# Ruby 1.9の高速なFiberの実装

東京大学大学院
情報理工学系研究科創造情報学専攻

芝 哲史
笹田 耕一

# 本発表の構成

- Ruby 1.9の高速なFiberの実装
  - Fiberとは
  - Fiberのボトルネック
  - 提案
  - 実装
  - 評価
  - まとめ

•Fiber ⇒ Ruby 1.9におけるコルーチン
•他の言語でも行われている手法で
高速化し評価
•Ruby 1.9.2に取り入れてもらうことが目標

# Fiberとは

- Ruby 1.9で新しく追加されたクラス
- 並行計算をサポート

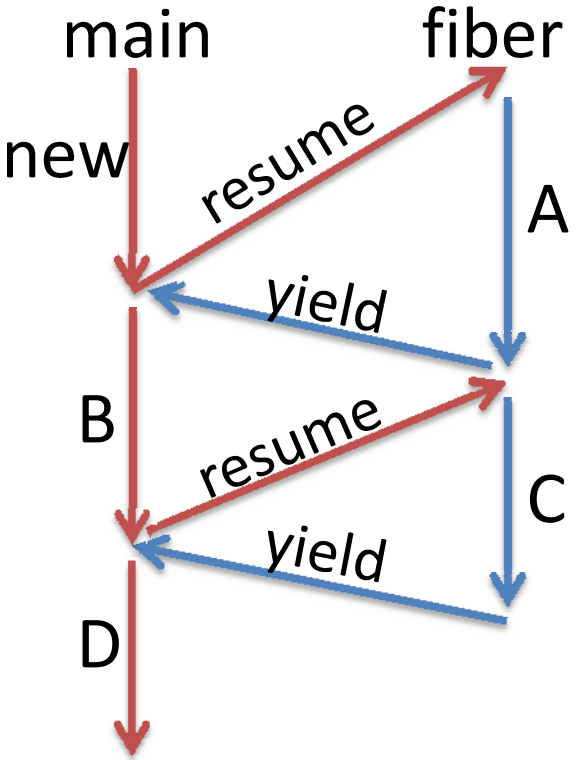
- 同様の機能は昔から存在
  - コルーチン
  - セミコルーチン

- Fiberという名称
  - Threadクラスを意識した名称
  - Threadよりも軽量だからFiber
  - Fiberはノンプリエンプティブ

# サンプルコード

```
fiber = Fiber.new do
  A
  Fiber.yield
  C
  Fiber.yield
end

fiber.resume
B
fiber.resume
D
```



# ThreadとFiber

- Threadと比べたFiberの利点
  - 同期的な並行処理を軽量に実行可能
  - スケジューリング可能
- Ruby 1.9のRuby Thread
  - カーネルレベルなのでシステムコールが必要
  - 切り替え先はカーネルスケジューラが指定
- Fiber
  - ユーザレベルなのでシステムコールは不要
  - 切り替え先はプログラマが指定

# Fiberの活用例[1/2]

- ゲームプログラミング

Fiber使用前

```
if state == 0
  A
elsif state <= 100
  B
  …
end
state += 1
return
```

Fiber使用後

```
A
Fiber.yield
100.times do
  B
  Fiber.yield
end
…
```

# Fiberの活用例[2/2]

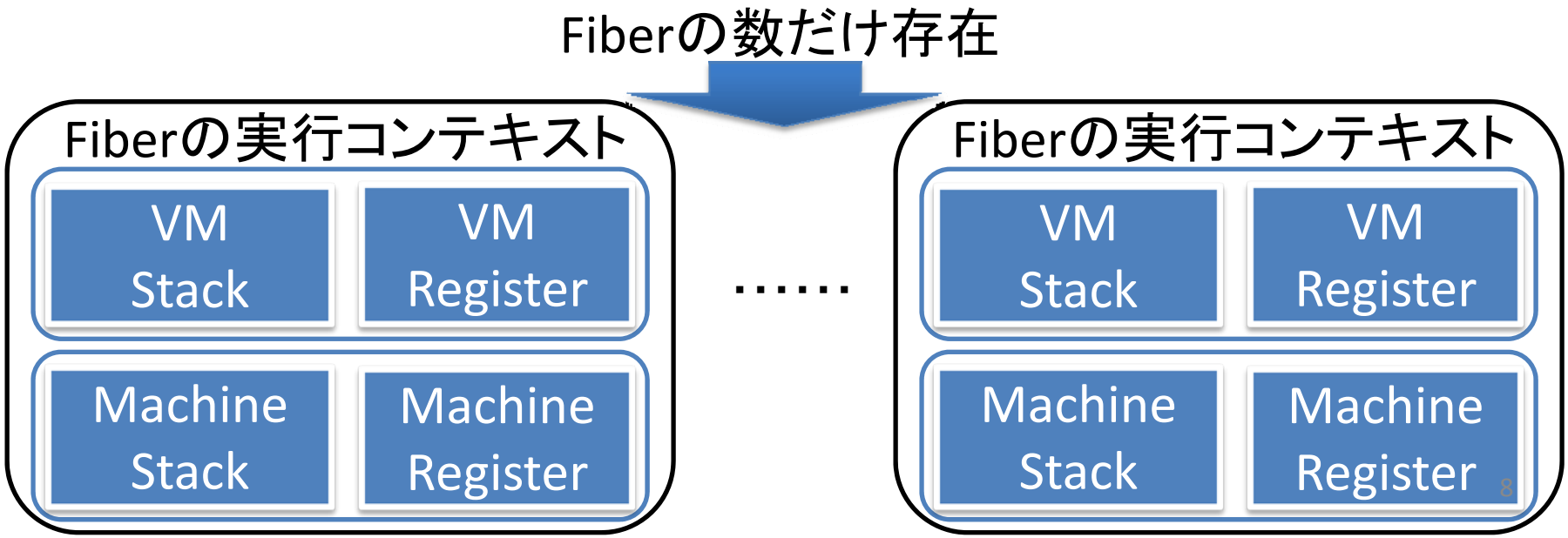
- 内部イテレータから外部イテレータへの変換
  - Enumerator, Generator

```
iter = Fiber.new do
  array1.each do |o1|
    Fiber.yield o1
  end
end

array2.each do |o2|
  o2 += iter.resume
end
```

# Fiberの実現法

- Fiber全てが実行コンテキストを保持
- Fiberの生成、削除 ⇒ 実行コンテキストの割当、解放
- Fiberの切り替え ⇒ 実行コンテキストの切り替え

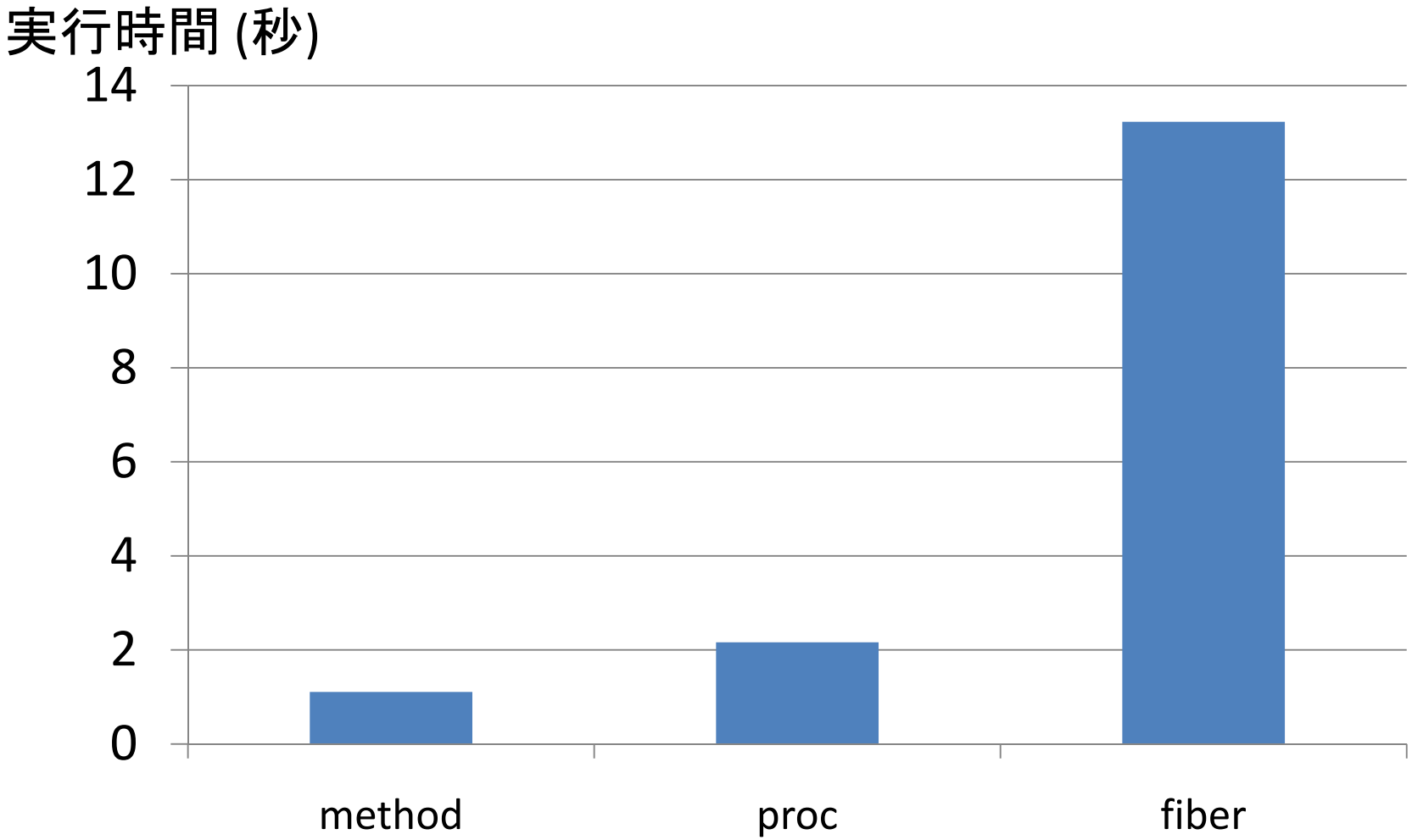


# Fiberの切り替えの速度

- ベンチマーク

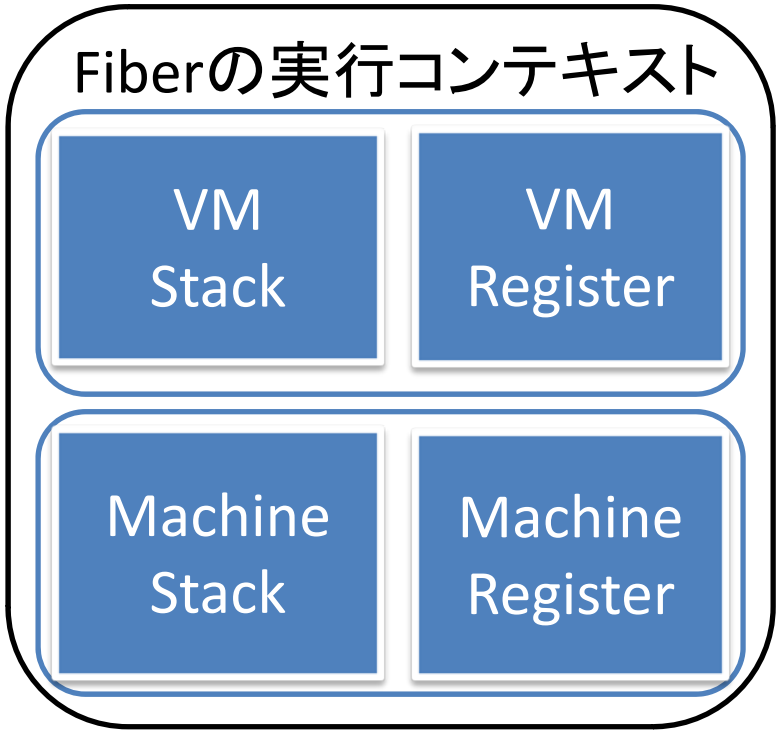
```
#Method
  def test
  end
  10000000.times { test }
#Proc
  p = Proc.new {}
  10000000.times { p.call }
#Fiber
  fiber = Fiber.new { Fiber.yield while true }
  10000000.times { fiber.resume }
```

# Fiberの切り替えの速度



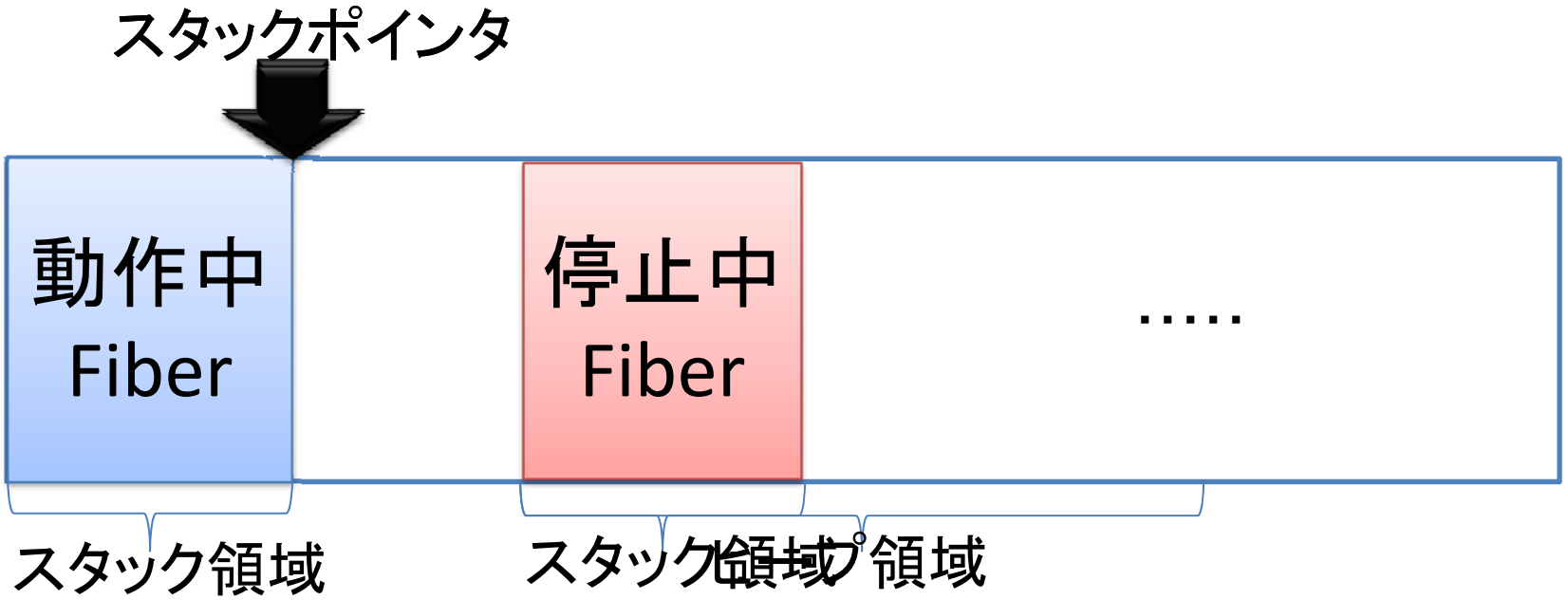
# 実行コンテキストの切り替え

- VMのスタック、レジスタ
  - ポインタの切り替え
- マシンレジスタ
  - set/longjmp
- マシンスタック
  - set/longjmp + memcpy



# マシンスタックの切り替え

- Rubyの現在の実装



# コピー処理の理由

- 移植性向上のため
  - OSやH/Wに依存した記述を排除するため
- Tips：歴史的経緯
  - Ruby 1.8以前のユーザレベルスレッドも、このような実装

# OSやH/Wに依存した処理

- アーキテクチャの数だけ下記のような分岐が必要

```
#if defined(__i386) || defined(__i386__)
#if __GLIBC__ == 2 && defined(JB_SP)
   /* x86-linux-glibc2 */
#elif defined(__linux__) && defined(_I386_JMP_BUF_H)
   /* x86-linux-libc5 */
#elif defined(__FreeBSD__)
   /* x86-FreeBSD */
#elif defined(__NetBSD__) || defined(__OpenBSD__)
   /* x86-NetBSD, x86-OpenBSD */
#elif defined(__solaris__) && _JBLEN == 10
   /* x86-solaris */
#elif defined(__MACH__) && defined(_BSD_I386_SETJMP_H)
   /* x86-macosx */
#endif
```

# 問題点と課題

- 問題点
  - 移植性のためにマシンスタックをコピーしているため、Fiberのコンテキスト切り替えが遅い

- 課題
  - **移植性**の高い、
    **高速**なマシンスタック切り替え手法

# 提案

- 移植性を考慮したFiberの高速化
  - システムが提供するコンテキスト切り替え手法を利用したFiberのコンテキスト切り替え

- （既知の）方法でRubyのFiberを高速化
- 本研究の貢献
  - 実際にRubyに対して実装、様々な環境で評価
    - 本当にちゃんと動くかどうか
    - どれくらい速くなったか
  - Rubyでは初めて

# 既存の手法

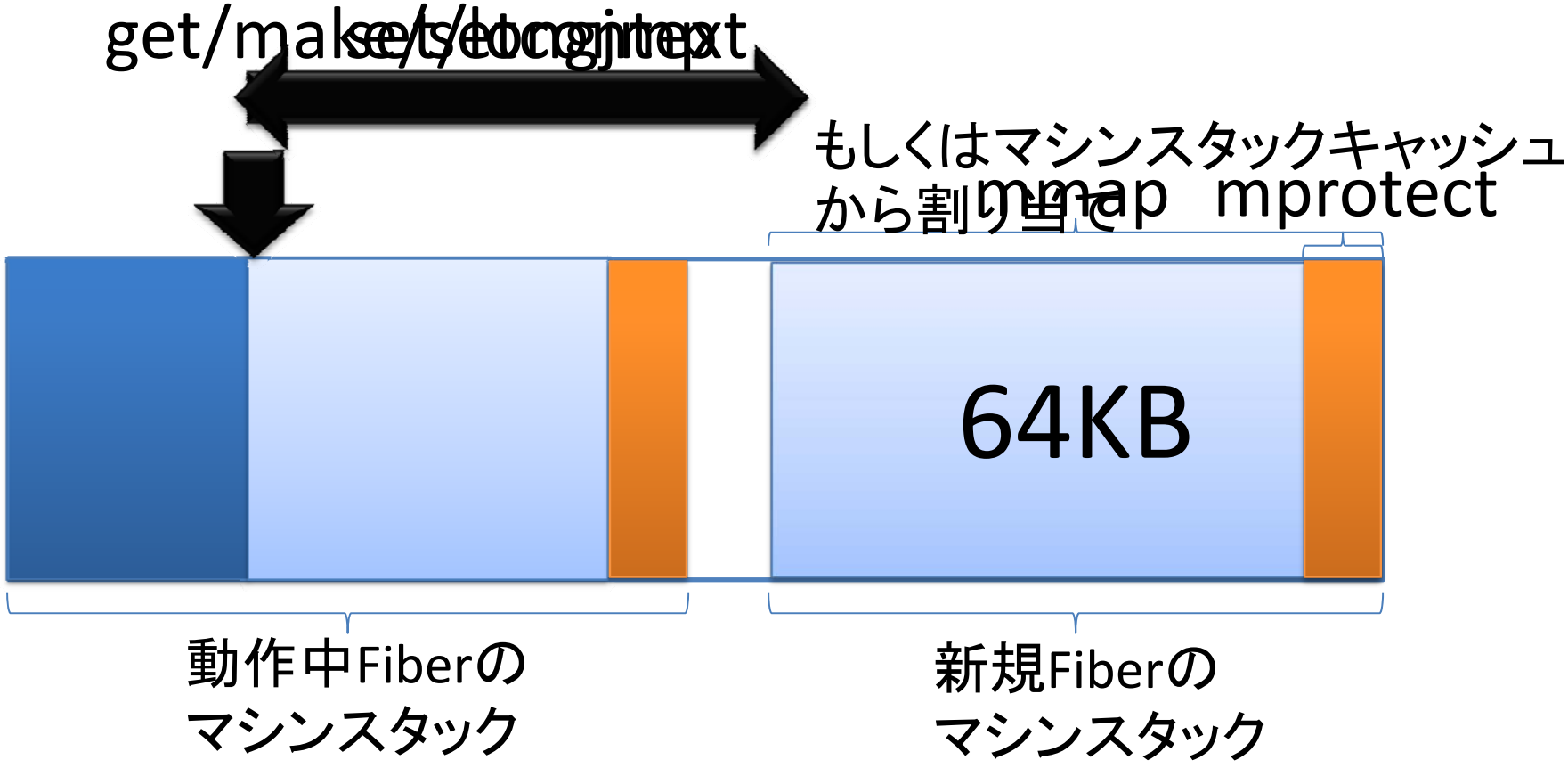
- 既存のコルーチンライブラリ
  - Coro (Perl)
  - Coco (Lua)
- コピー無しのマシンスタックの切り替え手法
  - アセンブリ言語
  - set/longjmpのjmpbuf書き換え
  - sigaltstack
  - **ucontext**
  - **Win32Fiber**

# 実装手法

- システムが提供する移植性の高いコンテキスト切り替え手段 (API) を利用
  - POSIX OS (UNIX) の場合
    - makecontext, setcontext, …
  - Windows の場合
    - CreateFiberEx, SwitchToFiber, …
  - その他
    - 既存手法をそのまま利用

# 実装

- POSIX OSの場合



# 工夫したこと

- マシンスタックキャッシュ
  - 終了したFiberのマシンスタックをN個解放せずに保持して、再利用(今回は N = 10)
- get/set/makecontextとset/longjmpの併用
  - setcontextはsigprocmaskシステムコールを伴う
  - RubyのFiberはスレッドをまたげないので切り替え元Fiberと切り替え先Fiberのシグナルマスクは同じ
  - 最初の切り替え以降はset/longjmp

# 基本性能の評価

- 評価項目
  - 移植性
  - 速度
    - コンテキストスイッチの速度
    - 生成、終了速度
  - 最大生成数

- 実装の呼び名
  - Ruby 1.9の既存の実装 ⇒ 既存手法
  - 今回行った実装 ⇒ 提案手法

# 移植性の評価

9 / 11 OS
4 / 4 CPU
で動作

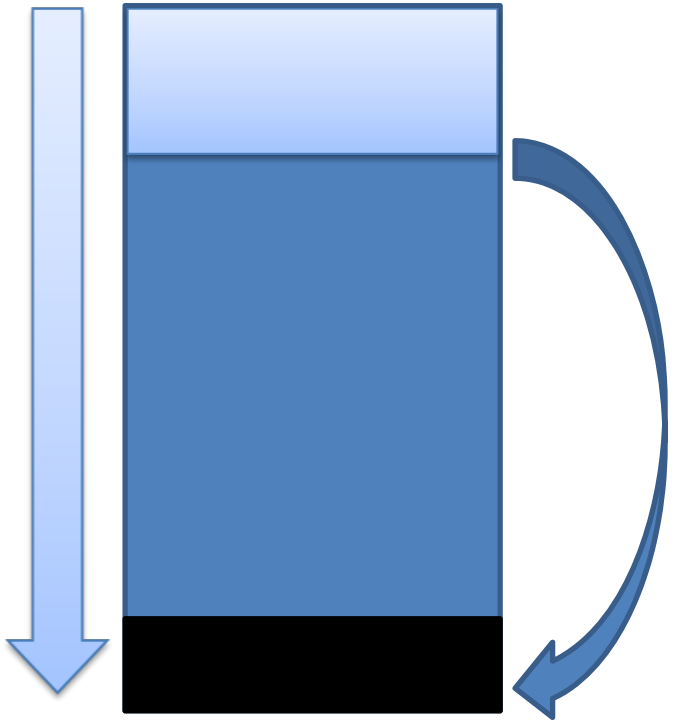
| OS | CPU | 結果 |
|---|---|---|
| Windows XP | x86 | ○ |
| Windows 7 | x86 | ○ |
| Linux 2.6.31 | x86 | ○ |
| Linux 2.6.24 | x86_64 | ○ |
| Linux 2.6.31 | Cell (PowerPC) | ○ |
| MacOSX 10.5.8 | x86 | ○ |
| SunOS 5.10(Solaris) | | ○ |
| SunOS 5.11(OpenSolaris) | | ○ |
| FreeBSD 7.2 | | ○ |
| NetBSD 5.0.1 | x86(VM) | × |
| OpenBSD 4.5 | x86(VM) | × |
| DragonFlyBSD 2.4.1 | x86(VM) | ○ |

Pthreadとの相性

APIに未対応

# NetBSDの場合

- Pthread系APIのマシンスタックの位置に依存した処理
- マシンスタックの位置が変わるとTLSまで変わる



```
pthread_mutex_lock{
    ptr = SP & ~(STACKSIZE – 1)
    ptrの操作
}
```

# 速度と最大生成数の評価環境

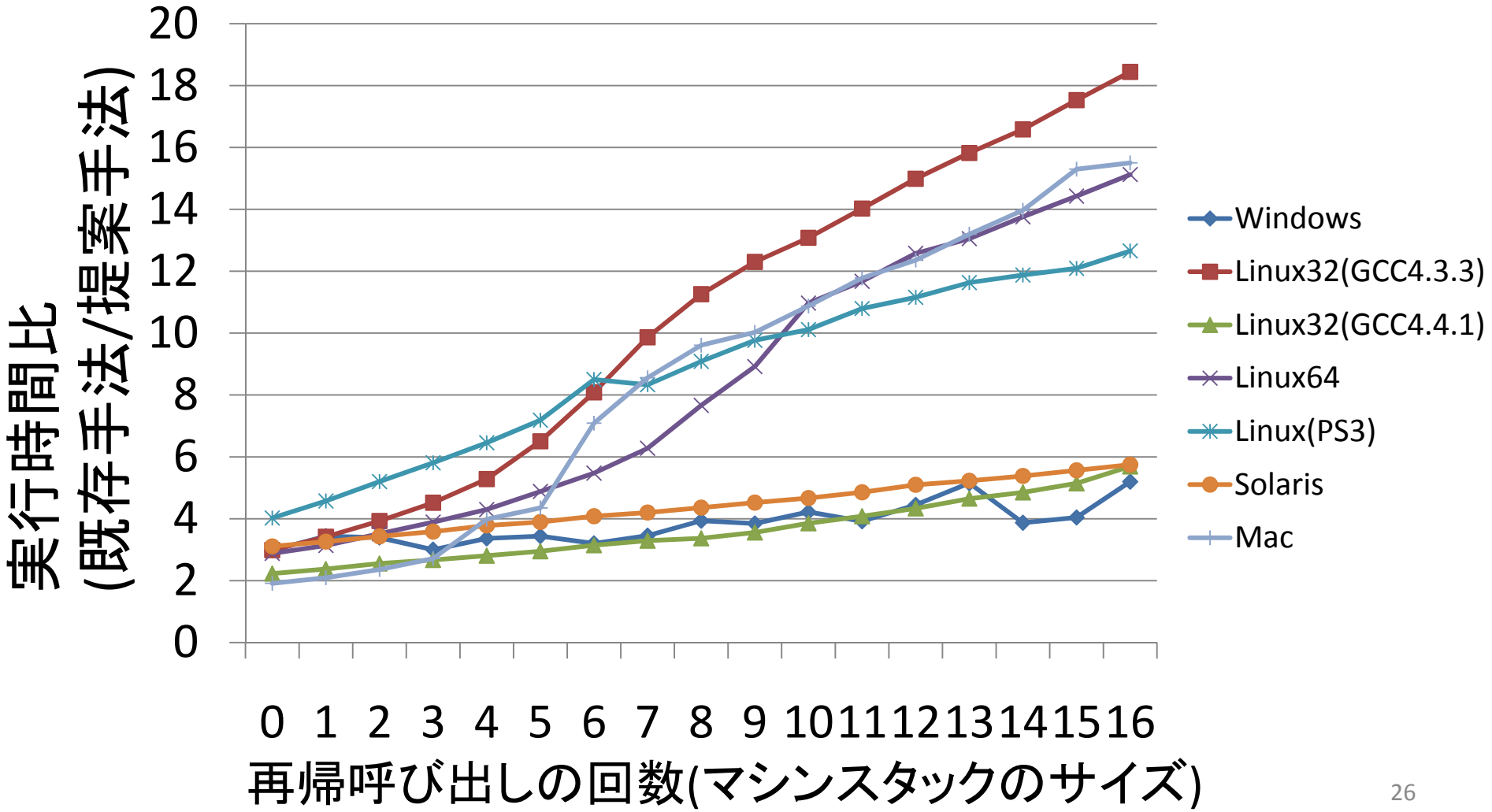
| 環境 | OS | CPU | コンパイラ |
|---|---|---|---|
| Windows | Windows7<br>32-bit | Intel Core2Duo<br>2.00GHz | VC2008 |
| Linux32<br>(GCC4.3.3) | Linux 2.6.31<br>32-bit | IntelCore2Quad<br>2.66GHz | GCC4.3.3 |
| Linux32<br>(GCC4.4.1) | Linux 2.6.31<br>32-bit | IntelCore2Quad<br>2.66GHz | GCC4.4.1 |
| Linux64 | Linux 2.6.26<br>64-bit | IntelXeon<br>2.00GHz | GCC4.3.2 |
| Linux<br>(PS3) | Linux 2.6.28<br>64-bit | Cell 3.2GHz | GCC4.1.2 |
| Solaris | SunOS 5.10<br>32-bit | sparcv9 1.167GHz | GCC3.4.3 |
| MacOSX | MacOSX 10.5.8<br>32-bit | IntelCore2Duo<br>2.20GHz | GCC4.0.1 |

# コンテキストスイッチの速度の評価

- マイクロベンチマーク
  - Fiberを1本生成
  - マシンスタックを伸ばすtimesメソッドをX回再帰呼出し
    - マシンスタックのコピー処理がボトルネック
    - 速度への影響を計測
  - Fiberのコンテキストスイッチを300万回

- 提案手法と既存手法の実行時間比で評価

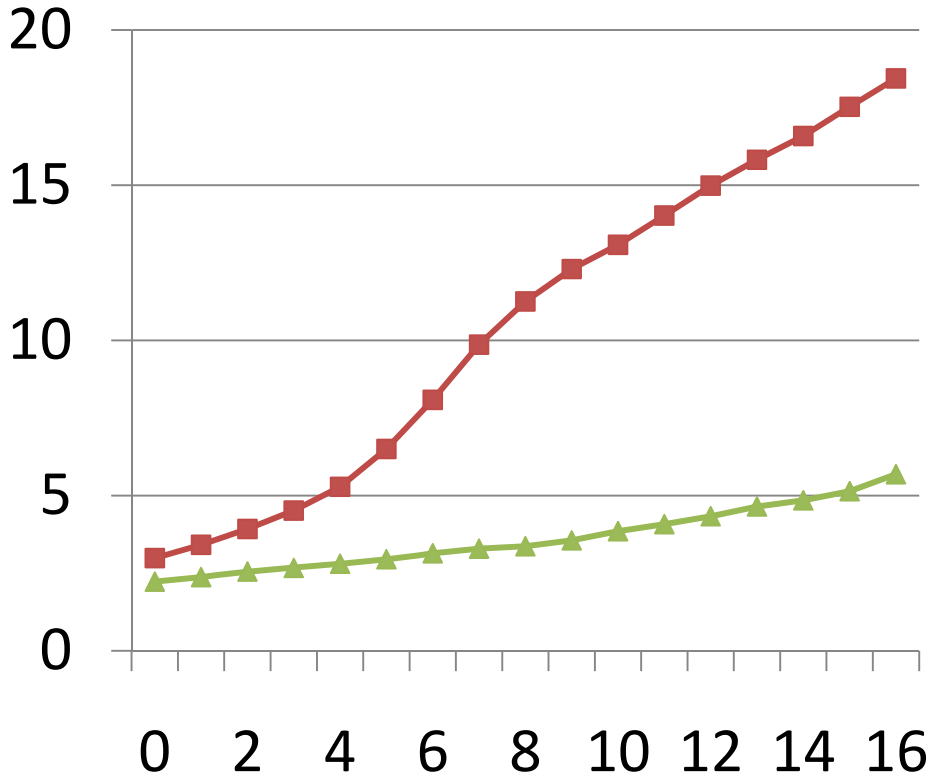
# コンテキストスイッチの速度の評価



# コンパイラの影響

- コンパイラによるマシンスタック消費量の差
  - 速度への影響が大きい

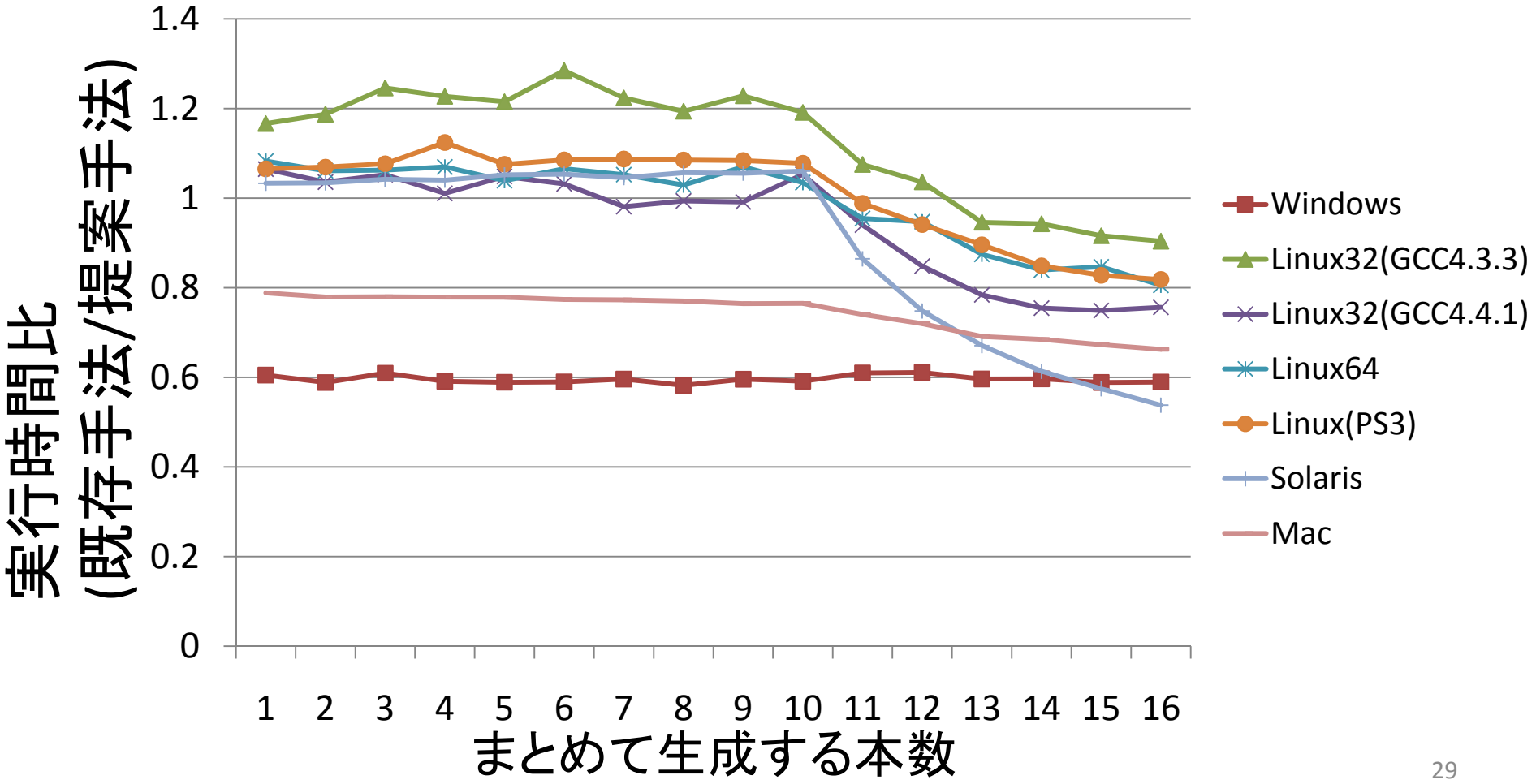
| | マシンスタック増加量(バイト) |
|---|---|
| GCC4.3.3 | 2048 |
| GCC4.4.1 | 1040 |



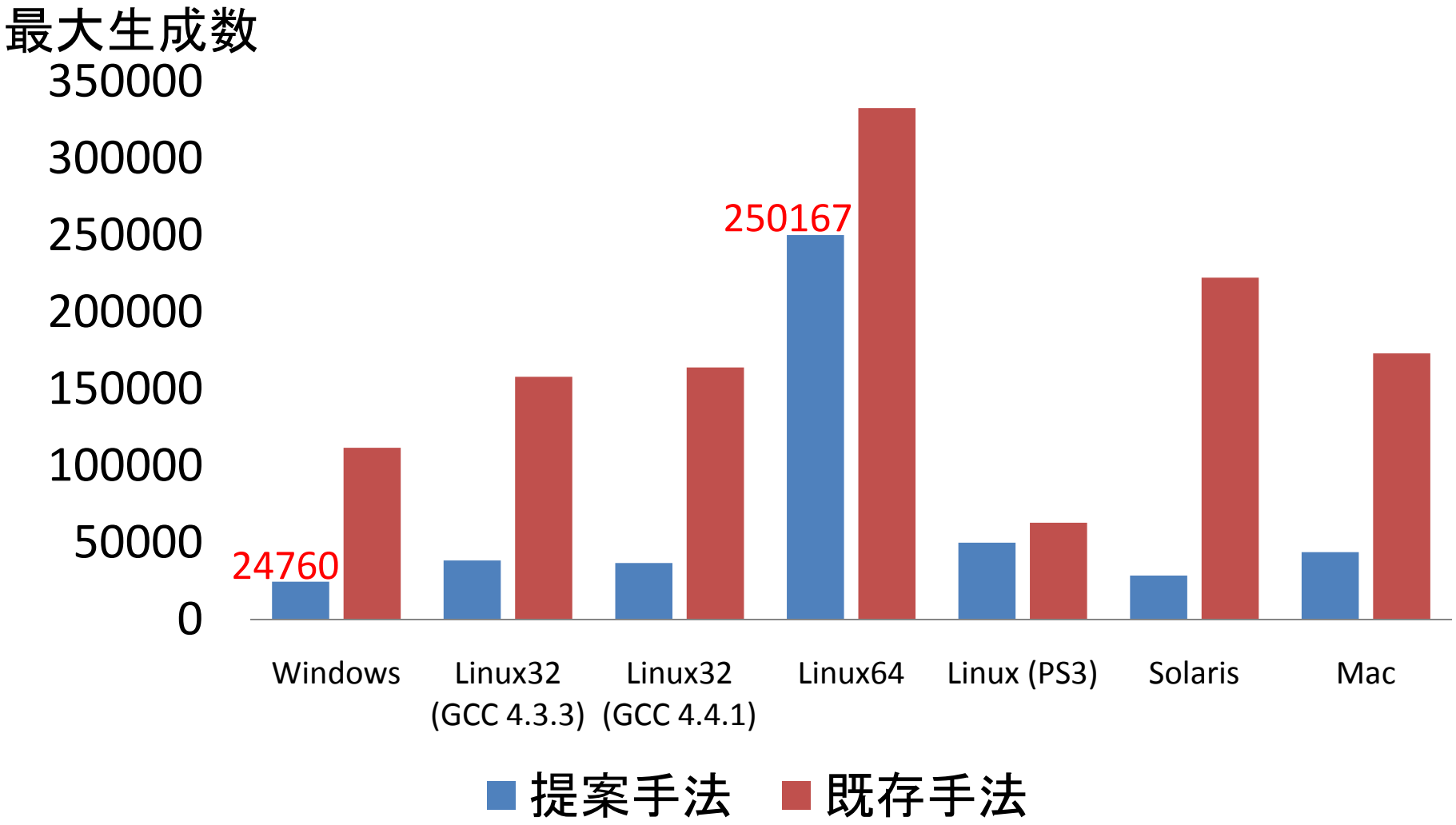
# 生成、終了速度の評価

- マイクロベンチマーク
  - マシンスタックキャッシュの影響を計るために以下を繰り返す
    - FiberをX本まとめて生成
    - FiberをX本まとめて終了
  - 最終的に100万本のFiberの生成、終了処理
  - マシンスタックキャッシュの本数は10
  - まとめて生成する本数が10を超えると速度差のつきかたが変わるはず
- 既存手法と提案手法の実行時間比で評価

# 生成、終了速度の評価


実行時間比
(既存手法/提案手法)
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
まとめて生成する本数
Windows
Linux32(GCC4.3.3)
Linux32(GCC4.4.1)
Linux64
Linux(PS3)
Solaris
Mac

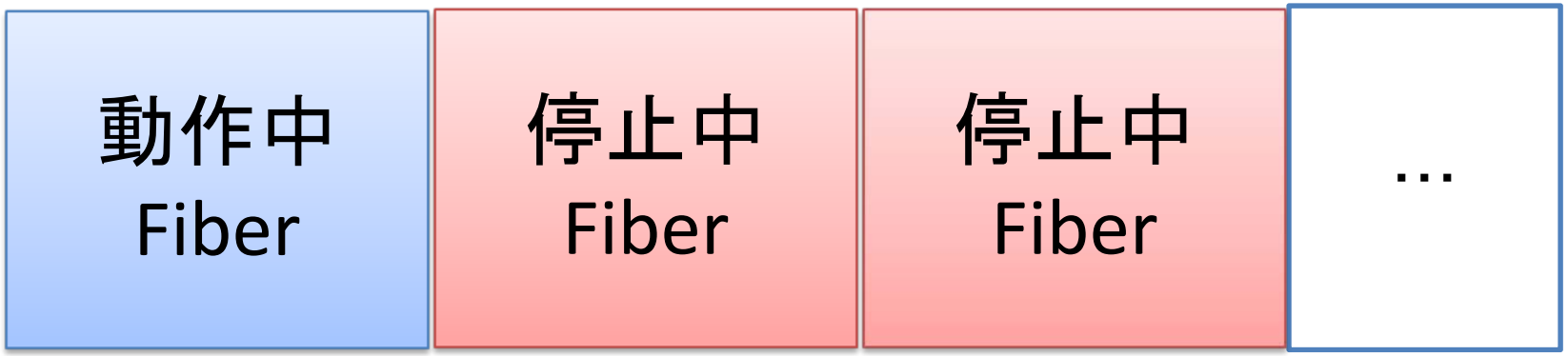
# 最大生成数の評価


最大生成数
350000
300000
250000
200000
150000
100000
50000
0
24760
250167
Windows
Linux32 (GCC 4.3.3)
Linux32 (GCC 4.4.1)
Linux64
Linux (PS3)
Solaris
Mac
提案手法
既存手法

# 最大生成数の差

- 停止中のFiberのアドレススペースの使い方

提案手法




既存手法

# 基本性能の評価結果

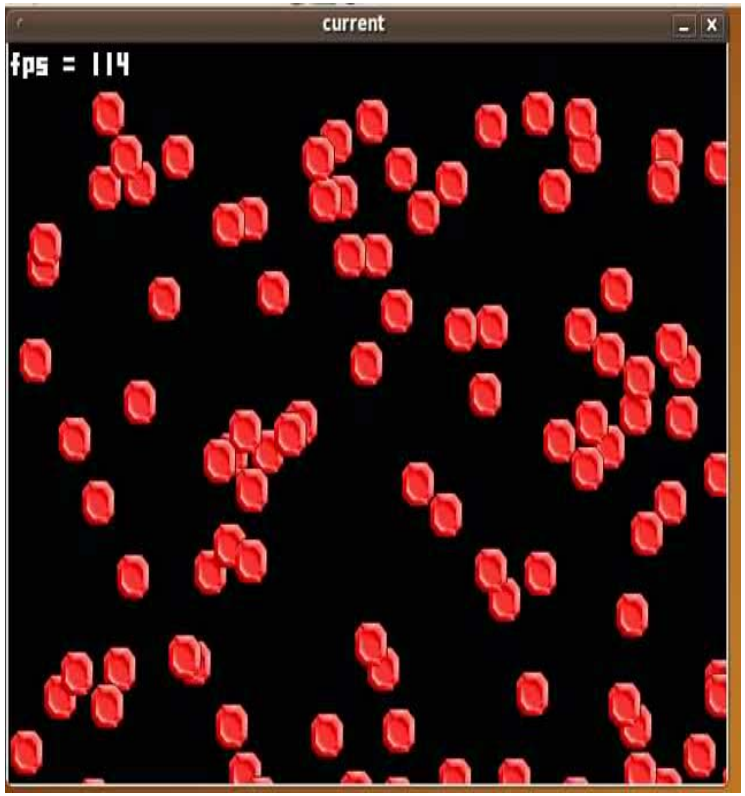
- 移植性 ⇒ 概ね達成
  - 動く環境 (Windows, Linux, MacOSX, Solaris, FreeBSD, DragonFlyBSD)
  - 動かない環境 (OpenBSD, NetBSD)
- 速度 ⇒ 達成
  - コンテキストスイッチは提案手法の方が2倍以上高速
    - マシンスタックサイズに比例した速度差
  - ただし、生成、終了は既存手法の方が高速
- 最大生成数 ⇒ 許容範囲
  - 数万のFiberを生成可能
  - ただし、最大生成数は既存手法の方が多い

# ゲームプログラミングにおける評価

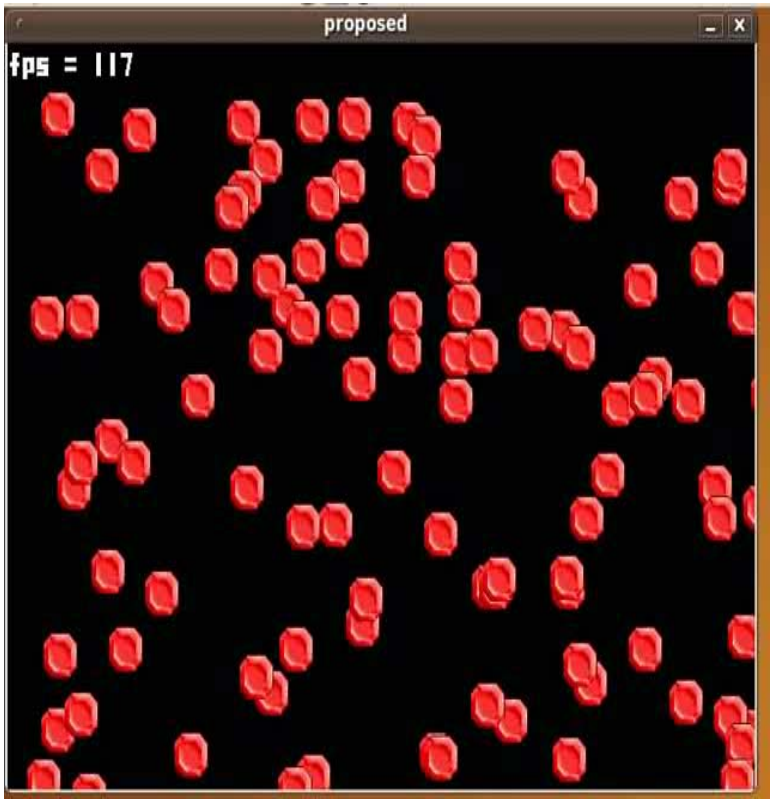
- ゲームプログラミングのマイクロベンチマーク
  - 画面内の描画対象のオブジェクト1つ1つにFiberを対応付け
  - Fiberの中で各オブジェクトの位置情報を更新
  - メインループでオブジェクトの位置情報を全て更新し、画面に反映
  - 1秒間に画面を更新できた回数 (FPS) で
    速度差を評価
  - 描画対象のオブジェクトが増すほど、Fiberの切り替え処理の割り合いが増し、速度差がつく

# ゲームプログラミングにおける評価

- 100個のオブジェクトを動かしてみた場合



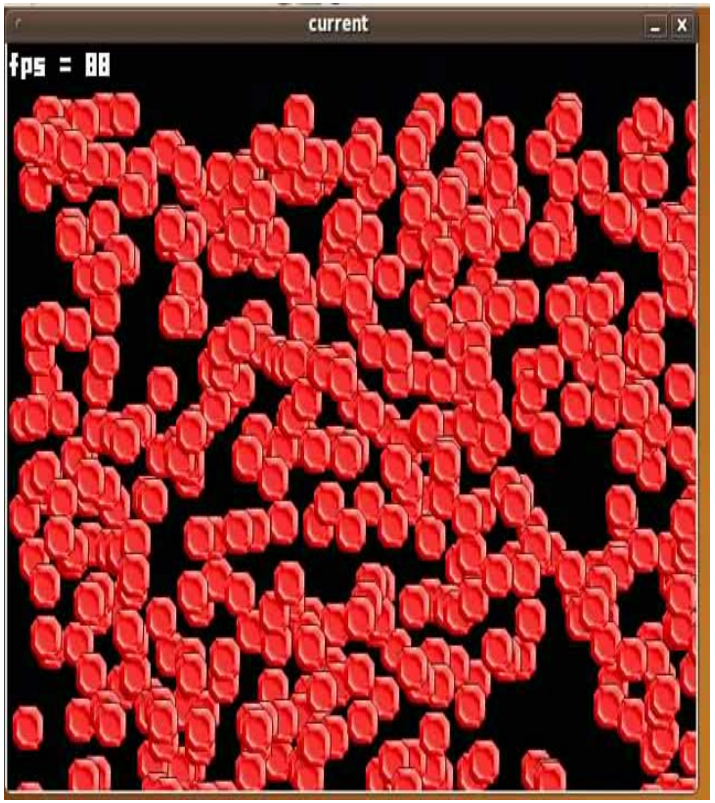

既存手法
FPS 115-120



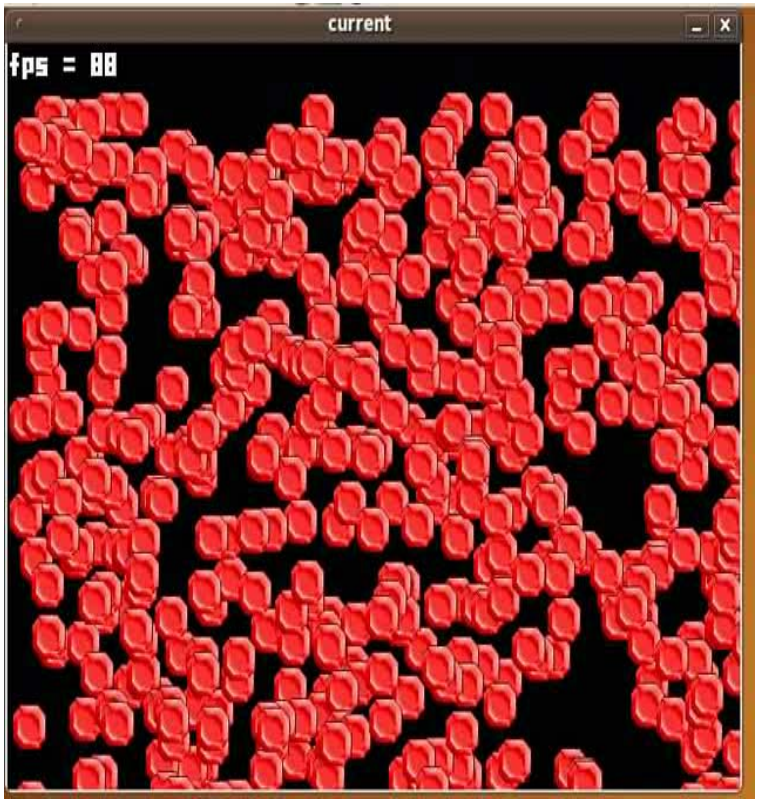

提案手法
FPS 115-120

# ゲームプログラミングにおける評価

- 500個のオブジェクトを動かしてみた場合



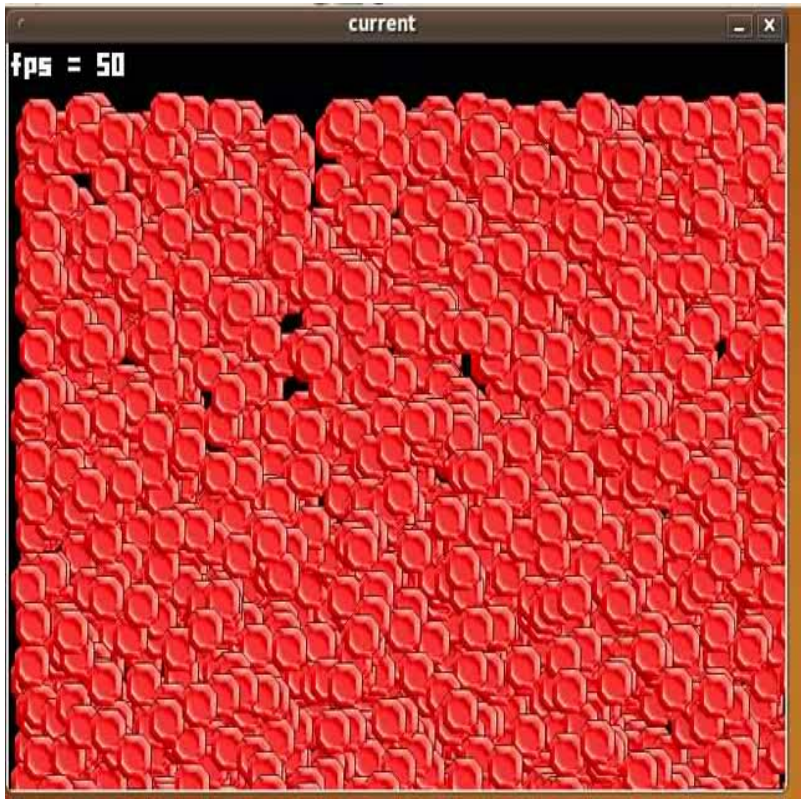

既存手法
FPS 85-90



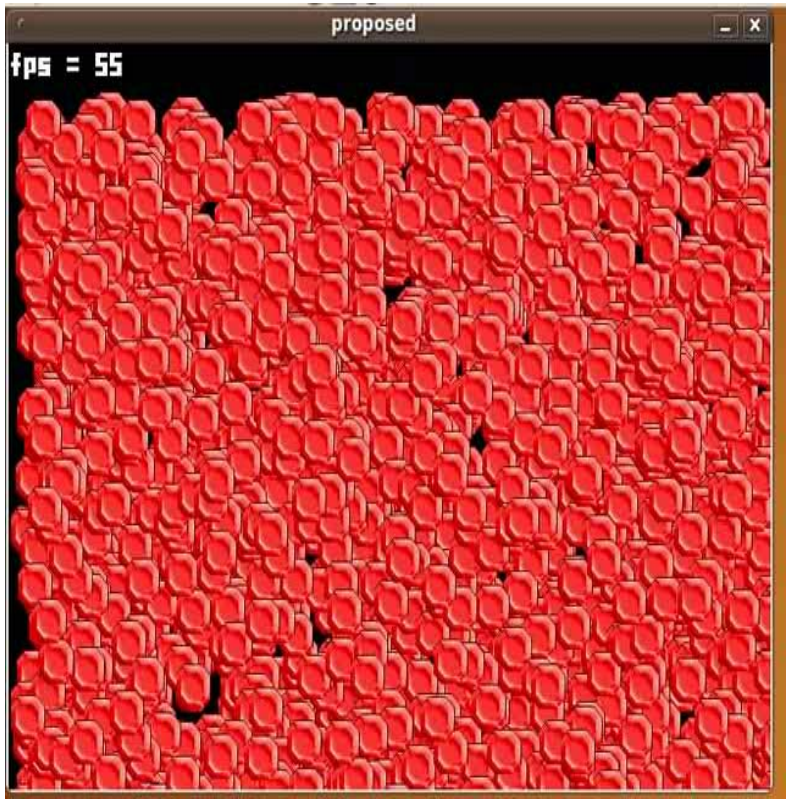

提案手法
FPS 85-90

# ゲームプログラミングにおける評価

- 2000個のオブジェクトを動かしてみた場合




既存手法
FPS 50-55




提案手法
FPS 55-60

# まとめ

- Ruby 1.9のFiberは便利だが遅かった
  - コピー処理を伴うマシンスタック切り替え
- 提案
  - コピーを行わないマシンスタック切り替え
  - 多くの環境で利用可能なAPIを使用
- 基本性能の評価
  - 移植性　　多くの環境で動作
  - 速度　　高速化
  - 最大生成数　　数万のFiberを生成可能
- ゲームプログラミングでも高速化（Fiberの数次第で）
- 今後の課題
  - Ruby 1.9.2に取り入れてもらえるように、交渉
  - 更なる速度向上

ご清聴ありがとうございました

# set/longjmpでコピーを伴わないマシンスタック切り替えが実装できない理由

- setjmpを呼んだ時点のマシンレジスタを保存
  longjmpで復元
- setjmpを呼んだ場所にしかlongjmpで飛べない