CHAPTER 7

UNIXプロセスの環境

7.1 はじめに

次章でプロセス制御の基本操作を解説する前に、単一プロセスの環境を調べておく必要がある。プログラムを実行したときどのようにmain関数が呼ばれるのか、新たなプログラムにどのようにコマンド行引数が渡されるのか、典型的なメモリ配置はどのようになるのか、メモリを追加割り付けする方法、プロセスが環境変数をどのように使用するのか、プロセスを終了するさまざまな方法について調べる。さらに、longjmpとsetjmp関数とそれらのスタックとの相互作用も調べる。そして最後に、プロセスのリソースリミット(資源利用上の制限)について調べる。

7.2 main関数

Cのプログラムは、main関数を呼び出して実行が始まる。main関数のプロトタイプはつぎのようになる。

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

*argc*はコマンド行引数の個数であり、*argv*は引数を指すポインタの配列である。これらについては7.4節で述べる。

(8.9節で述べるexec関数の1つによって)カーネルがCプログラムを起動すると、main関数を呼ぶ前に特別な起動ルーティンが呼ばれる。実行可能なプログラムファイルでは、プログラムの開始アドレスとしてこの起動ルーティンを指定する。つまり、ccなどのCコンパイラが起動した

リンケージエディタが設定する。起動ルーティンはカーネルから値(コマンド行引数と環境変数)を受け取り、前に示したように main 関数が呼ばれるように設定する。

7.3 プロセスの終了

プロセスを終了する方法は5つある。

1. 正常終了:
   (a) main から戻る。
   (b) exit を呼ぶ。
   (c) _exit を呼ぶ。
2. 異常終了:
   (a) abort (第10章参照) を呼ぶ。
   (b) シグナル (第10章参照) で終了させられる。

前節で述べた起動ルーティンは、main 関数から戻ると exit 関数を呼ぶように書かれている。起動ルーティンをCで書いたとすると(しばしばアセンブラで書いてあるが)、main の呼び出しはつぎのようになる。

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

● exit と _exit 関数 ●

この2つの関数はプログラムを正常に終了する。_exit は直接カーネルに戻り、exit はさまざまな後始末を行ってからカーネルに戻る。

```
#include <stdlib.h>

void exit(int status);

#include <unistd.h>

void _exit(int status);
```

8.5節では、これら2つの関数が終了プロセスの親と子のような別のプロセスに与える影響について議論する。

> ヘッダーが異なるのは、exit は ANSI C で規定されているが、_exit は POSIX.1 で規定されているからである。

歴史的に、exit 関数は標準入出力ライブラリの後始末を常に行う。つまり、オープンしているすべてのストリームに対して fclose 関数を呼ぶのである。5.5節では、これによりバッファされた出力データがフラッシュされる(ファイルへ書き出される)ことを述べた。

exit と _exit のいずれの関数も、終了状態 (exit status) と呼ばれる1つの整数引数を取る。UNIX のほとんどのシェルには、プロセスの終了状態を調べる方法がある。もし、(a) 終了状態を与えずにこれらの関数を呼んだ場合、(b) 値を指定せずに main で return を実行した場合、(c) main 関数の最後に達した(暗黙の return)場合、プロセスの終了状態は未定義である。したがって、古典的なつぎの例は不完全である。

```
#include    <stdio.h>

main()
{
    printf("hello, world\n");
}
```

なぜなら、main 関数の最後に達するため、値(終了状態)を指定せずにCの起動ルーティンに戻るからである。

```
    return(0);
```

あるいは

```
    exit(0);
```

を追加すれば、このプログラムを起動したプロセス(シェル)に終了状態0を与えられる。さらに、main の宣言は

```
int main(void)
```

とすべきである。次章では、プロセスからプログラムを実行し、そのプロセスの終了を待ち、終了状態を取り出す方法を見ていく。

> main は整数を返すと宣言しておいて、(return の代わりに) exit を使うとコンパイラや UNIX の lint(1) プログラムから不必要な警告を得ることがある。これは、コンパイラには main の exit が return と同じであることが分からないからである。警告メッセージはつぎのようなものである。"control reaches end of nonvoid function." (しだいに厄介に思えてくる) これらの警告を抑えるには、main では exit の代わりに return を使うことである。しかし、このようにするとプログラムで exit を呼び出している場所を UNIX の grep ユーティリティで探せなくなる。別の解決方法は、main は int を返すのではなく void を返すと宣言し、exit を使い続けることである。これでコンパイラの警告は抑えられるが、(特にプログラムの作法としては) 正しくない。ANSI C と POSIX.1 のいずれでも定義されているとおり、本書では main は整数を返すとし、コンパイラの警告には我慢することにする。

● atexit 関数 ●

ANSI C では、exit が自動的に呼び出す最大32個の関数をプロセスに登録できる。これらを終了ハンドラ (exit handlers) と呼び、atexit 関数を呼び出して登録する。

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

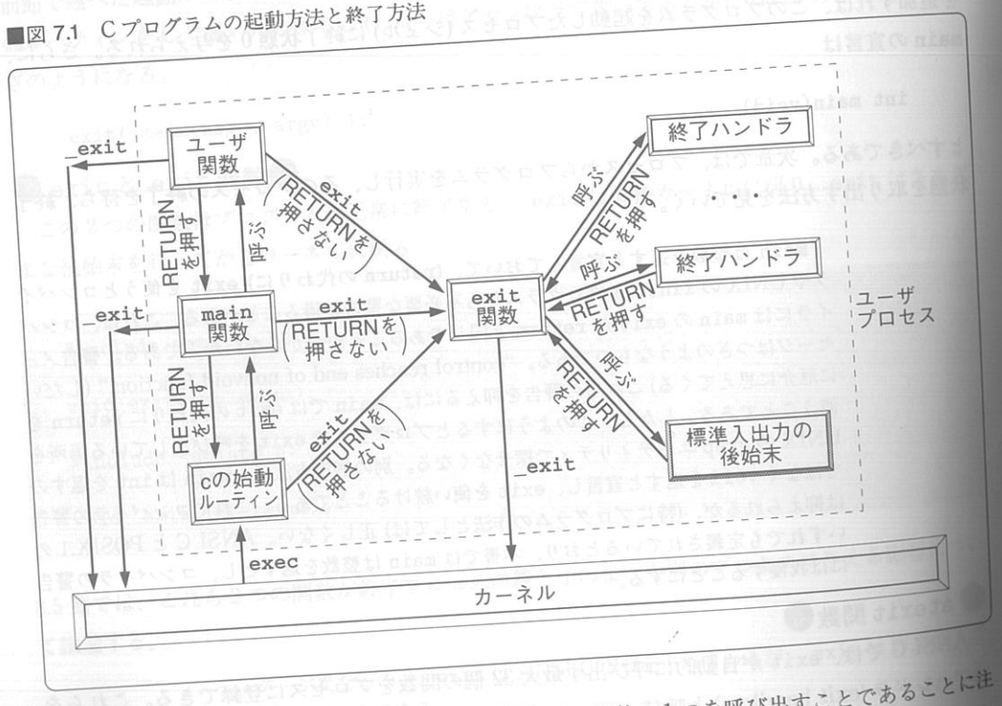
戻り値: 成功ならば0、エラーならば非ゼロ

この宣言から、atexit の引数には関数のアドレスを渡すことが分かる。登録された関数は引数なしで呼び出され、しかも値を返さない。exit 関数は、登録順と逆順にこれらの関数を呼び出す。各関数は登録された回数だけ呼び出される。

これらの終了ハンドラは ANSI C で追加され、SVR4 と 4.3+BSD のどちらでも使える。システム V や 4.3BSD の初期のバージョンでは、これらの終了ハンドラは使えない。

ANSI C と POSIX.1 では、exit はまず終了ハンドラを呼び出し、続いてオープンしているすべてのストリームを fclose でクローズする。図 7.1 に C プログラムの起動方法とさまざまな終了方法をまとめた。

■図 7.1　C プログラムの起動方法と終了方法



カーネルがプログラムを実行する唯一の方法は、exec 関数の 1 つを呼び出すことであることに注意してほしい。プロセスが自発的に終了する唯一の方法は、明示的に (exit を呼ぶことで) あるいは暗黙のうちに _exit を呼ぶことである。また、プロセスは (図 7.1 には示していないが) シグナルで終了を強要される。

●プログラム例●

プログラム 7.1 に atexit 関数の使い方を示す。

▼プログラム 7.1　終了ハンドラの例

```
#include    "ourhdr.h"

static void my_exit1(void), my_exit2(void);

int
main(void)
{
    if (atexit(my_exit2) != 0)
        err_sys("can't register my_exit2");

    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");
    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");

    printf("main is done\n");
    return(0);
}

static void
my_exit1(void)
{
    printf("first exit handler\n");
}

static void
my_exit2(void)
{
    printf("second exit handler\n");
}
```

プログラム 7.1 を実行するとつぎのようになる。

```
$ a.out
main is done
first exit handler
first exit handler
second exit handler
```

main では exit を呼ばずに return を実行したことに注意してほしい。□

7.4 コマンド行引数

プログラムを実行するとき、exec を呼び出すプロセスは新しいプログラムにコマンド行引数を渡せる。これは UNIX シェルの普通の動作である。これまでの章で既に多くの例を見てきた。

●プログラム例●

プログラム 7.2 は、コマンド行のすべての引数を標準出力へエコーする。(UNIX の標準の echo(1)

プログラムは0番目の引数を出力しない。)

▼プログラム7.2　コマンド行のすべての引数を標準出力へエコーする

```
#include    "ourhdr.h"

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

    for (i = 0; i < argc; i++)      /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);
    exit(0);
}
```

このプログラムをコンパイルし、実行ファイルをechoargと名付けると、つぎのようになる。

```
$ ./echoarg arg1 TEST foo
argv[0]: ./echoarg
argv[1]: arg1
argv[2]: TEST
argv[3]: foo
```

ANSI CとPOSIX.1のいずれでも、argv[argc]はnullポインタであることが保証される。よって、引数処理の繰り返しの部分をつぎのように書いてもよい。

```
for (i = 0; argv[i] != NULL; i++)
```

□

7.5 環境リスト

各プログラムには環境リストも渡される。引数リストと同様に、環境リストは文字ポインタの配列であり、各ポインタはnullで終端されたCの文字列のアドレスである。ポインタを収めた配列のアドレスは、大域変数environに入っている。

```
extern char **environ;
```

例えば、環境に5つの文字列があるとすると図7.2のようになる。ここでは各文字列を終えるnullバイトを明示した。environを環境ポインタ、ポインタの配列を環境リスト、それらが指す文字列を環境文字列と呼ぶ。

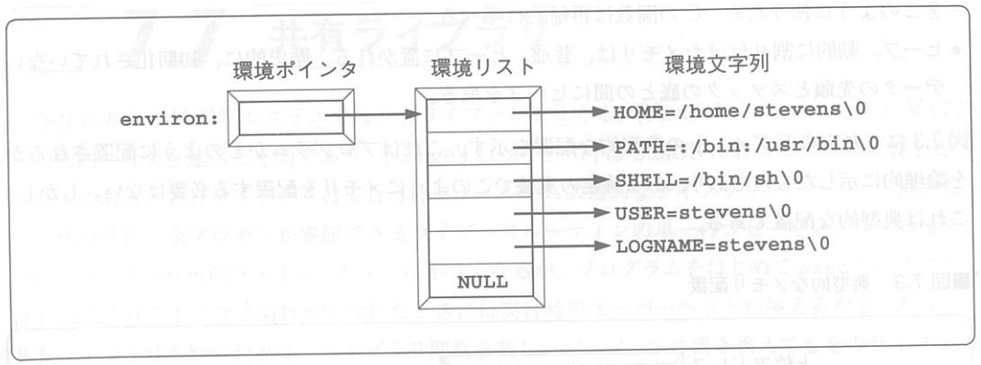
図7.2に示すように、環境は便宜的につぎの形式の文字列である。

name=value

あらかじめ定義されたほとんどの名前はすべて大文字であるが、これも単に便宜上のことである。

歴史的に、ほとんどのUNIXシステムではmain関数の第3引数に環境リストのアドレスを渡す。

■図7.2　5つのC文字列から成る環境




```
int main(int argc, char *argv[], char *envp[]);
```

ANSI Cではmain関数の引数は2つであると規定しており、大域変数environがあるため3番目の引数を使う利点がないが、POSIX.1では(可能なかぎり)第3引数の代わりにenvironを使うように規定している。特定の環境変数を参照するには、environ変数の代わりにgetenvとputenv関数(7.9節で述べる)を使う。しかし、環境全体を調べるにはenvironポインタを使う必要がある。

7.6 Cプログラムのメモリ配置

歴史的に、Cプログラムはつぎの要素から構成されている。

- テキストセグメント。これは、CPUが実行する機械語命令群である。(テキストエディタ、Cコンパイラ、シェルなど)頻繁に使用するプログラムのコピーがメモリに1つあるだけで済むように、通常テキストセグメントは共有される。また、プログラムが命令を誤って書き換えないように、テキストセグメントはしばしば読み取り専用である。
- 初期化されたデータセグメント。普通は単にデータセグメントと呼ばれ、プログラム中の初期化された変数を保持する。例えば、関数の外側にあるつぎのようなCの宣言

```
int   maxcount = 99;
```

では、この変数を初期値とともに初期化されたデータセグメントに置く。
- 初期化されていないデータセグメント。このセグメントはしばしば「bssセグメント」とも呼ばれる。この名称は、昔のアセンブラの“block started by symbol”に由来する。このセグメントのデータは、プログラムの開始前にカーネルによって0に初期化される。関数の外側にあるつぎのようなCの宣言

```
long   sum[1000];
```

では、この変数を初期化されないデータセグメントに置く。
- スタック。関数呼び出しごとの情報とともに、自動変数を保持する場所である。関数が呼び出される度に、戻りアドレス、呼び出し側の(マシンのレジスタなどの)環境についての情報をスタッ