マルチスレッドの概要 

Micro Focus Net Express®

マルチスレッドプログラミング

第 5 版
2003 年 4 月

このマニュアルでは、Net Express を使用したマルチスレッドプログラミングについて説明します。 具体的には、スレッドの同期について詳しく説明し、マルチスレッドプログラムを作成するために必要な情報とコード例を示します。

マルチスレッドの概要 

実行の同期と競合の解決

第 1 章 : マルチスレッドの概要

「マルチスレッド」という用語には次のような意味があります。

- オペレーティングシステムが提供する機能で、アプリケーションは、この機能に基づいて、プロセス内で実行するスレッドを作成できます。
- オペレーティングシステムが提供するマルチスレッド機能を利用したアーキテクチャのアプリケーションを指します。

マルチスレッドとオペレーティングシステム

最近のオペレーティングシステムでは、多数の異なるユーザがプログラム (プロセス) を同時に実行できます。 1 つの中央処理装置 (CPU) が搭載されたハードウェアシステムでオペレーティングシステムが動作する場合は、厳密にはこれらのプロセスが同時に実行されません。プロセスが同時に実行されているように見えるのみです。 オペレーティングシステムは CPU とその他のシステムリソースを必要なプロセスに割り当て、これらのリソースを実行中のプログラムから別のプログラムに (プリエンプティブ方式で) 切り替えます。 このリソースの切替えにより、複数のプログラムが同時に実行されているように見えます。

最近の多数のオペレーティングシステムでは、これと非常に良く似た方法により、各プロセス内で複数のスレッドを使用できます。 ここでもまた、オペレーティングシステムは CPU とさまざまなリソースを個別のスレッドに割り当て、これらのリソースの割り当てを実行中のスレッドから別のスレッドに切り替えます。 そのため、複数のスレッドが同時に実行されているように見えます。

では、オペレーティングシステムがプロセスとスレッドを同じように処理するのであれば、この 2 つをどのように区別するのでしょうか ?

すべてのプロセスは、アドレス領域、オープンファイル、他のシステムリソースを個別にもっています。そのため、同時に実行されている他のプロセスに対して与える影響を最小限、または影響がない状態で、各プロセスを実行できます。 ただし、リソースを個別にもつことで、次のランタイムコストがかかります。

- 各プロセスが各アドレス領域についてシステムメモリを消費します。
- オペレーティングシステムが、実行コンテキストをあるプロセスから別のプロセスに切り替えるたびに、多くの内部情報を更新する必要があります。

スレッドは 1 つのプロセス内で実行されるので、プロセスに関連する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、同じオープンファイル、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、スレッドはよく「簡易プロセス」と呼ばれます。

マルチスレッドとアプリケーション

マルチスレッドアプリケーションとは、オペレーティングシステムが提供するマルチスレッド機能を利用したアーキテクチャのアプリケーションを指します。 通常、マルチスレッドアプリケーションは、プロセス内の各スレッドに特定のジョブを割り当てます。各スレッドはさまざまな方法で互いに通信し合い、それぞれのアクションの同期を取ります。 たとえば、データ処理アプリケーションを設計する場合に、1 つのスレッドでグラフィックユーザインターフェイスを完全に処理し、別のスレッドでそのアプリケーションの実際の作業を行います。 このアーキテクチャにより、アプリケーション内の実際の作業とユーザインターフェイスを完全に分離できます。

他には、マルチスレッドアプリケーションをクライアント / サーバアプリケーションのサーバ側で使用する方法もあります。 サーバを設計する場合に、主コントローラスレッドまたは通信スレッドに送信されたそれぞれのサービス要求に対して、その要求を処理し、結果をコントローラに返す別のサービススレッドを作成できます。 このアーキテクチャにより、コントローラはこれらの結果をクライアントに送り返します。 このアーキテクチャの主な利点は、サーバが 1 つのクライアントの要求に拘束されて他のクライアントの要求に迅速に応答できないという事態が発生しないことです。

サーバから応答をただちに (または高速で) 得るクライアント / サーバアプリケーションは、従来、マルチスレッドを使用しないで作成されてきました。 これは、通常、主コントローラプロセスまたは通信プロセスが各クライアントの要求に対して個別のサービスプロセスを作成するために可能でした。 マルチスレッドがこれらのアプリケーションで必要とされる 主な理由は、システムリソースの有効利用です。

スレッドの作成時に必要なオペレーティングシステムのリソース (メモリ、コンテキストスワップのオーバヘッドなど) はわずかなので、ファイルや他のリソースを簡単に共有できます。 一方、プロセスの作成には、より多くのシステムリソースが必要なので、プロセス間で互いに通信したり、ファイルを共有したりすることがより難しくなります。 マルチスレッドを使用すると、既存のハードウェアリソースを最大限に利用し、リソースの共有を簡略化できます。

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データ項目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。

たとえば、次に示すコードを考えます。

```
Working-Storage Section.        Working-Storage Section.
01 group-item.                  01 group-item.
  05 field-1  pic x.              05 field-1  pic x.
  05 field-2  pic x.              05 field-2  pic x.
procedure division.             procedure division.
move 'A' to field-1             move 'C' to field-2
move 'B' to field-2             move 'D' to field-1
display group-item              display group-item
```

次に示すスレッドの実行順序を考えます。

処理のステップ	スレッド **1** の実行	スレッド **2** の実行
1	`move 'A' to field-1`	
2		`move 'C' to field-2`
3	`move 'B' to field-2`	
4		`move 'D' to field-1`
5	`display group-item`	
6		`display group-item`

この例では、どちらのスレッドも予期した値を表示しません。 予期していたスレッド 1 の結果は 'AB' で、スレッド 2 の結果は 'DC' です。 ただし、両方のスレッドが実際に表示する値は 'DB' です。 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てください。詳細については、『実行の同期と競合の解決』の章で説明します。

マルチスレッドの概要　　マルチスレッドアプリケーションの作成

第 2 章 : 実行の同期と競合の解決

マルチスレッドの実行がアプリケーション内で制御されない場合は、予期しない結果が生じます。 アプリケーションを実行して目的の結果を得るには、スレッドの実行の同期を取り、スレッド間のデータ競合を解決する必要があります。この問題を解決するには、さまざまな方法があります。 使用する方法は、アプリケーションのデータアクセス特性や、アプリケーションを構成する個々の COBOL プログラム内で完全なマルチスレッドを実現するための要件によって異なります。 使用できる方法は次のとおりです。

- コンパイラ指令で割り当てられたプログラム属性
- データ項目へのアクセス時に、複数のスレッドが競合する可能性があるかどうかを決定するためのデータ項目属性
- 同期プリミティブ

マルチスレッドプログラム属性

COBOL プログラム内のマルチスレッドに影響を与える主なプログラム属性は 3 つあります。 これらの属性は、プログラムのコンパイル時にマルチスレッドのコンパイラ指令を取り込んだり、取り外したりすることで割り当てられます。 使用する指令はプログラムのシステム作業領域の割り当てに影響を与え、コンパイルされたプログラムはシステムによって自動的にロックされることがあります。 プログラムは、次のように指定できます。

- マルチスレッドコンパイラ指令を指定せず、プログラムをマルチスレッドにしません。
- シリアル属性をもたせます。 シリアルプログラムでは、作業領域が静的に割り当てられます。プログラムは、開始時にロックされ、終了時にロック解放されます。
- 再入可能属性をもたせます。 再入可能なプログラムでは、スタック上のシステム作業領域はすべて動的に割り当てられ (これらの領域でのスレッドの競合を回避)、複数のスレッドが同時にそのプログラムを実行できます。

非マルチスレッド指定

プログラムのコンパイル時にマルチスレッドコンパイラ指令を指定しないと、非マルチスレッドプログラムになります。 この場合は、システム作業領域は静的に割り当てられ、競合の対象となります。 この方法には、呼び出しの高速化、マルチスレッド間でのスタックの有効利用など、いくつかの利点がありますが、非マルチスレッドプログラムで、確実に一度に 1 つのスレッドのみが実行されるかどうかはアプリケーションによって異なります。ただし、呼び出し側プログラムの暗黙的なロジックにより、非マルチスレッドプログラムで、一度に 1 つのスレッドのみ実行するようにできます。たとえば、1 つのスレッドが非マルチスレッドプログラムを呼び出すようにアプリケーションを設計します。 または、呼び

出されたプログラムが実行される直前に呼び出し側プログラム内で同期プリミティブ (ミューテックスなど) の 1 つをロックし、呼び出されたプログラムが戻る前にそのロックを解放します。

シリアルプログラム

シリアル属性をもつプログラムでは、システム作業領域は静的に割り当てられます。プログラムは開始時にロックされ、終了時にロック解放されます。 このロック機能により、一度に 1 つのスレッドのみがプログラムを実行することになるので、システムまたはユーザ作業領域での競合を回避できます。 他の明示的なアプリケーションのロジックは必要ありません。

コンパイル時に SERIAL コンパイラ指令を指定すると、プログラムにシリアル属性を与えます。

通常の COBOL プログラムをシリアルプログラムとして指定すると、プログラムのソースコードを変更しないでマルチスレッドアプリケーションにインクルードできます。

一方、シリアルプログラムの欠点は次のとおりです。

- マルチスレッドのレベルはアプリケーション内で制限されます。 マルチスレッドアプリケーションでは、理想的には最大性能を発揮するために可能な限り多くのコードを無制限に実行できることが必要となります。
- プログラムには、ロックおよびロック解放処理に多くのランタイムコストがかかります。 このランタイムコストは呼び出しの処理速度とアプリケーション全体の性能に大きく影響します。 そのため、このコンパイラ指令でコンパイルするモジュール数を可能な限り少なくし、呼び出されるプログラムには必ず非マルチスレッド属性を設定する必要があります。 たとえば、プログラム A が、シリアル属性でコンパイルされ、かつアプリケーション内でプログラム B と C を呼び出す唯一のプログラムである場合には、アプリケーションの設計により、プログラム B と C はすでにシリアル化されているので、非マルチスレッド属性でコンパイルできます。

再入可能なプログラム

REENRANT コンパイラ指令を指定してコンパイルすると、マルチスレッドプログラムを再入可能なプログラムに設定できます。 マルチスレッドアプリケーションでは、ほとんどの (すべてではない) モジュールに対して、再入可能なプログラムを使用してください。

REENTRANT(1) を指定した場合には、コンパイラが生成したすべての一時作業領域はそれぞれのスレッドに割り当てられます。 環境部とデータ部に割り当てられたすべてのユーザデータ領域と FD ファイル領域は、すべてのスレッドで共有されます。 プログラマは、CBL_ 同期呼び出しを使用して、プログラムのデータを確実にシリアル化する必要があります。

REENTRANT(2) を指定した場合は、システム作業領域の他に、作業場所節 (Working-Storage Section) とファイル節 (File Section) のすべてのデータはそれぞれのスレッドに割り当てられます。 Local-Storage は、スタック上に動的に割り当てられます。 この割り当てに

より、これらの領域での競合が回避されるため、複数のスレッドから同時に呼び出された場合でもプログラムは問題なく動作します。 プログラムのロックまたはロック解放は必要ありません。 このコンパイラ指令設定の欠点は、スレッド間でデータを共有しないことです (EXTERNAL で定義されたデータを除く)。

REENTRANT(2) は、プログラムをマルチスレッドアプリケーションで素早く、簡単に動作させる方法です。ただし、通常は REENTRANT(1) を使用してコンパイルしてください。

再入可能なプログラムでは、その Working-Storage Section と File Section 内のデータ項目で起こる可能性があるすべての競合を解決する必要があります。 1 つ以上の技法を使用して、データ競合を解決する必要があります。詳細については、次を参照してください。

データ属性の使用

データ項目の属性は、そのデータ項目にアクセスするスレッドで競合が発生するかどうかを決定します。 スレッドは次のデータ項目に対しては競合しません。

- Local-Storage Section または Thread-Local-Storage Section で定義されたデータ項目
- THREAD-LOCAL データ属性で定義されたデータ項目

プログラムで定義された他のデータ項目はスレッド間で共有されるため、競合が発生する可能性があります。 次の場合に競合が発生することがあります。

- Working-Storage Section と File Section で定義されたデータ
- ファイル状態フィールドなどの暗黙的に定義された他のデータ項目
- コンパイラ専用レジスタに格納されたデータ (RETURN-CODE レジスタを除く)

通常、Local-Storage Section で定義されたデータは、再帰的な COBOL プログラムで使用されます。プログラムが再帰的に呼び出されるたびに、このデータのインスタンスが新規にスタックに割り当てられます。 マルチスレッドアプリケーションの各スレッドは独自のスタックを持っているので、この属性は、再入可能なプログラムで競合しない一時的な作業項目を定義するのに理想的です。 設計によっては、再入可能な COBOL プログラムを 1 つのスレッド内で再帰的に使用できます。

Local-Storage Section で定義されたデータの欠点は、再入可能または再帰的なプログラムが終了するとデータ項目を見失うことです。 プログラムの終了後に再度そのプログラムを開始すると、Local-Storage Section で定義されたデータ項目の値は未定義になります。 スレッドの要請により複数の呼び出しについてプログラムで状態を保存する必要がある場合は、別の機構を使用する必要があります。

これらの問題は、Thread-Local-Storage Section または THREAD-LOCAL 属性により、スレッドローカルとして定義されたデータを使用すると解決します。 スレッドローカルデータは各スレッドに対して一意で、呼び出し全体に対して固定されているため、value 句で初期化できます。 スレッドローカルデータはスレッド固有の Working-Storage Section のデータ項目として表示できます。 この固定データは、ほとんどの再入可能なプログラムでの競

合問題を解決するのに非常に役立ちます。 多くの場合は、ファイル処理を行わないプログラムで Working-Storage Section 見出しを Thread-Local-Storage Section 見出しに変更するのみで、完全に再入可能なプログラムにできます。 読み込み専用の定数はすべて Working-Storage Section で、読み書きデータはすべて Thread-Local-Storage Section で定義すると、データの割り当てを微調整できます。

スレッドローカルデータの欠点は次のとおりです。

- データの値をスレッド間で容易に通信し合うことができません。このため、Working-Storage Section でデータを定義し、適切なスレッド同期処理を行います。
- プログラムの呼び出しでオーバヘッドが増えます。通信量の多いモジュールで使用すると、アプリケーション全体の実行処理性能に大きな影響を与えます。
- EXTERNAL 項目を Thread-Local-Storatge Section で指定できません。 現在、スレッドローカルデータをアプリケーションモジュール間で共有する必要がある場合は、CBL_TSTORE_n ルーチンを使用する必要があります。

スレッドではプライベートなデータ以外のデータが必要となることがあります。 大部分のマルチスレッドアプリケーションは、厳密なプロトコルのもとで各スレッドがアクセスする共有データを使用して通信し合い、スレッドの実行を調整します。 COBOL では、競合によるデータの破壊を防ぐさまざまな同期プリミティブと Working Storage Section または File Section のデータを併用することでこれを実現します。 同期プリミティブの詳細については、次を参照してください。

同期プリミティブの使用

マルチスレッドアプリケーションで目的の結果を得るには、共有データにアクセスするスレッド間で同期を取ることが重要です。 スレッド間で行われるさまざまなデータアクセスの性質を理解することは、使用する同期プリミティブと方式を決定するための第一歩です。 データアクセスの特性を理解すると、すべてのスレッド間で使用するデータ項目の同期方式も簡単に理解できます。 次では、データを共有する上での一般的な問題とそれらの解決策について概説します。

ミューテックスの使用

データ共有の最も単純な問題は、処理中のある時点で複数のスレッドが同時に互いに排他的に共有データへアクセスする場合です。 この共有データにアクセスするコードの部分はクリティカルセクションと呼ばれます。これらのクリティカルセクションは、共有データ項目に論理的に関連付けられたミューテックスを使用することで保護できます。 「ミューテックス (mutex)」は、mutual exclusion (相互排他) から派生した用語です。 データ項目に関連付けられたミューテックスは、クリティカルセクションが実行される前にロックされ、終了された後にロック解放されます。

保護するデータにアクセスする前に、すべてのスレッドがミューテックスをロックすることが非常に重要です。 スレッドが 1 つでもこの同期方式に失敗し、ミューテックスをロックできなかった場合は、予期しない結果が発生します。

たとえば、次のコードは表に項目を追加するため、または表内の項目数を調べるためにその表にアクセスする 2 つのクリティカルセクションを保護する方法を示しています。 作業場所節のデータ項目 table-*xxx* は、table-mutex によって保護されます。

例 - クリティカルセクションの保護

この例では、あるスレッドが表を読み込んでいる間に別のスレッドがその表にデータを追加するのを防ぐために、ミューテックスが必要となります。

```
 Working-Storage Section.
 78  table-length        value 20.
 01  table-mutex         usage mutex-pointer.
 01  table-current       pic x(4) comp-x value 0.
 01  table-max           pic x(4) comp-x value 0.
 01  table-1.
    05  table-item-1     pic x(10) occurs 20.
 Local-Storage Section.
 01  table-count         pic x(4) comp-x.
 01  table-i             pic x(4) comp-x.

* シングルスレッドモードで実行される初期化コードです。
     move 0 to table-max
     open table-mutex

* table-1 に項目を追加します。これは、クリティカルセクションです。
     set table-mutex to on
     if table-current < table-length
         add 1 to table-current
         move 'filled in' to table-item-1(table-current)
     end-if
     set table-mutex to off

* table-1 の項目数を数えます。これは、クリティカルセクションです。
     set table-mutex to on
     move 0 to table-count
     perform varying table-i from 1 by 1
             until table-i > table-current
         if  table-item-1(table-i) = 'filled in'
             add 1 to table-count
         end-if
     end-perform
     set table-mutex to off
```

ミューテックスを使用する上での 1 つの問題は、アプリケーション内でのマルチスレッドのレベルが厳密に制限されることです。 たとえば、表に項目を追加するプログラム、または表内の項目数を数えるプログラムなどいくつかのプログラムからなるアプリケーションがあるとします。 このアプリケーションでマルチスレッド処理を最大にするために、複数の

スレッドを使用して表内の項目数を同時に数えたいとします。 ただし、あるプログラムが表に項目を追加している場合には、別のプログラムで項目の追加や項目数の計算を行いたくありません。

この問題を解決するには同期プリミティブを使用します。同期プリミティブは、読み取り専用のクリティカルセクションでは複数のスレッドをアクティブにし、書き込み可能なクリティカルセクションではあるスレッドがアクティブのときは他のスレッドのアクセスを許可しません。このような同期プリミティブの例としては、モニタがあります。

モニタの使用

モニタは、ミューテックスでは容易に処理できない特別なアクセス問題を解決します。 たとえば、データの読み込みは多数のスレッドで同時に行い、データの書き込みは 1 つのスレッドのみで行うことができます。 スレッドがデータを書き込んでいる間は、他のスレッドからの読み込みアクセスを禁止できます。

モニタは、クリティカルセクションで使用し、クリティカルセクションが保護データで実行するデータアクセスの種類を宣言します。データアクセスの種類とは、読み込み、書き込み、または参照を指します。

モニタの同期機能を拡張し、クリティカルセクションで参照を行うこともできます。この参照機能は、実際のアプリケーションで大変役立ちます。 参照用のクリティカルセクションは、データの読み込みや保護データの書き込みを行います。条件によっては、保護データ項目の書き込みを行わない場合もあります。 このクリティカルセクションがアクティブの場合には、単にデータを読み込むのみのクリティカルセションは複数実行できますが、参照または書き込みを行う他のクリティカルセクションは実行できません。 参照しているスレッドが保護データへの書き込みを必要と判断した場合は、参照ロックから書き込みロックへの変換を要求します。 変換プロセスは、データを読み込むクリティカルセクションがすべて終了した後で実行され、データの読み込みや書き込みを行う他のクリティカルセクションがデータにアクセスするのを防ぎます。 参照用のクリティカルセクションは、保護データ項目に排他的にアクセスし、書き込み処理を行います (この状態は、参照用のクリティカルセクションが保護データ項目を読み込んだときの状態を保証します)。

例 - モニタを使用した複数スレッドのアクセス制御

次のコードは、表内の項目数を数えたり、表に項目を追加したりする複数のスレッドのアクセスを制御するモニタを示します。 このコードでは、次の処理を行います。

- 項目数を数えるために複数のスレッドがデータへアクセスすることを可能にします。
- 項目数を数えるスレッドが動作しているときには、項目を表に追加するスレッドを実行しません。
- 表に項目を追加するスレッドを 1 つのみ実行します。このスレッドの実行中は、項目数を数える他のスレッドによる表へのアクセスを禁止します。

```
Working-Storage Section.
78  table-length                value 20.
01  table-monitor             usage monitor-pointer.
```

```
 01  table-current              pic x(4) comp-x value 0.
 01  table-1.
     05  table-item-1           pic x(10) occurs table-length.
 Local-Storage Section.
 01  table-count                pic x(4) comp-x.
 01  table-i                    pic x(4) comp-x.
  . . .
*> シングルスレッドモードで実行される
*> 初期化コードです。
     move 0 to table-current
     open table-monitor

* table-1 に項目を追加します。これは、書き込み用のクリティカルセクションです。
     set table-monitor to writing
     if table-current < table-length
         add 1 to table-current
         move 'filled in' to table-item-1(table-current)
     end-if
     set table-monitor to not writing

* table-1 の項目数を数えます。これは、読み込み用のクリティカルセクションです。
     set table-monitor to reading
     move 0 to table-count
     perform varying table-i from 1 by 1
             until table-i > table-current
         if  table-item-1(table-i) = 'filled in'
             add 1 to table-count
         end-if
     end-perform
     set table-monitor to not reading
```

例 - モニタを使用する参照用のクリティカルセクション

簡単な参照用のクリティカルセクションの例を示します。

```
 Working-Storage Section.
 01  data-monitor       usage monitor-pointer.
 01  data-value         pic x(4) comp-x value 0.

* シングルスレッドモードで実行される
* 初期化コードです。
     open data-monitor

* table-1 に項目を追加します。これは、参照用のクリティカルセクションです。
     set data-monitor to browsing
if data-value < 20
```

```
        set data-monitor to writing
            converting from browsing
        add 5 to data-value
        set data-monitor to not writing
    else
        set data-monitor to not browsing
    end-if
```

このような簡単な確認のために参照ロックを使用することはお奨めしません。 通常、参照ロックを使用する必要があるのは、実際に書き込みロックが必要かどうかを決定するために多大な作業が必要で、アプリケーション全体でマルチスレッドを最大限に利用する場合のみです。 他にも、アプリケーションでマルチスレッドのレベルを最大にするためのさまざまなモニタ変換があります。

セマフォの使用

セマフォは、同期プリミティブの一種で、ゲートのように、その値を減少することでコード内でのスレッドの実行を禁止したり、その値を増加させることでそのコード内での (他のスレッドの) 実行を許可したりします。 セマフォは、ミューテックスと似ているので、ミューテックスのかわりに使用できます。

例 - セマフォの使用

次のコードでは、セマフォの使用例を示します。

```
 Working-Storage Section.
 01  data-semaphore     usage semaphore-pointer.
 01  data-value         pic x(4) comp-x value 0.

* シングルスレッドモードで実行される初期化コードです。
    open data-semaphore
    set data-semaphore up by 1      *> 増加として初期化します。

* data-value の変更を追加します。これは、クリティカルセクションです。
    set data-semaphore down by 1
    add 1 to data-value
* セマフォにより他のスレッドの実行が可能になります。
    set data-semaphore up by 1
```

OPEN 文の直後で、セマフォが 1 増加されていることに注目してください。これにより、最初のセマフォの獲得は成功しますが、それ以降の獲得はそのセマフォが再度解放されるまではすべてブロックされます。

セマフォは、ミューテックスほど効果的ではありませんが、より柔軟性があります。1 つのスレッドがセマフォを単に解放するのみで、他のスレッドはその解放されたセマフォを獲得できます。 ミューテックスと対比すると、 ミューテックスの場合は、常に解放される前に獲得する必要があります。ミューテックスの獲得操作と解放操作は同じスレッド内で行う必要があります。 セマフォはあるスレッドから別のスレッドへのシグナルとなります。

例 - 2 つのスレッド間でハンドシェイクを確立するためのセマフォの使用

次のコードでは、2 つのスレッド間のハンドシェイクを確立するために異なる 2 つのセマフォを使用します。 このハンドシェイクにより、一方のスレッドは新規データ値の生成を通知し、他方のスレッドはそのデータ値の使用を通知できます。

```
 Working-Storage Section.
 01  produced-semaphore          usage semaphore-pointer.
 01  data-value                  pic x(4) comp-x value 0.
 01  consumed-semaphore          usage semaphore-pointer.

* シングルスレッドモードで実行される初期化コードです。
     open produced-semaphore
     open consumed-semaphore
     set consumed-semaphore up by 1

* このコードは、data-value を生成するために 1 度実行されます。
     set consumed-semaphore down by 1
     add 10 to data-value
* data-value の変更を通知するシグナルです。
     set produced-semaphore up by 1

* data-value を変更するために待機状態にある別のスレッドも、
* 1 度のみ、このコードを実行します。
     set produced-semaphore down by 1
     display data-value
* data-value の使用を通知するシグナルです。
     set consumed-semaphore up by 1
```

この例では、作成者 / 使用者間の問題として知られる、もう一つの一般的な同期問題を示しています。 最も単純な作成者 / 使用者間の問題は、データを生成する 1 つのスレッドとそのデータを使用する もう 1 つのスレッドがある場合に、使用スレッドの実行中は処理対象のデータが常に存在するように、これらの作成スレッドと使用スレッド間で実行の同期を取る必要があるということです。

上記のセマフォは解放されていないセマフォ数を数え、多数のスレッドがセマフォを獲得しブロック解除を行うようにします。 セマフォをこのように数えると、使用スレッドがデータ値を獲得するまでの間、ブロックして待機する前に、作成スレッドは複数のデータの値 (通常は配列で) を生成できます。

例 - セマフォの数え方

次のコードは、作成側と使用側の組の簡単な例を示します。作成側は、データ表がいっぱいになるまでデータの値を作成できます。データ表がいっぱいになると、作成側は値が使用されるまで新たに値を作成しません。

```
 Working-Storage Section.
 78  table-size                        value 20.
 01  produced-semaphore          usage semaphore-pointer.
```

```
 01  filler.
     05  table-data-value       pic x(4) comp-x
                                          occurs table-size
times value 0.
 01  consumed-semaphore          usage semaphore-pointer.
 Local-Storage Section.
 01  table-i                                pic x(4) comp-x.

* シングルスレッドモードで実行される初期化コード
     open produced-semaphore
     open consumed-semaphore
*   値を 20 回増加します。
     set consumed-semaphore up by table-size

* 作成スレッド
     move 1 to table-i
     perform until 1 = 0
         set consumed-semaphore down by 1
         add table-i to table-data-value(table-i)
         set produced-semaphore up by 1
         add 1 to table-i
         if  table-i > table-size
             move 1 to table-i
         end-if
     end-perform.

* 使用スレッド
     move 1 to table-i
     perform until 1 = 0
         set produced-semaphore down by 1
         display '現在の作成値：'
                 table-data-value(table-i)
         set consumed-semaphore up by 1
         add 1 to table-i
         if  table-i > table-size
             move 1 to table-i
         end-if
     end-perform.
```

イベントの使用

あるスレッドが別のスレッドに、注意を喚起する必要が生じたことをシグナルとして通知できる点で、イベントはセマフォと似ています。 イベントは、セマフォと比べて柔軟ですが、少し複雑になります。 その理由の 1 つは、イベントを一度ポストすると、そのイベントを明示的にクリアする必要があるということです。

例 - イベント同期の使用

次のコード例では、セマフォ同期のかわりにイベント同期を使用して作成者 / 使用者間の問題を解決します。

```
 Working-Storage Section.
 01  produced-event     usage event-pointer.
 01  data-value         pic x(4) comp-x value 0.
 01  consumed-event     usage evnt-pointer.

* シングルスレッドモードで実行される初期化コード
     open produced-event
     open consumed-event
     set consumed-event to on    *> 「ポスト済み」として初期化します。

* 作成側のプロトコルです。
     wait for consumed-event
     set consumed-event to false *> イベントをクリアします。
     add 10 to data-value

* data-value の変更を通知するシグナルです。
     set produced-event to true  *> イベントをポストします。

* 使用側のプロトコルです。data-value の変更を
* 待ちます。
     wait for produced-event
     set produced-event to false *> イベントをクリアします。
     display data-value

* 実行可能であることを他のスレッドへ通知するシグナルです。このスレッドは
* data-value をもっています。
     set consumed-event to true  *> イベントをポストします。
```

上記のコードを実行するスレッドが 2 つ (作成スレッドと使用スレッド) のみの場合は、問題なく動作します。 これ以外のスレッドが作成スレッドまたは使用スレッドとして起動された場合は、予期しないイベントが発生します。 これは、イベントがセマフォとは異なり、一度ポストされるとそのイベントの発生を待っているスレッドをすべてアクティブにするためです。セマフォの場合は、セマフォが解放された後は 1 つのスレッドのみの実行を可能にします。 イベントがポストされ、待ち状態のスレッドがアクティブ化されると、これらの各スレッドはそのイベント (イベントのクリアを含む) に対して何らかのアクションを行うべきかどうかを決定する必要があります。

最後のポイントとして、待ち状態の複数のスレッドが存在し、特定の用途のために独自の同期オブジェクトの構築を可能にする場合には、イベントの使用は難しくなります。

マルチスレッドの概要　　マルチスレッドアプリケーションの作成

実行の同期と競合の解決　　マルチスレッドコンパイラ指令

第 3 章 : マルチスレッドアプリケーションの作成

マルチスレッドアプリケーションを作成するときには、アプリケーション内の各プログラムをどのように動作させるか、つまり、標準 COBOL プログラム、シリアルプログラム、または再入可能なプログラムのうち、どの種類のプログラムを使用するかを考慮する必要があります。 これら 3 種類のプログラムの中で、REENTRANT(1) プログラムとしてコンパイルされたプログラムには特に注意が必要です。

また、次の事項も考慮する必要があります。

- アプリケーションの初期化
- スレッドの操作
- スレッドローカルデータの作成

マルチスレッドプログラミングに関する一般的な問題についても知る必要があります。

マルチスレッドアプリケーション用ランタイムシステム

マルチスレッドアプリケーション用ランタイムシステムでは、マルチスレッドアプリケーションとシングルスレッドアプリケーションの両方を処理する必要があります。 その結果、マルチスレッド用ランタイムシステムでは、標準のマルチスレッド COBOL アプリケーションで必要とする多くの同期処理コストがかかります。 さらに、ほとんどの場合に、これらのコストはアプリケーションがマルチスレッドまたはシングルスレッドかどうかに関わらず必要です。 このために、さらにシングルスレッドアプリケーションを最高速度で実行するために、Micro Focus 社では 2 つのランタイムシステムを用意しました。1 つはすべてのアプリケーションをサポートするマルチスレッド用ランタイムシステム、もう 1 つはシングルスレッドアプリケーションのみをサポートするシングルスレッド用ランタイムシステムです。 アプリケーションの実行時、またはアプリケーションのリンク時にどちらのランタイムシステムを使用するかを選択できます。

アプリケーションと一緒に配布するファイルについては、アプリケーションがシングルスレッド用ランタイムシステムまたはマルチスレッド用ランタイムシステムのどちらにリンクされているかによって異なります。 詳細については、ヘルプトピック『使用するランタイムファイルの決定』を参照してください。

再入可能なプログラム作成時の考慮事項

すべてのプログラムを再入可能なプログラムとしてコンパイルできるとは限りません。 使用する COBOL 機能によっては、再入可能なプログラムのコンパイルを妨げることがあります。ANSI 標準の COBOL に含まれるこのような COBOL 機能は古いので、使用しないでください。 次の機能を使用するプログラムは、REENTRANT 指令を指定してコンパイルすることはできません。

- ALTER 文
- ON 文
- COBOL DEBUG 機能
- 手続き部オーバーレイ - NOSEG または NOOVL 指令のどちらかを指定すると、この機能を使用していてもコンパイルできます。
- PROGRAM-ID 文の IS INITIAL 句
- STICKY-LINKAGE コンパイラ指令
- STICKY-PERFORM コンパイラ指令

これ以外に、再入可能なプログラムを作成する場合に注意する必要のある制約は、次のとおりです。

- ほとんどのファイル処理文は、前述の同期プリミティブの 1 つで明示的に保護する必要があります。 ファイルハンドラは内部的にスレッドを保護します。ただし、ユーザプログラムでは、ファイルレコード、バッファ、および状態フィールドはロックしないで設定されます。 これにより、競合が発生し、これらのフィールドが破壊されます。 このため、ファイル処理文を保護するための手段を行ってください。
- ファイル操作の実行は、グローバルなファイル処理ミューテックスによってファイルハンドラ内でシリアル化されます。 これは、ある時点で 1 つのスレッドのみがファイルにアクセスできることを意味します。 ただし、ファイルからの読み込みが何らかの理由で遅れた場合は、問題が発生します。たとえば、レコードのロック処理です。 この場合には、他のスレッドは遅延している読み込みが完了するまで、すべてのファイル処理操作を実行できません。
- 整列操作 (SORT) の実行は、グローバルなファイル処理ミューテックスにより整列ハンドラ内でシリアル化されます。 整列操作の実行中に、他のスレッドが別の整列操作またはファイル処理操作を実行できません。 INPUT 手続きと OUTPUT 手続きの実行は、整列操作を開始したスレッド内で行われます。 これらの手続き内のファイル処理は、正常に実行されます。
- DISPLAY 文を実行するには、同期プリミティブの 1 つでユーザを保護する必要があります。 コンソール表示ハンドラは、内部的にスレッドを保護します。ただし、表示が複雑な場合は、複数の表示ハンドラ呼び出しに分割されます。 そのため、データ項目を交互に表示します。 一般に、マルチスレッドアプリケーションでは、コンソール I/O に割り当てるスレッドを 1 つのみにするのが最良の方法です。
- ネストされたプログラムは、新しく作成するスレッドの開始点になることはできません。 ENTRY ポイント、コンパイルユニットの一番外側のプログラム、およびその他の言語の外部エントリポイントのみが、スレッド作成の開始点となることができま

す。

- ACCEPT 文の実行は、受諾ハンドラ内でシリアル化されます。 このシリアル化により、元の ACCEPT 処理が終了するまで、他のすべてのスレッドは ACCEPT 操作または DISPLAY 操作が実行できなくなります。

次に挙げる性能とリソースに関する弊害は、マルチスレッドアプリケーションの再入可能な COBOL プログラムで発生します。

- Thread-Local-Storage Section を使用すると、プログラムエントリに多大なコストがかかる場合があります。 可能な場合には、かわりに Local-Storage Section をスレッドローカル作業変数に対して使用します。 プログラムエントリのオーバヘッドは、CALL 速度の最適化を使用すると、最小にすることができます。
- 再入可能なプログラムは、標準 COBOL プログラムやシリアル COBOL プログラムよりも多くのスタック領域を使用します。 これは、システム作業領域を静的にではなく動的に、スタックに割り当てる必要があるためです。

マルチスレッドライブラリルーチン

名前による呼び出しライブラリルーチンは、プログラミングインターフェイスであり、これを介してアプリケーションにマルチスレッドを実装し、制御できます。 ライブラリルーチンは次の目的で使用します。

- スレッドの制御
- スレッドの同期
- スレッド固有データの処理

スレッド制御ルーチン

スレッドを制御するライブラリルーチンを使用して、別のスレッドの終了を待ち、その戻り値を獲得するスレッドを実装できます。

これらのライブラリルーチンを使用して作成したスレッドでは、次の処理が必要になります。

- STOP RUN を使用して終了します。
- CBL_THREAD_EXIT を使用して終了します。
- CBL_THREAD_KILL を使用して強制終了します。
- 元のエントリポイントから戻ります。

スレッドはスレッド制御ルーチン以外で作成される場合があります。 ただし、このような

スレッドも、ランタイムシステムサービスを使用すると、ランタイムシステムで認識されます。 これらのスレッドにはスレッド制御ライブラリルーチンを通してアクセスできます。 スレッドが使用するライブラリルーチンは次のとおりです。

- CBL_THREAD_LIST_START、CBL_THREAD_LIST_NEXT、および CBL_THREAD_LIST_END ルーチン。 スレッドは、これらのルーチンのリストにも表示されます。
- CBL_THREAD_IDDATA_ALLOC と CBL_THREAD_IDDATA_GET ルーチン
- スレッド自身から情報を取得する CBL_THREAD_SELF ルーチン
- CBL_THREAD_SUSPEND と CBL_THREAD_RESUME ルーチン
- 使用中にスレッド自身をデタッチする CBL_THREAD_DETACH ルーチン。ただし、この呼び出しは事実上は何もしません。 デタッチ呼び出しに関係なく、他言語プログラムのスレッドが使用するリソースは、そのスレッドがランタイムシステムの使用を終了し、cobthreadtidy() を呼び出したときに解放されます。 cobthreadtidy() を呼び出さずに、他言語プログラムのスレッドを終了するのは規約違反です。

複数言語を使用する環境で、ランタイムシステムを使用しているためにランタイムシステムに認識されているスレッドが、そのランタイムシステムまたはスレッド制御ルーチン以外で作成されている場合は、次のどれかを実行する必要があります。

- 終了する前に `cobthreadtidy()` を呼び出します。 この呼び出しは、ランタイムシステムに対して、スレッドがサービスを必要としなくなったことを通知し、スレッド状態データをクリアできるようにします。
- `cobtidy()` を呼び出して、このアプリケーションの実行時には COBOL ランタイムシステムを再度使用する必要がないことを通知します。
- `cobexit()` を呼び出して実行単位を終了します。
- CBL_THREAD_KILL を呼び出す別のスレッドを使用して、スレッドを強制終了しないでください。 スレッドが最初に作成された機構と一致する強制終了機構を使用してください。
- CBL_THREAD_WAIT を呼び出す別のスレッドを使用して待機しないでください。 スレッドが最初に作成された機構と一致する待機機構を使用してください。
- CBL_THREAD_EXIT ルーチンを使用してスレッドを終了しないでください。 スレッドが最初に作成された機構と一致する終了機構を使用してください。

アプリケーションは、CBL_GET_OS_INFO または CBL_THREAD_SELF を呼び出して、プログラムが使用するランタイムシステムがスレッド制御ルーチンをサポートするかどうかを確認できます。

例 - プログラムがサポートするランタイムシステムの確認

```
      call "CBL_THREAD_SELF" using thread-id
       on exeception
*        cbl_thread ルーチンはサポートされていません。
      end-call
      if  return-code = 1008
*        シングルスレッド専用 rts で実行中です。
      end-if
```

スレッド同期ルーチン

Net Express には、モニタ、セマフォ、ミューテックス、およびイベントでの 4 種類の同期オブジェクトが用意されています。 これら 4 種類の同期オブジェクトの制御は COBOL 構文と名前による呼び出しライブラリルーチンによって行われます。

スレッド固有データを処理するルーチン

ランタイムライブラリルーチン CBL_ALLOC_THREAD_MEM とライブラリルーチン CBL_TSTORE_n を使用して、スレッドの作成中にしか存在しない動的データ、選択したスレッドローカルデータまたは外部スレッドローカルデータを作成できます。

アプリケーションの初期化

マルチスレッドアプリケーションの初期化にはいくつかの注意が必要です。 正しいプログラミング方法としては、最初に、アプリケーションがマルチスレッドをサポートするランタイムシステムで動作しているかどうかを決定する必要があります。 次の例を参照してください。

例 - ランタイムシステムの確認

```
 Working-Storage Section.
 01 thread-id     usage pointer.
 . . .
* シングルスレッドモードで実行される初期化コードです。
* このコードで、ランタイムシステムがマルチスレッド。または、
* CBL_THREAD_ ルーチンをサポートしているかを確認します。
      call 'CBL_THREAD_SELF' using thread-id
       on exception
*         cbl_thread ルーチンはサポートされていません。
      end-call
      if  return-code = 1008
*         シングルスレッド rts で実行中です。
      end-if
```

この例では、ランタイムライブラリルーチン CBL_THREAD_SELF を使用します。 COBOL 構文では、マルチスレッドの多くの機能を使用できますが、COBOL ランタイムライブラリ

ルーチンでもすべての機能を使用できます。 ランタイムライブラリルーチンでは、COBOL 構文では使用できない高度なマルチスレッドプログラミング機能を使用できます。

ランタイムシステムがマルチスレッドをサポートしていることをアプリケーションが認識した場合は、同期プリミティブのすべてを適切な OPEN 文によって初期化する必要があります。 最も簡単な方法は、アプリケーション内で他のスレッドが作成される前に、主プログラムの一部として初期化を行う方法です。 ただし、アプリケーション設計、モジュール構成、または複数言語などの理由により、この方法で初期化ができない場合には、Micro Focus 社が提供する各プログラムごとの初期化済みミューテックスを使用できます。 このミューテックスには、CBL_THREAD_PROG_LOCK と CBL_THREAD_PROG_UNLOCK ランタイムライブラリルーチンを使用してアクセスします。 これらのルーチンを使用すると、プログラムのローカル同期プリミティブのハンドルをアプリケーションの実行中に確実に 1 回のみ初期化できます。

例 - ライブラリルーチンを使用したスレッドのロック

次のコードでは、ライブラリルーチンを使用して、プログラムをマルチスレッドモードで実行中に初期化する例を示します。

```
 Working-Storage Section.
 01 first-flag          pic x comp-x value 1.
    88 first-time        value 1.
    88 not-first-time    value 0.


* マルチスレッドモードで実行される初期化コードです。
* プログラムのローカルデータは確実かつ適切に初期化されます。

     if first-time then
         call 'CBL_THREAD_PROG_LOCK'
         if first-time then
* プログラムのローカルデータと同期オブジェクトを初期化します。
                ...
             set not-first-time   to true
         end-if
         call 'CBL_THREAD_PROG_UNLOCK'
     end-if
```

レベル-88 のデータ項目 first-time を二重チェックしていることに注目してください。 これは、マルチスレッドアプリケーションの優れた最適化方法です。 最適化の目的は、意図するアクションがすでに実行されている場合にミューテックスをロックするオーバヘッドをなくすことです。 この例では、複数のスレッドが、正しく初期化される前のプログラムを実行すると、first-time が真であることを検出し、CBL_THREAD_PROG_LOCK を呼び出します。 ただし、first-time が真の間に 1 つのロックのみはロックを獲得します。 ロックを取得したスレッドが初期化を行います。

初期化を終了し、初期化を行ったスレッドがロックを取得してる間に、first-time フラグが偽に設定されます。 これ以後にロックを取得しようとするスレッドは、first-time

が偽なので初期化がすでに完了していると判断し、プログラムのロックをただちに解放します。 初期化後に起動されたスレッドはすべて、first-time フラグが偽のために、プログラムロックの取得を行いません (そのため、プログラムエントリのオーバヘッドが減ります)。

初期化が行われているかどうかを示すフラグは、可能な限り簡単なものにしてください (1 バイトのデータ項目など)。

スレッドの操作

マルチスレッド用ランタイムシステムは、START 文または CBL_THREAD_n ライブラリルーチンを使用して、ランタイムシステム自身が作成したスレッドをサポートします。また、オペレーティングシステムによって直接作成された他言語のスレッドもサポートします。 COBOL 構文および CBL_THREAD_n ルーチンは汎用性があるため、スレッドの操作に適しています。 さらに、CBL_THREAD_n ルーチンは、他言語で作成されたプログラムからも呼び出せます。そのため、マルチスレッドアプリケーションはランタイムシステムの高度な機能を十分に利用できます。

スレッドハンドル

一般に、ランタイムシステムはスレッドハンドルでスレッドを識別します。 スレッドハンドルは、次のモジュールとスレッドで使用します。

- START 文または CBL_THREAD_CREATE ルーチンの戻り値として、スレッドを作成するモジュール
- CBL_THREAD_SELF ルーチンからの戻り値ごとのスレッド

スレッドハンドルは、各種のマルチスレッド COBOL 文または CBL_THREAD_n ルーチンに対して、一意にスレッドを識別します。 十分に注意を払えば、作成スレッドのみでなく、実行単位に含まれるアクティブなスレッドでもスレッドハンドルを使用できます。 スレッドの存在期間とそのスレッドハンドルの存在期間は必ずしも同じではないことを覚えておいてください。 スレッドの作成方法やスレッドに対する処理によっては、スレッドが終了してもそのスレッドハンドルがまだ有効な場合があります。 システムがそのハンドルに関連付けられたリソースを回復するために、スレッドのハンドルを明示的にデタッチするかどうかはユーザが判断します。

スレッドをハンドルからデタッチする方法はいくつかあります。 スレッドが START 文または CBL_THREAD_CREATE ルーチンで作成された場合は、デフォルトでは、作成されたハンドルはデタッチされます。 また、他言語で作成したスレッドがランタイムシステムで認識されると、作成されるハンドルは常に自動的にデタッチされます。 さらに、CBL_THREAD_DETACH を呼び出すと、デタッチされていないハンドルをデタッチできます。 どの場合も、デタッチされたハンドルは、そのスレッドが終了するとすぐに無効になるので常に注意して使用する必要があります。 ハンドルがデタッチされていないスレッドがすでに終了している場合には、そのスレッドハンドルで CBL_THREAD_DETACH を呼び出すと、ただちにそのハンドルは無効になります。

デタッチされていないハンドルは、戻り値を獲得する場合 (WAIT 文または CBL_THREAD_WAIT ルーチンを使用)、およびスレッドの終了後に (CBL_THREAD_IDDATA_GET ルーチンを使用) スレッド識別データを検査する場合に便利です。 START 文を指定すると、デタッチされていないスレッドハンドルを戻し、新規作成されたスレッドを識別できます。 CBL_THREAD_CREATE ルーチンを指定すると、スレッドが作成されてデタッチされていないハンドルが戻されたかどうかを示すフラグを使用できます。

スレッドの作成と終了

スレッドは次のどれかを使用して作成します。

- START 文
- CBL_THREAD_CREATE ルーチン
- スレッドを作成するための特定のオペレーティングシステムの呼び出し

最初の 2 つの方法で作成されたスレッドは COBOL スレッドと呼ばれます。 オペレーティングシステムによって直接作成されたスレッドは他言語スレッドと呼ばれます。

COBOL スレッドの作成と操作が汎用的で優れているので、ここでは、COBOL スレッドについてのみ説明します。

スレッドの開始点は、ネストされてないプログラム名、COBOL エントリポイント、または C 言語などの他言語で記述された外部ルーチンであることが必要です。ネストされたプログラム名、節名、または段落名を開始点とすることはできません。

開始点の名前は、テキスト文字列で指定できます。 テキスト文字列を使用してエントリポイントを見つける処理のオーバヘッドは、CALL 識別子 を検索するのと同じことです。 START 文に手続きポインタを使用すると、このオーバヘッドを回避できます。

作成された各スレッドの開始点のパラメータは 1 つのみです。 このパラメータを渡すときに BY CONTENT 指定を使用すると、システムはスレッドを作成する前にコピーを作成し、呼び出し側のスレッドが START 文からの戻り値に基づいて元のパラメータを自由に変更できるようにします。

ハンドルがデタッチされていないスレッドを作成 (IDENTIFIED BY 句を使用) すると、WAIT 文を使用して戻り値を後で取得できます。 WAIT 文の終了後、指定したスレッドハンドルは無効となり、そのスレッドに関連付けられたリソースはすべて解放されます。

作成されたスレッドで STOP RUN RETURNING 文を使用すると、実行単位は終了しません。この文は戻り値を返し、スレッドを終了するのみです。 これは、次のコードと同等です。

```
call 'CBL_THREAD_EXIT' using by value address of thread-parm.
```

他言語スレッドの STOP RUN、または CBL_THREAD_CREATE 以外で作成された実行単位の主スレッドは、アクティブな COBOL スレッドがすべて終了するのを待って、実行単位を終了します。

スレッドからの戻り値は常にポインタです。 このポインタにより、簡単なデータ構造体と

複雑なデータ構造体の両方を返すことができます。

スレッドは、STOP RUN を実行するか、または、CBL_THREAD_EXIT を呼び出してスレッド自身を終了できます。 また、通常、EXIT PROGRAM または GOBACK により開始点のプログラムが終了すると、スレッドも終了します。 また、他のスレッドが COBOL スレッドを終了するのに役立つ場合もあります。 たとえば、CBL_THREAD_KILL ルーチンを使用すると、COBOL スレッドが終了します。

例 - スレッドの作成と終了

```
$set reentrant

 Data Division.
 Working-Storage Section.
 01 thread-handle           usage pointer.
 01 thread-return           usage pointer.
 Linkage Section.
 01 thread-parm             picture x(32).
 01 thread-return-record    picture x(32).
 Procedure Division.
*> 開始点
     call 'CBL_THREAD_CREATE'
           using    'CREATED'
                    'This is a 32 character parameter'
           by value 0  *> オプションパラメータのサイズ
                    1   *> デタッチしないためのフラグ
                    0  *> デフォルトの優先順位
                    0  *> デフォルトスタック
           by reference  thread-handle
     if  return-code = 0
         call 'CBL_THREAD_WAIT' using
                              by value thread-handle
                              by reference thread-return
         set address of thread-return-record
                                          to thread-return
         display thread-return-record
     end-if
     stop run.
Entry "CREATED" using thread-parm.
     display thread-parm
     stop run returning address of thread-parm.
```

このアプリケーションは、開始点が CREATED であるスレッドを作成します。 作成されたスレッドはパラメータを表示し、親スレッドで使用するためにそのパラメータのアドレスを返します。 CREATED スレッドの STOP RUN RETURNING は、実行単位を終了しません。かわりに、戻り値を返し、スレッドを終了します。 これは、次のコードと同等です。

```
call 'CBL_THREAD_EXIT' using by value address of thread-parm.
```

スレッドの取り消し

CBL_THREAD_CREATE で作成されたスレッドは、CBL_THREAD_KILL ルーチンで取り消すことができます。 CBL_THREAD_KILL を使用して、別の方法で作成されたスレッドを強制終了できません。 この場合には、COBOL スレッドはただちに異常終了されますが、通常は、CBL_THREAD_KILL を一般的なアプリケーションスレッドコントロールの一部として使用しないでください。 主な理由は、スレッドの終了時に、ユーザとシステムの同期リソースが、正しくロック解放されないためです。 そのため、ユーザアプリケーションとランタイムシステムの間の同期が影響を受け、アプリケーションで重大な問題が発生することがあります。

CBL_THREAD_KILL は、主スレッドの重大エラーハンドラでは、適切に使用できます。 このエラーハンドラでは、スレッドハンドルを CBL_THREAD_LIST_n ルーチンによって獲得できます。スレッドをすべて取り消し、STOP RUN でアプリケーションを終了します。 この方法では、同期プリミティブをロックする必要性が最小限になるため、CBL_THREAD_KILL のランダム使用に比べると危険ではありません。ただし、実行単位の終了時にファイル破壊やデッドロックが発生する可能性はあります。

いずれにせよ、ほとんどのアプリケーションでは CBL_THREAD_KILL を使用しないようにしてください。 そのためには、終了フラグをもつスレッド識別データを作成 (CBL_THREAD_IDDATA_ALLOC ルーチンを使用) します。 そのデータは、ロックが保持されていないレベルの各スレッドでポーリングできます。 終了フラグが設定されている場合は、ポーリングスレッドを正常に終了できます。

例 - スレッド識別データと終了フラグの作成

```
******************** MAINPROG.CBL ********************
 identification division.
 program-id. mainprog.

 Data Division.
 Local-Storage Section.
 01  iddata-ptr              usage pointer.
 01  sub-iddata-ptr          usage pointer.
 01  sub-handle              usage pointer.
 Linkage Section.
 01  iddata-record.
     05  iddata-name         pic x(20).
     05  iddata-term         pic x comp-x value 0.

 Procedure Division.

* 識別データを確立します。- 識別データの割り当て時に
* データは初期化されません。
* ポインタの取得後に初期化を行います。
```

```
    call 'CBL_THREAD_IDDATA_ALLOC' using
                                by value zero
                              length of iddata-record
    call 'CBL_THREAD_IDDATA_GET'   using iddata-ptr
                               by value 0
    set address of iddata-record to iddata-ptr
    move 'main' to iddata-name

* サブスレッドを作成します。

* 開始点
    call 'CBL_THREAD_CREATE' using
                            'SUBPROG '
                 by value 0   *> パラメータはありません。
                          0   *> オプション - パラメータのサイズ
                          0   *> デタッチするためのフラグ
                          0   *> デフォルトの優先順位
                          0   *> デフォルトスタック
              by reference sub-handle
    if  return-code not = 0
        display 'スレッドを作成できません。'
        stop run
    end-if

* 子スレッドが識別データを作成するまで待機します。
* その後、終了フラグを設定します。

    set sub-iddata-ptr to NULL
    perform until 1 = 0
        call 'CBL_THREAD_IDDATA_GET'
               using    sub-iddata-ptr
               by value sub-handle
        if  sub-iddata-ptr not = null
            exit perform
        end-if
        call 'CBL_THREAD_YIELD'
    end-perform
    set address of iddata-record to sub-iddata-ptr
    move 1 to iddata-term

* 子スレッドがこのスレッドを再開するまで待機します。

    call 'CBL_THREAD_SUSPEND' using by value 0
```

```
     display 'RTS の終了時にすべての同期が' &
             '完了します。'
     stop run.
 end program mainprog.
************************ SUBPROG.CBL ***********************
 identification division.
 program-id. subprog.

 Data Division.
 Working-Storage Section.
 01 sub-iddata.
    05  sub-name             pic x(20) value 'sub'.
    05  sub-term             pic x comp-x value 0.
Local-Storage Section.
 01  iddata-ptr              usage pointer.
 01  thread-handle           usage pointer.
 01  thread-state            pic x(4) comp-x.
 01  parent-handle           usage pointer.
Linkage Section.
 01  iddata-record.
     05  iddata-name         pic x(20).
     05  iddata-term         pic x comp-x value 0.
 Procedure Division.

* 識別データを確立します。- データは初期化されます。
     call 'CBL_THREAD_IDDATA_ALLOC' using
                                  sub-iddata
                          by value length of sub-iddata

* 親スレッドを検索し、これを再開します。
*
     call 'CBL_THREAD_LIST_START' using thread-handle
                                        thread-state
                                        iddata-ptr
     set parent-handle to NULL
     perform until thread-handle = null
                or return-code not = 0
         if iddata-ptr not = null
             set address of iddata-record to iddata-ptr
             if  iddata-name = 'main'
                 set parent-handle to thread-handle
                 exit perform
             end-if
         end-if
         call 'CBL_THREAD_LIST_NEXT' using thread-handle
```

```
                                                thread-state
                                                iddata-ptr
    end-perform
    call 'CBL_THREAD_LIST_END'
    if  parent-handle = NULL
        display '同期エラー'
        stop run
    end-if
    call 'CBL_THREAD_RESUME' using by value parent-handle
    call 'CBL_THREAD_IDDATA_GET'   using iddata-ptr
                                   by value 0
    set address of iddata-record to iddata-ptr
    perform until iddata-term = 1
        call 'CBL_THREAD_YIELD'
    end-perform
    exit program.
end program subprog.
```

この長めのコード例では、実際にはスレッドとアプリケーションの終了についてハンドシェイクを確立するのみです。 これについて説明する前に、主スレッドのハンドルをパラメータとして子スレッドに渡せば、このようなハンドシェイクの確立をより簡単に行えることに注目する必要があります。 この方法を使用すると、識別データに依存したり、スレッドリストをステップ実行する必要はありません。

最初に、スレッド識別データの 2 種類の作成方法に注目してください。 最初の方法は、親スレッドで、初期化済みでないデータを作成し、識別データのポインタを獲得して、割り当てられたメモリ領域を初期化します。 もう 1 つの方法は、子スレッドで、初期化済みのデータを作成し、アプリケーションによる識別データの割り当てと初期化が同時に実行される可能性を排除します。 アプリケーションで使用する方法は、予測する識別データの競合の程度によります。

また、子スレッドがスレッド識別データを作成するのを待つ親スレッドでのループと、親スレッドが終了フラグを設定するのを待つ子スレッドでのループにも注目してください。 CBL_THREAD_YIELD を呼び出すと、これらのループがハードウェアのビジー状態で待機するのを回避できますが、イベント同期オブジェクト、条件同期オブジェクト、または CBL_THREAD_SUSPEND と CBL_THREAD_RESUME を使用してコードを作成することをお奨めします。

最後に、CBL_THREAD_LIST_ API を使用している点に注目してください。 この API により、スレッドは、ランタイムシステムで認識されているすべてのスレッドをステップ実行し、スレッドハンドル、スレッド状態、識別データポインタを取得できます。 この例では、ハンドルおよび識別データポインタのみを使用しています。ただし、対象のスレッドが、デタッチされたスレッド、中断されたスレッド、または他言語のスレッドであるかどうかを呼び出しスレッドに通知することができる点で状態情報も便利です。

CBL_THREAD_LIST API が指定されていると、これを使用するスレッドは以後 CBL_THREAD_ を呼び出す場合に制約を受け、他のスレッドはすべて CBL_THREAD_ 呼び出しとその他いくつかのランタイムシステム呼び出しをまったく使用できなくなります。

この理由により、リストのステップ実行中に実行するコード量は最小限に抑え、リストがロックされている間は、ファイル入出力またはユーザ入出力を行わないことが重要です。この制限は、CBL_THREAD_LIST_END 呼び出しによりステップ実行が終了すると、すぐに解除されます。

スレッドの中断

多くのマルチスレッドアプリケーションでは、スレッドを終了しないで、「中断」するのは珍しいことではありません。 たとえば、クライアント / サーバアーキテクチャでは、主サービススレッドが要求を見つけるために入力キューをポーリングする場合があります。要求が見つからない場合は、再度入力キューをポーリングする前に CPU を別のプロセスまたはスレッドに使用させます。 これは、CBL_THREAD_YIELD を呼び出すことで簡単に行うことができます。CBL_THREAD_YIELD を呼び出すと、アプリケーションの別のスレッド (オペレーティングシステムによっては、別のプロセスの別のスレッド) に CPU を譲ります。

他に、スレッドが CPU の所有権を無期限に放棄し、何らかのイベントが確実に発生した場合のみに CPU を使用できます。 CBL_THREAD_SUSPEND により、中断されたスレッドのスレッドハンドルを使用して別のスレッドが CBL_THREAD_RESUME を呼び出すまで呼び出し側スレッドを中断できます。 スレッドは対象となるスレッドがそれ自身を中断する前に、CBL_THREAD_RESUME を 1 回以上呼び出せます。この場合には、CBL_THREAD_SUSPEND の呼び出しは呼び出し側スレッドにすぐに戻り、CPU の所有権を放棄しません。 この操作は、セマフォを数えるのに非常に似ています。作成者 / 使用者間の問題は、CBL_THREAD_SUSPEND と CBL_THREAD_RESUME ルーチンのみを使用すると解決します。

スレッドの識別

スレッドをアプリケーション内で作成された他のスレッドと区別すると役立つことがあります。 たとえば、4 つのスレッドをもつアプリケーションで 2 つのスレッドが作成側と使用側の関係をなす場合には、お互いのスレッドのスレッドハンドルを知ることはこれら連動する 2 つのスレッドにとって役に立ちます。 これらのハンドルを取得した後は、CBL_THREAD_SUSPEND と CBL_THREAD_RESUME の呼び出しのみを使用してすべての同期を取ることができます。 アプリケーション内の各スレッドがそれぞれの名前を作成し (かつ、名前がその機能に関連する)、その名前をスレッドハンドルに関連付けると、連動するスレッドはスレッド名のリストを検索し、お互いのハンドルを検出できます。

グローバルにアクセス可能なデータを各スレッドとそのハンドルに関連付けると、終了フラグを保持し、CBL_THREAD_KILL を使用する可能性を未然に防ぐことができます。 アプリケーションの各スレッドは、終了要求を確認するためにその終了フラグをポーリングできます。 終了するスレッドは、正常に終了する前に、ロックが保持されていないことと、アクションの同期を取る必要がないことを確認します。

スレッドのグローバルにアクセス可能なデータは、スレッド内で CBL_THREAD_IDDATA_ALLOC ルーチンを実行するとスレッドハンドルに関連付けられます。 スレッドハンドルがすでに認識されている場合は、CBL_THREAD_IDDATA_GET を呼び出してデータを取得します。スレッドハンドルが認識されていない場合は、n ルーチンを使用してこのデータを取得します。

例 - グローバルにアクセス可能なデータとスレッドハンドルの関連付け

次の例では、スレッド内で CBL_THREAD_IDDATA_ALLOC を呼び出して、グローバルにアクセス可能なデータをスレッドハンドルに関連付ける方法を示します。 スレッドハンドルがすでに認識されている場合は、CBL_THREAD_IDDATA_GET を呼び出してデータを取得します。スレッドハンドルが認識されていない場合は、CBL_THREAD_LIST_n ルーチンを使用してこのデータを取得します。

```
********************* MAINPROG.CBL ********************
 identification division.
 program-id. mainprog.

 Data Division.
 Local-Storage Section.
 01 iddata-ptr       usage pointer.
 01 sub-iddata-ptr   usage pointer.
 01 sub-handle       usage thread-pointer.
 Linkage Section.
  01  iddata-record.
     05 iddata-name  pic x(20).
   05 iddata-term    pic x comp-x value 0.

 Procedure Division.

* 識別データを確立します。- 識別データの割り当て時に
* データは初期化されません。
* ポインタの取得後に初期化を行います。

     call 'CBL_THREAD_IDDATA_ALLOC' using
                                    by value zero
                                    length of iddata-record
     call 'CBL_THREAD_IDDATA_GET'   using iddata-ptr
                                    by value 0
     set address of iddata-record   to iddata-ptr
     move 'main' to iddata-name

* サブスレッドを作成します。

     start 'SUBPROG '  identified by sub-handle

* 子スレッドが識別データを作成するまで待機します。
* その後、終了フラグを設定します。

     set sub-iddata-ptr to NULL
     perform until 1 = 0
         call 'CBL_THREAD_IDDATA_GET' using sub-iddata-ptr
```

```
                                        by value sub-handle
         if  sub-iddata-ptr not = null
             exit perform
         end-if
         call 'CBL_THREAD_YIELD'
     end-perform
     set address of iddata-record to sub-iddata-ptr
     move 1 to iddata-term

*> 子スレッドがこのスレッドを再開するまで待機します。

     call 'CBL_THREAD_SUSPEND' using by value 0
     display 'RTS の終了時にすべての同期が' &
             '完了します。'
     wait for sub-handle   *> スレッドのリソースを解放します。
     stop run.
 end program mainprog.
************************** SUBPROG.CBL ****************************
 identification division.
 program-id. subprog.

 Data Division.
 Working-Storage Section.
 01 sub-iddata.
    05 sub-name           pic x(20) value 'sub'.
    05 sub-term           pic x comp-x value 0.
 Local-Storage Section.
 01 iddata-ptr            usage pointer.
 01 thread-handle         usage pointer.
 01 thread-state          pic x(4) comp-x.
 01 parent-handle         usage pointer.
 Linkage Section.
 01  iddata-record.
    05 iddata-name        pic x(20).
    05 iddata-term        pic x comp-x value 0.
Procedure Division.

* 識別データを確立します。- データは
* 初期化されます。
     call 'CBL_THREAD_IDDATA_ALLOC'
                          using sub-iddata
                          by value length of sub-iddata

* 親スレッドを検索し、これを再開します。
```

```
    call 'CBL_THREAD_LIST_START'
                      using thread-handle
                            thread-state
                            iddata-ptr
    set parent-handle to NULL
    perform until thread-handle = null
                or return-code not = 0
        if iddata-ptr not = null
            set address of iddata-record   to iddata-ptr
            if  iddata-name = 'main'
                set parent-handle to thread-handle
                exit perform
           end-if
        end-if
        call 'CBL_THREAD_LIST_NEXT' using thread-handle
                                          thread-state
                                          iddata-ptr
    end-perform
    call 'CBL_THREAD_LIST_END'
    if  parent-handle = NULL
        display '同期エラー'
        stop run
    end-if
    call 'CBL_THREAD_RESUME' using by value parent-handle
    call 'CBL_THREAD_IDDATA_GET' using iddata-ptr
                                 by value 0
    set address of iddata-record   to iddata-ptr
    perform until iddata-term = 1
        call 'CBL_THREAD_YIELD'
    end-perform
    exit program.
end program subprog.
```

この例では、スレッドとアプリケーションを終了するためのハンドシェイクを確立します。 このようなハンドシェイクは、主スレッドのハンドルをパラメータとして子スレッドに渡すことでより簡単に行えることに注目してください。 この方法を使用すると、識別データに依存したり、スレッドリストをステップ実行する必要はありません。

この例では、スレッドの識別データを作成する 2 種類の方法を示しています。 最初の方法は、親スレッドで、初期化済みでないデータを作成し、識別データのポインタを獲得して、割り当てられたメモリ領域を初期化します。 もう 1 つの方法は、子スレッドで、初期化されたデータを作成し、アプリケーションによる識別データの割り当てと初期化が同時に実行される可能性を排除します。 アプリケーションで使用する方法は、予測する識別データの競合の程度によります。

また、子スレッドがスレッド識別データを作成するのを待つ親スレッドでのループと、親スレッドが終了フラグを設定するのを待つ子スレッドでのループにも注目してください。

CBL_THREAD_YIELD を呼び出すと、これらのループがハードウェアのビジー状態で待機するのを回避できますが、イベント同期オブジェクト、条件同期オブジェクト、または CBL_THREAD_SUSPEND と CBL_THREAD_RESUME を使用してコードを作成することをお奨めします。

最後に、CBL_THREAD_LIST_n ルーチンを使用している点に注目してください。 これらのルーチンにより、スレッドは、ランタイムシステムで認識されているすべてのスレッドをステップ実行し、スレッドハンドル、スレッド状態、識別データポインタを取得できます。この例では、ハンドルおよび識別データポインタのみを使用しています。ただし、対象のスレッドが、デタッチされたスレッド、中断されたスレッド、または他言語のスレッドであるかどうかを呼び出しスレッドに通知することができる点で状態情報も便利です。

CBL_THREAD_LIST_n ルーチンが指定されていると、これを使用するスレッドは以後 CBL_THREAD_n を呼び出す場合に制約を受け、他のスレッドはすべて CBL_THREAD_n 呼び出しとその他いくつかのランタイムシステム呼び出しをまったく使用できなくなります。 この理由により、リストのステップ実行中に実行するコード量は最小限に抑え、リストがロックされている間は、ファイル入出力またはユーザ入出力を行わないことが重要です。 この制限は、CBL_THREAD_LIST_END 呼び出しによりステップ実行が終了すると、すぐに解除されます。

他言語のスレッド

複数言語を使用する環境では、オペレーティングシステムの機能によって直接作成されたスレッドと、START 文または CBL_THREAD_CREATE ルーチン以外で作成されたスレッドに対してはいくつかの制限と要件があります。

- これらのスレッドは、CBL_THREAD_n ルーチンの関数 (たとえば、CBL_THREAD_EXIT または CBL_THREAD_KILL) ではなく、オペレーティングシステムの正当なスレッド終了方法で終了する必要があります。
- 作成されたスレッドハンドルは常にデタッチされています。 そのため、これらのスレッドは、WAIT 文または CBL_THREAD_WAIT ルーチンを使用して待機できません。戻り値はありません。
- スレッドまたはアプリケーションのそれぞれで COBOL ランタイムシステムのサービスが不要になった場合には、スレッドは、`cobthreadtidy()` および `cobtidy()` を呼び出す必要があります。 この呼び出しにより、ランタイムシステムはメモリを解放し、スレッド状態の情報をクリアできます。
- `cobtidy()`、`cobexit()` または COBOL STOP RUN 文により、呼び出し側スレッドは「主」スレッドとして処理されます。 その結果、ランタイムシステムは、クリア処理の完了前に、CBL_THREAD_CREATE スレッドがすべて終了するのを待ちます。

注：ランタイムシステムがサポートするすべてのマルチスレッド機能は、他言語スレッドで作成されたハンドルを検出し、使用状況が無効な場合に報告します。 ただし、(上記で説明するとおり) COBOL またはランタイムシステムコードを実行したスレッドが終了する前に

このスレッド対するランタイムシステムのリソースを解放するかどうかは完全にアプリケーションに依存します。

呼び出されるプログラムの取り消し

CANCEL 操作では、操作の完了時間が制限されるため、CANCEL 操作が終了する前に取り消されたモジュールがスレッドで実行される可能性があります。 このため、マルチスレッドアプリケーションでは、CANCEL 文を使用しないことをお奨めします。

最適化とプログラミングのヒント

最適化された効率的なプログラムを作成するには、次のことに注意してください。

- アプリケーションがシングルスレッドランタイムシステムとマルチスレッドランタイムシステムのどちらで実行されているかを確認するコードを、アプリケーションに常に追加してください。
- 可能であれば、NOREENTRANT コンパイラ指令と NOSERIAL コンパイラ指令 (デフォルト) を使用してください。 これらのコンパイラ指令を使用すると、スタック領域と呼び出しエントリのオーバヘッドを最小化できます。ただし、これらの指令を使用するプログラムを一度に 1 つのスレッドのみでアクティブにするのはプログラミングによります。
- REENTRANT(1) コンパイラ指令を使用すると、スタック領域の使用量が増加しますが、プログラムの呼び出し速度に大きく影響することはありません。
- SERIAL プログラム属性を使用すると、呼び出し速度に大きく影響しますが、プログラムに必要なスタック領域の使用量には影響しません。
- Local-Storage Section を使用してスレッドの作業変数を保持します。 これにより、スタックの使用量が増えますが、データの割り当ては高速で行います。
- Thread-Local-Storage Section を使用すると、呼び出し速度に大きく影響します。 Working-Storage Section のかわりに Thread-Local-Storage Section を使用すると、簡単にアプリケーション全体でスレッドを保護できます。
- 簡単なロック処理には、ミューテックスを使用してください。 ミューテックスは、最も高速な同期構造体です。
- 読み書きの問題が発生する可能性のあるプログラムでマルチスレッドを最も効率化するには、モニタを使用します。
- シングルスレッドモードでは、可能であれば同期データ項目をすべて初期化してください。 初期化できない場合は、first-time フラグを使用してください。このフラグは、CBL_THREAD_PROG_LOCK を呼び出す前に一度チェックされ、ロックを取得し

た後に再度チェックされます。

- スレッドとスレッドハンドルの相違を忘れないでください。 デタッチされていないスレッドは、必要な場合のみに作成します。
- CBL_THREAD_KILL は、どうしても必要な場合のみに使用します。使用する場合は、実行後すぐに実行単位を終了してください。 可能な場合は、かわりに終了ポーリングを使用します。
- CBL_THREAD_YIELD よりも、CBL_THREAD_SUSPEND を使用します。 ビジーによる待機状態は回避してください。
- CBL_THREAD_LIST_START 呼び出しと CBL_THREAD_LIST_END 呼び出しの間の作業量は最小限にします。
- 特定の状況では、アプリケーションは Thread-Local-Storage Section よりも CBL_TSTORE_n ルーチンを使用して、各スレッドメモリを保存することができます。
- 発生したデッドロックを修正するのではなく、デッドロックが発生しないように注意してください。
- 読み込みロックをモニタにネストさせないでください。ネストさせる必要がある場合は、シングルスレッドのデッドロックが発生する可能性があることを認識してください。

マルチスレッドアプリケーションで使用できるコンパイラ指令は、マルチスレッドアプリケーションおよびシングルスレッドアプリケーションの性能に影響します。

マルチスレッドアプリケーションの作成

第 4 章 : マルチスレッドコンパイラ指令

次の 2 つのコンパイラ指令は、マルチスレッドプログラムをシリアル化し、再入を可能にします。

- REENTRANT - プログラムの再入を制御します。
- SERIAL - プログラムのシリアル化を制御します。

次のコンパイラ指令は、マルチスレッドプログラムの性能を最適化できます。

- PARAMCOUNTCHECK - エントリポイントのパラメータ数のチェックを制御します。
- FASTCALLS - 呼び出されるプログラムの動作を制御します。
- FIXOPT - 特定の制御領域の配置を制御します。
- FASTLINK - パラメータの処理を制御します。

コンパイラ指令のデフォルト設定は次のとおりです。

```
PARAMCOUNTCHECK NOFASTCALLS NOREENTRANT NOSERIAL FIXOPT
```

FASTLINK 指令をこれらのデフォルトの指令に設定した場合は、TYPECHECK と FIXOPT が設定され、動作しないので注意してください。 プログラムを最適化して処理速度を向上させる場合は、指令を次のように設定してください。

```
NOTYPECHECK
FASTCALLS
NOREENTRANT
NOSERIAL
NOFIXOPT
FASTLINK
```

これらのコンパイラ指令を利用するには、プログラムの変更が必要になることがあるので注意してください。

マルチスレッドアプリケーションの作成