Ruby とプロセス
spawn について

産業技術総合研究所
情報技術研究部門
田中哲

2009-04-15　SEA & FSIJ 合同フォーラム

発表の内容

- 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
- open3

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

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

プロセス起動法の起源

- `c`　　perl, shell
- system(c)　　perl, C
- exec(c)　　perl, C
- IO.popen(c)　　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
- 標準入出力を繋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
- exec(c)　　使っていい状況は限定的
- IO.popen(c)　　標準エラー出力を扱えない
- 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

- プロセスを起動する関数

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

- コマンドラインを与える
- プロセスの終了は待たない
- pid を返す

シェルを経由しないコマンド起動

spawn("make", "all")

- 複数の文字列を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

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

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

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 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 => :close
- 子プロセスの fd => [:child, 子プロセスの fd]
- :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

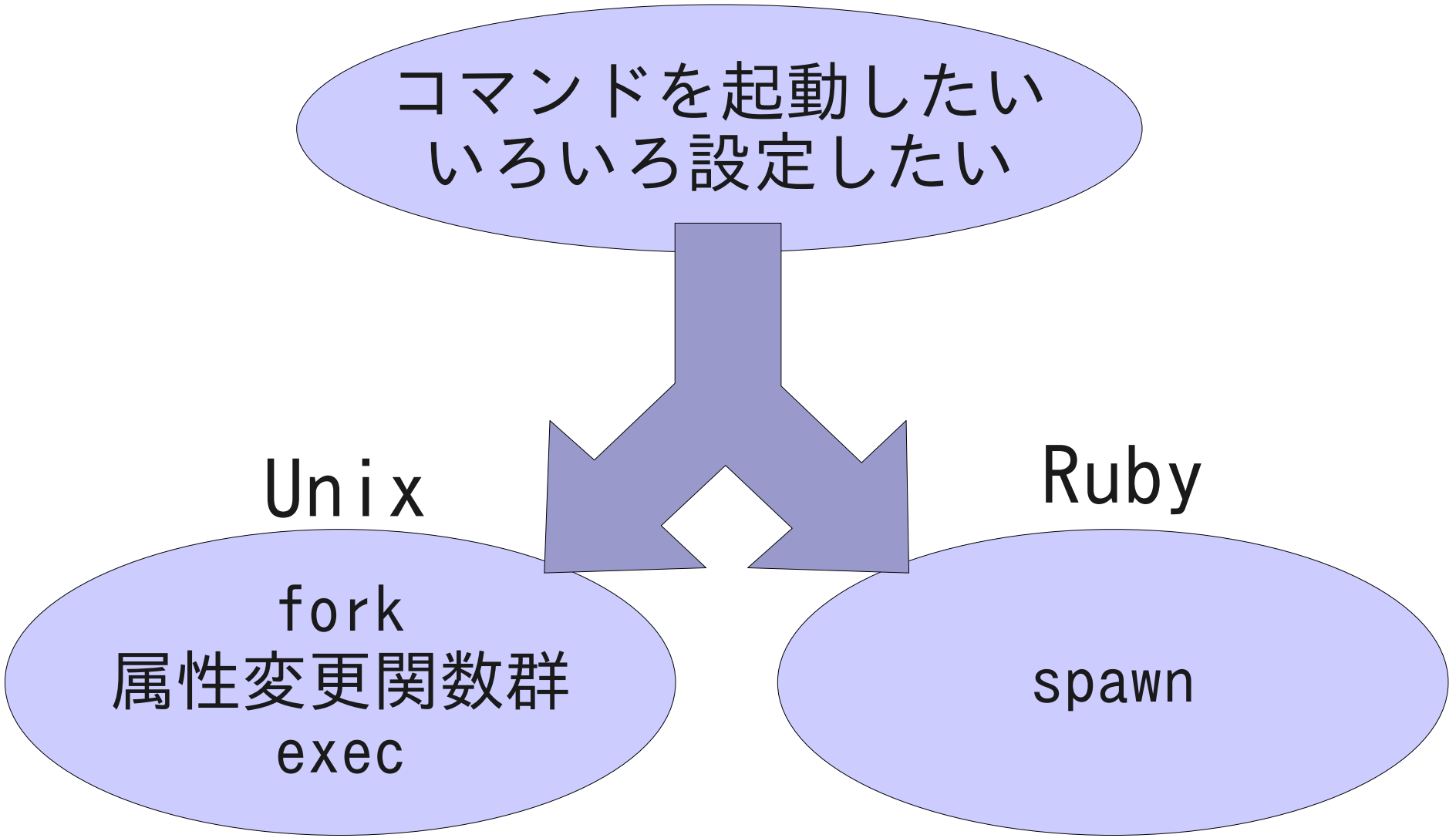
fd の指定

- 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

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・他のスレッド

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

- よくある疑問
- 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 => "log", :err => [:child, :out]
- 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)
- 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

open3

- spawn はプリミティブ
- 高レベルな支援を提供
  - パイプラインの構成
  - 出力のキャプチャ
  - 自動wait
- 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

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
- 注意: この発表で触れている機能は大部分が Ruby 1.9.1 で実装されているが、一部 Ruby 1.9.2 の部分がある