マルチスレッドのプログラミング

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650–960–1300

Part Number 806–7118–10
2001 年 2 月

原典: *Multithreaded Programming Guide*

Part No: 806-5257-10

Revision A

目次

はじめに

このマニュアルは、Solaris™ オペレーティング環境で使用される POSIX スレッドと Solaris スレッドに対応した、マルチスレッドのプログラミングインタフェースの解説書です。このマニュアルでは、アプリケーションを作成するプログラマを対象に、マルチスレッドを使った新しいプログラムの作成方法と、既存のプログラムをマルチスレッド化する方法を説明します。

このマニュアルは、POSIX スレッドと Solaris スレッドの両方の実装を扱っていますが、ほとんどの説明は POSIX スレッドを想定して書かれています。Solaris スレッドだけに適用される情報については、独立した章を設けて解説しています。

このマニュアルは、読者が次の基礎知識を持っていることを前提にして書かれています。

- UNIX SVR4 システム — 特に Solaris オペレーティング環境
- C プログラミング言語 — マルチスレッドが `libthread` ライブラリで実装されている
- 並行プログラミング (逐次プログラミングに対して) の原理 — マルチスレッドでは、機能の相互作用についての考え方を変える必要があります。以下に参考文献を示します。
  - 『*Algorithms for Mutual Exclusion*』、Michel Raynal (MIT Press, 1986)
  - 『*Concurrent Programming*』、Alan Burns & Geoff Davies (Addison-Wesley, 1993)
  - 『*Distributed Algorithms and Protocols*』、Michel Raynal (Wiley, 1988)

- 『オペレーティングシステムの概念 *(I、II)*』、シルバーシャッツ、ピーターソン著、宇都宮・福田訳、培風館、原典『*Operating System Concepts*』、Silberschatz、Peterson、Galvin (Addison-Wesley, 1991)

- 『並行プログラミングの原理』、M・ベンアリ、渡辺訳、啓学出版、原典『*Principles of Concurrent Programming*』、M. Ben-Ari (Prentice-Hall, 1982)

Sun のマニュアルの注文方法

専門書を扱うインターネットの書店 Fatbrain.com から、米国 Sun Microsystems™, Inc. (以降、Sun™ とします) のマニュアルをご注文いただけます。

マニュアルのリストと注文方法については、`http://www1.fatbrain.com/documentation/sun` の Sun Documentation Center をご覧ください。

内容の紹介

第 1 章では、このリリースにおけるスレッドの実装について概説します。

第 2 章では、デフォルトの属性をもつスレッドの作成方法を中心に、一般的な POSIX スレッドライブラリルーチンについて説明します。

第 3 章では、デフォルト以外の属性をもつスレッドの生成方法を説明します。

第 4 章では、スレッドライブラリの同期ルーチンについて説明します。

第 5 章では、マルチスレッドをサポートするためにオペレーティング環境に加えられた変更を説明します。

第 6 章では、マルチスレッドの安全性に関する問題を説明します

第 7 章では、マルチスレッド対応のアプリケーションのコンパイルとデバッグの基礎を説明します。

第 8 章では、Solaris スレッド (POSIX スレッドと対比して) のインタフェースについて説明します。

第 9 章では、マルチスレッドアプリケーションを作成するプログラマに関係する問題について説明します。

付録 A では、POSIX スレッド用にコードを指定する方法を示します。

付録 B では、Solaris スレッドの中にバリアを設ける例を示します。

付録 C では、ライブラリルーチンの安全レベルの一覧を示します。

Sun のオンラインマニュアル

`http://docs.sun.com` では、Sun が提供しているオンラインマニュアルを参照することができます。マニュアルのタイトルや特定の主題などをキーワードとして、検索を行うこともできます。

表記上の規則

このマニュアルでは、次のような字体や記号を特別な意味を持つものとして使用します。

表 P–1 表記上の規則

字体または記号	意味	例
`AaBbCc123`	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コード例を示します。	`.login` ファイルを編集します。 `ls -a` を使用してすべてのファイルを表示します。 `system%`
`AaBbCc123`	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して示します。	`system% su` `password:`
AaBbCc123	変数を示します。実際に使用する特定の名前または値で置き換えます。	ファイルを削除するには、`rm` *filename* と入力します。
『 』	参照する書名を示します。	『コードマネージャ・ユーザーズガイド』を参照してください。

表 P–1 表記上の規則 続く

字体または記号	意味	例
「 」	参照する章、節、ボタンやメニュー名、強調する単語を示します。	第 5 章「衝突の回避」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅を超える場合に、継続を示します。	`sun% grep '^#define \` `XV_VERSION_STRING'`

ただし AnswerBook2™ では、ユーザーが入力する文字と画面上のコンピュータ出力は区別して表示されません。

コード例は次のように表示されます。

- C シェル

```
machine_name% command y|n [filename]
```

- C シェルのスーパーユーザー

```
machine_name# command y|n [filename]
```

- Bourne シェルおよび Korn シェル

```
$ command y|n [filename]
```

- Bourne シェルおよび Korn シェルのスーパーユーザー

```
# command y|n [filename]
```

[ ] は省略可能な項目を示します。上記の例は、*filename* は省略してもよいことを示しています。

| は区切り文字 (セパレータ) です。この文字で分割されている引数のうち 1 つだけを指定します。

キーボードのキー名は英文で、頭文字を大文字で示します (例: Shift キーを押します)。ただし、キーボードによっては Enter キーが Return キーの動作をします。

ダッシュ (-) は 2 つのキーを同時に押すことを示します。たとえば、`Ctrl-D` は `Control` キーを押したまま `D` キーを押すことを意味します。

一般規則

- このマニュアルでは、「IA」という用語は、Intel 32 ビットのプロセッサアーキテクチャを意味します。これには、Pentium、Pentium Pro、Pentium II、Pentium II Xeon、Celeron、Pentium III、Pentium III Xeon の各プロセッサ、および AMD、Cyrix が提供する互換マイクロプロセッサチップが含まれます。

第 1 章

マルチスレッドの基礎

マルチスレッドという用語は、「複数の制御スレッド」または「複数の制御フロー」という意味で使われます。従来の UNIX のプロセスは、1 つの制御スレッドで動作していましたが、マルチスレッド (MT) では、1 つのプロセスを複数のスレッドに分割し、それぞれのスレッドが独立に動作します。

プログラムをマルチスレッド化すると、次のような利点が生まれます。

- アプリケーションの応答性が改善される
- マルチプロセッサをより効率的に利用できる
- プログラム構造が改善される
- システムリソースが節約できる

この章では、マルチスレッドについての用語、利点、および概念を説明します。こうした事柄について理解できている方は、第 2 章に進んでください。

マルチスレッドに関する用語の定義

表 1–1 で、このマニュアルで使われている主な用語を紹介します。

表 1–1 マルチスレッドに関する用語の定義

用語	定義
プロセス	`fork(2)` システムコールで生成される UNIX 環境 (ファイル記述子やユーザ ID などのコンテキスト) で、プログラムを実行するために設定される。
スレッド	プロセスのコンテキスト内で実行されるひとまとまりの命令
`pthread` (POSIX スレッド)	POSIX 1003.1c に準拠したスレッドインタフェース
Solaris スレッド	POSIX に準拠しない、Sun Microsystems™ のスレッドインタフェース。`pthread` より先に存在
シングルスレッド化	1 プロセス 1 スレッドで動作させること
マルチスレッド化	1 プロセス複数スレッドで動作させること
ユーザレベルのスレッドまたはアプリケーションレベルのスレッド	(カーネル空間に対応する) ユーザ空間に位置し、スレッドライブラリルーチンによって管理されるスレッド
軽量プロセス (LWP)	カーネルコードやシステムコールを実行する、カーネル内部のスレッド
結合スレッド	LWP に固定的に結合したスレッド
非結合スレッド	カーネルのサポートなしでコンテキストが非常にすばやく切り替わるデフォルトの Solaris スレッド
属性オブジェクト	不透明なデータ型と関連操作のための関数が含まれ、POSIX スレッド、mutex、条件変数の調整可能な部分を共通化するために使用される
相互排他ロック	共有データへのアクセスをロック / ロック解除する機能
条件変数	状態が変化するまでスレッドをブロックする機能
計数型セマフォ	メモリーを使用する同期機構

表 1–1　マルチスレッドに関する用語の定義　続く

用語	定義
並列性	2 つ以上のスレッドが同時に実行されている状態を表す概念
並行性	2 つ以上のスレッドが進行過程にある状態を表す概念。仮想的な並列性としてタイムスライスを包含する、一般化された形の並列性

マルチスレッドの標準への適合

マルチスレッドのプログラミングという概念の起源は、少なくとも 1960 年代にまでさかのぼります。マルチスレッドが UNIX システム上で開発されたのは 1980 年代の中期になります。マルチスレッドの意味とそのサポートに必要な機能については合意がありますが、マルチスレッドを実装するためのインタフェースはさまざまです。

この数年間、POSIX (Portable Operating System Interface) 1003.4a というグループによって、スレッドプログラミングの標準化についての作業が行われてきました。この標準はいまや承認されるに至っています。この『マルチスレッドのプログラミング』は、POSIX 規格の P1003.1b 最終草稿 14 (リアルタイム) と P1003.1c 最終草稿 10 (マルチスレッド) に基づいています。

このマニュアルは、POSIX スレッド (`pthread` とも言います) と Solaris スレッドの両方を対象にしています。Solaris スレッドは Solaris 2.4 以降のリリースで利用できますし、POSIX スレッドとも機能的に異なりません。しかし、Solaris スレッドより POSIX スレッドのほうが移植性が高いので、このマニュアルではマルチスレッドを POSIX の立場から解説しています。Solaris スレッドに固有な事柄については、第 8 章で説明します。

マルチスレッドの利点

アプリケーションの応答性の改善

互いに独立した処理を含んでいるプログラムは、設計を変更して、個々の処理をスレッドとして定義できます。たとえば、マルチスレッド化された GUI のユーザは、ある処理が完了しないうちに別の処理を開始できます。

マルチプロセッサの効率的な利用

スレッドによって並行化されたアプリケーションでは、ほとんどの場合、利用可能なプロセッサ数を考慮する必要はありません。そのようなアプリケーションでは、プロセッサを追加するだけで性能が目に見えて改善されます。

行列の乗算のような並列性の度合いが高い数値計算アプリケーションは、マルチプロセッサ上でスレッドを実装することにより、処理速度を大幅に改善できます。

プログラム構造の改善

ほとんどのプログラムは、単一のスレッドで実現するよりも複数の独立した (あるいは半独立の) 実行単位の集合体として実現した方が効果的に構造化されます。マルチスレッド化されたプログラムの方が、シングルスレッド化されたプログラムよりもユーザのさまざまな要求に柔軟に対応できます。

システムリソースの節約

共有メモリーを通して複数のプロセスが共通のデータを利用するようなプログラムは、複数の制御スレッドを使用していることになります。

しかし、各プロセスは完全なアドレス空間とオペレーティング環境上での状態をもちます。そのような大規模な状態情報を作成して維持しなければならないという点で、プロセスはスレッドに比べて時間的にも空間的にも不利です。

さらに、プロセス本来の独立性のため、他のプロセスに属するスレッドと通信したり同期を取ったりする際に、プログラマは面倒な処理をしなくてはなりません。

スレッドと RPC の併用

スレッドと遠隔手続き呼び出し (RPC) パッケージを組み合わせると、メモリーを共有していないマルチプロセッサ (たとえば、ワークステーションの集合体) を活用できます。この方法では、アプリケーションの分散処理を比較的簡単に実現でき、ワークステーションの集合体を 1 台のマルチプロセッサのシステムとして扱います。

たとえば、最初にあるスレッドがいくつかの子スレッドを生成します。それらの子スレッドは、それぞれが遠隔手続き呼び出しを発行して、別のワークステーション上の手続きを呼び出します。結果的に、最初のスレッドが生成した複数のスレッドは、他のコンピュータとともに並列的に実行されます。

マルチスレッドの基本概念

並行性と並列性

マルチスレッドプロセスがシングルプロセッサ上で動作する場合は、プロセッサが実行リソースを各スレッドに順次切り替えて割り当てるため、プロセスの実行状態は並行的になります。

同じマルチスレッドプロセスが共有メモリー方式のマルチプロセッサ上で動作する場合は、プロセス中の各スレッドが別のプロセッサ上で同時に走行するため、プロセスの実行状態は並列的になります。

プロセスのスレッド数がプロセッサ数と等しいか、それ以下であれば、スレッドをサポートするシステム (スレッドライブラリ) とオペレーティング環境は、各スレッドがそれぞれ別のプロセッサ上で実行されることを保証します。

たとえば、スレッドとプロセッサが同数で行列の乗算を行う場合は、各スレッド (と各プロセッサ) が 1 つの行の計算を担当します。

マルチスレッドの構造

従来の UNIX でもスレッドという概念はすでにサポートされています。各プロセスは 1 つのスレッドを含むので、複数のプロセスを使うようにプログラミングすれば、複数のスレッドを使うことになります。しかし、1 つのプロセスは 1 つのアドレス空間でもあるので、1 つのプロセスを生成すると 1 つの新しいアドレス空間が作成されます。

新しいプロセスを生成するのに比べると、スレッドを生成するのはシステムへの負荷ははるかに小さくなります。これは、新たに生成されるスレッドが現在のプロセスのアドレス空間を使用するからです。スレッドの切り替えに要する時間は、プロセスの切り替えに要する時間よりもかなり短いです。その理由の 1 つは、スレッドを切り替える上でアドレス空間を切り替える必要がないことです。

同じプロセスに属するスレッド間の通信は簡単に実現できます。それらのスレッドは、アドレス空間を含めあらゆるものを共有しているからです。したがって、あるスレッドで生成されたデータを、他のすべてのスレッドがただちに利用できます。

マルチスレッドをサポートするインタフェースは、サブルーチンライブラリで提供されます (POSIX スレッド用は `libpthread` で、Solaris スレッド用は `libthread` です)。カーネルレベルとユーザレベルのリソースを切り離すことによって、マルチスレッドは柔軟性をもたらします。

ユーザレベルのスレッド

スレッドは、マルチスレッドのプログラミングにおいて基本となるプログラミングインタフェースです。ユーザレベルのスレッド[1] はユーザ空間で処理されるため、カーネルのコンテキストスイッチの負荷を増やすことはありません。何百ものスレッドを使用するようなアプリケーションでも、カーネルのリソースをそれほど使用しなくてもすみます。どのくらいの量のカーネルリソースをアプリケーションが必要とするかは、主にアプリケーション自体の性質で決まります。

スレッドは、それらが存在するプロセスの内部からだけ参照でき、アドレス空間や開いているファイルなどすべてのプロセスリソースを共有します。スレッドごとに固有な状態としては次のものがあります。

- スレッド識別子
- レジスタ状態 (プログラムカウンタとスタックポインタを含む)
- スタック
- シグナルマスク
- 優先順位
- スレッド専用記憶領域

スレッドはプロセスの命令とそのデータの大半を共有するので、あるスレッドが行なった共有データの変更は、プロセスの他のすべてのスレッドから参照できます。

1. ユーザレベルのスレッドという呼称は、システムプログラマだけが関係するカーネルレベルのスレッドと区別するためのものです。このマニュアルは、アプリケーションプログラマ向けであるため、カーネルレベルのスレッドについては触れません。

スレッドが自分と同じプロセス内の他のスレッドとやり取りを行う場合は、オペレーティング環境を介する必要はありません。

デフォルトでは、スレッドは非常に軽量です。しかし、スレッドをより厳格に制御したいアプリケーションでは (たとえば、スケジューリングの方針をより厳密に適用したい場合など)、スレッドを結合できます。アプリケーションがスレッドを実行リソースに結合すると、そのスレッドはカーネルのリソースとなります (詳細は、27ページの「システムスコープ (結合スレッド)」を参照してください)。

以下に、ユーザレベルのスレッドの利点を要約します。

- 独自のアドレス空間を生成する必要がないので、生成に伴うシステムへの負荷が小さくてすみます。わずかな量の仮想メモリを、実行時のアドレス空間に確保するだけです。
- 同期がカーネルレベルでなくユーザレベルでとられるため、高速な同期が可能です。
- スレッドライブラリ `libpthread` と `libthread` によって、簡単に管理できます。

軽量プロセス

スレッドライブラリは、カーネルによってサポートされる軽量プロセス (LWP) と呼ばれる制御スレッドを基礎としています。LWP は、コードやシステムコールを実行する仮想的な CPU と考えることができます。

通常、スレッドを使用するプログラミングで LWP を意識する必要はありません。以下に述べる LWP の説明は、27ページの「プロセススコープ (非結合スレッド)」で述べるスケジューリングスコープの違いを理解する際の参考にしてください。

注 - Solaris 2、Solaris 7、および Solaris 8 オペレーティング環境の LWP と SunOS™ 4.0 LWP ライブラリの LWP は同じものではありません。後者は Solaris 2、Solaris 7、および Solaris 8 オペレーティング環境ではサポートされていません。

`fopen()` や `fread()` などの `stdio` ライブラリルーチンが `open()` や `read()` などのシステムコールを使用するのと同じように、スレッドインタフェースも LWP インタフェースを使用します。

軽量プロセス (LWP) はユーザレベルとカーネルレベルの橋渡しをします。各プロセスは 1 つ以上の LWP を含み、それぞれの LWP は 1 つ以上のユーザスレッドを実行

します (図 1–1 を参照)。スレッドが生成されるときは、通常それに伴ってユーザのコンテキストが作成されますが、LWP は生成されません。

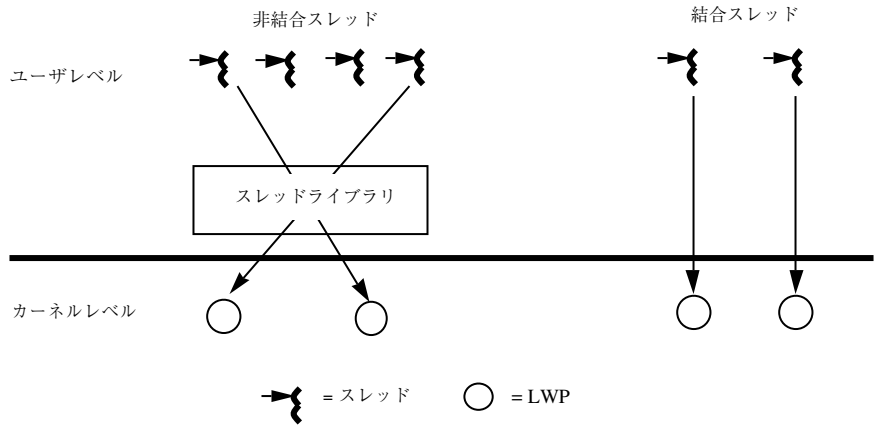

図 1–1　ユーザレベルスレッドと軽量プロセス

各 LWP はカーネルプールの中のカーネルリソースであり、スレッドに割り当てられ (接続され) たり、割り当てを解除され (切り離され) たりします。この割り当て / 割り当て解除はスレッド単位ごとに、スレッドがスケジュールされるか、生成または破棄されたときに行われます。

スケジューリング

POSIX はスケジューリングの方針として、先入れ先出し (SCHED_FIFO)、ラウンドロビン (SCHED_RR)、カスタム (SCHED_OTHER) の 3 つを規定しています。SCHED_FIFO は待ち行列ベースのスケジューラで、優先レベルごとに異なる待ち行列をもっています。SCHED_RR は FIFO に似ていますが、各スレッドに実行時間の制限があるという点が異なります。

SCHED_FIFO と SCHED_RR は両方とも POSIX のリアルタイム拡張機能です。SCHED_OTHER がデフォルトのスケジューリング方針です。

SCHED_OTHER 方針、および POSIX の SCHED_FIFO 方針と SCHED_RR 方針のプロパティのエミュレートについては、174ページの「LWP とスケジューリングクラス」を参照してください。

スケジューリングスコープ (スケジューリングの適応範囲) として、プロセススコープ (非結合スレッド用) とシステムスコープ (結合スレッド用) の 2 つが使用できます。スコープの状態が異なるスレッドが同じシステムに同時に存在でき、さらに同

じプロセスにも同時に存在できます。通常、スコープは適応範囲を設定します。その範囲内でスレッドの方針が有効となります。

プロセススコープ (非結合スレッド)

非結合スレッドは PTHREAD_SCOPE_PROCESS として生成されます。このようなスレッドはユーザ空間内でスケジュールされ、LWP プールの中の使用可能な LWP に対して接続されたり切り離されたりします。LWP はこのプロセス内のスレッドにのみ使用可能です。つまり、スレッドはこれらの LWP にスケジュールされるわけです。

通常は PTHREAD_SCOPE_PROCESS スレッドを使用します。そうすれば、LWP の間でスレッドの使い回しができるので、スレッドの効率が良くなります (また、Solaris スレッドを THR_UNBOUND 状態で生成するのと同じことになります)。スレッドライブラリは、他のスレッドを考慮しつつ、どのスレッドがカーネルのサービスを受けるかを決定します。

システムスコープ (結合スレッド)

結合スレッドは PTHREAD_SCOPE_SYSTEM として生成されます。結合スレッドは、LWP に永久に結合されます。

それぞれの結合スレッドは、初めから終わりまで特定の LWP に結び付けられています。これは Solaris スレッドを THR_BOUND 状態で生成するのと同じことです。スレッドを結合することによって、スレッドに代替シグナルスタックを与えたり、リアルタイムスケジューリングで特別なスケジューリング属性を使用したりできます。すべてのスケジューリングは、オペレーティング環境で行われます。

注 - 結合と非結合のいずれのスレッドの場合でも、他のプロセスからスレッドに直接アクセスしたり、他のプロセスに移動したりできません。

取り消し

スレッド取り消しによって、スレッドはそのプロセス中の他のスレッドの実行を終了させることができます。取り消しの対象となるスレッドは、取り消し要求を保留しておき、取り消しに応じる際にアプリケーション固有のクリーンアップを実行できます。

pthread 取り消し機能では、スレッドの非同期終了または遅延終了が可能です。非同期取り消しはいつでも起こりうるものですが、遅延取り消しは定義されたポイントでのみ発生します。遅延取り消しがデフォルトタイプです。

同期

同期を使用すると、並行的に実行されているスレッドに関して、プログラムの流れと共有データへのアクセスを制御することが可能になります。

相互排他ロック (mutex ロック)、読み取り/書き込みロック、条件変数、セマフォという 4 つの同期モデルがあります。

- 相互排他ロックは、特定のコードセクションを実行する、あるいは特定のデータをアクセスするスレッドを一度に 1 つだけに制限します。
- 読み取り / 書き込みロックを使用することによって、プロテクトのかけられている共有リソースに対する並行読み取りや排他書き込みが可能になります。リソースを変更するには、スレッドがまず排他書き込みロックを獲得する必要があります。すべての読み取りロックが開放されない限り、排他書き込みロックは許可されません。
- 条件変数は、特定の条件が満たされるまでスレッドをブロックします。
- 計数型セマフォは通常、リソースへのアクセスを調整します。計数の値は、セマフォにアクセスできるスレッド数の上限です。設定された値に達したセマフォはブロックされます。

64 ビットアーキテクチャ

アプリケーション開発者にとって、Solaris 64 ビット版と 32 ビット版のオペレーティング環境の主な相違点は、使用する C 言語のデータ型です。64 ビットのデータ型では、long 型とポインタが 64 ビット幅の、LP64 モデルを使用します。その他の基本データ型は 32 ビット版と同じです。32 ビットのデータ型は、int、long、およびポインタが 32 ビットの、ILP 32 モデルを使用します。

64 ビット環境を使用する場合の、主な特徴と注意すべき点について、以下に簡単に説明します。

- 巨大な仮想アドレス空間

64 ビット環境では、プロセスは最大 64 ビットすなわち 18E (エクサ) バイトの仮想アドレス空間を持つことができます。これは、現在の 32 ビット環境における最大 4G バイトの 40 億倍です。ただし、ハードウェアの制約上、一部のプラットフォームでは 64 ビットのアドレス空間を完全にはサポートしていません。

巨大な仮想アドレス空間では、デフォルトのスタックサイズ (32 ビット版では 1M バイト、64 ビット版では 2M バイト) で作成できるスレッドの数も多くなります。デフォルトのスタックサイズで作成できるスレッドの数は、32 ビットシステムで約 2000、64 ビットシステムで約 8 兆です。

- カーネルメモリーの読み取り

  カーネルは LP64 オブジェクトであり、内部では 64 ビットのデータ構造を使用するため、`libkvm`、`/dev/mem`、または`/dev/kmem` を使用する既存の 32 ビットアプリケーションは正常に動作しません。これらのアプリケーションは 64 ビットプログラムに変換する必要があります。

- `/proc` の制限

  `/proc` を使う 32 ビットプログラムでは、32 ビットのプロセスは見ることができますが、64 ビットのプロセスを解釈することはできません。プロセスを記述する既存のインタフェースとデータ構造は、64 ビットのプロセスを収容できるだけの容量がありません。これらのプログラムが 32 ビットと 64 ビットの両プロセスに対して動作できるようにするには、64 ビットプログラムとしてコンパイルし直す必要があります。

- 64 ビットのライブラリ

  32 ビットのアプリケーションは 32 ビットのライブラリと、64 ビットのアプリケーションは 64 ビットのライブラリと、リンクしている必要があります。システムライブラリには、古くなったもの以外はすべて、32 ビットと 64 ビットの両方が用意されています。ただし、64 ビットのライブラリは静的な形式では提供されていません。

- 64 ビット演算

  32 ビット版の従来の Solaris でも、64 ビット演算が行えましたが、64 ビット版では、整数の演算やパラメタの引き渡しに、マシンの 64 ビットレジスタを全面的に使用できるようになりました。

- 大容量のファイル

  アプリケーションが必要としているのが大容量ファイルのサポートだけである場合は、32 ビットのままで大容量ファイルのインタフェースを使用することもでき

ます。ただし、64 ビットの機能を最大限に活かすためには 64 ビットに変換することをお勧めします。

第 2 章

スレッドを使った基本プログラミング

スレッドライブラリ

この章では、POSIX スレッドライブラリ `libpthread(3T)` に入っている基本的なスレッドのプログラミングルーチンについて説明します。この章で説明するスレッドはデフォルトのスレッド (デフォルトの属性値をもつスレッド) です。マルチスレッドのプログラミングで最もよく使われるのがこの種のスレッドです。

第 3 章では、デフォルト以外の属性をもつスレッドの生成方法と使用方法を説明します。

注 - 属性はスレッド生成時にのみ指定されます。スレッドを使用中は変更できません。

この章で紹介する POSIX (`libpthread`) ルーチンのプログラミングインタフェースは、オリジナルの Solaris マルチスレッドライブラリ (`libthread`) のものと類似しています。

次の表は、特定のタスクとその説明が記載されている箇所を示しています。

- 40ページの「スレッド固有データキーの設定」
- 41ページの「スレッド固有データキーの取得」
- 44ページの「スレッド識別子の取得」
- 45ページの「スレッド識別子の比較」
- 45ページの「スレッドの初期化」
- 46ページの「スレッドの実行明け渡し」
- 47ページの「スレッド優先順位の設定」
- 48ページの「スレッド優先順位の取得」
- 48ページの「シグナルのスレッドへの送信」
- 49ページの「呼び出しスレッドのシグナルマスクの変更」
- 51ページの「安全な fork」
- 51ページの「スレッドの終了」
- 54ページの「スレッドの取り消し」
- 55ページの「取り消しを有効または無効にする」
- 56ページの「取り消しタイプの設定」
- 57ページの「取り消しポイントの設定」
- 57ページの「スタックへハンドラをプッシュする」
- 58ページの「スタックからハンドラを取り出す」

デフォルトのスレッドの生成

属性オブジェクトを指定しなければ NULL となり、下記の属性を持つデフォルトスレッドが生成されます。

- 非結合
- 切り離されていない
- デフォルトのスタックとデフォルトのスタックサイズ
- 親の優先順位

`pthread_attr_init()` でデフォルト属性オブジェクトを生成し、この属性オブジェクトを使ってデフォルトスレッドを生成することもできます。詳細は、61ページの「属性の初期化」の節を参照してください。

pthread_create(3THR)

pthread_create(3THR) は、現在のプロセスに新しい制御スレッドを追加します。

```
プロトタイプ:
int pthread_create(pthread_t *tid, const pthread_attr_t *tattr,
    void*(*start_routine)(void *), void *arg);


#include <pthread.h>

pthread_attr_t ()tattr;
pthread_t tid;
extern void *start_routine(void *arg);
void *arg;
int ret;

/* デフォルト動作 */
ret = pthread_create(&tid, NULL, start_routine, arg);

/* デフォルト属性による初期化 */
ret = pthread_attr_init(&tattr);
/* デフォルト動作指定 */
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

必要な状態動作を持つ *tattr* で pthread_create() 関数が呼び出されます。*start_routine* は新しいスレッドで実行する関数です。*start_routine* が復帰すると、スレッドは終了状態を *start_routine* で戻される値に設定して終了します (詳細は、33ページの「pthread_create(3THR) 」を参照してください)。

pthread_create() が正常終了すると、生成されたスレッドの識別子が *tid* の指す記憶場所に格納されます。

スレッドの生成で属性引数として NULL を使用するのは、デフォルト属性を使用するのと同じ効果があります。どちらの場合もデフォルトのスレッドが生成されます。*tattr* は初期化されると、デフォルト動作を獲得します。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると pthread_create() は失敗し、対応する値を返します。

EAGAIN

システム制限を超えました。たとえば、生成する LWP が多すぎます。

EINVAL

tattr の値が無効です。

スレッドの終了待ち

pthread_join(3THR)

pthread_join(3THR) 関数は、スレッドの終了を待ちます。

```
プロトタイプ:
int pthread_join(thread_t tid, void **status);

#include <pthread.h>

pthread_t tid;
int ret;
int status;

/* スレッド「tid」の終了待ち、status の指定あり */
ret = pthread_join(tid, &status);

/* スレッド「tid」の終了待ち、status の指定は NULL */
ret = pthread_join(tid, NULL);
```

pthread_join() 関数は、指定したスレッドが終了するまで呼び出しスレッドをブロックします。

指定するスレッドは、現在のプロセス内のスレッドで、しかも切り離されていないものでなければなりません。スレッドの切り離しについては、63ページの「切り離し状態の設定」を参照してください。

status が NULL でなければ、pthread_join() の正常終了時に *status* の指す記憶場所に終了したスレッドの終了状態が格納されます。

複数のスレッドが、同じスレッドの終了を待つことはできません。そのような状態が発生すると、あるスレッドは正常に戻りますが、他のスレッドは ESRCH エラーを戻し失敗します。

pthread_join() の復帰後は、そのスレッドに関連付けられていたスタック領域がそのアプリケーションで再利用できるようになります。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると、pthread_join() は失敗し、次の値を返します。

ESRCH

tid で指定したスレッドは、現在のプロセス内の切り離されていない正しいスレッドでありません。

EDEADLK

tid に呼び出しスレッドを指定しました。

EINVAL

tid の値が無効です。

pthread_join() ルーチンの引数は 2 つあり、ある程度柔軟な使い方ができます。特定のスレッドが終了するまで待つ場合は、そのスレッドの識別子を第 1 引数として指定します。

終了したスレッドの終了コードを調べたい場合は、それを受け取る領域のアドレスを指定します。

pthread_join() は、切り離されていないスレッドに対してだけ有効であることに注意してください。終了時のタイミングで特に同期をとる必要がないスレッドは、切り離して生成してください。

切り離されたスレッドが頻繁に使用するスレッドであり、切り離されていないスレッドは特に必要な場合に限って使用するものと考えてください。

簡単なスレッドの例

例 2–1 では、あるスレッドが最上位の手続きを実行し、手続き fetch() を実行する補助スレッドを生成します。手続き fetch() は複雑なデータベース検索を行い、処理に多少時間がかかります。

メインスレッドでは検索結果も必要ですが、その間に行うべき処理があります。そこで必要な処理を行なってから、pthread_join() で補助スレッドの終了を待ちます。

新しいスレッドへの引数 *pbe* がスタックパラメタとして渡されます。これが可能なのは、メインスレッドが自分の子スレッドの終了を待つからです。通常は、malloc(3C) でヒープから領域を割り当てる方が、スレッドのスタック領域で (スレッドが終了した場合なくなるか、再度割り当てられる) アドレスを受け渡すよりもよいでしょう。

例 2–1 簡単なスレッドプログラム

```
void mainline (...)
{
        struct phonebookentry *pbe;
        pthread_attr_t tattr;
        pthread_t helper;
        int status;

        pthread_create(&helper, NULL, fetch, &pbe);

            /* この間、他の処理を行う */

        pthread_join(helper, &status);
        /* ここでは結果を確実に使用できる */
}

void fetch(struct phonebookentry *arg)
{
        struct phonebookentry *npbe;
        /* データベースから値を取り出す */

        npbe = search (prog_name)
            if (npbe != NULL)
                *arg = *npbe;
        pthread_exit(0);
}

struct phonebookentry {
        char name[64];
        char phonenumber[32];
        char flags[16];
}
```

スレッドの切り離し

pthread_detach(3THR)

pthread_detach(3THR) は、*detachstate* 属性を PTHREAD_CREATE_JOINABLE に設定して生成されたスレッドの記憶領域を再利用するための、pthread_join(3THR) に代わるもう 1 つの方法です。

```
プロトタイプ:
int pthread_detach(thread_t tid);

#include <pthread.h>

pthread_t tid;
int ret;

/* スレッド tid を切り離す */
ret = pthread_detach(tid);
```

pthread_detach() 関数は、スレッド *tid* のための記憶領域がそのスレッドの終了時に再利用できることを、この実装に対して示すために使われます。*tid* が終了していない場合、pthread_detach() によって、そのスレッドが終了することはありません。同じスレッドに対して複数の pthread_detach() 呼び出しが行われたときの効果は不定です。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると、pthread_detach() は失敗し、対応する値を返します。

EINVAL

tid は、有効なスレッドではありません。

ESRCH

tid は、現在のプロセスの中の有効な切り離されていないスレッドではありません。

スレッド固有データキーの作成

シングルスレッドの C プログラムでは、データは局所データと広域データという 2 つの基本的なクラスに分類されます。一方、マルチスレッドの C プログラムでは、これに第 3 のクラスであるスレッド固有データ (Thread-Specific Data (TSD)) が追加されます。これは広域データと似ていますが、スレッドごとの専用のデータである点が異なります。

スレッド固有データ (TSD) は、スレッド単位で維持管理されます。TSD は、特定のスレッド固有のデータを定義し参照する唯一の手段となります。スレッド固有データの各項目は、プロセス内のすべてのスレッドから参照可能な特定のキー (*key*) と関連付けられます。そのキーを使用することによって、スレッドはスレッド単位で維持管理されるポインタ (void *) にアクセスできます。

pthread_key_create(3THR)

pthread_key_create(3THR) は、プロセス内のスレッド固有データを識別するためのキーを割り当てます。このキーはプロセス内のすべてのスレッドから参照可能で、すべてのスレッドでそのキーが作成された時点では、初期値として NULL が関連付けられています。

pthread_key_create() は、キーの使用前に各キーについて 1 回呼び出します。暗黙の同期はありません。

作成されたキーに対して、各スレッドは特定の値を結び付けることができます。その値はスレッドに固有で、スレッドごとに独立に維持管理されます。スレッド単位での割り当ては、キーがデストラクタ関数 (destructor()) で作成された場合は、スレッドの終了時にその割り当てを解除されます。

```
プロトタイプ:
int pthread_key_create(pthread_key_t *key,
    void (*destructor) (void *));

#include <pthread.h>

pthread_key_t key;
int ret;

/* デストラクタを指定しないキーの作成 */
ret = pthread_key_create(&key, NULL);

/* デストラクタを指定したキーの作成 */
ret = pthread_key_create(&key, destructor);
```

pthread_key_create() が正常終了すると、割り当てられたキーは *key* が指す位置に格納されます。このキーに対する記憶領域とアクセスとの同期は呼び出し側の責任でとらなければなりません。

各キーに任意で、デストラクタ関数を関連付けることができます。あるキーが NULL でないデストラクタ関数を持っていて、スレッドがそのキーに対して NULL 以外の値を関連付けている場合、そのスレッドの終了時に現在関連付けられている

値を指定してデストラクタ関数が呼び出されます。どの順番でデストラクタ関数が呼び出されるかは不定です。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると pthread_key_create() は失敗し、次の値を戻します。

EAGAIN

キーの名前空間が使い果たされました。

ENOMEM

仮想記憶が足りないので、新しいキーを作成できません。

スレッド固有データキーの削除

pthread_key_delete(3THR)

pthread_key_delete(3THR) は、既存のスレッド固有データキーを削除します。キーに関連付けられているどのメモリーも解放できます。これはキーが無効で、参照されるとエラーが戻されるためです。Solaris スレッドには、これに相当する関数はありません。

```
プロトタイプ:
int pthread_key_delete(pthread_key_t key);

#include <pthread.h>

pthread_key_t key;
int ret;

/* 前に作成されたキー */
ret = pthread_key_delete(key);
```

キーが削除された後、pthread_setspecific() または pthread_getspecific() 呼び出しでそのキーが参照されると、EINVAL エラーが戻されます。

削除関数を呼び出す前にスレッド固有のリソースを解放するのは、プログラマの責任です。この関数はデストラクタ関数をいっさい呼び出しません。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると pthread_key_create() は失敗し、対応する値を返します。

EINVAL

key の値が有効ではありません。

スレッド固有データキーの設定

pthread_setspecific(3THR)

pthread_setspecific(3THR) は、スレッド固有な割り当てを、指定したスレッド固有データキーに設定します。

```
プロトタイプ:
int pthread_setspecific(pthread_key_t key, const void *value);

#include <pthread.h>

pthread_key_t key;
void *value;
int ret;

/* 前に作成されたキー */
ret = pthread_setspecific(key, value);
```

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると pthread_setspecific() は失敗し、対応する値を返します。

ENOMEM

仮想記憶が足りません。

EINVAL

キーが無効です。

注 - pthread_setspecific() スレッドがすでに使用しているキーに対してそのスレッドの新しい割り当てを設定した場合は、メモリーリークが発生する可能性があります。

スレッド固有データキーの取得

pthread_getspecific(3THR)

pthread_getspecific(3THR) は、*key* についての呼び出しスレッドの割り当てを取得し、それを *value* が指している記憶場所に格納します。

```
プロトタイプ:
void * pthread_getspecific(pthread_key_t key);

#include <pthread.h>

pthread_key_t key;
void *value;

/*  前に作成されたキー */
value = pthread_getspecific(key);
```

戻り値

エラーは戻されません。

スレッド固有データの広域性と局所性の例

例 2–2 は、あるマルチスレッドプログラムからの抜粋です。このコードは任意の数のスレッドによって実行されますが、2 つの広域変数 *errno* と *mywindow* は、実際には各スレッドにとって局所的な変数として参照されます。

例 2–2　スレッド固有データの広域性と局所性

```
body() {
    ...

    while (write(fd, buffer, size) == -1) {
        if (errno != EINTR) {
            fprintf(mywindow, "%s\n", strerror(errno));
            exit(1);
        }
    }

    ...

}
```

errno を参照すれば、そのスレッドが呼び出したルーチンから戻されたシステムエラーコードがわかります。他のスレッドが呼び出したシステムコールではありません。つまり、スレッドによる errno の参照は、スレッドごとに異なる記憶領域を参照します。

変数 *mywindow* は、それを参照するスレッドの専用のウィンドウに接続される stdio ストリームを参照するための変数です。errno と同様、スレッドによる *mywindow* の参照は、スレッドごとに異なる記憶領域、つまり異なるウィンドウを参照します。唯一の違いは、errno はスレッドライブラリが面倒を見てくれるのに対し、*mywindow* はプログラマが自分で管理しなければならないことです。

例 2–3 は、*mywindow* の参照がどのように働くかを示しています。プリプロセッサは、*mywindow* の参照を _mywindow() 手続きの呼び出しに変換します。

このルーチンは pthread_getspecific() を呼び出し、広域変数 *mywindow_key* (これは実際の広域変数) と出力用のパラメタ *win* を渡します。*win* には、そのスレッドのウィンドウの識別子が戻されます。

例 2–3　広域参照から局所参照への変換

```
thread_key_t mywin_key;

FILE *_mywindow(void) {
    FILE *win;

    pthread_getspecific(mywin_key, &win);
    return(win);
}
```

(続く)

```
#define mywindow _mywindow()

void routine_uses_win( FILE *win) {
    ...
}

void thread_start(...) {
    ...
    make_mywin();
    ...
    routine_uses_win( mywindow )
    ...
}
```

変数 *mywin_key* は、スレッド毎に実体を持つことができる変数のまとまりを識別します。つまり、これらの変数はスレッド固有データです。各スレッドは make_mywin() を呼び出し、そこで自分専用のウィンドウを初期化し、参照用に *mywindow* の自分専用のインスタンスを配置します。

make_mywin() を呼び出したスレッドは、*mywindow* を安全に参照できるようになり、さらに _mywindow() の実行後は、自分専用のウィンドウを参照できるようになります。結果的に、*mywindow* の参照は、そのスレッドの専用のデータの直接の参照であるかのように見えます。

例 2–4 は、以上の処理を示しています。

例 2–4 スレッド固有データの初期化

```
void make_mywindow(void) {
    FILE **win;
    static pthread_once_t mykeycreated = PTHREAD_ONCE_INIT;

    pthread_once(&mykeycreated, mykeycreate);

    win = malloc(sizeof(*win));
    create_window(win, ...);

    pthread_setspecific(mywindow_key, win);

 }

void mykeycreate(void) {
    pthread_keycreate(&mywindow_key, free_key);

 }
```

(続く)

```
void free_key(void *win) {
    free(win);
}
```

まず最初に、*mywin_key* キーに一意的な値を取得します。これはスレッド固有データのクラスを識別するために使用するキーです。具体的には、make_mywin() を呼び出す最初のスレッドが pthread_key_create() を呼び出します。その結果、この関数の第 1 引数に一意なキーが割り当てられます。第 2 引数はデストラクタ関数で、このスレッド固有データ項目のスレッド専用インスタンスをスレッドの終了時に解放するためのものです。

次に、呼び出し側の、このスレッド固有データ項目のインスタンスのために記憶領域を確保します。記憶領域を確保した後、create_window() ルーチンが呼び出されます。このルーチンでは、スレッドのためにウィンドウを設定し、そのウィンドウを参照するために *win* の指す記憶領域を設定します。最後に pthread_setspecific() が呼び出され、*win* 内の値 (つまり、ウィンドウの参照が格納されている記憶領域の位置) とキーとが結び付けられます。

その後、スレッドは pthread_getspecific() を呼び出して上記の広域キーを渡します。その結果、スレッドが pthread_setspecific() を呼び出して、このキーに関連付けた値を取得できます。

スレッドが終了するときは、pthread_key_create() で設定したデストラクタ関数が呼び出されます。各デストラクタ関数は、そのスレッドが pthread_setspecific() でキーに値を設定している場合だけ呼び出されます。

スレッド識別子の取得

pthread_self(3THR)

pthread_self(3THR) を使用して、呼び出しスレッドのスレッド識別子 (thread identifier) を取得します。

```
プロトタイプ:
pthread_t  pthread_self(void);

#include <pthread.h>

pthread_t tid;

tid = pthread_self();
```

戻り値

呼び出しスレッドのスレッド識別子 (thread identifier) が戻されます。

スレッド識別子の比較

pthread_equal(3THR)

pthread_equal(3THR) は、2 つのスレッドのスレッド識別番号を比較します。

```
プロトタイプ:
int  pthread_equal(pthread_t tid1, pthread_t tid2);

#include <pthread.h>

pthread_t tid1, tid2;
int ret;

ret = pthread_equal(tid1, tid2);
```

戻り値

tid1 と *tid2* が等しければ 0 以外の値が戻されます。そうでなければ、0 が戻されます。*tid1* または *tid2* が無効なスレッド識別番号の場合は、結果は予測できません。

スレッドの初期化

pthread_once(3THR)

pthread_once(3THR) は、初めて呼び出されたときに初期化ルーチンを呼び出します。2 回目以降の pthread_once() 呼び出しは何の効果もありません。

```
プロトタイプ:
int  pthread_once(pthread_once_t *once_control,
    void (*init_routine)(void));

#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;
int ret;

ret = pthread_once(&once_control, init_routine);
```

once_control パラメタは、該当する初期化ルーチンがすでに呼び出されているかどうかを判定します。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると pthread_once() は失敗し、対応する値を返します。

EINVAL

once_control または *init_routine* が NULL です。

スレッドの実行明け渡し

sched_yield(3RT)

sched_yield(3RT) は、現在のスレッドから同じ優先順位か、より高い優先順位をもつ別のスレッドに実行権を譲ります。

```
プロトタイプ:
int  sched_yield(void);

#include <sched.h>

int ret;

ret = sched_yield();
```

戻り値

正常終了時は 0 です。そうでなければ -1 が戻され、errno にエラー条件が設定されます。

ENOSYS

この実装では、sched_yield(3R) はサポートされていません。

スレッド優先順位の設定

pthread_setschedparam(3THR)

pthread_setschedparam(3THR) は、既存のスレッドの優先順位を変更します。この関数はスケジューリング方針には影響を与えません。

```
プロトタイプ:
int  pthread_setschedparam(pthread_t tid, int policy,
    const struct sched_param *param);

#include <pthread.h>

pthread_t tid;
int ret;
struct sched_param param;
int priority;

/* sched_priority がスレッドの優先順位になる */
sched_param.sched_priority = priority;

/* サポートされている方針のみ。それ以外は ENOTSUP を生じる */
policy = SCHED_OTHER;

/* 対象スレッドのスケジューリングパラメタ */
ret = pthread_setschedparam(tid, policy, &param);
```

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

設定しようとした属性の値が無効です。

ENOTSUP

サポートされていない属性値を設定しようとしました。

スレッド優先順位の取得

pthread_getschedparam(3THR)

pthread_getschedparam(3THR) は、既存のスレッドの優先順位を取得します。

```
プロトタイプ:
int  pthread_getschedparam(pthread_t tid, int policy,
    struct schedparam *param);

#include <pthread.h>

pthread_t tid;
sched_param param;
int priority;
int policy;
int ret;

/* 対象スレッドのスケジューリングパラメタ */
ret = pthread_getschedparam (tid, &policy, &param);

/* sched_priority にスレッドの優先順位が含まれる */
priority = param.sched_priority;
```

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、対応する値を返します。

ESRCH

tid で指定した値が既存のスレッドを表していません。

シグナルのスレッドへの送信

pthread_kill(3THR)

pthread_kill(3THR) は、スレッドにシグナルを送ります。

```
プロトタイプ:
int  pthread_kill(thread_t tid, int sig);

#include <pthread.h>
#include <signal.h>


int sig;
pthread_t tid;
int ret;

ret = pthread_kill(tid, sig);
```

tid で指定したスレッドに *sig* で指定したシグナルを送ります。*tid* は、呼び出しスレッドと同じプロセス内のスレッドでなければなりません。引数 *sig* は、`signal(5)` のリスト中の値でなければなりません。

sig が 0 のときはエラーチェックだけが行われ、シグナルは実際には送られません。これにより *tid* で指定したスレッド識別子が有効であるかどうかを調べることができます。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると `pthread_kill()` は失敗し、次の値が戻されます。

`EINVAL`

sig は正しいシグナル番号ではありません。

`ESRCH`

現在のプロセス内で *tid* で指定したスレッドが見つかりません。

呼び出しスレッドのシグナルマスクの変更

pthread_sigmask(3THR)

`pthread_sigmask(3THR)` は、呼び出しスレッドのシグナルマスクの変更や照会を行います。

```
プロトタイプ:
int pthread_sigmask(int how, const sigset_t *new, sigset_t *old);

#include <pthread.h>
#include <signal.h>


int ret;
sigset_t old, new;

ret = pthread_sigmask(SIG_SETMASK, &new, &old); /* 新しいマスクを設定する */
ret = pthread_sigmask(SIG_BLOCK, &new, &old); /* マスクをブロックする */
ret = pthread_sigmask(SIG_UNBLOCK, &new, &old); /* マスクのブロックを解除する */
```

引数 *how* は、シグナルマスクの変更方法を指定します。以下のいずれかの値を指定できます。

- SIG_BLOCK — *new* で指定したシグナルを現在のシグナルマスクに追加します。*new* はブロックしようとするシグナルの集合です。
- SIG_UNBLOCK — *new* で指定したシグナルを現在のシグナルマスクから削除します。*set* はブロックを解除しようとするシグナルの集合です。
- SIG_SETMASK — 現在のシグナルマスクを *new* で指定したシグナルに置き換えます。*new* は新しいシグナルマスクを示します。

new の指定が NULL の場合、*how* の値は無視され、スレッドのシグナルマスクは変更されません。現在ブロックされているシグナルを照会するときは、引数 *new* の値に NULL を指定してください。

old の指定が NULL でなければ、*old* の指すアドレスに変更前のシグナルマスクが格納されます。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると pthread_sigmask() は失敗し、次の値が戻されます。

EINVAL

how の値が定義されていません。

安全な fork

pthread_atfork(3THR)

169ページの「解決策 − pthread_atfork(3T)」の pthread_atfork(3THR) の説明を参照してください。

```
プロトタイプ:

int pthread_atfork(void (*prepare) (void), void (*parent) (void),
    void (*child) (void) );
```

スレッドの終了

pthread_exit(3THR)

pthread_exit(3THR) は、スレッドを終了させます。

```
プロトタイプ:
void  pthread_exit(void *status);

#include <pthread.h>

int status;

pthread_exit(&status); /* status を示して終了 */
```

pthread_exit() は呼び出しスレッドを終了させます。スレッド固有に割り当てられているデータもすべて解放されます。スレッドが切り離されていない場合は、そのスレッドに対する (ブロックされた) 終了待ちが行われるまで、そのスレッド識別子と *status* により示される終了状態は保持されます。(ブロック化された) スレッドが切り離されている場合は、*status* は無視され、そのスレッド識別子がただちに再利用できるようになります。スレッドの切り離しについては、63ページの「切り離し状態の設定」を参照してください。

戻り値

status の指定が NULL でなければ、呼び出しスレッドが終了すると、終了状態が *status* の内容に設定されます。

スレッド終了処理の完了

スレッドの終了には下記の方法があります。

- 最初の (一番外側の) 手続きであるスレッド起動ルーチンから戻る (pthread_create(3T) のマニュアルページを参照)
- pthread_exit(3T) を呼び出して、終了状態を指定する
- POSIX 取り消し関数によって終了する (pthread_cancel(3T) のマニュアルページを参照)

デフォルトでは、他のスレッドが当該スレッドに対して「終了待ち」を行い、その消滅を確認するまでの間、スレッドは残存します。これはデフォルトの pthread_create() 生成属性の「切り離されていない」と同じです (詳細は、pthread_detach(3T) のマニュアルページを参照してください)。「終了待ち」操作が行われると当該スレッドの終了状態が取得され、その後、当該スレッドが消滅します。

特に注意すべき特別な場合があります。メインスレッド (すなわち、main() を呼んでいるもの) が main() 呼び出しから戻るか、exit(3C) を呼び出す場合です。この操作が行われるとプロセス全体が終了し、プロセス内のスレッドもすべて終了してしまいます。このため、メインスレッドが main() から処理途中で戻ることがないよう十分注意しなければなりません。

メインスレッドが単に pthread_exit(3T) を呼び出した場合は、メインスレッドが終了するだけです。プロセス内の他のスレッドとプロセスは、その後も存続します。(すべてのスレッドが終了するとプロセスは終了します。)

取り消し

POSIX スレッドは、スレッドプログラミングに取り消し可能性 (取り消し機能) という考え方を導入しました。取り消し機能を使用することによって、スレッドはそのプロセスの他の任意のスレッドまたは全スレッドを終了させることができます。関連のある一群のスレッドの以降の操作がすべて有害または不必要な状況では、取り消しは 1 つの有効な方法です。好ましい方法は、すべてのスレッドを取り消し、そのプロセスを矛盾のない状態に戻してから起点まで戻ることです。

スレッドの取り消しの例としては、非同期的に生成される取り消し条件、たとえば実行中のアプリケーションを閉じるまたは終了するというユーザの要求などがあります。また、複数のスレッドが関わっているタスクの完了などもあります。最終的にスレッドの 1 つがそのタスクを完了させたのに、他のスレッドが動作し続けてい

る場合は、その時点でそれらのスレッドは何の役にも立っていないため、すべて取り消したほうがよいでしょう。

取り消しには危険が伴います。そのほとんどは、不変式の復元と共有リソースの解放処理に関係します。不注意に取り消されたスレッドは mutex をロック状態のままにすることがあり、その場合はデッドロックを引き起こします。あるいは、どこか特定できないメモリー領域を割り当てられたままにすることもあるので解放できなくなります。

pthread ライブラリでは、取り消しをプログラムにより許可したり禁止したりする取り消しインタフェースを規定しています。ライブラリでは、どの点で取り消しが可能かを示す一群のポイント (取り消しポイント) も定義しています。さらに、取り消しハンドラ (クリーンアップサービスを提供する) の有効範囲を定義して、意図した時と場所に確実に働くようにできます。

取り消しポイントの配置と取り消しハンドラの効果は、アプリケーションに対する理解に基づくものでなければなりません。mutex は明らかに取り消しポイントではないので、ロックしている時間は必要最小限に留めるべきです。

非同期取り消しの領域は、宙に浮いたリソースや未解決の状態を生じさせるような外部に依存しないシーケンスに限定してください。入れ子の代替取り消し状態から復帰するときは、取り消し状態を復元するように注意してください。このインタフェースは、復元を容易に行えるように次の機能を提供しています。pthread_setcancelstate(3T) は、参照される変数の中に現在の取り消し状態を保存します。pthread_setcanceltype(3T) は、これと同じ方法で現在の取り消しタイプを保存します。

取り消しが起こりうる状況は、次の 3 通りです。

- 非同期に
- 実行シーケンス中の、この規格で定義されているさまざまなポイントで
- アプリケーションで指定された個々のポイントで

デフォルトでは、取り消しが起こりうるのは POSIX 規格で定義されているような、明確に定義されたポイントに限られます。

いずれの場合も、リソースと状態が起点と矛盾しない状態に復元されるように注意してください。

取り消しポイント

スレッドの取り消しは、取り消しが安全な場合にだけ行なってください。pthread 規格では、下記のような取り消しポイントが規定されています。

- pthread_testcancel(3T) 呼び出しを通してプログラムで設定されるスレッドの取り消しポイント
- pthread_cond_wait(3T) または pthread_cond_timedwait(3T) で特定の条件の発生を待っているスレッド
- pthread_join(3T) で他のスレッドの終了を待っているスレッド
- sigwait(2) でブロックされたスレッド
- ある種の標準ライブラリコール。通常、これらはスレッドがブロックできる関数です。詳細は、cancellation(3T) のマニュアルページを参照してください。

デフォルトでは、取り消しが有効 (使用可能) です。アプリケーションで取り消しを無効 (使用不可) にした場合は、再び有効にするまで、すべての取り消し要求が据え置かれます。

取り消しを無効にする方法については、55ページの「pthread_setcancelstate(3THR)」を参照してください。

スレッドの取り消し

pthread_cancel(3THR)

pthread_cancel(3THR) は、スレッドを取り消します。

プロトタイプ:

```
int pthread_cancel(pthread_t thread);

#include <pthread.h>

pthread_t thread;
int ret;

ret = pthread_cancel(thread);
```

取り消し要求がどのように扱われるかは、対象となるスレッドの状態によって異なります。その状態を判定する関数として、pthread_setcancelstate(3T) と pthread_setcanceltype(3T) の 2 つがあります。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、対応する値を返します。

ESRCH

指定されたスレッド ID に対応するスレッドが見つかりません。

取り消しを有効または無効にする

pthread_setcancelstate(3THR)

pthread_setcancelstate(3THR) は、スレッドの取り消し機能を有効 (使用可能) または無効 (使用不可) にします。スレッドが生成されると、デフォルトでは取り消し機能が有効になります。

```
プロトタイプ:

int pthread_setcancelstate(int state, int *oldstate);

#include <pthread.h>

int oldstate;
int ret;

/* 有効にする */
ret = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);

/* 無効にする */
ret = pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);
```

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると pthread_setcancelstate() は失敗し、対応する値を戻します。

EINVAL

状態が PTHREAD_CANCEL_ENABLE でも PTHREAD_CANCEL_DISABLE でもありません。

取り消しタイプの設定

pthread_setcanceltype(3THR)

pthread_setcanceltype(3THR) は、取り消しタイプを遅延モードまたは非同期モードに設定します。スレッドが生成されると、デフォルトでは取り消しタイプが遅延モードに設定されます。遅延モードにあるスレッドは、取り消しポイント以外では取り消すことができません。非同期モードにあるスレッドは、実行中の任意のポイントで取り消すことができます。非同期モードを使用するのは好ましくありません。

```
プロトタイプ:

int pthread_setcanceltype(int type, int *oldtype);

#include <pthread.h>

int oldtype;
int ret;

/* 遅延モード */
ret = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &oldtype);

/* 非同期モード */
ret = pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &oldtype);
```

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

PTHREAD_CANCEL_DEFERRED または PTHREAD_CANCEL_ASYNCHRONOUS タイプではありません。

取り消しポイントの設定

pthread_testcancel(3THR)

pthread_testcancel(3THR) は、スレッドの取り消しポイントを設定します。

```
プロトタイプ:

void pthread_testcancel(void);

#include <pthread.h>

pthread_testcancel();
```

pthread_testcancel() 関数が実際に機能するのは、取り消し機能が有効にされていて、しかも遅延モードになっているときです。取り消し機能が無効になっている状態で、この関数を呼び出しても何の効果もありません。

pthread_testcancel() を挿入するのは、スレッドを取り消しても安全なシーケンスに限定してください。pthread_testcancel() 呼び出しを通してプログラムで設定される取り消しポイントの他にも、pthread 規格では、いくつかの取り消しポイントが規定されています。詳細は、54ページの「取り消しポイント」を参照してください。

戻り値はありません。

スタックへハンドラをプッシュする

クリーンアップハンドラは、割り当てられたリソースのクリーンアップや不変式の復元など、諸条件を起点のものと矛盾しない状態に復元するためのものです。クリーンアップハンドラの管理には、pthread_cleanup_push(3T) 関数と pthread_cleanup_pop(3T) 関数を使用します。

クリーンアップハンドラは、プログラムの同じ字句解析上の範囲でプッシュされてポップされます。プッシュとポップは、常に対になっていなければなりません。そうでないと、コンパイルエラーになります。

pthread_cleanup_push(3THR)

pthread_cleanup_push(3THR) は、クリーンアップハンドラをクリーンアップスタック (LIFO) にプッシュします。

```
プロトタイプ:

void pthread_cleanup_push(void(*routine)(void *), void *args);

#include <pthread.h>

/* ハンドラ「routine」をクリーンアップスタックにプッシュする */
pthread_cleanup_push (routine, arg);
```

スタックからハンドラを取り出す

pthread_cleanup_pop(3THR)

pthread_cleanup_pop(3THR) は、クリーンアップハンドラをクリーンアップスタックから取り出します。

この関数への引数が 0 以外なら、指定のハンドラがスタックから取り除かれて実行されます。引数が 0 の場合は、ハンドラはポップされるだけで実行されません。

0 以外の引数を指定して pthread_cleanup_pop() を有効に呼び出せるのは、スレッドが pthread_exit(3T) を明示的または暗黙的に呼び出した場合か、取り消し要求を受け付けた場合です。

```
プロトタイプ:
void pthread_cleanup_pop(int execute);

#include <pthread.h>

/* 「func」をクリーンアップスタックからポップし、「func」を実行する */
pthread_cleanup_pop (1);

/* 「func」をポップするが、「func」を実行しない */
pthread_cleanup_pop (0);
```

戻り値はありません。

第 3 章

スレッド生成時の属性設定

前章では、デフォルト属性を使ったスレッド生成の基礎について説明しました。この章では、スレッド生成時における属性の設定方法を説明します。

なお、pthread だけが属性と取り消しを使用するので、この章で取り上げている API は POSIX スレッドのみに対応します。それ以外は、Solaris スレッドと pthread は機能的にはほぼ同じです。(両者の類似点と相違点については、第 8 章を参照してください。)

- 75ページの「スケジューリングパラメタの設定」
- 76ページの「スケジューリングパラメタの取得」
- 77ページの「スタックの大きさの設定」
- 79ページの「スタックの大きさの取得」
- 81ページの「スタックアドレスの設定」
- 83ページの「スタックアドレスの取得」

属性

属性は、デフォルトとは異なる動作を指定する手段です。pthread_create(3T) でスレッドを生成する場合または同期変数を初期化する場合は、属性オブジェクトを指定できます。通常は、デフォルトで間に合います。

注 - 属性はスレッド生成時にのみ指定されます。スレッドを使用中は変更できません。

属性オブジェクトはプログラマからは「不透明」なため、代入によって直接変更できません。各オブジェクト型を初期化、設定、または削除するための関数のセットが用意されています。

いったん初期化して設定した属性は、プロセス全体に適用されます。属性を使用するための望ましいやり方は、必要なすべての状態の指定をプログラム実行の初期の段階で一度に設定することです。そうすれば、必要に応じて適切な属性オブジェクトを参照できます。

属性オブジェクトを使用することには、主に次の 2 つの利点があります。

- 第 1 に、コードの移植性が高まります。

  サポートされる属性は実装によって異なっていても、属性オブジェクトはインタフェースから隠されているので、スレッド実体を生成するための関数呼び出しを変更する必要はありません。

  移植の対象となる実装が、現在の実装にない属性をサポートしている場合は、新しい属性を管理するために準備が必要です。ただし、属性オブジェクトは明確に定義された位置で一度だけ初期化すればよいので、この移植作業は難しくはありません。

- 第 2 に、アプリケーションでの状態指定が簡素化されます。

一例として、同じプロセス内にスレッドの集合がいくつか存在し、それぞれが別のサービスを提供するとともに独自の状態要件をもっているという状況を考えてみます。

アプリケーションの初期段階のどこかの時点で、1 つのスレッドの属性オブジェクトを集合ごとに初期化できます。以降のすべてのスレッド生成は、そのタイプのスレッドについて初期化された属性オブジェクトを参照します。初期化フェーズは単純で現地仕様化されているので、後で変更が必要になっても、すばやく確実に実行できます。

属性オブジェクトの取り扱いで注意を要するのは、プロセス終了時です。オブジェクトが初期化されるときにメモリーが割り当てられます。このメモリーをシステムに戻す必要があります。pthread 規格には、属性オブジェクトを削除する関数呼び出しが用意されています。

属性の初期化

pthread_attr_init(3THR)

pthread_attr_init(3THR) は、オブジェクトの属性をデフォルト値に初期化します。その記憶領域は、実行中にスレッドシステムによって割り当てられます。

プロトタイプ:

```
int pthread_attr_init(pthread_attr_t *tattr);

#include <pthread.h>

pthread_attr_t tattr;
int ret;

/* 属性をデフォルト値に初期化する */
ret = pthread_attr_init(&tattr);
```

表 3–1 に属性 (*tattr*) のデフォルト値を示します。

表 3–1 *tattr* のデフォルト属性値

属性	値	結果
scope	PTHREAD_SCOPE_PROCESS	新しいスレッドは非結合 (LWP に固定的に結合されない)
detachstate	PTHREAD_CREATE_JOINABLE	スレッドの終了後に終了状態とスレッドが保存される
stackaddr	NULL	新しいスレッドはシステムによって割り当てられたスタックアドレスをもつ
stacksize	1M バイト	新しいスレッドはシステムによって定義されたスタックの大きさをもつ
priority		新しいスレッドは親スレッドの優先順位を継承する
inheritsched	PTHREAD_INHERIT_SCHED	新しいスレッドは親スレッドのスケジューリング優先順位を継承する
schedpolicy	SCHED_OTHER	新しいスレッドは Solaris で定義された固定的な優先順位スケジューリングを使用する。スレッドは、優先順位の高いスレッドに取って代わられるまで、あるいはブロックするか実行権を明け渡すまで動作する

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、対応する値を返します。

ENOMEM

メモリーが不足し、スレッド属性オブジェクトを初期化できないときに返されます。

属性の削除

pthread_attr_destroy(3THR)

pthread_attr_destroy(3THR) は、初期化時に割り当てられた記憶領域を削除します。その属性オブジェクトは無効になります。

```
プロトタイプ:
int pthread_attr_destroy(pthread_attr_t *tattr);

#include <pthread.h>

pthread_attr_t tattr;
int ret;

/* 属性を削除する */
ret = pthread_attr_destroy(&tattr);
```

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

tattr の値が無効です。

切り離し状態の設定

pthread_attr_setdetachstate(3THR)

スレッドを切り離された状態 (PTHREAD_CREATE_DETACHED) として生成すると、そのスレッドが終了するとすぐに、そのスレッド識別子とその他のリソースを再利用できます。呼び出したスレッドでスレッドの終了まで待ちたくない場合は、pthread_attr_setdetachstate(3THR) を使用してください。

スレッドを切り離されていない状態 (PTHREAD_CREATE_JOINABLE) として生成すると、そのスレッドを待つものとみなされます。つまり、そのスレッドに対して pthread_join(3T) を実行するとみなされます。

スレッドが切り離された状態か切り離されていない状態で作成されたかに関係なく、すべてのスレッドが終了するまでプロセスは終了しません。52ページの「スレッド終了処理の完了」にある、main() から処理途中で戻ることによって生じるプロセスの終了の説明を参照して下さい。

```
プロトタイプ:

int pthread_attr_setdetachstate(pthread_attr_t *tattr,int detachstate);

#include <pthread.h>

pthread_attr_t tattr;
int ret;

/* スレッド切り離し状態を設定する */
ret = pthread_attr_setdetachstate(&tattr,PTHREAD_CREATE_DETACHED);
```

注 - 明示的な同期によって阻止されなければ、新たに生成される切り離されたスレッドは、そのスレッドの生成元が pthread_create() から復帰する前に終了でき、そのスレッド識別子は別の新しいスレッドに割り当てることができます。

切り離されていない (PTHREAD_CREATE_JOINABLE) スレッドについては、そのスレッドの終了後に他のスレッドが終了待ちを行うことがきわめて重要です。そうしないと、そのスレッドのリソースが新しいスレッドに解放されません。これは通常、メモリーリークを招くことになります。終了待ちを行うつもりがない場合は、スレッド作成時に切り離されたスレッドとして作成してください。

例 3–1 切り離されたスレッドの生成

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
void *start_routine;
void arg
int ret;

/* デフォルト属性で初期化する */
ret = pthread_attr_init()(&tattr);
ret = pthread_attr_setdetachstate()(&tattr,PTHREAD_CREATE_DETACHED);
ret = pthread_create()(&tid, &tattr, start_routine, arg);
```

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

detachstate または *tattr* の値が無効です。

切り離し状態の取得

pthread_attr_getdetachstate(3THR)

pthread_attr_getdetachstate(3THR) は、スレッドの生成状態を取得します。これは「切り離された」または「切り離されていない」状態です。

```
プロトタイプ:

int pthread_attr_getdetachstate(const pthread_attr_t *tattr,
    int *detachstate;

#include <pthread.h>

pthread_attr_t tattr;
int detachstate;
int ret;

/* スレッドの切り離し状態を取得する */
ret = pthread_attr_getdetachstate (&tattr, &detachstate);
```

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

detachstate の値が NULL か、*tattr* の値が無効です。

スタックガードの大きさの設定

pthread_attr_setguardsize(3THR)

pthread_attr_setguardsize(3THR) は、*attr* オブジェクトの *guardsize* (ガードサイズ) を設定します。

guardsize 引数は、スタックポインタのオーバーフローを防ぐためのものです。ガードとともにスレッドのスタックが作成されると、実装は、スタックのオーバーフローの終わりに、スタックポインタのスタックオーバーフローの緩衝域として、余分のメモリーを割り当てます。このバッファにアプリケーションがオーバーフローすると、スレッドに SIGSEGV シグナルが配信されるなどのエラーが発生します。

ガードサイズ属性をアプリケーションで使用する目的は、次の 2 つです。

1. オーバーフローを防止すると、システムリソースが無駄になるおそれがあります。多くのスレッドが作成されるアプリケーションは、そのスレッドがスタックをオーバーフローしないことがわかっている場合には、ガード領域をオフにすることで、システムリソースを節約できます。
2. スレッドがスタックに割り当てたデータ構造が大きい場合は、スタックオーバーフローを検出するために、大きなガード領域が必要になることがあります。

guardsize が 0 の場合は、*attr* を使って作成したスレッドにはガード領域が含まれません。*guardsize* が 0 よりも大きい場合は、少なくとも *guardsize* バイトのガード領域が、*attr* を使って作成した各スレッドに割り当てられます。デフォルトでは、スレッドは実装で定義された 1 バイト以上のガード領域を持ちます。

POSIX では、*guardsize* の値を、設定可能なシステム変数 PAGESIZE (sys/mman.h の「PAGESIZE」を参照) の倍数に切り上げるように、実装が認められています。実装が *guardsize* の値を PAGESIZE の倍数に切り上げる場合は、*attr* を指定して pthread_attr_getguardsize() を呼び出すと、*guardsize* には前回 pthread_attr_setguardsize() を呼び出したときに指定されたガードサイズが格納されます。

```
#include <pthread.h>

int pthread_attr_setguardsize(pthread_attr_t *attr, size_t  guardsize);
```

戻り値

以下の戻り値は、pthread_attr_setguardsize() が失敗したことを示します。

EINVAL

引数 *attr* が無効であるか、引数 *guardsize* が無効であるか、あるいは *guardsize* に無効な値が含まれています。

スタックガードの大きさの取得

pthread_attr_getguardsize(3THR)

pthread_attr_getguardsize(3THR) は、*attr* オブジェクトの *guardsize* を取得します。

POSIX では、*guardsize* の値を、設定可能なシステム変数 PAGESIZE (sys/mman.h の「PAGESIZE」を参照) の倍数に切り上げる実装が認められています。実装が *guardsize* の値を PAGESIZE の倍数に切り上げる場合は、*attr* を指定して pthread_attr_getguardsize() を呼び出すと、*guardsize* には前回 pthread_attr_setguardsize() を呼び出したときに指定されたガードサイズが使用されます。

```
#include <pthread.h>

int pthread_attr_getguardsize(const pthread_attr_t *attr, size_t  *guardsize);
```

戻り値

以下の戻り値は、pthread_attr_getguardsize() が失敗したことを示します。

EINVAL

引数 *attr* が無効であるか、引数 *guardsize* が無効であるか、あるいは *guardsize* に無効な値が含まれています。

スコープの設定

pthread_attr_setscope(3THR)

pthread_attr_setscope(3THR) は、結合スレッド (PTHREAD_SCOPE_SYSTEM) または非結合スレッド (PTHREAD_SCOPE_PROCESS) を生成します。

注 - 結合スレッドと非結合スレッドの両方とも、指定されたプロセス内でのみアクセスできます。

```
プロトタイプ:

int pthread_attr_setscope(pthread_attr_t *tattr,int scope);

#include <pthread.h>

pthread_attr_t tattr;
int ret;

/* 結合スレッド */
ret = pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);

/* 非結合スレッド */
ret = pthread_attr_setscope(&tattr, PTHREAD_SCOPE_PROCESS);
```

この例には、属性を初期化するもの、デフォルト属性を変更するもの、pthread を生成するものの 3 つの関数呼び出しがあります。

```
#include <pthread.h>

pthread_attr_t attr;
pthread_t tid;
void start_routine;
void arg;
int ret;

/* デフォルト属性による初期化 */
ret = pthread_attr_init (&tattr);

/* 結合動作 */
ret =  pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);
ret = pthread_create (&tid, &tattr, start_routine, arg);
```

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

tattr の値は無効です。

スコープの取得

pthread_attr_getscope(3THR)

pthread_attr_getscope(3THR) は、スレッドのスコープを取得します。これはスレッドが結合するかしないかを示します。

```
プロトタイプ:

int pthread_attr_getscope(pthread_attr_t *tattr, int *scope);

#include <pthread.h>

pthread_attr_t tattr;
int scope;
int ret;

/* スレッドのスコープを取得する */
ret = pthread_attr_getscope(&tattr, &scope);
```

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

scope の値が NULL か、*tattr* の値が無効です。

スレッド多重度の設定

pthread_setconcurrency(3THR)

プロセス内の非結合スレッドは、同時にアクティブにする必要がある場合と、そうでない場合があります。デフォルトでは、スレッドの実装は、プロセスの処理を続行できる数のスレッドをアクティブにするように設定されています。デフォルト値のままにするとシステムリソースは節約できますが、最適の多重度ではない場合もあります。

pthread_setconcurrency(3THR) を使用すると、アプリケーションからスレッドの実装に、望ましい多重度 *new_level* を通知させることができます。この関数呼び

出しの結果として実装が提供する実際の多重度は、定義されていません (Solaris スレッドについては、227ページの「thr_setconcurrency(3THR)」参照)。

new_level が 0 の場合は、pthread_setconcurrency() が呼び出されなかったものとして、実装が任意の多重度を指定します。

アプリケーションは pthread_setconcurrency() を呼び出すときに、実装に目標多重度を通知します。実装はこの値を、要求ではなく参考として使用します。

```
#include <pthread.h>

int pthread_setconcurrency(int new_level);
```

戻り値

以下の戻り値は、pthread_setconcurrency() が失敗したことを示します。

EINVAL

new_level で指定された値が負の値です。

EAGAIN

new_level で指定された値を使用するとシステムリソースの容量を超えます。

スレッド多重度の取得

pthread_getconcurrency(3THR)

pthread_getconcurrency(3THR) は、pthread_setconcurrency() への前回の呼び出しで設定された値を返します。pthread_setconcurrency() 関数が呼び出されたことがない場合は、0 を返します。0 は、実装が多重度を指定したことを示します (Solaris スレッドについては、228ページの「thr_getconcurrency(3THR)」参照)。

```
#include <pthread.h>

int pthread_getconcurrency(void);
```

戻り値

pthread_getconcurrency() は常に、pthread_setconcurrency() の前回の呼び出しで設定された値を返します。pthread_setconcurrency() が呼び出されたことがない場合は、pthread_getconcurrency() は 0 を返します。

スケジューリング方針の設定

pthread_attr_setschedpolicy(3THR)

pthread_attr_setschedpolicy(3THR) は、スケジューリング方針を設定します。POSIX 規格の草稿ではスケジューリング方針の属性として、SCHED_FIFO (先入れ先出し)、SCHED_RR (ラウンドロビン)、SCHED_OTHER (実装で定義) を規定しています。

- SCHED_FIFO

  先入れ先出し。この方針でスケジュールしたスレッドは、優先順位の高いスレッドに割り込まれなければ、完了まで処理を進行します。スケジューリングの競合範囲がシステムであるスレッド (PTHREAD_SCOPE_SYSTEM) は、リアルタイム (RT) スケジューリングクラスに属し、呼び出しプロセスの実効ユーザ ID は 0 でなければいけません。スケジューリングの競合範囲がプロセス (PTHREAD_SCOPE_PROCESS) であるスレッド は、TS スケジューリングクラスに属します。

- SCHED_RR

  ラウンドロビン。この方針でスケジュールしたスレッドは、優先順位の高いスレッドに割り込まれなければ、システムによって定めれれた期間、処理を実行します。スケジューリングの競合範囲がシステムであるスレッド (PTHREAD_SCOPE_SYSTEM) は、リアルタイム (RT) スケジューリングクラスに属し、呼び出しプロセスの実効ユーザ ID は 0 でなければいけません。スケジューリングの競合範囲がプロセス (PTHREAD_SCOPE_PROCESS) であるスレッドの SCHED_RR は、TS スケジューリングクラスに属します。

SCHED_FIFO と SCHED_RR は POSIX では任意とされており、リアルタイム結合スレッドについてのみサポートされています

現在 pthread では、タイムシェアリングを示す Solaris の SCHED_OTHER のデフォルト値のみがサポートされています。スケジューリングの説明については、26ページの「スケジューリング」の節を参照してください。

```
プロトタイプ:

int pthread_attr_setschedpolicy(pthread_attr_t *tattr, int policy);

#include <pthread.h>

pthread_attr_t tattr;
int policy;
int ret;

/* スケジューリング方針を SCHED_OTHER に設定する */
ret = pthread_attr_setschedpolicy(&tattr, SCHED_OTHER);
```

戻り値

正常終了時は 0 です。それ以外の戻り値はエラーが発生したことを示します。以下のいずれかの条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

tattr の値は無効です。

ENOTSUP

属性をサポートされていない値に設定しようとしました。

スケジューリング方針の取得

pthread_attr_getschedpolicy(3THR)

pthread_attr_getschedpolicy(3THR) は、スケジューリング方針を取得します。現在 pthread では、Solaris ベースの SCHED_OTHER デフォルト値のみがサポートされています。

```
プロトタイプ:

int pthread_attr_getschedpolicy(pthread_attr_t *tattr, int *policy);

#include <pthread.h>

pthread_attr_t tattr;
int policy;
int ret;
```

(続く)

```
/* スレッドのスケジューリング方針を取得する */
ret = pthread_attr_getschedpolicy (&tattr, &policy);
```

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

policy の値が NULL か、*tattr* の値が無効です。

継承スケジューリング方針の設定

pthread_attr_setinheritsched(3THR)

pthread_attr_setinheritsched(3THR) は、継承スケジューリング方針を設定します。

継承 (*inherit*) 値の PTHREAD_INHERIT_SCHED (デフォルト) の意味は、生成スレッドで定義されたスケジューリング方針を使用し、pthread_create() 呼び出しで定義されたスケジューリング方針は無視するということです。PTHREAD_EXPLICIT_SCHED を使用した場合は、pthread_create() 呼び出しでの属性が使用されます。

```
プロトタイプ:

int pthread_attr_setinheritsched(pthread_attr_t *tattr, int inherit);

#include <pthread.h>

pthread_attr_t tattr;
int inherit;
int ret;

/* 現在のスケジューリング方針を使用する */
ret = pthread_attr_setinheritsched(&tattr, PTHREAD_EXPLICIT_SCHED);
```

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると、この関数は失敗し、対応する値を戻します。

EINVAL

tattr の値は無効です。

ENOTSUP

属性をサポートされていない値に設定しようとしました。

継承スケジューリング方針の取得

pthread_attr_getinheritsched(3THR)

pthread_attr_getinheritsched(3THR)
は、pthread_attr_setinheritsched() によって設定された、スケジューリング方針を返します。

```
プロトタイプ:

int pthread_attr_getinheritsched(pthread_attr_t *tattr, int *inherit);

#include <pthread.h>

pthread_attr_t tattr;
int inherit;
int ret;

/* 生成スレッドのスケジューリング方針を取得する */
ret = pthread_attr_getinheritsched (&tattr, &inherit);
```

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

inherit の値が NULL か、*tattr* の値が無効です。

スケジューリングパラメタの設定

pthread_attr_setschedparam(3THR)

pthread_attr_setschedparam(3THR) は、スケジューリングパラメタを設定します。

スケジューリングパラメタは param 構造体で定義します。ただし、サポートされるのは優先順位だけです。新たに生成されるスレッドは、この方針で動作します。

- SCHED_FIFO

  先入れ先出し。この方針でスケジュールしたスレッドは、優先順位の高いスレッドに割り込まれなければ、完了まで処理を進行します。スケジューリング競合範囲であるスレッド (PTHREAD_SCOPE_SYSTEM) は、リアルタイム (RT) スケジューリングクラスに属し、呼び出しプロセスの実効ユーザ ID は 0 でなければなりません。スケジューリング競合範囲がプロセス (PTHREAD_SCOPE_PROCESS) であるスレッドは、TS スケジューリングクラスに属します。

- SCHED_RR

  ラウンドロビン。この方針でスケジュールしたスレッドは、優先順位の高いスレッドに割り込まなければ、システムによって定められた期間、処理を実行します。スケジューリング競合範囲がシステムであるスレッド (PTHREAD_SCOPE_SYSTEM) は、リアルタイム (RT) スケジューリングクラスに属し、呼び出しプロセスの実効ユーザ ID は 0 でなければなりません。スケジューリング競合範囲がプロセス (PTHREAD_SCOPE_PROCESS) であるスレッドの SCHED_RR は、TS スケジューリングクラスに属します。

```
プロトタイプ:

int pthread_attr_setschedparam(pthread_attr_t *tattr,
    const struct sched_param *param);

#include <pthread.h>

pthread_attr_t tattr;
int newprio;
sched_param param;
newprio = 30;

/* 優先順位を設定する。それ以外は変更なし */
param.sched_priority = newprio;
```

(続く)

```
/* 新しいスケジューリングパラメタを設定する */
ret = pthread_attr_setschedparam (&tattr, &param);
```

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

param の値が NULL か、*tattr* の値が無効です。

pthread の優先順位は、子スレッドを生成する前に優先順位属性を設定するか、親スレッドの優先順位を変更してまた戻す、のいずれかの方法で管理できます。

スケジューリングパラメタの取得

pthread_attr_getschedparam(3THR)

pthread_attr_getschedparam(3THR) は、pthread_attr_setschedparam() によって設定されたスケジューリングパラメタを返します。

```
プロトタイプ:

int pthread_attr_getschedparam(pthread_attr_t *tattr,
    const struct sched_param *param);

#include <pthread.h>

pthread_attr_t attr;
struct sched_param param;
int ret;

/* 既存のスケジューリングパラメタを取得する */
ret = pthread_attr_getschedparam (&tattr, &param);
```

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、対応する値を戻します。

EINVAL

param の値が NULL か、*tattr* の値が無効です。

指定の優先順位をもつスレッドを生成する

スレッドを生成する前に優先順位属性を設定できます。子スレッドは、sched_param 構造体で指定した新しい優先順位で生成されます (この構造体には他のスケジューリング情報も含まれます)。

既存のパラメタを取得し、スレッドの優先順位を変更してから優先順位を再設定するという方法をお勧めします。

この方法の例を例 3–2 に示します。

例 3–2 優先順位を設定したスレッドの生成

```
#include <pthread.h>
#include <sched.h>

pthread_attr_t tattr;
pthread_t tid;
int ret;
int newprio = 20;
sched_param param;

/* デフォルト属性で初期化する */
ret = pthread_attr_init (&tattr);

/* 既存のスケジューリングパラメタを取得する */
ret = pthread_attr_getschedparam (&tattr, &param);

/* 優先順位を設定する。それ以外は変更なし */
param.sched_priority = newprio;

/* 新しいスケジューリングパラメタを設定する */
ret = pthread_attr_setschedparam (&tattr, &param);

/* 指定した新しい優先順位を使用する */
ret = pthread_create (&tid, &tattr, func, arg);
```

スタックの大きさの設定

pthread_attr_setstacksize(3THR)

pthread_attr_setstacksize(3THR) は、スレッドのスタックの大きさを設定します。

スタックサイズ属性は、システムが割り当てるスタックの大きさ (バイト数) を定義します。この大きさは、システムで定義された最小のスタックの大きさを下回ってはいけません。詳細は、79ページの「スタックについて」を参照してください。

```
プロトタイプ:

int pthread_attr_setstacksize(pthread_attr_t *tattr, int size);

#include <pthread.h>

pthread_attr_t tattr;
int size;
int ret;

size = (PTHREAD_STACK_MIN + 0x4000);

/* 新しい大きさを設定する */
ret = pthread_attr_setstacksize(&tattr, size);
```

上の例では、新しいスレッドが使用するスタックのバイト数が *size* に納められています。*size* の値が 0 ならば、デフォルトの大きさが使われます。ほとんどの場合、0 を指定すれば最善の結果が得られます。

PTHREAD_STACK_MIN は、スレッドを起動する上で必要なスタック空間の大きさです。しかし、アプリケーションコードを実行するのに必要なスレッドの関数が必要とするスタック空間の大きさは含まれていません。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

戻された値が PTHREAD_STACK_MIN より小さいか、システムの制限を超えているか、または *tattr* の値が無効です。

スタックの大きさの取得

pthread_attr_getstacksize(3THR)

pthread_attr_getstacksize(3THR) は、pthread_attr_setstacksize() によって設定された、スタックの大きさを返します。

```
プロトタイプ:

int pthread_attr_getstacksize(pthread_attr_t *tattr, size_t *size);

#include <pthread.h>

pthread_attr_t tattr;
int size;
int ret;

/* スタックの大きさを取得する */
ret = pthread_attr_getstacksize(&tattr, &size);
```

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

返された値が PTHREAD_STACK_MIN より小さいか、システムの制限を超えています。

スタックについて

通常、スレッドスタックはページ境界で始まり、指定した大きさは次のページ境界まで切り上げられます。アクセス権のないページがスタックの一番上に付加されることにより、ほとんどのスタックオーバーフローで、違反したスレッドに SIGSEGV シグナルが送られるようになります。呼び出し側によって割り当てられるスレッドスタックは、そのまま使われます。

スタックを指定するときは、スレッドを PTHREAD_CREATE_JOINABLE として生成してください。このスタックは、そのスレッドに対する pthread_join(3T) 呼び出しが戻るまで解放できません。これは、そのスレッドのスタックは、そのスレッ

ドが終了するまで解放できないからです。スレッドが終了したかどうかを確実に知るには、pthread_join(3T) を使用してください。

通常、スレッド用にスタック空間を割り当てる必要はありません。スレッドライブラリが、各スレッドのスタックとして 1M バイトの仮想記憶を割り当てます。このときスワップ空間は確保されません。(このライブラリは、mmap() の MAP_NORESERVE オプションを使って割り当てを行います。)

スレッドライブラリで生成される各スレッドスタックには、レッドゾーンがあります。スレッドライブラリはレッドゾーンとして、スタックオーバーフローを補足するためのページをスタックの一番上に付加します。このページは無効で、アクセスされるとメモリーフォルトになります。レッドゾーンは、自動的に割り当てられるすべてのスタックに付加されます。これは、その大きさがアプリケーションで指定されたかデフォルトの大きさかに関係なく行われます。

注 - 実行時のスタック要件は一定ではないので、指定したスタックがライブラリの呼び出しと動的リンクに必要な実行時要件を確実に満足するようにしなければなりません。

スタックとスタックの大きさの一方または両方を指定するのが適正であることはほとんどありません。専門家であっても、適切な大きさを指定したかどうかを判断するのは困難です。これは、ABI 準拠のプログラムでもスタックの大きさを静的に判定できないからです。スタックの大きさは、プログラムが実行される、それぞれの実行環境に左右されます。

独自のスタックを構築する

スレッドスタックの大きさを指定するときは、呼び出される関数に必要な割り当てを計算してください。これには、呼び出し手続きで必要とされる量、局所変数、情報構造体が含まれます。

デフォルトスタックと少し違うスタックが必要になることがあります。たとえば、スレッドで 1M バイトを超えるスタック空間が必要になる場合です。また、少し分かりにくいケースですが、デフォルトスタックが大きすぎる場合もあります。何千ものスレッドを生成するとすれば、デフォルトスタックでは合計サイズが数 G バイトにもなるため、仮想メモリが足りず、それだけのスタック空間を扱えないかもしれないからです。

スタックの大きさの上限は明らかであることが多いのですが、下限はどうでしょうか。スタックにプッシュされるスタックフレームを、その局所変数などを含めて、すべて扱えるだけのスタック空間が必要です。

マクロ PTHREAD_STACK_MIN を呼び出すと、スタックの大きさの絶対最小値が得られます。このマクロは、NULL 手続きを実行するスレッドに必要なスタック空間の大きさを戻します。実用的なスレッドに必要なスタック空間はもっと大きいので、スタックサイズを小さくするときは十分注意してください。

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
int ret;

int size = PTHREAD_STACK_MIN + 0x4000;

/* デフォルト属性で初期化する */
ret = pthread_attr_init(&tattr);

/* スタックの大きさも設定する */
ret = pthread_attr_setstacksize(&tattr, size);

/* tattr に大きさのみを指定する */
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

独自のスタックを割り当てるときは、その終わりにレッドゾーンを付加するために必ず mprotect(2) を呼び出してください。

スタックアドレスの設定

pthread_attr_setstackaddr(3THR)

pthread_attr_setstackaddr(3THR) は、スレッドスタックのアドレスを設定します。

stackaddr 属性は、スレッドのスタックのベースを定義するものです。これを NULL 以外の値に設定すると (NULL がデフォルト)、そのスタックはそのアドレスで初期化されます。

```
プロトタイプ:

int pthread_attr_setstackaddr(pthread_attr_t *tattr,void *stackaddr);

#include <pthread.h>

pthread_attr_t tattr;
void *base;
int ret;

base = (void *) malloc(PTHREAD_STACK_MIN + 0x4000);

/* 新しいアドレスを設定する */
ret = pthread_attr_setstackaddr(&tattr, base);
```

前の例では、新しいスレッドが使用するスタックのアドレスが *base* に格納されます。*base* の値が NULL ならば、pthread_create(3T) によって新しいスレッドに少なくとも PTHREAD_STACK_MIN バイトのスタックが割り当てられます。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、対応する値を戻します。

EINVAL

base または *tattr* の値が正しくありません。

次の例は、独自のスタックアドレスを指定してスレッドを生成する方法を示します。

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
int ret;
void *stackbase;

stackbase = (void *) malloc(size);

/* デフォルト属性で初期化する */
ret = pthread_attr_init(&tattr);

/* 属性に基底アドレスを設定する */
ret = pthread_attr_setstackaddr(&tattr, stackbase);

/* 属性 tattr に大きさのみを指定する */
ret = pthread_create(&tid, &tattr, func, arg);
```

次の例は、独自のスタックアドレスと独自のスタックの大きさを指定してスレッドを生成する方法を示しています。

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
int ret;
void *stackbase;

int size = PTHREAD_STACK_MIN + 0x4000;
stackbase = (void *) malloc(size);

/* デフォルト属性で初期化する */
ret = pthread_attr_init(&tattr);

/* スタックの大きさも設定する */
ret = pthread_attr_setstacksize(&tattr, size);

/* 属性に基底アドレスを設定する */
ret = pthread_attr_setstackaddr(&tattr, stackbase);

/* アドレスと大きさを指定する */
ret = pthread_create(&tid, &tattr, func, arg);
```

スタックアドレスの取得

pthread_attr_getstackaddr(3THR)

pthread_attr_getstackaddr(3THR) は、pthread_attr_setstackaddr() によって設定された、スレッドスタックのアドレスを返します。

```
プロトタイプ:

int pthread_attr_getstackaddr(pthread_attr_t *tattr,void **stackaddr);

#include <pthread.h>

pthread_attr_t tattr;
void *base;
int ret;

/* 新しいアドレスを取得する */
ret = pthread_attr_getstackaddr (&tattr, &base);
```

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、対応する値を戻します。

EINVAL

base または *tattr* の値が正しくありません。

第 4 章

同期オブジェクトを使ったプログラミング

この章では、スレッドで使用できる同期の手法と同期上の問題について説明します。

同期オブジェクトは、データと同じようにしてアクセスされるメモリー内の変数です。異なるプロセス内のスレッドは、通常はお互いに参照できませんが、スレッドが制御する共有メモリー内に格納されている同期オブジェクトを使用することにより、相互に同期をとることができます。

同期オブジェクトをファイルに置くこともできます。そうすれば、同期オブジェクトを作成したプロセスの消滅後も同期変数を有効にできます。

次の同期オブジェクトがあります。

- 相互排他ロック (mutex ロック)
- 条件変数

- セマフォ

以下のような状況で、同期は効果を発揮します。

- 同期が、共有データの整合性を保証する唯一の手段である場合。
- 異なるプロセス内のスレッド間で同じ同期オブジェクトを共同で使用する場合。同期オブジェクトを初期化するのは、連携するそれらのプロセスの中の1 つのプロセスに限るべきです。同期オブジェクトを初期化し直すと、そのロック状態が解除されることになるからです。
- 同期によって可変データの安全性を保証できる場合。
- プロセスがファイルをマッピングし、自分のスレッドにレコード形式のロックを獲得させることができる場合。ロックがいったん獲得されると、そのファイルをマッピングしているプロセス内のスレッドのうち、ロックを保持しているスレッド以外がそのロックを獲得しようとすると、そのロックが解放されるまでブロックされます。
- 整数のような単一の基本的な変数をアクセスするときでも同期が効果を持つことがあります。整数がバスのデータ幅にそろっていない、または整数がバスのデータ幅より大きいマシンでは、1 回のメモリーロードに複数のメモリーサイクルが必要な可能性があるからです。こうした状況はSPARC™ 版アーキテクチャのマシンでは生じませんが、プログラムの移植性を考慮すると、この問題は無視できません。

注 - 32 ビットアーキテクチャでは、`long long` 型は原子[1] としての処理対象ではなく、2 つの 32 ビット値として読み書きされます。`int` 型、`char` 型、`float` 型、およびポインタは、SPARC 版マシンと IA マシンでは原子的です。

相互排他ロック属性

相互排他ロック (mutex ロック) は、スレッドの実行を直列化したいときに使用します。相互排他ロックでスレッド間の同期をとるときは、通常はコードの危険領域が複数のスレッドによって同時に実行されないようにするという方法が用いられます。単一のスレッドのコードを保護する目的で相互排他ロックを使用することもできます。

1. 原子操作は、それ以上小さい操作に分割できません。

デフォルトの mutex 属性を変更するには、属性オブジェクトを宣言して初期化します。多くの場合、アプリケーションの先頭部分の一箇所で設定しますので、mutex 属性は、すばやく見つけて簡単に変更できます。表 4–1 に、この節で説明する mutex 属性操作関数を示します。

表 4–1 mutex 属性ルーチン

操作	参照先
mutex 属性オブジェクトの初期化	89ページの「pthread_mutexattr_init(3THR)」
mutex 属性オブジェクトの削除	90ページの「pthread_mutexattr_destroy(3THR)」
mutex の適用範囲設定	90ページの「pthread_mutexattr_setpshared(3THR)」
mutex のスコープの値の取得	91ページの「pthread_mutexattr_getpshared(3THR)」
mutex の型属性の設定	92ページの「pthread_mutexattr_settype(3THR)」
mutex の型属性の取得	94ページの「pthread_mutexattr_gettype(3THR)」
mutex 属性のプロトコルの設定	94ページの「pthread_mutexattr_setprotocol(3T)」
mutex 属性のプロトコルの取得	98ページの「pthread_mutexattr_getprotocol(3T)」
mutex 属性の優先順位上限の設定	99ページの「pthread_mutexattr_setprioceiling(3T)」
mutex 属性の優先順位上限の取得	100ページの「pthread_mutexattr_getprioceiling(3T)」
mutex の優先順位上限の設定	101ページの「pthread_mutex_setprioceiling(3T)」

表 4–1 mutex 属性ルーチン 続く

操作	参照先
mutex の優先順位上限の取得	103ページの「pthread_mutex_getprioceiling(3T)」
mutex の堅牢度属性の設定	104ページの「pthread_mutexattr_setrobust_np(3T)」
mutex の堅牢度属性の取得	106ページの「pthread_mutexattr_getrobust_np(3T)」

mutex のスコープ定義について、Solaris のスレッドと POSIX のスレッドとの相違点を表 4–2 に示します。

表 4–2 mutex の適用範囲の比較

Solaris	**POSIX**	定義
USYNC_PROCESS	PTHREAD_PROCESS_SHARED	このプロセスと他のプロセスのスレッドの間で同期をとるために使用する
USYNC_PROCESS_ROBUST	POSIX に相当する定義なし	異なるプロセスのスレッド間で安定的に同期をとるために使用する
USYNC_THREAD	PTHREAD_PROCESS_PRIVATE	このプロセスのスレッドの間でだけ同期をとるために使用する

mutex 属性オブジェクトの初期化

pthread_mutexattr_init(3THR)

pthread_mutexattr_init(3THR) は、このオブジェクトに関連付けられた属性をデフォルト値に初期化します。各属性オブジェクトのための記憶領域は、実行時にスレッドによって割り当てられます。

この関数が呼び出されたときの *pshared* 属性のデフォルト値は PTHREAD_PROCESS_PRIVATE で、初期化された mutex を 1 つのプロセスの中だけで使用できるという意味です。

```
プロトタイプ:
int pthread_mutexattr_init(pthread_mutexattr_t *mattr);

#include <pthread.h>


pthread_mutexattr_t mattr;
int ret;

/* 属性をデフォルト値に初期化する */
ret = pthread_mutexattr_init(&mattr);
```

mattr は不透明な型で、システムによって割り当てられた属性オブジェクトを含んでいます。*mattr* のスコープとして取り得る値は、PTHREAD_PROCESS_PRIVATE (デフォルト) と PTHREAD_PROCESS_SHARED です。

mutex 属性オブジェクトを再使用するには、pthread_mutexattr_destroy(3T) への呼び出しによって事前に削除しなければなりません。pthread_mutexattr_init() を呼び出すと、不透明なオブジェクトが割り当てられます。そのオブジェクトが削除されないと、結果的にメモリーリークを引き起こします。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると、この関数は失敗し、次の値を返します。

ENOMEM

メモリー不足のため、mutex 属性オブジェクトを初期化できません。

mutex 属性オブジェクトの削除

pthread_mutexattr_destroy(3THR)

pthread_mutexattr_destroy(3THR) は、pthread_mutexattr_init() によって生成された属性オブジェクトの管理に使用されていた記憶領域の割り当てを解除します。

```
プロトタイプ:
int pthread_mutexattr_destroy(pthread_mutexattr_t *mattr)

#include <pthread.h>

pthread_mutexattr_t mattr;
int ret;

/* 属性を削除する */
ret = pthread_mutexattr_destroy(&mattr);
```

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

mattr で指定された値が無効です。

mutex の適用範囲設定

pthread_mutexattr_setpshared(3THR)

pthread_mutexattr_setpshared(3THR) は、mutex 変数の適用範囲を設定します。

mutex 変数の値は、プロセス専用 (プロセス内) とシステム共通 (プロセス間) のどちらかです。*pshared* 属性を PTHREAD_PROCESS_SHARED 状態に設定して mutex を生成し、その mutex が共有メモリー内に存在する場合、その mutex は複数のプロセスのスレッドの間で共有できます。これは Solaris スレッドにおいて mutex_init() で USYNC_PROCESS フラグを使用するのに相当します。

```
プロトタイプ:
int pthread_mutexattr_setpshared(pthread_mutexattr_t *mattr,
    int pshared);

#include <pthread.h>

pthread_mutexattr_t mattr;
int ret;

ret = pthread_mutexattr_init(&mattr);
/*
 * デフォルト値にリセットする: private
 */
ret = pthread_mutexattr_setpshared(&mattr,
      PTHREAD_PROCESS_PRIVATE);
```

mutex の *pshared* 属性を PTHREAD_PROCESS_PRIVATE に設定した場合、その mutex を操作できるのは同じプロセスで生成されたスレッドだけです。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

mattr で指定された値が無効です。

mutex のスコープの値の取得

pthread_mutexattr_getpshared(3THR)

pthread_mutexattr_getpshared(3THR) は、pthread_mutexattr_setpshared() によって定義された、mutex 変数の適用範囲を返します。

```
プロトタイプ:
int pthread_mutexattr_getpshared(pthread_mutexattr_t *mattr,
    int *pshared);

#include <pthread.h>
```

(続く)

```
pthread_mutexattr_t mattr;
int pshared, ret;

/* mutex の pshared を取得する */
ret = pthread_mutexattr_getpshared(&mattr, &pshared);
```

属性オブジェクト *mattr* の *pshared* の現在値を取得します。これは PTHREAD_PROCESS_SHARED と PTHREAD_PROCESS_PRIVATE のどちらかです。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

mattr で指定された値が無効です。

mutex の型属性の設定

pthread_mutexattr_settype(3THR)

```
#include <pthread.h>

int pthread_mutexattr_settype(pthread_mutexattr_t  *attr , int type);
```

pthread_mutexattr_settype(3THR) は、 mutex の 型 (*type*) 属性を設定します。型 属性のデフォルト値は PTHREAD_MUTEX_DEFAULT です。

型 (*type*) 引数は mutex の型を指定します。有効な mutex 型を以下に示します。

PTHREAD_MUTEX_NORMAL

この型の mutex はデッドロックを検出しません。スレッドが、この mutex をロック解除しないでもう一度ロックしようとすると、スレッドはデッドロックします。別のスレッドによってロックされた mutex をロック解除しようとした場

合、引き起こされる動作は未定義です。また、ロック解除された mutex をロック解除しようとした場合、引き起こされる動作は不定です。

PTHREAD_MUTEX_ERRORCHECK

この型の mutex はエラーチェックを行います。スレッドがこの mutex をロック解除しないでもう一度ロックしようとすると、エラーを返します。別のスレッドがロックした mutex をロック解除しようとすると、エラーを返します。また、ロック解除された mutex をロック解除しようとするとエラーを返します。

PTHREAD_MUTEX_RECURSIVE

スレッドがこの mutex をロック解除しないでもう一度ロックしようとすると、正常にロックできます。PTHREAD_MUTEX_NORMAL 型の mutex ではロックを繰り返すとデッドロックが発生しますが、この型の mutex では発生しません。複数回ロックされた mutex を別のスレッドが獲得するときには、その前に同じ回数ロック解除する必要があります。あるスレッドがロックした mutex を別のスレッドがロック解除しようとすると、エラーが返されます。ロック解除されている mutex をスレッドがロック解除しようとすると、エラーが返されます。mutex の型は、プロセス共有属性が PTHREAD_PROCESS_PRIVATE の mutex に対してだけサポートされます。

PTHREAD_MUTEX_DEFAULT

このタイプの mutex を繰り返しロックしようとした場合、引き起こされる動作は未定義です。この型の mutex を、ロックしていないスレッドがロック解除しようとした場合、引き起こされる動作は未定義です。この型の、ロックされていない mutex をロック解除しようとした場合、引き起こされる動作は未定義です。この型の mutex は、他の mutex 型に割り当てることができます。Solaris スレッドでは、PTHREAD_PROCESS_DEFAULT は PTHREAD_PROCESS_NORMAL に割り当てられます。

戻り値

pthread_mutexattr_settype 関数は、正常に終了すると 0 を返します。それ以外の場合は、エラーを示す値を返します。

EINVAL

type の値が無効です。

EINVAL

attr で指定された値が無効です。

mutex の型属性の取得

pthread_mutexattr_gettype(3THR)

```
#include <pthread.h>

int pthread_mutexattr_gettype(pthread_mutexattr_t  *attr , int  *type);
```

pthread_mutexattr_gettype(3THR) は、pthread_mutexattr_settype() によって設定された、 mutex の 型 (*type*) 属性を取得します。型属性のデフォルト値は PTHREAD_MUTEX_DEFAULT です。

型 (*type*) 引数は mutex の型を指定します。有効な mutex 型を以下に示します。

- PTHREAD_MUTEX_NORMAL
- PTHREAD_MUTEX_ERRORCHECK
- PTHREAD_MUTEX_RECURSIVE
- PTHREAD_MUTEX_DEFAULT

各型の説明については、92ページの「pthread_mutexattr_settype(3THR)」を参照してください。

mutex 属性のプロトコルの設定

pthread_mutexattr_setprotocol(3T)

pthread_mutexattr_setprotocol(3T) は、mutex 属性オブジェクトのプロトコル属性を設定します。

```
#include <pthread.h>

int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol);
```

attr は、先の pthread_mutexattr_init() の呼び出しによって作成された mutex 属性オブジェクトを指します。

protocol には、mutex 属性オブジェクトに適用されるプロトコルを指定します。

pthread.h に定義可能な *protocol* の値は、PTHREAD_PRIO_NONE、PTHREAD_PRIO_INHERIT、または PTHREAD_PRIO_PROTECT です。

- PTHREAD_PRIO_NONE

  スレッドの優先順位とスケジューリングは、mutex の所有権の影響は受けません。

- PTHREAD_PRIO_INHERIT

  スレッド (thrd1 など) が所有する 1 つまたは複数の mutex が、より優先順位の高いスレッドによってブロックされている場合、これらの mutex が PTHREAD_PRIO_INHERIT で初期化されていると、このプロトコル値はスレッド (thrd1) の優先順位とスケジューリングに影響します。thrd1 は、より高い優先順位または thrd1 が所有する mutex を待っているスレッドの最高優先順位で実行されます。

  thrd1 が別のスレッド thrd3 が所有する mutex をブロックしている場合、同様の優先順位継承効果が thrd3 に対して再帰的に伝播されます。

  PTHREAD_PRIO_INHERIT を使用して、優先順位が逆転しないようにしてください。優先順位の低いスレッドが、そのスレッドより優先順位の高いスレッドが必要としているロックを保持していると、優先順位が逆転します。優先順位の高いスレッドは、優先順位の低いスレッドがロックを解除するまで実行を続行できないため、各スレッドは本来の優先順位が逆転しているかのように扱われます。

  シンボル _POSIX_THREAD_PRIO_INHERIT が定義されている場合、プロトコル属性値 PTHREAD_PRIO_INHERIT で初期化された mutex では、その mutex の所有者が終了すると Solaris オペレーティング環境で次の動作が発生します。

注 - 所有者終了時の動作は、pthread_mutexattr_setrobust_np() の *robustness* 引数の値によって異なります。

- mutex のロックが解除されます。
- 次の所有者がその mutex を獲得し、エラーコード EOWNERDEAD が返されます。
- mutex の次の所有者は、mutex によって保護されている状態を整合させるよう試行する必要があります。これは、前の所有者が終了したときに状態が不整合のままになっている可能性があるためです。所有者が状態を整合させることに成功すると、その mutex に対して pthread_mutex_init() を呼び出して、mutex をロック解除します。

注 - pthread_mutex_init() が前の初期化で呼び出されたが、まだ mutex を削除していない場合、mutex は初期化し直されません。

- 所有者が状態を整合させることができない場合は、pthread_mutex_init() は呼び出さず、mutex をロック解除します。この場合には、すべての待機者が呼び起こされ、それ以降の pthread_mutex_lock() へのすべての呼び出しは mutex の獲得に失敗し、エラーコード ENOTRECOVERABLE が返されます。この時点で、pthread_mutex_destroy() を呼び出して mutex を削除し、pthread_mutex_init() を呼び出して初期化し直すことによって、mutex の状態を整合させることができます。
- EOWNERDEAD を持つロックを獲得したスレッドが終了すると、次の所有者がエラーコード EOWNERDEAD を持つロックを獲得します。

- PTHREAD_PRIO_PROTECT

  あるスレッドが、PTHREAD_PRIO_PROTECT で初期化された 1 つまたは複数の mutex を所有する場合に、このプロトコル値は、スレッド (thrd2 など) の優先順位とスケジューリングに影響します。thrd2 は、より高い優先順位または自分が所有しているすべての mutex の中で最も高い優先順位で実行します。thrd2 が所有するいずれかの mutex でブロックされているより優先度の高いスレッドは、thrd2 のスケジューリングには影響を与えません。

スレッドが PTHREAD_PRIO_INHERIT または PTHREAD_PRIO_PROTECT で初期化された mutex を所有しており、sched_setparam() の呼び出しなどによってそのスレッドの元の優先順位が変更されている場合は、スケジューラは新しい優先順位のスケジューリングキューの末尾にそのスレッドを移動しません。同様に、PTHREAD_PRIO_INHERIT または PTHREAD_PRIO_PROTECT で初期化された mutex をスレッドがロック解除して、そのスレッドの元の優先順位が変更されている場合は、スケジューラは新しい優先順位のスケジューリングキューの末尾にそのスレッドを移動しません。

PTHREAD_PRIO_INHERIT で初期化された mutex と PTHREAD_PRIO_PROTECT で初期化された mutex を複数同時に所有しているスレッドは、これらのプロトコルのいずれかで獲得された最高の優先順位で実行します。

戻り値

pthread_mutexattr_setprotocol() は、正常終了すると 0 を返します。それ以外の戻り値は、エラーが発生したことを示しています。

次のどちらかの条件が検出されると、pthread_mutexattr_setprotocol() は失敗し、対応する値を返します。

ENOSYS

_POSIX_THREAD_PRIO_INHERIT と _POSIX_THREAD_PRIO_PROTECT のどちらのオプションも定義されておらず、この実装はこの関数をサポートしていません。

ENOTSUP

protocol で指定された値はサポートされていない値です。

次のどちらかの条件が検出されると、pthread_mutexattr_setprotocol() は失敗し、対応する値を返します。

EINVAL

attr または *protocol* に指定した値は無効です。

EPERM

呼び出し元はこの操作を行うための権限を持っていません。

mutex 属性のプロトコルの取得

pthread_mutexattr_getprotocol(3T)

pthread_mutexattr_getprotocol(3T) は、mutex 属性オブジェクトのプロトコル属性を取得します。

```
#include <pthread.h>

int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *attr, int *protocol);
```

attr は、先の pthread_mutexattr_init() の呼び出しによって作成された mutex 属性オブジェクトを指します。

protocol には、プロトコル属性が入ります。値は、PTHREAD_PRIO_NONE、PTHREAD_PRIO_INHERIT、または PTHREAD_PRIO_PROTECT です。

戻り値

pthread_mutexattr_getprotocol() は、正常終了すると 0 を返します。それ以外の戻り値は、エラーが発生したことを示しています。

次の条件が検出されると、pthread_mutexattr_getprotocol() は失敗し、対応する値を返します。

ENOSYS

_POSIX_THREAD_PRIO_INHERIT と _POSIX_THREAD_PRIO_PROTECT のどちらのオプションも定義されておらず、この実装はこの関数をサポートしていません。

次のどちらかの条件が検出されると、pthread_mutexattr_getprotocol() は失敗し、条件に対応する値を返します。

EINVAL

attr に指定した値は無効です。

EPERM

呼び出し元はこの操作を行うための権限を持っていません。

mutex 属性の優先順位上限の設定

pthread_mutexattr_setprioceiling(3T)

pthread_mutexattr_setprioceiling(3T) は、mutex 属性オブジェクトの優先順位上限属性を設定します。

```
#include <pthread.h>

int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr, int prioceiling, int *oldceiling);
```

attr は、先の pthread_mutexattr_init() の呼び出しによって作成された mutex 属性オブジェクトを指します。

注 - *attr* mutex 属性オブジェクトに優先順位上限属性が含まれるのは、シンボル _POSIX_THREAD_PRIO_PROTECT が定義されている場合だけです。

---

prioceiling には、初期化された mutex の優先順位上限を指定します。優先順位上限は、mutex によって保護されている重要領域が実行される最小の優先レベルを定義します。*prioceiling* は、SCHED_FIFO によって定義される優先順位の最大範囲内にあります。優先順位が逆転しないように、特定の mutex をロックするすべてのスレッドの中で最も高い優先順位と同じかまたはそれを上回る優先順位を *prioceiling* として設定します。

oldceiling には古い優先順位上限の値が入ります。

戻り値

pthread_mutexattr_setprioceiling() は、正常終了すると 0 を返します。それ以外の戻り値は、エラーが発生したことを示しています。

次のいずれかの条件が検出されると、pthread_mutexattr_setprioceiling() は失敗し、対応する値を返します。

ENOSYS

オプション _POSIX_THREAD_PRIO_PROTECT が定義されておらず、この実装はこの関数をサポートしていません。

次のどちらかの条件が検出されると、pthread_mutexattr_setprioceiling() は失敗し、対応する値を返します。

EINVAL

attr または *prioceiling* に指定した値は無効です。

EPERM

呼び出し元はこの操作を行うための権限を持っていません。

mutex 属性の優先順位上限の取得

pthread_mutexattr_getprioceiling(3T)

pthread_mutexattr_getprioceiling(3T) は、mutex 属性オブジェクトの優先順位上限属性を取得します。

```
#include <pthread.h>

int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *attr, int *prioceiling);
```

attr は、先の pthread_mutexattr_init() の呼び出しによって作成された属性オブジェクトを指します。

注 - *attr* mutex 属性オブジェクトに優先順位上限属性が含まれるのは、シンボル _POSIX_THREAD_PRIO_PROTECT が定義されている場合だけです。

---

pthread_mutexattr_getprioceiling() は、初期化された mutex の優先順位上限、*mutex* を *prioceiling* で返します。この上限は、mutex によって保護されている重要領域が実行される最小の優先レベルを定義します。*prioceiling* は、SCHED_FIFO によって定義される優先順位の最大範囲内にあります。優先順位が逆転しないように、特定の mutex をロックするすべてのスレッドの中で最も高い優先順位と同じかまたはそれを上回る優先順位を *prioceiling* として設定します。

戻り値

pthread_mutexattr_getprioceiling() は、正常終了すると 0 を返します。それ以外の戻り値は、エラーが発生したことを示しています。

次の条件が検出されると、pthread_mutexattr_getprioceiling() は失敗し、対応する値を返します。

ENOSYS

オプション _POSIX_THREAD_PRIO_PROTECT が定義されておらず、この実装はこの関数をサポートしていません。

次のどちらかの条件が検出されると、pthread_mutexattr_getprioceiling() は失敗し、対応する値を返します。

EINVAL

attr に指定した値は無効です。

EPERM

呼び出し元はこの操作を行うための権限を持っていません。

mutex の優先順位上限の設定

pthread_mutex_setprioceiling(3T)

pthread_mutex_setprioceiling(3T) は、mutex の優先順位上限を設定します。

```
#include <pthread.h>

int pthread_mutex_setprioceiling(pthread_mutexatt_t *mutex, int prioceiling, int *old_ceiling);
```

pthread_mutex_setprioceiling() は *mutex* の優先順位上限、つまり *prioceiling* を変更します。pthread_mutex_setprioceiling() は、mutex のロックが解除されている場合 mutex をロックするか、または mutex を正常にロックできるようになるまでブロックして、mutex の優先順位上限を変更し、mutex を開放します。mutex をロックするプロセスでは、優先順位保護プロトコルを守る必要はありません。

注 - mutex 属性オブジェクト、つまり *mutex* に優先順位上限が含まれるのは、シンボル _POSIX_THREAD_PRIO_PROTECT が定義されている場合だけです。

---

pthread_mutex_setprioceiling() が正常に終了すると、優先順位上限の以前の値が *old_ceiling* で返されます。pthread_mutex_setprioceiling() が失敗すると、mutex の優先順位上限は元のままになります。

戻り値

pthread_mutex_setprioceiling() は、正常終了すると 0 を返します。それ以外の戻り値は、エラーが発生したことを示しています。

次の条件が検出されると、pthread_mutexatt_setprioceiling() は失敗し、それに対応する値を返します。

ENOSYS

オプション _POSIX_THREAD_PRIO_PROTECT が定義されておらず、この実装はこの関数をサポートしていません。

次のいずれかの条件が検出されると、pthread_mutex_setprioceiling() は失敗し、対応する値を返します。

EINVAL

prioceiling で要求された優先順位が範囲外です。

EINVAL

mutex で指定された値は現在の既存の mutex を参照していません。

ENOSYS

この実装は mutex の優先順位上限プロトコルをサポートしていません。

EPERM

呼び出し元はこの操作を行うための権限を持っていません。

mutex の優先順位上限の取得

pthread_mutex_getprioceiling(3T)

pthread_mutex_getprioceiling(3T) は、mutex の優先順位上限を取得します。

```
#include <pthread.h>

int pthread_mutex_getprioceiling(const pthread_mutexatt_t *mutex, int *prioceiling);
```

pthread_mutex_getprioceiling() は、*mutex* の優先順位上限、つまり *prioceiling* を返します。

戻り値

pthread_mutex_getprioceiling() は、正常終了すると 0 を返します。それ以外の戻り値は、エラーが発生したことを示しています。

次の条件が検出されると、pthread_mutexatt_getprioceiling() は失敗し、対応する値を返します。

ENOSYS

オプション _POSIX_THREAD_PRIO_PROTECT が定義されておらず、この実装はこの関数をサポートしていません。

次のいずれかの条件が検出されると、pthread_mutex_getprioceiling() は失敗し、対応する値を返します。

EINVAL

mutex で指定された値は現在の既存の mutex を参照していません。

ENOSYS

この実装は mutex の優先順位上限プロトコルをサポートしていません。

EPERM

呼び出し元はこの操作を行うための権限を持っていません。

mutex の堅牢度属性の設定

pthread_mutexattr_setrobust_np(3T)

pthread_mutexattr_setrobust_np(3T) は、mutex 属性オブジェクトの堅牢度属性を設定します。

```
#include <pthread.h>

int pthread_mutexattr_setrobust_np(pthread_mutexatt_t *attr, int *robustness);
```

注 - pthread_mutexattr_setrobust_np() が適用されるのは、シンボル _POSIX_THREAD_PRIO_INHERIT が定義されている場合だけです。

---

attr は、先の pthread_mutexattr_init() の呼び出しによって作成された mutex 属性オブジェクトを指します。

robustness は、mutex の所有者が終了した場合の動作を定義します。pthread.h に定義可能な *robustness* の値は、PTHREAD_MUTEX_ROBUST_NP または PTHREAD_MUTEX_STALLED_NP です。デフォルト値は、PTHREAD_MUTEX_STALLED_NP です。

- PTHREAD_MUTEX_ROBUST_NP

  mutex の所有者が終了すると、それ以降の pthread_mutex_lock() へのすべての呼び出しは、指定しない方法で進行過程からブロックされます。

- PTHREAD_MUTEX_STALLED_NP

  mutex の所有者が終了すると、mutex はロック解除されます。この mutex の次の所有者が獲得し、エラーコード EOWNERDEAD が返されます。

注 - 作成するアプリケーションは、このタイプの mutex について、pthread_mutex_lock() から出力される戻りコードをチェックする必要があります。

  - この mutex の新しい所有者は、mutex によって保護されている状態を整合させる必要があります。これは、前の所有者が終了したときに状態が不整合のままになっている可能性があるためです。

- 新しい所有者が状態を整合できる場合は、その mutex に対して pthread_mutex_consistent_np() を呼び出して、mutex をロック解除します。
- 新しい所有者が状態を整合できない場合は、その mutex に対して pthread_mutex_consistent_np() を呼び出さずに、mutex をロック解除してください。

  すべての待機者が呼び起こされ、それ以降の pthread_mutex_lock() へのすべての呼び出しは mutex の獲得に失敗し、エラーコード ENOTRECOVERABLE を返します。この時点で、pthread_mutex_destroy() を呼び出して mutex を削除し、pthread_mutex_int() を呼び出して初期化し直すことによって、mutex の状態を整合させることができます。

EOWNERDEAD を持つロックを獲得したスレッドが終了すると、次の所有者がリターンコード EOWNERDEAD を持つロックを獲得します。

戻り値

pthread_mutexattr_setrobust_np() は、正常終了すると 0 を返します。それ以外の戻り値は、エラーが発生したことを示しています。

次の条件のいずれかが検出されると、pthread_mutexattr_setrobust_np() は失敗し、対応する値を返します。

ENOSYS

オプション _POSIX_THREAD_PRIO__INHERIT が定義されていないか、あるいはこの実装が pthread_mutexattr_setrobust_np() 関数をサポートしていません。

ENOTSUP

robustness で指定された値はサポートされていません。

次の条件が検出されると、pthread_mutexattr_setrobust_np() は失敗します。

EINVAL

attr または *robustness* で指定された値は無効です。

mutex の堅牢度属性の取得

pthread_mutexattr_getrobust_np(3T)

pthread_mutexattr_getrobust_np(3T) は、mutex 属性オブジェクトの堅牢度属性を取得します。

```
#include <pthread.h>

int pthread_mutexattr_getrobust_np(pthread_mutexatt_t *attr, int *robustness);
```

注 - pthread_mutexattr_getrobust_np() が適用されるのは、シンボル _POSIX_THREAD_PRIO_INHERIT が定義されている場合だけです。

---

attr は、先の pthread_mutexattr_init() の呼び出しによって作成された mutex 属性オブジェクトを指します。

robustness は、mutex 属性オブジェクトの堅牢度属性の値です。

戻り値

pthread_mutexattr_getrobust_np() は、正常終了すると 0 を返します。それ以外の戻り値は、エラーが発生したことを示しています。

次の条件のいずれかが検出されると、pthread_mutexattr_getrobust_np() は失敗し、対応する値を返します。

ENOSYS

オプション _POSIX_THREAD_PRIO__INHERIT が定義されていないか、あるいはこの実装が pthread_mutexattr_getrobust_np() 関数をサポートしていません。

ENOTSUP

robustness で指定された値はサポートされていません。

次の条件が検出されると、pthread_mutexattr_getrobust_np() は失敗します。

EINVAL

attr または *robustness* で指定された値は無効です。

相互排他ロックの使用方法

表 4–3 に、この章で説明する mutex ロック操作関数を示します。

表 4–3 相互排他ロック操作ルーチン

操作	参照先
mutex の初期化	108ページの「pthread_mutex_init(3THR)」
mutex の整合性保持	109ページの「pthread_mutex_consistent_np(3T)」
mutex のロック	110ページの「pthread_mutex_lock(3THR)」
mutex のロック解除	113ページの「pthread_mutex_unlock(3THR)」
ブロックしないで行う mutex のロック	114ページの「pthread_mutex_trylock(3THR)」
mutex の削除	115ページの「pthread_mutex_destroy(3THR)」

デフォルトスケジューリング方針 `SCHED_OTHER` は、スレッドによるロックの獲得順序を指定していません。複数のスレッドが mutex を待っているときの獲得の順序は不定です。競合するときは、スレッドを優先順位でブロック解除するというのがデフォルト動作です。

mutex の初期化

pthread_mutex_init(3THR)

pthread_mutex_init(3THR) は、*mp* が指す mutex をデフォルト値に初期化 (*mattr* が NULL の場合) するか、pthread_mutexattr_init() ですでに設定されている mutex 属性を指定するときに使用します。(Solaris スレッドについては、248 ページの「mutex_init(3THR)」を参照)。

```
プロトタイプ:
int pthread_mutex_init(pthread_mutex_t *mp,
    const pthread_mutexattr_t *mattr);

#include <pthread.h>

pthread_mutex_t mp = PTHREAD_MUTEX_INITIALIZER;
pthread_mutexattr_t mattr;
int ret;

/* mutex をデフォルト値に初期化する */
ret = pthread_mutex_init(&mp, NULL);

/* mutex を初期化する */
ret = pthread_mutex_init(&mp, &mattr);
```

初期化された mutex は、ロック解除状態になります。mutex は、プロセス間で共有されているメモリー内または個々のプロセス専用のメモリー内に置かれます。

注 - mutex メモリーは、初期化する前にクリアしてゼロにする必要があります。

---

mattr を NULL にするのは、デフォルト mutex 属性オブジェクトのアドレスを渡すのと同じことですが、メモリーのオーバーヘッドがありません。

mutex を静的に定義する場合、マクロ PTHREAD_MUTEX_INITIALIZER により、デフォルト属性を持つように直接初期化できます。

mutex ロックは、他のスレッドが使用している可能性がある間は再初期化したり削除したりしてはいけません。どちらの動作も正しく行われなければプログラムで障害が発生します。mutex を再初期化または削除する場合、アプリケーションがその mutex を使用していないことが確実でなければなりません。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると、この関数は失敗し、対応する値を戻します。

EBUSY

mp で示されたオブジェクト (初期化されているが、まだ削除されていない mutex) の再初期化の試行が検出されました。

EINVAL

mattr 属性値が無効です。その mutex は変更されていません。

EFAULT

mp が指す mutex のアドレスが無効です。

mutex の整合性保持

pthread_mutex_consistent_np(3T)

```
#include <pthread.h>
int pthread_mutex_consistent_np(pthread_mutex_t *mutex);
```

注 - pthread_mutex_consistent_np() が適用されるのは、シンボル _POSIX_THREAD_PRIO_INHERIT が定義され、かつプロトコル属性値 PTHREAD_PRIO_INHERIT で初期化されている mutex に対してのみです。

---

mutex の所有者が終了すると、mutex が不整合になる可能性があります。

pthread_mutex_consistent_np が、*mutex* の所有者の終了後に mutex オブジェクト、*mutex* を整合させます。

不整合の mutex を獲得するには、pthread_mutex_lock() を呼び出します。戻り値 EOWNERDEAD は不整合な mutex であることを示します。

pthread_mutex_consistent_np() は、pthread_mutex_lock() への前の呼び出しによって獲得された mutex を保持している間に呼び出してください。

mutex によって保護されている重要領域が、終了した所有者によって不整合の状態のままになっている可能性があるので、mutex によって保護されている重要領域を整合させることができる場合にのみ mutex を整合させてください。

整合された mutex に対して
`pthread_mutex_lock()`、`pthread_mutex_unlock()` および
`pthread_mutex_trylock()` を呼び出すと、通常の方法で動作します。

不整合でない、あるいは保持されていない mutex に対する
`pthread_mutex_consistent_np()` の動作は、定義されていません。

戻り値

`pthread_mutex_consistent_np()` は、正常終了すると 0 を返します。それ以外の戻り値は、エラーが発生したことを示しています。次の条件が検出されると、関数は失敗し、対応する値を返します。

次の条件が検出されると、`pthread_mutex_consistent_np()` は失敗します。

`ENOSYS`

オプション `_POSIX_THREAD_PRIO_INHERIT` が定義されていないか、あるいはこの実装が `pthread_mutex_consistent_np()` 関数をサポートしていません。

次の条件が検出されると、`pthread_mutex_consistent_np()` は失敗します。

`EINVAL`

mutex で指定された値は無効です。

mutex のロック

pthread_mutex_lock(3THR)

```
プロトタイプ:
int pthread_mutex_lock(pthread_mutex_t *mutex);

#include <pthread.h>

pthread_mutex_t mutex;
```

(続く)

```
int ret;

ret = pthread_ mutex_lock(&mp); /* mutex を獲得する */
```

pthread_mutex_lock(3THR) は、*mutex* が指す mutex をロックします。pthread_mutex_lock() が戻ると、呼び出しスレッドが mutex をロックした状態になっています。mutex が別のスレッドによってすでにロックされている (所有されている) 場合は、呼び出しスレッドは mutex が使用可能になるまでブロックされます (Solaris スレッドについては、250ページの「mutex_lock(3THR)」を参照)。

mutex 型が PTHREAD_MUTEX_NORMAL の場合、デッドロックの検出は行われません。mutex をもう一度ロックしようとするとデッドロックが発生します。スレッドが、ロックされていない mutex やロック解除された mutex をロック解除しようとした場合、引き起こされる動作は未定義です。

mutex 型が PTHREAD_MUTEX_ERRORCHECK の場合は、エラーチェックが提供されます。すでにロックされた mutex をもう一度ロックしようとすると、エラーが返されます。ロックされていない mutex やロック解除された mutex をロック解除しようとすると、エラーが返されます。

mutex 型が PTHREAD_MUTEX_RECURSIVE の場合は、mutex はロックの回数を記録します。スレッドが最初に正常に mutex を獲得すると、ロック計数は 1 に設定されます。この mutex をスレッドがさらにロックするたびに、ロックカウントが 1 ずつ増えます。スレッドが mutex をロック解除するたびに、ロックカウントが 1 ずつ減ります。ロックカウントが 0 になると、その mutex を別のスレッドが獲得できるようになります。ロックされていない mutex やロック解除された mutex をロック解除しようとすると、エラーが返されます。

mutex 型が PTHREAD_MUTEX_DEFAULT の場合、繰り返し mutex をロックしようとすると、引き起こされる動作は未定義です。mutex をロックしていないスレッドがロック解除しようとした場合、引き起こされる動作は未定義です。また、ロックされていない mutex をロック解除しようとした場合、引き起こされる動作は未定義です。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると、この関数は失敗し、次の値を戻します。

`EAGAIN`

mutex の再帰的なロックが最大数を超えるため、mutex を獲得できません。

`EDEADLK`

現在のスレッドがすでにその mutex を獲得しています。

シンボル `_POSIX_THREAD_PRIO_INHERIT` が定義されていて、mutex がプロトコル属性値 `PTHREAD_PRIO_INHERIT` で初期化されており、`pthread_mutexattr_setrobust_np()` の *robustness* 引数が `PTHREAD_MUTEX_ROBUST_NP` である場合、この関数は失敗し、次の値を返します。

`EOWNERDEAD`

この mutex の前の所有者が mutex を保持している間に終了しました。現在この mutex は、呼び出し元によって所有されています。呼び出し元は、mutex によって保護された状態を整合させるよう試行する必要があります。

呼び出し元が状態を整合させることができた場合、その mutex に対して `pthread_mutex_consistent_np()` を呼び出して、mutex をロック解除します。これ以降の `pthread_mutex_lock()` の呼び出しは正常に動作します。

呼び出し元が状態を整合させることができない場合は、その mutex に対して `pthread_mutex_init()` は呼び出さず、mutex をロック解除します。これ以降の `pthread_mutex_lock()` のすべての呼び出しは mutex の獲得に失敗し、エラーコード `ENOTRECOVERABLE` を返します。

`EOWNERDEAD` を持つロックを獲得した所有者が終了すると、次の所有者が `EOWNERDEAD` を持つロックを獲得します。

`ENOTRECOVERABLE`

獲得しようとしている mutex は、ロックの保持中に終了した前の所有者によって回復不能にされた状態を保護しています。mutex は獲得されませんでした。ロックが以前に `EOWNERDEAD` を指定されて獲得され、所有者が状態をクリーンアップできず、mutex の状態を整合させないで mutex をロック解除した場合に、この状況が発生します。

ENOMEM

同時に保持される mutex の上限数を超えています。

mutex のロック解除

pthread_mutex_unlock(3THR)

pthread_mutex_unlock(3THR) は、*mutex* が指す mutex のロックを解除します。(Solaris スレッドについては、251ページの「mutex_unlock(3THR)」を参照)。

```
プロトタイプ:
int pthread_mutex_unlock(pthread_mutex_t *mutex);

#include <pthread.h>

pthread_mutex_t mutex;
int ret;

ret = pthread_mutex_unlock(&mutex); /* mutex を解除する */
```

pthread_mutex_unlock() は、*mutex* が指す mutex オブジェクトを解放します。mutex を解放する方法は、mutex の 型属性に依存します。pthread_mutex_unlock() が呼び出されたときに、指定された *mutex* が指す mutex オブジェクトでブロックされているスレッドがあり、この呼び出しによって mutex が使用できるようになると、スケジューリング方針に基づいて mutex を獲得するスレッドが決定されます。PTHREAD_MUTEX_RECURSIVE のタイプの mutex の場合、mutex が使用可能になるのは、カウントが 0 になり、pthread_mutex_unlock() を呼び出したスレッドがこの *mutex* のロックを解除したときです。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、次の値を戻します。

EPERM

現在のスレッドは mutex を所有していません。

ブロックしないで行う **mutex** のロック

pthread_mutex_trylock(3THR)

pthread_mutex_trylock(3THR) は、*mutex* が指す mutex のロックを試みます。(Solaris スレッドについては、251ページの「mutex_trylock(3THR)」を参照)。

```
プロトタイプ:
int pthread_mutex_trylock(pthread_mutex_t *mutex);

#include <pthread.h>

pthread_mutex_t mutex;
int ret;

ret = pthread_mutex_trylock(&mutex); /* mutex のロックを試みる */
```

この関数はブロックしない点を除いて、pthread_mutex_lock() と同じ働きをします。*mutex* が参照している mutex オブジェクトが、現在のスレッドを含むいずれかのスレッドによってロックされている場合は、呼び出しはただちに返されます。mutex オブジェクトがロックされていなければ、呼び出しスレッドがロックを獲得します。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると、この関数は失敗し、次の値を戻します。

EBUSY

mutex が指している mutex はすでにロックされているため、獲得できません。

EAGAIN

mutex に繰り返し行われたロック回数が最大数を超えるため、mutex を所有できません。

シンボル _POSIX_THREAD_PRIO_INHERIT が定義されていて、mutex がプロトコル属性値 PTHREAD_PRIO_INHERIT で初期化されており、pthread_mutexattr_setrobust_np() の *robustness* 引数が PTHREAD_MUTEX_ROBUST_NP である場合、この関数は失敗し、次の値を返します。

EOWNERDEAD

この mutex の前の所有者が mutex を保持している間に終了しました。現在この mutex は、呼び出し元によって所有されています。呼び出し元は、mutex によって保護された状態を整合させるよう試行する必要があります。

呼び出し元が状態を整合させることができた場合、その mutex に対して pthread_mutex_consistent_np() を呼び出して、mutex をロック解除します。これ以降の pthread_mutex_lock() の呼び出しは正常に動作します。

呼び出し元が状態を整合させることができない場合は、その mutex に対して pthread_mutex_init() は呼び出さず、mutex をロック解除します。これ以降の pthread_mutex_trylock() のすべての呼び出しは mutex の獲得に失敗し、エラーコード ENOTRECOVERABLE を返します。

EOWNERDEAD を持つロックを獲得した所有者が終了すると、次の所有者が EOWNERDEAD を持つロックを獲得します。

ENOTRECOVERABLE

獲得しようとしている mutex は、ロックの保持中に終了した前の所有者によって回復不能にされた状態を保護しています。mutex は獲得されませんでした。ロックが以前に EOWNERDEAD を指定されて獲得され、所有者が状態をクリーンアップできず、mutex の状態を整合させないで mutex をロック解除した場合に、この状況が発生します。

ENOMEM

同時に保持される mutex の上限数を超えています。

mutex の削除

pthread_mutex_destroy(3THR)

pthread_mutex_destroy(3THR) は、*mp* が指す mutex に関連するすべての状態を削除します。(Solaris スレッドについては、250ページの「mutex_destroy(3THR)」を参照)。

```
プロトタイプ:
int pthread_mutex_destroy(pthread_mutex_t *mp);
```

(続く)

```
#include <pthread.h>

pthread_mutex_t mp;
int ret;

ret = pthread_mutex_destroy(&mp); /* mutex を削除する */
```

mutex の記憶領域は解放されません。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、次の値を返します。

EINVAL

mp で指定された値が、初期化された mutex オブジェクトを表していません。

mutex ロックのコード例

例 4–1 に、mutex ロックを示すコードの一部を示します。

例 4–1 mutex ロックの例

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long long count;

void
increment_count()
{
     pthread_mutex_lock(&count_mutex);
    count = count + 1;
     pthread_mutex_unlock(&count_mutex);
}

long long
get_count()
{
    long long c;
```

(続く)

```
    pthread_mutex_lock(&count_mutex);
     c = count;
    pthread_mutex_unlock(&count_mutex);
     return (c);
}
```

例 4–1 の 2 つの関数は、相互排他 (mutex) ロックをそれぞれ別の目的で使用しています。`increment_count()` 関数は、相互排他ロックによって共有変数の原子的操作による更新を保証しています。`get_count()` 関数は、相互排他ロックによって 64 ビット値の *count* が原子的に読み取られるようにしています。32 ビットアーキテクチャでは、`long long` は実際には 2 つの 32 ビット値として処理されます。

整数はほとんどのマシンで共通のワードサイズであるため、整数値の読み取りは原子操作です。

ロック序列の使用

同時に 2 つのリソースをアクセスすることがあります。一方のリソースを使用しているとき、もう一方のリソースも必要となる場合があります。2 つのスレッドが同じ 2 つのリソースを要求しようとして両者が異なる順序で、対応する相互排他ロックを獲得しようとする場合に問題が生じることがあります。たとえば 、2 つのスレッドがそれぞれ mutex の 1 と 2 をロックした場合、次に各スレッドが互いにもう一方の mutex をロックしようとするとデッドロックが発生します。例 4–2 に、デッドロックが発生する場合のシナリオを示します。

例 **4–2** デッドロック

スレッド **1**	スレッド **2**
`pthread_mutex_lock(&m1);`	`pthread_mutex_lock(&m2);`
	`/* リソース 2 を使用 */`
`/* リソース 1 を使用 */`	`pthread_mutex_lock(&m1);`
`pthread_mutex_lock(&m2);`	
	`/* リソース 1 と 2 を使用 */`
`/* リソース 1 と 2 を使用 */`	`pthread_mutex_unlock(&m1);`
`pthread_mutex_unlock(&m2);`	`pthread_mutex_unlock(&m2);`
`pthread_mutex_unlock(&m1);`	

この問題を回避する最善の方法は、スレッドで複数の mutex をロックする場合、常に同じ順序でロックすることです。ロックが常に規定された順序で実行されれば、デッドロックは起こらないはずです。この方法をロック序列と呼び、mutex に論理的な番号を割り振ることにより mutex に順序を付けます。

自分がある番号を持つ mutex を保持しているときより小さい番号が割り振られている mutex はロックできないという規定を守るようにします。

ただし、この方法は常に使用できるとは限りません。規定と違う順序で相互排他ロックを獲得しなければならないこともあるからです。そのような状況でデッドロックを防ぐには、`pthread_mutex_trylock()` を使用します。デッドロックが避けられないような事態が生じた場合は、ある 1 つのスレッドが現在保持している mutex のロックを解除する必要があります。

例 **4–3**　条件付きロック

スレッド **1**	スレッド **2**
`pthread_mutex_lock(&m1);`	`for (; ;)`
`pthread_mutex_lock(&m2);`	`{ pthread_mutex_lock(&m2);`
	`if (pthread_mutex_trylock(&m1)==0)`
	`/* 獲得成功 */`
`/* 解放 */`	`break;`
`pthread_mutex_unlock(&m2);`	`/* 獲得失敗 */`
`pthread_mutex_unlock(&m1);`	`pthread_mutex_unlock(&m2);`
	`}`
	`/* ロックを獲得し、解放 */`
	`pthread_mutex_unlock(&m1);`
	`pthread_mutex_unlock(&m2);`

例 4–3 では、スレッド 1 は mutex を規定通りの順序でロックしようとしていますが、スレッド 2 ではロックの順序が違います。デッドロックが発生しないようにするために、スレッド 2 は mutex の 1 を慎重にロックしなければなりません。これは、mutex の 1 が解放されるまで待つとすると、スレッド 1 との間にデッドロックの関係が生じる恐れがあるからです。

これを防ぐため、スレッド 2 は `pthread_mutex_trylock()` を呼び出し、mutex がロックされていなければロックします。ロックされていれば、スレッド 2 はただちにエラーを返します。その時点で、スレッド 2 は mutex の 2 を解放しなければなりません。その結果、スレッド 1 は mutex の 2 をロックでき、最終的には mutex の 1 と 2 の両方を解放します。

片方向リンクリストの入れ子のロック

例 4–4 と例 4–5で、一度に 3 つのロックを獲得する場合を説明します。この例では、デッドロックを防ぐために規定された順序でロックします。

例 4–4　片方向リンクのリスト構造体

```
typedef struct node1 {
    int value;
    struct node1 *link;
    pthread_mutex_t lock;
} node1_t;

node1_t ListHead;
```

この例で使用する片方向リンクのリスト構造体は、各ノードに相互排他ロックを含んでいます。このリストから特定のノードを削除する場合は、最初に *ListHead* (これが削除されることはない) の位置からリストをたどって目的のノードを探します。

この検索を同時並行的に行われる削除から保護するために、各ノードをロックしてからノードの内容にアクセスしなければなりません。すべての検索が *ListHead* の位置から開始されるので、常にリストの順序でロックされます。このため、デッドロックは決して発生しません。

目的のノードが見つかった時は、この変更がそのノードと直前のノードの両方に影響を与えるため、両方をロックします。直前のノードのロックが常に最初に獲得されるので、ここでもデッドロックの心配はありません。例 4–5 は、片方向リンクリストから特定のノードを削除する C コードを示しています。

例 4–5　片方向リンクリストの入れ子のロック

```
node1_t *delete(int value)
{
    node1_t *prev, *current;

    prev = &ListHead;
    pthread_mutex_lock(&prev->lock);

     while ((current = prev->link) != NULL) {
        pthread_mutex_lock(&current->lock);
        if (current->value == value) {
            prev->link = current->link;
            pthread_mutex_unlock(&current->lock);

             pthread_mutex_unlock(&prev->lock);
            current->link = NULL;
            return(current);
        }
        pthread_mutex_unlock(&prev->lock);
        prev = current;
    }
    pthread_mutex_unlock(&prev->lock);
```

(続く)

```
    return(NULL);
}
```

循環リンクリストの入れ子のロック

例 4–6 は、前述のリスト構造を修正して循環リストにしたものです。先頭のノードとして識別されるノードはありません。スレッドは適当な 1 つのノードに関連付けられると、そのノードと次のノードに対して操作を行います。この状況ではロック序列は適用できません。明らかに階層 (つまり、リンクをたどる順番) が循環的だからです。

例 4–6 循環リンクリスト

```
typedef struct node2 {
    int value;
    struct node2 *link;
    pthread_mutex_t lock;
} node2_t;
```

例 4–7 では 2 つのノードをロックし、両方のノードに対してある操作を行なっている C コードを示します。

例 4–7 循環リンクリストの入れ子のロック

```
void Hit Neighbor(node2_t *me) {
    while (1) {
        pthread_mutex_lock(&me->lock);
        if (pthread_mutex_lock(&me->link->lock)!= 0) {
            /* ロック失敗 */
            pthread_mutex_unlock(&me->lock);
            continue;
        }
        break;
    }
    me->link->value += me->value;
    me->value /=2;
    pthread_mutex_unlock(&me->link->lock);
    pthread_mutex_unlock(&me->lock);
}
```

条件変数の属性

条件変数は、ある条件が真になるまでスレッドを原子的にブロックしたいときに使用します。必ず相互排他ロックとともに使用します。

条件変数を使うと、特定の条件が真になるまでスレッドを原子的にブロックできます。この条件判定は、相互排他ロックにより保護された状態で行います。

条件が偽のとき、スレッドは通常は条件変数でブロック状態に入り、相互排他ロックを原子的操作により解除して、条件が変更されるのを待ちます。別のスレッドが条件を変更すると、そのスレッドはそれに関連する条件変数にシグナルを送り、その条件変数でブロックしているスレッドを呼び起こします。呼び起こされたスレッドは再度相互排他ロックを獲得し、条件を再び評価します。

異なるプロセスに所属するスレッドの間で、条件変数を使って同期をとるためには、連携するそれらのプロセスの間で共有される書き込み可能なメモリーに、条件変数の領域を確保する必要があります。

スケジューリング方針は、ブロックされたスレッドがどのように呼び起こさるかを決定します。デフォルト SCHED_OTHER の場合、スレッドは優先順位に従って呼び起こされます。

条件変数の属性は、使用する前に設定して初期化しておかなければなりません。条件変数の属性を操作する関数を表 4–4 に示します。

表 4–4 条件変数の属性

操作	参照先
条件変数の属性の初期化	123ページの「pthread_condattr_init(3THR)」
条件変数の属性の削除	124ページの「pthread_condattr_destroy(3THR)」
条件変数のスコープの設定	125ページの「pthread_condattr_setpshared(3THR)」
条件変数のスコープの取得	126ページの「pthread_condattr_getpshared(3THR)」

表 4–4　条件変数の属性　続く

条件変数のスコープ定義について、Solaris スレッドと POSIX スレッドの相違点を表 4–5 に示します。

表 4–5　条件変数のスコープの比較

Solaris	POSIX	定義
USYNC_PROCESS	PTHREAD_PROCESS_SHARED	このプロセスと他のプロセスのスレッドの間で同期をとるために使用する。
USYNC_THREAD	PTHREAD_PROCESS_PRIVATE	このプロセスのスレッドの間でだけ同期をとるために使用する。

条件変数の属性の初期化

pthread_condattr_init(3THR)

pthread_condattr_init(3THR) は、このオブジェクトに関連付けられた属性をデフォルト値に初期化します。各属性オブジェクトのための記憶領域は、実行時にスレッドシステムによって割り当てられます。この関数が呼び出されたときの *pshared* 属性のデフォルト値は PTHREAD_PROCESS_PRIVATE で、初期化された条件変数を 1 つのプロセスの中だけで使用できるという意味です。

```
プロトタイプ:
int pthread_condattr_init(pthread_condattr_t *cattr);

#include <pthread.h>
pthread_condattr_t  cattr;
int ret;

/* initialize an attribute to default value */
ret = pthread_condattr_init(&cattr);
```

cattr は不透明なデータ型で、システムによって割り当てられた属性オブジェクトを格納します。*cattr* のスコープとして取りうる値は、PTHREAD_PROCESS_PRIVATE (デフォルト) と PTHREAD_PROCESS_SHARED です。

条件変数属性を再使用するには、pthread_condattr_destroy(3T) によって事前に削除しなければなりません。pthread_condattr_init() 呼び出しは、不透明なオブジェクトへのポインタを戻します。そのオブジェクトが削除されないと、結果的にメモリーリークを引き起こします。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると、この関数は失敗し、対応する値を返します。

ENOMEM

メモリーが足りなくて、スレッド属性オブジェクトを初期化できません。

EINVAL

cattr で指定された値が無効です。

条件変数の属性の削除

pthread_condattr_destroy(3THR)

pthread_condattr_destroy(3THR) は記憶領域を解除し、属性オブジェクトを無効にします。

```
プロトタイプ:
int pthread_condattr_destroy(pthread_condattr_t *cattr);

#include <pthread.h>
pthread_condattr_t cattr;
int ret;

/* 属性を削除する */
ret = pthread_condattr_destroy(&cattr);
```

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

cattr で指定された値が無効です。

条件変数のスコープの設定

pthread_condattr_setpshared(3THR)

pthread_condattr_setpshared(3THR) は、プロセス専用 (プロセス内) とシステム共通 (プロセス間) のどちらかに条件変数のスコープを設定します。*pshared* 属性を PTHREAD_PROCESS_SHARED 状態に設定して条件変数を生成し、その条件変数が共有メモリー内に存在する場合、その条件変数は複数のプロセスのスレッドの間で共有できます。これは、オリジナルの Solaris スレッドにおいて mutex_init() で USYNC_PROCESS フラグを使用するのに相当します。

mutex の *pshared* 属性を PTHREAD_PROCESS_PRIVATE (デフォルト値) に設定した場合、その mutex を操作できるのは同じプロセスで生成されたスレッドに限られます。PTHREAD_PROCESS_PRIVATE を使用した場合、その動作はオリジナルの Solaris スレッドにおいて cond_init() 呼び出しで USYNC_THREAD フラグを使用したとき、すなわち局所条件変数と同じになります。PTHREAD_PROCESS_SHARED は広域条件変数に相当します。

```
プロトタイプ:
int pthread_condattr_setpshared(pthread_condattr_t *cattr,
    int pshared);

#include <pthread.h>

pthread_condattr_t cattr;
int ret;

/* 全プロセス */
ret = pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_SHARED);

/* 1 つのプロセス内 */
ret = pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_PRIVATE);
```

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

cattr または *pshared* の値が無効です。

条件変数のスコープの取得

pthread_condattr_getpshared(3THR)

pthread_condattr_getpshared(3THR) は属性オブジェクト *cattr* の *pshared* の現在のスコープ値を取得します。これは PTHREAD_PROCESS_SHARED と PTHREAD_PROCESS_PRIVATE のどちらかです。

```
プロトタイプ:
int pthread_condattr_getpshared(const pthread_condattr_t *cattr,
    int *pshared);

#include <pthread.h>

pthread_condattr_t cattr;
int pshared;
int ret;

/* 条件変数の pshared 値を取得する */
ret = pthread_condattr_getpshared(&cattr, &pshared);
```

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

cattr の値が無効です。

条件変数の使用方法

この節では条件変数の使用方法を説明します。表 4–6 にそのための関数を示します。

表 4–6 条件変数関数

操作	参照先
条件変数の初期化	127ページの「pthread_cond_init(3THR)」
条件変数によるブロック	129ページの「pthread_cond_wait(3THR)」
特定のスレッドのブロック	130ページの「pthread_cond_signal(3THR)」
時刻指定のブロック	132ページの「pthread_cond_timedwait(3THR)」
全スレッドのブロック解除	134ページの「pthread_cond_broadcast(3THR)」
条件変数の削除	135ページの「pthread_cond_destroy(3THR)」

条件変数の初期化

pthread_cond_init(3THR)

`pthread_cond_init(3THR)` は、*cv* が指す条件変数をデフォルト値 (*cattr* が NULL) に初期化します。また、`pthread_condattr_init()` ですでに設定してある条件変数の属性を指定することもできます。*cattr* を NULL にするのは、デフォルト条件変数属性オブジェクトのアドレスを渡すのと同じですが、メモリーのオーバーヘッドがありません。(Solaris スレッドについては、252ページの「cond_init(3THR)」を参照)。

```
プロトタイプ:
int pthread_cond_init(pthread_cond_t *cv,
    const pthread_condattr_t *cattr);

#include <pthread.h>

pthread_cond_t cv;
pthread_condattr_t cattr;
int ret;

/* 条件変数をデフォルト値に初期化する */
ret = pthread_cond_init(&cv, NULL);

/* 条件変数を初期化する */
ret = pthread_cond_init(&cv, &cattr);
```

静的に定義された条件変数は、マクロ PTHREAD_COND_INITIALIZER で、デフォルト属性をもつように直接初期化できます。この効果は、NULL 属性を指定して pthread_cond_init() を動的に割り当てるのと同じです。エラーチェックは行われません。

複数のスレッドで同じ条件変数を同時に初期化または再初期化しないでください。条件変数を再初期化または削除する場合、アプリケーションでその条件変数が現在使用されていないことを確認しなければなりません。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

cattr で指定された値が無効です。

EBUSY

その条件変数は現在使用されています。

EAGAIN

必要なリソースが利用できません。

ENOMEM

メモリー不足のため条件変数を初期化できません。

条件変数によるブロック

pthread_cond_wait(3THR)

pthread_cond_wait(3THR) は、*mp* が指す相互排他ロックを原子操作により解放し、*cv* が指す条件変数で呼び出しスレッドをブロックします。(Solaris スレッドについては、254ページの「cond_wait(3THR)」を参照)。

```
プロトタイプ:
int pthread_cond_wait(pthread_cond_t *cv,pthread_mutex_t *mutex);

#include <pthread.h>

pthread_cond_t cv;
pthread_mutex_t mp;
int ret;

/* 条件変数でブロック */
ret = pthread_cond_wait(&cv, &mp);
```

ブロックされたスレッドを呼び起こすには、pthread_cond_signal() か pthread_cond_broadcast() を使います。また、スレッドはシグナルの割り込みによっても呼び起こされます。

pthread_cond_wait() が戻ったからといって、条件変数に対応する条件の値が変化したと判断することはできません。このため、条件をもう一度評価しなければなりません。

pthread_cond_wait() が戻るときは、たとえエラーを戻したときでも、常に mutex は呼び出しスレッドがロックし保持している状態にあります。

pthread_cond_wait() は、指定の条件変数にシグナルが送られてくるまでブロック状態になります。pthread_cond_wait() は原子的操作により、対応する mutex ロックを解除してからブロック状態に入り、ブロック状態から戻る前にもう一度原子的操作によりロックを獲得します。

通常の用法は次のとおりです。mutex ロックの保護下で条件式を評価します。条件式が偽のとき、スレッドは条件変数でブロック状態に入ります。別のスレッドが条件の値を変更すると、条件変数にシグナルが送られます。その条件変数でブロックされていた (1 つまたは全部の) スレッドは、そのシグナルによってブロックが解除され、もう一度 mutex ロックを獲得しようとします。

呼び起こされたスレッドが pthread_cond_wait() から戻る前に条件が変更されることもあるので、mutex ロックを獲得する前に、待ち状態の原因となった条件をもう一度評価しなければなりません。条件チェックを while() ループに入れ、そこで pthread_cond_wait() を呼び出すようにすることをお勧めします。

```
pthread_mutex_lock();
    while(condition_is_false)
        pthread_cond_wait();
pthread_mutex_unlock();
```

条件変数で複数のスレッドがブロックされているとき、それらのスレッドが、どの順番でブロックが解除されるかは不定です。

注 - pthread_cond_wait() は取り消しポイントです。保留状態になっている取り消しがあって、呼び出しスレッドが取り消しを有効 (使用可能) にしている場合、そのスレッドは終了し、ロックしている間にクリーンアップハンドラの実行を開始します。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、次の値を返します。

EINVAL

cv または *mp* で指定された値が無効です。

特定のスレッドのブロック解除

pthread_cond_signal(3THR)

pthread_cond_signal(3THR) は、*cv* が指す条件変数でブロックされている 1 つのスレッドのブロックを解除します。(Solaris スレッドについては、255ページの「cond_signal(3THR)」を参照)。

```
プロトタイプ:
int pthread_cond_signal(pthread_cond_t *cv);

#include <pthread.h>

pthread_cond_t cv;
int ret;

/* ある条件変数がシグナルを送る */
ret = pthread_cond_signal(&cv);
```

pthread_cond_signal() は、シグナルを送ろうとしている条件変数で使用されたものと同じ mutex ロックを獲得した状態で呼び出してください。そうしないと、関連する条件が評価されてから pthread_cond_wait() でブロック状態に入るまでの間に、条件変数にシグナルが送られる可能性があり、その場合 pthread_cond_wait() は永久に待ち続けることになります。

スケジューリング方針は、ブロックされたスレッドがどのように呼び起こされるかを決定します。SCHED_OTHER の場合、スレッドは優先順位に従って呼び起こされます。

スレッドがブロックされていない条件変数に対して pthread_cond_signal() を実行しても無視されます。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、次の値を戻します。

EINVAL

cv が指すアドレスが正しくありません。

例 4–8 pthread_cond_wait() と pthread_cond_signal() の使用例

```
pthread_mutex_t count_lock;
pthread_cond_t count_nonzero;
unsigned count;

decrement_count()
{
    pthread_mutex_lock(&count_lock);
    while (count == 0)
        pthread_cond_wait(&count_nonzero, &count_lock);
```

(続く)

```
    count = count - 1;
    pthread_mutex_unlock(&count_lock);
}

increment_count()
{
    pthread_mutex_lock(&count_lock);
    if (count == 0)
        pthread_cond_signal(&count_nonzero);
    count = count + 1;
    pthread_mutex_unlock(&count_lock);
}
```

時刻指定のブロック

pthread_cond_timedwait(3THR)

```
プロトタイプ:
int pthread_cond_timedwait(pthread_cond_t *cv,
    pthread_mutex_t *mp, const struct timespec *abstime);

#include <pthread.h>
#include <time.h>

pthread_cond_t cv;
pthread_mutex_t mp;
timestruct_t abstime;
int ret;

/* 条件変数で指定した時刻までブロック */
ret = pthread_cond_timedwait(&cv, &mp, &abstime);
```

pthread_cond_timedwait(3THR) は、*abstime* で指定した時刻を過ぎるとブロック状態を解除する点を除いて、pthread_cond_wait() と同じ動作をします。pthread_cond_timedwait() が戻るときは、たとえエラーを戻したときでも、常に mutex は呼び出しスレッドがロックして保持している状態です。(Solaris スレッドについては、254ページの「cond_timedwait(3THR)」を参照)。

pthread_cond_timedwait() のブロック状態が解除されるのは、条件変数にシグナルが送られてきたときか、一番最後の引数で指定した時刻を過ぎたときです。

注 - pthread_cond_timedwait() は、取り消しポイントでもあります。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、次の値を戻します。

EINVAL

cv または *abstime* が不当なアドレスを指しています。

ETIMEDOUT

abstime で指定された時刻を過ぎています。

時間切れの指定は時刻で行うため、時間切れ時刻を再計算する必要がないので、効率的に条件を再評価できます (詳細は、例 4–9 を参照してください)。

例 4–9 時刻指定のブロック

```
pthread_timestruc_t to;
pthread_mutex_t m;
pthread_cond_t c;
...
pthread_mutex_lock(&m);
to.tv_sec = time(NULL) + TIMEOUT;
to.tv_nsec = 0;
while (cond == FALSE) {
    err = pthread_cond_timedwait(&c, &m, &to);
    if (err == ETIMEDOUT) {
        /* 時間切れの場合の処理 */
        break;
    }
}
pthread_mutex_unlock(&m);
```

全スレッドのブロック解除

pthread_cond_broadcast(3THR)

```
プロトタイプ:
int pthread_cond_broadcast(pthread_cond_t *cv);

#include <pthread.h>

pthread_cond_t cv;
int ret;

/* 条件変数すべてがシグナルを受ける */
ret = pthread_cond_broadcast(&cv);
```

pthread_cond_broadcast(3THR) は、*cv* (pthread_cond_wait() で指定された) が指す条件変数でブロックされている、すべてのスレッドのブロックを解除します。スレッドがブロックされていない条件変数に対して pthread_cond_broadcast() を実行しても無視されます。(Solaris スレッドについては、255ページの「cond_broadcast(3THR)」を参照)。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、次の値を戻します。

EINVAL

cv が指すアドレスが正しくありません。

条件変数に対するブロードキャストの例

条件変数でブロックされていたすべてのスレッドが、もう一度相互排他ロックを争奪するようになるので慎重に使用してください。たとえば、pthread_cond_broadcast() を使用すると可変量のリソースに対して、そのリソースが解放される時にスレッド間で争奪させることができます (詳細は、例 4–10 を参照してください)。

例 4–10　条件変数に対するブロードキャスト

```
pthread_mutex_t rsrc_lock;
pthread_cond_t rsrc_add;
unsigned int resources;

get_resources(int amount)
{
    pthread_mutex_lock(&rsrc_lock);
    while (resources < amount) {
        pthread_cond_wait(&rsrc_add, &rsrc_lock);
    }
    resources -= amount;
    pthread_mutex_unlock(&rsrc_lock);
}

add_resources(int amount)
{
    pthread_mutex_lock(&rsrc_lock);
    resources += amount;
    pthread_cond_broadcast(&rsrc_add);
    pthread_mutex_unlock(&rsrc_lock);
}
```

上記のコード例の add_resources() で、次の点に注意してください。相互排他ロックの範囲内では、resources の更新と pthread_cond_broadcast() の呼び出しはどちらを先に行なってもかまいません。

pthread_cond_broadcast() は、シグナルを送ろうとしている条件変数で使用されたものと同じ相互排他ロックを獲得した状態で呼び出してください。そうしないと、関連する条件変数が評価されてから pthread_cond_wait() でブロック状態に入るまでの間に条件変数にシグナルが送られる可能性があり、その場合 pthread_cond_wait() は永久に待ち続けることになります。

条件変数の削除

pthread_cond_destroy(3THR)

pthread_cond_destroy(3T) は、*cv* が指す条件変数を削除します。(Solaris スレッドについては、253ページの「cond_destroy(3THR)」を参照)。

```
プロトタイプ:
int pthread_cond_destroy(pthread_cond_t *cv);

#include <pthread.h>

pthread_cond_t cv;
int ret;

/* 条件変数を削除する */
ret = pthread_cond_destroy(&cv);
```

条件変数の記憶領域は解放されません。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、次の値を戻します。

EINVAL

cv で指定された値が無効です。

「呼び起こし忘れ」問題

`pthread_cond_signal()` または `pthread_cond_broadcast()` を呼び出すとき、スレッドが条件変数に関連する相互排他ロックを保持していないと「呼び起こし忘れ」(lost wake-up) という問題が生じることがあります。

次のすべての条件に該当する場合は、そのシグナルには効果がないので「呼び起こし忘れ」が発生します。

- あるスレッドが `pthread_cond_signal()` または `pthread_cond_broadcast()` を呼び出す
- 別のスレッドが条件の評価と `pthread_cond_wait()` 呼び出しの間にある
- 待ち状態のスレッドがない

「生産者 / 消費者」問題

「生産者 / 消費者」問題は、並行プログラミングに関する問題の中でも一般によく知られているものの 1 つです。この問題は次のように定式化されます。サイズが有

限の 1 個のバッファと 2 種類のスレッドが存在します。一方のスレッドを生産者、もう一方のスレッドを消費者と呼びます。生産者がバッファにデータを入れ、消費者がバッファからデータを取り出します。

生産者は、バッファに空きができるまでデータを入れることができません。消費者は、バッファが空の間はデータを取り出すことができません。

特定の条件のシグナルを待つスレッドの待ち行列を条件変数で表すことにします。

例 4–11 では、そうした待ち行列として *less* と *more* の 2 つを使用しています。*less* はバッファ内の未使用スロットを待つ生産者のための待ち行列で、*more* は情報が格納されたバッファスロットを待つ消費者のための待ち行列です。また、バッファが同時に複数のスレッドによってアクセスされないようにするために、相互排他ロック (mutex ロック) も使用しています。

例 4–11 「生産者 / 消費者」問題と条件変数

```
typedef struct {
    char buf[BSIZE];
    int occupied;
    int nextin;
    int nextout;
    pthread_mutex_t mutex;
    pthread_cond_t more;
    pthread_cond_t less;
} buffer_t;

buffer_t buffer;
```

例 4–12 は、生産者側の処理です。最初に、mutex をロックしてバッファデータ構造 (`buffer`) を保護します。次に、これから作成するデータのための空きがあるか確認します。空きがない場合は、`pthread_cond_wait()` を呼び出して、「バッファ内に空きがある」を表す条件 *less* にシグナルが送られてくるのを待つスレッドの待ち行列に入ります。

同時に、`pthread_cond_wait()` の呼び出しによって、スレッドは mutex のロックを解除します。生産者スレッドは、条件が真になって消費者スレッドがシグナルを送ってくれるのを待ちます (詳細は、例 4–12 を参照してください)。条件にシグナルが送られてくると、*less* を待っている一番目のスレッドが呼び起こされます。しかし、そのスレッドは `pthread_cond_wait()` が戻る前に、mutex ロックを再び獲得する必要があります。

このようにして、バッファデータ構造への相互排他アクセスが保証されます。その後、生産者スレッドはバッファに本当に空きがあるか確認しなければなりません。空きがある場合は、最初の未使用スロットにデータを入れます。

このとき、バッファにデータが入れられるのを消費者スレッドが待っている可能性があります。そのスレッドは、条件変数 *more* で待ち状態となっています。生産者スレッドはバッファにデータを入れると、pthread_cond_signal() を呼び出して、待ち状態の最初の消費者を呼び起こします (待ち状態の消費者がいないときは、この呼び出しは無視されます)。

最後に、生産者スレッドは mutex ロックを解除して、他のスレッドがバッファデータ構造を操作できるようにします。

例 4–12 「生産者 / 消費者」問題 — 生産者

```
void producer(buffer_t *b, char item)
{
    pthread_mutex_lock(&b->mutex);

    while (b->occupied >= BSIZE)
        pthread_cond_wait(&b->less, &b->mutex);

    assert(b->occupied < BSIZE);

    b->buf[b->nextin++] = item;

    b->nextin %= BSIZE;
    b->occupied++;

    /* 現在の状態:  「b->occupied < BSIZE かつ b->nextin はバッファ内
       の次の空きスロットのインデックス」または「b->occupied == BSIZE
       かつ b->nextin は次の (占有されている) スロットのインデックス。これは
       消費者によって空にされる (例: b->nextin == b->nextout) 」 */

    pthread_cond_signal(&b->more);

    pthread_mutex_unlock(&b->mutex);
}
```

上記のコード例の assert() 文の用法に注意してください。コンパイル時に NDEBUG を定義しなければ、assert() は次のように動作します。すなわち、引数が真 (0 以外の値) のときは何も行わず、引数が偽 (0) のときはプログラムを強制的に終了させます。このように、実行時に発生した問題をただちに指摘できる点がマルチスレッドプログラムに特に適しています。assert() はデバッグのための有用な情報も与えてくれます。

/* 現在の状態: ... で始まるコメント部分も assert() で表現した方がよいかもしれません。しかし、論理式で表現するには複雑すぎるので、ここでは文章で表現しています。

上記の assert() やコメント部分の論理式は、どちらも不変式の例です。不変式は、あるスレッドが不変式中の変数を変更している瞬間を除いて、プログラムの実行により偽の値に変更されない論理式です (もちろん assert() の論理式は、どのスレッドがいつ実行した場合でも常に真であるべきです)。

不変式は非常に重要な手法です。プログラムテキストとして明示的に表現しなくても、プログラムを分析するときは不変式に置き換えて問題を考えることが大切です。

上記の生産者コード内のコメントで表現された不変式は、スレッドがそのコメントを含むコード部分を処理中には常に真となります。しかし、それを mutex_unlock() のすぐ後ろに移動すると、必ずしも常に真とはなりません。assert() のすぐ後ろに移動した場合は、真となります。

つまり、この不変式は、生産者または消費者がバッファの状態を変更しようとしているとき以外は、常に真となるような特性を表現しています。スレッドは mutex の保護下でバッファを操作しているとき、この不変式の値を一時的に偽にしてもかまいません。しかし、処理が完了したら不変式の値を再び真に戻さなければなりません。

例 4–13 は、消費者の処理です。この処理の流れは生産者の場合と対称的です。

例 4–13 「生産者 / 消費者」問題 — 消費者

```
char consumer(buffer_t *b)
{
    char item;
    pthread_mutex_lock(&b->mutex);
    while(b->occupied <= 0)
        pthread_cond_wait(&b->more, &b->mutex);

    assert(b->occupied > 0);

    item = b->buf[b->nextout++];
    b->nextout %= BSIZE;
    b->occupied--;

    /* 現在の状態:  「b->occupied > 0 かつ b->nextout はバッファ内の
       最初の占有されているスロットのインデックス」または「b->occupied == 0
       かつ b-> nextout は次の (未使用) スロットのインデックス。これは生産者側
       によっていっぱいにされる (例: b->nextout == b->nextin) 」 */

    pthread_cond_signal(&b->less);
    pthread_mutex_unlock(&b->mutex);
```

(続く)

```
    return(item);
}
```

セマフォ

セマフォは、E.W. ダイクストラ (Dijkstra) が 1960 年代の終わりごろに考案したプログラミング手法です。ダイクストラのセマフォモデルは、鉄道線路の運行をモデル化したものです。一度に一本の列車しか走れない単線の鉄道線路を思い浮かべてください。

この鉄道線路を保護するのがセマフォです。列車は単線区間に入るとき、セマフォの状態が進行許可状態になるのを待たなければなりません。列車が単線区間に入るとセマフォの状態は、他の列車が単線区間に入るのを禁止する状態に変化します。単線区間から出る列車は、セマフォの状態を進行許可状態に戻して他の列車が単線区間に入ることができるようにしなければなりません。

コンピュータ内のセマフォは、単一の整数で表現されます。スレッドは進行が許可されるのを待ち、その後進行したことを知らせるためにセマフォに対して P 操作を実行します。

この操作をもう少し具体的に説明しましょう。スレッドは、セマフォの値が正になるのを待たなければなりません。その後 1 を引くことでセマフォの値を変更します。これが P 操作です。処理を完了したセマフォは、V 操作を実行します。この操作は 1 を加えることでセマフォの値を変更します。ここで必ず守らなければならないことがあります。これらの各操作を原子操作により行うことです。これは操作が分断されると、操作途中にセマフォに対する別の操作が行われる危険性があるからです。P 操作では、1 を引く直前のセマフォの値が正でなければなりません (結果的に、引いた後の値が負にならないことと、その値が引く前の値よりも 1 だけ小さいことが保証されます)。

P 操作と V 操作のどちらの演算操作でも干渉が生じないようにしなければなりません。たとえば、同じセマフォに対して 2 つの V 操作が同時に行われた場合、そのセマフォの新しい値は最初よりも 2 だけ大きくなっていなければなりません。

ダイクストラがオランダ人だったこともあり、P と V の記号的な意味は現在ではほとんど忘れられています。参考までに、P はオランダ語の「prolagen」という単語を表します。その語源は「proberen te verlagen」で、「小さくする」という意味です。また、V は「verhogen」を表し、「大きくする」という意味です。このことは、ダイクストラのテクニカルノート『*EWD 74*』で説明されています。

`sema_wait(3R)` と `sema_post(3R)` は、ダイクストラの P 操作と V 操作にそれぞれ対応しています。また、`sema_trywait(3R)` は、P 操作の条件付きの形式です。この関数は、呼び出しスレッドがセマフォの値を差し引くために待たなければならない場合は、ただちに 0 以外の値を返します。

セマフォは、2 進セマフォとカウント用セマフォの 2 種類に大別されます。2 進セマフォは 0 と 1 のどちらかの値しかとりません。一方、カウント用セマフォは負以外の任意の値をとることができます。2 進セマフォは、論理的には相互排他ロック (mutex ロック) と似ています。

必須要件ではありませんが、mutex はロックを保持しているスレッドだけがそのロックを解放すべきものです。一方、セマフォには「スレッドがセマフォを保持している」という概念がないので、どのスレッドも V 操作 (すなわち、`sem_post(3R)`) を実行できます。

カウント用セマフォは、mutex とともに使用される条件変数と同等の能力があります。多くの場合、条件変数よりもカウント用セマフォを使用した方がコードが簡素化されます (詳細は、後述の例を参照してください)。

mutex とともに条件変数を使用する場合は、プログラムのどの部分を保護するかが自然な形で明らかになりました。ところが、セマフォでは必ずしもそうはなりません。強力だからといって安易に使うとプログラムが不統一で理解しにくくなります。このため、「並行プログラミングにおける *go to*」と呼ばれています。

計数型セマフォ

セマフォは、負の値をとらない整数のカウンタと考えることができます。通常は、リソースに対するアクセスの調整をはかる目的で、次のように使用されます。最初に、使用可能なリソースの数をセマフォに初期設定します。その後、スレッドはリソースが追加されるときにセマフォの値を原子的操作によって 1 増やし、リソースが削除されるときに原子的操作によって 1 減らします。

セマフォの値が 0 になった場合は、リソースがないことを意味します。この場合、セマフォの値を 1 減らそうとすると、スレッドはセマフォの値が 0 より大きくなるまでブロックされます。

表 4–7　セマフォに関するルーチン

操作	参照先
セマフォの初期化	142ページの「sema_init(3THR)」
セマフォの加算	145ページの「sema_post(3THR)」
セマフォの値によるブロック	145ページの「sema_wait(3THR)」
セマフォの減算	146ページの「sema_trywait(3THR)」
セマフォの削除	147ページの「sema_destroy(3THR)」

セマフォは、その獲得と解放を同じスレッドで行う必要がないため、シグナルハンドラで行われているような非同期のイベント通知を実現できます。また、セマフォ自身が状態を持っているため、条件変数を使用する場合と違って相互排他ロックを獲得しなくても非同期で使用できます。ただし、セマフォは相互排他ロックほど効率的ではありません。

セマフォで複数のスレッドがブロックされているとき、それらのスレッドがどの順番でブロック解除されるかは、特に指定しなければ不定です。

セマフォは、使用する前に初期化されている必要がありますが、属性はありません。

セマフォの初期化

sema_init(3THR)

```
プロトタイプ:
int sem_init(sem_t *sem, int pshared, unsigned int value);

#include <semaphore.h>

sem_t sem;
```

(続く)

```
int pshared;
int ret;
int value;

/* セマフォの初期化 */
pshared = 0;
value = 1;
ret = sem_init(&sem, pshared, value);
```

sema_init(3THR) は、*sem* が指すセマフォ変数を *value* の値に初期設定します。*pshared* の値が 0 なら、そのセマフォはプロセス間で共有できません。*pshared* の値が 0 以外なら、そのセマフォはプロセス間で共有できます。(Solaris スレッドについては、256ページの「sema_init(3THR)」を参照)。

複数のスレッドから同じセマフォを初期化してはいけません。

また、一度初期化したセマフォは、他のスレッドで使用されている可能性があるので再初期化してはいけません。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件のいずれかが検出されると、この関数は失敗し、次の値を戻します。

EINVAL

value の値が SEM_VALUE_MAX を超えています。

ENOSPC

そのセマフォを初期化するのに必要なリソースが使い果たされています。セマフォの制限 SEM_NSEMS_MAX に達しています。

EPERM

そのセマフォを初期化するのに必要な特権をそのプロセスがもっていません。

プロセス間スコープでセマフォを初期化する

pshared の値が 0 の場合は、そのプロセス内のスレッドだけがそのセマフォを使用できます。

```
#include <semaphore.h>

sem_t sem;
int ret;
int count = 4;

/* このプロセスでのみ使用 */
ret = sem_init(&sem, 0, count);
```

プロセス間スコープでセマフォを初期化する

pshared の値が 0 以外の場合は、他のプロセスによってそのセマフォは共有されます。

```
#include <semaphore.h>

sem_t sem;
int ret;
int count = 4;

/* プロセス間で共有 */
ret = sem_init(&sem, 1, count);
```

名前付きセマフォ

sem_open(3R)、sem_getvalue(3R)、sem_close(3R)、sem_unlink(3R) の各関数が、名前付きセマフォを開く、取得する、閉じる、削除するのにそれぞれ使用できます。sem_open() では、ファイルシステムの名前空間で名前が定義されたセマフォを生成できます。

名前付きセマフォはプロセス間で共有されるセマフォに似ていますが、*pshared* 値ではなくパス名で参照される点が異なります。

名前付きセマフォの詳細
は、sem_open(3R)、sem_getvalue(3R)、sem_close(3R)、sem_unlink(3R) のマニュアルページを参照してください。

セマフォの加算

sema_post(3THR)

```
プロトタイプ:
int sem_post(sem_t *sem);

#include <semaphore.h>

sem_t sem;
int ret;

ret = sem_post(&sem); /* セマフォを加算する */
```

sema_post(3THR) は、*sem* が指すセマフォの値を原子操作によって 1 増やします。そのセマフォでブロックされているスレッドがある場合は、そのスレッドのうちの 1 つのスレッドがブロック解除されます。(Solaris スレッドについては、257 ページの「sema_post(3THR)」を参照)。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、次の値を戻します。

EINVAL

sem が指すアドレスが正しくありません。

セマフォの値によるブロック

sema_wait(3THR)

```
プロトタイプ:
int sem_wait(sem_t *sem);

#include <semaphore.h>

sem_t sem;
int ret;
```

(続く)

```
ret = sem_wait(&sem); /* セマフォの値の変化を待つ */
```

sema_wait(3THR) は、*sem* が指すセマフォの値が 0 より大きくなるまでスレッドをブロックし、0 より大きくなったらセマフォの値を原子操作によって 1 減らします。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると、この関数は失敗し、次の値を返します。

EINVAL

sem が無効なアドレスを指しています。

EINTR

この関数にシグナルが割り込みを行いました。

セマフォの減算

sema_trywait(3THR)

```
プロトタイプ:
int sem_trywait(sem_t *sem);

#include <semaphore.h>

sem_t sem;
int ret;

ret = sem_trywait(&sem); /* セマフォの値の変化を待つ */
```

sema_trywait(3THR) は、*sem* が指すセマフォの値が 0 より大きい場合は原子操作によって 1 減らします。この関数はブロックしない点を除いて、sem_wait() と同じ働きをします。つまり、失敗した場合にはすぐに戻ります。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると、この関数は失敗し、次の値を返します。

`EINVAL`

sem が無効なアドレスを指しています。

`EINTR`

この関数にシグナルが割り込みを行いました。

`EAGAIN`

そのセマフォはすでにロックされているので、`sem_trywait()` でただちにロックできません。

セマフォの削除

sema_destroy(3THR)

```
プロトタイプ:
int sem_destroy(sem_t *sem);

#include <semaphore.h>

sem_t sem;
int ret;

ret = sem_destroy(&sem); /* セマフォを削除する */
```

`sema_destroy(3THR)` は、*sem* が指すセマフォを削除します。セマフォの記憶領域は解放されません。(Solaris スレッドについては、258ページの「sem_destroy(3RT)」を参照)。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、次の値を返します。

EINVAL

sem が指すアドレスが正しくありません。

「生産者 / 消費者」問題 ー セマフォを使った例

例 4–14 のデータ構造は、条件変数による「生産者 / 消費者」問題のコード例 (例 4–11 参照) のデータ構造と似ています。2 つのセマフォでそれぞれ、バッファの使用済スロット数と未使用スロット数を表します。これらのセマフォは、未使用スロットができるまで生産者を待たせ、使用済スロットができるまで消費者を待たせます。

例 4–14 「生産者 / 消費者」問題 ー セマフォを使った例

```
typedef struct {
    char buf[BSIZE];
    sem_t occupied;
    sem_t empty;
    int nextin;
    int nextout;
    sem_t pmut;
    sem_t cmut;
} buffer_t;

buffer_t buffer;

sem_init(&buffer.occupied, 0, 0);
sem_init(&buffer.empty,0, BSIZE);
sem_init(&buffer.pmut, 0, 1);
sem_init(&buffer.cmut, 0, 1);

buffer.nextin = buffer.nextout = 0;
```

ここでは、もう一組の (バイナリ) セマフォを使用しています。これは 2 値型セマフォで、相互排他ロック (mutex ロック) と同じ働きをします。この 2 つのセマフォは、複数の生産者と複数の未使用スロットが存在する場合と、複数の消費者と複数の使用済みスロットが存在する場合に、バッファへのアクセスを制御します。本来このような場合では mutex を使用すべきですが、セマフォの使用例を示すために特に使用しています。

例 4–15 「生産者 / 消費者」問題 — 生産者

```
void producer(buffer_t *b, char item) {
    sem_wait(&b->empty);

    sem_wait(&b->pmut);

    b->buf[b->nextin] = item;
    b->nextin++;
    b->nextin %= BSIZE;

    sem_post(&b->pmut);

    sem_post(&b->occupied);
}
```

例 4–16 「生産者 / 消費者」問題 — 消費者

```
char consumer(buffer_t *b) {
    char item;

    sem_wait(&b->occupied);

    sem_wait(&b->cmut);

    item = b->buf[b->nextout];
    b->nextout++;
    b->nextout %= BSIZE;

    sem_post(&b->cmut);

    sem_post(&b->empty);

    return(item);
}
```

読み取り / 書き込みロック属性

読み取り / 書き込みロック によって、保護された共有リソースに対する並行する複数の読み取りと排他的な書き込みが可能になります。読み取り / 書き込みロックは単一の実体で、「読み取り」または「書き込み」モードでロック可能です。リソースを変更するには、スレッドはまず排他的な書き込みロックを獲得する必要があり

ます。排他的に書き込みロックは、全ての読み取りロックが解放されるまで有効になりません。

データベースへのアクセスは読み取り / 書き込みロックに同期させることができます。読み取り操作によってレコードの情報が変更されることはないので、読み取り / 書き込みロックではデータベースのレコードを並行して読み取ることができます。データベースを更新する場合、書き込み操作は排他的な書き込みロックを獲得しなければなりません。

デフォルトの読み取り / 書き込みロック属性を変更するには、属性オブジェクトを宣言し、これを初期化しなければなりません。読み取り / 書き込みロック属性はアプリケーションのコードの開始位置にまとめて設定してある場合が多いので、その場所を素早く見つけて簡単に修正できます。ここで説明した読み取り / 書き込みロック属性を操作する関数を、次の表に示します。

読み書きロックに関する Solaris スレッドの実装については、228ページの「pthread に相当するものがある同期関数 — 読み取り / 書き込みロック」 を参照してください。

表 4–8 読み取り / 書き込みロック属性のルーチン

操作	参照先
読み取り／書き込みロック属性の初期化	151ページの「pthread_rwlockattr_init(3THR)」
読み取り／書き込みロック属性の削除	151ページの「pthread_rwlockattr_destroy(3THR)」
読み取り／書き込みロック属性の設定	152ページの「pthread_rwlockattr_setpshared(3THR)」
読み取り／書き込みロック属性の取得	153ページの「pthread_rwlockattr_getpshared(3THR)」

読み取り / 書き込みロック属性の初期化

pthread_rwlockattr_init(3THR)

```
#include <pthread.h>

int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```

pthread_rwlockattr_init(3THR) は、読み取り / 書き込みロック属性オブジェクト *attr* の、実装によって定義されたすべての属性を、デフォルト値に初期化します。

初期化済みの読み取り / 書き込みロック属性オブジェクトを指定して pthread_rwlockattr_init を呼び出した場合、その結果は未定義です。読み取り / 書き込みロック属性オブジェクトを使って初期化された読み取り / 書き込みロックは、属性オブジェクトに影響を与えるどんな関数（削除を含む）の影響も受けないためです。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。

ENOMEM

読み取り / 書き込みロック 属性オブジェクトを初期化するためのメモリーが足りません。

読み取り / 書き込みロック属性の削除

pthread_rwlockattr_destroy(3THR)

```
#include <pthread.h>

int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

pthread_rwlockattr_destroy(3THR) は、読み取り / 書き込みロック属性オブジェクトを削除します。削除したオブジェクトを、pthread_rwlockattr_init()

の呼び出しによって再び初期化する前に使った場合、その結果は未定義です。実装によっては、pthread_rwlockattr_destroy() は、*attr* が参照するオブジェクトに不正な値を設定する場合もあります。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。

EINVAL

attr が示す値は無効です。

読み取り / 書き込みロック属性の設定

pthread_rwlockattr_setpshared(3THR)

```
#include <pthread.h>

int pthread_rwlockattr_setpshared(pthread_rwlockattr_t  *attr, int  pshared);
```

pthread_rwlockattr_setpshared(3THR) は、プロセス共有の読み取り / 書き込みロック属性を設定します。

PTHREAD_PROCESS_SHARED

読み取り / 書き込みロックが割り当てられているメモリーにアクセスできるすべてのスレッドに、読み取り / 書き込みロックの操作を許可します。複数のプロセスによって共有されているメモリに置かれた読み取り / 書き込みロックに対しても有効です。

PTHREAD_PROCESS_PRIVATE

読み取り / 書き込みロックを操作できるのは、そのロックを初期化したスレッドと同じプロセス内で作成されたスレッドだけです。異なるプロセスのスレッドから読み取り / 書き込みロックを操作しようとした場合、その結果は未定義です。プロセス共有の属性のデフォルト値は、PTHREAD_PROCESS_PRIVATE です。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。

EINVAL

attr または *pshared* が示す値は無効です。

読み取り / 書き込みロック属性の取得

pthread_rwlockattr_getpshared(3THR)

```
#include <pthread.h>

int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t  *attr, int *pshared);
```

pthread_rwlockattr_getpshared(3THR) は、プロセス共有の読み取り / 書き込みロック属性を取得します。

pthread_rwlockattr_getpshared() は、*attr* が参照する初期化済みの属性オブジェクトから、プロセス共有の属性の値を取得します。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。

EINVAL

attr または *pshared* が示す値は無効です。

読み取り / 書き込みロックの使用

読み取り / 書き込みロックの属性を設定したあとに、読み取り / 書き込みロックそのものを初期化します。次の関数を使って、読み取り / 書き込みロックを初期化または削除したり、ロックまたはロック解除したり、ロックを試みたりできます。ここで説明した読み取り / 書き込みロック属性を操作する関数を、次の表に示します。

表 4–9　読み取り / 書き込みロック属性のルーチン

操作	参照先
読み取り／書き込みロックの初期化	154ページの「pthread_rwlock_init(3THR)」
読み取り／書き込みロックの読み取りロック	155ページの「pthread_rwlock_rdlock(3THR)」
非ブロック読み取り／書き込みロックの読み取りロック	156ページの「pthread_rwlock_tryrdlock(3THR)」
読み取り／書き込みロックの書き込みロック	157ページの「pthread_rwlock_wrlock(3THR)」
非ブロック読み取り／書き込みロックの書き込みロック	158ページの「pthread_rwlock_trywrlock(3THR)」
読み取り／書き込みロックの解除	159ページの「pthread_rwlock_unlock(3THR)」
読み取り／書き込みロックの削除	160ページの「pthread_rwlock_destroy(3THR)」

読み取り / 書き込みロックの初期化

pthread_rwlock_init(3THR)

```
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *rwlock, const pthread_rwlockattr_t *attr);

pthread_rwlock_t  rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

pthread_rwlock_init(3THR) により、*attr* を参照される属性を使用して、*rwlock* が参照する読み取り / 書き込みロックを初期化します。*attr* が NULL の場合、デフォルトの読み取り / 書き込みロック属性が使われます。この場合の結果は、デフォルトの読み取り / 書き込みロック属性オブジェクトのアドレスを渡す場合と同じです。いったん初期化したロックは、繰り返して使用するために再び初期化する

必要はありません。初期化が成功すると、読み取り / 書き込みロックは初期化され、ロックが解除された状態になります。初期化済みの読み取り / 書き込みロックを指定して、pthread_rwlock_init() を呼び出した場合、その結果は不定です。最初に初期化しないで読み取り / 書き込みロックを使用した場合も、その結果は不定です。Solaris スレッドについては、229ページの「rwlock_init(3THR)」を参照してください。

デフォルトの読み取り / 書き込みロック属性を使用するのであれば、PTHREAD_RWLOCK_INITIALIZER というマクロを使用して、静的に割り当てられている読み取り / 書き込みロックを初期化できます。この場合の結果は、パラメータ *attr* に NULL を指定して pthread_rwlock_init() を呼び出し、動的に初期化したときと同じです。ただし、エラーチェックが実行されません。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。

pthread_rwlock_init() が正常に終了しなかった場合、*rwlock* は初期化されず、*rwlock* の内容は未定義です。

EINVAL

attr または *rwlock* が示す値は無効です。

読み取り / 書き込みロックの読み取りロック

pthread_rwlock_rdlock(3THR)

```
#include <pthread.h>

int  pthread_rwlock_rdlock(pthread_rwlock_t *rwlock );
```

pthread_rwlock_rdlock(3THR) は、*rwlock* が参照する読み取り / 書き込みロックに読み取りロックを適用します。書き込みがロックを保持せず、読み取り / 書き込みロックでブロックされている書き込みもない場合は、呼び出しスレッドは読み取りロックを獲得します。書き込みがロックを保持せず、ロック待ちの書き込みがある場合は、呼び出しスレッドが読み取りロックを獲得するかどうかは不定です。書き込みが読み取り / 書き込みロックを保持している場合は、呼び出しスレッドは読み

取りロックを獲得しません。読み取りロックが獲得されない場合、呼び出しスレッドは読み取りロックを獲得するまでブロックします。つまり、呼び出しスレッドは、pthread_rwlock_rdlock() から戻り値を取得しません。呼び出し時に、呼び出しスレッドが *rwlock* に書き込みロックを保持する場合、その結果は不定です。

書き込み側がいつまでもロックを獲得できない事態を避けるために、書き込みが読み取りに優先するように実装できます。たとえば、Solaris スレッドの実装では、書き込みが読み取りに優先します。231ページの「rw_rdlock(3THR) 」 を参照してください。

スレッドは、*rwlock* に複数の並行的な読み取りロックを保持できます。つまり、pthread_rwlock_rdlock() の呼び出しが *n* 回成功します。この場合、スレッドは同数の読み取りロック解除を行わなければなりません。つまり、pthread_rwlock_unlock() を *n* 回呼び出さなければなりません。

pthread_rwlock_rdlock() が、初期化されていない読み取り / 書き込みロックに対して呼び出された場合、その結果は不定です。

読み取りのための読み取り / 書き込みロックを待っているスレッドにシグナルが送られた場合、スレッドはシグナルハンドラから戻ると、見かけ上割り込みがなかった場合と同様に、読み取りのための読み取り / 書き込みロック待ちを再開します。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。

EINVAL

attr または *rwlock* が示す値は無効です。

非ブロック読み取り / 書き込みロックの読み取りロック

pthread_rwlock_tryrdlock(3THR)

```
#include <pthread.h>

int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

pthread_rwlock_tryrdlock(3THR) は、pthread_rwlock_rdlock() と同様に読み取りロックを適用します。ただし、いずれかのスレッドが *rwlock* に書き込みロックを保持しているか、*rwlock* で書き込みスレッドがブロックされている場合、この関数は失敗します。Solaris スレッドについては、232ページの「rw_tryrdlock(3THR)」 を参照してください。

戻り値

rwlock が参照する読み取り / 書き込みロックオブジェクトに対する読み取りロックが獲得された場合、戻り値は 0 です。それ以外の戻り値は、エラーが発生したことを示します。

EBUSY

書き込みが読み取り / 書き込みロックを保持しているか、読み取り / 書き込みロックで書き込みスレッドがブロックされているため、読み取りのための読み取り / 書き込みロックを獲得できません。

読み取り / 書き込みロックの書き込みロック

pthread_rwlock_wrlock(3THR)

```
#include <pthread.h>

int  pthread_rwlock_wrlock(pthread_rwlock_t *rwlock );
```

pthread_rwlock_wrlock(3THR) は、*rwlock* が参照する読み取り / 書き込みロックに書き込みロックを適用します。ほかのスレッド (読み取り側または書き込み側) が *rwlock* という読み取り / 書き込みロックを保持していない場合、呼び出しスレッドは書き込みロックを獲得します。これ以外の場合、スレッドは、ロックを獲得するまでブロックされます。つまり、pthread_rwlock_wrlock() の呼び出しから戻りません。呼び出し時に、呼び出しスレッドが読み取り / 書き込みロックを保持している場合 (読み取りロックと書き込みロックのどちらでも) の結果は不定です。

書き込み側がいつまでもロックを獲得できない事態を避けるために、書き込みが読み取りに優先するように実装することが許されています。たとえば、Solaris スレッ

ドの実装では、書き込みが読み取りに優先します。233ページの「rw_wrlock(3THR)」を参照してください。

pthread_rwlock_wrlock() が、初期化されていない読み取り / 書き込みロックに対して呼び出された場合、その結果は不定です。

書き込みのための読み取り / 書き込みロックを待っているスレッドにシグナルが送られた場合、スレッドはシグナルハンドラから戻ると、見かけ上割り込みがなかった場合と同様に、書き込みのための読み取り / 書き込みロック待ちを再開します。

戻り値

rwlock が参照する読み取りは / 書き込みロックオブジェクトの書き込みロックが獲得された場合、あり得る戻りの記述が存在しないことを示します。

非ブロック読み取り / 書き込みロックの書き込みロック

pthread_rwlock_trywrlock(3THR)

```
#include <pthread.h>

int pthread_rwlock_trywrlock(pthread_rwlock_t  *rwlock);
```

pthread_rwlock_trywrlock(3THR) は、pthread_rwlock_wrlock() と同様に書き込みロックを適用します。ただし、いずれかのスレッドが現時点で *rwlock* (読み取り用または書き込み用) を保持している場合、この関数は失敗します。Solaris スレッドについては、234ページの「rw_trywrlock(3THR)」を参照してください。

pthread_rwlock_trywrlock() が、初期化されていない読み取り / 書き込みロックに対して呼び出された場合、その結果は不定です。

書き込みのための読み取り / 書き込みロックを待っているスレッドにシグナルが送られた場合、スレッドはシグナルハンドラから戻ると、見かけ上割り込みがなかった場合と同様に、書き込みのための読み取り / 書き込みロック待ちを再開します。

戻り値

rwlock が参照する読み取り / 書き込みロックオブジェクトの書き込みロックを獲得した場合、戻り値は 0 です。それ以外の戻り値は、エラーが発生したことを示します。

EBUSY

読み取りまたは書き込みでロック済みのため、書き込みのための読み取り / 書き込みロックを獲得できません。

読み取り / 書き込みロックの解除

pthread_rwlock_unlock(3THR)

```
#include <pthread.h>
```

pthread_rwlock_unlock(3THR) は、*rwlock* が参照する読み取り / 書き込みロックオブジェクトに保持されたロックを解放します。呼び出しスレッドが *rwlock* という読み取り / 書き込みロックを保持していない場合、その結果は不定です。Solaris スレッドについては、234ページの「rw_unlock(3THR) 」 を参照してください。

pthread_rwlock_unlock を呼び出して読み取り / 書き込みロックオブジェクトから読み取りオブジェクトを解放しても、この読み取り / 書き込みロックオブジェクトに他の読み取りロックが保持されている場合、読み取り / 書き込みロックオブジェクトは読み取りにロックされたままになります。pthread_rwlock_unlock() が、呼び出しスレッドによる最後の読み取りロックを解放すると、呼び出しスレッドはこのオブジェクトの所有者でなくなります。pthread_rwlock_unlock() がこの読み取り / 書き込みロックオブジェクトの最後の読み取りロックを解放すると、読み取り / 書き込みロックオブジェクトはロックが解除され、所有者のない状態になります。

pthread_rwlock_unlock() を呼び出し、読み取り / 書き込みロックオブジェクトから書き込みオブジェクトを解放すると、読み取り / 書き込みロックオブジェクトはロックが解除され、所有者のない状態になります。

pthread_rwlock_unlock() を呼び出した結果として読み取り / 書き込みロックオブジェクトがロック解除されたときに、複数のスレッドが書き込みのための読み取り / 書き込みロックオブジェクトの獲得を待っている場合は、スケジューリン

グ方針を使用して、書き込みのための読み取り / 書き込みロックオブジェクトを獲得するスレッドが決定されます。また、複数のスレッドが読み取りのための読み取り / 書き込みロックオブジェクトの獲得を待っている場合も、スケジューリング方針を使用して、読み取りのための読み取り / 書き込みロックオブジェクトを獲得するスレッドの順番が決定されます。さらに、複数のスレッドが読み取りロックと書き込みロック両方のために *rwlock* にブロックされている場合は、読み取り側と書き込み側のどちらが先にロックを獲得するのかは規定されていません。

`pthread_rwlock_unlock()` が、初期化されていない読み取り / 書き込みロックに対して呼び出された場合、その結果は不定です。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。

読み取り / 書き込みロックの削除

pthread_rwlock_destroy(3THR)

```
#include <pthread.h>

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);

pthread_rwlock_t  rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

`pthread_rwlock_destroy(3THR)` は、*rwlock* が示す読み取り / 書き込みロックオブジェクトを削除し、このロックで使用されていたリソースを解放します。削除したオブジェクトを、`pthread_rwlock_init()` の呼び出しによって再び初期化する前に使用した場合、その結果は不定です。実装によっては、`pthread_rwlock_destroy()` は、*rwlock* が参照するオブジェクトに不正な値を設定する場合もあります。いずれかのスレッドが *rwlock* を保持しているときに `pthread_rwlock_destroy()` を呼び出した場合の結果は不定です。初期化されていない読み取り / 書き込みロックを削除しようとした場合に発生する動作も不定です。また、削除された読み取り / 書き込みロックオブジェクトは、再度 `pthread_rwlock_init()` で初期化できます。削除した読み取り / 書き込みロックオブジェクトを初期化せずに参照した場合も不定です。Solaris スレッドについては、235ページの「rwlock_destroy(3THR) 」 を参照してください。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。

EINVAL

attr または *rwlock* が示す値は無効です。

プロセスの境界を越えた同期

今までに説明した同期プリミティブは、プロセスの境界を越えて使用するように設定できます。具体的には次のようにします。まず、その同期変数の領域が共有メモリーに確保されるようにし、次に該当する `init()` ルーチンを呼び出しますが、これはそのプリミティブをプロセス間共有属性で初期化した後に行います。

「生産者 / 消費者」問題の例

例 4–17 は、前述の「生産者 / 消費者」問題の生産者と消費者をそれぞれ別のプロセスで表現したものです。メインルーチンは、0 に初期化されたメモリーを自分のアドレス空間にマッピングし、子プロセスと共有します。

子プロセスが 1 つ生成され、消費者の処理が実行されます。親プロセスは生産者の処理を実行します。

この例では、生産者と消費者を呼び出す各駆動ルーチンも示しています。`producer_driver()` は `stdin` から文字を読み込み、`producer()` を呼び出します。`consumer_driver()` は `consumer()` を呼び出して文字を受け取り、`stdout` に書き出します。

例 4–17 のデータ構造は、条件変数による「生産者 / 消費者」の例のデータ構造 (例 4–4 を参照) と同じです。2 つのセマフォでそれぞれ、いっぱいになったバッファ数と未使用バッファ数を表します。これらのセマフォは、未使用バッファができるまで生産者を待たせ、バッファがいっぱいになるまで消費者を待たせます。

例 4–17 プロセスの境界を越えた同期の例

```
main() {
    int zfd;
```

(続く)

```
    buffer_t *buffer;
    pthread_mutexattr_t mattr;
    pthread_condattr_t cvattr_less, cvattr_more;

    zfd = open("/dev/zero", O_RDWR);
    buffer = (buffer_t *)mmap(NULL, sizeof(buffer_t),
        PROT_READ|PROT_WRITE, MAP_SHARED, zfd, 0);
    buffer->occupied = buffer->nextin = buffer->nextout = 0;

    pthread_mutex_attr_init(&mattr);
    pthread_mutexattr_setpshared(&mattr,
        PTHREAD_PROCESS_SHARED);

    pthread_mutex_init(&buffer->lock, &mattr);
    pthread_condattr_init(&cvattr_less);
    pthread_condattr_setpshared(&cvattr_less, PTHREAD_PROCESS_SHARED);
    pthread_cond_init(&buffer->less, &cvattr_less);
    pthread_condattr_init(&cvattr_more);
    pthread_condattr_setpshared(&cvattr_more,
        PTHREAD_PROCESS_SHARED);
    pthread_cond_init(&buffer->more, &cvattr_more);

    if (fork() == 0)
        consumer_driver(buffer);
    else
        producer_driver(buffer);
}

void producer_driver(buffer_t *b) {
    int item;

    while (1) {
        item = getchar();
        if (item == EOF) {
            producer(b, '\0');
            break;
        } else
            producer(b, (char)item);
    }
}

void consumer_driver(buffer_t *b) {
    char item;

    while (1) {
        if ((item = consumer(b)) == '\0')
            break;
        putchar(item);
    }
}
```

スレッドライブラリによらないプロセス間ロック

一般的には推奨できる方法ではありませんが、Solaris スレッドでスレッドライブラリを使用せずにプロセス間ロックを行うことも可能です。詳細は、259ページの「プロセス間での LWP の使用」 を参照してください。

プリミティブの比較

スレッドで使われる最も基本的な同期プリミティブは、相互排他ロックです。相互排他ロックは、メモリー使用量と実行時間の両面で最も効率的な機構です。相互排他ロックの主要目的は、リソースへのアクセスを直列化することです。

相互排他ロックに次いで効率的なプリミティブは、条件変数です。条件変数の主要目的は、状態の変化に基づいてスレッドをブロックすることです。つまり、スレッド待ち機能の提供です。条件変数でスレッドをブロックする場合は、その前に相互排他ロックを獲得しなければなりません。また、`pthread_cond_wait()` から戻った後に相互排他ロックを解除しなければいけません。また、対応する `pthread_cond_signal()` 呼び出しまで状態の変更が行われる間、相互排他ロックを保持しておかなければなりません。

セマフォは、条件変数より多くのメモリーを消費しますが、状況によっては条件変数よりも簡単に使用できます。セマフォ変数は、制御でなく状態に基づいて機能するからです。また、ロックのように保持するという概念もありません。スレッドをブロックしているセマフォに対して、どのスレッドもセマフォの値を 1 増やすことができます。

読み取り / 書き込みロックを使用すると、保護されたリソースに対する、並行する複数の読み取り操作や排他的な書き込み操作ができます。読み取り / 書き込みロックは単一の実体で、読み取りモードまたは 書き込み モードでロック可能です。リソースを変更するには、まずスレッドは排他書き込みロックを取得する必要があります。排他書き込みロックは、すべてのロックが解放されるまで使用できません。

第 5 章

オペレーティング環境が関係するプログラミング

この章では、マルチスレッドと Solaris オペレーティング環境との関係について説明します。また、マルチスレッドをサポートするために Solaris オペレーティング環境に、どのような変更が加えられたかについても説明します。

プロセスの生成 − fork

Solaris オペレーティング環境における `fork()` 関数のデフォルト処理は、POSIX スレッドでの `fork()` の処理方法とはいくらか違っています。ただし、Solaris オペレーティング環境は両方の機構をサポートしています。

表 5–1 は、Solaris と `pthread` での `fork()` の処理について、相違点と類似点を示しています。POSIX スレッドまたは Solaris スレッドの側に相当するインタフェースがない項目については、「—」が記入されています。

表 5–1　POSIX と Solaris での fork() の処理の比較

	Solaris オペレーティング環境のインタフェース	**POSIX** スレッドのインタフェース
fork1 モデル	fork1(2)	fork(2)
汎用 fork モデル	fork(2)	—
fork - 安全	—	pthread_atfork(3T)

fork1 モデル

表 5–1 で示すように、pthread の fork(2) 関数の動作は、Solaris の fork1(2) 関数の動作と同じです。pthread の fork(2) 関数と Solaris の fork1(2) 関数はどちらも新しいプロセスを生成し、子プロセスに完全なアドレス空間の複製を作成しますが、スレッドについては呼び出しスレッドのみを複製します。

これは、子プロセスが生成後ただちに exec() を呼び出すような場合に利用します。実際、多くの場合に fork() を呼び出した後行われることです。この場合、子プロセスは fork() を呼び出したスレッド以外のスレッドの複製は必要としません。

子プロセスでは、fork() を呼び出してから exec() を呼び出すまでの間に、ライブラリ関数を呼び出さないようにします。ライブラリ関数の中には、fork() 呼び出し時に親の中で保持されているロックを使用するものがあるからです。子プロセスは exec() ハンドラの 1 つが呼び出されるまで、「非同期シグナル安全」操作しか行えません。

fork1 モデルにおける安全性の問題とその解決策

共有データのロックのような通常の考慮事項に加えて、次のような問題があります。実行されているスレッドが 1 つ (fork() を呼び出したスレッド) しかないときに、ライブラリは子プロセスを fork することに関して上手に処理しなくてはなりません。この場合の問題は、子プロセスの唯一のスレッドが、その子プロセスに複製されなかったスレッドによって保持されているロックを占有しようとする可能性があることです。

これは、多くのプログラムが遭遇するような問題ではありません。ほとんどのプログラムは、fork() から復帰した直後に子プロセス内で exec() を呼び出します。しかし、子プロセス内で何かの処理を行なってから exec() を呼び出す場合、または exec() をまったく呼び出さない場合、子プロセスはデッドロックに遭遇するでしょう。

ライブラリの作成者は安全な解決策を提供してください。もっとも、fork に対して安全なライブラリを提供しなくても (このような状況が稀であるため) 大きな問題にはなりません。

たとえば、T1 が何かを出力している途中で (その間、printf() のためにロックを保持している)、T2 が新しいプロセスを fork すると仮定します。この場合、子プロセス内で唯一のスレッド (T2) が printf() を呼び出せば、すぐさまデッドロックに陥ります。

POSIX の fork() と Solaris の fork1() は、それを呼び出したスレッドのみを複製します。(Solaris の fork() を呼び出せば、すべてのスレッドが複製されるので、この問題は生じません。)

デッドロックを防ぐには、fork 時にこのようなロックが保持されないようにしなければなりません。そのための最も明瞭なやり方は、fork を行うスレッドに、子プロセスによって使われる可能性のあるロックをすべて獲得させることです。printf() でそのようなことはできないので (printf() は libc によって所有されているため)、fork() の呼び出しは printf() を使用していない状態で行うようにしなければなりません。

ライブラリでロックを管理するには、次の操作を実行します。

- そのライブラリで使用するすべてのロックを明確に指定します。
- そのライブラリで使用するロックのロック順序を明確に指定します。(厳密なロック順序を使用しない場合は、ロックの獲得を管理する上で細心の注意が必要です。)
- fork 呼び出し時にそれらのロックを獲得できるよう段取りします。Solaris スレッドでは、これを手作業で行わなければなりません。fork1() を呼び出す直前にロックを獲得し、その後ただちに解放します。

次の例では、ライブラリによって使用されるロックのリストは {L1,...Ln} で、これらのロックのロック順序も L1...Ln です。

```
mutex_lock(L1);
mutex_lock(L2);
```

(続く)

```
fork1(...);
mutex_unlock(L1);
mutex_unlock(L2);
```

pthread では、pthread_atfork(f1, f2, f3) の呼び出しをライブラリの .init() セクションに追加できます。f1、f2、f3 の定義は次のとおりです。

```
f1() /* このプロセスが fork する前に実行される */
{
 mutex_lock(L1); |
 mutex_lock(...); | -- ロックの順に並べる
 mutex_lock(Ln); |
 } V

f2() /* このプロセスが fork した後に子の中で実行される */
 {
 mutex_unlock(L1);
 mutex_unlock(...);
 mutex_unlock(Ln);
 }

f3() /* このプロセスが fork した後に親の中で実行される */
 {
 mutex_unlock(L1);
 mutex_unlock(...);
 mutex_unlock(Ln);
 }
```

デッドロックのもう 1 つの例として、mutex をロックした、親プロセス内のスレッド (Solaris の fork1(2) を呼び出したものではない) が考えられます。この mutex はロック状態で子プロセスにコピーされますが、その mutex をロック解除するためのスレッドはコピーされません。このため、その mutex をロックしようとする子プロセス内のスレッドは永久に待つことになります。

仮想 *fork* — *vfork(2)*

標準の vfork(2) 関数は、マルチスレッドプログラムでは危険です。vfork(2) は、呼び出しスレッドだけを子プロセスにコピーする点が fork1(2) に似ています。ただし、スレッドに対応した実装ではないので、vfork() は子プロセスにアドレス空間をコピーしません。

子プロセス内のスレッドで、exec(2) を呼び出す前にメモリーを変更しないよう十分注意してください。vfork() では、親プロセスのアドレス空間が子プロセスにそのまま渡されます。子プロセスが exec() を呼び出すか終了すると、親プロセスにアドレス空間が戻されます。したがって、子プロセスが、親プロセスの状態を変更しないようにすることが大切です。

たとえば、vfork() を呼び出してから exec() を呼び出すまでの間に、新しいスレッドを生成することは大変危険です。これが問題となるのは、fork1 モデルを使用した場合と、子プロセスが exec() の呼び出しの他にも何か行う場合だけです。ほとんどのライブラリは「fork - 安全」ではないので、pthread_atfork() を使用することによって fork に対する安全性を実装してください。

解決策 − *pthread_atfork(3T)*

pthread_atfork() を使用すれば、fork1 モデルを使用したときのデッドロックが防止されます。

```
#include <pthread.h>

int pthread_atfork(void (*prepare) (void), void (*parent) (void),
    void (*child) (void) );
```

pthread_atfork() 関数は、fork() を呼び出したスレッドのコンテキストで fork() の前後に呼び出される fork のハンドラを宣言します。

- *prepare* ハンドラは fork() の起動前に呼び出されます。
- *parent* ハンドラは fork() の復帰後に親の中で呼び出されます。
- *child* ハンドラは fork() の復帰後に子の中で呼び出されます。

これらのどれでも NULL に設定できます。連続する pthread_atfork() 呼び出しの順序が重要です。

たとえば、*prepare* ハンドラが、必要な相互排他ロックをすべて獲得し、次に *parent* ハンドラと *child* ハンドラがそれらを解放するといった具合です。このようにすると、プロセスが fork される「前」に、関係するロックがすべて fork 関数を呼び出すスレッドによって保持されるので、子プロセスでのデッドロックが防止されます。

汎用 fork モデルを使用すれば、166ページの「fork1 モデルにおける安全性の問題とその解決策」で述べたデッドロックの問題は回避されます。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、この関数は失敗し、次の値を戻します。

ENOMEM

テーブル空間が足りないので、fork ハンドラのアドレスを記録できません。

汎用 fork モデル

Solaris の fork(2) 関数は、子プロセスにアドレス空間とすべてのスレッド (および LWP) の複製を作成します。この方法を使用するのは、子プロセスで exec(2) をまったく呼び出さないが親のアドレス空間のコピーを使用するなどの場合です。汎用 fork 機能は POSIX スレッドにはありません。

なお、プロセス内のあるスレッドが Solaris の fork(2) を呼び出すと、割り込み可能なシステムコール処理中にブロックされたスレッドは EINTR を戻すので注意してください。

また、親プロセスと子プロセスの両方に保持されるロックを作成しないよう十分注意してください。このような状況が生じる可能性があるのは、ロックを共有可能なメモリー上に割り当てている (つまり、mmap() で MAP_SHARED フラグを指定した) 場合です。fork 1 モデルを使用する場合、これは問題になりません。

正しい fork の選択

アプリケーションの中での fork() のセマンティクスが「汎用 fork」と「fork 1」のどちらであるかは、該当するライブラリとリンクすることによって決定されます。–lthread を指定してリンクすると、fork() のセマンティクスは「汎用 fork」になり、–lpthread を指定してリンクすると、fork() のセマンティクスは「fork 1」になります (コンパイルオプションの説明は、図 7–1 を参照してください)。

すべての fork に関係する注意事項

どの fork() 関数についても、呼び出した後で広域的状態を使用するときには注意が必要です。

たとえば、あるスレッドがファイルを逐次的に読み取っているときに同じプロセス内の別のスレッドが fork() 関数を 1 つ呼び出すと、両方のプロセスにファイル読

み取り中のスレッドが存在することになります。fork() 後はファイル記述子のシークポインタが共有されるため、親プロセス内のスレッドがデータを読み取ると、子プロセス内のスレッドは残りのデータを読み取ります。この結果、連続読み取りアクセスに切れ目ができます。

プロセスの作成 ー exec(2) と exit(2) について

システムコール exec(2) と exit(2) の動作は、アドレス空間内のすべてのスレッドを削除する点を除いて、シングルスレッドのプロセスの場合と変わりません。どちらのシステムコールも、スレッドを含むすべての実行リソースが削除されるまでブロック状態になります。

exec() は、プロセスを再構築するときに LWP を 1 つ生成します。さらにプロセス起動時に初期スレッドを生成します。通常、初期スレッドが処理を終えると exit() を呼び出し、プロセスは削除されます。

プロセス内のすべてのスレッドが終了すると、そのプロセスも終了します。複数のスレッドをもつプロセスから exec() 関数が呼び出されると、すべてのスレッドが終了し、新しい実行可能イメージがロードされ実行されます。デストラクタ関数は呼び出されません。

タイマ、アラーム、およびプロファイル

LWP ごとのタイマ (timer_create(3R) を参照) とスレッドごとのアラーム (alarm(2) または setitimer(2) を参照) についての「サポート中止」のご案内が Solaris 2.5 リリースでされています。どちらの機能も、この節で説明するプロセスごとの代替物によって補足されています。

各 LWP は、その LWP に結合されているスレッドが使用できるリアルタイムインタバルタイマとアラームを持っています。このタイマとアラームは、一定時間が経過すると 1 つのシグナルをスレッドに送ります。

各 LWP は、その LWP に結合されているスレッドが使用できる仮想時間インタバルタイマ、またはプロファイル用のインタバルタイマも持っています。このインタバルタイマは一定時間が経過すると、それを所有している LWP に SIGVTALRM シグナルまたは SIGPROF シグナルを送ります。

LWP ごとの POSIX タイマ

Solaris 2.3 と 2.4 リリースでは、`timer_create(3R)` 関数が戻すタイマオブジェクトは、そのタイマ ID が呼び出し LWP の中だけで意味をもち、その期限切れシグナルが呼び出し LWP に送られるというものでした。このため、POSIX タイマ機能を使用できるスレッドは、結合スレッドに限られていました。

さらに、この制限された使用方法でも、Solaris 2.3 と 2.4 リリースのマルチスレッドアプリケーションでの POSIX タイマは、生成されるシグナルのマスキングおよび `sigvent` 構造体からの関連値の送信について信頼性に欠けるところがありました。

Solaris 2.5 以降のリリースでは、マクロ `_POSIX_PER_PROCESS_TIMERS` を定義してコンパイルされたアプリケーション、あるいはシンボル `_POSIX_C_SOURCE` に対して 199506L より大きな値を指定してコンパイルされたアプリケーションは、プロセスごとのタイマを作成できます。

Solaris 2.5 リリースより前のリリースでコンパイルされたアプリケーション、あるいは機能評価マクロを使わずにコンパイルされたアプリケーションは、引き続き LWP ごとの POSIX タイマを作成します。将来のリリースでは、LWP ごとのタイマを作成するための呼び出しが、プロセスごとのタイマを戻すようになる予定です。

プロセスごとのタイマのタイマ ID は、どの LWP からでも使用できます。期限切れシグナルは、特定の LWP に向けられるのではなく、そのプロセスに対して生成されます。

プロセスごとのタイマは、`timer_delete(3R)` の呼び出し時またはそのプロセスの終了時にのみ削除されます。

スレッドごとのアラーム

Solaris オペレーティング環境 2.3 と 2.4 リリースでは、`alarm(2)` または `setitimer(2)` の呼び出しは、呼び出し LWP の中だけで意味をもっていました。生成した LWP が終了すると、こうしたタイマは自動的に削除されました。このため、`alarm()` や `setitimer()` を使用できるスレッドは、結合スレッドに限られていました。

さらに制限された使用方法でも、Solaris オペレーティング環境 2.3 と 2.4 のマルチスレッドアプリケーションでの `alarm()` タイマと `setitimer()` タイマは、これらの呼び出しを行なった結合スレッドからのシグナルのマスキングについて信頼性に欠けるところがありました。このようなマスキングが必要とされなければ、結合スレッドから出された、これら 2 つのシステムコールの動作は信頼できるものでした。

Solaris オペレーティング環境 2.5 以降のリリースでは、–lpthread (POSIX) スレッドとリンクしたアプリケーションは、alarm() を呼び出したときにプロセスごとの SIGALRM 通知を受けるようになります。alarm() で生成される SIGALRM は、特定の LWP に向けられるのではなく、そのプロセスに対して生成されます。このアラームは、そのプロセスの終了時にリセットされます。

Solaris オペレーティング環境 2.5 リリースより前のリリースでコンパイルされたアプリケーション、あるいは –lpthread とリンクされていないアプリケーションは、alarm() または setitimer() で生成されるシグナルの、LWP ごとの送信を引き続き行います。

将来のリリースでは、ITIMER_REAL フラグを指定した alarm() または setitimer() の呼び出しによって、SIGALRM がそのプロセスに送られる予定です。他のフラグについては、setitimer() で引き続き LWP ごとの送信が行われる予定です。setitimer() のフラグで ITIMER_REAL フラグ以外のものについては、生成されるシグナルが、その呼び出しを行なった LWP に送信されることに変わりはなく、したがって結合スレッドからしか使用できません。

プロファイル

profil(2) で、各 LWP に専用のバッファ、または複数の LWP で共有のバッファを用意することにより、LWP ごとにプロファイルを有効にすることが可能です。プロファイルデータの更新は、LWP ユーザ時間のクロック更新単位ごとに行われます。プロファイルの状態は、生成元の LWP から継承されます。

大域ジャンプ ー setjmp(3C) と longjmp(3C)

setjmp() と longjmp() の有効範囲は、1 つのスレッド内だけに制限されます。この制限は、ほとんどの場合は問題となりません。しかし、この制限は、シグナルを扱うスレッドが longjmp() を使用できるのは、setjmp() が同一スレッド内で実行されている場合だけであることを意味します。

リソースの制限

リソースの制限は、そのプロセス全体に課せられ、プロセス内のすべてのスレッドが全体でどれだけリソースを使用しているかによって決まります。リソースの弱い制限値を超えた場合は、制限に違反したスレッドにシグナルが送られます。プロセス内で使用されているリソースの合計は、`getrusage(3B)` で調べることができます。

LWP とスケジューリングクラス

第 1 章の「スケジューリング」の節で説明しているように、Solaris の `pthread` 実装でサポートしているスケジューリング方針は `SCHED_OTHER` だけです。それ以外は POSIX のオプションです。

POSIX の `SCHED_FIFO` 方針と `SCHED_RR` 方針は、Solaris 標準の機構を使って複製またはエミュレートできます。この節では、これらのスケジューリング機構について説明します。

Solaris のカーネルには、プロセスのスケジューリングに関する 3 つのクラスがあります。最も優先順位が高いスケジューリングクラスは、リアルタイム (`RT`) クラスです。その次はシステムクラスで、ユーザプロセスには適用されません。最も低いのはタイムシェア (`TS`) クラスで、デフォルトのスケジューリングクラスです。

スケジューリングクラスは、LWP ごとに維持管理されます。プロセスが生成されると、そのプロセスの初期 LWP は、親プロセスのスケジューリングクラスと作成元の LWP の優先順位を継承します。その後に、非結合スレッドを実行させるために生成される LWP も、このスケジューリングクラスと優先順位を継承します。

プロセス内のすべての非結合スレッドは、同じスケジューリングクラスと優先順位が与えられます。各スケジューリングクラスは、そのクラスに設定可能な優先順位に従って、スケジューリング対象の LWP の優先順位を、全体のディスパッチ優先順位に対応付けます。

結合スレッドは、結合している LWP と同じスケジューリングクラスと優先順位をもちます。あるプロセス内の各結合スレッドは、カーネルから参照可能な固有のスケジューリングクラスと優先順位を持っています。結合スレッドは、システム内のすべての LWP との関係の中でスケジューリングされます。

スレッドの優先順位は、LWP リソースへのアクセスを調整します。デフォルトでは、LWP はタイムシェアクラスです。計算量の多いマルチスレッドの場合、スレッドの優先順位はあまり役立ちません。MT ライブラリを使って多くの同期を行うマルチスレッドアプリケーションでは、スレッドの優先順位がより意味をもちます。

スケジューリングクラスは、システムコール `priocntl(2)` で設定します。最初の 2 つの引数で、この設定の適用範囲を呼び出し側の LWP に限定したり、1 つ以上のプロセスのすべての LWP にしたりすることが可能です。3 番目の引数はコマンドで、次のいずれか 1 つを指定できます。

- `PC_GETCID` − 指定したクラスの、クラス識別子とクラス属性を取得します。
- `PC_GETCLINFO` − 指定したクラスの、クラス名とクラス属性を取得します。
- `PC_GETPARMS` − プロセス、プロセスに関係する LWP、またはプロセスのグループについてのクラス識別子とクラス固有のスケジューリングパラメタを取得します。
- `PC_SETPARMS` − プロセス、プロセスの LWP、またはプロセスのグループについてのクラス識別子とクラス固有のスケジューリングパラメタを設定します。

`priocntl()` は結合スレッドにだけ使用します。非結合スレッドの優先順位を変更する場合は、`pthread_setprio(3T)` を使用してください。

タイムシェアスケジューリング

タイムシェアスケジューリングでは、このスケジューリングの LWP に処理リソースが公平に配分されます。カーネルのそれ以外の部分は、ユーザに対する応答時間に悪影響を与えないようにプロセッサを短時間ずつ使用します。

システムコール `priocntl(2)` は、1 つ以上のプロセスの `nice()` レベルを設定します。`priocntl()` による `nice()` レベルの変更は、そのプロセス内のタイムシェアクラスのすべての LWP に適用されます。`nice()` レベルの範囲は通常は 0〜+20 で、スーパーユーザ特権をもつプロセスの場合は -20〜+20 です。この値が小さいほど優先順位が高くなります。

タイムシェアクラスの LWP をディスパッチする優先順位は、LWP のその時点での CPU 使用率と `nice()` レベルに基づいて計算されます。タイムシェアスケジューラにとって、`nice()` レベルは、LWP 間の相対的な優先順位を表します。

LWP の `nice()` レベルが大きいほど、その LWP に配分される CPU 時間は少なくなりますが 0 になることはありません。多くの CPU 時間をすでに消費している

LWP は、CPU 時間をほとんど (あるいは、まったく) 消費していない LWP よりも優先順位が下げられます。

リアルタイムスケジューリング

リアルタイム (RT) クラスは、プロセス全体またはプロセス内の 1 つ以上の LWP に適用できます。ただし、スーパーユーザ特権が必要です。

タイムシェアクラスの nice(2) レベルとは異なり、リアルタイムクラスを指定された LWP には、個々の LWP 単位または複数の LWP 単位で優先順位を設定できます。priocntl(2) システムコールで、プロセス内のリアルタイムクラスのすべての LWP の属性を変更できます。

スケジューラは、最も高い優先順位を持つリアルタイムクラスの LWP をディスパッチします。優先順位の高い LWP が実行可能状態になると、それよりも優先順位の低い LWP は、実行リソースを横取りされます。実行リソースを横取りされた LWP は、そのレベルの待ち行列の先頭に置かれます。

リアルタイムクラスの LWP は、実行リソースが横取りされたり、一時停止したり、リアルタイム優先順位が変更されたりしない限り、プロセッサの制御を保持し続けます。リアルタイムクラスの LWP には、タイムシェアクラスのプロセスよりも絶対的に高い優先順位が与えられます。

新しく生成された LWP は、親プロセスまたは親 LWP のスケジューリングクラスを継承します。リアルタイムクラスの LWP は、親のタイムスライス (リソース割り当て時間) を有限または無限指定に関係なく継承します。

有限タイムスライスを指定された LWP は、処理が終了するか、入出力イベント待ちなどによってブロックされるか、より優先順位の高い実行可能なリアルタイムプロセスによって実行リソースを横取りされるか、またはタイムスライスが満了するまで実行を続けます。

無限タイムスライスを指定された LWP が実行を停止するのは、LWP が終了するか、ブロックされるか、または実行リソースが横取りされたときだけです。

LWP のスケジューリングとスレッドの結合

スレッドライブラリは、非結合スレッドを実行するための、実行リソース内の LWP 数を自動的に調整します。これには次の目的があります。

- ブロックされていない LWP がなくなったという理由だけで、プログラムがブロックされないようにする。

  たとえば、実行可能な非結合スレッドの数が LWP の数より多いとき、アクティブなすべてのスレッドがカーネル内で無期限の待ち状態 (たとえば端末からの入力待ち) になりブロックしていると、待ち状態のスレッドのどれかが返るまでプロセスの処理が進まなくなります。

- LWP を効率的に使用する。

  たとえば、スレッドごとに 1 つの LWP を生成したとすると、いつもアイドル状態であるような LWP の数が多くなり、使われていない LWP が要求するリソースによってオペレーティング環境の負荷が増大します。

タイムスライスが適用されるのは LWP であって、スレッドではありません。つまり、LWP が 1 つしか存在しなければ、プロセス内でタイムスライスは行われません。その LWP 上のスレッドは、スレッド間同期機構でブロックされるか、実行リソースを横取りされるか、または終了するまで実行を続けます。

スレッドに対する優先順位は、`pthread_setprio(3T)` で設定できます。優先順位の低い非結合スレッドは、それよりも優先順位の高い非結合スレッドが実行可能になっていないときだけ LWP に割り当てられます。ただし、結合スレッドは自分専用の LWP をもつので、LWP を争奪することはありません。
なお、`pthread_setprio()` で設定されるスレッド優先順位は、CPU に対してではなく LWP に対するスレッドのアクセスを調整します。

スケジューリングをきめ細かく制御する必要がある場合は、スレッドを LWP に結合します。多数の非結合スレッドが 1 つの LWP を争奪するような状況では、きめ細かい制御を実現できないからです。

特に、優先順位の低い非結合スレッドが優先順位の高い LWP 上にあり CPU 上で実行されていて、優先順位の低い LWP に割り当てられた優先順位の高い非結合スレッドが実行されていないことがあります。このようにスレッドの優先順位は、CPU へのアクセスについての 1 つのヒントにすぎません。

リアルタイムスレッドは、外部からの入力に対して迅速な応答が必要なときに使用します。たとえば、マウスの動きを追跡するスレッドは、マウスボタンのクリックにただちに反応しなければなりません。そのスレッドを LWP に結合すれば、必要なときにいつでも LWP を使用できるようになります。その LWP をリアルタイムスケジューリングクラスに割り当てれば、マウスボタンのクリックに迅速に反応するようにスケジューリングされます。

SIGWAITING — 待ち状態のスレッドのための LWP の生成

スレッドライブラリは通常、プログラムを実行するのに十分な数の LWP が実行リソース内に存在することを保証します。

プロセス内のすべての LWP が無期限の待ち状態でブロックされる (たとえば、端末またはネットワークからの読み取りがブロックされる) と、オペレーティング環境は SIGWAITING というシグナル (新たに導入されたシグナル) をプロセスに送ります。このシグナルはスレッドライブラリで処理されます。このとき、実行待ちのスレッドがプロセス内にあれば新しい LWP を生成し、適当な待ち状態のスレッドを選択して、新しい LWP に割り当てて実行します。

SIGWAITING 機構は、複数のスレッドが計算を目的としていて、もう 1 つ別のスレッドが実行可能になった場合に新しい LWP が生成されるかどうかを保証していません。計算を目的とするスレッドは、LWP の不足のために複数の実行可能なスレッドが動作するのを妨げることがあります。

thr_setconcurrency(3THR) を呼び出すことによって、これを防ぐことができます。POSIX スレッドで thr_setconcurrency() を使用すると POSIX 準拠ではなくなりますが、計算量の多い状況で非結合スレッド用の LWP が不足するのを回避するには、この使い方が望ましいでしょう。(POSIX に完全に準拠し LWP の不足も回避する唯一の方法は、PTHREAD_SCOPE_SYSTEM 結合スレッドのみを生成することです。)

thr_setconcurrency(3THR) 関数の使用方法の詳細は、276ページの「スレッドの並行度 (Solaris スレッドの場合のみ)」を参照してください。

Solaris スレッドでは、thr_create(3THR) 呼び出しで THR_NEW_LWP を使って、別の LWP を生成するという方法もあります。

LWP の存在時間

アクティブなスレッドが少なくなると、実行リソース内の一部の LWP は必要なくなります。LWP の数がアクティブなスレッドの数より多いとき、スレッドライブラリは不要な LWP を削除します。スレッドライブラリは LWP の存在時間を監視し、長い間 (現行では 5 分間) 使用されていない LWP は削除します。

シグナルの拡張

UNIX の従来のシグナルモデルが、スレッドに対しても自然な方法で使用できるように拡張されています。この拡張の主な特徴は、シグナルに対する処置がプロセス全体に適用され、シグナルマスクはスレッドごと適用されることです。プロセス全体に適用されるシグナル処置は、signal(2)、sigaction(2) などの従来の機構を使って設定します。

シグナルハンドラが SIG_DFL または SIG_IGN に対して設定されている場合、シグナル (終了、コアダンプ、停止、継続、無視) を受け取ると、対象となるプロセス全体に対して指示された動作を行います。つまり、プロセス内のすべてのスレッドが対象となります。これらのシグナルでハンドラをもたないものについては、どのスレッドがシグナルを拾うかという問題は重要ではありません。これは、シグナルの受信による処置はプロセス全体に行われるからです。シグナルについては、signal(5) のマニュアルページを参照してください。

各スレッドは、スレッド専用のシグナルマスクを持っています。これによって、スレッドが使用するメモリーまたはその他の状態をシグナルハンドラも使用する限りは、スレッドは特定のシグナルをブロックできます。同じプロセス内のすべてのスレッドは、sigaction(2) またはそれに相当する機能によって設定されるシグナルハンドラを共有します。

あるプロセス内のスレッドが、別のプロセス内の特定のスレッドにシグナルを送ることはできません。kill(2) または sigsend(2) によるシグナルはプロセスに送られ、そのプロセス内のシグナルを受け取ることができるスレッドのどれか 1 つによって処理されます。

非結合スレッドは、代替シグナルスタックを使用できません。結合スレッドは、その状態が実行リソースと関連付けられているので、代替シグナルスタックを使用できます。代替シグナルスタックを使用するには、sigaction(2) を使ってシグナルを受け取れる状態にして、次に sigaltstack(2) で代替シグナルスタックを宣言して、使用可能な状態にします。

アプリケーションは、プロセス固有のシグナルハンドラを元にして、スレッド固有のシグナルハンドラを使用できます。プロセス全体のシグナルハンドラが、シグナルを処理するスレッドの識別子を、スレッド固有のシグナルハンドラのテーブルへのインデックスとして使用する方法があります。識別子 0 のスレッドは存在しません。

シグナルは、トラップや例外条件の同期シグナルと、割り込みの非同期シグナルの 2 つに大別されます。

従来の UNIX と同様、シグナルが保留状態のときに同じシグナルが再度発生しても無視されます。保留状態のシグナルは、カウンタではなく 1 ビットで表現されるからです。つまり、シグナルの転送はべき等です。

シングルスレッドのプロセスのときと同様、スレッドがシステムコールを呼び出してブロックされている間にシグナルを受け取ると、そのシステムコールは EINTR エラーを返すか、あるいはそれが入出力のシステムコールの場合には要求したバイト数が全部転送されないで戻ります。

マルチスレッドプログラムでは、特に pthread_cond_wait(3THR) に対するシグナルの影響に注意する必要があります。この関数は、通常 pthread_cond_signal(3THR) か pthread_cond_broadcast(3THR) に応答して戻ります。しかし、この関数で待ち状態になっているスレッドが従来の UNIX のシグナルを受け取ると、この関数は EINTR エラーで戻ります。詳細は、189ページの「条件変数で待っているときの割り込み (Solaris スレッドのみ)」を参照してください。

同期シグナル

トラップ (SIGILL、SIGFPE、SIGSEGV など) は、ゼロ除算を行なったり、自分に明示的にシグナルを送ったりすることによって、スレッド自体が発生させるものです。トラップは、そのトラップを発生させたスレッドだけが処理します。プロセス内の複数のスレッドが、同じ種類のトラップを同時に発生させて処理することもできます。

同期シグナルの場合には、シグナルの概念を容易に個々のスレッドに適用するように拡張できます。シグナルを処理するのが、そのシグナルを発生させたスレッド自体だからです。

ただし、そのスレッドがシグナルを処理するように準備されていない (たとえば、sigaction(2) でシグナルハンドラを設定していない) 場合は、同期シグナルを受け取るスレッドについてハンドラが起動されます。

同期シグナルは通常、スレッドだけでなく、プロセス全体に悪影響を及ぼすような重大な事態を意味するので、プロセスを終了させた方がよい場合が多くあります。

非同期シグナル

割り込み (SIGINT、SIGIO など) は、あらゆるスレッドに対して、プロセス外部のなんらかの動作が原因で非同期的に発生します。非同期シグナルは、他のスレッド

から明示的に送られてきたシグナルの場合も、ユーザが Control-C キーを入力したなどの外部動作を表す場合もあります。非同期シグナルの処理は、同期シグナルの場合に比べると複雑です。

割り込みは、その割り込みを受け取るようにシグナルマスクが設定されている、どのスレッドでも処理できます。複数のスレッドが、割り込みを受け取ることができるように設定されている場合は、その中の 1 つのスレッドだけが選択されます。

複数の同じシグナルがプロセスに送られた場合、スレッドがそのシグナルをマスクしていなければ、それぞれのシグナルを別のスレッドで処理できます。また、すべてのスレッドがマスクしているときは、「保留」の印が付けられ、最初にマスク解除したスレッドによって処理されます。

継続セマンティクス法

継続セマンティクス法は、従来から行われてきたシグナル処理方法です。これは、シグナルハンドラから復帰したときに割り込みが発生した時点から実行を再開する方法です。この方法は、シングルスレッドのプロセスで非同期シグナルを扱うのに適しています (詳細は、例 5–1 を参照してください)。

また、PL/1 などの一部のプログラミング言語の例外処理機構でも使用されています。

例 5–1 継続セマンティクス法

```
unsigned int nestcount;

unsigned int A(int i, int j) {
    nestcount++;

    if (i==0)
        return(j+1)
    else if (j==0)
        return(A(i-1, 1));
    else
        return(A(i-1, A(i, j-1)));
}

void sig(int i) {
    printf("nestcount = %d\n", nestcount);
}

main() {
    sigset(SIGINT, sig);
    A(4,4);
```

(続く)

```
}
```

シグナルに関する操作

pthread_sigmask(3THR)

pthread_sigmask(3THR) は、スレッドのシグナルマスクを設定するための関数です。つまり、sigprocmask(2) システムコールがプロセスに対して行うのと同じ操作をスレッドに対して行います。新しいスレッドが生成されると、その初期状態のシグナルマスクは生成元から継承されます。

マルチスレッドプロセス内で sigprocmask() を呼び出すのは、pthread_sigsetmask() を呼び出すのと同等です。詳細は、sigprocmask(2) のマニュアルページを参照してください。

pthread_kill(3THR)

pthread_kill(3THR) は、特定のスレッドにシグナルを送るための関数で、スレッド用の kill(2) と考えることができます。これは、プロセスにシグナルを送るものではありません。プロセスに送られたシグナルは、プロセス内のどのスレッドでも処理できます。pthread_kill() で送られたシグナルは、指定されたスレッドだけが処理できます。

pthread_kill() でシグナルを送ることができるのは、現在のプロセス内のスレッドに限られることに注意してください。スレッド識別子 (thread_t 型) の有効範囲が局所的であるため、現在のプロセス以外のプロセス内のスレッドを指定できないからです。

宛先のスレッドでシグナルの受信時に行われる処置 (ハンドラ、SIG_DFL、SIG_IGN) は通常どおり広域的です。この意味は、たとえば、あるスレッドに SIG*XXX* を送信する場合、そのプロセスにとっての SIG*XXX* シグナル処置がそのプロセスを終了させることであれば、宛先スレッドがこのシグナルを受け取ったとき、そのプロセス全体が終了するということです。

sigwait(2)

マルチスレッドプログラムでは、sigwait(2) が好まれるインタフェースです。これは、非同期的に生成されるシグナルを非常にうまく扱えるからです。

sigwait() は、*set* 引数に指定したシグナルが呼び出しスレッドに送られてくるまで、そのスレッドを待ち状態にします。スレッドが待っている間は、*set* 引数で指定したシグナルのマスクが解除され、復帰時に元のシグナルマスクが設定し直されます。

set 引数で識別されるすべてのシグナルは、呼び出しスレッドを含むすべてのスレッドでブロックする必要があります。そうしないと、sigwait() は正確に動作しません。

非同期シグナルからプロセス内のスレッドを隔離したい場合は、sigwait() を使用します。非同期シグナルを待つスレッドを 1 つ生成しておき、他のスレッドは、現在のプロセスに送られてくる可能性のある非同期シグナルをすべてブロックするように生成します。

新しい sigwait の実装

Solaris オペレーティング環境 2.5 以降のリリースでは、Solaris オペレーティング環境 2.5 バージョンの新しい sigwait() と POSIX.1c バージョンの sigwait() の 2 種類を使用できます。新しいアプリケーションとライブラリでは、できるだけ POSIX 規格インタフェースを使用してください。Solaris オペレーティング環境バージョンは、将来のリリースではサポートされない可能性があるからです。

注 - Solaris オペレーティング環境 2.5 バージョンの新しい sigwait() は、シグナルの無視という処置を無効にしません。以前の sigwait(2) の動作に依存しているアプリケーションは、ダミーのシグナルハンドラをインストールして、その処置を SIG_IGN からハンドラをもつように変更しない限りブレークする可能性があるため、このシグナルに対する sigwait() 呼び出しでこのシグナルは捕捉されます。

これら 2 つのバージョンの sigwait() の構文は下記のとおりです。

```
#include <signal.h>

/* Solaris 2.5 バージョン */
int sigwait(sigset_t *set);
```

(続く)

```
/* POSIX.1c バージョン */
int sigwait(const sigset_t *set, int *sig);
```

指定のシグナルが送られてくると、POSIX.1c `sigwait()` は保留されているそのシグナルを削除し、*sig* にそのシグナルの番号を入れます。同時に複数のスレッドから `sigwait()` を呼び出すこともできますが、受け取るシグナルごとに 1 つのスレッドだけの `sigwait()` だけが返ってきます。

`sigwait()` を使うと、非同期シグナルを同期的に扱うことができます。つまり、非同期シグナルを扱うスレッドから `sigwait()` だけを呼び出すと、シグナルが到着しだい戻ります。`sigwait()` の呼び出し側も含むすべてのスレッドで非同期シグナルをマスクすることによって、非同期シグナルを特定のシグナルハンドラだけに処理させることができます。非同期シグナルを安全に処理することが可能です。

すべてのスレッドですべてのシグナルを常にマスクし、必要なときだけ `sigwait()` を呼び出すようにすれば、アプリケーションはシグナルに依存するスレッドに対してはるかに安全になります。

通常、`sigwait()` を使うときは、シグナルを待つスレッドを 1 つまたは複数生成します。`sigwait()` はマスクされているシグナルであっても受け取るため、それ以外のスレッドでは誤ってシグナルを受け取ることがないように、対象となるシグナルをすべてブロックしてください。

シグナルが送られてくると、スレッドは `sigwait()` から復帰し、シグナルを処理し、さらに次のシグナルを待ちます。このシグナル処理スレッドでは、非同期保護関数以外の関数も使用でき、他のスレッドとも通常の方法で同期をとることができます (非同期保護カテゴリについては、199ページの「マルチスレッドインタフェースの安全レベル」を参照してください)。

注 - 同期シグナルに対しては、`sigwait()` を決して使わないでください。

---

sigtimedwait(2)

`sigtimedwait(2)` は、指定時間が経過してもシグナルが送られてこなかったときにエラーで復帰する点を除いて、`sigwait(2)` と似ています。

スレッド指定シグナル

UNIX のシグナル機構が、スレッド指定という考え方で拡張されています。これは、シグナルがプロセスではなく特定のスレッドに送られるという点を除いて、通常の非同期シグナルと似ています。

独立したスレッドで非同期シグナルを待つ方が、シグナルハンドラを実装して、そこでシグナルを処理するよりも安全で簡単です。

非同期シグナルを処理するよりよい方法は、非同期シグナルを同期的に処理することです。具体的には、183ページの「sigwait(2)」で説明した `sigwait(2)` を呼び出すことにより、スレッドはシグナルの発生を待つことができます。

例 5–2 非同期シグナルと `sigwait(2)`

```
main() {
    sigset_t set;
    void runA(void);
    int sig;

    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    pthread_sigsetmask(SIG_BLOCK, &set, NULL);
    pthread_create(NULL, 0, runA, NULL, PTHREAD_DETACHED, NULL);

    while (1) {
        sigwait(&set, &sig);
        printf("nestcount = %d\n", nestcount);
        printf("received signal %d\n", sig);
    }
}

void runA() {
    A(4,4);
    exit(0);
}
```

この例は、例 5–1 を修正したものです。メインルーチンは `SIGINT` シグナルをマスクし、関数 A を呼び出す子スレッドを生成し、最後に `sigwait()` を呼び出して `SIGINT` シグナルを待ちます。

対象となるシグナルが、計算を行うためのスレッドでマスクされていることに注目してください。計算を行うためのスレッドは、メインスレッドのシグナルマスクを継承するからです。メインスレッドは `SIGINT` から保護されており、`sigwait()` の内部でだけ `SIGINT` に対するマスクが解除されます。

また、sigwait() を使用しているとき、システムコールから割り込まれる危険性がないことも注目してください。

完了セマンティクス法

シグナルを処理するもう 1 つの方法に、完了セマンティクス法があります。

完了セマンティクス法を使用するのは、シグナルが重大な障害が発生したことを示しているために現在のコード部を継続実行しても意味がない場合です。問題の原因となったコード部を引き続き実行する代わりに、シグナルハンドラが実行されます。つまり、シグナルハンドラによって、当該コード部の処理が完了されます。

例 5–3 で、if 文の then 部分の本体が問題のコード部です。setjmp(3C) の呼び出しは、プログラムの現在のレジスタ状態を *jbuf* に退避して 0 で復帰します。そして、このコード部が実行されます。

例 5–3 完了セマンティクス法

```
sigjmp_buf jbuf;
void mult_divide(void) {
    int a, b, c, d;
    void problem();

    sigset(SIGFPE, problem);
    while (1) {
        if (sigsetjmp(&jbuf) == 0) {
            printf("Three numbers, please:\n");
            scanf("%d %d %d", &a, &b, &c);
            d = a*b/c;
            printf("%d*%d/%d = %d\n", a, b, c, d);
        }
    }
}

void problem(int sig) {
    printf("Couldn't deal with them, try again\n");
    siglongjmp(&jbuf, 1);
}
```

SIGFPE (浮動小数点例外条件) が発生すると、シグナルハンドラが呼び出されます。

シグナルハンドラは、siglongjmp(3C) を呼び出します。この関数は、*jbuf* に退避されていたレジスタ状態を復元し、プログラムを sigsetjmp() 部分から復帰させます (プログラムカウンタとスタックポインタも退避されています)。

このとき、sigsetjmp(3C) は siglongjmp() の第 2 引数である 1 を返します。その結果、問題のコード部はスキップされ、while ループの次の繰り返しに入ります。

sigsetjmp(3C) と siglongjmp(3C) をマルチスレッドプログラムで使うこともできますが、別のスレッドで呼び出された sigsetjmp() の結果を使って、siglongjmp() を呼び出すことはできません。

また、sigsetjmp() と siglongjmp() は、シグナルマスクを退避または復元しますが、setjmp(3C) と longjmp(3C) は、シグナルマスクを退避または復元しません。

シグナルハンドラでは、sigsetjmp() と siglongjmp() を使用してください。

完了セマンティクス法は、例外条件の処理でよく使用されます。特に Sun Ada™ プログラミング言語では、このモデルが使用されています。

注 - 同期シグナルに対して sigwait(2) を決して使用しないでください。

シグナルハンドラと「非同期シグナル安全」

スレッドに対する安全性と似た概念に、「非同期シグナル安全」があります。「非同期シグナル安全」操作は、割り込まれている操作を妨げないことが保証されています。

「非同期シグナル安全」に関する問題が生じるのは、現在の操作がシグナルハンドラによる割り込みで動作を妨げる可能性があるときです。

たとえば、プログラムが printf(3S) を呼び出している最中にシグナルが発生し、そのシグナルを処理するハンドラ自体も printf() を呼び出すとします。その場合は、2 つの printf() 文の出力が混ざり合ってしまいます。これを避けるには、printf() がシグナルに割り込まれたときにシグナルハンドラが printf() を呼び出さないようにします。

この問題は、同期プリミティブでは解決できません。シグナルハンドラと同期対象操作の間で同期をとろうとすると、たちまちデッドロックが発生するからです。

たとえば、printf() が自分自身を相互排他ロックで保護していると仮定します。あるスレッドが printf() を呼び出している最中に、つまり相互排他ロックを保持した状態にある時に、シグナルにより割り込まれたとします。

(printf() 内部にいるスレッドから呼び出されている) ハンドラ自身が printf() を呼び出すと、すでに相互排他ロックを保持しているスレッドが、もう一度相互排他ロックを獲得しようとします。その結果、即座にデッドロックとなります。

ハンドラと操作の干渉を回避するには、そうした状況が決して発生しないようにするか (通常は危険領域でシグナルをマスクする)、シグナルハンドラ内部では「非同期シグナル安全」操作以外は使用しないようにします。

スレッドのマスクを設定することは、負荷の小さなユーザレベルの操作であるため、関数やプログラムの一部分を「非同期シグナル安全」のために修正しても負荷は大きくなりません。

POSIX が「非同期シグナル安全」を保証しているルーチンだけを表 5–2 に示します。どのようなシグナルハンドラも、これらの関数を安全に呼び出すことができます。

表 5–2 「非同期シグナル安全」関数

_exit()	fstat()	read()	sysconf()
access()	getegid()	rename()	tcdrain()
alarm()	geteuid()	rmdir()	tcflow()
cfgetispeed()	getgid()	setgid()	tcflush()
cfgetospeed()	getgroups()	setpgid()	tcgetattr()
cfsetispeed()	getpgrp()	setsid()	tcgetpgrp()
cfsetospeed()	getpid()	setuid()	tcsendbreak()
chdir()	getppid()	sigaction()	tcsetattr()
chmod()	getuid()	sigaddset()	tcsetpgrp()
chown()	kill()	sigdelset()	time()
close()	link()	sigemptyset()	times()

表 5–2 「非同期シグナル安全」関数 続く

creat()	lseek()	sigfillset()	umask()
dup2()	mkdir()	sigismember()	uname()
dup()	mkfifo()	sigpending()	unlink()
execle()	open()	sigprocmask()	utime()
execve()	pathconf()	sigsuspend()	wait()
fcntl()	pause()	sleep()	waitpid()
fork()	pipe()	stat()	write()

条件変数で待っているときの割り込み (Solaris スレッドのみ)

スレッドが条件変数で待っている最中にシグナルが送られてきた場合の動作は、従来の規約では、割り込まれたシステムコールが EINTR エラーで戻るというものでした (ただし、プロセスは終了しないと仮定します)。

新たに注意すべき点は、cond_wait(3T) または cond_timedwait(3T) が復帰した時点で、mutex はロックし直されていることです。

Solaris では、スレッドが cond_wait() または cond_timedwait() でブロックされているとき、マスクされていないシグナルがスレッドに送られてくるとシグナルハンドラが呼び出され、その後 cond_wait() または cond_timedwait() は mutex をロックした状態で戻ります。

これは mutex がシグナルハンドラ内でロックされていることを意味します。シグナルハンドラは、スレッドの後処理をする必要があるかもしれないからです。このことは、Solaris オペレーティング環境 2.5 リリースでは成り立ちますが将来は変更される可能性があるので、この動作に依存しないでください。

注 - POSIX スレッドでは、pthread_cond_wait(3T) はシグナルから復帰しますが、これはエラーではありません。pthread_cond_wait() は、ブロックが仮に解除されたことを示すため 0 を戻します。

例 5–4 で説明します。

例 5–4 条件変数で待っているときの割り込み

```
int sig_catcher() {
    sigset_t set;
    void hdlr();

    mutex_lock(&mut);

    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    sigsetmask(SIG_UNBLOCK, &set, 0);

    if (cond_wait(&cond, &mut) == EINTR) {
        /* シグナルが発生、ロックは保持されている */
        cleanup();
        mutex_unlock(&mut);
        return(0);
    }
    normal_processing();
    mutex_unlock(&mut);
    return(1);
}

void hdlr() {
    /* シグナルハンドラ内でロックは保持される */
    ...
}
```

sig_catcher() が呼び出された時点では、すべてのスレッドで SIGINT シグナルがブロックされているものとします。さらに、sigaction(2) によって hdlr() が SIGINT シグナルのシグナルハンドラとして設定されているものとします。SIGINT シグナルをマスク解除して、cond_wait() で待ち状態になっているスレッドに SIGINT シグナルが送られてくると、スレッドは最初に mutex をロックし直し、その後 hdlr()を呼び出して、cond_wait() から EINTR エラーで戻ります。

sigaction() で SA_RESTART フラグを指定したとしても、ここでは意味がないことに注意してください。cond_wait(3T) はシステムコールではないため、自動的に再呼び出しされないからです。cond_wa でスレッドがブロックされているときにシグナルが送られてくると、cond_wait() は常に EINTR エラーで戻ります。
なお、アプリケーションでは、割り込まれた cond_wait() が相互排他ロックを再

獲得することに依存しないでください。この動作は、将来変更される可能性があります。

入出力の問題

マルチスレッドプログラミングの利点の 1 つは、入出力の性能を高めることができることです。従来の UNIX の API では、この点に関してほとんどサポートされていませんでした。つまり、ファイルシステムに用意されている機構を利用するか、ファイルシステムを完全にバイパスするかのどちらかの選択肢しかありませんでした。

この節では、入出力の並行化やマルチバッファによって入出力の柔軟性を向上させるためのスレッドの使用方法を説明します。また、同期入出力 (スレッドを使用) と非同期入出力 (スレッドを使用することも使用しないこともある) の相違点と類似点についても説明します。

遠隔手続き呼び出しとしての入出力

従来の UNIX のモデルでは、入出力は同期的に行われるように見え、入出力装置に対して、あたかも遠隔手続き呼び出しを行なっているように見えました。入出力の呼び出しが復帰した時点では、入出力は完了しているか、少なくとも完了しているように見えます (たとえば、書き込み要求はオペレーティング環境内のバッファにデータを転送しただけで戻ることがあります)。

このモデルの利点は、プログラマは手続き呼び出しの考え方に馴れているので、簡単に理解できることです。

従来の UNIX システムにはなかった代替モデルに非同期モデルがあります。このモデルでは、入出力要求は操作を開始させるだけで、プログラム側がなんらかの方法で操作の完了を検出しなければなりません。

この方法は同期モデルほど簡単ではありませんが、従来のシングルスレッドの UNIX プロセスでも並行入出力などの処理が可能であるという利点があります。

非同期性の管理

非同期入出力のほとんどの機能は、マルチスレッドプログラムによる同期入出力で実現できます。具体的には、要求を出した後でその要求の完了をチェックするという非同期入出力の操作を行う代わりに、独立したスレッドで同期入出力を実行します。メインスレッド側は、pthread_join(3T) などによって入出力操作の完了を確認します。

非同期入出力

各スレッドの同期入出力で同じ効果を実現できるため、非同期入出力が必要になることはほとんどありません。ただし、スレッドで実現できない非同期入出力機能もあります。

簡単な例は、ストリームとしてテープドライブへ書き込みを行う場合です。この場合、テープに書き込まれている間はテープドライブを停止させないようにし、テープに書き込むデータをストリームとして送っている間は高速にテープを先送りします。

これを行うためにカーネル内のテープドライバは、以前のテープへの書き込み操作が完了したことを知らせる割り込みに応答する時に、待ち行列に入っている書き込み要求を発行する必要があります。

スレッドでは、書き込み順序を保証できません。スレッドの実行される順序が不定だからです。たとえば、テープに対して順番どおり書き込みを行おうとしても不可能です。

非同期入出力操作

```
#include <sys/asynch.h>

int aioread(int fildes, char *bufp, int bufs, off_t offset,

     int whence, aio_result_t *resultp);

int aiowrite(int filedes, const char *bufp, int bufs,
    off_t offset, int whence, aio_result_t *resultp);

aio_result_t *aiowait(const struct timeval *timeout);


int aiocancel(aio_result_t *resultp);
```

aioread(3) と aiowrite(3) の形式は、pread(2) と pwrite(2) の形式にそれぞれ似ています。違いは、引数リストの最後に引数が 1 つ追加されていることです。aioread() または aiowrite() を呼び出すと、入出力操作が開始されます(あるいは、入出力要求が待ち行列に入れられます)。

この呼び出しはブロックされずに復帰し、*resultp* の指す構造体に終了状態が戻されます。これは aio_result_t 型の項目で、次のフィールドで構成されています。

```
int aio_return;
int aio_errno;
```

呼び出しが失敗すると、aio_errno にエラーコードが設定されます。そうでない場合は、このフィールドには操作要求が正常に待ち行列に入れられたことを示す AIO_INPROGRESS が設定されます。

非同期入出力操作の完了は、aiowait(3) で待つことができます。この関数は、最初の aioread(3)、または aiowrite(3) で指定した aio_result_t 構造体へのポインタを返します。

この時点で aio_result_t には、read(2) または write(2) のどちらかが非同期バージョン以外で呼ばれた時と同じ情報が設定されます。この read または write が正常終了した場合、aio_return には読み書きされたバイト数が設定されます。異常終了した場合、aio_return には -1、aio_errno にはエラーコードが設定されます。

aiowait() には *timeout* 引数があり、呼び出し側の待ち時間を設定できます。ここに NULL ポインタを指定すれば、無期限に待つという意味になります。また、値 0 が設定されている構造体を指すポインタの場合は、まったく待たないという意味になります。

非同期入出力操作を開始し別の処理を行なって aiowait() で操作の完了を待つ、あるいは操作完了時に非同期的に送られてくる SIGIO を利用するという方法もあります。

保留状態の入出力操作を取り消すときは、aiocancel() を使用します。このルーチンを呼び出すときは、取り消そうとする非同期入出力操作の結果を格納するアドレスを引数で指定します。

共有入出力と新しい入出力システムコール

複数のスレッドが同じファイル記述子を使って同時に入出力操作を行う場合、従来の UNIX の入出力インタフェースがスレッドに対して安全ではない場合があります。この問題が生じるのは、入出力が逐次的に行われない場合です。システムコール `lseek(2)` でファイルオフセットを設定し、そのオフセットで次の `read(2)` または `write(2)` を呼び出してファイル内の操作開始位置を指定する場合です。このとき、同じファイル記述子に対して複数のスレッドが `lseeks()` を実行してしまうと矛盾が生じます。

この矛盾は、新しいシステムコール `pread(2)` と `pwrite(2)` で回避できます。

```
#include <sys/types.h>
#include <unistd.h>

ssize_t pread(int fildes, void *buf, size_t nbyte, off_t offset);

ssize_t pwrite(int filedes, void *buf, size_t nbyte,
    off_t offset);
```

これらのシステムコールの動作は、ファイルオフセットを指定するための引数が追加されていることを除いて、`read(2)` と `write(2)` とそれぞれ同じです。`lseek(2)` の代わりに、この引数でオフセットを指定すれば、複数のスレッドから同じファイル記述子に対して安全に入出力操作を実行できます。

getc(3S) と putc(3S) の代替

標準入出力に関して、もう 1 つ問題があります。`getc(3S)` や `putc(3S)` などは、マクロとして実装されているため非常に高速に動作するという理由でよく使用されています。プログラムのループ内で使うときも、効率を気にする必要がないからです。

しかし、これらは、スレッドに対して安全になるよう変更されたため、以前よりも負荷が大きくなっています。変更後、(少なくとも) 2 つの内部サブルーチンが、相互排他のロックと解除のために呼び出されています。

この問題を回避するために、これらの代替マクロとして `getc_unlocked(3S)` と `putc_unlocked(3S)` が提供されています。

これらの代替マクロは mutex をロックしないので、スレッドに対して安全ではないオリジナルの `getc(3S)` と `putc(3S)` と同程度に高速です。

しかし、それらをスレッドに対して安全な方法で使うためには、標準入出力ストリームを保護する mutex を `flockfile(3S)` と `funlockfile(3S)` で明示的にロックまたは解除しなければなりません。ループの外側を `flockfile()` と `funlockfile()` で囲み、ループの内側で `getc_unlocked()` と `putc_unlocked()` を呼び出します。

第 6 章

安全なインタフェースと安全ではないインタフェース

この章では、関数とライブラリについて、マルチスレッドに対する安全レベルを定義します。

「スレッド安全」

「スレッド安全」とは、データアクセスの競合 (つまり、複数のスレッドがデータをアクセスして変更するときに、その順番によってデータの値が正しくなったり正しくなくなったりする状況) を回避することです。

スレッド間でデータを共有する必要がない場合は、スレッドごとに専用のコピーを与えますが、共有する必要がある場合には、明示的に同期をとることによってプログラムが確定的な動きをするように制御する必要があります。

手続きが「スレッド安全」とは、その手続きが複数のスレッドによって同時に実行されても論理的な正しさが失われないことです。実際は、安全性は次の 3 段階で区別されます。

- 「スレッド安全ではない」

- 「スレッド安全」− 直列化
- 「スレッド安全」− MT-安全

「スレッド安全ではない」手続きであっても、mutex をロックする命令と解除する命令で囲めば、その処理は直列化され「スレッド安全」になります。例 6–1 は `fputs()` を簡略化したもので、最初のルーチンは「スレッド安全ではない」例です。

2 番目のルーチンは直列化した例です。ここでは、1 つの mutex で手続きを並行実行させないようにしています。これは通常必要とされる同期よりも強い同期となります。2 つのスレッドが `fputs()` を使って異なるファイルに出力するときは、一方がもう一方を待たせる必要はありません。両者の間で同期をとる必要があるのは、同じ出力ファイルを共有しているときだけです。

最後のルーチンは、「MT-安全」の例です。ここではファイルごとに mutex をロックしているので、2 つのスレッドが異なるファイルに同時に出力できます。つまり、ルーチンが「MT-安全」であるとは、「スレッド安全」で、しかもそのルーチンの実行が性能に悪影響を及ぼさないことを意味します。

例 6–1 「スレッド安全」の段階

```
/* スレッド安全ではない */
fputs(const char *s, FILE *stream) {
    char *p;
    for (p=s; *p; p++)
        putc((int)*p, stream);
}

/* 直列化 */
fputs(const char *s, FILE *stream) {
    static mutex_t mut;
    char *p;
    mutex_lock(&m);
    for (p=s; *p; p++)
        putc((int)*p, stream);

    mutex_unlock(&m);
}

/* MT-安全 */
mutex_t m[NFILE];
fputs(const char *s, FILE *stream) {
    static mutex_t mut;
    char *p;
    mutex_lock(&m[fileno(stream)]);
    for (p=s; *p; p++)
        putc((int)*p, stream);
```

(続く)

```
    mutex_unlock(&m[fileno(stream)]0;
}
```

マルチスレッドインタフェースの安全レベル

『*man pages section 3: Threads and Realtime Library Functions*』の スレッドについて、インタフェースのスレッドサポートの安全レベルのカテゴリを表 6–1 にリストしています。(これらのカテゴリの詳細は、`Intro(3)` のマニュアルページを参照してください)

表 6–1 インタフェースの安全レベル

カテゴリ	説明
Safe「安全」	このコードをマルチスレッドアプリケーションから呼び出しても安全
Safe with exceptions「例外付きで安全」	例外の内容については、マニュアルページの「注意事項 (NOTES)」の節を参照
Unsafe「安全ではない」	このインタフェースをマルチスレッドアプリケーションで使用するのは危険。ただし、複数のスレッドが、ライブラリ内で同時に実行されないようにアプリケーション側が対応すれば使用できる
MT-Safe「MT-安全」	このインタフェースは、マルチスレッドアクセスに完全に対応している。つまり、安全であると同時に並行性もサポートしている
MT-Safe with exceptions「例外付きで MT-安全」	例外については、『*man pages section 3*』の「注意事項 (NOTES)」の節を参照

表 6–1 インタフェースの安全レベル 続く

カテゴリ	説明
Async-Signal-Safe「非同期シグナル安全」	このルーチンをシグナルハンドラから安全に呼び出すことができる。「非同期シグナル安全」ルーチンは、シグナルが割り込んでも自己デッドロックにならない
Fork1-Safe「fork1-安全」	このときインタフェースは、Solaris の `fork1(2)` または POSIX の `fork(2)` が呼び出されたときに、保持していたロックを解放する

『*man pages section 3*』からのインタフェースの安全レベルについては付録 C を参照してください。該当するマニュアルページを参照してレベルを確認してください。

次の理由により安全化されていない関数もあります。

- その関数を「MT-安全」にすると、シングルスレッドアプリケーションの性能に悪影響を及ぼす。
- その関数が、安全ではないインタフェースを持っている。たとえば、スタックに確保したバッファへのポインタを戻すような関数です。こうした関数には、リエントラント (再入可能) な代替関数が用意されている場合があります。オリジナルの関数名の末尾に「`_r`」が付いているのがリエントラントな関数です。

注意 - 関数名の末尾に「`_r`」が付いていない関数がマルチスレッドに対して安全かどうかは、マニュアルページを参照してください。「MT-安全」ではないことが明記されている関数は、同期機構で保護するか、初期スレッド以外では使用しないでください。

「安全ではない」インタフェースのためのリエントラント関数

危険なインタフェースをもつ多くの関数には、「MT-安全」な代替関数が用意されています。これらの関数は、オリジナルの関数名の末尾に「`_r`」を付けることで区別されます。Solaris 環境に用意されている「`_r`」ルーチンを表 6–2 に示します。

表 6–2　リエントラント関数

asctime_r(3c)	gethostbyname_r(3n)	getservbyname_r(3n)
ctermid_r(3s)	gethostent_r(3n)	getservbyport_r(3n)
ctime_r(3c)	getlogin_r(3c)	getservent_r(3n)
fgetgrent_r(3c)	getnetbyaddr_r(3n)	getspent_r(3c)
fgetpwent_r(3c)	getnetbyname_r(3n)	getspnam_r(3c)
fgetspent_r(3c)	getnetent_r(3n)	gmtime_r(3c)
gamma_r(3m)	getnetgrent_r(3n)	lgamma_r(3m)
getauclassent_r(3)	getprotobyname_r(3n)	localtime_r(3c)
getauclassnam_r(3)	getprotobynumber_r(3n)	nis_sperror_r(3n)
getauevent_r(3)	getprotoent_r(3n)	rand_r(3c)
getauevnam_r(3)	getpwent_r(3c)	readdir_r(3c)
getauevnum_r(3)	getpwnam_r(3c)	strtok_r(3c)
getgrent_r(3c)	getpwuid_r(3c)	tmpnam_r(3s)
getgrgid_r(3c)	getrpcbyname_r(3n)	ttyname_r(3c)
getgrnam_r(3c)	getrpcbynumber_r(3n)	
gethostbyaddr_r(3n)	getrpcent_r(3n)	

「非同期シグナル安全」関数

「非同期シグナル安全」関数とは、シグナルハンドラから安全に呼び出すことができる関数のことです。それらは、POSIX 規格「IEEE Std 1003.1-1990, 3.3.1.3 (3)(f)」の 55 ページで定義されています。POSIX 規格の「非同期シグナル安全」関数に加え、スレッドライブラリの次の 3 つの関数も「非同期シグナル安全」関数です。

- `sema_post(3T)`
- `thr_sigsetmask(3T)`。`pthread_sigmask(3T)` と類似
- `thr_kill(3T)`。`pthread_kill(3T)` と類似

ライブラリの「MT-安全」レベル

マルチスレッドプログラムから呼び出される可能性のあるルーチンは、どれも「MT-安全」であるべきです。

つまり、同時に呼び出される可能性のあるルーチンは、並行実行されても正しく実行されることが必要です。このため、マルチスレッドプログラムで使用するすべてのライブラリインタフェースは、「MT-安全」でなければなりません。

現状では、すべてのライブラリが「MT-安全」ではありません。代表的な「MT-安全」ライブラリを表 6–3 に示します。その他のライブラリも、最終的には「MT-安全」なものに修正されます。

表 6–3 「MT-安全」なライブラリの例

ライブラリ	備考
`lib/libc`	安全ではないインタフェースには、「`*_r`」(セマンティクスはしばしば異なる) 形式の「スレッド安全」なインタフェースがある
`lib/libdl_stubs`	静的スイッチのコンパイルをサポート
`lib/libintl`	国際化ライブラリ
`lib/libm`	System V Interface Definition、Edition 3、X/Open、および ANSI C に準拠した算術ライブラリ

表 6–3 「MT-安全」なライブラリの例 続く

ライブラリ	備考
lib/libmalloc	空間を効率的に使用したメモリーの割り当てライブラリ。詳細は、malloc(3X) のマニュアルページを参照
lib/libmapmalloc	mmap(2) ベースの代替メモリー割り当てライブラリ。詳細は、mapmalloc(3X) のマニュアルページを参照
lib/libnsl	TLI インタフェース、XDR、RPC クライアントとサーバ、netdir、netselect、getXXbyYY インタフェースは安全ではない。ただし、getXXbyYY_r 形式のインタフェースは「スレッド安全 」
lib/libresolv	スレッド固有の errno をサポート
lib/libsocket	ネットワーク接続用のソケットライブラリ
lib/libw	複数バイトロケールをサポートするためのワイド文字とワイド文字列の関数
lib/straddr	ネットワーク名前アドレス変換ライブラリ
lib/libX11	X11 ウィンドウライブラリルーチン
lib/libC	C++ 実行時共有オブジェクト

「スレッド安全ではない」ライブラリ

「MT-安全」であることが保証されていないライブラリのルーチンを、マルチスレッドプログラムから安全に呼び出すためには、それらの呼び出しがシングルスレッドで行われるようにしなければなりません。

第 7 章

コンパイルとデバッグ

この章では、マルチスレッドプログラムのコンパイルとデバッグについて説明します。

マルチスレッドアプリケーションのコンパイル

ヘッダファイル、定義フラグ、リンクなどについては、オプションが多数あります。

コンパイルの準備

マルチスレッドプログラムのコンパイルとリンクには、次のものが必要です。C コンパイラ以外は、Solaris オペレーティング環境に付属しています。

- 標準 C コンパイラ
- インクルードファイル
  - `<thread.h>` と `<pthread.h>`
  - `<errno.h>`、`<limits.h>`、`<signal.h>`、および `<unistd.h>`
- 標準 Solaris リンカ `ln(1)`

- Solaris スレッドライブラリ (libthread) と POSIX スレッドライブラリ (libpthread)。セマフォ用の POSIX リアルタイムライブラリ (librt) も必要な場合があります。
- 「MT-安全」ライブラリ
  (libc、libm、libw、libintl、libnsl、libsocket、libmalloc、libmapmalloc など)

セマンティクスの選択 ー Solaris または POSIX

一部の関数 (表 7–1 に示した関数など) は、POSIX 1003.1c 規格でのセマンティクスが Solaris オペレーティング環境 2.4 リリースでのセマンティクスと異なっています (後者は、より前の POSIX 草稿に基づいています)。関数の定義はコンパイル時に選択します。パラメタと戻り値の相違点については、『*man pages section 3*』を参照してください。

表 7–1 POSIX と Solaris でセマンティクスの異なる関数

sigwait(2)	asctime_r(3C)
ctime_r(3C)	getlogin_r(3C)
ftrylockfile(3S) - 新規	getgrgid_r(3C)
getgrnam_r(3C)	getpwuid_r(3C)
getpwnam_r(3C)	ttyname_r(3C)
readdir_r(3C)	

Solaris の fork(2) 関数はすべてのスレッドを複製しますが (汎用 fork 動作)、POSIX の fork(2) 関数は Solaris の fork1() 関数と同様、呼び出しスレッドのみを複製します (fork1 動作)。

alarm(2) の処理も異なります。Solaris のアラームはそのスレッドの LWP に向けられますが、POSIX のアラームはプロセス全体に向けられます (詳細は、172ページの「スレッドごとのアラーム」を参照してください)。

<thread.h> または <pthread.h> の組み込み

インクルードファイル <thread.h> は、旧リリースの Solaris オペレーティング環境と上方互換性のあるコードをコンパイルするときに使用します (–lthread ライブラリとともに使用します)。このライブラリには両方のインタフェース、すなわち Solaris セマンティクスをもつインタフェースと POSIX セマンティクスをもつインタフェースが含まれています。POSIX スレッドで thr_setconcurrency(3T) を呼び出すためには、<thread.h> を組み込む必要があります。

インクルードファイル <pthread.h> は、POSIX 1003.1c 規格で定義されているマルチスレッドインタフェースに適合するコードをコンパイルするときに使用します (–lpthread ライブラリとともに使用します)。POSIX 完全準拠を実現するには、定義フラグ _POSIX_C_SOURCE を下記のように 199506 以上の値 (long) に設定する必要があります。

cc [*flags*] *file*... **-D_POSIX_C_SOURCE=***N* (*N* は 199506L)

Solaris スレッドと POSIX スレッドを同じアプリケーションの中で混用できます。それには、<thread.h> と <pthread.h> の両方を組み込み、–lthread と –lpthread のどちらかのライブラリとリンクします。

両者を混用した場合、コンパイルで –D_REENTRANT を指定し、リンクで –lthread を指定すると、Solaris セマンティクスが支配します。逆にコンパイルで –D_POSIX_C_SOURCE を指定し、リンクで –lpthread を指定すると、POSIX セマンティクスが支配します。

_REENTRANT または _POSIX_C_SOURCE の指定

POSIX 動作を望む場合は、–D_POSIX_C_SOURCE フラグで 199506L 以上の値を指定してアプリケーションをコンパイルしてください。Solaris 動作を望む場合は、–D_REENTRANT フラグを指定してマルチスレッドプログラムをコンパイルしてください。これは、アプリケーションのすべてのモジュールに当てはまります。

混用アプリケーションの場合 (たとえば、Solaris スレッドを POSIX セマンティクスで使用する場合)、コンパイルで –D_REENTRANT フラグと –D_POSIX_PTHREAD_SEMANTICS フラグを指定します。

単一のスレッドのアプリケーションをコンパイルするときは、–D_REENTRANT も –D_POSIX_C_SOURCE フラグも指定しないでください。これらのフラグを指定しなければ、errno、stdio などの以前の定義がすべてそのまま効力を持ちます。

注 - スレッドライブラリ (libthread.so.1 または libpthread.so.1) にリンクされておらず、–D_REENTRANT フラグが指定されていない、シングルスレッドのアプリケーションをコンパイルしてください。これによって、putc(3s) などのマクロが再入可能な関数呼び出しに変換されるときに生じる性能の低下が少なくなります。

要約すると、-D_POSIX_C_SOURCE が指定された POSIX アプリケーションは、表 7–1 に記載されているルーチンに関して、POSIX 1003.1c セマンティクスを持ちます。–D_REENTRANT のみが指定されたアプリケーションは、これらのルーチンに関して Solaris セマンティクスを持ちます。また、–D_POSIX_PTHREAD_SEMANTICS が指定された Solaris アプリケーションは、これらのルーチンに関して POSIX セマンティクスを持ちますが、Solaris スレッドインタフェースを使用することもできます。

-D_POSIX_C_SOURCE と –D_REENTRANT の両方が指定されたアプリケーションは、POSIX セマンティクスを持ちます。

libthread または libpthread とのリンク

POSIX スレッドの動作を望む場合は、–lpthread ライブラリをロードしてください。Solaris スレッドの動作を望む場合は、–lthread ライブラリをロードします。POSIX のプログラマでも、–lthread を指定してリンクすることにより、Solaris での fork() と fork1() の区別を維持したい場合があるでしょう。–lpthread を実行すると、fork() の動作を Solaris の fork1() 呼び出しと同じものにし、alarm(2) の動作を変更します。

libthread を使用するには –lthread を ld コマンドでは –lc の前、cc コマンドでは最後にそれぞれ指定してください。

libpthread を使用するには –lpthread を ld コマンドでは –lc の前、cc コマンドでは最後にそれぞれ指定してください。

スレッドを用いないプログラムをリンクするときは、–lthread と –lpthread は指定しないでください。指定すると、リンク時にマルチスレッド機構が設定され、実行時に動作してしまいます。これは、シングルスレッドアプリケーションの実行速度を低下させ、リソースを浪費し、デバッグの際に誤った結果をもたらします。

図 7–1 は、コンパイルオプションを図解したものです。

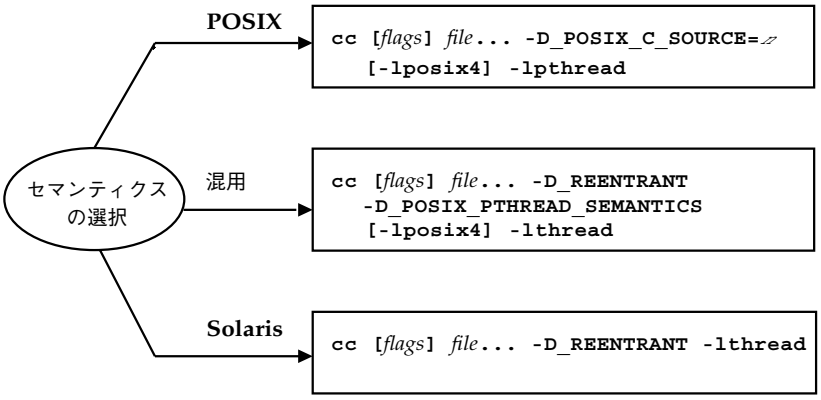

図 7–1　コンパイルフローチャート

混用の場合は、thread.h と pthread.h の両方を組み込む必要があります。

リンクで –lthread も –lpthread も指定しないと、libthread と libpthread に対するすべての呼び出しが動作しなくなります。実行時ライブラリ libc には、libthread と libpthread 内の関数の仮エントリが NULL 手続きとして数多く定義されています。正しい手続きは、libc とスレッドライブラリ (libthread または libpthread) の両方がリンクされたときに、そのスレッドライブラリによって挿入されます。

次のように正しくないフラグを指定して ld コマンドでプログラムをリンクすると、C ライブラリの動きが保証できなくなります。

```
.o's ... -lc -lthread ... (正しくない)
```

または

```
.o's ... -lc -lpthread ... (正しくない)
```

注 - スレッドを使用する C++ プログラムでは、アプリケーションをコンパイルしてリンクするには、–lthread ではなく –mt オプションを使用します。–mt オプションは libthread とリンクし、ライブラリを適切な順序でリンクします。–lthread オプションを使用すると、プログラムがコアダンプすることがあります。

リンク時の **POSIX** セマフォ用 -lposix4 の指定

Solaris セマフォルーチン sema_*(3T) は、libthread ライブラリに入っています。それに対し、POSIX 1003.1c セマフォルーチン sem_*(3R) を必要とする場合

は、-lposix4 ライブラリをリンクします (セマフォルーチンについては、140ページの「セマフォ」を参照してください)。

新旧のモジュールのリンク

表 7–2 に、マルチスレッド化されたオブジェクトモジュールと、以前のオブジェクトモジュールをリンクする場合の注意事項を示します。

表 7–2 コンパイル時の _REENTRANT フラグの有無

ファイルの種類	コンパイル時の指定	参照方法	戻す情報
以前のオブジェクトファイル (スレッド化されていない) と新しいオブジェクトファイル	_REENTRANT または _POSIX_C_SOURCE フラグなし	静的記憶領域	従来の errno
新しいオブジェクトファイル	_REENTRANT または _POSIX_C_SOURCE フラグあり	_errno (新しいバイナリエントリポイント)	スレッド定義の errno のアドレス
libnsl[1] の TLI を使用するプログラム	_REENTRANT または _POSIX_C_SOURCE フラグあり (必須)	_t_errno (新しいエントリポイント)	スレッド定義の t_errno のアドレス

1. TLI の広域エラー変数を得るために tiuser.h を組み込む必要があります。

代替の 1 レベル libthread ライブラリのリンク

標準の Solaris スレッドの実装は、おそらくより少ない軽量プロセス（LWP）にユーザレベルのスレッドが多重化された、2 レベルスレッディングモデル上に構築されます。LWP は、オペレーティングシステムによってプロセッサに振り分けられる、基本実行ユニットです。このメカニズムは、アプリケーションを 1 レベルのセマンティクスで書くためにスレッドを 1 対 1 で LWP に関連付ける（THR_BOUND および PTHREAD_SCOPE_SYSTEM フラグ）標準実装で提供されます。

Solaris 8 オペレーティング環境では、ユーザレベルスレッドが 1 対 1 で LWP に関連付けられる 1 レベルモデルの代替スレッド実装を提供します。この実装は、標準

実装よりもシンプルで、いくつかのマルチスレッド化アプリケーションにとっては便利です。これは、POSIX スレッドおよび Solaris スレッドの両方に、標準実装と全く同じインタフェースを提供します。

代替の実装にリンクするには、プログラムへのリンクの際に次の実行パス –R オプションを使用します。

POSIX スレッドの場合は、以下を使用します。

```
cc -mt ... -lpthread ... -R /usr/lib/lwp          (32-bit)
cc -mt ... -lpthread ... -R /usr/lib/lwp/64       (64-bit)
```

Solaris スレッドの場合は、以下を使用します。

```
cc -mt ... -R /usr/lib/lwp             (32-bit)
cc -mt ... -R /usr/lib/lwp/64          (64-bit)
```

以前に標準スレッドライブラリにリンクされたマルチスレッド化プログラムについては、環境変数 LD_LIBRARY_PATH および LD_LIBRARY_PATH_64 を以下のように設定して、実行時プログラムを代替スレッドライブラリに結合することができます。

```
LD_LIBRARY_PATH=/usr/lib/lwp
LD_LIBRARY_PATH=/usr/lib/lwp:/usr/lib/lwp/64
```

環境変数 LD_LIBRARY_PATH が安全なプロセスに対して有効な場合は、この変数によって指定される信頼できるディレクトリのみが実行時リンカーの検索規則の増補に使用されることに注意してください。

代替の 1 レベルスレッド実装を使用する場合は、そのライブラリは非結合スレッドを使用して標準実装よりも多くの LWP を作成することがあります。LWP がオペレーティングシステムメモリーを消費するのに対し、スレッドはユーザレベルメモリーのみ消費します。このように、数千のスレッドを作成するこのライブラリにリンクされたマルチスレッド化アプリケーションは、同数の LWP を作成し、そのアプリケーションをサポートするために必要なリソースからシステムを実行することになります。

マルチスレッドプログラムのデバッグ

よく起こるミス

以下に、マルチスレッドプログラミングでよく起こるミスを示します。

- 呼び出し側のスタックへのポインタを新しいスレッドの引数として渡す。
- 広域メモリー (変更が可能で、かつ共有されている状態) をアクセスするときに同期機構で保護していない。
- 2 つのスレッドが異なる順序で、同じ組の広域リソースへの権利を獲得しようとしてデッドロックが発生する。この場合は、一方のスレッドが最初のリソースを獲得し、もう一方のスレッドが 2 番目のリソースを獲得し、どちらかがリソースを放棄するまで処理が進まなくなります。
- すでに保持しているロックを獲得しようとする (再帰的なデッドロック)。
- 同期機構の安全性に見えない間隙が生じている。これは、同期機構によって保護されているプログラム内で同期機構をいったん解除し、再度獲得してから戻る関数を呼び出していることが原因です。関数の呼び出し側から見ると広域データが保護されているようでも、実際には保護されていません。
- UNIX のシグナルとスレッドを組み合わせて使っている。非同期的なシグナルの処理には `sigwait(2)` を使用するほうがよいでしょう。
- `setjmp(3B)` と `longjmp(3B)` を使用し、相互排他ロックを解放せずにロングジャンプする。
- `*_cond_wait(3T)` または `*_cond_timedwait(3T)` の呼び出しから復帰した後、条件の再評価に失敗した。
- デフォルトスレッドを `PTHREAD_CREATE_JOINABLE` として生成した場合は、その記憶領域を `pthread_join(3T)` で再利用しなければならないことを忘れている。なお、`pthread_exit(3T)` は記憶領域を解放しません。
- 入れ子の深い再帰呼び出しを行なったり、大量の自動配列を使用したりする。マルチスレッドプログラムは、シングルスレッドプログラムよりもスタックの大きさの制限が厳しいので問題の原因となります。
- スタックの大きさの指定が適切でないか、デフォルト以外のスタックを使用している。

次の点にも注意してください。マルチスレッドプログラムの動きは、特にバグがある場合には、同じ入力で続けて実行しても再現性がないことがよくあります。これは、スレッドのスケジューリングの順序が定まっていないからです。

一般にマルチスレッドプログラムのバグは、決定的というよりも統計的な発生傾向を示します。このため実行レベルの問題を見つけるには、ブレークポイントによるデバッグよりもトレースの方が有効です。

TNF ユーティリティによる追跡とデバッグ

TNF ユーティリティ (Solaris システムの一部) は、アプリケーションとライブラリからの性能解析情報の収集、追跡、デバッグに使用します。TNF ユーティリティは、カーネルおよび複数のユーザプロセスとスレッドからの追跡情報を集約するので、マルチスレッドコードに特に有用です。

TNF ユーティリティを使用すると、マルチスレッドプログラムの追跡とデバッグが容易になります。

`truss(1)` の使用

システムコールとシグナルの追跡については、`truss(1)` を参照してください。

`adb(1)` の使用

マルチスレッドプログラム内ですべてのスレッドを結合するときは、スレッドと LWP とは同義になります。その場合は、マルチスレッドプログラミングをサポートする以下の `adb` コマンドを用いて、各スレッドにアクセスできます。

表 7–3 マルチスレッド対応の `adb` コマンド

pid`:A`	*pid* で指定したプロセスに接続する。プロセスと、そのすべての LWP は停止する。
`:R`	プロセスから切り離す。プロセスと、そのすべての LWP は再開される。
`$L`	(停止した) プロセス内の有効な LWP を一覧表示する。
n`:l`	フォーカスを *n* で指定した LWP に切り替える。

表 7–3　マルチスレッド対応の adb コマンド　続く

`$l`	現在のフォーカスの LWP を表示する。
num`:i`	*num* で指定したシグナルを無視する。

以下のコマンドは、条件付きブレークポイントを設定するためによく使用されます。

表 7–4　adb ブレークポイントの設定

[*label*],[*count*]`:b` [*expression*]	*expression* の評価結果が 0 のときにブレークポイントにヒットする。
foo`,ffff:b <g7-0xabcdef`	*g7* = `0xABCDEF` (16 進数値) のときに *foo* で停止する。

dbx の使用

dbx ユーティリティでは、C++、ANSI C、FORTRAN のソースプログラムをデバッグしたり、実行したりできます。dbx のコマンドは、デバッガと同じコマンドを受けつけますが、標準端末 (tty) インタフェースを使用する点が異なります。dbx とデバッガのどちらも、現在はマルチスレッドプログラムのデバッグをサポートしています。dbx とデバッガの詳細は、`dbx(1)` のマニュアルページおよび『*Sun WorkShop* 入門』を参照してください。

以下に示す表 7–5 にある dbx のオプションは、すべてマルチスレッドアプリケーションをサポートできます。

表 7–5　dbx のマルチスレッドプログラム用オプション

オプション	意味
`cont at line [sig signo id]`	*line* で指定した行から *signo* で指定したシグナルで実行を再開する。*id* は実行を再開するスレッドまたは LWP を指定する (デフォルトの値は *all*)。
`lwp`	現在の LWP を表示する。指定の LWP (*lwpid*) に切り替える。

表 7–5 dbx のマルチスレッドプログラム用オプション 続く

オプション	意味
`lwps`	現在のプロセスの、すべての LWP を一覧表示する。
`next ... tid`	指定のスレッドをステップ実行する。関数呼び出しをスキップするときは、その関数呼び出しの間だけ、すべての LWP の実行が暗黙のうちに再開される。実行可能でないスレッドをステップ実行できない。
`next ... lid`	指定の LWP をステップ実行する。その LWP のスレッドが実行可能であることが必要。関数をスキップするとき、すべての LWP の実行が暗黙のうちに再開されることはない。
`step... tid`	指定のスレッドをステップ実行する。関数呼び出しをスキップするときは、その関数呼び出しの間だけ、すべての LWP の実行が暗黙のうちに再開される。実行可能でないスレッドをステップ実行できない。
`step... lid`	指定の LWP をステップ実行する。関数をスキップするとき、すべての LWP の実行が暗黙のうちに再開されることはない。
`stepi... lid`	指定の LWP
`stepi... tid`	指定のスレッドが実行可能である LWP
`thread`	現在のスレッドを表示する。指定のスレッド (*tid*) に切り替える。以下の *tid* のデフォルト値は現在のスレッド
`thread -info [ tid ]`	指定のスレッドの全情報を表示する。
`thread -locks [ tid ]`	指定のスレッドが保持しているロックを一覧表示する。
`thread -suspend [ tid ]`	指定のスレッドを停止状態にする。
`thread -continue [ tid ]`	指定のスレッドの停止状態を解除する。
`thread -hide [ tid ]`	指定のスレッド (または現在のスレッド) を見えなくする。このスレッドは、`threads` オプションのリストには表示されない。

表 7–5 dbx のマルチスレッドプログラム用オプション 続く

オプション	意味
`thread -unhide [ tid ]`	指定のスレッド (または現在のスレッド) の隠蔽を解除する。
`allthread-unhide`	全スレッドの隠蔽を解除する。
`threads`	全スレッドを一覧表示する。
`threads-all`	通常は表示されないスレッド (ゾンビ) を表示する。
`all|filterthreads-mode`	`threads` オプションのスレッド一覧表示にフィルタをかけるかどうかを指定する。
`auto|manualthreads-mode`	スレッドリストの自動更新機能を有効にする。
`threads-mode`	現在のモードをエコーする。以前の任意の書式に続けてスレッドまたは LWP の ID を指定すれば、指定のエンティティのトレースバックを得ることができる。

第 8 章

Solaris スレッドを使ったプログラミング

この章では、Solaris スレッドと POSIX スレッドのアプリケーションプログラミングインタフェース (API) を比較し、POSIX スレッドにはない Solaris の機能について説明します。

Solaris スレッドと POSIX スレッドの API の比較

Solaris スレッド API と POSIX スレッド (pthread) API は、どちらもアプリケーションソフトウェアに並列性を導入する手段です。どちらの API もそれ自体で完結したものですが、Solaris スレッドの関数と pthread の関数を同じプログラムの中で併用することもできます。

ただし、2 つの API は完全に一致しているわけではありません。Solaris スレッドは pthread にない関数をサポートしていて、pthread には Solaris インタフェースでサポートされない関数が含まれています。同じ関数については、機能が実質的に同じでも使用する引数が異なることがあります。

2 つの API を組み合わせて使用すれば、それぞれ他方にない機能を補い合うことができます。また、同じシステムで、Solaris スレッドだけを使用するアプリケーションを実行する一方で、pthread だけを使用するアプリケーションを実行することもできます。

API の主な相違点

Solaris スレッドと pthread は、API の動作や構文も非常によく似ています。主な相違点を表 8–1 に示します。

表 8–1 Solaris スレッドと pthread の相違点

Solaris スレッド **(**libthread**)** に固有	**POSIX** スレッド **(**libpthread**)** に固有
スレッド関数名の接頭辞が thr_ で、セマフォ関数名の接頭辞が sema_	スレッド関数名の接頭辞が pthread_ で、セマフォ関数名の接頭辞が sem_
読み取り / 書き込みロック	属性オブジェクト (Solaris の多くの引数やフラグは pthread の属性オブジェクトと同等である)
デーモンスレッドが生成可能	取り消しセマンティクス
スレッドの停止と再開	スケジューリング方針
並行度の設定 (新しい LWP の要求) と並行度の取得	

関数比較表

表 8–2 は、Solaris スレッドの関数と pthread の関数を比較対照したものです。なお、Solaris スレッドの関数と pthread の関数が本質的に同じものとして並記されている場合でも、その引数は異なっていることがあります。

pthread または Solaris スレッドの側に相当するインタフェースがない場合は、「-」が記入されています。pthread 欄の項目で「POSIX 1003.4」または「POSIX.4」が付記されているものは、POSIX 規格のリアルタイムの仕様の一部で pthread の一部ではありません。

表 **8–2** Solaris スレッドと POSIX pthread の比較

Solaris スレッド **(**libthread**)**	pthread **(**libpthread**)**
thr_create()	pthread_create()
thr_exit()	pthread_exit()
thr_join()	pthread_join()
thr_yield()	sched_yield() POSIX.4
thr_self()	pthread_self()
thr_kill()	pthread_kill()
thr_sigsetmask()	pthread_sigmask()
thr_setprio()	pthread_setschedparam()
thr_getprio()	pthread_getschedparam()
thr_setconcurrency()	pthread_setconcurrency()
thr_getconcurrency()	pthread_getconcurrency()
thr_suspend()	-
thr_continue()	-
thr_keycreate()	pthread_key_create()
-	pthread_key_delete()

表 8–2　Solaris スレッドと POSIX pthread の比較　続く

Solaris スレッド (libthread)	pthread (libpthread)
thr_setspecific()	pthread_setspecific()
thr_getspecific()	pthread_getspecific()
-	pthread_once()
-	pthread_equal()
-	pthread_cancel()
-	pthread_testcancel()
-	pthread_cleanup_push()
-	pthread_cleanup_pop()
-	pthread_setcanceltype()
-	pthread_setcancelstate()
mutex_lock()	pthread_mutex_lock()
mutex_unlock()	pthread_mutex_unlock()
mutex_trylock()	pthread_mutex_trylock()
mutex_init()	pthread_mutex_init()
mutex_destroy()	pthread_mutex_destroy()
cond_wait()	pthread_cond_wait()
cond_timedwait()	pthread_cond_timedwait()

表 8–2　Solaris スレッドと POSIX pthread の比較　続く

Solaris スレッド (libthread)	pthread (libpthread)
cond_signal()	pthread_cond_signal()
cond_broadcast()	pthread_cond_broadcast()
cond_init()	pthread_cond_init()
cond_destroy()	pthread_cond_destroy()
rwlock_init()	pthread_rwlock_init()
rwlock_destroy()	pthread_rwlock_destroy()
rw_rdlock()	pthread_rwlock_rdlock()
rw_wrlock()	pthread_rwlock_wrlock()
rw_unlock()	pthread_rwlock_unlock()
rw_tryrdlock()	pthread_rwlock_tryrdlock()
rw_trywrlock()	pthread_rwlock_trywrlock()
-	pthread_rwlockattr_init()
-	pthread_rwlockattr_destroy()
-	pthread_rwlockattr_getpshared()
-	pthread_rwlockattr_setpshared()
sema_init()	sem_init() POSIX 1003.4
sema_destroy()	sem_destroy() POSIX 1003.4

表 8–2　Solaris スレッドと POSIX pthread の比較　続く

Solaris スレッド **(**libthread**)**	pthread **(**libpthread**)**
sema_wait()	sem_wait() POSIX 1003.4
sema_post()	sem_post() POSIX 1003.4
sema_trywait()	sem_trywait() POSIX 1003.4
fork1()	fork()
-	pthread_atfork()
fork() (複数スレッドコピー)	-
-	pthread_mutexattr_init()
-	pthread_mutexattr_destroy()
cond_init() の type() 引数	pthread_mutexattr_setpshared()
-	pthread_mutexattr_getpshared()
-	pthread_mutex_attr_settype()
-	pthread_mutex_attr_gettype()
-	pthread_condattr_init()
-	pthread_condattr_destroy()
cond_init() の type() 引数	pthread_condattr_setpshared()
-	pthread_condattr_getpshared()
-	pthread_attr_init()

表 8–2　Solaris スレッドと POSIX pthread の比較　続く

Solaris スレッド (libthread)	pthread (libpthread)
-	pthread_attr_destroy()
thr_create() の THR_BOUND フラグ	pthread_attr_setscope()
-	pthread_attr_getscope()
-	pthread_attr_setguardsize()
-	pthread_attr_getguardsize()
thr_create() の stack_size() 引数	pthread_attr_setstacksize()
-	pthread_attr_getstacksize()
thr_create() の stack_addr() 引数	pthread_attr_setstackaddr()
-	pthread_attr_getstackaddr()
thr_create() の THR_DETACH フラグ	pthread_attr_setdetachstate()
-	pthread_attr_getdetachstate()
-	pthread_attr_setschedparam()
-	pthread_attr_getschedparam()
-	pthread_attr_setinheritsched()
-	pthread_attr_getinheritsched()

表 8–2　Solaris スレッドと POSIX pthread の比較　続く

Solaris スレッド **(**libthread**)**	pthread **(**libpthread**)**
-	pthread_attr_setsschedpolicy()
-	pthread_attr_getschedpolicy()

この章で説明する Solaris スレッドの関数を使用するには、リンクで Solaris スレッドライブラリ (–lthread) を指定しなければなりません。

Solaris スレッドと pthread で機能的にほとんど変わらない場合は (関数名と引数が違うとしても)、正しいインクルードファイルと関数プロトタイプを示した簡単な例を挙げているだけです。Solaris スレッドで戻り値が記述されていないものについては、『*man pages section 3*』から該当するページを探して、その関数の戻り値を調べてください。

Solaris 関連の関数の詳細は、pthread の関連マニュアルで類似した名前の関数を調べてください。

Solaris スレッドの関数で pthread にない機能をもつものについて、詳しく説明しています。

Solaris スレッドに固有の関数

- 225ページの「スレッド実行の停止」
- 226ページの「停止しているスレッドの再開」
- 226ページの「スレッドの並行度の設定」
- 228ページの「スレッドの並行度の取得」

thr_suspend(3THR)

thr_suspend(3THR) は、*target_thread* で指定したスレッドの実行をただちに停止させます。thr_suspend() が正常終了した時点で、指定のスレッドは実行状態ではありません。

停止しているスレッドに対して再度 thr_suspend() を発行しても効果はありません。停止しているスレッドをシグナルで呼び起こすことはできません。スレッドが実行を再開するまでシグナルは保留状態のままです。

```
#include <thread.h>

int thr_suspend(thread_t tid);
```

次の例では、pthread で定義されている pthread_t *tid* と Solaris スレッドの thread_t *tid* が同じです。*tid* 値は、代入によっても型変換によっても使用できます。

```
thread_t tid; /* thr_create() からの tid */

/* pthread_create() で生成されたスレッドからの */
/* Solaris tid に相当する pthread */
pthread_t ptid;

int ret;

ret = thr_suspend(tid);

/* 型変換で pthread ID 変数を使用する */
ret = thr_suspend((thread_t) ptid);
```

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、thr_suspend() は失敗し、対応する値を返します。

ESRCH

現在のプロセスに *tid* が存在しません。

停止しているスレッドの再開

thr_continue(3THR)

thr_continue(3THR) は、停止しているスレッドの実行を再開します。再開したスレッドに対して再度 thr_continue() を発行しても効果はありません。

```
#include <thread.h>

int thr_continue(thread_t tid);
```

停止しているスレッドがシグナルで呼び起こされることはありません。送られたシグナルは、そのスレッドが thr_continue() で再開されるまで保留されます。

pthread で定義されている pthread_t *tid* と Solaris スレッドの thread_t *tid* が同じです。*tid* 値は、代入によっても型変換によっても使用できます。

```
thread_t tid; /* thr_create() からの tid */

/* pthread_create() で生成されたスレッドからの Solaris tid に */
/* 相当する pthread */
pthread_t ptid;

int ret;

ret = thr_continue(tid);

/* 型変換で pthread ID 変数を使用する */
ret = thr_continue((thread_t) ptid)
```

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下の条件が検出されると、thr_continue() は失敗し、対応する値を戻します。

ESRCH

現在のプロセスに *tid* が存在しません。

スレッドの並行度の設定

Solaris スレッドは、デフォルトでは、非結合スレッドの実行に使用するシステム実行リソース (LWP) を、有効なスレッドの実際の数に合わせて調整しようとします。

Solaris スレッドパッケージは完璧な判定はできなくても、少なくともプロセスが実行を継続できるようにします。

どれだけの数の非結合スレッド (実行するコードまたはシステムコール) を同時に有効すべきか見当がつく場合は、thr_setconcurrency() で指定してください。使用されているスレッドの数を取得するには、thr_getconcurrency() を使用してください。

thr_setconcurrency(3THR)

thr_setconcurrency(3THR) は、アプリケーションの中で必要とする並行度の目標値をシステムに指示します。システムは、十分な数のスレッドを有効にして、プロセスが実行を継続できるようにします。

```
#include <thread.h>


int new_level;
int ret;


ret = thr_setconcurrency(new_level);
```

プロセス内の非結合スレッドを同時に有効にする必要があるかどうかは、状況によって変化します。スレッドシステムのデフォルト設定では、システムリソースを節約することを前提にして、プロセスに必要な数のスレッドを有効にします。また、並行度を小さくしすぎてプロセスがデッドロックに陥るといった事態が生じないように並行度も調整されます。

このようなデフォルトの設定では効果的な並行度が得られない場合、アプリケーション側は thr_setconcurrency() の *new_level* の指定で、スレッドシステムに並行度の目標値を指示できます。

同時に有効になるスレッドの実際の数は、*new_level* より大きいことも小さいこともあります。

計算を目的とするスレッドが複数存在するアプリケーションでは、thr_setconcurrency() によって実行リソースの並行度を調整しておかないと、実行可能なすべてのスレッドのスケジューリングが適切に行われないことがあります。

thr_create() の THR_NEW_LWP フラグでも並行度に影響を与えることができます。これには、現在の並行度を 1 だけ大きくする効果があります。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると、thr_setconcurrency() は失敗し、対応する値を返します。

EAGAIN

指定の並行度ではシステムリソースの制限を超えます。

EINVAL

new_level の値が負です。

スレッドの並行度の取得

thr_getconcurrency(3THR)

thr_getconcurrency(3THR) は、thr_sgetconcurrency() で設定された並行度の現在値を取得します。同時に有効になっているスレッドの実際の数は、この値より大きいことも小さいこともあります。

```
#include <thread.h>

int thr_getconcurrency(void)
```

戻り値

thr_getconcurrency() は、常に並行度の現在の値を戻します。

pthread に相当するものがある同期関数 — 読み取り / 書き込みロック

読み取り / 書き込みロックを使用すると、同時に書き込み操作ができるスレッドを 1 つだけに制限する一方、読み取り操作は同時に複数のスレッドからできるようになります。

■ 229ページの「読み取り / 書き込みロックの初期化」

- 231ページの「読み取りロックの獲得」
- 232ページの「読み取りロックの獲得 (ブロックなし)」
- 233ページの「書き込みロックの獲得」
- 234ページの「書き込みロックの獲得」
- 234ページの「読み取り / 書き込みロックの解除」
- 235ページの「読み取り / 書き込みロックの削除」

すでに読み取りロックを保持しているスレッドがある場合、他のスレッドがさらに読み取りロックを獲得できますが、書き込みロックを獲得するときは待たなければなりません。すでに書き込みロックを保持しているスレッドがある場合、あるいは書き込みロックの獲得を待っているスレッドがある場合、他のスレッドは読み取りと書き込みのどちらのロックを獲得するときも待たなければなりません。

読み取り / 書き込みロックは、相互排他ロックよりも低速です。しかし、書き込みの頻度が低く、かつ多数のスレッドから並行的に読み取られるようなデータを保護するときに特に性能を改善します。

現在のプロセス内のスレッドと他のプロセス内のスレッドの間で、読み取り / 書き込みロックを使って同期をとる場合は、連携するそれらのプロセスの間で共有される書き込み可能なメモリーに、読み取り / 書き込みロックの領域を確保し (mmap(2) のマニュアルページを参照)、その読み取り / 書き込みロックをプロセス間同期用に初期化します。

複数のスレッドが読み取り / 書き込みロックを待っている場合のロックの獲得順序は、特に指定しなければ不定です。ただし、書き込み側がいつまでもロックを獲得できないような事態を回避するため、Solaris スレッドパッケージでは書き込み側が読み取り側より優先されます。

読み取り / 書き込みロックは、使用する前に初期化する必要があります。

読み取り / 書き込みロックの初期化

rwlock_init(3THR)

```
#include <synch.h>  (または #include <thread.h>)
```

(続く)

```
int rwlock_init(rwlock_t *rwlp, int type, void * arg);
```

rwlock_init(3THR) は、*rwlp* が指す読み取り / 書き込みロックを初期化してロック解除状態に設定します。*type* には次のいずれかを指定できます (*arg* は現在は無視されます)。(POSIX スレッドについては、154ページの「pthread_rwlock_init(3THR)」を参照)。

- USYNC_PROCESS

  このプロセス内のスレッドと他のプロセス内のスレッドとの間で同期をとることができるようにします。*arg* は無視されます。

- USYNC_THREAD

  このプロセス内のスレッドの間だけで同期をとることができるようにします。*arg* は無視されます。

複数のスレッドから同じ読み取り / 書き込みロックを同時に初期化してはいけません。0 に初期化したメモリーに領域を確保することによって、読み取り / 書き込みロックを初期化することもできます。その場合は、*type* に USYNC_THREAD を指定したものとみなされます。一度初期化した読み取り / 書き込みロックは、他のスレッドで使われている可能性があるので再初期化してはいけません。

プロセス内スコープでの読み取り / 書き込みロックの初期化

```
#include <thread.h>

rwlock_t rwlp;
int ret;

/* このプロセスの中だけで使用する */
ret = rwlock_init(&rwlp, USYNC_THREAD, 0);
```

プロセス間スコープでの読み取り / 書き込みロックの初期化

```
#include <thread.h>

rwlock_t rwlp;
int ret;

/* すべてのプロセスの間で使用する */
ret = rwlock_init(&rwlp, USYNC_PROCESS, 0);
```

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

引数が無効です。

EFAULT

rwlp または *arg* が無効なアドレスを指しています。

読み取りロックの獲得

rw_rdlock(3THR)

```
#include <synch.h> (または #include <thread.h>)

int rw_rdlock(rwlock_t *rwlp);
```

rw_rdlock(3THR) は、*rwlp* が指す読み取り / 書き込みロックの読み取りロックを獲得します。指定した読み取り / 書き込みロックが書き込み用にすでにロックされている場合、呼び出しスレッドは書き込みロックが解放されるまでブロックされます。そうでなければ、読み取りロックを獲得します。(POSIX スレッドについては、155ページの「pthread_rwlock_rdlock(3THR)」を参照)。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると、この関数は失敗し、対応する値を戻します。

EINVAL

引数が無効です。

EFAULT

rwlp が無効なアドレスを指しています。

読み取りロックの獲得 (ブロックなし)

rw_tryrdlock(3THR)

```
#include <synch.h>  (または #include <thread.h>)

int rw_tryrdlock(rwlock_t *rwlp);
```

rw_tryrdlock(3THR) は、*rwlp* が指す読み取り / 書き込みロックの読み取りロックを獲得しようとします。指定した読み取り / 書き込みロックが書き込み用にすでにロックされている場合は、エラーを戻します。そうでなければ、呼び出しスレッドは読み取りロックを獲得します。(POSIX スレッドについては、156ページの「pthread_rwlock_tryrdlock(3THR)」を参照)。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

引数が無効です。

EFAULT

rwlp が無効なアドレスを指しています。

EBUSY

rwlp が指す読み取り / 書き込みロックがすでにロックされています。

書き込みロックの獲得

rw_wrlock(3THR)

```
#include <synch.h>  (または #include <thread.h>)

int rw_wrlock(rwlock_t *rwlp);
```

rw_wrlock(3THR) は、*rwlp* が指す読み取り / 書き込みロックの書き込みロックを獲得します。指定した読み取り / 書き込みロックが、読み取りまたは書き込み用にすでにロックされている場合、呼び出しスレッドは、すべての読み取りロックと書き込みロックが解放されるまでブロックされます。読み取り / 書き込みロックの書き込みロックを保持できるスレッドは一度に 1 つに限られます。(POSIX スレッドについては、157ページの「pthread_rwlock_wrlock(3THR)」を参照)。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

引数が無効です。

EFAULT

rwlp が不当なアドレスを指しています。

書き込みロックの獲得

rw_trywrlock(3THR)

```
#include <synch.h>  (または #include <thread.h>)

int rw_trywrlock(rwlock_t *rwlp);
```

rw_trywrlock(3THR) は、*rwlp* が指す読み取り / 書き込みロックの書き込みロックを獲得しようとします。指定した読み取り / 書き込みロックが、読み取りまたは書き込み用にすでにロックされている場合はエラーを戻します。(POSIX スレッドについては、158ページの「pthread_rwlock_trywrlock(3THR)」を参照)。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

引数が無効です。

EFAULT

rwlp が無効なアドレスを指しています。

EBUSY

rwlp が指す読み取り / 書き込みロックがすでにロックされています。

読み取り / 書き込みロックの解除

rw_unlock(3THR)

```
#include <synch.h>  (または #include <thread.h>)

int rw_unlock(rwlock_t *rwlp);
```

rw_unlock(3THR) は、*rwlp* が指す読み取り / 書き込みロックのロックを解除します。解除の対象となる読み取り / 書き込みロックは、ロックされていて、呼び出しスレッドが読み取り用または書き込み用に保持しているものでなければなりません。その読み取り / 書き込みロックが使用可能になるのを待っているスレッドが他にある場合は、そのスレッドのうちの 1 つがブロック解除されます。(POSIX スレッドについては、159ページの「pthread_rwlock_unlock(3THR)」を参照)。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

引数が無効です。

EFAULT

rwlp が無効なアドレスを指しています。

読み取り / 書き込みロックの削除

rwlock_destroy(3THR)

```
#include <synch.h>  (または #include <thread.h>)

int rwlock_destroy(rwlock_t *rwlp);
```

rwlock_destroy(3THR) は、*rwlp* が指す読み取り / 書き込みロックを削除します。読み取り / 書き込みロックの記憶領域は解放されません。(POSIX スレッドについては、160ページの「pthread_rwlock_destroy(3THR)」を参照)。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると、この関数は失敗し、対応する値を返します。

EINVAL

引数が無効です。

EFAULT

rwlp が無効なアドレスを指しています。

読み取り / 書き込みロックの例

例 8–1 では、銀行口座に関する処理で読み取り / 書き込みロックを使用しています。口座残高に対して複数のスレッドが並行的に読み取り専用アクセスできますが、書き込みは 1 つのスレッドだけに制限されます。get_balance() 関数中のロックは、当座預金の残高 (checking_balance) と普通預金の残高 (saving_balance) を合計する演算が、原子操作によって行われることを保証するため必要です。

例 8–1 銀行口座の読み取り / 書き込み

```
rwlock_t account_lock;
float checking_balance = 100.0;
float saving_balance = 100.0;
...
rwlock_init(&account_lock, 0, NULL);
...

float
get_balance() {
    float bal;

    rw_rdlock(&account_lock);
    bal = checking_balance + saving_balance;
    rw_unlock(&account_lock);
    return(bal);
}

void
transfer_checking_to_savings(float amount) {
    rw_wrlock(&account_lock);
    checking_balance = checking_balance - amount;
    saving_balance = saving_balance + amount;
    rw_unlock(&account_lock);
}
```

pthread に相当するものがある Solaris スレッドの関数

操作	参照先
スレッドの生成	238ページの「thr_create(3THR)」
最小のスタックの大きさの取得	241ページの「thr_min_stack(3THR)」
スレッド識別子の取得	242ページの「thr_self(3THR)」
スレッドの実行明け渡し	242ページの「thr_yield(3THR)」
シグナルのスレッドへの送信	242ページの「thr_kill(3THR)」
呼び出しスレッドのシグナルマスクのアクセス	243ページの「thr_sigsetmask(3THR)」
スレッドの終了	243ページの「thr_exit(3THR)」
スレッドの終了待ち	243ページの「thr_join(3THR)」
スレッド固有データ用キーの作成	245ページの「thr_keycreate(3THR)」
スレッド固有データ用キーの設定	245ページの「thr_setspecific(3THR)」
スレッド固有データ用キーの取得	245ページの「thr_getspecific(3THR)」

操作	参照先
スレッド優先順位の設定	246ページの「thr_setprio(3THR)」
スレッド優先順位の取得	247ページの「thr_getprio(3THR)」

スレッドの生成

thr_create(3THR) は、Solaris スレッドライブラリルーチンの中で最も精巧なルーチンの 1 つです。

thr_create(3THR)

thr_create(3THR) は、現在のプロセスに新しい制御スレッドを追加します。(POSIX スレッドについては、33ページの「pthread_create(3THR) 」を参照)。

新しいスレッドは保留状態のシグナルは継承しませんが、優先順位とシグナルマスクを継承することに注意してください。

```
#include <thread.h>

int thr_create(void *stack_base, size_t stack_size,
    void *(*start_routine) (void *), void *arg, long flags,
    thread_t *new_thread);

size_t thr_min_stack(void);
```

stack_base — 新しいスレッドが使用するスタックのアドレスを指定します。NULL を指定すると、新しいスレッドに *stack_size* バイト以上の大きさをもつスタックが割り当てられます。

stack_size — 新しいスレッドが使用するスタックのバイト数を指定します。0 を指定するとデフォルト値が使用されます。通常は 0 を指定してください。それ以外の値を指定する場合は、thr_min_stack() で戻された値よりも大きな値を指定してください。

通常は、スレッドのためのスタック空間を割り当てる必要はありません。スレッドライブラリが、各スレッドのスタック用に 1M バイトの仮想記憶をスワップ空間の予約なしで割り当てます。(スレッドライブラリは、mmap(2) の MAP_NORESERVE オプションを使って割り当てます。)

start_routine — 新しいスレッドで実行する関数を指定します。`start_routine()` で指定した関数が終了すると、スレッドはその関数の戻り値を終了状態に設定して終了します (詳細は、243ページの「thr_exit(3THR)」を参照) してください。

arg — `void` で記述される任意のもの。通常は 4 バイト値です。それよりも大きな値は、そのポインタを引数とすることによって間接的に渡さなければなりません。

引数は 1 つしか指定できません。複数の引数を与えるためには、それらを 1 つのものとして (構造体に入れるなどの方法で) コーディングしてください。

flags — 生成されるスレッドの属性を指定します。通常は 0 を指定します。

flags の値は、以下に示すフラグのビット単位の論理和となります。

- `THR_SUSPENDED` — 新しいスレッドを停止させます。`thr_continue()` でスレッドを再開するまで *start_routine* は実行されません。このフラグは、スレッドを実行する前に優先順位の変更などを行いたいときに使用します。切り離されたスレッドの終了は無視されます。
- `THR_DETACHED` — 新しいスレッドを切り離します。その結果、このスレッドのスレッド識別子やその他のリソースが、スレッド終了後ただちに再利用できるようになります。このフラグは、スレッドの終了を待つ必要がないときに設定してください。

注 - 明示的な同期によって阻止されなければ、停止していない切り離されたスレッドは、そのスレッドの生成元が `thr_create()` から復帰する前に終了でき、そのスレッド識別子は別の新しいスレッドに割り当てることができます。

- `THR_BOUND` — 新しいスレッドを LWP に固定的に結合します (新しいスレッドは結合スレッドになります)。
- `THR_NEW_LWP` — 非結合スレッドの並行度を 1 だけ増やします。この効果は、`thr_setconcurrency(3T)` で並行度を 1 だけ増やす場合と似ていますが、`thr_setconcurrency()` 関数で設定される並行度には影響しません。通常、`THR_NEW_LWP` を指定すると、非結合スレッドを実行する LWP プールに新しい LWP が 1 つ追加されます。
- `THR_BOUND` と `THR_NEW_LWP` の両方を指定すると、通常は 2 つの LWP が生成されます。1 つは結合スレッドのための LWP で、もう 1 つは非結合スレッドを実行する LWP プールに追加される LWP です。
- `THR_DAEMON` — 新しいスレッドをデーモンにします。デーモンでないスレッドがすべて終了すると、プロセスは終了します。デーモンスレッドは、プロセスの終了状態に影響を与えず、また終了するスレッド数にも含まれません。

プロセスの終了には、次の 2 通りの方法があります。1 つは exit() を呼び出す方法です。もう 1 つは、プロセス内のスレッドのうち THR_DAEMON フラグを指定せずに生成されたすべてのスレッドが thr_exit(3T) を呼び出す方法です。アプリケーションまたはそれが呼び出すライブラリでは、終了判断の際に無視される (数えられない) ようなスレッドを生成できます。THR_DAEMON フラグは、プロセスの終了条件に関係しないスレッドを生成するときに指定します。

new_thread — NULL 以外を指定すると、*new_thread* の指すアドレスに新しいスレッドのスレッド識別子が格納されます。この引数が指す記憶領域は、呼び出し側の責任で確保しなければなりません。このスレッド識別子は、呼び出し側のプロセス内でだけ有効です。

スレッド識別子が特に必要でなければ、*new_thread* に 0 を指定してください。

戻り値

正常終了時は 0 です。それ以外の戻り値は、エラーが発生したことを示します。以下のいずれかの条件が検出されると、thr_create() は失敗し、対応する値を戻します。

EAGAIN

システム制限を超えました。たとえば、生成された LWP が多すぎます。

ENOMEM

新しいスレッドを生成するための十分なメモリーがありません。

EINVAL

stack_base が NULL でなく、しかも *stack_size* に thr_min_stack() の戻り値より小さな値を指定しました。

スタックの動作

Solaris スレッドでのスタックの動作は、通常は pthread の場合と同じです。スタックの設定と操作の詳細は、79ページの「スタックについて」を参照してください。

thr_min_stack() を呼び出すと、スタックの大きさの絶対最小値が得られます。この関数は、NULL 手続きを実行するスレッドに必要なスタック空間の大きさを戻します。実用的なスレッドに必要なスタック空間はもっと大きいので、スタックの大きさを小さくするときは十分注意してください。

独自のスタックを指定する方法は 2 通りあります。1 つは、thr_create() でスタックアドレスを NULL に指定し、スタック空間の割り当てをスレッドライブラリに任せる方法です。スタックの大きさを指定するパラメタには、希望の大きさを指定します。

もう 1 つの方法は、thr_create() でスタックアドレスを指定して、スタックをすべて自分で管理する方法です。この場合は、スタック空間の割り当てだけでなく解放もユーザ自身で行う必要があります。つまり、スレッドの終了時にスタックを処分しなければなりません。

独自のスタックを割り当てる場合は、mprotect(2) を呼び出して、スタックの最後に必ずレッドゾーンを付加してください。

最小のスタックの大きさの取得

thr_min_stack(3THR)

thr_min_stack(3THR) は、スレッドの最小のスタックの大きさを取得します。

```
#include <thread.h>

size_t thr_min_stack(void);
```

NULL スレッドを実行するために必要なスタック空間の大きさが戻されます (NULL スレッドとは、中身のない (NULL) 手続きを実行するために生成されるスレッドのことです)。

スレッドが NULL 手続きでなく通常の手続きを実行する場合
は、thr_min_stack() の戻り値よりも大きなスタックの大きさを割り当てなければなりません。

スレッドの生成時に、ユーザが独自のスタックを指定する場合は、そのスレッドを実行するために十分な大きさのスタック空間を、ユーザ自身が確保しなければなりません。動的にリンクされるような実行環境では、スレッドのスタックの大きさの最小限必要な量を見積もることは困難です。

通常、ユーザ独自のスタックが必要になることはまれです。実際、アプリケーション側が実行環境を完全に制御するなどのごく限られた状況でしか必要になりません。

ユーザは、スレッドライブラリにスタックの割り当てを任せることができます。スレッドライブラリのデフォルトのスタックは、すべてのスレッドの要求を満たします。

スレッド識別子の取得

thr_self(3THR)

`thr_self(3THR)` は、呼び出しスレッドの識別子を取得します。(POSIX スレッドについては、44ページの「pthread_self(3THR)」を参照)。

```
#include <thread.h>

thread_t thr_self(void);
```

スレッドの実行明け渡し

thr_yield(3THR)

`thr_yield(3THR)` は、現在のスレッドから同じ優先順位か、より高い優先順位をもつ別のスレッドに実行権を譲ります。それ以外は何の効果もありません。`thr_yield()` の呼び出しスレッドがそうするという保証はありません。

```
#include <thread.h>

void thr_yield(void);
```

シグナルのスレッドへの送信

thr_kill(3THR)

`thr_kill(3THR)` は、スレッドにシグナルを送ります。(POSIX スレッドについては、48ページの「pthread_kill(3THR)」を参照)。

```
#include <thread.h>
#include <signal.h>


int thr_kill(thread_t target_thread, int sig);
```

呼び出しスレッドのシグナルマスクのアクセス

thr_sigsetmask(3THR)

thr_sigsetmask(3THR) は、呼び出しスレッドのシグナルマスクの変更や照会を行います。

```
#include <thread.h>
#include <signal.h>


int thr_sigsetmask(int how, const sigset_t *set, sigset_t *oset);
```

スレッドの終了

thr_exit(3THR)

thr_exit(3THR) はスレッドを終了させます。(POSIX スレッドについては、51 ページの「pthread_exit(3THR)」を参照)。

```
#include <thread.h>

void thr_exit(void *status);
```

スレッドの終了待ち

thr_join(3THR)

thr_join(3THR) 関数はスレッドの終了を待ちます。(POSIX スレッドについては、34ページの「pthread_join(3THR)」を参照)。

```
#include <thread.h>

int thr_join(thread_t tid, thread_t *departedid, void **status);
```

指定したスレッドの終了待ち

```
#include <thread.h>

thread_t tid;
thread_t departedid;
int ret;
int status;

/* スレッド「tid」の終了待ち、status の指定あり */
ret = thr_join(tid, &departedid, (void**)&status);

/* スレッド「tid」の終了待ち、status の指定なし */
ret = thr_join(tid, &departedid, NULL);

/* スレッド「tid」の終了待ち、departedid と status の指定なし */
ret = thr_join(tid, NULL, NULL);
```

tid が (thread_t) 0 の場合は、thr_join() はプロセス内の切り離されていない任意のスレッドの終了を待ちます。つまり、スレッド識別子を指定しなければ、切り離されていないスレッドのどれかが終了すると thr_join() が復帰します。

任意のスレッドの終了待ち

```
#include <thread.h>

thread_t tid;
thread_t departedid;
int ret;
int status;

/* スレッド「tid」の終了待ち、status の指定あり */
ret = thr_join(NULL, &departedid, (void **)&status);
```

thr_join() でスレッド識別子として NULL を指定すると、プロセス内の切り離されていない任意のスレッドの終了を待ちます。*departedid* には、終了したスレッドのスレッド識別子が格納されます。

スレッド固有データ用キーの作成

関数名と引数を別にすれば、スレッド固有データは Solaris のものも POSIX のものも同じです。この節では、Solaris の関数の概要を説明します。

thr_keycreate(3THR)

`thr_keycreate(3THR)` は、プロセス内のスレッド固有データを識別するためのキーを割り当てます。(POSIX スレッドについては、38ページの「pthread_key_create(3THR)」を参照)。

```
#include <thread.h>

int thr_keycreate(thread_key_t *keyp,
    void (*destructor) (void *value));
```

スレッド固有データ用キーの設定

thr_setspecific(3THR)

`thr_setspecific(3THR)` は、呼び出しスレッドで、値 (*value*) とスレッド固有データのキー (*key*) を結び付けます。(POSIX スレッドについては、40ページの「pthread_setspecific(3THR)」を参照)。

```
#include <thread.h>

int thr_setspecific(thread_key_t key, void *value);
```

スレッド固有データ用キーの取得

thr_getspecific(3THR)

`thr_getspecific(3THR)` は、*key* で指定したキーに結び付けられている現在の値を、*valuep* が指している位置に格納します。(POSIX スレッドについては、41ページの「pthread_getspecific(3THR)」を参照)。

```
#include <thread.h>

int thr_getspecific(thread_key_t key, void **valuep);
```

スレッド優先順位の設定

Solaris スレッドでは、優先順位が親と異なるスレッドを生成する場合、SUSPEND モードで生成します。そして、停止状態のときに `thr_setprio(3T)` 関数を使ってスレッド優先順位を変更し、実行を再開します。

通常、非結合スレッドのスケジューリングは、プロセス内の他のスレッドとの関係だけを考慮した単純な優先順位に基づいて行われます。その他の調整が行われたり、カーネルが関係したりすることはありません。スレッドの優先順位は通常は同一であり、生成側プロセスの優先順位を継承します。

thr_setprio(3THR)

`thr_setprio(3THR)` は、現在のプロセス内の *tid* で指定したスレッドの優先順位を、*newprio* で指定した優先順位に変更します。(POSIX スレッドについては、47 ページの「pthread_setschedparam(3THR)」を参照)。

```
#include <thread.h>

int thr_setprio(thread_t tid, int newprio)
```

スレッドのスケジューリングは、デフォルトの設定では、最低の優先順位を表す 0 から最大整数までの範囲の固定的な優先順位に基づいて行われます。*tid* で指定されたスレッドは、自分より優先順位の低いスレッドから実行リソースを横取りし、自分より優先順位の高いスレッドには実行リソースを譲ります。

```
thread_t tid;
int ret;
int newprio = 20;

/* 停止状態のスレッドを生成する */
ret = thr_create(NULL, NULL, func, arg, THR_SUSPEND, &tid);

/* 停止状態の子スレッドに対して新しい優先順位を設定する */
ret = thr_setprio(tid, newprio);

/* 停止状態の子スレッドを新しい優先順位で開始する */
```

(続く)

```
ret = thr_continue(tid);
```

スレッド優先順位の取得

thr_getprio(3THR)

thr_getprio(3THR) は、スレッドの現在の優先順位を取得します。各スレッドは生成側の優先順位を継承します。thr_getprio() は、*tid* で指定されたスレッドの現在の優先順位を、*newprio* が指している位置に格納します。(POSIX スレッドについては、48ページの「pthread_getschedparam(3THR)」を参照)。

```
#include <thread.h>

int thr_getprio(thread_t tid, int *newprio)
```

pthread に相当するものがある同期関数 — 相互排他ロック

mutex の初期化

mutex_init(3THR)

```
#include <synch.h> (または #include <thread.h>)

int mutex_init(mutex_t *mp, int type, void *arg));
```

mutex_init(3THR) は、*mp* が指す相互排他ロック (mutex ロック) を初期化します。*type* には、次のいずれかを指定できます (*arg* は現在は無視されます)。(POSIX スレッドについては、108ページの「mutex の初期化」を参照)。

- USYNC_PROCESS

  このプロセス内のスレッドと他のプロセス内のスレッドとの間で同期をとることができるようにします。

- USYNC_PROCESS_ROBUST

  このプロセス内のスレッドと他のプロセス内のスレッドとの間で確実に同期をとることができるようにします。

- USYNC_THREAD

  このプロセス内のスレッドの間でだけ同期をとることができるようにします。

USYNC_PROCESS ロックした状態でプロセスが終了すると、次にそのロックを要求したスレッドは滞ります。これは、クライアントプロセスとロックを共有するシステムで起こる問題で、クライアントプロセスが強制的に終了されることがあり得るからです。ロックしたままプロセスが終了する問題を回避するには、USYNC_PROCESS_ROBUST で mutex をロックします。USYNC_PROCESS_ROBUST には次の 2 つの機能があります。

- プロセスが終了するときに、そのプロセスで獲得されたロックをすべて解除します。
- 強制終了されたプロセスが獲得したロックを次に要求するスレッドは、そのロックと共に、エラーを受け取ります。エラーは、前にロックを獲得していたスレッドがロックしたまま終了したことを示します。

0 に初期化されたメモリーに領域を確保することによって mutex を初期化することもできます。その場合は *type* に USYNC_THREAD を指定したものと仮定されます。

複数のスレッドから同じ mutex を同時に初期化してはいけません。一度初期化した mutex は、他のスレッドが使用している可能性があるので再初期化してはいけません。

プロセス内スコープでの *mutex*

```
#include <thread.h>

mutex_t mp;
int ret;

/* このプロセスの中だけで使用する */
ret = mutex_init(&mp, USYNC_THREAD, 0);
```

プロセス間スコープでの *mutex*

```
#include <thread.h>

mutex_t mp;
int ret;

/* すべてのプロセスの間で使用する */
ret = mutex_init(&mp, USYNC_PROCESS, 0);
```

プロセス間スコープの確実な *mutex*

```
#include <thread.h>

mutex_t mp;
int ret;

/* to be used among all processes */
ret = mutex_init(&mp, USYNC_PROCESS_ROBUST, 0);
```

mutex の削除

mutex_destroy(3THR)

```
#include <thread.h>

int mutex_destroy (mutex_t *mp);
```

mutex_destroy(3THR) は、*mp* が指す mutex を削除します。mutex を格納する領域は解放されません。(POSIX スレッドについては、115ページの「pthread_mutex_destroy(3THR)」を参照)

mutex の獲得

mutex_lock(3THR)

```
#include <thread.h>

int mutex_lock(mutex_t *mp);
```

mutex_lock(3THR) は、*mp* が指す mutex をロックします。mutex がすでにロックされている場合は、使用可能になるまで呼び出しスレッドがブロックされます (ブロック状態のスレッドは、優先順位別の待ち行列に入れられます)。(POSIX スレッドについては、110ページの「pthread_mutex_lock(3THR)」を参照)。

mutex の解除

mutex_unlock(3THR)

```
#include <thread.h>

int mutex_unlock(mutex_t *mp);
```

mutex_unlock(3THR) は、*mp* が指す mutex のロックを解除します。mutex はロックされていなければならず、しかも呼び出しスレッドがその mutex を最後にロックした (つまり、現在保持している) スレッドでなければなりません。(POSIX スレッドについては、113ページの「pthread_mutex_unlock(3THR)」を参照)。

mutex の獲得 (ブロックなし)

mutex_trylock(3THR)

```
#include <thread.h>

int mutex_trylock(mutex_t *mp);
```

mutex_trylock(3THR) は、*mp* が指す mutex をロックしようとします。この関数はブロックしない点を除いて、mutex_lock() と同じ働きをします。(POSIX スレッドについては、114ページの「pthread_mutex_trylock(3THR)」を参照)。

pthread に相当するものがある同期関数 — 条件変数

- 255ページの「特定のスレッドのブロック解除」
- 255ページの「全スレッドのブロック解除」

条件変数の初期化

cond_init(3THR)

```
#include <thread.h>

int cond_init(cond_t *cv, int type, int arg);
```

cond_init(3THR) は、*cv* が指す条件変数を初期化します。*type* には、次のいずれかを指定できます (*arg* は現在は無視されます)。(POSIX スレッドについては、123ページの「pthread_condattr_init(3THR)」を参照)。

- USYNC_PROCESS

  現在のプロセス内のスレッドと他のプロセス内のスレッドとの間で同期をとることができるようにします。*arg* は無視されます。

- USYNC_THREAD

  現在のプロセス内のスレッドの間でだけ同期をとることができるようにします。*arg* は無視されます。

0 に初期化されたメモリーに領域を確保することによって、条件変数を初期化することもできます。その場合は、*type* に USYNC_THREAD を指定したものと仮定されます。

複数のスレッドから、同じ条件変数を同時に初期化してはいけません。一度初期化した条件変数は他のスレッドが使用している可能性があるので、再初期化してはいけません。

プロセス内スコープでの条件変数

```
#include <thread.h>

cond_t cv;
int ret;

/* このプロセスの中だけで使用する */
```

```
ret = cond_init(cv, USYNC_THREAD, 0);
```

プロセス間スコープでの条件変数

```
#include <thread.h>

cond_t cv;
int ret;

/* すべてのプロセスの間で使用する */
ret = cond_init(&cv, USYNC_PROCESS, 0);
```

条件変数の削除

cond_destroy(3THR)

```
#include <thread.h>

int cond_destroy(cond_t *cv);
```

cond_destroy(3THR) は、*cv* が指す条件変数を削除します。条件変数を格納する領域は解放されません。(POSIX スレッドについては、124ページの「pthread_condattr_destroy(3THR)」を参照)。

条件変数によるブロック

cond_wait(3THR)

```
#include <thread.h>

int cond_wait(cond_t *cv, mutex_t *mp);
```

cond_wait(3THR) は、*mp* が指す mutex を原子操作により解放し、*cv* が指す条件変数で、呼び出しスレッドをブロックします。ブロックされたスレッドを呼び起こすには、cond_signal() か cond_broadcast() を使います。また、スレッドはシグナルや fork() の割り込みによっても呼び起こされます。(POSIX スレッドについては、129ページの「pthread_cond_wait(3THR)」を参照)。

条件変数による指定時刻付きブロック

cond_timedwait(3THR)

```
#include <thread.h>

int cond_timedwait(cond_t *cv, mutex_t *mp, timestruct_t abstime)
```

cond_timedwait(3THR) は、*abstime* で指定した時刻を過ぎるとブロック状態を解除する点を除いて、cond_wait() と同じ動作をします。(POSIX スレッドについては、132ページの「pthread_cond_timedwait(3THR)」を参照)。

cond_timedwait() が戻るときは、たとえエラーを戻したときでも、常に mutex は呼び出しスレッドがロックし保持している状態にあります。

cond_timedwait() のブロック状態が解除されるのは、条件変数にシグナルが送られてきたときか、一番最後の引数で指定した時刻を過ぎたときです。時間切れの指定は時刻で行うため、時間切れの時刻を再計算する必要がないので、効率的に条件を再評価できます。

特定のスレッドのブロック解除

cond_signal(3THR)

```
#include <thread.h>

int cond_signal(cond_t *cv);
```

cond_signal(3THR) は、*cv* が指す条件変数でブロックされている 1 つのスレッドのブロックを解除します。この関数は、シグナルを送ろうとしている条件変数で使用されたのと同じ相互排他ロックを獲得した状態で呼び出してください。そうしないと、関連する条件が評価されてから cond_wait() でブロック状態に入るまでの間に、条件変数にシグナルが送られる可能性があります。この場合、cond_wait() は永久に待ち続けることになります。

全スレッドのブロック解除

cond_broadcast(3THR)

```
#include <thread.h>

int cond_broadcast(cond_t *cv);
```

cond_broadcast(3THR) は、*cv* が指す条件変数でブロックされている全スレッドのブロックを解除します。スレッドがブロックされていない条件変数に対して cond_broadcast() を実行しても無視されます。

pthread に相当するものがある同期関数 — セマフォ

セマフォの操作は Solaris オペレーティング環境と POSIX 環境の両方で同じです。関数名は、Solaris オペレーティング環境で sema_ だった関数名が pthread では sem_ に変わっています。

セマフォの初期化

sema_init(3THR)

```
#include <thread.h>

int sema_init(sema_t *sp, unsigned int count, int type,
    void *arg);
```

sema_init(3THR) は、*sp* が指すセマフォ変数に *count* の値を初期設定します。*type* には、次のいずれかを指定できます (*arg* は現在は無視されます)。

USYNC_PROCESS: 現在のプロセス内のスレッドと他のプロセス内のスレッドとの間で同期をとることができるようにします。ただし、セマフォを初期化するプロセスは 1 つだけに制限してください。*arg* は無視されます。

USYNC_THREAD: 現在のプロセス内のスレッドの間でだけ同期をとることができるようにします。*arg* は無視されます。

複数のスレッドから同じセマフォを同時に初期化してはいけません。一度初期化したセマフォは他のスレッドが使用している可能性があるので、再初期化してはいけません。

プロセス内スコープでのセマフォ

```
#include <thread.h>

sema_t sp;
int ret;
int count;
count = 4;
```

(続く)

```
/* このプロセスの中だけで使用する */
ret = sema_init(&sp, count, USYNC_THREAD, 0);
```

プロセス間スコープでのセマフォ

```
#include <thread.h>

sema_t sp;
int ret;
int count;
count = 4;

/* すべてのプロセスの間で使用する */
ret = sema_init (&sp, count, USYNC_PROCESS, 0);
```

セマフォの加算

sema_post(3THR)

```
#include <thread.h>

int sema_post(sema_t *sp);
```

sema_post(3THR) は、*sp* が指すセマフォの値を原子操作によって 1 増やします。そのセマフォでブロックされているスレッドがある場合は、そのスレッドのうちの 1 つのスレッドがブロック解除されます。

セマフォの値によるブロック

sema_wait(3THR)

```
#include <thread.h>

int sema_wait(sema_t *sp);
```

sema_wait(3THR) は、*sp* が指すセマフォの値が、0 より大きくなるまでスレッドをブロックし、0 より大きくなったらセマフォの値を原子操作によって 1 減らします。

セマフォの減算

sema_trywait(3THR)

```
#include <thread.h>

int sema_trywait(sema_t *sp);
```

sema_trywait(3THR) は、*sp* が指すセマフォの値が 0 より大きい場合、原子操作によって 1 減らします。この関数はブロックしない点を除いて、sema_wait() と同じ働きをします。

セマフォの削除

sem_destroy(3RT)

```
#include <thread.h>

int sema_destroy(sema_t *sp);
```

sem_destroy(3RT) は、*sp* が指すセマフォを削除します。セマフォを格納する領域は解放されません。

プロセスの境界を越えた同期

今までに説明した 4 種類の同期プリミティブは、プロセスの境界を越えて使用するように設定できます。具体的には次のようにします。まず、その同期変数の領域が共有メモリーに確保されるようにします。次に、それぞれの初期化ルーチン (init) を呼び出すとき、引数 *type* に USYNC_PROCESS を指定します。

以上により、その同期変数に対する操作は、*type* が USYNC_THREAD のときとまったく同じように実行されます。

```
mutex_init(&m, USYNC_PROCESS, 0);

rwlock_init(&rw, USYNC_PROCESS, 0);

cond_init(&cv, USYNC_PROCESS, 0);

sema_init(&s, count, USYNC_PROCESS, 0);
```

プロセス間での LWP の使用

プロセス間でロックと条件変数を使用する場合、必ずしもスレッドライブラリを使用しなければならないわけではありません。基本的にはスレッドライブラリを使用するものの、それが望ましくないときは、_lwp_mutex_* インタフェースと _lwp_cond_* インタフェースを次のようなやり方で使用するというアプローチを使用できます。

1. ロックと条件変数を通常どおり (shmop(2) または mmap(2) を使用して) 共有メモリーに確保します。
2. 新たに割り当てられたオブジェクトを USYNC_PROCESS タイプとして初期化します。この初期化のために使用できるインタフェースはないので (_lwp_mutex_init(2) と _lwp_cond_init(2) は存在しない)、それらのオブジェクトは静的に割り当てて初期化したダミーオブジェクトを使って初期化します。

たとえば、lockp を初期化するには次のようにします。

```
lwp_mutex_t *lwp_lockp;
lwp_mutex_t dummy_shared_mutex = SHAREDMUTEX;
 /* SHAREDMUTEX は /usr/include/synch.h の中で定義されている */
...
...
lwp_lockp = alloc_shared_lock();
*lwp_lockp = dummy_shared_mutex;
```

同様に、条件変数については次のようにします。

```
lwp_cond_t *lwp_condp;
lwp_cond_t dummy_shared_cv = SHAREDCV;
 /* SHAREDCV は /usr/include/synch.h の中で定義されている */
...
...
lwp_condp = alloc_shared_cv();
*lwp_condp = dummy_shared_cv;
```

「生産者 / 消費者」問題の例

例 8–2 では、「生産者 / 消費者」問題の生産者と消費者をそれぞれ別のプロセスで表現しています。メインルーチンは、0 に初期化されたメモリーを自分のアドレス空間にマッピングし、それを子プロセスと共有します。`mutex_init()` と `cond_init()` を呼び出さなければならないのは、それらの同期変数のタイプが `USYNC_PROCESS` だからです。

子プロセスが 1 つ生成され、消費者の処理が実行されます。親プロセスは生産者の処理を実行します。

この例では、生産者と消費者を呼び出す各駆動ルーチンも示しています。`producer_driver()` は `stdin` から文字を読み込み、`producer()` を呼び出します。`consumer_driver()` は `consumer()` を呼び出して文字を受け取り、`stdout` に書き出します。

例 8–2 のデータ構造は、条件変数による「生産者 / 消費者」のコーディング例のデータ構造と同じです (詳細は、119ページの「片方向リンクリストの入れ子のロック」を参照してください)。

例 8–2　「生産者 / 消費者」問題 − USYNC_PROCESS を使った例

```
main() {
    int zfd;
    buffer_t *buffer;

    zfd = open(‘‘/dev/zero’’, O_RDWR);
    buffer = (buffer_t *)mmap(NULL, sizeof(buffer_t),
        PROT_READ|PROT_WRITE, MAP_SHARED, zfd, 0);
    buffer->occupied = buffer->nextin = buffer->nextout = 0;

    mutex_init(&buffer->lock, USYNC_PROCESS, 0);
    cond_init(&buffer->less, USYNC_PROCESS, 0);
    cond_init(&buffer->more, USYNC_PROCESS, 0);
    if (fork() == 0)
        consumer_driver(buffer);
    else
        producer_driver(buffer);
}

void producer_driver(buffer_t *b) {
    int item;

    while (1) {
        item = getchar();
        if (item == EOF) {
            producer(b, ‘\0’);
            break;
        } else
            producer(b, (char)item);
    }
}

void consumer_driver(buffer_t *b) {
    char item;

    while (1) {
        if ((item = consumer(b)) == ’\0’)
            break;
        putchar(item);
    }
}
```

子プロセスが 1 つ生成され、消費者の処理が実行されます。親プロセスは生産者の処理を実行します。

fork() と Solaris スレッドに関する問題

Solaris スレッドと POSIX スレッドでは、fork() の動作に関する定義が異なります。fork() の問題の詳細は、171ページの「プロセスの作成 − exec(2) と exit(2) について」を参照してください。

Solaris libthread は、fork() と fork1() の両方をサポートします。fork() 呼び出しは「汎用 fork」セマンティクスをもち、スレッドと LWP を含むプロセス内のすべてを複製します。つまり、親の完全なクローンを作成します。一方、fork1() 呼び出しで作成されるクローンはスレッドを 1 つしかもちません。プロセスの状態とアドレス空間は複製されますが、スレッドについては呼び出しスレッドが複製されるだけです。

POSIX libpthread は 、fork() のみをサポートします。そのセマンティクスは、Solaris スレッドにおける fork1() と同じです。

fork() のセマンティスクが「汎用 fork」と「fork1」のどちらになるかは、どちらのライブラリを使用するかで決まります。–lthread を使ってリンクすれば「汎用 fork」セマンティクス、–lpthread を使ってリンクすれば「fork1」セマンティクスになります。

詳細は、208ページの「libthread または libpthread とのリンク」を参照してください。

第 9 章

プログラミング上の指針

この章では、スレッドを使ったプログラミングのための指針を示します。ほとんどの内容は Solaris スレッドと POSIX スレッドの両方に当てはまりますが、両者で機能的な違いがある点については、その旨を明記します。この章では、シングルスレッドとマルチスレッドの考え方の違いを中心に説明します。

広域変数の考慮

現状では大半のコード、特に C プログラムから呼び出されるライブラリルーチンは、シングルスレッドアプリケーション向けに設計されています。シングルスレッド用のコードでは、次のように仮定していました。

- 広域変数に書き込んだ内容をしばらくたってから読み取りしても、その内容は以前と同じである。

- 上記のことは、広域的でない静的記憶領域についても成立する。
- 同期をとるべきものがないので、同期は必要ない。

次に、上記の仮定が原因で生じるマルチスレッドプログラム上の問題とその対処方法を示します。

従来のシングルスレッドの C と UNIX では、システムコールで検出されたエラーの扱いに関して一定の決まりがあります。システムコールは、関数値として任意の値を戻すことができます (たとえば、`write()` は転送したバイト数を戻します)。ただし、値 -1 は、エラーが生じたことを示すために予約されています。つまり、システムコールから -1 が戻された場合は、システムコールが失敗したことを意味します。

例 9–1 広域変数と errno

```
extern int errno;
...
if (write(file_desc, buffer, size) == -1) {
    /* システムコールが失敗 */
    fprintf(stderr, ''something went wrong, ''
        ''error code = %d\n'', errno);
    exit(1);
}
...
```

戻り値と混同されがちですが、実際のエラーコードは広域変数 `errno` に格納されます。システムコールが失敗した場合は、`errno` を調べればエラーの内容を知ることができます。

ここで、マルチスレッド環境において 2 つのスレッドが同時に失敗し、異なるエラーが発生したと仮定します。このとき、両方のスレッドがエラーコードは `errno` に入っていると期待しても、1 つの `errno` に両方の値を保持することは不可能です。このように、マルチスレッドプログラムでは、広域変数による方法は使用できません。

スレッドでは、この問題を解決するために、スレッド固有データという新しい記憶クラスを導入しています。このスレッド固有データは、スレッド内の任意の手続きからアクセスできるという点で広域変数と似ています。ただし、これはそのスレッドに専用の領域です。つまり、2 つのスレッドが同じ名前のスレッド固有データをアクセスしても、それぞれ異なる変数をアクセスしていることになります。

したがって、スレッドを使用しているときは、スレッドごとに `errno` の専用のコピーが与えられるので、`errno` の参照がスレッドに固有なものとなります。この実

装においては、errno をマクロにして関数呼び出しを行うことでこれを可能にしています。

静的局所変数の利用

例 9–2 は、前述の errno と同様の問題を示すものです。ただし、ここでは広域的な記憶領域ではなく静的な記憶領域が問題となります。関数 gethostbyname(3N) は、コンピュータ名を引数として与えられて呼び出されます。その戻り値はある構造体のポインタで、その構造体には指定したコンピュータと、ネットワークを通して通信するために必要な情報が入っています。

例 9–2 gethostbyname() の問題

```
struct hostent *gethostbyname(char *name) {
    static struct hostent result;
        /* ホストデータベースから名前を検索 */
        /* result に答えを入れる */
    return(&result);
}
```

一般に、局所変数へのポインタを返すというのはよい方法ではありません。上記の例では、変数が静的なために正常に動作します。しかし、2 つのスレッドが異なるコンピュータ名で同時に関数を呼び出すと、静的記憶領域の衝突が生じます。

静的記憶領域の代わりに前述の errno のように、スレッド固有データを使用するという解決方法も考えられますが、動的記憶割り当てのため処理が重くなります。

このような問題を解決する方法は、gethostbyname() の呼び出し側が結果を戻すための記憶領域を呼び出し時に指定してしまうことです。具体的には、このルーチンに出力引数を 1 つ追加して、呼び出し側から与えます。そのためには、gethostbyname() に新しいインタフェースが必要です。

Solaris スレッドでは、この種の問題の多くを解決するために、上記のテクニックが使われています。通常、新しいインタフェース名は、末尾に「_r」を付けたものです。たとえば、gethostbyname(3N) は、gethostbyname_r(3N) となります。

スレッドの同期

アプリケーション内のスレッドは、データやプロセスリソースを共有するときに相互に同期をとりながら連携して動作しなければなりません。

問題となるのは、ある特定のオブジェクトの操作を複数のスレッドが呼び出すときです。シングルスレッド環境では、そのようなオブジェクトに対するアクセスの同期上の問題は生じませんが、例 9–3 に示すように、マルチスレッドでは注意する必要があります。(Solaris の `printf(3S)` は「MT-安全」です。この例では、`printf()` がマルチスレッドに対応していないと仮定したときに生じる問題を示しています。)

例 9–3 `printf()` の問題

```
/* スレッド 1: */
    printf("go to statement reached");

/* スレッド 2: */
    printf("hello world");


ディスプレイ上の表示:
    go to hello
```

シングルスレッド化

同期上の問題の解決策として、アプリケーション全域で 1 つの相互排他ロック (mutex ロック) を使用するという方法が考えられます。そのアプリケーション内で実行するスレッドは、実行時に必ず mutex をロックし、ブロックされた時に mutex を解除するようにします。このようにすれば、同時に複数のスレッドが共有データをアクセスすることはなくなるので、各スレッドから見たメモリーは整合性を保ちます。

しかし、これは事実上のシングルスレッドプログラムであり、この方法にはほとんど利点がありません。

リエントラント (再入可能)

よりよい方法として、モジュール性とデータのカプセル化の性質の利用があります。複数のスレッドから同時に呼び出されても正しく動作する関数を「リエントラ

ント (再入可能)」関数と呼びます。再入可能な関数を作成するには、その関数にとって何が正しい動作なのかを把握することが必要です。

複数のスレッドから呼び出される可能性のある関数は、再入可能にしなければなりません。そのためには、関数のインタフェースまたは実装方法を変更する必要があります。

リエントラントの問題は、メモリーやファイルなどの広域的な状態におかれているものをアクセスする関数で生じます。それらの関数では、広域的なものをアクセスする場合、スレッドの適当な同期機構で保護する必要があります。

モジュール内の関数をリエントラントにする基本的な方法は、コードをロックする方法とデータをロックする方法の 2 通りがあります。

コードロック

コードロックは関数の呼び出しのレベルで行うロックで、その関数の全体がロックの保護下で実行されることを保証するものです。コードロックが成立するためには、すべてのデータアクセスが関数を通して行われることが前提となります。また、データを共有する関数が複数あるとき、それらを同じロックの保護下で実行することも必要です。

一部の並列プログラミング言語では、モニタという構造が用意されています。モニタは、その対象範囲内に定義されている関数に対して、暗黙の内にコードロックを行います。相互排他ロック (mutex ロック) によって、モニタを実装することも可能です。

同じ相互排他ロックの保護下にある関数または同じモニタの対象範囲内にある関数は、互いに原子操作的に実行されることが保証されます。

データロック

データロックは、データ集合へのアクセスが一貫性をもって行われることを保証します。データロックも、基本的な概念はコードロックと同じです。しかし、コードロックは共有される (広域的な) データのみへの参照を囲むようにかけます。相互排他ロックでは、各データ集合に対応する危険領域を同時に実行できるスレッドはせいぜい 1 つです。

一方、複数読み取り単一書き込みロックでは、それぞれのデータ集合に対して複数スレッドが同時に読み取り操作を行うことができ、1 つのスレッドが書き込み操作を行うことができます。複数読み取り単一書き込みロックのように、それぞれ異な

るデータ集合を操作するか、同じデータ集合で衝突を起こさないようにすれば、同一モジュール内で複数のスレッドを実行できます。つまり、通常はコードロックよりもデータロックの方が、並行度を高くすることができます。(Solaris には読み取り / 書き込みロック機能が組み込まれていることに注意してください。)

プログラムで、(相互排他ロック、条件変数、セマフォなどの) さまざまなロックを使用するときの方針を説明します。できる限り並列性を高めるためにきめ細かくロックする、つまり必要なときだけロックして不要になったらすぐ解除するという方法と、ロックと解除に伴うオーバヘッドをできる限り小さくするため長期間ロックを保持する、つまりきめの粗いロックを行うという方法が考えられます。

ロックをどの程度きめ細かくかけるべきかは、保護の対象となるデータの量によって異なります。最もきめの粗いロックは、全データを 1 つのロックで保護します。保護対象のデータをいくつに分割してロックするかは、非常に重要な問題です。ロックのきめが細かすぎても、性能に悪影響を及ぼします。それぞれのロックと解除に伴うオーバヘッドは、ロックの数が多いと無視できなくなるからです。

通常、ロックを使用する方針は次のとおりです。最初は、きめを粗くロックします。次にボトルネックを特定したら、それが緩和されるようにロックのきめを細かくしていきます。これは妥当な解決策ですが、並列性を最大にすることとロックに伴うオーバヘッドを最小にすることのどちらをどの程度優先させるかは、ユーザが判断してください。

不変式

コードロックとデータロックについて、複雑なロックを制御するためには「不変式」が重要な意味をもちます。不変式とは、常に真である条件または関係のことです。

不変式は、並行実行環境に対して次のように定義されます。すなわち不変式とは、関連するロックが行われるときに条件や関係が真になっていることです。ロックが行われた後は偽になってもかまいません。ただし、ロックを解除する前に真に戻す必要があります。

あるロックが行われるときに真となるような条件または関係も不変式です。条件変数では、条件という不変式を持っていると考えることができます。

例 9–4 `assert(3X)` による不変式のテスト

```
    mutex_lock(&lock);
    while((condition)==FALSE)
```

(続く)

```
        cond_wait(&cv,&lock);
    assert((condition)==TRUE);
      .
      .
      .
    mutex_unlock(&lock);
```

上記の assert() 文は、不変式を評価しています。cond_wait() 関数は、不変式を保存しません。このため、スレッドが戻ったときに不変式をもう一度評価しなければなりません。

他の例は、双方向リンクリストを管理するモジュールです。双方向リンクリストでは、直前の項目の前向きポインタと直後の項目の後ろ向きポインタが同じものを指すという条件が成立します。これは不変式のよい例です。

このモジュールでコードロックを使用するものと仮定し、1 つの広域的な相互排他ロック (mutex ロック) でモジュールを保護することにします。項目を削除したり追加したりするときは相互排他ロックを獲得し、ポインタの変更後に相互排他ロックを解除します。明らかに、この不変式はポインタの変更中のある時点で偽になります。しかし、相互排他ロックを解除する前に真に戻されています。

デッドロックの回避

ある一組のスレッドが一連のリソースの獲得で競合したまま、永久にブロックされた状態に陥っているとき、その状態をデッドロックと呼びます。実行可能なスレッドがあるからといって、デッドロックが発生していないという証拠にはなりません。

代表的なデッドロックは、「自己デッドロック」です (「再帰的なデッドロック」とも言います)。自己デッドロックは、すでに保持しているロックをスレッドがもう一度獲得しようとしたとき発生します。これは、ちょっとしたミスで簡単に発生してしまいます。

たとえば、一連のモジュール関数で構成されるコードモニタを考えます。各モジュール関数が実行中に保持する相互排他ロックがどれも同じであると、このモジュール内の相互排他ロックの保護下にある関数間で呼び出しが行われた場合に、たちまちデッドロックが発生します。また、ある関数がこのモジュールの外部の

コードを呼び出し、そこから再び同じ相互排他ロックで保護されているモジュールを呼び出した場合にもデッドロックが発生します。

この種のデッドロックを回避するには、モジュールの外部の関数を呼び出す場合、その関数が再び元のモジュールを呼び出さないことを確認できないときは各不変式を再び真にして、すべてのモジュール内のロックを解除してから呼び出すようにします。次に、その呼び出しを終了してもう一度ロックを獲得した後、所定の状態を評価して、意図している操作がまだ有効であるか確認します。

もう 1 つ別の種類のデッドロックがあります。スレッド 1、2 がそれぞれ mutex A、B のロックを獲得しているものと仮定します。次に、スレッド 1 が mutex B を、スレッド 2 が mutex A をロックしようとすると、スレッド 1 は mutex B を待ったままブロックされ、スレッド 2 は mutex A を待ったままブロックされます。結局、両方のスレッドは身動きがとれなくなって永久にブロックされ、デッドロック状態となります。

この種のデッドロックを回避するには、ロックを行う順序を一定に保ちます。この方法を「ロック階層」と呼びます。すべてのスレッドが常に一定の順序でロックを行う限り、この種のデッドロックは生じません。

しかし、ロックを行う順序を一定に保つという規則を守っていればよいとは必ずしも言えません。たとえば、スレッド 2 が mutex B を保持している間に、モジュールの状態に関して数多くの仮定条件を立てた場合、次に mutex A のロックを獲得するために mutex B のロックを解除し、規則通り mutex A のロックを獲得した後にもう一度 mutex B のロックを獲得しても先の仮定は無駄になり、モジュールの状態をもう一度評価しなければならなくなります。

通常、ブロックを行う同期プリミティブには、ロックを獲得しようとしてできなかった場合にエラーとなる類似のプリミティブ (たとえば、`mutex_trylock()`) が用意されています。これを使うと、競合がなければロック階層を守らないという方法でロックを実行できます。競合があるときは、通常は保持しているロックをいったん解除してから、順番にロックを実行しなければなりません。

スケジューリングに関するデッドロック

マルチスレッドプログラムでは、ロックが獲得される順序がシステム的に不定であることが原因で、特定のスレッドがロック (通常は条件変数) を獲得できるように見えても、実際にはロックを獲得できないという問題があります。

通常、この問題は次のような状況で起こります。スレッドが、保持していたロックを解除し、少し時間をおいてからもう一度ロックを獲得するものとします。このと

き、ロックはいったん解除されたので、他のスレッドがロックを獲得したと考えがちです。しかし、ロックを保持していたスレッドは、ブロックされなければロック解除後も引き続き実行され、もう一度ロックを獲得するので、結局その間に他のスレッドは実行されません。

通常、この種の問題を解決するには、もう一度ロックを獲得する前に `thr_yield(3T)` を呼び出します。これで他のスレッドが実行され、そのスレッドはロックを獲得できるようになります。

必要なタイムスライスの大きさはアプリケーションに依存するため、スレッドライブラリでは特に制限していません。`thr_yield()` を呼び出して、スレッドが共有する時間を設定してください。

ロックに関する指針

次に、ロックのための簡単な指針を示します。

- ロックを長期間保持しないでください。たとえば、入出力時にロックを保持したままにすると性能が低下することがあります。
- モジュールから外部の関数を呼び出す場合、その関数が元のモジュールを呼び出す可能性があるときはロックを解除してください。
- 一般に、初めは大まかに調べるというやり方で臨み、ボトルネックを見つけます。そして、ボトルネックを軽減するのに必要なら、きめ細かなロックを追加していきます。ロックが保持される時間は通常はそれほど長くなく、競合もめったに起こりません。実際に競合のあったロックだけを調整してください。
- 複数のロックを使用する場合は、デッドロックを回避するために、すべてのスレッドで同じ順序でロックを獲得するようにしてください。

その他の基本的な指針

- 外部から手続きなどを流用する場合、その安全性を確認してください。

  マルチスレッドプログラムから、マルチスレッド化されていないコードをそのまま呼び出すことはできません。
- マルチスレッドプログラムでは、初期スレッドからのみ「MT-安全ではない」コードに安全にアクセスできます。

これは初期スレッドに対応する静的記憶領域が、初期スレッドによってだけ使用されることを保証します。

- Sun から提供されるライブラリは、「安全」であると明記されていなければ、「安全ではない」とみなされます。

  リファレンスマニュアルのエントリにインタフェースが「MT-安全」であると明示的に記載されていない場合は、そのインタフェースは「安全ではない」と考えるべきです。

- コンパイルフラグでソースのバイナリレベルでの非互換性を吸収してください。(詳細は、第 7 章「コンパイルとデバック」を参照してください。)

  - `-D_REENTRANT` を使用すると`-lthread` ライブラリによるマルチスレッドが有効になります。

  - `-D_POSIX_C_SOURCE` と `-lpthread` を使用すると、POSIX スレッドの動作になります。

  - `-D_POSIX_PTHREADS_SEMANTICS` と `-lthread` を使用すると、Solaris スレッドと pthread の両方のインタフェースが有効になりますが、2 つのインタフェースが衝突したときは POSIX インタフェースが優先されます。

- ライブラリを「MT-安全」にする場合、プロセスの広域的な操作はスレッド化しないでください。

  広域的な操作 (または広域的な副作用のある処理) をスレッド化しないでください。たとえば、ファイル入出力をスレッド単位の操作に変更しても、複数のスレッドがファイルに同時にアクセスできません。

  スレッド特有の動きやスレッドとして認識される動きは、スレッド機能を使って実現してください。たとえば、`main()` の終了時に `main()` のスレッドだけを終了したい場合は、`main()` の最後を次のようにします。

```
thr_exit();
 /* NOT REACHED */
```

スレッドの生成と使用

スレッドパッケージは、スレッドのデータ構造、スタック、および LWP をキャッシュするので、非結合スレッドを繰り返し生成してもシステムに対する負荷は大きくなりません。

プロセスや結合スレッドの生成と比べて、非結合スレッドの生成にはかなりのオーバヘッドがあります。実際そのオーバヘッドは、1 つのスレッドを停止して他のスレッドを開始するといったコンテキストスイッチを行う場合の非結合スレッドでの同期を行うのにかかる負荷と同程度です。

したがって、必要に応じてスレッドを生成したり削除したりするほうが、専用の処理要求を待つスレッドを維持管理するより効率的です。

たとえば、RPC サーバがよい例です。RPC サーバは要求が送られてきたらスレッドを生成し、応答を返したらスレッドを削除します。要求を処理するためのスレッドを、常に維持管理しません。

スレッドの生成のオーバヘッドがプロセス生成のオーバヘッドと比べて小さいといっても、数個の命令を実行するのにかかる負荷に比べると効率的ではありません。少なくとも数千の機械語命令が続くような処理を対象にして、スレッドを生成してください。

軽量プロセス (LPW)

図 9–1 に LWP、ユーザレベル、およびカーネルレベルの関係を示します。

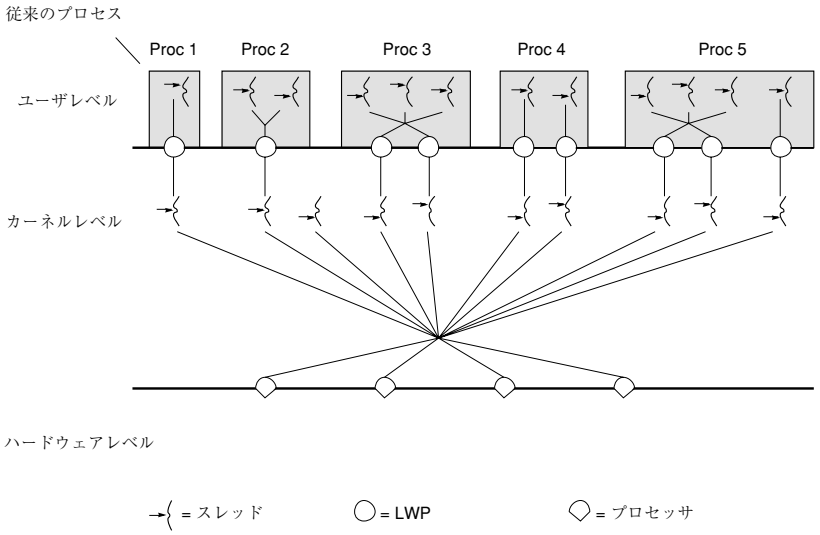

図 9–1　マルチスレッドのレベルと関係

ユーザレベルのスレッドライブラリは、適切なプログラミングが行われていて、オペレーティング環境が正常に動作している限り、現在実行可能なユーザレベルのスレッド数に対して適切な数の利用可能な LWP が存在することを保証しています。しかし、ユーザレベルのスレッドと LWP の間には一対一の関係は存在しないので、ユーザレベルのスレッドはある LWP から別の LWP へと自由に移動できます。

Solaris スレッドでは、プログラマは同時にいくつのスレッドを実行させるかをスレッドライブラリに指定できます。

たとえば、最大 3 個のスレッドを同時に実行させるように指定すると、少なくとも 3 個の LWP が必要になります。3 個のプロセッサが利用可能なら、それらのスレッドは並列に実行されます。しかし、プロセッサが 1 つしか存在しないときは、オペレーティング環境が単一のプロセッサ上で 3 つの LWP を並行化します。すべての LWP がブロックされた場合は、もう 1 つ別の LWP がスレッドライブラリによって実行リソースに追加されます。

同期をとるためにユーザスレッドがブロックすると、そのスレッドが接続している LWP は、実行可能な別のスレッドと接続します。この移行にはコルーチンリンケージが使われ、システムコールは使われません。

オペレーティング環境は、どの LWP をどのプロセッサでいつ実行させるかを決定します。各プロセスにあるユーザスレッドや、ユーザスレッドがいくつ実行可能になっているかなどをオペレーティング環境は認識していません。

カーネルは、LWP のスケジューリングクラスと優先順位に従って、LWP を CPU リソースに割り当てます。同様に、スレッドライブラリはスレッドを LWP に割り当てます。

各 LWP はカーネルによって独立に振り分けられ、独立したシステムコールを実行し、独立したページフォルトを引き起こし、マルチプロセッサのシステム上では並列に動作します。

LWP の機能の中には、特別なスケジューリングクラスなどのように、スレッドからは直接には参照できないものがあります。

非結合スレッド

スレッドライブラリは、必要に応じて LWP を生成し、実行可能なスレッドを LWP に割り当てます。スレッドが割り当てられた LWP はスレッドの状態を引き継ぎ、スレッドの一連の命令を実行します。スレッドが同期機構によりブロック状態になるか、別のスレッドを実行しなければならないような状態が生じると、現在のスレッドの状態はプロセスのメモリーに退避され、スレッドライブラリはその LWP に別のスレッドを割り当てて実行します。

結合スレッド

非結合スレッドで起こりがちなことですが、スレッド数を LWP 数よりも大きくすると不利な場合があります。

たとえば、行列を並列に計算するために、行列の行を複数のスレッドに振り分ける場合を考えます。各プロセッサに 1 つの LWP が存在するが各 LWP に複数のスレッドを割り当てる場合は、各プロセッサの時間がスレッドの切り替えのために費やされます。このような場合は、各 LWP には単一のスレッドを割り当てて、行列の行を振り分けるスレッドの数を減らし、スレッドを切り替える回数を減らすほうが効果的です。

LWP に固定的に結合されたスレッドと非結合スレッドを混在させると都合がよい場合もあります。

たとえば、リアルタイムアプリケーションでは、一部のスレッドにシステム全体での優先順位を与えてリアルタイムでスケジューリングし、他のスレッドにバックグラウンドで計算を実行させます。もう 1 つの例はウィンドウシステムです。ウィンドウシステムでは大半の処理を非結合スレッドで実行し、マウスに関する処理を優先順位の高いリアルタイムの結合スレッドで実行します。

ユーザレベルのスレッドがシステムコールを発行すると、そのスレッドを実行している LWP はカーネル内部に入り、少なくともシステムコールが完了するまでの間はスレッドに接続されたままの状態となります。

結合スレッドは、非結合スレッドより負荷がかかります。結合スレッドは、結合している LWP の属性を変更することがあるので、終了時に LWP がキャッシュされることはありません。オペレーティング環境は、結合スレッドに対して生成時に新しい LWP を与え、終了時にその LWP を削除します。

結合スレッドを使用するのは、次の場合に限ってください。すなわち、結合している LWP を通してのみ利用可能なリソース (たとえば、仮想時間インタバルタイマ、代替スタック) をスレッドが必要としている場合か、スレッドをカーネルから参照可能にしてシステム内のすべての実行可能なスレッドとの関係でスケジューリングされるようにする (たとえば、リアルタイムスケジューリング) 場合です。

すべてのスレッドが同時に実行可能になることが期待される場合は、非結合スレッドを使用してください。そうすれば LWP とスレッドのリソースが効率的にキャッシュされるので、スレッドの生成と削除が高速で行われるようになります。`thr_setconcurrency(3T)` を使用すると、Solaris スレッドに対して同時に有効にしたいスレッド数 (目標値) を伝えることができます。

スレッドの並行度 (Solaris スレッドの場合のみ)

特に指定しなければ Solaris のスレッドは、非結合スレッドを実行するためのシステム実行リソース (LWP) の数を実行可能なスレッドの数と同じになるように調整します。この調整は完全なものではありませんが、少なくともプロセスの処理が進行することは保証されます。

同時に実行可能にすべき (コードやシステムコールを実行する) 非結合スレッドの数がわかっている場合は、`thr_setconcurrency(3THR)` によって、その値をスレッドライブラリに指示してください。

例

- ユーザごとにスレッドが必要なデータベースサーバでは、同時にアクセスするユーザ数をスレッド並行度として指定します。
- クライアントごとにスレッドが必要なウィンドウサーバでは、同時に実行されるクライアント数をスレッド並行度として指定します。
- 読み取りスレッドと書き込みスレッドが 1 つずつ必要なファイルコピープログラムでは、2 をスレッド並行度として指定します。

各スレッドの生成時に THR_NEW_LWP フラグを指定して、並行度を 1 つ増やす方法もあります。

スレッドの並行度を計算するときは、プロセス間 (USYNC_PROCESS) 同期変数でブロックされている非同期スレッドも、実行可能なスレッドとして数えてください。結合スレッドは LWP と等価で、Solaris スレッドの並行度のサポートを必要としないので、実行可能なスレッドには数えません。

効率

すでにあるスレッドを再起動するよりも、thr_create(3T) で新しく生成した方が短時間ですみます。つまり、使用しないスレッドをそのまま残しておいて後で再起動するより、必要に応じて新しいスレッドを生成し使い終わったら thr_exit(3T) で終了させる方が効率的です。

スレッドの生成に関する指針

次に、スレッドを使用するときの簡単な指針を示します。

- 十分な仕事量をもつ独立した活動にスレッドを使用してください。
- CPU の並行度を活用したいときにスレッドを使用してください。
- 結合スレッドは、どうしても必要なときだけ使用してください。つまり、結合する LWP の機能が必要なときだけ使用してください。

マルチプロセッサへの対応

マルチスレッドでは、主に並列性とスケーラビリティという点でマルチプロセッサを活用できます。プログラマは、マルチプロセッサと単一プロセッサのメモリーモデルの違いを考慮に入れておかなければなりません。

メモリーの一貫性は、メモリーを問い合わせるプロセッサと直接的な相関関係にあります。単一プロセッサの場合、メモリーを参照するプロセッサは 1 つしかないのでメモリーは一貫しています。

マルチプロセッサの性能を高めようとすると、メモリーの一貫性が緩められることになります。あるプロセッサによるメモリーへの変更が、他のプロセッサから見たメモリーイメージにただちに反映されるとは限りません。

共有される広域変数を使用するときに同期変数を使用すれば、この複雑さを回避できます。

バリア同期を使用すると、マルチプロセッサ上での並列性をうまく制御できる場合があります。付録 B「Solaris スレッドの例 − barrier.c」にバリアの一例を示しています。

マルチプロセッサに関して、もう 1 つの問題があります。共通の実行ポイントに到達するまで全スレッドが待たなければならないようなケースでは、同期の効率が問題となります。

注 - 共有メモリーにアクセスするためにスレッドの同期プリミティブを必ず使用する場合は、上記の項目は重要ではありません。

アーキテクチャ

スレッドが、Solaris のスレッド同期ルーチンを使用して共有記憶領域へのアクセスの同期をとるときは、共有メモリー型のマルチプロセッサ上でプログラムを実行することと、単一プロセッサ上でプログラムを実行することは同じことになります。

しかし、プログラマはあえてマルチプロセッサ特有の機能を活用したり、同期ルーチンを迂回する「トリック」を使用したりすることがあります。例 9–5 と例 9–6では、そうしたトリックの危険性を示しています。

通常のマルチプロセッサアーキテクチャがサポートしているメモリーモデルを理解することは、この危険性を理解する助けとなります。

マルチプロセッサの主な構成要素は、次のとおりです。

- プロセッサ本体
- ストアバッファ (プロセッサとキャッシュを接続する)
- キャッシュ (最近アクセスされたまたは変更された記憶領域の内容を保持する)
- メモリー (全プロセッサによって共有される主記憶領域)

従来の単純なモデルでは、各プロセッサがメモリーに直接接続されているかのように動作します。つまり、あるプロセッサが特定の位置にデータを格納すると同時に別のプロセッサが同じ位置からデータをロードした場合、2 番目のプロセッサは最初のプロセッサが格納したデータをロードします。

キャッシュは、メモリーアクセスの高速化のために使われ、キャッシュ間の整合性が維持されているときは、データの整合性も保たれます。

この単純なモデルの問題点は、データの整合性を保つためプロセッサをしばしば遅延させなければならないことです。最新のマルチプロセッサでは、各種の手法でそうした遅延を回避していますが、メモリーデータの整合性を失わせています。

次の 2 つの例で、それらの手法と効果を説明します。

共有メモリー型のマルチプロセッサ

例 9–5 は、「生産者 / 消費者」問題の代表的な解決方法です。

このプログラムは、現在の SPARC ベースのマルチプロセッサでは正しく動作しますが、すべてのマルチプロセッサが強く順序付けられたメモリーをもつことを想定しています。したがって、このプログラムには移植性がありません。

例 9–5 「生産者 / 消費者」問題 — 共有メモリー型のマルチプロセッサ

```
                    char buffer[BSIZE];
                    unsigned int in = 0;
                    unsigned int out = 0;

void                                char
producer(char item) {                 consumer(void) {
                                       char item;
    do
        ;/* 処理なし */                   do
    while                                   ;/* 処理なし */
        (in - out == BSIZE);                while
                                              (in - out == 0);
    buffer[in%BSIZE] = item;                 item = buffer[out%BSIZE];
    in++;                                   out++;
}                                   }
```

このプログラムは、生産者と消費者がそれぞれ 1 つしか存在せず、かつ共有メモリー型のマルチプロセッサ上で動作するときは正しく動作します。*in* と *out* の差が、バッファ内のデータ数となります。

生産者はバッファに空きができるまで、この差を繰り返し計算しながら待ちます。消費者は、バッファにデータが入れられるのを待ちます。

強く順序付けられたメモリー (たとえば、あるプロセッサのメモリーへの変更が他のプロセッサにただちに伝わるようなメモリー) では、この方法は成立します (BSIZEが 1 ワードで表現できる最大整数より小さい限り、*in* と *out* が最終的にオーバーフローしても成立します) 。

共有メモリー型のプロセッサは、必ずしも強く順序付けられたメモリーをもつ必要はありません。つまり、あるプロセッサによるメモリーへの変更が、他のプロセッサにただちに伝わるとは限りません。あるプロセッサによって、メモリーに 2 つの変更が加えられた場合、メモリーの変更がただちに伝わらないので、他のプロセッサから検出できる変更の順序は最初の順序と同じであるとは限りません。

変更内容は、まず「ストアバッファ」に入れられます。このストアバッファは、キャッシュからは参照できません。

プロセッサは、データの整合性を保証するためにストアバッファをチェックします。しかし他のプロセッサから、このストアバッファは参照できません。つまり、あるプロセッサが書き込んだ内容は、キャッシュに書き込まれるまで他のプロセッサから参照できません。

同期プリミティブ (第 4 章「同時オブジェクトを使ったプログラミング」を参照) は、特別な命令でストアバッファの内容をキャッシュにフラッシュしています。したがって、共有データをロックで保護すればメモリーの整合性が保証されます。

メモリーの順序付けが非常に弱い場合は、例 9–5では問題が生じます。消費者は、生産者によって *in* が 1 つ増やされたことを、対応するバッファスロットへの変更を知る前に知る場合があるからです。

あるプロセッサのストア順序が、別のプロセッサからは違った順序で見えることがあるため、これを「弱い順序付け」と呼びます (ただし、同じプロセッサから見たメモリーは常に整合性を保っています)。この問題を解決するには、相互排他ロックを使用して、ストアバッファの内容をキャッシュにフラッシュしなければなりません。

最近は、メモリーの順序付けが弱くされる傾向にあります。このため、プログラマは広域データや共有データをロックで保護することに一層注意してください。

例 9–5 と例 9–6 で示すようにロックは重要です。

Peterson のアルゴリズム

例 9–6 は、Peterson のアルゴリズムの実装例です。これは 2 つのスレッド間での相互排他を扱うアルゴリズムです。このコードでは、危険領域に同時に複数のスレッドが存在しないことを保証しようとしています。さらに、スレッドが `mut_excl()` を呼び出すと、危険領域に「素早く」入ることを保証しています。

ここで、スレッドは危険領域に入ると素早く抜け出るものとします。

例 **9–6**　2 つのスレッド間での相互排他が成立するか

```
void mut_excl(int me /* 0 または 1 */) {
    static int loser;
    static int interested[2] = {0, 0};
    int other; /* 局所変数 */

    other = 1 - me;
    interested[me] = 1;
    loser = me;
    while (loser == me && interested[other]);

    /* 危険領域 */
    interested[me] = 0;
}
```

このアルゴリズムは、マルチプロセッサのメモリーが強く順序付けられているときは成立します。

ストアバッファを装備したマルチプロセッサでは、(一部の SPARC ベースのマルチプロセッサも装備しています)、スレッドがストア命令を実行すると、データがストアバッファに入れられます。このバッファの内容は最終的にキャッシュに送られますが、すぐに送られるとは限りません。(各プロセッサのキャッシュはデータの整合性を維持していますが、変更されたデータはキャッシュにすぐには送られません。)

複数のデータが格納されたとき、その変更はキャッシュ (およびメモリー) に正しい順序で伝わりますが、通常は遅延を伴います。SPARC ベースのマルチプロセッサでは、この性質のことを「トータルストア順序 (TSO) をもつ」と言います。

あるプロセッサが A 番地にデータを格納して次に B 番地からデータをロードして、別のプロセッサが B 番地にデータを格納して次に A 番地からデータをロードした場合、「最初のプロセッサが B 番地の新しい値を得る」と「2 番目のプロセッサが A 番地の新しい値を得る」の一方または両方が成立し、かつ「両方のプロセッサが以前の値を得る」というケースは起こりえないはずです。

さらに、ロードバッファとストアバッファの遅延が原因で、上記の起こりえないケースが起こることがあります。

このとき Peterson のアルゴリズムでは、それぞれ別のプロセッサで実行されている 2 つのスレッドが特定の配列の自分のスロットにデータを格納し、別のスロットからデータをロードしています。両方のスレッドは以前の値 (0) を読み取り、相手がいないものと判定し、両方が危険領域に入ってしまいます。(この種の問題は、プログラムのテスト段階では発生せず、後になって発生することがあるので注意してください。)

この問題は、スレッドの同期プリミティブを使用すると回避できます。それらのプリミティブには、ストアバッファをキャッシュに強制的にフラッシュする特別な命令が含まれているからです。

共有メモリー型の並列コンピュータでのループの並列化

多くのアプリケーション、特に数値計算関係のアプリケーションでは、他の部分が本質的に逐次的であっても、`while` 部分のアルゴリズムを並列化できます (詳細は、次の例を参照してください)。

```
スレッド1                                  スレッド2 ～ スレッド n

while(many_iterations) {                   while(many_iterations) {

   sequential_computation
   --- バリア ---                              --- バリア ---
   parallel_computation                       parallel_computation
}                                          }
```

たとえば、完全な線型計算で一組の行列を作成し、それらの行列に対する操作を並列アルゴリズムで実行し、操作結果からもう一組の行列を作成し、それらの行列を並列的に操作するといった処理が考えられます。

こうした計算の並列アルゴリズムの特徴は、計算中はほとんど同期をとる必要はありませんが、並列計算を始める前に逐次計算が終了していることを確認するために、関連するすべてのスレッドの同期をとる必要があることです。

バリアには、並列計算を行なっているすべてのスレッドを、関係しているすべてのスレッドがバリアに達するまで待たせるという働きがあります。スレッドは全部がバリアに達したところで解放され、一斉に計算を開始します。

まとめ

このマニュアルでは、スレッドのプログラミングに関する重要な問題を幅広く取り上げて説明しました。付録 A「アプリケーションの例 − マルチスレッド化された grep」には、pthread プログラムの例が記載されています。この例の中で、今までに説明した多くの機能やスタイルが使用されています。また、付録 B「Solaris スレッドの例」には、Solaris スレッドを使用したプログラムの例が記載されています。

参考資料

マルチスレッドについてさらに詳しく知りたい方は、次の書籍をお読みください。

- 『*Programming with Threads*』(Steve Kleiman、Devang Shah、Bart Smaalders 共著、Prentice-Hall 発行、1995年)

付録 **A**

アプリケーションの例 ー マルチスレッド化された grep

tgrep の説明

サンプルプログラム tgrep は、grep のマルチスレッドバージョンで、grep(1) と find(1) を組み合わせたものです。tgrep は元の grep のオプションを、–w (単語検索) 以外はすべてサポートします。さらに、独自のオプションもあります。

デフォルトでは、tgrep の検索は次のコマンドで実行します。

```
find . -exec grep [ options ] pattern {} \;
```

大きなディレクトリ構造に対して使用した場合、tgrep は find よりも素早く結果を戻すことができます。ただし、そのスピードは使用できるプロセッサの数に左右されます。単一プロセッサのマシンでは約 2 倍のスピードが得られ、4 個のプロセッサを搭載したマシンでは約 4 倍のスピードが得られます。

–e オプションは、tgrep によるパターン文字列の解釈を変更します。–e オプションを指定しなければ、単純な (リテラルな) 文字列検索が行われます。–e オプションを指定すると、スレッドに対して安全なパブリックドメインバージョンの正規表現ハンドラが使用されます。この正規表現を使用した場合は、検索が遅くなります。

–B オプションは、TGLIMIT という環境変数の値によって検索に使用するスレッドの数を制限するよう tgrep に指示します。TGLIMIT が設定されていない場合は、このオプションは機能しません。tgrep では多くのシステムリソースが使われる可

能性があるので、タイムシェアリングシステムで使用する場合は、このオプションを指定します。

オンラインソースコードの入手方法

tgrep のソースコードは、Catalyst Developer's CD に含まれています。コピーの入手方法については、ご購入先にお問い合わせください。

次に、マルチスレッド main.c モジュールだけを示します。その他のモジュール(正規表現ハンドラなど)、マニュアル、および Makefile も Catalyst Developer's CD で入手できます。

例 A–1 tgrep プログラムのソースコード

```
/* Copyright (c) 1993, 1994  Ron Winacott                                */
/* This program may be used, copied, modified, and redistributed freely */
/* for ANY purpose, so long as this notice remains intact.              */

#define _REENTRANT

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <errno.h>
#include <ctype.h>
#include <sys/types.h>
#include <time.h>
#include <sys/stat.h>
#include <dirent.h>

#include "version.h"

#include <fcntl.h>
#include <sys/uio.h>
#include <pthread.h>
#include <sched.h>

#ifdef MARK
#include <prof.h> /* to turn on MARK(), use -DMARK to compile (see man prof5)*/
#endif

#include "pmatch.h"

#define PATH_MAX                1024 /* max # of characters in a path name */
#define HOLD_FDS                6  /* stdin,out,err and a buffer */
#define UNLIMITED               99999 /* The default tglimit */
#define MAXREGEXP               10  /* max number of -e options */

#define FB_BLOCK                0x00001
```

```
#define FC_COUNT                0x00002
#define FH_HOLDNAME             0x00004
#define FI_IGNCASE              0x00008
#define FL_NAMEONLY             0x00010
#define FN_NUMBER               0x00020
#define FS_NOERROR              0x00040
#define FV_REVERSE              0x00080
#define FW_WORD                 0x00100
#define FR_RECUR                0x00200
#define FU_UNSORT               0x00400
#define FX_STDIN                0x00800
#define TG_BATCH                0x01000
#define TG_FILEPAT              0x02000
#define FE_REGEXP               0x04000
#define FS_STATS                0x08000
#define FC_LINE                 0x10000
#define TG_PROGRESS             0x20000

#define FILET                   1
#define DIRT                    2

typedef struct work_st {
    char                *path;
    int                 tp;
    struct work_st      *next;
} work_t;

typedef struct out_st {
    char                *line;
    int                 line_count;
    long                byte_count;
    struct out_st       *next;
} out_t;

#define ALPHASIZ        128
typedef struct bm_pattern {     /* Boyer - Moore pattern                */
        short            p_m;           /* length of pattern string     */
        short            p_r[ALPHASIZ]; /* "r" vector                   */
        short           *p_R;           /* "R" vector                   */
        char            *p_pat;         /* pattern string               */
} BM_PATTERN;

/* bmpmatch.c */
extern BM_PATTERN *bm_makepat(char *p);
extern char *bm_pmatch(BM_PATTERN *pat, register char *s);
extern void bm_freepat(BM_PATTERN *pattern);
BM_PATTERN      *bm_pat;  /* the global target read only after main */

/* pmatch.c */
extern char *pmatch(register PATTERN *pattern, register char *string, int *len);
extern PATTERN *makepat(char *string, char *metas);
extern void freepat(register PATTERN *pat);
extern void printpat(PATTERN *pat);
PATTERN         *pm_pat[MAXREGEXP];  /* global targets read only for pmatch */

#include "proto.h"  /* function prototypes of main.c */

/* local functions to POSIX only */
void pthread_setconcurrency_np(int con);
int pthread_getconcurrency_np(void);
```

```
void pthread_yield_np(void);

pthread_attr_t  detached_attr;
pthread_mutex_t output_print_lk;
pthread_mutex_t global_count_lk;

int             global_count = 0;

work_t          *work_q = NULL;
pthread_cond_t  work_q_cv;
pthread_mutex_t work_q_lk;
pthread_mutex_t debug_lock;

#include "debug.h"  /* must be included AFTER the
                       mutex_t debug_lock line */

work_t          *search_q = NULL;
pthread_mutex_t search_q_lk;
pthread_cond_t  search_q_cv;
int             search_pool_cnt = 0;    /* the count in the pool now */
int             search_thr_limit = 0;   /* the max in the pool */

work_t          *cascade_q = NULL;
pthread_mutex_t cascade_q_lk;
pthread_cond_t  cascade_q_cv;
int             cascade_pool_cnt = 0;
int             cascade_thr_limit = 0;

int             running = 0;
pthread_mutex_t running_lk;

pthread_mutex_t stat_lk;
time_t          st_start = 0;
int             st_dir_search = 0;
int             st_file_search = 0;
int             st_line_search = 0;
int             st_cascade = 0;
int             st_cascade_pool = 0;
int             st_cascade_destroy = 0;
int             st_search = 0;
int             st_pool = 0;
int             st_maxrun = 0;
int             st_worknull = 0;
int             st_workfds = 0;
int             st_worklimit = 0;
int             st_destroy = 0;

int             all_done = 0;
int             work_cnt = 0;
int             current_open_files = 0;
int             tglimit = UNLIMITED;    /* if -B limit the number of
                                           threads */
int             progress_offset = 1;
int             progress = 0;  /* protected by the print_lock ! */
unsigned int    flags = 0;
int             regexp_cnt = 0;
char            *string[MAXREGEXP];
int             debug = 0;
int             use_pmatch = 0;
char            file_pat[255];  /* file patten match */
```

```
PATTERN         *pm_file_pat; /* compiled file target string (pmatch()) */

/*
 * Main: This is where the fun starts
 */
int
main(int argc, char **argv)
{
    int         c,out_thr_flags;
    long        max_open_files = 0l, ncpus = 0l;
    extern int  optind;
    extern char *optarg;
    int         prio = 0;
    struct stat sbuf;
    pthread_t tid,dtid;
    void        *status;
    char        *e = NULL, *d = NULL; /* for debug flags */
    int         debug_file = 0;
    struct sigaction sigact;
    sigset_t    set,oset;
    int         err = 0, i = 0, pm_file_len = 0;
    work_t      *work;
    int         restart_cnt = 10;

    /* NO OTHER THREADS ARE RUNNING */
    flags = FR_RECUR;  /* the default */

    while ((c = getopt(argc, argv, "d:e:bchilnsvwruf:p:BCSZzHP:")) != EOF) {
        switch (c) {
#ifdef DEBUG
        case 'd':
            debug = atoi(optarg);
            if (debug == 0)
                debug_usage();

            d = optarg;
            fprintf(stderr,"tgrep: Debug on at level(s) ");
            while (*d) {
                for (i=0; i<9; i++)
                    if (debug_set[i].level == *d) {
                        debug_levels |= debug_set[i].flag;
                        fprintf(stderr,"%c ",debug_set[i].level);
                        break;
                    }
                d++;
            }
            fprintf(stderr,"\n");
            break;
        case 'f': debug_file = atoi(optarg); break;
#endif      /* DEBUG */

        case 'B':
            flags |= TG_BATCH;
#ifndef __lock_lint
        /* locklint complains here, but there are no other threads */
            if ((e = getenv("TGLIMIT"))) {
                tglimit = atoi(e);
            }
            else {
                if (!(flags & FS_NOERROR))  /* order dependent! */
```

```
                    fprintf(stderr,"env TGLIMIT not set, overriding -B\n");
                flags &= ~TG_BATCH;
            }
#endif
            break;
        case 'p':
            flags |= TG_FILEPAT;
            strcpy(file_pat,optarg);
            pm_file_pat = makepat(file_pat,NULL);
            break;
        case 'P':
            flags |= TG_PROGRESS;
            progress_offset = atoi(optarg);
            break;
        case 'S': flags |= FS_STATS;    break;
        case 'b': flags |= FB_BLOCK;    break;
        case 'c': flags |= FC_COUNT;    break;
        case 'h': flags |= FH_HOLDNAME; break;
        case 'i': flags |= FI_IGNCASE;  break;
        case 'l': flags |= FL_NAMEONLY; break;
        case 'n': flags |= FN_NUMBER;   break;
        case 's': flags |= FS_NOERROR;  break;
        case 'v': flags |= FV_REVERSE;  break;
        case 'w': flags |= FW_WORD;     break;
        case 'r': flags &= ~FR_RECUR;   break;
        case 'C': flags |= FC_LINE;     break;
        case 'e':
            if (regexp_cnt == MAXREGEXP) {
                fprintf(stderr,"Max number of regexp's (%d) exceeded!\n",
                        MAXREGEXP);
                exit(1);
            }
            flags |= FE_REGEXP;
            if ((string[regexp_cnt] =(char *)malloc(strlen(optarg)+1))==NULL){
                fprintf(stderr,"tgrep: No space for search string(s)\n");
                exit(1);
            }
            memset(string[regexp_cnt],0,strlen(optarg)+1);
            strcpy(string[regexp_cnt],optarg);
            regexp_cnt++;
            break;
        case 'z':
        case 'Z': regexp_usage();
            break;
        case 'H':
        case '?':
        default : usage();
        }
    }
    if (flags & FS_STATS)
        st_start = time(NULL);

    if (!(flags & FE_REGEXP)) {
       if (argc - optind < 1) {
            fprintf(stderr,"tgrep: Must supply a search string(s) "
                    "and file list or directory\n");
            usage();
        }
        if ((string[0]=(char *)malloc(strlen(argv[optind])+1))==NULL){
            fprintf(stderr,"tgrep: No space for search string(s)\n");
```

```
        exit(1);
    }
    memset(string[0],0,strlen(argv[optind])+1);
    strcpy(string[0],argv[optind]);
    regexp_cnt=1;
    optind++;
}

if (flags & FI_IGNCASE)
    for (i=0; i<regexp_cnt; i++)
        uncase(string[i]);

if (flags & FE_REGEXP) {
    for (i=0; i<regexp_cnt; i++)
        pm_pat[i] = makepat(string[i],NULL);
    use_pmatch = 1;
}
else {
    bm_pat = bm_makepat(string[0]); /* only one allowed */
}

flags |= FX_STDIN;


max_open_files = sysconf(_SC_OPEN_MAX);
ncpus = sysconf(_SC_NPROCESSORS_ONLN);
if ((max_open_files - HOLD_FDS - debug_file) < 1) {
    fprintf(stderr,"tgrep: You MUST have at least ONE fd "
            "that can be used, check limit (>10)\n");
    exit(1);
}
search_thr_limit = max_open_files - HOLD_FDS - debug_file;
cascade_thr_limit = search_thr_limit / 2;
/* the number of files that can be open */
current_open_files = search_thr_limit;

pthread_attr_init(&detached_attr);
pthread_attr_setdetachstate(&detached_attr,
    PTHREAD_CREATE_DETACHED);

pthread_mutex_init(&global_count_lk,NULL);
pthread_mutex_init(&output_print_lk,NULL);
pthread_mutex_init(&work_q_lk,NULL);
pthread_mutex_init(&running_lk,NULL);
pthread_cond_init(&work_q_cv,NULL);
pthread_mutex_init(&search_q_lk,NULL);
pthread_cond_init(&search_q_cv,NULL);
pthread_mutex_init(&cascade_q_lk,NULL);
pthread_cond_init(&cascade_q_cv,NULL);

if ((argc == optind) && ((flags & TG_FILEPAT) || (flags & FR_RECUR))) {
    add_work(".",DIRT);
    flags = (flags & ~FX_STDIN);
}
for ( ; optind < argc; optind++) {
    restart_cnt = 10;
    flags = (flags & ~FX_STDIN);
  STAT_AGAIN:
    if (stat(argv[optind], &sbuf)) {
        if (errno == EINTR) { /* try again !, restart */
```

```
            if (--restart_cnt)
                goto STAT_AGAIN;
        }
        if (!(flags & FS_NOERROR))
            fprintf(stderr,"tgrep: Can't stat file/dir %s, %s\n",
                    argv[optind], strerror(errno));
        continue;
    }
    switch (sbuf.st_mode & S_IFMT) {
    case S_IFREG :
        if (flags & TG_FILEPAT) {
            if (pmatch(pm_file_pat, argv[optind], &pm_file_len))
                DP(DLEVEL1,("File pat match %s\n",argv[optind]));
                add_work(argv[optind],FILET);
        }
        else {
            add_work(argv[optind],FILET);
        }
        break;
    case S_IFDIR :
        if (flags & FR_RECUR) {
            add_work(argv[optind],DIRT);
        }
        else {
            if (!(flags & FS_NOERROR))
                fprintf(stderr,"tgrep: Can't search directory %s, "
                        "-r option is on. Directory ignored.\n",
                        argv[optind]);
        }
        break;
    }
}

pthread_setconcurrency_np(3);

if (flags & FX_STDIN) {
    fprintf(stderr,"tgrep: stdin option is not coded at this time\n");
    exit(0);                        /* XXX Need to fix this SOON */
    search_thr(NULL);
    if (flags & FC_COUNT) {
        pthread_mutex_lock(&global_count_lk);
        printf("%d\n",global_count);
        pthread_mutex_unlock(&global_count_lk);
    }
    if (flags & FS_STATS)
        prnt_stats();
    exit(0);
}

pthread_mutex_lock(&work_q_lk);
if (!work_q) {
    if (!(flags & FS_NOERROR))
        fprintf(stderr,"tgrep: No files to search.\n");
    exit(0);
}
pthread_mutex_unlock(&work_q_lk);

DP(DLEVEL1,("Starting to loop through the work_q for work\n"));

/* OTHER THREADS ARE RUNNING */
```

```
while (1) {
    pthread_mutex_lock(&work_q_lk);
    while ((work_q == NULL || current_open_files == 0 || tglimit <= 0) &&
           all_done == 0) {
        if (flags & FS_STATS) {
            pthread_mutex_lock(&stat_lk);
            if (work_q == NULL)
                st_worknull++;
            if (current_open_files == 0)
                st_workfds++;
            if (tglimit <= 0)
                st_worklimit++;
            pthread_mutex_unlock(&stat_lk);
        }
        pthread_cond_wait(&work_q_cv,&work_q_lk);
    }
    if (all_done != 0) {
        pthread_mutex_unlock(&work_q_lk);
        DP(DLEVEL1,("All_done was set to TRUE\n"));
        goto OUT;
    }
    work = work_q;
    work_q = work->next;  /* maybe NULL */
    work->next = NULL;
    current_open_files--;
    pthread_mutex_unlock(&work_q_lk);

    tid = 0;
    switch (work->tp) {
    case DIRT:
        pthread_mutex_lock(&cascade_q_lk);
        if (cascade_pool_cnt) {
            if (flags & FS_STATS) {
                pthread_mutex_lock(&stat_lk);
                st_cascade_pool++;
                pthread_mutex_unlock(&stat_lk);
            }
            work->next = cascade_q;
            cascade_q = work;
            pthread_cond_signal(&cascade_q_cv);
            pthread_mutex_unlock(&cascade_q_lk);
            DP(DLEVEL2,("Sent work to cascade pool thread\n"));
        }
        else {
            pthread_mutex_unlock(&cascade_q_lk);
            err = pthread_create(&tid,&detached_attr,cascade,(void *)work);
            DP(DLEVEL2,("Sent work to new cascade thread\n"));
            if (flags & FS_STATS) {
                pthread_mutex_lock(&stat_lk);
                st_cascade++;
                pthread_mutex_unlock(&stat_lk);
            }
        }
        break;
    case FILET:
        pthread_mutex_lock(&search_q_lk);
        if (search_pool_cnt) {
            if (flags & FS_STATS) {
                pthread_mutex_lock(&stat_lk);
                st_pool++;
```

```
                    pthread_mutex_unlock(&stat_lk);
                }
                work->next = search_q;  /* could be null */
                search_q = work;
                pthread_cond_signal(&search_q_cv);
                pthread_mutex_unlock(&search_q_lk);
                DP(DLEVEL2,("Sent work to search pool thread\n"));
            }
            else {
                pthread_mutex_unlock(&search_q_lk);
                err = pthread_create(&tid,&detached_attr,
                                     search_thr,(void *)work);
                pthread_setconcurrency_np(pthread_getconcurrency_np()+1);
                DP(DLEVEL2,("Sent work to new search thread\n"));
                if (flags & FS_STATS) {
                    pthread_mutex_lock(&stat_lk);
                    st_search++;
                    pthread_mutex_unlock(&stat_lk);
                }
            }
            break;
        default:
            fprintf(stderr,"tgrep: Internal error, work_t->tp not valid\n");
            exit(1);
        }
        if (err) {  /* NEED TO FIX THIS CODE. Exiting is just wrong */
            fprintf(stderr,"Could not create new thread!\n");
            exit(1);
        }
    }

 OUT:
    if (flags & TG_PROGRESS) {
        if (progress)
            fprintf(stderr,".\n");
        else
            fprintf(stderr,"\n");
    }
    /* we are done, print the stuff. All other threads are parked */
    if (flags & FC_COUNT) {
        pthread_mutex_lock(&global_count_lk);
        printf("%d\n",global_count);
        pthread_mutex_unlock(&global_count_lk);
    }
    if (flags & FS_STATS)
        prnt_stats();
    return(0); /* should have a return from main */
}

/*
 * Add_Work: Called from the main thread, and cascade threads to add file
 * and directory names to the work Q.
 */
int
add_work(char *path,int tp)
{
    work_t      *wt,*ww,*wp;

    if ((wt = (work_t *)malloc(sizeof(work_t))) == NULL)
        goto ERROR;
```

```
    if ((wt->path = (char *)malloc(strlen(path)+1)) == NULL)
        goto ERROR;

    strcpy(wt->path,path);
    wt->tp = tp;
    wt->next = NULL;
    if (flags & FS_STATS) {
        pthread_mutex_lock(&stat_lk);
        if (wt->tp == DIRT)
            st_dir_search++;
        else
            st_file_search++;
        pthread_mutex_unlock(&stat_lk);
    }
    pthread_mutex_lock(&work_q_lk);
    work_cnt++;
    wt->next = work_q;
    work_q = wt;
    pthread_cond_signal(&work_q_cv);
    pthread_mutex_unlock(&work_q_lk);
    return(0);
 ERROR:
    if (!(flags & FS_NOERROR))
        fprintf(stderr,"tgrep: Could not add %s to work queue. Ignored\n",
                path);
    return(-1);
}

/*
 * Search thread: Started by the main thread when a file name is found
 * on the work Q to be serached. If all the needed resources are ready
 * a new search thread will be created.
 */
void *
search_thr(void *arg) /* work_t *arg */
{
    FILE        *fin;
    char        fin_buf[(BUFSIZ*4)];  /* 4 Kbytes */
    work_t      *wt,std;
    int         line_count;
    char        rline[128];
    char        cline[128];
    char        *line;
    register char *p,*pp;
    int            pm_len;
    int         len = 0;
    long        byte_count;
    long        next_line;
    int         show_line;  /* for the -v option */
    register int slen,plen,i;
    out_t       *out = NULL;    /* this threads output list */

    pthread_yield_np();
    wt = (work_t *)arg; /* first pass, wt is passed to use. */

    /* len = strlen(string);*/  /* only set on first pass */

    while (1) {  /* reuse the search threads */
        /* init all back to zero */
        line_count = 0;
```

```
byte_count = 0l;
next_line = 0l;
show_line = 0;

pthread_mutex_lock(&running_lk);
running++;
pthread_mutex_unlock(&running_lk);
pthread_mutex_lock(&work_q_lk);
tglimit--;
pthread_mutex_unlock(&work_q_lk);
DP(DLEVEL5,("searching file (STDIO) %s\n",wt->path));

if ((fin = fopen(wt->path,"r")) == NULL) {
    if (!(flags & FS_NOERROR)) {
        fprintf(stderr,"tgrep: %s. File \"%s\" not searched.\n",
                strerror(errno),wt->path);
    }
    goto ERROR;
}
setvbuf(fin,fin_buf,_IOFBF,(BUFSIZ*4));  /* XXX */
DP(DLEVEL5,("Search thread has opened file %s\n",wt->path));
while ((fgets(rline,127,fin)) != NULL) {
    if (flags & FS_STATS) {
        pthread_mutex_lock(&stat_lk);
        st_line_search++;
        pthread_mutex_unlock(&stat_lk);
    }
    slen = strlen(rline);
    next_line += slen;
    line_count++;
    if (rline[slen-1] == '\n')
        rline[slen-1] = '\0';
    /*
    ** If the uncase flag is set, copy the read in line (rline)
    ** To the uncase line (cline) Set the line pointer to point at
    ** cline.
    ** If the case flag is NOT set, then point line at rline.
    ** line is what is compared, rline is what is printed on a
    ** match.
    */
    if (flags & FI_IGNCASE) {
        strcpy(cline,rline);
        uncase(cline);
        line = cline;
    }
    else {
        line = rline;
    }
    show_line = 1;  /* assume no match, if -v set */
    /* The old code removed */
    if (use_pmatch) {
        for (i=0; i<regexp_cnt; i++) {
            if (pmatch(pm_pat[i], line, &pm_len)) {
                if (!(flags & FV_REVERSE)) {
                    add_output_local(&out,wt,line_count,
                                     byte_count,rline);
                    continue_line(rline,fin,out,wt,
                                  &line_count,&byte_count);
                }
                else {
```

```
                    show_line = 0;
                } /* end of if -v flag if / else block */
                /*
                ** if we get here on ANY of the regexp targets
                ** jump out of the loop, we found a single
                ** match so do not keep looking!
                ** If name only, do not keep searcthing the same
                ** file, we found a single match, so close the file,
                ** print the file name and move on to the next file.
                */
                if (flags & FL_NAMEONLY)
                    goto OUT_OF_LOOP;
                else
                    goto OUT_AND_DONE;
            } /* end found a match if block */
        } /* end of the for pat[s] loop */
    }
    else {
        if (bm_pmatch( bm_pat, line)) {
            if (!(flags & FV_REVERSE)) {
                add_output_local(&out,wt,line_count,byte_count,rline);
                continue_line(rline,fin,out,wt,
                              &line_count,&byte_count);
            }
            else {
                show_line = 0;
            }
            if (flags & FL_NAMEONLY)
                goto OUT_OF_LOOP;
        }
    }
  OUT_AND_DONE:
    if ((flags & FV_REVERSE) && show_line) {
        add_output_local(&out,wt,line_count,byte_count,rline);
        show_line = 0;
    }
    byte_count = next_line;
  }
OUT_OF_LOOP:
  fclose(fin);
  /*
  ** The search part is done, but before we give back the FD,
  ** and park this thread in the search thread pool, print the
  ** local output we have gathered.
  */
  print_local_output(out,wt);  /* this also frees out nodes */
  out = NULL; /* for the next time around, if there is one */
ERROR:
  DP(DLEVEL5,("Search done for %s\n",wt->path));
  free(wt->path);
  free(wt);

  notrun();
  pthread_mutex_lock(&search_q_lk);
  if (search_pool_cnt > search_thr_limit) {
      pthread_mutex_unlock(&search_q_lk);
      DP(DLEVEL5,("Search thread exiting\n"));
      if (flags & FS_STATS) {
          pthread_mutex_lock(&stat_lk);
          st_destroy++;
```

```
                pthread_mutex_unlock(&stat_lk);
            }
            return(0);
        }
        else {
            search_pool_cnt++;
            while (!search_q)
                pthread_cond_wait(&search_q_cv,&search_q_lk);
            search_pool_cnt--;
            wt = search_q;  /* we have work to do! */
            if (search_q->next)
                search_q = search_q->next;
            else
                search_q = NULL;
            pthread_mutex_unlock(&search_q_lk);
        }
    }
    /*NOTREACHED*/
}

/*
 * Continue line: Special case search with the -C flag set. If you are
 * searching files like Makefiles, some lines might have escape char's to
 * contine the line on the next line. So the target string can be found, but
 * no data is displayed. This function continues to print the escaped line
 * until there are no more "\" chars found.
 */
int
continue_line(char *rline, FILE *fin, out_t *out, work_t *wt,
              int *lc, long *bc)
{
    int len;
    int cnt = 0;
    char *line;
    char nline[128];

    if (!(flags & FC_LINE))
        return(0);

    line = rline;
  AGAIN:
    len = strlen(line);
    if (line[len-1] == '\\') {
        if ((fgets(nline,127,fin)) == NULL) {
            return(cnt);
        }
        line = nline;
        len = strlen(line);
        if (line[len-1] == '\n')
            line[len-1] = '\0';
        *bc = *bc + len;
        *lc++;
        add_output_local(&out,wt,*lc,*bc,line);
        cnt++;
        goto AGAIN;
    }
    return(cnt);
}

/*
```

```
 * cascade: This thread is started by the main thread when directory names
 * are found on the work Q. The thread reads all the new file, and directory
 * names from the directory it was started when and adds the names to the
 * work Q. (it finds more work!)
 */

void *
cascade(void *arg)  /* work_t *arg */
{
    char        fullpath[1025];
    int         restart_cnt = 10;
    DIR         *dp;

    char        dir_buf[sizeof(struct dirent) + PATH_MAX];
    struct dirent *dent = (struct dirent *)dir_buf;
    struct stat   sbuf;
    char        *fpath;
    work_t      *wt;
    int         fl = 0, dl = 0;
    int         pm_file_len = 0;

    pthread_yield_np();  /* try toi give control back to main thread */
    wt = (work_t *)arg;

    while(1) {
        fl = 0;
        dl = 0;
        restart_cnt = 10;
        pm_file_len = 0;

        pthread_mutex_lock(&running_lk);
        running++;
        pthread_mutex_unlock(&running_lk);
        pthread_mutex_lock(&work_q_lk);
        tglimit--;
        pthread_mutex_unlock(&work_q_lk);

        if (!wt) {
            if (!(flags & FS_NOERROR))
                fprintf(stderr,"tgrep: Bad work node passed to cascade\n");
            goto DONE;
        }
        fpath = (char *)wt->path;
        if (!fpath) {
            if (!(flags & FS_NOERROR))
                fprintf(stderr,"tgrep: Bad path name passed to cascade\n");
            goto DONE;
        }
        DP(DLEVEL3,("Cascading on %s\n",fpath));
        if (( dp = opendir(fpath)) == NULL) {
            if (!(flags & FS_NOERROR))
                fprintf(stderr,"tgrep: Can't open dir %s, %s. Ignored.\n",
                        fpath,strerror(errno));
            goto DONE;
        }
        while ((readdir_r(dp,dent)) != NULL) {
            restart_cnt = 10;  /* only try to restart the interupted 10 X */

            if (dent->d_name[0] == '.') {
                if (dent->d_name[1] == '.' && dent->d_name[2] == '\0')
```

```
                continue;
            if (dent->d_name[1] == '\0')
                continue;
        }

        fl = strlen(fpath);
        dl = strlen(dent->d_name);
        if ((fl + 1 + dl) > 1024) {
            fprintf(stderr,"tgrep: Path %s/%s is too long. "
                    "MaxPath = 1024\n",
                    fpath, dent->d_name);
            continue;  /* try the next name in this directory */
        }
        strcpy(fullpath,fpath);
        strcat(fullpath,"/");
        strcat(fullpath,dent->d_name);

      RESTART_STAT:
        if (stat(fullpath,&sbuf)) {
            if (errno == EINTR) {
                if (--restart_cnt)
                    goto RESTART_STAT;
            }
            if (!(flags & FS_NOERROR))
                fprintf(stderr,"tgrep: Can't stat file/dir %s, %s. "
                        "Ignored.\n",
                        fullpath,strerror(errno));
            goto ERROR;
        }

        switch (sbuf.st_mode & S_IFMT) {
        case S_IFREG :
            if (flags & TG_FILEPAT) {
                if (pmatch(pm_file_pat, dent->d_name, &pm_file_len)) {
                    DP(DLEVEL3,("file pat match (cascade) %s\n",
                                dent->d_name));
                    add_work(fullpath,FILET);
                }
            }
            else {
                add_work(fullpath,FILET);
                DP(DLEVEL3,("cascade added file (MATCH) %s to Work Q\n",
                            fullpath));
            }
            break;

        case S_IFDIR :
            DP(DLEVEL3,("cascade added dir %s to Work Q\n",fullpath));
            add_work(fullpath,DIRT);
            break;
        }
    }

ERROR:
    closedir(dp);

DONE:
    free(wt->path);
    free(wt);
    notrun();
```

```
        pthread_mutex_lock(&cascade_q_lk);
        if (cascade_pool_cnt > cascade_thr_limit) {
            pthread_mutex_unlock(&cascade_q_lk);
            DP(DLEVEL5,("Cascade thread exiting\n"));
            if (flags & FS_STATS) {
                pthread_mutex_lock(&stat_lk);
                st_cascade_destroy++;
                pthread_mutex_unlock(&stat_lk);
            }
            return(0); /* pthread_exit */
        }
        else {
            DP(DLEVEL5,("Cascade thread waiting in pool\n"));
            cascade_pool_cnt++;
            while (!cascade_q)
                pthread_cond_wait(&cascade_q_cv,&cascade_q_lk);
            cascade_pool_cnt--;
            wt = cascade_q;  /* we have work to do! */
            if (cascade_q->next)
                cascade_q = cascade_q->next;
            else
                cascade_q = NULL;
            pthread_mutex_unlock(&cascade_q_lk);
        }
    }
    /*NOTREACHED*/
}

/*
 * Print Local Output: Called by the search thread after it is done searching
 * a single file. If any oputput was saved (matching lines), the lines are
 * displayed as a group on stdout.
 */
int
print_local_output(out_t *out, work_t *wt)
{
    out_t       *pp, *op;
    int         out_count = 0;
    int         printed = 0;

    pp = out;
    pthread_mutex_lock(&output_print_lk);
    if (pp && (flags & TG_PROGRESS)) {
        progress++;
        if (progress >= progress_offset) {
            progress = 0;
            fprintf(stderr,".");
        }
    }
    while (pp) {
        out_count++;
        if (!(flags & FC_COUNT)) {
            if (flags & FL_NAMEONLY) {  /* Pint name ONLY ! */
                if (!printed) {
                    printed = 1;
                    printf("%s\n",wt->path);
                }
            }
            else {  /* We are printing more then just the name */
                if (!(flags & FH_HOLDNAME))
```

```
                    printf("%s :",wt->path);
                if (flags & FB_BLOCK)
                    printf("%ld:",pp->byte_count/512+1);
                if (flags & FN_NUMBER)
                    printf("%d:",pp->line_count);
                printf("%s\n",pp->line);
            }
        }
        op = pp;
        pp = pp->next;
        /* free the nodes as we go down the list */
        free(op->line);
        free(op);
    }

    pthread_mutex_unlock(&output_print_lk);
    pthread_mutex_lock(&global_count_lk);
    global_count += out_count;
    pthread_mutex_unlock(&global_count_lk);
    return(0);
}

/*
 * add output local: is called by a search thread as it finds matching lines.
 * the matching line, its byte offset, line count, etc. are stored until the
 * search thread is done searching the file, then the lines are printed as
 * a group. This way the lines from more then a single file are not mixed
 * together.
 */

int
add_output_local(out_t **out, work_t *wt,int lc, long bc, char *line)
{
    out_t       *ot,*oo, *op;

    if (( ot = (out_t *)malloc(sizeof(out_t))) == NULL)
        goto ERROR;
    if (( ot->line = (char *)malloc(strlen(line)+1)) == NULL)
        goto ERROR;

    strcpy(ot->line,line);
    ot->line_count = lc;
    ot->byte_count = bc;

    if (!*out) {
        *out = ot;
        ot->next = NULL;
        return(0);
    }
    /* append to the END of the list; keep things sorted! */
    op = oo = *out;
    while(oo) {
        op = oo;
        oo = oo->next;
    }
    op->next = ot;
    ot->next = NULL;
    return(0);

 ERROR:
```

```
    if (!(flags & FS_NOERROR))
        fprintf(stderr,"tgrep: Output lost. No space. "
                "[%s: line %d byte %d match : %s\n",
                wt->path,lc,bc,line);
    return(1);
}

/*
 * print stats: If the -S flag is set, after ALL files have been searched,
 * main thread calls this function to print the stats it keeps on how the
 * search went.
 */

void
prnt_stats(void)
{
    float a,b,c;
    float t = 0.0;
    time_t  st_end = 0;
    char    tl[80];

    st_end = time(NULL); /* stop the clock */
    printf("\n----------------- Tgrep Stats. --------------------\n");
    printf("Number of directories searched:           %d\n",st_dir_search);
    printf("Number of files searched:                 %d\n",st_file_search);
    c = (float)(st_dir_search + st_file_search) / (float)(st_end - st_start);
    printf("Dir/files per second:                     %3.2f\n",c);
    printf("Number of lines searched:                 %d\n",st_line_search);
    printf("Number of matching lines to target:       %d\n",global_count);

    printf("Number of cascade threads created:        %d\n",st_cascade);
    printf("Number of cascade threads from pool:      %d\n",st_cascade_pool);
    a = st_cascade_pool; b = st_dir_search;
    printf("Cascade thread pool hit rate:             %3.2f%%\n",((a/b)*100));
    printf("Cascade pool overall size:                %d\n",cascade_pool_cnt);
    printf("Number of search threads created:         %d\n",st_search);
    printf("Number of search threads from pool:       %d\n",st_pool);
    a = st_pool; b = st_file_search;
    printf("Search thread pool hit rate:              %3.2f%%\n",((a/b)*100));
    printf("Search pool overall size:                 %d\n",search_pool_cnt);
    printf("Search pool size limit:                   %d\n",search_thr_limit);
    printf("Number of search threads destroyed:       %d\n",st_destroy);

    printf("Max # of threads running concurrenly:     %d\n",st_maxrun);
    printf("Total run time, in seconds.               %d\n",
           (st_end - st_start));

    /* Why did we wait ? */
    a = st_workfds; b = st_dir_search+st_file_search;
    c = (a/b)*100; t += c;
    printf("Work stopped due to no FD's:  (%.3d)       %d Times, %3.2f%%\n",
           search_thr_limit,st_workfds,c);
    a = st_worknull; b = st_dir_search+st_file_search;
    c = (a/b)*100; t += c;
    printf("Work stopped due to no work on Q:         %d Times, %3.2f%%\n",
           st_worknull,c);
    if (tglimit == UNLIMITED)
        strcpy(tl,"Unlimited");
    else
        sprintf(tl,"   %.3d   ",tglimit);
```

```
    a = st_worklimit; b = st_dir_search+st_file_search;
    c = (a/b)*100; t += c;
    printf("Work stopped due to TGLIMIT:  (%.9s) %d Times, %3.2f%%\n",
           tl,st_worklimit,c);
    printf("Work continued to be handed out:          %3.2f%%\n",100.00-t);
    printf("--------------------------------------------------\n");
}
/*
 * not running: A glue function to track if any search threads or cascade
 * threads are running. When the count is zero, and the work Q is NULL,
 * we can safely say, WE ARE DONE.
 */
void
notrun (void)
{
    pthread_mutex_lock(&work_q_lk);
    work_cnt--;
    tglimit++;
    current_open_files++;
    pthread_mutex_lock(&running_lk);
    if (flags & FS_STATS) {
        pthread_mutex_lock(&stat_lk);
        if (running > st_maxrun) {
            st_maxrun = running;
            DP(DLEVEL6,("Max Running has increased to %d\n",st_maxrun));
        }
        pthread_mutex_unlock(&stat_lk);
    }
    running--;
    if (work_cnt == 0 && running == 0) {
        all_done = 1;
        DP(DLEVEL6,("Setting ALL_DONE flag to TRUE.\n"));
    }
    pthread_mutex_unlock(&running_lk);
    pthread_cond_signal(&work_q_cv);
    pthread_mutex_unlock(&work_q_lk);
}

/*
 * uncase: A glue function. If the -i (case insensitive) flag is set, the
 * target strng and the read in line is converted to lower case before
 * comparing them.
 */
void
uncase(char *s)
{
    char        *p;

    for (p = s; *p != NULL; p++)
        *p = (char)tolower(*p);
}

/*
 * usage: Have to have one of these.
 */

void
usage(void)
{
    fprintf(stderr,"usage: tgrep <options> pattern <{file,dir}>...\n");
```

```
    fprintf(stderr,"\n");
    fprintf(stderr,"Where:\n");
#ifdef DEBUG
    fprintf(stderr,"Debug     -d = debug level -d <levels> (-d0 for usage)\n");
    fprintf(stderr,"Debug     -f = block fd's from use (-f #)\n");
#endif
    fprintf(stderr,"          -b = show block count (512 byte block)\n");
    fprintf(stderr,"          -c = print only a line count\n");
    fprintf(stderr,"          -h = Do NOT print file names\n");
    fprintf(stderr,"          -i = case insensitive\n");
    fprintf(stderr,"          -l = print file name only\n");
    fprintf(stderr,"          -n = print the line number with the line\n");
    fprintf(stderr,"          -s = Suppress error messages\n");
    fprintf(stderr,"          -v = print all but matching lines\n");
#ifdef NOT_IMP
    fprintf(stderr,"          -w = search for a \"word\"\n");
#endif
    fprintf(stderr,"          -r = Do not search for files in all "
                               "sub-directories\n");
    fprintf(stderr,"          -C = show continued lines (\"\\\")\n");
    fprintf(stderr,"          -p = File name regexp pattern. (Quote it)\n");
    fprintf(stderr,"          -P = show progress. -P 1 prints a DOT on stderr\n"
                   "               for each file it finds, -P 10 prints a DOT\n"
                   "               on stderr for each 10 files it finds, etc...\n");
    fprintf(stderr,"          -e = expression search.(regexp) More then one\n");
    fprintf(stderr,"          -B = limit the number of threads to TGLIMIT\n");
    fprintf(stderr,"          -S = Print thread stats when done.\n");
    fprintf(stderr,"          -Z = Print help on the regexp used.\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"Notes:\n");
    fprintf(stderr,"      If you start tgrep with only a directory name\n");
    fprintf(stderr,"      and no file names, you must not have the -r option\n");
    fprintf(stderr,"      set or you will get no output.\n");
    fprintf(stderr,"      To search stdin (piped input), you must set -r\n");
    fprintf(stderr,"      Tgrep will search ALL files in ALL \n");
    fprintf(stderr,"      sub-directories. (like */* */*/* */*/*/* etc..)\n");
    fprintf(stderr,"      if you supply a directory name.\n");
    fprintf(stderr,"      If you do not supply a file, or directory name,\n");
    fprintf(stderr,"      and the -r option is not set, the current \n");
    fprintf(stderr,"      directory \".\" will be used.\n");
    fprintf(stderr,"      All the other options should work \"like\" grep\n");
    fprintf(stderr,"      The -p patten is regexp; tgrep will search only\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"      Copy Right By Ron Winacott, 1993-1995.\n");
    fprintf(stderr,"\n");
    exit(0);
}

/*
 * regexp usage: Tell the world about tgrep custom (THREAD SAFE) regexp!
 */
int
regexp_usage (void)
{
    fprintf(stderr,"usage: tgrep <options> -e \"pattern\" <-e ...> "
            "<{file,dir}>...\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"metachars:\n");
    fprintf(stderr,"    . - match any character\n");
    fprintf(stderr,"    * - match 0 or more occurrences of previous char\n");
```

```
    fprintf(stderr,"    + - match 1 or more occurrences of previous char.\n");
    fprintf(stderr,"    ^ - match at beginning of string\n");
    fprintf(stderr,"    $ - match end of string\n");
    fprintf(stderr,"    [ - start of character class\n");
    fprintf(stderr,"    ] - end of character class\n");
    fprintf(stderr,"    ( - start of a new pattern\n");
    fprintf(stderr,"    ) - end of a new pattern\n");
    fprintf(stderr,"    @(n)c - match <c> at column <n>\n");
    fprintf(stderr,"    | - match either pattern\n");
    fprintf(stderr,"    \\ - escape any special characters\n");
    fprintf(stderr,"    \\c - escape any special characters\n");
    fprintf(stderr,"    \\o - turn on any special characters\n");
    fprintf(stderr,"\n");
    fprintf(stderr,"To match two diffrerent patterns in the same command\n");
    fprintf(stderr,"Use the or function. \n"
            "ie: tgrep -e \"(pat1)|(pat2)\" file\n"
            "This will match any line with \"pat1\" or \"pat2\" in it.\n");
    fprintf(stderr,"You can also use up to %d -e expressions\n",MAXREGEXP);
    fprintf(stderr,"RegExp Pattern matching brought to you by Marc Staveley\n");
    exit(0);
}

/*
 * debug usage: If compiled with -DDEBUG, turn it on, and tell the world
 * how to get tgrep to print debug info on different threads.
 */

#ifdef DEBUG
void
debug_usage(void)
{
    int i = 0;

    fprintf(stderr,"DEBUG usage and levels:\n");
    fprintf(stderr,"-------------------------------------------------\n");
    fprintf(stderr,"Level                    code\n");
    fprintf(stderr,"-------------------------------------------------\n");
    fprintf(stderr,"0                 This message.\n");
    for (i=0; i<9; i++) {
        fprintf(stderr,"%d                %s\n",i+1,debug_set[i].name);
    }
    fprintf(stderr,"-------------------------------------------------\n");
    fprintf(stderr,"You can or the levels together like -d134 for levels\n");
    fprintf(stderr,"1 and 3 and 4.\n");
    fprintf(stderr,"\n");
    exit(0);
}
#endif

/* Pthreads NP functions */

#ifdef __sun
void
pthread_setconcurrency_np(int con)
{
    thr_setconcurrency(con);
}

int
pthread_getconcurrency_np(void)
```

```
{
    return(thr_getconcurrency());
}

void
pthread_yield_np(void)
{
/*     In Solaris 2.4, these functions always return - 1 and set errno to ENOSYS */
    if (sched_yield())  /* call UI interface if we are older than 2.5 */
        thr_yield();
}

#else
void
pthread_setconcurrency_np(int con)
{
    return;
}

int
pthread_getconcurrency_np(void)
{
    return(0);
}

void
pthread_yield_np(void)
{
    return;
}
#endif
```

付録 **B**

Solaris スレッドの例 ― barrier.c

barrier.c プログラムは、Solaris スレッドのためのバリアの実装例です。(バリアの定義については、282ページの「共有メモリー型の並列コンピュータでのループの並列化」を参照してください。)

例 **B–1** Solaris スレッドの例 ― barrier.c

```
#define _REENTRANT

/* インクルードファイル */

#include <thread.h>
#include <errno.h>

/* 定数とマクロ */

/* データ宣言 */

typedef struct {
     int     maxcnt;     /* スレッドの最大数 */
     struct _sb {
        cond_t  wait_cv;   /* バリアで待つスレッドの cv */
        mutex_t wait_lk;   /* バリアで待つスレッドの mutex */
        int     runners;   /* 実行するスレッド数 */
     } sb[2];
     struct _sb   *sbp;    /* 現在のサブバリア */
} barrier_t;



/*
 * barrier_init - バリア変数を初期化する
 */

int
barrier_init( barrier_t *bp, int count, int type, void *arg ) {
     int n;
```

```
     int i;

     if (count < 1)
         return(EINVAL);

     bp->maxcnt = count;
     bp->sbp = &bp->sb[0];

     for (i = 0; i < 2; ++i) {
#if defined(__cplusplus)
     struct barrier_t::_sb *sbp = &( bp->sb[i] );
#else
     struct _sb *sbp = &( bp->sb[i] );
#endif
     sbp->runners = count;

     if (n = mutex_init(&sbp->wait_lk, type, arg))
             return(n);

     if (n = cond_init(&sbp->wait_cv, type, arg))
             return(n);
     }
     return(0);
}

/*
 * barrier_wait - 全部が到着するまでバリアで待つ
 */

int
barrier_wait(register barrier_t *bp) {
#if defined(__cplusplus)
     register struct barrier_t::_sb *sbp = bp->sbp;
#else
     register struct _sb *sbp = bp->sbp;
#endif
     mutex_lock(&sbp->wait_lk);

     if (sbp->runners == 1) {      /* バリアに最後に到着したスレッド */
           if (bp->maxcnt != 1) {
           /* 実行スレッドカウントをリセットし、サブバリアを切り替える */
                  sbp->runners = bp->maxcnt;
                  bp->sbp = (bp->sbp == &bp->sb[0])
                              ? &bp->sb[1] : &bp->sb[0];

                  /* 待ちスレッドを呼び起こす */
                  cond_broadcast(&sbp->wait_cv);
            }
        } else {
           sbp->runners--;         /* 1 小さい実行スレッド */

           while (sbp->runners != bp->maxcnt)
                  cond_wait( &sbp->wait_cv, &sbp->wait_lk);
        }

     mutex_unlock(&sbp->wait_lk);

     return(0);
}
```

```
/*
 * barrier_destroy - バリア変数を削除する
 */

int
barrier_destroy(barrier_t *bp) {
        int     n;
        int     i;

        for (i=0; i < 2; ++ i) {
                if (n = cond_destroy(&bp->sb[i].wait_cv))
                        return( n );

                if (n = mutex_destroy( &bp->sb[i].wait_lk))
                        return(n);
        }

        return(0);
}


#define NTHR    4
#define NCOMPUTATION 2
#define NITER   1000
#define NSQRT   1000

        void *
compute(barrier_t *ba )
{
 int count = NCOMPUTATION;

 while (count--) {
  barrier_wait( ba );
  /* 並列計算 */
 }
}

main( int argc, char *argv[] ) {
        int             i;
        int             niter;
        int             nthr;
        barrier_t       ba;
        double          et;
        thread_t        *tid;

        switch ( argc ) {
          default:
          case 3 :      niter   = atoi( argv[1] );
                        nthr    = atoi( argv[2] );
                        break;

          case 2 :      niter   = atoi( argv[1] );
                        nthr    = NTHR;
                        break;

          case 1 :      niter   = NITER;
                        nthr    = NTHR;
                        break;
        }
```

```
        barrier_init( &ba, nthr + 1, USYNC_THREAD, NULL );
        tid = (thread_t *) calloc(nthr, sizeof(thread_t));

        for (i = 0; i < nthr; ++i) {
                int     n;

                if (n = thr_create(NULL, 0,
          (void *(*)( void *)) compute,
         &ba,NULL, &tid[i])) {
                        errno = n;
                        perror("thr_create");
                        exit(1);
                }
        }

 for (i = 0; i < NCOMPUTATION; i++) {
         barrier_wait(&ba );
  /* 並列アルゴリズム */
 }

 for (i = 0; i < nthr; i++) {
  thr_join(tid[i], NULL, NULL);
 }

}
```

付録 C

「MT-安全」ライブラリインタフェース

付録 C では、『*man pages section 3*』 に記載されているインタフェースの安全レベルを示します (安全性の分類については、199ページの「マルチスレッドインタフェースの安全レベル」を参照してください)。

表 C–1 ライブラリルーチンの MT-安全レベル

ライブラリルーチン	安全レベル
a64l(3C)	MT-安全
abort(3C)	安全
abs(3C)	MT-安全
accept(3N)	安全
acos(3M)	MT-安全
acosh(3M)	MT-安全
addch(3X)	安全ではない
addchnstr(3X)	安全ではない
addchstr(3X)	安全ではない
addnstr(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
addnwstr(3X)	安全ではない
addsev(3C)	MT-安全
addseverity(3C)	安全
addstr(3X)	安全ではない
addwch(3X)	安全ではない
addwchnstr(3X)	安全ではない
addwchstr(3X)	安全ではない
addwstr(3X)	安全ではない
adjcurspos(3X)	安全ではない
advance(3G)	MT-安全
aiocancel(3)	安全ではない
aioread(3)	安全ではない
aiowait(3)	安全ではない
aiowrite(3)	安全ではない
aio_cancel(3R)	MT-安全
aio_error(3R)	非同期シグナル安全
aio_fsync(3R)	MT-安全
aio_read(3R)	MT-安全
aio_return(3R)	非同期シグナル安全

表 C–1 ライブラリルーチンの MT-安全レベル 続く

ライブラリルーチン	安全レベル
aio_suspend(3R)	非同期シグナル安全
aio_write(3R)	MT-安全
alloca(3C)	安全
arc(3)	安全
ascftime(3C)	MT-安全
asctime(3C)	安全ではない。asctime_r() を使用
asin(3M)	MT-安全
asinh(3M)	MT-安全
assert(3C)	安全
atan(3M)	MT-安全
atan2(3M)	MT-安全
atanh(3M)	MT-安全
atexit(3C)	安全
atof(3C)	MT-安全
atoi(3C)	MT-安全
atol(3C)	MT-安全
atoll(3C)	MT-安全
attroff(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
attron(3X)	安全ではない
attrset(3X)	安全ではない
authdes_create(3N)	安全ではない
authdes_getucred(3N)	MT-安全
authdes_seccreate(3N)	MT-安全
authkerb_getucred(3N)	安全ではない
authkerb_seccreate(3N)	安全ではない
authnone_create(3N)	MT-安全
authsys_create(3N)	MT-安全
authsys_create_default(3N)	MT-安全
authunix_create(3N)	安全ではない
authunix_create_default(3N)	安全ではない
auth_destroy(3N)	MT-安全
au_close(3)	安全
au_open(3)	安全
au_user_mask(3)	MT-安全
au_write(3)	安全
basename(3G)	MT-安全
baudrate(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
beep(3X)	安全ではない
bessel(3M)	MT-安全
bgets(3G)	MT-安全
bind(3N)	安全
bindtextdomain(3I)	例外付きで安全
bkgd(3X)	安全ではない
bkgdset(3X)	安全ではない
border(3X)	安全ではない
bottom_panel(3X)	安全ではない
box(3)	安全
box(3X)	安全ではない
bsearch(3C)	安全
bufsplit(3G)	MT-安全
byteorder(3N)	安全
calloc(3C)	安全
calloc(3X)	安全
callrpc(3N)	安全ではない
cancellation(3T)	MT-安全
can_change_color(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
catclose(3C)	MT-安全
catgets(3C)	MT-安全
catopen(3C)	MT-安全
cbc_crypt(3)	MT-安全
cbreak(3X)	安全ではない
cbrt(3M)	MT-安全
ceil(3M)	MT-安全
cfgetispeed(3)	MT-安全、非同期シグナル安全
cfgetospeed(3)	MT-安全、非同期シグナル安全
cfree(3X)	安全
cfsetispeed(3)	MT-安全、非同期シグナル安全
cfsetospeed(3)	MT-安全、非同期シグナル安全
cftime(3C)	MT-安全
circle(3)	安全
clear(3X)	安全ではない
clearerr(3S)	MT-安全
clearok(3X)	安全ではない
clntraw_create(3N)	安全ではない
clnttcp_create(3N)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
clntudp_bufcreate(3N)	安全ではない
clntudp_create(3N)	安全ではない
clnt_broadcast(3N)	安全ではない
clnt_call(3N)	MT-安全
clnt_control(3N)	MT-安全
clnt_create(3N)	MT-安全
clnt_create_timed(3N)	MT-安全
clnt_create_vers(3N)	MT-安全
clnt_destroy(3N)	MT-安全
clnt_dg_create(3N)	MT-安全
clnt_freeres(3N)	MT-安全
clnt_geterr(3N)	MT-安全
clnt_pcreateerror(3N)	MT-安全
clnt_perrno(3N)	MT-安全
clnt_perror(3N)	MT-安全
clnt_raw_create(3N)	MT-安全
clnt_spcreateerror(3N)	MT-安全
clnt_sperrno(3N)	MT-安全
clnt_sperror(3N)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
clnt_tli_create(3N)	MT-安全
clnt_tp_create(3N)	MT-安全
clnt_tp_create_timed(3N)	MT-安全
clnt_vc_create(3N)	MT-安全
clock(3C)	MT-安全
clock_gettime(3R)	非同期シグナル安全
closedir(3C)	安全
closelog(3)	安全
closepl(3)	安全
closevt(3)	安全
clrtobot(3X)	安全ではない
clrtoeol(3X)	安全ではない
color_content(3X)	安全ではない
compile(3G)	MT-安全
condition(3T)	MT-安全
cond_broadcast(3T)	MT-安全
cond_destroy(3T)	MT-安全
cond_init(3T)	MT-安全
cond_signal(3T)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
cond_timedwait(3T)	MT-安全
cond_wait(3T)	MT-安全
confstr(3C)	MT-安全
connect(3N)	安全
cont(3)	安全
conv(3C)	例外付きで MT-安全
copylist(3G)	MT-安全
copysign(3M)	MT-安全
copywin(3X)	安全ではない
cos(3M)	MT-安全
cosh(3M)	MT-安全
crypt(3C)	安全
crypt(3X)	安全ではない
cset(3I)	例外付きで MT-安全
csetcol(3I)	例外付きで MT-安全
csetlen(3I)	例外付きで MT-安全
csetno(3I)	例外付きで MT-安全
ctermid(3S)	安全ではない。 ctermid_r() を使用

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
ctime(3C)	安全ではない。 ctime_r() を使用
ctype(3C)	例外付きで MT-安全
current_field(3X)	安全ではない
current_item(3X)	安全ではない
curses(3X)	安全ではない
curs_addch(3X)	安全ではない
curs_addchstr(3X)	安全ではない
curs_addstr(3X)	安全ではない
curs_addwch(3X)	安全ではない
curs_addwchstr(3X)	安全ではない
curs_addwstr(3X)	安全ではない
curs_alecompat(3X)	安全ではない
curs_attr(3X)	安全ではない
curs_beep(3X)	安全ではない
curs_bkgd(3X)	安全ではない
curs_border(3X)	安全ではない
curs_clear(3X)	安全ではない
curs_color(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
curs_delch(3X)	安全ではない
curs_deleteln(3X)	安全ではない
curs_getch(3X)	安全ではない
curs_getstr(3X)	安全ではない
curs_getwch(3X)	安全ではない
curs_getwstr(3X)	安全ではない
curs_getyx(3X)	安全ではない
curs_inch(3X)	安全ではない
curs_inchstr(3X)	安全ではない
curs_initscr(3X)	安全ではない
curs_inopts(3X)	安全ではない
curs_insch(3X)	安全ではない
curs_insstr(3X)	安全ではない
curs_instr(3X)	安全ではない
curs_inswch(3X)	安全ではない
curs_inswstr(3X)	安全ではない
curs_inwch(3X)	安全ではない
curs_inwchstr(3X)	安全ではない
curs_inwstr(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
curs_kernel(3X)	安全ではない
curs_move(3X)	安全ではない
curs_outopts(3X)	安全ではない
curs_overlay(3X)	安全ではない
curs_pad(3X)	安全ではない
curs_printw(3X)	安全ではない
curs_refresh(3X)	安全ではない
curs_scanw(3X)	安全ではない
curs_scroll(3X)	安全ではない
curs_scr_dump(3X)	安全ではない
curs_set(3X)	安全ではない
curs_slk(3X)	安全ではない
curs_termattrs(3X)	安全ではない
curs_termcap(3X)	安全ではない
curs_terminfo(3X)	安全ではない
curs_touch(3X)	安全ではない
curs_util(3X)	安全ではない
curs_window(3X)	安全ではない
cuserid(3S)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
data_ahead(3X)	安全ではない
data_behind(3X)	安全ではない
dbm_clearerr(3)	安全ではない
dbm_close(3)	安全ではない
dbm_delete(3)	安全ではない
dbm_error(3)	安全ではない
dbm_fetch(3)	安全ではない
dbm_firstkey(3)	安全ではない
dbm_nextkey(3)	安全ではない
dbm_open(3)	安全ではない
dbm_store(3)	安全ではない
db_add_entry(3N)	安全ではない
db_checkpoint(3N)	安全ではない
db_create_table(3N)	安全ではない
db_destroy_table(3N)	安全ではない
db_first_entry(3N)	安全ではない
db_free_result(3N)	安全ではない
db_initialize(3N)	安全ではない
db_list_entries(3N)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
db_next_entry(3N)	安全ではない
db_remove_entry(3N)	安全ではない
db_reset_next_entry(3N)	安全ではない
db_standby(3N)	安全ではない
db_table_exists(3N)	安全ではない
db_unload_table(3N)	安全ではない
dcgettext(3I)	例外付きで安全
decimal_to_double(3)	MT-安全
decimal_to_extended(3)	MT-安全
decimal_to_floating(3)	MT-安全
decimal_to_quadruple(3)	MT-安全
decimal_to_single(3)	MT-安全
def_prog_mode(3X)	安全ではない
def_shell_mode(3X)	安全ではない
delay_output(3X)	安全ではない
delch(3X)	安全ではない
deleteln(3X)	安全ではない
delscreen(3X)	安全ではない
delwin(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
del_curterm(3X)	安全ではない
del_panel(3X)	安全ではない
derwin(3X)	安全ではない
des_crypt(3)	MT-安全
DES_FAILED(3)	MT-安全
des_failed(3)	MT-安全
des_setparity(3)	MT-安全
dgettext(3I)	例外付きで安全
dial(3N)	安全ではない
difftime(3C)	MT-安全
dirname(3G)	MT-安全
div(3C)	MT-安全
dladdr(3X)	MT-安全
dlclose(3X)	MT-安全
dlerror(3X)	MT-安全
dlopen(3X)	MT-安全
dlsym(3X)	MT-安全
dn_comp(3N)	安全ではない
dn_expand(3N)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
doconfig(3N)	安全ではない
double_to_decimal(3)	MT-安全
doupdate(3X)	安全ではない
drand48(3C)	安全
dup2(3C)	安全ではない。非同期シグナル安全
dupwin(3X)	安全ではない
dup_field(3X)	安全ではない
dynamic_field_info(3X)	安全ではない
ecb_crypt(3)	MT-安全
echo(3X)	安全ではない
echochar(3X)	安全ではない
echowchar(3X)	安全ではない
econvert(3)	MT-安全
ecvt(3)	MT-安全
ecvt(3C)	安全ではない
el(32_fsize.3E)	安全ではない
el(32_getehdr.3E)	安全ではない
el(32_getshdr.3E)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
el(32_newehdr.3E)	安全ではない
el(32_newphdr.3E)	安全ではない
el(32_xlatetof.3E)	安全ではない
el(32_xlatetom.3E)	安全ではない
elf(3E)	安全ではない
elf_begin(3E)	安全ではない
elf_cntl(3E)	安全ではない
elf_end(3E)	安全ではない
elf_errmsg(3E)	安全ではない
elf_errno(3E)	安全ではない
elf_fill(3E)	安全ではない
elf_flagdata(3E)	安全ではない
elf_flagehdr(3E)	安全ではない
elf_flagelf(3E)	安全ではない
elf_flagphdr(3E)	安全ではない
elf_flagscn(3E)	安全ではない
elf_flagshdr(3E)	安全ではない
elf_getarhdr(3E)	安全ではない
elf_getarsym(3E)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
elf_getbase(3E)	安全ではない
elf_getdata(3E)	安全ではない
elf_getident(3E)	安全ではない
elf_getscn(3E)	安全ではない
elf_hash(3E)	安全ではない
elf_kind(3E)	安全ではない
elf_memory(3E)	安全ではない
elf_ndxscn(3E)	安全ではない
elf_newdata(3E)	安全ではない
elf_newscn(3E)	安全ではない
elf_next(3E)	安全ではない
elf_nextscn(3E)	安全ではない
elf_rand(3E)	安全ではない
elf_rawdata(3E)	安全ではない
elf_rawfile(3E)	安全ではない
elf_strptr(3E)	安全ではない
elf_update(3E)	安全ではない
elf_version(3E)	安全ではない
encrypt(3C)	安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
endac(3)	安全
endauclass(3)	MT-安全
endauevent(3)	MT-安全
endauuser(3)	MT-安全
endnetconfig(3N)	MT-安全
endnetpath(3N)	MT-安全
endutent(3C)	安全ではない
endutxent(3C)	安全ではない
endwin(3X)	安全ではない
erand48(3C)	安全
erase(3)	安全
erase(3X)	安全ではない
erasechar(3X)	安全ではない
erf(3M)	MT-安全
erfc(3M)	MT-安全
errno(3C)	MT-安全
ethers(3N)	MT-安全
ether_aton(3N)	MT-安全
ether_hostton(3N)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
ether_line(3N)	MT-安全
ether_ntoa(3N)	MT-安全
ether_ntohost(3N)	MT-安全
euccol(3I)	安全
euclen(3I)	安全
eucscol(3I)	安全
exit(3C)	安全
exp(3M)	MT-安全
expm1(3M)	MT-安全
extended_to_decimal(3)	MT-安全
fabs(3M)	MT-安全
fattach(3C)	MT-安全
fclose(3S)	MT-安全
fconvert(3)	MT-安全
fcvt(3)	MT-安全
fcvt(3C)	安全ではない
fdatasync(3R)	非同期シグナル安全
fdetach(3C)	安全ではない
fdopen(3S)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
feof(3S)	MT-安全
ferror(3S)	MT-安全
fflush(3S)	MT-安全
ffs(3C)	MT-安全
fgetc(3S)	MT-安全
fgetgrent(3C)	安全ではない。 fgetgrent_r() を使用
fgetpos(3C)	MT-安全
fgetpwent(3C)	安全ではない。 fgetpwent_r() を使用
fgets(3S)	MT-安全
fgetspent(3C)	安全ではない。 fgetgrent_r() を使用
fgetwc(3I)	MT-安全
fgetws(3I)	MT-安全
field_arg(3X)	安全ではない
field_back(3X)	安全ではない
field_buffer(3X)	安全ではない
field_count(3X)	安全ではない
field_fore(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
field_index(3X)	安全ではない
field_info(3X)	安全ではない
field_init(3X)	安全ではない
field_just(3X)	安全ではない
field_opts(3X)	安全ではない
field_opts_off(3X)	安全ではない
field_opts_on(3X)	安全ではない
field_pad(3X)	安全ではない
field_status(3X)	安全ではない
field_term(3X)	安全ではない
field_type(3X)	安全ではない
field_userptr(3X)	安全ではない
fileno(3S)	MT-安全
file_to_decimal(3)	MT-安全
filter(3X)	安全ではない
finite(3C)	MT-安全
flash(3X)	安全ではない
floating_to_decimal(3)	MT-安全
flockfile(3S)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
floor(3M)	MT-安全
flushinp(3X)	安全ではない
fmod(3M)	MT-安全
fmtmsg(3C)	安全
fnmatch(3C)	MT-安全
fn_attribute_add(3N)	安全
fn_attribute_assign(3N)	安全
fn_attribute_copy(3N)	安全
fn_attribute_create(3N)	安全
fn_attribute_destroy(3N)	安全
fn_attribute_first(3N)	安全
fn_attribute_identifier(3N)	安全
fn_attribute_next(3N)	安全
fn_attribute_remove(3N)	安全
fn_attribute_syntax(3N)	安全
FN_attribute_t(3N)	安全
fn_attribute_valuecount(3N)	安全
fn_attrmodlist_add(3N)	安全
fn_attrmodlist_assign(3N)	安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
fn_attrmodlist_copy(3N)	安全
fn_attrmodlist_count(3N)	安全
fn_attrmodlist_create(3N)	安全
fn_attrmodlist_destroy(3N)	安全
fn_attrmodlist_first(3N)	安全
fn_attrmodlist_next(3N)	安全
FN_attrmodlist_t(3N)	安全
fn_attrset_add(3N)	安全
fn_attrset_assign(3N)	安全
fn_attrset_copy(3N)	安全
fn_attrset_count(3N)	安全
fn_attrset_create(3N)	安全
fn_attrset_destroy(3N)	安全
fn_attrset_first(3N)	安全
fn_attrset_get(3N)	安全
fn_attrset_next(3N)	安全
fn_attrset_remove(3N)	安全
FN_attrset_t(3N)	安全
fn_attr_get(3N)	安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
fn_attr_get_ids(3N)	安全
fn_attr_get_values(3N)	安全
fn_attr_modify(3N)	安全
fn_attr_multi_get(3N)	安全
fn_attr_multi_modify(3N)	安全
fn_bindinglist_destroy(3N)	安全
fn_bindinglist_next(3N)	安全
FN_bindinglist_t(3N)	安全
fn_composite_name_append_comp(3N)	安全
fn_composite_name_append_name(3N)	安全
fn_composite_name_assign(3N)	安全
fn_composite_name_copy(3N)	安全
fn_composite_name_count(3N)	安全
fn_composite_name_create(3N)	安全
fn_composite_name_delete_comp(3N)	安全
fn_composite_name_destroy(3N)	安全
fn_composite_name_first(3N)	安全
fn_composite_name_from_string(3N)	安全
fn_composite_name_insert_comp(3N)	安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
fn_composite_name_insert_name(3N)	安全
fn_composite_name_is_empty(3N)	安全
fn_composite_name_is_equal(3N)	安全
fn_composite_name_is_prefix(3N)	安全
fn_composite_name_is_suffix(3N)	安全
fn_composite_name_last(3N)	安全
fn_composite_name_next(3N)	安全
fn_composite_name_prefix(3N)	安全
fn_composite_name_prepend_comp(3N)	安全
fn_composite_name_prepend_name(3N)	安全
fn_composite_name_prev(3N)	安全
fn_composite_name_suffix(3N)	安全
FN_composite_name_t(3N)	安全
fn_compound_name_append_comp(3N)	安全
fn_compound_name_assign(3N)	安全
fn_compound_name_copy(3N)	安全
fn_compound_name_count(3N)	安全
fn_compound_name_delete_all(3N)	安全
fn_compound_name_delete_comp(3N)	安全

表 C–1 ライブラリルーチンの MT-安全レベル 続く

ライブラリルーチン	安全レベル
fn_compound_name_destroy(3N)	安全
fn_compound_name_first(3N)	安全
fn_compound_name_from_syntax_attrs	安全
fn_compound_name_get_syntax_attrs(3N)	安全
fn_compound_name_insert_comp(3N)	安全
fn_compound_name_is_empty(3N)	安全
fn_compound_name_is_equal(3N)	安全
fn_compound_name_is_prefix(3N)	安全
fn_compound_name_is_suffix(3N)	安全
fn_compound_name_last(3N)	安全
fn_compound_name_next(3N)	安全
fn_compound_name_prefix(3N)	安全
fn_compound_name_prepend_comp(3N)	安全
fn_compound_name_prev(3N)	安全
fn_compound_name_suffix(3N)	安全
FN_compound_name_t(3N)	安全
fn_ctx_bind(3N)	安全
fn_ctx_create_subcontext(3N)	安全
fn_ctx_destroy_subcontext(3N)	安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
fn_ctx_get_ref(3N)	安全
fn_ctx_get_syntax_attrs(3N)	安全
fn_ctx_handle_destroy(3N)	安全
fn_ctx_handle_from_initial(3N)	MT-安全
fn_ctx_handle_from_ref(3N)	安全
fn_ctx_list_bindings(3N)	安全
fn_ctx_list_names(3N)	安全
fn_ctx_lookup(3N)	安全
fn_ctx_lookup_link(3N)	安全
fn_ctx_rename(3N)	安全
FN_ctx_t(3N)	安全
fn_ctx_unbind(3N)	安全
fn_multigetlist_destroy(3N)	安全
fn_multigetlist_next(3N)	安全
FN_multigetlist_t(3N)	安全
fn_namelist_destroy(3N)	安全
fn_namelist_next(3N)	安全
FN_namelist_t(3N)	安全
fn_ref_addrcount(3N)	安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
fn_ref_addr_assign(3N)	安全
fn_ref_addr_copy(3N)	安全
fn_ref_addr_create(3N)	安全
fn_ref_addr_data(3N)	安全
fn_ref_addr_description(3N)	安全
fn_ref_addr_destroy(3N)	安全
fn_ref_addr_length(3N)	安全
FN_ref_addr_t(3N)	安全
fn_ref_addr_type(3N)	安全
fn_ref_append_addr(3N)	安全
fn_ref_assign(3N)	安全
fn_ref_copy(3N)	安全
fn_ref_create(3N)	安全
fn_ref_create_link(3N)	安全
fn_ref_delete_addr(3N)	安全
fn_ref_delete_all(3N)	安全
fn_ref_description(3N)	安全
fn_ref_destroy(3N)	安全
fn_ref_first(3N)	安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
fn_ref_insert_addr(3N)	安全
fn_ref_is_link(3N)	安全
fn_ref_link_name(3N)	安全
fn_ref_next(3N)	安全
fn_ref_prepend_addr(3N)	安全
FN_ref_t(3N)	安全
fn_ref_type(3N)	安全
fn_status_advance_by_name(3N)	安全
fn_status_append_remaining_name(3N)	安全
fn_status_append_resolved_name(3N)	安全
fn_status_assign(3N)	安全
fn_status_code(3N)	安全
fn_status_copy(3N)	安全
fn_status_create(3N)	安全
fn_status_description(3N)	安全
fn_status_destroy(3N)	安全
fn_status_diagnostic_message(3N)	安全
fn_status_is_success(3N)	安全
fn_status_link_code(3N)	安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
fn_status_link_diagnostic_message(3N)	安全
fn_status_link_remaining_name(3N)	安全
fn_status_link_resolved_name(3N)	安全
fn_status_link_resolved_ref(3N)	安全
fn_status_remaining_name(3N)	安全
fn_status_resolved_name(3N)	安全
fn_status_resolved_ref(3N)	安全
fn_status_set(3N)	安全
fn_status_set_code(3N)	安全
fn_status_set_diagnostic_message(3N)	安全
fn_status_set_link_code(3N)	安全
fn_status_set_link_diagnostic_message	安全
fn_status_set_link_remaining_name(3N)	安全
fn_status_set_link_resolved_name(3N)	安全
fn_status_set_link_resolved_ref(3N)	安全
fn_status_set_remaining_name(3N)	安全
fn_status_set_resolved_name(3N)	安全
fn_status_set_resolved_ref(3N)	安全
fn_status_set_success(3N)	安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
FN_status_t(3N)	安全
fn_string_assign(3N)	安全
fn_string_bytecount(3N)	安全
fn_string_charcount(3N)	安全
fn_string_code_set(3N)	安全
fn_string_compare(3N)	安全
fn_string_compare_substring(3N)	安全
fn_string_contents(3N)	安全
fn_string_copy(3N)	安全
fn_string_create(3N)	安全
fn_string_destroy(3N)	安全
fn_string_from_composite_name(3N)	安全
fn_string_from_compound_name(3N)	安全
fn_string_from_contents(3N)	安全
fn_string_from_str(3N)	安全
fn_string_from_strings(3N)	安全
fn_string_from_str_n(3N)	安全
fn_string_from_substring(3N)	安全
fn_string_is_empty(3N)	安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
fn_string_next_substring(3N)	安全
fn_string_prev_substring(3N)	安全
fn_string_str(3N)	安全
FN_string_t(3N)	安全
fn_valuelist_destroy(3N)	安全
fn_valuelist_next(3N)	安全
FN_valuelist_t(3N)	安全
fopen(3S)	MT-安全
forms(3X)	安全ではない
form_cursor(3X)	安全ではない
form_data(3X)	安全ではない
form_driver(3X)	安全ではない
form_field(3X)	安全ではない
form_fields(3X)	安全ではない
form_fieldtype(3X)	安全ではない
form_field_attributes(3X)	安全ではない
form_field_buffer(3X)	安全ではない
form_field_info(3X)	安全ではない
form_field_just(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
form_field_new(3X)	安全ではない
form_field_opts(3X)	安全ではない
form_field_userptr(3X)	安全ではない
form_field_validation(3X)	安全ではない
form_hook(3X)	安全ではない
form_init(3X)	安全ではない
form_new(3X)	安全ではない
form_new_page(3X)	安全ではない
form_opts(3X)	安全ではない
form_opts_off(3X)	安全ではない
form_opts_on(3X)	安全ではない
form_page(3X)	安全ではない
form_post(3X)	安全ではない
form_sub(3X)	安全ではない
form_term(3X)	安全ではない
form_userptr(3X)	安全ではない
form_win(3X)	安全ではない
fpclass(3C)	MT-安全
fpgetmask(3C)	MT-安全

表 C–1 ライブラリルーチンの MT-安全レベル 続く

ライブラリルーチン	安全レベル
fpgetround(3C)	MT-安全
fpgetsticky(3C)	MT-安全
fprintf(3S)	MT-安全 (setlocale() を除く)
fpsetmask(3C)	MT-安全
fpsetround(3C)	MT-安全
fpsetsticky(3C)	MT-安全
fputc(3S)	MT-安全
fputs(3S)	MT-安全
fputwc(3I)	MT-安全
fputws(3I)	MT-安全
fread(3S)	MT-安全
free(3C)	安全
free(3X)	安全
freenetconfigent(3N)	MT-安全
free_field(3X)	安全ではない
free_fieldtype(3X)	安全ではない
free_form(3X)	安全ではない
free_item(3X)	安全ではない
free_menu(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
freopen(3S)	MT-安全
frexp(3C)	MT-安全
fscanf(3S)	MT-安全
fseek(3S)	MT-安全
fsetpos(3C)	MT-安全
fsync(3C)	非同期シグナル安全
ftell(3S)	MT-安全
ftok(3C)	MT-安全
ftruncate(3C)	MT-安全
ftrylockfile(3S)	MT-安全
ftw(3C)	安全
func_to_decimal(3)	MT-安全
funlockfile(3S)	MT-安全
fwrite(3S)	MT-安全
gconvert(3)	MT-安全
gcvt(3)	MT-安全
gcvt(3C)	安全ではない
getacdir(3)	安全
getacflg(3)	安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
getacinfo(3)	安全
getacmin(3)	安全
getacna(3)	安全
getauclassent(3)	安全ではない
getauclassent_r(3)	MT-安全
getauclassnam(3)	安全ではない
getauclassnam_r(3)	MT-安全
getauditflags(3)	MT-安全
getauditflagsbin(3)	MT-安全
getauditflagschar(3)	MT-安全
getauevent(3)	安全ではない
getauevent_r(3)	MT-安全
getauevnam(3)	安全ではない
getauevnam_r(3)	MT-安全
getauevnonam(3)	MT-安全
getauevnum(3)	安全ではない
getauevnum_r(3)	MT-安全
getauuserent(3)	安全ではない
getauusernam(3)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
getbegyx(3X)	安全ではない
getc(3S)	MT-安全
getch(3X)	安全ではない
getchar(3S)	MT-安全
getcwd(3C)	安全
getdate(3C)	MT-安全
getenv(3C)	安全
getfauditflags(3)	MT-安全
getgrent(3C)	安全ではない。 getgrent_r() を使用
getgrgid(3C)	安全ではない。 getgrgid_r() を使用
getgrnam(3C)	安全ではない。 getgrnam_r() を使用
gethostbyaddr(3N)	安全ではない。 gethostbyaddr_r() を使用
gethostbyname(3N)	安全ではない。 gethostbyname_r() を使用
gethrtime(3C)	MT-安全
gethrvtime(3C)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
getlogin(3C)	安全ではない getlogin_r() を使用
getmaxyx(3X)	安全ではない
getmntany(3C)	安全
getmntent(3C)	安全
getnetbyaddr(3N)	安全ではない。 getnetbyaddr_r() を使用
getnetbyname(3N)	安全ではない。 getnetbyname_r() を使用
getnetconfig(3N)	MT-安全
getnetconfigent(3N)	MT-安全
getnetgrent(3N)	安全ではない。 getnetgrent_r() を使用
getnetname(3N)	MT-安全
getnetpath(3N)	MT-安全
getnwstr(3X)	安全ではない
getopt(3C)	安全ではない
getparyx(3X)	安全ではない
getpass(3C)	安全ではない
getpeername(3N)	安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
getprotobyname(3N)	安全ではない。 getprotobyname_r() を使用
getprotobynumber(3N)	安全ではない。 getprotobynumber_r() を使用
getprotoent(3N)	安全ではない。 getprotoent_r() を使用
getpublickey(3N)	安全
getpw(3C)	安全
getpwent(3C)	安全ではない。 getpwent_r() を使用
getpwnam(3C)	安全ではない。 getpwnam_r() を使用
getpwuid(3C)	安全ではない。 getpwuid_r() を使用
getrpcbyname(3N)	安全ではない。 getrpcbyname_r() を使用
getrpcbynumber(3N)	安全ではない。 getrpcbynumber_r() を使用
getrpcent(3N)	安全ではない。 getrpcent_r() を使用
getrpcport(3N)	安全ではない
gets(3S)	MT-安全
getsecretkey(3N)	安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
getservbyname(3N)	安全ではない。 getservbyname_r() を使用
getservbyport(3N)	安全ではない。 getservbyport_r() を使用
getservent(3N)	安全ではない。 getservent_r() を使用
getsockname(3N)	安全
getsockopt(3N)	安全
getspent(3C)	安全ではない。 getspent_r() を使用
getspnam(3C)	安全ではない。 getspnam_r() を使用
getstr(3X)	安全ではない
getsubopt(3C)	MT-安全
getsyx(3X)	安全ではない
gettext(3I)	例外付きで安全
gettimeofday(3C)	MT-安全
gettxt(3C)	例外付きで安全
getutent(3C)	安全ではない
getutid(3C)	安全ではない
getutline(3C)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
getutmp(3C)	安全ではない
getutmpx(3C)	安全ではない
getutxent(3C)	安全ではない
getutxid(3C)	安全ではない
getutxline(3C)	安全ではない
getvfsany(3C)	安全
getvfsent(3C)	安全
getvfsfile(3C)	安全
getvfsspec(3C)	安全
getw(3S)	MT-安全
getwc(3I)	MT-安全
getwch(3X)	安全ではない
getwchar(3I)	MT-安全
getwidth(3I)	例外付きで MT-安全
getwin(3X)	安全ではない
getws(3I)	MT-安全
getwstr(3X)	安全ではない
getyx(3X)	安全ではない
get_myaddress(3N)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
gmatch(3G)	MT-安全
gmtime(3C)	安全ではない。 gmtime_r() を使用
grantpt(3C)	安全
gsignal(3C)	安全ではない
halfdelay(3X)	安全ではない
hasmntopt(3C)	安全
has_colors(3X)	安全ではない
has_ic(3X)	安全ではない
has_il(3X)	安全ではない
havedisk(3N)	MT-安全
hcreate(3C)	安全
hdestroy(3C)	安全
hide_panel(3X)	安全ではない
host2netname(3N)	MT-安全
hsearch(3C)	安全
htonl(3N)	安全
htons(3N)	安全
hyperbolic(3M)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
hypot(3M)	MT-安全
iconv(3)	MT-安全
iconv_close(3)	MT-安全
iconv_open(3)	MT-安全
idcok(3X)	安全ではない
idlok(3X)	安全ではない
ieee_functions(3M)	MT-安全
ieee_test(3M)	MT-安全
ilogb(3M)	MT-安全
immedok(3X)	安全ではない
inch(3X)	安全ではない
inchnstr(3X)	安全ではない
inchstr(3X)	安全ではない
inet(3N)	安全
inet_addr(3N)	安全
inet_lnaof(3N)	安全
inet_makeaddr(3N)	安全
inet_netof(3N)	安全
inet_network(3N)	安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
inet_ntoa(3N)	安全
initgroups(3C)	安全ではない
initscr(3X)	安全ではない
init_color(3X)	安全ではない
init_pair(3X)	安全ではない
innstr(3X)	安全ではない
innwstr(3X)	安全ではない
insch(3X)	安全ではない
insdelln(3X)	安全ではない
insertln(3X)	安全ではない
insnstr(3X)	安全ではない
insnwstr(3X)	安全ではない
insque(3C)	安全ではない
insstr(3X)	安全ではない
instr(3X)	安全ではない
inswch(3X)	安全ではない
inswstr(3X)	安全ではない
intrflush(3X)	安全ではない
inwch(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
inwchnstr(3X)	安全ではない
inwchstr(3X)	安全ではない
inwstr(3X)	安全ではない
isalnum(3C)	例外付きで MT-安全
isalpha(3C)	例外付きで MT-安全
isascii(3C)	例外付きで MT-安全
isastream(3C)	MT-安全
iscntrl(3C)	例外付きで MT-安全
isdigit(3C)	例外付きで MT-安全
isencrypt(3G)	MT-安全
isendwin(3X)	安全ではない
isenglish(3I)	例外付きで MT-安全
isgraph(3C)	例外付きで MT-安全
isideogram(3I)	例外付きで MT-安全
islower(3C)	例外付きで MT-安全
isnan(3C)	MT-安全
isnan(3M)	MT-安全
isnand(3C)	MT-安全
isnanf(3C)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
isnumber(3I)	例外付きで MT-安全
isphonogram(3I)	例外付きで MT-安全
isprint(3C)	例外付きで MT-安全
ispunct(3C)	例外付きで MT-安全
isspace(3C)	例外付きで MT-安全
isspecial(3I)	例外付きで MT-安全
isupper(3C)	例外付きで MT-安全
iswalnum(3I)	例外付きで MT-安全
iswalpha(3I)	例外付きで MT-安全
iswascii(3I)	例外付きで MT-安全
iswcntrl(3I)	例外付きで MT-安全
iswctype(3I)	MT-安全
iswdigit(3I)	例外付きで MT-安全
iswgraph(3I)	例外付きで MT-安全
iswlower(3I)	例外付きで MT-安全
iswprint(3I)	例外付きで MT-安全
iswpunct(3I)	例外付きで MT-安全
iswspace(3I)	例外付きで MT-安全
iswupper(3I)	例外付きで MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
iswxdigit(3I)	例外付きで MT-安全
isxdigit(3C)	例外付きで MT-安全
is_linetouched(3X)	安全ではない
is_wintouched(3X)	安全ではない
item_count(3X)	安全ではない
item_description(3X)	安全ではない
item_index(3X)	安全ではない
item_init(3X)	安全ではない
item_name(3X)	安全ではない
item_opts(3X)	安全ではない
item_opts_off(3X)	安全ではない
item_opts_on(3X)	安全ではない
item_term(3X)	安全ではない
item_userptr(3X)	安全ではない
item_value(3X)	安全ではない
item_visible(3X)	安全ではない
j0(3M)	MT-安全
j1(3M)	MT-安全
jn(3M)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
jrand48(3C)	安全
kerberos(3N)	安全ではない
kerberos_rpc(3N)	安全ではない
keyname(3X)	安全ではない
keypad(3X)	安全ではない
key_decryptsession(3N)	MT-安全
key_encryptsession(3N)	MT-安全
key_gendes(3N)	MT-安全
key_secretkey_is_set(3N)	MT-安全
key_setsecret(3N)	MT-安全
killchar(3X)	安全ではない
krb_get_admhst(3N)	安全ではない
krb_get_cred(3N)	安全ではない
krb_get_krbhst(3N)	安全ではない
krb_get_lrealm(3N)	安全ではない
krb_get_phost(3N)	安全ではない
krb_kntoln(3N)	安全ではない
krb_mk_err(3N)	安全ではない
krb_mk_req(3N)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
krb_mk_safe(3N)	安全ではない
krb_net_read(3N)	安全ではない
krb_net_write(3N)	安全ではない
krb_rd_err(3N)	安全ではない
krb_rd_req(3N)	安全ではない
krb_rd_safe(3N)	安全ではない
krb_realmofhost(3N)	安全ではない
krb_recvauth(3N)	安全ではない
krb_sendauth(3N)	安全ではない
krb_set_key(3N)	安全ではない
krb_set_tkt_string(3N)	安全ではない
kvm_close(3K)	安全ではない
kvm_getcmd(3K)	安全ではない
kvm_getproc(3K)	安全ではない
kvm_getu(3K)	安全ではない
kvm_kread(3K)	安全ではない
kvm_kwrite(3K)	安全ではない
kvm_nextproc(3K)	安全ではない
kvm_nlist(3K)	安全ではない

表 C–1 ライブラリルーチンの MT-安全レベル 続く

ライブラリルーチン	安全レベル
kvm_open(3K)	安全ではない
kvm_read(3K)	安全ではない
kvm_setproc(3K)	安全ではない
kvm_uread(3K)	安全ではない
kvm_uwrite(3K)	安全ではない
kvm_write(3K)	安全ではない
l64a(3C)	MT-安全
label(3)	安全
labs(3C)	MT-安全
lckpwdf(3C)	MT-安全
lcong48(3C)	安全
ldexp(3C)	MT-安全
ldiv(3C)	MT-安全
leaveok(3X)	安全ではない
lfind(3C)	安全
lfmt(3C)	MT-安全
lgamma(3M)	安全ではない。lgamma_r() を使用
libpthread(3T)	Fork1-安全、MT-安全、非同期シグナル安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
libthread(3T)	Fork1-安全、MT-安全、非同期シグナル安全
line(3)	安全
link_field(3X)	安全ではない
link_fieldtype(3X)	安全ではない
linmod(3)	安全
lio_listio(3R)	MT-安全
listen(3N)	安全
llabs(3C)	MT-安全
lldiv(3C)	MT-安全
lltostr(3C)	MT-安全
localeconv(3C)	例外付きで安全
localtime(3C)	安全ではない。localtime_r() を使用
lockf(3C)	MT-安全
log(3M)	MT-安全
log10(3M)	MT-安全
log1p(3M)	MT-安全
logb(3C)	MT-安全
logb(3M)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
longjmp(3C)	安全ではない
longname(3X)	安全ではない
lrand48(3C)	安全
lsearch(3C)	安全
madvise(3)	MT-安全
maillock(3X)	安全ではない
major(3C)	MT-安全
makecontext(3C)	MT-安全
makedev(3C)	MT-安全
mallinfo(3X)	安全
malloc(3C)	安全
malloc(3X)	安全
mallopt(3X)	安全
mapmalloc(3X)	安全
matherr(3M)	MT-安全
mbchar(3C)	例外付きで MT-安全
mblen(3C)	例外付きで MT-安全
mbstowcs(3C)	例外付きで MT-安全
mbstring(3C)	例外付きで MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
mbtowc(3C)	例外付きで MT-安全
media_findname(3X)	MT-安全ではない
media_getattr(3X)	MT-安全
media_setattr(3X)	MT-安全
memalign(3C)	安全
memccpy(3C)	MT-安全
memchr(3C)	MT-安全
memcmp(3C)	MT-Safe
memcpy(3C)	MT-安全
memmove(3C)	MT-安全
memory(3C)	MT-安全
memset(3C)	MT-安全
menus(3X)	安全ではない
menu_attributes(3X)	安全ではない
menu_back(3X)	安全ではない
menu_cursor(3X)	安全ではない
menu_driver(3X)	安全ではない
menu_fore(3X)	安全ではない
menu_format(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
menu_grey(3X)	安全ではない
menu_hook(3X)	安全ではない
menu_init(3X)	安全ではない
menu_items(3X)	安全ではない
menu_item_current(3X)	安全ではない
menu_item_name(3X)	安全ではない
menu_item_new(3X)	安全ではない
menu_item_opts(3X)	安全ではない
menu_item_userptr(3X)	安全ではない
menu_item_value(3X)	安全ではない
menu_item_visible(3X)	安全ではない
menu_mark(3X)	安全ではない
menu_new(3X)	安全ではない
menu_opts(3X)	安全ではない
menu_opts_off(3X)	安全ではない
menu_opts_on(3X)	安全ではない
menu_pad(3X)	安全ではない
menu_pattern(3X)	安全ではない
menu_post(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
menu_sub(3X)	安全ではない
menu_term(3X)	安全ではない
menu_userptr(3X)	安全ではない
menu_win(3X)	安全ではない
meta(3X)	安全ではない
minor(3C)	MT-安全
mkdirp(3G)	MT-安全
mkfifo(3C)	MT-安全、非同期シグナル安全
mktemp(3C)	安全
mktime(3C)	安全ではない
mlock(3C)	MT-安全
monitor(3C)	安全
move(3)	安全
move(3X)	安全ではない
movenextch(3X)	安全ではない
moveprevch(3X)	安全ではない
move_field(3X)	安全ではない
move_panel(3X)	安全ではない
mq_close(3R)	MT-安全

表 C–1 ライブラリルーチンの MT-安全レベル 続く

ライブラリルーチン	安全レベル
mq_getattr(3R)	MT-安全
mq_notify(3R)	MT-安全
mq_open(3R)	MT-安全
mq_receive(3R)	MT-安全
mq_send(3R)	MT-安全
mq_setattr(3R)	MT-安全
mq_unlink(3R)	MT-安全
mrand48(3C)	安全
msync(3C)	MT-安全
munlock(3C)	MT-安全
munlockall(3C)	MT-安全
mutex(3T)	MT-安全
mutex_destroy(3T)	MT-安全
mutex_init(3T)	MT-安全
mutex_lock(3T)	MT-安全
mutex_trylock(3T)	MT-安全
mutex_unlock(3T)	MT-安全
mvaddch(3X)	安全ではない
mvaddchnstr(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
mvaddchstr(3X)	安全ではない
mvaddnstr(3X)	安全ではない
mvaddnwstr(3X)	安全ではない
mvaddstr(3X)	安全ではない
mvaddwch(3X)	安全ではない
mvaddwchnstr(3X)	安全ではない
mvaddwchstr(3X)	安全ではない
mvaddwstr(3X)	安全ではない
mvcur(3X)	安全ではない
mvdelch(3X)	安全ではない
mvderwin(3X)	安全ではない
mvgetch(3X)	安全ではない
mvgetnwstr(3X)	安全ではない
mvgetstr(3X)	安全ではない
mvgetwch(3X)	安全ではない
mvgetwstr(3X)	安全ではない
mvinch(3X)	安全ではない
mvinchnstr(3X)	安全ではない
mvinchstr(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
mvinnstr(3X)	安全ではない
mvinnwstr(3X)	安全ではない
mvinsch(3X)	安全ではない
mvinsnstr(3X)	安全ではない
mvinsnwstr(3X)	安全ではない
mvinsstr(3X)	安全ではない
mvinstr(3X)	安全ではない
mvinswch(3X)	安全ではない
mvinswstr(3X)	安全ではない
mvinwch(3X)	安全ではない
mvinwchnstr(3X)	安全ではない
mvinwchstr(3X)	安全ではない
mvinwstr(3X)	安全ではない
mvprintw(3X)	安全ではない
mvscanw(3X)	安全ではない
mvwaddch(3X)	安全ではない
mvwaddchnstr(3X)	安全ではない
mvwaddchstr(3X)	安全ではない
mvwaddnstr(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
mvwaddnwstr(3X)	安全ではない
mvwaddstr(3X)	安全ではない
mvwaddwch(3X)	安全ではない
mvwaddwchnstr(3X)	安全ではない
mvwaddwchstr(3X)	安全ではない
mvwaddwstr(3X)	安全ではない
mvwdelch(3X)	安全ではない
mvwgetch(3X)	安全ではない
mvwgetnwstr(3X)	安全ではない
mvwgetstr(3X)	安全ではない
mvwgetwch(3X)	安全ではない
mvwgetwstr(3X)	安全ではない
mvwin(3X)	安全ではない
mvwinch(3X)	安全ではない
mvwinchnstr(3X)	安全ではない
mvwinchstr(3X)	安全ではない
mvwinnstr(3X)	安全ではない
mvwinnwstr(3X)	安全ではない
mvwinsch(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
mvwinsnstr(3X)	安全ではない
mvwinsnwstr(3X)	安全ではない
mvwinsstr(3X)	安全ではない
mvwinstr(3X)	安全ではない
mvwinswch(3X)	安全ではない
mvwinswstr(3X)	安全ではない
mvwinwch(3X)	安全ではない
mvwinwchnstr(3X)	安全ではない
mvwinwchstr(3X)	安全ではない
mvwinwstr(3X)	安全ではない
mvwprintw(3X)	安全ではない
mvwscanw(3X)	安全ではない
nanosleep(3R)	MT-安全
napms(3X)	安全ではない
nc_perror(3N)	MT-安全
nc_sperror(3N)	MT-安全
ndbm(3)	安全ではない
netdir(3N)	MT-安全
netdir_free(3N)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
netdir_getbyaddr(3N)	MT-安全
netdir_getbyname(3N)	MT-安全
netdir_mergeaddr(3N)	MT-安全
netdir_options(3N)	MT-安全
netdir_perror(3N)	MT-安全
netdir_sperror(3N)	MT-安全
netname2host(3N)	MT-安全
netname2user(3N)	MT-安全
newpad(3X)	安全ではない
newterm(3X)	安全ではない
newwin(3X)	安全ではない
new_field(3X)	安全ではない
new_fieldtype(3X)	安全ではない
new_form(3X)	安全ではない
new_item(3X)	安全ではない
new_menu(3X)	安全ではない
new_page(3X)	安全ではない
new_panel(3X)	安全ではない
nextafter(3C)	MT-安全

表 C–1 ライブラリルーチンの MT-安全レベル 続く

ライブラリルーチン	安全レベル
nextafter(3M)	MT-安全
nftw(3C)	例外付きで安全
nis_add(3N)	MT-安全
nis_clone_object(3N)	安全
nis_creategroup(3N)	MT-安全
nis_db(3N)	安全ではない
nis_destroygroup(3N)	MT-安全
nis_destroy_object(3N)	安全
nis_dir_cmp(3N)	安全
nis_domain_of(3N)	安全
nis_error(3N)	安全
nis_first_entry(3N)	MT-安全
nis_freenames(3N)	安全
nis_freeresult(3N)	MT-安全
nis_freeservlist(3N)	MT-Safe
nis_freetags(3N)	MT-安全
nis_getnames(3N)	安全
nis_getservlist(3N)	MT-安全
nis_groups(3N)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
nis_ismember(3N)	MT-安全
nis_leaf_of(3N)	安全
nis_lerror(3N)	安全
nis_list(3N)	MT-安全
nis_local_directory(3N)	MT-安全
nis_local_group(3N)	MT-安全
nis_local_host(3N)	MT-安全
nis_local_names(3N)	MT-安全
nis_local_principal(3N)	MT-安全
nis_lookup(3N)	MT-安全
nis_map_group(3N)	MT-安全
nis_mkdir(3N)	MT-安全
nis_modify(3N)	MT-安全
nis_modify_entry(3N)	MT-安全
nis_names(3N)	MT-安全
nis_name_of(3N)	安全
nis_next_entry(3N)	MT-安全
nis_perror(3N)	安全
nis_ping(3N)	MT-安全

表 C–1 ライブラリルーチンの MT-安全レベル 続く

ライブラリルーチン	安全レベル
nis_print_group_entry(3N)	MT-安全
nis_print_object(3N)	安全
nis_remove(3N)	MT-安全
nis_removemember(3N)	MT-安全
nis_remove_entry(3N)	MT-安全
nis_rmdir(3N)	MT-安全
nis_server(3N)	MT-安全
nis_servstate(3N)	MT-安全
nis_sperrno(3N)	安全
nis_sperror(3N)	安全
nis_sperror_r(3N)	安全
nis_stats(3N)	MT-安全
nis_subr(3N)	安全
nis_tables(3N)	MT-安全
nis_verifygroup(3N)	MT-安全
nl(3X)	安全ではない
nlist(3E)	安全
nlsgetcall(3N)	安全ではない
nlsprovider(3N)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
nlsrequest(3N)	安全ではない
nl_langinfo(3C)	例外付きで安全
nocbreak(3X)	安全ではない
nodelay(3X)	安全ではない
noecho(3X)	安全ではない
nonl(3X)	安全ではない
noqiflush(3X)	安全ではない
noraw(3X)	安全ではない
NOTE(3X)	安全
notimeout(3X)	安全ではない
nrand48(3C)	安全
ntohl(3N)	安全
ntohs(3N)	安全
offsetof(3C)	MT-安全
opendir(3C)	安全
openlog(3)	安全
openpl(3)	安全
openvt(3)	安全
overlay(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
overwrite(3X)	安全ではない
p2close(3G)	安全ではない
p2open(3G)	安全ではない
pair_content(3X)	安全ではない
panels(3X)	安全ではない
panel_above(3X)	安全ではない
panel_below(3X)	安全ではない
panel_hidden(3X)	安全ではない
panel_move(3X)	安全ではない
panel_new(3X)	安全ではない
panel_show(3X)	安全ではない
panel_top(3X)	安全ではない
panel_update(3X)	安全ではない
panel_userptr(3X)	安全ではない
panel_window(3X)	安全ではない
pathfind(3G)	MT-安全
pclose(3S)	安全ではない
pechochar(3X)	安全ではない
pechowchar(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
perror(3C)	MT-安全
pfmt(3C)	MT-安全
plot(3)	安全
pmap_getmaps(3N)	安全ではない
pmap_getport(3N)	安全ではない
pmap_rmtcall(3N)	安全ではない
pmap_set(3N)	安全ではない
pmap_unset(3N)	安全ではない
pnoutrefresh(3X)	安全ではない
point(3)	安全
popen(3S)	安全ではない
post_form(3X)	安全ではない
post_menu(3X)	安全ではない
pos_form_cursor(3X)	安全ではない
pos_menu_cursor(3X)	安全ではない
pow(3M)	MT-安全
prefresh(3X)	安全ではない
printf(3S)	MT-安全 (setlocale() を除く)
printw(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
psiginfo(3C)	安全
psignal(3C)	安全
pthreads(3T)	Fork1-安全、MT-安全、非同期シグナル安全
pthread_atfork(3T)	MT-安全
pthread_attr_destroy(3T)	MT-安全
pthread_attr_getdetachstate(3T)	MT-安全
pthread_attr_getinheritsched(3T)	MT-安全
pthread_attr_getschedparam(3T)	MT-安全
pthread_attr_getschedpolicy(3T)	MT-安全
pthread_attr_getscope(3T)	MT-安全
pthread_attr_getstackaddr(3T)	MT-安全
pthread_attr_getstacksize(3T)	MT-安全
pthread_attr_init(3T)	MT-安全
pthread_attr_setdetachstate(3T)	MT-安全
pthread_attr_setscope(3T)	MT-安全
pthread_attr_setstackaddr(3T)	MT-安全
pthread_attr_setstacksize(3T)	MT-安全
pthread_cancel(3T)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
pthread_cleanup_pop(3T)	MT-安全
pthread_cleanup_push(3T)	MT-安全
pthread_condattr_destroy(3T)	MT-安全
pthread_condattr_getpshared(3T)	MT-安全
pthread_condattr_init(3T)	MT-安全
pthread_condattr_setpshared(3T)	MT-安全
pthread_cond_broadcast(3T)	MT-安全
pthread_cond_destroy(3T)	MT-安全
pthread_cond_init(3T)	MT-安全
pthread_cond_signal(3T)	MT-安全
pthread_cond_timedwait(3T)	MT-安全
pthread_cond_wait(3T)	MT-安全
pthread_create(3T)	MT-安全
pthread_detach(3T)	MT-安全
pthread_equal(3T)	MT-安全
pthread_exit(3T)	MT-安全
pthread_getschedparam(3T)	MT-安全
pthread_getspecific(3T)	MT-安全
pthread_join(3T)	MT-安全

表 C–1 ライブラリルーチンの MT-安全レベル 続く

ライブラリルーチン	安全レベル
pthread_key_create(3T)	MT-安全
pthread_key_delete(3T)	MT-安全
pthread_kill(3T)	MT-安全、非同期シグナル安全
pthread_mutexattr_destroy(3T)	MT-安全
pthread_mutexattr_getprioceiling(3T)	MT-安全
pthread_mutexattr_getprotocol(3T)	MT-安全
pthread_mutexattr_getpshared(3T)	MT-安全
pthread_mutexattr_init(3T)	MT-安全
pthread_mutexattr_setprioceiling(3T)	MT-安全
pthread_mutexattr_setprotocol(3T)	MT-安全
pthread_mutexattr_setpshared(3T)	MT-安全
pthread_mutex_destroy(3T)	MT-安全
pthread_mutex_getprioceiling(3T)	MT-安全
pthread_mutex_init(3T)	MT-安全
pthread_mutex_lock(3T)	MT-安全
pthread_mutex_setprioceiling(3T)	MT-安全
pthread_mutex_trylock(3T)	MT-安全
pthread_mutex_unlock(3T)	MT-安全
pthread_once(3T)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
pthread_self(3T)	MT-安全
pthread_setcancelstate(3T)	MT-安全
pthread_setcanceltype(3T)	MT-安全
pthread_setschedparam(3T)	MT-安全
pthread_setspecific(3T)	MT-安全
pthread_sigmask(3T)	MT-安全、非同期シグナル安全
pthread_testcancel(3T)	MT-安全
ptsname(3C)	安全
publickey(3N)	安全
putc(3S)	MT-安全
putchar(3S)	MT-安全
putenv(3C)	安全
putmntent(3C)	安全
putp(3X)	安全ではない
putpwent(3C)	安全ではない
puts(3S)	MT-安全
putspent(3C)	安全ではない
pututline(3C)	安全ではない
pututxline(3C)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
putw(3S)	MT-安全
putwc(3I)	MT-安全
putwchar(3I)	MT-安全
putwin(3X)	安全ではない
putws(3I)	MT-安全
qeconvert(3)	MT-安全
qfconvert(3)	MT-安全
qgconvert(3)	MT-安全
qiflush(3X)	安全ではない
qsort(3C)	安全
quadruple_to_decimal(3)	MT-安全
rac_drop(3N)	安全ではない
rac_poll(3N)	安全ではない
rac_recv(3N)	安全ではない
rac_send(3N)	安全ではない
raise(3C)	MT-安全
rand(3C)	安全ではない。rand_r() を使用
random(3C)	安全ではない
raw(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
rcmd(3N)	安全ではない
readdir(3C)	安全ではない。readdir_r() を使用
read_vtoc(3X)	安全ではない
realloc(3C)	安全
realloc(3X)	安全
realpath(3C)	MT-安全
recv(3N)	安全
recvfrom(3N)	安全
recvmsg(3N)	安全
redrawwin(3X)	安全ではない
refresh(3X)	安全ではない
regcmp(3G)	MT-安全
regcomp(3C)	MT-安全
regerror(3C)	MT-安全
regex(3G)	MT-安全
regexec(3C)	MT-安全
regexpr(3G)	MT-安全
regfree(3C)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
registerrpc(3N)	安全ではない
remainder(3M)	MT-安全
remove(3C)	MT-安全
remque(3C)	安全ではない
replace_panel(3X)	安全ではない
resetty(3X)	安全ではない
reset_prog_mode(3X)	安全ではない
reset_shell_mode(3X)	安全ではない
resolver(3N)	安全ではない
restartterm(3X)	安全ではない
res_init(3N)	安全ではない
res_mkquery(3N)	安全ではない
res_search(3N)	安全ではない
res_send(3N)	安全ではない
rewind(3S)	MT-安全
rewinddir(3C)	安全
rexec(3N)	安全ではない
rint(3M)	MT-安全
ripoffline(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
rmdirp(3G)	MT-安全
rnusers(3N)	MT-安全
rpc(3N)	例外付きで MT-安全
rpcbind(3N)	MT-安全
rpcb_getaddr(3N)	MT-安全
rpcb_getmaps(3N)	MT-安全
rpcb_gettime(3N)	MT-安全
rpcb_rmtcall(3N)	MT-安全
rpc_broadcast_exp(3N)	MT-安全
rpc_call(3N)	MT-安全
rpc_clnt_auth(3N)	MT-安全
rpc_clnt_calls(3N)	MT-安全
rpc_clnt_create(3N)	MT-安全
rpc_control(3N)	MT-安全
rpc_createerr(3N)	MT-安全
rpc_rac(3N)	安全ではない
rpc_reg(3N)	MT-安全
rpc_soc(3N)	安全ではない
rpc_svc_create(3N)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
rpc_svc_err(3N)	MT-安全
rpc_svc_reg(3N)	MT-安全
rpc_xdr(3N)	安全
rresvport(3N)	安全ではない
rstat(3N)	MT-安全
ruserok(3N)	安全ではない
rusers(3N)	MT-安全
rwall(3N)	MT-安全
rwlock(3T)	MT-安全
rwlock_destroy(3T)	MT-安全
rwlock_init(3T)	MT-安全
rw_rdlock(3T)	MT-安全
rw_tryrdlock(3T)	MT-安全
rw_trywrlock(3T)	MT-安全
rw_unlock(3T)	MT-安全
rw_wrlock(3T)	MT-安全
savetty(3X)	安全ではない
scalb(3C)	MT-安全
scalb(3M)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
scalbn(3M)	MT-安全
scale_form(3X)	安全ではない
scale_menu(3X)	安全ではない
scanf(3S)	MT-安全
scanw(3X)	安全ではない
sched_getparam(3R)	MT-安全
sched_getscheduler(3R)	MT-安全
sched_get_priority_max(3R)	MT-安全
sched_get_priority_min(3R)	MT-安全
sched_rr_get_interval(3R)	MT-安全
sched_setparam(3R)	MT-安全
sched_setscheduler(3R)	MT-安全
sched_yield(3R)	MT-安全
scrl(3X)	安全ではない
scroll(3X)	安全ではない
scrollok(3X)	安全ではない
scr_dump(3X)	安全ではない
scr_init(3X)	安全ではない
scr_restore(3X)	安全ではない

表 C–1 ライブラリルーチンの MT-安全レベル 続く

ライブラリルーチン	安全レベル
scr_set(3X)	安全ではない
seconvert(3)	MT-安全
secure_rpc(3N)	MT-安全
seed48(3C)	安全
seekdir(3C)	安全
select(3C)	MT-安全
sema_destroy(3T)	MT-安全
sema_init(3T)	MT-安全
sema_post(3T)	MT-安全、非同期シグナル安全
sema_trywait(3T)	MT-安全
sema_wait(3T)	MT-安全
sem_close(3R)	MT-安全
sem_destroy(3R)	MT-安全
sem_getvalue(3R)	MT-Safe
sem_init(3R)	MT-Safe
sem_open(3R)	MT-安全
sem_post(3R)	非同期シグナル安全
sem_trywait(3R)	MT-安全
sem_unlink(3R)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
sem_wait(3R)	MT-安全
send(3N)	安全
sendmsg(3N)	安全
sendto(3N)	安全
setac(3)	安全
setauclass(3)	MT-安全
setauevent(3)	MT-安全
setauuser(3)	MT-安全
setbuf(3S)	MT-安全
setcat(3C)	MT-安全
setjmp(3C)	安全ではない
setkey(3C)	安全
setlabel(3C)	MT-安全
setlocale(3C)	例外付きで安全
setlogmask(3)	安全
setnetconfig(3N)	MT-安全
setnetpath(3N)	MT-安全
setscrreg(3X)	安全ではない
setsockopt(3N)	安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
setsyx(3X)	安全ではない
setterm(3X)	安全ではない
settimeofday(3C)	MT-安全
setupterm(3X)	安全ではない
setutent(3C)	安全ではない
setvbuf(3S)	MT-安全
set_current_field(3X)	安全ではない
set_current_item(3X)	安全ではない
set_curterm(3X)	安全ではない
set_fieldtype_arg(3X)	安全ではない
set_fieldtype_choice(3X)	安全ではない
set_field_back(3X)	安全ではない
set_field_buffer(3X)	安全ではない
set_field_fore(3X)	安全ではない
set_field_init(3X)	安全ではない
set_field_just(3X)	安全ではない
set_field_opts(3X)	安全ではない
set_field_pad(3X)	安全ではない
set_field_status(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
set_field_term(3X)	安全ではない
set_field_type(3X)	安全ではない
set_field_userptr(3X)	安全ではない
set_form_fields(3X)	安全ではない
set_form_init(3X)	安全ではない
set_form_opts(3X)	安全ではない
set_form_page(3X)	安全ではない
set_form_sub(3X)	安全ではない
set_form_term(3X)	安全ではない
set_form_userptr(3X)	安全ではない
set_form_win(3X)	安全ではない
set_item_init(3X)	安全ではない
set_item_opts(3X)	安全ではない
set_item_term(3X)	安全ではない
set_item_userptr(3X)	安全ではない
set_item_value(3X)	安全ではない
set_max_field(3X)	安全ではない
set_menu_back(3X)	安全ではない
set_menu_init(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
set_menu_items(3X)	安全ではない
set_menu_mark(3X)	安全ではない
set_menu_opts(3X)	安全ではない
set_menu_pad(3X)	安全ではない
set_menu_pattern(3X)	安全ではない
set_menu_sub(3X)	安全ではない
set_menu_term(3X)	安全ではない
set_menu_userptr(3X)	安全ではない
set_menu_win(3X)	安全ではない
set_new_page(3X)	安全ではない
set_panel_userptr(3X)	安全ではない
set_term(3X)	安全ではない
set_top_row(3X)	安全ではない
sfconvert(3)	MT-安全
sgconvert(3)	MT-安全
shm_open(3R)	MT-安全
shm_unlink(3R)	MT-安全
show_panel(3X)	安全ではない
shutdown(3N)	安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
sigaddset(3C)	MT-安全、非同期シグナル安全
sigdelset(3C)	MT-安全、非同期シグナル安全
sigemptyset(3C)	MT-安全、非同期シグナル安全
sigfillset(3C)	MT-安全、非同期シグナル安全
sigfpe(3)	安全
sigismember(3C)	MT-安全、非同期シグナル安全
siglongjmp(3C)	安全ではない
significand(3M)	MT-安全
sigqueue(3R)	非同期シグナル安全
sigsetjmp(3C)	安全ではない
sigsetops(3C)	MT-安全、非同期シグナル安全
sigtimedwait(3R)	非同期シグナル安全
sigwaitinfo(3R)	非同期シグナル安全
sin(3M)	MT-安全
single_to_decimal(3)	MT-安全
sinh(3M)	MT-安全
sleep(3B)	非同期シグナル安全
sleep(3C)	安全
slk_attroff(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
slk_attron(3X)	安全ではない
slk_attrset(3X)	安全ではない
slk_clear(3X)	安全ではない
slk_init(3X)	安全ではない
slk_label(3X)	安全ではない
slk_noutrefresh(3X)	安全ではない
slk_refresh(3X)	安全ではない
slk_restore(3X)	安全ではない
slk_set(3X)	安全ではない
slk_touch(3X)	安全ではない
socket(3N)	安全
socketpair(3N)	安全
space(3)	安全
spray(3N)	安全ではない
sprintf(3S)	MT-安全
sqrt(3M)	MT-安全
srand(3C)	安全ではない
srand48(3C)	安全
srandom(3C)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
sscanf(3S)	MT-安全
ssignal(3C)	安全ではない
standend(3X)	安全ではない
standout(3X)	安全ではない
start_color(3X)	安全ではない
step(3G)	MT-安全
str(3G)	MT-安全
strcadd(3G)	MT-安全
strcasecmp(3C)	安全
strcat(3C)	安全
strccpy(3G)	MT-安全
strchr(3C)	安全
strcmp(3C)	安全
strcoll(3C)	例外付きで安全
strcpy(3C)	安全
strcspn(3C)	安全
strdup(3C)	安全
streadd(3G)	MT-安全
strecpy(3G)	MT-安全

表 C–1 ライブラリルーチンの MT-安全レベル 続く

ライブラリルーチン	安全レベル
strerror(3C)	安全
strfind(3G)	MT-安全
strfmon(3C)	MT-安全
strftime(3C)	MT-安全
string(3C)	安全
string_to_decimal(3)	MT-安全
strlen(3C)	安全
strncasecmp(3C)	安全
strncat(3C)	安全
strncmp(3C)	安全
strncpy(3C)	安全
strpbrk(3C)	安全
strptime(3C)	MT-安全
strrchr(3C)	安全
strrspn(3G)	MT-安全
strsignal(3C)	安全
strspn(3C)	安全
strstr(3C)	安全
strtod(3C)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
strtok(3C)	安全ではない。 strtok_r() を使用
strtol(3C)	MT-安全
strtoll(3C)	MT-安全
strtoul(3C)	MT-安全
strtoull(3C)	MT-安全
strtrns(3G)	MT-安全
strxfrm(3C)	例外付きで安全
subpad(3X)	安全ではない
subwin(3X)	安全ではない
svcerr_auth(3N)	MT-安全
svcerr_decode(3N)	MT-安全
svcerr_noproc(3N)	MT-安全
svcerr_noprog(3N)	MT-安全
svcerr_progvers(3N)	MT-安全
svcerr_systemerr(3N)	MT-安全
svcerr_weakauth(3N)	MT-安全
svcfd_create(3N)	安全ではない
svcraw_create(3N)	安全ではない

表 C–1 ライブラリルーチンの MT-安全レベル 続く

ライブラリルーチン	安全レベル
svctcp_create(3N)	安全ではない
svcudp_bufcreate(3N)	安全ではない
svcudp_create(3N)	安全ではない
svc_auth_reg(3N)	MT-安全
svc_control(3N)	MT-安全
svc_create(3N)	MT-安全
svc_destroy(3N)	MT-安全
svc_dg_create(3N)	MT-安全
svc_fds(3N)	安全ではない
svc_fd_create(3N)	MT-安全
svc_getcaller(3N)	安全ではない
svc_reg(3N)	MT-安全
svc_register(3N)	安全ではない
svc_tli_create(3N)	MT-安全
svc_tp_create(3N)	MT-安全
svc_unreg(3N)	MT-安全
svc_unregister(3N)	安全ではない
svc_vc_create(3N)	MT-安全
swab(3C)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
swapcontext(3C)	MT-安全
syncok(3X)	安全ではない
sysconf(3C)	MT-安全、非同期シグナル安全
syslog(3)	安全
system(3S)	安全ではない
taddr2uaddr(3N)	MT-安全
tan(3M)	MT-安全
tanh(3M)	MT-安全
tcdrain(3)	MT-安全、非同期シグナル安全
tcflow(3)	MT-安全、非同期シグナル安全
tcflush(3)	MT-安全、非同期シグナル安全
tcgetattr(3)	MT-安全、非同期シグナル安全
tcgetpgrp(3)	MT-安全、非同期シグナル安全
tcgetsid(3)	MT-安全
tcsendbreak(3)	MT-安全、非同期シグナル安全
tcsetattr(3)	MT-安全、非同期シグナル安全
tcsetpgrp(3)	MT-安全、非同期シグナル安全
tcsetpgrp(3C)	MT-安全
tdelete(3C)	安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
telldir(3C)	安全
tempnam(3S)	安全
termattrs(3X)	安全ではない
termname(3X)	安全ではない
textdomain(3I)	例外付きで安全
tfind(3C)	安全
tgetent(3X)	安全ではない
tgetflag(3X)	安全ではない
tgetnum(3X)	安全ではない
tgetstr(3X)	安全ではない
tgoto(3X)	安全ではない
threads(3T)	Fork1-安全、MT-安全、非同期シグナル安全
thr_continue(3T)	MT-安全
thr_create(3T)	MT-安全
thr_exit(3T)	MT-安全
thr_getconcurrency(3T)	MT-安全
thr_getprio(3T)	MT-安全
thr_getspecific(3T)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
thr_join(3T)	MT-安全
thr_keycreate(3T)	MT-安全
thr_kill(3T)	MT-安全、非同期シグナル安全
thr_main(3T)	MT-安全
thr_min_stack(3T)	MT-安全
thr_self(3T)	MT-安全
thr_setconcurrency(3T)	MT-安全
thr_setprio(3T)	MT-安全
thr_setspecific(3T)	MT-安全
thr_sigsetmask(3T)	MT-安全、非同期シグナル安全
thr_stksegment(3T)	MT-安全
thr_suspend(3T)	MT-安全
thr_yield(3T)	MT-安全
tigetflag(3X)	安全ではない
tigetnum(3X)	安全ではない
tigetstr(3X)	安全ではない
timeout(3X)	安全ではない
timer_create(3R)	例外付きで MT-安全
timer_delete(3R)	例外付きで MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
timer_getoverrun(3R)	非同期シグナル安全
timer_gettime(3R)	非同期シグナル安全
timer_settime(3R)	非同期シグナル安全
tmpfile(3S)	安全
tmpnam(3S)	安全ではない。 tmpnam_r() を使用
TNF_DECLARE_RECORD(3X)	MT-安全
TNF_DEFINE_RECORD(3.3X)	MT-安全
TNF_DEFINE_RECORD_1(3X)	MT-安全
TNF_DEFINE_RECORD_2(3X)	MT-安全
TNF_DEFINE_RECORD_4(3X)	MT-安全
TNF_DEFINE_RECORD_5(3X)	MT-安全
TNF_PROBE(3.3X)	MT-安全
TNF_PROBE(3X)	MT-安全
TNF_PROBE_0(3X)	MT-安全
TNF_PROBE_1(3X)	MT-安全
TNF_PROBE_2(3X)	MT-安全
TNF_PROBE_4(3X)	MT-安全
TNF_PROBE_5(3X)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
tnf_process_disable(3X)	MT-安全
tnf_process_enable(3X)	MT-安全
tnf_thread_disable(3X)	MT-安全
tnf_thread_enable(3X)	MT-安全
toascii(3C)	例外付きで MT-安全
tolower(3C)	例外付きで MT-安全
top_panel(3X)	安全ではない
top_row(3X)	安全ではない
touchline(3X)	安全ではない
touchwin(3X)	安全ではない
toupper(3C)	例外付きで MT-安全
towlower(3I)	例外付きで MT-安全
towupper(3I)	例外付きで MT-安全
tparm(3X)	安全ではない
tputs(3X)	安全ではない
trig(3M)	MT-安全
truncate(3C)	MT-安全
tsearch(3C)	安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
ttyname(3C)	安全ではない。 ttyname_r() を使用
ttyslot(3C)	安全
twalk(3C)	安全
typeahead(3X)	安全ではない
t_accept(3N)	MT-安全
t_alloc(3N)	MT-安全
t_bind(3N)	MT-安全
t_close(3N)	MT-安全
t_connect(3N)	MT-安全
t_error(3N)	MT-安全
t_free(3N)	MT-安全
t_getinfo(3N)	MT-安全
t_getstate(3N)	MT-安全
t_listen(3N)	MT-安全
t_look(3N)	MT-安全
t_open(3N)	MT-安全
t_optmgmt(3N)	MT-安全
t_rcv(3N)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
t_rcvconnect(3N)	MT-安全
t_rcvdis(3N)	MT-安全
t_rcvrel(3N)	MT-安全
t_rcvudata(3N)	MT-安全
t_rcvuderr(3N)	MT-安全
t_snd(3N)	MT-安全
t_snddis(3N)	MT-安全
t_sync(3N)	MT-安全
t_unbind(3N)	MT-安全
uaddr2taddr(3N)	MT-安全
ulckpwdf(3C)	MT-安全
ulltostr(3C)	MT-安全
unctrl(3X)	安全ではない
ungetc(3S)	MT-安全
ungetch(3X)	安全ではない
ungetwc(3I)	MT-安全
ungetwch(3X)	安全ではない
unlockpt(3C)	安全
unordered(3C)	MT-安全

表 C–1 ライブラリルーチンの MT-安全レベル 続く

ライブラリルーチン	安全レベル
unpost_form(3X)	安全ではない
unpost_menu(3X)	安全ではない
untouchwin(3X)	安全ではない
update_panels(3X)	安全ではない
updwtmp(3C)	安全ではない
updwtmpx(3C)	安全ではない
user2netname(3N)	MT-安全
use_env(3X)	安全ではない
utmpname(3C)	安全ではない
utmpxname(3C)	安全ではない
valloc(3C)	安全
vfprintf(3S)	非同期シグナル安全
vidattr(3X)	安全ではない
vidputs(3X)	安全ではない
vlfmt(3C)	MT-安全
volmgt_check(3X)	MT-安全
volmgt_inuse(3X)	MT-安全
volmgt_root(3X)	MT-安全
volmgt_running(3X)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
volmgt_symdev(3X)	MT-安全
volmgt_symname(3X)	MT-安全
vpfmt(3C)	MT-安全
vprintf(3S)	非同期シグナル安全
vsprintf(3S)	MT-安全
vsyslog(3)	安全
vwprintw(3X)	安全ではない
vwscanw(3X)	安全ではない
waddch(3X)	安全ではない
waddchnstr(3X)	安全ではない
waddchstr(3X)	安全ではない
waddnstr(3X)	安全ではない
waddnwstr(3X)	安全ではない
waddstr(3X)	安全ではない
waddwch(3X)	安全ではない
waddwchnstr(3X)	安全ではない
waddwchstr(3X)	安全ではない
waddwstr(3X)	安全ではない
wadjcurspos(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
watof(3I)	MT-安全
watoi(3I)	MT-安全
watol(3I)	MT-安全
watoll(3I)	MT-安全
wattroff(3X)	安全ではない
wattron(3X)	安全ではない
wattrset(3X)	安全ではない
wbkgd(3X)	安全ではない
wbkgdset(3X)	安全ではない
wborder(3X)	安全ではない
wclear(3X)	安全ではない
wclrtobot(3X)	安全ではない
wclrtoeol(3X)	安全ではない
wconv(3I)	例外付きで MT-安全
wcscat(3I)	MT-安全
wcschr(3I)	MT-安全
wcscmp(3I)	MT-安全
wcscoll(3I)	MT-安全
wcscpy(3I)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
wcscspn(3I)	MT-安全
wcsetno(3I)	例外付きで MT-安全
wcslen(3I)	MT-安全
wcsncat(3I)	MT-安全
wcsncmp(3I)	MT-安全
wcsncpy(3I)	MT-安全
wcspbrk(3I)	MT-安全
wcsrchr(3I)	MT-安全
wcsspn(3I)	MT-安全
wcstod(3I)	MT-安全
wcstok(3I)	MT-安全
wcstol(3I)	MT-安全
wcstombs(3C)	例外付きで MT-安全
wcstoul(3I)	MT-安全
wcstring(3I)	MT-安全
wcswcs(3I)	MT-安全
wcswidth(3I)	MT-安全
wcsxfrm(3I)	MT-安全
wctomb(3C)	例外付きで MT-安全

表 C–1 ライブラリルーチンの MT-安全レベル 続く

ライブラリルーチン	安全レベル
wctype(3I)	MT-安全
wcursyncup(3X)	安全ではない
wcwidth(3I)	MT-安全
wdelch(3X)	安全ではない
wdeleteln(3X)	安全ではない
wechochar(3X)	安全ではない
wechowchar(3X)	安全ではない
werase(3X)	安全ではない
wgetch(3X)	安全ではない
wgetnstr(3X)	安全ではない
wgetnwstr(3X)	安全ではない
wgetstr(3X)	安全ではない
wgetwch(3X)	安全ではない
wgetwstr(3X)	安全ではない
whline(3X)	安全ではない
winch(3X)	安全ではない
winchnstr(3X)	安全ではない
winchstr(3X)	安全ではない
windex(3I)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
winnstr(3X)	安全ではない
winnwstr(3X)	安全ではない
winsch(3X)	安全ではない
winsdelln(3X)	安全ではない
winsertln(3X)	安全ではない
winsnstr(3X)	安全ではない
winsnwstr(3X)	安全ではない
winsstr(3X)	安全ではない
winstr(3X)	安全ではない
winswch(3X)	安全ではない
winswstr(3X)	安全ではない
winwch(3X)	安全ではない
winwchnstr(3X)	安全ではない
winwchstr(3X)	安全ではない
winwstr(3X)	安全ではない
wmove(3X)	安全ではない
wmovenextch(3X)	安全ではない
wmoveprevch(3X)	安全ではない
wprintw(3X)	安全ではない

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
wredrawln(3X)	安全ではない
wrefresh(3X)	安全ではない
wrindex(3I)	MT-安全
write_vtoc(3X)	安全ではない
wscanw(3X)	安全ではない
wscasecmp(3I)	MT-安全
wscat(3I)	MT-安全
wschr(3I)	MT-安全
wscmp(3I)	MT-安全
wscol(3I)	MT-安全
wscoll(3I)	MT-安全
wscpy(3I)	MT-安全
wscrl(3X)	安全ではない
wscspn(3I)	MT-安全
wsdup(3I)	MT-安全
wsetscrreg(3X)	安全ではない
wslen(3I)	MT-安全
wsncasecmp(3I)	MT-安全
wsncat(3I)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
wsncmp(3I)	MT-安全
wsncpy(3I)	MT-安全
wspbrk(3I)	MT-安全
wsprintf(3I)	MT-安全
wsrchr(3I)	MT-安全
wsscanf(3I)	MT-安全
wsspn(3I)	MT-安全
wstandend(3X)	安全ではない
wstandout(3X)	安全ではない
wstod(3I)	MT-安全
wstok(3I)	MT-安全
wstol(3I)	MT-安全
wstring(3I)	MT-安全
wsxfrm(3I)	MT-安全
wsyncdown(3X)	安全ではない
wsyncup(3X)	安全ではない
wtimeout(3X)	安全ではない
wtouchln(3X)	安全ではない
wvline(3X)	安全ではない

表 C–1 ライブラリルーチンの MT-安全レベル 続く

ライブラリルーチン	安全レベル
xdr(3N)	安全
xdrmem_create(3N)	MT-安全
xdrrec_create(3N)	MT-安全
xdrrec_endofrecord(3N)	安全
xdrrec_eof(3N)	安全
xdrrec_readbytes(3N)	安全
xdrrec_skiprecord(3N)	安全
xdrstdio_create(3N)	MT-安全
xdr_accepted_reply(3N)	安全
xdr_admin(3N)	安全
xdr_array(3N)	安全
xdr_authsys_parms(3N)	安全
xdr_authunix_parms(3N)	安全ではない
xdr_bool(3N)	安全
xdr_bytes(3N)	安全
xdr_callhdr(3N)	安全
xdr_callmsg(3N)	安全
xdr_char(3N)	安全
xdr_complex(3N)	安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
xdr_control(3N)	安全
xdr_create(3N)	MT-安全
xdr_destroy(3N)	MT-安全
xdr_double(3N)	安全
xdr_enum(3N)	安全
xdr_float(3N)	安全
xdr_free(3N)	安全
xdr_getpos(3N)	安全
xdr_hyper(3N)	安全
xdr_inline(3N)	安全
xdr_int(3N)	安全
xdr_long(3N)	安全
xdr_longlong_t(3N)	安全
xdr_opaque(3N)	安全
xdr_opaque_auth(3N)	安全
xdr_pointer(3N)	安全
xdr_quadruple(3N)	安全
xdr_reference(3N)	安全
xdr_rejected_reply(3N)	安全

表 C–1 ライブラリルーチンの MT-安全レベル 続く

ライブラリルーチン	安全レベル
xdr_replymsg(3N)	安全
xdr_setpos(3N)	安全
xdr_short(3N)	安全
xdr_simple(3N)	安全
xdr_sizeof(3N)	安全
xdr_string(3N)	安全
xdr_union(3N)	安全
xdr_u_char(3N)	安全
xdr_u_hyper(3N)	安全
xdr_u_int(3N)	安全
xdr_u_long(3N)	安全
xdr_u_longlong_t(3N)	安全
xdr_u_short(3N)	安全
xdr_vector(3N)	安全
xdr_void(3N)	安全
xdr_wrapstring(3N)	安全
xprt_register(3N)	MT-安全
xprt_unregister(3N)	MT-安全
y0(3M)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

ライブラリルーチン	安全レベル
y1(3M)	MT-安全
yn(3M)	MT-安全
ypclnt(3N)	安全ではない
yperr_string(3N)	安全ではない
ypprot_err(3N)	安全ではない
yp_all(3N)	安全ではない
yp_bind(3N)	安全ではない
yp_first(3N)	安全ではない
yp_get_default_domain(3N)	安全ではない
yp_master(3N)	安全ではない
yp_match(3N)	安全ではない
yp_next(3N)	安全ではない
yp_order(3N)	安全ではない
yp_unbind(3N)	安全ではない
yp_update(3N)	安全ではない
_NOTE(3X)	安全
_tolower(3C)	例外付きで MT-安全
_toupper(3C)	例外付きで MT-安全
__nis_map_group(3N)	MT-安全

表 C–1　ライブラリルーチンの MT-安全レベル　続く

索引

数字

A

C

D

E

F

G

K

L

M

N

P

R

S

T

U

V

W

X

あ

い

え