Vol.2018-OS-144 No.1
2018/7/30

Nodeプログラミングモデルを活用したC++およびElixirの実行環境の実装

山崎 進[1,a)]　森 正和[2,b)]　上野 嘉大[2,c)]　高瀬 英希[3,d)]

概要：Node.js では，コールバックを用いて I/O を非同期的に扱ってノンプリエンプティブなマルチタスクにする機構 Node プログラミングモデルが備わっている．これにより，ウェブサーバーのメモリ使用量を格段に減らすことができ，同時セッション最大数やレイテンシが改善される．我々は C++ と Elixir で同様の機構を実装した． C++ への実装を Zackernel と称し，Elixir への実装を軽量コールバックスレッドと称している．Zackernel は RFID のような極端に小規模で消費電力の少ない IoT システムを組む場合のカーネルとしての用途，軽量コールバックスレッドはクラウドサーバーでの用途をそれぞれ想定している．Zackernel は C++ 11 で採用された匿名関数を利用して，dispatch メソッドにて次に呼び出すべき関数をキューから読み込んで呼び出すという原理で実現する．軽量コールバックスレッドは，関数のリストをキューとして保持し，キューの先頭の関数の実行が終わったら次の関数を呼び出すという原理で実現する．これらと従来のプロセス/スレッドとの 1 プロセス/スレッドあたりのメモリ消費量を比較する評価実験を行なった．その結果，Zackernel は 1 スレッドあたり 204 バイトと最も少なく，軽量コールバックスレッドが 1 スレッドあたり 1332 バイト，Elixir プロセスが 1 プロセスあたり 2835 バイト，C++ 11 スレッドが 1 スレッドあたり約 546KB であった．今後，プロセス間通信の機能を実装し，ベンチマークプログラムに適用して性能を評価したい．

キーワード：Elixir, C++, Node.js, マルチタスク

Implementation of Runtime Environments of C++ and Elixir with the Node Programming Model

Yamazaki Susumu[1,a)]　Mori Masakazu[2,b)]　Ueno Yoshihiro[2,c)]　TAKASE Hideki[3,d)]

Abstract: Node.js has the Node programming model, which is a non-preemptive multi-tasking mechanism that processes I/O asynchronously using callbacks. Thus, it improves maximum concurrent sessions and latency of web servers because it requires few memory. We implement it in C++ and Elixir. We call the implementation in C++ and Elixir Zackernel and light-weight callback threads (LCTs), respectively. We assume that Zackernel will be used as a kernel in an extremely small and power-saving IoT system such as an RFID-based IoT, and that LCTs will be used as a cloud server. Zackernel is implemented using anonymous functions, which has been adopted by C++ 11, by the principle that the `dispatch` method calls back the next process function, with reading from the scheduler queue. LCTs is realized by the priciple that implements the scheduler queue as a function list and call back the next process function after running the function of the head of the queue. We examine and evaluate the consumption of memory a per process or thread of these and the traditional processes and threadsl. Thus, Zackernel, LCTs, Elixir processes and C++ 11 threads consume 204 bytes, 1332 bytes, 2835bytes and 546KB a per thread or process, respectively. We will implement an inter-process communication mechanism, apply it a benchmark, and evaluate performance of it.

Keywords: Elixir, C++, Node.js, multi-tasking

1. はじめに

Node.js では，コールバックを用いて I/O を非同期的に扱ってノンプリエンプティブなマルチタスク処理にする機構，Node プログラミングモデルが備わっている [5]．これにより，ウェブサーバーのメモリ使用量を格段に減らすことができ，同時セッション最大数やレイテンシが改善される [3]．

もし Node プログラミングモデルと同様の機構を，Javascript よりパフォーマンスの良いプログラミング言語で実装したならば，ウェブサーバーの性能を格段に向上させることができる．そこで，我々は C++ と Elixir [6] の 2 つのプログラミング言語を選んで実装を試みた．C++ を選んだのは，RFID のような極端に小規模で消費電力の少ないような電子回路を用いて IoT デバイスを構築することを意図し，できるだけ小さなメモリ容量で実現でき，かつ C++ 11 は Node プログラミングモデルの実現に必要な匿名関数を備えているからである．Elixir を選んだのは，Elixir で書かれたウェブサーバーフレームワークである Phoenix を用いることで，極めてレスポンス性の高いウェブサーバーを構築できる [1] からである．

我々の C++ への実装を Zackernel と称し，Elixir への実装を軽量コールバックスレッドと称している．これらを組合わせることで，世界中にばらまいた膨大な数の RFID による極小 IoT デバイスと，それらから随時リクエストを受け付ける IoT サーバーからなる IoT システムを構築することができる．

本報告では，Zackernel と軽量コールバックスレッドをどのように実装したのかを紹介し，メモリ消費量に焦点を当てて評価を行なう．

本報告のこの後の構成は次のとおりである．第 2 章では着想の元となった Node プログラミングモデルについて紹介する．第 3 章では Zackernel の設計と実装を，第 4 章では軽量コールバックスレッドの設計と実装をそれぞれ示す．第 5 章ではメモリ消費量の評価を行う．最後に第 6 章でまとめと将来課題について述べる．

2. Node プログラミングモデル

Node プログラミングモデル [5] の一例として，Node.js のウェブサイト [2] に掲載されているプログラム例を図 1 に示す．このウェブサーバーのプログラムを実行してウェブブラウザで `http://localhost:3000` にアクセスすると `Hello World` と表示される．このウェブサーバーに接続があるごとに，下記のコールバック関数が呼び出されるが，この際にスレッドを生成してスタック領域を確保するようなことはしない．

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.listen(port, hostname, () => {
  console.log(
    'Server running at http://${hostname}:${port}/'
  );
});
```

図 1 Node.js のコード例
Fig. 1 A Sample Code of Node.js

```
(req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
}
```

コールバック関数は，この例のように**匿名関数**として定義することもできるため，プログラムのメソッドや関数の中に記述できる．そのため，処理の流れを分断することなくプログラミングできる．

コールバック関数の呼出しは，通常のメソッドや関数の呼出しと同等である．したがって，スタックメモリのような広大なメモリ領域を消費することなく，かつ迅速に呼び出すことができる．

3. Zackernel

我々は Node プログラミングモデル [5] に着想を得て，C++ で実装した，コールバック関数を用いてマルチタスク処理にするライブラリである Zackernel を開発した[*1]．同様のものに libevent [4] があるが，Zackernel は C++ 11 で導入された匿名関数を用いることで可読性を改善している．

Zackernel の内部構成について説明する [8]．Zackernel の Schedule クラスはコールバックする関数を保持し，Schedule 同士を線形リスト構造でつないでいる．Zackernel の Zackernel クラスは Schedule のキューを保持する．核心と

1 北九州市立大学
University of Kitakyushu
2 有限会社デライトシステムズ
Delight Systems Co., Ltd.
3 京都大学
Kyoto University
a) zacky@kitakyu-u.ac.jp
b) mori@delightsystems.com
c) delightadmin@delightsystems.com
d) takase@i.kyoto-u.ac.jp

[*1] Zackernel: an Engine for IoT, available at `https://github.com/zackernel/zackernel`

なる dispatch メソッドは，次に呼び出すべき関数をキューから読み込んで呼び出す．dispatch に再入している時にはコールバックする関数を呼び出さない，そうでない場合のみコールバックする関数を呼び出すロジックにすることで，スタックオーバーフローにならないようにしている．

Zackernel はノンプリエンプティブであるので，長い処理を実行すると制御が戻らない．我々は長い処理のほとんどはループかスリープであることに着目した．ループとしては `zLoop`，`zFor`，`zWhile`，`zDoWhile` を用意しており，それぞれ，無限ループ，`for` 文，`while` 文，`do while` 文に対応する．これらはループ 1 回分 (さらには条件文 1 回分) を実行したのちに，実行可能なタスクが他にないかを確認し，もしあればそのタスクに実行権を譲って実行可能待ちに入る．

またスリープとしては `sleep` を用意している．`sleep` は第 1 引数に待ち時間，第 2 引数にコールバック関数を取る．意味としては `sleep` を実行してから第 1 引数に指定した時間を待った後で第 2 引数に与えられたコールバック関数を呼び出す．実現方法としては，まず，`sleep` の待ち行列を形成し，それぞれで次の `sleep` 待ちタスクを起こす時刻を記録しておく．実行可能なタスクがなくなった時点で，次の `sleep` 待ちタスクの起動時刻が，それまでに実行可能なタスクで消費した時間を加味して起動すべき時刻になっていると判定した場合，そのタスクを実行する．起動すべき時刻になっていない場合にはじめて本当にスリープする．

このようにループとスリープを実装することで，ノンプリエンプティブでありながら，実用上問題なく使えるようになった．

Zackernel でのアプリケーションプログラム例を**図 2** に示す．

変数 `led1` と `led2` はメモリマップド I/O で接続された LED であるとする．`main` 関数では，Zackernel を初期化したあと，Zackernel が提供する `fork` によって，関数 `blinkLed1` と `blinkLed2` を並行に呼び出す．

`blinkLed1` 中の `zLoop` は，引数に指定された (匿名) 関数を無限ループさせる．`[&] {}` という記述は C++ 11 で提供される匿名関数であり，外側の関数の名前空間を引き継いだ上で独立した関数を定義する．この中で `led1 = true;` を実行することで LED1 を点灯したあと，Zackernel が提供する `sleep` によって第 1 引数で指定されている 500ms の間スリープした後，第 2 引数で指定する (匿名) 関数を実行する．この中で LED1 を消灯し，500ms の間スリープする．ここまで実行したところで `zLoop` の作用により，再び `led1 = true;` を実行する．

`blinkLed2` も同様であるが，先に LED2 を消灯してから 500ms スリープし LED2 を点灯して 500ms スリープして先頭に戻る．結果として，これらのコードにより，LED1 と LED2 を交互に 500ms おきに点滅させることができる．

```
volatile bool led1 = false;
volatile bool led2 = false;

void blinkLed1() {
  zLoop([&] {
     led1 = true;
     sleep(500, [&] {
       led1 = false;
       sleep(500, [&] {});
     });
  });
}

void blinkLed2() {
  zLoop([&] {
     led2 = false;
     sleep(500, [&] {
       led1 = true;
       sleep(500, [&] {});
     });
  });
}

void main() {
  Zackernel::init();
  fork(blinkLed1, blinkLed2);
}
```

図 2 Zackernel のコード例
Fig. 2 A Sample Code of Zackernel

Zackernel のコード行数は約 900 行であり，すべて C++ で記述されておりアセンブリ言語を一切含まない．プロセスやスレッドでマルチタスクを実現した場合には必ずアセンブリ言語記述を含み機種依存が生じてしまうが，Zackernel ではこの問題は生じない．Zackernel を macOS で静的ライブラリとして構成した場合のサイズは，合計約 134KB である．

4. 軽量コールバックスレッド

我々は，Node プログラミングモデル [5] に着想を得て，Elixir で実装した，コールバック関数を用いてマルチタスク処理にするライブラリである軽量コールバックスレッド[*2]を開発した．現状では Elixir で記述されている．

軽量コールバックスレッドの実装にあたり，Receptor と Worker という 2 つの Elixir プロセスを用意する．Receptor は軽量コールバックスレッドへのリクエストを受取り，すぐさま Worker にコマンドを送った後，すぐに次のリクエストを待つ．こうすることで，遅滞なくリクエストを受け取ることができるようにする．

一方，Worker は Receptor のプロセス ID と実行環境変数 `env` を引数とする関数である．`env` には現状では次の 3 つの情報が入っている．

- `:queue`: 実行キュー，すなわち次以降に実行するコー

[*2] ZeamCallback: ZeamCallback, available at `https://github.com/zeam-vm/zeam_callback`

```
pid = ZeamCallback.Receptor.new
send(pid, {:spawn,
  fn(tid) ->
    IO.puts "foo #{tid}"
  end})
send(pid, {:spawn,
  fn(tid) ->
    IO.puts "bar #{tid}"
  end})
```

図 3　軽量コールバックスレッドのコード例
Fig. 3　A Sample Code of Light-weight Callback Threads

ルバック関数の待ち行列を表すリスト．

- `:threads`: 現在の軽量コールバックスレッド ID とコールバック関数の対応関係を保持するマップ．
- `:next_tid`: 次に軽量コールバックスレッドを新規生成した時の ID を記録する．

Worker は最初に Receptor からのすべてのコマンドを受け取り，実行キュー `env[:queue]` に登録していく．Receptor からのコマンドが空になったら，実行キューからコマンドを 1 つ取り出し，そのコマンドの軽量コールバックスレッドの ID を引数にしてコマンドが指す関数をコールバックして Worker を再帰呼び出しする．実行キューに何もなかった場合は 10ms スリープして，Worker を再帰呼び出しする．

Worker の持つ環境変数は，適宜 Receptor に送られてバックアップされる．将来的には，もし Worker が何らかの理由で無反応になってしまったときには Worker を再起動する機能を実装する予定である．

軽量コールバックスレッドを用いた Elixir プログラム例を**図 3** に示す．`pid = ZeamCallback.Receptor.new` で軽量コールバックスレッドを保持するプロセスを生成し，`pid` にプロセス ID を格納する．続く 2 つの `send(pid, {:spawn, ...})` は，それぞれ 1 つずつ軽量コールバックスレッドを新規作成して起動する．引数で与えられた `fn(tid) ->` から `end` までがコールバックされる匿名関数である．`IO.puts` は文字列を表示する関数である．`tid` には軽量コールバックスレッドの ID が格納される．1 つめの軽量コールバックスレッドでは，`tid` の値が 0 なので，`foo 0` が表示される．2 つめの軽量コールバックスレッドでは `tid` の値が 1 なので，`bar 0` が表示される．これらが順番に実行するようにスケジュールされるので，`foo 0`，`bar 1` と表示される．

軽量コールバックスレッドの基本機能は正味約 150 行で構成される．Elixir のみで記述されており，他言語は用いていない．コンパイルした時のオブジェクトファイルである BEAM コードのサイズは合計約 12KB である．

5. 評価

実験で用いた環境を**表 1** に示す．

表 1　実行環境
Table 1　Runtime Environment

	Mac Pro (Mid 2010)
OS	macOS Sierra 10.12.6
Elixir	1.6.1 (OTP 20.3.6)

表 2　実験結果: Zackernel と C++ 11 スレッドのメモリ消費量の比較 (表)
Table 2　Results: Table of Comparison of Memory Size of Zackernel and C++ 11 Thread

Num. of Tasks	Zackernel	C++ 11 Thread
1	0	536576
10	0	5365760
100	0	53657600
1000	2097152	537624576
2000	2097152	1094123520
5000	4194304	N/A
10000	5242880	N/A
20000	7340032	N/A
50000	10485760	N/A

Zackernel のメモリ消費量を C++ 11 スレッドを用いた場合と比較する評価実験を行なった．C 言語による macOS でのメモリ消費量の測定には `mach/mach.h` に定義している `task_basic_info` 構造体を用いて `virtual_size` で示される仮想メモリサイズを用いた．この値はバイト単位で表されるが，実際にはページ単位でメモリを扱うので，ページサイズの倍数値になる．そのため，ページサイズより小さなメモリ量の変化は測定結果に現れない．また，何回か実行するとばらつきを持った測定値となったので，繰返し測定して最初に 3 回同じ測定値になった場合の値を採用した．作成するスレッドの数は 1,10,100,1000,2000,5000,10000,20000,50000 の 9 通りで測定した．これらの測定結果を最小 2 乗法で傾きを求めることで，1 スレッドあたりの使用メモリ量を測定した．

Zackernel と C++ 11 スレッドのメモリ消費量を比較した実験結果を**表 2** に示す．それぞれの相関係数は，Zackernel で 0.8396，C++ 11 スレッドで 0.99993 であった．C++ 11 スレッドでは良好な結果が得られたと判断できるが，Zackernel ではスレッド数が小さい場合にページサイズより小さなメモリ変化しかなかったので測定値に現れず，結果的に相関係数の絶対値が小さくなった．Zackernel では 1 スレッドあたり 204 バイト，C++ 11 スレッドでは 1 スレッドあたり約 546KB 消費していることがわかった．すなわち，Zackernel は C++ 11 スレッドの約 2,700 分の 1 のメモリ消費量である．

また軽量コールバックスレッドのメモリ消費量を，Elixir プロセスを用いた場合と比較する評価実験を行なった．Elixir にはメモリ消費量を測定するために `:erlang.memory` という関数が用意されている．この関数は全体のメモリ量

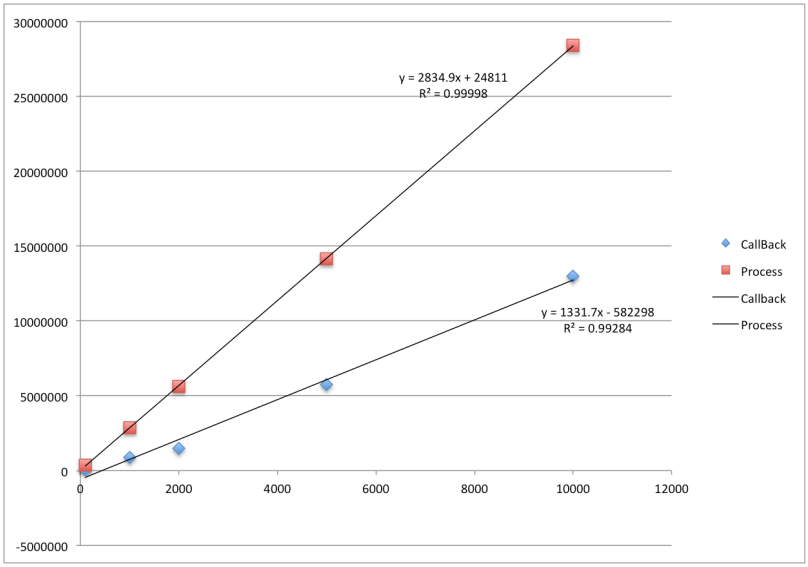

図 4 実験結果: 軽量コールバックスレッドとプロセスのメモリ消費量の比較 (散布図)

Fig. 4 Results: Scatterplot of Comparison of Memory Size of Light-weight Callback Threads and Processes

表 3 実験結果: 軽量コールバックスレッドとプロセスのメモリ消費量の比較 (表)

Table 3 Results: Table of Comparison of Memory Size of Light-weight Callback Threads and Processes

Num. of Tasks	CallBack (bytes)	Process (bytes)
100	76144	358848
1000	881656	2877360
2000	1496296	5636232
5000	5734120	14160616
10000	13004640	28402248

やプロセスで使用しているメモリ量などを集計することができる．本研究では，軽量コールバックスレッドもしくは Elixir プロセスを一定数新規作成する前後の全体のメモリ量の変化を計測した．作成するスレッドもしくはプロセスの数は 100, 1000, 2000, 5000, 10000 の 5 通りで測定した．これらの測定結果を最小 2 乗法で傾きを求めることで，1 スレッドもしくは 1 プロセスあたりの使用メモリ量を測定した．

軽量コールバックスレッドと Elixir プロセスのメモリ消費量を比較した実験結果を**表 3** と**図 4** に示す．それぞれの相関係数は軽量コールバックスレッドの場合で 0.99284，Elixir プロセスの場合で 0.99998 であったので，実験結果は良好であったと考えられる．傾きより，軽量コールバックスレッドの場合で 1 スレッドあたり 1332 バイト，Elixir プロセスの場合で 1 プロセスあたり 2835 バイト消費していることがわかった．すなわち，軽量コールバックスレッドはプロセスの約半分のメモリ消費量である．

これらの 1 スレッド/プロセスあたりのメモリ消費量を比較すると，Zackernel が 204 バイトと最も少なく，軽量コールバックスレッドが 1332 バイト，Elixir プロセスが 2835 バイトと続き，C++ 11 スレッドが約 546KB となる．処理系レベルから設計を見直すことで，軽量コールバックスレッドを Zackernel 並みのメモリ消費量に抑える最適化を施せる余地があるのかもしれない．

6. まとめと将来課題

本研究では Node プログラミングモデルと同様の機構でマルチタスク処理をするを C++ と Elixir で実装し，それぞれ Zackernel，軽量コールバックスレッドと呼称した．

これらと従来のプロセス/スレッドとの 1 プロセス/スレッドあたりのメモリ消費量を比較する評価実験を行なった．その結果，Zackernel は 1 スレッドあたり 204 バイトと最も少なく，軽量コールバックスレッドが 1 スレッドあたり 1332 バイト，Elixir プロセスが 1 プロセスあたり 2835 バイト，C++ 11 スレッドが 1 スレッドあたり約 546KB であった．

将来課題としては，まずそれぞれの方法でコンテキストスイッチにどのくらいの時間を要するのかを測定することが挙げられる．コンテキストスイッチに要する時間は，マルチプロセス／マルチスレッドのシステム実装を評価する際に，1 スレッド/プロセスあたりのメモリ消費量と並んで重要な特性値である．今後，環境を整備して測定を試みたい．

本提案方式は，現状ではノンプリエンプティブであるため，利用できる状況が限られる．Zackernel ではプリエンプティブマルチタスクと同様の使い勝手にするために，ループ中の 1 回 1 回の繰返しの際に他のタスクが起動可能かを判定するロジックを実装した．この方式をより使いやすくするために，プログラミング言語処理系に手を入れて，プ

ログラム中のループにフックを挿入する事を考えている．また，Elixir においても，ループの代わりに用いられる再帰呼び出しや，Enum や Flow [7] を用いた map 計算のようなまとまった処理を実行する際に，Zackernel と同様の仕組みを入れる方式が考えられる．

Zackernel はトリッキーなプログラミングになっていることから保守性が悪く，メモリリークの問題がまだ多く残っている．メモリリークが起こっている理由としては，通常の C++ プログラム同様，ガーベジコレクションされないという要因もありうるが，そのほかにも，Elixir のような関数型言語ではなく C++ で実装しているため，末尾再帰の最適化がなされないことから，コールバックする際に再帰呼び出しが深くなってしまう要因もありうる．

軽量コールバックスレッドについては，現状では実行キューで保持されているメモリ領域を適切に解放していないので，GC が有効に機能しない問題があるので，改善したい．プロセス間通信の仕組みを実装することで，ようやく実用的なプログラムを書くことができる．もしそれが実現できた時には，軽量コールバックスレッドを用いるように Phoenix を実装しなおすことを考えている [8]．また，Zackernel 並みに 1 スレッドあたりのメモリ消費量を抑えるような言語処理系の設計・実装を試みる．

参考文献

[1] Fedrecheski, G., Costa, L. C. P. and Zuffo, M. K.: Elixir programming language evaluation for IoT, *2016 IEEE International Symposium on Consumer Electronics (ISCE)*, pp. 105–106 (online), DOI: 10.1109/ISCE.2016.7797392 (2016).

[2] Foundation, N.: Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. (2009). `https://nodejs.org/`.

[3] McCune, R. R.: Node.js Paradigms and Benchmarks (2011). STRIEGEL, GRAD OS F' 11, PROJECT DRAFT.

[4] Provos, N. and Mathewson, N.: libevent: an event notification library (2005). `http://libevent.org`.

[5] Tilkov, S. and Vinoski, S.: Node.js: Using JavaScript to Build High-Performance Network Programs, *IEEE Internet Computing*, Vol. 14, No. 6, pp. 80–83 (online), DOI: 10.1109/MIC.2010.145 (2010).

[6] Valim, J.: Elixir: Elixir is a dynamic, functional language designed for building scalable and maintainable applications. (2013). `https://elixir-lang.org`.

[7] Valim, J.: Flow: Computational parallel flows on top of GenStage (2017). `https://github.com/elixir-lang/flow`.

[8] 山崎　進，森　正和，上野嘉大，高瀬英希：Elixir の軽量コールバックスレッドの実装と Phoenix の同時セッション最大数・レイテンシ改善の構想，第 2 回 Web System Architecture 研究会, 福岡 (2018). The paper and the presenation are available at `https://zeam-vm.github.io/papers/callback-thread-2nd-WSA.html` and `https://zeam-vm.github.io/zeam-WSA-20180512/#/`, respectively.