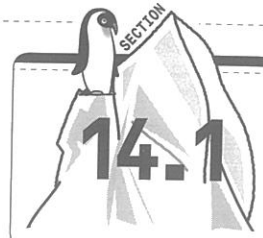


14.1 カレントディレクトリ

まず最初はディレクトリについて話します。

getcwd(3)

プロセスには「今いるディレクトリ」という属性が用意されています。これをプロセスの**カレントワーキングディレクトリ**（current working directory）または単にカレントディレクトリと言います。そのカレントディレクトリのパスを得る関数が、getcwd()です。cwdがcurrent working directoryの略なのはすぐわかるでしょう。

```
#include <unistd.h>

char *getcwd(char *buf, size_t bufsize);
```

getcwd()は、自プロセスのカレントディレクトリをbufに書き込みます。成功したらbufを返し、失敗したらNULLを返してerrnoをセットします。特に、パスがbufsizeバイト以上になるときはエラーERANGEを返すことに注意してください。

なお、getcwd()の他にgetwd()という関数もありますが、こちらはバッファオーバーフローの危険があるので絶対に使ってはいけません。

パスのためのバッファを確保する

ところで、getcwd()のbufのサイズはどれくらい取ればよいのでしょうか。これは一般に「パスを格納するバッファはどれくらい用意すればよいか」という問題だと考えることができます。この問題は第10章で話したreadlink()でも存在していました。その答えをここでまとめて解説しておきます。

まず、間違っている方法は、limits.hをインクルードしてPATH_MAXの値を使うことです（リスト14.1）。この方法はバッファの扱いが楽なため、かつてはよく使われていました。

リスト14.1 パスを書き込むバッファを確保する（誤）

```
#include <unistd.h>
#include <limits.h>

{
    char buf[PATH_MAX];

    if (!getcwd(buf, sizeof buf)) {
        /* エラー */
    }
}
```

この方法の問題は、PATH_MAXでは足りない場合があることです。昔はPATH_MAXの値が固定でしたが、現在はカーネル動作中に値を変更できるようになっているので、ビルド時に定数を見るだけでは不十分です。システムコールpathconf()を使うとこの点は改善できますが、本書では説明しません。

正しい方法は、malloc()を使ってバッファを確保し、とりあえず試してみる方法です（リスト14.2）。バッファの長さが足りなかったらrealloc()でバッファを増やし、再度試します。なお、こちらはmalloc()を使いますから、使い終わったらfree()する必要があります。

リスト14.2 パスを書き込むバッファを確保する（正）

```
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

#define INIT_BUFSIZE 1024

char*
my_getcwd(void)
{
```

```
    char *buf, *tmp;
    size_t size = INIT_BUFSIZE;

    buf = malloc(size);
    if (!buf) return NULL;
    for (;;) {
        errno = 0;
        if (getcwd(buf, size))
            return buf;
        if (errno != ERANGE) break;
        size *= 2;
        tmp = realloc(buf, size);
        if (!tmp) break;
        buf = tmp;
    }
    free(buf);
    return NULL;
}
```

chdir(2)

自プロセスのカレントディレクトリを変更するには、chdir()を使います。chdirはchange working directoryの略です。

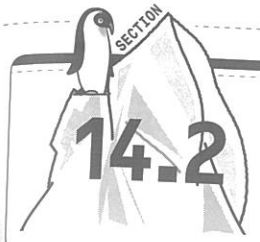
```
#include <unistd.h>

int chdir(const char *path);
```

chdir()は、自プロセスのカレントディレクトリをディレクトリpathに変更します。成功したら0を返します。失敗したら-1を返してerrnoをセットします。

他のプロセスのカレントディレクトリ

APIで変更できるのは自プロセスのカレントディレクトリだけです。他のプロセスのカレントディレクトリを変更する方法はありません。知るだけでよければ、シンボリックリンク「/proc/プロセスID/cwd」が使えます。



14.2 環境変数

環境変数（environment variable）とは、プロセスの親子関係を通じて伝播するグローバル変数のようなものです。普段からシェルでPATHやEDITORといった変数を操作していると思いますが、そのPATHやEDITORが環境変数です。一般的には、常に設定しておきたいユーザ独自の設定などをプログラムに伝えるために使われます。例えば環境変数LESSに"-i"を設定しておくと、lessは常に-iオプションを付けているかのように動作します。同様の働きをする環境変数には、MORE，GZIPなどがあります。プログラムが意識する環境変数はたいてい各コマンドのmanページに記載されていますから、確認してみてください。

最後に、特に一般的で重要な環境変数を表14.1に挙げておきます。

表 14.1　重要な環境変数

名前	意味
PATH	コマンドの存在するディレクトリ
TERM	使っている端末の種類。linux，kterm，vt100など
LANG	ユーザのデフォルトロケール。日本語ならja_JP.UTF-8やja_JP.EUC-JP
LOGNAME	ユーザのログイン名
TEMP	一時ファイルを置くディレクトリ。/tmpなど
PAGER	manなどで起動するテキスト閲覧プログラム。lessなど
EDITOR	デフォルトエディタ。viやemacsなど
MANPATH	manのソースを置いているディレクトリ
DISPLAY	X Window Systemのデフォルトディスプレイ

environ

環境変数にはグローバル変数environを介してアクセスできます。型はchar**ですから、図示すると図14.1のようになるでしょう。

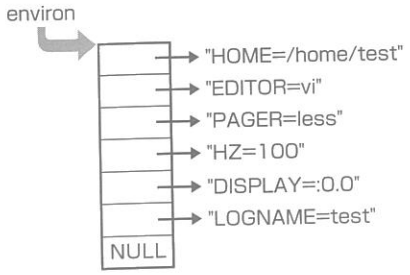


図14.1　environの構造

例えば、自プロセスの環境変数をすべて表示するにはリスト14.3のように書きます。

リスト14.3　env.c

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

extern char **environ;

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

    for (p = environ; *p; p++) {
        printf("%s\n", *p);
    }
    exit(0);
}
```

environはどのヘッダファイルでも宣言されていないので、environに直接アクセスする場合はこのように自分でextern宣言しておく必要があります。

なお、environの指す先は後述するputenv()で移動することがありますから、変数に保存しておいてあとで使ったりしてはいけません。

getenv(3)

```
#include <stdlib.h>

char *getenv(const char *name);
```

getenv()は、環境変数nameの値を検索して返します。環境変数nameが見つからなければNULLを返します。

getenv()が返す文字列もputenv()などの処理で移動する場合があるので、値を使い回してはいけません。また戻り値の文字列に書き込んでもいけません。

putenv(3)

```
#include <stdlib.h>

int putenv(char *string);
```

putenv()は、環境変数の値をセットします。引数のstringは「名前＝値」の形式でなければいけません。形式が間違っている場合に何が起きるのかは不定です。なお、putenv()は渡したstringをそのまま使い続けるので、stringの領域は静的に確保しておくかmalloc()で割り当てる必要があります。

成功したときは0を返します。失敗したときは-1を返してerrnoをセットします。

なお、環境変数の設定には他にsetenv()、unsetenv()、clearenv()というAPIも使えます。