Net Express

マルチスレッド プログラミング

Micro Focus NetExpress™

マルチスレッド プログラミング

Micro Focus®

第 4 版
1998 年 10 月

序文

このマニュアルでは、NetExpress 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細に説明し、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。

対象読者

このマニュアルは、NetExpress を使用してマルチスレッド COBOL プログラムを作成するすべてのプログラマとシステム設計者を対象としています。また、ビジネス コンピューティング、Microsoft Windows および NetExpress の一般的な概念に関して十分な知識をお持ちの方が対象になります。

関連マニュアル

- NetExpress および COBOL システムの他のコンポーネントのオンライン ヘルプ

表記規約

- **Enter** は、キャリッジ リターンまたは Enter キーを示します。コマンドを入力する箇所では、Enter キーは明示的には示されていません。暗黙的に行の終わりで Enter キーを押すことを示しています。
- 16進数は引用符で囲み、その前に小文字の "x" または "h"を記します。例えば、x"9D"、 h"03FF"のようになります。 "x" は 16 進数が文字列を表すときに使用します。"h"は、数値を表すときに使用します。
- COMP-X データ型、COMP-5 データ型、および PIC 99 を併用せずに、PIC X を使用します。 PIC 99 とは異なり、PIC X はデータ項目の大きさを示します。また、COMP-X も、指定されたバイト数のバイナリ項目を定義するのでより詳細に説明できます。
- キートップおよびメニューの選択肢は強調表示されています。
- 使用する環境によっては、このマニュアルに示される画面表示と実際の画面表示が多少異なる場合があります (例えば、バージョン番号など)。この違いは、ソフトウェアの動作には影響しません。
- このマニュアルで使用するキーは、環境によっては使用できないことがあります。状態キーやファンクション キーなどを使用する箇所では、物理的なキーストロークではなく、そのキーに割り当てられた動作を選択すると考えてください。指定されたキーが使用環境でサポートされていない場合は、付属の *リリース ノート* を参照して同等のキーを見つけてください。
- 「ウィンドウ」は画面上の区切られた領域を示します。通常はフル画面よりも小さい領域です。"Windows" は Microsoft Windows 3.1 またはそれ以降のソフトウェア製品を示します。
- オンライン ヘルプはこのマニュアルでは記述されていません。メニューで **[ヘルプ]**を選択するか、またはダイアログ ボックスで **[ヘルプ]** ボタンを押して、必要な箇所でヘルプ情報を表示してください。

コマンド行の書式表記は次のとおりです。

- **斜体は、ユーザーが入力する名称を示す総称的な用語です。**
- **角かっこ ([ ]) は、オプションを示します。**
- **中かっこ ({ }) は、 { } の中からオプションを選択する必要があることを示します。{ } にオプションが 1 つしかない場合は、そのオプションの反復を示します。**
- **中かっこ { } または角かっこ [ ] に続く省略 (. . .) は 各かっこ内のオプションを反復できることを意味します。特に明記されていない限り、反復回数に制限はありません。角かっこ [ ] と省略 (. . .) が使用される場合は、オプションをすべて省略できます。**
- **コマンド行がページ内に収まらない場合は、次の行に継続されます。継続行は字下げされます。**
- **コマンド行のオプションは、 / オプションまたは - オプションで指定できます。**

目次

序文 ii

対象読者 ii

関連マニュアル ii

表記規約 ii

第1章 マルチスレッドの概要 1-1

1.1 マルチスレッドとオペレーティング システム 1-1

1.2 マルチスレッドとアプリケーション 1-2

第2章 実行の同期と競合の解決 2-1

2.1 マルチスレッドのプログラム属性 2-1

2.1.1 非マルチスレッド指定 2-1

2.1.2 シリアル プログラム 2-2

2.1.3 再入可能なプログラム 2-2

2.2 データ属性の使用 2-3

2.3 同期プリミティブの使用 2-4

2.4 ミューテックスの使用 2-5

2.4.1 モニタの使用 2-6

2.4.2 セマフォの使用 2-8

2.4.3 イベントの使用 2-10

第3章 マルチスレッド アプリケーションの作成 3-1

3.1 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 3-1

3.2 再入可能なプログラム作成時の注意事項 3-2

3.3 マルチスレッド ライブラリ ルーチン 3-3

3.3.1 スレッド制御ルーチン 3-4

3.3.2 **スレッド同期ルーチン** 3-6

3.3.3 **スレッド固有データを処理するルーチン** 3-6

3.3.4 **アプリケーションの初期化** 3-6

3.4 **スレッドの操作** 3-8

3.4.1 **スレッド ハンドル** 3-8

3.4.2 **スレッドの作成と終了** 3-9

3.4.3 **スレッドの取り消し** 3-11

3.4.4 **スレッドの中断** 3-14

3.4.5 **スレッドの識別** 3-15

3.4.6 **他言語のスレッド** 3-18

3.5 **最適化とプログラミングのヒント** 3-19

第4章 マルチスレッド コンパイラ指令 4-1

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することができます。
- 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に基づくアプリケーションを指します。

1.1 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) を同時に実行することができます。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、厳密にはこれらのプロセスが同時に実行されるわけではありません。プロセスが同時に実行されているように見えるだけです。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 (優先順位に基づいて) 切り替えます。このリソースの切り替えにより、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することができます。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 リソースを個別に持っています。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 0 に抑えた状態で、各プロセスを実行することができます。ただし、リソースを個別に持つことで、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、および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持っています。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 セットの切り替えだけです。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。

1.2 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 システムのマルチスレッド機能を利用したアーキテクチャに基づくアプリケーションのことです。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 ユーザー 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うようにすることができます。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ー インターフェイスを完全に分離することができます。

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/サーバー アプリケーションのサーバー側で使用する方法もあります。サーバーを設計する場合に、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 スレッドを作成することができます。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すことができます。このアーキテクチャの主な利点は、サーバーが 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。

サーバーから応答をただちに (あるいは高速で) 得るクライアント/サーバー アプリケーションは、何年もの間マルチスレッドを使用せずに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。では、マルチスレッドがこれらのアプリケーションで必要なのはなぜでしょうか？ 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することができます。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を最大限に利用し、リソースの共有を簡略化することができます。

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壊される可能性があります。

たとえば、次に示すコードを考えてみてください。

```
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てみてください。

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期待した値を表示しません。 期待していたスレッド 1 の結果は 'AB' で、期待していたスレッド 2 の結果は 'DC' です。しかし、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。

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判断するためのデータ項目属性
- 同期プリミティブ

2.1 マルチスレッドのプログラム属性

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。
- シリアル属性を持たせます。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、起動時にロックされ、終了時にロック解除されます。
- 再入属性を持たせます。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する) 、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。

2.1.1 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 コンパイラ指令を全く指定しないと、非マルチスレッド プログラムにすることができます。この場合、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 つのスレッドだけ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

のスレッドだけしか実行できないようにすることはできます。例えば、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終了する前にそのロックを解除します。

2.1.2 シリアル プログラム

シリアル属性を持つプログラムでは、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、終了時にロック解除されます。このロック機能により、一度に 1 つのスレッドだけ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ー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ることができます。

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せずにマルチスレッド 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に制限されます。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除処理にかなりのランタイム 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。例えば、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、アプリケーションの設計により、プログラム B と C は既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しておいて良いのです。

2.1.3 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することができます。マルチスレッド アプリケーションでは、ほとんどの (すべてではない場合) 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、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ー 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、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スタック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除は必要ありません。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、 なるべく 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り得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、次のセクションを参照してください。

2.2 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 で定義したデータ
- ファイル状態フィールドなどの暗黙的に定義した他のデータ項目
- 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することもできます。

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がなくなることです。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

ータ項目の値は不特定なものになります。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 句で初期化することができます。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、ファイル処理を行わないプログラムで Working-Storage Section ヘッダを Thread-Local-Storage Section ヘッダに変更するだけで、完全に再入可能なプログラムにすることができます。読み込み専用の定数はすべて Working-Storage Section で、読み込み/書き込みデータはすべて Thread-Local-Storage Section で定義すると、データの割り当てを微調整することができます。

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、適切なスレッドの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があります。
- EXTERNAL 項目を Thread-Local-Storage Section で指定することはできません。現在、スレッド ローカル データをアプリケーションのモジュール間で共有しなければならない場合は、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、次の項を参照してください。

2.3 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 アクセスの特徴を理解すると、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。次のトピックでは、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。

2.4 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することができます。「ミューテックス (mutex)」は、**mut**ual **ex**clusion (相互排他) から派生した言葉です。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除されます。

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、予期しない結果が発生します。

例えば、次のコードはテーブルに項目を追加するため、またはテーブル内の項目数を調べるためにそのテーブルにアクセスする 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、あるスレッドがテーブルを読み込んでいる間に別のスレッドがそのテーブル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-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

*> テーブル-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

*> テーブル-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を厳密に制限することです。例えば、 項目をテーブルに追加するプログラム、またはテーブル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は、複数のスレッドを使って同時にテーブル内の項目数を数えるようにしたいと考えるでしょう。しかし、あるプログラムがテーブル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しようとは思わないはずです。

この問題を解決するには同期プリミティブを使います。同期プリミティブは、読み込み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。

2.4.1 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。例えば、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 つのスレッドだけ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することができます。

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たいへん役立ちます。参照モードでデータアクセスするクリティカル セクションは、データの読み込みや保護データ項目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かもしれません。このクリティカル セクションがアクティブの場合、単にデータを読み込むだけのクリティカル セクションは複数実行することができますが、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、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。ブラウザは、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(この書き込み前のデータ項目の状態は、ブラウザが保護データ項目を読み込んだときの状態と同じです)。

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、テーブル内の項目数を数えたり、テーブル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、項目をテーブルに追加するスレッドを実行しません。
- テーブルに項目を追加するスレッドを 1 つだけ実行します。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テーブル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

*> テーブル-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

*> テーブル-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 - モニタを使用するブラウザのクリティカルセクション

簡単なブラウザのクリティカル セクションを次に示します。

```
 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

*> テーブル-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勧めしません。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判断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だけです。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。

2.4.2 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、ミューテックスと同じようなもので、ミューテックスの代わりに使用することができます。

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             *> 初期化時に値を増やします。

*> データ値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
 set data-semaphore up by 1             *> セマフォにより他のスレッドの実行が可能になりました。
```

OPEN 文の直後で、セマフォが 1 追加されていることに注目してください。これにより、最初のスレッドは処理さ

れますが、それ以降のスレッド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だけで、他のスレッドはその解放されたセマフォを獲得することができます。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することができます。

```
 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、データの値を生成するために 1 度実行されます。
        set consumed-semaphore           down by 1
        add 10 to data-value
        set produced-semaphore           up by 1 *> データ値の変更を通知するシグナル
です。

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

この例では、作成側と使用側の問題として知られる、もう 1 つの一般的な同期問題を示しています。最も単純な作成側と使用側の問題は、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)を生成することができます。

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 テーブルが一杯になるまでデータの値を作成することができます。データ テーブルが一杯になると、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-valuepic 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です。
        open produced-semaphore
        open consumed-semaphore
        set consumed-semaphore  up by table-size *> 値を 20 回増やします。

*> 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.
```

2.4.3 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、セマフォ同期の代わりにイベント同期を使用して作成側と使用側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です。
        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        *> イベントを解除します。
        add 10 to data-value
*> データ値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。データ変更を待ちます。
        wait for produced-event
        set produced-event      to false        *> イベントを解除します。
        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。このスレッドはデータ値を持っています。
        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)だけの場合、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、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 つのスレッドだけ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 (イベントの解除を含む) に対して何らかのアクションを行うべきかどうかを判断する必要があります。

最後の点は、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。

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、次のこと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。

3.1 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 COBOL プログラム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、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、さらにはシングルスレッド 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。アプリケーショをリンクする時点で、アプリケーションの実行時にどちらのランタイム システムを使用するかを選択することができます。

アプリケーションと一緒に使用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、ヘルプのトピック *「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。

3.2 再入可能なプログラム作成時の注意事項

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には、以前に説明した同期プリミティブの 1 つを使用した明示的なユーザー保護が必要です。ファイル ハンドラは内部的にスレッドを保護します。しかし、ユーザー 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せずに設定されます。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。ファイル処理文を保護する手段を取ってください。
- 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 つのスレッドだけ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。例えば、レコードのロック処理です。この場合、 他のスレッドは、遅延している読み込みが終了するまで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することはできません。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 つでユーザー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します。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 つだけにするのが最良の方法です。

- ネストしたプログラムを、新規作成したスレッドの開始点にすることはできません。スレッド作成の開始点にできるのは、エントリ ポイント、コンパイル単位の一番外側のプログラム、および他言語の外部エントリ ポイントだけです。

- ACCEPT 文の実行は、小入力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 (小入力) 操作または DISPLAY (表示) 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 エントリのオーバヘッドが大きくなります。可能な場合には、代わりに 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 処理速度を最適化すると、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。

3.3 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 インタフェースであり、これを介してアプリケーションにマルチスレッドを導入し、制御することができます。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

3.3.1 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することができます。

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が、ランタイム システムのサービスを使用すると、このようなスレッドもランタイム 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。しかし、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、次のいず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。この呼び出しにより、ランタイム システム のサービスが不要になり、スレッド状態データをクリアしてもよいことをランタイム システムに対して通知します。

- `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チェックすることができます。

例 - プログラムがサポートするランタイム システムのチェック

```
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
```

3.3.2 スレッド同期ルーチン

NetExpress では、モニタ、セマフォ、ミューテックス、およびイベントの 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。

3.3.3 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 データを作成することができます。

3.3.4 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判断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 システムのチェック

```
 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か
*> 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、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することができます。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一回だけ初期化することができます。

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既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 つのスレッドだけはロックを獲得します。ロックを獲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獲得している間に、first-time フラグが偽に設定されます。これ以後にロックを獲得しようとするスレッドは、first-time が偽なので初期化が既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除します。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 ロックの獲得を行いません (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)。

3.4 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 構文と CBL_THREAD_*n* ルーチンは汎用性があるために、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すことができます。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することができます。

3.4.1 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、様々なマルチスレッドの COBOL 文または CBL_THREAD_*n* ルーチンへのハンドルを一意に識別します。十分に注意を払えば、作成スレッドだけでなく、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 ハンドルを使用することができます。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ーが判断します。

スレッドからハンドルを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、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することができます。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既に終了している場合、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 ルーチンを使用して)、およびスレッドの終了後に (CBL_THREAD_IDDATA_GET ルーチンを使用して) 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。

3.4.2 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 ID、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のいずれかでなければなりません。ネストされたプログラム名、節名、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することができます。テキスト文字列を使用してエントリ ポイントを検索するオーバヘッドは、 CALL *識別子* を検索するのと同じことです。START 文の目的語として手続きポインタを使用すると、このオーバヘッドを回避することができます。

作成された各スレッドの開始点のパラメータは 1 つだけです。このパラメータを渡すときに語句 BY CONTENT 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 句を使用して)すると、WAIT 文を使用して戻り値を後で取得することができます。WAIT 文が終了した後、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だけです。これは、次のコードと同じ意味になります。

```
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、CBL_THREAD_EXIT を呼び出してスレッド自身を終了することができます。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。例えば、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
                   *> デタッチしないためのフラグ
                 0  *> デフォルトの優先度
                 0  *> デフォルトのスタック
         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。代わりに、戻り値を返し、スレッドを終了します。これは、次のコー

ドと同じことになります。

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

3.4.3 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することはできません。この場合、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、ユーザーとシステムの同期リソースが、正しくロック解除または解放されないためです。そのため、ユーザー 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、適切に使用することができます。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、終了フラグを持つスレッド識別データを作成 (CBL_THREAD_IDDATA_ALLOC ルーチンを使用して) 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することができます。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することができます。

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
*> 初期化データを提供しません。
*> 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   *> デフォルトの優先度
                                0   *> デフォルトのスタック
                     by reference sub-handle
 if  return-code not = 0
     display 'unable to create thread'
     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
*>
*> 子がこのスレッドを回収するまで待ちます。
*>
   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。 - 初期化データを提供します。
 call 'CBL_THREAD_IDDATA_ALLOC'
                     using sub-iddata
                     by value length of sub-iddata
*>
*> 親スレッドを検索し、回収します。
*>
 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だけです。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。この方法をとると、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した方がよいでしょう。

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 ポインタを取得することができます。この例では、ハンドルおよび識別データ ポインタだけを使用しています。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全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 I/O またはユーザー I/O を行わない方がよいでしょう。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。

3.4.4 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、スレッドを終了せずに、「中断」するのは珍しいことではありません。例えば、クライアント/サーバー 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 の使用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にのみ CPU を使用することもできます。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することができます。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 回以上呼び出すことができます。この場合、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 の使用権を放棄しません。この操作は、セマフォを数えるのに非常に似ています。作成側と使用側の問題は、CBL_THREAD_SUSPEND と CBL_THREAD_RESUME ルーチンだけを使用すると解決します。

3.4.5 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。例えば、4 つのスレッドを持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、 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 の呼び出しだけ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することができます。

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、終了要求をチェックするためにその終了フラグをポーリングすることができます。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 ハンドルが既に認識されている場合は、CBL_THREAD_IDDATA_GET ルーチンを使用してこのデータを取得します。一方、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。

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 ハンドルが既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
*> 初期化データを提供しません。
*> ポインタの取得後に初期化します。
*>
 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取得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クリアします。
 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。- 初期化データを提供します。
*>
 call 'CBL_THREAD_IDDATA_ALLOC'
                  using sub-iddata
                  by value length of sub-iddata
*>
*> 親スレッドを検索し、これを取得します。
*>
 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。この方法をとると、識別データに依存したり、またはスレッド リストをステップ実行する必要はなくなります。

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した方がよいでしょう。

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 ポインタを取得することができます。この例では、ハンドルおよび識別データ ポインタだけを使用しています。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全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 I/O またはユーザー I/O を行わない方がよいでしょう。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。

3.4.6 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 (例えば、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させることができません。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、 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することができます。

- `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、クリア処理の開始前に、CBL_THREAD_CREATE スレッドがすべて終了するのを待ちます。

備考 : 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。しかし、(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。

3.5 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 スレッドとマルチスレッドのどちらのランタイム システム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必ず追加します。

- できるだけ、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 つのスレッドだけ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 の代わりに 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。

- 読み込み/書き込み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、できれば、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 ハンドルの違い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、必要な場合にだけ作成します。

- CBL_THREAD_KILL は、どうしても必要な場合にだけ使用します。使用する場合は、実行後できるだけ早く実行単位を終了してください。可能な場合は、代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。

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することができます。

- PARAMCOUNTCHECK - エントリ ポイント 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、プログラムを変更する必要がある場合があるので注意してください。

索引

アプリケーション

初期化 3-6

マルチスレッドの作成 3-1

アプリケーションの初期化 3-6

イベント 2-10

競合の解決 2-1

コンパイラ指令

SERIAL 2-2, 4-1

性能の最適化 4-1

マルチスレッド 4-1

REENTRANT 2-2, 4-1

コンパイラ指令によるプログラム性能の最適化 4-1

再入可能属性 2-2

再入可能なプログラム 2-2

性能の限界 3-3

制約 3-2

注意事項 3-1

作成側と使用側の問題 2-9

実行の同期 2-1

SERIAL **コンパイラ指令** 4-1

SERIAL **コンパイラ指令** 2-2

シリアル属性 2-2

シリアル プログラム 2-2

シングル スレッド アプリケーション

ランタイム システム 3-1

スレッド

作成 3-9

識別 3-14

終了 3-9

他言語 3-18

中断 3-14

取り消し 3-10

ランタイム システムにより作成したスレッド ... 3-7

スレッド ハンドル 3-8

デタッチ 3-8

スレッド固有データ ライブラリ ルーティン 3-6

スレッド制御ライブラリ ルーチン 3-4

スレッド操作 3-7

スレッド同期ライブラリ ルーチン 3-5

スレッドの作成 3-9

スレッドの識別 3-14

スレッドの終了 3-9

スレッドの中断 3-14

スレッドの同期 2-4

スレッドの取り消し 3-10

Thread-Local-Storage Section 2-3

スレッド ローカル データ属性 2-3

セマフォ 2-8

属性

再入可能 2-2

シリアル 2-2

スレッド ローカル 2-3

データ 2-3

データ

競合 2-3, 2-4

競合の解決 2-3

属性 2-3

データ競合の解決 2-3

同期プリミティブ 2-4

イベント 2-10

セマフォ 2-8

ミューテックス 2-4

モニタ 2-6

プログラミング

ヒント 3-18

プログラミングの最適化 3-18

プログラム

再入可能 2-2

シリアル 2-2

非マルチスレッド指定 2-1

プログラム属性 2-1

マルチスレッド

マルチスレッドアプリケーションの作成 3-1

マルチスレッド

アプリケーション 1-2

オペレーティング システム 1-1

コンパイラ指令 4-1

スレッドの同期 2-4

はじめに 1-1

非マルチスレッド指定 2-1

プログラム属性 2-1

ライブラリ ルーチン 3-3

マルチスレッド プログラム

性能の最適化 4-1

マルチスレッド アプリケーション

ランタイム システム 3-1

マルチスレッド アプリケーションの作成 3-1

ミューテックス 2-4

モニタ 2-6

ライブラリ ルーティン

スレッド固有データ 3-6

スレッド同期 3-5

ライブラリ ルーチン

スレッド制御 3-4

ライブラリ ルーチン 3-3

ランタイム システム 3-1

REENTRANT

コンパイラ指令 4-1

REENTRANT **コンパイラ指令** 2-2