open3 のはなし

田中哲
独立行政法人 産業技術総合研究所 情報技術研究部門
東京Ruby会議03
2010-02-28

目的

- プロセスを動かす良い方法を提供する
- 使いやすい API をデザインする

プロセス起動の用途

- 出力を less 経由でユーザに見せる
- エディタを起動してユーザになにか入力させる
- lpr を起動してプリントアウト
- 大きなデータを sort でソート
- w3m で HTML を表示
- feh で画像を表示

Ruby のプロセス起動の問題

どちらの道も罠だらけ

解決

- spawn メソッドの新設
- open3 ライブラリの拡張

プロセス

- 「プロセスとは、情報処理においてプログラムの動作中のインスタンスを意味し、全ての変数やその他の状態を含む。」(Wikipedia より)
- ps コマンド出力の 1行
- さまざまな属性を持つ
  - プロセスID
  - メモリ空間
  - ファイルディスクリプタ
  - カレントディレクトリ
  - リソースリミット
  - etc.

プロセス属性
カレントディレクトリ

- 相対パス解釈の起点
- シェルによる指定
  - cd /usr
  - cd 後は bin が /usr/bin を意味する

プロセス属性
リソースリミット

- プロセスが使う資源の制約
  - core ファイルのサイズ
  - CPU時間
  - データサイズ
  - etc.
- シェルによる指定
  - core dump 禁止: ulimit -c 0
  - ulimit 後に起動するプロセスに適用される

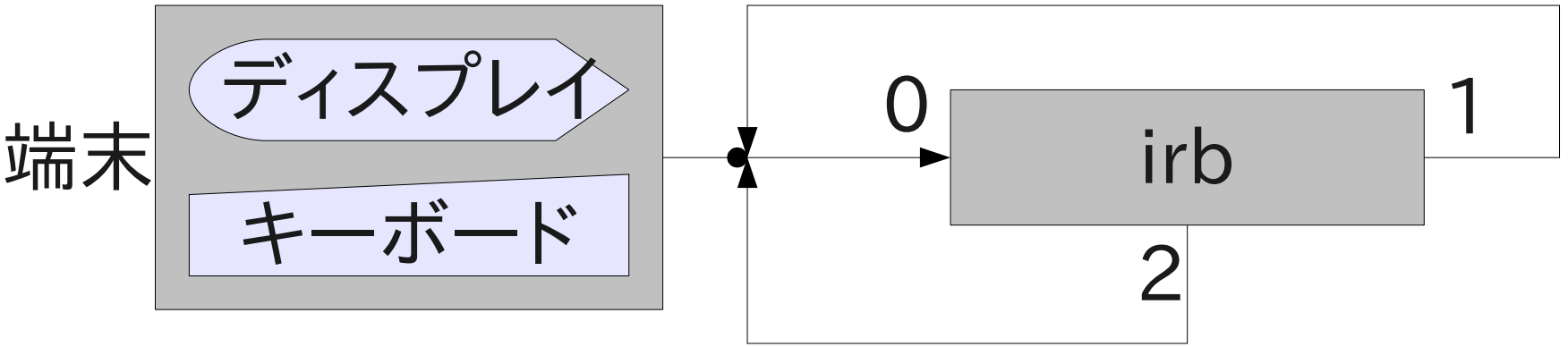
プロセス属性
ファイルディスクリプタ

- file descriptor (fd)
- プロセスがファイル等にアクセスするための番号
- ファイルを open すると fd が割り当てられる
- fd を使って read/write する
- 使い終わったら close する

```
fd = open(filename, flags);
ret = read(fd, buf, buflen);
close(fd);
```

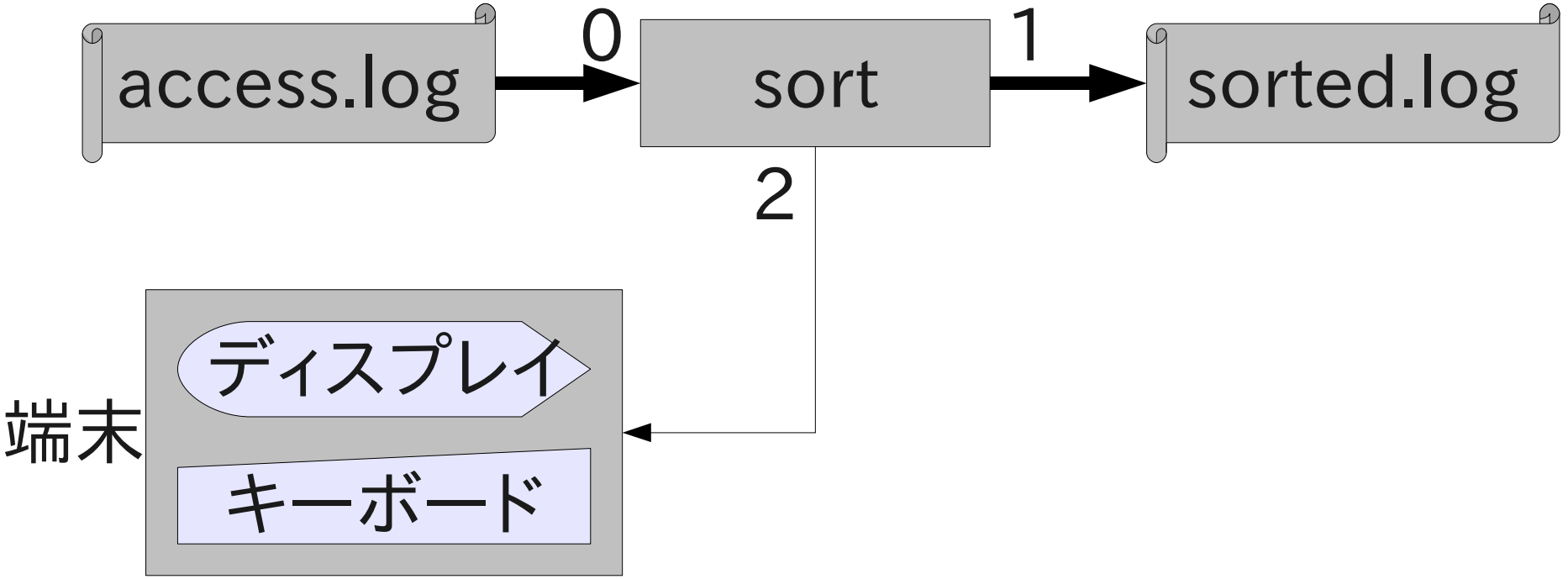
標準入出力

- fd の 0, 1, 2番は標準的な用途がある
  プロセスの外から与えられ open/close 不要
  - 0: 標準入力
  - 1: 標準出力
  - 2: 標準エラー出力
- 0, 1, 2 は基本的には端末につながる
  - 端末: xterm, VT100 等



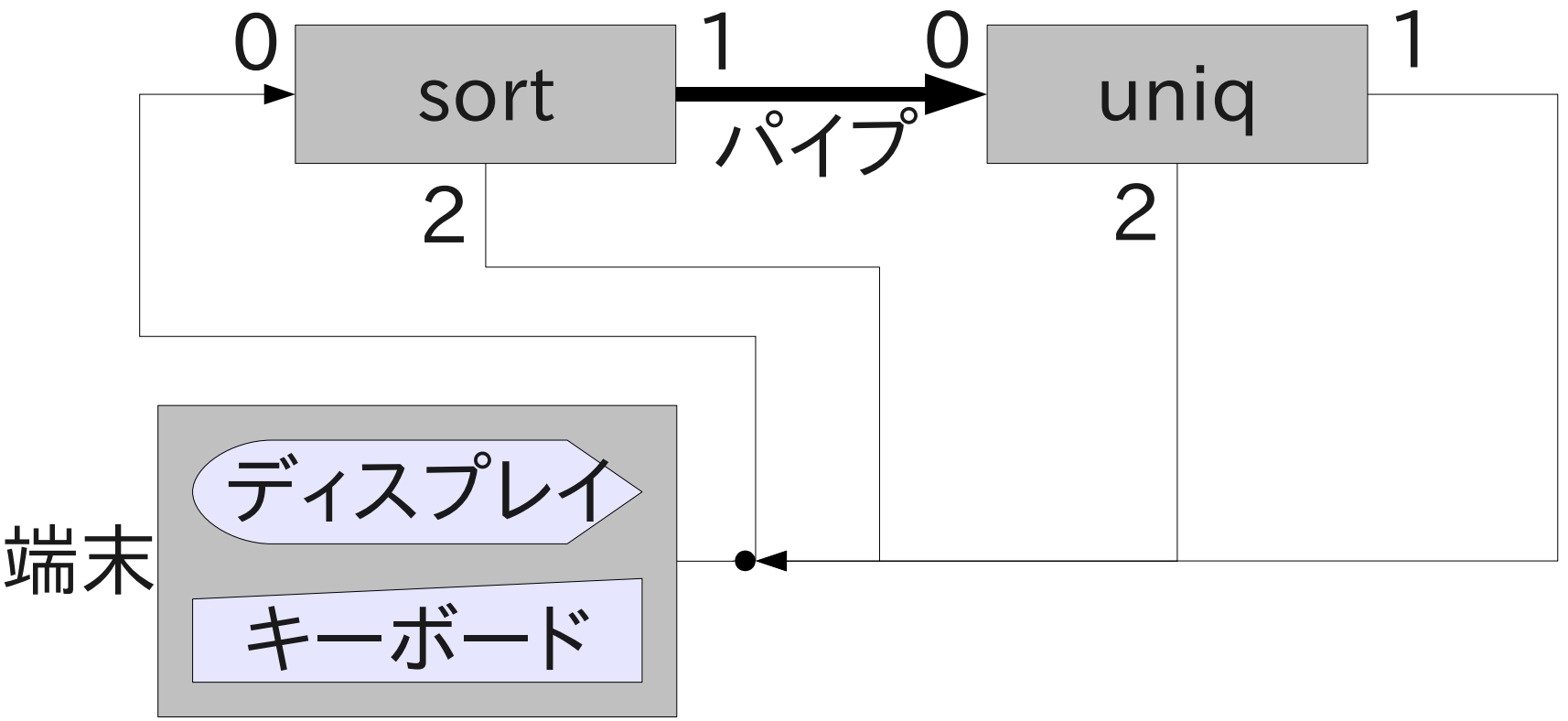
リダイレクト

- プロセスの標準入出力をファイルにつなぎかえる
- シェルの記法: sort < access.log > sorted.log



パイプ

- プロセスとプロセスをつなぐ
- シェルの記法: sort | uniq



Unix のプロセス起動

- fork
  fork を呼び出したプロセスの複製を作る
- exec
  exec を呼び出したプロセスを指定したコマンドで置き換える
- fork と exec の組み合わせでコマンド起動
- fork と exec の間にプロセス属性を変更
  リダイレクト、パイプ等を実現

非Unix のプロセス起動

- fork と exec が分かれているのは Unix の特徴
- 非 Unix では分かれていない
- プロセス属性を指定する引数が複雑になりがち

Ruby で実現したいこと

- OS が提供しているプロセス起動機能を使いたい
- シェルでできることを Ruby でもやりたい
- ポータブルに記述可能であってほしい
  (Unix と非 Unix で場合分けしたくない)
- できることはなんでも可能であってほしい
- よくやることは簡単であってほしい

API 案
既存のメソッドを拡張する?

- fork
  そう簡単には非Unix では動きそうにない
  ポータブルにはならない
- system
  プロセスを待たないことを可能にするのは奇妙
  なんでも可能にするには適切でない
- IO.popen
  パイプを作らないことを可能にするのは奇妙
  なんでも可能にするには適切でない
- `command`
  構文上さまざまな指定を付加できない

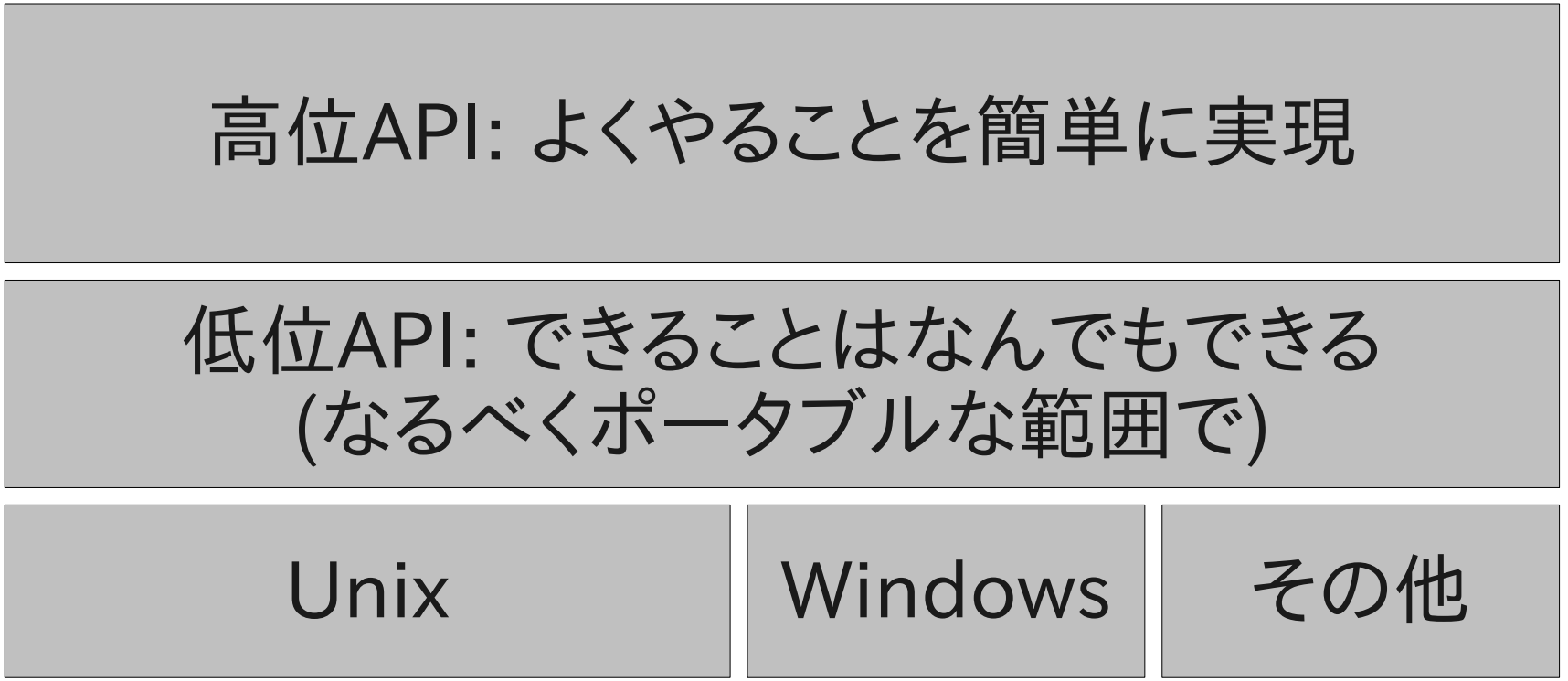
うまくいきそうにない

要求の衝突

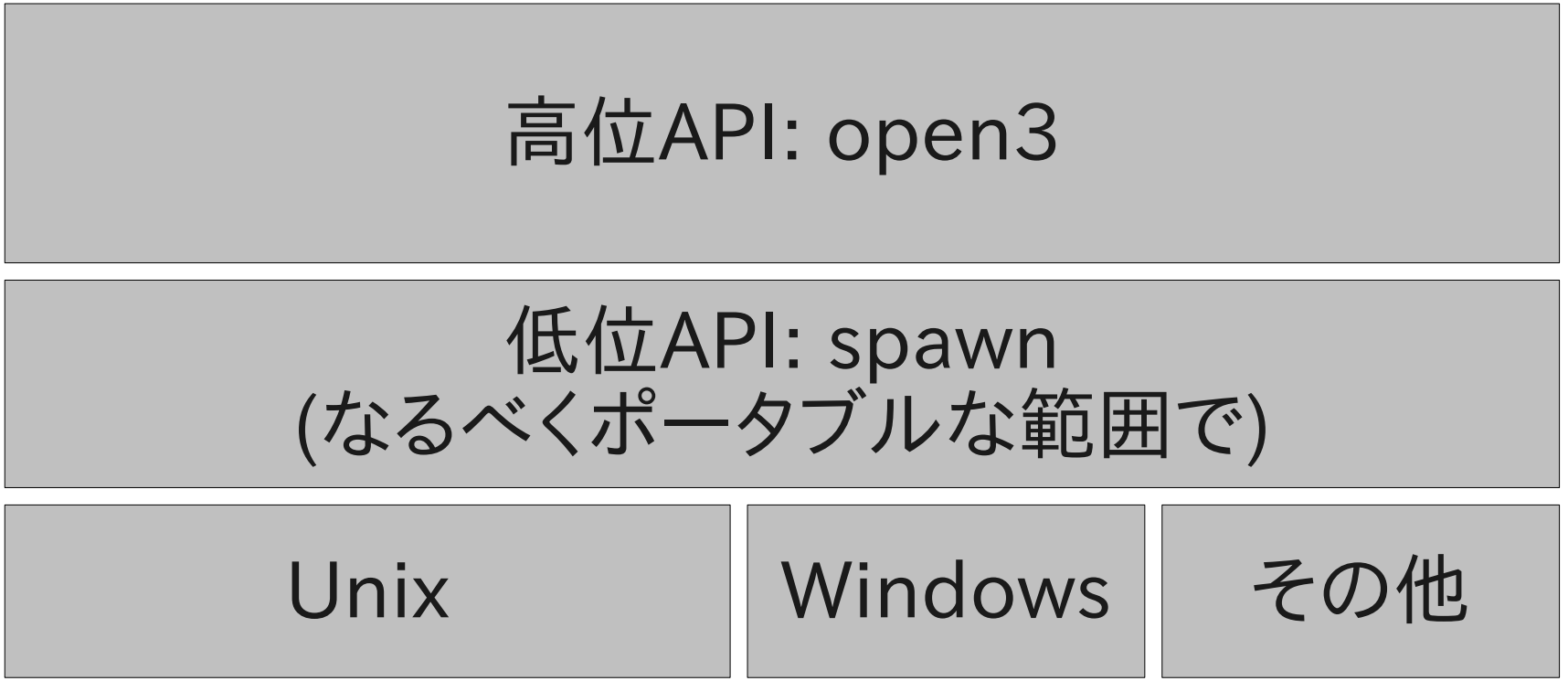
- よくやることを簡単にできるのが欲しい
  - プロセスの終了を待ってほしい
  - wait くらい自動的にしてほしい
  - パイプラインをいっきに作ってほしい
    例: zcat ruby.1.gz | nroff -man | less
- なんでも可能なのが欲しい
  - プロセスの終了を待たないことが可能
  - 呼出側が Process.wait メソッドで待つことが可能

ひとつのメソッドですべての要求を満たすのは困難

高位･低位API へ分割


高位API: よくやることを簡単に実現
低位API: できることはなんでもできる
(なるべくポータブルな範囲で)
Unix
Windows
その他

高位･低位API へ分割



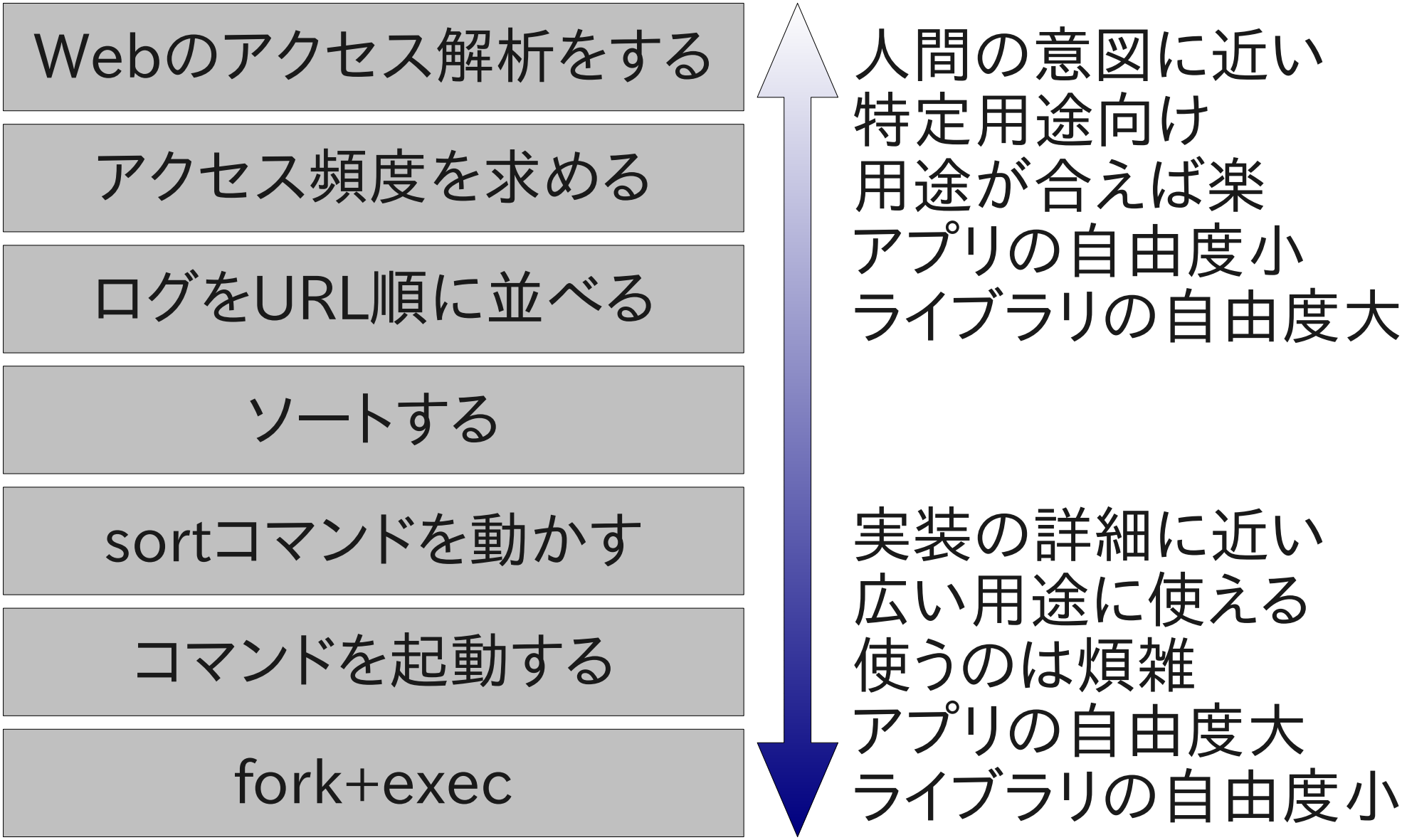
高位API: open3

- とりあえずシェルのパイプラインくらいまで提供
  例: zcat ruby.1.gz | nroff -man | less
- シェルの嫌なところは避ける
  - エスケープは不要にする
  - パイプライン内のすべてのコマンドのステータスを得る
- パイプラインよりも複雑な用途は将来の課題

低位API: spawn

- 「なんでも」って結局なにができればいい?
- fork では子プロセスでコードを動かせる
  でも fork はポータブルでない
- exec 抜きの fork はポータブルでないので諦める
- exec するならコマンド起動
- 起動したプロセスの属性を指定できればいい
  - カレントディレクトリ
  - リソースリミット
  - リダイレクト
  - etc.

想定されるユースケースの例



ライブラリのレイヤ

- どのレイヤのライブラリを提供するか
- 高位になるほどライブラリの自由度が高く「賢い」
  - サイズで sort メソッドと sort コマンドを切り替える
  - Unix では fork+exec, Windows では別の方法
- 低位になるほど詳細な制御が可能になる
  - 絶対 sort メソッドを使いたい
  - exec 抜きで fork だけ使いたい

open3 と spawn のレイヤ

- open3
  - OS 間で共通の機能にフォーカスした高位 API (標準入出力･パイプ)
  - 複数のコマンドをいっきに起動できるくらいに高位
  - コマンドとコマンド以外を抽象化するほど高位ではない
- spawn
  - OS 固有の機能も可能な限り指定できる程度に低位 (リソースリミットなど Unix 固有のプロセス属性)
  - OS 間で共通の機能については OS の違いを気にしないで済む程度に高位

spawn について

spawn

- Ruby における今までのプロセス起動
- spawn とはどういうものか
- なぜそういう仕様になったのか

Ruby のプロセス起動

- `command`
- system(command)
- IO.popen(command, mode)
- exec(command)
- fork { ... }

プロセス起動法の起源

- `command`　perl, shell
- system(command)　perl, C
- IO.popen(command, mode)　perl, C
- exec(command)　perl, C
- fork { ... }　perl, C

不満

- シェルを使いたくないこともある
- コマンドの終了を待ちたくないこともある
- 標準入出力をつなぎ変えたいこともある
- リソースリミットを設定したいこともある
- その他いろいろ

プロセス起動の問題

- `c`　　同期的・常にシェル経由
- system(c)　　同期的・リダイレクトできない
- IO.popen(c)　　標準エラー出力を扱えない
- exec(c)　　プロセス生成に fork が必要
- fork { ... }　　Windows・NetBSD 4 で動かない

シェルの機能を使えば問題を軽減できるが、シェル経由になる上にポータブルでない

ジェネリックなプリミティブがない

perl の解決法

- Windows では fork のエミュレーションを行う
- 別スレッドで動く別インタプリタ
- 個々にカレントディレクトリを持てるようエミュレート
- ほかにもいろいろエミュレート
- エミュレートできない部分は諦める

ruby の解決法

- spawn 関数の導入
- fork + プロセス属性設定 + exec
- プロセス属性
  - カレントディレクトリ
  - 環境変数
  - 標準入力･出力･エラー
  - など

spawn の基本

```
pid = spawn("make all")
Process.wait pid
```

- コマンドラインを与える
  概念的にはシェル経由
- プロセスの終了は待たない
- プロセスIDを返す
  Process.wait で終了を待てる

シェルを経由しないコマンド起動

spawn("make", "all")

- 引数を分けて与えるとシェルを経由しない

spawn(["make", "make"], "all")

- 最初の引数を配列にすると argv[0] も指定できる
- この形式ではコマンド引数のない場合もシェルを回避できる
  spawn(["make", "make"])

リダイレクト

spawn("make all", :out => "make.log")

- 標準出力をリダイレクトできる

spawn("make all", :err => :out)

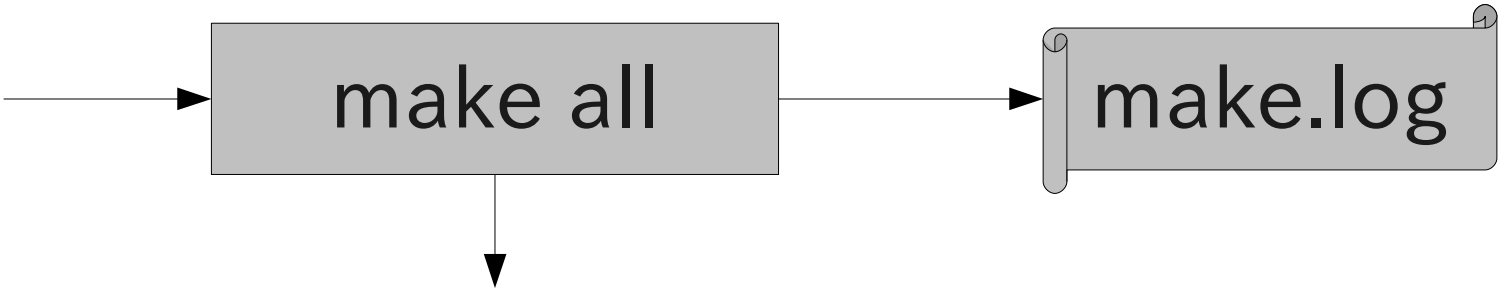
- 標準エラー出力を標準出力にマージできる

spawn("make all", :out => :err, :err => :out)

- 標準エラー出力と標準出力を入れ替えられる

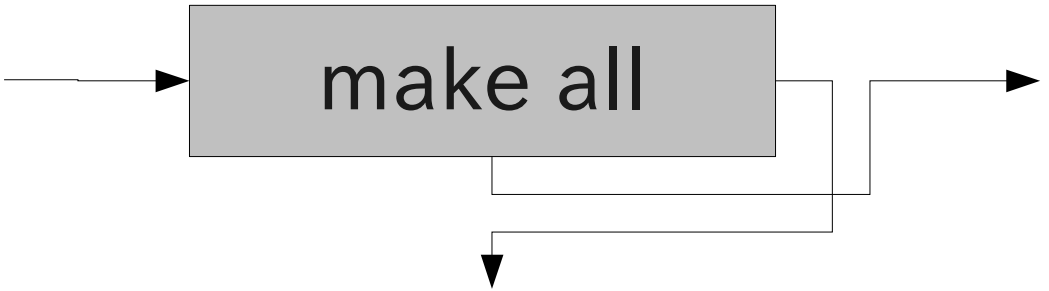
リダイレクト

spawn("make all", :out => "make.log")



spawn("make all", :err => :out)

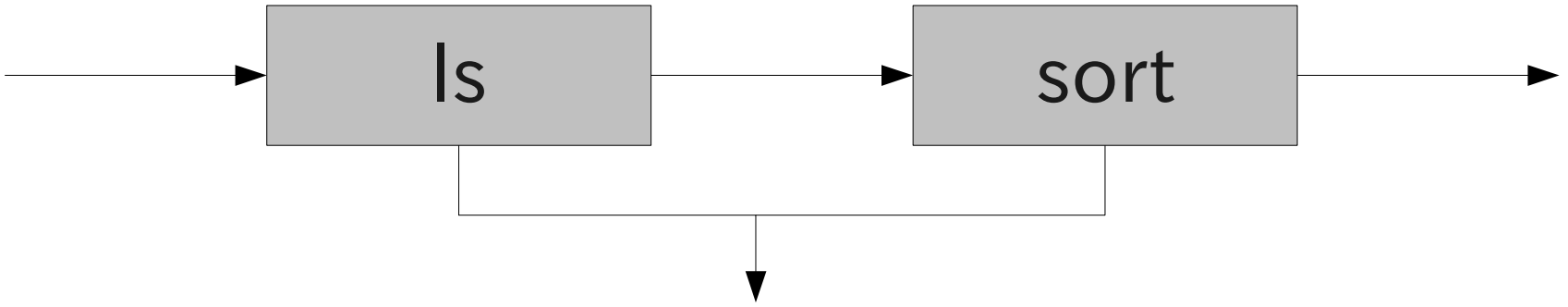
spawn("make all", :out => :err, :err => :out)



パイプ

```
IO.pipe {|r, w|
  spawn("ls", :out => w)
  spawn("sort -r", :in => r)
}
```

- パイプでコマンドをつなげられる



環境変数

spawn({"LC_ALL"=>"C"}, "ls -l")

- 環境変数を指定できる

spawn("ls -l", :unsetenv_others=>true)

- 指定しなかった環境変数をクリアできる
- この例では環境変数が空になる

その他いろいろ

spawn("ls", :chdir => "/usr/bin")

- カレントディレクトリの指定

spawn("make all", :rlimit_core => 0)

- リソースリミットによる core dump の抑制

spawn("ps jaxww", :pgroup=>true)

- 新しいプロセスグループにする

spawn の一般形

spawn(env, command, option)

- env はハッシュによる環境変数の指定 (省略可能)
  - {..., "name"=>"value", ...}
  - {..., "name"=>nil, ...}　　特定の環境変数を除去
- command はコマンド
  - "command line"　　シェル経由
  - "command", "arg1", ...　　1引数以上・非シェル
  - ["command", "arg0"], "arg1", ...　　0引数以上・非シェル
- option はハッシュによるその他の指定 (省略可能)

option (fd以外)

- :unsetenv_others => bool
- :pgroup => true, pgid, nil
- :rlimit_foo => limit or [cur, max]
- :chdir => path
- :umask => int

option (fd : file descriptor)

子プロセスにおける fd 設定の指定

- 子プロセスの fd => 親プロセスの fd
- 子プロセスの fd => ファイル名
- 子プロセスの fd => [ファイル名, mode, perm]
- 子プロセスの fd => [:child, 子プロセスの fd]
- 子プロセスの fd => :close
- :close_others => bool
  3番以降で指定しなかった fd を close するか
  spawn でのデフォルトは true

fd の指定

- 整数
- IO オブジェクト (STDIN とか)
- :in　　0 と同じ
- :out　　1 と同じ
- :err　　2 と同じ
- 上記を複数まとめた配列 ([:out, :err] とか)

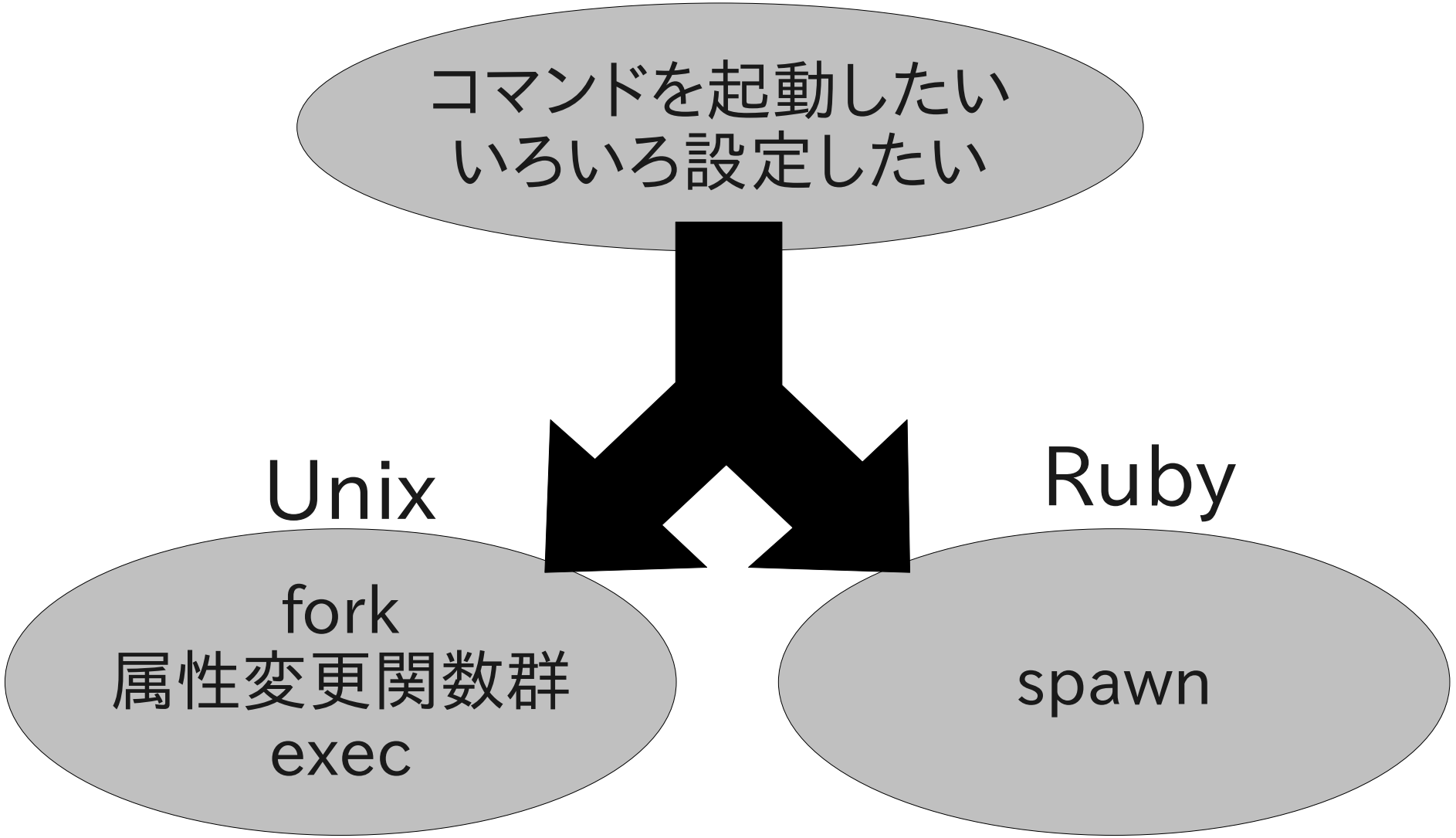
system, exec, IO.popen の拡張

- system, IO.popen, exec も spawn 同様のオプションを使える
- system(env, command..., option)
- exec(env, command..., option)
- IO.popen([env, command, option], mode)

spawn のデザイン

- こんなに多機能なものを単一関数にしていいのか?
- 簡単なことが難しくなっていないか?
- 将来の拡張に耐えられるか?
- fd の挙動は適切か?

こんなに多機能なものを単一関数にしていいのか？



想定される聴衆

- level 1: C言語を知っている
- level 2: fork と exec なら大学で習った
  (shell を作る実習とか)
- level 3: その類の試験なら 100点な自信がある
- level 4: fork/exec と thread の組合せで痛い目にあったことがある
- level 5: fork/exec は見限って posix_spawn に興味がある

fork

- Unix で新しいプロセスを作るシステムコール
- プロセスの複製を作る

複製されるもの

ファイルディスクリプタ・close-on-execフラグ・シグナル処理の設定・メモリ空間・uid・euid・suid・gid・egid・sgid・プロセスグループID・セッション・制御端末・カレントディレクトリ・ルートディレクトリ・umask・リソースリミット・環境変数・nice・スケジューラクラス・forkを呼び出したスレッド

複製されないもの

pid・ppid・リソース使用量・tms構造体のプロセス時間・処理待ちのシグナル・非同期入出力・forkを呼び出した以外のスレッド

だいたいは複製される

exec

- プロセスを他の実行ファイルに切り替えるシステムコール
- 実行ファイルのパスと引数を指定する
- 成功すると制御は戻ってこない

コマンド実行 = fork + exec

- コマンドを実行するには fork と exec を使う
- fork で作った子プロセスで exec する

fork と exec が別になっている理由

- Unix で古くからある疑問
- いっきにやってしまうシステムコールの方が便利なんじゃないの?
- fork と exec の間にやりたいことがあるから
  - リダイレクト
  - パイプ
  - 権限の放棄
  - カレントディレクトリの移動
  - など

Ruby で fork を避ける理由

- Ruby の fork は NetBSD 4 で動かないから
- NetBSD 4 で fork した子プロセスではスレッドが動かないから
- Ruby 1.9 はタイマースレッドというスレッドを常に必要とするから
- 結果として、NetBSD 4 では子プロセスで Ruby のコードを動かせない
- POSIX では、fork した子プロセスでは exec するまでは async-signal-safe な関数しか保証されない
- なお Windows という理由もある

fork がなければどうするか

- fork + 属性変更 + exec をひとまとめで提供
- 似たような話
  - POSIX: posix_spawn
  - Windows: spawn
  - Windows: CreateProcess

posix_spawn

- POSIX (ADVANCED REALTIME)
- 引数はかなり複雑

```
int posix_spawn(pid_t *restrict pid, const char *restrict path,
        const posix_spawn_file_actions_t *file_actions,
        const posix_spawnattr_t *restrict attrp,
        char *const argv[restrict], char *const envp[restrict]);
```

- 引数を操作する関数:
  posix_spawnattr_*() が 14個
  posix_spawn_file_actions_*() が 5個

簡単なことが難しくなっていないか?

- posix_spawn は興味のない引数も指定する必要がある

```
int ret;

pid_t pid;

char *args[3] = { "/bin/ls", "/usr", NULL };

ret = posix_spawn(&pid, "/bin/ls", NULL, NULL, args, envp);
```

- spawn では余計なことは指定しなくていい
  spawn("/bin/ls /usr")
- キーワード引数に感謝

将来の拡張に耐えられるか？

- たとえば、現在 nice 値を指定できない
- キーワード引数により、互換性を保って拡張できる
  spawn("make all", :nice => 10)

fdの挙動は適切か？

- シェルや posix_spawn のようなリダイレクトの操作列でなく、最終的な状態を指定する
  - make > log 2>&1
  - spawn "make", [:out, :err] => "log"
- デフォルトでは 3番以降の fd を継承しない
  - Unix では継承するのが普通
  - 継承するには明示的に指定する
    r, w = IO.pipe
    spawn("valgrind", "--log-fd=#{w.fileno}", w=>w)

リダイレクトの記述法

標準出力と標準エラーを入れ替える:

- シェルの記法はリダイレクト操作の列を書く
  - make all 3>&1 1>&2 2>&3 3>&-
  - 3>&1　　dup2(1,3)
  - 1>&2　　dup2(2,1)
  - 2>&3　　dup2(3,2)
  - 3>&-　　close(3)
- spawn では、子と親の fd の関係を書く
  spawn("make all", :out => :err, :err => :out)
  => の左が子のfd, 右が親の fd

POSIX の判断

- posix_spawn では、Ruby の spawn のような形式も検討された
- が、最終的にはシェルのような指定になった
- 理由 (RATIONALE)
  - fd に空きがないとき、実行できない場合がある
  - 複雑な処理が必要

Ruby の(私の)判断

- fd に空きがないとき、実行できない場合がある
  - 子プロセス内でのエラー検知のためパイプを使っている
  - 毒を喰らわば皿まで
- 複雑な処理が必要
  - 私が実装すればいい
  - JRuby とかにはがんばってもらう
    (Unix に比べれば別実装はずっと少ない)

fd を継承しないのがデフォルト

- デフォルトでは 3番以降の fd を継承しない
  - Unix では継承するのが普通
  - 継承するには明示的に指定する
    r, w = IO.pipe
    spawn("valgrind", "--log-fd=#{f.fileno}", w=>w)
  - system, exec はデフォルトで継承する (互換性)
- 理由
  - 継承すると、上の例で r を close する記述が必要
  - 他のスレッドが open した fd を close するのは無理
- 実装は完璧ではないが、だいたい動く
  - そのうち改善するかも

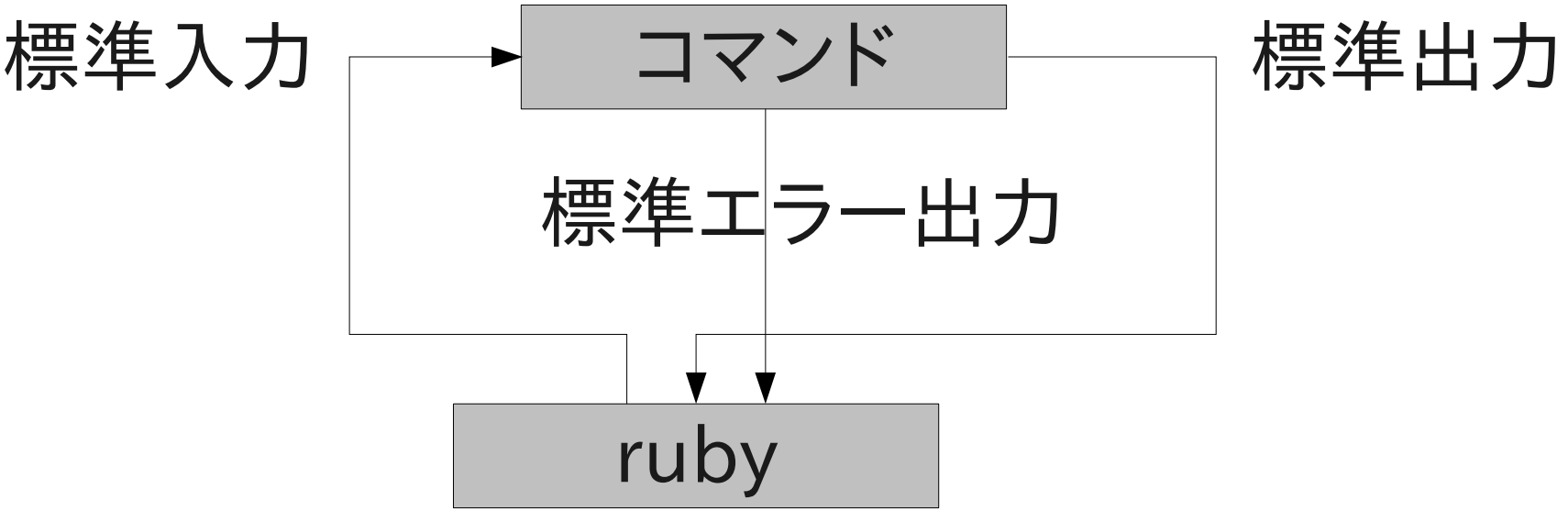
spawn のまとめ

- Ruby の spawn 関数はプロセスを起動するプリミティブ
- fork + プロセス属性設定 + exec
- Ruby 上で fork に触れなくてすむ
- ポータブルでよい

open3 について

open3 ライブラリ

- 標準添付ライブラリ
- 標準入力･標準出力･標準エラー出力の 3つのパイプでコマンドと通信する
- 名前は Perl 由来



open3 の使用例

標準入力　標準出力　標準エラー出力

```
require 'open3'
Open3.popen3("nroff -man") {|i, o, e|
  Thread.start {
    ARGF.each {|line|
      i.print line
    }
    i.close
  }
  o.each {|line|
    print ":", line
  }
}
```

ARGFから読んで
nroff の標準入力に送る

nroff の標準出力を読んで
Rubyの標準出力に送る
ただし行頭に : をつける

Ruby における open3 の歴史

- 1998-09-17 [ruby-list:9587]
  Shoichi OZAWA:「まえに open3 相当のものを作ったので、下につけときます」(31行)
- 2000-10-20 [ruby-list:25538]
  Takaaki Tateishi:「waitを忘れると、ゾンビが大量に発生してしまうことがありました」
  - [ruby-list:25561]
    Akinori MUSHA:「子がさらに fork して孫を作ってすぐに死んでやると、親はすぐに子の亡骸を wait で回収でき、孫は死んだ時点で orphan となるの init に回収されると思います」
    → double fork を行うように変更

今日の open3 (Ruby 1.8)

- 2010-02-28 15:11:11 nobu r26783
  - lib/open3.rb (Open3#popen3): use Thread.detach instead of double-fork, so that the exit status can be obtained.
- 2010-02-28 15:12:40 nobu r26784
  - lib/open3.rb (Open3#popen3): ignore trap and at_exit also when exec failed. [ruby-dev:30181]

発表前にコミットするメソッド by nobu?

ゾンビプロセス

- 終了した後、親により wait されていないプロセス
- 終了ステータスを保持している
- 親が wait でステータスを得ると、プロセステーブルから消される
- 親が wait しないで死ぬと、子は init の養子になる
- init は常に wait しまくってゾンビを除去している
- double fork は init を意図的に利用してゾンビを避けるテクニック

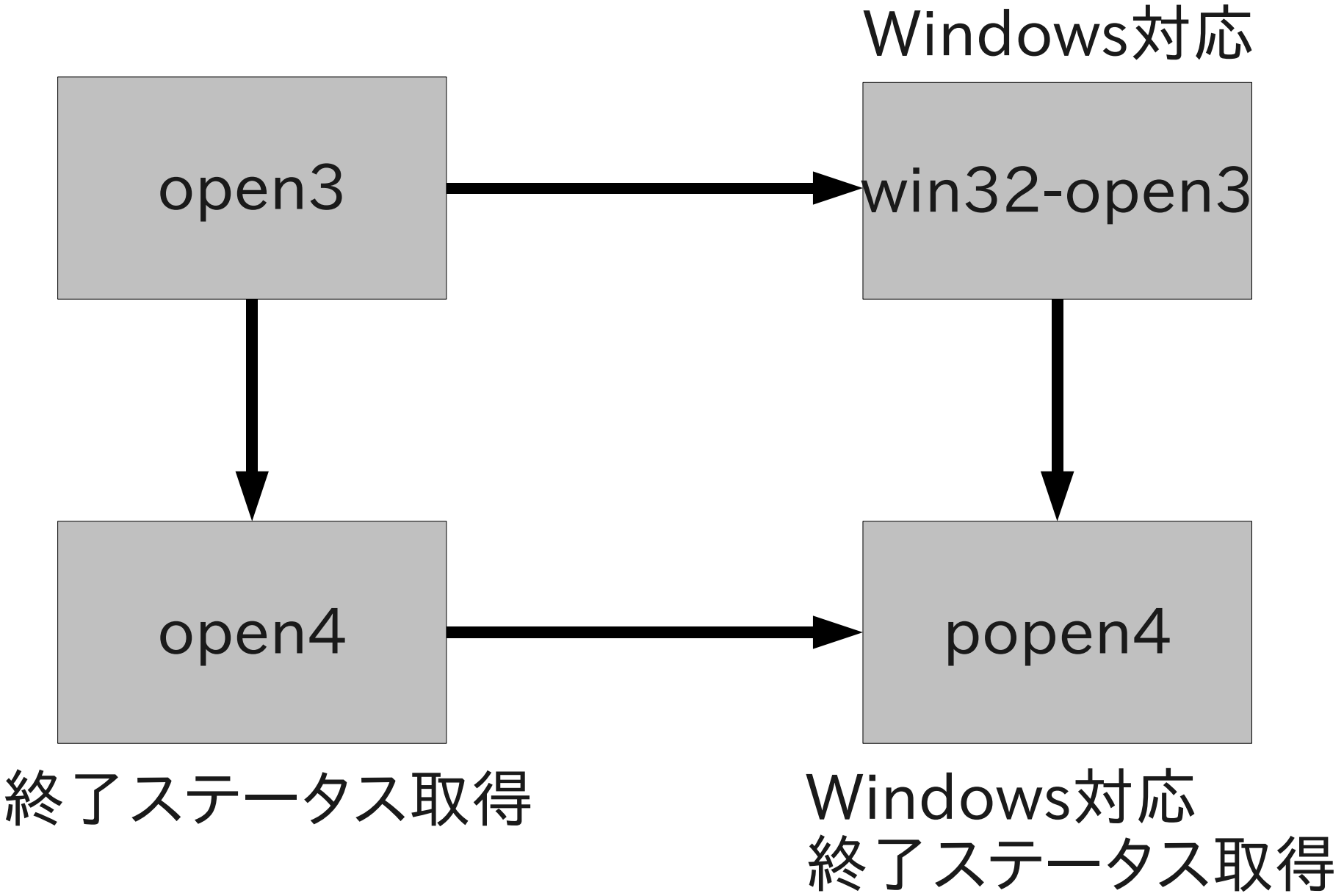
open3 でいわれる問題

- ゾンビが発生する → double fork で解決済
- Windows で動かない
  fork メソッドを使っているから
- 終了ステータスを取得できない
  double fork を行っているから
- pid が得られない
  シグナルを送るのが困難
  double fork を行っているから

open3 の類の乱立

- win32-open3 (Daniel J. Berger, 2004)
- open4 (ara.t.howard, 2005)
- popen4 (John-Mason P. Shackelford, 2006)

機能の違い


Windows対応
open3
win32-open3
open4
popen4
終了ステータス取得
Windows対応
終了ステータス取得

Open3.popen3 の拡張
spawn による解決

- Open3.popen3(command) {|i, o, e| ... }
  →
  Open3.popen3(command) {|i, o, e, t| ... }
- t は command の終了を待つスレッド
  内部で呼んだ Process.detach の返り値
- fork でなく spawn を使う
- command の部分は spawn に渡されオプションも使用可能

Process.detach

- プロセスを wait するスレッドを作る
- Ruby による擬似実装:

```
def Process.detach(pid)
    t = Thread.new { Process.wait(pid); $? }
    t[:pid] = pid
    def t.pid() t[:pid] end
    t
end
```

- スレッドは終了ステータスを値として終わる
  t.value: コマンドの終了を待ってステータスを得る
- t.pid でプロセスID が得られる
  Process.kill(:INT, t.pid): SIGINT 送信

open3 における detach の効果

- ゾンビプロセスが発生しない
  detach が起動したスレッドが wait してくれる
- double fork もしないので pid が得られる
  pid メソッドを使う
- 終了ステータスも得られる
  スレッド終了時の値がステータスになる

open3 の使用例 (pid, ステータス)

```
require 'open3'
Open3.popen3("nroff -man") {|i, o, e, t|
  p t.pid
  Thread.start {
    ARGF.each {|line|
      i.print line
    }
    i.close
  }
  o.each {|line|
    print ":", line
  }
  p t.value
}
```

pid表示

waitスレッド

終了ステータス表示

Open3.popen3 の解決

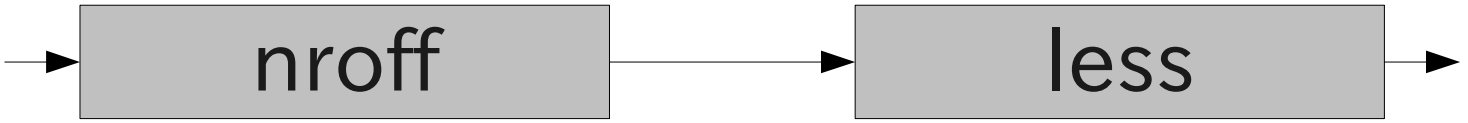
- ゾンビが発生する
  Process.detach により解決
- Windows で動かない
  fork でなく spawn を使うことにより解決
- 終了ステータスを取得できない
  double fork を避けて Process.detach で解決
- pid が得られない
  double fork を避けて Process.detach で解決

spawn のオプションも使用可能

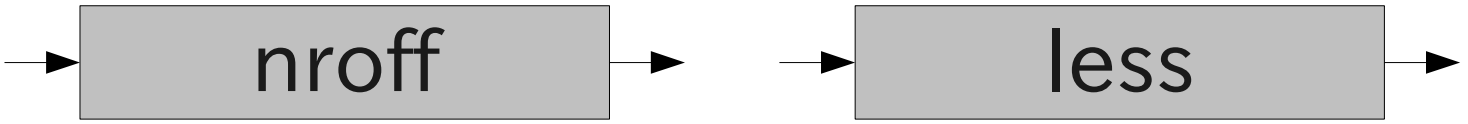
- カレントディレクトリを変える
  ```
  Open3.popen3("pwd", :chdir=>"/") {|i,o,e,t|
      p o.read.chomp        結果は "/"
  }
  ```
- シェルを避けることも可能
  ```
  Open3.popen3("echo", "a") {|i, o, e, t| ... }
  Open3.popen3(["echo", "argv0"], "a") {|i, o, e, t| ... }
  ```

Open3.popen3 だけでは足りない 高位API の必要性

- 出力を文字列で得たい
  標準出力と標準エラー出力を両方適切に読むのは難しい
- 標準出力と標準エラー出力をマージできない
- パイプラインを作れない
  nroff -man | less
  nroff から less へ流れる 1本のパイプが欲しい

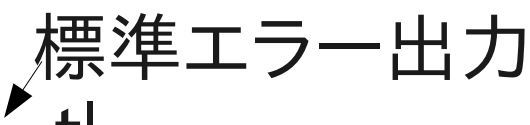


open3 で個々にコマンドを起動するとパイプが余る



open3 と標準エラー出力

```
require 'open3'
Open3.popen3("nroff -man") {|i, o, e, t|
  Thread.start {
    ARGF.each {|line|
      i.print line
    }
    i.close
  }
  o.each {|line|
    print ":", line
  }
  p t.value
}
```



- 標準エラー出力 e を読んでいない
- nroff がエラーをたくさん吐くとパイプが詰まる
- パイプが詰まると nroff が終了しない

標準エラー出力の扱い

- 標準出力と別々のパイプにするのは扱いが難しい
  - 適切に読むにはスレッドか select を使う
- 標準エラー出力はパイプにしないほうがいいことも多い
  - 親から継承する
  - 典型的には端末にエラーメッセージが出る
- ひとつのパイプにマージしていいケースもある
  - Cシェルの >& や |& という記述に対応
  - ひとつのパイプなら詰まらせずに読むのは簡単

高位API のメソッド追加

- 出力を文字列で得る
  os, es, st = Open3.capture3("nroff -man")
- 標準出力と標準エラー出力をマージ
  Open3.popen3("nroff -man") {|i, o, e, t| } →
  Open3.popen2e("nroff -man") {|i, oe, t| }
- パイプライン
  Open3.pipeline("nroff -man", "less")

open3 のメソッド

- Open3.popen3
- Open3.popen2
- Open3.popen2e

- Open3.capture3
- Open3.capture2
- Open3.capture2e

- Open3.pipeline_rw
- Open3.pipeline_r
- Open3.pipeline_w
- Open3.pipeline_start
- Open3.pipeline

open3 が提供する関数

- Open3.popen*
  コマンドをひとつ起動する
  IO.popen に類似
- Open3.capture*
  コマンドをひとつ起動して、結果を文字列で得る
  `command` に類似
- Open3.pipeline*
  複数のコマンドからなるパイプラインを起動する

Open3.popen*

- Open3.popen3(command) {|i, o, e, t| ... }
- Open3.popen2(command) {|i, o, t| ... }
  - 標準エラー出力は変えない
  - IO.popen(command, "r+") に類似
- Open3.popen2e(command) {|i, oe, t| ... }
  - 標準出力と標準エラー出力をマージする
  - Cシェルの >& や |& のような機能

Open3.capture*

- 文字列で入力を与え、文字列で出力を得る
- outstr, errstr, status = Open3.capture3(command, :stdin_data=>"")
- outstr, status = Open3.capture2(command, :stdin_data=>"")
  - 標準エラー出力は変えない
  - `command` に類似
- outerrestr, status = Open3.capture2e(command,:stdin_data=>"")
  - 標準出力と標準エラー出力をマージする
  - `command 2>&1` に類似

Open3.pipeline_rw

- Open3.pipeline_rw(c1,c2,...,[opts]) {|i, o, ts| }
  - Open3.pipeline_rw("zcat", "nroff -man") {|i, o, ts| }
  - c1 | c2 | ... というパイプラインの生成
  - i は最初のコマンドの標準入力へ書き込むパイプ
  - o は最後のコマンドの標準出力から読み出すパイプ
  - ts は各コマンドに対応する waitスレッドの配列
  - 個々のコマンドは文字列もしくは配列で spawn 同様
    - "zcat ruby.1.gz"
    - ["zcat", "ruby.1.gz"]
      引数を個々に指定 (エスケープ不要)
    - ["zcat ruby.1.gz", :chdir=>"/"]
      オプションも使用可能

Open3.pipeline*

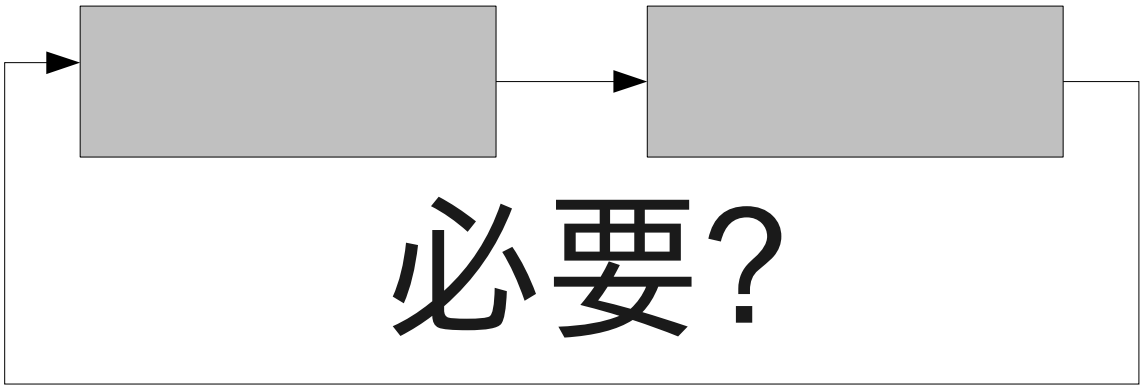
- Open3.pipeline_r(c1,c2,...,[opts]) {|o, ts| }
  - 最初のコマンドの標準入力は変えない
- Open3.pipeline_w(c1,c2,...,[opts]) {|i, ts| }
  - 最後のコマンドの標準出力は変えない
- Open3.pipeline_start(c1,c2,...,[opts]) {|ts| }
  - 最初のコマンドの標準入力は変えない
  - 最後のコマンドの標準出力は変えない
- Open3.pipeline(c1, c2,...,[opts])
  - ブロックをつけなくても終了を待つ
  - 終了ステータスの配列を返す

Open3 が解決したこと

- 標準出力と標準エラー出力を両方適切に読むのは難しい (不適切に読むとパイプが詰まる)
  Open3.capture* により解決
- 標準出力と標準エラー出力をマージできない
  Open3.*2e で解決
- パイプラインを作れない
  Open3.pipeline* で解決

複数コマンドの起動

- Open3.pipeline*
- 直線的なパイプラインしか作れない
- その制約は合理的か？



- そもそも複数コマンドの起動を支援すべきか？

spawn によるパイプラインは厄介

- zcat ruby.1.gz | nroff -man | less
- spawn による実装

```
r1, w1 = IO.pipe
pid1 = spawn("zcat ruby.1.gz", :out => w1)
w1.close
r2, w2 = IO.pipe
pid2 = spawn("nroff -man", :in => r1, :out => w2)
r1.close
w2.close
pid3 = spawn("less", :in => r2)
r2.close
[pid1, pid2, pid3].each {|pid| Process.wait pid }
```

- Open3.pipeline による実装

```
Open3.pipeline("zcat ruby.1.gz", "nroff -man", "less")
```

複数コマンドで支援していないこと

- 3番以降の fd
  標準入出力 (0, 1, 2) しか支援しない
  コマンド毎にオプションを指定することは可能
- 分岐したり合流したり回ったりするパイプ
  直線的なパイプラインしか支援しない
  一般的なグラフは扱わない
  - プロセス≒ノード
  - パイプ≒エッジ

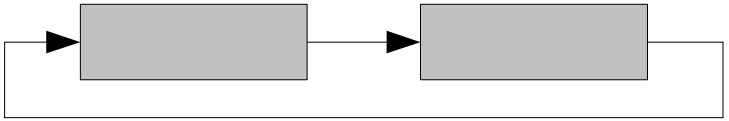
支援が不要な傍証

- 3番以降の fd
  Windows では子プロセスに継承不能
  [ruby-dev:35436] なかむら(う)
- パイプのサイクル
  バッファリングしすぎるとデッドロックして致命的
  普通のコマンドはその点では信用できない
- ひとつのパイプの読み出し側に複数プロセス
  書き込んだものがどのプロセスで読まれるか不明

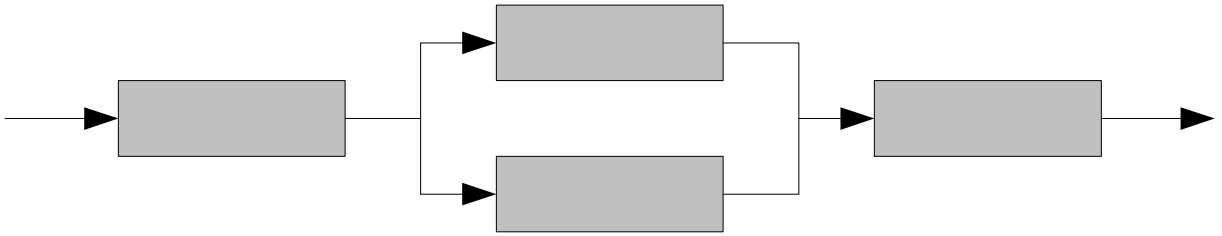
グラフの形

- 直線的: シェルで使用可能であからさまに有用

- サイクルを含むもの: 有用ではなさそう



- 上記以外 (直線的以外の DAG)
  用法が見つかったらメソッドの追加を考える



shell ライブラリ

shell ライブラリはシェルの記法を模した DSL

- シェル
  zcat < ruby.1.gz | nroff -man
- open3
  Open3.pipeline(["zcat", :in=>"ruby.1.gz"], "nroff -man")
- shell ライブラリ

```
Shell.def_system_command "zcat"
Shell.def_system_command "nroff"
sh = Shell.new
sh.transact {
  (zcat < "ruby.1.gz") | nroff("-man") > STDOUT
}
```

shell と open3 の比較

- カレントディレクトリが Shell オブジェクト単位
  open3/spawn で同様にするなら :chdir オプションをすべての呼び出しにつける
- パイプやリダイレクトがシェルの記法を模している
  open3 はシェルの記法は気にしていない
- コマンド間のパイプは ruby 経由
  Open3.pipeline のパイプはコマンド間を直結
- less は動かない (標準出力がパイプになるため)
  Open3.pipeline の最終段なら動く
- シェルの構文を模すためにちょっと無理ぎみ

まとめ

- spawn はポータブルな低位 API
  - いままでは良いプリミティブがなかった
  - fork よりもポータブルで高位
- open3 はコマンドを起動する高位 API
  - 欠点を直してかなり拡張した
  - shell ライブラリといくらか重なるがそれよりは低位
- 低位API と高位 API に分けたデザイン
- パイプの用法の考察