Ruby VM におけるスタック領域の拡張の提案と実装

青山学院大学理工学部
情報テクノロジー学科 Dürst 研究室

学籍番号：15813031

小池 翔

目 次

第 1 章　序論 **1**
1.1　研究背景と目的 1
1.2　本研究の対象 1
1.3　本論文の構成 2

第 2 章　プログラミング言語 Ruby とその実装 **3**
2.1　プログラミング言語 Ruby 3
2.1.1　Ruby のソースコンパイル 3
2.1.2　Ruby プログラムの実行 4
2.1.3　Ruby オブジェクトの構造体 8
2.1.4　メソッドとクラスの定義 10
2.1.5　メモリ確保 11
2.1.6　スタックオーバーフロー 12
2.1.7　スレッド 13
2.2　YARV 15
2.2.1　YARV 命令列の生成 15
2.2.2　YARV 命令列の実行 17
2.2.3　変数アクセス 20

第 3 章　Linux OS におけるメモリマッピング **24**
3.1　メモリマッピングの種類 24
3.2　マッピングの作成：`mmap` 25
3.3　マッピングの削除：`munmap` 26

第 4 章　スタック領域確保の拡張 **27**
4.1　既存の実装 27
4.1.1　スタック領域の確保 27
4.1.2　スタック領域の解放 27
4.2　提案手法の実装 28
4.2.1　スタック領域確保の実装 28
4.2.2　スタック領域の解放の拡張 29

第 5 章　スタックオーバーフローの対処の拡張　31
5.1 既存の実装 31
5.1.1 制御フレーム 31
5.1.2 スタックオーバーフロー 31
5.2 提案手法 33
5.3 スタックオーバーフローの回避を行う関数の実装 34
5.3.1 `ext_stack` 関数の呼出し 34
5.3.2 スタックオーバーフローの検出 35
5.3.3 `ext_stack` 関数の引数 35
5.3.4 変数定義 36
5.3.5 スタックオーバーフローの回避 37
5.3.6 スタックの移動 37
5.3.7 各ポインタの設定 38
5.3.8 制御フレームの初期化 40
5.3.9 評価 41

第 6 章　結論　44
6.1 まとめ 44
6.2 今後の展望 44

謝辞　45

参考文献　46

付録　48

第1章　序論

1.1　研究背景と目的

オブジェクト指向スクリプト言語 Ruby は表現力の高い文法を備えている．そのため Java や C++ などの他の言語と比べ，少量の記述でメタプログラミングやアルゴリズムに忠実なコーディングが実現できる．また様々な API を持つコアクラスライブラリを備えた動的プログラミング言語である [1].

Ruby はその柔軟性からドメイン固有言語 (DSL) , サーバサイドアプリケーションの記述言語，業務システムの構築のための汎用言語など様々な環境下で利用されることが増えている．

Ruby の VM である YARV では，各スレッドにつきスタックを一つ用意するが，スタック一つあたりのサイズを大きくするとメモリ資源が枯渇してしまう．しかし反対に小さくしすぎた場合に，ブロックの階層が深くなることが避けられないプログラムでは，スタック領域を使い切ってしまいスタックオーバーフローを起こしやすくなる．

高性能並列科学技術計算を実行するためにクラウドコンピューティングが注目されており，Web アプリケーションにおいては応対の改善，GVL が解除され効果を発揮する IO 処理や拡張ライブラリではマルチスレッドが有効である [2]．マルチスレッドでは，省メモリのために，スタック一つあたりのサイズは，小さい方が望ましいが，十分な再帰も処理できるようにするべきである．

よって，本研究では，スタックオーバーフローが起きた際に従来のエラー処理ではなく，それを回避する方法を提案する．元のスタック領域よりも大きいスタック領域を確保し，中身を移動することでより深い再帰を処理できるように拡張する．また，新しいスタック領域を従来の実装で確保していたヒープ領域ではなく，仮想アドレス空間から確保することで他スレッドのスタック領域との衝突を避け，マルチスレッドに対応させる．

1.2　本研究の対象

本研究では，仮想アドレス空間からメモリ領域を確保するために，Linux OS のシステムコールを使った．Linux OS は，開発者の多くが使用しており，サーバの

構築に長けているためサーバサイドアプリケーションの記述言語として頻繁に使用される Ruby の拡張をする OS の対象としては十分である．

また，JRuby や mruby など様々な処理系の中から本研究では，まつもとゆきひろが C 言語で開発した Matz Ruby Interpreter (MRI) を対象とした．MRI は，Ruby の公式の処理系であり最も広く使われている [3].

1.3 本論文の構成

本論文の構成は，第 2 章でプログラミング言語 Ruby の実装について説明し，第 3 章では Linux OS におけるメモリマッピングについて述べる．第 4 章で本研究におけるスタック領域の確保の実装について詳述し，第 5 章では，本研究で拡張したスタックオーバーフローへの対処を行う関数の説明と，実現できたものと実現できなかったものについて評価をする．第 6 章では，本研究のまとめを行い，今後の展望を述べる．

第2章　プログラミング言語 Ruby とその実装

2.1　プログラミング言語 Ruby

プログラミング言語 Ruby は，まつもとゆきひろによって開発されたオブジェクト指向スクリプト言語である．マルチスレッドや例外機能が簡単にプログラミングできることから，直感的な文法とその表現力の高さが特徴である．本節では，C 言語によって開発された MRI : Matz Ruby Interpreter (以降 MRI) について説明していく．

2.1.1　Ruby のソースコンパイル

2.1 節で記述したとおり，Ruby は C 言語によって実装されており，実行時の `ruby` コマンドは，バイナリを呼び出している．Ruby のインストールは，Windows OS では， http://rubyinstaller.org/downloads/ からインストーラーをダウンロードする方法がある．Mac OS では，Homebrew, rbenv など様々な方法があるが，今回はソースファイルが必要なため，バージョン管理システムである Subversion を用いて Ruby の公式サイトよりソースコードをチェックアウトし，コンパイルする．Subversion を用いたチェックアウトのコマンドは以下を用いる [4].

```
svn co http://svn.ruby-lang.org/repos/ruby/trunk ruby
```

Ruby をソースコードから自分のマシンにいれるには，まずチェックアウトしてきたディレクトリで `autoconf` コマンドで `configure` を作成し，`./configure` で実行する．その際にオプションを指定することで開発が容易になることがある．また，すべてのオプションは，`configure --help` にて見ることができる．その中で重要なものを説明する．

Ruby は，オプションを指定しない場合 `/usr/local` にインストールされるが，以下のようにオプションを用いて `--prefix` に任意のインストール先を指定することができる [5].

```
./configure --prefix=DIR
```

また以下のように環境変数 CFLAGS に -g を設定することで gcc をデバッグオプションにすることができる．

```
./configure CFLAGS=-g
```

RDOC によるドキュメントの作成を無視するオプションの設定は，以下のとおりである．

```
./configure– disable-install-doc
```

次に make コマンドで Ruby のソースコードのコンパイルを行う．make コマンドでは，まず Ruby の主要部分を集めたスタティックライブラリ libruby-static.a を作成する．次に Ruby をコンパイルするために必要な miniruby を作成する．これは，拡張ライブラリをコンパイルするためには Ruby が必要となるが Ruby は拡張ライブラリをコンパイルしてからでないと作成できないという矛盾を解消するためである．miniruby を用いて拡張ライブラリも含めた ruby を作成する．また，BASERUBY を指定することで miniruby ではなく任意の Ruby を使用することも可能である．以上が，make コマンドの役割である [6].

最後に，管理者権限で make install コマンドを用いて Ruby のインストールを終了する．

2.1.2 Ruby プログラムの実行

Ruby プログラムは，実行されるまでに 3 工程を経ている．まずコードを字句解析することで，Ruby 言語内で使われる単語の列であるトークン列へと変換する．次に構文解析で，トークン列を AST ノードという Ruby の構文として意味のあるグループに分ける．そして，最終的に Ruby は仮想マシンである YARV で実行することができる命令列に変換する．以上の内容を図 2.1 にまとめた．各プロセスについて説明する．

字句解析

Ruby の C コードには，文字を一文字ずつ読み込み，その内容に従って処理するループ処理が含まれている [7]．トークンには，変数やメソッド，クラスなどの名前を示す識別子トークンと予約語トークンの二つがある．

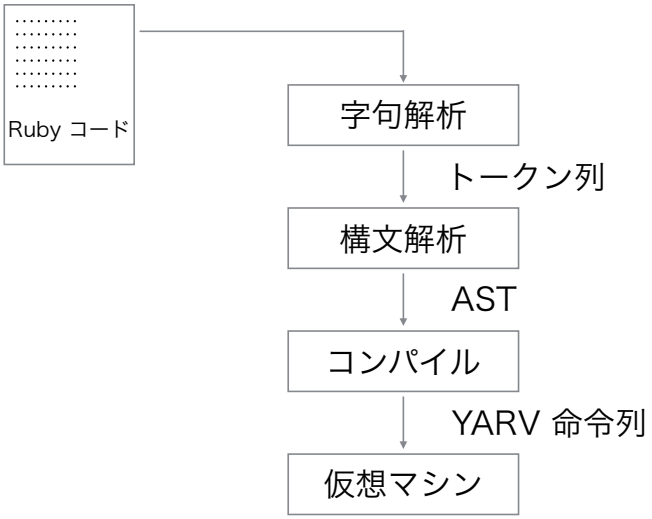

図 2.1: Ruby プログラムの実行フロー

構文解析

Ruby は Bison というパーサジェネレータを使用して構文解析を行う．Bison には GLR 法や IELR 法が備わっている [8]．Ruby は LALR (Look Ahead, Left to right, Rightmost derivation) 法でトークン列を左から右に処理していき，必要に応じて「先読み」することで AST ノードを作成する．また `ruby` コマンドに `-dump parsetree` オプションをつけることで AST ノードに関するデバッグ情報を出力することができる．

コンパイル

1.9.0 以前では，`eval` ルーチンが直接読み込んでいた AST ノードを YARV では，コンパイルすることでバイトコードに変換する．このバイトコードを YARV 命令列という．これは 2.2.1 節で詳述する．

エントリーポイント

次に Ruby がどのように動くかを C プログラムレベルで説明する．ソースコード 2.1 は，main.c の環境依存のプリプロセッサ命令部分を取り除いたものである．

```
1  int
2  main(int argc, char **argv)
3  {
4    ruby_sysinit(&argc, &argv);
5    {
6      RUBY_INIT_STACK;
7      ruby_init();
8      return ruby_run_node(ruby_options(argc, argv));
9    }
10 }
```

ソースコード 2.1: main.c

4 行目の ruby_sys_init 関数では，コマンドライン引数と標準入出力を初期化する ruby コマンドを初期化するための関数である [9]．6 行目の RUBY_INIT_STACK マクロは，ガベージコレクション (以降：GC) の際に必要となるスタックの開始位置を把握し，7 行目の ruby_init 関数で処理系や組み込みモジュールの初期化を行っている．ruby_init 関数は ruby_setup 関数を呼び出すが，ruby_setup 関数もまたいくつかの関数を呼び出す．その中で重要な関数について説明していく．

```
1  void
2  Init_BareVM(void)
3  {
4    rb_vm_t * vm = ruby_mimmalloc(sizeof(*vm));
5    rb_thread_t * th = ruby_mimmalloc(sizeof(*th));
6    if (!vm || !th) {
7      fprintf(stderr, "[FATAL] failed to allocate memory\n");
8      exit(EXIT_FAILURE);
9    }
10   MEMZERO(th, rb_thread_t, 1);
11   rb_thread_set_current_raw(th);
12
13   vm_init2(vm);
14   vm->objspace = rb_objspace_alloc();
15   ruby_current_vm = vm;
16
17   Init_native_thread();
18   th->vm = vm;
19   th_init(th, 0);
20   ruby_thread_init_stack(th);
21 }
```

ソースコード 2.2: Init_BareVM 関数

ソースコード 2.2 は，Init_BareVM 関数の定義である．この関数は仮想マシンを表す rb_vm_t 構造体と，スレッドを表す rb_thread_t 構造体を一個ずつ作成

する．vm は，13 行目の vm_init2 関数で，th は 10 行目の MEMZERO マクロによってメモリ領域をゼロクリアする．

19 行目の th_init 関数の定義から重要な部分のみを抜粋したものがソースコード 2.3 である．altstack は，シグナルハンドラ用の代替スタックである．SEGV のシグナルを受け取った際，本スタックではスタック領域に空きがないためエラー処理が不可能である．よって altstack をあらかじめ別に用意しておくことでシグナルの処理を代替することが可能となっている．

12 行目の th->vm->default_params.thread_vm_stack_size は，ソースコード 2.4 で定数として設定されており，決定したスタックのサイズを th->stack_size に格納しておく．また VALUE は Ruby 固有の構造体であり 2.1.3 節で説明する．

13 行目と 15 行目では，スタック領域の下端と上端を格納している．th->stack に thread_recycle_stack 関数を用いて，スタックの下端のアドレスを格納させる．スタックの上端は，制御フレームを指すポインタである cfp に格納する．

ソースコード 2.2 の 20 行目にある ruby_thread_init 関数は，スレッド生成時に呼ばれてスタックの大きさやスタックのアドレス情報などを取得しスレッドの情報として記憶する関数である．

ソースコード 2.1 の 8 行目は，評価器の実行で引数に用いられている．引数に用いられている ruby_options 関数はコマンドライン引数の解釈となっている．

```
1  static void
2  th_init(rb_thread_t *th, VALUE self)
3  {
4    th->self = self;
5
6    // allocate thread stack
7  #ifdef USE_SIGALTSTACK
8    th->altstack = malloc(rb_sigaltstack_size());
9  #endif
10
11   th->stack_size = th->vm->default_params.thread_vm_stack_size
12           / sizeof(VALUE);
13   th->stack = thread_recycle_stack(th->stack_size);
14
15   th->cfp = (void *)(th->stack + th->stack_size);
16
17   // 最初の制御フレームをプッシュ
18   // th のメンバの初期化
19 }
```

ソースコード 2.3: th_init 関数

```
#define RUBY_VM_THREAD_VM_STACK_SIZE (16*1024*sizeof(VALUE))
```

ソースコード 2.4: スタックサイズの指定

2.1.3 Ruby オブジェクトの構造体

Ruby では RVALUE という固定長の構造体を用いてメモリ管理を容易にすると共にメモリの断片化を防いでいる [10]．RVALUE は複数の異なる構造体をメンバとして持っており，それらはオブジェクトのクラスごとに使い分けられている．ソースコード 2.5 は RVALUE 構造体の定義で，表 2.1 にオブジェクトのクラスと構造体の対応を記す [6]．

```
1  typedef struct RVALUE {
2    union {
3      struct {
4        VALUE flags;
5        struct RVALUE *next;
6      } free;
7      struct RBasic basic;
8      struct RObject object;
9      struct RClass klass;
10     struct RFloat flonum;
11     struct RString string;
12     struct RArray array;
13     struct RRegexp regexp;
14     struct RHash hash;
15     struct RData data;
16     struct RTypedData typeddata;
17     struct RStruct rstruct;
18     struct RBignum bignum;
19     struct RFile file;
20     struct RNode node;
21     struct RMatch match;
22     struct RRational rational;
23     struct RComplex complex;
24
25     // 省略
26   } as;
27 } RVALUE;
```

ソースコード 2.5: RVALUE 構造体

Ruby では，これらの構造体を用いてオブジェクトを表現し，扱うときはポインタを経由する．構造体はオブジェクトのクラスに対応して異なるが，ポインタのほ

表 2.1: 構造体とクラスの対応

構造体	クラス
RObject	以下に当てはまらないものすべて
RClass	クラスオブジェクト
RFloat	小数
RString	文字列
RArray	配列
RRegexp	正規表現
RHash	ハッシュ
RFile	IO，File，Socket など
RData	上記以外の C レベルで定義されたクラスすべて
RStruct	Ruby の構造体 Struct クラス
RBignum	大きな整数

うは常に VALUE 型を用いる．VALUE は typedef unsigned long VALUE と定義されているためポインタとして使う際には，キャストすることが必要となる．キャストする際は RSTRING や RARRAY などのマクロ関数を用いる．またソースコード 2.6 のように irb を用いてクラスのインスタンスを生成した際に表示される 16 進数の値が，オブジェクトへの VALUE ポインタである．図 2.2 には，例として文字列を作成した際の RString, RClass, VALUE の関係を示す．以上が Ruby と C のデータを一元化する API である．

```
$ irb
> num = Numeric.new
=> #<Numeric:0x007fd4e18d7788>
```

ソースコード 2.6: インスタンスの作成

続いて VM に Ruby プログラムのコールスタックを把握させるために必要な構造体について説明する．コールスタックには，どのメソッドが他のメソッド，関数，ブロックを呼び出したかの情報が格納される．

```
1 typedef struct rb_control_frame_struct {
2   const VALUE *pc;
3   VALUE *sp;
4   const rb_iseq_t *iseq;
5   VALUE self;
6   const VALUE *ep;
7   const void *block_code;
8 } rb_control_frame_t;
```

ソースコード 2.7: rb_control_frame_t 構造体

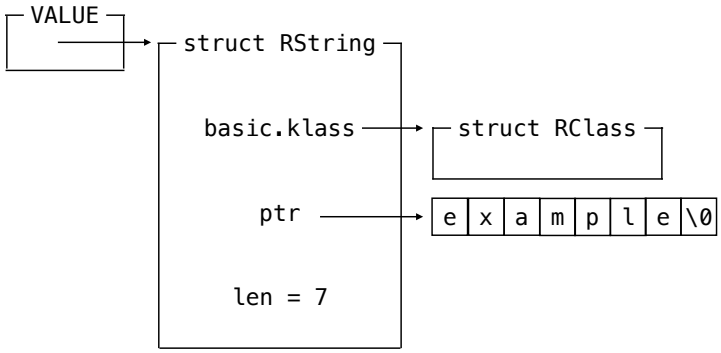

図 2.2: VALUE と 構造体とクラスの関係

rb_control_frame_t 構造体そのものは現在の制御フレームを指すポインタであり，ソースコード 2.3 の 15 行目で rb_control_frame_t 構造体変数 cfp は，初期化されている．rb_control_frame_t 構造体の重要なメンバについて表 2.2 にまとめる．

表 2.2: データ構造

メンバ	内容
pc (プログラムカウンタ)	実行すべき命令の場所を示す
sp (スタックポインタ)	内部スタックの一番上を示す
ep (環境ポインタ)	ローカル変数が内部スタックのどこにあるのかを示す

積みあげられることで次第に高くなっていくのが通常のスタックイメージだが，制御フレームがプッシュされると実際にはアドレスは高位アドレスから低位アドレスに向かって伸びていく．したがって，コールスタックの開始位置は，スタック領域の上端である．sp は，内部スタックを指すためスタック領域の下端から上位アドレスに向かって上がっていく．sp の初期値と，制御フレームの位置を示す cfp の関係を図示すると図 2.3 のようになる．内部スタックとコールスタックについては，2.2 節で説明する．

2.1.4 メソッドとクラスの定義

クラスメソッドを定義した際に C レベルでは rb_define_singleton_method 関数で処理をしている．この関数は obj に特異メソッド name を定義し，メソッド

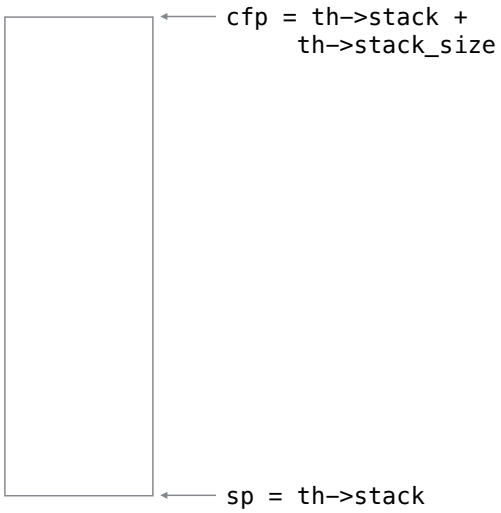

図 2.3: cfp と sp の関係

の実体を func に関数ポインタで与える [11]．rb_define_singleton_method 関数の定義をソースコード 2.8 に示した．

```
void rb_define_singleton_method(VALUE obj, const char *name,
    VALUE (*func)(), int argc)
```

ソースコード 2.8: rb_define_singleton_method 関数

例としてソースコード 2.9 のように Ruby レベルで Thread#new を呼び出したときは，ソースコード 2.10 のように解釈される．これによって Thread#new メソッドを呼び出すと，thread_s_new 関数が呼び出されることを確認できる．

```
thr = Thread.new{sleep 1}
```

ソースコード 2.9: クラスメソッド

```
rb_define_singleton_method(rb_cThread, "new", thread_s_new, -1);
```

ソースコード 2.10: rb_define_singleton_method 関数の例

2.1.5 メモリ確保

Ruby の実装におけるメモリ領域の確保をする関数について説明する．

ALLOCA_N マクロ

ALLOCA_N マクロは，任意の型のメモリを任意の数スタックフレームに割り当てる [12]．このメモリは関数が終わると自動的に解放される．通常 malloc 関数でメモリを確保すると必要なくなった時点で free 関数を呼び出しメモリの解放をする必要がある．しかし，ALLOCA_N マクロで呼び出す alloca 関数は，メモリ領域を確保するという機能までは一緒だが alloca 関数を呼び出した関数が終了すると同時に自動でメモリ領域を解放する点が異なる．

ALLOC マクロ

ALLOC マクロは，任意の型のメモリを割り当てる．メモリ割り当てに失敗した場合 GC を行いそれでもメモリ割り当てに失敗する場合は NoMemory Error を出力する．つまりこの関数が返り値を返したときは常にメモリの割り当てには成功している [13]．この関数を用いて変数などの領域を確保する．

ALLOC_N マクロ

ALLOC_N マクロは，任意の型のメモリを割り当てる．しかし，ALLOC マクロと割り当てるメモリの個数を指定できることが異なる．

2.1.6 スタックオーバーフロー

関数の中から自分自身を呼び出す再帰呼出しを用いることで簡潔な表現が可能になる．これはプログラムの可読性が大いに向上できるため，フィボナッチ数列やハノイの塔，階乗の計算，クイックソートなどで頻繁に用いられる．しかし，再帰呼出しは入れ子の深さを意味する再帰数が大きくなると用意されたスタック領域を使い切ってしまいスタックオーバーフローを起こす可能性がある．ソースコード 2.11 は，スタックオーバーフローを起こすプログラムの例で，スタックオーバーフローを起こした際の表示をその下に表す．

```
1  def foo2(n)
2    print "#{n} "
3    1.times do
4      if n <=12
5        foo1(n)
6      end
7    end
8  end
9
10 def foo1(n)
```

```
11    1.times do
12      n += 1
13      foo2(n)
14    end
15  end
16
17  foo1(0)
```

ソースコード 2.11: スタックオーバーフローを起こす Ruby プログラム

```
1 2 3 4 5 6 7 8 9 10 11 12 for_thesis_2.rb:13:in ‘times’:
stack level too deep (SystemStackError)
from for_thesis_2.rb:13:in ‘foo1’
from for_thesis_2.rb:5:in ‘block in foo2’
from for_thesis_2.rb:3:in ‘times’
from for_thesis_2.rb:3:in ‘foo2’
from for_thesis_2.rb:15:in ‘block in foo1’
from for_thesis_2.rb:13:in ‘times’
from for_thesis_2.rb:13:in ‘foo1’
from for_thesis_2.rb:5:in ‘block in foo2’
 ... 62 levels...
from for_thesis_2.rb:15:in ‘block in foo1’
from for_thesis_2.rb:13:in ‘times’
from for_thesis_2.rb:13:in ‘foo1’
from for_thesis_2.rb:19:in ‘<main>’
```

2.1.7 スレッド

スレッドは，メモリ空間を共有して実行される制御の流れのことである [14]. Ruby は，一つのプログラム内で複数の処理を同時に実行するマルチスレッド処理を実現することが可能である.

スレッドの実装

Ruby では，YARV にネイティブスレッド処理機構を用いて実装することで並列実行が可能な Ruby スレッドを実現した [15]. しかし，複数の Ruby スレッドが同時にネイティブメソッドを実行することのないように排他制御を行う必要がある. そのため，VM 中の全 Ruby スレッドが共有する単一のロック (GVL : Giant VM Lock) を用いているので，同時に実行されるネイティブスレッドは一つである.

しかし，IO 関連では GVL を解放するためスレッドは同時に実行される．また，拡張ライブラリでは GVL を操作できるため複数スレッドを同時に実行するような拡張ライブラリを作成することが可能である．

メインスレッド

メインスレッドは，プログラムの開始と同時に生成されるスレッドのことでメインスレッドが終わるときは，他のスレッドを終了させ実行中のプログラムも終了する．

スレッドの作成

Ruby におけるスレッドは Thread クラスのインスタンスとして表される．ソースコード 2.12 のように Thread#new を実行することでスレッドの作成を行う．

このとき C レベルでどのような処理が行われるかは，2.1.4 節で述べたとおりネイティブメソッドを作成する rb_define_singleton_method 関数から確認することができ，ソースコード 2.13 に示す．

```
t = Thread.new { sleep(1) }
```

ソースコード 2.12: スレッドの作成

```
rb_define_singleton_method(rb_cThread, "new", thread_s_new, -1);
```

ソースコード 2.13: Thread.new 実行時に呼び出される関数

ソースコード 2.13 より Thread#new が実行されると，thread_s_new が呼び出されることがわかる．ソースコード 2.14 は thread_s_new 関数のエラー処理を取り除いたものであり 5 行目の rb_thread_alloc 関数が最終的に th_init 関数を呼ぶことでスタック領域を確保している．これについては，ソースコード 2.3 で説明済みであり，また 4.1.1 節で詳述する．

```
1  static VALUE
2  thread_s_new(int argc, VALUE *argv, VALUE klass)
3  {
4    rb_thread_t *th;
5    VALUE thread = rb_thread_alloc(klass);
6
7    rb_obj_call_init(thread, argc, argv);
8    GetThreadPtr(thread, th);
9    return thread;
10  }
```

ソースコード 2.14: thread_s_new 関数

2.2 YARV

YARV : Yet Another Ruby VM (以降 YARV) は笹田耕一が高速化を目指し開発を進めたスタックマシンモデルの MRI を拡張した公式の処理系である [16]．変数などを保持する内部スタックをスタック領域の下端から，関数の呼出しをたどるためのコールスタックをスタック領域の上端から伸ばすことで変数アクセス，ネストなどに対応している．コールスタックに積む制御フレームと内部スタックに積むスタックフレームを図 2.4 に示す．ここで，制御フレームの sp は，内部スタックの上端を指し，ep は，内部スタック内を指し変数アクセスを可能にする．これは，2.2.3 にて詳述する．

Ruby 1.9.0 以前では，木構造の抽象構文木（AST : Abstract Syntax Tree）を再帰的に辿る純粋なインタプリタであったため Ruby プログラムの実行時間が長くなるという問題点があった．しかし YARV は JIT : Just In Time 方式でソースコードをバイトコードへコンパイルした後に，そのバイトコードを仮想マシン上で実行する [17]．そのため抽象構文木をたどる必要がなく Ruby プログラムを従来より高速に実行することが可能となった．この節では，YARV 命令列，および Ruby を実行する上での YARV の働きについて説明する．

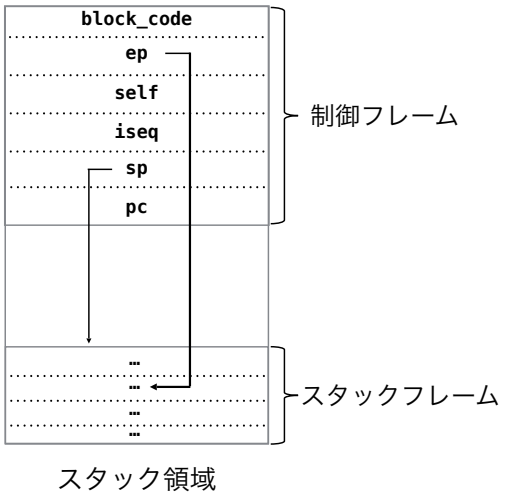

図 2.4: コールスタックと内部スタック

2.2.1 YARV 命令列の生成

まず第一に，ソースコード 2.15 を例に YARV が Ruby プログラムをどのように解釈するかを見ていく．ソースコード 2.16 は，このコードの YARV 命令列である．

```
puts 3+5
```

ソースコード 2.15: 単純な Ruby コード

```
1  == disasm: #<ISeq:<compiled>@<compiled>>==============
2  0000 trace 1 ( 1)
3  0002 putself
4  0003 putobject 3
5  0005 putobject 5
6  0007 opt_plus <callinfo!mid:+, argc:1, ARGS_SIMPLE>, <callcache>
7  0010 opt_send_without_block <callinfo!mid:puts, argc:1,
8       FCALL|ARGS_SIMPLE>, <callcache>
9  0013 leave
```

ソースコード 2.16: YARV 命令列

Ruby の関数およびメソッド呼出しは, ある一連のパターンに沿って YARV 用に関数呼出しをコンパイルする. まず, メソッドのレシーバを内部スタックにプッシュする. 10.times というメソッド呼出しの場合 10 が times メソッドのレシーバにあたる. その後, 引数を内部スタックにプッシュし, メソッドないし関数呼出しを行う.

しかし puts メソッドは, トップレベルのスコープから呼び出されている. そこで, 3 行目 putself という YARV 命令列で関数呼出しを self ポインタの現在値を puts メソッドのレシーバとして使う. self には Ruby 起動時に自動生成される Object クラスのインスタンスである top self オブジェクトのポインタが設定される. コンパイル時 3+5 は 3.+(5) と解釈されるため 4 行目でレシーバ 3 を内部スタックにプッシュし, 5 行目で引数 5 を内部スタックにプッシュしている.

メソッド呼出しを行う命令列は, 汎用性のある send と特化命令とに分けられる. 特化命令は, 高速化を目的に作られ特定のセレクタ (メソッド名), 特定の引数の数の場合, コンパイル時に通常のメソッド呼出しから特化命令に置き換えられる [17]. opt_plus の他には opt_size, opt_le, opt_aref などが特化命令である.

7 行目の opt_send_without_block は puts メソッドを呼び出し, 先ほどの opt_plus の結果を引数とする. 以上が, ソースコード 2.15 を YARV 命令列に変換した結果である.

次にソースコード 2.17 を例に, ブロックを使用する Ruby プログラムの YARV 命令列について説明する. ソースコード 2.17 に対する YARV 命令列をソースコード 2.18 に示す.

ソースコード 2.18 7 行目の send は,times メソッドを呼び出しており block in <compiled> でメソッド呼出しがブロック引数を含んでいることを意味する. ブロック引数内の YARV 命令列は, 9 行目から 24 行目の新たに生成されるローカルテーブルに保存される.

```ruby
3.times do |n|
  puts n
end
```

ソースコード 2.17: ブロックを含む Ruby コード

```
== disasm: #<ISeq:<compiled>@<compiled>>====================
== catch table
|-------------------------------------------------------------
0000 trace 1 ( 1)
0002 putobject 3
0004 send <callinfo!mid:times, argc:0>, <callcache>,
   block in <compiled>
0008 leave
== disasm: #<ISeq:block in <compiled>@<compiled>>===========
| catch type: redo st: 0002 ed: 0010 sp: 0000 cont: 0002
| catch type: next st: 0002 ed: 0010 sp: 0000 cont: 0010
|-------------------------------------------------------------
local table (size: 2, argc: 1 [opts: 0, rest: -1, post: 0, block: -1,
           kw: -1@-1, kwrest: -1])
[ 2] n<Arg>
0000 trace 256 ( 1)
0002 trace 1 ( 2)
0004 putself
0005 getlocal_OP__WC__0 2
0007 opt_send_without_block <callinfo!mid:puts, argc:1,
   FCALL|ARGS_SIMPLE>, <callcache>
0010 trace 512 ( 3)
0012 leave ( 2)
```

ソースコード 2.18: YARV 命令列

2.2.2 YARV 命令列の実行

2.2 節で述べたように YARV は引数や計算結果を保持する内部スタックと，どのメソッドから呼ばれたかを記憶するコールスタックの二つでできている．ここでソースコード 2.7 で説明した rb_control_frame_t 構造体のいくつかのメンバと共にソースコード 2.15 を例に説明する．

図 2.5 は，プログラムの実行を始める内部スタックと YARV 命令列の初期状態を表したものである．sp は，内部スタックの一番上を示し pc は，次に実行すべき命令の場所を示している．

trace は，プログラムの実行が始まる合図のようなもので内部スタックなどを変更することはなく pc を次の命令に進める．命令列 putself を実行することで self を内部スタックにプッシュし，続く二つの命令列 putobject でレシーバの 3 と引数の 5 を内部スタックにプッシュする．このとき pc は次の命令列である opt_plus を指す．ここまでの様子を図 2.6 に示す．

次に pc は，指している命令列である opt_plus を実行する．sp にもっとも近いものからレシーバ，引数として一つずつ解釈し命令列 opt_plus を実行する．

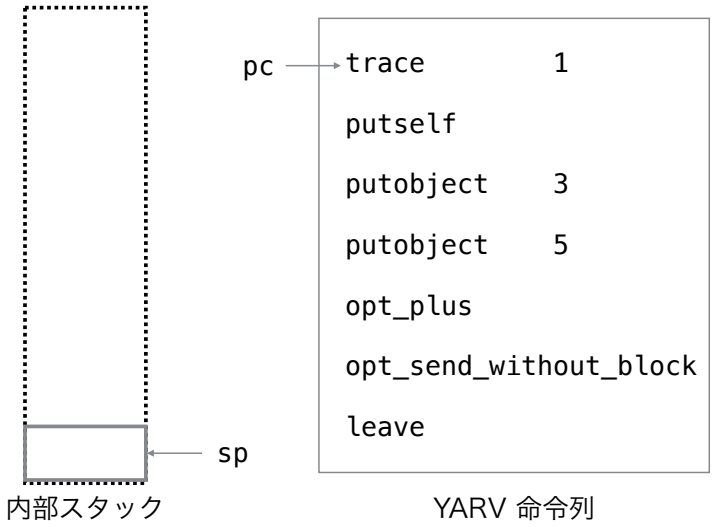

図 2.5: sp と pc

またその結果である 8 を内部スタックにプッシュし pc は，次の命令列を指す．ここでの様子を図 2.7 に示す．

puts メソッドは，コンソールに 8 を表示したのち内部スタックには nil をプッシュして pc を次にすすめる．最後に命令列 leave を実行すると同時にプログラムを終了する．

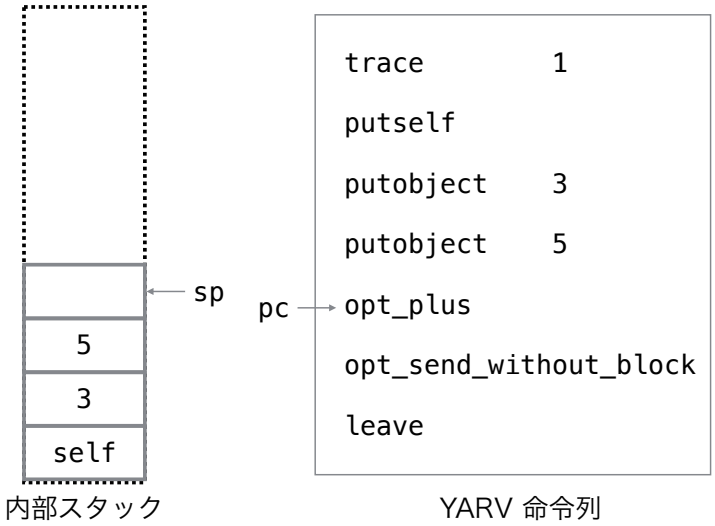

図 2.6: `putobject` 実行後の内部スタックの状態

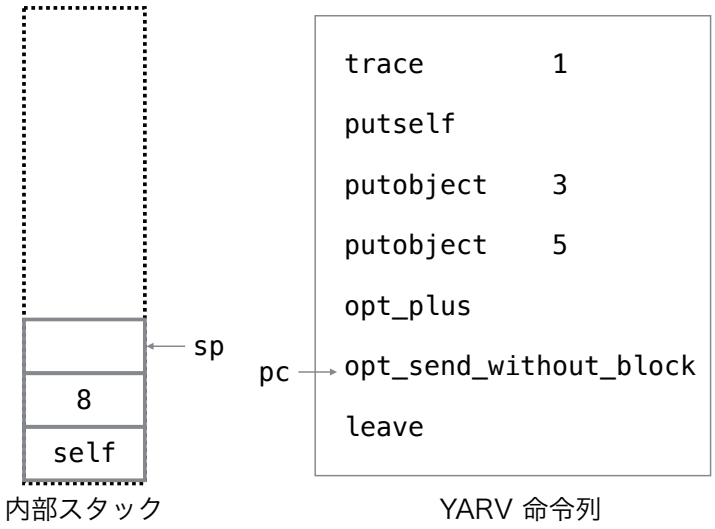

図 2.7: opt_plus 実行後のスタック状態

2.2.3 変数アクセス

Ruby は，変数に保存した値を内部スタックに格納する．変数への代入や値を読み出すために，ローカル変数アクセス，動的変数アクセスという二つの方法と rb_control_frame_t 構造体のメンバである環境ポインタ（EP）を使用する．

ローカル変数アクセス

Ruby プログラムでメソッド呼出しを行うと YARV は，その呼び出されたメソッド内で宣言されたローカル変数用のスペースを内部スタック上に用意する．以下のソースコード 2.19 のようにメソッド内で変数を定義した場合に内部スタックは図 2.8 のようになる．

```
1 def example
2   str = "hoge"
3   puts str
4 end
```

ソースコード 2.19: ローカル変数アクセスを行うプログラム

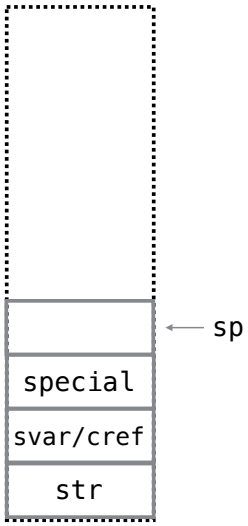

図 2.8: ブロックを解釈するときの内部スタックの初期状態

svar/cref には，特殊変数のテーブルへのポインタを special には，ブロックに関連する情報を記憶する．Ruby は，現在の sp の指す位置に文字列 "hoge" をプッシュするが，その際に環境ポインタ（以降 ep）を sp - 1 に設定する．つまり ep は，現在のメソッド用のローカル変数が内部スタック上のどこにあるかを指していることになる．その後 sp が移動しても ep は動かさない．変数に値を

代入する場合は命令列 setlocal_OP_WC_0 arg を用い，今回のプログラムでは，setlocal_OP_WC_0 2 となる．ここまでの状態を図 2.9 に示す．

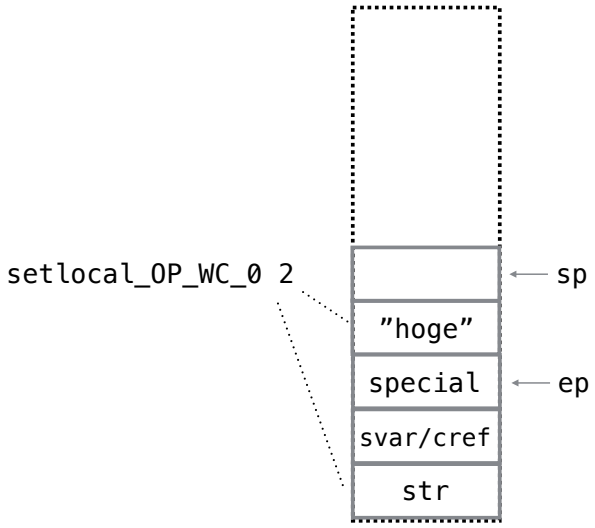

図 2.9: ep 設定後の内部スタックの状態

また puts メソッドを実行するには，変数を取得しなければならない．変数の取得には getlocal_OP_WC_0 arg 命令を使用する．今回のプログラムでは getlocal_OP_WC_0 2 となる．引数 2 は，変数 str に対する数値インデックスであり，str は ep - 2 の位置に存在することを表している．この様子を図 2.10 に示す．

動的変数アクセス

ソースコード 2.20 のように，異なるスコープで定義された変数を読み出そうとするとき，Ruby は動的変数アクセスを行う．2 行目までの内部スタックの状態を図 2.11 に示す．次に 4 行目の puts メソッドの直前までの内部スタックの状態を図 2.12 に示す．3 は Integer#times のレシーバであり，その上の svar/cref，special は C コードが使用するために必要となる．さらにブロックを解釈するために再度 svar/cref， special を積んでいる．

puts メソッドを実行するときに，変数 str の値を取得する必要がある．しかし，別スコープで定義されているためローカル変数アクセスのように ep を辿ることは不可能である．そこで getlocal_OP_WC_1 arg 命令を使用する．今回 arg には 2 が入る．これは先ほどと同様に数値インデックスとなっている．1 は ep を一つたどることを意味している．つまり previous ep から数値インデックス 2 の部分に str が存在していることを示している．

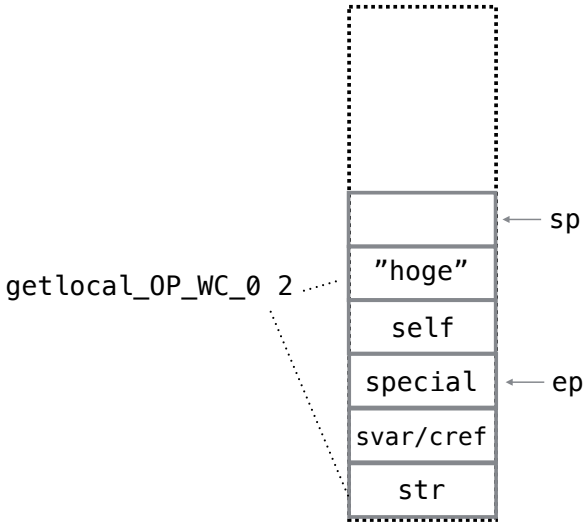

図 2.10: 変数の値を読み出す際の内部スタックの状態

```
1  def example
2    str = "hoge"
3    3.times do
4      puts str
5    end
6  end
```

ソースコード 2.20: 動的変数アクセスを行うプログラム

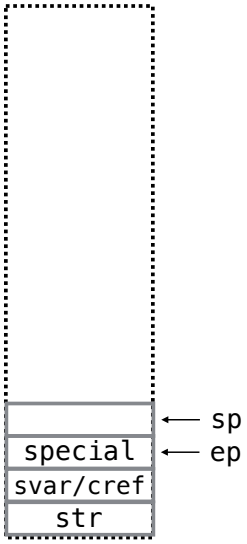

図 2.11: 変数を定義するときの内部スタックの状態

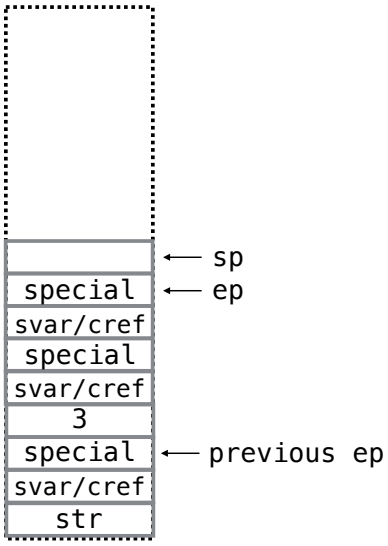

図 2.12: puts メソッドを実行する直前の内部スタックの状態

第3章　Linux OS におけるメモリマッピング

本研究では，スタックオーバーフローが起き，より大きいスタック領域を確保する際に，ヒープ領域で確保していた他のスタック領域との衝突を避ける目的で，仮想アドレス空間からメモリ領域を取得する必要がある．これは，スタック移動後に古いスタック領域へのアクセスを禁止することやスタックの中身を明らかにすることが可能になり開発の効率を上げることにも繋がる．

これらの実現のために，システムコールを用いる必要がある．Ruby は，Ruby on Rails の普及から Web サービスに利用されることが多く，サーバサイドアプリケーションの記述言語として使用されることが非常に多い．よって，本研究ではサーバの構築に頻繁に使用される Linux OS 上で開発を進めた．この章では，メモリマッピングの基礎知識および　Linux OS におけるシステムコールの説明を行っていく．

3.1　メモリマッピングの種類

メモリマッピングには，マッピングの種類が 2 種類，変更の可視性が 2 種類ある．それらを組み合わせることで必要な機能をもたせたメモリマッピングが可能となる．

ファイルマッピング

ファイル内容を，自プロセスの仮想メモリへ直接マッピングし，メモリへのアクセスはそのままファイル内容へのアクセスとなる．また，マッピング内のメモリページは必要に応じてファイルからロードされる．

無名マッピング

利用可能なメモリ領域を仮想アドレス空間から確保する．対応するファイルを持たず，マッピングしたページ内容は 0 で初期化される．アプリケーション実行

中に OS から追加のメモリリソースを確保する方法として主に用いられており，UNIX 系 C ライブラリの malloc 関数の実装に用いられている．

非共有マッピング

マッピング内のデータが変更しても，他プロセスからはその変更を見ることができない．初期化後の変更は，プロセス内に閉じる仕組みとなっている．

共有マッピング

マッピング内のデータが変更されると，そのマッピングを共有する他プロセスからもその変更を見ることが可能である．

以上の内容を以下の表 3.1 に示す [18]．また，本研究では非共有無名マッピングを用いた．

表 3.1: メモリマッピングの種類

	ファイル	無名
非共有	ファイル内容で初期化	メモリ割り当て
共有	メモリマップ I/O, プロセス間で共有	プロセス間で共有

3.2 マッピングの作成：mmap

mmap システムコールは，自プロセスの仮想アドレス空間にマッピングを作成する．ソースコード 3.1 は，mmap システムコールのプロトタイプである．

返り値は，成功時に新規マッピングの先頭アドレス，エラー時に MAP_FAILED となる．addr には，マッピングを配置するアドレスを指定する．NULL を指定したときは，カーネルが適切なアドレスに配置する．length には，マッピングサイズをバイト単位で指定し，prot には，マッピングに設定する保護フラグを指定する．表 3.2 の PROT_NONE かそれ以外の三つを OR 演算で結合したものを指定する．

flags には，マッピングを制御するオプションフラグを指定する．MAP_PRIVATE か MAP_SHARED のどちらかを必ずセットする必要がある．MAP_PRIVATE の場合は非共有マッピングを作成し，MAP_SHARED の場合は共有マッピングを作成する．

fd と offset については，ファイルマッピングのときのみ必要となり，本研究では必要ではないため説明を省略する．

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,
    int fd, off_t offset);
```

ソースコード 3.1: mmap システムコール

表 3.2: メモリマッピングの種類

フラグ	説明
PROT_NONE	アクセス禁止
PROT_READ	読み取り可能
PROT_WRITE	書き込み可能
PROT_EXEC	実行可能

3.3 マッピングの削除：munmap

munmap システムコールは，mmap システムコールで確保していたマッピングを自プロセスの仮想アドレス空間から削除する．ソースコード 3.2 は munmap システムコールのプロトタイプである．

返り値は，成功時に 0 を，エラー時には -1 となる．addr には，削除するマッピングの先頭アドレスを指定し，length には削除するマッピングの大きさを指定する．

```
#include <sys/mman.h>
int munmap(void *addr, size_t length);
```

ソースコード 3.2: munmap システムコール

第4章　スタック領域確保の拡張

この章では，スタックを確保する関数および前章で説明したシステムコールを用いて作成したマッピング関数について説明する．

4.1　既存の実装

4.1.1　スタック領域の確保

スタック領域の確保は, ソースコード 2.3 の 13 行目の thread_recycle_stack 関数で行っている．以下のソースコード 4.1 に抜粋する．

ここで引数の th->stack_size は，確保するスタック領域の大きさが格納されている．これは vm_core.h で決定されており任意のサイズにチューニングが可能である．また，thread_recycle_stack 関数は，ALLOC_N マクロを呼び出し malloc 関数で実際にスタック領域のメモリを確保する．ALLOC_N マクロの説明については，2.1.5 節にあるので割愛する．最終的に th->stack には，確保したスタック領域の先頭アドレスが入っている．

開発の効率化をはかる際は, thread_recycle_stack 関数の代わりに 4.2.1 節で述べる mapping 関数でスタック領域を確保することで，スタック領域の中身が明らかになる．本研究の最終的な実装では，スタック領域の衝突を避けるために，スタックオーバーフローの処理をしない場合は thread_recycle_stack 関数での領域確保のみとなる．

```
th->stack = thread_recycle_stack(th->stack_size);
```

ソースコード 4.1: スタック領域の確保

4.1.2　スタック領域の解放

従来の方法では，最終的にスタック領域を必ず malloc 関数を用いて確保していたため，free 関数で領域を返還する必要がある．

スタック領域解放の準備を rb_thread_recycle_stack_release 関数で行う．その中の ruby_xfree 関数で free 関数を用いて実際にスタック領域を解放している．ruby_xfree 関数の内容をソースコード 4.2 に示す．

```
1  void
2  ruby_xfree(void *x)
3  {
4    ruby_sized_xfree(x, 0);
5  }
6
7  void
8  ruby_sized_xfree(void *x, size_t size)
9  {
10   if (x) {
11     objspace_xfree(&rb_objspace, x, size);
12   }
13 }
14
15 static void
16 objspace_xfree(rb_objspace_t *objspace, void *ptr,
17     size_t old_size)
18 {
19   // 省略
20
21   free(ptr);
22
23   // 省略
24 }
```

ソースコード 4.2: スタック領域の解放

4.2 提案手法の実装

4.2.1 スタック領域確保の実装

本研究では，マルチスレッド時のスタック領域の拡張の際に他のスレッドのスタック領域との衝突を避ける目的で新しく確保するスタック領域は mmap システムコールを用いた．また，Ruby のソースコードは，マクロや関数を多く用いており，スタック領域を確保するための関数やシステムコールおよびエラー処理などを直接書くべきではないと判断した．そのため，本研究で実装したスタック領域を確保する関数の名前を mapping 関数とし，ソースコード 4.3 に定義を示す．

```
1  static void *
2  mapping(size_t size)
3  {
4    // declaration
5    void *mem;
6    int fd;
7    fd = open("/dev/zero",O_RDWR);
8
9    // error handler
10   if(fd == -1){
11     vm_stackoverflow();
12   }
13   else{
14     mem = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE,
15         fd, 0);
16   }
17
18   // error handle
19   if(mem == MAP_FAILED){
20     vm_stackoverflow();
21   }
22
23   close(fd);
24   return mem;
25 }
```

ソースコード 4.3: 新しいスタック領域確保のための mapping 関数

mapping 関数は，非共有無名マッピングを行う関数である．引数に確保したいサイズを取り，返り値は，確保した領域の先頭アドレスとなる．10 行目のエラー処理は fd の初期化が失敗した場合であり，18 行目のエラー処理は，領域確保に失敗した場合のエラー処理である．また，fd の初期化について /dev/zero ファイルは UNIX 系 OS の仮想デバイスファイルであり，内容がすべて 0 である．そのファイルを読み書き両用で開くことで fd を初期化する．

4.2.2 スタック領域の解放の拡張

従来，スタック領域の解放は，rb_thread_recycle_stack_release 関数が呼び出す ruby_xfree 関数内の free 関数で行っていた．しかし，本研究では前節で述べたとおりスタック領域の拡張分は mapping 関数で確保する．よって，解放する領域が必ずしも malloc 関数で確保されたものでないため mapping 関数で確保された領域も解放できるようにすべきである．

スタック領域の解放についてはそれほど大きくならないため関数を作ることな

く直接書き換えた．引数で得たスタックの開始アドレスと GET_THREAD マクロを用いて現在のスレッドを取得し，解放するスタック領域を把握する．mapping 関数で確保したスタック領域であれば munmap システムコールでマッピングの削除を行い，malloc 関数で確保した領域であった場合は従来とおり ruby_xfree 関数で解放する．

```
1  void
2  rb_thread_recycle_stack_release(VALUE *stack)
3  {
4    // 省略
5    th = GET_THREAD();
6    if(munmap(stack,th->stack_size) == -1){
7      if(stack != NULL){
8        free(stack);
9        stack = NULL;
10     }
11   }
12 }
```

ソースコード 4.4: メモリ領域解放 (提案手法)

第5章　スタックオーバーフローの対処の拡張

この章では，従来スタックオーバーフローがどのようなときに生じ，どのような処理をしていたのかを述べた後に，本研究で提案および実装したプログラムについて説明する．

5.1　既存の実装

5.1.1　制御フレーム

YARV は，スタック領域の上端からコールスタックとして制御フレームポインタ：cfp を下位アドレスにデクリメントしていく．また，スタック領域の下端からは内部スタックとしてスタックポインタ：sp を上位アドレスに向かってインクリメントしていく．制御フレーム一つは YARV 命令列で記述されたメソッド，つまり Ruby レベルで定義されたメソッドやブロックの呼出し一回に対応しており，return とともに一つずつ降ろされていく．いくつか進んだあとの状態を例として図 5.1 に示す．

また,図 5.2 は,新しい制御フレームをプッシュする役割を持つ vm_push_frame 関数の概要である．vm_push_frame 関数は，新しくプッシュした制御フレームの初期化作業の役割も担っている．

5.1.2　スタックオーバーフロー

Ruby には，def で定義されたメソッドと，2.1.4 節で述べた C 言語で実装されたネイティブメソッドがある．ネイティブメソッドの実体は C で記述された関数であるため関数呼出しが行われる．そしてその中に Ruby のメソッドの呼出しがあった場合は，vm_exec 関数がネストされて呼び出される．そのため Ruby で定義されたメソッドとネイティブメソッドが呼び合い続けると return で制御フレームをポップすることなくスタック領域に積み続けるため，スタック領域を使いきってしまいスタックオーバーフローを起こす．

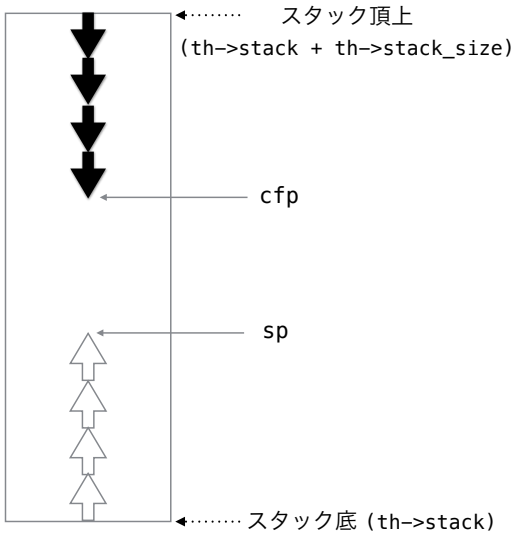

図 5.1: スタック内の sp と cfp の状態

本研究では，スタックオーバーフローの検出に CHECK_VM_STACK_OVERFLOW0 マクロを用いる．CHECK_VM_STACK_OVERFLOW0 マクロは vm_push_frame 関数で呼び出されており，ソースコード 5.1 のように定義されている．

ソースコード 5.1 の 3 行目から 6 行目でスタックオーバーフローの検出を行っている．再帰数が多くなり，スタック領域に空きがなくなったことで生じるスタックオーバーフローを検出しているのは，6 行目の条件である．cfp と sp が衝突しかけたときにスタックオーバーフローとなり vm_stackoverflow 関数を呼び出しエラー処理を行う．また，スタックオーバーフローが起きない場合は特別な処理をせずに終了する．

```
1  #define RUBY_CONST_ASSERT(expr) (1/!!(expr))
2  #define VM_STACK_OVERFLOWED_P(cfp, sp, margin) \
3    (!RUBY_CONST_ASSERT(sizeof(*(sp)) == sizeof(VALUE)) || \
4    !RUBY_CONST_ASSERT(sizeof(*(cfp)) ==
5      sizeof(rb_control_frame_t)) || \
6    ((rb_control_frame_t *)((sp) + (margin)) + 1) >= (cfp))
7  #define WHEN_VM_STACK_OVERFLOWED(cfp, sp, margin) \
8    if (LIKELY(!VM_STACK_OVERFLOWED_P(cfp, sp, margin)))
9      {(void)0;} else /* overflowed */
10 #define CHECK_VM_STACK_OVERFLOW0(cfp, sp, margin) \
11   WHEN_VM_STACK_OVERFLOWED(cfp, sp, margin) vm_stackoverflow()
12 #define CHECK_VM_STACK_OVERFLOW(cfp, margin) \
13   WHEN_VM_STACK_OVERFLOWED(cfp, (cfp)->sp, margin)
14     vm_stackoverflow()
```

ソースコード 5.1: スタックオーバーフローの検出

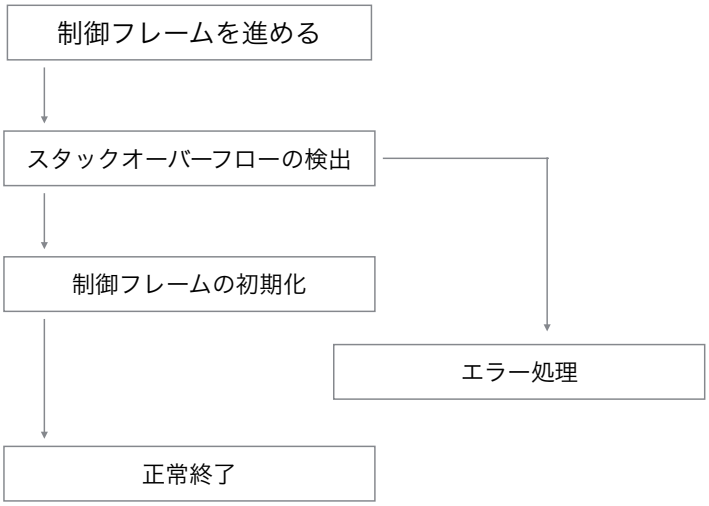

図 5.2: vm_push_frame 関数の概要

5.2 提案手法

本研究では，制御フレームのプッシュやスタックオーバーフローの検出，変数アクセスなど，スタックオーバーフローを処理する箇所以外はほとんど変えない．スタックオーバーフローを起こしたときのみ，従来のエラー処理に移る代わりに本研究で拡張したスタックオーバーフローの処理を行う．スタックオーバーフローの処理の拡張の目的は，より大きなスタック領域を確保し，内部スタックおよびコールスタックを移動することでスタックオーバーフローを回避することである．

本実装では，元のスタック領域の 2 倍の大きさを確保することとした．スタック領域の移動について，以下の図 5.3 に示す．図に見られるように，スタック領域には，スタック領域の上端から降りてくるコールスタックと，スタック領域の下端から上がってくる内部スタックの二つがある．

また，スタック領域の移動の際に設定をしなければならないポインタが三つある．最初の二つは，5.1.2 節で説明したとおり，スタックオーバーフローの検出に用いられる cfp と sp である．cfp は，コールスタックの下端を指しており，sp は，内部スタックの上端を指している．2.2.3 節で述べたとおり内部スタックには，変数の情報も格納されている．よって，変数アクセスのために ep も新しいスタック領域内で設定をしなければならない．

cfp は，n 番目と n - 1 番目の差が常に一定であるため，元々積まれている数を把握することで設定することが可能である．一方 sp は，ローカル変数の個数などに依存しているため必ずしも一定の距離移動するとは限らない．よって sp の移動は，元のスタック領域の開始アドレスと新しく確保したスタック領域の開始アドレスの距離をとることで相対的に設定することとした．

また ep について初期化段階では，sp - 1 に位置づけられるが，sp は，内部スタックの一番上を指すため可変であると共に，一定の大きさ進むわけではないため，ep にとって絶対的に位置を割り出す材料にはなり得ない．そのため ep の移動についても，sp と同様に相対的に設定することとした．

また，スタックオーバーフローの処理の拡張を行う関数を ext_stack 関数と名付けた．

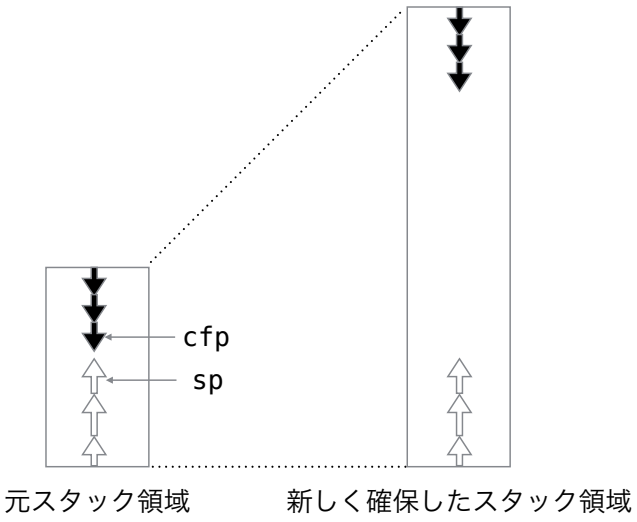

図 5.3: スタック領域の移動

5.3 スタックオーバーフローの回避を行う関数の実装

5.3.1 ext_stack 関数の呼出し

5.1.2 節で述べたとおり，従来は vm_push_frame 関数内でスタックオーバーフローの検出を CHECK_VM_STACK_OVERFLOW0 マクロを用いて行い，スタックオーバーフローが検出された場合のみ処理が行われた．しかし ext_stack 関数では，スタックオーバーフローを検出し，スタック領域を移動した際に，cfp，sp および ep が呼出し元と変わる．そのため，スタックオーバーフローが検出されない場合は，呼出し元と同じ cfp を，スタックオーバーフローが検出された場合は，移動先のスタック領域において設定した cfp を返り値とする．

よって，vm_push_frame 関数では，従来のようなマクロの呼出しではなく，ソースコード 5.2 の 3 行目のように呼び出す．2 行目の th->check については，スタック領域の移動を行った際のフラグであり，本研究で rb_thread_t 構造体のメンバに追加した．5 行目では，ext_stack 関数の返り値を現在のスレッドの

cfp に代入することで ext_stack 関数でスタック領域の移動が行われた場合も，行われない場合も cfp を正しく設定している．

```
1 rb_control_frame_t *cfp_new;
2 th->check = 0;
3 cfp_new = ext_stack(cfp,sp,local_size+stack_max,th,local_size,
4   iseq,type,self,specval,cref_or_me,pc);
5 th->cfp = cfp_new;
```

ソースコード 5.2: ext_stack 関数の呼出し

5.3.2 スタックオーバーフローの検出

ext_stack 関数では，従来の方法を用いてスタックオーバーフローの検出を一番最初に行う．ソースコード 5.3 の 1 行目から 3 行目について，前節で述べたとおりスタックオーバーフローを起こさない場合は，呼出し元と同じ cfp を返す．本研究のスタック領域の移動については，13 行目部分で実現する．

```
1  if(!VM_STACK_OVERFLOWED_P(cfp, sp, margin)){
2    return cfp;
3  }
4  else{
5    // check trigger for stackoverflow
6    if(!RUBY_CONST_ASSERT(sizeof(*(sp)) == sizeof(VALUE))){
7      vm_stackoverflow();
8    }else if(!RUBY_CONST_ASSERT(sizeof(*(cfp))
9      == sizeof(rb_control_frame_t))){
10     vm_stackoverflow();
11   }else if(((rb_control_frame_t *)((sp) + (margin)) + 1)
12     >= (cfp)){
13     // extend the stack
14   }
15 }
```

ソースコード 5.3: ext_stack 関数のスタックオーバーフローの検出

5.3.3 ext_stack 関数の引数

ソースコード 5.4 は，ext_stack 関数のプロトタイプである．引数とその役割を表 5.1 に示す．cfp ，sp および margin は，スタックオーバーフローの検出に用いられる．th は，現在のスレッドを表している．その他の引数は新しくプッシュする制御フレームの初期化や，内部スタックにプッシュする．これについては，5.3.8 節で述べる．

```
static rb_control_frame_t *
ext_stack(rb_control_frame_t *cfp, VALUE *sp, int margin,
    rb_thread_t *th, int local_size, const rb_iseq_t *iseq,
    VALUE type, VALUE self, VALUE specval, VALUE cref_or_me,
    const VALUE *pc)
```

ソースコード 5.4: ext_stack 関数のプロトタイプ

表 5.1: ext_stack 関数の引数

引数名	役割
cfp	コントロールフレームポインタ
sp	スタックポインタ
margin	スタックオーバーフローの検出用
th	現在のスレッド
local_size	ローカル変数総数 + 1
iseq	命令列
type	制御フレームの種類
self	レシーバー
specval	特殊変数
cref_or_me	Qnil，T_IMEMO 型
pc	プログラムカウンタ

5.3.4 変数定義

ext_stack 関数ではいくつかの変数を定義している．変数の一覧を表 5.2 に示す．new_size は，新しく確保したスタック領域の大きさであり，元のスタック領域の大きさを示す old_size の二倍の値が設定されている．cfp_base および sp_base は，元のスタック領域における cfp と sp であり，cfp_move と sp_move は，新しいスタック領域における cfp と sp である．5.2 で説明したとおり sp と ep の移動には，元のスタック領域と新しいスタック領域の開始アドレスのオフセットが必要となり，diff に記憶させる．for_finish , j および tmp_sp の三つの変数は sp の設定の際に用いる．for_finish はループの終了条件として，j はループの回数として，tmp_sp はループ内で一時的に記憶する用途で用いられる．count は，コールスタックの移動に用いられる．stack_for_mremap は，新しいスタック領域の開始アドレスを表す．

表 5.2: ext_stack 関数内の変数

変数名
new_size
old_size
cfp_base
cfp_move
sp_base
sp_move
diff
for_finish
j
tmp_sp
count
stack_for_mremap

5.3.5 スタックオーバーフローの回避

スタックオーバーフローを回避するために，元のスタック領域よりも大きなスタック領域を確保する必要がある．4.2.1 節の mapping 関数を用いてソースコード 5.5 の 1 行目のように stack_for_mremap に新しいスタック領域の開始アドレスを格納する．3 行目は，元のスタック領域の開始アドレスを格納しており，4 行目と 5 行目で元のスタック領域と新しいスタック領域それぞれについて上端を格納する．

```
1 stack_for_mremap = mapping(new_size);
2
3 sp_base = th->stack;
4 cfp_move = (void *)(stack_for_mremap + new_size);
5 cfp_base = (void *)(th->stack + old_size);
```

ソースコード 5.5: スタック領域の確保と変数の初期化

5.3.6 スタックの移動

元のスタック領域内の内部スタックとコールスタックは，その中身を変えることなく新しいスタック領域にコピーする必要がある．スタックのコピーは，スタック領域の下端からの内部スタックと，上端からのコールスタックの二つに分かれており，これらを個別に memcpy 関数を用いてコピーする．

コールスタックのコピーについて memcpy 関数の第三引数である大きさを決めるにあたって制御フレームいくつ分をコピーすれば良いのかを知る必要がある．

そこで cfp が一定数移動することを利用して任意の cfp 二つの間にいくつ制御フレームが積まれているのかを把握する機能を持つ count_cfp 関数を作成した．ソースコード 5.6 は count_cfp 関数の定義である．

```
1 static int count_cfp(rb_control_frame_t *base_cfp,
2      rb_control_frame_t *cfp){
3   int count = 0;
4   while(base_cfp != cfp){
5     base_cfp--;
6     count++;
7   }
8   return count;
9 }
```

ソースコード 5.6: count_cfp 関数

memcpy 関数の第一引数には，コピー先の開始アドレスを知る必要がある．ソースコード 5.7 のように制御フレームの数だけ cfp_move をデクリメントする．その後 memcpy 関数を用いてコールスタックのコピーを行い，cfp_move に再度新しいスタック領域の上端を格納させて終了する．

```
1 for(int i=0;i<count;i++){
2   cfp_move--;
3 }
4
5 memcpy(cfp_move,cfp,(sizeof(rb_control_frame_t *)*count));
6 cfp_move = (void *)(stack_for_mremap + new_size);
```

ソースコード 5.7: コールスタックのコピー

次に，内部スタックのコピーについて詳述する．内部スタックは，各ネストの変数の数などによって一つあたりの大きさが異なり，コールスタックのように数を知ることでコピーする大きさは定まらない．よって ext_stack 関数の引数 sp と元のスタック領域の下端との差をコピーする大きさとした．また，size_t 型にキャストした後，8 倍することで VALUE 型と大きさを揃える役割をもたせた．ソースコード 5.8 は，内部スタックのコピーの様子を表したものである．

```
memcpy(stack_for_mremap,th->stack,((size_t)(sp - th->stack)*8));
```

ソースコード 5.8: 内部スタックのコピー

5.3.7 各ポインタの設定

設定するポインタのセットの数は，制御フレームの数だけあるのでループを回す条件は，ソースコード 5.9 の 2 行目のようになる．元のスタック領域の上端

を指していた cfp_base を一つずつデクリメントしていき，引数 cfp の直前で終了する．残りの一回は，新しく積む制御フレームの初期化作業であるため別途処理をする．これは 5.3.8 節で詳述する．

rb_control_frame_t 構造体は，sp ，ep の他にもメンバを保持している．しかし，それらはスタック領域内にはなくスタックの移動によって再度設定をする必要がない．そのため，3 行目にあるとおり，一度 cfp_base のメンバをすべて cfp_move に代入する．制御フレーム一つに対し進む距離が一定である cfp は，この作業で設定が終了する．24 行目と 25 行目で，各 cfp_base と cfp_move をデクリメントし，ループを繰り返す．

sp の移動は 4 行目から 18 行目と，28 行目から 36 行目の二つに分かれている．これは，制御フレームの初期化を行うために 2 行目で条件を引数 cfp の直前までにしたのが理由であり，どちらも行っている作業は同じである．概要としては，一つ前の sp から今の sp までインクリメントしていき，その都度 sp の設定を行う．tmp_sp は 1 行目にあるとおり，元のスタック領域の下端を指しており 2 行目から始まるループ内でインクリメントされていく．また for_finish は，4 行目にあるとおり，今の sp を記憶させることで 9 行目から始まるループの終了条件とする．9 行目から始まるループでは 10 行目で cfp_base->sp から，元のスタック領域と新しいスタック領域のオフセットを引くことで新しいスタック領域内において cfp_move->sp を設定している．ループ終了後，繰り返した数 j を足すことで cfp_base->sp を元の値に戻して sp の設定を終了する．また 6 行目，7 行目にあるとおり cfp_base->sp が NULL だった場合は，新しいスタック領域での設定が必要ないため，cfp_move->sp には NULL を代入する．

ep の設定は，19 行目から 23 行目で行っている．sp と同様に，元のスタック領域と新しいスタック領域のオフセットを用いて設定を行う．ep についても，cfp_base->ep が，NULL の場合，新しいスタック領域において設定する必要がないため，cfp_move->ep には NULL を格納する．

```
1  tmp_sp = th->stack;
2  while (cfp_base > cfp) {
3    *cfp_move = *cfp_base;
4    for_finish = cfp_base->sp;
5    j=0;
6    if (cfp_base->sp == NULL) {
7      cfp_move->sp = NULL;
8    }
9    else {
10     while (tmp_sp <= for_finish) {
11       cfp_move->sp = cfp_base->sp - diff;
12       cfp_base->sp--;
13       cfp_move->sp--;
14       tmp_sp++;
```

```
15        j++;
16      }
17      cfp_base->sp += j;
18      cfp_move->sp += j;
19    }
20    if (cfp_base->ep == NULL) {
21      cfp_move->ep = NULL;
22    }
23    else {
24      cfp_move->ep = cfp_base->ep - diff;
25    }
26    cfp_move--;
27    cfp_base--;
28  }
29
30  j = 0;
31  while (tmp_sp <= arg_sp) {
32    cfp_move->sp = sp - diff;
33    sp--;
34    cfp_move->sp--;
35    tmp_sp++;
36    j++;
37  }
38  cfp_move->sp += j;
```

ソースコード 5.9: 各ポインタの設定

5.3.8 制御フレームの初期化

新しくプッシュする制御フレームの初期化をソースコード 5.10 に示す．1 行目は，引数 sp から元のスタック領域と新しいスタック領域のオフセットを引くことで制御フレームの初期化に必要な引数をチューニングしている．

3 行目から 6 行目は，新しくプッシュする制御フレームの初期化を行っている．8 行目から始まる for ループでは，ローカル変数の数だけ内部スタックに領域を用意し，12 行目から 14 行目では引数を用いて内部スタックを進める．2.2.3 節で述べたとおり，ep を sp - 1 に設定する作業を 16 行目と 17 行目で行っている．

```
1   sp -= diff;
2
3   cfp_move->pc = (VALUE *)pc;
4   cfp_move->iseq = (rb_iseq_t *)iseq;
5   cfp_move->self = self;
6   cfp_move->block_code = NULL;
```

```
7
8   for (int i=0; i < local_size; i++) {
9     *sp++ = Qnil;
10  }
11
12  *sp++ = cref_or_me;
13  *sp++ = specval;
14  *sp = type;
15
16  cfp_move->ep = sp;
17  cfp_move->sp = sp + 1;
```

ソースコード 5.10: 制御フレームの初期化

5.3.9 評価

本研究では，テストプログラムとしてソースコード 5.11 と 5.12 を使用した．これらのプログラムは，スタックの大きさを 6 KB と標準より小さくした場合にスタックオーバーフローとなる．しかし，ext_stack 関数を用いることでスタックオーバーフローを回避し，正常に処理を終わらせることが可能となった．

Ruby 2.4.0 において，スタックの大きさの標準は 1024 KB に設定されている．ソースコード 5.11 について，本研究では 6 KB から始め 2 倍に拡張したため 12 KB と標準の 85 分の 1 のメモリ使用量で実行することが可能となる．ソースコード 5.11 は，スタックの大きさのチューニングを厳密に行えば 8 KB で実行することが可能であるがプログラムごとにスタックの大きさのチューニングをすることは，開発者にとって負担となるため本研究は省メモリ化に有効的であると考えられる．

また ソースコード 5.12 のようなマルチスレッドを含むプログラムでは，本来であればもっとも再帰の深くなるスレッドにスタックの大きさを合わせなければならなく，今回の例では thr1 を処理可能であるスタックの大きさに合わせる必要がある．そのため，スタックの大きさをチューニングしても 40 KB 必要であったが，本研究では 36 KB で正常な処理が可能であり省メモリ化に成功したと言える．また，図 5.4 に示した通り，スレッドの本数が増えるほど省メモリ化に有効的である．

```
1  def foo2(n)
2  puts "#{n}"
3    1.times do
4      if n <=12
5        foo1(n)
6      end
7    end
```

```
8  end
9
10 def foo1(n)
11   1.times do
12     n += 1
13     foo2(n)
14   end
15 end
16
17 foo1(0)
```

ソースコード 5.11: ext_stack 関数で実行可能となったプログラム

```
1  def foo3(a)
2    if a <= 13 then
3      a += 1
4      foo1(a)
5    end
6  end
7
8  def foo2(n)
9    foo3(n)
10 end
11
12 def foo1(num)
13   1.times do
14     foo2(num)
15   end
16 end
17
18 thr1 = Thread.new { foo1(0) }
19 thr2 = Thread.new { foo1(3) }
20 thr3 = Thread.new { foo1(5) }
21 thr4 = Thread.new { foo1(10) }
22 thr5 = Thread.new { foo1(12) }
23
24 thr1.join
25 thr2.join
26 thr3.join
27 thr4.join
28 thr5.join
29 puts "finish"
```

ソースコード 5.12: ext_stack 関数で実行可能となったマルチスレッドを含むプログラム

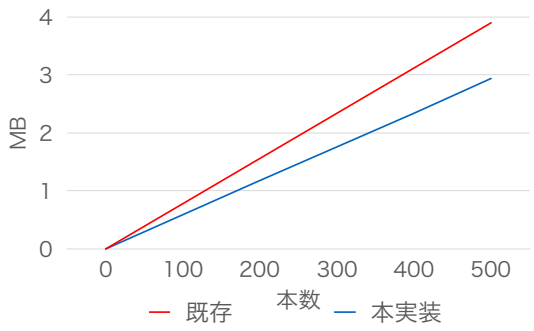

図 5.4: スレッド本数とメモリ使用量

しかし,新しく確保したスタック領域内でセグメンテーション違反（Segmentation Fault）を起こす箇所が存在したため正常に処理できなかったプログラムもあった．このセグメンテーション違反の原因としては，二つ推測ができる．一つ目は，内部スタックおよびコールスタックの移動の際に誤りがあると考えられる．スタック移動時のループの終了条件や memcpy 関数の引数に誤りがあることで新しいスタック領域内に余分なものを移動したり，誤った位置に移動してしまっていることが原因と考えられる．二つ目は，マルチスレッド時に拡張するスタック領域の数が複数に対応できていないことが考えられる．本研究では，仮想アドレス空間からスタック領域の拡張分を確保することでメモリの重複を避けた．しかし，複数のスタック領域を拡張しようとする際に，他のスタック拡張領域とメモリの重複が生じてしまいセグメンテーション違反を起こす．

第6章　結論

6.1　まとめ

本研究では，プログラミング言語 Ruby の VM である YARV のスタック領域の拡張を行った．省メモリのためにスレッド一つあたりのスタック領域を小さくすることが望ましいが，スタックの大きさと再帰の深さは対応しているため，もっとも再帰の深いスレッドにスタックの大きさを合わせなければならない．

そこで本研究では，開発者による利用率が多い Linux OS において，スタックオーバーフローが生じた際に，元のスタック領域よりも大きいスタック領域を新しく確保し，元のスタック領域の内部スタックおよびコールスタックを新しいスタック領域にコピーし，スタック領域に余裕を持たせることでスタックオーバーフローを回避し，正常に処理ができるような関数を実装した．

結果として，スタックオーバーフローになっていたプログラムを正常に処理することが可能となり，スレッド一つあたりのスタック領域を小さくすることに成功した．本研究では，ヒープ領域からスタック領域を確保して，スタック領域の拡張を行う場合のみ mmap システムコールを用いて仮想アドレス空間よりメモリを確保した．これにより，他のスタック領域とのメモリの重複を避け，マルチスレッド時にもスレッド一つあたりの大きさを小さくすることが可能になった．よって，従来のもっとも再帰が深いスレッドに合わせていた実装に対し省メモリ化の成功が見られた．

現在，まだ一部のプログラムは正常に処理することができない．これは，内部スタックおよびコールスタックの移動が完全でないことが原因だと考えられる．また，マルチスレッド時に拡張するスタック領域の数を複数に対応させることも本研究の課題である．この課題を解決し，Ruby に組み込まれることでより多くのプログラムの省メモリ化に貢献することを強く願う．

6.2　今後の展望

スタック領域の拡張を複数回可能にすることで，ほぼ再帰の深さを関係なく正常に処理することができるだけでなく，スレッドに割り当てるスタック領域の最初の大きさを最小限にすることが可能となるため，より省メモリにすることが期待できる．

謝辞

本論文を執筆するにあたり，多くの方々から多大なる御指導および御指摘を頂きました．このような研究の契機と環境を与えてくださり，親身に御指導して頂いた Martin J. Dürst 教授に心から感謝申し上げます．ならびに，丁寧に論文の添削をしてくださった松原俊一助教のお心遣い，御指導に感謝申し上げます．

また，YARV の開発者である笹田耕一氏に研究室まで足を運んでいただき本研究を始めるきっかけをくださったことに感謝申し上げます．そして，YARV の構造についてご教授を頂きましたことを厚く御礼申し上げます．研究をはじめとする多くの助言や指摘をしてくれた研究室の友人や先輩のおかげで研究を続けてこれました．

皆様本当にありがとうございました．

2017 年　1 月

参考文献

[1] まつもとゆきひろ，David Flanagan．プログラミング言語 Ruby．オライリー・ジャパン，2009．

[2] 田浦健次朗原 健太朗．アト峻レス空間の大きさに制限されないスレット峻移動を実現する PGAS 処理系，2011．

[3] Ruby アソシエーション： Ruby 処理系の概要．http://www.ruby.or.jp/ja/tech/install/ruby/implementations.html，2017/1/24 閲覧．

[4] Ruby core． https://www.ruby-lang.org/en/community/ruby-core/，2016/12/29 閲覧．

[5] Ruby のインストール．https://www.ruby-lang.org/ja/documentation/installation/，2016/12/29 閲覧．

[6] 青木峰郎．Ruby ソースコード完全解説．インプレス，2002．

[7] Pat Shaughnessy 著，島田浩二・角谷信太郎共訳．Ruby のしくみ（Ruby Under a Microscope）．オーム社，2014．

[8] Bison 公式．https://www.gnu.org/software/bison/，2017/1/5 閲覧．

[9] YARV ソースコード一人勉強会． http://d.hatena.ne.jp/hzkr/20061027，2016/12/30 閲覧．

[10] 太田眞敬．Ruby におけるメモリ管理改善手法の提案．東京工業大学　大学院情報理工学研究科　数理・計算科学専攻，修士論文，2010．

[11] Function rb_define_singleton_method (Ruby 2.4.0)． https://docs.ruby-lang.org/ja/latest/function/rb_define_singleton_method.html，2017/1/5 閲覧．

[12] Macro alloca_n (ruby 2.4.0)． https://docs.ruby-lang.org/ja/latest/function/ALLOCA_N.html，2017/1/23 閲覧．

[13] Function ruby_xmalloc (ruby 2.4.0)． https://docs.ruby-lang.org/ja/latest/function/ruby_xmalloc.html，2017/1/5 閲覧．

[14] 川端崇裕，横堀直也，橋本大輔，佐野伸幸．独習 Ruby．翔泳社．

[15] 笹田耕一，松本行弘，前田敦司，並木美太郎ほか． Ruby 用仮想マシン YARV における並列実行スレッドの実装． 情報処理学会論文誌（PRO），Vol．48，pp．1-16，2007．

[16] 笹田耕一． YARV : Yet Another Ruby VM． http://www.atdot.net/yarv/，2016/12/29 閲覧．

[17] 笹田耕一． 高速な Ruby 用仮想マシンの開発． 情報処理，Vol．50，No．12，p．1212，2009．

[18] 千住次郎 訳Michael Kerrisk 著．LINUX プログラミングインターフェース．オライリー・ジャパン，2012．

付録

A. 質疑応答

発表後の質疑応答

Lopez 先生の質問（要約）

スタックオーバーフローを回避し，実行した例はないのか.

当日の回答

再帰呼び出しのプログラムとメソッド同士が呼び合うプログラムを例に説明しました.

大原 先生の質問（要約）

拡張回数は無限なのか.

当日の回答

現段階は，実装が完璧でないため一度きりとなっています．しかし，さらなる省メモリ化の目的で拡張回数を複数回にすることを今後の展望に挙げています.

大原 先生の質問（要約）

問題のあるプログラムに無制限に動かれると困る可能性があるが，拡張回数は無限で良いのか.

当日の回答

拡張回数を無限にすることは不可能だと思っています.

後日の回答

現在の Ruby の実装では，通常考えられる再帰の深さを処理するのに十分という見解で 1024KB と設定されています．よって，本研究でも拡張回数は複数可能となっても 1024 KB を越えない設定が好ましいです.

フォーラムの質疑応答

Lopez 先生の質問

スタックオーバーフローが起きにくい事例やその性能を定量的に評価できないか.

回答

Ruby では, Kernel#caller メソッドを用いてコールスタックが取得可能です. よって, Array#size メソッドを用いることでスタック領域を使用しているコールスタックは, ある程度把握できますが内部スタックに関して把握する手段がありません. そのため, 定量的な評価が難しいと考えています.

LOPEZ 先生の質問

必要なスタックサイズを事前に計算して設定することができないか.

回答

計算可能性理論における停止性問題（Halting Problem）に対し, アラン・チューリングが停止性問題を解くチューリング機械が存在しないことを証明しました. よって, スタックオーバーフローが理由で止まってしまうプログラムを選別するプログラムは作り出せないため不可能だと考えます.