ファイルディスクリプタ と擬似ファイル 副題: Unix的な考え方 (全てがファイル)

田浦健次朗

目次

Unix 的なもの

リダイレクト, パイプの仕組み

擬似ファイル

Unixの特徴(1) — 出力先の変更

- 「端末に出力するプログラム」がそのまま「ファイルに出力するプログラム」になる

```
1 int main() { printf("hi world\n"); }
```

- 普通に走らせると端末へ出力

```
1 $ ./hello
2 hi world
```

- 出力先を変更 (リダイレクト) する「だけ」でファイルに書ける

```
1 $ ./hello > hi
```

ファイルを読み書きするプログラムを書くのに, ファイルを開く必要がない

Unixの特徴(2) — 入力先の変更

- 同様に「端末から入力するプログラム」がそのまま「ファイルから入力するプログラム」になる

```
1  int main() {
2    int x;
3    scanf("%d", &x);
4    printf("%d\n", x + 1);
5    return 0;
6  }
```

- 端末から

```
1  $ ./exec/plus_1
2  3  # 入力
3  4  # 出力 (3 + 1)
```

- ファイルから

```
1  $ cat three
2  3
3  $ ./plus_1  < three
4  4
```

Unixの特徴(3) — パイプでプロセス間通信

- 「端末に出力するプログラム」がそのまま「プロセスにデータを送るプログラム」になる

```
1 $ echo 10 | ./plus_1
```

- plus_1 自身は同様に scanf を呼んでいるだけ
- よく使う実例

```
1 $ ps auxww | grep firefox
```

Unixの特徴

- 同一のプログラムで色々な対象 (端末, 普通のファイル, 別のプロセス) への入出力が可能
- そもそもプログラムは今, 読み書きしているものが, (普通の) ファイルであるかどうかすら意識せずに書ける
  - え? `printf` は端末に書く関数じゃないの?
  - 否. 「ファイルディスクリプタ 1 番」に書いている
  - それが何とつながっているかで出力先が変わる
- 複数のプログラムを容易に (|で) 組み合わせ可能
- ⇒ 一つのプログラム (コマンド) は単純に, それらを組み合わせて複雑な処理を簡単に

それらを可能にした大元の考え方は, プロセスの外部とのやり取りは全て「ファイルディスクリプタ」を経由して行われるという考え方

大元の思想 (多分こうだったんじゃないか劇場)

- プロセスのアドレス空間は分離されていて, 勝手に読み書きできるのはそのプロセスのアドレス空間のみ

大元の思想 (多分こうだったんじゃないか劇場)

- プロセスのアドレス空間は分離されていて, 勝手に読み書きできるのはそのプロセスのアドレス空間のみ
- プロセスの外とのやり取りは?

大元の思想 (多分こうだったんじゃないか劇場)

- プロセスのアドレス空間は分離されていて, 勝手に読み書きできるのはそのプロセスのアドレス空間のみ
- プロセスの外とのやり取りは?
- そのひとつとしてファイル API があった
  - fd = open(...);
  - read(fd, buf, sz);
  - write(fd, buf, sz);

大元の思想 (多分こうだったんじゃないか劇場)

- プロセスのアドレス空間は分離されていて, 勝手に読み書きできるのはそのプロセスのアドレス空間のみ
- プロセスの外とのやり取りは?
- そのひとつとしてファイル API があった
  - fd = open(...);
  - read(fd, buf, sz);
  - write(fd, buf, sz);
- ⇒ プロセスの外との情報の出し入れは全てこれ—ファイルディスクリプタに read/write を発行する—でやる

大元の思想 (多分こうだったんじゃないか劇場)

- プロセスのアドレス空間は分離されていて, 勝手に読み書きできるのはそのプロセスのアドレス空間のみ
- プロセスの外とのやり取りは?
- そのひとつとしてファイル API があった
  - fd = open(...);
  - read(fd, buf, sz);
  - write(fd, buf, sz);
- ⇒ プロセスの外との情報の出し入れは全てこれ—ファイルディスクリプタに read/write を発行する—でやる
- ネットワーク, 他のプロセスとの通信, OS からの情報取得, etc.

Unix 的なもの

- プロセス外とのやり取りはみな「ファイルディスクリプタへの read/write」で統一
- さらに, 擬似的なファイル (ファイル名などはあるが, 実体は 2 次記憶と無関係なファイル) で多くの機能を提供
- “everything is file”

ファイル由来ではないファイルディスクリプタ

- ソケット
  - 目的：他のプロセスとの通信 (同一計算機内, ネットワーク越し)
  - 作り方：socket システムコール
- パイプ
  - 目的：他のプロセスとの通信 (同一計算機内)
  - 作り方：pipe システムコール
- タイマー fd
  - 目的：時間が来たら readable になる
  - 作り方：timerfd_open システムコール
- シグナル fd (説明しない)
  - 目的：シグナルを OS から受け取る
  - 作り方：signalfd システムコール

擬似的なファイル

- 普通のファイルのように見える (ファイル名があり, プログラムからは普通に `open`) が,
- 挙動はいわゆる普通のファイルの挙動 (最後に write したものが read で読まれる) とは違う
- 例
  - 名前付きパイプ (FIFO)
  - `/proc` ファイルシステム
  - tmpfs
  - サウンド, ビデオなどデバイスの入出力

Unix 的なもの

リダイレクト, パイプの仕組み

擬似ファイル

ファイルディスクリプタの子プロセスへの継承

- 開いているファイルディスクリプタは, fork (プロセス生成) 時に子プロセスへ引き継がれる (親が作ったファイルディスクリプタを, 子プロセスも使える)

```
1 int fd = open( ... );
2 pid_t pid = fork();
3 if (pid == 0) {
4   /* 子プロセス */
5   read(fd, buf, sz); /* OK */
6 }
```

- exec 後もそのまま有効であり続ける

```
1 int fd = open("bar", ...);
2 pid_t pid = fork();
3 if (pid == 0) {
4   /* 子プロセス */
5   execv("./foo", ...);
6 }
```

foo の中でも, もし fd の値がわかれば, bar が読める

標準入出力

- ほとんどのプログラムは,「0, 1, 2 番のディスクリプタが開かれている」ことを前提に書かれている
  - 0 : 入力 (標準入力)
  - 1 : 出力 (標準出力)
  - 2 : 出力 (標準エラー出力)
- 入 (出) 力リダイレクトは fd の値を 0 (1) に付け替える (→ dup2 システムコール)

dup2 システムコール

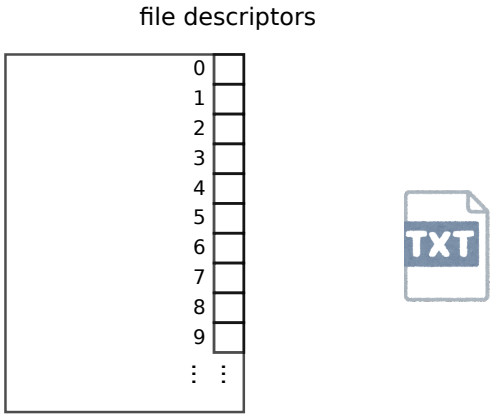
- `int err = dup2(`*oldfd*`, `*newfd*`);`
- ファイルディスクリプタ *oldfd* を *newfd* でも使えるようにする
- 例えば以下は, ファイル `bar` を 0 番でも (fd でも) 読めるようにする

```
1 int fd = open("bar", ...);
2 dup2(fd, 0);
```

入力リダイレクト

- “*cmd* < *filename*” 相当のことをする (シェルのような) プログラム

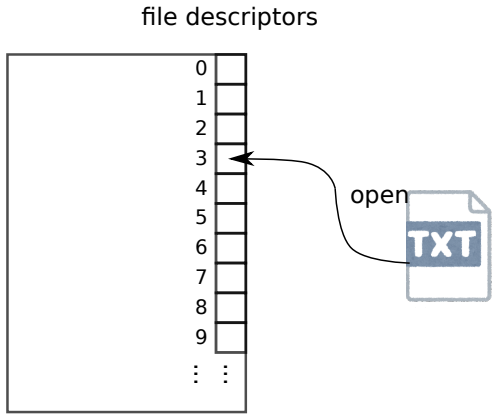
```
1  const int fd = open(filename, O_RDONLY);
2  pid_t pid = fork();
3  if (pid) { /* 親プロセス */
4    close(fd); /* 親は不要なfd を閉じる */
5  } else { /* 子プロセス */
6    /* fd -> 0 へ付け替え
7    (0を読むとfilename が読める) */
8    if (fd != 0) {
9      dup2(fd, 0);
10     close(fd);
11   }
12   execvp(cmd, ..);
13   ... }
```



入力リダイレクト

- “*cmd* < *filename*” 相当のことをする (シェルのような) プログラム

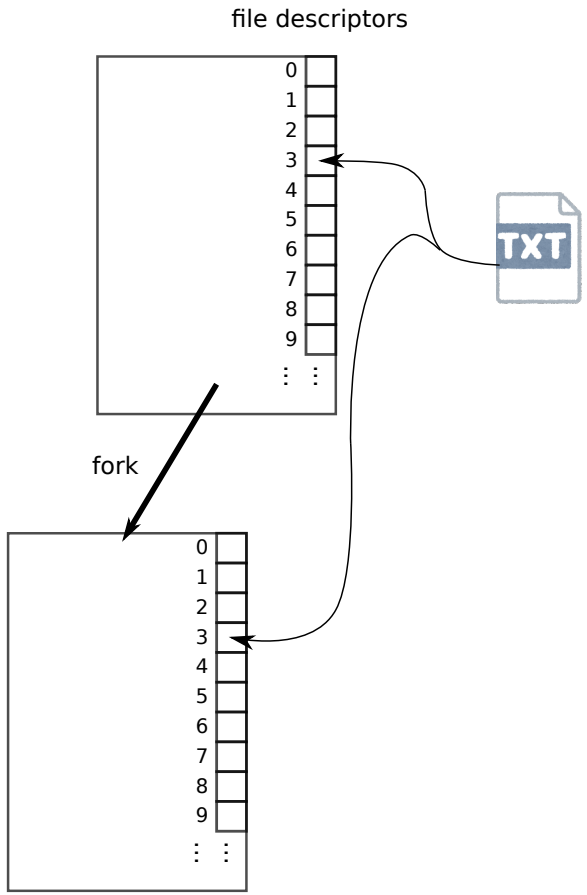
```
1  const int fd = open(filename, O_RDONLY);
2  pid_t pid = fork();
3  if (pid) { /* 親プロセス */
4    close(fd); /* 親は不要なfd を閉じる */
5  } else { /* 子プロセス */
6    /* fd -> 0 へ付け替え
7    (0を読むとfilename が読める) */
8    if (fd != 0) {
9      dup2(fd, 0);
10     close(fd);
11   }
12   execvp(cmd, ..);
13   ... }
```



入力リダイレクト

- “*cmd* < *filename*” 相当のことをする (シェルのような) プログラム

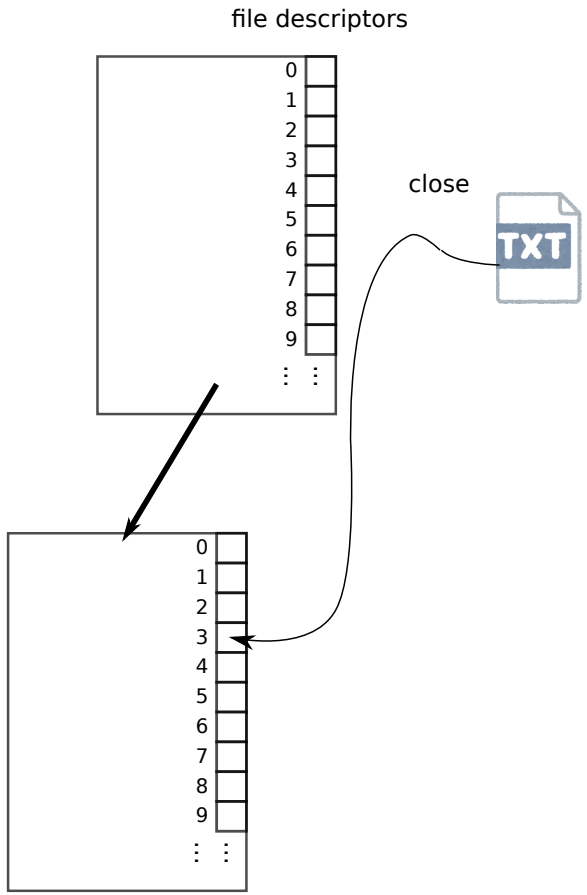
```
1  const int fd = open(filename, O_RDONLY);
2  pid_t pid = fork();
3  if (pid) { /* 親プロセス */
4    close(fd); /* 親は不要なfd を閉じる */
5  } else { /* 子プロセス */
6    /* fd -> 0 へ付け替え
7    (0を読むとfilename が読める) */
8    if (fd != 0) {
9      dup2(fd, 0);
10     close(fd);
11   }
12   execvp(cmd, ..);
13   ... }
```



入力リダイレクト

- “*cmd* < *filename*” 相当のことをする (シェルのような) プログラム

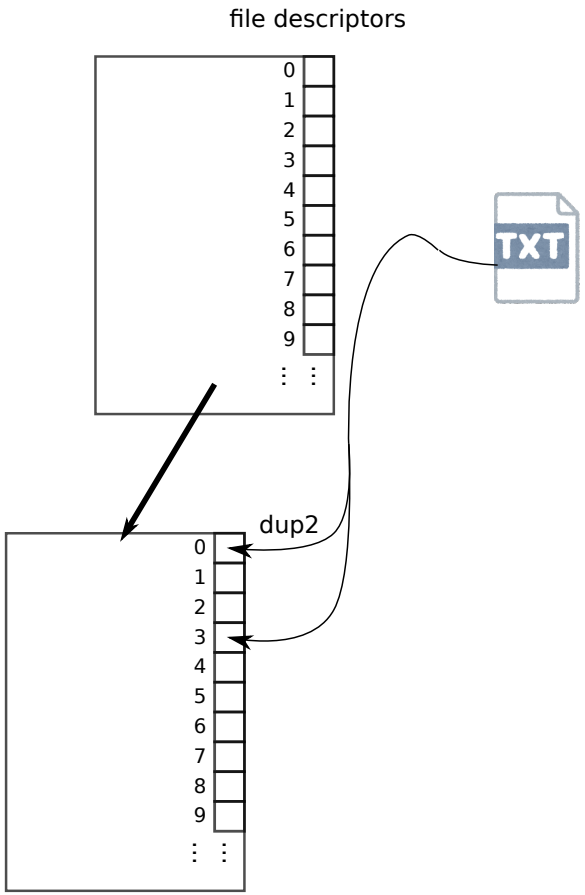
```
1  const int fd = open(filename, O_RDONLY);
2  pid_t pid = fork();
3  if (pid) { /* 親プロセス */
4    close(fd); /* 親は不要なfd を閉じる */
5  } else { /* 子プロセス */
6    /* fd -> 0 へ付け替え
7    (0を読むとfilename が読める) */
8    if (fd != 0) {
9      dup2(fd, 0);
10     close(fd);
11   }
12   execvp(cmd, ..);
13   ... }
```



入力リダイレクト

- “*cmd* < *filename*” 相当のことをする (シェルのような) プログラム

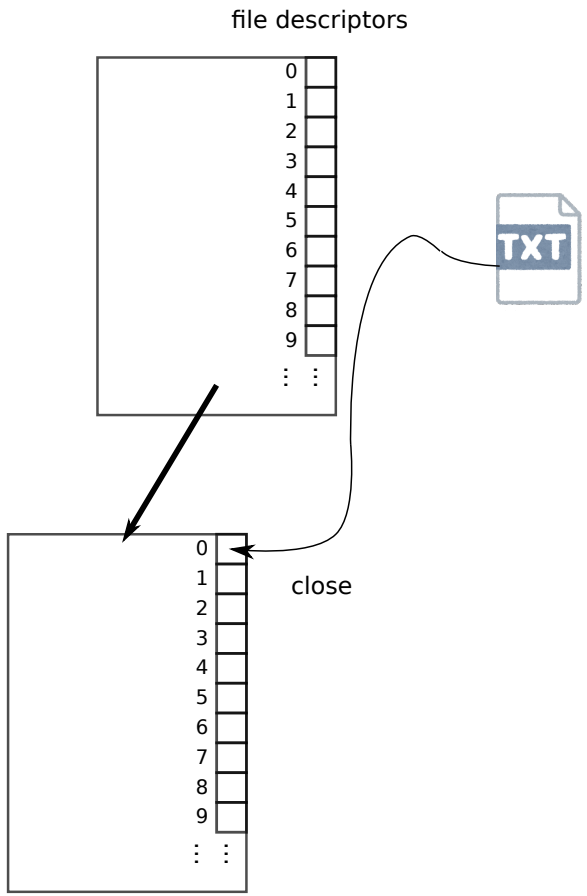
```
1  const int fd = open(filename, O_RDONLY);
2  pid_t pid = fork();
3  if (pid) { /* 親プロセス */
4    close(fd); /* 親は不要なfd を閉じる */
5  } else { /* 子プロセス */
6    /* fd -> 0 へ付け替え
7    (0を読むとfilename が読める) */
8    if (fd != 0) {
9      dup2(fd, 0);
10     close(fd);
11   }
12   execvp(cmd, ..);
13   ... }
```



入力リダイレクト

- “*cmd* < *filename*” 相当のことをする (シェルのような) プログラム

```
1  const int fd = open(filename, O_RDONLY);
2  pid_t pid = fork();
3  if (pid) { /* 親プロセス */
4    close(fd); /* 親は不要なfd を閉じる */
5  } else { /* 子プロセス */
6    /* fd -> 0 へ付け替え
7    (0を読むとfilename が読める) */
8    if (fd != 0) {
9      dup2(fd, 0);
10     close(fd);
11   }
12   execvp(cmd, ..);
13   ... }
```



出力リダイレクト

- “*cmd* > *filename*” 相当のことをする (シェルのような) プログラム

```
1  const int fd = creat(filename);
2  pid_t pid = fork();
3  if (pid) { /* 親プロセス */
4    close(fd); /* 親は不要なfd を閉じる */
5  } else { /* 子プロセス */
6    /* fd -> 1 へ付け替え
7    (1に書くとfilename に書ける) */
8    if (fd != 1) {
9      close(fd);
10     dup2(fd, 1);
11   }
12   execvp(cmd, ..);
13   ... }
```

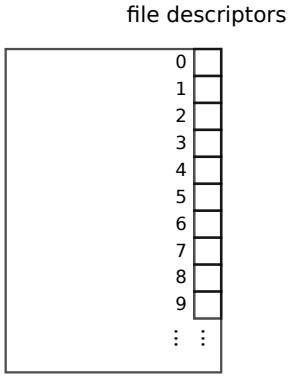
pipe システムコール

- `int rw[2]; int err = pipe(rw);`
  - `rw[0]`, `rw[1]` に, それぞれ「読み出し用」「書き込み用」のファイルディスクリプタを書き込み
  - `rw[1]` に書いたデータが `rw[0]` から読み出せる (パイプ)
  - もちろん実際の読み書きは read, write で行える
- これと, fork 時にファイルディスクリプタが継承する仕組みを使い, 親子プロセス間での通信が可能

パイプ (親 → 子)

親 → 子へデータを送るパターン

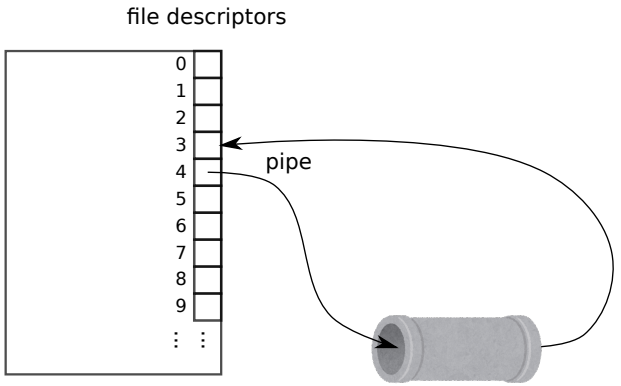
```
1  /* 親がw に書いたものが子の標準入力(0)から
          読めるようにする */
2  int rw[2];
3  pipe(rw);
4  int r = rw[0], w = rw[1];
5  pid_t pid = fork();
6  if (pid) { /* 親プロセス */
7    close(r);
8    ... w に書き込む ...
9    close(w);
10 } else { /* 子プロセス */
11   close(w);
12   dup2(r, 0);
13   close(r);
14   execvp(...); /* 0番から読むコマンド */
15 }
```



パイプ (親 → 子)

親 → 子へデータを送るパターン

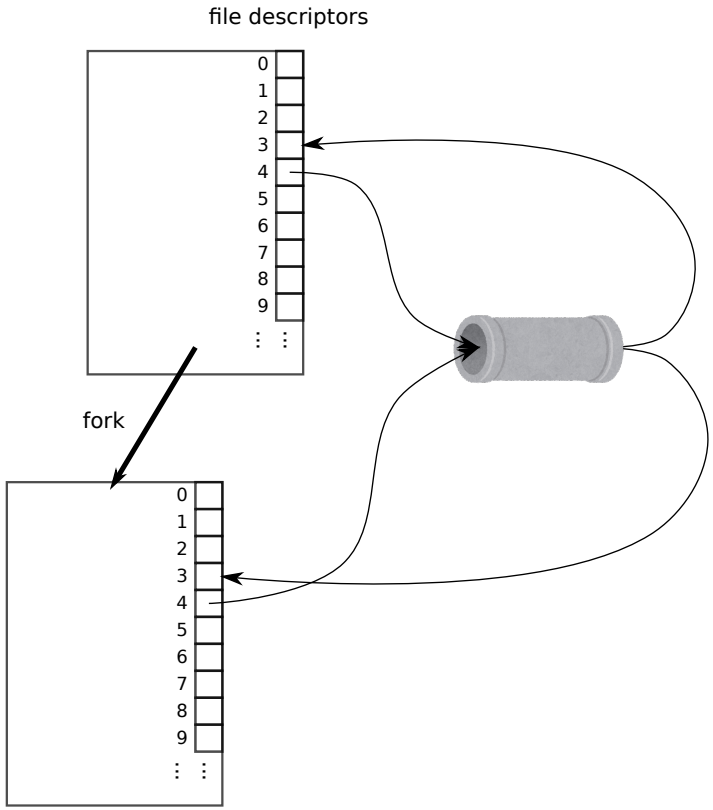
```
1  /* 親がw に書いたものが子の標準入力(0)から
          読めるようにする */
2  int rw[2];
3  pipe(rw);
4  int r = rw[0], w = rw[1];
5  pid_t pid = fork();
6  if (pid) { /* 親プロセス */
7    close(r);
8    ... w に書き込む ...
9    close(w);
10 } else { /* 子プロセス */
11   close(w);
12   dup2(r, 0);
13   close(r);
14   execvp(...); /* 0番から読むコマンド */
15 }
```



パイプ (親 → 子)

親 → 子へデータを送るパターン

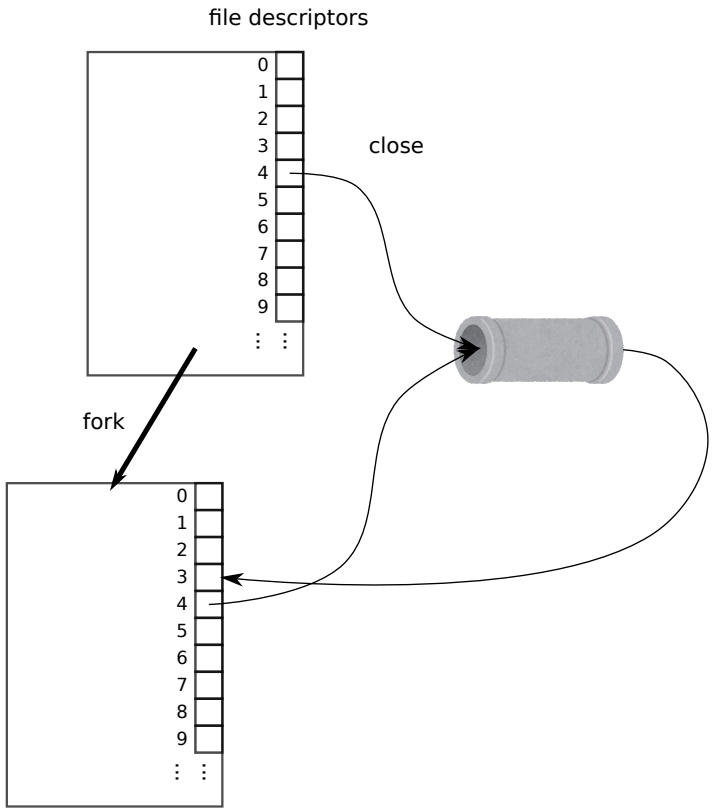
```
1  /* 親がw に書いたものが子の標準入力(0)から
          読めるようにする */
2  int rw[2];
3  pipe(rw);
4  int r = rw[0], w = rw[1];
5  pid_t pid = fork();
6  if (pid) { /* 親プロセス */
7    close(r);
8    ... w に書き込む ...
9    close(w);
10 } else { /* 子プロセス */
11   close(w);
12   dup2(r, 0);
13   close(r);
14   execvp(...); /* 0番から読むコマンド */
15 }
```



パイプ (親 → 子)

親 → 子へデータを送るパターン

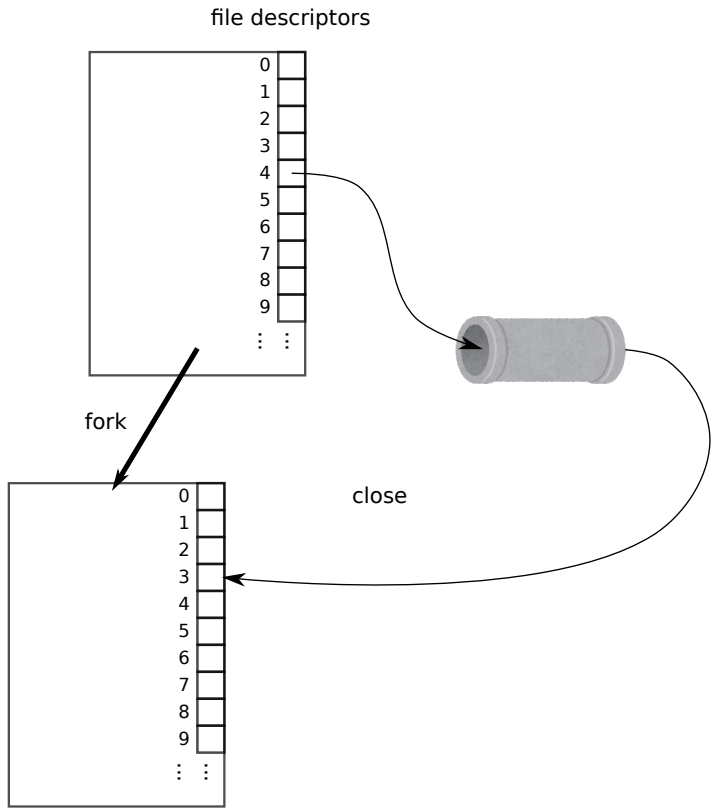
```
1  /* 親がw に書いたものが子の標準入力(0)から
          読めるようにする */
2  int rw[2];
3  pipe(rw);
4  int r = rw[0], w = rw[1];
5  pid_t pid = fork();
6  if (pid) { /* 親プロセス */
7    close(r);
8    ... w に書き込む ...
9    close(w);
10 } else { /* 子プロセス */
11   close(w);
12   dup2(r, 0);
13   close(r);
14   execvp(...); /* 0番から読むコマンド */
15 }
```



パイプ (親 → 子)

親 → 子へデータを送るパターン

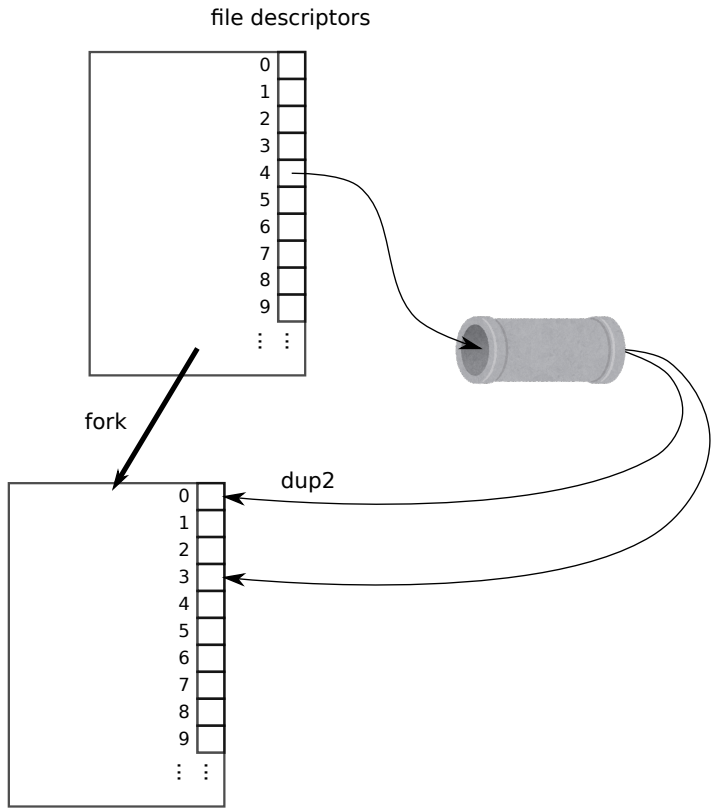
```
1  /* 親がw に書いたものが子の標準入力(0)から
          読めるようにする */
2  int rw[2];
3  pipe(rw);
4  int r = rw[0], w = rw[1];
5  pid_t pid = fork();
6  if (pid) { /* 親プロセス */
7    close(r);
8    ... w に書き込む ...
9    close(w);
10 } else { /* 子プロセス */
11   close(w);
12   dup2(r, 0);
13   close(r);
14   execvp(...); /* 0番から読むコマンド */
15 }
```



パイプ (親 → 子)

親 → 子へデータを送るパターン

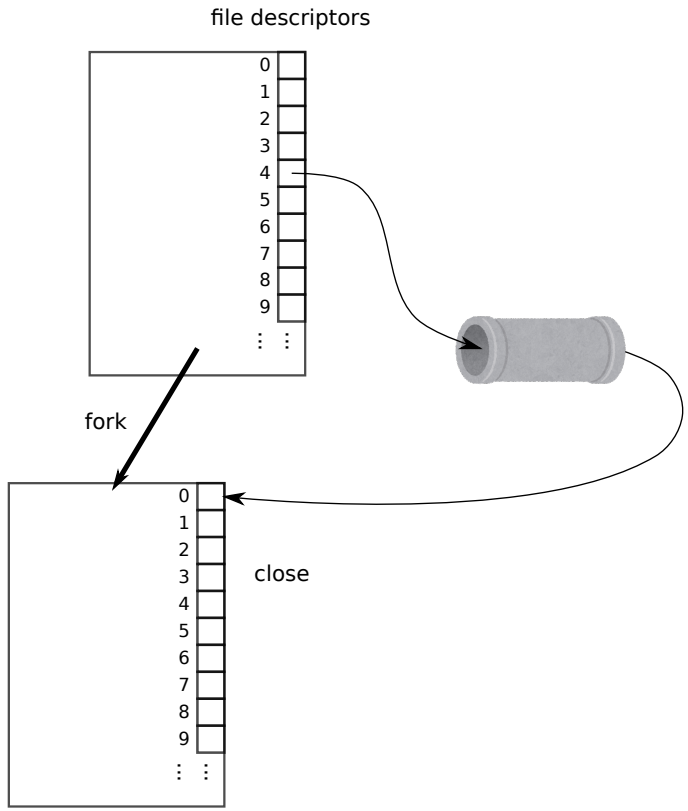
```
1  /* 親がw に書いたものが子の標準入力(0)から
          読めるようにする */
2  int rw[2];
3  pipe(rw);
4  int r = rw[0], w = rw[1];
5  pid_t pid = fork();
6  if (pid) { /* 親プロセス */
7    close(r);
8    ... w に書き込む ...
9    close(w);
10 } else { /* 子プロセス */
11   close(w);
12   dup2(r, 0);
13   close(r);
14   execvp(...); /* 0番から読むコマンド */
15 }
```



パイプ (親 → 子)

親 → 子へデータを送るパターン

```
1  /* 親がw に書いたものが子の標準入力(0)から
          読めるようにする */
2  int rw[2];
3  pipe(rw);
4  int r = rw[0], w = rw[1];
5  pid_t pid = fork();
6  if (pid) { /* 親プロセス */
7    close(r);
8    ... w に書き込む ...
9    close(w);
10 } else { /* 子プロセス */
11   close(w);
12   dup2(r, 0);
13   close(r);
14   execvp(...); /* 0番から読むコマンド */
15 }
```



パイプ (子 → 親)

子 → 親へデータを送るパターン

```
1  /* 子が標準出力 (1)に書いたものが親のr から読めるようにする */
2  int rw[2];
3  pipe(rw);
4  int r = rw[0], w = rw[1];
5  pid_t pid = fork();
6  if (pid) { /* 親プロセス */
7    close(w);
8    ... r から読み込む ...
9    close(r);
10 } else { /* 子プロセス */
11   close(r);
12   dup2(w, 1);
13   close(w);
14   execvp(...); /* 1番へ書くコマンド */
15 }
```

- 注: popen ライブラリ関数がこれに相当

C 言語ストリーム API

- C 言語では Unix の open, read, write の代わりに, 以下の API を使うことが多い
  - `FILE * fp = fopen(filename, mode);`
  - `fread(buf, size, n, fp);`
  - `fwrite(buf, size, n, fp);`
- `FILE` — ファイル構造体
- 標準入出力に対応する, `FILE *`型の変数がある
  - `stdin` : $\leftrightarrow 0$
  - `stdout` : $\leftrightarrow 1$
  - `stderr` : $\leftrightarrow 2$

高水準なファイル入出力

- FILE *に対しては, より高水準または簡便な API もある
  - fgetc(*fp*); — 1 文字入力
  - fgets(*s*, *size*, *fp*); — 1 行入力
  - fprintf(*fp*, *format*, ...); — 値を文字列に変換して出力
  - fscanf(*fp*, *format*, ...); — 文字列を値に変換しながら入力
- 以下は想像通り
  - getchar() ≡ fgetc(stdin);
  - gets(*s*); ≡ fgets(*s*, ∞, stdin); (危険)
  - printf(*fp*, *format*, ...); ≡ fprintf(stdout, *format*, ...);
  - scanf(*fp*, *format*, ...); ≡ fscanf(stdin, *format*, ...);

ファイルディスクリプタからファイル構造体

- (open などで得た) ファイルディスクリプタに対応した, ファイル構造体を作ることが可能

```
1 FILE * fp = fdopen(fd, mode);
```

- FILE *を得るには fopen を使わないといけないわけではない
- 使いたい API に応じて使い分けることが可能

Unix 的なもの

リダイレクト, パイプの仕組み

擬似ファイル

擬似ファイル

- ファイル =2 次記憶上のデータ, と決めつけるのをやめるのが出発点
- open して, read/write 出来るもの (それで有用な動作をするもの) は全て「ファイル」にしてしまえ

名前付きパイプ (FIFO)

- `int err = mkfifo(`*pathname*`, `*mode*`);`
- 同名のコマンドもある
- あるプロセスが書き込んだものが, 読み出すと出てくる
- ファイルシステム上に名前を持つ以外, パイプとほぼ同じ機能
- コマンド使用例

```
$ mkfifo q
$ cat q   # ブロック
```

```
$ echo hello > q
```

/proc ファイルシステム

- プロセスや, OS 内部の情報を読み出し, 変更できるためのファイル群
- いろいろ開いてみると良い
  - `/proc/cpuinfo` : cpu 数, 機種名など
  - `/proc/meminfo` : メモリサイズや利用状況など
  - `/proc/`*pid*`/...` : プロセス *pid* に関する様々な情報
- これらを読むのに普通のファイルを読むコマンド (cat, grep, etc.) が使えるのも「Unix 的」
- これらが実際に 2 次記憶 (HDD?) の中に書かれているわけではない

cgroups ファイルシステム

- プロセスの集合に割り当てる資源 (CPU, メモリ, etc.) を制御する機能
- 使用例

```
1 sudo mount -t cgroup2 none dir
```

- グループ → *dir* 下のディレクトリで表現
- 詳しくは `05_memory.pdf` の cgroups の節参照

tmpfs

- 実体が (普通の) メモリ上にあるファイルシステム
- 再起動時にはデータが失われる
- だが, 一部の (少量の) ファイルを高速にアクセスしたい場合には向く
- ただしメモリを消費する
- ならば OS のキャッシュに任せたほうが良いという説もある
- 使用例

```
1 mkdir my_dir
2 sudo mount -t tmpfs -o size=100M,mode=0755 tmpfs my_dir
3 sudo chown user:group my_dir
```

デバイスファイル

- 入出力装置 (カメラ, マイク, etc.) も, あるファイルを読み書きすることで制御やデータの取得が行えるようになっている
- 詳細は装置ごとに異なるので深入りしないが, これも Unix 的な考え方の一例 (read, write して意味がある動作をするものは, みなファイルとして見せる)
- 単純なデバイスファイル
  - `/dev/null`：書いてもなにもおきない, 読んでもすぐに EOF になる
  - `/dev/zero`：書いてもなにもおきない, 読むと無限に 0 が読み出される
  - `/dev/urandom`：乱数 (バイナリのバイト列) が読み出される