Linux システムコール演習 1

岡野浩三

2023 年 6 月 13 日

Linux は正確には UNIX ではない．また POSIX にも準処していない．ただし準処していない理由は準処の手続きを種々の理由によりしていないだけであり，技術的には準拠しているといってよいだろう．したがって Linux のシステムコールを用いた C プログラムの演習を通じて UNIX のシステムコールを用いた C プログラムの演習を行ったといってもそう間違いではないだろう．

問 1
POSIX とは何か?調べよ．

1 Simple cat command

簡単な cat プログラムからはじめてみよう．ここで言う「簡単な」の意味はオプション処理なし，標準入力からの読み込みなし，エラー処理も単純に最低限のメッセージ処理の後プログラムの停止という意味になる．

問 2
標準の *cat* コマンドについて *man* 等で調べよ．

まずは通常の C 標準ライブラリを用いたプログラム例を示す．

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

int main(int argc, char* argv[]) {
    int i;
    for(i=1; i < argc; i++) { // we start from the 1st argument
        FILE *f;
        int c;
        f = fopen(argv[i], "r");
        if (f==0) {
            perror(argv[i]);
            exit(1);
        }
```

```
        while ((c = fgetc(f)) != EOF) {
            if (putchar(c) < 0 )
                exit(1);
        }
        fclose(f);
    }
    exit(0);
}
```

このファイルを catstd.c とすると

```
$cc -o cat catstd.c
$./cat catstd.c
```

などで実行できる．引数の数を 2 つ以上に増やしたり，存在しないファイルを指定したり，あるいは無引数で実行してみよう．またエラー終了する際の終了ステータスも確認してみよう (方法はわかりますね？).

演習 1

ここまで実行せよ．実際に引数にさまざまなファイルを渡して実行すること．

fopen(), fclose(), perorr() を使うために`#include <stdio.h>`を，そして exit() を使うために `#include <stdlib.h>` 使っている．なおプログラムの本体は C プログラムの初歩で習う内容なので解説は省略する．ファイルを使うために FILE 型の構造体 (struct) を使っているが，一般のプログラマはこれがどういうものかは通常見えないようになっている．

FILE 型が具体的にどのように定義されているかは /usr/include/ 下のヘッダーファイルを地道に検索するれば見つかる．しかしながらマクロ命令などを解析して追っていくのはかなりの苦労でしょう．

cc コマンドはプリプロセス (すなわち *.h などに書かれているマクロの展開など) の作業結果を表示するオプションがあるので今回はそれを使う．

```
$cc -E catstd.c
```

の結果を追っかけると FILE 型は _IO_FILE 型で再定義され_IO_FILE 型は複数のエントリを持つ構造体であることが読み取れる．_IO_FILE 型はおそらくバッファやその他現在のファイルの読み書き位置を記録する構造体であろうことがエントリから読み取れる．

注: 上記の結果ですら想像に以上に膨大になる．

```
$cc -E catstd.c > out.txt
```

で out.txt に結果をリダイレクトし，じっくり out.txt を眺めるとよい．

なお，通常の使い方では，特に理由がない限りは最初に提示したような stdio.h で定義されている関数を使ってプログラムを記述すべきであろう．

ここまでをまとめると

1. ファイルの入出力は (あるいはそれに限らず) 互換性などの理由により特に理由がなければ通常は stdio.h (あるいは標準ライブラリ) で定義された関数を用いるべき.

2. stdio.h で定義された関数はバッファの機能を用いておりおそらくは，直接システムコール (のラッパー) を呼ぶよりは効率よく処理する.

余談となるが，OS の C プログラムを理解するために必要なことは C プログラムの基本知識やコンピュータアーキテクチャの基本知識はもちろんのこと，それ以上にヘッダーファイルに書かれている膨大なマクロを理解することが挙げられる．この作業は想像以上に厳しい.

この資料は OS の講義の演習の位置づけですので，この cat をあえてシステムコール (のラッパ関数) を用いて記述しなおしてみる.

システムコールとして以下の 4 つを新たに使う.

open, close, read, write

名前からわかるように open, close の 2 つについては fopen(), fclose() にほぼ対応している.

read, write は原則としてまとまったデータ単位で読み書きをするほうが，効率よく動作できる．そのためバッファを引数にとる．また，標準ライブラリを用いたバージョンでは FILE 型を用いたが，システムコールではファイルやストリームの指定にファイルディスクリプタと呼ばれる整数値を用いる.

read() は引数を 3 つとる．第一引数はファイルディスクリプタ，第二引数はバッファ構造体，第三引数は最大読み込みバイト数を表す.

返り値は実際に読み込んだバイト数になる．EOF の場合は 0，エラーの場合は-1 となる.

実際の仕様を

```
$man 2 read
```

で確かめよ．この際，インクルードすべきファイルが unistd.h であることもわかる (おそらく unix standard の略でしょう).

write() も同様に引数を 3 つとる．第一引数はファイルディスクリプタ，第二引数はバッファ構造体，第三引数は最大書き込みバイト数を表す.

返り値は実際に書き込んだバイト数になる．EOF の場合は 0，エラーの場合は-1 となる.

open() は第一引数にファイルパスを渡す．第二引数はオープンするときのモードを指定する．この値は fcntl.h に記述される．`O_RDONLY`, `O_WRONLY`, `O_RDWR` が用いられる．これらの意味は

```
$man 2 write
```

で確かめよ．実際にはこれらとオプションでさらにファイルの存在に関係し挙動を変えられるオプションフラグを指定することもできる．それらのオプションはビット和をとる演算子 | を用いて併記してあたえることになる．返り値はファイルディスクリプタになる．エラーのときは-1 が返る.

close() は引数にファイルディスクリプタをとり，そのストリーム (ファイル) を閉じる.

プログラム起動時には通常，ファイルディスクリプタ 0,1, 2 がすべてオープンになっている．またそれぞれ，標準入力，標準出力，標準エラー出力に割り当てられている.

プログラムを以下に示す．標準ライブラリ版と見比べれば特に解説はいらないと判断する．なお，無限ループの表記テクニックとして while(TRUE) を使っている.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

void do_cat(const char *path);

int main(int argc, char *argv[]) {
    int i;
    for (i=1; i < argc; i++) {
        do_cat(argv[i]);
    }
    exit(0);
}

#define BUFFER_SIZE 20148
#define TRUE 1

void do_cat(const char* path) {
    int fd;
    unsigned char buf[BUFFER_SIZE];
    int n;

    fd = open(path, O_RDONLY);
    if (fd < 0) {
        perror(path);
        exit(1);
    }
    printf("current fd= %d\n", fd);
    while(TRUE) {
        n=read(fd, buf, sizeof buf);
        if (n < 0) {
            perror(path);
            exit(1);
        }
        if (n == 0) {
            break;
        }
        if (write(STDOUT_FILENO, buf, n) < 0) {
```

```
            perror(path);
            exit(1);
        }
    }
    if (close(fd) < 0) {
        perror(path);
        exit(1);
    }
}
```

演習 2
システムコール版を記述し，動かしてみよ．

次に，do_cat() をあえて別のファイルに移動させる．以下のように catmain.c と catsub.c の２つのファイルを作成することになる．
ヒント: 前に作成したものからコピーして，あとは不要な行を削除すると簡単に作れます．

```
$cat catmain.c
```

```
#include <stdlib.h>

void do_cat(const char *path);

int main(int argc, char *argv[]) {
    int i;
    for (i=1; i < argc; i++) {
        do_cat(argv[i]);
    }
    exit(0);
}
```

```
$cat catsub.c
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#define BUFFER_SIZE 20148
#define TRUE 1
```

```
void do_cat(const char* path) {
    int fd;
    unsigned char buf[BUFFER_SIZE];
    int n;

    fd = open(path, O_RDONLY);
    if (fd < 0) {
        perror(path);
        exit(1);
    }
    printf("current fd= %d\n", fd);
    while(TRUE) {
        n=read(fd, buf, sizeof buf);
        if (n < 0) {
            perror(path);
            exit(1);
        }
        if (n == 0) {
            break;
        }
        if (write(STDOUT_FILENO, buf, n) < 0) {
            perror(path);
            exit(1);
        }
    }
    if (close(fd) < 0) {
        perror(path);
        exit(1);
    }
}
```

2 Makefile

このような小さなプログラムではあまり効果がないが，大きなプログラムでは機能ごとファイルに分割すると見通しが良くなる．catmain.c の 3 行目で do_cat() の宣言だけしていることに注意すること．この宣言により，do_cat() の実装がどこか別のファイルで行われていることをコンパイラに知らせることができる．

これらのファイルを分割コンパイルするには以下のように cc を実行する．

```
$cc -c catmain.c
$cc -c catsub.c
$cc -o cat catmain.o catsub.o
```

想像できると思われるが１行目，２行目でそれぞれ catmain.c, catsub.c を個別にコンパイルしている．コンパイル結果はそれぞれ catmain.o, catsub.o に格納される．

３行目で catmain.o, catsub.o を結合し，一つの実行ファイル cat を作成している．

演習 3

catmain.c catsub.c の *include* の各行をそれぞれ独立に１行ごと削除してコンパイル結果を観察せよ．何がわかるだろうか? コンパイラの *warning* は無視できるだろうか？実際に削除するのではなく，コメントアウトを活用しよう．

演習 4

catmain.c の３行目 *do_cat()* を削除するとコンパイルはどうなるだろうか？

分割してコンパイルしている利点の 1 つは変更のあったファイルだけコンパイルをし直せばよいことである．これにより大規模ソフトウェア開発の際のコンパイル作業で無駄なコンパイルをしなくてもよくなる．もちろん，最後の結合作業（上の 3 行目）は必要である．

次に make を使ってみる．

以下の内容を持つ Makefile を作成する．

```
$cat Makefile
```

```
cat: catmain.o catsub.o
        cc -o cat catmain.o catsub.o
catmain.o: catmain.c
        cc -c catmain.c
catsub.o: catsub.c
        cc -c catsub.c
```

2,4,6 行目の先頭はスペースではなく tab を用いることに注意する．

このファイルは以下の内容を表してしている．

1 行目 ファイル cat は catmain.o catsub.o に依存してる．

2 行目 cat を生成するには cc -o cat catmain.o catsub.o を実行する．

3 行目 ファイル catmain.o は catmain.c に依存してる．

4 行目 catmain.o を生成するには cc -c catmain.c を実行する．

5 行目 ファイル catsub.o は catsub.c に依存してる．

6 行目 catsub.o を生成するには cc -c catsub.c を実行する．

「依存している」の意味は「ファイルの中身が変わったら，その変更の影響がある.」というくらいの意味である．実際にはファイルの中身をいちいち調べているわけではなく，例えば 4 行目の場合 catmain.o と catmain.c のファイルの修正時刻を見て，catmain.c のほうが新しければ catmain.o も修正が必要であると判断する.

実際に make でコンパイルしてみよう．すでに cat, catmain.o catsub.o があるので以下をまずは実行して，これらのファイルを消そう．あやまって *.c を消さないように！

```
$rm catmain.o catsub.o cat
```

make の実行は極めてシンプルである.

```
$make
```

これにより，カレントディレクトリの下にある Makefile に従って，コンパイルが次々実行され cat ファイルが生成される.

次に mainsub.c を修正しよう．実際にエディタで修正してもよいが簡単のため touch コマンドを使い，ファイルの修正時刻だけ修正する.

```
$touch catsub.c
```

念のため以下を実行し，catsub.c と catsub.o の修正時刻の違いを確認しよう.

```
$ls -l catsub.?
```

再度 make を実行しよう.

```
$make
```

catsub.c のみが再コンパイルされているのがわかるだろうか？

演習 5
同様に *catmain.c* を修正して *make* を実行してみよ.

演習 6
同様に *catmain.c, catsub.c* を両方修正して *make* を実行してみよ.

演習 7
cat のみを 削除して *make* を実行してみよ.

演習 8
cat は削除せず *catmain.o catsub.o* を 削除して *make* を実行してみよ.

問 3
実際のプログラミングでは *Makefile* を書くときはマクロを使ってもう少し，系統的に記述する．どのように記述するのか調べよ.

演習 9
以下の記述が複数回現れている．このような記述は *1* つにまとめて関数化するほうが望ましい．*die(const char* path)* という関数にくくりだし，記述を簡潔にせよ *(*なおこのような記述改善をリファクタリングと呼ぶ*)*.

```
{
        perror(path);
        exit(1);
}
```

演習 10
ファイルの読み書きに関する標準ライブラリあるいはシステムコールとして *fseek, fseeko, ftell,ftello, rewind, fileno, fdopen, fflush, lseek, ioctl fcntl* などがある．これらについて調べよ．調べる際，標準ライブラリであるかシステムコールであるかの区別を明確にすること．なお *dup* も重要なシステムコールであるがこれについては後で *(fork* とともに*)* 説明することになろう．

open, close, read, write などをシステムコールと便宜上呼んでいるが厳密にはこれらは glibc がラッパ関数として提供しているシステムコール呼び出しのための C 言語関数である．したがって，通常の C プログラムからは通常の関数呼び出しと同じ使い方ができる．

一方，本当の生身のシステムコールをするためには機械語のトラップ (ソフトウェア割り込み) 等を使うのでアセンブラ記述で記述する必要がある．glibc はそのようなアセンブラ記述によるシステムコール呼び出しと一般の C プログラムからのシステムコール呼び出しを C 言語の関数呼び出しとして実現するためのインタフェースギャップをライブラリの形で解決し，ついでに互換性，移植性，効率化など複数の問題をまとめて解決している．

3 debug

C プログラムのデバッグはいくつか方法があるが，ここでは古典的に gdb を用いる方法を紹介する．gdb は CUI で動作するデバッガであり，古くから標準的なデバッガとして用いられてきた．プログラムの文単位での実行やブレークポイントの指定，変数値の確認ができる．

gdb を用いるにはコンパイルの際 -g オプションを付ける必要がある．Makefile を用いている場合，以下のように変更する．

```
$cat Makefile
```

```
cat: catmain.o catsub.o
        cc -o cat catmain.o catsub.o
catmain.o: catmain.c
        cc -c -g catmain.c
catsub.o: catsub.c
```

```
        cc -c -g catsub.c
clean:
        /bin/rm cat catmain.o catsub.o
```

ここで /bin/rm の前はタブである必要がある．

この記述では clean というターゲットをついでに作成した．make clean を実行するとこれら３つのファイルを削除できる．

このように記述して，実行ファイル cat を作成する．make を次のように実行して新たに cat を作り直す．

```
$make clean
$make
```

gdb の実行前に gdb のインストールが必要かもしれない．

インストールはいつものように apt-get を用いる．

```
$apt-get update
$apt-get install gdb
```

を実行する．必要ならコンテナを更新する．

```
$gdb cat
```

で cat に対してデバッグを実行する．

```
(gdb) b do_cat
```

で　関数 do_cat に対してブレークポイントを作る．ブレークポイントに実行が差し掛かると一時停止することになる．

```
(gdb)  run catmain.c
```

で引数 catmain.c を使って実行することになる．この後ブレークポイントまで自動で実行される．

```
(gdb)  s
```

で関数 do_cat のなかに入り 1step 実行することになる．

なお，ここで n を入れると next となり，do_cat の呼び出しが終わったところまで制御が進む．

```
(gdb)  p path
```

で，変数 path の内容を確認できる．

コマンド n, s などで実行していき，必要なところで変数を確認してデバッグをしていく．終了は quit

参考文献

[1] 青木峰郎:ふつうの Linux プログラミング Linux の仕組みから学べる gcc プログラミングの王道, ソフトバンククリエイティブ, ISBN-13: 978-4797328356, 2005.