オペレーティングシステム2004
ファイル管理 (1)

2003年11月5日

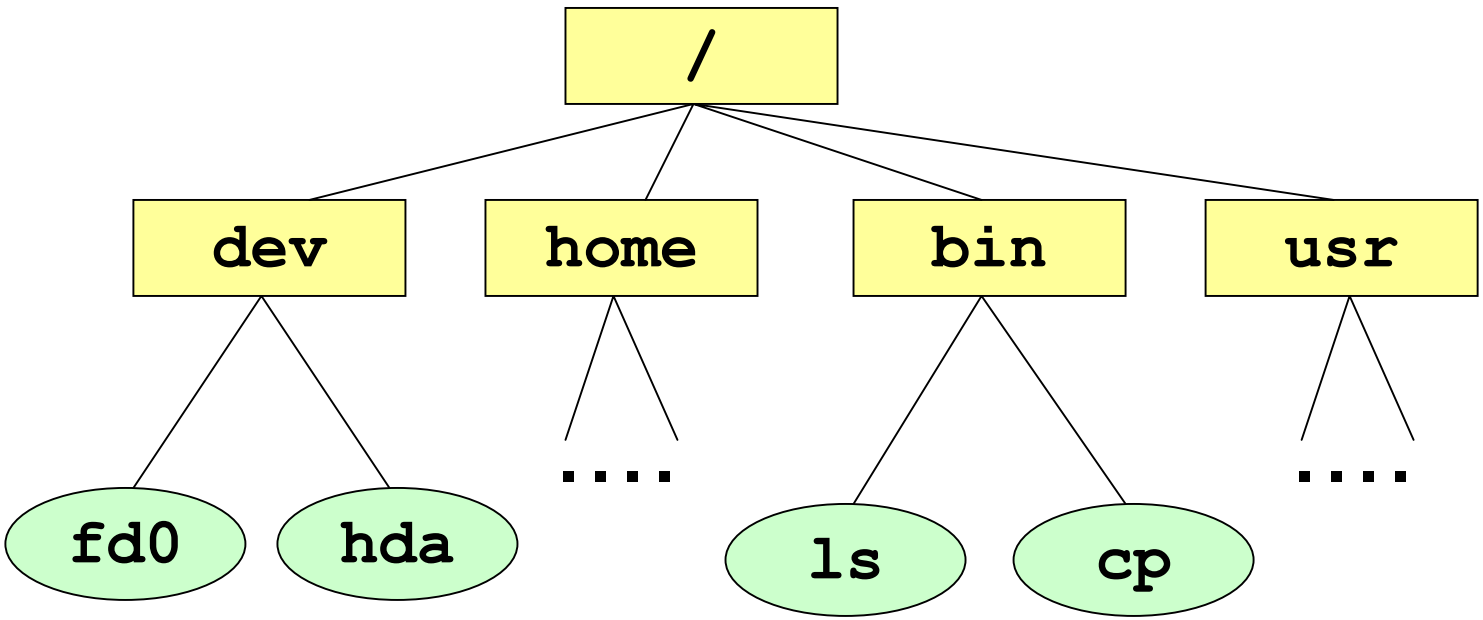
海谷 治彦

目次

- (UNIX系)ファイルシステムの概要
- プロセスとファイルの関係
- 仮想ファイルシステム VFS
  - C言語 関数へのポインタ
- ディスクの物理構造とフォーマット

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．

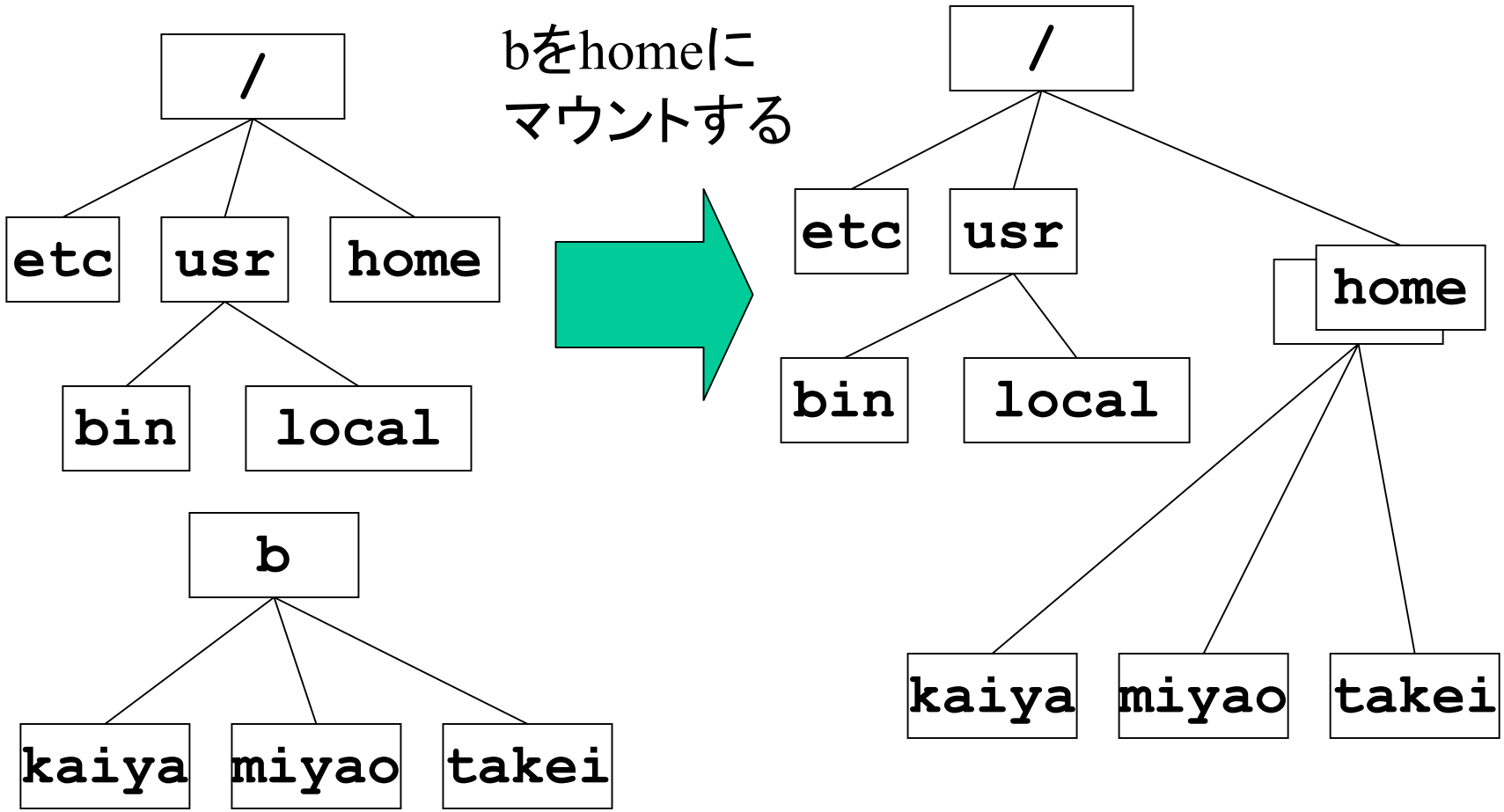


文献5 p.15

mount: ファイルシステムの接木

- 実際のファイルシステムは，複数のファイルシステムを接木して，一つの大きな木構造を成している.
- これによって，物理的/論理的に異なるファイルシステム群を1つの構造にまとめることができる.
  - 物理的: HDやFDやCDやネットワークデバイス
  - 論理的: Ext2やFATやNTFSやSMB等
    - Ext2はLinux標準のファイルシステム
    - FATは古いWindows(MSDOS)のファイルシステム
    - NTFSはWin/NtやXPのファイルシステム
    - SMBはWindowsのネットワークドライブ

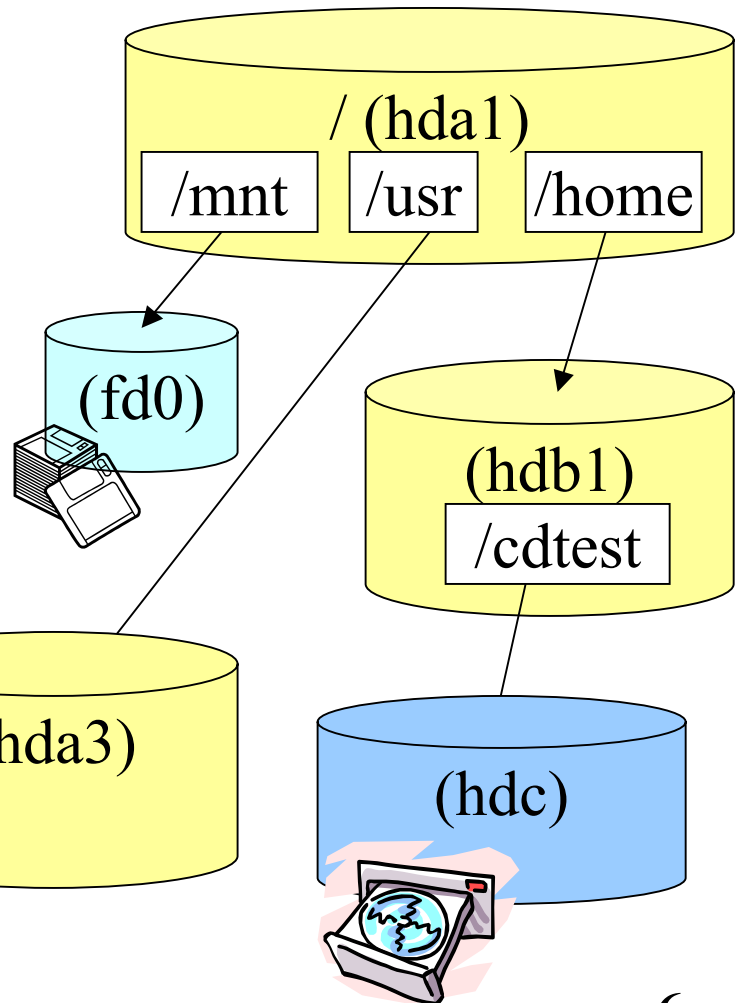
mountの概念図



文献1 p.45 改

mountの実際

```
[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 ~]%
```



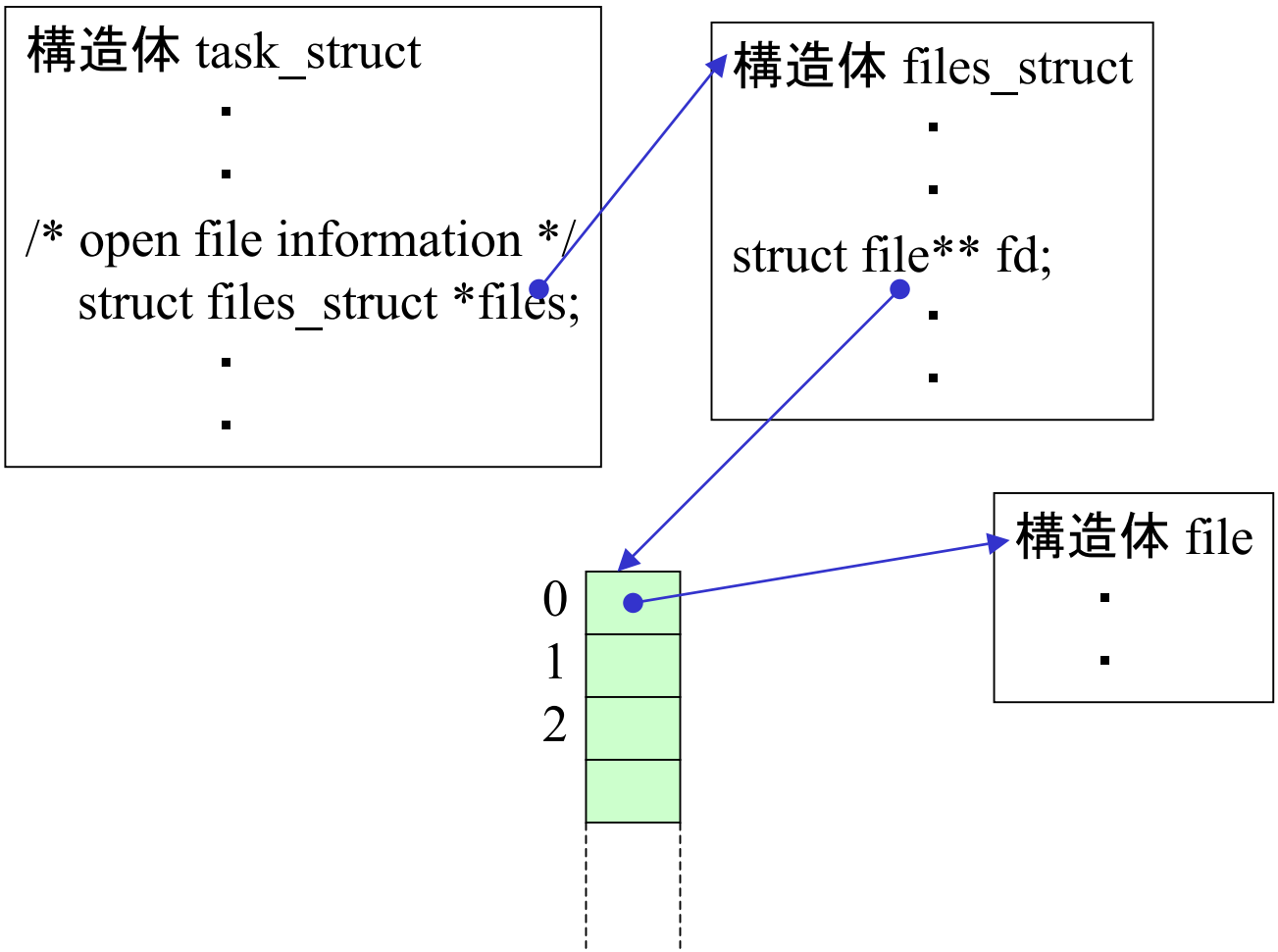
プロセスとファイル

- 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.
- 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.

File Descriptor

- あるプロセスが参照しているファイル(特殊ファイルを含む)は，File Descriptorと呼ばれる数値で識別される.
- 番号，0, 1, 2 は全プロセス共通に標準入力，出力，エラーに対応.
- この数値は構造体fileの配列の添え字になっている.
  - 構造体 fileについては後述.

構造体の関係


構造体 task_struct
/* open file information */
struct files_struct *files;
構造体 files_struct
struct file** fd;
構造体 file
0
1
2

openやfopen

- 普段，fopen関数なんかで開けてるFILEなんかも，結局は，前述のFile Descriptorを参照している.
- openシステムコールやread,writeシステムコールはFile Descriptorを直接使ってファイルの操作を行う.

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

この辺は本日扱わない.

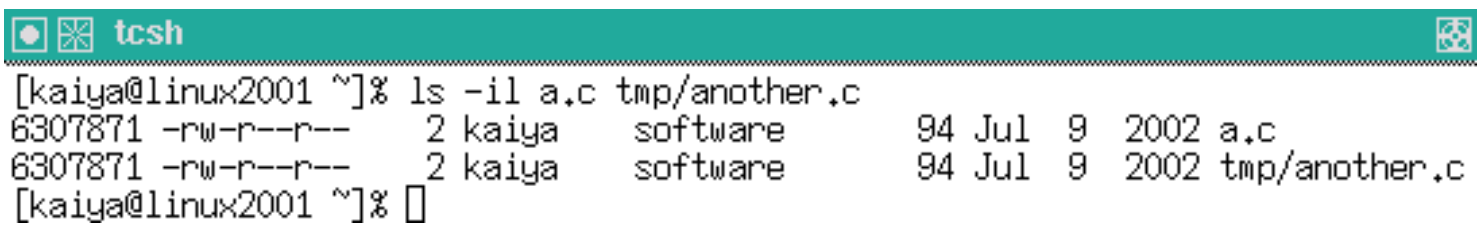
inode番号とリンク

- 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．

include/linux/fs.h の338行目辺りで定義

(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.



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.

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

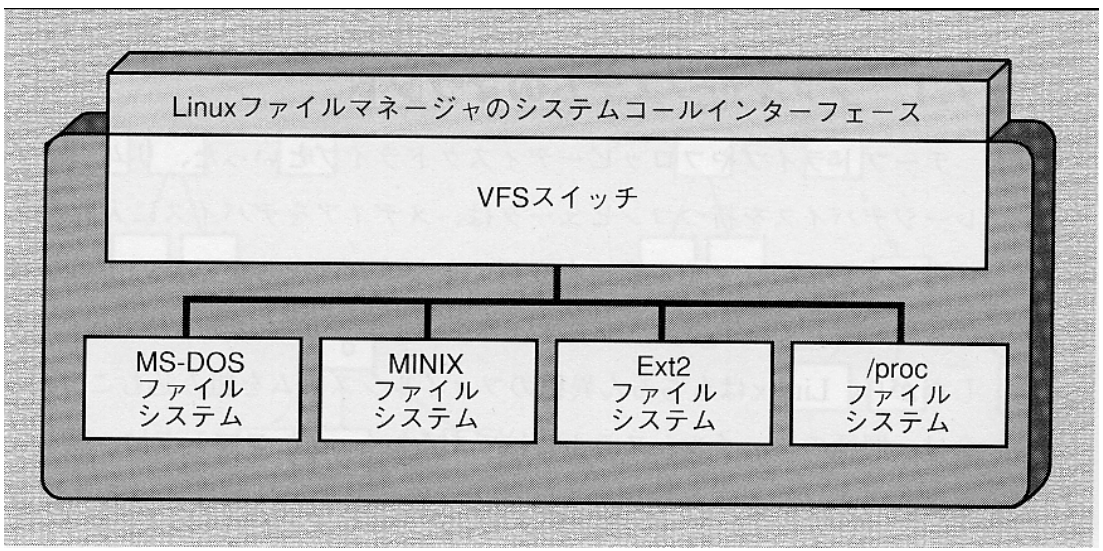
- open アクセスするためにファイルを開ける関数.
- close 逆に閉じる関数
- read 読む関数
- write 書く関数
- lseek 現在の読み書き位置を移動させる関数. (ビデオの早送り巻き戻しみたいなもの)
- rename ファイル名の付け替え
- unlink ディレクトリからのエントリ削除.

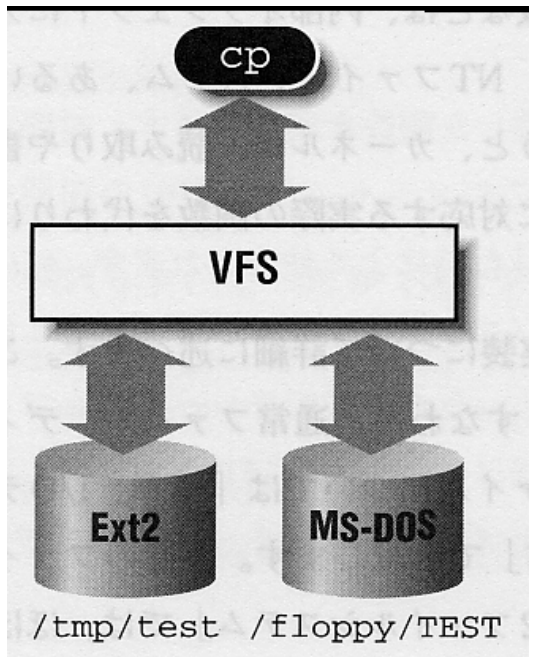
これらはシステムコールなので，実際には，カーネルへの作業要求依頼である.

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.

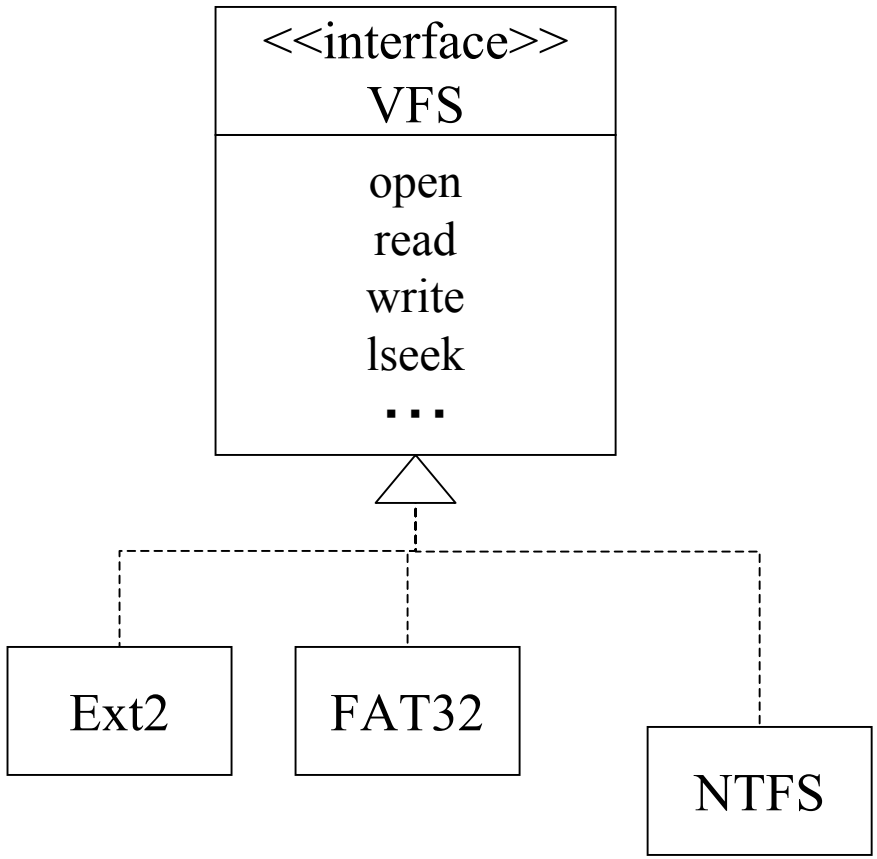
VFSの概念図


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

VFSはinterface

- VFSは(UML等でいうところの)Interfaceにあたる.
  - Linuxはオブジェクト指向言語で書いてあるのではないので概念上の話.



VFSが扱うファイルシステムの種類

- ディスクベースのファイルシステム
  - Ext2, FAT, NTFS, ISO9660 等のマシンに直接接続された装置内のファイルシステム群
- ネットワークファイルシステム
  - NFSやSMBなどのネットワークファイルシステム
- 特殊ファイルシステム
  - /proc/の下のファイルや /dev/pts等

特殊ファイルの例 端末がファイルに?!

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.

システム固有の関数までの道の例

1. ユーザーは**read()**等のシステムコールを呼ぶ.
2. カーネルは**sys_read()**を呼ぶ.
3. この関数は, カーネル内に保持されている開き済なファイルを示す構造体 **file**のインスタンスを読む.
4. **file**のメンバーに構造体**f_op**というのがあり, これに関数へのポインタが列挙されている.
5. **f_op**内のこのポインタ群がシステム固有の関数である.

file ->f_op->read(….) という間接的な呼び方

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

```
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.

システム固有の関数までの道

1. ユーザーは**read()**等のシステムコールを呼ぶ.
2. カーネルは**sys_read()**を呼ぶ.
3. この関数は, カーネル内に保持されている開き済なファイルを示す構造体 **file**のインスタンスを読む.
4. **file**のメンバーに構造体**f_op**というのがあり, これに関数へのポインタが列挙されている.
5. **f_op**内のこのポインタ群がシステム固有の関数である.

file ->f_op->read(….) という間接的な呼び方

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
}
```

構造体 file

include/linux/fs.h
の423行目くらい

```
struct file {
    struct file     *f_next, **f_pprev;
    struct dentry       *f_dentry;
    struct file_operations  *f_op; // コイツがInterfaceにあたる
    mode_t          f_mode;
    loff_t          f_pos;
    unsigned int        f_count, f_flags;
    unsigned long       f_reada, f_ramax, f_raend, f_ralen, f_rawin;
    struct fown_struct  f_owner;
    unsigned int        f_uid, f_gid;
    int         f_error;

    unsigned long       f_version;

    /* needed for tty driver, and maybe others */
    void            *private_data;
};
```

構造体 file_operations

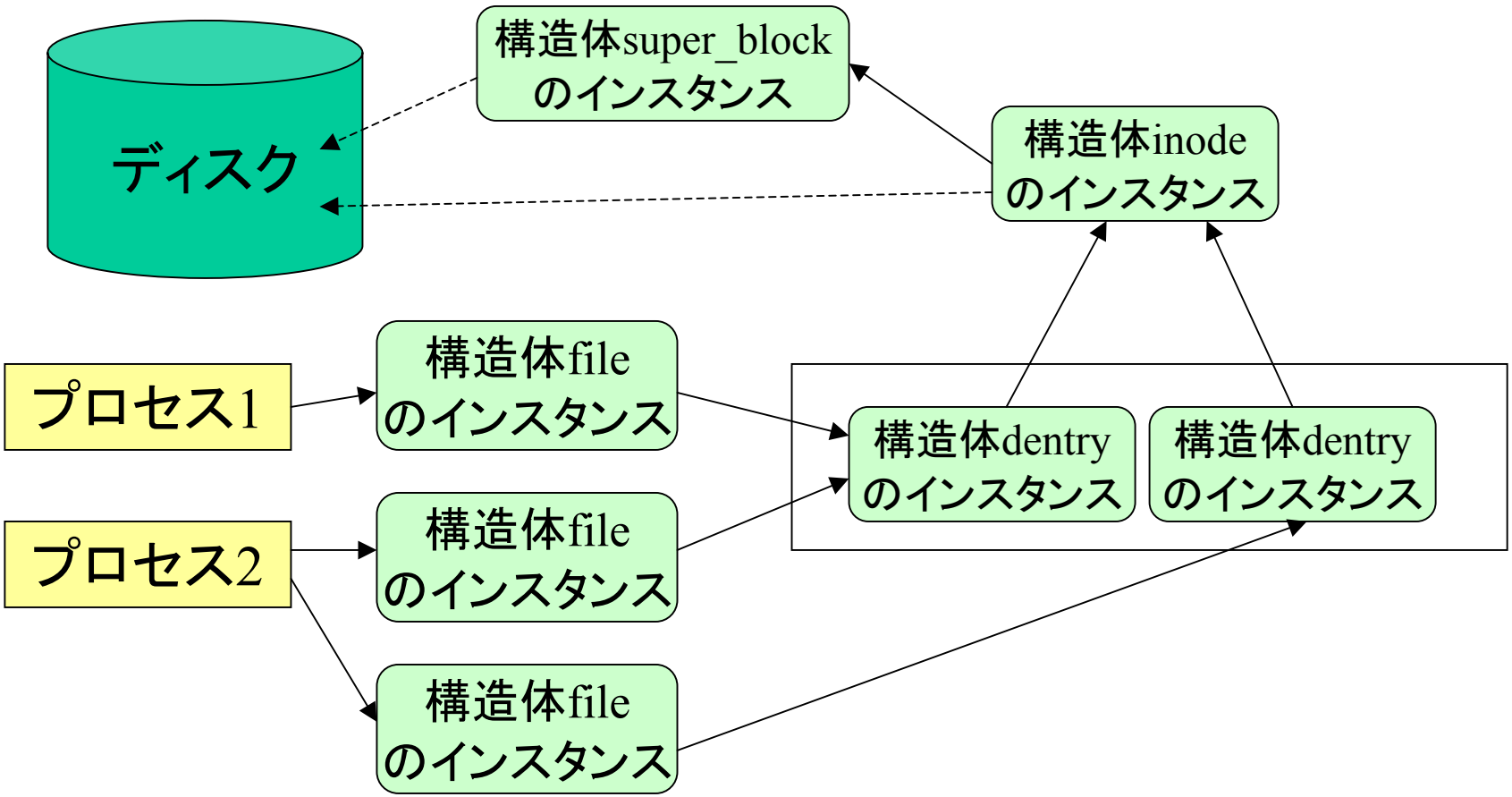
```
struct file_operations {
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct 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 *);
};
```

include/linux/fs.h
の810行目くらい

メンバが関数へのポインタとなっており，ポインタの指す実体がFSの実体毎に異なる.

全てのファイルシステムに全ての関数が定義されているわけではない.

プロセスとVFS間の関係



個々のプロセスがファイルシステムにたどり着くまで，いくつかの構造体のインスタンスを経由します．

各構造体の意味の説明

- file オープンされているファイルの情報を保持する. プロセスがファイルにアクセスしている間しかインスタンスは存在しない.
- dentry ディレクトリのエントリを現すインスタンス.
- inode 個々のファイルについての一般的情報.
- super_block マウントされたファイルシステムに関する情報を保持.

構造体 dentry

```
struct dentry {
    int d_count;
    unsigned int d_flags;
    struct inode  * d_inode;    /* Where the name belongs to - NULL is negative$
    struct dentry * d_parent;   /* parent directory */
    struct dentry * d_mounts;   /* mount information */
    struct dentry * d_covers;
    struct list_head d_hash;    /* lookup hash list */
    struct list_head d_lru;     /* d_count = 0 LRU list */
    struct list_head d_child;   /* child of parent list */
    struct list_head d_subdirs; /* our children */
    struct list_head d_alias;   /* inode alias list */
    struct qstr d_name;
    unsigned long d_time;       /* used by d_revalidate */
    struct dentry_operations  *d_op;
    struct super_block * d_sb;  /* The root of the dentry tree */
    unsigned long d_reftime;    /* last time referenced */
    void * d_fsdata;        /* fs-specific data */
    unsigned char d_iname[DNAME_INLINE_LEN]; /* small names */
};
```

include/linux/dcash.h
の58行目くらい

構造体 inode

```
struct inode {
    struct list_head    i_hash;
    struct list_head    i_list;
    struct list_head    i_dentry;

    unsigned long       i_ino;
    unsigned int        i_count;
    kdev_t          i_dev;
    umode_t         i_mode;
    nlink_t         i_nlink;
    uid_t           i_uid;
    gid_t           i_gid;
    kdev_t          i_rdev;
    off_t           i_size;
    time_t          i_atime;
    time_t          i_mtime;
    time_t          i_ctime;
    unsigned long       i_blksize;
    unsigned long       i_blocks;
    unsigned long       i_version;
    unsigned long       i_nrpages;
    struct semaphore    i_sem;
    struct semaphore    i_atomic_write;
    struct inode_operations *i_op;
    struct super_block  *i_sb;
    struct wait_queue   *i_wait;
    struct file_lock    *i_flock;
    struct vm_area_struct   *i_mmap;
    struct page     *i_pages;
    struct dquot        *i_dquot[MAXQUOTAS];
// 長いので以下省略
};
```

inlcude/linux/fs.h の338行目くらい

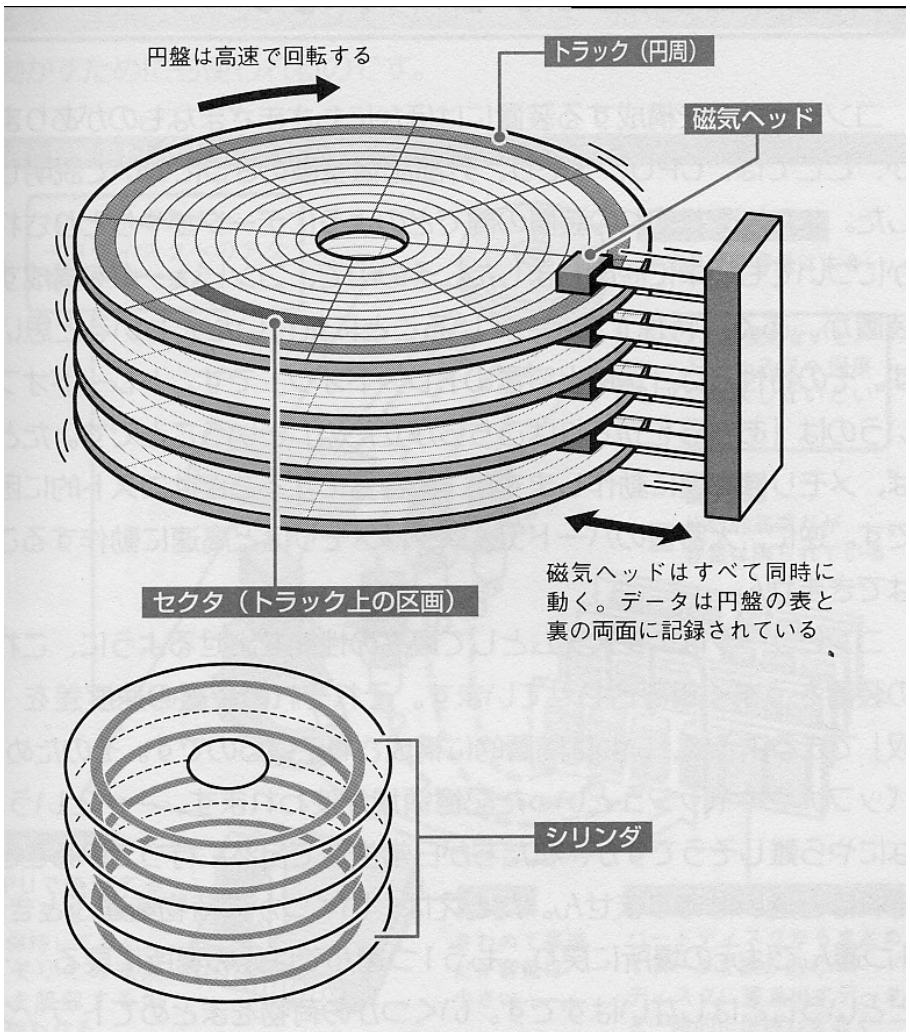
構造体 super_block

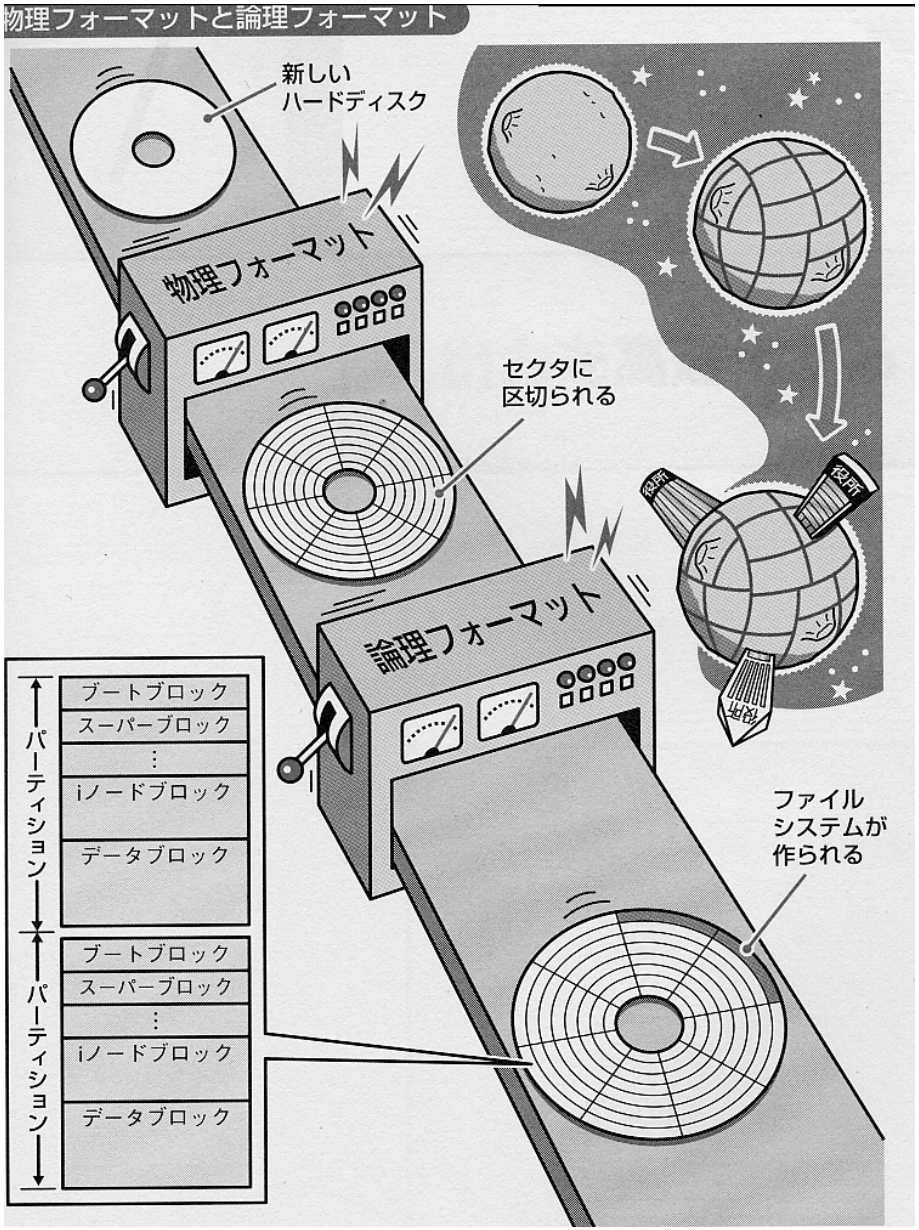
```
struct super_block {
    struct list_head    s_list;     /* Keep this first */
    kdev_t          s_dev;
    unsigned long       s_blocksize;
    unsigned char       s_blocksize_bits;
    unsigned char       s_lock;
    unsigned char       s_rd_only;
    unsigned char       s_dirt;
    struct file_system_type *s_type;
    struct super_operations *s_op;
    struct dquot_operations *dq_op;
    unsigned long       s_flags;
    unsigned long       s_magic;
    unsigned long       s_time;
    struct dentry       *s_root;
    struct wait_queue   *s_wait;

    struct inode        *s_ibasket;
    short int       s_ibasket_count;
    short int       s_ibasket_max;
    struct list_head    s_dirty;    /* dirty inodes */
   // 以下長いので略
}
```

include/linux/fs.h
の528行目くらい

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