Linux 用ログ構造化ファイルシステム nilfs の設計と実装

天海良治[†] 一二三尚[†] 小西隆介[†]
佐藤孝治[†] 木原誠司[†] 盛合敏[†]

Linux のローカルファイルシステムとしてログ構造化ファイルシステム nilfs を開発した．nilfs はログ構造化方式を採用したことで，ディスクブロックの上書きがなくなり，障害発生時の被害を最小限に抑えることができる．また，高速なスナップショット作成とファイルシステムチェックが可能となった．ファイルのディスクブロックの管理と inode 管理には B-Tree を採用し，大容量ディスクの使用，大規模ファイルの作成，inode 個数の事実上の上限廃止を実現している．本稿では nilfs の内部構造と実装方法について述べる．

The Design and Implementation of "nilfs", a Log-Structured File System for Linux

Yoshiji Amagai,[†] Hisashi Hifumi,[†] Ryusuke Konishi,[†]
Koji Sato,[†] Seiji Kihara[†] and Satoshi Moriai[†]

We developed a Log-Structured local file system "nilfs" for Linux operating system. The Log-Structured file system prevents data write operations from overwriting the disk blocks. So, it minimizes a damage of file data and file system consistency on hardware failure. The Log-Structured system is able to create a snapshot of the file system immediately and to check the file system speedy. The "nilfs" implements file and inode block management by B-Tree structure. The B-Tree structure enables the "nilfs" to create a huge file, to store a large number of files. This paper presents the design and implementation of the "nilfs".

1. はじめに

ファイルシステムにはまず次の機能が求められる．

データが確実に書けること，書いたデータが正しいことが確認できること（信頼性）．ディスク装置の故障や，電源切断といったシステムの障害に対処できること（可用性，耐故障性）．障害があったとき書いたデータが高速に，かつできるだけ多く復旧可能で，かつ復旧したデータに矛盾がないこと（復旧性）．障害発生時の書込み操作は，復旧した時には，完了しているかまたは書込み操作開始前のどちらかの状態となること．ファイルシステム全体が壊れないこと（堅牢性）．

この確実なデータ保持という基本的機能を備えた上で，性能や運用の容易さといった機能が活用される．

ファイルデータの書出しには，ブロックの確保，ユーザデータの書出し，間接ブロックの初期化や書換え，inode の書換えといった一連の入出力操作の順序が重要である．順序を守っていれば，ブロックの書換え中に障害があったとき，間接ブロックや inode が間違ったブロックを指してファイルシステムの首尾一貫性を失うようなことにはならない．また，すでにあるブロックの上書きは，書込み中のシステム障害によるデータ破壊の危険性を常に抱えている．

Linux では伝統的な ext2，ジャーナリング機能をもった ext3[1)]，XFS[2)]，ReiserFS[3)]，JFS[4)] など，多くのローカルファイルシステムが使用できる．特にジャーナリング機能は，ファイルシステムの首尾一貫性の確認を容易にし，ファイルシステムチェックの高速化，信頼性，堅牢性の向上に貢献している．

だが，例えば ext3 では，ジャーナリングログを利用者に隠されている通常のファイルに書出している．通常ファイルである以上，そのログを置くファイルシステムの信頼性がログの信頼性を決定する．ファイルシステムより下位のレイヤでディスク書込みのエレベータアルゴリズムが適用されていれば，ディスクヘッドのシーク時間を短縮するために，ファイルシステムの

† NTT サイバースペース研究所 OSS コンピューティングプロジェクト
NTT Cyber Space Laboratories Open Source Software Computing Project

首尾一貫性を保つ正しい順序を無視した書出しが実行されうる．万一，壊れたジャーナルファイルを元に復旧処理をしたら，ファイルシステムはさらに破壊されることになる．

われわれは，Linux のファイルシステムのさらなる信頼性，堅牢性の確保のため，ログ構造化ファイルシステム (Log-Structured File System, LFS) に注目した．LFS はディスクブロックの書込みをすべてデータの追記とする．すでに存在するブロックを上書きしないことで，上書きに起因する問題を解決する．また，ブロック書出しをまとめて実行することで，書出しの順序を正しく維持した上で高速な書出しができる．現在，Linux 2.6 のカーネルモジュールとして，ログ構造化方式のローカルファイルシステム nilfs (New Implementation of a Log-Structured File System) を開発中である．

本稿の構成は以下のとおりである．まず，2 章で開発の目標を述べる．3 章で従来のファイルシステムと LFS の原理について簡単に紹介する．4 章，5 章でわれわれの開発している LFS のディスクレイアウト，処理の流れについて述べる．6 章で性能評価を示す．7 章でまとめと今後の課題を述べる．

2. 開 発 目 標

Linux で新しくファイルシステムを開発するにあたって，以下の目標を設定した．

(1) Linux のセマンティクスを守ること

(2) カーネルコードへの変更は避けること
導入を容易とし，Linux カーネルの頻繁なバージョンアップへの追従性を確保する．

(3) 高信頼であること．壊れにくく，ファイルシステムでデータベース並みの信頼性をもつこと

(4) 高可用であり，停止時間が短いこと．障害発生後の復旧時間が短いこと

(5) 運用性に優れること．例えば，構成の変更が容易にでき，スナップショットが短時間で取れること

(6) 性能・機能が現存の ext3 ファイルシステムらと同等かそれ以上であること

(7) スケーラビリティのあること
大容量ディスクに対応するだけではなく，ディスクの増設に対して柔軟に対応できることが必要．

(8) 将来は，分散やクラスタなどの大規模ネットワークでのストレージに対応すること

(9) さまざまな言語に対応できること
主にファイル名の扱いを念頭においている．

(10) ユーザフレンドリであること
システム障害の原因の多くは人間の操作誤りであるので，誤りにくい，誤っても復旧が容易なシステムを提供する．

(11) 新機能の追加，拡張が容易であること

この開発目標には将来の課題も含まれている．このうち，われわれはまず高い信頼性と堅牢性の確保を目標としている．

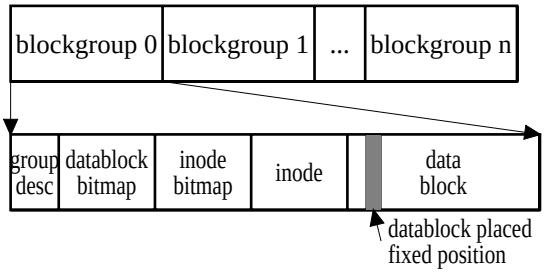

図 1 従来のファイルシステムのブロック配置

3. ログ構造化ファイルシステムについて

ファイルシステムの破壊はディスク上の既存ブロックの上書きが大きな原因である．LFS は上書き回避に根本的な解決を与える方式である．

3.1 従来の方式のブロック配置

伝統的な UNIX ファイルシステムでは，ディスクの使用方法は図 1 のようになっている．

ファイル管理のための inode，ファイルのデータブロック，大きなファイルを管理するための間接ブロック，未使用ブロックを管理するビットマップは一度割当てたらその位置が動くことはない．ブロック内のデータを書換えるときにはブロックの上書きとなる．

ファイルは truncate システムコールで大きさを縮めることができる．このときには割当てられていたブロックは解放され，次に再びデータを追加したときには，未使用ブロックから新たなブロックを割当てる．ファイルブロック番号（ファイルの先頭からのバイト位置をブロックのバイトサイズで除したもの）とディスクブロック番号（ディスクパーティションの先頭からのブロック番号）の対応は truncate されない限り不変である．

ext2/3 のブロックグループや，BSD FFS[5] のシリンダーグループなど，ディスク全体をいくつかのグループに分けて使用することが多いが，これは 1 つのファイルのブロックと関係する管理用データを物理的にディスクの近接位置に配置できるようにしてアクセスの高速化を図ることが大きな目的である．

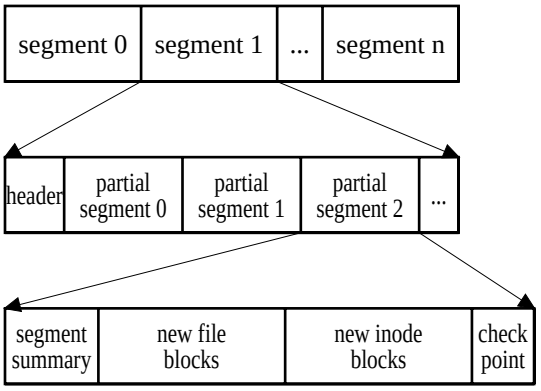

図 2 LFS のブロック配置

3.2 LFS のブロック配置

これに対し，LFS では図 2 のようにデータ書出し時に常に新しいディスク位置を割当てる．ファイルの最後にデータを追加していくときは従来方式でも新しいブロックを割当てるが，LFS ではファイルの途中のデータを書換えるときも新たなブロックを割当てる．従来方式で固定位置にあったディスク上の inode も，内容に変更があれば新しいブロックを割当てて書出す．ログをファイルに追記していくように，ブロックをディスク領域に追加して書出していくので，ログ構造化方式と呼ばれる．

LFS の最大の利点は，書出し時に常に新しいブロックを割当てることで，書出し中に電源の切断などの障害があったときも，それまであったブロックを壊すことがないことである．カーネルメモリ上のファイルバッファ（ページキャッシュ）をディスクに書出すときに新たなディスクアドレスを決定する．ここで，ディスクアドレスを，ファイルデータのブロック，ファイルデータの間接ブロック，inode ブロック，inode ブロックを保持するための間接ブロック，inode 全体の管理のルートとなるブロック，の順に連続して割当て，この順で書出せば，ファイル変更中のどの段階でもファイルシステムが中途半端な状態になることはない．ファイルシステムの首尾一貫性を保つための順序と，ディスクに書出す順序が同じになるので，正しい順序でも高速に書出すことができる．障害発生時にもディスクの書出しが連続してまとまっているので，ファイルシステムの整合性検査 (fsck) は検査する位置を特定することができる．

毎回新しいディスクブロックに書出すので，ファイルブロック番号に対応するディスクブロック番号は書出しごとに変わっていく．また，inode ブロックの場所も変わっていくので，inode 番号からその inode を納めたブロックへの対応付けの機能が必要となる．この点は，対応が固定であった従来方式と異なる．

さらに，ディスクの容量は有限なので未使用領域がなくなる前に，論理的に上書きをした領域，消去したファイルが占めていた領域などを回収し，未使用領域に戻すガーベージコレクション (Garbage Collection, GC) が必要である．また，ファイルの一部を上書きしていくと，ディスク上のデータブロックの位置が離れ，ファイルが断片化する．これによりファイルの連続順読込みの性能が低下するので，必要に応じてデータブロックを再配置する必要もある．GC と再配置は LFS で新たに必要となる機能で性能にも影響する．だが，逆に GC のタイミングで空き領域をまとめたり，再配置するといった従来の固定位置配置ではできなかったブロック操作が可能となる．

LFS では，GC を効率的に実現するためにディスクをブロックより大きな単位で管理する手法がとられる．例えば 4M バイトの固定サイズでディスクを区切ってこれを単位とする．この管理単位をセグメントと呼ぶ．従来のファイルシステムのブロックグループやシリンダーグループと目的は異なる．

3.3 関 連 研 究

実用的なログ構造化ファイルシステムのひとつが，ネットワークオペレーティングシステム Sprite[6] で 1990 年頃使用されてた Sprite-LFS[7] である．inode もログとして書出すが，inode の位置を管理するので inode map と呼ばれる構造体を作る．inode map そのものもログとして書出す．不可分なディレクトリ操作については別にジャーナルログを使用している．

その後，BSD 系 OS で Sprite-LFS を参考に実装されたのが BSD-LFS[8] である．BSD-LFS では inode を ifile と呼ぶ通常のファイルに置くことで，inode 数の上限をなくす，inode 操作のための特別なコードを減らす，ユーザ空間で実行される GC プロセスがこのファイルを特別な機構なしに参照できるようにする，といった機能を実現している．ディレクトリの不可分操作も書出すログデータそのものに不可分であることを記録するようにしている．

計算機のローカルファイルシステムとして実現された例は比較的少ない．現状では BSD 系のオープンソース OS である NetBSD[9] で利用できる程度である．Linux では kernel2.2 の ext2 ファイルシステムのブロック配置部分にログ構造を組み込んだ LinlogFS[10] が開発されたが，実験段階に留まっており，現在のカーネルバージョンにも追従していない．

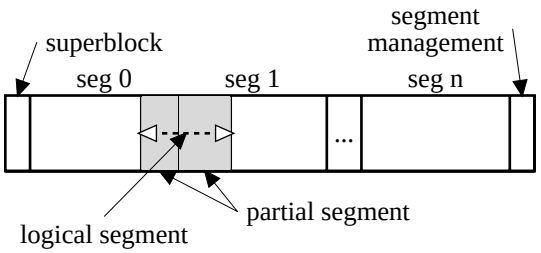

図 3 nilfs のディスクレイアウト

4. ディスクレイアウト

nilfs の第 1 版では，信頼性と堅牢性の確保を第 1 の目標とした．性能のチューニングは未着手である．ただし，将来にわたって nilfs が使用できるように，ファイルサイズと inode 番号を 64 ビット化し，ファイルブロック管理に B-Tree[11)12)] を使ったルート inode 方式を採用した．inode ブロックの管理もメモリ上スーパーブロック構造体にルートを置く B-Tree で管理することとした．

この章では，まずディスク全体のレイアウトを示す．次にファイルブロックの管理構造を説明し，それらのディスク上の配置を説明する．

4.1 ディスクレイアウト

図 3 に nilfs のディスクレイアウトを示す．いくつかの部分に分けて使用する．

スーパーブロック ファイルシステム自体のパラメータ，および，もっとも最近書出した部分セグメントの位置が格納されている．これが有効なデータのルートとなる情報である．スーパーブロックの情報は重要なので，ディスクの複数の固定位置に複製を置く．

セグメント GC によるディスクブロック管理の単位で基本的には固定長．ただし，スーパーブロックを含んでいる先頭のセグメントやスーパーブロックの複製やセグメント管理ブロックをおいているセグメントはそのぶんだけ小さい．

部分セグメント 書出しの単位．sync システムコールや pdflush デーモンなどでキャッシュ上のデータをディスクに書出す．4.3 節に詳細を示す．

論理セグメント ディレクトリ操作などファイルの生成，削除，名前変更の順序が定められている操作の結果（トランザクション）として書出される一連のブロックが，セグメントの境界を越えることがある．トランザクションをなすひと続きの部分セグメントを論理セグメントと呼んでいる．

例えば，図 3 の影の部分が，セグメント境界を越えて 2 つに分かれた部分セグメントである．この 2 つの部分セグメントに不可分な操作が含まれているので，障害復旧時には論理セグメントとして扱うことを示している．

セグメント管理ブロック セグメントの使用状況をまとめたブロック．GC が利用する．ディスク中に分散して複数ある．このブロックとスーパーブロック，スーパーブロックの複製だけがディスクブロックを上書きで書換える．書換え中の障害発生に備え，同じ情報を複数箇所に書出している．利用時には，チェックサムと情報同士のビット比較によって正しく書かれたことを確認できる．

4.2 ブロック管理

ファイルシステムに保持すべき基本的データは，まずユーザのファイルのブロックである．このファイルの個々の管理用データである inode 情報もディスクに格納する．これらと共に，名前を階層的に管理するためのディレクトリがある．これらに共通するのは，マッピングを保持する点である．ファイルでは，ファイルブロック番号からディスクブロック番号へのマッピングであるし，inode では，inode 番号からその inode が格納されているディスクブロック番号へのマッピングが必要である．ディレクトリは名前から inode 番号へのマッピングを保持する．

nilfs では，ファイルのマッピングと inode のマッピングに，ディスク上で平衡木を実現する B-Tree を採用した．これにより，共通の B-Tree 操作ルーチンで，大きなファイルの効率的な管理と，柔軟で効率的な inode 管理が可能となった．B-Tree には変種として，順アクセスの高速化を図った B^+-Tree[12)]，記憶空間の利用率の向上を図った B*-Tree[12)] などがある．今回はもっとも基本的な B-Tree を使用している．ファイルのブロック管理には順アクセスに対応した B^+-Tree が適しているが，nilfs では採用できなかった．B^+-Tree では木の葉にあたるノード同士をリンクで連結している．LFS はブロックが書換えられるとそれは新しい位置に移動するので，その位置を格納していたノードも書換えられ，新しい位置に移動する．葉がリンクされていると，この書換えがすべての葉に伝搬することになり，ファイルへのデータ追加は木の全体のコピーの作成を招いてしまう．この動作はファイルシステムとして許容できない．ファイルの順アクセスの高速化は，先読み制御などの手法で行なうことになる．

B-Tree の構築に必要となるブロックが B-Tree 中間ノードである．中間ノードには 64 ビットのキー，64 ビットのポインタを 1 組として，ブロックにまとめて格納している．ファイル管理用 B-Tree のキーは

ファイルブロック番号である．inode 管理用 B-Tree のキーは inode 番号そのものを使う．ポインタ部にはディスクブロック番号が格納される．ファイルブロック B-Tree のルートはそれを管理する inode に格納する．個々のファイルごとに B-Tree が存在する．inode B-Tree のルートはスーパーブロックに格納する．inode の B-Tree はファイルシステムに 1 つだけである．

現在は，サイズの小さなファイルについても B-Tree を構成して管理している．例えば，小さなファイル用に inode にポインタ部だけを置く，といったことは考えられるが将来の課題である．

ファイルブロック，ファイル管理 B-Tree 中間ノードのブロック，inode ブロック，inode 管理 B-Tree 中間ノードのブロックはすべてログの形でディスクに追記される．ブロックは，変更されたときには新しいディスクブロック番号をもつことになる．これには B-Tree のポインタ部を書換えることで対応する．キー部分には変更がないので，木の形が変わることはない．

新規のファイルなどはまずメモリ上のページキャッシュにのみ存在し，ディスクには通常は遅れて書かれる．書かれる前にも他のプロセスがオープンして読込むなどの操作が可能でなければならない．このため B-Tree の構造はディスク上と共に，メモリ上にも存在する．メモリ上の B-Tree 中間ノードのポインタ部には，ディスクブロック番号か，またはその下位ノードのブロックを読込んだページキャッシュのアドレス（ページキャッシュを管理するバッファヘッド構造体へのポインタ）が格納されている．木の探索やエントリの挿入で木を辿る場合，ポインタ部がディスクブロックであればそのブロックを読込み，ポインタ部をバッファヘッドのアドレスに書換える☆．このように木はメモリ上にすべて展開されているわけではない．

名前から inode 番号へのマッピングを保持しているディレクトリも B-Tree を採用する計画である．だが，可変長の文字列をキーとする B-Tree は実装が複雑で，現在は実現していない．今の nilfs のディレクトリは ext2 のディレクトリ実装をほぼそのまま移植したものである．そのため，大量のファイルがあるディレクトリ操作の性能はよくない．

4.3 部分セグメントのレイアウト

部分セグメントの内容をあげる．図 4 のように 3 つの部分からなる．

セグメントサマリ　部分セグメントの管理用のデータ．主な内容は，データ領域のチェックサム，セグメントサマリ自体のチェックサム，セグメントの長さ，データ領域の種類ごとのブロック数，この部分セグメントの生成時刻などである．セグメントサマリは通常のデータアクセスでは参照されない．GC と障害復旧が必要とする．

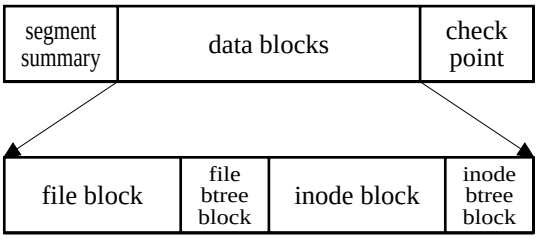

図 4　部分セグメント

データ領域　ファイルデータブロック，ファイルブロック管理 B-Tree の中間ブロック，inode を納めたブロック，inode 管理 B-Tree の中間ブロックの順に格納されている．

チェックポイント　チェックポイントは部分セグメントの最後尾に置かれる．部分セグメントの終りを示すとともに，これを書くことで，それまでの書出しが成功していることを示す．

ここに納められるもっとも重要な情報は，チェックポイント作成時のディスク上 inode の管理 B-Tree のルートブロック番号である．部分セグメントの最後に inode のルートの情報を書出すことで，ファイルシステムの状態が更新される．チェックポイント自体のチェックサムもチェックポイントに書かれていて，チェックポイント情報が正しく書かれていることが確認できる．

4.4 信頼性の確保

nilfs の目標である信頼性の確保のため，次の機能を実現した．

チェックサム　サマリ，チェックポイントなどファイルシステムの管理上重要なデータにはチェックサムを付加して書込みが正しく完了しているかが確認できるようにした．またユーザデータ部分についても部分セグメントごとにチェックサムをおき，障害復旧時にデータの正当性を確認できるようにしている．ただし，ブロックごとにチェックサムを置くことはしていない．

書出し順序の保持　メモリ上で変更になったデータをディスクに書出すとき，まず必要なデータをすべて集めて部分セグメントを作る．このときにデータを先に，管理情報を後にといった書出し順序を決めることができる．カーネルの下位の入出力部や，デバ

☆ 元のディスクブロック番号はバッファキャッシュ側に残っている

イスドライバで要求した書出しの順序を変えないように指示している．書出しはディスクのブロック順に実行されるので，信頼性確保の順序を守っても書出し速度の低下はない．

最小限の上書き操作 nilfs でブロックの上書きが発生するのは，スーパーブロックの更新，スーパーブロックの複製の更新，セグメント管理ブロックの更新だけである．これらはファイルシステム全体の管理のための情報で，固定位置に置かれる．ユーザデータや inode，それらを管理する B-Tree のブロックは決して上書きされない．上書きするデータは複数の位置に同じデータを書くことで信頼性を確保している．

この 3 つの機能でファイルシステムの首尾一貫性の確保と高速な障害復旧機能を確保する．

5. 処理の流れ

5.1 カーネル 2.6 のバッファ管理

通常のファイルの入出力はすべてページキャッシュを経由する．読込み時にはファイルの対象のブロックがすでにキャッシュに読込まれているかどうかを判定する．書込み時も，すでにあるブロックの一部を書換えるのであればまずそのブロックをキャッシュに読込むし，新規に書込むときにはまずページを用意する．

ディスクのブロックがすでにページキャッシュに読込まれているかどうかの判定のため，Linux2.6 では，読込んだページを木構造 (radix-tree) に保持している．木のルートはメモリ内 inode 構造体にあり，ファイルの論理的なブロック番号から計算できる値をキーとして読込んだページを探索することができる．ファイルのデータを格納したブロックはこの方法ですでにキャッシュに読込んだかどうかが判定できるが，ファイルブロックを管理するための間接ブロックやビットマップを保持するためのブロックには，論理的なファイルブロック番号はない．そこで，Linux2.6 ではメモリ上 block_device 構造体に inode 構造体を 1 つ確保 (bd_inode) し，そこにディスクブロック番号をキーとした木構造を保持して間接ブロックなどを読込んだページを登録している．ブロックデバイスにひとつだけなので，そのデバイスから読込んだすべてのファイルの間接ブロックがそこに登録されている．

5.2 nilfs のメモリ上のデータ構造

Linux カーネルに変更を加えない方針をとったので，ブロックデバイス入出力，ページキャッシュ管理，システムコールからの処理の実行は Linux の処理に合わせなければならない．

LFS での大きな違いは，ファイルを新規に作成してデータを書いていくときには，ディスク上の位置は実際に書出す直前まで決まらないことである．ユーザデータのブロックについては論理的なファイルブロック番号を使った Linux の標準的なブロック管理が可能だが，ディスクブロック番号のない新規 B-Tree 中間ノードは bd_inode に登録することができない．そこで，ディスクブロック番号に代わるものとして，メモリ内のバッファヘッドのアドレスを用いる．nilfs ではメモリ上の中間ブロックの管理用に radix-tree を使っている．この radix-tree のルートはメモリ上 inode 構造体に保持し，ファイルごとにメモリ上中間ブロックを管理するようにし，木が大きくならないようにした．

5.3 セグメント構築処理

sync システムコールや pdflush デーモンから起動されるデータ書出し処理は以下の順序で進む．

(1) 変更のあったファイルデータのページを集める
(2) 同様に inode 管理の B-Tree 中間ノード，ファイルブロック管理の B-Tree 中間ノードのページをそれぞれ登録されている radix-tree から集める
(3) inode 管理 radix-tree から変更のあった inode を集める
(4) ディスクブロック番号が変わることで書換えることになる B-Tree 中間ノードのブロックもこの時点で集めておく（5.4 参照）
(5) 書出すブロックに新たなディスクブロック番号を割当てる．ファイルデータブロック，ファイルブロック管理用 B-Tree の中間ブロック，inode を納めたブロック，inode 管理用 B-Tree の中間ブロックの順に割当て，書出し順序を決定する
(6) B-Tree や radix-tree でディスクブロック番号を保持している構造体はこの時点で新しいブロック番号に書換える
(7) この順序でカーネルのブロックデバイス入出力部（bio 部）に書出しの要求をする

POSIX セマンティクスを維持するためには，ディレクトリ操作は排他的に実行されなければならない．このため，現在は，セグメント構築とディレクトリ変更を含むような操作を排他的に実行している．セグメント構築はファイルシステム全体にかかわる操作なので，ユーザプロセスのディレクトリ変更要求はセグメント構築中にはロックされている．また，個別のファイルのみをディスクに書出す fsync システムコールについても，現在はファイルシステム全体の書出しと同じ動作となっている．このロックの期間を短くすること，個別 fsync の実現は今後の課題である．

5.4 dirty 伝搬

LFS は信頼性確保によい方式だが，上書きをしないために書出しのデータ量が増える．

ここで，ファイルのあるブロックを書換えたとしよう．このブロックは新たな位置に書出される．すると，このブロックを管理している B-Tree 中間ノードのポインタ部が変更される．中間ノードも上書きしないので，この中間ノードも新たな位置に書出される．同様に，この中間ノードを指していた上位の中間ノード，または，B-Tree のルートブロックのポインタ部が変更され，このブロックも新たな位置に書出されることになる．このように，ブロックの書換えによって木のルート方向へのブロックはすべて書出されることになる☆．ブロックを上書きすればそのブロックだけを書出せばよいので，LFS の書出し量は大きい．

特に，ファイルの書出しオープン時に，オプション O_SYNC を指定して書込みが直ちに物理デバイスに反映されるよう要求したとき，nilfs では大きな空間的オーバーヘッドが発生する．データを 1 ブロックだけ書出すのにも dirty 伝搬による B-Tree のブロックの書出しが発生し，それにセグメントサマリ，チェックポイントを加えて部分セグメントを書出すことになる．例えば，現在の nilfs に 1Mbyte のファイルを作るとき，4Kbyte ブロックで通常は 265 ブロックを書出すが，O_SYNC 指定では 4Kbyte ごとに write したとき 1541 ブロックを書出す．

現在，われわれは dirty 伝搬による量の増大については信頼性確保のためのオーバーヘッドとして許容している．O_SYNC については，チェックポイントを簡易なものにする等の対策を検討中である．

5.5 ガーベージコレクション（GC）

nilfs の GC は，まずセグメントサマリと B-Tree の木の内容からブロックが使用中かどうかを判定し，使用中ブロックに dirty フラグを付ける．これをセグメント内の各ブロックについて実行する．dirty の付いたブロックは次のセグメント書出して新しい場所に移動する．現在は実装中で速度測定などはできていない．

5.6 スナップショット

nilfs では，ファイルシステムの首尾一貫性をもったバックアップの作成や，多数のファイルの変更を回復するといった目的のため，ファイルシステムのスナップショットを作成することができる．上書きのない LFS では，GC で回収されるまでは過去のデータがすべてディスクに残っている．よって，ブロック管理情報さえ得られればスナップショットのために新たに保存しておくべき情報はない．nilfs ではブロック管理情報であるブロック管理 B-Tree，inode，inode 管理 B-Tree をすべてログとして管理している．つまり，inode のルートが格納されているブロック番号がスナップショットに他ならない．この番号は部分セグメントのチェックポイントに格納されている．nilfs のチェックポイントはファイルシステムのスナップショットそのものである．実際，mount コマンドにチェックポイントのブロック番号を指定することで，その時点のファイルシステムを読出し専用ファイルシステムとして任意のディレクトリにマウントすることができる．

ただし，すべてのチェックポイントをスナップショットとして保持すると，GC が回収できる領域がなくなる．よって，チェックポイントのうち，利用者の指定したものだけを保持し，他は GC で回収する

ディスクの未使用領域がある程度あれば，消去したファイルの領域が直ちに回収される可能性は低い．スナップショット指定をせずとも，直前に消してしまったファイルを復活するといった機能は実現できる．利用者の操作誤りを救済するユーザフレンドリなファイルシステムの 1 つの機能として実現予定である．

将来，ディレクトリを B-Tree で実装した後，ディレクトリ単位でスナップショットをとる機能の実現性を検討する．これが実現できれば，elephant[13] システムのように利用者のファイル変更の適切なバージョン（elephant の landmark にあたる）のスナップショットを自動的に取得する機能も導入できる．

5.7 セキュリティ面の課題

過去のデータがすべて保持されることはセキュリティ面では問題となる．rm コマンドで消去したり，ランダムデータでファイルを上書きすることは LFS ではデータを破棄することにはならない．効果的なセキュリティ確保は今後の課題である．

6. 評　　価

iozone で nilfs と ext3 ファイルシステムを比較した．測定計算機は，Pentium 4 3.0GHz，メモリ 1G バイト，Linux kernel 2.6.10，ディスクは Ultra ATA 接続，回転数 7,200rpm，バッファ8MByte である．ディスクブロック，ページキャッシュのサイズはどちらも 4KByte である．測定は，データ量を 32MByte から 512MByte，1 回の write システムコールで書出す単位を 2KByte から 8KByte で測定した．ファイルの書出し開始から fsync システムコールで変更が実際にディスクに反映されるまでの時間を測定し，スループッ

☆ ディレクトリの木構造に沿って書出すのではない

表 1 ファイルの連続書出し (MB/s)
Table 1 continuous write

	EXT3			nilfs		
	2K	4K	8K	2K	4K	8K
32M	45.6	45.6	48.0	51.5	53.6	54.3
128M	46.7	49.3	49.8	51.7	53.3	54.0
512M	52.6	53.4	53.0	50.3	52.4	51.9

表 2 ファイルのランダム書出し (MB/s)
Table 2 random write

	EXT3			nilfs		
	2K	4K	8K	2K	4K	8K
32M	20.3	13.3	23.9	59.9	83.2	84.7
128M	8.1	5.7	17.5	47.5	82.6	83.8
512M	3.0	3.4	6.6	31.0	59.0	67.4

表 3 O_SYNC を指定した測定結果 (MB/s)
Table 3 measurement with O_SYNC option

	EXT3			nilfs		
	2K	4K	8K	2K	4K	8K
	ファイルの連続書出し					
32M	1.3	2.0	6.8	2.8	5.2	9.5
128M	1.2	2.7	6.8	2.8	5.1	9.3
512M	1.3	2.6	5.3	2.6	4.8	8.7
	ファイルのランダム書出し					
32M	1.2	2.4	5.1	3.1	6.2	10.7
128M	0.9	1.8	3.5	3.1	6.1	11.0
512M	0.6	1.3	2.4	2.6	5.1	8.9

ト MByte/sec を計算した．ext3 のジャーナリングは，デフォルトの ordered mode である．ブロックデバイスレイヤでは kernel2.6 のデフォルトである anticipatory（予測付きエレベータ）アルゴリズムが適用されている．nilfs では GC は動作していない．

まず，単純なファイルへの連続書出し速度を計測した．結果を**表 1** に示す．nilfs は書出しファイルが大きくなると性能が低下する．カーネルメモリの使用量が大きいこと，書出すブロック数が ext3 より多いこと，などが原因と考えている．

次に，存在するファイルのランダムな位置のデータを書換える場合の計測結果を**表 2** に示す．シークの少ないログ構造化方式の大きな効果が認められる．

上記の測定に O_SYNC を指定した結果を**表 3** に示す．nilfs は書出しブロック数は多くなるが，シーク動作が抑えられているので，結果として ext3 よりよい値が得られている．

nilfs は性能チューニングを行なっていない現状でも，ext3 と匹敵するか場合によっては高い書出し性能が出ることが確認できた．今後，チューニングを進めたい．また，性能を損なわない GC の実装を進めたい

7. ま と め

Linux 用の新しいローカルファイルシステム nilfs について目標，設計，実装について述べた．B-Tree とルート inode 方式の近代的なログ構造化方式が実現できた．nilfs は GPL で公開する予定である．今後，性能チューニング，i386 アーキテクチャ以外のマシンでの動作確認，ツール類の整備をしていく．同時に，ディレクトリの B-Tree 実装，スナップショット機能の拡充を進めていく予定である．

参 考 文 献

1) Stephen Tweedie: Journaling the Linux ext2fs Filesystem, *LinuxExpo '98*, 1998
2) Project XFS Linux, *http://oss.sgi.com/projects/xfs/*
3) ReiserFS, *http://www.namesys.com/*
4) JFS for Linux, *http://jfs.sourceforge.net/*
5) Marshall K. McKusick, William N. Joy, Samuel J. Leffler and Robert S. Fabry: A Fast File System for UNIX, *Computer Systems*, Vol.2, No.3, pp.181–197, 1984
6) J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson and B. B. Welch: The Sprite Network Operating System, *Computer Magazine of the Computer Group News of the IEEE Computer Group Society, ACM CR 8905-0314*, Vol.21, No.2, 1998
7) Mendel Rosenblum and John K. Ousterhout: The Design and Implementation of a Log-Structured File System, *ACM Transactions on Computer Systems*, Vol.10, No.1, pp.26–52, 1992
8) Margo I. Seltzer, Keith Bostic, Marshall K. McKusick and Carl Staelin: An Implementation of a Log-Structured File System for UNIX, *USENIX Winter*, pp.307–326, 1993
9) The NetBSD Project, *http://www.netbsd.org/*
10) Christian Czezatke and M. Anton Ertl: LinLogFS — A Log-Structured Filesystem For Linux, *Freenix Track of Usenix Annual Technical Conference*, pp.77–88, 2000
11) R. Bayer and E. McCreight: Organization and Maintenance of Large Ordered Indexes, *Acta Informatica* Vol.1, Fasc. 3, pp.173–189, 1972
12) Douglas Comer: The Ubiquitous B-tree, *ACM Computing Surveys* Vol.11, No.2, pp.121–138, June 1979
13) D.J. Santry, M.J. Feeley, N.C. Hutchinson and A.C. Veitch: Elephant: The file system that never forgets, *Proceeding of IEEE Hot Topics in Operating Systems*, pp.2–7, March 1999.