オペレーティングシステム

システムソフトウェア論（理）

システムソフトウェア（工）

14. UNIX (1)

平成25年度 春学期

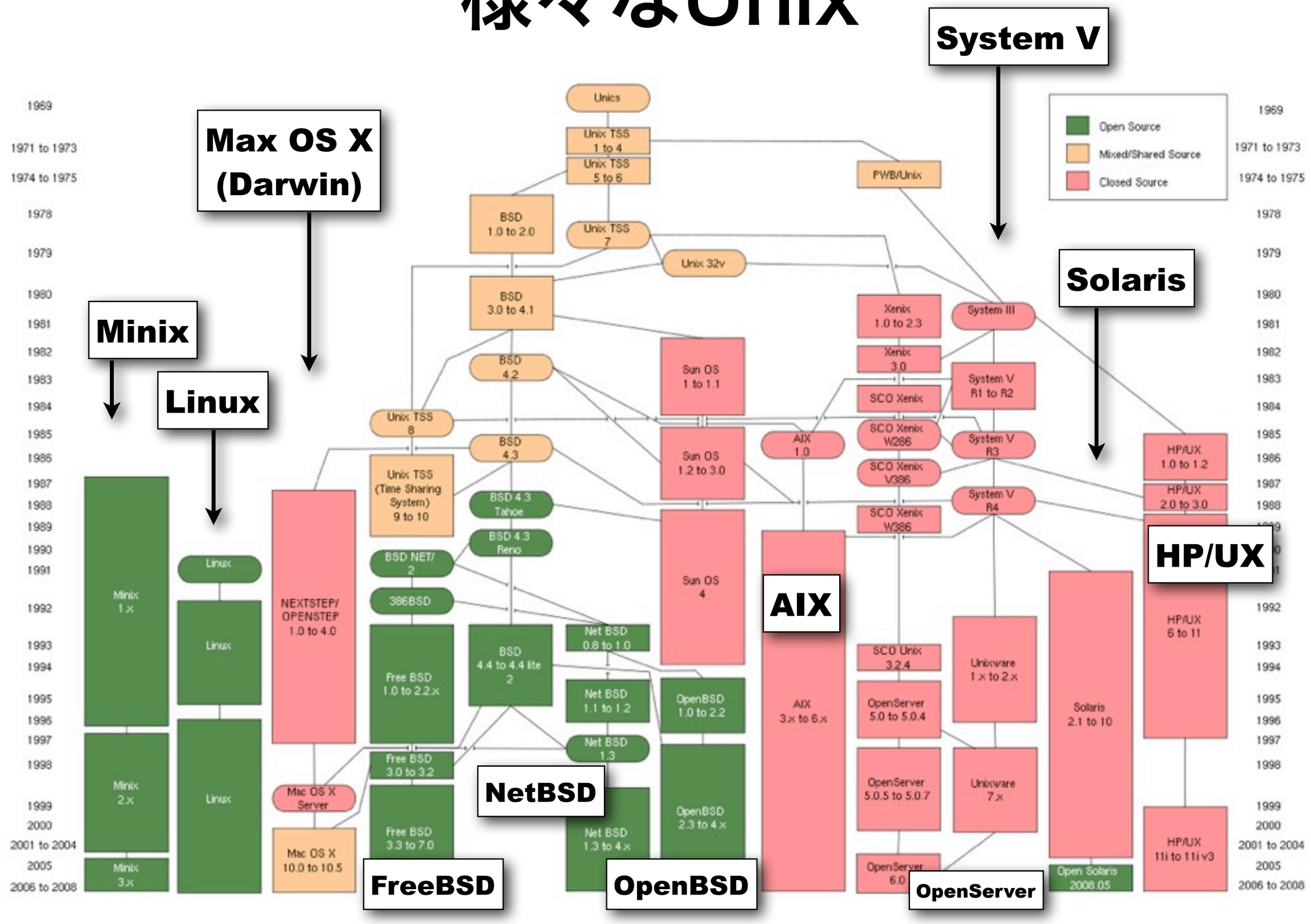
担当: 荻原剛志

UNIX

- 1969年、AT&Tベル研究所で開発開始
  - ✦ 記述言語はC。移植性に優れている。
  - ✦ 70〜90年代、カリフォルニア大学バークレー校が開発の中心となった**BSD** (Berkeley Software Distribution)系Unixが配布
  - ✦ AT&Tが開発した **System V** (ファイヴ) 系とBSD系が勢力を二分（90年代）→Unix戦争
  - ✦ Linuxの台頭、標準Unix仕様、業界の淘汰
  - ✦ 日本はそのころ「Σ(シグマ)計画」で絶賛大失敗中……。
- オープンソース、商用(proprietary) Unixシステム、「Unixライク」なものなど多種が存在

http://en.wikipedia.org/wiki/Unix

様々なUnix



UNIX仕様

- POSIX（ポジックス）
  - ✦ 各種のOSに共通APIを定め、アプリケーションの相互互換性を高める目的でIEEEが策定。ISO規格
  - ✦ システムコール、標準関数などを規定
- SUS (Single UNIX Specification)
  - ✦ “Unix” を名乗るために必要とされる仕様。IEEEと業界団体である The Open Group が策定。
  - ✦ POSIXを包含する
  - ✦ 準拠マークとして UNIX 98、UNIX 03 など

※ GUI、例えば X Windowは Unixと直接関係しない。

各種UNIX (1)

- Mac OS X / Mac OS X Server
  - Apple。BSD系。UNIX 03準拠。
  - カーネルは Darwin。源流は Mach OS + BSD。
  - iOSもカーネルは共通するが Unixではない。
  - オープンソースOSの Darwin は UNIX 03準拠。
- AIX
  - IBM。System V系。UNIX 98準拠
  - 信頼性、高速性に優れる
- HP/UX
  - Hewlett-Packard。System V系。UNIX 03準拠
  - サーバ運用に実績がある

各種UNIX (2)

- Solaris（ソラリス）
  - Sun Microsystems。System V系（BSD系機能も包含）。UNIX 03準拠
  - 同じ Sunが以前開発した Sun OS は BSD系。
- OpenServer / UnixWare
  - SCO (Santa Cruz Operation)。System V系
- Linux、FreeBSD / NetBSD / OpenBSD
  - SUSに準拠するディストリビューションはない
    - 審査などに時間とコストがかかるため
  - **ディストリビューション** = 利用者がインストールしたり、利用できる形にまとめた頒布形態

各種UNIX (3)

- Windows（Windows単体は Unixではない）
  - ✦ Windows NT系（Windows 2000まで）は POSIX準拠のサブシステムを搭載
  - ✦ XP以降は Interix サブシステムをインストールすることでUnixの機能が利用可能。Windows 7 は SUA (Subsystem for UNIX-based Applications) をインストールする
- ✦ Windows上でUnixライクな環境を提供するものに **Cygwin**（シグウィン）があるが、Cygwinはライブラリ群であって OS（またはそのサブシステム）ではない

システムコール

- Unixでは、C言語の関数呼び出しとして OSの機能を利用できる。これをシステムコールと呼ぶ。
  - 典型的には、レジスタなどに必要な情報をコピーしてから**トラップ命令**で割込みを行い、カーネルを呼ぶ。
  - Unixの種類やバージョンによって、あるAPIがシステムコールかどうかの扱いは異なることがある。
  - オンラインマニュアルでは、セクション２がシステムコール、セクション３がライブラリ関数である。
- プロセス操作、ファイル操作、ファイルやソケットを用いた入出力、シグナルハンドラの管理、時刻や各種パラメータの管理、など。

プロセスID

- プロセスは固有の番号、プロセスIDを持つ。
  - ✦ マシンの起動後、1から順番に通し番号が振られて行く
- プロセスは新しいプロセスを生成することができる。生成元を親プロセス、生成された側を子プロセスと呼ぶことにする
- 自身のプロセスIDは getpid() で、親プロセスのプロセスIDは getppid() で調べることができる。
- 子プロセスは親プロセスの環境変数を引き継ぐ。

環境変数

- プロセスは実行開始時に、親プロセスから各種のパラメータを受け継ぐ
- パラメータは文字列で、次の形式のものが多数連続したもの。これを環境変数と呼ぶ。

環境変数名＝値を表す文字列

- プログラムからは、関数 main() の３番目の引数として、あるいは getenv() 関数で参照可能。
- コマンドラインから実行したプログラムの親プロセスはその時のシェルになるので、シェルで設定した環境変数がプログラムから参照できる

プログラム例 —コマンドラインの引数

```
#include <stdio.h>

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

    for (i = 0; i < argc; i++)
        printf("%d: \"%s\"\n", i, argv[i]);
    return 0;
}
```

argv は文字列（ポインタ）の配列
argc は配列の要素の個数

実行例

```
$ ./printarg a 012 '(@ @;'
0: "./printarg"
1: "a"
2: "012"
3: "(@ @;"
```

プログラム例 —環境変数の印刷

```
#include <stdio.h>

int main(int argc, char *const argv[],
                   char *const envp[])
{
    int i;

    for (i = 0; envp[i]; i++)
        printf("%d: \"%s\"\n", i, envp[i]);
    return 0;
}
```

envp は文字列（ポインタ）の配列
最後の要素は NULL

実行例

```
$ ./printenv
0: "TERM_PROGRAM=Apple_Terminal"
1: "TERM=xterm-color"
2: "SHELL=/bin/bash"
（以下略）
```

プロセスの生成

- プロセスがシステムコール fork() を呼ぶと、自分と同じプログラムを実行する子プロセスが生成される
  - 親も子も、fork() の次のコードを継続して実行しているように見える
  - fork() の返り値は、親プロセスでは子プロセスのプロセスID、子プロセスでは０であることから、自分が親か子かを区別できる
    - 注意： Linuxは cloneという独自のシステムコールを持つ
- 親プロセスは、システムコール wait() を使い、子プロセスが実行を終了するのを待つ
  - 子プロセスの終了コードを取得可能
  - 終了コード = main関数の返り値／exit() の引数

プログラム例 —fork()の動作

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

int main(void)
{
    int n;
    int pid = getpid();
    printf("pid = %d\n", pid);
    n = fork();
    pid = getpid();
    printf("pid = %d, %d\n", pid, n);
    return 0;
}
```

実行例

```
$ ./fork
pid = 68935
pid = 68935, 68936
pid = 68936, 0
```

親プロセスの出力

子プロセスの出力

プログラム例 —wait()の動作

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

int main(void)
{
    int n, status;
    printf("pid = %d\n", getpid());
    if (fork() == 0) {
        n = getpid();
        sleep(1); // 1秒待つ
        printf("pid = %d\n", n);
        return (n % 10); // 終了コード
    }
    n = wait(&status);
    if (n > 0)
        printf("child = %d, %d\n", n, status >> 8);
    return 0;
}
```

実行例

```
$ ./wait
pid = 69264
pid = 69265
child = 69265, 5
```

子プロセスだけが実行する

親プロセスだけが実行する

子プロセスの終了コード

プログラムの実行

- プロセスはシステムコール execl() または execv() などを使って、指定したプログラムを実行するプロセスになることができる
  - ✦ これらのシステムコールでは、実行ファイルのパスが必要で、引数も（あれば）指定できる
  - ✦ 環境変数の設定も可能
- Unixでは、指定したプログラムを実行するプロセスを生成するのではなく、次の手順に従う

  (1) 自分のプロセスのコピーを生成

  (2) 必要なら、入出力などの設定を行う（後述）

  (3) 子プロセスが自ら、指定したプログラムを実行するプロセスに変化する

プログラム例 —execl()の動作

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

int main(void)
{
    int status;
    printf("begin...\n");
    if (fork() == 0) { //子プロセスだけが実行する
        execl("/bin/date", "date", "+%H:%M", NULL);
    }
    (void)wait(&status);
    printf("end...\n");
    return 0;
}
```

実行例

```
$ ./exec
begin...
10:28
end...
```

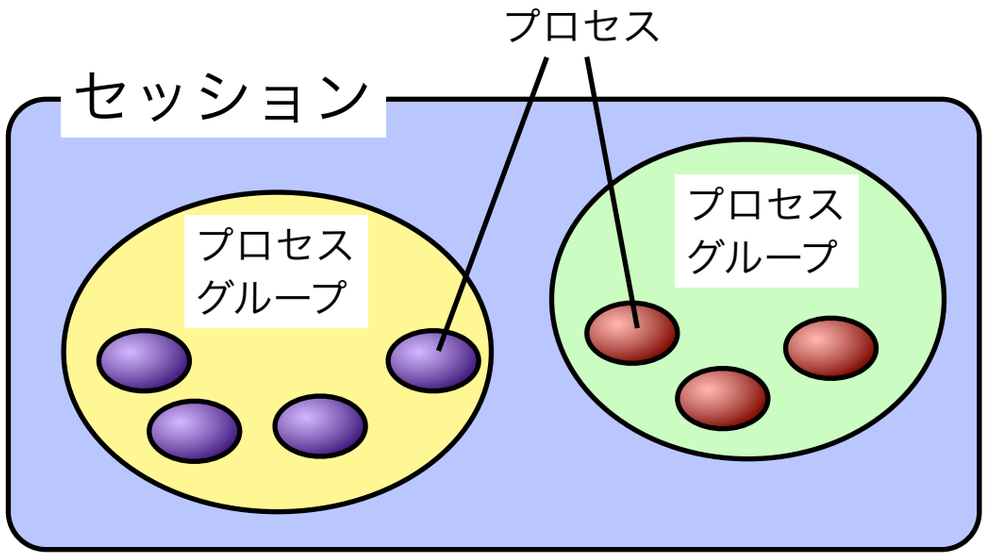
親プロセスだけが実行する

普通にコマンドを打つと：

```
$ date +%H:%M
10:27
```

セッションとプロセスグループ

- 各プロセスは、属性値としてセッションIDとプロセスグループIDを持つ
  - ✦ 関連のある複数のプロセスを管理するための概念
    - 例えば、シェルはパイプで連結された複数のプロセスに同じプロセスグループIDを割り当てる
  - ✦ getsid(), getpgid() で参照、setsid(), setpgid() で設定
- セッションIDとプロセスグループIDは fork() によるプロセスの生成で親から引き継がれる



ファイルディスクリプタ

- プロセスは入出力のための情報をテーブルの形で保持する。これにアクセスするためのインデックス（整数）を**ファイルディスクリプタ**(file descriptor) と呼ぶ。
- ファイルディスクリプタはファイルだけではなく、ディレクトリやデバイス（特殊ファイル）、パイプやソケットも扱う。
- システムコールで入出力を行う。例：read(), write()
- すべてのプロセスでは、以下の３つのファイルディスクリプタがあらかじめ予約されている。
  - ０：標準入力（通常は端末からの入力）
  - １：標準出力（通常は端末への出力）
  - ２：標準エラー出力（通常は端末への出力）

ストリーム

- Unixには、FILE型データを使って入出力を行う標準関数群が用意されている。この仕組みを**ストリーム**(stream)と呼ぶ。例： getc()、printf()、scanf()
- ストリームはファイルディスクリプタを使っている。
- 入出力を扱うプログラムが記述しやすいように、バッファリングの仕組みを実装している。
  - ファイル入出力では、バッファリングの効果で、システムコールを直接扱うよりも高速なことがある
  - ネットワークやデバイスにアクセスするには、システムコールを使った方がよい
- 以下の３つのストリームがあらかじめ予約されている。
  - stdin、stdout、stderr
  - それぞれ、標準入力、標準出力、標準エラー出力に対応

プログラム例 —ストリームによる入出力

```
#include <stdio.h>   // 引数で指定したファイルを標準出力に書き出す

int main(int argc, const char *argv[])
{
    int ch;
    FILE *st;   // 自分でオープンするストリーム

    if (argc < 2) {  // 引数がないと、標準エラー出力にメッセージ
        fprintf(stderr, "Error: no file name.\n");
        return 1;
    }
    if ((st = fopen(argv[1], "r")) == NULL) {  // オープン
        fprintf(stderr, "Error: can't open\n");
        return 2;
    }
    while ((ch = getc(st)) != EOF)   // 1バイトずつ読む
        putc(ch, stdout);  // 標準出力に1バイト書く。putchar()と同じ
    fclose(st);   // クローズ
    return 0;
}
```

fcopy.c

プログラム例 —システムコールで書いてみた

```
#include <unistd.h>  // 引数で指定したファイルを標準出力に書き出す
#include <fcntl.h>
#define CHARS    40

int main(int argc, const char *argv[]) {
    int fd, len;              // fdは自分でオープンするディスクリプタ
    char buffer[CHARS];       // バッファ
    if (argc < 2) {  // 引数がないと、標準エラー出力にメッセージ
        write(2, "Error: no file name.\n", 21);
        return 1;
    }
    if ((fd = open(argv[1], O_RDONLY)) < 0) {  // オープン
        write(2, "Error: can't open\n", 18);
        return 2;
    }                         // 最大40バイト読み、読めた分を書き出す
    while ((len = read(fd, buffer, CHARS)) > 0)
        write(1, buffer, len);
    close(fd);    // クローズ
    return 0;
}
```

lcopy.c

実行例

- lcopy : 低レベル（システムコール）を使った版
- fcopy : ストリーム（getc、putcなど）を使った版
- 3MB程度のファイルを読み、/dev/null に書き出す（出力は捨てられる）
- シェルの time コマンドで実行時間を計測

```
$ time ./lcopy large.data > /dev/null
real  0m0.214s
user  0m0.052s
sys   0m0.156s

$ time ./fcopy large.data > /dev/null
real  0m0.067s
user  0m0.057s
sys   0m0.007s
```

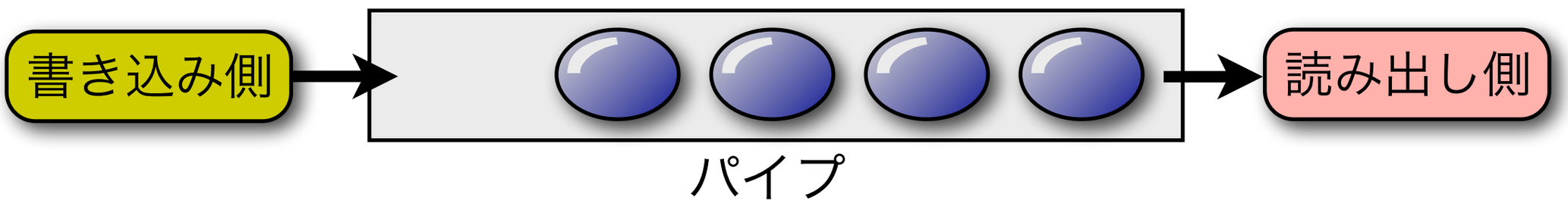
ストリームを使った方が速い。
内部のバッファリングが効率的に行われているため

プロセスと入出力

- 各プロセスにはあらかじめ標準入力、標準出力、標準エラー出力が備わっており、オープン、クローズの必要はない
  - 標準エラー出力：標準出力がファイルへの書き込みなどに使われている場合でもエラーを表示する目的で使う
- 親プロセスがオープンしていたファイルディスクリプタは、子プロセスに引き継がれる
  - シェルが使っていた入出力先が、シェルから起動されたプログラムの入出力先になる
  - この仕組みを利用して、リダイレクション（特にパイプ）を実現する

パイプ *pipe*

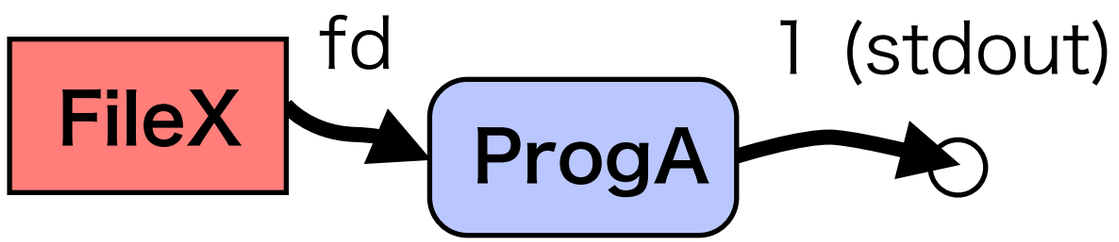
- Unixではバイト列を読み書きするパイプを、親子のプロセス間の通信に使う
  - ✦ read操作：パイプが空の時は待つ
  - ✦ write操作：パイプが満杯の時は待つ
  - ✦ パイプを使うプロセスは並列に動作する
  - ✦ パイプはUnixのシステムコールで利用できる
  - ✦ パイプは書き込み用と読み出し用のディスクリプタが対になっている



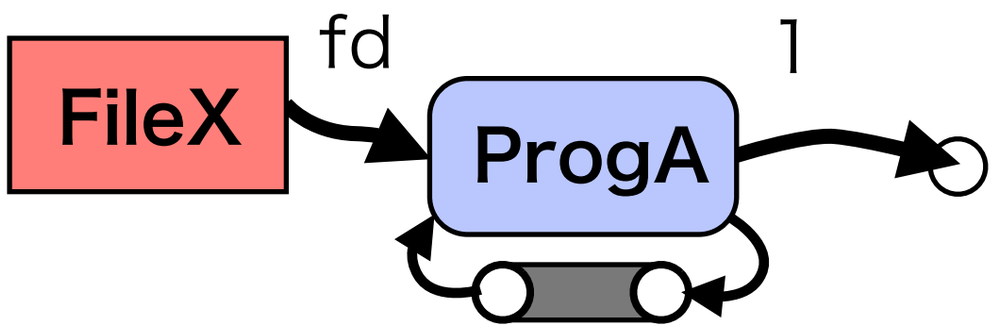
パイプの概要 (1)

- 例：親プロセスが、ファイルからの入力を前処理してくれる子プロセスを起動し、その結果を処理したい

(1) まずファイルをオープン

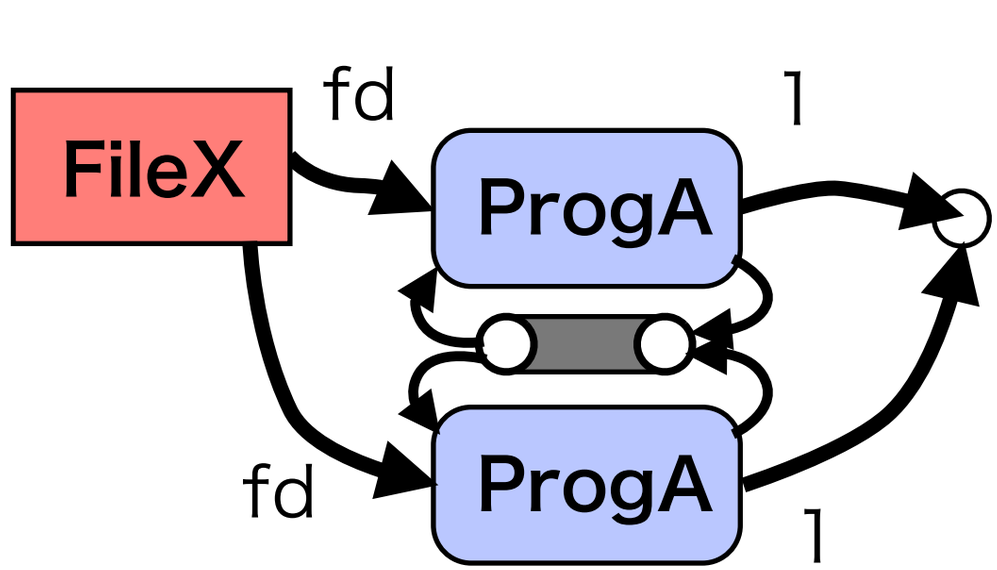


(2) pipe() でパイプを生成する



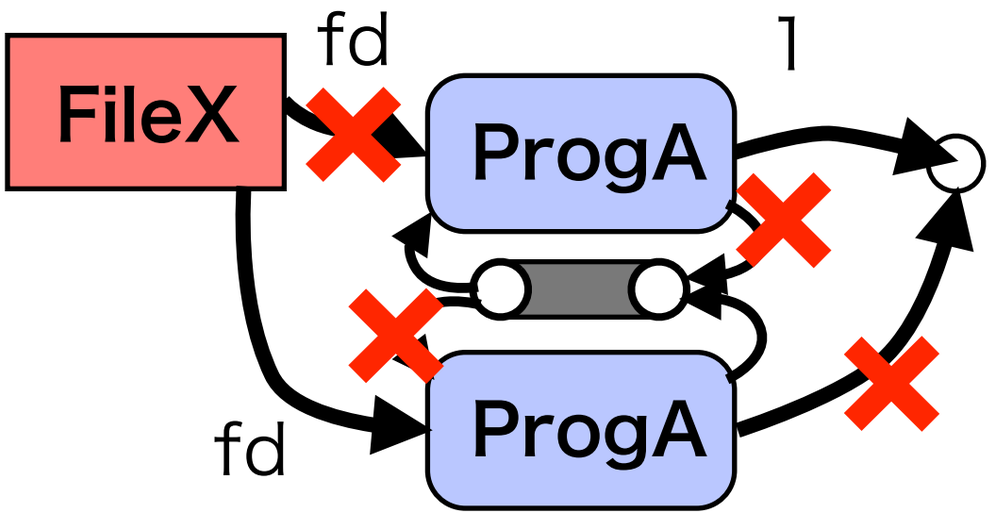
パイプは書き込み用と読み出し用のディスクリプタが対になっている

(3) fork() で子プロセスを生成

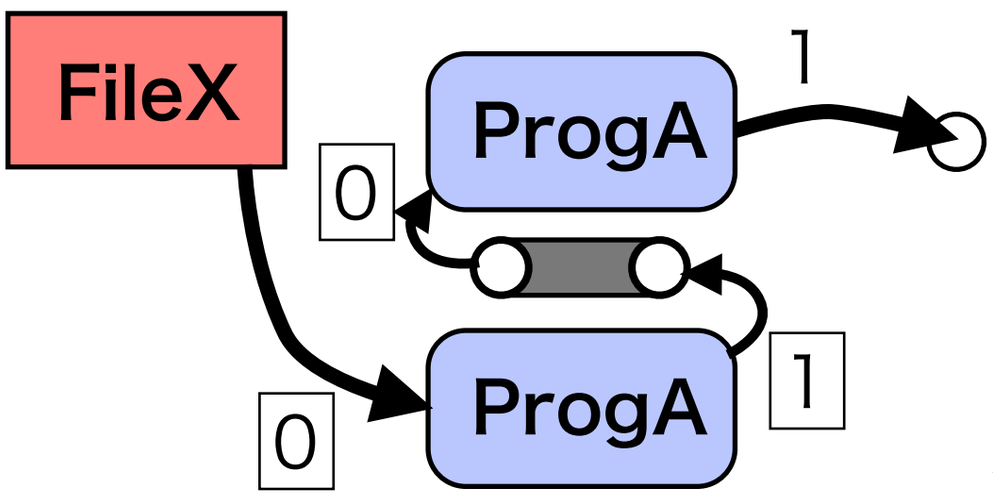


パイプの概要 (2)

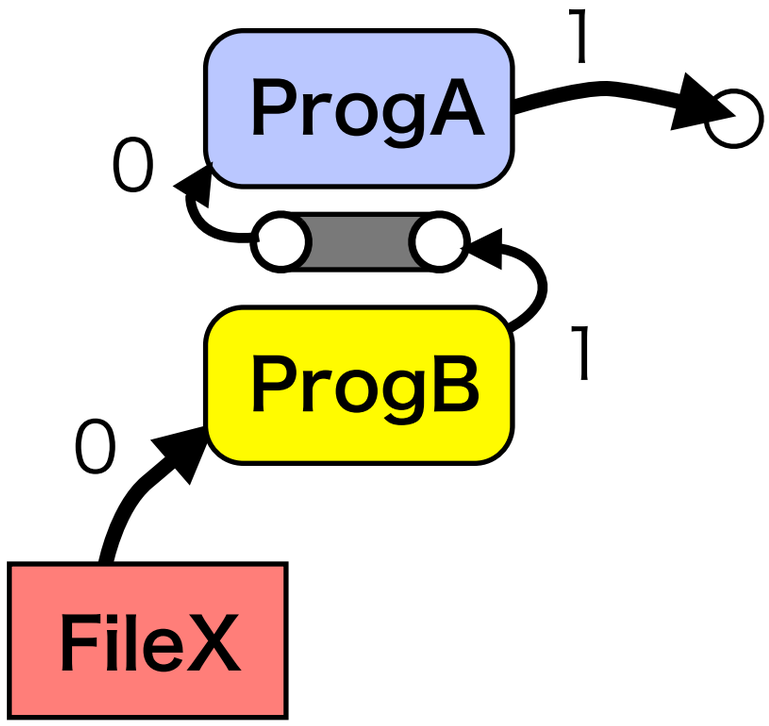
(4) 使わないディスクリプタをクローズ



(5) dup2() でディスクリプタを複製



(6) execl() で別のプログラムを
実行するプロセスになる

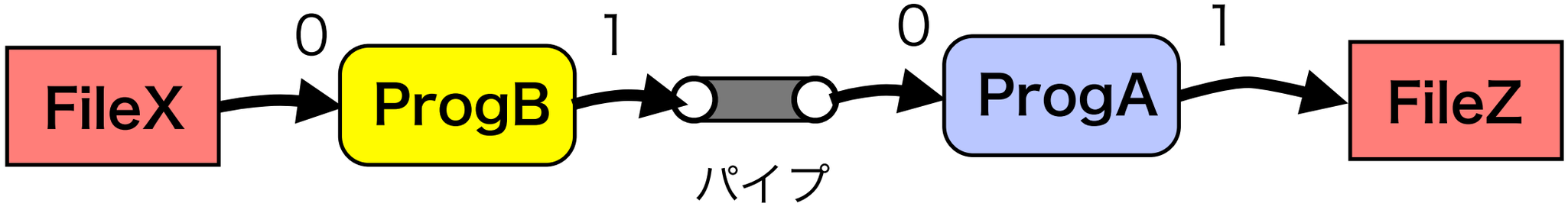


※ 似た動作は popen() 関数でも実現できる

パイプとシェル

- フィルタ：あるコマンドの出力を別のコマンドが入力として処理する
  - ✦ Unixのシェルでは、コマンドラインから容易にパイプを使ったフィルタ処理を実現できる
  - ✦ それ以外の構成をとる場合は、Cでプログラムを記述する必要がある
    - 必要とされることはあまりないので、通常はパイプではなく、一時ファイルへの出力などで対応すれば十分。

```
$ ProgB < FileX | ProgA > FileZ
```



パイプに関係するシステムコール

`int pipe(int fildes[2])`

- ２つのファイルディスクリプタを持つ配列が引数
- fildes[0] に書き込んだデータが fildes[1] から読み出されるようにパイプが構成される

`int dup2(int fildes, int fildes2)`

- ディスクリプタ fildes が示すファイルに関する情報を複製して、その情報にディスクリプタ fildes2 を使ってアクセスできるようにする。
- 例えばディスクリプタ fd が出力用ファイルに対してオープンしていたとする。ここで dup2(fd, 1) とすると、それ以降、標準出力(1)を使って同じファイルに書き込みができる。

パイプの簡単な例

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

#define Read    0
#define Write   1

int createChild(int fd)
{
    int pipefds[2], pid;

    if (pipe(pipefds) < 0) // パイプを生成
        return -1;
    if ((pid=fork()) < 0) { // プロセスを生成
        close(pipefds[Read]);
        close(pipefds[Write]);
        return -1;
    }
    if (pid==0) { // Child
        dup2(fd, 0);                // 元のディスクリプタを０に置き換える
        close(fd);                  // 元のディスクリプタはここで不要に
        dup2(pipefds[Write], 1);    // パイプの書き込み側を１に置き換える
        close(pipefds[Write]);      // パイプの書き込み側はここで不要に
        close(pipefds[Read]);       // パイプの読み込み側は不要
        if (execl("/usr/bin/tr", "tr", "a-z", "A-Z", NULL) < 0) {
            close(pipefds[Read]);
            close(pipefds[Write]);              // プログラムを切替える
            _Exit(1);
        }
    }
    // Parent
    close(fd);                  // 元のディスクリプタは不要
    dup2(pipefds[Read], 0);     // パイプの読み込み側を0に置き換える
    close(pipefds[Read]);       // パイプの読み込み側はここで不要に
    close(pipefds[Write]);      // パイプの書き込み側は不要
    return 1; // success
}
```

```
int main(int argc, const char *argv[]) {
    int fd, ch, status;

    if (argc <= 1 || (fd = open(argv[1], O_RDONLY)) < 0)
        return 1;
    if (createChild(fd) < 0)
        return 2;
    while ((ch = getchar()) != EOF) {
        putchar(ch);
    }
    wait(&status);
    return 0;
}
```

コマンドラインの引数として指定したテキストファイルをオープンし、子プロセスとして起動したフィルタコマンド tr で小文字をすべて大文字に変換した結果をパイプを経由して標準入力から１文字ずつ読み込んで出力する