第4章　ファイル関連のシステムコール

4.1　システムコールとエラー処理

第 1.6 節で述べたように，システムコールとはオペレーティングシステムカーネルが提供するサービスを直接呼び出す機能である．FreeBSD では 200 種類以上，Linux では 400 種類以上のシステムコールが用意されているようである．プログラム上はライブラリ関数もシステムコールも区別がないが，システムコールはカーネルの機能を直接使用するため，ライブラリ関数に比べて機能がより基本的である．

4.1.1　エラーの原因の特定

システムコールは入出力やプロセス制御などに関するものが多いため，エラーが発生する可能性があることを常に念頭に置いてプログラミングを行う必要がある．システムコールの実行中にエラーが発生すると通常は-1 または空ポインタ (NULL) が返されるが，これだけではエラーの原因を特定することができない．そこで，システムコール実行中のエラー原因を示すために “int errno” という外部変数が用意されており，システムコール実行中のエラーの原因を表す数値がこの変数に設定されるようになっている．errno という変数を使用するには#include <errno.h>が必要である．BSD 系の UNIX の場合は，errno の値がどのようなエラーを表しているかは “man 2 intro” を実行することによって調べることができる．Linux の場合は “man 3 errno” を実行すればよい．したがって，システムコールが-1 を返した場合は，errno という変数の値を調べることによってエラーの原因に対応した処理を行うことができる．

エラーの原因を表示するには，perror() というライブラリ関数を利用すると便利である．perror() の構文を図 4.1 に示す．perror() は string で示された文字列を表示した後に，直前に実行したシステムコールのエラーの原因を表示する．perror() を利用するには#include <stdio.h>が必要である．図 4.2 に，open() というシステムコールを使用し，perror() を使用してエラーの原因を表示している例を示す．open() の機能については第 4.3.1 節で述べる．

この例では 13 行目の open() によって，もし指定したファイルが存在しない場合はファイルを作成しようとしている．この open() がエラーとなるのは，指定したファイルが既に存在している場合や，そもそもファイルを作成する権利がない場合などさまざまな原因が考えられる．そこで 14 行目で，エラーの原因が「既にファイルが存在する」であるかを確認している (if (errno != EEXIST))．エラーの原因が「既にファイルが存在している」ではないときはこれ以上処理を進めることはできないので，15 行目でエラーの原因を表示し，16 行目でプログラムの実行を終了している．exit() に関しては次節で述べ

```
01: #include <stdio.h>
02:
03: void
04: perror(const char *string);
```

図 4.1: perror() の構文

```
01: #include <stdio.h>
02: #include <errno.h>
03: #include <fcntl.h>
04:
05: main()
06: {
07:     int fd;
08:     char filename[80];
09:
10:     /* filename[] へのファイル名の設定 */
11:     ...
12:     /* ファイルが存在しない場合はファイルを作成 */
13:     if ((fd = open(filename, O_WRONLY|O_CREAT|O_EXCL, 0644)) < 0) {
14:         if (errno != EEXIST) {
15:                 perror("open");
16:                 exit(1);
17:         }
18:         /* フィルがすでに存在していた */
19:         fprintf(stderr, "overwrite (y/n)?: ");
20:         /* 上書きしていいかの確認 */
21:         ...
22:         /* 存在していたファイルの内容を消去してオープン */
23:         if ((fd = open(filename, O_WRONLY|O_CREAT|O_TRUNC, 0644))
                                                                < 0) {
24:                 perror("open");
25:                 exit(1);
26:         }
27:     }
28:     /* 正常な処理の続き */
29:     ...
30: }
```

図 4.2: システムコールのエラー処理の例

```
01: #include <stdlib.h>
02:
03: void
04: exit(int status);
```

図 4.3: exit() の構文

る．ファイルが既に存在してる場合には上書きしてもいいかを確認した上で，23 行目で存在しているファイルの内容を消去してオープンしている．この open() がエラーの場合は 24 行目でエラーの原因を表示し，25 行目でプログラムの実行を終了している．正常にファイルを作成できた場合は，28 行目以降で正常な処理を続ける．

4.2 exit(): プログラムの終了

プログラム実行中に致命的なエラーが発生し，それ以上の実行が不可能になった場合は exit() 関数を呼び出してプログラムの実行を終了することになる．exit() の構文を図 4.3 に示す．プログラムが正常に終了した場合は status に EXIT_SUCCESS (= 0) を設定し，異常終了の場合は EXIT_FAILURE (= 1) を設定する．status & 0xff という値が親プロセスに返される．親プロセスに関しては第 5 章参照のこと．

exit() が返す値 (exit コード) については，<sysexits.h>というファイルにさらに細かい値が定義されている．BSD 系の UNIX の場合は，“man sysexits” を実行することにより，exit コードについて調べることができる．

4.3 ファイル入出力のシステムコール

ファイルの入出力を行うには，基本的にはまず open() でファイルをオープンし，read() で読み込み，write() で書き込み，入出力が済んだら close() でファイルをクローズする，という流れとなる．以下，それぞれのシステムコールの概略を述べる．繰り返すが，詳細な仕様は man コマンドで調べ，エラーにもきちんと対処することが大事である．

4.3.1 open()：ファイルのオープン

open() はデータの入出力のためにファイルをオープンするためのシステムコールである．open() の構文を図 4.4 に示す．path によってオープンしたいファイルのパス名を指定する．これは絶対パス名でも相対パス名でもよい．flags はオープンの仕方を指示する．主な flags の定義を表 4.1 に示す．論理和をとって複数の flags を指定することも可能である．

ファイルを作成する場合は，3 つ目の引数として “mode_t mode” を指定する．mode は作成するファイルのパーミッションを指定するものであり，rwxrwxrwx のうちの設定する

表 4.1: open() の主な flags

flags	意味
O_RDONLY	read-only でオープン
O_WRONLY	write-only でオープン
O_RDWR	read, write 可能でオープン
O_APPEND	ファイルの最後に書き込む
O_CREAT	ファイルが存在しない場合は作成
O_TRUNC	ファイルのサイズを 0 にする

```
01: #include <fcntl.h>
02:
03: int
04: open(const char *path, int flags, ...);
```

図 4.4: open() の構文

ビットを指定する．たとえば，rw-r--r--，つまりファイルの所有者は read/write が可能で同一グループとその他のユーザは read のみ可能，というパーミッションを指定する場合は 0644 とする．0644 のように先頭に 0 がついている定数は 8 進数であることを思い出して欲しい．

open() は整数値であるファイル記述子 (file descriptor) を返す．ファイル記述子には “int fd” という変数が使われることが多い (図 4.2 参照)．ファイル記述子とはオープンされているファイルを識別するための数値である．ファイルへの read/write においては，ファイル記述子により対象とするファイルを指定する．エラーの場合，open() は-1 を返し，エラーの原因は変数 errno に設定される．

4.3.2 read()：ファイルからの読み込み

read() はファイルからデータを読み込むためのシステムコールである．read() の構文を図 4.5 に示す．d には open() の返り値であるファイル記述子を指定する．buf はデータを格納するためのメモリオブジェクトを指すポインタである．ntytes は読み込むバイト数である．read() を続けて実行すると，以前読み込んだデータの次のデータが読み込まれる．

ここで注意しなければならないのは，第 3.4 節で述べたように，あらかじめデータを格納するメモリオブジェクトを用意し，buf がその先頭を指すようにしておくことである．read() が自動的にメモリオブジェクトを割り当てることはない．

read() は正常に読み込んだバイト数を返す．EOF (end of file) のときには 0 を返す．エラーのときには-1 を返し，エラーの原因は変数 errno に設定される．

```
01: #include <sys/types.h>        /* Linuxでは不要 */
02: #include <sys/uio.h>          /* Linuxでは不要 */
03: #include <unistd.h>
04:
05: ssize_t
06: read(int d, void *buf, size_t nbytes);
```

図 4.5: read() の構文

```
01: #include <sys/types.h>        /* Linuxでは不要 */
02: #include <sys/uio.h>          /* Linuxでは不要 */
03: #include <unistd.h>
04:
05: ssize_t
06: write(int d, const void *buf, size_t nbytes);
```

図 4.6: write() の構文

4.3.3 write(): ファイルへの書き込み

write() はファイルにデータを書き込むためのシステムコールである．write() の構文を図 4.6 に示す．d には open() の返り値であるファイル記述子を指定する．buf は書き込むデータが格納されているメモリオブジェクトを指すポインタである．ntytes は書き込むバイト数である．write() を続けて実行すると，以前書き込んだデータの次に新しいデータが書き込まれる．

write() は正常に書き込んだバイト数を返す．エラーのときには-1 を返し，エラーの原因は変数 errno に設定される．

4.3.4 close(): ファイルのクローズ

close() はオープンされているファイルをクローズするためのシステムコールである．close() の構文を図 4.7 に示す．d には open() の返り値であるファイル記述子を指定する．正常の場合，close() は 0 を返し，エラーの場合は-1 を返す．エラーの原因は変数 errno に設定される．

プロセスの終了時にはオープンされているファイルはすべてクローズされる．しかしプロセスが同時にオープンできるファイル数は限られているので，用の済んだファイルは明示的にクローズするのが望ましい．

```
01: #include <unistd.h>
02:
03: int
04: close(int d);
```

図 4.7: close の構文

```
01: #include <sys/tuypes.h>        /* BSD では不要 */
02: #include <unistd.h>
03:
04: off_t
05: lseek(int d, off_t offset, int whence);
```

図 4.8: lseek() の構文

4.3.5 lseek(): 読み込み/書き込み位置の変更

lseek() は読み込み/書き込み位置 (オフセット) を変更するためのシステムコールである．read() や write() はシーケンシャルにデータを読み込んだり書き込んだりするが，lseek() を用いることによりファイルの任意の位置からの読み込み/書き込みが可能になる．図 4.8 に lseek() の構文を示す．d には open() の返り値であるファイル記述子を指定する．whence の値によってオフセットは以下のように設定される．

- whence が SEEK_SET の場合には，オフセットは offset に設定される．
- whence が SEEK_CUR の場合には，オフセットは現在値に offset が加えられたものになる．
- whence が SEEK_END の場合には，オフセットは現在のファイルサイズに offset を加えたものになる．

なお，whence が SEEK_CUR や SEEK_END の場合，offset は負の値でもよい．

正常の場合，lseek() は結果としてのオフセットの値を返す．エラーの場合は -1 を返す．エラーの原因は変数 errno に設定される．

4.3.6 read(), write() のブロックサイズ

システムコールはカーネルの機能を直接呼び出すため，ハードウェアの性質やオペレーティングシステムの動作を考慮して使用しないと効率が悪くなることがある．read() や write() はその例である．第 2.2 節で述べたように，ハードディスクは固定長のブロックから構成されており，入出力はブロック単位で行われる．したがって，read() や write() はディスクブロックの境界位置からブロックサイズ単位で実行する方が効率が良い．最適

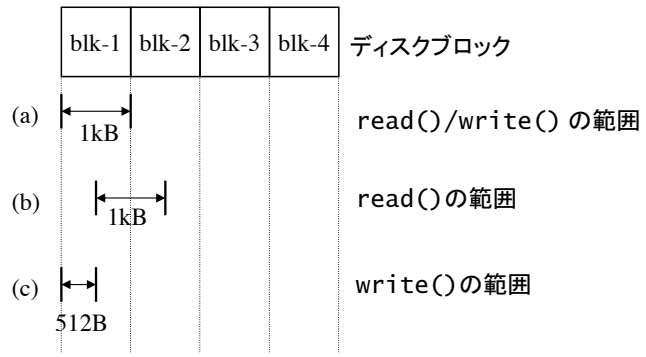

図 4.9: read(), write() の例

なブロックサイズを知るには，stat() または fstat() を用いて st_blksize を得ればよい (第 4.4.1 参照).

ここでは説明を簡単にするためにディスクブロックサイズを 1 KB とし，read() や write() で指定するディスクブロックのキャッシュが存在しない場合を考える．図 4.9 に 3 種類の read(), write() の例を示す．(a) はディスクブロックの境界位置から 1 ブロック分を read(), write() している．read() の場合は blk-1 をディスクから読み込むことになり，write() の場合は blk-1 のデータをすべて上書きすることになる．すなわちどちらも 1 度のディスクアクセスで済む.

これに対して (b) ではディスクブロックの境界をまたがって 1 ブロック分の read() を行っている．この場合，blk-1 と blk-2 の両方を読み込まないと read() の処理が完了しない．すなわち，(a) の場合は 1 度のディスクアクセスで済むが，(b) の場合は 2 度のディスクアクセスが必要となってしまい，効率が悪い.

(c) はディスクブロックのサイズ以下の書き込みを行っている．前述したようにハードディスクはブロック単位でしか入出力ができない．つまり (c) のようにディスクブロックの一部分だけに直接データを書き込むことはできない．したがって，まず blk-1 のデータをディスクから読み込み，このデータのうちの指定された部分を変更し，最後に 1 ブロック分のデータを blk-1 に書き戻すことになる．すなわち 2 度のディスクアクセスが必要になる．(a) のようにディスクブロック全体を書き込む場合は 1 度のディスクアクセスで済むが，(c) のようにディスクブロックの一部分だけを書換える場合は 2 度のディスクアクセスが必要となってしまう.

4.4 ファイルの属性に関するシステムコール

ls -l コマンドを実行すると，たとえば以下のような結果を表示する.

```
-rw-r--r--  1 tera  TLAB   406 Oct 23 11:33 a.c
```

各フィールドの意味を念のため説明しておく．先頭の “-” のフィールドはファイルのタイプを表す．“d” はディレクトリであり，“-” は通常のファイルである．“s” はシンボリックリンクである．“c” はキャラクタ型デバイスであり，“b” はブロック型デバイスであ

```
01: #include <sys/types.h>        /* BSD では不要 */
02: #include <sys/stat.h>
03: #include <unistd.h>           /* BSD では不要 */
04:
05: int
06: stat(const char *path, struct stat *sb);
07:
08: int
09: fstat(int fd, struct stat *sb);
```

図 4.10: stat() および fstat() の構文

る．これに続く “rw-r--r--” はファイルの所有者，グループ，その他のユーザに対する read/write/execute のパーミッションを表す．これに続く “1” はファイルへのリンク数を表す．次の “tera” はファイルの所有者であり，“TLAB” はファイルが属するグループである．次の “406” はバイト単位のファイルのサイズである．次の “Oct 23 11:33” は最後に更新した時刻である．最後の “a.c” はファイル名である．

このように，ファイルはデータ以外に上述のような属性情報を持つ．ls コマンドはファイルの属性情報を読み取って表示している．

4.4.1 stat(), fstat(): ファイルの属性の取得

stat() および fstat() はファイルの属性を取得するためのシステムコールである．stat() および fstat() の構文を図 4.10 に示す．stat() は対象となるファイルを文字列 (char *path) で指定するのに対し，fstat() はファイル記述子 (int fd) で指定する．機能は両方とも同じである．

sb が指す領域にファイルの属性情報が格納される．struct stat は<sys/stat.h>内で定義されている．sb は struct stat 型の変数を指すポインタである．何度も同じような指摘を繰り返すが，このポインタはメモリオブジェクトを指していなければならない．struct stat のメンバーを表 4.2 に示す．

i-node 番号とは，ファイルシステム内部におけるファイルの識別子となる整数である．ユーザはパス名によってファイルを識別するが，ファイルシステム内部では i-node 番号という整数でファイルを識別する．st_dev と st_ino の対によってシステム内でファイルを一意に指定することができる．

正常な場合，stat() および fstat() は 0 を返し，エラーの場合は-1 を返す．エラーの原因は変数 errno に設定される．

表 4.2: struct stat のメンバー

ファイルシステム関連	
st_dev	ファイルを格納しているデバイスの識別子
st_ino	ファイルの i-node 番号
st_nlink	ファイルへのハードリンクの数
時刻関連	
st_atime	ファイルのデータが最後にアクセスされた時刻
st_mtime	ファイルのデータが最後に変更された時刻
st_ctime	ファイルの属性データが最後に変更された時刻
st_birthtime	i-node が最初に作成された時刻
サイズ関連	
st_size	バイト単位のファイルのサイズ
st_blksize	入出力における最適なブロックサイズ
st_blocks	512 バイト単位のブロック数
アクセス関連	
st_uid	ファイルの所有者のユーザ識別子
st_gid	ファイルのグループ識別子
st_mode	ファイルのパーミッション

4.5 ディレクトリ情報の読み出し

ディレクトリとは，ファイル名と i-node 番号の対応関係を保持するファイルである．第 4.4.1 節で述べたように，カーネル内部ではファイルは i-node 番号という整数で識別される．カーネルは，ユーザが指定した文字列のパス名をディレクトリにアクセスすることで i-node 番号に変換し，ディスク上のファイルにアクセスする．ディレクトリ情報を読み出すためのライブラリ関数として opendir(), readdir() などが用意されている．

4.5.1 opendir(): ディレクトリのオープン

opendir() の構文を図 4.11 の 04，05 行目に示す．filename はディレクトリ名を表す文字列へのポインタである．正常の場合，opendir() はディレクトリストリームへのポインタ (DIR *) を返す．エラーの場合には NULL を返す．

4.5.2 readdir(): ディレクトリエントリの読み込み

readdir() の構文を図 4.11 の 07，08 行目に示す．dirp は opendir() の返り値であるディレクトリストリームへのポインタである．ライブラリ内部にディレクトリエントリの情報を格納するメモリオブジェクトが確保され，readdir() を呼び出すたびに次のディレクトリエントリの情報がこのメモリオブジェクトに格納される．readdir() はこのメモリ

```
01: #include <sys/types.h>        /* BSD では不要 */
02: #include <dirent.h>
03:
04: DIR *
05: opendir(const char *filename);
06:
07: struct dirent *
08: readdir(DIR *dirp);
09:
10: int
11: closedir(DIR *dirp);
```

図 4.11: opendir() と readdir() の構文

オブジェクトを指す struct dirent 型のポインタを返す．ディレクトリの末尾に達した場合は NULL を返す．

struct dirent の定義については “man 5 dir” を実行するにとにより得ることができるが，通常利用するのはファイル名だけであろう．ファイル名は struct dirent のメンバー “char d_name[]” に格納されている．

4.5.3 closedir(): ディレクトリのクローズ

ディレクトリ情報の利用が済んだら，closedir() を呼び出してディレクトリをクローズする．closedir() の構文を図 4.11 の 10，11 行目に示す．dirp は opendir() の返り値であるディレクトリストリームへのポインタである．

4.6 ストリームとしてのファイル入出力

第 4.3 節は通常のファイルやデバイスを “ファイル” として扱った場合のシステムコールについて述べた．本節では通常のファイルやデバイスを “ストリーム” として扱う場合について述べる．

4.6.1 fopen(): ストリームとしてのファイルのオープン

fopen() は通常のファイルやデバイスをストリームとしてオープンするための関数である．fopen() の構文を図 4.12 に示す．filename でファイルを指定する．mode でオープンするときのモードを文字列で指定する．表 4.3 に mode で指定できる文字列を示す．fopen() はファイルポインタ (FILE *) を返す．エラーの場合は NULL を返し，エラーの原因は外部変数 errno に設定される．

```
01: #include <stdio.h>
02:
03: FILE *
04: fopen(const char *restrict filename, const char *restrict mode);
```

図 4.12: fopen() の構文

表 4.3: fopen() の mode の値と機能

mode	機能
"r"	read 用にオープン．ストリームはファイルの先頭を指す．
"r+"	read および write 用にオープン．ストリームはファイルの先頭を指す．
"w"	write 用にオープン．ファイルの内容は消去される．ファイルが存在しない場合は新規に作成される．ストリームはファイルの先頭を指す．
"w+"	read および write 用にオープン．ファイルの内容は消去される．ファイルが存在しない場合は新規に作成される．ストリームはファイルの先頭を指す．
"a"	write 用にオープン．ファイルが存在しない場合は新規に作成される．ストリームはファイルの末尾を指す．
"a+"	read および write 用にオープン．ファイルが存在しない場合は新規に作成される．ストリームはファイルの末尾を指す．

```
01: #include <stdio.h>
02:
03: size_t
04: fread(void *restrict ptr, size_t size, size_t nitems,
          FILE *restrict stream);
```

図 4.13: fread() の構文

```
01: #include <stdio.h>
02:
03: size_t
04: fwrite(void *restrict ptr, size_t size, size_t nitems,
          FILE *restrict stream);
```

図 4.14: fwrite() の構文

```
01: #include <stdio.h>
02:
03: int
04: fclose(FILE *stream);
```

図 4.15: fclose() の構文

4.6.2 fread(): ストリームからのバイト列の入力

fread() の構文を図 4.13 に示す．fread() は stream で示されるストリームから，size バイトのオブジェクトを nitems 個読み込み，ptr が示す領域に格納する関数である．

fread() は読み込んだオブジェクト数を返す．エラーの場合または EOF の場合は 0 を返す．0 が返ってきたときの理由を調べるには，feof() や ferror() を使用する．

4.6.3 fwrite(): ストリームへのバイト列の出力

fwrite() の構文を図 4.14 に示す．fwrite() は stream で示されるストリームに，ptr が示す領域に格納されている size バイトのオブジェクトを nitems 個書き込む関数である．

fwrite() は書き込んだオブジェクト数を返す．エラーの場合は 0 を返す．

4.6.4 fclose(): ストリームのクローズ

fclose() の構文を図 4.15 に示す．fclose() は stream で示されるストリームをクローズする関数である．fclose() は正常の場合は 0 を返し，エラーの場合は EOF を返す．エラーの原因は外部変数 errno に設定される．

練習問題

1. 次のような場合に `open()` がどのようなエラーを返すかを確認しなさい．
   - 存在しないファイルを指定した．
   - ディレクトリを指定した．
   - write のパーミッションがないファイルを書き込みのためにオープンしようとした．
   - ファイル作成の権限がないディレクトリでファイルを作成しようとした．
2. 次のような場合に `open()` がどのような動作をするかを確認しなさい．`open()` の実行がブロックされてしまう場合は，ブロックされないように `flags` を設定し，どのようなエラーになるかを確認しなさい．
   - shared ロックがかかっているファイルを，shared ロックを指定してオープンしようとした．
   - shared ロックがかかっているファイルを，exclusive ロックを指定してオープンしようとした．
   - exclusive ロックがかかっているファイルを，shared ロックを指定してオープンしようとした．
   - exclusive ロックがかかっているファイルを，exclusive ロックを指定してオープンしようとした．
3. `cat` コマンドのように，コマンドラインの引数で指定したファイル (複数可) の内容を標準出力に出力するコマンド `mycat` を作成しなさい．
4. `cp` コマンドのように，コマンドラインの第 1 引数で指定したファイルの内容を第 2 引数のファイルにコピーするコマンド `mycp` を作成しなさい．ただし，第 2 引数で指定したファイルが存在する場合は，上書きしていいかをユーザに確認するようにしなさい．
5. 上記の問題において，1024 バイトごとに read()/write() する場合と，1 バイトごとに read()/write() する場合の実行時間を比べなさい．また，read()/write() の代わりに fread()/fwrite() を使用した場合はどうなるかを調べなさい．
6. `ls` コマンドのように，現在のディレクトリに存在するファイルに関するさまざまな情報を表示するコマンド `myls` を作成しなさい．ただし，表示内容は `ls -l` の実行結果と同じになるようにしなさい．