オペレーティング・システム　51－2
（1991. 7. 18）

ビルディング・ブロック・システムのための共有ソフトウェアバス～礎～

並木美太郎、高橋延匡

東京農工大　工学部　電子情報工学科

本報告では、OS をモジュール化し、拡張性を向上するための OS アーキテクチャについて述べる。個人用の計算機環境を手軽に構築するためには、個人の望む機能を持った OS 核を手軽に生成できる必要がある。このために OS 核の部品化を行うことを試みた。OS の各機能をモジュール化するために、ソフトウェアバスと称する仮想的な共有バスを用いてモジュールのインタフェースを統一した。また、ソフトウェアバスを実行し、ソフトウェアバスと OS 、ソフトウェアバス間の通信を行うハイパ OS 「礎」を開発した。

現在、「礎」は MC68030 VME バスシステム上で稼動し、OS/omicron とソフトウェアバスの移植作業が行われている。

Software Bus Architecture and Hyper OS "ISHIZUE" for Building Block Systems

Mitarou Namiki and Nobumasa Takahashi

Department of Computer Science, Faculty of Technology
Tokyo University of Agriculture and Technology

2-24-16 Naka-cho, Koganei-shi, Tokyo 184, Japan

This paper presents an OS architecture based on a module approach for extensibility. For personal computing environment, an OS kernel needs to be constructed easily and adapted to its use. This paper proposes a "Software Bus" architecture on which OS facility is constructed with independent modules that have unified interface. In order to execute "Software Bus", we designed Hyper OS "ISHIZUE" for module communication.

Hyper OS "ISHIZUE" has been implemented on a VME MC68030 CPU card. OS/omicron and "Software Bus" are intended to run on "ISHIZUE".

1. はじめに

汎用目的の OS は常に肥大化の歴史をたどってきた。汎用大型機の OS は無論のこと、UNIX、MS-DOS などのワークステーション、パーソナルコンピュータの OSも分散処理や GUI (Graphical User Interface)の機能拡張につれて、同様の傾向をたどっている。

OS 核の肥大化を抑え、システムの拡張性・信頼性向上のための方法として、階層化アプローチは古くから提唱されてきた[1]。より抽象化・仮想化の低い層から、仮想化の進んだ機能を実現する「積み重ね(Building Block)」のアプローチは、OS 構築の基本原理となっている。各層の仮想化の要件は、システム構築の目的とハードウェアの制約条件によって決定されてきた。特に、近年では、Mach[2]に代表されるよう、分散処理的見地からシステムソフトウェアの核の再編成が行われている。国内でも、プロセスの集合体としてシステムを構築するアプローチ[3]や、オブジェクトオリエンテッドの見地から OS を構築する研究[4]が行われている。

我々の研究グループでは、手書きユーザインタフェースの研究、卓上電子出版の研究、日本語情報処理の研究のための OS として OS/omicron を開発してきた[5]。これらの研究を行うために、その研究向けの機能を OS に追加したい、という要求がある。このためには、拡張を容易に行える機能が OS に必要となる。

本報告では、ソフトウェアバスとハイパ OSによる、OS の機能拡張を容易にするための OS アーキテクチャを提案する。さらにソフトウェアバス上のモジュールの実行とモジュール間の IPC を管理するための、ハイパ OS 「礎」の設計と実現について述べる。

2. OS の拡張性を向上するためのソフトウェアバスとハイパ OS アプローチ

2. 1 目的

パーソナルコンピュータやワークステーションは個人の計算機環境として必要不可欠なものになっている。パーソナルユースを指向した場合、計算機を個人の望む形態に変更できることが重要となる。例えば、新しいデバイスを接続したり、必要な機能をシステムに追加できることは、パーソナルコンピュータの利点の一つとなっている。ということは、一歩進めて、個人の望む計算機環境が、部品箱の中の部品を組合せるように構築できればすこぶる便利であろう。

本研究の目的は、単なる機能拡張だけでなく希望する OS 核を容易に生成する方式を得るために、OS のモジュラリティを向上させる OS アーキテクチャを開発することにある。無論、モジュラリティといった場合は、ワークステーション一台という単位から一つのルーチンという大きさまで考えられるが、本報告では、数個から数十個のサブルーチンをモジュールの基本単位とし、それらを組み合せてより大きな機能を持つモジュールを実現することにより、システムを構築することを意図している。

近年では、VME バスや Future バスなどの標準規格のバスとその CPU カードによる共有メモリ型マルチプロセッサシステムを入手できるようになった。CPU チップも RAM・ROM 内蔵 8bit CPU が存在する。10年先には、現在の CPU カードを1チップにできるのではなかろうか。その時に、現在の標準バスに CPU カードを挿入してシステムを組むように、バックプレーンに数MBの RAM を持った CPU チップ(または CPU モジュール)を複数個挿入してユーザインタフェースのために応答性のすぐれたシステムを構築したい。このようなシステムのために、ソフトウェアのモジュールの粒度により、関数、プロセス、プロセッサ、マシンという実行形態の選択が可能な機構を開発する。

OS の拡張性は、OS の設計、特にモジュール設計に起因するところが大きい。モジュール設計に対する一つの指針は、仮想化の水準に応じて機能を分割する階層的なアプローチである。この方法は、マルチプログラミングシステムの構築に成果を与えてきた。しかし、拡張性という観点からながめた場合、既存の仮想化の方針と異なる性質を持ったモジュールを追加することは、当然困難なこととなる。さらに、高い水準の仮想化はオーバーヘッドが問題となる。実際、OS/omicron第２版というシステム[6]で OS 層とユーザプログラム層の間にユーザ拡張部という層を設け、仮名漢字変換、フレームメモリの描画機能などの機能拡張を行ったが、表示一体型タブレットの制御機能は性能劣下によりこの層で実現できなかった。

別の指針は、ある処理単位、例えばファイル管理、タスク管理ごとにモジュールを分割することである。実際、近年では分散処理のためにクライアント/サーバモデルでシステムを構築する方向にある。しかし、この場合クライアント/サーバ間の通信方式が性能と拡張性を規定することになる。さらに、サーバ自身の変更や拡張性を考えなくてはならない。

いずれにせよ、システム内のモジュール間のインタフェースを、効率を落とすことなく統一することが必要となる。

２．２　ソフトウェアバス

OS のモジュラリティを向上させるためには、モジュール間のインタフェース仕様を明確にすること、モジュール分割の指針を与えること、さらに効率のよいモジュール間の通信機構が必要となる。そこで、我々は OS を構成するモジュールが管理する資源とその操作の形式を定め、各モジュールをソフトウェアバスという仮想的なバックプレーンで結合する方式を考案した[7]。

例えば、ファイルシステムを名前検索・ページ管理・ディスク管理という３つのモジュールに分割したとする。これらをソフトウェアバスというバックプレーンに挿入し、その間で交信を行ってファイル入出力を行うモデルとなっている(図１)。ディスクキャッシュを追加するときは、ディスク管理モジュールをキャッシュ管理とディスク入出力の２つのモジュールに分割すれば容易に変更できる。

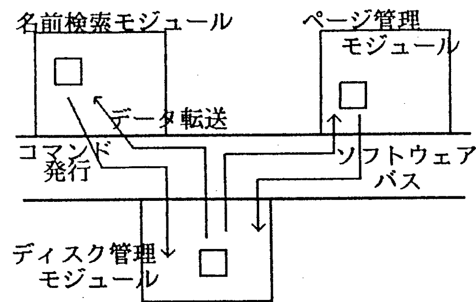

図１．ソフトウェアバスの概念図

ソフトウェアバスでは、モジュールが管理する資源をアドレス付けし、統一したインタフェースで各資源に対する操作を行う機構を提供する。ハードウェアのモデルで例えると、Multi バス、VME バスなどのコモンバスシステムの構成に類似している。図１に示すように、各資源間の操作は必ずバスを経由し、バスを経由しない資源の操作を仕様上禁止する。このような形態で OS の各機能を構築することにより、各資源とモジュールの独立性を高める。

このアーキテクチャは、次の特徴を有している。

(1) モジュール間のデータ転送・制御を行うために仮想的な共有バスを設定したこと

共有バスモデルを採用した理由は、ID 付けが容易なこと、データ転送を単なるメモリ転送にできること、その結果プログラミングが容易になるためである。

ソフトウェアバスは、(2)で述べるモジュール ID をモジュールの手続きのエントリに変換しモジュールへコマンドを発行する作業の他、モジュール間のバス使用に対する排他制御を行う。これは、モジュールをマルチプロセス・マルチプロセッサ上で実行することを可能とするためである。

論理的な機能単位一つに、一つのソフトウェアバスを割り当てる。機能単位が異なる場合は、異なるソフトウェアバスが割り当てられる。必要ならバスアダプタを介して異なるバスへアクセスする。

OS の各機能を分割し、それらをプロセスの集合体で実現しようという試みもあるが、ソフトウェアバスでは、モジュールの実行の基本をプロセス内の関数呼出しとし、小規模のモジュールに対しても効率を落とさないようにしている。

(2) モジュールが管理する資源をアドレス付けされたデータとして統一したこと

ソフトウェアバスでは、モジュールが管理する資源の単位を〈資源〉::= 〈資源実体，子資源名，資源属性〉と定めた。〈資源実体〉は、任意のバイト数を持つレコードの集合であり、一次元の線形なアドレス空間に写像されている。〈子資源名〉は〈資源実体〉に付随する資源の名前、〈資源属性〉は〈資源〉に固有な値である(図２)。例えば、ファイルはファイル実体とその下のファイルで定義される。

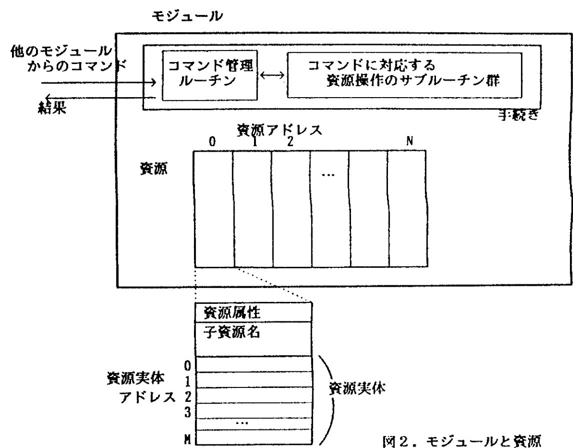

図２．モジュールと資源

モジュールは〈資源〉の集合を一次元のアドレス空間へ写像して管理する。一次元の線形なアドレス空間はランダムアクセスを行うため、子資源名はその資源の下部構造を管理するためである。モジュールには、システム全体でユニークな ID を割り当て、ソフトウェアバス上の資源は、〈モジュールID，資源アドレス〉で指定される。

(3) モジュールが管理する資源の操作を統一したこと

各モジュール内の資源に対するコマンドを表１に示す。資源に対する読み書きの操作の他、ファイルの版管理を行うため資源の複製を作成する DUPLICATE コマンドがある。コマンド LINK は、子資源名と資源を結び付けるコマンドである。複数の親資源から一つの子資源を共有する場合、論理的なファイル木を物理的なファイル木に変換するために導入した。

表１．ソフトウェアバスのコマンド

コマンド名	機能	コマンド名	機能
INITIALIZE	モジュールの初期化	DUPLICATE	資源の複製を作成
TERMINATE	モジュールの無効化	LINK	子資源の共有
OPEN	資源の使用開始宣言	LOCK	資源の排他制御
CLOSE	資源の使用使用終了宣言	UNLOCK	資源の排他制御
CREATE	子資源の作成	FLUSH	保持データの書込み
DELETE	子資源の消去	CANCEL	保持データの破棄
READ	資源実体の読込み	ABORT	資源操作の中断
WRITE	資源実体への書込み		
READ_STATE	資源属性の読込み		
WRITE_STATE	資源属性への書込み		
READ_NAME	子資源名の読込み		
WRITE_NAME	子資源名の書込み		

ソフトウェアバスは、共有メモリ型のマルチプロセッサシステム上で実現され、OS/omicron第３版[8]のファイルシステムに使用された。現在、ファイルシステムは、OS/omicronのタスクフォース(プロセス)として起動され、３つのタスク(スレッド)によりマルチサーバとして稼動している。しかも、ファイルの構造が異なるOS/omicron第２版と第３版のファイルシステムを同一のソフトウェアバス上に実現し、マルチファイルシステムとなっている。

２．３　ハイパＯＳ「礎」

ソフトウェアバスにより、OS の論理的な機能を分割し、それをモジュール化することが可能となった。ファイルシステムでは、シビアな応答性は要求されない。しかし、プリエンティブなスケジューリングを要求するモジュール、例えば表示一体型のタブレットのデバイスハンドラを含むモジュールをいかに扱うかが次に発生した問題である。

また、OS/omicron第３版はアノニマスカーネルであり、プロセスはどのプロッッサに割り当たるかはわからない構成となっている。これは、ユーザプロセスには有効であるが、応答性の要求されるプロセスは特定のプロセッサに割り当てた方が、時間設計が容易になる。したがって、あるソフトウェアバスやモジュールをプロセスやプロセッサに割り当て、プリエンティブに動かし、その間の通信を管理する機構が必要となる。

プロセスにプリエンティブが必要な場合、OS をリアルタイム核と従来の OS の２階層にわけたり、μカーネルと称する OS 核の上で時間制約の緩いプロセスを管理する OS を実現するアプローチがとられている。OS/omicronでも、従来プリエンティブが必要なデバイスハンドラなどを「江戸」[9]と呼ばれるハイパ OS 内に実現し、仮想化されたデバイス入出力のインタフェースを提供してきた。OS/omicron第３版もハイパ OS 上に実現されている。しかし、新規デバイスを接続し、そのためのモジュールを実現するには、ハイパ OS にデバイスハンドラを追加しなくてはならなかった。

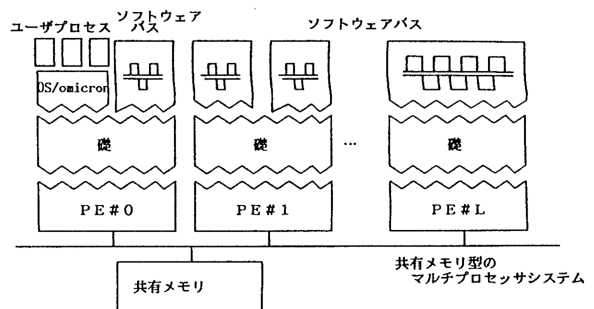

図３．ハイパＯＳ「礎」とソフトウェアバス

そこで、今回新たにハイパ OS 「礎」を設計・開発した。デバイスハンドラのモジュールを含むソフトウェアバスは、OS のプロセスとして実行されるのではなく、「礎」上のプロセスとして実行され、OS とソフトウェアバスの(礎の)プロセスは独立している(図３)。

３．ハイパＯＳ「礎」の設計方針

ハイパ OS 「礎」の設計方針と特徴は次のとおりである。

(1) 資源管理の対象を必要最小限とすること

「礎」は、資源管理を行うソフトウェアバスを実行するため、ソフトウェアバスの資源管理に制約を与えてはならない。そこで、「礎」の管理対象と機能を、CPU、割込み、(「礎」の)プロセス間通信、排他制御の４点とした。「礎」では、他のμカーネルのように、メモリ管理を行っていない。これは、仮想記憶管理も

ソフトウェアバスで実験したいためである。

(2) プリエンティブな核とすること

「礎」のプロセスは、タイムスライスを行わず、割込み駆動とする。また、「礎」プロセスは優先度をつける。ユーザプロセスのタイムスライスは、「礎」上で実行される OS が行う。

(3) ソフトウェアバス内のモジュール間、OS とソフトウェアバス間の通信機構を用意すること

これについては、後で述べる

(4) ソフトウェアバスのモジュールをロードするためのブートファイルシステムを用意すること

「礎」上で OS やデバイスハンドラが実行される。このとき、OS とデバイスハンドラをどこからロードするかが問題となる。つまり、「礎」はファイルシステムを持たないので、そのままでは OS をロードできないのである。ネットワークを使って、tftp などでロードする方法もあるが、サーバ側で同じ問題が生じる。

そのために、ブートストラップ専用のファイルシステムを作成した。このファイルシステムは、「礎」内部に組み込まれているのではなく、「礎」をロードするためのブート ROM に組み込んである。「礎」をロードするとき同時に OS やソフトウェアバスをロードする。「礎」の初期化ルーチンは、このロード情報をもとに「礎」のプロセスを生成する。

このファイルシステムは、区分編成のデータ管理構造とし、ディスクの動的なページ割当てを行わない。また、OS/omicron第３版のマルチファイルシステムにより、OS/omicron上で透過なアクセスができるようになっている。

４．「礎」のプロセスと割込み

「礎」のプロセスは、CPU のアドレス空間を互いに共有しあう。これにより、ソフトウェアバス間のデータ転送に、アドレス変換は不要となる。また、プロセス同士で手続きを共有しあうことも可能である。プロセスは、８段階の優先度を持っている。デバイスハンドラの優先度を OS の優先度より高くすれば、デバイス割込みに対処することもできる。また、プロセスは CPU の特権状態・ユーザ状態の両方で実行することができる。これは、OS の実現やデバイスハンドラのI/Oレジスタをアクセスするためである。

プロセス ID は、〈プロセッサID, プロセッサ内プロセス ID〉となっている。この ID はプロセス間通信で使用される。プロセッサ ID が 0 ならば、そのプロセスが属するプロセッサ内のプロセスを意味し、プロセッサ ID が 255 ならばブルードキャストを意味するが、現在実現されていない。

個々のプロセスは割込みを使用できる。トラップ命令、バスエラーなどのソフトウェア割込みは、個々のプロセスの事象ごとに発生する。さらにバスエラーは、そのフォールトアドレスの情報をプロセスにわたすことができる。デバイス割込みは、割込みの専用権を獲得したプロセスにおいて発生する。待ちを伴う SVC(SuperVisor Call)の実行中に割込みが発生すると、待ちは中断され即座に割込みルーチンに制御が移る。

プロセスの割込みエントリは、単一エントリで多重割込みを許さない構成とした。これは、割込み発生の回数を抑え、割込みスタックの管理を簡単にするためである。発生した割込み要因は、プロセス空間内におかれたビットベクトルでわかるようになっている。また、割込みマスクも SVC の発行回数を減らすため、プロセス空間内に配置した。

5．「礎」の通信機構

「礎」のプロセス間通信は、各プロセス間でコマンドパケットの交換を行うために設計された。設計にあたって、複雑なプロトコルを排除した。これは、ソフトウェアバスの機構上、各プロセスは密な関係にあるためである。コネクションは確定しており、ACK/NAK 制御、アクセス権のチェックも行わない。将来的には、ソフトウェアバスを分散システム上に拡張することも考えられるが、RPC[10]を作成するための基本機構を入れるにとどめてある。

むしろ、OS とソフトウェアバスのコマンドパケットの交換のために必要な機能を実現した。

(1) 送信は非同期型、受信は待ちを伴う形式とする

これは、OS 動かす場合、送信が非同期では OS のスレッドが停止してしまうためである。また、パケット受信の際に割込みを発生させることができる。これはソフトウェアバス側のパケット管理に柔軟性を持たせるためである。

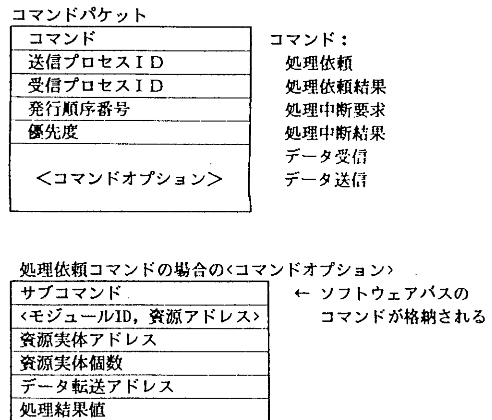

図4．「礎」プロセス間通信のパケット

パケットのキューは、各受信プロセスごとと、他プロセッサ用に用意した。

(2) パケットにパケット発行番号をいれ、OS 側で処理終了待ちを実現する

処理結果パケット待ちの処理を「礎」実現したのでは、通信管理の実験ができないので、OS 側で処理する方針とした。

(3) 処理依頼取り消しのパケットを導入する

OS 上のユーザプロセスを消去する際、ソフトウェアバスに発行したコマンドを取り消すことができないと、ユーザプロセスが獲得した資源を解放できない。そこで、「礎」のコマンドパケットに「以前に発行したコマンドを取り消す」コマンドを設けた。もし、対応するコマンドが受信側のキューにあるならば、「礎」はそのコマンドを取り外し、送信側に結果を返す。もし、受信側がすでにそのコマンドを取り込んでいたら、割込みを発生するようになっている。

(4) 通信の回数を減らすため、パケットそのものをバッファリングする

デバイスハンドラと OS を分離すると、バッファリングを行ってはいけないデータ転送(例えば会話型のデバイス、キーボード、マウスなど)の際、通信の回数が増加する。このため、コマンドキューに同種のパケットが存在する場合、まとめて一つのパケットとする処理を行っている。

「礎」のコマンドパケットの形式を図４に示す。「礎」の通信に関する SVC は次のとおりである。

・ポートID = ポートとして登録する(ポート名)

このプロセスがポート名に対応したサーバとなる。

・ポートID = ポートIDを得る(ポート名)
・パケットを送信する(ポートID, パケット)
　ポートID が負の場合、パケット中のプロセス ID へパケットを送信する。
・送信したプロセスID = パケットを受信する(パケット)

6．おわりに

OS のモジュラリティを向上し、拡張性を実現するためのソフトウェアバスとそのハイパ OS 「礎」について述べた。「礎」は 言語Ｃで約４Ｋ行、MC68030 VME CPU ボードのユニプロセッサシステム上で稼動している。プロセススイッチは、約 50μs である。言語Ｃからアセンブラに書き換えれば 30 μs ぐらいになるだろう。

現在、「礎」上に OS/omicron第２版・３版を移植し、手書きユーザインタフェースシステムをソフトウェアバスで実現する作業を進めている。今後の課題は、マルチプロセッサ用の拡張、OS の生成キットの作成、分散システムへの対応が残されている。

参考文献

[1]E.Dijkstra: The Structure of the 'THE'-Multiprogramming System, CACM, Vol.11, No.5(1968).
[2]M.Accetta, et al.: Mach: A New Kernel Foundation for UNIX Development , Proc. of Summer Usenix, 1986.
[3]田胡他: オペレーティング・システムの構造記述に関する一試み, 情報処理学会論文誌, Vol.25, No.4(1984).
[4]横手他: Muse: 次世代計算機環境構築のためのオペレーティング・システム 情報処理学会コンピュータ・システム・シンポジウム, 1991.
[5]高橋延匡: 研究プロジェクト総説: OS/omicron の開発, 情報処理学会 OS 研究会資料, 39-5, 1988.
[6]並木他: マルチプロセッサシステム向けのOS/omicronタスク管理の設計と実現, 情報処理学会論文誌, Vol.31, No.6, 1990.
[7]横関他: OS/omicron第３版ファイルシステム内部アーキテクチャの設計と実現, 情報処理学会 OS 研究会 47-2, 1990.
[8]岡野他: 並列処理用 OS カーネル "Omicron V3" とハイパ OS による共有メモリ型マルチプロセッサへの実装, 情報処理学会論文誌, Vol.32, No.5, 1991.
[9]岡野他: 多重 OS 「江戸」の設計と実現, 情報処理学会論文誌, Vol.30, No.8, 1989.
[10]A.Birrell, et al.: Implementing Remote Procedure Call, ACM Trans. on Compt. Syst., Vol.2, No.1, 1984.