

# UNIXシステムプログラミング

## 第5回 ファイル関連のシステムコール

2015年10月23日

情報工学科

寺岡文男

# 今日のねらい

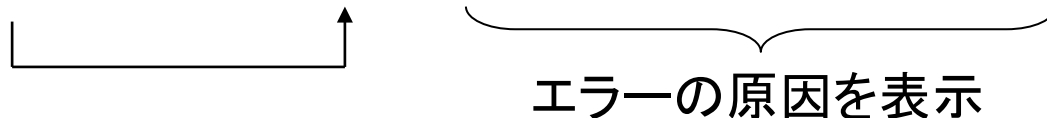
- ファイル関連の主なシステムコールの使い方を学ぶ
  - open( ), read( ), write( ), close( ), lseek( )
  - stat( ), fstat( )
  - flock( )
  - opendir( ), readdir( ), closedir( ) (ライブラリ関数)
  - fopen( ), fread( ), fwrite( ), fclose( ) (ライブラリ関数)
- エラーの扱い方を学ぶ
  - int errno
  - perror( )
- 演習
  - cp コマンドの簡易版である mycp を作る

# システムコール

- プログラム上の見た目はライブラリ関数と同じ
- システムコール:カーネルの機能を直接呼び出す
  - 機能がより基本的
  - 細かい制御ができる
- エラーについてより一層注意が必要
  - エラーが発生することを念頭に置いてプログラミングする  
(ライブラリ関数でも同様だが)
  - 特に, システムプログラムではエラーが発生してもプログラムが停止したり誤動作しないようにすることが重要

# errno と perror( )

- システムコールはエラー時には `-1` や `NULL` を返す
  - これだけではエラーの原因が分からない
- 外部変数として “**int errno**” が定義されており, システムコールのエラーの原因を示す整数が設定される
  - システムコールがエラーの場合, 直後に `errno` の値を調べれば原因が分かる
  - “`#include <errno.h>`” が必要
- エラーの原因を表示したいときには **perror( )** を利用
  - `void perror(char *string);`
  - e.g., `perror(“open”);` → “open: no such file or directory”

エラーの原因を表示

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
```

```
main( )
{
```

```
    int fd;
    char fname[80];
```

```
    // fname[]へのファイル名の設定
```

```
    if ((fd = open(fname, O_RDWR | O_CREAT | O_EXCL, 0644)) < 0) {
```

```
        if (errno != EEXIST) {
```

```
            perror("open");
```

```
            exit(1);
```

```
        }
```

```
    // ファイルが既に存在していた → 上書きの確認
```

```
    if ((fd = open(fname, O_RDWR | O_CREAT | O_TRUNC, 0644)) < 0) {
```

```
        perror("open");
```

```
        exit(-1);
```

```
    }
```

```
}
```

```
// 正常な処理の続き
```

## open() の例

- ファイルを作成、存在していたらエラー
- 存在していたら上書きしていいかを確認
- OKなら既存ファイルの内容を消去

エラーの判定

エラーの原因を判別

エラーの原因を表示

このような書き方に  
注意(条件式の中の  
代入文)

# open( ) : ファイルのオープン、作成

```
#include <fcntl.h>
```

```
int
```

```
open(char *path, int flags, ... );
```

ファイル名を指定

ファイルを作成する場合,  
パーミッションを指定

O\_RDONLY

read-onlyでオープン

O\_WRONLY

write-onlyでオープン

O\_RDWR

read/writeでオープン

O\_APPEND

ファイルの最後にデータを追加

O\_CREAT

ファイルが存在しない場合は作成

O\_TRUNC

ファイルのサイズを0にする

O\_EXCL

既存のファイルにO\_CREATを指定するとエラー

# ファイル記述子 (file descriptor)

- `open( )`は正常の場合, **ファイル記述子**を返す
  - ファイル記述子は整数
  - ファイル記述子の変数として “`int fd;`” がよく使われる
- 以降のファイルに対する操作では, 対象となるファイルをファイル記述子で指定する.
  - `read( ), write( ), close( )`
  - `lseek( )`
  - `fstat( )`
  - `flock( )`

# read( ) : データの読み込み

```
size_t  
read(int d, void *buf, size_t nbytes);
```

- **d** : ファイル記述子
- **buf** : 読込んだデータを格納するメモリオブジェクトへのポインタ  
必ずメモリオブジェクトを指していること
- **nbytes** : 読み込むバイト数を指定
- 返り値:
  - 正常 : 実際に読込んだバイト数
  - ファイルの終わり : 0
  - エラー : -1



# write( ) : データの書き込み

```
size_t  
write(int d, void *buf, size_t nbytes);
```

- **d** : ファイル記述子
- **buf** : 書き込むデータを格納するメモリオブジェクトへのポインタ  
必ずメモリオブジェクトを指していること
- **nbytes** : 書き込むバイト数を指定
- 返り値:
  - 正常 : 実際に書き込んだバイト数
  - エラー : -1

# close( ) : ファイルのクローズ

```
int  
close(int d);
```

- **d** : ファイル記述子
- プロセス終了時にはすべてのオープンされているファイルはクローズされる
- しかし、ファイル操作が済んだら close( ) すべき
  - 同時にオープンできるファイル数には上限がある

# lseek() : 読み込み/書き込み位置の変更

```
#include <unistd.h>
```

```
off_t
```

```
lseek(int d, off_t offset, int whence);
```

- **d** : ファイル記述子

オフセット : read/writeの位置

- **offset** : 変更するバイト数

- **whence** : 変更の基準点

**SEEK\_SET** : オフセット = offset

**SEEK\_CUR** : オフセット = 現在の位置 + offset

**SEEK\_END** : オフセット = ファイルの末尾 + offset

負の値でも可



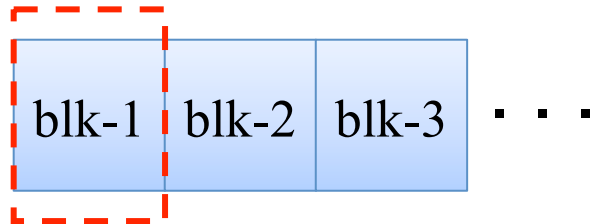
- 返り値

- 正常 : 変更後のオフセット
- エラー : -1

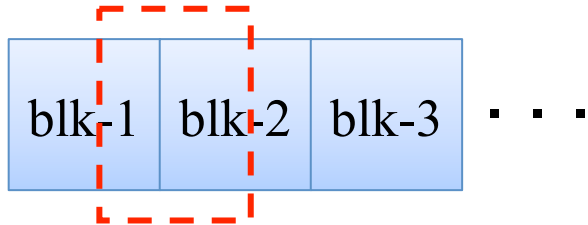
# read( ), write( )のブロックサイズ

- システムコール使用時には、ハードウェアやカーネルの特性を考慮すべき場合がある
- ハードディスクの入出力には注意が必要
  - ハードディスクは固定長のブロック単位で入出力
  - ブロックの境界からブロック単位で入出力するとよい
  - 最適な入出力のブロックサイズは `stat( )`, `fstat( )` で得られる(後述)

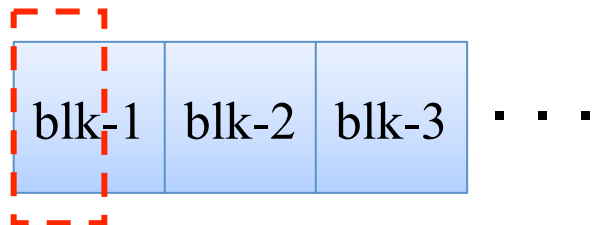
# read( ), write( )のブロックサイズ



- ブロック境界から1ブロック分を read
- → 1回のディスクアクセス



- ブロック境界以外から1ブロック分を read
- → 2回のディスクアクセス



- ブロック境界から1ブロック分以下を write
- → 2回のディスクアクセス
  - まずディスクから read し(1回目),  
書き込む部分のデータを書き換え,  
ディスクに書き戻す(2回目)

# stat( ), fstat( ) : ファイルの属性の取得

```
#include <sys/types.h>
#include <sys/stat.h>

int
stat(char *path, struct stat *sb);

int
fstat(int fd, struct stat *sb);
```

- **path** : ファイル名
- **fd** : ファイル記述子
- **sb** : 属性情報が格納されるメモリオブジェクトを指すポインタ  
必ずメモリオブジェクトを指していること

# struct stat のメンバー

ファイルシステム関連		サイズ関連	
st_dev	デバイスの識別子	st_size	バイト単位のサイズ
st_ino	i-node番号	st_blksize	I/Oの最適なブロックサイズ
st_nlink	ハードリンクの数	st_blocks	512バイト単位のブロック数
時刻関連		アクセス関連	
st_atime	最後のアクセス時刻	st_uid	所有者のユーザ識別子
st_mtime	最後の変更時刻	st_gid	グループ識別子
st_ctime	最後の属性変更時刻	st_mode	パーミッション
st_birthtime	i-nodeが作成された時刻		

**i-node** : ファイルの属性情報や, ファイルがどのディスクブロックから構成されているかを管理するデータ構造

**i-node番号** : カーネル内部のファイルの識別子

# ディレクトリ情報の読み込み

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

```
DIR *
opendir(char *filename);
```

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

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

opendir() : ディレクトリのオープン

readdir() : 次のエントリを得る

closedir() : ディレクトリのクローズ

- readdir()では、ライブラリ内部にメモリオブジェクトが確保され、次のディレクトリエントリの情報がこのメモリオブジェクトに格納される。
- このメモリオブジェクトへのポインタが返される。


- “man 5 dir” により struct dirent のメンバーを調べられる。
- 通常使うのは “char d\_name[ ]” 程度であろう。



# fopen( ): ストリームとしてファイルをオープン

```
#include <stdio.h>
```

```
FILE *  
fopen(char *filename, char *mode);
```

 ファイル名を指定

- mode の値
  - “r” read用にオープン. ストリームのポインタはファイルの先頭.
  - “r+” readおよびwrite用にオープン. ポインタは先頭.
  - “w” 既存の内容を消去, あるいはファイルを新規作成. ポインタは先頭.
  - “w+” readおよびwrite用にオープン. ファイルが存在しない場合は新規作成. 存在する場合は内容を消去. ポインタは先頭.
  - “a” write用にオープン. ファイルが存在しない場合は新規作成. ポインタは末尾.
  - “a+” readおよびwrite用にオープン. ファイルが存在しない場合は新規作成. ポインタは末尾.
- 返り値
  - 正常: ファイルポインタ (FILE \*)
  - エラー: NULL

# fread( ): ストリームからのバイト列のread

```
#include <stdio.h>
```

```
size_t
```

```
fread(void *ptr, size_t size, size_t nitems,  
      FILE *stream);
```

- 機能
  - stream から size バイトの大きさを持つオブジェクトを nitems 個読み込み, ptr が指す領域に格納する.
- 返り値
  - 正常: 読み込んだオブジェクトの数
  - EOF: 0
  - エラー: 0
  - EOFかエラーかは feof( )や ferror( )で調べる

# fwrite( ): ストリームへのバイト列のwrite

```
#include <stdio.h>
```

```
size_t
```

```
fwrite(void *ptr, size_t size, size_t nitems,  
       FILE *stream);
```

- ・ 機能
  - ・ ptr が指す領域にある, size バイトの大きさを持つ nitems 個のオブジェクトを stream に書き込む.
- ・ 返り値
  - ・ 正常: 書き込んだオブジェクトの数
  - ・ エラー: nitems より小さい値

# fclose( ): ストリームのクローズ

```
#include <stdio.h>
```

```
int  
fclose(FILE *stream);
```

- ・ stream をクローズする.
- ・ バッファされていたデータは書き込まれないので, fclose( ) の前に fflush( ) を呼び出すのがよい.
- ・ 返り値
  - ・ 正常: 0
  - ・ エラー: EOF

# 演習

- cp コマンドの簡易版である mycp を作成しなさい.
- 構文: `mycp file1 file2`
  - *file2* が既に存在する場合は, 上書きしてよいかを尋ねる
- 試してみよう
  - 1024バイトごとに `read()` / `write()` を呼び出す場合と, 1バイトごとに `read()` / `write()` を呼び出す場合で実行時間を比べてみよう.
  - `read()`, `write()` の代わりに `fread()`, `fwrite()` を使うと, 実行時間はどうか?