CHAPTER

14

プロセス間通信

14.1 はじめに

第8章では、プロセス制御の基本操作について述べ、複数プロセスを起動する方法を見た。しかこれらのプロセス間で情報を交換する手段は、オープンしておいたファイルを fork や exec すか、ファイルシステムを介すのみであった。ここでは、プロセスが互いに通信するための別法について述べる。IPC、つまり、プロセス間通信 (interprocess communication) である。IX の IPC は、異なる方式の寄せ集めであり、UNIX のすべての実装においてポータブルな存在しない。図 14.1 は、異なる実装において使用できる異なる方式の IPC をまとめたもの

UNIX の IPC のまとめ

PC の方式	POSIX.1	XPG3	V7	SVR2	SVR3.2	SVR4	4.3BSD	4.3+BSD
重) 付きパイプ)	•	•	•	•	•	•	•	•
パイプ (全二重) リームパイプ					•	•	•	•
ュー		•		•	•	•		
					•	•	•	•

ように、UNIX の実装にかかわらず信頼して使用できる唯一の IPC は半二重のパ

イプである。この図のはじめの7つのIPCは、同一ホスト上のプロセス間のIPCに制約されるのが普通である。終わりの2つのソケットとストリームのみが、異なるホスト上のプロセス間のIPCを一般的に扱える。(ネットワーク上のIPCの詳細については[Stevens 1990]を参照のこと。)この図のなかほどの3つのIPC(メッセージキュー、セマフォ、共有メモリ)はシステムVのみで使えると示してあるが、(SunOSやUltrixのような)バークレーUNIXから派生したベンダ提供のUNIXシステムでも、ベンダがこれら3形式のIPCを追加している。

IPCに関しては、POSIXの別のグループが作業中であり、最終結論はいまだに不明である。1994年以降でないと、IPCに関したPOSIXの結論は出ないようである。

IPCに関する議論を2つの章に分ける。本章では、古典的なIPC、つまり、パイプ、FIFO、メッセージキュー、セマフォ、共有メモリについて述べる。次章では、SVR4と4.3+BSDで使えるより高度なIPC、つまり、ストリームパイプ、名前付きストリームパイプ、および、さらに高度なIPCで可能になるものを見る。

14.2 パイプ

パイプ(pipe)はUNIXのもっとも古い形式のIPCであり、すべてのUNIXシステムで使用できる。これには2つの制約がある。

1. 半二重であること。データは一方向にしか流れない。
2. 共通の祖先を持つプロセス間でのみ使用できる。普通、パイプを作ったプロセスがforkを呼び親と子の間でパイプを使用する。

ストリームパイプ(15.2節)は第1の制約を克服するものであり、FIFO(14.5節)と名前付きストリームパイプ(15.5節)は第2の制約を克服することを見る。これらの制約にもかかわらず、このパイプはもっとも一般的に使用されるIPCである。

パイプを作成するにはpipe関数を呼ぶ。

```
#include <unistd.h>

int pipe(int filedes[2]);
```

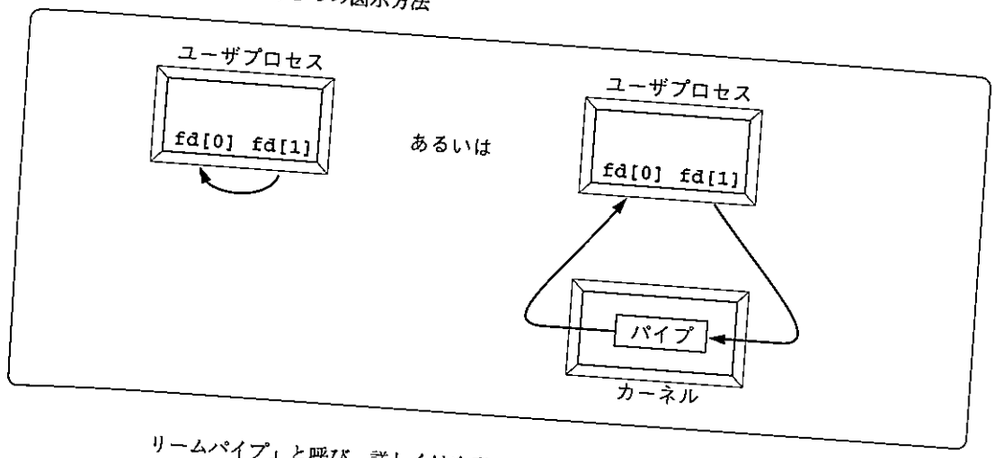
戻り値: 成功ならば0、エラー

*filedes*引数を介して記述子が2つ返される。*filedes[0]*は読み取り用にオープンされており、*filedes[1]*は書き込み用にオープンされている。*filedes[1]*への出力は*filedes[0]*からの入力である。

図14.2に示すように、パイプの図示方法は2種類ある。図の左半分では、パイプは1つのプロセスにつながっている。図の右半分は、パイプのデータがカーネル内を何度も通過することを表す。

SVR4においては、パイプは全二重である。いずれの記述子を読んでも書いても使える。また、図14.2の矢印は、両端に付けるべきである。このような全二重

■図14.2 UNIXパイプの2つの図示方法



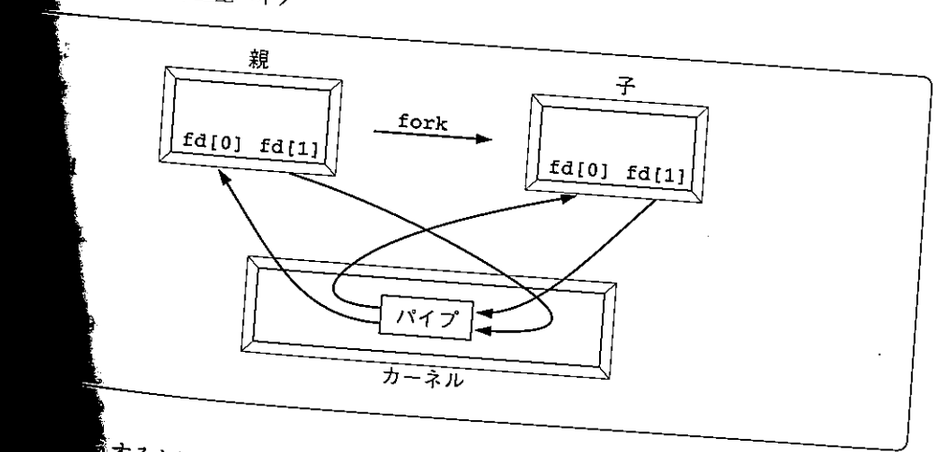
リームパイプ」と呼び、詳しくは次章で述べる。POSIX.1では半二重のパイプのみを規定しており、pipe関数は一方向のパイプを作成するとしている。

fstat関数(4.2節)は、パイプの端を指すファイル記述子のファイルの種類としてFIFOを返す。S_ISFIFOマクロでパイプかどうかを調べることができる。

POSIX.1では、stat構造体のst_sizeメンバーは、パイプに対しては未定義であると規定している。しかし、パイプの読み取り側を指すファイル記述子にfstat関数を適用すると、多くのシステムではパイプから読み取れるバイト数をst_sizeメンバーに収める。しかし、これには移植性がない。

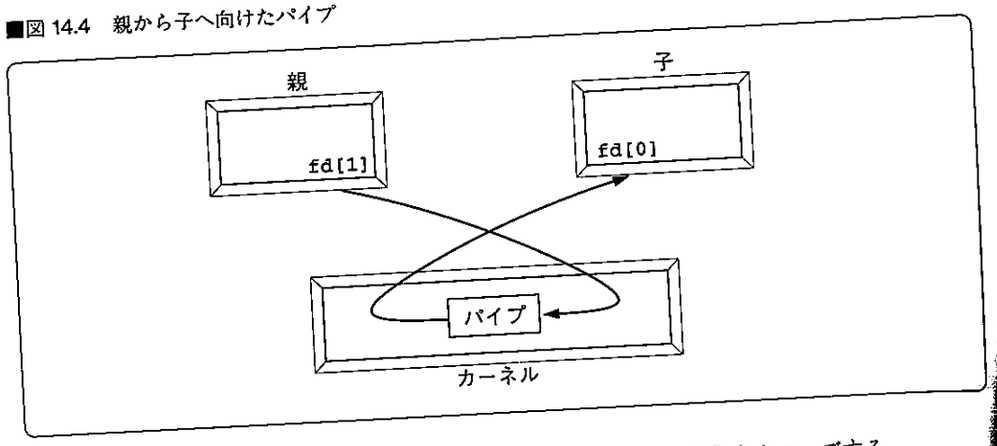
ープロセスのパイプは、ほとんど使い道がない。普通、親から子、あるいは、子から親への通信路を作成するために、プロセスはpipeを呼んで、続いてforkを呼ぶ。図14.3にこの様示す。

8 fork後の半二重パイプ



するかは、どちら向きにデータを流すかに依存する。親から子に向けたパイプで読み取り側(fd[0])をクローズし、子は書き込み側(fd[1])をクローズする。図を示す。

■図 14.4　親から子へ向けたパイプ



子から親へ向けたパイプでは、親は fd[1] をクローズし、子は fd[0] をクローズする。

パイプの一端をクローズするとき、つぎの規則を適用する。

1. 書き込み側をクローズしたパイプから read する場合、すべてのデータを読み取った後の read は、ファイルの終わりを表す 0 を返す。(技術的には、パイプへの書き手がいなくなるまでは、ファイルの終わりは保証されないというべきである。パイプを指す記述子を複製することは可能であり、複数プロセスがパイプを書き込みでオープンしておくことができる。しかし、普通はパイプには 1 つの読み手と 1 つの書き手がいるだけである。次章の FIFO では、しばしば 1 つの FIFO に複数の書き手がいる場合を見る。)
2. 読み取り側がクローズされたパイプに write すると、シグナル SIGPIPE が生成される。シグナルを無視したり、シグナルを捕捉してシグナルハンドラから戻ったとしても、write はエラー返し、errno には EPIPE が設定される。

パイプ (や FIFO) に書くとき、定数 PIPE_BUF はカーネルのパイプバッファのサイズを指定。PIPE_BUF 以下のバイトを write する場合、同じパイプ (や FIFO) を操作するプロセスが操作に入り込むことはない。複数プロセスがパイプ (や FIFO) に出力する場合に、PIPE_BU えるバイトを write すると、別の書き手のデータが間に入り込むことがある。

●プログラム例●

親から子へ向かうパイプを作成し、パイプにデータを送るプログラムをプログラム 14.1

▼プログラム 14.1　パイプを介して親から子へデータ転送する

```
#include    "ourhdr.h"

int
main(void)
{
    int     n, fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");

    if ( (pid = fork()) < 0)
        err_sys("fork error");

    else if (pid > 0) {     /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);

    } else {                /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }

    exit(0);
}
```

□

この例では、パイプを指す記述子に対して直接 read や write を行う。パイプを指す記述子を標準入力や標準出力へ複製すると、より興味深い。子はしばしば別のプログラムを exec し、そのプログラムは (作成したパイプである) 標準入力を読んだり、標準出力 (パイプ) に書いたりできる。

●プログラム例●

作成した出力を一度に 1 ページずつ表示するプログラムを考える。ページごとに表示する既存の UNIX ユーティリティを再作成する代わりに、ユーザの好みのページ表示プログラム (pager) を動する。全データを一時ファイルに書き出してから、そのファイルを表示するために system を代わりに、出力を直接ページ表示プログラムにパイプしたい。それには、パイプを作成し、子セスを fork し、子側の標準入力をパイプの読み取り側に設定し、ユーザのページ表示プログを exec する。プログラム 14.2 にこれを行う方法を示す。(この例では、表示するファイルのをコマンド行引数から取る。この種のプログラムでは、端末に表示すべきデータはメモリ中に)

グラム 14.2　ページ表示プログラムへファイルをコピーする

```
de    <sys/wait.h>
le    "ourhdr.h"

DEF_PAGER   "/usr/bin/more"     /* default pager program */


rgc, char *argv[])

n, fd[2];
pid;
line[MAXLINE], *pager, *argv0;
*fp;

!= 2)
```

```
        err_quit("usage: a.out <pathname>");
    if ( (fp = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);

    if (pipe(fd) < 0)
        err_sys("pipe error");

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid > 0) {                                    /* parent */
        close(fd[0]);       /* close read end */
            /* parent copies argv[1] to pipe */
        while (fgets(line, MAXLINE, fp) != NULL) {
            n = strlen(line);
            if (write(fd[1], line, n) != n)
                err_sys("write error to pipe");
        }
        if (ferror(fp))
            err_sys("fgets error");

        close(fd[1]);   /* close write end of pipe for reader */
        if (waitpid(pid, NULL, 0) < 0)
            err_sys("waitpid error");
        exit(0);

    } else {                                               /* child */
        close(fd[1]);   /* close write end */
        if (fd[0] != STDIN_FILENO) {
            if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin");
            close(fd[0]);   /* don't need this after dup2 */
        }

            /* get arguments for execl() */
        if ( (pager = getenv("PAGER")) == NULL)
            pager = DEF_PAGER;
        if ( (argv0 = strrchr(pager, '/')) != NULL)
            argv0++;        /* step past rightmost slash */
        else
            argv0 = pager;  /* no slash in pager */

        if (execl(pager, argv0, (char *) 0) < 0)
            err_sys("execl error for %s", pager);
    }
}
```

---

fork を呼ぶ前に、パイプを作成する。fork を呼んだ後、親はパイプの読み取り側を
し、子は書き込み側をクローズする。続いて、子は dup2 を呼び、パイプの読み取り側
とする。ページ表示プログラムが実行されると、その標準入力はパイプの読み取り側を

記述子を別の記述子に (fd[0] を子の標準入力に) 複製する場合、記述子が目的の値

ないように注意しなければならない。記述子が目的の値になっているときに dup2 と close を呼ぶと、唯一の記述子をクローズしてしまう。(2つの引数が同じ値である場合の 3.12 節で述べた dup2 の動作を思い出してほしい。) このプログラムでは、シェルが標準入力をオープンしていない場合、プログラムのはじめの fopen が使われていないもっとも小さい記述子 0 を使用するはずであり、fd[0] が標準入力に等しくなるはずはない。それにもかかわらず、記述子を別の記述子へ複製するために dup2 と close を呼ぶ場合には、予防策として、まず記述子を比較するようにするべきである。

ユーザのページ表示プログラムの名称を取得するための環境変数 PAGER の使い方に注意してほしい。これがうまくいかなければ、デフォルトのものを使う。これは環境変数の一般的な使い方である。□

●プログラム例●

8.8 節の 5 つの関数、TELL_WAIT、TELL_PARENT、TELL_CHILD、WAIT_PARENT、WAIT_CHILD を思い出してほしい。プログラム 10.17 ではシグナルを用いて実装を示した。プログラム 14.3 は、パイプを用いた実装である。

▼プログラム 14.3 親と子を同期するルーティン

```
#include    "ourhdr.h"

tatic int  pfd1[2], pfd2[2];

oid
ELL_WAIT()

  if (pipe(pfd1) < 0 || pipe(pfd2) < 0)
      err_sys("pipe error");


_PARENT(pid_t pid)

f (write(pfd2[1], "c", 1) != 1)
    err_sys("write error");


ARENT(void)

    c;

ead(pfd1[0], &c, 1) != 1)
rr_sys("read error");
 != 'p')
r_quit("WAIT_PARENT: incorrect data");


pid_t pid)
```

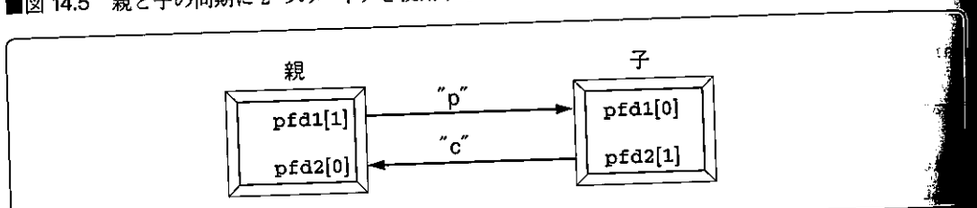
```
{
    if (write(pfd1[1], "p", 1) != 1)
        err_sys("write error");
}

void
WAIT_CHILD(void)
{
    char    c;

    if (read(pfd2[0], &c, 1) != 1)
        err_sys("read error");
    if (c != 'c')
        err_quit("WAIT_CHILD: incorrect data");
}
```

図 14.5 に示すように、fork する前にパイプを 2 個作成する。

■図 14.5 親と子の同期に 2 つのパイプを使用する



TELL_CHILD が呼ばれると親は上側のパイプに“p”を書き、TELL_PARENT が呼ばれると子は
のパイプに“c”を書く。対応する WAIT_xxx 関数は、ブロックする read で 1 文字を読む。

各パイプには余計な読み手がいるが、問題にはならない。つまり、子が pfd1[0] から読む
に加えて、親も上側のパイプの読み取り側である。親はこのパイプから読むことはないので
□

14.3 popen と pclose 関数

他プロセスの出力を読んだり、他プロセスへの入力を書いたりするために他プロセスへ
を作成する操作が多いため、標準入出力ライブラリには歴史的に popen と pclose 関数が
れら 2 つの関数は、これまでは自分自身で行ってきたやっかいな作業をすべて扱う。つま
プを作成し、子を fork し、未使用のパイプの端をクローズし、コマンドを実行するため
を exec し、そのコマンドの終了を wait するのである。


```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);
```

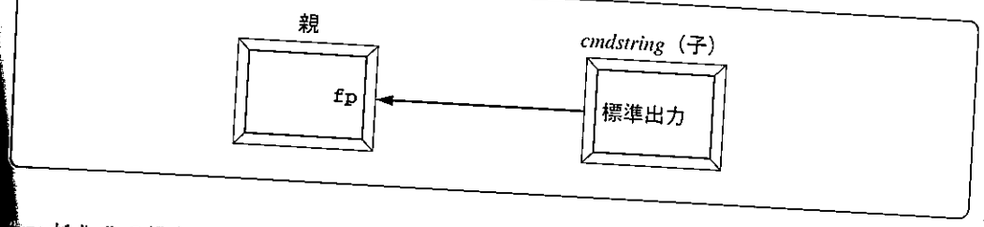
戻り値: 成功ならばファイルポインタ、エラーならば NULL

```
int pclose(FILE *fp);
```

戻り値: *cmdstring* の終了状態、エラーならば -1

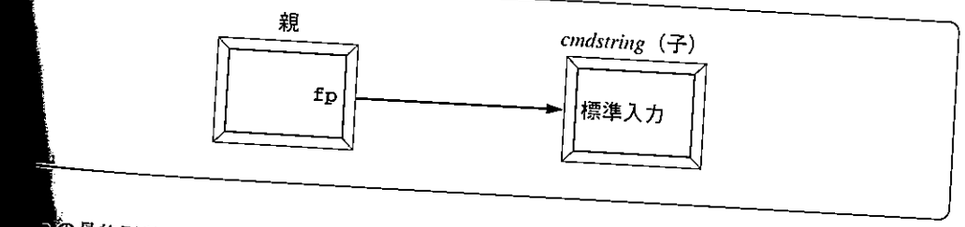
関数 popen は fork してから *cmdstring* を実行するために exec し、標準入出力のファイルポインタを返す。*type* が "r" の場合、*cmdstring* の標準出力がファイルポインタに接続される (図 14.6)。

■図 14.6 fp = popen(*command*, "r") の結果



type が "w" の場合、*cmdstring* の標準入力がファイルポインタに接続される (図 14.7)。

図 14.7 fp = popen(*command*, "w") の結果



の最終引数を覚える 1 つの方法は fopen の場合と同じで、*type* が "r" ならばファイルポ
を読むことができ、*type* が "w" ならばファイルポインタに書くことができる。
se 関数は標準入出力ストリームをクローズし、コマンドの終了を待ち、シェルの終了状態
(終了状態については 8.6 節で述べた。system 関数 (8.12 節参照) もこれを返す。) シェル
きなければ、pclose が返す終了状態は、シェルで exit(127) を実行したものと同じで

ng は、つぎのように Bourne シェルで実行される。

cmdstring

ルは *cmdstring* に現れるすべての特別な文字を展開する。このため、例えば、つぎの