14 UNIX プロセス間通信 (1) —パイプ—

この節では、UNIX で最も手軽なプロセス間通信の手段となっているパイプ(pipe)について説明する。

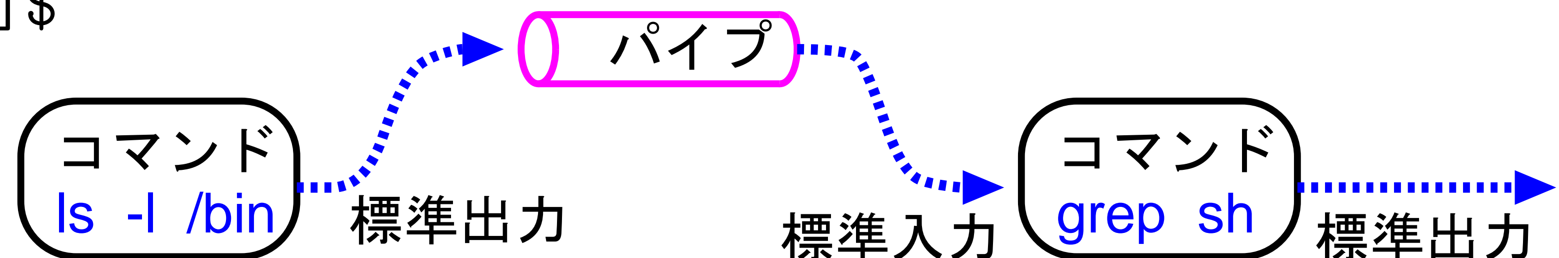
14–1 コマンドインタープリタにおけるパイプ機能の利用

tcsh や bash といったコマンドインタープリタ上においても、パイプを用いたプロセス間通信の機能を利用することがある。

例14. 1 (コマンドライン上でのパイプ機能の利用)

UNIXにおいては、1つのコマンドの標準出力がそのまま別のコマンドの標準入力に向かうことを指定して、2つのコマンドを並行して実行させることが出来る。

```
[motoki@x205b]$  ls -l /bin | grep sh
-rwxr-xr-x 1 root root  60592 Feb  4 2000 ash*
-rwxr-xr-x 1 root root 263064 Feb  4 2000 ash.static*
-rwxr-xr-x 1 root root 334960 Feb 16 2001 bash*
-rwxr-xr-x 1 root root 492720 Sep  8 2000 bash2*
lrwxrwxrwx 1 root root      3 Jan 22 2002 bsh -> ash*
lrwxrwxrwx 1 root root      4 Jan 22 2002 csh -> tcsh*
         . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
[motoki@x205b]$
```



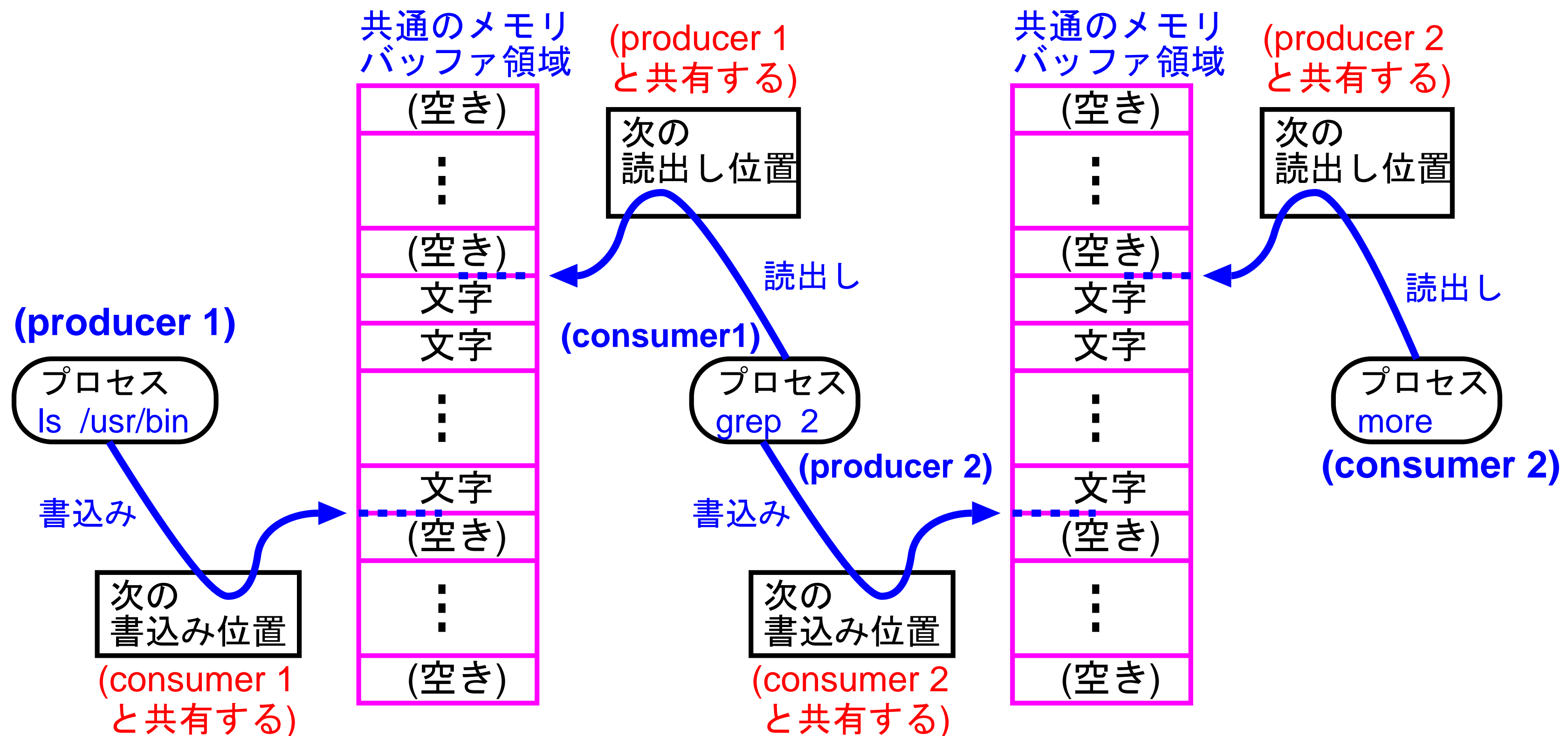
例14. 2 (3つのコマンドをパイプで繋げる)

3つ以上のコマンドをパイプで繋げることも可能である。

```
[motoki@x205b]$  ls /usr/bin | grep 2 | more
a2p*
addr2line*
afm2tfm*
   ......
gif2ps*
   ......
latex2html*
   ......
pdf2ps*
   ......
ps2epsi*
ps2pdf*
   ......
wx2_conv
[motoki@x205b]$
```

パイプ処理の中味：

パイプの実体は2つのプロセスの間で共有するメモリバッファである。



- メモリバッファの共有の仕方は、生産者と消費者の問題(13.2.2節) における producer と consumer の間のデータ領域の共有の仕方と同じである。

- パイプの両端のプロセスはOSの監視の下で並行に実行される。

(⟹ 高性能化)

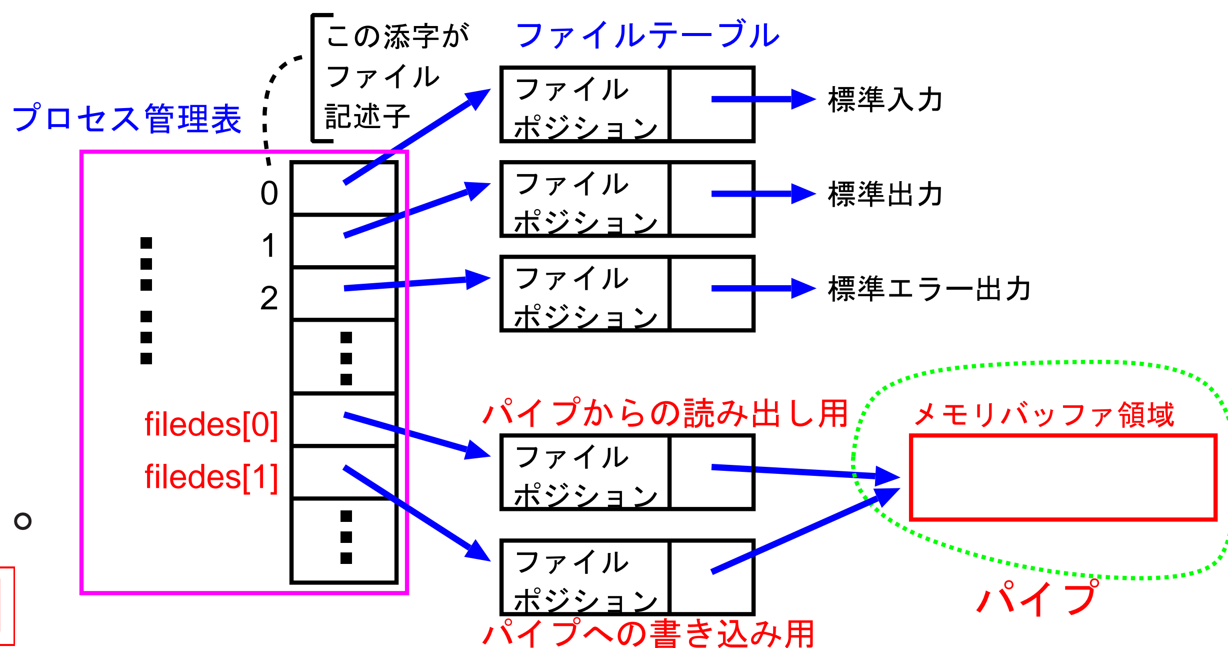
14–2 pipe( ) システムコール

UNIXにおいては、システムコール pipe( ) を用いてパイプを生成し、このパイプを用いて親子プロセス間, 子プロセス同士でプロセス間通信が出来るようになっている。

パイプ生成のシステムコール int pipe(int filedes[2]) :

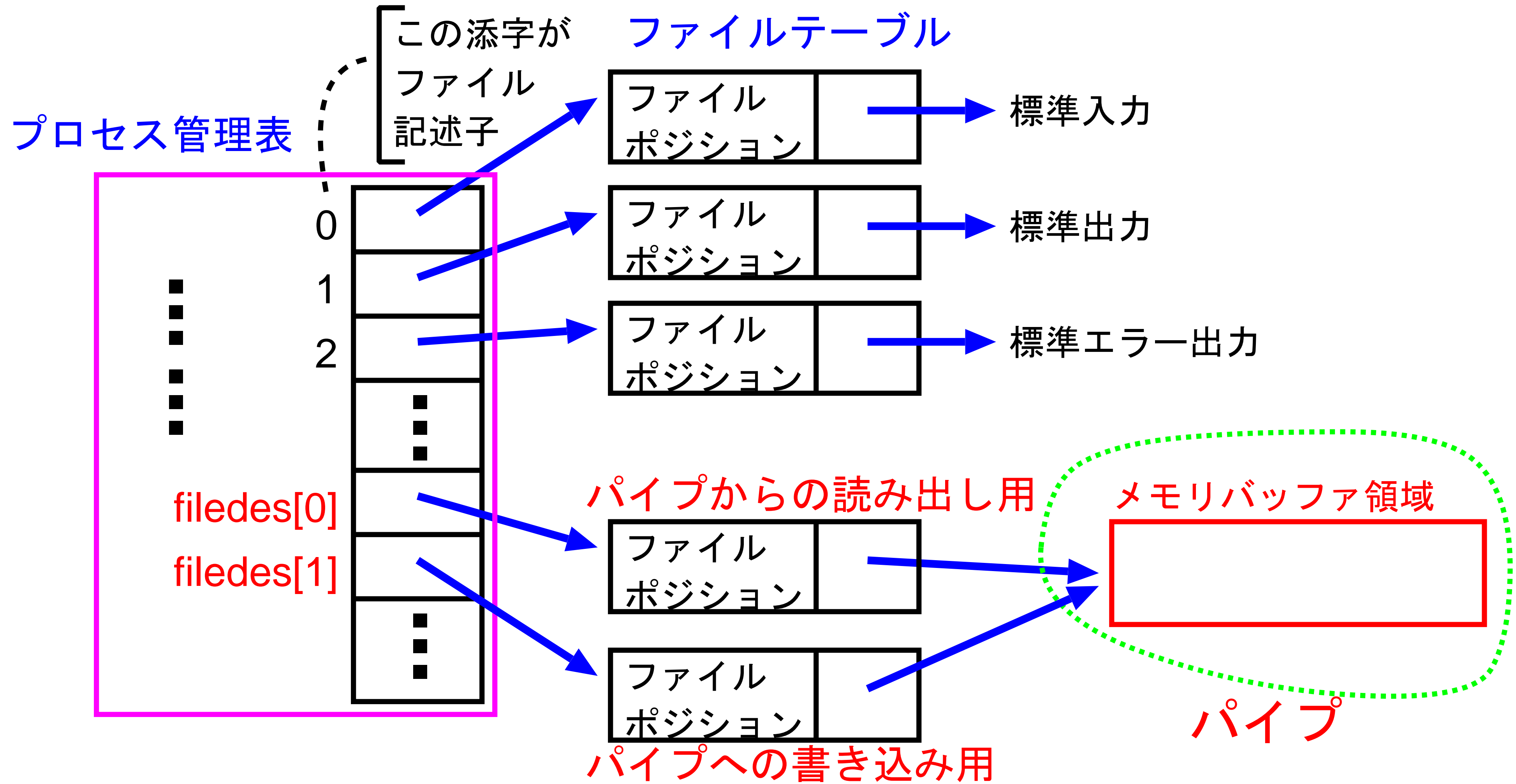
- ヘッダファイル<unistd.h> を include する。
- 新しくパイプが作られ、その読み出し端(i.e. パイプからのデータの出口)のファイル記述子が filedes[0] に、書き込み端のファイル記述子が filedes[1] に格納される。

次ページに拡大図



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

- 呼ばれると、新しくパイプが作られ、...



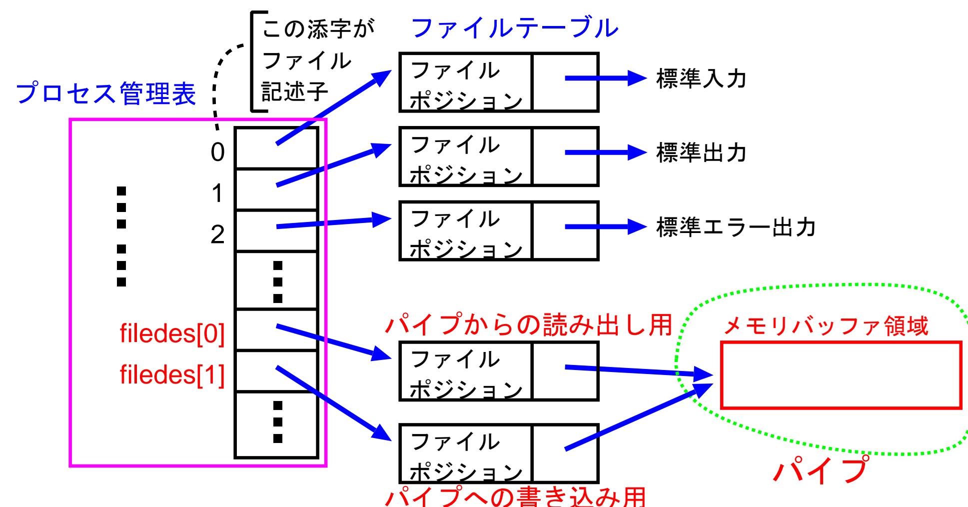
- ファイル記述子が出来てしまえばそれ以降の入出力の仕方は通常のファイルと同じである。

  write( ) でパイプにデータを書き込んだり、

  read( ) でパイプからデータを読み出す

  ことが出来る。

  但し、
  パイプには容量制限があって、
  容量を越えてデータを書き込もうとすると、そのwrite( )
  システムコールはブロックされる。



- パイプの読み出し端, 書き込み端を閉じるにはclose( ) システムコールを使って、各々

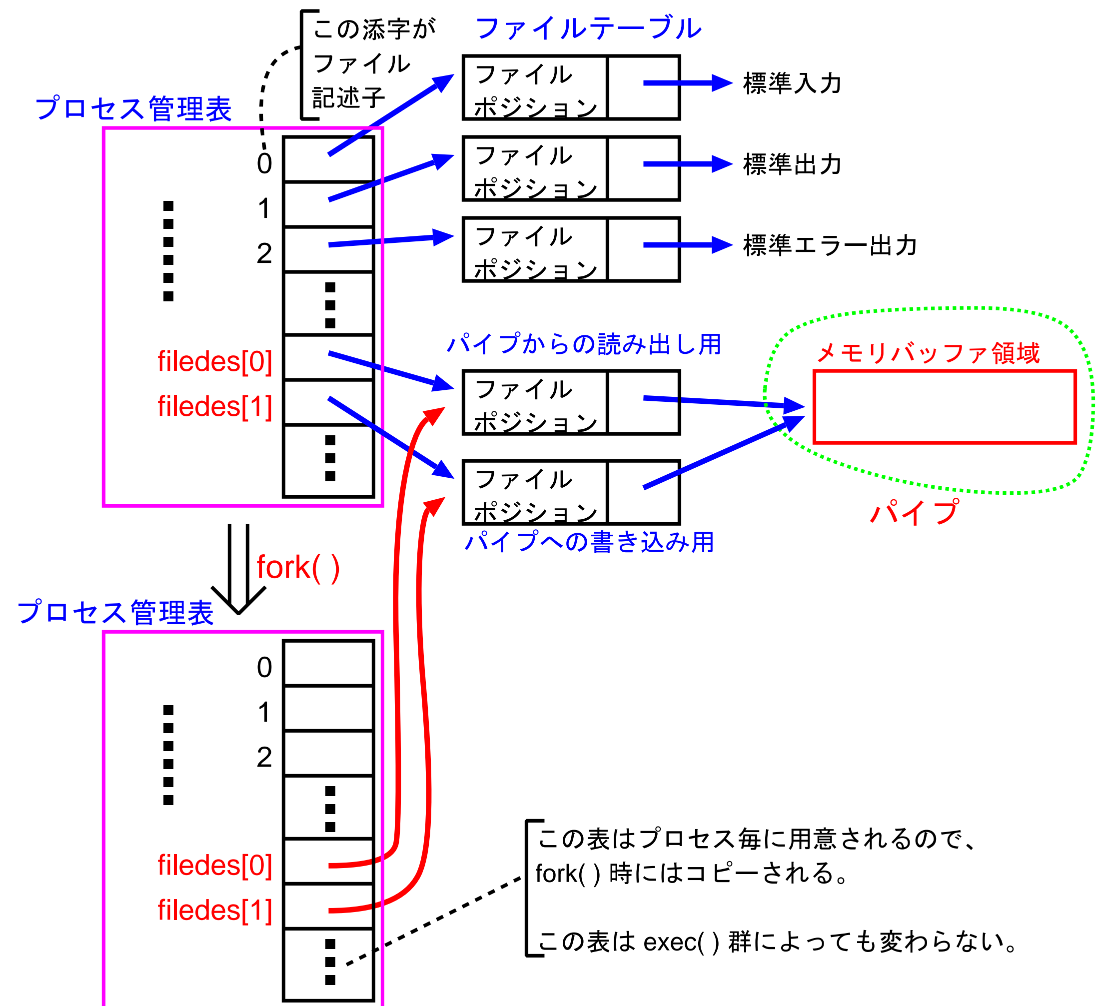
```
close(filedes[0]),
close(filedes[1])
```

  とすればよい。

- fork( ) システムコールによって生成された子プロセスは親プロセスのファイル記述子も全て受け継ぐ。

  さらに、exec( ) システムコール群を使って別のプログラムを overlay した場合も、新しいプログラムは元々あったファイル記述子を引き継ぐ。

⇨ このパイプを使って
親子プロセス間,
子プロセス同士
で通信が可能になる。



———

補足：

exec( ) システムコール群を使って別のプログラムをoverlayする場合、exec( ) 時にファイル記述子 (番号) を引き継ぎ新しいプログラムはそれをファイル記述子として認識するなどの手立てを取らないと、実際には新しいプログラムは標準入出力以外のファイル記述子を使うことは出来ない。

(引き継ぎの例としては山口 (監)「The UNIX Super Text 下」(技術評論社,1992)57.3 節を参照。)

例14. 3 (双方向パイプを使って親子のプロセス間で会話をする)

fork() した後、

① 最初に親プロセスが処理を進め、ひと纏まりの処理が終ったら、それを子プロセスにパイプで知らせ子プロセスからの連絡を待つ。

② 子プロセスは、親プロセスからの連絡を受けて処理を開始する。そして、ひと纏まりの処理が終ったら、それを親プロセスにパイプで知らせ親プロセスからの連絡を待つ。

③ 親プロセスは、子プロセスからの連絡を受けて処理を開始する。そして、ひと纏まりの処理が終ったら、それを子プロセスにパイプで知らせ子プロセスからの連絡を待つ。

④ …………

という風に、双方向パイプを使って親子のプロセス間で連絡を取りながら、親子で交互に処理を行っていくプログラムを次に示す。

```
[motoki@x205a]$  nl ipc-by-two-way-pipe.c
 1 /*********************************************/
 2 /* Operating-Systems/C-Programs/ipc-by-two-way-pipe.c   */
 3 /*-------------------------------------------*/
 4 /* 双方向パイプを使って                                 */
 5 /* 親プロセスと子プロセスの間で
                  コミュニケーションする例 */
 6 /*********************************************/
 7 #include <stdio.h>
 8 #include <stdlib.h>     /* for exit() library function */
 9 #include <unistd.h>     /* for pipe() and fork() system cal
10 #include <sys/types.h>  /* for wait() system call */
11 #include <sys/wait.h>   /* for wait() system call */

12 #define BUFSIZE  256

13 int main(void)
```

```
14 {
15   int k, insize, status,
16       pipefd_p2c[2],  /*file descripter of pipe from parent
17       pipefd_c2p[2];  /*file descripter of pipe from child
18   pid_t childpid;
19   char buf[BUFSIZE];
20
21   if ((pipe(pipefd_p2c))<0 || (pipe(pipefd_c2p))<0) {
22     perror("pipe");
23     exit(EXIT_FAILURE);
24   }

25   if ((childpid=fork())==-1) {
26     perror("can't fork");
27     exit(EXIT_FAILURE);
28   }else if (childpid==0) {  /*子プロセス*/
29     close(pipefd_p2c[1]); /*parent-->child の書き込み端*/
```

```
30      close(pipefd_c2p[0]); /*child-->parent の読み出し端*/
31      for (k=0; k<6; k++) {
                                  親プロセスからのメッセージの読み込み
32        for (insize=0; insize<BUFSIZE; insize++) {
33          read(pipefd_p2c[0], buf+insize,1);
34          if (buf[insize]=='\n')
35            break;          メッセージ終了→処理の切替え
36        }
37        printf("(child) It's my %d-th turn to process.\n",
                k);
38        sleep(3-k%3);    /* 子プロセスの処理の代わり */
39        write(pipefd_c2p[1], "It's your turn to process.\n",
               27);
40      }
41      exit(EXIT_SUCCESS);
42    }else {                           /*親プロセス*/
43      close(pipefd_p2c[0]); /*parent-->child の読み出し端*/
```

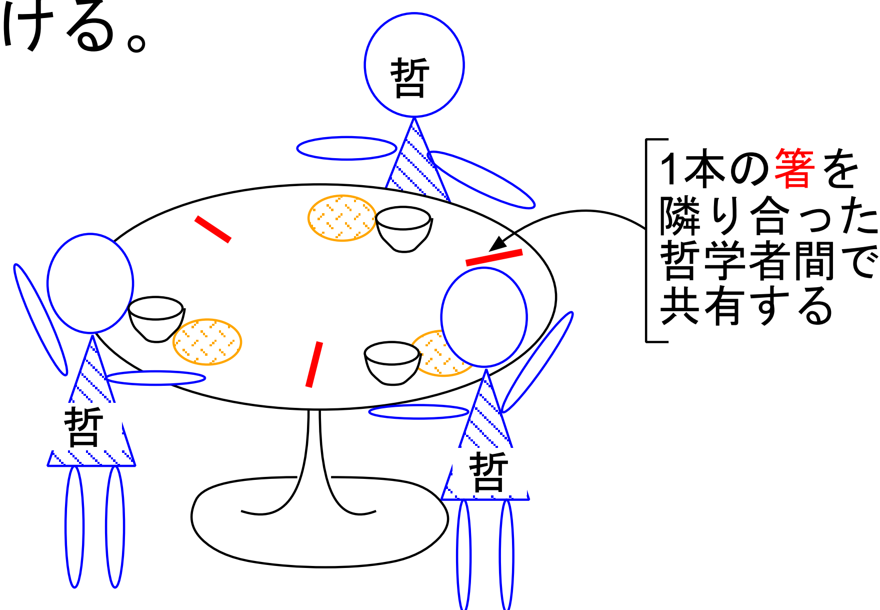
```
44      close(pipefd_c2p[1]); /*child-->parent の書き込み端*/
45      for (k=0; k<6; k++) {
46        printf("<PARENT> It's my %d-th turn to process.\n",
                 k);
47        sleep(k%2+1);    /* 親プロセスの処理の代わり */
48        write(pipefd_p2c[1], "It's your turn to process.\n",
                27);
                                 子プロセスからのメッセージの読み込み
49        for (insize=0; insize<BUFSIZE; insize++) {
50          read(pipefd_c2p[0], buf+insize,1);
51          if (buf[insize]=='\n')
52            break;             メッセージ終了→処理の切替え
53        }
54      }
55      wait(&status);
56      exit(EXIT_SUCCESS);
57    }
```

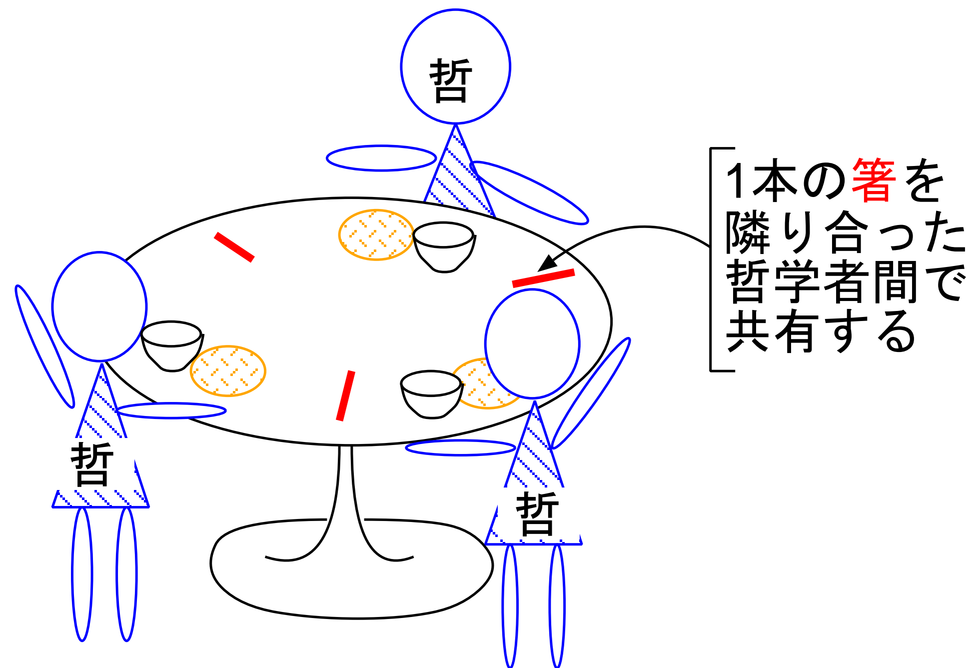
```
58 }
[motoki@x205a]$  gcc ipc-by-two-way-pipe.c
[motoki@x205a]$  ./a.out
<PARENT> It's my 0-th turn to process.
(child) It's my 0-th turn to process.
<PARENT> It's my 1-th turn to process.
(child) It's my 1-th turn to process.
<PARENT> It's my 2-th turn to process.
(child) It's my 2-th turn to process.
<PARENT> It's my 3-th turn to process.
(child) It's my 3-th turn to process.
<PARENT> It's my 4-th turn to process.
(child) It's my 4-th turn to process.
<PARENT> It's my 5-th turn to process.
(child) It's my 5-th turn to process.
[motoki@x205a]$
```

例14. 4 (食事する哲学者達)　**5人の食事する哲学者の問題**は、資源を共有する並行プロセスの間で**同期をとる代表的な問題の1つ**である。

5人の哲学者達が円卓を囲んでいる。

- 各々の哲学者の前にはご飯を盛った茶碗が置かれ、隣り合った哲学者の間には御箸が1本づつ置かれている。
- 自分の前に置かれた**ご飯を食べるためには**、各哲学者は自分の両脇に置かれた御箸(共有資源)を2本とも確保しなければならない。
- 哲学者達は一旦ご飯を食べ始めてもしばらくすると御箸を元の位置に返して物思いにふける。





御箸の確保を1本ずつ無雑作に行っていたのでは、
各々の哲学者が
右側の御箸1本だけを確保して左側の御箸が空くのを待つ、
という状態に陥る可能性がある。

⇨ こういったデッドロック状態を回避し、さらに飢餓状態の哲学者が出るのも避けられる、哲学者達(並行プロセス)の御箸(資源)の共有の仕方(e.g. 確保・解放の仕方)が求められる。

これが元々の「5人の食事する哲学者の問題」である。

この例では、この問題を使ってデッドロックが実際にどの様に起こるのかを例示する。その際、

- 簡単のため、哲学者の人数を**3人**としてシミュレーションを行った。
- パイプを使ってセマフォアを表した。

> 実際、
>
> パイプは親子・兄弟のプロセス間で共有する。
>
> 1つのプロセスがパイプからデータを読み出すと残りのプロセスはそのデータに触れることは出来ない。それゆえ、
>
> パイプにデータが入っている ⟺ 共有資源が空いている
>
> と見て、パイプ内のデータを読めたプロセスが共有資源を使う権利を獲得したことにすれば、パイプを使ってセマフォアを表せたことになる。

- どの哲学者も、①自分のすぐ左の御箸を確保, ②考え事, ③自分のすぐ右の御箸を確保, ④一口食べる, ⑤自分のすぐ左の御箸を解放, ⑥自分のすぐ右の御箸を解放, ⑦考え事、という動作を繰り返すことにした。

各々の御箸が利用可能かどうかを表す3本のパイプと哲学者を表す3つの子プロセスを生成し、3人の哲学者(プロセス)が各々の両隣の御箸(共有資源のペア)を確保したり解放したりしながら食事を進める様子をシミュレートするプログラムを次に示す。

[motoki@x205a]$ nl dining-phil-deadlock-pipe.c

```
 1 /*********************************************
 2 /* Operating-Systems/C-Programs/dining-phil-deadlock-pipe.c
 3 /*---------------------------------------------
 4 /* 単一プロセッサの場合は、パイプを使ってセマフォアを表す    */
 5 /* こともできる。そこで、この手法を用いて排他制御を行うこと */
 6 /* にして、
 7 /*     不用意に排他制御を行うとDeadlockの状態に陥ること     */
 8 /* を有名な「Dining Philosophersの問題」で確かめる。        */
 9 /* A.ケリー&I.ポール「CのABC(下)」アジソンウェスレイ       */
10 /*                                    ジャパン,12.5節
11 /*********************************************
12 #include <stdio.h>
```

```
13 #include <stdlib.h>      /* for exit() library function */
14 #include <unistd.h>      /* for pipe(),fork() and sleep() */
15 #include <sys/types.h>   /* for wait() system call */
16 #include <sys/wait.h>    /* for wait() system call */

17 #define  N  3            /* 哲学者の人数 */
18 #define  Left_chopstick(x)  (Chopstick[x])
19 #define  Right_chopstick(x) (Chopstick[((x)+1) % N])

20 typedef struct {
21   int pfd[2];
22 }Semaphore;

23 Semaphore Chopstick[N];
                                              哲学者の番号
24 void simulate_behaviour_of_philosopher(int k);
25 void Print_an_event(int k, char *event);
```

```
26 void initialize_semaphore();
27 void P(Semaphore sem);
28 void V(Semaphore sem);

29 int main(void)
30 {
31   int k, status;

32   initialize_semaphore();     セマフォア(箸)を用意

33   for (k=0; k<N; k++)                                   見出しの出力
34     printf("Philosopher%2d         ", k);               |
35   printf("\n");                                         |
36   for (k=0; k<N; k++)                                   |
37     printf("--------------------  ");                    |
38   printf("\n");                                         |
39
```

```
40    for (k=0; k<N; k++) {
41      if (fork()==0) {  /*子プロセス(哲学者)*/
42        simulate_behaviour_of_philosopher(k);
43        exit(EXIT_SUCCESS);
44      }
45    }
46                                /*親プロセス */
47    for (k=0; k<N; k++)
48      wait(&status);
49    return 0;
50  }

51  /* k番目の哲学者の動作をシミュレートする*/
52  void simulate_behaviour_of_philosopher(int k)
53  {                                         哲学者の番号
54    int i;
```

```
55    for (i=0; i<5; i++) {
56      P(Left_chopstick(k));        左箸の確保
57      Print_an_event(k,"pick up left stick");
58      Print_an_event(k,"      ***thinking***");
59      sleep(1);
60      P(Right_chopstick(k));       右箸の確保
61      Print_an_event(k,"pick up right stick");
62      Print_an_event(k,"***eating***");
63      sleep(1);
64      Print_an_event(k,"put down left stick");
65      V(Left_chopstick(k));        左箸の解放
66      Print_an_event(k,"put down right stick");
67      V(Right_chopstick(k));       右箸の解放
68      Print_an_event(k,"      ***thinking***");
69      sleep(1);
70    }
71  }
```

```
72 void Print_an_event(int k, char *event)
73 {
74   int i, indentsize;

75   indentsize=22*k;
76   for (i=0; i<indentsize; i++)
77     putchar(' ');
78   printf("%s\n", event);
79 }

80 /* セマフォアの初期設定, PV 操作 */
81 void initialize_semaphore()
82 {
83   int i;

84   for (i=0; i<N; i++) {
```

```
85      pipe(Chopstick[i].pfd);
86      write(Chopstick[i].pfd[1], "x", 1);
87              /* パイプにデータが入っている */
88              /*  <==> 共有資源が空いている */
89    }
90 }

91 void P(Semaphore sem)
92 {
93   char token;

94   read(sem.pfd[0], &token, 1);
95 }

96 void V(Semaphore sem)
97 {
98   write(sem.pfd[1], "x", 1);
```

```
99 }
[motoki@x205a]$  gcc dining-phil-deadlock-pipe.c
[motoki@x205a]$  ./a.out
Philosopher 0          Philosopher 1          Philosopher 2
--------------------   --------------------   --------------------
pick up left stick
      ***thinking***
                       pick up left stick
                             ***thinking***
                                              pick up left stick
                                                    ***thinking***
         Ctrl-Cで強制終了
[motoki@x205a]$
```

実行結果を見るとすぐにデッドロック状態になっていることが分かる。

実際、

別に仮想端末を開いてその時のプロセスの状態を見ると次の様に全ての

プロセスが待ち状態(STATの欄がS)になっていた。

```
[motoki@x205a]$  ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TI
      ......(途中省略)......
motoki    22825  0.0  0.0   1636   356 pts/1    S+   17:15   0:
motoki    22826  0.0  0.0   1636   180 pts/1    S+   17:15   0:
motoki    22827  0.0  0.0   1636   184 pts/1    S+   17:15   0:
motoki    22828  0.0  0.0   1636   184 pts/1    S+   17:15   0:
      ......(以下省略)......
[motoki@x205a]$
```

演習(食事する哲学者達)　≈レポート課題

上の例14.4のプログラムを修正して、デッドロックが起こらないようにせよ。

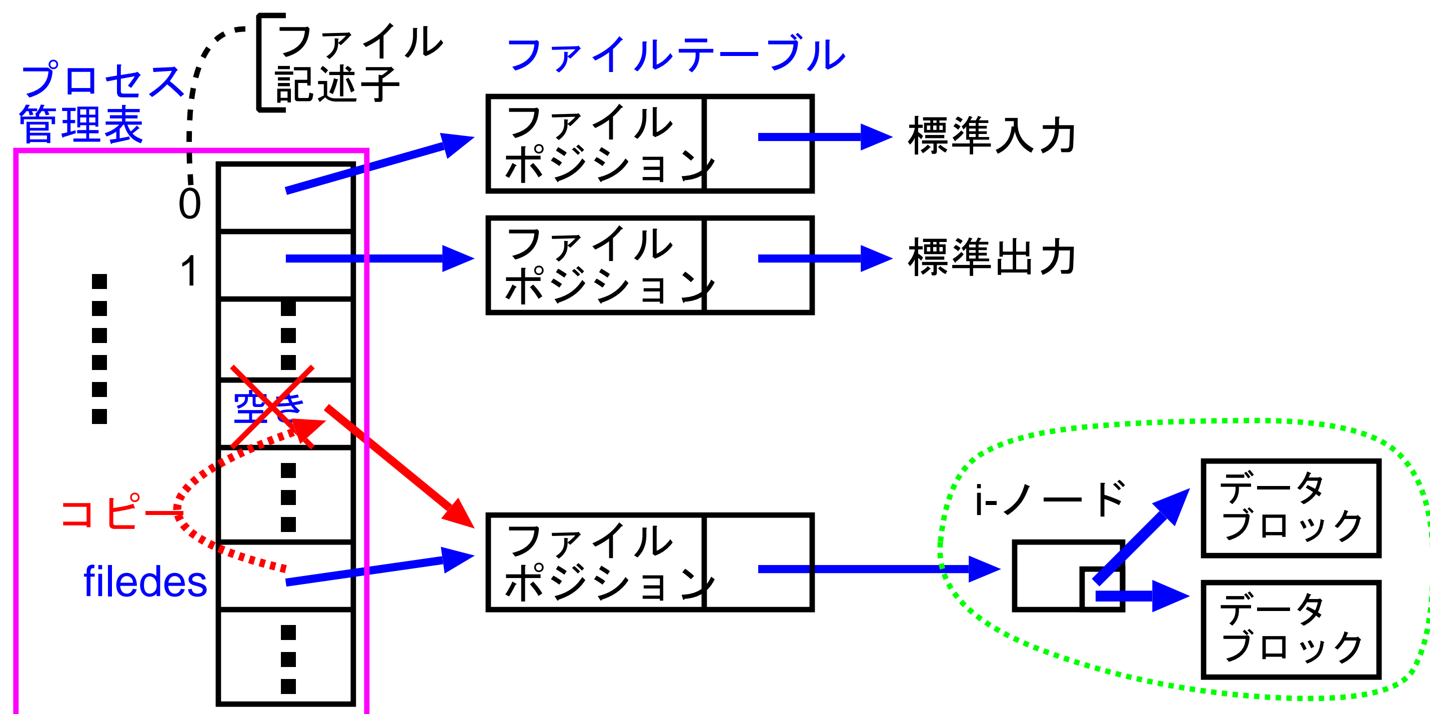
—

14–3 dup( ) システムコール

UNIXにおいては、システムコール dup( ) を用いて標準入出力のファイル記述子をパイプファイルと対応付けることが出来るようになっている。

ファイル記述子複製のシステムコール `int dup(int filedes)` :

- プロトタイプはヘッダファイル `<unistd.h>` の中で宣言されている。
- 引数で指定されたファイル記述子に対応するファイルテーブルへのポインタをその時点で空いている最小のファイル記述子とも結合させる。



p. 183からの引用：
exec( ) システムコール群を使って別のプログラムをoverlayする場合、exec( ) 時にファイル記述子(番号)を引き継ぎ新しいプログラムはそれをファイル記述子として認識するなどの手立てを取らないと、実際には新しいプログラムは標準入出力以外のファイル記述子を使うことは出来ない。

(引き継ぎの例としては山口(監)「The UNIX Super Text 下」(技術評論社,1992)57.3節を参照。)

- overlayの前に通信元プロセスのメッセージ送信のファイル記述子とパイプの入り口(書き込み端)を繋ぎ、通信先プロセスのメッセージ受信のファイル記述子とパイプの出口(読み出し端)を繋いでおくと、

  overlayするプログラムにファイル記述子を引き継ぐ

  という面倒な作業も不要になる。

  メッセージ発信や受信が標準入出力のファイル記述子を使って行われる場合には、この繋ぎ換えがdup( ) システムコールを使って可能になる。

入出力の標準化：

pipe( ) で割り当てられたファイル記述子をパイプの両端のプロセスに使ってもらおうとすると、メッセージの送受信に使うファイル記述子(番号)の部分が特殊になり、両端のプロセス自体の汎用性が全く無くなってしまう。

⇨ プロセスのメッセージ送受信を標準入出力を通じて行う様にすれば、パイプ両端のプロセスの組み合せの数だけ色々な処理を行うことが可能になる。
(実際、コマンドライン上でのパイプ処理はこういう考えで導入されている。)

コマンドライン上でのパイプ処理：

- 送信プロセスは標準出力にメッセージを出し、受信プロセスは標準入力からメッセージを受け取ると仮定する。
- コマンドライン上で programA | programB という処理を指定した場合、次の様に処理が進む。

(1) パイプを生成。

(2) fork() して子プロセスを2つ作る。(programA 用と programB 用の2つ。)

(3.1) programA 用の子プロセスでは、標準出力のファイル記述子とパイプの書き込み端を結合させる。

(3.2) programB 用の子プロセスでは、標準入力のファイル記述子とパイプの読み出し端を結合させる。

(4.1) programA 用の子プロセスでは、programA をoverlay し実行。

(4.2) programB 用の子プロセスでは、programB をoverlay し実行。

(5) 親プロセスは、並行して実行されている子プロセスが終了するのを待つ。

例14. 5 (cat ファイル | sort と同等の処理)

コマンド上での cat ファイル | sort というパイプ処理と同等の処理を行うプログラムを次に示す。

```
[motoki@x205a]$ nl connect-cat-and-sort-through-pipe.c
 1 /****************************************
 2 /* Operating-Systems/C-Programs/connect-cat-and-sort-throug
 3 /*--------------------------------------
 4 /* "cat ファイル|sort" と同等の処理を行う。*/
 5 /* 中西隆「ルーキーに贈る!UNIXプログラマ入門2.プログラミン...*/
 6 /*   基礎知識」(Software Design,1993年5月号,pp.13-28,技術...
 7 /****************************************
 8 #include <stdio.h>
 9 #include <stdlib.h>     /* for exit() library function */
10 #include <unistd.h>     /* for pipe(), fork(), dup() and ex
11 #include <sys/types.h>  /* for wait() system call */
12 #include <sys/wait.h>   /* for wait() system call */
```

```
13 int main(int argc, char *argv[])
14 {
15   int pipefd[2], status;
16   pid_t childpid;

17   if (pipe(pipefd) < 0) {
18     perror("pipe");
19     exit(EXIT_FAILURE);
20   }

21   if ((childpid=fork())==-1) {
22     perror("can't fork");
23     exit(EXIT_FAILURE);
24   }else if (childpid==0) { /*子プロセスcat*/
```

```
24    }else if (childpid==0) { /*子プロセスcat*/     図を参照
25      close(1);
26      dup(pipefd[1]);            標準出力とパイプの書き込み端を結合
27      close(pipefd[0]);          | 不要なファイル記述子を閉じる
28      close(pipefd[1]);          |
29      execl("/bin/cat", "cat", argv[1], NULL);
                                   ファイル名
30    }else if ((childpid=fork())==-1) {
31      perror("can't fork");
32      exit(EXIT_FAILURE);
33    }else if (childpid==0) { /*子プロセスsort*/
34      close(0);
35      dup(pipefd[0]);
36      close(pipefd[0]);
37      close(pipefd[1]);
38      execl("/bin/sort", "sort", NULL);
39    }else {                    /*親プロセス*/
```

```
39    }else {                                  /*親プロセス*/
40      close(pipefd[0]);
41      close(pipefd[1]);
42      wait(&status);
43      wait(&status);
44    }

45    return 0;
46  }
```

```
[motoki@x205a]$ gcc connect-cat-and-sort-through-pipe.c
[motoki@x205a]$ ./a.out connect-cat-and-sort-through-pipe.c
     (空行)
     (空行)
     (空行)
     (空行)
    close(0);
    close(1);
```

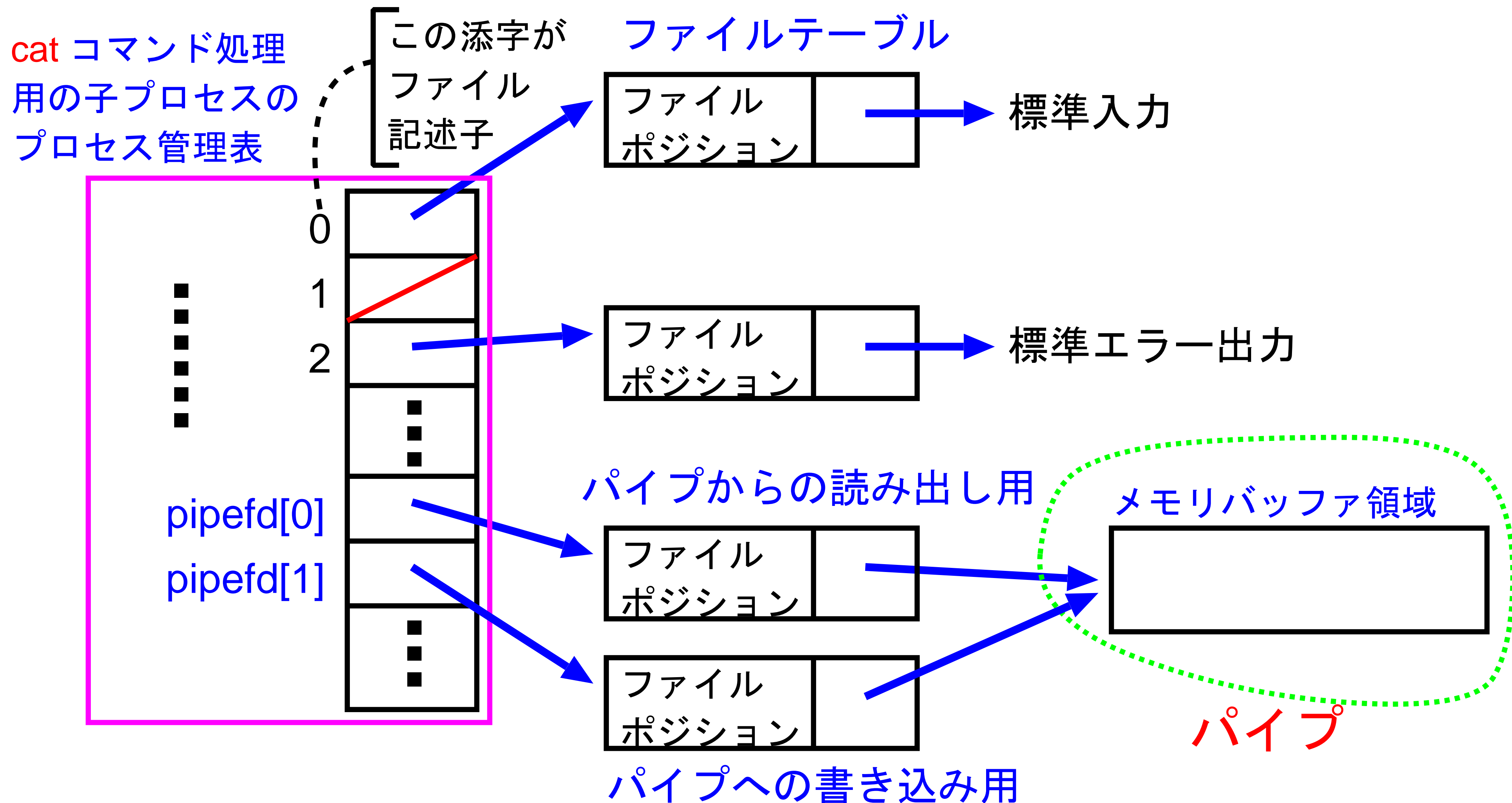
```
    close(pipefd[0]);
    close(pipefd[0]);
```

……(途中省略)……

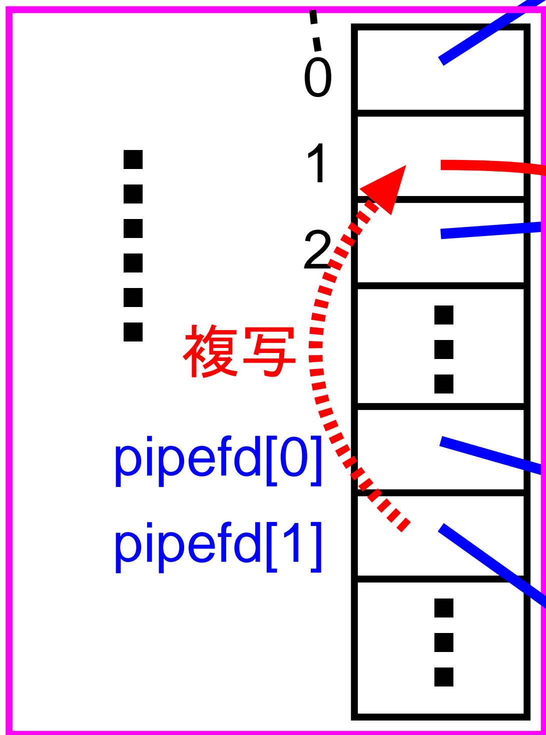
```
#include <stdlib.h>     /* for exit() library function */
#include <sys/types.h>  /* for wait() system call */
#include <sys/wait.h>   /* for wait() system call */
#include <unistd.h>     /* for pipe(), fork(), dup() and execl
/*   基礎知識」(Software Design, 1993年5月号, pp.13-28, 技術評 *
/* "cat ファイル | sort" と同等の処理を行う。
/* Operating-Systems/C-Programs/connect-cat-and-sort-through-p
/* 中西隆「ルーキーに贈る!UNIXプログラマ入門 2.プログラミング */
/**************************************************************
/**************************************************************
/*------------------------------------------------------------
int main(int argc, char *argv[])
```

```
{
}
[motoki@x205a]$
```

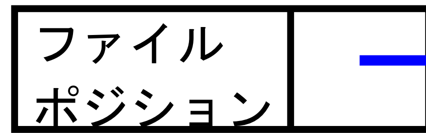
(25行目のclose()直後)



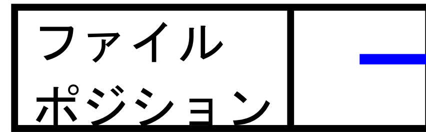
(26行目のdup( )直後)



ファイルテーブル

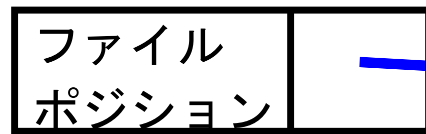


標準入力

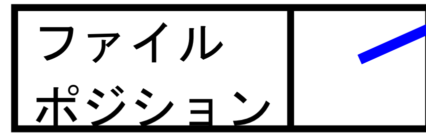


標準エラー出力

パイプからの読み出し用



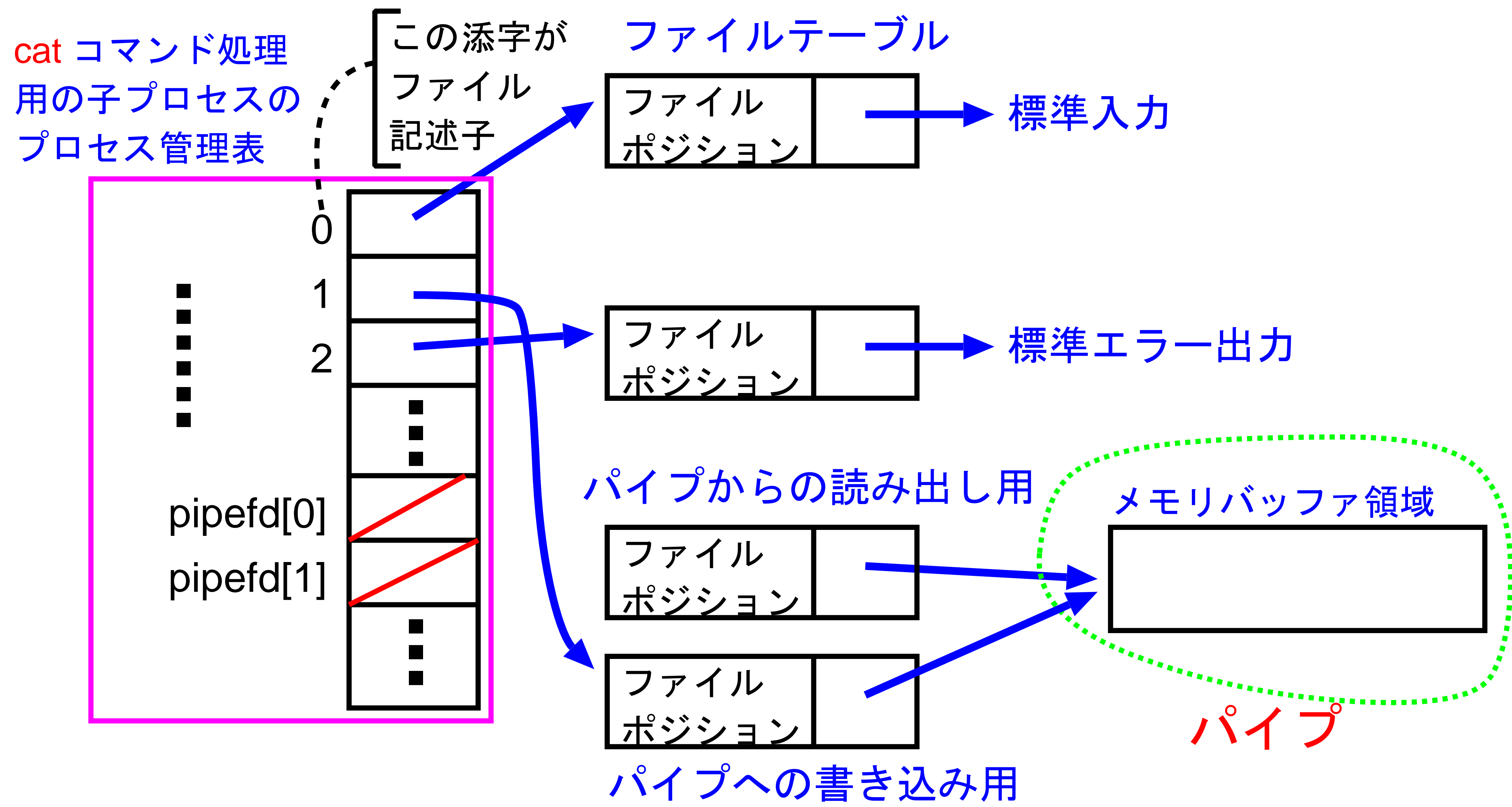
メモリバッファ領域



パイプ

パイプへの書き込み用

(28行目のclose()直後)



例 14. 6 (program1 | program2 と同等の処理)

先の例 14.5 のプログラムを、与えられた任意の 2 つのプログラムをパイプで繋げるように改造するのは簡単である。 次の通り。

```
[motoki@x205a]$  nl connect-2-programs-through-pipe.c
 1 /*************************************************
 2 /* Operating-Systems/C-Programs/connect-2-programs-through-
 3 /*-------------------------------------------------
 4 /* "program1|program2"と同等の処理を行う。
 5 /* 山口(監)「The UNIX Super Text 下」(技術評論社,1992)57.3.2*/
 6 /*************************************************
 7 #include <stdio.h>
 8 #include <stdlib.h>      /* for exit() library function */
 9 #include <unistd.h>      /* for pipe(), fork(), dup() and ex
10 #include <sys/types.h>   /* for wait() system call */
11 #include <sys/wait.h>    /* for wait() system call */

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

```
13 {
14   int pipefd[2], status;
15   char message[100];
16   pid_t childpid;

17   if (pipe(pipefd) < 0) {
18     perror("pipe");
19     exit(EXIT_FAILURE);
20   }

21   if ((childpid=fork())==-1) {
22     perror("can't fork");
23     exit(EXIT_FAILURE);
24   }else if (childpid==0) {  /* 子プロセス program1 */
```

```
24    }else if (childpid==0) {  /* 子プロセス program1 */
25      close(1);
26      dup(pipefd[1]);
27      close(pipefd[0]);
28      close(pipefd[1]);
29      execlp(argv[1], argv[1], NULL);
30      sprintf(message, "execlp(\"%s\",\"%s\",NULL)",
                argv[1], argv[1]);     execlpに失敗した時の処理
31      perror(message);
32      exit(EXIT_FAILURE);
33    }else if ((childpid=fork())==-1) {
34      perror("can't fork");
35      exit(EXIT_FAILURE);
36    }else if (childpid==0) {  /* 子プロセス program2 */
```

```
36    }else if (childpid==0) {  /* 子プロセス program2 */
37      close(0);
38      dup(pipefd[0]);
39      close(pipefd[0]);
40      close(pipefd[1]);
41      execlp(argv[2], argv[2], NULL);
42      sprintf(message, "execlp(\"%s\",\"%s\",NULL)",
                argv[2], argv[2]);        execlpに失敗した時の処理
43      perror(message);
44      exit(EXIT_FAILURE);
45    }else {                       /* 親プロセス */
46      close(pipefd[0]);
47      close(pipefd[1]);
48      wait(&status);
49      wait(&status);
50    }
```

```
51    return 0;
52 }
[motoki@x205a]$  gcc -o connect connect-2-programs-through-pip

[motoki@x205a]$  ./connect no no
execlp("no","no",NULL): そのようなファイルやディレクトリはありま
せん
execlp("no","no",NULL): そのようなファイルやディレクトリはありま
せん
[motoki@x205a]$  ./connect ls wc
     49      49    1216
[motoki@x205a]$  /bin/ls | /usr/bin/wc
     49      49    1216
[motoki@x205a]$
```

14–4 popen( ), pclose( ) ライブラリ関数

パイプを気軽に使うためのライブラリ関数も用意されています。

コマンド起動したプロセスと自プロセスをパイプで繋ぐ

ライブラリ関数　popen( ) :

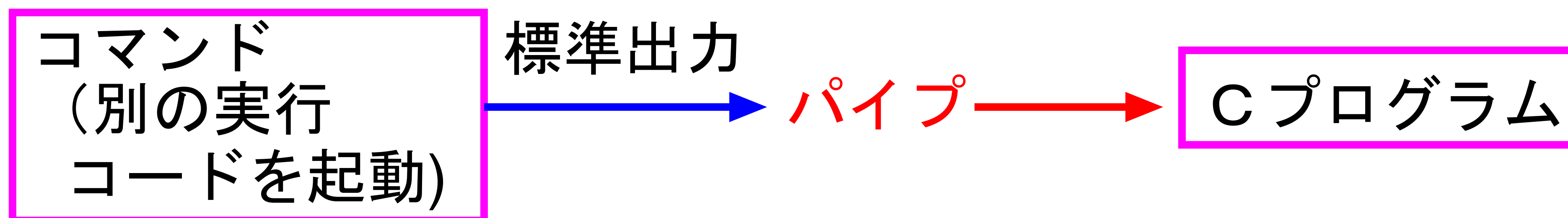
- 関数プロトタイプは

```
FILE *popen(char *command, char *type)
```

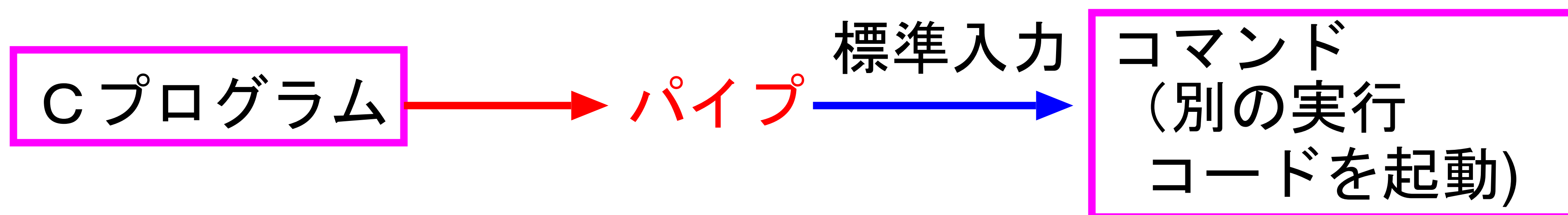
  で、ヘッダファイル <stdio.h> の中で宣言されている。

- 第1引数には別プロセス上で起動するプログラムを指定し、第2引数には "r" または "w" と指定する。

- popen(*command*, "r") が呼ばれると、
  第1引数で指定されたプログラムが別プロセス上で起動されその標準出力からの出力文字列を読み出すためのファイルポインタが関数値として返される。 あとはこのファイルポインタを指定して通常の入力を行うだけである。



- popen(*command*, "w") が呼ばれると 、
  第1引数で指定されたプログラムが別プロセス上で起動されその標準入力へ文字列を書き込むためのファイルポインタが関数値として返される。 あとはこのファイルポインタを指定して通常の出力を行うだけである。



パイプへのファイルポインタを閉じるライブラリ関数

`int pclose(FILE *stream) :`

- 関数プロトタイプはヘッダファイル `<stdio.h>` の中で宣言されている。
- 引数で指定されたパイプのストリームがクローズされる。
- 成功すると

  クローズしたパイプストリームと繋がったプロセスの

  終了状態値

  が返され、失敗すると -1 が返される。

例14. 7 (lsコマンドの出力を全て大文字に変換して表示)

lsコマンドの出力を全て大文字に変換して表示するプログラムは、popen( )
ライブラリ関数を使うと次の様に簡単に書ける。

```
[motoki@x205a]$  nl popen-ls-stream-to-toupper.c
 1 /****************************************************
 2 /* Operating-Systems/C-Programs/popen-ls-stream-to-toupper.
 3 /*---------------------------------------------------
 4 /* popen("ls", "r")でオープンしたストリームを大文字に変換し...
 5 /* A.ケリー&I.ポール「CのABC(下)」(アジソンウェスレイ
 6 /*                                    ジャパン/星雲社,1993)11.11節
 7 /****************************************************
 8 #include <stdio.h>
 9 #include <ctype.h>

10 int main(void)
11 {
```

```
12    char c;
13    FILE *pipestream;
14
15    pipestream = popen("ls", "r");
16    while ((c=getc(pipestream)) != EOF)
17      putchar(toupper(c));
18    pclose(pipestream);

19    return 0;
20 }
```

```
[motoki@x205a]$  gcc popen-ls-stream-to-toupper.c
[motoki@x205a]$  ./a.out
A.OUT
BUBBLESORT.C
CLOCK-BUBBLESORT-100
CLOCK-SORT-100.C
CONNECT
```

```
CONNECT-2-PROGRAMS-THROUGH-PIPE.C
CONNECT-CAT-AND-SORT-THROUGH-PIPE.C
DINING-PHIL-DEADLOCK-PIPE.C
DINING-PHIL-DEADLOCK-PIPE.C~
DINING-PHIL-DEADLOCK-SYSTEMV-SEM.C
```

……(途中省略)……

```
SHOW-LOGINNAME-OF-PROCESS.C
SHOW-REAL-USERNAME-OF-PROCESS.C
SHOW-REAL-USERNAME-OF-PROCESS2005.C
SHOW-RESOURCE-INFO-OF-PROCESS.C
SHOW-VARIOUS-IDS-ON-PROCESS.C
SHOW-VARIOUS-IDS-ON-PROCESS2005.C
[motoki@x205b]$
```

14–5 名前付きパイプ(FIFO)

血縁関係にないプロセス同士でこれまでに説明した「パイプ」と同等の処理を行うための、名前付きパイプ(または**FIFO**)と呼ばれる機構も用意されている。

これまでの「パイプ」がカーネル内に一時的に作られるのに対し、名前付きパイプはファイルシステム内に作られる。それゆえ、永続性もある。

ファイルの一種

名前付きパイプ生成のシステムコール

`int mkfifo(char *path, mode_t mode) :`

- `<sys/types.h>` と`<sys/stat.h>` を必要とする。
- 引数の指定に従って名前付きパイプ (**FIFO**) が生成される。
  成功すると **0** が返され、失敗すると **−1** が返される。
- 関数引数の `path` は、生成する名前付きパイプをフルパスで指定した文字列を表す。
- 関数引数の `mode` は、名前付きパイプの保護モードを表す。例えば自分には読み書きを許し、同じーグループのユーザと他人には読み出しだけ許す場合は、8進表示で `0644` と指定すればよい。

名前付きパイプが出来てしまえば

それ以降の入出力の仕方は通常のファイルの場合と同じである。最初に

(1) open( ) した後、

(2) write( ) を繰り返すか、
read( ) を繰り返すかし、最後に

(3) close( ) する

だけである。

ただ、いくつかの注意点がある。

名前付きパイプをオープンする際の注意：

- 複数のプロセスが読み出し専用に同時に同一の名前付きパイプをオープンすることができる。
- 複数のプロセスが書き込み専用に同時に同一の名前付きパイプをオープンすることができる。
- 読み出し専用に　open(path, O_RDONLY)　とした場合、他のプロセスが書き込み専用にオープンするまで待たされる。
- 書き込み専用に　open(path, O_WRONLY)　とした場合、他のプロセスが読み出し専用にオープンするまで待たされる。
- 読み込み専用に　open(path, O_RDONLY | O_NONBLOCK)　とした場合、すぐに呼出しから返る。
- 書き込み専用に　open(path, O_WRONLY | O_NONBLOCK)　とした場合、名前付きパイプが読み込み用にオープンされてないとopen()の実行は失敗しerrnoにENXIOがセットされる。

名前付きパイプをクローズする際の注意：

- 名前付きパイプに書き込んでいた全てのプロセスがパイプをクローズした時点で、パイプには**EOF**が書き込まれる。

例14. 8 (名前付きパイプを使って血縁のないプロセス間で会話をする)

例14.3と同様のことを名前付きパイプを使って行うプログラムを次に示す。例14.3と違って、ここでは独立に起動した2つのプログラム間で会話をしながら交互に処理(簡単のためsleep())を進めている。

```
[motoki@x205a]$  pwd
/home/motoki/Operating-Systems/C-Programs
[motoki@x205a]$  nl ipc-by-fifo-programA.c
 1 /****************************************************
 2 /* Operating-Systems/C-Programs/ipc-by-fifo-programA.c
 3 /*----------------------------------------------------
 4 /* 双方向名前つきパイプを使って2つのプログラム間で
 5 /*  コミュニケーションする際のMaster側のプログラム
 6 /****************************************************
 7 #include <stdio.h>
 8 #include <stdlib.h>     /* for exit() library function */
 9 #include <sys/types.h>  /* for mkfifo() system call */
```

```
10 #include <sys/stat.h>    /* for mkfifo() system call */
11 #include <unistd.h>      /* for open(),read(),write() system
12 #include <fcntl.h>       /* for open() system call */

13 #define BUFSIZE  256

14 int main(void)
15 {
16    int k, insize, fifofd_m2s, fifofd_s2m;
17    mode_t mode=0666;
18    char buf[BUFSIZE+1];  //"+1"は45行目でbuf[insize]が
                            //メモリ確保外になるのを避けるため
```

Master to Slave

```
19    if (mkfifo("fifo_m2s",mode) == 0) {
20      printf("(program A) mkfifo(\"fifo_m2s\",0666)"
               " succeeded.\n");
21    }else {
```

```
22      printf("(program A) We assume that FIFO \"fifo_m2s\" "
23              "already exists, and continue processing.\n");
24    }
```

Slave to Master

```
25    if (mkfifo("fifo_s2m",mode) == 0) {
26      printf("(program A) mkfifo(\"fifo_s2m\",0666)"
              " succeeded.\n");
27    }else {
28      printf("(program A) We assume that FIFO \"fifo_s2m\" "
29              "already exists, and continue processing.\n");
30    }

31    if ((fifofd_m2s=open("fifo_m2s", O_WRONLY)) < 0
32        || (fifofd_s2m=open("fifo_s2m", O_RDONLY)) < 0) {
33      perror("open");
34      exit(EXIT_FAILURE);
35    }
```

```
36    for (k=0; k<6; k++) {
37      printf("<PROGRAM A: master> It's my %d-th turn"
                " to process.\n", k);
38      sleep(k%2+1);    /* Masterプロセスの処理の代わり */
39      write(fifofd_m2s,
               "It's your turn to process.\n", 27);
40      for (insize=0; insize<BUFSIZE; insize++) {
41        read(fifofd_s2m, buf+insize, 1);
42        if (buf[insize]=='\n')
43          break;
44      }
45      buf[insize]='\0';
46      printf("(program B says) %s\n", buf);
47    }

48    close(fifofd_m2s);
```

```
49   close(fifofd_s2m);
50   return 0;
51 }
[motoki@x205a]$  nl ipc-by-fifo-programB.c
 1 /*******************************************************
 2 /* Operating-Systems/C-Programs/ipc-by-fifo-programB.c
 3 /*-------------------------------------------------------
 4 /* 双方向名前つきパイプを使って2つのプログラム間で
 5 /*  コミュニケーションする際のSlave側のプログラム          *
 6 /*******************************************************
 7 #include <stdio.h>
 8 #include <stdlib.h>     /* for exit() library function */
 9 #include <sys/types.h>  /* for mkfifo() system call */
10 #include <sys/stat.h>   /* for mkfifo() system call */
11 #include <unistd.h>     /* for open(),read(),write() systen
12 #include <fcntl.h>      /* for open() system call */
```

```
13 #define BUFSIZE  256

14 int main(void)
15 {
16   int k, insize, fifofd_m2s, fifofd_s2m;
17   mode_t mode=0666;
18   char buf[BUFSIZE];

19   if ((fifofd_m2s=open("fifo_m2s", O_RDONLY)) < 0
20       || (fifofd_s2m=open("fifo_s2m", O_WRONLY)) < 0) {
21     perror("open");
22     exit(EXIT_FAILURE);
23   }

24   for (k=0; k<6; k++) {
25     for (insize=0; insize<BUFSIZE; insize++) {
26       read(fifofd_m2s, buf+insize, 1);
```

```
27          if (buf[insize]=='\n')
28             break;
29       }
30       printf("(program B: slave) It's my %d-th turn"
                " to process.\n", k);
31       sleep(3-k%3);   /* Slaveプロセスの処理の代わり */
32       write(fifofd_s2m, "It's your turn to process.\n", 27);
33     }

34     close(fifofd_m2s);
35     close(fifofd_s2m);
36     return 0;
37  }
```

```
[motoki@x205a]$ gcc -o ipc-by-fifo-programA
                                         ipc-by-fifo-programA.c
[motoki@x205a]$ gcc -o ipc-by-fifo-programB
                                         ipc-by-fifo-programB.c
```

```
[motoki@x205a]$ ./ipc-by-fifo-programA
(program A) mkfifo("fifo_m2s",0666) succeeded.
(program A) mkfifo("fifo_s2m",0666) succeeded.
```

名前付きパイプの別端がオープンされるのを待つ状態に入る

待ち状態になるので、
ここで、別の仮想端末上でもう一方のプログラムを起動すると、2つの仮想端末上で実行が並行して進む。

```
[motoki@x205a]$ pwd
/home/motoki/Operating-Systems/C-Programs
[motoki@x205a]$ ./ipc-by-fifo-programB
(program B: slave) It's my 0-th turn to process.
(program B: slave) It's my 1-th turn to process.
(program B: slave) It's my 2-th turn to process.
(program B: slave) It's my 3-th turn to process.
(program B: slave) It's my 4-th turn to process.
(program B: slave) It's my 5-th turn to process.
```

プロセスが待ち状態から実行可能状態に変わる

```
<PROGRAM A: master> It's my 0-th turn to process.
(program B says) It's your turn to process.
<PROGRAM A: master> It's my 1-th turn to process.
(program B says) It's your turn to process.
<PROGRAM A: master> It's my 2-th turn to process.
(program B says) It's your turn to process.
<PROGRAM A: master> It's my 3-th turn to process.
(program B says) It's your turn to process.
<PROGRAM A: master> It's my 4-th turn to process.
(program B says) It's your turn to process.
<PROGRAM A: master> It's my 5-th turn to process.
(program B says) It's your turn to process.
[motoki@x205a]$
```

ここで、

- プログラム実行後、ディレクトリ内には2つの名前付きパイプ fifo_m2s と fifo_s2m が残っている。

```
[motoki@x205a]$  ls
        ......(途中省略)......
dining-phil-nodeadlock-systemV-sem.c  out
fifo_m2s|  ファイルの一種              popen-ls-stream-to-tou
fifo_s2m|                             print-status-of-file.c
fork-and-run-concurrently.c           print-status-of-file20
        ......(以下省略)......
[motoki@x205a]$
```