第 14 部

アーカイブサーバ

第 1 章

序論

1.1　はじめに

分散システムの重要性はますます高まってきている．マイクロプロセッサ技術の急速な進歩によって，マイクロプロセッサをベースとしたコンピュータは，大型機，ミニコンピュータに比較して，非常に強力かつ低価格になってきている．大規模なシステムをこのような小さなプロセッサをベースとしたコンピュータを多数用いて構成することは，様々な利点がある．まず，価格／性能の点で，以来の中央集中的なシステムと比較して優れている．システムを必要に応じて成長させていくことができることも利点の一つである．計算能力を増やしたいときは，その分だけプロセッサの数を増やせば良い．信頼性という点でも以来のシステムより優れている．システムの一部が壊れても，その部分を利用していない人たちは，それによって影響を受けることはない．しかし，システムを多数のプロセッサから構成することによって周辺機器，計算機上の情報などの計算機資源は分散してしまう．そこで，中央集中的なシステムでの利点も残しつつ，分散化を進めるためのシステムが分散システムである．ネットワークを通じて分散した計算機資源を共有することによって，中央集中的なシステムと同様の機能を提供する．

システムを利用するうえで，ユーザのファイルが複数のコンピュータに分散してしまい，利用するコンピュータによって，持つファイルが異なってしまうのは不便である．また，他のコンピュータからファイルをコピー（遠隔コピー）するにしても，同じファイルを同じネットワーク上の複数のコンピュータに持つのは無駄であるといえる．そこでこれらのファイルは分散（またはネットワーク）ファイルシステムによって，ネットワーク上で共有されるようになってきた．

ローカルエリアネットワーク（LAN）は組織の巨大化，またワークステーションの大量導入によって大きくなり，発展してきた．また，各 LAN は電話回線などを用いて接続されていたのが，最近になって電話回線等と

比較して高速な専用回線によって接続されるようになってきた．そのような環境では，各 LAN の間でのファイルの転送は以前よりもはるかに容易かつ高速で，LAN をまたがってファイルの共有を行なうことも可能になってきた．

現在，ファイルのバックアップ，ファイルシステムのダンプ等には磁気テープが主に使用されている．しかし，磁気テープは速度，容量，使い勝手，テープの保管の繁雑さといった点でバックアップ，ダンプのためのメディアとして満足できるものではない．普通に，ある期間おきにフルダンプをとり，その間はインクリメンタルダンプをとるとすれば，すぐに大量のテープがたまってしまうことになる．それらのテープの管理も大変であるが，それらのテープからリストアする場合はテープのかけかえだけでも大変である．

そこで，インターネットワーク上で大型の追記型の光ディスクドライブシステムを共有し，バックアップやダンプのために使用するということが考えられる．光ディスクは非常に大容量（600~800MByte）であるうえ，コンパクトである．追記型（一度書き込んだものは消すことができない）であるということは，バックアップやダンプのためにはそのファイルとしての性格上利点である．

本論文では，追記型の光ディスクを用いたシステムをインターネットワーク上でアーカイブファイルサーバーとして使用するため，追記型光ディスクの性質，アーカイブファイルサーバーの記憶メディアとして使用する場合の考慮点，実装方法について議論する．

1.2 本論文の構成

本論文は，１章で，本研究の目的を述べ，２章では，背景となる技術について述べる．３章では，光ディスク及びジュークボックスの特性について述べる．４章では，アーカイブファイルサーバーの設計について考察し，５章で，その実装方法について考える．

第 2 章

背景となる技術

分散システムより以前の段階では，現在使用しているのとは異なるコンピュータ上のファイル（遠隔ファイル）を使用したいときは，そのファイルをファイル転送コマンドを実行することによって，コンピュータ間でファイルを移動していた．分散ファイルシステムは，分散したコンピュータのファイルシステムを結び付けようとするものである．分散ファイルシステムによって結びつけられたファイルシステムでは，遠隔ファイルを，ローカルなファイルと同様にアクセスすることができる．この時ファイルシステムは，複数のコンピュータのファイルシステムから構成されているということが，利用者からは見えない（透明である）ということが重要である．この場合，ファイルシステムはいくつかの異なったファイルシステムの集合ではなく，仮想的な一つのファイルシステムとして，ユーザに提供される．

2.1 NFS

UNIX[1][78] 上では，現在 Sun マイクロシステム社の NFS (Sun Network FileSystem) [61] が広く使用されている．NFS はファイルシステムへの透過的な遠隔アクセスを提供するものである．NFS は，XDR(External Data Representation) を機種やシステムに依存しない形でプロトコルを表現するために用い，また RPC(Remote Procedure Call package) の上に実装することによって，異なったオペレーティングシステムやアーキテクチャの機種に用意に移植できるように設計された．NFS は，クライアント・サーバーモデルになっていて，リモートマシンのサーバーが，ローカルマシンのクライアントに対して，ある特定のファイルシステムを利用できるようにしている．

[1]UNIX は AT&T の商標である

2.1.1 設計目標

NFS は、同種でない機種を結んだネットワーク上で以下のことを実現することを目標とした.

- 機種とオペレーティングシステムに依存しない

  NFS サーバーがいろいろな異なった機種のクライアントに対して，ファイルを提供することを可能にするため，使用されるプロトコルは UNIX からは独立であるべきである．また，パーソナルコンピュータのような低レベルの機種上にも実装できるように，プロトコルは十分にシンプルでなければならない.

- クラッシュリカバリー容易さ

  クライアントがたくさんの異なったサーバーからリモートファイルシステムをマウントできるならば，クライアントがサーバーのクラッシュから用意にリカバリーできることが重要になる.

- アクセスの透過性

  プログラムから，リモートファイルをローカルファイルと全く同じ方法でアクセスできるようにする．特別なパスネームのパージングも，ライブラリも，再コンパイルも必要とはしない．プログラムは，ファイルがリモートであるかローカルであるか知らせることができるべきではない.

- クライアントでの UNIX のセマンティクスの維持

  UNIX マシンでの，透過的なアクセスのために，UNIX ファイルシステムのセマンティクスはリモートなファイルに対して維持されなければならない.

- 妥当なパフォーマンス

  もし，rcp のような，以来のネットワークのユーティリティーと比べて，NFS が早くなければ，たとえ使用するのが簡単であっても，誰も使用したくないであろう．Sun Network Disk protocol(ND) と同じぐらい高速，またはローカルディスクの 80%ぐらいのスピードを持つことが，目標となっている.

2.1.2 基本設計

NFS は，

- protocol
- server
- client

の，三つの主な要素から構成されている．

プロトコル

NFS プロトコルは以下のような特徴を持つ．

- RPC

  NFS のプロトコルは，Sun Remote Procedure Call (RPC) を使用している．RPC を持ちいることによって，リモートサービスの定義，構成，実装が用意になっている．NFS プロトコルは，プロシージャ，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，ローカルマシーンでのプロシジャーコールと同じように動くため，非常に使用しやすいものになっている．

- ステートレス

  NFS は，ステートレスプロトコルを使用している．それぞれのプロシージャコールへのパラメータは，その呼び出しを完了するために必要なすべての情報を含んでいる．そして，サーバーは，過去の要求についてのいかなる情報も蓄積しない．これによって，クラッシュからのリカバリーは非常に容易になる．サーバーがクラッシュした時，クライアントは NFS の要求を帰ってくるまで送るだけである．クライアントがクラッシュした時は，クライアントもサーバーもリカバリーのために，何も必要ではない．
  しかし，サーバーにおいて，何らかの情報が維持されているならば，リカバリーはもっと大変になる．クライアントもサーバーも，クラッシュを確実に探知する必要がある．サーバーは，サーバーの持っているクライアントのための状態を廃棄するため，クライアントのクラッシュを見つける必要がある．そして，サーバーの状態を再構成するため，クライアントはサーバーのクラッシュを察知する必要がある．

  ステートレスなプロトコルを使用することによって，複雑なクラッシュからのリカバリーを避けることができ，またプロトコルをシンプルにすることができる．クライアントが，応答が帰ってくるまで，単に要求を送り続けるだけならば，データはサーバーのクラッシュに

よって失われることはない．実際に，クライアントは，サーバーがクラッシュしてリカバーしたのか，またはサーバーが遅いのか，クライアントは，それらの違いを見分けることはできない．

- トランスポートレイヤからの独立

  RPC は，トランスポートに依存しないようにできている．そのため，上位レベルのプロトコルのコードに影響を与えることなく，新しいトランスポートプロトコルを，RPC のインプリメンテーションに埋め込むことができる．

- XDR

  NFS プロトコル,RPC は External Data Representation (XDR) のうえに作られている．XDR は，基本的なデータタイプのサイズ，バイトオーダー，アラインメント等を定義している．XDR を使用することによってプロトコルは機種や言語から独立になるだけでなく、定義することも容易になる．

NFS プロトコルでは，リモートマウントするための機能は提供していない．MOUNT プロトコルが，そのための機能を提供している．MOUNT プロトコルは，ディレクトリのパスネームを引数としてとり，クライアントがそのディレクトリを含むファイルシステムに対するアクセスパーミッションを持つならば，マウントのための情報を返す．リモートマウントのための機能を，異なったプロトコルにした理由は，プロトコルのオペレーティングシステムに依存した面を切り離し，また，新しいファイルシステムのアクセスパーミッションの方法を採用することが容易になる，といったことである．

NFS プロトコルプロシージャの概要は以下の通りである．

`null()` returns `()`

`lookup(dirfh, name)` returns `(fh, attr)`

`create(dirfh, name, attr)` returns `(newfh, attr)`

`remove(dirfh, name)` returns `(status)`

`getattr(fh)` returns `(attr)`

`setattr(fh, attr)` returns `(attr)`

`read(fh, offset, count)` returns `(attr, data)`

`write(fg, offset, count, data)` returns `(attr)`

`rename(dirfh, name, tofh, toname)` returns `(status)`

`link(dirfh, name, tofh, toanme)` returns `(status)`

`symlink(dirfh, name, string)` returns `(status)`

`readlink(fh)` returns `(string)`

`mkdir(dirfh, name, attr)` returns `(fh, newattr)`

`readdir(dirfh, cookie, count)` returns `(entries)`

`statfs(fh)` returns `(fsstats)`

サーバー

NFS サーバーはステートレスであるため，NFS リクエストを処理する時，結果を返す前に，変更されたデータを安定した記憶装置に書き留めておかなければならない．UNIX をベースとしたサーバーでは，ファイルシステムを変更したリクエストは，そのリクエストが返る時に変更されたデータはすべてディスクに書き込まれなければならない．つまり，例えば `write` リクエストにおいては，変更されたデータブロックだけではなくて，変更されたならば，inode を含むブロックも書き込まれなければならない．

サーバーを動作させるために必要な UNIX へのもう一つの変更は，inode 内のジェネレーションナンバーと，スーパーブロックにおけるファイルシステム id の追加である．inode ジェネレーションナンバーは，再利用された inode を以前のものと区別するために必要である．ジェネレーションナンバーは，inode がフリーになる度に増加される．

クライアント

クライアントの側では，NFS への透過的なインターフェースを提供している．リモートファイルへの透過的なアクセスをするために，パスネームの構造を変えない，リモートファイルのロケーティングの方法を使用する必要があった．いくつかの UNIX をベースとしたリモートファイルアクセスの方法は，リモートファイルの名前に *host:path* を使用するものである．しかし，この方法では，パスネームをパーズする既に存在するプログラムは，変更されなければならないため，透過的なアクセスができているとは言えない．

クライアントが，リモートファイルシステムをディレクトリへアタッチすることを許すことによって，ホストネームルックアップとファイルアドレスバインディングを一度に行なうことができる (Figure 2.1)．この方法は，クライアントはマウントする時のみ，ホストネームを扱うだけで良いという利点を持つ．また，クライアントをチェックすることによって，サーバーはファイルシステムのアクセスを制限することもできる．欠点は，マウントが完了するまで，リモートファイルが使用可能にならないことである．

一つのマシンにマウントされた，異なった型のファイルシステムへの透過的なアクセスは，カーネル内の新しいファイルシステムのインターフェースによって，提供される．それぞれのファイルシステムタイプは，二種類のオペレーションの集合をサポートする．VFS (Virtual File System) インターフェースは，ファイルシステム全体に対するオペレーションのプロシージャを定義する．vnode (Virtual Node) インターフェースは，あるファイルシステム内のファイルへのオペレーションのプロシージャを定義する．(Figure 2.2)

2.1.3 ファイルシステムインターフェース

VFS インターフェースは，一つのファイルシステム全体に対してなされるオペレーションを含んだストラクチャを用いてインプリメントされる．同様に，vnode インターフェースは，ファイルシステム内のノード（ファイルまたはディレクトリ）に対してなされるオペレーションを含んだストラクチャである．マウントされたファイルシステム毎に一つの VFS ストラクチャが、それぞれのアクティブなノード毎に一つの vnode ストラクチャがカーネル内にある．この抽象データ型のインプリメンテーションを使用することによって，カーネルは，使用しているファイルシステムがどのようにインプリメントされているかに関係なく，すべてのファイルシステムやノードを同じように扱うことができる．

各々の vnode は，その親 VFS とその上にマウントされた VFS へのポインタを持っている．これは，ファイルシステムのどのノードも，他のファイルシステムのマウントポイントになれることを意味している．root オペレーションが，VFS においてマウントされたファイルシステムのルート vnode を返すために提供されている．これは，カーネルないのパスネームのトラバースのルーチンの中で，マウントポイントを越えるために使用される．マウントされたファイルシステムの VFS は，マウントされたところの vnode へのポインタも，持っている．それによって，“..” を含んだパスネームはマウントポイントを越えて，トラバースすることができる．

VFS，vnode オペレーションに加えて，各々のファイルシステムタイプ

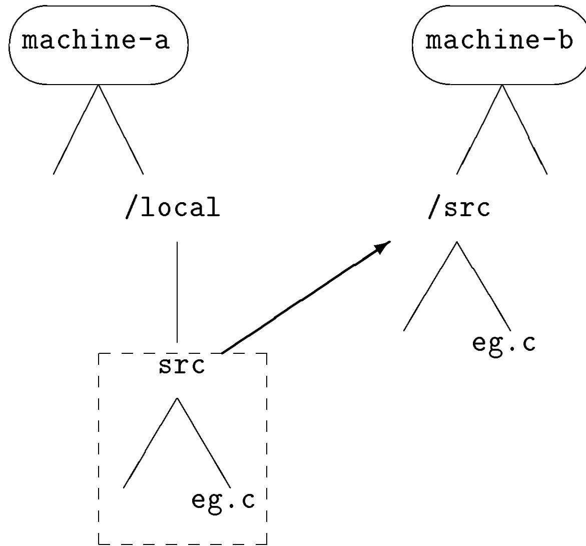

図 2.1: ファイルシステムのリモートマウント

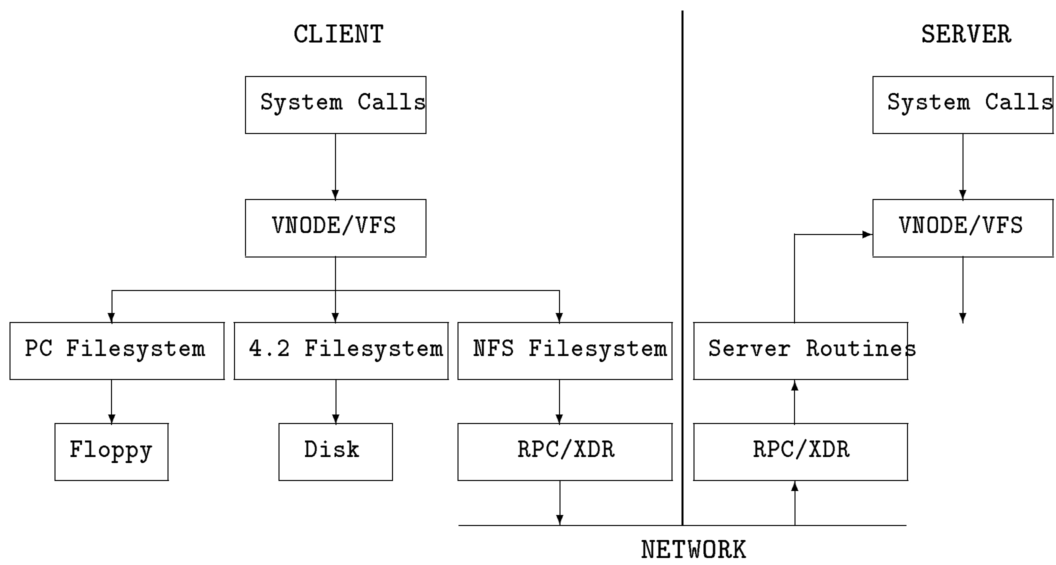

図 2.2: VFS，vnode の位置付け

は，普通のそしてルートファイルシステムをマウントするために，mount，mount_root オペレーションを提供しなければならない．

ファイルシステムインターフェースのために定義されたオペレーションは，以下の通りである．

- ファイルシステムオペレーション

  `mount(varies)`

  `mount_root()`

- VFS オペレーション

  `unmount(vfs)` retruns `(vnode)`

  `statfs(vfs)` retruns `(fsstatbuf)`

  `sync(vfs)`

- Vnode オペレーション

  `open(vnode, flags)`

  `close(vnode, flags)`

  `rdwr(vnode, uio, rwflag, flags)`

  `ioctl(vnode, cmd, data, rwflag)`

  `select(vnode, rwflag)`

  `getattr(vnode)` returns `(attr)`

  `setattr(vnode, attr)`

  `access(vnode, mode)`

  `lookup(dvnode, name)` returns `(vnode)`

  `create(dvnode, name, attr, excl, mode)` returns `(vnode)`

  `remove(dvnode, name)`

  `link(vnode, todvnode, toname)`

  `rename(dvnode, name, todvname, toname)`

  `mkdir(dvnode, name, attr)` returns `(dvnode)`

  `rmdir(dvnode, name)`

  `readdir(dvnode)` returns `(entries)`

  `symlink(dvnode, name, attr, to_name)`

`readlink(vp)` returns `(data)`

`fsync(vnode)`

`inactive(vnode)`

`bmap(vnode, blk)` returns `(devnode, mappedblk)`

`strategy(bp)`

`break(vnode, blockno)` returns `(buf)`

`brelse(vnode, buf)`

多くの vnode プロシージャは，NFS プロトコルプロシージャと 1 対 1 に対応している．一方，UNIX に依存した `open`，`close`，`ioctl` は，NFS プロトコルには現れない．`bmap`，`strategy`，`break`，`brelse` プロシージャは，バッファキャッシュを使用した読み書きをするために，使用される．

2.1.4　UNIX のファイルセマンティクス

NFS のクライアントでは，サーバーやプロトコルを変更すること無しに，UNIX ファイルシステムのセマンティクスにできるだけ従うようにしている．例えば，UNIX はオープンされたファイルを消去することを許している．プロセスは，ファイルをオープンし，そしてディレクトリエントリーからそのファイルを消去する．それによって，ファイルシステム内には，そのファイルの名前はなくなってしまうが，そのファイルに対して，読み書きを行なうことはできる．これは，ささいなことであるが，いやなことなので，最初はサポートされようとはされなかった．しかし，変更を加えたくない多くの（例えば csh，sendmail など）プログラムが，テンポラリーファイルとして，その様なファイルを使用しているため，サポートをする必要があった．

リモートファイルについて，オープンされたファイルを消去できるようにするためには，

1. クライアントの VFS で，ファイルがオープンされているかどうか，remove オペレーションの時にチェックする

2. もしそうならば，消去する代わりに，rename オペレーションを行なう

といったことをする必要があった．これによって，クライアントにとっては見えないような感じになって，読み書きは可能なままである．クライアントのカーネルは，vnode がアクティブでなくなった時に，その新しい名前を消去する．しかし，これは完全な解決方法とはなっていない．もし，ク

ライアントが rename と remove の間でクラッシュしたならば，ごみのファイルがサーバー上に残されてしまうからである．

リモートオープンファイルに関連した別の問題として，そのファイルのアクセスパーミッションが，リモートにオープンされている間に，変更され得るといったものがある．ローカルにアクセスする場合には，パーミッションはファイルがオープンされる時にのみチェックされるが，リモートにアクセスする場合には，NFS コールの度にチェックされる．これは，もしクライアントプログラムがファイルをオープンし，そしてパーミッションビットが変更されたならば，クライアントプログラムはもうリードパーミッションを持たないため，その後の read リクエストはフェイルしてしまう．この問題を避けるために，オープンする時に、どのクライアントがそのファイルをオープンしているのかをファイルテーブルに入れておき、その後のアクセス時のチェックはそれを用いる，といったことをしている．

同じリモートファイルを複数のクライアントが使用しているような場合，一つのクライアントで，それをコントロールすることはできないため，全ての UNIX open file semantics が維持されているわけではない．例えば，あるクライアントがファイルをオープンし、別のクライアントがそのファイルを消去した時、最初のクライアントの read 要求は失敗する．

2.2 Andrew File System

CMU の ITC で開発された AFS3.0[42] は，多くの実装技術を使用しており，以下に挙げる目標の多くを実現している．このシステムは，6 年に及ぶ大規模分散ファイルシステムに関する研究の成果として誕生した．

2.2.1 広域ファイルシステム

広域ファイルシステム（Wide-Area File System）を用いれば，正式に認められたコンピュータユーザならば，遠く離れた（たとえそれが大陸を挟んだ反対側や海を隔てた向こう側であっても）他の組織によって保存されているファイルを透過的にアクセスしたり，アップデートすることが可能となる．一般にこのようなシステムは，一つの権威によってではなく，いくつかのユーザ連合によって支えられている．ユーザのグループは，他のグループに自分たちのネットワークファイルシステム（セル (cell)）にアクセスできるようにしたのち，リモートセルにアクセスできるようになる．さらに，そのような連合の間では，共有ディレクトリ，保護，認証，ファイル転送プロトコル（あるいは機構）について同意する必要があるが，各自のセルについてのポリシーや管理については，各メンバが個別に行なっ

ていく．

以上のようなシステムは，慣例的に「国家規模（nationwide）のファイルシステム」と呼んではいる．またこのようなタイプのシステムでは，システムの下にぶらさがっているネットワークの存在を隠して，クライアントが（ネットワークを介してファイルにアクセスする場合でも）普通のファイルシステムコマンド（read, write, open, close など）を使えるようにしている．すなわち，広域ファイルシステムは，巨大な分散型の抽象データオブジェクトであり，限定されているとはいえ強力なデータベースにアクセスするための統一的な手段を提供する．また，広域ファイルシステムは，リモートログイン，ファイル転送，電子メールといった既存のネットワークサービスのごく自然な拡張であると考えることもできる．広域ファイルシステムは，TCP，IP，X.25 といった下位のネットワークプロトコルに依存するが，ハイパフォーマンスとリモートデータへの透過的なアクセスを実現するためには，セッション層，プレゼンテーション層，アプリケーション層のプロトコルを新たに多数用意する必要がある．国家規模のファイルサービスが必要な理由は，以下に挙げるようなもである．

- 広域ファイルサービスによって，個人やグループからなる共同研究は促進される．計算機産業だけをみても，組織間の共同研究が急速に広がりつつある．
- 広域ファイルサービスによって，機動力が増大する．家庭や旅行先にいても，ネットワークを介して，オフィスにいるような感覚でデータにアクセスすることができるのである．
- 広域ファイルシステムをデータベースとして使用すれば，遠隔地の情報をまるで手元にあるかのように参照することができる．適切なインデックスの形式が与えられていれば，広域ファイルサービスはレポート，設計仕様，プログラムソースを配布するための，非常に有効な手段になりうる．
- 適当な機密保護が確立できれば，購入したりライセンスを取りたいソフトウェアについて，ソフトウェアのベンダーのマシンの上のプログラムを遠隔操作で実行するということが可能なので，簡単に試すことができる．

以上を総合すると，広域ファイルサービスによって，コンピュータ支援による共同作業が非常に増加し，また簡単になると思われる．すなわち，国家規模のファイルシステムによって，多くの人々の作業環境は本質的に変化し，作業の柔軟性や生産性が大幅に向上するのである．

2.2.2 設計目標

アンドリューファイルシステムで目標とした，広域ファイルシステムモデルは，以下のようなものである．

- アクセス透過性

  ローカルシステム上でファイルをアクセス（ read や write ）するシステムコマンドが，リモートシステムに存在するファイルについても同様に扱えるようになっている必要がある．

- 認証と機密性

  サーバーとクライアントは，望ましくないアクセスに対してデータが保護されることを保証するために自分自身を認証できなくてはならない．

- セルの自律性

  セル（ファイルサーバーおよびクライアントの集まりであり，普通一つの組織に対してサービスし，普通一つの部屋とか建物といった場所の中にある）の管理者は，ユーザ ID の配分，ファイルサーバーの割当て，ユーザに対する記憶領域の割当ての管理，システムの構成といったことを，他のセルに影響を受けることなく独自に行なえなくてはならない．

- 他のファイルシステムとの共存

  広域ファイルシステムは，一つの組織によって使用されるファイルシステムのみであるとは限らない．したがって，クライアントが他のファイルシステムも同様に使えることが不可欠となる．

- 通信の効率的利用

  広域ファイルシステムはできるだけ通信の必要を減らすべきである．それによってパフォーマスが向上し，ネットワークの負荷のシステムへの影響も減少し，通信にかかるコストも下げることができる．

- 異常に対するセマンティクス

  システムやネットワークに異常が発生した場合，それによる影響を最小限にくいとめるべきである．さらにその影響を素早く解析できなければならない．

- 多様性

  広域ネットワークには，異なるハードウェア，異なるソフトウェアが含まれていることを考えれば，ファイルシステムは多くのマシン，大半のオペレーティングシステムをサポートしなければならない．ファイルサーバーについては UNIX で走っていると限定するにしても，クライアントは UNIX と同様に DOS，マッキントッシュOS，OS/2 上でも走らせられるようにすべきである．

- 名前の透過性

  ユーザはファイルを名前で参照できるようにして，その場所を突き止めるのはシステムに任せるべきである．ファイルの存在するセルを特定する必要はあるかも知れないが，ファイルの収められているファイルサーバーを特定する必要はないようにすべきである．

- パフォーマンス

  広域ファイルシステムは，ユーザに遠隔ファイルを使う気をなくさせないだけのパフォーマンスを保たなくてはならない．共同作業や共有の数を増やすには，十分なパフォーマンスが必要である．

- 機密保護

  データの共有を適度に許す一方で，細かい機密保護ができるように，保護情報はデータに付けるようにするべきである．

- システムの再構成

  セルの管理者は，どのクライアントにも影響を及ぼさないで，セルの構成を変えることができなければならない．たとえば，あるユーザのファイルをあるファイルサーバーから他のファイルサーバーへ，その変更をクライアントに知らせることなく，移動できなければ ならない．

- システムの規模

  広域ファイルシステムは数千個のセルを取り扱い，各セルが数千台のクライアントコンピュータをサポートできる必要がある．

- 共有アクセスのセマンティクス

  広域ファイルシステム上のファイルへの共有アクセスに関するセマンティクスが合理性を欠くものであってはならない．このセマンティクスは，ローカルファイルで用いられるセマンティクスと同様でなければならない．

2.2.3 有効な技術

これまで述べてきたような特徴を有する広域ファイルシステムを開発するのに役立つ技術がいくつか存在する．

- キャッシュ

  キャッシングは，おそらくそのうち最も重要な技術である．ファイルやディレクトリやそれらのアドレスをキャッシュすることによって，クライアントがネットワークを介してデータを転送したり，ファイルサーバーに要求を出したりする必要が減る．クライアントが（ディスクや RAM などを用いて）大きなローカルキャッシングを行なえば，必要なファイルをいちいちファイルサーバーから取り出さずにアクセスすることが可能になる．

- コールバック

  コールバックは，キャッシュの整合性を保ったり，キャッシュされた情報がアップデートされた場合の情報の保証に役立つ．コールバックプロシージャが起動されていれば，キャッシュされているデータに不都合が生じた場合，ファイルサーバーがそのことをクライアントに告げることができる．

- 論理ボリューム

  論理ボリュームを用いて，特定のユーザやプロジェクトに属しているファイルやディレクトリをまとめることができる．論理ボリュームは，記憶領域の割当てを管理したり，ファイルサーバー間でデータを移動する際に便利な概念である．

- ディレクトリサービス

  ディレクトリサービスを用いて，ファイル名やディレクトリ名を特定のファイルサーバーにマッピングすることができる．参照の局所性により，ディレクトリ情報はキャッシュするべきである．

- アクセス制御リスト

  ディレクトリ（場合によっては各ファイル）に付けられるアクセス制御リストは，ファイルへのアクセスを許すユーザやグループを規定する手段を提供する．これによって，管理者は広域ファイルシステムへの共有アクセスを細かく制御することができる．

- オーセンティケーション

MIT の Kerberos で用いられたような，暗号による認証技術によって，クライアントやファイルサーバーは互いに自分達を認証することができる．

- 高速なネットワーク

  NFSNet のような高速ネットワークを用いれば，T1（1 秒間に 1.5 メガバイトの速さ）を実現したり，ローカルエリアネットワーク間をより高速で接続することができる．

- RPC

  大きなファイルの塊を効率良く送ることのできるリモートプロシージャコールは，かなり細かい要求を出したり，高速でデータを効率良く転送するために必要である．各パケットを認識する，簡単な要求と反応のプロトコルでは，長距離ネットワークを介した場合，十分なバンド幅を得ることができない．

- レプリケーション

  何台かのサーバー上にディレクトリ，データ，ファイルの複製を作っておけば，ファイルシステムやネットワークの異常発生を防ぐだけでなく，システムのパフォーマンスを上げるのにも役立つ．read-only や read-mostly のデータの複製を作っておくのが最も簡単である．

2.2.4　インプリメンテーション

ファイルシステムそのものは，抽象的な AFS を実現するために用意されたいくつかの要素から成り立っている．

ファイルサーバー

AFS において最も重要なプログラムはファイルサーバープロセスであり（実際には，歴史的理由から，2 つの UNIX プロセスとして実装されている），各サーバーマシン上で稼働し，クライアントにファイルを提供する役目を担っている．

ファイルサーバーは，ボリュームと呼ばれるオブジェクトをほとんど排他的にに扱う．これは，小さなディスクパーティション（普通 1 〜 50MB）と考えることができ，サーバーマシンには数千のボリュームがロードされる．ボリュームはファイルとディレクトリの集まりであり，96 ビットのファイル ID（fid）で識別される．これが AFS の連結されるサブツリーを形成する．普通の UNIX システムにおいて各ディスクパーティションがマウント

されて一つのファイルシステムを形成するように，システム内の各ボリュームがマウントポイントで繋がって，グローバルな AFS の名前空間をなす．

もちろん，両者には違いがある．パーティションの場合と違い，ボリュームには常時固定されたディスクスペースは割り当てられない．その代わりに，ボリュームの多くがファイルサーバーマシン上の一つの UNIX のディスクパーティションを共有する．そして各ボリュームは，一つのボリュームに対する割当て量やそのディスクパーティションの使用可能なスペースの合計に制約があるものの，時間によって大きくなったり小さくなったりする．システム管理者がボリュームの割当て調整を動的に行なうことができるのである．

UNIX のマウントポイントの場合と異なり，AFS のボリュームのマウントポイントはファイルシステムのオブジェクトである．したがって，AFS にアクセスしているワークステーションは，すべて同じファイルシステムツリー構造を見ることになる．例えば，ファイル /afs/andrew.cmu.edu/usr/kazar は，usr.kazar というボリュームのマウントポイントである．このボリュームはユーザ Kazar の所有するファイルを含んでいる．

さらに，ボリュームはあるサーバーのディスクから他のサーバーのディスクへ，名前を変えることなく移動できる．このことは，システムにおける場所の透過性を実現するのに重要である．

ファイルサーバーは，ボリューム全体に対する操作（例えば，バックアップテープからボリュームをリストアする）およびボリューム内の各ファイルやディレクトリに対する操作（例えば，ファイルからデータブロックを取り出す）の両方に RPC を用いている．

キャッシュマネージャ

ファイルシステムにとって次に重要な要素として，AFS の各クライアントマシン上に置かれるキャッシュマネージャが挙げられる．キャッシュマネージャは，一番最近に使われたファイルやディレクトリをキャッシュし，UNIX のファイルシステムコールをキャッシュ参照の組み合わせおよび一つもしくはそれ以上のサーバーに対する RPC に変換する．また，キャッシュが一杯になった場合に，キャッシュからエントリを削除するということもキャッシュマネージャの大事な仕事である．

AFS 3.0 のキャッシュマネージャは，Sun Microsystems の vnode インタフェースを介して UNIX のカーネルとインタフェースをとる．vnode はもともとカーネルと Sun の NFS の間のすっきりしたインタフェースを提供するために設計されたものである．UNIX のカーネルはシステムコールを，外部のファイルシステムモジュールに対するプロシージャコールのセッ

トに変換し，必要な機能を実現する．

キャッシュマネージャとファイルサーバーの間では，ファイルは 96 ビットの fid によって識別される．他のシステムにあるような fid と違って，AFS の fid は不変なオブジェクトを名前をつけるものではない．同じ fid に名付られたファイルでも，別の時間には別の中身であることもありうる．

ファイルは，データ部分とステイタス部分に分けられる．データ部分は，8 ビットバイトのストリームからなっている．ステイタス部分は，ファイルの大きさ，そのファイルが作られた日付，保護モードといった情報が含まれている．重要なステイタス情報の一つは，ファイルのデータバージョンナンバーである．これはファイルのデータが変更される度にインクリメントされ，我々が使用する整合性のアルゴリズムの多くで用いられる．キャッシュマネージャは，ファイルの データ部分とステイタス部分を独立にキャッシュする．

ファイルの fid は，ファイルサーバーからファイルのデータ部分や状態に関する部分をいつでも検索できるようにするために必要となる重要な情報である．ファイルは不変ではないので，別の時間には別のデータを含むこともある．しかしながら，そのファイルの fid とデータバージョン数を確かめることによって，その内容を一意に識別することができる．

AFS を設計する過程において，パス名をそのファイルの fid に変換するのに要する処理時間が問題となると考えられた．. 特に，パス名を基本にした広範囲に渡る計算を処理するのに使える計算資源を，ファイルサーバーが十分に持てないのではないかと考えられた．このような理由か，パス名を取り扱う要求のすべてと共にパス名に関する情報のすべてをファイルサーバーインタフェースの外に置くことにされた．サーバーは，キャッシュマネージャがパス名から割り出した fid によってのみファイルを識別する．パス名に関する情報のすべてはキャッシュマネージャが管理する．キャッシュマネージャは，AFS の構成要素であり，サーバーが提供する自由に移動できるボリュームの集まりをつなげて一つのツリーにする．それは，パス名の中で AFS のマウントポイントに出会ったら，それを翻訳することによって行なわれる．

加えて，システムのどのボリュームがどのセルに属しているかを知っているのもキャッシュマネージャである．一人のユーザが異なるセルで持ち得るいろいろな識別子を追いかけることや，適切に認証された RPC 接続を正しい時に用いるのもキャッシュマネージャの義務である．すなわち，キャッシュマネージャには，一人のユーザに対して，一つのセル当たり一つの識別子の集合が与えられ，そのユーザについてあるセルを参照したときには適切に同一視できることを仮定している．

ボリュームロケーションサーバー

キャッシュマネージャは，どのサーバーがそのファイルを提供しているのかをどうやって知るのであろうか．ボリュームロケーションサーバーと呼ばれるプロセスが，この情報をキャッシュマネージャやネットワーク内の各所にある管理用プログラムに送っているのである．

キャッシュマネージャが，それまでにアクセスしたことのないボリューム内にのファイルを操作する必要が生じた場合，キャッシュマネージャはそのセルのボリュームロケーションサーバーにコンタクトをとって，どのサーバーがそのボリュームにアクセスできるかという情報を手に入れる．(現在のところ，read-only のボリュームだけが複数のサイトに存在してもよい）キャッシュマネージャはこの情報をキャッシュしているので，ボリュームを参照する場合のほとんどはボリュームロケーションサーバーとコンタクトをとる必要がない．場合によっては，オペレータコントロールによってボリュームがサーバー間を移動する．そのような場合に，キャッシュマネージャがそのボリュームを移動前の場所に参照しに行くと，特別なエラーコードが発生し，そのボリュームの移動先に関する情報をボリュームロケーションサーバーから手に入れるよう指示する．

オーセンティケーションサーバー

ファイルサーバーは，ファイルにアクセスにきたクライアントの認証を行う必要がある．AFS では基本的にに，クライアントのワークステーションは本質的に安全ではない，と考えている．つまり，物理的に操作できないような計算機センターのワークステーション上で実際にどのようなソフトウェアが走っているかを知る方法はないということである．したがって，そのようなワークステーションの UNIX のカーネルに対して，ファイルにアクセスしようとしているユーザの身元を確認することはできない．

それに加えて，このような特殊な環境下では，ワークステーションやパーソナルコンピュータがネットワークの中を自由に行き来できる．以上のような理由から，ネットワークを介してパスワードを送るというようなことは認めないものとした．イーサの中を飛び交うパスワードを捕まえるようなプログラムを書くことなど誰にでも簡単にできる．その代わりとして，ネットワーク環境において相互の認証を確立するために Needham と Schroeder によるアルゴリズムを採用した認証サーバーが書かれた．AFS 3.0 では，オリジナルのオーセンティケーションサーバーを MIT のアテナプロジェクトの Kerberos を採用した認証システムに置き変えた．

Kerberos はかなり複雑ではあるが，その背後にある基本的な考え方は

極めて単純である．クライアントは,（パスワードのような）共有秘密情報をネットワークを介して送ることによってではなく，共有秘密情報をも知っているサーバーからの暗号化された要求に答えることによって身元確認を行なっている．すなわち，サーバーの RPC パッケージが共有秘密情報で暗号化を行ない，それをクライアントに対して送る．クライアントの RPC パッケージがその要求を受け取り，暗号を外し，要求に答え，結果を共有秘密情報で暗号化してサーバーに返す．同じ要求が繰り返されることは決してないので，ある要求に対して行なった対応をその後の要求に対して単純に繰り返すことはあり得ない．

保護モード

ファイルサーバーは，認証機構によって RPC インタフェースの向こう側に位置するクライアントの身元を確認する．そして，ファイルサーバーは，システム内の各ディレクトリに付随したアクセス制御リストを調べることによって，いろいろなディレクトリやファイルに対するクライアントのアクセス権を規定する．このリストは，ディレクトリ自身およびそのディレクトリに含まれるすべてのファイルのアクセス権を規定している．アクセス制御リストは，対からなる配列である．対の最初の項目がユーザ名およびグループ名で，二番目がそのユーザやグループに関する権利情報である．

ディレクトリに対して記されているアクセス権は，次にあげたものの内の一つかいくつかである．

- lookup

  ディレクトリ内の名前の参照を許す．

- insert

  ディレクトリに新たなエントリを追加することを許す．

- delete

  ディレクトリからのエントリの削除を許す．

- administer

  ディレクトリのアクセス制御リストの修正を許す．

ディレクトリ内のファイルに対するアクセス権は，以下のようである．

- read

  ディレクトリ内のどのファイルについても読むことを許す．

- write

  ディレクトリ内のどのファイルについても書き込をを許す.

- lock

  ディレクトリ内のどのファイルについてもロックをセットすることを許す.

さらに AFS は，特別な方法を用いて UNIX のオーナ保護モードビットを解釈し，オーナの write や read のビットの立っていないファイルを保護する．例えば，ユーザ kazar のホームディレクトリのアクセスリストは次のようになる．

Access list for
/afs/andrew.cmu.edu/usr7/kazar is
Normal rights:
System:AnyUser rl
kazar rlidwka

これは，グループ System:AnyUser （システム内の全員からなっている）のメンバは kazar のホームディレクトリにあるファイルを読んだり，ディレクトリの中の名前を参照することができる．しかし，kazar として認証されたユーザだけがディレクトリやその中の任意のファイルを更新できる．

AFS ユーザは，ユーザグループ全体に対して特別の権利を認めるために，新しいアクセス制御グループを作ることが許されている．グループの構成員のデータベースは，プロテクションサーバーと呼ばれるサーバーによって維持され，このサーバーは「このグループのメンバーは誰か？」と「このユーザは，どのグループに属しているか？」という関連する 2 つの問いに答えることができる．このサーバーは，新しい AFS ユーザアカウントを作るための機能も備えている．このサーバーは，キャッシュマネージャや数多くのシステム管理用ユーティリティと同様に，RPC を経由してファイルサーバープロセスからアクセスされる．

オペレーションサーバー

これまでに挙げたサーバーはすべて，個々の UNIX のプロセスとして実現されている．これらのプロセスを管理する手助けとして，AFS はプロセスマネージャを提供している．このサーバーの仕事は，システム内の他のサーバーを制御したり，新しいプログラムをファイルサーバーマシン上にインストールしたり，失敗したプロセスを再起動したり，再起動するプ

ロセスのステイタスの報告を行なったり，それ以外にはシステムの操作を補助したりすることである．

キャッシュの整合性とパフォーマンス

分散ファイルシステムによってもたらされる整合性の保証は，そのファイルシステムの可能な実現方法に強い影響を与える．ハイパフォーマンスの分散ファイルシステムを構築する際には，まず整合性に関する適切なセマンティクスが求められる．それに加えて，整合性に関するセマンティクスは，アプリケーション，とりわけ分散アプリケーションの構築における分散ファイルシステムの有用性にも影響してくる．もしその分散ファイルシステが，分散アプリケーションの構築に十分な整合性を持ち合わせていないと，記憶領域や共有について分散ファイルシステムを用いることができずにシステムは失敗に陥ってしまう．

AFS では，キャッシュマネージャへの呼び出しは，キャッシュから最新のデータを返す必要がある．AFS は，キャッシュ内に無効なデータが含まれている場合に，ファイルサーバーがそれをクライアントのワークステーションに知らせるコールバックシステムを使っており，キャッシュマネージャの各呼び出しについてサーバーをポーリングしたり，単純にキャッシュのデータが最新のものであると仮定したりしない．ファイルサーバーが，ファイルのデータやステイタスをクライアントのワークステーションに送る場合，サーバーがそれらを変更する前にワークステーションに知らせるものとする，という取り決めを行なっても良い．（これは，他のユーザがファイルを書き換えに行く際に起こる）このような取り決めのことを，コールバックと呼ぶ．特定のファイルが変わったことを，適当なワークステーションに通知する プロセスのことをコールバックをブレイクすると言う．またこれは，そのファイルの内容や状態が変わる前に行なわれなければならない．

ワークステーションがキャッシュ内のあるファイルについてコールバックを終わらせない限り，情報が古いものかどうか気にすることなくそのファイルのデータやステイタスを読むことができる．これによってワークステーションがファイルサーバーにかける負荷を著しく減少させ，キャッシュの整合性の保証を弱めることもない．また，コールバック自体は本質的にシンプルな機能であるが，そのインプリメントはたいへんなものである．問題は二つの領域で生じる．一つはファイルサーバーの呼び出しとその同じファイルサーバーからのコールバックをブレイクするメッセージとの同期，もう一つはコールバックの取り決めをブレイクするときの通信の失敗である．

通信

AFS はマシン内部およびマシン間でのプロセス間通信のすべてに，Rx と呼ばれる RPC 機構を採用している．RPC は良く理解された通信モデルであるが，Rx にはユニークな特徴がいくつか存在する．

1. コア Rx パッケージとその認証モジュール間のインタフェースはそれ自体高度にモジュール化されているので，新しい認証モジュールの定義が容易であり，複数の異なる認証構成の共存が可能である．例えば，CMU の ITC で現在稼働しているファイルサーバーは，我々の古い認証のトークンと Kerberos の認証のチケットの両方をサポートしている．

2. Rx コールのパラメータとして大きさに制限のないバイトストリームを渡すことができるような高度な RPC インタフェースを，Rx は持っている．このような呼び出しが行われる場合，クライアント側の Rx コールは二つの部分に分かれて現れる．一方は入力パラメータを受渡しする部分で，もう一方は出力パラメータを返す部分である．この二つの部分の間に，呼び出し側は Rx ストリームに対しデータを読み書きできる．このデータはサーバーで呼ばれたプロシージャからも読み書きできる．結果として，リモートプロシージャコールで大きさに制限のないデータを受渡しするのに，非常に効率的で便利な方法である．AFS は，大きなファイルの塊（例えば，64KB）をキャッシュマネージャとファイルサーバーの間で転送する際には，この機構を用いている．

3. Van Jacobson による TCP の改良においてなされた幾つかの細かいチューニングを用いて Rx は設計されている．これによって Rx が，少なくとも現在の TCP 程度にネットワーク輻輳や可変速度の通信リンクに適用できるようになることが，期待されている．初期のテストでは，最初のバージョンの Rx が，500Kbps の長距離リンク上で約 400Kbps のデータ転送能力を持っていた．

相互操作性

AFS 3.0 のキャッシュマネージャは，NFS の実装と共存して，ワークステーションが NFS サーバーと AFS サーバーの両方にアクセスできるようにすることができるようになっている．さらに，CMU の計算機科学部において，NFS のプロトコルを用いて AFS のファイルを提供するサーバーが実装された．このサーバーによって NFS のクライアントは，ローカルエ

リアネットワーク上の中継機を介して広域ファイルシステムのサービスを受けることができる．システムは，NFS と AFS によって提供される最低レベルの機密性を提供する．

将来，UNIX ベースのどんなマシン上でも稼働して，しかもそのマシンから見えるようなファイルシステムを提供するようなエクスポートサーバーの構築が計画されている．そうした点においても，AFS のクライアントに対し，どのような分散ファイルシステムからでもファイルを提供することが可能にになるであろう．

第 3 章

光ディスクと光ジュークボックス

コンピュータの 2 次記憶装置に用いられているメディアは，現在はほとんどが磁気ディスクである．磁気ディスクは再書き込み可能であり，かつ他のメディアに比べて比較的高速であるため，広く使用されている．しかし，単位記憶容量に対するコストは比較的高い．また，ポータビリティも悪い．ここで用いる光ディスクは追記型であり，それほど高速ではないが，大容量で，ポータブルであるといった点では磁気ディスクより優れている．ここでは，光ディスク，多数の光ディスクを格納できる光ジュークボックスの特徴について述べ，それらを用いて提供することのできるサービスについて考察する．

3.1 光ディスク

ここで用いる光ディスクは，一度だけ書き込むことのできる Write Once Read Multiple(WORM) といった性質を持つものである．一般的には追記型と呼ばれるものである．即ち，一度書いてしまった部分は書き換えることはできない．そのため，以来磁気ディスクが用いられていた分野での使用は困難である．しかしながら，その容量は磁気ディスクと比較して非常に大きいといった特徴を持つ．

光ディスクの長所をまとめると次のようになる．

- 大容量
- 容量と比較して低価格
- 耐久性が高い
- 寿命が長い

容量は，5.25 インチの光ディスクの両面に記憶した場合一枚あたり 600～800MByte 記録することができる．光ディスクを一枚ずつフロッピーディスクのよう

形式	追記型
ディスク径	5.25 インチ
記憶容量	800MByte（両面）
ディスク回転数	1800rpm
アクセスタイム	85 msec（平均）
セクタ長	512byte/セクタ
記憶再生速度	5.5Mbps
ビットエラーレート	10^{-12}（訂正後）
インターフェース	SCSI
データ転送速度	最大 1.5MByte/sec
ディスクローディング時間	最大 4sec
ディスクアンローディング時間	最大 4sec

表 3.1: 光ディスクの仕様

に持ち運びできる耐久性を考えれば，この容量は非常に大きく最も大容量のフロッピーディスクと比較しても 60 倍以上で，耐久性はもっと高い．

逆に，短所は以下のような点である．

- 書き換えができない
- メディアの信頼性に若干の問題がある
- 速度が遅い

これらのうち，メディアの信頼性は改善されてきており，現在ではほとんど磁気ディスクと比較して遜色ない．速度の点ではシークに特に時間がかかり，かなり低速のハードディスクなみである．また，データの転送速度も若干低速である．

仕様の詳細は Table 3.1の通りである．

3.2　光ジュークボックス

光ジュークボックスはアーカイブファイルサーバーで使用するシステムである．多数のディスクを格納することができ，ディスクドライバにディスクをリモートにマウント，アンマウントして使用する．光ジュークボックスは次のような長所を持つ．

収容ディスク枚数	56 枚
内蔵ディスクドライブ	2 ドライブ
ディスクローディング時間	4sec（平均）
ディスクアンローディング時間	4sec（平均）
ディスクマウント時間	5sec（平均）
ディスクアンマウント時間	6sec（平均）

表 3.2: 光ジュークボックスの仕様

- 大容量
  単一システムで最大 45GByte，システムを組み合わせることによって最大 314GByte
  （最高 7 システムまで組み合わせることができる）
- 複数のディスクドライブによる高スループットの実現
- ディスクの入れ換えが容易

光ジュークボックスは以上のように非常に大きな容量を持つことができる．しかしながら，全てのディスクがディスクドライブにマウントされているわけではない．ディスクに読み書きをするためには，ディスクがディスクドライブにマウントされていなければマウントする必要がある．そのために必要な時間は，それ以外にアクセスをするために必要なシークタイム，回転待ち時間等と比較して，非常に大きくなってしまう．そのため，異なったディスクに次々とアクセスするような場合には非常に効率が悪くなる．

光ジュークボックスの詳細な仕様は Table 3.2の通りである．

3.3 システムの特徴

ここでは光ディスク，光ジュークボックスをアーカイブファイルサーバーの記憶装置として使用した場合の特徴を述べる．まず，利点となるものは次のような点である．

- 光ディスクの性質によるもの
  - 大容量
  - 追記性
    - 履歴が残る

    - ∗ 領域管理の容易さ
    - ∗ 読みだしの高速化

- 光ジュークボックスの性質によるもの
  - – オペレータによるディスクの入れ変え
  - – ディスク一枚毎の管理

追記型であるということは，二度と消すことができないことが非常に大きな欠点であることのように思われ，その利点は取り上げられることは少ない．しかし，二度と消すことができないため，一般に一度作製したら消去することの少ないないアーカイブファイル，ダンプファイルのためのメディアとしては非常に優れているといえる．人為的なミスによって，間違って消してしまうことがあり得ないからである．

また，追記型であることは領域の管理を非常に容易にする．消去でき，再書き込みのできるブロック型のディスクの場合，使用できる領域は連続でなくなり個別に管理しなくてはならない．そのため，現在使用できる領域を管理することは追記型の場合と比較して複雑になる．また，一つのファイルの内容を記憶する領域がディスク上に分散してしまった場合は，読み出しの効率は非常に悪くなるため，そのようなことがあまり起こらないように管理する必要がある．

以上のような消去，再書き込み可能なディスクの場合に考慮すべき問題は，追記型の場合は起こらない．一度書き込まれたブロックは再び使用可能になるということはあり得ない．書き込みは常に連続的に使用されていき，簡単にいえば，どこまで使用されたかだけを記憶しておけば良い．このように，連続的に領域は使用されていくため，読みだし時にセクターのシークといったことが頻繁に起こることはない．これによって，読み出しは高速に行なうことが可能になる．また，領域が分散してしまうことに配慮する必要もない．

光ジュークボックスは，光ディスクを容易に入れ換えることができる．もし，ジュークボックス内に光ディスクが入りきらなくなってきたら，人為的に，例えばオペレータが光ディスクを入れ換えて使用するということが考えられる．ジュークボックス内に格納していないディスクをも管理下に置くことによって，さらにジュークボックスの持つ容量は拡大する．

光ディスクはそれ自身を管理することも容易である．メディアとしての耐久性は比較的高く，例えばフロッピーディスク，ディスクパックなどのように管理に気をつける必要はあまりない．アーカイブファイルサーバーによって管理される必要のないディスクは取り外し，個別に管理されるこ

とが望ましい．しかし，ある光ディスクに様々な人のファイルが書かれてしまっている場合は，そのディスクが誰によって管理されるべきかはっきりしない．即ち，ディスクの所有者，管理者が明確になるようにディスクには書き込まれる必要がある．

3.4 提供されうるサービス

光ジュークボックスを使用し，ネットワークに接続された時に，提供されると望ましいと考えられるサービスをあげてみる．それらは，バックアップ的なサービス，要求に対して持っている情報を検索するようなデータベース的なサービス，管理的なサービス，バックアップをする際に補助的に利用するようなサービス等に分類することができる．

- バックアップ
  - ftp
  - rcp
  - rdump
  - NFS
  - nntp (ニュースの配布)
  - smtp （メッセージ又はメール・サービス）
  - ソフトウェアのバージョン管理
  - コピー
- データベース
  - ディレクトリサービス
  - SQL(System Query Language) 的インターフェース
- 管理的なサービス
  - アカウントの管理
  - オーセンティケーション
  - ディスクの登録
  - ディスクの保存期間
  - （自動的な）ソフトウェアのバージョン管理
  - ユーザのトランザクションの保存、管理

  - クォータ
  - ミラー
  - CD の使用のされかたについての管理サービス
    （パッキングのされかたなど）

- 補助的なサービス
  - コンプレッション
  - バッチ処理
  - 暗号化

これらの全てを最初からサポートすることはできない．バックアップ的なサービスの中で特に強い要求があり，また最初からサポートすべきだと考えられるのは

- ftp
- rdump

の二つでである．

ダンプは現在においても一つのファイルシステムに対して，磁気テープを数本用いて行い，磁気テープは交換しなければならない．そして，ダンプがとられるハードディスクは増加し続けているため，毎週インクリメンタルダンプをとるといったことは非常に苦痛になってきている．そこで，rdump をアーカイブファイルサーバーに対して使用することができれば，その様な苦痛から開放され，また毎週きちんとダンプをとることが可能になるため，環境を維持していくのに非常に役立つといえる．

ftp はインターネットワーク上でファイルのやりとりをする場合に，もっとも一般的に使用されるコマンドである．インテラクティブにファイルを転送できるものは他にないため，サポートすべきであると考えられる．

管理的なサービスは，サービスを正しく行なうためには必要不可欠なもの含んでいるため，それらは提供されなければならない．運営のために必要最小限なサービスは

- ユーザの管理
- ディスクの管理

バックアップ	ftp rdump rcp
管理	ユーザの管理 ディスクの管理
補助	バッチ処理

表 3.3: アーカイブファイルサーバーで提供されるサービス

といったものである．

補助的なサービスはそれだけでは意味をなさず，他のサービスと組み合わせて使用される種類のものである．その中で特に必要であると考えられるサービスは，バッチ処理である．インターネットワーク上で使用されるシステムが，そのシステムを使用する全ての人に対して使いやすいインテラクティブなサービスを提供することは難しい．アーカイブファイルサーバーでは，ディスクの交換等を含めて低速な記憶装置を使用するため，さらに難しいといえる．ところが，バックアップのサービスにおいて，ファイルシステムのダンプをとる場合はインテラクティブである必要はない．また，ファイル転送であっても，バックアップを光ディスク上にとるような場合では，インテラクティブである必要はない．このような場合，バッチ処理は非常に有効であるといえる．(インテラクティブな方法でなく，ファイルを転送できるコマンドとしては，例えば rcp といったものがある．)

これまで，提供されうるサービスと提供すべきサービスをあげてきたが，提供すべきサービスをまとめると Table 3.3のようになる．

これからの章では，提供されうるサービスも考慮しつつ，Table 3.3で示されるサービスを提供できるシステムの構築について述べていく．

第 4 章

アーカイブファイルサーバーの設計

アーカイブファイルサーバーは，光ディスク，光ジュークボックスといった以来の一般に使用されてきた磁気ディスクとは異なったメディア，記憶装置を用いる．これらは追記型であるとか，アクセスに時間がかかるといった点で，磁気ディスクとは大きく異なっている．また，インターネットワーク上で使用されることから，以来扱われなかったような多くの異なった背景を持つユーザーをサポートする必要がある．そこで，アーカイブファイルサーバーを設計する上で，ポイントとなる点，考慮すべき点について，この章では考察する．

4.1　ファイルシステム

アーカイブファイルサーバーは，光ディスクを主な記憶メディアとして使用する．ユーザーによって書き込まれたファイルは，基本的にすべて光ディスク上に書き込まれる．アーカイブファイルサーバーの管理するファイルはユーザーのファイルの他に，ユーザーを登録したファイル，ネットワーク上の他のホストを登録したファイルなど，システムを管理していくために必要なファイル，システムを動かすためのソフトウェア等がある．これらの中には時間とともに変化していき，また頻繁に参照されるため光ディスク上に作られるべきではないものもある．これらのファイルを記憶するためのメディアとしは，書き換え可能，かつ高速な磁気ディスクが適していると，考えられる．

このようにアーカイブファイルサーバーは，光ディスクと磁気ディスクを組み合わせて，ファイルシステムを構成するのであるが，それぞれはどのようにファイルシステム上に位置付けられ，どのように使用されるべきなのであろうか．

4.1.1 光ディスク，磁気ディスクのファイルシステム上での位置付け

光ディスクの，ファイルシステムを構成するメディアとしての位置付けを考える際のポイントは

- 追記性
- 低速性
- ポータビリティ

である．

追記型であるために，頻繁に書き換えられる，または書き加えられる可能性があり，古いものは必要なくなるようなファイルは光ディスク上に記録されるべきではない．

光ディスクは磁気ディスクと比較すると低速である．そこで，頻繁に利用されるファイルは光ディスク上に存在すると，読み出すのに非常に時間がかかる．そのようのなファイルは光ディスク上に記録されるべきではない．しかし，ユーザーのファイルは基本的にすべて光ディスク上に記録される．そのため，これは効率の問題と深く関係しているが，磁気ディスク上に読み出されキャシングされるということが考えられる．

ポータビリティは，光ディスクにどのようにユーザーのファイルを詰めていくかという，パッキングの方法に依存してくる．理想としては，一枚のディスクには同一ユーザーのファイルだけが存在するというのが望ましいが，現実にはこれは不可能であろう．ということは，あるユーザーのファイルは複数のディスク上に分散して格納されている．これらのファイルは，そのユーザーのディレクトリの下に存在するべきなので，そこにあるように見せかける必要がある．

磁気ディスクは，光ディスクが追記性を持ち，また比較的低速であるのに対して，

- 高速
- 消去，再書き込み可能

といった特徴を持つ．

高速であるため，よく読み出されるファイル，ディレクトリは，磁気ディスク上に置かれ，キャッシュされるべきであろう．

磁気ディスクにおいては，一度書き込まれた部分は何度も書き換えることが可能である．また，細かく書き加えていくことも可能である．これ

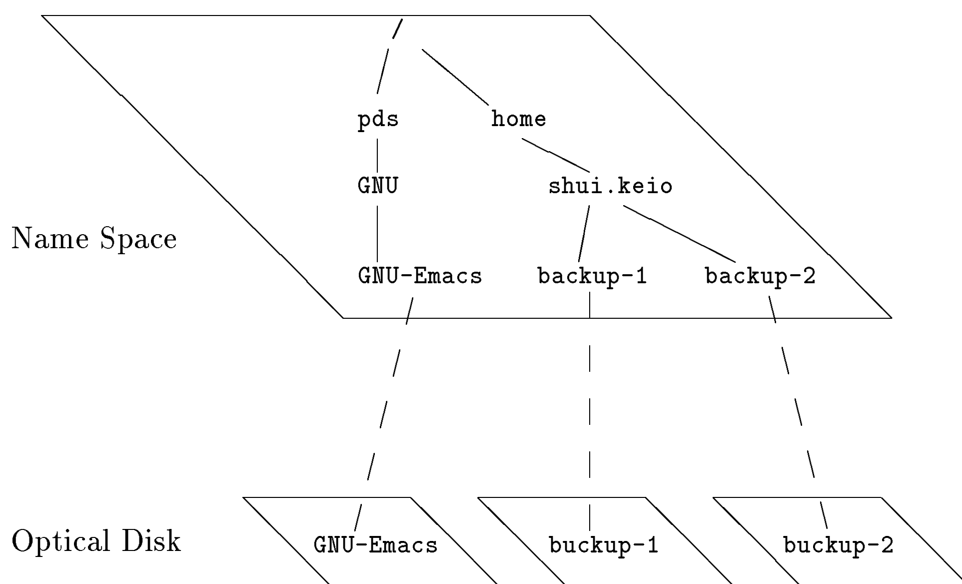

図 4.1: ネームスペースから光ディスクへのマッピング

は特にディレクトリを管理するためには重要で，光ディスク上にディレクトリ構造を作成することは難しい．

以上のことから，ユーザーのファイルは光ディスク上に記録するが，補助的に磁気ディスクを使用し，ディレクトリ構造は磁気ディスク上に作成する，という方法がとられるべきである．即ち，ネームスペースは磁気ディスク上に作られ，ユーザーのファイルは光ディスクにマッピングされる (Figure 4.1)．

4.1.2　ジュークボックスの外のディスクの取り扱い

光ジュークボックスの中に，格納できるディスクの枚数には限りがため，すべてのディスクを格納できるとは限らない．扱いたいディスクの枚数が，光ジュークボックスに格納できるディスクの枚数を越えてしまったら，ジュークボックスに格納できないディスクが出てくる．それらのジュークボックス内に格納できないディスクは，どのように取り扱われるべきであろうか．

ジュークボックス内に格納されていないディスクをアクセスするためには，誰かの手によってジュークボックス内に格納されなければならない．

そのためには，さらに時間がかかり，さらに低速になる．もしジュークボックスにディスクをいれるスペースがないとしたら，どれかディスクを抜いて，新しくディスクをいれる必要がある．アーカイブファイルサーバーが，ジュークボックス内に格納されているディスクしか管理対象にいれないならば，各ユーザーは，どのディスクにどのファイルが入っているかを知るには，一々ディスクを格納しなければならず，またどのディスクがアーカイブファイルサーバーに格納させられるかも，各ユーザーが知っている必要がある．

ジュークボックスの外にあるディスクもアーカイブファイルサーバーが管理する対象にいれるとすると，どのディスクが現在使用できるのか，どのディスクが今外に出されてしまっているのか，といったことについてユーザーが管理する必要はなく，アーカイブファイルサーバーから情報を得ることができる．

さらに，ジュークボックスの外にあるディスクも，格納されているディスクも同様にユーザーから見えるとしたら，どうなるであろうか．そうなれば，ユーザーはディスクの所在について全く意識する必要はなくなる．しかし，速度の点で問題が残る．あるジュークボックスの外に出てしまっているディスクにはいっているファイルのリストを出したい時，そこで長時間待たされるというのでは，うかつにファイルのリストも見ることができない．この問題は，ファイル名だけはキャッシュしておく，という方法で解決できる．これによって，ほぼユーザーはディスクの所在について意識することはなくなる．

4.1.3 ユーザのファイル管理

ユーザーのファイルは，ユーザーのホームディレクトリの下に置かれる．ユーザーのファイルは，いくつかの光ディスク上に分散しておかれている．これらのファイルは，ユーザーにどのように見えるべきであろうか．ここで，二つの方法が考えられる (Figure 4.2も参照).

1. どのディスクにはいっているかに関係なく，ホームディレクトリの下にあるように見える

2. どのディスクにはいっているかわかるように，各ディスク別にディスク名をつけたディレクトリを作る
   そのディスクにはいっているファイルはそのディスクの名前のついたのディレクトリの下に見える

光ディスクの存在を，ユーザーに意識させないという点では，1の方法の方が優れている．いくつかのファイルは同一のディスクに集めていれた

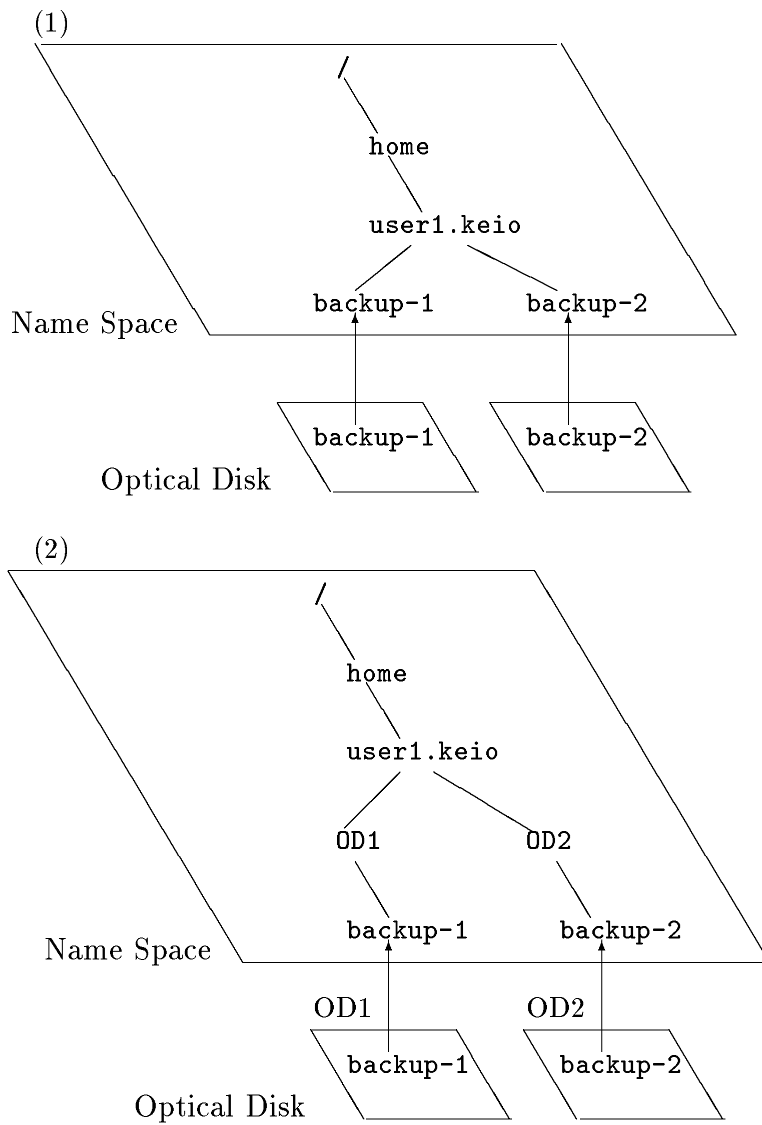

図 4.2: ユーザーのファイル管理

方がいいという場合は存在するが，そのような時，1 の方法では，どのファイルがどのディスクにはいっているのかそのままではわからないので，どのファイルはどのディスクにはいっているかを知らせる方法を提供する必要がある．

そこで，2 の方法のように，ホームディレクトリの下に光ディスクの名前をつけたディレクトリを置くようにすれば，どのファイルはどのディスクにはいっているかは，明らかになる．また，このようにすることによって，ファイルを記録するディスクの指定を明らかにすることができる．

光ディスク上には，ディレクトリは作りにくいので，光ディスク別のディレクトリの下には，ディレクトリは作らない．

4.1.4 保護

ファイルは，アクセスから保護される必要がある．許可を得たユーザーのみが，そのファイルをアクセスできるようにしなければならない．十分に保護をすることができなければ，ユーザーのプライバシーを保つことはできない．ところが，アーカイブファイルサイバーは光ディスクという，今迄とは異なった特徴を持つメディアを使用し，インターネットワーク上で使用されるため，様々なユーザーからアクセスされる．そのため，現在 UNIX のような少数を対象としたオペレーティングシステムでとられているような保護の考え方とは，異なった考えをする必要がある．そこで，ここではアクセスからのファイル，ディスクの保護について，どのように対処すべきか考察する．

アクセスの型

一般的なファイルシステムでは，ファイルに対するいくつかの異なったアクセスの型に対して，制御されたアクセス機能を提供している．UNIX においては，

- read
- write
- execute

を，別々に制御することができる．これらは，普通の再書き込みが可能な磁気ディスクをメディアとして使用しているような場合は意味がある．

アーカイブファイルサーバーの場合，“wirte”,“execute” は必要なのであろうか．“write” については，次のことが言える．

1. アーカイブファイルサーバーは，本来バックアップファイル，ダンプファイルといったファイルを対象としている．これらのファイルは再書き込みされるような性質のものではない．そのため，“wirte” は意味を持たない．

2. ファイルへの書き出し，再書き込みは，もとの情報が書き込まれていたディスク上のブロックについてみれば，それらが消去されることになる．しかし消去することは，追記型では不可能であるので，“write” は意味を持たない．

一度書き込まれた部分を無視するようにすれば，2 で言われていることは正しくない．しかし，そのためには，一度書き込まれた部分は無駄となってしまう．そこまでコストがかかるならば，1 の理由もさらに考慮すると，“wirte” はこの場合意味がないといえる．

“execute” に関しては，アーカイブファイルサーバーの管理する対象であるファイルの性質からいって，本質的に意味がないといえる．

したがって，アーカイブファイルサーバーの提供する必要のあるファイルに対するアクセスの型の制御は “read” だけである．

光ディスクを表すディレクトリの場合はどうであろうか．再び，

- read
- write
- execute

の三つについて考えてみる．UNIX においては，“read” はそのディレクトリを読めるかどうか，“write” はそのディレクトリに書き込めるかどうか，“execute” はそのディレクトリをサーチすることができるかどうか，の許可を意味する．

アーカイブファイルサーバーにおいても，ファイルの作成時のみ，書き込みは行なわれる．書き込みは，光ディスクに対して行なわれる．そのため，“write” のパーミッションを得るユーザーは，その光ディスクの所有者達ということになる．

“execute” は，UNIX においてはコマンドを他のユーザーに使用させても良いが，そのコマンドがあるディレクトリを見せたくはないような場合に使用される．しかし，アーカイブファイルサーバーでは，実行可能形式のファイルは扱わないため，これは意味がない．ディレクトリの “execute” には，そのディレクトリへ移動することができるかどうかの意味もある．ディレクトリを読むことはできるが，そこへ移動することも，そのディレ

クトリの下に存在するファイルを読むこともできないというのでは，あまり意味はない．そこで，“read” に “execute” の持つディレクトリへの移動の許可の意味も持たせることにする．

従って，光ディスクを表すディレクトリの場合，“read” および “wirte” を，アクセスの型の制御として提供する必要がある．

誰に許すかの指定

ファイルは，他のユーザーからのアクセスから保護されなければならない．では，誰に対してアクセスを許可するかは，どの様にして指定すべきであろうか．UNIX も含めて，多くのオペレーティングシステムではユーザーを三つの分類：

- owner （所有者）
- group （グループ）
- other （その他すべてのユーザー）

にわけ，識別する．しかし，インターネットワーク上での使用を考えた場合，この方法で十分であろうか．

インターネットワーク上では，ユーザーの属するオーガニゼーション，グループ等によって，より細かくしていできる方が望ましい．自分の属している組織のユーザーには見せてもよい，大学関係者には見せてもよい，といった要求は十分に考えられる．そこで，アーカイブファイルサーバーはこれに対処すべきで，十分な情報を持つアクセスリストを設定できなければならない．

単位

アクセスからファイルを保護するに当たって，どの単位でパーミッションを指定することができるかを考えてみる．UNIX おいては，アクセスパーミッションはファイル単位で指定することができる．UNIX において，これを指定するために必要な情報は，

- 所有者のユーザー ID
- 所有者のグループ ID
- ファイルのアクセスモード

と，僅かである.（それぞれに 16 ビットとっても，6 バイトである.）

アーカイブファイルサーバーにおいては，より細かいアクセス制御をすべきなので，これより遥かに多くの情報を必要とする．各ファイルがそのアクセスリストを持つとすると，

4.1.5　一枚の光ディスクの持つアトリビュート

一枚の光ディスクは，ファイルだけでなく，そのディスク自体の情報を持っている必要がある．また，ディスクに書かれたデータについての情報も持っていなくてはならない．一枚の光ディスクが持ってる必要のある情報はつぎのものである．

- ディスクの名前
- ディスクに含まれるファイルの種類
- アクセス権リスト
- 常駐かどうか
- インデックスリストの領域（どこからどこまで）
- インデックスリストの空き領域の開始アドレス
- ファイルのデータ領域
- ファイルの空きデータ領域の開始アドレス

しかし，これらのうち，すべてをディスク上に記録しておくことはできない．ファイルをディスクに書き込むたびに変化するパラメータは，ディスク上に記録しておくことは，不可能である．そのようなパラメータは，書き込まれたコンクリートな情報から，取り出す必要がある．

次のものが，実際にディスクに書き込まれるデータである．

- ディスクの名前
- ディスクに含まれるファイルの種類
- アクセス権リスト
- 常駐かどうか
- インデックスリストの領域
- ファイルのデータの領域

インデックスリスト及びファイルのデータ領域の空き領域の開始アドレスは，既に書かれているインデックスリストから計算される．

4.1.6 ファイルの持つアトリビュート

ファイルは，論理的には単なるデータの集まりであり，例えば UNIX では，バイトの並びに過ぎない．ファイルには，名前が付けられ，その名前で参照される．他に，ファイルは，作成者，作成日時，長さ等の属性を持つ．

ファイルの属性として一般的に持たれるが，光ディスクの場合持たれることのない，または持てない属性としては，

- 最後にアクセスされた時間
- 最後に変更が加えられた時間

といったものがある．これらは，アクセスされるたびに変更が加えられるため，光ディスク上に記録することはできない．

光ディスク上に書かれるファイルの持つ属性は，次の通りである．

- ファイルの名前
- ファイルの所有者
- ファイルの作成された日付
- アクセス権 (パーミッション)
- ファイルのサイズ
- ディスク上のアドレス

4.2 効率

光ディスク，光ジュークボックスは，一般に使用される磁気ディスクと比較して，非常に低速である．光ジュークボックスを，磁気ディスクと同じように使用すると，ディスクの入れ換え等に時間がかかり過ぎ，使用に耐えられないと思われる．そこで，何らかの対策をすることによって，その低速性をカバーする必要がある．

低速性を補うための方法として，ここでは

- 磁気ディスクをキャッシュとして使用
- ディスクへのアクセスのスケジューリング

を，考える．

4.2.1　キャッシュ

高速な CPU を持つコンピュータは，主記憶はその CPU のスピードにおいつかないため，主記憶にアクセスする時は CPU は，主記憶からデータが読み出されるまで待たなければならない．そこで，容量は小さいけれども，非常に高速なメモリを用意し，これから読み出されると思われるデータを，そこにおいておくことによって，CPU を高速に動作させようとする方法が用いられている．ここで，主記憶と比較して，容量は小さいけれども非常に高速なメモリをキャッシュという．

同様に，光ディスク，光ジュークボックスと比較して高速な磁気ディスクを，キャッシュとして使用することが，考えられる．頻繁に参照されるファイルは，磁気ディスクに一時的に記録しておき，そのファイルがアクセスされたら，磁気ディスク上にキャッシュしてあるファイルをアクセスすることによって，それらのファイルは高速に読み出すことができる．

光ディスク上にファイルを書き出す時も，一度磁気ディスク上に書きだし，その後，光ディスク上にコピーすることによって，ファイルの書き出しは早く終了することができる．

また，ネットワークを通じてファイルが送られてきた場合，光ディスクへ書いている途中で，コネクションが切れ，正しくファイルが転送されなかったとすると，途中まで光ディスク上に書かれたファイルは無効となり，書かれた領域は無駄になってしまう．そこで，一度ファイルを磁気ディスク上に記録することによって，光ディスクの追記性による無駄を無くすことができる．

4.2.2　スケジューリング

磁気ディスクによるファイルのキャッシングは，光ディスクとの間に高速な磁気ディスクをいれることによって，読み書きを高速に終わらせようというものであった．光ジュークボックスをアクセスする時，非常に時間がかかるのはディスクが

- ドライブ上になく，目的のディスクをドライブにマウントしなければならない
- 光ジュークボックスに格納されてなく，目的のディスクを格納しドライブにマウントしなければならない

場合である．従って，これらの動作をなるべく起こさないようにすれば，全体としての動作は早くなる．そこで，光ディスクへのアクセス要求をスケ

ジューリングすることによって，これらの時間の消費が大きい動作を減らし，スループットをあげようとするものである．

光ジュークボックスへのアクセスのスケジューリングをするにあたって，アクセスとしては

- READ
- WRITE

の二つがあるが，これらのうち “READ” アクセスの方が多いだろうと思われる．また，光ディスクへ “WRITE” する場合は磁気ディスク上に一度書かれる必要があるため，ユーザーは直接アクセスの遅さの影響を受けることはない．

したがって，スケジューリングは “READ” の場合を，主に考慮して行なわれるべきである．

4.3 ユーザーインターフェース

アーカイブファイルサーバーは，ユーザーのファイルを受け取ってディスク上に書き込んだり，ディスク上からファイルをユーザーへ送ったりする．そのために，ユーザーはアーカイブファイルサーバーに対してあるコマンドを送ることによって，ファイルの出し入れを行なう．これらは，アーカイブファイルサーバーがユーザーに対して提供する機能を用いて行なわれる．そこで，ここでは，アーカイブファイルサーバーがユーザーに提供する機能の，ユーザーインターフェースについて考える．

ユーザーインターフェースを基本的に分類すると，

- インテラクティブな方式
- バッチ式

の二つに分けられる．バッチ式は，ジョブを一まとめにして送り，結果を待つ方法である．インテラクティブな方式は，文字どおり会話的に処理を進めていく方式である．ここでは，この二つの方式について考えてみる．

4.3.1 インテラクティブな方式

インテラクティブなユーザーインターフェースのもとでは，コマンドは打ち込まれたらすぐに解釈，実行され，結果が出力される．UNIX,（少し古いが）TOPS といったユーザーフレンドリーなオペレーティングシス

テムでは，このようなインテラクティブなユーザーインターフェースが用いられている．

インテラクティブなユーザーインタフェースのもとでは，コマンドを打ち込み，実行させてから，その結果が得られるまでの時間が，問題になる．あまりにも長過ぎると，非常に使い難く感じてしまう．一つのコマンドを実行し，結果を得るまでに数十秒から数分かかるとしたら，現在のワークステーションを使用しているようなユーザーにとっては，そのシステムは使いものにならないと感じるであろう．

アーカイブファイルサーバーにおいては，ディスクのドライブからの出し入れ，交換に平均 20 秒前後の時間がかかり，ディスクがジュークボックスの外にある場合は 1 分前後の時間がかかると思われる．すなわち，ディスクをアクセスするコマンドを実行するたびに，最大 1 分の待ち時間がありえる．このままでは，とても使い易いインターフェースを作ることはできない．

ファイルの操作で，ディスクにアクセスする必要があるのは，

- ファイルのリード
- ファイルのライト
- ファイルのリストの出力

である．これらのうち，実際にディスクにアクセスしなければ，結果を得られないのは，“ファイルのリード” と “ファイルのリストの出力” である．“ファイルのライト” は，磁気ディスクへ一度キャッシュする必要があるため，光ディスクへの書き込みは後で行なえば良い．

“ファイルのリストの出力” は，“ファイルのリード” と比較して，また他のオペレーションと比較しても頻繁に行なわれると思われる．また，ファイルの属性だけ取り出した場合，その大きさは小さいものである．そこで，属性のみを磁気ディスク上にキャッシュしておき，リストを出力する時はそれらを持ちいる．それによって，光ディスクをアクセスする必要がなくなるので，光ディスクへのアクセスは減り，リストの出力は高速に行なわれる．

“ファイルのリード” に関しては，キャッシュが効果的に用いられること以外に，特に高速化の方法はない．

4.3.2　バッチ式の処理

バッチ式の処理では，一度に行ないたい処理をまとめて送り，それらのジョブはスケジューリングされ，順番が回ってきたら処理される．この

場合，ある時間内に終了すれば良く，インテラクティブなインターフェースに比べ，その応答速度は遅くとも良い．また，一度に行ないたい処理をまとめて送ることができるため，ある時間になったら自動的にバックアップをとるなどの処理も行ないやすい．そのため，アーカイブファイルサーバーのインターフェースとしては，インテラクティブな方式よりも，優れているといえる．

第 5 章

実装

前章で考察された設計方法に基づいて，本章では実装方法について考察する．

5.1 全体的な構成

従来のハードディスクの特性とは非常に異なっている光ディスクの特性，ジュークボックスの存在によって，カーネル内にファイルシステムを作ることは非常に難しい．従って，ここでは光ディスクのためのファイルシステムを，カーネルの外に構成する．カーネル内には，光ディスク，ジュークボックスを操作するための必要最小限の機能のみを持たせ，光ディスクファイルシステムはそれらを利用して構成される．光ディスクを使用するアプリケーションプログラムは，光ディスクファイルシステムの提供する機能を用いる．(Figure 5.1)

この方法では，UNIX のファイルシステム内に組み込まれることはないため，それまでのプログラムを用いて，光ディスク上のファイルを，UNIX ファイルシステム内のファイルと同じようにアクセスすることはできない．アクセスするために，新しいプログラムが必要となってくる．しかし，逆に UNIX ファイルシステムのセマンティクスにとらわれる必要はなくなり，光ディスク，ジュークボックスの特徴をいかすようにファイルシステムを作ることが可能になる．

カーネル内にファイルシステムを組み込まないため，光ディスクファイルシステムは，スケジューリングを行なうために，カーネル内のスケジューリングを利用することができない．そのため，光ディスクファイルシステム内に，独自のディスパッチループを持たなければならない．各アプリケーションが，ファイルシステムを同時に使用するためには，

1. 光ディスクファイルシステムを，ライブラリとして提供し，その中に相互排除の機能を設ける

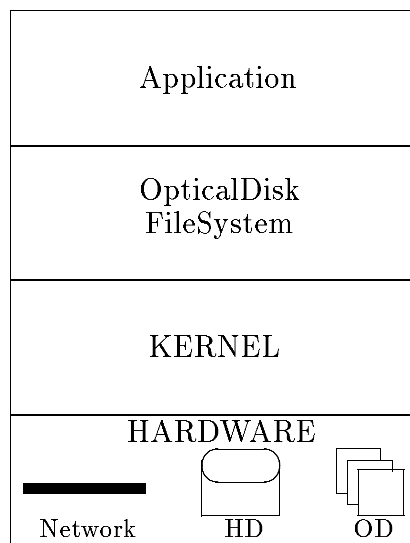

図 5.1: アーカイブファイルサーバーの構成

2. 光ディスクファイルシステムを一つのプロセスとして実現し，各アプリケーションはそのプロセスと交信することによって，ファイルシステムの持つ機能を利用する（クライアント・サーバー型）

といった方法が考えられるが，ここでは 2 の方法を用いる．1 の方法では，ライブラリの大きさが非常に大きくなり，各アプリケーションのサイズを非常に大きくしてしまう．また，光ディスクファイルシステムの構造を変更したい時，そのライブラリを用いているアプリケーションは，再び新しいライブラリとリンクし直す必要がある．2 の方法では，それらの点は解決されている．

5.2 ファイルシステムの構成

光ディスクファイルシステムは主に，

- 光ディスクファイルサブシステム
- ファイルキャッシュ
- ライブラリインターフェース

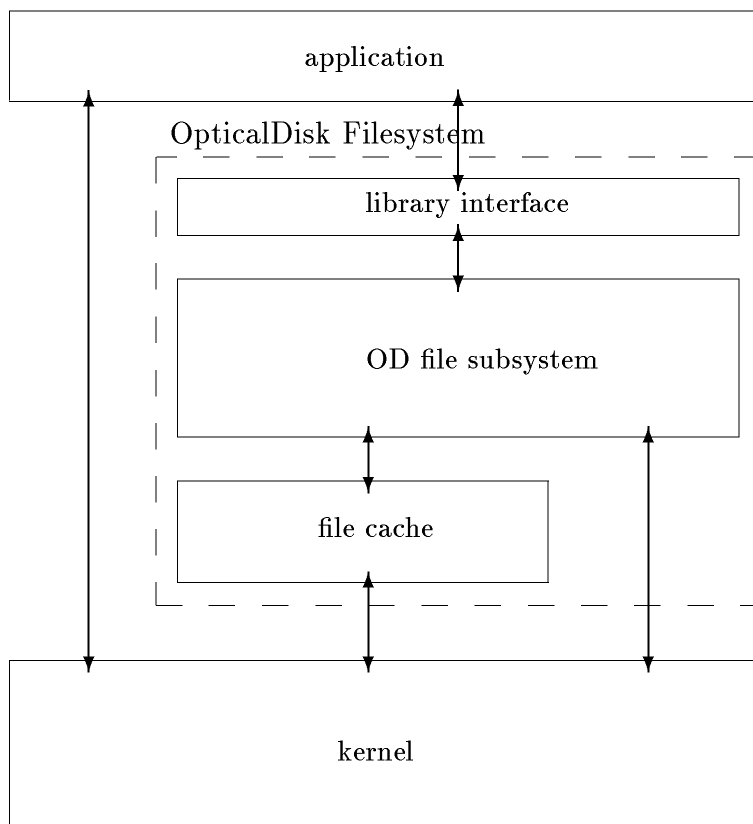

図 5.2: 光ディスクファイルシステムの構成

から構成される．(Figure 5.2)

光ディスクファイルサブシステムは，光ディスクファイルシステムを構成する最も大きな部分で，ファイルシステムの木構造の構成，光ディスクの管理等が行なわれる．

5.2.1 ファイルキャッシュ

ファイルキャッシュは，ファイルの出し入れを効率良く処理するために使用され，高速なハードディスクを用いて構成される．ファイルキャッシュは，ファイルアーカイブを一時的に記憶するためのものである．ファイルアーカイブは，普通のファイルが 1 〜 5KB の小さなファイルが多くあるのとは異なり，かなりの大きさを持つと考えられる．アプリケーションプログラムのソースファイルを，“tar” によってアーカイブし，“compress” によって圧縮したものでも，200KB から 1MB 程度の大きさを持つものは，多く存在する．例えば，エディタとして非常に良く使用されている “GNU Emacs” は，“tar” され “compress” されたもので 500KB である．

このような，大きなファイルをキャッシュし，効率を良くするためには，そのために用いるハードディスクは，十分な容量を持ち，また高速でなければならない．十分な容量を持たなければ，頻繁にアクセスされるファイルをハードディスク上に置くことができない．その結果，ハードディスク上にない，ファイルがアクセスされるたびに，光ディスクとハードディスクの間での，ファイルのコピーばかりが頻繁に起こり，キャッシュとしての役割を果たさないばかりか，返って効率を落すことになってしまう．また，十分に高速でなければ，キャッシュとして役に立たない．

5.2.2 ライブラリインターフェース

ライブラリインターフェースは，アプリケーションに対して，光ディスクを利用するためのインターフェースを提供する．アプリケーションプログラムは，このライブラリを用いることによって，容易にアーカイブファイルサーバーの機能を用いることができるようにする．ここでは，光ディスクファイルシステムと，その機能を用いるアプリケーションは，クライアント・サーバー型の形態を取るため，ライブラリはファイルシステムとの交信をサポートするものとなる．

ライブラリインターフェースは，次の関数から構成される．

- OD_Open_Server
  （光ディスクファイルシステムのサーバープロセスとの交信を開始する）

- OD_Close_Server
（サーバープロセスとの交信を終了する）
- OD_Open
（ファイルのオープンをする）
- OD_Close
（ファイルのクローズをする）
- OD_Create
（ファイルの作成をする）
- OD_Read
（ファイルの読み込みを行なう）
- OD_Write
（ファイルへの書き込みを行なう）
- OD_Lseek
（ファイルでのシークを行なう）
- OD_GetAttr
（ファイルのアトリビュートを得る）
- OD_GetFsStat
（ファイルシステムのステータスを得る）
- OD_ChangeDir
（カレントディレクトリを移動する）