Linux用ログ構造化ファイルシステム

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 (New Implementation of a Log-Structured File System) を開発中である．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．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の設計と実装方法，実装経験について述べる．

1. はじめに

ファイルシステムはOSの重要な機能である．永続データの保持は計算機のもっとも基本的な要求といえる．計算機に備わったローカルなハードディスクに加え，ネットワークを越えてファイルシステムを拡張する方式もいくつも考案され実用となっている．

1.1 ファイルシステムの基本機能

ここで，ローカルなファイルシステムに関して求められる機能を整理してみると以下の項目があげられる．

確実に書けていること，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そのものの故障に対処できること（可用性，耐故障性）．故障があったとき書いたデータを高速に，かつできるだけ多く復旧でき，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，復旧時には完了しているかまたは書込み操作以前の状態となること．ファイルシステム全体が壊れないこと（堅牢性）．書込み速度，読込み速度，ディスクの使用効率がよいこと（高性能）．ファイル名の多言語化，ディスクの増設，縮退が容易でかつヒューマンエラーの救済ができること（運用性，可伸性，使い易さ）．

さらに，Linuxでは，フラッシュメモリ向けといった特殊なデバイスに特化したものでなければ，POSIXのセマンティクスを守っていること，VFSと親和性のよいこと，カーネルのメモリ管理，入出力システムと整合性のとれていることも重要である．

† 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

1.2 Linuxのジャーナリングファイルシステム

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，すでに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，ジャーナリング機能だけで信頼性が確実になるわけではない．例えばext3では，ジャーナリングを利用者に隠されている通常のファイルにログを書出すことで実現している．このログの書出しでは，データの書出し，inodeの更新など，同期型の通常のファイル書込み操作が実行される．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．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たりしないよう配慮がいる．だが，例えばディスク書込みのエレベータアルゴリズムが適用されれば，ヘッドシーク時間の短縮のためにファイルシステムとしての正しい順序を無視した書出しが実行されることがある．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．システムの下位レイヤの動作が，信頼性の確保に影響しない方式が望まれる．

さらに，ext3のジャーナルモードでは，ユーザデータそのものもジャーナルファイルと実際のディスク領域との2ヶ所に書いている．書き出すデータが倍になること以上に，固定位置のジャーナルファイルと実際の書き出し位置の間にシーク動作が必要となることで書き込み性能が大幅に低下する．

性能を確保するため，デフォルト設定ではジャーナ

ルには更新の記録とメタデータのみを書き，ユーザデータは直接本来の位置に書出すメタデータジャーナリング (ordered mode) となり，この場合は，ユーザデータの保護が犠牲になる．

1.3 ディスクブロックの上書き

ファイルにランダムな書込みをしていて，あるディスクブロックを書いている時に電源障害が発生したら，そのブロックの内容は壊れていると思ったほうがよい．ジャーナルログファイルに書き出すときも同じである．

ファイルシステムの矛盾とユーザデータの破損を招く原因は，書込み順序の誤り，ディスクブロックの上書きが大きな原因である．障害発生時の復旧を目的としたジャーナリングは，このどちらの問題も部分的にしか解決しない．

1.4 LFS

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[5)6)], 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．すでに存在するディスクブロックを書換えないことで，書換えに起因する問題を解決する．また高速化のために書き出しの順序を犠牲にすることがない．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を開発中である．本論文では開発中のファイルシステムについて述べる．

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 LFS の原理について紹介する．4,5 章で我々の開発している LFS の設計と実装について述べ，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) 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

(2) 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

(3) 運用性に優れること．バックアップや誤消去したファイルの復旧のため，スナップショットが短時間で取れること

(4) 性能・機能が現存の ext2/ext3 ファイルシステムと同等かそれ以上であること

これらを実現する過程では，Linux のセマンティクスを守ることとカーネルコードへの変更を避けることを堅持した．導入を容易とし，カーネルのバージョンアップへの追従性を確保するためである．

実装では，まず基本的な動作が確認できるシンプルなコードを書くことを優先した．性能チューニングは，性能測定でボトルネックを把握した上で実施する予定である．

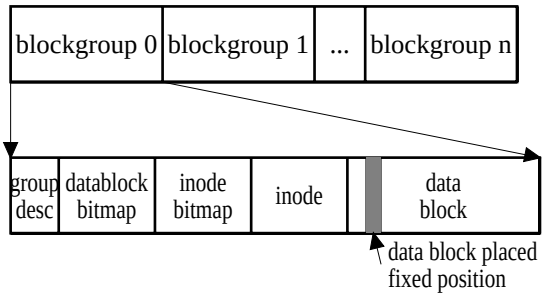

図 1 従来のファイルシステムのブロック配置

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書換え（上書き）が大きな原因である．LFS は書換えの回避について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 は固定位置に配置される．ファイルのブロックはデータを書出すシステムコール発行時に割り当てられる（実際のディスクへの書出しは後になるかもしれないが，ディスクのどの位置に書くかは先に決定される）．すでに書出したブロックの内容を書換えるときには，その同じブロックを上書きする．ファイルのサイズや変更時刻は書出し時に更新するが，記録する inode は固定位置にあるので，これもディスクブロックの上書きとなる．ファイルが大きくなったときには，ファイルのブロック位置を記録するためのブロック（間接ブロック）を割り当てて管理する．また，ディスク全体で未使用のブロックを特定するために，ビットマップが使われている．この間接ブロックやビットマップもディスク上の固定位置に書出されている．

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．このときには割り当てられていたブロックは開放される．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り当てる．

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

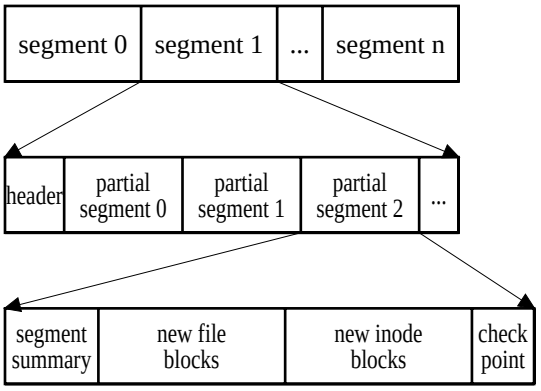

図 2 LFS のブロック配置

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．

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り当てる．ファイルの最後にデータを追加していくときは当然従来方式でも新しいブロックを割り当てるが，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り当てる．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り当てて書出す．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，ログ構造化ファイルシステムと呼ばれている．

LFS の利点は，書出しで常に新しいブロックを割り当てることで，書出し中に電源の切断などの障害があったときも他のブロックを壊さないことである．

また，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ディスクのブロックアドレスを決定する．ここで，ブロックアドレスを，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 を保持するための間接ブロック，inode 全体の管理の根となるブロック，の順に連続して割り当て，この順で書き出せば，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．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) では特定の場所のみ検査をすればよい．

また，従来方式ではディスクの異なった位置への複数回の書込みになるものが連続した書込みになる．これによりデータ書出し性能の大幅な向上が期待できる．シークを減らすために書出し順序を入れ換えるといった操作も不要である．

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，ディスク上の 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は異なる．

さらに，ディスクの容量は有限であるので，いずれ未使用領域はなくなる．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，データの断片化が発生する．ファイルの連続順読込みの性能の低下を招くので，必要に応じてファイルデータを整列する機能も必要である．

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よくとられる．例えば 4M バイトの固定サイズでディスクを区切り，未使用ブロックをこの単位で管理する．この管理単位をセグメントと呼ぶ．

ext2/3 のブロックグループ，BSD FFS[7] のシリンダグループなど，ディスクをいくつかの部分に区切って使用する例はあるが，ファイルブロックやそれに関連したブロックをディスクの近い位置にまとめ，読込みアクセス時のシーク量を減らして高速化することが目的である．これに対して，LFS のセグメントは，ブロックを近接させるのが目的ではなく，GC の管理単位としてのみ使用される．

LFS はデータベース分野で考案されたものである．極力ブロックの上書きを避けることで高い信頼性が得られる．だが，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 NetBSD で利用できる程度である．Linux でも 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[8] があったが，実験段階に留まった．これは，LFS の実装が難しいことに起因している．特に，ディスクブロック番号の対応や inode 番号からブロック番号への対応関係が毎回変化する点で，効率のよい実現が困難であった．

4. 設計と実装

われわれの開発中のファイルシステム nilfs の第 1 版では，信頼性と堅牢性の確保を主の目標とした．性

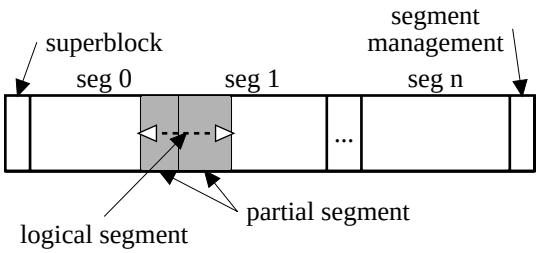

図 3 nilfs のディスクレイアウト

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[9] 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．

ファイルブロック，ファイル管理 B-Tree 中間ノード，inode ブロック，inode 管理 B-Tree 中間ノードはすべてログの形でディスクに追記される．

B-Tree はディスク上に木構造を構築するときによく使われるデータ構造である．平衡木の一種で，データを増やしていったとき木の高さが平衡するように木の組み替えをする．B-Tree には変種として順アクセスの高速化をはかった B^+-Tree，記憶空間の利用率の向上をはかった B*-Tree 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．B-Tree を構築するために必要としているブロックが 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．

B-Tree を採用した理由は，大きなファイルを効率的に管理するためであるとともに，動的に変化する論理/ディスクブロック番号対応，inode 番号/ブロック番号対応の機能を統一的かつ効率的に扱うためである．

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．

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（後述）の位置が格納されている．これが有効なデータの根となる情報である．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，スーパーブロックがあ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分だけ小さい．

部分セグメント 書出しの単位．sync/fsync 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．

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．そのような操作の途中で障害が発生したときには，その操作は完了しているか，または全く操作されなかったのどちらかの状態でなければならない．例えば，2 つの操作（1）ファイル A を生成,（2）ファイル B を生成，をこの順序で実行したとする．実行中に電源断などの障害が発生したとき，再起動後にファイル B だけが生成されていてファイル A がないことは許されない．この場合は，A も B もない，A だけある，A も B もある，のいずれかでなければならない．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．

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である．このブロックは GC が利用する．この情報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て，同じ情報を複数箇所に書出す．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．

現在の nilfs のセグメントサイズは 4M バイトで固定している．このサイズに依存したコードは含まれていないので，フォーマット (newfs) 時に変更するといったことは容易に実現可能である．

セグメントサイズは読み書きの性能にはほとんど影響しない．書込みの単位は部分セグメントであるし，読込みはブロックが単位で，セグメントを意識する場面はない．

だが，GC はセグメントを単位にディスクの未使用領域を管理する．管理のため，セグメントごとにメモリ上に構造体を用意するので，セグメントサイズは nilfs のトータルな性能には影響する．現状ではセグメントサイズと性能の関係は測定できていないが，ディ

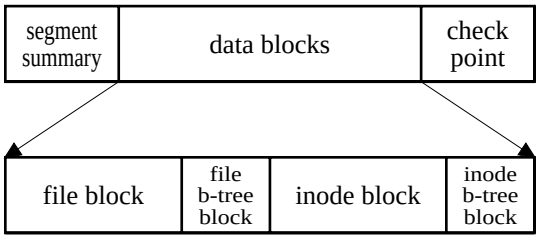

図 4 部分セグメント

スクサイズとセグメントサイズの関係について，newfs のデフォルトの決定，サイズ指定の指針を示す予定である．

4.2 部分セグメントのレイアウト

部分セグメントの内容を示す．図 4 のようにサマリ，データ，チェックポイントの 3 つの部分からなる．

セグメントサマリ 部分セグメント管理用のデータを納めてある．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．

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格納されている．

チェックポイント 部分セグメントの最後尾に置く．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．チェックポイントも通常のデータアクセスでは参照されない．

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 管理 B-Tree のルートのディスク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．

論理セグメントの最後のチェックポイントは，首尾一貫性のあるファイルシステムのルートとなる．このチェックポイントは，そのまま nilfs のスナップショットと考えることができる．ブロックの上書きがされないので，チェックポイント書込み時の状態はしばらくそのまま保持される．mount のオプションで指定することで，チェックポイントをファイルシステムとみなして読込み専用マウントする機能を計画中である．一貫性のある状態でのバックアップを取得したり，消去したり，過去のバージョンのファイルを復活させることが可能となる．

4.3 信頼性の確保

nilfs の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．

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. ファイルシステムの実装

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☆をキーとして読込んだページ

☆ 1 ページに複数のディスクブロックを保持できる場合にはペー

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（メンバー名は 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 bd_inode 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が付与されるので 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 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，ファイルごとに読込んだ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 ディレクトリ

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，可変長の名前をキーとする B-Tree は実装が複雑で，現在は実現していない．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，大量のファイルをもったディレクトリ操作の性能はよくない．

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．LFS であるので，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ことになる．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るのなら，そのブロックだけを書出せばよいので，LFS の書出し量はかなり大きい．我々はこの書出し量の増大について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．

5.5 ガーベージコレクション

nilfs のガーベージコレクションは，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する．次にそのブロックに 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．ただ，現状では実装中で速度測定などはできていない．

5.6 実 装 経 験

現在，nilfs は Linux カーネル 2.6.10 のローダブルモジュールとして開発している．2.6.11 でも軽微な変更で動作する．カーネル側の変更は不要である．現在は i386 アーキテクチャでのみ動作を確認している．

開発環境は，ソース管理に cvs，バグ管理や開発者間のコミュニケーション，ソース解析結果の保持に pukiwiki，Linux ソース閲覧に LXR Cross Reference を使用している．カーネルのデバッグにはエミュレータの qemu を一部で使用したが実機との違いから使用できない場合があった．また，kgdb といったデバッガは Linux カーネルのバージョンに追従しておらず，今回は使用しなかった．結局，カーネル内でメッセージを出力する printk にたよることとなった．

printk の出力は通常は syslogd または klogd 経由でファイルに書かれる．しかし大量のメッセージを書出すとメッセージのかなりの部分が欠落する．ユーザプロセスである syslogd がカーネルの実行中に動けないのは自明である．printk からのメッセージは /proc/kmsg を読むことでも得られる．軽い cat プログラムで/proc/kmsg を読込んでファイルにリダイレクションすることで，syslogd 経由の数十倍のメッセージを保存することができた．

カーネルモジュール開発中のバグは，ほとんどの場合計算機がハングする結果となる．ハングの原因は主

ジ単位の番号

☆ この inode の宣言部にはこんなコメントがついている
struct inode *bd_inode; /* will die */

にスピンロックやセマフォのデッドロックによるもの，メモリが逼迫して動かなくなるものに大別される．メモリの逼迫は，バッファの参照カウンタの増減ミスによって開放されるべき領域が開放されないことで発生する．これらを発見するために網羅的チェックが必要となった．スピンロック操作，セマフォ操作，バッファヘッドのカウンタ増減操作の関数すべてをマクロでデバッグ出力つきの関数に置き換え，計算機のハングまでに取得したメッセージを解析する．1G バイトを越えるようなメッセージが出力されるが，フィルタリングを工夫することで取り扱うことができる．

カウンタの増減操作はメッセージ出力とともに，モジュール内変数に操作回数を記録し，/proc 経由で変数内容を確認できるようにしている．これらの手法を組み合わせてモジュールをデバッグした．

スピンロックのデッドロックデバッグには qemu も役立った．

カーネルの関数や構造体メンバー名の名前には注意が必要である．例えば，ページを alloc_page() で割当ててもらったら，使用後に返却するときには __free_page() 関数を使わなければならない．下線の付かない free_page() も存在しているが，これは __get_free_page() 関数で割り当ててもらったページを返すのに使う．また，メモリ上の inode を保持する構造体に unsigned long のメンバー i_blksize があるが，これはファイルシステムのブロックサイズではなく，ディスク入出力時の最適なサイズを指定するものであった．これら，技術者の直感に反するミスリーディングな名前に起因するバグも経験した．

6. 評　　価

ベンチマークプログラム iozone を使用して ext3 ファイルシステムと比較した．使用した計算機は，Pentium 4 3.0GHz，メモリ 1G バイト，ディスクは ATA 接続 7,200rpm ディスクである．kernel のバージョンは 2.6.10 である．ext3 のジャーナリングは，デフォルトである ordered mode である．

まず，単純なデータの書出し速度を，書出すファイルの大きさ，1 回の write システムコールで書出す量を変えて測定した．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，スループットとして表示している．図 5 に結果を示す．

nilfs は書出しファイルが大きくなると性能が低下している．いまのところカーネルメモリの使用量が大きいこと，そもそも書出すブロック数が多いこと，チェックサム計算の負荷，などが原因と考えている．

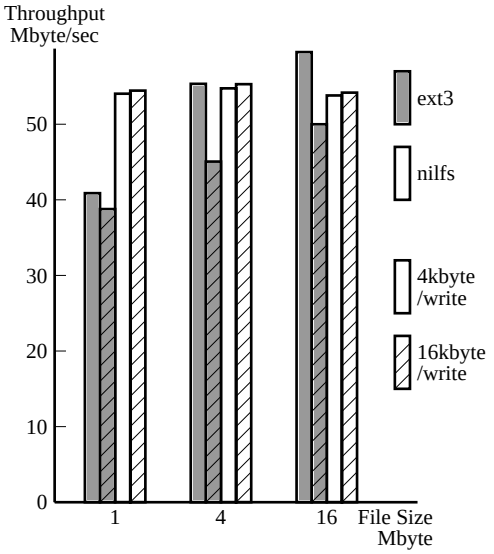

図 5　ファイル書出し

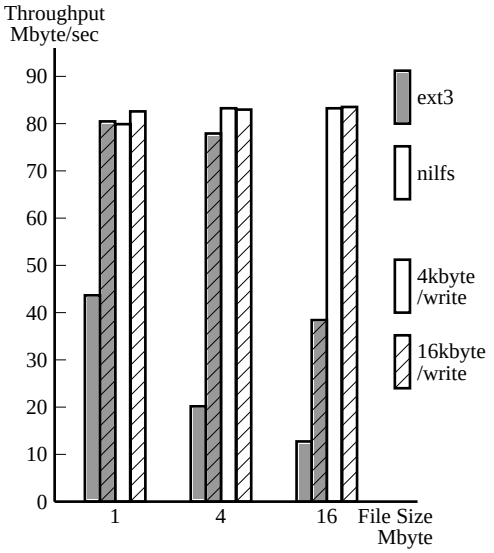

図 6　ランダム書出し

次にランダム書出しの結果を示す．ファイルの大きさが大きい場合は，nilfs は ext3 よりかなり高速に書出せていることがわかる．書出し時にシーク動作が少ない LFS の利点が出ている．

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それより高い書出し性能が得られた．

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，実装経験について述べた．ログ構造化方式を採用したことで，高い信頼性とユーザの利便性を確保できると期待している．

現在の状況を開発目標ごとにまとめる．

(1) **高信頼:** 正しい書出し順序の維持，上書きの排除は実現できている．

(2) **高可用:** mount 時に最後に書出されたセグメントの正当性をチェックしている．包括的な整合性チェックをする fsck ツールは未了．

(3) **運用性:** スナップショットの取得と，スナップショットを参照する基本的機構は実現できている．

(4) **性能・機能:** ext3 と同等か，場合によってはそれより高速である．

今後，安定性の確保，ツール類の整備をしていく．機能面では以下の項目を実現したい．

(1) 動的な構成の変更
ディスクの追加，取外しを無停止で可能にする．ファイルシステムサイズの動的な拡張や縮退機能を含む．

(2) 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

(3) 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．例えば，ディレクトリ名，ファイル名を UNICODE 対応にすれば解決するといった問題ではない．文字コードの唯一性の確保のための正規形の生成，いわゆる全角文字と半角文字といったフォントに関わる問題，表記や送り仮名のゆれといった個人の嗜好や個人辞書に関わることなど，解決すべき問題は多い．

(4) ユーザフレンドリ機能
システム障害の原因の多くは人間の操作誤りである．消去したファイルの復活など，人間の誤りをファイルシステムでも支えたい．また，現在はファイルを名前で特定しているが，キーワードの組や画像など，ファイルを特定する手段はもっと多様であってよい．

(5) 新機能の追加，拡張が容易であること

nilfs は GPL で公開する予定である．まだ基本的な性能測定が可能となった段階であるが，オープンソースとして皆様の協力を得て開発を進め発展させていきたい．

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) 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
6) 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
7) 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
8) Christian Czezatke and M. Anton Ertl: LinLogFS — A Log-Structured Filesystem For Linux, *Freenix Track of Usenix Annual Technical Conference*, pp.77–88, 2000
9) R. Bayer and E. McCreight: Organization and Maintenance of Large Ordered Indexes, *Acta Informatica* 1, Fasc. 3, pp.173–189, 1972