システムプログラミング

Ver. 1.2.1

徳山工業高等専門学校
情報電子工学科

目次

第 1 章

システムプログラミング

本講義は，オペレーティングシステム本体が持つ機能を直接に使用するようなシステムプログラムの作成を行う．システムプログラムをプログラミングする経験の中から，オペレーティングシステム本体が備えている機能の役割りや必要性を体感的に学ぶ．

1.1 システムプログラムとは

システムプログラムは，乱暴な言い方をするとアプリケーション以外のプログラムのことである．図 1.1 にコンピュータシステムの構成を簡単に示す．この図でアプリケーションプログラムとハードウェアを除いた，オペレーティングシステム本体（カーネル），ライブラリ，ミドルウェア，ユーティリティ，プログラム開発環境はシステムプログラムである．

1. カーネル（OS の本体）
2. ライブラリ（プログラムが使用するサブルーチン，DLL ...）
3. ミドルウェア（DBMS，Web サーバ ...）
4. ユーティリティ（ファイル操作，時計，シェル，システム管理 ...）
5. プログラム開発環境（エディタ，コンパイラ，アセンブラ，リンカ，インタプリタ ...）

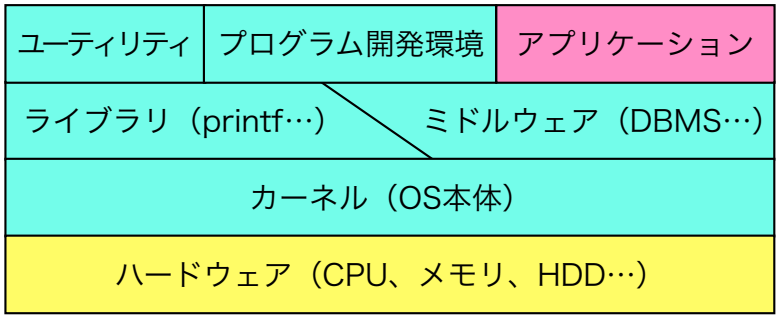


図 1.1　コンピュータシステムの構成

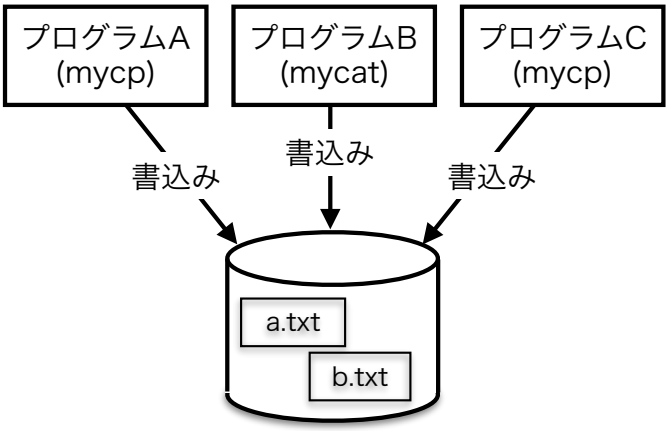


図 1.2　システムコールなし

1.2　システムプログラミングとは

システムプログラムを作成することをシステムプログラミングと呼ぶ．本講義では，システムプログラムの中でもユーティリティの作成を行う．Windows や macOS では GUI を備えた様々なユーティリティが準備されている[*1]．しかし，本講義の目的は「システムプログラミングを通してのオペレーティングシステムの体感的な理解」であるので，GUI の作成にエネルギーを費やしたくはない．そこで，CLI（Command Line Interface）のユーティリティを作成する．

本講義では，オペレーティングシステムを体感的に理解するために，オペレーティングシステムの機能を直接に使用する**簡単な *CLI* 版のユーティリティプログラムの作成（プログラミング）**を行う．

1.3　システムコール

一つのコンピュータシステムの中で複数のプログラムが同時に作動していることは，誰もが体験的に知っていると思う．しかし，それらのプログラムが勝手にシステムの**資源**にアクセスすると，資源の管理が正しく行えない可能性があり具合が悪い．例えばハードディスクでは，複数のプログラムが勝手にファイルを作成すると，複数のファイルがハードディスクの同じ領域に割当てられるかも知れない．

そこで，資源にアクセスするのはオペレーティングシステムの本体である**カーネル**だけに限り，カーネルが代表して資源を管理することにする．他のプログラムはカーネルに依頼し目的を達成する．その様子を図を使って説明する．

1. 図 1.2 のように，システムの中で同時に複数のプログラムが実行され，それぞれが**資源**にアクセスする必要がある．図の例では，三つのプログラムが同時にハードディスクにファイルを作ろうとしている．複数のプログラムが勝手にハードウェア**資源**（ハードディスク，プリンタ，メモリ ...）を操作すると具合が悪い．
2. そこで，図 1.3 のように資源を集中管理するプログラム，**カーネル**（OS の本体）を導入する．一

[*1] Windows や macOS の場合でも，GUI を備えていないユーティリティも，多数，存在する．

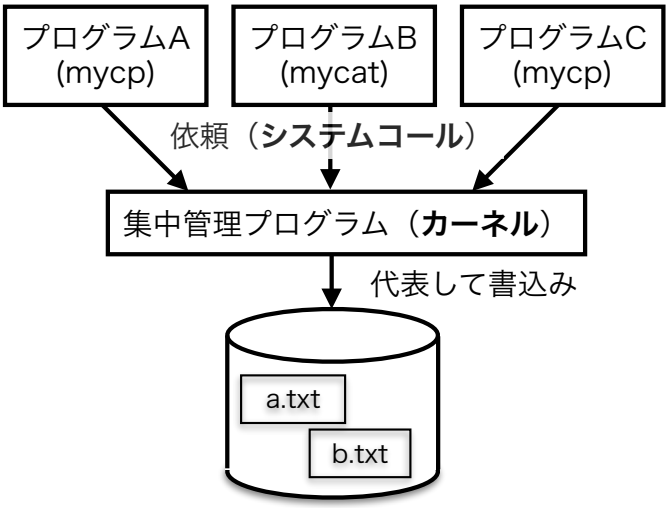


図 1.3　システムコールあり

般のプログラムは**システムコール**を用いてカーネルに処理を依頼する．

1.4　システムコールの使用

C 言語から，UNIX(macOS) のシステムコールを直接に利用することが可能である．以下の章では，macOS 上で C 言語を用いてシステムコールを直接に使用するプログラムを作成しながら，システムコールの機能を確認する．また，ユーティリティプログラムの簡単版を作成する．

```
// ディレクトリを作るユーティリティプログラム（mymkdir）の例
...
int main(int argc, char *argv[]) {
  if (argc!=2) {
    // エラー処理
    ...
  }
  mkdir(argv[1]);    // ディレクトリを作るシステムコール
  return 0;
}
```

練習問題

1. ハードウェア**資源**の例を挙げなさい．
2. **カーネル**の役割りを本章の範囲で説明しなさい．
3. **システムコール**の役割りを簡単に説明しなさい．
4. 自分がいつも使用しているコンピュータやスマートフォンのオペレーティングシステムの種類（Windows?）を調べなさい．

第 2 章

ファイル入出力システムコール

この章ではファイルの読み書きを行うシステムコールを紹介し，これらを直接に使用したユーティリティプログラムを作成してみる．C 言語を用いると，システムコールと同じ名前の関数を呼び出すことでシステムコールを呼び出すことができる．例えば open システムコールを使用するときは，`open()` 関数を呼び出す.

2.1 高水準入出力と低水準入出力

C 言語の入門で勉強した `fopen()`, `printf()`, `puts()`, `putchar()`, `fprintf()`, `fputs()`, `fputc()`, `scanf()`，`getchar()`，`fgets()`，`fgetc()`，`fclose()`... 等は**高水準入出力**関数と呼ばれる．これに対してシステムコールを直接使用する入出力を**低水準入出力**と呼ぶ.

表 2.1 に対応を示すように，高水準入出力関数は内部でファイル入出力を行うシステムコールを呼び出している．つまり，種類や機能が豊富な高水準入出力は，数種類の基本的な機能しか提供しない低水準入出力を用いて実現されていることになる.

表 2.1　高水準入出力関数とシステムコール

高水準入出力関数	対応するシステムコール
`fopen()`	open システムコール
`printf()`	write システムコール
`putchar()`	write システムコール
`fputc()`	write システムコール
...	...
`scanf()`	read システムコール
`getchar()`	read システムコール
`fgetc()`	read システムコール
...	...
`fclose()`	close システムコール

表 2.2　open システムコールの第 2 引数 oflag

以下の一つ	と	以下のいくつか
O_RDONLY（読み出し用）		O_APPEND（追記）
O_WRONLY（書き込み用）	+	O_CREAT（作成）
O_RDWR（読み書き両用）		O_TRUNC（切詰め）
		…

2.2　open システムコール

ファイルのオープンに使用するシステムコールである．詳細なマニュアルは UNIX(macOS) のターミナルで，man 2 open と入力すると表示される[*1]．

書式１（オープンするだけの場合）

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

解説　open システムコールを使用するプログラムの先頭では，fcntl.h をインクルードする必要がある．open システムコールは正常時には**ファイルディスクリプタ**（ゼロ以上の整数）を返す[*2]．エラーが発生した時は-1 を返す．エラー原因は perror() 関数で表示できる．

引数　path はオープンまたは作成するファイルのパス（名前），oflag はオープンの方法を表す。oflag は表 2.2 の記号定数を|[*3]で接続して書く．表の左側の記号定数を一つと，右側の記号定数を幾つか組合せることができる．例えばファイルの内容を消してから書き込み用にオープンしたい場合は，O_WRONLY|O_TRUNC のように書く．

使用例（書式１）

```
#include <fcntl.h>
...
int fdr, fdw, fda;
fdr=open("r.txt", O_RDONLY);              // 読み出し用にオープン
fdw=open("w.txt", O_WRONLY);              // 書き込み用にオープン
fda=open("a.txt", O_WRONLY|O_APPEND);     // 追記用にオープン

if (fdw<0) {                              // エラーチェック
  perror("w.txt");                        //   原因の表示
  exit(1);                                //   エラー終了
}
```

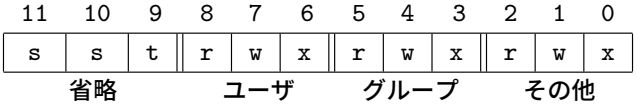
[*1] man は UNIX マニュアルを表示するユーティリティプログラムである．引数の 2 が表す第 2 章は，「システムコール」を解説している．本文中の例は「UNIX マニュアルの第 2 章の open」の項目を表示する．

[*2] stdin のファイルディスクリプタが 0，stdout のものが 1，stderr のものが 2 なので 3 以降の番号を返す．

[*3] |は，C 言語のビット毎の論理和演算子である．

ファイルの保護モード

open システムコールの第 3 引数（mode）は次のような 12bit の値である．



最初の 3bit の意味は難しいのでここでは説明を省略する．他のビットは rwx のどれかである．rwx の意味は次の通りである．

r : read 可（読み出し可能）

w : write 可（書き込み可能）

x : execute 可（実行可能）

例えば，第 8 ビットが 1 だったら、ユーザ（ファイルの所有者）が read（読み出し）可能の意味になる．ファイルのモードやユーザ（所有者），グループは次のようにして確認できる．

```
% ls -l a.txt
-rw-r--r--  1 sigemura  staff  0 Apr 11 05:53 a.txt
%
```

a.txt ファイルのモードの下位 9 ビットが 110100100 である．所有者は sigemura，グループは staff である．このファイルは sigemura が読み書きができる．staff グループに属するユーザは読むことだけできる．その他のユーザも読むことだけできる．

書式 2 （ファイル作成もする場合）

oflag に O_CREAT を含む場合は，該当ファイルが存在しないなら新規作成してからオープンする．新規作成するファイルの**保護モード**を mode で指定する．

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

mode_t は，符号なし 16bit 整数型である．mode は，作成されるファイルの保護モードである．mode は，8 進数で記述することが多い[*4]．8 進数と保護モードの対応は次のようになる．

```
0: ---      4: r--
1: --x      5: r-x
2: -w-      6: rw-
3: -wx      7: rwx
```

使用例 （書式 2）

```
fd=open("a.txt", O_WRONLY|O_CREAT, 0644);  // 作るファイルのモードは rw-r--r-- になる
```

[*4] C 言語では，数値を 0 で書き始めると 8 進数の意味になる．

2.3　read システムコール

読み出し用にオープン済みのファイルからデータを読み出すシステムコールである．１回目はファイルの先頭から指定されたバイト数を読み出す．２回目以降は，ファイルの前回読み終わった位置から続きを読み出す．このようにファイルの先頭から順に読み書きする方式は，**シーケンシャルアクセス（順次アクセス）**と呼ばれる．後で紹介する write システムコールもシーケンシャルアクセスを行う．

書式　（詳しくは `man 2 read` で調べる．）

```
#include <unistd.h>
ssize_t read(int fildes, void *buf, size_t nbyte);
```

解説　read システムコールを使用するプログラムの先頭では，`unistd.h` をインクルードする必要がある．オープン済みのファイルディスクリプタを渡し，ファイルからデータを読み出す．

read システムコールは `ssize_t` 型（64bit 整数）の値を返す．値は正常時には読んだデータのバイト数（正の値），EOF では `0` ，エラーが発生した時は`-1` である．エラーの原因は `perror()` 関数で表示できる．

引数　`fildes` はオープン済みのファイルディスクリプタ，`buf` はデータを読み出すバッファ領域を指すポインタ，`size_t` 型（符号なし 64bit 整数）の `nbyte` はバッファ領域の大きさ（バイト単位）である．

使用例１　`fd` はオープン済みのファイルディスクリプタ，`buf` はバッファ用の `char` 型の大きさ 100 の配列である．`char` 型は 1 バイトなので `buf` 配列のサイズは 100 バイトになる．

1 回目の `read()` ではファイルの先頭 100 バイトを読み出す．2 回目ではファイルの 101 バイト目から 100 バイトを読み出す．3 回目ではファイルの 201 バイト目から 100 バイトを読み出す．通常 `n` は 100 になるが，EOF に達した場合は，実際に読み込めたバイト数になる．

```
char buf[100];
n = read(fd, buf, 100); // 1回目
n = read(fd, buf, 100); // 2回目
n = read(fd, buf, 100); // 3回目
```

使用例２　ループでファイルの先頭から順にデータを読み出す例である．`n` の値が `0` 以下になったら EOF かエラーなのでループを終了する．

```
while ((n=read(fd, buf, 100)) > 0) { // 読む
   ... 読んだ n バイトのデータを処理する ...
}
```

2.4　write システムコール

書き込み用にオープン済みのファイルへデータを書き込むシステムコールである．ファイルの先頭から順にデータを書き込む（シーケンシャルアクセス）．ファイルの最後に達するまでは元々あったデータを上書きする．ファイルの最後に達した場合は書き込む度にファイルの長さが長くなる．

表 2.3 lseek システムコールの第 3 引数（whence）

whence	意 味
SEEK_SET	offset はファイルの先頭からのバイト数
SEEK_CUR	offset は現在の読み書き位置からのバイト数
SEEK_END	offset はファイルの最後からのバイト数

書式 （詳しくは man 2 write で調べる.）

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

解説 write システムコールを使用するプログラムの先頭では，unistd.h をインクルードする必要がある．書き込み用にオープン済みのファイルディスクリプタを渡し，ファイルにデータを書き込む．write システムコールが返す値は，ファイルに実際に書き込んだデータのバイト数である．

引数 fildes はオープン済みのファイルディスクリプタ，buf は書き込むデータを格納したバッファ領域を指すポインタ，nbyte は書き込むデータの大きさ（バイト単位）である．

使用例 ファイルに abc の 3 バイトを書き込む．

```
char *a = "abc";
n = write(fd, a, 3);        // nが3以外ならエラーが疑われる
```

2.5 lseek システムコール

オープン済みファイルの読み書き位置を移動するシステムコールである．lseek システムコールと組み合わせることで，read，write システムコールを用いたファイルの**ランダムアクセス（直接アクセス）**が可能になる．

書式 （詳しくは man 2 lseek で調べる.）

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

解説 オープン済みファイルの読み書き位置を offset に移動する．fildes はオープン済みのファイルディスクリプタである．offset の意味は whence によって変化する．lseek システムコールは off_t 型（64bit 整数）の値を返す．値はファイルの先頭を基準にした新しい読み書き位置（単位はバイト）である．エラーが発生した時は-1 を返す．エラーの原因は perror() 関数で表示できる．表 2.3 に whence の意味をまとめる．SEEK_CUR，SEEK_END では offset が負の値になることがある．

使用例 fd はオープン済みのファイルディスクリプタとする．

```
lseek(fd, SEEK_CUR, -100);   // 現在地からファイルの先頭方向に100バイト移動する．
```

2.6　close システムコール

ファイルを閉じる．

書式（詳しくは `man 2 close` で調べる．）

```
#include <unistd.h>
int close(int fildes);
```

解説　オープン済みのファイルを閉じる．引数はオープン済みのファイルディスクリプタである．
ファイルはプログラム終了時に自動的にクローズされるのでクローズし忘れは致命的ではないが，たくさんのファイルを開くプログラムでは不要になったものをクローズしないと，同時に開くことができるファイル数の上限を超えることがある．

使用例　`fd` はオープン済みのファイルディスクリプタとする．

```
close(fd);
```

練習問題

1. **ファイルディスクリプタ**とは何か説明しなさい．
2. **ファイルの保護モード**とは何か説明しなさい．
3. open システムコールの `oflag` を記号定数のビット毎の論理和で指定できる仕組みについて考察しなさい．
4. dd コマンドの機能について調査し，実装方法について考察しなさい．

第 3 章

高水準入出力と低水準入出力

ファイルを読み書きするための機能（API：Application Program Interface）は，C 言語の入門で勉強した高水準入出力と，前の章で勉強した低水準入出力（システムコール）の 2 種類がある．この章では高水準入出力と低水準入出力の関係を学ぶ．なお，以下では高水準入出力のことを**高水準** *I/O*，低水準入出力のことを**低水準** *I/O* と呼ぶことがある．

高水準 I/O 関数は，様々な機能を持つものが豊富に用意されており，プログラマーが便利に使用することができる．一方で OS カーネルの出入り口であるシステムコールの種類は少なくし，メモリに常駐する OS カーネルをシンプルにしている．

3.1 高水準 I/O のデータ構造

高水準 I/O 関数は `fopen()` が返したファイルポインタを用いて入出力先を区別する．以下ではファイルポインタが指すデータ構造について説明する．

3.1.1 FILE 構造体

高水準 I/O と低水準 I/O の関係を図 3.1 に示す．図で `fp` は FILE 型のポインタ（**ファイルポインタ**）である．`fp` は，`fopen()` が作成し初期化した FILE 構造体を指している．FILE 構造体の内部には，管理データ，データのバッファ，ファイルディスクリプタ（`fd`）等が格納される．`fd` の値は，`fopen()` が open システムコールを実行した時に決められる．

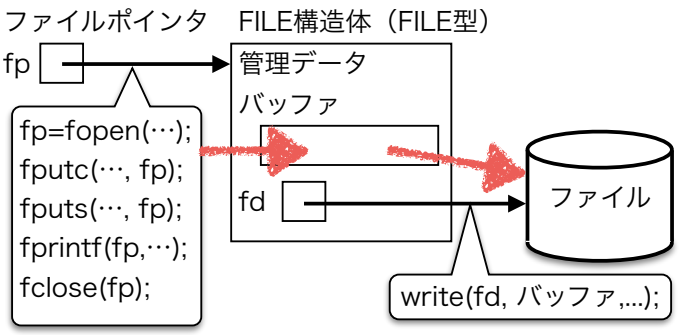


図 3.1　高水準と低水準の関係（書き込みの場合）

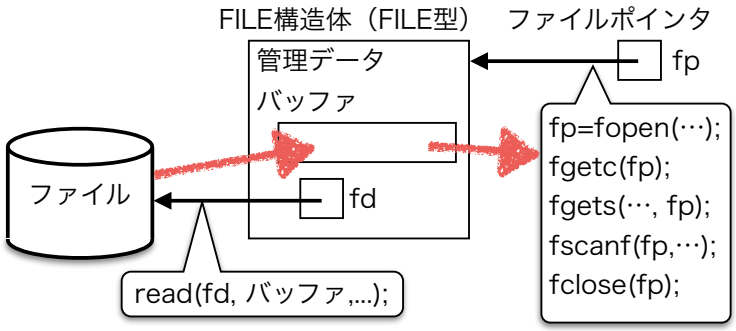


図 3.2　高水準と低水準の関係（読み込みの場合）

表 3.1　標準入出力ストリーム一覧

名称	*fd*	*fp*	通常の接続先
標準入力ストリーム	0	`stdin`	キーボード
標準出力ストリーム	1	`stdout`	ディスプレイ
標準エラー出力ストリーム	2	`stderr`	ディスプレイ

fd：ファイルディスクリプタ
fp：ファイルポインタ

3.1.2　バッファの役割

図 3.1 に示したように，高水準 I/O 関数は出力データを FILE 構造体の内部にある**バッファ**に書き込む．データはバッファがいっぱいになった時，または，その他，一定の条件を満たした時に `write()` システムコールによってファイルに書き込まれる．これは FILE 構造体のバッファにデータをためることにより、`write()` システムコールの実行回数を少なくする工夫である．一般に**システムコールは重い処理**なので，システムコールの実行回数が少なくなるような工夫が必要とされる[*1]．

図 3.2 に入力の場合を示す．入力でもシステムコールの回数が少なくなるようにバッファを使用する．入力の場合は `read()` システムコールでデータをバッファーにまとめて読み，`fgetc()` 等がバッファから入力データを必要に応じて取り出す仕組みになっている．

3.2　標準入出力

`scanf()`，`getchar()`，`printf()`，`putchar()` 等は引数にファイルポインタを持たない高水準 I/O 関数である．プログラム実行開始時にオープンされたファイルポインタ `stdin` や `stdout` を，これらは暗黙の内に使用する．これらのファイルポインタを通して読み書きするデータの流れを**標準入出力ストリーム**と呼ぶ．標準入出力ストリームの一覧と模式図を表 3.1 と図 3.3 に示す．

3.2.1　ユニファイド I/O

標準入出力ストリームを使用する `scanf()`，`getchar()`，`printf()`，`putchar()` 等は，ファイルポインタを引数に持つ `fscanf()`，`fgetc()`，`fprintf()`，`fputc()` に標準入出力ストリーム（`stdin`，`stdout` 等）を渡した場合と同じ働きをする．表 3.2 に同じ働きをする関数の対応を示す．例えば

[*1] `read()`，`write()` システムコールの重さは，一度に扱うデータの量にあまり左右されない．回数が重要である．

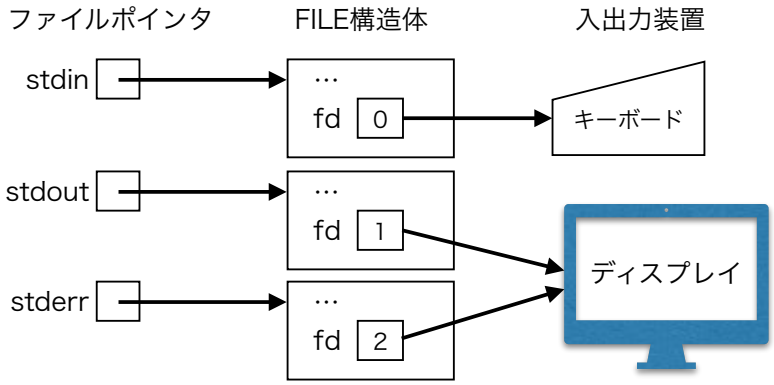


図 3.3　標準ストリームの構造

表 3.2　同じ役割をする入出力関数

標準ストリーム	同じ意味の呼出し	役割り
`scanf(...)`	`fscanf(stdin, ...)`	書式付きの入力
`getchar()`	`fgetc(stdin)`	1 文字入力
-	`fgets(stdin, ...)`	1 行入力
`printf(...)`	`fprintf(stdout, ...)`	書式付きの出力
`putchar(c)`	`fputc(c, stdout)`	1 文字出力
`puts(buf)`	`fputs(buf, stdout)`	1 行出力

`getchar()` は，`fgetc()` の引数に標準入力ストリームを表す `stdin` を渡したものと同じ働きをする．

このようにファイルへの読み書きも，キーボードやディスプレイ等の装置への入出力も，ファイルポインタやファイルディスクリプタを用いて同じ入出力関数や同じシステムコール（read/write）で行うことができる．ファイルも装置も同様に（同じ API で）扱う方式を**ユニファイド** *I/O* と呼ぶ．

3.2.2　標準入力ストリーム

ファイルポインタ `stdin` により参照され，`scanf()` や `getchar()` 等が暗黙に使用する入力ストリームである．ファイルディスクリプタ 0 はプログラム起動前にオープンされている．プログラムは起動時に `FILE` 構造体を作成し図 3.3 のように初期化する．

通常ファイルディスクリプタ 0 はキーホード用にオープンされるが，ファイルにリダイレクトすることも可能である[*2]．リダイレクトはプログラムが起動される前にシェルによって行われる．

3.2.3　標準出力ストリーム

ファイルポインタ `stdout` により参照され，`printf()` や `putchar()` 等が暗黙に使用する出力ストリームである．ファイルディスクリプタ 1 はプログラム起動前にオープンされている．プログラムは起動時に `FILE` 構造体を作成し図 3.3 のように初期化する．

通常ファイルディスクリプタ 1 はディスプレイ用にオープンされるが，ファイルにリダイレクトすることも可能である[*3]．

[*2] シェルで < を用いて標準入力ストリームをファイルにリダイレクトすることができる．
[*3] シェルで > を用いて標準出力ストリームをファイルにリダイレクトすることができる．

3.2.4　標準エラー出力ストリーム

ファイルポインタ stderr により参照され，エラーメッセージの出力用に使用するストリームである．標準出力と標準エラー出力を分けることにより，標準出力ストリームがファイルにリダイレクトされた場合でも，エラーメッセージをディスプレイに表示できる．stderr は，エラーメッセージが遅延なく表示されるように．**バッファリング**[*4]を行わない．

ファイルディスクリプタ 2 はプログラム起動前にオープンされている．プログラムは起動時に FILE 構造体を作成し図 3.3 のように初期化する．ファイルディスクリプタ 2 もファイルにリダイレクトすることが可能であるが，エラーメッセージが表示されなくなるので注意が必要である．

3.3　実装例

C--言語の高水準 I/O の実装例が, https://github.com/tctsigemura/C--/blob/v3.1.12/lib/stdio.cmm（C--言語で記述された約 400 行のプログラム）に公開してある．

3.4　低水準・高水準の性能比較

システムコールを直接使用する低水準 I/O は，上手にプログラミングすれば最高の性能を出すことができるが，下手な使い方をすると全く性能が出ない．一方で，高水準 I/O は自動的な**バッファリング**を行うので，誰が使用しても「ほどほど」の性能が出る．以下では，高水準 I/O を用いたファイルコピーコマンドと，低水準 I/O を用いたファイルコピーコマンドとの性能比較を行う．なお，以下は macOS 10.13 での実行例である．

1. プログラムを準備する
   ３年次に作成した高水準 I/O を用いたファイルコピーコマンドを mycp という名前で準備する．前回，作成した低水準 I/O を用いたファイルコピーコマンドでバッファサイズを 1 バイト（1B）にしたものを mycp2_1 という名前で準備する．バッファサイズを 1,024 バイト（1KiB）のものを mycp2_1024 という名前で準備する．
2. 大きめのファイルを作る
   以下のような操作を行う．この例では，「デタラメな内容の特殊なファイル/dev/urandom から 1,024 バイト（1KiB）ずつデータを読み込み aaa と言う名前のファイルに 1,024 バイト（1KiB）ずつ書き込む操作」を 10,240 回繰り返している．その結果，aaa と言う名前で，大きさが 10MiB（$10MiB = 1KiB \times 10Ki$）の内容がデタラメなファイルができる．

```
% dd if=/dev/urandom of=aaa bs=1024 count=10240  <-- 10MiBのファイルaaaを作る
10240+0 records in
10240+0 records out
10485760 bytes transferred in 0.073663 secs (142347719 bytes/sec)
% ls -l aaa
-rw-r--r--  1 sigemura  staff  10485760 Apr  3 12:07 aaa  <-- できている
%
```

[*4] バッファにデータをためること．

3. 実行時間の測定方法

   time コマンドを用いて実行時間を測定する．time コマンドは引数に渡されたコマンドを実行し，実行中にユーザプログラムが費やした時間（user），OS カーネルが費やした時間（system），実際の実行時間（total）を表示する．

   mycp2_1 を用いてファイル aaa を bbb にコピーする時間を測定した例を次に示す．実行時間が非常に短い場合は，コピープログラムにバグがあり何もしていない可能性があるので注意すること．

```
% rm bbb                <--- 念のため bbb を消す
rm: bbb: No such file or directory
% time ./mycp2_1 aaa bbb
./mycp2_1 aaa bbb  2.42s user 17.51s system 97% cpu 20.360 total
% cmp aaa bbb         <--- コピー結果が正常かチェック
%
```

4. 実行時間の測定

   上の方法で，mycp，mycp2_1，mycp2_1024 の三つのプログラムについて 5 回測定[*5]を行い平均を求める．5 回の平均を求めるのは測定値が誤差を含んでいるからである．なお，実行時間が長すぎて測定が困難な場合はファイルサイズを小さくしても良い．

[*5] 1 回目はイレギュラーなデータが出やすいので，実際には 6 回実行して 1 回目以外で測定する．

リスト 3.1　高水準 I/O を使用した mycp

```
#include <stdio.h>                                  // 入出力のために必要
#include <stdlib.h>                                 // exit のために必要

// err_exit : ファイルのオープンに失敗したときエラーメッセージを表示し終了
void err_exit(char *prog, char *fname) {
    fprintf(stderr,                                 // 標準エラー出力に
            "%s : can't open %s\n",                 // エラーメッセージを表示し
            prog, fname);
    exit(1);                                        // エラー終了
}

int main(int argc, char *argv[]) {
  FILE *fps;                                        // コピー元ファイル用
  FILE *fpd;                                        // コピー先ファイル用
  int  ch;                                          // コピー時使用

  if (argc != 3) {                                  // 引数の個数が予定と異なる
    fprintf(stderr,                                 // 標準エラー出力に
            "Usage: %s <srcfile> <dstfile>\n", // 使用方法を表示して
            argv[0]);
    exit(1);                                        // エラー終了
  }

  if ((fps = fopen(argv[1], "rb"))==NULL) {     // コピー元のオープン失敗
    err_exit(argv[0], argv[1]);
  }

  if ((fpd = fopen(argv[2], "wb"))==NULL) {     // コピー元のオープン失敗
    err_exit(argv[0], argv[2]);
  }

  while((ch=getc(fps)) != EOF) {                // EOF になるまで
    putc(ch ,fpd);                              // 1バイト毎のコピー
  }

  fclose(fps);                                      // ファイルクローズ
  fclose(fpd);

  return 0;                                         // 正常終了
}
```

第 4 章

ファイルシステム

ファイルは二次記憶装置（ストレージ）[*1]に格納された不揮発性のデータ記憶のことである．通常，一台の二次記憶装置には多数のファイルが記憶される．ファイルシステムとは，多数のファイルを二次記憶装置に記憶し管理するために実装された仕組みや，管理されているファイルの集合のことである．

ここでは，UNIX ファイルシステムを例に，ユーザから見たファイルシステムの構造や仕組みを紹介する．Windows や macOS[*2]等のファイルシステムも，基本的な考え方は UNIX と共通である．

4.1 ファイル木

図 4.1 にユーザから見た UNIX ファイルシステムの構造を示す．ファイルはディレクトリ（フォルダ）により階層的に管理されている．ディレクトリとファイルからなる図 4.1 のような構造をファイル木（ファイルツリー）と呼ぶ．

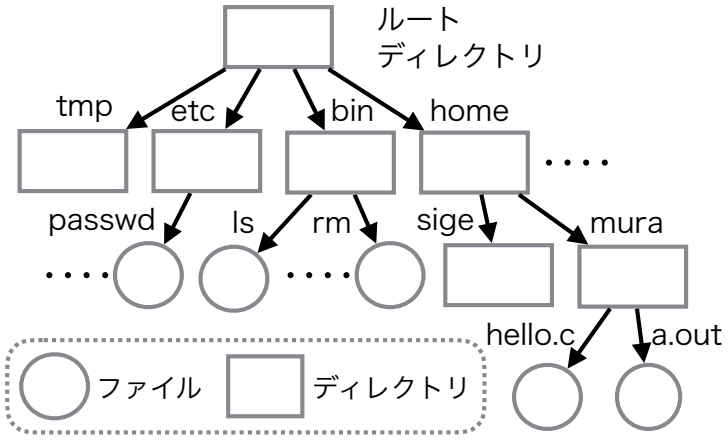


図 4.1 ユーザから見た UNIX ファイルシステムの構造

[*1] ハードディスク，USB メモリ，メモリカード，CD-ROM，SSD，磁気テープ等の外部記憶装置（補助記憶装置）のこと．通常これらは不揮発性であり，電源を切ってもデータが失われることはない．

[*2] macOS は UNIX の一種と考えてもよい．

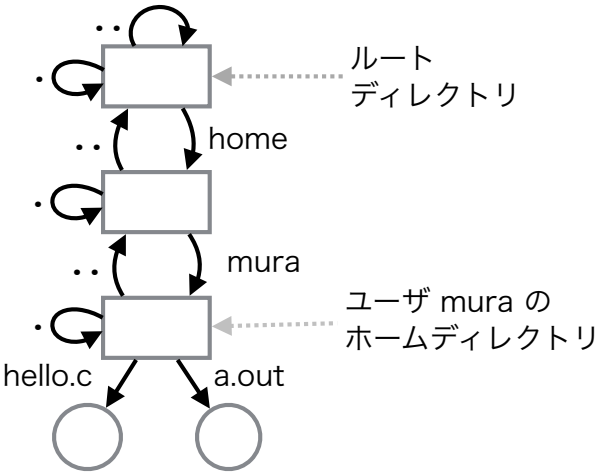


図 4.2　詳しいファイルシステムの構造

ファイル木はルートディレクトリを根（ルート）とする有向の木構造[*3]である．木構造の節点（ノード）はディレクトリであり，葉（リーフ）はファイルである．有向枝（エッジ）はリンクとも呼ばれる．リンクには必ず名前が付けられている．**ファイルの本体とリンクは独立している．**なお，UNIX ファイルシステムではディレクトリもファイルの一種と考える．

4.2　特別なディレクトリ

図 4.2 に特別なディレクトリを意識して書き直したファイル木の一部を示す．細部にこだわって描いた図 4.2 は木構造と呼ぶには相応しくないが，UNIX ファイルシステムの実装をより忠実に表現している．

親ディレクトリ

ファイルやディレクトリから見て根に近い側にあるディレクトリを親ディレクトリと呼ぶ．親ディレクトリの名前は「..」である．図 4.2 では「..」と名付けたリンクで表現している[*4]．

カレントディレクトリ

ファイルシステム内でのユーザの現在位置をカレントディレクトリと呼ぶ．カレントディレクトリの名前は「.」である．図 4.2 では「.」と名付けたリンクで表現している[*5]．

ルートディレクトリ

ファイル木の根をルートディレクトリと呼ぶ．ルートディレクトリの名前は「/」である．ルートディレクトリには親ディレクトリが存在しないので，図 4.2 では自身を親ディレクトリとしている．

ホームディレクトリ

ユーザがシステムにログインした時のデフォルトのカレントディレクトリをホームディレクトリと呼ぶ．ホームディレクトリの位置は UNIX システムの管理者が自由に決められる．図 4.2 のように`/home`

[*3] グラフ理論の木構造と似ているが閉路を許す場合もある．
[*4] UNIX ファイルシステムには実際に「..」リンクを表現するデータが格納されている．
[*5] UNIX ファイルシステムには「.」リンクを表現するデータも格納されている．

ディレクトリにユーザ名と同じ名前のディレクトリ準備し，ホームディレクトリにすることが多い[*6]．

多くの UNIX ツールではホームディレクトリのことを「~」と表記できる．しかし，これは UNIX ファイルシステムの機能ではない．単にツールが内部で名前の置換えを行ってるだけである．

4.3 パス

ファイル（ディレクトリも含む）は，あるディレクトリからそのファイルへの通り道であるパス（path）により特定できる．パスはファイル木のリンクに付いた名前を「/」で区切って書いた文字列として表現する．例えば図 4.1 右下の hello.c ファイルは，ルートディレクトリを起点とするパス/home/mura/hello.c で特定できる．同じファイルを/home/sige/../mura/./hello.c でも特定できる．

絶対パス

パスの先頭にある「/」はルートディレクリを表し，「/」で始まるパスは絶対パスと呼ばれる．絶対パスはルートディレクトリを起点にしたパスである．上記の/home/mura/hello.c 等は絶対パスの例である．

相対パス

「/」以外で始めたパスは相対パスと呼ばれる．相対パスはカレントディレクトリを起点にしたパスである．例えばカレントディレクトリが/home ディレクトリの時，図 4.1 右下の hello.c ファイルは，相対パス mura/hello.c で特定できる．カレントディレクトリが/home/sige ディレクトリの時は，相対パス../mura/hello.c で特定できる．

4.4 カレントディレクトリの変更と確認

UNIX ではプロセス（実行中のプログラム）毎にカレントディレクトリがある．プロセスのカレントディレクトリを変更しても他のターミナルやアプリに影響を与えないし，次回のログインに引継がれることもない．

cd コマンド

カレントディレクトリは cd コマンドで変更できる．

```
% cd パス        # パスのディレクトリへ移動する
```

pwd コマンド

カレントディレクトリのパスは pwd コマンドで確認できる．

```
% pwd            # カレントディレクトリのパスを表示する
```

ファイルシステムが図 4.1 の状態の時，mura ユーザが cd，pwd コマンドを操作した例をリスト 4.1 に示す．なお，mura ユーザのホームディレクトリは/home/mura とする．

[*6] macOS では，/Users ディレクトリに作られる．

リスト 4.1　cd，pwd コマンドの実行例

```
% pwd                          # カレントディレクトリは
/home/mura                     #    ホームディレクトリ
% ls                           # カレントディレクトリの
a.out   hello.c                #    ファイルを確認
% ls .
a.out   hello.c                # . を明示しても同じ結果
% cp hello.c h.c               # 相対パスだけ使用
% cp /home/mura/hello.c i.c    # 前半は絶対パス
% ls
a.out    h.c                   # コピーできている
i.c      hello.c
% cd ..                        # 親ディレクトリに移動
% pwd
/home                          # 移動できている
% cd ../bin                    # 隣のディレクトリに移動
% pwd
/bin                           # 移動できている
% cd ~                         # ホームディレクトリに移動
% pwd
/home/mura                     # 移動できている
```

4.5　リンク

ファイルを別名（別パス）で指定できると便利なことがある．UNIX ファイルシステムはファイルに別名を付ける方法を二つ準備している．

4.5.1　ハードリンク

これまでの例では一つのファイルに付き一つのリンクしか存在しなかったが，本来はいくつあっても構わない[*7]．このリンクのことを後に出てくるシンボリックリンクと区別するためにハードリンクと呼ぶ[*8]．hello.c ファイルに新たに二つリンクを追加した例を図 4.3 に示す．

ディレクトリは**リンクを格納する特殊なファイル**である．一つのディレクトリにいくつでもリンクを格納することができる．また，最初から存在したリンクと後で追加したリンクの間に優劣はない．

リンクの追加

リンクの追加は次のコマンドで行う．

```
% ln ファイルへのパス 追加するリンクのパス
```

図 4.2 の状態に二つのリンク h1.c，ex1.c を追加し，図 4.3 の状態に変える手順は次の通りである．なお，リンク ex1.c を追加するより前に SysPro ディレクトリを作っておく必要がある．

[*7] グラフ理論の木構造とはかけ離れていくが。。。

[*8] 単にリンクと呼ぶ時はハードリンクのことを指している．

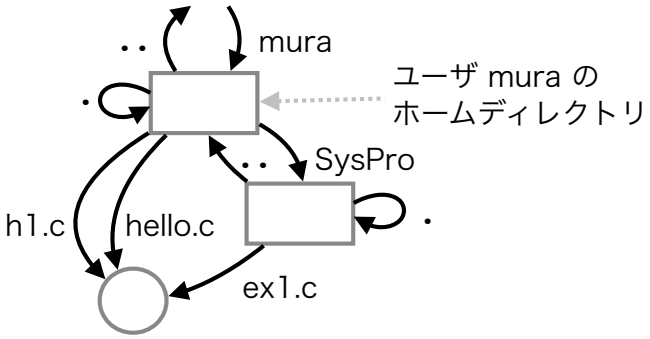


図 4.3 複数のリンクを持つファイル

```
% pwd
/home/mura                  # カレントディレクトリはここ
% ln hello.c h1.c           # hello.c にリンク h1.c を追加
% mkdir SysPro              # SysPro ディレクトリを作る
% ln hello.c SysPro/ex1.c   # リンク ex1.c を追加
% cat h1.c                  # hello.c の内容が表示される
% cat SysPro/ex1.c          # hello.c の内容が表示される
```

リンクの削除

リンクの削除は rm コマンドで行う.

```
% rm ファイルへのパス
```

rm コマンドはファイルを削除するコマンドと考えてきたが，正確にはリンクを削除するコマンドである．リンクを削除した結果，リンクを一つも持たなくなったファイル本体は削除される．**このようにリンクとファイル本体は別のものである．**図 4.3 の状態から図 4.2 の状態に戻す手順を次に示す.

```
% pwd
/home/mura          # カレントディレクトリはここ
% rm h1.c            # リンク h1.c を削除
% rm SysPro/ex1.c    # リンク ex1.c を削除
% rmdir SysPro       # SysPro ディレクトリを削除
```

4.5.2 シンボリックリンク

シンボリックリンク[*9]はパス（文字列）を格納した特殊なファイルである．オペレーティングシステムがパスを解析する途中でシンボリックリンクを見つけると，パス中のシンボリックリンク名をシンボリックリンクの内容で置き換えてからパスの解析を続ける．ハードリンクは同一の二次記憶装置内のファイルしかリンクできないが，シンボリックリンクにはこのような制約はない.

シンボリックリンクにはどんなパスでも書き込むことができる．まずリンク切れのシンボリックを

[*9] シンボリックリンクのことをソフトリンクと呼ぶこともある.

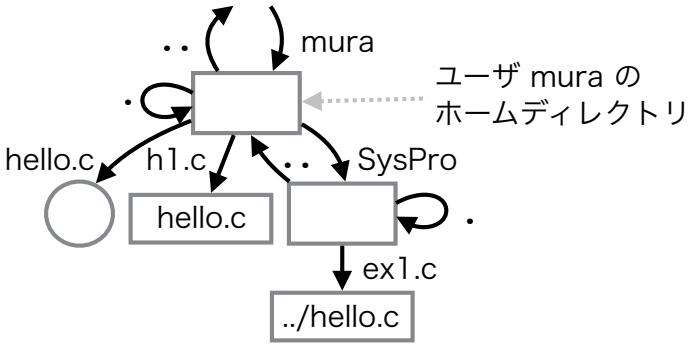


図 4.4　シンボリックリンク

作っておき，後にファイル本体を作ることも許される．一方でリンク先のファイルが削除された場合はリンク切れ状態になる．これを応用すると，置き換わったファイルを同じシンボリックリンクで参照し続けることができる．**リンク先ファイルが置換わることがシンボリックリンクの特徴である．**

シンボリックリンクの使用例

図 4.4 にシンボリックリンクを使用した例を示す．この例は図 4.3 のハードリンクをシンボリックリンクに置き換えたものである．この図では，シンボリックリンクを内部にパスを書いた長方形で表現している．シンボリックリンクもファイルの一種なので，ディレクトリからハードリンクによって接続されている．

ユーザ mura のホームディレクトリからの相対パス `h1.c` を用いてファイルを指定した場合，パスはシンボリックリンクの内容と置き換えられ `hello.c` になる．同様に相対パス `SysPro/ex1.c` は，`ex1.c` がシンボリックリンク名なので `SysPro/../hello.c` に置き換えられる．どちらの場合もホームディレクトリ直下の `hello.c` ファイルを指定したことになる．

シンボリックリンクの作成

シンボリックリンクは `ln` コマンドに`-s` オプションを付けたものを用いて作成する．

```
% ln -s リンクに書込むパス 作成するリンクのパス
```

図 4.2 の状態に二つのシンボリックリンク `h1.c`，`ex1.c` を追加し，図 4.4 の状態に変える手順は次の通りである．シンボリックリンク `ex1.c` に書込むパスは`../hello.c` になる．**シンボリックリンクに書込むパスにはシンボリックリンクが存在するディレクトリからの相対パスを用いる**[*10]**．**

```
% pwd
/home/mura                          # カレントディレクトリはここ
% ln -s hello.c h1.c                # リンク h1.c を作成
% mkdir SysPro                      # SysPro ディレクトリを作る
% ln -s ../hello.c SysPro/ex1.c     # ex1.c を作成
% cat h1.c                          # hello.c の内容が表示される
% cat SysPro/ex1.c                  # hello.c の内容が表示される
```

[*10] 絶対パスを書込むことも可能であるが普通は相対パスを用いる．例えばシステム管理者がユーザ `mura` のホームディレクトリの位置を変更しても，相対パスを用いておけばリンク切れにならない．

シンボリックリンクの削除

シンボリックリンクの削除も rm コマンドで行う.

```
% rm シンボリックリンクのパス
```

図 4.4 の状態から図 4.2 の状態に戻す手順を次に示す.

```
% pwd
/home/mura          # カレントディレクトリはここ
% rm h1.c            # リンク h1.c を削除
% rm SysPro/ex1.c    # リンク ex1.c を削除
% rmdir SysPro       # SysPro ディレクトリを削除
```

4.6 ファイルの属性

UNIX の各ファイル[*11]は，ファイル本体にデータだけでなく幾つかの属性情報（メタデータ）を持っている．ファイル名を属性情報の一部にしている OS もあるが，UNIX ではファイル名はファイル本体ではなくリンクに付属するので属性には含まれない.

4.6.1 主な属性

UNIX で用いられる主な属性情報は次の通りである．これらは stat コマンドや ls コマンドで表示できる.

種類 普通のファイル，ディレクトリ，シンボリックリンク等の区別.

保護モード open システムコールで紹介した rwxrwxrwx.

リンク数 ファイルを指しているハードリンクの数．リンク数が 0 になるとファイル本体が削除される．（例えば，図 4.3 の hello.c ファイルの場合は 3 になる）

所有者 所有者のユーザ番号.

グループ 属するグループのグループ番号.

ファイルサイズ ファイルの大きさ（バイト単位）.

最終参照日時 最後にアクセスした時刻.

最終変更日時 内容を最後に変更した時刻.

最終属性変更時刻 属性を最後に変更した時刻.

4.6.2 属性の表示方法

属性を表示する専用コマンドは stat であるが，ここでは，より手軽に使用できる ls コマンドを紹介する．ls コマンドは-l オプションを付けて実行すると，ファイルの属性情報の一部を表示する[*12]．以下に ls コマンドを実行した例を示す.

```
% ls -l a.txt
-rw-r--r--  1 mura  staff 10 May  1 18:18 a.txt
```

[*11] ディレクトリやシンボリックリンクも含む

[*12] より多くの属性情報を知りたい時は，stat コマンドを用いる.

表 4.1　chmod コマンドの引数の意味

文字	意味
u	所有者（ユーザ）
g	グループ
o	その他のユーザ
+	権利を与える
-	権利を取上げる
r	読出し
w	書込み
x	実行

実行例の表示は以下の意味を持っている．

ファイルの種類　一文字目の「-」はファイルが普通のファイルであることを表している．一文字目が「d」はディレクトリであること，「l」はシンボリックリンクであることを表す．

ファイルの保護モード　open システムコールで紹介したもの（rwxrwxrwx）．

リンク数　1 はリンク数が 1 であることを表している．

所有者　mura はファイルの所有者がユーザ mura であることを表している．メタ情報の内部表現はユーザ番号であるが，ls コマンドがユーザ名に変換して表示している．

グループ　staff はファイルがグループ staff に属することを表している．メタ情報の内部表現はグループ番号であるが，ls コマンドがグループ名に変換して表示している．

ファイルサイズ　10 はファイルのサイズが 10 バイトであることを表している．

最終変更日時　May 1 18:18 はファイルの最終変更日時である．

パス　a.txt はファイルへ到達するために使用したパスである．**パス名（ファイル名）はファイルの属性ではない．**

4.6.3　属性の変更方法

ファイルの保護モードは一般ユーザも変更する機会が多い．ファイルの保護モードの変更には chmod コマンドを用いる．以下に書式を，リスト 4.2 に使用例を示す．

```
% chmod OOO ファイル...      # 書式１
% chmod ugo+rwx ファイル...  # 書式２
% chmod ugo-rwx ファイル...  # 書式３
```

書式１　OOO は 3 桁の 8 進数である．8 進数で保護モードを指定する．8 進数の値は open システムコールの書式 2 と同じである．

書式２，３　ugo+-rwx の文字を組合せて保護モードの変更方法を記述する．各文字の意味は表 4.1 の通りである。例えば，所有者とグループに書込み権と実行権を与える場合なら ug+wx のように書く．その他のユーザの読出し権を取上げるなら o-r のように書く．

以下に chmod コマンドの使用例を示す．

リスト 4.2 chmod コマンドの使用例

```
% ls -l a.txt
-rw-r--r--  1 mura  staff 10 May  1 19:42 a.txt
% chmod 640 a.txt
% ls -l a.txt
-rw-r-----  1 mura  staff 10 May  1 19:42 a.txt
% chmod g+w a.txt
% ls -l a.txt
-rw-rw----  1 mura  staff 10 May  1 19:42 a.txt
% chmod g-r a.txt
% ls -l a.txt
-rw--w----  1 mura  staff 10 May  1 19:42 a.txt
```

第 5 章

ファイル操作システムコール

この章ではファイルを操作するシステムコールの中で重要なものについて学ぶ．前の章で使用したユーティリティコマンド（ln，rm，chmod 等）が内部で使用しているシステムコールである．

5.1 unlink システムコール

ファイルを削除するシステムコールである．rm コマンドは，このシステムコールを利用している．「ファイルの削除」は正確にはリンク（名前）の削除の意味である．ファイルはリンクを一つも持たなくなった時に削除される．シンボリックリンク等の削除にも使用できるが，ディレクトリの削除には使用できない．

書式 path 引数でリンクのパスを一つ指定する．

```
#include <unistd.h>
int unlink(const char *path);
```

解説 unink システムコールを使用するプログラムは unistd.h をインクルードする必要がある．unlink システムコールは，正常時に 0 ，エラー発生時に -1 を返す．エラー原因は perror() 関数で表示できる．

引数 path は削除するリンク（ファイル）のパスを表す文字列である．

使用例 "a.txt"ファイルを削除する例を示す．

```
// ファイルの削除
if (unlink("a.txt")<0) {   // "a.txt" 削除
  perror("a.txt");         // エラー原因表示
  exit(1);                 // エラー終了
}
```

5.2 mkdir システムコール

ディレクトリ（フォルダ）作成するシステムコールである．mkdir コマンドは，このシステムコールを使用している．

書式　path，mode の二つの引数で新規ディレクトリのパスと保護モードを指定する．

```
#include <sys/stat.h>
int mkdir(const char *path, mode_t mode);
```

解説　mkdir システムコールを使用するプログラムは sys/stat.h をインクルードする必要がある．mkdir システムコールは，正常時に 0 ，エラー発生時に -1 を返す．エラー原因は perror() 関数で表示できる．mkdir システムコールはパスに含まれる途中のディレクトリは作らない．途中のディレクトリは予め作成しておく必要がある．

引数　path は新規作成するディレクトリのパス，mode は新しいディレクトリの保護モード（rwxrwxrwx）である．

使用例　"newdir"ディレクトリを作成する例である．

```
// ディレクトリの作成
if (mkdir("newdir", 0755)<0) {   // "newdir" を rwxr-xr-x で作成
  perror("newdir");              // エラー原因表示
  exit(1);                       // エラー終了
}
```

5.3　rmdir システムコール

ディレクトリを削除するシステムコールである．rmdir コマンドは，このシステムコールを利用している．**空でないディレクトリを削除することはできない．**

書式　path 引数でディレクトリのパスを一つ指定する．

```
#include <unistd.h>
int rmdir(const char *path);
```

解説　rmdir システムコールを使用するプログラムは unistd.h をインクルードする必要がある．rmdir システムコールは，正常時に 0 ，エラー発生時に -1 を返す．エラー原因は perror() 関数で表示できる．

引数　path は削除するディレクトリのパスである．

使用例　"newdir"と名付けられたディレクトリを削除する例である．

```
// ディレクトリの削除
if (rmdir("newdir")<0) {  // "newdir" 削除
  perror("newdir");       // エラー原因表示
  exit(1);                // エラー終了
}
```

5.4　link システムコール

リンク（ハードリンク）を作成するシステムコールである．ln コマンドは，ハードリンクを作るとき，このシステムコールを利用している．

書式 既に存在するリンクのパスと新しく作るリンクのパスを指定する．

```
#include <unistd.h>
int link(const char *oldpath, const char *newpath);
```

解説 link システムコールを使用するプログラムは unistd.h をインクルードする必要がある．link システムコールは，正常時に 0 ，エラー発生時に -1 を返す．エラー原因は perror() 関数で表示できる．

引数 oldpath はもともと存在するリンクを表すパス，newpath は新しく作るリンクのパスである．

使用例１ ファイルにリンク"b.txt"を追加する例である．

```
// ハードリンクの作成
if (link("a.txt", "b.txt")<0) { // リンク"b.txt"を作る
  perror("link");               // "a.txt"と"b.txt"のどちらが原因か不明なので
  exit(1);                      // エラー終了
}
```

使用例２ unlink システムコールと組合せてファイルの移動（ファイル名の変更）に応用した例である．リンク"a.txt"で参照されていたファイルは，リンク"b.txt"で参照されるように変更される．

```
unlink("b.txt");                // 念のため"b.txt"を消す（エラーは無視）
if (link("a.txt", "b.txt")<0) { // リンク"b.txt"を作る
  ... エラー処理 ...
}
if (unlink("a.txt")<0) {        // リンク"a.txt"を消す．
    ... エラー処理 ...
}
```

5.5 symlink システムコール

シンボリックリンクを作成するシステムコールである．ln コマンドは，シンボリックリンクを作るとき（-s オプション使用時），このシステムコールを利用している．

書式 新規作成するシンボリックリンクのパスと，シンボリックリンクに書き込むパスを指定する．

```
#include <unistd.h>
int symlink(const char *path1, const char *path2);
```

解説 symlink システムコールを使用するプログラムは unistd.h をインクルードする必要がある．symlink システムコールは，正常時に 0 ，エラー発生時に -1 を返す．エラー原因は perror() 関数で表示できる．

引数 path1 はシンボリックリンクに書き込むパスである．path2 は新規に作成するシンボリックリンク自身のパスである．

使用例 内容が"a.txt"のシンボリックリンクを"b.txt"という名前で作る．シンボリックリンクの内容はチェックされない（リンク切れ状態でも良い）ので，エラーが発生した場合，原因は"b.txt"である．エラーメッセージには"b.txt"を含める．

```
// シンボリックリンクの作成
if (symlink("a.txt", "b.txt")<0) { // リンク"b.txt"を作る
  perror("b.txt");                 // エラー原因は必ず"b.txt"
  exit(1);                         // エラー終了
}
```

5.6　rename システムコール

ファイルの移動（ファイル名の変更）を行うシステムコールである．mv コマンドは，このシステムコールを利用している．

書式　新旧二つのパスを指定する．

```
#include <stdio.h>
int rename(const char *from, const char *to);
```

解説　rename システムコールを使用するプログラムは stdio.h をインクルードする必要がある．rename システムコールは，正常時に 0 ，エラー発生時に -1 を返す．エラー原因は perror() 関数で表示できる．

引数　from は古いパス to は移動後の新しいパスである．from で参照されていたファイルが to で参照されるようになる．

使用例　"a.txt"のパスを"b.txt"に変更する例である．

```
// ファイルの移動
if (rename("a.txt", "b.txt")<0) { // "a.txt" を "b.txt" に変更
  perror("rename");               // エラー原因がどっちのパスか不明
  exit(1);                        // エラー終了
}
```

5.7　chmod（lchmod）システムコール

ファイルの保護モードを変更するシステムコールである．chmod コマンドは，このシステムコールを利用している．

書式　ファイルのパスと新しい保護モードを指定する．

```
#include <sys/stat.h>
#include <unistd.h>
int chmod(const char *path, mode_t mode);
int lchmod(const char *path, mode_t mode);
```

解説　chmod システムコールを使用するプログラムは sys/stat.h をインクルードする必要がある．chmod システムコールは path で指定されるファイルの保護モードを変更する．シンボリックリンクが指定された場合，lchmod システムコールはシンボリックリンクが指すファイルではなく，シ

ンボリックリンクの保護モードを変更する．chmod システムコールは，正常時に 0 ，エラー発生時に -1 を返す．エラー原因は perror() 関数で表示できる．

引数 path は対象となるファイルのパスである．mode は新しい保護モード (rwxrwxrwx）である．保護モードの意味は open システムコールの章に解説がある．

使用例 "a.txt"の保護モードを rw-r--r--に変更する例である．

```
// ファイルモードの変更
if (chmod("a.txt", 0644)<0) { // ファイル"a.txt"のモードをrw-r--r--に変更
  perror("a.txt");            // エラー原因を表示
  exit(1);                    // エラー終了
}
```

5.8 readlink システムコール

readlink システムコールはシンボリックリンクに書き込まれている内容（パス）を読み出す．ls コマンドはシンボリックリンクの内容を表示するために，このシステムコールを利用している．

書式 シンボリックリンクを示すパスと，内容を読み出すバッファを指定する．

```
#include <unistd.h>
int readlink(const char *path, char *buf, size_t bufsize);
```

解説 readlink システムコールを使用するプログラムは unistd.h をインクルードする必要がある．readlink システムコールは path で指定されるシンボリックの内容を buf に読み出す．path はシンボリックリンクを示すものでなければならない．readlink システムコールは，正常時に読み出した文字数を，エラー発生時に-1 を返す．エラー原因は perror() 関数で表示できる．読み出した文字列は'\0' で終端**されない**ので注意を要する．

引数 path は目的のシンボリックリンクのパスである．buf は内容を読み出す領域（バッファ）のポインタ，bufsize は領域のサイズである．

使用例 シンボリックリンク"b.txt"の内容を読み出し表示する例である．読み出した文字列を'\0' で終端することを見越して，バッファの大きさ（100）より小さい領域サイズ（99）を指定している．

```
// シンボリックリンクの読み出し
char *name = "b.txt";
char buf[100];
int n = readlink(name, buf, 99); // シンボリックリンクの内容をbufに読み出す
if (n<0) {                       // エラーチェック
  perror(name);                  // エラー原因を表示
  exit(1);                       // エラー終了
}
buf[n]='\0';                     // C言語型の文字列として完成させる
printf("%s -> %s\n", name, buf); // ls -l 風に表示
```

参考：strtol 関数

chmod プログラムを作成するためには，保護モードをコマンド行引数から入力する必要がある．

書式：`chmod mode file ...`

mode を 8 進数で指定する場合，8 進数を表現する文字列から整数値（int 型）に変換しなければならない．strtol 関数は 8 進数を整数値に変換するために使用できる．簡易版の chmod プログラムを作るために必要な strtol 関数の使用例を以下に示す．

（詳細はオンラインマニュアル「`man 3 strtol`」で調べること．）

```
#include <stdlib.h>
  ...
  char *ptr;
  char *ostr = argv[1];                // 入力の8進数文字列
  int mod;

  mod = strtol(ostr, &ptr, 8);         // 8進数文字列の値を整数で求める
  if (*ostr=='\0' || *ptr!='\0') {
    fprintf(stderr, "\'%s\' : 8進数の形式が不正\n", ostr);
    return 1;
  }
  printf("%d,%x,%o\n", mod, mod, mod); // 10進，16進，8進で表示
  ...

// 実行例
% ./a.out 777
511,1ff,777
```

第 6 章

プロセスとジョブ

6.1 プロセス

皆さんは，これまで PC を使用してきた経験から，図 6.1 に示すように，同時に複数のプログラムが PC で実行されていることを理解しているだろう．また，同じプログラムが複数のウインドで同時に実行されることがあることも理解しているだろう[*1]．

つまり，一連の機械語命令である**プログラム**だけでなく，実行中のプログラムのインスタンスも意識する必要がある．実行中のプログラムのインスタンスのことを**プロセス**と呼ぶ．

プロセス ＝ 実行中のプログラム

6.1.1 プロセスの構造

図 6.2 にプロセスの構造を模式的に示す．プロセスは，プロセス名（プロセス番号）等の「プロセスの情報」や，前回プロセスを実行中だった時の「CPU の状態」を保存する領域（仮想 CPU）とそのプロセス専用の「メモリ空間」（仮想メモリ）を持つ．一つ一つのプロセスが一台のコンピュータに相当するるものを備えており，プロセスを一台の**仮想コンピュータ**と考えることもできる．

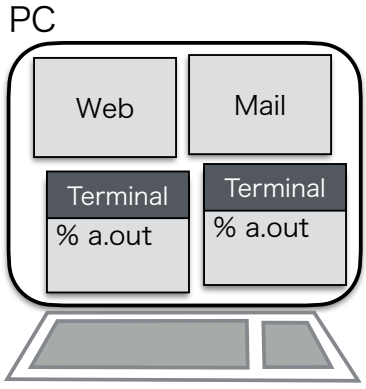


図 6.1　PC 上の複数プログラム

[*1] 図 6.1 の例では二つのターミナル（Terminal）夫々の中で，合計二つの a.out が同時に実行されている．

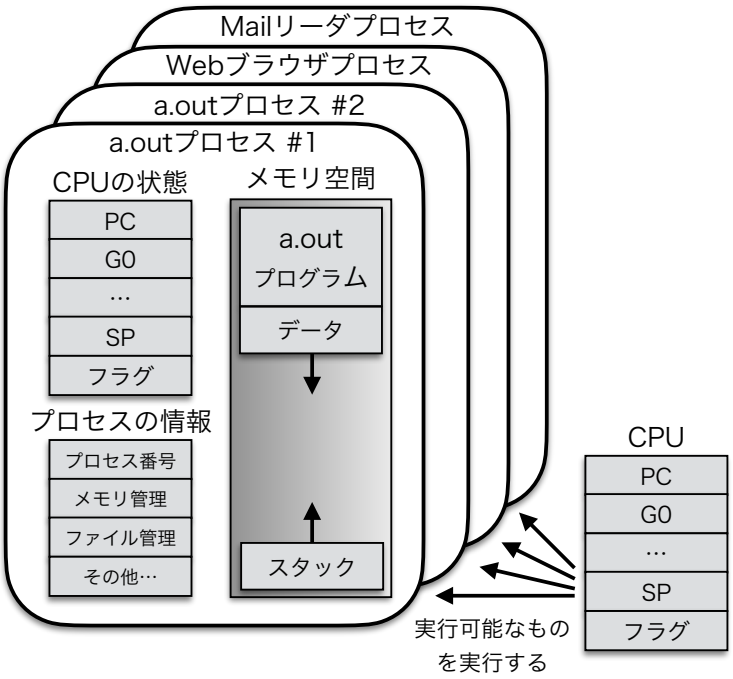


図 6.2　プロセスの構造

プロセス ＝ 仮想コンピュータ

PC 内には多数のプロセスが同時に存在し，これらの中から実行可能なもの[*2]を一つ[*3]選択し実物の CPU（実 CPU）で実行する．実行するプロセスを短時間に次々切替えることにより，複数のプロセスが同時に実行されているように見せかけることもできる．

6.1.2　プロセス関連の UNIX コマンド

ps コマンド

PC 内で実行中のプロセスの一覧表を状態付きで表示するコマンドである．macOS でターミナルを二つ開いた状態で，一方のターミナルで vi（vim）を起動し，もう一方のターミナルで ps コマンドを実行した例をリスト 6.1 に示す．

全てのプログラムはプロセスとして実行される．ターミナルや ps コマンド自身もプロセスとして実行されるが，macOS の標準では，これらは表示されない．`-zsh` はターミナルに入力されたコマンドを解釈して実行するシェルである．表示内容の意味は表 6.1[*4]の通りである．

制御端末はプロセスがどのターミナルで実行されているかを示す．ターミナルの名前はリスト 6.1 の最後の部分のように，ターミナルで tty コマンドを実行すると確認できる．

[*2] 例えば，ユーザのキー操作待ちのものは実行不可能なプロセスである．
[*3] CPU（または，コア）が複数ある場合は一つとは限らない．
[*4] リスト 6.1 の実行例に含まれていない項目も含んでいる．

リスト 6.1 ps の実行例

```
% ps
  PID TTY           TIME CMD
27828 ttys000    0:00.51 -zsh
46471 ttys000    0:00.62 vi chap6s.tex
38060 ttys001    0:00.10 -zsh
% tty
/dev/ttys001
%
```

表 6.1 ps コマンドの表示内容

欄	意味
PID	プロセス番号
TTY	制御端末
TT	TTY の簡易表示
TIME	プロセスがこれまでに CPU を使用した時間
CMD	プロセスを起動したコマンド
COMMAND	CMD と同じ
USER	誰の権限で実行しているか
%CPU	CPU の利用率
%MEM	メモリの利用率
VSZ	仮想記憶サイズ（KiB 単位）
RSS	常駐セット（KiB 単位）
STAT	プロセスの状態
STARTED	プロセスの開始時刻

表 6.2 ps コマンドのオプション（一部）

オプション	意味
u	ロングフォーマットで表示（詳しい表示）
a	他人のプロセスも表示
x	制御端末を持たないものも表示

ps コマンドのオプション

表 6.2 に ps コマンドのオプションの一部を[*5] リスト 6.2 にオプションを用いた時の実行例を示す．ps u を実行するとリスト 6.1 と同じ三つのプロセスに付いて，より詳しい表示がされた．表示内容の意味は表 6.1 の通りである．プロセスの状態は，表 6.3 の二文字を組合せて表現される．例えば，リスト 6.2 で vi は S+ なので，フォアグラウンドで実行中に短期間のスリープをしていることが分かる．

ps au を実行すると root[*6]のプロセスも表示されている．この実行例では ps コマンド自身も表示されている．ps コマンドは root のプロセスとして実行されていたことが分かる．

[*5] 最近の UNIX ではオプションが大きく変更されている．ここでは簡単で分かりやすい古いオプションの使用例を示す．

[*6] UNIX 系のオペレーティングシステムでは，システム管理者の名前が root である．

リスト 6.2　ps コマンドのオプション付き実行例

```
% ps u
USER       PID  %CPU %MEM      VSZ    RSS   TT  STAT STARTED      TIME COMMAND
sigemura 38060   0.5  0.0 408824272   5680 s001  S    11:03AM   0:00.15 -zsh
sigemura 46471   0.0  0.1 408816704   9264 s000  S+   11:33AM   0:00.74 vi chap6s.tex
sigemura 27828   0.0  0.0 409086416   7472 s000  S    10:35AM   0:00.51 -zsh
% ps au
USER       PID  %CPU %MEM      VSZ    RSS   TT  STAT STARTED      TIME COMMAND
root     60998   0.0  0.0 408766112   1808 s001  R+   12:08PM   0:00.01 ps au
sigemura 46471   0.0  0.1 408816704   9264 s000  S+   11:33AM   0:00.74 vi chap6s.tex
sigemura 38060   0.0  0.0 408824272   5680 s001  S    11:03AM   0:00.15 -zsh
root     38059   0.0  0.0 408646464   4128 s001  Ss   11:03AM   0:00.02 login -pfl sigemu
sigemura 27828   0.0  0.0 409086416   7472 s000  S    10:35AM   0:00.51 -zsh
root     27827   0.0  0.0 408646464   4464 s000  Ss   10:35AM   0:00.02 login -pf sigemur
% ps aux
USER               PID  %CPU %MEM      VSZ    RSS   TT  STAT STARTED      TIME COMMAND
_windowserver      181  30.5  1.5 410618752 251024   ??  Rs   23Mar23  64:38.93 /System/L
sigemura         43374  10.5  0.6 410251920  97664   ??  R    11:16AM   5:13.83 /Applicat
sigemura           846   4.3  0.6 410068912 102544   ??  S    24Mar23  24:58.94 /System/A
sigemura         12009   3.1  1.5 410249072 245024   ??  S    24Mar23   1:35.29 /System/A
root               418   1.1  0.1 409363184  13936   ??  Rs   23Mar23   3:45.99 /Library/
root               153   1.1  0.1 408268064  16240   ??  Ss   23Mar23   1:28.69 /System/L
...600行程度続く...
%
```

表 6.3　ps コマンド STAT 表示の意

一文字目	意味
I	20 秒以上 sleep している
S	20 秒未満の sleep
R	実行可能
T	一時停止状態（stop，Ctrl-Z）
Z	ゾンビ（Zombi）
...	

+

ニ文字目	意味
+	フォアグラウンド
s	セッションリーダ
...	

ps aux を実行すると全ユーザの全プロセスが表示される．TT が「??」のプロセスが制御端末を持たないものである．macOS では色々な権限で 600 程度のプロセスが実行されていることが分かる．

kill コマンド

プロセスにシグナル（ソフトウェア割込み）を送るコマンドである．通常，プロセスは予期しないシグナルを受取ると終了する．kill コマンドはシグナル（省略可能）とプロセス番号を引数にする．使用できるシグナルの一部を表 6.4 に，使用例をリスト 6.3 に示す[*7]．

[*7] コマンド行から簡単に起動できて，しばらく実行し続ける sleep を例にする．

表 6.4 kill コマンドのシグナル名と番号（一部）

番号	名前	意味
2	INT	終了（Ctrl-C と同じ）
9	KILL	強制終了
15	TERM	終了（オプション無しと同じ）
18	TSTP	一時停止（Ctrl-Z と同じ）
19	CONT	一時停止後の再開

リスト 6.3 kill コマンドの使用例

```
1  % sleep 10000 &                          <--- サンプル用プロセスを起動
2  [1] 75868
3  % ps
4    PID TTY           TIME CMD
5  38060 ttys001    0:00.22 -zsh
6  75868 ttys001    0:00.01 sleep 10000     <--- PIDが分かる
7  % kill 75868                             <--- プロセスを終了させる
8  [1]  + terminated  sleep 10000
9  % sleep 10000 &                          <--- 新しいサンプル用プロセスを起動
10 [1] 75871                                <--- 実はPIDはここでも分かる
11 % kill -TSTP 75871                       <--- プロセスを一時停止
12 [1]  + suspended  sleep 10000
13 % ps
14   PID TTY           TIME CMD
15 38060 ttys001    0:00.26 -zsh
16 75871 ttys001    0:00.00 sleep 10000     <--- プロセスは存在している
17 % kill -CONT 75871                       <--- プロセスを再開させる
18 % kill 75871                             <--- プロセスを終了させる
19 [1]  + terminated  sleep 10000
```

6.2 ジョブ

ジョブはユーザの視点からはプロセスとよく似たものに見える．ジョブはシェルが管理するプロセスのグループである．シェルは一度に起動されたひとまとまりのプロセスを一つのジョブとして扱う．リスト 6.4 に例を示す．

6.2.1 ジョブの種類

ジョブにはフォアグラウンド・ジョブとバックグラウンド・ジョブの 2 種類がある．夫々、次のような特徴がある．実行例はリスト 6.5 の通りである．

フォアグラウンド・ジョブ シェルがジョブの終了を待つ．終了したらプロンプトが表示される．

バックグラウンド・ジョブ コマンドの最後に&を付けて実行する．シェルがジョブの終了を待たない．

ジョブが終了していなくてもプロンプトが表示される．次のジョブと並列実行ができる．

リスト 6.4　ジョブの例

```
# 通常のコマンド実行
% vi hello.c                                    <-- 1プロセスが1ジョブ

# パイプを使用しファイルサイズ順にソートして表示
% ls -l | sort -n --key=5                       <-- 2プロセスが1ジョブ

# 二つのコマンド（ジョブ）を順次実行
% touch a.txt; chmod 777 a.txt                  <-- 2ジョブ

# 二つのコマンド（ジョブ）を並列実行
% touch a.txt & touch b.txt                     <-- 2ジョブ
```

リスト 6.5　フォアグラウンド・バックグラウンド

```
# フォアグラウンド・ジョブの場合
% sleep 1000                                    <-- ジョブの開始
^C                                              <-- 実行を終了させる
%                                               <-- 次のコマンド入力が可能になる

# バックグラウンド・ジョブの場合
%  sleep 1000 &                                 <-- & 付きで開始
[1] 6123                                        <-- ジョブ番号=1，PID=6123
%                                               <-- すぐに次のコマンド入力が可能
```

6.2.2　ジョブ制御

シェルと制御端末はジョブに対して働きかけをすることができる．

Ctrl-C　フォアグラウンド・ジョブに INT シグナルを送る．（通常，ジョブは終了する．）

Ctrl-Z　フォアグラウンド・ジョブに TSTP シグナルを送る．（ジョブは一時停止状態になる．）

jobs　そのシェルが管理しているジョブの一覧を表示する．

fg　バックグラウンド・ジョブや一時停止中のジョブをフォアグラウンドに切替える．ジョブ番号を指定することもできる．

bg　一時停止状態のジョブをバックグラウンドで再開する．ジョブ番号を指定することもできる．

リスト 6.6 にジョブ制御の使用例を示す．

リスト 6.6 ジョブ制御の使用例

```
% sleep 2000                                  <-- フォアグラウンドで起動
^Z
zsh: suspended  sleep 2000                   <-- 一時停止した
% bg                                          <-- バックグラウンドで再開
[1]  + continued  sleep 2000
% sleep 1000 &                                <-- 新しくバックグラウンドで起動
[2] 80228
% jobs                                        <-- 実行中のジョブを確認
[1]  - running    sleep 2000
[2]  + running    sleep 1000
% fg %1                                       <-- 1番をフォアグラウンドに変更
[1]  - running    sleep 2000                 <-- フォアグラウンドに変更された
^C                                           <-- Ctrl-C で終了
% jobs
[2]  + running    sleep 1000                 <-- 2番だけになった
```

第 7 章

シグナル

前の章では，プロセスを終了させたり一時停止させたりするためにシグナルが利用できることを学んだ．シグナルは動作中のプロセスに非同期的に（いつでも関係なく）イベントの発生を知らせる汎用的な仕組みである．JOB 制御やアプリケーションの強制終了，サーバプロセスの再起動などに使用される．

7.1 シグナルの特徴と使用目的

シグナルは，プロセスにイベントの発生を知らせるソフトウェア割り込みである．割込みのようなものであるから，プロセスはシグナルがいつ発生するか予測できない．シグナルは以下のような場合にイベントをプロセスに通知するために使用される．

1. **プロセスや *OS* がプロセスにイベントを通知する場合**
   kill コマンドを用いてシグナルをプロセスに送信する場合が一つの例である．kill コマンドの実行は，kill プログラムがプロセスとして実行されるのであるから，kill プロセスから目的のプロセスにシグナルを送っていることになる．
   もう一つの例はターミナルで Ctrl-C や Ctrl-Z を押した場合である．この場合は OS がターミナルに属するフォアグラウンドプロセスにシグナルを送る．
2. **プロセス自身の異常を通知する場合**
   0 での割り算を実行したり[*1]，ポインタの初期化をし忘れて異常なアドレスをアクセスしたり[*2]したことをプロセス自身に通知する．通常，この通知を受取るとプロセスは終了する[*3]
3. **プロセスが予約した時刻になった場合**
   プロセスは一定時間後にシグナルを発生するように予約できる．時間になるとアラームシグナル（`SIGALRM`）が発生する．

[*1] Floating point exception シグナル（`SIGFPE`）が発生する．
[*2] Segmentation fault シグナル（`SIGSEG`）が発生する．
[*3] 終了時に「`segmentation fault`」等が表示される．

表 7.1　よく使用されるシグナルの一覧

番号	記号名	デフォルト	説明
1	SIGHUP	終了	プロセスが終了していないときログアウトした.
2	SIGINT	終了	ターミナルで Ctrl-C が押された.
3	SIGQUIT	コアダンプ	ターミナルで Ctrl-\が押された.
4	SIGILL	コアダンプ	不正な機械語命令を実行した.
8	SIGFPE	コアダンプ	演算でエラーが発生した.
9	SIGKILL	終了	強制終了（**ハンドリングの変更ができない**）.
10	SIGBUS	コアダンプ	不正なアドレスをアクセスした場合など.
11	SIGSEG	コアダンプ	不正なアドレスをアクセスした場合など.
14	SIGALRM	終了	`alarm()` で指定した時間が経過した.
15	SIGTERM	終了	終了.
17	SIGSTOP	停止	一時停止（**ハンドリングの変更ができない**）.
18	SIGTSTP	停止	ターミナルで Ctrl-Z が押された.
19	SIGCONT	無視	一時停止中なら再開する.
20	SIGCHLD	無視	子プロセスの状態が変化した.

7.2　シグナル一覧

よく使用するシグナルの一覧を表 7.1 に示す．本当は 1 番から 31 番までのシグナルがあるが，よく使用されるものだけを掲載する[*4]．記号名は C 言語のプログラムで使用できるシグナルの名前である．デフォルトは，次の節で説明するシグナルハンドリングの初期状態を表す．SIGKILL と SIGSTOP は**ハンドリングを変更できない**．

7.3　シグナルハンドリング

受け取ったシグナルをプロセスがどのように扱うかを**シグナルハンドリング**と言う．シグナルハンドリングは，次の節で紹介する signal システムコールを用いて，プロセス自身がシグナルの種類ごとに予め決めておく．指定できるシグナルハンドリングは以下の三種類である．

1. **無視**（*ignore*）そのシグナルを無視する．
2. **捕捉・キャッチ**（*catch*）そのシグナルを受信し，登録しておいたシグナル処理ルーチン（シグナルハンドラ関数）を呼び出す．
3. **デフォルト**（*default*）シグナルの種類ごとに決められている初期のハンドリングであり，以下の四種類のどれかである．各シグナルのデフォルトが四種類のうちのどれかは表 7.1 から分かる．

   停止　プロセスは一時停止状態になる．

   無視　プロセスはそのシグナルを無視する．

   終了　プロセスは終了する．

   コアダンプ　プロセスは core ファイル[*5]を作成してから終了する．

[*4] 詳しくはオンラインマニュアル（`man 3 signal`）を読むこと．

[*5] プロセスのメモリイメージを書き出したファイルのこと．デバッグに使用することができる．macOS の標準ではコアダンプしてもコアファイルを作らないように設定されている．

7.4 signal システムコール

自プロセスのシグナルハンドリングは signal システムコール[*6]を使用して変更できる．

書式 次の通りである．

```
#include <signal.h>
sig_t signal(int sig, sig_t func);                    // macOSの場合
__sighandler_t signal(int sig, __sighander_t func); // Ubuntu Linuxの場合
```

解説 signal システムコールは，自プロセスの指定されたシグナルのハンドリングを変更する．sig_t 型は関数ポインタ型[*7]であり，signal.h をインクルードすることで自動的に宣言される．

引数 第 1 引数（sig）はハンドリングを変更するシグナルの種類である．種類は表 7.1 に示した番号または記号名で指定する．第 2 引数（func）は新しいハンドリングである．以下の 3 種類から一つを指定する．

1. SIG_IGN はシグナルを無視するようにする．
2. SIG_DFL はシグナルハンドリングをデフォルトに戻す．
3. **シグナルハンドラ関数**を指定する[*8]と，そのシグナルを捕捉（キャッチ）できるようになる．シグナルを捕捉した時にシグナルハンドラ関数が実行される．シグナルハンドラ関数の型は void func(int sig); でなければならない．引数 sig には捕捉したシグナルの番号が渡される．

戻り値 正常なら変更前のハンドリングが返される．これを sig_t 型の変数に記録しておけば，後でハンドリングをもとに戻すことができる．エラーが発生した場合は SIG_ERR が返され，errno 変数にエラー番号がセットされる[*9]．

プログラム例 signal システムコールを使用するプログラムの例を二つ示す．

1. **シグナルを無視する例**
   リスト 7.1 に SIGINT を一時的に無視するプログラムの例を示す．5 行で Ctrl-C を無視するようにハンドリングを変更する．6 行では Ctrl-C を押してもプログラムが終了しない状態になっている．7 行でハンドリングを元に戻し Ctrl-C で終了するようにする．
2. **シグナルを捕捉する例**
   リスト 7.2 に SIGINT を捕捉するプログラムの例を示す．9 行から 11 行の間を実行中は，Ctrl-C が押される度に handler() 関数が実行される．

[*6] 最近の UNIX や macOS では signal はライブラリ関数である．しかし，古い UNIX ではシステムコールだったので，ここでは「signal システムコール」と呼ぶ．
[*7] 関数のアドレスを表す型である．
[*8] 関数の名前を書けばよい．C 言語で関数名は関数を指す関数ポインタになる．
[*9] 後で perror() 関数が errno 変数を参照して，エラー原因を表示することができる．

リスト 7.1　シグナルを無視する例

```
1 #include <signal.h>
2
3 int main() {
4   ...
5   signal(SIGINT, SIG_IGN);  // ここから
6   ...
7   signal(SIGINT, SIG_DFL);  // ここまで
8   ...
9 }
```

リスト 7.2　シグナルを補足する例

```
1  #include <signal.h>
2
3  void handler(int n) {       // シグナルハンドラ（プロトタイプ宣言どおり）
4    ...                       // シグナル処理
5  }
6
7  int main() {
8    ...
9    signal(SIGINT, handler);  // ここから（void f(int)型の関数を引数にする）
10   ...
11   signal(SIGINT, SIG_DFL);  // ここまで
12   ...
13 }
```

7.5　シグナルハンドラの制約

シグナルは割り込みのようなものなので，ハンドラ関数はいつ呼び出されるか分からない．そのためシグナルハンドラ中でやって良い処理には強い制約がある．例えば printf() を実行している途中でシグナルが発生した場合を考えて欲しい．シグナルハンドラの内部で printf() 関数を呼出したらどうなるだろうか．何かまずいことが起こりそうな気がする．

7.5.1　制約がある理由

printf() 関数を例に制約が必要な理由を考えてみよう．printf() は高水準 I/O の関数なので出力する内容を一旦バッファに書き込む．バッファに書き込む処理の最中にシグナルが発生し，シグナルハンドラが呼出されたとする．

もしも，シグナルハンドラ中に printf() の呼び出しが含まれていると，新しい printf() も同じバッファに出力を書き込むので，バッファのデータが壊れるかもしれない．printf() に限らず多くのライブラリ関数は，シグナルハンドラから呼び出されることを前提に設計されていない．

7.5.2 やってもよいこと

printf() 関数の例では説明できなかった他の理由もあるので，シグナルハンドラでは次の三つのことしかやってはならない．

1. シグナルハンドラ関数のローカル変数の操作
2. volatile sig_atomic_t 型変数の読み書き[*10]
3. 非同期シグナル安全な関数の呼び出し

非同期シグナル安全な関数は，例えば次のような関数（システムコールも含む）である[*11]．

_exit(), alarm(), chdir(), chmod(), close(), creat(), dup(), dup2(), execle(), execve(), fork(), kill(), link(), lseek(), mkdir(), open(), pause(), read(), rename(), rmdir(), signal(), sleep(), stat(), time(), unlink(), wait(), write(), strcpy(), strcat(), ...

7.6 シグナルハンドラの例

リスト 7.3 に Ctrl-C を三回押すまで終了しないプログラムの例を示す．3 行はシグナルハンドラから操作して良い変数 flg を宣言している．4 行からがシグナルハンドラ関数である．シグナルハンドラは flg の単純な代入と，非同期シグナル安全な write() の実行しかしない．

10 行で handler() を SIGINT のハンドラ関数として登録している．11 行の while 文で，変数 flg が 1 になった回数を数えている．main() でも flg は単純な**参照**と**代入**しかしない．このように，ハンドラ関数ではフラグを立てることと，安全なシステムコールの呼び出し程度のことしかできない．

7.7 kill システムコール

プロセスがプロセスにシグナルを送信するシステムコールである．

書式 以下の通りである．

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

解説 kill システムコールは，送信先のプロセスとシグナルの種類を指定してシグナルを送信する．

引数 第 1 引数（pid）は送信先プロセスのプロセス番号である．pid_t はプロセス番号を格納するのに都合の良い整数型である．第 2 引数（sig）は送信するシグナルの種類である．種類は表 7.1 に示した記号名または番号で指定する．

戻り値 正常時に 0，異常時に-1 が返される．その際，エラー番号が errno 変数にセットされる．

プログラム例 リスト 7.4 に kill システムコールの使用例として，kill コマンドを簡単化したプログラ

[*10] sig_atomic_t 型は macOS では int 型である．volatile を付けると C コンパイラの最適化の対象外になる（詳細はコンパイラにより異なる．）．コンパイラの最適化は変数がシグナルハンドラなどから非同期にアクセスされることを前提にしていない．更に注意が必要なことは，これらの制約がある上に「読み書き」が単純な**参照**と**代入**のことしか指していないことである（インクリメントが正しく動く保証はない．）．

[*11] 詳細はオンラインマニュアル man 2 sigaction を参照のこと．

リスト 7.3　シグナルハンドラの例

```
1  #include <unistd.h>
2  #include <signal.h>
3  volatile sig_atomic_t flg = 0;        // シグナルハンドラが操作しても良い
4  void handler(int n) {
5    flg = 1;                            // 単純な代入
6    write(1, "Ctrl-C\n", 7);            // 非同期シグナル安全な関数の実行
7  }
8  int main(int argc, char **argv) {
9    int cnt = 0;
10   signal(SIGINT, handler);
11   while (cnt<3) {
12     if (flg) {                        // 単純な参照
13       cnt++;
14       flg = 0;                        // 単純な代入
15     }
16   }
17   return 0;
18 }
```

ム（mykill）を示す．このプログラムは，シグナル番号とプロセス番号を引数に実行し，指定されたシグナルを指定されたプロセスに送信する．11，12 行の `atoi()` 関数は，10 進数を表す文字列（例えば`"123"`）から整数値（int 型の 123）を求める関数である[*12]．リスト 7.5 に，このプログラムの使用例を示す．

7.8　シグナルと合わせて使うシステムコール

プロセスを待ち状態にしたり，シグナルを予約したりするシステムコール[*13]を紹介する．

7.8.1　sleep システムコール

自プロセスを指定された時間，またはシグナルを受信するまで，待ち状態にする．

書式　以下の通りである．

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

解説　sleep システムコールは時間を決めて自プロセスを待ち状態にする．もしも待ち状態になっている間にシグナルが届いた場合は，そのシグナルのハンドリングが**無視以外**なら，待ち状態が解除される（sleep システムコールが終了する．）．シグナルを捕捉した場合は直ちにシグナルハンドラが実行される．

[*12] 詳しくはオンラインマニュアル `man 3 atoi` で調べること．

[*13] これらはライブラリ関数であるが，古い UNIX ではシステムコールだったので，ここではシステムコールと呼ぶ．

リスト 7.4 簡易 kill プログラム（mykill）

```
1  #include <stdio.h>
2  #include <stdlib.h>                  // atoi のために必要
3  #include <signal.h>                  // kill のために必要
4
5  int main(int argc, char *argv[]) {
6    if (argc!=3) {
7      fprintf(stderr, "Usage : %s SIG PID\n", argv[0]);
8      return 1;
9    }
10
11   int sig = atoi(argv[1]);           // 第１引数
12   int pid = atoi(argv[2]);           // 第２引数
13
14   if (kill(pid,sig)<0) {
15     perror(argv[0]);
16     return 1;
17   }
18   return 0;
19 }
```

リスト 7.5 簡易 kill プログラム（mykill）の実行例

```
% ./mykill                                <-- 使い方が分からない
Usage : ./mykill SIG PID                  <-- 使い方を表示してくれる
% sleep 1000 &
[1] 13589                                 <-- sleep が JOB=1, PID=13589 だと分かる
% ./mykill 2 13589                        <-- PID=13589のプロセスにSIGINT(2)を送る
[1]  + interrupt  sleep 1000
% ./mykill 100 13589
./mykill: Invalid argument                <-- シグナル番号が不正
% ./mykill 2 13589
./mykill: No such process                 <-- プロセス番号が不正
%
```

引数 seconds は待ち時間を秒単位で指定する．

戻り値 待ち時間が経過して sleep システムコールが終了した場合は 0 が返される．シグナルで終了した場合は sleep 予定だった残りの秒数が返される．

プログラム例 リスト 7.6 に１秒に一度 hello と表示するプログラムを示す．

7.8.2 pause システムコール

時間制限がない sleep システムコールである．

リスト 7.6　sleep システムコールの使用例

```
1  #include <stdio.h>
2  #include <unistd.h>
3  int main() {
4    for (;;) {            // 無限ループ
5      printf("hello\n");  // hello表示
6      sleep(1);           // １秒待つ
7    }
8    return 0;
9  }
10
11 /* 実行例
12 % ./a.out
13 hello
14 hello
15 ^C                <-- Ctrl-C で止める
16 */
```

書式　以下の通りである．

```
#include <unistd.h>
int pause(void);
```

解説　pause システムコールはシグナルが到着するまでプロセスを待ち状態にする．シグナルを受信した時の動作は sleep システムコールと同様である．シグナルを受信した時，ハンドリングが**終了**になっている場合はプロセスが終了する．ハンドリングを**捕捉**にしてから待たなければならない．

戻り値　-1 が返される．（常時エラー）[*14]

プログラム例　リスト 7.7 はシグナルが三回発生するのを pause システムコールで待つプログラムである．ハンドリングを捕捉にしていないと一回目の Ctrl-C でプログラムが終了するので，何もしないハンドラ関数を登録している．

7.8.3　alarm システムコール

アラームシグナルの発生を予約するシステムコールである．

書式　以下の通りである．

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

解説　alarm システムコールは SIGALRM シグナルが seconds 秒後に発生するようにタイマーに予約をする．alarm システムコールは予約するだけで待ち状態になる訳ではない．

引数　seconds 秒後にシグナルを発生する．0 を指定した場合は以前の予約を取り消す．

[*14] シグナルによってエラーが発生し pause システムコールが打ち切られたという意味である．

リスト 7.7 pause システムコールの使用例

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <signal.h>
4  void handler(int n) {}          // 何もしないハンドラ関数
5  int main() {
6    signal(SIGINT, handler);      // SIGINTを捕捉に変更する
7    pause();                      // 1回目の  Ctrl-C を待つ
8    pause();                      // 2回目の  Ctrl-C を待つ
9    pause();                      // 3回目の  Ctrl-C を待つ
10   return 0;                     // Ctrl-C 3回で終了する
11 }
12 /* 実行例
13 % ./a.out
14 ^C^C^C%       <-- Ctrl-C 3回で終了した
15 */
```

リスト 7.8 alarm システムコールの使用例

```
1  #include <stdio.h>
2  #include <unistd.h>                  // alarm, pause のために必要
3  #include <signal.h>                  // signal のために必要
4  void handler(int n) {}               // 何もしないハンドラ関数
5  int main(int argc, char *argv[]) {
6    signal(SIGALRM, handler);          // SIGALRMを捕捉に変更する
7    alarm(3);
8    printf("pause()します\n");
9    pause();                           // プロセスが停止する
10   printf("pause()が終わりました\n");
11   return 0;
12 }
13 /* 実行例
14 % ./a.out
15 pause()します
16 pause()が終わりました          <-- 3秒後表示される
17 */
```

戻り値 通常は 0 が返される．既に別の予約がされていた場合は前の予約の残り時間を返し，今回の seconds を用いてタイマーを再起動する．

プログラム例 リスト 7.8 はアラームシグナルが発生するまで pause システムコールで待つプログラムの例である．alarm システムコールはシグナルの予約をするだけなので，pause システムコールでプログラムを停止する必要がある．

第 8 章

環境変数

この章では，環境変数について学ぶ．環境変数はシェルが管理する変数である．変数とはいえ C 言語の変数とは全く別の仕組みである．環境変数はシェルから，シェルによって起動されたプロセスにコピーされる．プロセス（プログラム）は実行時に環境変数の値を調べることができる．例えば，使用言語を表す環境変数を参照しエラーメッセージの言語を切り替えるようにプログラムを作っておけば，一種類のプログラムを世界中で使うことができる．

8.1 環境変数と使用例

リスト 8.1 に macOS や UNIX でよく使用される環境変数の「名前」と「値」の例を示す．

例えば LC_TIME 環境変数は日時の表示に使用する言語を決める．また，TZ 環境変数はどの地域の時刻を表示するか決める．date コマンド (現在時刻表示)，cal コマンド (カレンダー表示)，ls コマンド (ファイルの最終変更時刻表示) 等がこれらの環境変数の値により日時の表示を変化させる．

リスト 8.2 に，これら環境変数の値を変化しながらコマンドを実行した例を示す．例えば date コマンドは現在時刻を，LC_TIME 環境変数の値が C なら英語表記で表示するが，ja_JP.UTF-8 なら日本語表記で表示する．TZ 環境変数の値が Japan なら日本時間で表示するが，Cuba ならキューバ時間で表示する．このように環境変数はプログラムの振る舞いに影響を与える．

リスト 8.1　よく使用される環境変数

```
SHELL=/bin/zsh                  # 使用中のシェル
TERM=xterm-256color             # 使用中のターミナルエミュレータ
USER=sigemura                   # 現在のユーザ
PATH=/usr/bin:/bin:/usr...      # シェルがコマンドを探すディレクトリ一覧
PWD=/Users/sigemura             # カレントディレクトリのパス
HOME=/Users/sigemura            # ユーザのホームディレクトリ
LANG=ja_JP.UTF-8                # ユーザが使用したい言語(ja_JP.UTF-8(日本語))
LC_TIME=C                       # ユーザが日時の表示に使用したい言語（C言語標準）
TZ=Japan                        # どの地域の時刻を使用するか(日本)
CLICOLOR=1                      # ls コマンド等がカラー出力する(yes)
```

リスト 8.2　環境変数の変化が与える影響の例

```
% export LC_TIME=C               # LC_TIME環境変数を作ってC言語標準(米国英語)を表す値をセット
% date
Sun Apr  2 08:15:20 JST 2023    # 英語表記，日本時間の現在時刻
% ls -l Makefile
-rw-r--r--  1 sigemura  staff  128 Apr  1 10:40 Makefile
% LC_TIME=ja_JP.UTF-8           # LC_TIMEに日本語標記を表す値をセットして試す
% date                           # 日本語表記，日本時間の現在時刻を表示する
2023年 4月 2日 日曜日 08時16分44秒 JST
% ls -l Makefile
-rw-r--r--  1 sigemura  staff  128  4  1 10:40 Makefile
% export TZ=Cuba                # TZ環境変数を作ってキューバ時間を表す値をセット
% date                           # 日本語表記，キューバ時間の現在時刻
2023年 4月 1日 土曜日 19時19分50秒 CDT
% ls -l Makefile
-rw-r--r--  1 sigemura  staff  128  3 31 21:40 Makefile
%
```

8.2　環境変数を誰が決めるか

1. システム管理者
   システム管理者はユーザがログインした時の初期状態を決める．macOS では，/etc/zprofile ファイル等[*1]に書かれたスクリプトが全ユーザのシェル起動時に実行される．システム管理者は全ユーザに共通のシェルの初期化処理をここに書いておく．このスクリプトで全ユーザに共通の環境変数の初期化ができる．
2. ユーザの設定ファイル
   macOS ではホームディレクトリの.zprofile ファイル[*2]に自分専用の初期化処理を書くことができる．以下に.zprofile ファイルの例を示す[*3]．これは，PATH 環境変数の値を変更した後，LC_TIME，CLICOLOR 環境変数を新たに作成する例である．

   ```
   PATH="/usr/local/bin:$PATH:$HOME/bin:."
   export LC_TIME=C
   export CLICOLOR=1
   ```

3. ユーザによるコマンド操作
   シェルのコマンド操作で環境変数を操作することができる．但し，影響範囲は操作したウインドのシェルのみである．また，次回のログイン時には操作結果の影響は残らない．

[*1] Linux や macOS Mojave 以前では/etc/profile
[*2] Linux や macOS Mojave 以前では.bash_profile
[*3] 代入（=）の左右に空白を書いてはならない．以降の書式や実行例でも同様である．

8.3 環境変数の操作

以下では，コマンドラインで環境変数を操作する方法を解説する．

8.3.1 表示

printenv コマンドを用いて環境変数を表示する．

書式 name は環境変数の名前である．

```
printenv [name]
```

解説 name を省略した場合は，全ての環境変数の名前と値を表示する．name を書いた場合は該当する環境変数の**値だけ**表示する．該当する環境変数が無い場合は何も表示しない．

実行例 macOS 上での printenv コマンドの実行例を示す．環境変数の名前を省略して実行した場合は，全ての環境変数について「名前=値」形式で表示される．

```
% printenv
SHELL=/bin/zsh                  <--- 「名前=値」形式で表示
TERM=xterm-256color
USER=sigemura
...
% printenv SHELL                <--- SHELL環境変数を表示する
/bin/zsh                              (「値」だけ表示される)
% printenv NEVER
%                               <--- 何も表示されない
```

8.3.2 新規作成 (その１)

UNIX の標準シェル (sh) での操作方法を説明する．

書式 次の 2 ステップで操作を行う．

```
1  name=value
2  export name
```

解説 1 行で，一旦，シェル変数を作る．2 行でシェル変数を環境変数に変更する．

実行例 1 行は MYNAME 環境変数が存在するか確認している．MYNAME 環境変数は存在しないので何も表示されない．2，3 行で値が sigemura の MYNAME 環境変数を作った．4 行で MYNAME 環境変数を確認すると値が sigemura になっていることが分かる．

```
1 % printenv MYNAME             <--- MYNAMEは存在しない
2 % MYNAME=sigemura             <--- シェル変数MYNAMEを作る
3 % export MYNAME               <--- MYNAMEを環境変数に変更する
4 % printenv MYNAME
5 sigemura                      <--- 環境変数MYNAMEの値
6 %
```

8.3.3　新規作成 (その２)

macOS や Linux で使用されるシェル (zsh や bash) では，次のように１行の操作で環境変数を作ることができる．

書式　次の 1 ステップで環境変数を作ることができる．

```
export name=value
```

解説　一旦，シェル変数を作ることなく環境変数を作ることができる．

実行例　次のように動作確認ができる．

```
% printenv MYNAME              <--- MYNAME環境変数は存在しない
% export MYNAME=sigemura
% printenv MYNAME              <--- MYNAME環境変数ができていた
sigemura
%
```

8.3.4　値の変更

既に存在する環境変数の値を変更することができる．

書式　name は環境変数の名前，value は新しい値である．

```
name=value
```

解説　「環境変数の変更」と「シェル変数の作成」は書式だけでは区別が付かない．変数名を間違った場合，間違った名前で新しいシェル変数が作成されエラーにならないので注意が必要である．

実行例　MYNAME 環境変数が既に存在している場合の実行例を示す．

```
% printenv MYNAME                <--- 値を表示する
sigemura
% MYNAME=tetsuji                 <--- 値を変更する
% printenv MYNAME
tetsuji                          <--- 変更されている
%
```

8.3.5　値の参照

環境変数やシェル変数の値を利用することができる．

書式　name は環境変数の名前である．

```
$name
```

解説　$name は，シェルが入力されたコマンドの意味を評価する際に，変数の値に置き換えられる．

実行例 1　すでにある PATH 環境変数の値に新規ディレクトリを追加する[*4]例を示す．

[*4] システムが決めたディレクトリに，ユーザがディレクトリを追加することはよくある．

```
% printenv PATH                          # PATH の初期値を確認
/bin:/usr/bin
% PATH=$PATH:.                           # カレントディレクトリを追加
% printenv PATH
/bin:/usr/bin:.
% PATH=$PATH:$HOME/bin                   # ホームのbinを追加
% printenv PATH
/bin:/usr/bin:.:/User/sigemura/bin
%
```

実行例 2　環境変数 i の値を数値と見做してインクリメント[*5]する例を示す．printenv i の代わりに echo $i でも値を表示できる．

```
% export i=1                    # 環境変数 i を作る
% printenv i
1
% i=`expr $i + 1`               # クォートはバッククォート
% echo $i
2
%
```

8.3.6 変数の削除

unset コマンドを用いて，環境変数やシェル変数を削除することができる．

書式　name は変数の名前である．

```
unset name
```

解説　存在しない変数を unset してもエラーにならない．変数名を間違ってもエラーにならないので注意が必要である．

実行例　MYNAME 環境変数が既に存在している場合の実行例を示す．

```
% printenv MYNAME
tetsuji
% unset MYNAME
% printenv MYNAME
%                                # MYNAMEは存在しない
```

8.3.7 一時的な作成と値の変更

env コマンドを用いて，環境変数の値を一時的 (今回のコマンド実行の期間だけ) に変更してコマンド (プログラム) を実行したり，一時的に環境変数を作ってコマンドを実行したりすることができる．

書式　変数へ値を代入する指示が続いた後に，実行するコマンドが続く．

[*5] expr コマンドと sh（bash）のコマンド置換を使用している．

ロケール

LANG 環境変数や LC_TIME 環境変数にセットする値をロケール名と呼ぶ．ロケール名は「言語コード」,「国名コード」と「エンコーディング」の組み合わせで表現される．

- 言語コード[a]は ISO639 で定義された 2 文字コードである（日本語は”ja”）.
- 国名コード[b]は ISO3166 で定義された 2 文字コードである（日本は”JP”）.
- エンコーディングは，使用する文字符号化方式を示す．エンコーディングがターミナルエミュレータのテキストエンコーディングと一致していないと文字化けを起こす（macOS や Linux では UTF-8 方式が使用される.）.

使用可能なロケールの一覧は `locale -a` コマンドで表示できる．通常，macOS や Linux で日本語を使用する場合のロケール名は，言語コード=ja，国名コード=JP，エンコーディング=UTF-8 を組み合わせて次のようになる．

ja_JP.UTF-8（日本語_日本.UTF-8）

[a] 言語コードは https://ja.wikipedia.org/wiki/ISO_639-1 等を参照のこと.
[b] 国名コードは https://ja.wikipedia.org/wiki/ISO_3166-1 等を参照のこと.

```
env name1=value1 name2=value2 ... command
```

解説　env コマンドは代入形式のコマンド行引数が続く間，それらを環境変数の変更（作成）指示とみなし処理する．代入形式ではないコマンド行引数を見つけたら，それ以降を実行すべきコマンドとみなす.

実行例　ロケール[*6]とタイムゾーン[*7]を変更して date コマンドを実行する例である．日時表示用のロケールを格納する LC_TIME 変数を日本語表示を示す値に，タイムゾーンを格納する TZ 変数をキューバ時間を表す値に変更した上で，date コマンドを実行している．その後で date コマンドをもう一度実行し，環境変数の変更が一時的であることを確認している.

```
% date
Sun Apr  2 08:56:40 JST 2023            <--- 普通は日本時間，英語表記
% env LC_TIME=ja_JP.UTF-8 TZ=Cuba date
2023年 4月 1日 土曜日 19時56分55秒 CDT    <--- キューバ時間，日本語表記
% date
Sun Apr  2 08:57:00 JST 2023            <--- 後のコマンドに影響はない
%
```

[*6] 囲み記事「ロケール」を参照のこと.
[*7] 囲み記事「タイムゾーン」を参照のこと.

タイムゾーン

TZ 環境変数にタイムゾーンを表す値をセットする．macOS や UNIX の内部で時刻は協定世界時 (UTC) で管理されており，表示する際に現地時間に変換する．時刻に関係するプログラムは協定世界時と現地時間の変換方法を TZ 環境変数から知ることができる．日本の場合はタイムゾーン名が JST，協定世界時との時差が −9 時間なので，TZ=JST-9 となる．
この形式の他に/usr/share/zoneinfo/に置いてあるファイル名でタイムゾーンを指定することもできる．/usr/share/zoneinfo/Cuba ファイルが存在するので TZ=Cuba と指定できる．/usr/share/zoneinfo/Japan ファイルも存在するので TZ=Japan も指定できる．/usr/share/zoneinfo/Asia/Tokyo ファイルが存在するので TZ=Asia/Tokyo と指定しても良い．
なお，TZ 環境変数が定義されていない時は，OS のインストール時に選択した標準のタイムゾーンが用いられる．

8.4 環境変数の仕組み

環境変数を管理する仕組みと，環境変数をプロセス間で引き継ぐ仕組みを学ぶ．

8.4.1 シェルによる管理

シェルもプログラムの一種なので，図 6.2 に示したようなプロセスとして実行される．図 8.1 は，図 6.2 をシェルプロセスの場合に当てはめた図である．シェルプロセス（shell プロセス）は環境変数を自プロセスのデータ領域に[*8]記憶している．ユーザが export や unset 等のコマンドを入力すると，シェルプログラムのインタプリタが意味を解釈し環境変数を操作する．なお，export や unset のような，シェル自身によって処理されるコマンドを**内部コマンド**と呼ぶ．

8.4.2 プロセスへのコピー

図 8.2 に示すように，シェルは入力されたコマンドが内部コマンド以外 (**外部コマンド**) なら，コマンド名と同じ名前のプログラムを探し子プロセスとして起動する．この時シェルは，自身の環境変数を子プロセスにコピーするように OS カーネルに依頼する．OS カーネルは子プロセスのメモリ空間のどこか（例えばスタックの底）に環境変数をコピーする．子プロセスは，自身のメモリ空間にコピーされた環境変数を，参照・変更・削除できる．

8.4.3 変更した上でのコピー

プロセスへ環境変数をコピーする仕組みは，シェルに限らず全ての「他のプログラムを起動するプログラム」で共通に用いられる．8.3 で紹介した env コマンドも，この仕組を利用している．

env コマンドは**外部コマンド**である．env コマンドは他のプログラムを起動するプログラムとして実装できる．図 8.3 に env コマンドの仕組みを示す．env コマンドは自身の環境変数を変更した後，目的のプログラムを起動する．その時，新しいプログラムに変更後の環境変数がコピーされる．

[*8] 本当はスタック領域も使っているかも知れない．

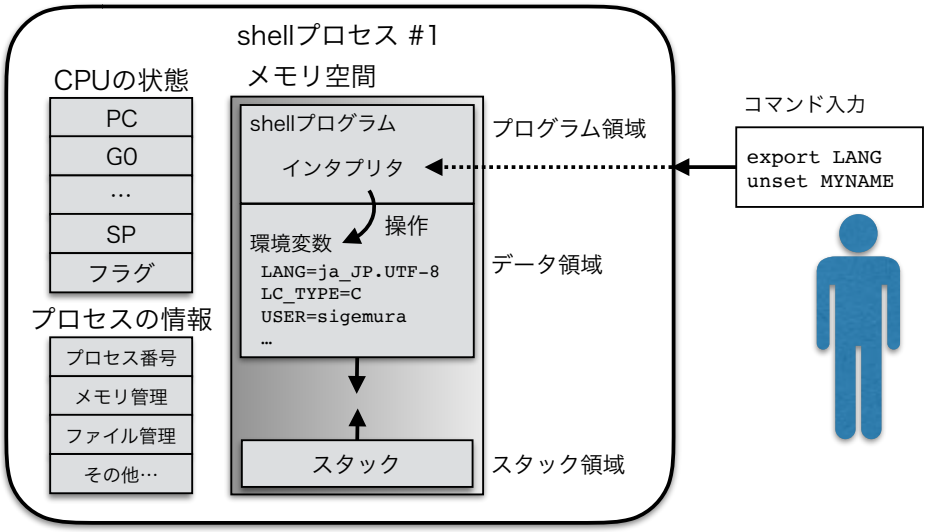


図 8.1　環境変数をシェルの内部で管理

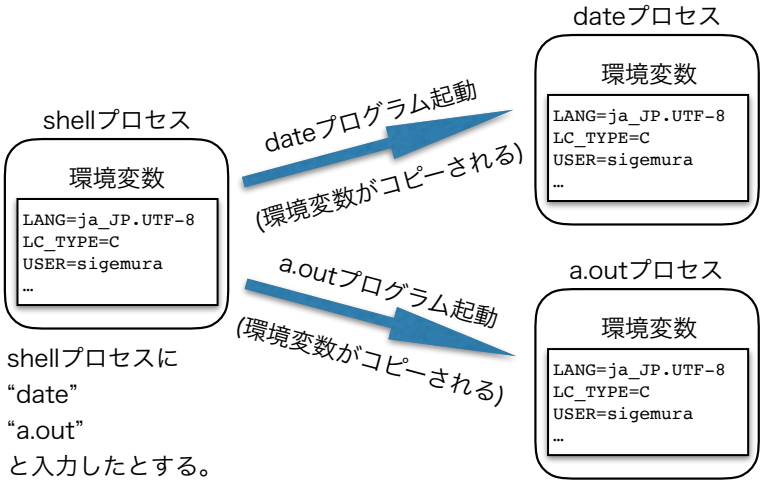


図 8.2　プログラム起動時の環境変数コピー

8.5　プログラムからの環境変数アクセス

C 言語プログラムから環境変数をアクセスする方法を紹介する．

8.5.1　読み出し

C 言語から環境変数を読み出すために，以下に紹介する二つの方法が使用できる．

1. ***main* 関数の *envp* 仮引数やグローバル変数 *environ* を用いる．**

   C 言語の `main()` 関数には，実は第三引数 `envp` が存在している．また，`environ` という名前の C 言語のグローバル変数も存在する．これらから環境変数のリストを読み出すことができる．

   書式　`environ` 変数はライブラリのどこかで定義されいて，C 言語プログラムからは 1 行目のよ

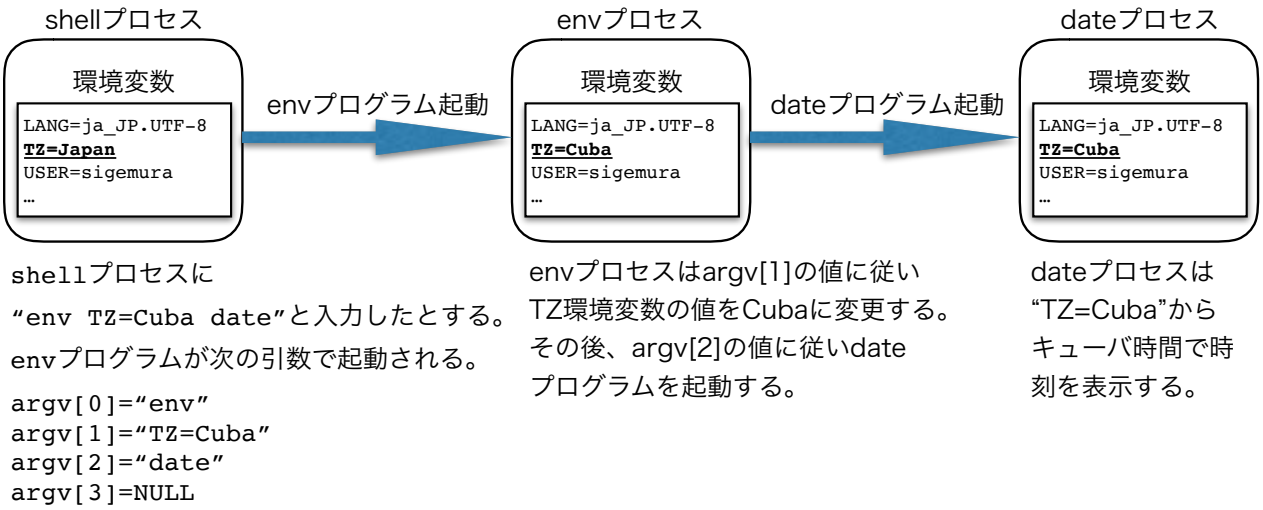


図 8.3　env プログラムの仕組み

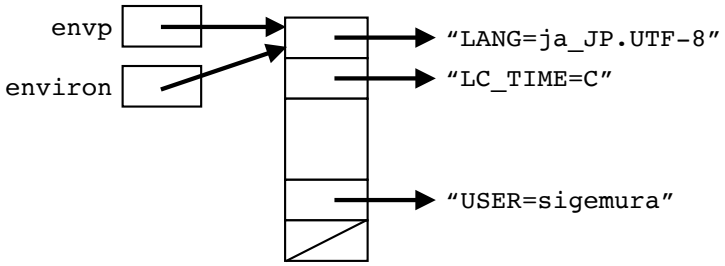


図 8.4　環境変数のデータ構造

うに宣言すれば参照できるようになる．main() の第三引数まで含めたプロトタイプ宣言は 2 行目の通りである．

```
1 extern char **environ;
2 int main(int argc, char *argv[], char *envp[]);
```

解説　environ 変数と main() 関数の envp 引数は，図 8.4 のように初期化される．例えば全ての環境変数を表示する C プログラムは次のように書くことができる．まず，2 行目のように書くことで environ 変数を参照可能になる．environ 変数は文字列を指すポインタの配列を指しているので，environ[i] の記述は LANG=ja_JP.UTF-8 等の文字列を意味する．配列の末尾には NULL ポインタが格納されているので，これを目印にループを終了する（4 行）．

```
1 #include <stdio.h>
2 extern char **environ;                    // 外部で定義されている
3 int main(int argc, char *argv[]) {        // 今回はenvpは不要
4   for (int i=0; environ[i]!=NULL; i++) {  // NULLが見つかるまで
5     printf("%s\n", environ[i]);           // 環境変数を印刷
6   }
7   return 0;                               // 必ず正常終了
8 }
```

```
/* 実行例
% ./a.out
LANG=ja_JP.UTF-8
TZ=Japan
HOME=/Users/sigemura
LC_TIME=C
...
 */
```

2. *getenv* **関数を用いる．**

getenv() ライブラリ関数を用いて名前で指定した環境変数の値を読み出すことができる．

書式　getenv() は次のような書式の関数である．

```
#include <stdlib.h>
char *getenv(const char *name);
```

解説　name には環境変数名を渡す．getenv() は環境変数の値を表す文字列を指すポインタを返す．name の環境変数が見つからない場合は NULL ポインタを返す．

プログラム例　次の C プログラムは LANG 環境変数の値を表示するものである．4 行で LANG 環境変数を探し，それの値（文字列）を指すポインタを返す．LANG 環境変数が見つかったら 6 行で LANG=に続けて値を表示する（表示例は 14 行）．見つからない場合は 8 行で見つからなかったことを意味するメッセージを表示する．

```
#include <stdio.h>
#include <stdlib.h>                    // getenv() のために必要
int main(int argc, char* argv[]) {
  char *val = getenv("LANG");          // LANG環境変数の値を調べる
  if (val!=NULL) {                     // 見つかったら
    printf("LANG=%s\n", val);          //   値を表示
  } else {                             // 見つからない時は
    printf("LANG does not exist.\n"); //   エラーメッセージを表示
  }
  return 0;                            // 正常終了
}
/* 実行例
% ./a.out
LANG=ja_JP.UTF-8
 */
```

8.5.2 操作

自プロセスのメモリ空間にコピーされた環境変数を操作する三つの C 言語ライブラリ関数を紹介する．なお，一旦，環境変数を追加・変更・削除する操作を行うと main() の仮引数 envp は使用できなくなる（値がデタラメになる）．常にグローバル変数 environ を使用するとトラブルが少ない．

1. *setenv* **関数**

   環境変数を新規に作成したり，値を上書きしたりする関数である．

   書式 setenv() は次のような書式の関数である．

   ```
   #include <stdlib.h>
   int setenv(const char *name, const char *val, int overwrite);
   ```

   解説 name は環境変数の名前，val は環境変数にセットする値である．overwrite は，0 の時に上書き禁止，それ以外の時に上書き許可を意味する．setenv() は正常時に 0，エラー時に-1 を返す．上書き禁止の時，既に同じ名前の環境変数が存在するとエラーになる．エラー時は errno 大域変数にエラー番号がセットされる．

   使用例 MYNAME 環境変数の値を sigemura にする例を示す．第 3 引数が 1 なので，MYNAME 環境変数が既に存在する場合は値の上書きになり，エラーにはならない．

   ```
   setenv("MYNAME", "sigemura", 1);
   ```

2. *putenv* **関数**

   環境変数を新規に作成したり，値を上書きしたりする関数である．

   書式 putenv() は引数を一つだけ持つ．

   ```
   #include <stdlib.h>
   int putenv(char *string);
   ```

   解説 string は NAME=VALUE 形式の文字列である (それ以外の形式の文字列を渡すとエラーになる)．putenv() は正常時に 0，エラー時に-1 を返す．エラー時は errno 大域変数にエラー番号がセットされる．putenv("NAME=VALUE"); は，setenv("NAME","VALUE",1); と同じ操作を行う．

   使用例 前出の setenv() の使用例と同じことを putenv() を用いて行う例を示す．

   ```
   putenv("MYNAME=sigemura");
   ```

   注意 NAME=VALUE 形式の文字列を格納して putenv() に渡した領域は，該当環境変数を記憶する領域として使い続けられる．この領域を書き換えたり，別の目的に再利用してはならない．

3. *unsetenv* **関数**

   環境変数を削除する関数である．

   書式 unsetenv() も引数を一つだけ持つ．

   ```
   #include <stdlib.h>
   int unsetenv(const char *name);
   ```

解説　name は削除する環境変数の名前である．unsetenv() は正常時に 0，エラー時に-1 を返す．
エラー時は errno 大域変数にエラー番号がセットされる．

使用例　MYNAME 環境変数を削除する例を示す．

```
unsetenv("MYNAME");
```

第 9 章

プロセスの生成とプログラムの実行

この章では新しいプロセスでプログラムを実行させる方法を学ぶ．プログラムを起動する方式には大きく分けて，*spawn* **方式**（スポーン：卵を産む方式）と，*fork-exec* **方式**（分岐-実行方式）がある．

9.1 spawn 方式

Windows 等で使用される方式である．新しい「(1) プロセスを作り」,「(2) プロセスを初期化し」,「(3) プログラムを実行する」の三つの仕事を一つの spawn システムコールで行う．fork-exec 方式を実現するためには，メモリ再配置のためのハードウェア機構を持っている必要がある．そのため組込用の小さなコンピュータでは spawn 方式しか選択肢がない場合がある．

UNIX 系 OS でも spawn 方式を使用できる．次に macOS の `posix_spawn` システムコールを紹介する．新しいプロセスの初期化を指示するデータ構造を渡すようになっている．

書式 次の通りである．

```
#include <spawn.h>
int posix_spawn(pid_t *pid, const char *path,
                const posix_spawn_file_actions_t *file_actions,
                const posix_spawnattr_t *attrp,
                char *const argv[], char *const envp[]);
```

解説 新しいプロセスを作り `path` で指定したプログラムを実行する．

引数 `pid` は新しいプロセスのプロセス番号を格納する変数を指すポインタ．`path` は実行するプログラムを格納したファイルのパスである．絶対パスでも相対パスでも良い．`file_actions`，`attrp` はプロセスの初期化を指示するデータ構造へのポインタ．`argv`，`envp` は実行されるプログラムに渡すコマンド行引数と環境変数である．

9.2 fork-exec 方式

UNIX 系の OS で用いられてきた方式のことである．以下に示す次の三つのステップで新しいプログラムを実行する．また，その様子を図 9.1 に模式的に示す．

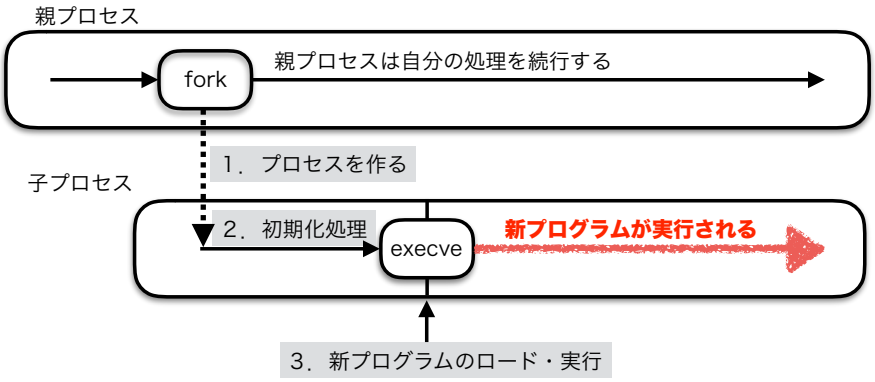


図 9.1　fork-exec 方式

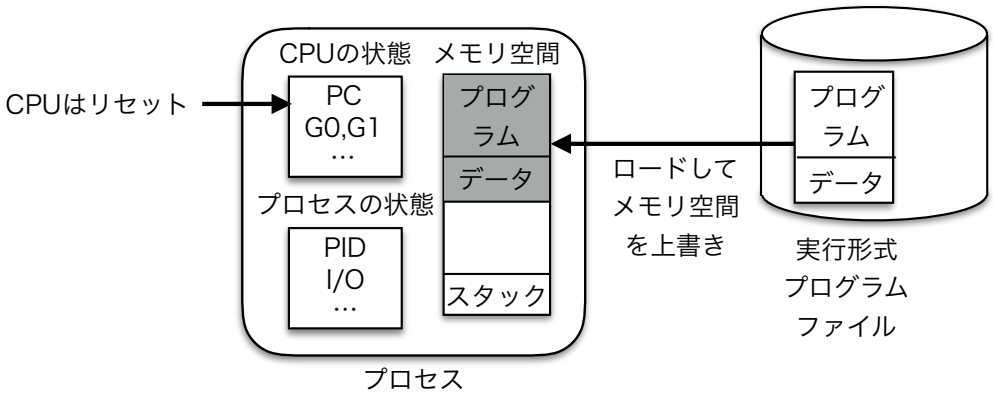


図 9.2　execve システムコールの仕組み

1. 親プロセスが新しいプロセス（子プロセス）を作る (*fork* **システムコール**).
2. ユーザが記述したプログラムに従い子プロセスが自ら初期化処理を行う.
3. 子プロセスが新しいプログラムをロード・実行 (*execve* **システムコール**) する.

この方式はユーザが記述したプログラムで初期化処理を行うので柔軟性が高い．以下では，fork-exec 方式について具体的なプログラム例を示しながら詳しく解説する.

9.2.1　プログラムのロードと実行 (execve システムコール)

まず，プロセスが新しいプログラムの実行を開始するために使用する execve システムコールを紹介する．図 9.1 では，作ったばかりの子プロセスに execve を実行させているが，普通のプロセスが execve システムコールを使用しても良い．fork と組み合わせて使用する例は後の方で説明することとし，ここでは execve システムコール単独で使用する例を示す.

図 9.2 に execve システムコールを実行するプロセスの様子を示す．プロセスが execve システムコールを実行すると，そのプロセスのメモリ空間に新しいプログラムがロードされる．execve システムコールを実行したプログラムは，新しいプログラムで上書きされ消えてしまう．プロセスの仮想 CPU はリセットされ，新しいプログラムが最初から実行される．プロセスは新しいプログラムを実行するように**変身**した（変身の術）．以下に，UNIX の execve システムコールの解説を示す.

リスト 9.1 execve の使用例 (その 1)

```
1  #include <stdio.h>          // perror のために必要
2  #include <unistd.h>         // execve のために必要
3  extern char **environ;
4  char *args[] = { "date", NULL };
5  char *execpath="/bin/date";
6  int main(int argc, char *argv[], char *envp[]) {
7    execve(execpath, args, environ);   // /bin/dateを自分と同じ環境変数で実行
8    perror(execpath);                  // ※ execveが戻ってきたらエラー！
9    return 1;
10 }
11 /* 実行例（英語表示、日本時間で表示される）
12 % ./exectest1
13 Sun Apr  2 09:20:24 JST 2023
14 */
```

書式 execve システムコールの書式は次の通りである．

```
#include <unistd.h>
int execve(const char *path, char *const argv[], char *const envp[]);
```

解説 自プロセスで path で指定したプログラムを実行する．正常時には execve を実行したプログラムは新しいプログラムで上書きされ消える．execve システムコールが戻る (次の行が実行される) のはエラー発生時だけである．

引数 path は実行するプログラムを格納したファイルのパスである．絶対パスでも相対パスでも良い．argv，envp は新しいプログラムに渡すコマンド行引数と環境変数である．

使用例１ リスト 9.1 に execve システムコールを用いて，/bin/date プログラムを実行するプログラムの例を示す．3 行は自身の環境変数を指すポインタ environ を宣言している．4 行で date プログラムの argv に渡す配列 args を準備した．7 行の execve システムコールで/bin/date プログラムをロード・実行する．args，environ は，date プログラムの main 関数の argv と envp に渡される．environ を渡したので，date プログラムはこのプログラムと同じ環境変数で実行される．execve システムコールが正常に実行された場合は，システムコールを発行したプログラムが date プログラムによって上書きされ消えるので 8 行以降は実行されない．8 行が実行されるのはシステムコールの実行に失敗した場合だけである．そこでエラー判定（if 文）なしにエラー処理（perror の実行など）を行ってよい．

使用例２ リスト 9.2 は，環境変数の一部を書き換えた上で，/bin/date プログラムを実行するプログラムの例である．これは，プログラムを実行する前に行う**初期化処理**の例でもある．まず 8 行で初期化処理として putenv() 関数を用いて自身の環境変数を書換える．execve システムコールに渡される environ は書換え後のものである．LC_TIME 環境変数を ja_JP.UTF-8 に書換えてあるので，15 行のように現在時刻が日本語で表示さる．

使用例３ リスト 9.3 は，全く新しい環境変数の一覧を渡して/bin/date プログラムを実行する例で

リスト 9.2　execve の使用例 (その 2)

```
1  #include <stdio.h>          // perror のために必要
2  #include <unistd.h>         // execve のために必要
3  #include <stdlib.h>         // putenv のために必要
4  extern char **environ;
5  char *args[] = { "date", NULL };
6  char *execpath="/bin/date";
7  int main(int argc, char *argv[], char *envp[]) {
8    putenv("LC_TIME=ja_JP.UTF-8");      // 自分の環境変数を変更する
9    execve(execpath, args, environ);    // /bin/dateを自分と同じ環境変数で実行
10   perror(execpath);                   // execveが戻ってきたらエラー！
11   return 1;
12 }
13 /* 実行例（日本語表示、日本時間で表示される）
14 % ./exectest3
15 2023年 4月 2日 日曜日 09時24分29秒 JST
16 */
```

リスト 9.3　execve の使用例 (その 3)

```
1  #include <stdio.h>          // perror のために必要
2  #include <unistd.h>         // execve のために必要
3  char *args[] = { "date", NULL };
4  char *envs[] = { "LC_TIME=ja_JP.UTF-8", "TZ=Cuba", NULL};
5  char *execpath="/bin/date";
6  int main(int argc, char *argv[], char *envp[]) {
7    execve(execpath, args, envs);       // /bin/dateを上記の環境変数で実行
8    perror(execpath);                   // execveが戻ってきたらエラー！
9    return 1;
10 }
11 /* 実行例（日本語表示、キューバ時間で表示される）
12 % ./exectest2
13 2023年 4月 1日 土曜日 20時25分52秒 CDT
14 */
```

ある．4 行で environ と同じデータ構造の envs 配列を作って，7 行で execve システムコールに渡した．envs 配列に LC_TIME，TZ 環境変数が格納されているので，これらが/bin/date プログラムにコピーされる．他の環境変数を/bin/date プログラムは必要としていないので，13 行のように言語とタイムゾーンを変更して正常に動作する．

使用例 4　リスト 9.4 は，複数のコマンド行引数がある場合の例である．/bin/echo プログラムを aaa，bbb を引数にして実行する．args 配列にプログラムの名前（argv[0] のための"echo"）を忘れないように注意すること．

リスト 9.4 execve の使用例 (その 4)

```
1  #include <stdio.h>          // perror のために必要
2  #include <unistd.h>         // execve のために必要
3  extern char **environ;
4  char *args[] = { "echo", "aaa", "bbb", NULL };  // "$ echo aaa bbb" に相当
5  char *execpath="/bin/echo";
6  int main(int argc, char *argv[], char *envp[]) {
7    execve(execpath, args, environ);   // /bin/echoを自分と同じ環境変数で実行
8    perror(execpath);                  // execveが戻ってきたらエラー！
9    return 1;
10 }
11 /* 実行例
12 % ./exectest4
13 aaa bbb                  <--- /bin/echo の出力
14 */
```

9.2.2 execve システムコールのラッパー関数

execve システムコールを使いやすくするライブラリ関数を紹介する．ここで紹介する関数は内部で execve システムコールを呼び出す．これらの関数は execve システムコールの**ラッパー（*wrapper*）関数**になっている．

書式 execve システムコールの四つのラッパー関数の書式をまとめて掲載する．

```
#include <unistd.h>
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execl(const char *path, const char *argv0, ... , *argvn, NULL);
int execlp(const char *file, const char *argv0, ... , *argvn, NULL);
```

解説 ***execv()* 関数**は環境変数を指定する必要がない exec 関数である．常に自プログラムの環境変数 environ を execve システムコールに渡す．execve システムコールとの関係は次のようになる．

execv("/bin/date", argv); → execve("/bin/date", argv, environ);

***execvp()* 関数**は，execv() 関数に PATH 環境変数を用いたプログラムファイルの検索機能を追加したものである．第一引数に/bin/date のようなパスではなく date のようなファイル名だけ渡すと，自動的に PATH 環境変数を使用した検索を行い/bin/date プログラムを発見して実行してくれる．

execvp("date", argv); → execve("/bin/date", argv, environ);

***execl()* 関数**は，execv() 関数の argv 配列の代わりに複数の文字列を渡すことができる関数である．文字列の個数は可変なので終わりを示す NULL を最後に置く必要がある．リスト 9.4 の execve システムコールを次のように書換えることで，同じ結果を得ることができる．argv[0] に渡す"echo"を忘れないように注意が必要である．

execl("/bin/echo", "echo", "aaa", "bbb", NULL);

リスト 9.5　標準出力をリダイレクトして/bin/echo を実行する例

```
1  #include <stdio.h>          // perror のために必要
2  #include <unistd.h>         // execl のために必要
3  #include <fcntl.h>          // open のために必要
4  char *execpath="/bin/echo";
5  char *outfile="aaa.txt";
6  int main(int argc, char *argv[], char *envp[]) {
7    close(1);                                        // 標準出力をクローズする
8    int fd = open(outfile,
9                  O_WRONLY|O_CREAT|O_TRUNC, 0644); // 標準出力をオープンしなおす
10   if (fd!=1) {                                     // 標準出力以外になってる
11     fprintf(stderr, "something is wrong\n");       //   原因が分からないが...
12     return 1;                                      //   何か変なのでエラー終了
13   }
14   execl(execpath, "echo", "aaa", "bbb", NULL);   // /bin/echoを実行
15   perror(execpath);                                // execlが戻ったらエラー！
16   return 1;
17 }
18 /* 実行例
19 % ./exectest5               <-- echo が実行されたはずなのに何も出力されない
20 % cat aaa.txt               <-- "aaa.txt" に
21 aaa bbb                    <--   echo の出力が保存されていた
22 */
```

***execlp()* 関数**は，execl() 関数に PATH 環境変数を用いたプログラムファイルの検索機能を追加したものである．

```
execlp("echo", "echo", "aaa", "bbb", NULL);
```

9.2.3　入出力のリダイレクト

execve する際，プロセス状態（図 9.2 参照）の一部が引き継がれる．例えば，PID（プロセス番号），オープン中のファイルや入出力，カレントディレクトリ，「無視」に設定されたシグナルハンドリング等は，新しいプログラムに引き継がれる．プログラムが最初から標準入力 (0)，標準出力 (1)，標準エラー出力 (2) を利用できるのは，execve 前にオープンされていた入出力を引き継ぐからである..

シェルがプログラムを実行する際は，シェルは標準入力をキーボード用にオープンし，標準出力と標準エラー出力をディスプレイ用にオープンした状態[*1]で目的のプログラムを execve する．シェルのリダイレクト（プログラムの入出力を切替える仕組み）は，リダイレクト先のファイルを標準入力・出力としてオープンした状態で目的のプログラムを execve することで実現している．

リスト 9.5 は，標準出力を aaa.txt ファイルにリダイレクトした状態で/bin/echo を実行するプログラムである．まず，初期化処理（図 9.1 参照）として 7 行と 8 行で標準出力のリダイレクトを行っ

[*1] 既にシェル自身用にオープン状態になっているので，わざわざ，オープンし直す必要は無い．

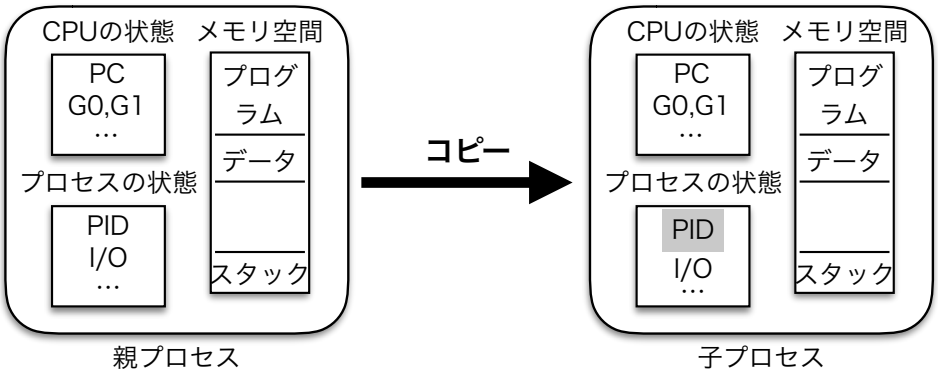


図 9.3 fork の仕組み

ている．7 行の `close(1)` は標準出力（ファイルディスクリプタ 1 番）をクローズする．`open()` は空きファイルディスクリプタを番号の小さい順に使用する．8 行の `open()` は `close(1)` で空きになった 1 番のファイルディスクリプタを返す．ここまでで，標準出力である 1 番のファイルディスクリプタがファイルにリダイレクトされた．次に，14 行の execve システムコールで自身が echo プログラムに変身する．

9.2.4 新しいプロセスを作る (fork システムコール)

execve システムコールはプロセスを新しいプログラムに変身させる．変身して新しいプログラムを実行したプロセスは終了してしまう．プロセスの数は減る一方である．新しいプロセスを作って，新しいプログラムを実行させる仕組みが必要である．fork システムコールは新しいプロセスを作成し自身をコピーする．つまり，**分身**を作るシステムコールである（分身の術）．次に UNIX の fork システムコールを説明する．

書式 fork の書式を示す．

```
#include <unistd.h>
pid_t fork(void);
```

解説 fork システムコールを実行した瞬間の，プロセスのコピーを作る．元のプロセスを**親プロセス**，コピーして作ったプロセスを**子プロセス**と呼ぶ．図 9.3 に模式図を示す．親プロセスと子プロセスでは PID（プロセス番号）を除き全く同じになる．CPU の状態もコピーされる（PC もコピーされる）ので，子プロセスは fork システムコールの途中から実行が開始される．

fork システムコールが終了する際，親プロセスには子プロセスの PID が返され，子プロセスには 0 が返される．プログラムはこの値を目印に自分が親か子か判断できる．エラー時は，親プロセスに -1 が返され子プロセスは作られない．

使用例 リスト 9.6 に fork システムコールを実行するプログラムの例を示す．以下ではこのプログラムの処理手順を説明する．最終的に親プロセスと子プロセスが同時に並行して実行される．

1. 6 行で親プロセスが fork システムコールを実行する．図 9.3 のように，新しいプロセス（子プロセス）が作られ，親プロセスの内容がコピーされる．子プロセスには，メモリ空間（プログラム，変数（データ），スタック），プロセスの状態（どのファイルをオープン中か，シグナルハ

リスト 9.6　fork システムコールの使用例

```
1  #include <stdio.h>          // printf, fprintf のために必要
2  #include <unistd.h>         // fork のために必要
3  int main() {
4    int x = 10;
5    int pid;
6    pid = fork();                              // この瞬間にプロセスがコピーされる
7    if (pid<0) {
8      fprintf(stderr, "forkでエラー発生\n");  // エラーの場合
9      return 1;
10   } else if (pid!=0)   {                     // 親プロセスだけが以下を実行する
11     x = 20;                                  //   親プロセスの x を書き換える
12     printf("親 pid=%d x=%d\n", pid, x);
13   } else {                                   // 子プロセスだけが以下を実行する
14     printf("子 pid=%d x=%d\n", pid, x);      //   子プロセスの x は初期値のまま
15   }
16   return 0;
17 }
18 /*  実行例
19 % ./forktest
20 親 pid=8079 x=20              // 親プロセスの出力
21 子 pid=0 x=10                 // 子プロセスの出力（xの値に注目）
22 */
```

ンドラの登録等)，CPU の状態（CPU レジスタの値，SP の値，PC の値，フラグの値）等，全ての情報がコピーされる．ただし，プロセス番号 (PID:Process ID) は親子プロセスで異なる．

2. 親プロセスは fork システムコールを完了しプログラムの実行を再開する．この時，fork システムコールの返り値は子プロセスの PID になる．
3. 子プロセスは fork システムコールを呼出した瞬間のコピーなので，fork システムコールが完了するところ（6 行）からプログラムの実行を開始する．この時，fork システムコールの返り値は 0 になる．
4. 7 行で fork システムコールでエラーが発生していないかチェックしている．
5. 10 行では fork システムコールの返り値から，自身が親プロセスか子プロセスか調べている．
6. 自身が親プロセスの場合は 11，12 行を実行する．11 行で親プロセスは変数 x を 20 に書き換える．12 行の printf() は 20 行のような出力をする．変数は自身のメモリ空間にあるので，x の書き換えは子プロセスに影響を与えない．
7. 自身が子プロセスの場合は 14 行を親プロセスと並行して実行する．子プロセスが参照する変数 x は自身のメモリ空間にあるので，親プロセスが値を書き換えた x とは別のインスタンスである．21 行のように 10 が表示される．

9.2.5 プロセスの終了と待ち合わせ

親プロセスは子プロセスを幾つか作成し，それらに同時に並行して処理を行わせる．子プロセスは処理を終えると終了する．子プロセスが処理を終えると，親プロセスは子プロセスが正常に終了したかチェックする．全ての子プロセスが正常に終了していれば処理全体が完了である．このような処理ができるように，子プロセスが処理結果と共に自身を終了する exit システムコール[*2]と，親プロセスが子プロセスの終了を待つ wait システムコールが準備されている．子プロセスは exit システムコールを実行してもすぐに消滅するわけではない．子プロセスは，親プロセスが wait システムコールを実行し終了ステータスを取り出すまで，待ち状態になる．この状態を *zombie* **状態**（表 6.3 参照）と呼ぶ．

1. *exit* **システムコール**

   exit システムコールを呼び出したプロセスを終了する．プロセスが終了する際に，処理結果を表す**終了ステータス**を親プロセスに返すことができる．

   書式 exit システムコールの書式を示す．

   ```
   #include <stdlib.h>
   void exit(int status);
   ```

   解説 自プロセスを終了する．exit を呼び出すとプロセスが終了するので exit は戻らない．`status` はプロセスの終了ステータスである．親プロセスは wait システムコールで `status` の値を受け取る．終了ステータスは下位 8bit が有効である（`0 <= status <= 255`）．

   なお C プログラムの main 関数は，スタートアップルーチンから次のように呼び出されている．

   ```
   exit(main(argc, argv, envp));
   ```

   main 関数を `return n;` で終了すると，`exit(n);` が実行されることになる．つまり，main 関数中では `return n;` と `exit(n);` が同じ意味になる．

2. *wait* **システムコール**

   親プロセスが子プロセスの終了を待つシステムコールである．

   書式 wait システムコールの書式を示す．

   ```
   #include <sys/wait.h>
   pid_t wait(int *status);
   WIFEXITED(status)   // 子プロセスがexitした(マクロ)
   WIFSIGNALED(status) // 子プロセスがシグナルで終了した(マクロ)
   WEXITSTATS(status)  // 子プロセスの終了ステータス(マクロ)
   WTERMSIG(status)    // 子プロセスを終了させたシグナルの番号(マクロ)
   ```

   解説 wait システムコールは終了した子プロセスのプロセス番号（PID）を返す．エラーが発生した場合に`-1` を返す．`status` には子プロセスが終了した理由等が格納される．status の内容は書式に掲載したマクロで読み取ることができる．

[*2] 正確には `exit()` 関数は`_exit` システムコールを呼び出すライブラリ関数である．`exit()` はファイル（ストリーム）のクローズ処理（バッファのフラッシュ等）をした後で`_exit` システムコールを呼び出す．

リスト 9.7　fork-exec のプログラム例

```
1  #include <stdio.h>                    // perror のために必要
2  #include <stdlib.h>                   // exit のために必要
3  #include <unistd.h>                   // fork, execve のために必要
4  #include <sys/wait.h>                 // wait のために必要
5  char *execpath="/bin/date";
6  int main(int argc, char *argv[], char *envp[]) {
7    int pid;
8    if ((pid=fork())<0) {               // 分身を作る
9      perror(argv[0]);                  // fork がエラーなら
10     exit(1);                          // 親プロセスをエラー終了
11   }
12   if (pid!=0) {                       // pid が 0 以外なら自分は親プロセス
13     int status;
14     while (wait(&status)!=pid)        //   子プロセスが終了するのを待つ
15       ;
16   } else {                            // pid が 0 なら自分は子プロセス
17     execl(execpath, "date", NULL);    //   date プログラムを実行（execlを使用してみた）
18     perror(execpath);                 //   exec が戻ってくるならエラー
19     exit(1);                          //   エラー時はここで子プロセスを終了
20   }
21   exit(0);                            // 親プロセスを正常終了
22 }
```

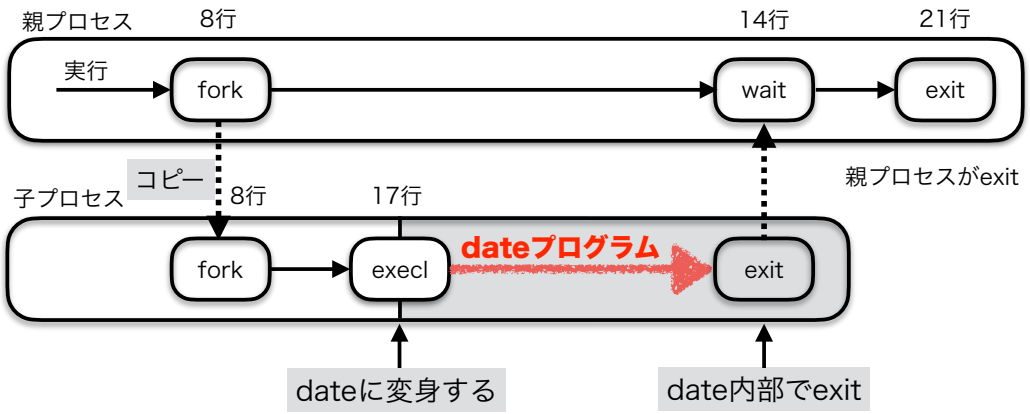


図 9.4　リスト 9.7 における fork-exec-wait-exit の関係

9.2.6　fork-exec 方式のプログラム例

リスト 9.7 に新しいプロセスで新しいプログラムを実行するプログラムの基本的な例を示す．このプログラムは，fork，exec，wait，exit システムコールを組み合わせて使用する一般的な例である．図 9.4 は，リスト 9.7 のプログラムの動きを解説したものである．

プログラムの 8 行でプロセスのコピーが作られる．12 行で自身が親プロセスか子プロセスか判定す

リスト 9.8 子プロセスで環境変数を変更しながら date を次々と実行する例

```
1  #include <stdio.h>                    // perror のために必要
2  #include <stdlib.h>                   // exit のために必要
3  #include <unistd.h>                   // fork, execve のために必要
4  #include <sys/wait.h>                 // wait のために必要
5  char *execpath="/bin/date";
6  int main(int argc, char *argv[], char *envp[]) {
7    int pid;
8    for (int i=1; argv[i]!=NULL; i++) {
9      if ((pid=fork())<0) {             // 分身を作る
10       perror(argv[0]);                // fork がエラーなら
11       exit(1);                        // 親プロセスをエラー終了
12     }
13     if (pid!=0) {                     // pid が 0 以外なら自分は親プロセス
14       int status;
15       while (wait(&status)!=pid)      //   子プロセスが終了するのを待つ
16         ;
17     } else {                          // pid が 0 なら自分は子プロセス
18       putenv(argv[i]);                //   環境変数を変更する
19       execl(execpath, "date", NULL);  //   date プログラムをロード・実行
20       perror(execpath);               //   execl が戻ってくるならエラー
21       exit(1);                        //   エラー時はここで子プロセスを終了
22     }
23   }
24   exit(0);                            // 親プロセスを正常終了
25 }
26 /* 実行例
27 % ./forkexec2 LC_TIME=ja_JP.UTF-8 LC_TIME=ru_RU.UTF-8 TZ=Cuba
28 2023年 4月 2日 日曜日 10時47分31秒 JST
29 воскресенье, 2 апреля 2023 г. 10:47:31 (JST)
30 Sat Apr  1 21:47:31 CDT 2023
31 */
```

る．親プロセスは 14 行の wait() で子プロセスが終了するまでブロックする．子プロセスは 17 行で date プログラムをロード・実行する．date プログラムの内部で exit システムコールが実行され，子プロセスが終了する．子プロセスが終了したら，親プロセスの wait() が終了ステータスを受け取り次に進む．親プロセスは 21 行の exit() で正常終了する．

9.2.7 環境変数を変更しながら fork-exec を繰り返す例

リスト 9.8 に少し実用的な例を示す．このプログラムは，実行例に示すようにコマンド行引数に指示された環境変数の変更を行った上で date プログラムを次々に実行する．環境変数の変更は子プロセス側で行うようになっているので，**親プロセスの環境変数は変化しない**．

リスト 9.9 は，環境変数の変更を親プロセスが fork する前に行うように変更したものである．リス

リスト 9.9　親プロセスで環境変数を変更しながら date を次々と実行する例

```
#include <stdio.h>                    // perror のために必要
#include <stdlib.h>                   // exit のために必要
#include <unistd.h>                   // fork, execve のために必要
#include <sys/wait.h>                 // wait のために必要
char *execpath="/bin/date";
int main(int argc, char *argv[], char *envp[]) {
  int pid;
  for (int i=1; argv[i]!=NULL; i++) {
    putenv(argv[i]);                  // 環境変数を変更する
    if ((pid=fork())<0) {             // 分身を作る
      perror(argv[0]);                // fork がエラーなら
      exit(1);                        // 親プロセスをエラー終了
    }
    if (pid!=0) {                     // pid が 0 以外なら自分は親プロセス
      int status;
      while (wait(&status)!=pid)      //   子プロセスが終了するのを待つ
        ;
    } else {                          // pid が 0 なら自分は子プロセス
      execl(execpath, "date", NULL);  //   date プログラムを実行（execlを使用してみた）
      perror(execpath);               //   exec が戻ってくるならエラー
      exit(1);                        //   エラー時はここで子プロセスを終了
    }
  }
  exit(0);                            // 親プロセスを正常終了
}
/* 実行例
% ./forkexec3 LC_TIME=ja_JP.UTF-8 LC_TIME=ru_RU.UTF-8 TZ=Cuba
2023年 4月 2日 日曜日 10時49分17秒 JST
воскресенье, 2 апреля 2023 г. 10:49:17 (JST)
суббота, 1 апреля 2023 г. 21:49:17 (CDT)
*/
```

ト 9.8 の実行結果との違いに注目して欲しい.

第 10 章

UNIX シェル

本章では簡易版の UNIX シェル（以下では myshell[*1] と呼ぶ）を紹介する．これの内容を理解し，更に，改造したりすることで，「fork-exec 方式」，「環境変数」，「リダイレクト」等の理解を深める．最後の課題で，myshell に幾つかの機能を追加できるようになることを目標とする．

10.1 UNIX のシェルとは

UNIX のシェルは CLI（Command Line Interface）方式のコマンドインタプリタ[*2]である．UNIX，Linux，macOS，WSL（Windows Subsystem for Linux）等で使用される．ユーザが入力したコマンド行を解析し実行する．またコマンド行だけではなく，コマンド列（シェルスクリプト）を書いたファイルを実行する機能も備えており，処理の自動化にも役立つ．実際に UNIX 系 OS の多くでは，システム起動時に自動的にサービスを立ち上げる等の処理に，シェルスクリプトが多用されている．8 章で紹介した`/etc/zprofile` や`.zprofile` もシェルスクリプトの一種である．

シェルはユーザプログラムの一種であり，ユーザプロセスとして実行される．macOS では `sh`，`bash`，`ksh`，`zsh`，`csh`，`tcsh` 等，多種類の UNIX のシェルが予めインストールされており利用可能である．macOS の場合，標準のログインシェル[*3]は `zsh` になっているが，好みで変更する[*4]ことも可能である．UNIX シェルは最低でも次の機能を持っている．

1. 外部コマンド（プログラム）を起動する機能
2. カレントディレクトリを変更する機能
3. 環境変数などの変数管理機能
4. 入出力のリダイレクト機能やパイプ機能（<，>，>>，|）
5. ジョブ制御機能（jobs，fg，bg，&，; など）
6. ファイル名の展開機能（ワイルドカード）
7. 繰り返しや条件判断機能
8. スクリプトの実行機能（処理の自動化）

[*1] `https://raw.githubusercontent.com/tctsigemura/SystemPrograming/master/myshell.c`

[*2] macOS の Finder や，Windows の Explorer は GUI 版のシェルである．

[*3] ターミナルを開いたとき最初に使用されるシェルのこと．

[*4] chsh コマンドで変更する．

リスト 10.1　メインループ（main()）

```
1  int main() {
2    char buf[MAXLINE+2];                          // コマンド行を格納する配列
3    char *args[MAXARGS+1];                        // 解析結果を格納する文字列配列
4    for (;;) {
5      printf("Command: ");                        // プロンプトを表示する
6      if (fgets(buf,MAXLINE+2,stdin)==NULL) {     // コマンド行を入力する
7        printf("\n");                             // EOF なら
8        break;                                    //   正常終了する
9      }
10     if (strchr(buf, '\n')==NULL) {              // '\n'がバッファにない場合は
11       fprintf(stderr, "行が長すぎる\n");        //   コマンド行が長すぎたので
12       return 1;                                 //   異常終了する
13     }
14     if (!parse(buf,args)) {                     // コマンド行を解析する
15       fprintf(stderr, "引数が多すぎる\n");      //   文字列が多すぎる場合は
16       continue;                                 //   ループの先頭に戻る
17     }
18     if (args[0]!=NULL) execute(args);           //   コマンドを実行する
19   }
20   return 0;
21 }
```

10.2　簡易 UNIX シェル（myshell）

myshell は C 言語で 70 行以内で記述可能な簡易版 UNIX シェルである．ここでは，シェルの仕組みを理解するための教材として用いる．コマンド行は空白区切りの文字列とする．myshell は以下の二つの機能しか持たない．

1. 外部コマンド（プログラム）を起動する機能
2. カレントディレクトリを変更する機能

10.2.1　基本構造（main() 関数）

myshell は，コマンド行の**入力**，**解析**，**実行**を入力が EOF になるまで繰り返す．これを，リスト 10.1 の main() 関数で行う．main() 関数の動作は次の通りである．

1. 6 行で fgets() 関数[*5]を用いてコマンド行を buf 配列に**入力**する．buf 配列には'\n'，'\0' で終端された文字列が格納されるはずである．
2. 10 行では，strchr() 関数[*6]を用いてバッファに'\n' が格納されていることを確認する．格納されていない場合は，「コマンド行が長すぎてバッファに入り切らない」エラーが発生した場合であ

[*5] fgets() 関数は C 言語の標準ライブラリ関数である．
[*6] strchr() 関数は C 言語の標準ライブラリ関数である．

リスト 10.2 コマンド行の解析ルーチン（parse()）

```
1  int parse(char *p, char *args[]) {              // コマンド行を解析する
2    int i=0;                                      // 解析後文字列の数
3    for (;;) {
4      while (isspace(*p)) *p++ = '\0';            // 空白を'\0'に書換える
5      if (*p=='\0' || i>=MAXARGS) break;          // コマンド行の終端に到達で終了
6      args[i++] = p;                              // 文字列を文字列配列に記録
7      while (*p!='\0' && !isspace(*p)) p++;       // 文字列の最後まで進む
8    }
9    args[i] = NULL;                               // 文字列配列の終端マーク
10   return *p=='\0';                              // 解析完了なら 1 を返す
11 }
```

る．簡易版のシェルなのでエラーからの復旧は諦めて，エラーメッセージを表示し終了する．

3. 14 行では，後で説明する parse() 関数を用いて buf 配列のコマンド行を**解析**し，execve システムコールに渡す args 配列を作る．もしも，コマンド行の文字列が多すぎて配列に格納しきれない時は，parse() 関数が 0 を返すのでエラーメッセージを表示して入力を無視する．
4. 18 行では，args 配列にコマンドが格納されていることを確認した後，後で説明する execute() 関数に依頼して args 配列のコマンドを**実行**する．

10.2.2 コマンド行の解析（parse() 関数）

parse() 関数がコマンド行を解析し execve() システムコールの第 2 引数で使用できる argv 形式に変換する．リスト 10.2 に myshell の parse() 関数を示す．動作は次の通りである．

1. コマンド行に入力された文字列と，解析結果を格納する文字列配列を引数に呼出される．
2. 4 行では，isspace() 関数[*7]を用いて文字列に先行する空白を読み飛ばす（ポインタ p を進める）．その際，空白を文字列の終端記号である'\0' に置き換えておく．
3. 5 行では，解析処理の完了を判断する．空白を読み飛ばした後で最初に見つかった文字が'\0' なら，コマンド行の最後まで達しているので for(;;) ループを脱出し解析を終了する．または，args 配列のサイズを超えそうになっていた場合も，文字列が多すぎてこれ以上処理できないのでループを終了する．
4. 処理が 6 行に進むのは，新しく次の文字列が見つかった場合である．新しい文字列の開始位置（ポインタ）を args 配列に記録する．
5. 7 行では，isspace() 関数を用いて文字列を終端する空白を探す．終端が見つかったら for(;;) ループの先頭に戻り，4 行で空白を'\0' に書換え C 言語の文字列として完成する．
6. 9，10 行では，文字列配列に配列の終端を表す NULL を書込み parse() 関数を終了する．その際，コマンド行の最後まで解析が終わっていれば 1（true）を，そうでなければ 0（false）を返す．

[*7] isspace() 関数は空白文字を判定する C 言語の標準ライブラリ関数である．スペース，タブ，改行などを空白と判定する．

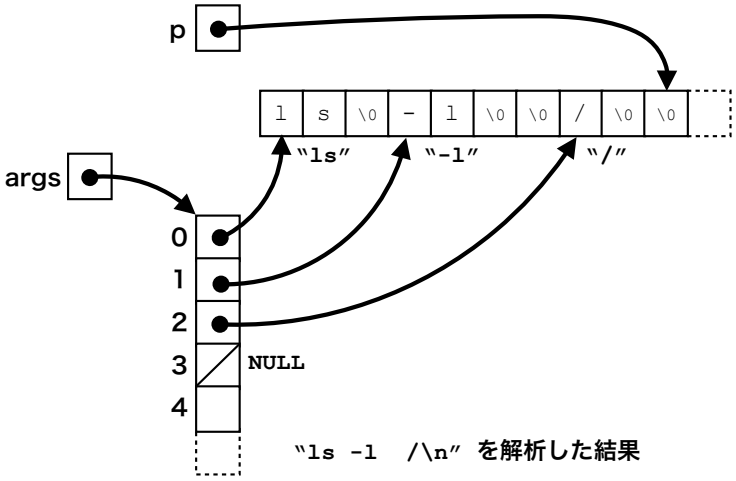


図 10.1 parse() 関数の実行結果例

図 10.1 に parse() 関数実行後のデータ構造の例を示す．この例は，コマンド行文字列 "ls␣-l␣␣/\n" を解析した後のデータ構造である．渡された文字列の空白を'\0' で上書きし"ls", "-l", "/" の 3 つの文字列が作られている[*8]．文字列配列の終端は NULL で表現する．args 配列は各文字列の先頭を指すポインタを格納する[*9]ことで文字列配列を表現する．文字列の最後の '\n' は '\0' に書き換わっている[*10]．

10.2.3 コマンドの実行（execute()）

リスト 10.3 に myshell の execute() 関数を示す．この関数は parse() 関数が作った args 配列を受け取り，内部コマンドなら自分で実行し，外部コマンドなら子プロセスを作って実行させる．

2 行ではコマンドの名前を調べている．cd コマンドは内部コマンドなので，5 行で自ら chdir() システムコールを実行している[*11]．8 行以下は外部コマンドの処理である．10 行で子プロセスを作り，15 行で子プロセスがコマンドを実行する．親プロセスは 19 行で子プロセスの終了を待つ．

15 行では execve() システムコールの代わりに execvp() 関数を用いているので，PATH 環境変数を用いたプログラムの自動的な探索が行われる．

[*8] リスト 10.2 の 4 行で'\0' に書換えている．
[*9] リスト 10.2 の 6 行でポインタを args 配列に格納している．
[*10] '\n' は isspace() 関数により空白と判断される．
[*11] 内部コマンドを追加するときは，cd コマンドと同様に，ここに追加する．

リスト 10.3 コマンドの実行ルーチン（execute()）

```
1  void execute(char *args[]) {                    // コマンドを実行する
2    if (strcmp(args[0], "cd")==0) {               // cd（内部コマンド）
3      if (args[1]==NULL || args[2]!=NULL) {       //   引数を確認して
4        fprintf(stderr,"Usage: cd DIR\n");        //     過不足ありなら使い方表示
5      } else if (chdir(args[1])<0) {              //   親プロセスが chdir する
6        perror(args[1]);                          //     chdirに失敗したらperror
7      }
8    } else {                                      // 外部コマンドなら
9      int pid, status;
10     if ((pid = fork()) < 0) {                   //   新しいプロセスを作る
11       perror("fork");
12       exit(1);
13     }
14     if (pid==0) {                               //   子プロセスなら
15       execvp(args[0], args);                    //     コマンドを実行
16       perror(args[0]);
17       exit(1);
18     } else {                                    //   親プロセスなら
19       while (wait(&status) != pid)              //     子の終了を待つ
20         ;
21     }
22   }
23 }
```

システムプログラミング Ver. 1.2.1

発行年月　２０２４年０３月　Ver.1.2.1
発　　行　独立行政法人国立高等専門学校機構
徳山工業高等専門学校
情報電子工学科　重村哲至
〒745-8585 山口県周南市学園台
sigemura@tokuyama.ac.jp