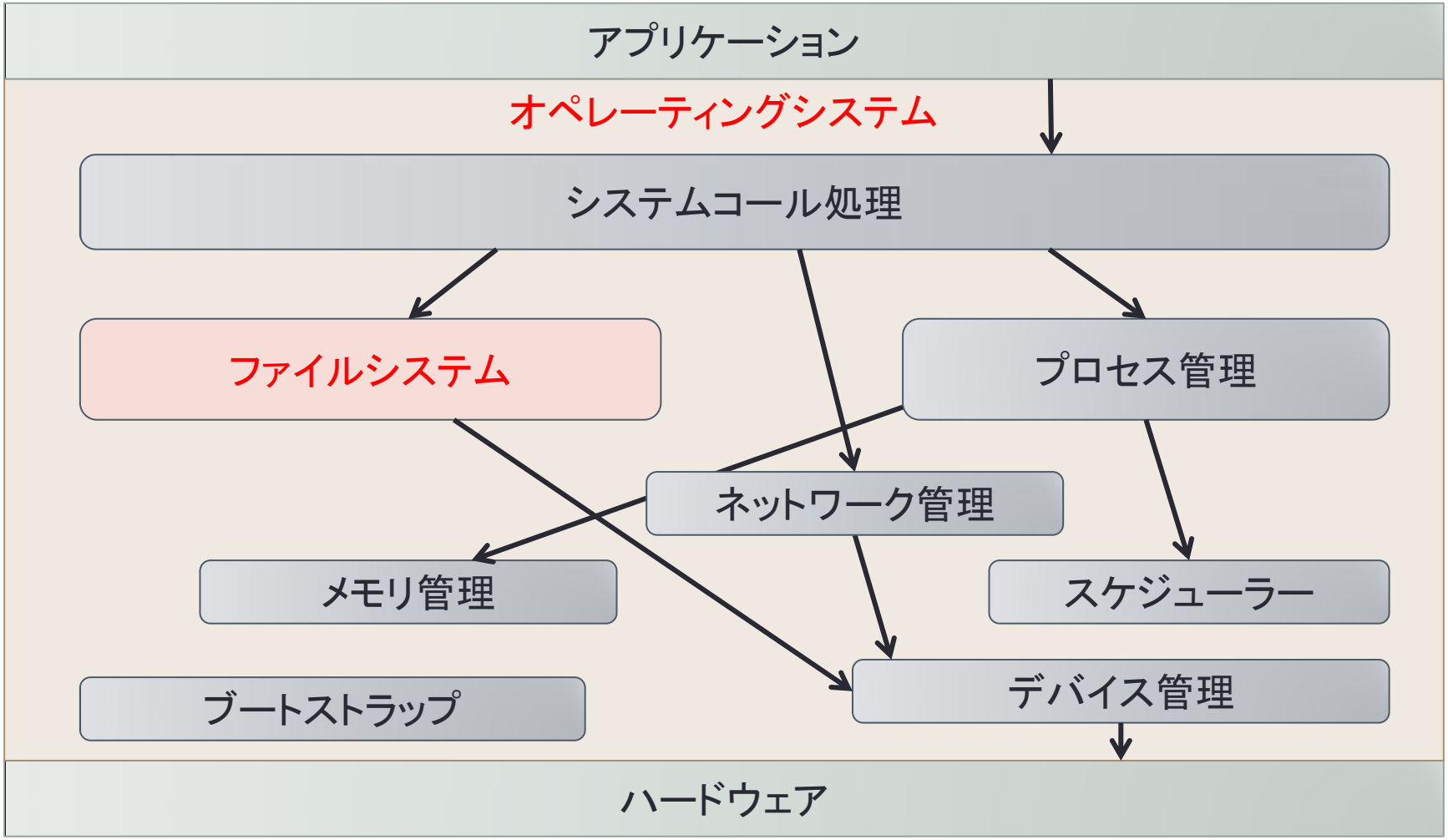
# ソフトウェアアーキテクチャ
# 第2回　ファイルシステム

環境情報学部
萩野　達也

スライドURL

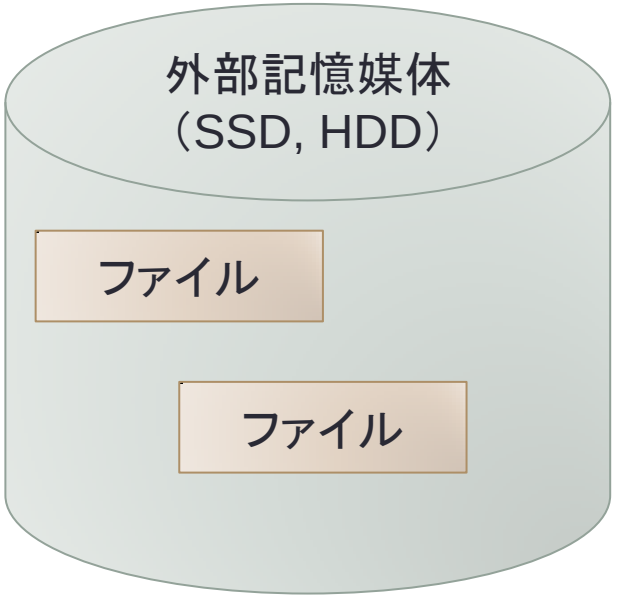
https://vu5.sfc.keio.ac.jp/slide/

# オペレーティングシステムの構成要素

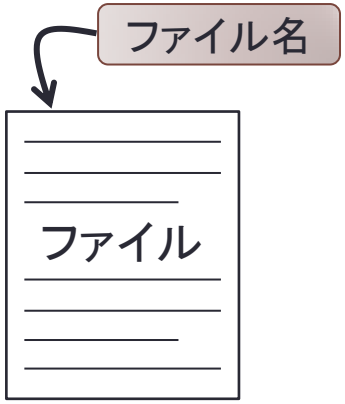


# ファイルとは

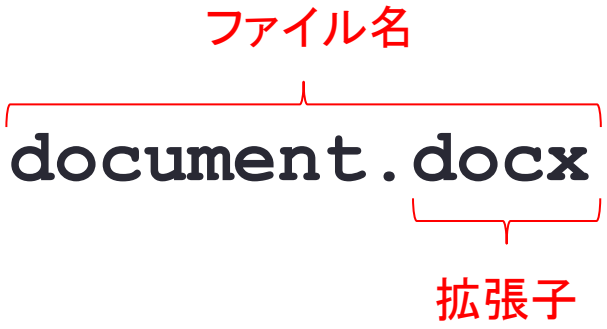
- 情報を外部記憶媒体に記録する単位
  - データセットとも呼ばれたことがある
- ファイルの特徴
  - ファイル情報は不揮発(non-volatile)
    - 電源を切っても消えない.
  - ファイル情報は永続的(persistent)
    - 続きから始めることができる.
- ファイルシステム
  - 外部記憶媒体上のファイルを管理
  - Windows → NTFS
  - MacOS → HFS
  - Unix → UFS
- ファイル関連の取り決め
  - ファイル名
  - ファイル構造
  - ファイルの種類
  - ファイルのアクセス方法



# ファイル名に関する取り決め

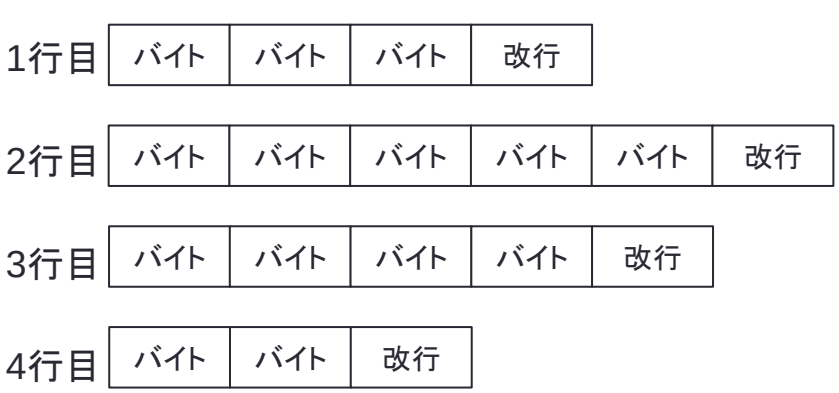


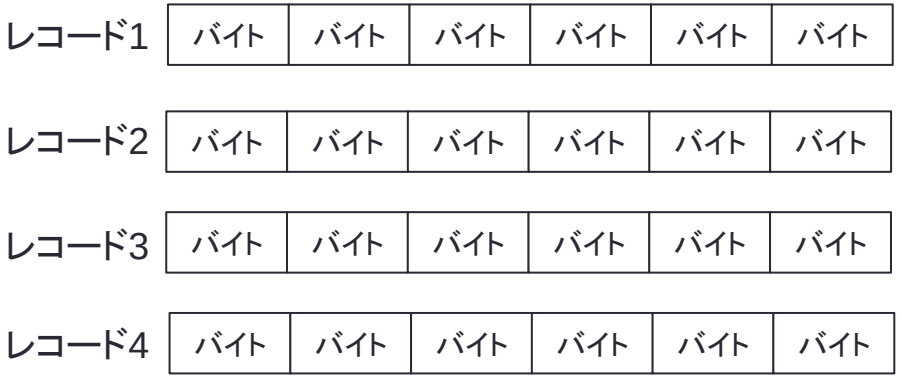

- 大文字および小文字の区別がない場合もある
  - ファイル名の文字コードはどうする?
- ファイル名の文字数には制限があるものもある
  - MS-DOSでは8+3
  - UNIXでは255
- ファイル拡張子（file extension）によりファイルの種類を表す場合もある
  - Windowsでは起動されるプログラムが異なる
  - Macでは以前は内部的に保持していた
  - UNIXはアプリケーション任せ
    - コンパイラは拡張子によって言語を判断



# ファイルの構造

- バイト列 vs レコード列
  - ファイルはバイトの並んだものであると考え，何の構造もない
  - ユーザは任意のフィールドを扱うことができる
  - テキストファイルの場合は改行文字（LFまたはCR）で区切る
  - ファイルは固定長のレコードの集まりとして取り扱われる
  - パンチ・カード時代には1 レコード80 文字としていた
  - レコード単位で読み書きが可能である.



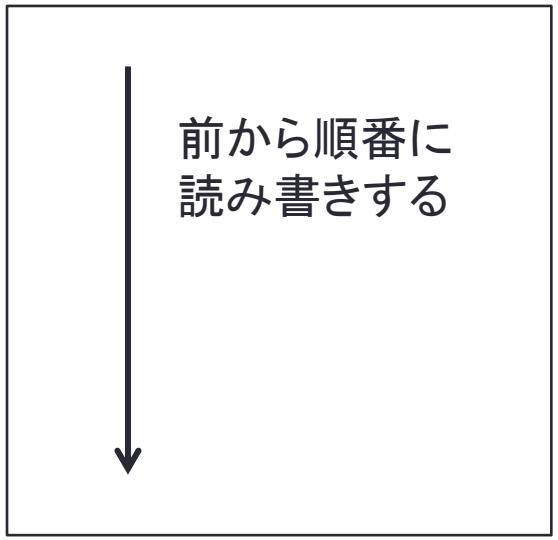



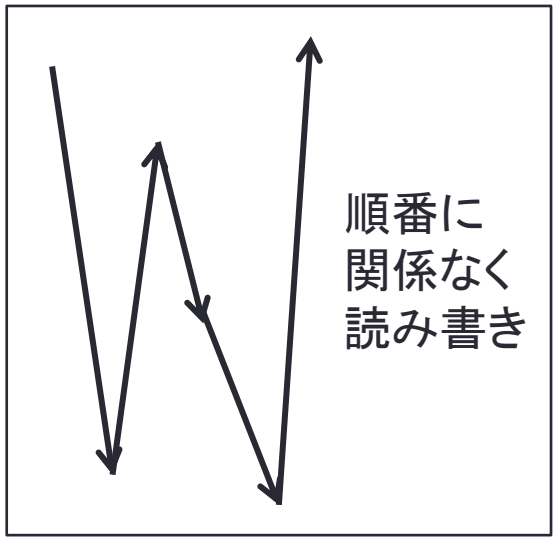
# ファイルの種類

- 通常ファイル（regular file）
  - テキストファイルまたはバイナリの2 種類がある
- ディレクトリ（directory）
  - ファイルを管理するためのファイル
  - フォルダー（folder）とも呼ばれる
- 文字型特殊ファイル（character special file）
  - 入出力関連のデバイスを表す.
  - 端末，プリンタ，ネットワークなどのシリアル型のデバイス
- ブロック型特殊ファイル（block special file）
  - ディスクなどのブロック単位で利用するデバイス
  - HDDやSSDなど
  - ファイルシステムを作ることができる

# ファイルのアクセス方法

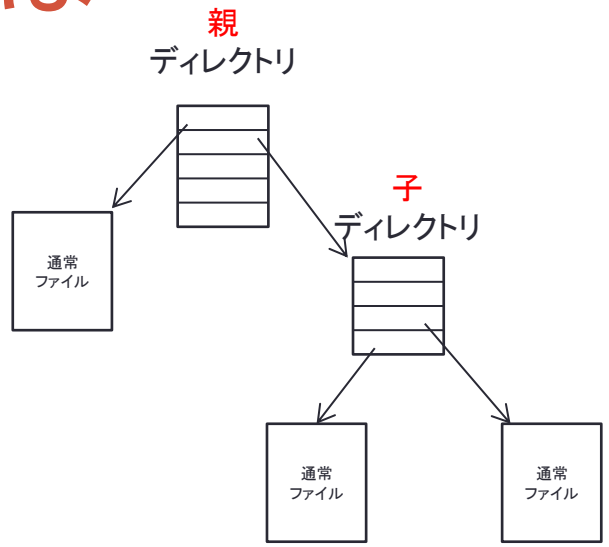
- 逐次アクセス vs 直接アクセス
  - sequential access
  - プロセスはファイルのバイトやレコードを順番に読むことしかできない.
  - スキップしたり，順序を外れたアクセスはできない.
  - 巻き戻しだけできる場合もある.
  - random access
  - 順番に関係なく任意の位置のバイトやレコードをアクセスできる.
  - UNIX ではseek システムコールで読み書きする位置を指定できる.



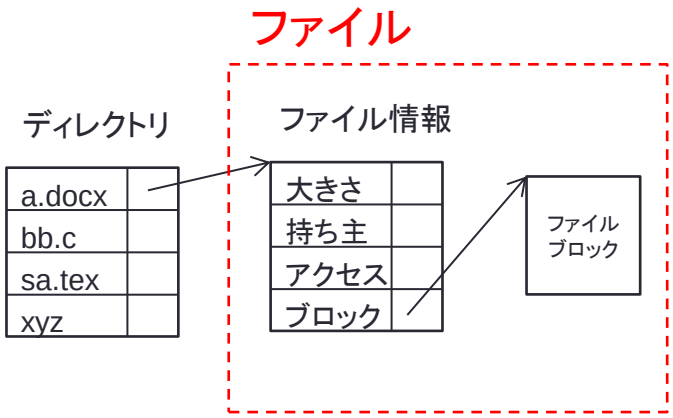



# 階層型ファイルシステムとは

- ディレクトリの中にディレクトリを入れることを許すことによって実現




- ディレクトリの中身
  - 2 種類の方法がある.
    - ファイル名と属性がエントリに入っている
    - ファイル名は入っているが，属性は別に管理され，そこへのポインタがエントリに入っている




- ファイルの指定方法
  - パス名により指定する
  - ディレクトリ名とファイル名を/や¥などの区切り記号つないだもの
    - UNIX → /
    - Windows → ¥
    - 旧Mac → :

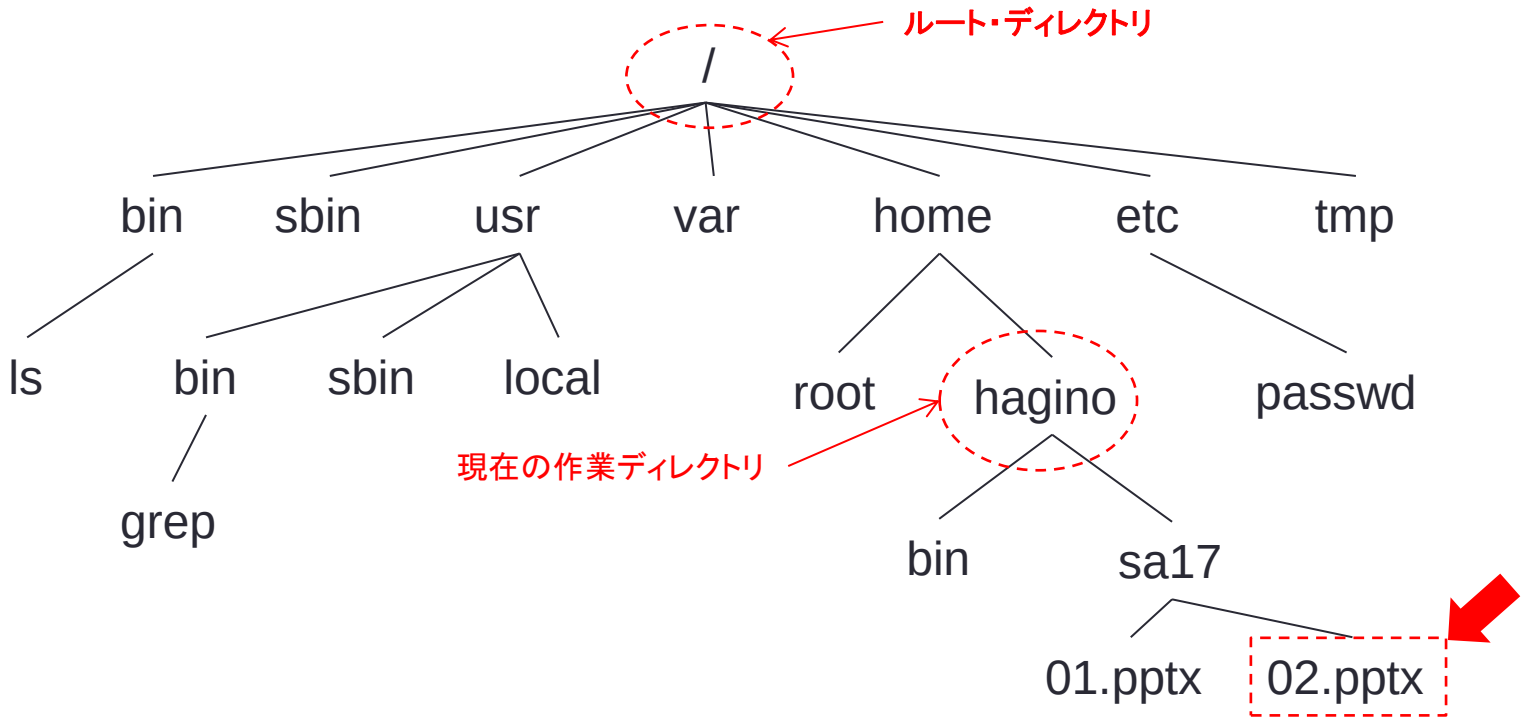
# 絶対パス名と相対パス名

- 絶対パス名（absolute path name）
  - ルート・ディレクトリからディレクトリのたどるサブディレクトリ名を/などで区切ることにより指定

**/home/hagino/sa17/02.pptx**

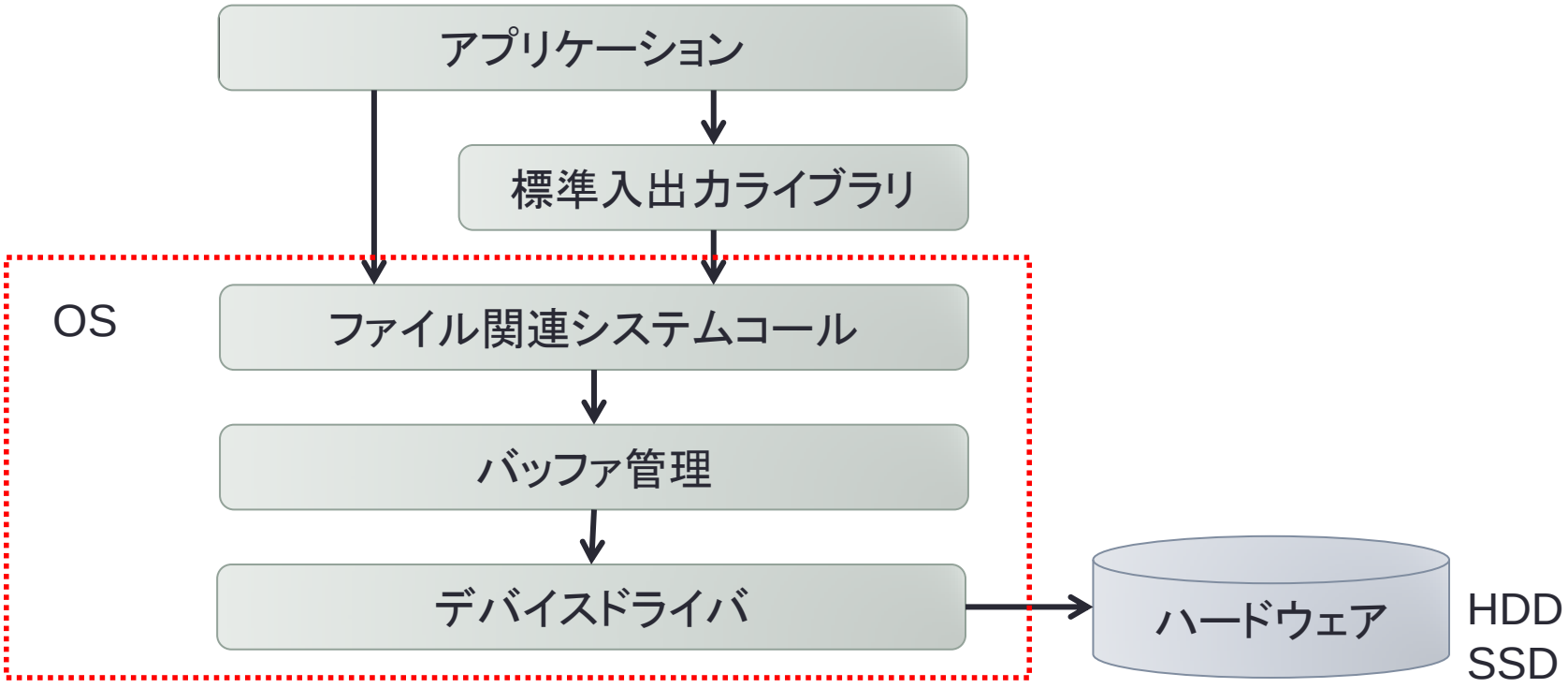
- 相対パス名（relative path name）
  - 作業ディレクトリ（working directory）からたどる

**sa17/02.pptx**

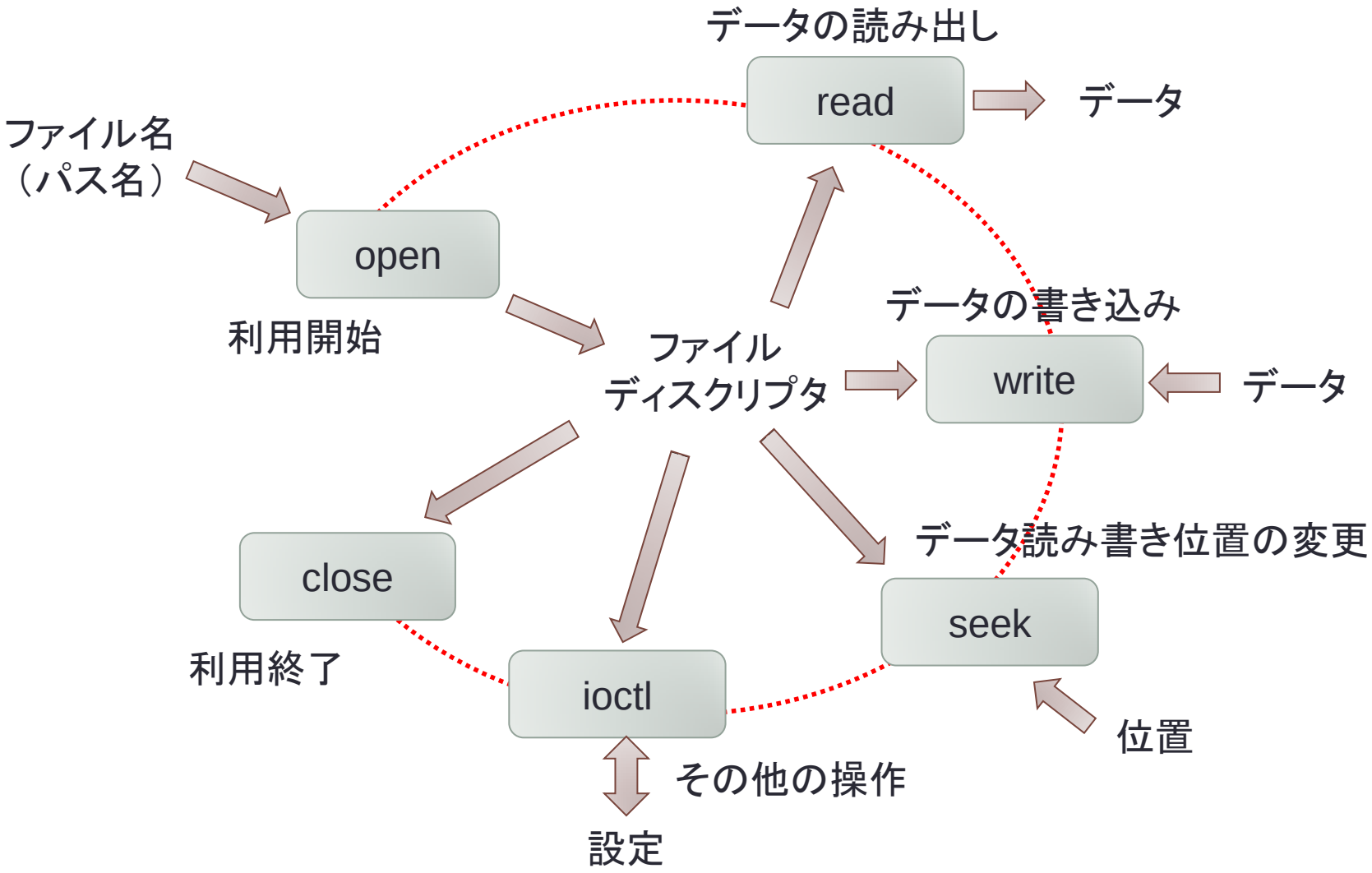


# ファイルの読み書きの仕組み

- UNIX を例に説明
- 直接システムコールを使ってファイルの読み書きを行なう場合もある
- 通常は標準入出力ライブラリ経由で利用



# ファイル関連のシステムコール



# ファイルディスクリプタ

- openで返される小さな数字
  - 読み書き操作を行うファイルを表す
  - 読み書きの位置を覚えている（read, writeで更新，seekで変更）

- プロセス毎に管理
  - 数字の意味はプロセスごとに異なる
  - 一つのプロセスでも同じファイルを2回openすると違うファイルディスクリプタ

- 特別なファイルディスクリプタ
  - 標準入力 → 0
  - 標準出力 → 1
  - 標準エラー出力 → 2

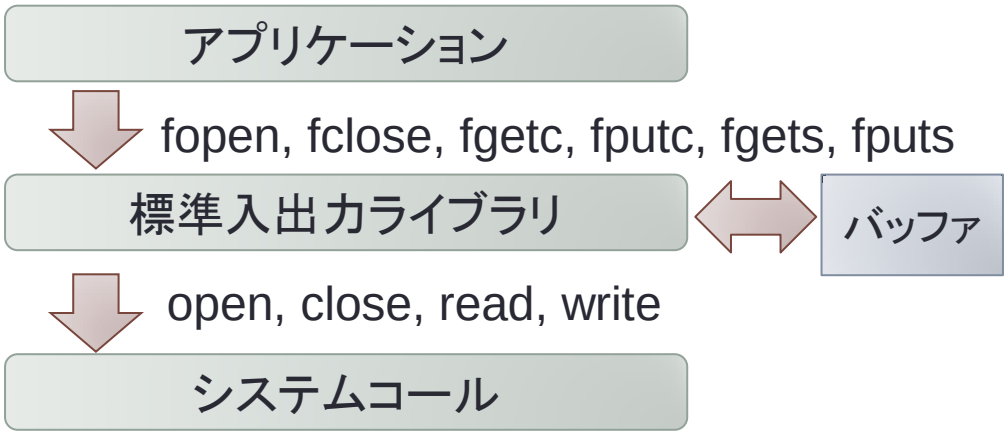
# ファイル関係のシステムコールを使い ファイルを読みコンソールに出力するプログラムの例

```
#include <fcntl.h>

int main(argc, char *argv[])
{
  int fd, n;
  char buf[512];

  fd = open(argv[1], O_RDONLY);
  while ((n = read(fd, buf, 512)) > 0) {
    write(1, buf, n);
  }
  close(fd);
}
```

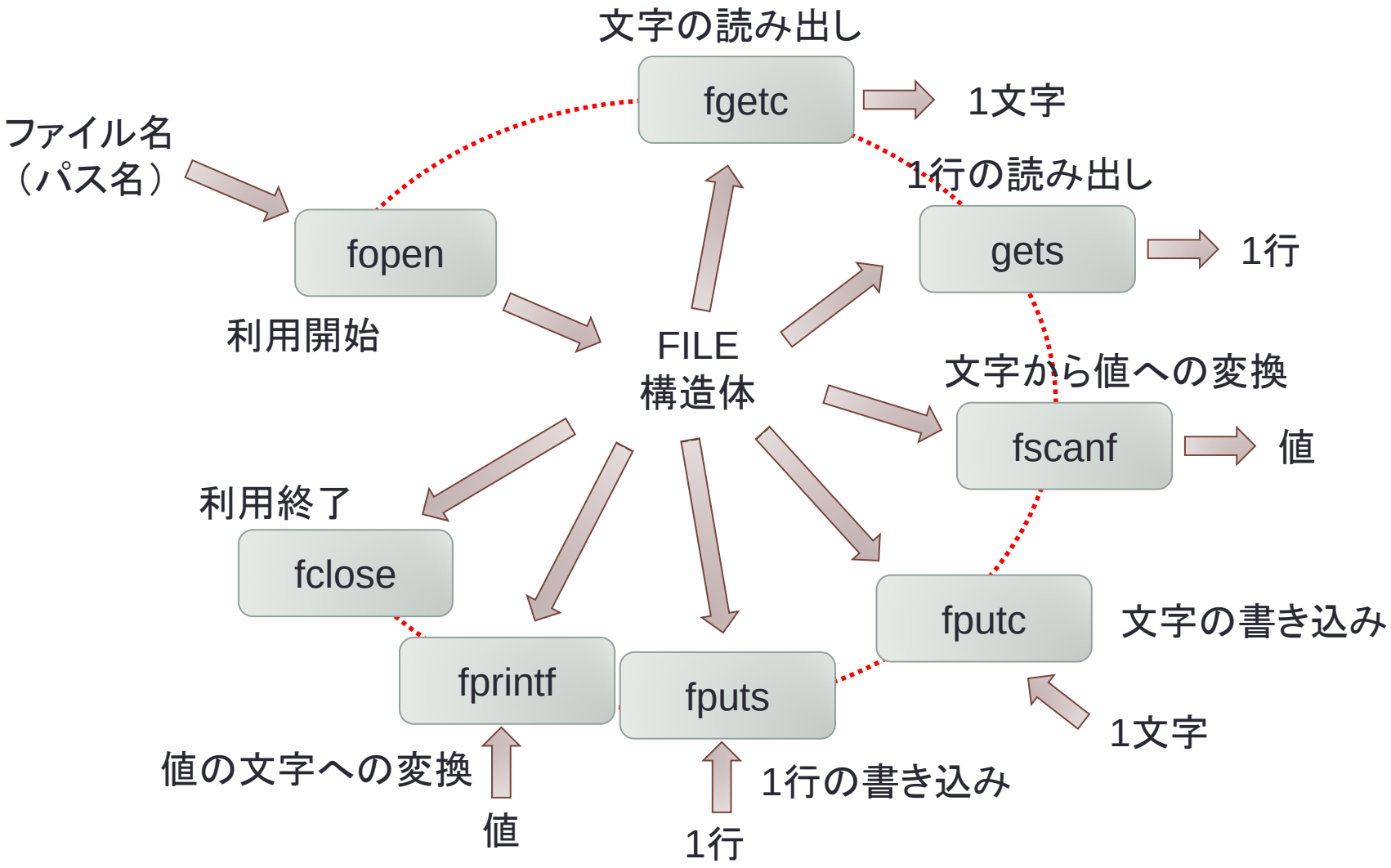
# 標準入出力ライブラリ

- システムコールの問題点
  - 使うのが難しい
    - 例：行単位でのアクセスはできない
  - 小さな単位でアクセスすると効率が悪い
    - システムコールはユーザからOSへの切り替えが必要で，ライブラリを呼び出すのに比べてコストが高い

- 標準入出力ライブラリ
  - 使いやすいAPI
    - 例：行単位で読み込む，1文字ずつ読み込む
  - アプリケーションからシステムコールへのアクセスを最適化
    - バッファリングする



# 標準入出力ライブラリの使い方



# 標準入出力ライブラリを使い ファイルを読みコンソールに出力するプログラムの例

```
#include <stdio.h>

int main(argc, char *argv[])
{
  FILE fp;
  int ch;

  fp = fopen(argv[1], "r");
  while ((ch = fgetc(fp)) >= 0) {
    fputc(ch, stdout);
  }
  fclose(fp);
}
```

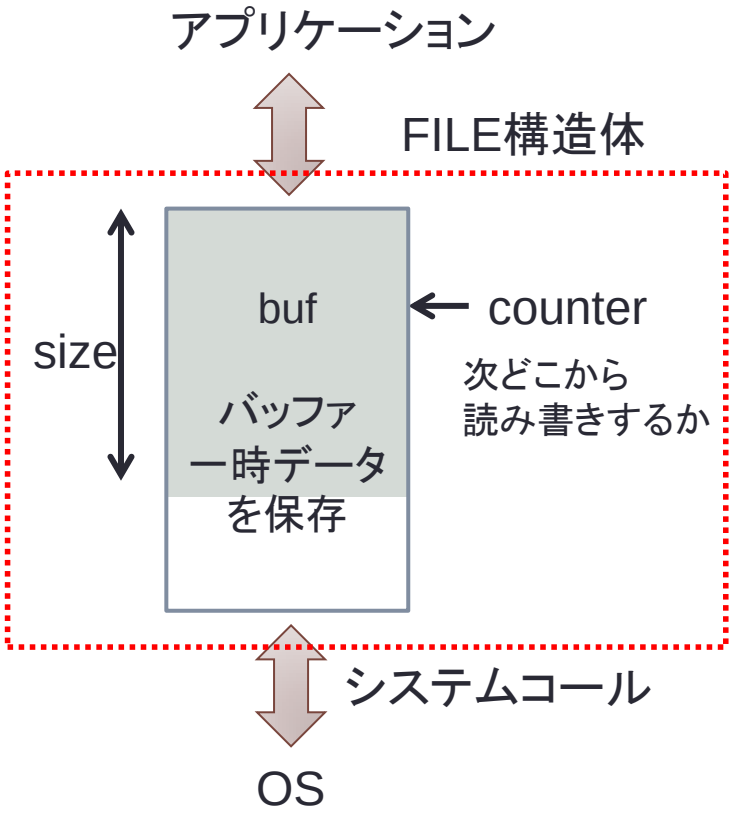
# 標準入出力ライブラリの実装

fopen

```
FILE *fopen(char *path, char *mode) {
  FILE *fp;
  fp = (FILE *)malloc(sizeof(struct FILE));
  fp->fd = open(path, ...);
  fp->size = 0;
  fp->counter = 0;
  return fp;
}
```

FILE構造体

```
struct FILE {
  int fd;
  char buf[BUFSIZE];
  int size;
  int counter;
};
typedef struct FILE FILE;
```

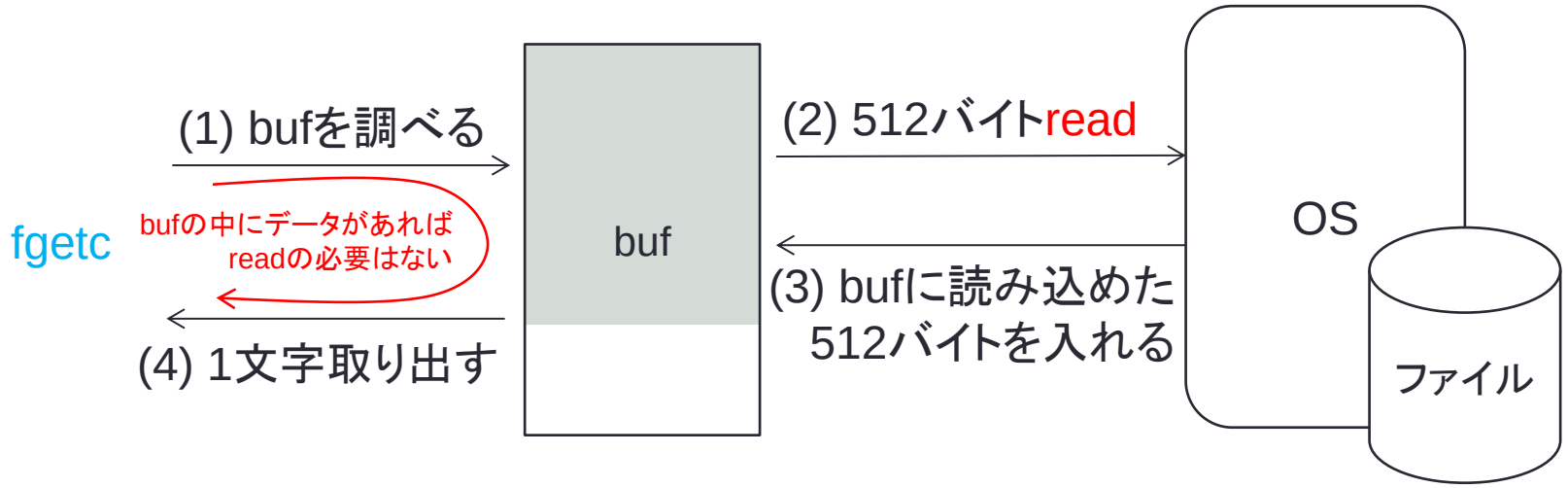


# fgetcの実装

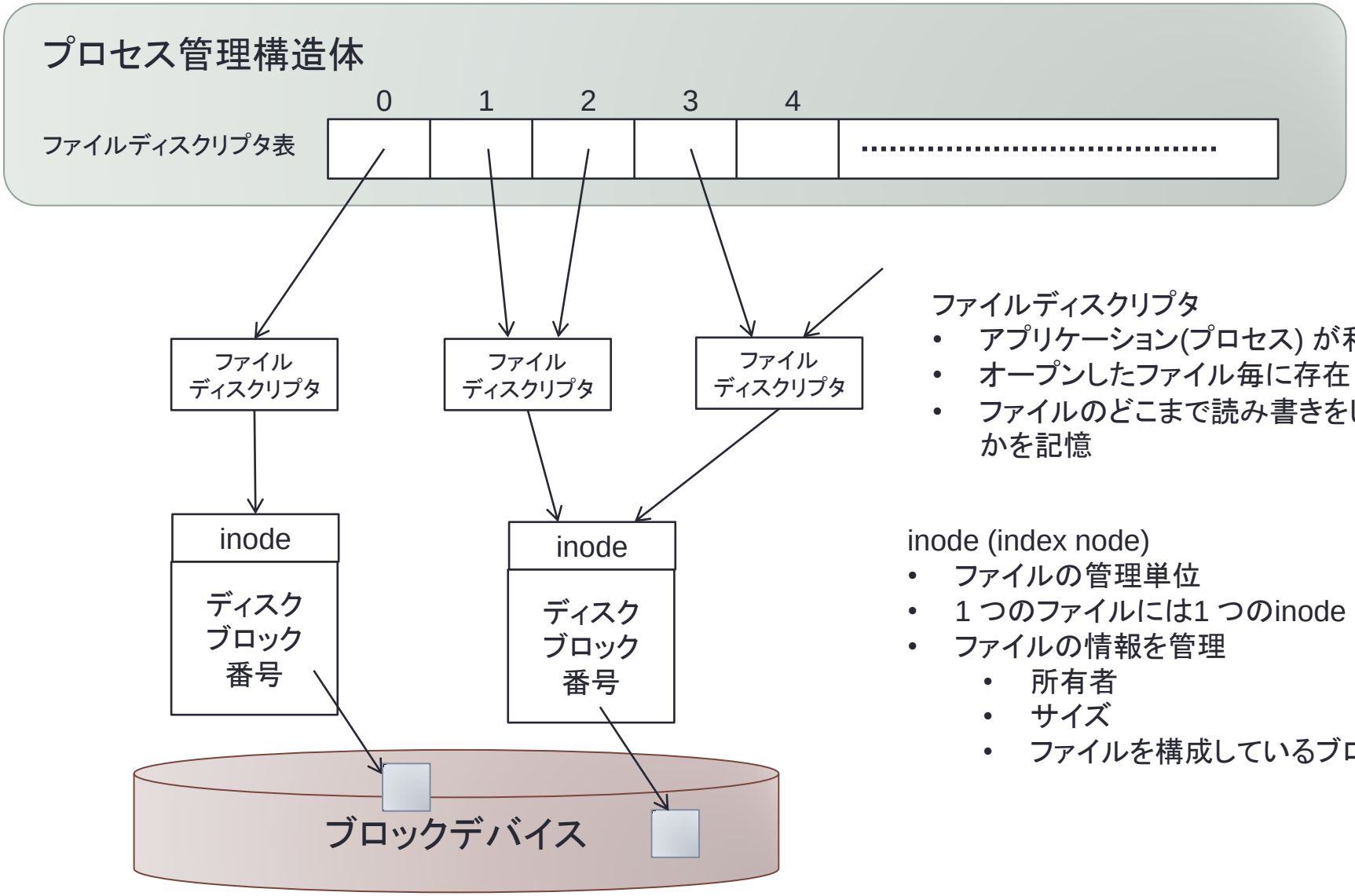
fgetc

```
int fgetc(FILE *fp) {
  if (fp->counter >= fp->size) {
    fp->size = read(fp->fd, fp->buf, BUFSIZE);
    if (fp->size <= 0) return -1;
    fp->counter = 0;
  }
  return fp->buf[fp->counter++];
}
```

BUFSIZE = 512
と仮定



# ファイルシステムコールの実装




ファイルディスクリプタ
- アプリケーション(プロセス) が利用
- オープンしたファイル毎に存在
- ファイルのどこまで読み書きをしたかを記憶

inode (index node)
- ファイルの管理単位
- 1 つのファイルには1 つのinode
- ファイルの情報を管理
  - 所有者
  - サイズ
  - ファイルを構成しているブロック

# openの実装

- 与えられたパス名に対応するファイルディスクリプタを返す
- namei を呼び出しパス名を inode に変換
- プロセス構造体の空きファイルディスクリプタに新しいファイルディスクリプタの構造体へのポインタを設定

```
int open(char *path, int flags, ...) {
  struct inode *ip;
  int fd;
  struct file *fp;
  ip = namei(path);
  if (ip) {
    fp = 新しいファイルディスクリプタの構造体を作成;
    fp->inode = ip;
    fp->offset = 0;
    fp->refcount = 1;
    fd = proc の未使用ファイルディスクリプタ;
    proc->file[fd] = fp;
    return fd;
  }
  else return -1;
}
```

file構造体

```
struct file {
  struct inode *inode;
  long offset;
  int refcount;
};
```

# namei

- ディレクトリの中に指定された名前のエントリが存在するかどうかを調べそのinode を返す

```
struct inode *namei(char *path) {
  struct inode *dp;
  if (*path == '/') {
    dp = proc->root_directory;
    path++;
  } else dp = proc->current_working_directory;
  while (*path) {
    name = path から次の/までの部分を取り出す;
    dp = lookup(dp, name);
  }
  return dp;
}
```

ディレクトリ内でファイルを探す

# ファイルの実装方法

- ファイルはディスク上の複数のブロックから構成
  - ファイル毎に利用しているブロック番号を記録する必要がある
- UNIX では inode（index node）により管理
  - FATファイルシステムではFATが管理

inode構造体

```
struct inode {
  short mode;
  int uid
  int gid;
  long size;
  ...
  int atime;
  int mtime;
  int ctime;
  ...
  int direct[12];
  int indirect[3];
};
```

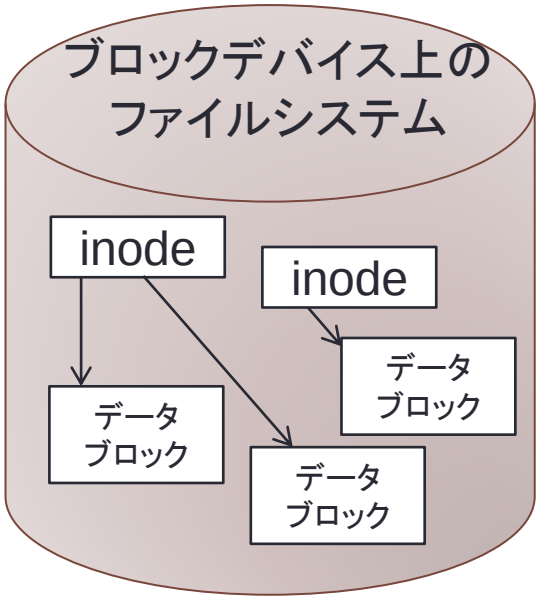
所有者

サイズ

変更日付

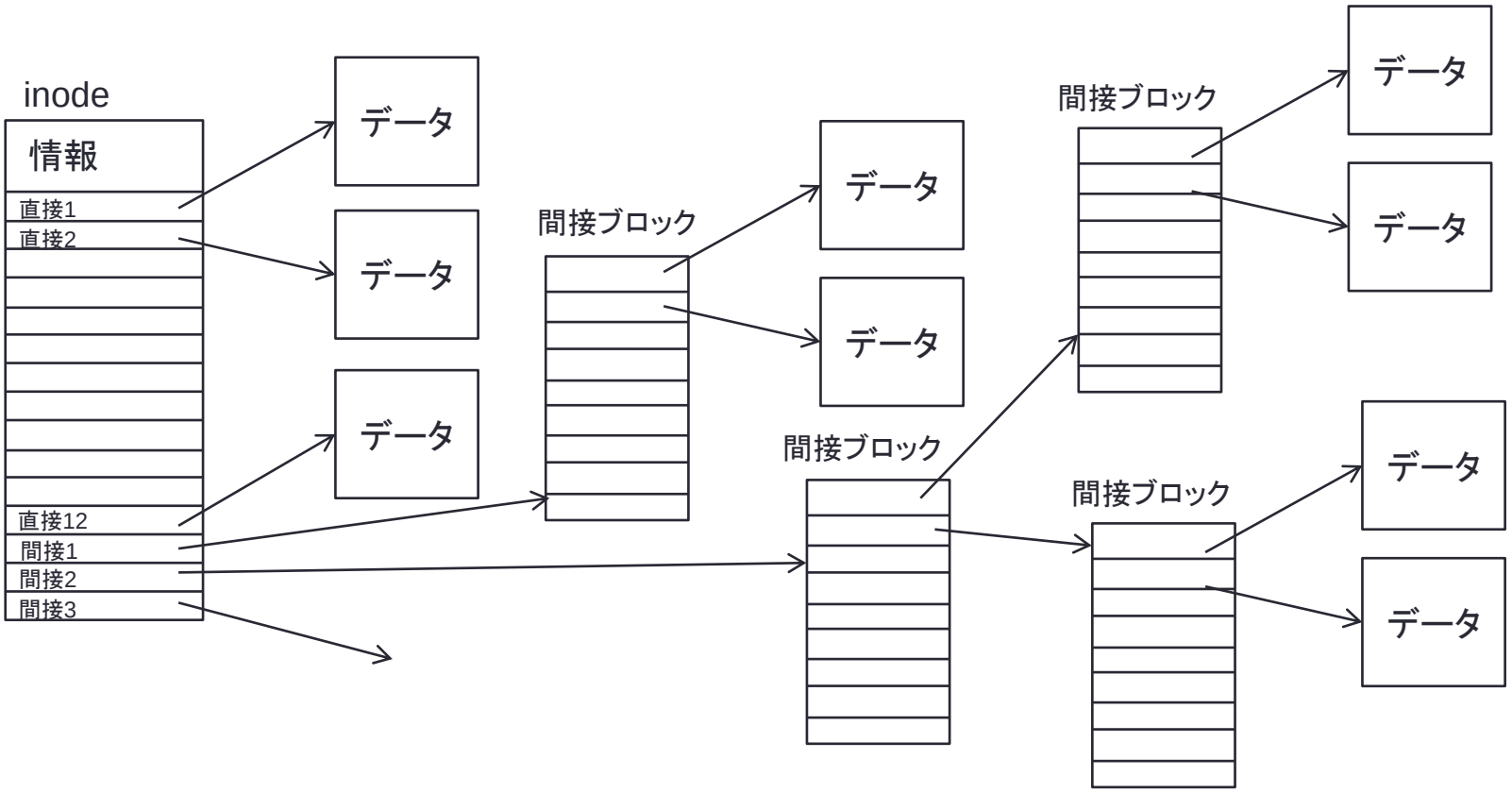
直接参照ブロックの番号（12個）

間接参照ブロック番号（3個）



# 直接参照と間接参照

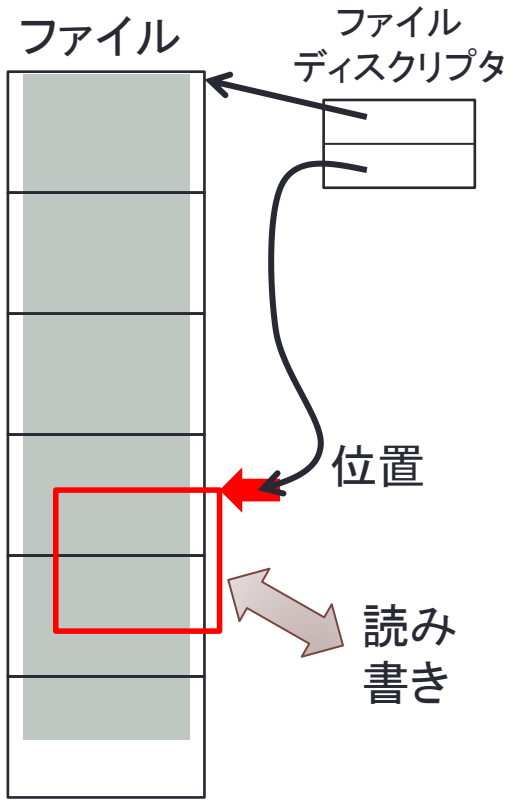
- inodeから直接参照できるブロックは12個のみ
  - 1ブロック512バイトの場合512×12=6KB以下のファイル
- 間接参照ではブロックの配列のブロックを記録



# ブロック番号の計算

- ファイルのオフセットからブロック番号を計算
  - read, writeではオフセットからブロック番号を計算し，そのブロックにアクセスする

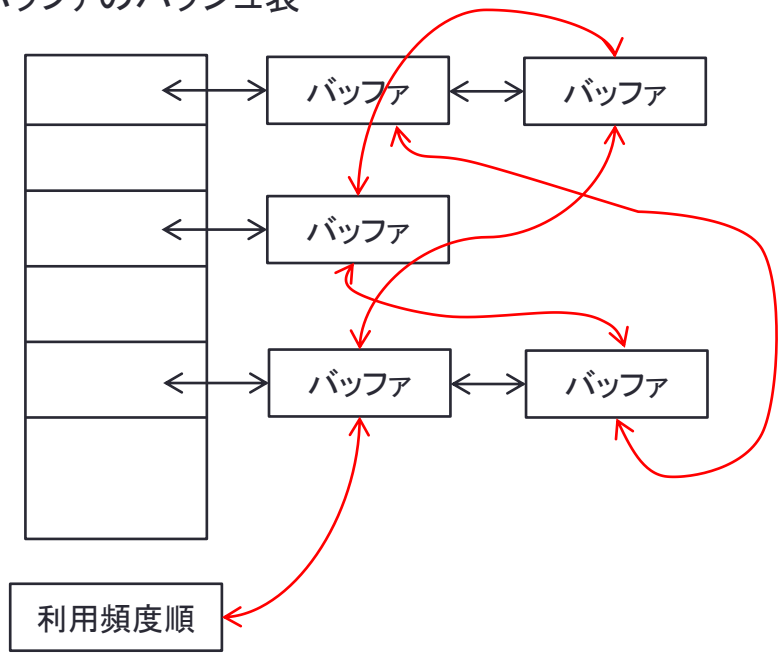
```
int balloc(struct inode *ip, long offset) {
   struct buf *bp;
   int i;
   blk = (offset + BLKSIZE - 1) / BLKSIZE;
   if (blk < 12) return ip->direct[blk];
   blk -= 12;
   blocks = BLKSIZE/sizeof(int);
   for (i = 0; i < 3; i++) {
      if (blk < blocks) break;
      blk -= blocks;
      blocks *= BLKSIZE/sizeof(int);
   }
   bp = getblock(ip->indirect[i])
   while (i-- > 0) {
      blocks /= BLKSIZE/sizeof(int);
      bp = getblock(bp->buf[blk/blocks]);
      blk %= blocks;
   }
   return bp->buf[blk];
}
```



# バッファリング

- ディスクブロックはOS内でキャッシュ（バッファリング）される
- 読み込み
  - 初回：ディスクからブロックを読み込みメモリ内にキャッシュする
  - 二回目以降：ディスクの代わりにキャッシュを読む
- 書き込み
  - すぐにはディスクに書き込まない
  - たまってから一挙に書き込む

バッファのハッシュ表

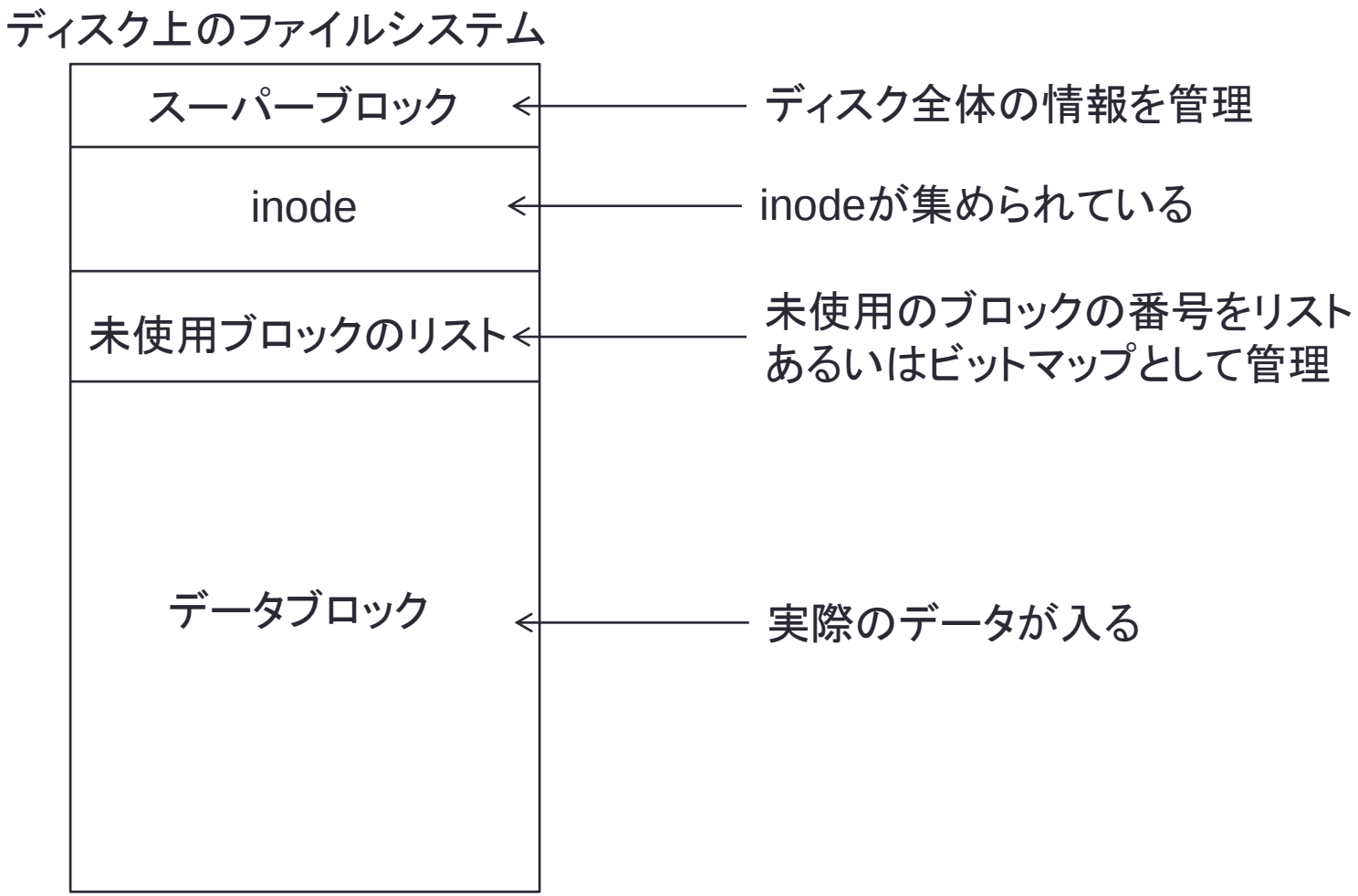



buf構造体

```
struct buf {
  struct buf *next, *prev;
  struct buf *queue_next, *queue_prev;
  struct inode *inode;
  int dirty;
  int blkno;
  char buf[BLKSIZE];
};
struct buf buffers[HASH];
```

# ディスクのフォーマット

- 新しいディスクを使うときにはフォーマット（初期化する）



# まとめ

- ファイルシステム
  - UNIXのファイルシステムを例
- システムコールと標準入出力ライブラリ
  - 標準入出力ライブラリの実装
  - ファイルディスクリプタ
- ファイルシステムの実装
  - inode
  - 直接参照ブロックと間接参照ブロック