並行並列プログラミング

松井 敏

自己紹介

- 👱 松井 敏(まつい びん)
- 👜 Codeer プログラマ(本業) & HACARUS C#&CI/CD メンター(副業)
- 🏆 Microsoft MVP for Developer Technologies 2012-2024
- 📚 Unity5 3Dゲーム開発講座 ユニティちゃんで作る本格アクションゲーム
- 💻 C#読書会主催、Greek Alphabet Software Academy TA
- ❤️ プログラム、マンガ、料理、睡眠、妻&子供

本日のゴールについて

- 並行並列プログラミングは難しい
- きっちり説明出来るようになるのは結構理解を深める必要がある
- 正直今日聞いただけで全て分かったとはならないと思う
- それだけ難しい内容なので前提がないとそもそも会話自体がしづらい
- 前提知識のキーワードを知らないと説明が分からないこともありえる
- 間違っても良いので並行並列の違いが分かり、会話が出来るようになることがゴール

アジェンダ

- 並行並列について
  - 辞書的な説明、イメージ的な説明、プログラム的な説明、キーワードの説明
- 特定の言語で非同期並列プログラミングとパフォーマンスについて
  - コードとデモを交えてパフォーマンスの文脈で実際の非同期並列の使い方
- 再度並行並列について
  - まとめ

今回の資料とChatGPT

- 今回の資料を作るのに一番やったことはChatGPTとの会話
- 兎に角疑問が出たら聞くことを続けていた
- 何度同じことを聞いても文句を言わないのが一番のメリットｗ
- 自分が得意なジャンル以外の知識を得るのにとても役に立った
- ChatGPTに聞いてもちらほら並行並列の使い方が間違っていた
- 何となくそもそもの集合知が間違っている気もする

並行並列の辞書的な説明

- 先ずはWikipedia上の言葉の定義から
- 並行計算（へいこうけいさん、英: Concurrent computing）

  > 並行計算とは、複数の計算あるいはアルゴリズムを、同一期間に同時実行させつつ相互に同調（コンカレント）させて、次の期間開始までに互いに完遂させるという計算形態を意味している

- 並列計算（へいれつけいさん、英語: parallel computing）

  > 並列計算は、コンピュータにおいて特定の処理をいくつかの独立した小さな処理に細分化し、複数の処理装置（プロセッサ）上でそれぞれの処理を同時に実行させることである。

- 辞書を読んで、その内容を把握できる人はかなり自頭の良い人。僕は無理。

コラム：並行並列は単語が似すぎている

- 並行と並列は単語が似ているため、誤解しやすい
- 英語だとConcurrentとParallelでこっちで覚えるという話も聞く
- この英単語の日本語訳を見るとさらに混乱した。。

> Concurrent:同時(発生)の、伴う、(...と)同時に起こって、共同に作用する、協力の、一致の、同意見の
>
> Concurrency:同時並行
>
> Parallel:平行の、(...と)平行して、(事柄など)相等しい、相似する、並行する、(...と)相似して、一致して、対応して、並列の

並行並列のイメージ的な説明

- 並行処理：洗濯の最中に料理をしよう
  - 「洗濯」というタスクと、「料理」というタスクを、同時に実行。これが、並行処理。

- 並列処理：ふたりで片付けを済ませよう
  - 「皿洗い」というタスクを、ふたりで一緒に行う。これが、並列処理。

引用:【実は別物】並行と並列、プログラミングでの違いを知ろう

並行のイメージ：補足

- 並行処理：洗濯と料理は比較的 1 人で両方やっても効率的にやりやすいかも。
- どちらも待ちが入るポイントがあるから
- これが例えば読書とプログラミングならどうか？

並行並列のプログラム的な説明

- 並行（Concurrent）

  > 複数のタスクが同時に「実行されているように見える」状態。実際には同時に実行されていないかもしれないが、スレッドやコルーチンがタスク間を効率的に切り替えながら実行。これは、シングルプロセッサのシステムでも実現可能。

- 並列（Parallel）

  > 同時に複数のタスクを実行すること。複数のプロセッサやコアを利用して、異なるタスクを同時に実行。例えば、マルチプロセッシング（複数のプロセスを実行）やマルチスレッド（同一プロセス内で複数のスレッドを実行）によって並列処理が行われる。

プロセス

- プロセスとは、OSによって管理される実行中のプログラムの単位
- プログラムが実行されると、そのプログラムはプロセスとしてOSによってメモリ空間を割り当てられる
- プロセスは他のプロセスから完全に独立しており、各プロセスはそれぞれ独自のメモリ空間を持っている

スレッド

- プロセスの中で実行される処理の最小単位。プロセスもスレッドの一種
- 1つのプロセス内で複数のスレッドを持つことができる
- プロセスに比べて、スレッドは軽量
- 同じプロセス内のスレッドはプロセスのメモリ空間を共有している

コルーチン、タスク：補足

- コルーチン（英: co-routine）とはプログラミングの構造の一種。サブルーチンがエントリーからリターンまでを一つの処理単位とするのに対し、コルーチンはいったん処理を中断した後、続きから処理を再開できる。接頭辞 co は協調を意味するが、複数のコルーチンが中断・継続により協調動作を行うことによる。C#の文脈ではあまり出てこないがUnityでは独自実装がある。

- タスクは、「任務、課題」、「仕事、職務」、「役割、目的」などの意味を持つ英単語。またコンピュータ処理における仕事の単位。OSや応用範囲により意味が異なる。プロセスと同義。スレッドと同義。タスク並列性などの用語では両者を区別しない場合もある。Microsoft Windowsでは、アプリケーションプロセスのことをタスクと呼ぶことがある。C#にはTaskクラスがある。

- タスクも複数意味を持つので要注意キーワード。

プロセスとスレッドまとめ

特徴	プロセス (Process)	スレッド (Thread)
メモリ空間	独立している	同じプロセス内のスレッドは共有している
コスト	スレッドより重い	軽量で低コスト
実行の独立性	完全に独立	スレッドがクラッシュすると同じプロセス内の他のスレッドに影響を与える可能性がある

並列並行のプログラム的な説明（再度）

- 並行（Concurrent）：複数のタスクが同時に「実行されているように見える」状態を指します。実際には同時に実行されていないかもしれませんが、スレッドやコルーチンがタスク間を効率的に切り替えながら実行します。これは、シングルプロセッサのシステムでも実現可能です。

- 並列（Parallel）：同時に複数のタスクを実行することを指します。複数のプロセッサやコアを利用して、異なるタスクを同時に実行します。例えば、マルチプロセッシング（複数のプロセスを実行）やマルチスレッド（同一プロセス内で複数のスレッドを実行）によって並列処理が行われます。

- ん？**マルチプロセスや**マルチスレッド？？

プロセスとスレッドの意味は共通

- プロセスとスレッドはほぼすべてのモダンなオペレーティングシステム（OS）で共通して持っている基本的な概念。
- 各OSによってプロセスやスレッドの具体的な実装や管理方法には違いがあるが、それぞれの役割や意味は一般的に同じ。
- そう考えると並列処理はプロセスよりスレッドを使った方が効果的に思える

何故マルチプロセス？

- プロセスはスレッドよりも作成や切り替えが高コスト
- プロセスはメモリが独立している。データを共有出来ない。
- 普通に考えてファイルに書き出すよりメモリを見る方が簡易で高速
- なんでマルチプロセス？？

Pythonの並行処理・並列処理

- Pythonの並列処理・並行処理をしっかり調べてみたの中でマルチプロセスの話が出てくる。
- Pythonは言語仕様で1つのスレッドしか動かない。そのため並列処理にマルチスレッドを使っても恩恵が受けられない
- 結果Pythonでは並行並列処理をマルチプロセスでやる
- 制限はCPUバウンドな処理に限られ、I/Oバウンドな処理ではスレッドを使っても効率的な処理が可能なのでスレッドが使われないわけではない。

並行並列について言語ごとの違い

- C#: 非同期プログラミング（async/await）が一般的。並列処理は主にTPLやマルチスレッドを通して行われる。
- Java: 並行処理が中心。非同期プログラミングはあるが、構文的なサポートはC#ほど強力ではない。
- Go: 並行処理が主流で、ゴルーチンを活用。非同期のキーワードはあまり使わない。
- Python: 非同期プログラミングが強調され、asyncioがよく使われる。並行処理もサポートされるが、GILの制約あり。
- JavaScript: 非同期プログラミングが中心。イベントループを使った非同期処理が非常に重要。
- Rust: 並行処理と非同期処理の両方が重要視されている。async/awaitやスレッドを使った並行処理が使われる。

ポイント

つまり並行の考え方は言語に依っても変わる部分がある

同期処理 (Synchronous Processing)

- 定義: タスクが順番に実行され、1つのタスクが完了するまで次のタスクに進まないこと。
- 例: A→B→Cの順番で処理が実行され、Aが完了しない限りBに進めない。
- まあ、普通に組んだら同期処理。

非同期処理 (Asynchronous Processing)

- 定義: 特定のタスクが完了を待たずに次のタスクに進むこと。あるタスクが実行中に、別のタスクが並行して進行する。
- 例: タスクAが実行中に、別のタスクBが同時に実行される可能性がある。Aが完了を待つことなくBが進行できる。
- 非同期処理は言語に依っても実装難易度が違う。

並行/並列/同期/非同期まとめ

用語	実行方法
並行処理	複数のタスクを**交互に少しずつ**実行
並列処理	複数のタスクを**同時に**実行
同期処理	タスクを**順番に**実行
非同期処理	タスクの**完了を待たずに**次のタスクを進行

並行のメリット

- タスクの同時進行・ユーザー体験の向上・応答性の向上
  - I/O待ちや長時間かかる計算を含むプログラムでは、他のタスクを並行して進めることで、待ち時間を有効活用できる
- リソースの効率的利用・効率的なリソース利用
  - CPU、メモリ、I/Oなどを効率的に使用するため、待ち時間の長い操作と他の計算を並行して処理することで、全体の処理をスムーズに進められる。例えば、CPUが待機する時間を減らし、その間に別の処理を行うことが可能
- スケーラビリティの向上
  - 並行処理を使うことで、システム全体がより多くのタスクやクライアントに対応できるようになる。並行処理を効果的に活用することで、大規模なサーバーやクラウドサービスが複数のリクエストを同時に処理でき、スケーラビリティを向上する
- パフォーマンスの向上
  - マルチコアでは、並行処理を使用することで各コアで異なるタスクを同時に実行でき、処理速度を向上させることができる。単一のタスクを1つのコアに任せるよりも、複数のコアを使ってタスクを分割して並行に実行することで、全体のパフォーマンスが向上する

Google Chrome：補足

- Google Chromeはマルチプロセス。
- セキュリティ強化
- 安定性の向上
- 最適化されたパフォーマンス
- リソース管理
- スケーラビリティの向上
- ガベージコレクションの負担軽減

> 引用:なぜGoogle Chromeは多くのプロセスを開いているのか？

パフォーマンスのメリットはほぼ並列処理の話でマルチプロセス固有の話ではなかった

並行のメリットでパフォーマンス

- 並行のメリットでパフォーマンスがあげられるが、個人的な見解として、並行処理で実際の速度を上がるのは難しめ。
- 例えば玉入れをするのに、並行処理をして速度が上がるか？
- 実際に速く処理するのであれば並列処理。にもかかわらず並行処理の文脈でちらほら出てくる。
- 並行処理はどちらかというと遅くみせないためのUI/UXのテクニックだと思う。

ここまでのまとめ

- 並行処理は複数のタスクを交互に少しずつ実行すること
- 並列処理は複数のタスクを同時に実行すること
- 並行並列はプログラム言語に依っても違いがある
- 並行処理は主に遅くみせないためのUI/UXのテクニック
- 並列処理は主にパフォーマンス向上のため
- キーワード:並行、並列、同期、非同期、プロセス、スレッド、タスク

並行並列の具体的な話

- ここからは概念や意味的な話よりも具体的なコードを踏まえた方が理解しやすい
- ただ並行並列そして同期非同期は言語によっても差異がある
- 自分の得意な言語の方がよりしっかりした説明が出来そう
- 言語は特定するがなるべく一般化したその言語だけの話にならないようにしたつもり

C#の並行並列

- C#において「並行」というキーワードはあまり使われない
- 代わりに「非同期」というキーワードがよく使われる
- 実際C#の書籍のタイトルでも「非同期並列」が使われる

パフォーマンスについて

- 非同期処理は遅くみせないためのUI/UXのテクニック
- 遅くみせないためのUI/UXのテクニックもパフォーマンスのテクニック
- 並列処理はパフォーマンス向上のため
- つまりここからは非同期並列とパフォーマンスの話

C#非同期並列プログラミングとパフォーマンス

C#非同期並列プログラミングとパフォーマンス

ターンアラウンドタイム、スループット、レスポンスタイム

- ターンアラウンドタイム（Turnaround Time）
  - ターンアラウンドタイムは、あるプロセスやタスクが開始されてから、すべての処理が完了するまでの総時間
- スループット（Throughput）
  - スループットは、一定時間内にシステムが処理できるタスクやジョブの量
  - 高いスループットは、システムが同時により多くの作業を効率よく処理できること
- レスポンスタイム（Response Time）
  - レスポンスタイムは、システムがリクエストを受けてから最初の応答を返すまでの時間
  - ユーザーから見て、システムがどれだけ「速く反応」してくれるかに関連する

レストランの具体例

- あるレストランで、複数のテーブルからの注文を受け付け、それをキッチンで調理する。
- ターンアラウンドタイム:注文が入ってから料理が完成してお客さんに提供されるまでの時間。
- スループット:一定時間内に処理できる注文の数。例えば、1時間あたりに何件の注文を処理できるかを示す
- レスポンスタイム :レストランの場合、注文を受けてから「注文を受け付けました」と最初に伝えられるまでの時間

レストランの具体例：同期処理の場合

- 例えば複数の注文が来た場合、同期的にやれば2つ目の料理が完成するには1つ目の料理が完成している必要がある
- ターンアラウンドタイムは料理1の時間 + 料理2の時間
- 料理1のレスポンスタイムはすぐだが、料理2のレスポンスタイムは料理1が終わってからになる
- スループットは1

レストランの具体例：並列処理の場合

- 一番簡単な高速化はスループットを2に上げること。2人でやれば料理は別々に作れる
- ターンアラウンドタイムは料理1と料理2で料理に時間がかかった方になる
- 2つ目の料理のレスポンスタイムもターンアラウンドタイムもそれぞれの時間で済むので速くなる
- 1つ目の料理のレスポンスタイムは変わらないが、レストランは料理1つを作ることが目的ではないのであまり意味はない
- これが並列処理なのでスループットは2

レストランの具体例：並行処理の場合

- さて、これを並行処理でやろうとすると1人2つの料理をするので同時進行がカギになる
- 少なくとも同期処理は止めたいので、まず非同期処理にしてマルチスレッドで作業する。
- これで2つ目のレスポンスタイムは縮められる。
- たとえ非同期にしても例えばサラダとメインを同時に作るのは難しい。
- 並列処理でスループットを上げるのは結構難しい。
- 例えば火入れみたいなタイミングで上手く並行化するとスループットが上がってターンアラウンドタイムは早く出来るかも
- でも非同期にしていれば、例えばオーブンに入れておいて、後どれぐらいかかるかを伝えることが出来る
- 非同期処理はこういう使い方をするのが現実的

パフォーマンスの遅さとは

- もう少し具体的にパフォーマンスの観点から考えてみる
- パフォーマンスが遅いとはどういうことか？
- 遅さを感じるポイントは2つある
- 1つはターンアラウンドタイム。ゲームで読み込みまで1分かかればそれは遅いと感じる。
- もう1つはレスポンスタイム。アクションに対してリアクションがなかったら遅いと感じる。
- チェックボックスのチェックやテキスト入力に 1 秒かかるとストレスがかかる
- DEMO

遅くても良いUI

- ボタンを押してメッセージボックスが出るまで 1 秒はそれほどストレスを感じない
- DEMO
- UIにおいて、時間がかかっても良いコントロールと良くないコントロールがある
- 個人的にボタンは時間がかかっても良い数少ないコントロール
- ただし、いくらボタンでも待てる時間は限界がある

ボタンのレスポンスタイムを上げる

- レスポンスタイムは結構シビアで一般的に3秒間リアクションがないと人は不安を感じるらしい
- UIで押したら押せない状態にする
- DEMO
- UI表現だけでレスポンスタイムは変わった。アクションに対するリアクションがレスポンスタイム。

ターンアラウンドタイムは変わらなくても待てるUX表現にする

- アニメーションするとより待てる
- DEMO
- ターンアラウンドタイムは変わってないが、待ってもよい気分にユーザーになれた。はず。

パフォーマンスのコツ

- 遅い処理は遅くても問題ないコントロールに処理をまとめる
- 遅い処理は相当早くなっても誰も気づかない。逆に言えばより遅くなっても気づかれづらい

簡単なUI/UX対応で済むケースもある

- パフォーマンスが直接向上しなくても、UI/UXで体感速度は下げることが出来る
- この対応だけでも悪くはない
- 社内アプリみたいなケースではこれで逃げることも選択肢

同期処理の限界

- ただこの対応は根本的な問題がある。同期処理の場合、重い処理だとUIが固まる
- これは商用アプリでは致命的
- UIを固まらなくなるするにはマルチスレッドを使う必要がある
- DEMO

UIコンポーネントの制限

- 但し、UIコンポーネントの制限としてUIスレッド以外からの書き込むと例外が出る
- DEMO

どの状況でもスレッドの意識は重要

- UIスレッドでの書き込みをする方法は幾つかかある
- C#のやりかたを覚える必要はないが、.NET固有の制限ではないので、逃げる術が必要なことの理解は必要
- DEMO
- 処理として速くなったわけではないがUIが固まらないアプリになった

並行プログラミングの読みにくさ

- ただ、先ほどのプログラムは、ロジックとUI処理が依存していて上から下に読んでも理解しづらい
- 理想的にはどう書きたいか？
- その形に限りなく近くする
- DEMO

async, await

- えーしんく、あうえいと
- C#5.0で発表された新しい非同期プログラミング
- 当時あまりにセンセーショナルで永遠の忠誠を誓った記憶があるｗ
- 今では他の言語にも派生している。JavaScript、Python、Kotlin、Swift、Rustなどでも可能
- 概念的はあったが、初出はF#？
- 個人的にはasyncいるかなと思ったけれど常識になってしまった

async, awaitは単純に書けても単純な実装ではない

- async,awaitは書き方はシンプル
- 上手く使えば手軽にかなり効果的に使える
- でも実装をみるとかなり大事なのは分かるかも
- 重要：単純に書けるからと言って単純な実装ではない

async, awaitの浸食

- 一度使えば呼び元にもasync,awaitは必要。
- C#の作法としては関数名の末尾もAsyncになる
- これはトコトンまで伝播する。浸食するとも言う。
- 適当な逃げ方をすると殆どの場合首を絞める

並列処理

- C#の並列処理は専用のライブラリもある
- Task Parallel LibraryとPLINQ
- どちらもわりと手軽に並列処理を書くことが出来る
- 注意点として順番通りにループが回ることが保証されなくなる
- 実装にも依るがこれだけで数倍速くなることもある
- DEMO
- 並列プログラミングはC#,Python,Rust,Java,Goなどでも扱える

SIMD：コラム

- スレッドを使って並列に処理するのではなく 1 命令でデータを並列に処理する方法もある
- SIMD(Single Instruction Multiple Data)
- System.Runtime.Intrinsics (イントリンシック)の命令(例えばAvx.Addなど)で、1回に256ビット処理するプログラムが書ける
- 例えばfloat型の配列なら256ビット / 32ビットで8個まとめて扱える。...単純に8倍速くなるではないが、状況によっては2-8倍以上速くなることもある
- CPUの命令語がサポートされているかどうかのif文を書いて使ったりする
- C++,Python,Rust,Java,GoなどでもSIMDは扱える

```
if (Avx2.IsSupported)
{
    var a = Vector256.Create(1, 2, 3, 4, 5, 6, 7, 8);
    var b = Vector256.Create(11, 12, 13, 14, 15, 16, 17, 18);
    var c = Avx2.Add(a, b);
    for (int i = 0; i < Vector256<int>.Count; i++)
    {
        Debug.WriteLine(c.GetElement(i));
    }
}
```

非同期並列プログラミングとパフォーマンスのまとめ

- レスポンスタイムが遅いと人は3秒で不安を感じる
- アクションに対してリアクションがあればUIだけで、レスポンスを早く見せることが出来る
- async,awaitを使えば、非同期プログラミングで、ターンアラウンドタイムが変わらなくても、UXで待つ事が気になりづらくなる
- 並列プログラミングはTPLなどのライブラリを使えば、スループットを上げて処理を速くすること出来る
- キーワード:ターンアラウンドタイム、スループット、レスポンスタイム、async,await

並行並列の再度まとめ

並行並列の再度まとめ

並行並列のプログラム的な説明

- 並行（Concurrent）

  > 複数のタスクが同時に「実行されているように見える」状態。実際には同時に実行されていないかもしれないが、スレッドやコルーチンがタスク間を効率的に切り替えながら実行。これは、シングルプロセッサのシステムでも実現可能。

- 並列（Parallel）

  > 同時に複数のタスクを実行すること。複数のプロセッサやコアを利用して、異なるタスクを同時に実行。例えば、マルチプロセッシング（複数のプロセスを実行）やマルチスレッド（同一プロセス内で複数のスレッドを実行）によって並列処理が行われる。

マルチスレッド

- 並行・並列プログラムはマルチスレッドを使って実装することが一般的
- ただし、並行はマルチスレッドでないと実装出来ないわけではない。コルーチンやイベント駆動などで実装することも出来る
- マルチスレッドからは扱えない、スレッドセーフではないライブラリもある
- マルチスレッドにして同時書き込みを考慮するとロックが必要になることも多い
- ロックによってパフォーマンスが劣化することも多い
- マルチスレッドになるとデバッグも難しくなる

並行のメリット

- 応答性の向上: ユーザーインターフェースの応答性を保つため、バックグラウンドでタスクを実行。
- I/O待ち時間の削減: ファイルやネットワークアクセス時の待ち時間を最小化。
- マルチコアプロセッサの有効活用: 複数のコアでタスクを並列実行し、処理速度を向上

処理速度を向上する設計はかなりの難易度。少なくとも机上で完全な説明が出来ないなら止めた方が良い

非同期のメリット

- C#ではそもそも並行と言う用語を使わない。応答性の向上やI/O待ち時間の削減には非同期を使う
- 非同期プログラムはasync,awaitを使うことで比較的シンプルな書き方で実装出来る
- 非同期はUI/UXの向上のために使うのが吉。うまく使えばパフォーマンスの劣化が気になりづらく出来る
- ただし内部実装含め決してシンプルなわけでない

並列のメリット

- 自分でマルチスレッドプログラムを駆使しして並列を期待するのはかなり厳しい
- 一方で言語によっては高レベルな抽象化されてライブラリが用意されており、比較的手軽に並列処理を実現できる。
- 但し、インデックスが連続しないなど制限はある
- 正しく使えば爆速になることも何度も経験している

並列処理の動きについて

- 並列処理のライブラリは高レベルな抽象化により、開発者は複雑なスレッド管理を意識せずに比較的手軽に並列処理を実現できる。
- C#に限らずこのようなライブラリが用意されている言語は複数ある。
- ただ、本当の意味でかならず並列で動くかはOS次第。並列で動くかもしれないし、並列で動かないかもしれない。
- あくまで並列で動かして欲しいというお気持ち表明みたいなものでしかない

最後に：並行並列を使うべきか

- 応答性を上げるシンプルな非同期以外ではレビューでも必ず厳しめに見るようにしている
- 落ちない、バグらない仕組みを目指すとすれば、避ける判断も重要
- 例えばよく分からないバグの場合、非同期が出てくると、それだけで疑いの目をかけるのは事実。
- 最低限使用する理由がコメントとして書いてないとそれだけでリスク
- 可能な限り使わない方が絶対的に平和
- トレードオフでパフォーマンスや応答が避けられない場合の最終判断として使うことはありえる
- その場合でも使用範囲は最小限になるように努力する
- 並列処理に至っては、使ってNGなわけではないが、自分は業務では使ったことは一度もない。趣味だけ。。

本日のゴール（再度）

- 並行並列プログラミングは難しい
- きっちり説明出来るようになるのは結構理解を深める必要がある
- 正直今日聞いただけで全て分かったとはならないと思う
- それだけ難しい内容なので前提がないとそもそも会話自体がしづらい
- 前提知識のキーワードを知らないと説明が分からないこともありえる
- 間違っても良いので並行並列の違いが分かり、会話が出来るようになることがゴール
- 使う時は正しく内容を把握して、容量用法を守ってお使いください

個人の感想

- ちなみにですが、いわずもがな今日の講義で最も学びを得たのは僕です
- 並行並列について、並行は非同期でasync,awaitを使う。並列はTPLを使うみたいな説明で書くつもりでした
- 例えば：並行は非同期処理で実現出来ます。と書こうとしてたが、非同期処理でないと実現出来ないわけではない。
- 例えば：並行はマルチスレッドで実現出来る。と書こうとしてたが、マルチスレッドでないと実現出来ないわけではない。
- 例えば：TPLで書いても必ずマルチコアを使って動くとは保証されていない
- コア使わず、スレッド複数使っているだけならそれは並行と言えるかもしれない。コントロール外。
- 当初自分が考えていたより、並行並列のプログラムは曖昧になってきた（ただし定義は曖昧ではない）

おしまい

- ありがとうございました。

登壇後見直しポイント

- 並行並列同期非同期の違いについてもう少し理解と説明にページを割いた方が良さげ
- 特に並行と非同期の違いについては分かりづらいので分かりやすい図を作る方が良いかも
- スレッドとプロセスの軽量について何が軽い、何が重いかはもう少し理解（CPUクロックとかの話）
- マルチプロセスのメリットについて出た、タブ切り替え時に完全に止めることが出来るという話の深堀
- メッセージループについてフワッとしか説明出来なかったので理解を深める
- Invokeについて質問されたとこともポイント。
- コルーチンについても質問されたので、別途追加説明を作る方が良いかも
- asyncキーワードが何故必要かはもう少し深堀理解をしておきたい。（いらないと思っている気持ちは変わらず）
- 全体的に最初の章が、一番質問されたし、自分の理解度も甘めの部分が多いので、説明、理解共に深めたい。
  - 最初は概念的ポイントでそれ以降は自分の見解なので、根本理解が甘いことの示唆だと思う
  - アセンブラレベルで非同期を考えるとよく分かると言われた。