(19)日本国特許庁(JP)　(12)特　許　公　報(B2)　(11)特許番号
特許第3722156号
(P3722156)
(45)発行日　平成17年11月30日(2005.11.30)　(24)登録日　平成17年9月22日(2005.9.22)

(51)Int.Cl.[7]　F I
G06F 9/46　G06F 9/46　340A
G06F 15/16　G06F 15/16

請求項の数 3　(全 15 頁)

(21)出願番号　特願平6-510792
(86)(22)出願日　平成5年7月1日(1993.7.1)
(65)公表番号　特表平8-502612
(43)公表日　平成8年3月19日(1996.3.19)
(86)国際出願番号　PCT/GB1993/001382
(87)国際公開番号　WO1994/010628
(87)国際公開日　平成6年5月11日(1994.5.11)
審査請求日　平成10年1月16日(1998.1.16)
審判番号　平11-14641
審判請求日　平成11年9月13日(1999.9.13)
(31)優先権主張番号　9222799.0
(32)優先日　平成4年10月30日(1992.10.30)
(33)優先権主張国　英国(GB)

早期審査対象出願

(73)特許権者
タオ　グループ　リミテッド
イギリス　アールジー6　1エイゼット　リーディング　アーレイ　サットン　パーク　アヴェニュ　サットンズ　ビジネス　パーク　62／63
(74)代理人
弁理士　谷　義一
(74)代理人
弁理士　阿部　和夫
(72)発明者　ヒンスレイ，クリストファー，アンドリュウ
英国　アールジー12　8ユーゼット　バークシャ州　ブラックネル　ハランド　パインズ　97

最終頁に続く

(54)【発明の名称】データ処理システム

(57)【特許請求の範囲】

【請求項１】

データ処理システムにおいて、

ローカル・メモリを有する少なくとも１つのデータ・プロセッサと、

前記システム内の前記プロセッサの各々に常時利用可能であり、前記システム内の前記プロセッサの各々が、前記プロセッサの１つに、または前記プロセッサの２つ以上に割り当てられたプロセスを実行し、または共同して実行することを可能にする機械コード・セグメントのセットと、

前記機械コード・セグメントのセットから、前記プロセッサの１つ以上により実行されるプロセスに必要な複数の機械コード・セグメントを選択する手段と、

前記プロセッサの１つ以上のローカル・メモリにおいて、前記選択された複数の機械コード・セグメントを、前記プロセッサの１つ以上により実行される実行可能プロセスにバインドする手段とを備え、

（ａ）前記機械コード・セグメントの各々は、必要に応じて、かつ他のいかなる機械コード・セグメントから独立に、前記プロセッサの各々により個別に選択可能であり、かつ前記プロセッサの各々のローカル・メモリに個別にロード可能である個々のプログラム・ツールを構成し、（ｂ）前記機械コード・セグメントのセットは、前記システム内の前記プロセッサのすべての機械コードと異なるバーチャル・プロセッサの機械コードにより、当該バーチャル・プロセッサ用に書かれており、

前記システム内のデータ・プロセッサの各々は、当該データ・プロセッサによるプロセス

のロード時または実行時にのみ、前記バーチャル・プロセッサの機械コードにより書かれた前記選択された機械コード・セグメントを自らの機械コードに翻訳し、該翻訳した機械コード・セグメントを、当該データ・プロセッサのローカル・メモリにおいて、実行可能プロセスにバインドするように動作することを特徴とするデータ処理システム。

【請求項２】

請求項１に記載のデータ処理システムにおいて、該システムは、プロセスを並列に実行するように構成されたプロセッサのランダム・ネットワーク内のノードとして互いにリンクされた複数のプロセッサを備え、前記バーチャル・プロセッサ用に書かれた前記機械コード・セグメントのセットは、前記ネットワーク内の前記プロセッサのすべてに共通であり、それらのプロセッサにより実質的に同時に使用できることを特徴とするデータ処理システム。

【請求項３】

請求項１または２に記載のデータ処理システムにおいて、該システムは少なくとも２つのプロセッサを備え、それらのプロセッサの機械コードは互いに異なり、前記バーチャル・プロセッサの機械コードとも異なることを特徴とするデータ処理システム。

【発明の詳細な説明】

本発明はデータ処理システムに関し、さらには、データ処理システムを動作させるためのオペレーティング・システムに関する。特に、本発明はコンピュータのオペレーティング・システムに関する。

コンピュータまたはコンピュータ・システムが機能するためには、１つまたは２つ以上のプロセッサ、入出力手段、メモリおよび各種周辺デバイスなどのハードウェア、およびハードウェアをなんらかの方法で実行させるオペレーティング・プログラムと、オペレーション（操作、演算など）を実行するようにオペレーティング・プログラムに指示する上位レベルのプログラムとの両形体になったソフトウェアを備えている必要がある。多くのオペレーティング・システムが設計されているが、これらは利用可能な種々のプロセッサから可能な限り最良のパフォーマンスを引き出すことを試みている。これらのオペレーティング・システムの多くは、１つの特定のプロセッサに対して特有なものとなっている。さらに、これらは、１つまたはわずかの処理言語しかサポートしていない。

本発明は、改良したコンピュータおよびコンピュータ・オペレーティング・システムを実現することを目的としている。

本発明によれば、ネットワーク内にノードとして相互結合された複数のデータ・プロセッサを備えたデータ処理システム(data processing system)が提供され、そのネットワークはプロセスを次のように実行するように構成されている。すなわち、第１ノードは、あるプロセスを実行する命令を受け取ると、その時点でそのプロセスを実行するのに最も適しているのは、そのノード自身であるか、隣接ノードであるかを判断し、その判断に基づいてそのプロセスをそのノード自身のプロセッサに実行させるか、特定の隣接プロセッサに実行させるかを選択するようになっている。

一実施例では、第１ノードは、プロセスの実行に必要な処理スペース（空間）を知らせるメッセージを各隣接ノードに送ることができる。また、第1ノードは、そこに十分なスペースが残っているかどうかを自身でも判断し、各隣接ノードは第１ノードに対する応答として、スペースが使用可能であるかどうかを示す情報を返すように構成されており、第１ノードはその応答を自身の判断と比較し、最もスペースが残っているノードへプロセスを引き渡すか、自身でプロセスを実行するようになっている。

別の実施例では、より高い「パワー・レーティング」(power rating)をもつノードが選択されるようになっている。ここで、パワー・レーティングとは、ノードの１秒あたりの実効オペレーション数をそのノードで実行されるプロセス数で除し、そのノードで利用可能なオフチップ通信速度を表す数を乗じたレーティングのことである。他の方法や方式を用いて、この判断を行うことも可能である。

本発明の第２の形態によれば、１つ以上のデータプロセッサと、それらのプロセッサのすべてに共通かつ常時利用可能である複数のコード・セグメント（機械コードのセグメント

）と、選択された複数のコード・セグメントを、１つ以上のプロセッサにより実行される実行可能プロセスにバインド(bind)する手段とを備えたデータ処理システムが提供される。このデータ処理システムでは、個々のコード・セグメントは、プロセスのロード時または実行時にのみ、実行可能プロセスにバインドされる。

以下、添付図面を参照して本発明の種々実施例について説明するが、これらは単なる例にすぎない。添付図面において、

第１図は、コンピュータ・システムのプログラム編成を示す系統図である。

第２図は、第１図の編成の階層構造を示す図である。

第３図は、コンピュータ・システムのセットアップ時の、いくつかの関連マイクロプロセッサ相互間の関係を示す図である。

第４図は、処理の後続ステージにおける、第３図と同じ関係を示す図である。

第５図は、プロセス・アロケーション（割振り）のためのフローチャートを示す系統図である。

以下では、複数のプロセッサをもつコンピューティング・ネットワーク(computing network)について説明するが、これらのプロセッサは、例えば、同一ウェファまたはチップ上といったように、同一近隣エリアに配置しておくことも、あるいは導線による配線(hard wiring)、光ファイバ、長距離通信回線、あるいは他の手段によってリンク（相互連結）されたネットワーク内に分散配置しておくことも可能である。ここで説明しているシステムは十分に柔軟性があるので、これらの構成や他の構成のいずれでも構わない。システムはプロセッサが１つだけの場合もある。１つまたは複数のプロセッサは、オペレーティング・システムの制御の下でオペレーション（操作や演算）、プログラムおよびプロセスを実行するように構成されている。このオペレーティング・システムによれば、バーチャル（仮想）プロセッサ(virtual processor)として知られ、定義済み独自の言語をもっている想像上のプロセッサ用に書かれたプログラムを作ることが可能である。このバーチャル・プロセッサによれば、バーチャル・プロセッサ用に書かれたプログラム・コードはロード時にだけ各プロセッサのネーティブ・コード（固有の機械コード）に翻訳(translate)されるので、どのタイプのターゲット・プロセッサ(target processor)を使用することも、いろいろなタイプのプロセッサを組み合わせて使用することも可能である。従って、実行可能ファイルは、修正または再コンパイルしなくても、サポートされるどのプロセッサへも転送することができる。本発明のシステムに適したプロセッサの代表例としては、Inmos T800 Transputer、Motorrola 680X0、Intel 80386/80486、TM534C40 Achimedes ARMおよびSUN SPARCがある。他のプロセッサも本発明のシステムに適していることはもちろんである。このオペレーティング・システムは、２つ以上のオペレーションを同時に実行できるような並列処理(parallel processing)向きに改造されている。これは、どの時点でも１つのオペレーションだけが実行されるようにデータとオペレーションをパイプライン化する必要があった従来のコンピューティング・システムとは対照的である。マルチタスキング(multi-tasking)機能とマルチユーザ(multi-user)機能も含まれている。

システムはデータ流れ駆動型(data flow-driven)であるので、基本的には複数のコード・セグメントまたはツールを含んでいるが、これらはロード時にのみ、実行可能プロセスにバインド（結合）される。従って、これらのツールの各々は非常に短くすることができ、ツール自体は、その簡単でほとんどありふれたものにできる。比較のために述べておくが、従来のコンピューティングでは、ライブラリや関数などのルーチンがバインドされるのは、リンキング(linking)として知られている、プロセス内のコンパイル・ステージの直後においてであることに注目されたい。その結果得られるファイルは、多量で巨大で扱いずらい、完全な自己完結(self-contained)型の実行可能イメージであり、ロード時の修正がほとんど不要になっている。このファイルはプロセッサ間で移植不能(not portable)であり、メモリ容量の小さいシステムのような、コンポーネントに変更されたときその変更コンポーネントに適応させることが容易でない。さらに、そのコンポーネント・パーツを他のタスク用に再使用することが不可能であり、事実、変更がまったく不可能になっている。本発明で使用される小さなコンポーネントは、最後に瞬間にだけ寄せ集められてオペ

レーションが作られるので（ジャスト・インタイム・プロセス(just in time process)）、完全に移植可能であることはもちろんであり、プロセスが異なるジョブを実行している場合であっても、２つまたはそれ以上のプロセス間で同時に共有することも可能である。ツールには、特定のアプリケーション用に設計されたものと、特定のアプリケーションを意識しないで設計されたものとがある。遅延バインディング・プロセス(late binding process)によると、プログラム・コンポーネントを実行時にバインド（結び付け）することも可能である。従って、（公称的に）容量の大きい大型アプリケーションは、事実上、メモリ容量の小さいシステムで実行できるので、アプリケーションは、任意の１時点で必要になる特定のコード・セグメント（１つまたは複数）だけをメモリで利用できるように選択することができる。なお、これについては後述する。

システム内のデータは、メッセージの形でコード・コンポーネント間で受け渡しされる。メッセージを送受信できるコード・コンポーネントはプロセス(process)と呼ばれ、ツールの集合と、これらのツールのバインディング・ロジック(binding logic)および処理されるデータとから構成されている。本システムは第１図に概略図で示されているが、同図には、プログラム・コードの中央カーネル(central kernel)が示されている。カーネルは１つの「ブロック」であり、ネットワーク内のすべてのプロセッサによって使用される基本的オペレーティング・システムである。カーネルは８～16キロバイトになっているのが代表的である。カーネルの外側には、概略図で示すように、複数のツール・オブジェクト(tool object)２があり、これはプロセスを構成する集合になっている。プログラム・コードの第３レベルはツール３で示されている。代表的なプロセスは図に矢印で示されている。このように、P1と名づけたプロセスは、カーネル・コード１と２つのツール・オブジェクトM1,M2から構成されている。これらのツール・オブジェクトの各々は、長さが数百バイトにすぎないが、ツール３の１つまたは２つ以上を利用することができる。この例では、ツール・オブジェクトM1はツールT1,T2およびT3を利用し、ツール・オブジェクトはM2はT3（再利用），T4,T5およびT6を利用している。従って、図に示すように、相対的に複雑なプロセスまたはプログラムが、相対的に小さい、複数のセグメントを使用することによって実行することができ、これらのセグメントは、その必要とする時にだけ寄せ集められる。各プロセスを別々のプロセッサ上に置いておくと、２つ以上のプロセスを同時に実行することが可能である。プロセスの実際の配置は以下でその概要を説明している自動的方法(automatic method)によって行われるが、この方法はユーザからも、実際にシステム全体からも透過(transparent)になっている。従って、最適な負荷バランス化(load balancing)と通信効率が保証される。その結果として、通信し合う多数のプロセスを、自動的に並列に実行させることができる。

システムの３つの基本的なコンポーネント・オブジェクトが第２図に示されている。これらは、データ・オブジェクト、ツール・オブジェクト（ツール）およびプロセス・オブジェクトからなっている。プロセス・オブジェクト(process object)は、種々のツール・オブジェクトを呼び出し(call)するハーネス（harness－引き具）の働きをする。プロセスは、基本的には、アクティビティ(activity)のロジック・フロー（論理の流れ）を指示して、ツール・オブジェクトがデータに対して実行するように指示するものである。従って、プロセス・オブジェクトは、第２図に示す階層構造の最上位に置かれている。ツール・オブジェクト(tool object)は従来の関数呼び出し(functional call)またはサブルーチンに対応するものである。これは参照(reference)によってのみ使用できるので、プロセス・オブジェクトなどの、他のなんらかによってアクティベート（activate－起動、実行開始、活性化）させる必要がある。これは、Ｃ言語の関数がコードのメイン・ボディ（main body）から呼び出されるのと似ているが、主な違いは、本発明のシステムでは、各コード・セグメント、つまり、ツールが完全に独立していることである。最後に、データ・オブジェクト(data object)はデータ・ファイルに対応するものである。データ・ファイルにはそのファイルに関する情報と、ファイルの処理方法に関する情報を入れておくことができる。この情報には例えば、データ構造を操作するときに使用されるツールを指し示す情報と、そのファイルに関係するローカル（局所）データを含めることが可能である。プ

ロセスは、ASCIIデータ・ストリームのような、公開的に(public)定義された入出力をもつことができる。従って、プロセスは、その作成者だけでなく、第三者のプロセスによっても使用することができる。公開的になっていないプロセスは、私的(private)の入出力構造をもつことになるので、入出力インタフェースの細部が公開的に利用可能にされない限り、特定のプログラムの作成者だけが自由に使用できるにすぎない。１つまたは２つ以上のプロセスを結合して、完成された作業ジョブ(complete job of work)であるアプリケーションを作ることができる。なお、アプリケーションは２つまたはそれ以上のプロセッサ上で実行させることが可能であることを注意されたい。

上述したように、アプリケーションは多重タスキング処理および／または並列処理が可能である。プロセスは他のプロセスは、子プロセス(child process)をアクティベートすることができ、アクティベートされるこれらプロセス自体も子（孫）プロセス(grandchild process)をアクティベートすることができる。以下、同様である。これらのプロセスは利用可能なプロセッサ・ネットワーク全体にわたって動的に分散化されるが、アプリケーションまたはプログラマには実際の分散化は知らされない。プロセス間の通信は、親子階層(parent-child hierarchy)に基づいて行われる。「マスタ」となるノードまたはプロセッサはそれ自体では存在しないので、かなり多くのプロセッサからなる、任意のトポロジ構造(topology)の使用が可能である。第５図は、ネットワーク内の種々のプロセッサ間にプロセスを分散するときの、１つの可能なアルゴリズムを示している。

親プロセス(parent process)が子プロセスと共同で処理することを望んでいるとき、メッセージがまず最初に第１プロセッサ、特に、そのプロセッサ上のカーネル・プロセスに渡される。このプロセスは、プロセッサとの間で受け渡しされるメッセージを処理し、プロセス・コントロール(process control)／メッセージ・スイッチャ(message switcher)の働きをする。"A"と名づけた、第１プロセッサのカーネルは、次に、そのプロセッサの「パワー・レーティング」(power rating)を計算する。パワー・レーティングとは、基本的には、その特定プロセスを実行するためのそのプロセッサ内の能力またはスペースのバロメータである。パワー・レーティングは、１秒当りのプロセッサの実効オペレーション数をそのプロセッサ上で実行されるプロセス数で除し、そのプロセッサで利用できるオフチップ通信速度を表す数を乗じたそのプロセッサのレーティング（定格）と定義することができる。パワー・レーティングは他の方法で求めることも可能である。例えば、特定のプロセスを実行するためのメッセージは、そのプロセスで必要とするバイト数が付随している。従って、パワー・レーティングには、そのバイト数がそのプロセッサのローカル・メモリに残っているかどうかの判断を含めることが可能である。このメモリは、トランスピュータ(transputer)のようなプロセッサでは、オンチップ・メモリ(on-chip memory)になっているが、オフチップ(off-chip)である場合もある。次に、ローカル・カーネルは、各自のパワー・レーティングを計算するように、ネットワーク内のすべての隣接プロセッサのカーネルに指示する（ここで、隣接とは、トポロジ的には隣り合っているが、物理的には、各プロセッサが遠く離れた位置にある場合もあることを意味することを覚えていてほしい）。次に、隣接プロセッサの各々は、それぞれのパワー・レーティングを示すメッセージを親プロセッサＡに送り返す。次に、Ａのメール・ガーディアン（mail guardian－メール管理プログラム）はパワー・レーティングをすべて比較し、Ａ自身のパワー・レーティングがその隣接プロセッサのそれより大または等しいかを判断する。もし、そうであれば、Ａのメール・ガーディアンは自身がプロセスを引き受けることを決定する。これは、子プロセスをインストール（導入）するように自身の動的バインダ（dynamic binder－動的結合プログラム）とトランスレータ（翻訳プログラム）（以下参照）に指示することによって行われ、命令側プロセッサにメッセージを送って、受信側プロセッサ上の固有のメールボックス・アドレス(unique mail box address)を知らせる。プロセッサＡのパワー・レーティングがその隣接プロセッサのそれより大でも、等しくもなければ、最大のパワー・レーティングをもつプロセッサが選択されて、そのプロセスを引き受けることになる。全てのパワー・レーティングが等しければ、プロセッサＡがそのプロセスを引き受けることになる。別の方法として、隣接プロセッサを任意に選択することも可能である。そ

のプロセスが別のプロセッサにアロケート（割振り）されていれば、プロセッサＡは、"take X Bytes,Process name,parent mail box no."（Xバイトが必要、プロセス名、親メールボックス番号）を伝えるメッセージをそのプロセッサへ送付する。
あるプロセッサが子プロセスを引き受けていることが分かったときには、その動的プロセス・アロケーション(process allocation dynamics)を中止して、そのプロセスを実行させることが可能である。しかし、同じアロケーション・シーケンスを新しいプロセッサから繰り返すのが、もっと一般的である。つまり、第５図に示すフローチャートでは、新たに選択されたプロセッサ、つまり、プロセッサＢがプロセッサＡとなり、サイクルは、パワー、レーティングの計算を隣接プロセッサに要求するステップから再び開始される。従って、現在のアクティビティが最小であり、従って、プロセスを自動的に実行するための容量が最大であるローカル・プロセッサを見つける探索が、起点となるマスタ親(originating master parent)であるセンタ・ポイントからあふれ出ていく(flood out)傾向がある。これにより、プロセッサのネットワーク間の負荷バランス化と密結合(tightly bound)プロセス間のローカル通信が十分に保証されることになる。さらに、上記から理解されるように、マスタ・ノードが必要でないので、どのネットワークにおいても、どのプロセッサでも、任意の時点で実行されるプロセスは１つだけであるのが通常であり、タイプが同一であれば、どの隣接プロセッサでも同じである。別の方法として、かなりの実施例において、ユーザは、それを望むならば、どのプロセッサにまたはどのタイプのプロセッサにプロセスを実行させるかを選択することができる。例えば、ネットワークに２タイプのプロセッサがある場合に、例えば、メモリ容量によっては、これらのタイプの一方が他方よりも特定のプロセスを実行するのに適している場合がある。この場合、ユーザは、このプロセスを特定タイプのプロセッサに実行させることを指定することができる。また、プロセスは、従ってプログラマは、そのプロセスがネットワーク内のどこかに置かれているかを正確に知っている必要がないということも理解されよう。メッセージはデータまたはコードいずれか一方をプロセッサ間で伝達できるので、メッセージを実行可能コードの形体にすることができる。
別の実施例として、プロセッサは、その「パワー・レーティング」に関する情報をプロセッサ相互間で継続的に受け渡すことができるようにしてもよい。これは、通常の通信過程では、受け渡しされるメッセージに要れて組込み情報(embedded information)として受け渡すことができるが、メッセージがプロセッサ間で受け渡しされないような期間には、一定のパワー・レーティング情報の交換だけ行うので、未使用の通信時間を効率よく使用することができる。この場合、各プロセッサは、例えば、ルックアップ・テーブル(look-up table)をもつことができる。このテーブルは、そのプロセッサのすぐ隣のプロセッサのステータス（状況）と自身の当面のステータス(immediate status)を示している。パワー・レーティング情報は、例えば、プロセッサ間で受け渡しされる通常の各メッセージの最後または先頭の１バイトまたは数バイトの形で受け渡すことができる。この実施例では、プロセスを実行する要求を受け取ったとき、プロセッサは、そのプロセスを自身で引き受けるかどうか、あるいはそれを隣接プロセッサへ渡すかどうかを即時に判断できるので、それがどのプロセッサであるかを知ることになる。要求を隣接プロセッサへ渡すと判断したときは、その要求が送り出され、受信側プロセッサは、そのプロセスを自身で実行するのに最も適していると判断する時点まで、判断プロセスを再び開始する。
プロセスが取り得るステート（状態）には次の５つの区分がある。すなわち、
1. 処理活動中(actively processing)
2. 入力待ち(waiting for input)
3. 取り消し(withdrawing)
4. 非アクティブ状態(inaotive)
5. 休眠状態(asleep)
ステート１と２は解説の必要がないので説明を省略する。ステート３は、プロセスが取り消しコマンドを受け取ると発生する。このステートにあるときは、プロセスは、非アクティブ状態（非活動状態）になるために必要なアクティビティを実行する。そのようなアク

ティビティとしては、例えば、緊急メールの処理、もしいれば、その子プロセッサすべてへの取り消しメッセージの送付、一般的には、非アクティブ状態になるために都合のよいステートの準備などがある。そのプロセスがアプリケーションの最上位レベルを構成していれば、その正確な現ステートをメッセージとしてディスク・ファイルにセーブしておくのが普通である。ステート４は、プロセスが事実上終了していることを示している。従来のシステムでは、これは、メモリ内にプロセスが存在しなくなり、そのメモリ・スペースが次の使用に備えてクリアされることを意味している。しかるに、本発明によるシステムでは、プロセスは非アクティブ状態とマークされるが、そのメモリ・スペースが実際に必要でないか、あるいはその必要がなくなるときまで、プロセスは除去されない。従って、同じローカル・プロセッサの別のプロセスはステート４にある非アクティブ・プロセスを参照すれば、そのプロセスは即時にステート２に移ることができる。ステート５は、例えば、プロセスがディスクまたは他の記憶媒体上のような永久記憶(Permanent store)に入っているときのプロセスの状態である。プロセスがステート４の非アクティブ状態にあり、そのメモリ空間が別のプロセス用に必要であるときは、そのプロセスはステート４からステート５へ移る。つまり、プロセスはディスク上にストアされ、ローカル・プロセッサのメモリから除去される。ステート５にあるプロセスが特定のローカル・プロセッサで必要になったときには、そのプロセスは永久記憶からロードされる。この場合、プロセッサはメールを利用することができる。このメールは３つのソースからのものが使用できる。すなわち、ディスク上のプロセスのステートから、あるいはプロセス自身のヘッダ、つまり、プロセスの一部を構成する情報から、あるいはプロセスを最初の場所にロードさせる親プロセスからのメールである。この結果、プロセスはステート４からステート２へ移って、それからステート１へ移ることができる。

「ディスク」という用語が上で用いられているが、これは、物理的記憶媒体上に格納されたデータ・オブジェクト（データ対象物）「ファイル」の集まりを収容しているデバイスのことである。

上述したように、すべてのプロセスには、メールボック機能(mailbox facilities)が自動的に与えられるので、プロセスはメールを読み取って、メール・アドレスが分かっている他のプロセスへメールを送付することができる。このプロセスとしては、例えば、子プロセス、プロセスの親、例えば、ディスク・ドライブやディスプレイのような、名前付き資源(named resource)、記憶アドレス全体が分かっている非アクティブ状態のプロセス、メールボックス・アドレスが分かっている他のプロセスがある。アクティブ・プロセスがメールを非アクティブ・プロセスへ送付した場合に、そのメールで非アクティブ・プロセスを子プロセスとして実行させることができるが、目的プロセス(target process)が目覚めるまでそのメールをストアしておくこともできる。

アクティブ状態の２つのプロセス間で受け渡しされるメッセージは、次のような構造にすることが可能である。

ビット数	説明
32	メッセージ長（バイト数）
32	メッセージ・タイプ・マスク－メッセージのタイプ。例：コード、デバッグ情報、エラー・データ
32	メッセージ・データ・オフセット－メッセージの先頭を指しているので、実際にDTM（下記参照）を固定長にしないことが許される
32	次デスティネーション・ポインタ(NDP) －応答が送られるべき宛先のアドレスを示している。
64	発信側メールボックス(Originator Mailbox)
64	デスティネーション・ターゲット・メールボックス(Destination Target Mailbox - DTM)－プロセスIDのリスト
64	２番目のDTM
..	（２番目以降のDTM)
..	
..	
(xxx)	メッセージ・データ

メッセージ長は、メッセージ全体の長さのバイト数である。

次デスティネーション・ポインタが指しているデスティネーション・ターゲット・メールボックスが"0"であれば、応答の必要がないか、あるいは応答が予期されない。メッセージのデスティネーション・メールボックスの配列(array)が前方(onward)にあっても、そのことは、特定のメッセージが転送されることを意味しない。有効なDTMが配列にあるときは、メッセージに対する応答がそのプロセスへ転送されることを意味する。メッセージが応答を必要としない単純なケースでは、DTM配列には、３つの値が入れられる。それは送信側プロセス・メールボックス、ターゲット・プロセス・メールボックス、０である。メッセージをターゲットID 0へ送信しようとする場合は、そのメッセージはシステム・メール・マネージャ(system mail manager)へ発送される。メッセージが送信側に対する応答を必要とする場合は、そのメッセージが有するDTM配列は４つの値からなっている。それは送信側メールボックス、デスティネーション・メールボックス、送信側メール・ボッ

クス、０である。
データ・ストリームを処理するプロセスのパイプラインは、（処理パイプの数）＋送信側メールボックスとパイプ内のプロセスの１つである最終的デスティネーション（宛先）とを含む２エレメント、をもつDTM配列を受信する。
「フォーク」パイプ方式('Forked'pipe scheme)が可能であるが、この方式によると、一方のプロセスがアクティブ状態でフォーク(fork)を作成し、他方のプロセスがアクティブ状態でそのフォークとあとで結合できることが必要である。単純なリニア・パイプ(linear pipe)はUNIXフィルタ（これは同業者にとって公知である）と同じようなやり方で働くが、並列に実行可能で、ネットワーク内に接続しておくことができる、再使用可能なツール・オブジェクトだけを必要としている。
フォークおよび結合メッセージング(forked and jointed messaging)は、単純な入出力フィルタ・ツールと共にシェル(shell)を使用してプログラミングを行うと、自動的に処理される。
デスティネーション・プロセスのメモリ空間に置かれるとメッセージ構造体は、さらにメッセージ長の前に付けたされた２つの追加エレメントを有する。

ビット数	説明
32	順方向リンク・ポインタ
32	逆方向リンク・ポインタ
32	メッセージ長（バイト数）
32	メッセージ・タイプ・マスク

以下前記と同じ
順方向リンクと逆方向リンクは、プロセスの着信メールボックス(incoming mailbox)に対してリンクされたメッセージ・リストを構成する他のメッセージがあれば、それらを指している。
メールはプロセッサによって次の２ステージで読み取られる。つまり、メッセージ用に動的に割り振られたバッファ内から、メッセージのサイズが最初に読み取られ、次にメッセージの残り部分が読み取られる。このメカニズムによると、障害のあるノードや何らかの理由で、メッセージを受信できないカーネルを迂回してメッセージを宛先を変えて転送することができる。
メッセージは、メール・グループ・ディストリビュータ(mail group distributor)によって、２つ以上のデスティネーションへ分配することが可能である。あるプロセスがあるプロセス・グループに属していることを知っていれば、そのプロセスは、そのグループの一員であることを宣言するメール・グループ・ディストリビュータへメッセージを送ることができる。別の方法として、プロセスは、そのメンバ・プロセスすべてを定義しているメール・グループ・ディストリビュータへメッセージを送ることも可能である。通知を受けたとき、メール・グループ・ディストリビュータは、そのグループのために使用されるメールボックスIDのすべてのメンバ・プロセスに知らせる。グループのすべてのメンバに分配されるメールはグループ・メールボックスへ送られ、コピーが送信元を除くすべてのメンバへ送られる。１つのプロセスは２つ以上のメール・グループに属することができるので、サブグループ(sub-group)化が可能である。私用メッセージ(private message)は、他のすべてのプロセスに知られることなく、グループ内のプロセス間で送信することも可能である。
次のようなタイプのツールが使用可能である。すなわち、オペレーティング・システムのコア（core－中心）を形成するパーマネント・ツール（permanent tools－常駐ツール）である。このツールはブートアップ・シーケンス(boot-up sequence)によってアクティベートされ、そのあとで非アクティベート(disable)することはできない。すべてのプロセッサには、利用可能な各パーマネント・ツールのコピーが常に用意されている。

セミパーマネント・ツール(semi-permanent tools)は、すべてのプロセッサのためにブート・プロセスによってアクティベートされる。ユーザは、どのセミパーマネント・ツールをアクティベートできるかを選択することができる。いったんアクティベートされたツールは非アクティベートすることができない。
ライブラリ・ツールは、必要時に、名前付きのツール・ライブラリ(named library of tools)から使用される。いったんアクティベートされると、これらのツールは、メモリの制約上、ツールを非アクティベートする必要が起こるまで、ツールを参照するアプリケーションが実行されているプロセッサのメモリにキャッシュ（貯蔵）されたままになっている。
アプリケーション・ツール(application tools)はバーチャル・ツール(virtual tool)と非バーチャル・ツール(non-virtual tool)の２種類がある。バーチャル・ツールは、アプリケーションが実行されているときアクティブである必要はないが、プロセスがそのツールを参照しようとすると、必要に応じてアクティベートされる。実行中でないときは、バーチャル・ツールは、メモリ・スペースが別の状態で必要になる場合を除き、キャッシュされたままになっている。従って、利用可能なメモリに収まらないような、大きなアプリケーションのツールを自動的に「オーバレイ」(overlay)することが利用可能となっている。非バーチャル・ツールは、アプリケーションと一緒にロードされ、その実行前にアクティブな箇所(active place)に置かれている。従って、これらのツールは常に利用可能であり、実行時にはメモリ内にある。
ネットワーク内の各プロセッサまたはプロセッサ・グループは、動的バインダ・トランスレータ(dynamic binder and translator-"DBAT")と呼ばれるパーマネント・ツールを含んでいる。DBATはプロセッサのネーティブ・コード（固有の機械コード）で書かれているので、サポートされるプロセッサが異なるごとに、DBATの固有のバージョンが必要である。DBATはバーチャル処理コードをプロセッサの特定コードのネーティブ・コードに変換する機能を備えている。各DBATは、ツール・リストを使用する。このリストはパーマネントおよび外部ツール・リスト(permanent and external tool list-PETL)であり、すべてのパーマネント・ツールとセミパーマネント・ツールに関する情報と（このリストはどのノードの場合も同じである）、ノードによって現在参照されているか、あるいは以前参照されたが、まだアクティブであるか、あるいは非アクティブであるが、まだ使用可能であるライブラリのような、すべての外部ツールとに関する情報を個々のノード別に収めている。
メッセージを実行するコマンドを受信すると、オブジェクトはプロセスのローカル・メモリ・エリアに読み込まれる。この場合も、プロセッサという用語は、２つ以上のプロセッサがネットワーク内で１つのDBATと結合されていることを意味する。オブジェクトによりパスが作られる。そのパスで、DBATはツールをPETLに追加する。
外部ツールのPETLへの追加は、次のように行われる。そのツールがすでにPETLの中にあれば、これは使用可能として受け付けられる。しかし、そのツールがPETLの中になければ、これは読み込まれ、使用可能ツール・リストの中にリンク（連係）される。この新たに挿入された、このツールが他のツール参照を含んでいれば、DBATはその参照を同じように繰り返し処理する。DBATはすべての外部ツールが使用可能になるまで続ける。
DBATは、プロセスと新たにインストール（組込み）されたツールのVPコードを「翻訳(translate)」してターゲット・プロセッサへ渡すが、その際に、コード内のツールを指しているポインタ情報の挿入と訂正を行う。未解決なバーチャル・ツール参照はTRAPに変換され、ツールの完全パス名(full pathname)を指しているパラメータと一緒にDBAT内部のバーチャル・ツール・ハンドラ(virtual tool handler)に渡される。完了すると、DBATは制御をプロセスに引き渡す。
各外部ツールが着信プロセス(incoming process)によって参照されると、その使用状況フラグ(usage flag)がインクリメント（増分）されるので、ツールは、そのツールを現在使用しているプロセスの数を表す値をもつことになる。各プロセスが非アクティベートされると、使用状況フラグはデクリメント（減分）される。プロセスがツールを使用していないときは、そのツールは使用状況値がゼロになるので、PETLから除去することで非アクテ

ィベート化して、そのメモリを解放することができる。これが行われるのは、メモリ・空間・ギャベージコレクション(memory space garbage collection)が行われるときだけであるので、システムはツールをキャッシュ（貯蔵）することができる。セミパーマネント・ツールとパーマネント・ツールは、初期使用状況値を"1"にしてブートアップ時にDBATによってインストールされるので、これらのツールが非アクティベートされることはないと保証される。バーチャル・ツールは実行中にプロセスからコール（呼出）される。バーチャル・ツールは、ツールのパスと名前を指すポインタと、ツールの通常パラメータとを使用して参照される。

DBAT内のバーチャル・ツール・ハンドラ（操作子）はPETLをチェックすることにより、ツールがすでに使用可能であるかどうかを確かめる。そうであれば、ツールは通常の外部ツールの場合と同じように使用される。使用可能なツールが存在しなければ、DBATはツールをアクティベートしてから（このプロセスは、プロセス・スタートアップ時にアクティベートされた外部ツールについての場合と同じである）、ツール参照を引き渡す。これは、ギャベージコレクションでも、着信ツールに見合う十分なメモリが見つからなかったとき、メモリ制約が原因で障害を発生する場合を除き、完全に透過(transparent)になっている。このメモリを保証するのにバーチャル・メモリ（仮想メモリ）手法を用いることができる。

第３図と第４図は、オペレーション状態にあるシステムを概念的に示す図である。中央の「バーチャル」プロセッサVPに概念的に接続された３つのプロセッサ（またはプロセッサ・アレイ）４が示されている。これらの図では、各プロセッサは、オンチップまたはオフチップので総メモリをもっているものと想定している。ブートアップ時に、各プロセッサは、DBATおよび他のすべてのパーマネント・ツール($P_1 ... P_n$)ならびにセミパーマネント・ツール($SP_1 ... SP_m$)を取り入れる。この状態を示したのが第３図である。すべてのプロセッサには同一グループのパーマネント・ツールとセミパーネント・ツールが含まれている。

第４図はその後の状態を示している。あるプロセッサは、ライブラリ・ツールL_1,L_2,L_3およびL_4を使用するプロセスを実行するように指示されている。従って、プロセッサはこれらライブラリ・ツールを読み込んでいる。他方、このプロセッサは特定のプロセスを実行する必要に応じて、種々のバーチャルツールまたは他のツールを読み込むことも可能である。別のプロセッサはツールL_5,L_6,L_7およびL_8を読み込んでいる。この最後のプロセッサは現在どのプロセスも実行していない。２つまたはそれ以上のプロセッサは、ライブラリのコピーをローカル・メモリに読み込んでおけば、同じライブラリ機能を使用できることはもちろんである。プロセスが終了したとき、ライブラリ・ツールは、再び必要になるまで、あるいはギャベージコレクションが行われるまで、それぞれのローカル・メモリにキャッシュされたままになっている。上述したように、いくつかのプロセスを任意のプロセッサ・ネットワーク上で同時に稼動できることは勿論である。

DBATは、VPコードをネーティブ・プロセッサ・コードへ変換することを扱うほかに、いくつかの実施例では、ネーティブ・プロセッサ・コーディングを認めることも可能である。ネーティブ・コーディングを使用すると、コードの移植性が妨げられるが、ハードウェア固有のツール（一般的には、ドライバ）のストラテジ（strategy－戦略）として受け入れることが可能になる。

アプリケーションの拘束を受けないコピーを防ぐようにするために、アプリケーションは、ネーティブ・コード・オブジェクト能力をそのまま使用できるようにし、オリジナルVPコーディングをハウスに残しておけば、特定のプロセッサをベースとしたいろいろなプロダクトを作ることができる。しかし、この特徴を使用すると、ミックスされたプロセッサ・タイプを備えた並列システム上でアプリケーションを実行させることができなくなる。

外部参照ツールが使用可能に用意されていないと、DBATのアクションの下でエラーが表示されるので、欠落ツール(missing tool)の詳細を知ることができる。そのとき、ユーザは、その欠落ツールを別のツールで一時的に置き換えるか、あるいは他のものに永続的に代えるかどうかを決めることができる。ツール紛失は、アクセス経路(route)または通路(pa

ss)が脱落しているか、壊れていることから起こるが、例えば、フロッピ・ディスクなどの交換可能媒体(removable medium)が目下アクセス不能である場合にも起こる。オブジェクトはネットワークから出し入れすることができるが、これを行うと、アプリケーション全体、その従属プロセス(dependent process)のすべて、などが除去されてしまうことがよくある。

上述したように、バーチャル・プロセッサは、移植可能な「実行可能」コードを構築するために使用される想像上のプロセッサである。上述したように、VPと互換性のある(VP compatible)アセンブラおよびコンパイラの出力は、DBATを使用してバインド（結び付け）を極端に遅らせて行う方式(extremely late binding system)を採用しているので、リンキング（連係処理）を行う必要がない。

VPコード中の「実行可能」コードはDBATによって処理されて、ロード時にそのホスト・ターゲート・プロセッサへ渡されてから、制御がコードに渡される。VPは16×32ビット・レジスタ・セットを備え、32ビット・リニア・アドレス空間(linear address space)が想定されているが、16には限定されない。レジスタR0からR15までは完全に汎用レジスタ(general purpose)である。IP、セグメント・ポインタ、フラグなどのレジスタは想定レジスタ(assumed register)である。つまり、これらレジスタは、必要時にターゲット・プロセッサによってインプリメント（履行）される。VPコードはオペレーティングの原理に関係なく、最新のターゲット・プロセッサへの「変換」が非常に単純化して設計されている。32ビット・プロセッサ以外の、他のワード・サイズのプロセッサの扱いも簡単になっている。VPレジスタ・セットの使用は最終のターゲット実行可能コードでは除かれない。レジスタ・セットはターゲット・レジスタ・セットといっしょにメモリにインプリメントされることがしばしば行われ（ゴースト・レジスタ(ghost register)と呼ばれる）、ターゲット・レジスタ・セットはVPオペレーション・セットをメモリ・レジスタ上にインプリメントする必要が行ったとき使用される。このようにVPコーディングをネーティブ・プロセッサによって疑似翻訳させると、ある種のプロセッサでは若干速度が低下するという欠点があるが、全体的には得られる利点は非常に多い。キャッシュ・メモリ(cached memory)または高速メモリを備えたプロセッサ（トランスピュータなど）は、特に、ネーティブ・レジスタ・セットがトランスピュータ内にあるときに非常に小さい場合には、パフォーマンス（処理能力、性能）が向上することを実際に示すことができる。この方式を用いると、外出タスク(outgoing task)のレジスタ・セットの位置をセーブするメモリ書込みと着信タスクのレジスタ位置をリストア（復元）するメモリ読取りとを除き、ネーティブ・プロセッサのステート（状態）を無視できるので、タスク切替え時間(task switching time)が非常に高速化される。トランスピュータ・アーキテクチャによれば、このことさえも不要になる。

ターゲット・プロセッサ上で最終的に実行されるコードの効率は、それぞれVPコンパイラとDBATによって行われる、コンパイルとアセンブリの２ステージに左右される。コンパイラがVP出力を最も効率的な形体に縮減する能力は邪魔されないが、明らかに、ピープホール最適化(peephole optimisation)はほとんど、あるいはまったく行うことができない。DBATは、ある種のVPコード・シーケンスによって起こることがある、冗長的なネーティブ・レジスタ／ゴースト・レジスタの移動を取り除くことにより、主として生成される最終的コードについてピープホール最適化を行う。

トランスピュータでは、VPゴースト・レジスタを使用すると、従来のレジスタを使用した場合と同等の、あるいは向上したパフォーマンスが得られる。ゴースト・レジスタは、トランスピュータでは、オンチップRAMに実装され、実行されるすべてのプロセスのための「データ・キャッシュ」を効率よく生成する。プロセスは、Inmos T800トランスピュータでは、最高48個までがオンチップ・メモリ・ゴースト・レジスタを使用することができる。48個を越えるプロセスには、オフチップ・メモリにスペースが必要である。

上述してきたシステムには、ウィンドウ、マウスなどを使用してオペレーションを行うためのグラフィカル・ユーザ・インタフェースを含めることが可能である。ウィンドウはディスプレイ手段に対して表示されてマウスやキーボード、その他の入力手段を使用して移

動させて操作を加えることも、その内容に操作を加えることも可能である。

【図１】

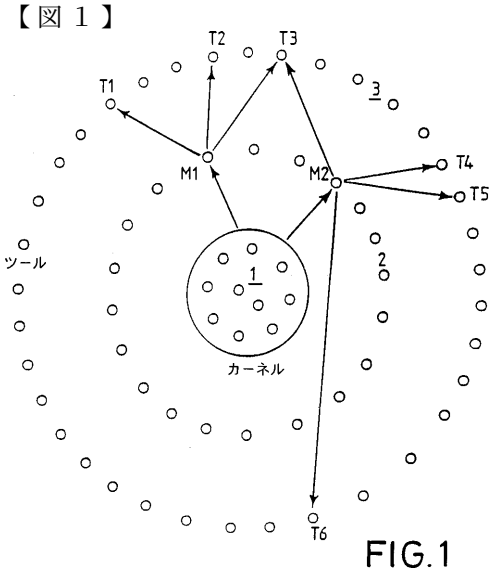

FIG.1

【図２】

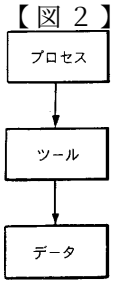

FIG.2.

【図３】

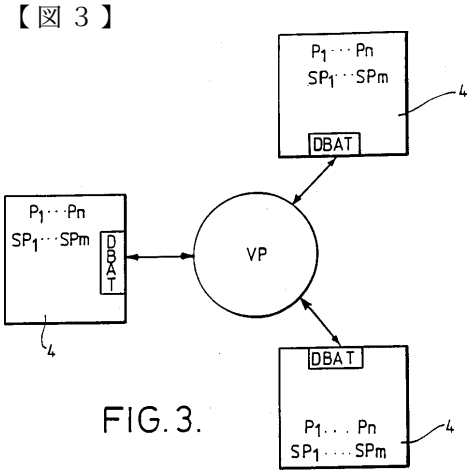

FIG.3.

【図４】

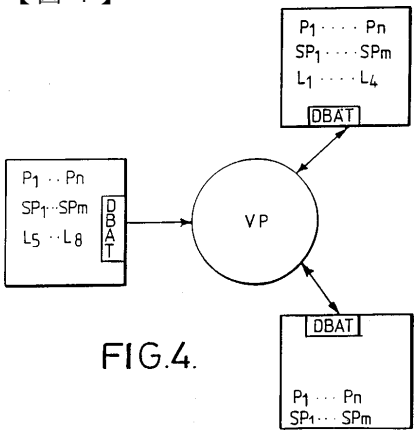

FIG.4.

【図５】

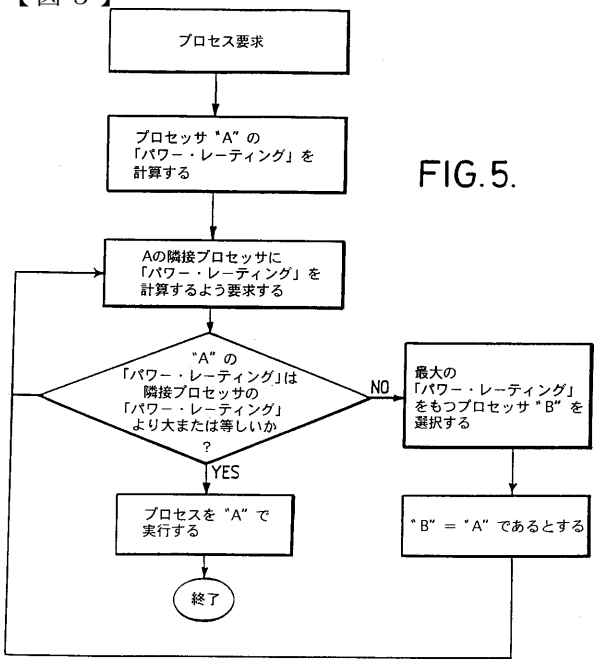
プロセス要求
プロセッサ "A" の「パワー・レーティング」を計算する
Aの隣接プロセッサに「パワー・レーティング」を計算するよう要求する
"A" の「パワー・レーティング」は隣接プロセッサの「パワー・レーティング」より大または等しいか ?
NO
YES
最大の「パワー・レーティング」をもつプロセッサ "B" を選択する
プロセスを "A" で実行する
"B" = "A" であるとする
終了

FIG.5.

フロントページの続き

合議体
審判長　川名　幹夫
審判官　橋本　正弘
審判官　松浦　功

(56)参考文献　特開平４－２８９９５５（ＪＰ，Ａ）
特開平３－２２９３２７（ＪＰ，Ａ）

(58)調査した分野(Int.Cl.[7]，ＤＢ名)
G06F9/46，G06F9/44