COBOL によるマルチスレッドアプリケーション

目次

はじめに 2

1 スレッド競合の問題点 2

2 スレッド競合の回避手段 3

3 COBOL によるスレッドプログラミング 4

3.1 スレッドの生成・終了 4

3.2 スレッド固有メモリ領域 6

3.3 同期方式の実現 8

4 Java, .NET のマルチスレッド環境での COBOL の利用 10

4.1 .NET CLR 上におけるスレッド毎のメモリ分離 10

4.2 JVM 上におけるスレッド毎のメモリ分離 12

4.3 マルチスレッド Java アプリケーションからのネイティブ COBOL アプリケーションの利用 17

5 マルチスレッド環境でのデータベースアクセス 19

5.1 OpenESQL の ODBC 機能を使ったマルチスレッドアプリケーション開発 19

5.2 OpenESQL の ADO.NET、JDBC 機能を使ったマルチスレッドアプリケーション開発 21

5.3 Pro*COBOL を使ったマルチスレッドアプリケーション開発 23

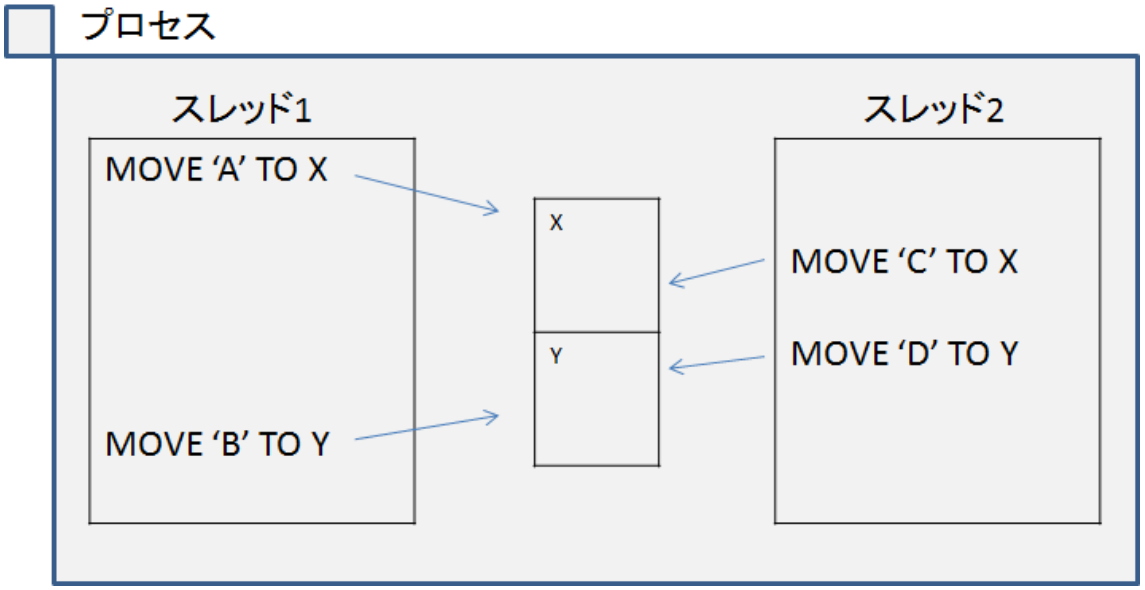
おわりに 25

はじめに

COBOL の歴史はバッチ処理からスタートしました。バッチ処理は大量データを順次処理する方式であり、複数の処理の並列実行を想定していません。その後 COBOL がオンライン処理で利用されるようになっても、CICS や TUXEDO などに代表されるオンライン基盤はプロセスプーリング方式を主流としており、COBOL のプログラムがマルチスレッドで実行されることはあまりありませんでした。
しかし昨今では、JVM や .NET 環境で COBOL を活用することが自然になってきたため、COBOL 開発者もプログラムがマルチスレッドで動作することを念頭に置くことが必要になるケースが出てきています。
本書では、このようなケースで COBOL 開発者が注意しなければならない事項について解説します。

1 スレッド競合の問題点

スレッドとは、一つのプロセス内で個別の実行コンテキストを保持して並行して処理を実行することができる軽量の実行単位です。プロセスは個々でメモリ、CPU、開かれたファイル、データベース接続などのリソースを持ちますが、スレッドはプロセスが持つこれらのリソースを共有しながら、個別にスレッドごとのメモリと CPU 実行コンテキストを持つことができます。
スレッドは 1 つのプロセス内で実行されるので、プロセス内のすべてのスレッドは、データおよびコードの同じアドレス領域、その他の大部分のリソースを共有します。 また、各スレッドはそれぞれ固有のレジスタセットとスタック領域をもっています。 そのため、スレッドの作成には大量のシステムメモリは必要ありません。 同じプロセス内のあるスレッドから別のスレッドに実行コンテキストを切り替えるときに、オペレーティングシステムが行うことは CPU の所有権とレジスタセットの切替えのみです。 この理由により、スレッドは「軽量プロセス」と呼ばれます。
一方、マルチスレッドアプリケーションにも欠点があります。 各スレッドは、同じプロセスの他のスレッドと共有する可能性のあるリソースを認識している必要があります。 プログラマは、複数のスレッドが同時に実行されることに注意してください。また、2 つのスレッドが実行同期を取らずに同じメモリ領域に書き込みを行うと、そのデータ項目が破損する可能性があります。例として 2 つのスレッドが以下のような順序で同じメモリ領域に書き込んでいるケースを考えます。



この例では、どちらのスレッドも予期した値を獲得しません。スレッド 1 の予期する結果は 'AB' で、スレッド 2 のときする結果は 'CD' です。 ところが両方のスレッドが実際に獲得する値は 'CB' です。 このようにマルチスレッドアプリケーションを作成する場合は、各スレッド間で実行の同期を取り、スレッド間でデータが競合しないようにしなければなりません。
ここで挙げた例はメモリ領域のスレッド間競合問題です。この他に以下のような競合問題があり得ます。

1) プロセスとしてオープン済みのファイルをあるスレッドがクローズしてしまったために他のスレッドが読むことができなくなった。
2) あるスレッドがデータベースの接続属性を変更してしまったために、この接続を他のスレッドが使用した場合に予期せぬ結果となった。
3) 複数のスレッドが同じデータベース接続に対して同じカーソルを同時に作成して結果セットにアクセスしたために双方に異なる結果が返った。

2 スレッド競合の回避手段

このようなスレッド間競合問題を解決する手段は以下の 2 種類に大別されます:

1) 競合するリソースをスレッドごとに割り当てる
この手段は本来スレッドを使用するメリットであるリソース削減の効果が薄れます。しかし、スレッド対応されていないアプリケーションをできるだけ変更せずにマルチスレッド環境で使用するという目的には適しています。
例えばメモリ領域であればプロセス内で全スレッドから共有される static 領域やヒープ領域を使わずにスレッド内でローカルに割り当てられたメモリを使用するなどの方法がこれに相当します。

2) 共有リソースが競合しないようにスレッドごとに使用時間をずらす
この手段であれば少ないリソースで並行処理を実現することが可能です。一方でプログラミングは複雑になりデバッグも困難になりがちです。
この方法は同期方式と呼ばれ、一言でいえば「リソースを更新して利用するまでの間スレッドで占有する」というものです。1. で挙げた例の場合、スレッド 1 が X と Y に ‘A’ と ‘B’ を書き込みその領域を使用し終わった後で、スレッド 2 が X と Y に ‘C’ と ‘D’ を書き込みその領域を使用すれば、両方のスレッドはそれぞれ予期した結果を得ることができます。
これを実現する方式は次の 2 つに大別されます:

> セマフォのような同期プリミティブを確保してから処理を開始し、終了したら開放する
> 処理の単位をシンクロナイズド属性にして他スレッドからの割り込みを禁止する

これらの手段をアプリケーションで利用する方法としては以下のようなものがあります。

1) オペレーティングシステムやフレームワークが提供するスレッド対応 API
Windows や Linux などの OS は Thread Local Storage と呼ばれるスレッド内でのみアクセス可能なメモリ領域を確保する API を提供しています。同様な機能は Java や .NET のクラスライブラリでも提供されています。
また、同期方式をとる場合にはセマフォの操作を行うための API も提供されています。

2) プログラミング言語が提供するスレッド対応言語要素
Java ではあるコードブロックに Synchronized 属性を付加することによって、そのブロック内の処理は他のスレッドから割り込まれることなく処理されることが保証されます。同じことは C# の lock キーワードや Visual Basic の SyncLock キーワードとしてもサポートされています。
COBOL プログラマに提供されているスレッド対応構文については 3. で詳しく解説します。

3) データベース管理システムが提供するスレッド対応 API
アプリケーションプログラムからのデータベースアクセスは、接続やカーソルなどのデータベース管理システムや ODBC/ADO/JDBC などのレイヤーが提供するリソースを使用します。これらのリソースもやはりプロセス内のスレッド間で共有されうるため競合の可能性があります。このような問題を解決するために、データベースアクセスに固有のスレッド競合回避手段が提供されています。これについては 5. で詳しく解説します。

3 COBOL によるスレッドプログラミング

COBOL で書かれた既存のビジネスロジックを新しい環境で再利用する場合、その新しい環境がマルチスレッドであるためにスレッド競合解決しなければならなくなるケースがあります。再利用するプログラムはもともとマルチスレッドで実行されることを予期していませんので、スレッド競合問題を解決する何らかの手段が必要となります。本章では COBOL 言語で利用可能なスレッド対応プログラミングについて説明します。

3.1 スレッドの生成・終了

再利用する COBOL プログラムがスレッド対応を余儀なくされるケースはあっても、純粋に COBOL のみで書かれたアプリケーションが意図的にスレッドを生成して操作するケースはあまりありません。しかしそれは可能です。たとえばバッチ処理であっても必ずしもすべて逐次的に処理しなければならないわけではなく、部分的には並行処理が可能な場合もあります。このような場合、並列実行ジョブネットを組むことによってコンピュータで利用可能な CPU・メモリリソースを最大限に活用するのが通常の方法ですが、一つの COBOL プログラムの内部で複数のスレッドを生成して 1 プロセス内での並列処理を行うこともできます。
以下の例を見てみましょう。

```
WORKING-STORAGE SECTION.
01  WK-COUNTER    PIC 9(10).
01  WK-CTR-EAST   PIC 9(10).
01  WK-CTR-WEST   PIC 9(10).
01  WK-CTR-SOUTH  PIC 9(10).
01  WK-CTR-NORTH  PIC 9(10).
PROCEDURE DIVISION.
    MOVE ZERO TO WK-COUNTER.
    PERFORM DO-EAST.  <* 東地区の処理
    PERFORM DO-WEST.  <* 西地区の処理
    PERFORM DO-SOUTH. <* 南地区の処理
    PERFORM DO-NORTH. <* 北地区の処理
    DISPLAY WK-COUNTER.
    STOP RUN.
DO-EAST.
    MOVE ZERO TO WK-CTR-EAST.
    PERFORM 1000000000 TIMES ADD 1 TO WK-CTR-EAST.
    ADD WK-CTR-EAST TO WK-COUNTER.
DO-WEST.
    MOVE ZERO TO WK-CTR-WEST.
    PERFORM 1000000000 TIMES ADD 1 TO WK-CTR-WEST.
    ADD WK-CTR-WEST TO WK-COUNTER.
DO-SOUTH.
    MOVE ZERO TO WK-CTR-SOUTH.
    PERFORM 1000000000 TIMES ADD 1 TO WK-CTR-SOUTH.
    ADD WK-CTR-SOUTH TO WK-COUNTER.
DO-NORTH.
    MOVE ZERO TO WK-CTR-NORTH.
    PERFORM 1000000000 TIMES ADD 1 TO WK-CTR-NORTH.
    ADD WK-CTR-NORTH TO WK-COUNTER.
```

この例では、東西南北 4 地区の業務処理を逐次処理しています。ここでは説明を簡略化するために各地区の処理は 10 億回の足し算にしてあります。全体の処理結果としては 40 億回の足し算が実行されます。この 4 地区の処理をそれぞれ異なるスレッドで実行するためには、Visual COBOL 固有の START 文を使用します。また、スレッド個別で実施された加算結果を全体のカウンタに集計する処理は、個々のスレッドの処理終了を待ち合わせてからに

しなければなりません。このために Visual COBOL は WAIT FOR 文をサポートしています。これらの構文を使用して上記の処理をマルチスレッドに書き換えると以下のようになります。

```
WORKING-STORAGE SECTION.
01  WK-COUNTER    PIC 9(10).
01  WK-CTR-EAST   PIC 9(10).
01  WK-THR-EAST   USAGE THREAD-POINTER.
01  WK-CTR-WEST   PIC 9(10).
01  WK-THR-WEST   USAGE THREAD-POINTER.
01  WK-CTR-SOUTH  PIC 9(10).
01  WK-THR-SOUTH  USAGE THREAD-POINTER.
01  WK-CTR-NORTH  PIC 9(10).
01  WK-THR-NORTH  USAGE THREAD-POINTER.
PROCEDURE DIVISION.
    MOVE ZERO TO WK-COUNTER.
    START "DO-EAST"  IDENTIFIED BY WK-THR-EAST.  <* 東地区の処理
    START "DO-WEST"  IDENTIFIED BY WK-THR-WEST.  <* 西地区の処理
    START "DO-SOUTH" IDENTIFIED BY WK-THR-SOUTH. <* 南地区の処理
    START "DO-NORTH" IDENTIFIED BY WK-THR-NORTH. <* 北地区の処理
    WAIT FOR WK-THR-EAST.  ADD WK-CTR-EAST  TO WK-COUNTER.
    WAIT FOR WK-THR-WEST.  ADD WK-CTR-WEST  TO WK-COUNTER.
    WAIT FOR WK-THR-SOUTH. ADD WK-CTR-SOUTH TO WK-COUNTER.
    WAIT FOR WK-THR-NORTH. ADD WK-CTR-NORTH TO WK-COUNTER.
    DISPLAY WK-COUNTER.
    STOP RUN.

    ENTRY "DO-EAST".
    MOVE ZERO TO WK-CTR-EAST.
    PERFORM 1000000000 TIMES ADD 1 TO WK-CTR-EAST. GOBACK.
    ENTRY "DO-WEST".
    MOVE ZERO TO WK-CTR-WEST.
    PERFORM 1000000000 TIMES ADD 1 TO WK-CTR-WEST. GOBACK.
    ENTRY "DO-SOUTH".
    MOVE ZERO TO WK-CTR-SOUTH.
    PERFORM 1000000000 TIMES ADD 1 TO WK-CTR-SOUTH. GOBACK.
    ENTRY "DO-NORTH".
    MOVE ZERO TO WK-CTR-NORTH.
    PERFORM 1000000000 TIMES ADD 1 TO WK-CTR-NORTH. GOBACK.
```

START 文はスレッドを新規生成してそのスレッドポインタを返します。スレッドポインタは USAGE THREAD-POINTER で宣言された COBOL のデータ項目でありスレッドの識別子を格納します。WAIT FOR 文でこの識別子を指定して特定のスレッドの終了を待ち合わせることができます。
上記の両方を 4 コア搭載の Linux マシン上で実行してみました。その結果を以下に示します。

スレッドを使用しない逐次実行のケース

```
$ time ./test1
4000000000

real    0m11.002s
user    0m10.957s
sys     0m0.014s
```

スレッドを使用した並列実行のケース

```
$ time ./test2
4000000000
```

```
real    0m5.611s
user    0m11.090s
sys     0m0.026s
```

このようにレスポンスタイムが約半分に短縮されていることがわかります。これはスレッドによる並列処理の結果4個のCPUコアをフル稼働させることができたことを示しています。オペレーティングシステムによるスレッドコンテキストのスイッチング処理のオーバーヘッドがありますので決して4分の1にはならないことに注意してください。

3.2 スレッド固有メモリ領域

COBOLのWORKING-STORAGE SECTIONで宣言されたデータ項目はJavaや.NETではstatic属性に相当するオブジェクトであり、プロセス内にはただ一つだけ存在します。複数のスレッドで同じCOBOLプログラムが実行されるとそれらの間でWORKING-STORAGE SECTIONのデータ項目は共有されます。
以下の例題を見てみましょう。

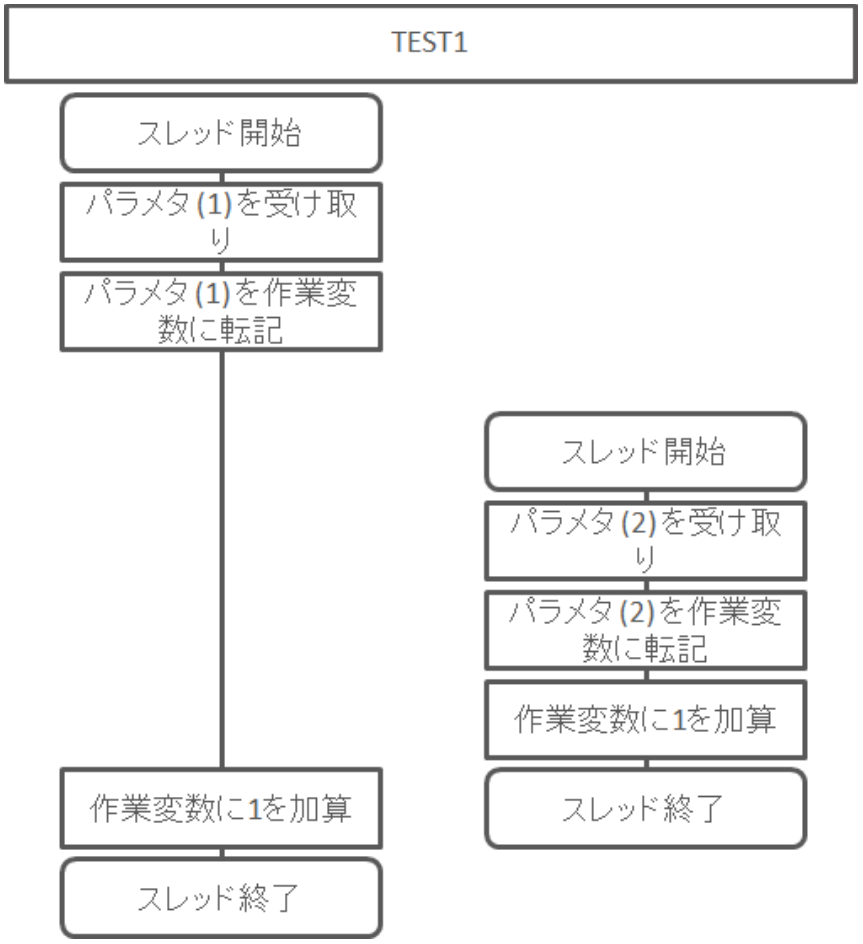
TEST1.cbl

```
WORKING-STORAGE SECTION.
01  THREAD-NO  PIC 99.
PROCEDURE DIVISION.
    MOVE 1 TO THREAD-NO.
    START "THREAD1" USING BY CONTENT THREAD-NO.
    CALL "Sleep" USING BY VALUE 300.
    MOVE 2 TO THREAD-NO.
    START "THREAD1" USING BY CONTENT THREAD-NO.
    STOP RUN.
```

THREAD1.cbl

```
WORKING-STORAGE SECTION.
01  WK-COUNTER  PIC 99.
LINKAGE SECTION.
01  THREAD-NO  PIC 99.
PROCEDURE DIVISION USING THREAD-NO.
    MOVE THREAD-NO TO WK-COUNTER.
    IF THREAD-NO = 1
      CALL "Sleep" USING BY VALUE 600
    END-IF.
    ADD 1 TO WK-COUNTER.
    DISPLAY "結果: " WK-COUNTER " (スレッド " THREAD-NO ")".
    GOBACK.
```

TEST1はTHREAD1というCOBOLプログラムをSTART文を使用して2つのスレッドとして生成しています。その際のUSING BY CONTENT句は、スレッド生成時のパラメタをスレッド個別の領域にコピーして渡すことを示しています。この例では数値の1と2を順番に渡しています。また、2回のSTART文の間にはWindows APIのSleep関数の呼び出しが入っており、0.3秒間待ち合わせるようにしてあります。
THREAD1は受け取ったパラメタをWORKING-STORAGE SECTIONで宣言した作業項目にコピーしてそれに1を加算して結果を表示しています。ただし最初に起動されたスレッドに関しては、作業項目にコピーしてから1を加算するまでの間にSleep呼び出しによって0.6秒の待ち合わせを入れてあります。
この結果この例題は以下のような時系列で動作します。



期待される結果は、パラメタ 1 を受け取ったスレッドはそれに 1 を加算して 2 が表示され、パラメタ 2 を受け取ったスレッドはそれに 1 を加算して 3 が表示されることです。しかしこれをそのまま実行すると以下のような表示となります。

```
C:¥demo>cobol test1.cbl nognt;
Micro Focus COBOL
Version 2.2.02166 Copyright (C) Micro Focus 1984-2014. All rights reserved.
* チェック終了：エラーはありません

C:¥Users¥jk¥Desktop¥demo01>cobol THREAD1.cbl nognt;
Micro Focus COBOL
Version 2.2.02166 Copyright (C) Micro Focus 1984-2014. All rights reserved.
* チェック終了：エラーはありません

C:¥demo>run test1.int
結果: 03 (スレッド 02)
結果: 04 (スレッド 01)
C:¥demo>
```

これは WORKING-STORAGE で宣言されている作業変数 THREAD-NO が両方のスレッドで共有されており、2 つ目のスレッドで 1 を加算した結果が最初のスレッドにも有効になってしまうために発生している事象です。
この問題を解決するには作業変数 THREAD-NO を THREAD-LOCAL-STORAGE SECTION で宣言します。

```
THREAD-LOCAL-STORAGE SECTION.
01  WK-COUNTER  PIC 99.
```

これによって作業変数 THREAD-NO は両方のスレッドでそれぞれ個別に持つようになります。この結果以下の期待される実行結果となります。

```
C:¥demo>run test1.int
結果: 03 (スレッド 02)
結果: 02 (スレッド 01)
C:¥demo>
```

COBOL プログラムの書き方によっては、上記のようなスレッド間の共有を心配しなくてもよいデータ項目もあります。定数のように固定の値を保持し続けるようなものがその一例です。こういったものまで THREAD-LOCAL-STORAGE に移動すると無駄なメモリを消費することになります。しかし、既存の COBOL プログラムを新たにマルチスレッド環境で再利用する際に、どの変数がスレッド間で共有されてはいけないのかを特定することは困難な作業となります。このような場合のために、Visual COBOL ではコンパイル指令 REENTRANT(2) を提供しています。この指令を使用すると WORKING-STORAGE で宣言されている項目をすべて暗黙的に THREAD-LOCAL-STORAGE で宣言されているものとしてコンパイルします。このため、上記の例題は作業変数が WORKING-STORAGE で宣言されたままの状態で以下のように期待されたように動作させることができます。

```
C:¥demo>cobol test1.cbl NOGNT;
Micro Focus COBOL
Version 2.2.02166 Copyright (C) Micro Focus 1984-2014. All rights reserved.
* チェック終了：エラーはありません

C:¥Users¥jk¥Desktop¥demo01>cobol THREAD1.cbl NOGNT REENTRANT(2);
Micro Focus COBOL
Version 2.2.02166 Copyright (C) Micro Focus 1984-2014. All rights reserved.
* チェック終了：エラーはありません

C:¥demo>run test1.int
結果: 03 (スレッド 02)
結果: 02 (スレッド 01)
C:¥demo>
```

この方法では、メモリの消費量が増加するためマルチスレッド処理の一つの利点であるリソース削減には寄与しませんが、CPU 資源をフル活用した並列処理は実現できるようになります。

3.3 同期方式の実現

Visual COBOL では、プログラムを SERIAL 指令でコンパイルするとそのプログラムは同時に複数のスレッド上で実行することが許可されず後続の呼び出しは待たされるようになります。このため WORKING-STORAGE で宣言された項目を使用していても、複数のスレッドで同時に更新することがなくなるため 3.2 で述べたような問題を回避することができます。
3.2 の例題をこの指令を使用してコンパイルして実行すると、以下のように期待された結果を表示します。

```
C:¥demo>cobol TEST1.cbl NOGNT;
Micro Focus COBOL
Version 2.2.02166 Copyright (C) Micro Focus 1984-2014. All rights reserved.
* チェック終了：エラーはありません

C:¥Users¥jk¥Desktop¥demo01>cobol THREAD1.cbl NOGNT SERIAL;
Micro Focus COBOL
Version 2.2.02166 Copyright (C) Micro Focus 1984-2014. All rights reserved.
* チェック終了：エラーはありません

C:¥demo>run test1.int
```

```
結果: 02 (スレッド 01)
結果: 03 (スレッド 02)
C:¥demo>
```

この場合は例題が以下のような時系列で動作しています。

この方法では、スレッド固有メモリ領域を使用する方法と比較すると、メモリ消費を節約することができますが、一方で CPU 資源を最大限に活用できるというマルチスレッド処理の利点を十分に活かすことができないという制限があります。

4 JAVA, .NET のマルチスレッド環境での COBOL の利用

Micro Focus Visual COBOL を使えば、レガシー、オープンレガシー環境で稼働する COBOL プログラムをそのまま .NET Framework や JVM 上でクラスライブラリとしてリユースすることが可能です。しかし、第３章で述べたように従来の COBOL はマルチスレッドで実行されることを予期していないため、同章で紹介したような技術がコーディングに反映されていません。.NET や JVM 上へ移行する際にマルチスレッドを意識したかたちにリファクタリングすることも可能ですが、それに伴う修正・テスト工数を鑑みますとインパクトは小さくありません。Visual COBOL はマルチスレッドによる実行を意識してコーディングされていない COBOL プログラムであっても、.NET の CLR 上及び JVM 上でスレッド毎に固有のメモリ領域を割り当てる技術を用意しています。本章では .NET Managed COBOL 開発及び COBOL for JVM 開発で利用可能な当該技術について解説いたします。この技術はスレッド毎のメモリ領域割り当てに限定されたものではありません。例えば Web Application から COBOL プログラムを呼び出す場合に COBOL で使用するメモリ領域をセッション毎に割り当てるよう応用することも可能です。この技法については別に公開していますホワイトペーパ「COBOL - .NET プログラミング言語の新たな選択肢」をご参照ください。

4.1 .NET CLR 上におけるスレッド毎のメモリ分離

従来の記法でコーディングされた COBOL プログラムはマルチスレッドによる実行を意識してコーディングされていません。そのためこのようなプログラムをマルチスレッド環境より呼び出した場合、プロセス内で単独のファイルロックを利用し、またメモリ領域も各スレッドで共通の領域を参照するため、予期しないエラーや意図しない結果が返ってくることが予想されます。Visual COBOL では .NET Managed COBOL 開発向けに MicroFocus.COBOL.RuntimeServices というアセンブリを提供しています。ここに含まれる RunUnit クラスを介して COBOL プログラムを呼び出せば、その RunUnit 毎にアドレススペース、ファイルロック、OpenESQL 接続を割り当てることが可能です。つまり、スレッド毎に RunUnit を起動し COBOL を呼び出せばこれらをスレッド単位で分離することを実現できます。

まずは簡単な例題を使って、マルチスレッドにより実行を意識されて開発されていない COBOL をマルチスレッドで実行した場合の問題点を見てみます。下記のように初期値 0 の「AAA」という変数を５回カウントアップして、各カウントアップの結果を DISPLAY 表示するプログラムを考えてみます。例示の便宜上実行時のスレッド ID を引数として受取りそれをカウントアップとともに表示するロジックも加えています。

```
PROGRAM-ID. COBPGM AS "COBCLASS.COBPGM".
DATA DIVISION.
WORKING-STORAGE SECTION.
01  AAA    PIC 9(3) VALUE ZERO.
01  THREADNUM-DISP  PIC 9(1) VALUE ZERO.
PROCEDURE DIVISION USING BY VALUE THREADNUM AS binary-long.
    MOVE THREADNUM TO THREADNUM-DISP.
    PERFORM 5 TIMES
        COMPUTE AAA = AAA + 1
        DISPLAY "FROM THREAD: " THREADNUM-DISP ", VALUE: " AAA
    END-PERFORM.
    GOBACK.
END PROGRAM COBPGM.
```

これを下記のようなパラレルでマルチスレッド実行するプログラムから呼び出してみます。各スレッド中で COBOL プログラムはインスタンス化され、スレッド ID を引数として COBOL プログラムに指定し呼び出します。

```
class mthreaddemo
{
    private COBCLASS.COBPGM cobmod;
    void ParallelizedMethod(int id)
    {
        cobmod = new COBCLASS.COBPGM();
        Console.WriteLine("Thread ID:{0} starts", id);
```

```
            cobmod.COBPGM(id);
            Console.WriteLine("Thread ID:{0} ends", id);
        }

        public void go()
        {
            Parallel.For(0, 3, ParallelizedMethod);
        }
    }
```

すると下図のように各スレッドでプロセス内で単独の WORKING-STORAGE SECTION に定義された「AAA」を参照するため、正しくカウントアップされません。

```
file:///C:/work/WhitePaper/MultiThread/dotNET/Test/Csh
Thread ID:0 starts
Thread ID:1 starts
FROM THREAD: FROM THREAD: 1, VALUE: 002
FROM THREAD: 1, VALUE: 003
FROM THREAD: 1, VALUE: 004
FROM THREAD: 1, VALUE: 005
FROM THREAD: 1, VALUE: 006
Thread ID:1 ends
Thread ID:2 starts
FROM THREAD: 2, VALUE: 001
FROM THREAD: 2, VALUE: 002
FROM THREAD: 2, VALUE: 003
FROM THREAD: 2, VALUE: 004
FROM THREAD: 2, VALUE: 005
Thread ID:2 ends
1, VALUE: 006
FROM THREAD: 1, VALUE: 007
FROM THREAD: 1, VALUE: 008
FROM THREAD: 1, VALUE: 009
FROM THREAD: 1, VALUE: 010
Thread ID:0 ends
```

続いて、COBOL プログラムを呼び出す C# プログラム中のパラレル実行されるメソッド「ParallelizedMethod」を下記のように書き換えてみます。

```
void ParallelizedMethod(int id)
{
    MicroFocus.COBOL.RuntimeServices.RunUnit CobRunUnit
        = new MicroFocus.COBOL.RuntimeServices.RunUnit();
    try
    {
        cobmod = new COBCLASS.COBPGM();
        Console.WriteLine("Thread ID:{0} starts", id);
        CobRunUnit.Add(cobmod);
        cobmod.COBPGM(id);
        Console.WriteLine("Thread ID:{0} ends", id);
    }
    finally
    {
        CobRunUnit.StopRun();
    }

}
```

RunUnit をスレッド毎に用意します

COBOL のインスタンスを RunUnit に割り当てます

実行後は RunUnit を破棄します

このプログラムを実行すると WORKING-STORAGE SECTION 中の変数も各スレッドで分離されているため、下図のように各スレッドで１から５までカウントアップされます。

```
file:///C:/work/WhitePaper/MultiThread/dotNE
Thread ID:0 starts
FROM THREAD: 0, VALUE: 001
FROM THREAD: 0, VALUE: 002
FROM THREAD: 0, VALUE: 003
FROM THREAD: 0, VALUE: 004
FROM THREAD: 0, VALUE: 005
Thread ID:0 ends
Thread ID:2 starts
FROM THREAD: 2, VALUE: 001
FROM THREAD: 2, VALUE: 002
FROM THREAD: 2, VALUE: 003
FROM THREAD: 2, VALUE: 004
FROM THREAD: 2, VALUE: 005
Thread ID:2 ends
Thread ID:1 starts
FROM THREAD: 1, VALUE: 001
FROM THREAD: 1, VALUE: 002
FROM THREAD: 1, VALUE: 003
FROM THREAD: 1, VALUE: 004
FROM THREAD: 1, VALUE: 005
Thread ID:1 ends
```

4.2　JVM 上におけるスレッド毎のメモリ分離

マルチスレッド実行が意識されていない従来の記法でコーディングされた COBOL プログラムを COBOL for JVM の技術を使って JVM 上で実行する場合も .NET の例で見たようにスレッド毎にメモリが分離されていないことによる問題は同様に発生し得ます。COBOL for JVM で JVM クラスを COBOL で開発する場合においても上の項で確認した RunUnit の技術が **com.microfocus.cobol.runtimeservices.RunUnit** より提供されています。

本技術を利用すれば、JVM 上でマルチスレッドで実行される Java プログラムより従来の記法でコーディングされた COBOL を呼び出す際もスレッド毎のメモリ分離を実現できます。

ここでは、まず下記のような COBOL プログラムを考えてみます。このプログラムではファイルを OPEN EXTEND して１レコードをファイルに追記して閉じています。レコードの先頭カラムには例示の便宜上の理由から引数渡しされたスレッド番号を挿入しています。

```
        ENVIRONMENT DIVISION.
        INPUT-OUTPUT SECTION.
        SELECT AFILE ASSIGN TO "AFILE.dat".
        DATA DIVISION.
        FILE SECTION.
        FD AFILE.
        01 AFILE-REC.
           03 REC-THREAD-ID    PIC 9(2).
           03 REC-VALUE        PIC X(8).
        PROCEDURE DIVISION USING BY VALUE THREADNUM AS binary-long.
            DISPLAY "PGM CALLED: " THREADNUM.
            OPEN EXTEND AFILE.
            MOVE THREADNUM TO REC-THREAD-ID.
            MOVE ALL "A" TO REC-VALUE.
            WRITE AFILE-REC.
            CLOSE AFILE.
            GOBACK.
```

これをまずは、下記のような Java プログラムから３つのスレッドを使って並行呼び出ししてみます。

```
public class MthreaddemoJ {
        public static void main(String[] args) {
                COBFILE cobmod1 = new COBFILE();
                COBFILE cobmod2 = new COBFILE();
                COBFILE cobmod3 = new COBFILE();

                ThreadClass thread1 = new ThreadClass(0, cobmod1);
                ThreadClass thread2 = new ThreadClass(1, cobmod2);
                ThreadClass thread3 = new ThreadClass(2, cobmod3);
                Thread t1 = new Thread(thread1);
                Thread t2 = new Thread(thread2);
                Thread t3 = new Thread(thread3);
                t1.start();
                t2.start();
                t3.start();
        }
}

class ThreadClass implements Runnable {
        COBFILE cobmod;
        int threadnum;

        ThreadClass(int tid, COBFILE cobmod){
                this.cobmod = cobmod;
                this.threadnum = tid;
        }

        public void run(){
```

```
                System.out.println(MessageFormat.format
                                        ("Thread ID:{0} starts", threadnum));
                this.cobmod.COBFILE(threadnum);
                System.out.println(MessageFormat.format("Thread ID:{0} ends", threadnum));
        }
}
```

更にここでは、Open の際にロックを取得できなかった際に RETRY させるファイルハンドラ構成オプションを有効にします。

```
[XFH-DEFAULT]
RETRYOPEN=10
RETRYTIME=ON
```

全てのファイルを対象に Open の RETRY を 10 秒間繰り返すよう構成しています

この Java プログラムを実行すると下図のようにそもそもマルチスレッドによる実行自体ができません。

```
問題 | @ Javadoc | 宣言 | コンソール
<終了> MthreaddemoJ [Java アプリケーション] C:¥Program Files (x86)¥Java¥jdk1.7.0_60¥bin¥javaw.exe (20:
com.microfocus.cobol.program.COBOLRuntimeException: 119  名前が重複している [COBFILE]
        at com.microfocus.cobol.program.Control.programEnter(Unknown Source)
        at COBFILE.COBFILE(COBFILE.cbl:65535)
        at ThreadClass.run(MthreaddemoJ.java:40)
        at java.lang.Thread.run(Thread.java:745)
com.microfocus.cobol.program.COBOLRuntimeException: 119  名前が重複している [COBFILE]
        at com.microfocus.cobol.program.Control.programEnter(Unknown Source)
        at COBFILE.COBFILE(COBFILE.cbl:65535)
        at ThreadClass.run(MthreaddemoJ.java:40)
        at java.lang.Thread.run(Thread.java:745)
Thread ID:0 starts
Thread ID:1 starts
Thread ID:2 starts
PGM CALLED: +0000000000
Thread ID:0 ends
```

それでは、COBOL を呼び出すスレッド処理が書かれた Runnable インターフェースの抽象メソッドである Run メソッドを下記のように改造します。ここでは、ファイルのロックやメモリ参照をスレッド単位で独立させるよう COBOL プログラムを RunUnit 経由で実行させています。

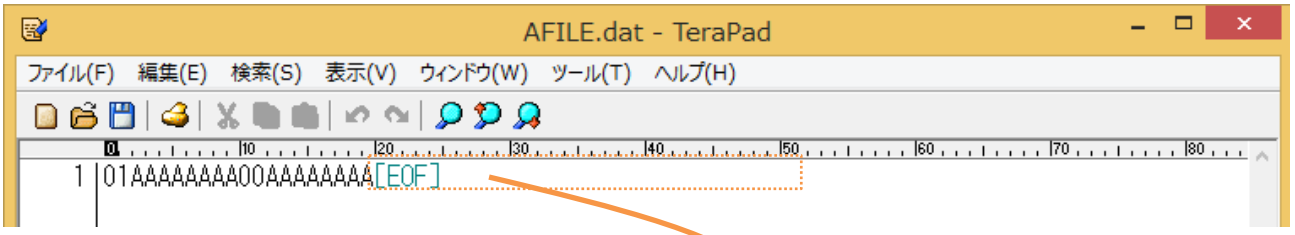
```
        public void run(){
                System.out.println(MessageFormat.format
                                                ("Thread ID:{0} starts", threadnum));

                com.microfocus.cobol.runtimeservices.RunUnit cobrunit =
                                new com.microfocus.cobol.runtimeservices.RunUnit();
                try {
                        cobrunit.Add(this.cobmod);
                        this.cobmod.COBFILE(threadnum);
                } finally {
                        cobrunit.StopRun();
                }
                System.out.println(MessageFormat.format("Thread ID:{0} ends", threadnum));
        }
```

これを実行すると下図のように改造前と違い正常に処理が終了します。

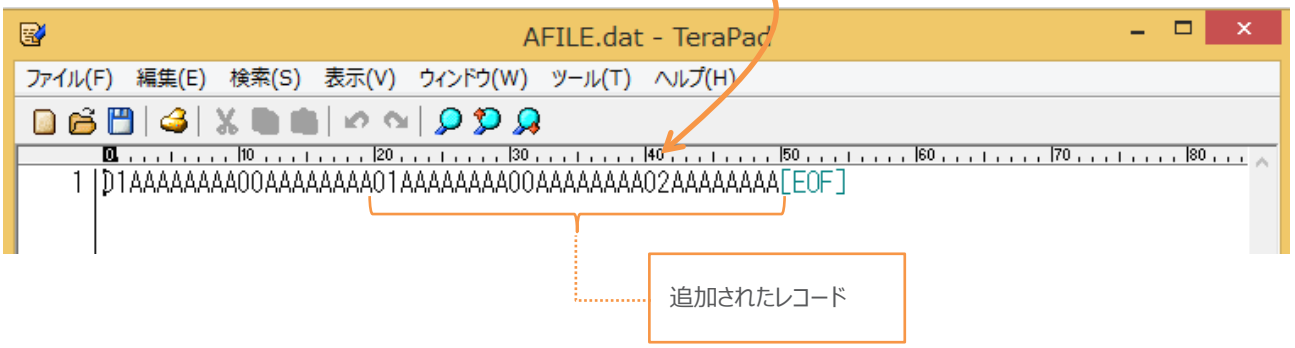
```
問題  @ Javadoc  宣言  コンソール
<終了> MthreaddemoJ [Java アプリケーション] C:¥Program Files (x86
Thread ID:0 starts
Thread ID:1 starts
Thread ID:2 starts
PGM CALLED: +0000000001
PGM CALLED: +0000000002
PGM CALLED: +0000000000
Thread ID:1 ends
Thread ID:0 ends
Thread ID:2 ends
```

また、COBOL プログラムが参照するファイルを確認すると意図した通りにスレッド単位でレコードが追記されており、マルチスレッドで並行実行しているにも関わらず正しく排他制御できていることがわかります。

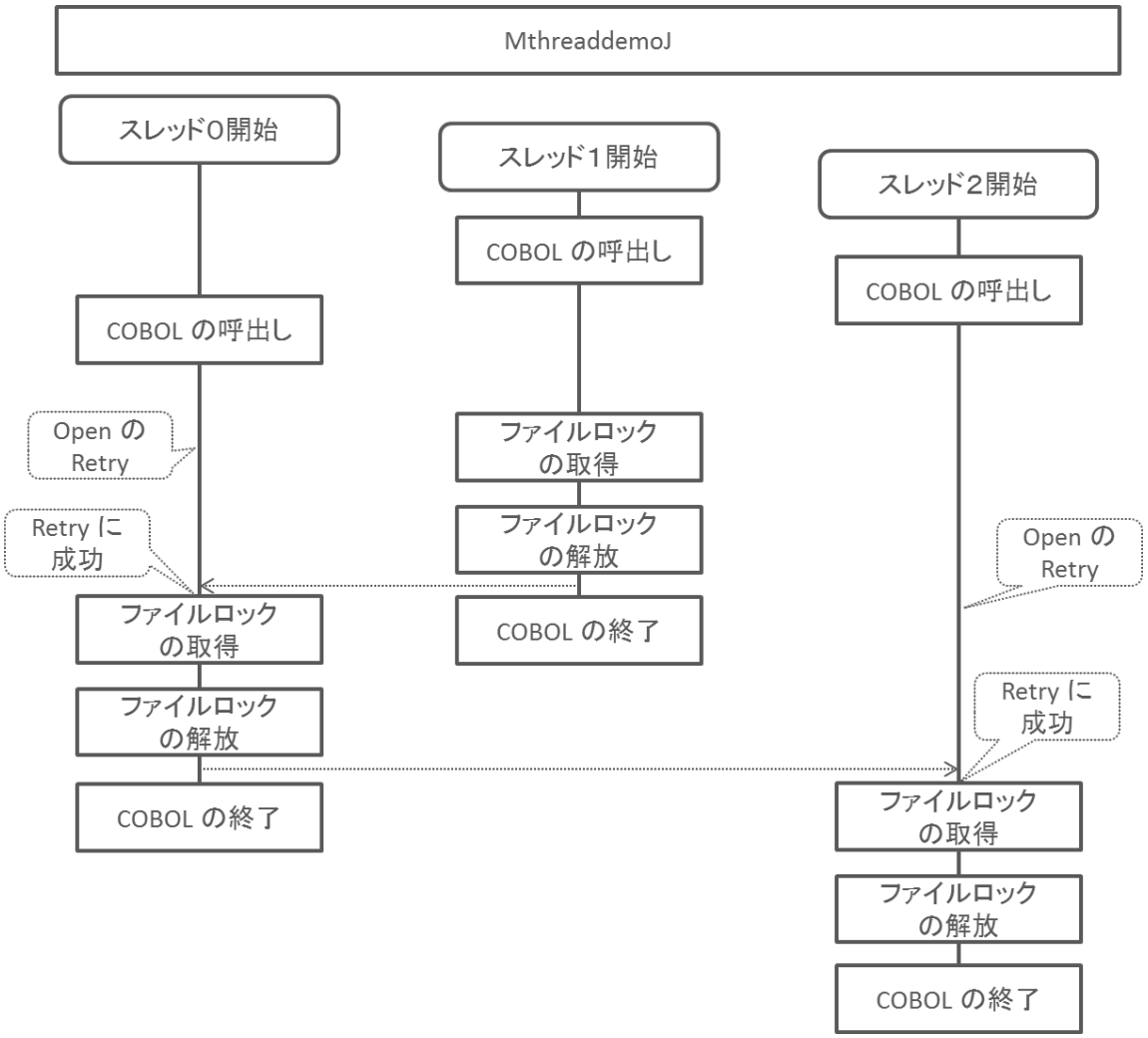
実行前



実行後



以上の結果からこの例題は内部的には以下のようなフローで処理されていたことがわかります。マルチスレッドにより並列に処理が走ってはいますが、COBOL プログラムを割り当てた RunUnit 毎にファイルのロックが存在するため、排他の仕組みが働いています。これが仮に単一のロックを全スレッドで共有するということになりますと、スレッド１のファイルオープンは成功しますが、スレッド０でこのファイルをオープンする際に既にロックが取得できているにも関わらずファイルを２重にオープンしようとしているということで OPEN 文の箇所で実行時エラーとなります。



4.3 マルチスレッド Java アプリケーションからのネイティブ COBOL アプリケーションの利用

4.1、4.2 ではマルチスレッドアプリケーションにて .NET の MSIL コードや javabyte コードにコンパイルされた COBOL アプリケーションを利用する場合の技法を紹介いたしました。ここでは、COBOL をネイティブアプリケーションとして運用する場合の対応策を考えてみます。例えば、JSP/Servlet のような Web Application の Servlet コンテナは基本的に Servlet のインスタンスを 1 つしか作成しませんが、マルチアクセスによる同時実行が求められるため、マルチスレッドアプリケーションとして動作します。そのため、Servlet から呼ばれるメソッド等はスレッドセーフである必要があります。Java とネイティブの COBOL アプリケーションを連携させることを考えた場合、com.microfocus.cobol.RuntimeSystem クラスを使って JNI 呼び出しする方法や COBOL 専用の Application Server 「Enterprise Server」にホスティングした COBOL アプリケーションを Java Connector Architecture に従って呼び出す方法が挙げられます。後者に関しては JVM と異なるプロセスで動作するため、この点の考慮は不要です。しかし、前者に関しては JVM と同じプロセスで動作するため、呼ばれる COBOL 側もスレッドセーフな作りにする必要があります。つまり、Servlet から利用される COBOL プログラムを開発する際は、3.2 及び 3.3 で上述したようなメモリ領域の分離や同期方式を意識して設計する必要があります。

JNI(Java Native Interface) は Java 言語から Java 以外のネイティブアプリケーションを呼び出すための技術です。JNI による呼び出しは通常複雑なコーディングが要求されますが、Visual COBOL で開発したアプリケーションに関しては製品同梱される mfcobol.jar にパッケージされた各種クラスを使って簡単にコーディングする仕組みが提供されています。以下は Servlet 中にて JNI による COBOL の呼出しを記述した例になります [1]。ここではダイナミックリンクライブラリにビルドされた COBOL アプリケーション CountUp.dll を Servlet クラスから呼び出しています。COBOL に渡すパラメータはセッション毎に独立しています。

```
import com.microfocus.cobol.RuntimeSystem;
import com.microfocus.cobol.lang.ParameterList;

public class CallCOBOL extends HttpServlet {
    int outnum;
    int currentnum;
    ParameterList parms;
                                        :
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
                                                throws ServletException, IOException {
        HttpSession session = request.getSession(false);
         if (session == null){
            session = request.getSession(true);
            session.setAttribute("countnum", "0");
        }else{
            try{
                currentnum = Integer.parseInt(session.getAttribute("countnum").toString());
                parms = new ParameterList();
                parms.add(currentnum);
                outnum = RuntimeSystem.cobcall("CountUp", parms);
                  session.setAttribute("countnum", Integer.toString(outnum));
            } catch(Exception e) {
            }
        }
        RequestDispatcher dispatcher = request.getRequestDispatcher("/CountUp.jsp");
        dispatcher.forward(request, response);
    }
                                        :
```

COBOL に渡すパラメータを準備しています。

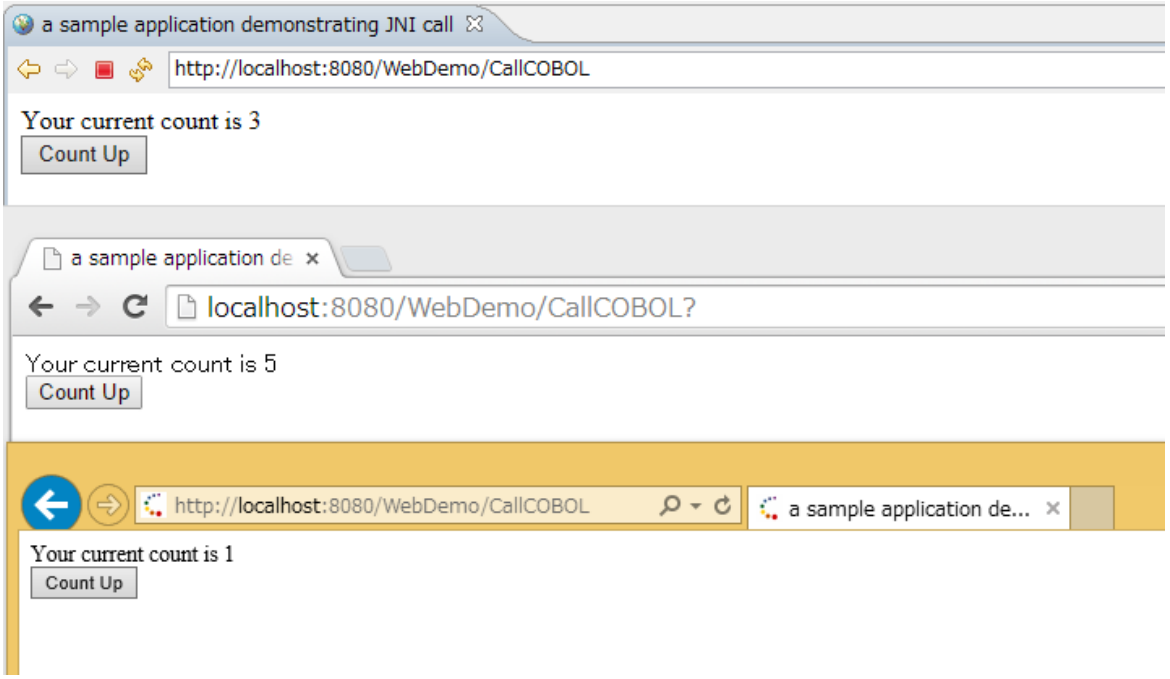
COBOL を呼び出しています。

[1] JNI を使った COBOL との連携技術に関するより詳細な情報は製品マニュアルや別 White Paper 冊子「Micro Focus による COBOL と Java の融合」を参照してください。

上の Servlet から呼ばれる COBOL プログラムのコーディング例です。WORKING-STORAGE SECTION に定義された領域は各スレッドで競合しないように REENTRANT(2) 指令を指定してスレッド毎に領域を割り当てるか、SERIAL 指令を指定して排他で処理させるかの対策が必要となります。ここでは１行目に $SET SERIAL を記述し、スレッド間の排他・同期制御を図っています。プログラムは引数で受け取った数値項目を１つカウントアップしその結果を返す単純なものです。

```
$SET SERIAL
 IDENTIFICATION DIVISION.
 PROGRAM-ID. "CountUp".
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 COPY JAVATYPES.
 01 RETVAL          jint.
 LINKAGE SECTION.
 01 INVAL           jint.
 PROCEDURE DIVISION USING INVAL.
 1.
     COMPUTE RETVAL = INVAL + 1.
     EXIT PROGRAM  RETURNING RETVAL.
```

この JSP/Servlet Application を複数のクライアントから実行すると、クライアント毎にそれぞれ０からカウントアップします。セッション単位で処理が分離されているため、下図のようにクライアント毎に独自にカウントアップします。実際にカウントアップする COBOL アプリケーションはスレッドセーフに開発されているため、このようなマルチユーザによる同時アクセスの可能性があるアプリケーションでも正常に動作します。



5 マルチスレッド環境でのデータベースアクセス

データベース連携アプリケーションを構築する場合、データベースへの接続やカーソル等のリソースの扱いを考慮に入れて設計します。それがマルチスレッドアプリケーションとなりますとそれらについては更に慎重な設計が必要となります。これまで見てきたようにマルチスレッドアプリケーションはアドレススペースを共有しつつ仮想的に並行実行されるアプリケーションです。この共有されるエリアを意識せずにかつてシングルスレッドで実行していたプログラムをマルチスレッドで実行すると思わぬ落とし穴に陥ることがあります。例えば、接続が共有エリアを使って管理されるにも関わらずマルチスレッドでそのまま実行すると、最初に接続要求を発行するスレッドは接続に成功するかもしれませんが、他のスレッドでも続けて接続要求を発行すると既に開かれている接続に対して接続要求しようとするので成功せずアプリケーションが終了する可能性もあります。本章では COBOL によるデータベース連携アプリケーション開発で利用される代表的な技術 OpenESQL、Pro*COBOL を使ってマルチスレッドアプリケーション開発する際のポイントを取り上げます。

5.1 OpenESQL の ODBC 機能を使ったマルチスレッドアプリケーション開発

Micro Focus Visual COBOL に付属する OpenESQL は ODBC、ADO.NET、JDBC を通じたデータベースアクセスを提供する統合化プリプロセッサです。この統合化プリプロセッサはスレッドセーフに設計されているため、接続やカーソル等のリソースに対して排他制御、同期制御するようプログラマが敢えてコーディングする必要はありません。OpenESQL の ODBC 連携機能には **THREAD** という OpenESQL のコンパイラオプションがあり、これを使って、OpenESQL で使用する接続やカーソル等のリソースをスレッド間で共有する、或いは個々で持たせるか制御することができます。THREAD オプションのデフォルト値は「SHARE」です。この場合、全ての OpenESQL のリソースが全スレッドで共有されます。従いまして、個々のスレッドで固有の接続を確立しその接続を利用して独立した環境下で何かデータ操作をするといった処理ができません。Visual COBOL にはこのような接続やカーソル等といった OpenESQL のリソースをスレッド間で分離させるためオプションが用意されています。これを使えば、スレッド間で OpenESQL リソースの排他を意識する必要がありません。この分離を実現するには THREAD オプションに「**ISOLATE**」を指定します。（このオプションを指定すると OpenESQL はスレッド毎に 72K バイトのオーバーヘッドを消費します。）

ここではスレッド間で OpenESQL リソースを分離する簡単な例を考えてみます。以下のようにスレッドを２つ生成して同じプログラムをそこで実行させるメインプログラムがあります。同じプログラムを２スレッドで実行させますが、それぞれに対してスレッドを識別させるための１バイトのデータを渡します。マルチスレッドで並行処理されるプログラムはプログラム中からデータベースへの接続を確立し、１０件のレコードをカーソル・フェッチで取り出します。カーソルはそれぞれ同じテーブルを参照しますが引数渡しされるデータに応じて範囲を変更させます。このプログラムは OpenESQL オプション THREAD に ISOLATE を指定しているため、それぞれ独立した接続が確立され、カーソルも同名で定義していますが、それぞれ異なるカーソルとして扱われます。

メインプログラム

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 THREAD-ID1          PIC X(01) VALUE "A".
01 THREAD-ID2          PIC X(01) VALUE "B".
01 WK-THREAD1          USAGE THREAD-POINTER.
01 WK-THREAD2          USAGE THREAD-POINTER.
PROCEDURE DIVISION.
    START "ODBCSUB" USING THREAD-ID1
                    IDENTIFIED BY WK-THREAD1.
    START "ODBCSUB" USING THREAD-ID2
                    IDENTIFIED BY WK-THREAD2.
    GOBACK.
```

サブプログラム

```
$SET SQL(DBMAN=ODBC THREAD=ISOLATE) REENTRANT(2)
 DATA DIVISION.
 WORKING-STORAGE SECTION.
      EXEC SQL INCLUDE SQLCA END-EXEC.
      EXEC SQL BEGIN DECLARE SECTION END-EXEC.
      EXEC SQL INCLUDE Person END-EXEC.
      EXEC SQL END DECLARE SECTION END-EXEC.
 01 DISP-ID              PIC ZZ9.
 01 DISP-FIRSTNAME       PIC X(20) VALUE SPACE.
 01 DISP-LASTNAME        PIC X(20) VALUE SPACE.
 LINKAGE SECTION.
 01 THREAD-ID            PIC X(01).
 PROCEDURE DIVISION USING THREAD-ID.
     EXEC SQL CONNECT TO 'ZZZ' USER 'xxx.yyy' END-EXEC.
     DISPLAY "THREAD-ID: " THREAD-ID " connection established"

     EVALUATE THREAD-ID
     WHEN "A"
        MOVE 1  TO ID-S
        MOVE 10 TO ID-E
     WHEN "B"
        MOVE 11 TO ID-S
        MOVE 20 TO ID-E
     END-EVALUATE.

     EXEC SQL
          DECLARE PERSONCSR CURSOR FOR SELECT
              A.BusinessEntityID ,A.FirstName ,A.LastName
          FROM Person.Person A
          WHERE A.BusinessEntityID BETWEEN :ID-S AND :ID-E
     END-EXEC.
     EXEC SQL  OPEN PERSONCSR END-EXEC.
     DISPLAY "THREAD-ID: " THREAD-ID " cursor opened"

     PERFORM UNTIL SQLSTATE >= "02000"
         EXEC SQL
             FETCH PERSONCSR  INTO
             :Person-BusinessEntityID
            ,:Person-FirstName ,:Person-LastName
         END-EXEC
         IF SQLSTATE < "02000"
            MOVE Person-BusinessEntityID TO DISP-ID
            MOVE FUNCTION DISPLAY-OF(Person-FirstName)
                  TO DISP-FIRSTNAME
            MOVE FUNCTION DISPLAY-OF(Person-LastName)
                  TO DISP-LASTNAME
            DISPLAY "THREAD-ID: " THREAD-ID
                    " " DISP-ID ", " DISP-FIRSTNAME
                    DISP-LASTNAME
         END-IF
     END-PERFORM.

     EXEC SQL CLOSE PERSONCSR END-EXEC.
     EXEC SQL DISCONNECT CURRENT END-EXEC.
     GOBACK.
```

OpenESQL のオプション THREAD に ISOLATE を指定します。

プログラム中で利用する変数をスレッドローカルとして扱います。

マルチスレッドで実行するプログラム中から接続処理を発行します。

引数渡しされるデータに応じて参照範囲を定めます。

全スレッド共通のカーソル名でカーソルを定義します。

カーソルをオープンします。カーソル名は同じですが、それぞれ独立した接続に紐づけられた異なるカーソルとして扱われるため、この時点で DB 上ではスレッドに応じた SELECT 文が実行されます。

下図はこの例題の実行結果の例となります。スレッドの切り替えのタイミングは OS の管理下にあるため、実行順は実行の度に変わる可能性はありますが、いずれにしても今回の結果のように複数のスレッドでそれぞれ分離された OpenESQL のリソースを使っておりカーソルで定義したデータをそれぞれ全件参照します。カーソルは同名で定義していますが異なる接続と紐づいているため、独立したカーソルとして扱われます。例えば、複数のパーティションにまたがる単一のテーブルを全件参照するようなアプリケーションがある場合、このように複数のスレッドでそれぞれ異なるパーティションに格納されたレコードを並列で参照するようにするとパフォーマンスが向上する可能性もあります。

```
"C:¥Program Files (x86)¥Micro Focus¥Visual COBOL¥bin64¥run.exe" ..
THREAD-ID: A connection established
THREAD-ID: A cursor opened
THREAD-ID: A   1, Ken                Sanchez
THREAD-ID: A   2, Terri              Duffy
THREAD-ID: A   3, Roberto            Tamburello
THREAD-ID: A   4, Rob                Walters
THREAD-ID: A   5, Gail               Erickson
THREAD-ID: A   6, Jossef             Goldberg
THREAD-ID: A   7, Dylan              Miller
THREAD-ID: B connection established
THREAD-ID: B cursor opened
THREAD-ID: B  11, Ovidiu             Cracium
THREAD-ID: B  12, Thierry            D'Hers
THREAD-ID: B  13, Janice             Galvin
THREAD-ID: B  14, Michael            Sullivan
THREAD-ID: B  15, Sharon             Salavaria
THREAD-ID: B  16, David              Bradley
THREAD-ID: B  17, Kevin              Brown
THREAD-ID: B  18, John               Wood
THREAD-ID: B  19, Mary               Dempsey
THREAD-ID: B  20, Wanida             Benshoof
THREAD-ID: A   8, Diane              Margheim
THREAD-ID: A   9, Gigi               Matthew
THREAD-ID: A  10, Michael            Raheem
```

5.2 OpenESQL の ADO.NET、JDBC 機能を使ったマルチスレッドアプリケーション開発

OpenESQL コンパイラオプション THREAD の ISOLATE は ODBC 機能でのみ提供されているオプションです。COBOL プログラムを .NET のクラスライブラリや JVM クラスとして利用する場合はこのオプションを使った分離は実現できません。しかし、これらにコンパイルして利用するような開発では、Web Application 等セッション単位等で情報を分離する必要があるシーンでの利用が多く想定されます。このようなシーンでマルチスレッドアプリケーションを開発し、スレッド単位に OpenESQL のリソースを分離する必要があれば、前章で紹介した RunUnit を使用します。RunUnit 配下で実行すると接続やカーソル等の OpenESQL のリソースも RunUnit 単位で割り当てられます。

前項で使用した COBOL プログラムを下記のように、引数パラメータの型、接続文、EVALUATE 文の条件だけ変更してそれを .NET のクラスライブラリにコンパイルします。

```
                                        :
        PROCEDURE DIVISION USING BY VALUE THREAD-ID AS BINARY-LONG.
            EXEC SQL
                CONNECT TO 'ZZZ'
            END-EXEC.
            DISPLAY "THREAD-ID: " THREAD-ID " connection established"

            EVALUATE THREAD-ID
            WHEN 0
               MOVE 1  TO ID-S
               MOVE 5  TO ID-E
            WHEN 1
               MOVE 6  TO ID-S
               MOVE 10 TO ID-E
            WHEN 2
               MOVE 11 TO ID-S
               MOVE 15 TO ID-E
            END-EVALUATE.
                                        :
```

LINKAGE SECTION を使わずここで .NET の int に相当する COBOL の型を引数指定します。

認証情報は ADO のデータソースに登録可能なためここでは認証情報は指定していません。

ここでは条件式を３つにして、レコードの検索範囲は５件ずつに変更しています。

これを 4.2 で例示した RunUnit 経由で COBOL をメソッド呼び出しする C# のプログラムから３スレッドで並列呼び出してみます。下図はその出力例となります。この結果をみるとスレッド１はスレッド２が接続を確立した直後に新たに接続を確立しています。RunUnit により OpenESQL のリソースが分離できているからこそこのように同一プロセスより複数の接続要求を発行できています。RunUnit による分離がない場合は、このような同時接続は許可されません。

```
file:///C:/work/WhitePaper/MultiThread/OpenESQL/dotNet/CsharpCo...
THREAD-ID: +0000000000 connection established
THREAD-ID: +0000000000 cursor opened
THREAD-ID: +0000000000   1, Ken                Sanchez
THREAD-ID: +0000000000   2, Terri              Duffy
THREAD-ID: +0000000000   3, Roberto            Tamburello
THREAD-ID: +0000000000   4, Rob                Walters
THREAD-ID: +0000000000   5, Gail               Erickson
Thread ID:0 ends
Thread ID:2 starts
THREAD-ID: +0000000002 connection established
THREAD-ID: +0000000001 connection established
THREAD-ID: +0000000002 cursor opened
THREAD-ID: +0000000002  11, Ovidiu             Cracium
THREAD-ID: +0000000002  12, Thierry            D'Hers
THREAD-ID: +0000000002  13, Janice             Galvin
THREAD-ID: +0000000002  14, Michael            Sullivan
THREAD-ID: +0000000002  15, Sharon             Salavaria
Thread ID:2 ends
THREAD-ID: +0000000001 cursor opened
THREAD-ID: +0000000001   6, Jossef             Goldberg
THREAD-ID: +0000000001   7, Dylan              Miller
THREAD-ID: +0000000001   8, Diane              Margheim
THREAD-ID: +0000000001   9, Gigi               Matthew
THREAD-ID: +0000000001  10, Michael            Raheem
Thread ID:1 ends
```

5.3 Pro*COBOL を使ったマルチスレッドアプリケーション開発

Oracle 社は Oracle Database 接続のネイティブ COBOL アプリケーション開発用に Pro*COBOL という Precompiler を提供しています。同社はこの Pro*COBOL が Micro Focus の COBOL 製品と連携して動作する設計しており、Micro Focus 社の製品は正式に稼働対象コンパイラとして Certify されています。本項ではこの Pro*COBOL が提供するマルチスレッドアプリケーション開発機能について概説いたします。ここで紹介する技術の詳細は Oracle 社が公開する Pro*COBOL Programmer's Guide 等のドキュメントを参照してください。

Pro*COBOL は Oracle Database への接続、カーソル、インライン・オプションを紐づけることができるランタイム・コンテキストという機能を提供しています。マルチスレッドアプリケーションにおいて、このランタイム・コンテキストを例えばスレッド毎に用意して、それぞれに接続を割り当てればスレッド単位で接続を分離することが可能です。つまり同環境下であれば、それぞれに割り当てられた接続等のリソースを利用するため、ミューテックスに関する考慮が不要となります。

以下はマルチスレッドアプリケーションにおいて各スレッドがそれぞれのランタイム・コンテキストに割り当てられた独自の接続を利用して並列処理を実現するコーディング例となります。本例ではまず、メインのプログラムにてランタイム・コンテキストを２つ用意します。そしてそれぞれのランタイム・コンテキストに同一のデータベースに対する独立した接続を割り当てた後にスレッドを２つ起動し異なるサブプログラムをその上で実行させます。この際、異なるスレッドで実行するそれぞれのサブプログラムに独立したランタイム・コンテキスト渡します。これらのサブプログラムはともに同じデータベース上の同じテーブルにある異なるレコードに対してカーソル・フェッチをしてデータを参照します。

メインプログラム

```
$SET REENTRANT(2)
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  PROCOBMAIN.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
     EXEC SQL INCLUDE SQLCA END-EXEC.
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 LOGON-STRING          PIC X(40) VALUE "user/password".
 01 CONTEXT-AREA1         SQL-CONTEXT.
 01 CONTEXT-AREA2         SQL-CONTEXT.
     EXEC SQL END DECLARE SECTION END-EXEC.
 01 WK-THREAD1            USAGE THREAD-POINTER.
 01 WK-THREAD2            USAGE THREAD-POINTER.
 PROCEDURE DIVISION.
     EXEC SQL ENABLE THREADS END-EXEC.
     EXEC SQL CONTEXT ALLOCATE :CONTEXT-AREA1 END-EXEC.
     EXEC SQL CONTEXT ALLOCATE :CONTEXT-AREA2 END-EXEC.

     EXEC SQL CONTEXT USE :CONTEXT-AREA1 END-EXEC.
     EXEC SQL CONNECT :LOGON-STRING END-EXEC.

     EXEC SQL CONTEXT USE :CONTEXT-AREA2 END-EXEC.
     EXEC SQL CONNECT :LOGON-STRING END-EXEC.

     START "PROCOBSUB1" USING CONTEXT-AREA1
                        IDENTIFIED BY WK-THREAD1.

     START "PROCOBSUB2" USING CONTEXT-AREA2
                        IDENTIFIED BY WK-THREAD2.

     GOBACK.
```

REENTRANT(2) を指定し EXTERNAL 属性を指定した項目を除いて変数はスレッドローカルとして扱います。

プロセスがマルチスレッドで実行するための初期化処理をします。

ランタイム・コンテキストをメモリに割り当てます。

ランタイム・コンテキスト「CONTEXT-AREA1」に切り替えます。

ランタイム・コンテキスト「CONTEXT-AREA２」に切り替えます。

ランタイム・コンテキストを渡してスレッドを起動します。

サブプログラム１

```
$SET REENTRANT(2)
 IDENTIFICATION DIVISION.
 PROGRAM-ID. PROCOBSUB1.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
     EXEC SQL INCLUDE SQLCA END-EXEC.
     EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 HV-EMPNO                PIC S9(4) COMP.
 01 HV-ENAME                PIC X(10)  VARYING.
     EXEC SQL END DECLARE SECTION END-EXEC.
 01  DISP-EMPNO             PIC 9(4).
 LINKAGE SECTION.
 01  LNK-CTX                 SQL-CONTEXT.
 PROCEDURE DIVISION USING LNK-CTX.
     EXEC SQL CONTEXT USE :LNK-CTX END-EXEC.

     EXEC SQL
       DECLARE EmpCSR CURSOR FOR
           SELECT EMPNO, ENAME FROM EMP
           WHERE EMPNO BETWEEN 7500 AND 7700
     END-EXEC.
     EXEC SQL OPEN EmpCSR END-EXEC.
     PERFORM UNTIL SQLCODE NOT = 0
         EXEC SQL
             FETCH EmpCSR INTO :HV-EMPNO, :HV-ENAME
         END-EXEC
         IF SQLCODE = 0 THEN
              MOVE HV-EMPNO TO DISP-EMPNO
              DISPLAY DISP-EMPNO ": " HV-ENAME-ARR(1:HV-ENAME-LEN)
         END-IF
     END-PERFORM.

     EXEC SQL CLOSE EmpCSR END-EXEC.
     EXEC SQL COMMIT WORK RELEASE END-EXEC.
     EXEC SQL CONTEXT FREE :LNK-CTX END-EXEC.
     GOBACK.
```

引数渡しされたランタム・コンテキストを利用するよう指示します。

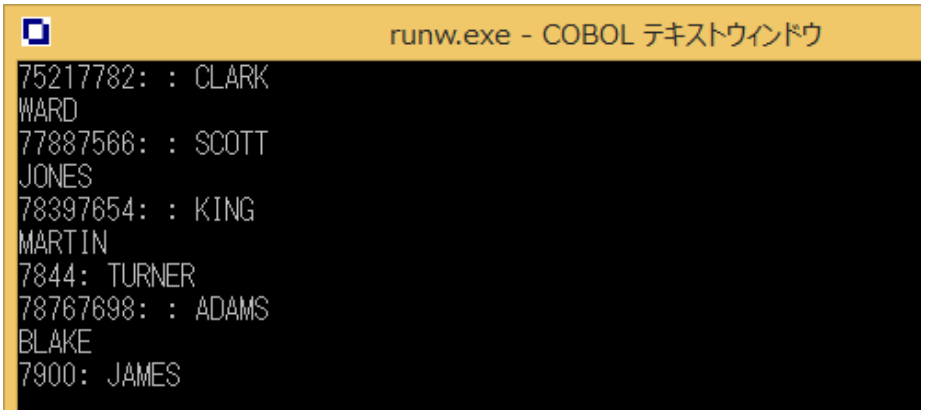
ランタム・コンテキストをメモリから解放します。

サブプログラム２(サブプログラム１との差分のみ記します)

```
                                        :
 PROGRAM-ID. PROCOBSUB2.
                                        :
     EXEC SQL
       DECLARE EmpCSR CURSOR FOR
           SELECT EMPNO, ENAME FROM EMP
           WHERE EMPNO BETWEEN 7701 AND 7900
     END-EXEC.
                                        :
```

参照するレコード範囲がサブプログラム１と異なります。

下図はこのアプリケーションの実行イメージとなります。同じデータベース上の同じテーブルを参照するプログラムにも関わらず並列で処理されていることが出力結果からわかります。



おわりに

以上、マルチスレッド環境で COBOL プログラムを正しく動作させるための複数の方法を見てきました。
これからも COBOL プログラマが最初からスレッドを意識したプログラミングを使用して開発する機会はそれほど多くはないと考えられます。一方で、既存の COBOL プログラムを再利用しようとした際にたまたまその環境がマルチスレッド環境であるというケースは少なくないはずです。手段は一つではありません。プログラムを正しく動作させるだけでなく、移行コストの削減と実行性能の向上を両立させるための試行錯誤によって目的に即した正しい手段を採用して頂く必要があります。