#14
ファイルシステムのプログラミング

2022 年度 / プログラミング及び実習 **III**

角川裕次

龍谷大学　先端理工学部

もくじ

今回 (#14) の内容： シラバスでの該当部分

小テーマ: ファイルシステムのプログラミング

第 22 回： ファイルとディレクトリ
第 23 回： ファイルシステム
第 24 回： ファイル操作のシステムコール

重要概念リスト

- ファイルシステム
- ディレクトリ
- 高水準 I/O と低水準 I/O
- システムコール open/read/write/close/unlink/stat
- stat 構造体
- ファイルディスクプリタ
- ディレクトリアクセス関数群 opendir, readdir, closedir
- dirent 構造体

今回の実習・課題 (manaba へ提出)

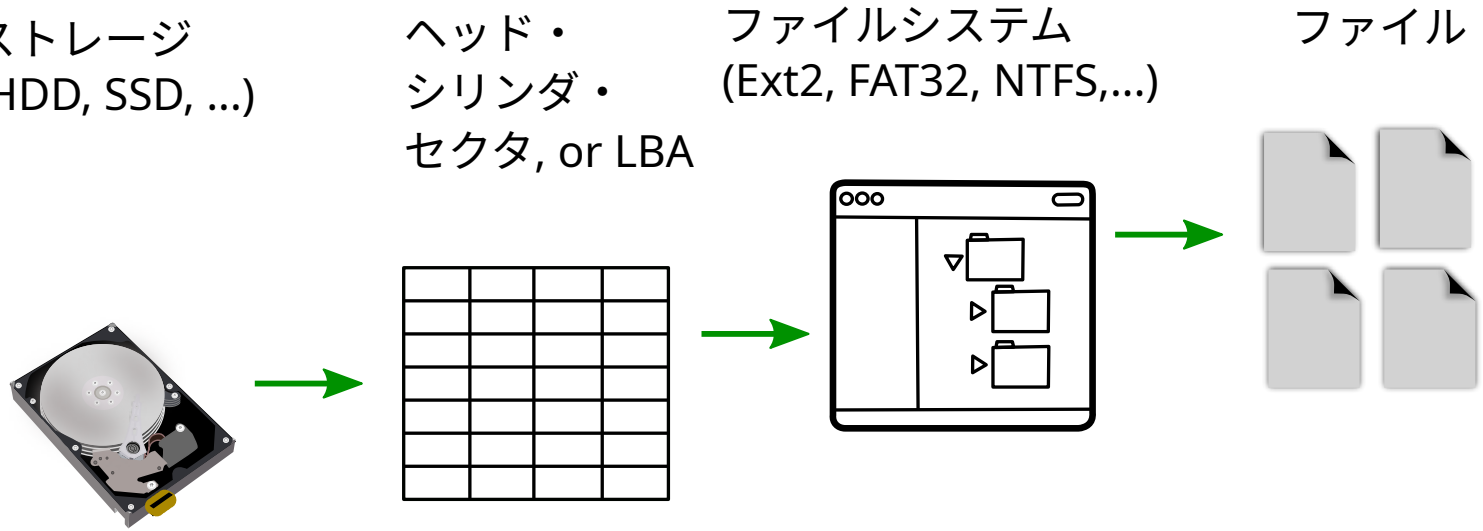
実習内容と課題内容は講義途中に提示します

(作成したファイル類は manaba に提出)

(独自) ファイルシステムとは

★重要★

ストレージ上のデータ抽象化・管理機構



ハードディスク等のストレージの特徴

- 512 バイト等のブロック単位で入出力
- アクセス対象ブロックの指定：
  ヘッド番号, シリンダ番号, セクタ番号の組
  あるいは Logical Block Address (LBA) によるセクタの通し番号

ファイルシステム：抽象化・管理機構・アクセス法を提供

- 直接ハードディスクを扱う困難さを解決

ファイルシステムの機能

ユーザに見せるもの

- ディレクトリ (フォルダ)
- ファイル

プログラミングインターフェース (API)

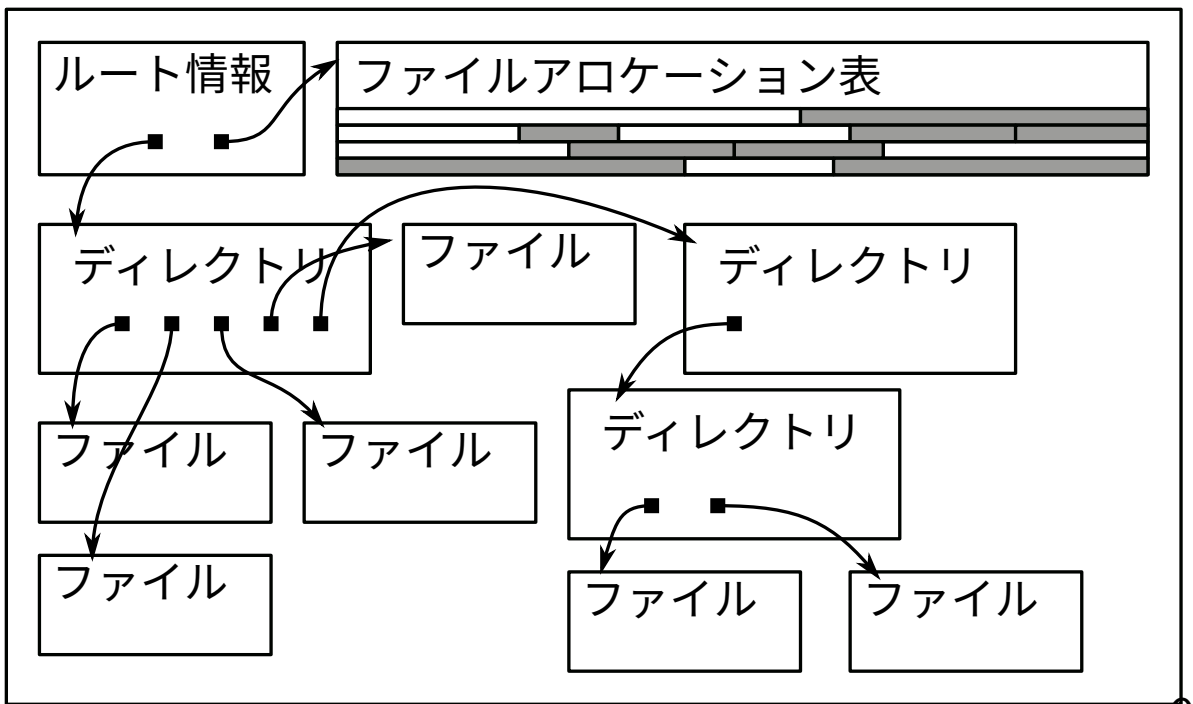
- ファイル：open, close, read, write
- ディレクトリ：内容一覧, ファイル作成, ファイル削除,

データ保全

- 不整合の検出・修復
- データ多重化・ジャーナリングによる故障耐性
- ロック機構によるアトミック更新

ファイルシステム：ストレージ上の巨大なデータ構造

- ルート情報：管理情報や最上位ディレクトリへのリンク等
- ファイルアロケーション表 (FAT)：使用/未使用領域を記憶
- ディレクトリ：ファイル/ディレクトリへのリンク表
- ファイル：ユーザーが利用するデータ本体



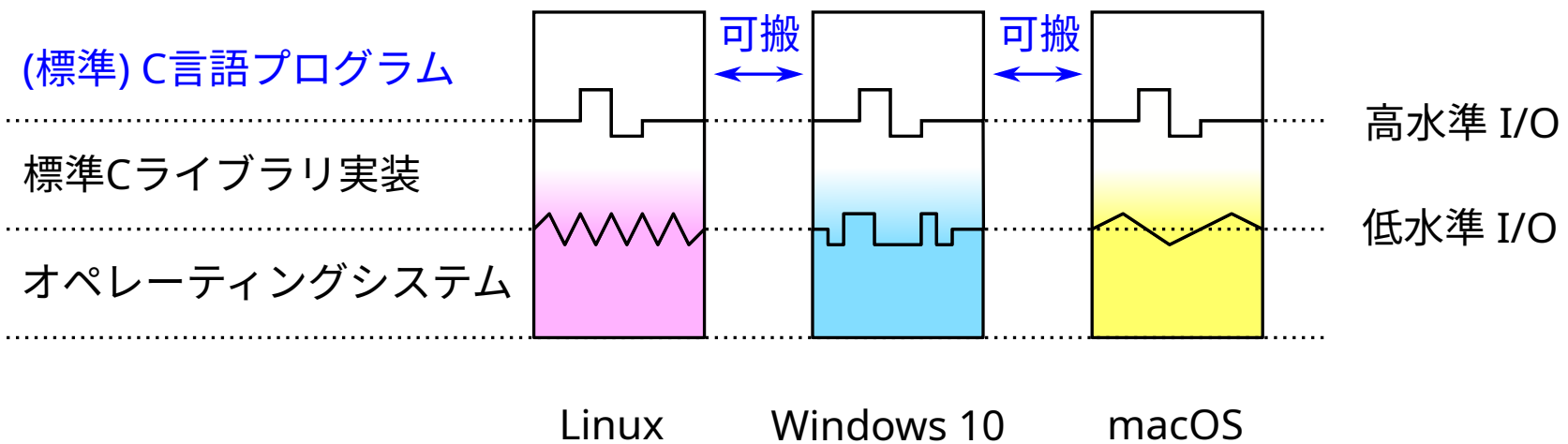
(独自) ファイル操作のシステムコール

低水準/高水準ファイル I/O

★重要★

高水準ファイル I/O：C 言語標準ライブラリ
- 可搬性あり：OS の違いを隠蔽する抽象インターフェースとその実装
- `FILE*`, `fopen`, `printf`, `fgets`, `fclose` など

低水準ファイル I/O：OS システムコール群
- 可搬性なし：OS ごとに関数群や使用方法が異なる
- `open`, `write`, `read`, `close` など (Linux/Unix 系の場合)



★重要★

Unix 系 OS (含 Linux/Ubuntu) の低水準ファイル I/O

※ Unix 系 OS 固有 (Windows/Mac の場合は分からないです．すみません)

ファイルデスクプリタ (file descripter, FD)

- オープンしたファイルを示す整数値
- 同時に複数のファイルをオープンした時の区別で重要
- ユーザープログラム側：read/write/close での対象ファイル指定
- OS 側：ファイルアクセス状態の管理

低水準ファイル I/O の主なシステムコール

- `open`：ファイルをオープン, FD が返される
- `read`：読み出し
- `write`：書き込み
- `close`：ファイルをクローズ
- `unlink`：ファイルを削除
- `stat`：ファイルの情報を取得

システムコール (system call) とは?

システムコール (system call)：OS の機能の呼び出し

- プログラムを書くときには関数のように呼び出せる
- 関数のように見えるが実は OS が激しく動作する

システムコール呼出を行うアセンブリ言語命令 (機械語命令)

- x86-64：`syscall` 命令
- MIPS：`syscall` 命令
- ARM：`swi` 命令

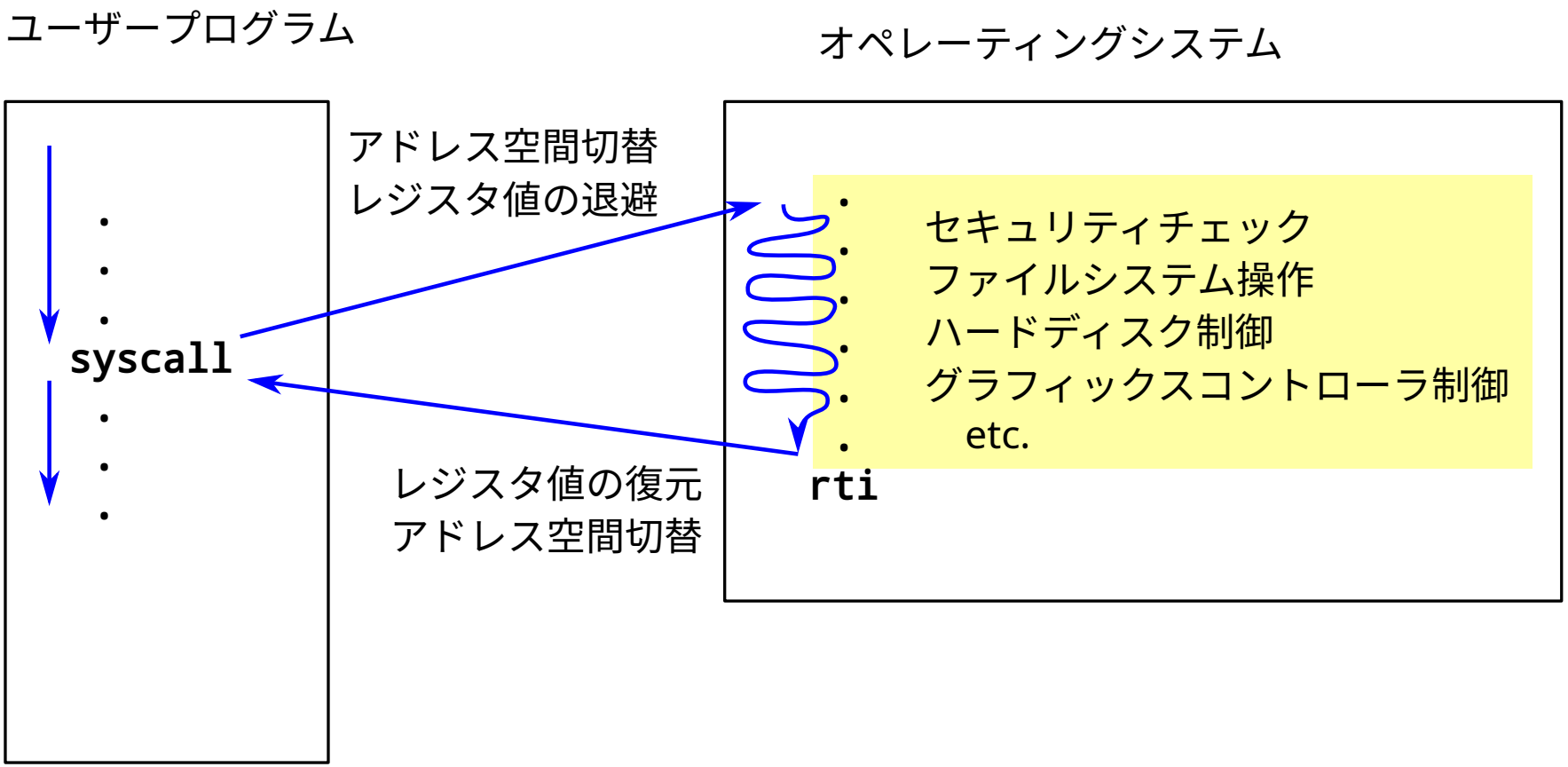
システムコール呼出命令を実行で起きること

1. ユーザープログラムの実行を一時中断して OS へ実行を移す
2. OS はパラメータを解析してファイル I/O 等の動作を行う
   (すごく時間を要する; アクセス権チェック, ハードウエアの動作など)
3. ユーザープログラムの実行へ復帰

システムコールの実行で起きること

実行は OS にいったん移って様々な複雑な動作がいっぱい起きる

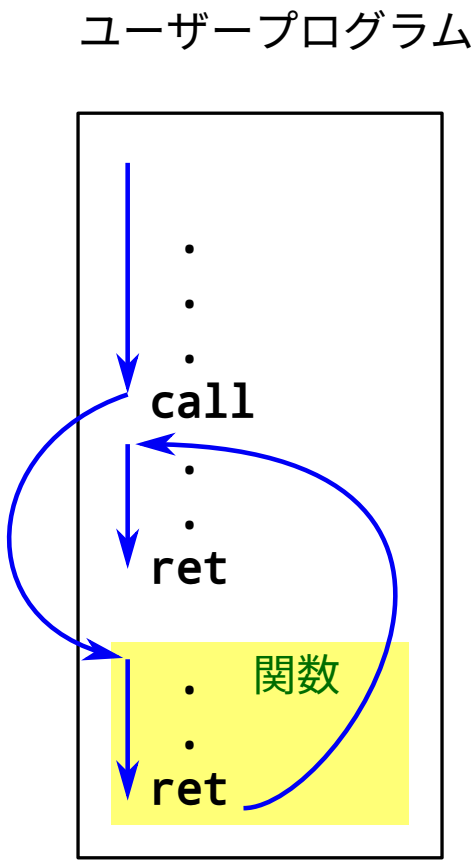
- かなりの実行時間がかかる



関数呼び出しの実行で起きること

実行はユーザープログラムの中だけで完結

- 実行時間はあまりかからない



基礎コードパターン

★重要★

コードの全体像

```
/* オープン */
int fd = open(pathname, 【アクセス指定フラグ】);

/* ファイルへの読み出しまたは書き込み */
 【read()/write() 呼出】

/* クローズ */
close(fd);
```

読み込みコードパターン

★重要★

```
int fd = open(pathname, O_RDONLY);
if (fd < 0) {
    perror("open");
    exit(1);
}
for (;;) {
  char buff[BUFSIZ];
  int n = read(fd, buff, sizeof(buff));
  if (n < 0) {
    break;
  }
   【 読み込み内容を処理 】
}
close(fd);
```

書き込みコードパターン

★重要★

```
int fd = open(pathname, O_WRONLY);
if (fd < 0) {
    perror("open");
    exit(1);
}
int n = write(fd, buff, datasize);
if (n < 0) {
    perror("write");
    exit(1);
}
close(fd);
```

通常はこの書き方で OK (だが完全ではない; 詳しくは後述)

open システムコール (形式 1)

ファイルをオープン：open

関数 `int open(const char *pathname, int flags);`
ファイルをオープンしてファイルディスクプリタを返す

引数

- `pathname`：アクセス対象のファイル名 (文字列)
- `flags`：オープン時のアクセス法の指定 (後述)

返却値

- オープンしたファイルディスクプリタ
- -1：エラーの場合

ヘッダ：`#include <sys/types.h>`
ヘッダ：`#include <sys/stat.h>`
ヘッダ：`#include <sys/fcntl.h>`

オープン時のアクセス法の指定 flags

アクセス法の指定

- O_RDONLY
  読み出し限定 (Read-Only) でオープン
- O_WRONLY
  書き込み限定 (Write-Only) でオープン
- O_RDWR
  読み・書き可能 (Read/Write) でオープン

アクセス法の追加指定 (ビットごとの論理和で追加指定が可)

- O_APPEND
  ファイル末尾に書き込み位置を設定
- O_CREAT
  新規作成 (追加の引数が必要; 形式 2 を見よ)
- O_TRUNC
  書き込みオープンの際にファイルの長さを 0 に切り詰める

例

- `fd = open("log.txt", O_WRONLY|O_TRUNC);`

open システムコール (形式 2; flag に O_CREAT 指定時)

ファイルをオープン：open

関数

```
int open(const char *pathname, int flags, mode_t mode);
```

ファイルをオープンしてファイルディスクプリタを返す

引数

- pathname：アクセス対象のファイル名 (文字列)
- flags：オープン時のアクセス法の指定
- mode：ファイル新規作成時のモード指定 (後述)

返却値

- オープンしたファイルディスクプリタ
- -1：エラーの場合

ヘッダ：`#include <sys/types.h>`
ヘッダ：`#include <sys/stat.h>`
ヘッダ：`#include <sys/fcntl.h>`

ファイル新規作成時のモード指定 mode

作成されるファイルのパーミッションを指定 (詳細は省略)

- S_IRUSR — 所有者 (user) にリード許可 (read)
- S_IWUSR — 所有者 (user) にライト許可 (write)
- S_IXUSR — 所有者 (user) に実行許可 (execute)
- S_IRGRP — 同一グループ (group) のユーザにリード許可 (read)
- S_IWGRP — 同一グループ (group) のユーザにライト許可 (write)
- S_IXGRP — 同一グループ (group) のユーザに実行許可 (execute)
- S_IROTH — 他ユーザ (other) にリード許可 (read)
- S_IWOTH — 他ユーザ (other) にライト許可 (write)
- S_IXOTH — 他ユーザ (other) に実行許可 (execute)

※ビットごとの論理和で複数の同時指定が可能

例：書き込み, 新規作成, 所有者 (自分) は読み書き実行可能

```
int fd = open("f.txt", O_WRITE|O_CREAT,
               S_IRUSR|S_IWUSR|S_IXUSR);
```

creat システムコール

ファイルの新規作成：creat

関数 `int creat(const char *pathname, mode_t mode);`
ファイルを作成・オープンしてファイルディスクプリタを返す

`creat(pathname, mode)` は
`open(pathname, O_CREAT|O_WRONLY|O_TRUNC, mode)` と同一

ヘッダ：`#include <sys/types.h>`
ヘッダ：`#include <sys/stat.h>`
ヘッダ：`#include <sys/fcntl.h>`

create ではなくて creat です

open : システムコール? 関数?

Q. 関数と同じに見えるけど本当にシステムコールなの?

A. 関数の形で呼び出せる仕組みにしてあるよ

- open 関数の内部でシステムコール呼出しを実行
- ソースコードレベルでは関数/システムコールの違いの意識は不要

ライブラリ関数のソースコードはたぶんこう (憶測)

- open 関数を作ってその中でシステムコール呼出

```
int open(...) {
  システムコールパラメータをレジスタに設定;
  syscall命令;
  return 実行結果のファイルディスクプリタ;
}
```

Q. 高水準 I/O の fopen はどうなっているの?

A. ライブラリ関数のソースコードはたぶんこう (憶測)

- fopen 関数を作ってその中で open 関数を呼出
- open 関数の中でシステムコールが呼び出される

```
FILE *fopen(...) {
  FILE *fp = FILE構造体を確保;
  int fd = open(...);
  fpに値を適切に設定
  return fp;
}
```

close システムコール

ファイルをクローズする：close

関数 `int close(int fd);`

引数

- `fd`：クローズ対象のファイルディスクプリタ

返却値

- 0：クローズ成功
- -1：クローズ失敗

ヘッダ：`#include <unistd.h>`

read システムコール

ファイルを読み出す：read

関数 `int read(int fd, void *buf, size_t count);`
ファイルから `count` バイトを読み出して `buf` へ格納

引数

- `fd`：読み出し対象のファイルディスクプリタ
- `buf`：ファイル内容の格納先を示すポインタ
- `count`：読み出すバイト数

返却値

- 読み出しに成功したバイト数
- 0：ファイル終端の場合
- -1：エラーが発生した場合

ヘッダ：`#include <unistd.h>`

write システムコール

ファイルへ書き込む：write

関数 `ssize_t read(int fd, const void *buf, size_t count);`
`buf` から `count` バイトをファイルへ書き込む

引数

- `fd`：書き込み対象のファイルディスクプリタ
- `buf`：書き込み内容の格納先を示すポインタ
- `count`：書き込むバイト数

返却値

- 書き込みに成功したバイト数
- 0：全く書き込みがなされなかった場合
- -1：エラーが発生した場合

ヘッダ：`#include <unistd.h>`

書き込みコードパターン (その 2)

一度の write 呼出で全てが書かれない場合への対応策

- 特に書き込み先がネットワーク越しの場合に起きることあり

```
int fd = open(pathname, O_WRONLY);
if (fd < 0) {
    perror("open");
    exit(1);
}
int done = 0;
while (done < datasize) {
  int n = write(fd, &buff[done], datasize-done);
  if (n < 0) {
    perror("write");
    exit(1);
  }
  done += n;
}
close(fd);
```

書き残しがなくなるまで残り部分の書き込みを繰り返す

- write 呼出で実際に書かれたバイト数を勘定

unlink システムコール

通常ファイルをディレクトリエントリから削除：unlink

関数 `int unlink(const char *pathname);`
対象ファイルのリンクカウントを 1 減らす

- リンクカウントが 0 になれば実際にファイルを削除
- (そうれなければファイルはまだ削除されない)

引数

- `pathname`：削除対象のファイル
  ※ディレクトリは対象に出来ない (通常ファイルに限定)

返却値

- 0：成功した場合
- -1：エラーが発生した場合

ヘッダ：`#include <unistd.h>`

stat システムコール

ファイルの情報を取得：stat

関数 `int stat(const char *pathname, struct stat *statbuf);`

引数

- `pathname`：削除対象のファイル
- `statbuf`：情報の格納先 (`stat` 構造体へのポインタ; 後述)

返却値

- 0：成功した場合
- -1：エラーが発生した場合

ヘッダ：`#include <sys/types.h>`
ヘッダ：`#include <sys/stat.h>`
ヘッダ：`#include <unistd.h>`

stat 構造体

stat システムコールによるファイル情報の格納で用いる
構造体のメンバ (主要なもののみを抜粋)

```
struct stat {
  mode_t   st_mode;  /* ファイルタイプとモード */
  nlink_t  st_nlink; /* ハードリンク数 */
  uid_t    st_uid;   /* ファイル所有者のユーザ ID */
  gid_t    st_gid;   /* ファイル所有者のグループ ID */
  off_t    st_size;  /* ファイルサイズ(バイト数) */
  struct timespec st_atim; /* 最終アクセス時間 */
  struct timespec st_mtim; /* 最終更新時間 */
  struct timespec st_ctim; /* 最終ステータス変化時間 */
};
```

メンバ st_mode の値 (主なもの)

- S_IFREG — 通常ファイル
- S_IFDIR — ディレクトリ
- S_IFMT — 値の取り出し用のビットマスクパターン

ファイル種別の区別を表示する例

通常ファイル/ディレクトリかの 2 種を判別

```
void print_file_type(char *pathname) {
  struct stat sb;
  stat(pathname, &sb);
  switch (sb.st_mode & S_IFMT) {
  case S_IFREG: printf("regular file\n"); break;
  case S_IFDIR: printf("directory\n");    break;
  default:      printf("other file\n");   break;
  }
}
```

通常ファイル: 以下のファイルなどが該当

- PDF ファイル (.pdf)
- ワープロ文書 (.docx), 表計算ファイル (.xlsx)
- 画像ファイル (.jpeg, .png, .gif)
- テキストファイル (.txt)
- C 言語ソースファイル (.c, .h)
- Python ソースファイル (.py)
- 実行バイナリファイル

fstatus.c： ファイル情報を表示する

fstatus.c : ファイル情報を表示する

指定ファイルの情報を stat システムコールで入手 & 表示

表示する情報

- ファイルタイプ (通常ファイル/ディレクトリ/他)
- 所有者のユーザ ID とグループ ID
- ファイルサイズ
- 最終アクセス時間
- 最終更新時間

実行例 (1)

ファイル名の指定に dirlist.c を入力した場合

```
$ ./fstatus
ファイル：dirlist.c
ファイルタイプ： 通常ファイル
所有者のユーザID： 1001
所有者のループID： 1001
ファイルサイズ　　： 1197 バイト
最終アクセス時間 ： Thu Jul  1 02:33:16 2021
最終更新時間　　　： Tue Nov 24 08:14:34 2020
```

ファイル名の指定に xxx を入力した場合

```
$ ./fstatus
ファイル：xxx
NO SUCH FILE: xxx
```

※ 対象ファイルの有無や表示内容：ひとりひとり違う

実行例 (2)

ファイル名の指定に /home を入力した場合

```
$ ./fstatus
ファイル：/home
ファイルタイプ：ディレクトリ
所有者のユーザID：0
所有者のループID：0
ファイルサイズ　　：4096 バイト
最終アクセス時間：Wed Jun 30 23:18:46 2021
最終更新時間　　：Sat May  8 11:24:39 2021
```

ファイル名の指定に /dev/null を入力した場合

```
$ ./fstatus
ファイル：/dev/null
ファイルタイプ：その他
所有者のユーザID：0
所有者のループID：0
ファイルサイズ　　：0 バイト
最終アクセス時間：Mon Jun 21 19:12:35 2021
最終更新時間　　：Mon Jun 21 19:12:35 2021
```

※ 対象ファイルの有無や表示内容：ひとりひとり違う

ソースコード fstatus.c (1/4) : ヘッダ, 定数, プロトタイプ宣言

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <sys/types.h>
#include <dirent.h>
#include <sys/stat.h>

#define MAX_FNAME  1024
void get_filename(char *fname, int n);
void fstatus(char *fname);
```

ソースコード fstatus.c つづき (2/4)
main 関数

```
/* main 関数 */
int main(void) {
  char fname[MAX_FNAME];
  /* ファイル名の読み込み */
  get_filename(fname, MAX_FNAME);
  /* 指定ファイルの情報を表示 */
  fstatus(fname);
  return 0;
}
```

ソースコード fstatus.c つづき (3/4)
get_filename 関数 — ファイル名の読み込み

```
/* ファイル名の読み込み */
void get_filename(char *fname, int n) {
  /* プロンプト表示 */
  printf("ファイル:");
  /* ファイル名読み込み */
  if (fgets(fname, n, stdin) == NULL) {
    exit(1);
  }
  /* 行末の '\n' を削除 */
  char *p = strchr(fname, '\n');
  if (p != NULL) {
    *p = '\0';
  }
}
```

ソースコード fstatus.c つづき (4/4)
fstatus 関数 — ファイル情報の表示

```
/* 指定ファイルの情報を表示 */
void fstatus(char *fname) {
  struct stat sb;
  if (stat(fname, &sb) < 0) {
    printf("NO SUCH FILE: %s\n", fname);
    return;
  }
  /* 情報を表示 */
  printf("ファイルタイプ : ");
  switch (sb.st_mode & S_IFMT) {
  case S_IFREG : printf("通常ファイル\n"); break;
  case S_IFDIR : printf("ディレクトリ\n"); break;
  default:       printf("その他\n");         break;
  }
  printf("所有者のユーザID : %d\n", sb.st_uid);
  printf("所有者のループID : %d\n", sb.st_gid);
  printf("ファイルサイズ   : %ld バイト\n", sb.st_size);
  printf("最終アクセス時間 : %s", ctime(&sb.st_atime));
  printf("最終更新時間     : %s", ctime(&sb.st_mtime));
}
```

関数 ctime() : 日時を文字列に変換

(独自) ファイルとディレクトリ

ディレクトリ内容の読み出し

ディレクトリ：ファイルの一種だがずいぶん特殊

- ファイルシステム固有のデータフォーマット

自力でやるのはかなり面倒

アクセスに便利な関数群あり：これらを使うのが楽

- `opendir()` — ディレクトリをオープン
- `readdir()` — ディレクトリ中の次の項目を得る
- `closedir()` — ディレクトリをクローズ

★重要★

ディレクトリアクセス関数群：*XXX*dir()

ディレクトリの最初から１つづつ順にエントリを読み出してゆく

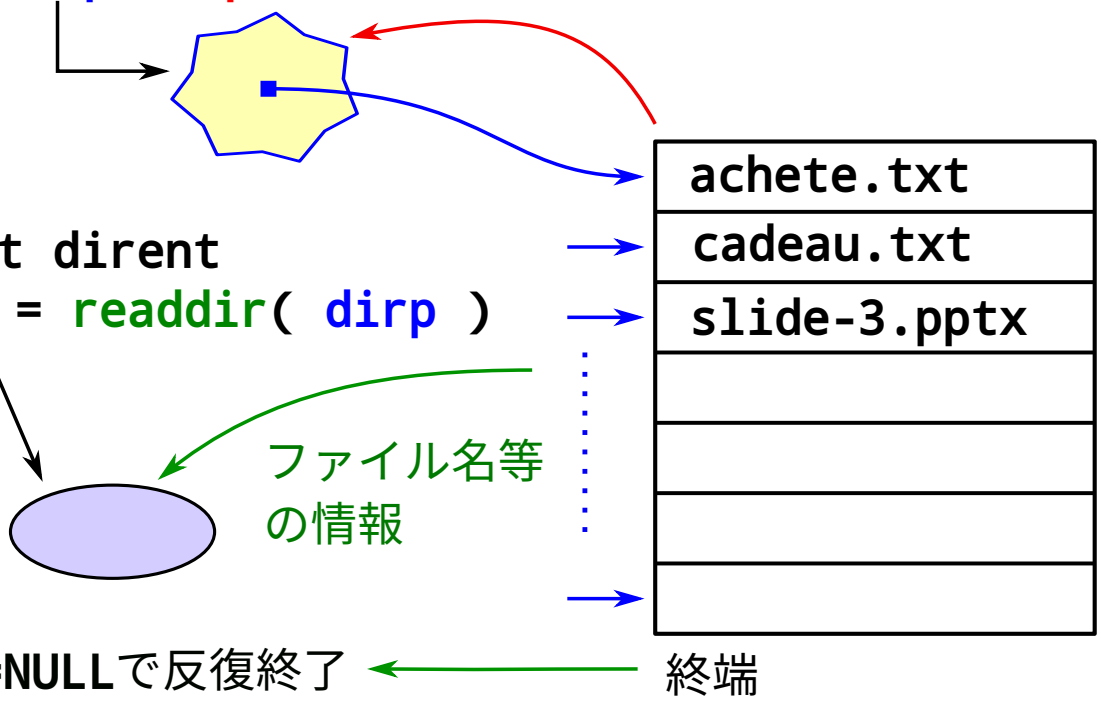
- エントリ：１つのファイルに関する情報 (ファイル名等)

1. オープン

```
DIR *dirp = opendir( directory name )
```

2. リード（反復）

```
struct dirent
  *de = readdir( dirp )
```



3. クローズ

```
closedir( dirp )
```

ディレクトリの内容

エントリ (ディレクトリエントリ; directory entry) のリスト

- エントリ：1 つのファイルに関する情報
- ファイル名, ファイルの種別など

アクセス法：エントリを先頭からひとつづつ順に読み出してゆく

- 配列のようにはアクセスできないのが普通
- `readdir()` 関数を使用

ディレクトリ内の珍しいエントリ

`..` (ピリオド 2 つ)

- `..`: 親ディレクトリ
- 親ディレクトリの指定で必要
- 例：`cd ..` の実行

`.` (ピリオド 1 つ)

- `.`: このディレクトリ自体
- 現ディレクトリの指定で必要
- 例：`./prog` の実行

ディレクトリ構造体

ディレクトリ構造体 struct dirent

ディレクトリエントリの情報を保持する構造体

```
struct dirent {    /*一部のメンバーのみ*/
  unsigned char d_type;
  char d_name[256];
};
```

メンバ d_type : ファイル種別 (OS/環境依存)

- DT_DIR : ディレクトリ
- DT_REG : 通常ファイル
- 他にもあり (省略)

メンバ d_name : エントリ名 (ファイル/ディレクトリ名) の文字列

ヘッダ : #include <sys/types.h>
ヘッダ : #include <dirent.h>

ディレクトリのオープン

ディレクトリのオープン：opendir

```
DIR *opendir (const char *name);
```

ディレクトリをオープンする

- ディレクトリエントリの読み出し位置をディレクトリの最初に設定

引数

- `name`：オープン対象のディレクトリ (文字列)

ヘッダ：`#include <sys/types.h>`
ヘッダ：`#include <dirent.h>`

ディレクトリのクローズ

ディレクトリのクローズ：closedir

```
int closedir (DIR *dirp);
```

引数

- `dirp`：クローズの対象

ヘッダ：`#include <sys/types.h>`
ヘッダ：`#include <dirent.h>`

ディレクトリエントリの読み出し

ディレクトリエントリの読み出し：readdir

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

現在の読み出し位置のディレクトリエントリを読み出す

引数

- `dirp`：読み出し対象

ヘッダ：`#include <sys/types.h>`
ヘッダ：`#include <dirent.h>`

コードパターン

```
/*ディレクトリファイルをオープン*/
DIR *dir = opendir(【対象ディレクトリ名文字列】);
if (dir == NULL){ /*オープン失敗*/
  perror("opnedir");
  exit(1);
}
/*ディレクトリエントリを順次リード*/
for (;;) {
  struct dirent *de = readdir(dir);
  if (de == NULL) { /*終わり*/
    break;
  }
   【de内の情報へアクセス】
}
/*ディレクトリファイルをクローズ*/
closedir(dir);
```

files.c：カレントディレクトリのファイル一覧

実行例

```
$ ./files
Makefile
..
.
files.c
files
```

注意：ソートされているとは限らない (ソートされていないのが普通)

補足：シェル上での `ls -a -f` コマンドの実行と同等

- `-a` オプション：`.` で始まるファイルも表示
- `-f` オプション：ソートせずに表示

```
$ ls -fa
Makefile   ..   .   files.c   files
```

files.c ソースコード (1/3)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <dirent.h>

void scan_dir(char *dirname);
void print_dirent(struct dirent *de);

/*カレントディレクトリの内容一覧を表示*/
int main(void) {
  scan_dir(".");
  return 0;
}
```

files.c ソースコード (2/2)

ディレクトリの先頭のエントリから順次表示

```
/*指定ディレクトリをスキャンして表示関数を呼ぶ*/
void scan_dir(char *dirname) {
  /*ディレクトリファイルをオープン*/
  DIR *dirp = opendir(dirname);
  if (dirp == NULL){ /*オープン失敗*/
    perror("opnedir");
    exit(1);
  }
  /*ディレクトリエントリを順次リード*/
  for (;;) {
    struct dirent *de = readdir(dirp);
    if (de == NULL) { /*終わり*/
      break;
    }
    /*ディレクトリエントリを表示*/
    print_dirent(de);
  }
  /*ディレクトリファイルをクローズ*/
  closedir(dirp);
}
```

files.c ソースコード (3/3)

```
/*ディレクトリエントリを表示*/
void print_dirent(struct dirent *de)
{
  printf("%s\n", de->d_name);
}
```

単に d_name (ファイル/ディレクトリ名) を表示するだけ

各ファイルの他の情報の表示も簡単に実現できますよね

dirlist.c： ファイル一覧 (種別付き)

dirlist.c：ファイル一覧を種別付きで表示

機能：ファイル種類 (通常ファイル / ディレクトリ) も表示

- 通常ファイル：行頭に F を表示
- ディレクトリ：行頭に D を表示

実行例

```
$ ./dirlist
F Makefile
D ..
D .
F dirlist.c
F files.c
F files
F dirlist
```

dirlist.c：実装方針

以下は files.c と同じ

- `main` 関数
- `scan_dir` 関数

ディレクトリエントリの表示部分 (`print_dirent` 関数) だけを変更

ファイル種別：ディレクトリ構造体 `struct dirent` のメンバ `d_type`

- `DT_REG`：通常ファイル
- `DT_DIR`：ディレクトリ

print_dirent 関数の実装

```
/*ディレクトリエントリを表示*/
void print_dirent(struct dirent *de)
{
  switch (de->d_type) { /*種別*/
  case DT_REG: /*通常ファイル*/
    printf("F %s\n", de->d_name);
    break;
  case DT_DIR: /*ディレクトリ*/
    printf("D %s\n", de->d_name);
    break;
  default:     /*他*/
    printf("? %s\n", de->d_name);
    break;
  }
}
```

おわり

番外編の課題 1

ディレクトリ内容の入れ子を表示するプログラムの作成

```
$ xtree
.
|-- Ex
|   |-- hello.c
|   `-- x-11.c
|-- Figs
|   |-- Makefile
|   |-- chap00-intro
|   |   |-- book-clang-shibata.jpg
|   |   `-- expl-wsl.PNG
|   |-- chap00a-arch
|   |   |-- cpu-mem-01-overview.svg
|   |   `-- exec-var-i1c-func.svg
|   |-- chap00f-filesys
|   |   |-- call-1-syscall.pdf
|   |   |-- call-1-syscall.svg
|-- Makefile
|-- pg3-01-arch.tex
|-- pg3-02-func.tex
`-- tex2pdf.sh
```

番外編の課題 2

指定ファイルを再帰的に探してどこにあるかを表示するプログラムを作成

```
$ xfind
ファイル名: call-1-syscall.svg
./Figs/chap00f-filesys/call-1-syscall.svg
```

番外編の課題 3

指定のファイルの内容をすべてメモリに読み込む関数の作成

- `int read_file_to_buffer(char *f, struct buffer *bp);`
- 読み込み成功時に 0, 失敗時に-1 を返す
- 読み込み対象のファイル名は `f`
- 実際のファイル内容とファイルサイズは引数の構造体変数に記録
- 読み込みのための構造体

```
struct buffer {
  unsigned char *data; /* ファイル内容 */
  size_t         size; /* ファイルサイズ */
};
```

テキストエディタ構造概略

- `int cursor` でカーソル位置を定義
- `bp->data[cursor]` 付近の文字データを画面上に描画
- キーボード操作により `bp->data[cursor]` を変更して再描画
- 文字挿入、文字削除、行削除など...