Unix修正主義
Ruby API is Improved Unix API

田中哲
Tanaka Akira

産業技術総合研究所
National Institute of Advanced Industrial Science and Technology (AIST)

RubyKaigi2010

今日の話

- これまでの話
  - Ruby がどうデザインされているか
  - Unix の機能をどういうデザインで提供するか
- これからの話
  - いくつかの問題と改善案

これまでの話

IO のメソッド (Ruby 1.4.6)

<< binmode close close_read close_write closed? each each_byte each_line eof eof? fcntl fileno flush getc gets ioctl isatty lineno lineno= pos pos= print printf putc puts read readchar readline readlines reopen rewind seek stat sync sync= sysread syswrite tell to_i to_io tty? ungetc write

C (stdio) 由来

<< binmode close close_read close_write closed? each each_byte each_line eof eof? fcntl fileno flush getc gets ioctl isatty lineno lineno= pos pos= print printf putc puts read readchar readline readlines reopen rewind seek stat sync sync= sysread syswrite tell to_i to_io tty? ungetc write

Unix 由来

<< binmode close close_read close_write closed? each each_byte each_line eof eof? fcntl fileno flush getc gets ioctl isatty lineno lineno= pos pos= print printf putc puts read readchar readline readlines reopen rewind seek stat sync sync= sysread syswrite tell to_i to_io tty? ungetc write

Perl 由来

<< binmode close close_read close_write closed? each each_byte each_line eof eof? fcntl fileno flush getc gets ioctl isatty lineno lineno= pos pos= print printf putc puts read readchar readline readlines reopen rewind seek stat sync sync= sysread syswrite tell to_i to_io tty? ungetc write

C++ 由来

<< binmode close close_read close_write closed? each each_byte each_line eof eof? fcntl fileno flush getc gets ioctl isatty lineno lineno= pos pos= print printf putc puts read readchar readline readlines reopen rewind seek stat sync sync= sysread syswrite tell to_i to_io tty? ungetc write

Windows 由来

<< binmode close close_read close_write closed? each each_byte each_line eof eof? fcntl fileno flush getc gets ioctl isatty lineno lineno= pos pos= print printf putc puts read readchar readline readlines reopen rewind seek stat sync sync= sysread syswrite tell to_i to_io tty? ungetc write

Ruby 独自

<< binmode close close_read close_write closed? each each_byte each_line eof eof? fcntl fileno flush getc gets ioctl isatty lineno lineno= pos pos= print printf putc puts read readchar readline readlines reopen rewind seek stat sync sync= sysread syswrite tell to_i to_io tty? ungetc write

stdio → IO

- int fprintf(FILE *, const char *, ...);
  →
  IO#printf(format, ...)

- size_t fread(void *, size_t, size_t, FILE *);
  →
  IO#read([length])

- 一般形:
  FILE * を引数にもつ関数
  →
  IO のメソッド

FILE* を引数に持つ stdio の関数

clearerr

fclose → close

feof → eof, eof?

ferror

fflush → flush

fgetc → getc

fgetpos → pos

fgets → gets

fprintf → printf

fputc → putc

fputs → puts

fread → read

freopen → reopen

fscanf

fseek → seek

fsetpos → pos=

ftell → tell

fwrite → write

getc → getc

putc → putc

rewind → rewind

setbuf

setvbuf

ungetc → ungetc

vfprintf → printf

IO ≒ オブジェクト指向 stdio

- FILE * を引数にもつ関数に対応する IO のメソッド
- 関数名の先頭の f を取り除く
  - fprintf → printf,
  - fread → read, ...
- メソッド名を Ruby 的に調整
  eof?, pos=, ...
- 都合が悪い関数は無視
  - エラーまわりは例外があって事情が違う
  - setbuf, setvbuf は内部的
  - ...

Ruby IO まとめ

- 半分以上のメソッドは独自でない
- Unix とその周辺に由来するものが多い
- Ruby は Unix 文化圏に属する
- Unix の I/O 機構をオブジェクト指向化したもの
- Unix使いにとって新しく覚えることが少ない
- Unix使いにとって使いやすい
- 多くの人が知っている知識を尊重して使いやすさを実現する

実践するのも難しくないでしょ?

Ruby と feof()

- feof() → IO#eof?
- 実は動作が違う

C言語 FAQ 12.2

Q: なぜ以下のコードは最後の行を2回コピーするのか。

```
while(!feof(infp)) {
        fgets(buf, MAXLINE, infp);
        fputs(buf, outfp);
}
```

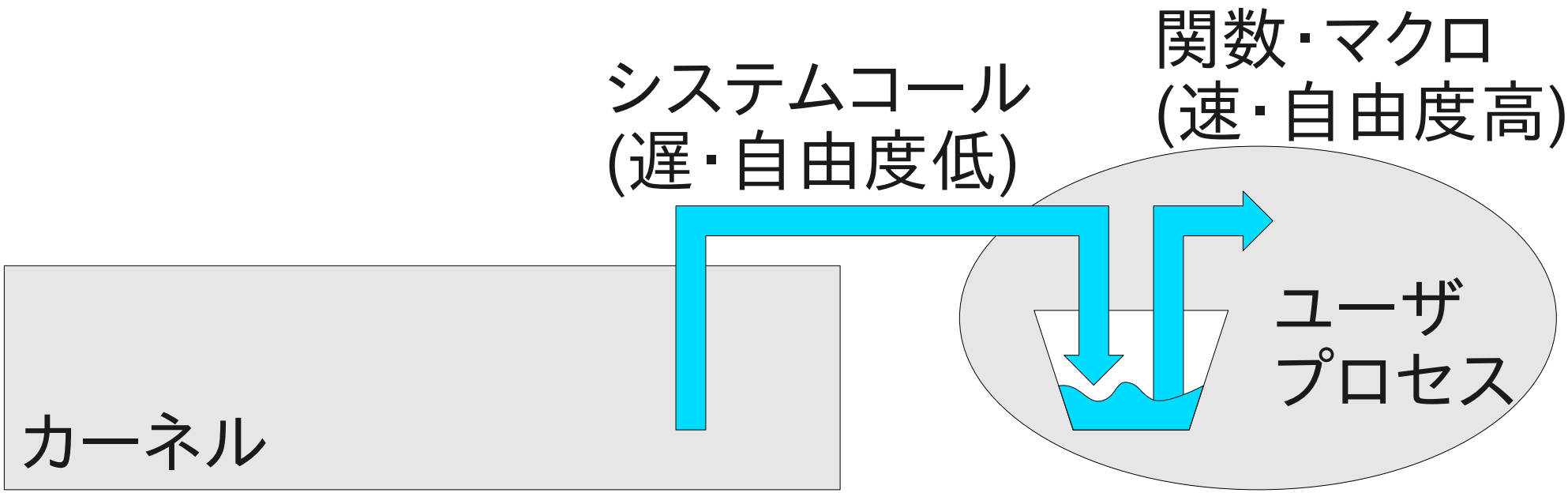
A: Cでは、EOFは入力ルーチンが読もうとしてファイルの終わり (End-Of-File) にたどり着いた後であることを示しているだけである (言い換えればC言語のI/OはPascalのI/Oとは異なる)。たいていは入力関数(この場合はfgets)の戻り値をチェックすればよい。feof()を使う必要がまったくない場合が多い。

EOFを判断する関数

- 考えられる動作が 2種類ある
  - Pascal の動作　　これから読むと EOF になるか?
  - C の動作　　すでに EOF に出会ったか?
- Pascal の動作を期待する人は多い
  FAQ になるほどに
- C の feof() の動作は異なるので間違う
- Ruby の IO#eof? は Pascal の動作
  実際にバッファにちょっと読んでみて確かめる
- stdio に比べて動作がわかりやすい

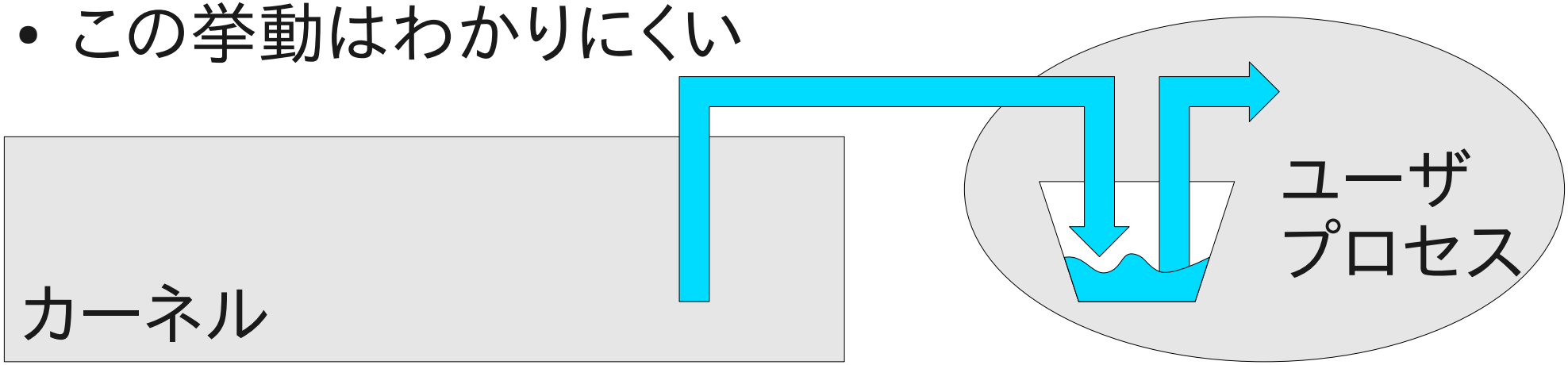
stdio とバッファリング

- stdio はデータをプロセス内でバッファリングする
- システムコールは遅いので一度のシステムコールでたくさん読み書きして回数を減らす
- 行読み込みなどで、読みすぎた部分をとっておく



IO#eof? とバッファリング

- IO#eof? は終端かどうか確かめるために内部的に入力を行うことがある
- もともとバッファが空でなければ終端でない
- バッファが空なら実際に読んでみる
- データが読めたら終端でない
- 読んだデータはバッファに取っておく
- この挙動はわかりにくい



IO#eof?

- IO#eof? は Pascal の動作
- stdio の feof() に比べるとわかりやすい
- とはいえ裏で読み込みを行うのは分かりにくい
  - 例: 端末に適用するとブロックするかもしれない
- でも実はいらない子

Python と feof()

一方、Pythonは
EOF判定の
メソッドを提供し
なかった

feof() まとめ

- Ruby の IO#eof? は C の feof() とは異なり、プログラマの期待にそった動作をするので使いやすい
- C の仕様に従うのではなく、ユーザの期待に応えているのが重要
- 必ずしも C の仕様のすべてがユーザの期待にそっているわけではない
- ただし内部的な動作はわかりにくい
- Python の見識は素晴らしい

EOF フラグ

- feof() は FILE 構造体の中の EOF フラグを読み出す関数
- Ruby 1.8 では feof() が真の場合 io.read(n) は読み込みに挑戦せずに nil を返す
- tail -f もどきを実装するのに困る
  EOF に出会った後にファイルが伸びたぶんを読めない
- Ruby 1.9 で stdio を捨てるときに EOF フラグを実装せずに済ました
- tail -f が問題なく実装できて幸せになった

本当にいらない子?

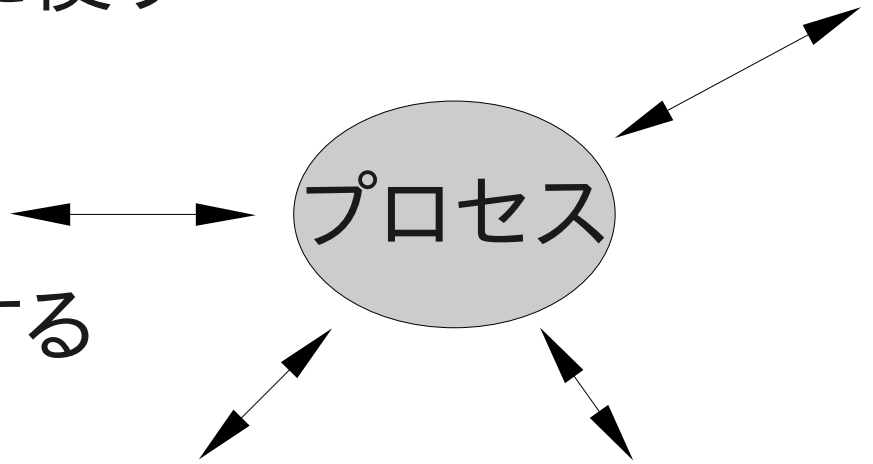
- 以下に納得できるか?
- % ./ruby -ryaml -e 'p YAML.load(STDIN)'
  "a"　　# 入力
  ^D^D^D^D^D^D^D　　# EOF を 7回
  "a"　　# 結果
- YAML パーザの終了に EOF が 7回必要
- パーザで先読みを参照するたびに読んでいる?
- 端末に限定した EOF フラグがいったん入ったが文句が出て revert された
- 挙動を期待に近づける余地がまだあるかも?

Ruby と select システムコール

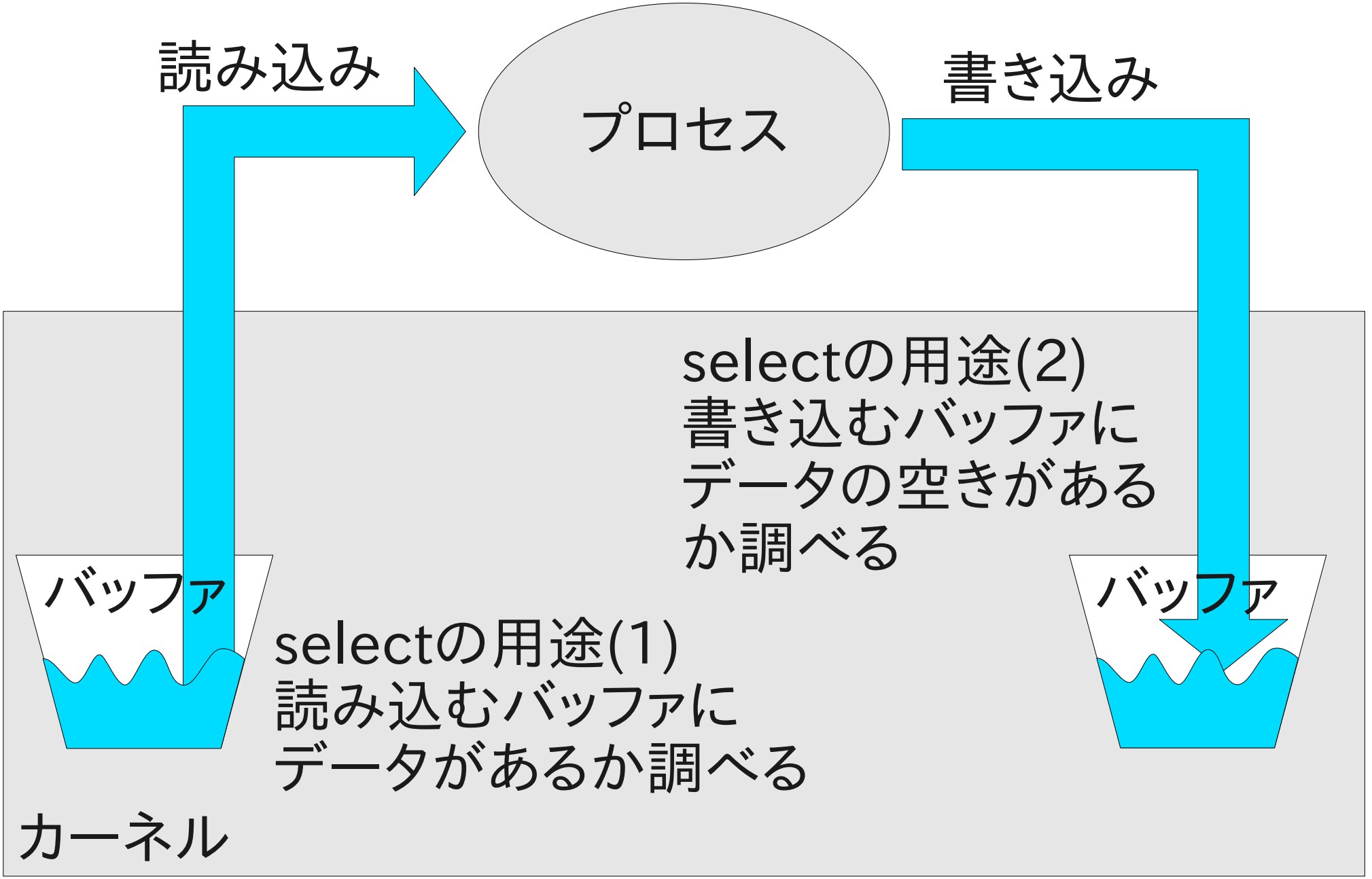
- select → IO.select
- 実は動作が違う

select

- select は Unix のシステムコール
- 即座に I/O 可能かどうか検査する
  - 読み込み可能か?
  - 書き込み可能か?
- あるいは I/O 可能になるまで待つ
- 複数の相手と通信する時に使う

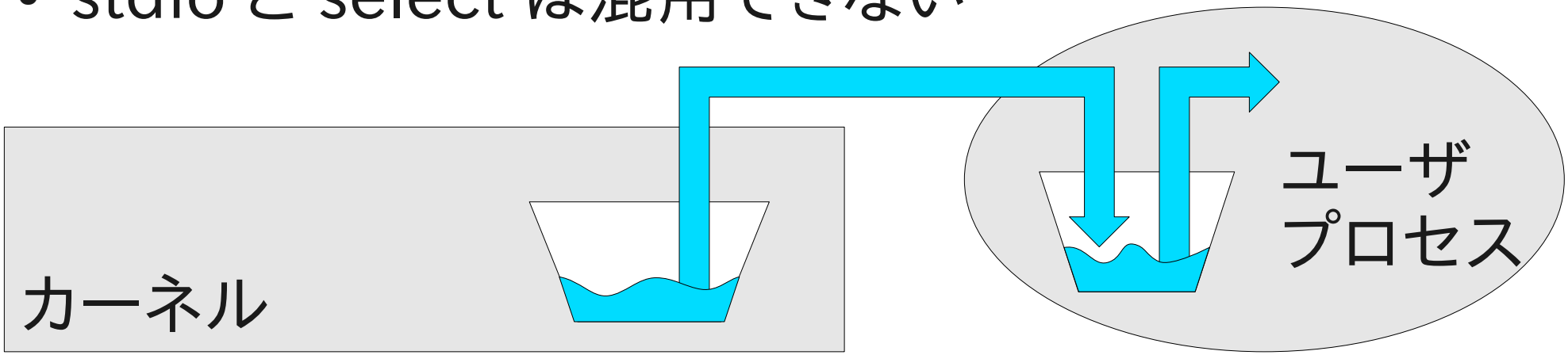


select の動作


読み込み
プロセス
書き込み
selectの用途(2)
書き込むバッファに
データの空きがある
か調べる
バッファ
バッファ
selectの用途(1)
読み込むバッファに
データがあるか調べる
カーネル

stdio と select

- stdio はデータをプロセス内でバッファリングする
- selectはシステムコールなので stdio のバッファを考慮しない (できない)
- stdio のバッファにデータがあっても select はデータがないと考えるかもしれない
- データがあるのにないと思って待つのは間違い
- stdio と select は混用できない



Ruby 1.8 の IO.select

- select システムコール → IO.select
- システムコールは stdio のバッファを考慮しない
- IO.select は考慮する
  - 以下のいずれかの場合に読み込み可能と判断する
  - stdio のバッファが空でない
  - select システムコールが読み込み可能と判断する
- FILE 構造体の中身を直接アクセスする必要がある
  これはポータブルでない
- IO.select とバッファを使うメソッドは混用できる
  あたりまえのように思えるが、あたりまえではない

Ruby 1.9 と stdio

- Ruby 1.9 の IO は stdio を使わない
- stdio の代わりにバッファリングを独自に実装
- select システムコールはそのバッファを考慮しない (できるわけがない)
- stdio のバッファと異なり、バッファが空かどうか判断するのが簡単・ポータブルになった

select まとめ

- IO.select は一見 select システムコールを呼ぶだけと見えて、実は違う
- プロセス内の IO バッファを考慮する
- バッファリングする IO のメソッドと混用できて使いやすい
  - 次の getc がブロックするかどうか調べるのに使える

Ruby と read システムコール

- ある時点で読めるだけ読みたいことがある
  - 到着しているデータの量はわからない
  - 到着しているデータの終端の印もわからない
- 例えばストリームの中継で必要になる
  - 中身に関係なく、到着したデータを転送する
- stdio では難しい
  getc で可能だが、Ruby で getc を使うと遅すぎる
  (1byte 毎のメソッド呼び出しは耐えられない)
- readシステムコールの動作が都合がいい
  与えたバッファ長を上限として読めるだけ読む

IO#sysread による中継

- Ruby では IO#sysread により read システムコールを呼び出せる
- sysread は Perl由来
- def relay(i, o)
  begin
    loop {
      o.write i.sysread(4096)
    }
  rescue EOFError
  end
end
relay(STDIN, STDOUT)

だいたい動く

relay の問題

- ノンブロッキングモードを考慮していない
- バッファリングするメソッドとの混用

ノンブロッキングモード

- Unix は fd をノンブロッキングモードに設定できる
- ノンブロッキングモードでは、データが到着していないときに読み込むと、ブロックするかわりにエラーになる
- relay(i,o) の i, o がノンブロッキングモードだとうまく動かないかも
  (ノンブロッキングモードのエラーに対応してない)
- stdio はノンブロッキングモードをサポートしてない
  これらを組み合わせは「破滅の処方」 by Stevens (UNIX Network Programming Vol.1)

io.sysread(maxlen) の詳細 (1.8)

- データがすでに到着しているときの動作
  - maxlen バイトを上限として読み込む
- データがまったく到着していないときの動作
  - io がブロッキングモードあるいは他にスレッドがある
    - データが到着するのを待って読む込む
  - io がノンブロッキングモードかつ他にスレッドがない
    - Errno::EAGAIN もしくは Errno::EWOULDBLOCK 例外

データが未到着	マルチスレッド	シングルスレッド
ブロッキングモード	ブロック	ブロック
ノンブロッキングモード	ブロック	例外

?

io.sysread(maxlen) の中身 (1.8)

1. 他にスレッドがあったら (マルチスレッドなら)
   a) select で各スレッドにデータが到着するのを待つ
   b) データの到着したスレッドのひとつを選んで実行再開
2. read システムコールを呼び出す
   データが到着している場合 (EOF の場合も含む)
   a) 読み込む
   データが到着していない場合
   a) ブロッキングモードならデータの到着を待って読み込む
   b) ノンブロッキングモードなら EAGAIN などになって例外
3. 読み込んだデータを返す

io.sysread(maxlen) の中身 (1.8)

1. 他にスレッドがあったら (マルチスレッドなら)
   a) select で各スレッドにデータが到着するのを待つ
   b) データの到着したスレッドのひとつを選んで実行再開
2. read システムコールを呼び出す
   データが到着している場合 (EOF の場合も含む)
   a) 読み込む
   データが到着していない場合
   a) ブロッキングモードならデータの到着を待って読み込む
   b) ノンブロッキングモードなら EAGAIN などになって例外
3. 読み込んだデータを返す

io.sysread(maxlen) がブロック

データが未到着な場合

	マルチスレッド	シングルスレッド
ブロッキングモード	select がブロック	read がブロック
ノンブロッキングモード		例外

sysread とノンブロッキングモード

- ノンブロッキングモードなら EAGAIN/EWOULDBLOCK 例外が発生するかもしれない
- ノンブロッキングモードにしてもマルチスレッドならブロックするかもしれない

注意しないと失敗する

注意しても失敗する
どのライブラリがバックグラウンドスレッドを使うか知っていますか？

IO#sysread

- IO#sysread はノンブロッキングモードでもブロックすることがある
  - Rubyでのノンブロッキングモードは信用できない
  - Ruby は Unix のノンブロッキングモードを改悪
- 中継の場合は、むしろブロックして欲しい
  データが到着しない限り、行う処理はない
  - でもノンブロッキングモードの例外が発生する可能性は残っている
- ブロックして欲しい場合も欲しくない場合もうれしくない

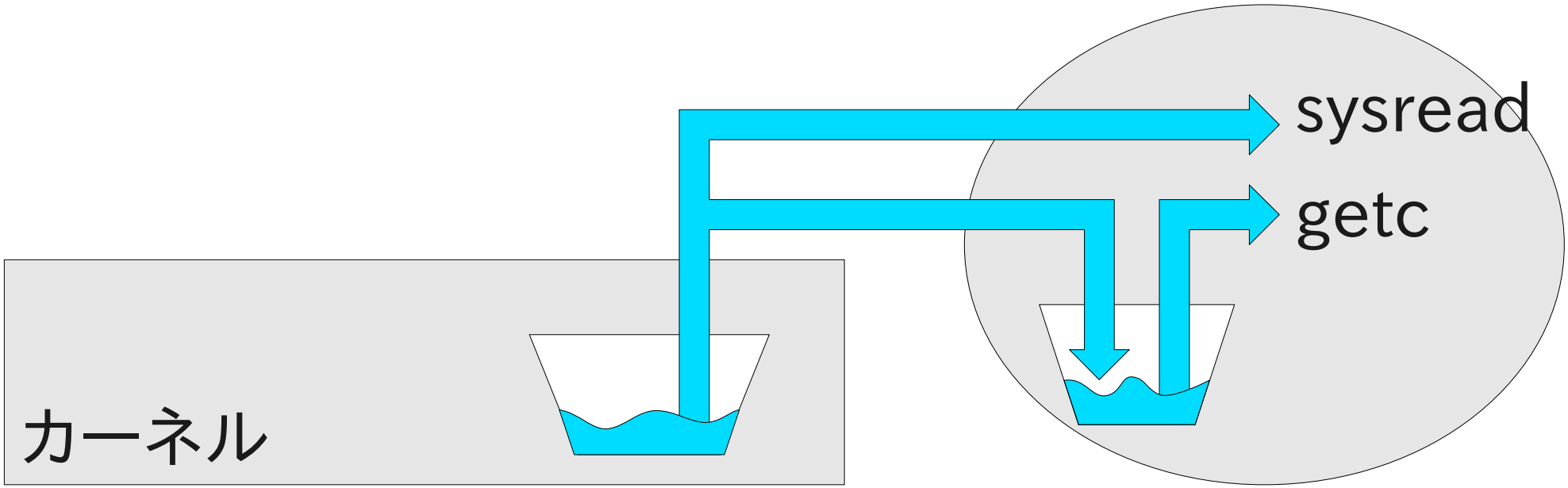
中継 (再)

- 中継ではデータが到着していなければ待ちたい
- しかし EAGAIN/EWOULDBLOCK になるかも
- def relay(i, o)
  begin
  loop {
  begin
  buf = i.sysread(4096)
  rescue Errno::EAGAIN, Errno::EWOULDBLOCK
  IO.select([i]); retry
  end
  o.write buf
  }
  rescue EOFError
  end
  end
  relay(STDIN, STDOUT)

例外になったら select で待ってやりなおす
ノンブロッキングモードでも動く
長くなって悲しい

sysread とバッファ

- sysread は read システムコールを呼び出す
- プロセス内の IO バッファは使わない
- バッファを使うメソッド (例: getc) と混用すると何が起きるか?



sysread と getc の混用

- 混用は禁止されていて例外になる

```
% ruby -e '
        p STDIN.getc
        p STDIN.sysread(10)'
abc       # 入力
"a"       # getc の結果
-e:1:in `sysread': sysread for buffered IO (IOError)
        from -e:1:in `<main>'
```

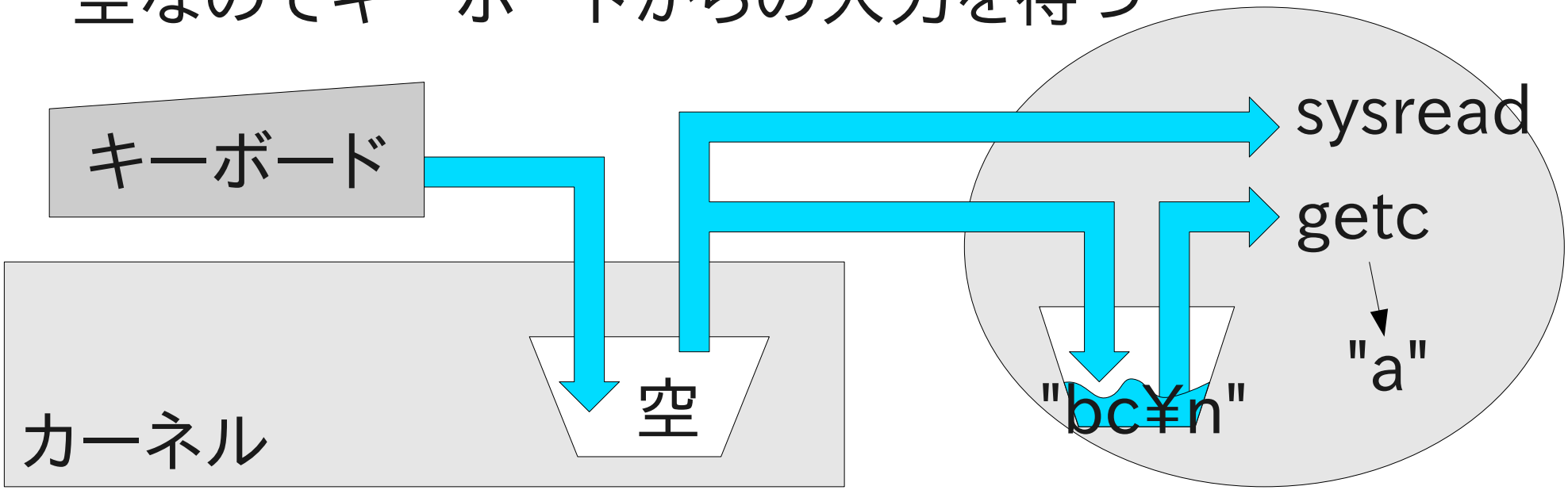
- sysread 例外条件: プロセス内バッファが空でない
- もし混用が許されていたら sysread はブロックする

Perl は混用できる

- % perl -e '
  print getc(STDIN), "¥n";
  sysread(STDIN, $buf, 10);
  print "$buf¥n";'
  abc # 入力
  a # getc の結果
  # 入力待ちになってブロック
- "bc" (と改行) はどこにいった?
- perldoc -f sysread には混用は混乱を招くかもしれないという注意書きがある

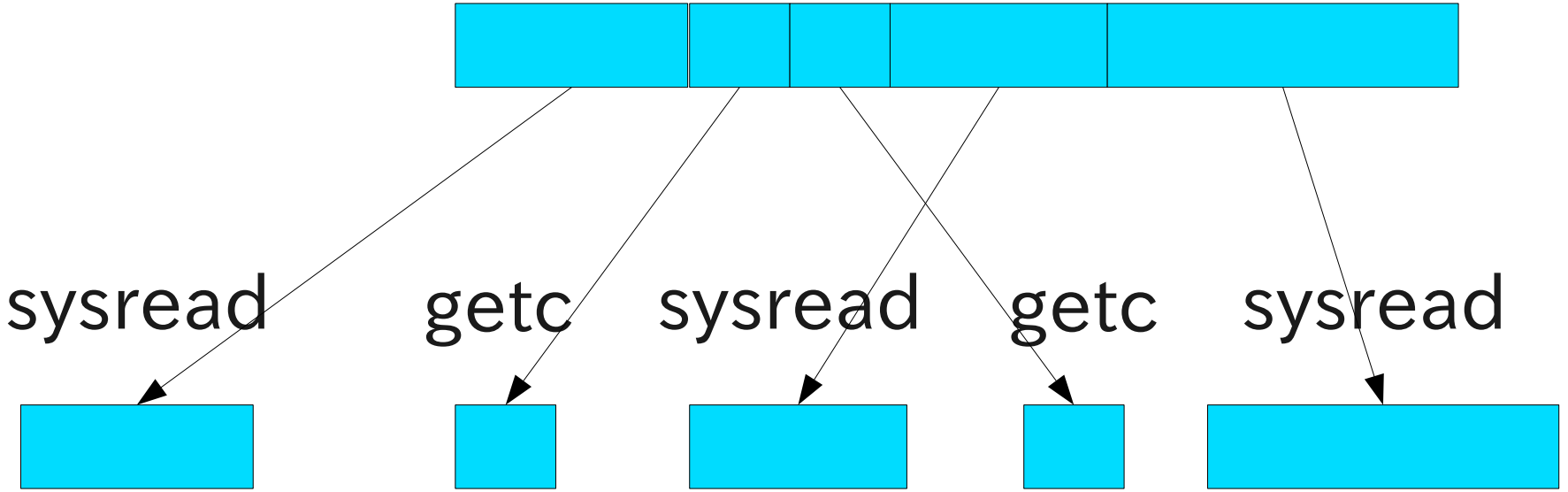
sysread のブロック

- キーボードからカーネルのバッファへ "abc¥n" が送られる
- getc は "abc¥n" をプロセス内に取り込み、"a" だけを返す (カーネル内のバッファは空になる)
- sysread はカーネルから読もうとするがバッファが空なのでキーボードからの入力を待つ



混用の混乱

- 存在するはずのデータが読み出せないことがある
  sysread はプロセス内バッファのデータを読まない
  カーネルにデータが無ければブロックする
- データの順序が変わることがある
  sysread はプロセス内バッファのデータを無視して
  その後のカーネル内のデータを読む



sysread の混用の扱い

- IO#sysread は read システムコールを呼ぶ
- IO#getc など、バッファを経由するメソッドとの混用は禁止されている
- なお IO#eof? もバッファを扱うので sysread と混用できない
  IO#eof? は一見副作用がなさそうなので間違いやすい
- これは混用による混乱を防いでいる
- Perl では混用 (と混乱) の自由がある

混用と中継

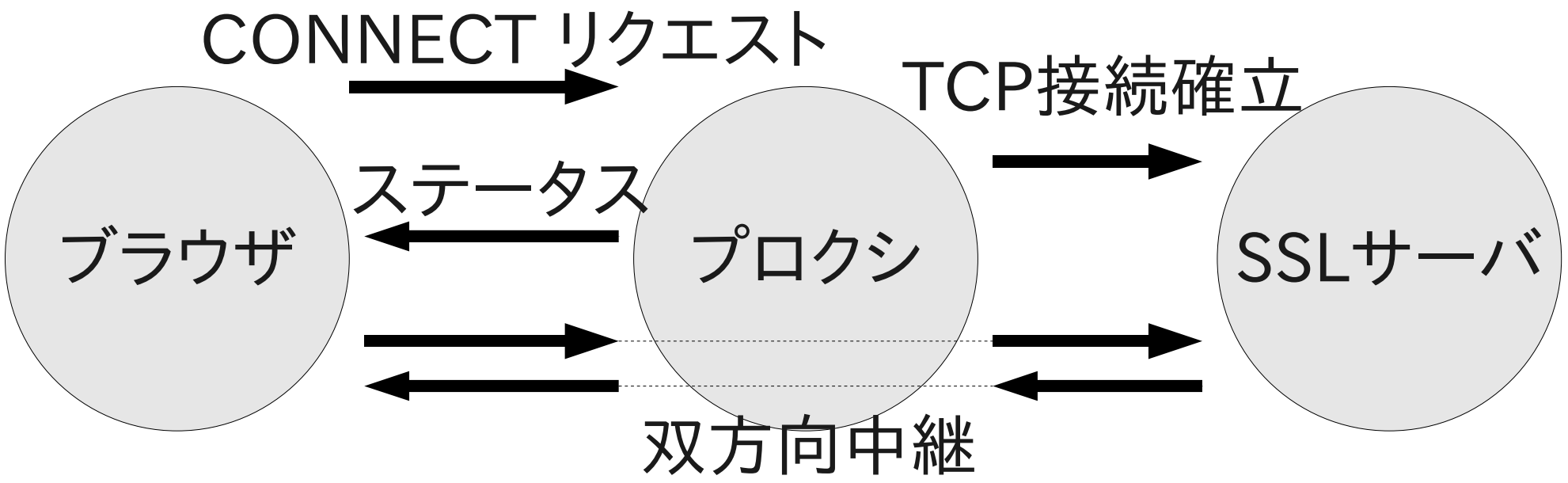
- バッファが空でない場合、動かないかもしれない

  STDIN.getc　　# バッファにデータが残るかも
  relay(STDIN, STDOUT) # sysread で例外発生?

混用したい具体例: HTTP CONNECT

- SSL PROXY などに使う
- プロトコルの内容:
  C: CONNECT ssl-server:443 HTTP/1.1
  C:
  S: HTTP/1.1 200 Connection established.
  S:
  C,S: (SSL通信)
- 最初の HTTP な部分は行指向
- SSL 通信はバイナリで双方向
  - どちらからデータがくるかわからない
  - 到着するデータの量もわからない

HTTP CONNECT によるプロクシ



プロクシの(手抜き)実装

- c = proxy_sock.accept
  connect_req = c.gets("¥r¥n¥r¥n")
  s = TCPSocket.open(...)
  c.print "HTTP/1.1 200 ...¥r¥n¥r¥n"
  Thread.new { relay c, s }
  Thread.new { relay s, c }
- バッファを使う gets の後に sysread を使う relay 混用禁止で動かないことがありえる
- gets のかわりに sysread を使わないといけない 必要以上にデータを読み込んじゃったらちゃんと残しておいて後で中継しないといけない

gets が使えない → readpartial

- gets 相当を sysread で実装しないといけない
- 理不尽
- なんですでにあるものを使えないのか
- 混用可能な (混乱しない) sysread が欲しい
- そこで IO#readpartial

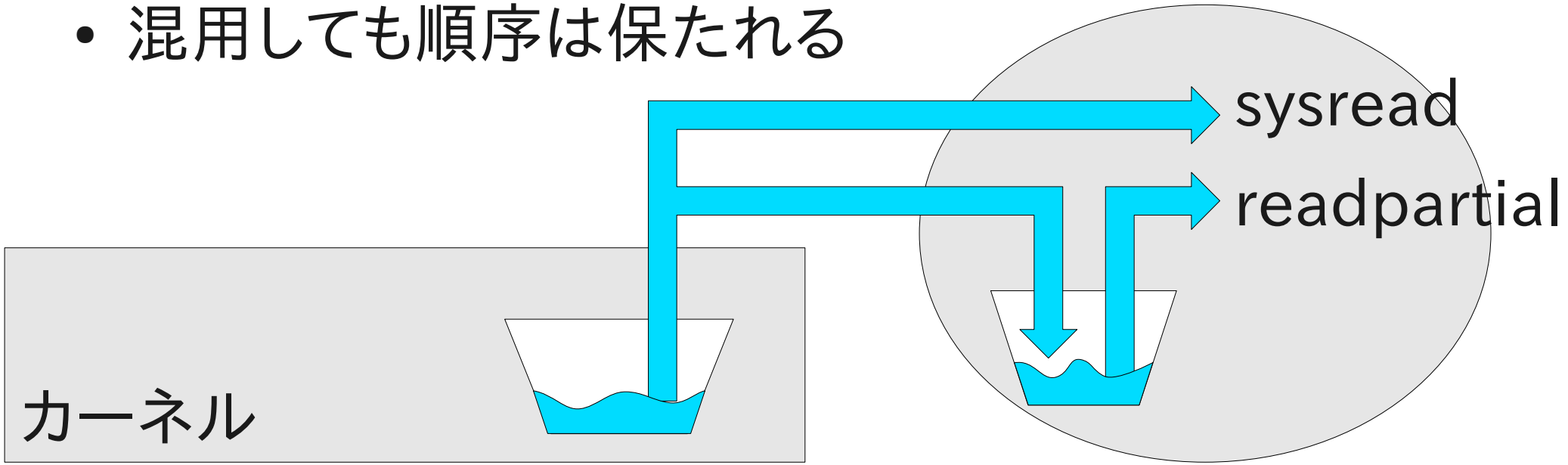
中継 (再々)

- sysread のかわりに readpartial を使う
- readpartial はノンブロッキングモードに影響されない (EAGAIN などの rescue が不要)
- def relay(i, o)
  begin
    loop {
      o.write i.readpartial(4096)
    }
  rescue EOFError
  end
end
relay(STDIN, STDOUT)

前述のプロクシも動く
ノンブロッキングモードでも動く

IO#readpartial(maxlen) の中身

- プロセス内バッファが空でなければそこから読む (read システムコールは呼ばない)
- プロセス内バッファが空だったときだけ read システムコールを呼び出す
- EAGAIN/EWOLDBLOCK なら待って再挑戦
- 混用しても順序は保たれる



readpartial の利点

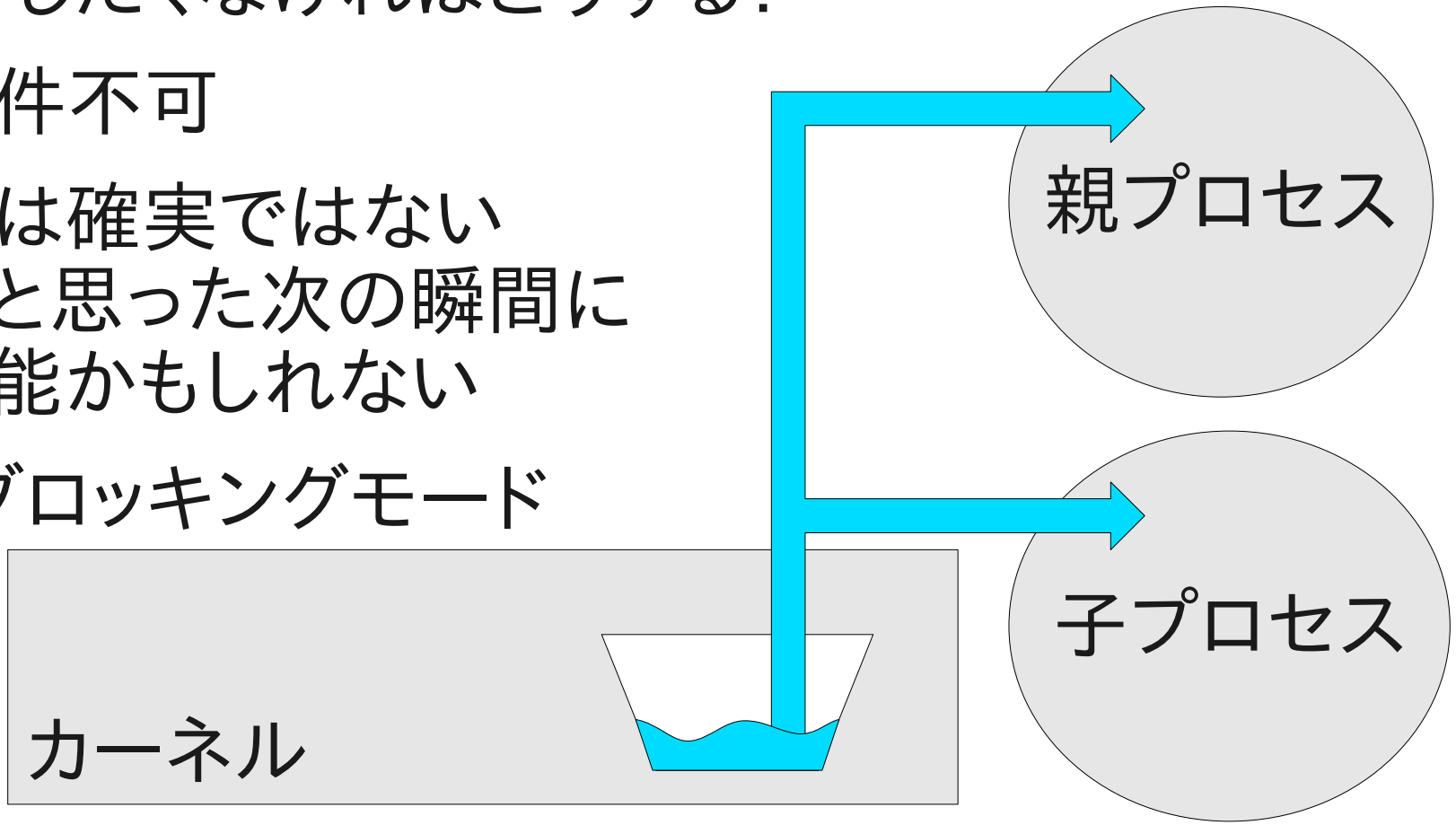
- IO のバッファを使う他のメソッドと混用できる
  混用を禁止するのでなく、混用してもちゃんと動く
  - プロセス内バッファから読む sysread
  - 用途が広く使いやすい
    HTTP CONNECT の中継にも使える
- ノンブロッキングモードに影響されない
  - EAGAIN/EWOULDBLOCK を rescue しなくてよい
  - 考えなければならないことが少ない

Unix のノンブロッキングモード

- 端末、パイプ、ソケットなどに対して、ブロックするかわりにエラーになったり中途半端に終わるモード
- ビジーループを避けるため select の併用が必須
- open 毎にモードがあり、fd を継承すればプロセス間で共有される
- ノンブロッキングモードにすると、その fd の読み込み・書き込み両方に影響する
- 用途
  - 複数プロセスでひとつのパイプ・ソケットから読み込むときにブロックしたくない
  - 書き込むときにブロックしたくない

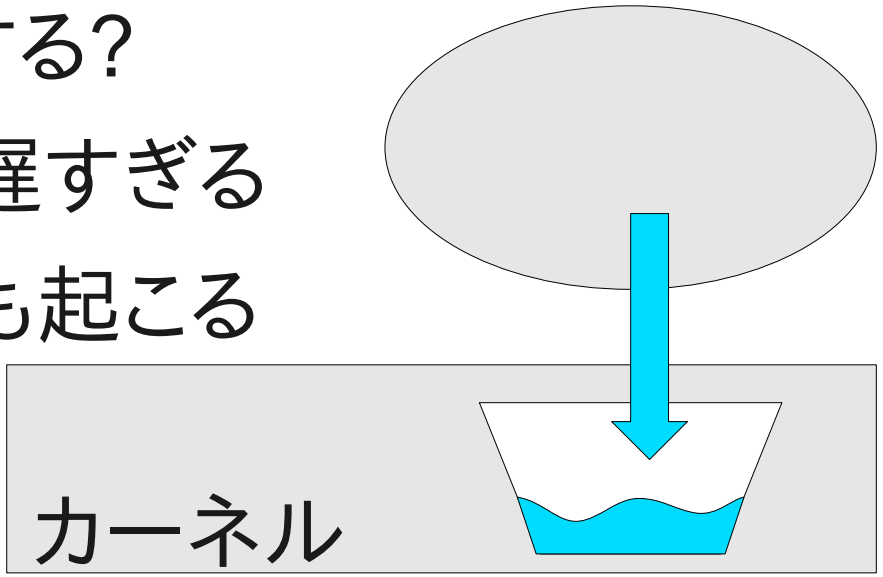
ノンブロッキングな読み込み

- 複数プロセスがカーネル内のひとつのバッファから読み込む
- ブロックしたくなければどうする?
- 競合条件不可
- select は確実ではない
  可能だと思った次の瞬間には不可能かもしれない
- 要ノンブロッキングモード



ノンブロッキングな書き込み

- ブロッキングモードの write システムコールは与えられたデータをすべて書き込むまでブロックする (read システムコールが 1byte でも読めたら返ってくるのとは異なる)
- select のバッファが空いているというのは 1byte 以上空いているという意味しかない
- ブロックしたくなければどうする?
- 1byte ずつ write するのは遅すぎる
- 複数プロセスなら競合条件も起こる
- 要ノンブロッキングモード



ノンブロッキングなメソッド

- ノンブロッキングモードの sysread は信用できない
  マルチスレッドだとブロックする
- syswrite も同様に信用できない
- readpartial はノンブロッキングモードでも確実にブロックする
- 逆に確実にブロックしないのも欲しい
- そこで read_nonblock, write_nonblock

io.read_nonblock(maxlen) の中身

1. io のプロセス内バッファにデータがあったらそのデータを maxlen を上限として返す
2. io をノンブロッキングモードに設定する
3.(残念なことに、ここに競合条件がある)
4. read システムコールを呼び出す
   - データが到着していない場合EAGAIN などの例外発生
   - データが到着している場合、読み込む
5. 読み込んだデータを返す

select しない!

io.write_nonblock(maxlen) の中身

1. io のプロセス内バッファにデータがあったらそのデータを書き出す (ここでブロックする可能性有り)
2. io をノンブロッキングモードに設定する
3. (残念なことに、ここに競合条件がある)
4. write システムコールを呼び出す
   - バッファに空きがなければ EAGAIN などの例外発生
   - 空きが充分でなければ、いっぱいになるまで書き込んで書き込んだ量を返す
   - 空きが充分なら、すべて書き込んで書き込んだ量を返す
5. 書き込んだ量を返す

select しない!

select しない

- {read,write}_nonblock は select しない
- read,writeシステムコール直前にノンブロッキングモードに設定するのでほぼブロックしない
  そのため select でブロックするか調べない
- fd が共有されていて、他のプロセスがタイミング良くブロッキングモードに変えたらブロックするかもしれない (競合条件)
- {read,write}_nonblock はブロッキングモードには戻さないので、複数プロセスでも一貫してこれらを使っている限り上記の競合条件は起きない

ノンブロッキングモード非依存

- 基本的にIOのメソッドはノンブロッキングモードかどうかで動作が変化しない
  (例外: sysread, syswrite, 1.8のread)
- ブロックするメソッドはノンブロッキングモードであっても (select と再挑戦を行って) ブロックする
- ブロックしないメソッドはブロッキングモードであっても (ノンブロッキングモードに設定して) ブロックしない
- モードを気にしなくてよいので使いやすい
- read と write で (メソッド呼び出し毎に) 独立にノンブロッキングにするかどうか選べる

readシステムコール まとめ

- read システムコールはノンブロッキングモードかどうかで動作が変わって良くない
- プロセス内バッファとの組み合わせで問題発生
- Rubyのマルチスレッド実装が問題をさらに拡大
- 用法を整理して別メソッドを用意
  - データがなければブロックしたい: readpartial
  - データがなくてもブロックしたくない: read_nonblock
- どちらもプロセス内バッファ、マルチスレッドと協調する

writeシステムコール まとめ

- write システムコールはノンブロッキングモードかどうかで動作が変わって良くない
- プロセス内バッファとの組み合わせで問題発生
- Rubyのマルチスレッド実装が問題をさらに拡大
- 別メソッドを用意: write_nonblock
- プロセス内バッファ、マルチスレッドと協調する

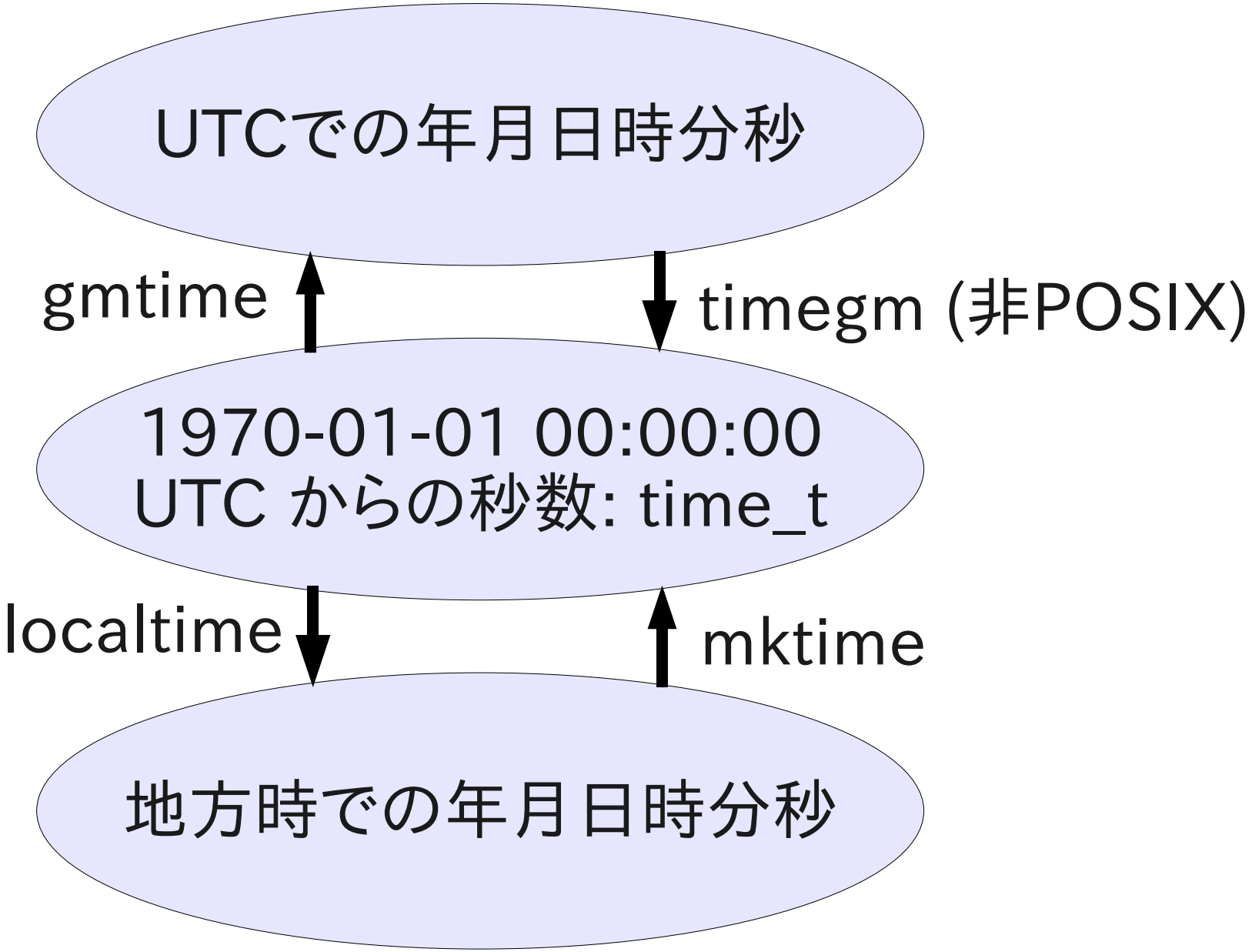
Ruby と fork システムコール

- Unix には昔からfork システムコールがある
- 後に Unix はマルチスレッドになった
- fork とマルチスレッドは相容れない
- Ruby 1.9 は常にマルチスレッド (タイマースレッド)
- Windows には fork がない
- spawn メソッドの導入
- 詳細は以下を参照
  open3のはなし、東京Ruby会議03、2010-02-28

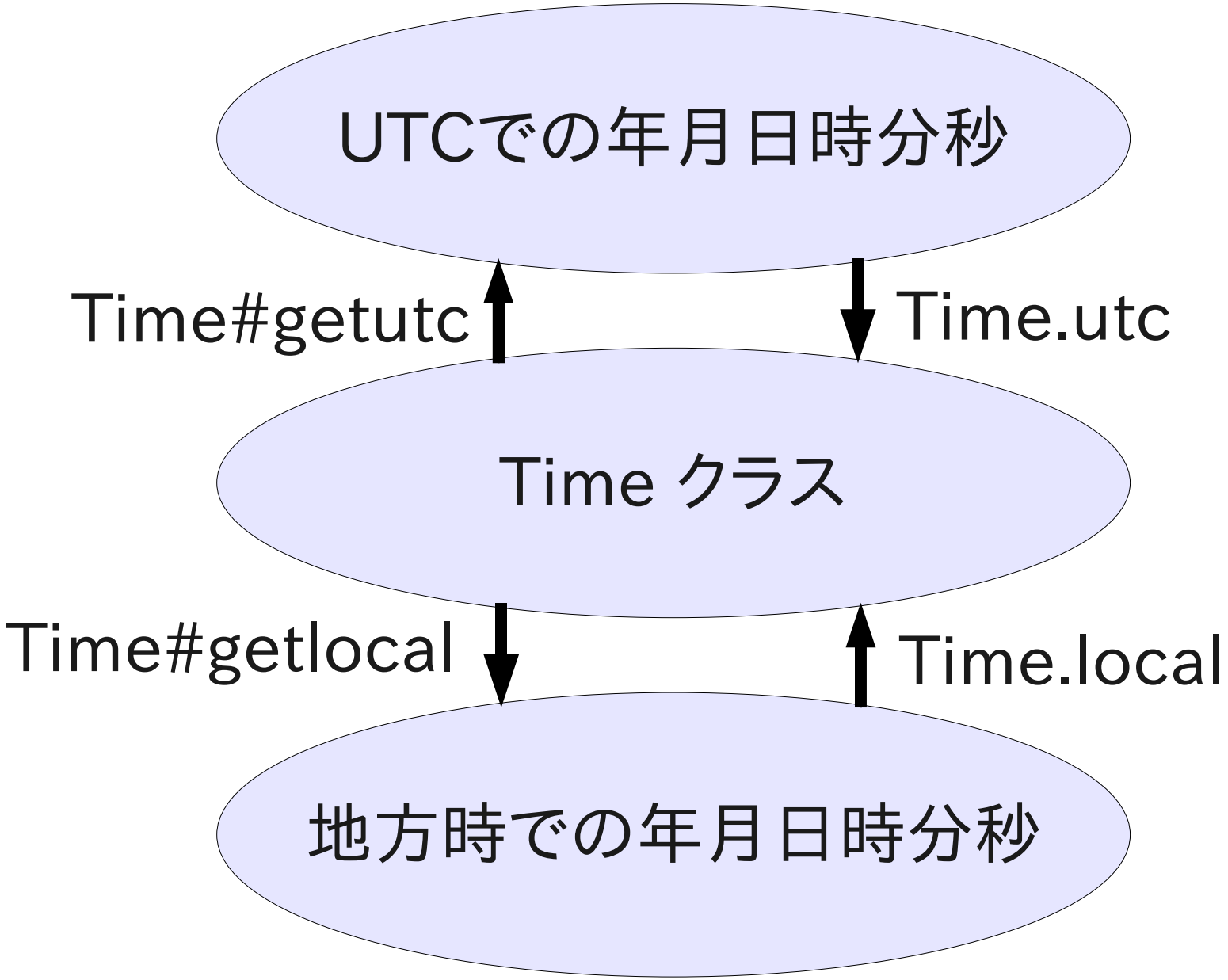
Ruby と2038年問題

- 1970-01-01 の前後 2**31秒しか表現できない
- 夏時間が問題を厄介にする
- 閏秒がさらに問題を厄介にする
- どうにか解決
- 詳細は以下を参照
  Ruby における 2038年問題の解決、札幌Ruby会議02、2009-12-05

Unix の時刻関数


UTCでの年月日時分秒
gmtime
timegm (非POSIX)
1970-01-01 00:00:00
UTC からの秒数: time_t
localtime
mktime
地方時での年月日時分秒

Ruby の時刻


UTCでの年月日時分秒
Time#getutc
Time.utc
Time クラス
Time#getlocal
Time.local
地方時での年月日時分秒

UTCからの変換

- timegm は標準でないのであるとは限らない
- でも Time.utc は常にある
- Ruby では Unix に足りない部分を補完している
- ていうかなんで Unix で標準になってないの?
  (閏秒がなければポータブルに計算できるから?)

年月日時分秒

- Unix では struct tm 構造体で表現される
- 以下のフィールドを持つ
  - tm_sec　　秒 [0,60]
  - tm_min　　分 [0,59]
  - tm_hour　　時 [0,23]
  - tm_mday　　日 [1,31]
  - tm_mon　　月 [0,11]
  - tm_year　　年 (1900年が起点)
  - tm_wday　　曜日 [0,6] (日曜日=0)
  - tm_yday　　年初からの日数 [0,365]
  - tm_isdst　　夏時間フラグ

なんで[1,12]じゃないの？

西暦のままでいいのに

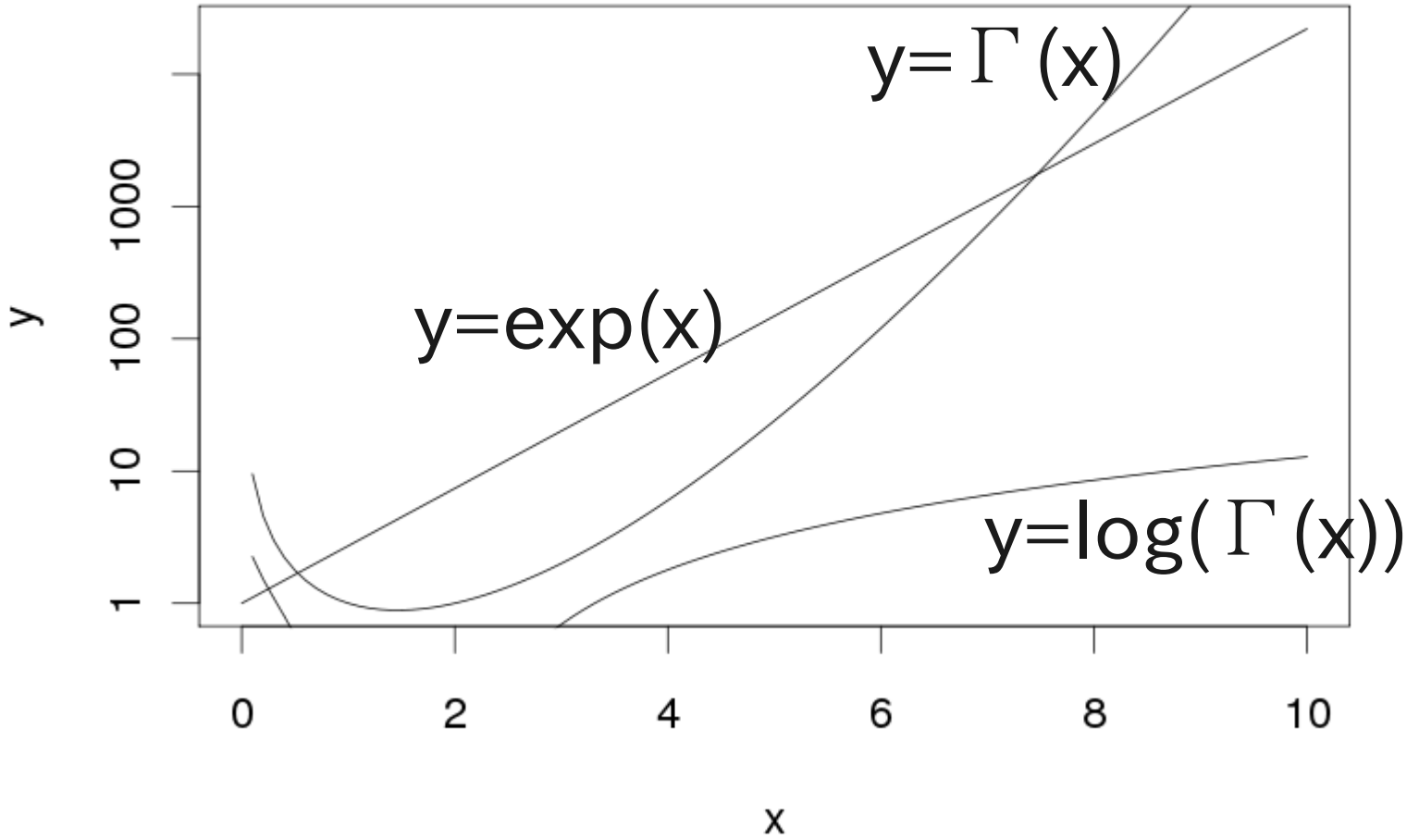
1で始めるのが普通らしい

Ruby での月と年

- Time#mon は 1 から 12 を返す
- Time#year は西暦の年をそのまま返す
- Time#yday は 1月1日が1
- 人間がふだん使っている表現そのままでギャップがなく、使いやすい

ガンマ関数 $\Gamma(x)$

- 指数関数よりさらに大きくなる
- $\Gamma(172)$ で IEEE 754 倍精度がオーバーフロー
- 大きすぎるので普通は対数をとって使う: $\log(\Gamma(x))$



gamma の歴史

- 4.2BSD は gamma() は log(Γ(x))
- 4.3BSD は加えて lgamma() も log(Γ(x))
- 4.4BSD は gamma() を Γ(x) に変更
- NetBSD, FreeBSD, OpenBSD は gamma を log(Γ(x)) に戻す
- POSIX は lgamma() を log(Γ(x)) として定義
  tgamma() を Γ(x) として定義
  tgamma は true gamma の略
  gamma() は定義してない

「論語」 孔子

- 「過ちては改むるに憚ること勿れ」
- 過ちを犯したら、ためらわないで改めよ。

- 残念ながら互換性に勝てなかった
  互換性>>>>>>>>>>>>>孔子

Ruby とガンマ関数

- Math.lgamma() が log(Γ(x))
- Math.gamma() が Γ(x)
- もともと Math.gamma はなかった
- t (true) と称する必要はない

- 言語を変えるときは互換性を気にせず間違いを正す好機
- 他の言語の間違いを繰り返す必要はない
- 盲目的に他の言語の仕様に従うのはやめよう

これまでの話のまとめ

- Unix にもいろいろ失敗がある
- Ruby ではいろいろ直している
- おそらく、直した部分は Unix使いにも受け入れられている

これからの話

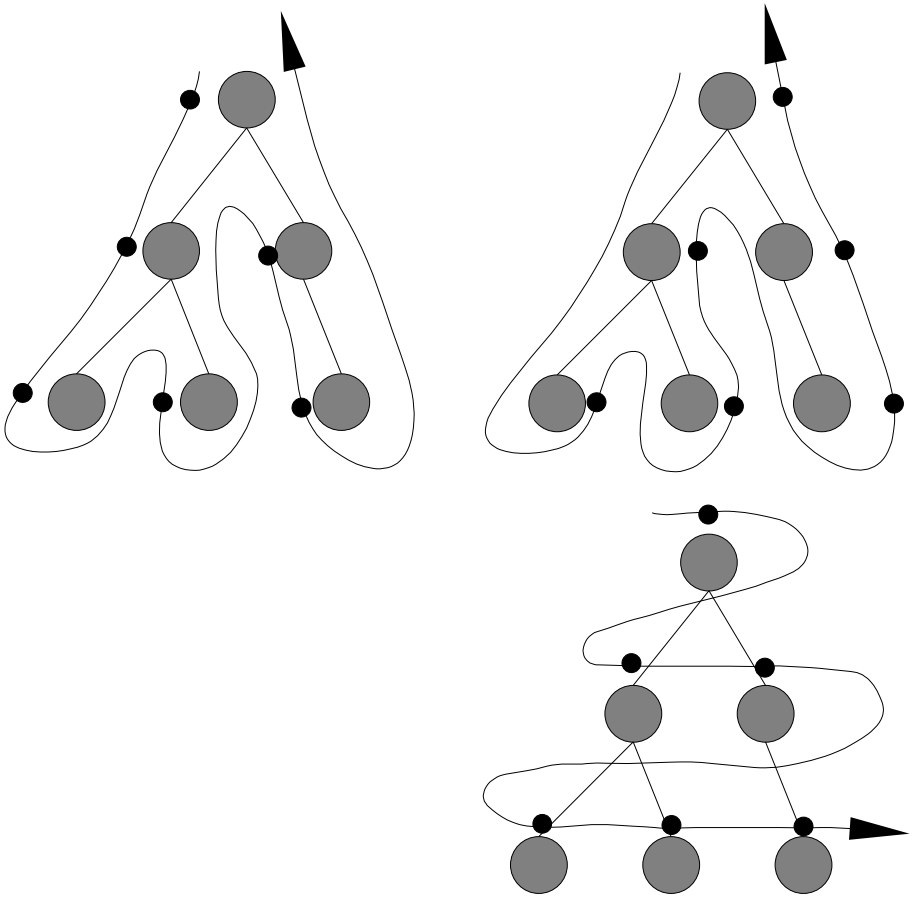
未来は無保証

find -depth

- Unix には find コマンドがある
- find コマンドには -depth オプションがある
- Ruby には find ライブラリがある
- find ライブラリには -depth オプション相当の機能はない
- find ライブラリにその機能を拡張する?

find -depth の失敗

- 名前が間違っている
- 深さ優先: depth first
  - 行きがけ順: preorder
  - 帰りがけ順: postorder
- 幅優先: breadth first



- find は常に深さ優先
- -depth をつけないと行きがけ順
- -depth をつけると帰りがけ順
- 帰りがけ順のオプションが深さ優先という名前

find ライブラリに帰りがけ順をつける

- depth という名前は使わない
- 失敗は繰り返さない

デフォルトで FD_CLOEXEC

- Unix のコマンド起動は 2段階
  - fork で親プロセスを複製して子プロセス生成
  - execve でプロセスの中身をコマンドに置き換える
- オープンした fd は fork で子プロセスに継承される
- execve で継承されるかどうかは FD_CLOEXEC フラグで制御 (継承するのがデフォルト)
- Ruby では fd に片っ端から FD_CLOEXEC をつけるのはどうか

意図しない fd 継承の問題

- 以下で cat コマンドの実行にかかる時間は?
  io = IO.popen("cat -n", "w")
  Thread.new { system("sleep 5") }
  io.puts "a"
  io.close
- おそらく5秒くらいかかる
- sleep に cat につながっている fd が継承される
- sleep が終わるまでは cat は EOF を検出できない
- cat は sleep の終了を待ち、それは 5秒かかる
- もし、system が呼ばれる前に io.close まで済めばすぐ終わる

意図しない fd 継承は避ける

- 継承するのは Unix の伝統だがトラブル
- sleep について fd を継承する必要はどこにもない
- Ruby 1.9 では継承を拒否することができる

```
io = IO.popen("cat -n", "w")
Thread.new {
  system("sleep 10", :close_others=>true)
}
io.puts "a"
io.close
```

- デフォルトでないのは互換性のため
- なお厳密にいえば完璧でない

デフォルトで :close_others=>true ?

- 互換性を無視すれば悪くない
- 実際、IO.popen と spawn ではデフォルト
- デフォルトでないのは system と exec
- でも将来的には足りない
  - MVM (Multiple Virtual Machine)
  - GVLのないネイティブスレッド対応

MVM: Multiple Virtual Machine

- 単一プロセスで複数の VM を動かす計画
- 各 VM はネイティブスレッドで並行に動く (GVL みたいなロックはない)
- ある VM が fd を open したタイミングで他の VM が fork したらどうなる?
- いまは IO.popen の内部の微妙なところでスレッドがスイッチすることはない。しかし MVM なら現実的にありうる
- そこで FD_CLOEXEC
- FD_CLOEXEC な fd は execve で継承されないので問題を防げる

FD_CLOEXEC による解決

- この問題は Ruby だけでなく C でも起こる
- POSIX の解決は O_CLOEXEC と F_DUPFD_CLOEXEC
- open のフラグに O_CLOEXEC を指定すると結果の fd は最初から FD_CLOEXEC が設定される
- open してから FD_CLOEXEC を設定する場合の競合条件がない
- Linux では 2.6.23 からサポート
  他にも dup3 とか pipe2 とか MSG_CMSG_CLOEXEC いろいろ

POSIX のやりかたを Ruby へ

- 案1: File::CLOEXEC で O_CLOEXEC を提供
  - O_CLOEXEC がない環境ではアプリケーションが fcntl で FD_CLOEXEC を設定
  - pipe2 など、定数ひとつでは済まない
  - アプリケーションがちゃんとやってくれるとは思えない
- 案2: 内部的にデフォルトで FD_CLOEXEC を設定
  - O_CLOEXEC のない環境では Ruby 自体が fcntl で FD_CLOEXEC を設定
  - pipe2 などは内部的にあったら使う
  - 非互換性が発生
  - Perl ではそうやっている

有望そうな案

- 中期的にはデフォルトで :close_others => true
- 長期的には内部的に FD_CLOEXEC をデフォルトで設定 (MVM 導入のタイミング?)

openat

- 新しい POSIX で定義されたシステムコール
- カレントディレクトリのかわりに使うディレクトリを fd で指定できる
- unlinkat など、*at というシステムコールがたくさん追加されている
- ディレクトリの再帰的な削除で競合条件を除去できる (セキュリティホールを防ぐのに役に立つ)
- スレッド毎カレントディレクトリを実現することも可能

open と openat

- int open(const char *path, int oflag, ...);
- int openat(int fd, const char *path, int oflag, ...);
- 相対パスの起点を示す fd が増えている
- fd はオープンしたディレクトリ
- カレントディレクトリはプロセスに大域的だが、openat では局所的に指定できる
- Unix の問題点の解決

openat を Ruby へ

open に dir を追加する

- 案1: File.openat(dir, filename, mode, perm)
- 案2: File.open(filename, mode, perm, base: dir)
- 案3: dir.open(filename, mode, perm)
- 案4: open([dir, filename], mode, perm)
- 案5: Pathname に dir を含められるようにする
- 案6: カレントディレクトリをスレッド毎にする

考慮する点 (1)

openat がない環境の扱い

- openat がない環境で、openat の利点が得られないのは許容できる
- アプリケーションで場合分けはしたくない (案1)
- Dir や IO にはパス名も記録されている
  それらで指定すればパス名とfdを同時に指定可能
  openat がなければパス名の連結で処理
- openat のない環境でカレントディレクトリをスレッド毎にするのは難しい (案6)
  GVL を仮定すれば不可能ではない？
  MVM は openat がないと無理

考慮する点 (2)

renameat, linkat の扱い

- int renameat(int oldfd, const char *old,
  int newfd, const char *new);
- リネーム元とリネーム先の両方に別の起点ディレクトリを指定できる
  (linkat もリンク元とリンク先で同様)
- Dir のインスタンスメソッド (案3) はうまくない
- スレッド毎カレントディレクトリ (案6)では扱えない

考慮する点 (3)

プログラマが覚えないといけないことを減らす

- 一貫性のある仕様は覚えやすい
- メソッドをたくさん増やすのは良くない (案1, 3)
- 新しい引数をたくさん増やすのも良くない (案2)
- renameat, linkat で一貫性が保てないのは良くない (案2,3)

考慮する点 (4)

配列の書き換えの危険性

- openat の途中で配列を書き換えられちゃったらどうなる? (案4)
- 対処不可能というわけではないけれど

考慮する点 (5)

互換性

- 必要以上に非互換性を発生させるべきでない
- カレントディレクトリをスレッド毎にするのは非互換性があるかもしれない（案6)

有望そうな案

openat の機能をすべて提供する方向:

- 案4: open([dir, filename], mode, perm)
- 案5: Pathname に dir を含められるようにする

機能を部分的に提供する方向:

- 案6: カレントディレクトリをスレッド毎にする

プロセスID のオブジェクト化

- プロセスもスレッドも実行主体
- 待つ:
  - Process.wait pid
  - thread.join
- 殺す:
  - Process.kill signal, pid
  - thread.kill
- なぜスレッドはオブジェクトなのに pid は整数?
- pid.join と書けるようなオブジェクトにしたい
- open3 ではスレッドを使ったが大げさ

system() 中の割り込み

- 以下で sleep 中に ^C で割り込むと :done は表示されるか？
  % ruby -e 'system("sleep 3"); p :done'
- じつは表示される
- ^C は sleep を殺すが、ruby は死なない
- もともと vi (というか ex) など対話的な環境でユーザが入力したコマンドを動かす場合のための仕様
- 非対話的な場合は ruby も死んでくれたほうがうれしい
- 指定するオプションを作る？

Unixの大失敗: 非同期signal

- 説明するには時間が足りない
- 要約
  - 非同期イベントで任意の機械語命令間で割り込まれる
  - 安全な状態に至るまで待つことができない
  - 安全に使える関数がほとんどなく、安全な状態になるまで待つべき
  - Ruby は VM 的に安全になるまで待つが、ライブラリ的な安全は保証されない
  - 同じ不幸がレイヤが上がって起きている
  - 根本的にどうにかすべき
  - pthread cancel や Java など参考になるものがある

案を練る方法

- 問題を発見する
- 用法を調べる (超重要)
  - 普通のプログラミングの中で発見した問題ならその状況が用法のひとつ
- 他の処理系の解法をサーベイ
  - うまくいっている方法
  - うまくいっていない方法
- Rubyに適用するいろいろな案を考える
- 用法をうまく解決できるか検討する

まとめ

- Ruby は Unix 文化圏
- でも失敗は直す (こともある)
- Ruby 側の都合も考慮する