コード関数をコールすることができる。ネイティブ・コードは、Ｊａｖａプログラミングおよびその実行の制約を受けず、したがって、それほど制御の良くない実行動作という代償を払って、よりプラットフォームに固有なプログラム可能性を提供する。ＪａｖａアプリケーションおよびＪａｖａアプレットのための処理環境、ならびにネイティブ・コードの使用について、下記でより十分な説明を行う。
【０００４】
処理環境
Ｊａｖａプログラミング言語は、各プログラムが１つまたは複数のオブジェクト・クラスおよびインターフェースを含んでいるオブジェクト指向プログラミング言語である。プログラムがマシン依存の実行可能プログラム・コードにコンパイルされる多くのプログラミング言語とは異なり、Ｊａｖａプログラミング言語で書かれたクラスは、マシン非依存のバイトコード・クラス・ファイルにコンパイルされる。各クラスは、クラス・ファイル・フォーマットと呼ばれるプラットフォーム非依存フォーマットのコードおよびデータを含む。実行ビークルとして動作するコンピュータ・システムは、各クラス内のコードを実行することを担う、バーチャル・マシンと呼ばれるプログラムを含む。
【０００５】
アプリケーションは、独立型Ｊａｖａアプリケーションとして、またはＨＴＭＬ（ハイパ

ーテキスト・マークアップ言語）文書内のアプレット・タグによって識別され、ブラウザ・アプリケーションによってロードされるＪａｖａ「アプレット」として設計することができる。アプリケーションまたはアプレットに関連付けられたクラス・ファイルは、ローカル・コンピュータ・システム上に、またはネットワークを介してアクセス可能なサーバ上に記憶することができる。各クラスは、必要に応じて、「クラス・ローダ」によってＪａｖａバーチャル・マシンにロードされる。

【０００６】

ネットワーク上で、サーバからクラスに対するアクセスをクライアントに提供するために、サーバ上でウェブ・サーバ・アプリケーションが実行されて、「ウェブ・ページ」とも呼ばれるＨＴＭＬ文書に対する、ＵＲＬ（ユニフォーム・リソース・ロケータ）を含んだＨＴＴＰ要求に応答する。クライアント・プラットフォーム上で実行中のブラウザ・アプリケーションが、ＨＴＭＬ文書を受け取ったとき（例えば、ＵＲＬをウェブ・サーバに転送することによってＨＴＭＬ文書を要求した結果として）、ブラウザ・アプリケーションは、そのＨＴＭＬを構文解析して、そのＨＴＭＬ文書内でアプレット・タグに出会ったとき、指定のバイトコード・クラス・ファイルのダウンロードを自動的に開始する。

【０００７】

Ｊａｖａアプレットのクラスは、Ｊａｖａアプレットの実行中に最初に参照されたとき、オンデマンドで、ネットワークから（サーバ上に記憶されている）、またはローカル・ファイル・システムからロードされる。バーチャル・マシンは、各クラス・ファイルを探し出して、ロードし、クラス・ファイルの構文解析を行って、クラスの様々な構成要素にメモリを割振り、そのクラスを他の既にロードされているクラスとリンクする。この処理は、クラス内のコードが、バーチャル・マシンによって容易に実行可能なようにする。

【０００８】

ＪａｖａアプリケーションおよびＪａｖａアプレットはしばしば、クラス・ライブラリを使用する。クラス・ライブラリ内のクラスは、「ネイティブ・メソッド」と呼ばれるものを含む。アプリケーションおよびアプレットは、時として、ネイティブ・メソッドを有しているクラスを含む。ネイティブ・メソッドは、「ネイティブ」というキーワード、スレッドの名前、メソッドの戻り値の型、およびメソッドに渡されるすべてのパラメータを指定する。Ｊａｖａプログラミング言語で書かれる「標準メソッド」（すなわち、非ネイティブ・メソッド）とは対照的に、ネイティブ・メソッドには、それぞれのクラス内でのボディが存在しない。その代わりに、ネイティブ・メソッドのルーチンは、リンクされたライブラリをサポートする任意のプラットフォームに固有のリンク機能を使用して実行時にバーチャル・マシン内の任意のクラスと動的にリンクされる、ネイティブ・コード（例えば、ＣまたはＣ＋＋プログラミング言語で書かれ、バイナリ・フォームにコンパイルされたコード）によって実行される。

【０００９】

例えば、Ｓｏｌａｒｉｓ™またはＵＮＩＸの環境では、バイナリ・フォームのネイティブ・コードを含むリンクされたライブラリは、「ｓｏ．」ファイルとして書かれた「共有オブジェクト」ライブラリとして実装することができる。Ｗｉｎｄｏｗｓの環境では、リンクされたライブラリは、「．ｄｌｌ」ファイルとして書かれた動的にリンクされた（または動的にロード可能な）ライブラリのフォームを取ることが可能である。ネイティブ・コードは、そうでなければ、Ｊａｖａプログラミング言語によってサポートされない機能、例えば、任意のプラットフォームの専用ハードウェア（例えば、表示ハードウェア）またはソフトウェア（例えば、データベース・ドライバ）とインターフェースをとることなどを実行するために使用することができる。ネイティブ・コードはまた、レンダリングなどの計算上インテンシブな機能をスピードアップするために使用することもできる。

【００１０】

ネイティブ・メソッドを含むクラスはまた、それれぞれのリンクされたライブラリをロードするコールも含む。

Ｓｙｓｔｅｍ．ＬｏａｄＬｉｂｒａｒｙ（“Ｓａｍｐｌｅ”）

ここで、「Ｓａｍｐｌｅ」は、システムを動作させるホスト（例えば、ＵＮＩＸ、Ｗｉｎｄｏｗｓ等）により、「ｌｉｂＳａｍｐｌｅ．ｓｏ」または「Ｓａｍｐｌｅ．ｄｌｌ」という名前が付けられたファイルに通常、格納されている、リンクされたライブラリの名前である。通常、リンクされたライブラリは、関連付けられたクラスがバーチャル・マシン内でインスタンス化された時点でロードされる。

【００１１】

ネイティブ・コードのリンクされたライブラリは、リンクされたライブラリがクラス内のネイティブ・メソッドのメソッド署名を認識できるようにするため、関連付けられたクラスのスタブおよびヘッダ情報とともにコンパイルされる。次に、ネイティブ・メソッドの実施が、リンクされたライブラリ内のネイティブ・コード関数（Ｃ関数など）として行われる。実行時に、ネイティブ・メソッドに対するコールが行われたとき、コールされたメソッドに対応するリンクされたライブラリ内の機能に制御が渡される（例えば、ネイティブ・メソッド・スタック上にネイティブ・メソッド・フレームを押し付けることを介して）。リンクされたライブラリ内のネイティブ・コードは、その機能を実行して、Ｊａｖａアプリケーションまたはｊａｖａアプレットに制御を戻す。

【００１２】

図１は、処理システムのためのコンパイル環境および実行環境を図示している。コンパイル環境で、ソフトウェア開発者が、データ構造、メソッド実装、および他のクラスに対する参照を含んだ、プログラマ可読クラス定義を含むソース・ファイル１００を作成する（例えば、Ｊａｖａプログラミング言語で）。ソース・ファイル１００は、Ｊａｖａコンパイラ１０１に提供され、これがソース・ファイル１００を、Ｊａｖａバーチャル・マシンによって実行可能なバイトコードを含むコンパイルされた「クラス」ファイル１０２にコンパイルする。バイトコード・クラス・ファイル１０２は、サーバ上に記憶され（例えば、一時記憶または永久記憶で）、ネットワークを介してダウンロードのために利用可能である。別法では、バイトコード・クラス・ファイル１０２は、ローカルに、クライアント・プラットフォーム上のディレクトリ内に格納することができる。

【００１３】

Ｊａｖａ実行環境は、バイトコード・クラス・ファイルを実行することができ、また実行中に必要なとき、オペレーティング・システム１０９に対してネイティブ・オペレーティング・システム（「Ｏ／Ｓ」）コールを実行することができる、Ｊａｖａバーチャル・マシン（ＪＶＭ）１０５を含む。Ｊａｖａバーチャル・マシン１０５は、バイトコード・クラスのマシン非依存性と基礎となっているコンピュータ・ハードウェアのマシン依存命令セットならびにオペレーティング・システム１０９のプラットフォーム依存コールとの間で抽象化のレベルを提供している。

【００１４】

クラス・ローダおよびバイトコード検証器（「クラス・ローダ」）１０３は、必要に応じてＪａｖａバーチャル・マシンに、バイトコード・クラス・ファイル１０２をロードすること、ならびにクラス・ライブラリ１０４をサポートすることを担う。クラス・ローダ１０３はまた、各クラスのバイトコードを検証して、セキュリティ規則の適切な実行および実施を維持する。実行時システム１０８のコンテキスト内では、バイトコードが、ハードウェア１１０内のプロセッサ（または複数のプロセッサ）によって実行され得るように、インタープリタ１０６がバイトコードを直接に実行するか、または「ジャスト・イン・タイム」（ＪＩＴ）コンパイラ１０７がバイトコードをマシン・コードに変換するかのいずれかである。例えば、リンクされたライブラリ１１１という形式のネイティブ・コードは、関連付けられたネイティブ・メソッドを含むクラス（例えば、クラス・ライブラリ１０４からの）がバーチャル・マシン内でインスタンス化されたときにロードされる。

【００１５】

バーチャル・マシン１０５の実行時システム１０８は、汎用スタック・アーキテクチャをサポートする。この汎用スタック・アーキテクチャが基礎となるハードウェア１１０によってサポートされる方法は、バーチャル・マシンの特定の実装によって決定され、バイト

コードが解釈されるスレッド、またはＪＩＴコンパイルされるスレッドに反映される。実行時システムの他のエレメントは、スレッド管理（例えば、スケジューリング）機構およびごみ収集機構を含む。

【００１６】

図２は、実行時システム１０８内のスタック・アーキテクチャをサポートする実行時データ・エリアを図示している。図２で、実行時データ・エリア２００は、１つまたは複数のスレッド・ベースのデータ・エリア２０７を含む。スレッド・ベースの各データ・エリア２０７は、プログラム・カウンタ・レジスタ（ＰＣ　ＲＥＧ）２０８、ローカル変数ポインタレジスタ（ＶＡＲＳ　ＲＥＧ）２０９、フレーム・レジスタ（ＦＲＡＭＥ　ＲＥＧ）２１０、オペランド・スタック・ポインタ・レジスタ（ＯＰＴＯＰ　ＲＥＧ）２１１、スタック２１２（例えば、標準メソッドの場合）、およびオプションとして、ネイティブ・メソッド・スタック２１６を含む。スタック２１２は、オペランド・スタック２１４およびローカル変数２１５を含んだ１つまたは複数のフレーム２１３を含む。ネイティブ・メソッド・スタック２１６は、１つまたは複数のネイティブ・メソッド・フレーム２１７を含む。

【００１７】

実行時データ・エリア２００はさらに、共有ヒープ２０１を含む。ヒープ２０１は、そこからすべてのクラスのインスタンスおよび配列に対するメモリが割振られる実行時データ・エリアである。共有ヒープ２０１は、すべてのスレッドの間で有されるメソッド・エリア２０２を含む。メソッド・エリア２０２は、ロードされた各クラス・ファイルから抽出された情報を記憶するための１つまたは複数のクラス・ベースのデータ・エリア２０３を含む。例えば、クラス・ベースのデータ・エリア２０３は、コンスタント・プール２０４、フィールドおよびメソッド・データ２０５、ならびにメソッドのためのコードおよびコンストラクタ２０６を含み得る。

【００１８】

バーチャル・マシンは、一度に多くの実行のスレッドをサポートすることができる。各スレッドは、それ自体のスレッド・ベースのデータ・エリア２０７を有する。どの時点でも、各スレッドは、そのスレッドに対する「現行メソッド」である、単一のスレッドのコードを実行している。「現行メソッド」がネイティブ・メソッドではない場合、プログラム・カウンタ・レジスタ２０８は、現在、実行されているバーチャル・マシン命令のアドレスを含む。「現行メソッド」がネイティブ・メソッドである場合、プログラム・カウンタ・レジスタ２０８の値は、未定義となる。フレーム・レジスタ２１０が、メソッド・エリア２０２内の現行メソッドのロケーションにポイントする。

【００１９】

各スレッドは、そのスレッドと同時に作成されたプライベート・スタック２１２を有する。スタック２１２は、スレッドによって呼び出された標準メソッドに関連付けられた１つまたは複数のフレーム２１３を格納する。フレーム２１３は、データおよび部分的結果を格納するため、ならびにダイナミック・リンキング、方法への戻り値、およびディスパッチ例外を実行するために使用される。標準・メソッドが呼び出される度に新規フレームが作成され、スタックに押し上げられる。既存フレームは、そのメソッドが完了するとスタックから外され、破壊される。スレッドによって作成されるフレームはそのスレッドにとって局所的であり、通常、他のどのスレッドからも直接参照することはできない。

【００２０】

現在実行中のメソッドのためのフレームである１つのフレームだけが、制御の所与のスレッドのいずれかのポイントで活動中である。このフレームは、「現行フレーム」と呼ばれ、そのメソッドは「現行メソッド」として知られている。フレームは、そのメソッドが別のメソッドを呼び出したり、そのメソッドが完了した場合は、現行でなくなる。メソッドが呼び出されると、新規フレームが作成され、制御がその新規フレームに移るとそれが現行となる。メソッドが戻ると、現行フレームはそのメソッドの呼び出しの結果がいかなるものであれ、それを前フレームに返す。次いで、前フレームが現行フレームになると、現

行フレームは破棄される。
【００２１】
各フレーム２１３は、一組の固有のローカル変数２１５と固有のオペランド・スタック２１４とを有する。ローカル変数ポインタ・レジスタ２０９は、現行フレームのローカル変数２１５を含むワードの配列の基数を指すポインタを含む。オペランド・スタック・ポインタ・レジスタ２１１は、現行フレームのオペランド・スタック２１４のトップを指す。大部分のバーチャル・マシン命令は、現行フレームのオペランド・スタックから値を取り、その上で動作し、同じオペランド・スタックに結果を戻す。オペランド・スタック２１４は、メソッドに引数を引き渡し、メソッドの結果を受け取ることにも使用される。
【００２２】
ネイティブ・メソッド・スタック２１６は、ネイティブ・メソッドをサポートしてネイティブ・メソッド・フレーム２１７を格納する。各ネイティブ・メソッド・フレームは、スレッド実行制御、メソッド引き数およびリンクされたライブラリでネイティブ・コード関数として実行される標準メソッドとネイティブ・メソッドとの間でやり取りされるメソッドの結果のための機構を提供する。
【００２３】
ネイティブ・メソッドは、クラスの標準メソッドとしてではなくリンクされたライブラリ内のネイティブ・コードによって実行されるので、ネイティブ・メソッドはＪａｖａプログラム言語およびバイトコード検証器によって課される制限を受けることがない。すなわち、コンパイルされたＪａｖａアプリケーションおよびアプレット用バイトコードと異なり、リンクされたライブラリにあるネイティブ・コードは、実行時にチェックなしで進行する、望まない、不正行為の対象となりやすくなる場合がある。例えば、「ワイルド」ポインタ（例えば、８エレメント配列で９エレメントを指すポインタなど、禁止された範囲を値が超えたポインタ）の発生や不適切な（制限された、すなわち範囲外の）メモリ位置をアドレス指定する可能性のあるメモリ・アクセス機構の使用により、メモリ・アクセス・エラーが起こることがある。したがって、ネイティブ・メソッドの使用により、大部分はポインタの使用に基づいて、特定バーチャル・マシンのデバッグの実行をより困難にするプログラム・バグの分類が可能になる。
【００２４】
さらに、ネイティブ・コードは、ブロッキング・システム・コール（すなわち、未指定の期間、外部イベントの発生を待つことのできるコール）を含むことができる。バーチャル・マシンがその固有のスレッド管理とスケジューリングとを実行するならば、リンクされたライブラリ内のネイティブ・コード関数に制御が渡された時に発生するブロッキング・システム・コールは、バーチャル・マシン全体の実行をブロックすることができる。
【００２５】
大部分のバーチャル・マシンを実行すると、「ネイティブ・スレッディング」を使用することによってネイティブ・コードに関連付けられたブロッキング問題が回避される。すなわち、バーチャル・マシンの複数のスレッドと、バーチャル・マシンが実行中の１つまたは複数のプログラム（例えば、アプリケーションおよび／またはアプレット）が、基礎となるプラットフォームのスレッド、例えばＵＮＩＸスレッドとして実行される。このメソッドでは、バーチャル・マシンのスレッドを同時に実行することができる。ただし、ネイティブ・スレッディングが使用されると、バーチャル・マシンは、計画中のスレッドに優先して基礎となるオペレーティング・システムに制御を引き渡さなければならない。したがって、ネイティブ・スレッディングを使用すると、スレッドの挙動はオペレーティング・システム依存およびハードウェア依存になる。バーチャル・マシン実行中の並行性に関係したバグの効果的なデバッグは不確かなものとなる。それは、ネイティブ・スレッディングを使用した、スレッドを実行する相対的なタイミングが、異なるオペレーティング・システムおよびハードウェアのプラットフォームを通して変化することがあるからである。
【００２６】

図３Ａおよび３Ｂは、実行環境下におけるスレッドの使用を示すブロック図である。図３Ａは、ネイティブ・スレッディングを使用しないバーチャル・マシンを含む。図３Ｂは、ネイティブ・スレッディングを使用するバーチャル・マシンを含む。

【００２７】

図３Ａで、オペレーティング・システム１０９はハードウェア１１０の上で実行されており、バーチャル・マシン１０５はオペレーティング・システム１０９の上で実行されている。バーチャル・マシン１０５では、アプレット１（３００）およびアプレット２（３０１）のような複数のアプリケーションおよび／またはアプレットが実行されている。アプレット１およびアプレット２は、それぞれに１つまたは複数のバイトコード・クラス・ファイルを含むことができる。リンクされたライブラリ（ＬＩＢ）は、ネイティブ・メソッドをサポートするためにアプレット２に関連付けられている。ライブラリ３０２は、関連付けられたネイティブ・メソッドを含むアプレット２のクラスがバーチャル・マシン１０５内にインスタンス化される時にロードおよびリンクされる。ライブラリ３０２のネイティブ・コードは、ライブラリ・リンキング機能をサポートするオペレーティング・システム１０９およびハードウェア１１０の上で直接に実行される。

【００２８】

実行の複数スレッドは、バーチャル・マシン１０５内で処理される。例えば、アプレット１は２つのスレッドＴ１およびＴ２を有することができ、アプレット２は２つのスレッドＴ５およびＴ６を有することができ、バーチャル・マシン自体は、ごみ収集などのバーチャル・マシンのプロセスを実行する２つのスレッドＴ３およびＴ４を有することができる。スレッドＴ１ないしＴ６は、バーチャル・マシン１０５内のＶＭスレッド・スケジューラ３０３によって管理およびスケジューリングされる。ＶＭスレッド・スケジューラ３０３は、例えば優先順位およびタイム・スライシング方法に基づいて、オペレーティング・システム・レベルで、グループＴ１ないしＴ６のどのスレッドが現在実行中のバーチャル・マシンのスレッド、ＴＶＭであるかを選択する。

【００２９】

Ｊａｖａバーチャル・マシンは、通常、実行中のスレッドが処理中のリソースを特定の間隔で他のスレッドに譲渡する「協調スケジューリング」をサポートするが、さもなければ現行スレッドの実行に関連付けられた遅延が起こる可能性がある。例えば、優先順位の高いスレッドは、譲渡操作を利用して現行スレッドに取って代わることができる。プロセッサ・リソースの譲渡は、標準メソッドにおいて明示的にプログラムされる必要はない。バーチャル・マシンは、協調してスケジューリングするために、解釈のプロセスやコンパイルされたコードの、メソッドが呼び出した時やループの途中（例えば、逆方向ブランチ）のような実行に適した時点で譲渡を挿入することができる。

【００３０】

オペレーティング・システム１０９は、選択されたバーチャル・マシン・スレッドＴＶＭを含む多くのスレッドに対していつでも同時に働くことができる。例えば、オペレーティング・システム１０９は、オペレーティング・システムの他のアプリケーションまたは処理をサポートするスレッドＴＡ－ＴＺを含むことが出来る。ＯＳスレッド・スケジューラ３０４は、グループＴＡ－ＴＺのどのスレッドおよびＴＶＭが、所与の時間に基礎となるハードウェア１１０によって実行されるかを決定する。ハードウェア１１０が複数のプロセッサをサポートしているならば、複数のスレッドを異なるプロセッサで同時に実行するようにＯＳスレッド・スケジューラ３０４によってスケジュールすることができる。

【００３１】

図３Ａの実施において、バーチャル・マシン・スレッド（例えば、Ｔ１－Ｔ６）は、例えば、図に示すように、スレッドＴ６はライブラリ３０２のネイティブ・コードによってサポートされているアプレット１のネイティブ・メソッドを呼び出すように、ネイティブ・メソッドのための機能を実行するために実行制御をリンクされたライブラリ（例えば、ＬＩＢ３０２）に移すことができる。スレッドＴ６は、スレッドＴ６が現行でバーチャル・マシン・スレッドＴＶＭとしてオペレーティング・システム１０９に渡されているので、

制御をライブラリ３０２に移すことができる。バーチャル・マシンの他のスレッドは、協調スケジューリングに従ってスレッドＴ６がイールド(yield)するのを待たなければならない。

【００３２】

ただし、ライブラリ３０２の制御の移動は、バーチャル・マシン実行の問題を増加させることになる。バーチャル・マシンで実行中のクラスは、通常、他のクラスのメソッドのみを呼び出し、一般的にシステムを直接呼び出すことはしない。ネイティブ・コードは、しかし、その機能依存であり、ブロッキング・システム・コールを頻繁に行うことができる。ネイティブ・コードはリンクされたライブラリでコンパイルされたコードとして独立に実行されるので、バーチャル・マシンのインタープリタおよびコンパイラはバイパスされ、制御が標準メソッドに戻されるまで協調スケジューリングを強行することはできない。バーチャル・マシンは、したがって、ネイティブ・コードに明示的なyield()コールを与えるネイティブ・コード・プログラマーに依存する。

【００３３】

ネイティブ・コード・ライブラリ３０２が、ファイルをダウンロードするためのＩ／Ｏコールのようなシステムコールをブロックしたら、バーチャル・マシン内のスレッドＴ６と、したがってオペレーティング・システム・レベルのスレッドＴＶＭは、システム・コールが完了するまで、例えばダウンロードが終了するまでブロックする。バーチャル・マシン全体の実行は、ライブラリ３０２のネイティブ・コードによって実行制御が維持されているので、システムコールの期間中もブロックされる。ブロッキング・システムコールが完了するのに比較的長時間掛かるので、バーチャル・マシン１０９のすべてのスレッドを同様にブロックされるのは好ましくない。アプレット１およびアプレット２ならびにバーチャル・マシン１０５のパフォーマンスは、ライブラリ３０２のブロッキング・システムコールによって低下する。このため、多くのバーチャル・マシンの実行において、図３Ｂに示すようにネイティブ・スレッディングが使用される。

【００３４】

図３Ｂでは、ＶＭスレッド・スケジューラ３０３は、バーチャル・マシンの複数のスレッドをオペレーティング・システム・レベルのスレッドとして実行する。このスレッドは、スレッドＴＶＭ１ないしＴＶＭｎとしてラベリングされる。ＶＭスレッド・スケジューラ３０３は、どのバーチャル・マシン・スレッド（Ｔ１－Ｔ６）がＯＳスレッドＴＶＭ１ないしＴＶｎとして所与の時間にオペレーティング・システム１０９に渡されるのかを決定する。バーチャル・マシン１０５の各スレッドが基礎となるオペレーティング・システム１０９の個別スレッドとして実行されるような極端な場合には、バーチャル・マシン１０５はＶＭスレッド・スケジューラ３０３の実行に先立つことができ、またスレッドのスケジューリングについてＯＳスレッド・スケジューラ３０４に完全に依存することができる。

【００３５】

図３Ｂを実行すると、複数のスレッドがバーチャル・マシン１０５において同時にアクティブになることができる。すなわち、ライブラリ３０２のネイティブ・コードによるブロッキング・システム・コールは、バーチャル・マシン１０５を完全にブロックすることができないということである。むしろ、グループＴＶＭ１ないしＴＶＭｎの１つのスレッド、すなわちライブラリ３０２に制御を渡したスレッド（すなわち、バーチャル・マシンのスレッドＴ６に対応するオペレーティング・システムのスレッド）は、ブロックされるが、残りのスレッドＴＶＭ１ないしＴＶＭｎは実行することができる。

【００３６】

しかし、バーチャル・マシンの複数のスレッドをＯＳスレッドまたはネイティブ・スレッドとして実行することによって、バーチャル・マシン１０５は、バーチャル・マシンにおけるスレッドのスケジューリングを、ＶＭスレッド・スケジューラ３０３からスレッド・スケジューラ３０４に効果的に制御を譲渡する。ＶＭスレッド・スケジューラ３０３に制御が比較的欠けているので、バーチャル・マシンのスレッド間で同期エラーが発生するこ

とがある。複雑な問題については、ネイティブ・スレッディングがＯＳスレッド・スケジューラ３０４に依存しているため、バーチャル・マシン１０５およびアプレット１およびアプレット２が、異なるタイミング・パラメータおよびスケジューリング・プロセスを有する異なるオペレーティング・システム１０９および／または異なるハードウェア１１０上で実行される場合には、同期エラーは起こる可能性がないか、または別の形式で起こることがある。したがって、エラーは容易に反復することはできず、システムのデバッグはより複雑化する。

【００３７】

オブジェクト指向プログラミング

下記で参照のためにオブジェクト指向プログラミングの一般的な説明を行う。オブジェクト指向プログラミングは、ある種の基本的ビルディング・ブロックを組み合わせ、そのビルディング・ブロック間に関係を築くことによりコンピュータ・プログラムを作る方法である。オブジェクト指向プログラミング・システムにおけるビルディング・ブロックは「オブジェクト」と呼ばれる。１つのオブジェクトは、データ構造（１つまたはそれ以上のインスタンス変数）およびそのデータを使用しまたはそのデータに影響を及ぼすオペレーション（メソッド）をグループ化するプログラミング単位である。したがってオブジェクトは、データおよびそのデータ上で実行される１つまたはそれ以上のオペレーションまたは手続きから成る。データとオペレーションを単一のビルディング・ブロックに結合することを「カプセル化」と呼ぶ。

【００３８】

オブジェクトは「メッセージ」を受け取るとそのメソッドのうち１つを実行するように命令することができる。メッセージはあるメソッドを実行するためにオブジェクトに送信されるコマンドまたは命令である。メッセージは、メソッド選択（例えばメソッド名）およびゼロまたはそれ以上の引数からなる。メッセージは受信先のオブジェクトに、どのオペレーションを実行するのかを伝える。

【００３９】

オブジェクト指向プログラミングの一利点はメソッドを呼び出すメソッドである。メッセージがオブジェクトに送信される際、メッセージはオブジェクトに対してあるメソッドをどのように実行するかを命令する必要がない。必要なのはオブジェクトがメソッドを実行するよう要求することだけである。これによりプログラム開発が非常に簡易になる。

【００４０】

オブジェクト指向プログラミングの言語は主に「クラス」体系に基づく。クラスに基づくオブジェクト指向プログラミング体系の一例は、一般にアディソン・ウェスレー出版社から１９８９年に出版されたアデル・ゴールドバーグおよびデビッド・ロブソン著「Ｓｍａｌｌｔａｌｋ－８０：Ｔｈｅ　Ｌａｎｇｕａｇｅ」で説明される。

【００４１】

クラスは通常、クラスについてのフィールド（例えば変数）およびメソッドの両方を含むオブジェクトのタイプを定義する。オブジェクト・クラスを使用してオブジェクトの特定のインスタンスを生成することができる。オブジェクト・クラスのインスタンスは、その変数およびクラスについて定義されたメソッドを含む。同一クラスの複数のインスタンスは、オブジェクトクラスから生成することができる。オブジェクト・クラスから生成された各インスタンスは、同一のタイプまたはクラスであるということになる。

【００４２】

例として説明すると、従業員オブジェクト・クラスは「ｎａｍｅ」および「ｓａｌａｒｙ」インスタンス変数と「ｓｅｔ＿ｓａｌａｒｙ」メソッドを含むことができる。従業員オブジェクト・クラスのインスタンスは、組織内の各従業員について生成すなわちインスタンス化することができる。各オブジェクト・インスタンスは「ｅｍｐｌｏｙｅｅ」タイプであるということになる。各従業員オブジェクト・インスタンスは、「ｎａｍｅ」および「ｓａｌａｒｙ」インスタンス変数と「ｓｅｔ＿ｓａｌａｒｙ」メソッドを含む。各従業員オブジェクト・インスタンス内で「ｎａｍｅ」および「ｓａｌａｒｙ」の変数と関連付

けられた値は、組織内の従業員の氏名および給料を含む。メッセージがある従業員の従業員オブジェクト・インスタンスに送信されると、「ｓｅｔ＿ｓａｌａｒｙ」メソッドを呼び出して、従業員の給料（すなわち、従業員の従業員オブジェクト内の「給料」変数と関連付けられた値）を変更させることができる。

【００４３】

クラスの階層は、オブジェクト・クラス定義が１つまたはそれ以上のサブクラスを持つように定義することができる。サブクラスはその親（およびそのまた親などの）定義を継承する。階層中の各サブクラスは、その親のクラスが指定する振る舞いを追加または変更することがある。オブジェクト指向プログラミング言語の中には、サブクラスが１つ以上の親クラスからクラス定義を継承する多重継承を支持するものもある。Ｊａｖａプログラミング言語など他のプログラミング言語は単一継承だけを支持し、その場合サブクラスは１つだけの親クラスのクラス定義を継承するように制限される。Ｊａｖａプログラミング言語は、定数および抽象メソッド宣言のセットを含む「インターフェース」として知られるメカニズムをも用意している。オブジェクト・クラスは、インターフェース内で定義された抽象メソッドを実施することができる。単一継承および多重継承はどちらもインターフェースに使用することができる。すなわちインターフェースは、１つ以上の親インターフェースからインターフェース定義を継承することができる。

【００４４】

オブジェクトは、オブジェクト指向プログラミング環境において関連するコードおよび変数を含むモジュールを指すのに使用される一般的な用語である。ソフトウェア・アプリケーションにはオブジェクト指向プログラミング言語を使用して書き込みをすることができ、その場合プログラムの機能性はオブジェクトを使用して実施される。

【００４５】

発明の概要

バーチャル・マシン環境においてネイティブ・コードを変換および実行するメソッドおよび装置が提供され、ポインタ・チェッキング、スレッド制御、他の有用な特性が可能になる。バーチャル・マシン実施のデバッグは、ネイティブ・コードのバイナリ変換によって容易になる。バイナリ変換はプラットフォームの独立性を高め、スレッド管理およびスケジューリングをより制御することができるようになり、ネイティブ・コード内のメモリ・アクセス・エラー識別に対する予防手段をとる。バーチャル・マシン環境でネイティブ・コードが実行されると、ネイティブ・コードは中間フォームに変換される。この中間フォームは処理され、どこでメモリ・アクセスおよびブロッキング・システム・コールが発生しているのかを判断する。妥当性チェックがメモリ・アクセス・コールに挿入され、各コールがアクセスすべき各メモリの一部が許容範囲内にあるかを判断する。すなわちワイルド・ポインタおよびネイティブ・コードに関連するメモリ・アクセス・エラーの他のソースが識別される。ブロッキング・システム・コールは非ブロッキングの変形と入れ換えられ、「イールド（ｙｉｅｌｄ）」オペレーションがシステム・コールおよびループに挿入される。

【００４６】

メモリ・アクセス妥当性チェックおよび非ブロッキング・システム・コールを組み込む修正されたネイティブ・コードはバーチャル・マシンによりコンパイルおよびインタープリートされ、ネイティブ・コードが定義するルーチンを実行する。修正されたネイティブ・コードは他のスレッドをブロックしないので、スレッドのスケジューリングは基本となるオペレーティング・システムではなくバーチャル・マシンにより管理され、連携スケジューリングが実行される。

【００４７】

発明の詳細な説明

本発明は、バーチャル・マシン環境でネイティブ・コードを変換し、実行するメソッドと装置である。以下の説明では、本発明の実施形態をより完全に説明するために多くの特定の詳細が述べられる。しかしながら、当業者にはそれらの特定の詳細なしでも本発明を実

施できることは明らかである。他の例では良く知られた要素は本発明を曖昧にしないようにその詳細については述べない。

【００４８】

Ｊａｖａプログラミング・ランゲージ及びＪａｖａバーチャル・マシンに関して述べるが、本発明はネイティブ・メソッド又は関数を含むいかなるバーチャル・マシンでも実施することができる。

【００４９】

コンピュータ実行環境（ハードウエア）の実施形態

本発明実施形態は、図４に示されたコンピュータ４００のような一般目的のコンピュータで実行されるコンピュータ可読コードの形のコンピュータ・ソフトウエアとして実装することができ、またはそのようなコンピュータで走るＪａｖａ実行環境で実行可能なバイトコード・クラス・ファイルの形のコンピュータ・ソフトウエアとして実装することができる。キーボード４１０とマウス４１１が双方向システムバス４１８に接続されている。キーボードとマウスはユーザ入力をコンピュータ・システムへ送り、そのユーザ入力をプロセッサ４１３へ送るためのものである。他の適切な入力デバイスがマウス４１１やキーボード４１０に追加し又はそれらに代えて用いることも可能である。双方向システムバス４１８に接続されたＩ／Ｏ（入出力）ユニット４１９はプリンタ、Ａ／Ｖ（オーディオ／ビデオ）Ｉ／ＯなどのＩ／Ｏ要素を代表している。

【００５０】

コンピュータ４００はビデオ・メモリ４１４、メイン・メモリ４１５及び大容量記憶装置４１０を含む。それらはすべてキーボード４１０，マウス４１１及びプロセッサ４１３とともに双方向システムバス４１８に接続されている。大容量記憶装置は、磁気的、光学的又は光磁気的記憶システムまたはその他のいかなる利用可能な大容量技術のような固定及び着脱自在なメディア双方を含む。バス４１８は、例えば、ビデオメモリ４１４やメインメモリ４１５をアドレスするアドレス・ラインを含む。同様に、システムバス４１８は、例えば、プロセッサ４１３、メインメモリ４１５，ビデオメモリ４１４及び大容量記憶装置４１２のような部材の間でデータを転送するデータバスをも含む。データとアドレス用の個々のラインの代わりに多重化データ／アドレス・ラインを用いても良い。

【００５１】

本発明の１実施形態では、プロセッサ４１３は、６８０Ｘ０プロセッサのようなモトローラによって製造されたマイクロプロセッサ、８０Ｘ８６又はペンティアム・プロセッサのようなインテルによって製造されたマイクロプロセッサ、又はサン・マイクロシステムズ・インコーポレーテッドからのＳＰＡＲＣマイクロプロセッサである。しかしながら、他の適切なマイクロプロセッサ又はマイクロ・コンピューを使用可能である。メインメモリ４１５は、ランダム・アクセス・メモリ（ＤＲＡＭ）を含む。ビデオ・メモリ４１４は、二重ポート・ビデオ・ランダム・アクセス・メモリである。ビデオ・メモリ４１４の一つのポートはビデオ増幅器４１６に接続されている。ビデオ増幅器４１６は陰極線管（ＣＲＴ）ラスタ・モニタ４１７を駆動するために使用される。ビデオ増幅器４１６は従来良く知られており、任意の適切な装置によって実現できる。この回路はビデオ・メモリ４１４に記憶されたピクセル・データをモニタ４１７によって使用するのに適したにラスタ信号に変換する。代わりに、ビデオ・メモリはフラット・パネル又は液晶ディスプレイ（ＬＣＤ）又はその他のデータを適切に表示するデバイスを駆動するために用いられる。

【００５２】

コンピュータ４００はバス４１８へ接続された通信インターフェース４２０を含む。通信インターフェース４２０は、ネットワーク４２１を介してローカル・ネットワーク４２２へ接続し、２方向データ通信を行っている。例えば、通信インターフェース４２０が統合サービス・デジタル・ネットワーク（ＩＳＤＮ）カード又はモデムである場合、通信インターフェース４２０は、ネットワーク・リンク４２１の一部となっている、電話線の対応するタイプのデータ通信接続を構成している。もし、通信インターフェース４２０がローカル・エリア・ネットワーク（ＬＡＮ）カードである場合、通信インターフェース４２０

はネットワーク・リンク４２１を介して適合するＬＡＮとデータ通信接続を行う。通信インターフェースは、ケーブル・モデム又はワイヤレス・インターフェースでも良い。どのような実施形態においても、通信インターフェース４２０は、情報の多くのタイプを現しているデジタル・データ・ストリームを運ぶ電子的、電磁的又は光学的信号を送りかつ受け取る。

【００５３】

ネットワーク・リンク４２１は一つ又はより多くのネットワークを通して他のデータ・デバイスにデータを通信する。例えば、ネットワーク・リンク４２１は、ローカル・ネットワーク４２２を通して、ローカル・サーバ・コンピュータ４２３へ又はインターネット・サービス・プロバイダ（ＩＳＰ）４２４によって動作させられるデータ装置へ接続されている。ＩＳＰ４２４は、今一般に「インターネット」４２５といわれているワールド・ワイド・パケット・データ通信サービスを通してデータ通信サービスを実施する。ローカル・ネットワーク４２２とインターネット４２５の双方はデジタル・データ・ストリームを運ぶ電子的、電磁的又は光学的信号を利用する。コンピュータへ及びコンピュータからデジタル・データを運ぶ、多くのネットワークを通る信号やネットワーク・リンク４２１上にあり通信インターフェースを通る信号は、典型的には情報を運ぶ搬送波の形である。

【００５４】

コンピュータ４００は、ネットワーク、ネットワーク・リンク４２１及び通信インターフェース４２０を通してプログラム・コードを含むメッセージを送り、かつデータを受け取る。例えば、インターネットでは、リモート・サーバ・コンピュータ４２６が、インターネット４２５、ＩＳＰ４２４、ローカルネットワーク４２２及び通信インタフェース４２０を通してアプリケーション・プログラムに対して要求されたコードを送る。

【００５５】

受け取られたコードはプロセッサ４１３によって受け取ったときに実行されてもよいが、後に実行するために大容量記憶装置４１２または他の不揮発性記憶装置に記憶しても良い。また、実行して記憶しても良い。このようにして、コンピュータ４００はアプリケーション・コードを搬送波の形で獲得する。本発明の実施形態においては、そのようなダウンロードされたアプリケーションの例には、バーチャル・マシン、クラス・ローダ、クラス・バイトコード・ファイル、クラス・ライブラリおよびここに述べられるネイティブ・コードを変換し、実行する装置のような、実行環境の一つ以上の要素を含んでいる。

【００５６】

アプリケーション・コードはコンピュータ・プログラム製品のいかなる形態にも組み込むことができる。コンピュータ・プログラム製品は、コンピュータ可読コード又はデータを記憶し、転送するように構成された、あるいはコンピュータ可読コード又はデータメディアが組み込まれたメディアを含む。コンピュータ・プログラム製品の例としては、ＣＤ－ＲＯＭディスク、ＲＯＭカード、フロッピー・ディスク、磁気テープ、コンピュータ・ハードディスク、ネットワーク上のサーバ及び搬送波などである。

【００５７】

上述したコンピュータ・システムは例示の目的だけである。本発明の実施形態は、バーチャル・マシンをサポートする、組み込まれたデバイス（例えばウエブ・ホーンなど）や「ｔｈｉｎ」クライアント処理環境（例えばネットワーク・コンピュータ（ＮＣ）その他）を含むコンピュータ・システム又はプログラミング環境、処理環境のいずれのタイプでも実施できる。

【００５８】

ネイティブ・コードのバイナリ変換

先に述べたように、バーチャル・マシン内で実行されるクラスはリンクされたライブラリ内のネイティブ・コード関数によって実施されるネイティブ・メソッドを含む。本発明の実施形態によれば、リンクされたライブラリのネイティブ・コードはバーチャル・マシンのコンポーネントによって処理され、実行され、従来技術のネイティブ・メソッド実行プロセスに対して共通スケジューリングが可能で、強化したデバッキング能力を与える。ネ

イティブ・コードの処理は、バイナリ変換手順の一部として、「ワイルド」ポインタによって引き起こされるようなメモリ・アクセス・バグに関するチェックを挿入すること、及びネイティブ・スレッディングの必要なしに、ブロッキング・システム・コールをバーチャル・マシンにおける共通スケジューリングを可能とする非ブロッキング・バリアントで置き換えることを含む。

【００５９】

バイナリ変換は典型的にはデバッキングオペレーションの間にバーチャル・マシンで可能で、通常の動作の間は不可能である。例えば、可能なとき、「System.loadLibrary()」コールが、バーチャル・マシン内でインタプリートされた又はコンパイルされた実行に関する指定されたライブラリのバイナリ変換となる。不可能なとき、指定されたライブラリは標準のメソッドでロードされ、かつリンクされれる。いくつかの実施形態では、デバッキング処理の間のみに実行できるのではなく、バイナリ変換がいつでも実行できる。

【００６０】

図５は本発明実施形態によるバイナリ変換を実施するためのメソッドのフロー図である。ステップ５００で、リンクされたライブラリからそれぞれのネイティブ・コードを獲得する。このステップは、例えば、それぞれのクラスがバーチャル・マシン内でインスタンス化されるときリンクされたライブラリのソース・ファイルを決定し、かつそのフォース・ファイルからネイティブ・コード（すなわちマシン・コード）のバイナリ・フォームを読みとることを含む。バイナリ変換はバーチャル・マシンにおける実行に先立って実施される。

【００６１】

ステップ５０１では、ネイティブ・コードのバイナリ・フォームがバーチャル・マシンのバイナリ変換コンポーネントによって、バイトコード、アブストラクト・シンタックス・ツリー又は制御フロー・グラフのような中間フォームに変換される。バイトコードは、例えば、Ｊａｖａコンパイラ（図１の要素１０１）によって生成された標準バイトコードと同じように実施される。アブストラクト・シンタックス・ツリーや制御フロー・グラフは、ツリーやグラフのノードとしての実行オペレーションを特定するプログラム実行を現している。典型的には、中間フォーム（ここでは変換されたフォームともいう）はメモリ・アクセス・ポイント及び／又はコール及びブランチ・オペレーションの識別を単純化するフォームである。

【００６２】

ステップ５０２では、メモリ・アクセス・コールの位置が決定され、かつ制限されたまたは制限区域外のメモリの部分にアクセスしようとしているならば、メモリ・アクセス・コールが実行している間に信号にチェックが挿入される。信号は、例えばエラー・メッセージを表示する（例えば、ダイアログボックスに）、ログファイルにエラーをログする、例外をスローすることを含み、またはそれらのアクションのいくつかの組み合わせを含む。実行中に、ワイルド・ポインタとネイティブ・コードに関連する他のメモリ・アクセス・バグのデバッキングが、不法な各メモリ・アクセス・イベントが起こったときにそれを報告することによって決定される。メモリ・アクセス・チェックの一部として、ネイティブ・コードの要素によって正規にアクセスできるメモリの部分がポインタ値と比較するために追跡される。

【００６３】

ステップ５０３で、ブロッキング・システム・コールが中間フォームで識別され、可能ならば、システム・コールの非ブロッキング・バリアントが挿入される。ステップ５０４で、「yield()」関数がコールとループに挿入される。ループに対するイールド・ポイント（yield()関数が挿入されるポイント）が、例えば、バックワード・ブランチ・オペレーションに基づいて決定される。ステップ５０３と５０４の効果は、可能な限り、バーチャル・マシン及びいかなる実行しているアプリケーション及び／又はアプレットの実行をネイティブ・コードのアクティビティに関する依存から自由にする。バーチャル・マシンの他のスレッドはネイティブ・コードのシステム・コールによってブロックされず、かつコ

ールでかつループ内で他の待機中のスレッドに処理リソースをイールドするためにイールド・ポイントが確立される。バーチャル・マシンがすべての関連するスレッドの共通スケジューリングを実行できる。この共通スケジューリングによって、同期又は同時に関連しているバグを信頼性を持って識別し、かつ基礎となるオペレーティング・システムやハードウエアと独立に修正する。

【００６４】

ステップ５０５で、中間のすなわち変換されたフォームにおけるリバイズされたネイティブ・コードがバーチャル・マシンでコンパイルされ、インタプリートされ、関数をそこで実行する。いくつかの実施形態では、さらなる変換ステップが実施され、中間フォームを標準インタープリタ及びＪＩＴコンパイラでインタプリートしたりコンパイルするのに適切なバイトコードに変換する。ネイティブ・コード関数の実行に関連するスレッドのスケジューリングがバーチャル・マシンの他のあらゆるインタプリートされた又はコンパイルされたプロセスのように、ＭＶスレッド・スケジューリング・プロセスによって制御される。メモリ・アクセス・チェックが実施されると、違反がログされる。必要な分離された非同期スレッドが生成され、バーチャル・マシンにおいて他のスレッドと独立に他の処理が起こることができるようになる。

【００６５】

場合により、ネイティブ・コードはステップ５０１において完全にはパース（構文解析）を行えない。ルーチンの始め又は見込まれたブランチ(computed branch) のようなコードのある部分（ないし側面）は、変換されたコードがステップ５０５において実行される（例えば、ルーチンが現実にコールされた時）までは知り得ないことがある。このために、変換プロセスは、ステップ５０５からステップ５０１へと、フィードバックの矢印５０６で示されるように戻ることができる。このフィードバックは、それまではパース（構文解析）をされていないネイティブ・コードをパース（構文解析）および変換（又は、既にパースをされたネイティブ・コードの再パース）を、実行過程で決定された新たな情報に基づいて行う目的のものである。

【００６６】

バイナリ変換を受けるネイティブ・メソッドの一般的な中間フォームが図６Ａおよび図６Ｂに示されている。図６Ａは、一例としての実行ブロックの一般化したコントロール・フローを示す図であって、本発明の実施形態に従ってネイティブ・コードのブロックを中間フォームへと変換することが例示されている。図６Ｂは、図６Ａの一般化したコントロール・フローに本発明の実施形態に従って変形を付加したものを示す。この実施形態では、ループ内でのイールド点の表示はバックワード・ブランチ・オペレーションの発生に基づいている。

【００６７】

例示したオペレーション（動作）の表示のために、図６Ａおよび図６Ｂでは、次の略号が用いられている。

ＲＤ　＝メモリ・リード・オペレーション
ＷＲ　＝メモリ・ライト・オペレーション
ＢＲ　＝ブランチ・オペレーション（例：if）
ＭＣ　＝メソッド（関数）コール
ＢＳＣ＝ブロッキング・システム・コール
ＯＰ　＝他の一般的・オペレーション（その他）
ＣＨＫ＝ポインタ・チェック・オペレーション
ＹＬＤ＝イールド・オペレーション
ＮＢＳＣ＝非ブロッキング・システム・コール
ＦＬＡＧ＝信号処理アクセス違反

【００６８】

図６Ａにおいて、実行ブロックは、一般的オペレーション６００から始まり、続いてリード・オペレーション６０３が行われ、さらに、一般的オペレーション６０５，６０６，６

０７が順に続く。一般的オペレーション６０７の後で、ライト・オペレーション６１０が遂行され、続けて、一般的オペレーション６１２，６１３、そして、ブランチ・オペレーション６１４が行われる。このブランチ・オペレーションでは、一般的オペレーション６１６へと進むか、一般的オペレーション６０６へと分岐して戻るかされる。一般的オペレーション６１６から、メソッド・コール６１８が行われ、それに、一般的オペレーション６１９，ブロッキング・システム・コール６２１Ａ，一般的オペレーション６２２が続く。

【００６９】

バイナリ変換にとって興味あるオペレーションは、リード・オペレーション６０３，ライト・オペレーション６１０，ブランチ・オペレーション６１４，メソッド・コール６１８，ブロッキング・システム・コール６２１Ａであり、それらは図中で強調されている。メモリ・アクセスに関するリード・オペレーション６０３およびライト・オペレーション６１０は、ポインタ・チェックの挿入のためのセットである。ブランチ・オペレーション６１４およびメソッド・コール６１８は、yield() コールの挿入のためのセットである。ブロッキング・システム・コール６２１Ａは、非ブロッキング・システム・コール・バリアントで置換するためのセットである。

【００７０】

図６Ｂには、図６Ａの実行ブロックの中間フォームに対する変形が示されている。リード・オペレーション６０３は、ポインタ・チェック・オペレーション６０１，ブランチ・オペレーション６０２，リード・オペレーション６０３およびフラッグ・オペレーション６０４で置換されている。チェック・オペレーション６０１は、ポインタ値がリーガル（適法な）レンジ内か否かを決定し、それにブランチ・オペレーション６０２が続く。ブランチ・オペレーション６０２は、ポインタが有効であればリード・オペレーション６０３を行い、さもなければ、ポインタ・チェックで無効なポインタが示されていることを知らせるフラグ・オペレーション６０４を遂行する。オペレーション６０３，６０４は何れもオペレーション６０５へ進む。

【００７１】

ライト・オペレーション６１０は、リード・オペレーション６０３のために行われた上述した挿入に類似のチェックおよびフラッグの挿入を受ける。ライト・オペレーション６１０は、ポインタ・チェック・オペレーション６０８，ブランチ・オペレーション６０９，ライト・オペレーション６１０およびフラッグ・オペレーション６１１で置換される。ポインタ・チェック・オペレーション６０８は、ポインタ値がリーガル・レンジ内にあるか否かを決定するものであり、それにブランチ・オペレーション６０９が続く。ブランチ・オペレーション６０９は、ポインタが有効であればライト・オペレーション６１０を実行し、さもなければ、ポインタ・チェックにより無効ポインタと示されたことを知らせるフラグ・オペレーションを遂行する。オペレーション６１０，６１１は何れもオペレーション６１２へ進む。

【００７２】

バックワード・ブランチ・オペレーション６１４には、オペレーション６０６へのリターン・ループ内に挿入されたイールド・オペレーション６１５が伴っている。挿入されたイールド・オペレーション６１５によって、他のスレッドには、ブランチ・オペレーション６１４により形成されるループが今一度開始される前にプロセッサ・リソースを得る機会が許される。これにより、長いループの再帰プロセスのために、他のスレッドからプロセッサ・リソースが奪われることが防止される。同様に、イールド・オペレーション６１７がメソッド・コール６１８の前に挿入され、他のスレッドは、もし必要であれば、カレント・スレッドによる新たなメソッドの開始前に実行を許される。

【００７３】

ブロッキング・システム・コール６２１Ａは、図６Ｂにおいて、非ブロッキング・システム・コール６２１Ｂで置換されている。イールド・オペレーション（６２０）を、任意的に、システム・コールの前に挿入してもよい。必要であれば、非ブロッキング・システム

・コール６２１Ｂは、新たな非同期スレッドを生成して変換された関数の動作を独立的に実行されるスレッドとして遂行し得る。

【００７４】

バーチャル・マシンによりインタープリートまたはコンパイルをされると、リバイズされた実行ブロック（図６Ｂ）は、従来技術でのネイティブ・メソッドの実行に比べて、意味のあるデバッギングおよびスケジューリングの利点を提供する。

【００７５】

図７は、本発明の実施形態に従ってバイナリ変換を実現する実行環境を示すブロック図を示す。図７において、オペレーティング・システム１０９はハードウエア１１０でランし、バーチャル・マシン１０５はオペレーティング・システム１０９上でランする。オペレーティング・システム１０９はハードウエア１１０によってサポートされる。図３Ａおよび図３Ｂに示されるように、バーチャル・マシン１０５およびオペレーティング・システム１０９は、スレッド実行の管理のために、ＶＭスレッド・スケジューラ３０３およびＯＳスレッド・スケジューラ３０４をそれぞれ有する。また、バーチャル・マシン１０５は、バイナリ変換・プロセス７０１を有する。

【００７６】

バーチャル・マシン１０５内で実行されるのは、アプレット１（３００）およびアプレット２（３０１）のような複数のアプリケーション及び／又はアプレットである。アプレット１およびアプレット２それぞれは、１以上のバイトコードのクラス・ファイルを含むことができる。リンクされたライブラリ（ＬＩＢ）３０２は、アプレット２に付随させられてネイティブ・メソッドをサポートする。ライブラリ・ファイル３０２のネイティブ・コードは、バーチャル・マシン１０５のバイナリ変換・プロセス７０１によってパース（構文解析）および変換（トランスレーション）されて、変換されたライブラリ７００が発生される。

【００７７】

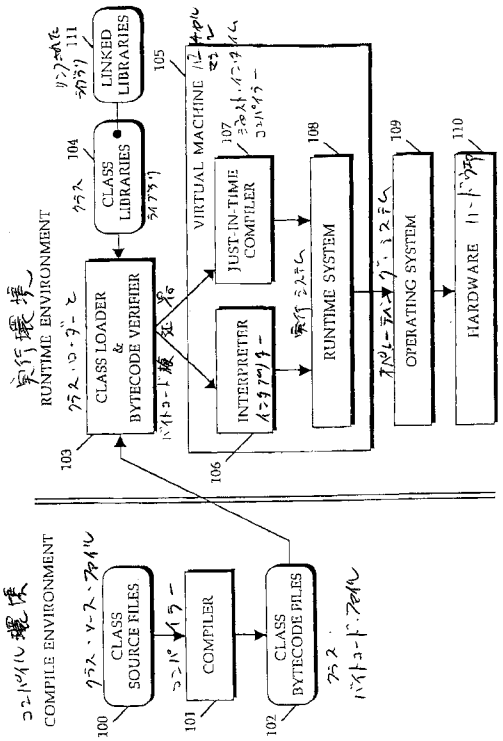
変換されたライブラリ７００は、ネイティブ・コードの中間フォーム（メモリ・アクセス・チェック，イールド，非ブロッキング・システム・コール・バリアントを含んでいる）から成っている。アプレット２のネイティブ・メソッドがスレッドＴ６によってコールされると、変換されたライブラリ７００はバーチャル・マシン１０５内でインタープリートまたはコンパイルをされて所望の関数が実行される。変換されたライブラリ７００の中間フォームに依存して、変換されたライブラリを得るためのインタープリートまたはコンパイルのプロセスは、アプレット１およびアプレット２のクラスに適用されるインタープリートまたはコンパイルのプロセスと異なることも、異ならないこともある。しかし、バーチャル・マシン１０５によって、変換されたライブラリ７００に対して行われる一般的オペレーションや実行される管理は、アプレット１およびアプレット２のためのものと一致している。実施形態によっては、変換されたライブラリ７００は、スレッドＴ６において、ネイティブ・メソッド・スタック２１６を介するのではなく、スタック２１２のフレームを介して、処理され得る。変換されたライブラリ７００は、あたかも、不変更のネイティブ・コード中の欠点を持たない追加のスタンダード・メソッドを提供しているかのように、実行できる。

【００７８】

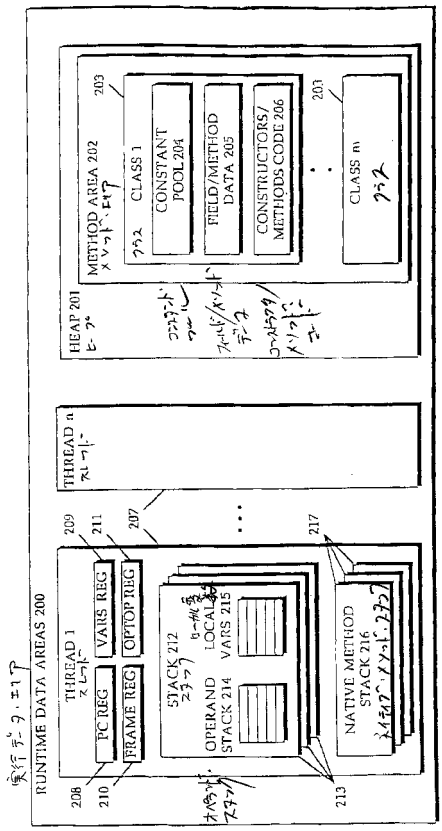
リンクされたライブラリの変換されたネイティブ・コード関数は、オペレーティング・システム１０９で遂行される、リンクされたライブラリの独立したプロセスではなくして、バーチャル・マシン１０５を通して実行される故に、そして、変換されたライブラリ７００内にブロッキング・コールが存在しない故に、ＶＭスレッド・スケジューラ３０３によって、共通スケジューリングを実行できる。従って、図３Ｂで実現されるようなネイティブ・スレッディングは不要となる。バーチャル・マシン１０５内でのスレッド・イベントの同期は、基礎にあるオペレーティング・システムおよびハードウエアから独立しており、デバッギングはオペレーティング・システムをベースとする同時発生事項の考慮なしに遂行可能となる。

【００７９】
以上にバーチャル・マシン環境においてネイティブ・コードの変換と実行を行うメソッドおよび装置を、１またはそれ以上の実施の形態とともに、記載した。本発明は、請求の範囲およびそれに均等な範囲によって定められるものである。
【図面の簡単な説明】
【図１】　コンパイルおよびランタイム環境のブロック図である。
【図２】　バーチャル・マシンの実施形態のランタイム・データ・アクセスのブロック図である。
【図３Ａ】　リンクされたライブラリを介して実装された多重アプレットおよびネイティブ・コードをサポートするバーチャル・マシンを有するランタイム環境のブロック図である。
【図３Ｂ】　ネイティブ・メソッド操作を使用するバーチャル・マシンを有するランタイム環境のブロック図である。
【図４】　本発明の実施形態のために適切な実行環境を提供することができるコンピュータ・システムの実施形態のブロック図である。
【図５】　本発明の実施形態に従ったバイナリ変換プロセスの流れ図である。
【図６Ａ】　本発明の実施形態に従ったネイティブ・コードのブロックの中間フォームへのバイナリ変換を図示する例示的実行ブロックの汎用的な制御フロー図である。
【図６Ｂ】　本発明の実施形態に従って実行される変更を加えた図６Ａの汎用的な制御フロー図である。
【図７】　本発明の実施形態に従ってネイティブ・コードのバイナリ変換を実施するバーチャル・マシンを有するコンピュータ・システムのブロック図である。

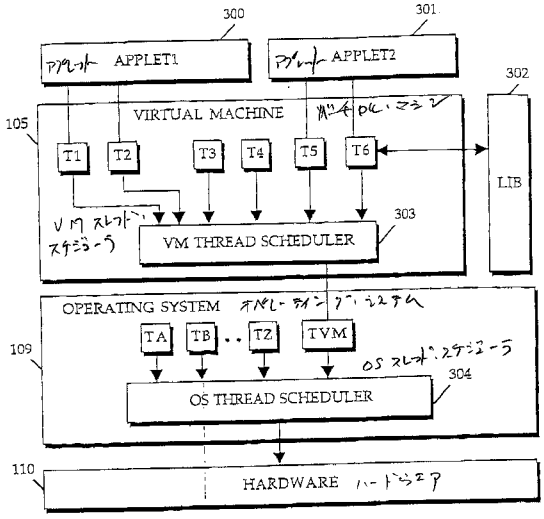
【図１】



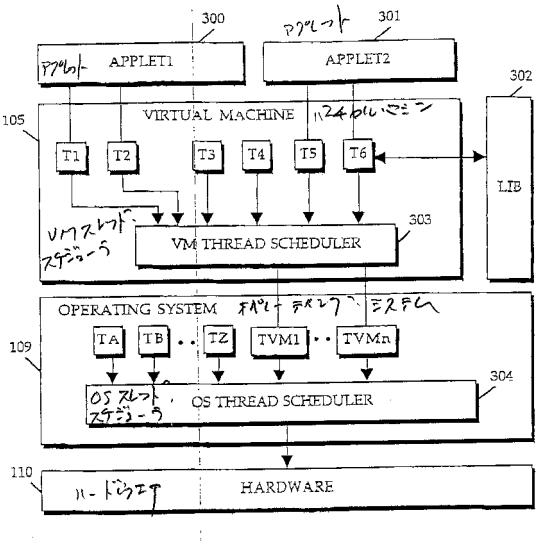
【図２】



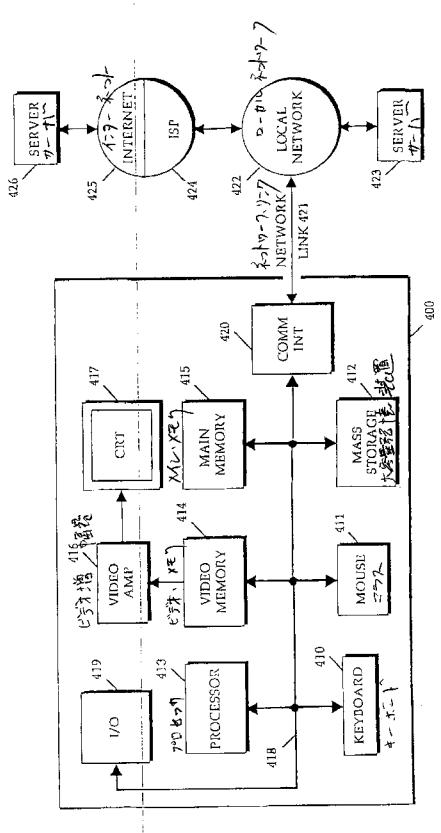
【図３Ａ】



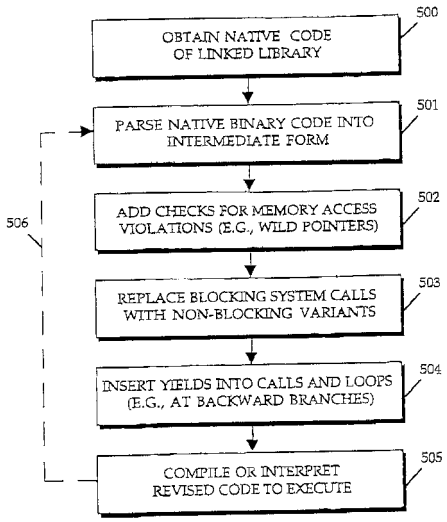
【図３Ｂ】



【図４】

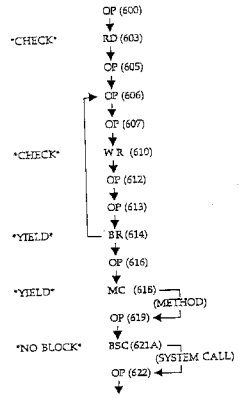


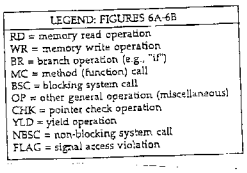
【図５】



５００・・リンクされたライブラリのネイティブ・コードを得る。
５０１・・ネイティブ・コードを中間フォームへとパース（構文解析）をする。
５０２・・メモリ・アクセス違反を探すチェック（例えば、ワイルド・ポインタ）を付加する。
５０３・・ブロッキング・システム・コールを非ブロッキング・バリアントで置き換える。
５０４・・コールおよびループ中にイールド（例えば、バックワード・ブランチ）を挿入する。
５０５・・リバイズされたコードを実行のためにコンパイルまたはインタープリートする。

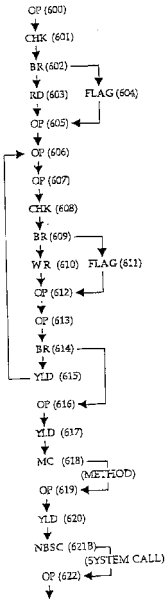
【図６Ａ】

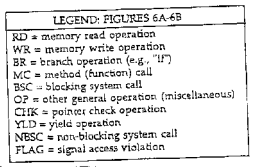




ＲＤ　＝メモリ・リード・オペレーション
ＷＲ　＝メモリ・ライト・オペレーション
ＢＲ　＝ブランチ・オペレーション（例：if）
ＭＣ　＝メソッド（関数）コール
ＢＳＣ＝ブロッキング・システム・コール
ＯＰ　＝他の一般的オペレーション（その他）
ＣＨＫ＝ポインタ・チェック・オペレーション
ＹＬＤ＝イールド・オペレーション
ＮＢＳＣ＝非ブロッキング・システム・コール
ＦＬＡＧ＝信号処理アクセス違反

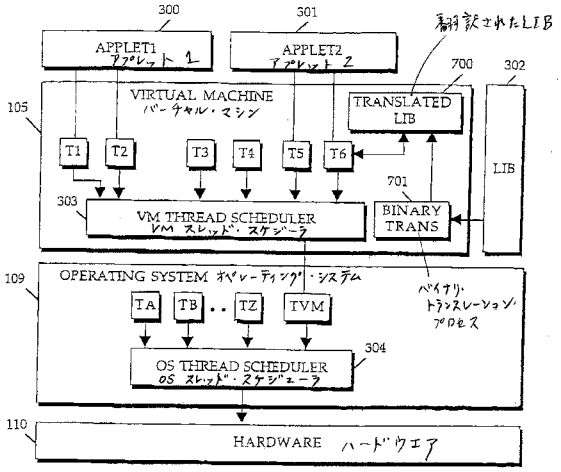
【図６Ｂ】





ＲＤ　＝メモリ・リード・オペレーション
ＷＲ　＝メモリ・ライト・オペレーション
ＢＲ　＝ブランチ・オペレーション（例：if）
ＭＣ　＝メソッド（関数）コール
ＢＳＣ＝ブロッキング・システム・コール
ＯＰ　＝他の一般的オペレーション（その他）
ＣＨＫ＝ポインタ・チェック・オペレーション
ＹＬＤ＝イールド・オペレーション
ＮＢＳＣ＝非ブロッキング・システム・コール
ＦＬＡＧ＝信号処理アクセス違反

【図７】



フロントページの続き

(56)参考文献 特開平６－２３１００３（ＪＰ，Ａ）
特開平８－１６１１７９（ＪＰ，Ａ）
国際公開第９７／３１３０９（WO，Ａ１）
特開平９－１７１４６７（ＪＰ，Ａ）
特開平６－２４２９４２（ＪＰ，Ａ）

(58)調査した分野(Int.Cl.，ＤＢ名)
G06F 9/45 - 9/455
G06F 11/28
G06F 9/46