情報処理学会論文誌　コンピューティングシステム　Vol. 2　No. 1　110–122 (Mar. 2009)

ログ構造化ファイルシステムNILFSの設計と実装

佐藤孝治[†1]　小西隆介[†1]　木原誠司[†1]
天海良治[†1]　盛合敏[†1]

本論文では，ログ構造化ファイルシステム NILFS の設計と実装について述べる．NILFS は任意の時点における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，ソフトウェア障害やユーザエラーからデータを保護する．また，ディスク上のデータ構造はつねに一貫した状態に保たれるため，システム障害後の迅速な復旧が可能である．従来のログ構造化ファイルシステムとは異なり，ディスクアドレス変換を用いることにより，複数のスナップショットが存在する状況で，クリーナは不要になったディスク領域を効率的に回収することができる．評価実験により，NILFS は Ext3 と比べて遜色ない性能を有することを示す．

The Design and Implementation of the NILFS Log-Structured File System

Koji Sato,[†1] Ryusuke Konishi,[†1] Seiji Kihara,[†1]
Yoshiji Amagai[†1] and Satoshi Moriai[†1]

This paper describes the design and implementation of the NILFS log-structured file system. NILFS provides file system snapshots at any point in time and protects data from software failures and user errors. The log-structured disk layout maintains on-disk data structures consistently and achieves fast recovery from system failures. Unlike other log-structured file systems, the cleaner can reclaim obsolete disk space efficiently under the existence of multiple snapshots using disk address translation mechanisms. The results of evaluation experiments show that the performance of NILFS is comparable to that of Ext3.

†1 日本電信電話株式会社 NTT サイバースペース研究所
NTT Cyber Space Laboratories, NTT Corporation

1. はじめに

ファイルシステムの信頼性は計算機システムにおける重要な要件である．ファイルシステムに保存されているデータは様々な障害から保護されなければならない．データはハードウェア障害，ソフトウェア障害，停電，ユーザエラーなどによって失われたり，矛盾した状態になったりすることがある．RAID などのハードウェア冗長化技術はハードウェア障害からデータを守ることができるが，ソフトウェア障害やユーザエラーなどによるデータの損失には有効ではない．従来，このような障害からデータを守るために定期的なテープバックアップが用いられてきた[3]．しかし，この方法はデータの保存や復旧に多くの時間がかかる．また，バックアップ周期が長いため，その間に変更されたデータを復旧させることはできない．

近年，ディスクの大容量化や低価格化にともない，データの変更履歴をディスクに保存しておくことが現実的になった．過去のある時点のデータはバックアップ，データマイニング，ソフトウェアテストなど，様々な用途に利用することができる[1]．以前のデータにアクセスできるようにするために，スナップショットやバージョニングの機能を備えたファイルシステムや継続的データ保護（Continuous Data Protection（CDP））が提案されている．

スナップショットファイルシステム[8),16),24)] はファイルシステムの状態をスナップショットとして保存することができる．これにより，以前のデータの復旧やオンラインバックアップが可能となる[3]．しかし，スナップショットはユーザからの要求時やあらかじめスケジュールされた時点に作成されるため，操作ミスにより変更または削除されたファイルを元の状態に復旧させるためには，その操作が実行される前にスナップショットを作成しておかなければならない．

バージョニングファイルシステム[5),14),18),22)] はファイルのオープン/クローズセッションごと，または書き込みごとにファイルの変更履歴を自動的に保存する．そのため，ユーザは事前に準備することなく，以前のファイルを復旧させることができる．しかし，オープン/クローズセッションごとのバージョニングでは，セッション内の変更は 1 つのバージョンに集約されるため，その間の変更を元に戻すことはできない．一方，書き込みごとのバージョニングでは，任意の時点のファイルを復旧させることができるが，以前のバージョンを再構築するためには，それまでの変更を順次取り消す必要があるため，変更量に比例してオーバヘッドが大きくなる．

CDP[23)] はすべてのデータの変更を主ストレージとは別のストレージに連続的に保存する．これにより，ストレージの状態を任意の時点に復旧させることができる．しかし，変更

履歴の保存期間が過ぎると，それらは順次削除されるため，ある時点のストレージの状態を長期間保存しておくことはできない．

様々な障害からデータを守るために，我々はログ構造化ファイルシステム NILFS を開発している．NILFS は任意の時点における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，ソフトウェア障害やユーザエラーからデータを保護する．また，ディスク上のデータ構造はつねに一貫した状態に保たれるため，システム障害後の迅速な復旧が可能である．

本論文では，NILFS 第 2 版の設計と実装について述べる．以前の NILFS 第 1 版[12] では，任意の時点におけるファイルシステムのスナップショットを作成することは可能であったが，不要になったディスク領域を回収するためのクリーナが実装されていなかった．NILFS 第 2 版では，従来のログ構造化ファイルシステムとは異なり，ディスクアドレス変換を用いることにより，複数のスナップショットが存在する状況で，効率的なクリーナを実現することを可能にしている．

2. ログ構造化ファイルシステム

様々な障害からデータを守るために，NILFS はログ構造化ファイルシステム（Log-Structured File System（LFS））[17),19)] を基本としている．LFS はファイルシステムのデータ構造を連続的なログとして保存する．ファイルシステムへの変更は上書きされず，ログの論理的な末尾に追記される．LFS では，ディスクは固定長の**セグメント**に分割される．論理的に連続したログはセグメントのリストとして構成される．ディスク容量は有限であるため，LFS はすでに書き込まれたセグメントを定期的に**クリーニング**する．**クリーナ**はセグメント内の有効なブロックである**ライブブロック**をログの末尾に再度書き込み，セグメントを再利用できるようにする．変更または削除されて無効になった**デッドブロック**は捨てられる．1 回の書き込みでセグメントをすべて埋められるとは限らないため，セグメントは 1 つ以上の**部分セグメント**に分割される．セグメントはクリーニングの単位であり，部分セグメントは 1 回の書き込みの単位である．

LFS の構造を**図 1** に示す．ここでは，セグメント $_1$ は 2 つの部分セグメントで構成されている．部分セグメント $_{12}$ はファイル $_1$ のデータブロック F_{11} と F_{12}，ファイル $_2$ のデータブロック F_{21} と F_{22}，およびファイル $_1$ とファイル $_2$ の i ノードを保持する i ノードブロック I を含む．次に，ファイル $_1$ のデータブロック F_{12} を変更し，データブロック F_{13} を追加する．また，ファイル $_2$ のデータブロック F_{22} を削除する．このとき，変更されたデータブロック F_{12} は上書きされずに，新しいデータブロック F_{13} とともに部分セグメント $_{21}$ に書き込まれる．また，i ノードブロック I は変更や追加，削除されたデータブロックを反映するように変更され，同様に部分セグメント $_{21}$ に書き込まれる．このように，ファイルシステムへの変更はログの末尾に順次追記されていく．

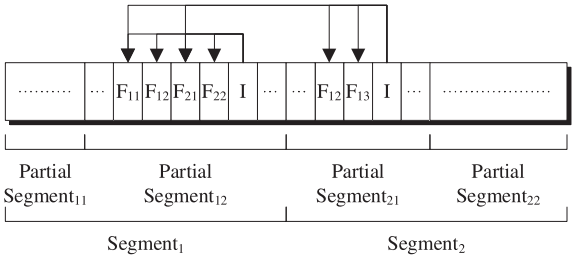

図 1　ログ構造化ファイルシステム
Fig. 1　Log-structured file system.

LFS は定期的に**チェックポイント**を作成する．チェックポイントはファイルシステムの一貫した状態を保持しており，システム障害後の復旧に利用される．LFS は変更されたブロックを元のブロックとは異なる位置に書き込むため，ファイルの i ノードは固定位置に配置されなくなる．そのため，i ノード番号から現在の i ノードの位置を特定する **i ノードマップ**を用意する．また，セグメントを管理するための**セグメント利用テーブル**を用意する．クリーナはこの情報を用いて回収すべきセグメントを選択する．チェックポイントが作成されるとき，i ノードマップやセグメント利用テーブルはディスクに書き込まれる．

LFS では，データは上書きされず，クリーナがデータを回収するまでは以前のファイルシステムの状態がそのまま保存されているため，これを利用して永続的な**スナップショット**を実現することができる[15)]．また，ディスク上のデータ構造がつねに一貫した状態に保たれるため，ジャーナリング[21),25)] や Soft Updates [7)] のような付加的な機構を用いることなく，システム障害後の迅速な復旧が可能である．しかし，スナップショットの実現は効率的なクリーニングを困難にする．いくつかの LFS [10),11)] はスナップショットをサポートしているが，その機能は著しく制限されており，バックアップ用途に限定される．これとは対照的に，NILFS は柔軟なスナップショット作成と効率的なクリーニングを実現している．

3. 設　　計

過去のどの時点のデータが必要になるかは事前には分からないため，ファイルシステムは

すべてのデータの変更履歴を保存し，任意の時点の状態に戻れるようにしておかなければならない．しかし，ディスク容量は有限であるため，それらを永久に保存しておくことはできない．そのため，NILFS では，作成後一定期間内のチェックポイントはすべて保存しておき，それを過ぎると，指定されたチェックポイントのみを永続的に保存し，その他は削除するようにする．永続的に保存されるチェックポイントをスナップショットと呼ぶ．

3.1　チェックポイントとスナップショット

従来の LFS は定期的にチェックポイントを作成し，システム障害が発生したときには，最新のチェックポイントを用いてファイルシステムを一貫した状態に復旧させることができる[17),19)]．しかし，チェックポイントはファイルシステム内部で復旧のためにのみ用いられており，ユーザがチェックポイントを利用する方法は提供されていない．

NILFS はチェックポイントを過去の時点におけるスナップショットとしてユーザがアクセスできるように拡張する．チェックポイントはチェックポイント作成間隔が過ぎたときや，sync システムコールなどによりデータがフラッシュされたときに作成される．また，ユーザは明示的にチェックポイントを作成することができる．NILFS はユーザが指定した保護期間の間，チェックポイントを保存する．保護期間内のチェックポイントを一貫した状態に保つために，保護されたチェックポイントに含まれるブロックをクリーナが回収することは禁止される．そのために，ブロックが現在のファイルシステムで変更または削除されていたとしても，保護されたチェックポイントに含まれているならば，ブロックはライブであるとする．ライブブロックの判定方法についての詳細は 3.2 節で述べる．保護期間が過ぎると，チェックポイントに含まれるブロックはデッドになり，クリーナにより回収される．クリーナがブロックを回収すると，そのブロックを含んでいたチェックポイントは削除される．また，ユーザは（保護期間中であっても）明示的にチェックポイントを削除することができる．

チェックポイントを作成するとき，NILFS はファイルシステムの状態を保持するためのチェックポイントエントリを作成する．チェックポイントエントリはチェックポイント作成時の i ノードマップ，チェックポイントの作成時刻，チェックポイントがスナップショットかどうかを示すスナップショットフラグ，その他の管理情報を含んでいる．チェックポイントには一意なチェックポイント番号が割り当てられる．チェックポイント番号はチェックポイントを作成するたびに単調増加していく．また，チェックポイントエントリを管理するための**チェックポイントマップ**を用意する．チェックポイントマップはチェックポイントが作成されるときにディスクに書き込まれる．

チェックポイントはスナップショットにすることでユーザから利用可能となる．スナップショットは永続的なチェックポイントであり，保護期間が過ぎても削除されない．スナップショットを一貫した状態に保つために，スナップショットに含まれるブロックはライブであるとする．任意の時点のスナップショットはその時点のチェックポイントが存在する限り，さかのぼって作成することが可能である．従来のスナップショットファイルシステムとは異なり，NILFS ではデータの変更や削除に対してユーザが事前に準備することなく，以前のデータにアクセスすることが可能である．また，現在のファイルシステムの状態をスナップショットにすることもできる．スナップショットはマウント可能な読み出し専用ファイルシステムであり，標準的なファイルシステムインタフェースでアクセスすることができる．そのため，ファイルシステムに書き込みを行わない既存のアプリケーションは，変更なしにスナップショット上で動作する．チェックポイントからスナップショットを作成するには，チェックポイントエントリのスナップショットフラグを立て，スナップショットを管理しているスナップショットリストにチェックポイントエントリをつなぐだけであるため，高速に行うことができる．スナップショットは現在のファイルシステムと同じデータ構造を持つため，それらは同じ効率でアクセスすることができる．また，チェックポイントやスナップショットの存在は現在のファイルシステムの性能に影響しない．チェックポイント番号の最大値やチェックポイントマップの最大サイズなど，実装上の制限を除けば，チェックポイントやスナップショットの数に論理的な制限はなく，任意個のチェックポイントやスナップショットを作成することができる．複数のスナップショットを同時にマウントして，利用することもできる．

チェックポイントとスナップショットの作成例を**図 2** に示す．チェックポイント CP_1 は保護期間外にあり，それに含まれていたブロックがクリーナにより回収されたため，削除されている．スナップショット SS_2 は保護期間外にあるが，スナップショットであるため，永

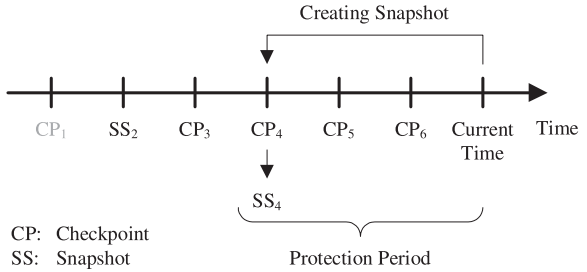

図 2　チェックポイントとスナップショット
Fig. 2　Checkpoints and snapshots.

続的に保存される．チェックポイント CP_3 は保護期間外にあるが，それに含まれているブロックがまだ 1 つも回収されていないため，削除されていない．チェックポイント CP_4 から CP_6 は保護期間内にある．スナップショット SS_4 はチェックポイント CP_4 が存在する限り，さかのぼって作成することができる．

3.2　ディスクアドレス変換

従来の LFS はブロックが現在のファイルシステムに含まれているかどうかを調べることによりライブブロックを判定する[17),19)]．クリーナはファイルの i ノード番号とブロックのオフセットを用いて，現在のファイルシステムにおけるブロックの位置を特定する．それが現在調べているブロックの位置と同じならば，ブロックはライブであり，そうでなければデッドである．NILFS では，ブロックが現在のファイルシステムで変更または削除されていたとしても，保護されたチェックポイントやスナップショットに含まれているならば，ライブでなければならないため，現在のファイルシステムに加えて，保護されたチェックポイントやスナップショットも調べる必要がある．しかし，従来の判定方法を保護されたチェックポイントやスナップショットに対して単純に適用することは，それらの数に比例した処理時間が必要になるため，現実的でない．

また，クリーナがライブブロックを移動させるとき，そのブロックを参照しているブロックを特定し，新しい位置を参照するように変更しなければならない．複数のブロックがライブブロックを参照しているならば，それらをすべて変更する必要がある．これは変更されたブロックを参照するブロックのさらなる変更を引き起こす．このようなクリーニングによる連鎖更新はディスクアクセスを著しく増加させ，性能低下の原因となる．

これらの問題を解決するために，NILFS では**ディスクアドレス変換**（Disk Address Translation（DAT））を導入する．**仮想ブロック番号**をブロックに割り当て，ブロックの参照に用いる．DAT は仮想ブロック番号から物理ディスクアドレスへのマッピングを管理する．仮想ブロック番号は新しいブロックを作成したとき，または既存ブロックを変更したときに割り当てられる．ブロックがディスクに書き込まれるとき，仮想ブロック番号に物理ディスクアドレスが対応付けられる．クリーナがライブブロックを移動させると，ブロックの新しい位置を指すようにマッピングの物理ディスクアドレスを変更するが，仮想ブロック番号は変更されない．そのため，ライブブロックを参照しているブロックを変更する必要はなく，クリーニングによる連鎖更新を防ぐことができる．クリーナがデッドブロックを回収すると，ブロックに割り当てられていた仮想ブロック番号は解放される．解放された仮想ブロック番号は再利用することができる．

また，DAT はブロックの**有効期間**を管理している．ブロックの有効期間はそのブロックを含むチェックポイントの範囲であり，開始チェックポイント番号と終了チェックポイント番号で表される．開始チェックポイント番号は，新しいブロックが作成されるか，または既存ブロックが変更され，それを含むチェックポイントが作成されたときのチェックポイント番号である．終了チェックポイント番号は，ブロックが変更または削除され，それを含むチェックポイントが作成されたときのチェックポイント番号である．有効期間が始まったときの終了チェックポイント番号は未定である．これはブロックが最新であり，現在のファイルシステムに含まれることを示す．ライブブロックはブロックの有効期間を用いて効率的に判定することができる．有効期間の終了チェックポイント番号が未定，または有効期間内に保護されたチェックポイントやスナップショットのチェックポイント番号が 1 つ以上存在するならば，ブロックはライブ，そうでなければデッドである．

DAT は効率的なクリーニングを可能にする．仮想ブロック番号はブロックの参照をディスク上の物理的位置から独立させ，クリーニングにともなう連鎖更新を防ぐ．また，ブロックの有効期間を用いてライブブロックを効率的に判定する．これは DAT を調べるだけであり，保護されたチェックポイントやスナップショットを網羅的に探索する必要はない．

4. 実　　装

NILFS は Linux ファイルシステムとして実装されており，カーネル 2.6 で動作する．サポートしているプロセッサアーキテクチャは x86，x86_64，PowerPC である．NILFS 本体はカーネルモジュールであり，クリーナやその他のユーティリティはユーザ空間で動作する．NILFS ライブラリは ioctl システムコールを用いて，NILFS カーネルモジュールとユーザ空間のユーティリティとの間のインタフェースを提供する．大規模で大量のファイルを扱うことができるように，NILFS は 64 ビットデータ構造を用いている．

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．ディスクは固定長のセグメントに分割され，セグメントはさらに 1 つ以上の部分セグメントに分割される．部分セグメントはセグメントサマリとデータブロックで構成される．セグメントサマリはクリーニングや復旧に使われる情報を保持しており，チェックサム，サイズ，書き込み時刻，次のセグメントへのポインタ，ファイル数とファイル情報の配列を含んでいる．ファイル情報は i ノード番号，書き込み時のチェックポイント番号，ファイルブロックの位置を特定するための情報を含んでいる．データブロックはファイルを構成するデータを含むブロックや間接ブロックである．

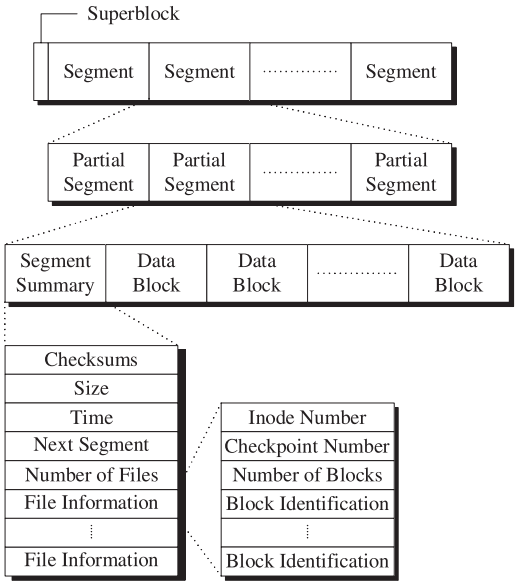

図 3　ディスクレイアウト
Fig. 3　Disk layout.

4.2　ファイル

ファイルは i ノードで管理される．i ノードはブロック単位のオフセットを仮想ブロック番号に変換するためのマッピングを保持している．様々なサイズのファイルを効率的に扱うために，ファイルサイズの変化に応じて異なるマッピングを動的に切り替える．小さなサイズのファイルはオフセットをインデックスとする仮想ブロック番号の配列で管理される．配列は i ノード内に保持される．現在の実装では，6 個の仮想ブロック番号を持つ．それ以上のサイズのファイルは B+木[4] で管理される．B+木のルートノードは i ノード内に保持され，他の間接ノードはブロックに保持される．

4.3　メタデータ

NILFS では，DAT，チェックポイントマップ，セグメント利用テーブル，i ノードマップをそれぞれ DAT ファイル，チェックポイントファイル，セグメント利用ファイル，i ノードファイルと呼ばれるファイルとして実装している．これらを総称してメタデータファイルと呼ぶ．メタデータファイルはファイルシステム内部のデータ構造を管理しており，ユーザ空間からは見えない特別なファイルである．**図 4** に示すように，ファイルシステム全体はメタデータによる木構造で表現される．DAT ファイル，チェックポイントファイル，セグメント利用ファイルは最新のバージョンのみが保持される．一方，i ノードファイルは保存されているチェックポイント（スナップショットを含む）に対応する複数のバージョンが保持される．古い DAT ファイル，チェックポイントファイル，セグメント利用ファイル，また，削除されたチェックポイントに対応する i ノードファイルはクリーニングされる．

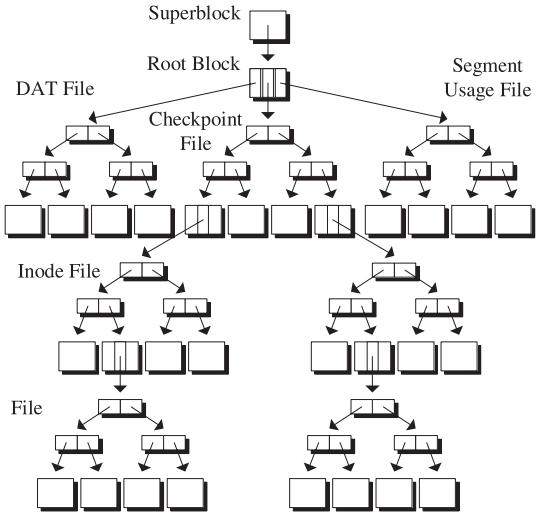

図 4　メタデータの階層構造
Fig. 4　Hierarchical structure of metadata.

ファイルシステムのメタデータをファイルとして実装している理由は以下の 3 つである．第 1 に，メタデータファイルは上書きされないため，システム障害後，通常ファイルと同様の方法で一貫した状態に戻すことができ，迅速な復旧が可能となる．第 2 に，ファイルシステムの使用状況に応じてメタデータファイルのサイズを増減させることが可能であり，i ノード数やチェックポイント数の制限をなくすことができる．第 3 に，ページキャッシュを用いることにより，メタデータファイルへのアクセスを高速化することができる．特に DAT は，仮想ブロック番号の割当てや解放，物理ディスクアドレスへの変換のために頻繁に参照，変更されるため，ページキャッシュの利用は有効である．

4.3.1　DAT ファイル

DAT ファイルは DAT エントリを保持している．DAT エントリは仮想ブロック番号で特定され，仮想ブロック番号に対する物理ディスクアドレスとブロックの有効期間を保持している．また，DAT ファイルはビットマップを用いて仮想ブロック番号の割当てや解放を管理している．

仮想ブロック番号は DAT ファイルを通じて物理ディスクアドレスに変換されるため，DAT ファイル自身のブロックは仮想ブロック番号を用いて参照することができない．そのため，DAT ファイルのブロックは物理ディスクアドレスを用いて参照する．DAT ファイルは最新のバージョンのみが保持されるため，DAT ファイルのライブブロックは，従来の LFS と同様に，ブロックが現在のファイルシステムに含まれているかどうかで判定できる．

ファイル以外の DAT の実装方法としては，DAT をディスクの予約領域に格納する方法がある[6]．この方法では，DAT ファイルのような仮想ブロック番号を使用しない特別なファイルは必要ないが，DAT の一貫性保持や高速なアクセスを実現するための機構が別途必要となる．また，DAT をあらかじめディスクの固定位置に割り当てるため，サイズを動的に変更することができない．

4.3.2　チェックポイントファイル

チェックポイントファイルはチェックポイント番号をインデックスとするチェックポイントエントリの配列を保持している．チェックポイントが作成されると，チェックポイントエントリが作成され，チェックポイントファイルの末尾に追加される．チェックポイントが削除されると，チェックポイントエントリが削除され，その位置はホールとなる．また，チェックポイントファイルはスナップショットリストを管理している．スナップショットリストはスナップショットになっているチェックポイントのチェックポイントエントリを保持しており，ライブブロックの判定に使われる．

4.3.3　セグメント利用ファイル

セグメント利用ファイルはセグメントごとにセグメント利用エントリを保持している．セグメント利用エントリはセグメントの書き込み時刻やセグメントが使用中かどうかを示すフラグなどを含んでいる．

4.3.4　i ノードファイル

i ノードファイルはメタデータファイル以外の通常ファイルの i ノードを保持している．i ノードは i ノード番号で特定される．また，i ノードファイルはビットマップを用いて i ノードの割当てや解放を管理している．チェックポイントが作成されるたびに i ノードファイルの新しいバージョンが作成され，その i ノードを含むチェックポイントエントリがチェックポイントファイルに保存される．

4.3.5　ルートブロック

DAT ファイル，チェックポイントファイル，セグメント利用ファイルの i ノードはルートブロックと呼ばれる特別なブロックに格納される．ルートブロックはチェックポイントが作成されるたびにディスクに書き込まれる．古いルートブロックはクリーニングされる．

4.3.6　スーパブロック

ファイルシステム全体はスーパブロックで管理される．スーパブロックは固定位置に配置され，ルートブロックを含む部分セグメントの位置を保持する．他のブロックと異なり，スーパブロックのみ上書きされる．

4.4　ページキャッシュ

Linux ファイルシステムはディスクアクセスを減らすためにページキャッシュを利用している．ファイルのデータブロックを保持するページはファイルごとに管理され，ファイルの i ノードとブロックのオフセットにより特定される．間接ブロックはオフセットを持たないため，すべてのファイルの間接ブロックは，ファイルが置かれているデバイスに対応するデバイスファイルによりまとめて管理され，デバイスファイルの i ノードとブロックの物理ディスクアドレスにより特定される．

データブロックのキャッシュは LFS でも正しく機能するが，間接ブロックのキャッシュは LFS には適用できない．LFS はブロックがディスクに書き込まれる直前に物理ディスクアドレスを割り当てるため，メモリ上で新しいブロックを作成したときや既存ブロックを変更したときには，インデックスとしての物理ディスクアドレスはまだ決定されていないからである．これに対処するために，BSD LFS はオフセットとして符号付き整数を使用し，ブロックの論理的位置に基づいて，0 または正数をデータブロックに，負数を間接ブロックに静的に割り当てている[19]．この方法は Fast File System[13] に基づくブロックマッピングではうまく動作する．しかし，NILFS ではブロックマッピングに B^+木を用いており，データブロックを挿入，削除するとき，木の均衡を保つために間接ブロックが分割，結合され，その論理的位置が動的に変化するため，この方法を適用することはできない．

この問題を解決するために，NILFS ではそれぞれのファイルごとに間接ブロックのキャッシュを用意する．間接ブロックを含むページはファイルの i ノードとブロックの仮想ブロック番号で特定される．仮想ブロック番号はメモリ上で新しいブロックを作成したときや既存ブロックを変更したときに割り当てられるため，キャッシュのインデックスとして用いるこ

とができる．間接ブロックのキャッシュにより，仮想ブロック番号から物理ディスクアドレスへの変換とディスクからの読み出しの両方を省略することが可能となる．4.5 節で述べるように，この間接ブロックのキャッシュはクリーニングのときにも使われる．

DAT ファイルも間接ブロックのキャッシュを用いるが，DAT ファイルのブロックは仮想ブロック番号を持たないため，他のインデックスが必要になる．そこで，物理ディスクアドレス空間を 2 つに分割する．物理ディスクアドレス空間の半分は実際の物理ディスクアドレスとする．もう半分は新しくメモリ上に作成された間接ブロックに一時的に割り当て，参照やキャッシュインデックスに使用する．このアドレスは間接ブロックがディスクに書き込まれるとき，実際の物理ディスクアドレスに修正される．

ファイルを変更するとき，変更されたデータブロックに新しい仮想ブロック番号が割り当てられるが，ファイルの i ノード番号やブロックのオフセットはそのままである．そのため，変更前のファイルを含むスナップショットと変更後のファイルを含むスナップショットを同時にマウントすると，それらのファイルのデータブロックがキャッシュ内で区別できなくなってしまう．これを解決するために，NILFS ではマウントされたスナップショットごとにメモリ上のスーパブロックをそれぞれ割り当てる．i ノードキャッシュ内で，i ノードはスーパブロックと i ノード番号で特定されるため，同じファイルで同じオフセットのブロックであっても，異なるスナップショットに含まれるならば，それぞれのスナップショットごとに別々のキャッシュに保持される．

4.5 クリーナ

クリーナはユーザ空間で動作するデーモンプロセスである．クリーナは設定ファイルで記述されたクリーニングポリシに基づいて，いつ，どのセグメントを，いくつ回収するかを決定する．クリーニングのポリシと機構の分離により，様々なユーザの要求や環境に応じたポリシを用意し，それらに柔軟に対応することが可能となる．現在の実装では，ファイルシステムの一貫性を保つため，クリーニングの間，ファイルシステムへの書き込み，チェックポイントやスナップショットの作成，削除はサスペンドされる．クリーナは以下のステップを実行する．

(1)　回収するセグメントを選択する．
(2)　選択されたセグメント内の各ブロックがライブかデッドかを判定する．
(3)　ライブブロックをページキャッシュに読み出す．
(4)　デッドブロックを含んでいるチェックポイントを削除する．
(5)　セグメントを未使用の状態にする．
(6)　ライブブロックに新しい物理ディスクアドレスを割り当てる．また，デッドブロックの仮想ブロック番号を解放する．
(7)　ライブブロックと変更されたメタデータファイルをディスクに書き込む．

各ステップの詳細を以下に述べる．

ステップ (1) では，回収するセグメントを選択するためにセグメント利用ファイルを読み出し，ポリシに基づいてセグメントの重要度を計算する．重要度の低いセグメントが選択される．セグメントの選択は性能上重要である[17]．現在の実装ではタイムスタンプポリシのみをサポートしている．タイムスタンプポリシは新しく書き込まれたセグメントに高い重要度を割り当てる．古いセグメントは変更または削除されたブロックを多く含む傾向にあるため，このポリシは多くの状況で適切に動作する．また，コスト–利益ポリシ[17] のような他のポリシの実装も検討している．

ステップ (2) では，選択されたセグメント内のライブブロックを判定するためにセグメントサマリを読み出す．ライブブロックの判定方法はそのブロックを含むファイルに依存する．メタデータファイル以外の通常ファイルや i ノードファイルでは，ブロックの有効期間を用いる．セグメントサマリのファイル情報に保持されている仮想ブロック番号を用いて DAT ファイル内の DAT エントリを特定し，ブロックの有効期間を取得する．そして，有効期間の終了チェックポイント番号が未定，または有効期間内に保護されたチェックポイントやスナップショットのチェックポイント番号があるかどうかでライブブロックを判定する．チェックポイントファイルやセグメント利用ファイルでもブロックの有効期間を用いるが，これらのファイルでは，変更または削除されたブロックはつねにデッドであるため，ブロックが最新であるかどうか，つまり，有効期間の終了チェックポイント番号が未定かどうかでライブブロックを判定する．DAT ファイルでは，セグメントサマリのファイル情報を用いて現在のファイルシステムにおけるブロックの物理ディスクアドレスを特定し，それが現在調べているブロックの位置と同じかどうかでライブブロックを判定する．

ステップ (3) では，ライブブロックを移動させるために，それらをページキャッシュに読み出す．このとき，ファイルのデータブロックとそれを変更したブロックがともにライブであると，これらは同じ i ノード番号とオフセットを持つため，キャッシュ内で区別できなくなる．これは 4.4 節で述べた問題と同じであるが，ブロックがマウントされたスナップショットに含まれるとは限らないため，同じ解決方法を用いることができない．そのため，ここではシャドウ i ノードを導入する．シャドウ i ノードはクリーニングの間だけ存在するメモリ上の特別な i ノードであり，ファイルごとにライブブロックをキャッシュするために

使われる．セグメントサマリのファイル情報はファイルの i ノード番号とファイルが書き込まれたときのチェックポイント番号を保持しており，シャドウ i ノードはこれらを用いて特定される．通常の i ノードの代わりにシャドウ i ノードを用いることを除いて，4.4 節で述べた方法と同様の方法でデータブロックや間接ブロックをキャッシュする．

ステップ (4) では，デッドブロックの有効期間内に作成されたチェックポイントのチェックポイントエントリを削除する．削除されたチェックポイントエントリの位置はホールとなる．

ステップ (5) では，回収されたセグメントのセグメント利用エントリを未使用の状態にする．このセグメントはクリーニングが完了すると利用可能になる．

ステップ (6) では，ライブブロックの移動先の物理ディスクアドレスを決定し，DAT エントリを変更する．また，デッドブロックの DAT エントリを解放する．これらの変更はクリーニングの完了後に有効となる．そのため，DAT エントリの変更や解放は DAT ファイルのシャドウ i ノード上で行い，クリーニングの最後で本来の DAT ファイルの i ノードに反映させる．

ステップ (7) では，キャッシュに読み出したライブブロックと変更したメタデータファイルをディスクに書き込む．書き込みが完了すると，クリーナはさらにセグメントを回収するように指示されるまでスリープする．

4.6　ファイルシステムの復旧

ファイルシステムの復旧は次のステップで行われる．

(1)　最新のチェックポイントを用いてファイルシステムのデータ構造を初期化する．

(2)　チェックポイント以降の変更をファイルシステムに反映させる．

ステップ (1) では，スーパブロックが指している部分セグメントから，セグメントサマリ内の次のセグメントへのポインタを順次たどって最新のルートブロックを特定し，その時点のチェックポイントを用いてファイルシステムのデータ構造を初期化する．スーパブロック以外のデータは上書きされないため，ルートブロック以下のデータ構造は一貫した状態に保たれている．また，システム障害に備えてスーパブロックの複製を作成しておき（現在は未実装），チェックサムを用いてスーパブロックの有効性を確認する．そのため，チェックポイントの作成中に障害が発生しても，システムが矛盾した状態になることはない．

ステップ (2) では，チェックポイント以降に書き込まれた有効な部分セグメントにおける変更をファイルシステムに適用し，可能な限りファイルシステムをシステム障害発生直前の状態まで復旧させる．部分セグメントの有効性はセグメントサマリ内のチェックサムを用いて確認する．論理的に 1 つの操作（ディレクトリ操作）が複数の部分セグメントにまたがるときは，その操作を構成するすべての部分セグメントが有効なときのみ復旧させる．また，チェックポイント以降に書き込まれた部分セグメントを順次読み出すために，セグメントサマリ内の次のセグメントへのポインタを用いる．

5. 評　　価

NILFS の性能を評価するためにベンチマークテストを実行した．評価実験に使用したシステムの構成を**表 1** に示す．ページキャッシュサイズ以上のファイルにおける測定を容易にするために，メモリサイズはカーネルのブートパラメータで制限した．測定されるファイルシステムは OS とは独立したディスクに構築した．測定は 5 回行い，平均値を計算した．

5.1　読み書き性能

NILFS の基本的なファイルシステム性能を測定するために IOzone[9] を実行し，Linux の標準ファイルシステムである Ext3[25] と比較した．読み出し，再読み出し，ランダム読み出し，書き込み，再書き込み，ランダム書き込みの各テストを実行した．ファイルサイズは 1 MB から 1,024 MB，レコードサイズは 4 KB と 64 KB である．測定には fsync によるフラッシュが含まれる．NILFS ではブロックサイズは 4 KB，セグメントサイズは 8 MB である．チェックポイントの保護期間は各テストの実行時間よりも長くなるように設定した．これにより，テスト中に作成されたチェックポイントはすべて保護される．スナップショットは作成していない．デッドブロックが存在しないため，クリーナは動作しない．Ext3 ではブロックサイズは 4 KB であり，ジャーナリングは ordered モードを使用した．

読み出し，再読み出し，ランダム読み出しのテスト結果をそれぞれ**図 5**，**図 6**，**図 7** に示す．読み出しテストでは，32 MB 以下のファイルのとき，NILFS は Ext3 よりも高速であり，64 MB 以上のファイルのとき，Ext3 よりも低速である．再読み出しテストでは，64 MB 以下のファイルのとき，NILFS は Ext3 とほぼ同じ性能であり，128 MB 以上のファイルの

表 1　評価実験に使用したシステムの構成
Table 1　System configuration used for evaluation experiments.

CPU	Intel Core 2 Duo E6700 2.66 GHz
Chipset	Intel 975X Express
Memory	256 MB PC2-5300 DDR-2 SDRAM
HDD	80 GB Serial ATA 7200 rpm 8 MB Cache (read/write cache enabled) × 2
Linux	Fedora 9 x86_64 (kernel 2.6.25.9)
NILFS	2.0.3

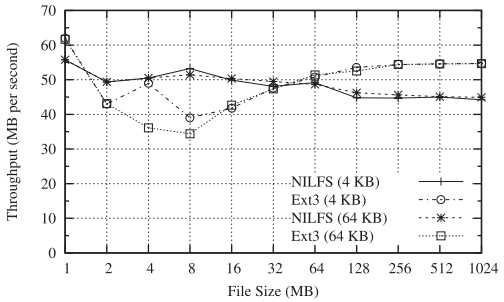

図 5　読み出し性能
Fig. 5　Read performance.

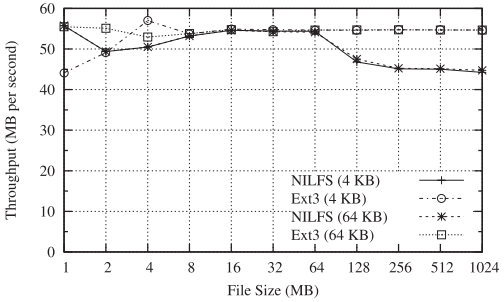

図 6　再読み出し性能
Fig. 6　Reread performance.

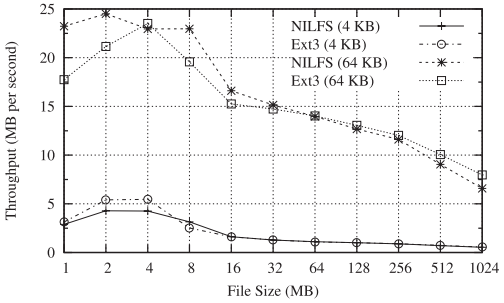

図 7　ランダム読み出し性能
Fig. 7　Random read performance.

とき，Ext3 よりも低速である．ランダム読み出しテストでは，4 MB 以下のファイルを除いて，NILFS は Ext3 とほぼ同じ性能である．読み出しテストや再読み出しテストでは，大きなファイルにおいて，NILFS の性能が低下している．これはブロックの断片化によるものである．NILFS はデータを上書きしないため，ブロックの断片化が起こりやすく，ディスクシークが増加してしまっている．ブロックの再配置による断片化の軽減やブロックの先読み方法の改良などによる読み出し性能の向上は今後の課題である．

書き込み，再書き込み，ランダム書き込みのテスト結果をそれぞれ**図 8**，**図 9**，**図 10** に示す．書き込みテストでは，4 MB のファイルとレコードサイズが 4 KB でファイルサイズが 8 MB のファイルを除いて，NILFS は Ext3 よりも高速である．再書き込みテストでは，8 MB 以下のファイルを除いて，NILFS は Ext3 とほぼ同じ性能である．ランダム書き込みテストでは，レコードサイズが 64 KB でファイルサイズが 8 MB 以下のファイルを除いて，NILFS は Ext3 よりも高速である．NILFS では，セグメントサマリやメタデータファイルのために書き込むデータ量が増加するが，複数の書き込みを単一の連続書き込みに変換するため，全体として書き込み性能が高くなっている．

5.2　クリーニングオーバヘッド

LFS では，ディスクフルにならないように定期的にディスクをクリーニングしなくてはならない．クリーナは LFS の性能に大きく影響する[20]．クリーナによる性能への影響を評価するために，クリーニング周期あたりの回収セグメント数を変化させたときの書き込み性能を測定した．測定では 512 MB のファイルに対してランダム書き込みを行った．レコードサイズは 4 KB と 64 KB である．チェックポイントの保護期間は 1 秒で，スナップショットは存在しない．そのため，変更されたブロックはすぐにデッドになる．ランダム書き込みにより，セグメント内のライブブロックとデッドブロックの割合はセグメントによって様々になる．測定結果を**図 11** に示す．クリーニングを行わなかったときの性能に対して，1 セグメント/秒の速度でクリーニングを行ったときの性能は約 58%，2 セグメント/秒では約 51%に低下している．効率的なクリーニングの実現は今後の重要な課題である．そのため，クリーニングオーバヘッドを小さくするセグメント選択ポリシの実装を検討している．また，現在の実装では，ユーザが使用状況に合わせてクリーニング速度を設定する必要があるため，ディスクのアクティビティや空き容量の変化に応じてクリーニング速度を動的に制御する方法[2] について検討している．

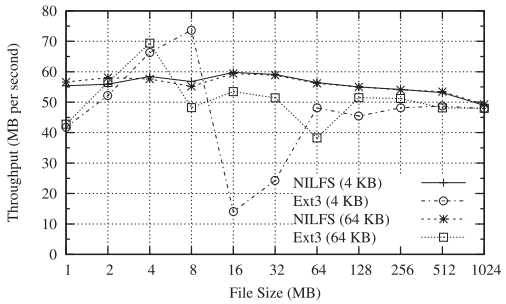

図 8　書き込み性能
Fig. 8　Write performance.

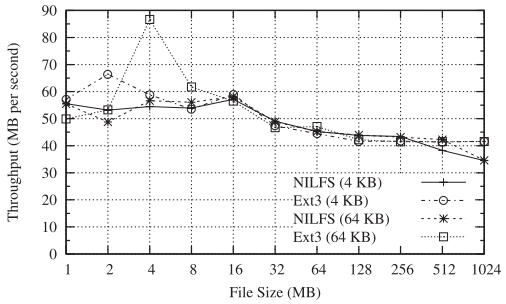

図 9　再書き込み性能
Fig. 9　Rewrite performance.

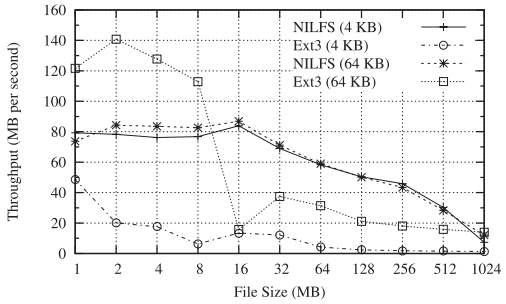

図 10　ランダム書き込み性能
Fig. 10　Random write performance.

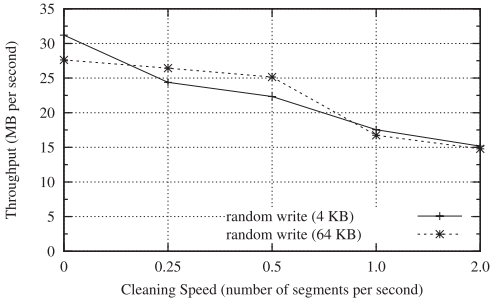

図 11　クリーニングオーバヘッド
Fig. 11　Cleaning overhead.

6. 関 連 研 究

Sprite LFS [17] は Sprite OS 上の LFS であり，ディスクシークの除去により書き込み性能を向上させることを目的としている．BSD LFS [19] は Sprite LFS を改良し，堅牢性の向上や BSD UNIX への統合を図っている．Sprite LFS や BSD LFS は周期的にチェックポイントを作成するが，それらをスナップショットとして利用する方法は提供されていない．

Spiralog [11] は OpenVMS 上の LFS であり，オンラインバックアップのためのスナップショットを提供している．しかし，スナップショットはバックアップと連携しており，バックアップ処理の間しか存在できない．

LFS with a Garbage Collection [10] は Linux 上の LFS であり，読み出し専用マウント可能なスナップショットをサポートしている．しかし，スナップショットはユーザが要求したときに作成されるため，任意の時点のスナップショットをさかのぼって作成することはできない．また，作成できるスナップショットは 1 つに限られる．さらに，スナップショットを一貫した状態に保つために，スナップショットがアンマウントされるまで，スナップショットのデータを含んでいないセグメントであっても，セグメントがスナップショットよりも古いならば，クリーナはそのセグメントを回収することができない．

Logical Disk（LD）[6] はディスクへの抽象インタフェースを定義している．LFS に基づいた LD の実装（LLD）では，仮想ブロック番号を用いることにより，ブロックの変更や移動にともなう連鎖更新を防いでいる．しかし，LLD はスナップショットをサポートしておらず，ライブブロックの判定にブロックの有効期間は用いられていない．また，ブロック番

号変換マップはディスクの予約領域に保存されており，システム障害後に変換マップを再構築するためには，すべてのセグメントサマリを読み出さなければならないため，復旧に時間がかかる．

Write Anywhere File Layout（WAFL）[8]，Ext3cow [16]，Zettabyte File System（ZFS）[24] などのファイルシステムはスナップショットをサポートしている．しかし，スナップショットはユーザからの要求時またはあらかじめスケジュールされた時点に作成されるため，任意の時点のスナップショットを作成することはできない．また，WAFL では，作成できるスナップショット数は 255 個に限られる．Ext3cow や ZFS などの多数のスナップショットを作成できるファイルシステムでは，継続的にスナップショットを作成し，一定期間後にそれらを削除する機構を別途用意することにより，NILFS における保護されたチェックポイントと同様のデータ保護を実現することができる．しかし，この方法では，保護されたチェックポイントを実現するためにスナップショットを用いており，それらは一定期間後に削除されてしまうため，ある時点のファイルシステムの状態を永続的に保存することができない．

Elephant [18] や Versionfs [14] は保存ポリシに基づいてファイルの重要なバージョンを自動的に保存する．ファイルごとに保存ポリシを設定することができ，バージョニングを柔軟に制御することができる．しかし，ファイルの新しいバージョンはファイルのオープン/クローズセッションで定義され，セッション内の変更は 1 つのバージョンに集約されるため，その間の変更を元に戻すことはできない．

Comprehensive Versioning File System（CVFS）[22] は侵入解析のためにすべてのファイルのすべてのバージョンを保存する．ファイルの変更ごとに新しいバージョンが作成され，保存期間が過ぎると削除される．CVFS の目的は必要なディスクスペースを最小化し，履歴の保存期間を長くすることである．そのため，特定のバージョンを長期間保存することは想定されていない．また，以前のバージョンを再構築するためには，変更数に比例してオーバヘッドがかかる．

Wayback [5] はユーザレベルのバージョニングファイルシステムである．ファイルの書き込みごとに取り消しログを記録し，ファイルの新しいバージョンを自動的に作成する．取り消しログを適用することにより，ファイルを以前のバージョンに戻すことができる．しかし，要求されるバージョンに到達するまで，取り消しログを順次適用していく必要があるため，古いバージョンほど復旧させるのに時間がかかる．また，ユーザレベルでの実装にともなう性能上のオーバヘッドが大きい．

CDP [23] はすべてのデータの変更を主ストレージとは別のストレージに連続的に保存することにより，ストレージの状態を任意の過去の時点に復旧させることを可能にしている．主ストレージに障害が発生したときには，変更履歴を用いてその直前の状態を再構築することができるため，データの損失を最小限に抑えることができる．しかし，変更履歴の保存期間が過ぎるとそれらは順次削除されるため，ある時点のストレージの状態をスナップショットとして長期間保存しておくことはできない．つねにすべてのデータの変更履歴を保存し続けるため，復旧すべき時点が長期間にわたる場合，大容量ストレージが必要となる．

7. ま と め

本論文では，ログ構造化ファイルシステム NILFS の設計と実装について述べた．NILFS は任意の時点におけるスナップショットの作成を可能にするとともに，DAT による効率的なクリーニングを実現した．また，評価実験により，NILFS は Ext3 と比べて遜色ない性能を有することを示した．今後は，読み出しや書き込みの性能向上やクリーナの改良を進める予定である．NILFS は http://www.nilfs.org/から入手可能である．

参 考 文 献

1) Azagury, A., Factor, M.E., Satran, J. and Micka, W.: Point-in-Time Copy: Yesterday, Today and Tomorrow, *Proc. 10th Goddard Conference on Mass Storage Systems and Technologies*, pp.259–270 (2002).
2) Blackwell, T., Harris, J. and Seltzer, M.: Heuristic Cleaning Algorithms in Log-Structured File Systems, *Proc. USENIX 1995 Technical Conference*, pp.277–288 (1995).
3) Chervenak, A.L., Vellanki, V. and Kurmas, Z.: Protecting File Systems: A Survey of Backup Techniques, *Proc. Joint NASA and IEEE Mass Storage Conference*, pp.17–31 (1998).
4) Comer, D.: The Ubiquitous B-tree, *ACM Computing Survey*, Vol.11, No.2, pp.121–138 (1979).
5) Cornel, B., Dinda, P.A. and Bustamante, F.E.: Wayback: A User-level Versioning File System for Linux, *Proc. USENIX 2004 Annual Technical Conference*, pp.19–28 (2004).
6) de Jonge, W., Kaashoek, M.F. and Hsieh, W.C.: The Logical Disk: A New Approach to Improving File Systems, *Proc. 14th ACM Symposium on Operating Systems Principles*, pp.15–28 (1993).
7) Ganger, G.R., McKusick, M.K., Soules, C.A.N. and Patt, Y.N.: Soft Updates: A

Solution to the Metadata Update Problem in File Systems, *ACM Trans. Computer Systems*, Vol.18, No.2, pp.127–153 (2000).
8) Hitz, D., Lau, J. and Malcolm, M.: File System Design for an NFS File Server Appliance, Technical Report 3002, Network Appliance (2001).
9) IOzone Filesystem Benchmark. http://www.iozone.org/
10) Jambor, M., Hruby, T., Taus, J., Krchak, K. and Holub, V.: Implementation of a Linux Log-Structured File System with a Garbage Collector, *ACM SIGOPS Operating System Review*, Vol.41, No.1, pp.24–32 (2007).
11) Johnson, J.E. and Laing, W.A.: Overview of the Spiralog File System, *Digital Technical Journal*, Vol.8, No.2, pp.5–14 (1996).
12) Konishi, R., Amagai, Y., Sato, K., Hifumi, H., Kihara, S. and Moriai, S.: The Linux Implementation of a Log-structured File System, *ACM SIGOPS Operating Systems Review*, Vol.40, No.3, pp.102–107 (2006).
13) McKusick, M.K., Joy, W., Leffler, S. and Fabry, R.S.: A Fast File System for UNIX, *ACM Trans. Computer Systems*, Vol.2, No.3, pp.181–197 (1984).
14) Muniswamy-Reddy, K.-K., Wright, C.P., Himmer, A. and Zadok, E.: A Versatile and User-Oriented Versioning File System, *Proc. 3rd USENIX Conference on File and Storage Technologies*, pp.115–128 (2004).
15) Ousterhout, J. and Douglis, F.: Beating the I/O Bottleneck: A Case for Log-Structured File Systems, *ACM SIGOPS Operating Systems Review*, Vol.23, No.1, pp.11–28 (1989).
16) Peterson, Z.N.J. and Burns, R.: Ext3cow: A Time-Shifting File System for Regulatory Compliance, *ACM Trans. Storage*, Vol.1, No.2, pp.190–212 (2005).
17) Rosenblum, M. and Ousterhout, J.K.: The Design and Implementation of a Log-Structured File System, *ACM Trans. Computer Systems*, Vol.10, No.1, pp.26–52 (1992).
18) Santry, D.S., Feeley, M.J., Hutchinson, N.C., Veitch, A.C., Carton, R.W. and Ofir, J.: Deciding when to forget in the Elephant file system, *Proc. 17th ACM Symposium on Operating Systems Principles*, pp.110–123 (1999).
19) Seltzer, M., Bostic, K., McKusick, M.K. and Staelin, C.: An Implementation of a Log-Structured File System for UNIX, *Proc. USENIX Winter 1993 Conference*, pp.307–326 (1993).
20) Seltzer, M., Smith, K.A., Balakrishnan, H., Chang, J., McMains, S. and Padmanabhan, V.: File System Logging Versus Clustering: A Performance Comparison, *Proc. USENIX 1995 Technical Conference*, pp.249–264 (1995).
21) Sillicon Graphics, Inc.: XFS: A high-performance journaling filesystem. http://oss.sgi.com/projects/xfs/
22) Soules, C.A.N., Goodson, G.R., Strunk, J.D. and Ganger, G.R.: Metadata Efficiency in Versioning File Systems, *Proc. 2nd USENIX Conference on File and Storage Technologies*, pp.43–58 (2003).
23) Storage Networking Industry Association: *CDP Buyer's Guide 2nd edition* (2006).
24) Sun Microsystems, Inc.: *Solaris ZFS Administration Guide* (2008).
25) Tweedie, S.: EXT3, Journaling Filesystem (2000). http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html

(平成 20 年 7 月 23 日受付)
(平成 20 年 11 月 2 日採録)

佐藤　孝治（正会員）

1989 年慶應義塾大学理工学部数理科学科卒業．1991 年同大学大学院理工学研究科計算機科学専攻修士課程修了．同年日本電信電話株式会社入社．以来，分散システム，マルチメディアシステム，オペレーティングシステムの研究に従事．現在，NTT サイバースペース研究所に所属．日本ソフトウェア科学会会員．

小西　隆介（正会員）

1991 年東北大学工学部情報工学科卒業．1993 年同大学大学院修士課程修了．同年日本電信電話株式会社に入社．CTRON オペレーティングシステムの開発実用化に従事．その後，動的再構成可能 LSI の研究開発等を経て，2004 年より NTT サイバースペース研究所にて，NILFS の開発と Linux 開発コミュニティへの提案活動を推進．現在に至る．

木原　誠司（正会員）

1990 年東京工業大学理学部情報科学科卒業．1992 年同大学大学院修士課程修了．同年日本電信電話株式会社に入社，オペレーティングシステムとコンピュータネットワークの研究開発に従事．現在，NTT サイバースペース研究所主幹研究員．OS カーネルと仮想化技術の研究開発に従事．ACM 会員．

天海　良治（正会員）

1983 年電気通信大学電気通信学部計算機科学科卒業．1985 年同大学大学院修士課程修了．同年日本電信電話株式会社入社．以来，プログラミングパラダイム，計算機ネットワーク，ストレージシステムの研究に従事．現在 NTT サイバースペース研究所主任研究員．日本ソフトウェア科学会会員．

盛合　　敏（正会員）

1983 年東北大学工学部電気工学科卒業．1988 年同大学大学院博士課程（情報工学専攻）修了．工学博士．同年日本電信電話株式会社入社．入社以来，プロトコル処理，インターネットシステム運用，分散 OS，リアルタイム OS，セキュア OS の研究に従事．2001〜2004 年株式会社ぷららネットワークス．2004 年より NTT サイバースペース研究所主幹研究員．高信頼 OS カーネル，仮想マシン，ユビキタスコンピューティング基盤の研究に従事．日本ソフトウェア科学会，USENIX 各会員．