Ruby I/O 機構の改善
— stdio considered harmful —

田中 哲

akr@m17n.org

産業技術総合研究所 情報技術研究部門

概要

Ruby は開発版 (1.9) において I/O のバッファリングを stdio でなく独自に行うようになった。本稿ではその理由と効果を解説する。stdio は C の標準入出力ライブラリであり、バッファリングを行ってシステムコール頻度を抑えて効率を上げるとともに行指向の入出力などをサポートしている。しかし、バッファの状態を確認するポータブルな方法がないことや、nonblocking I/O に関わる問題に加え、複数の実装間の差異による不都合など、様々な問題が存在する。これらの問題の中には stdio を使っている限り対処が困難なものがあり、stdio を除去することによって多くの問題に対処できる。

1 はじめに

C 言語には入出力機能の標準として、stdio (Standard Input / Output) というライブラリが定義されている。プログラミング言語 Ruby[rub] は stdio をバッファリングに使用する I/O 機構を持っているが、さまざまな問題があるため Ruby 1.9 では stdio を使わないでバッファリングを行うようになった。本稿では、その問題と解決法を解説する。

現時点[1]において、Ruby の安定版は Ruby 1.8 系列であり、開発版は Ruby 1.9.0 である。

著者は以下の方針で Ruby に対して修正を行った。

- Ruby 安定版 (1.8.3) においてバッファリング機構に stdio を使用したままで可能な限りの解決を図った
- Ruby 開発版 (1.9.0) において stdio を捨ててさらに完全な解決を図った

これにより表 1 のように多くの問題が改善された。

2 節で Ruby の I/O システム、3 節 で stdio、4 節 で POSIX の I/O について述べる。5 節 で stdio の様々な問題とその対処を述べ、6 節 でその対処によって起きた問題を述べる。7 節 で今後の課題について述べ、8 節 で他のシステムとの比較を行い、9 節 でまとめる。

2 Ruby I/O システム

Ruby はクラスベースのオブジェクト指向言語であり、I/O は IO クラスによって実現される。Unix 環境で動作する Ruby は図 1 のように、入出力のバッファリングを stdio を利用して実現する。ここで stdio は様々な関数を提供しているが、Ruby で実際の入出力に使用するのは getc, ungetc, fwrite だけである。

[1] 2005–03

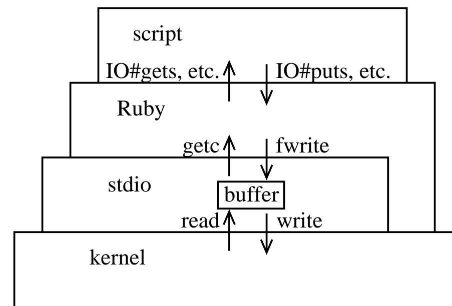

図 1: Ruby の動作レイヤ

IO クラスは stdio の FILE 構造体を内部に持ち、stdio の関数群に似たメソッド群を持っている。また、fcntl, fstat などの POSIX で定義された関数もメソッドとして提供している。

IO クラスは stdio の FILE 構造体を使用してバッファリングを行う。通常は図 2 のようにひとつの FILE 構造体を使用するが、socket などの双方向ストリームでは図 3 のようにふたつの FILE 構造体を使用する。

IO クラスは書き込み時にバッファリングを行うかどうかを選択できる。バッファリングを行わないのを sync mode と呼ぶ。それに対してバッファリングを行うのを buffering mode と呼ぶ。IO インスタンスの初期状態でバッファリングを行うかどうかはインスタンスが生成された方法によって異なる。初期状態はファイルについては buffering mode であり、socket などについては sync mode である。このように違うのは、socket などについては flush を忘れて問題が起こることが多いためである。

また、IO クラスのサブクラスには図 4 のように File

表 1: I/O 機構の変化

	以前の Ruby 1.8	Ruby 1.8 改善版	Ruby 1.9
スレッドの read 待ち	△	△	○
errno をクリアする stdio 実装	△	△	○
ungetc	△	△	○
双方向ストリーム	△	△	○
読み込み・書き込みの切替え	△	△	○
nonblocking write におけるデータ消滅	×	△	○
nonblocking IO#read の動作	×	×	○
readpartial の提案・実装	×	○	○
Solaris の 256 個制限	×	×	○
EOF フラグ	×	△	○
拡張ライブラリの非互換性	○	○	△
テキストモード	△	△	△
popen のポータビリティ	○	○	○
Windows 上の双方向 popen	○	○	×
stdio を使うライブラリとの協調	○	○	×

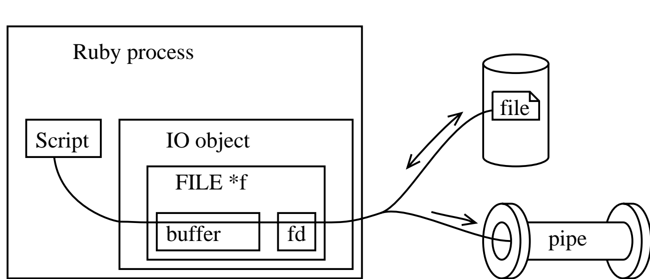

図 2: 双方向ストリーム以外の IO

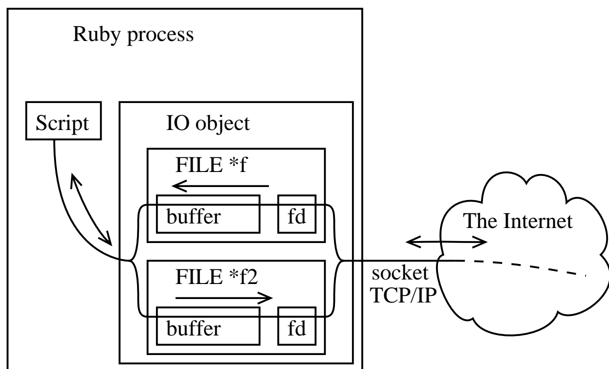

図 3: 双方向ストリーム

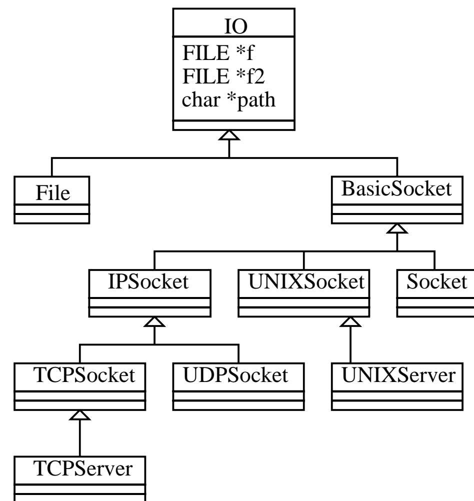

図 4: IO 関連クラス

クラス、socket 関連クラスがある。File クラスは open 時のパス名を保持し、パス名を利用するメソッドを持つ[2]。socket 関連クラスではそれぞれのクラスで socket 特有のメソッドが定義される。

表 2 に stdio の関数と IO クラスのメソッドの対応を示す。Ruby の表記で XXX.xxx というのはクラスメソッド、XXX#xxx というのはインスタンスメソッド

[2] パス名を保持する path メンバは IO クラスで定義されるが、File クラス以外ではファイル名は入らない。

表 2: stdio と IO クラスの対応

C	Ruby
fopen	File.open
popen	IO.popen
fdopen	IO.new
freopen	IO#reopen
fclose,pclose	IO#close
fileno	IO#fileno
fflush	IO#flush
rewind	IO#rewind
fseek	IO#seek
ftell	IO#tell
fgetpos	IO#pos
fsetpos	IO#pos=
fgetc,getc	IO#getc
fgets,gets	IO#gets
fread	IO#read
fputc,putc	IO#putc
fputs,puts	IO#puts
fprintf,printf	IO#printf
fwrite	IO#write
ungetc	IO#ungetc
feof	IO#eof?

を意味する。この表のように、IO クラスは stdio の関数に対応する名前のメソッドを持っており、C プログラマに馴染みやすくなっている。ただし、それらのメソッドは必ずしも対応する関数を使って実装されているわけではない。また、場合によっては動作も微妙に異なる。たとえば、puts 関数と IO#puts メソッドは両方とも一行出力という機能を持っているが、puts 関数が常に改行を追加して出力するのに対し、IO#puts メソッドは与えられた文字列の末尾が改行でないときのみ改行を追加するという違いがある。

2.1 スレッド

Ruby はユーザレベルでスレッドを実装している。この実装は OS の支援無しに実現されており、OS に関わらずポータブルに使用できる。

このユーザレベルスレッドは OS に認識されていないため、I/O などのシステムコールがブロックした場合は、そのシステムコールを発行したスレッドだけでなく、そのプロセス内のすべてのスレッドがブロックする。これを避けるために Ruby では select システムコールを用いて I/O を多重化する。つまり、I/O 操作の前に select でその操作が可能であることを確認し、操作が即座にできない場合は他のスレッドにコンテキストスイッチを行う。この方法は複数プロセスが競合しない読み込み操作に関してはほとんどの場合適切に動作する。

複数プロセスが競合する場合や書き込み操作に関してプロセス全体のブロックを防ぐには 4.3 節で述べる nonblocking I/O を併用する必要がある。このため、そのような場合にはプロセス全体がブロックする。

3 標準入出力ライブラリ stdio

3.1 stdio の機能

stdio はプログラミング言語 C[KR88] の標準入出力ライブラリである。stdio は高水準入出力ライブラリとも呼ばれ、FILE 構造体を使用する I/O ルーチンを中心として、プロセス起動 (popen)、書式付き入出力 (printf, scanf) などの機能を持っている。

I/O ルーチンは FILE 構造体内のバッファを利用し、低水準入出力機能 (read, write システムコール) をラップし、行単位の入出力などの高水準入出力機能を実現している。

3.2 stdio の歴史

stdio は最初 PWB 1.0[3]に含まれてリリースされ、後に Version 7 に搭載された [DC84]。このライブラリについては『詳解 UNIX プログラミング [Ste00]』に以下の記述がある。

> 標準入出力ライブラリは 1975 年頃に Dennis Ritchie が作成した。これは、Mike Lesk が書いたポータブルな入出力ライブラリの主要な改定版であった。驚くべきことであるが、15 年以上にわたって標準入出力ライブラリはほとんど書き直されていない。

つまり、Version 7 を祖先に持つ Unix、つまりほとんどの商用 Unix は Dennis Ritchie の stdio ライブラリを元にした実装を使用している[4]。

ただし、近年の自由 Unix は異なる stdio 実装を用いている。

- GNU/Linux が使用している stdio は glibc に含まれており、これは Per Bothner が実装したものが元になっている [Perb]。
- 4.4BSD が使用している stdio は Chris Torek が実装したものである [CSR]。4.4BSD を元にしている Unix には NetBSD, FreeBSD, OpenBSD などがある。

また、stdio は ANSI C[Ans89] として規格化されており、C の実装には stdio を含めなければならない。したがって、非 Unix 環境における C 言語にも stdio は存在し、様々な実装がある。

[3]PWB 1.0, Version 7 などの関係については [Unib] を参照のこと。

[4]たとえば Version 7 と Solaris の FILE 構造体を比べると、メンバの名前、型、順序について類似性が認められる。

4　POSIX I/O

4.1　基本 I/O システムコール

Unix における基本的な入出力機能は POSIX で定義されている。open, read, write, close が基本的なシステムコールである。基本的な使用法は open でファイルに対して fd (file descriptor) を割り当て、fd に対して read, write で読み込み・書き込みを行い、close で fd を解放するというものである。

read, write は次のように宣言される。

```
ssize_t read(int fd, void *buf, size_t nbyte);
ssize_t write(int fd, const void *buf, size_t nbyte);
```

ここで fd は open などによって与えられる file descriptor で外部資源を同定するものであり、read では読み込み元、write では書き込み先として使用される。buf, nbyte はバッファであり、read では読み込んだデータの格納先、write では書き込むデータを格納してある場所である。返値は読み込めたサイズ・書き込めたサイズであり、エラー時には -1 となると同時に errno が設定される。また、read において EOF が検出された場合は返値が 0 になる。なお、(blocking mode において) read は 1 バイト以上データを読み込めるまでブロックするのに対し、write はすべてのデータを書き込むまでブロックする。低水準入出力 read, write の挙動を疑似コードで付録 A に示す。

また、非 Unix 環境で改行コードが異なる場合、text mode の変換はこのレイヤで行われることが多い。

4.2　select

POSIX には、入出力の多重化を行う select システムコールがある。select は fd に対する I/O がブロックせずに実行できるかどうかを調べるシステムコールである。これにより、プロセスは I/O でブロックするかわりに他の処理を行うことができ、ユーザレベルのスレッド実装などに使われる。

ただし、ある fd が select によって書き込み可能であることが示された場合、ブロックせずに書き込めることが保証される量は fd が指す対象によって異なり、それ以上の量を書き込もうとした場合にはブロックする可能性がある。このため、書き込み時のブロックを避けるためには次に述べる nonblocking I/O を併用する必要がある。また、複数のプロセスがひとつの fd から読み込みを行う場合もブロックを避けるためには nonblocking I/O を併用する必要がある。これは select の時点と read の時点の間に状況が変化する可能性があり、select の時点では読み込み可能であっても、read の時点ではそうではなくなっている場合があるためである。

4.3　nonblocking I/O

nonblocking I/O は、fcntl システムコールによって fd を nonblocking mode と設定できる機能である。nonblocking mode ではシステムコールがブロックしなくなる。nonblocking mode でないときにブロックしてしまう状況では nonblocking mode では代わりに EAGAIN エラーとなる。nonblocking mode に対して通常の mode を blocking mode と呼ぶ。

nonblocking mode かどうかは file table に記録される。そのため、dup や fork によって file descriptor が複製された場合 nonlocking mode は共有されるが、同じファイルを複数回 open した場合には共有されない。

4.4　stdio と nonblocking I/O

nonblocking I/O が導入されたのは Version 7 よりも後である [Unia]。つまり、stdio の設計時には nonblocking I/O は存在しなかったため、そのデザインは nonblocking I/O を想定していない。そのように想定していなかったものを stdio と組み合わせて使用すると、Stevens が『破滅の処方』と呼んだように [Ste99]、様々な問題が発生する。

5　Ruby における stdio の問題と対策

本節では、stdio にかかわる様々な問題への対策について個々に述べる。

5.1　スレッドの read 待ち

stdio は FILE 構造体内の読み込みバッファにデータが貯っているかどうかを判定する機能を提供していない。このため、getc などの読み込み関数が読み込み待ちでブロックするかどうかを効率良く判定することができない。

getc がブロックするのは読み込みバッファが空であり、かつ、read システムコールがブロックするときである。後者は select システムコールにより判断できるが、前者は stdio にその機能がない。したがって、1 バイト毎に select を呼べばブロックするかどうかを判定できるが、これは効率が悪い。本来はバッファが空であるときだけ select を呼べば済むはずである。

Ruby はスレッド機能を提供している。このため、あるスレッドが読み込み操作でブロックしているときには他のスレッドが動作する。これをユーザレベルで実現するためには、読み込み操作がブロックするかどうかを内部的に判断し、ブロックするときには他のスレッドにコンテキストスイッチする必要がある。

しかし、前述のように読み込み操作がブロックするかどうかをポータブルに判断すると効率が悪くなる。そのため、ポータブルではないが、FILE 構造体の中身を参照してバッファが空であるかどうかを調べることによってブロックするかどうかを判断している。

この方法はポータブルではないため、5 種類の実装を想定して configure により FILE 構造体の中身を判断している。しかし、この方法では想定されていない実装では動作できないし、また、64bit Solaris など FILE 構造体の中身がヘッダファイルに記載されていない環境[5]はサポート困難である。

Ruby 1.9 ではバッファリングを独自に行うようになったため、読み込みバッファが空であるかどうかを

[5] 64bit Solaris の FILE 構造体の内容は `long __pad[16];` と定義されている。

ポータブルに判断することができるようになった。これにより 64bit Solaris などでも適切に動作できるようになった。

5.2 errno をクリアする stdio 実装

stdio の実装によっては、stdio の関数内でシステムコールがエラーになったときに設定される errno をクリアしてしまうことがある。たとえば、HP–UX において fd が nonblocking mode のときに fwrite を使用し、書き込み途中で EAGAIN で失敗した場合、fwrite の終了時に errno が 0 となる[6]。このため、呼び出し側では失敗の原因が EAGAIN であることがわからない。

EAGAIN により失敗した場合、続きを書き込めるようになった後に再度書き込まなければならない。しかし、EAGAIN になったことがわからなければ、その動作を行うという判断が出来ない。

Ruby では HP–UX のときに限り、fwrite が失敗し errno が 0 であるときには EAGAIN で失敗したものと想定するという ad hoc な対処を行っていた。しかし、この対処は他のエラーで失敗した場合に不適切な動作を行うことになる。

しかし、stdio を使わずに直接 write システムコールを呼べば、実際の errno を確実に確認できる。Ruby 1.9 では、そのようにすることにより、ad hoc な対処を取り除き、適切な解決を行うことが出来た。

5.3 ungetc

IO#ungetc メソッドは C の ungetc 関数で実装されている。

ここで、規格では ungetc 関数が正しく動作するタイミングは限られている。たとえば、ungetc で戻せることが保証されているのは 1 バイトだけであり、2 回以上続けて ungetc を呼び出すことはできない。また、書き込みの直後で動作する保証もない。しかし、stdio は任意の時点で ungetc が動作可能かどうかを判断する機能を提供していない。

このため、IO#ungetc が単純に ungetc を呼び出すだけの実装では、不適切なタイミングで ungetc を呼び出すことが出来てしまう。そして、不適切な状態で ungetc を呼び出すと、Segmentation fault がおこる場合もある[7]。これを避けるためには、ungetc が動作できるかどうかを保持するフラグを管理しなければならない。

しかし、ungetc が動作可能であるかどうかは、本質的には読み込みバッファに空きがあるかどうかで決まる。そのため、付加的なフラグで管理するのは冗長であり、バグの入り込む余地を広げてしまう。Ruby 1.9 では独自にバッファリングを行うためバッファの空きを確認でき、フラグを管理しなくても IO#ungetc を安全に実装できるようになった。

5.4 双方向ストリーム

TCP ソケットや端末のような双方向ストリームをバッファリングする場合、読み込みバッファと書き込みバッファを別々に持たなければならない。これに対し通常のファイルではバッファはひとつでよい。この違いは、読み込みバッファが空でない状態で書き込み操作を行った場合、双方向ストリームでは読み込みバッファをそのままにしておかなければならないのに対し、通常のファイルでは読み込みバッファを破棄してファイルポインタを戻さなければならないためである。

FILE 構造体はバッファをひとつしか持っていないため、Ruby の IO オブジェクトは双方向ストリームを図 3 のようにふたつの FILE 構造体を使用して扱う。

しかし、Ruby は FILE 構造体をふたつ使うかひとつで済ますかは IO オブジェクトの生成のされかたで判断していた。たとえば、TCPSocket.new で生成したときには双方向ストリームとしてふたつの FILE 構造体を使用し、File.open で生成したときには通常のファイルとしてひとつの FILE 構造体を使用する。このため、端末などの双方向ストリームなデバイスファイルを File.open でオープンすると、通常のファイルと判断してひとつの FILE 構造体しか使わず不適切な動作をしてしまう。

Ruby 1.9 では双方向ストリームかどうかは IO オブジェクトの生成時には判断せず、異なる方法によって判断する。判断するタイミングは読み込みバッファが空でない状態で書き込み操作を行ったときである。このとき、通常のファイルであれば読み込みバッファを破棄して lseek でファイルポインタを戻してから書き込みを行わなければならず、また、双方向ストリームであればそのまま書き込みを行わなければならない。そこで、Ruby 1.9 はまず通常のファイルと仮定して lseek を行い、それが成功した場合は通常のファイルと判断し、バッファを破棄して書き込みを行う。そして、lseek が失敗した場合は双方向ストリームであると判断し、バッファを破棄せずに書き込みを行う。

この判断は stdio でバッファリングを行っている場合には実装できない。これは fseek が失敗した場合、読み込みバッファの内容が破棄されてしまうためである。

なお、4.4BSD の端末では lseek が失敗しないため、IO オブジェクトの生成時に isatty により端末かどうかを判定し、端末の場合には始めから双方向ストリームとして扱う。

5.5 読み込み・書き込みの切替え

規格では、stdio の読み込み関数と書き込み関数を連続して呼び出してはならないことになっている。書き込みの後に読み込みを行う場合には fflush, fseek などで書き込みバッファを空にして初期状態に戻さなければならない。また、読み込みの後に書き込みを行う場合には fseek などで読み込みバッファを空にして初期状態に戻さなければならない。

この制約は stdio の読み込みバッファと書き込みバッファがひとつのバッファを共有しており、実装によってはバッファがどちらであるかを考慮しないためである。伝統的に Ritchie の stdio はこれを考慮しないため、読み込み・書き込みを連続して行うと問題が生じる。ただし、4.4BSD や glibc の stdio はこれを考慮して自動的にバッファを空にするため、読み込み・書き込みを連続して行っても問題はない。

stdio は書き込み・読み込みを行ってもよいかどうかを判断する機能を提供しない。このため、その判断を

[6][ruby–dev:22545]
[7][ruby–dev:22330]

行うにはアプリケーションがフラグを管理しなければならない。

Ruby 1.9 では読み込みバッファ・書き込みバッファを別々に持ち、それぞれが空であるかどうかも判断できる。これにより、フラグを管理しなくても書き込み・読み込みを行うにあたって適切な動作を確実に判断できる。

5.6 nonblocking write におけるデータ消滅

書き込みバッファ内のデータが flush されて書き込まれるときに EAGAIN で失敗した場合、stdio の実装によってはまだ書き込まれていないデータが捨てられてしまうことがある。

この問題は nonblocking mode において起きるが、stdio でバッファリングを行う限りは解決困難である。これは、EAGAIN で失敗するかどうかは事前に調べることができず、また、事後にはすでにデータが捨てられてしまっていて復旧できないためである。

Ruby 1.9 は独自にバッファリングを行うため、EAGAIN で失敗してもデータを捨てないという動作を実装でき、素直に解決できる。

Ruby 1.8 は stdio でバッファリングを行うため、素直な解決は出来ない。そこで、sync mode に限っては stdio のバッファリングを迂回するという解決を図った。いままでの sync mode は書き込み操作のたびに fflush することによって実装されていたが、これを止めて直接 write システムコールで書き込むようにした。これにより、sync mode であれば nonblocking mode でもデータを捨てることはなくなった。buffering mode では依然として問題が残っているが、ソケットやパイプはデフォルトで sync mode となるので、多くの場合の問題を解決できたと考えられる。

5.7 nonblocking IO#read の動作

Ruby 1.8 において IO#read メソッドは fd が nonblocking mode かどうかで動作が以下のように変化する。

- blocking mode の場合には指定した長さのデータを読み込む。このときに、パイプなどで指定した長さのデータが到着していなくて即座に読めない場合は、指定した長さのデータが得られるまでブロックする。

- nonblocking mode の場合には指定した長さを上限としてその時点で読み込めるだけ読み込む。このときに、パイプなどで指定した長さのデータが到着していなくても即座に読み込めるぶんだけを結果とする。なお、データをまったく読み込めなければ EAGAIN 例外を発生する。

この挙動の変化は「その時点で読み込めるだけ読み込む」という用途に対応するためのものである。しかし、メソッドの動作が状態によって変化することはプログラミングを難しくする。これはプログラムの内部に nonblocking mode が必要な箇所と blocking mode が必要な箇所の両方があった場合、mode を適切なタイミングで設定しなければならないためである。また、nonblocking mode において即座に読み込めるデータがなかったときに例外を発生するという挙動は busy loop を誘発するという問題もある。

そして、適切なタイミングで nonblocking mode を設定することは一般には難しい。これは、nonblocking mode は IO#read だけでなく他の操作にも影響を与えるためであり、また、スレッド・プロセス間で共有されるグローバルな属性であるためである。ここで、nonblocking mode を必要とする例としては、write システムコールでプロセス全体がブロックするのを防ぐことがあげられる。しかし、これを防ぐために nonblocking mode を設定すると IO#read にも影響が及ぶ。そのため、nonblocking mode を設定した場合には IO#read の呼び出しをすべて確認して blocking mode の IO#read の動作を期待している場合には対処しなければならない。とくにライブラリ内部で IO#read が使用されている場合、対処にはライブラリの変更を必要とするため困難である。

また、nonblocking mode はスレッド・プロセス間で共有されるグローバルな属性であるため、複数のスレッド・プロセスが異なる mode を期待する場合、すべての期待を満たすのは困難である。たとえば、stdio は nonblocking mode の fd に対しては不適切に動作する。そのため、stderr のような多くのプロセスで共有される fd を nonblocking mode に設定すると、様々なプロセスで問題が生じる。また、ある Ruby プロセスがある fd が blocking mode であることを期待したとしても、他のプロセスが nonblocking mode に変更した場合、その影響を受けてしまう。

そして「その時点で読み込めるだけ読み込む」という用途は select と 5.8 節で述べる readpartial を用いれば nonblocking mode を使用しなくても実現可能である。以下に疑似コードで実現例を示す。

```
def IO#read_nonblock(maxlen)
  if IO.select([self], nil, nil, 0)
    readpartial(maxlen)
  else
    raise Errno::EAGAIN
  end
end
```

このように利点がないにも関わらず問題が多いため、Ruby 1.9 では nonblocking mode か blocking mode かに関わらず IO#read はたとえブロックしても指定した長さのデータを読み込むという挙動に変更した。これにより、プログラムの開発過程において fd を後から nonblocking mode に設定する必要性が判明して設定した場合でも、既存のプログラム内の IO#read の挙動は変化せず、プログラムが壊れないことが保証される。また、他のプロセスが fd を nonblocking mode に変更したとしてもその fd を共有している Ruby プロセスはプログラマが期待する blocking mode の挙動のままで動作し、プログラムは期待通りに動作する。

5.8 readpartial の提案・実装

ある種のアプリケーションでは「読み込めるようになるまでブロックし、その時点で読み込めるだけ読み込む」機能を必要とする。この機能は、読み込むデータの長さが予期できず、かつ、読み込むデータの末尾

に改行などのマークがついていないときに必要になる。このような状況は port forwarder で到着したデータを読み込むとき、CVS プロトコルで圧縮された 1 行を読み込むとき、リモートホストのシェルと対話してプロンプトまで読むときなど、さまざまなものがある。

この「読み込めるようになるまでブロックし、その時点で読み込めるだけ読み込む」という機能はほぼ read システムコールと同じであるが、stdio にはそのような機能は用意されていない。そのため、stdio のバッファを経由してそのような動作を行うメソッドは用意されていなかった。これにより、Ruby でこの機能を実現するには、1 バイトづつ select でデータが存在することを確かめてから読み出すか、IO#sysread を使う必要があった。しかし、これらにはそれぞれに問題がある。まず、1 バイトづつ読み出すのは効率が悪いという問題がある。また、IO#sysread は stdio を使用せずに直接システムコールの read を呼び出すため、stdio のバッファに入ったデータを読み出せない。このため、stdio のバッファを使用するメソッドと混用できないという問題がある。

そこで、新規メソッド readpartial を提案・実装した。このメソッドの簡易版の動作を疑似コードで以下に示す。

```
def IO#readpartial(maxlen)
  if 読み込みバッファが空
    begin
      sysread(maxlen)
    end until 1 バイト以上読み込んだ
  else
    読み込みバッファから最大 maxlen 読み込む
  end
end
```

つまり、このメソッドはバッファが空でないときはバッファから読み、そうでないときは read システムコールを呼び出す。

この readpartial を Ruby 1.8 と Ruby 1.9 の両方に実装した。ここで Ruby 1.8 は stdio でバッファリングを行っているため、読み込みバッファが空かどうかを判定する機能を stdio が提供していないことが問題となる。しかし、Ruby の場合には 5.1 節で述べたようにその機能をすでに実装しているため、stdio を使っていてもポータビリティを低下させずに実装可能である。Ruby 1.9 は独自のバッファリングを行っているためそのような問題はなく、素直に実装できる。

5.9 Solaris の 256 個制限

Solaris の FILE 構造体には fd が 256 未満でなければならないという制限がある。これは FILE 構造体の fd を保持するメンバが unsigned char 型であるためである。しかし、OS がプロセスに提供する fd の最大数は resource limit (RLIMIT_NOFILE) によって決まり、この制限は 256 よりも大きくすることが可能である。つまり、stdio を使う限り、資源を OS が提供するにもかかわらず使用できない場合がある。

Ruby 1.9 ではこの制限を回避するため、FILE 構造体を必要としない限り生成しないことにした。これにより、FILE 構造体が必要ない場合には RLIMIT_NOFILE の限界まで IO インスタンスを生成できるようになった。

5.10 EOF フラグ

stdio の FILE 構造体内には EOF が既に返されたかどうかを示すフラグが保持されており、feof 関数で読み出せる。たとえば、getc が EOF を返したときにこのフラグは真に設定される。ここではこのフラグを EOF フラグと呼ぶ。

しかし、C FAQ の項目 [CFA] にもなっているように、EOF フラグは混乱を招くことが多い。典型的な誤解は、それ以上読み込めるデータがないときに真になるというものである。しかし、実際には読み込めるデータがないだけでは真にはならず、さらにその後で読み込み操作を行って EOF が返されたときに真となる。このようにユーザが期待するものとは異なる機能であるため、Ruby では feof に直接対応する機能を持つメソッドは提供していない。そのかわりに、ユーザが期待する機能 (上記の典型的な誤解) を実現するメソッドを IO#eof? として提供している。つまり、IO#eof? メソッドはファイルポインタがファイルの最後を指しているなど、それ以上データが読み込めないときに真を返すメソッドである。

また、EOF フラグが真のときにデータを読み出せるかどうかは stdio の実装によって異なる。この差異は、次のような状況で問題となる。

1. あるファイルを fopen して FILE 構造体 f を得る
2. そのファイルを getc(f) が EOF を返すまで読み込む
3. 他のプロセスがそのファイルにデータを追加する
4. そのデータを getc(f) で読み出す

このような場合、最後の getc(f) 呼び出しのときには f の EOF フラグは真になっている。このとき、getc が実際のファイルを調べずに即座に EOF を返す実装と、そうはならずにデータを読み出せる実装がある。たとえば 4.4BSD stdio ではデータを読み出せない。逆に glibc stdio ではファイルが長くなっていればデータを読み出せる。この挙動は `tail -f` のようなプログラムを記述するときに重要となり、ポータブルに動作させるためには読み込む前に clearerr 関数で EOF フラグをクリアしなければならない。

Ruby の実装では、内部で EOF フラグを不適切に使用している箇所が存在した。

まず、IO#read メソッドの返値が EOF フラグに依存して空文字列 "" と nil に変化するという挙動があった。IO#read は引数を省略した場合、EOF に到達するまでファイルの残りすべてを読み出すという動作をする。ここで、ファイルが空であるなどファイルに残りがない場合に、EOF フラグが偽の場合には "" を返した上で EOF フラグを真に設定し、真の場合には nil を返すように実装されていた[8]。このように複雑な挙動は理解が難しく、とくに IO クラスと互換なクラスを実装するときに問題となる。実際、IO クラスと多態であることを意図して実装されたクラスには StringIO, Zlib::GzipReader, OpenSSL::SSL::SSLSocket などがあるが、これらの EOF フラグの動作はすべて IO クラスと非互換性な部分があった。

また、前述の通り stdio の実装によっては EOF フラグが真のときにデータを読み出せない。そして、clearerr

[8] この挙動は Ruby 1.8.1 以降のものである。Ruby 1.8.0 ではさらに IO#read を呼び出した時点でファイルポインタを取得し、その値にも依存して返値が変化していた。

関数を呼び出すためのメソッドが IO クラスに用意されていないため、EOF フラグを偽にクリアすることが難しい。このため、`tail -f` のようなプログラムを記述することが困難であった。

このように EOF フラグには問題が多いにもかかわらず利点が見当たらなかったため、Ruby 1.8 では IO#read を変更して EOF フラグへの依存を低減した。具体的には、IO#read で引数を省略した場合には常に文字列を返すようにした。これにより、IO 互換なクラスの実装が容易になった。また、EOF フラグを常に偽にクリアすることも検討している。

また、Ruby 1.9 の IO オブジェクトでは EOF フラグの実装を除去した。これにより、IO 互換なクラスの実装が容易になり、`tail -f` のようなプログラムを記述することも問題なくなった。また、IO クラスのメソッドを C で記述する場合に、たとえ間違っても EOF フラグを使ってしまうことができなくなった。

6 stdio の除去によって起こる問題

前節で述べたように、バッファリングを独自に行うことによって様々な問題が解決できたが、いくつか逆に問題が発生した。それらの問題について以下に述べる。

6.1 拡張ライブラリの非互換性

Ruby の拡張ライブラリに非互換性が生じた。これは、拡張ライブラリから IO インスタンス内の FILE 構造体にアクセスすることができたためである。Ruby 1.9 では IO インスタンスが FILE 構造体を持つとは限らないため、そのようなことを行う拡張ライブラリは動作できない。そのような拡張ライブラリはコンパイルエラーとなる。

このような拡張ライブラリの具体例としては、標準添付のものとしては io/wait, openssl, pty, socket があり、非標準添付のものとしては少なくとも ruby/GD, termios がある。

この問題に対しては、典型的には以下のどちらかの対処で解決できる。

- fileno で fd を取り出すだけの用途に FILE 構造体を使用している場合、かわりに fd を保持しているメンバを使用する。
- FILE 構造体がどうしても必要な場合、FILE 構造体を生成して使用する。

なお、Ruby 1.9 は開発版であり、ある程度の非互換性は許容されているため、この非互換性は問題視されていない。

6.2 テキストモード

Ruby 1.9 でのバッファリングはテキストモードを考慮せずに実装した。つまり、open 時にユーザの指定通りにテキストモードを指定する他はテキストモードの処理は実装しなかった。このため、テキストモードで何らかの問題が起こる可能性があることが予想されていた。

結果としては、読み込みバッファを破棄してファイルポインタを戻すときに問題が生じることが判明した。これは、バッファに入っているデータのサイズだけファイルポインタを戻すのに、バッファに入っているデータは改行コードが LF に変換されているため、Windows など改行コードが CRLF という 2 バイトである環境では、サイズが変化して戻す量を間違えてしまうためである。

この問題は、読み込みバッファを破棄しないかぎり起こらない。したがって、シーケンシャルに読み込む場合など、読み込みバッファの破棄が不要な場合には破棄しないことによって問題を避けられる。読み書き両用など、バッファの破棄が必要になる場合は OS のテキストモード実装に頼っている限りは解決困難である。

6.3 popen のポータビリティ

stdio にはプロセスを起動して通信する popen という関数があるが、この関数は FILE 構造体へのポインタを返値とするため、FILE 構造体を使わないためには popen は使用できない。しかし、この関数はポータブルには再実装できない。そのため、個々の環境に対して実装するようにすると、新しい環境に Ruby を移植するときに必要な作業を増やしてしまう。

そこで、fall back の実装として popen を使用できるようにした。つまり、プロセスの起動と終了は popen, pclose を使用するが、起動直後に fileno で fd を取り出し、実際の通信時には FILE 構造体を使用しないという実装を行った。これにより、popen があればポータブルに動作するという性質を保てた。

6.4 Windows 上の双方向 popen

Unix および Windows 版の Ruby 1.8 では、IO.popen でプロセスを起動するときに読み書き両用モードを指定でき、双方向通信ができる。

これは、プロセスからの読み込み用と書き込み用のそれぞれに単方向パイプを使用し、stdio の popen のかわりに直接 fork などを利用して実装している。

しかし、この実装では読み込み用と書き込み用のふたつの fd を IO オブジェクトが持つことになり、fcntl などの読み込み・書き込み以外の操作の動作が不明瞭になるという問題がある。

そこで、Ruby 1.9 では socketpair を用いてひとつの fd で双方向通信を実現した。socketpair を使用して双方向通信を実現するのは 4.4BSD の popen でも行われており、問題ないと判断したためである。

しかし、Windows には socketpair がないことが後から判明し、Windows では IO.popen での双方向通信ができないことになった。ただし、1.8 でも Unix と Windows 以外では双方向通信は使えないため、ポータブルなプログラムではいずれにせよこの機能は使うべきでない。

6.5 端末と EOF フラグ

端末における EOF は入力の終了ではない。端末からプログラムに EOF を送るには、たとえばキーボードで `^D` (CTRL-D) を押下するが、そのあとでも入力

は出来るし、プログラムが端末から読み込めばその入力を受け取れる。この性質を利用するプログラムは例外的であるが、たとえば csh などの対話的なシェルには端末から EOF が送られても終了せずに対話を継続する機能がある。

シェルなどの例外を除く大部分のプログラムでは EOF を受け取った場合それを入力の終了と解釈するため、その時点以降は端末から読み込むべきでない。読み込んだ場合、端末から余計に入力を受け取ってしまうことになるし、入力がないときにはブロックしてしまう可能性がある。つまり、プログラムへの入力を終了するのに ^D を複数回押下しなければならない場合がある。

このように EOF 以降に読み込まないことを保証するには、EOF を受け取った以降は読み込み操作を行わないというフラグを用意すれば良い。このためには stdio の EOF フラグが使用できる。しかし、EOF フラグは 5.10 節 で述べたように除去したため使用できない。そのため、必要ならば同様の機構を再実装しなければならない。なお、将来的には EOF フラグを再実装し、端末に対してだけ使用することも考えられる。

6.6 stdio を使うライブラリとの協調

C のライブラリが stdio を用いてバッファリングしている場合、そのライブラリと Ruby が異なるバッファを使用するために問題が起こることが懸念されている。

しかし、現在のところ実際の問題は発見されていない。

7 将来課題

Ruby の I/O 機構はまだ改善の余地がある。考えられる改善を以下に述べる。

7.1 安全な signal 処理

Ruby には signal が到着したときに任意の処理を実行できるという trap という機能がある。

一般に、signal が到着するタイミングは予想できない。そのため、signal handler で任意の処理 (再入可能でない処理) を行うのは危険である。たとえば、データ構造を変更している最中に signal handler が起動し、その中でそのデータ構造を使用する処理を行うと予期しない結果となり得る。したがって、trap によって実行される処理は、実行しても安全な状態になるまで遅延しなければならない。

しかし、この遅延は十分に短くなければならない。とくに、I/O によるブロックが終るまで遅延するのは許容できない。

この要求を満たし、signal を遅延させるが遅延させすぎずに処理することは一般には困難である。しかし、nonblocking I/O を使用する方法を含め、いくつかの方法で実現可能であることが知られている。

nonblocking I/O は 5.6 節の問題などで使用困難であったが、Ruby 1.9 では使用可能になっている。そこで、今後 Ruby にとって適切な signal の処理方法を検討し、提案・実装していく必要がある。

7.2 nonblocking I/O 非依存性の向上

5.7 節で述べたように、nonblocking mode かどうかで動作が変わるメソッドには問題がある。Ruby 1.9 では IO#read に対処したが、他の動作が変わるメソッドとして、TCPServer#accept がある。

これも同様に対処する必要があるか検討しなければならない。

7.3 速度向上

今回 Ruby 1.9 で実装したバッファリングは素朴なものである。

それに対し、各 OS の stdio ではさまざまな工夫によって速度向上を図っている。そのような工夫を導入できないかどうか検討する必要がある。そのような工夫にはたとえば以下のものがあげられる。

- バッファをメモリのブロックバウンダリに合わせる
- ファイルとの入出力においてファイルのブロックバウンダリに合わせてバッファリングを行う

8 関連研究

8.1 Perl

Perl はコンパイル時にいくつかの IO システムを選択できる。5.8.0 からは PerlIO[Pera] がデフォルトであり、それ以前は stdio を使用するのがデフォルトであった。つまり、Ruby と同様に stdio を捨てて独自の機構に変更した経緯がある。

Ruby 1.9 と PerlIO を比較すると、PerlIO のほうが高機能である。たとえば、PerlIO では複数のレイヤを階層化できるなどの機能がある。それに対して、Ruby 1.9 は単純にバッファリングを再実装しただけであり、レイヤを構成する場合には Ruby 自身のオブジェクト指向機能によって行われる。

8.2 Python

Python は File オブジェクトを通して I/O を行うが、Python 2.4 の時点では File オブジェクトは stdio を使っている。ただし、Python の作者が stdio の問題を認識している発言[9]はあるため、将来的に捨てる可能性はある。

また、Ruby と異なり FILE 構造体の内容への直接アクセスは行っていないが、Python のスレッドは OS が提供するスレッド機構を使用しているため、5.1 節で述べた問題は解決されている。ただし、スクリプトレベルで select による I/O の多重化を行うライブラリでは依然として問題になる。そのため、asyncore など、そのようなライブラリではシステムコール (os.read など) を直接使用しなければならず、stdio のバッファを使用するメソッドは使用できない。

[9]comp.lang.python, 1999-02-09,
<199902090403.XAA24154@eric.cnri.reston.va.us>

8.3 Gauche

Gauche は Gauche 0.5.3 までは stdio を使用していたが、0.5.4 からは独自のバッファリングを行っている [Kaw]。独自バッファリングを行うようになった理由としては、「ポータブルに char-ready?を実装する方法がない」「データ長を正確に指定した読み込みがやりにくい」「マルチバイト文字列絡みの最適化がやりにくい」という 3 つの理由があげられている。最初の理由は 5.1 節で述べたものと同じである。ふたつめの理由はいまのところ Ruby で必要だという要求はない。最後の理由は、Ruby ではまだ I/O 機構にはマルチバイトの処理が入っていないため必要ないが、将来的には必要になることが予想される。

9 まとめ

本稿では、Ruby が stdio によるバッファリングを捨てた様々な理由について述べた。大きな問題は、バッファが空であるかどうかなどの状況を調べる機能が stdio に用意されていないことである。また、stdio の設計時には存在しなかったことから仕方のない面もあるが、nonblocking I/O との相性が悪いのは致命的である。このような問題を避けるために Ruby 1.9 はバッファリングを独自に行うようにした。

参考文献

[Ans89] American National Standard for Information Systems – Programming Language C, 1989. X3.159-1989.

[CFA] C FAQ 12.2: Why won't the code " while(!feof(infp)) fgets(buf, MAXLINE, infp); fputs(buf, outfp); " work? http://www.eskimo.com/~scs/C-faq/q12.2.html.

[CSR] Computer Systems Research Group 1979 – 1993. http://www.netbsd.org/People/CSRG-contrib.html.

[DC84] Ian F. Darwin and Geoffrey Collyer. A History of UNIX before Berkeley: UNIX(R) Evolution, 1975-1984, 1984. http://www.daemonnews.org/199903/history.html.

[Kaw] Shiro Kawai. Gauche:BufferedIO. http://www.shiro.dreamhost.com/scheme/wiliki/wiliki.cgi?Gauche%3aBuffer%edIO.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, Inc., 1988.

[Pera] C API for Perl's implementation of IO in Layers. http://perldoc.perl.org/perliol.html.

[Perb] Software by Per Bothner. http://per.bothner.com/software/.

[rub] Ruby home page. http://www.ruby-lang.org.

[Ste99] W. Richard Stevens. UNIX ネットワークプログラミング第 2 版 Vol.1. ピアソン・エデュケーション, 1999. 篠田陽一 訳.

[Ste00] W. Richard Stevens. 詳解 UNIX プログラミング. ピアソン・エデュケーション, 2000. 大木敦雄 訳.

[Unia] Unix FAQ part4: How do I check to see if there are characters to be read without actually reading? http://www.faqs.org/faqs/unix-faq/faq/part4/section-2.html.

[Unib] Unix History. http://www.levenez.com/unix/.

APPENDIX

A read, write の挙動

```
#define CHECK_SIGNAL \
  if (signal が到着している) { \
    signal handler を呼ぶ \
    if (転送がすでに始まっている) \
      return 転送量; \
    if (SA_RESTART が設定されていない) { \
      errno = EINTR; \
      return -1; \
    } \
    goto start; \
  }

ssize_t read(int fd, void *buf, size_t n)
{
  start: CHECK_SIGNAL
  if (fd は nonblocking mode) {
    if (カーネル内バッファにデータがない) {
      errno = EAGAIN;
      return -1;
    }
  }
  else {
    while (カーネル内バッファにデータがない)
      CHECK_SIGNAL
  }
  /* カーネル内バッファにはバイト列か EOF がある */
  if (EOF がある)
    return 0;
  else {
    カーネル内バッファから転送する (最大 n バイト)
    return 転送量;
  }
}

ssize_t write(int fd, const void *buf, size_t n);
{
  start: CHECK_SIGNAL
  if (fd は nonblocking mode) {
    if (カーネル内バッファに空きがない) {
      errno = EAGAIN;
      return -1;
    }
    else {
      カーネル内バッファへ転送する (最大 n バイト)
      return 転送量;
    }
  }
  else {
    while (まだ n バイト転送していない) {
      while (カーネル内バッファに空きがない)
        CHECK_SIGNAL
      カーネル内バッファへ転送できるだけ転送する
    }
    return n;
  }
}
```