

ext2 ファイルシステム

数多くのファイルシステムをサポートしているLinuxだが、通常一般的に使用されているのは、ネイティブファイルシステムであるext2 ファイルシステム (second extended filesystem) だ。ここではこのext2 ファイルシステムの特徴をソースファイルを参照しながら紹介しよう。

Linuxのネイティブファイルシステム Ext2を知る

文：小松克行
Text: Ktsuyuki Komatsu

Linuxのファイルシステム

Linuxのファイルシステムは、図1のように拡張されてきた。

Kernel 2.0まではこれらすべてのファイルシステムがサポートされていたが、Kernel 2.2以降ではxiaとextのサポートは打ち切られている (カーネルソースに含まれていない)。これらのファイルシステムの特徴は表1のようになる。

このほか、LinuxはFAT (MS-DOS)、FAT32 (Windows 98) やNTFS (Windows NT) およびHFS (Macintosh) といった他のOSのファイルシステムや、NFS、SMBFSといったネットワークファイルシステム、ISO9660 といったCD-ROMのファイルシステムをサポート

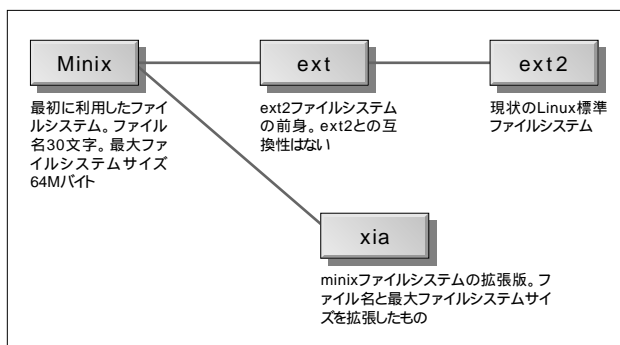


図1 Linuxのファイルシステム

トしている。

Linuxがサポートするファイルシステムのソースは、< Kernelソースディレクトリ > /fsディレクトリに集められている。このディレクトリ直下のファイルは、ファイルシステムに依存しない部分のソースである。各ファイルシステム固有のソースファイルは、< Kernelソースディレクトリ > /fs/<ファイルシステム名>ディレクトリに個別に存在する。ext2ファイルシステムの場合は、< Kernelソースディレクトリ > /fs/ext2/となる。

VFS (Virtual File System)

Linuxは複数の異なるファイルシステムをサポートしており、それぞれのファイルシステム上のファイルへのアクセスは、共通のオープン、読み書きなどのシステムコールを使用して行えるようになっている。この統一的なアクセスを実現するためにLinuxは、VFS (仮想ファイルシステム) という中間レイヤー (層) をそれぞれのファイルシステムの上に用意し、ファイルシステム処理の抽象化を実

	Minix	ext	Xia	ext2
最大ファイルシステムサイズ	64Mバイト	2Gバイト	2Gバイト	4Tバイト
最大ファイルサイズ	64Mバイト	2Gバイト	64Mバイト	2Gバイト
最大ファイル名サイズ	16/30	255	248	255
3つのファイル時刻をサポート	x	x		
拡張可	x	x	x	
可変ブロック長	x	x	x	

表1 Linuxのファイルシステム

現している (図2)。

VFSのファイルシステムの抽象化は、linux/fs.hの*_operations構造体を見ると、ファイルシステムの操作をファイル、i-node、スーパーブロック、ディスククォータのそれぞれに対する操作としてまとめていることがわかる (リスト1~4)。

リスト1 ファイルに対する操作

```
struct file_operations {
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char *, size_t, loff_t *);

    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);

    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);

    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *);
    int (*fasync) (int, struct file *, int);
    int (*check_media_change) (kdev_t dev);
    int (*revalidate) (kdev_t dev);
    int (*lock) (struct file *, int, struct file_lock *);
};
```

リスト2 i-nodeに対する操作

```
struct inode_operations {
    struct file_operations * default_file_ops;
    int (*create) (struct inode *, struct dentry *, int);
    int (*lookup) (struct inode *, struct dentry *);
    int (*link) (struct dentry *, struct inode *, struct dentry *);
    int (*unlink) (struct inode *, struct dentry *);
    int (*symlink) (struct inode *, struct dentry *, const char *);
    int (*mkdir) (struct inode *, struct dentry *, int);
    int (*rmdir) (struct inode *, struct dentry *);
    int (*mknod) (struct inode *, struct dentry *, int, int);
    int (*rename) (struct inode *, struct dentry *, struct inode *, struct dentry *);
    int (*readlink) (struct dentry *, char *, int);
    struct dentry * (*follow_link) (struct dentry *, struct dentry *, unsigned int);
    int (*readpage) (struct file *, struct page *);
    int (*writepage) (struct file *, struct page *);
    int (*bmap) (struct inode *, int);
    void (*truncate) (struct inode *);
    int (*permission) (struct inode *, int);
    int (*smap) (struct inode *, int);
    int (*updatepage) (struct file *, struct page *, unsigned long, unsigned int, int);
    int (*revalidate) (struct dentry *);
};
```

ext2 ファイルシステムの構造

ext2ファイルシステムの構造は、スーパーブロックとi-nodeを管理単位とするUNIXの標準的なファイルシステムと考えることができる。これらを詳しく見ていくことにする。

スーパーブロック (superblock)

スーパーブロックは、ファイルシステムの基本的な属性を保持する領域である。FATやNTFSファイルシステムでは、FATブートセクタやNTFSブートセクタに相当する。

スーパーブロックの構造はext2_fs.hのext2_super_block構造体で定義されており、スーパーブロックの大きさは1024バイトである (表2)。

Kernel 2.2系で取り入れられたパフォーマンス用のフィールドは、プリアロケート用のブロック数のフィールドだ。これらは、パフォーマンスの向上やデータブロックの断片化を避けるために使用される。

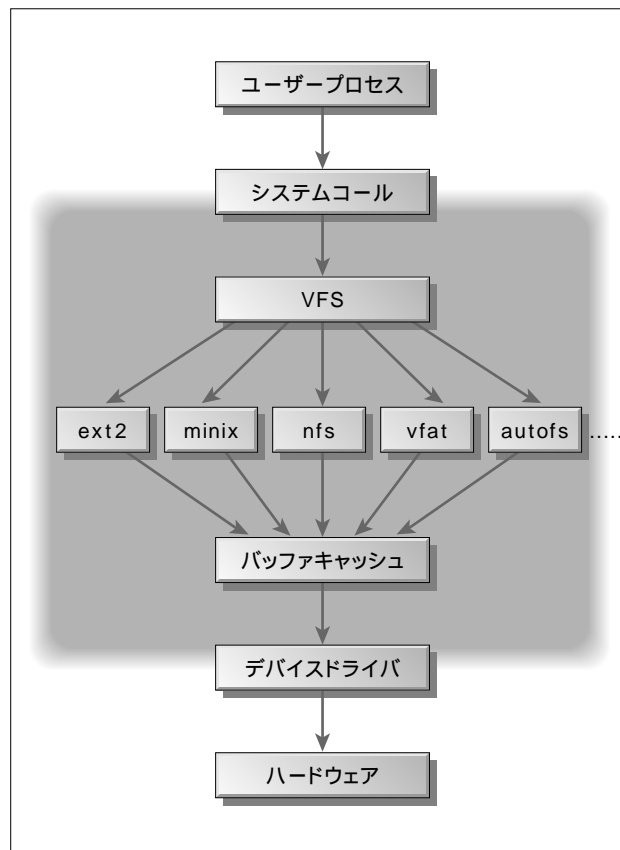


図2 VFSレイヤー

i-node

i-nodeは、ファイルの実体を表現する構造体である。ファイル名はディレクトリエントリに記述されるだけで、i-nodeでは管理していない。

i-nodeの構造はext2_fs.hのext2_inode構造体で定義される(表3)。i-nodeの大きさは128バイトとなる。

i-nodeが持つブロックのポインタ配列は15エントリだが、このうち直接参照に使用されるのは12エントリ分だけで、残りのエントリはそれぞれ、間接参照、ダブル間接参照、トリプル間接参照用に使用される(図3)。

そのため、直接参照だけで表現可能なファイルサイズは、ブロックサイズが1024、2048、4096バイトのいずれかなので、それぞれ12K、24K、48Kバイトということになる。

間接参照を利用するとそのエントリが示すブロックをi-nodeのブロックポインタ配列として使用できる。つまり、 $\text{ブロックサイズ} \div 4$ (ブロックポインタの大きさ) 個分のエントリが使用できるようになる。ブロックサイズが1024バイトの場合は256エントリ、ファイルサイズに換算すると256Kバイト拡張されることになる。

リスト3 スーパーブロックに対する操作

```
struct super_operations {
    void (*read_inode) (struct inode *);
    void (*write_inode) (struct inode *);
    void (*put_inode) (struct inode *);
    void (*delete_inode) (struct inode *);
    int (*notify_change) (struct dentry *, struct iattr *);
    void (*put_super) (struct super_block *);
    void (*write_super) (struct super_block *);
    int (*statfs) (struct super_block *, struct statfs *, int);
    int (*remount_fs) (struct super_block *, int *, char *);
    void (*clear_inode) (struct inode *);
    void (*umount_begin) (struct super_block *);
};
```

リスト4 ディスクォータ関係の操作

```
struct dqquot_operations {
    void (*initialize) (struct inode *, short);
    void (*drop) (struct inode *);
    int (*alloc_block) (const struct inode *, unsigned long,
                      uid_t, char);
    int (*alloc_inode) (const struct inode *, unsigned long,
                      uid_t);
    void (*free_block) (const struct inode *, unsigned long);
    void (*free_inode) (const struct inode *, unsigned long);
    int (*transfer) (struct inode *, struct iattr *, char, uid_t);
};
```

ダブル間接参照では、間接参照ブロックをさらに間接参照として利用する。ファイルサイズとしては、 $(\text{ブロックサイズ} \div 4) \times (\text{ブロックサイズ} \div 4) \times \text{ブロックサイズ}$ となり、ブロックサイズが1024バイトの場合は、64Mバイトを表現できる。

トリプル間接参照では、さらに間接参照が増え、3段階となる。これを利用すると $(\text{ブロックサイズ} \div 4) \times (\text{ブロックサイズ} \div 4) \times (\text{ブロックサイズ} \div 4) \times \text{ブロック}$

オフセット	サイズ	型	フィールド名	内容
0000	4バイト	__u32	s_inodes_count	i-node数
0004	4バイト	__u32	s_blocks_count	Block数
0008	4バイト	__u32	s_r_blocks_count	予約ブロック数
000c	4バイト	__u32	s_free_blocks_count	空きブロック数
0010	4バイト	__u32	s_free_inodes_count	空きi-node数
0014	4バイト	__u32	s_first_data_block	最初のデータブロック
0018	4バイト	__u32	s_log_block_size	log2(ブロックサイズ) - 10
001c	4バイト	__s32	s_log_frag_size	log2(1024/フラグメントサイズ)
0020	4バイト	__u32	s_blocks_per_group	グループあたりのブロック数
0024	4バイト	__u32	s_frags_per_group	グループあたりのフラグメント数
0028	4バイト	__u32	s_inodes_per_group	グループあたりのi-node数
002c	4バイト	__u32	s_mtime	マウント時刻
0030	4バイト	__u32	s_wtime	書き込み時刻
0034	2バイト	__u16	s_mnt_count	マウント回数
0036	2バイト	__s16	s_max_mnt_count	最大マウント回数
0038	2バイト	__u16	s_magic	署名(0xEF53)
003a	2バイト	__u16	s_state	ファイルシステム状態
003c	2バイト	__u16	s_errors	エラー時の処理
003e	2バイト	__u16	s_minor_rev_level	マイナーリビジョンレベル
0040	4バイト	__u32	s_lastcheck	最終チェック時刻
0044	4バイト	__u32	s_checkinterval	最大チェック間隔
0048	4バイト	__u32	s_creator_os	OS
004c	4バイト	__u32	s_rev_level	リビジョンレベル
0050	2バイト	__u16	s_def_resuid	予約ブロックのデフォルトユーザーID
0052	2バイト	__u16	s_def_resgid	予約ブロックのデフォルトグループID
0054	4バイト	__u32	s_first_ino	最初の非予約i-node
0058	2バイト	__u16	s_inode_size	i-nodeの大きさ
005a	2バイト	__u16	s_block_group_nr	ブロックグループ番号
005c	4バイト	__u32	s_feature_compat	互換フラグ
0060	4バイト	__u32	s_feature_incompat	非互換フラグ
0064	4バイト	__u32	s_feature_ro_compat	リードオンリー互換フラグ
0068	16バイト	__u8	s_uuid	ボリュームの128ビットのUUID
0078	16バイト	char	s_volume_name	ボリューム名
0088	64バイト	char	s_last_mounted	最後にマウントしたディレクトリ名
00c8	4バイト	__u32	s_algorithm_usage_bitmap	圧縮ボリュームのアルゴリズム
00cc	1バイト	__u8	s_prealloc_blocks	プリアロケートブロック数 ^{*2}
00cd	1バイト	__u8	s_prealloc_dir_blocks	ディレクトリのプリアロケートブロック数 ^{*3}
00ce	2バイト	__u16	s_padding1	パディング
00d0	816バイト	__u32	s_reserved	予約領域

表2 ext2_super_block構造体(サイズ1024バイト)

サイズで、ブロックサイズが1024バイトの場合は、16G バイトを表現可能だ。しかし現状、i-nodeのサイズフィールドの制限からext2ではファイルサイズは4Gバイト、VFSのレイヤーで2Gバイトに制限されてしまっている。

またext2のi-nodeでは、ブロックを表現するのに範囲指定を行うことはできない。つまり100から200の消費ブロックを表現するのに、100~200とはできず、100、101、102、...、199、200とそのすべてのブロック番号をポインタ配列に記述する必要がある。

このほかext2のi-nodeでは、フラグを設定することができる(リスト5)。これには、chattr(フラグを設定する)とlsattr(フラグの状態を表示する)を使用する。フラグでは、セキュアデリート(使用データブロックを0x00などで上書きしてから開放する)、アクセスタイムを記録しない、アペンドのみ許可、変更を許可しない、同期モードでの書き込み、アンデリートなどをファイル単位で設定できるようになっている(詳細はmanを参照)。しかし、残念ながらすべての機能が現時点で実装されているわけではないようだ。

ビットマップ(bitmap)

ビットマップは、ディスク上の領域の割り当て状態について、割り当てられているか割り当てられていないかを保持する1ビットのフラグの配列である。ext2ファイルシステムには、ブロック用とi-node用の2種類のビットマップがあり、ブロックとi-nodeを別々に管理している。

オフセット	サイズ	型	フィールド名	内容
0000	2バイト	__u16	i_mode	ファイルモード
0002	2バイト	__u16	i_uid	ユーザーID
0004	4バイト	__u32	i_size	サイズ(バイト)
0008	4バイト	__u32	i_atime	アクセス時刻
000c	4バイト	__u32	i_ctime	作成時刻
0010	4バイト	__u32	i_mtime	更新時刻
0014	4バイト	__u32	i_dtime	削除時刻
0018	2バイト	__u16	i_gid	グループID
001a	2バイト	__u16	i_links_count	リンク数
001c	4バイト	__u32	i_blocks	ブロック数
0020	4バイト	__u32	i_flags	ファイルフラグ
0024	4バイト	union	osd1	OS依存部1
0028	60バイト	__u32	i_block	ブロックのポインタ配列(15エン트리)
0064	4バイト	__u32	i_version	ファイルバージョン(NFS用)
0068	4バイト	__u32	i_file_acl	ファイルACL
006c	4バイト	__u32	i_dir_acl	ディレクトリACL
0070	4バイト	__u32	i_faddr	フラグメントアドレス
0074	12バイト	union	osd2	OS依存部2

表3 ext2_inode構造体

ブロックグループ(group)

ext2ファイルシステムではパーティション上をブロックグループと呼ばれる領域に分割して管理する(図4)。ブロックグループは、BSDのFFS(Fast File System)のシリンダグループに相当する概念で、アクセスの局所化によるパフォーマンスアップを目的としている。

ext2ファイルシステムでは、単純にブロック数をブロックグループの単位としており、ひとつのブロックグループは8192ブロックとなっている。ブロックグループは、以下の領域で構成される。

スーパーブロックのコピー

ブロックグループディスクリプタ(記述子)

ブロックビットマップ

i-nodeビットマップ

i-nodeテーブル

データブロック

なおブロックグループディスクリプタには、ブロックビットマップ、i-nodeビットマップ、i-nodeテーブルのブロックポインタとブロックグループ内のブロックとi-nodeの空き数、そしてディレクトリ数が記述される。サイズは32

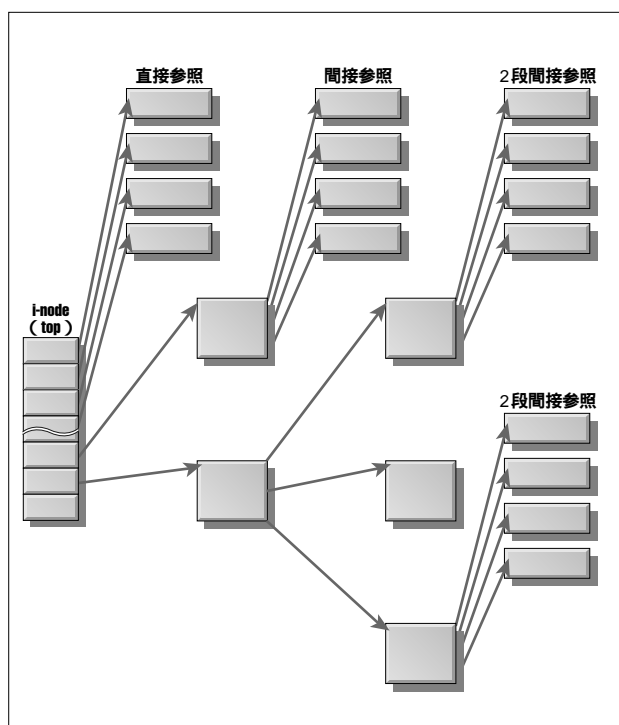


図3 データブロックのアロケート

バイト。

各ブロックグループの状態は、このディスクリプタを参照するだけで把握できるようになっている。

ブロックとフラグメント

ブロックはファイルシステムの割り当ての基本単位である。ほかのファイルシステムでは、クラスタ、アロケーションユニットなどと呼ばれている。ext2 ファイルシステムのブロックサイズは、1024、2048、4096バイトのいずれかであり、ext2_fs.h の EXT2_MIN_BLOCK_SIZE および EXT2_MAX_BLOCK_SIZE でその上限と下限が定義されている。

ext2 ファイルシステムのフラグメントの大きさも 1024、2048、4096 バイトのいずれかであり、ext2_fs.h の EXT2_MIN_FRAG_SIZE および EXT2_MAX_FRAG_SIZE でその上限と下限が定義されている。このフラグメントは、i-node の構造体にフラグメントアドレスがあることから分かるように、ひとつのブロックに複数の小さなファイルを格納するために使用されるもので、フラグメントサイズはそのしきい値を表している。しかし、現状の ext2 ファイルシステムでは、パフォーマンスを優先したためか、このフラグメントは機能していない。

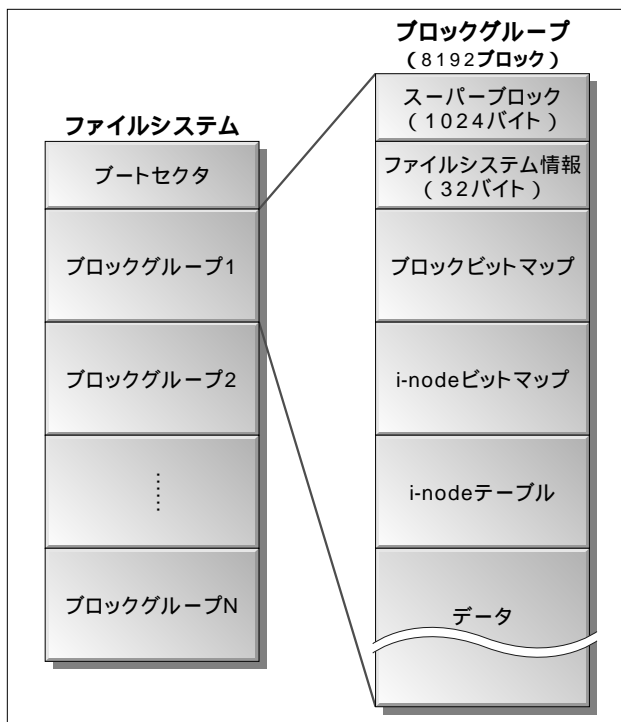


図4 ブロックグループ

ディレクトリ

ディレクトリは、ファイル名とファイルの実体である i-node とを結び付けている。ディレクトリは、ディレクトリ属性を持ったファイルであり、複数のエントリを持つ。各エントリは、表4の構造を持つ。

ディレクトリエントリの大きさはファイル名の長さによって変わり、全体が4の倍数となるように調整される。つまり、ディレクトリエントリの大きさは、ファイル名が1バイト~4バイトなら12バイト、5バイト~8バイトなら16バイトと、4バイトごとに増加する。

たとえばファイル名が“bin”のように3バイトなら、ディレクトリエントリの大きさは12バイトとなるが、ファイル名が“sbin”のように4バイトでもディレクトリエントリの大きさは変わらず12バイトである。また、ファイル名がヌルターミネートされているとは限らないので、必ずファイル名の長さ (name_len) を使って参照する必要がある。

ディレクトリもファイルと同様に割り当てられるため、ブロック単位の割り当てとなる。なおディレクトリエントリはグループの境界をまたがらないように配置される。

リスト5 i-nodeフラグ

```
/*
 * Inode flags
 */
#define EXT2_SECRM_FL      0x00000001 /* Secure deletion */
#define EXT2_UNRM_FL      0x00000002 /* Undelete */
#define EXT2_COMPR_FL     0x00000004 /* Compress file */
#define EXT2_SYNC_FL      0x00000008 /* Synchronous updates */
#define EXT2_IMMUTABLE_FL 0x00000010 /* Immutable file */
#define EXT2_APPEND_FL    0x00000020 /* writes to file may only
                                append */
#define EXT2_NODUMP_FL    0x00000040 /* do not dump file */
#define EXT2_NOATIME_FL   0x00000080 /* do not update atime */
                                /* Reserved for compression usage... */
#define EXT2_DIRTY_FL     0x00000100
#define EXT2_COMPRBLK_FL  0x00000200 /* One or more compressed
                                clusters */
#define EXT2_NOCOMP_FL    0x00000400 /* Don't compress */
#define EXT2_ECOMPR_FL    0x00000800 /* Compression error */
                                /* End compression flags --- maybe not all used */
#define EXT2_BTREE_FL     0x00001000 /* btree format dir */
#define EXT2_RESERVED_FL 0x80000000 /* reserved for ext2 lib */
#define EXT2_FL_USER_VISIBLE 0x00001FFF /* User visible flags */
#define EXT2_FL_USER_MODIFIABLE 0x00000FFF /* User modifiable flags */
```

ディレクトリエントリとi-node番号

ext2ファイルシステムでは、ファイルを識別するためにi-nodeが使われる。ディレクトリエントリは、i-nodeを指定して名前を付けているに過ぎないので、同一のi-nodeに別のファイル名を付けることも可能だ。

ハードリンクはlnコマンドを使って作成するが、これはディレクトリエントリを新しく作る際に、元のファイルと同じファイルのi-nodeを指定しているにすぎない。このとき、i-nodeのリンク数(i_links_count)がいくつかのディレクトリエントリから参照されているかを示している。

またディレクトリは、明示的に操作を指定せずに作られるハードリンクの例だ。ディレクトリエントリの“.”は自分自身、“..”は親ディレクトリへのハードリンクとなる。つまり、“.”のディレクトリエントリには、そのディレクトリ名のディレクトリエントリと同じi-node番号、“..”は親ディレクトリのi-node番号が書き込まれる。

ディレクトリエントリを作成するとリンク数が2となるのは、“.”もそのディレクトリを指しているからである。また、ディレクトリの中にディレクトリを作るとリンク数がひとつずつ増えるのは“..”が親ディレクトリを指していることによる。

なおハードリンクは、i-node番号が基準となるので、同一ファイルシステム(同一パーティション)内でしか作成することはできない。

ディレクトリエントリの操作とi-nodeの操作

ディレクトリエントリに書き込まれている情報とi-nodeに書き込まれている情報を見比べれば、lsなどのコマンド群による操作がファイルシステムに対するどのような操作に翻訳されるかを予想することができる。ここでいくつか紹介しておこう。

・mvコマンド

同一ファイルシステム内でのファイルやディレクトリの

オフセット	サイズ	型	フィールド名	内容
0000	4バイト	__u32	inode	i-node番号
0004	2バイト	__u16	rec_len	ディレクトリエントリの大きさ(4の倍数)
0006	1バイト	__u8	name_len	ファイル名の長さ(1~255)
0007	1バイト	__u8	file_type	ファイルタイプ
0008	可変長	char	name	ファイル名(最大255バイト)

表4 ext2_dir_entry_2構造体

移動は、ディレクトリエントリをあるディレクトリから別のディレクトリに移動するという操作に翻訳することができる。このときi-nodeに関する操作は必要ない。もちろん移動先のディレクトリでは、ディレクトリエントリの増加によりディレクトリファイル自体を拡張する必要がある場合もあるので、実際の動作の詳細はもっと複雑になる。

・chmodコマンド

ファイルの属性の変更は、i-nodeのファイルモード(i_mode)を変更するという操作になる。

・touchコマンド

すでに存在するファイルの日時の変更は、i-nodeのアクセス時刻(i_atime)や更新時刻(i_mtime)の変更となる。

・lsコマンド

lsコマンドでファイル名だけを表示する場合には、ディレクトリエントリだけを読む操作となる。しかし、ファイル名以外の情報を取得しようとする、i-nodeを読み出して各種の属性を取得する必要がある。このため、名前だけを表示する場合に比べれば、属性の表示は負荷が大きい。

ファイルの情報

ひとつのファイルを構成するには、ext2ファイルシステムではどのぐらいの情報が必要となるのだろうか？ 今までの説明から分かるようにひとつのファイルを構成する情報は、さまざまな個所に分散して記録されることになる。単純にファイルを作成しただけでも以下のような情報が必要となる。

ファイルの名前を持つディレクトリエントリ

ファイルのi-node

ファイルのデータが格納されたブロック

ファイルのi-nodeの割り当てを記録するi-nodeのビットマップ

ファイルのブロックの割り当てを記録するblockのビットマップ

プリアロケーション

ext2ファイルシステムでは、フラグメントを押さえ、パフォーマンスの低下を押さえるためにプリアロケーションという技法を用いている。

プリアロケーションは、ファイル、ディレクトリ、i-nodeそれぞれの操作で有効で、デフォルトのブロックのプリアロケーションブロック数は8となっている。これは、linux/ext2_fs.hヘッダファイルのEXT2_DEFAULT_

PREALLOC_BLOCKSで定義されている。この値はスーパーブロックに書き込まれており、必要に応じて変更することができる。

```
#define EXT2_PREALLOCATE
#define EXT2_DEFAULT_PREALLOC_BLOCKS 8
```

アロケートの実際

ではext2ファイルシステムでどのようにディレクトリ、ファイルが作成されるのか、ごくごく簡単に説明しておこう。実際には、いくつかのアルゴリズムの組み合わせで配置先が決定されるのだが、ここではごく単純な例を紹介する(リスト6)。

まずmke2fsでファイルシステムを作成すると/(ルート)ディレクトリが作成される(これはあくまであるファイルシステム内のトップディレクトリ)。

/(ルート)ディレクトリは、ブロックグループ1の最初のi-nodeエントリを消費する。ここで新たにディレクトリを作成すると、同一ブロックグループではなく、異なるブロックグループ、ここではブロックグループ2に作成される。結論をいうと、ディレクトリの場合、親ディレクトリとはなるべく異なる(ファイルシステムのレイアウトバランス)、しかし近くの(アクセスの局所化)ブロックグループに作成される。これには、先ほど説明したブロックグループディスクリプタが効力を発揮する。ブロックグループディスクリプタにディレクトリの数のエントリがあるのはこのためだ。

一方ファイルは、属するディレクトリと同じグループ内になるべく作成される。こちらもアクセスの局所化を図るためだ。

基本的なアルゴリズムは以上のようなものだが、ディスクスペースの使用率が高くなってくると、最終的にはそのファイルシステムの先頭からベタに空きスペースをサーチするような方法がとられるようになる。

ext2fs ツール

ext2ファイルシステム用のツールには、次のようなものがある。

e2fsck **ファイルシステムのチェックと修復を行う。**
e2label **ファイルシステムのボリュームラベルを設定する。**

mke2fs **ファイルシステムを作成する。**
debugfs **ファイルシステムのデバッグを行う。**
dumpe2fs **ファイルシステムの各種パラメータを表示する。**
tune2fs **ファイルシステムの設定を変更する。**

圧縮機能

ext2ファイルシステムに圧縮機能を付け加えるe2comprパッケージが以下から入手できる。

e2compr: Transparent compression for ext2 filesystem
<http://debs.fuller.edu/e2compr/>

e2comprパッケージはカーネルに対するパッチと、e2comprに対応した各種のツール群から構成される。e2comprパッケージを使うには、圧縮機能に対応したツールを使う必要がある。

Windows関連

Windowsでext2ファイルシステムを扱うためのツールがいくつか提供されている。

FSDEXT2: Second extended file system for Windows 95
<http://www.yipton.demon.co.uk/>
Windows 95からext2ファイルシステムをマウント(ただしリードオンリー)できる。

Explore2fs、the WIN32 explorer for Linux ext2fs partitions
<http://uranus.it.swin.edu.au/jn/linux/Explore2fs.htm>
Windows NT 4.0、Windows 95からext2ファイルシステムを操作できるエクスプローラ・ライクなユーティリティ。

Ext2fs Home Page

Ext2fs Home Page

<http://web.mit.edu/tytso/www/linux/ext2.html>

E2fsprogs: Ext2 Filesystem Utilities

<http://web.mit.edu/tytso/www/linux/e2fsprogs.html>

Ext2ファイルシステムは、大容量ディスクを扱うには、ブロックグループのサイズなど管理単位が小さすぎる感があるが、今後、削除されたファイルの復元やファイルのオンライン圧縮、さらにアクセスコントロールリストなどの実装が計画されている。Ext2ファイルシステムは、まだまだ開発途上なのである。