第 7 部

アプリケーション (ファイルシステム)

第 1 章

序論

1.1 はじめに

今日では、計算機はコンピュータネットワークによって相互に接続され使用されている。このような広域分散環境の中でユーザは、電子メールや電子掲示板、ファイル転送、リモート端末などのアプリケーションを利用することができる。

このようなサービスとは別に、異なる計算機上に分散している資源を共有するための仕組みも研究されてきた。ファイルシステムについては NFS(Network File System)、RFS(Remote File Sharing) などの異機種間共有ファイルシステム、データベースについては NIS(Network Information Service) などが実現され、またこれらの土台となる技術として従来のサーバ・クライアントモデルの発展型である RPC(Remote Procedure Call: 遠隔手続き呼び出し) など、それまでの Unix にはなかった様々な機能・サービスの拡張がなされてきた。

これらの機能・サービスの拡張は常に、異なるホストによる違いを意識しないで利用できるように配慮がされ実装されてはいるが、いくつかの問題点も残されている。

もし使用する計算機が全て同じ機種・アーキテクチャの場合は、同じ設定、同じ構成でシステムを構築することによって、複数のホスト上でまったく同じ使用感を得ることはたやすい。しかし、幾つかの異なる機種・アーキテクチャの計算機をネットワークによって接続し、それぞれのホストの特徴を生かして、使い分けるのが現在の一般的な使用形態であり、そのような環境で様々なホストを使用し、なおかつどのホストでも同じ使用感を得るのは難しいのが現状である。

例えば、あるユーザのホームディレクトリを単一にし、異なる機種・アーキテクチャ間で共有することを考えてみよう。こうすることによってユーザはどのホストからも同じファイル群をアクセスすることができる。このように共有できるファイルを一箇所に格納し管理する方法は一般的である。しかし、ホームディレクトリにある機種・アーキテクチャに依存するようなファイルもすべて同等に扱われてしまい、ユーザはそのようなファイルに対してはその依存性を意識しつつ使用しなくてはならない。当然、弊害が起こることもあり得る。また、このような問題はホームディレクトリの例には限らない。同じ名前が付いたコマンドでも機種・オペレーティングシステムによって異なる機能を有していたり、実行結果がまるで異なるような場合もあり、そういった点での配慮も必要

である。

Unix のファイルシステムがネットワークが発達する以前に設計・実装されており、ネットワークに対応する機能は後からしかもそれまでのセマンティクスをできるだけくずないように設計・実装されてきたことによってこの問題が起こったといえる。

本論文では、このような共有ファイルシステムの問題に対して、ファイルシステムの可視性を制御することによる解決方法を考察し、アーカイブディレクトリという概念を提案する。

1.2 本論文の構成

以下本論文では、第 2 章で共有ファイルシステムの現状として NFS について解析し、論理的ファイル共有について述べる。第 3 章では可視性制御によるファイルシステムの研究について解析し、第 4 章で我々が提案するアーカイブディレクトリの仕様を、第 5 章ではその実装について述べる。第 6 章でその評価を行い、第 7 章で今後の課題について検討し、第 8 章で結論を述べる。

第 2 章

共有ファイルシステム

前章で述べたように現在使用されている共有ファイルシステムにはいくつかの問題点がある。この章では現在の共有ファイルシステムについてその機能・構造の概略を説明した上で、その問題点について再考する。

2.1　現在の共有ファイルシステム

複数のホストに分散したファイルシステムを共有する方法として、スーパールートやリモートマウントなど様々な手法が提案され、実装されてきた。最近ではリモートマウント方式を用いるのが一般的である。特にまた Sun microsystems, Inc. が開発した分散ファイルシステムである NFS は、このリモートマウント方式を用いており、多くのシステムで実装され、稼働している。

2.1.1　リモートマウント

リモートマウントとは、NFS や RFS などに導入されている概念で、Unix のファイルシステムの特徴の一つである論理ボリュームとマウントの概念を拡張したものである。具体的には、他の計算機上にあるファイルシステムの一部 (部分木) を自分のファイルシステムの枝にマウントすることによって、あかたも自分のファイルシステムの一部であるように見せることができるものである (図 2.1参照)。マウントされた他のホストのファイルシステム上のファイルは、それがどのホストのものであるか意識せず参照することができる。このような状態を位置透過性を備えているという。

2.1.2　NFS: Network File System

NFS は、異なるオペレーティングシステム間の接続を意識して設計された分散ファイルシステムである。そのため NFS では、XDR(eXternal Data Representation) を用いて機種・アーキテクチャに依存しない形式のプロトコルを使用し、RPC の上に実装されている。

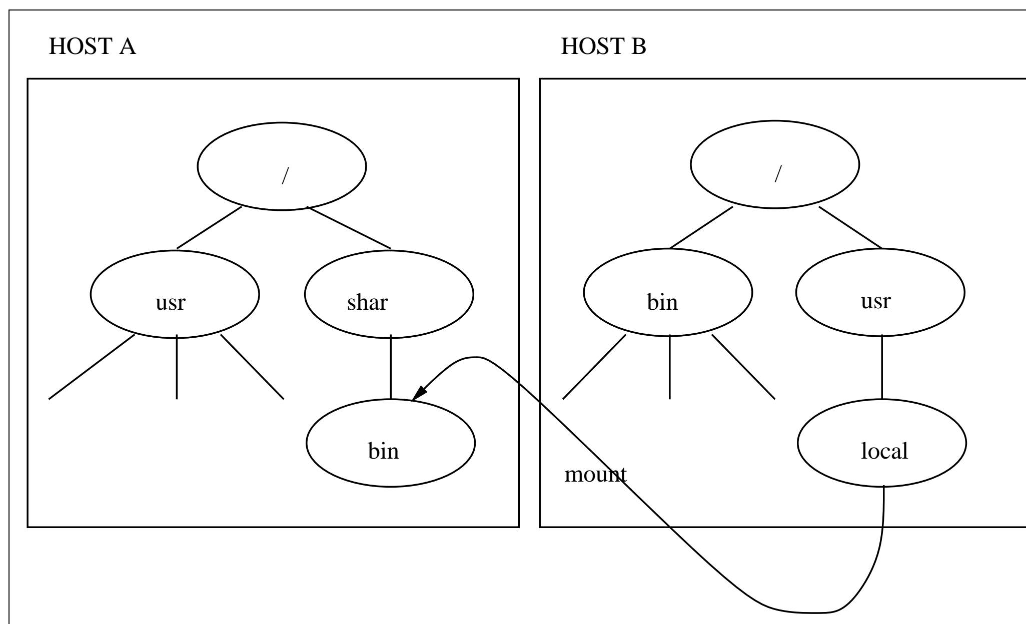

図 2.1: リモートマウントの概念

ステートレス

NFS は、ステートレスな構造をしている。ここでいうステートレスとは、サーバには過去の要求についてのいかなる情報も蓄積せずに、情報の保持はすべてクライアントの責任とすることを意味する。これによってネットワークの故障などの事故時の回復が容易になる。しかし、ステートレスな構造にすることによって、今までの Unix のローカルファイルに対するアクセスのセマンティクスを保持することが困難になる場合もある[1]。

遠隔手続き呼び出し (RPC)

NFS のプロトコルは、遠隔手続き呼び出し (RPC) を使用している。RPC を用いることによって、リモートサービスの定義、設計、実装が容易になっている。NFS のプロトコルは、手続き、その引数と結果、副作用の集合から定義されている。RPC は同期的に動き、プロセス間通信という複雑な手続きがプログラマからは見えず、ローカルマシンでの手続き呼び出しと同じように使用できるため、非常に便利である (図 2.2参照)。

VFS と vnode

Unix のファイルシステムはファイルシステム ID と inode に対応するファイルやディレクトリから構成されている。inode にはファイルを管理するための様々な情報が含まれており、一つのファイルシステム中で唯一の番号を持っている。しかし、他の計算機のファイルシステムをリモートマウントで使用すると、一つの計算機中に同じファイルシステム ID、同じ inode を持った違うファイルやディレクトリが存在する可能性があり、inode

[1]例えば、デバイスファイルなどがアクセスできないなどである。

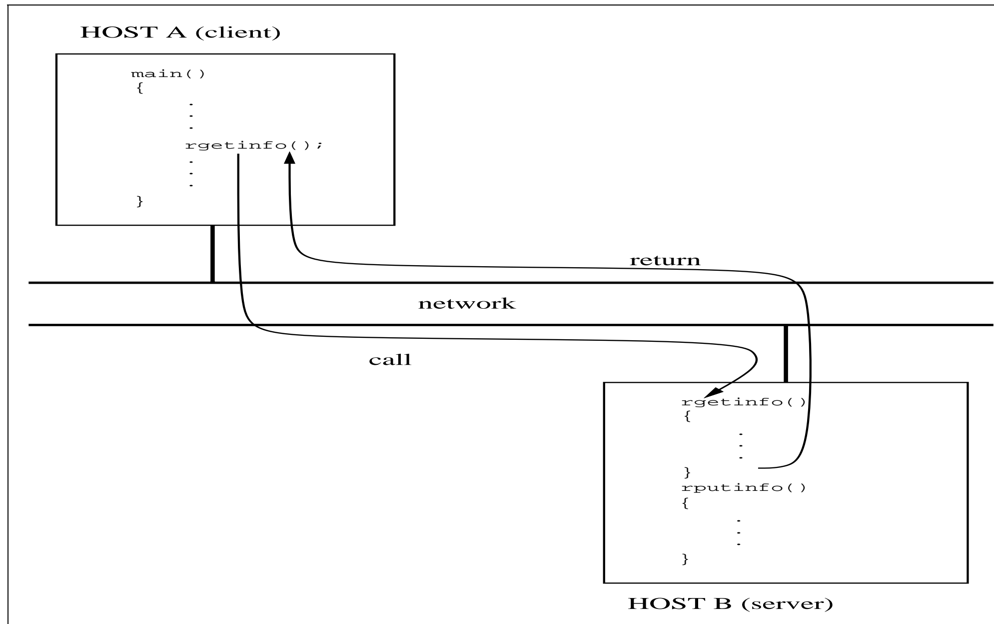

図 2.2: 遠隔手続き呼び出し

の一貫性が失われてしまう。このような問題を解決するために、カーネル内に新しいファイルシステムのインタフェースが実装・提供された。それらが inode をネットワークワイドに拡張した vnode(Virtual Node) と VFS(Virtual File System) である。VFS はファイルシステム群全体に対する操作について、vnode はあるファイルシステム内のファイルの操作について定義している。NFS ではシステムコールが呼ばれると、目的のファイルの vnode を見つけて、そのシステムコールに対応した vnode 操作関数が呼ばれる。その関数の中で目的のファイルがローカルなものかリモートなものか判断し、それぞれに対応したファイル操作関数が選択される。目的のファイルがリモートファイルだった場合、その操作の要求は NFS のクライアントに渡され、さらに対応する RPC によって NFS のサーバに要求が出される (図 2.3参照)。

2.2　物理的共有と論理的共有

このように現在の共有ファイルシステムは、リモートマウントという概念によって実現されているのが一般的である。NFS、RFS ではリモートマウント方式が採用されており、特に NFS は多くの機種・オペレーティングシステムで実装され、稼働していることはすでに述べた。このリモートマウントによるファイル共有方式には様々な利点があるが、異なる機種・アーキテクチャ間での “物理的な共有ファイルシステム” を提供しているにすぎないとも言える。

本論文ではファイル共有の概念を一歩進めて “論理的な共有ファイルシステム” について考える。これは、ファイルをどのホストからでも同じように扱えるような機構をそなえたファイルシステムである。このような機構を持ったファイルシステムとして、これ

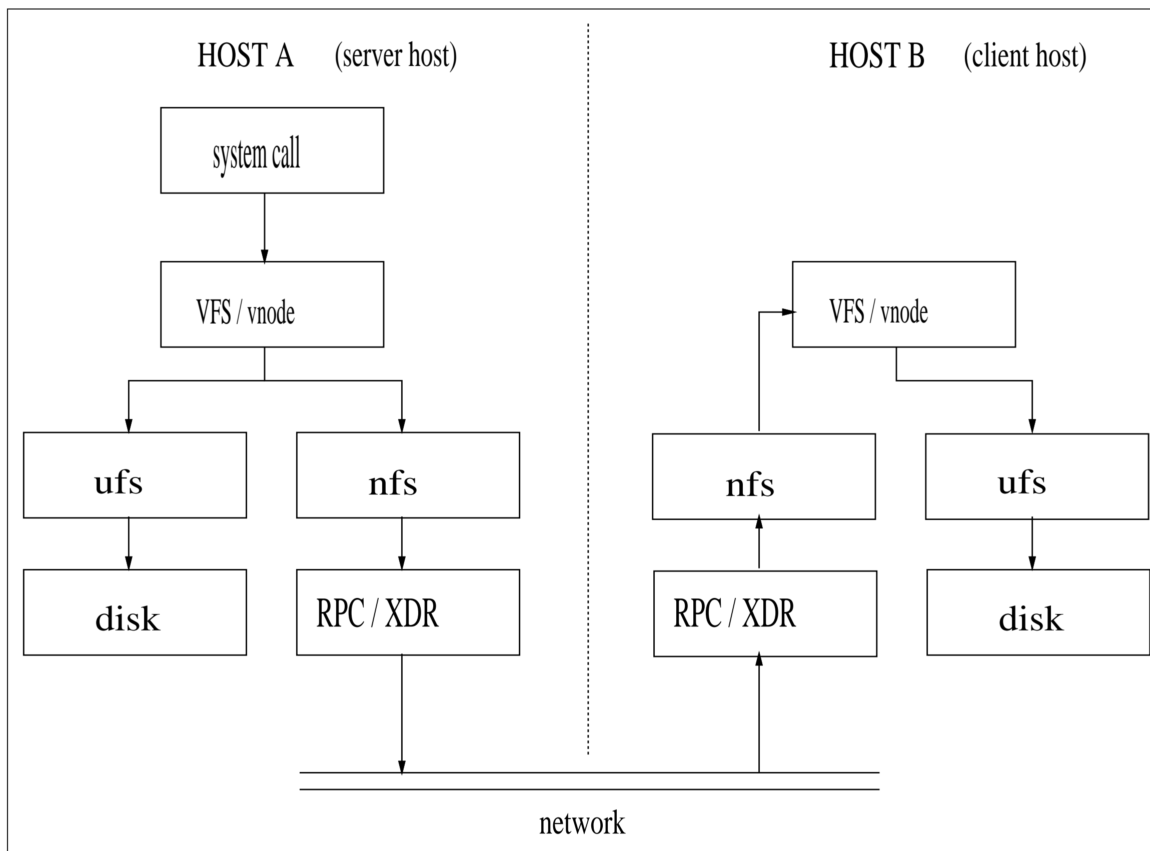

図 2.3: NFS の処理の流れ

までいくつかの研究が発表されているが、現在まだ標準になるようなものは存在しない。

例えば、テキストファイルについては現在の共有ファイルシステムでも充分に論理的共有がなされている。どのホストでも同じように参照し、編集することが可能である。

これに対しバイナリファイルは、一部のデータ用のものを除いて、ほとんど論理的共有が行われていないと言わざるを得ない。特に実行形式のファイル (ロードモジュール) の場合、その CPU アーキテクチャ、オペレーティングシステムの違いは決定的である。このようなファイルの物理的共有は意味を持たず、それどころか間違って使用しようとすると、意図しない結果など様々な弊害を起こしてしまう。これは、Unix にはその実行形式のファイルが自分のシステムで実行できるものかどうか判別する機構が確立していないことにも一因がある。

このように、共有できない機種依存性の高いファイルであるにもかかわらず物理的共有がなされてしまうようなファイルは、機種・アーキテクチャごとにまとめて、別々なディレクトリに格納しておき、それらをログイン時の環境設定によって、ユーザごとにそのホスト用の設定を行うというような方法によって無意味な共有による弊害を回避していることが多い (図 2.4参照)。このような設定は手間のかかるものであり、新しいホストやオペレーティングシステムの導入や、管理方針の修正などによって変更が必要になる。このような手間はできるだけ削減されるのが望ましいし、このような方法による解決は根本的な解決方法とは言い難い。

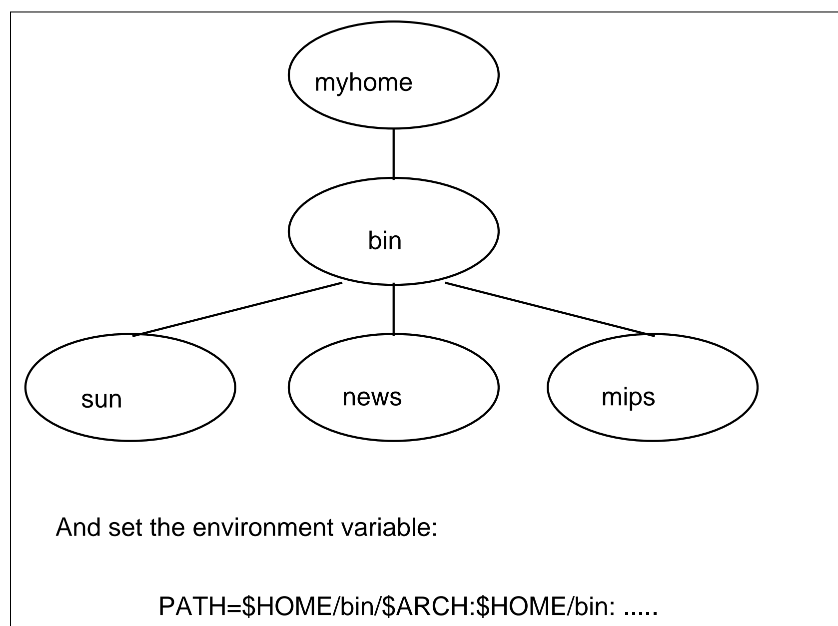

図 2.4: アーキテクチャごとにファイルを整理

2.3 問題点の整理

これまでの議論により、異なる機種・アーキテクチャ間でネットワーク上のファイル共有を行うには、ファイルに関する情報として現在の共有ファイルシステムが管理しているものだけでは不充分であり、ファイルごとに機種・アーキテクチャなどの情報も管理する必要があるということがわかる。このような情報を管理するファイルシステムを設計・実装・使用することによって物理的共有と論理的共有を一致させれば、異なる機種・アーキテクチャ間で位置透過性が保たれるようになる。これは、Unix のファイルシステムにおけるコンセプトであるファイルの一意的なパスによる名前付けが、ネットワーク上で共有される機種・アーキテクチャに依存性の高いファイルにも適応されることに他ならない。

つまり、

- ネットワーク上において、機種・アーキテクチャに対して透過なファイルシステム。
- ユーザにとってすべてのホストでの使用感が統一されること。

を考慮することが重要であると考えられる。

第 3 章

従来の研究とその問題点

前章までに述べたような異なる機種・アーキテクチャ間の共有ファイルシステムに関する問題の解決方法はいくつか提案されている。以下に示す 2 つが主なものである。

- コンディショナル・シンボリックリンク
- ファイルシステムの可視性制御

前者のコンディショナル・シンボリックリンクは、ファイルのパス名とそのファイルの実体との関係を制御する方法で、すでにいくつかの商業ベースのシステムに実装されている。後者のファイルシステムの可視性制御は、ファイルシステムの木構造の見え方を制御する方法で、いくつかの研究が発表されている。

3.1 コンディショナル・シンボリックリンク

コンディショナル・シンボリックリンクは、環境変数やカーネル変数によってシンボリックリンクのリンク先を自動的に変化させる機能である (Figure 3.1参照)。この機能は、Pyramid オペレーティングシステム [74] や SONY の NEWS-OS[75] に実装されている。

3.1.1 NEWS-OS の実装例

NEWS-OS のコンディショナル・シンボリックリンクは、環境変数によってバージョン、ハードウェアなどに依存するファイル[1]のリンク先を切り替えるのに使用されている。シンボリックリンクを行う際に、

ln -s $*env*==*val*?*name1*:*name2* *name*

とすることによって、環境変数 *env* の値が *val* ならば *name* は *name1* が、そうでなければ *name2* が参照される。また、複数の条件を指定する際には、

expr1?*name1*:expr2?*name2*: ... :*namen*

[1]具体的には、X ウィンドウシステムやカーネルのコンフィギュレーションファイルなど。

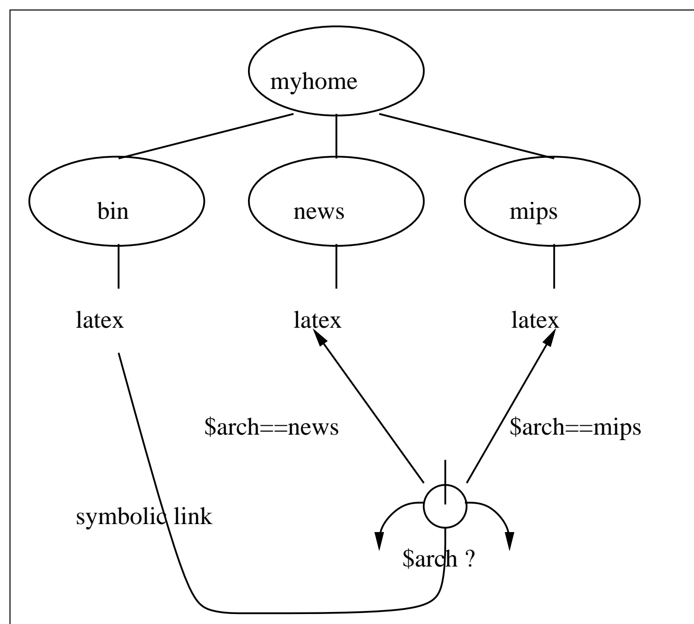

図 3.1: コンディショナル・シンボリックリンク

のようにする。

NEWS-OS では、そのカーネル内にコンディショナル・シンボリックリンクを処理するロジックが加えられている。

3.1.2 利点

コンディショナル・シンボリックリンクは、ユーザにとっては通常のシンボリックリンクと同様に利用でき、ユーザが自分の環境において制御し易い。任意のファイル間にリンクを張れるのはシンボリックリンクの利点である。

実装も比較的簡単に行える。シンボリックリンクを解釈するルーチンを書き換えて条件判断のロジックを追加するのは、そう難しいことではない。他のコマンドもそのまま使用できる。

実際に使用されている例からもわかるように、この方法は比較的固定された、ファイルやディレクトリの切り替えには適しているといえる。

3.1.3 問題点

コンディショナル・シンボリックリンクを利用できないホストとはファイルのパスを共有できない。

新しいアーキテクチャのマシンを追加する場合は、すべてのリンクを張り替えなくてはならない。リンク先を切り替える条件が複雑になった場合、その評価のオーバヘッドによってアクセス速度が低下する。多用すると、実際のファイルシステムの木構造が複雑になってしまい、管理がしづらくなる。

NEWS-OS のようにカーネルの変更による実装を行うためにはカーネルの再構成が必要で、開発の作業量が多い。

シンボリックリンクとしての問題点

コンディショナル・シンボリックリンクはシンボリックリンクに条件判断のロジックを追加したものであるため、シンボリックリンク自体の問題点を残している。

シンボリックリンクによって指されているファイルの側からリンクの存在を認識することはできない。そのため、ディレクトリを相対的に移動する際、自分では意図しないディレクトリに移動してしまうような不都合が起こる。また、リンクによって指されているファイルが削除されたり、移動されたりしても、リンクを張るファイルの側からリンク先になくなったことは認識できない。

3.2　ファイルの可視性制御

ファイルの可視性制御は、ユーザにファイルシステムの物理的な木構造とは異なる木構造を見せることによって使いやすいファイルシステムを提供する方法である。具体的には、その時その環境でユーザにとって不必要なファイル、利用できないファイルを見えないようにするものである。このため、ファイルの可視性制御による方法の共通の利点として、ユーザにとって、“**今見えているファイル**” は “**利用できるファイル**” であるので、非常に自然で理解しやすいという点があげられる。

ファイルの可視性制御によるファイルシステムの構築は、その可視性を制御する目的によって様々なシステムが考られ、プログラムの開発環境においてバージョンの管理を容易にする目的をもったものや、機種・アーキテクチャに依存するファイルを制御するものなど、いくつかの研究が発表されている。

3.2.1　3-D File System

3-D File System[76] は、ソフトウェア開発の際のメンテナンス、バージョンの管理、ソフトウェアの配布の際の簡便化を目的としたファイルシステムである。*viewpath* という概念によってファイルの可視性の制御を可能にしている。

viewpath

ソフトウェアの開発中には、複数のバージョンを利用する場合がある。開発者用のバージョンと配布用のバージョンとを用意するような場合がそうである。しかも、両方にはまったく同一内容のファイルが存在するので、そのようなファイルは共有したい。

ここで、それらのファイルが Figure 3.2のように別々のディレクトリ ofc、dvl に格納されているとする。配布用のバージョンを要求する人はディレクトリ ofc の下を参照すれば良いが、開発用のバージョンを要求する人はディレクトリ dvl の下だけ参照すれば良

いのではなく、ディレクトリ ofc の下の適切なファイルも参照しなければならない。そこで、これらのディレクトリがマージされ、Figure 3.3のように見えれば良いわけである。

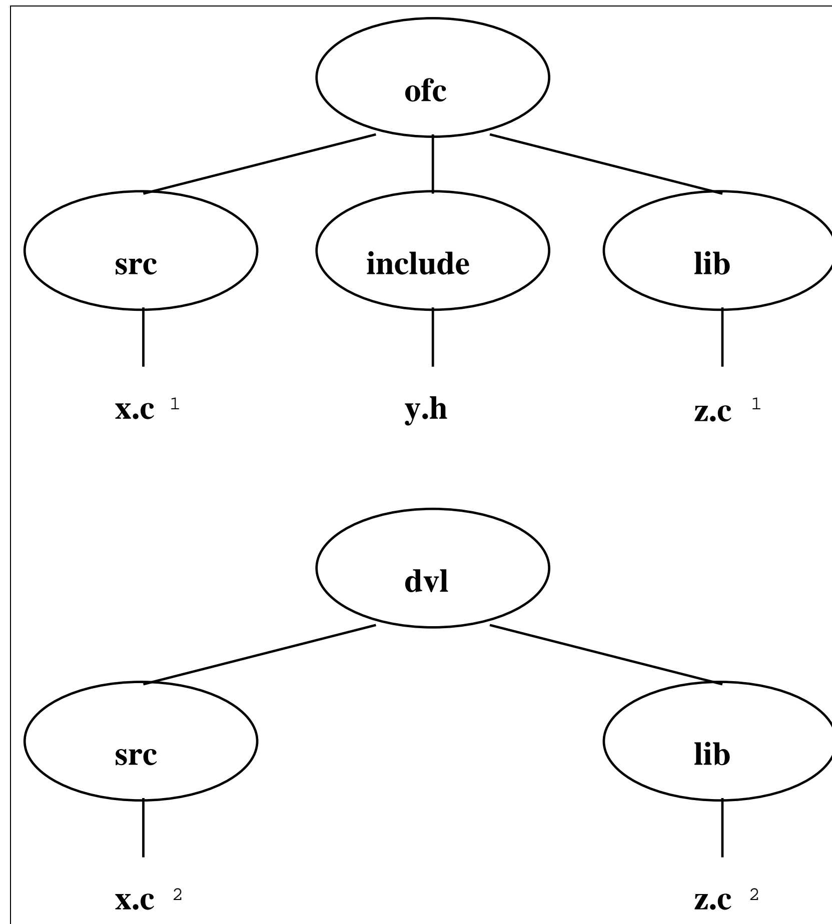

図 3.2: 実際のディレクトリ

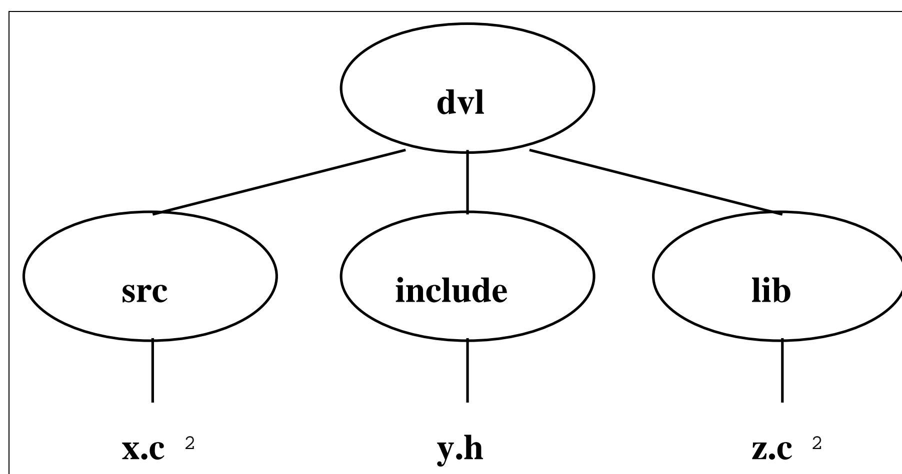

図 3.3: マージされたディレクトリ

3-D File System では、ファイルアクセス時に発行されるシステムコールが *Viewpathing Table* というパス名変換テーブルを参照することによってこれを可能にしている。この *Viewpathing Table* の書き方によって、機種・アーキテクチャごとにファイルを制御できる。

異なるバージョンのファイルへのアクセスには、“...” を用いる。Figure 3.3のマージされたディレクトリの例でカレントディレクトリが dvl/src であれば、“x.c” で dvp/src/x.c を、“.../x.c” では ofc/src/x.c を参照することができる。Unix のファイルシステムのセマンティクスである “.”、“..” が 2 次元的な移動であるのに対して、“...” はマージされたディレクトリのファイル間のアクセスに使用し 3 次元的なアクセスを意味する。

3-D File System では、ソフトウェアのバージョン管理のために VCS(Version Control System) というインタフェースが提供されている。VCS はシェルの機能を含んでおり、バージョン管理用にいくつかのサブコマンドが用意されている。

実装

3-D File System の現在の実装は、C 言語のライブラリといくつかのコマンドを書き換えることによって行われている。

利点

Viewpathing Table の変更は、VCS の内蔵コマンドによって行われるので、ユーザごとの設定がし易い。

またライブラリの変更による実装は比較的容易である。

問題点

バージョン管理を主眼において開発されており、機種・アーキテクチャごとにファイルを制御する用途に使用するには適していない。

現在の 3-D File System は read only のシステムであり、ファイルの書き込みに関してはまったく考慮されていない。

またライブラリを変更することによって実装されているため、ライブラリのソースが必要である。また、様々なアプリケーションプログラムをリンクし直さなければならない[2]。

3.2.2 TFS: Translucent File Service

TFS[77][78] は、Sun の SunOS4.1 から標準で実装されているファイルシステムである。

TFS は、3-D File System と同様にソフトウェアの配布、保守時のファイル管理を目的として開発された。

[2]ただし、オペレーティングシステムにライブラリのダイナミックリンケージを行う機構が備えられていれば再リンクは必要ない (例えば、SunOS など)。

TFS では、ディレクトリを多重にマウントすることによってできる階層の上下関係によって可視性を制御する方法を用いている。通常の NFS マウントの場合、マウントするとその下になるディレクトリやファイルは見えなくなってしまう。しかし、TFS のマウントではマウントでできる階層は透明なので下のファイルは透けて見える。同名のファイルが複数の階層に存在する場合、下のファイルはそれより上にあるファイルによって隠蔽され、それらのファイルのうち最上の階層にあるファイルが見える (Figure 3.4参照)。

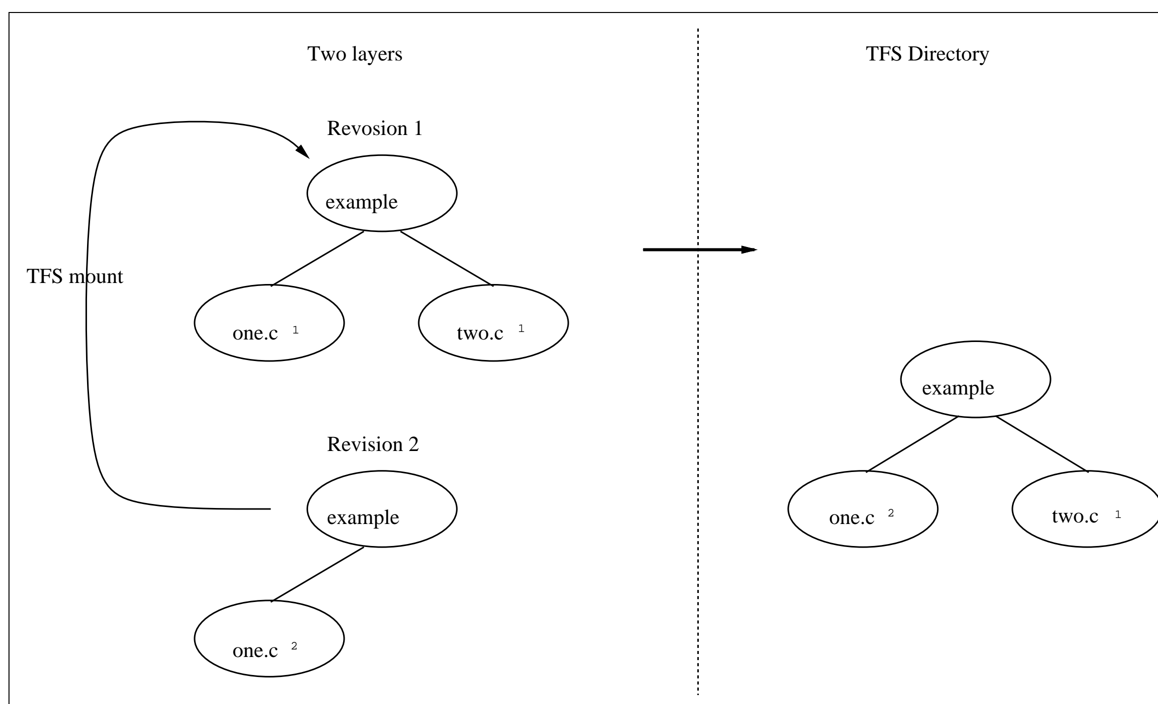

図 3.4: 2 階層の TFS ディレクトリ

copy-on-write

TFS では、最も上の階層だけ書き込み可能で、その他の階層はすべて読み込み専用である (Figure 3.5。読み込み専用の階層にあるファイルを変更しようすると、書き込み可能な階層 (最上の階層) に複製された後、変更される。

whiteout

TFS では、読み込み専用の階層にあるファイルを削除することはできない。もし、削除する必要がある時は、*whiteout* というエントリ最上の階層に作成することによって削除の代替とする。透明な階層に修正液を塗って下を隠すのと同じである。

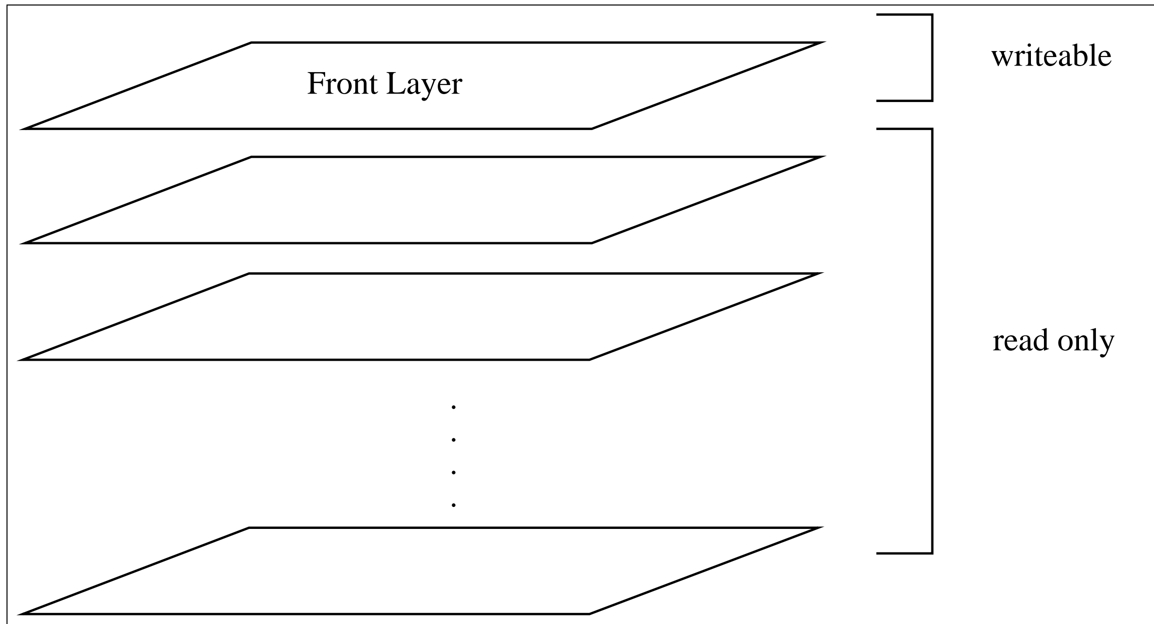

図 3.5: 階層の読み書き

ディレクトリの管理

TFS では、ディレクトリを読んだり (readdir) ファイルを探したり (lookup) する場合、それぞれの階層を最前から最後へと順々に探索しなければならない。その際、途中で見つかったファイルに関してはそれより下の階層を探索する必要はないし、*whiteout* のエントリがあれば、その時点でそのファイルは存在しないことにして良い。

しかしながら、ディレクトリの階層の数が多くなるとやはり探索のオーバヘッドが大きくなる。そこで一度 readdir や lookup の操作が行われると、その結果は最前の階層に backfiles というファイルにキャッシュとして保存される。最前の階層以外に backfiles が存在する場合、それは readdir や lookup の操作の時に参照にされ、それより下の階層を探索する必要がなくなる。

実装

現在 TFS は NFS サーバとして実装されている[3]。TFS で管理されているファイルへのアクセスはすべてサーバ tfsd を介して行われる。TFS のファイルは、カーネルにとって NFS のファイルとして解釈される。

TFS ファイルへのアクセス要求は、RPC を使って NFS のプロトコルでカーネルから tfsd に送られる。すでに存在するプロトコルのみを使用しているので、カーネル自体の変更を必要としない。システムコールの仕様も変更していない。よって、今までのプログラムも変更せずに動作する。

[3]しかし SunOS4.1 に実装されているものは、vnode に TFS という型が存在し、TFS 用の vnode 操作関数や VFS インタフェースがカーネルにリンクされている。

現在の実装でも、充分実用になる性能が得られている。ただし、巨大なファイルを読み書きするような場合に多少の速度の劣化が見られる。これは、複数の tfsd を起動しての read(2)、write(2) の並列化が実現されていないため、read(2)、write(2) のオーバヘッドが大きいためである。

利点

実装方法による利点はすでに述べた。

read only のシステムであった 3-D File System に比べ、TFS の *copy-on-write* という書き込みの概念は直観的にも理解しやすい。

問題点

TFS のセットアップには、mount(8) のオプションでマウントの型を指定するか mount_tfs(8) を使用するため、ユーザごとの設定はしづらい。

前述したように、ディレクトリの階層の数が多くなったときや、巨大なファイルを読み書きする際のオーバヘッドは大きい。

TFS でマウントしていくディレクトリは可視性が制御できるが、マウントされるディレクトリの元のパスが削除されるわけではないので、ファイルシステムの木構造は複雑になってしまう。

3.2.3 アーカイブディレクトリ

アーカイブディレクトリは、昨年の論文 [79] で我々が提案した概念である。

機種・アーキテクチャに依存するファイルを管理する場合、機種・アーキテクチャごとにディレクトリを作成し、そこにファイルをまとめておく方法とファイルごとに機種・アーキテクチャをまとめておく方法がある。いうまでもなく、前者のような管理の方法をとっているのが、今まで例にあげてきた 3-D File System や TFS である。

そこで、後者の方法を検討してみる。各機種・アーキテクチャ用に複数存在する同名のファイル群は論理的には同じ位置に置かれることが望ましい。現在の Unix のセマンティクスでは、同じ位置に複数のファイルを置くことはできない。逆に考えれば、そのように見せることができれば良いということである。

その一塊のファイル群と同名のディレクトリを作成し、そのディレクトリの中にそのファイル群をそれぞれ機種・アーキテクチャ名にリネームして格納することにする。この “ファイル群と同じ名前をもったディレクトリ” を**アーカイブディレクトリ** (*archive-directry*) と定義する (Figure 3.6参照)。

パス名の変換

このようにすると、アーカイブディレクトリにまとめられたファイルに通常のパス名でアクセスしようとすると、期待していたファイルにはアクセスできず、アーカイブディ

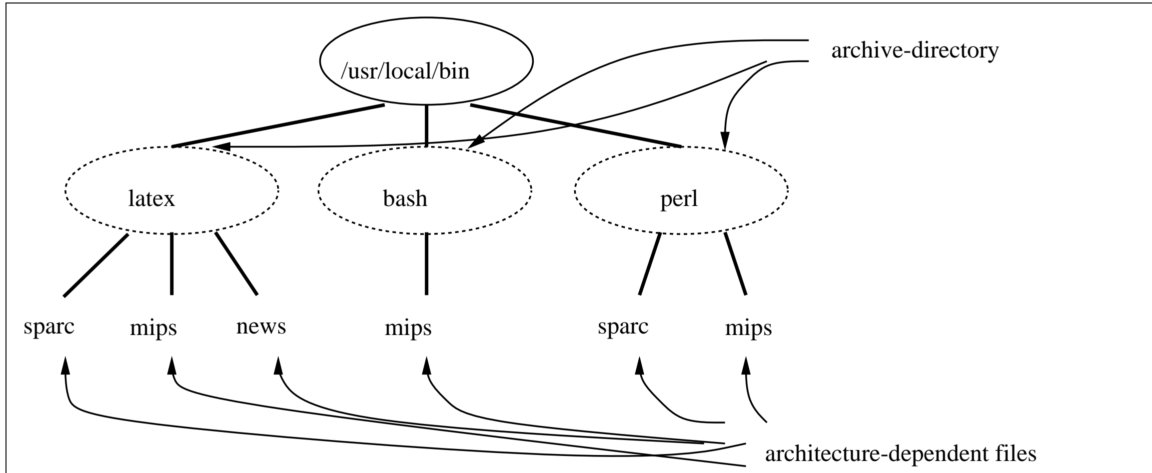

図 3.6: アーカイブディレクトリ

レクトリ自体にアクセスしてしまう。そこで、パス名を変換する機構が必要になる。

ユーザがあるファイルにアクセスしようとする。その時、そのファイルがアーカイブディレクトリだったら与えられたパス名をアーカイブディレクトリの中のその時の機種・アーキテクチャに適合した正しいファイルを指すように変換する (Figure 3.7参照)。機種・アーキテクチャの設定は環境変数で行う。そのため、ユーザ独自の環境を構築しやすい。

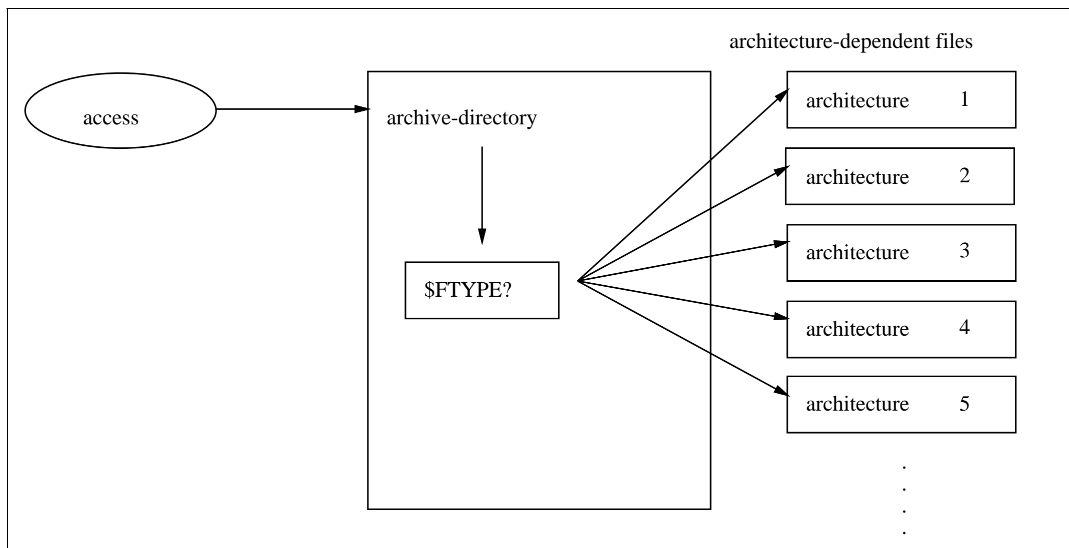

図 3.7: パスの変換

このような構造のため、その構造自体がパス名を変換する機構を補助することになり、3-D File System のように *Viewpathing Table* のような変換テーブルの必要もない。また、ファイル名を変更することによって inode などの構造を変更することなく情報を付加することができる。

利点

ディレクトリは一段深くなるだけなので、基本的なファイルシステムの木構造はそのまま保てる。

見かけ上は、読み込みに関しても、書き込みに関しても Unix のセマンティクスをくずしていない。

機種・アーキテクチャの設定は環境変数で行うので、ユーザ独自の環境を構築しやすい。

問題点

昨年の実装では、chdir(2) に副作用が起こることが確認されている。

chdir(2) では、/usr/local がアーカイブディレクトリで、システムが mips だった場合、“chdir /usr/local” は、カレントディレクトリは、実際には/usr/local/mips に移動する。ここで、“chdir ..” とすると、本来ならカレントディレクトリは/usr に移動すべきなのだが、実際には/usr/local に移動するだけで、ユーザにとってはまったく移動していないことになる (Figure 3.8参照)。

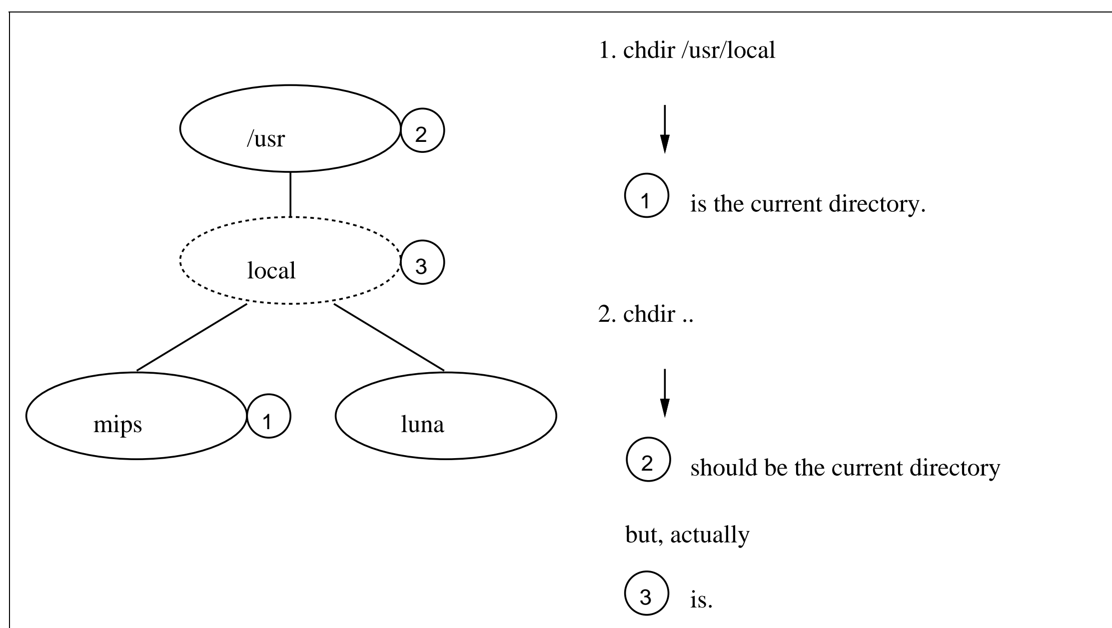

図 3.8: chdir(2) の副作用

3.3 まとめ

以上の比較から、機種・アーキテクチャに依存するファイルを整理し、ユーザにとって不必要なファイルを隠蔽するための方法としては、コンディショナル・シンボリックリンクでは不十分であることがわかる。シンボリックリンク自体の問題など解決しにくい問題が多いためである。

そこで、ファイルの可視性制御による方法を検討することにする。いくつか提案されている方法のうち、我々が提案したアーカイブディレクトリによる可視性制御には、他の研究と比較していくつかの有利な点がある。

しかしながら、現在の C のライブラリによる実装にはいくつかの問題点も残されている。

本論文では、その実装方法を見直し、より良い実装方法によるアーカイブディレクトリの実現を試みる。

第 4 章

アーカイブディレクトリの仕様と設計

この章では、アーカイブディレクトリの仕様について考察する。

4.1 仕様の説明

4.1.1 アーカイブディレクトリの構造

ファイルが元々もっていたファイル名はアーカイブディレクトリで代表されるため、各ファイルがその名前を保持する必要はなくなる。一方、どの機種・アーキテクチャのためのものかの情報を保持する必要があるので、ファイル名を機種・アーキテクチャ名にして、アーカイブディレクトリの下に格納する。こうすることによっていたずらに木構造を複雑にしたりすることなく、オーバヘッドの削減にもなる。

4.1.2 アーカイブディレクトリの判別

アーカイブディレクトリを新たなファイルの属性として定義するには inode を拡張する必要があるが、そのためにはカーネルの変更が必要である。そこで、現在はファイルのパーミッションにおいて、

ディレクトリである かつ **setUID ビットが立っている**

場合、アーカイブディレクトリとしている。

ファイルのパーミッションを調べるには、システムコール stat(2) を呼ばなければならない。システムコールはユーザアプリケーションとは違い、特権レベルで実行されるので、レベルの切り替え時のレジスタやスタックの退避などのオーバヘッドが大きい点に注意しなければならない。

4.1.3 識別用の環境変数

機種・アーキテクチャを識別するには環境変数を使用する。環境変数 $FTYPE$ に機種・アーキテクチャ名を記述しておく。複数の機種・アーキテクチャ名を同時に指定することも可能で、その場合は “:(コロン)” で区切ることにする。例えば

FTYPE=mips:sysv:penelope

となっていた場合、アーカイブディレクトリの下に mips、sysv、penelope のうちどれかが存在すれば、そのファイルがアクセスされる。

4.1.4　chdir(2) における副作用

前章でも述べたがこの副作用は、/usr/local などのシステム全体を管理するだけなら問題になりにくいが、ユーザが自分のホームディレクトリ以下にアーカイブディレクトリを用いるような場合頻繁に起こると推測される[1]。よって、この副作用は解決すべき問題である。

4.1.5　隠蔽されたファイルへのアクセス

そのシステムで使えないファイルは、可視性制御によって通常見えない。また見る必要はないはずである。しかし、見る手段をなくすことには問題がある。ライブラリによるアーカイブディレクトリの実装では、ls(1) に新しいオプションを追加することによって隠されたファイルを見ることができるようにしてある。この方法では、ファイルアクセスに関連するようなコマンドすべてにオプションを追加しなければならず、ソースプログラムを変更し、コンパイルし直さなければならない。またソースプログラムのないコマンドについては、隠されたファイルにアクセスができなくなってしまう。

そこで今回は、新しいセマンティクスを導入する。“...” によって隠されてファイルへのアクセスを許すことにする。例えば、前章の図 3.6のような木構造の場合

```
% ls /usr/local/bin/.../latex/
mips  news  sparc
%
```

のような、また同じく図 3.8のような木構造の場合

```
% ls /usr/local/...
luna  mips
% ls /usr/local/.../luna/bin
a2p         ctags       etags       lharc       s2p
a2ps        ea2ps       gs          mush        taintperl
bash        emacs       h2ph        perl        wterm
%
```

のような結果が得られるようになる。

このセマンティクスの導入によって、コマンドのソースプログラムを変更したりコンパイルし直したりする必要がなくなる。

[1]なぜなら、一般のユーザが頻繁に/usr/local などのディレクトリに移動するようなことは少ない。

第 5 章

実装

ここでは、いくつかの実装方法を提案・比較し、実際の実装について述べる。

5.1　各種実装方法の検討

アーカイブディレクトリの各種実装方法について比較・検討する。

5.1.1　C のライブラリの変更による方法

前版のアーカイブディレクトリはこの方法で実装されている。

ファイルをアクセスする時に呼ばれるシステムコールのフロントエンドとして、アーカイブディレクトリのパス名の変換を行うロジックを持った関数を実装している。すべての C プログラムは libc.a というライブラリをリンクするので、このライブラリの中にシステムコールのフロントエンドを入れておく。その後、アプリケーションプログラムをリンクし直すことによってアーカイブディレクトリに対応させることができる。

利点

　開発者以外のユーザに影響を与えずに実験・開発ができる。

　フロントエンドのライブラリは通常のプログラム作成と同じ手法で開発することができる。

問題点

　同じ方法で実装されている 3-D File System のところでも述べたように、ファイルアクセスを行うアプリケーションプログラムを、リンクあるいはコンパイルし直さなければならない。ソースがないものに関しては対応できない。

結論としては、実験的実装には優れた方法であるといえる。

5.1.2　カーネルの変更による方法

ファイルへのアクセスに関係するシステムコール自体を変更しパス名変換のロジックを実装する方法には、次のような特徴がある。

利点

　システムコール自体が変更されるため、今あるアプリケーションプログラムの変更をまったく必要としない。

　カーネルの変更による実装は自由度が高い。カーネルは CPU の特権レベルで実行されるため、どのような操作も可能である。

　カーネル内に機能を追加することによって、システムコール自体の処理は複雑になり、それ自体のオーバヘッドは増す。しかし、システムコールの呼び出し回数は増加しない。パス名変換のロジックはそれほど重い処理ではないので、システムコールの呼び出し回数が全体のスループットに影響すると考えられる。このため、余計なシステムコールが増えないのはオーバヘッドの低下になる。

問題点

　カーネルのソースを用意しなければならない。Unix の場合、ソースとバイナリのライセンスは別扱いであり、カーネルの変更には、ソースのライセンスが必要である。

　カーネルの変更による実験はその計算機を使用する他のユーザに与える影響が大きい。そのため、実験用のホストを用意するなどの配慮が必要になる。また、カーネルの変更による実装は、デバッグがしにくい。

　近年 Unix のカーネルは多機能を追及するあまり肥大化している。しかし、カーネルからいろいろなルーチンを追い出しカーネル自体を小さくした方が有利な点が多いと考えられる。Mach などの分散オペレーティングシステムではカーネルの機能は必要最小限に設計されているなどの例から見ても、カーネルにロジックを追加するのはあまり良い方法とは考えられない。

カーネル自体の変更による実装方法は、アーカイブディレクトリの仕様が一般的になり、動作が安定するようになってから、最終的に行う作業であると考えられる。

5.1.3　ファイル管理用デーモンによる方法

アーカイブディレクトリの管理を行うデーモンを作成する。アーカイブディレクトリやその内部のファイルをアクセスするような場合、必ずこのデーモンを介してアクセスが行われる。このデーモンの中にパス名を変換するロジックを実装する。

利点

　カーネルの変更を必要としない。よって、システムコールの仕様も変わらず、今までのアプリケーションプログラムはすべて変更・リコンパイルなしで動作する。

　同様な方法で実装されている TFS の文献 [77] からしても、充分な性能が得られることが期待できる。

　カーネルの変更による方法に比べて、開発が容易で移植性も高い。

問題点

このようなデーモンを設計するには、ファイルアクセス時のトラップや NFS のプロトコル仕様、RPC などに関する詳しい知識が必要である。

充分な性能が得られるとはいうものの、上記の 2 つの方法よりはオーバヘッドが大きいことが予想される。

オーバヘッドの問題があるものの、NFS のように複数のデーモンを走らせてで管理するなどの解決方法が期待できる。
このような仕組みのデーモンは、TFS、automount、amd[1]などいくつかの実装例があり、これらのサーバは一般でも広く使用されている。
そこで今回は、この管理用デーモンを作成する方法で実装を行うことにする。

5.2 ファイルアクセスをトラップする方法

アーカイブディレクトリの管理を行うデーモン vcfsd(View Control File System Daemon) は、アーカイブディレクトリやその内部のファイルにアクセスがあった場合に、パス名の変換を行う。つまりあるファイルにアクセスがあったことをトラップする必要がある。トラップの仕組みはこのデーモンの重要な要素の一つである。
ファイルアクセスのトラップはシステムコール mount(2) を使用して実現できる。

5.2.1 mount(2)

現在 mount(2) は、ローカルディスクのマウントと NFS によるリモートマウントをサポートしている。この後、それぞれ NFS タイプ、ufs タイプのマウントと呼ぶことにする。
mount(2) の仕様は次のようになっている。

```
int mount(type, dir, M_NEWTYPE|flags, data)
char *type;
char *dir;
int flags;
caddr_t data;
```

第一引数の **type** には、マウントのタイプを文字列で渡す (例えば “nfs”)。このタイプによって、第四引数の **data** の構造体の解釈が異なる。
NFS タイプの場合、**data** で指されるデータの構造体は次のようになっている。

```
struct nfs_args {
        struct sockaddr_in  *addr; /* file server address */
```

[1]ファイルシステムのマウントを動的に行うデーモン。automount は SunOS に標準で実装されており、amd はフリーソフトウェアである。

```
        fhandle_t  *fh;        /* File handle to be mounted */
        int        flags;      /* flags */
        int        wsize;      /* write size in bytes */
        int        rsize;      /* read size in bytes */
        int        timeo;      /* initial timeout in .1 secs */
        int        retrans;    /* times to retry send */
        char       *hostname;  /* server's hostname */
        int        acregmin;   /* attr cache file min secs */
        int        acregmax;   /* attr cache file max secs */
        int        acdirmin;   /* attr cache dir min secs */
        int        acdirmax;   /* attr cache dir max secs */
        char       *netname;   /* server's netname */
};
```

この構造体の “struct sockaddr_in *addr” にはそのファイルシステムを管理するファイルサーバのポートを登録する。

つまり NFS タイプのマウントでは、マウントを行う際にファイルサーバ (NFS サーバ) のポートを登録しておき、マウントしたディレクトリ以下のファイルがアクセスされる際、そのポートに対してカーネルから NFS のプロトコルでリクエストが発行される。

マウントシステムコールを発行する際に、vcfsd のポートを指定すれば、カーネルからのリクエストは vcfsd に送られることになる。

5.2.2　NFS サーバ

NFS サーバとは、ファイルアクセスのリクエストを受け取りその結果を返すサーバプロセスである。NFS・TFS における nfsd・tfsd や automountd、amd も NFS サーバである。vcfsd も NFS サーバの形式をとる。

前述のように、vcfsd のポートを mount(2) で登録しておくと、カーネルからのリクエストは NFS のプロトコルに定められた形式の RPC[2]として vcfsd に送られる。これらの RPC を処理する関数はファイルアクセス用のシステムコールとほぼ一対一に対応している[3]。表 5.1にその一覧を示す。

RPC で呼ばれる NFS サーバ側に、RPC のリクエストを処理する関数群を用意しておく。これらの関数にパス名を変換するロジックを実装する。

[2]この NFS のリクエストのプロトコルの詳細は RFC1094 に定義されている。

[3]ただし、NFS はステートレスな構造になっているので、サーバ側に状態を保持する必要があるいくつかのシステムコールに対応した RPC はない (例えば open など)。

nfsproc_null	nfsproc_create
nfsproc_getattr	nfsproc_remove
nfsproc_setattr	nfsproc_rename
nfsproc_root	nfsproc_link
nfsproc_lookup	nfsproc_symlink
nfsproc_readlink	nfsproc_mkdir
nfsproc_read	nfsproc_rmdir
nfsproc_writecache	nfsproc_readdir
nfsproc_write	nfsproc_statfs

表 5.1: NFS の RPC

5.2.3 ファイルハンドル

NFS のプロトコルではファイル操作の際、ファイルハンドル (*file handle*) と呼ばれる識別子を使用している。

クライアント側ではファイルハンドルの中身のデータ構造について知る必要はない。クライアント側はリクエストの結果としてサーバ側から与えられたファイルハンドルを保持しておき、次のファイル操作時に利用する。このためファイルハンドルは、クライアント側では *opaque data*(不明瞭なデータ) と呼ばれる。

サーバ側では、ファイルハンドルによって操作の対象となるファイルを一意に決定することができれば、ファイルハンドルを自由に定義して良い。例えば、NFS では一般にファイルシステムの ID と inode が使用されている。

vcfsd では、ファイルハンドルに認証用のデータとサーバでのファイルエントリへのインデックスを使用する。vcfsd ではマウントされたファイルへの実際の操作はさらにシステムコールを通して行うので、ファイルシステムを識別するような識別子は必要ない。

5.3 ディレクトリの移動

vcfsd を使用してディレクトリを管理する場合、そのディレクトリを別な位置に移動する必要がある。vcfsd が管理するディレクトリはファイルアクセスをトラップするためにマウントしなければばならない。しかし、マウントするとそのマウントポイント以下に今まで存在したファイル・ディレクトリはアクセスできなくなるからである。

例えば、/usr 以下のディレクトリを vcfsd で管理するとする。ファイルアクセスをトラップするためには/usr をマウントポイントにする必要があるので、あらかじめ/usr 以下のファイルは全て/vcfs/usr に移動しておく。ここで、vcfsd が/vcfs/usr を/usr にマウントすれば元々/usr にあったファイルが vcfsd で管理できるようになる (図 5.1)。

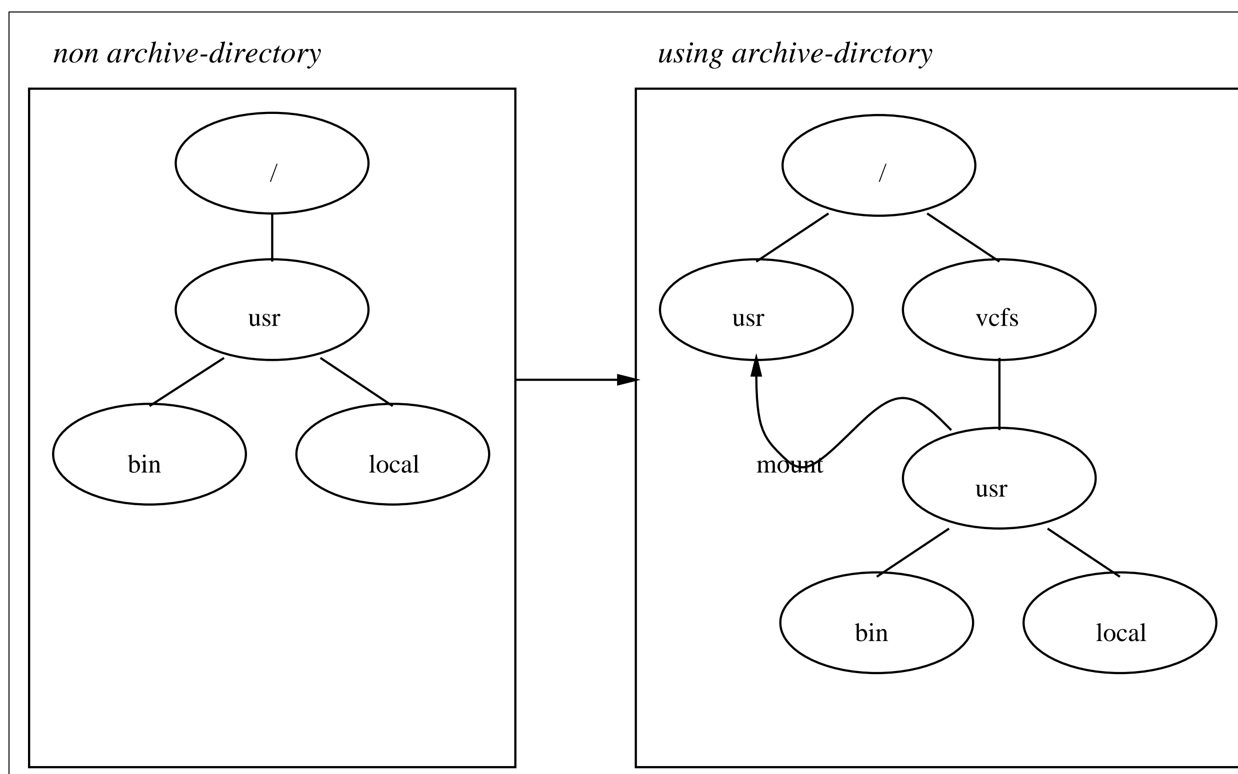

図 5.1: ディレクトリの移動

5.4 パス名の変換

前記の図 5.1のようなディレクトリ構成で管理を行う場合、パス名の変換は次のように行う。

1. 環境変数から機種・アーキテクチャ名を調べる。
2. 指定されたパス名の先頭に実体の置いてあるディレクトリ名 (図では “/vcfs”) を付加する。
3. 付加したパス名中のアーカイブディレクトリ内にその機種・アーキテクチャ名のファイルが存在するか調べる。存在しなければエラーを返す。
4. 付加したパス名の最後にさらに機種・アーキテクチャ名を付加する。
5. そのパス名を返す。

5.5 リクエストとデータの流れ

リクエストとデータの処理の流れは次の通りである (図 5.2参照)。

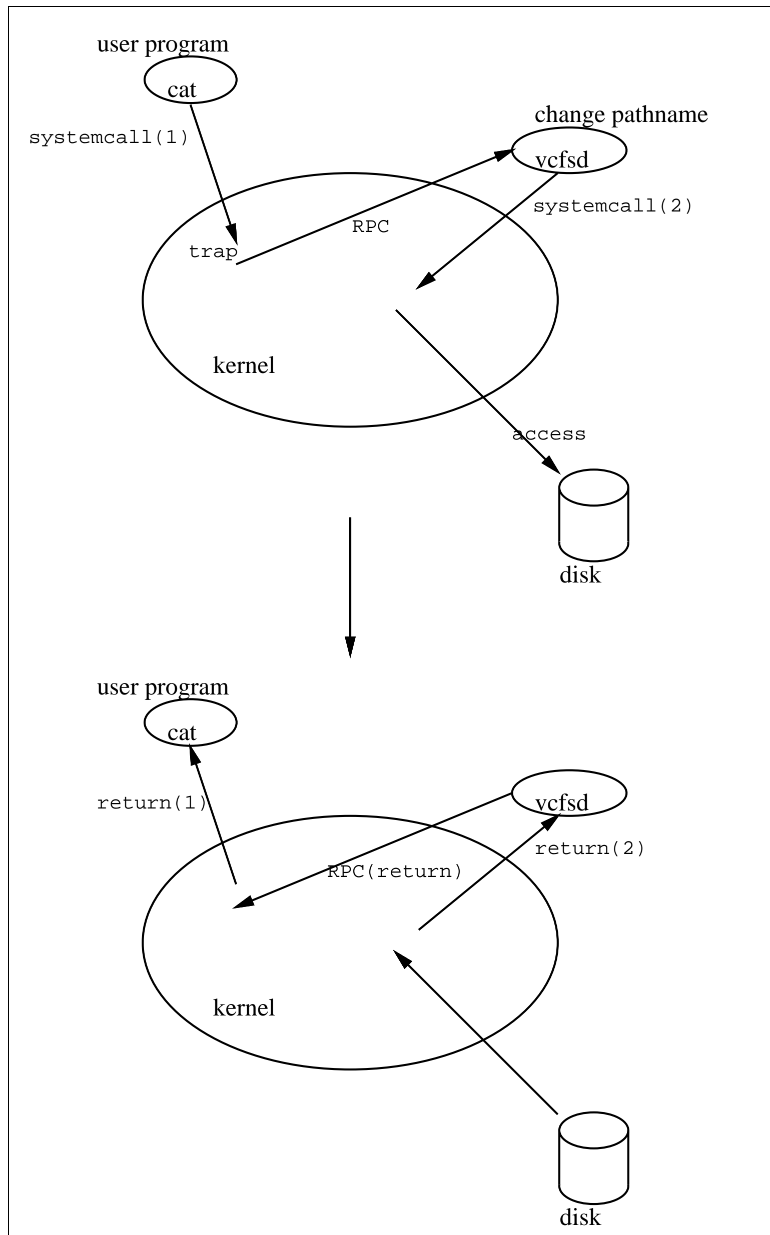

図 5.2: リクエストとデータの流れ

1. アプリケーションプログラム (図では cat) がファイルアクセスのため、システムコール (1) を呼ぶ。
2. カーネルは、アクセスされたファイルが vcfsd によって管理されているファイルシステムなので、そのリクエストを RPC で vcfsd に送る。
3. vcfsd はリクエストを受け取り、ファイルハンドルによって示されるパスがアーカイブディレクトリかどうか調べる。
4. アーカイブディレクトリであればパス名を変換し、リクエストに対応したシステムコール (2) を呼ぶ。
5. カーネルはシステムコール (2) に従いファイルをアクセスし、結果 (2) を vcfsd に返す。
6. vcfsd はその結果 (2) を RPC の返り値に加工し、カーネルに返す。
7. カーネルはその返り値をシステムコール (1) の結果 (1) としてアプリケーションプログラムに返す。

5.6　“...” の処理

隠されたファイルへのアクセスを許すセマンティクスには”...” を使用することは設計の章で述べた。

“...” の処理は、まずパス名中に “...” が現れたら、その親のディレクトリがアーカイブディレクトリかどうか調べる。アーカイブディレクトリであれば、パス名の変換をせずに、“...” をパス名から削除する。つまり “...” はアーカイブディレクトリ自体を指せば良い。

第 6 章

評価

6.1　他のシステムとの比較

可視性制御を行うシステムといっても、設計思想や使用目的が異なる他のシステムとの比較を単純に行うことはできない。しかし、ここでは次に挙げるような点から見た比較をしてみる。

書き込みの扱い 書き込みに対する扱いには、各システムで特徴がでている。3-D File System は Read Only のシステムであるので書き込みに対する制御機構を設けていない。TFS では *copy-on-write* という概念を導入して書き込み可能な階層を用意している。これに対してアーカイブディレクトリでは書き込みに対しても制御機構が働き、変更したいファイルの実体が変更されるので非常に自然である。

導入のし易さ 導入のし易さは、主に実装方法によるものである。前版のアーカイブディレクトリや 3-D File System のようなライブラリの変更による実装では、ライブラリをリンクし直す作業が必要であるし、新たなコマンドを用意したりコマンドオプションの追加などを行う必要がある。TFS や現在のアーカイブディレクトリのような管理用デーモンによる実装ではそのような作業は必要ない。NFS サーバは他のマシンへの移植性が高いのでその点でも有利である。

木構造の変化 導入のし易さとも関連するが、すでに構築されたファイルシステムの木構造を変化させるのは大変な作業である。しかも大幅な木構造の変化はユーザに与える影響が大きい。アーカイブディレクトリでは、その導入にあたってほとんど木構造を変化させる必要がない。

以上のような比較の結果から、アーカイブディレクトリは自由度が高く、他のシステムと比べても自然な使用感が得られる。

6.2　オーバヘッド

アーカイブディレクトリで管理されるファイルをアクセスするときのオーバヘッドについて考察する。

6.2.1　システムコールの増加によるオーバヘッド

設計の章でも述べたように、アーカイブディレクトリであることを調べるには、システムコールを呼び出さなければならない。このオーバヘッドは意外に大きい。昨年の評価データによると、アーカイブディレクトリに対応したものはシステムコールの呼び出し回数が 2 倍から 3 倍に増加している。しかし、実際のアプリケーションプログラムではファイルのオープンなどは頻繁に起こるわけではないので、この程度のオーバヘッドは目をつぶれる範囲であると見なせる。

6.2.2　NFS サーバとしてのオーバヘッド

今回実装した vcfsd では、システムコールの増加に伴うオーバヘッドに加えて、NFS サーバとしてのオーバヘッドも存在する。

アーカイブディレクトリで用いるファイルシステム上のファイルへのアクセスは必ず vcfsd を通るので、そのためのオーバヘッドが起こる。実際同じような方法で実装されている他のサーバでも、多少の速度劣化が見られる。しかし、NFS サーバとして実装することの利点を考慮すれば、このようなオーバヘッドに見合う性能が得られると見なせる。

第 7 章

今後の課題

今後の課題・作業をいくつか挙げておく。

アーカイブディレクトリ作成コマンド

現在アーカイブディレクトリの作成は手動で行っている。具体的にはディレクトリを作成し、それに setUID ビットを立てる。この作業を一緒にするようなコマンドが必要である。

バージョン管理用ツール

現在のアーカイブディレクトリは基本的に機種・アーキテクチャに依存するファイルの管理を行うのが目的で設計されているので、プログラム開発時のバージョン管理についてはそれを目的に設計された他のシステムの方が優れている点が多い。プログラミング環境として使用する場合、バージョンの管理は非常に重要である。

アーカイブディレクトリは自由度が高いので、アーカイブディレクトリの利点を生かしたバージョンを管理するための専用コマンドが用意されることが望ましい。現在、バージョン管理用ツールの仕様などについて検討中である。

マウントの重ね合わせ

TFS で提供されたマウントの重ね合わせの概念をアーカイブディレクトリに取り込むことを検討している。透明なマウントはアーカイブディレクトリと併用すれば、バージョン管理だけでなく様々な用途に応用できると考えられる。その際には透明なマウントの実装と合わせて、vnode による実装方法を考えている。現在の vcfsd だけでなく、vnode にアーカイブディレクトリを用いるファイルシステムを新しいタイプのファイルシステムとして登録するようにし、VFS、vnode に対するインタフェースライブラリを作成し、カーネルにリンクする[1]。これによって現在のオーバヘッドが改善され、ファイル操作の性能を向上することができると考えられる。

[1]現在の NFS の実装はこのようになっている。

第 8 章

結論

アーカイブディレクトリの概念を実装することにより、NFS などの分散ファイルシステムでは実現されていなかった機種・アーキテクチャに依存するファイルやディレクトリをユーザから見えないように制御することができた。

NFS サーバとして実装することによって、カーネルを変更することなくパス名の変換を行うことができた。しかもライブラリの変更による実装のように現在使用されているアプリケーションプログラムをリンクし直す必要もなくアーカイブディレクトリによる制御が可能になった。

オーバヘッドに関しては、まだ定量的な測定をしていないが無視できる程度で抑えることができたと思われる。

アーカイブディレクトリはファイルシステムの木構造中の部分木を隠すことにも使用できる。ユーザに提供する木構造を単純化することによって巨大化したファイルシステムのアクセスを容易にすることができる。