

# VC++まわりの非同期処理

～VC++とC++/CXについて～

CommunityOpenDay2014

2014/3/22 Sat

Room metro大阪 遥佐保

はじめに

# 自己紹介

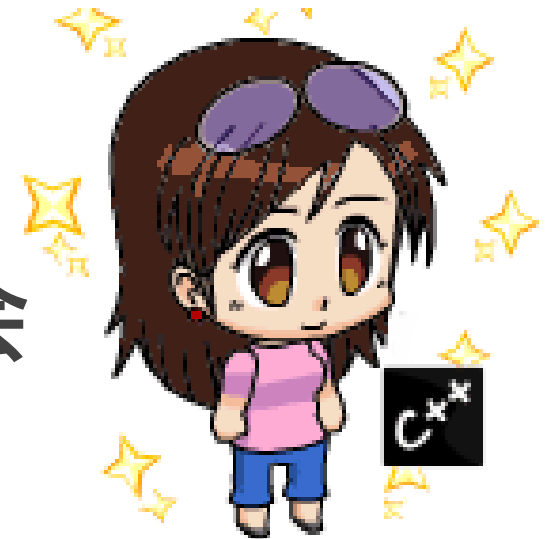
@hr\_sao

Microsoft MVP for Client Development



出没コミュニティ

- Room metro
- C++テンプレート完全ガイド読書会



# 本日の目的

VisualStudio2013 特にVC++の  
非同期処理関連について  
理解度を深める

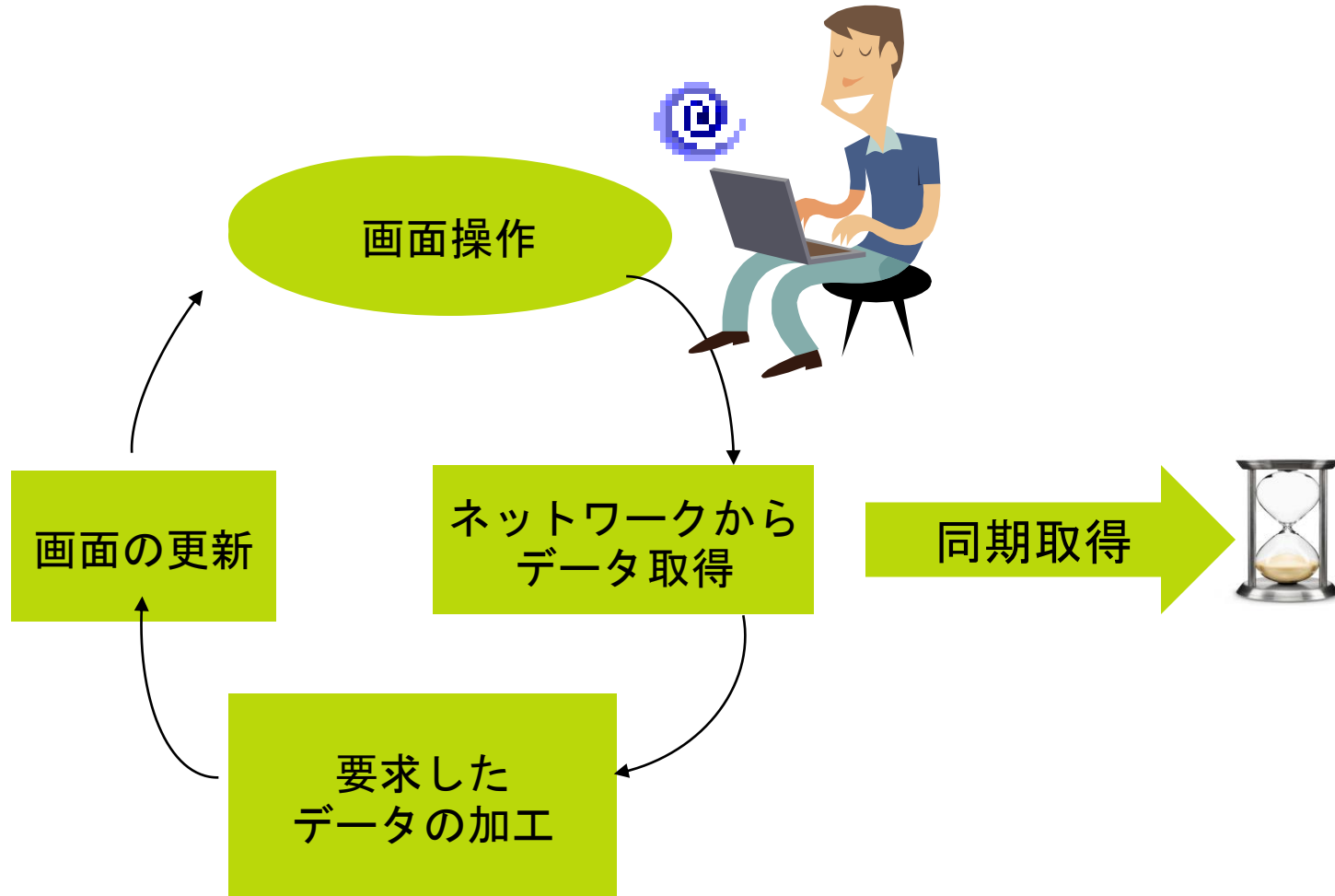
# Topics

1. 非同期って何よ?
2. プロセス/スレッド
3. VC++非同期処理
4. C++/CX非同期処理
5. まとめ
6. おまけ

# 1. 非同期って何よ?

# 待つか？待たないか？ - 待つ時

## Webサイトへのアクセス



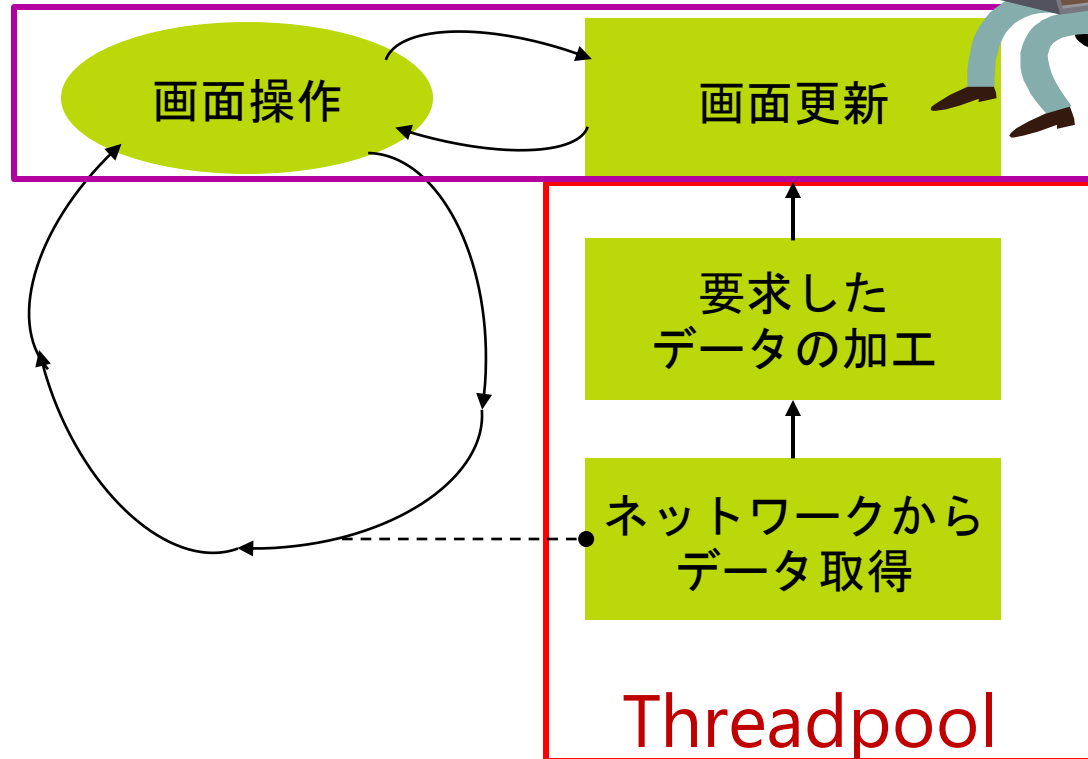
1. 画面操作

2. データ取得

3. (2.の取得を待ってから)  
画面の更新

# 待つか？待たないか？ - 待たない時

(例) イベントドリブン + 処理キュー (Ajaxなど)



1. 画面操作

2. 画面更新

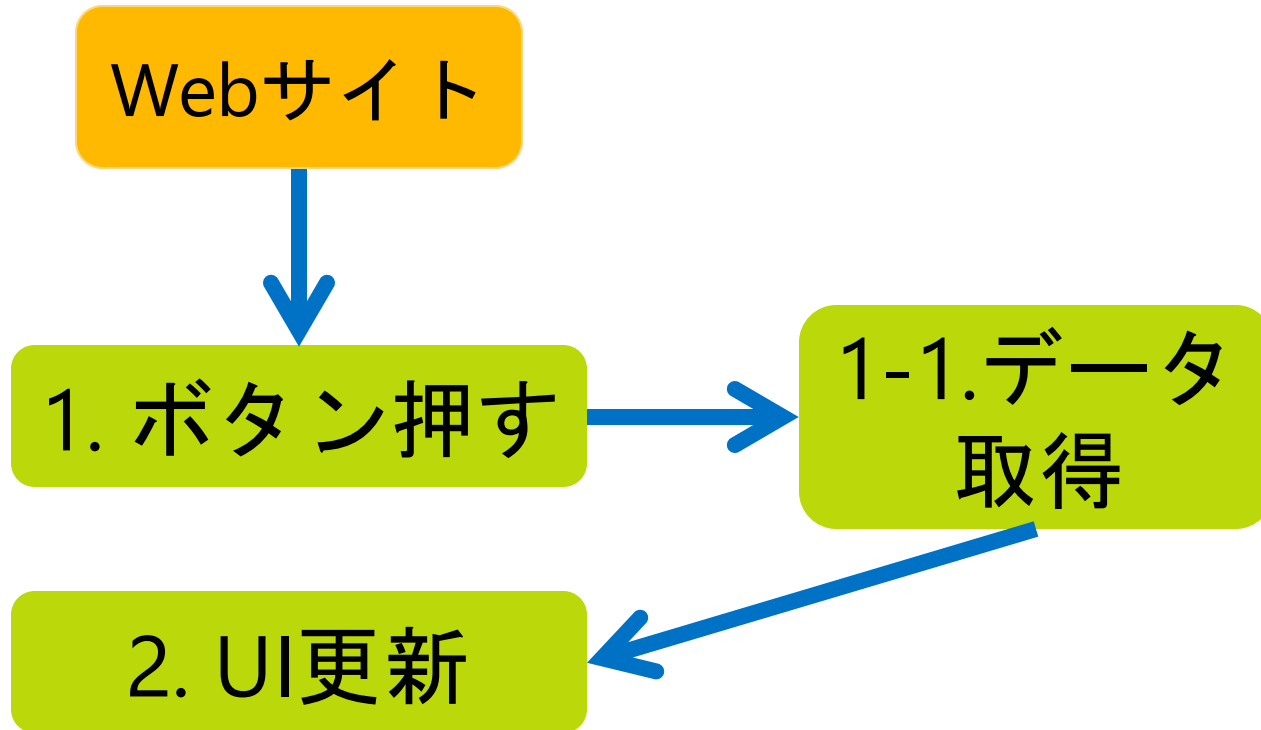
(何も待っていない)



# 同期処理/非同期処理について

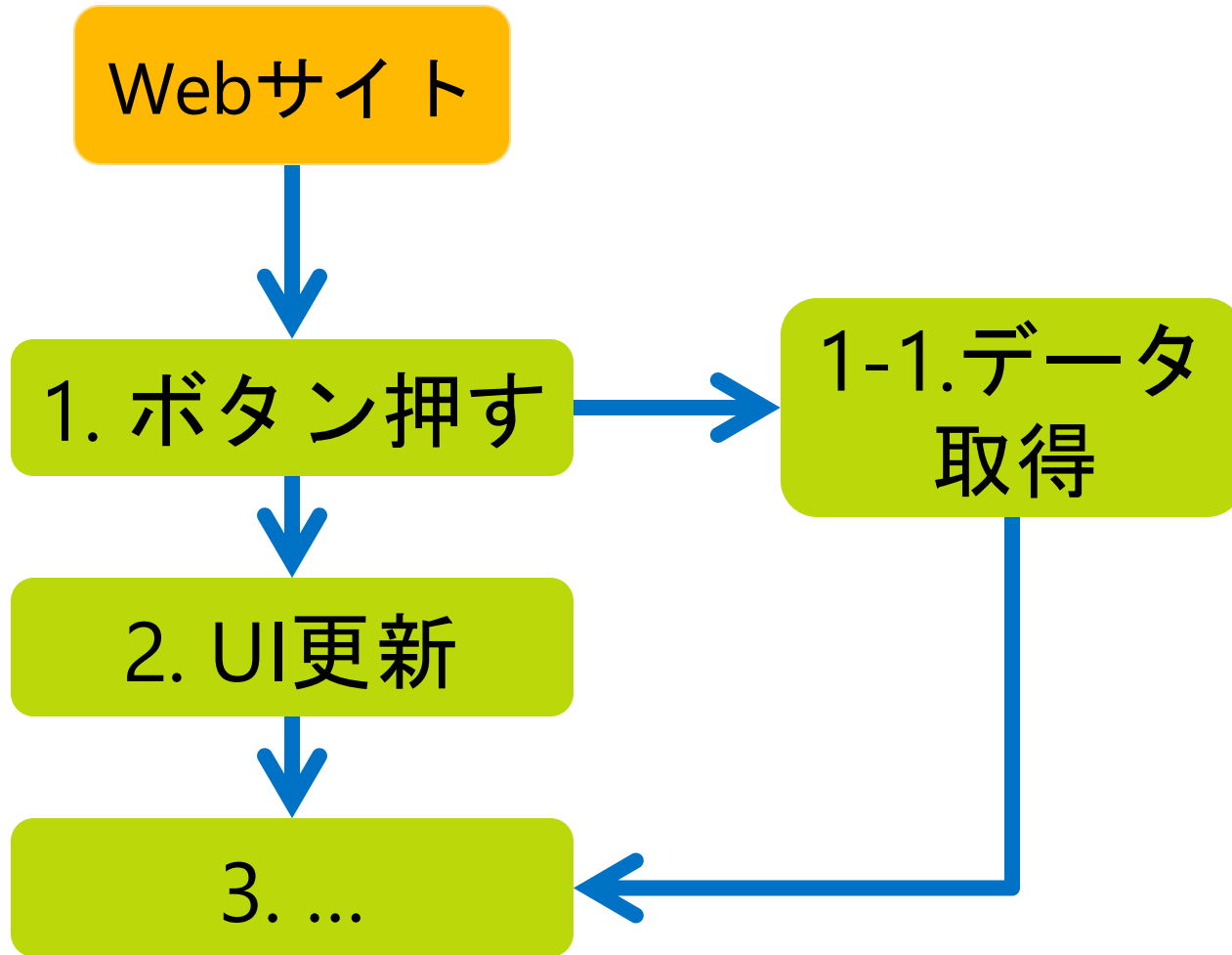


# 同期処理 / シングルスレッド



直前の処理が  
終わってから  
次の処理へ

# 非同期処理



1-1 の処理の  
終了を待たずに  
2 の処理を行う

# マルチスレッド

行列計算

前の処理を待たずに次へ  
1-1と1-2は同時に処理

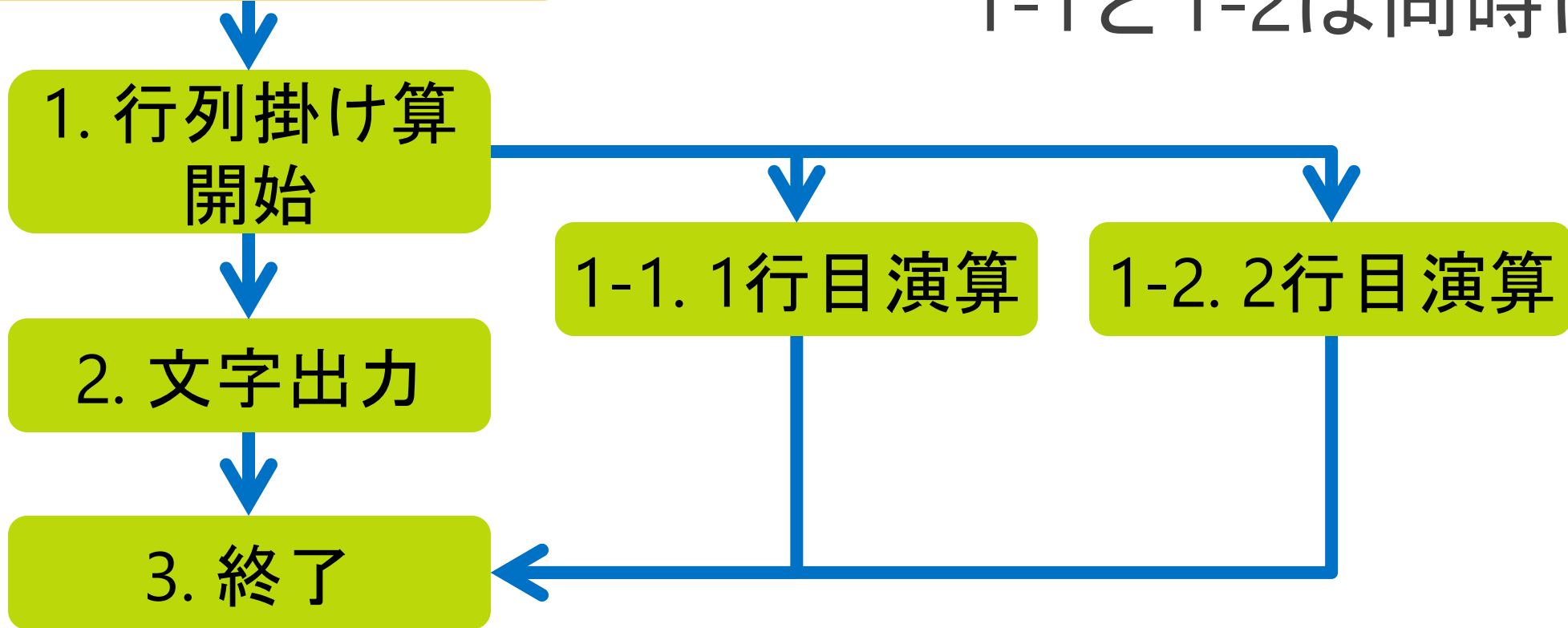
1. 行列掛け算  
開始

2. 文字出力

3. 終了

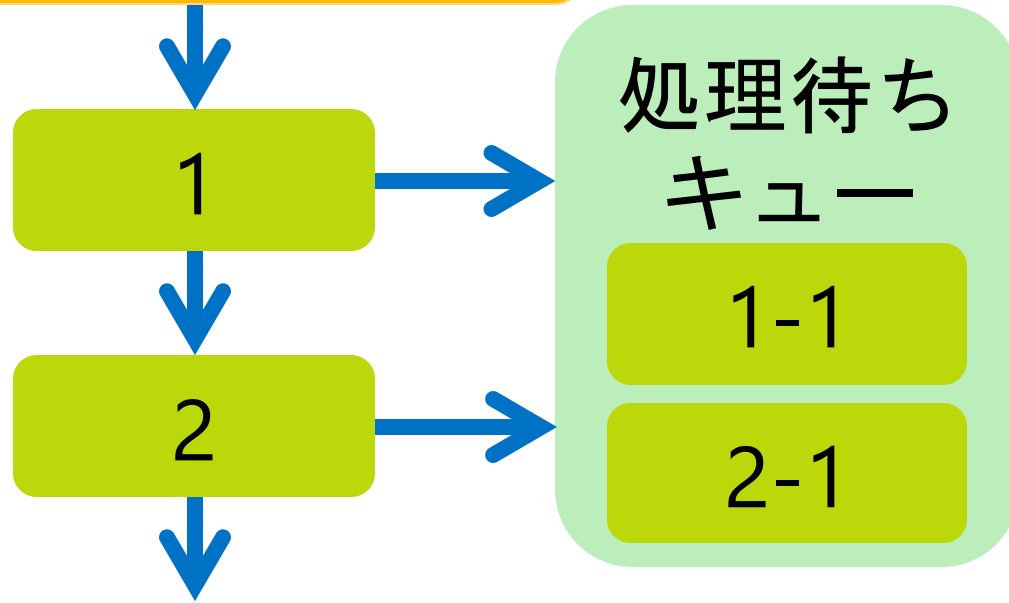
1-1. 1行目演算

1-2. 2行目演算



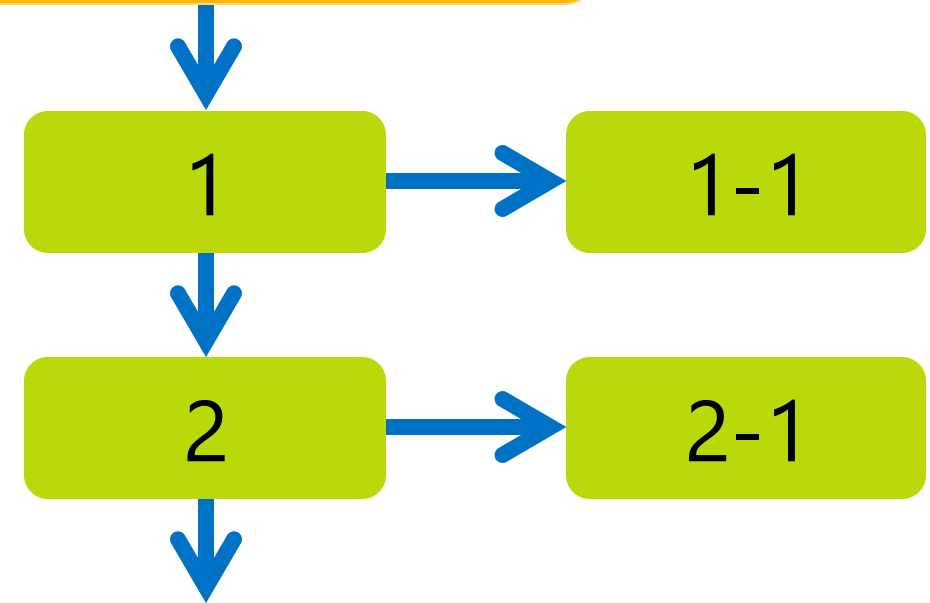
# 非同期処理とマルチスレッドの概念

非同期処理



前の処理を待たずに次へ  
(同時処理かどうかは  
無関係)

マルチスレッド



前の処理を待たずに次へ  
1と1-1は同時に処理

あれ？

わたしは「**非同期処理**」を調べてた  
待たないだけじゃないの？

「**スレッド**」って何でしょうか？

## 2. プロセス/スレッド

非同期処理を理解するために、  
遠回りに見えますが  
プロセスやスレッドの仕組みについて  
紹介します  
(例はWindowsOS)



# プロセスについて



# プロセス (1)

→ アプリの実行単位

CPU/メモリなどリソース割り当ての単位

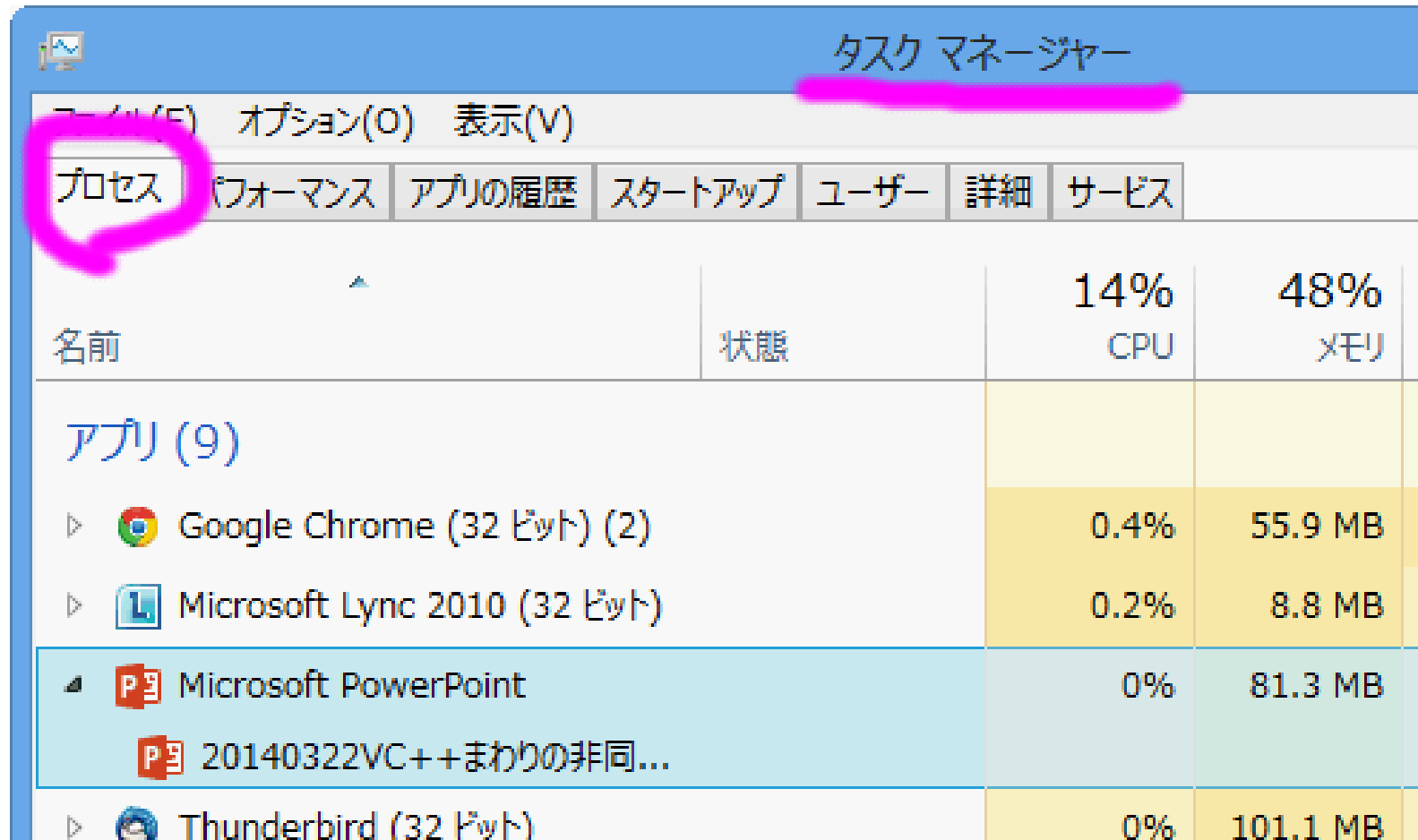
(≒ タスク by Windows)

Windowsは

- マルチプロセス
- マルチタスク

などで表現されている

ことが多い



名前	状態	CPU	メモリ
アプリ (9)			
▷ Google Chrome (32 ビット) (2)		0.4%	55.9 MB
▷ Microsoft Lync 2010 (32 ビット)		0.2%	8.8 MB
◀ Microsoft PowerPoint		0%	81.3 MB
▷ 20140322VC++まわりの非同...			
▷ Thunderbird (32 ビット)		0%	101.1 MB

# プロセス (2)

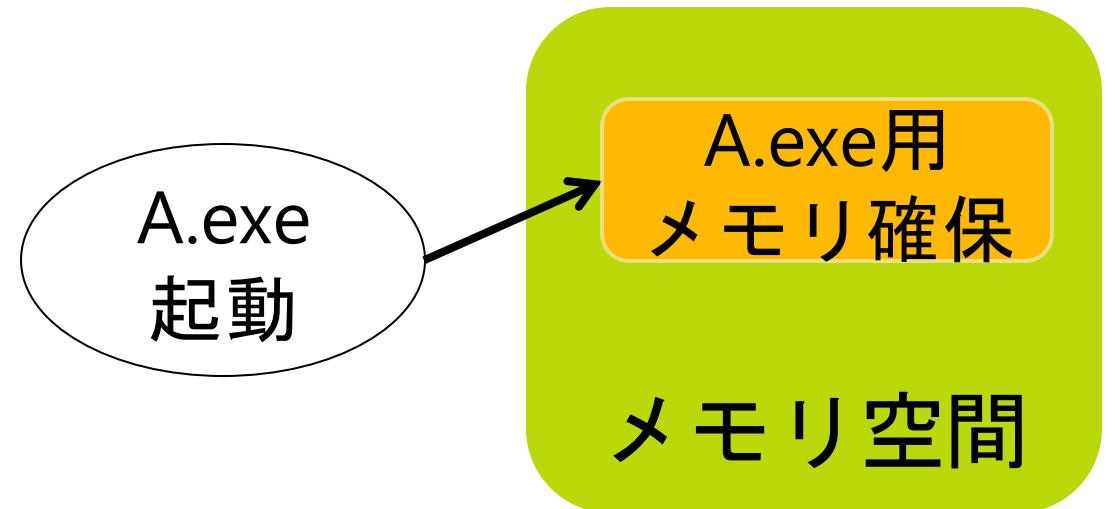
アプリケーション実行時のプロセスの動き

- 実行する前に、メモリ内に領域を確保
- プロセスが持つもの（アプリの管理情報）

実行イメージ

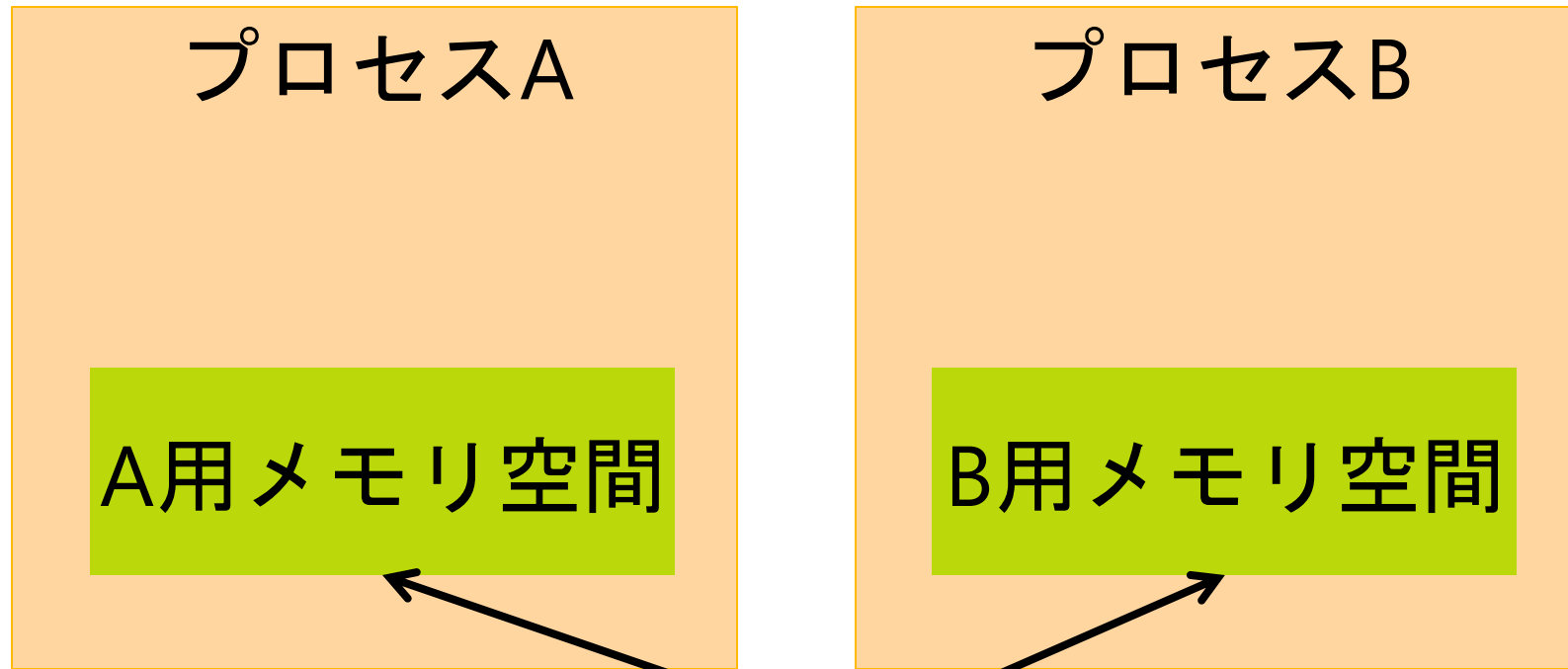
アプリが使うメモリ情報

アプリが使うCPUの情報



# プロセス (3)

異なるプロセス間ではリソースを共有しない



メモリ  
共有しない

プロセス間通信の仕組みで、実際には情報のやりとりは可能

# スレッドについて

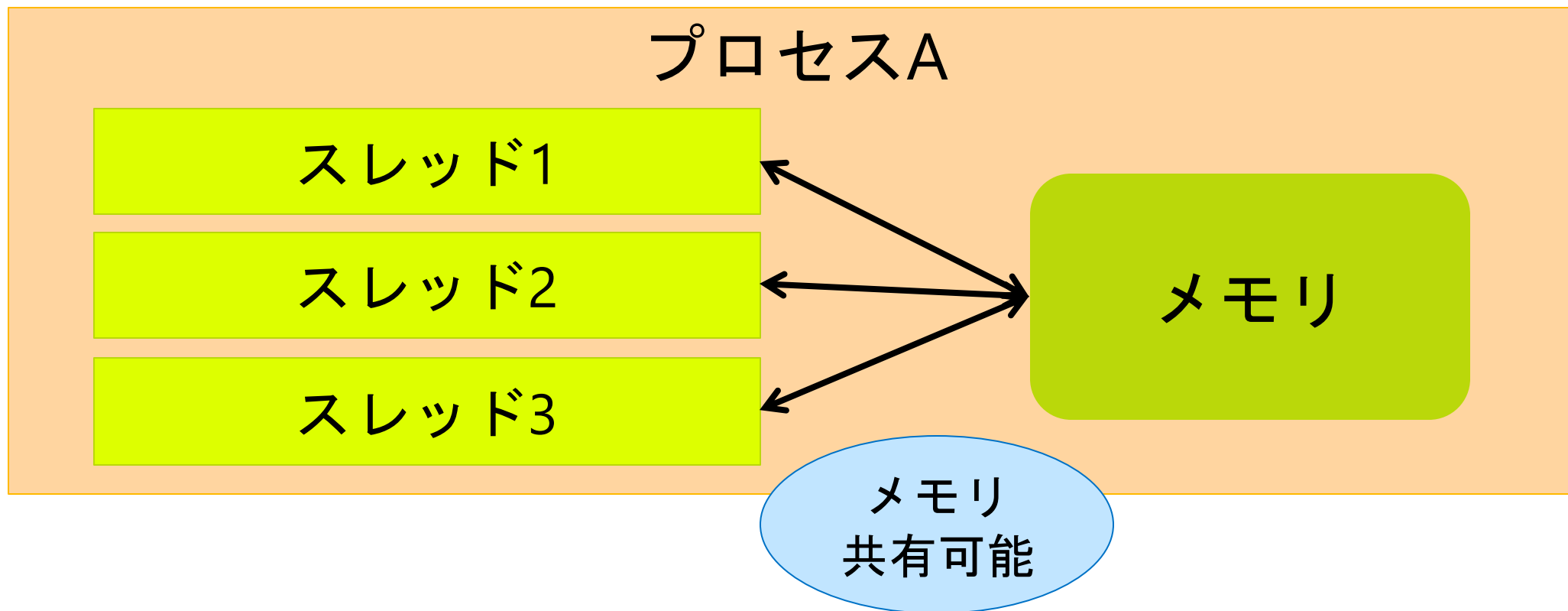


# スレッド

プロセスの内部の処理単位

→ プロセス自身のメモリをスレッド間で共有している

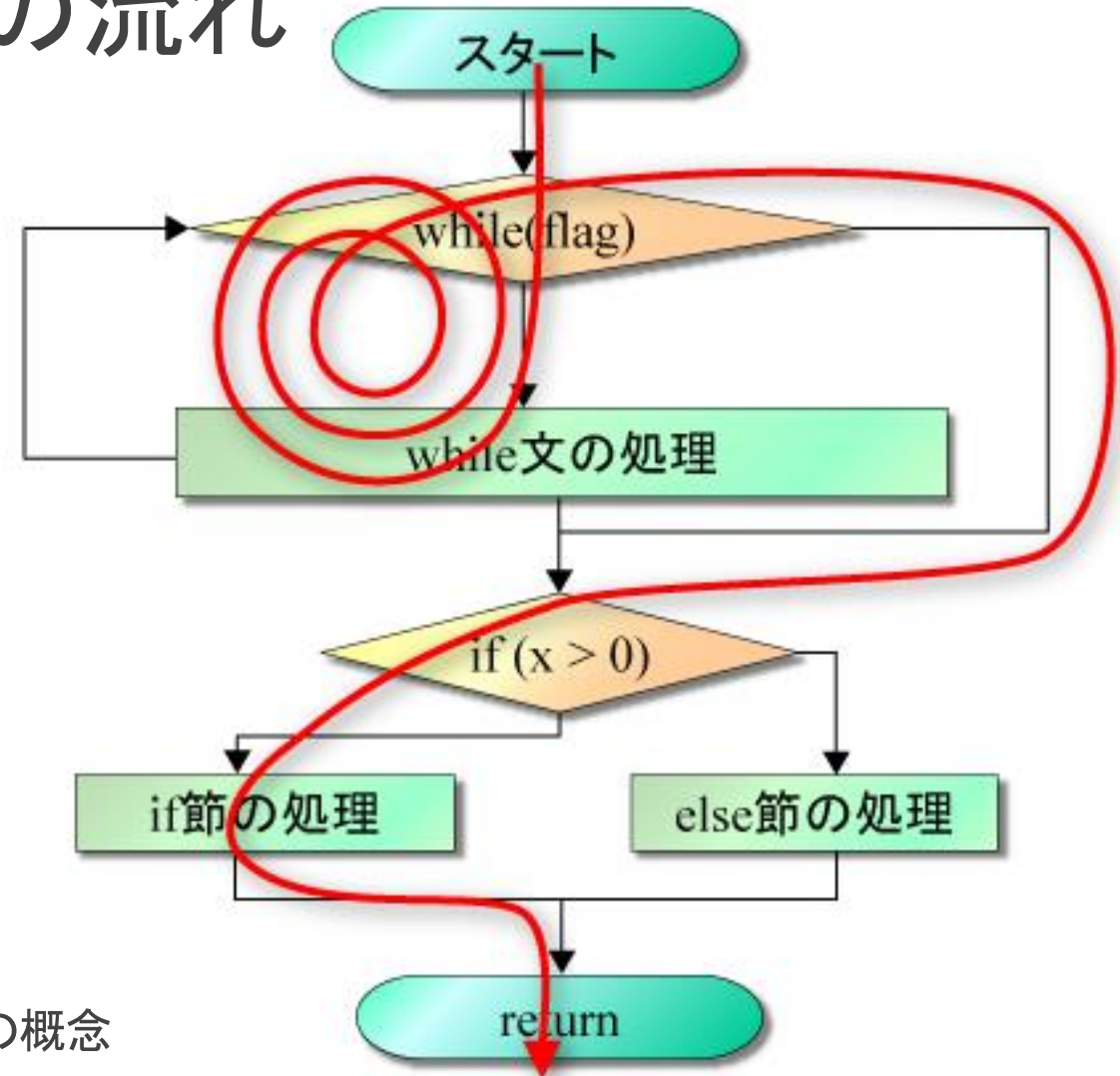
(番外編)  
スレッドがプロセスで実装  
言語VMでスレッド制御  
などもある



# スレッドは処理の流れ

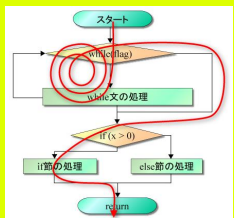
スレッドとは、処理の一連の流れ

CPU利用の単位となる

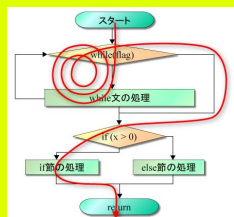


## プロセスA

### スレッド1



### スレッド2



...

(参考)第2回 実行メカニズムの理解に欠かせない「スレッド」の概念  
<http://itpro.nikkeibp.co.jp/article/COLUMN/20070416/268374/>

同時に処理するということ





# シングルスレッド

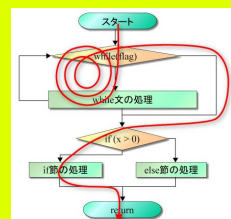
1プロセス、1スレッド

= シングルスレッドのプロセス

スレッドの  
処理の流れは1つ

プロセスA

スレッド1



JavaScriptはシング  
ルスレッドだけど  
非同期実装可能

# マルチスレッド (1)

1プロセス、複数スレッド

= マルチスレッドのプロセス

複数スレッドは  
同時に並列して処理が可能  
(マルチコアの場合)

プロセスA

スレッド1

スレッド2

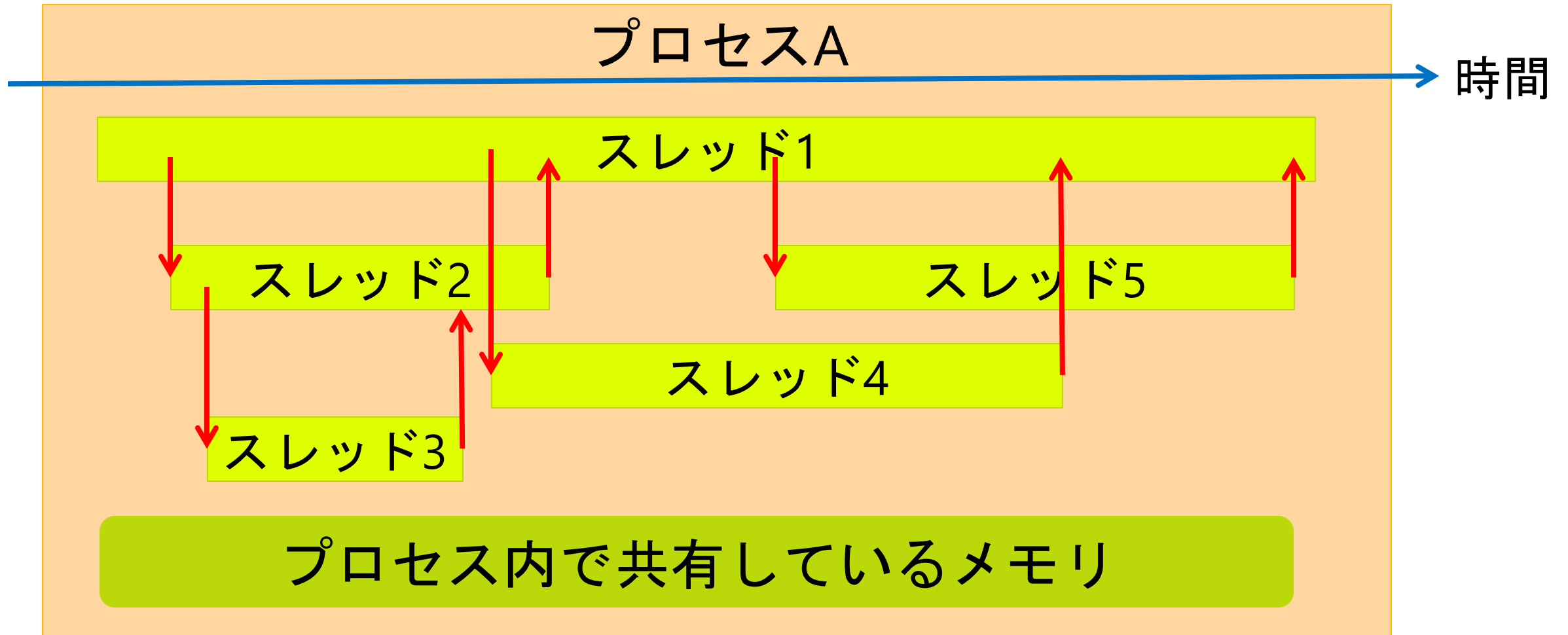
スレッド3

スレッド4

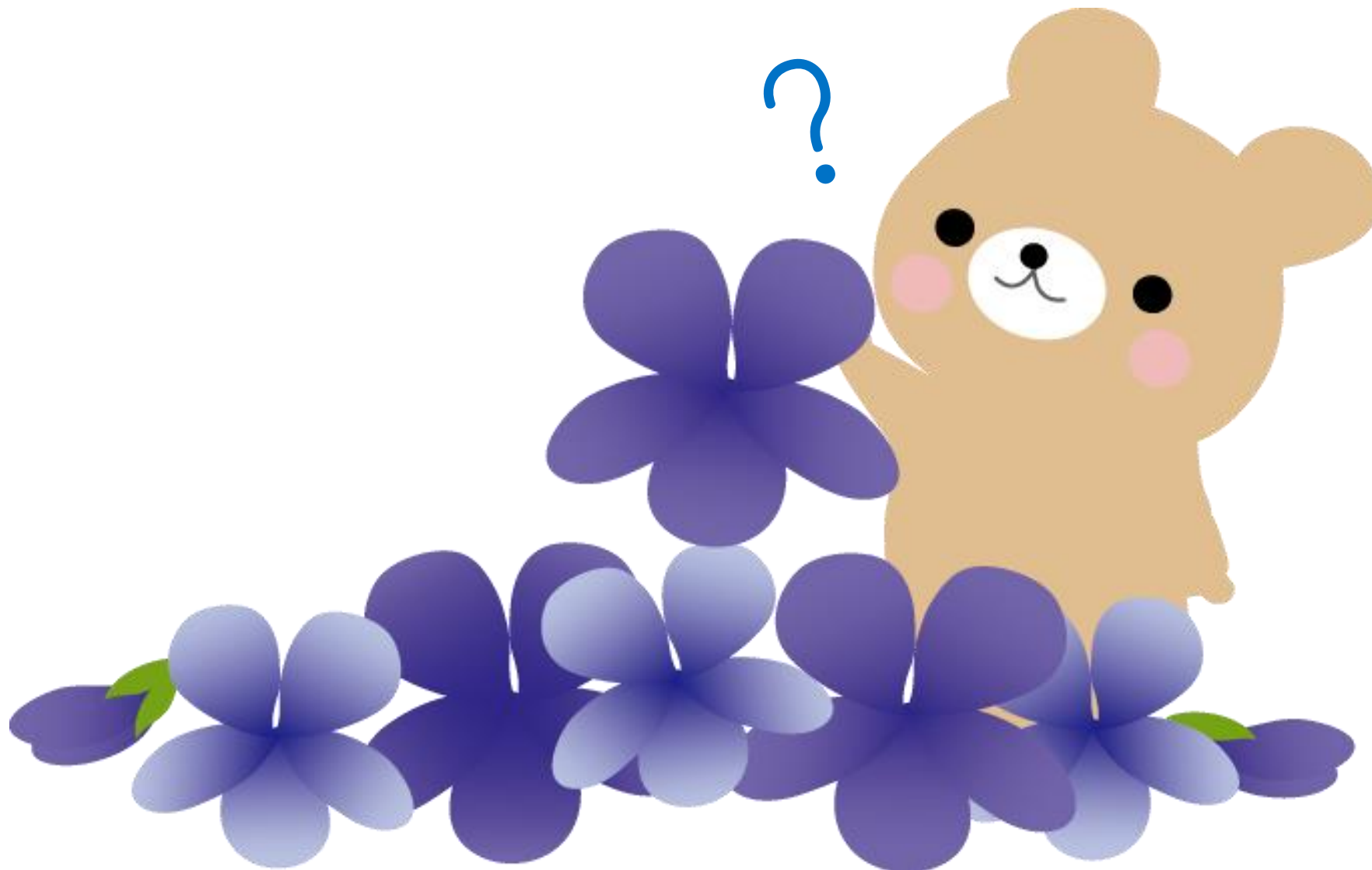
スレッド5

# マルチスレッド (2)

マルチスレッドの処理の流れイメージ



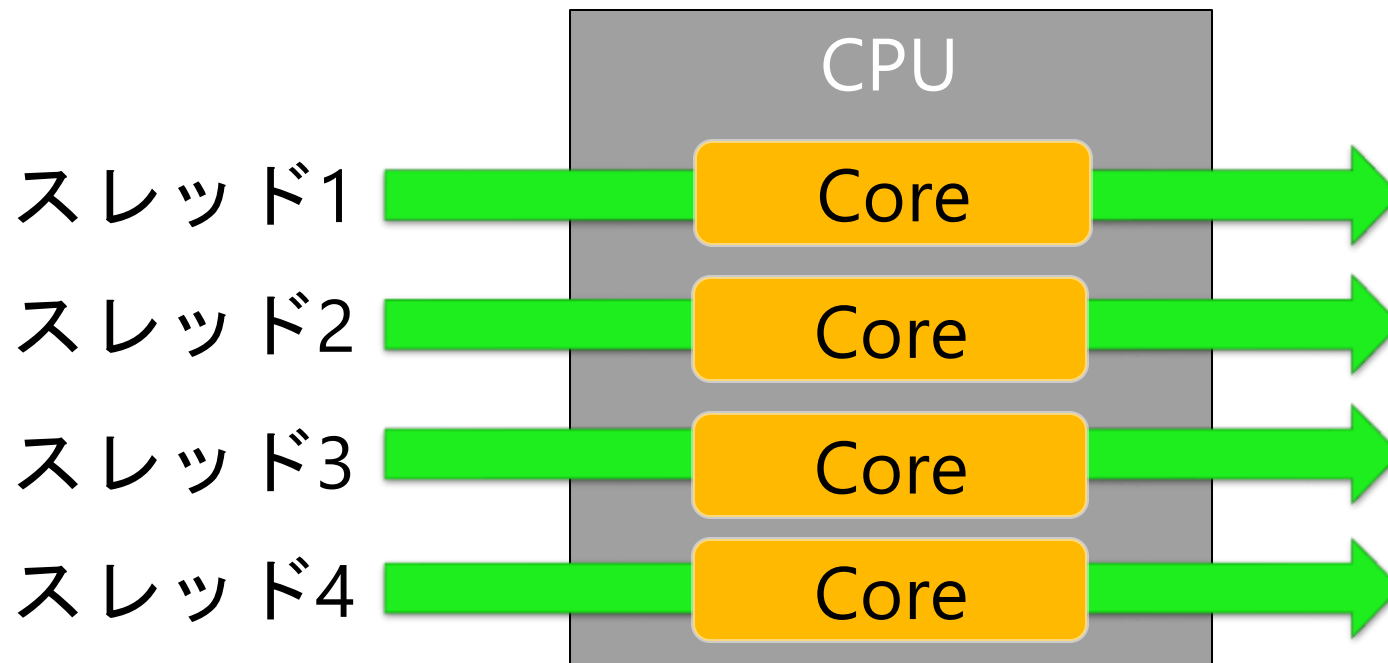
どうやって同時に処理するのか



# 1CPU-N Core マルチスレッド(マルチコア)

- CPUの中にスレッドを処理できるCPUを複数入れればよい (コア)

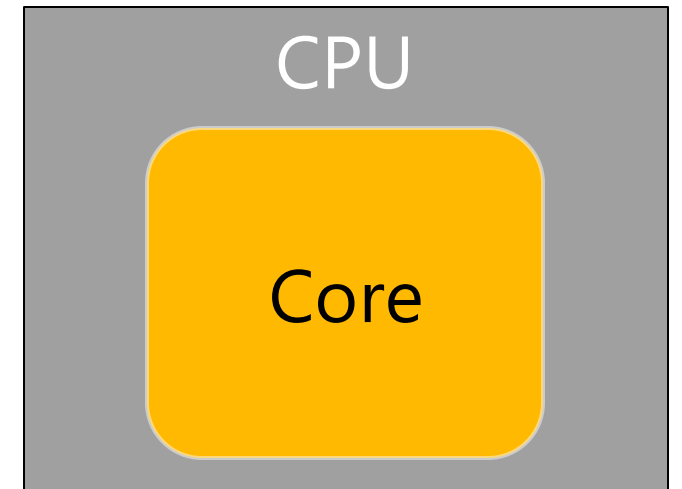
→ 1CPU / 4core なら、4スレッドを処理可能



Coreの数だけ  
平行して処理が可能!

(余談)

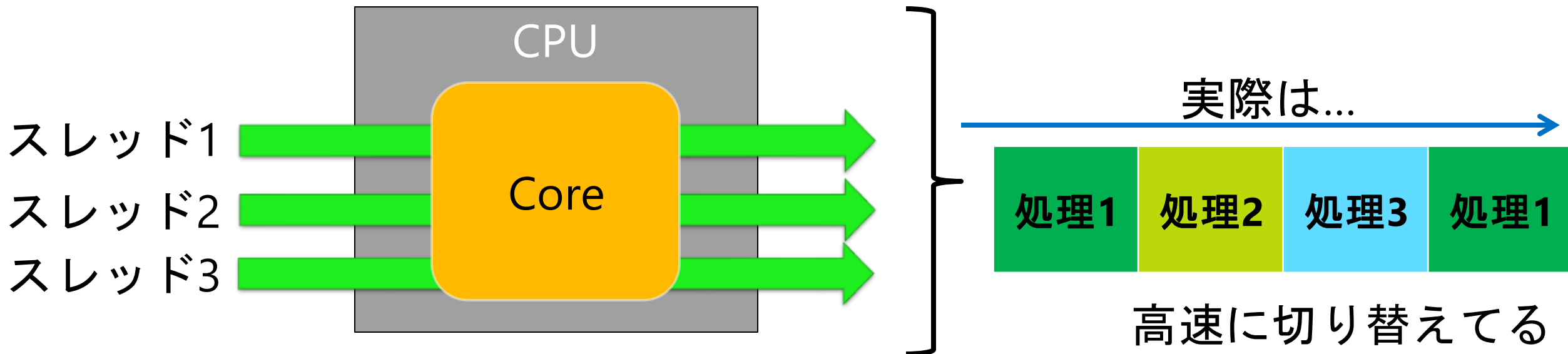
1CPU シングルCoreだと  
マルチスレッドは  
無理ですか？



# 1CPU シングルCore マルチスレッド

- OSが短い間隔で各々のスレッドを切り替えて、  
疑似的に複数のCPUがあるように振る舞う

複数のスレッドを並列して動作させている  
(ように見える) →切り替えに処理がかかり、(ry



現在では、

- シングルCoreのマルチスレッド化  
ではなく
- Core数増加によるマルチスレッド化  
の時代に...



# 「プロセスやスレッド」

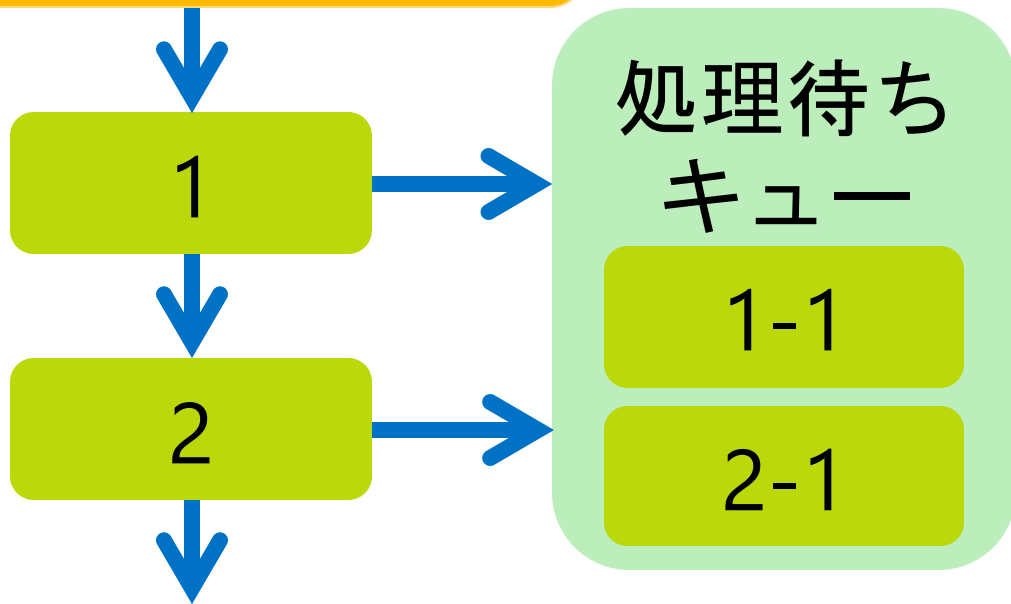
の概念は判った

で、何なのよ？

非同期処理の話はどうなった？

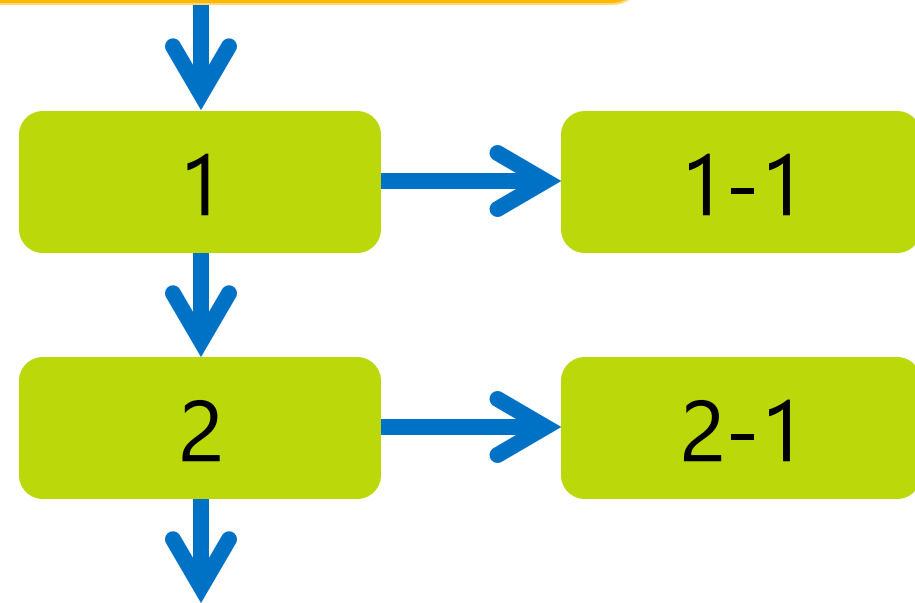
# 非同期処理とマルチスレッドの概念

非同期処理



前の処理を待たずに次へ  
(**同時処理かどうかは無関係**)

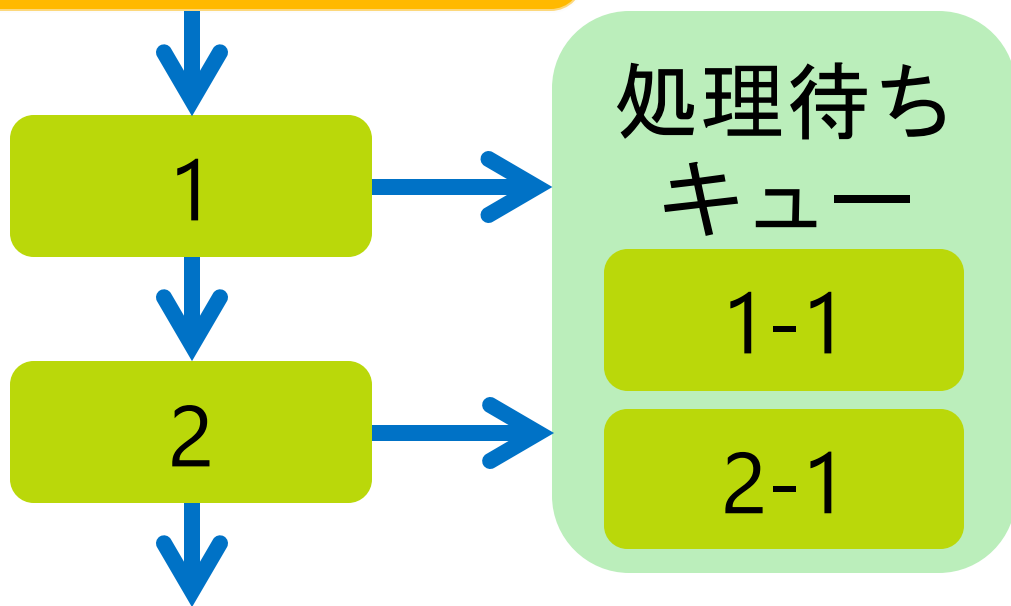
マルチスレッド  
(マルチコアの場合)



前の処理を待たずに次へ  
1と1-1は**同時に処理**

# 非同期処理とマルチスレッドの目的

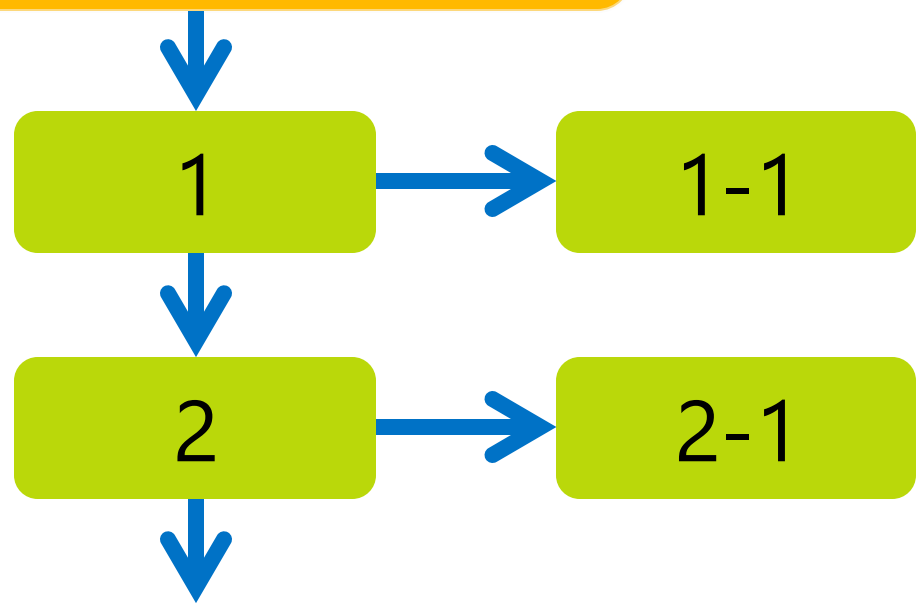
非同期処理



**並行**処理


目的は処理の流れを止めないこと

マルチスレッド  
(マルチコアの場合)



**並列**処理

目的は高速化



# 並列と並行 非同期処理の活用

# 並列と並行

混同して説明している資料が多い

**並列**高速化演算処理（ex. GPGPUなどを使う）

非同期処理 + **並行**処理（ex. イベント + 処理キュー）

例えば、後者でOSスレッドが使える場合は  
並行処理は並列処理として動作が可能になる

並行処理 → 並列処理に昇格！

# 非同期処理の活用例

1. レスポンスタイムの向上
2. スループットの向上
3. マルチスレッドの効率的利用

連載.NETマルチスレッド・プログラミング入門：第1回 マルチスレッドはこんなときに使う  
<http://www.atmarkit.co.jp/ait/articles/0503/12/news025.html>

# 非同期処理の活用例

	1つプロセス内の処理	複数プロセス間の処理	他リソース間の処理
非同期処理	<p>レスポンスタイムの向上 →「待たない」 並行処理</p> <p>マルチスレッドの 効率的利用 →並列処理</p>	<p>スループット向上</p> <p>ex) ディスクアクセスの間に、CPUを利用する</p>	<p>レスポンスタイムの向上</p> <p>ex) DBからデータを取得するなど、「非同期」でよく使われる例はこれ</p>

# 3. VC++非同期处理



# VC++2013では

	プロジェクト 例	種類	機能の例
非同期処理	C++	C++11 <future>	std::thread / std::promise std::async
	VC++ Windows Runtime	C++/CX ppltasks.h	concurrency::task
並列処理 (余談)	DirectX (GPGPU利用)	C++AMP amp.h	concurrency::parallel_for_each

C++11 async



# C++11のスレッド事情

## <thread> std::thread

C++11からマルチスレッドがサポートされた  
少し低レイヤで、結構生々しいコードになる

// 従来のC++03までは、各OSに依存したコード  
// なので、boost::thread などライブラリを利用

# std::thread sample

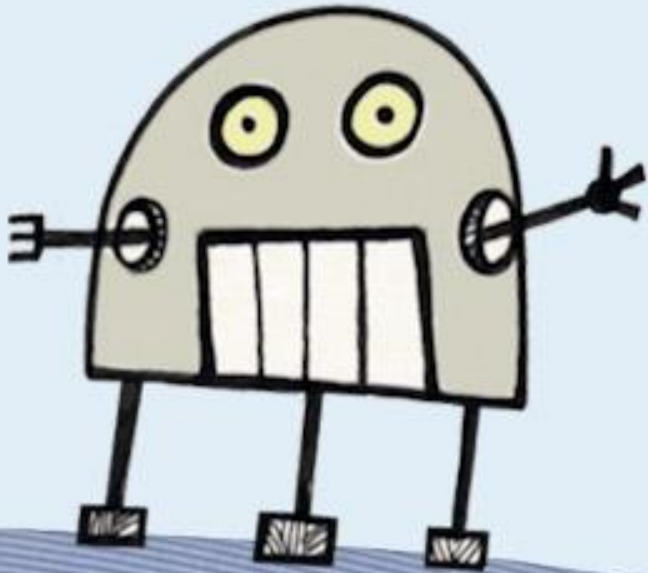
```
#include "stdafx.h"
#include <iostream>
#include <future>
int main(){
    int num = 0;
    std::promise<int> p00; // promise宣言 非同期プロバイダ
    std::thread t00([ &num, &p00 ](){ // thread で別タスクを実行する
        ++ num;
        p00.set_value( num ); // 非同期処理で返すものを設定
    });
    std::future<int> f00 = p00.get_future(); // 非同期受取り Obj宣言
    int result = f00.get(); // タスク処理を待つ (同期を取る)
    t00.join();
    return( 0 );
}
```

# std::async sample

```
#include "stdafx.h"
#include <iostream>
#include <future>
int main( ){
    int num = 0;
    std::future a00( [ &num ]( ){ // asyncで別タスク実行
        ++ num;
        return( num ); // 非同期処理で返すものを設定
    } );
    int result = a00.get(); // 同期を取る
    return( 0 );
}
```

threadに比べてasyncは  
ちょっとだけ簡易になった

# VC++2012/2013 async



# VC++のスレッド事情

**<ppltasks.h>**

**using concurrency**

**→PPL:Parallel Patterns Library(MS提供)**

VC++で用意されている非同期処理含めた  
並列処理は concurrency という名前空間にまとめ  
られている

- タスクの並列処理（そのために、スレッドも利用）
- 並列アルゴリズムもある

# concurrency::task sample

```
#include "stdafx.h"
#include <iostream>
#include <ppltasks.h> // <future>でもOK
int main( ) {
    int num = 0;
    concurrency::task<int>t01([ &num ](){ // taskで別タスク実行
        ++ num;
        return( num ); // 非同期処理で返すものを設定
    });
    int result = t01.get(); // 同期を取る
    return( 0 );
}
```

std::asyncと同じ形



# .then()を用いたTaskのつなげ方 (1)

```
concurrency::task<int> t1([]()  
{ return( 1 ); });
```

型は合わせることに注意

```
concurrency::task<int> t2 = t1.then([](int n)  
{ return( ++n ); });
```

```
concurrency::task<int> t3 = t2.then([](int n)  
{ return( ++n ); });
```

```
int result = t3.get() // 同期
```

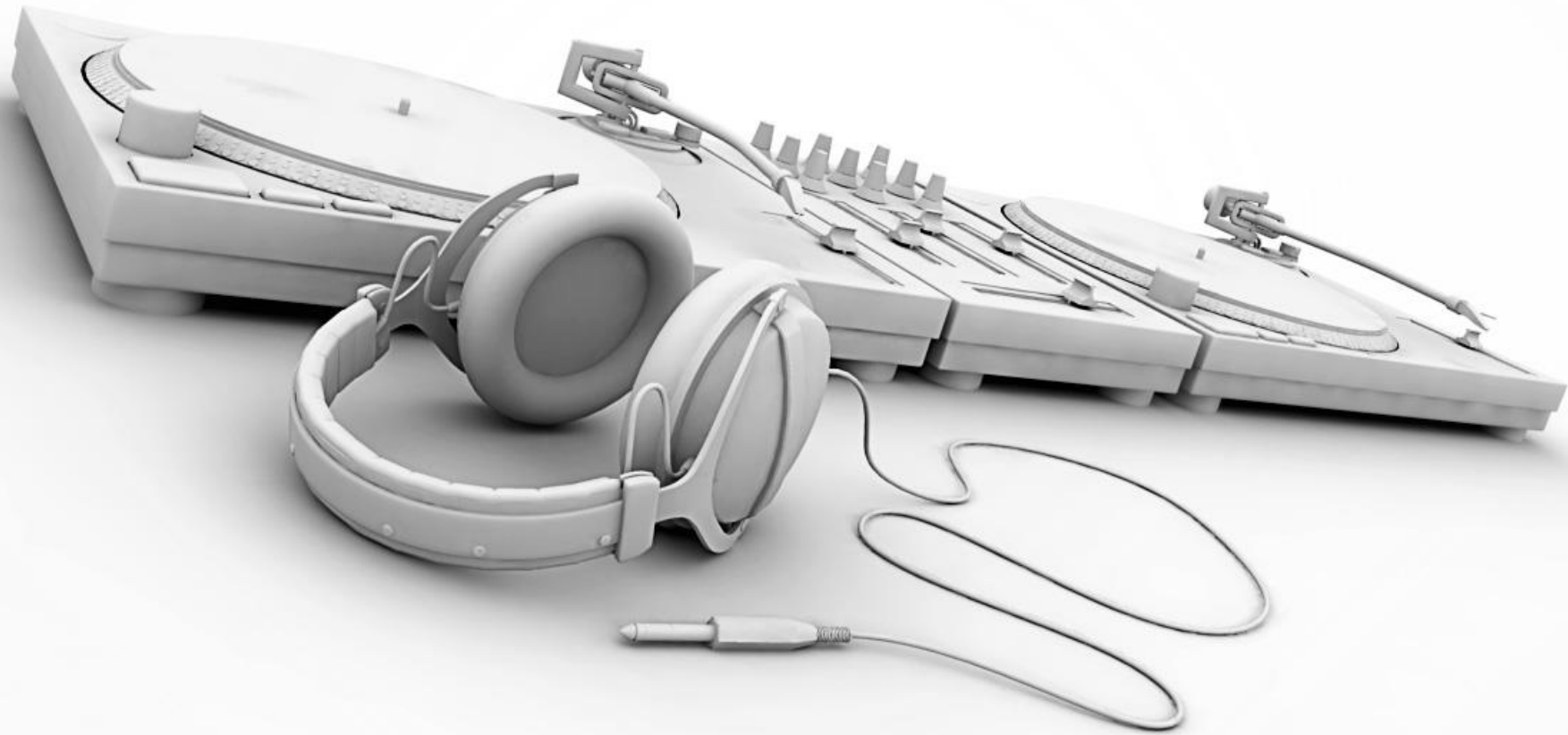
.then で戻り値を別のタスクに渡す  
どんどんつなげることが可能

# .then()を用いたTaskのつなげ方 (2)

```
concurrency::task<int> t([]()  
{  
    return( 1 );  
}).then([](int n)  
{  
    return( ++n );  
}).then([](int n)  
{  
    return( ++n );  
})
```

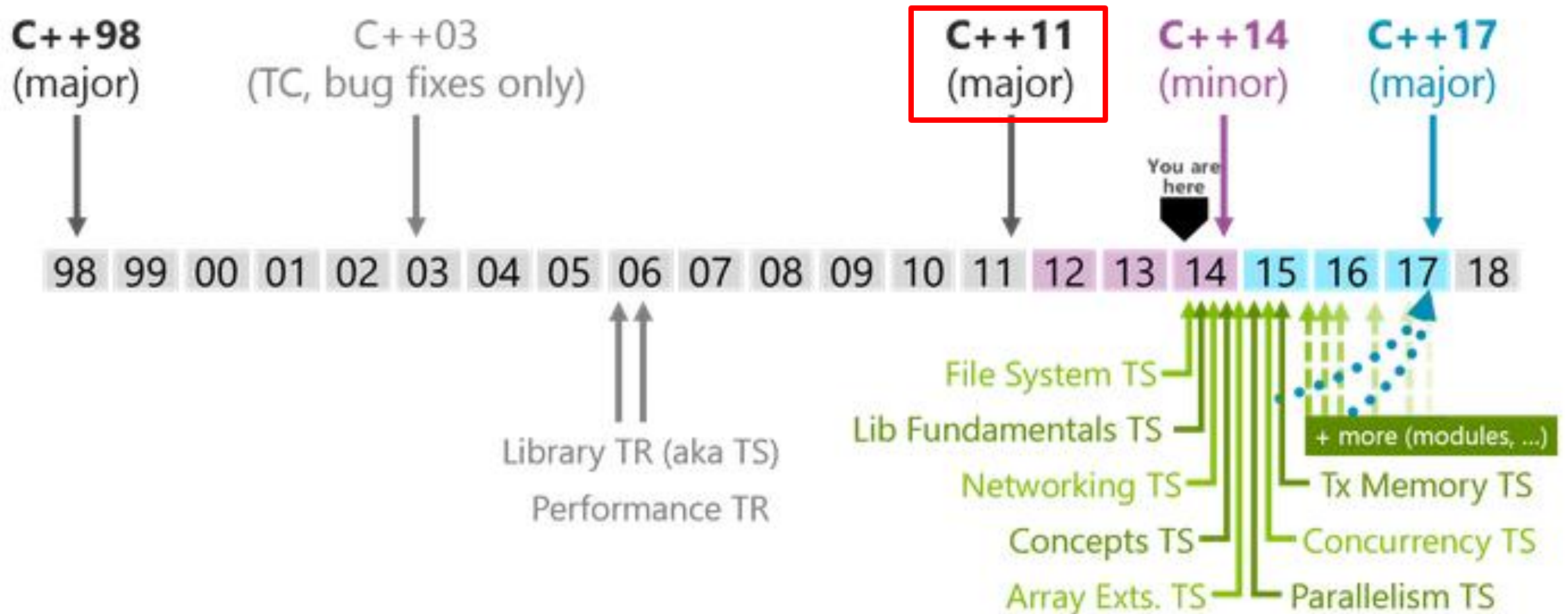
# VC++2013

## November 2013 CTP async



# Recently Published: C++11 (2011)

<http://isocpp.org/std/status>



# 最新のVC++2013事情

## C++11/C++14/C++17(予定)

### C++11

Rvalue references (対応完了)

ref-qualifiers

constexpr (一部)

Alignment

Inheriting constructors

Defaulted and deleted functions (対応完了)

Extended sizeof

noexcept (一部)

### C++11 Concurrency

Magic statics

### C++11 C99

\_\_func\_\_ (対応完了)

### C++14

auto and decltype(auto) return types

Generic lambdas (一部)

### C++17 (予定) Concurrency TS(?)

**Resumable functions and await** (一部)

# MSの提案しているVC++async

## resumable / await

C#の async/await のC++バージョン

- 関数宣言で `__resumable`
- `__resumable` では `concurrency::task<T>` を返す
- 処理実施 `__await`
- Main では `__await` が書けない などなど

# resumable/await Sample

```
#include <future>
#include <pplawait.h>
concurrency::task<void> my_proc(void) __resumable{
    auto x = []() __resumable->concurrency::task<void>
    {
        std::cout << "abc." << std::endl;
    };
    __await x();
    std::cout << "def." << std::endl;
}
int main() {
    auto task = my_proc();
    task.wait();
}
```

# 4. C++/CX非同期处理



# C++/CX事情

## C++11 前提のコーディング

- 自作スレッドは使わない(PPL Task標準利用)
- `std::shared_ptr`, ラムダ式など積極的に利用
- C++の規格のバージョンアップに伴ってコーディング方法もここ数年変化

# C++/CX Sample .then版

```
void App1::MainPage::my_btn_click(
    Platform::Object^ sender,
    Windows::UI::Xaml::RoutedEventArgs^ e)
{
    task<StorageFile^>(
        KnownFolders::DocumentsLibrary->CreateFileAsync(
            my_txt->Text
            ,CreationCollisionOption::ReplaceExisting)
    ).then([this](StorageFile^ file)
        {
            my_btn_01->Content = “ファイル作成しました”;
        });
}
```

# C++/CX Sample resumable/await版

```
concurrency::task<void>
App1::MainPage::my_btn_click(
    Platform::Object^ sender,
    Windows::UI::Xaml::RoutedEventArgs^ e) __resumable
{
    auto file = __await file->CreateFileAsync(
        my_txt->Text,
        CreationCollisionOption::ReplaceExisting);
    my_btn_01->Content = “ファイル作成しました”;
}
```

# 5. まとめ

# Summary

1. 非同期処理とは、待たない処理、処理の流れを止めない手法のことです  
本来の目的と、並列化や高速化を混同しないように！
2. C++11標準のタスク処理  
+ マイクロソフト提供のPPL タスク処理がある！
3. resumable/await は C#の async/await

# 6. おまけ

# めとべや公式 WPFアプリ操作ライブラリ



RM.Friendly.WPFStandardControls





# Nugetで入手できます！

## めとべや で検索



The screenshot shows the NuGet Package Manager console in Visual Studio. The window title is "Automation - NuGet パッケージの管理". The search bar at the top right contains the text "codeer", which is circled in red. The search results are displayed in a list view. The first result, "Friendly", is selected and highlighted in blue. The details for the selected package are shown on the right side of the console.

Automation - NuGet パッケージの管理

インストール済みのパッケージ: 安定版パッケージのみ | 並べ替え基準: 関連 | codeer

オンライン

- すべて
- nuget.org
- Microsoft and .NET
- 検索結果

更新プログラム

各パッケージのライセンスは、パッケージの所有者によって提供されます。Microsoft は、サードパーティのパッケージについて、一切責任を負わず、いかなるライセンスも提供しません。

1

設定(S) | 閉じる

**Friendly**  
Friendly Common Interfaces.

**Friendly.Windows** インストール(I)  
You can be friends with WinApp. And You can do an...

**Friendly.Windows.Grasp**  
You will be able to get the desired window. This library is built on Friendly Layer.

**Friendly.Windows.NativeStandardContr...**  
You will be able to manipulate Win32 Controls. This library is built on Friendly La...

**TestAssistant.GeneratorToolKit**  
You can make plugins for Test.Assistant.

**Friendly.FormsStandardControls**  
You will be able to manipulate WinForms Controls. This library is built on Friendly La...

作成者: Codeer  
識別: Codeer.Friendly.Windows  
バージョン: 2.0.0  
最終発行日: 2014/01/22  
ダウンロード: 56  
ライセンス  
ライセンスの表示  
プロジェクト情報  
不正使用を報告  
説明:  
You can be friends with WinApp. And You can do anything to it!  
タグ: Friendly Testing Windows WPF WinForms Win32 Automation  
依存関係:  
Codeer:Friendly (>= 2.0.0)  
上記の各項目には、追加のライセンス条項が適用される副次的な依存関係が存在する場合があります。



- 簡単にWPFのコントロールを操作。
- 実行の同期非同期が選択可能。
- 最終的には何でもできる！  
→というのは、下位レイヤで使ってるのが・・・

Only one



# Codeer.Friendly Windowsアプリ操作系最強！

他プロセスの  
メソッド、プロパティー、フィールド  
を何でも呼び出すことができる。

ありがとうございました

