# 10.1 ディレクトリの内容を読む

まずは、ディレクトリエントリをリストするAPIを紹介します。ディレクトリも基本的には普通のファイルと似たようなものです。つまり、open()してread()してclose()することができます。

ただ、ディレクトリは普通のファイルと違い、中に何でも記録できるというわけではありません。ディレクトリに記録されているのは、ファイル1つの情報を表す構造体の列です。ですから、そこから読み込めば構造体の列が読み出せます（図10.1）。


図10.1　ディレクトリにつながるストリーム

つまり、ディレクトリもバイト列であるのですが、それと同時に構造体の列でもあるわけです。ディレクトリの中身を表現するために使われるこの構造体を、**ディレクトリエントリ**（directory entry）と呼びます。

普通のプログラムにしてみれば、ディレクトリを普通のファイルと同じようにopen()やread()で操作するよりも、ディレクトリエントリの列であることをもっと前面に押し出したAPIで操作できるほうが便利でしょう。そして、実際にLinuxのAPIはそのようになっています。

ディレクトリエントリのストリームを扱うAPIは普通のファイルを扱うAPIと似ていて、openとreadとcloseの操作があります。それぞれの名前はopendir(), readdir(), closedir()です。順番に見ていきましょう。





## opendir(3)

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

DIR *opendir(const char *path);
```

opendir()は、パスpathにあるディレクトリを読み込みのために開きます。戻り値はDIRという型へのポインタで、これは構造体ストリームを管理するための構造体です。DIR型の意味は、FILE型と対比して考えるとイメージが湧きやすいでしょう。

## readdir(3)

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

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

readdir()は、ディレクトリストリームdからエントリを1つ読み込みます。読み込んだエントリをstruct dirent（DIRectory ENTry）で返します。エントリがなくなるか、または読み込みに失敗するとNULLを返します。

struct direntの内容はOSによって違いますが、Linuxには少なくともエントリの名前を表す「char *d_name」が存在します。d_nameは普通のCの文字列（'\0'で終端されている）なので、そのままprintf()やfputs()に渡せます。なお、戻り値のポインタが指すstruct direntは、次にreaddir()を呼んだ時点で上書きされるので注意してください。つまり、readdir()で得たstruct direntをどこかに保存しておいてあとで使うようなことはできません。

## closedir(3)

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

int closedir(DIR *d);
```

closedir()は、ディレクトリストリームdを閉じる関数です。成功したら0を返し、失敗したら-1を返します。

また、この他にもfseek()とftell()に対応するseekdir()とtelldir()もありますが、使用頻度が低いので説明は省略します。必要になったときにmanページを見てください。

## lsコマンドを作る

ディレクトリを読む例として、エントリの名前（つまりファイル名）を表示するだけの簡単なlsコマンドを作ってみましょう（リスト10.1）。

リスト 10.1　ls.c

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

static void do_ls(char *path);

int
main(int argc, char *argv[])
{
    int i;

    if (argc < 2) {
        fprintf(stderr, "%s: no arguments\n", argv[0]);
        exit(1);
    }
    for (i = 1; i < argc; i++) {
        do_ls(argv[i]);
    }
    exit(0);
}

static void
do_ls(char *path)
{
    DIR *d;
    struct dirent *ent;

    d = opendir(path);              /* ① */
    if (!d) {
        perror(path);
        exit(1);
    }
    while (ent = readdir(d)) {  /* ② */
        printf("%s\n", ent->d_name);
    }
    closedir(d);                    /* ①' */
}
```



main()の内容はもう説明するまでもないと思いますので、do_ls()だけを解説しましょう。

① まず、パスpathにあるディレクトリをopendir()で開きます。ここでpathが存在しなかったり、ディレクトリでなかったりするとNULLが返ってくるので、チェックしてexit()します。今回はperror()が必要なのは1ヶ所だけなのでdie()は定義しませんでした。

①'opendir()したら必ず対応するclosedir()が必要です。

② エントリがなくなるまで（つまり、NULLが返ってくるまで）readdir()でエントリを読んで名前を出力します。

## lsコマンドの実行例

作ったばかりの簡易lsコマンドを使ってみましょう。システムのlsコマンドと違って引数が必須なので注意してください。

```
$ ./ls .
..
ls.c
.
ls
```

「.」と「..」の意味はご存知だと思います。「.」はそのディレクトリ自身を表し、「..」は1つ上のディレクトリを表します。

システムのlsコマンドは-aオプションを付けない限り「.」で始まるエントリ（ドットファイル）を表示しないので、普通は「.」と「..」も表示されません。しかし、それはあくまでlsコマンドがやっていることです。readdir()にとってはドットファイルだからといっても変わるところはありません。

また、readdir()で読み込んだエントリは名前順にソートされているとは限らないので、上記のリストではバラバラの順序で表示されています。システムのlsコマンドの出力がいつも名前順になっているのは、lsコマンドがファイル名をソートしているからです。システムのlsコマンドでも-Uオプションを付けるとソートせずに出力するようになるので、試してみるとよいでしょう。

## ディレクトリツリーのトラバース

ここまではディレクトリ中のファイルにアクセスする方法を説明してきました。しかし、ディレクトリの中にまたディレクトリがあって、その中までたどっていきたい場合、つまり再帰的にたどりたい場合もあると思います。このような操作を「ディレクトリツリーの**トラバース**（traverse）」と言います。

基本的には、トラバースの場合もopendir()，readdir()，closedir()の3つを使って地道にディレクトリをたどっていけば実装できます。ただし、トラバースのときには少し注意してほしい点がいくつかあります。



まず、先ほど見た「.」と「..」の存在です。よくある誤りは、これをすっかり忘れて次のようなプログラムを書いてしまうことです。

```
void
traverse(path)
{
    DIR *d = opendir(path);
    struct dirent *ent;

    while (ent = readdir(d)) {
        if (entがディレクトリ) {
            traverse(path/ent);
        }
        何かする
    }
}
```

どこが悪いのかわかるでしょうか。まず、この関数は「.」を排除していないので、「dir」「dir/.」「dir/./.」「dir/./././.」「dir/./././.」…を順番にたどってしまい、無限再帰に陥ってしまいます。また「..」を排除しないと結局ルートディレクトリまでさかのぼることになり、ファイルシステム全体を扱うことになってしまいます。この問題を避けるには、明示的に「.」と「..」を除外しなければいけません。

2つ目はシンボリックリンクです。例えば処理中のディレクトリの中にルートディレクトリを指すシンボリックリンクがあったらどうなるでしょうか。何も対処をしないとルートディレクトリに処理が移ってしまい、その中にはまたルートディレクトリを指すシンボリックリンクがありますから、無限ループに入ってしまいます。この問題を避けるには、後述するlstat()を使ってシンボリックリンクを明示的に除外する必要があります。

このように、トラバースはなかなか面倒な処理です。GNU libcにはディレクトリツリーをトラバースするためのfts(3)という専用APIがありますが、これは現在でも可搬性の低い部類に入るAPIなので、できれば避けてください。また、地道にディレクトリツリーをトラバースするサンプルコードを、本書のサポートサイトで配布しています。こちらも参考にしてください。