加速器制御システム用コルーチンベース非同期ブリッジ

COROUTINE-BASED ASYNCHRONOUS BRIDGE FOR ACCELERATOR CONTROL SYSTEMS

ジメネズ・アレクシャンドレ
Alexandre Gimenez
RIKEN Center

Abstract

This paper introduces the concept of a "Co-routine Asynchronous bridge", used in the software within Particle Accelerator Control Systems, especially helpful for GUI implementation. Coroutines are a relatively old technique that has become increasingly popular, as it increases programmer productivity by simplifying program flow. It has been adopted in many languages (C#, Python, Perl, soon standard C++). However, coroutines are not mainstream yet, for several reasons, including the lack of coroutine-compatible software libraries. For coroutines to become mainstream, we need to combine existing library interfaces (including blocking calls, call-backs and event loops) and coroutine-based asynchronous code that invokes those libraries. This paper presents a strategy which allows wrapping any of the three interface types above into coroutine-compatible objects. A sample C++ implementation is developed as a case study and used to wrap existing software libraries with a coroutine-compatible shell.

1. はじめに

プログラミング習得の早い段階では、人間が把握しやすい「入力➡処理➡出力」という同期プログラミングパターンが多く使われている。一方、計算機内の処理は常に割り込みにより、非同期的に実行される。割り込みの間に処理がない場合は資源を節約でき、非同期インターフェイスを持つAPIも豊富にある。故に、商用ソフトウェア開発では非同期プログラミングが原則である。しかし、人間は非同期的に考えながらソフトウェア開発を行うには抵抗があり、その対策としての選択肢は下記の様になる。

(ア) 意識きて非同期プログラミングで開発すること(コールバック、イベントなどを用いて)

(イ) 計算機の非同期処理を隠蔽し、同期に似たプログラミング言語やツールを使う。

筆者の経験では、(イ)の方がソフトウェア開発者の生産性を高めるが、非同期隠蔽レイヤの複雑さにより最終成果物の品質が下がり、資源の無駄遣いも発生じる。

本論文では、従来の非同期隠蔽方法の分析を行い、コルーチンという選択肢のメリットについて、実験の結果を解析しながら記述する。

2. 非同期 API を同期に

ソフトウェアは下流 API かハードウェアかを叩きながら動作する。その下層のインターフェイスが非同期の場合、同期中継レイヤを挟むことが多い。スレッドを用い、非同期下流操作と上流制御を切り分けるという方法は一般的になっているが、スレッド間のデータ転送と同期処理(得にロックの使い方)に、不具合が発生しやすい。また、スレッドの作成、維持と消滅処理は比較的重いため、資源やリアルタイムがかかる(Figure 1)。

他方、スタックフルコルーチンを使い、同じスレッドに複数のコールスタックを同時に成立させ、上流・下流が協働しつつ、制御を渡し合う動作ができる。その仕組みは、Cooperative Multi-Tasking とも言うが、制御を渡す際にコールスタックを保存し、戻った時に保存されたスタックを復活することにより、上流制御から、同期操作と同様の効果が得られる。この仕組みはスレッドより細かい糸という意味で、「fiber」とも言われている。スレッドよりメリットはいくつかあるが、主にロックが不要ため不具合が生じにくいと考えられる。ただし、すべての fiber は定期的に自ら制御を譲らなければならないことがデメリットとなる(yield). Figure 2 を参照。

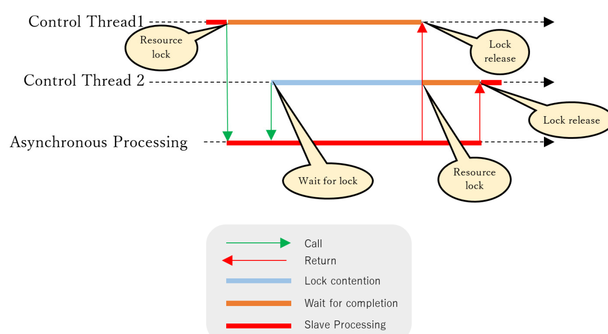

Figure 1: Using threads to encapsulate asynchronous processing.

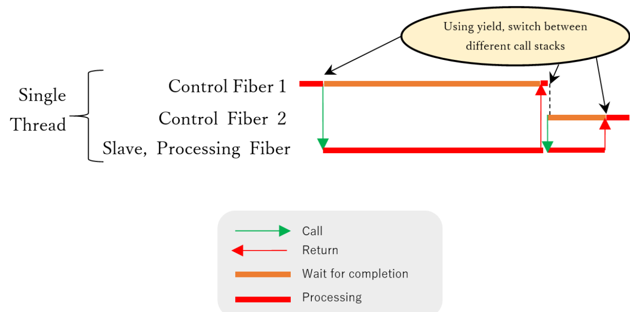

Figure 2: Using fibers to encapsulate asynchronous processing.

3. 動機・コルーチンの規格化

Cooperative Multi-Tasking 自体は斬新な技術ではないが、先述した「fiber」というコンセプトは、最初にコルーチンとして提案された。[1]と[2]は、サブルーチンの定義を広げ、値を戻しても状態が保存されるコンセプトが 30 年以上前に誕生した。このコンセプトを用いた技術は現在主流になっていないが、過去 5 年の間に、コルーチンを中心としたプラットフォーム(google node.js)とコルーチンを用いたプログラミング言語が増加している。

さらに、加速器制御ソフトウェアで使用されている C++ 言語へのコルーチン規格化も進んでいる。C++17 をターゲットにした定款もあり[3]、その他いくつかのコルーチンプロポーザルも存在する。その変化を考慮し、既に実装段階に進んだ Boost.Context を使い、加速器制御で使用可能なソフトウェア開発実験した。その構成を下記に記述する。

4. 既存 API との協働動作

コルーチンを利用すると、全層をコルーチンベースにすることが理想ではあるが、ほとんどの開発企画は既存 API の上に行われている。下流 API のインターフェイスがコルーチン非対応の場合、コルーチンから直接呼出せない。その対策として、既存 API をコルーチン対応にするライブラリを作成し、「Asynchronous Bridge」と定義した(コルーチンベース非同期ブリッジ)。

Asynchronus Bridge での制御は、Task・Device・Resource という抽象で実行される。

- Task は、fiber と同様のコルーチンスタイル実行単位であり、全タスクは同じスレッドで実行される
- Resource は隠蔽対象の下流 API を表現する
- Device は、Task から Resource へのアクセスを中継する。開発者は Device を継承して対象隠蔽 API を叩く想定。

タスクの状態を管理する「Scheduler」も定義する。タスクの状態は、実行中・Device 応答待ち・再生待ちのうちの一つである。Scheduler は、再生待ちタスクを、再生待ち FIFO キューに登録する。Task・Device・Resource の動作を下記に示す(Figure 3 参照)。

1) Task から呼ばれ、引数から必要な情報を Device に渡し、Device を起動
2) Device は下流 API を呼び出し、スレッドやコールバックを、必要に応じて登録
3) yield によって Task を一時停止(Task を Device 応答待ち状態にする)
   - 「再生待ちキュー」が空でない場合、その一位タスクを取得し再生(再生されたタスクは yield から戻る)
   - 「再生待ちキュー」が空の場合、Scheduler 全体は一時停止状態へ
4) 下流 API の処理終了時、戻り値を登録し、該当タスクを検索
5) 下記のいずれかが実行
   - 「再生待ちキュー」が空でない、もしくは実行中タスクが存在する場合、タスクを再生待ちキューに追加(後ろ)
   - 「再生待ちキュー」が空、かつ実行中タスクがない場合(全 Scheduler 停止状態)、Scheduler や該当タスクが再生(実行中状態へ)

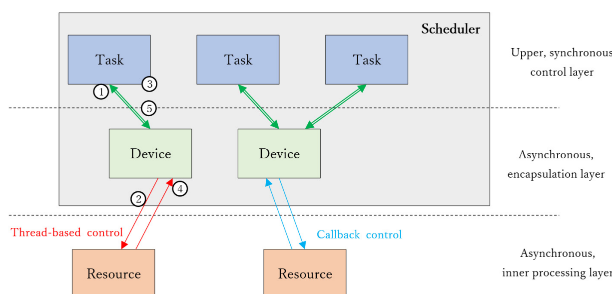

Figure 3: Asynchronous Bridge Operation.

5. 加速器でのコルーチンの使い道

加速器制御環境では、非同期ソフトウェアと非同期 API の利用が多い。

- ネットワーク転送
- GUI
- ハードウェア検出・制御(特に割り込み処理があるもの)

SPring-8 で開発中の「Spring-8 II」の進捗状況に合わせ次の実験開発を選択した。

- Qt ライブラリを隠蔽したコルーチンベース GUI 制御ツール
- Paho/C++ライブラリを隠蔽したコルーチンベース MQTT アクセスライブラリ。

代表的な例として、Win32 Message Loop と Dialog 入力テストも行った。

6. Qt コルーチンベース GUI

Qt フレームワークでは、Widgets というコンポネントを利用しユーザーの入出力を処理する。入力 Widget と出力 Widget 数個を一つの Device に包含して、タスクからの制御は理想の「入力➡処理➡出力」に近い動作に出来る。Figure 4 を参照。

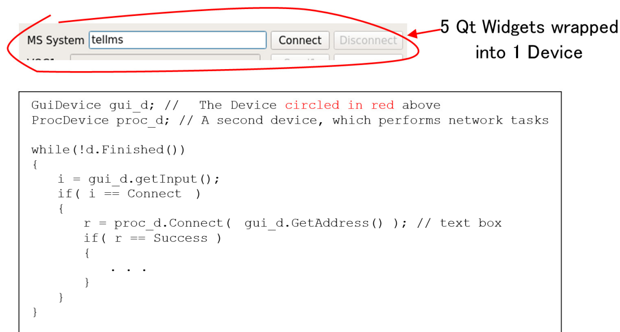

Figure 4: Synchronous access to Qt GUI.

すべてのタスクが停止された直後 Scheduler も停止し、次の Scheduler 再生は必ず前回と同じスレッドで行わなければならない。しかし、下流の API 処理終了時、Device のコードは別のスレッドで実行される可能性がある。故に、Qt ライブラリとの統合には、Qt イベントループとの協働動作が必要。下流処理終了後 Scheduler のスレッドに切り替えるには、下記の仕組みが必要（Figure 5 を参照)。

- 常に、Scheduler と全タスクを Qt のイベントループスレッドで実行
- 下流 API が戻った際、Qt へ signal を転送（ネーティブ Qt signal を使用）
- Qt のイベントループスレッドでその signal を受ける
- シグナルハンドラーから Scheduler を再生。

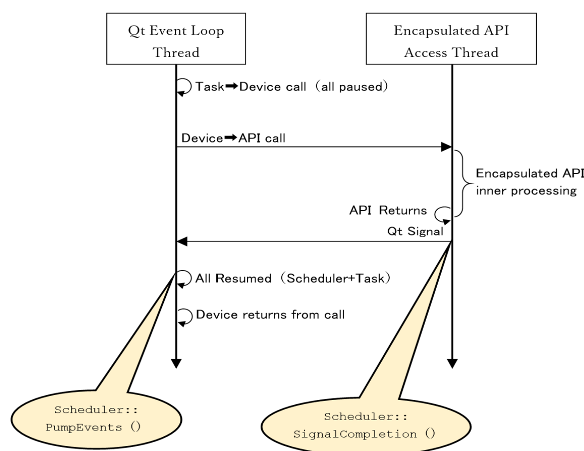

Figure 5: Cooperation with Qt event loop when the scheduler and all tasks are paused.

7. イベントベースフレームとの汎用統合

Qt 以外のイベントベースフレームワークとでも Scheduler を容易に統合できるように、Scheduler に汎用再生 API を装備した。

1) Scheduler 再生が必要時、Scheduler::SignalCompletion の仮想メソッドが呼び出される
2) SignalCompletion には、PumpEvents()のメソッドの呼び出しを Scheduler スレッドで起こす役目がある
3) 既存フレームワークと統合する際、Scheduler からクラスを継承、SignalCompletion をオーバーライドし、PumpEvents()を Scheduler スレッドで呼び出す。スレッド変換は、フレームワークが提供する仕組みで行う（例、上記の Qt の場合、Qt signal であった)。

上記の作戦で、Win32 との統合実験も行った。

8. MQTT コルーチンベースインターフェイス

Paho MQTT 非同期ライブラーはコールバックベースである。通常のコールバック統合を下記に示す。

- Asynchronous Bridge の Device を継承
- タスクが Device を呼んだ際、Device 内にコールバックを登録し、下流 API を呼び出す
- Device 内のコールバックが呼ばれた時、データを集め SignalCompletion() でタスクに制御を戻す（コールバックが実行されるスレッドが予測できないため)。

MQTT の接続状態管理と非同期イベントがあるため、Paho API コールの数と受信イベントの数が異なる。故に、イベントを受け、複数タスクに転送するコンポネントが必要。そのブロードキャストコンポネント(MQTTPump)もコルーチンベースにしたため、上流タスクの構成は、入力➡処理➡出力（例、MQTT Receive➡処理➡MQTT Send の繰り返し)のようなプログラムに作成可能。

9. ブロッキング下流 API との統合

下流、隠蔽対象 API がブロッキングの場合、API を呼ぶスレッドが必要。

- 初期化時に、必要な API アクセススレッドを作成
- Asynchronous Bridge の Device を継承
- タスクが Device を呼び出した際、引数データを API アクセススレッドに転送し、そのスレッドから API を呼び出す（一時停止されたタスクのスタック上にデータ転送するため、データ転送仕組みは容易）
- 下流 API が戻った際、データを集め、SignalCompletion()でタスクに制御を戻す（コールバック統合と同様の処理)。

Asynchronous Bridge では、API アクセススレッドの数と発行タイミングが調節出来る。

Spring-8 独自制御ブロッキング API を対象にし、ブロッキング API コルーチンアクセス実験を行なった。同じプロトタイプを Qt と統合し、GUI でユーザーがそのブロッキング API を任意制御できるツールとして導入される予定。

10. 結論

人間の脳の動作と計算機内の動作は異なる。故に、常に進化中開発ツールを利用し、開発者は成果物の目的から離れた、計算機専用コンセプトを考慮しながら作業する。順次処理を表現する同期プログラムで開発がで

きるなら、生産性も、保守性も、最終成果物の品質も改善できる。コルーチンベース開発は、その理想への一歩である。本論文で発表したコルーチン対応性を向上させる技術とその結果分析で、アプリケーションの一部をコルーチンベースに出来ること確認できた。

参考文献

[1] " Design of a Separable Transition-Diagram Compiler", Melvin E. Conway.
[2] "The Art of Computer Programming", Donald Knuth, 1.4.2 章、ISBN 0-201-89683-4.
[3] Co-Routines TS/C++17 対象 ; http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0057r5.pdf
[4] Boost.Fiber ライブラリ; http://www.boost.org/doc/libs/1_64_0/libs/fiber/doc/html/index.html