Cプログラムの典型的なメモリ配置を調べ、プロセスでどのようにメモリを動的に取得・解放するかを調べた。環境を操作する関数ではメモリ割り付けを起動する必要があるため、環境操作用関数を詳しく調べることは意義深かった。プロセス内で大域的な分岐を行う関数、setjmp と longjmp を示した。そして、SVR4 と 4.3+BSD で使用できるリソースリミットの説明で本章を終えた。

7.13 演習問題

7.1 80386 上の SVR4 と 4.3+BSD で、"hellow, world"を出力して exit や return を呼ばないプログラムを実行すると、プログラムの終了状態は(シェルで調べると) 13 になる。なぜか？

7.2 プログラム 7.1 が printf の出力を実際に出力するのはいつか？

7.3 (a) main から argc と argv を引数として関数に渡す、(b) main で argc と argv を大域変数へコピーする、ということをせずに、main から呼ばれた関数がコマンド行引数を調べる方法はあるか？

7.4 プログラムを実行するときに、データセグメントの 0 番地を参照できないようにする UNIX システムもある。なぜか？

7.5 終了ハンドラ用に新しいデータ型 Exitfunc を C の typdef を用いて定義せよ。このデータ型を用いて atexit のプロトタイプを書き改めよ。

7.6 calloc を用いて long の配列を割り付けた場合、配列は 0 に初期化されるか？ calloc を用いてポインタの配列を割り付けた場合、配列は null ポインタで初期化されるか？

7.7 7.6 節の終わりの size コマンドの出力で、ヒープやスタックに関するサイズがないのはなぜか？

7.8 7.7 節で 2 つのファイルサイズ (104859 と 24576) は、それぞれのテキストサイズとデータイズの和に等しくない。なぜか？

7.9 7.7 節において、単純なプログラムであるにもかかわらず、共有ライブラリを使用するしで実行可能ファイルのサイズがなぜ違うのか？

7.10 7.10 節の終わりで、関数が自動変数へのポインタを返せない理由を示した。つぎのコード正しいか？

```
int
f1(int val)
{
    int     *ptr;

    if (val == 0) {
        int     val;

        val = 5;
        ptr = &val;
    }
    return(*ptr + 1);
}
```


CHAPTER

8 プロセス制御

8.1 はじめに

本章では、UNIX が提供するプロセス制御について述べる。つまり、新規プロセスの作成、プログラムの実行、プロセスの終了のことである。さらに、プロセスに関連したさまざまな ID、つまり実/実効/保存ユーザ ID とグループ ID、および、これらがプロセス制御操作にどのように影響するかを調べる。解釈実行ファイルと system 関数についても述べる。そして、ほとんどの UNIX システムで使えるプロセスの実行記録について述べて本章を終える。プロセス制御関数についてさまざまな角度から調べることになるだろう。

8.2 プロセス識別子

プロセスには、非負整数の一意なプロセス ID がある。プロセス ID はプロセスを一意に識別する識別子であるため、一意性を保証するために他の識別子の一部としても利用される。5.13 節の tmpnam 関数は、プロセス ID を名称に含めることで一意なパス名を作成する。

プロセスがいくつかある。プロセス ID 0 は、通常、スケジュールプロセスであり、スワッパと呼ばれる。このプロセスに対応するプログラムはディスク上にはない。つまり、これはシステムの一部であり、システムプロセスと呼ばれる。プロセス ID 1 は、通常、init プロセスであり、ブートストラップを完了するとカーネルが起動する。このプロセスのプログラムファイルは、システムの古いバージョンでは /etc/init であるが、新しいバージョンでは /sbin/init である。init プロセスは、カーネルがブートストラップを完了した後に、UNIX システムを立ち

上げる責任を持つ。init は、通常、システムに依存した初期化ファイル (/etc/rc* ファイル) を読み込み、システムを (マルチユーザなどの) 特定の状態にする。init プロセスは決して終了しない。これは普通のユーザプロセスであり (スワッパのようにカーネル内部にあるシステムプロセスではないが)、スーパーユーザ特権で動く。本章の後ほどに、init がどのように親のない子 (オーファンド) プロセスの親になるかを述べる。

仮想記憶を実装した UNIX では、プロセス ID 2 がページデーモン (pagedaemon) であるものもある。このプロセスは仮想記憶システムのページングに責任を持つ。スワッパと同様に、ページデーモンもカーネルプロセスである。

プロセス ID に加えて、各プロセスにはその他の識別子もある。つぎの関数はこれらの識別子を返す。

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

pid_t getpid(void);                  戻り値: 呼び出したプロセスのプロセス ID

pid_t getppid(void);                 戻り値: 呼び出したプロセスの親のプロセス ID

uid_t getuid(void);                  戻り値: 呼び出したプロセスの実ユーザ ID

uid_t geteuid(void);                 戻り値: 呼び出したプロセスの実効ユーザ ID

gid_t getgid(void);                  戻り値: 呼び出したプロセスの実グループ ID

gid_t getegid(void);                 戻り値: 呼び出したプロセスの実効グループ ID
```

これらの関数はエラーで戻らないことに注意してほしい。次節で fork 関数を述べるときに親のプロセス ID について述べる。実および実効ユーザ ID とグループ ID については 4.4 節で述べた。

8.3 fork 関数

UNIX カーネルで新規にプロセスを作成する「唯一」の方法は、既存のプロセスが fork 関数を呼ぶことである。(これは、スワッパ、init、ページデーモンなどの前節に述べた特別なプロセスには適用されない。これらのプロセスはブートストラップの過程でカーネルが特別に作成する。)

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

pid_t fork(void);                    戻り値: 子には 0、親には子のプロセス ID、エラーなら
```

fork を呼ぶことで作成された新規プロセスを、子プロセス (child process) と呼ぶ。この

び出しは 1 回であるが、2 度戻る。戻るときの唯一の違いは、子には値 0 を戻し、親には新しい子のプロセス ID を値として戻すことである。子のプロセス ID を親に戻す理由は、プロセスは複数の子を持つことができ、子のプロセス ID を取得するための関数がないからである。fork が子に 0 を戻すのは、プロセスには 1 つの親しかなく、したがって、親のプロセス ID は getppid を呼ぶことで取得できるからである。(プロセス ID 0 は、常にスワッパが使用するため、子のプロセス ID が 0 になることはあり得ない。)

親と子のどちらも、fork を呼んだつぎの命令から実行を続ける。子は親の複製である。例えば、子は親のデータ空間、ヒープ、スタックのコピーを得る。これは子にコピーされたものであり、親と子が同じメモリ領域を共有するのではない。テキストセグメント (7.6 節参照) が読み取り専用であれば、親と子がテキストセグメントを共有することもしばしばある。

fork に続けてしばしば exec が呼ばれるため、現在のほとんどの実装では、親のデータ、スタック、ヒープを完全にコピーすることはない。代わりに、コピーオンライト (copy-on-write、COW) と呼ばれる技法を用いる。親と子がこれらの領域を共有し、カーネルは領域のプロテクションを読み取り専用に変更する。どちらかのプロセスがこれらの領域を修正しようとすると、カーネルはその部分のメモリ、仮想記憶システムでは典型的には「ページ」だけを複製する。[Bach 1986] の 9.2 節、[Leffler et al. 1989] の 5.7 節がこの機能に詳しい。

●プログラム例●

プログラム 8.1 は fork 関数の使用例である。

▼プログラム 8.1 fork 関数の例

```
#include    <sys/types.h>
#include    "ourhdr.h"

int     glob = 6;       /* external variable in initialized data */
char    buf[] = "a write to stdout\n";

int
main(void)
{
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("write error");
    printf("before fork\n");    /* we don't flush stdout */

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {        /* child */
        glob++;                 /* modify variables */
        var++;
    } else
        sleep(2);               /* parent */
```

```
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

このプログラムを実行するとつぎのようになる。

```
$ a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89    子の変数は変更されている
pid = 429, glob = 6, var = 88    親側のコピーは変更されていない
$ a.out > temp.out
$ cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89
before fork
pid = 431, glob = 6, var = 88
```

一般には、親の前に子の実行が始まるのか、その逆になるのか、分からない。これはカーネルのスケジューリングアルゴリズムに依存する。親と子が互いに同期する必要がある場合には、ある種のプロセス間通信が必要になる。プログラム 8.1 では、親側では 2 秒待ち、子には実行させた。これが十分である保証はない。レースコンディションに関して 8.8 節で述べるときに、この種の同期やその他の種類の同期について述べる。10.16 節では、fork の後でシグナルを用いた親と子の同期方法を示す。

プログラム 8.1 における入出力関数と fork の相互作用に注意してほしい。第 3 章では write 関数はバッファリングしないことを述べた。fork の前に write を呼ぶため、そのデータは標準出力に一度書かれる。しかし、標準入出力ライブラリはバッファリングする。5.12 節では、標準出力が端末装置につながっていると行バッファされ、それ以外のときは完全にバッファされることを述べた。プログラムを対話的に実行すれば、改行文字で標準出力バッファがフラッシュされるため、printf の出力行は 1 つだけである。しかし、標準出力をファイルにリダイレクトすると printf の出力を 2 つ得る。後者の場合、fork の前に printf を 1 回呼ぶだけであるが、fork を呼んだ時点では行はバッファのなかにある。親のデータ空間を子にコピーするときに、このバッファも子にコピーされる。親と子のいずれの標準入出力バッファにも行が入っているのである。exit の直の 2 つ目の printf は、バッファにデータを追加するだけである。各プロセスが終了すると、バッファの内容がフラッシュされる。□

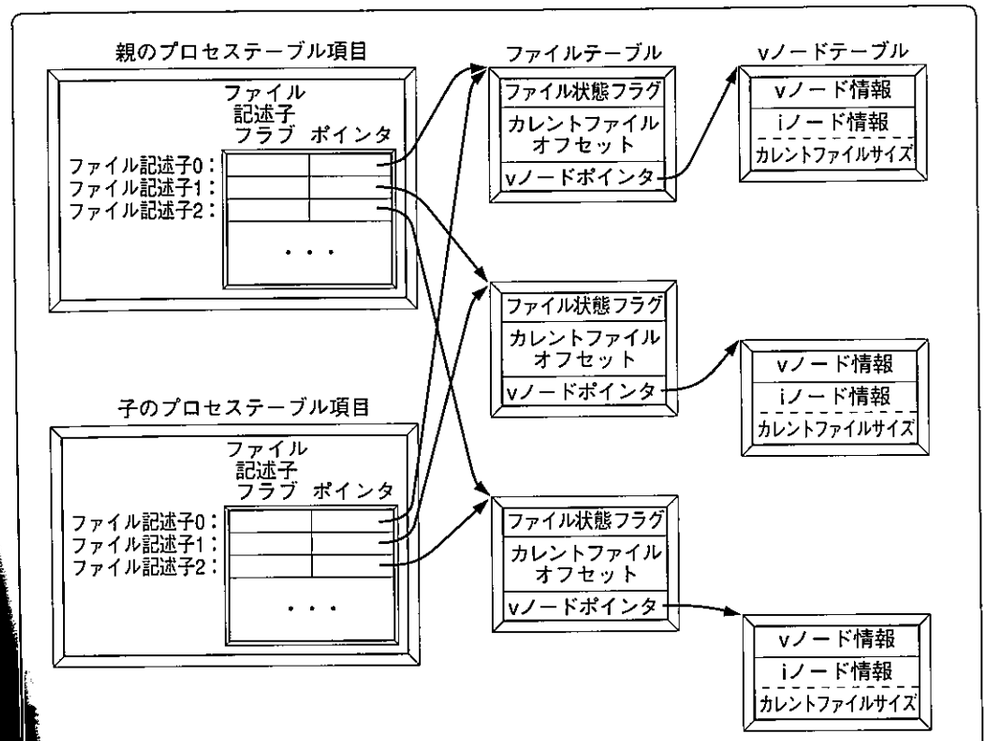
●ファイルの共有●

プログラム 8.1 で注意すべきもう 1 つの点は、親の標準出力をリダイレクトすると、子の出力もリダイレクトされることである。fork の特性の 1 つは、親がオープンしているすべての記述子を子に複製することである。各記述子ごとに dup 関数が実質的に呼ばれるため、「複製された (duplicated)」というのである。親と子は、オープンされた各記述子に対するファイルテーブル項目を共有する (図 3.4 を思い出してほしい)。

標準入力、標準出力、標準エラー出力に対して異なる 3 つのファイルをオープンしたプロ

考える。fork から戻ると図 8.1 に示したようになる。

■図 8.1　fork 後の親と子のオープンしたファイルの共有



子が同じファイルオフセットを共有することが重要である。子をフォーク (fork) してからその了を待つ (wait) プロセスを考える。いずれのプロセスも通常の処理過程で標準出力へ書き出すとする。(シェルなどにより) 親が標準出力をリダイレクトしている場合、子が標準出力へ書くと、親のファイルオフセットを子が更新する。この場合、親が子の終了を待っている (wait) 子は標準出力へ書き出せる。親は子の終了後、子が書き出したものに追加されることを承知で準出力へ書き出せる。親と子が同じファイルオフセットを共有していなければ、この種の相実現することは難しく、親が明示的に動作する必要がある。

を待つ (wait) などの) 何らかの同期を取らずに親と子が同じ記述子に書き出すと、(fork 記述子がオープンされていると仮定すると) 両者の出力は混じり合ってしまう。(プログラたように) こうすることも可能であるが、普通はそのように動作させない。

で記述子を扱う場合が 2 つある。

了を待つ。この場合、親は自分の記述子に関して何もする必要がない。子が終了した読み書きした共有された記述子のファイルオフセットはそのように更新される。

れぞれ独自に扱う。fork の後、親は不必要な記述子をクローズし、子も同じようにすにしてお互いのオープンされた記述子が干渉しないようにする。ネットワークサー

バではしばしばこのようにする。

オープンされているファイル以外にも、親から子へ継承される属性は数多くある。

- 実ユーザ ID、実グループ ID、実効ユーザ ID、実効グループ ID
- 補足グループ ID
- プロセスグループ ID
- セッション ID
- 制御端末
- セットユーザ ID フラグとセットグループ ID フラグ
- カレント作業ディレクトリ
- ルートディレクトリ
- ファイルモード作成マスク
- シグナルマスクとシグナル処理
- オープンされたファイル記述子のクローズオンイグゼックフラグ
- 環境
- 共有メモリセグメント
- リソースリミット

親と子の違いはつぎのとおりである。

- fork からの戻り値が異なる。
- プロセス ID が異なる。
- 2 つのプロセスは異なる親プロセス ID を持つ。子の親プロセス ID は親であるが、親の親プロセス ID は変わらない。
- 子側の tms_utime、tms_stime、tms_cutime、tms_ustime の値は 0 に設定される。
- 親が設定したファイルロックは、子には継承されない。
- 子に対する保留アラームはクリアされる。
- 子に対する保留シグナルの集合は空になる。

これらの機能の多くについては、まだ説明していない。後の章で説明する予定である。

fork が失敗する原因は、主に、(a) システムに既に数多くのプロセスが存在する (通常、何かが変であることを意味する)、(b) 実ユーザ ID が同じプロセスの数の合計がシステムのリミットを超える、の 2 つである。図 2.7 において、CHILD_MAX は特定の実ユーザ ID あたり同時に動かせるプロセスの最大個数を指定することだったのを思い出してほしい。

fork の使い方は 2 つある。

1. コードの異なる部分を親と子が同時に実行できるように、プロセスが自分自身を複製する場合。これはネットワークサーバで普通に行われる。親はクライアントからのサービス要求を待つ。要求が到着すると、親は fork を呼んで要求を子に処理させる。親はつぎのサービス要求の到着を待つ状態に戻る。
2. プロセスが別のプログラムを実行したい場合。これはシェルで普通に行われる。この場合、

fork から戻った直後に exec を行う (8.9 節で述べる)。

上述の 2 の操作 (fork に exec が続く) を、スポーン (spawn) と呼ばれる 1 つの操作にまとめているオペレーティングシステムもある。exec を伴わない fork の使いみちも多いため、UNIX では 2 つに分けている。さらに、2 つの操作に分けているため、子は、入出力の切り替え、ユーザ ID、シグナル処理などのプロセスごとの属性を fork と exec の間で変更できる。第 14 章では、この例を数多く見る。

8.4 vfork 関数

関数 vfork は fork と同じ呼び出し方で、同じ値を返す。しかし、2 つの関数の意味は異なる。

> vfork は、4BSD の仮想記憶リリースの初期段階で登場した。[Leffler et al. 1989] の 5.7 節には、つぎのように述べられている。「非常に効率的ではあるが、vfork には特異な意味があり、一般にはアーキテクチャ上の欠点と思われる。」
> もちろん、SVR4 と 4.3+BSD のいずれでも vfork を使える。
> vfork を呼ぶ場合には、ヘッダー<vfork.h>を取り込む必要があるシステムもある。

(前節の最後の箇条書きの 2 の場合) 新しいプログラムを起動する (exec) ための新規プロセスを作成するために vfork を使う。プログラム 1.5 のシェルの骨格は、この種のプログラムの例でもある。vfork は fork のように新規プロセスを作成するが、(vfork の直後に、子は exec や exit を呼ぶため) 子はアドレス空間を参照しないので、子に親のアドレス空間を完全にコピーするわけではない。その代わりに、親のアドレス空間において子は exec か exit を実行するまで動き続ける。この最適化により、ページ方式の仮想記憶を実装した UNIX では効率がよい。(前節で述べたように、exec を伴った fork の効率を向上するためにコピーオンライト (copy-on-write) を使う実装もある。)

2 つの関数が異なるもう 1 つの点は、vfork は、子が exec か exit を呼ぶまで、子が実行されることを保証することである。子がどちらかの関数を呼ぶと、親の実行が再開される。(つまり、これら 2 つの関数を呼ぶ前に子が親の動作に依存するような場合には、デッドロックを引き起こす。)

●プログラム例●

fork を vfork に置き換えたプログラム 8.1 を見てほしい。標準出力への write は削除した。また子が exec か exit を呼ぶまでは、親はカーネルによって待ち状態にされることが保証されているので、親が sleep を呼ぶ必要はない。

プログラム 8.2 vfork 関数の例

```
lude    <sys/types.h>
lude    "ourhdr.h"

 glob = 6;       /* external variable in initialized data */


oid)
```

```
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;
    printf("before vfork\n");   /* we don't flush stdio */

    if ( (pid = vfork()) < 0)
        err_sys("vfork error");
    else if (pid == 0) {        /* child */
        glob++;                 /* modify parent's variables */
        var++;
        _exit(0);               /* child terminates */
    }

    /* parent */
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

このプログラムを実行するとつぎのようになる。

```
$ a.out
before vfork
pid = 607, glob = 7, var = 89
```

ここでは、子が変数の値を増やすと、親側の値も変わる。子は親のアドレス空間で実行されるため、これは驚くにあたらない。しかし、この振舞いはfork とは異なる。

プログラム 8.2 では exit の代わりに _exit を呼んでいることに注意してほしい。8.5 節で述べたように、_exit は標準入出力バッファをフラッシュしない。代わりに exit を呼ぶと、出力は異なる。

```
$ a.out
before vfork
```

ここでは、親の出力 printf がなくなってしまう！ これは、子が exit を呼ぶと、標準入出力のすべてのストリームがフラッシュの後クローズされるからである。標準出力もこれに含まれる。子がこれを行ったとしても、子は親のアドレス空間で動作しているため、標準入出力のすべての FILE オブジェクトに対する変更は、親のものを変更する。その後、親が printf を呼んでも、標準出力はクローズされており、printf は −1 を返す。□

[Leffler et al. 1989] の 5.7 節には、fork と vfork の実装に関するその他の情報も載っている。

演習 8.1 と 8.2 は、vfork に関する議論の続きである。

8.5 exit 関数

7.3 節で述べたようにプロセスを正常終了する方法は 3 つあり、異常終了は 2 とおりある。

1. 正常終了:
   (a) main 関数で return を実行する。7.3 節で見たように、これは exit を呼ぶのと同じである。
   (b) exit 関数を呼ぶ。この関数は ANSI C で定義されており、atexit を呼んで登録した終了ハンドラの呼び出しとすべての標準入出力ストリームのクローズを含む。ANSI C では、ファイル記述子、マルチプロセス (親と子)、ジョブ制御を扱わないため、この関数の定義は UNIX システムにおいては不完全である。
   (c) _exit 関数を呼ぶ。この関数は exit から呼ばれ、UNIX に固有な詳細を扱う。_exit は POSIX.1 で規定されている。

   UNIX のほとんどの実装では、exit(3) は標準 C ライブラリの関数であるが、_exit(2) はシステムコールである。

2. 異常終了:
   (a) abort を呼ぶ。これは SIGABRT シグナルを発生するため、つぎの項目の特殊な場合である。
   (b) プロセスが特定のシグナルを受け取ったとき (シグナルについて詳しくは第 10 章で述べる)。シグナルは、(例えば、abort 関数を呼んで) プロセス自身、他のプロセス、カーネルが生成できる。カーネルがシグナルを生成する例としては、プロセスが自身のアドレス空間外のメモリ位置を参照したり、0 で除算をした場合である。

プロセスの終了の仕方によらず、最終的にはカーネル内の同じコードが実行される。カーネルのこのコードは、プロセスがオープンしているすべての記述子をクローズし、使用していたメモリを解放することなどを行う。

上述のいずれの場合においても、終了したプロセスが親プロセスに対してどのように終了したかを通知したい。exit と _exit 関数の場合は、引数として最終状態を渡せばよい。しかし、異常終了の場合、異常終了の原因を表す終了状態は (プロセスではなく) カーネルが生成する。いずれにしても、プロセスの親は終了状態を (次節に述べる) wait か waitpid 関数で取得できる。

(exit や _exit の引数や main からの戻り値である) 最終状態 (exit status) と終了状態 (termination status) を区別していることに注意してほしい。最終的に _exit が呼ばれると、カーネルが状態を終了状態に変換する (図 7.1 を思い出してほしい)。図 8.2 に、親が子の終了状態を調べる方法を示す。子が正常終了すると、親は子の最終状態を取得できる。

fork 関数について述べたとき、fork を呼んだ後には子は親プロセスを持つことは明らかだった。ここでは、終了状態を親に戻すことを述べているが、子が終了する前に親が終了してしまうとどうなるのであろうか？ init プロセスが、親が終了してしまったすべてのプロセスの親プロセスになるというのが答えである。プロセスは init から継承されることを見てきた。プロセスが終了すると、通常、カーネルはすべてのプロセスを調べて、終了したプロセスを親とするプロセスが存在するかどうか調べる。もしあれば、そのようなプロセスの親のプロセス ID を 1 (init のプロセス ID) に変更する。このようにして、各プロセスには必ず親が存在することを保証している。

終了する前に子が終了するとどうなるかについても考慮する必要がある。子が完全に消滅してしまうと、子が終了したかどうかを親が調べようとしたときに子の終了状態を取得できなくなってしまう。カーネルは終了したプロセスに関するある種の情報を保存しておき、終了したプロセスの親が wait や waitpid を呼んだときにその情報を利用できるようにする必要がある。必要最小限

の情報は、プロセスID、プロセスの終了状態、プロセスが使用したCPU時間である。カーネルはプロセスが使用していたメモリを解放でき、オープンしていたファイルもクローズできる。UNIXの用語では、終了してはいるが親がその情報を取得していないプロセスをゾンビ(zombie)と呼ぶ。ps(1)コマンドはゾンビプロセスの状態をZと表示する。長時間動き続けて多くの子プロセスをフォーク(fork)するプログラムを書いた場合、これらのプロセスの終了状態を取得しないでおくと、これらはゾンビになる。

10.7節に述べるように、システムVにはゾンビを避けるための規格にはない方法がある。

考慮すべき最後の点は、initが継承したプロセスが終了するとどうなるかである。これはゾンビになるのであろうか？答えは"no"である。なぜなら、initは子が終了するとwait関数を呼んで終了状態を取得するように書かれているためである。このようにして、initはシステムにゾンビがあふれることを防いでいる。「initの子の1つ」といった場合、initが直接生成した(9.2節で述べるgettyなどの)プロセスか、親が終了してしまったためにinitに引き継がれたプロセスを意味する。

8.6 waitとwaitpid関数

プロセスが正常にあるいは異常な終了をすると、親はカーネルから送られるSIGCHLDシグナルで通知を受ける。子の終了は非同期な事象である(親の実行中のどの時点でも起こり得る)ため、このシグナルはカーネルから親への非同期な通知である。親は、このシグナルを無視するか、シグナルが発生すると起動される関数(シグナルハンドラ)を提供するかを選べる。デフォルトではこのシグナルは無視される。この選択については第10章で述べる。ここでは、waitやwaitpidを呼んだプロセスはつぎのようになることを認識してほしい。

- (すべての子プロセスが実行中であると)ブロックされる。
- (子が終了しており、終了状態が取得されるのを待っていると)直ちに子の終了状態を得る。
- (子プロセスが存在しないと)直ちにエラーで戻る。

SIGCHLDシグナルを受け取った後にプロセスからwaitを呼べば、直ちにwaitから戻ると期待できる。しかし、適当にwaitを呼ぶとブロックする場合がある。

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

pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int options);

        2つの関数の戻り値: 成功ならばプロセスID、0(後述)、あるいはエラーならば-1
```

2つの関数の違いはつぎのとおりである。

- waitは子プロセスが終了するまで呼び出し側をブロックするが、waitpidはブロックを防ぐオプションを有する。
- waitpidは子の終了を待たない。どのプロセスの終了を待つかを制御するさまざまなオプションを持つ。

子が既に終了しておりゾンビになっていると、waitは子の状態を取得して直ちに戻る。さもなければ、子が終了するまで呼び出し側をブロックする。呼び出し側をブロックしていて複数の子がいる場合、waitは子の1つが終了した時点で戻る。関数からはプロセスIDが返されるので、どの子が終了したか常に分かる。

どちらの関数においても、引数*statloc*は整数へのポインタである。この引数がnullポインタでなければ、引数が指す場所に終了したプロセスの終了状態が格納される。終了状態に興味がなければ、この引数にnullポインタを指定すればよい。

伝統的にこれら2つの関数が返す整数型の状態は実装ごとに定義されており、(正常終了の)最終状態を表すビット、(異常終了の)シグナル番号を表すビット、コアファイルの作成を表すビットなどから構成されている。POSIX.1では、<sys/wait.h>で定義されたさまざまなマクロを用いて終了状態を調べるように規定している。プロセスがどのように終了したかを調べるための互いに排他的な3つのマクロがあり、すべてWIFで始まる名前である。これら3つのマクロのどれが真であるかに基づいて、最終状態、シグナル番号などを得るために別のマクロを用いる。これらを図8.2に示す。9.8節でジョブ制御について述べるときに、プロセスがどのように休止するかについて述べる。

■図8.2 waitとwaitpidが返した終了状態を調べるためのマクロ

マクロ	意味
WIFEXITED(*status*)	正常終了した子から返された*status*の場合に真。この場合、WEXITSTATUS(*status*)を実行すれば、子プロセスがexitや_exitに渡した引数の下位8ビットを取得できる。
WIFSIGNALED(*status*)	(受け取り処理をしなかったシグナルを送られて)異常終了した子から返された*status*の場合に真。この場合、WTERMSIG(*status*)を実行すれば、終了原因となったシグナル番号を取得できる。さらに、(POSIX.1では未定義であるが)SVR4と4.3+BSDでは、終了プロセスのコアファイルが作成されていた場合に真となるマクロWCOREDUMP(*status*)を定義している。
WIFSTOPPED(*status*)	現在休止している子から返された*status*の場合に真。この場合、WSTOPSIG(*status*)を実行すれば、子を休止させたシグナル番号を取得できる。

プログラム例

プログラム8.3の関数pr_exitは図8.2に挙げたマクロを使って、終了状態を出力する。この関数は本書のさまざまなプログラムで使っている。

プログラム8.3 最終状態を出力する

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"
```

```
pr_exit(int status)
{
    if (WIFEXITED(status))
        printf("normal termination, exit status = %d\n",
                WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("abnormal termination, signal number = %d%s\n",
                WTERMSIG(status),
#ifdef  WCOREDUMP
                WCOREDUMP(status) ? " (core file generated)" : "");
#else
                "");
#endif
    else if (WIFSTOPPED(status))
        printf("child stopped, signal number = %d\n",
                WSTOPSIG(status));
}
```

この関数では、WCOREDUMP マクロが定義されていれば、これを使うことに注意してほしい。

プログラム 8.4 は pr_exit 関数を呼んで、さまざまな終了状態を出力する。

▼プログラム 8.4　さまざまな最終状態を出力する

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"

int
main(void)
{
    pid_t   pid;
    int     status;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)          /* child */
        exit(7);

    if (wait(&status) != pid)           /* wait for child */
        err_sys("wait error");
    pr_exit(status);                    /* and print its status */

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)          /* child */
        abort();                /* generates SIGABRT */

    if (wait(&status) != pid)           /* wait for child */
        err_sys("wait error");
    pr_exit(status);                    /* and print its status */

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)          /* child */
        status /= 0;            /* divide by 0 generates SIGFPE */

    if (wait(&status) != pid)           /* wait for child */
        err_sys("wait error");
    pr_exit(status);                    /* and print its status */

    exit(0);
}
```

プログラム 8.4 を実行するとつぎのようになる。

```
$ a.out
normal termination, exit status = 7
abnormal termination, signal number = 6 (core file generated)
abnormal termination, signal number = 8 (core file generated)
```

残念ながら、WTERMSIG で取得したシグナル番号をシグナル名称に対応付けるポータブルな方法はない。(1 つの方法は 10.21 節にある。) <signal.h> ヘッダーを調べて、SIGABRT の値は 6、SIGFPE の値は 8 であることを確認する必要がある。□

既に述べたように、複数の子がいる場合、wait は任意の終了した子を返す。(終了を待ちたいプロセス ID を知っていると仮定して) 特定のプロセスの終了を待つにはどうすればよいか？ UNIX の古いバージョンでは、wait を呼んで、戻されたプロセス ID を目的のものと比較する必要があった。終了したプロセスが目的のものでなかった場合には、プロセス ID と終了状態を保存して、再度 wait を呼ぶ必要がある。目的のプロセスが終了するまでこれを繰り返す。続いて別のプロセスの終了を待つ場合には、既に終了したプロセスのリストを見て終了しているかどうか調べ、そのなかになければ再度 wait を呼ぶ。特定のプロセスを待つ関数が必要なのである。POSIX.1 では、waitpid 関数がこの (それ以上の) 機能を与える。

waitpid 関数は、POSIX.1 で新しく追加され、SVR4 と 4.3+BSD で使える。しかし、システム V と 4.3BSD の初期のバージョンにはない。

waitpid に対する *pid* 引数の解釈は、その値に依存する。

- *pid* == −1 の場合、任意の子プロセスを待つ。このような waitpid は wait と等価である。
- *pid* > 0 の場合、プロセス ID が *pid* に等しい子を待つ。
- *pid* == 0 の場合、呼び出し側のプロセスのグループ ID と同じグループ ID を持つ任意の子を待つ。
- *pid* < −1 の場合、*pid* の絶対値に等しいプロセスグループ ID を持つ子を待つ。

(プロセスグループについては 9.4 節で述べる。) waitpid は終了した子プロセスのプロセス ID を返し、その終了状態を *statloc* に格納する。wait では、呼び出し側プロセスには子がないというのが唯一のエラーであった。(その他の起こり得るエラーとしては、関数の呼び出しがシグナルで中断され

る場合である。これについては第10章で述べる。) しかし、waitpidでは、指定したプロセスやプロセスグループが存在しなかったり、呼び出し側のプロセスの子でない場合にエラーとなる可能性がある。

オプションの引数でwaitpidの動作をさらに制御できる。この引数には、0または図8.3に示した定数のビットごとの論理和を指定する。

■図8.3 waitpidに対するオプションの定数

定数	意味
WNOHANG	*pid*で指定した子が終了・休止していなくてもwaitpidをブロックしない。この場合戻り値は0である。
WUNTRACED	ジョブ制御を実装したシステムでは、*pid*で指定した子が休止しており、かつ、休止後に状態を報告していなければ、その状態を返す。WIFSTOPPEDマクロは、戻り値が休止している子プロセスに対応するものかどうかを判定する。

SVR4には、規格にはない2つのオプション定数がある。WNOWAITは、waitpidが返したプロセスの終了状態を保存する指定であり、そのプロセスに対して再度waitpidできる。WCONTINUEDは、*pid*で指定した子が実行を再開しているが、状態を報告していなければ、その状態を取得する。

waitpid関数は、wait関数にはない3つの機能を提供する。

1. (waitは終了した任意の子の状態を返すが、) waitpidは特定のプロセスを待つことができる。popen関数を説明するときに、この機能に立ち戻る。
2. waitpidは、ブロックしないwaitの機能を果たす。子の状態は取得したいが、ブロックしては困る状況もある。
3. waitpidは(WUNTRACEDオプションで)ジョブ制御を扱える。

●プログラム例●

8.3節のゾンビプロセスの説明を思い出してほしい。子をフォークするがその終了を待たずに、かつ、自分が終了するまでは子をゾンビにしたくないようなプロセスを書きたいときには、forkを2度呼ぶとよいのである。プログラム8.5はこれを行う。

▼プログラム8.5 forkを2度呼んでゾンビプロセスを避ける

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"

int
main(void)
{
    pid_t   pid;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {        /* first child */
        if ( (pid = fork()) < 0)
            err_sys("fork error");
        else if (pid > 0)
            exit(0);    /* parent from second fork == first child */

        /* We're the second child; our parent becomes init as soon
           as our real parent calls exit() in the statement above.
           Here's where we'd continue executing, knowing that when
           we're done, init will reap our status. */

        sleep(2);
        printf("second child, parent pid = %d\n", getppid());
        exit(0);
    }

    if (waitpid(pid, NULL, 0) != pid)   /* wait for first child */
        err_sys("waitpid error");

    /* We're the parent (the original process); we continue executing,
       knowing that we're not the parent of the second child. */

    exit(0);
}
```

2番目の子でsleepを呼んで、親のプロセスIDを出力する前に1番目の子が終了することを保証する。フォークの後で、親と子のどちらかが実行を続ける。どちらが最初に実行を始めるかは分からない。2番目の子でsleepを行わずに、かつ、forkの後で親より早く実行が再開されると、プロセスID 1の代わりに本当の親のプロセスIDが出力される。

プログラム8.5を実行するとつぎのようになる。

```
$ a.out
$ second child, parent pid = 1
```

シェルは、もとのプロセスが終了するとプロンプトを出力することに注意してほしい。これは、2番目の子が親のプロセスIDを出力する前である。□

8.7 wait3とwait4関数

4.3+BSDには2つの関数、wait3とwait4が追加されている。POSIX.1の関数であるwaitやwaitpidが提供できない機能でこれら2つの関数が提供する唯一の機能は、終了したプロセスのすべての子プロセスが使用したリソースの総計をカーネルが報告するための追加引数がある点である。

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>

pid_t wait3(int *statloc, int options, struct rusage *rusage);

pid_t wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);
```

2つの関数の戻り値: 成功ならばプロセスID、0、エラーならば-1

SVR4のBSD互換ライブラリにはwait3関数がある。

リソース情報には、ユーザCPU時間量、システムCPU時間量、ページフォルトの回数、受け取ったシグナルの個数などの情報がある。詳しくはgetrusage(2)のマニュアルページを参照してほしい。このリソース情報は、終了した子プロセスについてのみ取得でき、休止している子プロセスについては取得できない。(このリソース情報は、7.11節で述べたリソースリミットとは異なる。)図8.4には、さまざまなwait関数に使える引数をまとめた。

■図8.4 いろいろなシステムで使えるwait関数の引数

関数	*pid*	*options*	*rusage*	POSIX.1	SVR4	4.3+BSD
wait				•	•	•
waitpid	•	•		•	•	•
wait3		•	•		•	•
wait4	•	•	•			•

8.8 レースコンディション

レースコンディション(race condition)とは、複数のプロセスが共有データについて何かを行う場合、プロセスの実行順序に依存して結果が異なることである。forkの後に親と子のどちらがはじめに実行されるかを(明示的に、あるいは、暗黙のうちに)仮定してforkの後のコードを書くと、fork関数はレースコンディションの温床となる。一般に、どちらのプロセスがはじめに動き出すか分かっていたとしても、そのプロセスが動き始めた直後に何が起こるかはシステムの負荷とカーネルのスケジュールアルゴリズムに依存する。

プログラム8.5の2番目の子が親のプロセスIDを出力する部分は、レースコンディションの能性のある例である。2番目の子が1番目の子より先に動き始めると、その親は1番目の子に一方、1番目の子が先に動き始め、かつ、exitを実行する十分な時間があれば、2番目の子のロセスはinitとなる。sleepを呼び出してはいるが、何の保証にもならない。システムの非常に高い場合、1番目の子が動く機会を与えられる前に、2番目の子はsleepから戻って再開する可能性がある。この種の問題は、「ほとんどの場合」は正しく動作するため、デが困難になりやすい。

プロセスで子の終了を待つには、必ずwait関数の1つを呼ぶ必要がある。プログラム8.5のように子が親の終了を待つ場合には、つぎのような繰り返しを使う。

```
while (getppid() != 1)
    sleep(1);
```

この種の繰り返し(ポーリング(polling)と呼ぶ)の問題点は、条件を調べるために1秒に1回呼び出しが働くため、CPU時間を無駄使いしていることである。

レースコンディションとポーリングを回避するには、複数プロセス間で通知し合う方法が必要である。シグナルを使うことができ、10.16節でその方法の1つを説明する。さまざまなかたちのプロセス間通信(IPC)も使用できる。これらについては第14章と第15章で述べる。

親子関係がある場合には、しばしばつぎのようなシナリオを使う。fork後には、親も子もそれぞれが行うべき初期操作がある。例えば、親はログファイルの記録を子のプロセスIDで更新し、子は親のためにファイルを作成する必要があるかもしれない。各プロセスは互いにそれぞれの初期操作を完了したことを伝え合って、独自の処理を始める前にお互いの完了を待つ。具体的には、つぎのようなシナリオとなる。

```
#include  "ourhdr.h"

TELL_WAIT();    /* TELL_xxx と WAIT_xxx に必要な設定を行う */

if ( (pid = fork()) < 0)
    err_sys("fork error");
else if (pid == 0) {            /* 子 */

    /* 子側で必要なことを行う */

    TELL_PARENT(getppid());     /* 準備ができたことを親に通知 */

    WAIT_PARENT();              /* 親を待つ */

    /* 子側は独自の処理を続行する */
    exit(0);
}

/* 親側で必要なことを行う */

TELL_CHILD(pid);                /* 準備ができたことを子に通知 */

WAIT_CHILD();                   /* 子を待つ */

/* 親側は独自の処理を続行する */
exit(0);
```

ourhdr.hには必要な変数が定義されていると仮定する。5つのルーティン、TELL_WAIT、TELL_PARENT、TELL_CHILD、WAIT_PARENT、WAIT_CHILDはマクロでも関数でもよい。ここでTELLとWAITルーティンの実現方法を説明する。10.16節ではシグナルを用いる方法を、15.2節ではストリームパイプを用いる方法を示す。これら5つのルーティンを用いる例

を見てみよう。

●プログラム例●

プログラム 8.6 は 2 つの文字列を、子から 1 つ、親から 1 つ出力する。カーネルが 2 つのプロセスを動かす順序とそれぞれのプロセスの実行期間に出力が依存するため、このプログラムにレースコンディションが存在する。

▼プログラム 8.6　レースコンディションを有するプログラム

```
#include    <sys/types.h>
#include    "ourhdr.h"

static void charatatime(char *);

int
main(void)
{
    pid_t   pid;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
    }
    exit(0);
}

static void
charatatime(char *str)
{
    char    *ptr;
    int     c;

    setbuf(stdout, NULL);           /* set unbuffered */
    for (ptr = str; c = *ptr++; )
        putc(c, stdout);
}
```

標準出力をアンバッファドとして、1 文字出力する度に write が呼ばれるようにする。この例の目的は、カーネルができるだけ多く 2 つのプロセスを切り替えてレースコンディションを実演することである。(このようにしなければ、つぎに示すような出力は得られない。奇妙な出力が出なければレースコンディションが存在しないのであるが、単にそれは特定のシステムでは発生しないというだけのことである。) つぎの実際の出力は、いかに結果が変わるかを表す。

```
$ a.out
output from child
output from parent
$ a.out
oouuttppuutt  ffrroomm  cphairledn
t
$ a.out
oouuttppuutt  ffrroomm  pcahrielndt

$ a.out
ooutput from parent
utput from child
```

プログラム 8.6 を TELL と WAIT 関数を使うように直す必要がある。プログラム 8.7 がそれである。追加する行の先頭にプラス記号を付けた。

▼プログラム 8.7　プログラム 8.6 をレースコンディションを避けるように変更

```
 #include    <sys/types.h>
 #include    "ourhdr.h"

 static void charatatime(char *);

 int
 main(void)
 {
     pid_t   pid;

+    TELL_WAIT();
+
     if ( (pid = fork()) < 0)
         err_sys("fork error");
     else if (pid == 0) {
+        WAIT_PARENT();      /* parent goes first */
         charatatime("output from child\n");
     } else {
         charatatime("output from parent\n");
+        TELL_CHILD(pid);
     }
     exit(0);
 }

 static void
 charatatime(char *str)
 {
     char    *ptr;
     int     c;

     setbuf(stdout, NULL);           /* set unbuffered */
     for (ptr = str; c = *ptr++; )
         putc(c, stdout);
 }
```

プログラムを実行すると、出力は予想どおりになる。つまり、2 つのプロセスの出力が混ざる

プログラム 8.7 では親が先に実行した。fork に続く行をつぎのように変更すると、子が先に実行される。

```
else if (pid == 0) {
    charatatime("output from child\n");
    TELL_PARENT(getppid());
} else {
    WAIT_CHILD();        /* 子が先に実行される */
    charatatime("output from parent\n");
}
```

演習 8.3 はこの例の続きである。□

8.9 exec 関数

8.3 節では fork 関数の 1 つの使い方として、新しいプロセス (子) を作り、それによって exec 関数の 1 つを呼んで別のプログラムを実行することを述べた。プロセスが exec 関数の 1 つを呼ぶと、そのプロセスは完全に新しいプログラムに置き換えられ、新しいプログラムの main 関数から実行が始まる。exec の過程では、新しいプロセスが作られるわけではないので、プロセス ID は変わらない。exec は現プロセス (現プロセスのテキスト、データ、ヒープ、スタックの各セグメント) をディスクからのまったく新しいプログラムで置き換える。

exec 関数には 6 つの変形版があるが、単に「exec 関数」と呼ぶことにする。これは 6 つのうちのどれでも使えることを意味する。6 つの関数で UNIX のプロセス制御の基本操作が完成する。fork で新たなプロセスを作り、exec 関数で新たなプログラムを始める。exit 関数と 2 つの wait 関数で、終了と終了の待ち合わせを操作する。我々が使用するプロセス制御の基本操作はこれらだけである。後の節で popen や system などの追加機能を有する関数を作成するために、exec 関数の基本操作を用いる。

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *) 0 */ );

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, ...
           /* (char *) 0, char *const envp[] */ );

int execve(const char *pathname, char *const argv[], char *const envp[]);

int execlp(const char *filename, const char *arg0, ... /* (char *) 0 */ );

int execvp(const char *filename, char *const argv[]);
```

6 つの関数の戻り値: エラーならば-1、成功すれば戻らな

これらの関数の第 1 の違いは、はじめの 4 つはパス名引数を取るが、残りの 2 つはファイル名引数を取ることである。*filename* 引数を指定すると、つぎのように扱う。

- *filename* にスラッシュが含まれれば、パス名として扱う。
- さもなければ、PATH 環境変数で指定されるディレクトリ群から実行可能ファイルを探索する。

PATH 変数には、コロンで区切った (パス接頭辞と呼ばれる) ディレクトリのリストを指定する。例えば、*name=value* の形式の環境文字列、

```
PATH=/bin:/usr/bin:/usr/local/bin/:.
```

は、探索するディレクトリを 4 つ指定する。(長さ 0 の接頭辞もカレントディレクトリを意味する。*value* の先頭にコロンを指定するか、文字列中に 2 つのコロンを指定するか、*value* の末尾にコロンを指定する。)

> 機密保護上、探索パスにカレントディレクトリを含めるべきでは「ない」。これについては、[Garfinkel and Spafford 1991] を参照してほしい。

execlp や execvp の 2 つの関数がパス接頭辞の 1 つから実行可能ファイルを探し出したとき、そのファイルがリンケージエディタが生成したマシン実行可能なファイルでない場合には、ファイルはシェルスクリプトであると仮定して、シェルに対する入力ファイルに *filename* を指定して /bin/sh を起動する。

第 2 の違いは、引数リストの渡し方にある (l はリストの略であり、v はベクトルの略である)。関数 execl、execlp、execle では、新しいプログラムに対するコマンド行引数は、個別の引数で指定する必要がある。引数の終わりは null ポインタである。他の 3 つの関数 (execv、execvp、execve) では、引数を指すポインタの配列を作り、この配列のアドレスを 3 つの関数の引数に指定する。

ANSI C のプロトタイプを使用する以前は、3 つの関数 execl 、execle、execlp のコマンド引数の表し方はつぎのようなものであった。

```
char *arg0, char *arg1, ..., char *argn, (char *) 0
```

はコマンド行引数の終わりには null ポインタがあることを特に示している。この null ポインタを定数 0 で表す場合、整数引数として解釈されないように明示的にポインタへ矯正する必要がある。int のサイズが char * のサイズと異なる場合、exec 関数に対する実際の引数は誤ったものとなる。

の違いは、新しいプログラムへ環境リストを渡すことである。名前が e で終わる 2 つの関数 (と execve) では、環境文字列を指すポインタの配列を指すポインタを渡せる。しかし、残りの関数でも、既存の環境を新しいプログラムへコピーするために呼び出し側の environ を使用する。(7.9 節と図 7.5 の環境文字列の説明を思い出してほしい。setenv と putenv の関数を使えるシステムでは、現在の環境とそれに続く子プロセスの環境を変更できるが、親の環境は変更できないことを述べた。) プロセスは通常自分の環境を子に伝えるが、子に環境を設定する場合もある。1 つの例を後ほど述べるが、それは login プログラムが新しいシェルを始めるときである。我々がログインすると、login は数個の変数だけを定義した

特定の環境を作り、シェルの始動ファイルをとおして環境に変数を追加できるようにする。

ANSI C のプロトタイプを使用する前は、execle の引数をつぎのように表していた。

```
char *pathname, char *arg0, ..., char *argn, (char *) 0, char *envp[]
```

最終引数は、環境文字列を指す文字型ポインタの配列のアドレスであることを示す。ANSI C のプロトタイプでは、すべてのコマンド行引数、null ポインタ、*envp* ポインタを '...' という省略した表記で示すため、上の表し方と異なる。

これら 6 つの exec 関数の引数は覚えにくい。関数名に現れる文字が多少助けになる。文字 "p" は関数が *filename* 引数を取り、PATH 環境変数を使って実行可能ファイルを探すことを意味する。文字 "l" は関数が引数リストを取ることを意味する。これは *argv*[] ベクトルを取ることを意味する文字 "v" と排他的である。最後に、文字 "e" は、現在の環境を使う代わりに、関数が *envp*[] 配列を取ることを意味する。図 8.5 にこれら 6 つの関数の違いを示す。

■図 8.5　6 つの exec 関数の違い

関数	*pathname*	*filename*	引数リスト	*argv*[]	environ	*envp*[]
execl	•		•		•	
execlp		•	•		•	•
execle	•		•		•	
execv	•			•	•	
execvp		•		•	•	•
execve	•			•	•	
(名前中の文字)		p	l	v		e

各システムごとに、引数リストと環境リストの合計サイズの上限がある。図 2.7 から、この上限は ARG_MAX で与えられる。POSIX.1 システムでは、この値の最小値は 4096 バイトである。シェルのファイル名リストの展開機能を用いるときに、この上限に触れる場合がある。例えば、コマンド

```
grep _POSIX_SOURCE /usr/include/*/*.h
```

で、シェルがつぎのようなエラーを表示するシステムもある。

```
arg list too long
```

歴史的に、システム V の上限は 5120 バイトである。4.3BSD と 4.3+BSD の配布時の上限は 20480 バイトである。筆者が使用しているシステムでは (プログラム 2.1 の出力を参照すると) メガバイトまで許される！

exec の後にはプロセス ID は変化しないことを述べたが、これ以外にも、呼び出し側のプロセスから新しいプログラムが継承する属性がある。

- プロセス ID と親のプロセス ID
- 実ユーザ ID と実グループ ID
- 補足グループ ID
- プロセスグループ ID

- セッション ID
- 制御端末
- アラームクロックが切れるまでの残り時間
- カレント作業ディレクトリ
- ルートディレクトリ
- ファイルモード作成マスク
- ファイルロック
- プロセスシグナルマスク
- 保留シグナル
- リソースリミット
- tms_utime、tms_stime、tms_cutime、tms_ustime の値

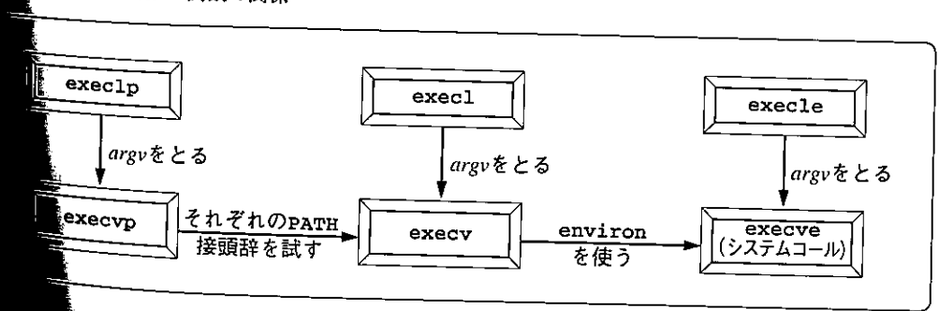
オープンしているファイルの扱いは、各記述子のクローズオンイグゼック (close-on-exec) フラグの値に依存する。図 3.2 と 3.13 節での FD_CLOEXEC フラグの説明を思い出してほしい。プロセスのオープンしている各記述子にはクローズオンイグゼックフラグがある。このフラグを設定していると、exec で記述子はクローズされる。さもなければ、exec の後も記述子はオープンされたままである。fcntl を用いてクローズオンイグゼックフラグを設定しないかぎり、デフォルトは exec の後も記述子をオープンしておくことに注意してほしい。

POSIX.1 では、オープンしたディレクトリストリーム (4.21 節の opendir 関数) は exec の後にはクローズされることを要請する。これは、通常、opendir 関数において、オープンしたディレクトリストリームに対応する記述子に fcntl を用いてクローズオンイグゼックフラグを設定することで行う。

exec の後でも実ユーザ ID と実グループ ID は同じであるが、実行したプログラムファイルのセットユーザ ID とセットグループ ID ビットに依存して実効 ID は変化し得ることに注意してほしい。新しいプログラムにセットユーザ ID ビットが設定されていると、実効ユーザ ID はプログラムファイルの所有者 ID になる。さもなければ、実効ユーザ ID は変わらない (実ユーザ ID には設定されない)。グループ ID も同様に扱われる。

UNIX の多くの実装では、これら 6 つの関数のうち execve だけが、カーネルに対するシステムコールであり、残りの 5 つはこのシステムコールを起動する単なるライブラリ関数である。これら 6 つの関数の関係を図 8.6 に示す。

■8.6　6 つの exec 関数の関係



この図から、ライブラリ関数 execlp と execvp は、PATH 環境変数を処理して、*filename* で指定した実行可能ファイルを含む最初のパス接頭辞を探すことが分かる。

●プログラム例●

プログラム 8.8 は exec 関数をの使用例である。

▼プログラム 8.8　exec 関数の例

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"

char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int
main(void)
{
    pid_t   pid;

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {    /* specify pathname, specify environment */
        if (execle("/home/stevens/bin/echoall",
                   "echoall", "myarg1", "MY ARG2", (char *) 0,
                   env_init) < 0)
            err_sys("execle error");
    }
    if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {    /* specify filename, inherit environment */
        if (execlp("echoall",
                   "echoall", "only 1 arg", (char *) 0) < 0)
            err_sys("execlp error");
    }
    exit(0);
}
```

まず execle を呼ぶが、これにはパス名と特定の環境が必要である。つぎは execlp を呼んでいるが、ファイル名を使い、呼び出し側の環境を新しいプログラムに渡す。execlp がうまくいく理由は、現在のパス接頭辞の 1 つにディレクトリ /home/stevens/bin があるからである。新しいプログラムの最初の引数 argv[0] にパス名のファイル名部分を指定していることに注意してほしい。この引数に完全パス名を指定するシェルもある。

プログラム 8.8 で 2 回 exec されるプログラム echoall をプログラム 8.9 に示す。これはすべてのコマンド行引数と環境リスト全体を出力する簡単なプログラムである。

▼プログラム 8.9　すべてのコマンド行引数とすべての環境文字列を出力する

```
#include    "ourhdr.h"

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

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

    for (ptr = environ; *ptr != 0; ptr++)   /* and all env strings */
        printf("%s\n", *ptr);

    exit(0);
}
```

プログラム 8.8 を実行するとつぎのようになる。

```
$ a.out
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
argv[0]: echoall
$ argv[1]: only 1 arg
USER=stevens
HOME=/home/stevens
LOGNAME=stevens
ここには示さないが、さらに 31 行表示される
EDITOR=/usr/ucb/vi
```

目の exec の argv[0] と argv[1] の間にシェルのプロンプトが表示されることに注意してほ
これは、親が子プロセスの終了を wait していないからである。□

8.10 ユーザ ID とグループ ID の変更

id 関数を用いて実ユーザ ID と実効ユーザ ID を変更できる。同様に、setgid 関数で実グ
ID と実効グループ ID を変更できる。