Veritas™ File System プログラマーズリファレンスガイド

Solaris

5.0

N18948F

symantec™

Veritas File System
プログラマーズリファレンスガイド

Veritas File System 5.0

サードパーティ（第三者）製ソフトウェアの権利に関する通知

本製品には、特定のサードパーティ製ソフトウェアが配布、組み込み、または同梱されている場合があります。また、本製品のインストールおよび使用にともない、サードパーティ製ソフトウェアの使用を推奨する場合があります。同サードパーティ製ソフトウェアのライセンスは、著作権の保有者により別途付与されます。サードパーティのソフトウェアの使用に必要なライセンスおよび著作権に関する情報については、本製品リリースノートのサードパーティに関する章を参照してください。

Solaris は Sun Microsystems, Inc. の商標です。

テクニカルサポート

製品のサポートを受けるには、http://support.veritas.com ページへアクセスし「Phone Support」または「E-mail Support」をクリックします。このページから TechNote、Software Alerts、ソフトウェアのダウンロード、ハードウェア互換性リスト、VERITAS Email Notifications サービス などにアクセスすることもできます。「Knowledge Base Search」機能を使用し、製品ドキュメントのリリースなどの製品情報へアクセスすることができます。

目次

第 3 章 MVS（Multi-volume support）

第 4 章 名前付きデータストリーム

第 5 章 VxFS I/O アプリケーションインターフェース

1

Veritas File System Software Developer's Kit

この章では、次の内容について説明します。

Software Developer's Kit について

Veritas File System（VxFS）Software Developer's Kit（SDK）は、アプリケーションプログラミングインターフェース（API）を使って Veritas File System の各種の機能とコンポーネントを修正して調整するために必要な情報を、開発者に提供します。これらの API は、VxFS Software Developer's Kit（SDK）に付属しています。

このマニュアルで説明しているほとんどの API は、VxFS 4.1 およびそれ以降のリリースで使えます。

73 ページの第 5 章「VxFS I/O アプリケーションインターフェース」で取り上げる API は、VxFS 4.0 以降のリリースで使え、それより前の一部のリリースでも使えます。

File System Software Developer's Kit の機能

このセクションでは、SDK によってアクセス可能な VxFS 機能の概要について説明します。

API ライブラリインターフェース

この SDK で特に重要な API ライブラリインターフェースは、`vxfsutil` ライブラリと VxFS IOCTL 指示語です。このライブラリには、アプリケーションから VxFS ファイルシステムの機能を利用するために使う API 呼び出しの集合が含まれています。すべての API インターフェースのマニュアルページが用意されています。このライブラリには、次の機能の API が含まれています。

API	機能
inotopath	i ノードとパスの検索
nattr	名前付きデータストリーム
FCL	FCL（File Change Log）
MVS	MVS（Multi-volume support）
キャッシュアドバイザリ	IOCTL 指示語
エクステント	IOCTL 指示語
フリーズ / アンフリーズ	IOCTL 指示語

VxFS API ライブラリ `vxfsutil` は、Veritas File System 製品とは別に、単独でインストールできます。このライブラリは、スタブライブラリとダイナミックラ

イブラリを組み合わせて実装されています。アプリケーションをスタブライブラリ libvxfsutil.a でコンパイルすると、任意の VxFS 環境に移動できます。さらに、そのアプリケーションを VxFS ターゲット上で実行できます。スタブライブラリは VxFS ターゲットにあるダイナミックライブラリを検出します。

スタブライブラリは vxfsutil.so ダイナミックライブラリのデフォルトのパスを使います。多くの場合、デフォルトのパスを使う必要があります。ただし、デフォルトのパスは、環境変数 *LIBVXFSUTIL_DLL_PATH* に vxfsutil.so ライブラリのパスを設定することによって上書きできます。この構造によって、VxFS の他のリリースとの互換性に関する問題を最小限に抑えながら、アプリケーションを配備できます。

FCL（File Change Log）

VxFS FCL（File Change Log）は、ファイルシステム内のファイルおよびディレクトリへの変更を記録します。FCL は、バックアップ製品、Web 巡回ロボット、検索および索引エンジン、レプリケーションソフトウェアなど、ファイルシステム全体をスキャンして、前回のスキャン以降に変更された箇所を検出するアプリケーションによって使われます。

15 ページの「FCL（File Change Log）」を参照してください。

MVS（Multi-volume support）

MVS 機能を使って、VxFS ファイルシステムで複数の Veritas Volume Manager（VxVM）ボリュームを下位ストレージとして使えます。管理者やアプリケーションがファイルの格納場所を制御することができるので、低コストで効果的に処理効率を向上させることができます。この機能は、Veritas Volume Manager とともに使う場合に限り、使えます。また、一部の機能には追加のライセンスキーが必要です。

51 ページの「MVS（Multi-volume support）」を参照してください。

VxFS I/O

VxFS は SVID（System V Interface Definition）必要条件に準拠し、NFS（Network File System）を使ったユーザーアクセスをサポートしています。他のファイルシステムでは利用できない処理効率機能を必要とするアプリケーションは、VxFS の利点を活用することができます。

Software Developer's Kit パッケージ

この SDK は 2 つのパッケージ、VRTSfssdk と VRTSfsmnd で構成されています。VRTSfssdk パッケージには、ライブラリ、ヘッダーファイル、アプリケーションを開発してコンパイルするための VxFS API インターフェースの使用方法を示したソースおよびバイナリ形式でのサンプルプログラムが含まれています。VRTSfsmnd パッケージには、このマニュアルおよび API マニュアルページが含まれています。

VRTSfssdk パッケージのディレクトリ構造は次のとおりです。

src	いくつかのサブディレクトリに分けて、関心の高い各トピックのサンプルプログラムと GNU ベースの Makefile ファイルがあります。
bin	ソースディレクトリ内のすべてのサンプルプログラムへのシンボリックリンクがあり、バイナリに簡単にアクセスできます。
include	API ライブラリと ioctl インターフェースのヘッダーファイルがあります。
lib	あらかじめコンパイル済みの vxfsutil API インターフェーススタブライブラリがあります。
libsrc	vxfsutil API インターフェーススタブライブラリのソースコードがあります。

VRTSfssdk パッケージと VRTSfsmnd パッケージは、VxFS パッケージとは別に入手できます。アプリケーションまたはサンプルプログラムを実行するには、ライセンスを取得済みの VxFS ターゲットが必要です。また、インストール先のシステムに、必要な機能の VxFS ライセンスをインストールしてください。

必要なライブラリとヘッダーファイル

VRTSfssdk パッケージは /opt ディレクトリにインストールされます。関連ライブラリとヘッダーファイルは次の場所にインストールされます。

- /opt/VRTSfssdk/5.0/lib/libvxfsutil.a
- /opt/VRTSfssdk/4.1/lib/sparcv9/libvxfsutil.a
- /opt/VRTSfssdk/5.0/include/vxfsutil.h
- /opt/VRTSfssdk/5.0/include/sys/fs/fcl.h
- /opt/VRTSfssdk/5.0/include/sys/fs/vx_ioctl.h

標準 Veritas パス（/opt/VRTS/lib および /opt/VRTS/include）からこれらのファイルへのシンボリックリンクもあります。標準パスは、VxFS および VxFS SDK の最新リリースのデフォルトのパスです。

コンパイル環境

サンプルプログラムはコンパイル済みのバイナリとともに SDK パッケージによってインストールされます。サンプルプログラムの実行の必要条件は次のとおりです。

- 適切なバージョンの VRTSvxfs がインストールされている対象システム
- 一部のプログラムに必要な root 権限
- マウントされた vxfs 4.x/5.0 ファイルシステム。一部には、ファイルシステムが Veritas ボリュームセットにマウントされている必要のあるプログラムもあります。

メモ : 一部のプログラムでは、特別なボリューム設定（ボリュームセット）が必要なことがあります。また、一部のプログラムでは、ファイルシステムがボリュームセットにマウントされている必要があります。

異なるコンパイラによる再コンパイル

src ディレクトリまたは libsrc ディレクトリの再コンパイルに必要なツールは次のとおりです。

- gmake コマンドまたは make コマンド
- gcc コンパイラまたは cc コマンド

src ディレクトリや libsrc ディレクトリを再コンパイルするには

1. make.env ファイルを編集し、コンパイラへのパスで変更します。
2. src ディレクトリまたは libsrc ディレクトリに変更し、gmake コマンドまたは make コマンドを実行します。

   ```
   # gmake
   ```

3. アプリケーションの作成後、次のようにコンパイルします。

   ```
   # gcc -I /opt/VRTS/include -L /opt/VRTS/lib -ldl -o MyApp \
   MyApp.c libvxfsutil.a
   ```

src ディレクトリまたは libsrc ディレクトリをコンパイルするには、/opt/VRTSfssdk/5.0/make.env ファイルを次のように編集します。

1. ローカルシステムに置かれたコンパイラのパスを選択し、システム上のパスを指す CC を選択して編集します。

   ```
   CC=/opt/SUNWspro/bin/cc（任意の適切なパス）
   #CC=/usr/local/bin/gcc
   ```

2. src または libsrc に変更し、次のように入力します。

   ```
   # gmake （または make）
   ```

2

FCL（File Change Log）

この章では、次の内容について説明します。

FCL（File Change Log）ファイルについて

VxFS FCL（File Change Log）は、ファイルシステム内のファイルおよびディレクトリへの変更を記録します。一般的に FCL を使うアプリケーションは、通常は次のことを行う必要があります。

- ファイルシステム全体またはサブセットをスキャンする
- 最後のスキャン以降に行われた変更を検出する

これに該当するアプリケーションには、バックアップユーティリティ、Web クローラ、検索エンジン、複製プログラムが含まれることがあります。

メモ : FCL は、データが変更された時刻を追跡して変更の種類を記録しますが、実際のデータの変更内容は追跡しません。データが変更されたファイルを検査して、変更されたデータを特定するのはアプリケーションです。

記録される変更

FCL は、次のようなファイルシステムへの変更を記録します。

- 作成
- リンク
- リンクの解除
- 名前の変更
- データの追加
- データの上書き
- データの切り捨て
- 拡張属性の変更
- ホールのパンチ
- 様々なファイルプロパティの更新

メモ : FCL は、ディスクレイアウトバージョン 6 およびそれ以降でのみサポートされています。

FCL は、ファイルシステムの名前空間にある FCL ファイルと呼ばれるスパースファイルに変更を保存します。FCL ファイルは、常に、*/mount_point*/lost+found/changelog に保存されます。FCL ファイルは通常のファイルと同様に操作できますが、書き込みなど、実行できないユーザーレベルの操作もあります。

メモ : `open`（2）、`lseek`（2）、`read`（2）および `close`（2）の標準システムコールは、FCL ファイルのデータにアクセスできます。`mmap`（2）、`unlink`（2）、`ioctl`（2）などの他のすべてのシステムコールは FCL ファイルに対して使えません。

警告 : 将来の VxFS リリースとの互換性のため、FCL ファイルは名前空間から除外される可能性があります。標準的なシステムコールが働かなくなる可能性があります。したがって、新しいアプリケーションはすべて、プログラミングインターフェースを使って開発することを推奨します。

27 ページの「プログラミングインターフェース」を参照してください。

FCL ファイルの使用

VxFS は、ファイルシステムへの変更を、変更に関する情報を FCL ファイルに追加することによって追跡します。これにより、次のことを行えるようになります。

- FCL を使って、特定時点の後にファイルシステム全般、または特定のファイルに対して実行された操作の順序を判断します。たとえば、増分バックアップのアプリケーションは FCL ファイルをスキャンすると、ファイルシステムが最後にバックアップされてから追加されたファイルや、変更されたファイルを判断できます。
- FCL を設定し、ファイルのオープン回数、I/O 統計、アクセス情報（たとえばユーザー ID）などの追加情報を、その他の変更とともに追跡します。
  次に、この情報を使って次の情報を収集できます。
  - 領域の使用状況に関する統計。データの種類が異なる場合に、領域がどのように使われるかを判断できます。
  - ファイルシステム上の異なるファイルについての、異なるユーザーにわたる使用状況のプロファイル。最近どのデータが、どのユーザーによってアクセスされたかを判断するのに役立ちます。

  メモ : これらは VxFS 5.0 リリースの新しい機能です。

領域の使用状況

ファイルシステムがほぼ満杯になったとき、FCL を使って、領域の使用状況を追跡することができます。最近作成されたファイル（ファイルの作成）や書き込みの記録について FCL ファイルを検索し、新しく追加されたファイルや、最近サイズが増えた既存のファイルを特定できます。

アプリケーションのニーズによっては、FCL ファイル全体、または指定した時間範囲に対応する FCL ファイルの一部を検索できます。さらに、特定の名前で作成されたファイルを検索できます。たとえばユーザーが、大量の容量を使う *.mp3 ファイルをダウンロードしている場合、FCL ファイルを読み取り、*.mp3 という名前を付けて作成されたファイルを検索できます。

システムの完全スキャンの削減

VxFS は、FCL 対応のファイルシステム上で更新操作が実行されるたびに、FCL レコードを作成し、ログに記録します。更新操作には、作成、削除、名前の変更、モードの変更、書き込みが含まれます。したがって、増分バックアップのアプリケーションや、ファイル名、ファイル属性、内容のいずれかに基づいてファイルシステムのインデックスを維持するアプリケーションは、FCL ファイルを読み取り、以前のバックアップやインデックスの更新以後に変更されたファイルを検出することによって、システムの完全スキャンを回避できます。

ファイル履歴のトレース

FCL ファイルをスキャンし、ファイルの FCL レコードシーケンスをまとめることによって、ファイルの履歴をトレースできます。また、ファイルの作成、属性の変更、レコードの書き込み、ファイルの削除に関連した FCL レコードを使って、ファイルの履歴を追跡することもできます。

FCL ログ記録のアクティブ化

デフォルトで、FCL ロギングは非アクティブであり、`fcladm` コマンドを使うと、ファイルシステム単位で有効（アクティブ）にできます。

`fcladm`（1M）のマニュアルページを参照してください。

FCL がアクティブになっている場合、ファイルシステムに変更が発生したときに FCL ファイルに新しい FCL レコードが追加されます。FCL をオフにすると以後の記録は停止されますが、FCL ファイルは `lost+found/changelog` に残されます。FCL ファイルを削除できるのは、`fcladm rm` コマンドを使う場合のみです。

FCL には、FCL に記録されたイベントの一覧のほかに、FCL のレイアウトや内部表現を表す関連付けされたバージョンがあります。新しいバージョンの VxFS がリリースされるときには常に、次のことが起きます。

- FCL に記録される追加のイベントが存在することがある
- FCL の内部表現が変更されることがある

これにより、FCL のバージョンが更新されることになります。たとえば、VxFS 4.1 ではデフォルトはバージョン 3 でした。VxFS 5.0 ではデフォルトはバージョン 4 です。FCL バージョン 4 はバージョン 3 では利用可能でない追加のイベントのセット（ファイルのオープンなど）を記録します。VxFS 4.1 で開発されたアプリケーションに後方互換を提供するため、VxFS 5.0 には、アクティブ化中に

FCL バージョン（3 と 4 のどちらか）を指定するオプションが用意されています。指定したバージョンに応じて、新しいレコードの種類をログに記録することを許可するか、許可しないかが決まります。

VxFS 5.0 で新しく追加されたレコードの大半（たとえば、ファイルのオープンや I/O 統計など）については、ログ記録は省略可能で、デフォルトではオフにされます。これらのイベントの記録は、`fcladm` コマンドの *`set`*|*`clear`* オプションを使って有効または無効にできます。

FCL メタ情報は、ファイルシステムの状態、バージョン、追跡されるイベントのセットから構成され、再ブートとファイルシステムのマウント解除やマウントにわたって永続します。バージョンとイベントの情報は、FCL を再度アクティブ化しても永続します。

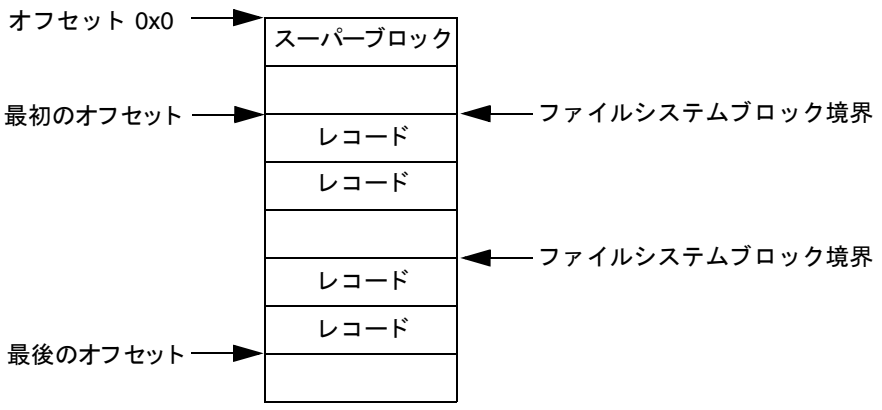
FCL ファイルのレイアウト

VxFS 4.1 では、FCL ファイルの内部レイアウトはユーザーに開示されており、アプリケーションは、`open`（2）、`read`（2）、`lseek`（2）などの標準のファイルシステムインターフェースを使って FCL にアクセスする必要がありました。ただしこの方法は、将来の互換性の問題につながる可能性があります。たとえば、基本の FCL レイアウトと FCL バージョンが変更された場合、それらの変更に対応するためにアプリケーションを変更し、再コンパイルする必要があります。

VxFS 5.0 には、オンディスク FCL レイアウトが変更された場合でもより高い互換性が提供される、新しいプログラミングインターフェースが導入されました。この API を使うと、アプリケーションにとって FCL レイアウトは問題にはなりません。そのため、ここでは FCL レイアウトについては基本的な説明のみを行います。

図 2-1 は、FCL ファイル形式を示しています。FCL ファイルは通常、FCL スーパーブロックと FCL レコードを含むスパースファイルです。FCL ファイルの最初の情報ブロックは、FCL スーパーブロックです。このブロックの次に、ファイルシステムで発生した変更に関する情報を含む FCL レコードはもちろん、省略可能なホールが続く場合があります。

図 2-1 FCL ファイル形式



FCL スーパーブロック

ファイルシステム内のファイルやディレクトリへの変更は、FCL レコードとして格納されます。スーパーブロックは、現在は FCL ファイルの先頭ブロック内に格納され、FCL ファイルの状態を示します。スーパーブロックは次の情報を示します。

- FCL ログが有効かどうか
- FCL がアクティブ化された時間
- 最初と最後の FCL レコードの現在のオフセット
- FCL ファイルのバージョン
- 現在追跡されているイベントセットのイベントマスク
- イベントマスクが最後に変更された時間

`fcladm on` コマンドを使って FCL が最初にアクティブ化されるときには、スーパーブロックのみを含む FCL ファイルが作成されます。スーパーブロックが削除されるのは、`fcladm rm` コマンドを使って FCL ファイルが削除されるときのみです。

`fcladm on` を使って FCL がアクティブ化されるときには、スーパーブロックの状態とアクティブ化時間が変更されます。ファイルシステムの任意のアクティビティが、結果的に FCL ファイルに追加されるレコードになると、最後のオフセットが更新されます。

イベントマスク

FCL ファイルのサイズが大きくなるにつれて、ファイルシステムのチューニングパラメータに応じて（`fcl_maxalloc` と `fcl_keeptime`）、最初のオフセットが更新されるときに、領域を空けるために FCL の先頭にある最も古いレコードが破棄されます。FCL で追跡されるイベントのセットが `fcladm set|clear` を使って変更されると、イベントマスクとイベントマスクの変更時間の更新も発生します。イベントマスクの変更は、結果として、FCL ファイルのログに記録される古いイベントマスクと新しいイベントマスクを含むイベントマスクの変更レコードにもなります。

FCL レコード

FCL レコードには、次の一般的な変更に関する情報が含まれます。

- 変更されたファイルの i ノード番号
  48 ページの「i ノード」を参照してください。
- 変更の時間
- 変更の種類
- レコードの種類に応じた省略可能な情報
  レコードの種類に応じて、FCL には次の情報が含まれる場合もあります。
  - 親 i ノード番号
  - ファイルの削除やリンクなどに関するファイル名
  - ファイルオープンレコードのコマンド名
  - I/O 統計レコードなどの実際の統計

20 ページの図 2-1 を参照してください。

レコードタイプ

表 2-1 に、FCL レコードタイプが生成される操作を示します。

表 2-1　　FCL レコードタイプ

FCL レコードが作成される操作	レコードタイプ
既存のファイルまたはディレクトリへのリンクの追加	VX_FCL_LINK
ファイルに追加書き込み	VX_FCL_DATA_EXTNDWRITE
ファイルまたはディレクトリの作成	VX_FCL_CREATE
名前付きデータストリームディレクトリの作成	VX_FCL_CREATE
シンボリックリンクの作成	VX_FCL_SYMLINK

表 2-1 FCL レコードタイプ

FCL レコードが作成される操作	レコードタイプ
共有および書き込み可能モードでファイルに mmap を実行	VX_FCL_DATA_OVERWRITE
Storage Checkpoint からファイルを移動	VX_FCL_UNDELETE
ホールをファイル内にパンチ	VX_FCL_HOLE_PUNCHED
ファイルまたはディレクトリを削除	VX_FCL_UNLINK
名前付きデータストリームディレクトリの削除	VX_FCL_UNLINK
ファイルまたはディレクトリの名前を変更	VX_FCL_RENAME
ファイル名を既存のファイル名に変更	VX_FCL_UNLINK VX_FCL_RENAME
ファイル属性（割り当てポリシー、ACL、拡張属性）の設定	VX_FCL_EATTR_CHG
ファイルのエクステント予約の設定	VX_FCL_INORES_CHG
ファイルエクステントサイズの設定	VX_FCL_INOEX_CHG
ファイルグループ所有権の設定	VX_FCL_IGRP_CHG
ファイルモードの設定	VX_FCL_IMODE_CHG
ファイルサイズの設定	VX_FCL_DATA_TRUNCATE
ファイルのユーザー所有権を設定	VX_FCL_IOWN_CHG
ファイルの mtime を設定	VX_FCL_MTIME_CHG
ファイルの切り捨て	VX_FCL_DATA_TRUNCATE
ファイル内の既存ブロックへの書き込み	VX_FCL_DATA_OVERWRITE
ファイルのオープン	VX_FCL_FILEOPEN
ファイルの I/O 統計を FCL に書き込み	VX_FCL_FILESTATS
FCL で追跡されるイベントのセットの変更	VX_FCL_EVNTMASK_CHG

メモ：`fcladm on` コマンドによって FCL がアクティブ化されると、表 2-1 に一覧表示されているすべてのイベントがデフォルトで記録されますが、`fileopen` と `filestat` は記録されません。これらの各イベントのアクセス情報もデフォルトでは記録されません。`fcladm` コマンドの *set* オプションを使って、オープン回数、I/O 統計、アクセス情報を記録します。
`fcladm`（1M）のマニュアルページを参照してください。

表 2-1 のレコードの種類は、`fcl_chgtype.t` に属します。これは、41 ページの表 2-2 の `fcl.h` ヘッダーファイルに定義された列挙型です。

特殊レコード

次のレコードタイプは、API 経由では可視でなくなりました。

- `VX_FCL_HEADER`
- `VX_FCL_NOCHANGE`
- `VX_FCL_ACCESSINFO`

一般的なレコードの順番

FCL ファイルには、ファイルシステム内のファイルが作成されたときから削除されるまでのライフサイクルが記録されます。FCL ファイルを作成するとき、通常、次の FCL レコードが順番にログに書き込まれます。
`VX_FCL_CREATE`
`VX_FCL_FILEOPEN`（ファイルオープンの追跡が有効にされている場合）
`VX_FCL_DATA_EXTNDWRITE`
`VX_FCL_IMODE_CHG`

ファイルを書き込むときには、書き込み操作ごとに、次のいずれかの FCL レコードがログに書き込まれます。書き込みがファイルの現在の末尾を越えるか、ファイル内部であるかによってレコードが異なります。
`VX_FCL_DATA_EXTNDWRITE`
`VX_FCL_DATA_OVERWRITE`

次に、a というファイルの名前が b に変更され、両方のファイルがファイルシステム内にある場合にログに書き込まれる FCL レコードの一般的な順番を示します。
`VX FCL_UNLINK`　（ファイル b がすでに存在する場合のファイル b のため）
`VX_FCL_RENAME`　（a から b に名前を変更するため）

FCL チューニングパラメータ

vxtunefsコマンドを使って、4 つの FCL チューニングパラメータを設定できます。vxtunefs（1M）のマニュアルページを参照してください。

fcl_keeptime

FCL レコードが、パージされる前に、FCL ファイル内に留まる時間を秒単位で指定します。パージされる場合、最も古いレコードからパージされます。最も古いレコードはファイルの先頭にあります。さらに、FCL への割り当てが fcl_maxalloc バイトを超えた場合に、ファイルの最初にあるレコードはパージされる可能性があります。デフォルト値は 0 です。fcl_maxalloc パラメータが設定されている場合、FCL に割り当てられた領域が fcl_maxalloc を超えると、レコードがログに留まっていた時間が fcl_keeptime の値より小さくても、レコードは FCL からパージされます。

推奨されるチューニング：fcl_keeptime チューニングパラメータは、レコードが確実に fcl_keeptime の長さだけ保存されるようにする必要がある場合にのみチューニングする必要があります。fcl_keeptime パラメータは、FCL のスキャンの間隔よりも大きな値に設定する必要があります。たとえば、FCL を 24 時間ごとにスキャンする場合は、fcl_keeptime を 25 時間に設定できます。これにより、FCL レコードを読み取り、処理する前にパージされてしまうことがなくなります。

fcl_maxalloc

FCL ファイルに割り当てる領域の最大容量をバイト単位で指定します。割り当てられた領域が fcl_maxalloc を超えた場合、ファイルの先頭にホールがパンチされます。この結果、最も古いレコードがパージされ、FCL スーパーブロックの最初の有効なオフセットが更新されます。

推奨されるチューニング：fcl_maxalloc の最小値は 4 MB です。最大割り当ては、管理者が FCL に割り当てるファイルシステムサイズに対する比率によって指定できます。デフォルト値は、ファイルシステムサイズの 3% です。

`fcl_winterval`

FCL が同じ i ノードに対する複数の上書き、拡張書き込み、または切り捨てレコードを記録する間隔を秒単位で指定します。これにより、FCL ファイル内の繰り返しレコード数を減らすことができます。`fcl_winterval` タイムアウトは i ノードごとです。i ノードがキャッシュから削除され、再度キャッシュに復帰する場合、その書き込み間隔はリセットされます。結果として、同じ書き込み間隔でファイルに複数のレコードを書き込めることになります。デフォルト値は 3600 秒です。

推奨されるチューニング：`fcl_winterval` チューニングパラメータは一般に、FCL のスキャンの間隔より短い時間の値に設定する必要があります。たとえば、FCL を 24 時間ごとにスキャンする場合、`fcl_winterval` は 24 時間未満に設定する必要があります。これにより、次のスキャンまでの間に、上書き、拡張または切り捨てが行われた各ファイルに対し、FCL に少なくとも 1 つのレコードが存在することになります。

`fcl_ointerval`

その範囲内であれば次のファイルのオープンによって追加の FCL レコードを作成しない間隔を秒単位で指定します。これは、とりわけ NFS 経由で頻繁にアクセスがある場合など、FCL のログにファイルオープンのレコードが繰り返し記録される数を減らすのに役立ちます。アクセス情報の追跡も有効にされている場合は、`fcl_ointerval` の間隔内に発生する次のファイルオープンのイベントによってレコードが作成されることがあります。これは、後のファイルオープンが異なるユーザーによって行われた場合です。同様に、i ノードがキャッシュから削除されて戻る場合や、FCL の同期がある場合は、同じオープン間隔内にファイルオープンのレコードが 2 つ以上存在することがあります。デフォルト値は 600 秒です。

推奨されるチューニング：ファイルオープンのレコードを使うアプリケーションで知る必要のあることが、前回 FCL をスキャンしてからファイルがユーザーにアクセスされたかどうかのみであれば、`fcl_ointerval` の期間を、スキャン間の時間の範囲内に設定できます。アプリケーションですべてのアクセスを追跡する必要がある場合は、チューニングパラメータを 0 に設定できます。

ファイルシステムが NFS 上で広範にアクセスされる場合、プラットフォームと NFS 実装によっては、非常に多くのファイルオープンのレコードがログに記録されることがあります。そのような場合は、FCL が繰り返しのレコードでいっぱいにならないように、チューニングパラメータをより高い値に設定することが推奨されます。

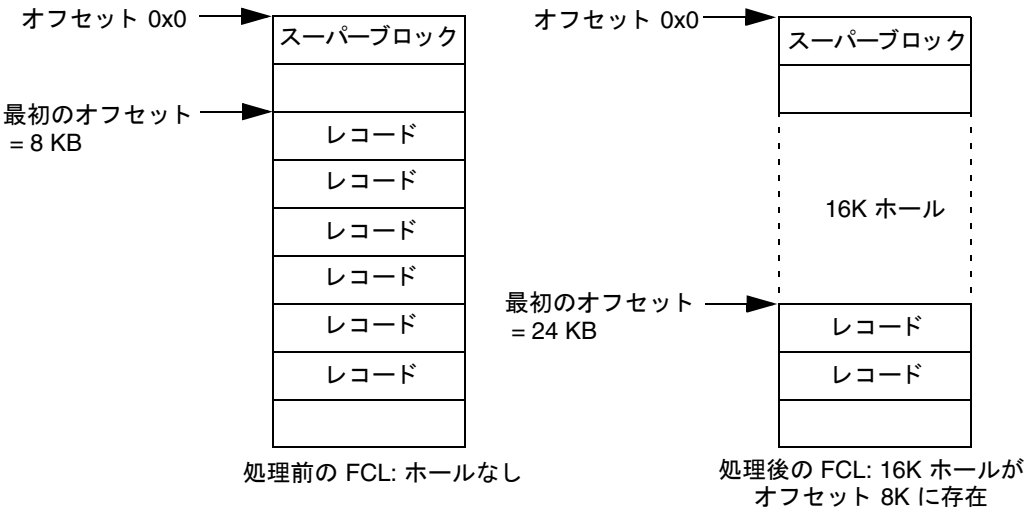
チューニングパラメータで FCL の拡張サイズを処理する方法

図 2-2 は、FCL ファイルのサイズが拡張するときのレコードページの例を示しています。左側の FCL ファイルには、8K のブロックがありますが、ホールはありません。ファイルシステムで発生したアクティビティは、FCL に記録され、拡張の結果、FCL ファイルは右側のようになります。

FCL サイズが、`fcl_maxalloc` チューニングパラメータで指定された最大許容サイズに到達する場合、古いレコードはパージされ領域が空けられます。FCL プログラムは、`fcl_keeptime` で指定された時間より長い時間を経過したレコードをパージするだけです。

空けられた領域は、常に内部ホールサイズの単位になります。図 2-2 では、ファイルシステムは FCL の領域を 8K 単位で空けます。FCL ファイルが初めて最大割り当てを超えたときに、古いレコード数が 20K の場合、プログラムは 16K をパージします。これにより、FCL スーパーブロックの次に 16K のホールが空きます。FCL スーパーブロックの最初の有効なオフセットは、24K に更新されます。

図 2-2 FCL レコードのページ例



プログラミングインターフェース

libvxfsutil: vxfs_fcl_sync を通して開示される既存のプログラミングインターフェースに加えて、VxFS 5.0 では、open（2）、lseek（2）、read（2）、close（2）という標準のシステムコールのセットによって FCL ファイルにアクセスする機構を置き換える、新しいプログラミングインターフェースのセットが提供されます。この API によって次の強化が行われています。

使いやすさ

この API によって、FCL エントリを解析する追加のコードを記述する必要が減ります。オンディスクの FCL レコードの大半は固定サイズで、i ノード番号やタイムスタンプなどのデフォルトの情報のみを含みます。ただし、一部のレコードは可変サイズのことがあります（たとえば、ファイルの削除やレコード名の変更）。これらのレコードには、削除されたファイルや名前が変更されたファイルの名前など、追加の情報が含まれます。

任意のファイルシステムブロックの先頭にある最初の数バイトが、（ファイル名がブロックの境界を越える場合に）常に有効な FCL レコードであるように、複数のオンディスクレコードにわたってファイルシステムブロックが分割されることがあります。以前は、ファイル名を取得するためにこれらのレコードを組み立てる追加のコードを記述する必要がありました。VxFS 5.0 API では、組み立てられた単一の論理レコードを直接読み取る機構が提供されます。API を使うことによってアプリケーションをより簡単に使えます。この API を使うと、アプリケーションでは必要なイベントのサブセットを示すフィルタを指定し、必要なレコードのみを返すことができます。

後方互換

この API により、アプリケーションは、FCL のレイアウト変更に関係なく FCL を読み取ることができます。たとえば、アプリケーションがオンディスクの FCL レコードに直接アクセスして解釈するシナリオを考えてみましょう。次の VxFS リリースで、新しいレコードが加えられたり、FCL ファイルにレコードが格納される方法が変更されたりした場合、（以前の VxFS バージョンにおける）変更に対応するため、アプリケーションを書き直すか、少なくとも再コンパイルする必要があります。

中間 API を使うと FCL のオンディスクのレイアウトはアプリケーションから隠されるため、FCL のディスクレイアウトが変更された場合でも、API が内部的にデータを変換し、必要な出力レコードをユーザーに返します。以後はユーザーアプリケーションを、再コンパイルや書き直しをしないで使い続けることができます。これにより、プログラムは FCL のレイアウト変更から隔離されて、既存のアプリケーションにより優れた互換性が提供されます。

API 関数

この API は次の種類の関数を使います。

- FCL レコードにアクセスするための関数
- オフセットとタイムスタンプをシークするための関数

FCL レコードにアクセスするための関数

次に示すのは、FCL レコードにアクセスするための一般的な関数です。

`vxfs_fcl_open`	FCL ファイルを開き、以後の操作に使えるハンドルを返します。API 経由で行われる、FCL ファイルの以後のアクセスはすべて、このハンドルを使う必要があります。
`vxfs_fcl_close`	FCL ファイルを閉じて、ハンドルに関連付けされたリソースをクリーンアップします。
`vxfs_fcl_getinfo`	FCL ファイルの状態（オン / オフ）とともに FCL のバージョン番号を返します。
`vxfs_fcl_read`	ユーザーにとって必要な FCL レコードを、ユーザーに渡されたバッファに読み取ります。
`vxfs_fcl_copyrec`	FCL レコードをコピーします。もとのレコードにポインタが含まれる場合は、新しい場所を指すようにポインタを再配置します。

FCL でオフセットとタイムスタンプをシークするための関数

終了した場所を起点にしたオフセット、または指定時間後の最初のレコードに至るオフセットに基づいて、FCL の特定のポイントにシークするオプションがユーザーに提供されています。次の関数は、FCL でオフセットとタイムスタンプをシークすることができます。

`vxfs_fcl_getcookie`	現在の FCL のアクティブ化時間と現在のオフセットを埋め込んだ非透過の構造体（以後は cookie と呼びます）を返します。この cookie を保存して後で `vxfs_fcl_seek` に渡すと、アプリケーションが前回終了した位置から読み取りを続けられます。
`vxfs_fcl_seek`	渡された cookie からデータを抽出し、指定されたオフセットまでシークします。cookie には、FCL のアクティブ化時間とファイルのオフセットが埋め込まれます。
`vxfs_fcl_seektime`	FCL で、指定する時間後の最初のレコードにシークします。

vxfs_fcl_open

```
int vxfs_fcl_open(char *pathname, int flags, void **handle);
```

この関数は FCL ファイルを開き、たとえば vxfs_fcl_read や vxfs_fcl_seek のような API 経由で FCL にアクセスする、以後の操作すべてで使う必要があるハンドルを返します。

vxfs_fcl_open には、*pathname と **handle という 2 つのパラメータがあります。*pathname には、FCL ファイル名またはマウントポイントを指すポインタを指定できます。*pathname がマウントポイントの場合、vxfs_fcl_open は、FCL がマウントポイントでアクティブ化されているかどうかを自動的に判断し、そのマウントポイントに関連付けされた FCL ファイルを開きます（現在は *mount_point*/lost+found/changelog）。

vxfs_fcl_open は次に、それが有効な FCL ファイルかどうかと、FCL ファイルのバージョンにライブラリとの互換性があるかどうかを判断します。vxfs_fcl_open 関数はそれから、FCL ファイルに関するメタ情報を非透過の内部データ構造体に取り入れて、ポインタを **handle に格納します。

システムコールの lseek（2）や read（2）と同様に、FCL ファイルの **handle には、次の読み取りを開始するファイル内の位置を示す内部オフセットがあります。FCL ファイルが正常に開かれると、このオフセットは FCL ファイルの最初の有効なオフセットに設定されます。

戻り値

正常に完了すると、呼び出し側に 0 が返され、ハンドルはヌル以外になります。正常に完了しないと、API は 0 以外の値を返し、ハンドルはヌルに設定されます。エラーを示すために、グローバルな値 **errno** も設定されます。

vxfs_fcl_close

vxfs_fcl_close は、ハンドルが参照した FCL ファイルを閉じます。このハンドルに割り当てられたデータ構造体はすべてクリーニングされます。vxfs_fcl_close の呼び出し後は、このハンドルを使わないでください。

パラメータ

```
void vxfs_fcl_close(void *handle)
```

*handle は、以前の vxfs_fcl_open の呼び出しによって返された有効なハンドルです。

vxfs_fcl_getinfo

```
int vxfs_fcl_getinfo(void *handle, struct fcl_info*fclinfo);
```

vxfs_fcl_getinfo 関数は、*fcl_info* によって示した FCL の情報構造体内にある、FCL ファイルに関する情報を返します。この情報は、FCL スーパーブロックから入手されます。

```
struct fcl_info {
    uint32_tfcl_version;
    uint32_tfcl_state;
};
```

各 FCL バージョンに関連付けされたレコードの種類を認識するインテリジェントアプリケーションは、*fcl_version* を使って、FCL ファイルに必要な情報が含まれるかどうかを判断できます。たとえば、バージョン 3 の FCL を使って、インテリジェントアプリケーションは、FCL レコードにアクセス情報がないと推定できます。さらに、fcl_state が FCLS_OFF である場合、アプリケーションは、ファイルシステムのアクティビティのために、FCL ファイルに追加されるレコードがないことも推定できます。

戻り値

0 は成功を示します。そうでない場合は、**errno** がエラーに設定され、0 以外の値が返されます。

vxfs_fcl_read

この関数を使うと、アプリケーションは FCL に論理レコードとして記録された実際のファイルまたはディレクトリの変更情報を読み取れます。各レコードは、struct fcl_record 型を返します。vxfs_fcl_read を使うと、アプリケーションで、必要なイベントのセットから成るフィルタを指定できます。

パラメータ

```
int vxfs_fcl_read(void *hndl, char *buf, size_t *bufsz,
uint64_t eventmask, uint32_t *nentries);
```

入力

この関数は次の入力があります。

- **hndl* は、以前の vxfs_fcl_open の呼び出しによって返されたポインタです。
- **buf* は、サイズが少なくとも **bufsz* のバッファを指すポインタです。
- **bufsz* はバッファのサイズを指定します。
- *eventmask* は、アプリケーションで必要なイベントのセットを指定するビットマスクです。これは、fcl.h ヘッダーで指定されているイベントマスクのセットの論理和にしてください。たとえば、*eventmask* が (VX_FCL_CREATE_MASK | VX_FCL_UNLINK_MASK) である場合、vxfs_fcl_read はファイルの作成と削除のレコードのみを返します。アプリケーションが、21 ページの表 2-1 に一覧表示されているすべてのレコードを読み取る必要がある場合、デフォルトの *eventmask* マスクに FCL_ALL_V4_EVENTS を指定することができます。これにより、FCL ファイル内の有効なバージョン 4 の FCL レコードがすべて返されます。

メモ : VX_FCL_EVNTMASKCHG_MASK が *eventmask* に設定され、vxfs_fcl_read によって返されたレコードに VX_FCL_EVNTMASK_CHG レコードが含まれている場合、そのレコードは常に、バッファ内にある最後のレコードです。これによってアプリケーションは、必要に応じて *eventmask* を再調整できます。さらに、アプリケーションは *eventmask* の変更レコードから、特定のイベントがもう記録されていないことを検出した場合に、以後の読み取りをやめることを決定できます。

- **nentries* は、この vxfs_fcl_read 呼び出しのバッファに読み取る必要があるエントリの数を指定します。**nentries* が 0 の場合、vxfs_fcl_read はバッファに収まるだけ多くのエントリを読み取ります。**nentries* が 0 以外で使用可能レコード数が **nentries* より少ない場合は、vxfs_fcl_read は使用可能な数だけのエントリを読み取り、読み取ったエントリの数を反映するように **nentries* を更新します。

出力

**buf* には、エラーがなければ、struct fcl_record 型の FCL レコードが **nentries* 個含まれます。

要求した数のエントリ（**nentries*）が、バッファのサイズに収まらない場合は、FCL_ENOSPC エラーが返されます。この場合、要求されるレコード数に必要なバッファサイズが格納されるように **bufsz* が更新されます。アプリケーションでこの方法を使ってより大きなサイズのバッファを再度割り当てて、もう一度 vxfs_fcl_read を呼び出すと便利です。**bufsz* は、エラーがない場合は変更されません。

vxfs_fcl_read は、FCL_ENOSPC を返した場合でも、不完全な FCL レコードをバッファに読み取る可能性があります。したがって、バッファの内容を信頼しないでください。

nentries* は、vxfs_fcl_read を呼び出してエラーがない場合に、バッファで読み取られたエントリの数を含むように更新されます。アプリケーションがファイルの最後に到達し、読み取るレコードがなくなったときには、nentries* と戻り値はどちらも 0 です。

注意 : 使用可能レコード数が **nentries* より少ない場合に、FCL_ENOSPC エラーが返されたのであれば、バッファには不正なデータが格納されている可能性があります。

戻り値

0 は成功を示し、0 以外の値はエラーを示します。

メモ : バッファに現在のレコードを格納する十分な領域がない場合、FCL_ENOSPC が返されます。バッファに必要な最小サイズが *bufsz に返されます。

---

vxfs_fcl_read の正常な呼び出し後は、現在のファイルの位置が進められるため、次の vxfs_fcl_read の呼び出しでは、次のレコードのセットが読み取られます。

vxfs_fcl_copyrec

vxfs_fcl_copyrec は、長さ *len* の FCL レコードを **src* から **tgt* にコピーし、**src* のデータのコピーを指すようにコピー先の FCL レコードのポインタを再配置します。FCL レコードをコピーする特別な操作が必要なのは、**src* から **tgt* への FCL レコードの単純なメモリのコピーでは、**tgt* にあるポインタが、データのコピーではなくコピー元の FCL レコードのデータを指したままになるためです。これにより、コピー元の FCL レコードのメモリが再利用されたときに問題を引き起こす可能性があります。

パラメータ

int vxfs_fcl_copyrec(*struct fcl_record *src*, *struct fcl_record *tgt*, *size_t len*);

- **src* は、有効な *fcl_record* 構造を指す必要があります。
- 渡される *len* はコピー元の FCL レコードの長さにしてください。この長さは、コピー元の FCL レコードの *fr_reclen* フィールドで示されます。
- **tgt* は、コピー先の FCL レコードを保持するために使われる、ヌル以外のメモリアドレスを指す必要があります。

メモ : vxfs_fcl_copyrec の呼び出し側は、コピー先の FCL レコードを保持するのに必要な領域が割り当てられていることを確認する必要があります（つまり、少なくとも *len* バイトが必要）。

---

vxfs_fcl_getcookie

vxfs_fcl_getcookie 関数と vxfs_fcl_seek 関数は、アプリケーションが以前処理した FCL ファイル内の位置を記憶しておくのに効果的です。これはその後、処理を再開するポイントとして使えます。これはアプリケーションにとって非常に便利なツールです。

33 ページの「vxfs_fcl_seek」を参照してください。

vxfs_fcl_getcookie 関数は、FCL ファイルの現在のアクティブ化時間と、FCL ファイル内の現在の位置を示すオフセットが埋め込まれている、非透過の fcl_cookie 構造体を返します。この cookie を vxfs_fcl_seek に渡すと、cookie で定義されている FCL ファイル内の位置にシークできます。

典型的な増分バックアッププログラムやインデックス更新プログラムは、FCL ファイルを最後まで読み取り、FCL レコードに基づいて処理を実行できます。アプリケーションは vxfs_fcl_getcookie を使って、FCL ファイル内の現在の位置に関する情報を取得し、ファイルのような永続的な構造に cookie を格納できます。アプリケーションが次回、増分操作を実行する必要があるときには、cookie を読み取って vxfs_fcl_seek に渡し、前に終了したポイントまでシークします。これによってアプリケーションは、新しい FCL レコードのみを読み取れます。

パラメータ

int vxfs_fcl_getcookie(*void *handle, struct fcl_cookie *cookie*)

- **handle* は、vxf_fcl_open の呼び出しによって返される FCL ファイルのハンドルです。
- **cookie* は、次のように定義された非透過のデータブロックを指すポインタです。

```
struct fcl_cookie {
    char    fc_bytes[24];
};
```

この cookie に格納されるデータは VxFS ライブラリに対して内部的です。アプリケーションでは、cookie の内部表現を仮定したり、cookie のデータを改変したりしないでください。

vxfs_fcl_seek

vxfs_fcl_seek を使うと、渡すフラグに応じて FCL ファイルの開始点または終了点にシークできます。

32 ページの「vxfs_fcl_getcookie」を参照してください。

パラメータ

int vxfs_fcl_seek(*void *handle, struct fcl_cookie *cookie, int where*)

- **handle* は、最近の vxfs_fcl_open の呼び出しによって返されたハンドルと同じにしてください。これは必ずしも、vxfs_fcl_getcookie で使われるハンドルと同じではありません。アプリケーションは 1 つのセッションで FCL ファイルを開き、cookie を取得し、FCL ファイルを閉じてから、後のセッションで FCL ファイルを開き、保存された cookie を提出することがあります。有効なハンドルは、FCL ファイルに対して開かれたセッションごとに、そのセッションの vxfs_fcl_open によって返されたハンドルです。

- **cookie* は、vxfs_fcl_getcookie の呼び出しで返された有効な cookie を指す必要があります。
  - 同じ FCL ファイル
  - 同じ FCL ファイルのチェックポイントの 1 つ
  - 同じ FCL ファイルのダンプまたはリストアされたコピーの 1 つ

  特定の cookie にとって有効な FCL ファイルを決定し、目的にかなう組み合わせで FCL ファイルを使うのはユーザーアプリケーションの役割です。

  メモ : **cookie* は、*where* が FCL_SEEK_START または FCL_SEEK_END の場合にヌルになることがあります。

- *where* は、FCL_SEEK_START、FCL_SEEK_END、FCL_SEEK_COOOKIE のいずれかにしてください。
  - *where* が FCL_SEEK_START または FCL_SEEK_END である場合、**cookie* 引数は無視され、vxfs_fcl_seek は FCL ファイルの開始点または終了点のどちらかにシークします。つまり、値はそれぞれ、最初の FCL レコードが始まる位置、または最後のレコードが終わる位置です。
  - *where* が FCL_SEEK_COOKIE である場合、vxfs_fcl_seek は **cookie* に格納されているアクティブ化時間とオフセットを抽出します。

  アプリケーションが最後に vxfs_fcl_getcookie 関数を実行したときから FCL が非アクティブ化（オフに）されている場合や、**cookie* に含まれるオフセットの位置にあったレコードがホールのパンチによってパージされた場合、vxfs_fcl_seek は FCL_EMISSEDRECORD エラーを返します。エラーが返されなければ、vxfs_fcl_seek は、現在のファイルの位置を、cookie に含まれているオフセットに設定します。さらに vxfs_fcl_read を呼び出すと、このオフセットからレコードが返されます。

戻り値

0 は成功を示し、0 以外の値はエラーを示します。

メモ : vxfs_fcl_seek は、FCL が再度アクティブ化された場合（つまり、FCL に記録されているアクティブ化時間が cookie で渡されたものとは異なる場合）や、FCL ファイルの最初の有効なオフセットが、cookie 内にあるオフセットより大きい場合には、FCL_EMISSEDRECORD を返します。

vxfs_fcl_seektime

vxfs_fcl_seektime 関数は、指定する時間と同じかそれ以降のタイムスタンプを持つ、FCL ファイルの最初のレコードにシークします。

パラメータ

int vxfs_fcl_seektime(*void *handle, struct fcl_timeval time*)

- **handle* は、以前の vxfs_fcl_open の呼び出しによって返された有効なハンドルです。
- *time* は、次のように定義された fcl_time_t 構造体型です。

```
struct fcl_time {
    uint32_t tv sec;
    unit32_t tv_nsec;
} fcl_time t;
```

メモ : fcl_time_t で指定する時間は、秒またはナノ秒が単位になることがありますが、gettimeofday のような標準のシステムコールによって返される時間は、秒またはマイクロ秒が単位になることがあります。したがって、変換が必要なことがあります。

vxfs_fcl_seektime は、FCL のエントリはタイムスタンプが減少しない順序になっていることを前提にして、線形より高速な（バイナリ）検索を実行し、指定する時間より大きなタイムスタンプを持つ FCL レコードを特定します。つまり、vxfs_fcl_seektime は、線形検索によって実行されるシークとは異なるレコードにシークすることがあります。

このため、vxfs_fcl_seektime インターフェースの信頼性は 100% ではありません。次の状況では、FCL のタイムスタンプの順序が正しくないことがあります。

- システムの時間が修正された場合
- FCL ファイルがクラスタにマウント済みのファイルシステムに置かれており、異なるノードの時間が同期していない場合

警告 : クラスタファイルシステムでは、システムクロックを同期しておくしくみ（NTP-Network Time Protocol など）を使って、vxfs_fcl_seektime インターフェースの正確さが確実に維持されるようにする必要があります。

戻り値

vxfs_fcl_seektime は成功すると 0 を返します。*time* パラメータの時間以降にレコードがない場合、vxfs_fcl_seektime は EINVAL を返します。

vxfs_fcl_sync

vxfs_fcl_sync 関数は FCL ファイル内に同期ポイントを設定します。この関数は後方互換のために維持されます。

VxFS 5.0 API が FCL ファイルにアクセスできるようにする前に、通常、アプリケーションは vxfs_fcl_sync を呼び出し、FCL を安定状態にして、読み取りを停止するための参照ポイントとして使うオフセットを FCL ファイルに設定します。次にアプリケーションはオフセットを格納し、最後に FCL を読み取ったときからファイルが変更されたかどうかを判定するのにそのオフセットを使います。次に vxfs_fcl_sync は、ファイルへの書き込みまたはファイルのオープンがあったかどうかをチェックします。同期オフセットの後に、対応する書き込みまたはオープンのレコードが少なくとも FCL に 1 つあるはずです。たとえ最後のレコードが書き込まれてから fcl_winterval または fcl_ointerval で指定された時間が経過していなくても、このようになります。

現在、VxFS 5.0 API FCL アクセスでは、vxfs_fcl_open を通して FCL ファイルが開かれるときに、同期が自動的に行われます。vxfs_fcl_open 関数は、同期ポイントを設定し、参照終了のオフセットを内部で決定します。

パラメータ

int vxfs_fcl_sync(*char *fname, uint64_t *offp*);

- **fname* は、FCL ファイル名を指すポインタです
- **offp* は、64 ビットオフセットのアドレスです

vxfs_fcl_sync は FCL ファイルを安定状態にして、アプリケーションが参照ポイントとして使用できるオフセットを使って **offp* を更新します。

FCL レコード

アプリケーションは vxfs_fcl_read 関数を通して FCL ファイルを読み取ります。vxfs_fcl_read は、次のタスクを実行します。

- FCL ファイルからデータを読み取る
- データを fcl_record 構造体に組み立てる
- アプリケーションに渡されたバッファにそれらのレコードを埋め込む

それぞれの fcl_record 構造体は、FCL に記録された論理イベントを表します。この構造体は次のように定義されています。

```
struct fcl_record {
  uint32_t  fr_reclen;         /* レコード長 */
  uint16_t  fr_op;             /* 操作の種類 */
  uint16_t  fr_unused1;        /* 未使用のフィールド */
  uint32_t  fr_acsinfovalid : 1;     /* fr_acsinfo フィールドは有効 */
  uint32_t  fr_newnmvalid : 1;  /* fr_newfilename フィールドは有効 */
  uint32_t  fr_pinogenvalid : 1; /* fr_fr_pinogen フィールドは有効 */
  uint32_t  fr_unused2 : 29;         /* 将来の使用 */
  uint64_t  fr_inonum;             /* i ノード番号 */
  uint32_t  fr_inogen;             /* i ノード世代数 */
  fcl_time_t fr_time;              /* 時間 */
  union fcl_vardata {
    char                  *fv_cmdname;
    struct fcl_nminfo      fv_nm;
    struct fcl_iostats    *fv_stats;
    struct fcl_evmaskinfo  fv_evmask;
} fr_var;
  uint64_t                 fr_tdino;     /* アクセス先ディレクトリ i ノード */
  char                    *fr_newfilename;         /* 名前の変更用 */
  struct fcl_acsinfo      *fr_acsinfo;             /* アクセス情報 */
};

struct fcl_nminfo {
  uint64_tfn_pinonum;/* 親 i ノード番号 */
  uint32_tfn_pinogen;/* 親 i ノード世代数 */
  char*fn_filename;
};

struct fcl_evmaskinfo {
  uint64_toldmask;/* 古いイベントマスク */
  uint64_tnewmask;/* 新しいイベントマスク */
};
```

定義

これらの定義はアクセスを容易にするために提供されています。

```
#define fr_cmdname     fr_var.fv_cmdname
#define fr_stats       fr_var.fv_stats
#define fr_oldmask     fr_var.fv_evmask.oldmask
#define fr_newmask     fr_var.fv_evmask.newmask
#define fr_pinonum     fr_var.fv_nm.fn_pinonum
#define fr_pinogen     fr_var.fv_nm.fn_pinogen
#define fr_filename    fr_var.fv_nm.fn_filename
```

fcl_iostats 構造体

VxFS 5.0 は、ファイルで起きる読み取りや書き込みの数などのファイルの I/O 統計を収集します。収集された統計は、ファイルごとのコア内構造体に維持され、FCL が統計の永続的な外部格納ストアとして機能します。統計情報は次の状況で FCL に書き込まれます。

- 統計情報がリセットされたとき
- 定期的な間隔で

これらの統計は、VX_FCL_FILESTAT レコードとして FCL から読み取れます。各レコードには、次の fcl_iostat 構造体で定義されたとおりの情報が含まれます。

```
struct fcl_iostats {
    uint64_t  nbytesread;   /* ファイルから読み取るバイト数 */
    uint64_t  nbyteswrite;  /* ファイルに書き込むバイト数 */
    uint32_t  nreads;       /* ファイルからの読み取り数 */
    uint32_t  nwrites;      /* ファイルに対する書き込み数 */
    uint32_t  readtime;     /* 読み取りの合計時間（秒単位）*/
    uint32_t  writetime;    /* 書き込みの合計時間（秒単位）*/
    struct {
        uint32_ttv_sec;
        uint32_ttv_nsec;
    } lastreset;/*          前回の統計のリセット時間 */
    uint32_t  nodeid;       /* レコード書き込み元のノード */
    uint32_t  reset;        /* リセットのために統計が書き込まれました */
};
```

FCL の iostat レコードはそれぞれ、*lastreset* 時間から次の *lastreset* 時間までの期間中に蓄積された I/O 統計を含みます。期間中の累積統計や集計は次のようにして算出できます。

- FCL をスキャン
- VX_FCL_FILESTATS 型のレコードを検索

たとえば、期間中の読み取り回数の合計を集計するには、一連の FCL ファイルをスキャンして、I/O 統計レコードを入手する必要があります。この情報には、

同じ *lastreset* 時間を持つ一連の VX_FCL_FILESTATS 型レコードが入っています。また、そのレコードに続いて、以降の *lastreset* 時間を持つ特定のファイル用の一連のレコードも入っています。

集計では、同じ *lastreset* 時間を持つレコード群の最終レコードの値のみを考慮し、次にこのような各レコードの読み取り回数を合計します。

fcl_acsinfo structure

アクセス情報の追跡を有効にすると、VxFS は各レコードとともに、次のようなアクセス情報をログに記録します。

- アプリケーションにアクセスしている有効な実際のユーザーとグループの ID
- ファイルのアクセス元のノード
- ユーザーアプリケーションのプロセス ID と各レコード

アプリケーションが FCL を読み取るとき、情報は fr_acsinfo フィールドに返されます。fr_acsinfo は、次のように定義されている FCL_acsinfo 構造体を指します。

```
struct fcl_acsinfo {
    uint32_tfa_ruid;
    uint32_tfa_rgid;
    uint32_tfa_euid;
    uint32_tfa_egid;
    uint32_tfa_pid;
    uint32_tfa_nodeid;
};
```

メモ：accessinfo は、独立したレコードの種類としては返されませんが、追加情報として他のレコードとともに返されます。さらに、accessinfo の情報は、すべてのレコードに存在するとは限りません（有効になっていない場合）。ただし、ファイルシステムのいくつかの内部操作（ファイルの切り捨てなど）で accessinfo が有効にされたときでも、アクセス情報が存在しないことがあります。アクセス情報を利用可能かどうかを判断する助けになるように、FCL レコードには fcl_acsinfovalid という名前のフラグが含まれ、そのフラグは特定のレコードに accessinfo が存在する場合にのみ 0 以外になります。

---

fcl_acsinfo 構造体のいくつかのフィールドはポインタなので、実際の内容を格納するためのメモリが必要です。これは、FCL レコードの直後に実際のデータを格納し、そのデータを指すポインタを更新することによって処理されます。レコード長の fr_reclen フィールドは、データ全体に相当するように更新されます。したがって、vxfs_fcl_read によって返される各 FCL レコードは可変サイズのレコードで、その長さは *fr_reclen_field* によって示されます。
40 ページの図 2-3 は、サンプルのリンクレコードで、データがどのようにレイアウトされるかを示しています。

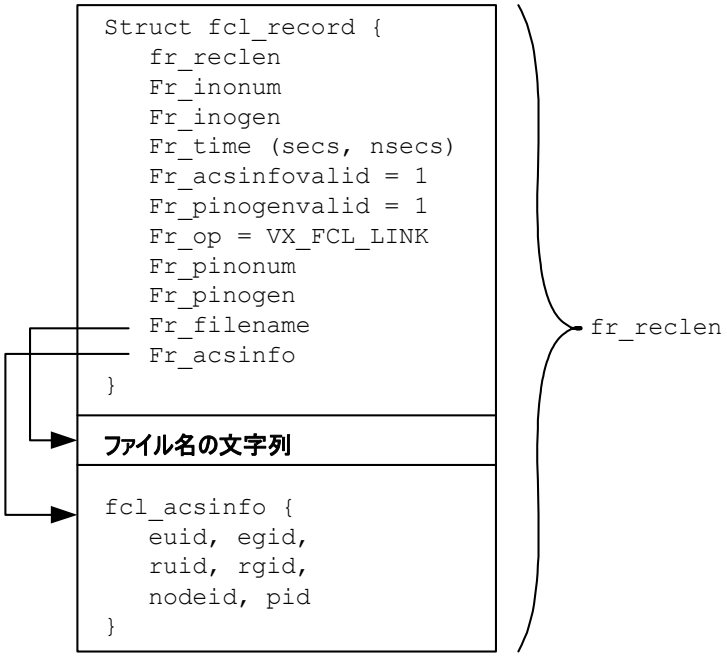


図 2-3　　サンプルのリンクレコード

次のコードサンプルは、vxfs_fcl_read の呼び出しによって返されたレコードのセットをスキャンし、ユーザー ID を出力します。

```
Struct fcl_record*fr;
Char        *tbuf;
…
         error = vxfs_fcl_read(fh, buf, &bufsz,
                           FCL_ALL_V4_EVENTS,
                           &nentries);
    tbuf = buf;
    while (--nentries) {
        fr = (struct fcl_record *)tbuf;
if (fr->fr_acsinfovalid) {
        printf("Uid %ld\n", fr->fr_acsinfo->uid;
}
        tbuf += fr->fr_reclen;
    }
```

メモ：FCL_ALL_V4_EVENTS はイベントマスクです。

30 ページの「vxfs_fcl_read」を参照してください。

レコードの構造体のフィールド

表 2-2 は、fcl_record 構造体の各フィールドを説明し、対象となる有効なレコードの種類を示します。

表 2-2 FCL レコードの構造体のフィールド

フィールド	説明	有効性
fr_reclen	FCL レコードの長さ。これには、FCL レコードの構造体の長さと、構造体の直後に格納されているデータの長さが含まれます。この長さは、vxfs_fcl_read によってバッファで返された FCL レコードをスキャンするときに使ってください。	すべてのレコードに対して有効です。
fr_inonum	変更されたファイルの i ノード番号。変更されたオブジェクトのフルパス名は、vxfs_inotopath_gen を使い、i ノード番号と世代数（fr_inogen）から作成されます。	レコードが FCL_EVNTMSK_CHG である場合を除き、すべての FCL レコードに対して有効です。イベントマスクの変更の場合、ファイルは暗黙として FCL ファイルです。
fr_op	この FCL レコードの操作（たとえば、作成、リンク解除、書き込み、ファイル属性の変更、その他の変更など）が含まれます。fr_op には、表 2-1 に一覧表示されているレコードの種類のいずれかの値が採用されます。 このパラメータを使って、FCL レコードのどのフィールドが有効であるかを判定します。	すべてのレコードに対して有効です。
fr_time	変更が FCL ファイルに記録された、およその時間。このフィールドを解釈するには、ctime 呼び出しを使います。	すべてのレコードに対して有効です。
fr_inogen	変更されたファイルの世代数。世代数と、（ファイルの）i ノード番号を組み合わせて vxfs_inotopath_gen に渡すことで、オブジェクトの正確なフルパス名を取得できます。世代数を指定しないと、返されるパス名は再利用された i ノードである可能性があります。	イベントマスクの変更とリンク解除を除き、すべての FCL レコードに対して有効です。イベントマスクの変更の場合、i ノード番号と世代数が暗黙です。リンク解除の場合、名前の逆引きルックアップによってファイル名を取得するために世代数は必要ありません。これは、ファイル名がレコードとともにすでに存在するためです。

表 2-2 FCL レコードの構造体のフィールド

フィールド	説明	有効性
fr_pinonum fr_pinogen fr_filename	削除や名前の変更のような FCL レコードについては、ディレクトリエントリが削除された場合、名前の逆引きルックアップによってファイル名を判断できません。同様にリンクレコードの場合、あいまいでない方法でファイル名を判断できません。したがってこれらの場合、変更されるファイルを含む親ディレクトリのファイル名、i ノード番号、世代数が記録されます。親ディレクトリの i ノード（fr_pinonum）と世代数（fr_pinogen）を、名前の逆引きルックアップ API とともに使うと、親ディレクトリの絶対パス名を識別できます。後続ファイル名を追加して、オブジェクトの完全な名前を生成できます。	FCL レコードが VX_FCL_UNLINK、VX_FCL_RENAME、VX_FCL_LINK のいずれかであるときに有効です。リンク解除と名前の変更、ファイル名と親 i ノード番号、世代数には、削除された古いファイルに関する情報が含まれます。リンクについては、新しいファイル名を表します。
fr_cmdname	fr_inonum と fr_inogen によって表されるファイルを開いたコマンドの短い名前です。	FCL レコードが VX_FCL_FILEOPEN である場合にのみ有効です。
fr_stats	FCL_iostat レコードを指すポインタ。fcl_iostat レコードには、ファイルで起きる読み取りや書き込みの数、読み取りや書き込みの平均時間などの I/O 統計が含まれます。これらの特定時点のレコードを使うと、ファイルについて一定期間にわたる集計や平均の I/O 統計を計算できます。	FCL レコードが VX_FCL_FILESTATS である場合にのみ有効です。
fr_oldmask fr_newmask	これらのフィールドはそれぞれ、古いイベントマスクと新しいイベントマスクを含みます。各イベントマスクは、fcl.h で定義されているマスクのセットの論理和です。	FCL レコードが VX_FCL_EVNTMASK_CHG である場合にのみ有効です。
fr_acsinfo	FCL_acsinfo 構造体を指すポインタ。この構造体は、特定の操作を実行したアプリケーションのユーザーやグループの ID、プロセス ID、アクセスしているノードの ID などの情報を含みます。	有効性は、fcl_acsinfovalid bit-field により判断されます。これは、潜在的にすべての種類のレコードとともに存在できます。これはオプションのフィールドです。

FCL レコードのコピー

vxfs_fcl_read によって返される各 FCL レコードは可変サイズで、fcl_record 構造体と、それに続く、レコードと関連付けされた追加のデータから構成されます。fcl_record 構造体内のポインタは、fcl_record 構造体の後に格納されたデータを指し、レコード長が、可変サイズレコードのサイズを指定します。ただし、FCL レコードのコア内コピーを作成するときには、コピー元から fr_reclen バイトのデータをコピー先に複製するだけではすみません。

単純なメモリのコピーでは、コピー元のレコードを、ポインタを通してコピー先のレコードに単にコピーします。この場合、コピー先のレコードに、コピー元のデータを指すポインタが残されます。最終的に、コピー元レコードのメモリが再利用されたり空けられたりしたときに問題を引き起こす可能性があります。レプリカ内のポインタは、コピー先レコードのデータを指すように修正する必要があります。したがって、FCL レコードのコア内コピーを作成するには、アプリケーションは、vxfs_fcl_copyrec 関数を使い、コピーした後にポインタの再配置を実行する必要があります。ユーザーアプリケーションはコピーに必要なメモリを割り当てる必要があります。

インデックス保守アプリケーション

このアプリケーション例は、Linux の locate プログラムに似た高速検索を可能にするために、ファイルシステムですべてのファイルのインデックスを維持するシステム用のものです。インデックスを定期的に更新するか、最後のインデックス更新以降のファイル変更で必要になるごとに更新する必要があります。基本的な実行手順と FCL API の呼び出し例を次に示します。

準備

アプリケーションを準備するには

1 FCL を有効にします。

```
# fcladm on mntpt
```

2 *fcl_keeptime* と *fcl_maxalloc* を必要な値に調整します。

```
# vxtunefs -o fcl_keeptime=value
# vxtunefs -o fcl_maxalloc=value
```

インデックス保守アプリケーションの最初の実行

アプリケーションをテストするには

1 FCL ファイルを開きます。

```
# vxfs_fcl_open(mntpt, 0, &fh);
```

2 終了点にシークします。

```
# vxfs_fcl_seek(fh, NULL, FCL_SEEK_END);
```

3 cookie を取得してファイルに格納します。

```
# vxfs_fcl_getcookie(fh, &cookie)
write(fd, cookie, sizeof(struct fcl_cookie));
```

4 インデックスを作成します。

5 FCL ファイルを閉じます。

```
# vxfs_fcl_close(fh);
```

インデックスを更新するための定期的な実行

アプリケーションを更新するには

1 FCL ファイルを開きます。

```
# vxfs_fcl_open(mntpt, 0, &fh);
```

2 cookie を読み取り、cookie までシークします。

```
# read(fd, &cookie, sizeof(struct fcl_cookie))
# vxfs_fcl_seek(fh, cookie, FCL_SEEK_COOKIE)
```

3 FCL ファイルを読み取り、それに応じてインデックスを更新します。

```
# vxfs_fcl_read(fh, buf, BUFSZ, FCL_ALL_v4_EVENTS, &nentries)
```

4 cookie を取得してファイルに格納し直します。

```
# vxfs_fcl_getcookie(fh, &cookie)
# write(fd, cookie, sizeof(struct fcl_cookie));
```

5 FCL ファイルを閉じます。

```
# vxfs_fcl_close(fh);
```

使用状況プロファイルのコンピューティング

次のアプリケーション例では、特定のファイルの使用状況プロファイル（つまり、特定のファイルにこの 1 時間内にアクセスしたユーザー）を計算します。

初期設定

このアプリケーション例には、ファイルオープンの追跡やアクセス情報などの追加情報が必要です。これらの情報は、FCL バージョン 4 でのみ利用可能です。正しい FCL バージョンを有効にしてください。

次の手順で、必要な初期設定を実行します。

アプリケーションを設定するには

1 FCL バージョン 4 をオンにします。

```
# fcladm -o version=4 on mntpt
```

メモ : この手順が失敗した場合は、fcladm print を使って、FCL バージョン 3 ファイルが存在するかどうかをチェックします。存在した場合は、fcladm rm を使って FCL バージョン 3 ファイルを削除してから、FCL バージョン 4 をオンにします。

VxFS 5.0 では、デフォルトの FCL バージョンはバージョン 4 です。既存の FCL ファイルが存在しない場合は、fcladm on *mntpt* コマンドを実行するとバージョン 4 の FCL が自動的に作成されます。

2 アクセス情報、ファイルオープン、I/O 統計の追跡を必要に応じて有効にします。

```
# fcladm set fileopen,accessinfo mntpt
```

3 fcl_keeptime、fcl_maxalloc、fcl_ointerval の各チューニングパラメータを必要に応じて設定します。次に例を示します。

```
# vxtunefs fcl_ointerval=value
```

4 FCL ファイルを閉じます。

```
# vxfs_fcl_close(fh);
```

手順の例

アプリケーションを利用する手順の例を次に示します。

1 FCL ファイルを開きます。

```
vxfs_fcl_open(mntpt, 0, &fh);
```

2 シークを実行するための時間を設定します。

- **a** gettimeofday を使って現在の時刻を取得します。
- **b** fcl_time_t に 1 時間前の時刻を設定します。
- **c** FCL ファイルでその時刻のレコードにシークします。

```
gettimeofday(&tm, NULL);
tm.sec -= 3600
vxfs_fcl_seektime(fh, tm);
```

3 適切なイベントマスクを使って、ファイルを終端まで読み取ります（アプリケーションファイルには、ファイルオープンのレコードとアクセス情報のみ必要です）。

- **a** ファイルの i ノード番号と世代数が、各 FCL レコードでシークされているものと同じであるかどうかをチェックします。
- **b** ファイルにアクセスしたユーザーに関する情報を必要に応じて表示します。

```
vxfs_fcl_read(fh, buf, BUFSZ, VX_FCL_FILEOPEN_MASK | \
VX_FCL_ACCESSINFO_MASK, &nentries);
```

オフホスト処理

シナリオによっては、ユーザーアプリケーションは、実働サーバーの帯域幅を節約し、FCL 処理のジョブを別のシステムにアウトソースするように選択することがあります。オフホスト処理を行うには、FCL ファイルをオフホストシステムに送る必要があります。FCL ファイルは通常のファイルとは異なるため、cp や ftp などのコマンドは機能しません。

FCL ファイルを送るには、fcladm dump コマンドを使って、まず FCL ファイルを通常のファイルにダンプする必要があります。その後、通常のファイル転送プログラムを使って、ファイルをオフホストシステムに転送できます。次の例を参照してください。

```
# fcladm -s savefile dump mntpt
# rcp savefile offhost-path
```

オフホストシステムで、fcladm コマンドの *restore* オプションを使って FCL ファイルを復元する必要があります。もとの FCL ファイルと異なり、復元したファイルは通常のファイルです。

```
# fcladm -s savefile restore restorefile
```

復元した FCL ファイルは、FCL API で使うために、引数として vxfs_fcl_open に渡すことができます。

警告 : 名前の逆引きルックアップ API は、オフホストシステムでは機能しません。オフホスト処理の機構は、アプリケーションが i ノード番号と世代数に対応している場合か、または i ノード番号に依存しないでファイル名を決定する方法がアプリケーションにある場合のみ使ってください。

VxFS と FCL のアップグレードとダウングレード

VxFS 4.1 は、FCL バージョン 3 のみをサポートしています。VxFS 5.0 は、FCL バージョン 3 とバージョン 4 の両方をサポートしていて、デフォルトはバージョン 4 です。システムを VxFS 4.1 から VxFS 5.0 にアップグレードするとき、ファイルシステムで FCL がオンになっていれば、既存のバージョン 3 の FCL ファイルはそのまま保持されます。VxFS 5.0 は、VxFS 4.1 の場合とまったく同様に、バージョン 3 の FCL でファイルシステム変更の追跡を続行します。

`read`(2) システムコールを使って FCL ファイルに直接アクセスする VxFS 4.1 アプリケーションは、FCL ファイルがバージョン 3 であれば、VxFS 5.0 でも動作を続行できます。ただし、新しいアプリケーションを開発する際には、API を使う必要があります。API は、FCL バーション 3 とバージョン 4 の両方をサポートしています。

VxFS 5.0 で新しく追加されたレコードの種類（ファイルオープンやアクセス情報など）を新しいアプリケーションで使う場合は、FCL はバージョン 4 である必要があります。

FCL バージョン 3 を直接読み取るアプリケーションをまだ実行している場合、それらのアプリケーションが新しい API を使うように書き直されるまで、FCL バージョン 4 にアップグレードできません。この API は、バージョン 3 とバージョン 4 を両方解釈できるので、バージョン 3 がまだ有効なまま、この API を使うようにアプリケーションをアップグレードできます。

FCL バージョン 4 へのバージョン 3 ファイルの変換

FCL バージョン 3 からバージョン 4 に移動するためのパスを次に示します。

1 FCL をオフにします。

```
# fcladm off mntpt
```

2 既存の FCL ファイルを削除します。

```
# fcladm rm mntpt
```

3 必要なバージョンで再度アクティブ化します。

```
# fcladm [-oversion=4] on mntpt
```

VxFS バージョンのダウングレード

データセンター操作の実行時に、新しいバージョン の VxFS を実行中のホストから古いバージョンを実行中のホストへ（たとえば VxFS 5.0 から VxFS 4.1 へ）、VxFS ファイルシステムを移行する必要がある可能性があります。このような移行は、オフホスト処理で発生することがあります。VxFS 5.0 で生成された FCL ファイルが FCL バージョン 3 である場合、VxFS 4.1 でそのまま使い続けることができます。ただし、FCL ファイルがバージョン 4 である場合は、VxFS 4.1 で使う前に、まず `fcladm rm` を使って削除し、FCL バージョン 3 として再度アクティブ化する必要があります。

このような処理を行わない場合、VxFS 5.0 で生成されたバージョン 4 の FCL ファイルを VxFS 4.1 で開こうとすると、操作は失敗します。

メモ : FCL のイメージを保存して送る前に、FCL を含むノードで同期化を呼び出すのは、オフホストアプリケーションの役割です。同期化によって FCL が安定状態になり、同期後のすべてのファイル書き込みやファイルオープンで FCL レコードが生成されるようになります。

パス名の逆引きルックアップ

パス名の逆引きルックアップ機能を使うと、ファイルやディレクトリの i ノード番号から、完全パス名を取得できます。i ノード番号は、ライブラリ関数 `vxfs_inotopath_gen` への引数として指定します。詳細については、`vxfs_inotopath_gen`（3）のマニュアルページを参照してください。

パス名の逆引きルックアップ機能は、次のような様々なアプリケーションで便利に使えます。

- VxFS FCL 機能のクライアント
- バックアップユーティリティとリストアユーティリティ
- レプリケーション製品

ファイルやディレクトリのパス名は非常に長くなることがあり、パス名を簡単に取得する手段が必要になります。そのため、これらのアプリケーションでは、通常、情報を i ノード番号で保存します。

i ノード

i ノード番号は、ファイルシステム内の各ファイルに付けられた一意の識別番号です。i ノードには、そのファイルに関連付けられたデータとメタデータが格納されます。ただし、その i ノードに対応するファイル名は含まれていません。したがって、i ノード番号からファイルの名前を特定することは、比較的難しくなります。`ncheck` コマンドを実行すると、ファイルシステム内の各ディレクトリがスキャンされ、i ノードの識別子からファイル名が取得されます。ただし、このプロセスには、長い時間がかかることがあります。VxFS のパス名の逆引きルックアップ機能を使うと、パス名を比較的すばやく取得できます。

メモ : シンボリックリンクにはファイルのパスが含まれていないため、パス名の逆引きルックアップ機能を使っても、ファイルへのシンボリックリンクを追跡することはできません。

ファイルの i ノード番号、世代数、および後続ファイル名（VX_FCL_LINK、VX_FCL_UNLINK、または VX_FCL_RENAME レコードの場合）を、パス名の逆引きルックアップを使って組み合わせると、各 FCL レコードへの完全パス名を生成できます。

vxfs_inotopath_gen

vxfs_inotopath_gen 関数は、マウントポイント名、i ノード番号、および i ノード世代数を受け取り、バッファを返します。このバッファには、i ノードを表す 1 つまたは複数（1 つの i ノードに複数リンクのリンクが張られている場合）の完全パス名が含まれています。i ノード世代数パラメータを指定すると、再利用された i ノードの誤った値がパス名として返されることはありません。このため、できるだけ、vxfs_inotopath_gen 関数を使ってください。

vxfs_inotopath 関数は、後方互換のためにのみ含まれています。
vxfs_inotopath 関数は i ノード世代数を引数として取得しません。

次に、vxfs_inotopath と vxfs_inotopath_gen の構文を示します。

```
int vxfs_inotopath(char *mount_point, uint64_t inode_number,
                        int all, char ***bufp, int *inentries)

int vxfs_inotopath_gen(char *mnt_pt, uint64_t inode_number,
                        unint32_t inode_generation, int all,
                        char ***bufp, int *nentries)
```

vxfs_inotopath 呼び出しの場合、すべての引数を、単一のパス名を得るための 0、またはすべてのパス名を得るための 1 にする必要があります。
mount_point 引数には、ファイルシステムマウントポイントを指定します。正常に返されると、*bufp* はパス名と、エントリ数を格納する nentries を格納する 2 次元文字ポインタをポイントします。返された 2 次元配列の各エントリは、サイズが *MAXPATHLEN* であり、アプリケーションを呼び出すことによって、配列自体とともに解放する必要があります。

vxfs_inotopath_gen 呼び出しは、vxfs_inotopath 呼び出しと同じですが、追加のパラメータ *inode_generation* を使う点が異なります。
vxfs_inotopath_gen 関数は、渡された *inode_generation* が i ノード番号の現在の世代に一致する場合に、指定された i ノード番号に関連付けられた 1 つまたは複数のパス名を返します。世代が異なる場合は、エラーが返されます。世代数が不明の場合、*inode_generation*=0 を指定します。これは vxfs_inotopath 呼び出しを使った場合と同じです。

vxfs_inotopath_gen 呼び出しおよび vxfs_inotopath 呼び出しは、バージョン 6 と 7 のディスクレイアウトでのみサポートされています。

3

MVS（Multi-volume support）

この章では、次の内容について説明します。

MVS について

MVS 機能を利用すれば、従来のようにファイルシステムごとに 1 つのボリュームを使うのではなく、複数の VxVM ボリュームを VxFS ファイルシステムの下位ストレージとして使えます。これらのボリュームごとに、処理効率、冗長性、コストなど、異なる特性を持たせることができます。また、処理効率の向上や管理上の目的で、ファイルシステムの各部分を、別々に切り離して使うこともできます。

管理者およびアプリケーションは、割り当てポリシーを使うことで、どのファイル、どのメタデータをどのボリュームに格納するかを制御できます。領域を割り当てるようなファイルシステム操作が行われるたびに、適用可能な割り当てポリシーが検索され、その操作に指定されているボリュームが検出されます。割り当てポリシーは、通常、新しい割り当てのみに適用されますが、既存のデータを新しい割り当てポリシーに従って移行するためのインターフェースも用意されています。

各割り当てには、次のレベルのポリシーを適用できます。

- ファイル単位のポリシー
- Storage Checkpoint 単位のポリシー
- ファイルシステム単位のポリシー

指定された割り当て操作に対して、最も限定的に定義された割り当てポリシーが使われます。

MVS API は次の基本カテゴリに分類できます。

- ファイルシステム内のボリュームを操作する API
- 割り当てポリシーの定義を操作する API
- 割り当てポリシーを適用する API

各 API は、`fsvoladm`（1M）コマンドおよび `fsapadm`（1M）コマンドにオプションを指定することによっても使えます。

`fsvoladm`（1M）と `fsapadm`（1M）のマニュアルページを参照してください。

MVS の利用

MVS 機能には、次のような利用方法があります。

- 特定のファイルやファイル階層を異なるボリュームに割り当てることができるように、ファイルの保存場所を制御します
- Storage Checkpoint に割り当てられたデータがファイルシステムの他のデータと区別されるように、Storage Checkpoint を分離します
- ファイルシステムメタデータとファイルデータを分離します
- ボリュームがファイルシステムにファイルとして表示されるようにボリュームをカプセル化します。これは、RAW ボリュームで実行されているデータベースでは特に便利です
- ボリュームを交換したり修理したりするために、ファイルをボリュームから移動することができます
- ファイルシステムを定期的にスキャンし、ストレージの使用パターンの変更に応じて割り当てポリシーを変更できるような、ストレージ最適化アプリケーションを構築することができます
- ボリュームの可用性 - MVS では、利用できないボリュームがあるときも、一部のボリュームを利用できます。これにより、コンポーネントデータ専用ボリュームが 1 つ以上失われた場合でも、MVS ファイルシステムをマウントできます。

メモ : ボリュームの可用性は、ディスクレイアウトバージョン 7 およびそれ以降のファイルシステムでのみサポートされています。バージョン 7 のディスクレイアウトは、可変長で大きいサイズの履歴ログレコード、2048 を超えるボリューム、大きいディレクトリハッシュ、DST（Dynamic Storage Tiering）のサポートを可能にします。

ボリューム API

ボリューム API を使うと、ファイルシステムへのボリュームの追加、ファイルシステムからのボリュームの削除、ファイルシステム内のボリュームの一覧表示、ボリューム内の領域の使用情報の取得などを実行できます。

MVS ファイルシステムは、VxVM ボリュームセットと一緒に使う場合に限り使えます。ボリュームセットは vxvset コマンドで管理します。

『Veritas Volume Manager 管理者ガイド』を参照してください。

ボリュームセットの管理

以下の例では、ボリュームセットを管理する方法を示します。

ボリュームをボリュームセットに変換するには

- myvol1 をボリュームセットに変換するには、次の関数呼び出しを使います。

```
# vxvset make myvset myvol1
```

ボリュームをボリュームセットに追加するには

- myvol2 を myvset ボリュームセットに追加するには、次の関数呼び出しを使います。

```
# vxvset addvol myvset myvol2
```

ボリュームセットのボリュームを一覧表示するには

- myvset のボリュームを一覧表示するには、次の関数呼び出しを使います。

```
# vxvset list myvset
```

ボリュームセットからボリュームを削除するには

- myvset から myvol2 を削除するには、次の関数呼び出しを使います。

```
# vxvset rmvol myvset myvol2
```

ファイルシステムのボリュームセットの問い合わせ

次の関数呼び出しでは、ファイルシステムのボリュームセットを問い合わせます。

ファイルシステムに関連付けされたすべてのボリュームを問い合わせるには

- ファイルシステムに関連付けされたすべてのボリュームを問い合わせるには、次の関数呼び出しを使います。

```
vxfs_vol_enumerate(fd, &count, infop);
```

単一のボリュームを問い合わせるには

- 単一のボリュームを問い合わせるには、次の関数呼び出しを使います。

```
vxfs_vol_stat(fd, vol_name, infop);
```

ファイルシステム内のボリュームの変更

次の関数呼び出しでは、ファイルシステム内のボリュームを修正します。

ボリュームを拡大または縮小するには

- ボリュームを拡大または縮小するには、次の関数呼び出しを使います。

```
vxfs_vol_resize(fd, vol_name, new_vol_size);
```

ファイルシステムからボリュームを削除するには

- ファイルシステムからボリュームを削除するには、次の関数呼び出しを使います。

```
vxfs_vol_remove(fd, vol_name);
```

ボリュームをファイルシステムに追加するには

- ボリュームをファイルシステムに追加するには、次の関数呼び出しを使います。

```
vxfs_vol_add(fd, new_vol_name, new_vol_size);
```

ボリュームのカプセル化とカプセル化の解除

次の関数呼び出しでは、ボリュームをカプセル化します。

RAW ボリュームをカプセル化するには

- 既存の RAW ボリュームをカプセル化し、そのボリュームの内容をファイルシステム内の 1 ファイルとして表示するには、次の関数呼び出しを使います。

```
vxfs_vol_encapsulate(encapsulate_name, vol_name, vol_size);
```

RAW ボリュームのカプセル化を解除するには

- 既存の RAW ボリュームのカプセル化を解除し、ファイルをファイルシステムから削除するには、次の関数呼び出しを使います。

```
vxfs_vol_deencapsulate(encapsulate_name);
```

割り当てポリシー API

MVS 機能を十分に活用できるように、VxFS では、ファイルまたはファイルグループをボリュームセット内の指定したボリュームに割り当てるための割り当てポリシーがサポートされます。

割り当てポリシーでは、ボリュームの一覧と割り当ての優先順位を指定します。ポリシーは、ファイル、ファイルシステム、またはファイルシステムから作成された Storage Checkpoint に割り当てることができます。ファイルシステム内のオブジェクトにポリシーを割り当てる場合は、メタデータとファイルデータの両方に割り当てポリシーを指定する必要があります。たとえば、ポリシーを 1 つのファイルに割り当てる場合、ファイルシステムは、ファイルデータとメタデータの両方の配置場所を認識する必要があります。ポリシーを指定しない場合は、ファイルシステムはデータをランダムに配置します。

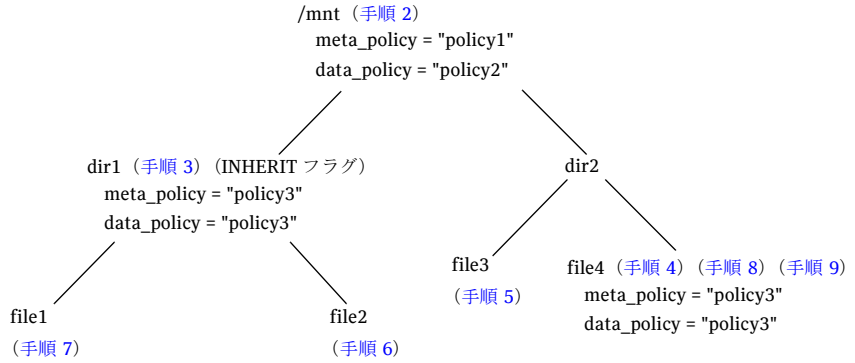
割り当てポリシーはファイルシステムごとに定義でき、いったん定義した後は保持されます。ファイルシステムで定義できる割り当てポリシーの数に制限はありません。いったんポリシーを割り当てると、新しく作成されるファイルの割り当てはそのポリシーにより決定されます。また、ポリシーが定義される前、ポリシーが割り当てられる前、またはファイルに対するポリシーが変更される前に割り当てられたファイルに対して、ポリシーを実施すると、そのファイルを適切なボリュームに再度割り当てることもできます。新しく作成されたファイルは、その親ディレクトリの割り当てポリシーを引き継ぐことができます。これを行うには、親ディレクトリにポリシーを割り当てるときに `FSAP_INHERIT` フラグを指定します。

現時点では、既存のファイルが現在どこに割り当てられているかを特定できるインターフェースはありません。ただし、これらの API を使って、ファイルに対してポリシーを割り当てて実施すると、適切なブロック確保を確実に行うことができます。

ファイル割り当ての指示

図 3-1 は、割り当てポリシーを使ってファイル割り当てを制御する方法を示しています。

図 3-1　　ファイル割り当ての指示



/mnt ファイルシステムには、vol-01、vol-02、vol-03 の 3 つのボリュームから成るボリュームセットがあります。これらのボリュームはそれぞれ、policy1、policy2、policy3 に対応します。

ファイル割り当てを指示するには

1. /mnt ファイルシステムに割り当てポリシーを作成します。
2. データ割り当てポリシーおよびメタデータ割り当てポリシーを、それぞれ policy1、policy2 として /mnt ファイルシステムに割り当てます。
3. データ割り当てポリシーおよびメタデータ割り当てポリシーを、INHERIT フラグを使って、どちらも policy3 として dir1 に割り当てます。
4. file4（100MB）を作成します。このファイルは vol-02 に割り当てられます。
5. file3（10MB）を作成します。このファイルは vol-02 に割り当てられます。
6. file2（100MB）を作成します。このファイルは vol-03 に割り当てられます。
7. file1（100MB）を作成します。このファイルは vol-03 に割り当てられます。
8. データ割り当てポリシーおよびメタデータ割り当てポリシーを、どちらも policy3 として file4 に割り当てます。
9. この割り当てポリシーを file4 に実施します。これにより、このファイルは vol-03 に再度割り当てられます。

このファイルシステムのポリシー割り当てでは、データは policy1 によって割り当てられ、メタデータは policy2 によって割り当てられます。これらのポリシーにより、ファイルは vol-01 および vol-02 に割り当てられますが、dir1 は例外です。このディレクトリには、vol-03 にファイルを割り当てるポリシーが優先します。

file3 と file4 は、作成されると、policy1 と policy2 の制御により、vol-02 に割り当てられます。file1 と file2 は、作成されると、policy3 の制御により、vol-03 に割り当てられます。

file4 は、作成された時点では vol-01 および vol-02 に割り当てられます。file4 を vol-03 に移動するには、policy3 を file4 に割り当て、そのポリシーを実施します。これにより、file4 は vol-03 に再度割り当てられます。

ポリシーの作成と割り当て

次の例では、マルチボリューム API を使ってポリシーを作成し、割り当てます。

ポリシーを作成して割り当てるには

1 ファイルシステムのポリシーを定義するには、次の関数呼び出しを使います。

```
vxfs_ap_define(fd, fsap_info_ptr, 0);
```

2 ポリシーをファイルシステムに割り当てるには、次の関数呼び出しを使います。

```
vxfs_ap_assign_fs(fd, data_policy, meta_policy);
```

3 ポリシーをファイルまたはディレクトリに割り当てるには、次の関数呼び出しを使います。

```
vxfs_ap_assign_file(fd, data_policy, meta_policy, 0);
```

4 ポリシーを Storage Checkpoint に割り当てるには、次の関数呼び出しを使います。

```
vxfs_ap_assign_ckpt( fd, checkpoint_name, data_policy,
    meta_policy);
```

5 パターンに基づく割り当てポリシーをディレクトリに割り当てるには、次の関数呼び出しを使います。

```
vxfs_ap_assign_file_pat(int fd, struct fsap_pattern_table \
#pat_assign, uint32_t flags);
```

60 ページの「パターンに基づくポリシー」を参照してください。

6 パターンに基づく割り当てポリシーをファイルシステムに割り当てるには、次の関数呼び出しを使います。

```
vxfs_ap_assign_fs_pat(int fd, struct fsap_pattern_table \
#pat_assign, uint32_t flags);
```

定義されたポリシーの問い合わせ

次の関数呼び出しでは、定義されているポリシーを問い合わせます。

ファイルシステムのすべてのポリシーを問い合わせるには

- ファイルシステムのすべてのポリシーを問い合わせるには、次の関数呼び出しを使います。

```
vxfs_ap_enumerate(fd, &count, fsap_info_ptr);
```

定義されている単一のポリシーを問い合わせるには

- 定義されている単一のポリシーを問い合わせるには、次の関数呼び出しを使います。

```
vxfs_ap_query(fs, fsap_info_ptr);
```

そのポリシーを割り当てられているファイルを問い合わせるには

- そのポリシーを割り当てられているファイルを問い合わせるには、次の関数呼び出しを使います。

```
vxfs_ap_query_file(fs, data_policy, meta_policy, 0);
```

Storage Checkpoint に割り当てられたポリシーを問い合わせるには

- Storage Checkpoint に割り当てられたポリシーを問い合わせるには、次の関数呼び出しを使います。

```
vxfs_ap_query_ckpt(fd, check_point_name, data_policy,
                                        meta_policy)
```

ディレクトリのパターンテーブルを問い合わせるには

- ディレクトリのパターンテーブルを問い合わせるには、次の関数呼び出しを使います。

```
vxfs_ap_query_file_pat(int fd, struct fsap_pattern_table \
**pat_query, uint32_t flags);
```

ファイルシステム全体のパターンテーブルを問い合わせるには

- ファイルシステム全体のパターンテーブルを問い合わせるには、次の関数呼び出しを使います。

```
vxfs_ap_query_fs_pat(int fd, struct fsap_pattern_table \
**pat_query, uint32_t flags);
```

ファイルへのポリシーの実施

次の関数呼び出しでは、ポリシーを実施します。

ファイルに対してポリシーを実施するには

- ファイルに対してポリシーを実施するには、次の関数呼び出しを使います。
  ```
  vxfs_ap_enforce_file(fd, data_policy, meta_policy);
  ```
  ポリシーを実施することによって、ファイルを別のボリュームに再度割り当てることができます。

ファイルでのポリシーの削除

次の関数呼び出しでは、ポリシーを削除します。

- ファイルでポリシーを削除するには、次の関数呼び出しを使います。
  ```
  vxfs_ap_remove(int fd, char *name);
  ```

パターンに基づくポリシー

fsapadm（1M）コマンドの assignfspat キーワードは、*mount_point* で指定されたファイルシステムにパターンテーブルを割り当てます。

パターンに基づく既存のテーブルがファイルシステムにある場合、テーブルは置き換わります。テーブルを正常に割り当てた後、パターンテーブルまたは継承可能な割り当てポリシーのないディレクトリにファイルが作成されると、ファイルシステムのパターンテーブルは有効になります。ファイルの名前、プロセス作成の UID と GID は、指定された順序でテーブルに定義されたパターンに一致します。最初に一致したパターンがファイルの割り当てポリシーの設定に使われます。

パターンが指定されていない場合、fsapadm コマンドは、既存のファイルシステムのパターンテーブルを削除します。

データ構造

fsap_info と fsdev_info のデータ構造は、vxfsutil.h ヘッダーファイルと libvxfsutil.a ライブラリファイルで調べられます。

vxfsutil.h ヘッダーファイルと libvxfsutil.a ライブラリファイルを参照してください。

簡単に参照できるように、これらのデータ構造体を以下に示します。

```
#define FSAP_NAMESZ           64
#define FSAP_MAXDEVS          256
#define FSDEV_NAMESZ          32

struct fsap_info {            /* ポリシーの構造 */
    char ap_name[FSAP_NAMESZ]; /* ポリシーの名前 */
    uint32_t ap_flags;        /* FSAP_CREATE | FSAP_INHERIT |
```

```
                                        FSAP_ANYUSER */
    uint32_t ap_order;          /* FSAP_ORDER_ASGIVEN |
                                    FSAP_ORDER_LEASTFULL |
                                    FSAP_ORDER_ROUNDROBIN */
    uint32_t ap_ndevs;          /* ボリュームの数 */
    char ap_devs[FSAP_MAXDEVS][FSDEV_NAMESZ];
                                /* このポリシーに関連付けられている
                                    ボリュームの名前 */
};

struct fsap_info {              /* ボリュームの構造 */
    int dev_id;                 /* 0 から n までの数値 */
    uint64_t dev_size;          /* ボリュームのサイズ（バイト）*/
    uint64_t dev_free;
    uint64_t dev_avail;
    char dev_name[FSDEV_NAMESZ];/* ボリュームの名前 */
};
```

ポリシーと API の使用

以下の例は、ボリューム vol-01、vol-02、および vol-03 で構成されたボリュームセット（volset）があることを前提としています。また、volset は、ファイルシステムマウントポイント /mnt にマウントされています。

割り当てポリシーの定義と割り当て

次の疑似コードに、割り当てポリシー API を使って、割り当てポリシーを定義し、割り当てる例を示します。

既存のファイルのデータブロックを特定のボリュームに再度割り当てるように、割り当てポリシーを定義して割り当てるには

- 既存のファイルのデータブロックを特定のボリューム（vol-03）に再度割り当てるには、次のようなコードを作成します。

```
/* ファイルのデータの移動のためのデータポリシーを作成します */

strcpy((char *) ap.ap_name, "Data_Mover_Policy");
ap.ap_flags = FSAP_CREATE;
ap.ap_order = FSAP_ORDER_ASGIVEN;
ap.ap_ndevs = 1;
strcpy(ap.ap_devs[0], "vol-03");

fd = open("/mnt", O_RDONLY);
vxfs_ap_define(fd, &ap, 0);

file_fd = open ("/mnt/file_to_move", O_RDONLY);
vxfs_ap_assign_file(file_fd, "Data_Mover_Policy", NULL, 0);

vxfs_ap_enforce_file(file_fd, "Data_Mover_Policy", NULL);
```

メモ：vxfs_ap_enforce_file2 API は、割り当てポリシーに一致するようにファイルのブロックを再度割り当てます。FSAP_ENF_STRICT フラグによって、割り当て順序どおりに割り当てられます。

vxfs_ap_enforce_file2（3）のマニュアルページを参照してください。

新しいファイルをディレクトリの下に割り当てるポリシーを作成するには

この例では、dir1 以下のファイルのメタデータを vol-01 に割り当て、ファイルデータを vol-02 に割り当てます。

- ディレクトリ dir1 の下に新しいファイルを割り当てるポリシーを作成するには、次のようなコードを作成します。

```
/* ポリシーを 2 つ定義します */

/* RAID5 ポリシーを作成します */

strcpy((char *) ap.ap_name, "RAID5_Policy");
ap.ap_flags = FSAP_CREATE | FSAP_INHERIT;
ap.ap_order = FSAP_ORDER_ASGIVEN;
ap.ap_ndevs = 1;
strcpy(ap.ap_devs[0], "vol-02");

fd = open("/mnt", O_RDONLY);
dir_fd = open("/mnt/dir1", O_RDONLY);

vxfs_ap_define(fd, &ap, 0);

/* ミラーポリシーを作成します */

strcpy((char *) ap.ap_name, "Mirror_Policy");
ap.ap_flags = FSAP_CREATE | FSAP_INHERIT;
ap.ap_order = FSAP_ORDER_ASGIVEN;
ap.ap_ndevs = 1;
strcpy(ap.ap_devs[0], "vol-01");

vxfs_ap_define(fd, &ap, 0);

/* ポリシーをディレクトリに割り当てます */

vxfs_ap_assign_file(dir_fd, "RAID5_Policy", "Mirror_Policy",
                    0);

/* ディレクトリ dir1 にファイルを作成します */
/* file1 のメタおよびデータブロックがそれぞれ vol-01 および vol-02 に
    割り当てられます */

file_fd = open("/mnt/dir1/file1");
write( file_fd, buf, 1024);
```

ボリューム API の使用

次の疑似コードに、ボリューム API を使った例を示します。

ファイルシステム内のボリュームを縮小または拡大するには

1 vxresize コマンドを使って物理ボリュームを拡大します。

2 vxfs_vol_resize() 呼び出しを使ってファイルシステムを縮小または拡大するには、次のようなコードを作成します。

```
/* ボリューム "vol-03" の stat を実行し、サイズ情報を取得します */

fd = open("/mnt");
vxfs_vol_stat(fd, "vol-03", infop);

/* 必要に応じてサイズを変更（縮小 / 拡大）します。この例では、ボリュームを
    半分に縮小します */

vxfs_vol_resize(fd, "vol-03", infop->dev_size / 2);
```

RAW ボリュームをファイルとしてカプセル化するには

1 ボリュームセットにボリュームを追加します。

2 RAW ボリューム vol-03 を、encapsulate_name という名前のファイルとして、ファイルシステム /mnt にカプセル化するには、次のようなコードを作成します。

```
/* RAW ボリューム vol-03 を取得し、それをカプセル化します。
    ボリュームの内容には、指定されたパス名でアクセス可能になります。 */

vxfs_vol_encapsulate("/mnt/encapsulate_name", "vol-03",
                                        infop->dev_size);

/* ボリュームへのアクセスは、ファイル /mnt/encapsulate_name への
    書き込み / 読み取りを通して行います */

encap_fd = open("/mnt/encapsulate_name");
write(encap_fd, buf, 1024);
```

RAW ボリュームのカプセル化を解除するには

◆ ファイルシステム /mnt にある encapsulate_name という名前の RAW ボリューム vol-03 のカプセル化を解除するには、次のようなコードを作成します。

```
/* カプセル化解除を使って RAW ボリュームを削除します。
    カプセル化の解除後、vol-03 はまだ volset の一部ですが、ファイル
    システムのアクティブな部分ではありません。 */

vxfs_vol_deencapsulate("/mnt/encapsulate_name");
```

4

名前付きデータストリーム

この章では、次の内容について説明します。

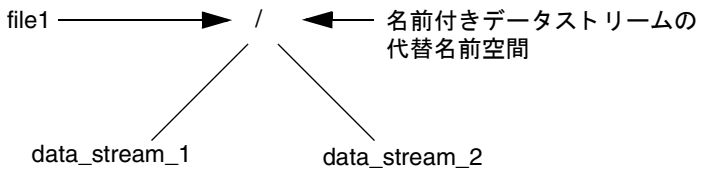
名前付きデータストリームについて

名前付きデータストリームは、複数のデータストリームを 1 つのファイルに関連付けます。デフォルトの名前なしデータストリームは、ファイル名を指定して呼び出した open() 関数が返すファイル記述子からアクセスできます。他のデータストリームは、ファイルに関連付けられている代替名前空間に保存されます。

メモ : 名前付きデータストリームは、名前付き属性と呼ばれる場合もあります。

図 4-1 に、ファイルに関連付けられた代替名前空間を示します。

図 4-1　　代替名前空間



file1 ファイルには 2 つの名前付きデータストリーム（data_stream_1 と data_stream_2）があります。

すべてのファイルが、名前付きデータストリームを保存するためのそれぞれ固有の代替名前空間を持ちます。代替名前空間は、VxFS でサポートされている名前付きデータストリーム API からアクセスできます。

名前付きデータストリームへは、名前付きデータストリームのライブラリ関数を使い、ファイル記述子を介してアクセスできます。アプリケーションは、名前付きデータストリームを開いてファイル記述子を取得し、そのファイル記述子を使って、read()、write() および mmap() 操作を実行できます。これらのシステムコールは、通常のファイルを操作する場合と同様に機能します。ファイルの名前付きデータストリームは、そのファイルに関連付けられている名前付きデータストリームが格納される隠しディレクトリ i ノードに保存されます。ファイルの隠しディレクトリ i ノードには、名前付きデータストリーム API からのみアクセスできます。

VxFS には、この機能を使うための管理コマンドがありません。ファイルの名前付きデータストリームの作成、読み取り、書き込みを行うための VxFS API が提供されています。

この機能には、Solaris 10 の管理コマンドとの互換性があります。

名前付きデータストリームの使用

名前付きデータストリームをアプリケーションで使うと、隠しファイルに情報を付加することができます。管理プログラムでこの機能を使って、ファイルの使用情報、バックアップ情報などを付加することができます。またアプリケーションでこの機能を利用して、付加したファイル情報を隠したり、収集したりすることもできます。たとえば、複数のファイルではなく 1 つのファイルで、あらゆる種類のテキスト、オーディオクリップ、ビデオクリップを持つマルチメディアドキュメントを作成できます。また、複数のユーザーから参照されるドキュメントでは、各ユーザーのコメントを名前付きデータストリームとしてファイルに付加することができます。

名前付きデータストリームのプログラミングインターフェース

次の標準システムコールによって、名前付きデータストリームを操作できます。

`open()`	名前付きデータストリームを開きます
`read()`	名前付きデータストリームを読み取ります
`write()`	名前付きデータストリームに書き込みます
`getdents()`	ディレクトリエントリを読み取り、ファイルシステムに依存しない形式にします
`mmap()`	メモリのページをマップします
`readdir()`	ディレクトリを読み取ります

VxFS の名前付きデータストリーム機能は、以下のアプリケーションプログラミングインターフェース関数から使えます。

vxfs_nattr_check(), vxfs_nattr_fcheck()

これらの関数は、フラグの値に応じた方法で、名前付きデータストリームの存在をチェックします。

vxfs_nattr_check() 関数は、パスで指定されたパス名によって、名前付きデータストリームの存在をチェックします。vxfs_nattr_fcheck() 関数は、fd で指定されたファイル記述子によって、名前付きデータストリームの存在をチェックします。

次に、これらの API の構文を示します。

```
int vxfs_nattr_check(int char .path, int \
flags);
int vxfs_nattr_fcheck(int fd, int flags);
```

vxfs_nattr_open()

open() システムコールと同様の機能を備えています。ただしパスは、指定されたファイル記述子に対する名前付きデータストリームとして解釈されます。vxfs_nattr_open() 操作が正常に完了した場合、戻り値は、名前付きデータストリームに関連付けられているファイル記述子になります。他の入出力関数は、このファイル記述子を使って名前付きデータストリームを参照できます。名前付きデータストリームのパスが「.」に設定されている場合、戻り値のファイル記述子は、「.」ディレクトリ v ノードの名前付きデータストリームを指します。

次に、vxfs_nattr_open() API の構文を示します。

```
int vxfs_nattr_open(int fd, char *path,
    int oflag, int cmode);
```

vxfs_nattr_link()

既存の名前付きデータストリームに新しいディレクトリエントリを作成し、そのリンクカウントを 1 ずつ増やします。名前付きデータストリーム名前空間には、既存の名前付きデータストリームへのポインタと、名前付きデータストリーム名前空間に作成された新しいディレクトリエントリへのポインタがあります。呼び出し元の関数には、名前付きデータストリームをリンクするための書き込み権限が必要です。

次に、vxfs_nattr_link() API の構文を示します。

```
int vxfs_nattr_link(int sfd, char *spath,
    char *tpath);
```

vxfs_nattr_unlink()

指定されたパスにある名前付きデータストリームを削除します。名前付きデータストリームのディレクトリエントリを削除するには、呼び出し元の関数に書き込み権限が必要です。

次に、vxfs_nattr_unlink() API の構文を示します。

```
int vxfs_nattr_unlink(int fd, char *path);
```

vxfs_nattr_rename() path1 で指定されている名前空間エントリを、2 番目の path2 で指定されている名前空間エントリに変更します。指定されたパスは、名前付きデータストリームのディレクトリ v ノードへのポインタに相対的に解決されます。

次に、vxfs_nattr_rename() API の構文を示します。

```
int vxfs_nattr_rename(int sfd, char *old,
    char *tnew);
```

vxfs_nattr_utimes() 名前付きデータストリームのアクセスと変更の時間を設定します。

次に、vxfs_nattr_utimes() API の構文を示します。

```
int vxfs_nattr_utimes(int sfd,
    const char *path,
    const struct timeval times[2]);
```

vxfs_nattr_open（3）、vxfs_nattr_link（3）、vxfs_nattr_unlink（3）、vxfs_nattr_rename（3）、vxfs_nattr_utimes（3）の各マニュアルページを参照してください。

名前付きデータストリームの一覧表示

ファイルの名前付きデータストリームの一覧を表示するには、次の例に示すように、その名前付きデータストリームのディレクトリ i ノードで getdents() を呼び出します。

名前付きデータストリームを一覧表示するには

1 名前付きデータストリームを一覧表示するには、次のようなコードを作成します。

```
fd = open("foo", O_RDWR); /* ファイル foo を開きます */
afd = vxfs_nattr_open(fd, "stream1",
        O_RDWR|O_CREAT, 0777) ;/* ファイル foo の名前付きデータスト
                                  リーム stream1 を作成します */
write (afd, buf, 1024) ;   /* 名前付きデータストリームファイルに書き
                              込みます */
read (afd, buf, 1024) ;    /* 名前付きストリームファイルから読み取り
                              ます */
dfd = vxfs_nattr_open(fd, ".", O_RDONLY);
                           /* ファイル foo の名前付きストリーム
                              ディレクトリを開きます */
getdents (dfd, buf, 1024) ;/* 名前付きストリームディレクトリのディレクトリ
                              エントリを読み取ります */
```

2 名前の逆引きルックアップコールを使って、ストリームファイルをパス名に解決します。その結果、パス名の形式は次のようになります。

/mount_point/file_with_data_stream/./data_stream_file_name

名前付きデータストリームの名前空間

「$vxfs:」で始まる名前は、将来使うために予約されています。名前が「$vxfs:」で始まるデータストリームを作成しようとすると、EINVAL エラーが発生します。

他のシステムコールにおける動作の変更

名前付きデータストリームディレクトリでは、「..」などの一部の属性が定義されていません。これらのフィールドにアクセスする操作は失敗することがあります。名前付きデータストリームディレクトリでディレクトリ、シンボリックリンク、デバイスファイルを作成しようとすると失敗します。また、名前付きデータストリームディレクトリまたは名前付きデータストリーム i ノードを指定して VOP_SETATTR() を呼び出した場合も失敗します。

名前付きデータストリームの問い合わせ

次の例では、API コールを使って、20 個の名前付きデータストリームを持つファイル named_stream_file を作成しています。

名前付きデータストリームは、ls コマンドでは表示されません。名前付きデータストリームは、作成されると、隠しディレクトリに格納されます。次に例を示します。

```
# ls -al named_stream_file
-r-xr-lr-x 1 root  other   1024    Aug 12 09:49named_stream_file
```

名前付きデータストリームに問い合わせるには

- getdents() または readdir_r() システムコールを使い、named_stream_file ファイルに、隠しディレクトリの内容（20 個の名前付きデータストリームを含む）について問い合わせます。

```
Attribute Directory contents for
    /vxfstest1/named_stream_file

0x1ff root other 1K Thu Aug 12 09:49:17 2004 .
0x565 root other 1K Thu Aug 12 09:49:17 2004 ..
0x177 root other 1K Thu Aug 12 09:49:17 2004 stream0
0x177 root other 1K Thu Aug 12 09:49:17 2004 stream1
0x177 root other 1K Thu Aug 12 09:49:17 2004 stream2
.
.
.
0x177 root other 1K Thu Aug 12 09:49:17 2004 stream17
0x177 root other 1K Thu Aug 12 09:49:17 2004 stream18
0x177 root other 1K Thu Aug 12 09:49:17 2004 stream19
```

API

名前付きデータストリーム API は、標準のシステムコールと VxFS API コールを併用して機能を提供します。

以下に、名前付きデータストリームに問い合わせるための擬似コードの例を示します。

```
/* ファイルを作成し、オープン */
if ((fd = open("named_stream_file", O_RDWR | O_CREAT | O_TRUNC,
    mode)) < 0) {
    sprintf(error_buf, "%s, Error Opening File %s ", argv[0],
            filename);
    perror(error_buf);
    exit(-1);
}

/* 従来どおり、通常ファイルに書き込み */
write(fd, buf, 1024);

/* ファイル named_stream_file に名前付きデータストリームを複数作成 */
for (i = 0; i < 20; i++) {
    sprintf(attrname, "%s%d", "stream", i);
    nfd = vxfs_nattr_open(fd, attrname, O_WRONLY | O_CREAT,
        mode);
    if (nfd < 0) {
        sprintf(error_buf,
            "%s, Error Opening Attribute file %s/./%s ",
            argv[0], filename, attrname);
        perror(error_buf);
        exit(-1);
    }
    /* データをストリームファイルに書き込む */
    memset(buf, 0x41 + i, 1024);
    write(nfd, buf, 1024);
    close(nfd);
}
```

コマンドリファレンス

cp、tar、ls または同様のコマンドを使って、名前付きデータストリームのあるファイルをコピーまたは一覧表示する場合、ファイルはコピーまたは一覧表示されますが、添付された名前付きデータストリームはコピーまたは一覧表示されません。

メモ : Solaris 9 以降のオペレーティング環境では、これらのコマンドで指定して、名前付きデータストリームを処理できる -@ オプションがあります。

5

VxFS I/O アプリケーションインターフェース

この章では、次の内容について説明します。

メモ : このマニュアルで説明している他の VxFS API と異なり、この章で説明する API はすべてのプラットフォームの以前のリリースの VxFS で使えます。ただし、VxFS キャッシュアドバイザリによって同時 I/O アクセスを実現する API は例外であり、これらは VxFS 4.1 以降のリリースで使えます。

フリーズとアンフリーズ

ファイルシステムをフリーズすると、ファイルシステムへのすべての I/O 処理が一時的にブロックされ、ファイルシステム上で sync が実行されます。現在の操作が完了して、ファイルシステムはディスクに同期されます。ファイルシステムのフリーズ処理は、ファイルシステムの、ボリュームレベルで一貫性のある安定したイメージを取得するために必要な手順です。

ボリュームレベルで一貫性のあるファイルシステムイメージは、ファイルシステムスナップショットツールを使って取得し、使うことができます。フリーズ処理によって、ダーティなメタデータやユーザーデータが格納されたファイルシステムキャッシュ内のすべてのバッファおよびページがフラッシュされます。さらに、この操作によって、ファイルシステムがアンフリーズされるまで、ファイルシステムに対するすべての新しい操作が保留されます。

VxFS には、アプリケーションプログラムから VxFS ファイルシステムをフリーズおよびアンフリーズするための ioctl インターフェースが用意されています。このインターフェースとは、VX_FREEZE、VX_FREEZE_ALL および VX_THAW のことです。

VX_FREEZE ioctl は、単一のファイルシステムに対して機能します。この ioctl を実行しているプログラムは、指定したファイルシステムをフリーズし、ファイルシステムをアンフリーズするまで、ファイルシステムに対するすべてのアクセスをブロックできます。ファイルシステムは、VX_FREEZE ioctl によって指定されたタイムアウト値の期限が切れるか、または VX_THAW ioctl がファイルシステムに対して実行されると、アンフリーズされます。

VX_THAW ioctl はフリーズされたファイルシステムに対して機能します。このインターフェースを使って、タイムアウト値が経過する前に、指定したファイルシステムをアンフリーズできます。

VX_FREEZE_ALL ioctl インターフェースは 1 つまたは複数のファイルシステムをフリーズします。VX_FREEZE_ALL ioctl は、フリーズ処理で複数のファイルシステムが指定された場合、原子的に実行します。VxFS は指定したファイルシステムへの同時アクセスをブロックし、1 回の書き込み処理で、複数のファイルシステムを変更する可能性のあるユーザー起動の書き込み処理を許可しません。VX_FREEZE_ALL は単一のファイルシステムでも使えるため、VX_FREEZE ioctl よりも VX_FREEZE_ALL の方が望ましいインターフェースです。

VX_FREEZE または VX_FREEZE_ALL ioctl を実行すると、クリーンなファイルシステムイメージが生成されます。このイメージは、ファイルシステムデバイスから分離した後でマウント可能になります。フリーズ要求に応じて、変更されたすべてのファイルシステムメタデータがディスクにフラッシュされます。ログ内には、分離したイメージをマウントする前に再生する必要がある保留中のファイルシステムトランザクションはありません。

VX_FREEZE インターフェースおよび VX_FREEZE_ALL インターフェースのどちらを使っても、ローカルにマウントされたファイルシステム、ローカルまたはリモートでマウントされたクラスタファイルシステムをフリーズできます。

次の表に、VxFS リリースとのフリーズ / アンフリーズの互換性を示します。

	VxFS 3.5	VxFS 4.0	VxFS 4.1	5.0
VX_FREEZE	ローカルファイルシステム	ローカルファイルシステム クラスタファイルシステム	ローカルファイルシステム クラスタファイルシステム	ローカルファイルシステム クラスタファイルシステム
VX_FREEZE_ALL	ローカルファイルシステム	ローカルファイルシステム	ローカルファイルシステム クラスタファイルシステム	ローカルファイルシステム クラスタファイルシステム

ファイルシステムをフリーズする場合は、ファイルシステムをターゲットにしている外部リソースへの影響を軽減するため、フリーズの適切なタイムアウト値を慎重に選択する必要があります。ファイルシステムがフリーズされている間、ユーザーまたはシステムのプロセスおよびリソースはブロックされます。指定したタイムアウト値が大きすぎると、リソースがブロックされる時間が長くなります。

ファイルシステムがフリーズしている間、VX_THAW ioctl に使うために、ファイルシステムのルートディレクトリからファイル記述子を取得しようとすると、ファイルシステムのフリーズ状態の結果として、呼び出し元のプロセスがブロックされます。ファイル記述子は、VX_FREEZE_ALL または VX_FREEZE ioctl を発行する前に取得しておく必要があります。

タイムアウト期間が期限切れになる前に、VX_THAW ioctl を使い、VX_FREEZE_ALL ioctl によってフリーズされたファイルシステムをアンフリーズします。

プログラミングインターフェースは次のようになります。

```
include <sys/fs/vx_ioctl.h>

int             timeout;
int             vxfs_fd;

/*
* timeout のアドレスを指定するというのはよくある間違いです。
* timeout のアドレスは渡さないでください。 非常に長いタイムアウト値と解釈されて
* しまいます
*/
 if (ioctl(vxfs_fd, VX_FREEZE, timeout))
```

```
        {perror("ERROR: File system freeze failed");
    }
```

複数のファイルシステムを対象にする場合は、次のようにします。

```
    int                vxfs_fd[NUM_FILE_SYSTEMS];
    struct             vx_freezeall freeze_info;

    freeze_info.num = NUM_FILE_SYSTEMS
    freeze_info.timeout = timeout;
    freeze_info.fds = &vxfs_fd[0];

    if (ioctl(vxfs_fd[0], VX_FREEZE_ALL, &freeze_info))
        {perror("ERROR: File system freeze failed");
    }

    for (i = 0; i < NUM_FILE_SYSTEMS; i++)
        if (ioctl(vxfs_fd[i], VX_THAW, NULL))
            {perror("ERROR: File system thaw failed");
        }
```

キャッシュアドバイザリ

VxFS を使うと、アプリケーションはファイルにアクセスするときに使うキャッシュアドバイザリを設定できます。キャッシュアドバイザリは、アプリケーションからファイルにアクセスする場合の望ましい選択です。どれを選択するかは、指定したアドバイザリによって実現される最適な処理効率によって決まり、ユーザーデータの一貫性を確保します。たとえば、データベースアプリケーションで、ダイレクト I/O を使って、データベースデータが格納されているファイルにアクセスすることを選択する場合や、バッファされている I/O アドバイザリを選択して、ファイルシステムレベルキャッシュの利点を得ることを選択する場合があります。使うキャッシュアドバイザリはアプリケーションが選択します。

ファイルにキャッシュアドバイザリを設定するには、まずファイルを開きます。キャッシュアドバイザリが要求されると、アドバイザリがメモリに記録されます。つまり、再ブートまたは再マウントすると、キャッシュアドバイザリは維持されないことになります。一部のアドバイザリは、ファイル記述子単位ではなく、ファイル単位で維持されます。これは、ファイル記述子を通して行なったアドバイザリの設定の影響が、同じファイルにアクセスする他のプロセスにも及ぶことを意味します。さらに、競合するアドバイザリは同じファイルへのアクセスに無効であることも意味します。2 つのアプリケーションが異なるキャッシュアドバイザリを設定すると、ファイルに対して最後に設定されたアドバイザリが両方のアプリケーションで使われます。VxFS ではアドバイザリの調整や、優先順位の設定は行われません。

最後のファイルを閉じた後もメモリから消去されないアドバイザリもあります。ファイルへのアクセスの管理に使われるファイルシステムメタデータがメモリに残っている限り、アドバイザリの記録もメモリに残ります。ファイルのファイルシステムメタデータがメモリからいつ削除されるかを予測することはできません。

すべてのアドバイザリは、VX_SETCACHE ioctl コマンドを使って、open() 呼び出しおよび ioctl() 呼び出しから返されたファイル記述子を使って設定します。

vxfsio（7）のマニュアルページを参照してください。

キャッシュアドバイザリには、次の I/O が含まれます。

- ダイレクト I/O
- 同時 I/O
- 非バッファ I/O
- その他のキャッシュアドバイザリ

ダイレクト I/O

ダイレクト I/O とは、ファイルアクセスのための非バッファ形式の I/O です。VX_DIRECT キャッシュアドバイザリを設定している場合、ユーザーは、ディスクとユーザーバッファ間でデータを直接転送し、読み書きするように要求していることになります。この機能により、データはカーネルにバッファリングされず、カーネルバッファとユーザーバッファ間のデータコピーが省略されるため、I/O に関連付けられている CPU オーバーヘッドを削減できます。また、バッファキャッシュの領域が使われないため、その分、アプリケーションキャッシュなど他の用途に活用できます。アプリケーションによっては、ダイレクト I/O 機能によって処理効率が大幅に向上します。

ダイレクト I/O として実行される I/O 操作は、整列に関する基準を満たす必要があります。通常、ディスクドライバ、ディスクコントローラ、システムメモリ管理ハードウェアおよびソフトウェアにより整列制約が決まります。ファイルオフセットは、セクタの境界（DEV_BSIZE）に揃える必要があります。すべてのユーザーバッファは、長い境界またはセクタの境界に揃える必要があります。ファイルオフセットがセクタの境界に揃えられていないと、VxFS は直接読み取りまたは書き込みではなく、通常の読み取りまたは書き込みを実行します。

要求がダイレクト I/O の整列制約を満たさない場合は、データ同期 I/O が実行されます。メモリマップ I/O を使ってファイルがアクセスされている場合は、ダイレクト I/O の代わりにデータ同期 I/O が実行されます。

ダイレクト I/O は、同期 I/O と同様のデータ整合性を保持するため、現在、同期 I/O を使っている多くのアプリケーションで使うことができます。ダイレクト I/O 要求で領域割り当てやファイルの拡張を実行しない限り、i ノードメタデータへの書き込みがただちに実行されることはありません。

ダイレクト I/O の CPU コストは、RAW ディスク転送とほぼ同じです。大容量ファイルへの順次 I/O を、大容量転送サイズを使ったダイレクト I/O に置き換えると、バッファ付き I/O と同等の速度でデータが転送できるため、CPU オーバーヘッドを削減できます。

ファイルの拡張中や、領域の割り当て中には、ダイレクト I/O は i ノード更新の書き込みを、アプリケーションに制御を戻す前に実行する必要があります。このため、ダイレクト I/O の一部の処理効率の優位性が低下します。

ダイレクト I/O アドバイザリは、ファイル記述子単位で保持されます。

同時 I/O

同時 I/O（VX_CONCURRENT）はファイルアクセスの I/O 形式の 1 つです。この I/O 形式によって、複数のプロセスが他の read() または write() の処理を妨げることなく、同じファイルへ読み書きできます。POSIX セマンティクスでは、ファイルに対する read() および write() の処理を、他の read() および write() の処理と一緒にシリアル化する必要があります。POSIX セマンティクスでは、読み取りは、書き込みが行われる前または後のどちらかにデータを読み取ります。VX_CONCURRENT アドバイザリをファイルに設定すると、キャラクタデバイスと同様に読み取りおよび書き込みがシリアル化されることはありません。このアドバイザリは通常、データへの高速のアクセスを必要とし、同一のファイルに対して、重複する書き込みを実行しないアプリケーションで使います。例として、データベースアプリケーションがあります。このようなアプリケーションでは、アプリケーションレベルで独自のロックを実行し、ファイルの同一領域への重複する書き込みを防止しています。

VX_CONCURRENT アドバイザリを使っている場合、同じファイルへの書き込み操作を調整し、書き込みの重複を防止することはアプリケーションまたはスレッドの役割です。同一ファイルへ重複して書き込みを行なった場合、結果は予測できません。アプリケーションにとっては、同一ファイルの同一領域への同時書き込み操作を避けることが最善です。

VX_CONCURRENT アドバイザリをファイルに設定する場合、VxFS はファイルへの読み取りおよび書き込みに、ダイレクト I/O を実行します。したがって、同時 I/O にもダイレクト I/O と同じアラインメント必要条件があります。

77 ページの「ダイレクト I/O」を参照してください。

同時 I/O が有効な場合、読み取りおよび書き込みは次のように実行されます。

- write() システムコールは、読み書きについて、排他ロックではなく共有ロックを取得します。
- write() システムコールは、ユーザーデータをコピーし、システムページキャッシュのページに書き込む代わりに、ディスクに対してダイレクト I/O を実行します。
- read() システムコールは、データをシステムページキャッシュ内のページに読み取ってページからユーザーバッファにコピーする代わりに、読み書きの共有ロックを取得し、ディスクからダイレクト I/O を実行します。

メモ : read() および write() のシステムコールは原子的ではありません。このアプリケーションでは、2 つのスレッドが同時にファイルの同じ領域に書き込まないようにする必要があります。

同時 I/O（CIO）は、VX_CONCURRENT アドバイザリフラグを設定した VX_SETCACHE ioctl コマンドを使って、ファイル記述子と ioctl() 操作によって設定できます。同時 I/O を使うのは、このファイル記述子から実行された read() および write() 操作に限られます。他のファイル記述子から実行された read() および write() 操作は、POSIX セマンティクスに従います。VX_CONCURRENT アドバイザリは、ファイルの VX_SETCACHE ioctl 記述子によって設定できます。

CIO は VxFS のライセンス機能です。

Quick I/O、ODM、CIO、VX_CONCURRENT アドバイザリの相互排他性

VX_CONCURRENT アドバイザリを、Quick I/O または ODM によってアクティブに開いているファイルに設定することはできません。VX_CONCURRENT アドバイザリが設定されたファイルを、Quick I/O または ODM によって同時に開くことはできません。Quick I/O と ODM のアクセスは、-o cio マウントオプションを使ってマウントされたファイルシステムのファイルに対しては使えません。

非バッファ I/O

VX_UNBUFFERED キャッシュアドバイザリの I/O 動作は、ダイレクト I/O と同じアラインメント制約が設定されている VX_DIRECT キャッシュアドバイザリと同じです。ただし、非バッファ I/O の場合、ファイルの拡張や領域の割り当ての実行中のファイル拡張に対するディスク上のメタデータの更新は、制御がアプリケーションに戻される前に同時に実行されることはありません。
VX_UNBUFFERED キャッシュアドバイザリはファイル記述子単位で保持されます。

その他のキャッシュアドバイザリ

VX_SEQ キャッシュアドバイザリは、ファイルがシーケンシャルにアクセスされることを示すファイル単位のアドバイザリです。ファイル記述子を介して、ファイルにこのアドバイザリを設定するプロセスは、現在同じファイルにアクセスしている他のプロセスのアクセスパターンに影響を与えます。VX_SEQ アドバイザリを使ったファイルの読み取りでは、最大量の先読みが実行されます。VX_SEQ アドバイザリを使ったファイルの書き込みでは、シーケンシャル書き込みアクセスが前提とされ、書き込み操作によって変更されたページはただちにフラッシュされません。代わりに、変更されたページはシステムページキャッシュに留まり、それらのページは現在の書き込み時点から一定の間隔を空けてフラッシュされます（フラッシュビハインド）。

VX_RANDOM キャッシュアドバイザリは、ファイルがランダムにアクセスされることを示すファイル単位のアドバイザリです。ファイル記述子を介して、ファイルにこのアドバイザリを設定するプロセスは、現在同じファイルにアクセスしている他のプロセスのアクセスパターンに影響を与えます。このアドバイザリは、先述のように、ファイルの読み取り操作で先読みを無効にし、ファイルのフラッシュビハインドを無効にします。フラッシュビハインドを無効にした結果、システムページキャッシュ内の最近の書き込み操作から変更されたページは、システムページャがスケジュールされ、実行されて、ダーティページをフラッシュするまで、ディスクにフラッシュされません。システムページャがスケジュールされる割合は、空きメモリの可用性や競合状況に基づきます。

メモ : VX_SEQ および VX_RANDOM は相互に非排他的なアドバイザリです。

エクステント

通常、ディスク領域には 512 バイトまたは 1024 バイト（*DEV_BSIZE*）のセクタが割り当てられ、このセクタから論理ブロックが構成されます。VxFS では、1024、2048、4096、8192 バイトの論理ブロックサイズがサポートされています。デフォルトのブロックサイズは、2 TB までのサイズのファイルシステムの場合は 1K、その他のファイルシステムサイズの場合は 8K です。ユーザーは mkfs コマンドを使ってファイルシステムを作成する際に、任意のブロックを選択できます。VxFS では、エクステントと呼ばれる 1 つ以上の隣接するブロックから構成されるグループ単位で、ファイルにディスク領域が割り当てられます。エクステントは 1 つ以上の連続する論理ブロックの集まりです。エクステントを使った場合、格納領域に連続するブロックが割り当てられると、複数のブロック単位でディスク I/O が可能になります。順次 I/O では、複数ブロックオペレーションは、ブロックを 1 つずつ割り当てる方法より大幅に高速です。

VxFS は、ファイルに対するエクステントの割り当てに、アグレッシブな割り当てポリシーを使います。エクステントによって、呼び出し元のアプリケーションは、領域を事前に割り当てたり、連続した領域を要求したりすることができます。この結果、I/O 処理効率が向上し、割り当てを実行するためのファイルシステムオーバーヘッドが減少します。追加書き込み操作の場合、ポリシーによって、以前に割り当てたエクステントを、書き込み操作のサイズ以上に拡張しようと試みます。連続する追加書き込み操作が検出されると、より大きな割り当てが試みられます。最終エクステントを拡張して書き込み全体を収めることができない場合は、連続していないエクステントが新しく割り当てられます。このポリシーは、余分な割り当てを残しておきますが、その割り当てはファイルを最後に閉じたとき、または一定の時間ファイルに書き込まれない場合に削除されます。それでもファイルシステムは、特にサイズが小さい場合に、連続していない大量のエクステントによって断片化する可能性があります。

エクステント属性

VxFS では、1 つ以上のエクステントから構成されるグループ単位で、ファイルにディスク領域が割り当てられます。一般に、VxFS の内部割り当てポリシーは、I/O 処理効率を最適化するようにエクステントを割り当てることと、断片化を削減することの 2 つの目標を達成しようとします。VxFS の割り当てポリシーは、領域を大きく割り当てることと、ファイルシステム内でデータに最も適した空き領域を割り当てることにより、断片化を最小限に抑えることで、これらの 2 つの目標の両方を達成しようとします。こうしたエクステントに基づく割り当てポリシーには、ブロック単位の割り当てポリシーより利点があります。エクステントに基づく割り当てポリシーでは、間接ブロックの割り当てがかなり少なくなるため、間接参照に必要なディスクアクセスの多くを省略できます。

VxFS では、`setext`（1）および `getext`（1）の 2 つの管理ツールと API によって、任意のファイルに対するエクステント割り当てをより細かく制御できます。アプリケーションがファイルに適用するポリシーは、エクステント属性と呼ばれます。VxFS には、アプリケーションからファイルに関連するエクステント属性の設定や表示を実行できるほか、領域の事前割り当てを実行できる API があります。

属性の詳細

ファイルに関連付ける基本的なエクステント属性には、「領域の予約」と「固定エクステントサイズ」の 2 つがあります。ファイルが使うエクステント領域を予約するか、または固定エクステントサイズを設定して、ファイルシステムのデフォルトの割り当てポリシーを無効にすることによって、領域の事前割り当てを行うことができます。これらの属性の方針を決める他のポリシーは、割り当ての処理の中で決定します。たとえば、次のような属性を設定できます。

- ファイル用に予約された領域が連続すること。
- 現在予約された領域を超えるファイルには、割り当てを実行しないこと。
- ファイルを閉じた時点で未使用の予約領域を解放すること。
- 領域は割り当てるが、領域の予約は実行しないこと。
- 割り当てられた領域を即座に反映して、ファイルのサイズを変更すること。

エクステント属性には、永続的なものと一時的なものがあります。永続的な属性は、そのファイルに関する情報としてディスク上に保存されます。一時的な属性は、ファイルを閉じた時点、またはシステムを再ブートした時点で失われます。永続的な属性はファイルのアクセス権限と類似しており、ファイルの i ノードに書き込まれます。ファイルのコピー、移動またはアーカイブを実行すると、ソースファイルの永続的な属性のみが新しいファイルに保持されます。

領域予約 : ファイルへの事前領域割り当て

領域の予約を実行すると、ファイルシステムの領域不足に起因するアプリケーションの障害を確実に回避できます。アプリケーションは、ジョブの実行を開始する前に、全ファイルに対応する領域を事前に割り当てることができます。領域を事前に割り当てることによって、処理効率が向上するようにファイルが適切に割り当てられ、領域の割り当てが必要となった場合でも、ファイルへのアクセスが低下しません。遅延の保証が要求されるアプリケーションでは、このようなリソースの割り当ては重要です。超大容量ファイルを処理する場合は、領域の予約を使うと、間接エクステントの使用を回避できます。また、領域の予約により、ファイルが連続した大容量エクステントで構成されるため、高性能処理効率と断片化の低減を実現できます。

VxFS は、データがファイルに書き込まれるときではなく、書き込みが要求されたときに、領域をファイルに事前に割り当てることができる API を提供しています。事前割り当てまたは予約によって、データがファイルに書き込まれる前に、ファイルに必要な領域がファイルに対応付けられるため、ファイルシステムの予期しない領域不足の状況を回避できます。領域はいつでもファイル用に予約でき、ファイルに予約された領域は、ファイルシステムの他のファイルに割り当てられません。API には、アプリケーションから予約された領域を含めるように、ファイルのサイズを変更できるオプションがあります。

領域予約では、ファイルに割り当てられるブロックの初期化を行いません。そのため、この機能は、ファイルのサイズが予約要求によって変更されない限り、適切な権限で実行しているアプリケーションに限定されます。ファイルに新しく割り当てられたブロックにあるデータは、以前に別のファイルのものであった可能性があります。

領域予約はディスクに保存されたファイルの永続的な属性です。ファイルにこの属性を設定すると、ファイルが切り捨てられたときにも属性が解放されません。予約領域を解放するには、同じ API から予約の設定を解除するか、ファイルを削除する必要があります。領域予約を指定した時にファイルサイズが予約領域より小さい場合、現在のファイルサイズから予約した容量までの領域がファイルに割り当てられます。ファイルを切り捨てても、予約した容量に満たない領域は解放されません。

固定エクステントサイズ

VxFS はファイルへの領域の割り当てに、書き込み要求の I/O サイズとデフォルトの割り当てポリシーを使います。ただし、アプリケーションによっては、デフォルトの割り当てポリシーが最適ではないことがあります。ファイルに固定エクステントサイズを設定すると、そのファイルのデフォルトの割り当てポリシーが上書きされます。アプリケーションでは、ファイルに割り当てられた新しいエクステントがすべて固定サイズになるように、固定エクステントサイズをアプリケーションの I/O サイズに一致するように設定できます。固定エクステントサイズを使うことにより、アプリケーションは割り当て回数を減少させ、ファイルに最適なエクステントサイズを確保することが可能になります。固定エクステントサイズ属性によって、追加書き込み操作で、VxFS は固定エクステントサイズで以前に割り当てられているエクステントを拡張して、エクステントの連続性を維持しようとします。最終エクステントを固定エクステントサイズで拡張できない場合は、連続していないエクステントが新しく割り当てられます。固定エクステントのサイズには、アプリケーションに適切なファイル I/O のサイズを考慮する必要があります。エクステントに基づく割り当てポリシーの利点を生かすことができない小さな固定エクステントサイズは使わないでください。

スパースファイルでも固定エクステントサイズを使えます。VxFS は、通常、システム定義のページサイズの倍数で I/O を実行します。スパースファイルに割り当てる場合、VxFS は割り当てに必要なページ I/O の容量に従って、ページサイズの倍数で領域を割り当てます。アプリケーションがサブページ単位の I/O を常に実行する場合は、ページサイズの倍数で固定エクステントサイズを使うことによって、割り当て回数を削減できます。

アプリケーションでは大きな固定エクステントサイズを使わないでください。大きな固定エクステントを割り当てようとすると、そのサイズのエクステントを取得できないために失敗する可能性があります。エクステントを小さくすると、割り当てにすぐに利用できる領域が多くなります。

カスタムアプリケーションでも、ディスク上のシリンダまたはストライプ境界に揃えるようにエクステントを整列させる場合など、特定の理由から固定エクステントサイズを使う場合があります。

固定エクステントサイズ属性は、ファイルシステムのブロックサイズを単位として指定します。この属性で指定するのは、新しいエクステントに割り当てる連続したファイル システムのブロック数、または既存のエクステントの末尾に追加するか、割り当てる連続したブロックの数です。この属性を持つファイルは、固定サイズエクステントまたは固定サイズエクステントの倍数の大きなエクステントを持ちます。

エクステント属性のアプリケーションプログラミングインターフェース

エクステント属性の現在の API は ioctl() です。アプリケーションは、ファイルを開き、ioctl() の呼び出しから得たファイル記述子を使って、エクステント属性の取得、設定または変更を実行できます。既存のエクステント属性の設定や変更を行うには、VX_SETEXT ioctl を使います。既存のエクステント属性がある場合に、それを取得するには、VX_GETEXT ioctl を使います。アプリケーションからエクステント属性の設定や変更を行うには、vx_ext の構造体に、属性情報をセットしてから、VX_SETEXT iotcl と構造体のアドレス（3 番目の引数）を指定して ioctl() を呼び出します。また、既存のエクステント属性がある場合にそれを取得するには、VX_GETEXT ioctl と先と同じ vx_ext 構造体のアドレス（同じく第 3 引数）を指定して、ioctl() を呼び出します。

```
    struct vx_ext {
        off_t   ext_size;   /* fs ブロックのエクステントサイズ */
        off_t   reserve;    /* fs ブロックの領域予約 */
        int     a_flags;    /* 割り当てフラグ */
}
```

ext_size 引数は、固定エクステントサイズの指定に使います。固定エクステントサイズの値は、ファイルシステムのブロックサイズ単位で指定します。固定エクステントサイズを設定する前に、ファイルシステムのブロックサイズを調べておいてください。固定エクステントサイズが不要な場合は、ゼロを使って、エクステントの割り当てにデフォルトの割り当てポリシーが使われるようにします。VX_SETEXT ioctl の正常な実行後、すぐに固定エクステント割り当てポリシーが有効になります。ただし、すでに間接ブロックを含んでいるファイルは例外です。この場合、固定エクステントポリシーは、ファイルの切り捨てによって現在のすべての間接ブロックが解放されるまで有効になりません。

reserve 引数は、ファイルに事前に割り当てる領域のサイズの指定に使います。このサイズは、ファイルシステムのブロックサイズ単位で指定します。事前に割り当てるサイズを設定する前に、ファイルシステムのブロックサイズを調べておいてください。ファイルがすでに事前に割り当てられている場合、VX_SETEXT ioctl を使って、現在の予約サイズを変更できます。指定した予約サイズが現在の予約よりも大きい場合は、新しく指定した予約サイズに合わせてファイルの割り当てが拡大されます。予約サイズが現在の予約よりも小さい場合は、予約サイズが減らされ、新しく設定した予約サイズまたは現在のファイルサイズまで割り当てが減らされます。ファイルの事前割り当てには、ファイルのサイズが変更されない限り、root 権限が必要であり、要求元のプロセスの ulimit を超えて事前割り当てのサイズを拡大することはできません。

VX_CHGSIZE フラグを参照してください。

ulimit（2）のマニュアルページを参照してください。

割り当てフラグ

VX_SETEXT ioctl で割り当てフラグを使って、割り当てポリシーをより細かく制御できます 割り当てフラグは vx_ext 構造体の *a_flag* 引数に指定して、次のことを指定します。

- アロケーションユニットを整列させるかどうか。
- アロケーションユニットを連続させるかどうか。
- 予約領域を超えてファイルを書き込めるようにするかどうか。
- ファイルを閉じた時点で未使用の予約領域を解放するかどうか。
- 領域の予約をファイルの永続的な属性にするかどうか。
- ファイルの予約領域をどの時点でそのファイルの一部にするか。

領域の予約での割り当てフラグ

VX_TRIM、VX_NOEXTEND、VX_CHGSIZE、VX_NORESERVE、VX_CONTIGUOUS、VX_GROWFILE の各フラグを使うと、領域の予約要求を変更できます。VX_NOEXTEND は変更できない永続的なフラグです。永続的な効果を持つ他のフラグもありますが、VX_GETEXT ioctl システムコールから取得できません。非永続的なフラグは、ファイルシステムキャッシュのファイルに対しても、ファイルがアクセスされなくなり、キャッシュから削除されるまで、アクティブなままです。

予約領域の解放

VX_TRIM フラグは、ファイルが最後に閉じられるときに、予約サイズをファイルサイズに合わせて切り捨てる必要があることを指定します。最後に閉じられるとき、VX_TRIM フラグが消去され、ファイルのサイズを超える未使用の予約領域が解放されます。これは、アプリケーションでファイルに十分な領域を必要とする場合に効果的です。ただし、その結果、ファイルがどれくらいのサイズになるかは不明です。大容量になると予想されるファイルを保持するために十分な領域を予約した場合、ファイルの書き込み後にファイルを閉じると、余分な領域が解放されます。

非永続的な予約

予約を永続的属性にすることが望ましくない場合は、VX_NORESERVE フラグを指定すれば、予約をファイルの永続的属性にすることなく、領域の割り当てを要求できます。このフラグは、一時予約が必要であり、ファイルが閉じられるときにファイルの末尾を超える領域を解放することが必要なアプリケーションで使えます。たとえば、アプリケーションで 1 MB のファイルをコピーする場合、VX_NORESERVE フラグを設定して 1 MB の領域の予約を要求できます。領域は割り当てられますが、ファイルの予約の設定値は、0 のままです。プログラムがなん

らかの理由で異常終了した場合、またはシステムがクラッシュした場合は、ファイルの末尾を超える未使用領域は解放されます。プログラムが正常に終了した場合は、ディスク上に予約領域が記録されないため、領域の解放も発生しません。

予約領域を超えた書き込みの禁止

VX_NOEXTEND フラグを指定すると、現在の予約領域を超えた書き込みが失敗するようになります。現在の予約領域を超えた書き込みには、ファイルへの新しい領域の割り当てが必要になります。ファイルに新しい領域を割り当てるには、予約領域を拡張する必要があります。ulimit コマンドは、ファイルに使われる領域に制限を設けることによって、同様の機能を提供します。

limit（1）のマニュアルページを参照してください。

連続した予約領域

VX_CONTIGUOUS フラグは、ファイルに割り当てられるすべての領域が、単一のエクステント割り当ての必要条件を満たす必要があることを指定します。予約要求に対応できるだけの大きさの 1 つのエクステントが存在しない場合、この要求は失敗します。たとえば、ファイルの作成後に、1 MB の連続した予約領域を要求すると、ファイルサイズは 0 に、予約領域は 1 MB に設定されます。このファイルには 1 MB のエクステントが 1 つ保持されます。さらに 3 MB の連続した領域の予約要求が発行されると、新しい要求は最初の 1 MB がすでに割り当てられていることを検出し、要求を満たすために 2 MB の予約領域を同じエクステント内から割り当てます。同じエクステントが 2 MB の領域を予約領域として使えない場合、要求は失敗します。デフォルトでエクステントは連続する割り当てを実行するように定義されています。VX_CONTIGUOUS は永続的なフラグでないため、VX_CONTIGUOUS フラグによって以前に割り当てられているファイルを復元する場合に、領域が連続して割り当てられなくなることがあります。

ファイルサイズに予約領域を含める

領域を予約する際に、VX_CHGSIZE を指定すると、予約領域を含めるようにファイルのサイズを変更できます。このフラグによって、予約された領域を初期化せずに、予約サイズに合わせてファイルのサイズが拡大されます。このフラグは、過去に他のファイルに含まれていたデータを初期化せずにファイルに含んでしまう可能性があるため、このフラグの使用は、適切な権限を持つユーザーに制限されます。このフラグを使わなければ、拡張書き込み操作が追加領域を必要とするまで、予約領域はファイルサイズに含まれません。即座にファイルサイズを変更するように予約領域を設定すると、数多くの一時ファイルが生成されます。アプリケーションは、この種類の予約により、アプリケーションから書き込み操作に伴う領域割り当てとファイルサイズ更新にかかわるオーバーヘッドをなくすことができるという効果が得られます。

前述のフラグは組み合わせて使うことができます。たとえば、VX_CHGSIZE と VX_NORESERVE を使うと、ファイルサイズは変更されますが、領域の予約は実行されません。ファイルサイズが切り捨てられ、領域が解放されます。VX_NORESERVE フラグを使わない場合は、ファイルサイズに応じた領域の予約がディスクに設定されます。

ファイルの増加分の読み取り

割り当てフラグ（*a.flag*）に VX_GROWFILE が設定されているとき、ファイルサイズは予約を含めるように変更されます。このフラグにより、ファイルの増加部分（現在のファイルサイズと操作が成功した後のサイズとの差）が読み取られます。VX_GROWFILE は永続的な影響を持ちますが、割り当てフラグとしては表示されません。このフラグは、VX_GETEXT ioctl により表示されます。

固定エクステントサイズでの割り当てフラグ

VX_ALIGN フラグを使って、固定エクステントサイズの割り当てフラグを指定できます。このフラグは、予約要求で指定しても無効になります。VX_ALIGN フラグは、将来のエクステントの割り当てがアロケーションユニットの開始位置を基準に固定エクステントサイズ境界で整列するように、アラインメント必要条件を指定します。このフラグを使うと、ディスクストライプ境界や物理ディスク境界にエクステントを整列できます。VX_ALIGN は永続的なフラグであり、VX_GETEXT ioctl システムコールによって取得できます。

エクステント属性 API の使用方法

まず、目的のファイルシステムが VxFS であることを確認してから、statfs() 呼び出しを使って、ファイルシステムブロックサイズを調べます。VxFS のタイプはほとんどのプラットフォームで MNT_VXFS であり、ファイルシステムのブロックサイズは *statfs.f_bsize* で返されます。VxFS エクステント属性 API を使って、エクステント属性情報の設定を行う、または解釈をするには、ブロックサイズを調べておく必要があります。

VX_SETEXT ioctl システムコールを呼び出すたびに、vx_ext 構造体のすべての要素が影響を受けます。

VX_SETEXT の使用手順

1. VX_GETEXT ioctl を呼び出して、現在の設定値があればその値を読み取ります。
2. 変更する現在の値を修正します。
3. VX_SETEXT ioctl を呼び出して、新しい値を設定します。

警告 : 前述の手順は慎重に実行してください。領域の予約を変更するときに、固定エクステントサイズを誤って設定解除してしまう可能性があります。VxFS と VxFS 以外のファイルシステムの間でファイルをコピーする場合は、エクステント属性を保持できません。vx_ext 構造体で返されるファイルの属性値は、ソースファイルシステムと異なるファイルシステムブロックサイズを持つ他の VxFS ファイルシステムに対しては効果が異なることに注意してください。ブロックサイズが異なる 2 つのファイルシステム間で、属性を持つファイルをコピーする場合は、異なるブロックサイズの属性値の変換が必要になる場合があります。

次に、MY_PREFERRED_EXTSIZE 属性の固定エクステントサイズを新しいファイル MY_FILE に設定するコード例の一部を示します。MY_PREFFERED_EXTSIZE はファイルシステムのブロックサイズの倍数であると仮定します。

```
#include <sys/fs/vx_ioctl.h>

struct vx_ext myext;

fd = open(MY_FILE, O_CREAT, 0644);

myext.ext_size = MY_PREFERRED_EXTSIZE;
myext.reserve = 0;
myext.flags = 0;

error = ioctl(fd, VX_SETEXT, &myext);
```

次に、MY_FILESIZE_IN_BYTES バイトの領域を新しいファイル MY_FILE に事前に割り当てるコード例の一部を示します。目的のファイルシステムのブロックサイズは THIS_FS_BLOCKSIZE であると仮定します。

```
#include <sys/fs/vx_ioctl.h>

struct vx_ext myext;

fd = open(MY_FILE, O_CREAT, 0644);

myext.ext_size =0;
myext.reserve = (MY_FILESIZE_IN_BYTES + THIS_FS_BLOCKSIZE) /
                                          THIS_FS_BLOCKSIZE;
myext.flags = VX_CHGSIZE;
error = ioctl(fd, VX_SETEXT, &myext);
```

索引

R

S

U

V

W

あ

え

き

こ