

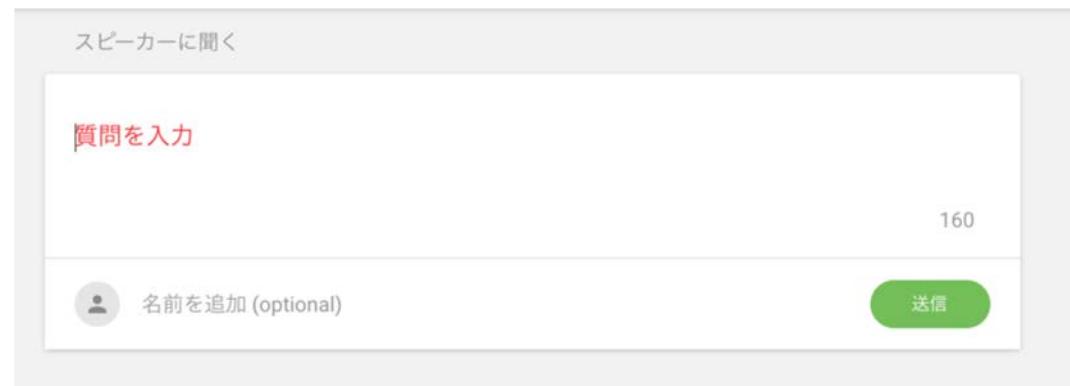
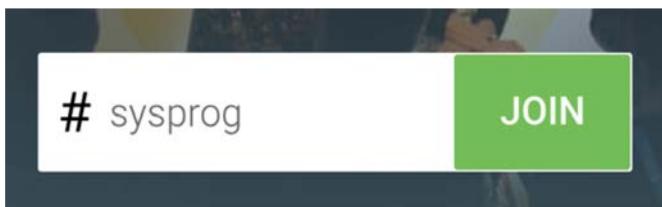
2019年度 システムプログラミング (CSC.T344)

2. 入出力

林晋平@情報理工学院

sli.do で質問しよう

- <https://sli.do/> スマホからでもOK
 - ログイン不要. コード #sysprog にアクセス
 - (匿名で) 質問できます
 - 疑問点を気兼ねなく投稿して下さい
 - 授業中(余裕があったら)拾います
 - ネタも投稿して構いませんが……(拾うかは気分次第)
 - 投稿された質問に投票  できます



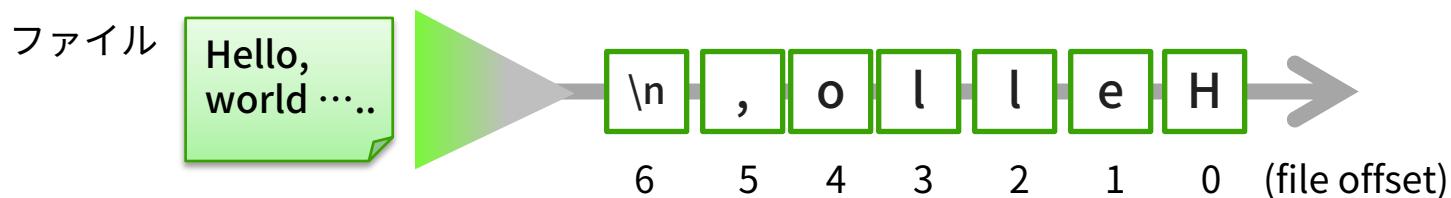
UNIX環境演習のヒント

- 標準入力と引数のファイルを（ほぼ）同一視するコマンド
 - cat, head, tail, wc, sort, uniq, grep, ...
 - 入力（ファイル）内の行を処理し、出力するタイプ
- すべてのコマンドがそうというわけではない
 - 例：kill
 - 使い方：kill プロセス番号

低レベル入出力

UNIXにおける「ファイル」

- ファイル：バイトが一列に並んだもの
 - バイットストリーム (byte stream) とも呼ばれる



- ファイル抽象
 - 様々なものが「ファイル」として操作される
 - 普通のファイル
 - d: ディレクトリ
 - l: シンボリックリンク
 - c: キャラクタデバイス (例: コンソール)
 - b: ブロックデバイス (例: ディスク)
 - s: ソケット (例: プロセス間通信)
 - P: パイプ

低レベル入出力

- システムコールによる入出力操作
 - バイト単位. 読んだデータはヌル終端しない
 - バッファリングなし (cf. 標準入出力ライブラリ)
- **open/close:** ファイルを開く/閉じる

```
#include <fcntl.h>
int open(const char *path, int flags, ...);
#include <unistd.h>
int close(int fd);
```

- **read/write:** ファイルから読む/へ書く

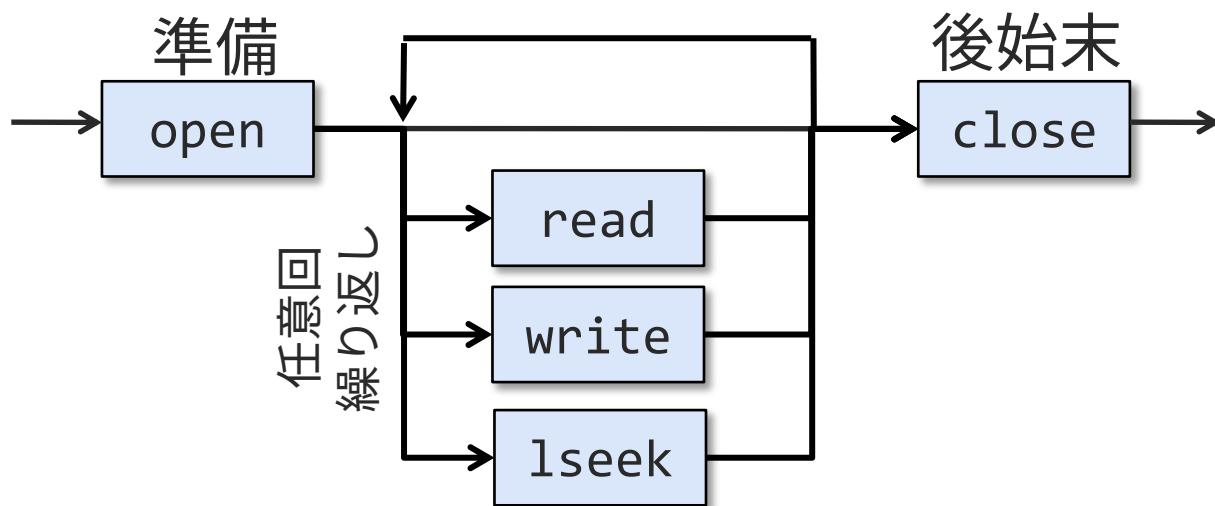
```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t nbytes);
ssize_t write(int fd, const void *buf, size_t nbytes);
```

- **lseek:** 位置決め

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

ファイル入出力の基本手順

- まず **open** でファイル読み書きを準備
 - ファイル記述子 (File Descriptor; fd) を受け取る
- **fd** を指定して処理 (位置決め, 読み, 書き) を繰り返す
- **fd** を指定して **close** で後始末 (資源の解放など)
 - ファイルはプロセス終了時に自動的にcloseされるが、明示的にcloseするのがよい作法



open: ファイル入出力の準備

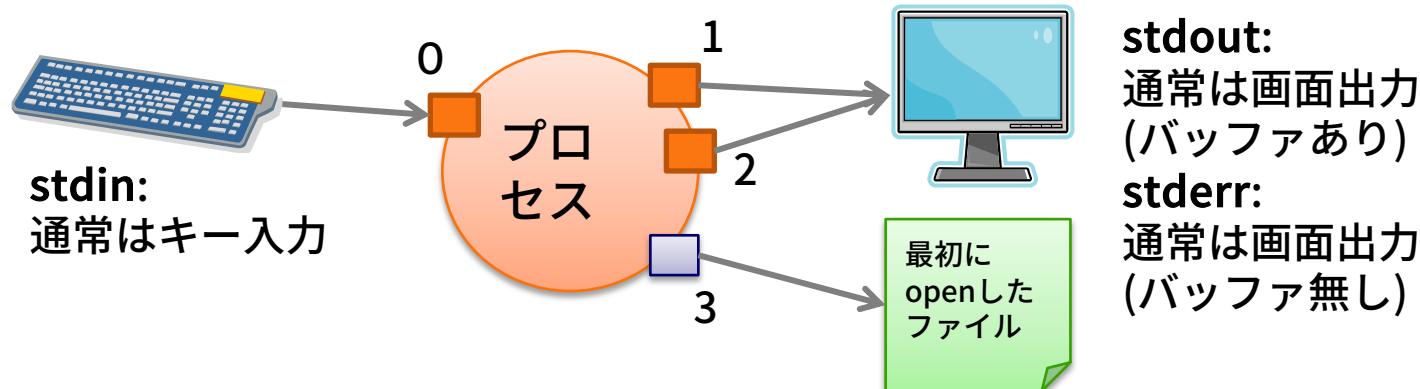
```
#include <fcntl.h>
int open(const char *path, int flags);
int open(const char *path, int flags, int mode);
```

- **返り値**
 - ファイル記述子. この値でカーネルが管理
- **引数**
 - path: 対象のファイルのパス
 - flags: 以下の定数マクロのビット和 (| でつなげる)
 - O_RDONLY 読むだけ
 - O_WRONLY 書くだけ
 - O_RDWR 読み書き両方
 - O_CREAT ファイルがなければ作る
 - O_TRUNC ファイルがあればまず空にする
 - mode: O_CREAT で作るファイルのアクセス許可ビット

} いずれか1つを
必ず指定

ファイル記述子 fd

- カーネルが管理するための情報
 - データ, ファイル構造体を管理
- 新しいファイルの fd は **3** から始まる
 - プロセスが起動すると自動的に3つ open
 - 0: `stdin` = standard input 標準入力
 - 1: `stdout` = standard output 標準出力
 - 2: `stderr` = standard error output 標準エラー出力



UNIXのファイルアクセス制御

- ユーザ, グループ
 - すべての UNIX ユーザはユーザID, (複数の) グループIDを持つ
 - id コマンドで確認可能
- ファイルの属性
 - 所有者 (のユーザID)
 - 所有グループ (のグループID)
 - アクセス許可ビット
- アクセス許可ビット
 - { 所有ユーザ, 所有グループ, グループ外 } のユーザが
 - { 読み, 書き, 実行 } できるか否か
 - を表すビットフラグ列
- 例

$1\textcolor{red}{1}0\textcolor{blue}{1}00100 = 0\textcolor{red}{6}\textcolor{blue}{44}_8$

読み 実読み 実読み 実
ユーザ グループ その他
- chmod コマンドにより変更可能

read/write: ファイルの読み書き

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t nbytes);
ssize_t write(int fd, const void *buf, size_t nbytes);
```

- 引数

- fd: ファイル記述子
- buf: バッファの先頭を指すポインタ
- nbytes: 読み込むバイト数

- 戻り値: 読み書きしたバイト数. エラー時は -1

- read: nbytes > 戻り値 → EOFまで読み込んだ
- write: nbytes != 戻り値 → 異常な状態

補足

size_t: 符号なし整数 (処理系非依存の表現)

ssize_t: 符号付き整数 (処理系非依存の表現)

void *: バッファの領域がどの型でも読みめる

ファイル入出力：例1

```
#include <fcntl.h> /* open */
#include <unistd.h> /* write, close */
int main(void) {
    int fd = open("tmp.txt",
                  O_WRONLY | O_CREAT | O_TRUNC,
                  0666);
    write(fd, "hello\n", 6);
    close(fd);
}
```

エラーチェックしよう 😕

書き込みのみ、ファイル作成、
存在したらサイズを0に
(|でビットの論理和を取り指定)

ちゃんと strlen を使おう 😕

実行結果

```
% gcc io1.c -o bin/io1
% bin/io1
% cat tmp.txt
hello
%
```

errno/perror: エラーの対処

```
#include <errno.h>
extern int errno;
#include <stdio.h>
void perror(const char *s);
```

- エラーコードの設定: **errno**

- システムコールはエラー発生時にエラーコードを **errno** に指定

- エラーの表示: **perror**

- perror は **errno** のエラーコードに基づき エラーメッセージを標準エラー出力に表示
 - “(引数): (errnoのエラーメッセージ)” という形でエラーを出力
 - 引数にはコマンド名や対象ファイル名など、 どこの何に対するエラーなのか特定できるような情報を渡す
 - コマンド名を指定すれば、パイプ等で複数のコマンドを実行しているとき、どのコマンドでエラーが出ているのかわかる

```
#include <errno.h> /* errno */
#include <fcntl.h> /* open */
#include <stdio.h> /* perror */
#include <stdlib.h> /* exit */
#include <string.h> /* strlen */
#include <unistd.h> /* write, close */
int main(void) {
    char *buf = "hello\n";
    int fd = open("tmp.txt", O_WRONLY | O_CREAT | O_TRUNC, 0666);
    if (fd == -1) {
        perror("io1");
        exit(errno);
    }
    if (write(fd, buf, strlen(buf)) == -1) {
        perror("io1");
        exit(errno);
    }
    if (close(fd) == -1) {
        perror("io1");
        exit(errno);
    }
}
```

ここでは6文字書けなかった
場合は省略

ファイル入出力：例2

```
#include <fcntl.h>      /* open */
#include <stdio.h>       /* printf */
#include <sys/types.h>   /* open, read */
#include <sys/uio.h>     /* read */
#include <unistd.h>      /* close, read */

int main(void) {
    char buf[] = "123456789"; bufのサイズは?
    int fd = open("tmp.txt", O_RDONLY);
    int n = read(fd, buf, 9); fooの内容を
                                最大9byte読込む
                                ※ 注意! ≠ 10byte
    printf("n=%d, buf=[%s]\n", n, buf);
    close(fd);
}
```

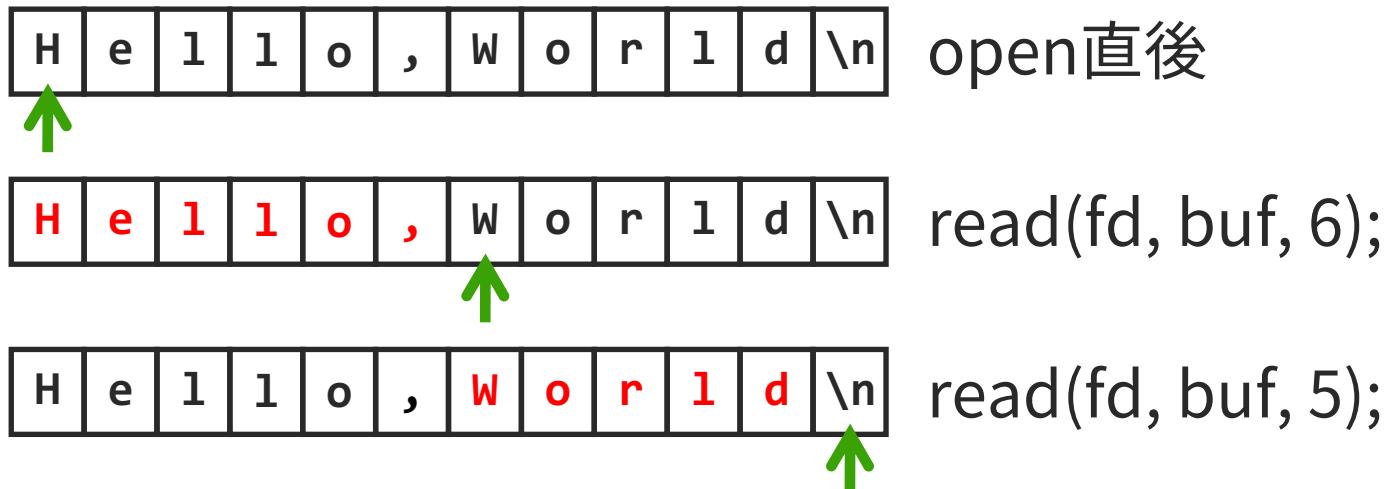
bufの「文字列」を出力
(\0終端を想定)

実行結果

```
% gcc io2.c -o bin/io2
% bin/io1 ; bin/io2
n=6, buf=[hello
789]
%
```

カレントファイルオフセット

- **readを2回すると、
2回目は前データの次から読む**
 - どこまで読んだか (= current file offset) を
カーネルが覚えている



lseek: 位置決め

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

● 引数

- fd: ファイル記述子
- offset: 相対位置 (off_tは符号付整数)
- whence: 以下のマクロで基準点を指定
 - SEEK_SET: ファイル先頭
 - SEEK_CUR: 現在の位置
 - SEEK_END: EOF

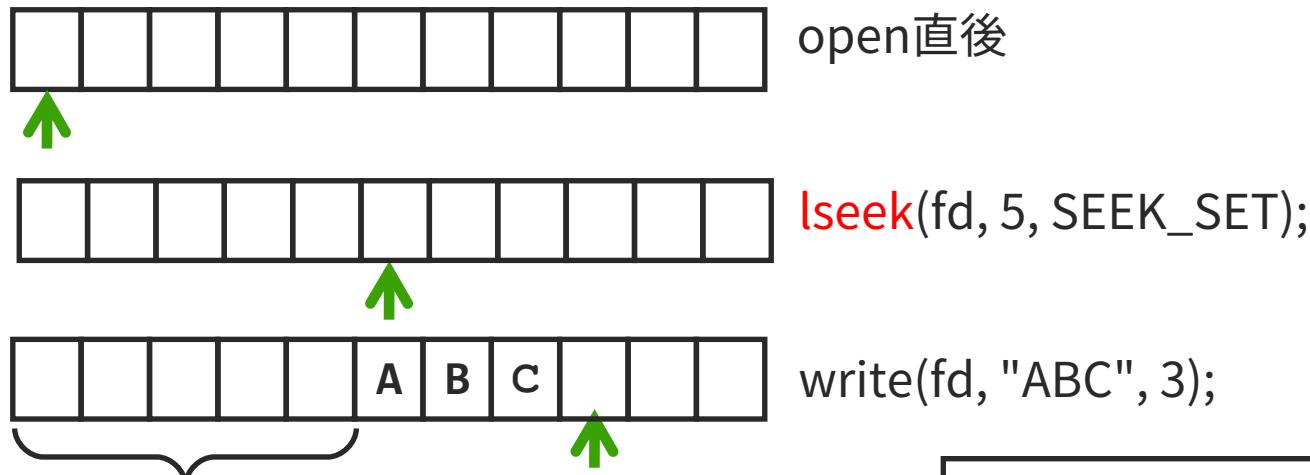
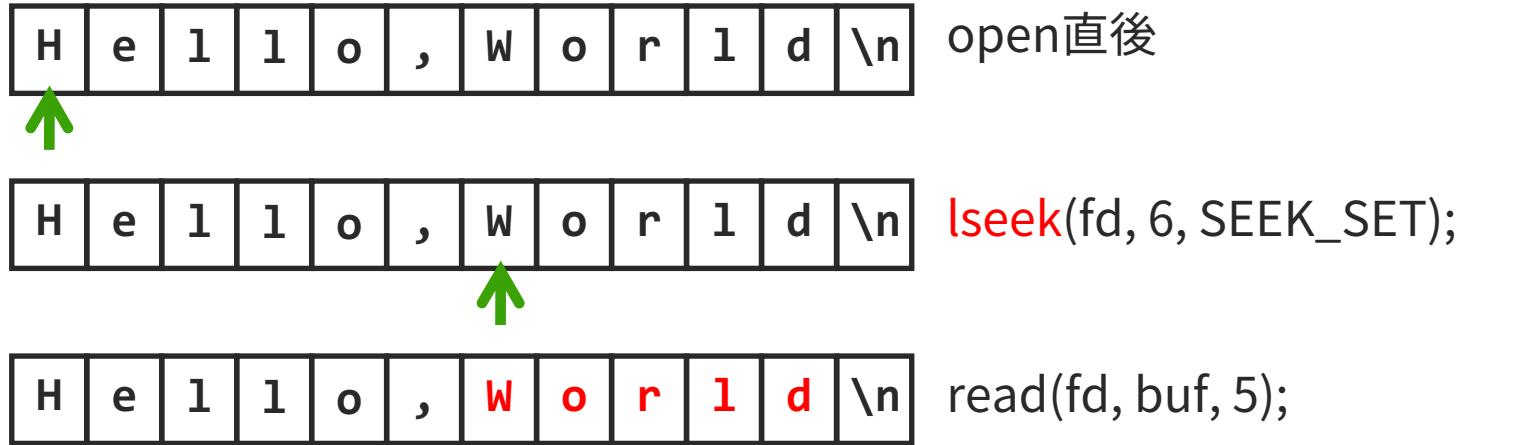
● 返り値

- 移動後の読み書き位置

● 例:

- lseek(fd, -5, SEEK_CUR): 現在位置から手前に5byte目
- lseek(fd, 15, SEEK_SET): 先頭から15byte目

lseek: 動作イメージ



NULLで埋まる = ホール (hole)

```
% od -c foo  
\0 \0 \0 \0 \0 A B C
```

クイズ

- このプログラム実行後の foo の中身は?

```
#include <fcntl.h> /* open */
#include <unistd.h> /* write */

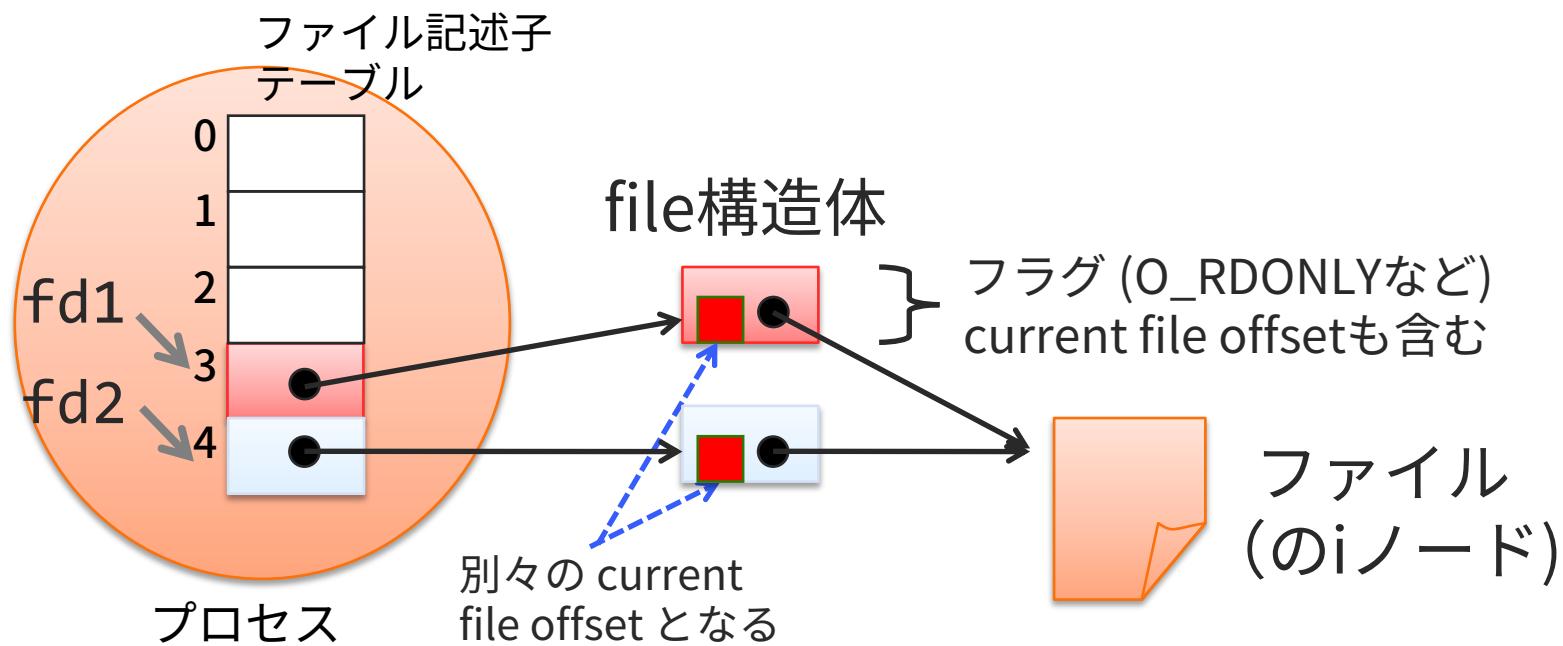
int main(void) {
    int fd1 = open("tmp.txt", O_WRONLY | O_CREAT | O_TRUNC);
    int fd2 = open("tmp.txt", O_WRONLY | O_CREAT | O_TRUNC);
    write(fd1, "ABCD", 4);
    write(fd2, "ab", 2);
    close(fd1);
    close(fd2);
}
```

```
% bin/ioquiz
% cat foo
```

クイズの解説

```
int fd1 = open("tmp.txt", ...);  
int fd2 = open("tmp.txt", ...);  
write(fd1, "ABCD", 4);  
write(fd2, "ab", 2);
```

- **open する毎に新しい current file offset 変数をカーネル内で確保する**
 - file 構造体は current file offset を保持



補足

- ファイル操作は基本的にアトミック操作
 - ファイル操作はプログラムからは不可分な一つの操作に見える
- **ioctl(), fcntl():**
ファイル抽象以外の操作に利用
 - ioctl: デバイスの制御
 - fcntl: ファイル記述子への操作

標準入出力ライブラリ

標準入出力ライブラリ

- ライブラリによる入出力操作
 - FILE構造体, fopen, fclose
 - fprintf, fscanf, fread, fwrite,
 - 文字・文字列・バイト単位, フォーマット処理
 - バッファリングあり (cf. 低レベル入出力)
- **(FILE *)型のデータ＝ストリーム**
 - 標準入出力ライブラリでファイル操作に用いる
 - 低レベル入出力のファイル記述子に相当
 - FILE構造体の中身を見てはいけない

```
#include <stdio.h>
FILE *fopen(const char *restrict path, const char *restrict mode);
int fclose(FILE *stream);
```

stdin, stdout, stderr

- 既存のストリームを指すマクロ
- fopen/openしなくても利用可能

| | ストリーム | ファイル記述子 | デフォルト | |
|---------|--------|---------------|-------|-------|
| 標準入力 | stdin | STDIN_FILENO | 0 | キーボード |
| 標準出力 | stdout | STDOUT_FILENO | 1 | 端末画面 |
| 標準エラー出力 | stderr | STDERR_FILENO | 2 | 端末画面 |

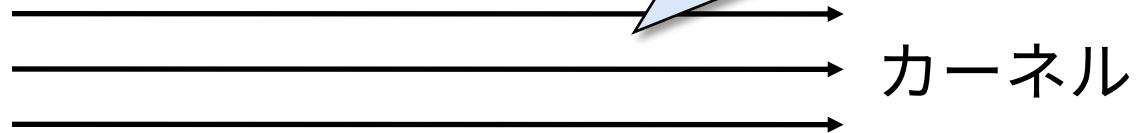
バッファ

- バッファ (buffer)
 - データ受け渡し用の記憶領域
 - 本来の意味は「緩衝」
- バッファリング (buffering)
 - バッファにデータをためて、データ入出力を高速化すること
- 標準入出力ライブラリはバッファを利用
 - バッファリングで低レベル入出力の発行回数を減らして速度向上
 - バッファにデータが残っているか注意が必要

バッファリングの概要

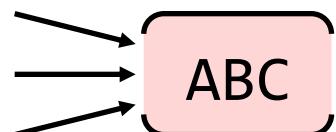
- バッファリングなし

```
write("A")  
write("B")  
write("C")
```



- バッファリングあり

```
printf("A")  
printf("B")  
printf("C")
```



小データを何度も
writeすると遅い

カーネル

まとめてwrite
するので高速

いったんバッファ
にためる

バッファリングの効果：例

- (手元の MacBook Pro 2017 で測った)

| 方法 | 時間 |
|-----------------|----------|
| fwrite, 10MBを1回 | 32ms |
| fputc, 1Bを10M回 | 512ms |
| write, 10MBを1回 | 13ms |
| write, 1Bを10M回 | 49,354ms |

遅い

fopen, fclose

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

```
FILE *fopen(const char *restrict path,  
            const char *restrict mode);
```

準備

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

後始末

● 引数

- path : 読み書きしたいファイルのパス名
- mode : 読み書きを文字列で指定
 - "r": 読み専用, O_RDONLY
 - "w": 書き専用, O_WRONLY | O_CREAT | O_TRUNC
 - "a": 追記専用, O_WRONLY | O_CREAT | O_APPEND
 - 上には "+" と "b" を組み合わせ可

読み書き

バイナリ

文字単位, 文字列単位の入出力

● 文字単位

- `getc()`, `putc()` - マクロ (またはインライン関数)
- `fgetc()`, `fputc()` - 関数
- `getchar() = getc(stdin)`,
`putchar(c) = putc(c, stdout)`

● 文字列 (行) 単位

- ×`gets()`, `puts()`
 - `gets`は使用禁止 (後方互換性のために存在)
 - バッファオーバーフローの原因
 - `gets`は改行を取る, `puts`は改行をつける
- `fgets()`, `fputs()`
 - ストリームを引数で指定
- `gets()`と`fgets()`は行単位で入力
 - cf. `puts()`と`fputs()`は複数行を一度に出力

バイト単位（バイナリ用）

- バイト単位（バイナリ用）
 - `fread()`, `fwrite()`
- バイナリ
 - テキスト形式（文字データ）以外のデータ
 - 例：`a.out`, JPEGなどの画像データ
- いつ `fread/fwrite` を使う?
 - 改行文字の変換を避けたいとき
 - Windows は "`\r\n`" ⇄ 標準ライブラリは "`\n`"
 - UNIX や Mac OS X の改行は "`\n`" なので関係なし
 - ヌル文字（"`\0`"）を出力したい時
 - `fputs()` では無理

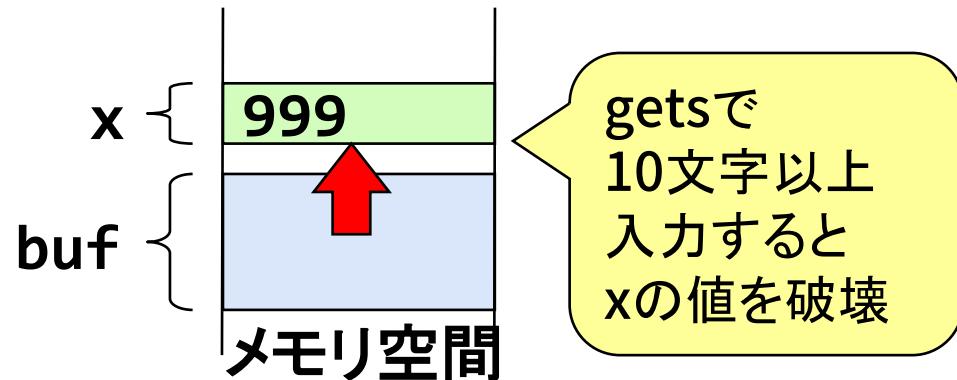
バッファオーバーフロー

● バッファオーバーフロー (buffer overflow)

- 確保した領域を超えたデータの格納
- 脆弱性 (vulnerability) や不安定動作の原因

● まずい例

```
char buf[10];  
int x = 999;  
gets(buf);
```



● バッファオーバーフローを避ける

- gets() を使わず, fgets() を使う
- sprintf() を使わず snprintf() を使う

フォーマット付き入出力

- フォーマット（書式）つき
 - `printf()`, `fprintf()`, `sprintf()`, `snprintf()`
 - `scanf()`, `fscanf()`, `sscanf()`
- 注意
 - `scanf()` 系では返り値をチェックすること
 - 返り値：マッチしたデータの個数
 - 指定より少なければ、入力に失敗している
 - `scanf()`ではなく、`fgets()`と`sscanf()`を使う
 - `scanf()`は失敗時にやり直せない
 - バッファオーバーフローを避ける
 - 書式とデータの型をあわせる

バッファリングの種類と設定

- バッファリングの種類

- 行バッファリング：人間が相手のとき
- フルバッファリング：機械が相手のとき
- バッファリングなし：stderrなどで使用

- バッファリングの設定

- fflush()：強制的に出力バッファをフラッシュ
- setbuf(), setvbuf()：バッファリングの設定