



マルチスレッドのプログラミング



Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 819-0390-12
2008 年 10 月

Sun Microsystems, Inc. (以下 米国 Sun Microsystems 社とします) は、本書に記述されている製品に含まれる技術に関連する知的財産権を所有します。特に、この知的財産権はひとつかそれ以上の米国における特許、あるいは米国およびその他の国において申請中の特許を含んでいることがあります。それらに限定されるものではありません。

本製品の一部は、カリフォルニア大学からライセンスされている Berkeley BSD システムに基づいていることがあります。UNIX は、X/Open Company, Ltd. が独占的にライセンスしている米国ならびに他の国における登録商標です。フォント技術を含む第三者のソフトウェアは、著作権により保護されており、提供者からライセンスを受けているものです。

U.S. Government Rights Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

この配布には、第三者によって開発された素材を含んでいることがあります。

本製品に含まれる HG-MinchoL、HG-MinchoL-Sun、HG-PMinchoL-Sun、HG-GothicB、HG-GothicB-Sun、および HG-PGothicB-Sun は、株式会社リコーがリョービマシクス株式会社からライセンス供与されたタイプフェースマスタをもとに作成されたものです。HeiseiMin-W3H は、株式会社リコーが財団法人日本規格協会からライセンス供与されたタイプフェースマスタをもとに作成されたものです。フォントとして無断複製することは禁止されています。

Sun、Sun Microsystems、Sun のロゴマーク、Solaris のロゴマーク、Java Coffee Cup のロゴマーク、docs.sun.com、ClusterTools、Java および Solaris は、米国およびその他の国における米国 Sun Microsystems 社またはその子会社の商標、登録商標もしくは、サービスマークです。

すべての SPARC 商標は、米国 SPARC International, Inc. のライセンスを受けて使用している同社の米国およびその他の国における商標または登録商標です。SPARC 商標が付いた製品は、米国 Sun Microsystems 社が開発したアーキテクチャに基づくものです。

OPENLOOK、OpenBoot、JLE は、サン・マイクロシステムズ株式会社の登録商標です。

Wnn は、京都大学、株式会社アステック、オムロン株式会社で共同開発されたソフトウェアです。

Wnn8 は、オムロン株式会社、オムロンソフトウェア株式会社で共同開発されたソフトウェアです。Copyright(C) OMRON Co., Ltd. 1995-2006. All Rights Reserved. Copyright(C) OMRON SOFTWARE Co., Ltd. 1995-2006 All Rights Reserved.

「ATOK for Solaris」は、株式会社ジャストシステムの著作物であり、「ATOK for Solaris」にかかる著作権、その他の権利は株式会社ジャストシステムおよび各権利者に帰属します。

「ATOK」および「推測変換」は、株式会社ジャストシステムの登録商標です。

「ATOK for Solaris」に添付するフェイスマーク辞書は、株式会社ビレッジセンターの許諾のもと、同社が発行する『インターネット・パソコン通信フェイスマークガイド』に添付のものを使用しています。

「ATOK for Solaris」に含まれる郵便番号辞書(7桁/5桁)は日本郵政公社が公開したデータを元に制作された物です(一部データの加工を行なっています)。

Unicode は、Unicode, Inc. の商標です。

本書で参照されている製品やサービスに関しては、該当する会社または組織に直接お問い合わせください。

OPEN LOOK および Sun Graphical User Interface は、米国 Sun Microsystems 社が自社のユーザおよびライセンス実施権者向けに開発しました。米国 Sun Microsystems 社は、コンピュータ産業用のビジュアルまたはグラフィカル・ユーザインタフェースの概念の研究開発における米国 Xerox 社の先駆者としての成果を認めるものです。米国 Sun Microsystems 社は米国 Xerox 社から Xerox Graphical User Interface の非独占的ライセンスを取得しており、このライセンスは、OPEN LOOK のグラフィカル・ユーザインタフェースを実装するか、またはその他の方法で米国 Sun Microsystems 社との書面によるライセンス契約を遵守する、米国 Sun Microsystems 社のライセンス実施権者にも適用されます。

本書で言及されている製品や含まれている情報は、米国輸出規制法で規制されるものであり、その他の国の輸出入に関する法律の対象となることがあります。核、ミサイル、化学あるいは生物兵器、原子力の海洋輸送手段への使用は、直接および間接を問わず厳しく禁止されています。米国の禁輸の対象としている国や、限定はされませんが、取引禁止顧客や特別指定国民のリストを含む米国輸出排除リストで指定されているものへの輸出および再輸出は厳しく禁止されています。

本書は、「現状のまま」をベースとして提供され、商品性、特定目的への適合性または第三者の権利の非侵害の黙示の保証を含みそれに限定されない、明示的であるか黙示的であるかを問わない、なんらの保証も行われぬものとします。

本製品が、外国為替および外国貿易管理法(外為法)に定められる戦略物資等(貨物または役務)に該当する場合、本製品を輸出または日本国外へ持ち出す際には、サン・マイクロシステムズ株式会社の事前の書面による承諾を得ることのほか、外為法および関連法規に基づく輸出手続き、また場合によっては、米国商務省または米国所轄官庁の許可を得ることが必要です。

原典: Multithreaded Programming Guide

Part No: 816-5137-12

Revision A

目次

はじめに	13
1 マルチスレッドの基礎	19
マルチスレッドに関する用語の定義	19
Solaris のマルチスレッドライブラリと標準	21
マルチスレッドの利点	22
アプリケーションの応答性の改善	22
マルチプロセッサの効率的な利用	22
プログラム構造の改善	23
システムリソースの節約	23
スレッドと RPC の併用	23
マルチスレッドの概念	24
並行性と並列性	24
マルチスレッドの構造	24
スレッドのスケジューリング	26
スレッド取り消し	26
スレッド同期	26
64 ビットアーキテクチャー	27
2 スレッドを使った基本プログラミング	29
スレッドのライフサイクル	29
Pthreads ライブラリ	30
デフォルトのスレッドの生成	30
スレッドの終了待ち	32
簡単なスレッドの例	33
スレッドの切り離し	34
スレッド固有データキーの作成	35

スレッド固有データキーの削除	37
スレッド固有データの設定	38
スレッド固有データの取得	38
スレッド固有データの大域性と局所性の例	39
スレッド識別子の取得	41
スレッド識別子の比較	42
スレッドの初期化ルーチンの呼び出し	42
スレッドの実行明け渡し	43
スレッドの方針およびスケジューリングパラメータの設定	44
スレッドの方針およびスケジューリングパラメータの取得	45
スレッド優先順位の設定	45
スレッドへのシグナルの送信	46
呼び出しスレッドのシグナルマスクのアクセス	47
安全な fork	48
スレッドの終了	48
スレッド終了処理の完了	49
スレッドの取り消し	49
スレッドの取り消し	51
取り消しを有効または無効にする	52
取り消しタイプの設定	52
取り消しポイントの設定	53
スタックへハンドラをプッシュする	54
スタックからハンドラを取り出す	54
3 スレッドの属性	57
属性オブジェクト	57
属性の初期化	58
属性の削除	60
切り離し状態の設定	61
切り離し状態の取得	62
スタックガードのサイズの設定	63
スタックガードのサイズの取得	64
スコープの設定	64
スコープの取得	65
スレッドの多重度の設定	66

スレッドの多重度の取得	66
スケジューリングポリシーの設定	67
スケジューリングポリシーの取得	68
継承スケジューリングポリシーの設定	69
継承スケジューリングポリシーの取得	70
スケジューリングパラメータの設定	70
スケジューリングパラメータの取得	71
スタックについて	72
スタックサイズの設定	74
スタックサイズの取得	75
スタックアドレスとサイズの設定	76
スタックアドレスとサイズの取得	77
4 同期オブジェクトを使ったプログラミング	79
相互排他ロック属性	80
mutex 属性オブジェクトの初期化	81
mutex 属性オブジェクトの削除	83
mutex のスコープ設定	83
mutex のスコープの値の取得	84
mutex の型属性の設定	84
mutex の型属性の取得	86
mutex 属性のプロトコルの設定	86
mutex 属性のプロトコルの取得	88
mutex 属性の優先順位上限の設定	89
mutex 属性の優先順位上限の取得	89
mutex の優先順位上限の設定	90
mutex の優先順位上限の取得	91
mutex の堅牢度属性の設定	92
mutex の堅牢度属性の取得	93
相互排他ロックの使用方法	94
mutex の初期化	94
mutex の整合性保持	96
mutex のロック	96
mutex のロック解除	98
ブロックしないで行う mutex のロック	99

指定した絶対時間までの mutex のロック	100
指定した時間間隔内の mutex のロック	101
mutex の削除	102
mutex ロックのコード例	103
スピンロックの使用	108
スピンロックの初期化	108
スピンロックの獲得	109
非ブロックスピンロックの獲得	110
スピンロックの解除	111
スピンロックの削除	111
条件変数の属性	112
条件変数の属性の初期化	113
条件変数の属性の削除	114
条件変数のスコープの設定	115
条件変数のスコープの取得	115
クロック選択条件変数の設定	116
クロック選択条件変数の取得	117
条件変数の使用方法	118
条件変数の初期化	118
条件変数によるブロック	119
1つのスレッドのブロック解除	121
時刻指定のブロック	122
間隔指定のブロック	124
全スレッドのブロック解除	125
条件変数の削除	126
「呼び起こし忘れ」問題	127
「生産者/消費者」問題	127
セマフォによる同期	131
名前付きセマフォと名前なしセマフォ	132
計数型セマフォの概要	132
セマフォの初期化	133
セマフォの加算	135
セマフォの値によるブロック	135
セマフォの減算	136
セマフォの削除	137
「生産者/消費者」問題—セマフォを使った例	138

読み取り/書き込みロック属性	139
読み取り/書き込みロック属性の初期化	140
読み取り/書き込みロック属性の削除	140
読み取り/書き込みロック属性の設定	141
読み取り/書き込みロック属性の取得	142
読み取り/書き込みロックの使用	142
読み取り/書き込みロックの初期化	143
読み取り/書き込みロックの読み取りロックの獲得	144
指定した絶対時間までの読み取り/書き込みロックの読み取りロックの獲得	145
読み取り/書き込みロックの非ブロック読み取りロックの獲得	146
読み取り/書き込みロックの書き込みロックの獲得	146
読み取り/書き込みロックの非ブロック書き込みロックの獲得	147
指定した絶対時間までの読み取り/書き込みロックの書き込みロックの獲得	148
読み取り/書き込みロックの解除	149
読み取り/書き込みロックの削除	150
バリアー同期の使用	151
同期バリアーの初期化	151
バリアーで同期をとるためのスレッドの待機	152
同期バリアーの削除	153
バリアー属性オブジェクトの初期化	153
バリアーのプロセス共有の属性の設定	154
バリアーのプロセス共有の属性の取得	155
バリアー属性オブジェクトの削除	155
プロセスの境界を越えた同期	156
「生産者/消費者」問題の例	156
プリミティブの比較	157
 5 Solaris ソフトウェアを使ったプログラミング	159
プロセスの生成時の fork の問題	159
fork1 モデル	160
汎用 fork モデル	163
正しい fork の選択	163
プロセスの作成: exec と exit の問題	164
タイマー、アラーム、およびプロファイル	164
タイマー	164

アラーム	164
マルチスレッドプログラムのプロファイリング	165
大域ジャンプ: <code>setjmp</code> と <code>longjmp</code>	165
リソースの制限	165
LWP とスケジューリングクラス	165
タイムシェアスケジューリング	166
リアルタイムスケジューリング	167
公平配分スケジューリング	168
固定優先順位スケジューリング	168
シグナルの拡張	168
同期シグナル	169
非同期シグナル	170
継続セマンティクス法	170
シグナルに関する操作	171
スレッド指定シグナル	173
完了セマンティクス法	174
シグナルハンドラと「非同期シグナル安全」	176
条件変数上で割り込まれた待機	177
入出力の問題	178
遠隔手続き呼び出しとしての入出力	178
非同期性の管理	179
非同期入出力	179
共有入出力と新しい入出力システムコール	180
<code>getc</code> と <code>putc</code> の代替	181
6 Solaris スレッドを使ったプログラミング	183
Solaris スレッドと POSIX スレッドの API の比較	183
API の主な相違点	184
関数比較表	184
Solaris スレッドに固有の関数	188
スレッド実行の停止	188
停止しているスレッドの再開	189
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	190
読み取り/書き込みロックの初期化	190
読み取りロックの獲得	192

読み取りロックの獲得の試行	192
書き込みロックの獲得	193
書き込みロックの獲得の試行	194
読み取り/書き込みロックの解除	194
読み取り/書き込みロックの削除	195
Solaris スレッドに類似した関数	196
スレッドの生成	197
最小のスタックサイズの取得	199
スレッド識別子の獲得	200
スレッドの実行明け渡し	201
スレッドへのシグナルの送信	201
呼び出しスレッドのシグナルマスクのアクセス	201
スレッドの終了	202
スレッドの終了待ち	203
スレッド固有データ用キーの作成	204
スレッド固有データの値の設定	205
スレッド固有データの値の取得	206
スレッド優先順位の設定	206
スレッド優先順位の取得	207
pthread に相当するものがある同期関数 — 相互排他ロック	208
mutex の初期化	208
mutex の削除	210
mutex の獲得	211
mutex の解除	211
mutex の獲得の試行	212
pthread に相当するものがある同期関数 — 条件変数	212
条件変数の初期化	212
条件変数の削除	214
条件変数によるブロック	214
条件変数による指定時刻付きブロック	215
時間間隔によるブロック	216
特定のスレッドのブロック解除	216
全スレッドのブロック解除	217
pthread に相当するものがある同期関数 — セマフォ	217
セマフォの初期化	217
セマフォの加算	219

セマフォの値によるブロック	219
セマフォの減算	220
セマフォの削除	220
プロセスの境界を越えた同期	221
「生産者/消費者」問題の例	221
fork() と Solaris スレッドに関する問題	223
7 安全なインタフェースと安全ではないインタフェース	225
スレッド安全	225
マルチスレッドインタフェースの安全レベル	226
安全でないインタフェースのための再入可能な関数	228
Solaris スレッドでの「非同期シグナル安全」関数	228
ライブラリの「MT-安全」レベル	229
「スレッド安全ではない」ライブラリ	230
8 コンパイルとデバッグ	231
マルチスレッドアプリケーションを開発するための Solaris 環境の設定	231
マルチスレッドアプリケーションのコンパイル	232
コンパイルの準備	232
スレッドの選択 — Solaris または POSIX	232
<thread.h> または <pthread.h>	233
マルチスレッドプログラムのコンパイルとリンク	233
リンク時の POSIX セマフォ用 -lrt の指定	236
代替スレッドライブラリ	236
マルチスレッドプログラムのデバッグ	236
マルチスレッドプログラムでよく起こるミス	236
DTrace による追跡とデバッグ	237
Performance Analyzer によるプロファイリング	238
Thread Analyzer を使用したデータの競合およびデッドロックの検出	238
dbx の使用	238
TNF ユーティリティーによる追跡とデバッグ	240
truss の使用	241
mdb の使用	241

9	プログラミング上の指針	243
	大域変数の考慮	243
	静的局所変数の利用	245
	スレッドの同期	245
	シングルスレッド化	246
	再入可能な関数	246
	デッドロックの回避	249
	スケジューリングに関するデッドロック	250
	ロックに関する指針	250
	デッドロックの検出	251
	マルチスレッドコードの基本的な指針	251
	スレッドの生成と使用	252
	マルチプロセッサへの対応	252
	アーキテクチャー	253
	スレッドプログラムの例	257
	参考資料	257
A	拡張の例:スレッドプールの実装	259
	スレッドプールについて	259
	スレッドプールの例について	260
	スレッドプールの関数	260
	スレッドプールのコード例	261
	スレッドプールの例が示している内容	272
	索引	273

はじめに

マルチスレッドのプログラミングは、Solaris® オペレーティングシステム (Solaris OS) 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。

このマニュアルでは、POSIX スレッドと Solaris スレッドの両方のインタフェースを扱っていますが、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、第6章「[Solaris スレッドを使ったプログラミング](#)」で説明しています。この2組のインタフェースは共通の実装を共有しており、互いに完全に互換性があります。POSIX スレッドインタフェースの呼び出しは、Solaris スレッドインタフェースの呼び出しと自由に混在させることができます。

POSIX スレッドについては、Single UNIX Specification Version 3 (<http://www.opengroup.org/>) を参照してください。

注- このリリースでは、SPARC® および x86 系列のプロセッサアーキテクチャー (UltraSPARC®, SPARC64, AMD64, Pentium, Xeon EM64T) を使用するシステムをサポートします。サポートされるシステムについては、Solaris 10 Hardware Compatibility List (<http://www.sun.com/bigadmin/hcl>) を参照してください。本書では、プラットフォームにより実装が異なる場合は、それを特記します。

本書では、「x86」という用語は AMD64 あるいは Intel Xeon/Pentium 製品系列と互換性のあるプロセッサを使用して製造された 32 ビットおよび 64 ビットシステムを意味します。サポートされるシステムについては、Solaris 10 Hardware Compatibility List を参照してください。

対象読者

このマニュアルは、マルチスレッドを使った新しいプログラムを作成するか、または既存のプログラムをマルチスレッド化しようとしているアプリケーション開発者を対象にしています。

このマニュアルを使用する開発者は、次のテクノロジーに精通し、使いこなせる必要があります。

- UNIX® SVR4 システム - 現在の Solaris リリースであることが望ましい
- C プログラミング言語 - マルチスレッドインタフェースは標準 C ライブラリによって提供される
- 並行または並列プログラミングの原理 (逐次プログラミングと対照する原理として)

このマニュアルの構成

第1章「[マルチスレッドの基礎](#)」では、このリリースにおけるスレッドの実装の構造的な概要について説明します。

第2章「[スレッドを使った基本プログラミング](#)」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 スレッドルーチンについて説明します。

第3章「[スレッドの属性](#)」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「[同期オブジェクトを使ったプログラミング](#)」では、スレッドの同期ルーチンについて説明します。

第5章「[Solaris ソフトウェアを使ったプログラミング](#)」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「[Solaris スレッドを使ったプログラミング](#)」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7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](#)」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8章「[コンパイルとデバッグ](#)」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9章「[プログラミング上の指針](#)」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「[拡張の例: スレッドプールの実装](#)」では、ワークスレッドのプールの実装方法について説明します。

関連する Sun 以外の Web サイト情報

このマニュアルでは、Sun 以外の URL を挙げ、関連する補足情報を示す場合があります。

注- このマニュアルで説明する Sun 以外の Web サイトの利用については、Sun は責任を負いません。こうしたサイトやリソース上の、またはこれらを通じて利用可能な、コンテンツ、広告、製品、その他の素材について、Sun は推奨しているわけではなく、Sun はいかなる責任も負いません。こうしたサイトやリソース上の、またはこれらを経由して利用可能な、コンテンツ、製品、サービスを利用または信頼したことによって発生した(あるいは発生したと主張される)いかなる損害や損失についても、Sun は一切の責任を負いません。

マニュアル、サポート、およびトレーニング

Sun の Web サイトでは、次の追加のリソースに関する情報を提供しています。

- マニュアル (<http://jp.sun.com/documentation/>)
- サポート (<http://jp.sun.com/support/>)
- トレーニング (<http://jp.sun.com/training/>)

関連情報

マルチスレッド化では、さまざまな点から、関数の対話について考慮する必要があります。以下に、推奨する参考文献を紹介します。

- 『Concurrent Programming』、Alan Burns、Geoff Davies 共著 (Addison-Wesley, 1993)
- 『Distributed Algorithms and Protocols』、Michel Raynal 著 (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)
- 『Principles of Concurrent Programming』、M. Ben-Ari 著 (Prentice-Hall, 1982)
- 『Programming with Threads』 (Steve Kleiman、Devang Shah、Bart Smaalders 共著 (Prentice-Hall, 1996)
 - 『Programming with POSIX Threads』、David R. Butenhof 著 (Addison-Wesley Professional, 1997)

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。	<code>.login</code> ファイルを編集します。 <code>ls -a</code> を使用してすべてのファイルを表示します。 <code>system%</code>
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。	<code>system% su</code> <code>password:</code>
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、 <code>rm filename</code> 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。
「 」	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。	<code>sun% grep '^#define \</code> <code>XV_VERSION_STRING'</code>

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。

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 プロセスは単一の制御スレッドを含んでいたのに対して、マルチスレッド (MT) は1つのプロセスを複数の実行スレッドに分割します。これらのスレッドはそれぞれ独立して実行されます。

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。すぐにマルチスレッドを使用する場合は、[第2章「スレッドを使った基本プログラミング」](#)へ進んでください。

マルチスレッドプログラミングに関する詳細な情報が必要な場合は、「はじめに」の [15 ページの「関連情報」](#) の節を参照してください。

- [19 ページの「マルチスレッドに関する用語の定義」](#)
- [21 ページの「Solaris のマルチスレッドライブラリと標準」](#)
- [22 ページの「マルチスレッドの利点」](#)
- [24 ページの「マルチスレッドの概念」](#)

マルチスレッドに関する用語の定義

[表 1-1](#) では、このマニュアルで使用する用語の一部について解説します。

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	<code>fork(2)</code> 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ー 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ます。
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。

表 1-1 マルチスレッドに関する用語の定義 (続き)

用語	定義
POSIX pthreads	POSIX スレッドに準拠したスレッドインタフェース。詳細については、 21 ページの「Solaris のマルチスレッドライブラリと標準」 を参照してください。
Solaris スレッド	POSIX スレッドに準拠しない、Sun Microsystems のスレッドインタフェース。pthread の祖先。
シングルスレッド	1 プロセス 1 スレッドで動作させること。実行は逐次処理で行われ、制御スレッドが 1 つに制限されます。
マルチスレッド	1 プロセス複数スレッドで動作させること。実行は複数の制御スレッドで行われ、並列または並行処理が使用されます。
ユーザーレベル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) ユーザー空間のスレッドルーチンによって管理されるスレッド。ユーザースレッドの作成および処理には、POSIX pthread や Solaris スレッドの API が使用されます。このマニュアルでは(また一般にも)、スレッドとはユーザーレベルのスレッドを指します。 注- 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、カーネルスレッドのプログラミングについては触れません。
軽量プロセス	カーネルコードとシステムコールを実行するカーネルスレッド (LWP)。LWP はシステムスレッドのスケジューラによって管理され、アプリケーションプログラマが直接制御することはできません。Solaris 9 以降、ユーザーレベルのすべてのスレッドに専用の LWP が用意されるようになりました。これは 1:1 スレッドモデルとも呼ばれます。
結合スレッド (廃止された用語)	Solaris 9 より前の、単一の LWP に永続的に結合されたユーザーレベルのスレッド。Solaris 9 以降は、すべてのスレッドに専用の LWP が用意されるようになったため、すべてのスレッドが結合スレッドです。非結合スレッドの概念は存在しなくなりました。
非結合スレッド (廃止された用語)	Solaris 9 より前の、単一の LWP に結合されているとは限らないユーザーレベルのスレッド。Solaris 9 以降は、すべてのスレッドに専用の LWP が用意されるようになったため、非結合スレッドの概念は存在しなくなりました。
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ます。これらのデータ型および関数は、POSIX スレッド、相互排他ロック (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ます。
相互排他ロック	共有データへのアクセスをロック / ロック解除するために使用されるオブジェクト。このようなオブジェクトは mutex と呼ばれます。

表 1-1 マルチスレッドに関する用語の定義 (続き)

用語	定義
条件変数	状態が変化するまでスレッドをブロックするために使用されるオブジェクト。
読み取り/書き込みロック	共有データに対して複数の読み取り専用アクセスを許可するが、共有データを変更する場合は排他的アクセスを許可するために使用されるオブジェクト。
計数型セマフォ	複数のスレッドからの共有リソースへのアクセスを調整するために 0 以上の整数カウントが使用される、メモリーベースの同期機構。
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含む、一般化された形の並列性。

Solaris のマルチスレッドライブラリと標準

マルチスレッドプログラミングの概念の起源は、少なくとも 1960 年代にさかのぼります。その後、UNIX システム上でマルチスレッドプログラミングが行われるようになったのは、1980 年代中盤のことです。マルチスレッドの意味とそのサポートに必要な機能に関する合意がありますが、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 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。この標準は最終的に承認され、現在は Single UNIX Specification (SUS) の一部になっています。最新の仕様は、[Open Group の Web サイト](#)で入手できます。Solaris 10 リリースから、Solaris OS は Open Group の UNIX 03 Product Standard (SUSv3) に準拠しています。

POSIX 標準が承認される前、Solaris のマルチスレッド API は、Solaris `libthread` ライブラリで実装されていました。Sun が開発したこのライブラリは、そのあとで UNIX International (UI) スレッド標準の基礎になっています。この `libthread` ライブラリは、1993 年に Solaris 2.2 リリースで導入されました。1995 年の Solaris 2.5 リリースでは、POSIX 標準のサポートが `libpthread` API によって追加され、それ以降は両方の API が使用可能になりました。Solaris 10 リリースからは、`libthread` ライブラリと `libpthread` ライブラリが標準の `libc` ライブラリに統合されました。

`libthread` ライブラリと `libpthread` ライブラリは、実行環境とコンパイル環境の両方の下位互換を提供するために維持されています。`libthread.so.1` および `libpthread.so.1` 共有オブジェクトは、`libc.so.1` に対するフィルタとして実装されています。詳細は、[libthread\(3LIB\)](#) と [libpthread\(3LIB\)](#) のマニュアルページを参照してください。

両方のスレッドライブラリがサポートされていますが、通常は POSIX ライブラリを使用するようにしてください。 [threads\(5\)](#) のマニュアルページには、POSIX スレッドと Solaris スレッドの相違点と類似点が記載されています。

この『マルチスレッドのプログラミング』は、POSIX 標準 IEEE Std 1003.1: 2001 の最新のリビジョン (ISO/IEC 9945:2003 と同 Single UNIX Specification Version 3 と呼ばれる) に基づいています。

Solaris スレッド固有の機能については、[第6章「Solaris スレッドを使ったプログラミング」](#)を参照してください。

マルチスレッドの利点

この節では、マルチスレッドの利点について概説します。

プログラムのマルチスレッド化には、次のような利点があります。

- [22 ページの「アプリケーションの応答性の改善」](#)
- [22 ページの「マルチプロセッサの効率的な利用」](#)
- [23 ページの「プログラム構造の改善」](#)
- [23 ページの「システムリソースの節約」](#)

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、個々の独立した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 のユーザーは、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。アプリケーションのパフォーマンスはプロセッサが増えると透過的に向上します。これは、利用可能なプロセッサ数に応じたスレッドのスケジューリングがオペレーティングシステムによって処理されるためです。マルチコアプロセッサやマルチスレッドプロセッサが使用可能な場合は、それらのコアやスレッドが OS によってプロセッサとして見なされるため、マルチスレッドアプリケーションの性能はそれに応じて向上します。

行列の乗算のような並列性の度合いが高い数値計算アルゴリズムや数値計算アプリケーションでは、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。

注- このマニュアルでは、特に断りのない限り、マルチプロセッサの説明はマルチコアプロセッサとマルチスレッドプロセッサにも該当します。

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。たとえば、多様なタスクを実行するスレッド化されていないプログラムでは、そのコードの大部分がそれらのタスクの調整のためだけに費やされる可能性があります。それらのタスクをスレッドとしてプログラミングすれば、コードを簡略化できます。マルチスレッド化されたプログラム(特に、複数の並行ユーザーにサービスを提供するプログラム)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ー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、ほかのプロセスに属するスレッドとの通信や同期など、プログラマに課される処理が多くなります。この処理には、異なるプロセスに含まれるスレッド間の通信の処理や、それらのスレッドの操作の同期などがあります。これらのスレッドが同じプロセスに含まれていると、通信や同期がはるかに簡単になります。

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、ほかのコンピュータとともに並列的に実行されます。

注-分散システム全体にわたって動作するアプリケーションのマルチスレッド化を実現するには、MPI (Message Processing Interface) の方が効果的なアプローチになることがあります。MPI の詳細は、<http://www-unix.mcs.anl.gov/mpi/> を参照してください。

Sun HPC ClusterTools™ ソフトウェアには、MPI のオープンソース実装である OMPI (Open MPI Message Passing Interface) が含まれています。ClusterTools の詳細は、[Sun HPC ClusterTools の製品ページ](#) を参照してください。

マルチスレッドの概念

この節では、マルチスレッドの基本概念について説明します。

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。この並行性は、複数のスレッドが進行しているが、実際にはこれらのスレッドが同時には実行されていないことを示しています。スレッド間の切り替えが高速なため、スレッドが同時に実行されているように見える可能性があります。

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実行されるため、プロセスの実行状態は並列的になります。これが本来の同時実行です。プロセス内のスレッド数が利用可能なプロセッサ数に等しいか、それより少ない場合は、オペレーティングシステムのスレッドサポートシステムによって、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が保証されます。たとえば、4つのスレッドを使用してプログラミングされ、2つのデュアルコアプロセッサを備えたシステム上で実行される行列の乗算では、結果の行を同時に計算するために4つのプロセッサコア上で各ソフトウェアスレッドを同時に実行できます。

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。各プロセスに単一のスレッドが含まれるので、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かわりに、スレッドを生成するとシステムへの負荷は小さくなります。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短くて済みます。これは、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からです。

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、もっとも重要な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、プロセス中のほかのすべてのスレッドがただちに利用できます。

ただしこのようにデータを共有すると、プログラマにとって別の一連の課題が発生します。データが複数のスレッドから一度に変更されたり、あるスレッドによって変更されている間に別のスレッドから同時に読み取られたりすることのないようにスレッドを同期化する必要があります。詳細については、[26 ページの「スレッド同期」](#)を参照してください。

ユーザーレベルのスレッド

スレッドは、マルチスレッドプログラミングにおける基本プログラミングインタフェースです。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。

ユーザーレベルのスレッドの状態

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
- プログラムカウンタ (PC) およびスタックポインタを含む登録状態
- 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プロセスデータの大部分を共有します。このため、あるスレッドによって共有データに加えられた変更は、プロセス内のほかのスレッドから認識できます。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。

注-ユーザー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。

スレッドのスケジューリング

POSIX 標準は、次の3つのスケジューリングポリシーを規定しています。先入れ先出し (SCHED_FIFO)、ラウンドロビン (SCHED_RR)、そしてカスタム (SCHED_OTHER) です。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。これらの方針で実行されているスレッドは、通常は特殊な特権が必要な Solaris のリアルタイム (RT) スケジューリングクラスに含まれます。SCHED_OTHER がデフォルトのスケジューリングポリシーです。SCHED_OTHER 方針で実行されているスレッドは、従来の Solaris タイムシェアリング (TS) スケジューリングクラスに含まれます。

Solaris は、ほかのスケジューリングクラス、つまり対話型タイムシェアリング (IA) クラス、公平分配 (FSS) クラス、および固定優先順位 (FX) クラスを提供しています。ここでは、これらの特殊なクラスについては説明していません。詳細は、Solaris [prioctl\(2\)](#) のマニュアルページを参照してください。

SCHED_OTHER 方針については、[165 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。

次の2つのスケジューリングスコープが使用できます。プロセススコープ (PTHREAD_SCOPE_PROCESS) とシステムスコープ (PTHREAD_SCOPE_SYSTEM) です。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。プロセススコープは、同一プロセス内の別スレッドとだけリソースの奪い合いを行うスレッドに影響を及ぼします。システムスコープは、システム内のその他のすべてのスレッドとリソースの奪い合いを行うスレッドに影響を及ぼします。ただし、Solaris 9 リリースから、これらの2つのスコープの差異は事実上なくなっています。

スレッド取り消し

スレッドは、そのプロセス中のほかのスレッドの実行終了を要求できます。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、取り消し要求に応じるとき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えます。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。

スレッド同期

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、同時に特定のコードセクションを実行するスレッド、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/書き込みロックは、保護された共有リソースに対して、並行読み取りおよび排他的書き込みを許可します。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。
- 通常、計数型セマフォは、リソースへのアクセスを調整します。計数の数は、セマフォによって保護されているデータに並行的にアクセスできるスレッド数の上限です。設定された値に達すると、この数が増えるまで、セマフォによって呼び出しスレッドがブロックされます。2進型セマフォ (数は1に設定) は、相互排他ロックと同様の動作を行います。

64ビットアーキテクチャー

アプリケーションの開発者にとって、Solaris 64ビットオペレーティング環境と32ビットオペレーティング環境の主な違いは、使用されるC言語データ型モデルです。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、ILP32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、32ビット環境における現時点の最大4Gバイトの40億倍にあたります。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。
巨大なアドレス空間では、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も多くなります。デフォルトのスタックサイズは、32ビット版では1Mバイト、64ビット版では2Mバイトです。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。
- カーネルメモリの読み取り
64ビットカーネルは、内部的に64ビットデータ構造を使用するLP64オブジェクトです。このため、libkvm、/dev/mem、/dev/kmemを使用する既存の32ビットアプリケーションは正しく動作しません。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 を使用する 32 ビットプログラムでは、32 ビットプロセスは認識されますが、64 ビットプロセスは認識されません。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 ビットライブラリ

32 ビットアプリケーションは 32 ビットのライブラリと、64 ビット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。

- 64 ビット演算

64 ビット 演算は以前の 32 ビット版 Solaris リリースで長い間使用されてきました。64 ビット実装では、整数の演算やパラメータの引き渡しに、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。

- 大規模ファイル

アプリケーションが大容量ファイルのサポートしか要求していない場合、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ができます。ただし、64 ビットの機能を最大限に活かすためには、64 ビットアプリケーションに変換することをお勧めします。

スレッドを使った基本プログラミング

この章では、POSIX スレッドの基本的な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もっともよく使われるのがこの種のスレッドです。デフォルト以外の属性を使用したスレッドの生成方法と使用方法についても、この章で説明します。

- 29 ページの「スレッドのライフサイクル」
- 30 ページの「Pthreads ライブラリ」

スレッドのライフサイクル

スレッドが作成されると、現在のプロセスに新しい制御スレッドが追加されます。どのプロセスでも、プログラムの `main()` ルーチンに、少なくとも1つの制御スレッドが存在します。プロセス内の各スレッドは同時に実行され、呼び出しプロセスの大域データにアクセスできます。さらに、各スレッドには、独自のプライベート属性と呼び出しスタックも存在します。

新しいスレッドを作成するために、実行中のスレッドが `pthread_create()` 関数を呼び出し、新しいスレッドが実行する関数へのポインタを渡します。新しいスレッドの関数には、スレッド属性とともに1つの引数を渡すこともできます。スレッドの実行は、`pthread_create()` 関数からの正常な復帰で始まります。スレッドは、そのスレッドとともに呼び出された関数が正常に完了したときに終了します。

また、スレッドが `pthread_exit()` ルーチンを呼び出した場合や、ほかの任意のスレッドがそのスレッドを明示的に終了するための `pthread_cancel()` を呼び出した場合にもスレッドを終了できます。スレッドはまた、そのスレッドを呼び出したプロセスの終了によって終了する場合もあります。

Pthreads ライブラリ

Pthreads API ライブラリは、100 を超える関数で構成されています。使用法のカテゴリでグループ化された関数の完全なリストについては、[pthreads\(5\)](#) のマニュアルページを参照してください。

この節では、スレッドの基本的なプログラミングに使用される関数について簡単に説明しています。これらの説明は、各関数が実行するタスクに従って構成され、関連する API 関数のマニュアルページへのリンクが含まれています。次のリストは、特定のタスクが説明されている場所を示しています。

- 30 ページの「デフォルトのスレッドの生成」
- 32 ページの「スレッドの終了待ち」
- 33 ページの「簡単なスレッドの例」
- 34 ページの「スレッドの切り離し」
- 35 ページの「スレッド固有データキーの作成」
- 37 ページの「スレッド固有データキーの削除」
- 38 ページの「スレッド固有データの設定」
- 38 ページの「スレッド固有データの取得」
- 39 ページの「スレッド固有データの大域性と局所性の例」
- 41 ページの「スレッド識別子の取得」
- 42 ページの「スレッド識別子の比較」
- 42 ページの「スレッドの初期化ルーチンの呼び出し」
- 43 ページの「スレッドの実行明け渡し」
- 44 ページの「スレッドの方針およびスケジューリングパラメータの設定」
- 45 ページの「スレッドの方針およびスケジューリングパラメータの取得」
- 45 ページの「スレッド優先順位の設定」
- 46 ページの「スレッドへのシグナルの送信」
- 47 ページの「呼び出しスレッドのシグナルマスクのアクセス」
- 48 ページの「安全な fork」
- 48 ページの「スレッドの終了」
- 49 ページの「スレッド終了処理の完了」
- 49 ページの「スレッドの取り消し」
- 51 ページの「スレッドの取り消し」
- 52 ページの「取り消しを有効または無効にする」
- 52 ページの「取り消しタイプの設定」
- 53 ページの「取り消しポイントの設定」
- 54 ページの「スタックへハンドラをプッシュする」
- 54 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。

- プロセススコープ
- 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
- 優先順位が0

`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。詳細については、[58 ページの「属性の初期化」](#)を参照してください。

pthread_create の構文

現在のプロセスへ新たに制御スレッドを追加するときは、`pthread_create(3C)` を使用します。

```
int pthread_create(pthread_t *restrict tid, const pthread_attr_t
    *restrict tattr, void*(*start_routine)(void *), void *restrict arg);
```

```
#include <pthread.h>

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

/* default behavior*/
ret = pthread_create(&tid, NULL, start_routine, arg);

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);
/* default behavior specified*/
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

`pthread_create()` 関数は、必要な状態動作を持つ `attr` で呼び出されます。`start_routine` は新しいスレッドで実行する関数です。`start_routine` 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。[31 ページの「pthread_create の構文」](#)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。

`pthread_create()` の属性引数(デフォルト属性)として `NULL` を指定すると、デフォルト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。

pthread_create の戻り値

pthread_create() は、正常終了時に 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

説明: システム制限を超えました。たとえば、生成するスレッド数が多すぎます。

EINVAL

説明: *tattr* の値が無効です。

EPERM

説明: 呼び出し元は、必要なスケジューリングパラメータまたはスケジューリングポリシーを設定するための適切なアクセス権を持っていません。

スレッドの終了待ち

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。

pthread_join の構文

スレッドの終了待ちを行うには、`pthread_join(3C)` を使用します。

```
int pthread_join(pthread_t tid, void **status);

#include <pthread.h>

pthread_t tid;
int ret;
void *status;

/* waiting to join thread "tid" with status */
ret = pthread_join(tid, &status);

/* waiting to join thread "tid" without status */
ret = pthread_join(tid, NULL);
```

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、[61 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場合、すべてのスレッドはそのスレッドが終了するまで待機します。そして、待機しているスレッドの1つが成功を返します。待機しているほかのスレッドは ESRCH エラーを返して失敗します。

`pthread_join()` の復帰後は、終了したスレッドに関連付けられていたデータ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。

pthread_join の戻り値

`pthread_join()` は、正常終了時に 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

説明: 指定のスレッド ID に対応するスレッドが見つかりません。

EDEADLK

説明: デッドロックが発生しています。たとえば、スレッドが自身を待機していたり、スレッド A とスレッド B が互いに待機し合っていたりします。

EINVAL

説明: 指定のスレッド ID に対応するスレッドは、切り離されています。

`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す。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。

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。`fetch()` 手続きは複雑なデータベース検索を行うため、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、こうした必要な処理を行ってから、`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* がスタックパラメー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。しかし、通常は、`malloc` でヒープから領域を割り当てる方法をお勧めします。スレッドのスタック領域にアドレスを渡す方法だと、スレッドが終了したときアドレスが無効になったり、割り当て直されたりする可能性があります。

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;  
    void *status;  
  
    pthread_create(&helper, NULL, fetch, &pbe);  
  
    /* do something else for a while */  
  
    pthread_join(helper, &status);  
    /* it's now safe to use result */  
}  
  
void *fetch(struct phonebookentry *arg)  
{  
    struct phonebookentry *npbe;  
    /* fetch value from a database */  
  
    npbe = search (prog_name)  
        if (npbe != NULL)  
            *arg = *npbe;  
    pthread_exit(0);  
}  
  
struct phonebookentry {  
    char name[64];  
    char phonenum[32];  
    char flags[16];  
}
```

スレッドの切り離し

[pthread_detach\(3C\)](#) 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、[pthread_join\(3C\)](#) の代わりに利用できます。

pthread_detach の構文

```
int pthread_detach(pthread_t tid);
```

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

```
pthread_t tid;
```

```
int ret;

/* detach thread tid */
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、アプリケーションに対して示すために使われます。スレッドは、必要なくなった時点で切り離すべきです。*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。

pthread_detach の戻り値

`pthread_detach()` は、正常終了時に 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

説明: *tid* は切り離されたスレッドです。

ESRCH

説明: *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 プログラムは、2つの基本クラスデータを持っています。局所データと大域データです。マルチスレッドの C プログラムには、さらに第3のクラスであるスレッド固有データ (TSD) が追加されます。スレッド固有データは大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。

注 - Solaris OS では、スレッドが大域変数のプライベートコピーを持つことができる代替機能がサポートされています。この機構は、スレッドローカルな記憶領域 (TLS) と呼ばれます。スレッドローカルにする変数を宣言するためにキーワード `__thread` が使用され、コンパイラは、これらの変数がスレッド単位に割り当てられるように自動的に配置します。詳細については、『[リンカーとライブラリ](#)』の第8章「[スレッド固有領域 \(TLS\)](#)」を参照してください。

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 (*key*) 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 の構文

```
int pthread_key_create(pthread_key_t *key,  
    void (*destructor) (void *));
```

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

```
pthread_key_t key;  
int ret;
```

```
/* key create without destructor */  
ret = pthread_key_create(&key, NULL);
```

```
/* key create with destructor */  
ret = pthread_key_create(&key, destructor);
```

[pthread_key_create\(3C\)](#) 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 (*key*) 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に大域的なキーです。スレッド固有データが生成された時点では、すべてのスレッドで、キーの初期値として `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。プロセス内のすべてのスレッドが共有するキーについては、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。

`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。

pthread_key_create の戻り値

`pthread_key_create()` は、正常終了時に 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、次の値を返します。

EAGAIN

説明: キーの名前空間が使い果たされました。

ENOMEM

説明: 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

既存のスレッド固有データキーを削除するには、`pthread_key_delete(3C)` を使用します。このキーは無効になっているので、キーに関連付けられているあらゆるメモリを解放できます。無効なキーが参照されると、エラーが返されます。

`pthread_key_delete` の構文

```
int pthread_key_delete(pthread_key_t key);
```

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

```
pthread_key_t key;
int ret;
```

```
/* key previously created */
ret = pthread_key_delete(key);
```

`key` を削除したあとで `pthread_setspecific()` または `pthread_getspecific()` 呼び出しを使用してそのキーを参照しようとすると、予期しない結果が発生します。

`pthread_key_delete()` 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。`pthread_key_create()` と `pthread_key_delete()` を繰り返し呼び出すと、問題が発生する可能性があります。

この問題は、Solaris 実装で、`key` の値が `pthread_key_delete()` によって無効にされたあとで再利用されないために発生します。`pthread_key_create()` を呼び出すたびに新しいキー値が割り当てられるため、キー情報を保持するために割り当てられる内部メモリが増えます。`pthread_key_create() ... pthread_key_delete()` の無限ループによって、最終的にはすべてのメモリが使い果たされます。可能であれば、目的の各キーに対して `pthread_key_create()` を 1 回だけ呼び出し、`pthread_key_delete()` は呼び出さないようにしてください。

`pthread_key_delete` の戻り値

`pthread_key_delete()` は、正常終了時に 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ます。以下の条件が検出されると `pthread_key_delete()` は失敗し、対応する値を返します。

EINVAL

説明: `key` の値が有効ではありません。

スレッド固有データの設定

指定したスレッド固有データキーに対して、スレッド固有の割り当てを設定するには、`pthread_setspecific(3C)` を使用します。

`pthread_setspecific` の構文

```
int pthread_setspecific(pthread_key_t key, const void *value);

#include <pthread.h>

pthread_key_t key;
void *value;
int ret;

/* key previously created */
ret = pthread_setspecific(key, value);
```

`pthread_setspecific` の戻り値

`pthread_setspecific()` は、正常終了時に 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

説明: 仮想メモリーが不足しています。

EINVAL

説明: キーが無効です。

注 - `pthread_setspecific()` は、新しい割り当てが設定されても、記憶領域を解放しません。既存の割り当ては、解放する必要があります。解放しない場合は、メモリーリークが発生することがあります。

スレッド固有データの取得

`key` の呼び出しスレッドの割り当てを取得し、この割り当て情報を `value` がポイントする記憶場所に格納するには、`pthread_getspecific(3C)` を使用します。

`pthread_getspecific` の構文

```
void *pthread_getspecific(pthread_key_t key);
```

```
#include <pthread.h>

pthread_key_t key;
void *value;

/* key previously created */
value = pthread_getspecific(key);
```

pthread_getspecific の戻り値

pthread_getspecific はエラーを返しません。

スレッド固有データの大域性と局所性の例

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、コードでは 2 つの大域変数 *errno* と *mywindow* を参照しています。これらの大域値は、実際には各スレッド専用の項目への参照になります。

例 2-2 スレッド固有データの大域性と局所性

```
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。ほか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。ヘッダーファイル *errno.h* をインクルードすると、*errno* の参照が *errno* のスレッド専用インスタンスの参照になります。これにより、あるスレッドによる *errno* の参照が、ほかのスレッドによる *errno* の参照とは別の記憶領域の場所を参照するようになります。

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。したがって、*errno* と同様に、あるスレッドによる *mywindow* の参照は、ほかのスレッドによる *mywindow* の参照とは別の記憶領

域を参照します。最終的には、この参照は異なるウィンドウへの参照になります。ここでの唯一の違いは、`errno` はシステムによって処理されるのに対し、`mywindow` はプログラマが処理しなければならない点です。

次の例は、`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、`pthread_getspecific()` に、大域変数 `mywindow_key` と、このスレッドのウィンドウの識別情報が入る出力パラメータ `win` を渡します。

例2-3 大域参照から局所参照への変換

```
thread_key_t mywin_key;

FILE *_mywindow(void) {
    FILE *win;
    win = pthread_getspecific(mywin_key);
    return(win);
}

#define mywindow _mywindow()

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

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

変数 `mywin_key` は、スレッドごと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()` を呼び出して自分専用のウィンドウを初期化し、スレッド固有データの参照用に自分専用の `mywindow` 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、参照の設定方法を示しています。

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_key_create(&mywindow_key, free_key);
}

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。その後、`create_window()` を呼び出して、スレッド用にウィンドウを設定します。*win* は、ウィンドウに割り当てられた記憶領域をポイントします。最後に `pthread_setspecific()` が呼び出され、*win* とキーとが結び付けられます。

続いて、スレッドは `pthread_getspecific()` を呼び出して、上記の大域キー (*key*) を渡します。その結果、スレッドは、以前の `pthread_setspecific()` 呼び出しで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

呼び出しスレッドのスレッド識別子 (thread identifier) を取得するには、`pthread_self(3C)` を使用します。

pthread_self の構文

```
pthread_t pthread_self(void);
```

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

```
pthread_t tid;
```

```
tid = pthread_self();
```

pthread_self の戻り値

pthread_self() は、呼び出しスレッドのスレッド識別子 (thread identifier) を返します。

スレッド識別子の比較

2つのスレッドのスレッド識別番号を比較するには、[pthread_equal\(3C\)](#) を使用します。

pthread_equal の構文

```
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

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

```
pthread_t tid1, tid2;  
int ret;
```

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

pthread_equal の戻り値

pthread_equal() は、*tid1* と *tid2* が等しいときは0以外の値、それ以外の場合は0を返します。*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。

スレッドの初期化ルーチンの呼び出し

スレッド化されたプロセス内の[pthread_once\(3C\)](#) は、最初に呼び出されたとき、初期化ルーチンを呼び出します。プロセス内の任意のスレッドからの2回目以降のpthread_once() の呼び出しは何の効果もありません。

pthread_once の構文

```
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` パラメータは、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。

pthread_once の戻り値

`pthread_once()` は、正常終了時に 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

説明: `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)` 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。すぐに実行可能なこのようなスレッドが存在しない場合、呼び出しスレッドは実行を継続します。`sched_yield()` 関数は Pthread API の一部ではなく、リアルタイムライブラリ関数に含まれている関数です。`sched_yield()` を使用するには、`<sched.h>` をインクルードする必要があります。

sched_yield の構文

```
int sched_yield(void);
```

```
#include <sched.h>
int ret;
ret = sched_yield();
```

sched_yield の戻り値

`sched_yield()` は、正常終了時に 0 を返します。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。

スレッドの方針およびスケジューリングパラメータの設定

個々のスレッドのスケジューリングポリシーおよびスケジューリングパラメータを変更するには、`pthread_setschedparam(3C)` を使用します。

pthread_setschedparam の構文

```
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 will be the priority of the thread */
sched_param.sched_priority = priority;
policy = SCHED_OTHER;

/* scheduling parameters of target thread */
ret = pthread_setschedparam(tid,
                             policy, &param);
```

サポートされている方針は、SCHED_FIFO、SCHED_RR、および SCHED_OTHER です。

pthread_setschedparam の戻り値

`pthread_setschedparam()` は、正常終了時に 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

説明: 設定しようとした属性の値が無効です。

EPERM

説明: 呼び出し元は、指定されたスレッドのスケジューリングパラメータまたはスケジューリングポリシーのいずれかを設定するための適切なアクセス権を持っていません。

ESRCH

説明: `tid` で指定した値が既存のスレッドを表していません。

スレッドの方針およびスケジューリングパラメータの取得

`pthread_getschedparam(3C)` は、個々のスレッドのスケジューリングポリシーおよびスケジューリングパラメータを取得します。

`pthread_getschedparam` の構文

```
int pthread_getschedparam(pthread_t tid, int *restrict policy,
                          struct sched_param *restrict param);

#include <pthread.h>

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

/* scheduling parameters of target thread */
ret = pthread_getschedparam (tid, &policy, &param);

/* sched_priority contains the priority of the thread */
priority = param.sched_priority;
```

`pthread_getschedparam` の戻り値

`pthread_getschedparam()` は、正常終了時に 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。

ESRCH

説明: `tid` で指定した値が既存のスレッドを表していません。

スレッド優先順位の設定

`pthread_setschedprio(3C)` は、指定されたスレッドのスケジューリング優先順位を設定します。

`pthread_setschedprio` の構文

```
int pthread_setschedprio(pthread_t tid, int prio);

#include <pthread.h>
```

```
pthread_t tid;
int prio;
int ret;
```

pthread_setschedprio の戻り値

pthread_setschedprio() は、正常終了時に 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

説明: *prio* の値は、指定されたスレッドのスケジューリングポリシーに対しては無効です。

ENOTSUP

説明: サポートされていない優先順位の値を設定しようとした。

EPERM

説明: 呼び出し元は、指定されたスレッドのスケジューリング優先順位を設定するための適切なアクセス権を持っていません。

ESRCH

説明: *tid* で指定した値が既存のスレッドを表していません。

スレッドへのシグナルの送信

スレッドにシグナルを送信するには、[pthread_kill\(3C\)](#) を使用します。

pthread_kill の構文

```
int pthread_kill(pthread_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.h\(3HEAD\)](#) 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。このエラーチェックは、*tid* の妥当性を検査する目的で使用できます。

pthread_kill の戻り値

pthread_kill() は、正常終了時に 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

説明: *sig* は正しいシグナル番号ではありません。

ESRCH

説明: 現在のプロセスに *tid* が存在しません。

呼び出しスレッドのシグナルマスクのアクセス

呼び出しスレッドのシグナルマスクを変更したり、検査したりするには、[pthread_sigmask\(3C\)](#) を使用します。

pthread_sigmask の構文

```
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); /* set new mask */
ret = pthread_sigmask(SIG_BLOCK, &new, &old); /* blocking mask */
ret = pthread_sigmask(SIG_UNBLOCK, &new, &old); /* unblocking */
```

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。*new* 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 の値が `NULL` の場合、*how* の値は重要ではありません。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。

pthread_sigmask の戻り値

`pthread_sigmask()` は、正常終了時に 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`

説明:*how* の値が定義されておらず、*old* が `NULL` です。

安全な fork

162 ページの「[解決策: pthread_atfork](#)」の `pthread_atfork(3C)` の説明を参照してください。

pthread_atfork の構文

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

pthread_atfork の戻り値

`pthread_atfork()` は、正常終了時に 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ます。以下の条件が検出されると `pthread_atfork()` は失敗し、対応する値を返します。

`ENOMEM`

説明: 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。

スレッドの終了

スレッドを終了するには、`pthread_exit(3C)` を使用します。

pthread_exit Syntax

```
void      pthread_exit(void *status);

#include <pthread.h>
void *status;
pthread_exit(status); /* exit with status */
```

`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、そのスレッド識別子と *status* によって示される終了状態が保持されます。これらのデータは、アプリケーションがそのスレッドを待機する `pthread_join()` を呼び出した時点で解放されます。スレッドが切り離されている場合は、*status* は無視されます。そのスレッド識別子はただちに再利用されます。スレッドの切り離しについては、[61 ページの「切り離し状態の設定」](#)を参照してください。

pthread_exit の戻り値

呼び出しスレッドは、終了状態 *status* で終了します。

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` を参照してください。
- `pthread_exit()` 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()` を参照してください。

デフォルトでは、ほか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` 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()` 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()` から処理途中で終了することがないように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` 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。プロセス内のそ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。

スレッドの取り消し

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ほかのスレッドの実行終了を要求することができます。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。

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ー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。スレッドの1つが最終的にタスクを完了させたのに、ほかのスレッドが動作し続ける場合があります。実行中のスレッドはその時点で何の役にも立っていないため、これらのスレッドは取り消されるべきです。

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` 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` または `pthread_cond_timedwait(3C)` で特定の条件の発生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、スレッドがブロックできる関数が含まれます。リストについては、[cancellation\(5\)](#) 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。場合によっては、アプリケーションで取り消しを無効(使用不可)にすることがあるかもしれません。取り消しを無効にすると、再度取り消し要求を有効にするまでの間、すべての取り消し要求が保留されます。

取り消しを無効にする方法については、[52 ページの「pthread_setcancelstate の構文」](#)を参照してください。

取り消しポイントの配置

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が割り当てられたままになり、メモリーを解放できなくなります。

標準C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。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)も定義されています。このライブラリを使用することにより、取り消しハンドラの有効範囲を定義して、意図した時点で、意図した場所に確実に作用するように設定できます。取り消しハンドラは、リソースと状態を起点と同じ条件に戻すクリーンアップサービスを提供します。

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(3C)` 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(3C)` は、同様に、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とおりです。

- 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
- `pthread_testcancel()` の呼び出しで

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。

スレッドの取り消し

スレッドを取り消すには、`pthread_cancel(3C)` を使用します。

`pthread_cancel` の構文

```
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(3C)` と `pthread_setcanceltype(3C)` があります。

pthread_cancel の戻り値

pthread_cancel() は、正常終了時に 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。

ESRCH

説明: 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

スレッドの取り消しを有効または無効にするには、[pthread_setcancelstate\(3C\)](#) を使用します。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。

pthread_setcancelstate の構文

```
int pthread_setcancelstate(int state, int *oldstate);

#include <pthread.h>

int oldstate;
int ret;

/* enabled */
ret = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);

/* disabled */
ret = pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);
```

pthread_setcancelstate の戻り値

pthread_setcancelstate() は、正常終了時に 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

説明: 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

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するには、[pthread_setcanceltype\(3C\)](#) を使用します。

pthread_setcanceltype の構文

```
int pthread_setcanceltype(int type, int *oldtype);

#include <pthread.h>

int oldtype;
int ret;

/* deferred mode */
ret = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &oldtype);

/* async mode*/
ret = pthread_setcanceltype(PTHREAD_CANCEL_ASYNCCHRONOUS, &oldtype);
```

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。

pthread_setcanceltype の戻り値

pthread_setcanceltype() は、正常終了時に 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

説明: PTHREAD_CANCEL_DEFERRED または PTHREAD_CANCEL_ASYNCCHRONOUS タイプではありません。

取り消しポイントの設定

スレッドの取り消しポイントを設定するには、[pthread_testcancel\(3C\)](#) を使用します。

pthread_testcancel の構文

```
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。詳細については、[50 ページの「取り消しポイント」](#)を参照してください。

`pthread_testcancel` の戻り値

`pthread_testcancel()` は値を返しません。

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、諸条件を起点のものと矛盾しない状態に復元するために使用します。矛盾しない状態には、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が含まれます。ハンドラの管理には、`pthread_cleanup_push(3C)` 関数と `pthread_cleanup_pop(3C)` 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` の構文

クリーンアップハンドラをクリーンアップスタック (LIFO) にプッシュするには、[`pthread\_cleanup\_push\(3C\)`](#) を使用します。

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

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

```
/* push the handler "routine" on cleanup stack */
pthread_cleanup_push (routine, arg);
```

`pthread_cleanup_push` の戻り値

`pthread_cleanup_push()` は値を返しません。

スタックからハンドラを取り出す

クリーンアップスタックからクリーンアップハンドラを取り出すには、[`pthread\_cleanup\_pop\(3C\)`](#) を使用します。

`pthread_cleanup_pop` の構文

```
void pthread_cleanup_pop(int execute);
```

```
#include <pthread.h>

/* pop the "func" out of cleanup stack and execute "func" */
pthread_cleanup_pop (1);

/* pop the "func" and DONT execute "func" */
pthread_cleanup_pop (0);
```

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()` 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。

`pthread_cleanup_pop` の戻り値

`pthread_cleanup_pop()` は値を返しません。

スレッドの属性

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。

属性オブジェクト

属性は、デフォルトのスレッド作成動作とは異なる動作を指定するための手段を提供します。`pthread_create(3C)` 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。

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。

- コードの移植性が高まります。
サポートされる属性が実装によって異なっても、スレッド実体を生成する関数呼び出しを変更する必要はありません。属性オブジェクトはインタフェースから隠されているため、これらの関数呼び出しは変更を必要としません。
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。
- 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する場合を考えます。それぞれのスレッドの集合は別個のサービスを提供します。それぞれの集合は独自の状態要件を持ちます。

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ローカライズされています。あと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。

pthread 関数を使用すると、スレッド属性オブジェクトを操作できます。これらの関数は、次の各節で説明されています。

- 58 ページの「属性の初期化」
- 60 ページの「属性の削除」
- 61 ページの「切り離し状態の設定」
- 62 ページの「切り離し状態の取得」
- 63 ページの「スタックガードのサイズの設定」
- 64 ページの「スタックガードのサイズの取得」
- 64 ページの「スコープの設定」
- 65 ページの「スコープの取得」
- 66 ページの「スレッドの多重度の設定」
- 66 ページの「スレッドの多重度の取得」
- 67 ページの「スケジューリングポリシーの設定」
- 68 ページの「スケジューリングポリシーの取得」
- 69 ページの「継承スケジューリングポリシーの設定」
- 70 ページの「継承スケジューリングポリシーの取得」
- 70 ページの「スケジューリングパラメータの設定」
- 71 ページの「スケジューリングパラメータの取得」
- 72 ページの「スタックについて」
- 74 ページの「スタックサイズの設定」
- 75 ページの「スタックサイズの取得」
- 76 ページの「スタックアドレスとサイズの設定」
- 77 ページの「スタックアドレスとサイズの取得」

属性の初期化

オブジェクト属性をデフォルト値に初期化するには、`pthread_attr_init(3C)` を使用します。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。

pthread_attr_initの構文

```
int pthread_attr_init(pthread_attr_t *tattr);

#include <pthread.h>

pthread_attr_t tattr;
int ret;

/* initialize an attribute to the default value */
ret = pthread_attr_init(&tattr);
```

表 3-1 は、属性 (*tattr*) のデフォルト値の一覧です。

表 3-1 *tattr* のデフォルト属性値

属性	値	結果
<i>scope</i>	PTHREAD_SCOPE_PROCESS	新しいスレッドはプロセス内のほかのスレッドと競合します。
<i>detachstate</i>	PTHREAD_CREATE_JOINABLE	スレッドの終了後に終了状態とスレッド ID が保存されます。
<i>stackaddr</i>	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持ちます。
<i>stacksize</i>	0	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サイズを持ちます。
<i>priority</i>	0	新しいスレッドは優先順位 0 を持ちます。
<i>inheritsched</i>	PTHREAD_EXPLICIT_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しません。
<i>schedpolicy</i>	SCHED_OTHER	新しいスレッドは、従来の Solaris タイムシェアリング (TS) スケジューリングクラスを使用します。
<i>guardsize</i>	PAGESIZE	スタックオーバーフローの保護。

注 - *inheritsched* 属性のデフォルト値は、将来の Solaris リリースでは `PTHREAD_EXPLICIT_SCHED` から `PTHREAD_INHERIT_SCHED` に変更される可能性があります。この変更によって問題が発生する可能性を避けるために、デフォルト値を受け入れるのではなく、`pthread_attr_setinheritsched()` を呼び出して *inheritsched* 属性を希望する値に設定するようにしてください。

pthread_attr_init の戻り値

`pthread_attr_init()` は、正常終了時に 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

説明: 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

初期化時に割り当てられた記憶領域を削除するには、`pthread_attr_destroy(3C)` を使用します。その属性オブジェクトは無効になります。

pthread_attr_destroy の構文

```
int pthread_attr_destroy(pthread_attr_t *tattr);
```

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

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

```
/* destroy an attribute */  
ret = pthread_attr_destroy(&tattr);
```

pthread_attr_destroy の戻り値

`pthread_attr_destroy()` は、正常終了時に 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

説明: *tattr* の値が無効です。

切り離し状態の設定

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。呼び出しスレッドがスレッドの終了を待機しない場合は、`pthread_attr_setdetachstate(3C)` を使用してください。

pthread_attr_setdetachstate(3C) の構文

```
int pthread_attr_setdetachstate(pthread_attr_t *tattr, int detachstate);
```

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;
/* set the thread detach state */
ret = pthread_attr_setdetachstate(&tattr, PTHREAD_CREATE_DETACHED);
```

PTHREAD_CREATE_JOINABLE により、スレッドが切り離されていない状態で生成された場合、アプリケーションはこのスレッドの完了を待機します。つまり、プログラムはスレッド上で `pthread_join()` を実行します。

スレッドが切り離された状態で生成されたか切り離されていない状態で生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。処理途中で `main()` から復帰することによって生じるプロセスの終了については、[49 ページの「スレッド終了処理の完了」](#)を参照してください。

注 - 明示的な同期によって阻止されなければ、そのスレッドの生成元が `pthread_create()` から復帰する前に、新たに生成される切り離されたスレッドが終了し、そのスレッド識別子が別の新しいスレッドに割り当てられることがあります。

切り離されていないスレッドには、終了後にほかのスレッドが終了待ちを行う必要があります。そうしないと、このスレッドのリソースが新しいスレッドに解放されず、メモリーリークが発生することになります。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;
```

例3-1 切り離されたスレッドの生成 (続き)

```
/* initialized with default attributes */
ret = pthread_attr_init (&tattr);
ret = pthread_attr_setdetachstate (&tattr, PTHREAD_CREATE_DETACHED);
ret = pthread_create (&tid, &tattr, start_routine, arg);
```

pthread_attr_setdetachstate の戻り値

pthread_attr_setdetachstate() は、正常終了時に 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

説明: *detachstate* または *tattr* の値が無効です。

切り離し状態の取得

スレッドの生成状態 (切り離された状態または切り離されていない状態) を取得するには、[pthread_attr_getdetachstate\(3C\)](#) を使用します。

pthread_attr_getdetachstate の構文

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

#include <pthread.h>
pthread_attr_t tattr;
int detachstate;
int ret;
/* get detachstate of thread */
ret = pthread_attr_getdetachstate (&tattr, &detachstate);
```

pthread_attr_getdetachstate の戻り値

pthread_attr_getdetachstate() は、正常終了時に 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

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

スタックガードのサイズの設定

attr オブジェクトの *guardsize* (ガードサイズ) を設定するには、`pthread_attr_setguardsize(3C)` を使用します。

`pthread_attr_setguardsize(3C)` の構文

```
#include <pthread.h>
int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);
```

guardsize 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。

- 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。アプリケーションが多くのスレッドを生成し、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分かっている場合には、ガード領域をオフにできます。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。
- 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 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。スレッドのスタックがガードとともに作成された場合、実装はスタックのオーバーフローの終わりに追加のメモリーを割り当てます。この追加メモリーは、スタックポインタのスタックオーバーフローに対するバッファーとして動作します。このバッファー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。

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。

準拠実装では、*guardsize* に含まれる値を、設定可能なシステム変数 `PAGESIZE` の倍数に切り上げることが認められています。`sys/mman.h` の `PAGESIZE` を参照してください。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。

`pthread_attr_setguardsize` の戻り値

`pthread_attr_setguardsize()` は、以下の場合に失敗します。

EINVAL

説明: 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。

スタックガードのサイズの取得

`attr` オブジェクトの `guardsize` (ガードサイズ) を取得するには、`pthread_attr_getguardsize(3C)` を使用します。

`pthread_attr_getguardsize` の構文

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

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

準拠実装では、`guardsize` に含まれる値を、設定可能なシステム変数 `PAGESIZE` の倍数に切り上げることが認められています。`sys/mman.h` の `PAGESIZE` を参照してください。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。

`pthread_attr_getguardsize` の戻り値

`pthread_attr_getguardsize()` は、以下の場合に失敗します。

`EINVAL`

説明: 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

スレッド `PTHREAD_SCOPE_SYSTEM` または `PTHREAD_SCOPE_PROCESS` の競合スコープを確立するには、`pthread_attr_setscope(3C)` を使用します。`PTHREAD_SCOPE_SYSTEM` スレッドは、システム内のすべてのスレッドと競合します。`PTHREAD_SCOPE_PROCESS` スレッドは、同じプロセス内のほかのスレッドと競合します。

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。

`pthread_attr_setscope` の構文

```
int pthread_attr_setscope(pthread_attr_t *tattr,int scope);
```

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

```
pthread_attr_t tattr;
```

```

int ret;

/* bound thread */
ret = pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);

/* unbound thread */
ret = pthread_attr_setscope(&tattr, PTHREAD_SCOPE_PROCESS);

```

この例には、3つの関数呼び出しがあります。1つめは属性を初期化する呼び出し、2つめはデフォルト属性を変更する呼び出し、3つめはpthreadを生成する呼び出しです。

```

#include <pthread.h>

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

/* initialized with default attributes */
ret = pthread_attr_init (&tattr);

ret = pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);
ret = pthread_create (&tid, &tattr, start_routine, arg);

```

pthread_attr_setscopeの戻り値

pthread_attr_setscope() は、正常終了時に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

説明: *tattr* に設定しようとした値は無効です。

スコープの取得

スレッドスコープを取得するには、[pthread_attr_getscope\(3C\)](#) を使用します。

pthread_attr_getscopeの構文

```

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

#include <pthread.h>

pthread_attr_t tattr;

```

```
int scope;
int ret;

/* get scope of thread */
ret = pthread_attr_getscope(&tattr, &scope);
```

pthread_attr_getscopeの戻り値

pthread_attr_getscope() は、正常終了時に 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

説明: *scope* の値が NULL か、*tattr* の値が無効です。

スレッドの多重度の設定

[pthread_setconcurrency\(3C\)](#) は、標準規格に準拠するための関数です。アプリケーションは、この関数を使用して、スレッドライブラリに目標多重度を通知します。() Solaris 9 リリースに実装されているスレッドでは、このインタフェースは無効です。実行可能スレッドはすべて LWP に接続されます。

pthread_setconcurrencyの構文

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

```
int pthread_setconcurrency(int new_level);
```

pthread_setconcurrencyの戻り値

pthread_setconcurrency() は、次の条件で失敗します。

EINVAL

説明: *new_level* で指定された値が負の値です。

EAGAIN

説明: *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。

スレッドの多重度の取得

[pthread_getconcurrency\(3C\)](#) は、pthread_setconcurrency() を前回呼び出したときに設定された値を返します。

pthread_getconcurrency の構文

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

```
int pthread_getconcurrency(void);
```

pthread_setconcurrency() 関数が以前に呼び出されていない場合は、pthread_getconcurrency() は 0 を返します。

pthread_getconcurrency の戻り値

pthread_getconcurrency() は常に、pthread_setconcurrency() を前回呼び出したときに設定された多重度を返します。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。

スケジューリングポリシーの設定

スケジューリングポリシーを設定するには、[pthread_attr_setschedpolicy\(3C\)](#) を使用します。POSIX 規格ではスケジューリングポリシーの値として、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。Solaris OS の場合、SCHED_OTHER スレッドは、従来のタイムシェアリング (TS) スケジューリングクラスで実行されます。

pthread_attr_setschedpolicy(3C) の構文

```
int pthread_attr_setschedpolicy(pthread_attr_t *tattr, int policy);
```

```
#include <pthread.h>
pthread_attr_t tattr;
int policy;
int ret;
```

```
/* set the scheduling policy to SCHED_OTHER */
ret = pthread_attr_setschedpolicy(&tattr, SCHED_OTHER);
```

■ SCHED_FIFO

先入れ先出しスレッドはリアルタイム (RT) スケジューリングクラスで実行され、呼び出しプロセスは特権付きである必要があります。このようなスレッドは、優先順位のより高いスレッドに割り込まれなければ、CPU を譲るかブロックされるまで実行を続けます。

■ SCHED_RR

ラウンドロビン。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、呼び出しプロセスの実効ユーザー ID が 0 であれば、リアルタイム (RT) スケジューリングクラスに設定されます。これらのスレッドは、優先順位のより高いスレッドに割り込まれなければ、または CPU を譲るかブ

ロックされなければ、システムに定義された期間実行を続けます。スケジューリング競合範囲がプロセススコープ (PTHREAD_SCOPE_PROCESS) のスレッドには、SCHED_RR を使用します。これは、TS スケジューリングクラスに基づいています。また、これらのスレッドの呼び出しプロセスは、実効ユーザー ID が 0 以外です。

ラウンドロビンスレッドはリアルタイム (RT) スケジューリングクラスで実行され、呼び出しプロセスは特権付きである必要があります。ラウンドロビンスレッドは、優先順位のより高いスレッドに割り込まれなければ、または CPU を譲るかブロックされなければ、システムに定義された期間実行を続けます。その時点で、優先順位の等しい別のリアルタイムスレッドに強制的に CPU が譲られます。

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

スケジューリングの詳細については、[26 ページの「スレッドのスケジューリング」](#)を参照してください。

pthread_attr_setschedpolicy の戻り値

pthread_attr_setschedpolicy() は、正常終了時に 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

説明: *tattr* に設定しようとした値は無効です。

ENOTSUP

説明: サポートされていない属性値を設定しようとした。

スケジューリングポリシーの取得

スケジューリングポリシーを取得するには、[pthread_attr_getschedpolicy\(3C\)](#) を使用します。

pthread_attr_getschedpolicy の構文

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

```
#include <pthread.h>  
pthread_attr_t tattr;  
int policy;  
int ret;
```

```
/* get scheduling policy of thread */
ret = pthread_attr_getschedpolicy (&tattr, &policy);
```

pthread_attr_getschedpolicy の戻り値

pthread_attr_getschedpolicy() は、正常終了時に 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

説明: *policy* の値が NULL か、*tattr* の値が無効です。

継承スケジューリングポリシーの設定

継承スケジューリングポリシーを設定するには、[pthread_attr_setinheritsched\(3C\)](#) を使用します。

pthread_attr_setinheritsched の構文

```
int pthread_attr_setinheritsched(pthread_attr_t *tattr, int inheritsched);
```

```
#include <pthread.h>
pthread_attr_t tattr;
int inheritsched;
int ret;
```

```
/* use creating thread's scheduling policy and priority*/
ret = pthread_attr_setinheritsched(&tattr, PTHREAD_INHERIT_SCHED);
```

inheritsched の値が PTHREAD_INHERIT_SCHED の場合は、生成スレッドのスケジューリングポリシーおよび優先順位が、作成されたスレッドのために使用されます。属性構造内のスケジューリングポリシーおよび優先順位は無視されます。*inheritsched* の値が PTHREAD_EXPLICIT_SCHED の場合は、属性構造のスケジューリングポリシーおよび優先順位が、作成されたスレッドのために使用されます。この場合、呼び出し元は、pthread_create() が成功するための十分な特権を持っている必要があります。

pthread_attr_setinheritsched の戻り値

pthread_attr_setinheritsched() は、正常終了時に 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

説明: *tattr* に設定しようとした値は無効です。

継承スケジューリングポリシーの取得

`pthread_attr_getinheritsched(3C)` は、属性構造に含まれている *inheritsched* 属性を返します。

`pthread_attr_getinheritsched` の構文

```
int pthread_attr_getinheritsched(pthread_attr_t *restrict tattr,
                                int *restrict inheritsched);
```

```
#include <pthread.h>
pthread_attr_t tattr;
int inheritsched;
int ret;
```

```
ret = pthread_attr_getinheritsched (&tattr, &inheritsched);
```

`pthread_attr_getinheritsched` の戻り値

`pthread_attr_getinheritsched()` は、正常終了時に 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

説明: *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ータの設定

`pthread_attr_setschedparam(3C)` は、スケジューリングパラメータを設定します。

`pthread_attr_setschedparam` の構文

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

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;
int newprio;
struct sched_param param;
newprio = 30;
/* set the priority; others are unchanged */
param.sched_priority = newprio;
/* set the new scheduling param */
ret = pthread_attr_setschedparam (&tattr, &param);
```

スケジューリングパラメータは *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のパラメータだけです。

pthread_attr_setschedparam の戻り値

pthread_attr_setschedparam() は、正常終了時に 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

説明: *param* の値が NULL か、*tattr* の値が無効です。

pthread の優先順位は、次のいずれかの方法で管理できます。

- 子スレッドを生成する前に優先順位属性を設定できます。
- 親スレッドの優先順位を変更して、また元に戻す

スケジューリングパラメータの取得

pthread_attr_getschedparam(3C) は、pthread_attr_setschedparam() で定義されたスケジューリングパラメータを返します。

pthread_attr_getschedparam の構文

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

```
#include <pthread.h>
pthread_attr_t attr;
struct sched_param param;
int ret;
/* get the existing scheduling param */
ret = pthread_attr_getschedparam (&tattr, &param);
```

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 構造体で指定された新しい優先順位で生成されます。この構造体にはその他のスケジューリング情報も含まれます。

優先順位を設定したスレッドの生成例

例 3-2 は、親スレッドとは異なった優先順位を持つ子スレッドを生成する例です。

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;

/* initialized with default attributes */
ret = pthread_attr_init (&tattr);

/* safe to get existing scheduling param */
ret = pthread_attr_getschedparam (&tattr, &param);

/* set the priority; others are unchanged */
param.sched_priority = newprio;

/* setting the new scheduling param */
ret = pthread_attr_setschedparam (&tattr, &param);

/* specify explicit scheduling */
ret = pthread_attr_setinheritsched (&tattr, PTHREAD_EXPLICIT_SCHED);

/* with new priority specified */
ret = pthread_create (&tid, &tattr, func, arg);
```

pthread_attr_getschedparamの戻り値

pthread_attr_getschedparam() は、正常終了時に 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

説明: *param* の値が NULL か、*tattr* の値が無効です。

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ます。指定したサイズはいずれも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はスタックのオーバーフローの最後に付加されます。このため、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。呼び出し側で割り当てられたスレッドスタックは、そのまま使用されます。

スタックを指定するときは、スレッドを `PTHREAD_CREATE_JOINABLE` として生成する必要があります。このスタックは、そのスレッドに対する `pthread_join(3C)` 呼び出しが終了まで解放できません。これは、スレッドが終了するまで、そのスレッドのスタックを解放できないからです。こうしたスレッドが確実に終了したことを確認するには、`pthread_join(3C)` を使用します。

スレッドへのスタック空間の割り当て

通常、スレッドにスタック空間を割り当てる必要はありません。システムは、各スレッドのスタックに対して、1M バイト (32 ビットシステムの場合) または 2M バイト (64 ビットシステムの場合) の仮想メモリを割り当てます。スワップ空間は予約されません。システムは、`mmap()` の `MAP_NORESERVE` オプションを使って割り当てを行います。

システムによって生成されたスレッドスタックは、それぞれレッドゾーンを持っています。システムはレッドゾーンとして、スタックオーバーフローを捕捉するためのページをスタックのオーバーフローの最後に付加します。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、そのサイズがアプリケーションで指定されたかデフォルトのサイズであるかに関係なく行われます。

注-実行時のスタック要件はライブラリ呼び出しや動的リンクによって異なります。したがって、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たすようにしなければなりません。

スタックとスタックサイズの一方または両方を指定するのが適正である場合はまれです。専門家であっても、適切なサイズを指定したかどうかを判断するのは困難です。ABI 準拠のプログラムでも、スタックサイズを静的に判定することはできません。スタックサイズは、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サイズを指定するときは、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、スレッドでデフォルトスタックサイズを超えるスタック空間が必要になる場合があります。また、少しわ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場合、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可能性があります。

スタックサイズ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。

スタックサイズの絶対最小値を取得するには、マクロ `PTHREAD_STACK_MIN` を呼び出します。マクロ `PTHREAD_STACK_MIN` は、`NULL` 手続きを実行するスレッドに必要な量のスタック空間を返します。実用的なスレッドに必要なスタック空間は最小スタックサイズより大きいので、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。

スタックサイズの設定

`pthread_attr_setstacksize(3C)` は、スレッドのスタックサイズを設定します。

`pthread_attr_setstacksize` の構文

```
int pthread_attr_setstacksize(pthread_attr_t *tattr, size_t size);
```

```
#include <pthread.h>
#include <limits.h>
pthread_attr_t tattr;
size_t size;
int ret;
size = (PTHREAD_STACK_MIN + 0x4000);
/* setting a new size */
ret = pthread_attr_setstacksize(&tattr, size);
```

`size` 属性は、システムが割り当てるスタックのサイズ(バイト数)を定義します。このサイズ(`size`)は、システムで定義された最小のスタックサイズを下回ってはいけません。詳細については、[72 ページの「スタックについて」](#)を参照してください。

`size` には、新しいスレッドが使用するスタックのバイト数が含まれます。`size` の値が 0 ならば、デフォルトのサイズが使われます。通常は 0 を指定してください。

`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量です。このスタック空間には、アプリケーションコードを実行するための、スレッドルーチンの要求量は考慮されていません。

例 3-3 スタックサイズの設定の例

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

pthread_attr_t tattr;
pthread_t tid;
int ret;
```

例3-3 スタックサイズの設定の例 (続き)

```

size_t size = PTHREAD_STACK_MIN + 0x4000;

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);

/* setting the size of the stack also */
ret = pthread_attr_setstacksize(&tattr, size);

/* only size specified in tattr */
ret = pthread_create(&tid, &tattr, start_routine, arg);

```

pthread_attr_setstacksizeの戻り値

pthread_attr_setstacksize() は、正常終了時に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

説明: *size* の値が PTHREAD_STACK_MIN より小さいか、またはシステムの制限を超過しているか、または *tattr* が無効です。

スタックサイズの取得

pthread_attr_getstacksize(3C) は、pthread_attr_setstacksize() で設定されたスタックサイズを返します。

pthread_attr_getstacksizeの構文

```

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

#include <pthread.h>
pthread_attr_t tattr;
size_t size;
int ret;
/* getting the stack size */
ret = pthread_attr_getstacksize(&tattr, &size);

```

pthread_attr_getstacksizeの戻り値

pthread_attr_getstacksize() は、正常終了時に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

説明: *tattr* または *size* が無効です。

スタックアドレスとサイズの設定

`pthread_attr_setstack(3C)` は、スタックアドレスとサイズを設定します。

`pthread_attr_setstack(3C)` の構文

```
int pthread_attr_setstack(pthread_attr_t *tattr, void *stackaddr, size_t stacksize);
```

```
#include <pthread.h>
#include <limits.h>
pthread_attr_t tattr;
void *base;
size_t size;
int ret;
base = (void *) malloc(PTHREAD_STACK_MIN + 0x4000);
/* setting a new address and size */
ret = pthread_attr_setstack(&tattr, base, PTHREAD_STACK_MIN + 0x4000);
```

stackaddr 属性は、スレッドのスタックのベースを定義します。*stacksize* 属性は、スタックサイズを指定します。*stackaddr* にデフォルトの `NULL` 以外が設定された場合、そのスタックはそのアドレスで初期化されます。このとき、スタックサイズは *stacksize* とみなされます。

base には、新しいスレッドが使用するスタックのアドレスが含まれます。*base* が `NULL` の場合、`pthread_create(3C)` は新しいスレッドに *stacksize* バイト以上のスタックを割り当てます。

`pthread_attr_setstack(3C)` の戻り値

`pthread_attr_setstack()` は、正常終了時に 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

説明: *base* または *tattr* の値が正しくありません。*stacksize* の値が `PTHREAD_STACK_MIN` 未満です。

次の例は、独自のスタックアドレスとサイズ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;
size_t size;

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);

/* setting the base address and size of the stack */
ret = pthread_attr_setstack(&tattr, stackbase, size);

/* address and size specified */
ret = pthread_create(&tid, &tattr, func, arg);
```

スタックアドレスとサイズの取得

[pthread_attr_getstack\(3C\)](#) は、`pthread_attr_setstack()` によって設定されたスレッドスタックアドレスおよびサイズを返します。

pthread_attr_getstack の構文

```
int pthread_attr_getstack(pthread_attr_t *restrict tattr,
                          void **restrict stackaddr, size_t *restrict stacksize);
```

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

```
pthread_attr_t tattr;
void *base;
size_t size;
int ret;

/* getting a stack address and size */
ret = pthread_attr_getstack (&tattr
, &base, &size);
```

pthread_attr_getstack の戻り値

`pthread_attr_getstack()` は、正常終了時に 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

説明: *tattr* の値が不正です。

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タイプについて説明します。また、同期を使用する場合と方法についても説明します。

- 80 ページの「相互排他ロック属性」
- 94 ページの「相互排他ロックの使用方法」
- 108 ページの「スピンロックの使用」
- 112 ページの「条件変数の属性」
- 118 ページの「条件変数の使用方法」
- 131 ページの「セマフォによる同期」
- 139 ページの「読み取り/書き込みロック属性」
- 142 ページの「読み取り/書き込みロックの使用」
- 151 ページの「バリアー同期の使用」
- 156 ページの「プロセスの境界を越えた同期」
- 157 ページの「プリミティブの比較」

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。異なるプロセスのスレッドは、スレッドによって制御されている共有メモリー内の同期オブジェクトによって情報をやり取りできます。通常、異なるプロセスのスレッドは互いに見えませんが、これらのスレッドは情報をやり取りできます。

同期オブジェクトをファイル内に配置することもできます。そうすれば、作成元のプロセスの消滅後も同期オブジェクトは存続します。

次の同期オブジェクトがあります。

- 相互排他ロック
- 条件変数
- 読み取り/書き込みロック
- セマフォ

同期は、次のような状況で効果を発揮します。

- 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、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にアクセスするときには、1回のメモリーロードで複数のメモリーサイクルを使用することがあります。整数がバスのデータ幅にそろっていない場合や、バスのデータ幅より大きい場合、複数のメモリーサイクルが必要になります。こうした状況はSPARC版アーキテクチャーのマシンでは発生しませんが、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ーでは、long long は不可分ではありません。不可分操作は、それ以上小さい操作に分割できません。long long は、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 アーキテクチャーマシンと x86 マシンでは不可分です。

相互排他ロック属性

相互排他ロック (mutex) 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、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を使用します。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。

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、mutex 属性を操作する関数を示します。

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	81 ページの「pthread_mutexattr_init の構文」
mutex 属性オブジェクトの削除	83 ページの「pthread_mutexattr_destroy の構文」

表 4-1 mutex 属性ルーチン (続き)

操作	参照先
mutex のスコープ設定	83 ページの「 pthread_mutexattr_setpshared の構文 」
mutex のスコープの値の取得	84 ページの「 pthread_mutexattr_getpshared の構文 」
mutex の型属性の設定	85 ページの「 pthread_mutexattr_settype の構文 」
mutex の型属性の取得	86 ページの「 pthread_mutexattr_gettype の構文 」
mutex 属性のプロトコルの設定	86 ページの「 pthread_mutexattr_setprotocol の構文 」
mutex 属性のプロトコルの取得	88 ページの「 pthread_mutexattr_getprotocol の構文 」
mutex 属性の優先順位上限の設定	89 ページの「 pthread_mutexattr_setprioceiling の構文 」
mutex 属性の優先順位上限の取得	89 ページの「 pthread_mutexattr_getprioceiling の構文 」
mutex の優先順位上限の設定	90 ページの「 pthread_mutex_setprioceiling の構文 」
mutex の優先順位上限の取得	91 ページの「 pthread_mutex_getprioceiling の構文 」
mutex の堅牢度属性の設定	92 ページの「 pthread_mutexattr_setrobust_np の構文 」
mutex の堅牢度属性の取得	93 ページの「 pthread_mutexattr_getrobust_np の構文 」

mutex 属性オブジェクトの初期化

mutex 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するには、[pthread_mutexattr_init\(3C\)](#) を使用します。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。

pthread_mutexattr_init の構文

```
int pthread_mutexattr_init(pthread_mutexattr_t *mattr);
```

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

```
pthread_mutexattr_t mattr;
int ret;
```

```
/* initialize an attribute to default value */
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格納します。*mattr* オブジェクト内の属性については、[表 4-2](#) を参照してください。

mutex 属性オブジェクトを再度初期化する場合は、あらかじめ pthread_mutexattr_destroy(3C) を呼び出して、オブジェクトを削除しておく必要があります。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。

表 4-2 *matrr* のデフォルトの属性値

属性	値	結果
<i>pshared</i>	PTHREAD_PROCESS_PRIVATE	初期化された mutex を 1 つのプロセス内で使用できます。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。
<i>type</i>	PTHREAD_MUTEX_DEFAULT	Solaris Pthreads 実装では、PTHREAD_MUTEX_DEFAULT は、デッドロックを検出しない PTHREAD_MUTEX_NORMAL に割り当てられます。
プロトコル	PTHREAD_PRIO_NONE	スレッドの優先順位とスケジューリングは、そのスレッドが所有する mutex の優先順位の影響を受けません。
<i>prioceiling</i>	-	<i>prioceiling</i> の値は、sched_get_priority_min() および sched_get_priority_max() 関数によって返される、SCHED_FIFO 方針に関する既存の優先順位の範囲から取得されます。この優先順位の範囲は、mutex が作成された Solaris バージョンによって決定されます。
<i>robustness</i>	PTHREAD_MUTEX_STALLED_NP	mutex の所有者が終了すると、この mutex に関する pthread_mutex_lock() への将来のすべての呼び出しは進行過程からブロックされます。

pthread_mutexattr_init の戻り値

pthread_mutexattr_init() は、正常終了時に 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

説明: 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(3C)` 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。

pthread_mutexattr_destroy の構文

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *mattr)
```

```
#include <pthread.h>
pthread_mutexattr_t mattr;
int ret;
/* destroy an attribute */
ret = pthread_mutexattr_destroy(&mattr);
```

pthread_mutexattr_destroy の戻り値

`pthread_mutexattr_destroy()` は、正常終了時に 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

説明: *mattr* で指定された値が無効です。

mutex のスコープ設定

`pthread_mutexattr_setpshared(3C)` は、mutex 変数のスコープを設定します。

pthread_mutexattr_setpshared の構文

```
int pthread_mutexattr_setpshared(pthread_mutexattr_t *restrict mattr,
    int *restrict pshared);
```

```
#include <pthread.h>
pthread_mutexattr_t mattr;
int ret;
ret = pthread_mutexattr_init(&mattr);
/* * resetting to its default value: private */
ret = pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_PRIVATE);
```

mutex 変数のスコープは、プロセス専用 (プロセス内) またはシステム共通 (プロセス間) です。複数のプロセスのスレッド間で mutex を共有させるには、*pshared* 属性を `PTHREAD_PROCESS_SHARED` に設定して、共有メモリー内に mutex を生成します。

`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。

pthread_mutexattr_setpshared の戻り値

`pthread_mutexattr_setpshared()` は、正常終了時に 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

説明:*mutex* 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(3C)` は、`pthread_mutexattr_setpshared()` によって定義された `mutex` 変数のスコープを返します。

pthread_mutexattr_getpshared の構文

```
int pthread_mutexattr_getpshared(pthread_mutexattr_t *restrict mutex,
                                  int *restrict pshared);
```

```
#include <pthread.h>
pthread_mutexattr_t mutex;
int pshared, ret;
/* get pshared of mutex */
ret = pthread_mutexattr_getpshared(&mutex, &pshared);
```

属性オブジェクト *mutex* 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。

pthread_mutexattr_getpshared の戻り値

`pthread_mutexattr_getpshared()` は、正常終了時に 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

説明:*mutex* で指定された値が無効です。

mutex の型属性の設定

`pthread_mutexattr_settype(3C)` は、`mutex` の型 (*type*) 属性を設定します。

pthread_mutexattr_settype の構文

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

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

型 (*type*) 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。以下に、有効な mutex 型を示します。

PTHREAD_MUTEX_NORMAL

説明: 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、mutex 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

説明: 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、エラーを返します。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、エラーが返されます。また、ロック解除された mutex をロック解除しようとする、エラーを返します。

PTHREAD_MUTEX_RECURSIVE

説明: 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、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、エラーが返されます。また、ロック解除された mutex をロック解除しようとする、エラーを返します。

PTHREAD_MUTEX_DEFAULT

説明: この属性は、ほかの mutex 型に割り当てることができます。Solaris の実装では、この属性は PTHREAD_PROCESS_NORMAL に割り当てられます。

pthread_mutexattr_settype の戻り値

pthread_mutexattr_settype 関数は、正常終了時に 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ます。

EINVAL

説明: *type* または *attr* の値が無効です。

mutex の型属性の取得

[pthread_mutexattr_gettype\(3C\)](#) 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。

pthread_mutexattr_gettype の構文

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

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

型 (*type*) 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`

それぞれの型については、85 ページの「[pthread_mutexattr_settype の構文](#)」を参照してください。

pthread_mutexattr_gettype の戻り値

`pthread_mutexattr_gettype()` は、正常終了時に 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ます。

`EINVAL`

説明: *type* で指定された値が無効です。

mutex 属性のプロトコルの設定

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

pthread_mutexattr_setprotocol の構文

```
#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 に定義されている *protocol* の値は、次の値のうちのいずれか 1 つです。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 の所有権の影響を受けません。

■ PTHREAD_PRIO_INHERIT

このプロトコル値は、所有しているスレッドの優先順位とスケジューリングに影響します。より優先順位の高いスレッドが thrd1 の所有する 1 つまたは複数の mutex 上でブロックされている場合、これらの mutex が PTHREAD_PRIO_INHERIT を使用して初期化されていると、thrd1 はより高い優先順位、すなわち 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。

優先順位の逆転を避けるには、PTHREAD_PRIO_INHERIT を使用します。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ません。

優先順位の継承を使用しない場合は、優先順位の低いスレッドが長期間実行をスケジュールされないために、優先順位の高いスレッドが同じ期間ブロックされる可能性があります。優先順位の継承では、優先順位の低いスレッドの優先順位を一時的に上げることによって、そのスレッドがすばやく実行をスケジュールされてロックを解放し、それによって優先順位の高いスレッドがそのロックを獲得できるようにします。優先順位の低いスレッドは、ロックを解放すると、元の低い優先順位に戻ります。

■ PTHREAD_PRIO_PROTECT

スレッドが PTHREAD_PRIO_PROTECT で初期化された mutex を 1 つ以上所有している場合、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 の中でもっと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。

PTHREAD_PRIO_INHERIT および PTHREAD_PRIO_PROTECT mutex 属性を使用できるのは、リアルタイム (RT) スケジューリングクラス SCHED_FIFO または SCHED_RR で実行されている特権付きプロセスだけです。

スレッドは、PTHREAD_PRIO_INHERIT で初期化された mutex と PTHREAD_PRIO_PROTECT で初期化された mutex を複数同時に所有できます。この場合、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。

pthread_mutexattr_setprotocol の戻り値

pthread_mutexattr_setprotocol() は、正常終了時に 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。

EINVAL

説明: *attr* または *protocol* に指定した値は無効です。

EPERM

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

pthread_mutexattr_getprotocol の構文

```
#include <pthread.h>
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *restrict attr,
int *restrict 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、次のプロトコル属性のうちいずれか 1 つが含まれています。PTHREAD_PRIO_NONE、PTHREAD_PRIO_INHERIT、または PTHREAD_PRIO_PROTECT です。これらはヘッダー <pthread.h> で定義されています。

pthread_mutexattr_getprotocol の戻り値

pthread_mutexattr_getprotocol() は、正常終了時に 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ます。

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

説明: *attr* で指定された値が NULL か、*attr* または *protocol* で指定された値が無効です。

EPERM

説明: 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(3C)` 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。

pthread_mutexattr_setprioceiling の構文

```
#include <pthread.h>
int pthread_mutexattr_setprioceiling(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 mutex 属性オブジェクトを指します。

prioceiling には、初期化された mutex の優先順位上限を指定します。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もっと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。

pthread_mutexattr_setprioceiling の戻り値

`pthread_mutexattr_setprioceiling()` は、正常終了時に 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ます。

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

説明: *attr* で指定された値が NULL または無効であるか、*prioceiling* が無効です。

EPERM

説明: 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(3C)` 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。

pthread_mutexattr_getprioceiling の構文

```
#include <pthread.h>
int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *restrict attr,
int *restrict 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。

`pthread_mutexattr_getprioceiling()` は、初期化された `mutex` の優先順位上限を `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もっと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。

`pthread_mutexattr_getprioceiling` の戻り値

`pthread_mutexattr_getprioceiling()` は、正常終了時に 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ます。

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

説明: `attr` で指定された値が NULL です。

EPERM

説明: 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_mutexattr_setprioceiling(3C)` は、`mutex` の優先順位上限を設定します。

`pthread_mutex_setprioceiling` の構文

```
#include <pthread.h>
int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mutex,
    int prioceiling, int *restrict 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、`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。

`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。

`pthread_mutex_setprioceiling` の戻り値

`pthread_mutex_setprioceiling()` は、正常終了時に 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ます。

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

説明:*prioceiling* で要求された優先順位が範囲外です。

EINVAL

説明:`mutex` が、`THREAD_PRIO_PROTECT` の値を持つ *protocol* 属性で初期化されていませんでした。

EPERM

説明: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_mutexattr_getprioceiling(3C)` は、`mutex` の優先順位上限を取得します。

pthread_mutex_getprioceiling の構文

```
#include <pthread.h>
int pthread_mutex_getprioceiling(const pthread_mutex_t *restrict mutex,
                                int *restrict prioceiling);
```

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

pthread_mutex_getprioceiling の戻り値

`pthread_mutex_getprioceiling()` は、正常終了時に 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。

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

説明:*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。

EPERM

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

pthread_mutexattr_setrobust_np の構文

```
#include <pthread.h>
int pthread_mutexattr_setrobust_np(pthread_mutexattr_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。

Solaris 10 および以前のリリースでは、`PTHREAD_MUTEX_ROBUST_NP` 属性は、自身も `PTHREAD_PRIO_INHERIT` プロトコル属性でマークされている mutex にのみ適用できます。この制限は、それ以降の Solaris リリースでは解消されています。

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 (通常はプロセスが異常終了したために) mutex をロック解除しないで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_STALLED_NP`

mutex の所有者が mutex をロック解除しないで終了すると、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_ROBUST_NP`

mutex の所有者が mutex をロック解除しないで終了すると、mutex はロック解除されます。この mutex の次の所有者がその mutex を獲得し、エラーコード `OWNERDEAD` が返されます。

注 - アプリケーションは `pthread_mutex_lock()` からのリターンコードを常にチェックして、`PTHREAD_MUTEX_ROBUST_NP` 属性で初期化された mutex がないかどうかを確認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()` を呼び出してから `mutex` をロック解除します。これによって `mutex` は整合性があるとしてマークされるため、以降の `pthread_mutex_lock()` および `pthread_mutex_unlock()` の呼び出しが通常の方法で動作します。
- 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_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。ただし、この `mutex` によって保護されていた状態は不整合のままになるため、何らかの形式のアプリケーション復旧が必要になります。

`EOWNERDEAD` を持つロックを獲得したスレッドが `mutex` をロック解除しないで終了すると、次の所有者が `EOWNERDEAD` のリターンコードを持つロックを獲得します。

`pthread_mutexattr_setrobust_np` の戻り値

`pthread_mutexattr_setrobust_np()` は、正常終了時に 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ます。

次の条件が検出されると、`pthread_mutexattr_setrobust_np()` は失敗する可能性があります。

`EINVAL`

説明: *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(3C)` は、`mutex` 属性オブジェクトの堅牢度属性を取得します。

`pthread_mutexattr_getrobust_np` の構文

```
#include <pthread.h>
int pthread_mutexattr_getrobust_np(const pthread_mutexattr_t *attr, int *robustness);
```

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。

pthread_mutexattr_getrobust_npの戻り値

pthread_mutexattr_getrobust_np() は、正常終了時に 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ます。

次の条件が検出されると、pthread_mutexattr_getrobust_np() は失敗する可能性があります。

EINVAL
説明: 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法

表 4-3 に、相互排他ロックを操作する関数を示します。

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	95 ページの「pthread_mutex_init の構文」
mutex の整合性保持	96 ページの「pthread_mutex_consistent_np の構文」
mutex のロック	97 ページの「pthread_mutex_lock の構文」
mutex のロック解除	98 ページの「pthread_mutex_unlock の構文」
ブロックしないで行う mutex のロック	99 ページの「pthread_mutex_trylock の構文」
指定した時間までの mutex のロック	100 ページの「pthread_mutex_timedlock() の構文」
指定した時間間隔内の mutex のロック	101 ページの「pthread_mutex_reltimedlock_np() の構文」
mutex の削除	102 ページの「pthread_mutex_destroy の構文」

デフォルトスケジューリングポリシー 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 SCHED_OTHER 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。SCHED_FIFO および SCHED_RR リアルタイムスケジューリングポリシーの下では、優先順位に従って待機スレッドをブロック解除します。

mutex の初期化

mp が指す mutex をデフォルト値に初期化する場合、またはすでに pthread_mutexattr_init() で設定されている mutex 属性を指定する場合は、pthread_mutex_init(3C) を使用します。mattr のデフォルト値は NULL です。

pthread_mutex_init の構文

```
int pthread_mutex_init(pthread_mutex_t *restrict mp,
                       const pthread_mutexattr_t *restrict mattr);
```

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

```
pthread_mutex_t mp = PTHREAD_MUTEX_INITIALIZER;
pthread_mutexattr_t mattr;
int ret;
```

```
/* initialize a mutex to its default value */
ret = pthread_mutex_init(&mp, NULL);
```

```
/* initialize a mutex */
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内または個々のプロセス専用のメモリ内に置かれます。

注 - PTHREAD_MUTEX_ROBUST_NP 属性で初期化される mutex の場合は、初期化の前に mutex メモリをゼロにクリア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、メモリのオーバーヘッドがありません。

静的に定義された mutex をデフォルト値に初期化するには、マクロ PTHREAD_MUTEX_INITIALIZER を使用します。

ほかのスレッドによって使用されている mutex は、初期化したり削除したりできません。初期化や削除の動作が正常に行われないと、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になければなりません。

pthread_mutex_init の戻り値

pthread_mutex_init() は、正常終了時に 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。

EBUSY

説明: *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

説明: *mattr* 属性値が無効です。その mutex は変更されていません。

EFAULT

説明: *mp* が指す *mutex* のアドレスが無効です。

mutex の整合性保持

堅牢な *mutex* の所有者がその *mutex* をロック解除しないで終了すると、その *mutex* はロック解除され、不整合としてマークされます。次の所有者が **EOWNERDEAD** のリターンコードを持つロックを獲得します。

`pthread_mutex_consistent_np()` は、その所有者が処理を終了したあとも *mutex* オブジェクト *mutex* の整合性を保持します。

pthread_mutex_consistent_np の構文

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

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、所有者の処理の失敗のため不整合の状態のままになっていることがあります。この場合は、この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。

pthread_mutex_consistent_np の戻り値

`pthread_mutex_consistent_np()` は、正常終了時に 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ます。

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

EINVAL

説明: 現在のスレッドが *mutex* を所有していないか、または *mutex* が不整合の状態を持つ **PTHREAD_MUTEX_ROBUST_NP** *mutex* ではありません。

mutex のロック

mutex が指す *mutex* をロックするには、`pthread_mutex_lock(3C)` を使用します。

pthread_mutex_lockの構文

```
int pthread_mutex_lock(pthread_mutex_t *mutex);

#include <pthread.h>

pthread_mutex_t mutex;
int ret;

ret = pthread_mutex_lock(&mp); /* acquire the mutex */
```

mutex は、pthread_mutex_lock() から制御が戻るとロックされます。呼び出しスレッドが所有者になります。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。

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うデッドロックが発生します。スレッドが、スレッドによって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、エラーチェックが行われます。スレッドが、すでにロックした mutex をもう一度ロックしようとするとう、エラーが返されます。スレッドが、スレッドによって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う、エラーが返されます。

mutex 型が PTHREAD_MUTEX_RECURSIVE の場合、mutex はロックカウンツの概念を持ちます。スレッドが初めて正常に mutex を獲得すると、ロックカウンツは 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、ロックカウンツが 1 ずつ増えます。スレッドが mutex をロック解除するたびに、ロックカウンツが 1 ずつ減ります。ロックカウンツが 0 になると、別のスレッドがこの mutex を獲得できるようになります。スレッドが、スレッドによって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う、エラーが返されます。

mutex 型 PTHREAD_MUTEX_DEFAULT は PTHREAD_MUTEX_NORMAL と同じです。

pthread_mutex_lockの戻り値

pthread_mutex_lock() は、正常終了時に 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。

EAGAIN

説明: 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

説明: 現在のスレッドがすでにその mutex を獲得しています。

`mutex` が `PTHREAD_MUTEX_ROBUST_NP` の堅牢度属性で初期化された場合、`pthread_mutex_lock()` は次のいずれかの値を返す可能性があります。

EOWNERDEAD

説明: 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()` を呼び出さないでください。代わりに `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 `mutex` を保持している間に終了すると、次の所有者が `EOWNERDEAD` を持つロックを獲得します。

ENOTRECOVERABLE

説明: 獲得しようとしている `mutex` は、この `mutex` の前の所有者によって回復不能のままにされた状態を保護していました。`mutex` は獲得されませんでした。この回復可能な状態は、次のような場合に発生します。

- 以前、このロックが `EOWNERDEAD` によって取得された
- 所有者が状態をクリーンアップできなかった
- 所有者が `mutex` の状態を整合させないで `mutex` をロック解除した

ENOMEM

説明: 同時に保持される `mutex` の上限数を超過しています。

mutex のロック解除

`mutex` が指す `mutex` をロック解除するには、`pthread_mutex_unlock(3C)` を使用します。

pthread_mutex_unlock の構文

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

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

```
pthread_mutex_t mutex;
```

```
int ret;
```

```
ret = pthread_mutex_unlock(&mutex); /* release the mutex */
```

`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、*mutex* オブジェクトでブロックされているスレッドがある場合、*mutex* が使用可能な状態になると、スケジューリングポリシーに基づいて *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。

pthread_mutex_unlock の戻り値

`pthread_mutex_unlock()` は、正常終了時に 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。

EPERM

説明: 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

mutex が指す *mutex* のロックを試行し、その *mutex* がすでにロックされていればただちに返るには、`pthread_mutex_trylock(3C)` を使用します。

pthread_mutex_trylock の構文

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

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

```
pthread_mutex_t mutex;
int ret;
```

```
ret = pthread_mutex_trylock(&mutex); /* try to lock the mutex */
```

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。

pthread_mutex_trylockの戻り値

pthread_mutex_trylock() は、正常終了時に 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。

EBUSY

説明: *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

説明: *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 で初期化されています。さらに、pthread_mutexattr_setrobust_np() の引数 robustness が PTHREAD_MUTEX_ROBUST_NP の場合、この関数は失敗し、次のいずれかの値を返します。

EOWNERDEAD

説明: 97 ページの「pthread_mutex_lockの戻り値」の説明を参照してください。

ENOTRECOVERABLE

説明: 97 ページの「pthread_mutex_lockの戻り値」の説明を参照してください。

ENOMEM

説明: 同時に保持される mutex の上限数を超過しています。

指定した絶対時間までの mutex のロック

指定した時間まで mutex オブジェクトのロックを試行するには、pthread_mutex_timedlock(3C) 関数を使用します。

この関数は無期限にブロックされることがない点を除いて、pthread_mutex_lock() 関数と同じ動作をします。その mutex がすでにロックされている場合は、mutex が使用可能になるまで呼び出しスレッドはブロックされますが、それもタイムアウトに達するまでの間だけです。mutex が使用可能になる前にタイムアウトが発生すると、関数は終了します。

pthread_mutex_timedlock() の構文

```
int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,  
                             const struct timespec *restrict abs_timeout);
```

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

pthread_mutex_t mutex;
timestruct_t abs_timeout;
int ret;

ret = pthread_mutex_timedlock(&mutex, &abs_timeout);
```

pthread_mutex_timedlock() の戻り値

pthread_mutex_timedlock() 関数は、mutex を正常にロックすると 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ます。

EINVAL

説明: mutex が PTHREAD_PRIO_PROTECT の値を持つプロトコル属性で作成されており、呼び出しスレッドの優先順位がこの mutex の現在の優先順位上限を超えています。

説明: mutex 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。

説明: プロセスまたはスレッドがブロックされており、abs_timeout パラメータで指定されたナノ秒のフィールド値が 0 未満または 10 億以上です。

ETIMEDOUT

説明: 指定されたタイムアウトの期限が切れる前に mutex をロックできませんでした。

97 ページの「[pthread_mutex_lock の戻り値](#)」の説明を参照してください。

指定した時間間隔内の mutex のロック

指定した時間が経過するまで mutex オブジェクトのロックを試行するには、[pthread_mutex_reltimedlock_np\(3C\)](#) 関数を使用します。

タイムアウトの期限が切れるのは、rel_timeout で指定された時間間隔 (CLOCK_REALTIME クロックで測定される) が経過した場合、または呼び出し時に rel_timeout で指定された時間間隔が負である場合です。

pthread_mutex_reltimedlock_np() の構文

```
int pthread_mutex_reltimedlock_np(pthread_mutex_t *restrict mutex,
    const struct timespec *restrict rel_timeout);
```

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

pthread_mutex_t mutex;
timestruct_t rel_timeout;
int ret;

ret = pthread_mutex_reltimedlock_np(&mutex, &rel_timeout);
```

pthread_mutex_reltimedlock_np() の戻り値

pthread_mutex_reltimedlock_np() 関数は、mutex を正常にロックすると 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ます。

EINVAL

説明: mutex が PTHREAD_PRIO_PROTECT の値を持つプロトコル属性で作成されており、呼び出しスレッドの優先順位がこの mutex の現在の優先順位上限を超えています。

説明: mutex 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。

説明: プロセスまたはスレッドがブロックされており、abs_timeout パラメータで指定されたナノ秒のフィールド値が 0 未満または 10 億以上です。

ETIMEDOUT

説明: 指定されたタイムアウトの期限が切れる前に mutex をロックできませんでした。

[97 ページの「pthread_mutex_lock の戻り値」](#)の説明を参照してください。

mutex の削除

mp が指す mutex に関連付けられた状態を削除するには、pthread_mutex_destroy(3C) を使用します。

pthread_mutex_destroy の構文

```
int pthread_mutex_destroy(pthread_mutex_t *mp);

#include <pthread.h>

pthread_mutex_t mp;
int ret;

ret = pthread_mutex_destroy(&mp); /* mutex is destroyed */
```

`mutex` の記憶領域は解放されません。

`pthread_mutex_destroy` の戻り値

`pthread_mutex_destroy()` は、正常終了時に 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

説明: *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` ロックを使用したコードの抜粋を示します。

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;

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

例 4-1 では、2 つの関数がそれぞれ別の目的で相互排他ロックを使用しています。`increment_count()` 関数は、相互排他ロックによって共有変数の不可分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` が不可分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ーでは、`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不可分操作です。

ロック階層の使用例

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、対応する相互排他ロックを異なる順序で獲得しようとする、問題が発生します。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、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、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
<code>pthread_mutex_lock(&m1);</code>	<code>pthread_mutex_lock(&m2);</code>
<code>/* use resource 1 */</code>	<code>/* use resource 2 */</code>
<code>pthread_mutex_lock(&m2);</code>	<code>pthread_mutex_lock(&m1);</code>
<code>/* use resources 1 and 2 */</code>	<code>/* use resources 1 and 2 */</code>
<code>pthread_mutex_unlock(&m2);</code>	<code>pthread_mutex_unlock(&m1);</code>
<code>pthread_mutex_unlock(&m1);</code>	<code>pthread_mutex_unlock(&m2);</code>

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、この方法を「ロック階層」と呼びます。この方法では、`mutex` に論理的な番号を割り振って、`mutex` に順序を付けます。

自分が持つ `mutex` の番号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、この方法が使用できない場合もあります。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
<pre>pthread_mutex_lock(&m1); pthread_mutex_lock(&m2); /* 解放 */ pthread_mutex_unlock(&m2); pthread_mutex_unlock(&m1);</pre>	<pre>for (; ;) { pthread_mutex_lock(&m2); if(pthread_mutex_trylock(&m1)==0) /* 獲得成功 */ break; /* 獲得失敗 */ pthread_mutex_unlock(&m2); } /* ロックを獲得し、解放 */ pthread_mutex_unlock(&m1); pthread_mutex_unlock(&m2);</pre>

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、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 1` を慎重にロックしなければなりません。これは、スレッド 2 が `mutex 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。

デッドロックの発生を防ぐため、スレッド 2 は `pthread_mutex_trylock()` を呼び出し、`mutex` が使用可能な状態であれば獲得します。`mutex` が使用可能な状態でない場合、スレッド 2 はただちに終了し、エラーを返します。その時点で、スレッド 2 は `mutex 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 2` をロックでき、その後 `mutex 1` と `mutex 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 では、一度に 3 つのロックを獲得する方法について説明します。デッドロックを防ぐため、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` の位置からリストをたどって目的のノードを探します。`ListHead` が削除されることはありません。

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、デッドロックが発生することはありません。

目的のノードが見つかったら、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コードです。

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);
    return(NULL);
}
```

例 4-5 片方向リンクリストの入れ子のロック (続き)

```
}

```

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、そのノードと近傍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;

```

以下は、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うCコードです。

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_trylock(&me->link->lock) != 0) {
            /* failed to get lock */
            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);
}

```

スピンロックの使用

スピンロックは、主に共有メモリ型のマルチプロセッサ上での使用に適した低レベルの同期機構です。呼び出しスレッドが、すでに別のスレッドによって保持されているスピンロックを要求する場合、2番目のスレッドは、そのロックが使用可能になったかどうかをテストするためのループに入ります。スピンはプロセッササイクルを浪費するため、ロックを獲得したら、短時間だけ保持するようにすべきです。呼び出し元は、ほかのスレッドがロックを獲得できるようにするためのスリープ操作を呼び出す前に、スピンロックを解除するようにしてください。

スピンロックは `mutex` と条件変数を使用して実装することもできますが、スピンロックを実行するための標準化された方法は `pthread_spin_*` 関数です。短期間のロックであれば、`pthread_spin_*` 関数に必要なオーバーヘッドははるかに少なくなります。

どういうロックを実行する場合も、スレッドのブロックを設定している間に消費されるプロセッサリソースと、ブロックされている間にスレッドによって消費されるプロセッサリソースとの間にトレードオフが発生します。スピンロックでは、スレッドのブロックを設定したあと、単純なループを実行して、ロックが使用可能になるまで不可分なロック処理を繰り返すためのリソースがほとんど必要ありません。スレッドは、待機している間もプロセッサリソースを消費し続けます。

スピンロックに比べると、`mutex` はスレッドのブロックにより大量のプロセッサリソースを消費します。相互排他ロックが使用できない場合、スレッドはスケジューリングの状態を変更して、自身を待機スレッドの待ち行列に追加します。ロックが使用可能になると、スレッドがロックを獲得する前に、これらの手順を逆にたどる必要があります。スレッドは、ブロックされている間、プロセッサリソースを消費しません。

したがって、スピンロックと `mutex` は別の目的に使用すると有効な場合があります。非常に短期間のブロックでは、スピンロックの方が全体的なオーバーヘッドは少なくなることがあります。スレッドがより長期間ブロックされる場合は、`mutex` の方が全体的なオーバーヘッドは少なくなることがあります。

スピンロックの初期化

スピンロックを使用するために必要なリソースを割り当て、ロックをロック解除状態に初期化するには、`pthread_spin_init(3C)` 関数を使用します。

`pthread_spin_init()` の構文

```
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);

#include <pthread.h>

pthread_spinlock_t lock;
```

```
int pshared;
int ret;

/* initialize a spin lock */
ret = pthread_spin_init(&lock, pshared);
```

pshared 属性は、次のいずれかの値を持ちます。

PTHREAD_PROCESS_SHARED

説明: スピン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、スピンロックの操作を許可します。このロックが複数のプロセスによって共有されているメモリーに割り当てられている場合にも、ロックの操作を許可します。

PTHREAD_PROCESS_PRIVATE

説明: スピンロックを、そのスピン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が操作できるようにします。異なるプロセスのスレッドからスピン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。

pthread_spin_init() の戻り値

pthread_spin_init() 関数は、正常終了時に 0 を返します。それ以外の場合は、次のいずれかのエラーコードが返されます。

EAGAIN

説明: 別のスピンロックを初期化するために必要なシステムリソースが不足しています。

EBUSY

説明: スピンロックが別のスレッドで使用されている間 (たとえば、pthread_spin_lock() の呼び出しで使用されている間) にそのロックの初期化または削除の試行が検出されました。

EINVAL

説明: *lock* で指定された値が無効です。

スピンロックの獲得

スピンロックを獲得するには、[pthread_spin_lock\(3C\)](#) を使用します。呼び出しスレッドは、ロックが別のスレッドによって保持されていなければ、そのロックを獲得します。それ以外の場合、スレッドは、そのロックが使用可能になるまで `pthread_spin_lock()` 呼び出しから復帰しません。呼び出し時に、呼び出しスレッドがロックを保持している場合の結果は不定です。

pthread_spin_lock() の構文

```
int pthread_spin_lock(pthread_spinlock_t *lock);

#include <pthread.h>

pthread_spinlock_t lock;
int ret;

ret = pthread_spin_lock(&lock); /* lock the spinlock */
```

pthread_spin_lock() の戻り値

pthread_spin_lock() 関数は、正常終了時に 0 を返します。それ以外の場合は、次のいずれかのエラーコードが返されます。

EDEADLK

説明: 現在のスレッドがすでにそのスピンロックを獲得しています。

EINVAL

説明: *lock* で指定された値が、初期化されたスピンロックオブジェクトを表していません。

非ブロックスピンロックの獲得

スピンロックを獲得し、ロックが別のスレッドによって保持されている場合はただちに失敗するには、[pthread_spin_trylock\(3C\)](#) 関数を使用します。

pthread_spin_trylock() の構文

```
int pthread_spin_trylock(pthread_spinlock_t *lock);

#include <pthread.h>

pthread_spinlock_t lock;
int ret;

ret = pthread_spin_trylock(&lock); /* try to lock the spin lock */
```

pthread_spin_trylock() の戻り値

pthread_spin_trylock() 関数は、正常終了時に 0 を返します。それ以外の場合は、次のいずれかのエラーコードが返されます。

EBUSY

説明: スレッドが現在スピンロックを所有しています。

EINVAL

説明: *lock* で指定された値が、初期化されたスピンロックオブジェクトを表していません。

スピンロックの解除

獲得されたスピンロックを解放するには、[pthread_spin_unlock\(3C\)](#) 関数を使用します。

pthread_spin_unlock() の構文

```
int pthread_spin_unlock(pthread_spinlock_t *lock);

#include <pthread.h>

pthread_spinlock_t lock;
int ret;

ret = pthread_spin_unlock(&lock); /* spinlock is unlocked */
```

pthread_spin_unlock() の戻り値

[pthread_spin_unlock\(\)](#) 関数は、正常終了時に 0 を返します。それ以外の場合は、次のいずれかのエラーコードが返されます。

EPERM

説明: 呼び出しスレッドがロックを保持していません。

EINVAL

説明: *lock* で指定された値が、初期化されたスピンロックオブジェクトを表していません。

スピンロックの削除

スピンロックを削除し、そのロックによって使用されているリソースをすべて解放するには、[pthread_spin_destroy\(3C\)](#) 関数を使用します。

pthread_spin_destroy() の構文

```
int pthread_spin_destroy(pthread_spinlock_t *lock);
```

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

```
pthread_spinlock_t lock;  
int ret;
```

```
ret = pthread_spin_destroy(&lock); /* spinlock is destroyed */
```

削除したロックを、`pthread_spin_init()` を呼び出し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。スレッドがロックを保持しているときに `pthread_spin_destroy()` が呼び出された場合、またはスレッドの初期化されていないスピンロックに対してこの関数が呼び出された場合の結果は不定です。

`pthread_spin_destroy()` の戻り値

EBUSY

説明: スピンロックが別のスレッドで使用されている間 (たとえば、`pthread_spin_lock()` の呼び出しで使用されている間) にそのロックの初期化または削除の試行が検出されました。

EINVAL

説明: `lock` で指定された値が無効です。

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不可分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不可分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。別のスレッドによって条件が変更されると、そのスレッドは関連した条件変数にシグナルを送り、1つ以上の待機スレッドに次の処理を実行させることができます。

- 呼び起こす
- 相互排他ロックを再度獲得する
- 条件を再評価する

次の条件のプロセスの間でスレッドの同期を取るには、条件変数を使用します。

- スレッドが書き込み可能なメモリーに割り当てられている
- 連携する複数のプロセスがメモリーを共有している

スケジューリングポリシーは、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。デフォルトスケジューリングポリシー `SCHED_OTHER` は、スレッド

が呼び起こされる順序を指定していません。SCHED_FIFO および SCHED_RR リアルタイムスケジューリングポリシーの下では、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。表 4-4 に、条件変数の属性を操作する関数を示します。

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	113 ページの「 pthread_condattr_init の構文 」
条件変数の属性の削除	114 ページの「 pthread_condattr_destroy の構文 」
条件変数のスコープの設定	115 ページの「 pthread_condattr_setpshared の構文 」
条件変数のスコープの取得	115 ページの「 pthread_condattr_getpshared の構文 」
クロック選択条件変数の属性の取得	117 ページの「 pthread_condattr_getclock の構文 」
クロック選択条件変数の属性の設定	116 ページの「 pthread_condattr_setclock の構文 」

条件変数の属性の初期化

このオブジェクトに割り当てられた属性をデフォルト値に初期化するには、[pthread_condattr_init\(3C\)](#) を使用します。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。

pthread_condattr_init の構文

```
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);
```

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 です。 *pshared* がこの値の場合は、初期化された条件変数がプロセス内で使用可能であることを示します。

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得る値は、PTHREAD_PROCESS_PRIVATE と PTHREAD_PROCESS_SHARED です。 PTHREAD_PROCESS_PRIVATE はデフォルト値です。

条件変数の属性を再利用するには、まず、この属性を `pthread_condattr_destroy(3C)` で初期化し直す必要があります。`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。

`pthread_condattr_init` の戻り値

`pthread_condattr_init()` は、正常終了時に 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。

ENOMEM

説明: 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

説明: *cattr* で指定された値が無効です。

条件変数の属性の削除

記憶領域を削除し、属性オブジェクトを無効にするには、`pthread_condattr_destroy(3C)` を使用します。

`pthread_condattr_destroy` の構文

```
int pthread_condattr_destroy(pthread_condattr_t *cattr);
```

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

```
/* destroy an attribute */
ret
= pthread_condattr_destroy(&cattr);
```

`pthread_condattr_destroy` の戻り値

`pthread_condattr_destroy()` は、正常終了時に 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

説明: *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(3C)` は、条件変数のスコープをプロセス専用 (プロセス内) またはシステム共通 (プロセス間) に設定します。

`pthread_condattr_setpshared` の構文

```
int pthread_condattr_setpshared(pthread_condattr_t *cattr, int pshared);
```

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

```
/* all processes */
ret = pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_SHARED);

/* within a process */
ret = pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_PRIVATE);
```

共有メモリー内に *pshared* 属性を生成し、その条件変数を `PTHREAD_PROCESS_SHARED` に設定した場合、この条件変数は複数のプロセスのスレッド間で共有できます。

`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。`PTHREAD_PROCESS_PRIVATE` はデフォルト値です。`PTHREAD_PROCESS_PRIVATE` は、局所条件変数として動作します。`PTHREAD_PROCESS_SHARED` の動作は、大域条件変数に相当します。

`pthread_condattr_setpshared` の戻り値

`pthread_condattr_setpshared()` は、正常終了時に 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

説明: *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(3C)` 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。

`pthread_condattr_getpshared` の構文

```
int pthread_condattr_getpshared(const pthread_condattr_t *restrict cattr,
                                int *restrict pshared);
```

```
#include <pthread.h>

pthread_condattr_t attr;
int pshared;
int ret;

/* get pshared value of condition variable */
ret = pthread_condattr_getpshared(&attr, &pshared);
```

属性オブジェクトの値は、PTHREAD_PROCESS_SHARED か PTHREAD_PROCESS_PRIVATE になります。

pthread_condattr_getpshared の戻り値

pthread_condattr_getpshared () は、正常終了時に 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

説明: *attr* の値が無効です。

クロック選択条件変数の設定

attr が参照する初期化された属性オブジェクト内のクロック属性を設定するには、[pthread_condattr_setclock\(3C\)](#) 関数を使用します。

pthread_condattr_setclock() が、CPU 時間クロックを表す *clock_id* 引数を指定して呼び出された場合、その呼び出しは失敗します。クロック属性は、pthread_cond_timedwait() のタイムアウトサービスを測定するために使用されるクロックのクロック ID です。クロック属性のデフォルト値は、システムクロック CLOCK_REALTIME を表します。この時点で、クロック属性に対して取り得るほかの値は CLOCK_MONOTONIC だけです。

pthread_condattr_setclock の構文

```
int pthread_condattr_setclock(pthread_condattr_t attr,
                             clockid_t clock_id);
```

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

```
pthread_condattr_t attr
clockid_t clock_id
int ret
```

```
ret = pthread_condattr_setclock(&attr &clock_id
```

pthread_condattr_setclockの戻り値

pthread_condattr_setclock() は、正常終了時に 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

説明: *clock_id* で指定された値が既知のクロックを表していないか、または CPU 時間クロックです。

クロック選択条件変数の取得

attr が参照する属性オブジェクトからクロック属性の値を取得するには、[pthread_condattr_getclock\(3C\)](#) 関数を使用します。クロック属性は、pthread_cond_timedwait() のタイムアウトサービスを測定するために使用されるクロックのクロック ID です。

pthread_condattr_getclockの構文

```
int pthread_condattr_getclock(const pthread_condattr_t *restrict attr,
                             clockid_t *restrict clock_id);
```

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

```
pthread_condattr_t attr
clockid_t clock_id
int ret
```

```
ret = pthread_condattr_getclock(&attr &clock_id
```

pthread_condattr_getclockの戻り値

pthread_condattr_getclock() は、正常終了時に 0 を返し、*attr* の *clock* 属性の値を *clock_id* 引数が参照するオブジェクトに格納します。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

説明: *attr* 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 4-5 に、使用可能な関数を示します。

表 4-5 条件変数関数

操作	参照先
条件変数の初期化	118 ページの「 pthread_cond_init の構文 」
条件変数によるブロック	119 ページの「 pthread_cond_wait の構文 」
特定のスレッドのブロック	121 ページの「 pthread_cond_signal の構文 」
時刻指定のブロック	122 ページの「 pthread_cond_timedwait の構文 」
間隔指定のブロック	124 ページの「 pthread_cond_reltimedwait_np の構文 」
全スレッドのブロック解除	125 ページの「 pthread_cond_broadcast の構文 」
条件変数の削除	126 ページの「 pthread_cond_destroy の構文 」

条件変数の初期化

cv が指す条件変数をデフォルト値に初期化する場合や、すでに `pthread_condattr_init()` で設定されている条件変数を指定する場合は、[pthread_cond_init\(3C\)](#) を使用します。

pthread_cond_init の構文

```
int pthread_cond_init(pthread_cond_t *restrict cv,
                      const pthread_condattr_t *restrict attr);
```

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

```
pthread_cond_t cv;
pthread_condattr_t attr;
int ret;
```

```
/* initialize a condition variable to its default value */
ret = pthread_cond_init(&cv, NULL);
```

```
/* initialize a condition variable */
ret = pthread_cond_init(&cv, &attr);
```

attr を NULL に設定するのは、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。

静的に定義された条件変数をデフォルト属性に初期化するには、マクロ `PTHREAD_COND_INITIALIZER` を使用します。`PTHREAD_COND_INITIALIZER` マクロは、`NULL` 属性を指定して `pthread_cond_init()` を動的に割り当てた場合と同じ効果を適用します。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。

pthread_cond_init の戻り値

`pthread_cond_init()` は、正常終了時に 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

説明: *cattr* で指定された値が無効です。

EBUSY

説明: その条件変数は現在使用されています。

EAGAIN

説明: 必要なリソースが利用できません。

ENOMEM

説明: 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

mp が指す相互排他ロックを不可分操作により解放し、*cv* が指す条件変数で呼び出しスレッドをブロックするには、`pthread_cond_wait(3C)` を使用します。

pthread_cond_wait の構文

```
int pthread_cond_wait(pthread_cond_t *restrict cv, pthread_mutex_t *restrict mutex);
```

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

```
pthread_cond_t cv;
pthread_mutex_t mp;
int ret;
```

```
/* wait on condition variable */
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()` を使用します。シグナルの割り込みによっても呼び起こすことができます。

`pthread_cond_wait()` が終了しても、条件変数に関連付けられた条件の値が変化すると判断することはできません。条件は再評価する必要があります。

`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、不可分操作により、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不可分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。この変更によって、その条件変数で待ち状態となっていた少なくとも1つのスレッドがブロック解除され、もう一度 `mutex` を獲得しようとします。

`pthread_cond_wait()` の位置から実行を続行する前に、待機の条件を再テストする必要があります。呼び起こされたスレッドが `mutex` を再度獲得して `pthread_cond_wait()` から戻る前に、条件が変更される可能性があります。また、待機スレッドが誤って呼び起こされる可能性があります。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();
```

スケジューリングポリシーは、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。デフォルトスケジューリングポリシー `SCHED_OTHER` は、スレッドが呼び起こされる順序を指定していません。`SCHED_FIFO` および `SCHED_RR` リアルタイムスケジューリングポリシーの下では、スレッドは優先順位に従って呼び起こされます。

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。

pthread_cond_wait の戻り値

pthread_cond_wait() は、正常終了時に 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

説明: *cv* または *mp* で指定された値が無効です。

1つのスレッドのブロック解除

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するには、[pthread_cond_signal\(3C\)](#) を使用します。

pthread_cond_signal の構文

```
int pthread_cond_signal(pthread_cond_t *cv);
```

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

```
pthread_cond_t cv;  
int ret;
```

```
/* one condition variable is signaled */  
ret = pthread_cond_signal(&cv);
```

関連付けられた条件は、シグナルを送ろうとしている条件変数と同じ mutex ロックを獲得した状態で変更します。そうしないと、これらの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が変更される可能性があり、その場合 pthread_cond_wait は永久に待ち続けることになります。

スケジューリングポリシーは、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。デフォルトスケジューリングポリシー SCHED_OTHER は、スレッドが呼び起こされる順序を指定していません。SCHED_FIFO および SCHED_RR リアルタイムスケジューリングポリシーの下では、スレッドは優先順位に従って呼び起こされます。

条件変数でブロックされるスレッドがない場合、pthread_cond_signal() を呼び出しても何も起こりません。

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;
```

例4-8 pthread_cond_wait() と pthread_cond_signal() の使用例 (続き)

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

pthread_cond_signal の戻り値

pthread_cond_signal() は、正常終了時に 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

説明: *cv* が指すアドレスが正しくありません。

例4-8 は、pthread_cond_wait() と pthread_cond_signal() の使用例です。

時刻指定のブロック

pthread_cond_timedwait(3C) は、*abstime* で指定され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 の構文

```
int pthread_cond_timedwait(pthread_cond_t *restrict cv,
                           pthread_mutex_t *restrict mp,
                           const struct timespec *restrict abstime);
```

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

```
pthread_cond_t cv;
```

```
pthread_mutex_t mp;
timestruct_t abstime;
int ret;

/* wait on condition variable */
ret = pthread_cond_timedwait(&cv, &
mp, &abstime);
```

`pthread_cond_timedwait()` は、たとえエラーを返す場合でも、常に呼び出しスレッドが `mutex` ロックし、所有した状態で終了します。()

`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。

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);
clock_gettime(CLOCK_REALTIME, &to);
to.tv_sec += TIMEOUT;
while (cond == FALSE) {
    err = pthread_cond_timedwait(&c, &m, &to);
    if (err == ETIMEDOUT) {
        /* timeout, do something */
        break;
    }
}
pthread_mutex_unlock(&m);
```

`pthread_cond_timedwait` の戻り値

`pthread_cond_timedwait()` は、正常終了時に 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

説明: `cv`、`mp`、または `abstime` が不当なアドレスを指しています。

EINVAL

説明: 同じ条件変数への並行した `pthread_cond_timedwait()` 操作に対して異なる `mutex` が指定されました。

ETIMEDOUT

説明: *abstime* で指定された時刻を過ぎています。

EPERM

説明: 呼び出し時に、*mutex* が現在のスレッドによって所有されていませんでした。

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く、効率的に条件を再評価できます。例 4-9 を参照してください。

間隔指定のブロック

`pthread_cond_reltimedwait_np(3C)` の使用方法は、1 点を除いて、`pthread_cond_timedwait()` の使用方法と同じです。 `pthread_cond_reltimedwait_np()` の場合は、最後の引数として、未来の絶対日時ではなく相対時間間隔を指定します。

`pthread_cond_reltimedwait_np` の構文

```
int pthread_cond_reltimedwait_np(pthread_cond_t *cv,  
                                pthread_mutex_t *mp,  
                                const struct timespec *reltime);
```

```
#include <pthread.h>  
#include <time.h>  
pthread_cond_t cv;  
pthread_mutex_t mp;  
timestruct_t reltime;  
int ret;
```

```
/* wait on condition variable */  
ret = pthread_cond_reltimedwait_np(&cv, &mp, &reltime);
```

`pthread_cond_reltimedwait_np()` は、たとえエラーを返す場合でも、常に呼び出しスレッドが *mutex* ロックし、所有した状態で終了します。

`()pthread_cond_reltimedwait_np()` 関数は、条件のシグナルを受け取るか、最後の引数に指定されている時間間隔が経過するまで、ブロックします。

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

pthread_cond_reltimedwait_npの戻り値

pthread_cond_reltimedwait_np() は、正常終了時に 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

説明: *reltime* で指定された値が無効です。

ETIMEDOUT

説明: *reltime* に指定されている時間間隔が経過しました。

全スレッドのブロック解除

pthread_cond_wait() で指定された *cv* 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するには、[pthread_cond_broadcast\(3C\)](#) を使用します。

pthread_cond_broadcastの構文

```
int pthread_cond_broadcast(pthread_cond_t *cv);
```

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

```
pthread_cond_t cv;
int ret;
```

```
/* all condition variables are signaled */
ret = pthread_cond_broadcast(&cv);
```

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。

pthread_cond_broadcast() は、条件変数でブロックされていたすべてのスレッドにもう一度相互排他ロックを争奪させるので、慎重に使用してください。()たとえば、pthread_cond_broadcast() を使用して、可変量のリソースを、解放時にスレッド間で争奪させることができます。[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);
```

例 4-10 条件変数に対するブロードキャスト (続き)

```
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);
}
```

`add_resources()` の部分に注目してください。相互排他ロックの範囲内では、`resources` の更新と `pthread_cond_broadcast()` の呼び出しのどちらを先に行なってもかまいません。

関連付けられた条件は、シグナルを送ろうとしている条件変数と同じ `mutex` ロックを獲得した状態で変更します。そうしないと、これらの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が変更される可能性があり、その場合 `pthread_cond_wait` は永久に待ち続けることになります。

`pthread_cond_broadcast` の戻り値

`pthread_cond_broadcast()` は、正常終了時に 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

説明: `cv` が指すアドレスが正しくありません。

条件変数の削除

`cv` が指す条件変数に関連付けられた状態を削除するには、`pthread_cond_destroy(3C)` を使用します。

`pthread_cond_destroy` の構文

```
int pthread_cond_destroy(pthread_cond_t *cv);
```

```
#include <pthread.h>

pthread_cond_t cv;
int ret;

/* Condition variable is destroyed */
ret = pthread_cond_destroy(&cv);
```

条件変数の記憶領域は解放されません。

pthread_cond_destroy の戻り値

pthread_cond_destroy() は、正常終了時に 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

説明: *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。

「呼び起こし忘れ」は次のすべての条件が揃った場合に発生します。

- スレッドが pthread_cond_signal() または pthread_cond_broadcast() を呼び出す
 - 条件をテストしてから pthread_cond_wait() を呼び出すまでの間に、ほかのスレッドが生成される
 - 待機しているスレッドが存在しない
- シグナルは無効になり、失われます

この問題は、条件に関連付けられた相互排他ロックを保持しないでテスト対象の条件が変更された場合に限り発生します。関連付けられた mutex を保持した状態でテスト対象の条件を変更した場合には、pthread_cond_signal() と pthread_cond_broadcast() を呼び出すときにそれらがその mutex を保持しているかどうかは影響しません。

「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、生産者がバッファに何か書き込むまで、このバッファから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 には、こうした待ち行列を 2 つ使用しています。1 つは *less* で、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す。もう 1 つは *more* で、情報が格納されたバッファー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 では、最初に、生産者のスレッドが *mutex* を獲得し、バッファデータ構造 (*buffer*) を保護します。この生産者スレッドは、次に、生成された項目のための空きがあることを確認します。空きがない場合は、*pthread_cond_wait()* を呼び出します。*pthread_cond_wait()* は、「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。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。

これにより、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する必要があります。空きがあれば、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、ほか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++;

    /* now: either b->occupied < BSIZE and b->nextin is the index
       of the next empty slot in the buffer, or
       b->occupied == BSIZE and b->nextin is the index of the
       next (occupied) slot that will be emptied by a consumer
       (such as b->nextin == b->nextout) */

    pthread_cond_signal(&b->more);

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

`assert()` 文の用法に注意してください。`NDEBUG` を定義してコードをコンパイルした場合を除き、`assert()` はその引数が真 (0 以外) として評価された場合は何も行いません。その引数が偽 (0) として評価された場合、プログラムは終了します。こうしたアサーションはマルチスレッドプログラムで特に役立ちます。`assert()` は、アサーションが失敗すると実行時に発生した問題をただちに指摘します。`assert()` の効果はほかにもあり、役立つコメントを提供します。

`/* 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。`

`assert` の部分もコメント部分も、不変式になっています。これらの不変式は、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が、次のような例外がありま

す。この例外は、あるスレッドが不変式中のプログラム変数を変更している瞬間に起こります。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実行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。

このため、この不変式は常に真となる特性を持っていますが、次のような例外があります。この例外は、生産者または消費者がバッファの状態を変更しているときに起こります。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 に、消費者のコードを示します。論理フロー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--;

    /* now: either b->occupied > 0 and b->nextout is the index
       of the next occupied slot in the buffer, or
       b->occupied == 0 and b->nextout is the index of the next
       (empty) slot that will be filled by a producer (such as
       b->nextout == b->nextin) */

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

    return(item);
}
```

セマフォによる同期

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。

この鉄道線路上の運行の同期を取るのがセマフォ (腕木信号機) 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、ほか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、ほか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。

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、値から 1 を引いて、セマフォの値を変更します。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。これらの操作は不可分的に行う必要があります。これらの操作を分割して、次の操作までの間に、セマフォに対してその他のアクションが実行されるようなことがあってはなりません。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いければなりません。

ダイクストラがオランダ人だったため、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。

`sem_wait(3RT)` と `sem_post(3RT)` 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。`sem_trywait(3RT)` 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、大きく 2 つに分類できます。1 つは 2 進型セマフォで、0 および 1 以外の値はとりません。もう 1 つは計数型セマフォで、0 以上の任意の値をとります。2 進型セマフォは、論理的に `mutex` と同じです。

必ずしも必須要件ではありませんが、`mutex` のロックは、ロックを保持しているスレッドがそのロックを解放するべきです。ただし、セマフォには「スレッドがセマフォを保持している」という概念がないため、任意のスレッドがV操作(`sem_post(3RT)`)を実行できます。

計数型セマフォは、`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、条件変数よりも計数型セマフォを使用した方がコードが簡素化されます。例 4-14、例 4-15、例 4-16 を参照してください。

しかし、`mutex` を条件変数といっしょに使用した場合は、暗黙のくくりが存在します。このくくりは、プログラム内で保護されている部分を明確に区別します。この動作はセマフォに限定されたものではなく、「並行プログラミングにおける *go to*」と呼ばれることがあります。セマフォは強力ですが、構造化されないあいまいな方法で使用してしまいがちです。

名前付きセマフォと名前なしセマフォ

POSIX セマフォは、名前付きの場合と名前なしの場合があります。名前なしセマフォは、プロセスメモリー内で割り当てられ、初期化されます。名前なしセマフォは、割り当てと初期化の状態によっては、複数のプロセスで使用できます。名前なしセマフォは、`fork()` から継承された専用セマフォであるか、またはその割り当てとマッピングが行われた通常ファイルのアクセス保護によって保護されたセマフォです。

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。名前付きセマフォは、複数のプロセスによる共有が可能です。名前付きセマフォは、固有のユーザー ID、グループ ID、および保護モードを持ちます。

`sem_open`、`sem_getvalue`、`sem_close`、`sem_unlink` 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。

名前付きセマフォの詳細は、`sem_open`、`sem_getvalue`、`sem_close`、`sem_unlink` のマニュアルページを参照してください。

計数型セマフォの概要

セマフォの概念は、0 以上の整数カウントです。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。スレッドは、リソースが追加されると不可分操作的にカウントを 1 加算し、リソースが削除されると不可分操作的に 1 減算します。

これ以上リソースが存在しなくなると、セマフォカウン트는0になります。この場合、スレッドがカウンートを1減らそうとすると、カウン트가0より大きくなるまでブロックされます。

表 4-6 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	133 ページの「sem_init の構文」
セマフォの加算	135 ページの「sem_post の構文」
セマフォの値によるブロック	136 ページの「sem_wait の構文」
セマフォの減算	136 ページの「sem_trywait の構文」
セマフォの削除	137 ページの「sem_destroy の構文」

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に利用できます。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。

スケジューリングポリシーは、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。デフォルトスケジューリングポリシー `SCHED_OTHER` は、スレッドが呼び起こされる順序を指定していません。`SCHED_FIFO` および `SCHED_RR` リアルタイムスケジューリングポリシーの下では、スレッドは優先順位に従って呼び起こされます。

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

`sem` が指す名前なしセマフォ変数を `value` の値に初期化するには、`sem_init(3RT)` を使用します。

`sem_init` の構文

```
int sem_init(sem_t *sem, int pshared, unsigned int value);

#include <semaphore.h>

sem_t sem;
int pshared;
int ret;
```

```
int value;

/* initialize a private semaphore */
pshared = 0;
value = 1;
ret = sem_init(&sem, pshared, value);
```

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。

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。

一度初期化したセマフォは、ほか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。

プロセス間スコープでセマフォを初期化する

pshared が 0 の場合、セマフォを使用できるのは、このプロセス内のスレッドだけです。

```
#include <semaphore.h>

sem_t sem;
int ret;
int count = 4;

/* to be used within this process only */
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 が 0 以外の場合、セマフォは複数のプロセスで共有可能です。

```
#include <semaphore.h>

sem_t sem;
int ret;
int count = 4;

/* to be shared among processes */
ret = sem_init(&sem, 1, count);
```

sem_init の戻り値

`sem_init()` は、正常終了時に 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

説明: value の値が SEM_VALUE_MAX を超えています。

ENOSPC

説明: 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

説明: 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。

セマフォの加算

sem が指すセマフォを不可分的に加算するには、`sem_post(3RT)` を使用します。

`sem_post` の構文

```
int sem_post(sem_t *sem);

#include <semaphore.h>

sem_t sem;
int ret;

ret = sem_post(&sem); /* semaphore is posted */
```

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

`sem_post` の戻り値

`sem_post()` は、正常終了時に 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

説明: *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

sem が指すセマフォカウン트가 0 より大きくなるまで呼び出しスレッドをブロックし、0 より大きくなったら不可分的にカウンートを減らすには、`sem_wait(3RT)` を使用します。

sem_wait の構文

```
int sem_wait(sem_t *sem);

#include <semaphore.h>

sem_t sem;
int ret;

ret = sem_wait(&sem); /* wait for semaphore */
```

sem_wait の戻り値

sem_wait() は、正常終了時に 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

説明: *sem* が指すアドレスが正しくありません。

EINTR

説明: 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

sem が指すセマフォ内のカウントが 0 より大きいときにこの値を不可分的に減らすには、[sem_trywait\(3RT\)](#) を使用します。

sem_trywait の構文

```
int sem_trywait(sem_t *sem);

#include <semaphore.h>

sem_t sem;
int ret;

ret = sem_trywait(&sem); /* try to wait for semaphore*/
```

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、失敗した場合にはただちに終了します。()

sem_trywait の戻り値

sem_trywait() は、正常終了時に 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

説明: *sem* が指すアドレスが正しくありません。

EINTR

説明: この関数にシグナルが割り込みを行いました。

EAGAIN

説明: 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

sem が指す名前なしセマフォに関連付けられた状態を削除するには、[sem_destroy\(3RT\)](#) を使用します。

sem_destroy の構文

```
int sem_destroy(sem_t *sem);
```

```
#include <semaphore.h>
```

```
sem_t sem;  
int ret;
```

```
ret = sem_destroy(&sem); /* the semaphore is destroyed */
```

セマフォの記憶領域は解放されません。

sem_destroy の戻り値

sem_destroy() は、正常終了時に 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

説明: *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—セマフォを使った例

例 4-14 のデータ構造は、例 4-11 の条件変数の例で使用されている構造とよく似ています。2つのセマフォは、いっぱいになったバッファー数と空のバッファー数をそれぞれ表します。これらのセマフォは、バッファーが空になるまで生産者を待たせ、バッファーがいっぱいになるまで消費者を待たせます。

例 4-14 「生産者/消費者」問題—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;
```

もう一組の2進型セマフォを `mutex` として使用しています。このセマフォは、複数の生産者が複数の空バッファースロットを使用するときと、複数の消費者が複数のいっぱいになったバッファースロットを使用するときのバッファー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、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。

データベースアクセスは、読み取り/書き込みロックと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ときは、書き込み操作は排他的書き込みロックを獲得する必要があります。

デフォルトの読み取り/書き込みロック属性を変更するときに、属性オブジェクトを宣言および初期化することができます。多くの場合、読み取り/書き込みロック属性は、アプリケーションの開始位置の1か所に設定してあります。アプリケーションの開始位置に設定すると、この属性の検出および修正が容易になります。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 4-7 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	140 ページの「 pthread_rwlockattr_init の構文 」
読み取り/書き込みロック属性の削除	140 ページの「 pthread_rwlockattr_destroy の構文 」
読み取り/書き込みロック属性の設定	141 ページの「 pthread_rwlockattr_setpshared の構文 」
読み取り/書き込みロック属性の取得	142 ページの「 pthread_rwlockattr_getpshared の構文 」

読み取り/書き込みロック属性の初期化

[pthread_rwlockattr_init\(3C\)](#) は、読み取り/書き込みロック属性オブジェクト *attr* を、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。

pthread_rwlockattr_init の構文

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

```
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```

[pthread_rwlockattr_init](#) を呼び出すときに、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た場合、結果は保証されません。読み取り/書き込みロック属性オブジェクトによ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。

pthread_rwlockattr_init の戻り値

[pthread_rwlockattr_init\(\)](#) は、正常終了時に 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ます。

ENOMEM

説明: 読み取り/書き込み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\(3C\)](#) は、読み取り/書き込みロック属性を削除します。

pthread_rwlockattr_destroy の構文

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

```
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

削除したオブジェクトを、`pthread_rwlockattr_init()` を呼び出し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()` により、`attr` が参照するオブジェクトに不正な値が設定される場合もあります。

pthread_rwlockattr_destroy の戻り値

`pthread_rwlockattr_destroy()` は、正常終了時に 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ます。

EINVAL

説明: `attr` で指定された値が無効です。

読み取り/書き込みロック属性の設定

`pthread_rwlockattr_setpshared(3C)` は、プロセス共通の読み取り/書き込みロック属性を設定します。

pthread_rwlockattr_setpshared の構文

```
#include <pthread.h>
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);
```

`pshared` ロック属性は、次のいずれかの値を持ちます。

PTHREAD_PROCESS_SHARED

説明: 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。読み取り/書き込みロックが複数のプロセスによって共有されているメモリーに割り当てられている場合にも、ロックの操作を許可します。

PTHREAD_PROCESS_PRIVATE

説明: 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。

pthread_rwlockattr_setpshared の戻り値

`pthread_rwlockattr_setpshared()` は、正常終了時に 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ます。

EINVAL

説明: `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(3C)` 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` の構文

```
#include <pthread.h>
int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *restrict attr,
                                   int *restrict pshared);
```

`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。

`pthread_rwlockattr_getpshared` の戻り値

`pthread_rwlockattr_getpshared()` は、正常終了時に 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ます。

`EINVAL`

説明: `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ら、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 4-8 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/書き込みロックの初期化	143 ページの「 pthread_rwlock_init の構文 」
読み取り/書き込みロックの読み取り ロック	144 ページの「 pthread_rwlock_rdlock の構文 」
非ブロック読み取り/書き込みロックの読 み取りロック	146 ページの「 pthread_rwlock_tryrdlock の構文 」
読み取り/書き込みロックの書き込み ロック	147 ページの「 pthread_rwlock_wrlock の構文 」
非ブロック読み取り/書き込みロックの書 き込みロック	147 ページの「 pthread_rwlock_trywrlock の構文 」

表 4-8 読み取り/書き込みロック属性のルーチン (続き)

操作	参照先
読み取り/書き込みロックの解除	149 ページの「 pthread_rwlock_unlock の構文 」
読み取り/書き込みロックの削除	150 ページの「 pthread_rwlock_destroy の構文 」

読み取り/書き込みロックの初期化

attr が参照する属性を使用して *rwlock* が参照する読み取り/書き込みロックを初期化するには、[pthread_rwlock_init\(3C\)](#) を使用します。

pthread_rwlock_init の構文

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

```
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                        const pthread_rwlockattr_t *restrict attr);
```

```
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

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。

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` というマクロで、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。

pthread_rwlock_init の戻り値

`pthread_rwlock_init()` は、正常終了時に 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

説明: *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の獲得

`pthread_rwlock_rdlock(3C)` 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。

`pthread_rwlock_rdlock` の構文

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

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock );
```

呼び出しスレッドは、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、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ブロックされます。つまり、呼び出しスレッドは、スレッドがロックを獲得するまで、`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することが許されています。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。

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、`pthread_rwlock_unlock()` を n 回呼び出して、同数の読み取りロック解除を行わ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。

スレッドシグナルハンドラは、読み取り/書き込みロックを待っているスレッドに送信されたシグナルを処理します。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。

`pthread_rwlock_rdlock` の戻り値

`pthread_rwlock_rdlock()` は、正常終了時に 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ます。

EINVAL

説明: `attr` または `rwlock` が示す値は無効です。

指定した絶対時間までの読み取り/書き込み ロックの読み取りロックの獲得

`pthread_rwlock_timedrdlock(3C)` 関数は、`pthread_rwlock_rdlock()` 関数と同様に、`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。

`pthread_rwlock_timedrdlock` の構文

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

```
int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
                               const struct timespec *restrict abs_timeout);
```

ほかのスレッドによるロック解除を待たなければロックを獲得できない場合、この待機は、指定されたタイムアウトの期限が切れると終了します。タイムアウトの期限が切れるのは、`abs_timeout` で指定された絶対時間 (`CLOCK_REALTIME` クロックで測定される) が経過した場合 (つまり、そのクロックの値が `abs_timeout` に等しくなるか、超えた場合)、または呼び出し時に `abs_timeout` で指定された絶対時間がすでに過ぎている場合です。

タイムアウトの解像度は、`CLOCK_REALTIME` クロックの解像度です。`timespec` データ型は、`<time.h>` ヘッダーで定義されています。ロックをただちに獲得できる場合は、この関数がタイムアウトで失敗することはありません。ロックをただちに獲得できた場合は、タイムアウトパラメータの妥当性を検査する必要はありません。

`pthread_rwlock_timedrdlock()` の呼び出しによって読み取り/書き込みロックでブロックされているスレッドに、シグナルハンドラを実行させるシグナルが送信された場合、このスレッドはシグナルハンドラから復帰すると、中断がなかったかのよう to ロックの待機を再開します。

呼び出しスレッドは、呼び出し時に `rwlock` に書き込みロックを保持しているとデッドロックに陥る可能性があります。

`pthread_rwlock_reltimedrdlock_np()` 関数は、タイムアウトが相対時間間隔として指定される点を除いて、`pthread_rwlock_timedrdlock()` 関数と同じ動作をします。

`pthread_rwlock_timedrdlock` の戻り値

正常に終了して、`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書き込みロックが獲得された場合、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ます。

ETIMEDOUT

説明: 指定されたタイムアウトの期限が切れる前にロックを獲得できませんでした。

EAGAIN

説明: ロックの読み取りロックの最大数を超えるため、読み取りロックを獲得できません。

EDEADLK

説明: 呼び出しスレッドは、*rwlock* をすでに保持しています。

EINVAL

説明: *rwlock* で指定された値が初期化された読み取り/書き込みロックオブジェクトを表していないか、タイムアウトのナノ秒の値が0未満または10億以上です。

読み取り/書き込みロックの非ブロック読み取りロックの獲得

`pthread_rwlock_tryrdlock(3C)` 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。

`pthread_rwlock_tryrdlock` の構文

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

```
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

`pthread_rwlock_tryrdlock` の戻り値

`pthread_rwlock_tryrdlock()` 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対する読み取りロックが獲得された場合、0 を返します。ロックが獲得されなかった場合は、エラーを示すエラー番号が返されます。

EBUSY

説明: 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の獲得

`pthread_rwlock_wrlock(3C)` 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。

pthread_rwlock_wrlockの構文

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

```
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock );
```

呼び出しスレッドは、ほかの読み取りスレッドまたは書き込みスレッドが *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、書き込みロックを獲得します。それ以外の場合、呼び出しスレッドはブロックされます。つまり、呼び出しスレッドは、スレッドがロックを獲得するまで、`pthread_rwlock_wrlock()` 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。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。

`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。

スレッドシグナルハンドラは、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送信されたシグナルを処理します。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。

pthread_rwlock_wrlockの戻り値

`pthread_rwlock_wrlock()` 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対する書き込みロックが獲得された場合、0 を返します。ロックが獲得されなかった場合は、エラーを示すエラー番号が返されます。

読み取り/書き込みロックの非ブロック書き込みロックの獲得

`pthread_rwlock_trywrlock(3C)` 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。

pthread_rwlock_trywrlockの構文

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

```
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

`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。

pthread_rwlock_trywrlockの戻り値

`pthread_rwlock_trywrlock()` は、正常に終了して、`rwlock` が参照する読み取り/書き込みロックに対する書き込みロックが獲得された場合、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ます。

EBUSY

説明: 読み取りまたは書き込みのための読み取り/書き込みロックがすでにロックされているので、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。

指定した絶対時間までの読み取り/書き込みロックの書き込みロックの獲得

`pthread_rwlock_timedwrlock(3C)` 関数は、`pthread_rwlock_wrlock()` 関数と同様に、`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が、指定された絶対時間までしかロックを適用しようとしません。

pthread_rwlock_timedwrlockの構文

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

```
int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock,
                                const struct timespec *restrict abs_timeout);
```

呼び出しスレッドは、ほかの読み取りスレッドまたは書き込みスレッドが `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、書き込みロックを獲得します。ほかのスレッドによるロック解除を待たなければロックを獲得できない場合、この待機は、指定されたタイムアウトの期限が切れると終了します。タイムアウトの期限が切れるのは、`abs_timeout` で指定された絶対時間 (CLOCK_REALTIME クロックで測定される) が経過した場合 (つまり、そのクロックの値が `abs_timeout` に等しくなるか、超えた場合)、または呼び出し時に `abs_timeout` で指定された絶対時間がすでに過ぎている場合です。 `pthread_rwlock_reltimedwrlock_np()` 関数は、タイムアウトが相対時間間隔として指定される点を除いて、`pthread_rwlock_timedwrlock()` 関数と同じ動作をします。

pthread_rwlock_timedwrlockの戻り値

正常に終了して、`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書き込みロックが獲得された場合、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ます。

ETIMEDOUT

説明: 指定されたタイムアウトの期限が切れる前にロックを獲得できませんでした。

EDEADLK

説明: 呼び出しスレッドは、*rwlock* をすでに保持しています。

EINVAL

説明: *rwlock* で指定された値が初期化された読み取り/書き込みロックオブジェクトを表していないか、タイムアウトのナノ秒の値が0未満または10億以上です。

読み取り/書き込みロックの解除

`pthread_rwlock_unlock(3C)` 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。

`pthread_rwlock_unlock` の構文

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

```
int pthread_rwlock_unlock (pthread_rwlock_t *rwlock);
```

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。

`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、このロックオブジェクトにほかの読み取りロックが保持されている場合、この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、この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、スケジューリングポリシーを使用して、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。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、スケジューリングポリシーを使用して、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。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。

`pthread_rwlock_unlock` の戻り値

`pthread_rwlock_unlock()` は、正常終了時に 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ます。

読み取り/書き込みロックの削除

`pthread_rwlock_destroy(3C)` は、*rwlock* が参照する読み取り/書き込みロックオブジェクトを削除し、このロックが使用していたすべてのリソースを解放します。

`pthread_rwlock_destroy` の構文

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

```
int pthread_rwlock_destroy(pthread_rwlock_t **rwlock);
```

削除したロックを、`pthread_rwlock_init()` を呼び出し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()` により、*rwlock* が参照するオブジェクトに不正な値が設定される場合もあります。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。

`pthread_rwlock_destroy` の戻り値

`pthread_rwlock_destroy()` は、正常終了時に 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ます。

EINVAL

説明: *attr* または *rwlock* が示す値は無効です。

バリアー同期の使用

タスク全体を続行させるために複数のタスクの完了を待つ必要がある場合は、バリアー同期を使用できます。POSIX スレッドでは、バリアー関数とともに、バリアーと呼ばれる同期オブジェクトを指定します。これらの関数は、バリアー上で同期させるスレッドの数を指定してバリアーを作成し、各スレッドを、タスクの実行後、すべてのスレッドがバリアーに到達するまでバリアーで待機するように設定します。最後のスレッドがバリアーに到達すると、すべてのスレッドが実行を再開します。

バリアー同期の詳細については、[257 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。

同期バリアーの初期化

バリアーにリソースを割り当て、その属性を初期化するには、`pthread_barrier_init(3C)` を使用します。

`pthread_barrier_init()` の構文

```
int pthread_barrier_init(pthread_barrier_t *barrier,
                        const pthread_barrierattr_t *restrict attr,
                        unsigned count);
```

```
#include <pthread.h>
pthread_barrier_t barrier;
pthread_barrierattr_t attr;
unsigned count;
int ret;
ret = pthread_barrier_init(&barrier, &attr, count);
```

`pthread_barrier_init()` 関数は、`barrier` が参照するバリアーを使用するために必要なすべてのリソースを割り当て、`attr` が参照する属性でバリアーを初期化します。`attr` が NULL の場合は、デフォルトのバリアー属性が使用されます。この場合の結果は、デフォルトのバリアー属性オブジェクトのアドレスを渡す場合と同じです。`count` 引数には、`pthread_barrier_wait()` を呼び出す必要のあるスレッドの数を指定します。それまでは、どのスレッドも呼び出しから正常に復帰できません。`count` で指定する値は 1 以上にする必要があります。

`pthread_barrier_init()` の戻り値

`pthread_barrier_init()` は、正常終了時に 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

説明: *count* で指定された値が 0 に等しいか、または *attr* で指定された値が無効です。

EAGAIN

説明: 別のバリアーを初期化するために必要なシステムリソースが不足しています。

ENOMEM

説明: メモリー不足のためバリアーを初期化できません。

EBUSY

説明: バリアーが別のスレッドによって使用されている間 (たとえば、`pthread_barrier_wait()` の呼び出しで使用されている間) にそのバリアーの削除の試行が検出されました。

バリアーで同期をとるためのスレッドの待機

指定したバリアーでスレッドの同期をとるには、`pthread_barrier_wait(3C)` を使用します。呼び出しスレッドは、必要な数のスレッドがこのバリアーを指定して `pthread_barrier_wait()` を呼び出すまでブロックされます。スレッドの数は、`pthread_barrier_init()` 関数で指定されます。

必要な数のスレッドがこのバリアーを指定して `pthread_barrier_wait()` を呼び出すと、1 つの不定のスレッドに定数 `PTHREAD_BARRIER_SERIAL_THREAD` が返され、残りの各スレッドには 0 が返されます。バリアーは次に、このバリアーを参照していた最新の `pthread_barrier_init()` 関数の結果として保持していた状態にリセットされません。

`pthread_barrier_wait()` の構文

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

```
#include <pthread.h>
pthread_barrier_t barrier;
int ret;
ret = pthread_barrier_wait(&barrier);
```

`pthread_barrier_wait()` の戻り値

`pthread_barrier_wait()` 関数は、正常終了時に `PTHREAD_BARRIER_SERIAL_THREAD` を返します。この値は `pthread.h` で定義されており、このバリアーで同期がとられた 1 つの任意のスレッドに返されます。ほかの各スレッドには、0 が返されます。それ以外の場合は、エラーコードが返されます。

EINVAL

説明: *barrier* で指定された値が、初期化されたバリアーオブジェクトを表していません。

同期バリアーの削除

バリアーがなくなったら、そのバリアーを削除するようにすべきです。 *barrier* が参照するバリアーを削除し、そのバリアーによって使用されているリソースをすべて解放するには、 `pthread_barrier_destroy(3C)` を使用します。

`pthread_barrier_destroy` の構文

```
int pthread_barrier_destroy(pthread_barrier_t *barrier);

#include <pthread.h>
pthread_barrier_t barrier;
int ret;
ret = pthread_barrier_destroy(&barrier);
```

`pthread_barrier_destroy` の戻り値

`pthread_barrier_destroy()` は、正常終了時に 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

説明: *barrier* の値が無効です。

EBUSY

説明: バリアーが別のスレッドによって使用されている間 (たとえば、`pthread_barrier_wait()` で使用されている間) にそのバリアーの削除の試行が検出されました。

バリアー属性オブジェクトの初期化

`pthread_barrierattr_init(3C)` 関数は、バリアー属性オブジェクト *attr* を、実装によってこのオブジェクト用に定義された属性のデフォルト値に初期化します。現在、プロセス共有の属性のみが提供されており、この属性を取得および設定するために `pthread_barrierattr_getpshared()` および `pthread_barrierattr_setpshared()` 関数が使用されます。

バリアー属性オブジェクトを使用して1つ以上のバリアーを初期化したあと、この属性オブジェクトに影響するどの関数 (削除を含む) を使用しても、前に初期化されたバリアーには影響しません。

pthread_barrierattr_init() の構文

```
int pthread_barrierattr_init(pthread_barrierattr_t *attr);
```

```
#include <pthread.h>
pthread_barrierattr_t attr;
int ret;
ret = pthread_barrierattr_init(&attr);
```

pthread_barrierattr_init() の戻り値

pthread_barrierattr_init() は、正常終了時に 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

説明: バリアー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。

バリアーのプロセス共有の属性の設定

pthread_barrierattr_setpshared() 関数は、*attr* が参照する初期化された属性オブジェクト内のプロセス共有の属性を設定します。プロセス共有の属性は、次のいずれかの値を持ちます。

PTHREAD_PROCESS_PRIVATE	バリアーを操作できるのは、そのバリアー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。これはプロセス共有の属性のデフォルト値です。
PTHREAD_PROCESS_SHARED	バリアーが割り当てられているメモリーにアクセスできる任意のスレッドがバリアーを操作できます。

pthread_barrierattr_setpshared() の構文

```
int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr, int pshared);
```

pthread_barrierattr_setpshared() の戻り値

pthread_barrierattr_setpshared() は、正常終了時に 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

説明: *attr* の値が無効か、または *pshared* に指定された新しい値が無効です。

バリアーのプロセス共有の属性の取得

`pthread_barrierattr_getpshared(3C)` 関数は、`attr` が参照する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。この値は、`pthread_barrierattr_setpshared()` 関数で設定されます。

`pthread_barrierattr_getpshared()` の構文

```
int pthread_barrierattr_getpshared(const pthread_barrierattr_t *restrict attr,
                                   int *restrict pshared);
```

`pthread_barrierattr_getpshared()` の戻り値

`pthread_barrierattr_getpshared()` は、正常終了時に 0 を返し、`attr` のプロセス共有の属性の値を `pshared` パラメータが参照するオブジェクトに格納します。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

説明:`attr` の値が無効です。

バリアー属性オブジェクトの削除

`pthread_barrierattr_destroy()` 関数は、バリアー属性オブジェクトを削除します。削除された `attr` 属性オブジェクトは、`pthread_barrierattr_init()` を使用して再初期化することができます。

バリアー属性オブジェクトを使用して1つ以上のバリアーを初期化したあと、このオブジェクトを削除しても、前に初期化されたバリアーには影響しません。

`pthread_barrierattr_destroy()` の構文

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

```
int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
```

`pthread_barrierattr_destroy()` の戻り値

`pthread_barrierattr_destroy()` は、正常終了時に 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

説明:`attr` の値が無効です。

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できます。プリミティブを設定するには、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。その前に、プロセス間に設定された共有属性でプリミティブを初期化しておく必要があります。

「生産者/消費者」問題の例

例 4-17 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たものです。メインルーチンは、子プロセスと共有されているメモリーを 0 で初期化し、それを自分のアドレス空間にマッピングします。

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つのセマフォは、いっぱいになったバッファ数と空のバッファ数をそれぞれ表します。これらのセマフォは、バッファが空にな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

```
main() {
    int zfd;
    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);
```

例 4-17 プロセスの境界を越えた同期 (続き)

```

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);
    }
}

```

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もっと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。

その次に効率の高いプリミティブは、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。条件変数は、スレッド待ち機能を提供します。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()` から戻ったあと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使用します。セマフォは、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す。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、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。

Solaris ソフトウェアを使ったプログラミング

この章では、マルチスレッドと Solaris ソフトウェア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 ソフトウェアに加えられた変更点についても説明します。

- 159 ページの「プロセスの生成時の `fork` の問題」
- 164 ページの「プロセスの作成: `exec` と `exit` の問題」
- 164 ページの「タイマー、アラーム、およびプロファイル」
- 165 ページの「大域ジャンプ: `setjmp` と `longjmp`」
- 165 ページの「リソースの制限」
- 165 ページの「LWP とスケジューリングクラス」
- 168 ページの「シグナルの拡張」
- 178 ページの「入出力の問題」

プロセスの生成時の **fork** の問題

Solaris 9 製品とそれ以前の Solaris リリースでは、`fork()` のデフォルトの処理方法が、POSIX スレッドでの `fork()` の処理方法と少し異なっていました。Solaris 9 以降のリリースでは、`fork()` は、常に POSIX スレッドの `fork` と同様に動作します。

表 5-1 に、Solaris スレッドと POSIX スレッドの `fork()` の処理の相違点と共通点を示します。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 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	<code>fork1(2)</code>	<code>fork(2)</code>
	<code>fork(2)</code>	

表 5-1 POSIX と Solaris での fork() の処理の比較 (続き)

	Solaris のインタフェース	POSIX スレッドのインタフェース
汎用 fork モデル	forkall(2)	forkall(2)
fork - 安全	—	pthread_atfork(3C)

fork1 モデル

表 5-1 で示すように、POSIX スレッドの fork(2) 関数の動作は、Solaris の fork1(2) 関数の動作と同じです。POSIX スレッドの fork(2) 関数も Solaris の fork1(2) 関数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。しかし、どちらの関数も、子プロセス内に呼び出しスレッドのみを複製します。

呼び出しスレッドを子プロセス内に複製する処理は、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よく使用します。実際、このような状況は、fork() を呼び出したあとによく発生します。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() 呼び出し時に親に保持されているロックを使用するものがあります。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。

「非同期シグナル安全」関数の詳細については、176 ページの「シグナルハンドラと「非同期シグナル安全」」を参照してください。

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一般的な考慮事項に加えて、次のような問題があります。ライブラリは、fork() を呼び出したスレッドだけが実行されているとき、子プロセスを fork することに上手に対処する必要があります。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。

ほとんどのプログラムでは、この問題は発生しません。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、子プロセス内で何らかの処理を行ってから 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可能性があります。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あり、printf() のためにロックを保持しているとします。この状態で、T2 が新しいプロセスを fork すると仮定しましょう。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 `forkall()` を呼び出した場合は、すべてのスレッドが複製されるので、この問題は発生しません。

しかし、`forkall()` では別の問題が発生する可能性があるため、使用する場合は注意が必要です。たとえば、あるスレッドが `forkall()` を呼び出した場合、ファイルへの入出力を実行する親スレッドが子プロセス内に複製されます。スレッドのコピーはどちらも同じファイルへの入出力を続行します。ただし、一方は親、他方は子に入出力を行うので、機能不全やファイルの破損が発生します。

`fork1()` の呼び出し時のデッドロックを防ぐには、`fork` 時に保持されているロックがないようにする必要があります。デッドロックを防ぐためのもっとも明瞭なやり方は、`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()` に対してすべてのロックを獲得する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。

ヒント-Sun Studio ソフトウェアに含まれている Thread Analyzer ユーティリティを使用すると、実行中のプログラム内のデッドロックを検出できます。詳細については、『[Sun Studio 12: スレッドアナライザユーザーズガイド](#)』を参照してください。

ライブラリ内でロックを管理するには、次の処理を行う必要があります。

- 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` 呼び出し時にすべてのロックを獲得できるよう段取りします。

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 $\{L1, \dots, Ln\}$ です。これらのロックのロック順序も $L1 \dots Ln$ です。

```
mutex_lock(L1);
mutex_lock(L2);
fork1(...);
mutex_unlock(L1);
mutex_unlock(L2);
```

Solaris スレッド、POSIX スレッドのどちらを使用する場合でも、`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() /* This is executed just before the process forks. */
{
    mutex_lock(L1); |
```

```
mutex_lock(...); | -- ordered in lock order
mutex_lock(Ln); |
} V

f2() /* This is executed in the child after the process forks. */
{
    mutex_unlock(L1);
    mutex_unlock(...);
    mutex_unlock(Ln);
}

f3() /* This is executed in the parent after the process forks. */
{
    mutex_unlock(L1);
    mutex_unlock(...);
    mutex_unlock(Ln);
}
```

仮想 fork-vfork

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 fork(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。

スレッドが exec(2) を呼び出す前に、子プロセス内のスレッドによってメモリーが変更されないように注意してください。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。子プロセスが親プロセスの状態を変更してはなりません。

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と非常に危険です。

解決策: pthread_atfork

fork 1 モデルを使用するときは必ず、pthread_atfork() を使ってデッドロックを防ぎます。

```
#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。

すべてのハンドラ引数は `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することがあります。次に、*parent* および *child* ハンドラがこれらの相互排他ロックを解放することがあります。必要なすべての相互排他ロックを獲得する *prepare* ハンドラは、プロセスが `fork` される前に、関連するすべてのロックが、`fork` 関数を呼び出すスレッドによって保持されているようにする必要があります。この方法で、子プロセスでのデッドロックを防ぐことができます。

詳細は、[pthread_atfork\(3C\)](#) のマニュアルページを参照してください。

汎用 fork モデル

Solaris `forkall(2)` 関数は、アドレス空間とすべてのスレッドを子プロセスに複製します。アドレス空間の複製は、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場合などに使用します。

プロセス内のあるスレッドが Solaris の `forkall(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` を返します。

親プロセスと子プロセスの両方に保持されるようなロックは作成しないでください。MAP_SHARED フラグを指定して `mmap()` を呼び出すことにより、共有メモリー内にロックを割り当てると、親プロセスと子プロセスの両方に保持されるロックが生成されます。fork1 モデルを使用すると、この問題は発生しません。

正しい fork の選択

Solaris 10 リリースから、`fork()` の呼び出しは `fork1()` の呼び出しと等価になりました。すなわち、子プロセスには、呼び出しスレッドだけが複製されます。この動作は、POSIX の `fork()` の動作と同じです。

以前の Solaris ソフトウェアのリリースでは、`fork()` の動作は、アプリケーションが POSIX スレッドライブラリにリンクされているかどうかによって決定されていました。-lthread (Solaris スレッド) とリンクされていても -lpthread (POSIX スレッド) とリンクされていなければ、`fork()` は `forkall()` と等価でした。-lpthread とリンクされていれば、-lthread とリンクされているかどうかに関係なく、`fork()` は `fork1()` と等価でした。()

Solaris 10 リリースから、マルチスレッドアプリケーションには、-lthread と -lpthread のどちらも不要になりました。マルチスレッドアプリケーションをコンパイルしていることを示すには、-mt オプションを使用します。どちらのアプリケー

シヨンプログラムインタフェースを使用する場合でも、標準Cライブラリがすべてのスレッド機能を提供します。アプリケーションで、すべてのforkセマンティクスを複製する必要がある場合は、`forkall()` を呼び出してください。

プロセスの作成:exec と exit の問題

`exec(2)` と `exit(2)` は、次の点を除いて、シングルスレッドのプロセスの場合と変わりません。マルチスレッドアプリケーションでは、これらの関数は、アドレス空間内のすべてのスレッドを削除します。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。

タイマー、アラーム、およびプロファイル

Solaris OS は、このいくつかのリリースの間に、アラーム、インターバルタイマー、およびプロファイルに関してプロセスごとのモードに展開してきました。

タイマー

LWP ごとであるリアルタイムプロファイルのインターバルタイマーを除き、すべてのタイマーがプロセスごとです。ITIMER_REALPROF タイマーについては、[setitimer\(2\)](#) のマニュアルページを参照してください。

プロセスごとのタイマーのタイマー 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 に対してではなく、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ーは、`timer_delete(3RT)` 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。

アラーム

アラームは、スレッドレベルではなくプロセスレベルで動作します。`alarm()` 関数は、シグナル SIGALRM を呼び出しスレッドではなく呼び出しプロセスに送信します。

マルチスレッドプログラムのプロファイリング

マルチスレッドプロセスの `profil()` システムコールは、そのプロセス内のすべての LWP およびスレッドに大域的な影響力を持ちます。スレッドが、個々のスレッドのプロファイリングのために `profil()` を使用することはできません。詳細は、[profil\(2\)](#) のマニュアルページを参照してください。

ヒント - Sun Studio ソフトウェアに含まれている Performance Analyzer ツールを使用すると、マルチスレッドプログラムやシングルスレッドプログラムの広範囲なプロファイリングを行うことができます。このツールでは、指定した任意の時点でのスレッドの動作を詳細に表示できます。詳細は、[Sun Studio Web ページ](#) および [Sun Studio Information Center](#) を参照してください。

大域ジャンプ: `setjmp` と `longjmp`

`setjmp()` と `longjmp()` のスコープは、1 つのスレッドに制限されています。この制限は、ほとんどの場合問題になりません。しかし、このスコープの制限は、シグナルを扱うスレッドが `longjmp()` を実行できるのは、`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(3C)` で調べることができます。

LWP とスケジューリングクラス

Solaris のカーネルには、ディスパッチ優先順位の 3 つの範囲があります。もっとも高い優先順位の範囲 (100 - 159) は、リアルタイム (RT) スケジューリングクラスに対応しています。中間の優先順位の範囲 (60 - 99) は、システム (SYS) スケジューリングクラスに対応しています。システム (system) クラスは、ユーザープロセスには適用されません。もっとも低い優先順位の範囲 (0 - 59) は、タイムシェアリング (TS)、対話型 (IA)、公平配分 (FSS)、および固定優先順位 (FX) スケジューリングクラスで共有されます。

スケジューリングクラスは、LWP ごとに管理されます。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。スレッドが生成されていく過程で、関連 LWP はこのスケジューリングクラスと優先順位を継承します。

スレッドは、関連付けられている LWP と同じスケジューリングクラスおよび優先順位を持ちます。プロセス内の各 LWP は、カーネルから参照される固有のスケジューリングクラスおよび優先順位を持つことができます。

同期オブジェクトへの競合は、スレッドの優先順位によって調節されます。デフォルトでは、LWP はタイムシェアクラスです。計算が大きな比率を占める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頻繁に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はより意味を持ちます。

スケジューリングクラスは、`prctl(2)` によって設定されます。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、次のいずれかのコマンドを指定できます。()

- `PC_GETCID` - 特定のクラスのクラス ID とクラス属性を取得します。
- `PC_GETCLINFO` - 特定のクラスのクラス名とクラス属性を取得します。
- `PC_GETPARMS` - 単一プロセス、単一プロセスの LWP、または複数のプロセス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ータを取得します。
- `PC_SETPARMS` - 単一プロセス、単一プロセスの LWP、または複数のプロセス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ータを設定します。

LWP のユーザーレベルの優先順位は、そのディスパッチ優先順位ではなく、そのクラス内の優先順位です。この優先順位は、`prctl()` システムコールのアプリケーションによる場合を除き、時間が経っても変更されることはありません。カーネルは、LWP のディスパッチ優先順位を、そのスケジューリングクラス、そのクラス内での優先順位、さらには最近使用された CPU 時間などのほかの要因に基づいて決定します。

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は、プロセッサリソースをタイムシェアリング (TS) および対話型 (IA) スケジューリングクラス内の LWP の間で公平に配分しようとしています。

`prctl(2)` の呼び出しは、1 つ以上のプロセスまたは LWP のクラス優先順位を設定します。タイムシェアクラスの優先順位の標準の範囲は -60 - +60 です。この値が大きいほど、カーネルのディスパッチ優先順位は高くなります。タイムシェアクラスのデフォルトの優先順位は 0 です。

プロセスに対する `nice` 値の古い概念 (`nice` 値が小さいほど高い優先順位を示す) は、TS、IA、および FSS のすべてのスケジューリングクラスのために維持されています。古い `nice` ベースの `setpriority(3C)` と `nice(2)` のインタフェースは、`nice` 値を優先順位の値に対応させることによって引き続き動作します。`nice` 値を設定すると優

先順位が変更され、その逆も同様です。nice 値の範囲は -20 - +20 です。0 の nice 値が 0 の優先順位に対応し、-20 の nice 値が +60 の優先順位に対応します。

タイムシェアクラスの LWP のディスパッチ優先順位は、LWP のその時点での CPU 使用率と、その LWP のクラス優先順位から計算されます。タイムシェアスケジューラにとって、クラス優先順位は、LWP 間の相対的な優先順位を表します。

LWP のクラス優先順位が低いほど、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 よりもディスパッチ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

リアルタイムクラス (RT) は、プロセス全体、または単一プロセス内の 1 つ以上の LWP に適用できます。リアルタイムクラスを使用するには、スーパーユーザー特権が必要です。

リアルタイムクラスの優先順位の標準の範囲は 0 - 59 です。リアルタイムクラス内の LWP のディスパッチ優先順位は、そのクラス優先順位プラス 100 に固定されています。

スケジューラは、もっと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。もっとも優先順位の高いリアルタイム LWP は、優先順位の低い LWP が実行可能になると、優先順位の低いほうの LWP を横取りします。横取りされた LWP は、レベルキューの先頭に配置されます。

リアルタイム LWP は、LWP が横取りされたり、LWP が一時停止したり、リアルタイム優先順位が変更されたりしないかぎり、プロセッサを制御し続けます。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 が実行を停止するのは、LWP が終了するか、入出力イベントでブロックされるか、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横取りされるか、タイムスライスが期限切れになったときだけです。

無限タイムスライスを持つ LWP は、LWP が終了するか、ブロックされるか、または横取りされるまで実行されます。

公平配分スケジューリング

公平配分スケジューラ (FSS) スケジューリングクラスでは、配分に基づいて CPU 時間を割り当てることができます。

公平配分スケジューラクラスの優先順位の標準の範囲は -60 - 60 です。この範囲は、スケジューラによって、TS および IA スケジューリングクラスと同じ範囲 (0 - 59) のディスパッチ優先順位に対応付けされます。プロセス内の LWP は、すべて同じスケジューリングクラスで実行する必要があります。FSS クラスでは、プロセス全体ではなく、個々の LWP のスケジュールを設定します。FSS および TS/IA のクラスを同時に使用すると、どちらのクラスも予期しないスケジュールで動作することがあります。

TS/IA と FSS のスケジューリングクラスプロセスが、同じ CPU を奪い合うことはありません。複数のプロセッサセットを使用する場合、TS/IP と FSS を同一システム内に混在させることができます。ただし、各プロセッサセット内のプロセスはすべて、TS/IA または FSS のどちらかのスケジューリングクラスでなければなりません。

固定優先順位スケジューリング

固定優先順位のスケジューリングクラス (FX) は、リソースの消費量とは無関係に、優先順位および時間量に固定値を割り当てます。プロセスの優先順位は、そのプロセス自体、または適切な特権が割り当てられたほかのプロセスだけが変更できます。FX については、`priocntl(1)` および `dispadm(1M)` のマニュアルページを参照してください。

固定優先順位スケジューラクラスの優先順位の標準の範囲は 0 - 60 です。この範囲は、スケジューラによって、TS および IA スケジューリングクラスと同じ範囲 (0 - 59) のディスパッチ優先順位に対応付けされます。

シグナルの拡張

従来の UNIX のシグナルモデルが、スレッドでも使用できるように、自然な方法で拡張されています。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(3C)`、`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` に設定されている場合、シグナルを受け取ると、受け取り側のプロセス全体に対して処理が実行されます。これらのシグナルには、終了、コアダンプ、停止、継続、無視が含まれます。これらのシグナルを受け取ったときの処理は、プロセス内のすべてのスレッドに対して実行されます。したがって、どのスレッドがシグナルを受け取るかという問題は重要ではありません。

ん。終了、コアダンプ、停止、継続、無視のシグナルは、ハンドラを持ちません。シグナルに関する基本的な情報については、[signal.h\(3HEAD\)](#) のマニュアルページを参照してください。

各スレッドに、固有のシグナルマスクがあります。スレッドとシグナルハンドラがメモリまたはほかの状態を共有しているときに、スレッドはシグナルマスクに基づいて特定のシグナルをブロックします。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)`、`sigsend(2)`、または `sigqueue(3RT)` からプロセスに送信されたシグナルは、そのプロセス内の受け入れ可能な任意のスレッドによって処理されます。

シグナルは、トラップ、例外、および割り込みに分類されます。トラップと例外は、同期的に生成されるシグナルです。割り込みは、非同期的に生成されるシグナルです。

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通常は無視されます。保留状態のシグナルは、カウンタではなくビットで表現されます。しかし、`sigqueue(3RT)` インタフェースを使用してシグナルを送信すれば、複数の同じシグナルのインスタンスを、プロセスのキューに格納することができます。

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、早期終了することがあります。スレッドが早期終了すると、`EINTR` エラーコードが返されます。また、入出力呼び出しの場合、要求したバイト数の一部が転送されません。

マルチスレッドプログラムでは、特に `pthread_cond_wait(3C)` への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(3C)` または `pthread_cond_broadcast(3C)` への応答の場合に限り、エラー (戻り値が 0) を返さずに終了します。しかし、待機中のスレッドが従来の UNIX シグナルを受信した場合、`pthread_cond_wait()` は、誤って呼び起こされた場合でも値 0 を返します。

同期シグナル

`SIGILL`、`SIGFPE`、`SIGSEGV` などのトラップは、スレッド上でゼロ除算や存在しないメモリの参照などの操作が行われたときに発生します。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。

同期シグナルに関しては、シグナルの概念を個々のスレッドに簡単に拡張できます。ハンドラは、同期シグナルを生成したスレッド上で起動します。

しかし、適切なシグナルハンドラが設定されていない場合、そのプロセスはトラップに対してデフォルトの処理を実行します。デフォルトの処理は、問題を起こしたスレッドが生成されたシグナル上でブロックされた場合も適用されます。このようなシグナルのデフォルトの処理では、プロセスの終了に通常コアダンプを伴います。

こうした同期シグナルは、通常、特定の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表します。このような場合には通常、プロセスを終了するのが適切です。

非同期シグナル

SIGINT、SIGIOなどの割り込みは、あらゆるスレッドに対して、プロセス外部の何らかの処理が原因で非同期的に発生します。これらの割り込みは、ほかのスレッドから明示的に送られてきたシグナルである場合もあれば、ユーザーによる Control - C キーの入力など、外部の処理を表す場合もあります。

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。

同じシグナルが複数発生し、1つのプロセスに送られた場合、それぞれのシグナルを別のスレッドで処理できます。ただし、それらのスレッドがシグナルをマスクしていない場合に限られます。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。シグナルハンドラが終了すると、割り込みが発生した時点からプロセスの実行が再開されます。このような制御の復帰方法は、シングルスレッドプロセスで非同期シグナルを扱うのに適しています。[例 5-1](#) を参照してください。

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、このような制御の復帰方法が使用されています。

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)
```

例5-1 継続セマンティクス法 (続き)

```

        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);
}

```

シグナルに関する操作

この節では、シグナルの操作について説明します。

- [171 ページの「スレッドのシグナルマスクの設定」](#)
- [171 ページの「特定のスレッドへのシグナルの送信」](#)
- [172 ページの「特定のシグナルの待機」](#)
- [173 ページの「特定のシグナルを指定された時間だけ待機」](#)

スレッドのシグナルマスクの設定

[pthread_sigmask\(3C\)](#) は、[sigprocmask\(2\)](#) がプロセスに対して行うのと同じ処理を、スレッドに対して行います。[pthread_sigmask\(\)](#) は、スレッドのシグナルマスクを設定します。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_sigmask\(\)](#) を呼び出すのと同様です。詳細は、[sigprocmask\(2\)](#) のマニュアルページを参照してください。

特定のスレッドへのシグナルの送信

[pthread_kill\(3C\)](#) は、スレッド用の [kill\(2\)](#) です。[pthread_kill\(\)](#) の呼び出しは、特定のスレッドにシグナルを送ります。特定のスレッドに送られるシグナルは、プロセスに送られるシグナルとは異なります。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ます。スレッド識別子 (*thread_t* 型) のスコープは局所的であるため、現在のプロセスのスコープにないスレッドは指定できません。

宛先スレッドでシグナルの受信時に行われる処理 (ハンドラ、`SIG_DFL`、または `SIG_IGN`) は、通常どおり大域的です。あるスレッドに、あるプロセスを終了させる `SIGXXX` を送信する場合、宛先スレッドがこのシグナルを受け取った時点でプロセス全体が終了します。

特定のシグナルの待機

マルチスレッドプログラムでは、`sigwait(2)` が優先インタフェースです。これは、非同期シグナルを正しく処理できるからです。()

`sigwait()` は、`sigwait()` 関数の *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、ほかのスレッドは、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することができます。

次の例は、`sigwait()` の構文を示しています。

```
#include <signal.h>
int sigwait(const sigset_t *set, int *sig);
```

指定のシグナルが送られてくると、`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()` では、非同期シグナルを同期的に扱うことができます。こうしたシグナルを扱うスレッドは、`sigwait()` を呼び出したあと、シグナルが到着するとすぐ終了します。`sigwait()` の呼び出し側を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安全に処理させることができます。

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。

通常はsigwait() を呼び出すスレッドを1つ以上作成して、シグナルを待機します。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。

シグナルを受け取ったシグナル処理スレッドは、sigwait() から復帰してそのシグナルを処理したあと、sigwait() を再度呼び出して次のシグナルを待機します。このシグナル処理スレッドは、非同期シグナル安全関数以外にも使用できます。このシグナル処理スレッドは、ほか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。非同期シグナル安全カテゴリについては、[226 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-sigwait() は、非同期シグナルを受け取れません。

特定のシグナルを指定された時間だけ待機

sigtimedwait(3RT) は sigwait(2) に似ていますが、sigtimedwait() は指定時間内にシグナルを受け取らなかった場合に失敗してエラーを返すという点が異なっています。詳細は、[sigtimedwait\(3RT\)](#) のマニュアルページを参照してください。

スレッド指定シグナル

UNIX のシグナル機構が、スレッド指定シグナルの概念で拡張されています。スレッド指定シグナルは、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ほうが、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。sigwait(2) を呼び出すことにより、スレッドはシグナルの発生まで待機できます。[172 ページの「特定のシグナルの待機」](#)を参照してください。

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_sigmask(SIG_BLOCK, &set, NULL);
    pthread_create(NULL, 0, runA, NULL, PTHREAD_DETACHED, NULL);
}
```

例5-2 非同期シグナルと `sigwait(2)` (続き)

```
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のコード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、システムコールの割り込みの危険がない点に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;
```

例5-3 完了セマンティクス法 (続き)

```

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。レジスタには、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、次のwhileループの繰り返しに入っている間に実行されます。

マルチスレッドプログラムでは、`sigsetjmp(3C)` と `siglongjmp(3C)` も使用できます。ただし、あるスレッドは、別のスレッドの `sigsetjmp()` の結果を使用する `siglongjmp()` を実行しない点に注意してください。

`sigsetjmp()` と `siglongjmp()` はシグナルマスクを退避および復元しますが、`setjmp(3C)` と `longjmp(3C)` はシグナルマスクを退避および復元しません。

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 - `sigwait(2)` を同期シグナルで使わないように注意してください。

シグナルハンドラと「非同期シグナル安全」

スレッド安全と似た概念に、「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、`printf(3C)` の呼び出し中にシグナルが発生し、このシグナルのハンドラが `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。この問題を避けるには、`printf()` がシグナルに割り込まれる可能性があるときは、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、同期プリミティブを使用しても解決できません。シグナルハンドラと同期対象の操作の同期をとろうとすると、たちまちデッドロックが発生します。

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。

このハンドラが `printf()` を呼び出す場合、すでに相互排他ロックを保持するスレッドが、新たに相互排他ロックを保持しようとし、その結果、即座にデッドロックが発生します。

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します。たとえば、危険領域でシグナルをマスクしたり、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する方法があります。

表 5-2 に、POSIX が「非同期シグナル安全」を保証しているルーチンを示します。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

<code>_Exit()</code>	<code>fpathconf()</code>	<code>read()</code>	<code>sigset()</code>
<code>_exit()</code>	<code>fstat()</code>	<code>readlink()</code>	<code>sigsuspend()</code>
<code>abort()</code>	<code>fsync()</code>	<code>recv()</code>	<code>socketatmark()</code>
<code>accept()</code>	<code>ftruncate()</code>	<code>recvfrom()</code>	<code>socket()</code>
<code>access()</code>	<code>getegid()</code>	<code>recvmsg()</code>	<code>socketpair()</code>
<code>aio_error()</code>	<code>geteuid()</code>	<code>rename()</code>	<code>stat()</code>
<code>aio_return()</code>	<code>getgid()</code>	<code>rmdir()</code>	<code>symlink()</code>
<code>aio_suspend()</code>	<code>getgroups()</code>	<code>select()</code>	<code>sysconf()</code>

表 5-2 「非同期シグナル安全」関数 (続き)

alarm()	getpeername()	sem_post()	tcdrain()
bind()	getpgrp()	send()	tcflow()
cfgetispeed()	getpid()	sendmsg()	tcflush()
cfgetospeed()	getppid()	sendto()	tcgetattr()
cfsetispeed()	getsockname()	setgid()	tcgetattr()
cfsetospeed()	getsockopt()	setpgid()	tcsendbreak()
chdir()	getuid()	setsid()	tcsetattr()
chmod()	kill()	setsockopt()	tcsetpgrp()
chown()	リンク()	setuid()	time()
clock_gettime()	listen()	shutdown()	timer_getoverrun()
close()	lseek()	sigaction()	timer_gettime()
connect()	lstat()	sigaddset()	timer_settime()
creat()	mkdir()	sigdelset()	times()
dup()	mkfifo()	sigemptyset()	umask()
dup2()	open()	sigfillset()	uname()
execle()	pathconf()	sigismember()	ulink()
execve()	pause()	sleep()	utime()
fchmod()	pipe()	signal()	wait()
fchown()	poll()	sigpause()	waitpid()
fcntl()	posix_trace_event()	sigpending()	write()
fdatasync()	pselect()	sigprocmask()	
fork()	raise()	sigqueue()	

条件変数上で割り込まれた待機

マスクされていない捕獲されたシグナルが条件変数上で待機しているスレッドに配信された場合、シグナルハンドラから復帰すると、スレッドは誤った呼び起こしによって条件待ち関数から復帰します。pthread_cond_wait() および pthread_cond_timedwait() は、pthread_cond_signal() または pthread_cond_broadcast() の呼び出しが別のスレッドによって実行されなかった場合でも 0 を返します。この場合、sigaction() へのフラグとして SA_RESTART が指定され

たかどうかは無視されます。pthread_cond_wait() および pthread_cond_timedwait() 関数は、自動的に再起動されません。いずれの場合も、条件待ちから復帰する前に、関連付けられている相互排他ロックを再度獲得します。

関連付けられている相互排他ロックを再度獲得しても、スレッドがシグナルハンドラを実行しているときに相互排他ロックを獲得しているという意味ではありません。シグナルハンドラ内では、相互排他ロックの状態は不定です。

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、あたかも入出力装置に対して遠隔手続き呼び出しを行なっているかのように、入出力が同期した状態に見えました。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するだけで終了することがあります。

このモデルの利点は、よく使われる手続き呼び出しの考え方が利用されている点です。

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す。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。

非同期モデルは同期モデルほど簡単ではありません。しかし、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。非同期入出力では、要求の発行後、入出力がいつ完了したかをチェックします。同期入出力では、独立したスレッドで入出力を同期的に実行できます。メインスレッド側は、`pthread_join(3C)` の呼び出しなどによって、あとで入出力操作の完了を確認できます。

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ます。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、テープは高速で先送りされます。

ストリーミングをサポートするには、カーネル内のテープドライバはスレッドを使用する必要があります。カーネル内のテープドライバは、割り込みに応答するときに、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。この割り込みは、以前のテープへの書き込み操作が完了したことを示すものです。

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、テープへの書き込み順序を指定することはできません。

非同期入出力操作

```
#include <aio.h>

int aio_read(struct aiocb *aiocbp);

int aio_write(struct aiocb *aiocbp);

int aio_error(const struct aiocb *aiocbp);

ssize_t aio_return(struct aiocb *aiocbp);

int aio_suspend(struct aiocb *list[], int nent,
    const struct timespec *timeout);

int aio_waitn(struct aiocb *list[], uint_t nent, uint_t *nwait,
    const struct timespec *timeout);

int aio_cancel(int fildes, struct aiocb *aiocbp);
```

`aio_read(3RT)` と `aio_write(3RT)` は、概念において `pread(2)` と `pwrite(2)` に似ています。ただし、入出力操作のパラメータが、`aio_read()` または `aio_write()` に渡される非同期入出力制御ブロック (`aiocbp`) に格納される点が異なります。

```
aiocbp->aio_fildes;    /* file descriptor */
aiocbp->aio_buf;        /* buffer */
aiocbp->aio_nbytes;     /* I/O request size */
aiocbp->aio_offset;     /* file offset */
```

さらに、必要に応じて、「`struct sigevent`」のメンバーで非同期通知タイプ (一般には、待ち行列に入れられたシグナル) を指定することができます。

```
aiocbp->aio_sigevent;  /* notification type */
```

`aio_read()` または `aio_write()` を呼び出すと、入出力操作が開始されます (または、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ます。

非同期操作の進行中のエラー状態や復帰状態を判定するために、`aiocbp` の値を `aio_error(3RT)` および `aio_return(3RT)` の引数として使用できます。

入出力操作の完了の待機

1 つ以上の未処理の非同期入出力操作の完了を、`aio_suspend()` または `aio_waitn()` を呼び出すことによって待機することができます。入出力操作の成功または失敗を判定するには、完了した非同期入出力制御ブロックに対して `aio_error()` および `aio_return()` を使用します。

`aio_suspend()` および `aio_waitn()` 関数には、呼び出し側の待ち時間を示す *timeout* 引数があります。NULL ポインタを指定すれば、呼び出し側が無期限に待つという意味になります。値 0 が設定されている構造体を指すポインタは、呼び出し側が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 `aio_suspend()` または `aio_waitn()` で操作の完了を待つことができます。あるいは、`aio_sigevent()` で指定された非同期通知イベントを発生させて操作の完了を通知するという方法もあります。

最後に、保留状態の非同期入出力操作を取り消すときは、`aio_cancel()` を呼び出します。この関数を呼び出すときは、入出力操作を開始するために使用された入出力制御ブロックのアドレスを指定します。

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。この問題は、`lseek(2)` システムコールがファイルオフセットを設定するとき、

入出力が逐次的に行われない場合に発生します。ファイルオフセットは、次の `read(2)` または `write(2)` 呼び出しでファイル内の操作開始位置を指定するときに使用されます。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 `lseek()` を実行してしまうと矛盾が生じます。

この問題を回避するには、システムコール `pread()` と `pwrite()` を使用します。

```
#include <sys/types.h>
#include <unistd.h>

ssize_t pread(int fildes, void *buf, size_t nbyte, off_t offset);

ssize_t pwrite(int fildes, void *buf, size_t nbyte,
               off_t offset);
```

`pread(2)` と `pwrite(2)` は、`read(2)` や `write(2)` と同様に機能しますが、ファイルオフセットの引数が追加されている点が異なります。`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 と putc の代替

標準入出力に関して、もう1つ問題があります。`getc(3C)` や `putc(3C)` などのルーチンは、マクロとして実装されていて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。`getc(3C)` と `putc(3C)` が高速であるため、これらのマクロは、プログラムのループ内でも効率を気にせずに使用できます。

しかし、`getc(3C)` と `putc(3C)` は、スレッドに対して安全になるように変更されたため、以前よりも負荷が大きくなっています。これらのマクロは、相互排他のロックと解除のために、少なくとも2つの内部サブルーチンの呼び出しを必要とします。

この問題を回避するため、代替マクロとして `getc_unlocked(3C)` および `putc_unlocked(3C)` が用意されています。

`getc_unlocked(3C)` と `putc_unlocked(3C)` は、相互排他ロックを獲得しません。これらの `getc_unlocked()` または `putc_unlocked()` マクロは、スレッドに対して安全ではない元の `getc(3C)` および `putc(3C)` と同程度の処理速度を実現しています。

しかし、これらのマクロをスレッドに対して安全な方法で使うためには、`flockfile(3C)` と `funlockfile(3C)` を使って、標準入出力ストリームを保護する `mutex` を明示的にロックし、解放する必要があります。これらのルーチンは、ループの外で呼び出します。`getc_unlocked()` や `putc_unlocked()` はループ内で呼び出します。

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。この章の内容は次のとおりです。

- 183 ページの「Solaris スレッドと POSIX スレッドの API の比較」
- 188 ページの「Solaris スレッドに固有の関数」
- 190 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」
- 196 ページの「Solaris スレッドに類似した関数」
- 208 ページの「pthread に相当するものがある同期関数 — 相互排他ロック」
- 212 ページの「pthread に相当するものがある同期関数 — 条件変数」
- 217 ページの「pthread に相当するものがある同期関数 — セマフォ」
- 223 ページの「fork() と Solaris スレッドに関する問題」

Solaris スレッドと POSIX スレッドの API の比較

Solaris スレッド API と pthread API は、同じ問題に対する 2 つの解決策です。それは、アプリケーションソフトウェアへの並列性構築です。どちらの API もそれ自体で完結したのですが、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 には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、それぞれ他方には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。表 6-1 に、主な相違点を示します。

表 6-1 Solaris スレッドと pthread の相違点

Solaris スレッド	POSIX スレッド
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_
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	スケジューリングポリシー

関数比較表

次の表は、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「(3RT)」が付記されているものは、POSIX.1b リアルタイム拡張機能ライブラリである librt 内の関数であり、pthread の一部ではありません。このライブラリ内の関数は、POSIX.1b リアルタイム拡張機能で指定されているほとんどのインタフェースを提供しています。

表 6-2 Solaris スレッドと POSIX pthread の比較

Solaris スレッド	pthread
thr_create()	pthread_create()
thr_exit()	pthread_exit()
thr_join()	pthread_join()
thr_yield()	sched_yield(3RT)()
thr_self()	pthread_self()
thr_kill()	pthread_kill()
thr_sigsetmask()	pthread_sigmask()
thr_setprio()	pthread_setschedparam()
thr_getprio()	pthread_getschedparam()

表 6-2 Solaris スレッドと POSIX pthread の比較 (続き)

Solaris スレッド	pthread
thr_setconcurrency()	pthread_setconcurrency()
thr_getconcurrency()	pthread_getconcurrency()
thr_suspend()	-
thr_continue()	-
thr_keycreate()	pthread_key_create()
-	pthread_key_delete()
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()
cond_reltimedwait()	pthread_cond_reltimedwait_np()
cond_signal()	pthread_cond_signal()
cond_broadcast()	pthread_cond_broadcast()
cond_init()	pthread_cond_init()
cond_destroy()	pthread_cond_destroy()

表 6-2 Solaris スレッドと POSIX pthread の比較 (続き)

Solaris スレッド	pthread
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(3RT)()
sema_destroy()	sem_destroy(3RT)()
sema_wait()	sem_wait(3RT)()
sema_post()	sem_post(3RT)()
sema_trywait()	sem_trywait(3RT)()
fork1()	fork()
-	pthread_atfork()
forkall() (複数スレッドコピー)	-
-	pthread_mutexattr_init()
-	pthread_mutexattr_destroy()
mutex_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()

表 6-2 Solaris スレッドと POSIX pthread の比較 (続き)

Solaris スレッド	pthread
-	pthread_condattr_getpshared()
-	pthread_attr_init()
-	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_setstack()
-	pthread_attr_getstack()
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()
-	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、Solaris スレッドライブラリ (-lthread) とリンクする必要があります。

Solaris スレッドと pthread では、関数名や引数が異なっていますが、実質上の操作は変わりません。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示すだけとします。Solaris スレッドの関数の戻り値が記述されていないものについては、『[SunOS リファレンスマニュアル 3: 基本ライブラリ関数](#)』で調べてください。

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 にはない機能を持つものについて、詳しく説明しています。

Solaris スレッドに固有の関数

この節では、Solaris スレッドに固有の次の関数について説明します。スレッドの実行を中断する関数と、中断したスレッドを再開する関数です。

スレッド実行の停止

`thr_suspend(3C)` は、`target_thread` で指定されたスレッドの実行をただちに停止します。`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。

`thr_suspend()` が目的のスレッドを停止するとき、そのスレッドが保持しているロックについては考慮されません。したがって、`thr_suspend()` は慎重に使用する必要があります。一時停止したスレッドに保持されているロックを必要とする関数が呼び出されると、デッドロックが発生します。

`thr_suspend` の構文

```
#include <thread.h>
```

```
int thr_suspend(thread_t tid);
```

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。

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; /* tid from thr_create() */

/* pthreads equivalent of Solaris tid from thread created */
/* with pthread_create() */
pthread_t ptid;

int ret;

ret = thr_suspend(tid);

/* using pthreads ID variable with a cast */
ret = thr_suspend((thread_t) ptid);
```

thr_suspend の戻り値

thr_suspend() は、正常終了時に 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

説明: 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(3C) 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。

thr_continue の構文

```
#include <thread.h>
```

```
int thr_continue(thread_t tid);
```

停止しているスレッドは、シグナルでは呼び起こされません。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; /* tid from thr_create()*/
```

```
/* pthreads equivalent of Solaris tid from thread created */
/* with pthread_create()*/
pthread_t ptid;
```

```
int ret;
```

```
ret = thr_continue(tid);
```

```
/* using pthreads ID variable with a cast */
ret = thr_continue((thread_t) ptid)
```

thr_continue の戻り値

thr_continue() は、正常終了時に 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、対応する値を返します。

ESRCH

説明: 現在のプロセスに *tid* が存在しません。

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。この節の内容は次のとおりです。

- 読み取り/書き込みロックの初期化
- 読み取りロックの獲得
- 読み取りロックの獲得の試行
- 書き込みロックの獲得
- 書き込みロックの獲得の試行
- 読み取り/書き込みロックの解除
- 読み取り/書き込みロックの状態の削除

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、ほか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、ほか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、読み取り/書き込みロックでは、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の性能を改善することができます。

このプロセス内のスレッドと別のプロセス内のスレッド間で同期をとるには、読み取り/書き込みロックを使用します。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に、読み取り/書き込みロックの領域を確保します。この動作の読み取り/書き込みロックのマッピングについては、[mmap\(2\)](#)のマニュアルページを参照してください。

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同じ優先順位の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

`rwlp` が指す読み取り/書き込みロックを初期化して、ロック状態を解除するには、`rwlock_init(3C)` を使用します。

rwlock_init の構文

```
#include <synch.h> (or #include <thread.h>)
```

```
int rwlock_init(rwlock_t *rwlp, int type, void *
arg);
```

type は、次のいずれかになります。

- `USYNC_PROCESS` 読み取り/書き込みロックを使って、このプロセスとほかのプロセスのスレッド間で同期をとることができます。*arg* は無視されます。
- `USYNC_THREAD` 読み取り/書き込みロックを使って、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、ほか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。

POSIX スレッドの場合については、[143 ページの「pthread_rwlock_init の構文」](#)を参照してください。

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;
/* to be used within this process only */
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;
/* to be used among all processes */
ret = rwlock_init(&rwlp, USYNC_PROCESS, 0);
```

rwlock_init の戻り値

`rwlock_init()` は、正常終了時に 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

説明: 引数が無効です。

EFAULT

説明: *rwlp* または *arg* が不当なアドレスを指しています。

読み取りロックの獲得

rwlp が指す読み取り/書き込みロック上で読み取りロックを獲得するには、[rw_rdlock\(3C\)](#) を使用します。

rw_rdlock の構文

```
#include <synch.h> (or #include <thread.h>)
```

```
int rw_rdlock(rwlock_t *rwlp);
```

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、呼び出しスレッドは読み取りロックを獲得します。POSIX スレッドの場合については、[144 ページ](#)の「[pthread_rwlock_rdlock の構文](#)」を参照してください。

rw_rdlock の戻り値

`rw_rdlock()` は、正常終了時に 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

説明: 引数が無効です。

EFAULT

説明: *rwlp* が指すアドレスが正しくありません。

読み取りロックの獲得の試行

rwlp が指す読み取り/書き込みロック上に読み取りロックを獲得しようとするには、[rw_tryrdlock\(3C\)](#) を使用します。

rw_tryrdlock の構文

```
#include <synch.h> (or #include <thread.h>)
```

```
int rw_tryrdlock(rwlock_t *rwlp);
```

読み取り/書き込みロックが書き込み用にロックされているとき、`rw_tryrdlock()` はエラーを返します。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 スレッドの場合については、[146 ページ](#)の「[pthread_rwlock_tryrdlock の構文](#)」を参照してください。

rw_tryrdlock の戻り値

rw_tryrdlock() は、正常終了時に 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

説明: 引数が無効です。

EFAULT

説明: *rwlp* が指すアドレスが正しくありません。

EBUSY

説明: *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

rwlp が指す読み取り/書き込みロック上で書き込みロックを獲得するには、[rw_writelock\(3C\)](#) を使用します。

rw_writelock の構文

```
#include <synch.h> (or #include <thread.h>)
```

```
int rw_writelock(rwlock_t *rwlp);
```

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 スレッドの場合については、[147 ページの「pthread_rwlock_rwlock の構文」](#) を参照してください。

rw_writelock の戻り値

rw_writelock() は、正常終了時に 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

説明: 引数が無効です。

EFAULT

説明: *rwlp* が指すアドレスが正しくありません。

書き込みロックの獲得の試行

`rwlp` が指す読み取り/書き込みロック上に読み取りロックを獲得しようとするには、[rw_trywrlock\(3C\)](#) を使用します。

rw_trywrlock の構文

```
#include <synch.h> (or #include <thread.h>)
```

```
int rw_trywrlock(rwlock_t *rwlp);
```

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ロックされているとき、`rw_trywrlock()` はエラーを返します。POSIX スレッドの場合については、[147 ページ](#)の「[pthread_rwlock_trywrlock の構文](#)」を参照してください。

rw_trywrlock の戻り値

`rw_trywrlock()` は、正常終了時に 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

説明: 引数が無効です。

EFAULT

説明: `rwlp` が指すアドレスが正しくありません。

EBUSY

説明: `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

`rwlp` が指す読み取り/書き込みロックを解除するには、[rw_unlock\(3C\)](#) を使用します。

rw_unlock の構文

```
#include <synch.h> (or #include <thread.h>)
```

```
int rw_unlock(rwlock_t *rwlp);
```

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ほかにある場合は、そのスレッドのうちの 1 つがブロック解除されます。POSIX スレッドの場合については、[149 ページ](#)の「[pthread_rwlock_unlock の構文](#)」を参照してください。

rw_unlock の戻り値

rw_unlock() は、正常終了時に 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

説明: 引数が無効です。

EFAULT

説明: *rwlp* が指すアドレスが正しくありません。

読み取り/書き込みロックの削除

rwlp が指す読み取り/書き込みロックに関連付けられた状態を削除するには、[rwlock_destroy\(3C\)](#) を使用します。

rwlock_destroy の構文

```
#include <synch.h> (or #include <thread.h>)
```

```
int rwlock_destroy(rwlock_t *rwlp);
```

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 スレッドの場合については、[150 ページの「pthread_rwlock_destroy の構文」](#) を参照してください。

[例 6-1](#) では、銀行口座の例を使って、読み取り/書き込みロックについて説明します。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、不可分操作によって確実に行われるために必要です。

例 6-1 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);
```

例 6-1 銀行口座の読み取り/書き込み (続き)

```
    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);
}
```

rwlock_destroyの戻り値

rwlock_destroy()は、正常終了時に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

説明: 引数が無効です。

EFAULT

説明: *rwlp* が指すアドレスが正しくありません。

Solaris スレッドに類似した関数

表 6-3 Solaris スレッドに類似した関数

操作	参照先
スレッドの生成	197 ページの「thr_create の構文」
最小のスタックサイズの取得	199 ページの「thr_min_stack の構文」
スレッド識別子の取得	200 ページの「thr_self の構文」
スレッドの実行明け渡し	201 ページの「thr_yield の構文」
スレッドへのシグナルの送信	201 ページの「thr_kill の構文」
呼び出しスレッドのシグナルマスクのアクセス	202 ページの「thr_sigsetmask の構文」
スレッドの終了	202 ページの「thr_exit の構文」

表 6-3 Solaris スレッドに類似した関数 (続き)

操作	参照先
スレッドの終了待ち	203 ページの「thr_join の構文」
スレッド固有データ用キーの作成	204 ページの「thr_keycreate の構文」
スレッド固有データの設定	205 ページの「thr_setspecific の構文」
スレッド固有データの取得	206 ページの「thr_getspecific の構文」
スレッド優先順位の設定	206 ページの「thr_setprio の構文」
スレッド優先順位の取得	208 ページの「thr_getprio の構文」

スレッドの生成

thr_create(3C) ルーチンは、Solaris スレッドインタフェースに含まれるルーチンのうち、もっとも精巧なルーチンの 1 つです。

現在のプロセスに新しい制御スレッドを追加するときは、thr_create(3C) を使用します。POSIX スレッドの場合については、31 ページの「pthread_create の構文」を参照してください。

thr_create の構文

```
#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);
```

新しいスレッドは保留中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します。

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 バイト以上のサイズ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。stack_size が 0 の場合、デフォルトのサイズが使用されます。通常は 0 を指定してください。stack_size が 0 以外の場合は、thr_min_stack() の戻り値よりも大きい値を指定してください。

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。システムが、各スレッドのスタック用に 1M バイトの仮想メモリーをスワップ空間の予約なしで割り当てます。システムは、`mmap(2)` の `-MAP_NORESERVE` オプションを使って割り当てを行います。

start_routine — 新しいスレッドが実行を開始するために使用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。202 ページの「`thr_exit` の構文」を参照してください。

arg — `void` で記述される任意の変数で、通常は 4 バイト値です。それよりも大きな値を渡す場合は、引数を変数を指す方法で間接的に渡す必要があります。

引数は 1 つしか指定できません。複数の引数をとらせるためには、それらを構造体に入れるなどして、1 つのものとしてコーディングします。

flags — 生成されるスレッドの属性を指定します。通常は 0 を指定してください。

flags の値は、以下に示す引数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。`THR_SUSPENDED` 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。
- `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。`THR_DETACHED` 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- 明示的な同期が割り当てられていないとき、停止していない切り離されたスレッドは失敗することがあります。失敗すると、そのスレッドの生成元が `thr_create()` から復帰する前に、そのスレッド ID が別の新しいスレッドに割り当てられます。

- `THR_BOUND` — 新しいスレッドを永続的に LWP 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。Solaris 9 リリースから、結合スレッドと非結合スレッドが区別されなくなりました。すべてのスレッドが結合スレッドとして扱われます。
- `THR_DAEMON` — 新しいスレッドをデーモンにします。デーモンスレッドは常に切り離されます。`THR_DAEMON` を指定すると、暗黙的に `THR_DETACHED` が適用されます。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。

プロセスを終了するには、`exit()` を呼び出すか、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から `thr_exit(3C)` を呼び出します。アプリケーションやそのプロセスによって呼び出されるライブラ

りでは、終了判断の際に無視される (考慮されない) 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 — *new_thread* が **NULL** 以外の場合には、**thr_create()** が正常終了したときに新しいスレッドの ID が格納された場所を指しています。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 **NULL** を指定してください。

thr_create の戻り値

thr_create() は、正常終了時に **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、対応する値を返します。

EAGAIN

説明: 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。

ENOMEM

説明: 新しいスレッドを生成するための十分なメモリがありません。

EINVAL

説明: *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。

最小のスタックサイズの取得

スレッドの最小スタックサイズを取得するには、**thr_min_stack(3C)** を使用します。

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、[72 ページの「スタックについて」](#) を参照してください。

thr_min_stack の構文

```
#include <thread.h>
```

```
size_t thr_min_stack(void);
```

thr_min_stack() は、**NULL** スレッドの実行に必要な空間の量を返します。**NULL** スレッドは、**NULL** 手続きを実行するために生成されたスレッドです。実用的なスレッドに必要なスタック空間は、スタックの絶対最小サイズより大きいので、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。

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サイズを割り当てなければなりません。

ユーザーが指定したスタックを使用してスレッドが作成される場合には、このスレッドの実行に必要な空間量を確保する必要があります。動的にリンクされる実行環境の場合には、スレッドの最小スタック要件を決定するのがさらにむずかしくなります。

カスタムスタックは、次の2つの方法で指定できます。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。スタックサイズを指定するパラメータには、希望のサイズを指定します。

もう1つの方法は、`thr_create()` にスタックのポインタを渡して、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、割り当ての解放もユーザー自身で行う必要があります。つまり、スレッドを終了するときに、そのスタックをユーザーが解放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。

通常、ユーザー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。

ユーザーは、システムにスタックの割り当てを任せることができます。システム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。

thr_min_stackの戻り値

エラーが未定義です。

スレッド識別子の獲得

呼び出しスレッドの ID を取得するには、`thr_self(3C)` を使用します。POSIX スレッドの場合については、[42 ページの「pthread_self の構文」](#)を参照してください。

thr_selfの構文

```
#include <thread.h>
```

```
thread_t thr_self(void);
```

thr_selfの戻り値

エラーが未定義です。

スレッドの実行明け渡し

[thr_yield\(3C\)](#) 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を譲ります。[thr_yield\(\)](#) には、それ以外の効果はありません。ただし、[thr_yield\(\)](#) の呼び出しスレッドが実行権を必ず譲るとは限りません。

thr_yield の構文

```
#include <thread.h>
```

```
void thr_yield(void);
```

thr_yield の戻り値

[thr_yield\(\)](#) は何も返さず、`errno` を設定しません。

スレッドへのシグナルの送信

[thr_kill\(3C\)](#) は、スレッドにシグナルを送ります。POSIX スレッドの場合については、[46 ページの「pthread_kill の構文」](#)を参照してください。

thr_kill の構文

```
#include <thread.h>
```

```
#include <signal.h>
```

```
int thr_kill(thread_t target_thread, int sig);
```

thr_kill の戻り値

[thr_kill\(\)](#) は、正常終了時に 0 を返します。次のいずれかの条件が検出された場合、[thr_kill\(\)](#) は失敗し、対応する値を返します。障害が発生したときには、シグナルは送信されません。

ESRCH

説明: スレッド ID (*thread*) で指定されたスレッドに関連付けられたスレッドがありません。

EINVAL

説明: *sig* 引数の値が 0 以外です。*sig* が無効であるか、サポートされていないシグナル番号です。

呼び出しスレッドのシグナルマスクのアクセス

呼び出しスレッドのシグナルマスクを変更または検査するには、[thr_sigsetmask\(3C\)](#) を使用します。

thr_sigsetmask の構文

```
#include <thread.h>
#include <signal.h>
int thr_sigsetmask(int how, const sigset_t *set,
                  sigset_t *oset);
```

thr_sigsetmask() は、呼び出しスレッドのシグナルマスクを変更または検査します。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。新しいスレッドは、呼び出しスレッドのシグナルマスクと優先順位を継承します。ただし、保留中のシグナルは継承の対象にはなりません。保留中のシグナルは、新しいスレッドでは空になります。

引数 *set* の値が NULL 以外の場合、*set* は、現在ブロックされているシグナルセットを変更するシグナルセットになります。*set* の値が NULL の場合、*how* の値は意味を持たず、スレッドのシグナルマスクは変更されません。この動作を利用して、現在ブロックされているシグナルについて問い合わせることができます。

how の値には、セットの変更方法を指定します。*how* は、次のいずれかの値をとります。

- SIG_BLOCK — *set* は、ブロックするシグナルセットになります。これらのシグナルは、現在のシグナルマスクに追加されます。
- SIG_UNBLOCK — *set* は、ブロックを解除するシグナルセットになります。これらのシグナルは、現在のシグナルマスクから削除されます。
- SIG_SETMASK — *set* は、新しいシグナルマスクになります。現在のシグナルマスクは、*set* で置き換えられます。

thr_sigsetmask の戻り値

thr_sigsetmask() は、正常終了時に 0 を返します。次のいずれかの条件が検出された場合、thr_sigsetmask() は失敗し、対応する値を返します。

EINVAL

説明:*set* が NULL 以外なのに、*how* の値が未定義です。

スレッドの終了

スレッドを終了するには、[thr_exit\(3C\)](#) を使用します。POSIX スレッドの場合については、[48 ページ](#)の「[pthread_exit Syntax](#)」を参照してください。

thr_exit の構文

```
#include <thread.h>

void thr_exit(void *status);
```

thr_exit の戻り値

thr_exit() は呼び出し側に値を返しません。

スレッドの終了待ち

ターゲットスレッドの終了を待つには、[thr_join\(3C\)](#) を使用します。POSIX スレッドの場合については、[32 ページの「pthread_join の構文」](#) を参照してください。

thr_join の構文

```
#include <thread.h>
```

```
int thr_join(thread_t tid, thread_t *departedid, void **status);
```

ターゲットスレッドは、現在のプロセスのメンバーでなければなりません。また、切り離されたスレッドやデーモンスレッドであってはなりません。

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。1つのスレッドだけが正常に終了します。ほかのスレッドは、ESRCH エラーを発行して終了します。

thr_join() は、ターゲットスレッドがすでに終了している場合に、呼び出しスレッドの処理をブロックすることはありません。

thr_join、指定したスレッドの終了待ち

```
#include <thread.h>
```

```
thread_t tid;
thread_t departedid;
int ret;
void *status;
```

```
/* waiting to join thread "tid" with status */
ret = thr_join(tid, &departedid, &status);
```

```
/* waiting to join thread "tid" without status */
ret = thr_join(tid, &departedid, NULL);
```

```
/* waiting to join thread "tid" without return id and status */
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。

thr_join、任意のスレッドの終了待ち

```
#include <thread.h>
```

```
thread_t tid;  
thread_t departedid;  
int ret;  
void *status;
```

```
/* waiting to join any non-detached thread with status */  
ret = thr_join(0, &departedid, &status);
```

thr_join() でスレッド識別子としてゼロを指定すると、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。

thr_join の戻り値

thr_join() は、正常終了時に 0 を返します。次のいずれかの条件が検出された場合、thr_join() は失敗し、対応する値を返します。

ESRCH

説明: ターゲットスレッド ID に対応する、切り離されていないスレッドが見つかりません。

EDEADLK

説明: デッドロックが検出されたか、ターゲットスレッドの値に呼び出しスレッドが指定されています。

スレッド固有データ用キーの作成

thr_keycreate(3C) は、プロセス内のスレッド固有データを識別するために使用される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に大域的なキーです。各スレッドは、このキーの生成時に値を 1 つずつ割り当てます。

スレッド固有のデータは、POSIX スレッドの場合も Solaris スレッドの場合も、関数名と引数を除いて同じです。この節では、Solaris の関数の概要を説明します。POSIX スレッドの場合については、[36 ページの「pthread_key_create の構文」](#)を参照してください。

thr_keycreate の構文

```
#include <thread.h>
```

```
int thr_keycreate(thread_key_t *keyp,  
void (*destructor) (void *value));
```

keyp は、各結合スレッド固有の値を個別に管理します。各スレッドは、スレッド固有のデータへのアクセスを許可するために、最初に固有要素の *keyp* に結合されます。キーが生成されるときに、すべてのアクティブなスレッドに対して、新しいキーに値 NULL が割り当てられます。また、スレッドが生成されるときに、新しいスレッドのすでに生成されているすべてのキーには、値 NULL が割り当てられます。

オプションの *destructor* 関数は、各 *keyp* に関連付けることができます。スレッドが終了するときに、*keyp* に NULL 以外 *destructor* が関連付けられ、そのスレッドの *keyp* に NULL 以外の *value* が関連付けられている場合には、現在関連付けられている *value* を使用して *destructor* が呼び出されます。スレッドが終了するときに 1 つ以上の *destructor* が存在する場合には、デストラクタは不特定に呼び出されます。

thr_keycreate の戻り値

thr_keycreate() は、正常終了時に 0 を返します。次のいずれかの条件が検出された場合、thr_keycreate() は失敗し、対応する値を返します。

EAGAIN

説明: 追加のスレッド固有のデータキーを生成するために必要なリソースがありません。または、キーの数が、PTHREAD_KEYS_MAX に指定されたプロセスごとの最大キー数を超えています。

ENOMEM

説明: *value* と *keyp* を関連付けるために必要なメモリーが不足しています。

スレッド固有データの値の設定

thr_setspecific(3C) は、呼び出しスレッドに対して、スレッド固有データキー *key* に *value* を結合します。POSIX スレッドの場合については、[38 ページ](#) の「[pthread_setspecific の構文](#)」を参照してください。

thr_setspecific の構文

```
#include <thread.h>
```

```
int thr_setspecific(thread_key_t key, void *value);
```

thr_setspecific の戻り値

thr_setspecific() は、正常終了時に 0 を返します。次のいずれかの条件が検出された場合、thr_setspecific() は失敗し、対応する値を返します。

ENOMEM

説明: *value* と *key* を関連付けるために必要なメモリーが不足しています。

EINVAL

説明: *key* が無効です。

スレッド固有データの値の取得

[thr_getspecific\(3C\)](#) は、呼び出しスレッドの *key* に結合された現在の値を *valuep* が指す場所に格納します。POSIX スレッドの場合については、[38 ページ](#) の「[pthread_getspecific の構文](#)」を参照してください。

thr_getspecific の構文

```
#include <thread.h>
```

```
int thr_getspecific(thread_key_t key, void **valuep);
```

thr_getspecific の戻り値

[thr_getspecific\(\)](#) は、正常終了時に 0 を返します。次のいずれかの条件が検出された場合、[thr_getspecific\(\)](#) は失敗し、対応する値を返します。

ENOMEM

説明: *value* と *key* を関連付けるために必要なメモリーが不足しています。

EINVAL

説明: *key* が無効です。

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。スレッドの優先順位は、停止している間に [thr_setprio\(3C\)](#) 関数を呼び出して変更されます。[thr_setprio\(\)](#) が終了すると、スレッドは実行を再開します。

優先順位の高いスレッドは、同期オブジェクトの競合を考慮しながら、優先順位の低いスレッドより優先されます。

thr_setprio の構文

[thr_setprio\(3C\)](#) は、*tid* に指定されたスレッドの優先順位を、現在のプロセス内で *newprio* に指定された優先順位に変更します。POSIX スレッドの場合については、[44 ページ](#) の「[pthread_setschedparam の構文](#)」を参照してください。

```
#include <thread.h>

int thr_setprio(thread_t tid, int newprio)

スレッドの有効な優先順位の範囲は、スケジューリングポリシーによって異なります。

thread_t tid;
int ret;
int newprio = 20;

/* suspended thread creation */
ret = thr_create(NULL, NULL, func, arg, THR_SUSPENDED, &tid);

/* set the new priority of suspended child thread */
ret = thr_setprio(tid, newprio);

/* suspended child thread starts executing with new priority */
ret = thr_continue(tid);
```

thr_setprio の戻り値

thr_setprio() は、正常終了時に 0 を返します。次のいずれかの条件が検出された場合、thr_setprio() は失敗し、対応する値を返します。

ESRCH

説明: *tid* で指定した値が既存のスレッドを表していません。

EINVAL

説明: *priority* の値は、指定されたスレッドのスケジューリングポリシーには無効です。

EPERM

説明: 呼び出し元は、優先順位を指定された値に設定するための適切なアクセス権を持っていません。

スレッド優先順位の取得

スレッドの現在の優先順位を取得するには、[thr_getprio\(3C\)](#) を使用します。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 スレッドの場合については、[45 ページ](#)の「[pthread_getschedparam の構文](#)」を参照してください。

thr_getprio の構文

```
#include <thread.h>
```

```
int thr_getprio(thread_t tid, int *newprio)
```

thr_getprio の戻り値

thr_getprio() は、正常終了時に 0 を返します。次の条件が検出された場合、thr_getprio() は失敗し、対応する値を返します。

ESRCH

説明: *tid* で指定した値が既存のスレッドを表していません。

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 の削除
- mutex の獲得
- mutex の解除
- mutex の獲得の試行

mutex の初期化

mp が指す mutex を初期化するには、[mutex_init\(3C\)](#) を使用します。POSIX スレッドの場合については、[94 ページ](#)の「[mutex の初期化](#)」を参照してください。

mutex_init(3C) の構文

```
#include <synch.h>
```

```
#include <thread.h>
```

```
int mutex_init(mutex_t *mp, int type, void *arg);
```

type は、次のいずれかになります。

- USYNC_PROCESS — mutex を使って、このプロセス内のスレッドとほか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ます。*arg* は無視されます。
- USYNC_PROCESS_ROBUST — mutex を使って、このプロセス内のスレッドとほか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ます。*arg* は無視されます。
- USYNC_THREAD — mutex を使って、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ます。*arg* は無視されます。

USYNC_PROCESS ロックした状態でプロセスが異常終了すると、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。
- 異常終了し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、そのロックを受け取ります。このロックを受け取るときに、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すエラーコードも受け取ります。

0 に初期化されたメモリ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。相互排他ロックは、ほか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 mutex

```
#include <thread.h>

mutex_t mp;
int ret;

/* to be used within this process only */
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;

/* to be used among all processes */
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_init の戻り値

mutex_init() は、正常終了時に 0 を返します。次のいずれかの条件が検出された場合、mutex_init() は失敗し、対応する値を返します。

EFAULT

説明: *mp* が指すアドレスが正しくありません。

EINVAL

説明: *mp* で指定された値が無効です。

ENOMEM

説明: mutex の初期化に必要なメモリーが不足しています。

EAGAIN

説明: mutex の初期化に必要なリソースが不足しています。

EBUSY

説明: アクティブな mutex の再初期化を試行していることが検出されました。

mutex の削除

mp が指す mutex に関連付けられた状態を削除するには、[mutex_destroy\(3C\)](#) を使用します。mutex の記憶領域は解放されません。POSIX スレッドの場合については、[102 ページの「pthread_mutex_destroy の構文」](#)を参照してください。

mutex_destroy の構文

```
#include <thread.h>
```

```
int mutex_destroy (mutex_t *mp);
```

mutex_destroy の戻り値

mutex_destroy() は、正常終了時に 0 を返します。次の条件が検出された場合、mutex_destroy() は失敗し、対応する値を返します。

EFAULT

説明: *mp* が指すアドレスが正しくありません。

mutex の獲得

mp が指す mutex をロックするには、[mutex_lock\(3C\)](#) を使用します。mutex がすでにロックされている場合、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。ブロックされたスレッドは、優先順位の高いキューで待機します。POSIX スレッドの場合については、[97 ページ](#)の「[pthread_mutex_lock の構文](#)」を参照してください。

mutex_lock の構文

```
#include <thread.h>

int mutex_lock(mutex_t *mp);
```

mutex_lock の戻り値

mutex_lock() は、正常終了時に 0 を返します。次のいずれかの条件が検出された場合、mutex_lock() は失敗し、対応する値を返します。

EFAULT

説明: *mp* が指すアドレスが正しくありません。

EDEADLK

説明: mutex はすでにロックされていて、呼び出しスレッドに保持されています。

mutex の解除

mp が指す mutex のロックを解除するには、[mutex_unlock\(3C\)](#) を使用します。この mutex はロックされていなければなりません。呼び出しスレッドは、最後にこの mutex をロックしたスレッド、すなわち所有者でなければなりません。POSIX スレッドの場合については、[98 ページ](#)の「[pthread_mutex_unlock の構文](#)」を参照してください。

mutex_unlock の構文

```
#include <thread.h>

int mutex_unlock(mutex_t *mp);
```

mutex_unlock の戻り値

mutex_unlock() は、正常終了時に 0 を返します。次のいずれかの条件が検出された場合、mutex_unlock() は失敗し、対応する値を返します。

EFAULT

説明: *mp* が指すアドレスが正しくありません。

EPERM

説明: 呼び出しスレッドは *mutex* を所有していません。

mutex の獲得の試行

mp が指す *mutex* のロックを試行するには、[mutex_trylock\(3C\)](#) を使用します。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 スレッドの場合については、[99 ページ](#)の「[pthread_mutex_trylock の構文](#)」を参照してください。

mutex_trylock の構文

```
#include <thread.h>
```

```
int mutex_trylock(mutex_t *mp);
```

mutex_trylock の戻り値

[mutex_trylock\(\)](#) は、正常終了時に 0 を返します。次のいずれかの条件が検出された場合、[mutex_trylock\(\)](#) は失敗し、対応する値を返します。

EFAULT

説明: *mp* が指すアドレスが正しくありません。

EBUSY

説明: アクティブな *mutex* の再初期化を試行していることが検出されました。

pthread に相当するものがある同期関数 — 条件変数

- 条件変数の初期化
- 条件変数の削除
- 条件変数によるブロック
- 条件変数による指定時刻付きブロック
- 時間間隔によるブロック
- 単一スレッドのブロック解除
- 全スレッドのブロック解除

条件変数の初期化

cv が指す条件変数を初期化するには、[cond_init\(3C\)](#) を使用します。

cond_init の構文

```
#include <thread.h>
```

```
int cond_init(cond_t *cv, int type, int arg);
```

type は、次のいずれかになります。

- **USYNC_PROCESS**。条件変数を使って、このプロセス内のスレッドとほか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ます。*arg* は無視されます。
- **USYNC_THREAD**。条件変数を使って、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ほか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。

POSIX スレッドの場合については、[113 ページの「pthread_condattr_init の構文」](#)を参照してください。

プロセス内スコープでの条件変数

```
#include <thread.h>
```

```
cond_t cv;
int ret;
```

```
/* to be used within this process only */
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;
```

```
/* to be used among all processes */
ret = cond_init(&cv, USYNC_PROCESS, 0);
```

cond_init の戻り値

`cond_init()` は、正常終了時に 0 を返します。次のいずれかの条件が検出された場合、`cond_init()` は失敗し、対応する値を返します。

EFAULT

説明: *cv* が指すアドレスが正しくありません。

EINVAL

説明: *type* を認識できません。

条件変数の削除

cv が指す条件変数に関連付けられた状態を削除するには、[cond_destroy\(3C\)](#) を使用します。条件変数を格納する領域は解放されません。POSIX スレッドの場合については、[114 ページ](#)の「[pthread_condattr_destroy の構文](#)」を参照してください。

cond_destroy の構文

```
#include <thread.h>

int cond_destroy(cond_t *cv);
```

cond_destroy の戻り値

`cond_destroy()` は、正常終了時に 0 を返します。次のいずれかの条件が検出された場合、`cond_destroy()` は失敗し、対応する値を返します。

EFAULT

説明: *cv* が指すアドレスが正しくありません。

EBUSY

説明: アクティブな条件変数の削除を試行していることが検出されました。

条件変数によるブロック

mp が指す `mutex` を不可分的に解放することにより、*cv* が指す条件変数上で呼び出しスレッドをブロックするには、[cond_wait\(3C\)](#) を使用します。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すことができます。

`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。

cond_wait の構文

```
#include <thread.h>

int cond_wait(cond_t *cv, mutex_t *mp);
```

cond_wait の戻り値

cond_wait() は、正常終了時に 0 を返します。次のいずれかの条件が検出された場合、cond_wait() は失敗し、対応する値を返します。

EFAULT

説明: *cv* が指すアドレスが正しくありません。

EINTR

説明: シグナルにより、待機中に割り込みが発生しました。

条件変数による指定時刻付きブロック

cond_timedwait(3C) は、cond_wait() と非常によく似ています。ただし、cond_timedwait() は、*abstime* で指定された時刻を過ぎるとブロック状態を解除します。POSIX スレッドの場合については、[122 ページの「pthread_cond_timedwait の構文」](#)を参照してください。

cond_timedwait の構文

```
#include <thread.h>
```

```
int cond_timedwait(cond_t *cv, mutex_t *mp, timestruct_t abstime);
```

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たときか、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。

cond_timedwait の戻り値

cond_timedwait() は、正常終了時に 0 を返します。次のいずれかの条件が検出された場合、cond_timedwait() は失敗し、対応する値を返します。

EFAULT

説明: *cv* が指すアドレスが正しくありません。

ETIME

説明: *abstime* で指定された時刻を過ぎています。

EINVAL

説明: *abstime* が無効です。

時間間隔によるブロック

`cond_reltimedwait(3C)` は、3 番目の引数の値を除いて、`cond_timedwait()` と非常によく似ています。`cond_reltimedwait()` の 3 番目の引数には、絶対的な時刻ではなく、相対的な時間間隔値を指定します。POSIX スレッドの場合については、`pthread_cond_reltimedwait_np(3C)` のマニュアルページを参照してください。

`cond_reltimedwait()` 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_reltimedwait()` 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たときか、最後の引数で指定した時間間隔が経過したときです。

`cond_reltimedwait` の構文

```
#include <thread.h>

int cond_reltimedwait(cond_t *cv, mutex_t *mp,
    timestruct_t reltime);
```

`cond_reltimedwait` の戻り値

`cond_reltimedwait()` は、正常終了時に 0 を返します。次のいずれかの条件が検出された場合、`cond_reltimedwait()` は失敗し、対応する値を返します。

EFAULT

説明: *cv* が指すアドレスが正しくありません。

ETIME

説明: *reltime* で指定された時間を過ぎています。

特定のスレッドのブロック解除

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するには、`cond_signal(3C)` を使用します。スレッドがブロックされていない条件変数に対して `cond_signal()` を実行しても無視されます。

`cond_signal` の構文

```
#include <thread.h>

int cond_signal(cond_t *cv);
```

cond_signal の戻り値

cond_signal() は、正常終了時に 0 を返します。次の条件が検出された場合、cond_signal() は失敗し、対応する値を返します。

EFAULT

説明: *cv* が指すアドレスが正しくありません。

全スレッドのブロック解除

cv 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するには、[cond_broadcast\(3C\)](#) を使用します。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。

cond_broadcast の構文

```
#include <thread.h>
```

```
int cond_broadcast(cond_t *cv);
```

cond_broadcast の戻り値

cond_broadcast() は、正常終了時に 0 を返します。次の条件が検出された場合、cond_broadcast() は失敗し、対応する値を返します。

EFAULT

説明: *cv* が指すアドレスが正しくありません。

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。この節の内容は次のとおりです。

- セマフォの初期化
- セマフォの加算
- セマフォの値によるブロック
- セマフォの減算
- セマフォの削除

セマフォの初期化

sp が指すセマフォ変数を *count* の値で初期化するには、[sema_init\(3C\)](#) を使用します。

sema_init の構文

```
#include <thread.h>
```

```
int sema_init(sema_t *sp, unsigned int count, int type,  
              void *arg);
```

type は、次のいずれかになります。

- `USYNC_PROCESS`。セマフォを使って、このプロセス内のスレッドとほか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`。セマフォを使って、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ます。*arg* は無視されます。

複数のスレッドから同時に同じセマフォを初期化してはいけません。一度初期化したセマフォは、ほか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;
```

```
/* to be used within this process only */  
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;
```

```
/* to be used among all the processes */  
ret = sema_init (&sp, count, USYNC_PROCESS, 0);
```

sema_init の戻り値

`sema_init()` は、正常終了時に 0 を返します。次のいずれかの条件が検出された場合、`sema_init()` は失敗し、対応する値を返します。

EINVAL

説明: *sp* が無効なセマフォを参照しています。

EFAULT

説明: *sp* または *arg* が無効なアドレスを指しています。

セマフォの加算

sp が指すセマフォを不可分的に加算するには、[sema_post\(3C\)](#) を使用します。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。

sema_post の構文

```
#include <thread.h>

int sema_post(sema_t *sp);
```

sema_post の戻り値

`sema_post()` は、正常終了時に 0 を返します。次のいずれかの条件が検出された場合、`sema_post()` は失敗し、対応する値を返します。

EINVAL

説明: *sp* が無効なセマフォを参照しています。

EFAULT

説明: *sp* が指すアドレスが正しくありません。

EOVERFLOW

説明: *sp* が指すセマフォの値が `SEM_VALUE_MAX` を超えています。

セマフォの値によるブロック

sp が指すセマフォ内のカウントが 0 より大きくなるまで呼び出しスレッドをブロックするには、[sema_wait\(3C\)](#) を使用します。カウントが 0 より大きくなると、不可分的に減算します。

sema_wait の構文

```
#include <thread.h>

int sema_wait(sema_t *sp);
```

sema_wait の戻り値

sema_wait() は、正常終了時に 0 を返します。次のいずれかの条件が検出された場合、sema_wait() は失敗し、対応する値を返します。

EINVAL

説明: *sp* が無効なセマフォを参照しています。

EINTR

説明: シグナルにより、待機中に割り込みが発生しました。

セマフォの減算

カウントが 0 より大きくなったとき、*sp* が指すセマフォ内のカウントを不可分的に減算するには、sema_trywait(3C) を使用します。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。

sema_trywait の構文

```
#include <thread.h>
```

```
int sema_trywait(sema_t *sp);
```

sema_trywait の戻り値

sema_trywait() は、正常終了時に 0 を返します。次のいずれかの条件が検出された場合、sema_trywait() は失敗し、対応する値を返します。

EINVAL

説明: *sp* が無効なセマフォを参照しています。

EBUSY

説明: *sp* が指すセマフォのカウントが 0 です。

セマフォの削除

sp が指すセマフォに関連付けられた状態を削除するには、sema_destroy(3C) を使用します。セマフォの記憶領域は解放されません。

sema_destroy(3C) の構文

```
#include <thread.h>
```

```
int sema_destroy(sema_t *sp);
```

sema_destroy(3C) の戻り値

sema_destroy() は、正常終了時に 0 を返します。次の条件が検出された場合、sema_destroy() は失敗し、対応する値を返します。

EINVAL

説明: *sp* が無効なセマフォを参照しています。

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。この境界を越えた設定を行うには、まず、その同期変数が共有メモリーセグメントに確保されるようにします。次に、USYNC_PROCESS 型に設定された適切な初期化ルーチン (init) を呼び出します。

型が USYNC_PROCESS に設定されている場合、同期変数の操作は、*type* が USYNC_THREAD であるときと同様に行われます。

```
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);
```

「生産者 / 消費者」問題の例

例 6-2 に、「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示します。メインルーチンは、子プロセスと共有されているメモリーを 0 で初期化し、それを自分のアドレス空間にマッピングします。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 6-2 のデータ構造は、条件変数によるソリューションで使ったものと同じです。105 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 6-2 USYNC_PROCESSを使用した「生産者/消費者」問題

```
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 10 リリースより前のリリースでは、Solaris スレッドと POSIX スレッドによる `fork()` の定義が異なっていました。`fork()` の問題の詳細は、[164 ページの「プロセスの作成: `exec` と `exit` の問題](#)」を参照してください。

Solaris `libthread` は、`fork()` と `fork1()` の両方をサポートしていました。`fork()` 呼び出しには、「汎用 `fork`」セマンティクスがあります。`fork()` は、スレッド、LWP を含めて、プロセス内のすべてを複製し、親の完全なクローンを生成しました。一方、`fork1()` 呼び出しで作成されるクローンはスレッドを 1 つしかもちませんでした。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した。

POSIX `libpthread` は、`fork()` のみをサポートしていました。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()` のセマンティクスが「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した。`-lthread` を使ってリンクすれば、`fork()` に「汎用 `fork`」セマンティクスが割り当てられます。`-lpthread` を使ってリンクすれば、`fork()` に「`fork1`」セマンティクスが割り当てられます。

Solaris 10 リリースから、Solaris スレッドと POSIX スレッドの `fork()` のセマンティクスが共通になりました。つまり、`fork1()` セマンティクスが呼び出し側だけを複製するようになりました。すべてを複製するセマンティクスが必要なアプリケーションには、新しい関数 `forkall()` が提供されています。

詳細は、[233 ページの「マルチスレッドプログラムのコンパイルとリンク](#)」を参照してください。

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。この章では、次の内容について説明します。

- 225 ページの「スレッド安全」
- 226 ページの「マルチスレッドインタフェースの安全レベル」
- 228 ページの「Solaris スレッドでの「非同期シグナル安全」関数」
- 229 ページの「ライブラリの「MT-安全」レベル」

スレッド安全

「スレッド安全」は、データの競合が回避されていることを示します。データの競合は、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ときに発生します。

スレッド間でデータを共有する必要がある場合は、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。

- 「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 7-1 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。この 1 つの `mutex` 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つ必要はありません。これらのスレッドは、出力ファイルを共有している場合にのみ同期を必要とします。

最後のルーチンは、「MT-安全」の例です。このバージョンでは、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ということになります。

例7-1 「スレッド安全」の段階

```
/* not thread-safe */
fputs(const char *s, FILE *stream) {
    char *p;
    for (p=s; *p; p++)
        putc((int)*p, stream);
}

/* serializable */
fputs(const char *s, FILE *stream) {
    static mutex_t mut;
    char *p;
    mutex_lock(&mut);
    for (p=s; *p; p++)
        putc((int)*p, stream);

    mutex_unlock(&mut);
}

/* MT-Safe */
mutex_t m[NFILE];
fputs(const char *s, FILE *stream) {
    char *p;
    mutex_lock(&m[fileno(stream)]);
    for (p=s; *p; p++)
        putc((int)*p, stream);
    mutex_unlock(&m[fileno(stream)]);
}
```

マルチスレッドインタフェースの安全レベル

関数やインタフェースのマニュアルページは、その関数またはインタフェースでのスレッドのサポート範囲を示しています。各マニュアルページの「ATTRIBUTES」セクションには、[表7-1](#)の安全レベルのカテゴリのいずれかに設定されたMTレベルの属性が示されています。これらのカテゴリの詳細は、[attributes\(5\)](#)のマニュアルページを参照してください。

マニュアルページに関数が「MT-安全」であることが明示的に記載されていない場合、その関数は安全でないと見なす必要があります。

表 7-1 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
「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。このインタフェースは、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
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228 ページの「 Solaris スレッドでの「非同期シグナル安全」関数 」を参照してください。
「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

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。
- ライブラリが安全で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。再入可能な関数の名前は、元の関数名の末尾に「_r」が追加された形式になっています。

注 - 名前の末尾に「_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、その関数のマニュアルページで確認してください。「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。

安全でないインタフェースのための再入可能な関数

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。「MT-安全」なルーチンの名前は、安全でないルーチンの名前の末尾に「_r」が追加された形式になっています。たとえば、`asctime()` の「MT-安全」なバージョンは `asctime_r()` です。表 7-2 に、Solaris 環境に用意されている「_r」ルーチンを示します。

表 7-2 再入可能な関数

<code>asctime_r(3c)</code>	<code>gethostbyname_r(3nsl)</code>	<code>getservbyname_r(3socket)</code>
<code>ctermid_r(3c)</code>	<code>gethostent_r(3nsl)</code>	<code>getservbyport_r(3socket)</code>
<code>ctime_r(3c)</code>	<code>getlogin_r(3c)</code>	<code>getservent_r(3socket)</code>
<code>fgetgrent_r(3c)</code>	<code>getnetbyaddr_r(3socket)</code>	<code>getspent_r(3c)</code>
<code>fgetpwent_r(3c)</code>	<code>getnetbyname_r(3socket)</code>	<code>getspnam_r(3c)</code>
<code>fgetspent_r(3c)</code>	<code>getnetent_r(3socket)</code>	<code>gmtime_r(3c)</code>
<code>gamma_r(3m)</code>	<code>getnetgrent_r(3c)</code>	<code>lgamma_r(3m)</code>
<code>getauclassent_r(3bsm)</code>	<code>getprotobyname_r(3socket)</code>	<code>localtime_r(3c)</code>
<code>getauclassnam_r(3bsm)</code>	<code>getprotobynumber_r(3socket)</code>	<code>nis_sperror_r(3nsl)</code>
<code>getauevent_r(3bsm)</code>	<code>getprotoent_r(3socket)</code>	<code>rand_r(3c)</code>
<code>getauevnam_r(3bsm)</code>	<code>getpwent_r(3c)</code>	<code>readdir_r(3c)</code>
<code>getauevnum_r(3bsm)</code>	<code>getpwnam_r(3c)</code>	<code>strtok_r(3c)</code>
<code>getgrent_r(3c)</code>	<code>getpwuid_r(3c)</code>	<code>tmpnam_r(3c)</code>
<code>getgrgid_r(3c)</code>	<code>getrpcbyname_r(3nsl)</code>	<code>ttyname_r(3c)</code>
<code>getgrnam_r(3c)</code>	<code>getrpcbynumber_r(3nsl)</code>	
<code>gethostbyaddr_r(3nsl)</code>	<code>getrpcent_r(3nsl)</code>	

Solaris スレッドでの「非同期シグナル安全」関数

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を「非同期シグナル安全」関数と呼びます。「IEEE Std 1003.1-2004」(POSIX) 規格には、表 5-2 に示されている「非同期シグナル安全」関数が定義されています。これらの標準の「非同期シグナル安全」関数に加え、Solaris スレッドインタフェースの次の関数も「非同期シグナル安全」関数です。

- `sema_post(3C)`
- `thr_sigsetmask(3C)`。 `pthread_sigmask(3C)` と類似
- `thr_kill(3C)`。 `pthread_kill(3C)` 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。したがって、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。次の表に、よく使用される「MT-安全」なライブラリを示します。これらのライブラリは、`/usr/lib` ディレクトリにあります。

表 7-3 「MT-安全」なライブラリの例

ライブラリ	コメント
<code>libc</code>	安全でないインタフェースの「スレッド安全」形式には、対応する「*_r」(セマンティクスが異なることが多い)形式の「スレッド安全」なインタフェースがある
<code>libm</code>	System V Interface Definition 第3版、X/Open および ANSI C に準拠した数学ライブラリ
<code>libmalloc</code>	領域を効果的に使用するメモリーの割り当てライブラリ。 <code>malloc(3MALLOC)</code> を参照
<code>libmapmalloc</code>	<code>mmap</code> を使用した代替メモリー割り当てライブラリ。 <code>mapmalloc(3MALLOC)</code> を参照
<code>libnsl</code>	TLI インタフェース、XDR、RPC クライアントとサーバー、 <code>netdir</code> 、 <code>netselect</code> 、および <code>getXXbyYY</code> インタフェースは安全ではないが、対応する <code>getXXbyYY_r</code> 形式のスレッド安全なインタフェースがある
<code>libresolv</code>	ドメインネームサーバーのライブラリルーチン
<code>libsocket</code>	ネットワーク接続用のソケットライブラリ
<code>libX11</code>	X11 ウィンドウシステムライブラリルーチン
<code>libCrun</code>	Sun C++ 5.0 コンパイラ用の C++ 実行時共有オブジェクト
<code>libCstd</code>	Sun C++ 5.0 コンパイラ用の C++ 標準ライブラリ
<code>libiostream</code>	Sun C++ 5.0 コンパイラ用の古典的 <code>iostream</code> ライブラリ

表 7-3 「MT-安全」なライブラリの例 (続き)

ライブラリ	コメント
libC.so.5	Sun C++ 4.0 コンパイラ用の C++ 実行時および <code>iostream</code> ライブラリ

「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。

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。この章では、次の内容について説明します。

- 231 ページの「マルチスレッドアプリケーションを開発するための Solaris 環境の設定」
- 232 ページの「マルチスレッドアプリケーションのコンパイル」
- 236 ページの「マルチスレッドプログラムのデバッグ」

マルチスレッドアプリケーションを開発するための Solaris 環境の設定

Solaris OS 上でソフトウェアを構築するには、必要なツールを開発マシンにインストールする必要があります。Solaris OS にバンドルされている標準ツールを使用する場合も、Sun Studio ツールを使用する場合も、まず開発者環境に適した Solaris ソフトウェアをインストールする必要があります。Solaris Express Developer Edition リリースを使用している場合は、開発者環境がデフォルトでインストールされます。

Solaris 10 リリースの場合、Solaris OS には、開発者向けの適切な Solaris ソフトウェアグループをインストールすると使用できる、いくつかの開発者ソフトウェアパッケージが含まれています。インストールのために、Solaris OS は、Solaris パッケージのコレクションであるソフトウェアグループに論理的に分割されています。

Solaris OS をインストールする場合は、開発者パッケージが含まれている次のいずれかのソフトウェアグループを選択する必要があります。

- 開発者システムサポート
- 全体ディストリビューション
- 全体ディストリビューションと OEM サポート

これらのソフトウェアグループでは、GNU C コンパイラ (gcc) や Java コンパイラ (javac) などのコンパイラを入手できます。また、GNU ソースレベルデバッガ (gdb) とモジュールデバッガ (mdb)、リンカー (ld)、ソース管理ユーティリティ (sccs)、make などのビルドユーティリティもインストールされます。これらのファイルは、/usr/css および /usr/sfw にインストールされます。

注 - このマニュアルでは、Sun Studio C コンパイラを使用したマルチスレッドアプリケーションの構築について説明します。Sun Studio ソフトウェアは、Solaris Express Developer Edition リリースに含まれています。Solaris 10 リリースを使用している場合は、<http://developers.sun.com/sunstudio/> の Web サイトから Sun Studio ソフトウェアをダウンロードできます。

マルチスレッドアプリケーションのコンパイル

この節では、Sun Studio C コンパイラを使用してマルチスレッドプログラムをコンパイルする方法について説明します。Sun Studio C コンパイラは並列プログラミング向けに最適化されており、ほかの C コンパイラにない多くの機能を備えています。C コンパイラの詳細については、『[Sun Studio 12: C ユーザーズガイド](#)』を参照してください。

コンパイルの準備

アプリケーションは、Solaris スレッドの場合は <thread.h> を、POSIX スレッドの場合は <pthread.h> をインクルードする必要があります。使用している API に対応するファイルか、またはアプリケーションが両方のスレッドの API を使用している場合は両方のファイルをインクルードするようにしてください。詳細は、[pthread.h\(3HEAD\)](#) のマニュアルページを参照してください。アプリケーションはまた、<errno.h>、<limits.h>、<signal.h>、<unistd.h> ファイルもインクルードする必要があります。

スレッドの選択 — Solaris または POSIX

Solaris での pthread の実装は、Solaris スレッドと完全に互換性があります。同じアプリケーションで Solaris スレッドと pthread の両方を使用できます。各スレッド実装の相違点については、[pthreads\(5\)](#) のマニュアルページを参照してください。また、相違点については、このマニュアルの第 6 章「[Solaris スレッドを使ったプログラミング](#)」も参照してください。

スレッドの相違点の 1 つに、fork 関数の動作があります。

Solaris 9 リリースでは、`fork()` 関数の動作は、アプリケーションが POSIX スレッドライブラリにリンクされているかどうかによって決定されていました。`-lthread` (Solaris スレッド) にリンクされ、`-lpthread` (POSIX スレッド) にリンクされていない場合は、`fork()` によって、親プロセスのすべてのスレッドが子スレッド内に複製されました。アプリケーションが `-lpthread` にリンクされている場合は、`-lthread` にもリンクされかどうかには関係なく、`fork()` は `fork1()` と同じであり、呼び出しスレッドのみが複製されました。

Solaris 10 リリースからは、`forkall()` 関数の呼び出しによって、親プロセスのすべてのスレッドが子プロセス内に複製されるようになりました。`fork1()` の呼び出しによって、呼び出しスレッドのみが子プロセス内に複製されます。Solaris 10 リリースでは、`fork()` の呼び出しは `fork1()` の呼び出しと等価です。つまり、呼び出しスレッドのみが子プロセス内に複製されます。これが、`fork()` に対して POSIX で規定されている動作です。アプリケーションで、すべての `fork` セマンティクスを複製する必要がある場合は、`forkall()` を呼び出してください。

<thread.h> または <pthread.h>

インクルードファイル `<thread.h>` には、Solaris スレッドの関数の宣言が含まれています。Solaris スレッドの任意の関数を呼び出すには、プログラムで `<thread.h>` をインクルードする必要があります。このファイルを使用すると、以前の Solaris ソフトウェアのリリースと互換性のあるコンパイルされたコードを生成できます。

インクルードファイル `<pthread.h>` は `pthread` の関数の宣言を含んでおり、プログラムで `pthread` を使用する場合に必要になります。

アプリケーションで `<thread.h>` と `<pthread.h>` の両方をインクルードすることによって、同じアプリケーション内に Solaris スレッドと POSIX スレッドを混在させることができます。その場合は、リンクおよびコンパイル時に、`pthread` の API でリンクするための `-lpthread` フラグを指定する必要があります。

`-mt` を使用した場合は、Solaris スレッドの API が自動的にリンクされます。`-lthread` を明示的に指定する代わりに、常に `-mt` オプションを使用してください。`pthread` を使用するには、リンクのコマンド行で `-mt` オプションと `-lpthread` オプションを指定します。`libpthread` ライブラリによって `libthread` へのインタフェースが提供されるため、`pthread` を使用する場合も `libthread` が引き続き必要です。

マルチスレッドプログラムのコンパイルとリンク

Sun Studio C コンパイラ (`cc`) には、マルチスレッドコードのコンパイルとリンクを行うための `-mt` オプションが用意されています。`-mt` オプションは、ライブラリの適切な順序でのリンクを保証するものです。

-mt オプションは、一貫性をもって使用する必要があります。-mt でのコンパイルとリンクを別のステップで行う場合は、リンクステップとコンパイルステップで -mt オプションを使用する必要があります。1つの変換ユニットを -mt でコンパイルおよびリンクする場合は、そのプログラムのすべてのユニットを -mt でコンパイルおよびリンクする必要があります。

Solaris 10 リリースと Solaris 9 リリースのコンパイルとリンクには同じコマンドを使用できますが、次のいくつかの相違点を考慮するようにしてください。

- Solaris 10 リリースでは、すべてのスレッド関数が libc ライブラリに含まれているため、明示的に libthread または libpthread ライブラリにリンクする必要はありません。
- Solaris 9 リリースでは、Solaris スレッドの API と pthread の API が別のライブラリ libthread と libpthread に含まれています。Solaris 10 には、libc.so.1 に対するフィルタとして libthread.so と libpthread.so の共有オブジェクトが実装されているため、Solaris 9 で別のライブラリを使用してコンパイルされたアプリケーションでも動作するはずです。アプリケーションが明示的に libthread または libpthread ライブラリにリンクされている場合は、Solaris 10 でコンパイルされたアプリケーションは Solaris 9 でも動作します。-mt でリンクすると、アプリケーションを Solaris 9 で実行したとき、Solaris の fork() と fork1() の区別が保持されます。-lpthread オプションでリンクすると、fork() は、Solaris 9 以前のリリースでの Solaris fork1() の呼び出しと同じ動作をします。
- Solaris 9 リリースでは、アプリケーションが lthread または lpthread にリンクされていない場合、-libthread と -libpthread のすべての呼び出しが無効です。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。本当の手続きは、libc とスレッドライブラリ (libthread または libpthread) の両方がアプリケーションに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。
- Solaris 9 以降のリリースでは、スレッドを使用しないプログラムをリンクするときに、-mt、-lthread または -lpthread を指定しても、意味上の違いは発生しません。余分なスレッドや LWP が生成されることはありません。メインスレッド (唯一のスレッド) が、従来のシングルスレッドプロセスとして実行されます。プログラムへの唯一の影響は、システムライブラリのロックが空の関数の呼び出しではなく本当のロックになる点です。競合しないロックを獲得する必要があります。

POSIX スレッド環境でのコンパイルとリンク

アプリケーションで pthread のみを使用するか、または Solaris スレッドと pthread の両方を使用している場合は、次のコマンドを使用してコンパイルとリンクを行います。

```
cc -mt [flag ...] file... [library...] -lpthread
```

-mt オプションが libthread ライブラリにリンクするに対して、-lpthread オプションは libpthread ライブラリにリンクします。libpthread によって libthread へのインタフェースが提供されるため、pthread を使用する場合は両方のフラグが必要です。

-mt オプションは、コマンド行の任意の場所に指定できます。-lpthread オプションは、ユーザーライブラリのあとに指定するようにします。-mt と -lpthread の相対位置は関係ありません。

たとえば、次の行はすべて等価です。

```
cc -mt -o myprog f1.o f2.o -lmylib -lpthread
cc      -o myprog f1.o f2.o -mt -lmylib -lpthread
cc      -o myprog f1.o f2.o -lmylib -mt -lpthread
cc      -o myprog f1.o f2.o -lmylib -lpthread -mt
```

cc コマンドのオプションの詳細については、Sun Studio cc(1) のマニュアルページおよび『[Sun Studio 12: C ユーザーズガイド](#)』を参照してください。

Solaris スレッド環境でのコンパイルとリンク

Solaris スレッド環境では、アプリケーションのコンパイルとリンク時に、次のオプションを指定します。

アプリケーションで Solaris スレッドのみを使用している場合は、次のコマンドを使用してコンパイルとリンクを行います。

```
cc -mt [flag ...] file... [library...]
```

-mt オプションは、libthread ライブラリにリンクします。

cc コマンドのオプションの詳細については、Sun Studio cc(1) のマニュアルページおよび『[Sun Studio 12: C ユーザーズガイド](#)』を参照してください。

混在スレッド環境でのコンパイルとリンク

アプリケーションで pthread と Solaris スレッドの両方の関数を使用している場合は、pthread のみのコンパイルに使用したのと同じコマンドでコンパイルとリンクを行うことができます。

```
cc -mt [flag ...] file... [library...] -lpthread
```

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。

リンク時の **POSIX** セマフォ用 `-lrt` の指定

Solaris セマフォルーチン `sema_*(3C)` は、標準 C ライブラリに入っています。これに対して、[131 ページの「セマフォによる同期」](#)で説明されているように、標準 `sem_*(3RT)` POSIX セマフォルーチンを取得するには、リンク時に `-lrt` ライブラリを指定します。

代替スレッドライブラリ

Solaris 8 リリースでは、`/usr/lib/lwp` (32 ビット) および `/usr/lib/lwp/64` (64 ビット) ディレクトリに、代替スレッドライブラリが実装されました。Solaris 9 リリースでは、このライブラリが `/usr/lib` および `/usr/lib/64` ディレクトリに標準スレッドとして実装されています。Solaris 10 リリース以降では、すべてのスレッド機能が `libc` に移動し、独立したスレッドライブラリは不要になりました。`/usr/lib/lwp` ディレクトリは、Solaris 8 アプリケーションの互換性のために維持されています。

マルチスレッドプログラムのデバッグ

ここでは、マルチスレッドプログラミングのバグの原因となる特性について説明します。また、プログラムのデバッグに役立つユーティリティについても説明します。

マルチスレッドプログラムでよく起こるミス

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。

- ポインタを新しいスレッドの引数として呼び出しスタックへ渡す。
- 大域メモリー (変更が可能で、かつ共有されている状態) にアクセスするときに同期機構で保護していないために、「データの競合」が発生する。1つのプロセス内の2つ以上のスレッドが同じメモリー位置に同時にアクセスし、少なくとも1つのスレッドがそのメモリー位置に書き込もうとすると、データの競合が発生します。スレッドが排他的ロックを使用してそのメモリーへのアクセスを制御していない場合は、アクセスの順序が決定性ではなくなるため、その順序に応じて、実行するたびに異なる計算結果が生成されることがあります。問題にはならないデータの競合 (たとえば、ビジー状態の待機にメモリーアクセスが使用されている場合) もありえますが、データの競合の多くはプログラムのバグです。データの競合の検出には、Thread Analyzer ツールが有効です。[238 ページの「Thread Analyzer を使用したデータの競合およびデッドロックの検出」](#)を参照してください。
- 2つのスレッドが異なる順序で、同じ組の大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ます。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。Thread Analyzer ツールは、デッドロックの検出にも有効です。238 ページの「[Thread Analyzer を使用したデータの競合およびデッドロックの検出](#)」を参照してください。

- 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)。
- 同期保護において見えない間隙が生じている。この保護における間隙は、保護されているコードセグメントに、呼び出し側に復帰する前に同期機構をいったん解放し、再度獲得する関数が含まれている場合に発生します。この結果、問題が発生します。呼び出し側から見ると大域データは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で使用しているときに、非同期的なシグナルの処理に `sigwait(2)` モデルを使用しない。
- `setjmp(3C)` と `longjmp(3C)` 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()` または `*_cond_timedwait()` 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 `pthread_join(3C)` で再利用しなければならないことを忘れている。
`pthread_exit(3C)` は記憶領域を解放しません。
- 深い入れ子の再帰呼び出しを行ったり、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、問題が発生することがあります。
- スタックサイズ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。

マルチスレッドプログラムの動作は、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。この動作は、スレッドのスケジューリングの順序が異なる場合に発生します。

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、ブレイクポイントによるデバッグよりも追跡の方が有効です。

DTrace による追跡とデバッグ

DTrace は、Solaris OS に組み込まれている総合的な動的追跡ツールです。DTrace の機能を利用して、マルチスレッドプログラムの動作を検査できます。DTrace は、実行中のプログラムに、指定した実行パス内の場所でデータを収集するためのプローブを挿入します。収集されたデータを検証して問題の領域を特定することができます。DTrace の使用の詳細は、『[Solaris 動的トレースガイド](#)』および『[DTrace ユーザーガイド](#)』を参照してください。

[Sun Developers Network](#) の Web サイトには、[DTrace Quick Reference Guide](#) など、DTrace に関するいくつかの記事が含まれています。

Performance Analyzer によるプロファイリング

Sun Studio ソフトウェアに含まれている Performance Analyzer ツールを使用すると、マルチスレッドプログラムやシングルスレッドプログラムの広範囲なプロファイリングを行うことができます。このツールでは、指定した任意の時点でのスレッドの動作を詳細に表示できます。詳細は、[Sun Studio Web ページ](#) および [Sun Studio Information Center](#) を参照してください。

Thread Analyzer を使用したデータの競合およびデッドロックの検出

Sun Studio ソフトウェアには、Thread Analyzer と呼ばれるツールが含まれています。このツールを使用すると、マルチスレッドプログラムの実行を解析できます。このツールでは、Pthread の API、Solaris スレッドの API、OpenMP 指令、Sun 並列指令、Cray® 並列指令、またはこれらのテクノロジーの混在を使用して記述されたコードに含まれているデータの競合やデッドロックなどのマルチスレッドプログラミングのエラーを検出できます。

『[Sun Studio 12: スレッドアナライザユーザズガイド](#)』を参照してください。

dbx の使用

dbx ユーティリティは、<http://developers.sun.com/sunstudio/> から入手可能な Sun Studio 開発者ツールに含まれているデバッガです。Sun Studio dbx コマンド行デバッガでは、C、C++、および Fortran で記述された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 を使用するには、まず端末ウィンドウで起動し、dbx コマンドを使用してプログラムを対話的にデバッグします。グラフィカルインタフェースの方が望ましい場合は、Sun Studio IDE (統合開発環境) のデバッグウィンドウで dbx の同じ機能を使用できます。dbx の起動方法については、dbx(1) のマニュアルページを参照してください。『[Sun Studio 12: dbx コマンドによるデバッグ](#)』[Sun Studio 12: Debugging a Program With dbx](#) のマニュアルを参照してください。Sun Studio IDE のデバッグ機能は、IDE のオンラインヘルプで説明されています。

マルチスレッドプログラムのデバッグの詳細については、『[Sun Studio 12: dbx コマンドによるデバッグ](#)』の第 11 章「マルチスレッドアプリケーションのデバッグ」を参照してください。dbx デバッガは、『[Sun Studio 12: dbx コマンドによるデバッグ](#)』の付録 B「イベント管理」で説明されているスレッドイベントのイベントハンドラを操作するためのコマンドを提供します。

Table 8-1 に記載されているすべての 表 8-1 オプションは、マルチスレッドアプリケーションをサポートします。

表 8-1 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	動作
cont at line [-sig signo id]	<i>line</i> で指定した行から <i>signo</i> で指定したシグナルで実行を再開する。 <i>id</i> は実行を再開するスレッドまたは LWP を指定する。デフォルト値は <i>all</i> 。
lwp [<i>lwpid</i>]	現在の LWP を表示する。指定の LWP [<i>lwpid</i>] に切り替える。
lwps	現在のプロセスの、すべての LWP を一覧表示する。
next ... <i>tid</i>	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 ... <i>lwpid</i>	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。指定のスレッドが実行可能である LWP。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... <i>tid</i>	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... <i>lwpid</i>	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... <i>lwpid</i>	指定の LWP で機械命令をステップ実行する (呼び出しにステップインする)。
stepi... <i>tid</i>	指定のスレッドが実行可能である LWP で機械命令をステップ実行する。
thread [<i>tid</i>]	現在のスレッドを表示するか、またはスレッド <i>tid</i> に切り替える。以降のすべての変型で、 <i>tid</i> を省略すると現在のスレッドになる。
thread -info [<i>tid</i>]	指定のスレッドの全情報を表示する。

表 8-1 dbx のマルチスレッドプログラム用オプション (続き)

オプション	動作
<code>thread -blocks [tid]</code>	指定されたスレッドが保持しているロックで、ほかのスレッドをブロックしているすべてのロックを一覧表示する。
<code>thread -suspend [tid]</code>	指定のスレッドを停止状態にして、実行されないようにする。停止状態のスレッドの <code>threads</code> リストには「S」が表示される。
<code>thread -resume [tid]</code>	指定のスレッドの停止状態を解除して、実行が再開されるようにする。
<code>thread -hide [tid]</code>	指定のスレッド (または現在のスレッド) を非表示にする。このスレッドは、 <code>threads</code> オプションのリストには表示されない。
<code>thread -unhide [tid]</code>	指定のスレッド (または現在のスレッド) の非表示を解除する。
<code>thread -unhide all</code>	全スレッドの非表示を解除する。
<code>threads</code>	全スレッドを一覧表示する。
<code>threads -all</code>	通常は表示されないスレッド (ゾンビ) を表示する。
<code>threads -mode all filter</code>	<code>threads</code> 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。フィルタが有効な場合、 <code>thread -hide</code> コマンドで非表示になっているスレッドは表示されない。
<code>threads -mode auto manual</code>	スレッドリストの自動更新機能を有効にする。
<code>threads -mode</code>	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。

TNF ユーティリティーによる追跡とデバッグ

Dtrace、Performance Analyzer、Thread Analyzer、および dbx は比較的最近のツールですが、アプリケーションやライブラリからの性能解析情報の収集、追跡、デバッグに以前の TNF ユーティリティーを引き続き使用することもできます。TNF ユーティリティーは、カーネルおよび複数のユーザープロセスとスレッドからの追跡情報を集約します。TNF ユーティリティーは従来から Solaris ソフトウェアに付属しています。これらのユーティリティーについては、`tracing(3TNF)` のマニュアルページを参照してください。

truss の使用

システム呼び出し、シグナル、およびユーザーレベル関数呼び出しの追跡方法については、[truss\(1\)](#)のマニュアルページを参照してください。

mdb の使用

mdb については、『[Solaris モジューラデバグ](#)』を参照してください。

次の mdb コマンドを使って、マルチスレッドプログラムの LWP にアクセスできます。

<code>\$l</code>	ターゲットがユーザープロセスである場合、代表スレッドの LWP ID を出力する。
<code>\$L</code>	ターゲットがユーザープロセスである場合、ターゲット内にある各 LWP の LWP ID を出力する。
<code>pid::attach</code>	<code>pid</code> で指定したプロセスに接続する。
<code>::release</code>	以前に追加されたプロセスまたはコアファイルを解放する。このようなプロセスを継続するには <code>prun(1)</code> を使用し、再開するには MDB などのデバグを適用する。

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。

`[ addr ] ::bp [+/-dDestT] [-c cmd] [-n count] sym ...`
指定された場所にブレークポイントを設定します。

`addr ::delete [ id | all ]`
指定された ID 番号のイベント指定子を削除します。

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、ユーティリティが異なる場合は、その動作を明記します。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。この章の内容は次のとおりです。

- 243 ページの「大域変数の考慮」
- 245 ページの「静的局所変数の利用」
- 245 ページの「スレッドの同期」
- 249 ページの「デッドロックの回避」
- 251 ページの「マルチスレッドコードの基本的な指針」
- 252 ページの「スレッドの生成と使用」
- 252 ページの「マルチプロセッサへの対応」
- 257 ページの「スレッドプログラムの例」

大域変数の考慮

従来のほとんどのコードは、シングルスレッドプログラム用に設計されています。このコード設計は、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によく見られます。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。

- 大域変数に書き込んだ内容をしばらく経ってから読み取った場合、その内容は以前と同じである
- 大域的でない静的記憶領域に書き込んだ内容をしばらく経ってから読み取った場合も、その内容は以前と同じである
- 変数への並行アクセスは起こらないので、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例を示します。

従来のシングルスレッドのCおよびUNIXでは、システムコールで検出されたエラーの処理につい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返すことができます。たとえば、`write()` は、転送されたバイト数を返します。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 大域変数と `errno`

```
extern int errno;
...
if (write(file_desc, buffer, size) == -1) {
    /* the system call failed */
    fprintf(stderr, "something went wrong, "
        "error code = %d\n", errno);
    exit(1);
}
...
```

通常の戻り値と混同されがちですが、実際のエラーコードは返されず、大域変数 `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、大域変数による方法は使用できません。

スレッドは、この問題を、新しい概念の記憶領域クラスであるスレッド固有データを使用することで解決します。この記憶領域は大域記憶領域と似ています。スレッド固有データは、スレッドの実行が可能な任意の手続きからアクセスできます。ただし、スレッド固有データは特定のスレッド専用です。2つのスレッドが同じ名前のスレッド固有データを参照した場合、これらのスレッドはそれぞれ異なる記憶領域を参照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。この実装では、`errno` をマクロにして関数呼び出しを拡張することで、スレッド固有のものとして `errno` を参照できるようになっています。

静的局所変数の利用

例 9-2 は、`errno` と同様の問題を示します。ただし、ここでは大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(3NSL)` 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;
    /* Lookup name in hosts database */
    /* Put answer in result */
    return(&result);
}
```

一般に、局所変数へのポインタを返すのはよい方法ではありません。この例では、変数が静的なため、ポインタは正常に動作します。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として、`errno` の問題の場合と同様にスレッド固有データを使用できます。しかし、この方法では、動的に記憶領域が割り当てられるため、呼び出しの負荷が高くなります。

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。このルーチンに出力引数を1つ追加して、呼び出し側から記憶領域を提供できるようにします。出力引数を追加するには、`gethostbyname()` 関数に新しいインタフェースが必要です。

この方法は、上記のような問題の多くを解決するために使用されています。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(3NSL)` は、`gethostbyname_r(3NSL)` 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。

複数のスレッドが、オブジェクトを操作する関数を呼び出すと、問題が発生します。シングルスレッドの場合、こうしたオブジェクトへのアクセスの同期は問題にはなりません。しかし、例 9-3 のように、マルチスレッドコードでの同期については考慮が必要です。`printf(3S)` 関数は、マルチスレッドプログラムの呼び出しに対して安全です。この例では、`printf()` がマルチスレッドに対して安全で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

```
/* thread 1: */
printf("go to statement reached");

/* thread 2: */
printf("hello world");

printed on display:
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であり、この方法にはほとんど利点がありません。

再入可能な関数

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。再入可能な関数は、複数のスレッドに同時に呼び出されても、正常に動作します。再入可能な関数を作成するときには、その関数にとって何が正しい動作なのかを把握しておく必要があります。

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。再入可能な関数にするためには、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。

再入可能の問題は、メモリーやファイルなど、大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、大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再入可能な関数にする方法として、コードロックとデータロックの2つがあります。

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、「モニター」という構造が用意されています。モニターは、そのスコープに定義されている関数に対して、暗黙的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、モニター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ーのスコープにある関数は、互いに不可分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 (大域的な) 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。複数のスレッドが個別のデータ集合を操作する場合には、1つのモジュール内で複数のスレッドを実行できます。特に、複数読み取り単一書き込みの場合には、1つのデータ集合に対するスレッドの競合は発生しません。つまり、通常はコードロックよりもデータロックの方が、多重度を高くすることができます。

プログラムで `mutex`、条件変数、セマフォなどのロックを使用するときに、どのような方法を使用すればよいかを説明します。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がよいでしょうか。それとも、ロックと解除に伴うオーバー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ほうがよいでしょうか。

ロックをどの程度きめ細かくかけるべきかは、ロックによって保護されるデータの量によって異なります。もっと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ーヘッドは、アプリケーション内のロックの数が多いと無視できなくなるからです。

一般には、きめの粗いロックから始めてボトルネックを特定し、その軽減のために必要な場所に、きめの細かいロックを追加します。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ー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、ユーザーが判断してください。

不変式とロック

コードロックでもデータロックでも、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(3C) による不変式のテスト

```
mutex_lock(&lock);
while((condition)==FALSE)
    cond_wait(&cv,&lock);
assert((condition)==TRUE);
.
.
.
mutex_unlock(&lock);
```

この `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。

ほかの例は、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の各項目で、直前の項目の前向きポインタは不変式のよい例になります。この前向きポインタは、直後の項目の後ろ向きポインタと同じ要素を指します。

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つの大域的な相互排他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操作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」(「再帰的なデッドロック」)です。自己デッドロック(再帰的なデッドロック)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ーについて考えてみましょう。このモジュール関数は、呼び出しの間 `mutex` ロックを保持します。この場合、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ると、たちまちデッドロックが発生します。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が、元のモジュールに依存している可能性がある場合、この関数を呼び出さないようにします。特に、不変式を再設定しないで再び元のモジュールを呼び出すような関数を呼び出さないことです。さらに、呼び出しの前にすべてのモジュールロックを解除するようにします。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。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、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。何も変化しない状態になります。つまり、これらのスレッドが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、ロックを獲得する順序、「ロック階層」を設定します。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、`mutex A` のロックを獲得したあとにもう一度 `mutex B` のロックを獲得すると、先の仮定が無効になります。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プリミティブが用意されています。一例として、

`pthread_mutex_trylock()` があります。このプリミティブの動作によって、競合が発生していないときには、スレッドによるロック階層違反が許可されます。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

ロックが獲得される順序が保証されていないため、特定のスレッドがロックを獲得できないという問題が発生します。

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、ほかのスレッドがロックを獲得したように見えます。しかし、ロックを保持していたスレッドはブロックされません。その結果、このスレッドは、ロックを解除してからロックを再度獲得する間も引き続き実行されます。この場合、ほか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、ロックを再度獲得する呼び出しの直前に `sched_yield(3C)()` を呼び出します。`sched_yield()` 関数は、その他のスレッドに実行とロックの獲得を許可します。

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、システムでは特に制限していません。`sched_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

ロックするときには、次の簡単な指針に従ってください。

- 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。したがって、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。

デッドロックの検出

Sun Studio Thread Analyzer は、プログラム内のデッドロックを検出するために使用できるツールです。Thread Analyzer は、実際のデッドロックだけでなく、潜在的なデッドロックも検出できます。潜在的なデッドロックは、特定の実行で常に発生するわけではありませんが、スレッドのスケジューリングやスレッドによるロック要求のタイミングによっては、プログラムの任意の実行で発生する可能性があります。実際のデッドロックは、プログラムの実行中に発生して、関連するスレッドをハングアップさせますが、これによってプロセス全体をハングアップさせるかどうかは場合によって異なります。

『Sun Studio 12: スレッドアナライザユーザーズガイド』を参照してください。

マルチスレッドコードの基本的な指針

- 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。
この方法で安全でないコードを参照すれば、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が保証されます。
- ライブラリを「MT-安全」にする場合、プロセスの大域的な操作はスレッド化しないでください。
大域的な操作 (または大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、次のようにします。

```
pthread_exit();
/*NOTREACHED*/
```

- 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い場合は、「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解消してください。
詳細は、第8章「コンパイルとデバッグ」を参照してください。

- `-mt` を使用すると、マルチスレッドが有効になります。
- `-mt` オプションで使用される `-lpthreads` は、POSIX スレッドの関数にリンクします。このフラグは、プログラムで `pthread` の関数を使用する場合にのみ必要です。
- `-mt` を使用した場合は、Solaris スレッドの API が自動的にリンクされます。`-lthread` を明示的に指定する代わりに、常に `-mt` オプションを使用してください。`libpthread` ライブラリによって `libthread` へのインタフェースが提供されるため、`pthread` を使用する場合も `libthread` が引き続き必要です。

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およびスタックをキャッシュするので、スレッドを繰り返し生成しても負荷は大きくありません。ただし、必要に応じてスレッドを生成したり削除したりする方が、専用の処理要求を待つスレッドを維持管理するより負荷が大きくなります。たとえば、RPC サーバーがよい例です。RPC サーバー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。

スレッドの生成にかかるオーバーヘッドは、プロセスの生成にかかるオーバーヘッドほど高くありません。しかし、スレッドの生成は、いくつかの命令を生成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。

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 (マルチコアプロセッサやマルチスレッドプロセッサを含む) 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。

注- このマニュアルでは、特に断りのない限り、マルチプロセッサの説明はマルチコアプロセッサとマルチスレッドプロセッサにも該当します。

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、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、ほか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。

共有される大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。

メモリー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。

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-共有メモリーの場所へのアクセスに常にスレッド同期プリミティブを使用する場合、ここで言及した内容は重要ではありません。第4章「同期オブジェクトを使ったプログラミング」を参照してください。

アーキテクチャー

スレッドは、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を同期します。スレッドを同期することによって、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は、単一プロセッサ上でプログラムを実行することと同じこととなります。

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に示すように、こうしたトリックは危険性をはらんでいます。

通常マルチプロセッサアーキテクチャー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。

- プログラムを実行するプロセッサ(コアとそれらのスレッドを含む)
- 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。

キャッシュを使って、平均メモリーアクセス時間を高速化できます。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;

/* Thread 1 (producer) */
void
producer(char item) {
    do
    {
        /* nothing */
    }
    while
        ((in - out) == BSIZE);

    buffer[in%BSIZE] = item;
    in++;
}

/* Thread 2 (consumer) */
char
consumer(void) {
    char item;
    do
    {
        /* nothing */
    }
    while
        ((in - out) == 0);

    item = buffer[out%BSIZE];
    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新しい項目のための空きができるまで、この差を繰り返し計算しながら待ちます。消費者は、バッファに項目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では、一方のプロセッサへの変更が、もう一方のプロセッサのメモリーにただちに伝えられます。強く順序付けられたメモリーでは、in と out が最終的にオーバーフローしても、このソリューションが成立します。このオーバーフローは、BSIZE が1ワードで表現できる最大の整数より小さい場合に発生します。

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、ほか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、異なったメモ

リー領域2箇所に変更が加えられたとき、どうなるか考えてみましょう。もう一方のプロセッサには、メモリーの変更がただちに伝わらないので、こ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、プログラムの整合性を保証するために、これらのストアバッファをチェックします。しかし、ほかのプロセッサから、このストアバッファは参照できません。このため、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、ほかのプロセッサから参照できません。

同期プリミティブは、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。[第4章「同期オブジェクトを使ったプログラミング」](#)を参照してください。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](#)で問題が発生します。消費者は、対応するバッファースロットへの変更を知る前に、生産者によってinが1つ増やされたことを知る場合があります。

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、このメモリーの順序付けを「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。

最近では、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、プログラムは大域データや共有データをロックで保護するよう注意する必要があります。

[例9-5](#)と[例9-6](#)に示すように、ロックは重要です。

Peterson のアルゴリズム

[例9-6](#)のコードは、2つのスレッド間の相互排他を処理する Peterson のアルゴリズムの実装例です。このコードは、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。スレッドは、`mut_excl()` を呼び出すと、「すばやく」危険領域に入ります。

ここで、スレッドは危険領域に入るとすばや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

```
void mut_excl(int me /* 0 or 1 */) {
    static int loser;
    static int interested[2] = {0, 0};
    int other; /* local variable */
```

例9-6 2つのスレッド間での相互排他 (続き)

```
other = 1 - me;
interested[me] = 1;
loser = me;
while (loser == me && interested[other])
    ;

/* critical section */
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。

SPARC ベースのマルチプロセッサを含む一部のマルチプロセッサは、ストアバッファを装備しています。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せんが、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ます。しかし、「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 つのスレッドが両方とも危険領域に入ります。各スレッドは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、あとになって発生する可能性があります。

この問題を回避するには、スレッドの同期プリミティブを使用します。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ます。第4章「同期オブジェクトを使ったプログラミング」を参照してください。

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、ほかの部分
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。このア
ルゴリズムでは、バリアー同期を使用して並列的な部分と逐次的な部分を調整でき
ます。

表 9-1 バリアー同期を使用したマルチスレッドの調整

逐次実行	並列実行
Thread 1	Thread 2 through Thread n
<pre>while(many_iterations) { sequential_computation --- Barrier --- parallel_computation }</pre>	<pre>while(many_iterations) { --- Barrier --- parallel_computation }</pre>

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ます。次に、これらの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がな
い点にあります。しかし、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、すべてのスレッドの同期をとる必要があります。

バリアーには、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ー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
部がバリアーに達したところで解放され、一斉に計算を開始します。

スレッドプログラムの例

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「拡張の例: スレッドプールの実装」](#)では、これまでに
紹介した機能や方法の多くを使用した pthread プログラムの例を掲載しています。

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場合は、『Programming with
Threads』(Steve Kleiman、Devang Shah、Bart Smaalders 共著、Prentice-Hall 発行、1995
年)を参照してください。この書籍には Solaris OS の変更が反映されていませんが、
その概念的な情報の多くは引き続き有効です。

拡張の例: スレッドプールの実装

この付録は、マルチスレッドプログラミングのインタフェースの有用なパッケージであるスレッドプールの実装例を示しています。

- [259 ページの「スレッドプールについて」](#)
- [260 ページの「スレッドプールの例について」](#)
- [272 ページの「スレッドプールの例が示している内容」](#)

スレッドプールについて

スレッドは、アプリケーションで多くの処理を一度に実行するための有効なパラダイムを提供します。実行すべきことがある場合は、その実行のためのスレッドを作成してください。スレッドを使用すると、アプリケーションのロジックを簡略化したり、複数のプロセッサを利用したりすることが可能ですが、作成するスレッドの数が多すぎると、リソースの競合によって全体的なアプリケーション性能の問題が発生することがあります。アプリケーションが、リソースの競合(たとえば、相互排他ロックの処理)のためにその時間の多くを消費して、有効な作業を実際に行う時間が少なくなる可能性があります。また、プロセスの作成より少ないとは言え、スレッドの作成も負荷になります。少量の作業を実行するためにスレッドを作成することは無駄です。

スレッドプールは、要求に応じて作業を実行する一連の匿名スレッドを管理します。スレッドは、すぐには終了しません。いずれかのスレッドがタスクを完了すると、そのスレッドはアイドル状態になり、別のタスクにディスパッチされるように準備されます。スレッドの作成と削除のオーバーヘッドは、プール内のアクティブなワークスレッドの数だけの作成と削除に制限されます。アプリケーションにはワークスレッドの数より多くのタスクが存在する場合があります、これが通常の状態です。リソースの競合を軽減することによって、プロセッサの利用率やスループットが改善されます。送信されたタスクは順番に、通常、タスクごとにスレッドを作成することによって実行される場合より高速に処理されます。

スレッドプールの例について

- [260 ページの「スレッドプールの関数」](#)
- [261 ページの「スレッドプールのコード例」](#)

スレッドプールの関数

thr_pool.h ヘッダーファイルは、次の関数インタフェースを宣言します。

- [260 ページの「thr_pool_create\(\)」](#)
- [260 ページの「thr_pool_queue\(\)」](#)
- [261 ページの「thr_pool_wait\(\)」](#)
- [261 ページの「thr_pool_destroy\(\)」](#)

thr_pool_create()

スレッドプールを作成します。複数のプールを作成できます。

```
typedef struct thr_pool thr_pool_t; /* クライアントには見えない */
```

```
thr_pool_t *thr_pool_create(uint_t min_threads, uint_t max_threads,  
                           uint_t linger, pthread_attr_t *attr);
```

min_threads プール内のスレッドの最小数。

max_threads プール内のスレッドの最大数。

linger タスクが入力されないときに、アイドル状態のスレッドが終了前に
 存続できる秒数。アイドル状態のスレッドが終了できるのは、ス
 レッドの最小数を超える余分なスレッドが存在する場合だけです。

attr すべてのワークスレッドの属性。これは NULL の場合があります。

エラーが発生した場合、thr_pool_create() は、errno にエラーコードを設定して
NULL を返します。

thr_pool_queue()

作業要求またはタスクをスレッドプールのジョブ待ち行列に入れます。

```
int thr_pool_queue(thr_pool_t *pool, void *(*func)(void *), void *arg);
```

pool thr_pool_create() から返されたスレッドプール識別子。

func 呼び出されるタスク関数。

arg タスク関数に渡される唯一の引数。

エラーが発生した場合、`thr_pool_queue()` は、`errno` にエラーコードを設定して -1 を返します。

`func` 引数と `arg` 引数が、31 ページの「[pthread_create の構文](#)」に示されている `pthread_create()` の `start_routine` 引数と `arg` 引数に似ていることに注意してください。`thr_pool_queue()` 関数は、既存のアプリケーション内の `pthread_create()` の置き換えとして使用できます。ただし、`pthread_create()` の代わりに `thr_pool_queue()` を使用する場合は、`pthread_join()` を使用してタスクの完了を待つことはできません。

`thr_pool_wait()`

スレッドプール内で待ち行列に入れられたすべてのジョブの完了を待ちます。

```
void thr_pool_wait(thr_pool_t *pool);
```

`pool` は、`thr_pool_create()` から返されたスレッドプール識別子です。

`thr_pool_destroy()`

待ち行列に入れられたすべてのジョブを取り消し、プールを削除します。アクティブにタスクを処理しているワークスレッドは取り消されます。

```
extern void thr_pool_destroy(thr_pool_t *pool);
```

`pool` は、`thr_pool_create()` から返されたスレッドプール識別子です。

スレッドプールのコード例

この節では、スレッドプールのコード例を示します。

- [261 ページの「thr_pool.h ファイル」](#)
- [263 ページの「thr_pool.c ファイル」](#)

`thr_pool.h` ファイル

このファイルは、この例で使用されている関数を宣言します。

例 A-1 `thr_pool.h`

```
/*
 * Declarations for the clients of a thread pool.
 */

#include <pthread.h>
```

例A-1 thr_pool.h (続き)

```
/*
 * The thr_pool_t type is opaque to the client.
 * It is created by thr_pool_create() and must be passed
 * unmodified to the remainder of the interfaces.
 */
typedef struct thr_pool thr_pool_t;

/*
 * Create a thread pool.
 * min_threads: the minimum number of threads kept in the pool,
 *               always available to perform work requests.
 * max_threads: the maximum number of threads that can be
 *               in the pool, performing work requests.
 * linger:      the number of seconds excess idle worker threads
 *               (greater than min_threads) linger before exiting.
 * attr:        attributes of all worker threads (can be NULL);
 *               can be destroyed after calling thr_pool_create().
 * On error, thr_pool_create() returns NULL with errno set to the error code.
 */
extern thr_pool_t *thr_pool_create(uint_t min_threads, uint_t max_threads,
                                   uint_t linger, pthread_attr_t *attr);

/*
 * Enqueue a work request to the thread pool job queue.
 * If there are idle worker threads, awaken one to perform the job.
 * Else if the maximum number of workers has not been reached,
 * create a new worker thread to perform the job.
 * Else just return after adding the job to the queue;
 * an existing worker thread will perform the job when
 * it finishes the job it is currently performing.
 *
 * The job is performed as if a new detached thread were created for it:
 * pthread_create(NULL, attr, void *(*func)(void *), void *arg);
 *
 * On error, thr_pool_queue() returns -1 with errno set to the error code.
 */
extern int thr_pool_queue(thr_pool_t *pool,
                          void *(*func)(void *), void *arg);

/*
 * Wait for all queued jobs to complete.
 */
extern void thr_pool_wait(thr_pool_t *pool);
```

例A-1 thr_pool.h (続き)

```
/*
 * Cancel all queued jobs and destroy the pool.
 */
extern void thr_pool_destroy(thr_pool_t *pool);
```

thr_pool.c ファイル

このファイルは、スレッドプールを実装します。

例A-2 thr_pool.c

```
/*
 * Thread pool implementation.
 * See <thr_pool.h> for interface declarations.
 */

#if !defined(_REENTRANT)
#define _REENTRANT
#endif

#include "thr_pool.h"
#include <stdlib.h>
#include <signal.h>
#include <errno.h>

/*
 * FIFO queued job
 */
typedef struct job job_t;
struct job {
    job_t *job_next; /* linked list of jobs */
    void (*job_func)(void *); /* function to call */
    void *job_arg; /* its argument */
};

/*
 * List of active worker threads, linked through their stacks.
 */
typedef struct active active_t;
struct active {
    active_t *active_next; /* linked list of threads */
    pthread_t active_tid; /* active thread id */
};

/*
```

例A-2 thr_pool.c (続き)

```

    * The thread pool, opaque to the clients.
    */
    struct thr_pool {
        thr_pool_t    *pool_forw;    /* circular linked list */
        thr_pool_t    *pool_back;    /* of all thread pools */
        pthread_mutex_t pool_mutex;   /* protects the pool data */
        pthread_cond_t pool_busycv;   /* synchronization in pool_queue */
        pthread_cond_t pool_workcv;   /* synchronization with workers */
        pthread_cond_t pool_waitcv;   /* synchronization in pool_wait() */
        active_t      *pool_active;   /* list of threads performing work */
        job_t          *pool_head;    /* head of FIFO job queue */
        job_t          *pool_tail;    /* tail of FIFO job queue */
        pthread_attr_t pool_attr;     /* attributes of the workers */
        int             pool_flags;    /* see below */
        uint_t          pool_linger;   /* seconds before idle workers exit */
        int             pool_minimum;  /* minimum number of worker threads */
        int             pool_maximum;  /* maximum number of worker threads */
        int             pool_nthreads; /* current number of worker threads */
        int             pool_idle;     /* number of idle workers */
    };

    /* pool_flags */
    #define POOL_WAIT    0x01    /* waiting in thr_pool_wait() */
    #define POOL_DESTROY 0x02    /* pool is being destroyed */

    /* the list of all created and not yet destroyed thread pools */
    static thr_pool_t *thr_pools = NULL;

    /* protects thr_pools */
    static pthread_mutex_t thr_pool_lock = PTHREAD_MUTEX_INITIALIZER;

    /* set of all signals */
    static sigset_t fillset;

    static void *worker_thread(void *);

    static int
    create_worker(thr_pool_t *pool)
    {
        sigset_t oset;
        int error;

        (void) pthread_sigmask(SIG_SETMASK, &fillset, &oset);
        error = pthread_create(NULL, &pool->pool_attr, worker_thread, pool);
        (void) pthread_sigmask(SIG_SETMASK, &oset, NULL);
    }

```

例A-2 thr_pool.c (続き)

```

        return (error);
    }

/*
 * Worker thread is terminating. Possible reasons:
 * - excess idle thread is terminating because there is no work.
 * - thread was cancelled (pool is being destroyed).
 * - the job function called pthread_exit().
 * In the last case, create another worker thread
 * if necessary to keep the pool populated.
 */
static void
worker_cleanup(thr_pool_t *pool)
{
    --pool->pool_nthreads;
    if (pool->pool_flags & POOL_DESTROY) {
        if (pool->pool_nthreads == 0)
            (void) pthread_cond_broadcast(&pool->pool_busycv);
    } else if (pool->pool_head != NULL &&
        pool->pool_nthreads < pool->pool_maximum &&
        create_worker(pool) == 0) {
        pool->pool_nthreads++;
    }
    (void) pthread_mutex_unlock(&pool->pool_mutex);
}

static void
notify_waiters(thr_pool_t *pool)
{
    if (pool->pool_head == NULL && pool->pool_active == NULL) {
        pool->pool_flags &= ~POOL_WAIT;
        (void) pthread_cond_broadcast(&pool->pool_waitcv);
    }
}

/*
 * Called by a worker thread on return from a job.
 */
static void
job_cleanup(thr_pool_t *pool)
{
    pthread_t my_tid = pthread_self();
    active_t *activep;
    active_t **activepp;

```

例A-2 thr_pool.c (続き)

```
(void) pthread_mutex_lock(&pool->pool_mutex);
for (activepp = &pool->pool_active;
    (activep = *activepp) != NULL;
    activepp = &activep->active_next) {
    if (activep->active_tid == my_tid) {
        *activepp = activep->active_next;
        break;
    }
}
if (pool->pool_flags & POOL_WAIT)
    notify_waiters(pool);
}

static void *
worker_thread(void *arg)
{
    thr_pool_t *pool = (thr_pool_t *)arg;
    int timedout;
    job_t *job;
    void *(*func)(void *);
    active_t active;
    timestruc_t ts;

    /*
     * This is the worker's main loop. It will only be left
     * if a timeout occurs or if the pool is being destroyed.
     */
    (void) pthread_mutex_lock(&pool->pool_mutex);
    pthread_cleanup_push(worker_cleanup, pool);
    active.active_tid = pthread_self();
    for (;;) {
        /*
         * We don't know what this thread was doing during
         * its last job, so we reset its signal mask and
         * cancellation state back to the initial values.
         */
        (void) pthread_sigmask(SIG_SETMASK, &fillset, NULL);
        (void) pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
        (void) pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);

        timedout = 0;
        pool->pool_idle++;
        if (pool->pool_flags & POOL_WAIT)
            notify_waiters(pool);
        while (pool->pool_head == NULL &&
```

例A-2 thr_pool.c (続き)

```

        !(pool->pool_flags & POOL_DESTROY)) {
    if (pool->pool_nthreads <= pool->pool_minimum) {
        (void) pthread_cond_wait(&pool->pool_workcv,
                                &pool->pool_mutex);
    } else {
        (void) clock_gettime(CLOCK_REALTIME, &ts);
        ts.tv_sec += pool->pool_linger;
        if (pool->pool_linger == 0 ||
            pthread_cond_timedwait(&pool->pool_workcv,
                                   &pool->pool_mutex, &ts) == ETIMEDOUT) {
            timedout = 1;
            break;
        }
    }
}
pool->pool_idle--;
if (pool->pool_flags & POOL_DESTROY)
    break;
if ((job = pool->pool_head) != NULL) {
    timedout = 0;
    func = job->job_func;
    arg = job->job_arg;
    pool->pool_head = job->job_next;
    if (job == pool->pool_tail)
        pool->pool_tail = NULL;
    active.active_next = pool->pool_active;
    pool->pool_active = &active;
    (void) pthread_mutex_unlock(&pool->pool_mutex);
    pthread_cleanup_push(job_cleanup, pool);
    free(job);
    /*
     * Call the specified job function.
     */
    (void) func(arg);
    /*
     * If the job function calls pthread_exit(), the thread
     * calls job_cleanup(pool) and worker_cleanup(pool);
     * the integrity of the pool is thereby maintained.
     */
    pthread_cleanup_pop(1);    /* job_cleanup(pool) */
}
if (timedout && pool->pool_nthreads > pool->pool_minimum) {
    /*
     * We timed out and there is no work to be done
     * and the number of workers exceeds the minimum.

```

例A-2 thr_pool.c (続き)

```
        * Exit now to reduce the size of the pool.
        */
        break;
    }
}
pthread_cleanup_pop(1);    /* worker_cleanup(pool) */
return (NULL);
}

static void
clone_attributes(pthread_attr_t *new_attr, pthread_attr_t *old_attr)
{
    struct sched_param param;
    void *addr;
    size_t size;
    int value;

    (void) pthread_attr_init(new_attr);

    if (old_attr != NULL) {
        (void) pthread_attr_getstack(old_attr, &addr, &size);
        /* don't allow a non-NULL thread stack address */
        (void) pthread_attr_setstack(new_attr, NULL, size);

        (void) pthread_attr_getscope(old_attr, &value);
        (void) pthread_attr_setscope(new_attr, value);

        (void) pthread_attr_getinheritsched(old_attr, &value);
        (void) pthread_attr_setinheritsched(new_attr, value);

        (void) pthread_attr_getschedpolicy(old_attr, &value);
        (void) pthread_attr_setschedpolicy(new_attr, value);

        (void) pthread_attr_getschedparam(old_attr, &param);
        (void) pthread_attr_setschedparam(new_attr, &param);

        (void) pthread_attr_getguardsize(old_attr, &size);
        (void) pthread_attr_setguardsize(new_attr, size);
    }

    /* make all pool threads be detached threads */
    (void) pthread_attr_setdetachstate(new_attr, PTHREAD_CREATE_DETACHED);
}

thr_pool_t *
```

例A-2 thr_pool.c (続き)

```

thr_pool_create(uint_t min_threads, uint_t max_threads, uint_t linger,
pthread_attr_t *attr)
{
    thr_pool_t    *pool;

    (void) sigfillset(&fillset);

    if (min_threads > max_threads || max_threads < 1) {
        errno = EINVAL;
        return (NULL);
    }

    if ((pool = malloc(sizeof (*pool))) == NULL) {
        errno = ENOMEM;
        return (NULL);
    }

    (void) pthread_mutex_init(&pool->pool_mutex, NULL);
    (void) pthread_cond_init(&pool->pool_busycv, NULL);
    (void) pthread_cond_init(&pool->pool_workcv, NULL);
    (void) pthread_cond_init(&pool->pool_waitcv, NULL);
    pool->pool_active = NULL;
    pool->pool_head = NULL;
    pool->pool_tail = NULL;
    pool->pool_flags = 0;
    pool->pool_linger = linger;
    pool->pool_minimum = min_threads;
    pool->pool_maximum = max_threads;
    pool->pool_nthreads = 0;
    pool->pool_idle = 0;

    /*
     * We cannot just copy the attribute pointer.
     * We need to initialize a new pthread_attr_t structure using
     * the values from the caller-supplied attribute structure.
     * If the attribute pointer is NULL, we need to initialize
     * the new pthread_attr_t structure with default values.
     */
    clone_attributes(&pool->pool_attr, attr);

    /* insert into the global list of all thread pools */
    (void) pthread_mutex_lock(&thr_pool_lock);
    if (thr_pools == NULL) {
        pool->pool_forw = pool;
        pool->pool_back = pool;
        thr_pools = pool;
    }
}

```

例A-2 thr_pool.c (続き)

```
    } else {
        thr_pools->pool_back->pool_forw = pool;
        pool->pool_forw = thr_pools;
        pool->pool_back = thr_pools->pool_back;
        thr_pools->pool_back = pool;
    }
    (void) pthread_mutex_unlock(&thr_pool_lock);

    return (pool);
}

int
thr_pool_queue(thr_pool_t *pool, void *(*func)(void *), void *arg)
{
    job_t *job;

    if ((job = malloc(sizeof (*job))) == NULL) {
        errno = ENOMEM;
        return (-1);
    }
    job->job_next = NULL;
    job->job_func = func;
    job->job_arg = arg;

    (void) pthread_mutex_lock(&pool->pool_mutex);

    if (pool->pool_head == NULL)
        pool->pool_head = job;
    else
        pool->pool_tail->job_next = job;
    pool->pool_tail = job;

    if (pool->pool_idle > 0)
        (void) pthread_cond_signal(&pool->pool_workcv);
    else if (pool->pool_nthreads < pool->pool_maximum &&
        create_worker(pool) == 0)
        pool->pool_nthreads++;

    (void) pthread_mutex_unlock(&pool->pool_mutex);
    return (0);
}

void
thr_pool_wait(thr_pool_t *pool)
{

```

例A-2 thr_pool.c (続き)

```

        (void) pthread_mutex_lock(&pool->pool_mutex);
        pthread_cleanup_push(pthread_mutex_unlock, &pool->pool_mutex);
        while (pool->pool_head != NULL || pool->pool_active != NULL) {
            pool->pool_flags |= POOL_WAIT;
            (void) pthread_cond_wait(&pool->pool_waitcv, &pool->pool_mutex);
        }
        pthread_cleanup_pop(1);    /* pthread_mutex_unlock(&pool->pool_mutex); */
    }

void
thr_pool_destroy(thr_pool_t *pool)
{
    active_t *activep;
    job_t *job;

    (void) pthread_mutex_lock(&pool->pool_mutex);
    pthread_cleanup_push(pthread_mutex_unlock, &pool->pool_mutex);

    /* mark the pool as being destroyed; wakeup idle workers */
    pool->pool_flags |= POOL_DESTROY;
    (void) pthread_cond_broadcast(&pool->pool_workcv);

    /* cancel all active workers */
    for (activep = pool->pool_active;
         activep != NULL;
         activep = activep->active_next)
        (void) pthread_cancel(activep->active_tid);

    /* wait for all active workers to finish */
    while (pool->pool_active != NULL) {
        pool->pool_flags |= POOL_WAIT;
        (void) pthread_cond_wait(&pool->pool_waitcv, &pool->pool_mutex);
    }

    /* the last worker to terminate will wake us up */
    while (pool->pool_nthreads != 0)
        (void) pthread_cond_wait(&pool->pool_busycv, &pool->pool_mutex);

    pthread_cleanup_pop(1);    /* pthread_mutex_unlock(&pool->pool_mutex); */

    /*
     * Unlink the pool from the global list of all pools.
     */
    (void) pthread_mutex_lock(&thr_pool_lock);
    if (thr_pools == pool)

```

例A-2 thr_pool.c (続き)

```
        thr_pools = pool->pool_forw;
    if (thr_pools == pool)
        thr_pools = NULL;
    else {
        pool->pool_back->pool_forw = pool->pool_forw;
        pool->pool_forw->pool_back = pool->pool_back;
    }
    (void) pthread_mutex_unlock(&thr_pool_lock);

    /*
     * There should be no pending jobs, but just in case...
     */
    for (job = pool->pool_head; job != NULL; job = pool->pool_head) {
        pool->pool_head = job->job_next;
        free(job);
    }
    (void) pthread_attr_destroy(&pool->pool_attr);
    free(pool);
}
```

スレッドプールの例が示している内容

この例は、スレッドを使用したプログラミングで注意が必要な側面の1つである、取り消しと予期しないスレッドの終了を示しています。ワークスレッドが、予期されるとおりにタスク関数から単に復帰するのではなく、thr_pool_queue() に渡されたタスク関数内から pthread_exit() を呼び出すことによって終了する可能性があります。スレッドプールは、pthread_cleanup_push() 関数内の終了を捉えることによってこの状態から回復できます。唯一の弊害は、ここで別のワークスレッドを作成する必要がある点です。アクティブにタスクを処理しているワークスレッドは、thr_pool_destroy() で取り消されます。thr_pool_wait() または thr_pool_destroy() の呼び出し側は、待機中にアプリケーションによって取り消される可能性があります。この状態も、pthread_cleanup_push() を使用して対処できます。

このパッケージの例はそのままでも有用ですが、ここには含まれていない、次のようないくつかの機能がアプリケーションに必要な可能性があります。

- fork() の安全性(pthread_atfork() を使用)。
- 個々のタスクの完了を待つ機能。
- より高速なメモリ割り当て(コーディング例は malloc() を使用)。

索引

数字・記号

2 進型セマフォ, 131
32 ビットアーキテクチャー, 80

A

Ada, 175
aio_cancel(), 180
aio_error(), 180
aio_error() 関数, 180
aio_read() 関数, 180
aio_return(), 180
aio_return() 関数, 180
aio_suspend() 関数, 180
aio_waitn() 関数, 180
aio_write() 関数, 180
aiocbp, 180
assert, 248
assert 文, 129

C

cond_broadcast
 構文, 217
 戻り値, 217
cond_destroy
 構文, 214
 戻り値, 214
cond_init, 221
 USYNC_THREAD, 221

cond_init (続き)
 構文, 213
 戻り値, 213
cond_reltimedwait
 構文, 216
 戻り値, 216
cond_signal
 構文, 216
 戻り値, 217
cond_timedwait
 構文, 215
 戻り値, 215
cond_wait
 構文, 214
 戻り値, 215

D

dbx, 199
dbx(1), 238-240
Dijkstra, E. W., 131
DTrace, 237

E

errno, 39, 244
 大域変数, 244
exec, 160, 162, 164
exit, 164, 198

F

flockfile, 181
fork, 163
fork1, 162, 163
funlockfile, 181

G

getc, 181
getc_unlocked, 181
gethostbyname, 245
gethostbyname_r, 245
getrusage, 165

I

I/O
同期, 178
非逐次的, 180
非同期, 178, 179-180
標準, 181

K

kill(), 169, 171

L

libc, 229, 234
libC, 230
libCrun, 229
libCstd, 229
libiostream, 229
libm, 229
libmalloc, 229
libmapmalloc, 229
libnsl, 229
libpthread, 234
libpthread, 21
libresolv, 229
libsocket, 229

libthread, 234
libthread, 21
libX11, 229
longjmp, 165, 175
lost wake-up, 127
-lposix4 ライブラリ, POSIX 1003.1 セマフォ, 236
lseek(2), 181
LWP, 定義, 20

M

malloc, 33
MAP_NORESERVE, 73
MAP_SHARED, 163
mdb(1), 241
mmap, 163
mmap(2), 73
mprotect, 200
「MT-安全」なライブラリ
C++ 実行時共有オブジェクト
C++ 4.0 コンパイラ用, 230
C++ 5.0 コンパイラ用, 229
C++ 標準ライブラリ
Sun C++ 5.x コンパイラ用, 229
getXXbyYY_r の形式のネットワークインタ
フェース, 229
mmap(2) を使用した代替メモリー割り当てライ
ブラリ, 229
X11 ウィンドウルーチン, 229
安全でないインタフェースの「スレッド安全」
形式, 229
古典的 iostreams
C++ 用, 229
数学ライブラリ, 229
スレッド固有の errno サポート, 229
ネットワーク接続用のソケットライブラ
リ, 229
領域に効果的に使用するメモリー割り当
て, 229
mutex, 定義, 20
mutex_destroy
構文, 210
戻り値, 210
mutex_init, 221

- mutex_init (続き)
 - USYNC_THREAD, 221
 - 構文, 208-209
 - 戻り値, 210
 - mutex_lock
 - 構文, 211
 - 戻り値, 211
 - mutex_trylock
 - 構文, 212
 - 戻り値, 212
 - mutex_unlock, 211
 - 戻り値, 211
 - mutex 型
 - PTHREAD_MUTEX_ERRORCHECK, 97
 - PTHREAD_MUTEX_NORMAL, 97
 - PTHREAD_MUTEX_RECURSIVE, 97
 - mutex、相互排他ロック, 248
 - mutex の初期化, 94
 - mutex のスコープ, 83
- N**
- NDEBUG, 129
 - netdir, 229
 - netselect, 229
 - nice 値とクラス優先順位, 166
 - NULL
 - スレッド, 74, 199
 - NULL スレッド, 199
 - NULL 手続き
 - libpthread スタブ, 234
 - libthread スタブ, 234
- P**
- PC, プログラムカウンタ, 25
 - PC_GETCID, 166
 - PC_GETCLINFO, 166
 - PC_GETPARMS, 166
 - PC_SETPARMS, 166
 - Performance Analyzer, 238
 - Peterson のアルゴリズム, 255
 - PL/1 言語, 170
 - pread, 181
 - printf, 176
 - 問題, 245
 - prctl(), 166
 - prctl, 166
 - PC_GETCID, 166
 - PC_GETCLINFO, 166
 - PC_SETPARMS, 166
 - prctl(2), PC_GETPARMS, 166
 - prologen, セマフォを小さくする、P 操作, 131
 - pthread_atfork, 162
 - 構文, 48, 162
 - pthread_attr_destroy
 - 構文, 60
 - 戻り値, 60
 - pthread_attr_getdetachstate
 - 構文, 62
 - 戻り値, 62
 - pthread_attr_getguardsize
 - 構文, 64
 - 戻り値, 64
 - pthread_attr_getinheritsched
 - 構文, 70
 - 戻り値, 70
 - pthread_attr_getschedparam
 - 構文, 71-72
 - 戻り値, 72
 - pthread_attr_getschedpolicy
 - 構文, 68-69
 - 戻り値, 69
 - pthread_attr_getscope
 - 構文, 65-66
 - 戻り値, 66
 - pthread_attr_getstack
 - 構文, 77
 - 戻り値, 77
 - pthread_attr_getstacksize
 - 構文, 75
 - 戻り値, 75
 - pthread_attr_init
 - 構文, 59-60
 - 属性値, 59
 - 戻り値, 60

- pthread_attr_setdetachstate
 - 構文, 61-62
 - 戻り値, 62
- pthread_attr_setguardsize
 - 構文, 63
 - 戻り値, 63
- pthread_attr_setinheritsched
 - 構文, 69
 - 戻り値, 69
- pthread_attr_setschedparam
 - 構文, 70-71
 - 戻り値, 71
- pthread_attr_setschedpolicy
 - 構文, 67-68
 - 戻り値, 68
- pthread_attr_setscope
 - 構文, 64-65
 - 戻り値, 65
- pthread_attr_setstack
 - 構文, 76
 - 戻り値, 76
- pthread_attr_setstacksize
 - 構文, 74-75
 - 戻り値, 75
- pthread_barrier_destroy
 - 構文, 153
 - 戻り値, 153
- pthread_barrier_init
 - 構文, 151
 - 戻り値, 151-152
- pthread_barrier_wait
 - 構文, 152
 - 戻り値, 152-153
- pthread_barrierattr_destroy
 - 構文, 155
 - 戻り値, 155
- pthread_barrierattr_getpshared, 戻り値, 155
- pthread_barrierattr_init
 - 構文, 154
 - 戻り値, 154
- pthread_barrierattr_setpshared, 戻り値, 154
- pthread_cancel
 - 構文, 51
 - 戻り値, 52
- pthread_cleanup_pop, 構文, 54-55
- pthread_cleanup_push, 構文, 54
- pthread_cond_broadcast, 120, 127, 169
 - 構文, 125-126
 - 戻り値, 126
 - 例, 125
- pthread_cond_destroy
 - 構文, 126-127
 - 戻り値, 127
- pthread_cond_init
 - 構文, 118-119
 - 戻り値, 119
- pthread_cond_reltimedwait_np
 - 構文, 124-125
 - 戻り値, 125
- pthread_cond_signal, 120, 127, 128, 169
 - 構文, 121-122
 - 戻り値, 122
 - 例, 121
- pthread_cond_timedwait
 - 構文, 122-123
 - 戻り値, 123
 - 例, 123
- pthread_cond_wait, 127, 128, 169
 - 構文, 119-121
 - 戻り値, 121
 - 例, 121
- pthread_condattr_destroy
 - 構文, 114
 - 戻り値, 114
- pthread_condattr_getclock, 戻り値, 117
- pthread_condattr_getpshared
 - 構文, 115-116
 - 戻り値, 116
- pthread_condattr_init
 - 構文, 113-114
 - 戻り値, 114
- pthread_condattr_setclock
 - 構文, 116
 - 戻り値, 117
- pthread_condattr_setpshared
 - 構文, 115
 - 戻り値, 115

`pthread_create`
 構文, 31
 戻り値, 32
`pthread_detach`
 構文, 34-35
 戻り値, 35
`pthread_equal`
 構文, 42
 戻り値, 42
`pthread_exit`
 構文, 48-49
 戻り値, 49
`pthread_getconcurrency`
 構文, 67
 戻り値, 67
`pthread_getschedparam`
 構文, 45
 戻り値, 45
`pthread_getspecific`, 構文, 38-39
`pthread_join`, 179
 構文, 32-33
 戻り値, 33
`pthread_join(3C)`, 73
`pthread_key_create`
 構文, 36
 戻り値, 36
 例, 40
`pthread_key_delete`
 構文, 37
 戻り値, 37
`pthread_kill()`, 171
`pthread_kill`
 構文, 46-47
 戻り値, 47
`pthread_mutex_consistent_np`
 構文, 96
 戻り値, 96
`pthread_mutex_destroy`
 構文, 102-103
 戻り値, 103
`pthread_mutex_getprioceiling`
 構文, 91
 戻り値, 91
`pthread_mutex_init`
 構文, 95
 戻り値, 95
`pthread_mutex_lock`
 構文, 97
 戻り値, 97
 例, 103, 106, 107
`pthread_mutex_setprioceiling`
 構文, 90
 戻り値, 90
`pthread_mutex_timedlock`
 構文, 100-101, 101-102
 戻り値, 101, 102
`pthread_mutex_trylock`, 105
 構文, 99
 戻り値, 100
`pthread_mutex_trylock(3C)`, 249
`pthread_mutex_unlock`
 構文, 98-99
 戻り値, 99
 例, 103, 106, 107
`pthread_mutexattr_destroy`
 構文, 83
 戻り値, 83
`pthread_mutexattr_getprioceiling`
 構文, 89
 戻り値, 90
`pthread_mutexattr_getprotocol`
 構文, 88
 戻り値, 88
`pthread_mutexattr_getpshared`
 構文, 84
 戻り値, 84
`pthread_mutexattr_getrobust_np`
 構文, 93
 戻り値, 94
`pthread_mutexattr_gettype`
 構文, 86
 戻り値, 86
`pthread_mutexattr_init`
 構文, 81-82
 戻り値, 82
`pthread_mutexattr_setprioceiling`
 構文, 89

- `pthread_mutexattr_setprioceiling` (続き)
戻り値, 89
- `pthread_mutexattr_setprotocol`
構文, 86
戻り値, 88
- `pthread_mutexattr_setpshared`
構文, 83-84
戻り値, 84
- `pthread_mutexattr_setrobust_np`
構文, 92
戻り値, 93
- `pthread_mutexattr_settype`
構文, 84
戻り値, 85
- `pthread_once`
構文, 42-43
戻り値, 43
- `PTHREAD_PRIO_INHERIT`, 87
- `PTHREAD_PRIO_NONE`, 87
- `PTHREAD_PRIO_PROTECT`, 87
- `pthread_rwlock_destroy`
構文, 150
戻り値, 150
- `pthread_rwlock_init`
構文, 143
戻り値, 143
- `pthread_rwlock_rdlock`
構文, 144
戻り値, 144
- `pthread_rwlock_timedrdlock`
構文, 145
戻り値, 145-146
- `pthread_rwlock_timedwrlock`, 構文, 148
- `pthread_rwlock_tryrdlock`
構文, 146
戻り値, 146
- `pthread_rwlock_trywrlock`
構文, 147
戻り値, 148
- `pthread_rwlock_unlock`
構文, 149-150
戻り値, 150
- `pthread_rwlock_wrlock`
構文, 147
- `pthread_rwlock_wrlock` (続き)
戻り値, 147
- `pthread_rwlockattr_destroy`
構文, 140-141
戻り値, 141
- `pthread_rwlockattr_getpshared`
構文, 142
戻り値, 142
- `pthread_rwlockattr_init`
構文, 140
戻り値, 140
- `pthread_rwlockattr_setpshared`
構文, 141
戻り値, 141
- `PTHREAD_SCOPE_PROCESS`, 26
- `PTHREAD_SCOPE_SYSTEM`, 26, 64
- `pthread_self`
構文, 42
戻り値, 42
- `pthread_setcancelstate`
構文, 52
戻り値, 52
- `pthread_setcanceltype`
構文, 53
戻り値, 53
- `pthread_setconcurrency`
構文, 66
戻り値, 66
- `pthread_setschedparam`
構文, 44
戻り値, 44
- `pthread_setschedprio`
構文, 45-46
戻り値, 46
- `pthread_setspecific`
構文, 38
戻り値, 38
例, 40
- `pthread_sigmask()`, 171
- `pthread_sigmask`
構文, 47-48
戻り値, 48
- `pthread_spin_destroy()`
構文, 111-112, 112

pthread_spin_init, 構文, 108-109
 pthread_spin_lock
 構文, 110
 戻り値, 110
 pthread_spin_trylock
 構文, 110
 戻り値, 110-111
 pthread_spin_unlock
 構文, 111
 戻り値, 111
 PTHREAD_STACK_MIN, 74
 pthread_testcancel, 構文, 53-54
 putc, 181
 putc_unlocked, 181
 pwrite, 181

R

_r, 245
 read, 181
 RPC, 23, 229, 252
 RT,, 「リアルタイム」を参照
 rw_rdlock
 構文, 192
 戻り値, 192
 rw_tryrdlock
 構文, 192
 戻り値, 193
 rw_trywrlock
 構文, 194
 戻り値, 194
 rw_unlock
 構文, 194
 戻り値, 195
 rw_wrlock
 構文, 193
 戻り値, 193
 rwlock_destroy
 構文, 195-196
 戻り値, 196
 rwlock_init
 USYNC_THREAD, 221
 構文, 191
 戻り値, 191

S

SA_RESTART, 177
 64ビット環境
 /dev/kmem, 27
 /dev/mem, 27
 libkvm, 27
 /proc の制限, 28
 大容量の仮想アドレス空間, 27
 大容量ファイルのサポート, 28
 データ型モデル, 27
 ライブラリ, 28
 レジスタ, 28
 sched_yield, 250
 構文, 43
 戻り値, 43
 sem_destroy
 構文, 137
 戻り値, 137
 sem_init
 戻り値, 134
 例, 138
 sem_post, 131
 構文, 135
 戻り値, 135
 例, 138
 sem_trywait, 131
 構文, 136
 戻り値, 137
 sem_wait, 131
 構文, 136
 戻り値, 136
 例, 138
 sema_destroy
 構文, 220
 戻り値, 221
 sema_init
 USYNC_THREAD, 221
 構文, 133-134, 218
 戻り値, 218
 sema_post, 229
 構文, 219
 戻り値, 219
 sema_trywait
 構文, 220

sema_trywait (続き)
 戻り値, 220
sema_wait
 構文, 219
 戻り値, 220
setjmp, 165, 174, 175
SIG_DFL, 168
SIG_IGN, 168
SIG_SETMASK, 47
sigaction(), 168, 169
SIGFPE, 169, 175
SIGILL, 169
SIGINT, 170, 174
SIGIO, 170
siglongjmp, 175
signal.h, 201, 202
sigprocmask(), 171
sigqueue(), 169
SIGSEGV, 72, 169
sigsend(), 169
sigsetjmp, 175
sigtimedwait(), 173
sigwait(), 172-173, 173
sigwait, 175
singly-linked list with nested locking, 例, 106
stack_base, 76, 197
stack_size, 74, 197
start_routine(), 198

T

THR_BOUND, 198
thr_continue, 198
 構文, 189
 戻り値, 189
thr_create, 200
 構文, 197-199
 戻り値, 199
THR_DAEMON, 198
THR_DETACHED, 198
thr_exit, 198
 構文, 202
 戻り値, 203

thr_getprio
 構文, 208
 戻り値, 208
thr_getspecific
 構文, 206
 戻り値, 206
thr_join
 構文, 203
 戻り値, 204
thr_keycreate
 構文, 204
 戻り値, 205
thr_kill, 229
 構文, 201
 戻り値, 201
thr_min_stack, 197, 199
thr_self, 構文, 200
thr_setprio
 構文, 206-207
 戻り値, 207
thr_setspecific
 構文, 205
 戻り値, 205
thr_sigsetmask, 229
 構文, 202
 戻り値, 202
thr_suspend
 構文, 188
 戻り値, 189
thr_yield, 構文, 201
Thread Analyzer, 238
TLI, 229
TS, 「タイムシェアスケジューリングクラス」を
 参照

U

/usr/include/errno.h, 232
/usr/include/limits.h, 232
/usr/include/pthread.h, 232
/usr/include/signal.h, 232
/usr/include/thread.h, 232
/usr/include/unistd.h, 232

/usr/lib, 32 ビットスレッドライブラリ、Solaris
 9, 236
 /usr/lib/lwp, 32 ビットスレッドライブラリ、
 Solaris 8, 236
 /usr/lib/lwp/64, 64 ビットスレッドライブラリ、
 Solaris 8, 236
 USYNC_PROCESS, 221
 mutex, 208
 条件変数, 213
 セマフォ, 218
 読み取り/書き込みロック, 191
 USYNC_PROCESS_ROBUST, mutex, 208
 USYNC_THREAD
 mutex, 208
 条件変数, 213
 セマフォ, 218
 読み取り/書き込みロック, 191

V

verhogen, セマフォを大きくする、v 操作, 131
 vfork, 162

W

write(2), 181

X

XDR, 229

あ

アーキテクチャー
 SPARC, 80, 254, 256
 マルチプロセッサ, 253-257
 新しいスレッドの停止, 198
 アプリケーションレベルのスレッド, 20
 誤った呼び起こし, 177
 アラーム, 164

アルゴリズム

MT より高速, 22
 逐次, 257
 並列, 257
 安全、スレッドインタフェース, 225-230
 安全性、スレッドインタフェース, 230

い

移植性, 80
 イベント通知, 133

え

遠隔手続き呼び出しRPC, 23

か

カスタムスタック, 73-74, 200
 片方向リスト, 例, 106
 完了セマンティクス, 174

き

キー

値の格納, 206
 値をキーに結合, 205
 危険領域, 255
 既存のスレッド固有データキーの削除, 37
 きめ細かいロック, 247
 きめの粗いロック, 247
 キャッシュ, スレッドデータ構造, 252
 キャッシュ、定義済み, 253
 競合, 249, 250
 共有データ, 25, 247
 共有メモリー型のマルチプロセッサ, 254
 局所変数, 245
 切り離されたスレッド, 61, 198

け

計数型セマフォ, 21, 131

軽量プロセス

スケジューリングクラス, 165

定義, 20

デバッグ, 241

結合

LWP スレッドを, 198

値をキーに, 205

結合スレッド, 20

定義, 20

こ

公平配分スケジューラ (FSS) スケジューリングクラス, 168

コードモニター, 247, 249

コードロック, 246, 248

固定優先順位スケジューリングクラス (FX), 168

さ

再入可能, 246

関数, 227, 228

説明, 246-248

方法, 246

作成

スタック, 73-74, 76, 197

し

時間切れ, 例, 124

シグナル(), 168

シグナル

SIG_BLOCK, 47

SIG_SETMASK, 47

SIG_UNBLOCK, 47

SIGSEGV, 72

アクセスマスク, 201

継承, 197

現在のマスクの置換, 47

スレッドへ送信, 201

シグナル (続き)

スレッドへの送信, 46, 171

同期, 169

ハンドラ, 168, 173

非同期, 169, 170, 173

保留, 189, 197

マスク, 25

マスクからの削除, 47

マスクされていない捕獲された, 177

マスクへの追加, 47

シグナルマスクのアクセス, 47

シグナルマスクの検査, 47, 201

シグナルマスクの置換, 47

シグナルマスクの変更, 47, 201

システムコール, エラー処理, 244

システムスケジューリングクラス, 165

実行の再開, 189

実行の続行, 189

自動, スタック割り当て, 73

循環リンクリスト, 例, 107

条件変数, 80, 112-117, 130

1つのスレッドのブロック解除, 121

間隔指定のブロック, 124

クロック選択の取得, 117

クロック選択の設定, 116

時刻指定のブロック, 122

状態の削除, 126

初期化, 118

スコープの取得, 115

スコープの設定, 115

スレッドのブロック解除, 125

属性の削除, 114

属性の初期化, 113

ブロック, 119

条件待ち, POSIX スレッド, 177

シングルスレッド

仮定, 243

コード, 80

定義, 20

プロセス, 164

す

スケジューリング

- クラス, 26
- システムクラス, 165
- スコープ, 26
- 方針, 26

スケジューリングクラス

- 公平配分スケジューラ (FSS), 168
- 固定優先順位スケジューラ (FX), 168
- タイムシェア, 166
- 優先順位, 166
- リアルタイム, 167

スコープ, 状態, 26

スタック, 252

- アドレス, 76, 197
- オーバーフロー, 73
- 解放, 200
- カスタム, 200
- 境界, 72
- 最小サイズ, 74
- サイズ, 74, 197
- 作成, 76, 197
- プログラマによって割り当てられた, 73-74, 200
- ポインタ, 25
- ポインタを返す, 227
- レッドゾーン, 73, 200

スタックサイズ, 74, 197, 199

- 最小, 199, 200
- 最小値の検索, 199

ストアバッファ, 256

スピンロック, 定義, 108-112

スレッド

- NULL, 74, 199
- 安全, 225
- 安全性, 230
- キー, 205
- 切り離された, 61, 198
- 結合, 203
- 識別子, 198, 199
- 識別子の獲得, 200
- シグナル, 177
- シグナルの送信, 201
- 実行の明け渡し, 201
- 終了, 48, 202

スレッド (続き)

- スタック, 227
- スレッド固有データ, 244
- 生成, 197, 199, 252
- 停止, 198
- 停止している, 189
- デーモン, 198
- 同期, 80, 158
- 同期モデル, 26
- 取り消し, 49, 51
- ユーザーレベル, 20, 25
- 優先順位, 197
- スレッドキー値の格納, 206
- スレッド固有キー割り当ての取得, 38-39
- スレッド固有キー割り当ての設定, 38
- スレッド固有データ, 35
 - 新しい記憶領域クラス, 244
 - 取得, 206
 - 設定, 205
 - 大域から局所, 40
 - 例, 39-41
- スレッド固有のキー
 - 作成, 36
 - 生成, 204
- スレッド識別子, 42
- スレッド識別子の比較, 42
- スレッド指定シグナル, 173
- スレッド生成, 終了状態, 31
- スレッド専用記憶領域, 25
- スレッド同期
 - 条件変数, 27
 - セマフォ, 27, 131-139
 - 相互排他ロック, 27, 80
 - 読み取り/書き込みロック, 27, 139
- スレッドの結合, 203
- スレッドの終了待ち, 32-33, 49, 61
- スレッドの取り消し, 26, 49
 - 取り消しポイント, 50
- スレッドの優先順位の検索, 208
- スレッドのライフサイクル, 29
- スレッドプール, 定義, 259
- スレッドへのシグナルの送信, 46
- スレッドローカルな記憶領域, 35
- スワップ空間, 73

せ

「生産者 / 消費者」問題, 138, 156, 221, 254

生成

スタック, 200

スレッド, 252

静的記憶領域, 243

セマフォ, 80, 131-139, 158

2 値型, 131

カウンタの減算, 136

加算, 135

計数型, 131

計数型、定義, 21

状態の削除, 137

初期化, 133

セマフォの値を小さくする, 131

セマフォの値を増やす, 131

名前付き, 132

名前なし, 132

プロセス間, 134

呼び出しスレッドのブロック, 135

そ

相互排他ロック, 80, 107

mutex 堅牢度属性の取得, 93

mutex 属性のプロトコルの取得, 88

mutex 属性のプロトコルの設定, 86

mutex 属性の優先順位上限の取得, 89

mutex 属性の優先順位上限の設定, 89

mutex の堅牢度属性の設定, 92

mutex の削除, 83

mutex の状態の削除, 102

mutex の優先順位上限の取得, 91

mutex の優先順位上限の設定, 90

型属性の設定, 85

初期化, 94

整合性保持, 96

相互排他スコープの取得, 84

属性, 81

デッドロック, 104

デフォルト属性, 80

ブロックしないで行うロック, 99

ロック, 96

ロック解除, 98

た

大域

状態, 246

データ, 247

副作用, 251

変数, 39, 243-244

タイマー, LWP ごと, 164

タイムシェアスケジューリングクラス, 166

ち

逐次アルゴリズム, 257

つ

ツール

dbx, 199

dbx(1), 238-240

Dtrace, 237

mdb(1), 241

Performance Analyzer, 238

Thread Analyzer, 238

強く順序付けられたメモリー, 254

て

定義されたスレッド, 19

ディスパッチ優先順位, 165

計算, 167

データ

競合, 225

共有, 25, 255

スレッド固有, 35

ロック, 246, 248

データの競合, 236, 238

テープドライブのストリーミング, 179

デーモンスレッド, 198

デッドロック

検出, 238

再帰的, 249

スケジューリング, 250

定義, 249

デバッグ, 236
 dbx, 199
 dbx(1), 238-240
 dtrace, 237
 mdb(1), 241
 再帰的デッドロック, 237
 条件待ちから復帰後に条件を再評価, 237
 相互排他ロックを解放しないでロングジャンプ, 237
 大域メモリーの同期がとられない, 236
 大量の自動配列, 237
 デッドロック, 236
 同期における見えない間隙, 237
 非同期シグナル, 237
 不適切なスタックサイズ, 237
 呼び出しスタックへポインタを渡す, 236
 デフォルトのスレッドの生成, 30

と

同期I/O, 178, 179
 同期オブジェクト
 条件変数, 80, 112-117, 130
 セマフォ, 80, 131-139, 217, 221
 相互排他ロック, 80, 107
 読み取り/書き込みロック, 195
 同期シグナル, 169
 登録状態, 25
 トータルストア順序, 256
 トラップ, 169
 スレッドによる処理, 169
 デフォルトの処理, 170

は

バリアー同期, 151-155, 257

ひ

ヒープから記憶領域を割り当て, malloc, 33
 非結合スレッド
 キャッシュ, 252

非結合スレッド (続き)
 定義, 20
 非逐次的I/O, 180
 非同期
 I/O, 178, 179
 イベント通知, 133
 シグナル, 169, 170, 173
 セマフォの使用, 133
 非同期シグナル安全
 関数
 POSIX, 176
 Solaris, 228
 シグナルハンドラ, 176
 スレッド, 173
 非同期入出力
 操作, 180
 待機中, 180
 標準, UNIX, 21
 標準 I/O, 181

ふ

不可分な、定義, 80
 複数読み取り、単一書き込みロック, 195
 不変式, 129, 248
 フラグthr_create, 198
 プログラマによって割り当てられたスタック, 73-74, 200
 プロファイリング, 238
 マルチスレッドプログラム, 165

へ

並列, アルゴリズム, 257
 変数
 条件, 80, 112-117, 130, 158
 大域, 243-244
 プリミティブ, 80

ほ

ボトルネック, 250

ま

マスクからのシグナルの削除, 47
マスクへのシグナルの追加, 47
マルチスレッド, 定義, 20
マルチスレッドアプリケーションのコンパイル, 232
マルチプロセッサ, 252-257

め

メモリー
一貫性, 252
順序, 弱い, 255
強く順序付けられた, 254

も

モニター, 249
モニター、コード, 247

ゆ

ユーザーレベルのスレッド, 20, 25
優先順位, 25
継承, 197, 208
スレッドに設定, 206-207
とスケジューリング, 167, 207
範囲, 207
優先順位の逆転, 87
優先順位の継承, 197

よ

読み取り/書き込みロック, 80, 195
attributes, 139
書き込みロックの獲得, 147
書き込みロックのロック, 147
削除, 150
読み取りロックの解放, 149
読み取りロックの獲得, 144
読み取りロックのロック, 146

読み取り/書き込みロック (続き)

ロック属性の削除, 140
ロック属性の取得, 142
ロック属性の設定, 141
ロックの初期化, 140, 143
弱いメモリー順序, 255

ら

ライブラリ
c ルーチン, 243
MT-安全, 229-230

り

リアルタイム, スケジューリング, 167
リソースの制限, 165

れ

例, スレッドプール, 259
レッドゾーン, 73, 200

ろ

ロック, 80, 247
きめ細かい, 247, 250
きめの粗い, 247, 250
コード, 247
指針, 250
条件, 105
相互排他, 80, 107
データ, 247-248
不変式, 248
読み取り/書き込み, 80, 195
ロック階層, 249

わ

割り込み, 169

