Linux システムコール演習 2

岡野浩三

令和 2 年 6 月 17 日

プロセスを新たに生成する場合 UNIX では fork と exec の 2 段階を経る．なお Windows では CreateProcess の 1 つですましている．fork は分身の術，exec は変身の術と思えばこの 2 段階は理解できるであろう．

典型的な fork のプログラムを以下に示す．

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

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

    if ((pid = fork()) < 0) {
        perror("fork was failed");
        exit(1);
    }
    if (pid == 0 ) {
        printf("I am a child\n");
        pid = getpid();
        printf("My process id is %d.\n", pid);
        exit(0);
    } else {
        int status;
        waitpid(pid, &status, 0);
        printf("I am the parent.\n");
        pid = getpid();
        printf("My process id is %d.\n", pid);
        exit(0);
    }
}
```

問 1
fork に必要なヘッダーファイルを *man* で調べよ．

fork は失敗したときは-1 が返る．成功したときはプロセスが文字通りフォークし (分身し)，親プロセスと子プロセスの 2 つに分かれる．親プロセスのほうには fork の返り値は子プロセスのプロセス ID となり，子プロセスのほうは 0 となる．

したがって，この戻り値を頼りに，以降，親プロセス，子プロセスで異なった処理ができる．

このプログラムは子プロセス側では システムコール getpid() を呼んで自プロセスのプロセス ID を取得し，それを表示している．

親プロセスもほぼ同じである．ただし，一般的には親プロセスは，システムコール waitpid で子プロセスを wait する必要がある．waitpid が実行された場合は親プロセスは子プロセスの実行終了を待って，終了したのち waitpid 以降の処理を行う．今回のように比較的すぐ終了する場合は waitpid を使わなくても良い．

演習 1
このプログラムを実行せよ．出力される文字の順番も観察せよ．

先ほどのプログラムでは`&status` を使っていなかった．以下のように変えてみる (main 以降のみ表示)

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

    if ((pid = fork()) < 0) {
        perror("fork was failed");
        exit(1);
    }
    if (pid == 0 ) {
        printf("I am a child\n");
        pid = getpid();
        printf("My process id is %d.\n", pid);
        exit(77);
    } else {
        printf("I am the parent.\n");
        int status;
        waitpid(pid, &status, 0);
        printf("My child retruns %d when he exits.\n", WEXITSTATUS(status));
        pid = getpid();
        printf("My process id is %d.\n", pid);
        exit(0);
    }
}
```

子プロセスの終了ステータスの値 (今回，適当に 77 を使った) を waitpid で変数 status に受け取り，終了ステータスの値を WEXITSTATUS(status) で表示している．なお，ここで WEXITSTATUS は status から終了ステータスの値を計算するマクロである．

次に親プロセスが waitpid() を呼び出さない場合について若干補足する．先に説明したように親プロセスもフォーク後すぐ終了するのであれば waitpid() を呼び出す必然性はない．逆にいうと親プロセスの処理が長いときに問題となる．

調べるために少しプログラムを変更してみよう．先ほどと同様に main 以降を以下に記す．

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

    if ((pid = fork()) < 0) {
        perror("fork was failed");
        exit(1);
    }
    if (pid == 0 ) {
        printf("I am a child\n");
        pid = getpid();
        printf("My process id is %d.\n", pid);
        exit(0);
    } else {
        printf("I am the parent.\n");
//          int status;
//          waitpid(pid, &status, 0);
        sleep(100);
        exit(0);
    }
}
```

改変版では親プロセスのほうで sleep(100) を呼び出している．単位は秒なのでこのプロセスは 100 秒ほど眠りつづけることになる．

```
I am the parent.
I am a child
My process id is 1448.
```

この 100 秒の間に別のターミナルから例によって以下のコマンドを実行してみよう．ここで fork はこのプログラムの名前である (適宜読み替えよ)．

```
$ps a |grep fork
```

するとたとえば以下のようになる．

```
1447 pts/4    S+     0:00 ./fork
1448 pts/4    Z+     0:00 [fork] <defunct>
1451 pts/1    S+     0:00 grep --color=auto fork
```

注目していただきたいのは

```
1448 pts/4    Z+     0:00 [fork] <defunct>
```

である．

このプログラムの子プロセスは早々と終了したはずなのに親プロセスが終了していないおかげで<defunct>となっている．またZというフラグがついている．OSからは「消滅したけどゾンビ状態だ」と宣言されてしまっている．

OSは親プロセスがwaitを呼び出す可能性がある限り，子プロセスの終了状態の情報を保持しておく必要がある．

そのため実際にはプロセスが消滅しているにもかかわらず，そのエントリをOSは消せていない状況を意味している．この状況は親プロセスがwaitを呼び出すか，あるいは親プロセスが消滅するまで続く．実際に親プロセスを終了したのち，再度psで確かめてみよ．こんどは「ゾンビ」が消えているはずである．

演習 2
ここまでのプログラムを実行せよ．

分身の術はこれくらいにしておく．単にプロセスを複製しただけでは本格的に別のプロセス(別のプログラム)になったとは言いがたい．次にまったく別のプログラムを動かすようにしてみる．そのため(変身の術)としてexecシステムコール(群)がある．

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

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

    if ((pid = fork()) < 0) {
        perror("fork was failed");
        exit(1);
    }
    if (pid == 0 ) {
        printf("(c)I am a child.\n");
        printf("(c)I will be ls command!\n");
```

```
        execl("/bin/ls", "ls", "-la", NULL);
        perror("exec was failed");
        exit(1);
    } else {
        printf("(p)I am the parent.\n");
        printf("(p)I will sleep a while.\n");
        int status;
        waitpid(pid, &status, 0);
        sleep(10);
        printf("(p)I will terminate myself.\n");
        exit(0);
    }
}
```

このプログラムでは親プロセスは子プロセスの終了を待って，かつ 10 秒ほどスリープし，その後正常終了している．

一方，子プロセスでは execl を使って /bin/ls コマンドに「変身」している．引数の 2 番目には argv[0] に相当することを記述することに注意．残りの引数は ls コマンドの引数となっている．引数の数は一般に可変なので，最後の引数を NULL として引数列の終了を表している．正常に exec できれば，exec の後の行は実行されない．逆に言えば実行される場合は exec が失敗したときであるのでここ (以降) にはエラー時の処理を記述するのが一般的である．

結局のところ fork とのあわせ技で，ls コマンドを実行する新しいプロセスを実現できたことになる．

なお exec システムコール群は引数の渡し方に応じて複数のシステムコール関数を用意している．詳細は各自で調べること．

問 2

exec システムコール群を調べよ．

演習 3

ここまでのプログラムを実行せよ．

皆が今使っているシェルプログラムも (きわめて抽象的にみれば) 結局のところ，ユーザからコマンドを文字列として標準入力からうけとり，その文字列を fork と exec で新規プロセスとして実行するということをずっと繰り返しているに過ぎない．

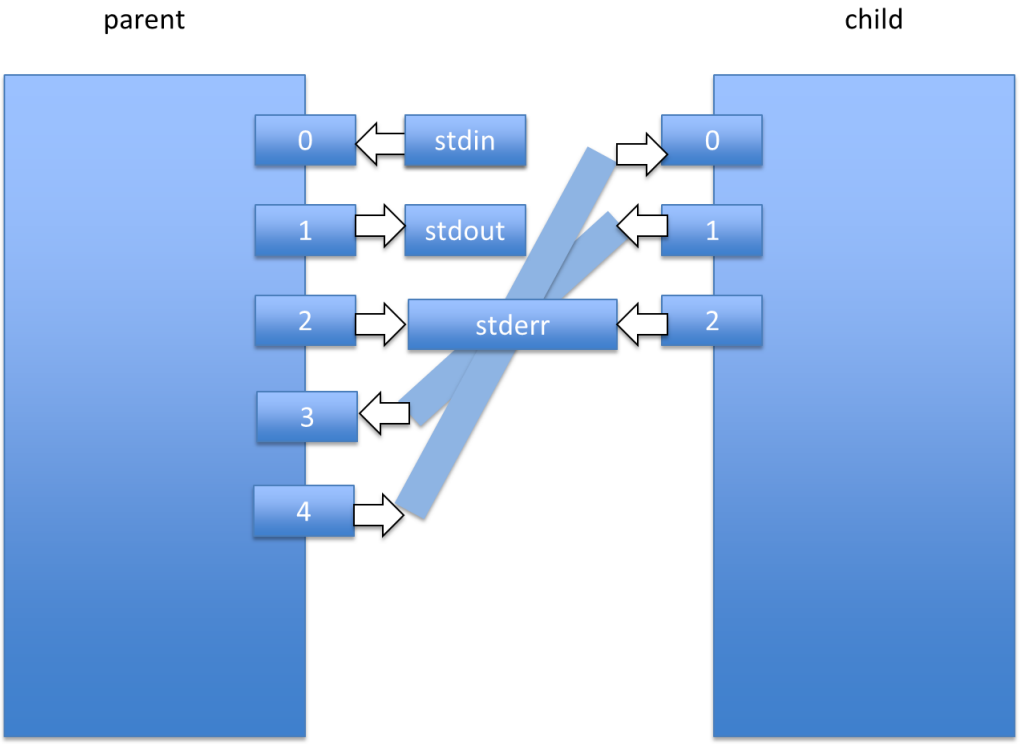
```
while(true) {
    printfプロンプト();
    command = ……; // コマンド入力
    if (command が実行可能)
        command = パスの正規化(command);
    if ((pid = fork()) < 0)エラー処理
```

```
        ;
    if ((pid == 0) {
        exec(command);エラー処理
        ;
    }
}
```

fork, exec ときたらパイプの話をするのがこの手のシステムコールの定石だ．

ここでは次の 2 つの例題を実装していく 1 つ目は `command1 | command2` のパターンである．

2 つ目は図で表すが以下の図のパターンである．



1 つ目は典型的なシェルのパイプの処理である．シェルでできるのでわざわざ C でプログラムする必然性は低いが理解のため取り上げる．

2 つ目のほうはたとえば command2(図では child) が複雑な計算をするコマンドで入力，出力をそれぞれ標準入出力を介して行うものを想像してもらうとよい．具体的に例を挙げると bc コマンドが考えられる．そして command1(図では parent) の標準入出力はいずれも標準のまま端末につながっており，command2 を内部で繰り返し用いることを想定している．ユーザから情報を標準入力から受け取り，最終的には標準出力に出力するのだが，その間の計算のところで複数回 command2 を使う．ここで「使う」の意味は command2 はプロセスとしてはずっと起動したままなのだが，command2 に対して，データを何度も送り，計算してもらい，そのつど計算結果を受け取るイメージである．

親プロセスが 4 番のファイルディスクリプタを指定して送り出したデータは子プロセスの 0 番 (すなわち標準入力) から読み出される．同様に子プロセスが 1 番 (すなわち標準出力) に書き出したデータは親プロセスの 3 番から読み出すことができる．

こちらの処理は，シェルの (無名) パイプで実現するのは無理であろう．基本的にシェルのパイプはデータが 1 方向に 1 本複数のコマンドを経由して流れていくだけだが，2 つ目のパターンではデータが 2 つのコマンド（プログラム）の間で行き来している．データの流れそのものはパイプを使えるが，パイプの接続先の「つけかえ工事」が必要である．

まずは簡単な 1 つ目からプログラムを見ていく．

`command1 | command2` を 1 つのコマンドとして実現するプログラムを例題とする．パイプとして 2 つのコマンドを組み合わせる例はいろいろ考えられるが今回は `ls -lR . | grep file` を考えよう．

file で指定した文字列に部分一致するファイルやディレクトリをカレントディレクトリ以下から探し出し，その long 形式での情報を得ることを意図している．

以下はそのプロラムである．

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

void die (const char *s) {
    perror(s);
    exit(1);
}

int main(int argc, char *argv[]) {
     int pid;
     int fds[2];

     if (argc < 2) {
          fprintf(stderr, "usage: %s file\n", argv[0]);
          exit(1);
     }

     if (pipe(fds) < 0 ){
          die("I cannot make a pipe");
     }
    // assert fds[0]==3 && fds[1]==4;
    // fds[0] for read and fds[1] for write

     if ((pid = fork()) < 0) {
```

```
        die("fork was failed");
    }
    if (pid == 0 ) { // child's side
        if(close(0) < 0) {
          die ("I cannot close stdin");
        }
        if(dup2(fds[0], 0) < 0) {
          die ("I cannot duplicate the pipe");
        }
        if(close(fds[0]) < 0) {
          die ("I cannot close unnecessary pipe");
        }
        if(close(fds[1]) < 0) {
          die ("I cannot close unnecessary pipe");
        }
        execl("/bin/grep", "grep", argv[1], NULL);
        die("exec was failed");
    } else {   // parent's side
        if(close(1) < 0) {
          die ("I cannot close stdout");
        }
        if(dup2(fds[1], 1) < 0) {
          die ("I cannot duplicate the pipe");
        }
        if(close(fds[0]) < 0) {
          die ("I cannot close unnecessary pipe");
        }
        if(close(fds[1]) < 0) {
          die ("I cannot close unnecessary pipe");
        }
        execl("/bin/ls", "ls", "-Rl", ".", NULL);
        die("exec was failed");
    }
}
```

全体的な構成は前回までの fork & exec の構成と同様である．

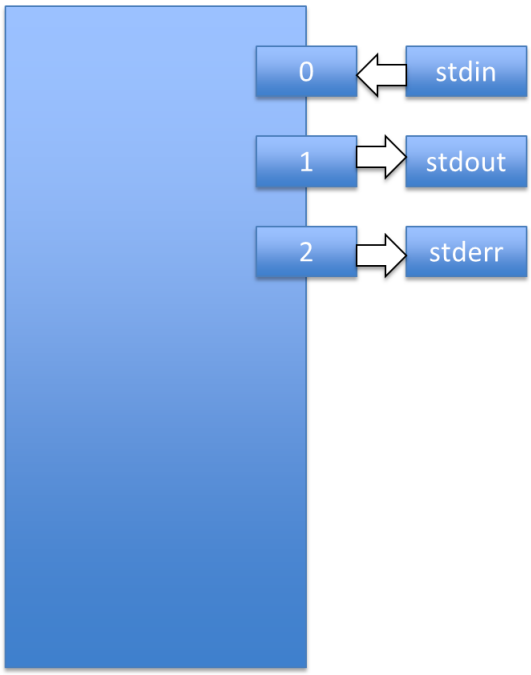
新規のシステムコールとして pipe(fds) が出てくる．ここで fds はファイルディスクリプタを納めるための int 型サイズ 2 の配列である．

pipe() は未割り当ての新規ファイルディスクリプタ 2 つを用いて，1 つ目のファイルディスクリプタから 2 つ目のファイルディスクリプタへのストリーム (stream) を作成する．そして成功すれば

その 2 つのディスクリプタの値を引数の配列に入れる．

ストリームとはデータの流れ (を表現するデータ構造) であり，実際には FIFO バッファである．

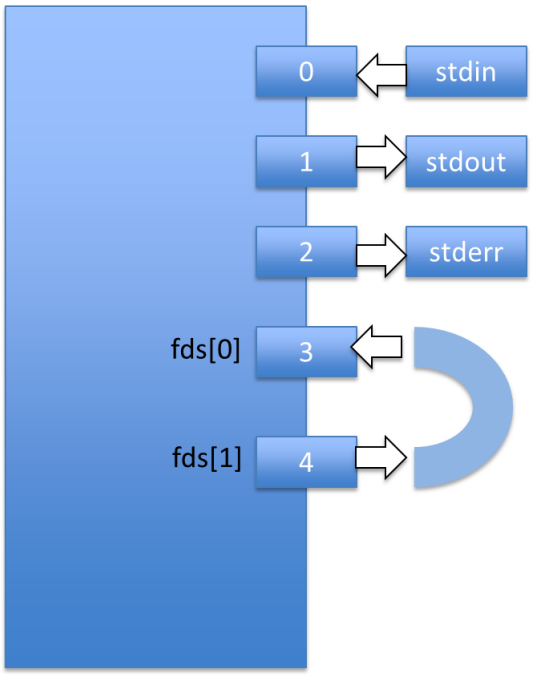
Main の処理において pipe の呼び出し前では次の図のようになっている．



ファイルディスクリプタ 0,1,2 にそれぞれ標準入力，標準出力，標準エラー出力がつながっている．ここで矢印はデータの流れの方向である．たとえば標準入力からのデータはファイルディスクリプタ 0 から読み取る (書き込むのではない) ことを表している．

```
if (pipe(fds) < 0 ){
    die("I cannot make a pipe");
}
```

のパイプ呼び出しが成功すると次の図のようになる．



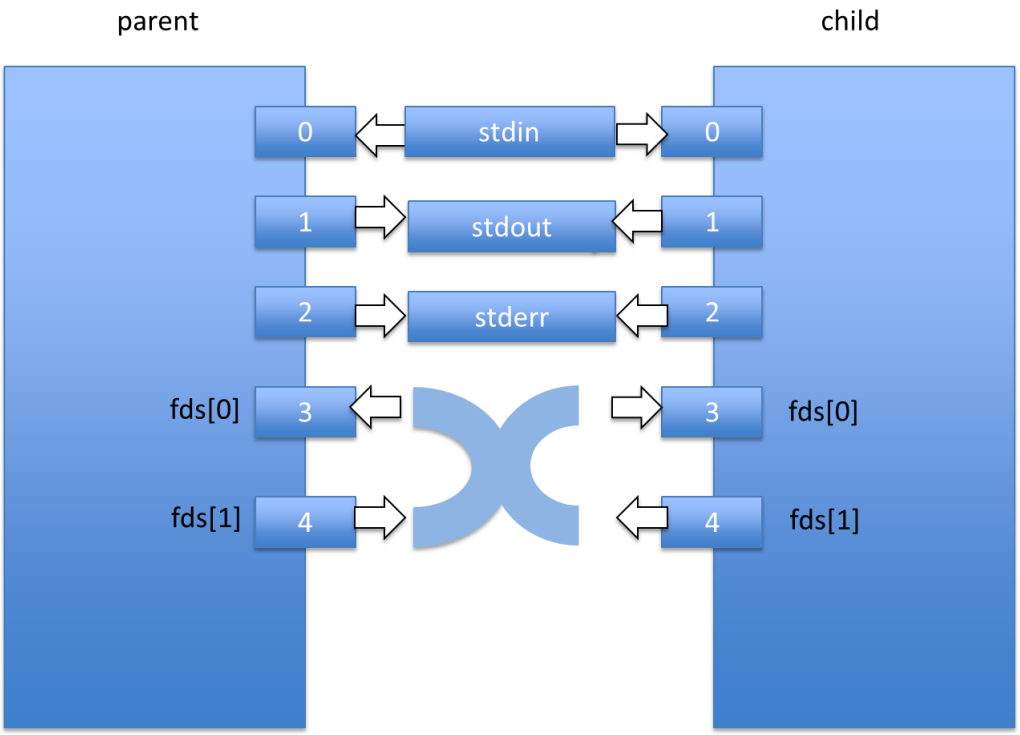
パイプ (FIFO キュー) が作成され，ファイルディスクリプタ 4 に出力されたデータは，ファイルディスクリプタ 3 から読み取りできるようになる．

また配列の要素 `fds[0]`, `fds[1]` の値はそれぞれファイルディスクリプタ 3 と 4 となっている．

このままではあまり有用な使い道はない．

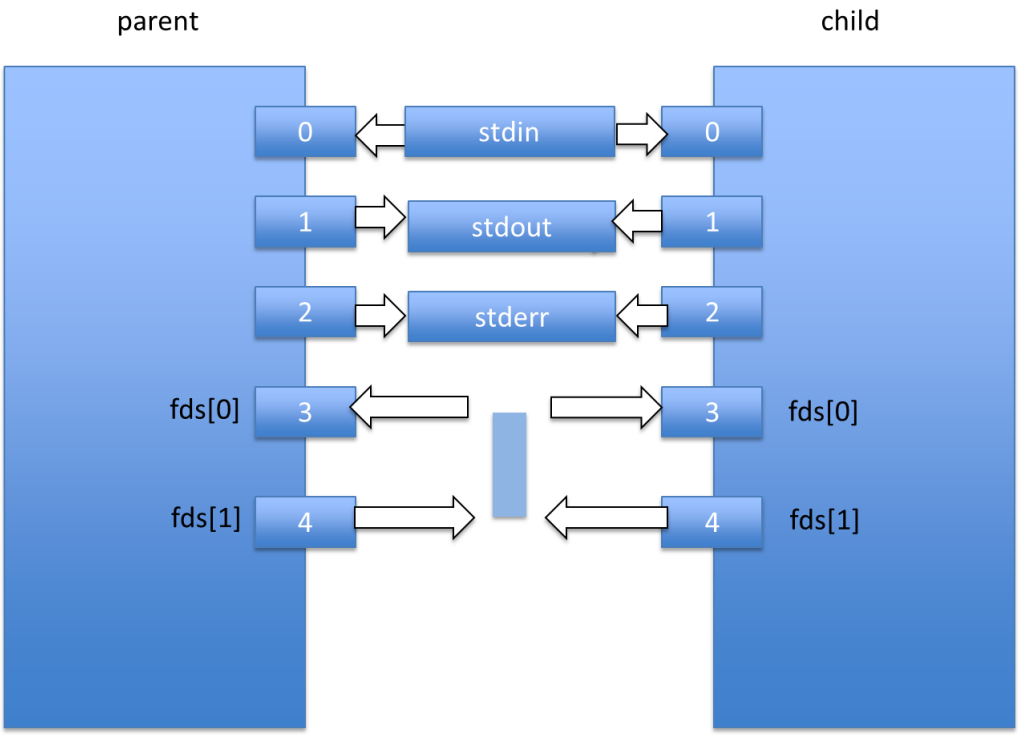
```
if ((pid = fork()) < 0) {
    die("fork was failed");
}
```

このフォークにより親子のプロセスに分離するが，ファイルディスクリプタがさしている構造体は親子間で共有している．したがってこの文の実行後は以下のようになっている．



親子プロセスで各ファイルディスクリプタがそれぞれ同じ共通の構造をさしていることに注意．

図ではパイプそのものも複製しているかのように読めるがパイプの実体は 1 つである．誤解のない表現をするなら以下の図のほうがより正確である．

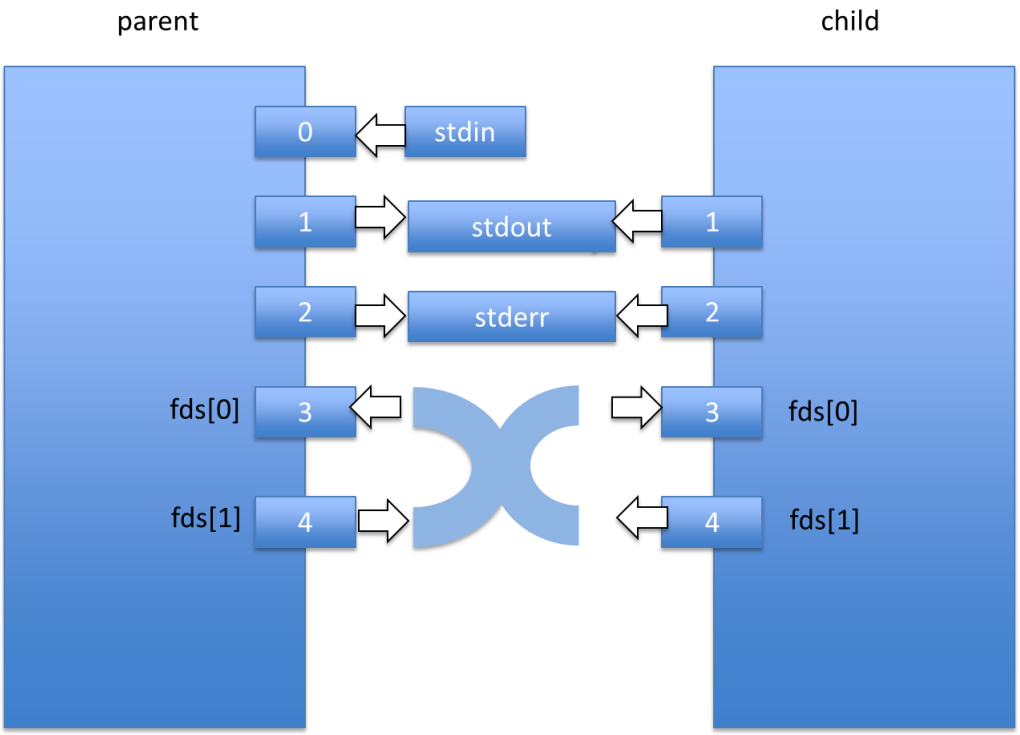


これ以降，親子で処理内容が異なる．

子プロセス側

```
if(close(0) < 0) {
    die ("I cannot close stdin");
}
if(dup2(fds[0], 0) < 0) {
    die ("I cannot duplicate the pipe");
}
if(close(fds[0]) < 0) {
    die ("I cannot close unnecessary pipe");
}
if(close(fds[1]) < 0) {
    die ("I cannot close unnecessary pipe");
 }
```

まず最初の close により次のようになる．



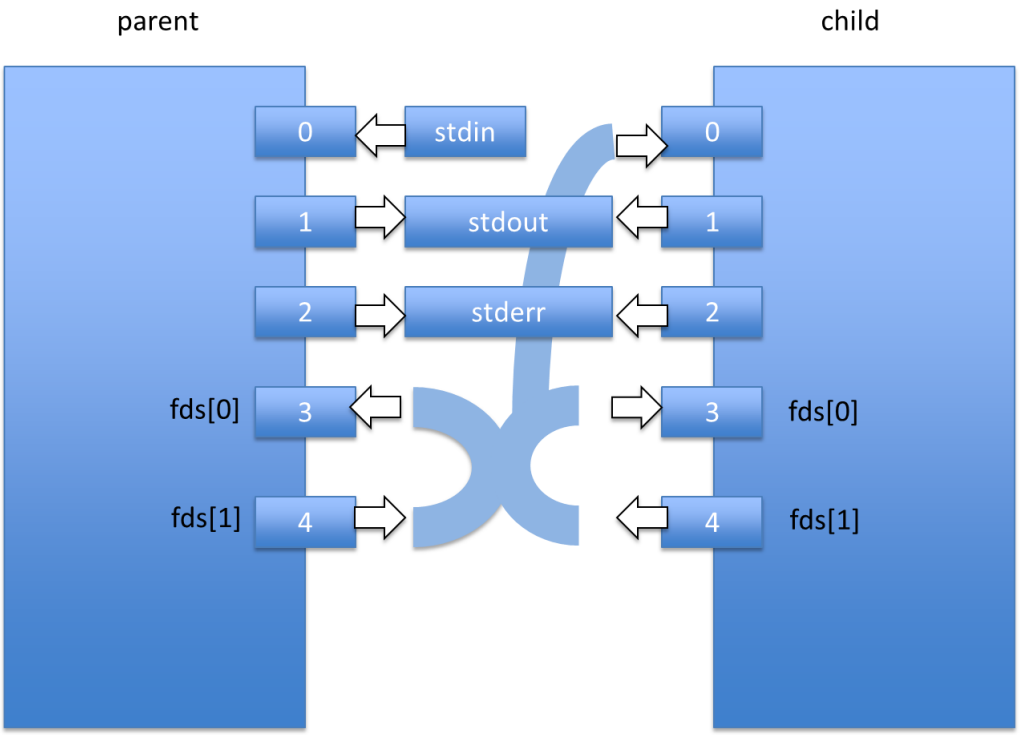
単に 子プロセス側のファイルディスクリプタ 0 が切り離される．

次に新たなシステムコール `dup2(fds[0], 0)` が呼ばれている．

dup2 は第一引数のファイルディスクリプタの指すストリームを第二引数のファイルディスクリプタにコピー (duplicate) する．

この場合，`fds[0]` すなわち，ファイルディスクリプタ 3 の指すストリームがファイルディスクリプタ 0(もとの標準入力) にコピーされる．

`dup2(fds[0], 0)` の実行直後は次の図のようになる．

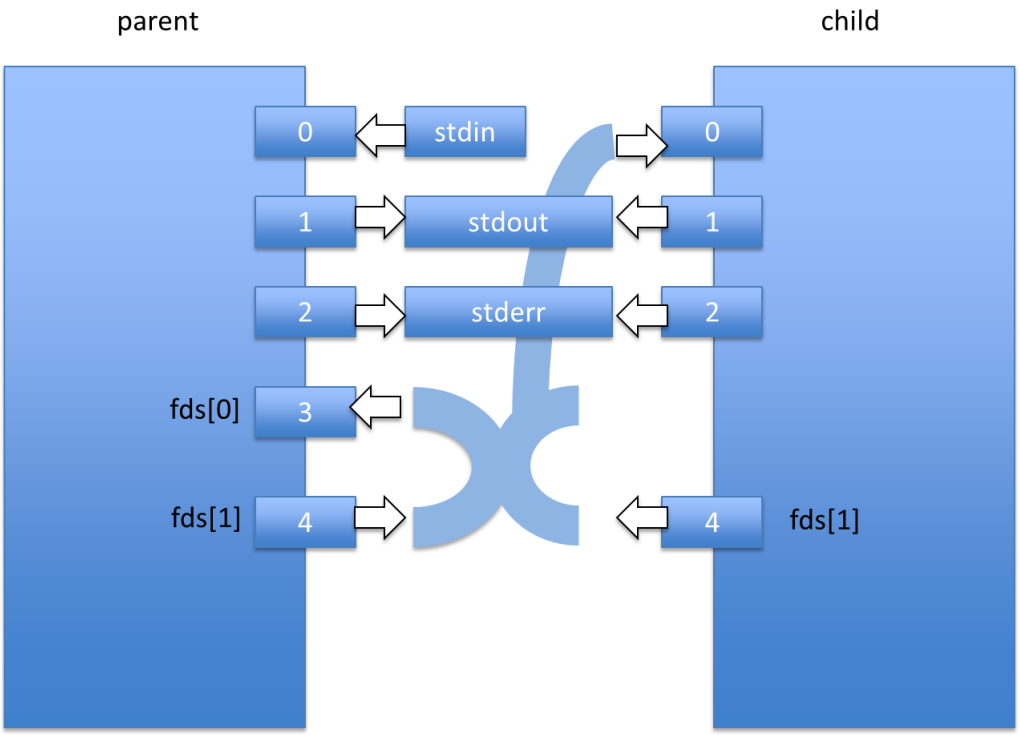


図のようにファイルディスクリプタ 0 の指すストリームがファイルディスクリプタ 3 (=fds[0]) のものと同様になる.

次の

```
if(close(fds[0]) < 0) {
    die ("I cannot close unnecessary pipe");
}
```

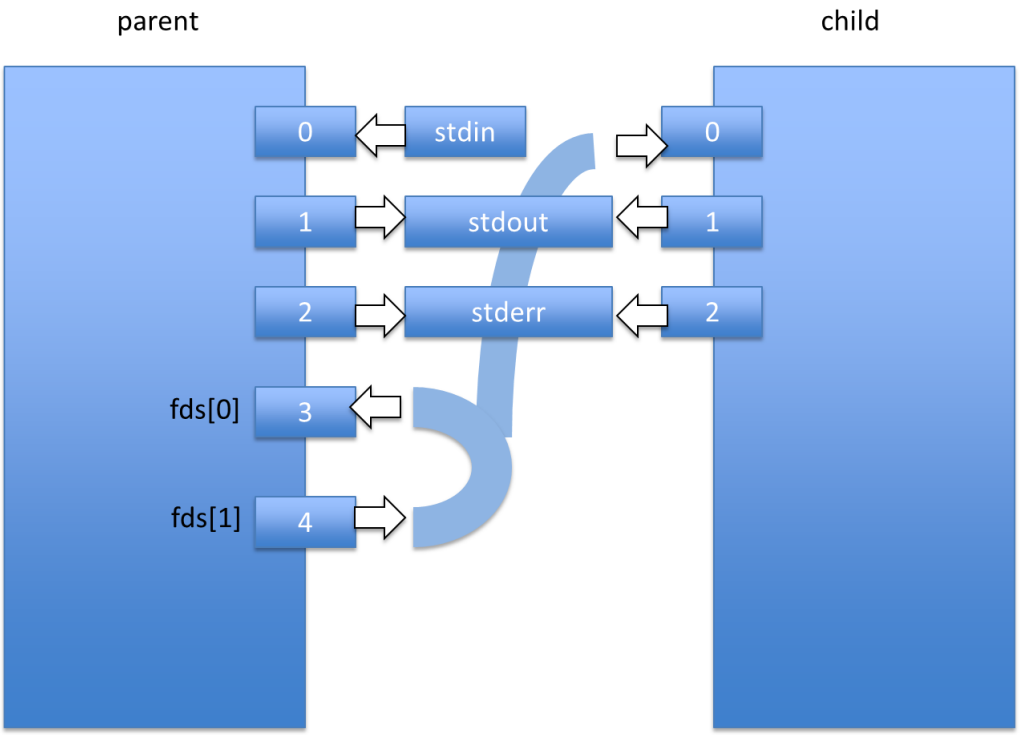
が成功すれば次のようになる.



次いで

```
if(close(fds[1]) < 0) {
    die ("I cannot close unnecessary pipe");
}
```

の実行により以下の図のようになる.

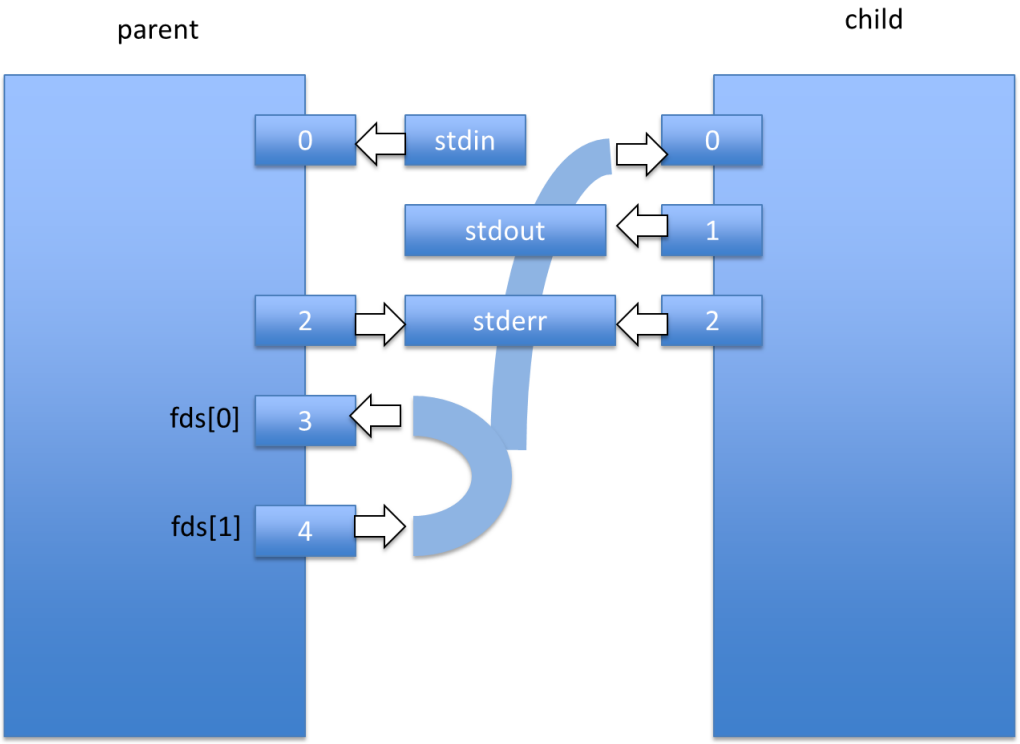


これにより子プロセスは親プロセスからのパイプのストリームを標準入力から読み取れるようになる．またこれ以降子プロセスは exec システムコールにより grep コマンド (プログラム) のプロセスに変身し，grep の振る舞いをするようになる．

ついで親プロセス側についてみていく．基本的には同様である．

```
if(close(1) < 0) {
    die ("I cannot close stdout");
}
```

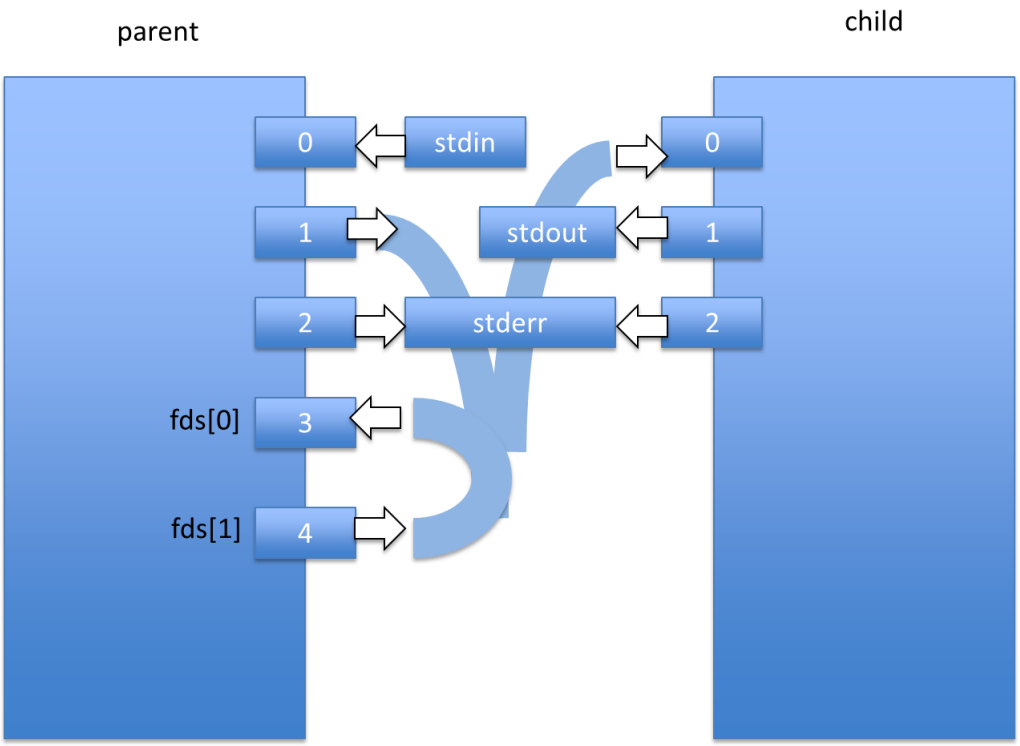
の実行により親プロセスの標準出力が一旦閉じられ，次の図のようになる．



次いで

```
if(dup2(fds[1], 1) < 0) {
    die ("I cannot duplicate the pipe");
}
```

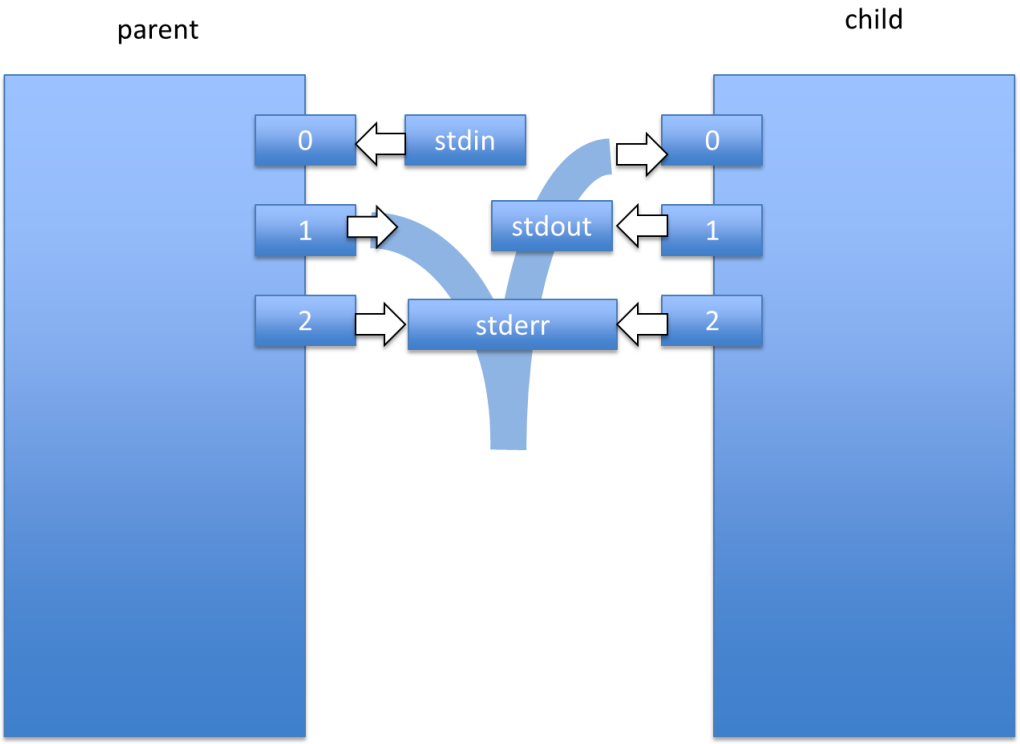
の実行が成功すると次の図のように再びファイルディスクリプタ 1 が有効となり，今度はパイプのストリームにつながる．



図ではファイルディスクリプタ 3 につながっているパイプの端が 1 にコピーされたかのような誤解をまねくかもしれないが，実際は `fds[1]` すなわちファイルディスクリプタ 4 につながっているパイプの端がコピーされている (データの流れの矢印の向きに注意)．また，くどいようだが，パイプの実体は 1 つである．

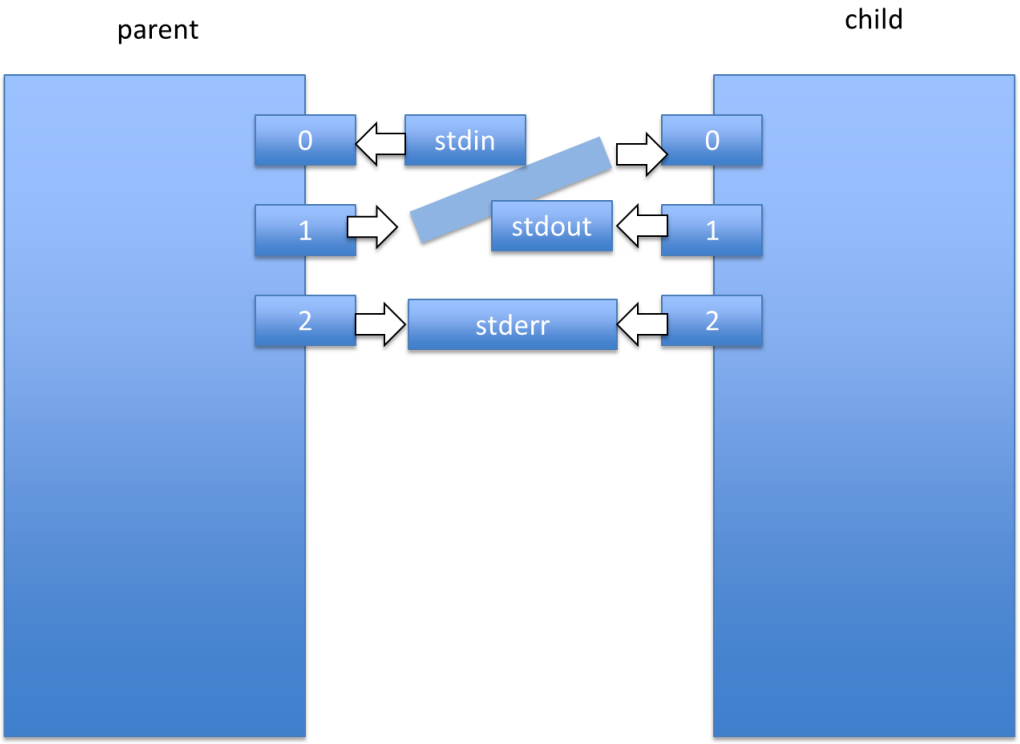
```
if(close(fds[0]) < 0) {
    die ("I cannot close unnecessary pipe");
}
if(close(fds[1]) < 0) {
    die ("I cannot close unnecessary pipe");
}
```

の 2 つを実行したあとは次の図のようになる．



この時点で親プロセスの標準出力がパイプを通じて子プロセスの標準入力につながっていることに注意.

もう少しパイプを整理して書くと次の図のようになる.



これ以降親プロセス側は exec システムコールにより ls -lR コマンドの振る舞いとなる．

これで全体として

```
ls -lR . | grep file
```

という処理と同等の処理になる．

演習 4

ここまでのプログラムを実行せよ．

注意: dup2 のシステムコールは第 2 引数がクローズされていなくても実際にクローズ処理をしたのち，duplicate を行う．したがって，上記の処理の dup2 の直前の close は実際には不要である．可読性のため，入れている．

問 3

上記と同じく `ls -lR . | grep file` の振る舞いをするプログラムを今度は親プロセス，子プロセスに割り当てるコマンドを「入れかえて」作成せよ．パイプの配管をうまく変えないと同じ振る舞いは期待できないはずである．

2 つ目の例として bc を子プロセスとして起動し，ユーザが cos, sin, tan を標準入力で指定すると，指定した関数の関数表を表示するプログラムを作る．

以下がプログラム例である．

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

```c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <float.h>
#include <sys/wait.h>

void die (const char *s) {
   perror(s);
   exit(1);
}


#define IN 0
#define OUT 1

int main(int argc, char *argv[]) {
    int pid;
    int c2p[2], p2c[2];

    if (pipe(c2p) < 0 ){
        die("I cannot make a pipe");
    }
    // assert c2p[IN]==3 && c2p[OUT]==4;
    // c2p[IN] for read and c2p[OUT] for write

    if (pipe(p2c) < 0 ){
        die("I cannot make a pipe");
    }
    // assert p2c[0]==5 && p2c[1]==6;
    // p2c[0] for read and fdsFom[1] for write

    if ((pid = fork()) < 0) {
        die("fork was failed");
    }
    if (pid == 0 ) { // child's side
        if(close(IN) < 0) {
          die ("I cannot close stdin");
        }
        if(dup2(p2c[IN], 0) < 0) {
```

```
            die ("I cannot duplicate the pipe");
        }
        if(close(p2c[IN]) < 0) {
            die ("I cannot close unnecessary pipe");
        }
        if(close(p2c[OUT]) < 0) {
            die ("I cannot close unnecessary pipe");
        }
        //
        if(close(OUT) < 0) {
            die ("I cannot close stdout");
        }
        if(dup2(c2p[OUT], 1) < 0) {
            die ("I cannot duplicate the pipe");
        }
        if(close(c2p[OUT]) < 0) {
            die ("I cannot close unnecessary pipe");
        }
        if(close(c2p[IN]) < 0) {
            die ("I cannot close unnecessary pipe");
        }
        execl("/usr/bin/bc", "bc", "-lq", NULL);
        die("exec was failed");
    } else {    // parent's side
        if(close(c2p[OUT]) < 0) {
            die ("I cannot close unnecessary pipe");
        }
        if(close(p2c[IN]) < 0) {
            die ("I cannot close unnecessary pipe");
        }
        // we use c2p[IN] and p2c[OUT] as read and write,
        // respectively.

#define BUFFSIZE 256
#define TRUE 1

        char buf[BUFFSIZE];

        FILE *in = fdopen(c2p[IN],"r");
```

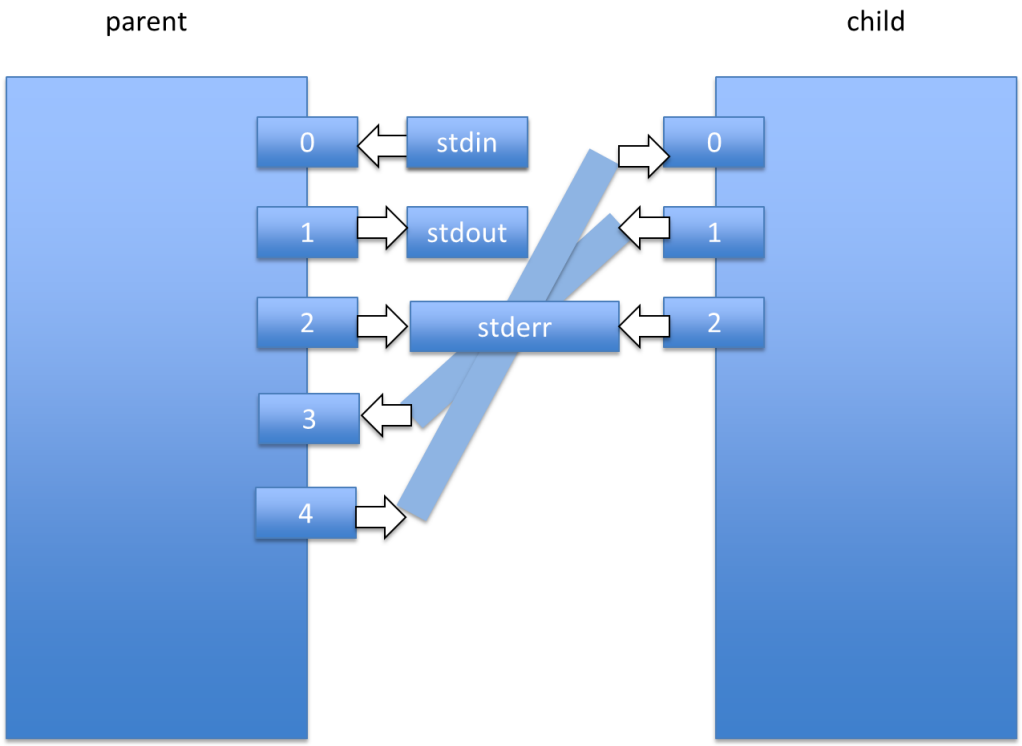
```
    FILE *out = fdopen(p2c[OUT], "w");

/* for bc function definition */
    fprintf(out, "define tan(x) {\n");
    fprintf(out, "return (s(x) / c(x))}\n");
    fprintf(out, "define sin(x) {\n");
    fprintf(out, "return (s(x))}\n");
    fprintf(out, "define cos(x) {\n");
    fprintf(out, "return (c(x))}\n");

    while (TRUE) {
        if (fgets(buf, BUFFSIZE, stdin) == NULL) {
            exit(1);
        }
        char op[5];
        if (strncmp(buf, "sin", 3) == 0) {
            strncpy(op, buf, 3);
            op[3]='\0'; // force null terminate
        } else if (strncmp(buf, "cos", 3) == 0) {
            strncpy(op, buf, 3);
            op[3]='\0'; // force null terminate
        } else if (strncmp(buf, "tan", 3) == 0) {
            strncpy(op, buf, 3);
            op[3]='\0'; // force null terminate
        } else {
            break;
        }
        int i;
        for (i = 1; i < 100; i++) {
            double f=((double)i)/100.0 * 3.14159265359;
            fprintf(out, "%s(%f)\n", op, f);
    //      fprintf(stdout,"%s(%f)\n", op, f);
            fflush(out);
            fgets(buf, BUFFSIZE, in);
            fprintf(stdout, "%s(%f) = %s", op, f, buf);
        }
    }
    fprintf(out, "quit\n");
    fflush(out);
```

```
        int status;
        waitpid(pid, &status, 0);
        exit(0);
    }
}
```

入力と出力の 2 箇所で「パイプ工事」を行うためパイプを p2c, c2p の 2 つ用意している．配管後のイメージを再掲すると以下のとおりである．



親プロセスのほうでは「配管工事」の終了後，

```
FILE *in = fdopen(c2p[IN],"r");
FILE *out = fdopen(p2c[OUT], "w");
```

によりファイルディスクリプタから FILE 構造体へのポインタを得ている．これは FILE を用いた標準入出力関数を (利便性のため) 以降で使いたいからであり，本質的な意味はない．

これ以降の処理は大きな while ループの中に for ループがある構造である．for ループは関数テーブルの作成で各エントリーごとの処理がボディにかかれている．ここで子プロセスの bc へ命令を送りつづけている．通常の fprintf はバッファ機能がついているため，必ずしも 1 行ごとデータが送られるわけではない．そこでバッファーの強制送り出し (バッファ詰まりの解消) のために fflush を呼び出している．

一方 while ループは無限ループとなっており，ユーザの入力にしたがって，bc への命令 (関数名) を変えるようになっている．規定以外の入力に対しては while ループを抜け出し，終了の処理へと向かう．

演習 5
実際に動かしてみよ．

演習 6
(sin, cos, tan 以外に*)* 使える関数を増やしてみよ．

ここまで紹介したパイプは名前無しパイプと呼ばれるものである．
名前無しパイプはこれらの例のように原則親子関係にあるプロセス間でしか共有できない．
一般の任意のプロセス間でパイプを共有するための機構として名前つきパイプが存在する．

問 4
名前つきパイプを調べよ．

その他にもソケットという機構もある．パイプが一方向であるのに対し，ソケットは 1 つのストリームで双方向通信ができる (これはファイルを読み書き両用でオープンするのとは異なり，相手とこちらとの純粋な双方向の通信である)．ソケットはネットワーク通信の基礎手続きとして重要である．

参考文献

[1] 青木峰郎:ふつうの Linux プログラミング Linux の仕組みから学べる gcc プログラミングの王道, ソフトバンククリエイティブ, ISBN-13: 978-4797328356, 2005.