TM-1254

PIMOS ファイルシステムの デイレクトリ管理方式

森　健

March, 1993

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03)3456-3191 ~5

Institute for New Generation Computer Technology

PIMOSファイルシステムのディレクトリ管理方式

森　健

概要

PIM(Parallel Inference Machine)は、ICOTが開発した並列推論マシンである。我々は、PIMOS(PIM Operating System)ファイルシステムをPIMのような疎結合並列マシン上で効率良くファイルを管理するシステムとして開発してきた。

本稿では、PIMOSファイルシステムのディレクトリ管理方式について述べる。疎結合並列マシン上で高速にファイル名検索を行うために、分散キャッシングを用いて各ノード上でファイル名検索を行うようにした。また、ディレクトリへのディスク領域の割り当て、解放の際に、領域の使用効率と処理速度のバランスを考慮した割り当て、解放方式を考案した。さらに、ディレクトリ更新操作時に原子的な更新をひとまとめにしたログを取り、システム障害発生時にもディレクトリとその内容が失われないようにした。

1　まえがき

PIMOSファイルシステム[1][2]は、疎結合並列マシン上で効率良くファイルを管理することを目的として開発された、実験的なファイルシステムである。疎結合並列マシン上で動作することから、LANを利用して接続された複数の計算機上で動作する、分散ファイルシステムに似た性質を持っている。また、知識ベースシステムからのファイルアクセスの土台としての利用や、一般ユーザの疎結合並列計算機上でのファイルアクセスを伴うプログラミング実験にも利用されることから、実用に耐えるだけの十分な性能と、システム障害に対する耐久性を考慮して設計されている。

現在PIMOSファイルシステムは知識ベースシステムによって既に利用されており、これは第0.5版と呼ばれている。この版ではデータファイルのみの管理機能を提供しており、ディレクトリの管理機能は未実装である。このため各データファイルの識別は、論理ボリューム名と各論理ボリューム毎に一意な番号との組によって行われており、一般ユーザの使い勝手に不満があった。そこでPIMOSファイルシステム第1.0版では、ディレクトリ管理機能を導入してこの欠点の改良を計った。ディレクトリ機能の装備により、ユーザは各ファイルに名前を付けて識別し、さらにそれらを階層的に管理することが可能となる。この時、疎結合並列推論マシンに適した管理方式であることと、既存の版に整合性よく追加実装ができることに留意して、ディレクトリ管理方式を設計した。

2　設計目標

ディレクトリ管理機能を設計するにあたり、上述のように疎結合並列マシンに適し、かつ既存の版に整合性良く実装可能な方式であることを基本とし、さらに以下の三点を重点的な目標とした。

- パス名検索の高速化
- ディスク領域の効率的な使用
- システム障害への高耐久性

以下それぞれの目標について、目標とした理由と解決すべき問題点を述べる。

2.1 パス名検索の高速化

ユーザが操作対象ファイルの指定に使用したパス名は、ディレクトリ管理機能によって検索され、システムの内部表現に変換される。システムの内部表現は、前記したように論理ボリューム名と各論理ボリューム内で一意の番号とを組にした形で表される。この各論理ボリューム内で一意の番号のことをファイル番号と呼ぶ。またパス名の検索の際には、パスに含まれるファイル名を一段辿る毎に、そのファイルの保護情報を読みだしてユーザのアクセス権を調べる必要がある。通常、ユーザがあるファイルを操作する場合、操作は一回では終わらず複数回行われると考えられ、パス名検索は一回の操作毎に毎回必要となるため、それを高速に行うことはユーザのファイル操作に対するレスポンスの向上に大きな効果がある。

各ファイルのファイル番号は、そのファイルの親ディレクトリのデータ部に格納されており、必要に応じてディスク装置から読み込む必要がある。またファイルの保護情報も、同様にディスク装置から読み込む必要がある。ここで、パス名を一段検索する度にディスク装置にアクセスしていては、処理速度が非常に遅くなってしまう。特に疎結合並列マシンでは、ユーザが利用しているノードと、アクセスが必要なディスク装置が接続されているノードとが別々の場合があり、このような場合にはノード間通信が必要となって更に処理速度が低下してしまう。また、ディスク装置へのアクセスがあまり頻繁になると、同じディスク装置にアクセスしようとしている別の処理が待たされて、システム全体のスループットも低下してしまう。

このノード間通信とディスク装置への I/O を削減して、ユーザへのレスポンスとシステム全体のスループットを向上させるために、ユーザの利用しているノードのローカルメモリ上に必要なデータをキャッシュする、分散キャッシングが不可欠である。この時、パス名の解析を高速に行うためにキャッシュする必要のある情報は何か、あるキャッシュ内容の更新時にそれをどうディスク装置上に反映するか、同時に他ノード上の同じキャッシュ内容をどう無効化、もしくは更新するかといった点を考慮する必要がある。

2.2 ディスク領域の効率的な使用

ディレクトリはそのメンバの増減に伴って、使用するディスク領域の割り当て、解放が必要になる。基本的には割り当てはメンバが増加して領域が不足した段階で、解放はメンバが減少して領域に空きが生じた場合に行えばよい。しかし一般にディレクトリの使用領域は、メンバの削除によってあちこちに隙間ができている。メンバの追加を行う場合に、メンバを追加する領域をいつもディレクトリの使用領域の先頭から探すのでは処理速度が低下してしまう。一方、メンバの追加は常に使用領域の最後に追加するという方法もあるが、これではメンバの削除によって生じた隙間が埋まらず、領域の使用効率が悪くなってしまう。メンバの削除の場合には、領域解放をあまり高い頻度で行うと解放処理のオーバーヘッドが大きくなって、処理速度が低下してしまう。しかし解放を行う頻度を低くしすぎると、今度はディレクトリは空き領域を多く抱え込むことになり、領域の使用効率が悪くなる。このようにメンバの追加、削除を行う場合には、処理速度とディスク領域の使用効率のバランスを十分に考慮して、メンバの追加、削除とそれに伴うディスク領域の割り当て、解放アルゴリズムを決める必要がある。

2.3 システム障害対策

PIMOS ファイルシステムが稼働する並列推論マシン PIM は、実験機としての性格が強いため、一般の商用機と比較してシステム障害が発生する可能性が高い。そこで、PIMOS ファイルシステムではシステム障害への対策として、ログ管理機能を実装している。これは、ファイルの更新操作があった時に、更新後のファイルのイメージををログファイルに書き出すもので、簡単に言えばログ書き出しが完了していればログにある内容が、ログ書き出し中に障害が発生した場合はもとからあった内容が、それぞれ最新の内容として残るというものである。第 0.5 版では、障害が発生してもデータファイルが失われることはなく、そのデータ内容は少なくともシステム立ち上げ時の内容まで復旧可能としていた。

ディレクトリを導入すると、データファイルの複製や移動のように、複数のディレクトリやデータファイルが一つの更新要求の対象となる場合がある。例えばデータファイルの移動の場合、

1. データファイルのリンク数を増やす
2. 移動先のディレクトリに対象データファイルを追加
3. 移動元のディレクトリから対象データファイルを抹消
4. データファイルのリンク数を減らす

といった一連の操作が行われる。このような場合、対象となる全ファイルに対する一連の操作が完了した時点で、初めて一つの更新要求が完了となる。この一連の操作はあくまでも原子的な更新であり、最終的に全ての操作が正しく完了した状態か、又は全ての操作が行われなかった状態にする必要がある。一方、ログは個々のファイルの更新操作に対応して採取されるため、個々の更新操作に対するログを一連の原子的なログとしてどう管理するかが問題になるが、この点は既にログ管理機能設計時に考慮済みであった [3]。しかし、複数のディレクトリやデータファイルに対する一連の原子的なログをどう採取するか、第 0.5 版で実現済みのデータファイルのみの更新ログ採取方式とどう整合性を取っていくかが問題となる。

3 基本設計

3.1 ディレクトリキャッシュ

第 0.5 版では、既にデータファイルの内容を分散キャッシュ管理している [4]。これはサーバ・クライアント形式で、ディスク装置が置かれているノードにサーバを置き、サーバから各ノード上のクライアントへのデータの提供や、各クライアントのキャッシュ無効化の指示を行うものである。ディレクトリキャッシュに関しても同様に、サーバ・クライアント形式を取ることとした。また、データキャッシュ管理では更新されたキャッシュ内容のディスク装置への反映に write - back 方式を採用していたが、これはシステム障害発生時にデータファイルの内容はある程度失われても構わず、それよりも通常のデータ更新処理を高速化しようと考えたためであった。しかしディレクトリに関しては、システム障害が発生してもその内容が失われてはいけないため、データキャッシュのような管理方式は使えない。そこで、ディレクトリに関しては write - through 方式のキャッシュ管理方式を採用した。以下にディレクトリキャッシュの基本設計を示す。

キャッシュする情報: キャッシュする情報は、大別して二種類である。まず、ユーザによって指定されたディレクトリやデータファイルの名前を、ファイルシステムの内部表現に変換するための両者の対応情報である。次に、パス名を辿りながらユーザのアクセス権を調べる時に使用する、各ディレクトリやデータファイルの属性情報 (保護情報) である。この二種類の情報があれば、パス名の検索は全てローカルノード上で可能となる。

キャッシングの単位: 名前とシステム内部表現の対応情報を持つ場合、パス名の一段毎に、即ち個々のディレクトリやデータファイル毎に、内部表現との対応情報を持つ方法と、全パス名に対して最終的に得られたディレクトリやデータファイルの内部表現を対応させて対応情報とする方法の二通りが考えられる。我々はキャッシュを無効化する場合の手間を考慮して、前者の方法を採用した。前者は無効化要求された一つのファイル名のキャッシュだけを即座に無効化してしまえばよいが、後者はキャッシュされている全パス名について、無効化されたファイル名が含まれるかどうかを調べなければならないからである。なお、属性情報のキャッシュに関しても同様にファイル単位とした。

write - through 方式: 上述したように、ディレクトリとその内容はシステム障害が発生しても失われてはいけない。このため、ディレクトリのキャッシュは write - through 方式とした。即ち、ディレクトリやその内容の更新を伴う処理は即座にサーバに伝えられ、キャッシュ無効化処理 (後述) の後、最新の情報がディスク装置 (ログ) に書き込まれる。

キャッシュの無効化: ある情報が複数のノード上にキャッシュされている場合、そのうちのどこか一つのノード上でキャッシュ内容が更新されると他のノード上にキャッシュされている同じキャッシュ内容を無効化、又は同じように更新してやる必要がある。我々はデータキャッシュと同様に、サーバが主体となって無効化を行う方式を採用した。即ち、ディレクトリに関する更新の情報は write - through であるために即座にサーバに伝えられ、サーバが同じ情報をキャッシュしている他のクライアントに当該キャッシュの無効化を伝える。サーバは無効化要求を送った全クライアントから無効化完了の通知を受け取るまで待ち、その後最新情報をディスク装置に反映する。

3.2 メンバの追加、削除と領域割り当て、解放アルゴリズム

ディレクトリへのメンバの追加、削除の処理は、ディレクトリの更新操作であるため、すべてサーバ側に送られてサーバ側で処理される。まずディレクトリへのメンバの追加と領域の割り当ては、以下のアルゴリズムで行う。

1. ディレクトリの大きさ、ディレクトリ全体での空き領域の大きさ、及びディレクトリ使用領域内の各ブロック毎の空き領域の大きさを覚えておく。
2. 一番最近メンバ(レコード)を追加したブロックを出発点として覚えておく。初期値は先頭ブロックである。
3. 出発点のブロックに追加できればそこに追加する。
4. 出発点のブロックに追加できない場合、ディレクトリの大きさと空き領域の大きさを比較する。
   - 空き領域が全体の 30% 未満ならば、新たなブロックを割り当ててそこに追加する。
   - 空き領域が全体の 30% 以上ならば、出発点から順にサーチして空き領域を探す。サーチは出発点−>最終ブロック−>先頭ブロック−>出発点という順で行い、追加できるブロックが見つかったらそこに追加する。もし一周しても追加可能なブロックが見つからなかった場合は、新たなブロックを割り当ててそこに追加する。
5. メンバを追加したブロックを次の出発点として覚えておく。また、ディレクトリの大きさや空き領域の大きさを更新しておく。

新たなブロック割り当てを行うための、ディレクトリの大きさと空き領域の大きさの比率を 30% としたのは、あくまでも暫定的なものである。実際にはシステムを稼働させて最適値を探る必要がある。

一方、ディレクトリからのメンバの削除と領域の解放については、あるディレクトリレコードが削除され、レコードが削除されたブロックが空になった場合に、最終ブロックの内容を空いたブロックにコピーし、最終ブロックを解放することとした。この方式では領域の無駄が少なくなるが、比較的領域解放の回数が多くなる。しかし、メンバの追加時に隙間から埋めていくような方式を採っているため、通常一つのブロックには複数のディレクトリレコードが含まれていると期待でき、メンバの削除毎に領域解放が起こる、即ち各ブロックに一ディレクトリレコードしか存在しないというようなことは稀であろうと考えられる。また、最悪の場合、このような状態になることもあるが、そのような場合の空き領域の詰め合わせ(コンパクション)は別途ユーティリティを用意して対処することとした。

3.3 原子的なログの採取方式

第0.5版のデータファイルの更新ログでは、データファイルのファイルレコードの更新後の内容をログに書き出していた。更新内容はデータファイルの管理を行うファイル管理部でメッセージに書き込み、それをログの管理を行うログ管理部に送り、ログへの書き出しを依頼していた。ディレクトリを導入した場合も同様な枠組とし、ディレクトリの操作に関するログは、ディレクトリの管理を行うディレクトリ管理部で作成してログ管理部に送ればよい。しかし、ディレクトリの操作ではディレクトリのファイルレコードとそのデータ(更新のあったメンバについての情報が格納されている部分)の二つがログの

対象となるため、ログは単独ではなく、必ず複数の原子的なログとなる。そこで、ログ管理部のインタフェースに追加修正を加え、原子的な複数の更新ログの内容を、一括してログ管理部に伝えるメッセージを用意した。このメッセージは具体的には、ファイルレコードのログ書き出し内容をまとめるタームと、ディレクトリデータ部のログ書き出し内容をまとめるタームとを、複数個リストで括ったものを引数として持っている。

また、前述のようにディレクトリとデータファイル両方のログを原子的に採取して書き出さなければならない場合があり、そのようなログはディレクトリ管理部、もしくはファイル管理部単体では採取できない。そこでこのような場合には、まずディレクトリに関するログをディレクトリ管理部で作成してそれをファイル管理部に送り、ファイル管理部でデータファイルの分のログを作成して、ディレクトリのログと合わせてログ管理部に送る方式を採用した。NFS 等の分散ファイルシステムで見られるような、ディレクトリとデータファイルを同じ枠組で管理する方法も考えられ[5]、その場合はディレクトリ管理部とファイル管理部を統合することになるが、それにはファイル管理部の大規模な修正が必要となるため、第0.5版で既に完成済みの部分との整合性を考えて本方式を採用した。この方式は、上述のログ管理部に追加した原子的なログの書き出し要求メッセージを利用し、さらにファイル管理部のインタフェースに多少手を加えるだけで実装可能である。具体的な処理の流れとしては、まずディレクトリ管理部でディレクトリに関するログ書き出し内容を上述のタームのリストにまとめ、メッセージにのせてファイル管理部に送る。次にファイル管理部でファイルに関するログ書き出し内容をタームにまとめ、それをディレクトリ管理部から渡されたタームのリストに追加して、ログ管理部に送るようにした。

4 実現方式詳細

4.1 論理ボリュームリンク

ファイルシステムは複数の論理ボリュームによって構成されている。そのうち一つの論理ボリュームのルートディレクトリを、システム全体のルートディレクトリとする。残りの論理ボリュームのルートディレクトリは、他の論理ボリュームのどこかのディレクトリに接続され、全体として一つの木構造を構成する。この接続を論理ボリュームリンクと呼び、論理ボリュームが接続されたディレクトリをマウントポイントと呼ぶ。システム全体のルートディレクトリと、論理ボリュームリンクの張り方は、ファイルシステム立ち上げ時に特定される。論理ボリュームリンクは、ファイルシステム稼働中にも動的に張り直すことができるが、マウントポイントを含む論理ボリュームはディスマウントできない。また、複数の論理ボリュームを同じマウントポイントにマウントすることは可能だが、一番最後にマウントされた論理ボリュームだけが見えるようになる。

4.2 ディレクトリの形式

各ディレクトリは、基本的にはデータファイルと同じ形でディスク装置に格納される。即ち、ファイル管理ファイルにディレクトリ自身に関する情報(種別、更新日時、保護情報、大きさ、データ格納場所へのポインタ等)を入れたファイルレコードが格納され、そのディレクトリのメンバに関する情報(メンバのファイル番号、ファイル名長、ファイル名)を入れたディレクトリレコードが、そのディレクトリのデータ領域に格納される。ディレクトリのファイルレコードの形式を表 1 に示す。

4.2.1 ディレクトリレコードの形式

ディレクトリデータ部はディレクトリファイルとも呼び、ディレクトリの各メンバの情報がディレクトリレコードとして格納されている。各ディレクトリレコードの形式を以下に示す。

- ディレクトリレコードには、ファイル名長(4 バイト)、対応するファイルのファイル番号(4 バイト)、およびファイル名を格納する。
- 自分自身と親ディレクトリのディレクトリレコードには、それぞれ「.」と「..」を設定する。使用者がこれらの名前のファイルを定義することは不可とする。

位置 (バイト)	長さ (バイト)	内容
0	4	ヘッダ (ログ管理用)
4	4	ファイル番号
8	2	ファイル種別と参照許可
10	2	結合数
12	2	所有者識別子
14	2	グループ識別子
16	8	最終更新時刻
24	16	未使用
40	8	ファイル長
48	4	ブロック数
52	64	ブロックマップ (0~15)
116	12	間接ブロックマップ (0~2)

表 1: ディレクトリのファイルレコードの形式

フィールド	長さ (バイト)	(値:) 内容
ヘッダ	4	ログ管理用
ファイル番号	4	ディレクトリのファイル番号
物理ブロック順番号	4	物理ブロックのディレクトリ内での順番号
空き領域長	4	*FAL*: $PBS - \sum DRL - 16$
ディレクトリレコードの並び	$\sum DRL$	1 個以上のディレクトリレコード
空き領域	*FAL*	無効データ

PBS: 物理ブロック長 (バイト数)
DRL: ディレクトリレコード長 (バイト数)

表 2: ディレクトリデータ部の物理ブロックの形式

- ファイル名長の最大は、(物理ブロックのバイト数 - 28) バイトに制限する。

ディレクトリのデータ部の物理ブロックの形式を表 2に、ディレクトリレコードの形式を表 3に示す。

4.3 ディレクトリのファイルレコードとデータの配置

今までまだ触れていない話題として、あるディレクトリのファイルレコードと、そのディレクトリのデータ部をディスク装置内のどこに配置すれば、ディスク領域の使用効率やアクセス速度の点で最適かといった問題がある。

PIMOS ファイルシステムの論理ボリューム管理では、各論理ボリュームを複数のシリンダグループに分割して管理しており、同一のシリンダグループ内にあるデータは高速に読み書きできることが期待できる。また、ディスク領域の使用効率を上げるため、領域割り当て単位を小ブロックと大ブロックの二通り用意している。データファイルの場合には、データが少ない間は小ブロックを割り当て、小ブロックでは足りなくなった場合には大ブロックを割り当てて小ブロックにあったデータをコピーし、以後領域が必要になったら大ブロックを割り当てている。この時、ファイルレコードとデータ領域をどのシリンダグループに割り当てるかに関しては、論理ボリューム全体から領域を使用するように、空きのある

フィールド	長さ (バイト)	(値:) 内容
ファイル名長	4	*FNL*: ファイル名の長さ (バイト数)
ファイル番号	4	対応するファイルのファイル番号
ファイル名	*FNFL*	先頭より *FNL* バイトがファイル名

FNFL: *FNL* 以上の最小の 4 の倍数

表 3: ディレクトリレコードの形式

シリンダグループから適当に割り当てている程度である。

ディレクトリについても同様に割り当てを行うこととし、始めは小ブロックを割り当て、領域が不足したら大ブロックにコピーし、以後大ブロックを割り当てることにした。ただし、大きさが小ブロックで足りている間は高速に読み取るためにファイルレコードと同じシリンダグループから割り当て、領域が不足して初めて大ブロックが割り当てられる時点で、大ブロックを他の空き領域の多いシリンダグループから割り当てるというアルゴリズムを採用した。この割り当て方式により、ディレクトリが小さい、即ちメンバ数が少ない間は高速に読み取ることができる。また、メンバ数が増えてディレクトリが大きくなった場合には、空き領域の多いシリンダグループから領域を使用するため、論理ボリューム全体としての領域使用効率が高くなる。この場合読み取りに要する時間は増えてしまうが、もともとデータ量が多くて読み取りには時間がかかってしまうので、読み取りを高速にすることよりも領域使用効率を重視した。

また、さらにディレクトリの各メンバについては、親ディレクトリに領域を割り当てたのと同じシリンダグループ上にファイルレコードを置くことにした。これにより、ディレクトリメンバの一覧表示等の際に、メンバの情報が高速に得られ、処理の高速化が期待できる。ディレクトリのファイルレコードとデータの配置の様子を、図 1に示す。

4.4 プロセス構成

ディレクトリ管理、ファイル管理、ログ管理まわりのプロセス構成を図 2に示す。
以下にディレクトリ管理関係のプロセスの概要を示す。

ディレクトリキャッシュ: 各ノード上に常駐し、ユーザからのディレクトリ操作要求を受ける。ディレクトリ検索高速化のためのキャッシングを行う。

ディレクトリ管理: サーバノード上に常駐し、ディレクトリキャッシュからのメッセージを対応するディレクトリプロセスに振り分ける。

ディレクトリ: 操作対象の各ディレクトリに対応して生成され、各ディレクトリの属性やデータを保持する。そして、各ディレクトリに関するキャッシングの支援、ディレクトリ更新内容の作成とログへの書き出し等の処理を行う。

ディレクトリマップ: ディレクトリプロセスに対応して生成され、ディレクトリで使用するブロックの割り当て、解放を行う。

ユーザからのディレクトリに対する操作は、ユーザインタフェース経由でディレクトリキャッシュに伝えられ、さらにディレクトリ管理に伝えられる。

4.5 ディレクトリ管理

4.5.1 クライアント側

ディレクトリキャッシュでは、ユーザから渡されたディレクトリ名やデータファイル名のアクセス権チェックを伴った検索を行い、それを論理ボリューム名とファイル番号の組に変換する。また、次回以降の検索の高速化のために、検索の際に得られた情報のキャッシングを行う。

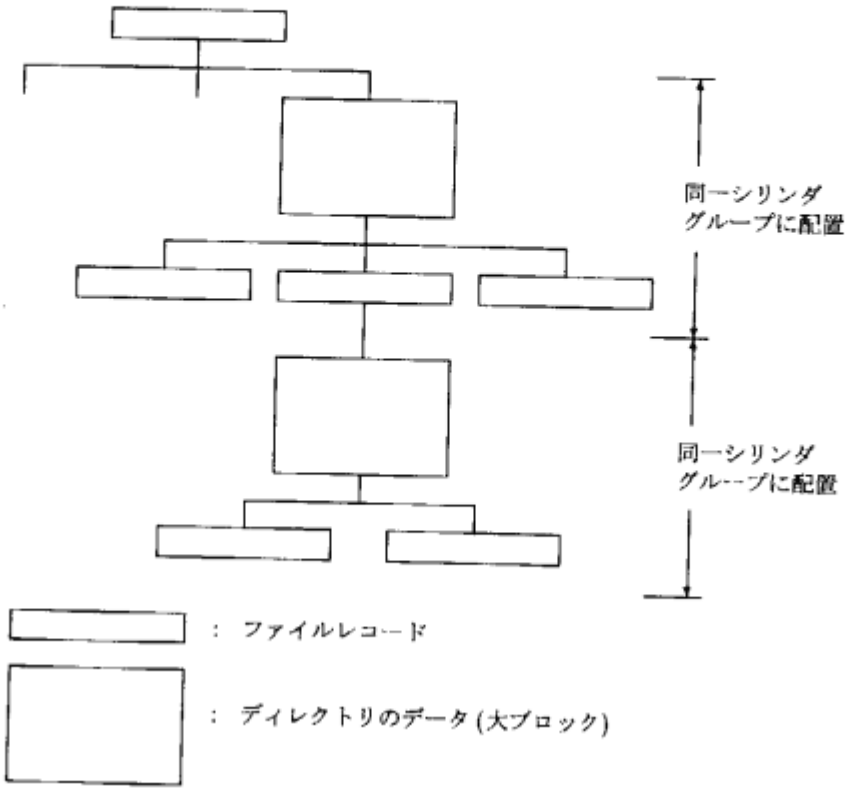

図 1: ディレクトリのファイルレコードとデータの配置

以下にディレクトリキャッシュプロセスがユーザから受けとる要求の種類を示す。

- ファイル生成要求 (create)
- ディレクトリ生成要求 (mkdir)
- ファイルオープン要求 (open)
- ファイル一覧要求 (list)
- ファイル登録要求 (link)
- ファイル抹消要求 (unlink)
- ディレクトリ抹消要求 (rmdir)
- メンバ移動要求 (move)
- ファイル情報取り出し要求 (attributes：種別、保護情報、更新時刻、リンク数、ファイル長)
- ファイル所有者識別子変更要求 (change_owner)
- ファイルグループ識別子変更要求 (change_group)
- ファイル保護情報変更要求 (change_permission)

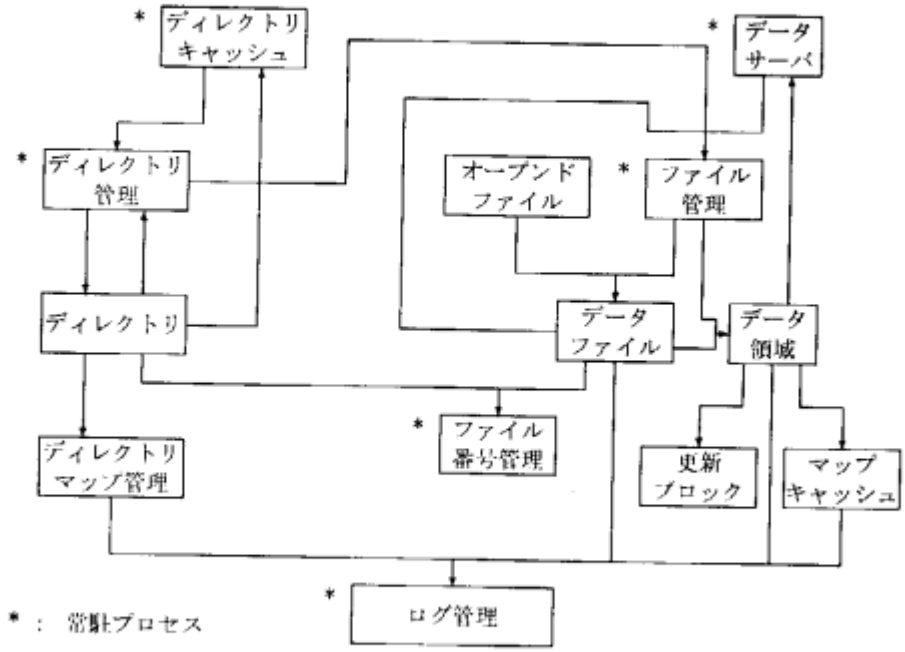

図 2: ディレクトリ管理／ファイル管理プロセス構成

- ファイル日付情報変更要求 (change_time)

前述のように、キャッシングする情報は名前を内部表現に変換するための情報と、アクセス権をチェックする際に使用する各ディレクトリやデータファイルの属性情報である。そこで、名前と内部表現の対応表と、内部表現からその属性情報を得るための表の、二つの表を持たせた。この二つの表の前者を名前キャッシュ、後者を情報キャッシュと呼び、それぞれLRUで総数管理する。情報キャッシュで名前からでなく内部表現から属性情報を引くようにしたのは、名前は一つのファイルに対して複数定義できるため、複数の名前が一つの属性情報を指してしまうのに対し、内部表現はファイルに対して一つのため属性情報とも一意に対応が取れ、属性情報のキャッシュの無効化が容易になるためである。また、情報キャッシュはその要素の種別によってキャッシュする情報が異なり、要素がデータファイル又はディレクトリの場合はファイル種別と保護情報を、要素がマウントポイント(論理ボリュームリンクが接続されたディレクトリ)の場合はファイル種別と接続先論理ボリューム名をキャッシュする。

また、キャッシュの管理はwrite - throughで行うため、ディレクトリやそのメンバの更新を伴う処理はすべてサーバに送られる。このため、サーバに送った要求の完了を待つための待ち処理表を用意する。待ち処理表はディレクトリ単位で管理し、同じディレクトリに対する要求は、待ち処理表内の同じ完了待ちリストに順番に加えられる。この時、一つの処理が完了するのを待たずに次の処理をサーバに送り、ディレクトリのメンバ一覧を見る場合等に有利にしてある。これは、同じディレクトリに対する処理をサーバ側で一元化しているために可能となっている。

4.5.2 サーバ側

サーバ側では各ノード上でのキャッシングの支援、ディレクトリ更新操作の処理、ディレクトリ検索及び更新の同時実行制御等を行う。サーバ側の窓口はディレクトリ管理プロセスだが、各ディレクトリに関する実質的な管理は、対応するディレクトリプロセスが行う。

ディレクトリ管理プロセスは受けとったメッセージを対応するディレクトリプロセスに配分したり、

ディレクトリプロセスの総数管理をして、必要に応じてその生成、消滅を行う。この時、ディレクトリプロセスの総数管理はLRUではなく、あらかじめプロセス個数の上限値を決め、プロセス個数が上限値に達した時に各ディレクトリプロセスに消滅要求を送る方式としている。これは、ディレクトリプロセスは生成された順番やアクセスの古さ、少なさよりも、クライアント側にキャッシュされなくなったものから消滅させたいためである。従って、消滅要求を受けとったディレクトリプロセスは、自身の管理するディレクトリの情報をキャッシュしているクライアントがいなければ消滅し、いれば消滅を拒否する。ディレクトリ管理プロセスでは消滅させたディレクトリプロセスの個数を数え、十分な個数だけ消滅しなかった場合は上限値を倍にする。ここで十分な個数というのがどのくらいかが問題になり、とりあえず全体の三割程度と考えているが、この値は実際にシステムを稼働させて評価してみる必要がある。

また、論理ボリュームリンクに対応するために、自身の管理する論理ボリューム内にあるマウント点の個数と、自身が論理ボリュームリンクしている接続元のディレクトリの論理ボリューム名とファイル番号を保持する。前者によって自分をディスマウントする際に下位に論理ボリュームリンクがあるかどうかのチェックを行い、後者によって論理ボリュームリンクを含むパスを逆にたどることが可能になる。

ディレクトリプロセスとディレクトリマッププロセスはファイル単位で生成され、一対一に対応している。ディレクトリマッププロセスは、ディレクトリプロセスでブロックが要、不要になった場合に、それに応じて領域の割り当て、解放を行う。

各ノード上でのキャッシングの支援、ディレクトリ更新操作の処理はディレクトリプロセスで行われる。ディレクトリプロセスは自身の管理対象のファイル種別を保持しており、それによって保持する情報が異なる。以下にディレクトリプロセスの保持情報をまとめる。

種別にかかわらない共通情報:

- 自論理ボリューム名
- 自ファイル番号

ディレクトリの場合の保持情報:

- メンバ表: メンバ名をキーとし、そのメンバのファイル番号、ディレクトリレコード所在ブロック番号、メンバ名をキャッシュしているノードのリスト、を値とする。
- ブロック表: ブロック番号をキーとし、そのブロックにディレクトリレコードが含まれるメンバのメンバ名とファイル番号、そのブロックの残り空き領域の大きさ、を値とする。
- このディレクトリの属性情報: ファイル種別、保護情報 (アクセス権、所有者識別子、グループ識別子)、リンク数、最終更新時刻、ファイル長
- 空き領域の大きさ
- このディレクトリの属性情報をキャッシュしているノードのリスト
- 最終ブロック番号とメンバ追加ブロック番 (次にメンバを追加しに行くブロックの番号)

マウント点の場合:

- 接続先論理ボリューム名
- このディレクトリがマウント点であることをキャッシュしているノードのリスト

データファイルの場合:

- このデータファイルの属性情報をキャッシュしているノードのリスト

上の情報一覧から明らかなように、種別がディレクトリの場合、ディレクトリの内容を全部読み出してメモリ上に表として保持している。つまり、各ディレクトリの内容をサーバ側でキャッシュする形になっており、これによってディスク装置へのI/Oを減少させ、処理の高速化を計っている。また、クライアント側のキャッシュと同様にwrite - throughで管理しており、ディレクトリの更新があった場合は即座にログに書き出し、ログ書き出し完了後にキャッシュ内容を更新する。また、更新処理時にはクライアント側のキャッシュの一貫性制御も行う必要がある。即ち、更新要求のあった情報(名前情報または属性情報)に関し、その情報をキャッシュしている全ディレクトリキャッシュに対して当該キャッシュの無効化を指示し、無効化完了を待って更新処理を行う。

種別がマウント点の場合、ディレクトリプロセスは論理ボリュームリンクが張られた時に生成され、以後消滅要求が来ても消滅しない。論理ボリュームがディスマウントされると通常のディレクトリに対するディレクトリプロセスと同じ状態になる。

4.6 ファイル管理

ディレクトリとデータファイルの原子的な操作は、データファイルの生成、登録(リンク)、抹消(アンリンク)、の三種である。以下にそれぞれの場合のログ書き出し内容を示す。

データファイルの生成: 親ディレクトリのファイルレコードの最終更新時刻とディレクトリの大きさを更新、ディレクトリファイルにメンバ追加、データファイルのファイルレコードを新規作成

データファイルの登録: 親ディレクトリのファイルレコードの最終更新時刻とディレクトリの大きさを更新、ディレクトリファイルにメンバ追加、データファイルのファイルレコードのリンク数を更新

データファイルの抹消: 親ディレクトリのファイルレコードの最終更新時刻とディレクトリの大きさを更新、ディレクトリファイルにメンバ追加、データファイルのファイルレコードのリンク数を更新

データファイルの生成と登録に関しては、ディレクトリ管理部から渡されたログ書き出し内容リストにデータファイル分のログを追加し、ログ管理部へと送る。データファイルの抹消に関しては、他の二種と同様の処理に加え、リンク数が0になったファイルを削除する。この時、ファイルがオープンされていなければ即座に、オープンされていた場合は全てのオープンがクローズされた時に削除処理を行う。

これらの他、データファイルの生成時に必要になるファイル番号は、従来はファイル管理部が自分でファイル番号管理部からファイル番号を得ていたが、ディレクトリ管理部がファイル番号管理部から得て、それをファイル管理部のファイル生成述語の引数として渡すように変更した。

4.7 ログ管理

前述のように、原子的な複数の更新ログ内容を一括して受けとるためのメッセージを追加した。メッセージは update(Term_list, ^Status) という形式で、リストで渡されるタームの種類は以下の三種である。

write_manager_file: ファイルレコードを書き出すためのタームで、ファイル番号と、書き出す内容を入れたストリングを持つ。

write_directory: ディレクトリファイルの内容を書き出すためのタームで、ファイル番号やブロックの位置指定等と、書き出す内容を入れたストリングを持つ。

invalidate_directory: ディレクトリファイル中の空になったブロック(ディレクトリブロック)の無効化のログを書き出すためのタームで、ファイル番号とブロックの位置指定を持つ。

また、ディレクトリではデータファイルの時にはログ対象ではなかったデータ部についてもログを取るため、無効なディレクトリブロックのログや、ディレクトリブロックの無効な実体(本来の格納場所)が生じる。そこで、障害復旧処理においてこれらを判定する機能を追加した。

無効なディレクトリブロックのログ: ディレクトリを削除する場合、ディレクトリレコードのログのみ書き出して、そのディレクトリの内容のログは書き出さない。これは、もはや意味のないディレクトリブロックのログを書き出す手間を省くためだが、今までに書き出されたそれらのディレクトリブロックのログはそのまま残っている。これらのログが誤って実体に反映されると、ファイルの内容やディレクトリの情報を破壊してしまう恐れがある。そこで、ディレクトリブロックを復旧する際にはディレクトリの大きさを参照し、それらのディレクトリブロックが本当にディレクトリに含まれているかを調べ、ログの有効性を確認しながら復旧処理を行う。

ディレクトリブロックの無効な実体: ディレクトリブロックの更新内容はログか実体にあり、どちらに最新の内容があるかは両者を比較して判定する。しかし、実体が割り当てられたばかりの初期状態だと、更新内容の新旧を示す情報が設定されていないため、実体を比較の対象としてはいけない(この場合はログに最新の内容がある)。そこで、各ディレクトリブロックの実体の有効性の情報(初期状態か否か)を、ディレクトリのファイルレコードに保持しておき、それを参照しながら復旧処理を行う。

実際にはこの情報は、ディレクトリのファイルレコード内のブロックマップの、各ブロックのマップの先頭ビットを利用する。ここで、実際にディレクトリのデータブロックに割り当てられるのは、論理ボリュームで管理する小ブロックまたは大ブロックで、実体の有効性の情報も小ブロックまたは大ブロック単位で保持していることになる。ログの大きさは物理ブロック単位であるので、実体が初期状態のディレクトリブロックのログを実体反映する場合、そのディレクトリブロックと同じ小ブロックまたは大ブロック内の、全ての物理ブロックの更新内容がそろった状態にしなければ、そのブロックの情報を更新できない。そこで実体が初期状態の場合は、ブロック単位で一括して実体反映を行うこととした。

5 あとがき

PIMOS ファイルシステムのディレクトリ管理方式について述べた。分散キャッシュを導入して情報の局所性を上げ、高速なパス名検索を実現した。また、処理速度とディスク領域使用効率のバランスに配慮したディレクトリへの領域割り当て、解放アルゴリズムを考案した。同時に、ディレクトリやそのメンバのファイルレコードと、ディレクトリレコードの配置についても、処理速度や論理ボリューム全体のディスク使用効率を考慮して、適切なシリンダグループに配置するようにした。さらに、原子的なログの採取方式を検討し、既存の版との整合性の良い実現方式を採用した。同時にディレクトリの更新ログの復旧処理を検討し、システム障害に備えた。課題としては各アルゴリズム中に含まれる数値の最適化が挙げられる。また、ディレクトリとデータファイルを同じ枠組で管理する方式について検討してみるのも、興味深い課題であろう。

参考文献

[1] Itoh,F., Chikayama,T., Mori,T., Sato,M., Kato,T. and Sato,T., "The Design of the PIMOS File System", In Proceedings of the International Conference on Fifth Generation Computer Systems, Vol.1, ICOT, Tokyo, 1992, pp.278-285.

[2] 伊藤他:"PIMOS ファイルシステムの設計 (1) 設計方針", 情報処理学会第 43 回全国大会, 3L-5, 1991

[3] 加藤他:"PIMOS ファイルシステムの設計 (4) ログ管理", 情報処理学会第 43 回全国大会, 3L-8, 1991

[4] 森他:"PIMOS ファイルシステムの設計 (2) データキャッシュ管理", 情報処理学会第 43 回全国大会, 3L-6, 1991

[5] Levy,E. and Silberschatz,A., "Distributed File Systems: Concepts and Examples", TR-89-04, Depertment of Computer Sciences, The University of Texas at Austin, Austin, 1989