

[C++11:スレッド・ライブラリひとめぐり](#)

C++11:スレッド・ライブラリひとめぐり【補足編:3】

C++

[WEB用を表示](#)

ツイート { 31

1

{ 24

G+

[επιτομή](#) [著]

2018/03/05 14:00

ダウンロード ↓ [サンプルファイル \(192.3 KB\)](#)

こんなはずじゃなかったんです。かなり昔に書いたスレッド・ライブラリの紹介があまりにざっくりしてたので、つけ足しのつもりで筆を進めてて気がつけば三部作になっちゃったという。標準C++ライブラリの一つthread support libraryの(補足のハズだった)解説、おしまいはスレッド間の同期をサポートしてくれるcondition_variableのおはなし。

```
C:\WINDOWS\system32\cmd.exe
100000 items produced. sum= 149
100000 items produced. sum= 150
produced sum = 299715
96791 items consumed. sum= 1446
103209 items consumed. sum= 155
consumed sum = 299715
続行するには何かキーを押してください
```

- 以前の記事『[C++11:スレッド・ライブラリひとめぐり](#)』
- 第1回『[C++11:スレッド・ライブラリひとめぐり【補足編:1】](#)』
- 第2回『[C++11:スレッド・ライブラリひとめぐり【補足編:2】](#)』

手分けのしかたは二通り

「マルチスレッドで速くする」とは、つまるところ同時に動ける複数のスレッドが仕事のかたまりを手分けして処理することで速くしてるわけですね。このとき「手分けのしかた」は大きく2つに分類できます。

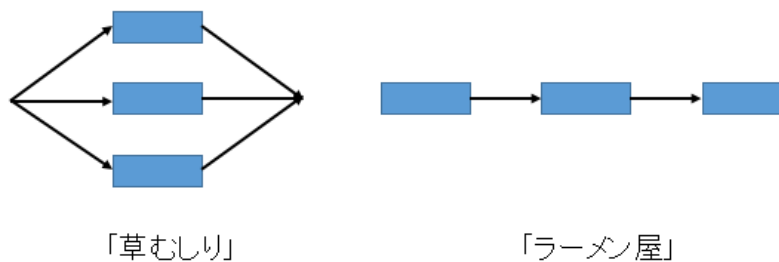


fig01

1つは庭の草むしり。1人でやるのはしんどいけれど、庭をN等分してN人が一斉に草をむしれば1人あたりのノルマは1/Nとなり、スピードはざっくりN倍になります。アレとコレをやらなきゃいけないけれど、アレとコレが独立してるなら2つのスレッドで同時にやれば早く片付きます。

大量の要素が並んだ配列の総和を求める(あんまり面白くない)サンプルを書いてみました。要素数Nの配列:vector<double> data(N)の総和を求めるのに2つのスレッドを起こし、それぞれ配列の前半部と後半部の和を求めます。両者が処理を完了したら、得られた2つの和を足せば総和が求まります。

list01

```
#include <iostream>
#include <random>
#include <thread>
#include <future>
#include <numeric>
#include <vector>
#include <chrono>

int main() {
    using namespace std;
    using namespace std::chrono;

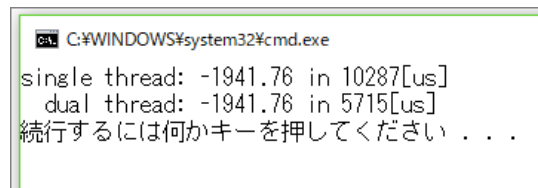
    const int N = 10000000;
    vector<double> data(N);

    mt19937 gen; // メルセンヌ・ツイスター
    normal_distribution<double> dist; // 正規分布(平均:0,標準偏差:1)
    auto rand = [&]() { return dist(gen); };
    generate_n(begin(data), N, rand);

    double sum;
    long long duration;
    high_resolution_clock::time_point start;
    high_resolution_clock::time_point stop;

    // single-thread
    start = high_resolution_clock::now();
    sum = accumulate(begin(data), end(data), 0.0);
    stop = high_resolution_clock::now();
    duration = duration_cast<microseconds>(stop - start).count();
    cout << "single thread: " << sum << " in " << duration << "[us]\n";

    // 2-threads
    start = high_resolution_clock::now();
    // forkして
    future<double> f0 = async([&]() { return accumulate(begin(data), begin(data)+N/2, 0.0); });
    future<double> f1 = async([&]() { return accumulate(begin(data)+N/2, begin(data)+N, 0.0); });
    // joinする
    sum = f0.get() + f1.get();
    stop = high_resolution_clock::now();
    duration = duration_cast<microseconds>(stop - start).count();
    cout << " dual thread: " << sum << " in " << duration << "[us]\n";
}
```



```
C:\WINDOWS\system32\cmd.exe
single thread: -1941.76 in 10287[us]
dual thread: -1941.76 in 5715[us]
続行するには何かキーを押してください . . .
```

fig02

2倍とまではいかないけれど、まあいいセンいってますかね。

もう一つはラーメン屋。とあるラーメン屋に電話で注文が入ってきます。電話が鳴るたびに注文を受け(入力)、ラーメンこしらえて(処理)、バイク飛ばして届けます(出力)。それぞれに5分かかるとすると、店主1人で切り盛りする"ワンオペ"だったら入力から出力まで15分、次の注文に応じられるのは15分後です。

バイトを雇って3人で店を回すなら、3人がそれぞれ入力／処理／出力を担当し、入力から処理へ／処理から出力へと仕事が流れていきます。お客様からすれば電話してから15分でラーメンが届くことに変わりはないけれど、注文を受けてから5分後には次の注文が受けられるんだから処理能力は3倍です。

condition_variableはラーメン屋の流れ作業を組み立てるのに不可欠なんです。マクラが長くてごめんなさいね。

生産者と消費者

そんなわけで、スレッド間のデータの受け渡しを考えます。データを生産し送り出すスレッドをproducer(生産者)、データを受け取って消費するスレッドをconsumer(消費者)と名付けておきます。

producerからconsumerへデータを受け渡さなきゃなりません。手始めにこんなコードではいかがでしょ。

list02

```
/*
    producer-consumer間のデータ受け渡しのための'ハコ'
*/
template<typename T>
class box {
private:
    T value_;    // ハコのナカミ
public:
    box() {}
    box(const T& value) : value_(value) {}
    void set(const T& value) { value_ = value; } // 送出
    const T& get() const    { return value_; } // 受取
};

#include <iostream>
#include <random>
#include <string>

void produce(box<int>& out) {
    using namespace std;
    const int N = 100000;
    random_device gen;
    uniform_int_distribution<int> dist(0,3); // 0~3のランダムな整数
    auto rand = [&]() { return dist(gen); };
    int sum = 0;
    for ( int i = 0; i < N; ++i ) {
        int value = rand();
        out.set(value); // 送出
        sum += value;
    }
    cout << to_string(N) + " items produced. sum= " + to_string(sum) + "\n";
}

void consume(box<int>& in) {
    using namespace std;
    int count = 0;
    int sum = 0;
    while ( true ) {
        int value = in.get(); // 受取
        if ( value < 0 ) break; // ここでloopを抜ける
        ++count;
        sum += value;
    }
    cout << to_string(count) + " items consumed. sum= " + to_string(sum) + "\n";
}
```

```

}

#include <thread>
#include <utility>

int main() {
    box<int> b;
    std::thread producer(produce, std::ref(b));
    std::thread consumer(consume, std::ref(b));
    producer.join();
    b.set(-1); // consumerを停止させるため
    consumer.join();
}

```

2つの関数produce()とconsume()を同時に起動すれば、やがて双方から同じ結果が出力される.....わけがない。

producerはboxに送出したデータをconsumerが受け取ってくれたか確認もせぬまま、お構いなしに次のデータを送出してますからね。consumerも同罪です。送出されたか確認もせずに受け取ってます。

producerとconsumerが互いに同期、つまりタイミングを合わせて送出／受取を行わないとconsumerに届くデータは消失するわ重複するわでマトモな結果は得られません。producerは「読んでいいよ」、consumerは「書いていいよ」を相手に通知し、そしてproducerは「書いていいよ」、consumerは「読んでいいよ」の通知を待つ機構が必要です。

ならばこれではどうだろう.....。

list03

```

template<typename T>
class box {
private:
    T value_;    // ハコの中カミ
    bool empty_; // 空か?
public:
    box() : empty_(true) {}
    box(const T& value) : value_(value), empty_(false) {}
    bool empty() const { return empty_; } // 空ならtrue
    bool occupied() const { return !empty_; } // 空でないならtrue
    void set(const T& value) { empty_ = false; value_ = value; } // 送出:空じゃなくなる
    const T& get()          { empty_ = true; return value_; } // 受取:空になる
};

```

```

#include <thread>

```

```

template<typename T>
class concurrent_box {
private:
    box<T> box_;    // ハコの中カミ
public:
    concurrent_box() : box_() {}
    concurrent_box(const T& value) : box_(value) {}

    void set(const T& value) {
        // 空じゃない間待つ
        while ( box_.occupied() ) { std::this_thread::yield(); }
        box_.set(value);
    }
}

```

```

const T& get() {
    // 空である間待つ
    while ( box_.empty() ) { std::this_thread::yield(); }
    return box_.get();
}

};

void produce(concurrent_box<int>& out) {
    ...
    out.set(データ);
    ...
}

void consume(concurrent_box<int>& in) {
    ...
    データ = in.get();
    ...
}

```

boxに空か否かを教えてくれるempty()、occupied()を用意し、boxを内包するconcurrent_boxは、set()内で空になるまでカラ回り、get()内で空じゃなくなるまでカラ回りさせてます。

カラ回りloop内のstd::this_thread::yield()を説明しておきます。スレッドには大きくWAIT／READY／RUNの3状態がありますよね。yield()はRUN状態にある現スレッド(this_thread)をいったんREADYに落とすことで他のREADYなスレッドにCPU(コア)を譲ります。条件が満たされるまで空loopをぶん回すと他のスレッドがなかなかRUN状態になれないのでいったんCPUを明け渡すんです。

感心しない実装ですねー。このloopは単に待つだけのためにアクセルべた踏みで空転させてる。電気代の無駄遣いです。本来なら条件が満たされるまでやることないんだからWAITしててほしいのですよ。

それともう一つ。このプログラム何度か動かすと、たまに結果が狂います。

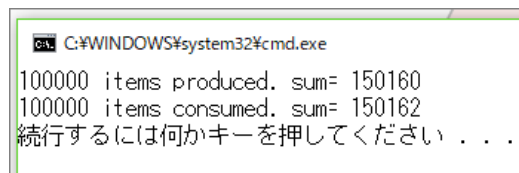


fig03

concurrent_boxに対して複数のスレッドが同時にset()／get()することがあるんだから、このままじゃdata race起こすんですよ。set()／get()できるスレッドは高々1つとなるようガードせにやらんです。concurrent_box内にstd::mutexを置いて、

list04

```

template<typename T>
class concurrent_box {
private:
    box<T> box_;    // ハコのナカミ
    std::mutex mtx_;
public:
    concurrent_box() : box_() {}
    concurrent_box(const T& value) : box_(value) {}

    void set(const T& value) {
        std::unique_lock<std::mutex> guard(mtx_); // set中は邪魔するな
        // 空じゃない間待つ
        while ( box_.occupied() ) { std::this_thread::yield(); }
    }
}

```

```

    box_.set(value);
}

const T& get() {
    std::unique_lock<std::mutex> guard(mtx_); // get中は邪魔するな
    // 空である間待つ
    while ( box_.empty() ) { std::this_thread::yield(); }
    return box_.get();
}
};

```

残念でした、これでもやっぱりダメ。てかむしろ改悪なんですわ。get()はデータが入ってくるのを待ってます。ってことは、他のスレッドがset()してくれるのを待っているってことです。ところがget()／set()は1つのmutexでガードされているのでget()が終わらぬ限りset()されません。鍵かけた空箱にバナナが入るのをひたすら待ってる哀れなサル状態、しかも肝心の鍵はサルが握ってるというdeadlock状態。これを解消するには、データが入ってくるのを待ってる間はmutexのガードを解いておかねばならんです。バナナが欲しけりや鍵開けとけと。

やれやれ、ようやく条件変数:condition_variableを語る準備ができました。マクラが長くてごめんなさいね(二度目)。

condition_variable

std::condition_variableに定義された主要メンバ関数は4つあり、そのうち2つは待つ側用、

- **void wait(unique_lock<mutex>& guard);**
guard.unlock()と同時に自分自身をブロックして待ち状態となり、ブロックが解けると同時にguard.lock()して戻ってきます。ブロック中はguardを解いてくれるのね。
- **template<class Predicate> void wait(unique_lock<mutex>& guard, Predicate pred);**
predは引数がなくてboolを返す関数オブジェクトで、待ちが解ける条件です。こいつのナカミは while (!pred()) { wait(guard); } つまり条件を満たさぬ間、ひたすらwait(guard) します。

wait()によるブロックを解く通知側にも2つ、

- **void notify_one();**
wait()で待ってるスレッドのうち、どれか1つのブロックが解けます。どのスレッドが動けるようになるかは運次第。
- **void notify_all();**
wait()で待ってる全スレッドのブロックが解けます。

condition_variableを使った"正しく動く"concurrent_boxはこんな実装になります。

list05

```

template<typename T>
class concurrent_box {
private:
    box<T> box_;    // ハコのナカミ
    std::mutex mtx_;
    std::condition_variable can_get_; // 条件変数:getできる
    std::condition_variable can_set_; // 条件変数:setできる
public:
    concurrent_box() : box_() {}
    concurrent_box(const T& value) : box_(value) {}

    void set(const T& value) {
        std::unique_lock<std::mutex> guard(mtx_);
        // 'setできるよ' を待つ
        can_set_.wait(guard, [this]() { return box_.empty(); });
        box_.set(value);
        // 'getできるよ' を通知
        can_get_.notify_one();
    }
}

```

```

T get() {
    std::unique_lock<std::mutex> guard(mtx_);
    // 'getできるよ' を待つ
    can_get_.wait(guard, [this]() { return box_.occupied(); });
    T value = box_.get();
    // 'setできるよ' を通知
    can_set_.notify_one();
    return value;
}
};

```

condition_variableを2つ使って、

- **get()**:データが入ってくるのを待つ／空になったことを通知する
- **set()**:空になるのを待つ／データが入ったことを通知する

ことでproducerとconsumerの同期を実現しています。"ハンドシェイク"と呼ばれる同期ですな。

```

cmd. C:\WINDOWS\system32\cmd.exe
100000 items produced. sum= 149515
100000 items produced. sum= 150200
produced sum = 299715
96791 items consumed. sum= 144609
103209 items consumed. sum= 155106
consumed sum = 299715
続行するには何かキーを押してください . . .

```

fig04

2つのproducerと2つのconsumerが1つのconcurrent_boxを介してデータを受け渡しています、電話回線2本で調理場に2人いるラーメン屋の様子です。注文を取りこぼすことなく処理できてます。

調子こいてアレンジしてみましょう。concurrent_boxは受け渡せるデータが1個だけなのでconsumerがモタつく(getが遅れる)とproducerがそれに引きずられてモタつきます。ハコを改め待ち行列にすればconsumerが多少モタついてもproducerはデータを送出できます。待ち行列:concurrent_queueの実装はこんなカンジ。

list06

```

template<typename T>
class concurrent_queue {
public:
    typedef typename std::queue<T>::size_type size_type;
private:
    std::queue<T> queue_;
    size_type capacity_; // 待ち行列の最大長

    std::mutex mtx_;
    std::condition_variable can_pop_;
    std::condition_variable can_push_;
public:
    concurrent_queue(size_type capacity) : capacity_(capacity) {
        if ( capacity_ == 0 ) {
            throw std::invalid_argument("capacity cannot be zero.");
        }
    }

    void push(const T& value) {

```

```

std::unique_lock<std::mutex> guard(mtx_);
can_push_.wait(guard, [this]() { return queue_.size() < capacity_; });
queue_.push(value);
can_pop_.notify_one();
}

T pop() {
    std::unique_lock<std::mutex> guard(mtx_);
    can_pop_.wait(guard, [this]() { return !queue_.empty(); });
    T value = queue_.front();
    queue_.pop();
    can_push_.notify_one();
    return value;
}
};

```

複数のwait()を解くnotify_all()を使ったサンプルは[元記事で紹介済](#)なので割愛御免。

あとがきにかえてオマケ:POSIX Threads for Windows

thread support libraryの下調べの最中、LinuxではおなじみのpthreadをWindowsにportしたライブラリ「[pthreads4w \(POSIX Threads for Windows\)](#)」を見つけた。

スレッド周りはLinuxとWindowsとで大きな違いがあるためにLinuxからWindowsへのコードの移植は少なからず面倒で、かくのごとくデリケートなスレッドでバグと目も当てられない悲惨な状況に追い込まれます。コードをいじらずほとんどそのままコンパイルできるpthreads4wはかなり重宝しています。pthreadでconcurrent_boxを作ってみました。[サンプルファイル](#)に入ってますよ。

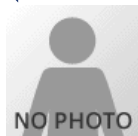
[バックナンバー](#)
[WEB用を表示](#)
[ツイート](#)
[31](#)
[1](#)
[24](#)
[G+](#)

PR [『マンガで分かるプログラミング用語辞典』プログラミング入門書の副教材としてぜひ](#)

PR [『Scott Guthrie氏 Blog翻訳』マイクロソフトの最新技術動向はここでチェック](#)

PR [『C#で始めるテスト駆動開発入門』C#でのTDD実践方法をステップバイステップで紹介](#)

著者プロフィール



επιστημη (エピステーメー)

C++に首まで浸かったプログラマ。Microsoft MVP, Visual C++ (2004.01～) だったり わんくま同盟でたまにセッションスピーカやったり 中国茶淹れてにわか茶人を気取ったり、あと Facebook とか。著書: - STL標準講座 (監修) -...

※プロフィールは、執筆時点、または直近の記事の寄稿時点での内容です
Article copyright © 2018 episteme, Shoeisha Co., Ltd.

[ページトップへ](#)

CodeZineについて

[各種RSSを配信中](#)



プログラミングに役立つソースコードと解説記事が満載な開発者のための実装系Webマガジンです。
掲載記事、写真、イラストの無断転載を禁じます。
記載されているロゴ、システム名、製品名は各社及び商標権者の登録商標あるいは商標です。

SE
SHOEISHA

ライオン
10190846(05)

[ヘルプ](#)
[広告掲載のご案内](#)
[著作権・リンク](#)
[スタッフ募集!](#)
[メンバー情報管理](#)
[メールバックナンバー](#)
[IT人材](#)
[教育ICT](#)
[マネー・投資](#)
[プロジェクトマネジメント](#)
[書籍・ソフトを買う](#)
[電験3種対策講座](#)

| | | | |
|----------------------|--------------------------|--------------------------|--------------------------|
| 免責事項 | マーケティング | ネット通販 | 電験3種ネット |
| 会社概要 | エンタープライズ | イノベーション | 第二種電気工事士 |
| | | ホワイトペーパー | |

 [メンバーメニュー](#) |  [ログアウト](#)