10 UNIX ファイル・システム

10–1 UNIX ファイルシステムの特徴

OS/360のファイルシステムとの比較：

	OS名	
	OS/360	UNIX
ファイル名空間	フラットな構造	木構造
ファイル編成	順編成, 直接編成	順編成のみ
アクセス方法	SAM,DAM, PAM,VSAM	特にない。(指定されたファイル位置から順次1バイトずつアクセスするのみ。)
ファイルのデータ構造	各種レコード形式が存在。（情報の基本単位はレコード。）	バイト列

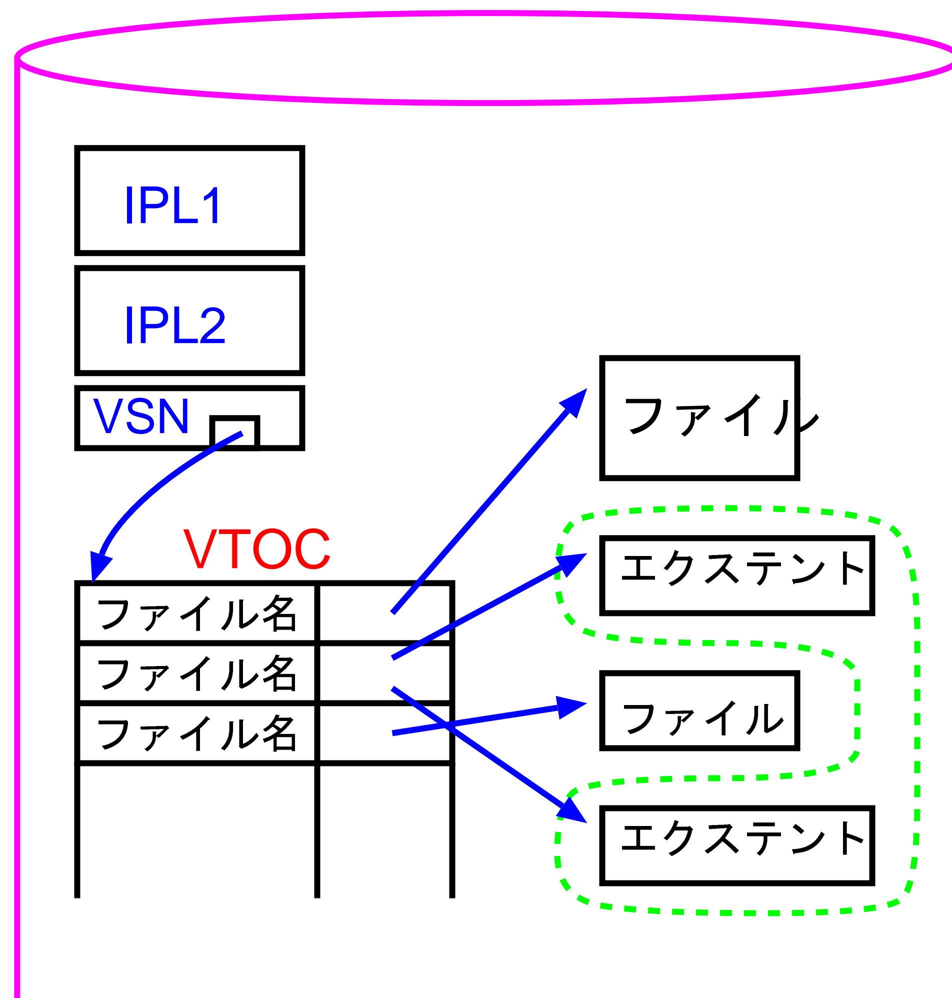
	OS名	
	OS/360	UNIX
ファイル管理	ディスクの中に**ボリューム目次 (VTOC)** と呼ばれる管理情報を置き、そこにディスク内の全てのファイルの名前と記憶位置を記録する。	個々のファイル(やディレクトリ) を管理するために **i-ノード**と呼ばれる構造体を用意し、そこに保護モードやユーザID, ファイル実体のアドレス等を保持する。 このi-ノードとOS/360における「ボリューム目次」に相当するディレクトリを組み合わせて木構造を構成する。

OS/360におけるファイル管理(概略)：

- ディスクの中にボリューム目次(VTOC)と呼ばれる管理情報を置き、そこにディスク内の全てのファイルの名前と記憶位置を記録する。

ファイルが記憶される媒体のことを総称的にボリュームと呼ぶ。

- ボリュームの中の最初の2つのブロックはIPL が占め、その後のボリューム通し番号(VSN)のレコードの中にボリューム目次のアドレスが置かれている。
このアドレスを用いてボリューム目次へのアクセス、更にはボリューム内のファイルへのアクセスが為される。



UNIXにおけるファイル管理(概略)：

- 個々のファイル(やディレクトリ)を管理するために**i-ノード**と呼ばれる構造体をファイル(やディレクトリ)毎に用意し、そこに保護モードやユーザID等の属性、ファイル実体のアドレス等を保持する。
- ディレクトリ(ファイル)はOS/360における「ボリューム目次」に相当するもので、そのディレクトリ内のファイル(やディレクトリ)の 名前 と i-ノード番号 のペアを保持する。

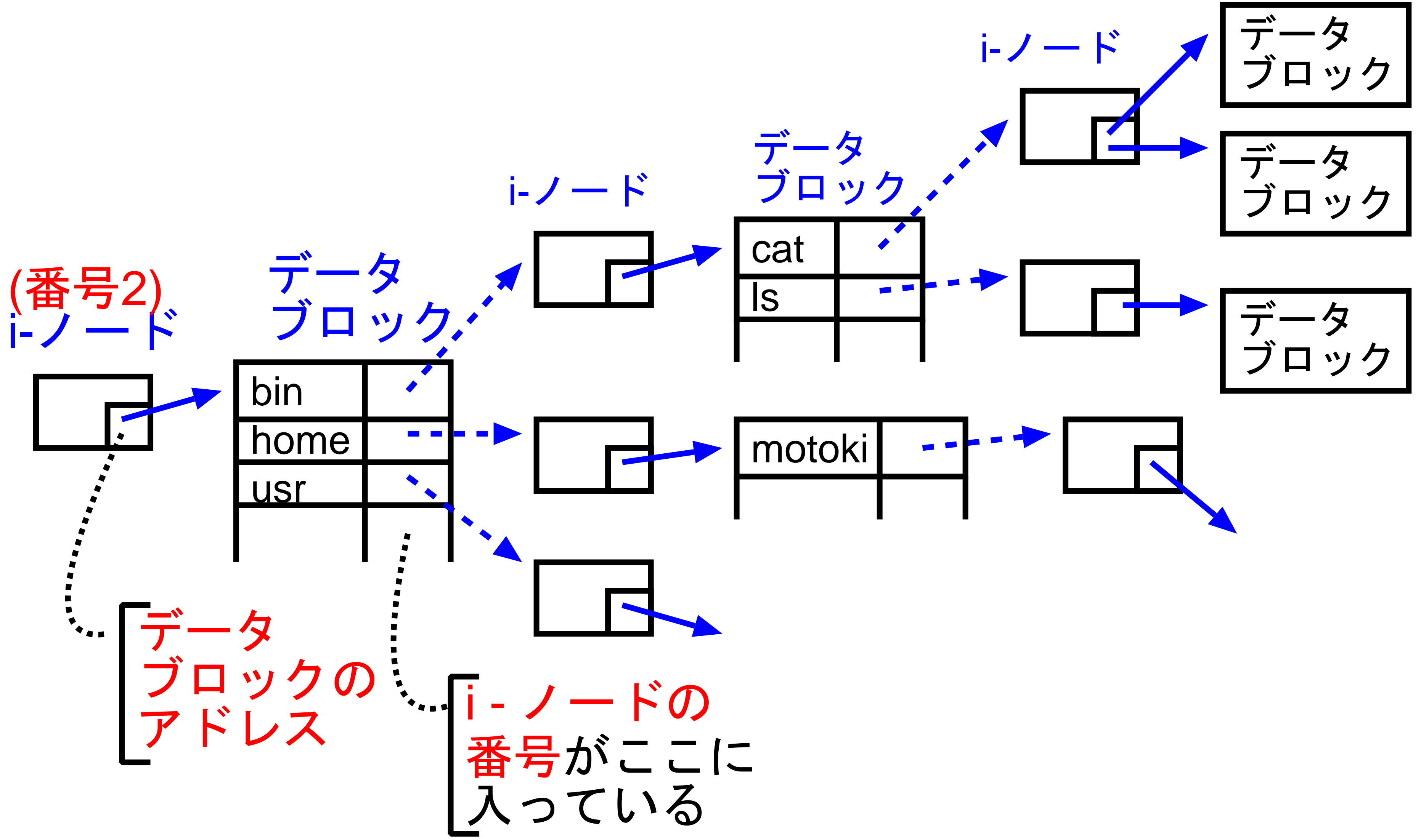
補足：

ディレクトリファイルに保存されているファイル名とi-ノード番号のペアを記憶されている順に列挙したければ、

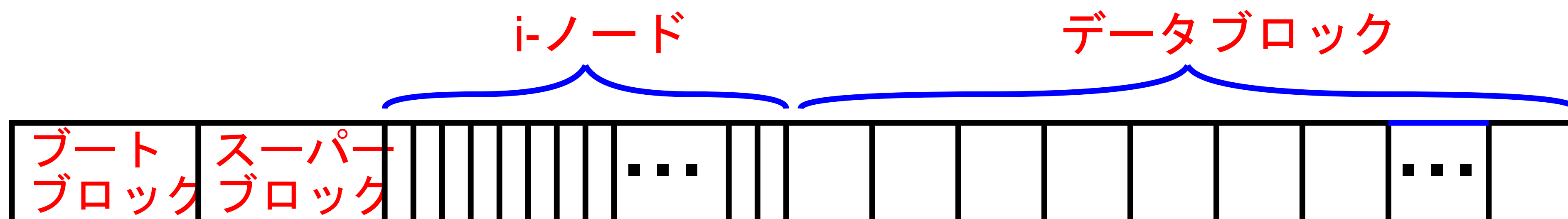
`ls -fi` ディレクトリ指定

とする。

- i-ノードとディレクトリファイルが交互に繋がって**木構造**が出来る。



- 伝統的な**UNIX システムのディスク**には次のようなブロックがこの順に並んでいる。



(ブートブロック) · · · ブートストラップのプログラムが入っている

(スーパーブロック) · · · ファイルシステムの基本パラメータが置かれている。例えば、ディスクの物理的なパラメータ (e.g. 回転速度, トラック数, シリンダ数, セクタ数), データブロックの大きさ, ...
。

(i-ノードの列) · · · 個々のファイル(やディレクトリ)を管理するための構造体であるi-ノードが並んでいる。i-ノードは、**1,2,3,...** と順に番号が付けられており、この内番号2のi-ノードがルートディレクトリのi-ノードと決められている。

(データブロックの列) · · · ファイルやディレクトリの中身が...

10–2 UNIXで扱うファイルの種類

UNIXにおいては、次の種類のファイルがある。

- 通常ファイル
- ディレクトリファイル
- シンボリックリンク
- 特殊ファイル
  - ブロック型特殊ファイル
  - 文字型特殊ファイル
- ソケット
- 名前付きパイプ
- ………

`ls`コマンドを`-l`オプション付きで実行した時、先頭(保護モードの左)に表示されるのがファイルの種類である。

先頭の文字	ファイルの種類
-	通常ファイル
d	ディレクトリファイル
l	シンボリックリンク
b	ブロック型特殊ファイル
c	文字型特殊ファイル
s	ソケット
p	名前付きパイプ

通常ファイル：

- テキストファイル,オブジェクトコード,画像ファイル, 音声ファイル, ... など、様々。
- ファイル内容に意味を与えるのはそのファイルを取り扱うプログラム。
- 順編成なので、データの挿入や一部削除はできない。(作り直しになる。)
- 作成者の所有物。

ディレクトリファイル：

- UNIX では、ディレクトリもファイルの一種。
- 作成者の所有物。

Emacs でディレクトリファイルを編集することもできます。(但し、テキストファイルの場合のように自由じゃありません。; DIRED モード,DIRectory EDit)

シンボリックリンク：

- ファイルに別名を付けるための特別なファイル。
- シンボリックリンクファイルの本体には、ファイルの実体を指す文字列が格納される。
- シンボリックリンクを張るには `ln` コマンドを使う。

`ln -s` [ファイルの実体を指す文字列] [リンク名]

特殊ファイル：

- 入出力装置やメモリ等を通常のファイルと同じように取り扱うために設けられている。
- デバイスファイルとも言う。
- 一般の UNIX システムでは特殊ファイルは /dev の下にある。
- ディスク装置のようにブロック単位で入出力を行う装置に対応したものを特にブロック型特殊ファイルと言う。
  また、それ以外の装置 (e.g. キーボード, ディスプレイ) に対応したものを文字型特殊ファイルと言う。
- 特殊ファイルはメジャー番号とマイナー番号という 2 種類の数値で識別される。

  メジャー番号は「SCSI ディスクなら 7 番」という風に装置の種類に対応しており、特殊ファイル取り扱い時にはこの番号に対応したデバイス制御プログラムが呼び出される。

  マイナー番号はデバイス制御プログラムに引数として渡される。

- Vine Linux 2.1.5の場合の特殊ファイルの例を次に列挙する。

  /dev/sd* ··· SCSIディスク　Small Computer System Interface

  /dev/hda ··· IDEプライマリチャネルのMasterディスク

  /dev/hdd ··· IDEセカンダリチャネルのSlaveディスク

  /dev/fd* ··· フロッピーディスク　↑ Integrated Device Electronics

  /dev/mem ··· メモリの物理空間

  /dev/kmem ··· カーネルの論理アドレス空間

  /dev/mouse ··· マウス

  /dev/tty* ··· 端末

  /dev/pty* ··· 擬似端末

  /dev/console ··· 管理用の端末

  /dev/null ··· ヌル。書き込むと全てを吸い込み、読み込むとEOFが返ってくる仮想的なデバイス。

補足：
コマンドの出力をどこにも出したくない時はリダイレクションで/dev/nullに書き出せば良い。また、空ファイルを作りたい時は /dev/null からコピーすれば良い。

- 特殊ファイルを作るために `mknod()` コマンド (**MaKe NODe**) が用意されている。(スーパーユーザしか使えない。) これに対応した`mknod()` というシステムコールもある。

ソケット：

- プロセス間通信の際に、通信の窓口として機能する様に設けられた特別なファイル。
- 代表的なソケットとして次のようなものがある。

```
[motoki@x205b]$ ls -l /dev/{log,printer}
srw-rw-rw-  1 root  root  0 Oct  4 20:48 /dev/log=
srw-------  1 root  root  0 Oct  4 20:48 /dev/printer=
[motoki@x205b]$ ls -l /tmp/.X11-unix/X0
srwxrwxrwx  1 root  root  0 Oct  4 20:48 /tmp/.X11-unix/X0=
[motoki@x205b]$
```

/dev/logは システム管理用のメッセージを集めるためのソケットで、書き込まれたメッセージはsyslogdというデーモンに送られる。

/dev/printerは プリンタへの出力用のためのソケットで、書き込まれたメッセージはlpdというプリンタデーモンに送られる。

/tmp/.X11-unix/X0は Xサーバと応用プログラムが通信するためのソケット。

名前付きパイプ：

- 普通のパイプでは、親子または兄弟関係にあるプロセス間でしか通信できない。 この制約をなくして、任意のプロセス(別ユーザのプロセス間も可)をパイプで繋げるために設けられた特別なファイル。

(⟹ **14.5節**)

- 名前付きパイプを作るには`mknod`コマンドを使う。(特殊ファイルを作る時と同じコマンド。)

10–3 ファイルのデータ構造

i-ノード：　　index-node

ファイルの中身はデータブロック(固定長)の中に入れられるが、場合によっては複数あるデータブロックを1つのファイルとして組み立てたり、ファイルに関する様々な属性・情報(名前以外)を保持したりするために、i-ノード(i-node)と呼ばれる構造体が用意されている。

具体的には、i-ノードには右のような情報が含まれている。

モード
リンク数
ユーザID
グループID
ファイルの大きさ
最終アクセス時刻
最終更新時刻
i-ノード最終更新時刻
直接ブロックの指定(8~ 16個)
1段間接ブロックの指定
2段間接ブロックの指定
3段間接ブロックの指定

- モード・・・ファイルの型、アクセスモード等の情報から成る。
  アクセスモードの変更のためには ... 。　また、ファイルを作成する際は、アクセスモードをパラメータとして指定する。
  昔のBSDだと次のような16ビットで構成されていた。

15　　12	11	10	9	8　　6	5　　3	2　　0
型	S	S	t	r w x (user)	r w x (group)	r w x (others)

ここで、各々のビットの意味は次の通り。

(15~ 12ビット目)
ファイルの型を表す。

lsコマンドの表示	15~ 12ビット目	ファイルの型
-	1000	通常ファイル
d	0100	ディレクトリファイル
l	1010	シンボリックリンク
b	0110	ブロック型特殊ファイル
c	0010	文字型特殊ファイル
s	1100	ソケット
p	0001	名前付きパイプ

(11ビット目) set-user-IDビットと呼ばれる。このビットが立っていると、ファイルとして格納されているプログラムが実行される時、(起動した人じゃなく) このファイルの所有者のユーザIDがプロセスの実効ユーザIDとして設定される。`ls -l`時は、このビットが立っていれば所有者の実行モードは `x` が`s` に, `-` が `S` に変わる。このビットを設定するには`chmod`コマンドを使う。

(10ビット目) set-group-IDビットと呼ばれる。このビットが立っていると、ファイルとして格納されているプログラムが実行される時、このファイルのグループIDがプロセスの実効グループIDとして設定される。

(9ビット目) stickyビットと呼ばれる。例えば、ディレクトリファイルの場合は、このビットが立っていると、その中のファイルを消したり名前を変えたりするのがファイルの所有者に限定されるようになる。

$\Longrightarrow$ 山口 (監)「The UNIX Super Text 上」(1992)9.2.1節

—

(8~ 6ビット目) ファイルの所有者自身に対するアクセスモード。

(5~ 3ビット目) ファイルの所有者と同じグループ内のユーザに対するアクセスモード。

(2~ 0ビット目) 他人に対するアクセスモード。

- ハードリンク数 · · · UNIXにおいては、ファイルの名前はそのファイルにアクセスするディレクトリファイルが持っている。
それゆえ、UNIXにおいては複数箇所のディレクトリから1つのファイルに別のファイル名でアクセスするということも可能である。

⇨ ファイルの必要性を見るために、
何か所から参照されているかをこのフィールドに保持する。
これが0になったら、このファイルのために使われているデータブロックは全て解放される。

既に出来ているファイルやディレクトリに別名を付けるために`ln`コマンドが用意されている。

補足：

lsコマンドを-lオプション付きで実行した時、保護モードの次に表示される数字が共用リンク数である。例えば、

```
[motoki@x205b]$ ls -al
合計 1896
drwxrwxr-x   4 motoki  motoki    4096 Oct  5 10:52 ./
drwx------  22 motoki  motoki    4096 Oct  5 08:46 ../
drwxrwxr-x   2 motoki  motoki    4096 Oct  4 20:49 C-Programs/
drwxrwxr-x   2 motoki  motoki    4096 Oct  4 20:49 Figs/
-rw-rw-r--   1 motoki  motoki   36320 Sep 30 00:43 OS-OHP.aux
    .........
[motoki@x205b]$
```

lnコマンドを使って(ハード) リンクの追加を行っていなければ

通常ファイルのリンク数＝1,

ディレクトリのリンク数＝(子ディレクトリの数)+2

となる。 各々のディレクトリには .. という親ディレクトリを意味する名前も . という自ディレクトリを意味する名前も登録され、親からだけでなく子ディレクトリや自ディレクトリからの(ハード) リンクも自動的に作られる。

- **ユーザID, グループID** · · ·
- **ファイルの大きさ** · · · ファイル実体を構成するバイト数。
- **最終アクセス時刻,最終更新時刻, i-ノード最終更新時刻** · · ·
  1970 年 1 月 1 日 0 時 0 分 (グリニッジ標準時) からの経過時間。単位は秒。

- ディスクブロックアドレス · · ·

  「ディスクブロック」とはディスク上の連続領域に並んだ物理的な入出力の単位のことで、磁気ディスクの場合はセクタを意味する。

  具体的には、ディスクブロックはブロック番号で指定する。i-ノードの中にはブロック番号を格納するために**n=8~ 16**個の領域が用意され、そのうち最初の**n−3**個の領域にはファイルを構成する最初の**n−3**個のデータブロック(直接ブロックと言う) の番号が格納される。

  ファイルが大きく**n−3**個のブロックに入り切らない場合は、最初の**n−3**個に続くデータブロックの番号の列を格納したデータブロック(**1**段間接ブロックと言う)を用意し、そのブロックの番号をi-ノードの**n−2**番目のブロック番号領域に格納される。

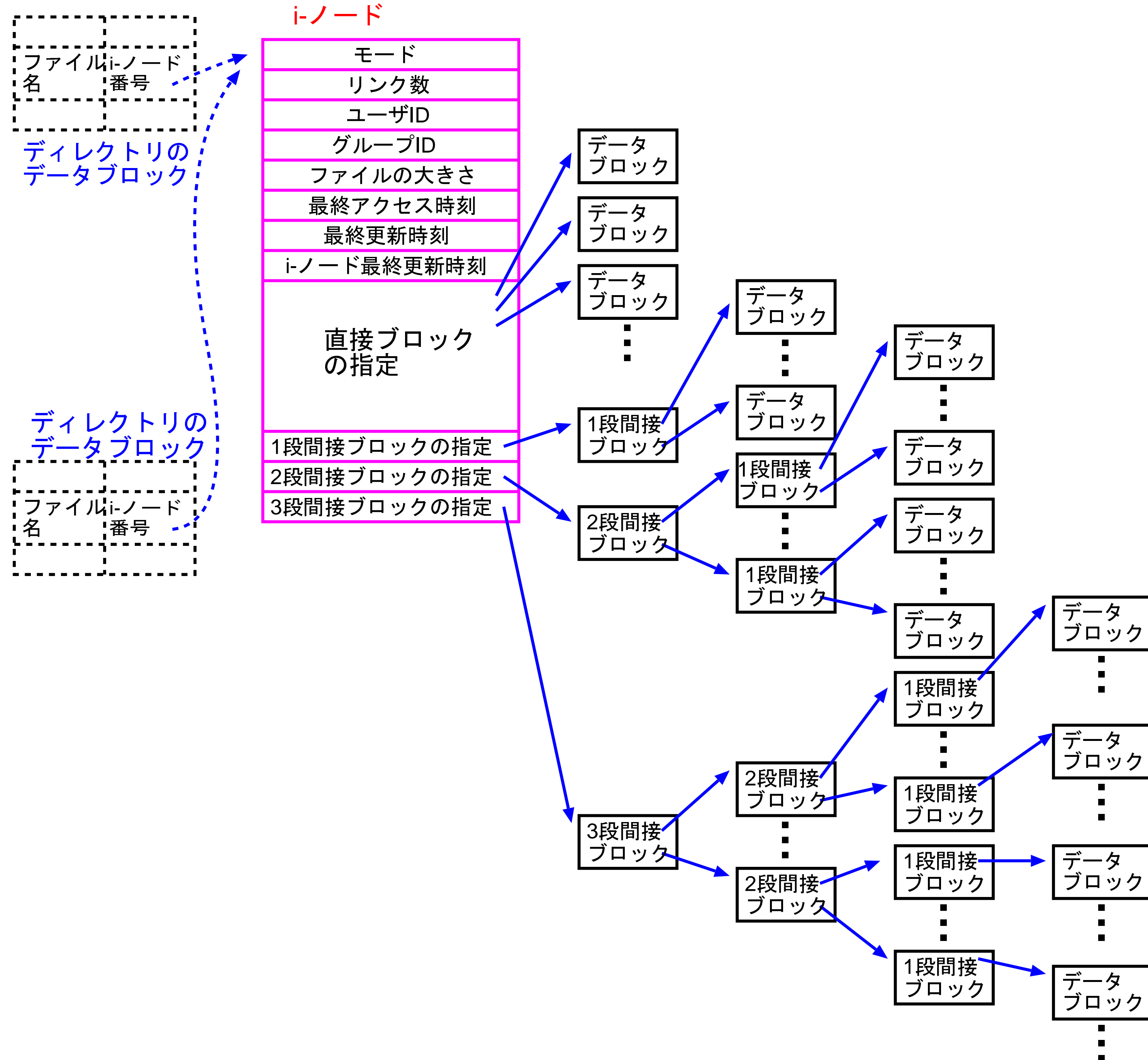
  それでも足りなければ、更に「**2**段間接ブロック」,「**3**段間接ブロック」が構成され、それらの番号がi-ノードの **n−1**番目, **n**番目 のブロック番号領域に格納される。

Vine Linux2.1.5だと、**i-ノードの構造体**は/usr/include/linux/fs.hの中で次の様に定義されている。

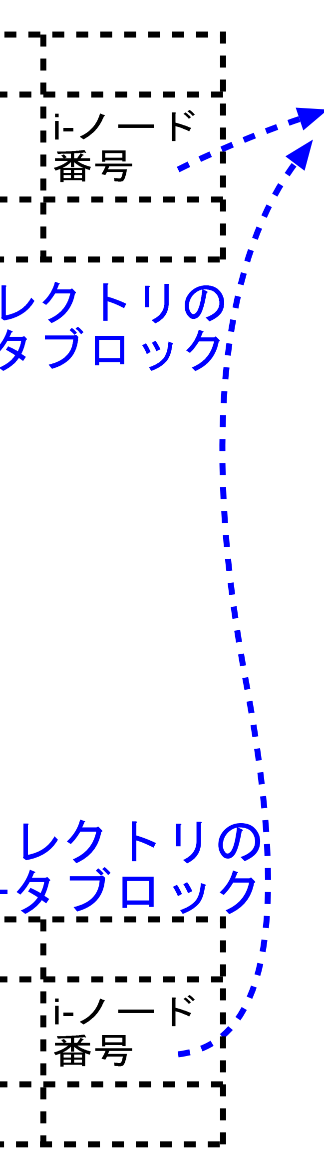
```
struct inode {
  struct list_head  i_hash;
  struct list_head  i_list;
  struct list_head  i_dentry;
  unsigned long     i_ino;
  unsigned int      i_count;
  kdev_t            i_dev;
  umode_t           i_mode;
  nlink_t           i_nlink;
  uid_t             i_uid;
  gid_t             i_gid;
  .........
  union {
    struct pipe_inode_info     pipe_i;
    struct minix_inode_info    minix_i;
    struct ext2_inode_info     ext2_i;
  .........
  } u;
};
```

初期のUNIXと比べるとかなりの拡張(e.g.他の種類のファイルシステムも扱えるようにした)が為されていて、複雑になっている。

通常ファイルの構造：



ディレクトリファイルの構造：そのディレクトリ内のファイル(やディレクトリ)の 名前 と **i-**ノード番号 のペアを保持する。



i-ノード

モード
リンク数
ユーザID
グループID
ファイルの大きさ
最終アクセス時刻
最終更新時刻
i-ノード最終更新時刻
直接ブロックの指定
1段間接ブロックの指定
2段間接ブロックの指定
3段間接ブロックの指定

データブロック
データブロック
データブロック
データブロック
データブロック
1段間接ブロック
2段間接ブロック
3段間接ブロック

次の様な表を保存。(BSD系の場合)

i-ノード番号	文字数	ファイル名
		.
		..
・・・	・・・	・・・

10–4 指定ファイルへのアクセス

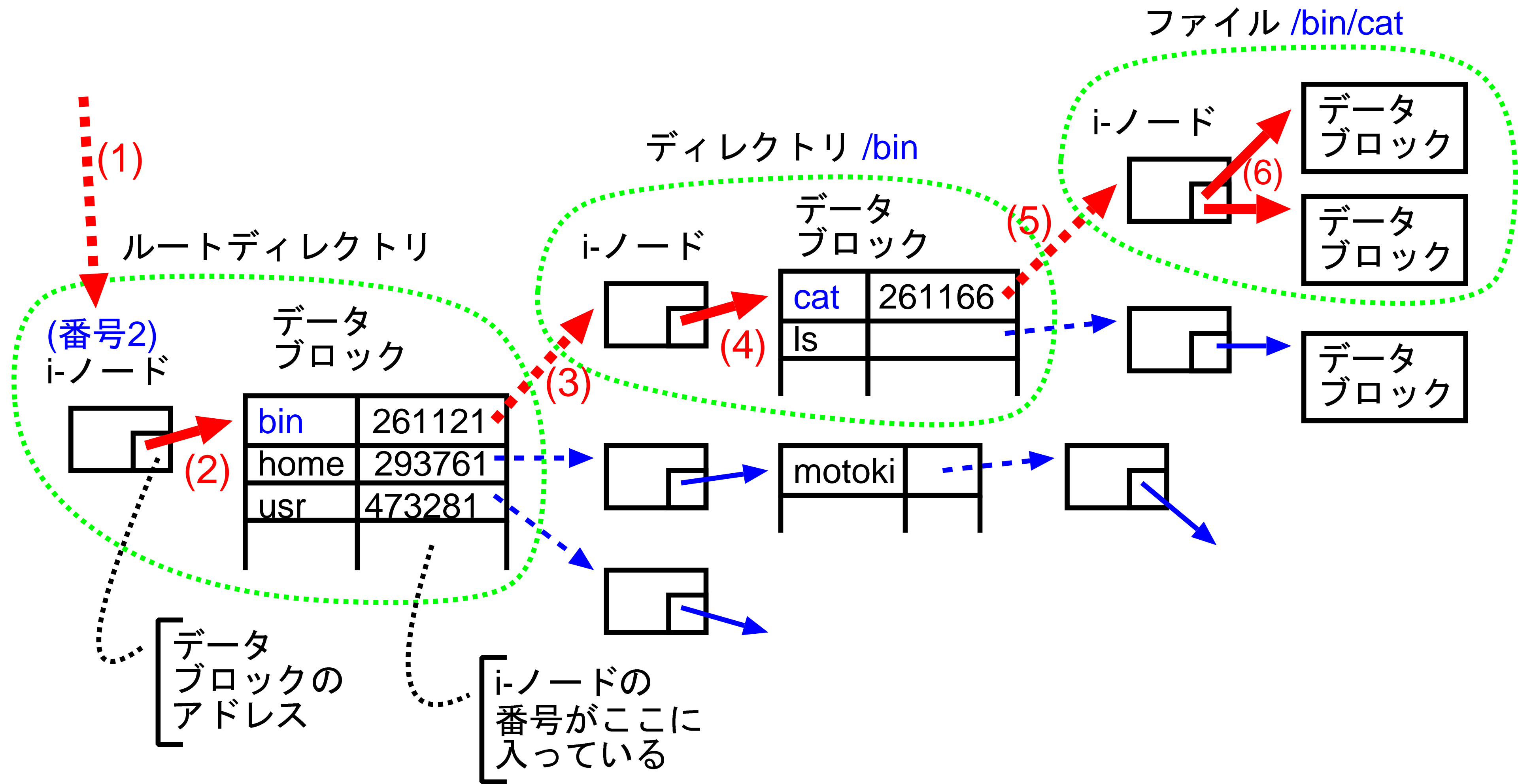
例10. 1 (/bin/catへのアクセス)　2つのディレクトリ / と /bin とファイル /bin/cat の i-ノード番号は次の様にして確認することが出来る。

```
[motoki@x205b]$  ls -ia /
      2 ./      212161 etc/          342721 mnt/    179521 tmp/
      2 ../     293761 home/         391681 opt/    473281 usr/
 261121 bin/    310081 lib/               1 proc/    97921 var/
      2 boot/       11 lost+found/   424321 root/
 195841 dev/         1 misc/         456961 sbin/
[motoki@x205b]$  ls -i /bin/cat
 261166 /bin/cat*
[motoki@x205b]$
```

補足：
2番のi-ノードが / 以外にも表示されている。これは、boot が別パーティションのルートディレクトリになっているためである。

___

⇨ /bin/cat へのアクセスは次の様に行われる。



補足：

statコマンドを用いれば各々のファイルやディレクトリの属性情報を表示することが出来る。例えば次の通り。

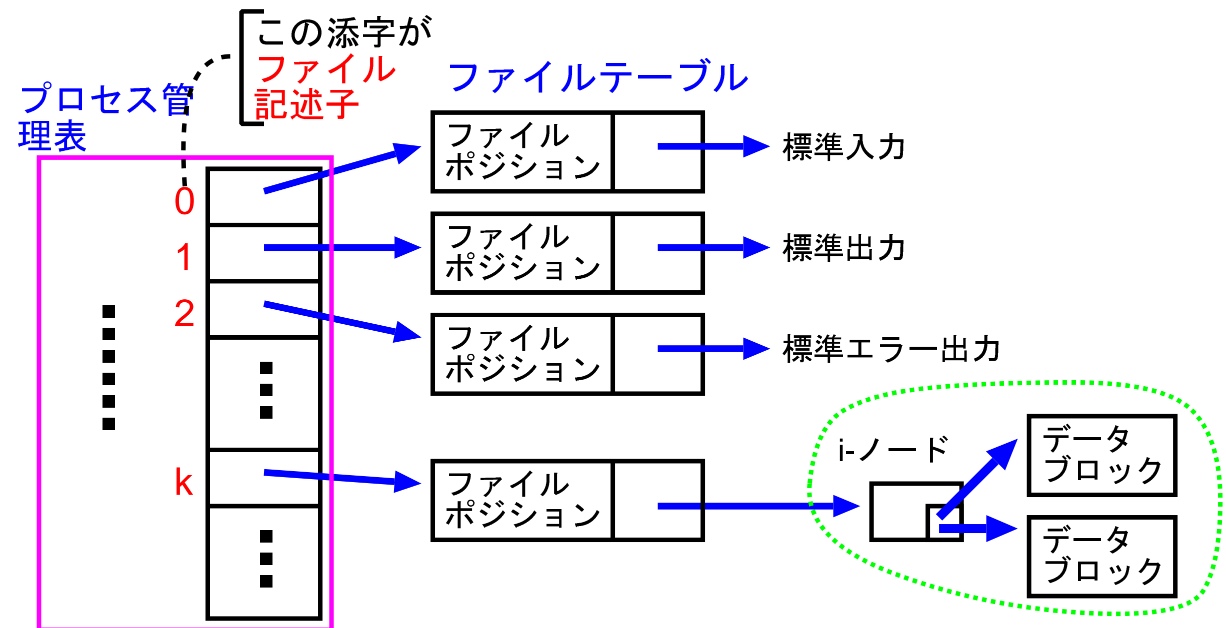
```
[motoki@x205b]$  stat /
  File: "/"
  Size: 4096         Filetype: Directory
  Mode: (0755/drwxr-xr-x) Uid: ( 0/ root) Gid: ( 0/ root)
Device:  3,5   Inode: 2          Links: 18
Access: Sat Oct  5 19:26:22 2002(00000.21:41:49)
Modify: Tue Jan 22 20:39:44 2002(00256.20:28:27)
Change: Tue Jan 22 20:39:44 2002(00256.20:28:27)
[motoki@x205b]$
```

10–5 ファイル入出力のシステムコール

ファイル記述子：　カーネルは、プロセスが開いたファイルに対して
①次に読み書きする位置を示すファイルポジションと
②ファイルの実体を束ねるi-ノードへのポインタ
から成る、ファイルテーブルと呼ばれる構造体を構成する。

プロセスが関与するファイルテーブルへのポインタの配列をプロセス毎に用意して管理する。

これらの配列の添字はファイル記述子と呼ばれ、ファイル入出力に利用することが出来る。



- ファイル記述子を使う場合は`<fcntl.h>` を宣言する。

- プロセス起動時には、各プロセスに対して標準入力, 標準出力, 標準エラー出力の3つのファイルがオープンして与えられる。

ファイル	ファイル記述子	<unistd.h>で定義されたマクロ
標準入力	0	STDIN_FILENO
標準出力	1	STDOUT_FILENO
標準エラー出力	2	STDERR_FILENO

- 初期のUNIX(Version7)では、各プロセスは0~ 19のファイル記述子が使用できた。

ファイル・オープンのシステムコール

`int open(char *path,int flags [,mode_t mode]) :`

- ヘッダファイル `<unistd.h>` と `<fcntl.h>` を必要とする。
- オープンしたファイルのファイル記述子を関数値として返す。失敗すると -1 を返す。
- 指定したファイルが無く第3引数がある場合は、新たに生成される。
- 関数引数の `path` は、オープンするファイルをフルパスで指定した文字列を表す。
- 関数引数の `flags` はファイルの使い方を表したもので、次のマクロを OR 演算子 (|) で繋げて指定する。
  - `O_RDONLY` · · · 読み出し専用にオープンする。
  - `O_WRONLY` · · · 書き込み専用にオープンする。
  - `O_RDWR` · · · 読み書き両用にオープンする。
  - `O_APPEND` · · · 追加書き込み用にオープンする。
  - `O_CREAT` · · · ファイルが存在しない場合に作成
  - ....
- 関数引数の `mode` は、ファイルの保護モードを表す。 ...

ファイル生成のシステムコール

`int creat(char *path, mode_t mode) :`

- ヘッダファイル `<unistd.h>` と`<fcntl.h>` を必要とする。
- 次の `open()` 関数実行と同等である。

```
open(path, O_CREAT|O_TRUNC|O_WRONLY, mode)
```

ファイルクローズのシステムコール `int close(int fd) :`

- ヘッダファイル `<unistd.h>` を必要とする。
- 成功すると **0** を返し、失敗すると **−1** を返す。
- 関数引数 `fd` の箇所にはファイル記述子を指定する。

ファイル読み込みのシステムコール

`int read(int fd, char *buf, int nbytes) :`

- ヘッダファイル <unistd.h> を必要とする。
- 現在操作中の位置(この位置を指しているポインタをファイルポインタと呼ぶ)からデータを読み込む。但し、データが無くなれば、そこで読み込みはおしまいである。
  読み込みに成功すれば読み込んだデータのバイト数が関数値として返され、ファイルポインタは読み込んだバイト数だけ移動する。 また、失敗すれば -1 が返される。
- 関数引数の fd は読み込み元のファイル記述子である。

注意：
標準ライブラリ関数 fopen() から返される値もやはり「ファイルポインタ」と呼んでいたが別物である。

- 関数引数の buf は読み込んだデータを保存するバッファのアドレスを表す。
- 関数引数の nbytes は読み込むデータのバイト数を表す。

ファイル書き込みのシステムコール

`int write(int fd, char *buf, int nbytes) :`

- ヘッダファイル <unistd.h> を必要とする。
- 成功すると書き出されたバイト数が返され、ファイルポインタは書き出されたバイト数だけ移動する。失敗すると -1 が返される。
- 関数引数の fd は書き出し先のファイル記述子
- 関数引数の buf は書き出すデータを保存するバッファのアドレスを表す。
- 関数引数の nbytes は書き出すデータのバイト数を表す。

read(), write() のためにシステムバッファが用意されている場合は、read(),write() はシステムバッファ上の仮想ファイルに対してだけ行われ磁気ディスクへの書き込みはしばらく後になる。

⇨ ファイルデータ上の矛盾発生を避ける等のために即座に磁気ディスクに書き込みたい場合は sync() システムコールを使う。

システムバッファが用意されない場合は、read(), write() の度に磁気ディスクにアクセスすることになるので、プログラムの実行速度がバッファサイズ nbytes によって大きく左右されることになる。

例 10. 2 (大文字↔小文字の反転)　コマンドラインの1番目の引数で指定したファイル内容の大文字と小文字を反転して、その結果を2番目の引数で指定したファイルに書き出すプログラムをファイル記述子を使って構成してみた。次に示す通りである。

```
[motoki@x205a]$  nl io-through-file-descriptor.c
 1 /*****************************************************
 2 /* Operating-Systems/C-Programs/io-through-file-descriptor.
 3 /*-----------------------------------------------------
 4 /* ファイル記述子を使ったプログラム例                          *
 5 /* A.ケリー&I.ポール「CのABC(下)」アジソンウェスレイ              */
 6 /*                         ジャパン/星雲社,1993,11.8節 */
 7 /*****************************************************
 8 #include <ctype.h>
 9 #include <fcntl.h>
10 #include <unistd.h>

11 #define BUFSIZE  1024
```

```
12 int main(int argc, char **argv)
13 {
14   char buffer[BUFSIZE];
15   int  in_fd, out_fd, in_size, k;

16   in_fd = open(argv[1], O_RDONLY);
17   out_fd = open(argv[2], O_WRONLY|O_EXCL|O_CREAT, 0644);
18   while ((in_size=read(in_fd, buffer, BUFSIZE)) > 0) {
19     for (k=0; k<in_size; k++) {
20       if (islower(buffer[k]))
21          buffer[k]=toupper(buffer[k]);
22       else
23          buffer[k]=tolower(buffer[k]);
24     }
25     write(out_fd, buffer, in_size);
26   }
```

```
27    close(in_fd);
28    close(out_fd);
29    return 0;
30 }
[motoki@x205a]$  gcc io-through-file-descriptor.c
[motoki@x205a]$  ./a.out io-through-file-descriptor.c out
[motoki@x205a]$  ls -l {io-through-file-descriptor.c,out}
-rw-rw-r--  1 motoki motoki 1044  9月 2日 11:00 io-through-fil
-rw-r--r--  1 motoki motoki 1044  9月 2日 13:40 out
[motoki@x205a]$  cat out
/**********************************************
/* oPERATING-sYSTEMS/c-pROGRAMS/IO-THROUGH-FILE-DESCRIPTOR.C *
/*--------------------------------------------*
/* ファイル記述子を使ったプログラム例                    */
/* a.ケリー&i.ポール「cのabc(下)」アジソ*/
/*             ジャパン/星雲社,1993,11.8節 */
/**********************************************
```

```
#INCLUDE <CTYPE.H>
#INCLUDE <FCNTL.H>
#INCLUDE <UNISTD.H>

#DEFINE bufsize  1024

MAIN(INT ARGC, CHAR **ARGV)
{
  CHAR BUFFER[bufsize];
  INT  IN_FD, OUT_FD, IN_SIZE, K;

     ......(以下省略)......
}
[motoki@x205a]$
```

ファイルポインタ設定のシステムコール

`long lseek(int fd, long offset, int whence) :`

- ヘッダファイル `<unistd.h>` を必要とする。
- 成功すると更新後のファイルポインタ値が返され、失敗すると -1 が返される。
- 関数引数の `fd` はファイル記述子である。
- 関数引数の `whence` はファイルポインタ設定の基点となるもので、次の3つのいずれかを指定する。
  - SEEK_SET(値=0) $\cdots$ ファイルの先頭(先頭要素の位置)
  - SEEK_CUR(値=1) $\cdots$ 現在のファイルポインタ
  - SEEK_END(値=2) $\cdots$ ファイルの末尾(最後の要素の次の位置)
- 関数引数の `offset` は第3引数 whence で指定された基点からの変位を表す。正なら先頭から末尾方向へのバイト数、負なら逆方向へ移動するバイト数を表す。

10–6 ファイル属性獲得と変更のシステムコール

ファイルのi-ノード情報を得るための3つのシステムコール：

- 各々の関数のプロトタイプは次の通り。

```
int stat(char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(char *path, struct stat *buf);
```

- ヘッダファイル <unistd.h> と<sys/stat.h> を必要とする。
- 第1引数で指定されたファイルの情報を第2引数で指定された領域に設定する。これが成功すると 0 が返され、失敗すると −1 が返される。
- stat() と fstat()においては指定されたファイルを確定するためにシンボリックリンクがたどられファイルの実体についての情報が得られる。
  一方、lstat()においてはシンボリックリンクはたどらない。
- 関数引数の path はファイルをフルパスで指定した文字列を表す。
- 関数引数の fd はファイル記述子を表す。

- 第2引数の構造体 struct stat はヘッダファイル <sys/stat.h> の中で次の様に定義されている。

```
struct stat {
  dev_t   st_dev; /*デバイス(i-ノードを格納しているデバイス*/
                  /*          のメジャー番号とマイナー番号)*/
  ino_t   st_no;      /* i-ノード番号    */
  mode_t  st_mode;    /* ファイルモード  */
  nlink_t st_nlink;   /* ハードリンク数  */
  uid_t   st_uid;     /* ユーザID   */
  gid_t   st_gid;     /* グループID */
  dev_t   st_rdev;    /* 特殊ファイルである場合のデバイス型 */
  off_t   st_size;    /* ファイルの大きさ  */
  unsigned long st_blksize;  /* Optimal block size for I/O.
  unsigned long st_blocks;   /* Number 512-byte blocks alloc
  time_t  st_atime; /*最終アクセス時刻    */
  time_t  st_mtime; /*最終更新時刻        */
  time_t  st_ctime; /*i-ノード最終更新時刻*/
};
```

補足： Vine Linux 2.1.5 だと<bits/stat.h>の中で定義し、それを<sys/stat.h>の中で include している。定義自体ももう少し複雑で、上記以外の構造体要素も幾つか含まれている。 構造体要素の型も一部のものは #ifndef 等を用いて場合分けされている。

例10. 3 (i-ノード上の主要ファイル属性を表示) i-ノードに格納されている主要情報を表示するプログラムを次に示す。

[motoki@x205a]$ nl print-status-of-file.c

```
 1 /*****************************************************
 2 /* Operating-Systems/C-Programs/print-status-of-file.c
 3 /*-----------------------------------------------------
 4 /* i-ノードに格納されている主要情報を使用するプログラム例      */
 5 /*C.オール「例題で学ぶLinuxプログラミング」ピアソン,7.2.5節 */
 6 /*****************************************************
 7 #include <stdio.h>
 8 #include <stdlib.h>
 9 #include <time.h>
10 #include <string.h>
11 #include <unistd.h>
12 #include <sys/stat.h>

13 int main(int argc, char **argv)
```

```
14 {
15   struct stat buf;
16   mode_t mode;
17   char   filetype[80];

18   if (argc != 2) {
19     printf("Give a file_name as a command parameter.\n");
20     exit(EXIT_FAILURE);
21   }else if (lstat(argv[1], &buf) < 0) {
22     perror("lstat");
23     exit(EXIT_FAILURE);
24   }

25   printf("  File Name: %s\n", argv[1]);
26   printf("     Device: %d,%d\n",
27          major(buf.st_dev), minor(buf.st_dev));
28   printf("  Inode No.: %d\n", buf.st_ino);
```

```
29    mode = buf.st_mode;
30    if (S_ISREG(mode))
31      strcpy(filetype, "Regular file");
32    else if (S_ISDIR(mode))
33      strcpy(filetype, "Directory");
34    else if (S_ISLNK(mode))
35      strcpy(filetype, "Symbolic Link");
36    else if (S_ISBLK(mode))
37      strcpy(filetype, "Block Device");
38    else if (S_ISCHR(mode))
39      strcpy(filetype, "Character Device");
40    else if (S_ISSOCK(mode))
41      strcpy(filetype, "Socket");
42    else if (S_ISFIFO(mode))
43      strcpy(filetype,"FIFO (named pipe)");
44    else
```

```
45      strcpy(filetype, "Unknown Type)");
46    printf("  File Type: %s\n", filetype);
47                                    ファイルの型を表すビット部分のマスク
48    printf("Access Mode: %#.3o\n", buf.st_mode & ~(S_IFMT));
49    printf(" Link Count: %d\n", buf.st_nlink);
50    printf("    User ID: %d\n", buf.st_uid);
51    printf("   Group ID: %d\n", buf.st_gid);
52    printf("  File Size: %d\n", buf.st_size);
53    printf("      Optimal block size for I/O: %d\n",
             buf.st_blksize);
54    printf("Number 512-byte blocks allocated: %d\n",
             buf.st_blocks);
55    printf("Time of last access        : %s",
             ctime(&buf.st_atime)); カレンダー時間→ローカル時刻を
56    printf("Time of last modification : %s",
             ctime(&buf.st_mtime));
57    printf("Time of last status change: %s",
```

```
              ctime(&buf.st_ctime));

58    return 0;
59 }
[motoki@x205a]$  gcc print-status-of-file.c
[motoki@x205a]$  ./a.out /
  File Name: /
     Device: 8,6
  Inode No.: 2
  File Type: Directory
Access Mode: 0755
 Link Count: 25
    User ID: 0
   Group ID: 0
  File Size: 4096
      Optimal block size for I/O: 4096
Number 512-byte blocks allocated: 8
```

```
Time of last access        : Mon Aug 29 12:49:13 2011
Time of last modification : Fri Sep  2 08:40:54 2011
Time of last status change: Fri Sep  2 08:40:54 2011
[motoki@x205a]$
```

補足：

Vine Linux 2.1.5 では S_IFMT や S_ISREG(), S_ISDIR(), ... といったマクロは <linux/stat.h>の中で次の様に定義されている。

```
#define S_IFMT  00170000
#define S_IFSOCK 0140000
#define S_IFLNK  0120000
#define S_IFREG  0100000
#define S_IFBLK  0060000
#define S_IFDIR  0040000
#define S_IFCHR  0020000
#define S_IFIFO  0010000
#define S_ISUID  0004000
#define S_ISGID  0002000
#define S_ISVTX  0001000

#define S_ISLNK(m)  (((m) & S_IFMT) == S_IFLNK)
#define S_ISREG(m)  (((m) & S_IFMT) == S_IFREG)
#define S_ISDIR(m)  (((m) & S_IFMT) == S_IFDIR)
#define S_ISCHR(m)  (((m) & S_IFMT) == S_IFCHR)
#define S_ISBLK(m)  (((m) & S_IFMT) == S_IFBLK)
#define S_ISFIFO(m) (((m) & S_IFMT) == S_IFIFO)
#define S_ISSOCK(m) (((m) & S_IFMT) == S_IFSOCK)
```

ファイルの保護モード変更のシステムコール chmod( ) と fchmod( ) :

- chmodコマンドに対応。
- 各々の関数のプロトタイプは<unistd.h> で次の様に定義されている。

```
int chmod(char *path, mode_t mode);
int fchmod(int fd, mode_t mode);
```

- ヘッダファイル <sys/types.h> と<sys/stat.h> を必要とする。
- スーパーユーザとファイルの所有者だけが実際に保護モードを変更できる。
- 第1引数で指定されたファイルの保護モードが第2引数の指定の様に変更される。これが成功すると 0 が返され、失敗すると −1 が返される。
- 関数引数の path はファイルをフルパスで指定した文字列を表す。
- 関数引数の fd はファイル記述子を表す。
- 関数引数の mode は、ファイルの保護モードを表す。(mode_t の実体は int である。)

ファイルの所有者変更のシステムコールchown() と fchown():

- chownコマンドに対応。
- 各々の関数のプロトタイプは<unistd.h> で次の様に定義されている。

```
int chown(char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
```

- ヘッダファイル <sys/types.h> と<unistd.h> を必要とする。
- 使用者は通常はスーパーユーザに限定される。
- 第1引数で指定されたファイルのユーザIDとグループIDが第2~ 3引数の指定の様に変更される。
  これが成功すると 0 が返され、失敗すると −1 が返される。
- 関数引数の path はファイルをフルパスで指定した文字列を表す。
- 関数引数の fd はファイル記述子を表す。

- 関数引数の owner と group は、変更後の新しいユーザID とグループID でシステムが理解できる値でなければならない。

補足：

idコマンドを使うとユーザID, グループID を知ることができます。
例えば、自分のID なら

```
[motoki@x205a]$ id
uid=500(motoki) gid=501(motoki) 所属グループ=501(motoki)
[motoki@x205a]$
```

ユーザのid番号⟹ユーザ名　の変換は、
getpwuid( ) ライブラリ関数等を用いて行なうことが出来る。

(⟹11.1 節)

10–7 ディレクトリ操作のライブラリ関数

ディレクトリファイルの場合も 、通常のファイル処理手順と同じ様に、

① 最初にディレクトリファイルをオープンし、
② ディレクトリから1つずつ要素(ディレクトリ内のファイル名とその i-ノード番号のペア)を取り出して関連する処理を行い、
③ 最後にディレクトリファイルをクローズする、

という手順で処理するための(UNIX標準)ライブラリ関数が用意されている。

ディレクトリから1つずつ要素が取り出されてできるストリーム(ディレクトリストリームと呼ぶ)が仮想的に構成され、そのストリームを介してディレクトリ要素を受け取ることになる。

ディレクトリ・オープンのライブラリ関数

DIR *opendir(const char *path) :

- ヘッダファイル <dirent.h> を必要とする。

- opendir( ) が呼ばれると、引数で指定されたディレクトリファイルをオープンし、ディレクトリストリームへのポインタを関数値として返す。これに失敗した場合は**NULL**を返す。

  この関数値は一般ファイルの場合の「ファイル記述子」に相当するもので、以後ディレクトリファイルからデータを取りこむ際のキーとなる。

- DIR は <dirent.h>の中で定義された構造体である。

- 関数引数の path は、オープンするディレクトリをフルパスで指定した文字列を表す。

ディレクトリ・クローズのライブラリ関数int closedir(DIR *dir) :

- ヘッダファイル <dirent.h> を必要とする。

- closedir( ) が呼ばれると、引数で指定されたディレクトリストリームの領域は解放されディレクトリがクローズされる。成功すると $\mathbf{0}$ を返し、失敗すると $-\mathbf{1}$ を返す。

- 関数引数 dir の箇所にはディレクトリストリームへのポインタを指定する。

ディレクトリストリームから次の要素を探し出す

ライブラリ関数 readdir() :

- 関数プロトタイプは次の通り。

```
struct dirent *readdir(DIR *dir);
```

- ヘッダファイル <dirent.h> を必要とする。
- readdir() が呼ばれると、引数で指定されたディレクトリストリームから次の要素を探し出し、そこへのポインタを関数値として返す。そして、「次の読み出し位置」を保持する変数を更新する。要素が無くなった場合、および失敗した場合は**NULL**を返す。
- 関数引数 dir の箇所にはディレクトリストリームへのポインタを指定する。

- 関数値のデータ型であるstruct dirent は次の様に定義された構造体である。

```
struct dirent {
  long     d_ino;                  /*ファイルのi-ノード番号*/
  off_t    d_off;                  /*次の要素へのオフセット*/
  unsigned short d_reclen;         /*このレコードの長さ*/
  char     d_name[NAME_MAX+1];     /*ファイルの名前*/
};
```

Vine Linux 2.1.5においては、 実際には構造体 struct dirent は <bits/dirent.h>の中で次の様に定義されている。

```
struct dirent
  {
#ifndef __USE_FILE_OFFSET64
    __ino_t d_ino;
    __off_t d_off;
#else
    __ino64_t d_ino;
    __off64_t d_off;
#endif
    unsigned short int d_reclen;
    unsigned char d_type;
    char d_name[256];  /*We must not include limits.h!*/
  };
```

例10. 4 (ディレクトリ内のファイルのリストを表示) ディレクトリファイル内に蓄えられているファイル名を列挙するプログラムを次に示す。

```
[motoki@x205a]$  nl list-directory-contents.c
     1  /*********************************************************
     2  /* Operating-Systems/C-Programs/list-directory-contents.c
     3  /*-------------------------------------------------------
     4  /* ディレクトリ内のファイルのリストを表示するプログラム      */
     5  /* C.オール「例題で学ぶLinuxプログラミング」ピアソン,8.3.3節 */
     6  /*********************************************************
     7  #include <stdio.h>
     8  #include <stdlib.h>
     9  #include <dirent.h>

    10  int main(int argc, char *argv[])
    11  {
    12    DIR *dir;
    13    struct dirent *ptr_to_dir_entry;
```

```
14    int num;

15    if (argc != 2) {
16      printf("Give a directory_name "
                 "as a command parameter.\n");
17      exit(EXIT_FAILURE);
18    }else if ((dir=opendir(argv[1])) == NULL) {
19      perror("opendir");
20      exit(EXIT_FAILURE);
21    }

22    num=0;
23    while ((ptr_to_dir_entry=readdir(dir)) != NULL)
24      printf("%3d : %s\n", ++num, ptr_to_dir_entry->d_name);

25    printf("Given directory contains %d files.\n", num);
26
```

```
27    closedir(dir);
28    return 0;
29 }
[motoki@x205a]$  gcc list-directory-contents.c
[motoki@x205a]$  ./a.out /
  1 : net
  2 : opt
  3 : dev
  4 : srv
  5 : media
  6 : lib64
  7 : lost+found
  8 : misc
  9 : ..
 10 : tmp
 11 : proc
 12 : usr
```

```
 13 : home
 14 : etc
 15 : .
 16 : boot
 17 : bin
 18 : sbin
 19 : .autofsck
 20 : var
 21 : sys
 22 : initrd
 23 : root
 24 : selinux
 25 : mnt
 26 : lib
Given directory contains 26 files.
[motoki@x205a]$
```

例10. 5 (ディレクトリ内のファイルのリストを表示2) ディレクトリ内の各ファイルの各種属性を`lstat()`システムコールで取り込んで種類、保護モード、サイズも一緒に表示したプログラムを次に示す。

```
[motoki@x205a]$ nl list-directory-contents2.c
 1 /*****************************************************
 2 /* Operating-Systems/C-Programs/list-directory-contents2.c
 3 /*----------------------------------------------------
 4 /* ディレクトリ内のファイルのリストを表示するプログラム */
 5 /*C.オール「例題で学ぶLinuxプログラミング」ピアソン,8.3.3節 */
 6 /* D.A.Curry「UNIX Cプログラミング」アスキー出版局,4.4節 */
 7 /*****************************************************
 8 #include <stdio.h>
 9 #include <stdlib.h>
10 #include <dirent.h>
11 #include <unistd.h>
12 #include <sys/stat.h>
```

```
13 int main(int argc, char *argv[])
14 {
15   DIR *dir;
16   struct dirent *ptr_to_dir_entry;
17   int num, filetype;
18   char filename[256];
19   char *mode[]={"---","--x","-w-","-wx",
20                 "r--","r-x","rw-","rwx"};
21   struct stat filestatus;

22   if (argc != 2) {
23     printf("Give a directory_name "
              "as a command parameter.\n");
24     exit(EXIT_FAILURE);
25   }else if ((dir=opendir(argv[1])) == NULL) {
26     perror("opendir");
27     exit(EXIT_FAILURE);
```

```
28   }

29   printf("Given directory %s¨ "
               "contains following files:\n\n"
30          "   mode      size(bytes)   filename\n"
31          "----------  -----------   --------\n",
32          argv[1]);
33   num=0;
34   while ((ptr_to_dir_entry=readdir(dir)) != NULL) {
35     ++num;
36     sprintf(filename, "%s/%s",
          argv[1], ptr_to_dir_entry->d_name);
37     if (lstat(filename, &filestatus) < 0){
38       perror("lstat");
39       fprintf(stderr, "   (lstat(%s, &filestatus))\n",
40                filename);
41       continue;
```

```
42      }
43      switch (filestatus.st_mode & S_IFMT) {
44      case S_IFREG:  filetype='-'; break;
45      case S_IFDIR:  filetype='d'; break;
46      case S_IFLNK:  filetype='l'; break;
47      case S_IFBLK:  filetype='b'; break;
48      case S_IFCHR:  filetype='c'; break;
49      case S_IFSOCK: filetype='s'; break;
50      case S_IFIFO:  filetype='p'; break;
51      default:       filetype='?'; break;
52      }
53      printf("%c%s%s%s  %10d   %s\n",
54             filetype,              右シフト
55             mode[(filestatus.st_mode>>6) & 07],
56             mode[(filestatus.st_mode>>3) & 07],
57             mode[filestatus.st_mode & 07],
58             filestatus.st_size,
```

```
59              ptr_to_dir_entry->d_name);
60    }

61    printf("Totally %d files exist.\n",num);
62    closedir(dir);
63    return 0;
64 }
[motoki@x205a]$  gcc list-directory-contents2.c
[motoki@x205a]$  ./a.out ~/Operating-Systems2011
Given directory "/home/motoki/Operating-Systems2011" contains

   mode      size(bytes)  filename
----------   -----------  ----------------
-rw-r--r--       201351   OS-PROJ.log
-rw-r--r--        11871   sec-hardware.tex
drwxr-xr-x         4096   Shukketsu
-rw-r--r--        25199   sec-role-and-structure.tex
```

```
-rw-r--r--     108851   sec-UNIX-file-system-PROJ.tex
-rw-r--r--     110109   sec-process-management.tex
drwxr-xr-x      36864   Figs
-rw-------      22398   Diff-lecture2011-2009.tex~
-rw-r--r--     121267   sec-UNIX-file-system.tex~
-rw-r--r--      34400   sec-how-os-manages-io-devices.tex~

     .............(途中省略).............

-rw-r--r--      97624   sec-paging-policy-PROJ.tex
-rw-r--r--      65086   sec-file-management-PROJ.tex
-rw-r--r--       8113   sec-virtual-storage-interface.tex
-rw-------      22687   Diff-lecture2011-2009.tex
-rw-r--r--      51762   sec-getting-process-features-PROJ.tex
-rw-r--r--      73579   sec-pipe-PROJ.tex
-rw-r--r--      28492   sec-mutual-exclusion-basics.tex~
-rw-r--r--      62458   OS-lecture.ind
```

```
-rw-r--r--        25202   sec-role-and-structure.tex~
-rw-r--r--        76589   sec-paging-policy.tex
Totally 103 files exist.
[motoki@x205a]$
```

Vine Linux 2.1.5での具体的な定義がp.135の補足の中で示されている。

補足：

Vine Linux 2.1.5では S_IFMT や S_ISREG(), S_ISDIR() , ... といったマクロは <linux/stat.h>の中で次の様に定義されている。

```
#define S_IFMT  00170000
#define S_IFSOCK 0140000
#define S_IFLNK  0120000
#define S_IFREG  0100000
#define S_IFBLK  0060000
#define S_IFDIR  0040000
#define S_IFCHR  0020000
#define S_IFIFO  0010000
#define S_ISUID  0004000
#define S_ISGID  0002000
#define S_ISVTX  0001000

#define S_ISLNK(m)  (((m) & S_IFMT) == S_IFLNK)
#define S_ISREG(m)  (((m) & S_IFMT) == S_IFREG)
#define S_ISDIR(m)  (((m) & S_IFMT) == S_IFDIR)
#define S_ISCHR(m)  (((m) & S_IFMT) == S_IFCHR)
#define S_ISBLK(m)  (((m) & S_IFMT) == S_IFBLK)
#define S_ISFIFO(m) (((m) & S_IFMT) == S_IFIFO)
#define S_ISSOCK(m) (((m) & S_IFMT) == S_IFSOCK)
```

ディレクトリを再度最初から読みたい時のライブラリ関数 void rewinddir(DIR *dir) :

- ヘッダファイル <dirent.h> を必要とする。
- rewinddir( ) が呼ばれると、引数で指定されたディレクトリストリームの最初の位置にストリームポインタが戻される。
- 関数引数 dir の箇所にはディレクトリストリームへのポインタを指定する。

現在の作業ディレクトリを知るためのライブラリ関数 getcwd( ) :

- 関数プロトタイプは<unistd.h> で次の様に定義されている。

```
char *getcwd(char *buf, int size);
```

- ヘッダファイル <unistd.h> を必要とする。
- getcwd( ) が呼ばれると、現在の作業ディレクトリの絶対パス名を引数で指定された領域に格納する。成功するとパス名格納の領域へのポインタを関数値として返し、(格納領域が小さすぎるなどの理由で)失敗するとNULLを返す。
- 関数引数の buf はパス名を格納するchar型配列へのポインタを表す。
- 関数引数の sizeは配列bufの大きさを表す。

現在の作業ディレクトリ変更のための システムコール chdir() と fchdir():

- 各々の関数のプロトタイプは<unistd.h> で次の様に定義されている。

```
int chdir(char *path);
int fchdir(int fd);
```

- ヘッダファイル <unistd.h> を必要とする。

- chdir() や fchdir() が呼ばれると、引数で指定されたディレクトリに作業ディレクトリを移す。成功すると **0** を返し、失敗すると -1 を返す。

- 関数引数の path はファイルをフルパスで指定した文字列を表す。

- 関数引数の fd はファイル記述子を表す。

ディレクトリ作成のためのシステムコール int mkdir(char *path, mode_t mode) :

- ヘッダファイル <unistd.h> と<fcntl.h> が必要とする。
- 成功すると 0 を返し、失敗すると -1 を返す。
- 関数引数の path は、作成するディレクトリを指定した文字列を表す。
- 関数引数の mode は、作成ディレクトリの保護モードを表す。

ディレクトリ削除のためのシステムコール int rmdir(char *path) :

- ヘッダファイル <unistd.h> を必要とする。
- 削除するディレクトリは空でなければならない。成功すると 0 を返し、失敗すると -1 を返す。
- 関数引数の path は、作成するディレクトリを指定した文字列を表す。

10–8 ファイル関連のその他の各種システムコール

(ハード)リンクを1つ削除するためのシステムコール

`int unlink(char *path) :`

- ヘッダファイル <unistd.h> を必要とする。
- unlink() が呼ばれると、引数で指定されたファイルへのリンクが削除される。同時にリンク先のファイルに付随したi-ノード中の「ハードリンク数」が1だけ減らされ、その結果 0 になるとファイル実体そのものも削除される。

  但し、このファイルをオープンしているプロセスがあると、クローズするまで削除されない。成功すると 0 を返し、失敗すると -1 を返す。
- 関数引数の path は、削除するリンク名を指定した文字列を表す。

(ハード) リンクを張るためのシステムコール

`int link(char *path, char *newpath) :`

- ヘッダファイル <unistd.h> を必要とする。
- link( ) が呼ばれると、第1引数で指定されたファイルへの新しいリンクを張る。同時にリンク先のファイルに付随したi-ノード中の「ハードリンク数」が1だけ増やされる。

  成功すると 0 を返し、(新しく張るリンクと同じものが既にある等の理由で) 失敗すると -1 を返す。
- 関数引数の path は、既に存在するリンク名を指定した文字列を表す。
- 関数引数の newpath は、新しいリンク名を指定した文字列を表す。

umask値を変更するためのシステムコール

`mode_t umask(mode_t mask) ;`

ファイルやディレクトリを生成する時、open() や creat() の引数の指定がどうなっているかに拘わらず常にアクセスを拒否することを指定出来る。

アクセスを常に拒否するかどうかを表す9ビットの値を umask値 と言う。(1がアクセス拒否を表す。)

umask値はプログラム/環境毎に設定され、
$\longrightarrow$ 環境変数の様に振る舞う。

大抵のシステムでは 0 または 022 に初期設定される。

- ヘッダファイル <sys/types.h> と<sys/stat.h> を必要とする。
- umask() が呼ばれると、引数で指定された通りにumaskが設定され、古いumask値が関数値として返される。
- 関数引数の mode は新しいumask値を表す。

ファイルへのアクセスチェックのためのシステムコール access( ) :

- 関数プロトタイプは次の通り。

```
int access(char *path, int acccesstype);
```

- ヘッダファイル <unistd.h> を必要とする。

- access( ) が呼ばれると、第1引数で指定されたファイルに対して第2引数で指定したアクセスが全て可能かどうかの判定を行う。全てアクセス可能なら 0 を返し、それ以外の場合は -1 を返す。

- 関数引数の path は、アクセス可能性を判定するファイルを指定した文字列を表す。

- 関数引数の acccesstype は、どういう種類のアクセスについて調べるかを表したもので、次のマクロをOR演算子(|)で繋げて指定する。

  R_OK · · · 読み込み可能かどうかを調べる。
  W_OK · · · 書き込み可能かどうかを調べる。
  X_OK · · · 実行可能かどうかを調べる。
  F_OK · · · ファイルが存在するかどうかを調べる。

システムバッファの内容をディスクに書き出すための システムコール void sync( ) :

- sync コマンドに対応。
- ヘッダファイル <unistd.h> を必要とする。
- sync( ) が呼ばれると、システムバッファ内のデータがディスクに書き込まれ、ディスク内のデータが最新のものに保たれる。
- 不用意にvoid sync( ) システムコールを多用するとシステムの性能低下を引き起こす可能性がある。