システムソフトウェアと
オペレーティング・システム 77－32
マルチメディア通信と分散処理 87－32
（1998．2．27）

OS/omicron 第4版におけるシステム拡張機構の設計と実現

佐藤元信†、早川栄一†、並木美太郎†、高橋延匡‡
†東京農工大学工学部　‡拓殖大学工学部

〒184-8588　東京都小金井市中町2-24-16
E-mail：motonobu@omicron.ei.tuat.ac.jp

＜概要＞
本稿では、OS/omicron第4版（V4）における拡張性を重視したOS資源管理機構の設計と実現について述べる。V4はマルチメディアデータ処理の支援を目標とし、電紙という紙を仮想化したデータモデルを提供する。電紙はユーザが自由に拡張できる属性を持ち、属性ごとに処理手続きを登録することができる。このような粒度の小さい処理を拡張したい場合、従来のOSのようなタスクを単位とした機能拡張では性能の面で問題がある。そこで、オブジェクトを単位として機能を拡張する方式を提案した。すべての資源をメモリとして抽象化することで、オブジェクトはメモリを操作するメソッドを提供するだけで記述できる環境を提供する。拡張された機能は仮想関数として記述され、少ないオーバヘッドで呼び出すことができる。

＜キーワード＞
OS/omicron第4版、OSの拡張性、オブジェクト指向、セグメントマッピング方式

Design and Implementation of Extensible Resource Management on OS/omicron V4

SATO Motonobu †, HAYAKAWA Eiichi †, NAMIKI Mitarou † and TAKAHASHI Nobumasa ‡
† Department of Computer Science, Tokyo University of Agriculture and Technology
‡ Department of Computer Science, Takushoku University

2-24-26 Nakacho Koganei, Tokyo, 184-8588 Japan
E-mail：motonobu@omicron.ei.tuat.ac.jp

＜Abstract＞
This pager describes an extensible resource management mechanism on OS/omicron V4. OS/omicron V4 aims at supporting multimedia data processing with "DENSHI" data model virtualized real paper. "DENSHI" has an attribute and processing method functions. A process and IPC based extension that utilized on traditional OS as UNIX has context switching overhead. We suggest an extended management method based on object model that handles resource access as memory interface. When we extend OS/omicron V4, we add objects with methods and call methods implemented as virtual function. This extension mechanism presents less overhead than IPC based. We have implemented this mechanism on OS/omicron V4, and have evaluated it.

＜Key words＞
OS/omicron V4, Extensible OS, Object Oriented, Segment Mapping Method

1. はじめに

従来の計算機では文字列処理や数値計算を目的としていたので、OS もこれらのデータを効率よく扱えるように、データをファイルという形で抽象化して提供していた。最近では計算機の処理能力が向上し、OS の管理するデータの対象がマルチメディアデータへ移行してきた。これらのデータには多様性・多態性という特徴があるので、OS の資源管理には柔軟な拡張性が求められている。

最近の OS では、このような要求に対応するためにマイクロカーネル構成を採用することで、OS の資源管理の拡張性を確保している。しかし、これらの OS ではタスクを拡張の単位としているので、ファイルシステムのキャッシュアルゴリズムだけを動的に変更したいというような粒度の細かな処理の拡張には不向きである。また、システムの階層ごとに提供されるインタフェースが異なり、AP として実現したモジュールを OS の資源管理に取り込むといったことが困難である。

筆者の所属する東京農工大学工学部高橋・並木研究室では、マルチメディアにおけるパターンデータ処理を指向した OS/omicron 第 4 版（V4）の研究 [1] を行っている。この OS では「電紙」と呼ばれるユーザが拡張できるデータモデルを提供する。そこで、資源をすべてメモリとして抽象化し、資源を管理するモジュールをオブジェクトとして部品化する資源管理方式を提案した。本稿では、V4 における OS の資源管理拡張方式の設計と実現について述べる。

2. OS/omicron 第 4 版における資源管理への要求

2.1 OS/omicron 第 4 版の概要

V4 はパターンデータ処理を指向した OS である。パターンデータの中でも特に手書きデータに着目し、紙を仮想化した「電紙」と呼ばれるデータモデルをシステムで提供する。電紙にはそのデータの性質を表す属性があり、属性ごとに処理手続きを登録することができる。どの手続きを呼べばよいかは実行時まで確定できないので、V4 ではダイナミックリンクを提供する。

また、ある電紙を別の電紙に変換するのは、属性変換として記述される。変換された電紙同士は、この電紙を変換した結果がこの電紙である、のようなリンク関係を持つ。これらの手続きは属性ごとに一つのモジュールとして定義される。

電紙データモデルは、ユーザが自由に属性を追加することができるので、システムがこのようなデータモデルを提供するには、柔軟な資源管理を提供しなければならない。そこで V4 ではマイクロカーネル構成を採用し、ワンレベルストアにより電紙が二次記憶上ある場合のリンク関係をポインタで表せるようにする。V4 の全体構成を図 1 に示す。

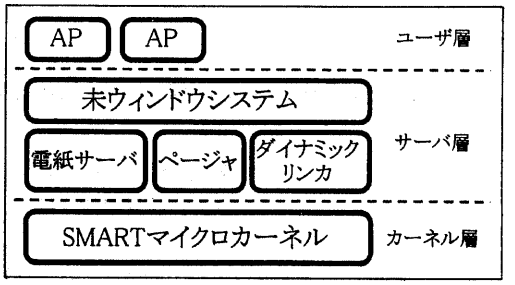

図 1　V4 の全体構成

2.2 機能拡張に対する要求

電紙データモデルを提供するには、OS の資源管理がデータの属性とその処理手続きの関係を管理しなければならない。属性はユーザ定義可能なので、ある属性を処理するためのモジュールは資源管理とは独立し、個々に追加できるような構成を取らなければならない。

個々の属性を管理するモジュールを拡張可能にするには、OS の提供するプリミティブを利用することでインタフェースを抽象化しなければならない。タスクを資源管理の対象とするような OS で実現する場合、個々のモジュールはサーバとして実装され、ポート間通信などのプリミティブによりインタフェースを抽象化することになる。

しかし、このような実装では電紙を操作するたびにタスク間通信のオーバヘッドがかかり、性能の面で問題となる。そこで、V4 の資源管理では、モジュール間のインタフェースを抽象化するための軽量なプリミティブを用意しなければならない。また、インタフェースを抽象化するために、すべての資源を統一的に管理できるような機構を提供しなければならない。

2.3 システム拡張機構の目的

先に示した要求から、システム拡張機構の目的を次のように設定した。

(1) 同じインタフェースで管理できるように資源を抽象化する
(2) 動的にモジュールを追加できる
(3) 少ないオーバヘッドで拡張された機能を呼び出せる

3. 設計方針

これらの目的を達成するために、システム拡張機構の設計方針を次のように定めた。

(1) 資源をオブジェクトとして管理する

V4 の提供する電紙データモデルでは、データが属性と手続きの組みを持つことになる。そこで、V4

の資源管理ではデータ、属性、手続きを一つの組みとして管理できるように、資源をオブジェクトとして管理する。

(2) 資源をすべてメモリとして抽象化する

UNIX に代表されるような従来の OS では、資源をファイルとして抽象化する。この方法ではすべての資源を open-close-read-write というインタフェースで扱うことができる。しかし、このインタフェースで扱いにくいものは、すべて ioctl という特殊なインタフェースに押し付けることになり、OS による抽象化を放棄する形となった。その結果、モジュールは独立しているが、個々のモジュールを差し替えるといったことが困難になった。そこで、V4 では資源はすべてメモリとして抽象化し、どのような資源でも特別な手続きを呼ぶことなくランダムアクセスできようにする。資源をメモリとして提供することで、すべての資源をランダムアクセス可能とし、オブジェクトとして定義できるようにする。

(3) オブジェクト指向言語の仮想関数を利用する

モジュール間の接続の透過性を保つには、何らかの形でインタフェースを抽象化し、呼び出される先のモジュールを隠蔽しなければならない。Mach ではポート間通信を使うことで実現されるが、タスク間通信のオーバヘッドが問題となり、粒度の細かい処理の呼出しには向かない。UNIX のファイルシステムの実装のように、関数ポインタ表により抽象化する方法 [2] もあるが、ユーザの責任で関数ポインタ表を管理しなければならないので、記述性の面で問題がある。そこで、V4 ではオブジェクト指向言語の提供する仮想関数を利用することで、関数ポインタ表をユーザが管理する手間を軽減する。仮想関数を用いれば、言語処理系が表の生成、管理を行うので、ユーザの間違いによる事故が発生しにくくなる。これ以降、言語 C++ をターゲット言語として説明する。

4. 拡張機構の設計

4.1 拡張のモデル

ここでは、システム拡張のモデルについて述べる。方針で述べたように、このシステムでは資源をすべてメモリとして抽象化し、オブジェクトとして管理する。オブジェクトは単にメモリを操作するメソッドを持つだけなので、メモリにマップされている資源が何であるかに関係なく利用することができる。

オブジェクトには木構造上の文字列である名前が付けられ、他のオブジェクトを利用する場合、記述言語の中では外部参照として記述される。外部参照は実行時にダイナミックリンクによりオブジェクトがリンクされ、利用できるようになる。他のオブジェクトのメソッドを呼び出す場合も同様にダイナミックリンクにより実行時に結合される。このバインドを実行時に変更することで、動的な変更を支援する。システム拡張のモデルを図 2 に示す。

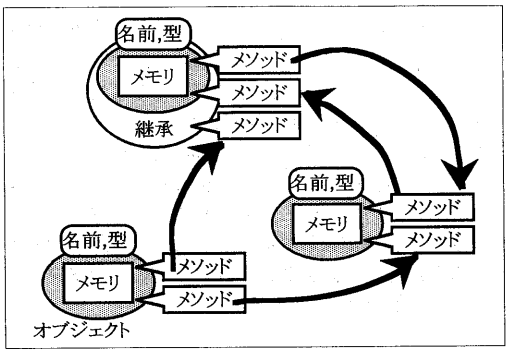

図 2 システム拡張のモデル

4.2 全体構成

ここでは個々のモジュールの役割を説明する。システム拡張機構の全体構成を図 3 に示す。

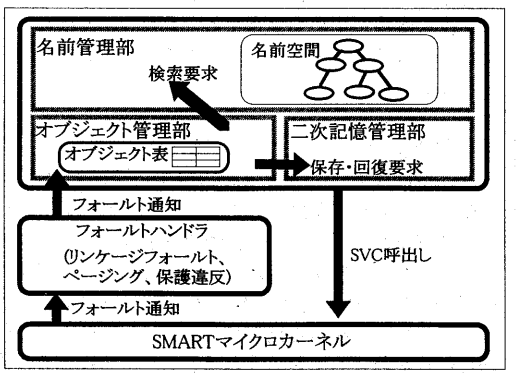

図 3 システム拡張機構の全体構成

(1) オブジェクト管理部

ここではシステムが発生するイベント（保護違反フォールトやページフォールト）をオブジェクトに通知するためにオブジェクトの管理を行う。システムの発生するフォールトは、仮想アドレスを返すだけなのでオブジェクトを特定することができない。ここではオブジェクトと仮想アドレスの対応を管理

する。生成されたオブジェクトには必ず名前が付けられ、名前、型（クラス）、オブジェクトへのポインタの組みとして名前管理部に登録される。

ここではフォールトアドレスからオブジェクトを特定するための処理や、オブジェクトの生成・消去といったインタフェースをユーザに提供する。

(2) 名前管理部

ここではシステム拡張機構が管理する資源の名前を管理する。管理している名前は、オブジェクト名、クラス名、メソッド名、外部変数名などである。クラス名は、クラスの継承関係に基づいて木を構成し、その木によってモジュールの管理を行う。

名前は型、実体へのポインタと組みになって管理される。システムが名前の意味について規定してしまうと拡張性が失われるので、ここでは対応関係だけを管理し、名前の登録、検索などのインタフェースだけを提供する。

(3) 二次記憶管理部

生成されたオブジェクトは、システム終了後も保存されるように永続化することができる。これは従来のファイルに当たる機能である。このモジュールでは、永続化されたオブジェクトを二次記憶上のどこに格納するかを管理する。二次記憶に格納する際は、再度メモリ上に展開したときに正しいアドレスを指示できるように、オブジェクトに含まれるポインタを変換してから保存する。ただし、保証されるポインタは永続オブジェクトを指しているポインタだけである。

4.3 拡張機構

ここでは実際に拡張する際の機構について述べる。拡張はクラスを定義したモジュールを単位として行う。ある機能を拡張したい場合、その機能を提供するクラスを継承したクラスを定義したモジュールを作成すればよい。利用する側は、基本クラスへのポインタを使ってオブジェクトを操作すればよい。拡張される機能は仮想関数として記述される。これらの関数の呼出しは、実行時のダイナミックリンクによりリンク先が確定される。

問題となるのは、オブジェクトの生成部分である。言語 C++ では演算子 new の後に示す型名はリテラルしか指定できないので、実行時に任意の型のオブジェクトを生成することができない。そこで、拡張機構では演算子 new に代わるインタフェースを提供する。このインタフェースは演算子 new と同様の働きをするが、後ろに記述する型名が変数であることが異なる。このインタフェースと利用例を図 4、5 に示す。

```
// インタフェース定義
void* ___new( char* type );
    type =  クラス名を示す文字列
```

```
PARENT*  obj;
// このインタフェースを使った場合
obj=___new( type );
```

```
// 従来の書き方
if( strcmp(type,"CHILD1")==0 ){
  obj=new CHILD1;
} else if( strcmp(type,"CHILD2")==0 ){
  obj=new CHILD2;
}
```

図 4 オブジェクト生成のインタフェース

```
class DISPLAY : public FRAME {
  ...
public:
  virtual  int  reduceColor();
};

class NEW_DISP : public DISPLAY {
  ...
public:
  int  reduceColor();
};

DISPLAY*  disp;
disp = ___new( "DISPLAY" );
disp->reduceColor();
```

図 5 オブジェクトの拡張例

図 5 はディスプレイドライバの減色処理関数を拡張可能にする例である。動的に変更・拡張がしたい手続きは、基本クラス DISPLAY で仮想関数として定義する。拡張する場合、DISPLAY を継承した新しいクラスを定義し、仮想関数を定義すればよい。利用する側は ___new () を利用することでオブジェクトを生成する。実行時にクラス名の解釈を変えることで、拡張されたオブジェクトを生成できるようになる。

新たに拡張されたクラスは、名前管理部の持つクラス木に登録される。ユーザが初めてこのクラスを利用したときにダイナミックリンクが発生し、オブジェクト管理部がフォールトを受け取る。オブジェクト管理部は名前管理部に名前の検索を依頼し、結果に基づいてリンクの処理を行う。この処理の流れを図 6 に示す。

4.4 メモリへの抽象化機構

資源のメモリへの抽象化には、セグメントマッピング方式 [3] を利用する。これはフォールトを利用することでユーザのメモリアクセスを監視し、それをタイミングとしてデバイスなどを該当するメモリにマッピングするという方式である。

この方式で扱う資源は大きく分けて 2 種類ある。一つはハードディスクのような固定された領域を持つよ

うな資源である。これらはページフォールトを利用することでメモリへ抽象化される。

もう一つは、キーボードデバイスのようにデータが順次格納されていく、サイズが未確定な資源である。ファイルのように書き込むごとに領域が広がるものも同様である。これらの資源をメモリに抽象化するには、ユーザのメモリアクセスを 1 バイト単位で監視しなければならない。そこで保護違反フォールトを利用する。これはセグメントを保護するためにアクセス可能な範囲が指定できるという機能で、これを未到着データへのアクセス監視に利用する。メモリへの抽象化機構を図7に示す。

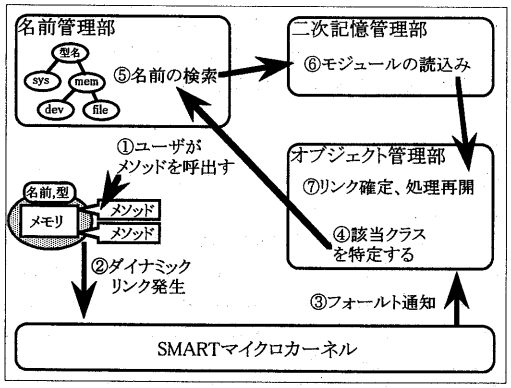

図6 拡張機構の処理

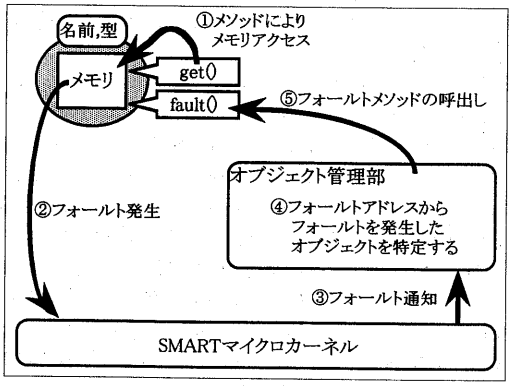

図7 メモリ抽象化機構の処理

5. 拡張機構の実現と評価

5.1 実現環境

これらの設計に基づき、システム拡張機構の実現を行った。実現はPC/AT互換機用に実現されたSMARTマイクロカーネル [4] 上で行った。また、記述言語として当研究室で開発中の言語 C++ 処理系 [5] を利用した。このコンパイラでは、二次元アドレス空間やダイナミックリンクに対応したコード生成を行う。

実現は拡張機構とシステムオブジェクト、いくつかのデバイスドライバに対して行った。デバイスドライバは、セグメントマッピング方式を採用したデバイス管理をこの拡張機構上にのせることで実現した。これらの実現規模を表1に示す。

表1 実現規模

モジュール名	行数
オブジェクト管理部	2025
名前管理部	3547
フォールトハンドラ	4281
システムオブジェクト	2878
デバイス管理	1324
デバイスドライバ	2426

5.2 性能評価

今回実現した拡張機構を、性能の面から評価した。測定環境は表2のとおりである。

表2 測定環境

ハードウェア	IBM Think Pad 560E
プロセッサ	MMX Pentium/166MHz
主記憶容量	47.6Mバイト
ハードディスク	EIDE 2Gバイト

5.2.1 測定項目

今回測定したのは次の三つの項目である。

(1) 拡張された機能の呼出し時間

オブジェクトにより追加された機能を呼び出すためのオーバヘッドを測定した。測定は、無処理の仮想関数を呼び出してから戻って来るまでの時間である。比較対象として、単なる関数呼出しとタスク呼出しの時間も測定した。

(2) ダイナミックリンクのオーバヘッド

拡張された機能を初めて呼び出したときにかかるオーバヘッドを測定した。測定はリンクの確定していないメソッドを呼び出してから戻ってくるまでの時間である。モジュールがすでにメモリ上に展開されている場合と、二次記憶上にある場合の 2 通りについて測定した。

(3) フォールトの通知時間

資源をメモリとして抽象化するためのプリミティブとなる、フォールト通知の性能を測定した。測定はオブジェクト管理部がフォールトを受け取ってから、オブジェクトのメソッドが呼び出して戻ってく

るまでの時間である。

測定結果を表3に示す。

表3　測定結果

測定項目	時間(μs)
仮想関数呼出し	3.77
通常の関数呼出し	1.21
タスク呼出し	59.30
ダイナミックリンク(メモリ上)	113.12
ダイナミックリンク(二次記憶上)	19958.96
フォールト通知	134.49

5.2.2 考察

それぞれの項目について考察する。

まず、(1) についてだが、仮想関数の呼出しは通常の関数呼出しの約 3 倍の時間がかかっている。この原因は仮想関数表による間接呼出しのオーバヘッドである。これに対して、従来の OS のようにタスク間通信を利用してモジュールを拡張する場合、V4 のコンテクストでは 59.3 μs かかり、本方式の約 16 倍の時間がかかる。粒度が細かい処理を拡張する場合には、このオーバヘッドが問題となる。この測定から、仮想関数による機能の拡張が有効であったことが分かる。

次に (2) について考察する。(1) の仮想関数による拡張では、必ず最初の一回目の呼出しでダイナミックリンクが発生する。このオーバヘッドは約 110 μs であり、仮想関数の呼出し時間と合わせると 113 μs 程度であり、タスクによる実現のおよそ倍の時間がかかることになる。しかし、このオーバヘッドがかかるのは最初の一回だけであり、それ以降は (1) のオーバヘッドだけになる。このことから、仮想関数とダイナミックリンクによる拡張機構は、3 回以上呼び出される機能では有効であることが分かる。

最後に (3) だが、この機能はメモリへの抽象化機構の中核であり、抽象化の性能を大きく左右する要素である。ページフォールトの場合、メモリへの抽象化をしない場合でも必ずページフォールトが発生するので、このオーバヘッドは他のシステムと比較しても問題にならないと考える。保護違反フォールトは通常のシステムでは保護に利用され、フォールト後はタスクを終了させるだけなのでオーバヘッドは問題とならない。しかし、本拡張機構では、ユーザのメモリアクセスの監視に利用するので、このオーバヘッドが重要になる。ハードウェアバッファの少ないデバイスなどでは、さらに性能を向上させなければならない。

6. おわりに

本稿では、OS/omicron 第 4 版におけるシステム拡張機構の設計と実現について述べた。本研究で得られた成果は次のとおりである。

・資源をメモリとして抽象化し、すべての資源をオブジェクトとして管理できる環境を提供した
・資源管理を動的に拡張できる機構を提供した
・オーバヘッドの少ない機能拡張を提供した

今後の課題としては、メモリとして抽象化する際のオーバヘッドを削減し、ウィンドウシステムなどをこの拡張機構上で実現して、この方式の有効性の検証をすることなどが挙げられる。

謝辞

本研究の一部は、文部省科学研究費補助金（基盤研究 (A) (2) 09358004、基盤研究 (B) (2) 08458064、奨励研究 (A) 09780255）により行われた。

参考文献

[1] Hayakawa, et.al : "Basic Design of SHOSHI Operating System that Supports Handwriting Interface", 情報処理学会論文誌, Vol.35 No.12

[2] M. K. McKusick, K. Bostic, M. J. Kareles, J. S. Quarterman : "The Design and Implementation of the 4.4 BSD UNIX Operating System", Addison-Wesley, 1996

[3] 佐藤元信, 森永智之, 早川栄一, 並木美太郎, 高橋延匡 : "OS/omicron 第 4 版のデバイス管理におけるシーケンシャルデバイス管理方式の拡張", 情報処理学会コンピュータシステムシンポジウム論文集, 1996, pp.179-186

[4] T. Morinaga, M. Sato, E. Hayakawa, M. Namiki and N. Takahashi : "Throw : An Efficient and Extensible Structure Model for Microkernel Architecture", Information Systems and Technologies for Network Society, 1997, pp.133-140

[5] 加藤泰志, 早川栄一, 並木美太郎, 高橋延匡 : "オブジェクト指向による OS/omicron 第 4 版を実現するための言語処理系の設計", 情報処理学会オブジェクト指向' 95 シンポジウム論文集, 1995, pp.103-108

[6] Andrew S. Tanenbaum : "Modern Operating Systems", Prentice-Hall International, Inc., 1992

[7] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young : "Mach A New Kernel Foundation For UNIX Development", USENIX summer' 86, pp.93-112, 1986