オペレーティングシステム

2022/11/8

海谷 治彦

目次

ファイルシステム

- ファイル
- ファイルの編成
- ファイルの操作
- ディレクトリ (フォルダ)
- ディレクトリの操作
- ファイルシステムの内部構造
- ファイル管理プログラム

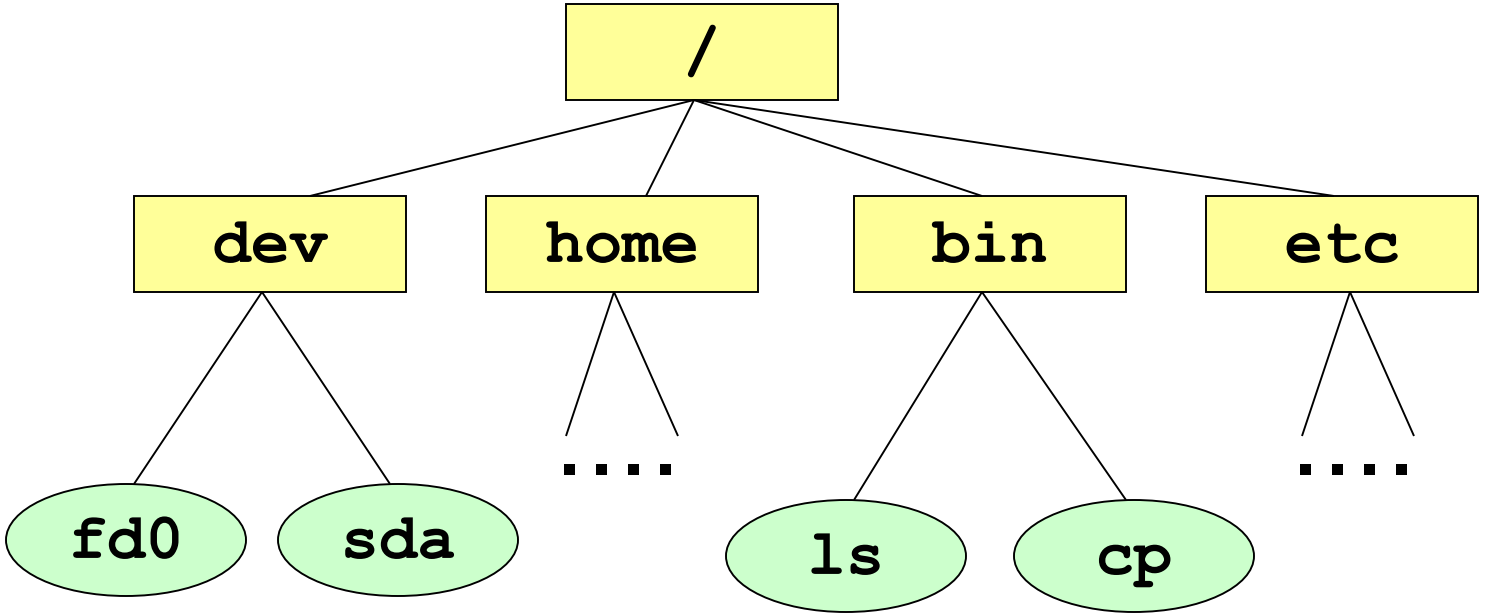
この辺は
情報科学リテラシの
復習となるので
軽くやる.

今回の焦点

- OSのファイルシステムは一般に木構造となっているが，ディスクもSSDもハード製造時点では中に木が入っているわけではない.
- このギャップをOSがどう埋めているかを理解する.
- 要は単なるデータ列のディスク等を，どのように木に変換しているかを理解する.

UNIX系ファイルシステムの概要

ご存知のとおり，UNIX系OS(その他も大抵そうだけど)は，階層的なファイルシステムを持っている．



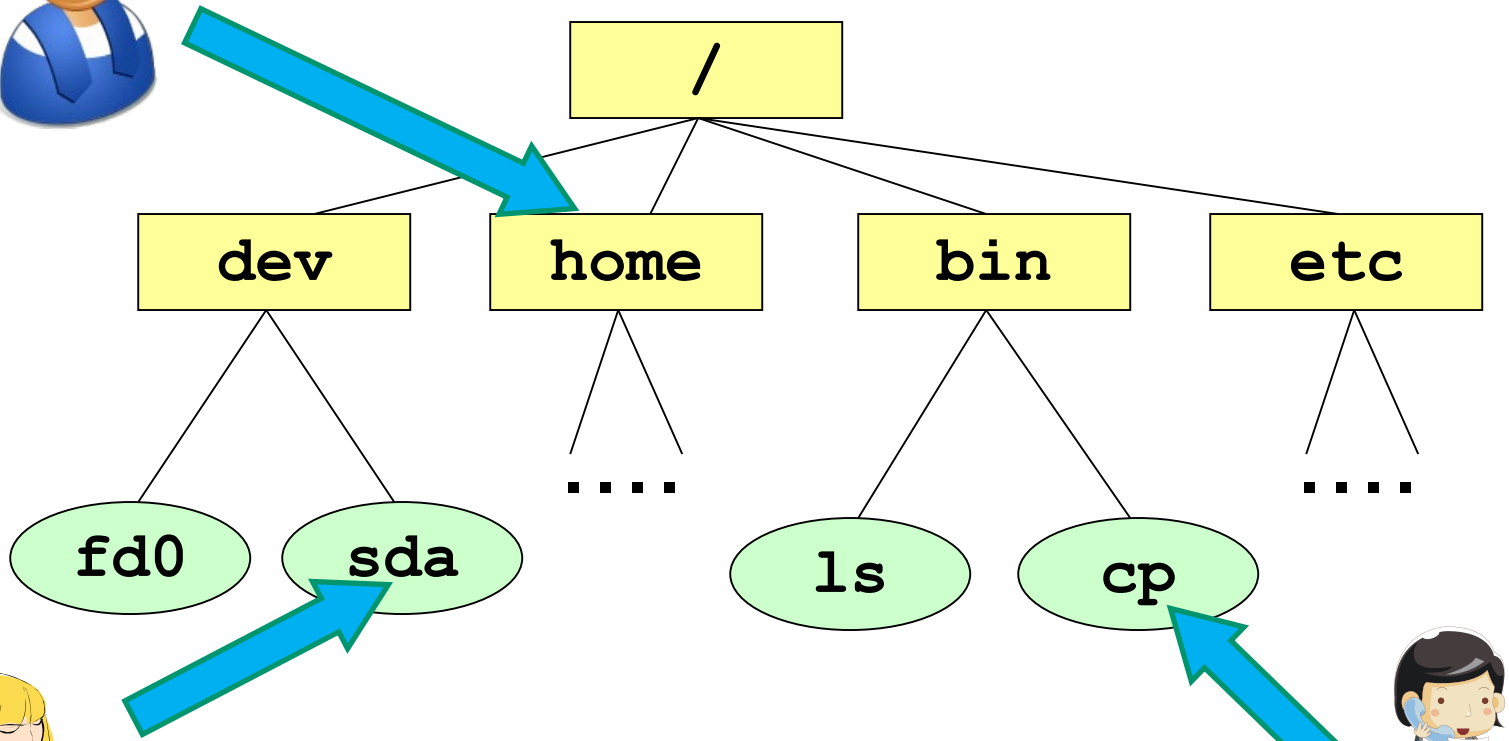
文献5 p.15

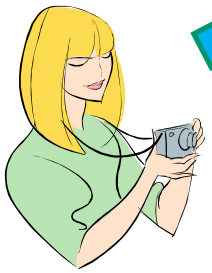
プロセスとファイル

- プロセスからはファイルシステムの一部(全体かもしれない)が参照可能となっている.
  - ファイルシステム自体は全プロセスが同じものを参照している.
    - ファイルを通してプロセス間で情報交換ができる.
- 参照可能なファイルシステムの最上位をルートディレクトリと呼ぶ.
- プロセス毎に，現在，注目しているディレクトリというのは異なる.
- このようなディレクトリを Current Working Directory と呼ぶ.

プロセスとファイルの関係イメージ

このプロセスはあるディレクトリを参照している.





このプロセスは
あるデバイスに
アクセスしている

このプロセスは
あるコマンドを
実行している.

ファイルの種類

- 通常ファイル
- ディレクトリ
- シンボリックリンク
- ブロック型デバイスファイル
- キャラクタ型デバイスファイル
- パイプ，名前付きパイプ
- ソケット

この辺は扱わない.

inode番号: ファイルの識別子

- ファイル名というのは，割と簡単につけかえられるので，実はファイルの識別子にはあまりなっていない.
- ファイル名はディレクトリからファイルを参照すための名前に過ぎない.
  - 同一ファイルを複数の名前で参照できる.
- ファイルはinodeというデータ構造で内部的には管理されており，inode番号がファイルを識別するidと考えることができる.

(ハード)リンク

- ファイルを特定ディレクトリ下に名前をつけて所属させるのがリンクである.
- 前述のように，ファイルの実体はinodeで管理されているので，
  - 同一実体のファイルを，異なるディレクトリに異なる名前で所属させることができる.

```
[kaiya@flute03 ~]$ ls -li a.c
17301562 -rw-r--r-- 1 kaiya info 119  5月 12 12:33 2014 a.c
[kaiya@flute03 ~]$ mkdir tmp
[kaiya@flute03 ~]$ ln a.c tmp/another.c
[kaiya@flute03 ~]$ ls -li tmp/another.c
17301562 -rw-r--r-- 2 kaiya info 119  5月 12 12:33 2014 tmp/another.c
[kaiya@flute03 ~]$
```

上記によって，a.c と tmp/another.c はファイルの実体は同じ.

シンボリックリンク

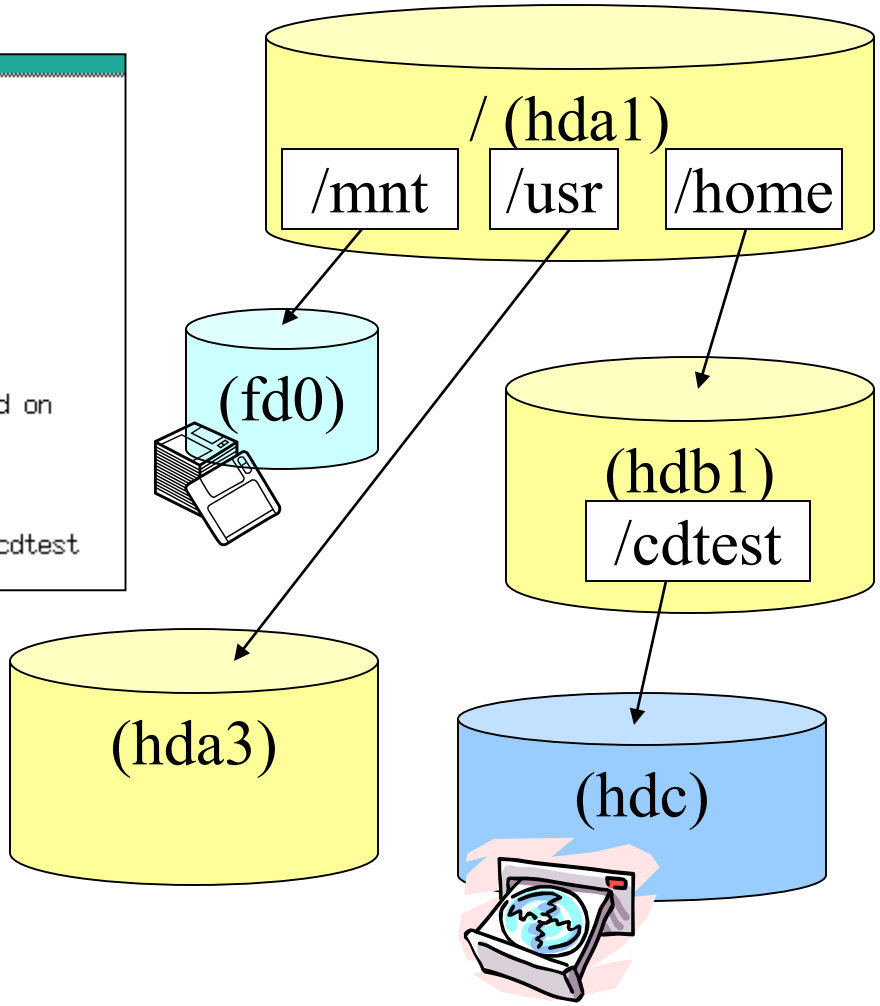
- ハードリンクには大きな制限がある.
  - 同じファイルシステムに所属するファイルにしか適用できない.
  - 一般ユーザーはディレクトリへのリンクをはれない.
- この制限を克服するためにシンボリックリンクが作られた.
  - 存在しないファイルにさえリンクが張れる.

Windows(NTFS)の場合

- ショートカット
  - ファイル/フォルダ
  - GUI上でファイルの別名をつける仕組み.
- シンボリックリンク
  - ファイル/フォルダ
  - ほぼLinuxに同じだが，管理者権限が必要.
- ハードリンク
  - ファイルのみ
  - ほぼLinux系と同じ Linuxの場合，フォルダも使えるけど
- ジャンクション
  - フォルダのみ
  - ほぼシンボリックリンクの下位互換，古い機能.

UNIX系では複数のファイルシステムで一つの木構造を作る

```
[kaiya@linux2001 ~]% !cat
cat /proc/mounts
/dev/root / ext2 rw 0 0
/proc /proc proc rw 0 0
usbdevfs /proc/bus/usb usbdevfs rw 0 0
/dev/hda3 /usr ext2 rw 0 0
none /dev/pts devpts rw 0 0
/dev/hdb1 /home ext2 rw 0 0
/dev/fd0 /mnt vfat ro 0 0
/dev/cdrom /home/cdtest iso9660 ro 0 0
[kaiya@linux2001 ~]% df
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/hda1              2016016     85192   1828412   4% /
/dev/hda3              2016044   1466612    447020  77% /usr
/dev/hdb1             76920416    928080  72084928   1% /home
/dev/fd0                  1423       764       659  54% /mnt
/dev/hdc                  1896      1896         0 100% /home/cdtest
[kaiya@linux2001 ~]% []
```



ファイルのアクセス権

- ユーザー，同一グループ，その他で分類
- モードはそれぞれ rwx
- 他3つほどのアクセス権情報を持つ.

例

```
[kaiya@linux2001 ~]% ll a.*
-rw-r--r--    1 kaiya    software       94 Jul  9  2002 a.c
-rw-r--r--    1 kaiya    software      432 Jun 29  2002 a.html
-rw-r--r--    1 kaiya    software    11226 Jun 15 02:37 a.jpg
-rwxr-xr-x    1 kaiya    software    12704 Jul 29  2002 a.out*
[kaiya@linux2001 ~]%
```

個々のファイル内のアクセス法

- 順次的(sequential)アクセス
  - Cのリスト構造やビデオテープのように前から順番にアクセスする方式.
  - UNIX系の通常ファイルは通常このアクセス形式をとる.
- ランダムアクセス
  - Cの配列のように順番に関係なく任意の位置のデータにアクセスできる方式.

通常ファイルのためのシステムコール

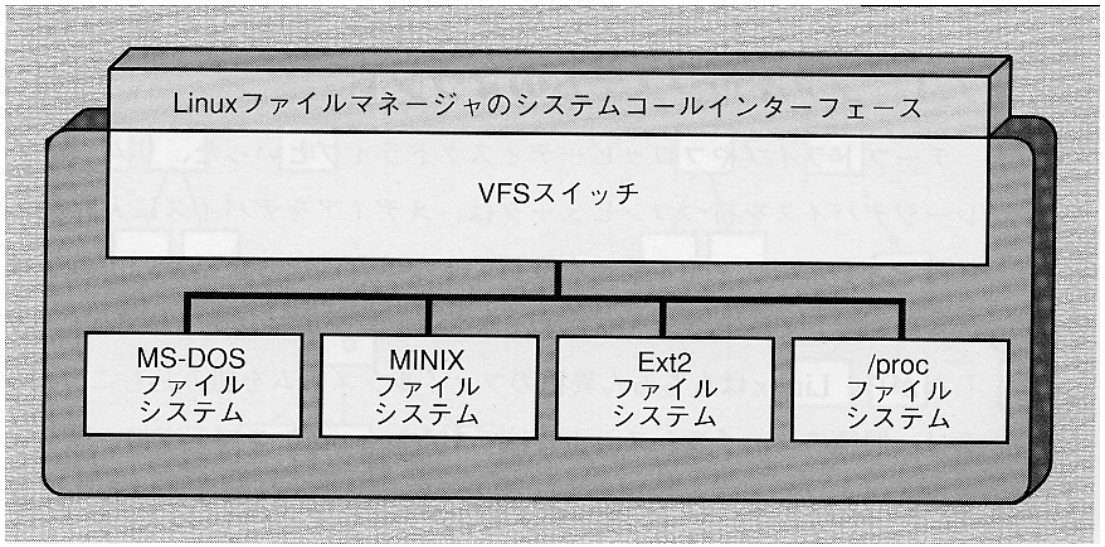
- open アクセスするためにファイルを開ける関数.
- close 逆に閉じる関数
- read 読む関数
- write 書く関数
- lseek 現在の読み書き位置を移動させる関数. (ビデオの早送り巻き戻しみたいなもの)
- rename ファイル名の付け替え
- unlink ディレクトリからのエントリ削除.

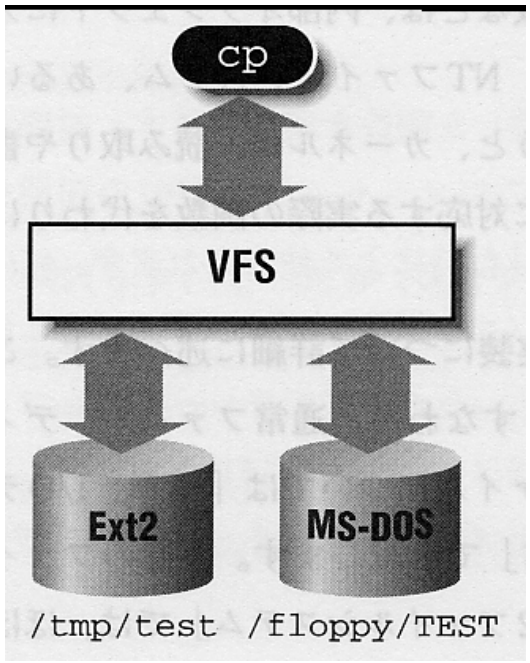
これらはシステムコールなので，実際には，カーネルへの作業要求依頼である.

仮想ファイルシステム VFS

- Linuxでは物理的・論理的に種類の異なるファイルシステムを統一的に扱うために,
- 仮想ファイルシステム(スイッチ)VFSというのを仲介役としておいて全てのファイルを扱うようにしている.
- これによって, ファイルシステムの種類や所在をユーザーは意識せずにファイルにアクセスできる.
  - システムコールの種類を変える必要が無い.

VFSの概念図


Linuxファイルマネージャのシステムコールインターフェース
VFSスイッチ
MS-DOS
ファイル
システム
MINIX
ファイル
システム
Ext2
ファイル
システム
/proc
ファイル
システム


cp
VFS
Ext2
MS-DOS
/tmp/test
/floppy/TEST

ファイルアクセスの実際

- 種類の違うファイルシステムへのアクセスには当然，異なる関数(機能)が必要である.
  - CDROMとHDとではデータの読み方は違うだろう.
- しかし，OS操作やアプリ上において，一般にはファイルシステムの実体の違いにより関数を使い分けすることは無い.
  - CDROMだろうが，HDだろうが，openで開けてreadで読む.
- ユーザーはVFSを経由してファイルアクセスするので，この点を考慮しなくてよい.
- 当然，VFSは一般的なファイルアクセス関数群(システムコール)とシステム固有のアクセス関数との対応を知っている.

参考: ファイル操作切り替えの戦略

本質的な戦略の説明.
実際はモちっと複雑.

```
ssize_t read_ext2(struct file *, char *, size_t, loff_t *){
  // ext2ファイルシステムを読む処理.
}

ssize_t read_fat32(struct file *, char *, size_t, loff_t *){
  // FAT32ファイルを読む処理.
}
```

```
main(){
ssize_t (*read)(struct file *, char *, size_t, loff_t *);

  if(ファイルがext2上にある)
    read=read_ext2;
  else if(ファイルがFAT32上にある)
    read=read_fat32;

  (*read)(&file, buf, size); // 実体に関係なくファイルを読む
}
```

補足: 関数へのポインタ

- 普通のCの構文の一種.
- 返り値と引数の型が一致している関数を, 変数(コレが関数へのポインタ)に代入して, 関数呼び出しを動的に変更することができる.
- 無論, 関数引数にも使える.

例: 動かしてみて

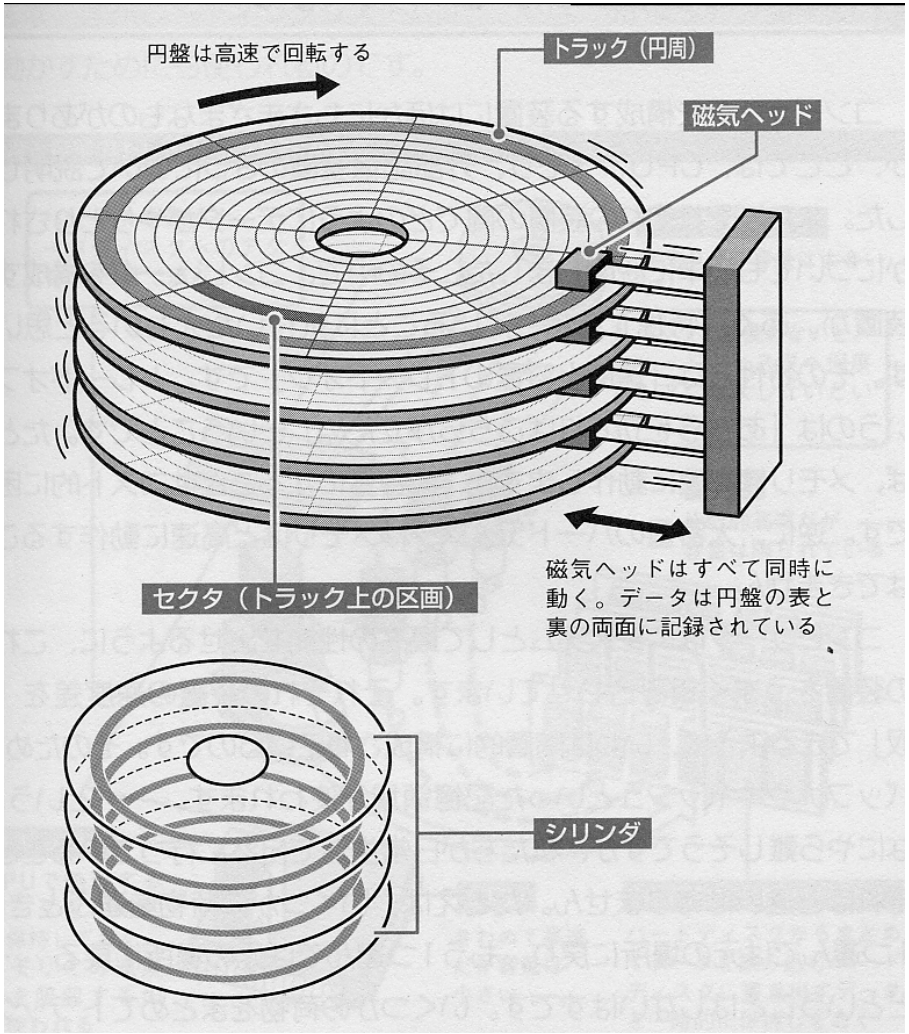
```
#include <stdio.h>

int add(int a, int b){
  return a+b;
}

int sub(int a, int b){
  return a-b;
}

main(){
int (*fp)(int, int); // ここで関数へのポインタを宣言.

  fp=add; // 関数名を代入できる
  printf("%d¥n", (*fp)(10, 4) );

  fp=sub; // 関数名を代入できる
  printf("%d¥n", (*fp)(10, 4) );
}
```

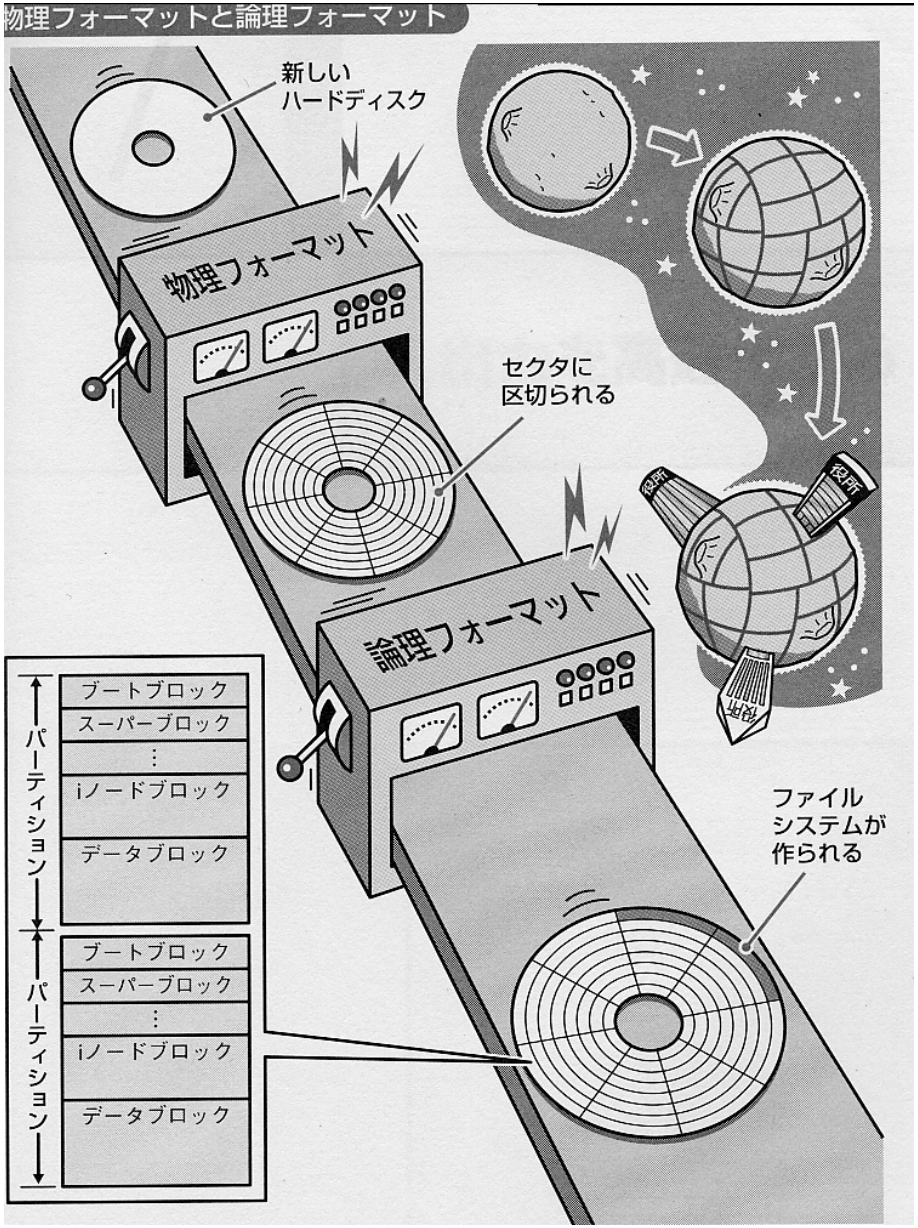
命令文は全く同じだが，fpの指している関数の実体が異なるため，計算結果も異なる.

ディスク等とファイルシステム

- 実際のHDやUSBキーは，物理的には単なるデータの配列に過ぎない．
- そのようなデータ配列を「木構造」に見せるための仕組みを解説する．
- 木構造に見せるための規則は複数種類あり，いわゆる「論理フォーマット」と呼ばれる．
- 例として，FATとext2の仕組みをそれぞれ示す．

ディスクの物理構造


円盤は高速で回転する
トラック（円周）
磁気ヘッド
セクタ（トラック上の区画）
磁気ヘッドはすべて同時に動く。データは円盤の表と裏の両面に記録されている
シリンダ



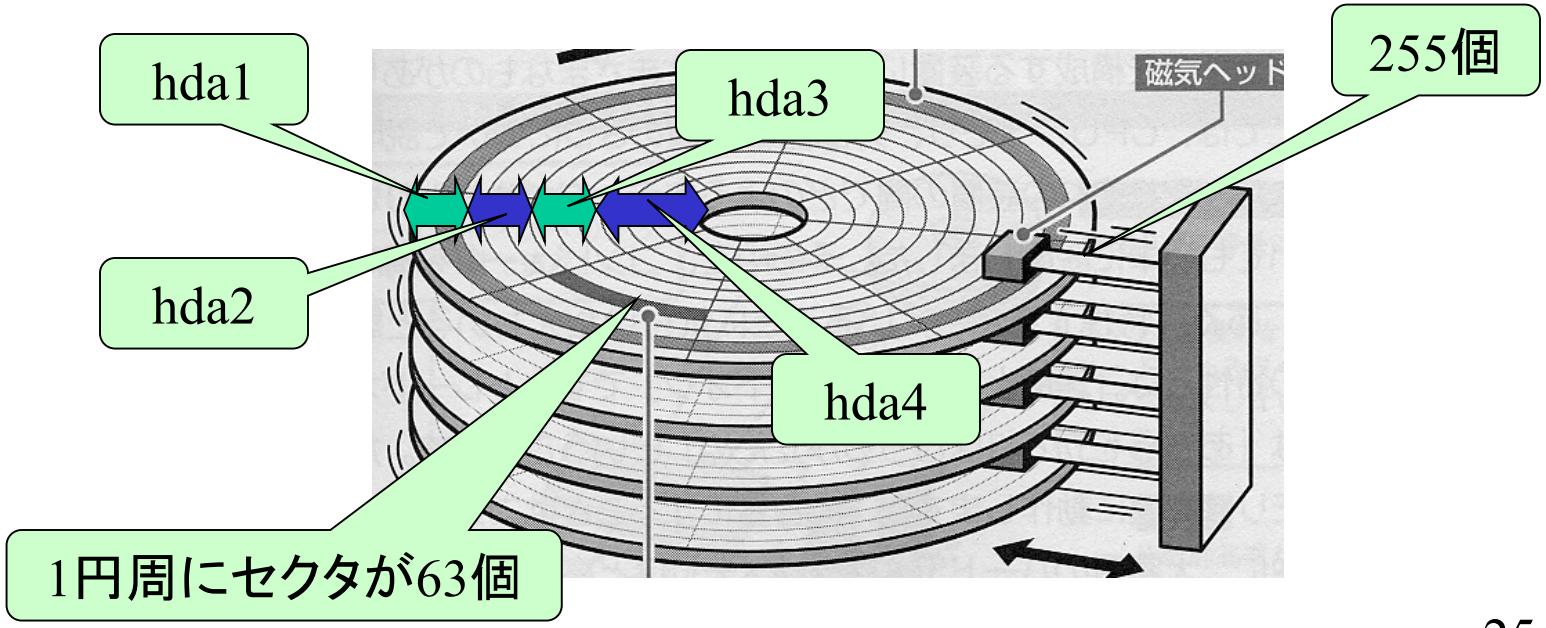
フォーマット

- 通常，隣接したシリンダ何枚かをグループ化し，それをパーティションとする.
- ディスク丸ごと1パーティションとしてもさしつかえない.

とあるディスクの情報の実例

```
ディスク /dev/hda: ヘッド 255, セクタ 63, シリンダ 2434
ユニット = シリンダ数 of 16065 * 512 バイト

  デバイス    始点  終点     ブロック     ID  システム
/dev/hda1      1   255    2048256    83  Linux
/dev/hda2    256   321     530145    82  Linux スワップ
/dev/hda3    322   576    2048287+   83  Linux
/dev/hda4    577  2434   14924385    83  Linux
```

4つのパーティション



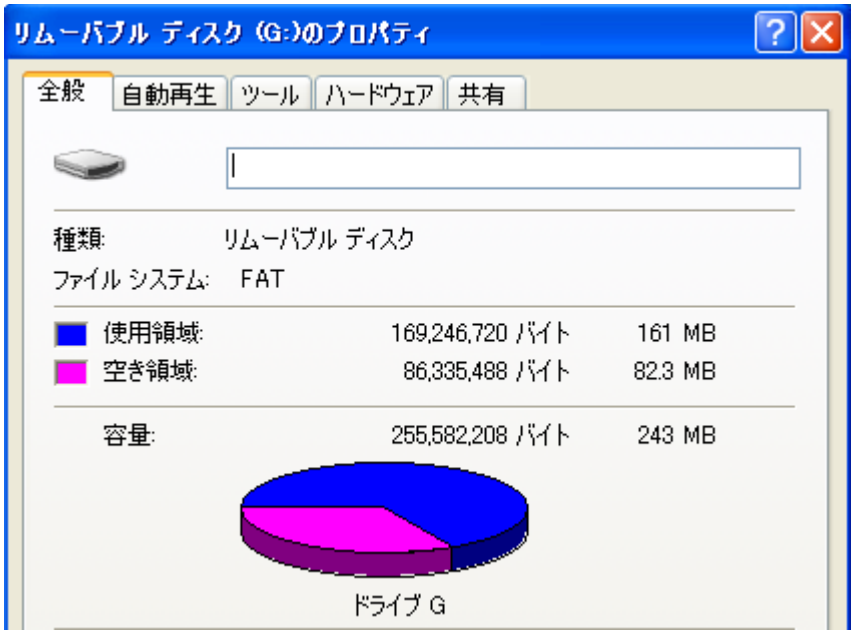
データ読書きの補足

- データは基本的にセクタ単位(512Bもしくは1024B単位)で物理的に読書きする.
- 大抵，どの論理フォーマット(ファイルシステム)でも，複数のセクタをグループ化して扱う.
  - FATの場合，クラスタ
  - Ext2の場合，ブロック

  と呼んだりする.
- 理由はセクタが小さすぎるからだろう.

FAT (File Allocation Table)

- 論理フォーマットの一種
- 旧Windowsで利用されていた.
- 今でも小ぶりなUSBメモリでは利用される.
- Linuxでも読書きできる.

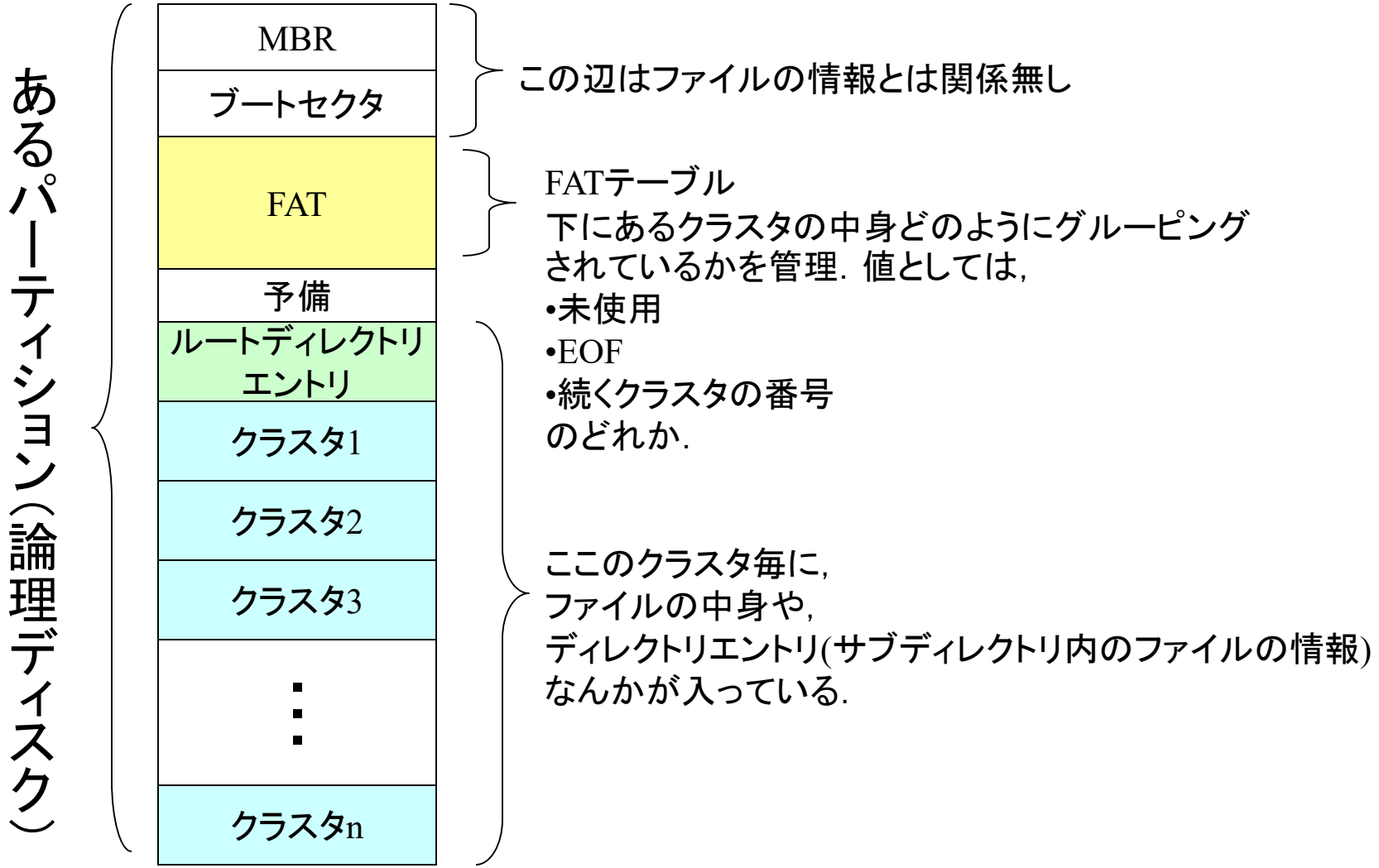


左記は私が使っていた
USBメモリ

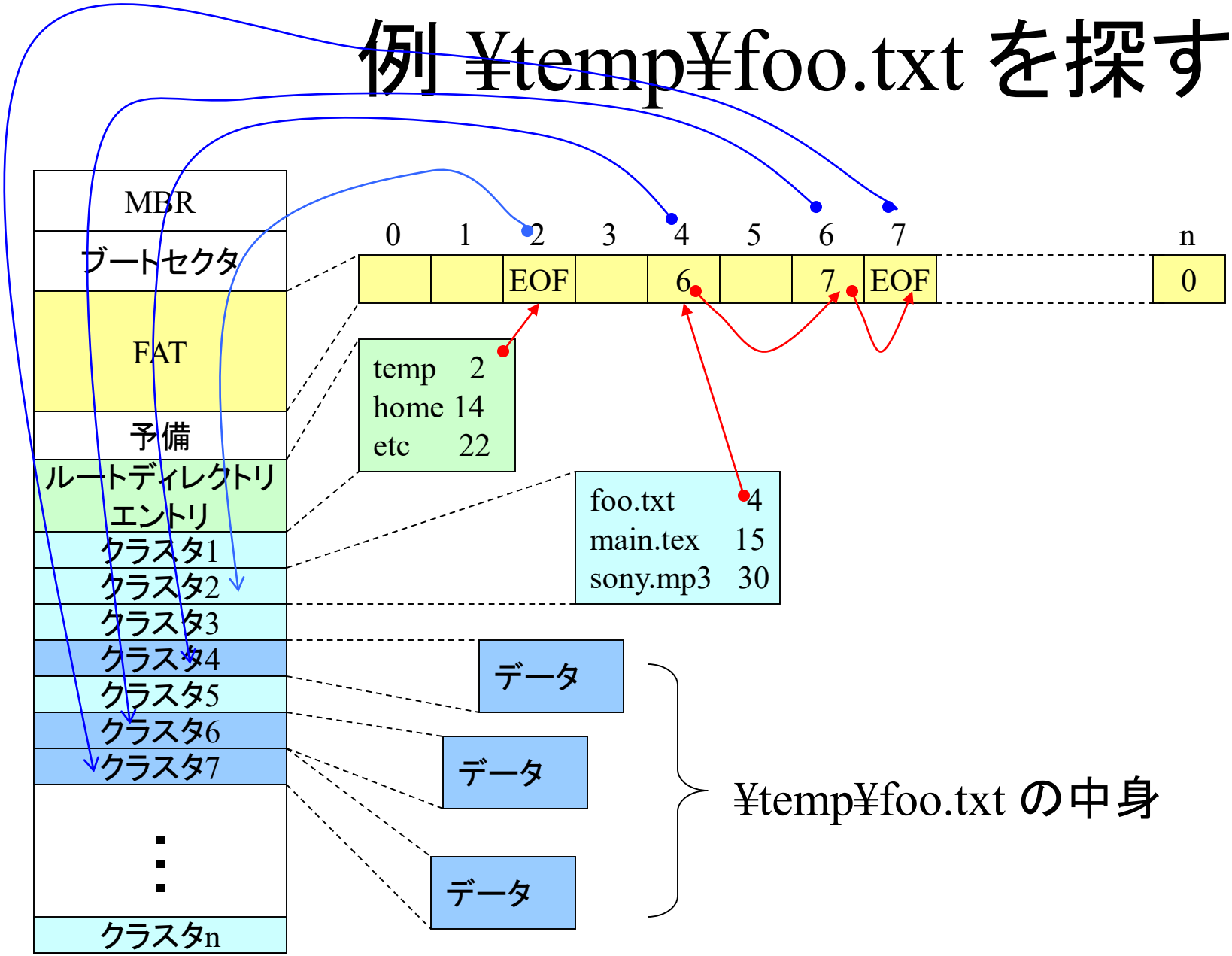
FATの内部形式

- 連続した2^n個のセクタをグループ化してクラスタとする.
  - とはいえディスクの先頭部分は例外
- ファイルは1個以上のクラスタに分割して配置されている.
- ファイルがどこにあるかの情報はFATとディレクトリエントリという情報で管理されている.

FATの内部構造


あるパーティション（論理ディスク）
MBR
ブートセクタ
この辺はファイルの情報とは関係無し
FAT
FATテーブル
下にあるクラスタの中身どのようにグルーピング
されているかを管理．値としては，
•未使用
•EOF
•続くクラスタの番号
のどれか．
予備
ルートディレクトリ
エントリ
クラスタ1
クラスタ2
クラスタ3
クラスタn
ここのクラスタ毎に，
ファイルの中身や，
ディレクトリエントリ(サブディレクトリ内のファイルの情報)
なんかが入っている．

例 ¥temp¥foo.txt を探す



FATの特徴

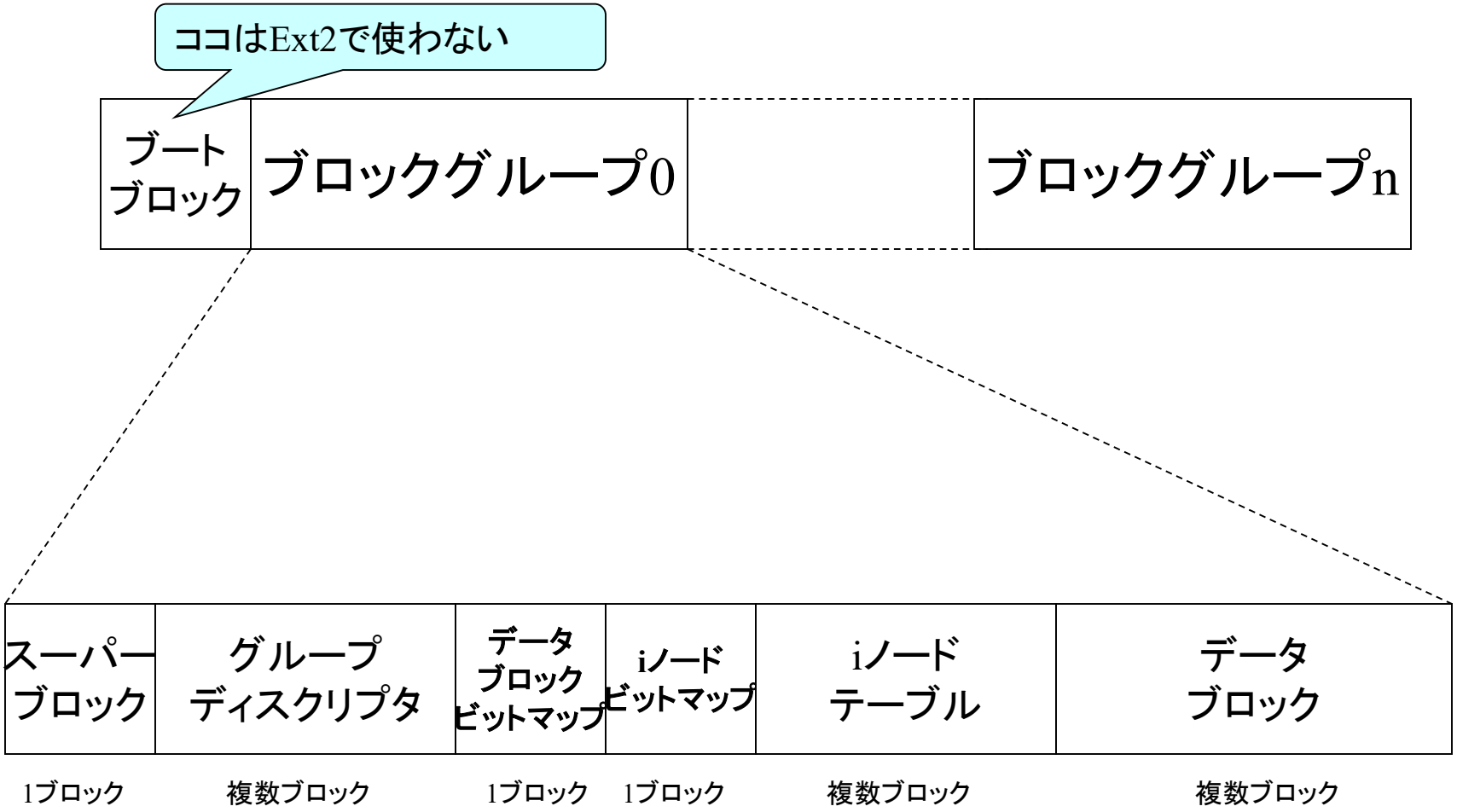
- 仕組みが単純 (?)
- (後述のext2でもそうだが)クラスタサイズが効率に影響する.
  - 小さいと大きなファイルを扱うのが不便.
    - 沢山のクラスタに実データが分散するため.
  - 大きいと小さいファイルを扱う場合無駄になる.
- ファイルの中身が複数のクラスタに分散し, クラスタが近接していない場合, アクセスが遅くなる.
  - いわゆる fragmentation (断片化)というヤツ.

Ext2 拡張ファイルシステム2

- 論理フォーマットの種類の1つ
- HDだけでなく，例えばUSBメモリ等もExt2でフォーマットできる.

- Linuxで基盤となるファイルシステムの種類.
  - 現在では ext4 やXFSが主流だが.
- パーティションをブロックという単位で分割して管理する.
  - 通常，1ブロック 1024B～4096B
  - 1セクタが512Bくらいなので，2～8セクタをグループ化
- ブロックをグルーピングし，同一グループのブロックは隣接トラックに配置される．よって，同一ブロック内のデータは短い時間間隔でアクセスできる.
  - ブロックグループ

パーティション内の構造



ブロックグループ内の項目

- スーパーブロック (1ブロック)
  - パーティションの情報が書いてある.
  - 各ブロックグループに複製を持っており，実際使うのは，グループ0のもののみ.
- グループディスクリプタ (複数ブロック)
  - グループの情報が書いてある.
  - 各ブロックグループに複製を持っており，実際使うのは，グループ0のもののみ.
- データブロックビットマップ (1ブロック)
  - データブロックが使用されているか否かを0/1情報で記述したビット列.
- iノードビットマップ (1ブロック)
  - データブロックと同様.
- iノードテーブル (複数ブロック)
  - 後述.
- データブロック (複数ブロック)
  - 個々のファイルの中身が入っているブロック郡

iノード

- Ext2ファイルシステム内のファイルはiノード番号で区別されている.
  - ファイル名は相対的な名前に過ぎない.
- ファイル固有の情報は,
  - ファイルの種類とアクセス権 (16bit)
  - 所有者のID (16bit)
  - バイト数 (32bit)
  - ハードリンクのカウンタ (16bit)

  等がある.
- 個々のiノードの情報は128バイトの構造体にまとめられている.

iノードテーブル

- 前述のiノードを示す構造体のインスタンス(1個128バイト)が並んでいるテーブル.
- ブロックサイズが1024Bなら,
  - 1ブロックに1024/128=8個のインスタンスが入る.
  - iノードビットマップは8192個のbitを持てる.
    - 別に最大bit数を利用しなくても良いが，利用したとすると・・・
  - iノードテーブルのために，最大 8192/8=1024 ブロックが必要となる.

Ext2でのファイルへたどり着く手順

1. ファイル名 (パス名)から，そのファイル(が指す実体)のiノード番号を得る.
2. iノード番号をもとに，iノードテーブル内の該当する構造体ext2_inodeのインスタンスを得る.
3. 構造体メンバーにファイル(の中身)が格納されているデータブロックの番号配列(__u32　i_block[EXT2_N_BLOCKS])があるので，それに従い，ファイルの中身をデータブロックから引きずり出す.

ext2_inodeの一部

```
struct ext2_inode {
        __u16   i_mode;         /* File mode */
        __u16   i_uid;          /* Owner Uid */
        __u32   i_size;         /* Size in bytes */
        __u32   i_atime;        /* Access time */
        __u32   i_ctime;        /* Creation time */
        __u32   i_mtime;        /* Modification time */
        __u32   i_dtime;        /* Deletion Time */
        __u16   i_gid;          /* Group Id */
        __u16   i_links_count;  /* Links count */
        __u32   i_blocks;       /* Blocks count */
        __u32   i_flags;        /* File flags */
        union {
                // ** 省略
        } osd1;                 /* OS dependent 1 */
        __u32   i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
        __u32   i_version;      /* File version (for NFS) */
        __u32   i_file_acl;     /* File ACL */
        __u32   i_dir_acl;      /* Directory ACL */
        __u32   i_faddr;        /* Fragment address */
        union {
                // ** 省略
        } osd2;                 /* OS dependent 2 */
};
```

全部で128バイト

ここが実際のデータが入っているブロックを格納している配列.
EXT2_N_BLOCKS は 15
コードの181行あたり.
後述のように最後の3つが間接，二重間接，三重間接ブロックに使われている.

include/linux/ext2_fs.h
の217行目あたりから

iノードとファイルの種類

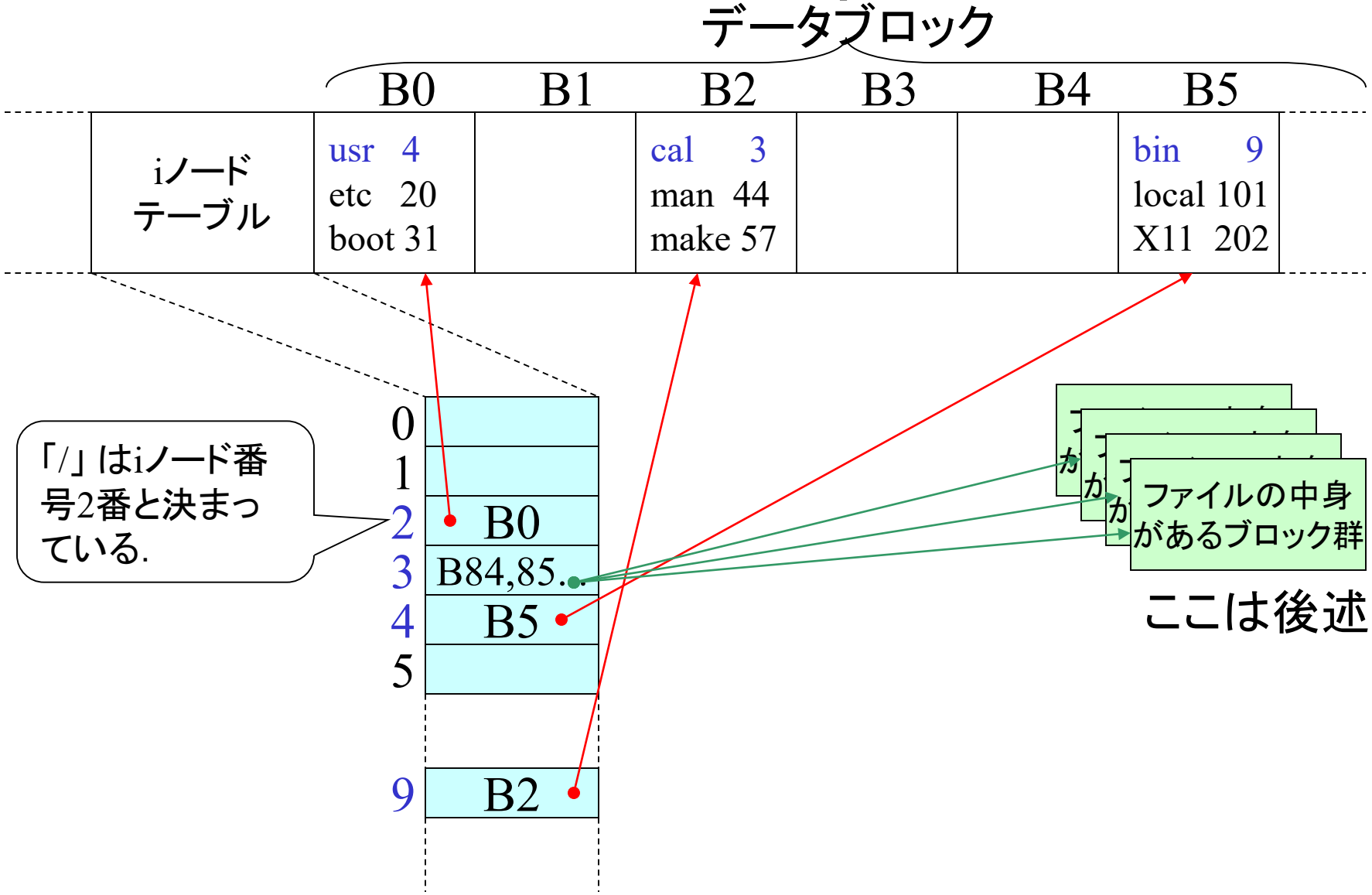
- Linux(UNIX)では全てのファイルにiノード(番号)がある.
- ディレクトリもシンボリックリンクもiノード番号を持つ.

ことを留意しておいてください.

パス名からiノード番号を得る

1. ルートディレクトリ / のiノード番号は「2」と決まっているので，現在の注目するiノードを2に設定する.
2. 注目しているiノードの構造体インスタンスに指定されているデータブロックの中身を見て，ディレクトリ内にあるファイル名とiノード番号の対応表を得る.
   a. 知りたいパスがディレクトリ内にあれば，対応する番号が結果.
   b. まだパスの途中なら，パスの途中であるディレクトリに対応するiノード番号を注目するiノードに設定しなおし，2に戻る.

例: /usr/bin/cal の番号を探す



iノード構造体とファイルの中身

- 前述のようにiノード構造体は(たった)128Bしかない.
- iノードが指しているファイルやディレクトリの中身は別途，データブロックに格納されている.
  - ディレクトリの場合，属しているファイルのリスト
- 中身が入っているデータブロックのありかは構造体のメンバ，i_block[] 配列に記録されている.

ext2_inodeの一部

```
struct ext2_inode {
        __u16   i_mode;         /* File mode */
        __u16   i_uid;          /* Owner Uid */
        __u32   i_size;         /* Size in bytes */
        __u32   i_atime;        /* Access time */
        __u32   i_ctime;        /* Creation time */
        __u32   i_mtime;        /* Modification time */
        __u32   i_dtime;        /* Deletion Time */
        __u16   i_gid;          /* Group Id */
        __u16   i_links_count;  /* Links count */
        __u32   i_blocks;       /* Blocks count */
        __u32   i_flags;        /* File flags */
        union {
                // ** 省略
        } osd1;                 /* OS dependent 1 */
        __u32   i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
        __u32   i_version;      /* File version (for NFS) */
        __u32   i_file_acl;     /* File ACL */
        __u32   i_dir_acl;      /* Directory ACL */
        __u32   i_faddr;        /* Fragment address */
        union {
                // ** 省略
        } osd2;                 /* OS dependent 2 */
};
```

全部で128バイト

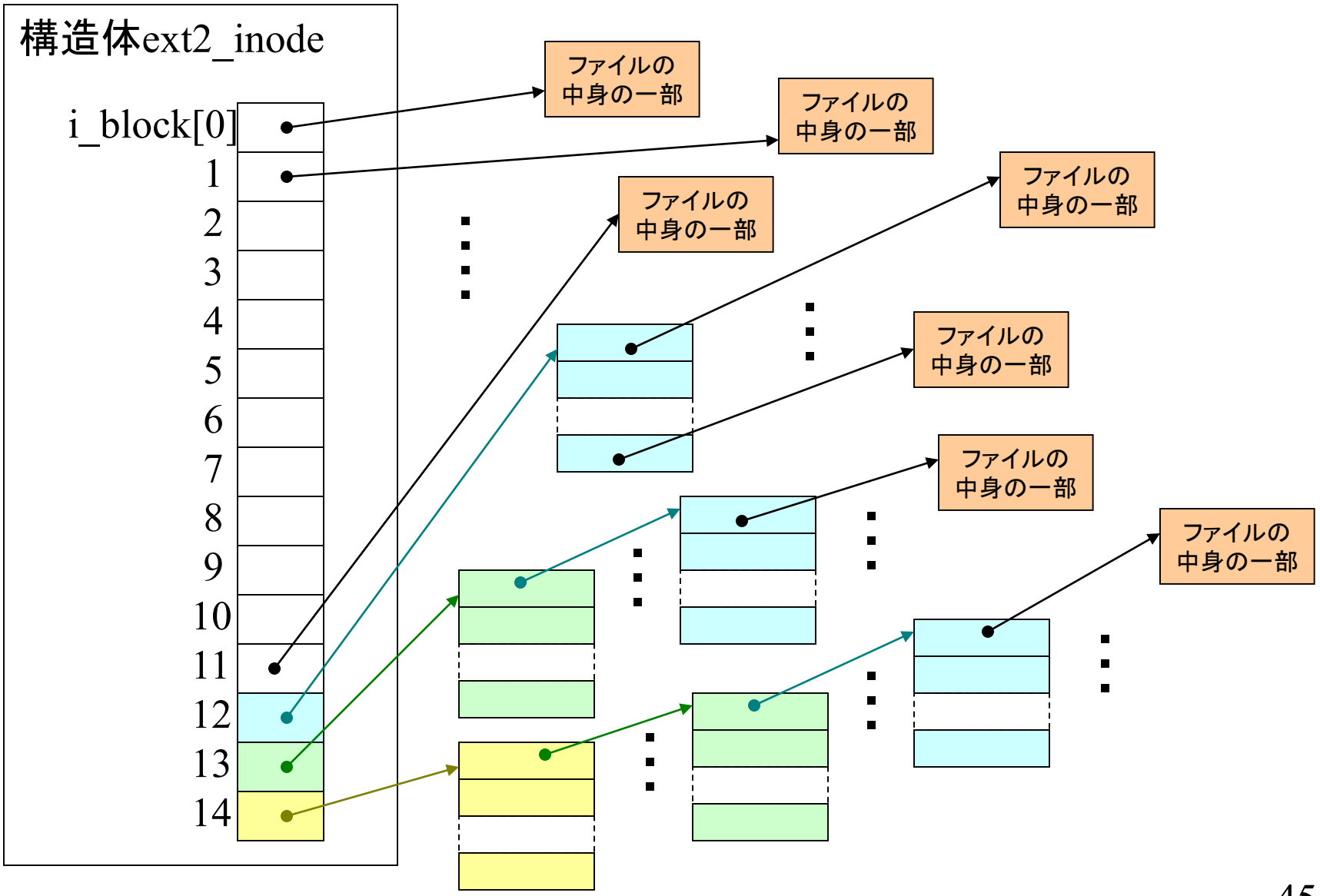
ここが実際のデータが入っているブロックを格納している配列.
EXT2_N_BLOCKS は 15
コードの181行あたり.
後述のように最後の3つが間接，二重間接，三重間接ブロックに使われている.

include/linux/ext2_fs.h
の217行目あたりから

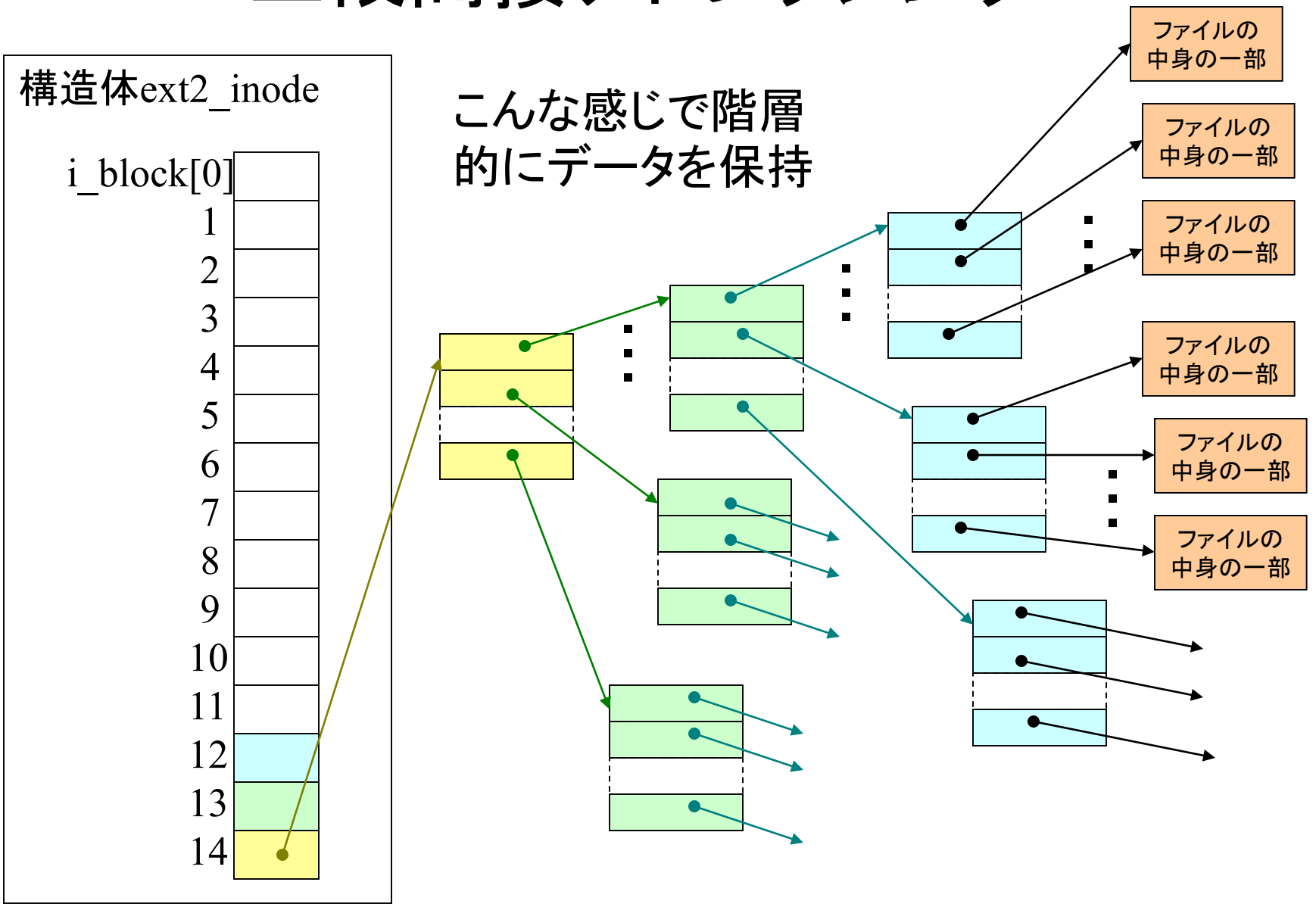
メンバ i_block[]

- EXT2_N_BLOCKS個の配列，通常この値は15.
- 配列要素は以下の4種類に分かれる.
  - i_block[0]～[11]の12個: 直接アドレッシング
    - ファイルの中身が入るデータブロックを直接指す.
  - i_block[12] 一段間接アドレッシング
  - i_block[13] 二段間接アドレッシング
  - i_block[14] 三段間接アドレッシング
    - 間接アドレッシングについては後述

iノード構造体とi_block[]のイメージ



三段間接アドレッシング

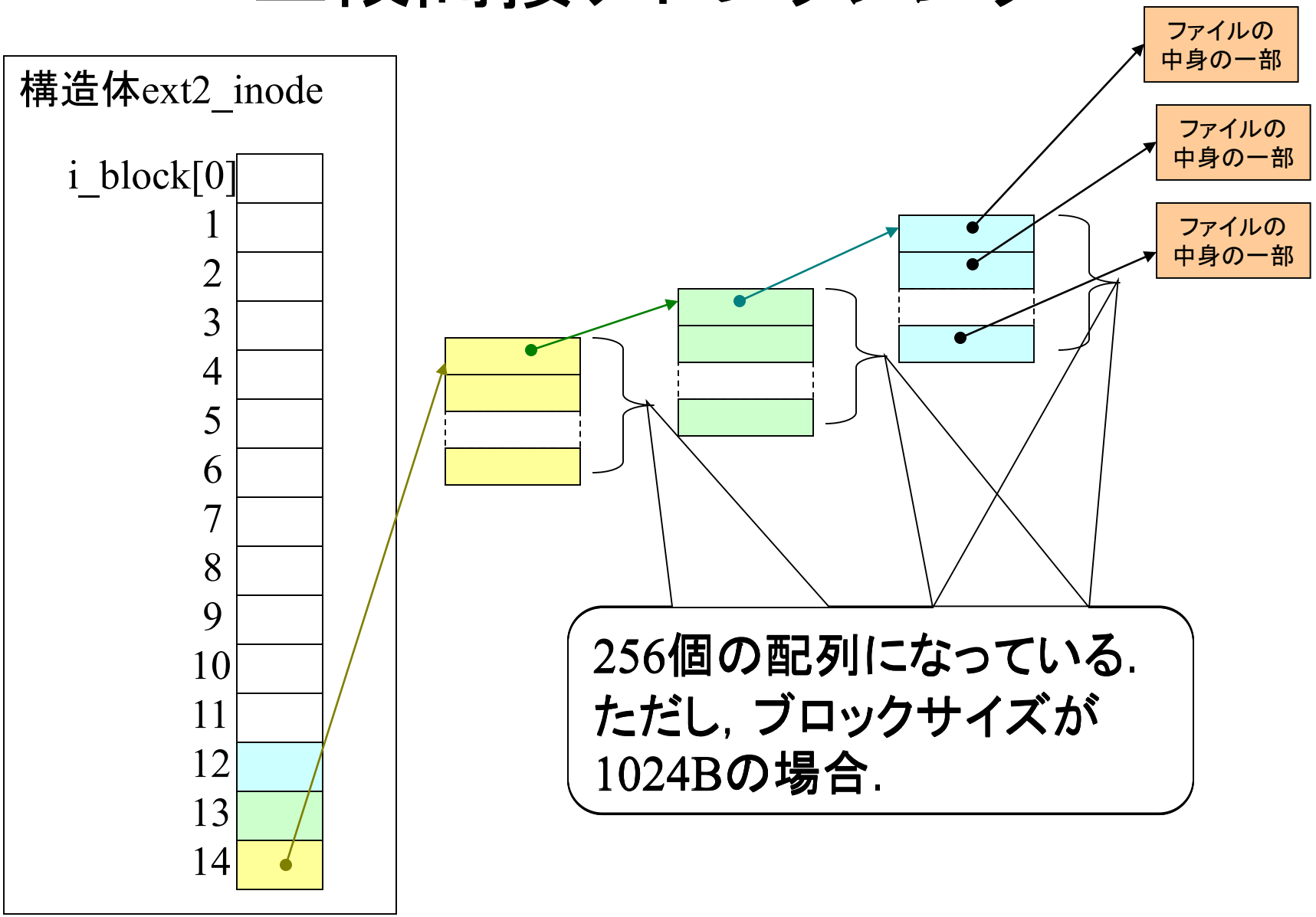


扱えるファイルサイズは？

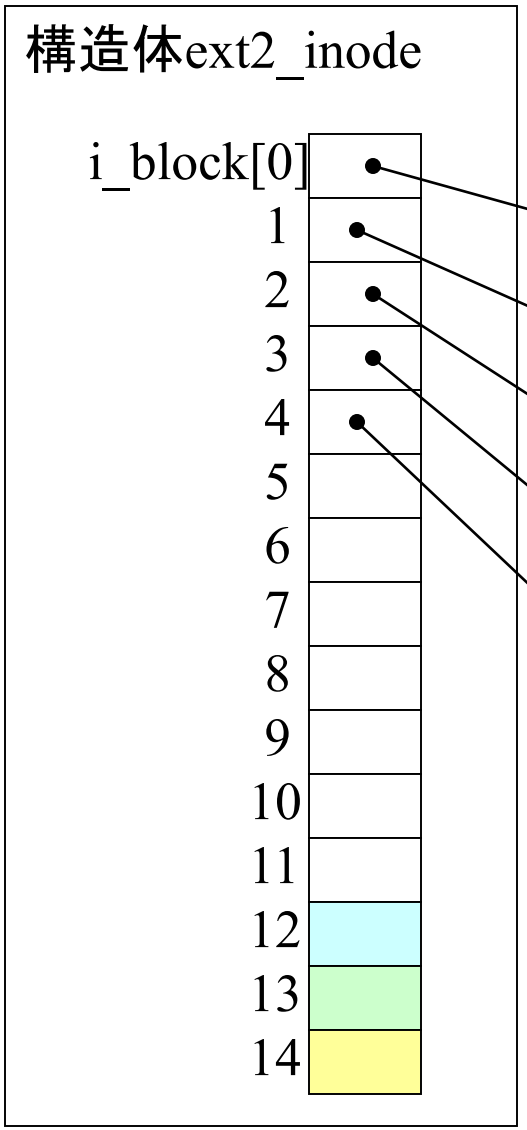
- 1ブロックを，1024Bとすると，間接ブロックを使わないと，1024*12≒12KBのファイルしか扱えない.
- 間接ブロック内に1つのブロックの位置を保存するには4B必要らしいので，間接ブロックからは，1024/4=256個のブロックを指すことができる.
- 13個目以降の要素では，
  - 間接 256*1024B≒262KB
  - 二重 256*256*1024B≒67MB
  - 三重 256*256*256*1024B≒17GB

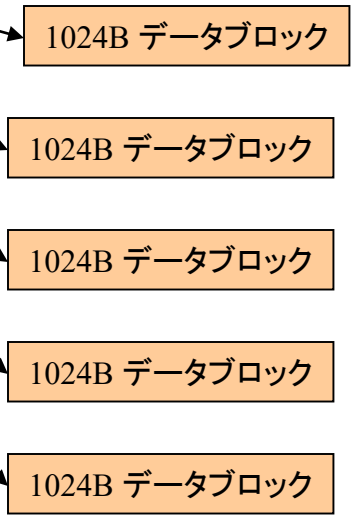
  結構大きなファイルが扱えますな.
- 実際はハードウェアの制約等により，際限なく大きなファイルが扱えるわけでない.

三段間接アドレッシング



ファイルの格納例 (1)



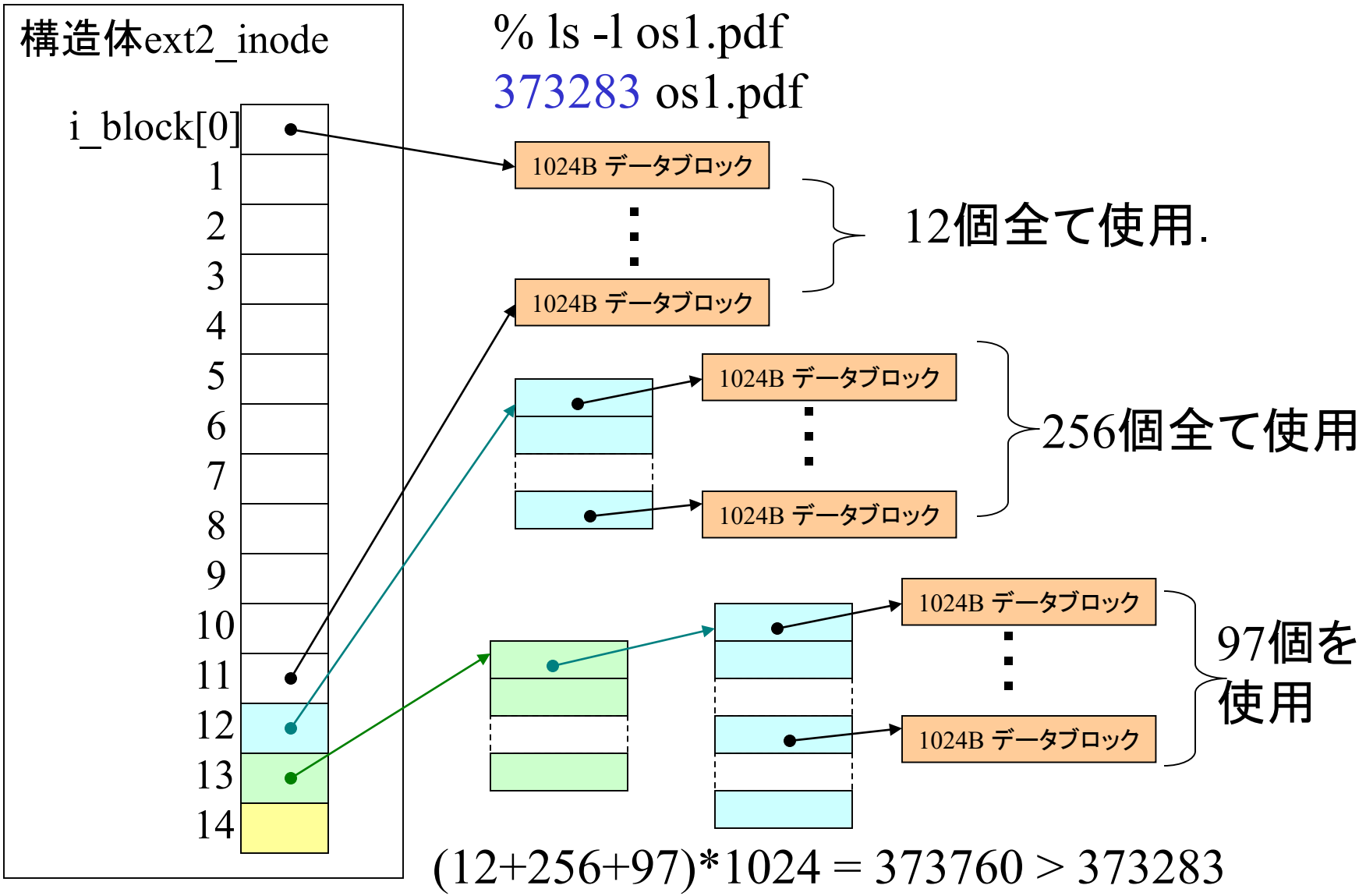


% ls -l exercise01/index.html
4341 exercise01/index.html

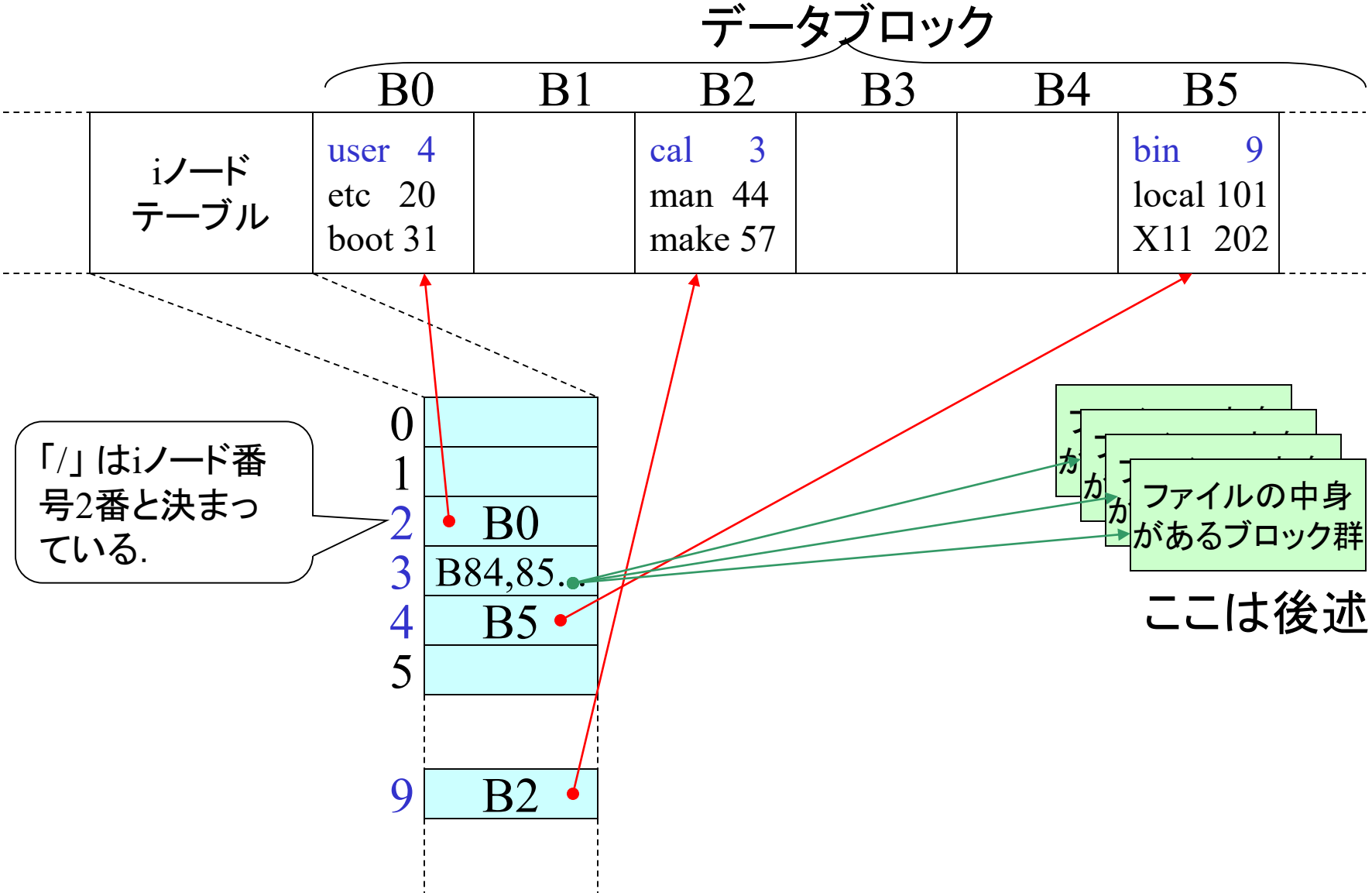
直接ブロック5個利用.

最後のブロックは245B
しか使ってないはず.
(4341-1024*4=245)

ファイルの格納例 (2)


構造体ext2_inode
i_block[0]
1
2
3
4
5
6
7
8
9
10
11
12
13
14
% ls -l os1.pdf
373283 os1.pdf
1024B データブロック
1024B データブロック
12個全て使用.
1024B データブロック
1024B データブロック
256個全て使用
1024B データブロック
1024B データブロック
97個を使用
(12+256+97)*1024 = 373760 > 373283

ディレクトリの中身もデータブロック



ディレクトリのデータブロック

- 構造体 ext2_dir_entry_2 のインスタンスが詰まっている.
- 上記構造体の一つ一つが，そのディレクトリ内にある他のファイルの情報を保持している.
- この構造体は可変長.
  - 最後のメンバーが文字列であるため.

ext2_dir_entry_2

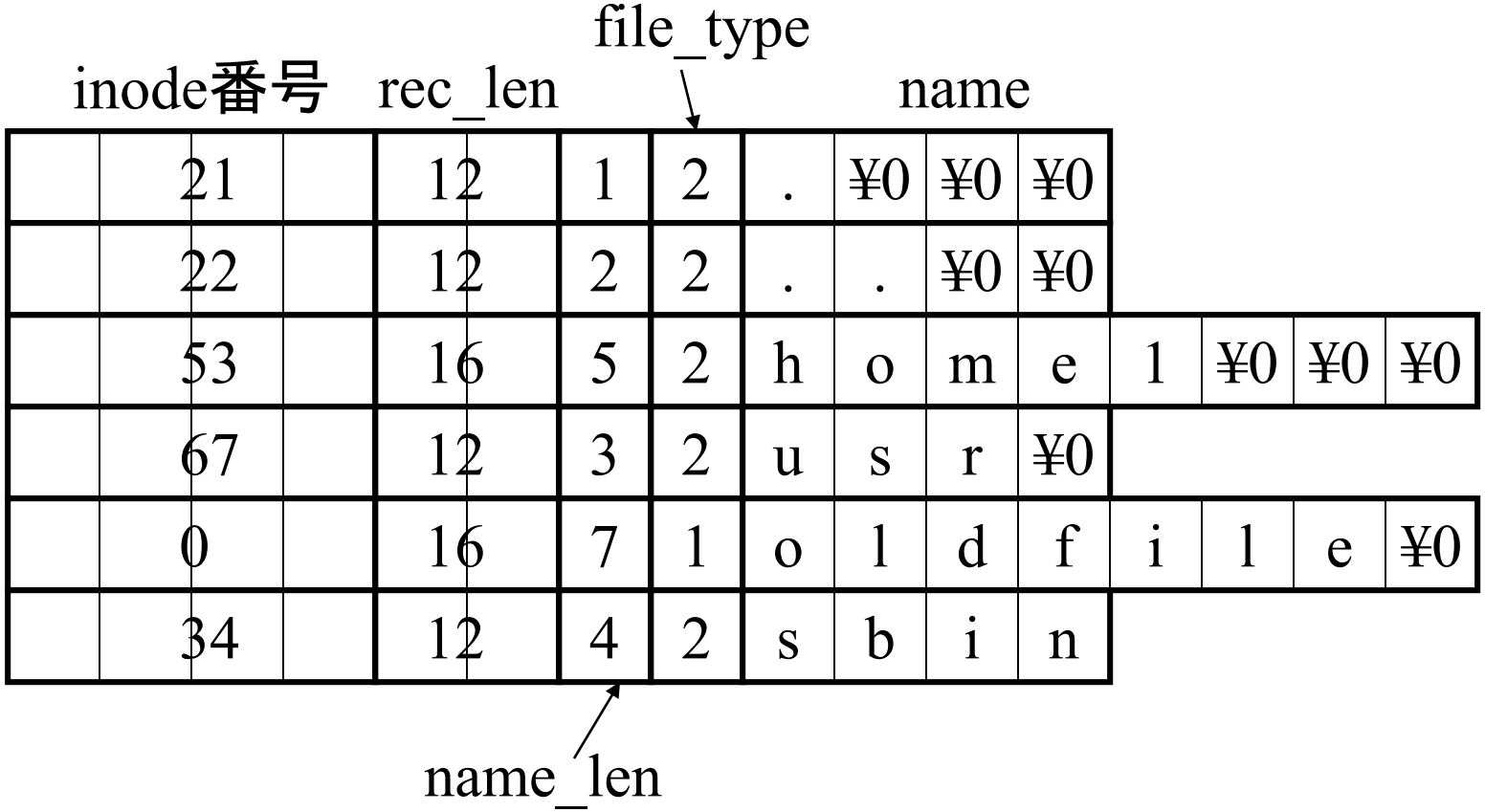
型: 1 通常ファイル, 2 ディレクトリ, 7 シンボリックリンク 他

```
// include/linux/ext2_fs.h の501行目

struct ext2_dir_entry_2 {
    __u32   inode; // そのディレクトリのiノード番号
    __u16   rec_len; // 構造体のサイズ, name[]のため可変
    __u8    name_len; // ファイル名の長さ ¥0 は含まず
    __u8    file_type;              // ファイルの型番号
    char    name[EXT2_NAME_LEN]; // ファイル名
};
```

通常, 255

効率化のため4の倍数長になっている. 不要な部分には ¥0 文字が詰めてある.

例: とあるデータブロックの中身



inode番号	rec_len	name_len	file_type	name
21	12	1	2	. ¥0 ¥0 ¥0
22	12	2	2	. . ¥0 ¥0
53	16	5	2	h o m e 1 ¥0 ¥0 ¥0
67	12	3	2	u s r ¥0
0	16	7	1	o l d f i l e ¥0
34	12	4	2	s b i n

シンボリックリング

- リンク先のパスが60バイト以下の場合，ext2_inodeのメンバーであるi_block[]に埋め込む.
  - i_block[]は1個32ビット(4バイト)であるため，15×4=60個.
- 60バイトを超える場合は，普通のファイル同様，データブロック内に保存する.

ext3

- Ext2と互換性のあるジャーナリングファイルシステム
- 昨今のLinuxで標準のext4の前バージョンである.
- ext4, ext2との互換性がある.

ジャーナリング・ファイルシステム

- 時間のかかるファイルシステムの整合性チェックを短時間に行うための仕組み.
- 具体的には最近の更新情報をジャーナルと呼ばれる場所に格納し，整合性チェックを高速化する.

ファイルシステムの整合性

- ブロックの内容は通常，メモリに複製を作り(キャッシュと呼ぶ)，そちらを操作することで，高速化をしている.
- 実際のブロックの内容とキャッシュは最終的(システム停止時等)には一致していないといけない.
- これが一致しているかどうかをチェックするのが整合性.

Ext3の戦略

- ブロック単位で処理を行う.
- ジャーナルという特別なファイル領域を準備しておく.
- キャッシュのディスクへの書き戻しを以下の三段階で行う.
  1. ジャーナルへのコミット: キャシュをまずジャーナルに書き込む.
  2. ファイルシステムへのコミット: キャッシュを実際のファイルシステムに書き込む.
  3. ジャーナルに書いたものを破棄する.

何故，前述の戦略が良いか？

- ジャーナルへのコミット中にクラッシュが起きた場合
  - キャシュの更新自体，無かったことにする．
  - すなわち，ここ最近のファイル更新はパーになる．
  - しかし，ファイルシステムの一貫性は保たれる．
- ファイルシステムへのコミット中にクラッシュした場合
  - ジャーナルに更新結果があるのでそれをコピーすればよい．
  - よって更新結果を完全に復旧できる．
- 要はファイルシステム本体に中途半端な更新情報が書き込まれるのを防ぐことができる．

どこまでジャーナルに入れるか？

以下の三種類がある.

- journalモード
  - 全てのブロックをジャーナルに一度入れる.
  - 無論，遅いが安全.
- orderedモード
  - データブロック以外(メタデータと呼ばれる)のブロックのみジャーナルに入れる.
  - ファイルシステムの構造(内容ではない)の安全性を重視.
- writebackモード
  - ファイルシステムが更新したメタデータのみをジャーナルに入れる.

演習2

- 750000Bのファイルをext2ファイルシステムに格納する場合，以下のそれぞれのアドレッシングで扱うブロック数を計算せよ.

1. 直接アドレッシング
2. 一段間接アドレッシング
3. 二段間接アドレッシング
4. 三段間接アドレッシング

- ただし，1ブロックのサイズは1024Bとし，間接アドレッシングにおいて一つのブロックを参照するのに必要なバイト数は4Bであるとする.
- また，アドレッシングで使う配列サイズは15で，最初の12を直接アドレッシングで利用し，残り3つをそれぞれ一段，二段，三段間接アドレッシングで利用するものとする.
- 答えだけでなく，途中の考え方や計算式等も含め文書にまとめてPDFかDOCXの形式で提出せよ. JPG等の画像ファイルは不許可.
- 図を用いても良い，手書きのスキャンやスマホ写真したものをワード等に貼り込んでもよい.

以上

アンケートのほう，
よろしくご提出ください