ファイル入出力と
プロセス間通信 (1)

2004年12月10日

海谷 治彦

目次

- まずはマニュアルをみよう.
  - 2章 システムコールインタフェース
  - 3章 汎用関数定義
- アンバッファー化入出力 (Unbuffered I/O)
  - open, read, write ...
  - lseek, dup ....
- 標準入出力ライブラリ
  - fopen, fscanf, fprintf ...
- 標準入力，標準出力，標準エラー
  - stdin, stdout, stderr
- 汎用ポインタ，システムデータ型
  - void*, size_t, pid_t 等

Linuxオンラインマニュアル

いまさらですが・・・・

- システムコールは2章，ライブラリ関数は主に3章，コマンドは1章にあります．
- プログラマから見れば，システムコール(正確にはシステムコールインタフェース)もライブラリも単なるライブラリ関数です．
- しかし，OSの観点からは結構違うことを既に理解してると思います．
- わかっている人にしかわからないような不親切な記述が多いですが，それでもがんばって読んで．

1章 コマンド

```
File  Edit  Setup  Control  Window  Help
[kaiya@linux2002 ~]% man 1 intro
```

```
INTRO(1)            Linux Programmer's Manual            INTRO(1)

名前
        intro - ユーザーコマンドの説明

説明
        この章はユーザー

著者
        著者と著作権
        見ること。こ

Linux
~
~
~
```

```
CP(1)                                                    CP(1)

名前
        cp - ファイルやディレクトリをコピーする

書式
        cp [オプション] file path
        cp [オプション] file... directory

        POSIX オプション: [-fipRr]

        GNU  オプション (簡略形式): [-abdfilprsuvxPR] [-S SUFFIX]
        [-V {numbered,existing,simple}] [--sparse=WHEN] [--help]
        [--version] [--]

説明
        cp  はファイル (あるいはそのように指定すればディレクトリ)
        をコピーする。 1 つのファイルを指定先にコピーしたり、複 数
        のファイルを指定ディレクトリにコピーしたりできる。

        最 後 の引き数が既に存在するディレクトリを指している場合、
        cp はコピー元の file をそれぞれ (同じ名前のまま) その ディ
```

```
File  Edit  Setup  Control  Window  Help
[kaiya@linux2002 ~]% man 1 cp
```

2章 システムコール

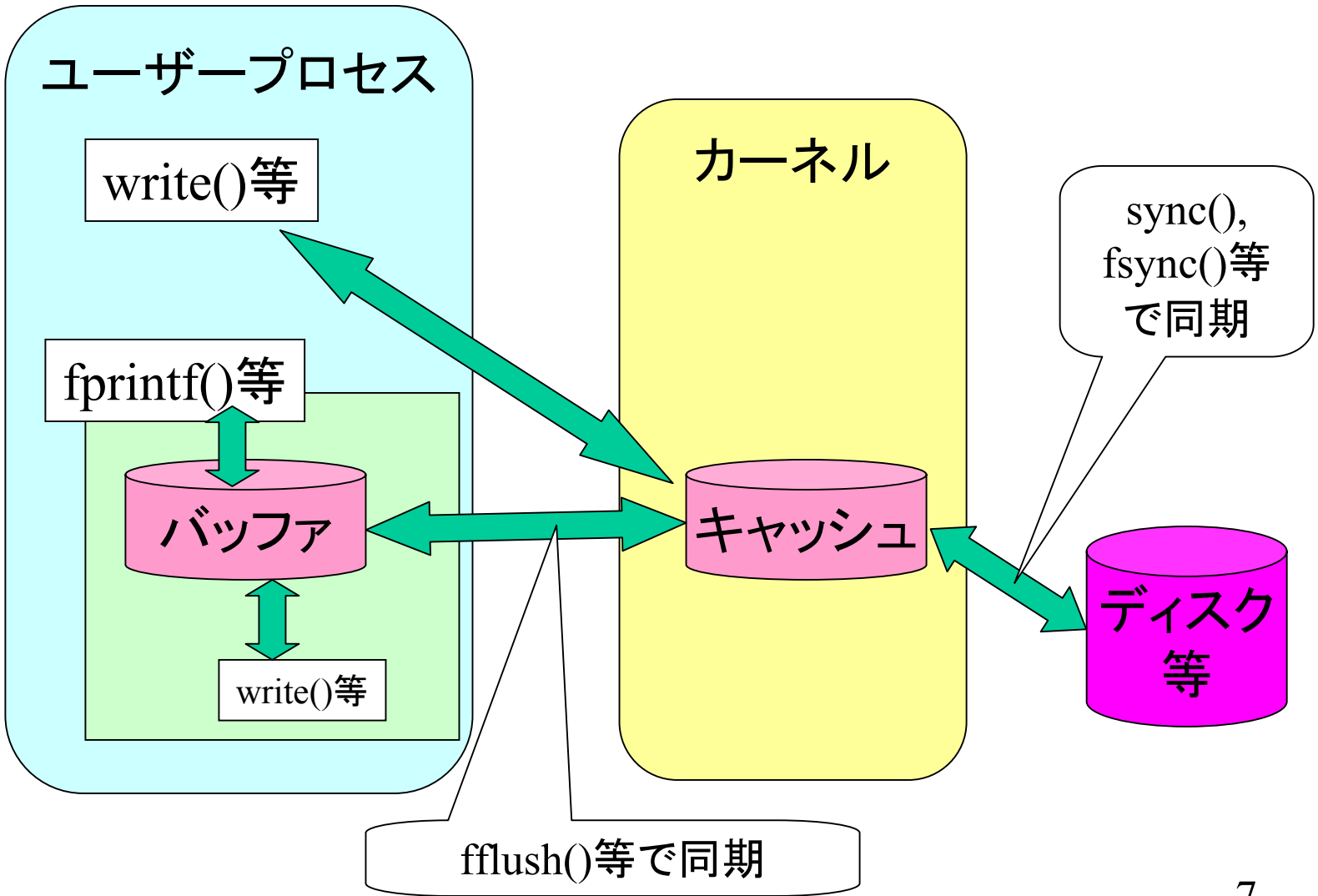
```
File  Edit  Setup  Control  Window  Help
[kaiya@linux2002 ~]% man 2 read
```

```
File  Edit  Setup  Control  Window  Help
READ(2)                Linux Programmer's Manual                READ(2)

名前
        read - ファイル・ディスクリプターから読み込む

書式
        #include <unistd.h>

        ssize_t read(int fd, void *buf, size_t count);

説明
        read() はファイル・ディスクリプター(file descriptor) fd か
        ら最大 count バイトを buf で始まるバッファーへ読み込もうと
        する。

        count がゼロならば、read() はゼロを返し他に何も起きない。
        count が SSIZE_MAX より大きければ、結果は特定できない。

返り値
        成功した場合、読み込んだバイト数を返す(ゼロはファイルの 終
        り を意味する)、ファイル位置はこの数だけ進められる。この数
        が要求した数より小さかったとしてもエラーではない; 例えば今
        す ぐには実際にそれだけの数しかない場合(ファイルの最後に近
        いのかもしれないし、パイプ(pipe)や端末(terminal)から読み込
```

システムコールとライブラリの違い

- システムコール(インタフェース)
  - カーネル(OS)に処理を依頼する.
    - ユーザーモードでは直接扱えないハードウェア等の資源へのアクセスを依頼する.
- ライブラリ関数
  - カーネルに処理依頼の必要がない処理.
    - 例えば, strlen()等.
  - システムコールのラッパー
    - open()に対するfopen()や, read()に対するfscanf()等.

データがデバイスに届くまで



何段かのコピー

- 前スライドのように，データが物理的に記録されるまで，何段かのコピーを作っている.
  - 加えて，CPU側でさらにキャシュしている場合もある.
- 少なくとも，標準入出力関数を使うより，システムコールを使ったほうがコピー回数が少ない.
- しかし，一般に標準入出力関数のほうが使い勝手が良い場合が多い.
  - 書式設定等ができるなど.

ファイル関係のシステムコール

- int open(const char *pathname, int flags);
  - ファイルを開ける関数，いいかえれば，
  - プロセスがファイルを操作可能な状態にする関数.
  - ファイルディスクリプタを返す.
- ssize_t read(int fd, void *buf, size_t count);
  - 読む関数.
- ssize_t write(int fd, const void *buf, size_t count);
  - 書く関数

シーケンシャルアクセス

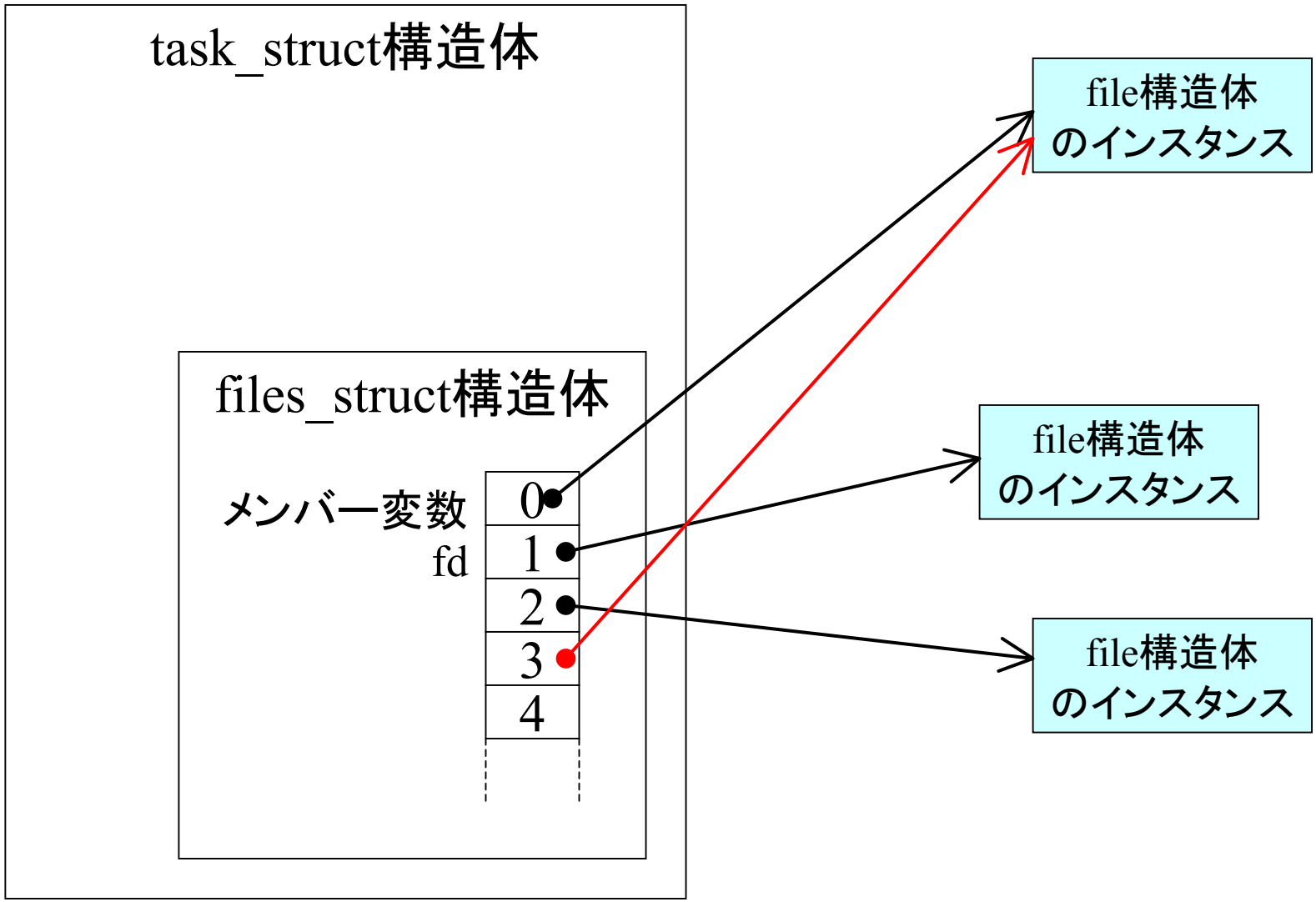
- readもしくはwriteを行う場合，(後述のlseek等を使わなければ，)ファイルの中身を順次アクセスしかできない.
  - 読みきった部分には戻れない.
- よってファイルの後戻りをしたい場合，
  - プログラム内に配列等として読み込んでおく.
  - lseekで読み位置を戻す.
    - ただし，どんなファイルでも戻せるわけじゃない.

  のどちらかの対処が必要.

ファイル・ディスクリプタ

- File Descriptor, 値は自然数値(0, 1, 2 ...)
- 実体は，カーネル内にある file構造体のインスタンスを指している.
- file構造体が保持している情報として重要なのは，「ファイル内の次に処理を行われる位置」
- 複数のファイルディスクリプタで同一のfile構造体のインスタンスを指すことができる.
  - ⇒ 異なるファイルディスクリプタ番号で同じファイルを操作できる.
  - ⇒ さらに，異なるプロセスが1つのfile構造体を共有することもできる.

概念図 (全部カーネルの中)



lseekとdup

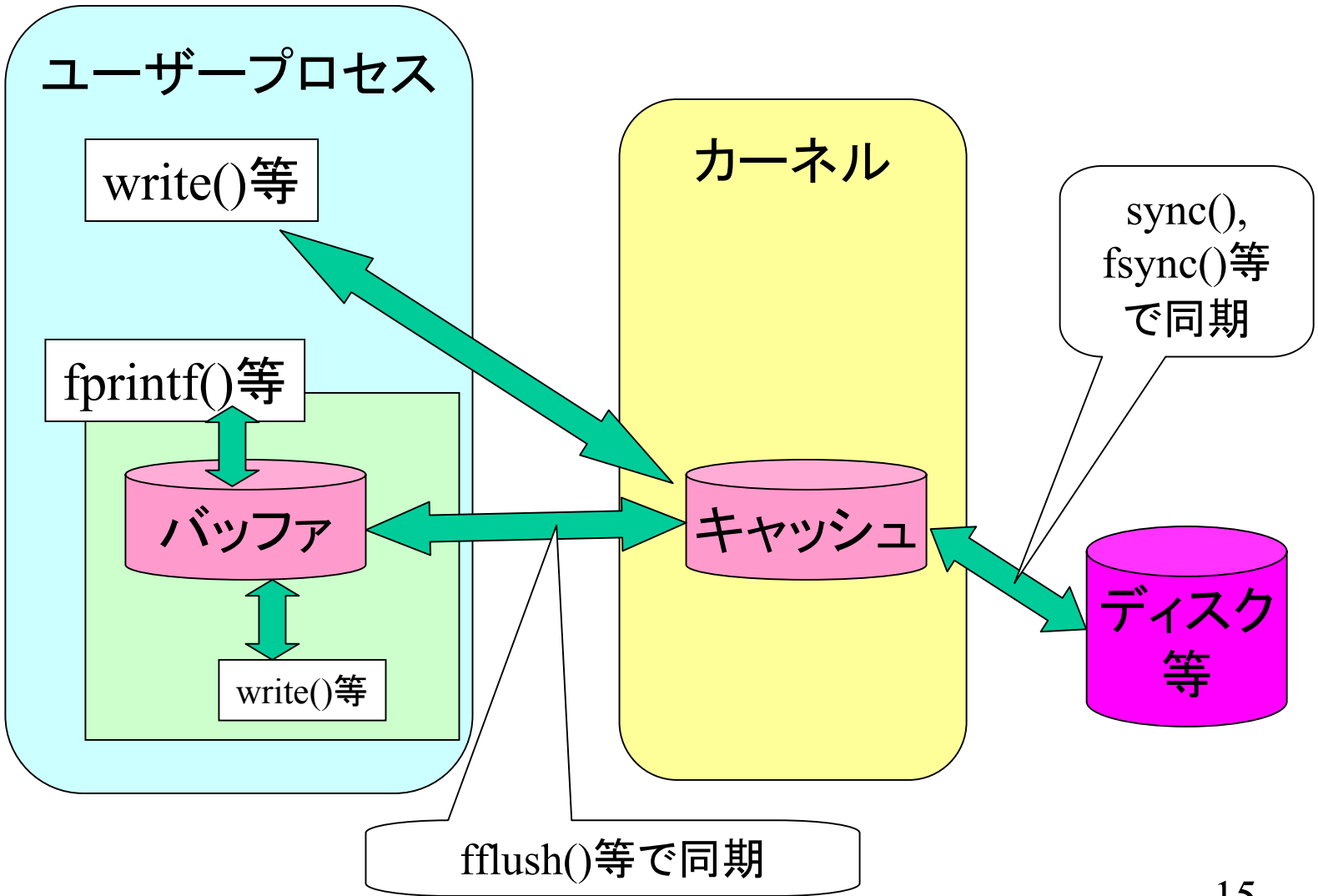
- off_t lseek(int fildes, off_t offset, int whence)
  - 特定のファイルディスクリプタの現在の読み出し位置を変更する.
  - ファイルを配列のようにランダムアクセスできる感じ.
  - 読み位置を変更できないファイルもある. (パイプ等)
- int dup(int oldfd)
  - ファイルディスクリプタの複製を作る.
  - 要は前ページの赤字の状態を作る.
  - 新たに利用されるディスクリプタの値は使っていない最小値となることが保障されている.

file構造体の共有を例示

```
main(){
int newfd;
char buf[100];
  // 標準入力をdupで複製
  if((newfd=dup(0))<0) exit(1); // dup fail.
  fprintf(stderr, "%d is duplicated.¥n", newfd);
  // 複製した方で読み位置を進めて見る
  if((int)lseek(newfd, 200, SEEK_CUR)<0) exit(2); // seek fail.
  // 複製もとの0から値を読んで，標準出力に表示すると，
  read(0, buf, 100); write(1, buf, 100);
  // 先頭からではなく，
  // さっき200B進めた位置から100B表示される.
}
```

再録

データがデバイスに届くまで



sync()とfsync()

- ともに，カーネル内のキャッシュを実際のディスク等の装置に書き戻すシステムコール.
- 無論，カーネルは定期的にこれらを実行しているが，気になる人はアプリケーションから呼び出してもよいだろう.
- sync()等をする前にOSやマシンが異常終了(例えば停電)すれば，無論，データは飛んでしまう.

標準入出力関数

- fopen, fprintf, fscanf等，お馴染みの関数群.
- これらはストリーム(データの流れ，というか列)に対する操作が中心となる.
- しかし，最大の特徴はバッファリング(buffering)である.

バッファリング (buffering)

- 前述の図のように，いきなりread/writeシステムコールを呼び出すのではなく，
- 記憶領域(コレのことをbufferと呼ぶ)にデータをある程度溜め込んでから入出力を行うこと.
- 結果としてシステムコールの呼び出し回数を減らすことができ，プログラムを効率化できる.
- しかし，現実には「書いたつもりのデータがすぐに書かれない」等が起こり，プログラマには悩みの種. (かも)

三種類のバッファリング

- 完全なバッファリング
  - バッファーのサイズ(マクロBUFSIZで規定)一杯にbufferingをする.
  - ディスク上のファイルはこの方式がデフォルト.
- 行バッファリング
  - 改行がくるかbufferサイズを超えるまでbufferingをする.
  - 端末装置とつながっている場合，この方式をとる.
  - stdin, stdoutは通常コレ.
- アンバッファド
  - bufferingをしない.
  - 可能な限り速やかに入出力を行う.
  - stderrは通常コレ.
  - 無論，システムコール呼び出しは頻繁になる.

バッファリング方式の変更

- setbuf, setvbuf関数(システムコールでは無論ない)で，バッファリング方式を変更できる.
- 以下の例ではstdoutを強制的に完全バッファリングにしている.
- 結果として，getchar()で文字を読んだあとのprintf命令が実行されるまで，Helloは出力(表示)されない.

```
#include <stdio.h>

main(){
  setvbuf(stdout, NULL, _IOFBF, BUFSIZ);
  printf("Hello ");
  getchar();
  printf("World %d¥n", BUFSIZ);
}
```

ファイルディスクリプタとストリーム

- FILE *fdopen (int fildes, const char *mode)
  - を用いて，ディスクリプタからストリームを生成することができる.
- すなわち，ディスクリプタに使いやすい皮をかぶせることができる.
- fopenで開けられない特殊なファイル(通信装置等)を使いやすくする際に用いられるらしい.
- int fileno(FILE *stream)
  - 逆にストリームからディスクリプタを得ることもできる.

ストリームの読書き

- 数えられないくらい関数があるのはご存知の通り.
- 読み用
  - fscanf, fgets, fgetc, fread ....
- 書き用
  - fprintf, fputs, fputc, fwrite ...
- バイナリファイルとテキストファイルの扱い等，微妙に異なる場合があるので厄介なことがある.
  - read, writeシステムコールの場合，バイナリ，テキストの区別はない.

ストリームの位置決め (1)

- long ftell(FILE *stream)
  - 現在の読み位置
- int fseek(FILE *stream, long offset, int whence)
  - 特定位置まで移動させる. lseekに対応する.
  - しかし, テキストファイルでは多少問題があるらしい.
- void rewind(FILE *stream)
  - 先頭までまき戻す.
  - これは結構よく見る.

ストリームに位置決め (2)

- int fgetpos(FILE *stream, fpos_t *pos)
  - 現在位置を*posに保存.
  - あとでfsetposで使う.
- int fsetpos(FILE *stream, fpos_t *pos)
  - *posで指定された位置に移動する.
- 上記のほうが(1)のよりお勧めらしい.

使い分けについて

- read, write と fprintf, fscanfを混ぜて使うことは不可能ではない.
- しかし，バッファリング問題もあり，わけわかんなくなるので，1つの入出力先ではどちらかに統一したほうが良いだろう.
- アプリケーション寄りのものは標準入出力関数を用いて，システム寄りはシステムコールを使うのが一般的か・・・

汎用ポインタとシステムデータ型

- 汎用ポインタ void*
  - どの型のポインタにもマッチする(明示的にキャストしなくていい)ポインタ.
  - 結果としてメモリを扱う関数ではchar*にかわり使われるようになった.
- システムデータ型 なんとか_t
  - 実体はintやlong等なのだが，ソースコードの移植性をよくするために，昨今では使われる.
  - sys/types.hに主に定義されているようだ.
  - ssize_t, pid_t, fpos_t 等.