基般

① ファイルシステム技術の最前線
―ファイルシステムの基礎から最先端ファイルシステムまで―

山口実靖（工学院大学）

ファイルシステムの役割

OS（Operating System）には，通常はファイルシステムというソフトウェアが含まれている．ファイルシステムは，ユーザに「ファイル」という概念や機能を与えるソフトウェアである．ファイルシステムの具体的な例としては，Windows で用いられている NTFS や，Linux で用いられている ext2/3/4 や XFS などがある．本章にて，まず「ファイル」について整理して解説し，続いてファイルシステムについて説明する．

ファイルとは，情報（ビットやバイト）の集まりをひとまとめにしたもの，あるいはその情報の入れ物である．プログラムで扱われる「変数」とは異なり，ファイルは恒久的であり，大量のデータを格納でき，複数のプロセスがアクセスできる．ソケット通信やパイプなどのプロセス間通信もファイルの仕組みを利用しているがファイルシステムの関与は小さいため，本稿ではこれは扱わず HDD（Hard Disk Drive）や SSD（Solid State Drive）などの永続化ストレージに格納されるファイルのみを解説する．また，本稿では「ストレージ」という表現を用いるがこれはファイルシステムを構築してファイルを格納する対象であり，具体的には HDD，SSD，USB メモリなどを想像すれば問題ない．

ファイルは，通常はファイルのデータ（中身）のほかにファイルの属性情報（メタデータと呼ばれる）を保持している．ほぼすべてのファイルシステムの実装が保持している典型的なメタデータとしてファイル名，ファイルサイズ，最終更新日時などがあり，少なくないファイルシステムが作成者や所有者，アクセス制限に関する情報なども有している．

ファイルシステムは，ユーザに「ファイル」という仕組みを提供するソフトウェアである．コンピュータ（ストレージデバイス）はハード的にはファイルという仕組みを有していないため，ファイルシステムがソフトウェア的にこれを提供している．HDD や SSD の装置は SCSI（Small Computer System Interface，スカジー．計算機と周辺機器間のデータ通信の規格の 1 つ）等のブロックレベルのアクセス要求しか受けつけず，ファイルレベルの要求は受け付けない．ブロックレベルのアクセス要求は「アドレス XX セクタから，XX セクタ分，読み込み」などの要求である．ストレージデバイスは「ファイル A.txt の XX バイト目から読み込み」というファイルの概念を前提としたアクセス要求を処理することはできない．

一方，通常のユーザはある情報を格納アドレスと関連させて管理することは苦手であり，情報を名前のついたファイルに格納する方法を好む．よって，ファイルシステムの主たる目的はユーザにファイルという機能を提供することであり，換言すると**図 -1** のようにブロックレベルのストレージ装置と，ファイルレベルのアクセスを行うユーザを仲介することになる．

ストレージとの比較においては，ファイルは名前を持っており，各ファイルが格納場所において独立している．格納場所において独立とは，あるファイルの情報の量を増やしていきあるファイルを拡大していっても，別のファイルの情報に影響を与えないということであり，ストレージはこのような機能は有していない．**図 -2** に，ファイル内アドレス（通常オフセットと呼ばれる）と，ストレージアドレスの関係の例を示す．あるユーザが名前が a.c でサイズが 2 ブロックのファイルを作成すると，ファイルシステムがそのファイルのブロック 2 個をスト

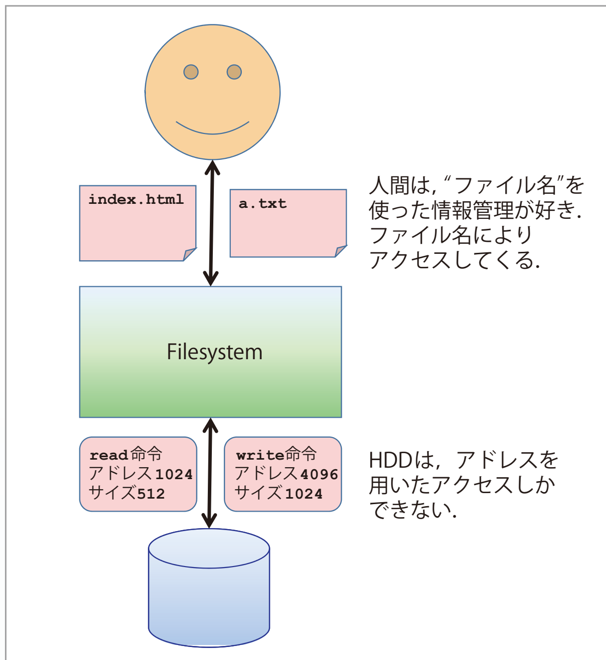

図 -1　ファイルレベルアクセスとブロックレベルアクセス

レージのいずれかのブロックに対応させて格納する．図 -2 の例では，図の (a) のようにファイルのブロック 0, 1 とストレージデバイスのブロック 0, 1 が対応している．続いて (b) のように，ユーザが名前が b.txt でサイズが 2 ブロックのファイルを作成し，同様にファイルシステムがストレージ上にこのファイル用の 2 ブロックを確保している．図の例では，ストレージのブロック 2，3 がファイルのブロック 0,1 に対応している．続いて，(c) のようにユーザがファイル a.c の大きさを 2 ブロックから 3 ブロックに拡大させると，ファイルシステムは同ファイルのブロック 2 を格納するブロックをストレージ上に確保する．同図 (c) の例では，ストレージのブロック 4 が用いられている．このように，物理ストレージ上で非連続的にファイルのデータが格納されていても，ユーザは a.c というファイルを 1 つの連続した情報として認識することができる. 当然, あるファイルを拡大することにより別のファイルの情報が上書きされることはない．また，ブロックサイズはほとんどの場合 4 キロバイトである．

ファイルは，アクセスする前に open する必要があり, アクセス後に close を行うが, これはあるファイルを今後用いることや使用を終えたことの OS への宣言である．多くの OS やファイルシステムの実装では，open の宣言をされたファイルの情報（後述する inode など）をメモリに格納するようになっている．

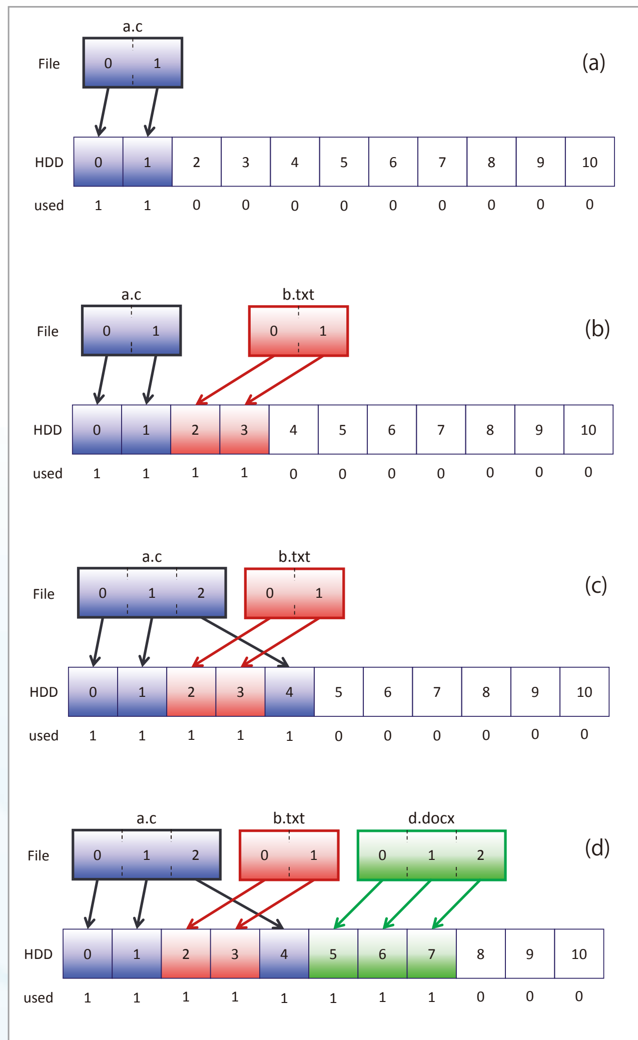

図 -2　ファイルのストレージへの格納の例

ファイルシステムの技術

ファイルシステムには多くの機能が存在している．本章にて，Journaling 機能の紹介を行う．

ファイルシステムがストレージに書き込みを行っている途中にコンピュータがクラッシュするなど，ファイルシステムの書き込みが正常に終了されないと，ファイルシステムはインコンシステントな（一貫性のない）状態に陥ることがある．たとえば，メタデータ領域が確保されたが，そのデータがどの

ファイルからも参照されていない状況などになる．メタデータ領域がインコンシステントな場合などは，ファイルの中身のデータにアクセスできなくなることもある．

このようなインコンシステントなデータを発見するために，OSはファイルシステム全体をスキャンするプログラム（Unix系OSにおけるfsckやMS-DOS/Windows系OSにおけるchkdskやscandiskなど）を実行する．ファイルシステムのスキャンは，ファイルシステムのサイズが大きい場合は非常に多くの時間を要する（数時間や数十時間かかるとの報告[1]もある）．この膨大なスキャン時間を短縮するための機能としてジャーナルがある．

ジャーナル機能を有するファイルシステムでは，メタデータへの変更を書き込む前にジャーナルにその変更内容を表す情報を書き込み，その後にファイルシステムに書き込みを行う．システムのクラッシュなどによりファイルシステムが正常に終了しなかった場合は，スキャンするプログラムはジャーナルを確認し，書き込みが正常に完了していない可能性がある場所を特定する．そして，書き込みを完了させたり，ロールバックさせたりする．正常に書き込みが行われていない可能性がある場所が分かるため，圧倒的に短い時間でファイルシステムのインコンシステントな場所を発見できる．ジャーナルは，インコンシステントな場所の発見を短時間でできるようにするためのものであり，書き込まれたデータを守るためのものではない．たとえばロールバックされた場合は，ユーザはクラッシュ前にファイルの書き込みが完了したとの通知をOSから受けるが，実際は書き込みは行われておらず，その書き込み内容は破棄されてしまうことになる．

ファイルシステムの実装

■ パーティション

ストレージデバイスのレイアウトとパーティションについて述べる．本節における「ストレージ」は主にHDDやSSDを想定している．USBメモリに

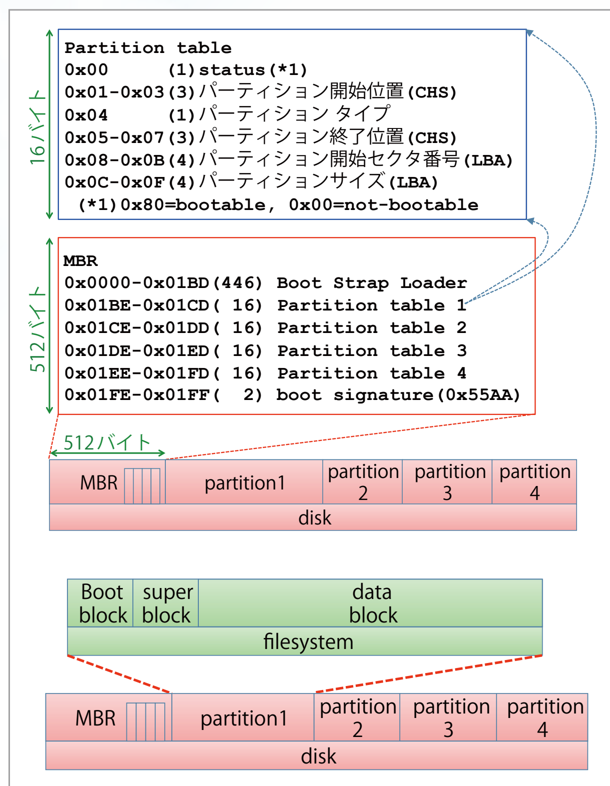

図-3　MBRとパーティション

関しては，Linuxなどはパーティショニングをサポートしているが，WindowsなどのOSはパーティショニングをサポートしていない．

ストレージデバイスは複数の「パーティション」に分割して使用することができる．たとえば，4テラバイトのHDDを購入した場合，それを2テラバイトのパーティション2個に分割して使うことが可能である．Windowsの場合これらはそれぞれ別のドライブ（C:とD:など）が割り当てられる．ストレージデバイスのパーティション管理手法としては，MBR（Master Boot Record）と，GPT（GUID Partition Table）がある．MBRは2テラバイト未満のパーティションのみサポートする従来の簡易な方法である．GPTは，2テラバイト以上のパーティションもサポートする手法である．本稿では，相対的に簡易なMBRの解説を行う．

ストレージデバイスは，通常512バイトのセクタ単位で扱われ，512バイトより細かい読み書きを行うことはできない．

図-3のように，HDDの最初のセクタはMBRと

```
# od -A x -t x1 -N 512 /dev/sda
000000 eb 63 90 10 8e d0 bc 00 b0 b8 00 00 8e d8 8e c0
000010 fb be 00 7c bf 00 06 b9 00 02 f3 a4 ea 21 06 00
        (略)
0001a0 0d 0a 00 bb 01 00 b4 0e cd 10 ac 3c 00 75 f4 c3
0001b0 00 00 00 00 00 00 00 00 81 29 02 00 00 00 00 20
0001c0 21 00 82 4b 81 0a 00 08 00 00 00 00 80 00 80 4b
0001d0 82 0a 83 fe ff ff 00 08 80 00 00 90 21 12 00 00
0001e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0001f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 aa
下線部 = パーティションテーブル 2
[ 80 4b 82 0a 83 fe ff ff 00 08 80 00 00 90 21 12 ]
開始セクタ (LBA) : 00 08 80 00 = 00800800(16)[sect]
        = 8390656(10)[sect] = 4195328[KB] = 4097[MB]
サイズ (LBA) : 00 90 21 12 = 12219000(16)[sect]
        = 304189440(10)[sect] = 152094720[KB] = 148530[MB]
```

図 -4　MBR のダンプ例

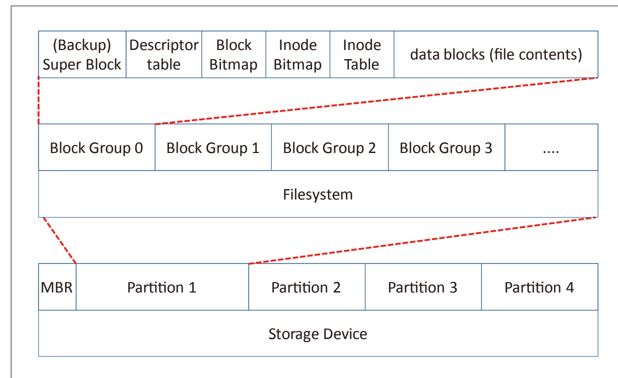

図 -5　ext2/3/4 パーティションレイアウト

呼ばれ，コンピュータの起動やパーティション情報の格納に使われる．MBR の最後部近辺の 446 バイト目から 509 バイト目までの 64 バイトにはパーティションテーブルが格納されている．基本パーティションは 4 個まで作成することができ，テーブルの各エントリは 16 バイトとなっている．各エントリはブートフラグ（ブート可能であるか否か），CHS（Cylinder Head Sector）形式におけるパーティション開始位置とタイプと終了位置，LBA（Logical Block Addressing）形式における開始位置とサイズが格納されている．CHS 形式と LBA 形式はともに，HDD の場所（セクタ番号）を表現する手法である．よりバイト数の多い LBA 形式でも最大パーティションサイズは 0xFFFFFFFF セクタであり，これは 2 テラバイト弱であり MRB 手法では 2 テラバイト以上のパーティションを作成できないことを理解できる．ストレージデバイス sda のパーティションテーブルは `fdisk -lu /dev/sda` のようなコマンドにより確認できるが，MBR のフォーマットを理解していれば**図 -4** のように `od -A x -t x1 -N 512 /dev/sda` などのダンプコマンドを用いてパーティションテーブルを直接見ることもできる．

パーティションとファイルシステムの関係は，**図 -5** の通りである．ストレージデバイスの一部にパーティションが作成され，そのパーティションの中にファイルシステムが構築される．よって，ストレージデバイスの先頭（アドレス 0 セクタ）と，ファイルシステムの先頭（ブロック 0）は，位置が一致しない．MBR があるためパーティション 1 の開始位置もストレージデバイスの先頭にはならない．

■ inode

「ファイルシステムの役割」にて記したように，ファイルシステムはファイルとそのデータの格納場所の関係を管理しなくてはならない．Unix 系の OS のほぼすべてで，この関係は inode（index-node）と呼ばれるデータ構造で管理されている．inode の仕様の詳細はファイルシステムの実装により異なるが，多くのファイルシステムにおいてファイルの属性やファイルデータを格納しているストレージのブロックアドレスを保持している．前述のように，通常は OS は open されたファイルの情報をメモリ内に保持するが，この inode 情報を保持することになる．次節で述べる ext2/3/4 などのファイルシステムでは，各 inode 情報のサイズは 128 バイトあるいは 256 バイトである．

■ ext2/3/4

ext2/3/4（Second, Third, Fourth Extended Filesystem）は Linux で広く使用されているファイルシステムである．伝統的に，多くの Linux ディストリビューションにて初期設定において ext2/3/4 が選択されており，OS インストール時に ext2/3/4 しか選択できないディストリビューションも存在した．

ext2 は初期のファイルシステム ext を拡張したものであり，ジャーナリングなどは備えていない．

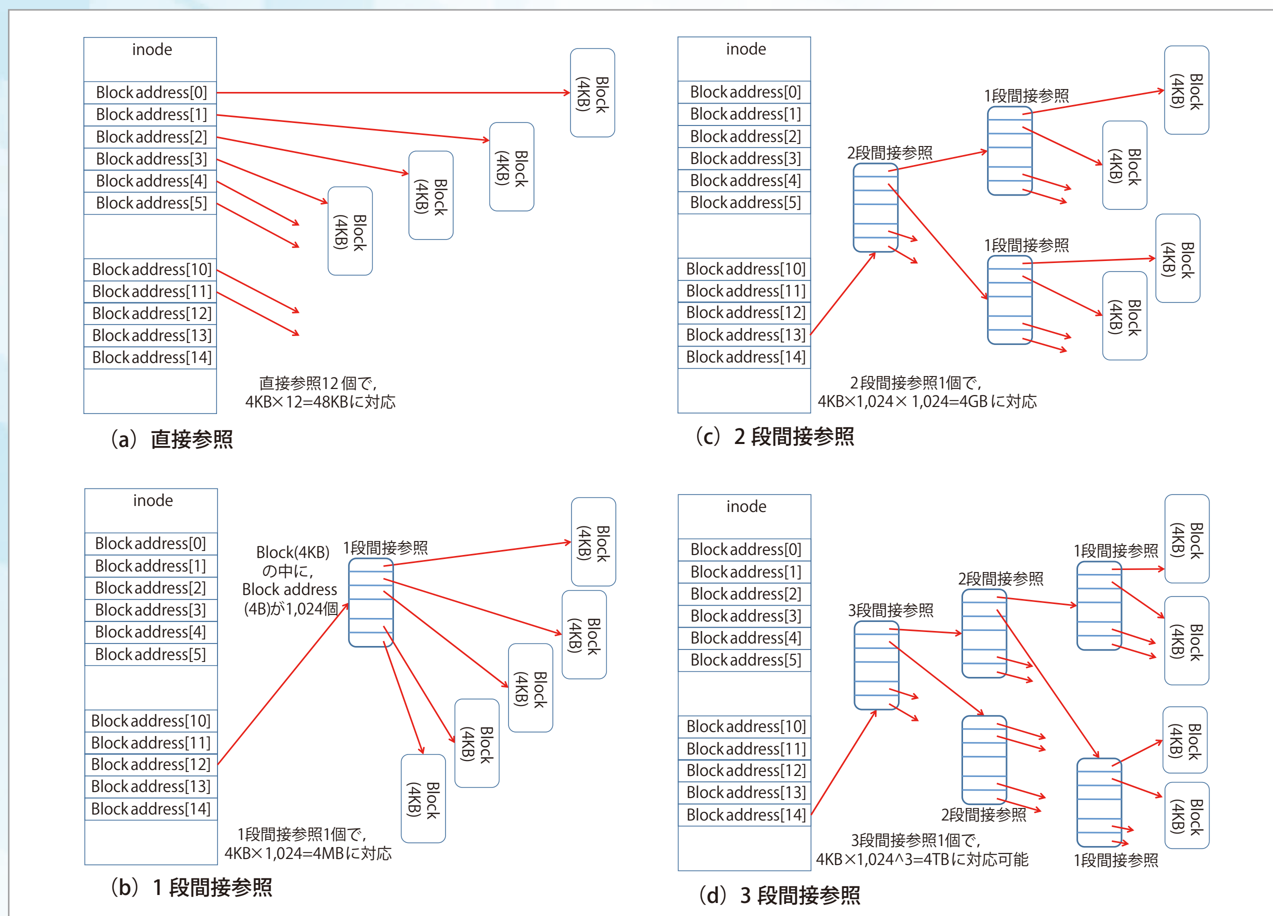

図-6　ext2/3/4 inodeのブロックアドレス

ext3は，ジャーナリング機能を備えたファイルシステムである．ext2と互換性が高く，ディスク上のデータレイアウトにも互換性がある．ext2でフォーマットされたファイルシステムをext3ファイルシステム実装でマウントしたり，その逆を行ったりすることができる．

ext2/3/4のinodeには，ファイルサイズ，ファイルモード（アクセス制限）情報，ユーザID，最終更新日時や最終アクセス時刻，ファイルのデータブロックが格納されているディスクブロックのアドレスなどが格納されている．通常，ブロックサイズは4キロバイトであり，ブロックアドレスは4バイトである．1ギガバイトのファイルは262,144個のブロックにより構成され，同ファイルのブロックアドレスの情報だけで1メガバイト（4バイト／個×262,144個）の情報が格納されることとなるが，当然inode内にこの情報を格納することはできない．ext2/3/4では，**図-6**のように直接参照ブロックと多段階接参照ブロックを用いることにより巨大なファイルに対応している．図-6(a)のように，inodeには直接参照のブロックアドレスが12個格納できる．よって，48キロバイト以下のサイズのファイルのデータブロックのアドレスはinode内に直接記述され，Block address[0]から[11]がファイル内ブロックの0から11に対応している．

次のブロックアドレス（Block address[12]）には，インデックス（1段階接参照のブロックのアドレス群）のブロックのアドレスが格納されている．Block address[12]のアドレスのブロックには，ブロックアドレスが最大で1,024個格納されており，各ブロックアドレスがファイル内ブロック12から1,035のアドレスとなっている．この1段階接参照により，1,036ブロック（約4メガバイト）以下のサイズのファイルに対応可能となる．ブロックサイズが4キロバイトであり，ブロックアドレスが4バイトであるため，インデックスには1,024個のブ

ロックアドレスが格納できることが分かる．

次の Block address[13] には，図 -6(c) のように 2 段間接参照のブロック（2 段インデックス）のアドレスが格納されている．すなわち Block address[13] に格納されているアドレスにあるブロックには，「1 段間接参照インデックス」のアドレスが 1,024 個格納されている．そして，各「1 段間接参照インデックス」にはデータブロックのアドレスが 1,024 個格納されている．よって，2 段間接参照の元では 1,024 × 1,024 個のブロックのアドレスが管理され，2 段インデックスのみで 4 ギガバイト分のファイルデータを管理できることになる．

図 -6(d) の 3 段間接参照インデックスも同様である．3 段間接参照のみで 1,024 × 1,024 × 1,024 個のブロックを管理し，4 テラバイトまで管理できる．ただし , 32 ビットの Linux 2.6 にはファイルサイズに 2 テラバイトの上限がある．

ファイル内のラインダムな位置にシークしたり，tail コマンドのように最後尾にアクセスを行ったりするような場合でも，ファイル先頭から順に辿っていく必要はなく，最大でも 3 段のポインタを辿ればファイル内の任意のブロックのディスク内アドレスを特定できることが分かる．

次に，ext2/3/4 のパーティションのレイアウトについて述べる．前述のように，パーティションの中にファイルシステムが構築される．ext2/3/4 はパーティションを 4 キロバイトごとのブロックに分割し，パーティションの先頭からブロック番号を割り当てていく．そして，32,768 ブロックごとにブロックグループを作成する．各ブロックグループは図 -5 のような構成をしており，各ブロックグループにスーパーブロック（これは先頭グループのスーパーブロックの内容が別のグループにバックアップコピーされることになる），グループディスクリプタ，データブロックビットマップ，inode ビットマップ, inode テーブルが置かれ, それ以外のブロックがデータブロックとしてファイルのデータ（インデックスも含む）の格納に用いられる. データブロックビットマップは，そのブロックグループ内の各ブロックが使用中であるか否かを示すビット情報のあつまりで，32,768 個のブロックの使用情報はちょうど 4 キロバイト（32,768 ビット）となり 1 個のブロックに格納されることとなる．

ext4 は，ext3 に対してエクステントのサポートなどの拡張を行ったファイルシステムである．ext4 は ext2/3 に対して後方互換性を有しており，ext2/3 などでフォーマットされたパーティションを ext4 としてマウントすることができる．拡張機能であるエクステントを用いていない ext4 フォーマットのパーティションは ext2/3 ファイルシステムでマウントすることができるが，エクステントを用いている ext4 パーティションは ext2/3 により正しくマウントすることはできない．

最新版である ext4 の各種値の上限は以下のようになっている [2)]．各種上限値は，CPU モード（32 ビット，64 ビット）やブロックサイズ（1 キロバイトから 64 キロバイト）に依存するが，64 ビットモードでブロックサイズ 4 キロバイトの例における上限値は，ブロック数は 2^{64} 個，inode 数は 2^{32} 個，ファイルシステムサイズは 64 ゼタバイト，エクステント使用時のファイルサイズは 16 テラバイト，ブロックマップ時（非エクステント時）のファイルサイズは 4 テラバイトとなっている．ブロックサイズはファイルシステムフォーマット時（mkfs 時）に設定することが可能であり，これを 64 キロバイトにすればファイルシステムサイズの上限は 1 ヨタバイト（1 ヨタは 10^{24}, 情報分野では 2^{80}), ファイルサイズの上限（エクステント時，非エクステント時ともに）は 256 テラバイトとすることができる．

ext4 には，以下に挙げるような関連するファイル群を（アドレス的に）近隣に配置しようとする試みが実装されている [2)]．第 1 に，ファイルが最初に作成されたときブロックアロケータは投機的に 8 キロバイトのブロックを確保しようとする．これは近いうちに書き込みされるだろうとの仮定に基づいている．第 2 に，遅延確保を行う．ファイルが多くの書き込みを要求する場合，ファイルシステムはファイルのストレージにおける格納位置の決定を遅

延する．第3に，（これはext3/4ともにであるが）ファイルのデータを同じブロックグループの中に保存しようとし，第4に同じディレクトリのinodeは同じブロックグループに保存しようとする．第5に,（これはローカリティを高めないが）ルートディレクトリの中にディレクトリを作る場合は負荷の少ないブロックグループを検索してそこに新しいディレクトリを作成する．関連するファイルを近隣に配置する意義は，HDDの場合はヘッドの移動の削減による性能の向上であり，可動部を持たないSSDの場合であってもローカリティの向上（近隣への配置）は各命令サイズの拡大によるI/O命令数の削減や書き込み処理を単一の削除ブロックの中に集中させるなどの意義を有している．

ext4はエクステントをサポートしている[3]．図-6のようなext2/3/4の伝統的なブロックマップ手法は，巨大なファイルの処理をする（すなわち連続して大量のブロックをディスク上に確保して処理をする）のには必ずしも適切ではない．すべてのブロックのブロックアドレスを個別に記録する必要があり，たとえば1,000ブロックのファイルを作成するには1,000ブロック分のブロックアドレスをinode内か間接参照インデックス内に書き込まなくてはならない．エクステント使用時は，`ext4_extent`の`ee_len`に1000と書き込むのみでよい．

ext3/4はジャーナリング機能を有している．典型的にはinode番号8のファイルをジャーナルに使うが，external journalを使えばジャーナルデータを他のパーティションにも置ける．

ジャーナルの最初のブロックは，ジャーナルのスーパーブロックである.ほかのブロックは,ジャーナルエントリーに使用される．各ジャーナルエントリーには,「どのブロックの更新が（これから）起きるか」が記録され，更新が行われた後は「更新が完了した」ことが記録される．ジャーナルはリングバッファになっている．

更新はトランザクションにて行われ，各トランザクションは通し番号を持っている．ジャーナル内のブロックはトランザクション管理データかファイル

0 Journal Super Block
1 Descriptor (Sequence: 100)
2 File system Metadata Block
3 File system Metadata Block
4 File system Metadata Block
5 Commit (Sequence: 100)
6 Descriptor (Sequence: 101)
7 File system Metadata Block
8 File system Metadata Block
9 Commit (Sequence: 101)

図-7　ジャーナルブロック群の例

システム更新のデータが書き込まれる．

図-7の例のように，各トランザクションはトランザクション通し番号やどのブロックが更新されるか等を含むDescriptorブロックで始まり，更新されたブロック（Descriptorブロックで記述されているブロック）のデータが続く．更新がディスクに書き込まれたら，Commitブロックが書き込まれる．そして，そのCommitブロックの後には次のトランザクションのDescriptorブロックが続く．

ジャーナルには2個のモードがあり,「メタデータ(Metadata)の更新のみをジャーナルに記録する」モードか「すべての更新（データブロックの更新を含む）をジャーナルに記録する」モードかを選べる．初期設定はメタデータのみが対象である．ext3のジャーナルはブロックレベルで行われる．つまり，あるブロック内のデータ（inodeなど）がわずかでも更新されたら，そのブロック全体がジャーナルに記録される．

■ OverlayFS

OverlayFS[4]は，ユニオンマウントファイルシステムの1つである．2014年にLinux 3.14よりLinuxカーネルのメインラインに統合されており，近年はコンテナ型仮想化システムDockerのストレージドライバとして採用され注目を集めている．ユニオンマウントファイルシステムは，複数のディレクトリを重ねて1つのディレクトリの見た目を提供するファイルシステムである．通常，変更（書き

込み）は最上位のディレクトリに対してのみ行われ，最上位以外のディレクトリは読み込み専用のディレクトリであっても問題ない．ユニオンマウントファイルシステムの有名な使用方法としては，CD-ROMのイメージファイルを書き込み可能な状態でマウントするということがある．読み込み専用のCD-ROMのディレクトリの上に，書き込み可能なHDDのディレクトリを重ねると，それはCD-ROMイメージファイルの中にあるファイルを読み込み可能でありかつ，書き込みも可能であるディレクトリとなる．OverlayFSは，上位と下位の2個のディレクトリを重ねて1つのディレクトリをユーザに提供するファイルシステムである．他のユニオンマウントファイルシステムと同様に，下位ディレクトリに対しては書き込みは行われない．

図-8は，OverlayFSの動作の概要を示している．上位ディレクトリと下位ディレクトリを重ねて1つの結合ディレクトリを作成している．図の(a)のように，上位ディレクトリまたは下位ディレクトリにある名前のファイルがあれば，結合ディレクトリにその名前のファイルが存在して見える．a.txtのように上下の両方にある場合は，上位ディレクトリのa.txtが結合ディレクトリの中から読み込みできる．

結合ディレクトリ内のファイルに書き込みを行うと，図-8(b)のような動作をする．上位ディレクトリにファイルがあるa.txtやb.txtの場合，書き込み可能である上位ディレクトリの中にあるファイルに上書きされる．d.txtのように上下ディレクトリともにその名前のファイルを保持していない場合は，上位ディレクトリにその名前のファイルが作成される．c.txtのように，下位ディレクトリにのみその名前のファイルが存在する場合は，copy_upと呼ばれる処理が行われる．まず，下位ディレクトリにあるc.txtが上位ディレクトリにコピーされ（copy_up），そのコピーされたファイルに対して書き込みが行われる．結果として，下位ディレクトリに対して書き込みを行わず，下位ディレクトリのみにあるファイルへの書き込みが達成される．

図-8(c)）は，ファイル削除時の動作を示している．

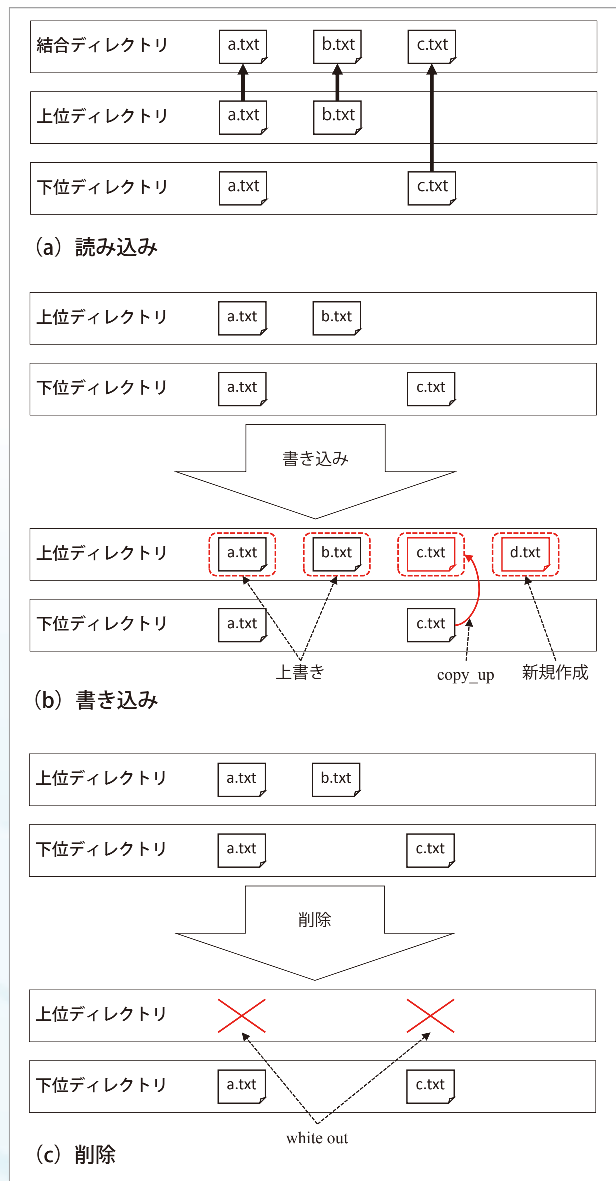

図-8　OverlayFS

下位ディレクトリにファイルが存在しないb.txtの場合の動作は容易に理解でき，上位ディレクトリの中にあるb.txtが削除され同ファイルが結合ディレクトリに存在しない状態になればよい．

下位ディレクトリにファイルが存在するa.txtとc.txtの例は少し複雑となる．これらのファイルが結合ディレクトリに存在しない状態にする必要があるが，下位ディレクトリに変更を行ってはならならない．この場合，上位ディレクトリにwhite outと呼ばれるものを作成して，下位ディレクトリのファイルにアクセス不能とする．white outは，メジャーナンバー／マイナーナンバーが0/0のキャラクタデ

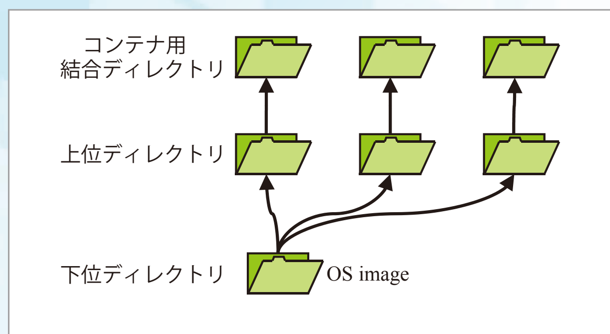

図 -9　下位ディレクトリの共有

バイスとして作成され，上位ディレクトリに white out が発見された場合は下位ディレクトリにある同名のファイルは無視される．white out 自体は隠し属性となっている[5]．

下位ファイルシステムは読み込み専用であるため，1 つのディレクトリを下位ディレクトリとして複数の結合ディレクトリを作成することができる．Docker では，**図 -9** のようにこの性質を効果的に活用している．すなわち，OS のシステムファイルが格納されている "OS image" のディレクトリを 1 つ作成し，この上に各コンテナ用の上位ディレクトリを重ね各コンテナ用の結合ディレクトリを作る．これにより，単一の OS 用システムファイルのディレクトリイメージから複数のコンテナ用ディレクトリを作成することが可能となる．この場合，各コンテナが OS のシステムファイルを複製してストレージ上の別のブロックに保持するのではないため，消費されるストレージブロックとページキャッシュが減少し，キャッシュがヒットしやすく性能が向上することなどが期待される．

■ XFS

XFS は，SGI 社が自社の OS である IRIX のために開発したファイルシステムであり，同 OS が主に画像処理に用いられことから伝統的に大きなサイズのファイルのアクセス性能が高いといわれている．ジャーナリング，エクステント，遅延確保を古くよりサポートしており，近年は多くの Linux ディストリビューションで初期設定で選択されているファイルシステムとなっている．

今後の展望

ファイルシステムは歴史のある技術であるが今でも発展している領域である．使用状況を問わず高い性能で動作するファイルシステムはなく，Linux などの OS 用には非常に多くのファイルシステムが開発されている．近年では，SSD を考慮したファイルシステムで完成度が高いものや，大規模データ処理に適したものなどの開発が期待されている．

参考文献

1）USB2.0 ストレージ HDD 2TB の fsck 所要時間，https://www.nexia.jp/server/1687/
2）Ext4 Disk Layout，https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout
3）Ext4 Design，https://ext4.wiki.kernel.org/index.php/Ext4_Design
4）Overlay Filesystem, https://github.com/torvalds/linux/commit/e9be9d5e76e34872f0c37d72e25bc27fe9e2c54c
5）Neil, B.: Overlay Filesystem,Whiteouts and Opaque Directories, https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt

（2017 年 9 月 6 日受付）

山口実靖（正会員）　sane@cc.kogakuin.ac.jp

2002 年東京大学大学院工学系研究科電子情報工学専攻博士課程修了．博士（工学）．同年より同大生産技術研究所学術研究支援員，産学官連携研究員，日本学術振興会特別研究員．2006 年工学院大学工学部講師．2007 年同大同学部准教授．オペレーティングシステム，I/O 高速化，通信プロトコルの研究に従事．電子情報通信学会，日本データベース学会各会員．