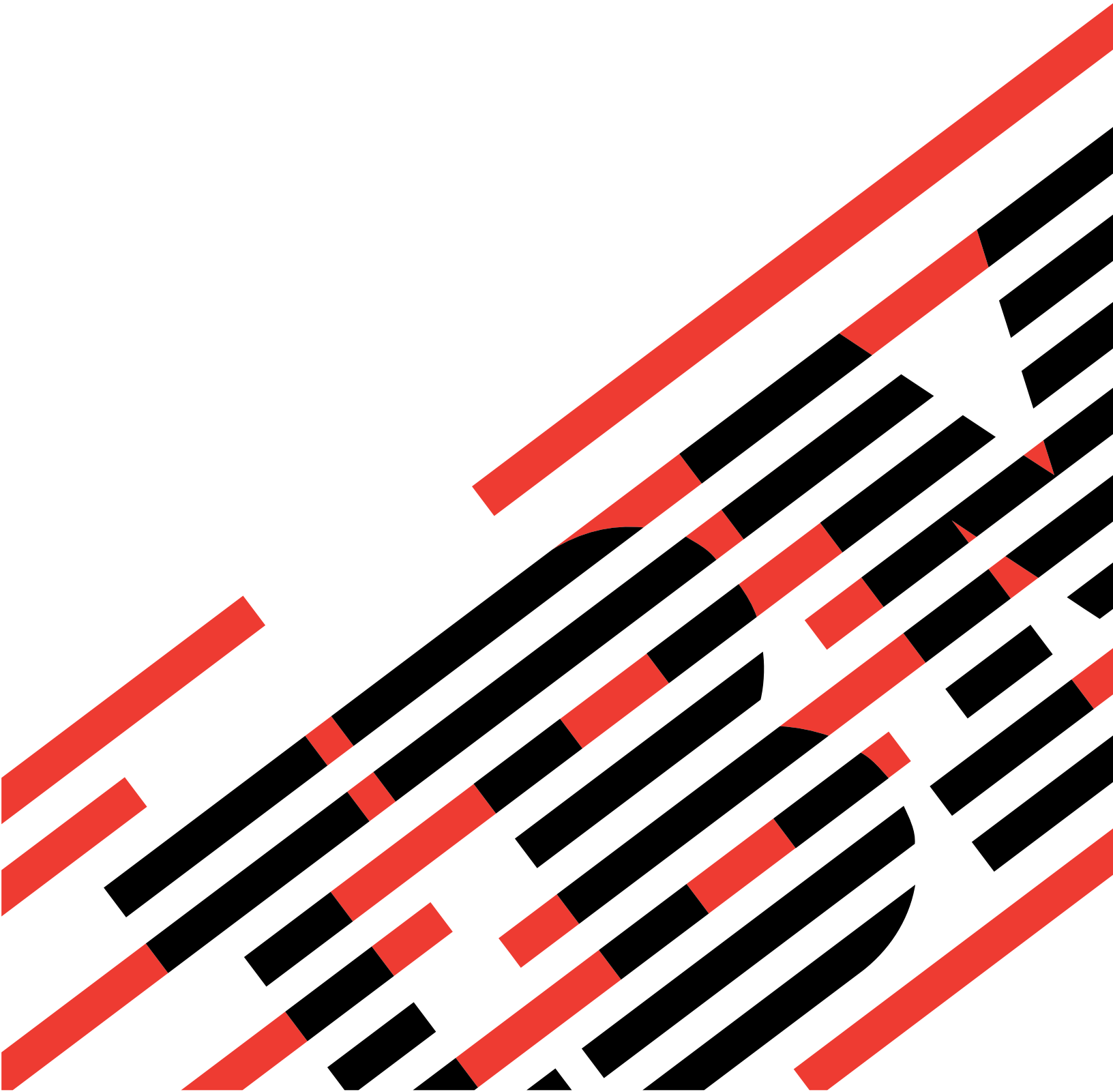
IBM

eserver

iSeries

マルチスレッド・アプリケーション



IBM

@server

iSeries

マルチスレッド・アプリケーション

目次

第 1 章 マルチスレッド・アプリケーション

スレッドとは

スレッドとは、実行時にプログラムの取るパス、実行されるステップ、およびそれらのステップが実行される順序のことです。スレッドは、コードをその開始位置から、特定の入力に対してあらかじめ定義された手順に従って順番通りに実行していきます。スレッドという語は、『制御スレッド』を短縮したものです。複数のスレッドを使用して、異なるアプリケーション・タスクを同時に実行することによって、アプリケーションのパフォーマンスを改善することができます。

OS/400 上のマルチスレッド・アプリケーションに関する作業の前に、以下のトピックについて十分に理解しておく必要があります。

スレッドの基本事項

- 5 ページの『第 3 章 OS/400 のスレッド』 - スレッドの基本概念と、 OS/400 でマルチスレッド・プログラムを使用するための考慮事項。
- 17 ページの『第 4 章 OS/400 スレッドの管理』 - OS/400 スレッド・カーネルにおける、基本的なスレッド管理タスクの説明。
- 35 ページの『第 5 章 スレッド・セーフティ』 - OS/400 のスレッド・セーフティの概念。

スレッドによるプログラミング

- 41 ページの『第 6 章 マルチスレッド・プログラミング・テクニック』 - マルチスレッド・プログラミングのさまざまなテクニックに関する情報。以下のものが含まれます。
  - アプリケーションの評価
  - 同期化のテクニック
  - リソースの初期化
  - スレッド固有のデータの使用
  - スレッド・セーフでない関数呼び出しの使用
  - 共通マルチスレッド・エラーの解決
- 67 ページの『第 7 章 言語アクセスとスレッド』 - OS/400 上での種々の言語によるスレッドのサポートの違い。
- 71 ページの『第 8 章 マルチスレッド・ジョブのデバッグとサービス』 - マルチスレッド・ジョブのデバッグとサービスに関する概念とテクニック。
- 81 ページの『第 9 章 マルチスレッド・アプリケーションにおけるパフォーマンスの考慮事項』

スレッドの使い方を示すコード (一部分)

- 85 ページの『第 10 章 例: スレッド』 - いくつかの一般的なスレッド・タスクの Pthread および Java の例。

第 2 章 トピックの印刷

この文書の PDF 版を参照用または印刷用にダウンロードし、表示することができます。 PDF ファイルを表示したり印刷したりするには、 Adobe® Acrobat® Reader が必要です。これは、Adobe Web サイト (www.adobe.com/prodindex/acrobat/readstep.html)  から、ダウンロードできます。

PDF 版をダウンロードし、表示するには、『マルチスレッド・アプリケーション』 (約 747 KB、94 ページ) を選択します。

表示用または印刷用の PDF ファイルをワークステーションに保存するには、次のようにします。
1. ブラウザーで PDF を開く (上記のリンクをクリックする)。
2. ブラウザーのメニューから、**「ファイル」**をクリックする。
3. **「名前を付けて保存」**をクリックする。
4. PDF を保存したいディレクトリーに進む。
5. **「保存」**をクリックする。

第 3 章 OS/400 のスレッド

スレッドという語は、『制御スレッド』を短縮したものです。スレッドとは、実行時にプログラムの取るパス、実行されるステップ、およびそれらのステップが実行される順序のことです。スレッドは、コードをその開始位置から、特定の入力に対してあらかじめ定義された手順に従って順番通りに実行していきます。

どのようなプログラムでも、*初期スレッド* と呼ばれる少なくとも 1 つのスレッドを持ちます。複数のスレッドを持つプログラムの場合、各スレッドは、プログラム内の他のスレッドと無関係にそれぞれのコードを実行します。

プロセスとは、プログラムのメモリーおよびリソースのコンテナーとなるものです。 iSeries システムでは、ジョブがプロセスに当たります。各プロセスには、プログラムがそのコードを実行するスレッド (タスク) が少なくとも 1 つあります。プロセス内の最初のスレッドのことを初期スレッドといいます。一部のプロセスではさらにいくつかのスレッドをサポートできますが、それらのことを 2 次スレッドといいます。

ここで説明する概念は、すべてのプログラミング言語に関係しています。それぞれの言語でこの概念がどのように実現されているかについては、各言語のプログラマーの手引きを参照してください。

スレッドの基本概念:

- 『スレッドのモデル』
- 6 ページの『スレッド・プログラムのモデル』
- 6 ページの『ジョブとジョブ・リソース』
- 7 ページの『スレッド・プライベート・データとスレッド固有データ』
- 8 ページの『マルチスレッド・プログラム起動コマンド』
- 8 ページの『活動化グループとスレッド』

マルチスレッド・プログラミングの考慮事項:

- 9 ページの『マルチスレッド・プログラミングでの通信についての考慮事項』
- 10 ページの『マルチスレッド・プログラミングでのデータベースについての考慮事項』
- 15 ページの『マルチスレッド・プログラミングでのデータ管理機能についての考慮事項』
- 16 ページの『マルチスレッド・プログラミングでのファイル・システムについての考慮事項』
- 16 ページの『マルチスレッド・プログラミングでの印刷ファイルについての考慮事項』

スレッドのモデル

スレッドの使用方法には、ユーザー・スレッドとカーネル・スレッドという異なるモデルがあります。

ユーザー・スレッド・モデルでは、すべてのプログラム・スレッドが同じプロセス・スレッドを共用します。ユーザー・スレッド・モデルでは、スレッドのアプリケーション・プログラミング・インターフェース (API) によってスケジューリング・ポリシーが常に適用され、新しいスレッドがいつ実行されるかが決まります。そのスケジューリング・ポリシーでは、1 つのプロセスの中でアクティブに実行できるのは一度に 1 つのスレッドだけです。オペレーティング・システム・カーネルは、プロセスの中で 1 つのタスクしか認識しません。

カーネル・スレッド・モデルの場合、カーネル・スレッドは 1 つのプロセスに関連する個別のタスクです。カーネル・スレッド・モデルでは、プリエンプティブ・スケジューリング・ポリシーが使われ、処理装置の使用権がどのスレッドに与えられるかがオペレーティング・システムによって決定されます。カーネル・スレッド・モデルでは、プログラミング・スレッドとプロセス・スレッドとが 1 対 1 に対応しています。 OS/400 では、カーネル・スレッド・モデルをサポートしています。

プラットフォームによっては、これらの 2 つのスレッド・モデルの組み合わせがサポートされており、これを通常 MxN スレッド・モデルといいます。各プロセスには、N 個のカーネル・スレッドを共用する M 個のユーザー・スレッドが含まれます。ユーザー・スレッドは、カーネル・スレッドより優先してスケジューリングされます。システムは、比較的『コストがかかる』カーネル・スレッドにしかリソースを割り振りません。

スレッド・プログラムのモデル

スレッド・プログラムの作成には、いくつかのモデルがあります。

OS/400 と他のプラットフォームとの大きな違いの 1 つは、 OS/400 ではコール / リターン・プログラム・モデルがサポートされるということです。その他のプラットフォームでは、あるプログラムが別のプログラムを呼び出す必要がある場合、第 2 のプログラムの実行のために第 2 のプロセスを開始しなければならないか、または最初のプログラムを第 2 のプログラムで置き換えなければなりません。プログラムを呼び出すために別のプロセスを開始することは、起動時間およびシステム・リソースの点でコストがかかります。そのようなコストを省くために、プログラマーは、よく使う機能をいくつかのダイナミック・リンク・ライブラリー (DLL) に分類します。これにより、DLL が提供するサービスがプログラムで必要になった場合、その DLL をロードして、そのサービスを提供する機能を呼び出すだけで済みます。

マルチスレッド・プログラムのために OS/400 のコール / リターン・プログラム・モデルがサポートされてはいますが、可能な限り、呼び出し元の活動化グループで実行されるサービス・プログラムまたは DLL を使うようにしてください。マルチスレッド・プログラムを他のプラットフォームから移植する場合、移植時にそのサービス・プログラムも使うことになるからです。

必須ではありませんが、マルチスレッド・アプリケーションで使用するすべてのプログラムは、統合言語環境 (ILE) プログラムにしてください。 ILE 以外のプログラムをマルチスレッド・プログラムで使う場合は、特別な考慮が必要になります。詳細は、 69 ページの『OPM 言語でのスレッドの考慮事項』 に説明されています。マルチスレッド・プログラムが既存のプログラムを呼び出す場合は、それらの既存のプログラムのスレッド・セーフティを評価する必要があります。

ジョブとジョブ・リソース

ジョブは、記憶域などのリソースのコンテナーとなるものです。ジョブは、単独では実行できません。

すべてのジョブには、2 種類の基本的な記憶域が関連しています。

データ:

データは、すべてのプログラム変数が保管される場所です。これは、グローバル変数と静的変数のための記憶域 (静的)、動的割り振り記憶域のための記憶域 (ヒープ)、および関数のローカル変数のための記憶域 (自動) に分かれています。プログラム変数のための記憶域は、プログラムが活動化された活動化グループから割り振られます。静的ストレージとヒープ記憶域には、その活動化グループで実行されている、すべてのスレッドからアクセスできます。自動記憶域とローカル・プログラム変数は、他のスレッドからはアクセスできません。

スタック:

スタックには、スレッド内のプログラムまたはプロシージャーの呼び出しフローについてのデータが入れられます。スタックは、自動記憶域と共に、各スレッドの作成時に割り振られます。 スタックと自動記憶域がスレッドによって使われる場合、それらはスレッド・リソースと見なされます。スレッドが終了した時点で、それらのリソースはプロセスに戻され、それ以降は別のスレッドによって使用可能になります。

ジョブ・リソースとは、ジョブ内のすべてのスレッドからアクセスできるリソースのことです。ジョブ・リソースには、以下のものが含まれます。

- コード化文字セット ID (CCSIDS)
- ロケール
- 環境変数
- ファイル記述子
- ジョブのスコープでオープンされるファイル
- シグナル・アクション・ベクトル
- シグナル・タイマー
- 現行作業ディレクトリー

活動化グループ内で実行されるすべてのスレッドは、活動化グループのリソース (静的およびヒープ記憶域など) を共有できます。あるスレッドがジョブ・リソースを変更すると、新しい値はただちにすべてのスレッドから可視になります。たとえば、1 つのスレッドでシグナル処理アクションを変更すれば、全スレッドに渡ってシグナル処理アクションを効率よく変更することができます。

スレッド・プライベート・データとスレッド固有データ

特定のリソースは、複数のスレッドで共用できません。スレッド間で共用できないデータは、スレッド・プライベート・データと呼ばれます。このスレッド・プライベート・データは、スレッドの使い方によって定義されます。 OS/400 では、以下のリソースがスレッド・プライベート・データとして定義されます。

スレッド ID:

スレッドを識別するために使う固有の整数値。

優先順位:

OS/400 では、ジョブ内のあるスレッドについて、他のスレッドとの関係に関して相対的な重要性を決定するスレッド優先順位を指定できます。スレッド優先順位は、ジョブの優先順位に対する差分値となるように定義されます。スレッドの優先順位はジョブの優先順位に加算されます。それがジョブの優先順位を超えることがあってはなりません。ジョブの優先順位が調整される場合、スレッドの優先順位はその新しいジョブの優先順位に対する相対値に調整されます。スレッド優先順位のデフォルト値は 0、 つまりジョブと同じ優先順位です。

セキュリティー情報

セキュリティー情報は、ユーザー・プロファイルおよびグループ・プロファイルも含め、スレッドごとに保持されます。スレッドが新しいスレッドを作成すると、新しいスレッドはそれを作成したスレッドからセキュリティー情報を継承します。

ライブラリー・リスト:

ライブラリー・リスト情報は、ユーザー・プロファイルおよびグループ・プロファイルも含め、スレッドごとに保持されます。スレッドが新しいスレッドを作成すると、新しいスレッドはそれを作成したスレッドからライブラリー・リスト情報を継承します。

シグナル・ブロッキング・マスク:

シグナル・ブロッキング・マスクは、スレッドに配布されないようにする一連の非同期シグナルを識別するものです。スレッドが新しいスレッドを作成すると、新しいスレッドはそれを作成したスレッドのシグナル・ブロッキング・マスクを継承します。

呼び出しスタック:

呼び出しスタックには、スレッド内のプログラム・フローまたはプロシージャーの呼び出しフローについてのデータが入れられます。スタックは、自動記憶域と共に、各スレッドの作成時に割り振られます。

自動記憶域:

自動記憶域は、関数のローカル変数のためのものです。

errno 変数:

C または POSIX システム呼び出しの結果を戻すために使われるプログラム変数。 errno は、スレッド内の関数呼び出しの最後の結果を戻す関数呼び出しです。

スレッドでは、スレッド固有データと呼ばれる、データ項目の独自のビューを持つことが可能です。スレッド固有データはスレッド・プライベート・データとは異なっています。スレッド・プライベート・データは、スレッドのインプリメンテーションによって定義されますが、スレッド固有データはアプリケーションによって定義されます。スレッドはスレッド固有記憶域を共用しませんが (1 つのスレッドに固有)、そのスレッド内のすべての関数からアクセスできます。通常は、キーによってスレッド固有記憶域に索引が付けられます。キーとは、すべてのスレッドから見えるグローバル値のことです。キーは、そのキーに関連付けられている記憶域のスレッド固有値を取り出すために使われます。

マルチスレッド・プログラム起動コマンド

| マルチスレッド・プログラムを呼び出すためには、プログラムを呼び出すジョブが、マルチスレッドをサポートしていなければなりません (マルチスレッド可能)。 OS/400 カーネル・スレッド・サポートでは、サポートされるジョブ・タイプのうち一部分でしかスレッドを作成できません。対話式および通信ジョブは、マルチスレッド対応サポートを提供していません。

| ジョブでマルチスレッドをサポートできるかどうかは、ジョブ記述の作成 (CRTJOBD) コマンドとジョブ記述の変更 (CHGJOBD) コマンドのマルチスレッド許可 (ALWMLTTHD) パラメーターによって制御されます。 OS/400 では、通信ジョブと対話式ジョブを除くすべてのタイプのジョブに関して、 ALWMLTTHD パラメーターが調べられます。事前開始ジョブ項目の追加 (ADDPJE) および事前開始ジョブ項目の変更 (CHGPJE) の 2 つのサブシステム記述ジョブ入力コマンドでは、その項目のジョブ記述に含まれている ALWMLTTHD 設定値を使うことによって、その項目によって開始されるジョブでマルチスレッドがサポートされるかどうかが制御されます。

spawn() API のプロセス生成 (Spawn Process) では、継承構造体の中で、子プロセスがマルチスレッドをサポートするように作成されるかどうかを制御する新しいフラグ・フィールドがサポートされています。そのフラグ・フィールド SPAWN_SETTHREAD_NP は、継承構造体に対する非標準の OS/400 プラットフォーム固有の拡張機能です。 spawn() API は、マルチスレッドをサポートするバッチ即時ジョブ、または事前開始ジョブを開始できる唯一のプログラミング手法です。 CALL コマンドとよく似た SPAWN コマンドを使うことによって、マルチスレッド・プログラムを簡単に呼び出すことができます。 OS/400 オプション 7、OS/400 例題ツール・ライブラリー、QUSRTOOLS の一部としてサンプルの SPAWN コマンドが用意されており、それを使用したり修正したりすることができます。

活動化グループとスレッド

| すべてのプログラムおよびサービス・プログラムは、活動化グループと呼ばれるジョブのサブ構造の中で活動化されます。このサブ構造には、静的ストレージとヒープ記憶域、および一時データ管理リソースを含め、プログラムを実行するために必要なリソースが含まれています。

マルチスレッド可能ジョブでは、2 つ以上のスレッドで活動化グループを共用できます。 1 つのスレッドで、異なる活動化グループが作成したいろいろなプログラムを実行できます。システムには、活動化グループ内で実行されるスレッドのリストもスレッド内の活動化グループのリストも保持されていません。活動化グループ中の他のスレッドが活動状態であるにもかかわらず、その中の 1 つのスレッドを終了すると、他のスレッドが破壊されたり、予期できない結果となったり、プロセスが異常終了したりする可能性があります。このような問題を避けるため、マルチスレッド可能ジョブ内で活動化グループを終了させるアクションを実行すると、ジョブが通常の方法で終了するようになっています。つまり、ジョブ内のすべてのスレッドを終了し、初期スレッドから EXIT (出口) ルーチンおよび C++ デストラクターを呼び出し、最後にファイルをクローズします。

活動化グループの処理では、以下のことを考慮してください。

活動化グループからの戻り:

ACTGRP(*NEW) で作成されたプログラムは、呼び出されるたびに、新しい活動化グループを起動します。活動化グループが活動化グループから戻ると、それは終了します。マルチスレッド可能ジョブでは、 ACTGRP(*NEW) で作成されたプログラムから戻るとジョブが終了します。

デフォルト活動化グループと名前付き活動化グループは、両方とも永続的です。デフォルト活動化グループは、ジョブ開始時に作成され、ジョブ終了時までは削除されません。名前付き活動化グループは、ジョブ内で最初に必要になった時点で作成されます。名前付き活動化グループから正常に戻った場合、活動化グループの状態は最後に使用された時点での状態になり、活動化グループは削除されません。したがって、マルチスレッド可能ジョブにおいて、デフォルト活動化グループまたは名前付き活動化グループから正常に戻った場合、ジョブは終了しません。

マルチスレッド可能ジョブの活動化グループで exit() または abort() を使った場合、そのジョブは必ず終了します。

未処理例外:

管理境界に回復機能委任されても例外が未処理のまま残り、管理境界がプログラム・エントリー・プロシージャー (PEP またはメインエントリー・ポイント) である場合、マルチスレッド可能ジョブは終了します。

活動化グループの再利用 RCLACTGRP:

マルチスレッド操作可能ジョブでは、このコマンドはジョブの初期スレッドでのみ呼び出すことができます。ジョブに 2 次スレッドが存在する場合、RCLACTGRP コマンドが実行される前に終了します。

関連情報は、 35 ページの『第 5 章 スレッド・セーフティ』に記載されています。

マルチスレッド・プログラミングでの通信についての考慮事項

OS/400 でサポートされている、唯一のスレッド・セーフティ通信プロトコルは、ソケットです。ソケットを使う場合、以下の制限事項があります。

ソケットのアプリケーション・プログラミング・インターフェース (API):

ソケットのインターフェースの多くはスレッド・セーフティですが、ネットワーク・ルーチンのほとんどは静的ストレージを使用するようにはなっていません。それらのルーチンは、スレッド・セーフティの ″_r″ のもので置き換えられています。たとえば、**gethostbyaddr()** の呼び出しは、**gethostbyaddr_r()** に置き換えてください。 ″_r″ ルーチンは、UNIX 定義と互換性があります。すべての ″_r″ 関数は、既存のサービス・プログラム QSOSRV2 の中にあります。

ソケットを使う AnyNet:

ソケットを使う AnyNet は、スレッド・セーフティと見なされ、マルチスレッド・プログラムでサポートされています。しかし、このサポートは公式のテストを受けていません。

マルチスレッド・プログラミングでのデータベースについての考慮事項

マルチスレッド・プログラミングでデータベースを使う場合には、いくつかの点を考慮する必要があります。

データ定義言語 (DDL):

データベースの構成、管理、およびセットアップ・タイプのインターフェースの多くはスレッド・セーフティです。スレッド・セーフティであるデータベース操作には、ファイルの作成 (Create File)、メンバーの追加 (Add Member)、ファイルの削除 (Delete File)、メンバーの除去 (Remove Member) が含まれます。コマンドがスレッド・セーフティであるかどうかを判断するには、 CL 解説書を参照するか、またはコマンドの表示 (DSPCMD) コマンドを使用してください。そのコマンドのオンライン・ヘルプ情報には、コマンドに適用されるスレッド・セーフティの必要条件がリストされています。

データベース・レコード入出力:

データベースは、操作 (読み取り、更新、挿入、または削除) の間、入出力操作を保護します。ファイルのオープン・インスタンスをスレッド間で共用する場合、入出力フィードバック域と入出力バッファーへのアクセスを逐次化することによって、それらの領域に含まれる有効な情報が参照されるようにする必要があります。それらの領域はアプリケーション制御下にあり、データベース操作完了後にデータベースによって保護することはできません。

例として、簡単な読み取り操作があります。スレッド 1 が読み取り処理中で、スレッド 2 がそれと同じオープン・インスタンスに対して入出力操作を実行する場合、スレッド 2 はスレッド 1 が読み取りを完了するまで待機します。スレッド 1 の読み取りの結果は、入出力バッファーに入れられます。制御がスレッド 1 に戻ると、スレッド 2 はその入出力操作を開始します。逐次化が行われない場合、スレッド 1 で結果が得られる前に、スレッド 2 が入出力バッファーの情報を変更してしまう可能性があります。

ファイルのオープン・インスタンスをスレッド間で共用しない場合、逐次化は必要ありません。

詳細は、 66 ページの『データベース・レコードの入出力とスレッド・セーフティ』を参照してください。

分散ファイル:

タイプ *SNA の分散データベース・ファイル (LCP) および分散データ管理 (DDM) ファイルへのアクセスは、スレッド・セーフティではありません。このタイプのデータベース・ファイルへのアクセスは、マルチスレッド可能ジョブでは拒否されます。 ICF ファイルはスレッド・セーフティにはできないため、それらのファイルと SNA 層全体はスレッド・セーフティではありません。それらのファイル・タイプの 1 つに対してオープンが試行されると、そのファイルをオープンしようとした関数に対して、CPF4380 メッセージ (マルチスレッド・プロセスで無効なオープン属性) が送られます。

トリガー・プログラム:

マルチスレッド・ジョブでは、トリガー・プログラムを呼び出すことができます。トリガー・プログラムに適用されるスレッド・セーフティに関する制限は、マルチスレッド・ジョブで実行されるその他のすべてのコードに適用されるものと同じです。物理ファイル・トリガーの追加

(ADDPFTRG) コマンド上のパラメーターにより、トリガーのスレッド・セーフティ状況、およびトリガーがマルチスレッド・ジョブ内で呼び出された場合にとるべき処置を指定することができます。

様式選択プログラム:

複数の様式を持つ論理ファイルの場合、様式選択プログラムの使用は、スレッド・セーフティではありません。マルチスレッド・ジョブでは、様式選択プログラムを使用しないでください。

ストアード・プロシージャー:

DB2 SQL ストアード・プロシージャー・サポートにより、SQL アプリケーションは SQL ステートメントを介して外部プログラムを定義して呼び出すことができます。ストアード・プロシージャーは、マルチスレッド・ジョブの中で呼び出すことができます。ストアード・プロシージャーに適用されるスレッド・セーフティに関する制限は、マルチスレッド・ジョブで実行されるその他のすべてのコードに適用されるものと同じです。トリガー・プログラムと違って、ストアード・プロシージャーのスレッド・セーフティ状況、およびストアード・プロシージャーがマルチスレッド・ジョブで呼び出された場合に取るべき処置を指定する方法はありません。

構造化照会言語 (SQL) ステートメント:

データ定義言語 (DDL) SQL ステートメントの使用は、スレッド・セーフティではありません。データ操作言語 (DML) ステートメントはスレッド・セーフティです。 SQL ステートメントのスレッド・セーフティの詳細については、 SQL 解説書の『スレッド』の項を参照してください。

SQL を使用してデータベースと対話する方法の例が、以下に記載されています。
12 ページの『例: ローカル SQL データベースの処理 (マルチスレッド Pthread プログラム)』

SQL 用のサーバー・モード:

マルチスレッド・アプリケーションを使って、データベースにアクセスするための望ましい方法は、 SQL 用のサーバー・モードを使用することです。ジョブは、複数のデータベース接続およびトランザクションを管理するために、 SQL 用のサーバー・モードを使用することができます。アプリケーションが SQL 用のサーバー・モードを使用する時、OS/400 はジョブ内で接続を使用します。これは、以前の OS/400 に増してカプセル化した形式で、現行のデータベース・コンテキストを表現するものです。これにより、複数のユーザーによるデータベースへの接続、同一のまたは異なるユーザーによるデータベースへの複数の接続、およびデータベースへの接続による複数の独立したトランザクションの存在が可能になります。

アプリケーションがデータにアクセスする前に、以下のメカニズムのいずれかを使用して、SQL 用のサーバー・モードを活動化してください。

- データ・アクセスが行われる前に、ODBC API の SQLSetEnvAttr() を使用し、SQL_ATTR_SERVER_MODE 属性を SQL_TRUE に設定する。
- データ・アクセスが行われる前に、ジョブの変更 API の QWTCHGJB() を使用し、 'Server mode for Structured Query Language' キーを設定する。
- Java を使用して、JDBC 経由でデータベースにアクセスする。 JDBC は、JDBC の必須のセマンティクスを保護するために自動的にサーバー・モードを使用します。

SQL 用のサーバー・モードは、以下のように動作します。

1. 組み込み SQL の場合、ジョブ内の各スレッドは、そのスレッド内に複数の接続がある場合でさえ、コミットまたはロールバックできる別々のトランザクションです。
2. ODBC、CLI、および JDBC の場合、それぞれの結合ハンドルは、データベースへの独立した接続を表し、別々のエンティティーとしてコミットおよび使用できます。

SQL 用の呼び出しレベル・インターフェース (CLI) は、スレッド・セーフティです。 CLI に関する詳細は、SQL 呼び出しレベル・インターフェース (ODBC) に記載されています。

SQL 用のサーバー・モードに関する詳細は、 SQL 解説書の『スレッド』の項に記載されています。

コミット可能トランザクション:

スレッドを導入しても、コミット可能トランザクションの有効範囲は変わりません。コミット可能作業単位の有効範囲は、ジョブ・レベルのコミットメント定義または活動化グループ・レベルのコミットメント定義のいずれかにすることができます。スレッド・コミット操作またはロールバック操作は、コミット定義の下で実行されるすべての操作をコミットまたはロールバックします。各スレッド (または複数のスレッドでなるグループ) ごとに別個のコミット可能トランザクションが必要なアプリケーションでは、別個の活動化グループを使うことによって、それらのトランザクションを管理しなければなりません。

例: ローカル SQL データベースの処理 (マルチスレッド Pthread プログラム)

```
/*************************************************************/
/* Testcase: SQLEXAMPLE                                      */
/*                                                           */
/* Function:                                                 */
/* Demonstrate how Embedded SQL can be used within a         */
/* threaded ILE C program.  This program creates and         */
/* populates an SQL Table with data.  Then, threads are      */
/* created which each open a cursor and read all the         */
/* data from the table.  A semaphore is used to show         */
/* how threads execution can be controlled.                  */
/*                                                           */
/* To compile program:                                       */
/* CRTSQLCI OBJ(QGPL/SQLEXAMPLE) SRCFILE(QGPL/QCSRC)         */
/* COMMIT(*NONE) RDB(*NONE) OBJTYPE(*MODULE)                 */
/* OUTPUT(*PRINT) DBGVIEW(*SOURCE)                           */
/*                                                           */
/* To bind program:                                          */
/* CRTPGM PGM(QGPL/SQLEXAMPLE)                               */
/* MODULE(QGPL/SQLEXAMPLE) ACTGRP(*CALLER)                   */
/*                                                           */
/* To invoke program:                                        */
/* SPAWN QGPL/SQLEXAMPLE                                     */
/*                                                           */
/*************************************************************/
#define _MULTI_THREADED
#include <pthread.h>
#include <sys/sem.h>
#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

static int semid;
static struct sembuf op_try1[1] = {0,0,0};
#define MAXTHREADS 2

void *threadrtn(void *parm);

int main(int argc, char **argv)

{
  int             rc;
  int             *status;
```

```
  pthread_t       thids[MAXTHREADS];
  EXEC SQL BEGIN DECLARE SECTION;
  int             i, j;
  char            insert[200];
  EXEC SQL END   DECLARE SECTION;

  EXEC SQL  INCLUDE SQLCA;
  EXEC SQL  INCLUDE SQLDA;

  /* create a new semaphore */
  semid=semget(IPC_PRIVATE,1,S_IRUSR|S_IWUSR);

  printf("¥nsemaphore created¥n");
  rc=semctl(semid, 0, SETVAL, 1);

  printf("semaphore inited¥n");
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  EXEC SQL CREATE TABLE QGPL/SQLEXMP (COL1 INT,COL2 INT);

  printf("SQL table created¥n");
  EXEC SQL WHENEVER SQLERROR GO TO :mainerror;

  for (i=1,j=100;i<10;i++,j++) {
     (void) sprintf(insert, "INSERT INTO QGPL/SQLEXMP ¥
                   VALUES(%d, %d)", i, j);
     EXEC SQL EXECUTE IMMEDIATE :insert;
  }

  printf("Table primed with data¥n");
  for (i=0;i<MAXTHREADS;i++)  {
     pthread_create(&thids[i], NULL, threadrtn, NULL);
  }
  printf("Threads created¥n");

  rc=semctl(semid, 0, SETVAL, 0);

  printf("Threads turned loose¥n");
  for (i=0;i<MAXTHREADS;i++)  {
     pthread_join(thids[i], &status);
  }

  printf("Threads joined¥n");
  return;

mainerror:
  printf("ERROR: sqlcode = %d sqlstate = %d¥n", SQLCODE, SQLSTATE);
}

/*********************************************************/
/* This thread will do the following:                    */
/*   - Declare a cursor for the example table            */
/*   - Block on a semaphore until initial thread         */
/*     is ready for us to run                            */
/*   - Open the cursor                                   */
/*   - Fetch data one row at a time in a loop until      */
/*     End of File is reached or an error occurs         */
/*   - Close the cursor and return                       */
/*********************************************************/
void *threadrtn(void *parm)
{
 EXEC SQL  INCLUDE SQLCA;
 EXEC SQL  INCLUDE SQLDA;

 EXEC SQL BEGIN DECLARE SECTION;
 long HV1, HV2;
 EXEC SQL END   DECLARE SECTION;
```

```
 EXEC SQL WHENEVER SQLERROR GO TO :thderror;
 EXEC SQL WHENEVER NOT FOUND GO TO :thdeof;

 EXEC SQL DECLARE C1 CURSOR FOR SELECT * FROM QGPL/SQLEXMP;

 /* block on semaphore */
 semop(semid,&op_try1[0],1);

 EXEC SQL OPEN C1;
 printf("thid:%.8x %.8x: cursor open¥n",pthread_getthreadid_np());

 /* Loop until End of File (EOF) */
 for (;;) {
    EXEC SQL FETCH C1 INTO :HV1, :HV2;
    printf("thid:%.8x %.8x: fetch done... COL1=%d COL2=%d¥n",
            pthread_getthreadid_np(), HV1, HV2);
 }

thderror:
 printf("thid:%.8x %.8x: sqlcode = %d sqlstate = %d¥n",
        pthread_getthreadid_np(), SQLCODE, SQLSTATE);
 EXEC SQL CLOSE C1;
 return;

thdeof:
 printf("thid:%.8x %.8x: Done!¥n",
        pthread_getthreadid_np());
 return;
}



Testcase output:
 semaphore created
 semaphore inited
 SQL table created
 Table primed with data
 Threads created
 Threads turned loose
 thid:00000000 00000022: cursor open
 thid:00000000 00000023: cursor open
 thid:00000000 00000023: fetch done... COL1=1 COL2=100
 thid:00000000 00000022: fetch done... COL1=1 COL2=100
 thid:00000000 00000023: fetch done... COL1=2 COL2=101
 thid:00000000 00000022: fetch done... COL1=2 COL2=101
 thid:00000000 00000023: fetch done... COL1=3 COL2=102
 thid:00000000 00000022: fetch done... COL1=3 COL2=102
 thid:00000000 00000023: fetch done... COL1=4 COL2=103
 thid:00000000 00000022: fetch done... COL1=4 COL2=103
 thid:00000000 00000023: fetch done... COL1=5 COL2=104
 thid:00000000 00000022: fetch done... COL1=5 COL2=104
 thid:00000000 00000023: fetch done... COL1=6 COL2=105
 thid:00000000 00000022: fetch done... COL1=6 COL2=105
 thid:00000000 00000023: fetch done... COL1=7 COL2=106
 thid:00000000 00000022: fetch done... COL1=7 COL2=106
 thid:00000000 00000023: fetch done... COL1=8 COL2=107
 thid:00000000 00000022: fetch done... COL1=8 COL2=107
 thid:00000000 00000023: fetch done... COL1=9 COL2=108
 thid:00000000 00000022: fetch done... COL1=9 COL2=108
 thid:00000000 00000023: Done!
 thid:00000000 00000022: Done!
 Threads joined
```

マルチスレッド・プログラミングでのデータ管理機能についての考慮事項

マルチスレッド・プログラムでデータを管理するには、いくつかの点を考慮する必要があります。

ファイルのオープン操作:

マルチスレッド可能ジョブでオープンできるのは、統合ファイル・システム・ストリーム・ファイル、印刷ファイル、およびタイプ *IP の分散データ管理 (DDM) ファイル、およびローカル・データベース・ファイルだけです。印刷ファイル、タイプ *IP DDM ファイル、またはローカル・データベース・ファイル以外の *FILE オブジェクトをオープンしようとすると、ファイルをオープンしようとする関数に対して CPF4380 エスケープ・メッセージが送られます。この CPF4380 エスケープ・メッセージは、マルチスレッド・プロセスのオープン属性が無効であることを知らせるものです。ローカル・データベース・ファイルには、物理ファイルと論理ファイルが含まれます。疎結合並列 (LCP) ファイルはローカルでなく、関数がファイルをオープンしようとすると、CPF4380 エスケープ・メッセージを送ります。ローカル・データベース・ファイルには、装置ファイルやその他の通信ファイルである保管ファイル (*SAVF) も含まれません。 SPOOL(*NO) を指定して印刷ファイルをオープンすることもできません。印刷ファイルのオープンで SPOOL(*NO) を指定すると、印刷ファイルをオープンしようとした関数に、CPF4380 エスケープ・メッセージが送られます。

マルチスレッド可能ジョブでは、共用オープンが可能です。しかし、オープン・データ・パス (ODP) をスレッド間で共用することはできません。ファイルが SHARE(*YES) でオープンされており、OPNSCOPE(*ACTGRPDFN) が指定されている場合、同じ活動化グループで実行中の同じスレッドにある後続の共用オープンは ODP を共用できます。ファイルが SHARE(*YES) でオープンされており、OPNSCOPE(*JOB) が指定されている場合、同じスレッドにあるファイルの後続の共用オープンは ODP を共用できます。

ファイルの一時変更:

マルチスレッド・アプリケーションは、初期スレッドでしか一時変更を発行できません。 2 次スレッドで一時変更を発行しようとすると、CPF180C エスケープ・メッセージが戻されます。ジョブ・レベルおよび活動化グループ・レベルの有効範囲の一時変更だけが 2 次スレッドに影響を与えます。呼び出しレベルの有効範囲の一時変更は不可視です。すべての有効範囲レベルの一時変更は、初期スレッドに影響を与えます。 使用できるファイル一時変更コマンドは、データベース・ファイルによる一時変更 (OVRDBF)、プリンター・ファイルによる一時変更 (OVRPRTF)、およびメッセージ・ファイルによる一時変更 (OVRMSGF) の 3 種類です。

一時変更の削除 (DLTOVR) コマンドを呼び出すことによって、初期スレッドにおける一時変更を削除することができますが、 2 次スレッドにおける一時変更を削除することはできません。 2 次スレッドにおける一時変更を削除すると、DLTOVR は、CPF180C エスケープ・メッセージを戻します。

リソース再利用コマンド:

OS/400 はリソースをトラッキングしないため、リソース再利用コマンドは、いずれもスレッド・セーフティではありません。オペレーティング・システムでは、どのリソースがどのスレッドによって使用中であるかは識別できません。 2 次スレッドで、リソースの再利用 (RCLRSC) コマンドと、活動化グループの再利用 (RCLACTGRP) コマンドを呼び出すことはできません。それらを呼び出すと、それらのコマンドからコマンドの呼び出し元に対して、CPF1892 エスケープ・メッセージが送られます。

マルチスレッド・プログラミングでのファイル・システムについての考慮事項

マルチスレッド・プログラミングでファイル・システムの処理を実行する場合には、いくつかの点を考慮する必要があります。

スレッド・セーフティ・ファイル・システム:

Root、QOpenSys、ユーザー定義 (UDFS)、QNTC、QSYS.LIB、QOPT、および QLANSrv の各ファイル・システムは、スレッド・セーフティであり、何も制限はありません。 QDLS、NFS、QFileSvr.400、および QNetWare ファイル・システムは、スレッド・セーフティではありません。スレッド・セーフティではないファイル・システムの、オブジェクトを表すファイルまたはファイル記述子をオープンしようとしたり、それらに対して統合ファイル・システム API またはコマンドを使おうとしたりすると、エラー ENOTSAFE で終了します。このエラーは、1 つのジョブ内に複数のスレッドがある場合にのみ発生します。

現行作業ディレクトリーを継承している場合、またはスレッド・セーフティではないファイル・システムのファイルを表すファイル記述子をオープンする場合に、マルチスレッドのジョブからプログラムを生成しようとすると、それはエラーとなり、 ENOTSAFE が出されます。

統合ファイル・システム API:

スレッド・セーフティ・ファイル・システムに存在するオブジェクトを対象とする場合、統合ファイル・システム API はすべてスレッド・セーフティです。 オブジェクトが存在している場所のパスがわからない場合は、パスを照会して、統合ファイル・システムのスレッド・セーフティ・インターフェースから安全にアクセスできるかどうかを確認できます。 pathconf()、fpathconf()、statvfs()、および fstatvfs() の各 API を使えば、あるパスまたはファイル記述子がいずれかのスレッド・セーフティ・ファイル・システムのオブジェクトを参照しているかどうかを判別できます。

マルチスレッド・プログラミングでの印刷ファイルについての考慮事項

マルチスレッド可能ジョブで使えるのは、SPOOL(*YES) でオープンされた印刷ファイルだけです。印刷ファイルをスプールしないでオープンすると、エラーとなり、オープンを要求した関数に CPF4380 (マルチスレッド・プロセスで無効なオープン属性) エスケープ・メッセージが送信されます。

ほとんどの SNA 文字ストリング (SCS)、高機能プリンター・データ・ストリーム (IPDS)、拡張印刷データ・ストリーム (AFPDS)、LINE、および USERASCII 書き込み操作は、スレッド・セーフティです。スレッド・セーフティでない書き込み操作には、SCS グラフィックスを使用するもの、システム /36 環境で使用するもの、ユーザー・インターフェース管理機能 (UIM) によって生成されたもの、およびシステム /36 PRPQ サブルーチンを使用するものがあります。ページと行のカウントは、カウンターを自動的に更新することができないので、不正確になる可能性があります。

印刷ファイルの一時変更については、 15 ページの『マルチスレッド・プログラミングでのデータ管理機能についての考慮事項』を参照してください。

第 4 章 OS/400 スレッドの管理

ここで説明する概念は、すべてのプログラミング言語に関係しています。それぞれの言語でこの概念がどのように実現されているかについては、各言語のプログラマーの手引きを参照してください。 Pthread API に関する特定の情報は、『OS/400 Pthread APIs』に記載されています。

以下のトピックがあります。

- 『スレッドの属性』
- 20 ページの『スレッドを開始する』
- 22 ページの『スレッドを終了する』
- 25 ページの『スレッドを取り消す』
- 28 ページの『スレッドを一時停止する』
- 30 ページの『スレッドを再開する』
- 30 ページの『スレッドの終了を待機する』
- 33 ページの『処理装置を別のスレッドに譲る』

スレッドの属性

スレッドの属性とは、スレッドの振る舞いに影響を与えるスレッドの特性のことです。どのようなプログラミング言語、およびアプリケーション・プログラミング・インターフェース (API) を使うかに応じて、さまざまな属性を使用できます。各属性の使い方やそれがスレッドに与える効果は、アプリケーションにとってスレッドの属性が、プログラミング言語および API のセットによって、どのように実現されているかに応じて異なります。スレッドの属性は、スレッド開始時に設定できますし、スレッドを活動状態で実行した後で変更することもできます。

下記に、よく使われるスレッドの属性とその影響について示します。

優先順位:

これは、別のスレッドまたはプロセスによって割り込まれるまで、システムが特定のスレッドに割り当てる処理時間の量に影響を与えます。

スタック・サイズ:

スタック・スペースが不足しているためにスレッドがエラーになるまで、スレッドが呼び出すことができる関数の数に影響を与えます。

名前:

アプリケーションによるスレッドのデバッグ、またはスレッドのアクションのトレースを実行する機能に影響を与えます。

スレッド・グループ:

一度に複数のスレッドを容易に管理する機能に影響を与えます。

切り離し状態:

スレッド終了時に、そのスレッドに関連付けられたリソースを再利用したり活動状態のままにしたりする方法に影響を与えます。

スケジューリング・ポリシー:

システム内またはアプリケーション内での、スレッドのスケジューリングの方法に影響を与えます。これはスレッドの優先順位と関係があります。

継承スケジューリング:

システムによってスレッドの優先順位が決められる方法に影響を与えます。

プログラム・サンプルとしては、以下の資料を参考にしてください。

- 『例: スレッドの属性を設定する (Pthread プログラム)』
- 19 ページの『例: スレッドの属性を設定する (Java)』

例: スレッドの属性を設定する (Pthread プログラム)

下記の例に、『切り離し状態』スレッド属性を Pthread プログラムで設定する方法を示します。

```
/*
Filename: ATEST10.QCSRC
The output of this example is as follows:
 Enter Testcase - LIBRARY/ATEST10
 Create a default thread attributes object
 Set the detach state thread attribute
 Create a thread using the new attributes
 Destroy thread attributes object
 Join now fails because the detach state attribute was changed
 Entered the thread
 Main completed
*/
#define  _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define checkResults(string, val) {             ¥
 if (val) {                                     ¥
   printf("Failed with %d at %s", val, string); ¥
   exit(1);                                     ¥
 }                                              ¥
}

void *theThread(void *parm)
{
   printf("Entered the thread¥n");
   return NULL;
}

int main(int argc, char **argv)
{
  pthread_attr_t        attr;
  pthread_t             thread;
  int                   rc=0;

  printf("Enter Testcase - %s¥n", argv[0]);

  printf("Create a default thread attributes object¥n");
  rc = pthread_attr_init(&attr);
  checkResults("pthread_attr_init()¥n", rc);

  printf("Set the detach state thread attribute¥n");
  rc = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
  checkResults("pthread_attr_setdetachstate()¥n", rc);

  printf("Create a thread using the new attributes¥n");
  rc = pthread_create(&thread, &attr, theThread, NULL);
  checkResults("pthread_create()¥n", rc);
```

```
  printf("Destroy thread attributes object¥n");
  rc = pthread_attr_destroy(&attr);
  checkResults("pthread_attr_destroy()¥n", rc);

  printf("Join now fails because the detach state attribute was changed¥n");
  rc = pthread_join(thread, NULL);
  if (rc==0) {
     printf("Unexpected results from pthread_join()¥n");
     exit(1);
  }
  sleep(2);

  printf("Main completed¥n");
  return 0;
}
```

スレッド属性については、 17 ページの『スレッドの属性』を参照してください。

例: スレッドの属性を設定する (Java)

下記の例に、『名前』スレッド属性を Java プログラムで設定する方法を示します。

```
/*
FileName: ATEST10.java

The output of this example is as follows:
Entered the testcase
Create a thread
Set some thread attributes
Start the thread
Wait for the thread to complete
Entered the thread: "theThread"
Testcase completed
*/

import java.lang.*;

public class ATEST10 {

   static class theThread extends Thread {
      public void run() {
         System.out.print("Entered the thread: ¥"" + getName() +
                          "¥"¥n");
      }
   }

   public static void main(String argv[]) {
      System.out.print("Entered the testcase¥n");

      System.out.print("Create a thread¥n");
      theThread t = new theThread();

      System.out.print("Set some thread attributes¥n");
      t.setName("theThread");

      System.out.print("Start the thread¥n");
      t.start();

      System.out.print("Wait for the thread to complete¥n");
      try {
         t.join();
      }
      catch (InterruptedException e) {
         System.out.print("Join interrupted¥n");
```

```
      }

      System.out.print("Testcase completed¥nj");
      System.exit(0);
   }

}
```

スレッド属性については、 17 ページの『スレッドの属性』を参照してください。

スレッドを開始する

アプリケーションで新しいスレッドを作成すると、スレッド・オブジェクト、制御構造、および実行時サポートがシステムによって初期設定されます。それらによって、新しいスレッドは言語構成要素およびシステム・サービスを安全に使用できるようになります。さらに、新しいスレッドを開始する前に、その新しいスレッドで使うアプリケーション・データおよびパラメーターを初期設定することが必要な場合もあります。

スレッドを作成するたびに、システムはそれに固有のスレッド識別コードを割り当てます。スレッド識別コードは、スレッドをデバッグしたり、トレースしたり、その他の管理活動を実行したりする場合に使用できる整数値です。このスレッド識別コードは、スレッドを直接操作するために使うことは普通しません。

スレッド・アプリケーション・プログラム・インターフェース (API) セットの多くは、新しく作成されたスレッドを表すスレッド・オブジェクトまたはハンドルも戻します。そのスレッド・オブジェクトを使うことによって、その新しいスレッドを操作したり、同期オブジェクトとしてスレッドが処理を終了するのを待機したりできます。

プログラム・サンプルとしては、以下の資料を参考にしてください。

『例: スレッドを開始する (Pthread プログラム)』

21 ページの『例: スレッドを開始する (Java)』

例: スレッドを開始する (Pthread プログラム)

下記の例に、Pthread プログラムでスレッドを開始し、それにパラメーターを渡す方法を示します。

```
/*
Filename: ATEST11.QCSRC
The output of this example is as follows:
 Enter Testcase - LIBRARY/ATEST11
 Create/start a thread with parameters
 Wait for the thread to complete
 Thread ID 0000000c, Parameters: 42 is the answer to "Life, the Universe and Everything"
 Main completed
*/
#define  _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define checkResults(string, val) {             ¥
 if (val) {                                     ¥
   printf("Failed with %d at %s", val, string); ¥
   exit(1);                                     ¥
 }                                              ¥
}
typedef struct {
      int               threadParm1;
   char     threadParm2[124];
```

```
} threadParm_t;

void *theThread(void *parm)
{
   pthread_id_np_t      tid;
   threadParm_t *p = (threadParm_t *)parm;
   tid = pthread_getthreadid_np();
   printf("Thread ID %.8x, Parameters: %d is the answer to ¥"%s¥"¥n",
          tid.intId.lo, p->threadParm1, p->threadParm2);
   return NULL;
}

int main(int argc, char **argv)
{
  pthread_t             thread;
  int                   rc=0;
  threadParm_t         *threadParm;

  printf("Enter Testcase - %s¥n", argv[0]);

  threadParm = (threadParm_t *)malloc(sizeof(threadParm));
  threadParm->threadParm1 = 42;
  strcpy(threadParm->threadParm2, "Life, the Universe and Everything");

  printf("Create/start a thread with parameters¥n");
  rc = pthread_create(&thread, NULL, theThread, threadParm);
  checkResults("pthread_create()¥n", rc);

  printf("Wait for the thread to complete¥n");
  rc = pthread_join(thread, NULL);
  checkResults("pthread_join()¥n", rc);

  printf("Main completed¥n");
  return 0;
}
```

スレッドの開始については、20 ページの『スレッドを開始する』を参照してください。

例: スレッドを開始する (Java)

このプログラムは、Java で作成されたプログラムでスレッドを開始する方法を示しています。

```
/*
FileName: ATEST11.java

The output of this example is as follows:
 Entered the testcase
 Create a thread with parameters
 Start the thread
 Wait for the thread to complete
 Thread Parameters: 42 is the answer to "Life, the Universe and Everything"
 Testcase completed
*/
import java.lang.*;


public class ATEST11 {

   static class theThread extends Thread {
      int               threadParm1;
      String            threadParm2;

      public theThread(int i, String s) {
```

```
        threadParm1 = i;
        threadParm2 = s;
      }
      public void run() {
        System.out.print("Thread Parameters: " + String.valueOf(threadParm1) +
                         " is the answer to ¥"" + threadParm2 + "¥"¥n");
      }
   }

   public static void main(String argv[]) {
      System.out.print("Entered the testcase¥n");

      System.out.print("Create a thread with parameters¥n");
      theThread t = new theThread(42, "Life, the Universe and Everything");

      System.out.print("Start the thread¥n");
      t.start();

      System.out.print("Wait for the thread to complete¥n");
      try {
        t.join();
      }
      catch (InterruptedException e) {
        System.out.print("Join interrupted¥n");
      }

      System.out.print("Testcase completed¥nj");
      System.exit(0);
   }

}
```

スレッドの開始については、20 ページの『スレッドを開始する』を参照してください。

スレッドを終了する

スレッドが処理を完了すると、終了のアクションを取り、システム・リソースを解放して他のスレッドが使用できるようにします。アプリケーション・プログラミング・インターフェース (API) のセットによっては、スレッド終了時にそのスレッドに関連付けられているリソースをアプリケーションが明示的に解放することが必要なものもあります。その他のスレッド・インプリメンテーション (Java など) では、該当する場合にリソースのガーベッジ・コレクションやクリーンアップを実行するものもあります。

スレッドの終了にはいくつかの方法があります。パフォーマンスの点でスレッドを終了する最善の方法は、スレッド開始時に呼び出された初期ルーチンから戻ることです。多くのスレッド API セットでは、初期ルーチンからの戻りがオプションでない場合にスレッドを終了させるためのメカニズムが提供されています。

一部の API セットでは、例外メカニズムもサポートされています。スレッド終了の例外メカニズムは、処理されない例外をスレッドが取った時点でスレッドが終了するようになっています。一例として、スレッドによって送出された Java 例外があります。

例外についての詳細、またはスレッドを終了するその他の方法については、各言語の資料を参照してください。

プログラム・サンプルとしては、以下の資料を参考にしてください。

23 ページの『例: スレッドを終了する (Pthread プログラム)』
24 ページの『例: スレッドを終了する (Java)』

例: スレッドを終了する (Pthread プログラム)

下記の例は、Pthread プログラムで、スレッドを終了する例です。

```
/*
Filename: ATEST12.QCSRC
The output of this example is as follows:
 Enter Testcase - LIBRARY/ATEST12
 Create/start a thread
 Wait for the thread to complete, and release its resources
 Thread: End with success
 Check the thread status
 Main completed
*/
#define  _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define checkResults(string, val) {             ¥
 if (val) {                                     ¥
   printf("Failed with %d at %s", val, string); ¥
   exit(1);                                     ¥
 }                                              ¥
}

const int THREADFAIL = 1;
const int THREADPASS = 0;

void *theThread(void *parm)
{
   printf("Thread: End with success¥n");
   pthread_exit(__VOID(THREADPASS));
   printf("Thread: Did not expect to get here!¥n");
   return __VOID(THREADFAIL);
}

int main(int argc, char **argv)
{
  pthread_t             thread;
  int                   rc=0;
  void                 *status;

  printf("Enter Testcase - %s¥n", argv[0]);

  printf("Create/start a thread¥n");
  rc = pthread_create(&thread, NULL, theThread, NULL);
  checkResults("pthread_create()¥n", rc);

  printf("Wait for the thread to complete, and release its resources¥n");
  rc = pthread_join(thread, &status);
  checkResults("pthread_join()¥n", rc);

  printf("Check the thread status¥n");
  if (__INT(status) != THREADPASS) {
     printf("The thread failed¥n");
  }

  printf("Main completed¥n");
  return 0;
}
```

スレッドの終了については、 22 ページの『スレッドを終了する』を参照してください。

例: スレッドを終了する (Java)

ここに示すのは、Java を使ってスレッドを終了する方法を示す例です。

```
/*
FileName: ATEST12.java

The output of this example is as follows:
 Entered the testcase
 Create a thread
 Start the thread
 Wait for the thread to complete
 Thread: End with success
 Check the thread status
 Testcase completed
*/
import java.lang.*;


public class ATEST12 {

   static class theThread extends Thread {
      public final static int THREADFAIL     = 1;
      public final static int THREADPASS     = 0;
      int         _status;

      public int status() {
         return _status;
      }
      public theThread() {
         _status = THREADFAIL;
      }
      public void run() {
         System.out.print("Thread: End with success¥n");
         _status = THREADPASS;
         /* End the thread without returning    */
         /* from its initial routine            */
         stop();
         System.out.print("Thread: Didn't expect to get here!¥n");
         _status = THREADFAIL;
      }
   }

   public static void main(String argv[]) {
      System.out.print("Entered the testcase¥n");

      System.out.print("Create a thread¥n");
      theThread t = new theThread();

      System.out.print("Start the thread¥n");
      t.start();

      System.out.print("Wait for the thread to complete¥n");
      try {
         t.join();
      }
      catch (InterruptedException e) {
         System.out.print("Join interrupted¥n");
      }
      System.out.print("Check the thread status¥n");
      if (t.status() != theThread.THREADPASS) {
         System.out.print("The thread failed¥n");
      }

      System.out.print("Testcase completed¥n");
```

```
        System.exit(0);
    }

}
```

スレッドの終了については、 22 ページの『スレッドを終了する』を参照してください。

スレッドを取り消す

スレッドを外部から終了する機能により、処理が長くかかる要求を実行するスレッドを、完了前に取り消すことができます。スレッドを取り消すために可能な方法は、使用しているスレッド・アプリケーション・プログラム・インターフェース (API) によって異なります。

一部の API セットでは、取り消しアクションのための well-defined points や、あるスレッドの終了時に別のスレッドを制御するための、その他のメカニズムが用意されています。一部の API セットでは、スレッドの終了前にクリーンアップ・コードを実行したり、取り消したスレッドの結果を特定の値に設定するメカニズムも用意されています。

スレッドの取り消し機能は十分注意して使用してください。使用する API セットで、well-defined cancelation points や、スレッドがアプリケーション・データおよびロックをクリーンアップするメカニズムが用意されていない場合、データが破壊されたり、アプリケーション内にデッドロックが発生する原因となる場合があります。

注: Java Development Kit バージョン 1.1 より後の Java のバージョンでは、スレッド・クラスに再開、停止、および一時停止メソッドを使用すべきではありません。それらのメソッドはあまり安全ではないからです。これらのメソッドにより提供される機能は、いくつかの変数の状態の検査など、他のメカニズムによって実装することができます。 Java の特定バージョンに推奨されるメカニズムについては、http://java.sun.com/docs/books/tutorial/ にある Sun 社の Java Tutorial を参照してください。

プログラム・サンプルとしては、以下の資料を参考にしてください。

『例: スレッドを取り消す (Pthread プログラム)』
26 ページの『例: スレッドを取り消す (Java)』

例: スレッドを取り消す (Pthread プログラム)

下記の例は、Pthread プログラムで、長時間実行しているスレッドを取り消す例です。

```
/*
Filename: ATEST13.QCSRC
The output of this example is as follows:
 Enter Testcase - LIBRARY/ATEST13
 Create/start a thread
 Wait a bit until we 'realize' the thread needs to be canceled
 Thread: Entered
 Thread: Looping or long running request
 Thread: Looping or long running request
 Thread: Looping or long running request
 Wait for the thread to complete, and release its resources
 Thread: Looping or long running request
 Thread status indicates it was canceled
 Main completed
*/
#define  _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#define checkResults(string, val) {             ¥
 if (val) {                                     ¥
   printf("Failed with %d at %s", val, string); ¥
   exit(1);                                     ¥
 }                                              ¥
}

void *theThread(void *parm)
{
   printf("Thread: Entered¥n");
   while (1) {
      printf("Thread: Looping or long running request¥n");
      pthread_testcancel();
      sleep(1);
   }
   return NULL;
}

int main(int argc, char **argv)
{
  pthread_t             thread;
  int                   rc=0;
  void                 *status;

  printf("Enter Testcase - %s¥n", argv[0]);

  printf("Create/start a thread¥n");
  rc = pthread_create(&thread, NULL, theThread, NULL);
  checkResults("pthread_create()¥n", rc);

  printf("Wait a bit until we 'realize' the thread needs to be canceled¥n");
  sleep(3);
  rc = pthread_cancel(thread);
  checkResults("pthread_cancel()¥n", rc);

  printf("Wait for the thread to complete, and release its resources¥n");
  rc = pthread_join(thread, &status);
  checkResults("pthread_join()¥n", rc);

  printf("Thread status indicates it was canceled¥n");
  if (status != PTHREAD_CANCELED) {
     printf("Unexpected thread status¥n");
  }

  printf("Main completed¥n");
  return 0;
}
```

スレッドの取り消しについては、 25 ページの『スレッドを取り消す』を参照してください。

例: スレッドを取り消す (Java)

下記の例は、Java プログラムで、長時間実行しているスレッドを取り消す例です。

```
/*
FileName: ATEST13.java

The output of this example is as follows:
 Entered the testcase
 Create a thread
 Start the thread
 Wait a bit until we 'realize' the thread needs to be canceled
 Thread: Entered
 Thread: Looping or long running request
 Thread: Looping or long running request
```

```
 Thread: Looping or long running request
 Wait for the thread to complete
 Thread status indicates it was canceled
 Testcase completed
*/
import java.lang.*;


public class ATEST13 {

   static class theThread extends Thread {
      public final static int THREADPASS     = 0;
      public final static int THREADFAIL     = 1;
      public final static int THREADCANCELED = 2;
      int         _status;

      public int status() {
         return _status;
      }
      public theThread() {
         _status = THREADFAIL;
      }
      public void run() {
         System.out.print("Thread: Entered¥n");
         try {
            while (true) {
               System.out.print("Thread: Looping or long running request¥n");
               try {
                  Thread.sleep(1000);
               }
               catch (InterruptedException e) {
                  System.out.print("Thread: sleep interrupted¥n");
               }
            }
         }
         catch (ThreadDeath d) {
            _status = THREADCANCELED;
         }
      }
   }

   public static void main(String argv[]) {
      System.out.print("Entered the testcase¥n");

      System.out.print("Create a thread¥n");
      theThread t = new theThread();

      System.out.print("Start the thread¥n");
      t.start();

      System.out.print("Wait a bit until we 'realize' the thread needs to be canceled¥n");
      try {
         Thread.sleep(3000);
      }
      catch (InterruptedException e) {
         System.out.print("sleep interrupted¥n");
      }
      t.stop();

      System.out.print("Wait for the thread to complete¥n");
```

```
      try {
         t.join();
      }
      catch (InterruptedException e) {
         System.out.print("Join interrupted¥n");
      }
      System.out.print("Thread status indicates it was canceled¥n");
      if (t.status() != theThread.THREADCANCELED) {
         System.out.print("Unexpected thread status¥n");
      }

      System.out.print("Testcase completed¥n");
      System.exit(0);
   }

}
```

スレッドの取り消しについては、25 ページの『スレッドを取り消す』を参照してください。

スレッドを一時停止する

アプリケーションでスレッドの処理を一時的に停止することが、役立つ場合があります。スレッドを一時停止すると、そのスレッドによって保持されているすべての属性やロックを含むスレッドの状態は、それが再開されるまで保持されます。

スレッドの一時停止は十分注意して使用してください。スレッドを一時停止すると、アプリケーションのデッドロックや、タイムアウト条件の原因となる可能性があります。スレッドの一時停止に関係する問題の多くは、それ以外のもっと安全なメカニズム (同期プリミティブなど) を使うことによって解決できます。追加情報は、41 ページの『スレッド間の同期化テクニック』に提供されています。

注: Java スレッド・クラスの一時停止メソッドは使用すべきではありません。詳細については、25 ページの『スレッドを取り消す』を参照してください。

プログラム・サンプルとしては、『例: スレッドを一時停止する (Java)』を参照してください。

例: スレッドを一時停止する (Java)

下記の例は、Java プログラムで、活動状態で実行しているスレッドを一時停止する例です。

```
/*
FileName: ATEST14.java

The output of this example is as follows:
 Entered the testcase
 Create a thread
 Start the thread
 Wait a bit until we 'realize' the thread needs to be suspended
 Thread: Entered
 Thread: Active processing
 Thread: Active processing
 Suspend the thread
 Wait a bit until we 'realize' the thread needs to be resumed
 Resume the thread
 Thread: Active processing
 Wait for the thread to complete
 Thread: Active processing
 Thread: Active processing
```

```
 Thread: Completed
 Testcase completed
*/
import java.lang.*;


public class ATEST14 {

   static class theThread extends Thread {
      public void run() {
         int        loop=6;
         System.out.print("Thread: Entered¥n");
         while (--loop > 0) {
            System.out.print("Thread: Active processing¥n");
            safeSleep(1000, "Thread: sleep interrupted¥n");
         }
         System.out.print("Thread: Completed¥n");
      }
   }

   public static void main(String argv[]) {
      System.out.print("Entered the testcase¥n");

      System.out.print("Create a thread¥n");
      theThread t = new theThread();

      System.out.print("Start the thread¥n");
      t.start();

      System.out.print("Wait a bit until we 'realize' the thread needs to be suspended¥n");
      safeSleep(2000, "Main first sleep interrupted¥n");
      System.out.print("Suspend the thread¥n");
      t.suspend();

      System.out.print("Wait a bit until we 'realize' the thread needs to be resumed¥n");
      safeSleep(2000, "Main second sleep interrupted¥n");
      System.out.print("Resume the thread¥n");
      t.resume();

      System.out.print("Wait for the thread to complete¥n");
      try {
         t.join();
      }
      catch (InterruptedException e) {
         System.out.print("Join interrupted¥n");
      }

      System.out.print("Testcase completed¥n");
      System.exit(0);
   }

   public static void safeSleep(long milliseconds, String s) {
      try {
         Thread.sleep(milliseconds);
      }
      catch (InterruptedException e) {
         System.out.print(s);
```

```
        }
    }

}
```

スレッドの中断については、 28 ページの『スレッドを一時停止する』を参照してください。

スレッドを再開する

アプリケーションでスレッドの処理を一時的に停止し、後で処理を再開することが役立つ場合があります。一時停止したスレッドの処理を、その時点から再開できます。再開したスレッドは、一時停止した時点と同じロックを保持しており、属性も同じです。

注: Java スレッド・クラスの一時停止メソッドは使用すべきではありません。詳細については、 25 ページの『スレッドを取り消す』を参照してください。

スレッドの終了を待機する

スレッドを使う場合、スレッドがいつ終了するかについて知っておくことは重要です。スレッドがアクションを実行したり、何かのイベントが発生するまで待機することを、スレッド同期といいます。

多くの場合、スレッドの終了を単に待機するだけで十分です。スレッドが終了すると、そのスレッドに割り当てられた処理が完了したこと、またはスレッドが失敗したことがアプリケーションに通知されます。スレッドによって状況が設定されており、使用中のアプリケーション・プログラム・インターフェース (API) セットでその状況がサポートされている場合は、その状況を使うことによって、スレッドが正常にその処理を完了したかどうかを判別できます。

スレッドが終了すると、システムはそのリソースを再利用します。あるスレッドのリソースを他のスレッドで再利用するために、スレッドの終了を待機することもできます。

サーバーなどの大規模なアプリケーションの場合には、一連のスレッドからなるグループの終了を待機するとよい場合があります。そのためには、アプリケーション作業の多くを『ワーカー (作業員)』スレッドに割り当てて、単一の『ボス (責任者)』スレッドでそれら従属スレッドの作業を調整するようにします。使用中の API のセットによっては、この種の待機を直接サポートしていることもあります。

プログラム・サンプルとしては、以下の資料を参考にしてください。
『例: スレッドを待機する (Pthread プログラム)』
32 ページの『例: スレッドを待機する (Java)』

例: スレッドを待機する (Pthread プログラム)

下記の例は、Pthread プログラムにおいていくつかのスレッドを開始し、それらの終了を待機し、完了後にその状況を検査する例です。

```
/*
Filename: ATEST13.QCSRC
The output of this example is as follows:
 Enter Testcase - LIBRARY/ATEST15
 Create/start some worker threads
 Thread 00000000 000001a4: Entered
 Thread 00000000 000001a4: Working
 Wait for worker threads to complete, release their resources
 Thread 00000000 000001a8: Entered
 Thread 00000000 000001a8: Working
 Thread 00000000 000001a5: Entered
```

```
 Thread 00000000 000001a5: Working
 Thread 00000000 000001a6: Entered
 Thread 00000000 000001a6: Working
 Thread 00000000 000001a7: Entered
 Thread 00000000 000001a7: Working
 Thread 00000000 000001a4: Done with work
 Thread 00000000 000001a8: Done with work
 Thread 00000000 000001a6: Done with work
 Thread 00000000 000001a7: Done with work
 Thread 00000000 000001a5: Done with work
 Check all thread's results
 Main completed
*/
#define  _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define THREADGROUPSIZE   5

#define checkResults(string, val) {             ¥
 if (val) {                                     ¥
   printf("Failed with %d at %s", val, string); ¥
   exit(1);                                     ¥
 }                                              ¥
}

void *theThread(void *parm)
{
   printf("Thread %.8x %.8x: Entered¥n", pthread_getthreadid_np());
   printf("Thread %.8x %.8x: Working¥n", pthread_getthreadid_np());
   sleep(15);
   printf("Thread %.8x %.8x: Done with work¥n", pthread_getthreadid_np());
   return NULL;
}

int main(int argc, char **argv)
{
  pthread_t             thread[THREADGROUPSIZE];
  void                 *status[THREADGROUPSIZE];
  int                    i;
  int                   rc=0;

  printf("Enter Testcase - %s¥n", argv[0]);

  printf("Create/start some worker threads¥n");
  for (i=0; i <THREADGROUPSIZE; ++i)
 {
    rc = pthread_create(&thread[i], NULL, theThread, NULL);
  checkResults("pthread_create()¥n", rc);
  }

  printf("Wait for worker threads to complete, release their resources¥n");
  for (i=0; i <THREADGROUPSIZE; ++i) {
rc = pthread_join(thread[i], &status[i]);
    checkResults("pthread_join()¥n", rc);
  }

  printf("Check all thread's results¥n");
  for (i=0; i <THREADGROUPSIZE; ++i) {
if (status[i] != NULL) {
     printf("Unexpected thread status¥n");
    }
  }
```

```
  printf("Main completed¥n");
  return 0;
}
```

スレッド処理の完了待ちについては、 30 ページの『スレッドの終了を待機する』を参照してください。

例: スレッドを待機する (Java)

下記の例は、Java プログラムにおいていくつかのスレッドを開始し、それらの終了を待機し、完了後にその状況を検査する例です。

```
/*
FileName: ATEST15.java

The output of this example is as follows:
 Entered the testcase
 Create some worker threads
 Start the thread
 Thread Thread-1: Entered
 Thread Thread-1: Working
 Thread Thread-2: Entered
 Thread Thread-2: Working
 Thread Thread-3: Entered
 Thread Thread-3: Working
 Thread Thread-4: Entered
 Thread Thread-4: Working
 Wait for worker threads to complete
 Thread Thread-5: Entered
 Thread Thread-5: Working
 Thread Thread-1: Done with work
 Thread Thread-2: Done with work
 Thread Thread-3: Done with work
 Thread Thread-4: Done with work
 Thread Thread-5: Done with work
 Check all thread's results
 Testcase completed
*/
import java.lang.*;


public class ATEST15 {
   public final static int THREADGROUPSIZE   = 5;

   static class theThread extends Thread {
      public final static int THREADPASS     = 0;
      public final static int THREADFAIL     = 1;
      int         _status;

      public int status() {
         return _status;
      }
      public theThread() {
         _status = THREADFAIL;
      }
      public void run() {
         System.out.print("Thread " + getName() + ": Entered¥n");
         System.out.print("Thread " + getName() + ": Working¥n");
         safeSleep(15000, "Thread " + getName() + " work");
         System.out.print("Thread " + getName() + ": Done with work¥n");
         _status = THREADPASS;
      }
   }

   public static void main(String argv[]) {
      int           i=0;
```

```
      theThread     thread[] = new theThread[THREADGROUPSIZE];

      System.out.print("Entered the testcase¥n");

      System.out.print("Create some worker threads¥n");
      System.out.print("Start the thread¥n");
      /* We won't use a ThreadGroup for this example, because we'd */
      /* still have to join all the threads individually           */
  for (i=0; i <THREADGROUPSIZE; ++i) {
  thread[i] = new theThread();
         thread[i].start();
      }

      System.out.print("Wait for worker threads to complete¥n");
      for (i=0; i <THREADGROUPSIZE; ++i) {
try {
            thread[i].join();
         }
         catch (InterruptedException e) {
            System.out.print("Join interrupted¥n");
         }
      }
      System.out.print("Check all thread's results¥n");
      for (i=0; i <THREADGROUPSIZE; ++1) {
if (thread[i].status() != theThread.THREADPASS) {
            System.out.print("Unexpected thread status¥n");
         }
      }

      System.out.print("Testcase completed¥n");
      System.exit(0);
   }

   public static void safeSleep(long milliseconds, String s) {
      try {
         Thread.sleep(milliseconds);
      }
      catch (InterruptedException e) {
         System.out.print(s);
      }
   }

}
```

スレッドの終了待ちについては、 30 ページの『スレッドの終了を待機する』を参照してください。

処理装置を別のスレッドに譲る

あるスレッドがシステム内の別のスレッドに処理装置を譲ることが、アプリケーションの役に立つことがあります。スレッドが処理装置を譲ると、優先順位がそれ以上である別のスレッドのうち、システム内部で活動状態になっているものが、即時実行できるようになります。それ以上の優先順位のスレッドの中で、現在実行可能な状態になっているものがない場合、処理装置を譲っても何の影響もありません。即時アクションを別にすれば、処理装置を譲っても、システム内のスレッドのスケジューリングに関して特に予測可能な動作はありません。

iSeries サーバーでは、完全にプリエンプティブなマルチタスキング・スケジューリング・アルゴリズムが提供されています。適切に作成されたアプリケーションのスレッドの場合、他のスレッドに処理装置を譲る必要はほとんどないはずです。もっと予測可能なアプリケーション・プログラム・インターフェース (API) を使うことによって、スレッド相互の同期を取ることができるからです。

第 5 章 スレッド・セーフティ

ある関数がスレッド・セーフであるのは、同一プロセス内の複数のスレッドで同時にその関数を開始できる場合です。ある関数がスレッド・セーフであるのは、その関数から呼び出されるすべての関数がスレッド・セーフである場合であり、そしてその場合のみです。 62 ページの『スレッド・セーフでない関数呼び出し』では、スレッド・セーフでない関数呼び出しに安全にアクセスする方法を説明しています。 41 ページの『既存のアプリケーションの評価』では、既存のコードのマルチスレッド化への、適合性の評価に関する情報を提供しています。

スレッドを使う前に、OS/400 でのスレッド・セーフティに関する下記の概念を十分に理解しておく必要があります。

ストレージ、ジョブ、および活動化グループでのスレッド・セーフティ

- 『記憶域の使用法とスレッド・アプリケーション』
- 36 ページの『ジョブを有効範囲とするリソースとスレッド・セーフティ』

活動化グループのスレッド・セーフティに関する情報は、8 ページの『活動化グループとスレッド』に記載されています。

API および CL コマンドでのスレッド・セーフティ

- 37 ページの『API スレッド・セーフティの分類』
- 38 ページの『CL コマンドとスレッド・セーフティ』
- 39 ページの『アクセスの拒否される関数とスレッド・セーフティ』
- 39 ページの『出口点』

記憶域の使用法とスレッド・アプリケーション

スレッド・アプリケーションを作成する場合、さまざまなクラスの記憶域の可視性と有効範囲を理解しておくことが大切です。アプリケーションで変数を宣言する場合、複数のスレッドからそれらの変数にアクセスしたり、使用したりすることが可能なことがあります。使用される記憶域の可視性と有効範囲は、多くの場合、アプリケーションに影響を及ぼします。

グローバル記憶域:

アプリケーションのソース・ファイルの 1 つで宣言したグローバル記憶域は、アプリケーションの他のすべてのソース・ファイルまたはモジュールから可視になります。アプリケーション内のコードを実行するすべてのスレッドは、同じグローバル記憶域を共用します。記憶域がこのように意図せずに共用されることは、しばしば安全性の点で問題になります。

静的ストレージ

静的ストレージは、記憶域または変数が宣言されたソース・ファイル、モジュール、関数の範囲でのみ可視であるグローバル記憶域です。そのモジュールのコードを実行するすべてのスレッドは、同一の静的ストレージを共用します。

ヒープ記憶域:

ヒープ記憶域は、アプリケーションによって動的に割り振られたり、割り振り解除されたりします (たとえば、C の malloc() と free()、あるいは Java または C++ の new および delete)。アプリケーション内のコードを実行するすべてのスレッドは、同じヒープを共用します。

割り振り済みヒープ記憶域へのポインターを、静的ストレージ、またはグローバル記憶域の中に格納することによって、別のスレッドがアクセスできるようにしたり、それを別のスレッドに渡したりすると、そのスレッドでその記憶域を使用したり、割り振り解除したりできるようになります。記憶域がこのように意図せずに共用されることは、しばしばスレッド・セーフティの点で問題になります。

自動記憶域:

関数、メソッド、またはサブルーチン専用の変数を宣言すると、使用している言語によって、自動記憶域またはローカル記憶域が自動的に割り振られます。ローカル記憶域は、プロセス内の他のスレッドから不可視です。それらのスレッドが関数、メソッド、またはサブルーチンを呼び出すと、そのスレッドは自動変数の新しいバージョンを割り振ります。各スレッドは、それぞれ独自の自動記憶域を取得します。逐次化と同期化の厄介な問題が関係しているため、スレッドが別のスレッドに由来する自動記憶域にアクセスすることがないようにしてください。

さらに、IBM OS/400 では、グローバル記憶域、静的ストレージ、ヒープ記憶域の有効範囲が、アプリケーション・コードの実行されている活動化グループの範囲内に制限されています。したがって、複数の異なる活動化グループで実行されていながら、同一のグローバル変数や静的変数を使用しているアプリケーション・コードまたはスレッドは、それらの変数、およびその記憶域のさまざまに異なるバージョンにアクセスすることになります。同じように、1 つの活動化グループの中で割り振られたヒープ記憶域が、それとは異なる活動化グループで割り振り解除されることはありません。

言語によっては、変数としてグローバル記憶域クラス、または静的ストレージ・クラスしかサポートしていないものもあります。スレッド・アプリケーションにそのような言語を使うのは容易ではありません。そのような言語では、スレッドを使わないでください。

ジョブを有効範囲とするリソースとスレッド・セーフティ

多くのシステム・リソースとアプリケーション・リソースは、定義されたジョブの内部でのみ使用可能です。マルチスレッド・アプリケーション、またはサービスを作成する場合は、ジョブを有効範囲とするリソースの使用について評価する必要があります。それらのリソースを使用して、プロセス内の他のスレッドと競合したり、悪い影響を与えたりしないようにするためです。

リソースによっては、有効範囲が活動化グループであるものもあります (たとえばオープンされているデータベース・ファイル)。ここでの文脈に関する限り、活動化グループを有効範囲とするリソースの処理では、ジョブを有効範囲とするリソースと同じ注意事項が当てはまります。これらの事項は、その活動化グループを使うすべてのスレッドに当てはまります。

アプリケーションの他のスレッドがリソースを使っているときに、あるスレッドがそれらを変更する場合、アプリケーションでは適切な同期テクニックを使用できますが、その場合、スレッド相互間でリソースが一貫して同じに見えるようにする必要があります。

ジョブまたは活動化グループを有効範囲とするリソースとして一般的なものには、次のものがあります。

ヒープ記憶域、静的ストレージ、およびグローバル記憶域:

最も一般的な共用リソースです。詳細については、記憶域の使用方法の部分を参照してください。

ファイル・オープン:

ファイルをオープンした後、ファイル・ハンドル・ポインターまたはファイル記述子番号を別のスレッドに渡すことによって、プロセス内のスレッド相互間で、統合ファイル・システム・ファイルおよびデータベース・ファイルを直接に共用できます。作業ディレクトリーは、常にプロセスの有効範囲になります。

ロケール:

アプリケーションのロケールは活動化グループ・リソースです。すべてのスレッドはそのロケールを共用します。ロケールを変更すると、照合順序やその他のロケール情報に関して他のスレッドに影響します。

CCSID:

ジョブの CCSID を変更すると、そのジョブ内のすべてのスレッドの現行のデータ変換、またはデータ表現に影響します。

環境変数:

一般的なアプリケーションでは、構成とオプショナルの動作に環境変数を使います。ジョブのすべてのスレッドがこれらの変数を共用します。

API スレッド・セーフティの分類

アプリケーション・プログラミング・インターフェース (API) を使用する前に、System Programming Interface Reference を参照して、マルチスレッド・プログラム内でその API を呼び出すことが、安全かどうかを判断してください。 System Programming Interface Reference に記載されている各 API には、それぞれスレッド・セーフのタイプがあります。スレッド・セーフのタイプとしては、下記の 3 つのものがあります。

スレッド・セーフ: yes (あり)

このタイプは、複数のスレッドで同時にかつ安全に、制限なしでその API を呼び出せることを示しています。また、このタイプは、この API によって呼び出されるすべての関数もスレッド・セーフであることを示しています。

スレッド・セーフ: conditional (条件付き)

このタイプは、その API によって提供される関数のすべてがスレッド・セーフであるとは限らないことを示しています。 API の使用上の注意の中に、スレッド・セーフの制限に関連した情報が含まれています。基本となるなんらかのシステム・サポートがスレッド・セーフではないため、またはその API から出口点が呼び出されることがあるため、多くの API は条件付きスレッド・セーフに分類されています。たとえば、ファイル・システム API の多くは、スレッド・セーフ・ファイル・システム内のファイルで使う限りは、完全にスレッド・セーフです。いくつかの条件付きスレッド・セーフ API は、特定の状況下でアクセスを拒否する場合があります。 API の使用上の注意の中に、その関数がアクセスを拒否する原因となる条件を説明してあります。

スレッド・セーフ: no (なし)

このタイプは、その API がスレッド・セーフではなく、マルチスレッド・プログラムで使用すべきでないことを示しています。一部のスレッド・セーフでない API はアクセスを拒否する可能性がありますが、ほとんどのスレッド・セーフでない API は拒否しません。 CL コマンドの場合とは違って、アクセスを拒否する API 以外の API のうちスレッド・セーフでない API が呼び出されたことを示す診断メッセージは、ジョブ・ログに出力されません。マルチスレッド・プログラムの中でスレッド・セーフでない関数を呼び出すために、いろいろなテクニックがあります。

CL コマンドとスレッド・セーフティ

ILE CL ランタイムおよびコンパイラー生成コードはスレッド・セーフです。 OPM CL プログラムは、スレッド・セーフではありません。バージョン 4 リリース 3 より前にコンパイルされた ILE CL コードまたは OPM CL コードの場合、 CL ランタイムは CPD000B 診断メッセージを出して実行を続けますが、結果は予測できません。これがスレッド・セーフであるかどうかは、基礎となるコードによります。

コマンド分析プログラムはスレッド・セーフです。スレッド・セーフ属性 (THDSAFE) が *NO、マルチスレッド・ジョブ・アクション属性 (MLTTHDACN) が *NORUN または *MSG に設定されているコマンドを、マルチスレッド操作が可能なジョブから呼び出すと、コマンド分析プログラムは次のいずれかを実行します。

- マルチスレッド・ジョブで MLTTHDACN が *NORUN に設定されている場合、コマンド分析プログラムは CPD000D 診断メッセージを送信し、コマンドは実行されません。 CPD000D 診断メッセージに続いて CPF0001 エスケープ・メッセージが出されます。
- マルチスレッド・ジョブで MLTTHDACN が *MSG に設定されている場合、コマンド分析プログラムは CPD000D 診断メッセージを送信し、プログラムの実行が継続します。ただし、結果は予測できません。
- 複数のスレッドを実行できるものの、マルチスレッドではないジョブの場合、コマンド分析プログラムは、スレッド・セーフでないコマンドが支障なく実行することを許可します。

MLTTHDACN が *RUN に設定されている場合、コマンド分析プログラムは、診断メッセージを送信せず、コマンドの実行を許可します。ただし、結果は予測できません。また、MLTTHDACN は、THDSAFE 値が *NO であるコメントにしか適用されません。コマンドの THDSAFE 値と MLTTHDACN 値を判別するには、コマンドの表示 (DSPCMD) を使用してください。

MLTTHDACN 値が *SYSVAL に設定されるコマンドもあります。この場合、コマンド分析プログラムはコマンドの処理方法を決定するために、QMLTTHDACN システム値を使用します。

OS/400 上でこの値の設定を表示するには、以下のように入力します。

```
DSPSYSVAL SYSVAL(QMLTTHDACN)
```

OS/400 上でこの値の設定を変更するには、以下のように入力します。

```
CHGSYSVAL SYSVAL(QMLTTHDACN) VALUE(x)
```

DSPSYSVAL からの出力例は、以下のとおりです。

```
                    システム値の表示


システム値  . . . . . : QMLTTHDACN
記述  . . . . . . . . : マルチスレッド・ジョブの処置



マルチスレッド・ジョブの処置  . . . . . . . . : 2
                                1=メッセージを搬出しないでスレッド・セーフでない関数を実行
                                2=スレッド・セーフでない関数を実行して通知メッセージを送出
                                3=スレッド・セーフでない関数を実行しない
```

以下のトピックに、CL コマンドとスレッドについての関連情報が記載されています。

- 41 ページの『第 6 章 マルチスレッド・プログラミング・テクニック』
- 62 ページの『スレッド・セーフでない関数呼び出し』

アクセスの拒否される関数とスレッド・セーフティ

システム保全性の問題とデータ破損の問題を避けるため、特定のアプリケーション・プログラミング・インターフェース (API) とコマンドは、条件付きスレッド・セーフおよび非スレッド・セーフになっています。それらの API およびコマンドは、すべてのアクセスを拒否したり一部のアクセスを拒否したりします。アクセスを拒否する基準として、下記の 3 つの基準があります。

マルチスレッド可能:

この場合、関数へのアクセスを拒否するかどうかは、マルチスレッド可能ジョブ属性を検査することによって決定されます。ジョブがスレッドをサポートしている場合、現在ジョブ内に存在するスレッドの数には関係なく、その関数を呼び出すことはできません。関数がマルチスレッドの可能性に基づいてアクセスを拒否する場合、その関数によって、CPF1892 (関数 &1 は使用できない) エスケープ・メッセージが呼び出し元に送られます。マルチスレッドの可能性に基づいてアクセスを拒否する関数は、別個のジョブで呼び出すか、またはまったく呼び出さないようにしてください。2 次スレッドから呼び出される、マルチスレッド可能ジョブ内でアクセスを拒否する関数の例としては、**リソースの再利用 (RCLRSC)** があります。詳細は、「CL 解説書」の リソースの再利用 (RCLRCS) コマンド を参照してください。

初期スレッド:

一部の関数は、ジョブの初期スレッドがその関数を呼び出すかどうかに基づいてアクセスを拒否します。 2 次スレッドで関数を呼び出した場合、その関数によって CPF180C (関数 &1 は使用できない) エスケープ・メッセージが、呼び出し元に送られます。初期スレッドへの要求を経路指定して、初期スレッドから関数を呼び出すことによって、 2 次スレッドからアクセスを拒否する関数を呼び出すことは可能です。初期スレッドしか使えない関数の例としては、データベース・ファイルへの一時変更があります。

複数のスレッド:

この場合、アクセスを拒否するかどうかは、ジョブ内のスレッドの数によって決定されます。ジョブに複数のスレッドがある場合、その関数によって CPF180B (関数 &1 は使用できない) エスケープ・メッセージが呼び出し元に送られます。エラー番号を戻すその他の関数は、errno を **ENOTSAFE** に設定します。

1 つのスレッドのみが活動状態の場合にのみ、関数を使わないようにすることにより、スレッドが複数個ある場合に、アクセスを拒否する関数を呼び出すことは可能です。このような関数の例としては、スレッド・セーフでないファイル・システムに含まれる、ファイルにアクセスするために使われるファイル・システム API があります。

出口点

OS/400 登録機能により、アプリケーション内の関数の出口点を定義し、それらの出口点で実行するプログラムを登録することができます。また、OS/400 のサービスの中には、出口プログラムを登録するための登録機能をサポートするものもあります。それらのサービスには、インストール時に登録される出口点が事前定義されています。登録機能自体はスレッド・セーフです。出口プログラム項目のスレッド・セーフティ属性とマルチスレッド・ジョブ・アクション属性を指定するために、登録機能を使用できます。

マルチスレッド・ジョブで出口プログラムを呼び出すことはできても、注意深く評価せずに既存の出口プログラムをスレッド・セーフと見なすことがないようにしてください。出口プログラムに適用される制限は、

マルチスレッド・ジョブで実行されるその他のすべてのコードに適用されるものと同じです。たとえば、スレッド・セーフな ILE 言語を使って作成した出口プログラムでなければ、スレッド・セーフにすることはできません。

スレッド・アプリケーションの出口点の関連情報は、41 ページの『既存のアプリケーションの評価』に記載されています。

トリガー・プログラム、ユーザー定義関数 (UDF)、およびストアード・プロシージャーは、出口点として機能することができます。トリガー・プログラムおよびストアード・プロシージャーの詳細については、10 ページの『マルチスレッド・プログラミングでのデータベースについての考慮事項』を参照してください。

第 6 章 マルチスレッド・プログラミング・テクニック

マルチスレッド・アプリケーション、または OS/400 用のサーバー・アプリケーションを作成する場合、次のようないくつかのテクニックに精通しておく必要があります。

- 『既存のアプリケーションの評価』
- 『スレッド間の同期化テクニック』
- 56 ページの『一回限りの初期化とスレッド・セーフティ』
- 58 ページの『スレッド固有のデータ』
- 62 ページの『スレッド・セーフでない関数呼び出し』
- 64 ページの『一般的なマルチスレッド・プログラミング・エラー』

このセクションで説明する概念は、すべてのプログラミング言語に当てはまります。それぞれの言語でそれらの概念がどのように実現されているかについては、各言語のプログラマーの手引きを参照してください。

既存のアプリケーションの評価

マルチスレッド・アプリケーションを作成するときは、アプリケーションのすべてのパーツ、またスレッド・セーフティのためにアプリケーションが使用するすべてのサービスを評価する必要があります。

アプリケーションで使用される各サービスまたはアプリケーション・プログラム・インターフェース (API) がどのように記憶リソースを使うかは、スレッド・セーフなアプリケーションを提供するために重要な考慮事項です。スレッド・セーフである方法で記憶域を使用しない場合は、アプリケーションのデータは破壊される可能性があります。

スレッド・セーフなアプリケーションを作成する点で他に重要なのは、そのアプリケーション、またはアプリケーション・サービスが依存する API と、システム・サービスです。スレッド・セーフにするには、使用する記憶域、また直接 / 間接的に使用するすべての API、およびサービスがスレッド・セーフでなければなりません。間接的に使用する API およびシステム・リソースは、評価するのが特に困難です。特定の API やシステム・サービスがスレッド・セーフかどうかは、解説書を参照してください。

呼び出すシステム・サービス、 API または他のアプリケーションのソース・コードを持っていないということが頻繁にあります。それらのサービスの多くは、スレッド・セーフか非スレッド・セーフかの分類がなされていません。そのため、それらのサービスはスレッド・セーフではないと見なさなければなりません。

スレッド間の同期化テクニック

作成しているコードがスレッド・セーフではあるが、スレッド間でのデータまたはリソースの共用から利点を得る場合、プログラミングの最も重要な点はスレッドの同期化を行う能力です。同期化とは 2 つ以上のスレッドの連携アクションです。各スレッドが、他のスレッドと同調して特定のポイントに達してから処理を続行するようにします。同期化を正しく使用せずにリソースの共用を試みると、アプリケーション・データは損傷を受けることがあります。

通常、2 つのスレッドの同期化には、1 つ以上の同期化プリミティブの使用が関係します。同期化プリミティブとは、アプリケーションで必要な同期化動作を得るためにアプリケーションが使用または作成する、低レベルの関数またはアプリケーション・オブジェクト (OS/400 オブジェクトではない) です。

最も一般的な同期化プリミティブは以下のとおりです。計算コストが低い方から高い方への順序になっています。

- 56 ページの『Compare and Swap (比較とスワップ)』
- 『mutex とスレッド』
- 45 ページの『セマフォーとスレッド』
- 48 ページの『条件変数とスレッド』
- 53 ページの『同期化プリミティブとしてのスレッド』
- 53 ページの『スペース・ロケーション・ロック』
- 55 ページの『オブジェクト・ロック』

62 ページの『スレッド・セーフでない関数呼び出し』は、スレッド・セーフではない関数にアクセスするためのセーフ・メソッドを記載しています。

これらの概念は、すべてのプログラミング言語に当てはまります。それぞれの言語でそれらの概念がどのように実現されているかについては、各言語のプログラマーの手引きを参照してください。

mutex とスレッド

mutex という言葉は、スレッド間での相互排他 (MUTual EXclusion) を実現するプリミティブ・オブジェクトの省略です。 mutex (相互排他) は、連携しているスレッドのうち一度に 1 つだけがデータにアクセスしたり、特定のアプリケーション・コードを実行するよう見届けるため、スレッド間で連携して使用します。入門編として、mutex のことをクリティカル・セクションやモニターのように考えることができます。

| 通常 mutex は、アプリケーションにより、保護対象のデータと論理的に関連付けられます。たとえば、
| PAYROLL DATA に PAYROLL MUTEX が関連付けられているとします。アプリケーション・コード
| は、PAYROLL DATA にアクセスする前に必ず PAYROLL MUTEX をロックします。別のスレッドがデ
| ータにアクセスする前に mutex を使おうとすると、mutex はそのデータへのアクセスを阻止します。

作成 (create)、ロック (lock)、アンロック (unlock)、および破棄 (destroy) は、mutex で実行される典型的な操作です。 mutex を正常にロックしたスレッドが、アンロックされるまでその mutex の所有者になります。 mutex のロックを試みるスレッドは、それがアンロックされるまで待機しなければなりません。所有者が mutex をアンロックすると、待機していたスレッドに制御が移り、今度はそのスレッドが mutex の所有者になります。 mutex の所有者は常に 1 つです。

mutex の待機操作は再帰的です。 mutex が再帰的であるため、所有者は何度でも繰り返し mutex をロックできます。アンロック要求の数が、正常に行われたロック要求の数と同じになるまで、mutex の所有者は変わりません。ユーザーが指定した時間が経過すると mutex 待機がタイムアウトになります。ロックを取得できなければ mutex 待機は即時に戻されます。詳細については、ご使用の API の資料で、アプリケーションで実際に使用可能な mutex プリミティブを参照してください。

プログラム・サンプルとしては、以下の資料を参考にしてください。

- 『例: mutex の使用 (Pthread プログラム)』
- 44 ページの『例: mutex の使用 (Java)』

例: mutex の使用 (Pthread プログラム)

次の例は、mutex を使って共用データへのアクセスを保護するいくつかのスレッドを開始する、 Pthread プログラムを示しています。

```
/*
Filename: ATEST16.QCSRC
The output of this example is as follows:
```

```
 Enter Testcase - LIBRARY/ATEST16
 Hold Mutex to prevent access to shared data
 Create/start threads
 Wait a bit until we are 'done' with the shared data
 Thread 00000000 00000019: Entered
 Thread 00000000 0000001a: Entered
 Thread 00000000 0000001b: Entered
 Unlock shared data
 Wait for the threads to complete, and release their resources
 Thread 00000000 00000019: Start critical section, holding lock
 Thread 00000000 00000019: End critical section, release lock
 Thread 00000000 0000001a: Start critical section, holding lock
 Thread 00000000 0000001a: End critical section, release lock
 Thread 00000000 0000001b: Start critical section, holding lock
 Thread 00000000 0000001b: End critical section, release lock
 Clean up
 Main completed
*/
#define  _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define checkResults(string, val) {             ¥
 if (val) {                                     ¥
   printf("Failed with %d at %s", val, string); ¥
   exit(1);                                     ¥
 }                                              ¥
}

#define                 NUMTHREADS   3
pthread_mutex_t         mutex = PTHREAD_MUTEX_INITIALIZER;
int                     sharedData=0;
int                     sharedData2=0;

void *theThread(void *parm)
{
   int   rc;
   printf("Thread %.8x %.8x: Entered¥n", pthread_getthreadid_np());
   rc = pthread_mutex_lock(&mutex);
   checkResults("pthread_mutex_lock()¥n", rc);
   /********** Critical Section ******************/
   printf("Thread %.8x %.8x: Start critical section, holding lock¥n",
            pthread_getthreadid_np());
   /* Access to shared data goes here */
   ++sharedData; --sharedData2;
   printf("Thread %.8x %.8x: End critical section, release lock¥n",
          pthread_getthreadid_np());
   /********** Critical Section ******************/
   rc = pthread_mutex_unlock(&mutex);
   checkResults("pthread_mutex_unlock()¥n", rc);
   return NULL;
}

int main(int argc, char **argv)
{
  pthread_t             thread[NUMTHREADS];
  int                   rc=0;
  int                   i;

  printf("Enter Testcase - %s¥n", argv[0]);

  printf("Hold Mutex to prevent access to shared data¥n");
  rc = pthread_mutex_lock(&mutex);
   checkResults("pthread_mutex_lock()¥n", rc);

  printf("Create/start threads¥n");
```

```
  for (i=0; i<NUMTHREADS; ++i) {
 rc = pthread_create(&thread[i], NULL, theThread, NULL);
     checkResults("pthread_create()¥n", rc);
  }

  printf("Wait a bit until we are 'done' with the shared data¥n");
  sleep(3);
  printf("Unlock shared data¥n");
   rc = pthread_mutex_unlock(&mutex);
   checkResults("pthread_mutex_lock()¥n",rc);

  printf("Wait for the threads to complete, and release their resources¥n");
  for (i=0; i <NUMTHREADS; ++i) {
   rc = pthread_join(thread[i], NULL);
    checkResults("pthread_join()¥n", rc);
  }

  printf("Clean up¥n");
  rc = pthread_mutex_destroy(&mutex);
  printf("Main completed¥n");
  return 0;
}
```

スレッドでの mutex の使用については、42 ページの『mutex とスレッド』を参照してください。

例: mutex の使用 (Java)

次の例は、コードのクリティカル・セクションを作成する Java プログラムを示しています。 Java では、コード・ブロックまたはメソッドで同期化キーワードを使うことにより、任意のオブジェクトまたは配列に、mutex と同じ機能をさせることができます。

```
/*
FileName: ATEST16.java
The output of this example is as follows:
 Entered the testcase
 Synchronize to prevent access to shared data
 Create/start the thread
 Thread Thread-1: Entered
 Thread Thread-2: Entered
 Wait a bit until we're 'done' with the shared data
 Thread Thread-3: Entered
 Unlock shared data
 Thread Thread-1: Start critical section, in synchronized block
 Thread Thread-1: End critical section, leave synchronized block
 Thread Thread-2: Start critical section, in synchronized block
 Thread Thread-2: End critical section, leave synchronized block
 Thread Thread-3: Start critical section, in synchronized block
 Thread Thread-3: End critical section, leave synchronized block
 Wait for the threads to complete
 Testcase completed
*/
import java.lang.*;


public class ATEST16 {
   public final static int NUMTHREADS   = 3;
   public static int sharedData  = 0;
   public static int sharedData2 = 0;
   /* Any real java object or array would suit for synchronization          */
   /* We invent one here since we have two unique data items to synchronize */
   /* An in this simple example, they're not in an object                   */
   static class theLock extends Object {
   }
   static public theLock lockObject = new theLock();
```

```
   static class theThread extends Thread {
      public void run() {
         System.out.print("Thread " + getName() + ": Entered¥n");
         synchronized (lockObject) {
            /********** Critical Section ******************/
            System.out.print("Thread " + getName() +
                             ": Start critical section, in synchronized block¥n");
            ++sharedData; --sharedData2;
            System.out.print("Thread " + getName() +
                             ": End critical section, leave synchronized block¥n");
            /********** Critical Section ******************/
         }
      }
   }

   public static void main(String argv[]) {
      theThread threads[] = new theThread[NUMTHREADS];
      System.out.print("Entered the testcase¥n");

      System.out.print("Synchronize to prevent access to shared data¥n");
      synchronized (lockObject) {

      System.out.print("Create/start the thread¥n");
         for (int i=0; i<NUMTHREADS; ++i) {
         threads[i] = new theThread();
        threads[i].start();
         }

         System.out.print("Wait a bit until we're 'done' with the shared data¥n");
         try {
            Thread.sleep(3000);
         }
          catch (InterruptedException e) {
           System.out.print("sleep interrupted¥n");
         }
         System.out.print("Unlock shared data¥n");
      }

      System.out.print("Wait for the threads to complete¥n");
      for(int i=0; i <NUMTHREADS; ++i) {
            threads[i].join();
         }
          catch (InterruptedException e) {
         System.out.print("Join interrupted¥n");
         }
      }

      System.out.print("Testcase completed¥n");
      System.exit(0);
   }

}
```

スレッドでの mutex の使用については、 42 ページの『mutex とスレッド』を参照してください。

セマフォーとスレッド

セマフォー (カウント・セマフォーとも呼ばれる) は、共用リソースへのアクセスを制御するために使用します。セマフォーは、1 つのインテリジェント・カウンターと考えることができます。どのセマフォーにもカレント・カウントがあり、その値は 0 以上です。

どのスレッドもカウントを減分して、セマフォーをロックすることができます (このことを、セマフォー待機ともいいます)。 0 を超えてカウントを減分しようとすると、呼び出し側のスレッドは別のスレッドが、そのセマフォーをアンロックするまで待機します。

どのスレッドもカウントを増分してセマフォーをアンロックすることができます (このことを、セマフォー通知ともいいます)。セマフォーを通知すると、待機中のスレッドがあればウェイクアップされます。

最も単純な形式 (初期カウント 1) では、セマフォーを 1 つの mutex (相互排他) と考えることができます。セマフォーと mutex の重大な違いは、所有権の概念です。セマフォーの場合、所有権は関連付けられません。 mutex とは違い、セマフォーを一度も待機 (ロック) したことがないスレッドが、セマフォーを通知 (アンロック) することが可能なのです。そのために、アプリケーションが予期せぬ動作を起こすことがあるので、できるだけそのような操作は避けてください。

OS/400 が提供するセマフォー API には、他にも次のような機能があります。

- ファイル許可に似たセマフォー許可機能を含む、完成度の高い管理機能。
- いくつかのセマフォーをセットにグループ化し、そのグループに対してアトミックな操作を実行する機能。
- マルチカウント待機や通知操作を行う機能。
- セマフォーのカウントが 0 になるまで待機する機能。
- 特定の状況下で別のスレッドが実施した操作を取り消す機能。

注: Java にはセマフォーを使用する機能がありません。

プログラム・サンプルとしては、『例: セマフォーを使って共用データを保護する (Pthread プログラム)』を参照してください。

例: セマフォーを使って共用データを保護する (Pthread プログラム)

次の例は、セマフォー・セットを使って共用データへのアクセスを保護するいくつかのスレッドを開始する、 Pthread プログラムを示しています。

```
/*
Filename: ATEST17.QCSRC
The output of this example is as follows:
 Enter Testcase - LIBRARY/ATEST17
 Wait on semaphore to prevent access to shared data
 Create/start threads
 Wait a bit until we are 'done' with the shared data
 Thread 00000000 00000020: Entered
 Thread 00000000 0000001f: Entered
 Thread 00000000 0000001e: Entered
 Unlock shared data
 Wait for the threads to complete, and release their resources
 Thread 00000000 0000001f: Start critical section, holding semaphore
 Thread 00000000 0000001f: End critical section, release semaphore
 Thread 00000000 00000020: Start critical section, holding semaphore
 Thread 00000000 00000020: End critical section, release semaphore
 Thread 00000000 0000001e: Start critical section, holding semaphore
 Thread 00000000 0000001e: End critical section, release semaphore
 Clean up
 Main completed
*/
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
```

```
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/stat.h>

#define checkResults(string, val) {             ¥
 if (val) {                                     ¥
   printf("Failed with %d at %s", val, string); ¥
   exit(1);                                     ¥
 }                                              ¥
}

#define                 NUMTHREADS   3
int                     sharedData=0;
int                     sharedData2=0;

/* Simple semaphore used here is actually a set of 1                  */
int                     semaphoreId=-1;
/* Simple lock operation. 0=which-semaphore, -1=decrement, 0=noflags  */
struct sembuf lockOperation = { 0, -1, 0};
/* Simple unlock operation. 0=which-semaphore, 1=increment, 0=noflags */
struct sembuf unlockOperation = { 0, 1, 0};

void *theThread(void *parm)
{
   int   rc;
   printf("Thread %.8x %.8x: Entered¥n", pthread_getthreadid_np());
   rc = semop(semaphoreId, &lockOperation, 1);
   checkResults("semop(lock)¥n",rc);
   /********** Critical Section ******************/
   printf("Thread %.8x %.8x: Start critical section, holding semaphore¥n",
          pthread_getthreadid_np());
   /* Access to shared data goes here */
   ++sharedData; --sharedData2;
   printf("Thread %.8x %.8x: End critical section, release semaphore¥n",
          pthread_getthreadid_np());
   /********** Critical Section ******************/
   rc = semop(semaphoreId, &unlockOperation, 1);
   checkResults("semop(unlock)¥n",rc);
   return NULL;
}

int main(int argc, char **argv)
{
  pthread_t             thread[NUMTHREADS];
  int                   rc=0;
  int                   i;


  printf("Enter Testcase - %s¥n", argv[0]);
  /* Create a private semaphore set with 1 semaphore that only I can use */
  semaphoreId = semget(IPC_PRIVATE, 1, S_IRUSR|S_IWUSR);
  if (semaphoreId &lt 0) { printf("semget failed, err=%d¥n",errno); exit(1); }
  /* Set the semaphore (#0 in the set) count to 1. Simulate a mutex */
  rc = semctl(semaphoreId, 0, SETVAL, (int)1);
  checkResults("semctl(SETALL)¥n", rc);

  printf("Wait on semaphore to prevent access to shared data¥n");
  rc = semop(semaphoreId, &lockOperation, 1);
   checkResults("semop(lock)¥n",rc);

  printf("Create/start threads¥n");
  for (i=0; i <NUMTHREADS; ++i) {
  checkResults("pthread_create()¥n", rc);
  }

  printf("Wait a bit until we are 'done' with the shared data¥n");
  sleep(3);
```

```
  printf("Unlock shared data¥n");
  rc = semop(semaphoreId, &unlockOperation, 1);
  checkResults("semop(unlock)¥n",rc);


  printf("Wait for the threads to complete, and release their resources¥n");
  for (i=0; i <NUMTHREADS; ++i) {
  rc = pthread_join(thread[i], NULL);
      checkResults("pthread_join()¥n", rc);
  }

  printf("Clean up¥n");
  rc = semctl(semaphoreId, 0, IPC_RMID);
  checkResults("semctl(removeID)¥n", rc);
  printf("Main completed¥n");
  return 0;
}
```

スレッドでのセマフォーの使用については、45 ページの『セマフォーとスレッド』を参照してください。

条件変数とスレッド

条件変数は、特定のイベントまたは条件が生じるまで、スレッドが待機できるようにします。また同じイベントまたは条件が生じるのを待っている他のスレッドにも通知します。スレッドはある条件変数で待機し、その条件変数で待機している別のスレッドまたはすべてのスレッドがアクティブになるよう、条件をブロードキャストすることができます。条件変数は、他のプラットフォームにおけるイベントを使ったスレッドの同期化と同じように考えることができます。

条件変数には、関連付けられた所有権というものがなく、通常は状態もありません。あるスレッドが、待機しているスレッドをウェイクアップするよう、条件変数に信号を出したときに、待機しているスレッドがなければ、信号は廃棄され、いかなるアクションも実行されません。状態のない条件変数とはそのような意味です。信号は失われたも同然です。あるスレッドが条件の信号を出した直後に、別のスレッドが何のアクションも実行せずに待機を始めるということはあり得ます。

mutex を使用するロック・プロトコルは、通常、条件変数と共に使用されます。ロック・プロトコルを使用する場合、ウェイクアップのための信号がスレッドで失われないように、アプリケーションで保証することができます。

プログラム・サンプルとしては、以下の資料を参考にしてください。

- 『例: 条件変数の使用 (Pthread プログラム)』
- 51 ページの『例: 条件変数の使用 (Java プログラム)』

例: 条件変数の使用 (Pthread プログラム)

次の例は、条件変数を使ってスレッドに条件を通知する Pthread プログラムを示しています。どのような mutex ロック・プロトコルが使用されているかに注目してください。

```
/*
Filename: ATEST18.QCSRC
The output of this example is as follows:
 Enter Testcase - LIBRARY/ATEST18
 Create/start threads
 Producer: 'Finding' data
 Consumer Thread 00000000 00000022: Entered
 Consumer Thread 00000000 00000023: Entered
 Consumer Thread 00000000 00000022: Wait for data to be produced
 Consumer Thread 00000000 00000023: Wait for data to be produced
 Producer: Make data shared and notify consumer
 Producer: Unlock shared data and flag
```

```
 Producer: 'Finding' data
 Consumer Thread 00000000 00000022: Found data or Notified, CONSUME IT while holding lock
 Consumer Thread 00000000 00000022: Wait for data to be produced
 Producer: Make data shared and notify consumer
 Producer: Unlock shared data and flag
 Producer: 'Finding' data
 Consumer Thread 00000000 00000023: Found data or Notified, CONSUME IT while holding lock
 Consumer Thread 00000000 00000023: Wait for data to be produced
 Producer: Make data shared and notify consumer
 Producer: Unlock shared data and flag
 Producer: 'Finding' data
 Consumer Thread 00000000 00000022: Found data or Notified, CONSUME IT while holding lock
 Consumer Thread 00000000 00000022: All done
 Producer: Make data shared and notify consumer
 Producer: Unlock shared data and flag
 Wait for the threads to complete, and release their resources
 Consumer Thread 00000000 00000023: Found data or Notified, CONSUME IT while holding lock
 Consumer Thread 00000000 00000023: All done
 Clean up
 Main completed
*/
#define  _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define checkResults(string, val) {             ¥
 if (val) {                                     ¥
   printf("Failed with %d at %s", val, string); ¥
   exit(1);                                     ¥
 }                                              ¥
}

#define                 NUMTHREADS   2
pthread_mutex_t         dataMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t          dataPresentCondition = PTHREAD_COND_INITIALIZER;
int                     dataPresent=0;
int                     sharedData=0;

void *theThread(void *parm)
{
   int   rc;
   int   retries=2;

   printf("Consumer Thread %.8x %.8x: Entered¥n", pthread_getthreadid_np());
   rc = pthread_mutex_lock(&dataMutex);
   checkResults("pthread_mutex_lock()¥n", rc);

   while (retries--) {
      /* The boolean dataPresent value is required for safe use of */
      /* condition variables. If no data is present we wait, other */
      /* wise we process immediately.                              */
      while (!dataPresent) {
         printf("Consumer Thread %.8x %.8x: Wait for data to be produced¥n");
         rc = pthread_cond_wait(&dataPresentCondition, &dataMutex);
         if (rc) {
            printf("Consumer Thread %.8x %.8x: condwait failed, rc=%d¥n",rc);
            pthread_mutex_unlock(&dataMutex);
            exit(1);
         }
      }
      printf("Consumer Thread %.8x %.8x: Found data or Notified, "
             "CONSUME IT while holding lock¥n",
             pthread_getthreadid_np());
      /* Typically an application should remove the data from being    */
      /* in the shared structure or Queue, then unlock. Processing     */
```

```
      /* of the data does not necessarily require that the lock is held */
      /* Access to shared data goes here */
      --sharedData;
      /* We consumed the last of the data */
      if (sharedData==0) {dataPresent=0;}
      /* Repeat holding the lock. pthread_cond_wait releases it atomically */
   }
   printf("Consumer Thread %.8x %.8x: All done¥n",pthread_getthreadid_np());
    rc = pthread_mutex_unlock(&dataMutex);
   checkResults("pthread_mutex_unlock()¥n", rc);
   return NULL;
}

int main(int argc, char **argv)

{
  pthread_t              thread[NUMTHREADS];
   int                   rc=0;
   int                   amountOfData=4;
   int                   i;

   printf("Enter Testcase - %s¥n", argv[0]);

   printf("Create/start threads¥n");
   for (i=0; i <NUMTHREADS; ++i) {
 rc = pthread_create(&thread[i], NULL, theThread, NULL);
     checkResults("pthread_create()¥n", rc);
   }

   /* The producer loop */
   while (amountOfData--) {
      printf("Producer: 'Finding' data¥n");
      sleep(3);

      rc = pthread_mutex_lock(&dataMutex);   /* Protect shared data and flag  */
      checkResults("pthread_mutex_lock()¥n", rc);
      printf("Producer: Make data shared and notify consumer¥n");
      ++sharedData;                          /* Add data                      */
      dataPresent=1;                         /* Set boolean predicate         */

      rc = pthread_cond_signal(&dataPresentCondition); /* wake up a consumer  */
      if (rc) {
         pthread_mutex_unlock(&dataMutex);
         printf("Producer: Failed to wake up consumer, rc=%d¥n", rc);
         exit(1);
      }

      printf("Producer: Unlock shared data and flag¥n");
      rc = pthread_mutex_unlock(&dataMutex);
      checkResults("pthread_mutex_lock()¥n",rc);
   }

   printf("Wait for the threads to complete, and release their resources¥n");
   for (i=0; i <NUMTHREADS; ++i) {
  rc = pthread_join(thread[i], NULL);
    checkResults("pthread_join()¥n", rc);
   }

    printf("Clean up¥n");
   rc = pthread_mutex_destroy(&dataMutex);
   rc = pthread_cond_destroy(&dataPresentCondition);
    printf("Main completed¥n");
    return 0;
}
```

スレッドでの条件変数の使用については、48 ページの『条件変数とスレッド』を参照してください。

例: 条件変数の使用 (Java プログラム)

次の例は、Java オブジェクトで wait および notify メソッドの形で条件変数を使用する Java プログラムを示しています。どのようなロック・プロトコルが使用されているかに注目してください。

```
/*
FileName: ATEST18.java
The output of this example is as follows:
 Entered the testcase
 Create/start the thread
 Consumer Thread-1: Entered
 Consumer Thread-1: Wait for the data to be produced
 Producer: 'Finding data
 Consumer Thread-2: Entered
 Consumer Thread-2: Wait for the data to be produced
 Producer: Make data shared and notify consumer
 Producer: Unlock shared data and flag
 Consumer Thread-2: Found data or notified, CONSUME IT while holding inside the monitor
 Consumer Thread-2: Wait for the data to be produced
 Producer: 'Finding data
 Producer: Make data shared and notify consumer
 Producer: Unlock shared data and flag
 Producer: 'Finding data
 Consumer Thread-2: Found data or notified, CONSUME IT while holding inside the monitor
 Consumer Thread-2: All done
 Producer: Make data shared and notify consumer
 Producer: Unlock shared data and flag
 Producer: 'Finding data
 Consumer Thread-1: Found data or notified, CONSUME IT while holding inside the monitor
 Consumer Thread-1: Wait for the data to be produced
 Producer: Make data shared and notify consumer
 Producer: Unlock shared data and flag
 Wait for the threads to complete
 Consumer Thread-1: Found data or notified, CONSUME IT while holding inside the monitor
 Consumer Thread-1: All done
 Testcase completed
*/
import java.lang.*;

/* This class is an encapsulation of the condition variable plus */
/* mutex locking logic that can be seen in the Pthread example   */
class theSharedData extends Object {
   int                     dataPresent;
   int                     sharedData;

   public theSharedData() {
      dataPresent=0;
      sharedData=0;
   }
   public synchronized void get() {
      while (dataPresent == 0) {
         try {
            System.out.print("Consumer " +
                             Thread.currentThread().getName() +
                             ": Wait for the data to be produced¥n");
            wait();
         }
          catch (InterruptedException e) {
            System.out.print("Consumer " +
                             Thread.currentThread().getName() +
                             ": wait interrupted¥n");
         }
      }
      System.out.print("Consumer " +
                       Thread.currentThread().getName() +
                       ": Found data or notified, CONSUME IT " +
                       "while holding inside the monitor¥n");
```

```
      --sharedData;
      if (sharedData == 0) {dataPresent=0;}
      /* in a real world application, the actual data would be returned */
      /* here                                                           */
   }
   public synchronized void put() {
      System.out.print("Producer: Make data shared and notify consumer¥n");
      ++sharedData;
      dataPresent=1;
      notify();
      System.out.print("Producer: Unlock shared data and flag¥n");
      /* unlock occurs when leaving the synchronized method */
   }
}


public class ATEST18 {
   public final static int NUMTHREADS = 2;
   public static theSharedData dataConditionEncapsulation = new theSharedData();

   static class theThread extends Thread {
      public void run() {
         int      retries=2;

         System.out.print("Consumer " + getName() + ": Entered¥n");
         while (retries-- != 0) {
            dataConditionEncapsulation.get();
            /* Typically an application would process the data outside */
            /* the monitor (synchronized get method here)              */
         }
         System.out.print("Consumer " + getName() + ": All done¥n");
      }
   }

   public static void main(String argv[]) {
      int            amountOfData = 4;
      theThread threads[] = new theThread[NUMTHREADS];
      System.out.print("Entered the testcase¥n");

      System.out.print("Create/start the thread¥n");
      for (int i=0; i <NUMTHREADS; ++i) {
threads[i] = new theThread();
         threads[i].start();
      }

      while (amountOfData-- != 0) {
         System.out.print("Producer: 'Finding data¥n");
         try {
            Thread.sleep(3000);
         }
          catch (InterruptedException e) {
          System.out.print("sleep interrupted¥n");
         }
         dataConditionEncapsulation.put();
      }

      System.out.print("Wait for the threads to complete¥n");
      for(int i=0; i <NUMTHREADS; ++i) {
  try {
            threads[i].join();
         }
          catch (InterruptedException e) {
         System.out.print("Join interrupted¥n");
         }
      }

      System.out.print("Testcase completed¥n");
```

```
      System.exit(0);
   }
}
```

スレッドでの条件変数の使用については、48 ページの『条件変数とスレッド』を参照してください。

同期化プリミティブとしてのスレッド

1 つのスレッドが、別のスレッドが完了するまで待機するときは、スレッド自体を同期化プリミティブとして使用することができます。ターゲットのスレッドが、すべてのアプリケーション・コードの実行を完了するまで、待機状態のスレッドは処理を行いません。他の同期化テクニックと比べると、この同期化メカニズムには少しの連携しかありません。

同期化プリミティブとして使用するスレッドには、他の同期化テクニックには見られる、所有者という概念がありません。スレッドは、ただ別のスレッドの処理が完了し、スレッドが終了するのを待つだけです。

22 ページの『スレッドを終了する』は、スレッドの終了に関する情報とコード例を記載しています。

スペース・ロケーション・ロック

スペース・ロケーション・ロックは、記憶域の任意の単一バイトに対する論理ロックを取得します。ロックにより記憶域の内容が変更されたり、アプリケーションから記憶域へのアクセスが影響されたりすることはありません。ロックは、システムが記録する単なる情報の一片にすぎません。

スペース・ロケーション・ロックは、mutex が提供するのと同じような連携ロックを提供します。ただし、スペース・ロケーション・ロックはいくつかの点で mutex と異なります。

- スペース・ロケーション・ロックは、保護対象となっているデータに直接使用することができます。スペース・ロケーション・ロックを使用するために、アプリケーションで追加オブジェクトを作成また保守する必要はありません。とはいえ、データにアクセスしているすべてのスレッドがスペース・ロケーション・ロックを使用していないと、アプリケーションは正しい結果を得られません。
- スペース・ロケーション・ロックにより、アプリケーションは、さまざまなロック要求タイプの使用を調整することができます。たとえば、スペース・ロケーション・ロックを使うと、2 つ以上のスレッドが同一データに対して共用ロックを取得することができます。
- スペース・ロケーション・ロックでは、余分のロック・タイプが提供されているため、所有者の概念が mutex とは微妙に異なります。各所有者が共用ロックを正常に取得できたのであれば、1 つの共用ロックに複数の所有者がいるということもあります。 1 つのスレッドが排他ロックを取得するには、すべての共用ロックをアンロックしなければなりません。
- アプリケーションに与えるパフォーマンス影響では、スペース・ロケーション・ロックと mutex に違いがあります。スペース・ロケーション・ロックの場合、他のスレッドとの競合を避けて 1 つのパスをロックするには、約 500 個の RISC (縮小命令セット・コンピューター) 命令が必要ですが、 mutex では約 50 個の RISC 命令で済みます。ただし、スペース・ロケーション・ロックには作成や削除についてのコストはありませんが、mutex の場合、作成や削除には約 1000 個の RISC 命令が必要です。

注: Java にはスペース・ロケーション・ロックを直接使用する機能がありません。スペース・ロケーション・ロックを使用するには、ポインターの使用が必要だからです。

プログラム・サンプルとしては、54 ページの『例: スペース・ロケーション・ロック (Pthread プログラム)』を参照してください。

例: スペース・ロケーション・ロック (Pthread プログラム)

次の例は、スペース・ロケーション・ロック同期化プリミティブを使用して、整数を動的に初期化する Pthread プログラムを示しています。

```
/*
Filename: ATEST19.QCSRC
The output of this example is as follows:
 Enter Testcase - LIBRARY/ATEST19
 Hold Lock to prevent access to shared data
 Create/start threads
 Thread 00000000 00000025: Entered
 Wait a bit until we are 'done' with the shared data
 Thread 00000000 00000026: Entered
 Thread 00000000 00000027: Entered
 Unlock shared data
 Wait for the threads to complete, and release their resources
 Thread 00000000 00000025: Start critical section, holding lock
 Thread 00000000 00000025: End critical section, release lock
 Thread 00000000 00000026: Start critical section, holding lock
 Thread 00000000 00000026: End critical section, release lock
 Thread 00000000 00000027: Start critical section, holding lock
 Thread 00000000 00000027: End critical section, release lock
 Main completed
*/
#define  _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <mih/milckcom.h>     /* Lock types           */
#include <mih/locksl.h>       /* LOCKSL instruction   */
#include <mih/unlocksl.h>     /* UNLOCKSL instruction */

#define checkResults(string, val) {             ¥
 if (val) {                                     ¥
   printf("Failed with %d at %s", val, string); ¥
   exit(1);                                     ¥
 }                                              ¥
}

#define                 NUMTHREADS   3
int                     sharedData=0;
int                     sharedData2=0;

void *theThread(void *parm)
{
   int   rc;
   printf("Thread %.8x %.8x: Entered¥n", pthread_getthreadid_np());
   locksl(&sharedData, _LENR_LOCK);   /* Lock Exclusive, No Read */
   /********** Critical Section ******************/
   printf("Thread %.8x %.8x: Start critical section, holding lock¥n",
          pthread_getthreadid_np());
   /* Access to shared data goes here */
   ++sharedData; --sharedData2;
   printf("Thread %.8x %.8x: End critical section, release lock¥n",
          pthread_getthreadid_np());
    unlocksl(&sharedData, _LENR_LOCK); /* Unlock Exclusive, No Read */
   /********** Critical Section ******************/
   return NULL;
}

int main(int argc, char **argv)

{
  pthread_t              thread[NUMTHREADS];
  int                   rc=0;
```

```
  int                    i;

  printf("Enter Testcase - %s¥n", argv[0]);

  printf("Hold Lock to prevent access to shared data¥n");
  locksl(&sharedData, _LENR_LOCK);   /* Lock Exclusive, No Read */

  printf("Create/start threads¥n");
  for (i=0; i <NUMTHREADS; ++i) {
     rc = pthread_create(&thread[i], NULL, theThread, NULL);
     checkResults("pthread_create()¥n", rc);
  }

  printf("Wait a bit until we are 'done' with the shared data¥n");
  sleep(3);
  printf("Unlock shared data¥n");
  unlocksl(&sharedData, _LENR_LOCK); /* Unlock Exclusive, No Read */

  printf("Wait for the threads to complete, and release their resources¥n");
  for (i=0; i <NUMTHREADS; ++i) {
  rc = pthread_join(thread[i], NULL);
    checkResults("pthread_join()¥n", rc);
  }

  printf("Main completed¥n");
  return 0;
}
```

このコーディング例に関連した情報については、以下を参照してください。

53 ページの『スペース・ロケーション・ロック』
56 ページの『一回限りの初期化とスレッド・セーフティ』

オブジェクト・ロック

オブジェクト・ロックを使用すれば、特定のシステムまたはアプリケーション・オブジェクトに対してロックを取得することができます。場合によっては、ユーザーが特定のオブジェクトに対して実行するアクションに対して、システムがオブジェクト・ロックを取得することもあります。システムは、いくつかのアクションに対してオブジェクト・ロックを認め、施行します。

オブジェクト・ロックを取得するとき、ロックをスレッド内で有効にしたり (スレッド有効範囲)、プロセス内で有効にしたり (プロセス有効範囲) することができます。同じプロセス内の 2 つのスレッドそれぞれが、1 つのシステム・オブジェクトに対するプロセス有効範囲ロックの取得を試みると、どちらのスレッドにもロックが与えられます。それらが同じプロセスに含まれているなら、一方のスレッドが、他方のスレッドがロックを取得するのを阻止することはありません。

オブジェクト・ロックを使って、同じプロセス内の 2 つのスレッドが、あるオブジェクトにアクセスできないようにするには、スレッド有効範囲のオブジェクト・ロックを使用する必要があります。スレッド有効範囲のオブジェクト・ロックは、同一プロセスにより取得されたプロセス有効範囲のオブジェクト・ロックと競合しません。

オブジェクト・ロックにより、アプリケーションは、さまざまなロック要求タイプの使用を調整することができます。 2 つ以上のスレッドが、同じシステム・オブジェクトに対して共用のスレッド有効範囲ロックを取得することができます。アプリケーションは、スペース・ロックの場合と同じような方法でさまざまなタイプのオブジェクト・ロックを取得することができます。このように、2 つ以上のスレッドが 1 つのターゲットに対してスレッド有効範囲の共用ロックを取得することができます。

オブジェクト・ロックでは余分のロック・タイプが提供されているため、所有者の概念が mutex とは微妙に異なります。スレッド有効範囲の共用ロックには、複数の所有者がいるということもあります。各所有者がスレッド有効範囲の共用ロックを正常に取得できたのであれば、 1 つのスレッドがスレッド有効範囲の排他ロックを取得するには、すべての共用ロックをアンロックしなければなりません。

Compare and Swap (比較とスワップ)

マシン・インターフェース (MI) の Compare and Swap (CMPSWP) 命令を使って、マルチスレッド・プログラムのデータにアクセスすることができます。 CMPSWP は、最初の比較オペランド値と 2 番目の比較オペランド値を比較します。それらが同等であれば、スワップ・オペランドが 2 番目の比較オペランドの位置に格納されます。同等でなければ、2 番目の比較オペランドが 1 番目の比較オペランドの位置に格納されます。

比較が等価ならば、2 番目の比較オペランドが比較のために取り出されてから、スワップ・オペランドが 2 番目の比較オペランドの位置に格納されるまで、別の CMPSWP 命令によるアクセスはできなくなります。

比較が等価でないならば、1 番目の比較オペランド位置への格納と、他の CMPSWP 命令によるアクセスについて原子性の保証はありません。このように、2 番目の比較オペランドだけは、並行処理制御で共用する変数でなければなりません。

次の例は、整数変数を原子的に増分または減分するために使用できる C マクロのコーディング例です。

```
#ifndef __cmpswp_h
    #inclue <mih/cmpswp.h>
#endif

#define ATOMICADD ( var, val, rc ) { ¥
    int aatemp1 = (var); ¥
    int aatemp2  = aatemp1 + val; ¥
    while( ! _CMPSWP( &aatemp1, &var, aatemp2 ) ) ¥
       aatemp2 = aatemp1 + val; ¥
    rc = aatemp2; ¥
}
```

ここで、var は増分または減分する整数、val は var に追加または減算する整数値、rc は結果の値を表します。

一回限りの初期化とスレッド・セーフティ

状況によっては、リソースの初期化を、スレッドがそのリソースを必要とするまで遅らせる必要があるかもしれません。ただし、アプリケーションによっては複数のスレッドが特定のリソースを使用する必要があるかもしれず、その場合はリソースをスレッド・セーフな方法で一回のみ初期化しなければなりません。

スレッド・セーフな方法で、複数回使用されるリソースを初期化する方法は、いくつかあります。そのほとんどの方法は、必要な初期化が完了したかどうかを、アプリケーションがすばやく判別できるようブール値を使用します。ブール・フラグに加えて同期化テクニックも使用し、初期化が完了したことを確認します。

プログラム・サンプルとしては、以下の資料を参考にしてください。
- 57 ページの『例: 一回限りの初期化 (Pthread プログラム)』

初期化およびスレッド・セーフティの追加情報については、以下を参照してください。
- 58 ページの『スレッド固有のデータ』
- 62 ページの『スレッド・セーフでない関数呼び出し』

- 64 ページの『一般的なマルチスレッド・プログラミング・エラー』

例: 一回限りの初期化 (Pthread プログラム)

次の例は、Pthread の一回限りの初期化サポートを使って、整数を動的に初期化する Pthread プログラムを示しています。

```
/*
Filename: ATEST20.QCSRC
The output of this example is as follows:
 Enter Testcase - LIBRARY/ATEST20
 Create/start threads
 Wait for the threads to complete, and release their resources
 Thread 00000000 00000007: Entered
 Thread 00000000 00000000: INITIALIZE RESOURCE
 Thread 00000000 00000007: The resource is 42
 Thread 00000000 00000006: Entered
 Thread 00000000 00000009: Entered
 Thread 00000000 00000008: Entered
 Thread 00000000 0000000a: Entered
 Thread 00000000 00000006: The resource is 42
 Thread 00000000 0000000a: The resource is 42
 Thread 00000000 00000009: The resource is 42
 Thread 00000000 00000008: The resource is 42
 Main completed
*/
#define  _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>t
#include <unistd.h>

#define checkResults(string, val) {             ¥
 if (val) {                                     ¥
   printf("Failed with %d at %s", val, string); ¥
   exit(1);                                     ¥
 }                                              ¥
}

#define                 NUMTHREADS   5
pthread_once_t          oneTimeInit = PTHREAD_ONCE_INIT;
int                     initialized = 0;
int                     resource    = 0;

void initFunction(void)
{
   printf("Thread %.8x %.8x: INITIALIZE RESOURCE¥n");
   resource = 42;
   /* Ensure that all initialization is complete and flushed */
   /* to storage before turning on this boolean flag         */
   /* Perhaps call a function or register an exception       */
   /* that causes an optimization boundary                   */
   initialized = 1;
}

void *theThread(void *parm)
{
   int   rc;
   printf("Thread %.8x %.8x: Entered¥n", pthread_getthreadid_np());
   if (!initialized) {
      rc = pthread_once(&oneTimeInit, initFunction);
      checkResults("pthread_once()¥n", rc);
   }
   printf("Thread %.8x %.8x: The resource is %d¥n",
          pthread_getthreadid_np(), resource);
   return NULL;
```

```
}

int main(int argc, char **argv)

{
  pthread_t              thread[NUMTHREADS];
  int                   rc=0;
  int                    i;

  printf("Enter Testcase - %s¥n", argv[0]);

  printf("Create/start threads¥n");
  for (i=0; i <NUMTHREADS; ++i) {
   rc = pthread_create(&thread[i], NULL, theThread, NULL);
   checkResults("pthread_create()¥n", rc);
  }

  printf("Wait for the threads to complete, and release their resources¥n");
  for (i=0; i <NUMTHREADS; ++i) {
  rc = pthread_join(thread[i], NULL);
     checkResults("pthread_join()¥n", rc);
  }

  printf("Main completed¥n");
  return 0;
}
```

一回限りの初期化、またスレッド・セーフな初期化については、 56 ページの『一回限りの初期化とスレッド・セーフティ』を参照してください。

スレッド固有のデータ

通常、スレッド化されていないアプリケーションは、グローバル記憶域を使用します。マルチスレッド・アプリケーションで実行するアプリケーション、またはアプリケーション・サービスを変更する場合、同期化テクニックを使って、グローバル記憶域が複数のスレッドにより、同時に変更されることがないようにしなければなりません。スレッド固有データは、1 つのスレッドが、他のスレッドから隠されているグローバル記憶域を保守できるようにします。

アプリケーションの設計によっては、多数のスレッドがアプリケーションのグローバル記憶域を共用している場合に、スレッドが正しく機能しないことがあります。グローバル記憶域の除去が可能でない場合は、スレッド固有データの使用を検討してください。

例として、クライアントに関する情報と現行トランザクションを、グローバル記憶域に格納するサーバーについて考慮してみましょう。このサーバーは、かなりの再設計を施さないかぎり、マルチスレッド環境でクライアント情報を共用することはできません。アプリケーションは、グローバル・クライアント情報を使用する代わりに、関数から関数へとクライアント情報を渡すことができます。

ただし、アプリケーションにとっては、クライアントおよびトランザクション情報をスレッド固有データとして保守した方が、それらの情報を変更してグローバル記憶域の使用を除去するより簡単です。新しいスレッドの作成時に、スレッドは、グローバル識別コード (またはキー) を使ってスレッド固有データを作成また格納します。こうして各クライアント (スレッド) は、固有かつグローバルなクライアント・データを持つようになります。

加えて、いくつかのアプリケーション・プログラミング・インターフェース (API) セットは、システムがデータ・デストラクター関数を自動的に呼び出す方法を提供しています。この関数は、スレッドの終了時にスレッド固有データをクリーンアップします。

プログラム・サンプルとしては、以下の資料を参考にしてください。
- 『例: スレッド固有データ (Pthread プログラム)』
- 60 ページの『例: スレッド固有データ (Java プログラム)』

例: スレッド固有データ (Pthread プログラム)

次の例は、スレッド固有データを作成する Pthread プログラムを示しています。

```
/*
Filename: ATEST22.QCSRC
The output of this example is as follows:
 Enter Testcase - LIBRARY/ATEST22
 Create/start threads
 Wait for the threads to complete, and release their resources
 Thread 00000000 00000036: Entered
 Thread 00000000 00000036: foo(), threadSpecific data=0 2
 Thread 00000000 00000036: bar(), threadSpecific data=0 2
 Thread 00000000 00000036: Free data
 Thread 00000000 00000037: Entered
 Thread 00000000 00000037: foo(), threadSpecific data=1 4
 Thread 00000000 00000037: bar(), threadSpecific data=1 4
 Thread 00000000 00000037: Free data
 Main completed
*/
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void foo(void);  /* Functions that use the threadSpecific data */
void bar(void);
void dataDestructor(void *data);

#define checkResults(string, val) {             ¥
 if (val) {                                     ¥
   printf("Failed with %d at %s", val, string); ¥
   exit(1);                                     ¥
 }                                              ¥
}

typedef struct {
  int          threadSpecific1;
  int          threadSpecific2;
} threadSpecific_data_t;

#define                 NUMTHREADS   2
pthread_key_t           threadSpecificKey;


void *theThread(void *parm)
{
   int               rc;
   threadSpecific_data_t    *gData;
   printf("Thread %.8x %.8x: Entered¥n", pthread_getthreadid_np());
   gData = (threadSpecific_data_t *)parm;
   rc = pthread_setspecific(threadSpecificKey, gData);
   checkResults("pthread_setspecific()¥n", rc);
   foo();
   return NULL;
}

void foo() {
   threadSpecific_data_t *gData = pthread_getspecific(threadSpecificKey);
   printf("Thread %.8x %.8x: foo(), threadSpecific data=%d %d¥n",
          pthread_getthreadid_np(), gData->threadSpecific1, gData->threadSpecific2);
```

```
   bar();
}

void bar() {
   threadSpecific_data_t *gData = pthread_getspecific(threadSpecificKey);
   printf("Thread %.8x %.8x: bar(), threadSpecific data=%d %d¥n",
          pthread_getthreadid_np(), gData->threadSpecific1, gData->threadSpecific2);
   return;
}

void dataDestructor(void *data) {
   printf("Thread %.8x %.8x: Free data¥n", pthread_getthreadid_np());
   pthread_setspecific(threadSpecificKey, NULL);
   free(data);
}


int main(int argc, char **argv)
{
  pthread_t             thread[NUMTHREADS];
  int                   rc=0;
  int                   i;
  threadSpecific_data_t        *gData;

  printf("Enter Testcase - %s¥n", argv[0]);
  rc = pthread_key_create(&threadSpecificKey, dataDestructor);
  checkResults("pthread_key_create()¥n", rc);

  printf("Create/start threads¥n");
  for (i=0; i <NUMTHREADS; ++i) {
       /* Create per-thread threadSpecific data and pass it to the thread */
     gData = (threadSpecific_data_t *)malloc(sizeof(threadSpecific_data_t));
     gData->threadSpecific1 = i;
     gData->threadSpecific2 = (i+1)*2;
     rc = pthread_create(&thread[i], NULL, theThread, gData);
     checkResults("pthread_create()¥n", rc);
  }

  printf("Wait for the threads to complete, and release their resources¥n");
  for (i=0; i <NUMTHREADS; ++i) {
  rc = pthread_join(thread[i], NULL);
    checkResults("pthread_join()¥n", rc);
  }

  pthread_key_delete(threadSpecificKey);
  printf("Main completed¥n");
  return 0;
}
```

マルチスレッド・アプリケーションで実行するように変更されたアプリケーションが、複数のスレッドによる同時の変更からグローバル記憶域を保護できるよう同期化テクニックを使うことに関しては、58 ページの『スレッド固有のデータ』を参照してください。

例: スレッド固有データ (Java プログラム)

次の例は、スレッド固有データを作成する Java プログラムを示しています。 Java スレッドがオブジェクトに作成されているため、スレッド固有データの使用は透過的です。 Java はガーベッジ・コレクションを行う言語です。データ・デストラクターまたは他のクリーンアップ・アクションがないことに注目してください。

```
/*
FileName: ATEST22.java
The output of this example is as follows:
```

```
 Entered the testcase
 Create/start threads
 Thread Thread-1: Entered
 Thread Thread-1: foo(), threadSpecific data=0 2
 Thread Thread-1: bar(), threadSpecific data=0 2
 Wait for the threads to complete
 Thread Thread-2: Entered
 Thread Thread-2: foo(), threadSpecific data=1 4
 Thread Thread-2: bar(), threadSpecific data=1 4
 Testcase completed
*/
import java.lang.*;


public class ATEST22 {
   public final static int NUMTHREADS = 2;

   static class theThread extends Thread {
      int               threadSpecific1;
      int               threadSpecific2;

      public theThread(int i, int i2) {
         threadSpecific1 = i;
         threadSpecific2 = i2;
      }
      public void run() {
            System.out.print("Thread " + getName() +
                        ": Entered¥n");
         foo();
         return;
      }
      void foo() {
         System.out.print("Thread " + getName() +
                        ": foo(), threadSpecific data=" +
                        String.valueOf(threadSpecific1) + " " +
                        String.valueOf(threadSpecific2) + "¥n");
         bar();
      }
      void bar() {
         System.out.print("Thread " + getName() +
                        ": bar(), threadSpecific data=" +
                        String.valueOf(threadSpecific1) + " " +
                        String.valueOf(threadSpecific2) + "¥n");
      }
   }

   public static void main(String argv[]) {
      System.out.print("Entered the testcase¥n");

      System.out.print("Create/start threads¥n");
      theThread threads[] = new theThread[NUMTHREADS];
      for (int i=0; i <NUMTHREADS; ++i) {
 threads[i] = new theThread(i, (i+1)*2);
         threads[i].start();
      }

      System.out.print("Wait for the threads to complete¥n");
      for (int i=0; i <NUMTHREADS; ++i) {
  try {
            threads[i].join();
         }
          catch (InterruptedException e) {
         System.out.print("Join interrupted¥n");
         }
      }
      System.out.print("Testcase completed¥nj");
      System.exit(0);
```

```
    }
}
```

マルチスレッド・アプリケーションで実行するように変更されたアプリケーションが、複数のスレッドによる同時の変更からグローバル記憶域を保護できるよう同期化テクニックを使うことに関しては、58 ページの『スレッド固有のデータ』を参照してください。

スレッド・セーフでない関数呼び出し

時折、マルチスレッド・アプリケーションが、スレッド・セーフではない関数やシステム・サービスへのアクセスを必要とすることがあります。それらの関数を呼び出す、安全な代替方法がいくつかあります。

代替方法を理解するために、アプリケーション・プログラミング・インターフェース (API) foo() を呼び出すプログラムを例として用います。関数 foo() はスレッド・セーフではないものとしてリストされているため、安全な呼び出し方法を見いださなければなりません。一般的な 2 つのオプションは、グローバルな mutex (相互排他) を使用する方法と、スレッド・セーフではない関数を呼び出す別個のジョブを使用する方法です。

スレッド・セーフティの追加情報については、以下を参照してください。
- 『スレッド・セーフでない関数を呼び出すグローバル mutex の使用』
- 63 ページの『スレッド・セーフでない関数を呼び出す個別ジョブの使用』
- 64 ページの『一般的なマルチスレッド・プログラミング・エラー』

スレッド・セーフでない関数を呼び出すグローバル mutex の使用

スレッド・セーフではない関数を呼び出すときに、考慮しなければならない代替策の 1 つは、グローバル mutex を使ってそれらの関数を実行する方法です。 foo() を呼び出すときに、FOO MUTEX と呼ばれる mutex (相互排他) のロックを試行することができます。この方法は foo() の内部部分についてすべて理解している場合は成功しますが、一般的なオペレーティング・システム関数の場合、安全性として完全ではありません。

たとえば、アプリケーション・プログラミング・インターフェース (API) foo() はスレッド・セーフではありません。なぜなら foo() は API bar() を呼び出し、bar() は threadUnsafeFoo() を呼び出すからです。threadUnsafeFoo() 関数が特定のストレージおよび制御ブロックを使う方法は、安全ではありません。この例では、ストレージおよび制御ブロックを DATA1 と呼びます。

API threadUnsafeFoo() は安全でないため、 foo() を使うアプリケーションやシステム・サービスはどれもスレッド・セーフではありません。それを複数のスレッドに対して使用すると、データが破壊されたり、未定義の結果が出されることがあるからです。

例のアプリケーションも、wiffle() と呼ばれるスレッド・セーフ API またはシステム・サービスを使用します。 wiffle() は waffle() を呼び出しますが、 waffle() はスレッド・セーフである方法でストレージおよび制御ブロック DATA1 にアクセスします。 (つまり、waffle() と、他の API から成るグループは内部 DATA1 MUTEX を使用します。)

例のアプリケーションは、foo() と wiffle() システム・サービス間の基礎的な DATA1 接続を把握していません。そのため、アプリケーションは wiffle() を呼び出す前に FOO MUTEX をロックしません。 FOO MUTEX を保留にしている foo() への呼び出しが wiffle() への呼び出しと同時にあったときに、アプリケーションがスレッド・セーフでない関数 foo() を使用しようとすると、DATA1 とそれが表している実体は破壊されます。

このシナリオも複雑ですが、実際にはより複雑です。関係しているすべての関数の内部詳細がわからないまま障害を解決するようなことがあるからです。

独自で開発したシリアル化により、オペレーティング・システムのサービスや関数をスレッド・セーフにすることは試行しないでください。

62 ページの『スレッド・セーフでない関数呼び出し』へ戻ってください。

スレッド・セーフでない関数を呼び出す個別ジョブの使用

新しいジョブを実行依頼して必要な処理を完了するための、スレッド・セーフなメカニズムがいくつかあります。

- アプリケーションがスレッド・セーフではない CL コマンドを使用する場合は、アプリケーション・プログラミング・インターフェース (API) Qp0zSystem() を使用することができます。この API は C 言語の system() 関数と似ていますが、違う点として、 CL コマンドを実行する前に新しいプロセスを開始します。また実行した CL コマンドの結果についての情報も戻します。 Qp0zSystem() の詳細については、System API Programming Reference を参照してください。
- アプリケーションがスレッド・セーフではない、API またはプログラムを呼び出す場合、spawn() API を使って新しいジョブを開始することができます。 spawn() API を使用する場合、アプリケーション環境のどの部分を子プロセスに複写するかをアプリケーションで決定する必要があります。 spawn() API では、開かれた IFS ファイル、ソケット、または他のリソースを親から子プロセスに継承することができます。 spawn() の詳細については、System API Programming Reference を参照してください。
- アプリケーションがスレッド・セーフではない複数の関数を、頻繁に使用する場合があります。その場合は、上に記述したメカニズムの 1 つを使って新しいジョブを開始してください。その新しいジョブは、アプリケーションから要求が出されたときに、スレッド・セーフではないコードを実行するサーバーとして、機能することができます。アプリケーションは、メッセージ待ち行列、データ待ち行列、またはセマフォーを使いながら、サーバーと通信することができます。

これらのジョブ・メカニズムのいずれかを使用する場合、異なるジョブを使ってアプリケーション処理を完了することに伴う、いくつかの考慮事項に注意しなければなりません。以下のとおりです。

入出力に関する考慮事項:

アプリケーションの入出力処理と対立することなく、サーバー・ジョブが通常の入出力を実行することはできません。アプリケーションの入出力処理は、システム・オブジェクトで保守されているファイル・オフセットまたはロックに影響することがあります。入出力の試行が、ジョブ間で競合するかもしれません。

パラメーターの指定:

スレッド・セーフではない関数に、ポインターまたはその他のタイプの複合パラメーターを渡すのは、それほど容易ではありません。この関数は、別個のジョブで実行されます。その関数に複合データを渡すときに生じるかもしれない問題を解決する必要があります。

関数呼び出しの戻り:

スレッド・セーフではない関数の結果は、上記のメカニズムで生じる、成功 / 失敗などの単純な情報より複雑になることがあります。パラメーターを渡す場合と同じような問題があります。

ジョブ属性:

アプリケーション環境をサーバー・ジョブで複写しようとすると、さらに複雑な問題が生じることがあります。たとえば、ユーザー ALICE としてアプリケーションを起動し、ユーザー BOB としてサーバー・ジョブを開始すると、アプリケーションは正しく動作しません。

62 ページの『スレッド・セーフでない関数呼び出し』へ戻ってください。

一般的なマルチスレッド・プログラミング・エラー

マルチスレッド・アプリケーションの作成時に頻繁に発生するプログラミング・エラーがいくつかあります。次のとおりです。

- 『スレッド・セーフでない関数の呼び出し』
- 『スレッド作成が許可されていないために生じる障害』
- 65 ページの『活動化グループの終了』
- 65 ページの『スレッド・モデルの混合』
- 66 ページの『マルチスレッド・プログラムでのコミット操作』
- 66 ページの『データベース・レコードの入出力とスレッド・セーフティ』

Pthread エラーのトラブルシューティングは、一般的な Pthread API エラーについて説明します。

71 ページの『第 8 章 マルチスレッド・ジョブのデバッグとサービス』は、マルチスレッド・プログラムのデバッグを実行するための、基本的なテクニックについて説明します。

スレッド・セーフでない関数の呼び出し

スレッド・アプリケーションの作成時に最も頻繁に起きるプログラミング・エラーは、スレッド・セーフでないアプリケーション・プログラミング・インターフェース (API) またはシステム・サービスの使用です。アプリケーションは、呼び出す各 API について、またそれがプロバイダーによりスレッド・セーフと認定されているかどうかを把握していなければなりません。その API またはシステム・サービスがスレッド・セーフであれば、それを実装するときは、スレッド・セーフである他の API またはシステム・サービスだけを使用します。

このことは、特にアプリケーションから、制御下にないユーザー作成コードを呼び出すときに問題になります。スレッド・セーフティのためにこのコードを妥当性検査することはできません。ユーザー作成コードがスレッド・セーフでない場合、それをアプリケーションの独自のプロセス・コンテキストで呼び出すと、アプリケーション・データが破壊されたり、アプリケーションが停止する可能性があります。

スレッド作成が許可されていないために生じる障害

OS/400 では、システム上のすべてのプロセスが、実行中いつでもスレッドを作成できるわけではありません。ジョブの作成時に特別なパラメーターを指定することができます。これらのパラメーターは、マルチスレッドを許可するようジョブに指示します。

| たとえば、アプリケーション・プログラミング・インターフェース (API) spawn() には、継承パラメーター
| で指定できるマルチスレッド・フラグがあります。それ以外の方法で、新しいジョブにスレッドの作成を許
| 可する OS/400 ジョブを実行依頼することができます。ジョブ・タイプによってはジョブ記述を使用する
| ことができ、その中で、ジョブ記述を使って開始されたジョブにスレッドの開始を許可することができま
| す。そうするには、ジョブ記述の作成 (CRTJOBD) コマンドまたはジョブ記述の変更 (CHGJOBD) コマン
| ドの、マルチスレッド許可 (ALWMLTTHD) パラメーターを使用します。

スレッドの開始を許可されていないジョブでスレッドの作成を試みると、スレッド作成は失敗します。障害の詳細については、使用している API または言語に固有の資料を参照してください。

同じ障害は、アプリケーションが C++ 静的オブジェクトのデストラクターまたは C/C++ プログラム終了ルーチンを実行する場合も発生します。ジョブの終了時にアプリケーション・コードでスレッドを作成することはできません。

活動化グループの終了

OS/400 統合言語環境 (ILE) プログラム・モデルは、1 つのアプリケーション・プログラムのリソースを 1 つのジョブ内でカプセル化する方法として、活動化グループを使用します。

活動化グループを使用している可能性がある、プロセスで複数のスレッドが存在している場合、システムが、予測可能かつ安全な方法でその活動化グループを終了することはできません。この問題を解決するため、マルチスレッド・ジョブで活動化グループを終了する方法は変更されました。

マルチスレッド・アプリケーションが、活動化グループを終了するアクション、または活動化グループが終了するのを阻止できないアクションを実行すると、システムは、その時点でプロセスを終了します。

この典型的な例は、マルチスレッドの利点を生かすために、現行の OS/400 アプリケーションを変更したときに見られます。現行の OS/400 アプリケーションは、ほとんどの場合、複数のプログラム・オブジェクトを使用します。他のプラットフォームとは異なり、OS/400 では、現在実行中のアプリケーション・コードと同じプロセス・コンテキスト内でプログラムを呼び出すことができます。

ジョブ中のいずれかのプログラムが C 関数 exit() または abort() を呼び出すか、使用中の活動化グループを終了するように指示すると、システムはその信号を、プロセスの終了要求と解釈してしまいます。

プログラムの活動化グループ属性のデフォルト値は *NEW です。マルチスレッド・ジョブ内でデフォルトの活動値 *NEW を使ってプログラムを呼び出した場合、プログラムの終了時には活動化グループとプロセスも終了します。プログラムが終了してもジョブの活動を続行させたい場合は、*DEFAULT または特定の活動化グループを使用するよう、プログラムを変更する必要があります。

スレッド・モデルの混合

Pthread アプリケーション・プログラミング・インターフェース (API) を、システムが提供する他のスレッド管理 API と混合しないでください。たとえば、Pthread と同じスレッドまたはプロセスで、Java や IBM Open Class ライブラリー・スレッドの実装を使用しないでください。さらに重要なこととして、あるスレッド実装から、別のアプリケーション・プログラミング・インターフェース (API) 実装を使ってスレッドを操作することは試行しないでください。予期せぬ結果が生じる場合があります。

たとえば、あるスレッドの優先順位が、 Pthread インターフェースおよび非 Pthread インターフェースのどちらからでも操作できるとします。そのように操作していれば、優先順位は常に正しく設定されます。しかし、Pthread インターフェース pthread_getschedparam() から戻される優先順位が正しいのは、優先順位の設定が pthread_setschedparam() インターフェースまたは他のインターフェースのいずれか (両方ではない) を使ってなされた場合のみです。そのようにせずに、複数のインターフェースを使ってスレッドの優先順位を設定すると、 pthread_getschedparam() からは、最後に使用した pthread_setschedparam() インターフェースで設定された、優先順位が戻されます。

同じように、アプリケーション・コードをネイティブのメソッドで実行しているなら、 pthread_exit() API を使って Java スレッドを終了することができます。ただし、pthread_exit() を使って Java スレッドを終了すると、予期せぬ結果が生じる場合があります。たとえば、いくつかの Java 環境スレッドの処理をバイパスしたり、 Java アプリケーションが予期していない、また処理できない仕方で Java スレッドを終了したりしてしまう場合があります。

マルチスレッド・プログラムでのコミット操作

OS/400 でのデータベース・トランザクションは、ジョブまたは活動化グループのいずれかに限定されます。マルチスレッド・アプリケーションが複数のスレッドで、複数のデータベース・トランザクションを同時に実行する場合、 1 つのスレッドでのコミット操作により、別のスレッドの活動までコミットされることがあります。

たとえば、あるアプリケーションに、client1 用のデータベース・トランザクションを実行している 1 つのスレッドと、 client2 用のデータベース・トランザクションを実行している別のスレッドがあるとします。client1 用の処理を実行しているスレッドが処理を完了し、変更をコミットします。このコミット操作により、client2 のために加えられた変更もすべてコミットされます。そのため、アプリケーションは、進行中の未完了トランザクションをすべて把握していなければなりません。

データベースが複数のスレッドと対話する方法の詳細については、 10 ページの『マルチスレッド・プログラミングでのデータベースについての考慮事項』を参照してください。

データベース・レコードの入出力とスレッド・セーフティ

OS/400 データベース・ファイルの入出力フィードバック領域を使って、入出力操作の結果をレコード入出力ユーザーに通信することができます。通常のアプリケーションでは、フィードバック領域の特性のためにスレッド・セーフティの問題が生じるかもしれません。

入出力操作を実行する際、実行時サポートとデータベース・サポートにより入出力操作のスレッド・セーフティが保証されます。入出力が完了した時点で、実行時サポートが保持していたロックがあれば解放され、制御がアプリケーションに戻されます。

入出力フィードバック領域を検査するときに、別のスレッドがその領域を同時に使用しないよう、アクセスをシリアル化するのはアプリケーションの責任です。 COBOL と RPG はすべての入出力操作でフィードバック領域を使用します。そのため、同じ共用ファイルに対していくつかの異なる言語で入出力を実行する場合は、フィードバック領域を使用していないように思える場合でも、ファイルへのアクセスをシリアル化してください。

たとえば、Thread 1 はレコード入出力操作で fileA を使用しているとします。システムは、それらの操作に関する情報を fileA のフィードバック領域に格納します。 Thread 1 の入出力操作が完了すると、Thread 1 はフィードバック領域のフィールドを検査しながら、フィードバック領域を操作します。 Thread 2 がレコード入出力操作のために fileA を使用する時点で、フィードバック領域が保護されていない場合には、システムは同じフィードバック領域を使用します。ここで、Thread 2 がフィードバック領域を同時に使用するため、Thread 1 には矛盾した結果が戻されることになります。このようなことは、ファイルの共用がファイル・ポインターまたはファイル記述子番号を使って行われるときに生じます。ファイルが各スレッドで開かれているなら、問題は生じません。

フィードバック領域にアクセスしているときに他のアプリケーション・コードまたはシステムによりそれが変更されることのないよう、同期化テクニックを使ってフィードバック領域を保護してください。

第 7 章 言語アクセスとスレッド

OS/400 でさまざまな言語によるスレッドのサポートの違いについては、以下のリンクを参照してください。この情報をもとに、ご使用のアプリケーションでどのようにスレッドを実装すればよいかを判断できます。

以下のセクションで説明されている概念やテクニックには、OS/400 でサポートされている言語でのスレッドの使用法に関する情報が含まれています。

- 『Java 言語でのスレッドの考慮事項』
- 68 ページの『C 言語でのスレッドの考慮事項』
- 68 ページの『C++ 言語でのスレッドの考慮事項』
- 68 ページの『ILE COBOL および RPG 言語でのスレッドの考慮事項』
- 69 ページの『OPM 言語でのスレッドの考慮事項』

このセクションで説明する概念は、すべてのプログラミング言語に当てはまります。各言語でそれらの概念がどのように実現されているかについては、その特定言語のプログラマーの手引きを参照してください。

Java 言語でのスレッドの考慮事項

java.lang.Thread クラスを使えば、 OS/400 カーネル・スレッドの利点を活用することができます。 Java スレッドは、OS/400 カーネル・スレッド・モデルの上位で操作を行います。各 Java スレッドは、プロセスで実行される多くのタスクの 1 つです。「スレッドの管理」セクションにリストされているすべてのアクティビティーが実行可能です。

Java 仮想計算機 (VM) マシンは、Java ガーベッジ・コレクションなどのサービスを実行するときは、必ずいくつかのスレッドを作成します。これらのスレッドはシステムが使用するため、アプリケーションでは使用してはなりません。

Java で利用不能なシステム関数は、固有のメソッドを使ってアクセスします。固有のメソッドは *PGM オブジェクトではありません。 ILE (統合言語環境) サービス・プログラム (*SRVPGM) からエクスポートされたプロシージャーです。これらの固有メソッドは、常にマルチスレッド・プロセスで実行されるので、スレッド・セーフです。 ILE COBOL、RPG IV、CL、C および C++ コンパイラーはスレッド・セーフです。

注: 標準的な C および C++ 関数がすべてスレッド・セーフではありません。特定の関数については、言語の解説書を参照してください。

ILE プログラム (*PGM) オブジェクトを呼び出す必要があるときは、 java.lang.Runtime.exec() を使って、プログラムを実行するための別のプロセスを開始します。

exit() および **abort()** の使用には十分注意してください。これらの関数はアプリケーションを終了します。プロセス、およびプロセスで実行中のすべてのスレッドも終了します。

一般的な Java マルチスレッド操作のコーディング例は、 85 ページの『第 10 章 例: スレッド』にあります。

C 言語でのスレッドの考慮事項

すべての C ライブラリー関数がスレッド・セーフではありません。マルチスレッド対応ジョブで既存の C アプリケーションを呼び出す前に、 *WebSphere Development Studio: ILE C/C++ Programmer's Guide (SC09-2712-03)* を参照し、使用するすべての関数がスレッド・セーフかどうかを判別してください。

マルチスレッド・ジョブで呼び出す前に、既存の C アプリケーションがスレッド・セーフかどうかを判別しなければなりません。 C プログラムがスレッド・セーフでない場合のために、スレッド・セーフでないプログラムをマルチスレッド・ジョブから呼び出すテクニックがあります。

C で作成されたプログラムを使用する場合、次のアクションを使用します。

ILE C アプリケーションの再作成

マルチスレッド・ジョブで呼び出す前に、既存の ILE C アプリケーション全部を、TGTRLS(*CURRENT) を使ってコンパイルし、バインドします。

***NEW 活動化グループの除去:**

マルチスレッド・アプリケーションでは *NEW 活動化グループを使用しません。

注: ACTGRP(*NEW) は、CRTPGM および CRTBNDC コマンドのデフォルトです。

ILE C *PGM オブジェクトの呼び出し:

活動化グループ *NEW を持っている ILE C *PGM オブジェクトを呼び出す必要がある場合、別のプロセスを開始して 2 番目のプログラムを実行するか、特定の活動化グループを指定して 2 番目のプログラムを作成してください。

スレッド・セーフティの関連情報については、 64 ページの『スレッド・セーフでない関数の呼び出し』を参照してください。

C++ 言語でのスレッドの考慮事項

C++ 実行時関数 new、delete、try、catch、および throw はスレッド・セーフです。 C++ ライブラリー関数のスレッド・セーフを判別するには、 *WebSphere Development Studio: ILE C/C++ Programmer's Guide (SC09-2712-03)* および *WebSphere Development Studio: C/C++ Language Reference (SC09-4815-00)* を参照してください。

ILE COBOL および RPG 言語でのスレッドの考慮事項

モジュールへのアクセスをシリアル化することによりマルチスレッド環境で安全に実行可能な、 ILE COBOL または ILE RPG モジュールを作成することができます。そのためには、PROCESS ステートメントに THREAD(SERIALIZE) を指定するか (COBOL の場合)、または Control 指定に THREAD(*SERIALIZE) を指定します (RPG の場合)。 1 つのモジュールをシリアル化すると、一度に 1 つのスレッドだけがそのモジュールで任意のプロシージャーを実行できます。たとえば、プロシージャー P1 および P2 を持っているモジュールについて考えてみましょう。 1 つのスレッドがプロシージャー P1 を実行しているならば、そのスレッドが P1 の実行を完了するまで、他のスレッドはプロシージャー P1 または P2 のいずれも実行できません。モジュールをシリアル化したとしても、COBOL または RPG プログラマーは、グローバル・ストレージとヒープ・ストレージがスレッド・セーフな方法でアクセスされるようにしなければなりません。たとえ RPG または COBOL プロシージャーが自動ストレージしか使用して

いないとしても、 RPG および COBOL はすべてのプロシージャーで、静的ストレージ制御ブロックを使用します。そのため、マルチスレッド環境で ILE RPG または ILE COBOL を使用する場合は、いつでも THREAD(*SERIALIZE) を指定しなければなりません。

スレッド・セーフティの関連情報については、 62 ページの『スレッド・セーフでない関数呼び出し』を参照してください。

OPM 言語でのスレッドの考慮事項

オリジナル・プログラム・モデル (OPM) プログラムは、スレッド・セーフではありません。マルチスレッド・アプリケーションが呼び出す前に、OPM プログラムを ILE にマイグレーションし、スレッド・セーフにしておく必要があります。ユーザー作成の Java ネイティブ・メッセージから OPM プログラムを呼び出さないでください。マルチスレッド・アプリケーションから OPM プログラムを呼び出す必要がある場合は、OPM プログラムを実行する別のプロセスを開始してください。

第 8 章 マルチスレッド・ジョブのデバッグとサービス

マルチスレッド・ジョブのデバッグとサービスについては、以下のリンク先に記載されている概念とテクニックを参照してください。

- 『スレッド関連データを報告するコマンド』
- 72 ページの『フライト・レコーダー』
- 76 ページの『スレッド情報を表示するためのオプション』
- 77 ページの『マルチスレッド・ジョブのデバッグ』
- 77 ページの『マルチスレッド・アプリケーションのテスト分野』

このセクションで説明する概念は、すべてのプログラミング言語に当てはまります。 各言語でそれらの概念がどのように実現されているかについては、その特定言語のプログラマーの手引きを参照してください。

64 ページの『一般的なマルチスレッド・プログラミング・エラー』は、共通して発生するマルチスレッド・プログラミングのエラーを記載しています。

スレッド関連データを報告するコマンド

ジョブのサービスのために使われるコマンドの多くは、マルチスレッド・プロセスをサポートするよう変更されていません。そのため、既存のコマンドは、ジョブ内の個々のスレッドに対してではなく、これまでと同じようにジョブに対して実行されます。ただし、以下のコマンドは、スレッド関連データを報告するために拡張されました。

ジョブのダンプ (DMPJOB) コマンド:

ジョブのダンプ・コマンドは、1 つのジョブ内のすべてのスレッドをダンプするようになりました。スレッド関連データは、コマンドの JOBTHD パラメーターにより使用可能になります。次の例は、スレッド関連データが含まれるマルチスレッド・ジョブ・ダンプを取得する方法を示しています。

```
STRSRVJOB JOB(000000/USER/JOBNAME)
DMPJOB PGM(*ALL) JOBAREA(*NONE) ADROBJ(*NO) JOBTHD(*YES)
ENDSRVJOB
```

ジョブのトレース (TRCJOB) コマンド:

ジョブのトレース・コマンドは、1 つのジョブ内のすべてのスレッドをトレースするようになりました。各トレース・レコードには、どのスレッドがレコードをログに記録したかを示すためのスレッド ID が付けられます。次の例は、マルチスレッド・ジョブのトレースを取得する方法を示しています。

```
STRSRVJOB JOB(000000/USER/JOBNAME)
    TRCJOB SET(*ON)
    ... tracing the serviced job
    TRCJOB SET(*OFF)
    ENDSRVJOB
```

第 1 障害データ検知 (FFDC) 関数:

FFDC は、プログラム診断依頼書 (APAR) に対して問題を報告するために、プログラム、サービス・プログラム、またはモジュールで使用できる関数です。 FFDC は問題をログに記録し、症状ストリングを構築し、問題分析データを収集します。

マルチスレッド・プログラムで使用する場合には、FFDC は完全には機能しません。マルチスレッド・プログラムで呼び出すと、FFDC のサポートは部分的になります。以下のサポートが含まれます。

- データ項目情報の収集、およびこの情報の QPSRVDMP スプール・ファイルへの格納。
- 障害ポイント、症状ストリング、および検出 / 懐疑されたプログラム情報の QPSRVDMP スプール・ファイルへの格納。ジョブ・ログにもこの情報が含まれます。
- ジョブのダンプ (**DMPJOB**) コマンドの開始。

オブジェクトのダンプ、問題項目の作成、または問題判別用のデータ収集に関する FFDC サポートは、マルチスレッド・ジョブでは提供されていません。

フライト・レコーダー

マルチスレッド・アプリケーションをデバッグするには、データをフライト・レコーダーに書き込むのが便利な方法です。フライト・レコーダーとは、アプリケーション内の問題を追跡できるようトレース情報を書き込むためのファイル、出力バッファー、またはその他のオブジェクトを指します。関数への入り口点と関数からの出口点は、フライト・レコーダーにトレースされる典型的な情報です。関数へのパラメーター、作業流れの大きな変更、またエラー条件なども頻繁に記録されます。

Pthread ライブラリーは、アプリケーションが問題をトレースするための手段を提供しています。トレースでは、任意選択トレース・ポイントを切り取り、レコーダーのオン / オフを切り替え、アプリケーションを再コンパイルする、つまりすべてのトレース・コードを除去することができます。

さまざまなレベルのトレースを使用することができます。アプリケーションがトレース・レベルを認める場合、該当するトレース・レベルでトレース・ポイントのキューを出したり、トレース・レベルを管理したりするのはアプリケーションの責任です。 Pthread ライブラリーは、そのためのアプリケーション・プログラミング・インターフェース (API)、マクロ、およびトレース・レベル変数を用意しています。詳細については、73 ページの『例: フライト・レコーダー出力例 (Pthread プログラム)』を参照してください。

Pthread トレースの場合、エラー・レベル・トレースには、エラー・レベル重大度のトレース・ポイントだけが表示されます。情報レベル・トレースでは、情報トレース・ポイントに加えて、すべてのエラー・レベル・トレース・ポイントがトレースされます。冗長レベル・トレース・ポイントでは、すべてのレベルがトレースされます。各トレース・ポイントには、スレッド ID、ミリ秒単位のタイム・スタンプ、およびトレース・データが自動的に含まれます。

| トレースを使用可能にする CL コマンド、表示する CL コマンド、および処理する CL コマンドが
| OS/400 の一部として組み込まれています。

トレース・ポイント用のトレース・バッファーは、ユーザー・スペース・オブジェクトとして QUSRSYS ライブラリーに作成されます。 トレース・バッファーには QP0Zxxxxxx という名前が付けられています。ここで xxxxxx は、トレースを実行したジョブの 6 桁ジョブ番号を表します。

トレースに関連した API は次のとおりです。詳細については、UNIX 版 API の資料を参照してください。

- Qp0zUprintf - フォーマット済みのトレース・データを印刷する
- Qp0zDump - フォーマット済みの 16 進データをダンプする

| • Qp0zDumpStack - スレッド呼び出しの呼び出しスタックをダンプする
| • Qp0zDumpTargetStack - ターゲット・スレッドの呼び出しスタックをダンプする
• Qp0zLprintf - フォーマット済みのジョブ・ログ・メッセージを印刷する

トレースを操作するための CL コマンドは次のとおりです。

• DMPUSRTRC - 指定されたジョブのトレース内容をダンプする
• CHGUSRTRC - 指定されたジョブのトレース属性 (サイズ、折り返し、クリア) を変更する
• DLTUSRTRC - ジョブのトレースに関連付けられた持続トレース・オブジェクトを削除する

プログラム・サンプルとしては、『例: フライト・レコーダー出力例 (Pthread プログラム)』を参照してください。

例: フライト・レコーダー出力例 (Pthread プログラム)

次の例は、システム提供のフライト・レコーダーまたはトレース・インターフェースを使用する Pthread プログラムを示しています。

```
/*
Filename: ATEST23.QCSRC
Use CL command DMPUSRTRC to output the following tracing
information that this example traces.

This information is put into a file QTEMP/QAP0ZDMP or to
standard output.

The trace records are indented and labeled based on thread id,
and millisecond timestamp of the time the tracepoint was cut.

The following trace output occurs when the optional parameter
'PTHREAD_TRACING' is NOT specified when calling this program.

If the optional parameter 'PTHREAD_TRACING' is specified, many
more tracepoints describing pthread library processing will occur.

Use the Pthread library tracepoints to debug incorrect calls to the
Pthreads library from your application.

Trace output ---------
User Trace Dump for job 096932/MYLIB/PTHREADT. Size: 300K, Wrapped 0 times.

--- 11/06/1998 11:06:57 ---
      0000000D:133520 Create/start a thread
      0000000D:293104 Wait for the thread to complete, and release their resources
       0000000E:294072 Thread Entered
       0000000E:294272 DB51A4C80A:001CD0 L:0008 Global Data
       0000000E:294416  DB51A4C80A:001CD0  00000000 00000002                      *.................*
       0000000E:294496 foo(), threadSpecific data=0 2
       0000000E:294568 bar(), threadSpecific data=0 2
       0000000E:294624 bar(): This is an error tracepoint
       0000000E:294680 Stack Dump For Current Thread
       0000000E:294736 Stack:  This thread's stack at time of error in bar()
       0000000E:333872 Stack:  Library    / Program     Module      Stmt    Procedure
       0000000E:367488 Stack:  QSYS       / QLESPI      QLECRTTH    774   : LE_Create_Thread2__FP12crtth_parm_t
       0000000E:371704 Stack:  QSYS       / QP0WPTHR    QP0WPTHR    1008  : pthread_create_part2
       0000000E:371872 Stack:  MYLIB      / PTHREADT    PTHREADT    19    : theThread__FPv
       0000000E:371944 Stack:  MYLIB      / PTHREADT    PTHREADT    29    : foo__Fv
       0000000E:372016 Stack:  MYLIB      / PTHREADT    PTHREADT    46    : bar__Fv
       0000000E:372104 Stack:  QSYS       / QP0ZCPA     QP0ZUDBG    87    : Qp0zDumpStack
       0000000E:379248 Stack:  QSYS       / QP0ZSCPA    QP0ZSCPA    276   : Qp0zSUDumpStack
       0000000E:379400 Stack:  QSYS       / QP0ZSCPA    QP0ZSCPA    287   : Qp0zSUDumpTargetStack
       0000000E:379440 Stack:  Completed
       0000000E:379560 foo(): This is an error tracepoint
       0000000E:379656 dataDestructor: Free data
      0000000D:413816 Create/start a thread
      0000000D:414408 Wait for the thread to complete, and release their resources
```

```
        0000000F:415672 Thread Entered
        0000000F:415872 DB51A4C80A:001CD0 L:0008 Global Data
        0000000F:416024  DB51A4C80A:001CD0  00000001 00000004                      *.............*
        0000000F:416104 foo(), threadSpecific data=1 4
        0000000F:416176 bar(), threadSpecific data=1 4
        0000000F:416232 bar(): This is an error tracepoint
        0000000F:416288 Stack Dump For Current Thread
        0000000F:416344 Stack:  This thread's stack at time of error in bar()
        0000000F:416552 Stack:  Library    / Program     Module      Stmt    Procedure
        0000000F:416696 Stack:  QSYS       / QLESPI      QLECRTTH    774   : LE_Create_Thread2__FP12crtth_parm_t
        0000000F:416784 Stack:  QSYS       / QP0WPTHR    QP0WPTHR    1008  : pthread_create_part2
        0000000F:416872 Stack:  MYLIB      / PTHREADT    PTHREADT    19    : theThread__FPv
        0000000F:416952 Stack:  MYLIB      / PTHREADT    PTHREADT    29    : foo__Fv
        0000000F:531432 Stack:  MYLIB      / PTHREADT    PTHREADT    46    : bar__Fv
        0000000F:531544 Stack:  QSYS       / QP0ZCPA     QP0ZUDBG    87    : Qp0zDumpStack
        0000000F:531632 Stack:  QSYS       / QP0ZSCPA    QP0ZSCPA    276   : Qp0zSUDumpStack
        0000000F:531704 Stack:  QSYS       / QP0ZSCPA    QP0ZSCPA    287   : Qp0zSUDumpTargetStack
        0000000F:531744 Stack:  Completed
        0000000F:531856 foo(): This is an error tracepoint
        0000000F:531952 dataDestructor: Free data
      0000000D:532528 Main completed
*/
#define _MULTI_THREADED
#include #include #include #include #include
#define checkResults(string, val) {             ¥
 if (val) {                                     ¥
   printf("Failed with %d at %s", val, string); ¥
   exit(1);                                     ¥
 }                                              ¥
}

typedef struct {
   int          threadSpecific1;
   int          threadSpecific2;
} threadSpecific_data_t;

#define                 NUMTHREADS   2
pthread_key_t           threadSpecificKey;

void foo(void);
void bar(void);
void dataDestructor(void *);

void *theThread(void *parm) {
   int                    rc;
   threadSpecific_data_t  *gData;
   PTHREAD_TRACE_NP({
                   Qp0zUprintf("Thread Entered¥n");
                  Qp0zDump("Global Data", parm, sizeof(threadSpecific_data_t));},
                  PTHREAD_TRACE_INFO_NP);
   gData = (threadSpecific_data_t *)parm;
   rc = pthread_setspecific(threadSpecificKey, gData);
   checkResults("pthread_setspecific()¥n", rc);
   foo();
   return NULL;
}

void foo() {
   threadSpecific_data_t *gData =
      (threadSpecific_data_t *)pthread_getspecific(threadSpecificKey);
   PTHREAD_TRACE_NP(Qp0zUprintf("foo(), threadSpecific data=%d %d¥n",
                               gData->threadSpecific1, gData->threadSpecific2);,
                  PTHREAD_TRACE_INFO_NP);
   bar();
   PTHREAD_TRACE_NP(Qp0zUprintf("foo(): This is an error tracepoint¥n");,
                  PTHREAD_TRACE_ERROR_NP);
}

void bar() {
   threadSpecific_data_t *gData =
      (threadSpecific_data_t *)pthread_getspecific(threadSpecificKey);
   PTHREAD_TRACE_NP(Qp0zUprintf("bar(), threadSpecific data=%d %d¥n",
```

```
                              gData->threadSpecific1, gData->threadSpecific2);,
                  PTHREAD_TRACE_INFO_NP);
   PTHREAD_TRACE_NP(Qp0zUprintf("bar(): This is an error tracepoint¥n");
                  Qp0zDumpStack("This thread's stack at time of error in bar()");,
                  PTHREAD_TRACE_ERROR_NP);
   return;
}

void dataDestructor(void *data) {
   PTHREAD_TRACE_NP(Qp0zUprintf("dataDestructor: Free data¥n");,
                  PTHREAD_TRACE_INFO_NP);
   pthread_setspecific(threadSpecificKey, NULL);   free(data);
   /* If doing verbose tracing we will even write a message to the job log */
   PTHREAD_TRACE_NP(Qp0zLprintf("Free'd the thread specific data¥n");,
                  PTHREAD_TRACE_VERBOSE_NP);
}

/* Call this testcase with an optional parameter 'PTHREAD_TRACING' */
/* If the PTHREAD_TRACING parameter is specified, then the         */
/* Pthread tracing environment variable will be set, and the       */
/* pthread tracing will be re initialized from its previous value. */
/* NOTE: We set the trace level to informational, tracepoints cut  */
/*       using PTHREAD_TRACE_NP at a VERBOSE level will NOT show up*/
int main(int argc, char **argv) {
   pthread_t              thread[NUMTHREADS];
   int                   rc=0;
   int                    i;
   threadSpecific_data_t *gData;
   char                   buffer[50];

   PTHREAD_TRACE_NP(Qp0zUprintf("Enter Testcase - %s¥n", argv[0]);,
                  PTHREAD_TRACE_INFO_NP);
   if (argc == 2 && !strcmp("PTHREAD_TRACING", argv[1])) {
      /* Turn on internal pthread function tracing support          */
      /* Or, use ADDENVVAR, CHGENVVAR CL commands to set this envvar*/
      sprintf(buffer, "QIBM_PTHREAD_TRACE_LEVEL=%d", PTHREAD_TRACE_INFO_NP);
      putenv(buffer);
      /* Refresh the Pthreads internal tracing with the environment */
      /* variables value.                                           */
      pthread_trace_init_np();
   }
   else {
      /* Trace only our application, not the Pthread code           */
      Qp0wTraceLevel = PTHREAD_TRACE_INFO_NP;
   }

   rc = pthread_key_create(&threadSpecificKey, dataDestructor);
   checkResults("pthread_key_create()¥n", rc);

   for (i=0; i threadSpecific1 = i;
      gData->threadSpecific2 = (i+1)*2;
      rc = pthread_create( &thread[i], NULL, theThread, gData);
      checkResults("pthread_create()¥n", rc);
      PTHREAD_TRACE_NP(Qp0zUprintf("Wait for the thread to complete, "
                                   "and release their resources¥n");,
                     PTHREAD_TRACE_INFO_NP);
      rc = pthread_join(thread[i], NULL);
      checkResults("pthread_join()¥n", rc);
   }

   pthread_key_delete(threadSpecificKey);
   PTHREAD_TRACE_NP(Qp0zUprintf("Main completed¥n");,
                  PTHREAD_TRACE_INFO_NP);
   return 0;
}
```

フライト・レコーダーを使った、マルチスレッド・アプリケーションのデバッグについては、 72 ページの『フライト・レコーダー』を参照してください。

スレッド情報を表示するためのオプション

ジョブの表示 (DSPJOB)、ジョブの処理 (WRKJOB)、および活動ジョブの処理 (WRKACTJOB) コマンドにより、 OS/400 ジョブに関連したスレッドを表示また処理することができます。それぞれのコマンドでは、ジョブ内でスレッドを表示または処理するためのオプションを選択します。「スレッドの処理」画面で、ジョブについて以下の情報を表示することができます。

スレッドのリスト:

この画面には、現在ジョブと関連付けられているすべてのスレッドが表示されます。リストの最初のスレッドは、プロセスの初期スレッドです。 8 桁の数値 (スレッド) は、スレッドの識別コードを表します。このリストは、ジョブのスレッド活動によって変化します。この画面には他にも、スレッドの状況 (活動中または待機中)、 CPU の合計使用量、補助記憶装置の合計入出力カウント、およびスレッドの実行優先順位などの情報が表示されます。

スレッド呼び出しスタック・オプション:

このオプションは、任意のスレッドの呼び出しスタックを表示します。状況によっては、呼び出しスタックが表示されないことがあります (スレッドが割り込み不能な命令を実行している場合)。そのような場合、スタック情報が利用可能でないことを示すメッセージが表示されます。

スレッド mutex オプション:

このオプションは、特定のスレッドと関連付けられているすべての mutex を表示します。その中には、保留中の mutex や、利用可能になるまでスレッドが待機している mutex も含まれます。指定したスレッドに、関連付けられた mutex がない場合には、そのジョブまたはスレッドについては mutex メッセージは表示されません。

スレッド・ロック・オプション:

このオプションは、スレッドが保留しているスレッド有効範囲オブジェクト・ロックをすべて表示します。また、利用可能になるまでスレッドが待機している、保留中のプロセス有効範囲ロックやスレッド有効範囲ロックもすべて表示します。指定されたスレッドに関連付けられたロックがない場合、「スレッドにロックがありません」というメッセージが表示されます。

スレッドの保留オプション:

このオプションは、特定スレッドの実行を一時停止できます。このオプションが効果的なのは、あるスレッドがループしている、あるいは他のシステム問題の原因になっている疑いがあるときです。スレッドの保留オプションは、スレッドの解放オプションと組み合わせて使用してください。ジョブの保留 (HLDJOB) とは異なり、「スレッドの保留」の要求は繰り返すと累積されます。保留中のスレッドの実行を再開するには、同じ回数だけ「スレッドの解放」要求を出さなければなりません。

スレッドの解放オプション:

このオプションは、一時停止状態にあるスレッドの実行を再開します。

スレッドの終了オプション:

このオプションは、特定のスレッドを終了します。予期せぬ結果が生じることがあるため、通常はこのオプションを使用しません。このオプションは、*SERVICE 特殊権限を必要としています。

ジョブの解放 (RLSJOB)、ジョブの保留 (HLDJOB)、ジョブの終了 (ENDJOB)、および異常ジョブの終了 (ENDJOBABN) コマンドは、1 つのマルチスレッド・ジョブ内のすべてのスレッドに影響します。同じようなスレッド・レベルのサポートを提供するコマンドの計画はありません。

マルチスレッド・ジョブのデバッグ

spawn() API は、デバッグ対象のマルチスレッド・プログラムを実際に呼び出す前にデバッグ・セッションを開始するためのメカニズムを提供します。 QIBM_CHILD_JOB_SNDINQMSG 環境変数を 1 に設定すると、この機能を制御することができます。この環境変数の使用については、 System API Reference に記載されている spawn() API の使用上の注意を参照してください。

SPAWN コマンド例は、OS/400 オプション 7、OS/400 サンプル・ツール・ライブラリー、 QUSRTOOL の一部として提供されており、それを使用また修正することができます。 QUSRTOOL ライブラリーのファイル QATTINFO にあるメンバー TP0ZINFO には、 SPAWN CL コマンドの作成方法が記載されています。 SPAWN コマンド使用法の詳細については、SPAWN CL コマンド、QUSRTOOL 例も参照してください。

マルチスレッド・プログラムおよびデバッグ・セッションを実行するジョブの開始プロセスは、必要なステップを実行するコマンド (SPAWN コマンドなど) を作成することにより単純化することもできます。そのステップは次のとおりです。

- ディスプレイ装置セッションにサインオンする
- マルチスレッド・プログラムを実行するジョブを開始する
- サービス・ジョブの開始 (STRSRVJOB) コマンドを使って、マルチスレッド・プログラムを実行しているジョブにサービスを提供する
- デバッグの開始 (STRDBG) コマンドを呼び出し、デバッグするプログラムを追加し、ブレークポイントを設定する
- スレッドがブレークポイントにヒットしたときにそのことをサービス・ジョブに通知し、ジョブがスレッドを停止するようにする
- デバッガー・サポートによりすべてのスレッドを停止する
- 表示されるデバッグ画面から有効なデバッグ・コマンドを発行する
- ブレークポイントをヒットしたスレッド、その後に他のすべてのスレッドを再開する
- デバッグが完了するまで上のサイクルを繰り返す
- デバッグの終了 (ENDDBG) コマンドを呼び出す

System API Reference、*WebSphere Development Studio: ILE C/C++ Programmer's Guide (SC09-2712)*  、および ILE 概念 (SD88-5033)  資料には、スレッドのデバッグ機能についての詳細が記載されています。

マルチスレッド・アプリケーションのテスト分野

テストは、マルチスレッド・プログラムの修正作業を検証するプロセスにおける大切なポイントです。マルチスレッド・プログラムをテストするときは、次のような主要な概念について考慮してください。

複数インスタンス:

マルチスレッド・プログラムをテストするには、プログラムの複数インスタンスを同時にアクティブにします。アプリケーションでスレッドの数が可変になっている場合には、プログラムの各インスタンスに、異なるスレッド数を設定してください。

システム作業負荷の変化:

マルチスレッド・プログラムをテストするには、実行するアプリケーションの組み合わせを変えながら、プログラムを繰り返し実行します。異なるアプリケーション間の対話により、タイミング問題や競争状態などの問題が明らかになるかもしれません。

作業負荷の大きい環境:

作業負荷の大きい環境では、競合、タイミング、およびパフォーマンスの問題が明らかになるかもしれません。

異なるハードウェア・モデル:

可能であれば、作業負荷や緊張レベルを変えながら、いくつかの異なるハードウェア・モデルでマルチスレッド・プログラムを実行してください。ハードウェア・モデルが異なると (特にマルチプロセッサー・システム)、さまざまな問題が明らかになるかもしれません。別のプラットフォームからアプリケーションを移植する場合は、アプリケーションの結果がどちらのプラットフォームでも同じであるかどうかを検査してください。

マルチスレッド・プログラムの正しさを妥当性検査する際、テストは考慮事項の 1 つにすぎません。コードの検査もとても重要です。それが問題判別の唯一の方法である場合も多くあります。マルチスレッド・プログラムのコードを検査する場合、「もしコードがこのポイントで中断されたとすると、いったいどのような問題が生じるか」と考えてください。問題が発生するかもしれないソースを判別できると、問題をあらかじめ回避できるかもしれません。一般的な問題は次のとおりです。

ハードウェア関連の問題:

通常は、1 つのハードウェア・モデルでシングルスレッド・プログラムを 1 つテストすれば十分です。他のハードウェア・モデルでも結果はまったく同じであると推測することができます。しかし、カーネル・スレッドをサポートするハードウェア・プラットフォームで実行しているマルチスレッド・プログラムの場合、シングルスレッド・プログラムの場合と同じ推測をすることはできません。これらのプラットフォームでは、各スレッドが、個別にディスパッチされたタスクで実行されます。プロセッサーの速度、主メモリーのサイズ、記憶域の容量、および他のハードウェア特性によっては、異なるハードウェアでマルチスレッド・プログラムを呼び出すと、タイミング問題や競争状態などが明らかになることがあります。マルチプロセッサー・システムでは、潜在的な問題がさらに多く見つかるかもしれません。マルチプロセッサー・システムでは、2 つの異なるスレッドが、同じコード順序を同時に実行する可能性があります。

Java 仮想計算機 (VM) の実装の問題:

Java アプリケーション開発者にとって問題となるのが、Java 仮想計算機 (JVM) の実装です。いくつかのプラットフォームでは JVM はシングルスレッドですが、 OS/400 では、JVM はマルチスレッドだからです。ユーザー・スレッドをサポートしている他のプラットフォーム上のアプリケーションと同じように、 Java アプリケーションでも同様の問題が生じることがあります。

スレッド・モデルの問題:

別のプラットフォームからアプリケーションを移植する場合は、相手のプラットフォームがどのスレッド・モデルをサポートしているかを把握している必要があります。そのプラットフォームがユーザー・スレッド・モデルをサポートしている場合には、 OS/400 上のアプリケーションでは問題が生じることもあります。これは、OS/400 がカーネル・スレッド・モデルをサポートしているからです。 2 つのモデルの大きな違いは、ユーザー・スレッド・モデルは連携スケジューリングを使用するのに対し、カーネル・スレッド・モデルはプリエンプティブ・スケジューリングを使用する点です。ユーザー・スレッド・モデルでは、ある時点でアプリケーションの 1 つのスレッドしかアクティブになれません。カーネル・スレッド・モデルでは、どのスレッドを実行するのが適格

かをマシンが判別します。カーネル・スレッドでは、アプリケーションの開発者がユーザー・スレッドの使用時にこれら潜在的な問題について考慮しなかったために、競争状態、リソースの競合、およびその他の問題が生じることがあります。

第 9 章 マルチスレッド・アプリケーションにおけるパフォーマンスの考慮事項

OS/400 におけるマルチスレッド・ジョブのパフォーマンスの考慮事項については、以下のリンクを参照してください。

- 『マルチスレッド・サーバーに関する推奨事項』
- 82 ページの『ジョブおよびスレッド優先順位』
- 82 ページの『スレッドの競合』
- 82 ページの『スレッド・アプリケーションに対する記憶域プール・サイズの影響』
- 83 ページの『記憶域プールの活動レベル』
- 83 ページの『パフォーマンスおよびスレッド・アプリケーション』

ここで説明する概念は、すべてのプログラミング言語に関係しています。それぞれの言語でこの概念がどのように実現されているかについては、各言語のプログラマーの手引きを参照してください。

マルチスレッド・サーバーに関する推奨事項

スレッドの利点を活用する典型的なサーバーを作成する場合、クライアント要求を待つ単一の listener スレッドと、クライアントから要求された操作を実行する複数のワーカー・スレッドを作成するという設計が一般的です。

スレッドの作成には、あまり多くのリソースを使わないようにしてください。ただし、サーバーによってはクライアントへの高速な対応が求められるため、サーバー・アプリケーション・プログラムが、実行を待機するワーカー・スレッドのプールを維持するようになっています。これは、新しいスレッドの作成を回避することを目的としています。

通常、ワーカー・スレッドはいくつかの同期化プリミティブを使って、クライアントからの処理要求を待ちます。各クライアント要求を処理するたびに新しいスレッドを作成するのではなく、 listener スレッドは、クライアント要求を待ち行列に入れて、待機中のワーカー・スレッドにシグナルを出します。シグナルに変数が使われることもあります。

普通、サーバー・アプリケーションは、クライアントのために処理するデータに関してまかされているものと見なされています。サーバーはマルチスレッド・アプリケーションを実行しているため、サーバーが実行するアクティビティーについていくつか考慮事項があります。

- マルチスレッド・サーバーからユーザー・アプリケーション・コードを呼び出すべきではありません。マルチスレッド・サーバーからユーザー・アプリケーション・コードを安全に実行するには、ユーザー・アプリケーション・コードが、元のマルチスレッド・サーバーが対応したのと同じ厳密な規則に従わなければなりません。この規則には、実行可能なアクションや、呼び出し可能な API などが関係しています。
- アプリケーションの他の部分と同じように、クライアント要求を安全に実行するのに必要な処理を評価する必要があります。
- クライアントに代わって処理することには、サーバーのプロセス・レベルのリソースにとって好ましくない影響があるかもしれません。たとえば、クライアントでのデータ表現に合わせるために CCSID を変更すると、ジョブ内の他のスレッドにも影響が出る恐れがあります。 CCSID はジョブ・リソースの 1 つです。

- サーバーは、ワーカー・スレッドのセキュリティー情報 (ユーザー・プロファイルおよびグループ・プロファイル) を変更し、それをクライアントにすることができます。ただし、サーバーがこのように処理するときは、そのスレッドがどのリソースを共用しているかを考慮しなければなりません。ワーカー・スレッドは、すでに開かれているすべてのジョブ・レベル・リソースに対してアクセスできますが、その中には同じジョブで他の特権ユーザーが作成、または開いたリソースが含まれている可能性があります。

ジョブおよびスレッド優先順位

OS/400 スレッドは、同じジョブ内の他のスレッドと競合するだけでなく、システム全体でもリソースのスケジューリングを巡って他のスレッドと競合します。システムは、リソースの処理をスケジューリングするために、いくつかの遅延コスト曲線 (優先順位範囲) を基にした遅延コスト・スケジューラーを使用します。

OS/400 では、優先順位の数値が小さい方が、スケジューリングに関して優先順位は高くなります。スレッド優先順位の調整値をジョブの優先順位に追加することにより、スレッドの優先順位を指定します。スレッドのデフォルト優先順位は、プロセス優先順位と同等か、スレッド優先順位の調整値 0 です。

異なるスレッド優先順位調整値をアプリケーションのスレッドに割り当てると、アプリケーションのパフォーマンスに直接影響があります。

スレッドの競合

あるスレッドが、別のスレッドがリソースの使用を終えるまで待機しなければならないときに競合が生じます。競合問題が生じるのは、大量のリソースへのアクセスを保護するためにアプリケーションが使用する相互排他 (mutex) が少なすぎるときです。また、少量のリソースを大量のスレッドが共用するときも、アプリケーションのスレッド間で競合が生じます。

リソースに対するスレッド間の競合により、コンテキストのスイッチやページングが生じることがあります。アプリケーション内の競合を減らすには、できるだけ短い時間だけロックを保持し、 1 つのロックが 2 つの別個の (あるいは無関係の) 共用リソースのために使用されないようにします。

リソースを待つためにスレッドがポーリングやスピンを行うと、スケーリングが阻まれます。この種の競合は、アプリケーションのパフォーマンスに対して重大な影響を与えます。ポーリングやスピンは他のスレッドやジョブに対しても悪影響を及ぼし、結果としてシステム・パフォーマンスを悪化されることもあります。リソースが使用可能になるまで待機するときにスレッドがポーリングやスピンを使わなくてもよいように、条件変数、セマフォー、mutexe、または他の同期化プリミティブを使用してください。

スレッド・アプリケーションに対する記憶域プール・サイズの影響

サブシステムのために指定される記憶域プール・サイズは、パフォーマンスや、作成可能なスレッド数に影響します。スレッドの作成や処理が遅いか、一貫してメモリー不足エラーで失敗する場合は、同じ記憶域プールを使う他のジョブに加えて、すべてのアプリケーション・スレッドを実行するのに使用可能なリソースが記憶域プールで不足している可能性があります。記憶域プールのサイズを大きくすれば、これらの問題がなくなるかもしれません。

記憶域プールのサイズは、システム状況の処理 (WRKSYSSTS) コマンドを使って判別できます。アクティブ・スレッドが多すぎるか、スレッドがシステム・リソースを使いすぎているなら、アプリケーションが実行されている記憶域プールに多くのページ不在が見られるはずです。記憶域プールのサイズを大きくすれば、ページ不在の数は減るかもしれません。

記憶域プールの活動レベル

記憶域プールの活動レベルとは、記憶域プール中の活動スレッドの数を指します。 500 スレッドを持った 1 つのジョブは、1 つのスレッドを持った 500 個のジョブと同じ数の活動レベル・スロットを使います。活動しているスレッドおよびジョブの数に対して活動レベルが低すぎると、スレッドは主記憶域からページアウトされ、短時間ですが不適格とマークされます。これはアプリケーションのパフォーマンスに対して重大な影響を与えます。

パフォーマンスおよびスレッド・アプリケーション

ジョブ・レベルのパフォーマンス・カウンターは、1 つのジョブのすべてのスレッドにより更新されます。活動状態のスレッドが複数あると、一般的なシステム・カウンターと、パフォーマンス・モニター固有のトランザクション境界カウンターの両方の正確さに影響があります。これらのカウンター用の自動同期化がないと、データが損失される可能性もあります。

初期スレッド・パフォーマンス情報は、スレッドとジョブ範囲データの組み合わせです。すべてのスレッドのスレッド有効範囲データを合計するのでないかぎり、スレッドとジョブ範囲データの両方を含んだ派生情報は無効です。

第 10 章 例: スレッド

次のリストは、「マルチスレッド・アプリケーションのプログラミング」情報で使用されているすべての例を示しています。プログラムの例として使用してください。

Pthread サンプル:

Java サンプル:

Printed in Japan

日本アイ・ビー・エム株式会社
〒106-8711　東京都港区六本木3-2-12