外部プログラムとキャッシングを利用した堆積可能汎用フィルタ・ファイル・オブジェクトの実現方式

新城 靖†

オブジェクトの堆積とは，object-based システムを構造化するためのモデルである．オブジェクトの堆積モデルでは，一様なインタフェースを持つオブジェクトの層を形成することにより，それらが持つ機能が統合される．本論文は，オブジェクトの堆積モデルに基づきフィルタ機能を提供するオブジェクトの実現方式と性能について述べている．フィルタとは，入力に対して，たとえば文字の変換，行単位の検索，ソーティングのような処理を行い，結果を出力するものである．本論文で述べているフィルタ・ファイル・オブジェクトの実現方式の特徴は，第1に，外部のフィルタ・プログラムを利用する点にある．これにより，1つのサーバで複数の種類の利用者定義のフィルタ機能を提供することが可能となる．第2の特徴は，キャッシングの技術を利用していることである．キャッシングにより高速化が実現されると同時に，フィルタ機能の実現が容易になっている．本論文では，実現したフィルタ・ファイル・オブジェクトの性能を示している．

An Implementation Method of Stackable Generic Filter File Objects Using External Programs and Caching

Yasushi Shinjo†

Object-stacking is a model for structuring object-based systems. In this model, a layer of objects with a uniform interface is constructed, and the functions of these objects are integrated. This paper describes an implementation method of filter file objects that provide filtering functions based on the object-stacking model. Filtering means processing of input data and writing results, such as translation of characters, selection of lines, and sorting. The first feature of the implementation method of the filter objects is the use of external filter programs. This method makes it possible to provide several kinds of user-defined filtering functions by a single server. The second feature is the use of a caching technique. Caching makes it possible to improve performance and simplify the implementation of the filter objects. This paper also shows the performance of the implemented filter file objects.

1. はじめに

UNIX の make コマンドは，本来はコンパイル作業を支援するために開発されたものである[6]．最近では，データの依存関係や複数の視点間の一貫性を維持する道具としても活用されている．ここで**視点（view）**とは，ある元データに対してなんらかの関数的な処理を施した結果で，元データが更新されたときには必ず更新されるようなものである．たとえば，ソーティング，圧縮，画像のデータ形式の変換といった処理を施した結果は，元データに対する視点と考えることができる．make コマンドは，与えられたファイルの最終更新時刻を調べ，依存関係に矛盾を発見すると，指定されたコマンドを実行し，矛盾を解消することを試みる．make コマンドを使えば，データの複数の視点間の一貫性を維持することがかなり容易になる．ただしそのためには，一貫性が必要になった時点で明示的に make コマンドを実行する必要がある．

分散型ファイル・システムの最も重要な研究課題の1つとして，キャッシングや複製（replication）の実現があげられる[8]．キャッシングや複製の本質は，高い性能と可用性を提供することを目的として，複数のノードにコピーを作成し，それらの間の一貫性を維持することである．これらは，あるファイルのローカルにある視点，すなわち，高速にアクセスでき，存在すれば必ず利用可能であるという性質を持つ視点の研究としてとらえることができる．今後は，単なるコピーだけではなく，複数の視点間の一貫性を維持することが重要な課題になってくることが予想される．

† 筑波大学電子・情報工学系
Institute of Information Sciences and Electronics, University of Tsukuba

ファイル・システムや World Wide Web システムにおいて複数の視点を効率的に提供することを目的として，オブジェクトの堆積モデルの研究を行っている[5),10),13)]．**オブジェクトの堆積（object-stacking）**は，object-based システムを構造化するためのモデルであり，その特徴は，一様なインタフェースを持つオブジェクトを結合することで，結合されたオブジェクトが持つ機能を統合して利用することができる点にある．この論文では，オブジェクトの堆積モデルに基づくファイル・システムにおいて最も重要なオブジェクトである**フィルタ・ファイル・オブジェクト（filter file object）**の実現について述べる．このオブジェクトは，UNIX のフィルタ型コマンドと同様に，1 つ 1 つは，単純なフィルタ機能を提供するファイル・オブジェクトであるが，それらを組み合わせることで，複雑な機能を提供することができるものである．

UNIX のフィルタ型コマンドは，結果をパイプを通して提供するのに対して，フィルタ・ファイル・オブジェクトは，結果をファイルの形で提供する．よって，ランダム・アクセスやシークといったファイルに対して許される任意のアクセス方法が利用可能である．さらに，キャッシングの技術を利用し，処理の結果を保存することで，大幅に性能を改善することが可能となる．こうして得られた処理の結果は，make コマンドを明示的に実行しなくても自動的に一貫性が維持される．オブジェクトの堆積に基づくファイル・システムには，フィルタ・ファイル・オブジェクトの他にも，オブジェクトの移動を実現する間接オブジェクトやディレクトリ内容について融合処理やフィルタ処理を行うディレクトリ・オブジェクトがある[10),11)]．

この論文で述べるフィルタ・ファイル・オブジェクトの実現方式の特徴は，第 1 に，外部のフィルタ・プログラムを利用する点にある．従来の研究では，フィルタ機能がサーバ内部で実現されていたため，圧縮や暗号化といったあらかじめ定義されたフィルタから選択して利用することしかできなかった[4)]．本実現方式により，従来の研究の応用である圧縮や暗号化に加えて，UNIX で利用可能なフィルタ機能の大部分や利用者定義のフィルタ機能を利用することが可能となる．

本実現方式の第 2 の特徴は，キャッシングの技術を利用していることである．フィルタ・ファイル・オブジェクトは，通常のファイルと比較してフィルタ処理のオーバヘッドが加わるが，キャッシングの技術の利用により，キャッシュが有効なときにはディスク入出力時間に束縛される程度の高速化が実現される．さらに，ファイル単位のキャッシングにより，ファイルの入出力位置の複雑な計算を行う必要がなくなるので，行単位やファイル単位のフィルタ機能の実現が容易になる．

オブジェクトの堆積に基づくファイル・システムにおいて解決すべき様々な課題[10)]のうち，本論文で述べるフィルタ・ファイル・オブジェクトにより，サーバ開発と実行時の資源に関する課題が解決される．オブジェクトの堆積では，基本的には 1 つのオブジェクトが 1 つの機能を提供する．そして 1 つのサーバは，1 種類のオブジェクトを管理する．よって多くの種類のフィルタ機能を提供しようとすると，それに相当する数のサーバを実現し，同時に実行しなければならない．よく利用されるフィルタ・ファイル・オブジェクトの開発を容易にし，実行時に必要な資源量を減らすことが，オブジェクトの堆積を実現するうえでの課題になっていた．この課題は，本論文で述べる外部のフィルタ・プログラムを利用するサーバにより解決される．このサーバは，単一のサーバでありながら様々な種類のフィルタ機能を提供することができる．これにより，多くの種類のサーバを開発し動作させることと比較して，実現のコストと実行時の資源が大きく節約される．

本論文は，次のように構成されている．2 章では，オブジェクトの堆積モデルとフィルタ・ファイル・オブジェクトの利用について述べる．UNIX のパイプとフィルタ・ファイル・オブジェクトと利用形態の比較も行う．3 章では，汎用のフィルタ・ファイル・オブジェクトの実現方式について述べる．その特徴は，外部のフィルタ型コマンドを利用していること，および，キャッシングを利用していることにある．4 章では，実現したフィルタ・ファイル・オブジェクトの性能について述べる．5 章では，関連した研究との比較を行う．6 章では，まとめを行う．

2. オブジェクトの堆積モデルとフィルタ・ファイル・オブジェクト

オブジェクトの堆積の基本的な考え方は，一様なインタフェースを持つオブジェクトを積み重ねることである．

オブジェクトとは，手続きとデータをカプセル化したものであり，公開された手続きによってのみ内部のデータの操作が許されるものである．オブジェクトの堆積では，オブジェクトは，**オブジェクト識別子（object identifier, object ID）**によりつねに間接的に参照される．分散型オペレーティング・システムでは，オブジェクトの管理と保護は，サーバ・プロセスにより行われる．たとえば，ファイル・オブジェク

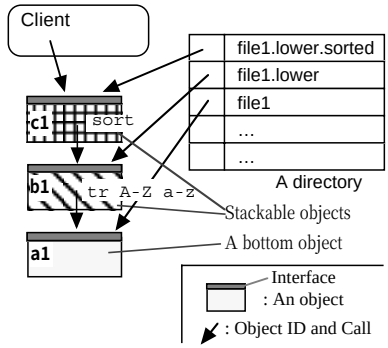

図 1 オブジェクトの堆積による機能の統合
Fig. 1 Integrating the functions of objects by stacking.

トは，ファイル・サーバにより管理されている．なお以下では，オブジェクトを中心に述べ，オブジェクトを管理しているサーバについては，必要なときにのみ言及する．

インタフェースとは，オブジェクトが受け付けることができる手続きの集合である．たとえば，ファイル・オブジェクトは，インタフェースとして読み込み手続きと書き込み手続きを持つ．

オブジェクト a1 の上にオブジェクト b1 を**積み重ねる**（to stack b1 on a1）とは，次の 2 つの条件を満たすことである．

(1) b1 が a1 のオブジェクト識別子を保持している．
(2) b1 が自分自身の機能を実現するために，必ず a1 を呼び出す．

このとき，上位層のオブジェクト b1 は，クライアントとして下位層のオブジェクト a1 を利用する．

図 1 に，3 つのオブジェクトが積み重ねられている様子を示す．上の 2 つのオブジェクトは，**堆積可能オブジェクト（stackable objects）**であり，それぞれ下位層のオブジェクトの識別子を保持している．最下位層のオブジェクトは，**基底オブジェクト（bottom object）**であり，他のオブジェクトの識別子を保持していない．

オブジェクトを自由に組み合わせて利用するためには，それらのオブジェクトのインタフェースが**一様（uniform）**である必要がある．一様なインタフェースのオブジェクトを利用することにより，積み重ねるオブジェクトの構成と順序を自由に選択することが可能となる．

図 1 では，次の 3 つの機能を持つファイル・オブジェクトが積み重ねられている．

(a1) データを保持する．
(b1) 大文字を小文字に変換する．
(c1) 行単位のソートを行う．

ここで，(b1) と (c1) は，堆積可能な**フィルタ・ファイル・オブジェクト**であり，下位層のファイル・オブジェクトの内容にフィルタ処理を行った結果を提供する．最上位層のファイル・オブジェクトをアクセスすることにより，これらの機能が統合される．すなわち，小文字に変換されソートされたデータを保持しているファイル・オブジェクトが作られている．

図 1 において，各堆積可能オブジェクトは，クライアント（上位層の堆積可能オブジェクトを含む）から読み込み要求を受け付けると，対応する下位層のオブジェクトにクライアントとして読み込み要求を送る．次に，下位層のオブジェクトから返されたデータに対して，ソートを行う，あるいは，大文字を小文字に変換する処理を行う．そして，処理を行った結果をクライアントに返す．基底オブジェクトは，普通のファイルと同様に，ディスク・ブロック等からデータを読み込み，他のオブジェクトを呼び出すことなく処理を行い，結果をクライアントに返す．

図 1 に示されたオブジェクトの場合，中間的な層もディレクトリに登録されており，名前によりアクセスすることが可能になっている．これは，オブジェクトの堆積により，**多重視点（multiple view）**が提供されていることを意味する．図 1 では，次のような視点が提供されている．

(a1) 変換前の元の視点
(b1) 小文字に変換された視点
(c1) 小文字に変換されソートされた視点

このような多重視点の機能を有効に利用するためには，堆積可能ファイル・オブジェクトと基底ファイル・オブジェクトを，図 1 に示したように同一のディレクトリに保存することが必要な場合がある．さらに一般的には，ファイル・オブジェクトの種類に関係なく，ファイル・オブジェクトに名前を付けることができる機能が重要となる[10]．NFS[14]のように，ファイル・オブジェクト本体と名前付けの機能を一体的に実現する必要がある場合，図 1 のように単一のディレクトリに置くことはできない．

この論文では，オブジェクトの堆積モデルに基づき，RPC でアクセス可能なユーザ空間のオブジェクトを接続して利用する方法について述べる．オブジェクトの堆積モデルは，文献 10) で示したような条件が整えば，システム空間のオブジェクトを接続するために使うことも可能である．システム空間にあるモジュールを接続する仕組みである，UNIX System V のストリーム[1] との比較は，5.3 節で行う．

2.1 UNIX のパイプとの比較

UNIX では，永続的なデータを保持するファイルへのアクセスとプロセス間通信機能であるパイプが，ある範囲で同一のインタフェースで利用することができるようになっている．UNIX で動作するフィルタ型コマンドの多くは，次のようなパイプの制約を守っている．

- データを先頭から末尾へと逐次的にアクセスしなければならない．
- 同一のデータは，1 度だけしかアクセスできない（テープを巻きもどすような操作を使うことができない）．

逆にいえば，ファイルには，次のような便利な性質がある．

- データをランダムにアクセスすることができる．
- 同一のデータを複数回アクセスすることができる．

このようなファイルの性質は，ソーティングのように，もともとランダム・アクセスが必要な処理や，画像データや内部にポインタなどを含む複雑な構造体のように，必ずしもデータの先頭から順に処理していくことができないデータに対するフィルタ処理に有用である．

一方，オブジェクトの堆積モデルに基づくフィルタ・ファイル・オブジェクトは，通常のファイルとまったく同一のインタフェースによりアクセスすることが可能になっている．すなわち，上で述べたファイルの性質を活用することが可能になっている．よって，UNIX のパイプとオブジェクトの堆積モデルに基づくフィルタ・ファイル・オブジェクトは，相補的な面があるといえる．すなわち，UNIX のパイプは，次のような応用に適している．

- 入出力データをストリームとして扱うもの．
- 1 度だけ利用するデータを生成するもの．

フィルタ・ファイル・オブジェクトは，次のような応用に適している．

- 入出力データをランダム・アクセス可能なファイルとして扱うもの．
- 何度も参照されるデータを生成するもの（生成されたデータは複数のクライアントによって同時に参照することができる）．

3 章で述べる実現方式によるフィルタ・ファイル・オブジェクトでは，オブジェクトの堆積モデルに基づく一般的なフィルタ・ファイル・オブジェクトと比較して，いくつかの制約がある．まず，内部でフィルタ処理を実現するために，パイプを入出力とするフィルタ型コマンドを使っているので，扱えるフィルタのクラスがパイプで扱えるものよりも狭くなっている．そのほかに，ランダム・アクセスが可能という性質については，クライアントに対しては提供されているが，自分自身の実現，すなわち，下位層のオブジェクトをアクセスするときには，活用されていない．

3. 堆積可能汎用フィルタ・ファイル・オブジェクト

オブジェクトの堆積では，基本的には 1 つのオブジェクトが 1 つの機能を提供する．1 つのオブジェクトで複数種類のフィルタ機能を提供できるような堆積可能フィルタ・ファイル・オブジェクトを，**堆積可能汎用フィルタ・ファイル・オブジェクト**（以後，単にフィルタ・ファイル・オブジェクト）と呼ぶ．この章では，汎用のフィルタ・ファイル・オブジェクトの実現方式について述べる．その特徴は，外部のフィルタ・プログラムを利用していること，および，キャッシングの技術を利用していることである．

3.1 実現環境

本論文で述べるオブジェクトの堆積に基づくファイル・システムは，RPC によりアクセスされる☆．RPC としては，現在 SunRPC を用いている[14]．これに，一様なインタフェースを生かして独自の手書きのスタブを加えて，引数と結果が 1 つずつという SunRPC の制限を緩和している．

サーバは，複数の軽量プロセス（lightweight process，スレッド）を含む[12]．クライアント・プロセスからの要求を受け付けるたびに軽量プロセスが作られる．

3.2 インタフェース

堆積可能フィルタ・ファイル・オブジェクトで最も重要な手続きは，オブジェクトを生成する手続き `create()` と，内容を返す手続き `read()` である．それぞれのインタフェースを，**図 2** と**図 3** に示す．ここで，キーワード `in` は，サーバ側から見て入力パラメータを表す．キーワード `out` と手続きのリターン・バリューは，出力パラメータを表す☆☆．

図 2 に示した手続き `create()` が呼ばれると，サーバは，`lowers` で指定されたオブジェクトを下位層のオブジェクトとして新たにオブジェクトを生成する．このとき，下位層のオブジェクトの型を取得し，積み重ねることができるかを検査し，積み重ねることがで

☆ 既存の UNIX のアプリケーションからこのような RPC に基づくサービスを利用する方法としては，NFS インタフェースを設ける方法[10]，互換ライブラリを用いる方法[9]，UNIX サーバを呼び出す方法[2),3)]と類似の方法を用いる方法が考えられる．

☆☆ キーワード `in` と `out` は，インタフェースを説明するために導入した表記であり，実際のプログラムには現れない．

```
stat_t create( in oid_t lowers[],
  in option_t options, out oid_t newobjs[])
```

図 2 オブジェクトを作成する手続きのインタフェース
Fig. 2 The interface of the object creation procedure.

```
stat_t read( in oid_t file, in int offset,
  in int icount, out char buf[],
  out int ocount );
```

図 3 ファイル・オブジェクトの内容を読み込む手続きのインタフェース
Fig. 3 The interface of the read procedure for file objects.

きないときには，クライアントにエラーを返す☆．そして，生成したオブジェクトの識別子を，newobjs に格納してクライアントに返す．

汎用のフィルタ・ファイル・オブジェクトの場合，どのような種類のフィルタを提供するかを指定する必要がある．これは，create() の引数 options により指定される．option_t は，文字列の配列である．一度設定されたオプションは，object_setopt() 手続きにより変更することもできる．また，object_getopt() 手続きにより，現在指定されているオプションを調べることもできる．

たとえば，図 1 のオブジェクトは，次のような引数で作られたものである．

(a1) lowers,options は空（配列の長さが 0）．

(b1) lowers[0] は，**(a1)** のオブジェクト識別子．options は，"tr"，"A-Z"，"a-z" ．

(c1) lowers[0] は，**(b1)** のオブジェクト識別子．options は，"sort"．

図 3 にインタフェースを示した手続き read() は，ファイルの内容を読み込むための手続きである．これは，オブジェクト識別子 file で指定されたファイルの offset バイト目から icount バイト分の内容を読み込むことを意味する．結果は，buf に返される． ocount は，実祭に読み込んだデータのバイト数である．

付録に，create() と read() を含む，オブジェクトの堆積モデルに基づくファイル・システムで用いられている RPC の手続きの一覧を示す．

3.3 フィルタの単位と性質

UNIX 上のフィルタ型コマンドは，その単位により次の 2 種類に分類される．

バイト単位 1 バイト入力すると 1 バイト出力することができるもの．入力と出力のバイト数が同じ．たとえば，2 章で述べた大文字を小文字に変換するフィルタが例としてあげられる．

バイト単位以外のもの ファイル単位や行単位で働くフィルタ．ファイル全体，または，一部（複数バイト）を読み込まないと出力することができない．ファイル単位の例としては，2 章で述べたソート，行単位の例としては，UNIX でよく使われる sed（stream editor，行単位の置き換え）や grep（パタンにマッチした行の出力）があげられる．

堆積可能フィルタ・ファイル・オブジェクトを実現するうえでは，バイト単位とそれ以外では，大きな差がある．その理由は，バイト単位では，1 回の read() 手続きの処理で，動作を完結させることができるのに対して，バイト単位以外のものでは，それができないからである．バイト単位では，read() 手続きを受け付けると，下位層のオブジェクトの同一の場所（offset と icount）を読み込めばよい．これに対して，バイト単位以外のものでは，単純に場所を決定することができない．

本論文では，バイト単位以外のフィルタを扱うことができる堆積可能フィルタ・ファイル・オブジェクトの実現方式を示す．このために，キャッシングの技術を用いる．このことについては，3.4 節で詳しく述べる．

フィルタ処理は，逆関数があるものとないものに分類される．逆関数があるものの例としては，圧縮と伸長，暗号化と復号化，バイナリ・ファイルのテキスト化（uuencode，base64 など）とその逆操作がある．UNIX のフィルタ型コマンドは，ほとんどが逆関数がないものである．

本論文で述べるフィルタ・ファイル・オブジェクトは，逆関数を持たないフィルタを対象とする．このことは，フィルタ・ファイル・オブジェクトは，読み込み専用になるので，書き込みを行う手続きが呼ばれたときには単にエラーを返せばよいことを意味する．ただし，自分自身は読み込み専用でも，下位層のオブジェクトの内容は変更される可能性がある．その場合は，自分自身の内容も変更しなければならない．

3.4 外部のフィルタ型コマンドとキャッシュを用いた実現

この節では，フィルタ処理を行う外部のフィルタ型コマンドとキャッシュを利用して堆積可能フィルタ・ファイル・オブジェクトを実現する方式について述べる．基本的な考え方は，フィルタ処理を外部のコマンドを呼び出して行い，その結果を別の基底ファイル・オブジェクトに保存し，それをキャッシュとして使う

☆ 堆積可能汎用フィルタ・ファイル・オブジェクトでは，3.4 節で述べるように create() の処理の中でキャッシュを作成する．この過程において，下位層のオブジェクトが手続き read() を受け付けることができるかどうかが検査される．

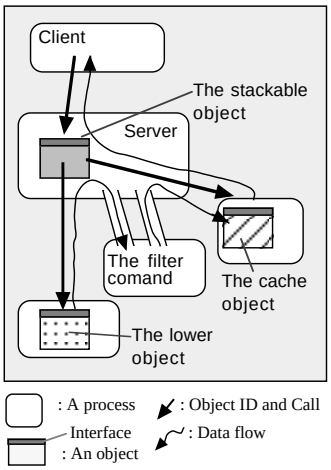

図 4 外部のフィルタ型コマンドによるキャッシュの作成（キャッシュ無効時）
Fig. 4 The creation of the cache by the external filter command (cache is invalid).

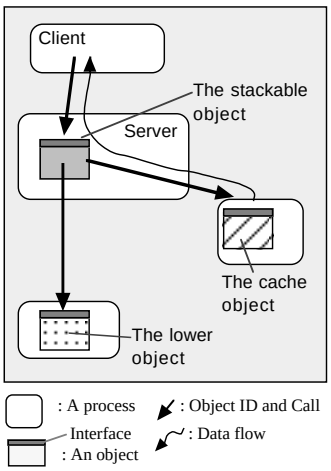

図 5 キャッシュの利用（キャッシュ有効時）
Fig. 5 The use of the cache (cache is valid).

ということである．

図 4 は，あるクライアントが，ある堆積可能フィルタ・ファイル・オブジェクトに対して読み込み要求を送ったときに，**キャッシュ用ファイル・オブジェクト**の内容が無効だったときの処理の様子を表している．**無効（invalid）**とは，以前にキャッシュ用ファイル・オブジェクトの内容（以後，単にキャッシュと呼ぶこともある）を作成した後で下位層のオブジェクトが変更された状況を意味する．逆に，**有効（valid）**とは，以前にキャッシュを作成した後で下位層のオブジェクトが変更されていない状況を意味する．このようにキャッシュが無効のときには，下位層のオブジェクトの内容

```
stat_t read( in oid_t file,
    in int offset, in int icount,
    out char buf[], out int ocount )
{
    object = fetch( file );
    if( !is_cache_valid(object) )
        cache_renew(object);
    status = read(object->cache, offset,
        icount, buf, ocount ); /* RPC */
    free( object );
    return( status );
}

cache_renew( object_t object )
{
    if( exists(object->cache) )
        delete( object->cache );
    object->cache = file_create(
        base_file_server_for_caching );
    filter_process_create( object->command,
        &pipe_output, &pipe_input );
    lwp_create( lower_to_pipe,
        object->lower, pipe_output );
    pipe_to_cache( pipe_input,
        object->cache );
}
```

図 6 堆積可能フィルタ・ファイル・オブジェクトで用いられているアルゴリズム
Fig. 6 The algorithms that are used in the stackable filter file object.

が読み込まれ，外部のフィルタ型コマンドに与えられている．外部のフィルタ型コマンドの出力は，キャッシュ用ファイル・オブジェクトに書き込まれている．最後に，最初に受け付けられた読み込み要求がそのままキャッシュ用ファイル・オブジェクトに転送され，結果がクライアントに返されている．

図 5 は，キャッシュ用ファイル・オブジェクトの内容が有効だったときの処理の様子を示している．下位層のオブジェクトの内容は読み込まれず，キャッシュ用ファイル・オブジェクトに要求が転送され，結果がクライアントに返されている．

手続き `read()` のアルゴリズムを，C 言語に似た構文を使って**図 6** に示す．`read()` は，RPC のサーバ側スタブから最初に呼び出される手続きである．`fetch()` は，オブジェクト識別子を元に，永続記憶中のオブジェクト（構造体）を検索し，メモリ中にオ

ブジェクトを割り当てる手続きである．次に，手続き is_cache_valid() を呼び出し，キャッシュの一貫性を確認する（is_cache_valid() については，3.5 節で述べる）．キャッシュが古くなっていた場合，以下で説明する手続き cache_renew() を呼び出し更新する．最終的には，キャッシュ用ファイル・オブジェクト（object->cache）に，RPCで自分が受け付けた要求をそのまま伝えている．そして，キャッシュ用ファイル・オブジェクトから返された結果をそのままクライアントに返す．free() は，メモリ中のオブジェクトを解放する手続きである．ただし，オブジェクトは，実際にはすぐに解放されず，参照カウンタが減らされるだけでそのままメモリ中に残される．そして，次に fetch() が呼ばれたときに再利用される．

cache_renew() は，キャッシュの内容を更新する手続きである．まず，キャッシュ用ファイル・オブジェクトが存在する場合，手続き delete() によりそれを削除する．そして，新しく空のファイル・オブジェクトを基底ファイル・サーバに作成する．そして，手続き filter_process_create() を呼び出し，フィルタ型コマンドのプロセスを生成する．このとき，このプロセスとサーバ・プロセスは，図 4 に示したように，2 本のパイプ pipe_input と pipe_output により結び付けられる．object->command は，このオブジェクトが 3.2 節で述べた手続き create() の引数 options で指定されたものから作成したものである．この結果として，pipe_output にデータを書き込むと，作成されたプロセスによりフィルタ処理が行われ，結果が pipe_input を読み込むことにより得られる．

3.3 節で述べたように，このオブジェクトは，バイト単位以外のフィルタも扱わなければならない．よって，pipe_output への出力と pipe_input からの入力は，同期しないものとして扱わなければならない．これらの 2 つの処理を独立させて行うために，cache_renew() は，2 つの軽量プロセス（スレッド）を用いている．1 つは，lwp_creat() により生成したもので，手続き lower_to_pipe() を実行し，下位層のオブジェクト object->lower の内容を読み込み，パイプ pipe_output へと出力する．もう 1 つの軽量プロセスは，自分自身（RPC の要求ごとに作られた軽量プロセス）であり，手続き pipe_to_cache() を実行する．これは，フィルタ・プロセスから送られてきた結果をパイプ pipe_input から読み込み，キャッシュ用のファイル・オブジェクト object->cache に書き込んでいる．

なお，RPC で create() 手続きが呼ばれて新しくオブジェクトを生成するときにもキャッシュが作成される．このとき，引数 options で与えられたフィルタ型コマンドがエラーを起こさないかどうか，さらに，下位層のオブジェクトが手続き read() を受け付けることができるかどうかが検査される．エラーが起きた場合，手続き create() は，作成中のオブジェクトを破棄し，クライアントにエラーを返す．

3.5 キャッシュの一貫性の制御

3.4 節では，読み込み要求を受け付けるたびに，is_cache_valid() という手続きを用いてキャッシュの一貫性を確認していると述べた．この手続きは，基本的には，キャッシュの作成を開始した時点での下位層のオブジェクトの最終更新時刻と現在の下位層のオブジェクトの最終更新時刻を比較し，両者が等しいときにはキャッシュの一貫性が保たれていると判断する．ここで注意することは，「キャッシュ用ファイル・オブジェクト」の最終更新時刻ではなく，キャッシュを更新する処理を開始した時点での「下位層のオブジェクト」の最終更新時刻が使われることである．キャッシュ用ファイル・オブジェクトの最終更新時刻は，最後にフィルタ処理の結果を書き込んだ時刻に変化する．たとえば，キャッシュ用オブジェクトへの最後の書き込みの前に下位層のオブジェクトに対して書き込みが行われた場合，キャッシュが無効になったと判断すべきである．しかし，単純に「キャッシュ用ファイル・オブジェクト」の最終更新時刻と「下位層のオブジェクト」の最終更新時刻を比較したならば，前者が新しくなっているので，キャッシュが有効であると誤った判断をしてしまうことになる．キャッシュの最終更新時刻として，キャッシュ用オブジェクトの最終更新時刻ではなく，キャッシュの作成を開始した時点での下位層のオブジェクトの最終更新時刻を用いることで，キャッシュが無効であるかどうかを正しく判断することができる．

なお，キャッシュの作成を開始した時点での下位層のオブジェクトの最終更新時刻は，そのフィルタ・ファイル・オブジェクト自身の最終更新時刻としても使われる．すなわち，クライアント（上位層のフィルタ・ファイル・オブジェクトも含む）から最終更新時刻を問い合わせられたときには，この時刻を返す．フィルタ・ファイル・オブジェクトの層が作られている場合，結果として層に含まれるすべてのオブジェクトの最終更新時刻は，基底ファイル・オブジェクトの最終更新時刻と等しくなる．

堆積可能フィルタ・ファイル・オブジェクトにおいて，強いキャッシュの一貫性を実現するためには，手

続き read() を実行している間，下位層のオブジェクトが変更されないようにロックする必要がある．しかしながら，RPC による 1 ブロックの読み込み要求ごとに，ロック・アンロックを行うことは，オーバヘッドが大きい．この節では，集中型システムである UNIX で利用されているのと同じ程度の一貫性を，RPC という結合が作られない通信プリミティブを用いて実現する方法について述べる．

UNIX のフィルタ型コマンドは，元データを保存しているファイルのロックを行っていない．したがって，処理の途中で元データを保存しているファイルが更新されたときには，古いデータに対するフィルタ処理の結果と新しいデータに対するフィルタ処理の結果が混在して出力されることになる．このような状況は，1 人の利用者が 1 つのシェルを通じて逐次的にコマンドを実行している限り生じない．本フィルタ・ファイル・オブジェクトも，これとほぼ同じレベルの一貫性を保証することを目的とする．

オブジェクトの堆積では，上位層のオブジェクトが下位層のオブジェクトの識別子を保持するが，逆に，下位層のオブジェクトが上位層のオブジェクトの識別子を保持することはない．よって，下位層のオブジェクトが更新されたとしても，そのことを上位層のオブジェクトに通知することはできない．このような環境で，上で述べたレベルの一貫性を保証するために，次のような方法を用いる．

(1) RPC で要求を受け付けたときにのみキャッシュの一貫性を調べる．下位層のオブジェクトが更新されたとしても，要求を受け付けない限り，キャッシュの内容を更新しない．
(2) 下位層のオブジェクトへ RPC を行い最終更新時刻を得て一貫性を確認したら，その**確認した時刻**を記録する．
(3) 再び一貫性を確認する場合，前回に確認した時刻から一定時間内であれば，キャッシュの一貫性は保たれていると見なし，RPC を行わない．

この結果，キャッシュが有効な状況では，キャッシュの一貫性を確認するための RPC の回数は，受け付けた読み込み要求の回数にはよらず，(3) で定めた時間によってのみ決定される．さらにオブジェクトの層ができていたとしても，キャッシュの一貫性を確認するための RPC の回数は，受け付けた読み込み要求の回数にはよらず，層の数に比例した回数となる．

ここでは 1 人の利用者が，すでに作成されている下位層のオブジェクト（書き込み可能な基底オブジェクト）とフィルタ・ファイル・オブジェクトに対して次のような操作を逐次的に行った場合を例として，キャッシュの一貫性を調べる間隔の影響を考えてみる．

(1) フィルタ・ファイル・オブジェクトの内容を読み込むコマンドを実行する．
(2) 下位層のオブジェクトを更新するコマンドを実行する．
(3) 再びフィルタ・ファイル・オブジェクトの内容を読み込むコマンドを実行する．

ここで，コマンドの実行にはエディタによる読み書きも含む．

この状況では，まず，(1) のコマンドの実行中にキャッシュの一貫性が調べらる．すなわち，下位層のオブジェクトの最終更新時刻が調べられる（必要ならば，キャッシュが更新される）．(3) のコマンドを実行したときに，(2) で行われた更新が反映された結果を出力することができるかどうかは，(1) のコマンドの実行中に行われたキャッシュの一貫性を検査した時刻と (3) のコマンドの実行を開始したときの時刻の差が問題となる．これがある時間内の場合，フィルタ・ファイル・オブジェクトはキャッシュの一貫性が保たれていると見なし，古い内容を返してしまう．一方，この差がある時間よりも大きい場合，再び下位層のオブジェクトの最終更新時刻を調べる．その結果，キャッシュが古くなっていることを検知し，キャッシュを更新し，新しい内容を返すことができる．

この一貫性を確認する間隔は，利用可能な計算機資源と利用者からの要求により定められる．現在は，この間隔を，1 秒としている．単一の利用者が対話的に利用しているときには，1 秒以内に複数のコマンドを打ち込むことは，非常に難しいと思われる．上の例では，(1) と (3) の差が 1 秒以上になると思われる．また，この数字は，4.1 節で用いた実験環境における RPC の往復時間と比べて，1000 倍以上大きい．

3.6 フィルタ・ファイル・オブジェクトの実現方式の評価

この節では，この章で述べたフィルタ・ファイル・オブジェクトの実現方式について，性能に関連しない評価を行う．性能に関連する考察は，4.5 節で行う．

この章では，パイプを入出力とする外部のフィルタ・プログラムとキャッシングの技術を利用するフィルタ処理の実現方式について述べた．この他にも，フィルタ処理を実現する方式としては，次のようなものがある．

(1) サーバ内部の手続きとして実現する[4),5)]．
(2) サーバ外部の RPC の手続きとして実現する[13)]．
(3) パイプを入出力とするフィルタ・プログラムを

利用するが，キャッシングを行わない[5),13)]．

本実現方式の目的は，1章で述べたように，オブジェクトの堆積モデルに基づくファイル・システムにおいて有用なフィルタ・ファイル・オブジェクトをより少ない開発コストと実行時の資源量で実現することである．以下では，このような観点から，本実現方式とこれらの実現方式との比較を行い，本実現方式を評価する．

(1)の方式では，プロセスの生成やプロセス間通信のオーバヘッドがなく，より高速な実現が可能である．特にバイト単位のフィルタならば，キャッシングを省略することで高速化が可能な場合もある．しかしながら，利用可能なフィルタの種類がサーバの内部に定義された手続きに限定されるという問題がある．よって，少数の利用頻度が高いフィルタの実現に適している．この方式と比較して，本方式の特徴は，プロセス生成やプロセス間通信のオーバヘッドがあるが，利用者定義のフィルタを利用できる点にある．よって，多種多様なフィルタを実現する必要があるときに適している．

(2)の方式では，フィルタを，通常の手続き呼び出しではなくRPCによりアクセスされる手続きにより実現する．(2)の方式では，パイプを使う方式と同様にプロセス間通信のオーバヘッドは存在するが，RPCのサーバが常駐しているので，フィルタ処理を行うプロセスを生成するオーバヘッドはない．この方式は，プロセスの起動に時間がかかる処理，たとえば，起動時に大きな辞書を読み込むようなフィルタ処理の応答時間を短縮したいときに適している[13)]．ただし，RPCのサーバが常駐するので，実行時のメモリ使用量は大きくなる．この方式と比較して本方式の特徴は，プロセス生成のオーバヘッドがあるが，フィルタ処理を行うプロセスを常駐させる必要がなく，利用者定義のフィルタを実現できる点にある．よって，多種多様なフィルタをより少ない資源で実現する必要があるときに適している．

(3)の方式は，フィルタを，パイプを入出力とするフィルタ・プログラムを利用して実現するが，キャッシングを行わない方式である．これは，HTTPによる1単位の資源の転送のように，データが先頭から末尾へと逐次的にアクセスされ，同一のデータは，ただ1度だけしかアクセスされないようなときに適している[13)]．この性質は，2.1節で議論したパイプの制約と一致している．またHTTPでは，キャッシングのみを行う堆積可能オブジェクトもすでに実現されているので，これと組み合わせて利用することも可能である．この方式と比較して本方式の特徴は，ランダム・アクセスに簡単に対応することができるようになっている点にある．もしキャッシングを行わないならば，要求されたデータを返すために，下位層のオブジェクトのどの位置を読み込めばよいかを計算しなければならない．しかし，3.3節で述べたように，バイト単位以外のフィルタでは，単純に場所を決定することができない．したがって，バイト単位以外フィルタにも対応するためには，すべてのデータを読み込み，フィルタ処理を行い，必要なデータが得られるまで読み捨てることになる．これに対して本実現方式では，3.4節で述べたように，受け付けた要求をそのままキャッシュ用ファイル・オブジェクトに転送しているだけである．このように，実現が非常に簡単になっている．

なお，キャッシングは，バイト単位以外のフィルタの実現を容易にするだけでなく，性能向上にも大きく寄与する．性能に関しては，4.5節で述べる．

3.5節では，ある一定時間キャッシュが有効であると仮定しチェックを行わない方法について述べた．このような方法は，NFSにおいても用いられている[14)]．NFSは，Sun Microsystems社によって開発されたネットワーク・ファイル・システムである．NFSでは，属性キャッシュの最小保持時間（標準では3秒）の間は，キャッシュを保持する．NFSの方法と比較して，本実現方式の特徴は，オブジェクトの層ができる点，および，キャッシュの内容が単純なコピーではなく，フィルタ処理を施した結果である点にある．

3.5節で論じたように，本フィルタ・ファイル・オブジェクトの利用者は，下位層のオブジェクトの更新について注意を払う必要がある．本フィルタ・ファイル・オブジェクトは，たとえば下位層のデータが更新されている途中にアクセスされると，古いデータに対するフィルタ処理の結果と新しいデータに対するフィルタ処理の結果を混在して返すことがある．したがって，元データの更新にどのくらいの時間がかかるか，どの程度の頻度で更新されるか，元データが更新されている最中にフィルタ処理の結果が必要になることがあるのか，新旧のデータが混在した結果が返されれることが許容されるかといったことを考慮する必要がある．なお，UNIXのフィルタ型コマンドにおいても基本的に同じ問題がある．

3.5節で述べたように，本フィルタ・ファイル・オブジェクトを用いて最新の更新内容が反映された結果を得るためには，更新が完了してから一定時間待つ必要はないが，前回アクセスした時刻から一定時間待つ必要がある．これは，オブジェクトの堆積モデルの1つの限界を示している．この問題を解決する方法の1つとして，オブジェクトの連結モデルを使う方法があ

る．オブジェクトの連結モデルについては，5.2 節で述べる．

この章で述べた実現方式には，パイプと比較して，次のような制限がある．

- 無限の長さのデータを生成するような基底オブジェクトを扱うことができない．
- キャッシュが無効だったとき，ファイル全体について処理を完了するまでデータが出力されないので，応答時間が長くなる．

後者の応答時間が長くなってしまうという性質については，4.5 節において述べる．

現在の実装では，3.4 節で述べたように，新しくオブジェクトを生成する段階でフィルタ型コマンドがエラーを起こさないかを調べている．この方法では，コマンドの引数のエラーなどを未然に防ぐことはできる．しかし，下位層のオブジェクトが変更されたことにともない発生するエラーには対応することができない．現在の実装では，標準エラー出力に出力されたエラー・メッセージを破棄しているが，これをファイルの形態で返すことも可能である．それには，3.2 節で述べたオブジェクトを生成する手続き `create()` の出力パラメータ `newobjs` を使う．現在は，その最初の要素しか利用していないが，2 番目以降の要素を利用することで，エラー・メッセージをファイルの形で返すことも可能である．

4. 性　　能

3 章で述べた方式に従って堆積可能汎用フィルタ・ファイル・オブジェクトを実現した．この章では，実現したオブジェクトの性能を示す．

本実現方式の第 1 の目的は，1 章で述べたように，多種多様なフィルタ・ファイル・オブジェクトをより少ない実現のコストと実行時の資源で実現することである．そのために，外部のプログラムとキャッシングの技術を用いている．本方式に基づいて実現したフィルタ・ファイル・オブジェクトの性能は，その外部のプログラムとキャッシュの状態によって大きく変化することが予想される．すなわち，キャッシュが有効なときには非常に高速で，キュッシュが無効のときには，それを更新するために外部コマンドを実行するので遅くなるという性質が予想される．キャッシュが無効のときの性能は，利用者によって与えられた外部プログラムに大きく依存するが，その他に，外部プログラムに依存せず，実現方式に深くかかわり性能を左右するものがある．それは，堆積されるフィルタ・ファイル・オブジェクトの数と下位層のファイル・オブジェクトの大きさである．この章では，これらの，外部プログラムの性質によらず，実現方式に深く関連している性質について焦点を当て，性能を測定する．これらの結果は，フィルタ・ファイル・オブジェクトを利用する際の指針を与える．

本実現方式では，3.5 節で述べたように対話的な環境で利用することを想定してキャッシュの一貫性を確認する間隔を決定している．対話的な利用では，ファイル全体を読み出すために要する実行時間に加えて応答時間が重要となる．応答時間とは，利用者が要求を与えてから最初の結果が現れるまでの時間である．以下の実験では，次のような方法を用いて実行時間と応答時間を測定する．

実行時間　ファイルの内容をすべて読み出すのに要する時間を測定する

応答時間　最初の 1 ブロック（8 k バイト）を読み込むまでの時間を測定する

本実現方式では，読み込み要求を受け付けたときには，必ずキャッシュ用ファイル・オブジェクトが作られる．その内容は，下位層のオブジェクトが更新されたときに無効になる．無効になった内容は，更新されなければならないが，更新作業は，下位層オブジェクトが更新された時点で開始されるのではなく，フィルタ・ファイル・オブジェクト自身が読み込み要求を受け付けた時点で開始される．したがって，キャッシュが無効な状況でアクセスされたときには，実行時間や応答時間が長くなることが予想される．この性質を確かめるために，以下の実験では，次のような方法を用いてキャッシュが有効な状況と無効な状況を作り出した．

キャッシュが有効な状況　フィルタ・ファイル・オブジェクトの内容を繰返し読み込む．

キャッシュが無効な状況　最下位層のオブジェクト（基底ファイル・オブジェクト）の最終更新時刻を進める．

4.1 実 行 環 境

実験を行った環境は，Sun ワークステーション（CPU UltraSPARC II 300 MHz，主記憶 128 MB）である．オペレーティング・システムは，SunOS 5.5 である．実験で用いたサーバとクライアントは，このオペレーティング・システム上で動作するプロセスとして実現されている．

実験では，ネットワーク通信の影響を排除して堆積可能フィルタ・ファイル・オブジェクトの性能を調べるために，すべての基底ファイル・オブジェクト，堆積可能オブジェクト，および，クライアントを同一のワークステーション上で動作させた．キャッシュと元

表 1 実験環境における RPC の往復時間時間
Table 1 The measured round trip times of the RPC in the experimental environment.

arguments (bytes)	results (bytes)	times (m sec)
0	0	0.17
0	8192	0.36

データを保存するための基底ファイル・オブジェクトは，同一のサーバ内に生成した．

これらのプロセスの間の通信は，UDP 上の Sun-RPC 3.9 により行った．この環境における SunRPC の基本的な通信性能を，**表 1** に示す．この時間は，単純な RPC を 100 回から 1000 回繰り返し，全体の時間を測定し，繰返し回数で割ったものである．

4.2 基底ファイル・オブジェクト

実験では，最下位層のファイル・オブジェクト，および，キャッシュ用ファイル・オブジェクトとして基底ファイル・オブジェクトが必要となる．今回の実験では，ディスク入出力処理のオーバヘッドを排除するために，主記憶にデータを保存する基底ファイル・オブジェクトを用いた．このオブジェクトは，書き込み要求を受け付けると，その内容を主記憶に保存する．読み込み要求を受け付けると，対応するファイルの内容を主記憶から読み出す．

4.3 軽量プロセス・ライブラリで用いた仮想プロセッサの数

実験で用いた軽量プロセス・ライブラリでは，カーネル・レベルの仮想プロセッサを，通常のプロセスを用いてエミュレートしている．仮想プロセッサの数は，単一プロセッサ上では，同時に実行可能な入出力（RPC によるプロセス間通信を含む）を表す．エミュレーションの問題により，仮想プロセッサの数が多くなると，ファイルを開いたり閉じたりする動作（パイプの作成を含む）のオーバヘッドが大きくなる．

今回の実験では，堆積可能フィルタ・ファイル・オブジェクトのサーバでは，仮想プロセッサ数を 10 に固定した．このとき，パイプの生成のオーバヘッドが，1 本あたり 1.1 ミリ秒，フィルタ・プロセスの生成（パイプ 2 本を含む）のオーバヘッドが，3.3 ミリ秒であった．基底ファイル・オブジェクトのサーバと，クライアントは，今回の実験では内部に並列性を持たないので，それぞれの仮想プロセッサ数を 1 に固定した．

4.4 結　　果

4.4.1 文字変換・ソート，および，圧縮・テキスト化

2 章の図 1 に示したように堆積可能フィルタ・ファイル・オブジェクトを生成し，性能を測定した．結果を，**表 2** に示す．この表で，各オブジェクトは，次のような意味を持つ．

表 2 堆積可能フィルタ・ファイル・オブジェクトの応答時間と実行時間（文字変換，ソート，圧縮，テキスト化）
Table 2 The measured execution and response times of the stackable filter file objects (character translation, sorting, compression, and text encoding).

object	size	cache valid		cache invalid	
	(KB)	resp	exec	resp	exec
base1	80	0.8	5.3	-	-
tr/base1	80	1.7	11	55	65
sort/tr/base1	80	6	16	250	260
base2	380	0.8	24	-	-
Z/base2	17	1.7	4.0	124	127
uue/Z/base2	23	2.5	5.9	176	179

resp: response time, exec: execution time, times are in milliseconds.

base1 800 人分のパスワードを含む基底ファイル・オブジェクト．

tr 大文字を小文字に変換する堆積可能フィルタ・ファイル・オブジェクト．生成時のオプションは，`tr A-Z a-z`．

sort ソートを行う堆積可能フィルタ・ファイル・オブジェクト．生成時のオプションは，`sort -t: +4`．

base2 ウィンドウの画面のイメージを含む基底ファイル・オブジェクト．

Z データを圧縮する堆積可能フィルタ・ファイル・オブジェクト．生成時のオプションは，`compress`．

uue データをテキストに符号化する堆積可能フィルタ・ファイル・オブジェクト．生成時のオプションは，`uuencode filename`．

表 2 で，tr/base1 は，基底ファイル・オブジェクト base1 の上に文字変換を行うオブジェクトを積み重ねたもの，sort/tr/base は，さらにソートするオブジェクトを積み重ねたものである．Z/base2，uue/Z/base2 も同様である．

表 2 で，キャッシュが有効（cache valid）のときには，いずれの場合も応答時間（resp）も実行時間（exec）も十分短い．この数字は，現在広く使われているディスクの平均シーク時間（8 ミリ秒程度）よりも短いので，キャッシュをディスクに保存した場合，ディスク入出力で実行時間が決定されると思われる．また，堆積可能オブジェクト Z/base2 は，基底オブジェクト base2 よりも実行時間が短くなっている．これは，フィルタ処理によりデータ量が小さくなったことによる．

表 2 で，キャッシュが無効（cache invalid）のとき，それぞれのフィルタ処理のため，応答時間（resp）と実行時間（exec）の両方が大きくなっているが，それ

らの差は小さい．これは，最初にアクセスされたときにキャッシュを作るために時間がかかるが，一度キャッシュが作成された後は，それを利用しているので時間がかからないということを示している．

4.4.2 堆積可能フィルタ・ファイル・オブジェクトの数の影響

図7は，堆積可能フィルタ・ファイル・オブジェクトの数を増やしたときのオーバヘッドを調べる実験の結果である．横軸は，堆積可能フィルタ・ファイル・オブジェクトの数である．縦軸は，ファイルを読み込む簡単なクライアントの応答時間，または，実行時間である．この実験では，フィルタ処理の影響を最小限にとどめ，堆積可能オブジェクトの性能を見るために，簡単なフィルタ型コマンド`/bin/cat`を用いた．`/bin/cat`は，入力をそのまま出力するフィルタである．ファイルの大きさは，基底ファイル・オブジェクトも各堆積可能フィルタ・ファイル・オブジェクトも80kバイトである．

図7では，キャッシュが有効（valid）のときには，応答時間（response）も実行時間（exec）も，堆積可能フィルタ・ファイル・オブジェクトの数にかかわりなくほぼ一定であることが分かる．また，キャッシュが無効（invalid）のときと比べてかなり短くなっている．このことから，キャッシュの効果が非常に大きいことが分かる．

図7でキャッシュが無効のときには，応答時間も実行時間もともに1層あたり60ミリ秒大きくなっていることが分かる．この数字は，ファイルの大きさが80kバイトのときの1層あたりのオーバヘッドである．

4.4.3 ファイルの大きさの影響

図8は，堆積可能フィルタ・ファイル・オブジェクトの数を2に固定し，ファイルの大きさを変化させたときのオーバヘッドを調べる実験の結果である．堆積可能フィルタ・ファイル・オブジェクトとしては，4.4.2項と同様に，簡単なフィルタ型コマンド`/bin/cat`より作成したものを用いた．すなわち，各々`/bin/cat`を1度だけ実行するような堆積可能フィルタ・ファイル・オブジェクトを2個作成し，基底オブジェクトの上に積み重ねた．図8の横軸は，ファイル・オブジェクトの大きさである．縦軸は，ファイルを読み込む簡単なクライアントの応答時間，または，実行時間である．実験では，基底ファイル・オブジェクトの大きさを，80kバイトから400kバイトまで変化させた．`/bin/cat`は，入力をそのまま出力するので，堆積可能フィルタ・ファイル・オブジェクトの大きさも同じである．

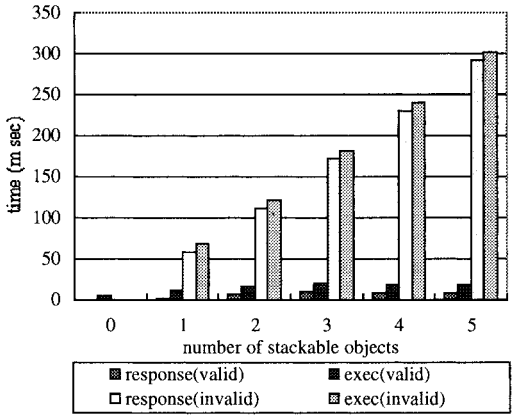

図7 堆積可能フィルタ・ファイル・オブジェクトの数を変化させたときの応答時間と実行時間
Fig. 7 The response and execution times according to the number of stackable filter file objects.

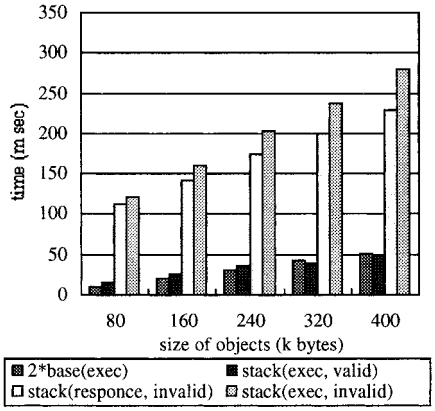

図8 ファイルの大きさを変化させたときの応答時間と実行時間
Fig. 8 The response and execution times according to the size of file objects.

図8では，比較のために基底ファイル・オブジェクトの実行時間を2倍したもの2*base(exec)をともに示している．基底オブジェクトの実行時間は，4.2節で述べたサーバのメモリからクライアントのメモリへRPCを経由してデータをコピーするために要する時間を意味する．その時間を2倍したものが2*base(exec)である．すなわち，2*base(exec)とは，横軸に示されたデータ量の2倍のデータをRPCでコピーするために要する時間である．

図8において，堆積可能フィルタ・ファイル・オブジェクトの実行時間stack(exec,valid)には，次の2つのRPCによるデータのコピーに要する時間が含まれている（図5参照）．

- キャッシュ用ファイル・オブジェクトから堆積可能フィルタ・ファイル・オブジェクトへのコピー
- 堆積可能フィルタ・ファイル・オブジェクトからクライアントへのコピー

2*base(exec) と stack(exec,valid) を比較すると，ほぼ等しいことが分かる☆．ゆえに，stack(exec,valid) のほとんどが RPC によるデータのコピーに費やされていることを意味する．

図 8 において，キャッシュが無効のときの堆積可能フィルタ・ファイル・オブジェクトの応答時間 stack(response,invalid) は，ファイルが大きくなるに従って大きくなることが分かる．これは，80 k バイトあたり約 30 ミリ秒の割合で大きくなっていることが分かる．この数字は，層の数が 2 のときの，80 k バイトあたりのオーバヘッドである．キャッシュが無効のときの堆積可能フィルタ・ファイル・オブジェクトの実行時間 stack(exec,invalid) は，応答時間 stack(response,invalid) に，キャッシュが有効時の実行時間 stack(exec,valid) を足したものとほぼ等しくなっている．このことは，3.4 節で述べたアルゴリズムの性質がよく現れている．

4.5 性能に関する考察

本実現方式の第 1 の目的は，1 章で述べたように，多種多様なフィルタ・ファイル・オブジェクトをより少ない実現のコストと実行時の資源で実現することである．そのために，外部のプログラムとキャッシングの技術を用いている．3.6 節では，性能以外の観点から本フィルタ・ファイル・オブジェクトの実現方式の評価を行った．この節では，このような目的のために用いた外部プログラムとキャッシングが，どのように性能面に影響を与えたかを考察する．そして，フィルタ・ファイル・オブジェクトの利用者に対して指針を与える．

4.4 節で示した実験結果より，キャッシュが有効な場合は，本フィルタ・ファイル・オブジェクトの 1 ブロックあたりの性能は，ディスクの平均シーク時間よりも短い．よって，キャッシュをディスクに保存した場合，ディスク入出力で処理時間が決定されると思われる．2.1 節では，堆積可能フィルタ・ファイル・オブジェクトに適した応用として，何度も参照されるデータを生成するものをあげた．そのような応用では，2 回目以降のアクセスからキャッシュが使われることになる．4.4 節の実験結果は，本方式に基づいて実現したフィルタ・ファイル・オブジェクトについても，その性質が提供されていること示している．

4.4.1 項で示したように，キャッシュが無効のときのフィルタ・ファイル・オブジェクトの性能は，フィルタ処理に大きく依存している．したがって，本フィルタ・ファイル・オブジェクトの利用者は，まず，その目的のフィルタ処理に要する時間に注意を払う必要がある．4.4.2 項で示したように，複数のフィルタ・ファイル・オブジェクトを積み重ねて使うとキャッシュが無効なときには応答時間や実行時間が加算される．また，4.4.3 項で示したように，元データを保存しているファイルが大きくなったときにも，それにともない応答時間や実行時間が増大する．したがって，本フィルタ・ファイル・オブジェクトの利用者は，フィルタの性質に加えてこれらのフィルタ・ファイル・オブジェクトに依存する性質についても，気をつける必要がある．

本実現方式では，キャッシュが無効な場合，ファイル全体についてフィルタ処理を完了するまでデータが出力されない．したがって，たとえ元のフィルタが短い応答時間を提供する性質，すなわち，データの一部を受け取った段階でデータの一部を出力する性質を持っていたとしても，応答時間が長くなってしまう．4.4 節では，この性質を実験によって確認した．これを改善するには，たとえば，図 6 に示したアルゴリズムを修正し，キャッシュの更新を完全に別の軽量プロセス（スレッド）で行う方法が考えられる．このとき，RPC の要求を処理している軽量プロセスは，キャッシュ更新の進行状況を監視しながら，必要なデータが揃った段階でそれを横からコピーし，キャッシュ更新の完了を待たずにクライアントに返すようにする．

5. 関 連 研 究

5.1 サーバの堆積

オブジェクトの堆積では，オブジェクトを単位として層が作られる．一方，サーバを単位として層を作る方法がある．これを，**サーバの堆積（server-stacking）**とよぶことにする．サーバの堆積モデルに基づくシステムとしては，UCLA Stackable File System がある[4]．このシステムでは，一様なインタフェースとしては，NFS の vnode インタフェースが用いられている[14]．

☆ ファイルの大きさが 320 k バイト以上のところでは，stack(exec,valid) が 2*base(exec) よりも若干短くなっている．これは，フィルタ・ファイル・オブジェクトの中でメモリ中のコピーの回数を 1 回減らす最適化を行っていることによる．フィルタ・ファイル・オブジェクトには，下位層のオブジェクトの最終更新時刻を調べるなどのオーバヘッドがある．コピーするデータ量が少ないときには，そのオーバヘッドのため遅くなっているが，コピーのデータ量が大きなところでは，メモリ中のコピー回数を減らす最適化の効果が現れている．

このシステムで使われているフィルタの実現方式と比較して本実現方式の特徴は，第 1 に，オブジェクトの堆積に基づいており，個々のファイルごとにフィルタの構成を自由に設定することができる点にある．第 2 に，本実現方式では，フィルタとして，UNIX のフィルタ型コマンドや利用者定義のフィルタ型コマンドを利用することができることである．第 3 に，キャッシュの一貫性を維持するために，本実現方式では，分散的な手法が用いられている．UCLA Stackable File System では，キャッシュの一貫性を維持するための集中化されたサーバが存在する．

5.2 オブジェクトの連結

オブジェクトの堆積を拡張し，上下 2 方向にオブジェクトのリンクを作る手法を**オブジェクトの連結（object-cascading）**と呼ぶ[11]．オブジェクトの連結モデルでは，**連結可能オブジェクト（cascadable object）**を利用して，高度な機能を提供する．オブジェクトの連結を使うことで，キャッシュの一貫性の制御をより効率的に行うことが可能になると思われる．たとえば，Spring システムでは，分散ファイル・システムにおいてキャッシュの一貫性を実現するファイル・システムを実現している[7]．Spring システムで使われている技術は，サーバの連結といえる．**サーバの連結（server-cascading）**とは，オブジェクトの堆積に対するサーバの堆積と同様に，サーバの単位で連結が実現されたものである．

Spring システムで使われているフィルタの実現方式と比較して，本実現方式の特徴は，高度な機能を外部のフィルタ型コマンドを利用して実現していること，オブジェクトを単位として層が形成されること，および，キャッシュの一貫性を保つために，3.5 節で述べたような時間を用いる手法を用いている点にある．Spring では，基本的にはサーバを単位として層が形成され，オブジェクトごとの特殊化には，interposition と呼ばれる機能と名前解析の機能を利用している．

5.3 UNIX System V のストリーム

ストリームは，UNIX System V に取り入れられたカーネル内のモジュールを利用するための仕組みである[1]．ストリームでは，利用者プロセスは，付加的な機能を提供するモジュールを**プッシュ**することができる．プッシュされたモジュールは，put と get という単純なインタフェースにより通信する双方向のフィルタとして動く．一度に複数のモジュールをプッシュすることも可能である．

ストリームに基づきプッシュ可能なモジュールは，連結可能オブジェクトの一種と見なすことができる．この方法と比較して，この論文で述べた堆積可能フィルタ・ファイル・オブジェクトの特徴は，複数の層を同時にアクセスすることができる点にある．ストリームにおいてあるモジュールがプッシュされた場合，その下のモジュールを直接アクセスすることはできなくなる．堆積可能フィルタ・ファイル・オブジェクトの場合は，下位層のオブジェクトの識別子を名前サーバに登録するなどしておけば，複数の層を同時に利用することができる．

6. おわりに

本論文では，オブジェクトの堆積モデルに基づきフィルタ機能を提供する堆積可能フィルタ・ファイル・オブジェクトの実現方式について述べた．この実現方式の特徴は，外部のフィルタ型コマンドとキャッシングの技術を用いている点にある．前者により，利用者定義のフィルタ機能が容易に利用可能になり，さらに，単一のサーバで様々な種類のフィルタ機能を提供することができるようになる．後者により，性能が大きく改善されるだけでなく，バイト単位以外のフィルタの実現が簡単になっている．キャッシュの一貫性の実現においては，一度一貫性がとれていることが確認された後は，ある一定時間は一貫性がとれていると仮定することで，RPC の回数を減らしている．キャッシュは，ファイル単位で作られるので，キャッシュが無効のときの応答時間は，層の数が増え，ファイルの大きさが大きくなるに従って長くなる．この性質を実験を通じて確認した．また，キャッシュが有効のときには，RPC の中継のオーバヘッドのために，基底オブジェクトの約 2 倍の時間がかかることが分かった．

本論文では，上位層のオブジェクトが下位層のオブジェクトを呼び出すという制約の下で，キャッシュの一貫性を保証する方法を示した．これは，プロセス間通信におけるクライアント・サーバ・モデルに基づいた，他のシステムでも利用可能であると思われる．今後は，オブジェクトの連結に基づくフィルタ・ファイル・オブジェクトを実現し，比較を行いたい．また，堆積可能オブジェクトにおけるアクセス制御やエラー処理についても研究していきたい．

参考文献

1) Bach, M.J.: *The design of the UNIX operating system*, Prentice-Hall (1986).
2) Golub, D., Dean, R., Forin, A and Rashid, R.: Unix as an Application Program, *Proc. Summer 1990 USENIX Conf.*, pp.87–97 (1990).

3) 萩野, 山岸：ToM マイクロカーネル, 電子情報通信学会論文誌（D-1）, Vol.J75-D-I, No.8, pp.555–562 (1992).
4) Heidemann, J. and Popek, G.: Performance of Cache Coherence in Stackable Filing, *14th SOSP, ACM Operating System Review*, Vol.29, No.5, pp.127–142 (1995).
5) 苅部, 新城, 清木：オブジェクト堆積モデルに基づくファイル・サーバの実現, 情報処理学会研究会報告, 93-OS-58, Vol.93, No.27, pp.9–16 (1993).
6) Kernighan, B.W, and Pike, B.: *The UNIX Programming Environment*, Prentice-Hall (1984).
7) Khalidi, L. and Nelson, M.: Extensible File Systems in Spring, *13th SOSP, ACM Operating System Review*, Vol.27, No.5, pp.1–14 (1993).
8) Levy, E. and Silberschatz, A.: Distributed File Systems: Concepts and Examples, *ACM Computing Surveys*, Vol.22, No.4, pp.321–374 (1990).
9) Mullender, S., Rossum, G., Tanenbaum, A., Renesse, R. and Staveren, H.: Amoeba: A Distributed Operating System for the 1990s, *IEEE Computer*, Vol.23, No.5, pp.44–53 (1990).
10) Shinjo, Y. and Kiyoki, Y.: The Object-Stacking Model for Structuring Object-Based Systems, *Proc. 2nd International Workshop on Object Orientation in Operating Systems* (*I-WOOOS'92*), pp.328–340 (1992).
11) 新城：オブジェクトの堆積・連結モデル, 第 6 回情報処理学会コンピュータシステム・シンポジウム, Vol.94, No.10, pp.31–38 (1994).
12) 新城, 清木：並列プログラムを対象とした軽量プロセスの実現方式, 情報処理学会論文誌, Vol.33, No.1, pp.64–73 (1992).
13) Shinjo, Y.: The World Wide Shell based on the Object-Stacking Model, *Proc. Worldwide Computing and Its Applications '97* (*WWCA97*), LNCS, Vol.1274, pp.410–425, Springer (1997).
14) *SunOS Network Programming*, Sun Microsystems (1992).

付録　オブジェクトの堆積モデルに基づくファイル・システムで用いられている RPC のインタフェース

オブジェクトの堆積モデルに基づくファイル・システムで用いられている RPC の手続きの一覧を示す.

`null()`　なにもしない. サーバの動作確認に使われる.

`create()`　オブジェクトを作成する.

`getattr()`　最終更新時刻などのオブジェクトの属性を返す.

`setattr()`　オブジェクトの属性を設定する.

`getlowers()`　下位層のオブジェクトを返す.

`object_getopt()`　`create()` で設定したオプションを返す.

`object_setopt()`　`create()` で設定したオプションを変更する.

`delete()`　オブジェクトを削除する.

`copy()`　オブジェクトを, 層の構成を保ったまま再帰的にコピーする.

`createlike()`　オブジェクトを, 層の構成を保ったまま再帰的にコピーする. ただし内容はコピーせず, 空のオブジェクトが作られる.

`read()`　ファイル・オブジェクトからデータを読み出す.

`write()`　ファイル・オブジェクトへデータを書き込む.

(平成 11 年 12 月 8 日受付)
(平成 12 年 4 月 6 日採録)

新城　靖（正会員）

1965 年生. 1993 年筑波大学大学院博士課程工学研究科電子・情報工学専攻修了. 同年琉球大学工学部情報工学科助手. 1996 年筑波大学電子・情報工学系講師. 分散型オペレーティング・システム, オブジェクト指向, 分散処理, 並列処理に興味を持つ. 平成 2 年情報処理学会第 40 回全国大会学術奨励賞, 平成 7 年情報処理学会山下記念研究賞受賞. 博士（工学）. ACM, IEEE CS, ソフトウェア科学会各会員.