修 士 論 文 の 和 文 要 旨

研究科・専攻	大学院　情報理工学研究科　　情報・ネットワーク工学専攻　博士前期課程		
氏　　名	青山　航	学籍番号	1631002
論 文 題 目	ディレクトリ毎にジャーナリングモードを設定可能なファイルシステム		

要　旨

ファイルの書き込み中にオペレーティング・システムが異常終了すると，ファイルシステムが破損してしまう可能性がある．従来のファイルシステムでは，ジャーナリングと呼ばれる技術で，ファイルシステムの信頼性を向上している．ジャーナリングとは，ファイルの作成・更新などファイルシステムを変更する際に，ログにその変更を記録する手法である．ジャーナリングの信頼性とファイルシステムの速度は，トレードオフの関係にある．そのため，既存のシステムは，ジャーナリングの方法(モード)を，信頼性を重視したもの，速度を重視したものなど，何種類か提供している．しかし，ジャーナリングモードはファイルシステムに一つしか設定できないため，ファイルシステム内のファイル毎に異なる特性に合わせることができない．

本研究では，この問題を解決するために，ジャーナリングモードをディレクトリ毎に設定可能なファイルシステム dajFS を提案し，Linux の ext3 ファイルシステムを拡張して実現する．dajFS を用いれば，ジャーナリングモードの設定単位を従来より細かくすることができ，ユーザに大きな負担を掛けずに，ファイルの特性に合ったモードを設定することが可能となる．評価として，SQLite が生成する一時ファイルの保存先ディレクトリのジャーナリングモードを変更する実験を行ったところ，システムの信頼性を損なわずに実行時間を最大で 36%削減することができた．

平成 29 年度修士論文

ディレクトリ毎にジャーナリングモードを設定可能なファイルシステム

電気通信大学
大学院情報理工学研究科
情報・ネットワーク工学専攻
コンピュータサイエンスプログラム

学籍番号　　: 1631002
氏名　　: 青山 航
主任指導教員 : 岩崎 英哉 教授
指導教員　　: 中山 泰一 准教授
提出日　　: 2017 年 1 月 29 日

要旨

ファイルの書き込み中にオペレーティング・システムが異常終了すると，ファイルシステムが破損してしまう可能性がある．従来のファイルシステムでは，ジャーナリングと呼ばれる技術で，ファイルシステムの信頼性を向上している．ジャーナリングとは，ファイルの作成・更新などファイルシステムを変更する際に，ログにその変更を記録する手法である．ジャーナリングの信頼性とファイルシステムの速度は，トレードオフの関係にある．そのため，既存のシステムは，ジャーナリングの方法 (モード) を，信頼性を重視したもの，速度を重視したものなど，何種類か提供している．しかし，ジャーナリングモードはファイルシステムに一つしか設定できないため，ファイルシステム内のファイル毎に異なる特性に合わせることができない．

本研究では，この問題を解決するために，ジャーナリングモードをディレクトリ毎に設定可能なファイルシステム dajFS を提案し，Linux の ext3 ファイルシステムを拡張して実現する．dajFS を用いれば，ジャーナリングモードの設定単位を従来より細かくすることができ，ユーザに大きな負担を掛けずに，ファイルの特性に合ったモードを設定することが可能となる．評価として，SQLite が生成する一時ファイルの保存先ディレクトリのジャーナリングモードを変更する実験を行ったところ，システムの信頼性を損なわずに実行時間を最大で 36% 削減することができた．

目次

1 はじめに

1.1 背景

ファイルシステムとは，ディスク上のデータを管理するオペレーティング・システム (OS) の一機構である．一般的なファイルシステムは，ディスクを一定長のブロックに分割し，実際のデータが格納されているブロック (実データブロック) と，i ノードやブロックの空き状況を示すビットマップが格納されているブロック (メタデータブロック) の両方を用いて管理する．

ユーザプログラムがファイルに書き込みを行うと，ファイルシステムは対応する複数のブロックに書き込みを行う．このとき，OS の異常終了などにより実データやメタデータの一部が失われてしまうと，ファイルシステムの管理状態が中間的な不完全状態に陥り (この状態を，整合性が損なわれていると呼ぶ)，ファイルシステムが破損してしまう可能性がある．ファイルシステムの整合性を保証するために，ファイルシステムの整合性検査 (fsck) [1] やジャーナリング [2]，soft-updates [3]，コピーオンライト [4] などの技術が提案されている．中でも，ジャーナリングは ext3 ファイルシステム (以下，ext3 と呼ぶ) [5]，JFS [6]，XFS [7]，ReiserFS [8]，NTFS [9] など様々なファイルシステムで用いられている．

ジャーナリングとは，ファイルの作成・更新などファイルシステムに対して変更 (トランザクション) を施す前に，変更履歴を記録する手法である．図 1.1 に，ジャーナリングを用いてファイルへ書き込みを行う様子を示す．ここで，D はファイルの実データ，M はファイルのメタデータを表す．ジャーナリングでは，ジャーナルという特殊なログファイルにファイルの変更内容 M と D を一旦記録し (図 1.1 (b))，記録の終了を確認した後に，保存領域の対象のファイルに書き込みを行い (図 1.1 (c))，ジャーナルから記録を削除する (図 1.1 (d))．OS が異常終了した場合，ジャーナルへの書き込みの状態を調べることによって，処理がどこまで進行したかがわ

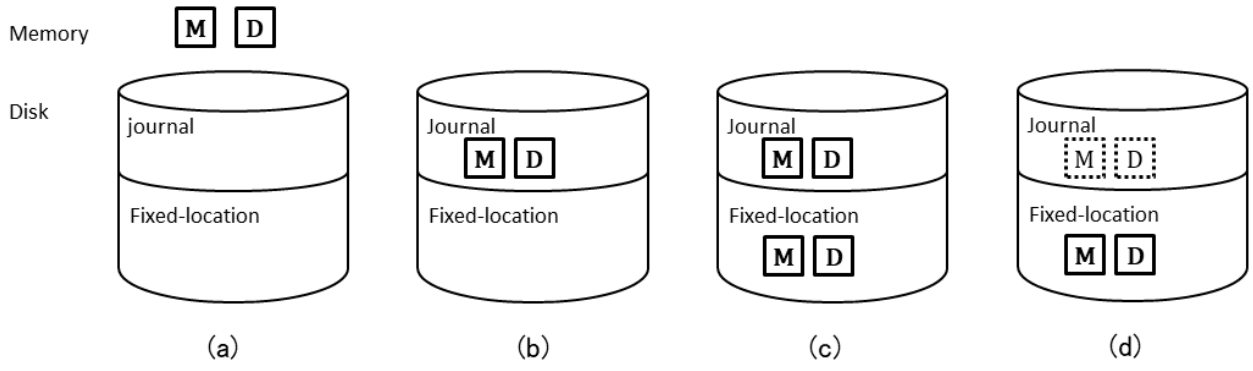


図 1.1　ジャーナリングを用いたファイル書き込み

かる．書き込みが完了していなければ，ファイルシステムに対する変更が行われる前の状態に保たれ，ジャーナルへの書き込みが完了していれば，保存領域のデータを更新後の状態にし，ファイルシステムの整合性が損なわれないようにする．

ext3 では，変更履歴に保存する対象や書き込みの順番を，ジャーナリングモードにより設定することができる．ジャーナリングモードとしては，信頼性を重視したもの，速度を重視したものなど，何種類か提供しており，重要なファイルには信頼性の高いモード，重要度があまり高くない一時ファイルには速度が速いモードというように，重要度等のファイルの特性に合ったモードを設定することが望まれる [10, 11]．しかし，既存の ext3 では，ファイルシステムに一つのジャーナリングモードしか設定できないため，ファイルシステム内のファイル毎に異なる特性に合わせることができない．

1.2 目的と方針

このような問題点を解決するため，本研究では，ジャーナリングモードをディレクトリ毎に設定することが可能なファイルシステム Directory-Adaptive Journaling FileSystem (dajFS) を提案する．このことにより，ジャーナリングモードの設定単位を従来より細かくすることができ，ユーザに大きな負担を掛けずに，ファイルの特性に概ね合ったモードを設定することが可能となる．

1.3 本論文の構成

本論文の構成は次の通りである．2 章で，ext3 とジャーナリングの処理について述べる．3 章では，既存研究の特徴と問題点を述べる．4 章では，本研究で提案するシステムの概要と動作の流れについて述べ，5 章ではシステムの実装について述べる．6 章で，評価を行い，最後に 7 章で本稿をまとめ，今後の課題を述べる．

2　ext3 ファイルシステム

2.1　ファイルシステムのレイアウト

ext3 は，ext2 ファイルシステムを拡張して開発されたジャーナリングファイルシステムである．ブロックデバイスにファイルシステムをマウントして利用する．

ext3 のレイアウトを図 2.1 に示す．ext3 では，ファイルシステムをブロックグループと呼ぶ均等な大きさの領域に分けて管理する．これは，FFS [12] の構造を基にしている．各ブロックグループには，i ノードビットマップ (IB)，データビットマップ (DB)，i ノードブロック (INODE) 及びデータブロック (DATA) がある．i ノードビットマップは，ブロックグループ内の空き i ノードブロックを管理するためのビットマップである．データビットマップは，ブロックグループ内の空きデータブロックを管理するためのビットマップである．i ノードブロックは，ファイルの i ノードが格納される領域である．データブロックは，実際のデータが格納される領域である．

ext3 では，ジャーナルをファイルシステム内のファイルとして保存するが，別のデバイスまたはパーティションに保存することもできる．本論文では，ジャーナルをファイルとして保存した場合のみを考える．ジャーナルに関するブロックは，後ほど説明する．

2.2　ファイルシステムのデータ構造

Linux では，異なるファイルシステムでも，共通したファイル操作をユーザに提供するために，共通ファイルモデルを導入している．共通ファイルモデルでは，ファイルシステム全体やファイ

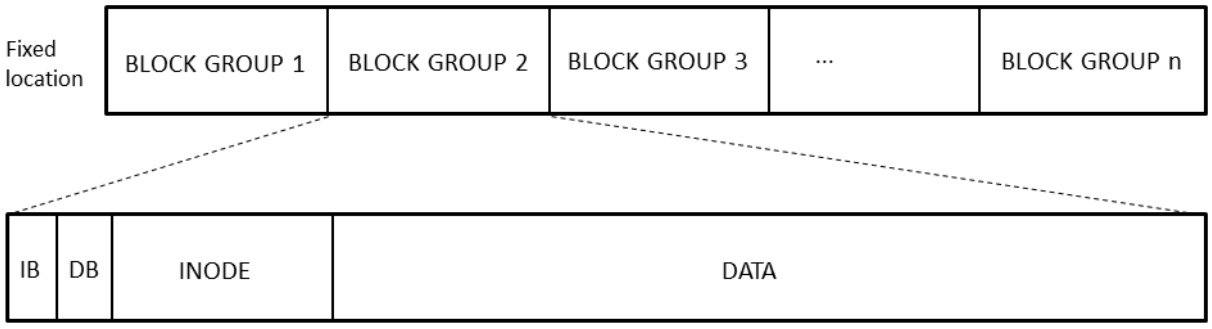


図 2.1　ext3 ファイルシステムのレイアウト

ルをオブジェクトとして扱う．ファイルシステムの設計者は，オブジェクトに read や ioctl といったファイル操作処理を設定する．

共通ファイルモデルのオブジェクトには，i ノードオブジェクト (struct inode)，ファイルオブジェクト (struct file)，アドレス空間オブジェクト (struct address_space) がある．i ノードオブジェクトは，ファイルをオブジェクトとしたものである．ここで Linux では，ディレクトリをファイルのリストの情報を持ったファイルとして扱う．

下に struct inode の定義を示す．

```
struct inode {
  umode_t      i_mode;
  kuid_t       i_uid;
  kgid_t       i_gid;
  unsigned int     i_flags;
  unsigned long    i_ino;
  unsigned long    i_state;
  const struct inode_operations *i_op;
  ...
};
```

i ノードオブジェクトは，ファイルの種類とアクセス権 (i_mode)，ファイル所有者のユーザ ID (i_uid)，グループ ID (i_gid)，ファイルの属性フラグ (i_flags)，i ノード番号 (i_ino)，i ノードの状態フラグ (i_state) や i ノード操作関数 (i_op) を持つ．ファイルの属性には，ファイルの更新時間を変更しないフラグや，ファイルを圧縮するフラグなどがある．ファイルの状態には，ファイルの実データがディスクに書き戻されていないフラグ I_DIRTY_PAGES や，ファイルのメタデータが書き戻されていないフラグ I_DIRTY_DATASYNC などがあり，どちらかのフラグが立っているファイルを「汚れている」と呼ぶ．ここで，これらのフラグを Dirty フラグと呼ぶ．i ノード操作関数には，ファイルを新規作成する create やシンボリックリンクの内容を読み込む readlink などがある．

次に struct file の定義を示す．

```
struct file {
  struct path   f_path;
  struct inode     *f_inode;
  atomic_long_t    f_count;
  loff_t       f_pos;
  const struct file_operations  *f_op;
  ...
}
```

ファイルオブジェクトは，open システムコールによってオープンされたファイルをオブジェクトとしたものである．ファイルオブジェクトは，ファイルのパスのキャッシュ (f_path)，ファイルオブジェクトに関連付けられている i ノードオブジェクト (f_inode)，ファイルオブジェクトの参照カウント (f_count)，現在のファイルの書き込み/読み込み位置 (f_pos) やファイル

操作関数 (i_op) を持つ．ファイル操作関数には，ファイルの読み込み read やファイルのパラメータを変更する ioctl などがある．

最後に，struct address_space の定義を示す．

```
struct address_space {
  struct inode    *host;
  unsigned long   nrpages;
  const struct address_space_operations *a_ops;
  ...
}
```

アドレス空間オブジェクトは，ページキャッシュ (後述) をオブジェクトとしたものである．アドレス空間オブジェクトは，このページキャッシュと対応している i ノードオブジェクト (host)，対応している i ノードオブジェクトが保有しているページキャッシュの総数 (nrpages)，アドレス空間操作関数 (a_ops) を持つ．アドレス空間操作関数には，ページキャッシュをディスクに書き込む writepage やページキャッシュを書き込み準備状態にする write_end などがある．

ext3 では，ディスク上の i ノードブロックを ext3_inode 構造体で扱っている．

```
struct ext3_inode {
  __le16  i_uid;
  __le16  i_gid;
  __le32  i_flags;
  __le32  i_block[EXT3_N_BLOCKS];
  ...
};
```

i ノードブロックは，ファイル所有者のユーザ ID(i_uid)，グループ ID(i_gid) などを 16 ビットで，ファイルの属性フラグ (i_flags) を 32 ビットで，実データへのポインタを 32 ビットでそれぞれ管理する．

Linux では，ディスク I/O を減らすために，ディスク上のデータをメモリにキャッシュする．これを，ディスクキャッシュと呼ぶ．ディスクキャッシュには，ファイルの実データをページ単位に分割したページキャッシュや，i ノードをキャッシュした i ノードキャッシュなどが存在する．Linux 2.6 以降では，i ノードオブジェクトを i ノードキャッシュと呼ぶ．i ノードブロックから i ノードキャッシュを取得には ext3_iget 関数を用い，i ノードキャッシュから i ノードブロックを更新するには ext3_do_update_inode 関数を用いる．

図 2.2 に，ディスクへの読み込みが発生した様子を示す．ここで，対象のファイルの i ノードは M_1，実データは D_1 から構成されているとする．カーネルは，対象のファイルがメモリにキャッシュされているかどうかを確認する (図 2.2 (a))．キャッシュされていなければ，ディスク内の i ノードを i ノードオブジェクトに，実データをページ単位に分割してページキャッシュに格納し (図 2.2 (b))，ユーザプロセスにデータを渡す (図 2.2 (c))．その後，同じファイルをアクセスすると，i ノードキャッシュとページキャッシュを用いてファイルを高速に読み込む．

Linux では，ファイルのオープン時に O_SYNC や O_DIRECT フラグを明示的に指定しない限り

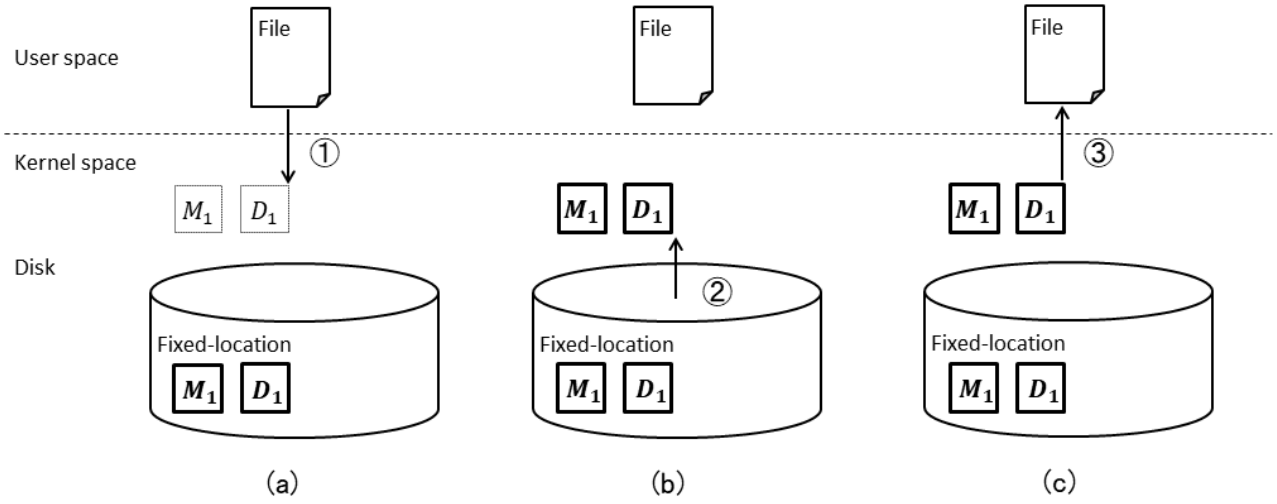


図 2.2　ファイルの読み込み (a) キャッシュミス (b) ディスクから対応するデータを読み込み (c) データをユーザプロセスに渡す

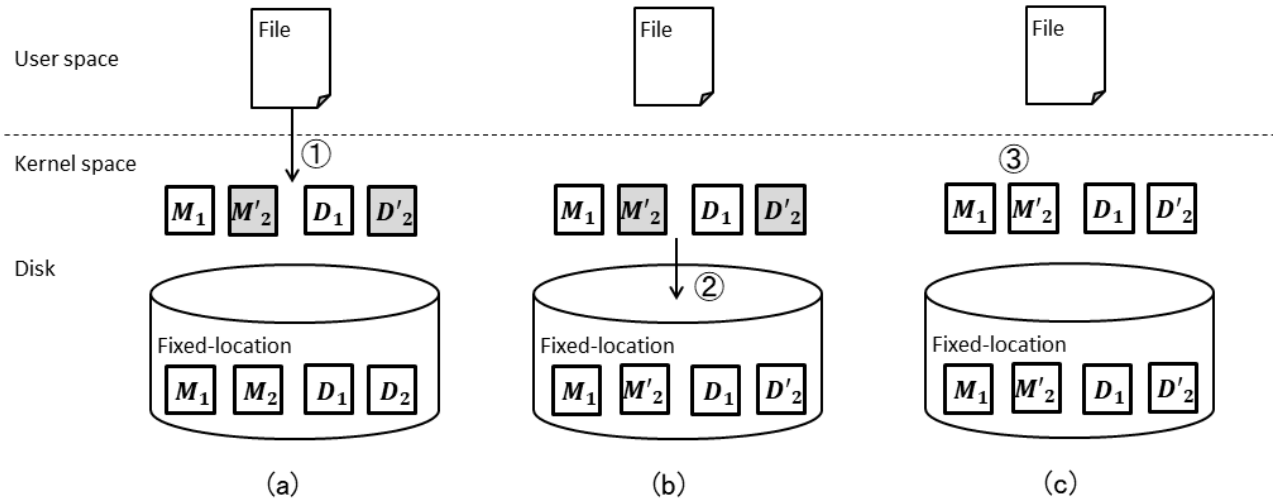


図 2.3　非同期書き込み．(a) write システムコールによる処理 (b) カーネルスレッドによる処理 (c) Dirty フラグのクリア

非同期 I/O となる．図 2.3 に，ファイルへの非同期書き込みを行う様子を示す．ここで，D は実データ，M はメタデータ，Dirty フラグが立っているデータを灰色で表す．非同期書き込みでは，ディスクキャッシュへの書き込みが完了すると，ユーザに書き込みが終了したことを通知する (図 2.3 (a))．この時，作成したページキャッシュおよび i ノードキャッシュに変更が加えられたことを表す Dirty フラグを立てる．その後，カーネルスレッド pdflush が，Dirty フラグを立っているデータのみディスクの保存領域に書き込む．(図 2.3 (b))．

2.3 ジャーナリング

ext3 は，Journaling Block Device (JBD) と呼ばれるカーネル組み込みのジャーナリング用のモジュールを利用している．JBD では，ジャーナルにデータを書き込む動作をコミット，コミットされたデータをディスクの保存領域に書き込む動作をチェックポイント，ジャーナルのデータを用いて整合性が損なわれているファイルシステムを復旧する操作をリカバリと呼ぶ．本論文でも，これらの用語を用いる．

2.3.1 ジャーナリングモード

ext3 では，ジャーナルに保存する変更履歴の種類や保存領域への書き込み順番をジャーナリングモードによって設定することができる．ジャーナリングモードでは，writeback，ordered，data の 3 種類がある．各ジャーナリングモードの動作の流れを図 2.4 に示す．

writeback モード (図 2.4 (a)) では，i ノードビットマップ，データビットマップ，i ノードブロック (以下，これら 3 つを合わせてメタデータと呼ぶ) をジャーナルに保存し，データブロック (以下，実データと呼ぶ) を保存領域に書き込む．このモードでは，ジャーナルと保存領域の書き込みの間の順番を強制されない．そのため，メタデータが有効でない実データを参照することがあるが，メタデータの整合性が損なわれることはない．

ordered モード (図 2.4 (b)) も同様に，メタデータをジャーナルに保存し，実データを保存領域に書き込む．しかし ordered モードでは，メタデータの書き込みは，実データの書き込みの後と順番を強制するため，メタデータは必ず有効な実データを参照する．

data モード (図 2.4 (c)) では，メタデータと実データの両方をジャーナルに保存する．そのため，メタデータと実データの整合性が損なわれることはない．

各ジャーナリングモードのオーバーヘッドと信頼性を，表 2.1 に示す．ここで，IB は i ノー

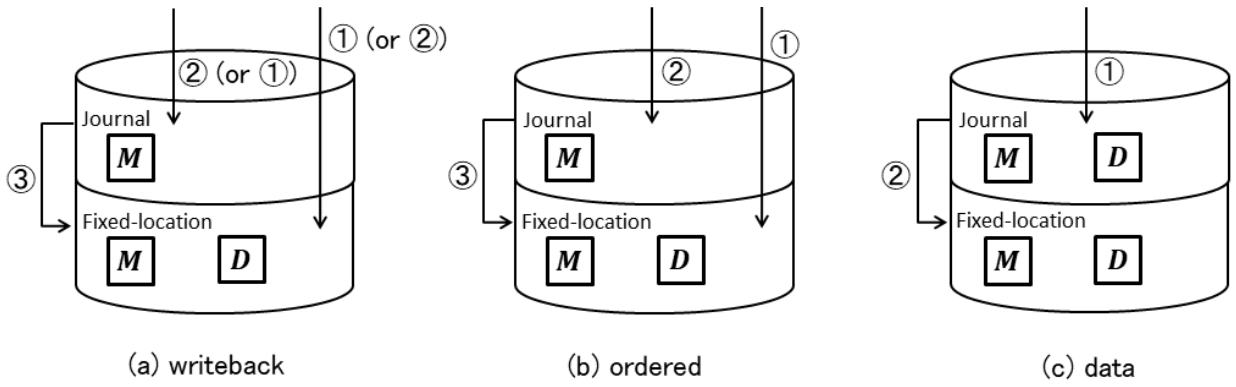


図 2.4　ext3 のジャーナリングモードにおける処理の流れ

表 2.1　ext3 のジャーナリングモード

モード	保存対象				保存領域への	オーバーヘッド	信頼性
	IB	DB	I	D	書き込み順番の強制		
writeback	✓	✓	✓		×	小	低
ordered	✓	✓	✓		$D \rightarrow M$	中	中
data	✓	✓	✓	✓	×	大	高

ドビットマップ，DB はデータビットマップ，I はファイルの i ノード，D はファイルの実データを表す．writeback モードは，メタデータが不正なデータを指す可能性があり信頼性は低いが，保存対象がメタデータのみでメタデータの書き込み順番の強制はないので，ディスク I/O 待ち時間が少なく，オーバーヘッドは小さい．ordered モードは，実データより前にメタデータがディスクに書き込まれることがないので，メタデータが不正なデータを指し示すことはない．しかし，実データの書き込みが完了するまでメタデータの書き込みを遅延させるため，writeback モードと比べてオーバーヘッドは大きい．data モードは，変更履歴を全て保存するため信頼度は最も高いが，同じデータをディスクに二回書き込む必要があるため，オーバーヘッドは大きい．

これらのことから，ジャーナリングの速度とファイルシステムの信頼性はトレードオフの関係にあるが，ジャーナリングモードはファイルシステム単位でしか設定できず，マウント時にモードが決定される．ジャーナリングの信頼性と速度を向上させる研究 [13, 14] はあるものの，ジャーナリングの設定粒度を変更することはできない．

2.3.2　ジャーナルのデータ構造

ext3 のジャーナル (図 2.1 の灰色部分) は，図 2.5 のような構造になっている．ジャーナルは，ジャーナルスーパーブロック (JS) と複数のトランザクションから構成され，循環バッファ

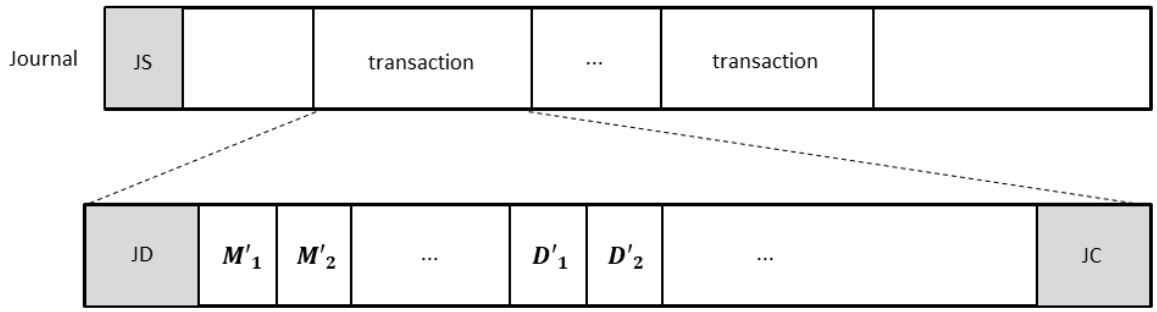


図 2.5　ジャーナルの構造

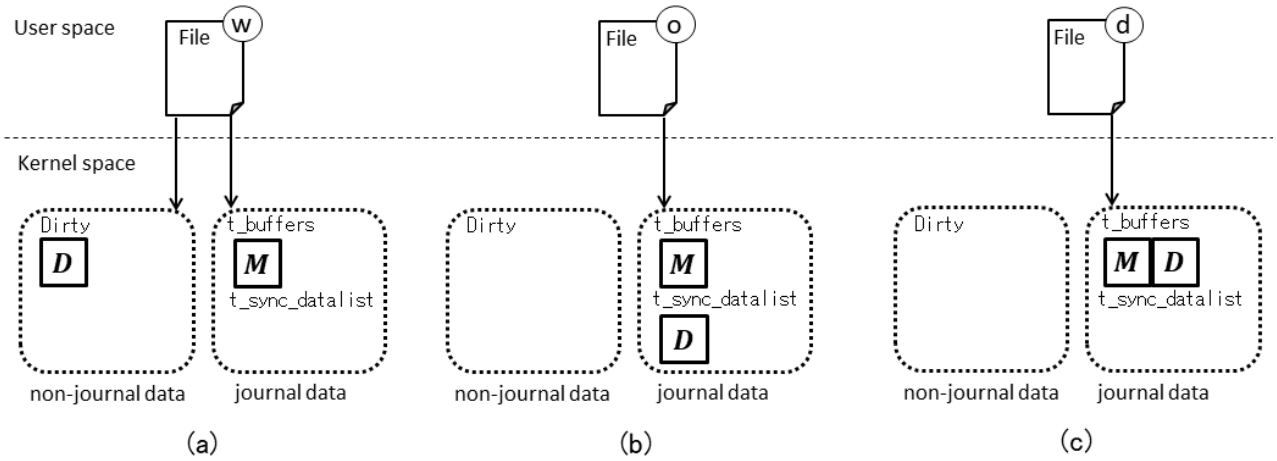


図 2.6　各モードのデータ構造

として使用される．ジャーナルスーパーブロックは，ジャーナルの大きさやトランザクションの開始位置などといったジャーナル全体の管理情報を保持する．トランザクションは，複数の変更履歴 (M' や D') をディスクリプタブロック (JD) とコミットブロック (JC) の間に挿入した構造となっている．各トランザクションには，シーケンシャルな番号 (トランザクション ID) が付与される．ジャーナリングでは，ジャーナルスーパーブロックが現在実行中のトランザクションのトランザクション ID を管理し，順番通りにトランザクションを処理する．ディスクリプタブロックは，トランザクション内の変更履歴に関する情報 (保存領域のブロック番号) を保持する．コミットブロックは，トランザクションの終わりを表す．

ext3 のトランザクションは，主に `t_buffers` と `t_sync_datalist` と呼ばれる二つの双方向リストでデータを管理する．`t_buffers` は，ジャーナルに書き込むデータを繋げた双方向リストである．JBD は，この双方向リストにデータを繋げるための API として，`journal_dirty_metadata` 関数をファイルシステムに提供している．`t_sync_datalist` は，保存領域に書き込むデータを繋げた双方向リストである．JBD は，この双方向リストにデータを繋げるための API として，`journal_dirty_data` 関数をファイルシステムに提供している．

各モードにおけるカーネル内部のデータ構造を図 2.6 に示す．ここで，図中の左の点線の枠はジャーナリングとは関係のないデータ構造 (non-journal data)，右の点線の枠はジャーナリングのデータ構造 (journal data) を表す．writeback モードは，ジャーナリングを行う M を `t_buffers` リストへ，保存領域へ書き込む対象の D をジャーナリングとは別のデータ構造の `Dirty` リストに追加する．`Dirty` リストに追加されたデータは，従来のファイルシステムと同様にカーネルスレッド pdflush がディスクへの書き込みを行う．ordered モードでは，ジャーナリングを行う M を `t_buffers` リストへ，ジャーナリングをしない D を `t_sync_datalist` リストに追加する．`t_buffers` リストと `t_sync_datalist` リストに追加されたデータは，カーネルスレッド kjournald がディスクへの書き込みを行う．ジャーナリング対象ではない D もジャー

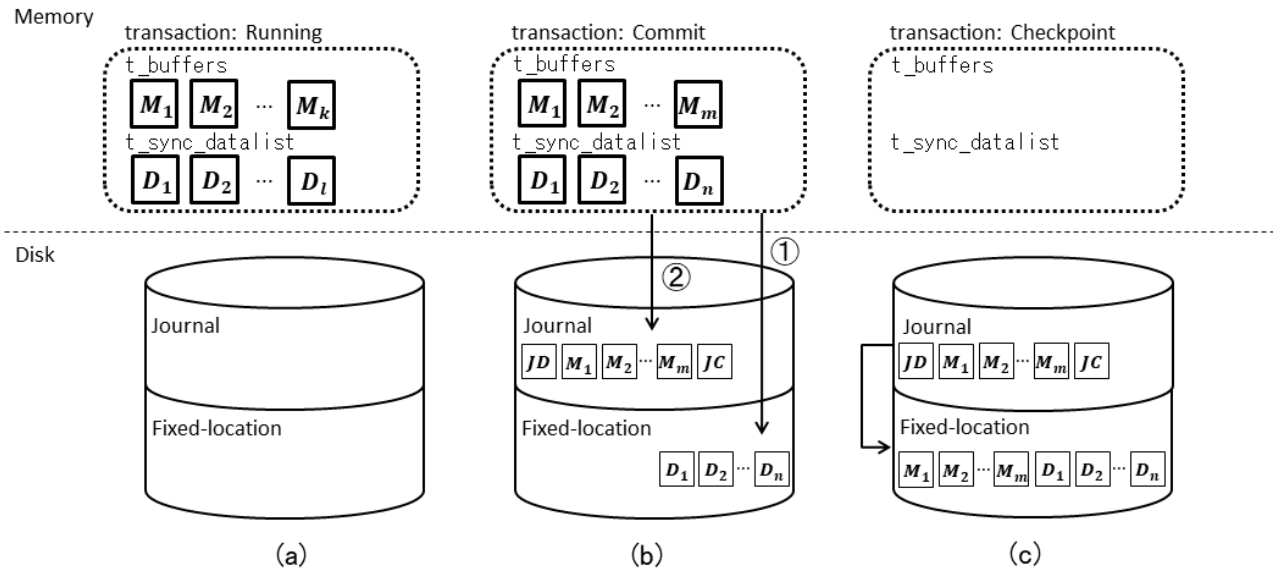


図 2.7　トランザクションの状態

ナリングのデータ構造でもある t_sync_datalist リストに追加することで，保存領域への書き込み順番を強制することができる．しかし，t_sync_datalist リストに追加されたデータは，ジャーナリングのトランザクションとして扱われるため，pdflush の書き込みと比べて実行時間が遅い．data モードでは，ジャーナリング対象の M，D を t_buffers リストに追加する．

トランザクションには，実行状態，コミット状態，チェックポイント状態の 3 つの状態がある．実行状態 (図 2.7 (a)) では，新しい更新履歴を追加することができる．この状態のトランザクションで同期命令やタイムアウトが発生すると，コミット状態へ遷移する．コミット状態 (図 2.7 (b)) では，新しい更新履歴を追加することはできず，このトランザクション内の変更履歴がまとめてコミットされる．トランザクション内のすべての更新履歴のジャーナルへ書き込みを終えると，チェックポイント状態へ遷移する．チェックポイント状態 (図 2.7 (c)) は，コミットが完了している状態である．この状態まで到達すると，OS が異常終了した場合でも，ファイルシステムをリカバリすることができる．トランザクション内のすべての更新履歴を保存領域へ書き込みを終えると，トランザクションを終了させる．

2.3.3　ext3 のファイル書き込みの流れ

ext3 のジャーナリングの処理では，Dirty フラグの代わりにジャーナリングを行うデータには JBDDirty フラグを立てることで，ジャーナリングを行うデータと行わないデータを区別する．JBDDirty フラグが立っているデータはカーネルスレッド kjournald が，Dirty フラグが立っているデータはカーネルスレッド pdflush が処理する．

図 2.8 に，ext3 が writeback モードでジャーナリングの処理を行うときの書き込みの様子

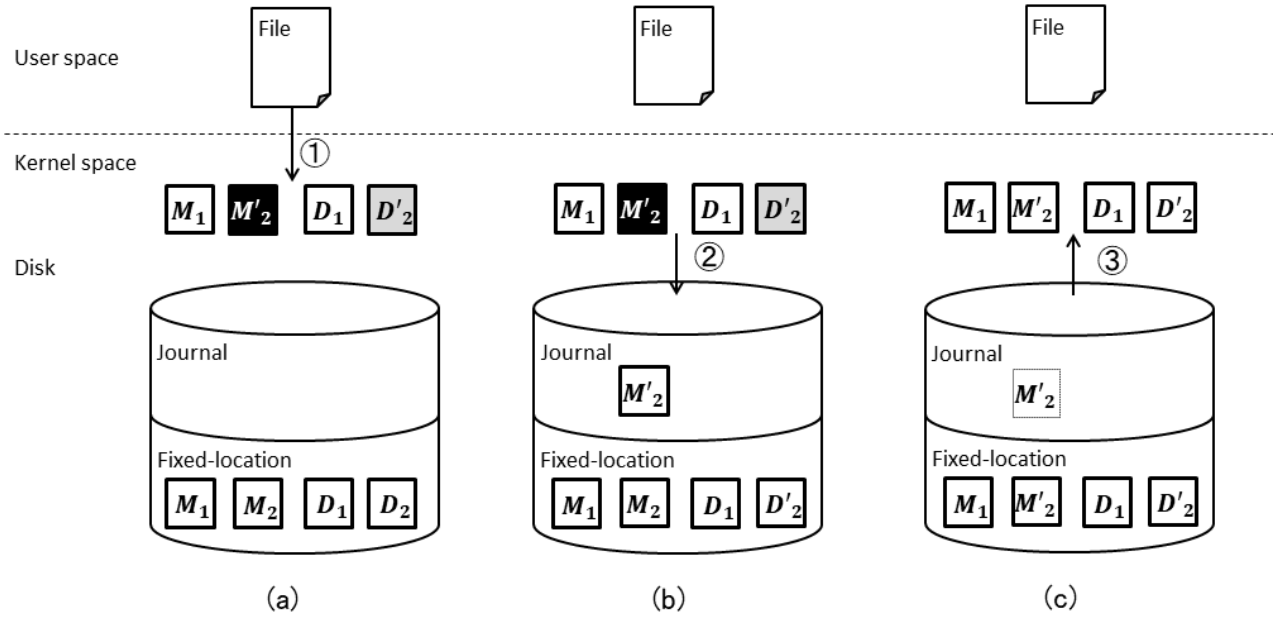


図 2.8　ext3 (writeback モード) におけるファイル書き込み．(a) write システムコールによる処理 (b) カーネルスレッドによる処理 (c) Dirty フラグと JBDDirty フラグのクリア

を示す．ここで，D は実データ，M はメタデータ，Dirty フラグが立っているデータを灰色，JBDDirty フラグが立っているデータを黒色で表す．

ext3 の writeback モードでは，ファイルの書き込みが発生すると，ジャーナリング対象であるメタデータに対して `journal_dirty_metadata` 関数，対象ではない実データに対して `block_write_end` 関数を実行する．`journal_dirty_metadata` 関数は，対象のメタデータに JBDDirty フラグを立て，実行中のトランザクションの `t_buffers` リストに追加する．`block_write_end` 関数は，対象のデータに Dirty フラグを立てる (図 2.8 (a))．カーネルスレッド pdflush は Dirty フラグの立っているデータをディスクに書き込み，kjournald は，JBDDirty フラグの立っているデータをコミット，チェックポイントを行う (図 2.8 (b))．その後，カーネルスレッドから書き込み完了の通知を受け取ると，対応するフラグを下ろす (図 2.8 (c))．

ordered モードの場合には，`block_write_end` 関数の代わりに `journal_dirty_data` 関数が，data モードの場合には `journal_dirty_metadata` 関数を実行する．`journal_dirty_data` 関数は，対象のデータに JBDDirty フラグを立て，実行中のトランザクションの `t_sync_datalist` リストに追加する．

2.3.4　JBD のファイル書き込みの流れ

JBD は，カーネルスレッド kjournald を生成し，JBDDirty フラグが立っているデータに対してジャーナリング処理を行う．kjournald は，以下の 9 つの処理を繰り返し実行する．

Phase 0: トランザクションにデータが追加され，一定時間経過 (デフォルトのタイムアウト時間は 5 秒) するまで待機する．

Phase 1: チェックポイント状態のトランザクション内の総数が閾値以上である，または同期命令を受信していたならば，チェックポイント状態のトランザクションのデータを保存領域に書き込み，そのトランザクションを終了させる．その後，ジャーナルスーパーブロックにチェックポイントしたトランザクションのトランザクション ID を書き込む．

Phase 2: 実行状態のトランザクションをコミット状態に遷移させる．以降，データの書き込みが発生した際には，そのプロセスを待機させ，次のトランザクションに追加させる．

Phase 3: コミット状態の全てのトランザクションに対して，`t_sync_datalist` リストに繋がっているデータの I/O 要求を発行する．I/O 完了通知を受信し次第，リストに繋がっているデータを削除する．

Phase 4: コミット状態の全てのトランザクションに対して，`t_sync_datalist` リストにデータが繋がっていない，かつ `t_buffers` リストにデータが繋がっているトランザクションのデータと，それに対応するディスクリプタブロックの I/O 要求を発行する．

Phase 5: **Phase 4** で発行した I/O 要求が完了するまで待機する．

Phase 6: **Phase 4** のトランザクションに対して，コミットブロックの I/O 要求を発行する．

Phase 7: **Phase 6** のトランザクションをチェックポイント状態に遷移させる．

Phase 8: ジャーナルスーパーブロックに格納されているトランザクション ID をカウントアップし，実行状態のトランザクションを生成する．

また，kjournald はスレッドの開始時に，必要であればファイルシステムのリカバリを行う．ジャーナルスーパーブロックに書き込まれている最後にチェックポイントをしたトランザクション ID と，最後に書き込まれたコミットブロックのトランザクション ID を比較する．前者が後者より小さければ，ジャーナルスーパーブロック ID 以降のトランザクションはチェックポイントされていない．kjournald は，チェックポイントされていないトランザクションをチェックポイント状態とし，保存領域に書き込む．

3 関連研究

これまでに，ジャーナリングモードの設定単位を細かくする研究はいくつか行われている．

Okeanos [15] は，書き込み時に複数のページキャッシュをまとめてコミットする Wasteless Journaling モードと，データサイズが閾値以下の書き込みには Wasteless Journaling を適用する Selective Journaling モードを持つファイルシステムである．図 3.1 に，Wasteless Journaling と Selective Journaling の動作例を示す．ここで，i ノードキャッシュは M'，ページキャッシュは D'_1，D'_2，D'_3，現在ディスクに書き込まれているデータ D_1，D_2，D_3 からの差分を灰色で表す (図 3.1 (a))．これまでの ext3 では，このような書き込みに対してページキャッシュ単位で書き込みを行うため，書き込む必要のないデータの書き込みまで行ってしまう．そのため，小さいサイズの書き込みがシステムのボトルネックとなる [16]．そこで，Wasteless Journaling モードでは，ページキャッシュの差分を結合して一つのデータ D' とする (図 3.1 (b-1))．カーネルスレッド kjournald は，D' に対してジャーナリングを行うため，ディスク I/O 待ち時間を削減することができる．しかし，ページキャッシュの差分を生成するためにデータを複製する必要があるため，無駄に複製されるデータも存在してしまう．

そこで，Selective Journaling モードでは，データサイズが閾値以下の書き込みには Wasteless Journaling モードを，それ以上の書き込みには ordered モードで書き込む (図 3.1 (b-2))．これにより，書き込むブロック数を減らすことで，ext3 の ordered モード以上の信頼性を保ちなが

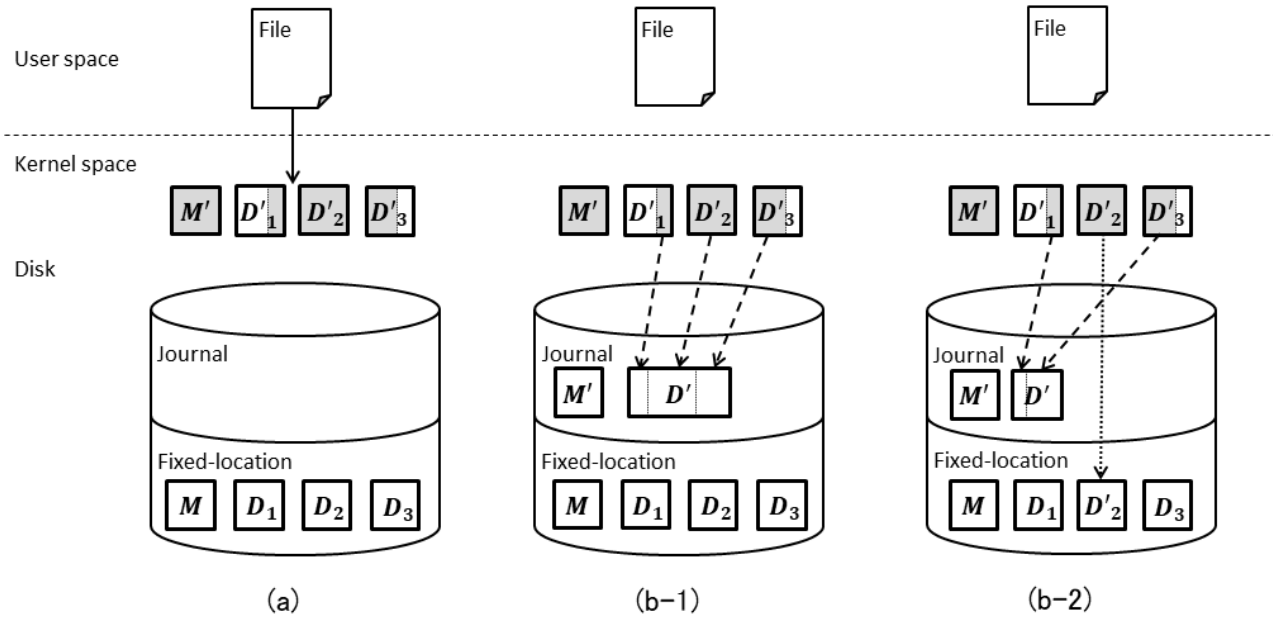


図 3.1　Okeanos ファイルシステム

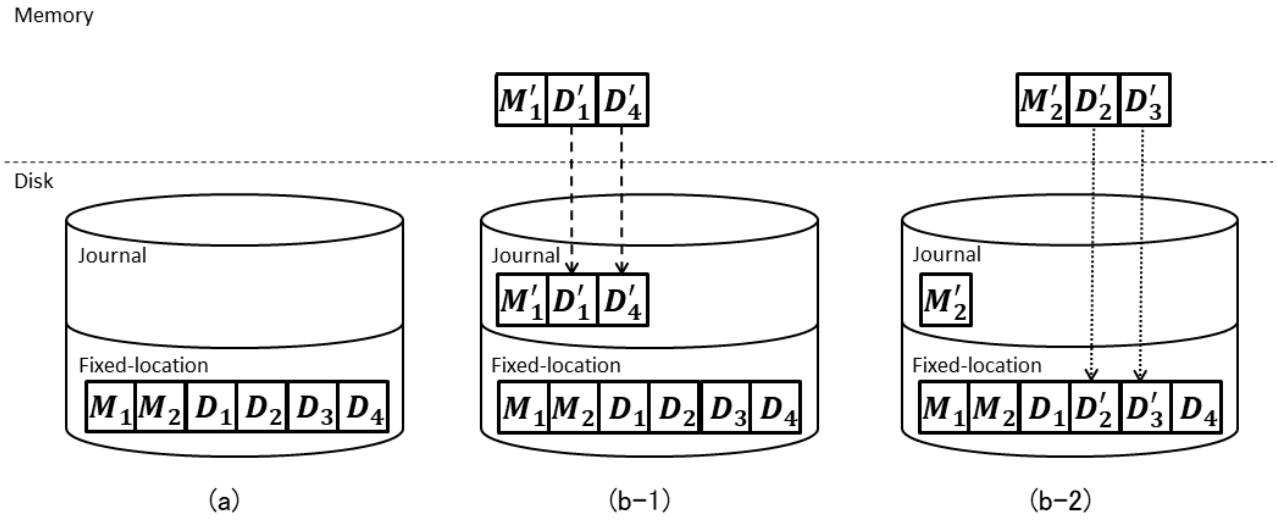


図 3.2　Adaptive Journaling

ら，ディスク I/O 待ち時間を減らすことができる．

Adaptive Journaling [10] は，トランザクション内のデータの I/O パターンから，ジャーナリングモードを自動的に設定する機構である．図 3.2 に，システムの動作例を示す．ここで，ディスクの保存領域は M_1，M_2，D_1，D_2，D_3，D_4 の順番に保存されており，一つ目のトランザクションで M'_1，D'_1，D'_4 を，二つ目のトランザクションで M'_2，D'_2，D'_3 を書き込む場合を考える (図 3.2 (a))．これまでの ext3 の ordered モードでは，一つ目のトランザクションで M'_1 をジャーナルに，D'_1，D'_4 を保存領域に書き込む．二つ目のトランザクションで M'_2 をジャーナルに，D'_2，D'_3 を保存領域に書き込む．その後，M'_1，M'_2 が保存領域に書き込まれる．一方，data モードは，一つ目のトランザクションで M'_1，D'_1，D'_4 を，二つ目のトランザクションで M'_2，D'_2，D'_3 をジャーナルに書き込む．その後，M'_1，D'_1，D'_4，M'_2，D'_2，D'_3 が保存領域に書き込まれる．しかし，ordered モードは一つ目のトランザクションに対して，シーケンシャルではない D'_1，D'_4 を書き込むため，ディスク I/O 待ち時間が増加してしまう．また，data モードは二つ目のトランザクションに対して，シーケンシャルな書き込み D'_3，D'_4 をシーケンシャルでない書き込みとして扱うため，ディスク I/O 待ち時間が増加してしまう．そこで，Adaptive Journaling では，トランザクション内の実データの書き込みパターンを確認し，シーケンシャルな書き込みかそうではないかで，ジャーナリングモードを自動的に設定する．シーケンシャルな書き込みの場合，ordered モード 図 3.2 (b - 1)，シーケンシャルでない書き込みの場合，data モード (図 3.2 (b - 2)) を適用することで，ディスクの無駄なシーク時間を減らす．

Okeanos はデータサイズに応じてジャーナリングモードを自動的に設定する．また，Adaptive Journaling はトランザクション単位でジャーナリングモードを自動的に設定する．しかし，どちらに関しても，ユーザ自身の判断に基くファイルの重要度に応じた設定をすることができない．

File-Adaptive Journaling [17] は，ファイル単位でジャーナリングモードを設定できる機構で

ある．ユーザは，予めファイルにモードを設定することで，当該ファイルに対する操作は設定したジャーナリングモードに基づいて処理されるようになる．このことで，ファイルシステム内に複数のモードを持つことを可能としている．しかし，File-Adaptive Journaling では，ファイル一つ一つにジャーナリングモードを設定する必要があり，設定が非常に煩雑である．例えば，新規ファイルにジャーナリングモードを設定する場合，ファイルの書き込み前にユーザがジャーナリングモードを設定しなければならない．それに対して本研究では，ディレクトリ単位でジャーナリングモードを設定するため，設定の煩雑さが軽減されている．

4 設計

4.1 システムの概要

dajFS は，ディレクトリ単位でジャーナリングモードを設定可能なファイルシステムである．同一ディレクトリ直下の全てのファイルに，そのディレクトリに設定されたジャーナリングモードを適用する．また，Linux ではファイルやディレクトリにリンクを貼ることでファイルに別名をつけ，異なる名前で同一ファイルにアクセスできるため，ファイルと親ディレクトリの関係が一対一になるとは限らない．そこで，複数の親ディレクトリが参照される可能性がある以下の 3 つの状況については，特別な処理を行う．

- ファイルを移動した場合，移動後は移動先のディレクトリのジャーナリングモードに従うことにする．ただし，移動元のディレクトリからファイルエントリを削除する操作は，元のジャーナリングモードに従う．
- ハードリンクに対する操作は，最後に貼られたリンク先ファイルの親ディレクトリのジャーナリングモードに従う．
- シンボリックリンクに対する操作は，リンク元ファイルの親ディレクトリのジャーナリングモードに従う．

新規ディレクトリには，親ディレクトリのモードを初期モードとして設定する．その後，子ディレクトリには，親ディレクトリのジャーナリングモードとは独立に，ジャーナリングモードを設定できる．設定できるジャーナリングモードは，ext3 と同じく，data，ordered，writeback に加えて，ジャーナリングの処理を行わずにファイル操作をする none のいずれかとする．

各ジャーナリングモードのオーバーヘッドと信頼性を，表 4.1 に示す．ここで，IB は i ノードビットマップ，DB はデータビットマップ，I はファイルの i ノード，D はファイルの実データを表す．none モードは，独自の方式でファイルの整合性を保証するアプリケーションに対し

表 4.1　dajFS のジャーナリングモード

モード	保存対象				保存領域への	オーバーヘッド	信頼性
	IB	DB	I	D	書き込み順番の強制		
none	✓	✓			×	極小	極低
writeback	✓	✓	✓		×	小	低
ordered	✓	✓	✓		$D \rightarrow M$	中	中
data	✓	✓	✓	✓	×	大	高

て使用することを想定している．例えば SQLite では，SQL 文をトランザクションとして扱うことで，データベースの整合性を保証する [18]．しかし，そのようなアプリケーションでも，ファイルシステムの整合性は保証することはできない．そこで none モードは，i ノードビットマップとデータビットマップのみをジャーナリングの対象とすることで，必要最低限の整合性を確保し，オーバーヘッドを最小限に抑える．

dajFS は，ext3 と同様にブロックデバイスに mkfs でファイルシステムを生成した後，mount でファイルシステムをマウントして利用する．mkfs によるファイルシステムの生成時に，ルートディレクトリのジャーナリングモードを設定することができる．ただし，ジャーナリングモードを指定していない場合は，none モードを設定する．また，ユーザは，ファイルシステムのマウント後に，本システムが提供するインタフェースを用いて，ディレクトリのジャーナリングモードを設定・変更することができる．

dajFS は，ext3 と互換性があるため，ext3 あるいは ext2 として使用していたパーティションを初期化せずにマウントすることが可能である．これらのファイルシステムは，ディレクトリにジャーナリングモードを持たないため，全てのディレクトリのジャーナリングモードは none モードで開始する．

4.2 ジャーナリングモードの設定単位

ここで, ジャーナリングモードの設定単位をディレクトリとすることについて考察する．Linux は，多数のファイルやディレクトリの名前とその内容を Filesystem Hierarchy Statndard [19] という規格として定めている．表 4.2 は，その規格の一例である．例えば，/etc の下にはシステ

表 4.2 Filesystem Hierarchy Standard で定義されているディレクトリ構造例

ディレクトリ	概要
/bin	シングルユーザモードで必要となるコマンドの実行ファイル群
/boot	ブートに必要となる Linux カーネルなどの静的ファイル群
/etc	システム全体に関わる固有の設定ファイル群
/etc/X11	X Window System 11 の設定ファイル群
/home	ユーザのホームディレクトリ群
/root	root ユーザのホームディレクトリ
/tmp	一時ファイル群
/var	可変なデータファイル群
/var/cache	アプリケーションのキャッシュデータ群
/var/lock	排他制御に使うロックファイル群
/var/mail	ユーザのメールボックス

ムの設定ファイルが多く置かれ，/tmp の下にはプログラムの一時ファイルが多く置かれる．このことから示唆されるように，同じディレクトリ内のファイルは共通する性質を持ち，その結果重要度が共通することが多い．従って，これらのファイルのジャーナリングモードを共通とし，モード設定の細かさの単位をディレクトリとする本システムの設計は妥当であると考えられる．

4.3 ジャーナリングモードの操作

本システムでは，ユーザがジャーナリングモードを操作できるよう以下のコマンドを用意した．

[名前]

setjournal

[書式]

setjournal [-r] *dir ... mode*

[説明]

指定したディレクトリ *dir* にジャーナリングモード *mode* を設定する．-r オプションを指定すると，子ディレクトリにも再帰的にモードを設定する．

[使用例]

dir1 と dir2 に writeback モードを設定する

```
> setjournal dir1 dir2 writeback
dir1: none -> writeback
dir2: none -> writeback
```

[名前]

lsjournal

[書式]

lsjournal [*dir*]

[説明]

指定したディレクトリ *dir* (デフォルトはカレントディレクトリ) 以下のディレクトリ構造を表示する．ディレクトリの末尾にのみモード情報が表示され，それ以外のファイルは名前のみ表示される．

モードは次のように表示される．

(w) writeback, (o) ordered, (d) data, ( ) none

[使用例]

図 4.1 (d) のディレクトリ構成において，dir 以下のジャーナリングモードを表示する

```
> lsjournal dir
dir/ (O)
- subdir/ (W)
- file
```

また，ファイルシステムの生成時にジャーナリングモードを設定できるように mke2fs コマンドを以下に拡張した．ここで，拡張元の mke2fs コマンドと同じオプションは省略する．

[名前]

mke2fs

[書式]

mke2fs [...] [-J *journal-option*] [-t *fs-type*] *device*

[説明]

device は対応するスペシャルファイルにファイルシステムを作成する．

本システムで，新たに追加したオプションは以下の通りである．

-J *journal-option*

root=*journal-mode*

ルートディレクトリのジャーナリングモードを *journal-mode* に設定する．

-t *fs-type*

daj: dajFS でファイルシステムを初期化する．

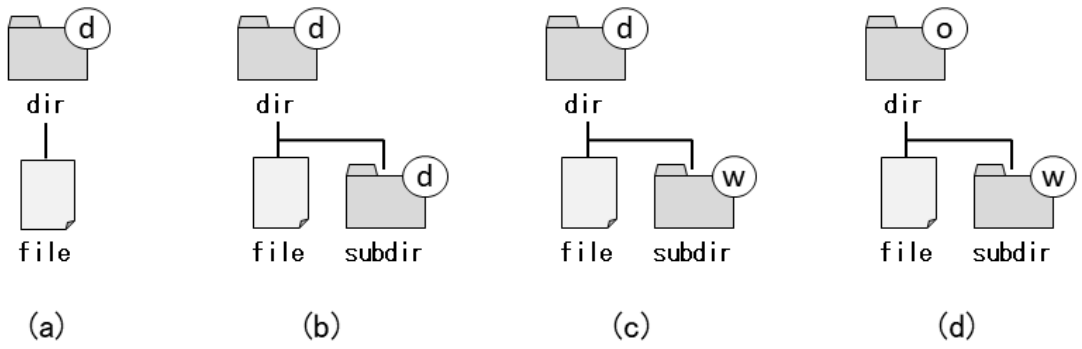


図 4.1　ジャーナリングモード設定の流れ

これらのコマンドを用いて，ジャーナリングモードを設定するときの動作を図 4.1 に示す．はじめに，ディレクトリ dir は data モードに設定されていて，dir 以下にファイル file が存在するものとする (図 4.1 (a))．ここで，dir 以下にディレクトリ subdir を新規作成すると，subdir は初期モードとして data モードが設定される (図 4.1 (b))．その後，ユーザは setjournal コマンドを用いて，subdir を writeback モードに設定することができる (図 4.1 (c))．さらに，ユーザは同じコマンドを用いて，dir を ordered モードに設定することもできる (図 4.1 (d))．

4.4 システムの動作例

図 4.2 を用いて，本システムの動作の流れを説明する．ここで，カレントユーザは aoyama とし，ホームディレクトリを/home/aoyama とする．ディレクトリ右上の丸の中の文字は，d は data，o は ordered，w は writeback モードが設定されていることを表し，モードが描かれていないディレクトリは none モードが設定されていることを表す．点線の矢印は参照先のファイルのシンボリックリンクであることを表す．ファイル上部にある数字はリンクが作成された順番を表す．例えば，/home/guest/minutes は，/home/guest/minutes が作成された後に~/minutes をハードリンクとして作成したことを表す．ただし，リンク数が 1 のファイルは数字を省略する．このとき，以下の 4 つのケースについて考える．図 4.3 にファイルの変更とファイルの移動の動作例を，図 4.4 にシンボリックリンクに対する操作とハードリンクに対する動作の動作例を示す．ファイル右上の丸の中の文字は，親ディレクトリのジャーナリングモードを示す．ここ

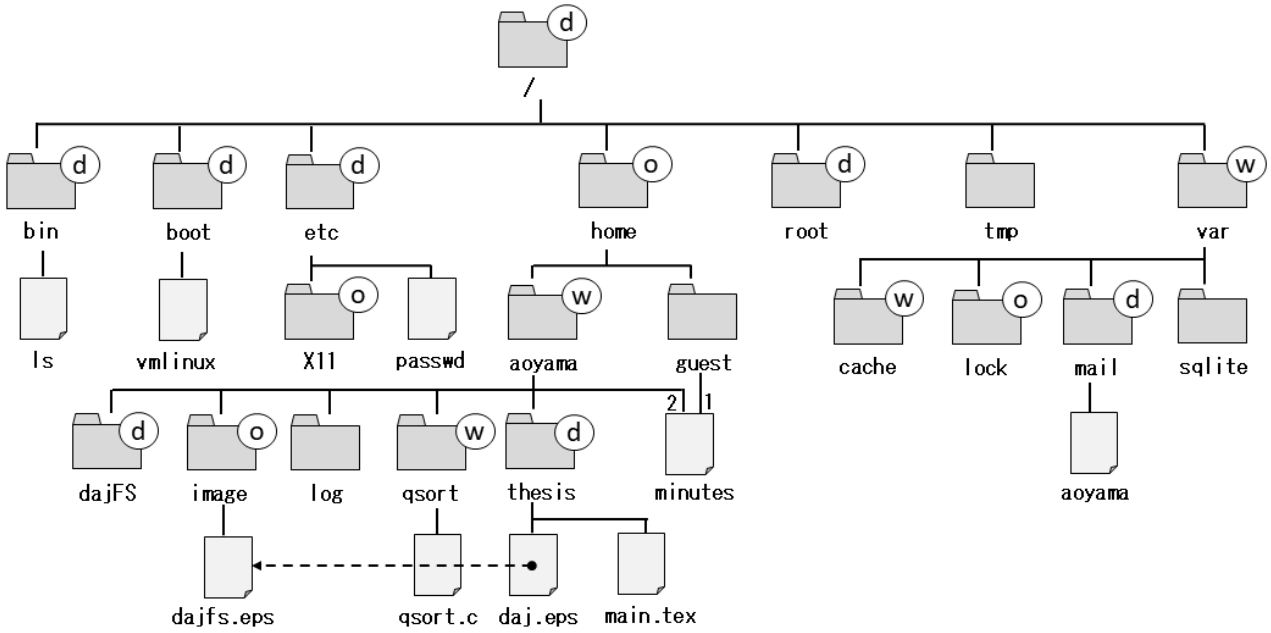


図 4.2　ディレクトリ構造例

で，IB は i ノードビットマップ，DB はデータビットマップ，JD はディスクリプタブロック，JC はコミットブロックを表す．

ケース 1: ファイルの変更 ~/qsort/qsort.c を編集する場合 (図 4.3 のケース 1)
変更前の ~/qsort/qsort.c の i ノードを I_q，実データを D_q とする．ここで，~/qsort/qsort.c の i ノードを I'_q，実データを D'_q へ変更した場合を考える．ファイルは親ディレクトリのジャーナリングモードに従うため，~/qsort のジャーナリングモードの writeback モードが適用される．従って，ジャーナルに JD, I'_q, JC が書き込まれ，保存領域に D'_q に書き込まれる．

ケース 2: ファイルの移動 /var/mail/aoyama を~/log に移動する場合 (図 4.3 のケース 2)
変更前の /var/mail の i ノードを I_m，実データを D_m，~/log の i ノードを I_l，実データを D_l とする．ここで，ファイルエントリの削除によって/var/mail の i ノードを I'_m, 実データを D'_m へ，ファイルエントリの追加によって~/log の i ノードを I'_l，実データを I'_l へ変更した場合を考える．ファイルを移動した場合には，移動元のディレクトリからファイルエントリを削除する操作は元のジャーナリングモードに従うため，ファイルエントリの削除は/var/mail のジャーナリングモードの data モードが適用される．また，ファイルエントリの追加は~/log のジャーナリングモードの none モードが適用される．従って，ジャーナルに JD, I'_m, D'_m, JC が書き込まれ，保存領域に I'_l, D'_l に書き込まれる．また，~/log/aoyama は移動先のディレクトリのジャーナリングモードの none モードに従うことにする．

ケース 3: シンボリックリンクに対する操作 ~/thesis/daj.eps を編集する場合 (図 4.4 の case 3)
変更前の ~/thesis/daj.eps の i ノードを I_s，実データを D_s とする．ここで，~/thesis/daj.eps の i ノードを I'_s，実データを D'_s へ変更した場合を考える．シンボリックリンクに対する操作は，リンク元ファイルの親ディレクトリのジャーナリングモードに従うため，リンク元ファイル~/image/dajfs.eps の親ディレクトリ~/image の ordered モードが適用される．従って，ジャーナルに JD, I'_s, JC が書き込まれ，保存領域に D'_s に書き込まれる．

ケース 4: ハードリンクに対する操作 /home/guest/minutes を編集する場合 (図 4.4 の case 4)
変更前の /home/guest/minutes の i ノードを I_h，実データを D_h とする．ここで，/home/guest/minutes の i ノードを I'_h，実データを D'_h へ変更した場合を考える．ハードリンクに対する操作は，最後に貼られたリンク先ファイルの親ディレクトリのジャーナリングモードに従うため，最後に貼られた~/の writeback モードが適用される．従って，ジャーナルに JD, I'_s, JC が書き込まれ，保存領域に D'_s に書き込まれる．

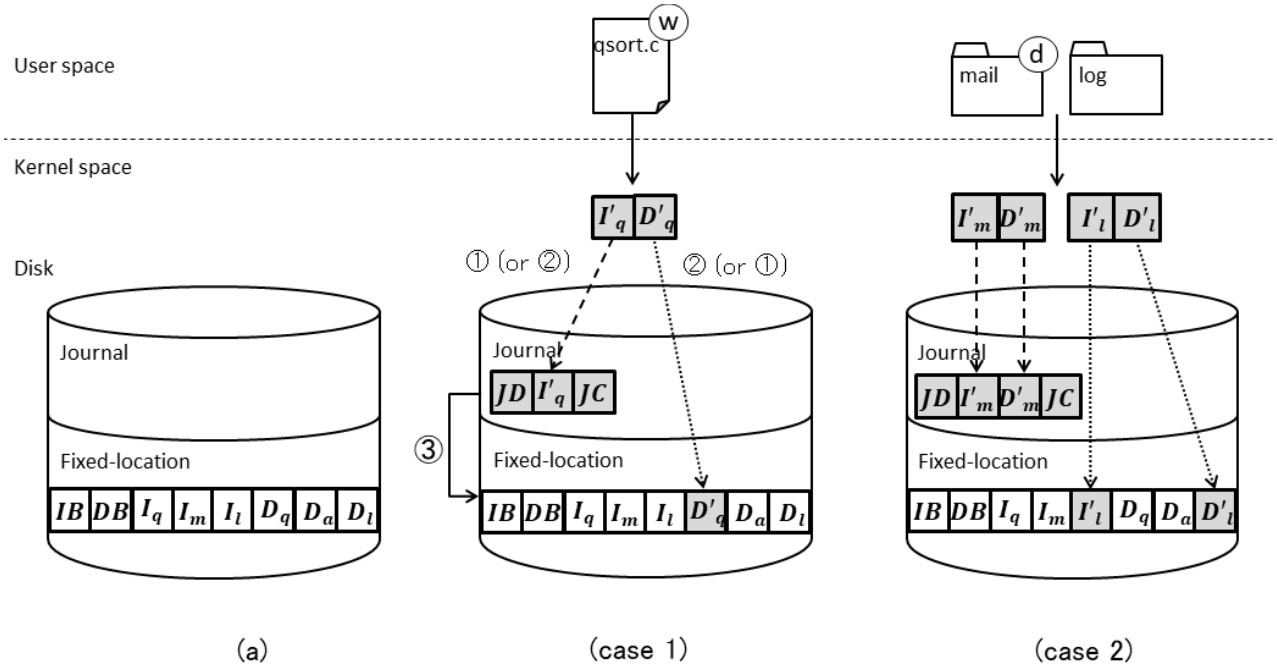


図 4.3　本システムの動作例 (ケース 1:ファイルの変更，ケース 2: ファイルの移動)

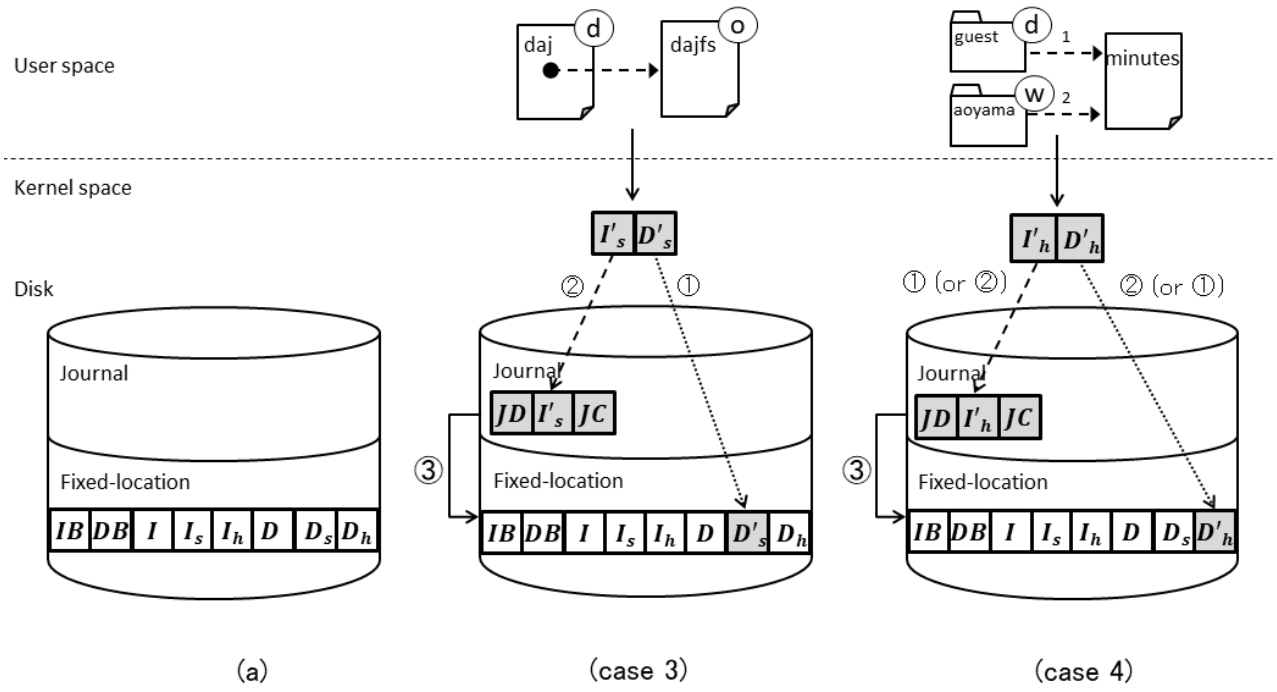


図 4.4　本システムの動作例 (ケース 3:シンボリックリンクに対する操作，ケース 4:ハードリンクに対する操作)

5 実装

dajFS は，Linux Kernel 4.2 の ext3 と JBD を拡張して実現する．また，ファイルシステムの生成時にルートディレクトリのジャーナリングモードを設定できるように，ファイルシステムユーティリティ E2fsprogs[*1]の `mke2fs` を拡張する．

5.1 ジャーナリングモードのデータ構造

ディレクトリにジャーナリングモード情報を保持するため，ディスク上の i ノードブロック (`struct ext3_inode`) とメモリ上にキャッシュされる i ノードキャッシュ (`struct inode`) にジャーナリングモードの情報を追加した．しかし，ディスク上のブロックは固定長であるため，i ノードブロックに新しくモードのための変数を追加することはできない．そこで，ジャーナリングモードをファイルの属性フラグの中に，既存の変数にモード情報を保持する．ext3 では，ファイルの属性フラグ (`i_flags`) を 32 ビットで管理しているが，12 ビットの未使用部分がある．そこで dajFS では，この未使用部分の 2 ビットに以下のモード情報を新たに保持するようにした．

`WRITEBACK_MODE_FL` 指定されたディレクトリは writeback モードである．
`ORDERED_MODE_FL` 指定されたディレクトリは ordered モードである．
`DATA_MODE_FL` 指定されたディレクトリは data モードである．
`NONE_MODE_FL` 指定されたディレクトリは none モードである．

i ノードキャッシュに関しては，属性フラグの中に以上のモード情報を保持する．dajFS で追加したモード情報は，従来の ext3 と同様に `ext3_iget` 関数で取得，`ext3_do_update_inode` 関数で更新される．

dajFS では，ファイル操作時のジャーナリングモードを参照するオーバーヘッドを減らすために，ファイルの i ノードにもジャーナリングモードをキャッシュとして自動的に格納する．そのタイミングは以下の通りである．

ファイルを新規作成した時

ファイルを新規作成した (i ノードを新規作成した) 時，新規作成したファイルは親ディレクトリのモード情報をキャッシュする．dajFS では，`ext3_new_inode` 関数を拡張することで実現した．`ext3_new_inode` 関数の定義を以下に示す．

[*1] `http://e2fsprogs.sourceforge.net/`

```
struct inode *ext3_new_inode(handle_t *handle, struct inode *dir,
        const struct qstr *qstr, umode_t mode)
```

この関数は，handle で指したトランザクションに対して，ディレクトリ dir にパーミッション mode の i ノードを作成する．この時，作成された i ノードの i_flags は，ディレクトリの i_flags を元に設定される．dajFS では，設定された i_flags からモード情報をキャッシュする．

ファイルを移動した時

ファイルを別ディレクトリへ移動した (親ディレクトリを変更した) 時，移動したファイルは移動先のディレクトリのモード情報をキャッシュする．dajFS では，ext3_rename 関数を拡張することで実現した．ext3_rename 関数の定義を以下に示す．

```
int ext3_rename (struct inode *old_dir, struct dentry *old_dentry,
        struct inode *new_dir, struct dentry *new_dentry)
```

この関数は，移動前のディレクトリ old_dir のファイルエントリ old_dentry を削除し，移動先のディレクトリ new_dir にファイルエントリ new_dentry を追加する．dajFS では，ディレクトリエントリの追加後に移動したファイルのモード情報を new_dir のモード情報に設定する．

親ディレクトリのジャーナリングモードが変更された時

親ディレクトリのモード情報が変更された時，ディレクトリ内のファイルは変更後のモード情報をキャッシュする．概要については，5.2 節で説明する．

5.2 ジャーナリングモードの設定と取得

ユーザがディレクトリに付与されたモードを更新と取得するインタフェースとして，ioctl システムコールを利用する．ext3 では，ext3_ioctl 関数によって実現している．

```
long ext3_ioctl(unsigned file *filp, unsigned int cmd,
                unsigned long arg);
```

この関数は，操作対象のファイルオブジェクト filp に対して，コマンド番号 cmd に対応する操作を引数 arg で実行する．Linux では，ユーザ空間からカーネルにパラメータを渡すときに用いられる．そこで dajFS では，ext3_ioctl 関数を拡張し，ジャーナリングモードを設定/取得するためのコマンド (IOC_GETJOURNAL/IOC_SETJOURNAL) を追加した．各コマンドの動作の流れを擬似コードを用いて説明する．

コマンド番号が IOC_GETJOURNAL の場合，対象のファイル inode がディレクトリであるか確認する．ディレクトリではなければエラー値を返し，ディレクトリであれば i ノードキャッシュのファイル状態フラグ i_flags を取得する．JOURNAL_MODE でマスクして，2 ビットの情報フ

ラグ mode を得る．その後，put_user 関数を用いて，モード情報をシステムコールの引数 arg に設定する．

```
    case IOC_GETJOURNAL: {
      unsigned int mode;
      struct inode *inode = file_inode(filp);

      if(S_ISDIR(inode->i_mode)){
        mode = inode->i_flags & JOURNAL_MODE;
        return put_user(mode, (int __user *) arg);
      }
      return -EFAULT;
    }
```

コマンド番号が IOC_SETJOURNAL の場合，対象のファイルオブジェクトがディレクトリであるか確認する．ディレクトリではなければエラー値を返し，ディレクトリであれば，get_user 関数を用いて ioctl システムコールの引数の arg を取得する．取得した引数 arg を，mode に格納し，JOURNAL_MODE でマスクする．ここで，モード情報の設定前に，journal_flush 関数を用いて kjournald で管理している全てのトランザクションをコミットしチェックポイントする．こうすることで，トランザクションの順番が逆転しないようにする．その後，対象のファイルのファイル状態フラグ中のモード情報として mode を設定し，ext3_do_update_inode 関数を用いて，i ノードブロックに書き込む．dajFS では，ファイルの i ノードにもジャーナリングモードをキャッシュしているため，set_subdirmode 関数を用いてディレクトリ内のファイルのモード情報を設定する．

```
  case IOC_SETJOURNAL: {
  unsigned int mode;
  struct inode *inode = file_inode(filp);

    if(S_ISDIR(inode->i_mode)) {
      journal_flush(journal);
      if (get_user(mode, (int __user *) arg)) {
        return -EFAULT;
      }
    } else
      return -EFAULT;

    mode = mode & JOURNAL_MODE;
    inode->i_flags = (inode->i_flags & ~JOURNAL_MODE) | mode;
    err = ext3_do_update_inode(inode);
    set_subdirmode(inode, mode);

    return err;
  }
```

dajFS では，ディレクトリの新規作成時には，親ディレクトリのジャーナリングモードを初期モードとして設定する．そこで，ext3 でディレクトリを新規生成する ext3_mkdir 関数を拡張する．

```
int ext3_mkdir(struct inode *dir, struct dentry *dentry, umode_t mode)
```

ext3_mkdir 関数は，ディレクトリ dir にパーミッション mode のファイルエントリ dentry を追加する．この関数は以下の流れで実行される．

Step 1: 新規ディレクトリ用の i ノードブロックを確保する．
Step 2: 新規ディレクトリ用の i ノードブロックを初期化する．
Step 3: 新規ディレクトリに"."と".."のファイルエントリを追加する．
Step 4: 新規ディレクトリの I/O 要求を発行する．
Step 5: ディレクトリ dir の I/O 要求を発行する．
Step 6: 新規ディレクトリの dentry と新規ディレクトリ用の i ノードをリンクする．

dajFS では，**Step 2** でモード情報の設定を行う．引数の dir から親ディレクトリのジャーナリングモードを取得し，新規ディレクトリの i_flags に追加する．
また本システムでは，ファイルシステムの生成時にルートディレクトリのジャーナリングモードを設定できるように mke2fs コマンドの追加オプションに root を追加した．ファイルシステムの生成後にルートディレクトリに対して，ioctl 関数の IOC_SETJOURNAL コマンドを実行する．

5.3 設定したジャーナリングモードのファイル書き込み

5.3.1 writeback/ordered/data モード

dajFS の writeback/ordered/data モードは，ext3 の writeback/ordered/data モードの実装を利用した．ext3 では，オブジェクトに登録する操作関数を変更することで各ジャーナリングモードの操作を実現している．例えば，アドレス空間オブジェクトの writepage 操作関数には，writeback モード用に ext3_writeback_writepage 関数，ordered モード用に ext3_ordered_writepage 関数，data モード用に ext3_journalled_writepage 関数が用意されている．しかし，これらの操作関数は，ファイルシステムのマウント時に決定してしまう．そこで dajFS では，ファイル毎に設定したモードに従ってこれらの関数を呼び出す新たなアドレス空間操作関数を実装した．例えば，writepage の場合には，以下の関数を定義した．

```
static int dajfs_writepage(struct page *page,
        struct writeback_control *wbc)
{
  struct inode *inode = page->mapping->host;
```

```
  unsigned int mode = inode->i_flags & JOURNAL_MODE;

  switch(mode){
    case JOURNAL_MODE_FL:
      return ext3_journalled_writepage(page, wbc);
    case ORDERED_MODE_FL:
      return ext3_ordered_writepage(page, wbc);
    case WRITEBACK_MODE_FL:
      return ext3_writeback_writepage(page, wbc);
  }
  ...
}
```

本システムで拡張を行ったアドレス空間操作関数は以下の通りである．各モードの操作関数の特徴とともに記す．

`writepage` ページキャッシュをディスクに書き込む．

writeback：Dirty フラグが立っているページキャッシュに関する I/O 要求を発行する．

ordered： JBDDirty フラグ立っているページキャッシュに関する I/O 要求を発行する．

data：JBDDirty フラグが立っているページキャッシュに関するジャーナリング用の I/O 要求を発行する．

`write_end` ページキャッシュを書き込み準備状態にする．

writeback：ページキャッシュに Dirty フラグを立てる．

ordered：ページキャッシュに JBDDirty を立て，キャッシュを現在のトランザクションの `t_sync_datalist` リストに追加する．

data：ページキャッシュに JBDDirty フラグを立て，キャッシュを現在のトランザクションの `t_buffers` リストに追加する．

`set_page_dirty` ページキャッシュに Dirty フラグを立てる．

writeback/ordered：ページキャッシュに Dirty フラグを立てる．

data：ページキャッシュに JBDDirty フラグを立てる．

`direct_IO` ページキャッシュをダイレクト I/O 転送する．

writeback/ordered：ダイレクト I/O 転送を行う．

data：ジャーナルを経由するダイレクト I/O 転送を行う．

`is_dirty_witeback` ページキャッシュに Dirty フラグが立っているかどうか判定する．

writeback/data：ページキャッシュに Dirty フラグを立てる．

ordered：ページキャッシュに JBDDirty フラグを立てる．

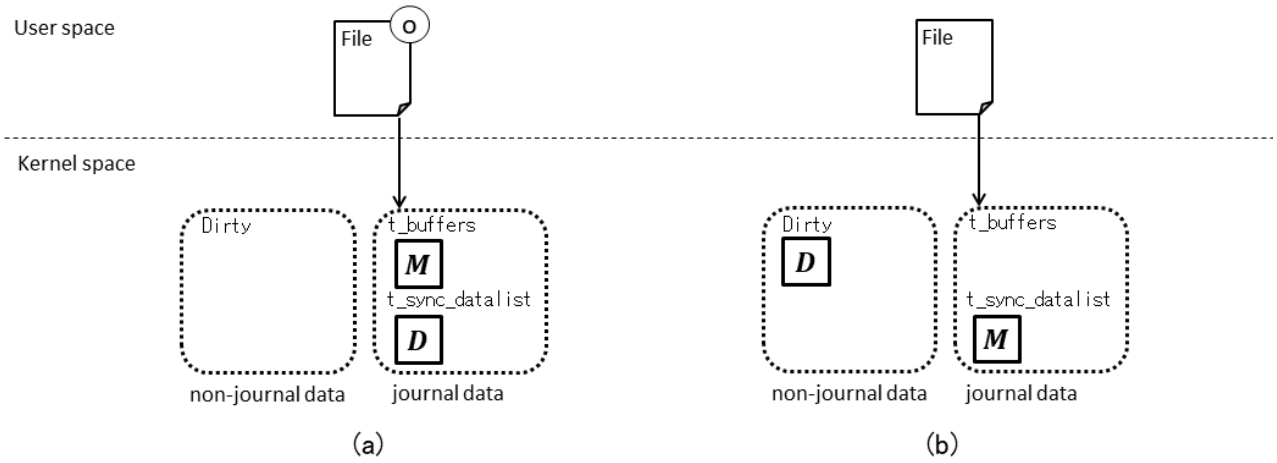


図 5.1　dajFS における none モードと ordered モードの比較

5.3.2　none モード

dajFS の none モードは，writeback モードを元に実装した．none モードは，i ノードビットマップとデータビットマップのみをジャーナリングの対象とする．しかし ext3 では，どのジャーナリングモードもファイルの i ノードはジャーナリングの対象となっているため，i ノードをジャーナリングの対象外にすることは難しい．そこで ordered モードを参考に，i ノードをジャーナリングの対象とするが，ジャーナルには書き込まないように扱うことにする (図 5.1)．

none モードを実現するために，JBD が提供している API の `journal_dirty_metadata` 関数を変更した．本システムでは，`journal_dirty_metadata` 関数内部でファイルのジャーナリングモードを取得する．対象のファイルが writeback/ordered/data モードの場合，従来通り JBDDirty フラグを立て，`t_buffers` リストに追加する．対象のファイルが none モードの場合，JBDDirty フラグを立て，`t_sync_datalist` リストに追加する．

既存の kjournald では，none モードの書き込み時，ジャーナルに書き込みを行わないトランザクションが生成される可能性がある．このようなトランザクションは，ファイルシステムの整合性の保証には関係ないため，削除することが望ましい．図 5.2 に，none モード時に無駄なディスク I/O が発生する例を示す．ここで，トランザクション (ID:1) はトランザクション ID が 1 のトランザクション，JS_1 は実行中のトランザクション ID が 1 であることを示すジャーナルスーパーブロック，JC_1 はトランザクション ID が 1 が終了したことを表すコミットブロックであるとする．初期状態 (図 5.2 (a)) から変更があったデータを灰色で表す．また，次の説明のフェーズ番号は 2.3.4 節におけるものを表す．

ファイルを i ノード I から I'，実データ D から D' に編集した場合を考える．none モードでは，i ノードと実データはトランザクション (ID:1) の `t_sync_datalist` リストに追加され

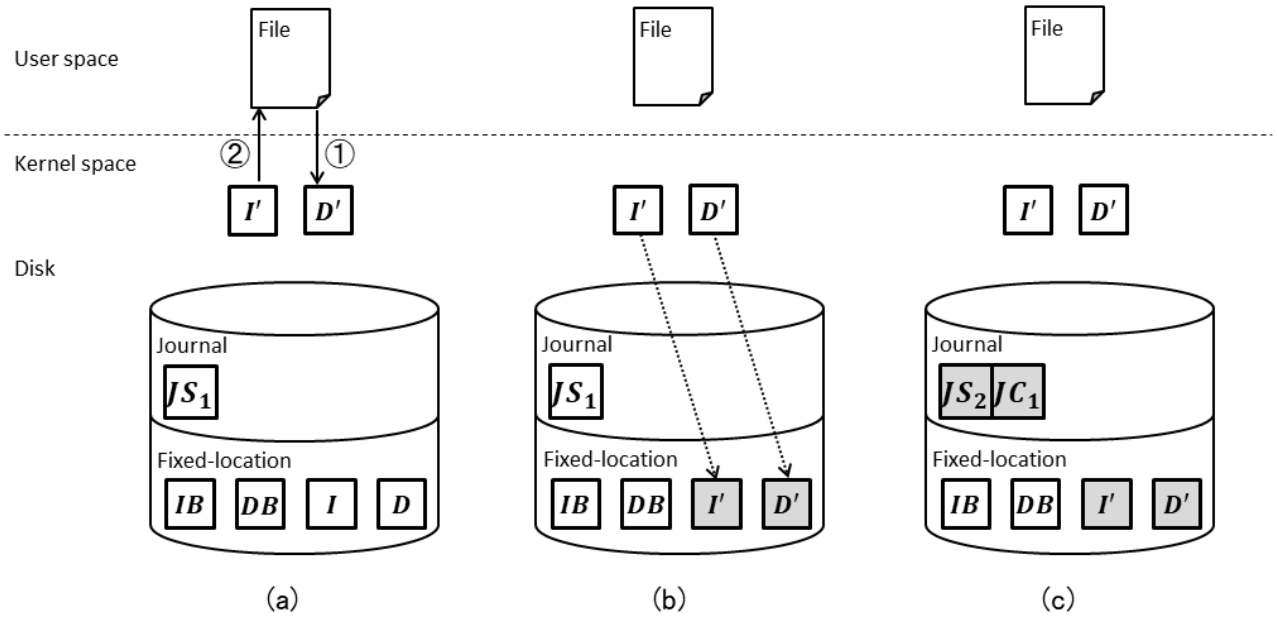


図 5.2　none モードにおいて無駄な書き込みが発生する例

る．ここで，タイムアウトで kjournald が起床すると，**Phase 2** でトランザクション (ID:1) をコミット状態に遷移させる．**Phase 3** では，コミット状態のトランザクション (ID:1) の `t_sync_datalist` リストに繋がっているデータ I' と D' を保存領域に書き込む (図 5.2 (b))．**Phase 6** で JC_1 をジャーナルに書き込み，トランザクション (ID:1) をチェックポイント状態に遷移させる．その後，**Phase 8** で JS_2 をジャーナルに書き込む (図 5.2 (c))．ここまでに，ジャーナルに JS_1，JC_1 と保存領域に I'，D' をディスクに書き込んだ．ジャーナルに書き込まれた JS_1 と JC_1 は，ファイルシステムをリカバリするうえで何の意味も持たない．

そこで本システムでは，上記のようなトランザクションがコミット状態になった際に，`t_sync_datalist` リストの書き込みだけを行い，以降の処理を行わないことにした．dajFS では，既存の kjournald と比較して，**Phase 3** と **Phase 4** の間に新しく処理を追加した．本システムで新たに拡張した kjournald は，以下の処理を繰り返し実行する．

Phase 0: トランザクションにデータを追加し，一定時間経過 (デフォルトのタイムアウト時間は 5 秒) するまで待機する．

Phase 1: チェックポイント状態のトランザクション内の総数が閾値以上である，または同期命令を受信していたならば，チェックポイント状態のトランザクションのデータを保存領域に書き込み，そのトランザクションを終了させる．その後，ジャーナルスーパーブロックにチェックポイントしたトランザクションのトランザクション ID を書き込む．

Phase 2: 実行状態のトランザクションをコミット状態に遷移させる．以降，データの書き

込みが発生した際には，そのプロセスを待機させ，次のトランザクションに追加させる．

Phase 3: コミット状態の全てのトランザクションに対して，t_sync_datalist リストに繋がっているデータの I/O 要求を発行する．I/O 完了通知を受信し次第，リストに繋がっているデータを削除する．

Phase 4: **Phase 1** で保存領域に書き込みを行っていない，かつ t_buffers リストにデータが繋がっていないならば，**Phase 0** に戻る．

Phase 5: コミット状態の全てのトランザクションに対して，t_sync_datalist リストにデータが繋がっていない，かつ t_buffers リストにデータが繋がっているトランザクションのデータと，それに対応するディスクリプタブロックの I/O 要求を発行する．

Phase 6: **Phase 4** で発行した I/O 要求が完了するまで待機する．

Phase 7: **Phase 4** のトランザクションに対してコミットブロックの I/O 要求を発行する．

Phase 8: **Phase 6** のトランザクションをチェックポイント状態に遷移させる．

Phase 9: ジャーナルスーパーブロックに格納されているトランザクション ID をカウントアップし，実行状態のトランザクションを生成する．

6 評価

提案システム dajFS を「拡張元の ext3 に提案手法を導入したことによるオーバーヘッド」と「ジャーナリングモードをディレクトリ単位に設定する妥当性」の二つの観点から評価する．実験には，表 6.1 のシステム構成を用いた．測定対象のファイルシステムは，OS がインストールされたディスクとは別に構築し，それぞれ HDD と SSD で実験を行った．実験は 5 回行い，平均値を計算した．ただし，CPU 稼働率が異常な値となるような外れ値は除いた．また，比較対象として，本システムの拡張元の ext3 (writeback モード，ordered モード，data モードをそれぞれ ext3-w，ext3-o，ext3-d と記す) と，ジャーナリングが実装されていないファイルシステムとして ext2 を用いた．

6.1 オーバーヘッドの計測

提案手法を導入したことによるオーバーヘッドを評価するために，ファイルシステムベンチマークツールを用いた．ここで，ファイルシステムのベンチマークはストレージの状態や I/O パターンに応じて測定結果が大きく偏りが生じる [20] ため，本実験では IOzone[*1] と Bonnie++[*2] の 2 つのベンチマークツールを用いてシステムの評価を行った．

6.1.1 IOzone

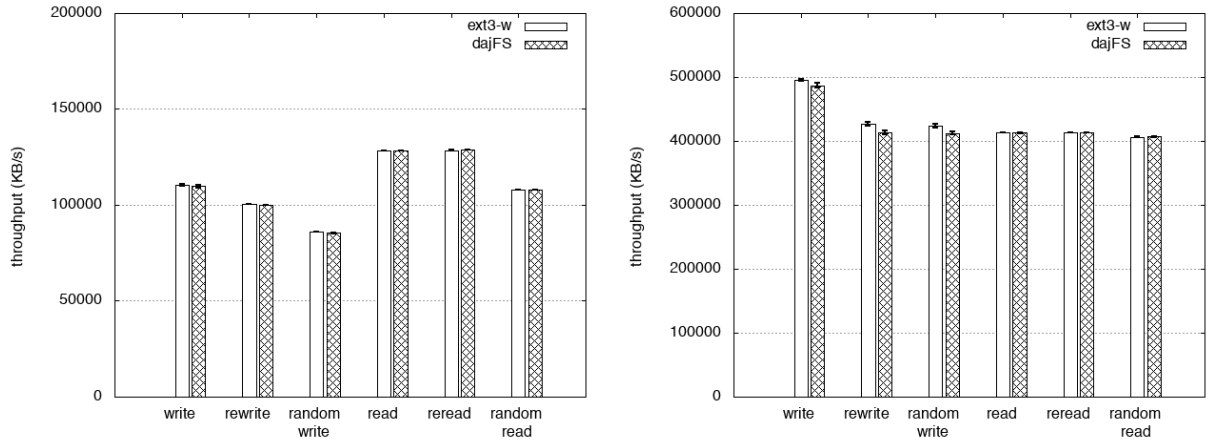
IOzone は，ファイルシステムの読み書き性能を計測するベンチマークツールの一つである．書き込み (write)，上書き (rewrite)，ランダム書き込み (random write)，読み込み (read)，再読み込み (reread)，ランダム読み込み (random read) の各テストを実行した．IOzone では，各テ

表 6.1 評価実験に使用したシステムの構成

ハードウェア	CPU	Intel Core i5-4570 3.20GHz 8M cache
	メモリ	8GB
	HDD	500GB Hitachi Serial ATA/3.0 7200rpm
	SSD	120GB Intel SSD 520 Series
ソフトウェア	Linux	Ubuntu16.04 x86_64
	カーネル	Kernel 4.2

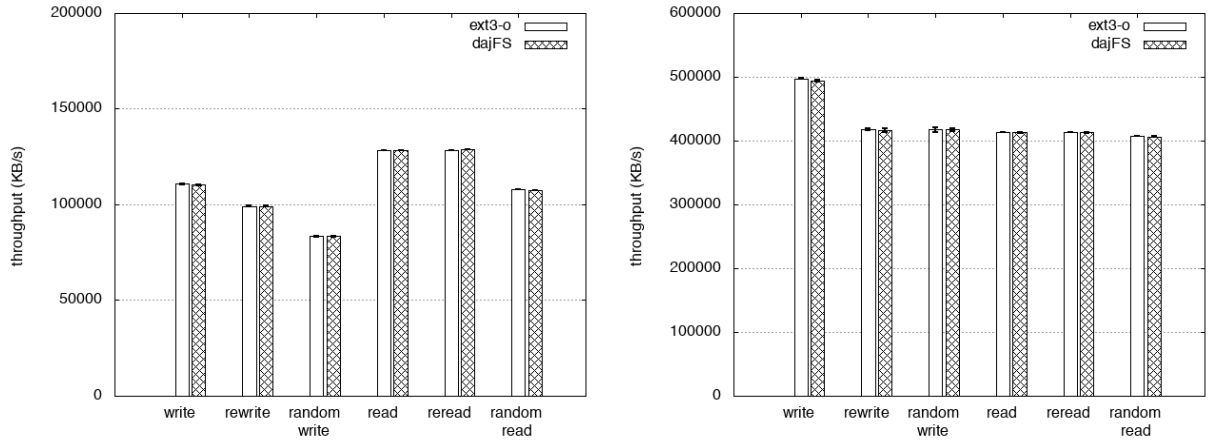
[*1] http://www.iozone.org/

[*2] https://www.coker.com.au/bonnie++/



(a) HDD (b) SSD

図 6.1　IOzone よる writeback モードの性能比較

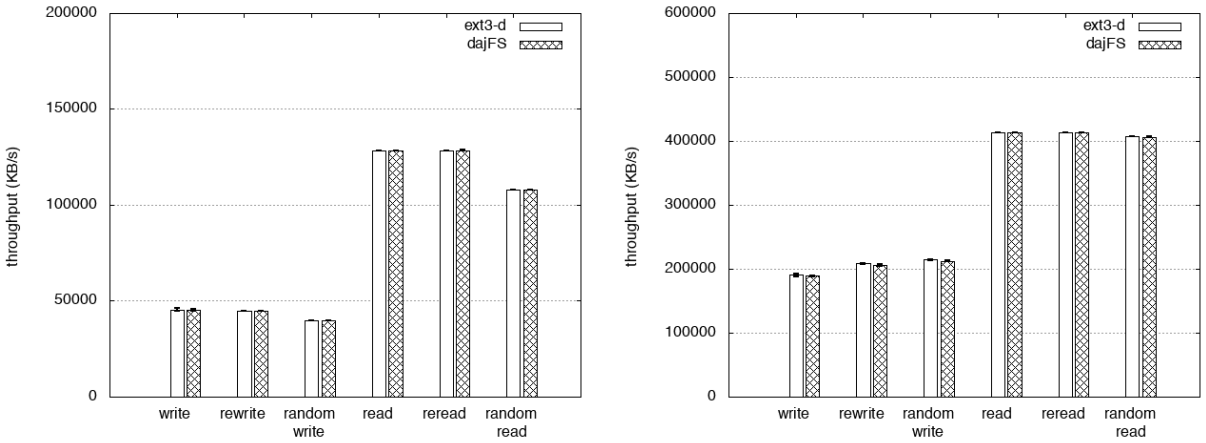


(a) HDD (b) SSD

図 6.2　IOzone よる ordered モードの性能比較

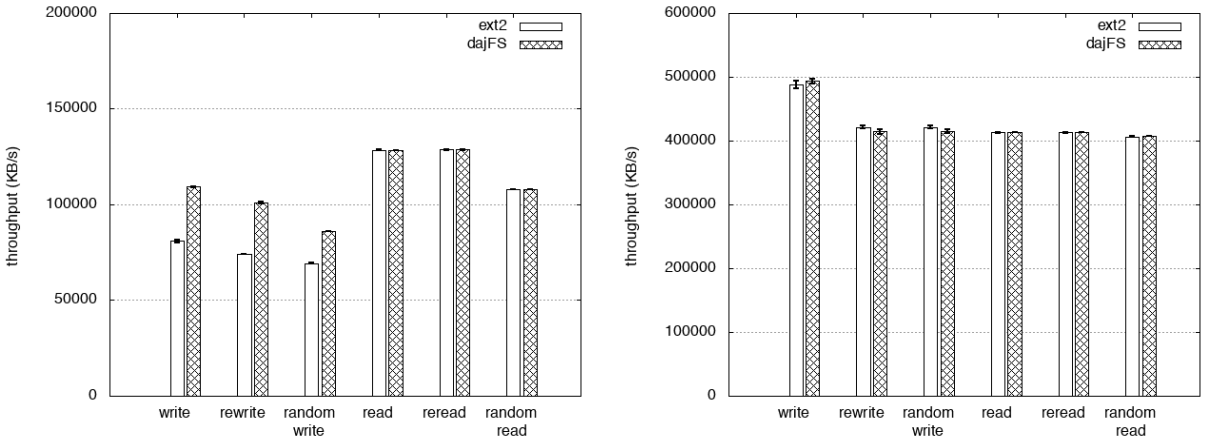
ストにおいて性能 (KB/s) を測定する．ファイルサイズは 16GB，1 回の write システムで書き出す単位を 8MB とした．実行結果には close，fsync，fflush の時間も含まれる．

ext2, ext3-w, ext3-o, ext3-d との比較結果を図 6.1 (writeback モード), 図 6.2 (ordered モード)，図 6.3 (data モード)，図 6.4 (none モード) に示す．まず，writeback モードに，ordered モード，data モードついて考察する．図 6.1，図 6.2，図 6.3 より，dajFS は HDD と SSD の双方で，書き込み性能 (書き込み，上書き，ランダム書き込み) に関して，ext3 との間に差は見られなかった．同様に，読み込み性能 (読み込み，再読み込み，ランダム読み込み) にも差は見られなかった．このことから，提案手法を導入したことによるオーバーヘッドはほとんどないことが示された．



(a) HDD　　(b) SSD

図 6.3　IOzone よる data モードの性能比較



(a) HDD　　(b) SSD

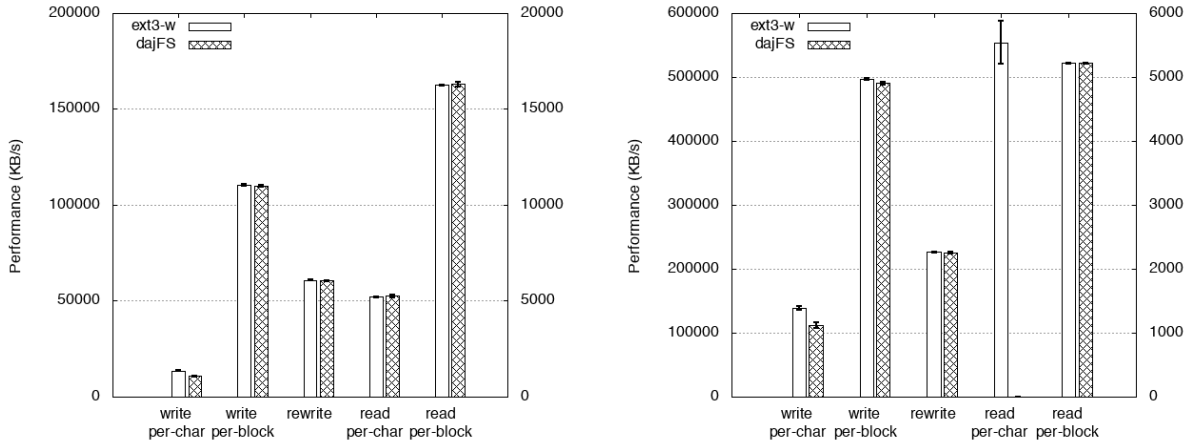
図 6.4　IOzone よる none モードの性能比較

次に，本システムが新たに追加したモード none について考察する．図 6.4 より，none モードは HDD と SSD の双方においても，読み込み性能に差は見られなかった．また，SSD において，none モードは ext2 と書き込み性能に差は見られなかった．しかし，HDD において，none モードの書き込み性能が ext2 を大きく上回った．これは，ext3 から新たに追加されたツリーベースのディレクトリインデックスの導入 [21] などによって，ディスクのシーク時間が削減されたことが原因ではないかと考えられる．

6.1.2 Bonnie++

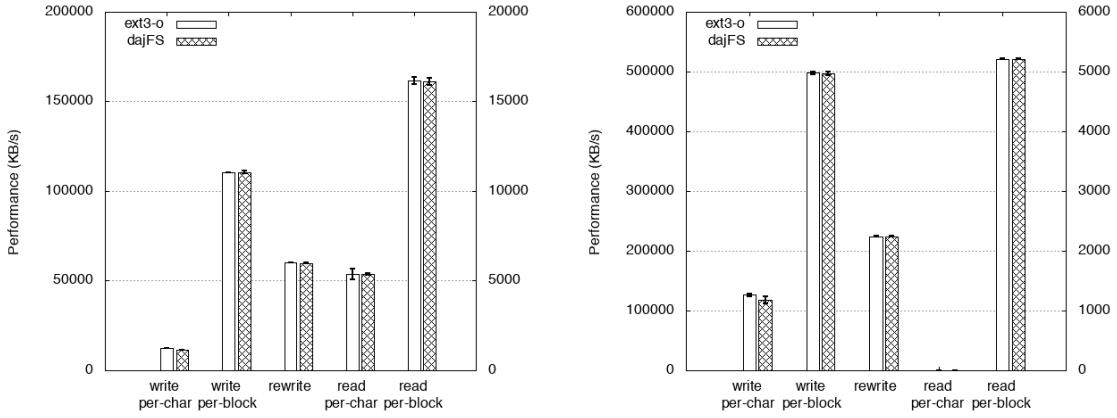
Bonnie++ は，ファイルシステムベンチマークツールの一つである．本実験では，文字単位の書き込み (write per-char)，ブロック単位の書き込み (write per-block)，ブロック単位の上書き (rewrite)，文字単位の読み込み (read per-char)，ブロック単位の読み込み (read per-block) の各テストを実行した．Bonnie++ では，各テストにおいて性能 (KB/s) を測定する．ファイルサイズは 16GB とし，write 時にバッファキャッシュを使用せずに計測した．

ext3-w，ext3-o，ext3-d，ext2 との比較結果を図 6.5，図 6.6，図 6.7，図 6.8 に示す．ただし，write per-char，read par-char は測定値が小さすぎるため，HDD では測定値を 10 倍，SSD で



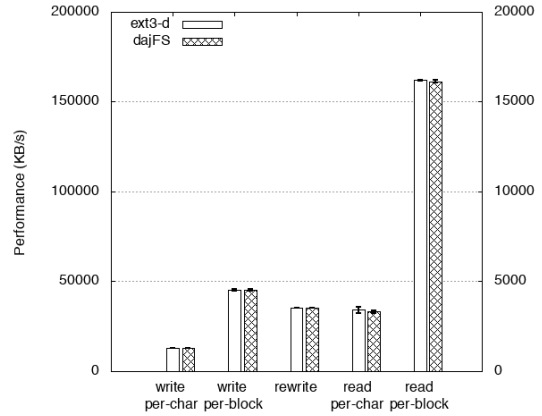
(a) HDD　　(b) SSD

図 6.5　Bonnie++ による writeback モードの性能比較

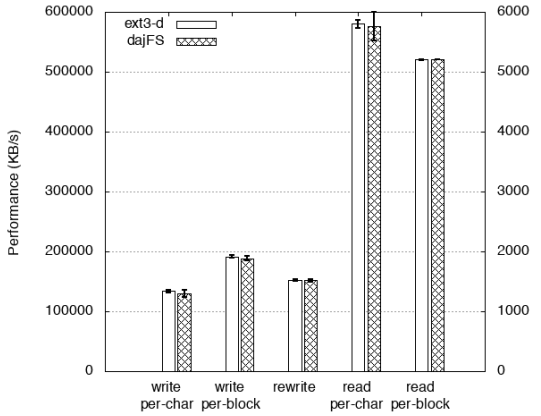


(a) HDD　　(b) SSD

図 6.6　Bonnie++ による ordered モードの性能比較

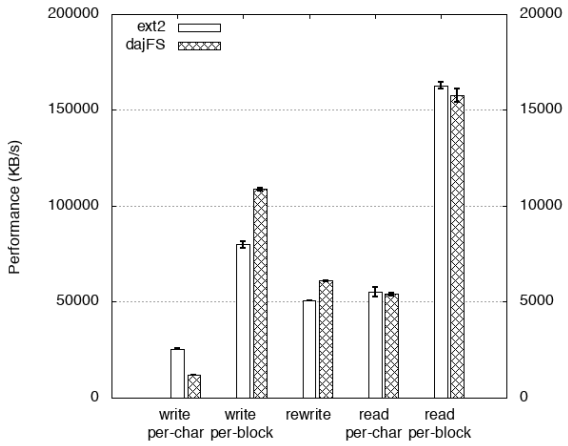


(a) HDD

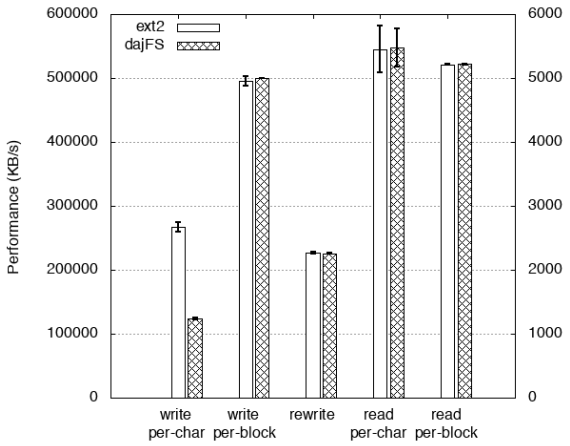


(b) SSD

図 6.7　Bonnie++ による data モードの性能比較



(a) HDD



(b) SSD

図 6.8　Bonnie++ による none モードの性能比較

は 100 倍で示している (グラフ右軸)．また，測定にかかった時間が短すぎるため，Bonnie++ では ext2 の read seq-file，ext3-o の read per-char や dajFS (ordered モード) の read per-char については測定できなかった．そのため，それらのグラフは表示していない．

まず，writeback モードに，ordered モード，data モードついて考察する．dajFS は HDD と SSD の双方で，ext3 と書き込み性能 (文字単位の書き込み，ブロック単位の書き込み，ブロック単位の上書き) に差は見られなかった．同様に，読み込み性能 (文字単位の読み込み，ブロック単位の読み込み) に差は見られなかった．

一方，none モードの書き込み性能には，大きな違いが見られた．HDD の場合， dajFS の文字単位の書き込みの性能は ext2 の性能の 50% 程度となってしまった．これは，ext2 にジャーナリングを導入したことによって，書き込み回数が増加したからではないかと考えられる．また，

dajFS のブロック単位の書き込みと上書きの性能は ext2 を大きく上回った．これは，IOzone と同様に ext3 から新たに追加された機能によって，ディスクのシーク時間が削減されたのではないかと考えられる．SSD の場合， dajFS の文字単位の書き込みの性能は ext2 の性能の 50% 程度となってしまった．こちらも HDD の場合と同様に，ext2 にジャーナリングを導入したことによる書き込み回数の増加が原因ではないかと考えている．

6.1.3 まとめ

none モードにおける文字単位の書き込みを除き，提案手法によるオーバーヘッドはほとんどないと言える．また，文字単位の書き込み測定値はシステムの性能に影響を与えるレベルではないと言える．以上のことから，dajFS は既存システムとほぼ同じ性能で，問題なく使用できることが示された．

6.2 モード設定単位の妥当性

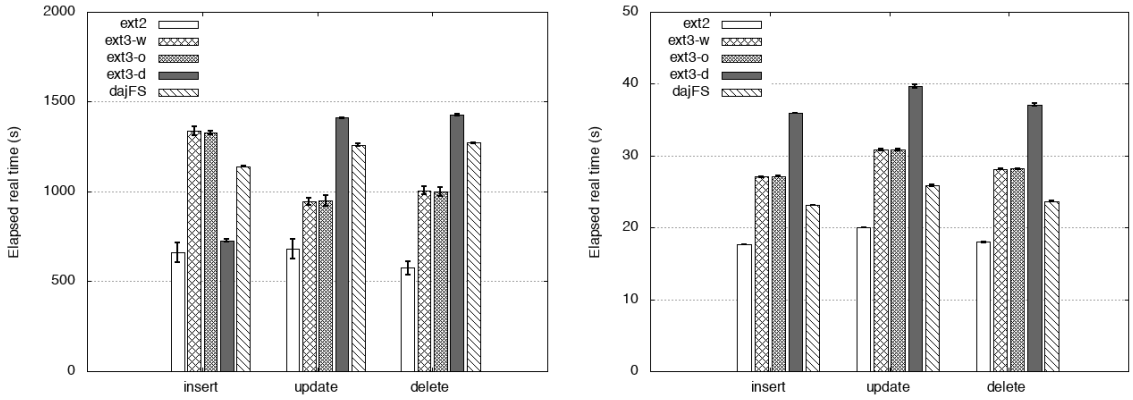
提案手法におけるジャーナリングモード設定単位の妥当性を確認するために，システムの整合性を損なわずに一部のディレクトリのモードを変更したときの実行時間を比較した．ここで，ベンチマークとして，SQLite[*3]，MySQL[*4]を用いた．

6.2.1 SQLite

SQLite は，オープンソースで公開されている関係データベース管理システムのひとつであり，システム独自でデータの整合性を確保している．これは，トランザクションの開始時にロールバックジャーナルファイルと呼ばれる一時ファイルを作成し，異常終了が発生した際にデータベースのロールバックするというものである．

SQLite5.7 でも ext3 と同様に，ロールバックジャーナルファイルの書き込み方法を変更するジャーナルモード (以後，SQLite ジャーナルモード) が存在する．SQLite ジャーナルモードは大きく分けて delete モードと wal モードの 2 つのモードが提供されている．delete モードでは，トランザクション開始から終了までの更新内容をロールバックジャーナルファイルに書き込み，コミット時に本体のデータベースファイルに用いて更新内容を反映する．wal モードでは，トランザクション開始から終了までの更新内容をロールバックジャーナルファイルに書き込み，データベースを切断するまで本体のデータベースファイルに更新内容を反映しない．しかし，どのモードにおいても，ロールバックジャーナルファイルと ext3 のジャーナルファイルで書き込みが重複してしまう．そこで，データベースを保存するディレクトリを data モード，ロールバッ

[*3] https://www.sqlite.org/
[*4] https://www.mysql.com/jp/



(a) HDD　　(b) SSD

図 6.9　SQLite の性能比較 (delete モード)

表 6.2　delete モードにおける SQLite の実行時間 (s) の比較

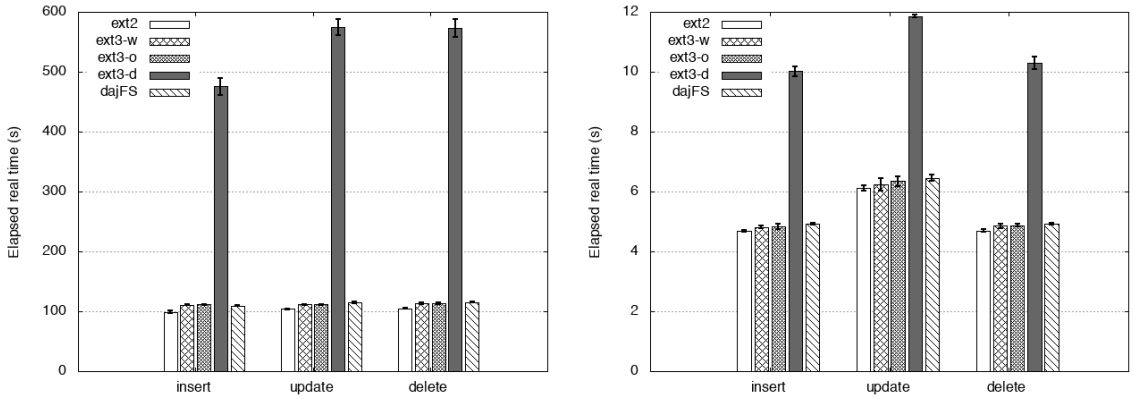
	HDD			SSD		
	insert	update	delete	insert	update	delete
ext3-d	728.58	1,411.42	1,428.22	35.98	39.70	37.11
dajFS	1,140.39	1,260.64	1,272.42	23.14	25.88	23.64
比率	157%	89%	89%	64%	65%	64%

クジャーナルファイルを保存するディレクトリを none モードに設定した．

整数型のデータと文字列型のデータを保持するテーブルにおいて，10,000 件のランダムなデータを挿入 (insert)，10,000 件のデータをランダムに更新 (update)，10,000 件のデータをランダムに削除 (delete) した時の実行時間を計測した．

SQLite ジャーナルモードが delete モードの場合と wal モードの場合の実験結果を，それぞれ図 6.9，図 6.10 に示す．また，各 SQLite ジャーナルモードにおける ext3-d と，ロールバックジャーナルファイルを none モードにした dajFS との実行時間 (s) の比較を表 6.2，表 6.3 に示す．

delete モードの実験の場合，ロールバックジャーナルファイルを none モードにした dajFS は，データの更新で HDD では 11%，SSD では 35%，データの削除で HDD では 11%，SSD では 36% の性能が改善された．しかし，HDD でのデータ挿入について，dajFS は ext3-d より性能が悪化した．ext3-d は，実データもジャーナリングの対象にすることで，ブロック単位のシーケンシャルな書き込みにとなったため速くなったことが原因ではないかと考えられる．．HDD は SSD と比較して，シーケンシャルな書き込みに比べて，シーケンシャルではない書き込みにかかる時間は遅いため，このような結果が得られた．一方，wal モードの場合，dajFS は ext2 と同等の性能を持ち，HDD の場合では ext3-d と比較して 80% 以上の性能改善が見られた．こ



(a) HDD (b) SSD

図 6.10　SQLite の性能比較 (wal モード)

表 6.3　wal モードにおける SQLite の実行時間 (s) の比較

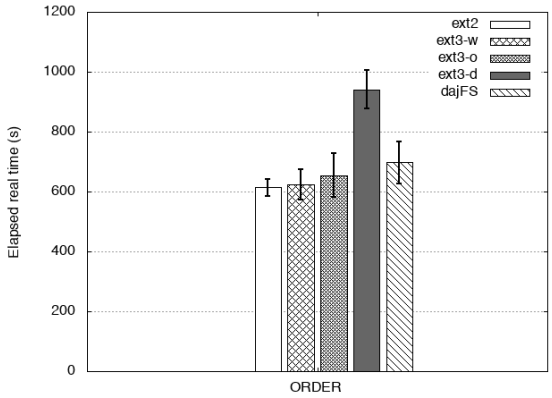
	HDD			SSD		
	insert	update	delete	insert	update	delete
ext3-d	475.66	574.53	572.65	10.02	11.86	10.29
dajFS	109.42	114.65	115.57	4.92	6.46	4.94
比率	23 %	20%	20%	49%	55%	48%

れは wal モードは，データベース本体への書き込みはデータベースを切断するまで書き込まないことと，ext3-d の性質が原因ではないかと考えられる．wal モードは，delate モードと比べてデータベースの書き込みが競合することがなく，高速に書き込むことができた．しかし ext3-d は，実データをジャーナリングを行う対象とするため，ロールバックジャーナルファイルの書き込みでオーバーヘッドが大きくなった．dajFS では，ロールバックジャーナルファイルを none モードとすることで，上記のオーバーヘッドを小さくすることができた．

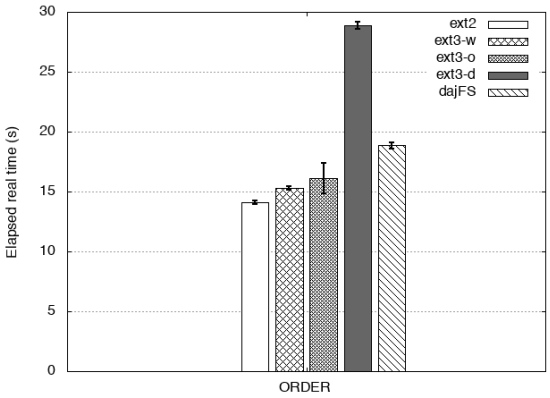
以上のことから，ディレクトリというモードの設定単位は妥当であり，適切なモードの設定を行うことでシステムの信頼性を損なわずにシステムの性能を向上させることができる．

6.2.2　MySQL

MySQL も，オープンソースで公開されている関係データベース管理システムであり，データの整合性の確保以外にも，処理の高速化のために一時ファイルを任意に生成することができる．例えば，ORDER BY や GROUP BY などが含まれている SQL 文を発行する際に，クイックソートの結果を一時ファイルに書き出す．しかし，一時ファイルはデータベース本体と比べてファイルの重要度は低い．そこで，`TMPDIR` 環境変数を利用して，SQL 文の発行時に生成される



(a) HDD



(b) SSD

図 6.11　MySQL の性能比較

表 6.4　MySQL の実行時間 (s) の比較

	HDD	SSD
ext3-d	941.22	26.86
dajFS	697.65	18.87
比率	88%	70%

一時ファイルを writeback モードに設定されたディレクトリに保存し，データベースファイルを保存するディレクトリを data モードに設定する．

実験には，整数型のデータ，文字列型のデータ、日付型のデータを保持するテーブルにおいて，10,000,000 件のランダムなデータに対して日付順にソートする時にかかる実行時間を計測した．

MySQL の実験結果を図 6.11 に示す．ext3-d と，一時ファイルを writeback モードにした dajFS との実行時間 (s) の比較を表 6.4 に示す．HDD の場合は，dajFS を用いて一時ファイルのジャーナリングモードを変更することで，ext3-d と比較して実行時間が 12% 削減された．また，SSD の場合は，dajFS を用いて一時ファイルのジャーナリングモードを変更することで，ext3-d と比較して実行時間が 30% 削減された．これらは data モードでは，MySQL で ORDER BY が含めれている SQL 文を発行した時に，一時ファイルの実データをジャーナリングの対象としたことがオーバーヘッドとなった．そこで dajFS では，一時ファイルのジャーナリングを行わないことで，オーバーヘッドを削減することができた．

6.2.3　まとめ

実験から，システム全体の信頼性を損なわずに性能を向上させることができた．このことからも，提案手法におけるモード設定の単位は妥当であると言える．また，提案手法によるモー

ドの設定を適切に行うことで，性能を大きく向上させることもできる．ただし，HDD における SQLite のデータ挿入のように性能が悪化することもあることがわかった．従ってユーザは，アプリケーションの動作を把握した上でディレクトリにジャーナリングモードを設定する必要がある．

6.3 ファイルシステムの信頼性

dajFS が整合性を確保できるかどうかを確認するため，システムクラッシュの実験を行った．本実験では，事前に設定したファイル (対象ファイルと呼ぶ) の I/O 要求中であれば，一定時間経過後に故意にカーネルパニックを発生させるカーネルスレッドを新たに実装した．カーネルスレッドを稼働させることにより，dajFS 内のファイルの書き込み中にカーネルパニックを発生させ，ファイルシステムの状態を確認する．本実験では，ファイルを新規作成し，1MB のデータを書き込んだ後に，カーネルパニックを発生させる．ファイルシステムの状態の確認には，ファイルシステム解説ツール The Sleuth Kit[*5]を用いた．

表 6.5 は，対象ファイルに設定するジャーナリングモード (writeback，ordered，data，none) に対して各 50 回カーネルパニックを発生させ，ファイルシステムの状態を示した結果である．ここで，ファイルシステムの状態を以下の 4 つに分類した．

State 1: ファイルシステムが破損していない．
ここで，ファイルシステムが破損していないとは，i ノードビットマップとデータビットマップが正常な値であることを意味する．

State 2: 書き込みを行ったファイルが破損していない (ext3-w と同等)．
ここで，ファイルが破損していないとは，i ノードが正常な値であることを意味する．

State 3: 書き込みを行ったファイルが文字化けしていない (ext3-o と同等)．
ここで，ファイルが文字化けしていないとは，i ノードが正常な実データを参照していることを意味する．

表 6.5 信頼性の評価

モード	State 1	State 2	State 3	State 4
none	✓			
writeback	✓	✓		
ordered	✓	✓	✓	
data	✓	✓	✓	✓

[*5] https://www.sleuthkit.org/

State 4: ファイルのデータが書き込み前か書き込み後のどちらかである (ext3-d と同等).
ここで，データが書き込み前か書き込み後のどちらかであるとは，データブロックが正常に書き込みができることを意味する.

dajFS では，全ての場合において，適切な整合性が確保されていることが確かめられた.

7 おわりに

本研究では，既存ファイルシステムにおけるジャーナリングモードの設定粒度が粗く，個々のファイルの重要度に対応できないという問題を解決するために，ディレクトリ毎にモードを設定できるファイルシステム dajFS を提案した．ジャーナリングモードをディレクトリ単位で設定することで，既存研究のような設定の煩雑さがなく，ファイルの重要度に対応した設定が可能である．

dajFS は，ext3 と比較してもオーバーヘッドはほとんどなく，既存システムと同様に使用できることが示された．また，ディレクトリ単位でジャーナリングモードを設定できる恩恵は大きかったと考えられる．アプリケーション独自でファイルの整合性を保証する SQLite に提案手法を試したところ，実行時間が最大で 80% 削減した．また，一時ファイルが多く作られる MySQL でも，実行時間が最大で 30% 削減した．以上のことから，提案手法の有用性を確認することができた．

今後の展望として，dajFS の正当性を保証しなければならないと考えられる．正当性の保証には，ファイルシステムをモデル化する手法 [22] が存在する．Oral ら [23] は，ext3，ReiserFS，JFS といったジャーナリングファイルシステムをモデル化し，これらのファイルシステムにバグがあることを指摘した．このように，広く一般で用いられているファイルシステムでもバグが見つかることもあるため，正当性の保証は大変重要である．dajFS は，ext3 を元に実装されたファイルシステムであるため，上であげた先行研究の結果を応用することでシステムの正当性を証明することができるものと期待できる．

また，dajFS を他のデバイスや環境下に対応させることが望まれる．例えば，dajFS は一つのブロックデバイスにマウントするディスクファイルシステムであるが，クラスタファイルシステムや分散ファイルシステムに対応させることが考えられる．近年では，大型ストレージにクラスタファイルシステムや分散ファイルシステムを用いて管理することに注目が集まっている．クラスタファイルシステムにはジャーナリングが導入されているものもあるが，これらのファイルシステムは，ジャーナリングの処理がボトルネックとなることが多い．Hatzieleftheriou ら [24] は，この問題を解決するため，クラスタファイルシステム用のジャーナリング処理を実装した．dajFS でも，クラスタファイルシステムに対応し，ジャーナリング処理を拡張することで利用の幅が広がると考えられる．

また，RAID [25] を考慮した設計の導入も考えられる．RAID はジャーナリングが確保することのできないハードウェア障害を復旧することができる．そのため，RAID 構成されたシステムに対してジャーナリングファイルシステムを利用することは，有用である．しかし，これらの手法は双方を使うことを想定されて実装されておらず，無駄な I/O 書き込みが発生してしまう．Denehy ら [26] は，新しくジャーナリングモード declared を導入して，性能を向上させた．現

在の dajFS は，4 つのモードのみ提供しているが，上記のようなモードを追加して，様々なシステム環境や要求に対応できるようにすれば，より有用性が増すだろう．

参考文献

[1] Kowalski, T. J.: Fsck - the UNIX File System Check Program, *UNIX Vol. II* (Hume, A. G. and McIlroy, M. D., eds.), W. B. Saunders Company, pp. 581–592 (1990).

[2] Hagmann, R.: Reimplementing the Cedar File System Using Logging and Group Commit, *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, New York, NY, USA, pp. 155–162 (1987).

[3] McKusick, M. K. and Ganger, G. R.: Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem, *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, Berkeley, CA, USA, pp. 24–24 (1999).

[4] Rosenblum, M. and Ousterhout, J. K.: The Design and Implementation of a Log-structured File System, *ACM Trans. Comput. Syst.*, Vol. 10, No. 1, pp. 26–52 (1992).

[5] Tweedie, S. C.: Journaling the Linux ext2fs Filesystem (1998).

[6] Best, S.: JFS for Linux, `http://jfs.sourceforge.net/` (2004).

[7] Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M. and Peck, G.: Scalability in the XFS File System, *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, Berkeley, CA, USA, pp. 1–1 (1996).

[8] Reiser, H.: ReiserFS, `https://reiser4.wiki.kernel.org/index.php/Main_Page` (2004).

[9] Solomon, D.: *Inside Windows NT 2nd ed*, Microsoft Press (1998).

[10] Prabhakaran, V., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H.: Analysis and Evolution of Journaling File Systems, *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, Berkeley, CA, USA, pp. 8–8 (2005).

[11] Rocha, P. E. and Bona, L. C. E.: Analyzing the Performance of an Externally Journaled Filesystem, *Proceedings of the 2012 Brazilian Symposium on Computing System Engineering*, SBESC '12, Washington, DC, USA, pp. 93–98 (2012).

[12] McKusick, M. K., Joy, W. N., Leffler, S. J. and Fabry, R. S.: A Fast File System for UNIX, *ACM Trans. Comput. Syst.*, Vol. 2, No. 3, pp. 181–197 (1984).

[13] Chidambaram, V., Pillai, T. S., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H.: Optimistic Crash Consistency, *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, New York, NY, USA, pp. 228–243 (2013).

[14] Lim, S.-H., Choi, H. J. and Park, D.-S.: Efficient Journaling Writeback Schemes for Reliable and High-performance Storage Systems, *Personal Ubiquitous Comput.*, Vol. 17,

No. 8, pp. 1761–1774 (2013).

[15] Hatzieleftheriou, A. and Anastasiadis, S. V.: Okeanos: Wasteless Journaling for Fast and Reliable Multistream Storage, *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC '11, Berkeley, CA, USA, pp. 19–19 (2011).

[16] Nightingale, E. B., Veeraraghavan, K., Chen, P. M. and Flinn, J.: Rethink the Sync, *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, Berkeley, CA, USA, pp. 1–14 (2006).

[17] Shen, K., Park, S. and Zhu, M.: Journaling of Journal is (Almost) Free, *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST '14, Berkeley, CA, USA, pp. 287–293 (2014).

[18] Hipp, D. R.: SQLite Write-Ahead Logging, `https://www.sqlite.org/wal.html` (2010).

[19] D. Quinlan, P. R. and Yeoh, C.: Filesystem Hierarchy Standard, `https://refspecs.linuxfoundation.org/fhs.shtml` (2015).

[20] Traeger, A., Zadok, E., Joukov, N. and Wright, C. P.: A Nine Year Study of File System and Storage Benchmarking, *Trans. Storage*, Vol. 4, No. 2, pp. 5:1–5:56 (2008).

[21] Tweedie, S.: EXT3, journaling filesystem (2000). `http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html`.

[22] Yang, J., Twohey, P., Engler, D. and Musuvathi, M.: Using Model Checking to Find Serious File System Errors, *ACM Trans. Comput. Syst.*, Vol. 24, No. 4, pp. 393–423 (2006).

[23] Arpaci-Dusseau, A. C.: Model-Based Failure Analysis of Journaling File Systems, *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, Washington, DC, USA, pp. 802–811 (2005).

[24] Hatzieleftheriou, A. and Anastasiadis, S. V.: Host-side Filesystem Journaling for Durable Shared Storage, *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST '15, Berkeley, CA, USA, pp. 59–66 (2015).

[25] Patterson, D. A., Gibson, G. and Katz, R. H.: A Case for Redundant Arrays of Inexpensive Disks (RAID), *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, New York, NY, USA (1988).

[26] Denehy, T. E., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H.: Journal-guided Resynchronization for Software RAID, *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST '05, Berkeley, CA, USA, pp. 7–7 (2005).

謝辞

本研究を行うに当たり，終始ご指導，ご助言を頂きました岩崎英哉教授，中野圭介准教授に深く感謝致します．また，研究を進めるにあたって熱心なご指導を頂きました研究室の皆様に心から御礼申し上げます．