
スレッドプログラミングガイド

Cocoa > Process Management



2010-04-28



Apple Inc.
© 2010 Apple Inc.
All rights reserved.

本書の一部あるいは全部を Apple Inc. から書面による事前の許諾を得ることなく複製（コピー）することを禁じます。また、製品に付属のソフトウェアは同梱のソフトウェア使用許諾契約書に記載の条件のもとでお使いください。書類を個人で使用する場合に限り1台のコンピュータに保管すること、またその書類にアップルの著作権表示が含まれる限り、個人的な利用を目的に書類を複製することを認めます。

Apple ロゴは、米国その他の国で登録された Apple Inc. の商標です。

キーボードから入力可能な Apple ロゴについても、これを Apple Inc. からの書面による事前の許諾なしに商業的な目的で使用すると、連邦および州の商標法および不正競争防止法違反となる場合があります。

本書に記載されているテクノロジーに関しては、明示または黙示を問わず、使用を許諾しません。本書に記載されているテクノロジーに関するすべての知的財産権は、Apple Inc. が保有しています。本書は、Apple ブランドのコンピュータ用のアプリケーション開発に使用を限定します。

本書には正確な情報を記載するように努めました。ただし、誤植や制作上の誤記がないことを保証するものではありません。

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
U.S.A.

アップルジャパン株式会社
〒163-1450 東京都新宿区西新宿
3丁目20番2号
東京オペラシティタワー
<http://www.apple.com/jp/>

Apple, the Apple logo, AppleScript, Carbon, Cocoa, Cocoa Touch, iMac, Mac, Mac OS, Objective-C, QuickTime, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

iOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under license.

Intel and Intel Core are registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java is a registered trademark of Oracle and/or its affiliates.

Ping is a registered trademark of Karsten Manufacturing and is used in the U.S. under license.

UNIX is a registered trademark of The Open Group

Apple Inc. は本書の内容を確認しておりますが、本書に関して、明示的であるか黙示的であるかを問わず、その品質、正確さ、市場性、または特定の目的に対する適合性に関して何らかの保証または表明を行うものではありません。その結果、本書は「現状有姿のまま」提供され、本書の品質または正確さに関連して発生するすべての損害は、購入者であるお客様が負うものとします。

いかなる場合も、Apple Inc. は、本書の内容に含まれる瑕疵または不正確さによって生じる直接的、間接的、特殊的、偶発的、または結果的損害に対する賠償請求には一切応じません。そのような損害の可能性があらかじめ指摘されている場合においても同様です。

上記の損害に対する保証および救済は、口頭や書面によるか、または明示的や黙示的であるかを問わず、唯一のものであり、その他一切の保証にかわるものです。Apple Inc. の販売店、代理店、または従業員には、この保証に関する規定に何らかの変更、拡張、または追加を加える権限は与えられていません。

一部の国や地域では、黙示あるいは偶発的または結果的損害に対する賠償の免責または制限が認められていないため、上記の制限や免責がお客様に適用されない場合があります。この保証はお客様に特定の法的権利を与え、地域によってはその他の権利がお客様に与えられる場合もあります。

目次

序章

はじめに 9

この書類の構成 9

関連項目 9

第1章

スレッドプログラミングの概要 11

スレッドの概要 11

スレッド処理の用語 12

スレッドに代わる手法 12

スレッド処理のサポート 14

スレッド処理パッケージ 14

実行ループ 15

同期ツール 16

スレッド間通信 16

設計のヒント 18

スレッドを明示的に作成しない 18

スレッドを適度にビジーにする 18

データ構造体を共有しない 19

スレッドおよびユーザインターフェイス 19

終了時のスレッドの動作に注意する 19

例外処理 20

スレッドをきちんと終了させる 20

ライブラリのスレッドの安全性 21

第2章

スレッドの管理 23

スレッドのコスト 23

スレッドの作成 24

NSThreadの使用 24

POSIXスレッドの使用 26

NSObjectを使用したスレッドの作成 27

その他のスレッド処理テクノロジーの使用 27

CocoaアプリケーションでのPOSIXスレッドの使用 28

スレッド属性の設定 28

スレッドのスタックサイズの設定 29

スレッドのローカルストレージの設定 29

スレッドのデタッチ状態の設定 29

スレッド優先度の設定 30

スレッドエントリルーチンの作成 31

自動解放プールの作成 31

例外ハンドラの設定 32

実行ループの設定 32
スレッドの終了 32

第3章 実行ループ 35

実行ループの構造 35
 実行ループモード 36
 入力ソース 37
 タイマーソース 40
 実行ループオブザーバ 40
 イベントの実行ループのシーケンス 41
実行ループを使用する状況 42
実行ループオブジェクトの使用 43
 実行ループオブジェクトの取得 43
 実行ループの設定 43
 実行ループの開始 44
 実行ループの終了 46
 スレッドの安全性と実行ループオブジェクト 46
実行ループソースの設定 46
 カスタム入力ソースの定義 46
 タイマーソースの設定 51
 ポートベースの入力ソースの設定 53

第4章 同期 61

同期ツール 61
 アトミック操作 61
 メモリバリアおよびvolatile変数 62
 ロック 62
 条件変数 64
 セレクトルーチンの実行 64
同期のコストおよびパフォーマンス 65
スレッドの安全性とシグナル 66
スレッドセーフな設計のヒント 66
 同期を完全に避ける 66
 同期の制限事項を理解する 67
 コードの正確性を脅かす要因に注意する 67
 デッドロックおよびライブロックへの配慮 68
 volatile変数の適切な使用 68
アトミック操作の使用 69
ロックの使用 71
 POSIXミューテックスロックの使用 72
 NSLockクラスの使用 72
 @synchronizedディレクティブの使用 73
 その他のCocoaロックの使用 73
条件変数の使用 76

NSConditionクラスの使用 76
POSIX条件変数の使用 77

付録 A **スレッドの安全性のまとめ 79**

Cocoa 79
 Foundationフレームワークのスレッドの安全性 79
 Application Kitフレームワークのスレッドの安全性 83
 Core Dataフレームワーク 86
Core Foundation 86

用語解説 87

改訂履歴 **書類の改訂履歴 89**

図、表、リスト

第 1 章 スレッドプログラミングの概要 11

表 1-1	スレッドの代替テクノロジー 13
表 1-2	スレッドテクノロジー 14
表 1-3	通信メカニズム 17

第 2 章 スレッドの管理 23

表 2-1	スレッド作成コスト 23
表 2-2	スレッドのスタックサイズの設定 29
リスト 2-1	C言語でのスレッドの作成 26
リスト 2-2	スレッドエントリポイントルーチンの定義 31
リスト 2-3	長時間処理に組み込んだ終了条件の検査 33

第 3 章 実行ループ 35

図 3-1	実行ループの構造およびソース 36
図 3-2	カスタム入力ソースの操作 47
表 3-1	定義済みの実行ループモード 37
表 3-2	ほかのスレッド上にあるセレクトの実行 39
リスト 3-1	実行ループオブザーバの作成 43
リスト 3-2	実行ループの実行 45
リスト 3-3	カスタム入力ソースオブジェクトの定義 48
リスト 3-4	実行ループソースのスケジューリング 48
リスト 3-5	入力ソースに含まれている作業の実行 49
リスト 3-6	入力ソースの無効化 49
リスト 3-7	実行ループソースの組み込み 50
リスト 3-8	アプリケーションデリゲートへの入力ソースの登録と除去 50
リスト 3-9	実行ループのスリープ解除 51
リスト 3-10	NSTimerを使用したタイマーの作成およびスケジューリング 52
リスト 3-11	Core Foundationを使用したタイマーの作成およびスケジューリング 52
リスト 3-12	メインスレッドの起動メソッド 53
リスト 3-13	Machポートのメッセージの処理 53
リスト 3-14	Machポートを使用したワーカースレッドの起動 54
リスト 3-15	Machポートを使用したチェックインメッセージの送信 55
リスト 3-16	メッセージポートの登録 55
リスト 3-17	Core Foundationメッセージポートの新しいスレッドへの接続 56
リスト 3-18	チェックインメッセージの受信 57
リスト 3-19	スレッド構造の設定 58

第4章

同期 61

表 4-1	ロックのタイプ 63
表 4-2	ミューテックスおよびアトミック操作のコスト 65
表 4-3	アトミックな数学的演算および論理演算 69
リスト 4-1	アトミック操作の実行 71
リスト 4-2	ミューテックスロックの使用 72
リスト 4-3	Cocoa条件変数の使用 76
リスト 4-4	Cocoa条件のシグナル送信 77
リスト 4-5	POSIX条件変数の使用 77
リスト 4-6	条件ロックへのシグナル送信 78

はじめに

スレッドは、単一のアプリケーションの内部で複数のコードパスを並行して実行できるテクノロジーの1つです。オペレーションオブジェクトやGrand Central Dispatch (GCD) などの新しいテクノロジーは、並行性を実装するための最新で効率的なインフラストラクチャになりますが、Mac OS X およびiOSには、スレッドを作成および管理するためのインターフェイスも用意されています。

この文書では、Mac OS Xで使用可能なスレッドパッケージを紹介して、それらの使用方法を説明します。この文書では、スレッド処理およびアプリケーション内部でのマルチスレッドコードの同期をサポートする関連テクノロジーについても説明します。

重要：新しいアプリケーションを開発する場合は、並行性を実装するMac OS Xの代替テクノロジーについて調べることをお勧めします。スレッド化されたアプリケーションを実装するために必要な設計手法についての知識がまだ十分でない場合であればなおさらです。これらの代替テクノロジーによって、複数の並行な実行パスを実装するために必要な作業が全体として簡素化され、従来のスレッドよりもずっと優れた性能を発揮します。これらのテクノロジーについては、『*Concurrency Programming Guide*』を参照してください。

この書類の構成

この文書は次の章と付録で構成されています。

- 「[スレッドプログラミングの概要](#)」（11 ページ）では、スレッドの概念およびアプリケーションを設計する上でのスレッドの機能について紹介します。
- 「[スレッドの管理](#)」（23 ページ）では、Mac OS Xにおけるスレッド処理テクノロジーおよびその使用方法を説明します。
- 「[実行ループ](#)」（35 ページ）では、二次スレッドでのイベント処理ループの管理方法について説明します。
- 「[同期](#)」（61 ページ）では、同期の問題についておよび複数スレッドによってデータの破損やプログラムのクラッシュを防ぐためのツールについて説明します。
- 「[スレッドの安全性のまとめ](#)」（79 ページ）では、Mac OS XおよびiOSに元々備わっているスレッドの安全性の概要を示し、主要なフレームワークのいくつかを説明します。

関連項目

スレッドに代わる手法については、『*Concurrency Programming Guide*』を参照してください。

この文書では、POSIXスレッドAPIの使用方法については、概略だけを説明しています。使用可能なPOSIXスレッドルーチンの詳細については、pthreadのmanページを参照してください。POSIXスレッドの詳細な説明および使用方法については、David R. Butenhof著『*Programming with POSIX Threads*』を参照してください。

スレッドプログラミングの概要

長い間、コンピュータの最大性能は、コンピュータの中核となる単一のマイクロプロセッサの速度によって、大きく制限されてきました。一方、個々のプロセッサ速度が実用上の限界に近づくに従い、各チップメーカーはマルチコア設計に転換して、複数のタスクを同時に実行できる能力をコンピュータに与えました。Mac OS Xのシステム関連タスクは、常にこのマルチコアを活用して実行されますが、独自のアプリケーションでもスレッドを介してマルチコアを活用できます。

スレッドの概要

スレッドを実行すると、複数パスによる実行を比較的手軽にアプリケーションに実装できます。システムレベルでは、プログラムの必要度に基づいて各プログラムに実行時間を配分しながら、プログラムが並列実行されます。一方、各プログラムの内部では、1つ以上のスレッドが実行されます。スレッドを使用すると、異なるタスクを同時またはほぼ同時に実行できます。スレッドの実際の実行はシステム自体によって管理され、使用可能なコアに対してスレッドの実行をスケジューリングしたり、ほかのスレッドを実行できるように必要に応じてプリエンプティブな割り込みを行ったりします。

スレッドを技術的に見れば、カーネルレベルのデータ構造体とアプリケーションレベルのデータ構造体を組み合わせて、コードの実行を管理できるようにしたものです。カーネルレベルの構造体では、スレッドに対するイベントのディスパッチと、使用可能なコアの1つで実行されるスレッドのプリエンプティブスケジューリングを連携させます。アプリケーションレベルの構造体には、関数呼び出しを格納するためのコールスタックと、スレッドの属性および状態を管理および操作するためにアプリケーションで必要とする構造体が含まれています。

並行アプリケーションでない場合、実行されるスレッドは1つだけです。このスレッドは、アプリケーションのmainルーチンと同時に開始および終了されて、さまざまなメソッドまたは関数の1つずつに順に分岐しながらアプリケーションの動作全体を実装します。これに対して、並行処理をサポートするアプリケーションの場合は、開始時点のスレッドは1つですが、必要に応じてスレッドを追加することにより、付加的な実行パスを作成します。新しいそれぞれのパスは、アプリケーションのmainルーチンのコードと独立して実行される、独自のカスタム開始ルーチンを持ちます。複数スレッドをサポートするアプリケーションには、次の2つの潜在的な利点があります。

- 複数スレッドのアプリケーションは、応答性が向上したと認識される可能性があります。
- 複数スレッドのアプリケーションは、マルチコアシステムでのリアルタイム性能が向上する可能性があります。

アプリケーションのスレッドが1つだけであれば、その1つのスレッドですべての処理を実行する必要があります。イベントに応答し、アプリケーションのウィンドウを更新し、アプリケーションの動作を実装するために必要なすべての演算処理を実行しなければなりません。スレッドが1つしかない場合は、一度に1つの処理しか実行できないことが問題になります。終了まで長時間かかる演算処理があった場合どうなるでしょう。必要な値を計算するためにコードがビジーな間、アプリケーションはユーザイベントに応答せず、ウィンドウを更新しません。この動作がある程度続けば、ア

アプリケーションはハングしていると判断され、強制終了されるおそれがあります。一方、カスタム演算処理を別のスレッドに移動すれば、アプリケーションのメインスレッドでユーザのインタラクティブな操作に随時応答できます。

現在ではマルチコアコンピュータが一般化しており、一部のタイプのアプリケーションでは、スレッドを使用することによってパフォーマンスが向上します。異なるタスクを実行するスレッドを異なるプロセッサコアでいっせいに実行できるため、アプリケーションの一定時間の作業量を増加できます。

もちろん、スレッドによって、アプリケーションのパフォーマンス上の問題がすべて解決されるわけではありません。スレッドは利点をもたらしますが、潜在的な問題を伴っています。複数実行パスをサポートするアプリケーションは、コードが大幅に複雑になります。各スレッドは、アプリケーションの状態情報を破損しないために、ほかのスレッドと動作を調和させる必要があります。単一のアプリケーションの各スレッドは同じメモリ空間を共有するため、各スレッドで共通するすべてのデータ構造体に自由にアクセスできます。2つのスレッドで同じデータ構造体を同時に操作しようとし、一方のスレッドによってもう一方のスレッドの変更を上書きすれば、データ構造体を破損する結果になります。適切な保護機構を組み込んである場合でも、コンパイラが実施する最適化によって軽微または軽微でないバグがコードに組み込まれないよう注意する必要があります。

スレッド処理の用語

スレッドおよびスレッドをサポートするテクノロジーの詳細を説明する前に、基本用語の定義を示す必要があります。

この文書では、CarbonのMultiprocessor Services APIやUNIXシステムとは異なる意味で「タスク」という用語を使用しています。以前のバージョンのMac OSでは、Multiprocessor Servicesを使用して作成したスレッドとCarbon Thread Manager APIを使用して作成したスレッドを区別するために、「タスク」という用語を使用していました。UNIXシステムでは、実行中のプロセスを「タスク」と呼ぶこともあります。実践的な意味では、Multiprocessor Servicesタスクはプリエンティブにスケジューリングされたスレッドに相当します。

Carbon Thread Manager APIおよびMultiprocessor Services APIはいずれもMac OS Xの従来のテクノロジーであることを踏まえて、この文書では次の用語を採用しています。

- **スレッド**という用語は、コード実行の個別のパスを示します。
- **プロセス**という用語は、実行中の実行可能プログラムを示します。プロセスは、複数のスレッドを含むことができます。
- **タスク**という用語は、実行する必要がある作業の抽象概念を示します。

スレッドに代わる手法

スレッドを独自に作成する場合の問題の1つとして、コードがわかりにくくなる点があります。スレッドは、アプリケーションで並行処理をサポートするための方法としては、比較的低レベルで複雑です。設計の選択肢による影響を十分に理解していないと、わずかな動作の変化という軽微な影響から、アプリケーションのクラッシュとユーザデータの破損という深刻な影響までを持つ可能性のある、同期やタイミングの問題がたやすく発生します。


スレッドや並行処理がまったく必要ない可能性についても考える必要があります。スレッドによって解決される具体的な問題は、複数のコードパスを同じプロセスの内部で同時に実行する方法です。ただし、長時間かかる作業だからといって、並行処理に向いているとは限りません。スレッドにより、メモリ消費とCPU時間の両方に関して、プロセスに多大なオーバーヘッドがかかります。目的とするタスクに対してこのオーバーヘッドが大きすぎたり、ほかのオプションを実装するほうが簡単であるとわかったりすることがあります。

表1-1に、スレッドに代わる手法のいくつかを示します。この表は、スレッドに代わるテクノロジー（オペレーションオブジェクト、GCDなど）と、既存の単一スレッドの有効活用に注力することによる代替手段の両方を含んでいます。

表 1-1 スレッドの代替テクノロジー

テクノロジー	説明
オペレーションオブジェクト	<p>Mac OS X v10.5で導入されたオペレーションオブジェクトは、通常は二次スレッドで実行されるタスクのラッパーです。このラッパーによってタスクを実行するときのスレッド管理の側面が隠されるため、開発者はスレッド管理から解放されてタスク自体に集中できます。通常は、オペレーションキューオブジェクトとこれらのオブジェクトを組み合わせて使用します。1つまたは複数のスレッド上でオペレーションオブジェクトの実行を実際に管理するのは、オペレーションキューオブジェクトです。</p> <p>オペレーションオブジェクトの使用方法的詳細については、『<i>Concurrency Programming Guide</i>』を参照してください。</p>
Grand Central Dispatch (GCD)	<p>Mac OS x v10.6で導入されたGrand Central Dispatchは、開発者がスレッドの管理ではなく実行する必要のあるタスクに集中できる、もう1つのスレッド代替テクノロジーです。GCDを使用する場合は、実行するタスクを定義して作業キューに追加します。作業キューは、適切なスレッド上のタスクのスケジューリングを処理します。作業キューでは、使用可能なコアの数および現在の負荷が考慮されるため、ユーザがスレッドを使用して独自に行うよりタスクの実行が効率的になります。</p> <p>GCDおよび作業キューの使用方法的詳細については、『<i>Concurrency Programming Guide</i>』を参照してください。</p>
アイドル時間通知	<p>比較的短時間なタスクで、優先度の低いタスクの場合は、アイドル時間通知を使用すると、アプリケーションがビジーでないときにタスクを一度に実行できます。Cocoaでは、NSNotificationQueueオブジェクトを使用してアイドル時間通知をサポートします。アイドル時間通知を要求するには、NSPostWhenIdleオプションを使用してデフォルトのNSNotificationQueueオブジェクトに通知を送信します。このキューは、実行ループがアイドルになるまで通知オブジェクトの配信を遅延させます。詳細については『<i>Notification Programming Topics</i>』を参照してください。</p>
非同期関数	<p>システムインターフェイスには、並行性を自動で実現する多数の非同期関数が含まれています。これらのAPIは、システムデーモンおよびプロセスを使用するかカスタムスレッドを作成してタスクを実行し、結果を返します（実際の実装は、コードと分離されているため、意識する必要はありません）。アプリケーションを設計するときに非同期動作を提供する関数を探して、同等な同期関数をカスタムスレッドで使用する代わりに非同期関数を使用することを検討します。</p>

テクノロジー	説明
タイマー	アプリケーションのメインスレッドでタイマーを使用すると、スレッドを必要とするほど大規模ではない一方で定期的な処理を必要とする、周期的なタスクを実行できます。タイマーについては、「 タイマーソース 」（40 ページ）を参照してください。
別プロセス	プロセスはスレッドより高負荷です。ただし、アプリケーションとの関連性がわずかにすぎないタスクの場合は、別プロセスを作成すると有用です。大量のメモリを必要とするタスクやroot権限を使用して実行する必要のあるタスクの場合などにプロセスを使用します。たとえば、64ビットのサーバプロセスを使用して大規模データセットの計算を実行しながら、32ビットのアプリケーションで結果をユーザに表示します。

 **警告：** fork関数を使用して別プロセスを起動する場合は、常にforkの呼び出しに続けてexecまたは同様な関数を呼び出す必要があります。Core Foundationフレームワーク、Cocoaフレームワーク、またはCore Dataフレームワークに明示的または暗黙的に依存するアプリケーションの場合は、これらのフレームワークを適切に動作させるには、exec関数の呼び出しを続ける必要があります。

スレッド処理のサポート

スレッドを使用する既存のコードに対して、MacOSXおよびiOSには、アプリケーションでスレッドを作成するための複数のテクノロジーが用意されています。さらに、この両方のシステムでは、スレッドで実行する必要のある作業の管理および同期もサポートしています。以降の各セクションでは、MacOSXおよびiOSでスレッドを扱う場合に知る必要のある、複数の主要テクノロジーについて説明します。

スレッド処理パッケージ

スレッドの基礎にある実装メカニズムはMachスレッドですが、Machレベルでスレッドを扱うことは、たとえあったとしてもまれです。通常は代わりに、もっと便利なPOSIX APIまたはいずれかの派生APIを使用します。一方Mach実装はプリエンティブ実行モデルやスレッドをスケジューリングする機能など、すべてのスレッドに対する基本機能を提供するため、両者は互いに独立です。

リスト 2-2にアプリケーションで使用できるスレッド処理テクノロジーを示します。

表 1-2 スレッドテクノロジー

テクノロジー	説明
Cocoaスレッド	Cocoaでは、NSThreadクラスを使用してスレッドを実装します。CocoaのNSObjectには、新しいスレッドを作成するメソッドやすでに実行されているスレッドでコードを実行するメソッドも用意されています。詳細については、「 NSThreadの使用 」（24 ページ）および「 NSObjectを使用したスレッドの作成 」（27 ページ）を参照してください。

テクノロジー	説明
POSIXスレッド	POSIXスレッドには、スレッドを作成するためのC言語ベースのインターフェイスが用意されています。Cocoaアプリケーションを作成しないスレッドを作成する場合は、POSIXスレッドが最適な選択肢です。POSIXインターフェイスは比較的容易に使用でき、スレッドを十分柔軟に設定できます。詳細については、「 POSIXスレッドの使用 」（26 ページ）を参照してください。
マルチプロセッシングサービス	マルチプロセッシングサービスは、従来のC言語ベースのインターフェイスであり、古いバージョンのMac OSからの移行途上にあるアプリケーションで使用されます。このテクノロジーはMac OS Xでのみ利用できます。新規開発では使用しないでください。代わりに、NSThreadクラスまたはPOSIXスレッドを使用してください。ただし、このテクノロジーの詳細については、 <i>Multiprocessing Services Programming Guide</i> を参照してください。

アプリケーションレベルでは、すべてのスレッドがほかのプラットフォームの場合と基本的に同様に動作します。開始されたスレッドは、3つの主要状態のいずれかで実行されます。つまり、実行、実行可能、またはブロック状態です。現在実行中でなければ、スレッドはブロックされていて入力を待機しているか、実行の準備ができていて一方で実行がスケジューリングされていないかのいずれかです。スレッドはこれらの状態の間をさまざまに遷移した後で、終了されて終了状態になります。

新しいスレッドを作成するときは、スレッドのエントリポイント関数（Cocoaスレッドの場合はエントリポイントメソッド）を指定する必要があります。このエントリポイント関数は、スレッドで実行するコードで構成されます。関数から復帰するか、スレッドを明示的に終了した場合、スレッドは永久的に停止され、システムによって回収されます。スレッドを作成するとメモリおよび時間の面で比較的大きなコストがかかるため、エントリポイント関数で大量の処理を実行するか、実行ループを設定して作業を反復実行できるようにすることをお勧めします。

使用可能なスレッド処理テクノロジーおよび使用方法の詳細については、「[スレッドの管理](#)」（23 ページ）を参照してください。

実行ループ

実行ループは、スレッド上で非同期に届くイベントを管理するために使用されるインフラストラクチャの1つです。実行ループは、スレッドの1つ以上のイベントソースを監視することによって動作します。イベントが到着するとシステムがスレッドをスリープ解除させて、イベントを実行ループにディスパッチします。実行ループでは、指定されたハンドラにイベントをディスパッチします。処理できる状態になったイベントが存在していない場合、実行ループはスレッドをスリープ状態にします。

作成するすべてのスレッドで実行ループを使用する必要があるわけではありませんが、すべてのスレッドで実行ループを使用すれば、ユーザ体験が向上されます。実行ループを使用すると、最低限の量のリソースを使用する、長時間継続するスレッドを作成できるようになります。実行ループでは、実行する処理のないスレッドをスリープ状態にするため、ポーリングが不要になります。ポーリングはCPUサイクルを浪費するだけでなく、プロセッサ自体をスリープ状態しない結果、電力節約の妨げになります。

実行ループを設定するために必要な作業は、スレッドを起動し、実行ループオブジェクトの参照を取得し、イベントハンドラを組み込んで、実行ループに実行を指示することだけです。CocoaおよびCarbonのいずれによるインフラストラクチャでも、メインスレッドの実行ループは自動的に設定されます。ただし、長時間存続する二次スレッドを作成する場合は、そのスレッド用の実行ループを独自に設定する必要があります。

実行ループの例および使用方法の詳細については、「[実行ループ](#)」（35 ページ）を参照してください。

同期ツール

スレッドプログラミングに付随する危険の1つに、複数スレッド間でのリソースの競合があります。複数のスレッドで同じリソースを同時に使用または変更しようとして、問題が発生するおそれがあります。この問題は、共有リソースを徹底的になくし、各スレッド用に完全に独立した個別のリソースセットを割り当てることによっても解消できます。リソースを完全に分離した状態で維持できない場合は、ロック、条件変数、アトミック操作、およびその他の手法を使用してリソースへのアクセスを同期させる必要があります。

ロックでは、一度に1つのスレッドだけにコードの実行を許可する、強引な形のコードの保護を実装できます。最も一般的なロックのタイプは、**ミューテックス**とも呼ばれる相互排他ロックです。別のスレッドによって保持されているミューテックスを取得しようとしたスレッドは、その別のスレッドがのロックを解放するまでブロックされます。ミューテックスロックは、複数のシステムフレームワークでサポートされていますが、すべてのフレームワークで基礎にあるテクノロジーは同じです。さらに、Cocoaには、再帰など異なるタイプの動作をサポートするミューテックスロックの複数バリエーションが用意されています。使用可能なロックのタイプの詳細については、「[ロック](#)」（62 ページ）を参照してください。

システムでは、ロックに加え、条件変数をサポートしています。これにより、アプリケーション内でのタスクの適切な順序付けが実現されます。条件変数は門番として機能し、スレッドによって示されている条件が真になるまで、スレッドをブロックします。条件が真になるとスレッドは解放され、続行できるようになります。POSIXレイヤおよびFoundationフレームワークは、いずれも条件変数を直接サポートしています（オペレーションオブジェクトを使用すると、オペレーションオブジェクト間の依存関係を設定してタスクの実行を順序付けできます。この手法は、条件変数による動作にとってもよく似ています）。

ロックと条件変数は、非常に一般的な並行設計の手法ですが、データへのアクセスを保護および同期する別の手法としてアトミック操作があります。アトミック操作は、スカラデータ型の数学的演算または論理演算を実行できる場合に、ロックに代わる軽量な手段になります。アトミック操作では特別なハードウェア命令を使用して、ほかのスレッドにアクセスの機会を与える前に、変数の変更を必ず完了させます。

使用可能な同期ツールの詳細については、「[同期ツール](#)」（61 ページ）を参照してください。

スレッド間通信

優れた設計を行えば、必要な通信の量を最小化できますが、いくつかのポイントではスレッド間通信が必要です（スレッドでは、アプリケーションからの要求に基づく作業を実行します。アプリケーションでは、この結果を受け取る必要があるはずです）。スレッドでは、新規ジョブ要求を処理したり、進捗状況をアプリケーションのメインスレッドに報告したりする必要があります。このような場合にスレッド間で情報を取得する方法が必要とされます。スレッドは同じプロセス空間を共有しているため、通信方法の選択肢はたくさんあります。

スレッド間通信の方法はたくさんあり、それぞれに長所と短所があります。「スレッドのローカルストレージの設定」に、Mac OS Xで利用できる最も一般的な通信メカニズムを示します（メッセージキューおよびCocoa分散オブジェクトは例外です。これらのテクノロジーはiOSでも使用できます）。この表の手法は、下にいくほど複雑です。

表 1-3 通信メカニズム

メカニズム	説明
ダイレクトメッセージ	Cocoaアプリケーションでは、ほかのスレッドのセレクタを直接実行する機能をサポートしています。この機能は、本質的に、1つのスレッドでほかの任意のスレッドのメソッドを実行できることを意味します。この方法によって送信されたメッセージはターゲットスレッドのコンテキストで実行されるため、メッセージは自動的にターゲットスレッドでシリアライズされます。入力ソースについては、「 Cocoa実行セレクタソース 」（38 ページ）を参照してください。
グローバル変数、共有メモリ、およびオブジェクト	2つのスレッド間で情報を通信するもう1つの簡単な方法は、グローバル変数、共有オブジェクト、またはメモリの共有ブロックを使用することです。共有変数は高速で単純ですが、ダイレクトメッセージに比べて脆弱な点があります。コードの正確性を確保するには、ロックまたはほかの同期メカニズムによって共有変数を入念に保護する必要があります。保護しないと、競合状態、データの破損、またはクラッシュに至るおそれがあります。
条件変数	条件変数は、コードの特定の部分をスレッドに実行させる条件を制御するために使用できる同期ツールです。条件変数は門番と考えることができ、指定された条件と一致する場合のみスレッドを実行できます。条件変数の使用方法の詳細については、「 条件変数の使用 」（76 ページ）を参照してください。
実行ループのソース	カスタム実行ループソースは、スレッドでアプリケーション固有のメッセージを受け取るために開発者が設定するソースです。実行ループのソースはイベント駆動型であるため、実行ループのソースでは、実行する処理のないスレッドを自動的にスリープ状態にします。その結果スレッドの効率が向上されます。実行ループおよび実行ループのソースについては、「 実行ループ 」（35 ページ）を参照してください。
ポートおよびソケット	ポートベースの通信は、2つのスレッド間で通信する方法としてはさらに緻密ですが、非常に信頼性の高い手法でもあります。ポートおよびソケットは、ほかのプロセスやサービスなどの外部エンティティとの通信に使用できる点がさらに重要です。ポートは効率性のために実行ループのソースを使用して実装されています。したがって、ポートで待機中のデータがない場合、スレッドはスリープ状態になります。実行ループおよびポートベースの入力ソースについては、「 実行ループ 」を（35 ページ）参照してください。
メッセージキュー	マルチプロセッシングサービスは、受信データおよび発信データを管理するための先入れ先出し（FIFO）キューの抽象化を定義します。メッセージキューは単純で便利ですが、ほかの通信手法ほど効率的ではありません。メッセージキューの使用法の詳細については、『 <i>Multiprocessing Services Programming Guide</i> 』を参照してください。

メカニズム	説明
Cocoa分散オブジェクト	分散オブジェクトは、ポートベースの通信を高レベルで実装できるCocoaテクノロジーです。このテクノロジーはスレッド間通信に使用できますが、オーバーヘッドが大きいため使用は基本的に推奨されません。分散オブジェクトは、プロセス間を移動するオーバーヘッドがすでに大きい、ほかのプロセスとの通信に適しています。詳細については『 <i>Distributed Objects Programming Topics</i> 』を参照してください。

設計のヒント

以降の各セクションでは、コードの正確性を確保する形でスレッドを実装するために有用なガイドラインを示します。これらのガイドラインの一部には、スレッド化した独自のコードでパフォーマンスを向上できるヒントも含まれています。すべてのパフォーマンス上のヒントに共通していますが、コードの変更中および変更後に適切なパフォーマンス統計情報を収集する必要があります。

スレッドを明示的に作成しない

スレッドを作成するコードの手動作成は、面倒であり、エラーを発生させがちであるため、可能な限り避ける必要があります。Mac OS XおよびiOSでは、ほかのAPIによって並行処理を間接的にサポートしています。スレッドを独自に作成する代わりに、非同期API、GCD、またはオペレーションオブジェクトを使用して、その処理を実行することを検討してください。これらのテクノロジーでは、スレッド関連の作業を暗黙で実行し、正確な実行を保証します。さらに、GCDやオペレーションオブジェクトなどのテクノロジーは、現在のシステム負荷に基づいてアクティブスレッドの数を調整することにより、独自のコードによるよりもずっと効率的にスレッドを管理するように設計されています。GCDおよびオペレーションオブジェクトの詳細については、『*Concurrency Programming Guide*』を参照してください。

スレッドを適度にビジーにする

スレッドを手動で作成および管理する場合は、貴重なシステムリソースのスレッドによる消費に留意してください。できる限り、適度に長時間存続する生産的なタスクだけをスレッドに割り当ててください。また、大部分の時間をアイドル状態で費やすスレッドは、ためらわず終了させてください。スレッドはささいでない量のメモリを使用し、一部は固定メモリであるため、アイドルスレッドを解放することはアプリケーションのメモリ占有量の削減に役立つだけでなく、解放される物理メモリが増加してほかのシステムプロセスでそのメモリを使用できるようにもなります。

重要： アイドルスレッドを停止する前に、基礎測定値としてアプリケーションの現在のパフォーマンスを必ず測定してください。変更を試行してからもう一度測定して、変更によってパフォーマンスが低下せず、実際に向上したことを確認してください。

データ構造体を共有しない

スレッド関連のリソースを競合させないためのもっとも単純で簡単な方法は、プログラム内のスレッドに、そのスレッドで必要とするすべてのデータの独自のコピーを持たせることです。並列コードは、スレッド間で通信およびリソースの競合を最小化した場合にもっとも効果を発揮します。

マルチスレッドアプリケーションの作成には、困難が伴います。念入りに作業してコード内のすべての適切な場所で共有データ構造体をロックしても、コードは安全でない動作をする場合があります。たとえば、特定の順序で共有データ構造体に変更されることをコードで予期していると問題が発生する場合があります。これを修正するためにトランザクションベースのモデルに変更すれば、その結果、複数スレッド化によるパフォーマンスの向上が帳消しになる可能性があります。リソース競合の回避を最優先にすることで、パフォーマンスに優れた簡潔な設計を実現できることがよくあります。

スレッドおよびユーザインターフェイス

グラフィカルユーザインターフェイスを持つアプリケーションの場合は、アプリケーションのメインスレッドでユーザ関連イベントを受け取って、インターフェイスの更新を開始することをお勧めします。この手法は、ユーザイベントの処理とウィンドウコンテンツの描画に関連する同期の問題を回避するために有用です。Cocoaなどの一部のフレームワークでは、通常この動作を必要とします。ただし、その他のフレームワークであってもメインスレッドでこの動作を採用すれば、ユーザインターフェイスを管理するためのロジックが簡素化されるという利点があります。

ほかのスレッドからグラフィカル操作を実行するほうが都合の良い、少数の注目すべき例外があります。たとえば、QuickTime APIには、ムービーファイルのオープン、ムービーファイルのレンダリング、ムービーファイルの圧縮、および画像のインポートとエクスポートなど、二次スレッドから実行できる多数の操作が含まれています。同様に、CarbonおよびCocoaでは、二次スレッドを使用して、画像を作成および処理したり、その他の画像関連の計算を実行したりできます。これらの操作に二次スレッドを使用すると、パフォーマンスが大幅に向上する場合があります。ただし、グラフィカル操作の詳細が不明な場合は、メインスレッドで実行してください。

QuickTimeスレッドの安全性の詳細については、「[Technical Note TN2125: Thread-Safe Programming in QuickTime](#)」を参照してください。Cocoaスレッドの安全性の詳細については、「[スレッドの安全性のまとめ](#)」（79 ページ）を参照してください。Cocoaでの描画の詳細については、『[Cocoa Drawing Guide](#)』を参照してください。

終了時のスレッドの動作に注意する

プロセスは、デタッチされていないすべてのスレッドが終了するまで実行されます。デフォルトではアプリケーションのメインスレッドのみがデタッチされていないスレッドとして作成されますが、ほかのスレッドも同様に作成できます。デタッチされたスレッドによる処理はすべてオプションであると見なされるため、ユーザがアプリケーションを終了させた場合は、デタッチされたすべてのスレッドを即座に終了させることが適切な動作であると見なされます。一方、アプリケーション

ンでバックグラウンドスレッドを使用してディスクへのデータの保存などの重要な作業を実行する場合は、デタッチされていないスレッドとして作成することで、アプリケーション終了時のデータ損失を回避できます。

デタッチされていないスレッド（合流可能スレッド）を作成するには、独自の余分な作業が必要です。高レベルのスレッドテクノロジーの大部分は、デフォルトでは、合流可能スレッドを作成しないため、POSIX APIを使用してスレッドを作成しなければならない場合があります。さらに、アプリケーションのメインスレッドにコードを追加して、デタッチされていないスレッドが最終的に終了するときにそのスレッドを合流させる必要があります。合流可能スレッドの作成については、「[スレッドのデタッチ状態の設定](#)」（29 ページ）を参照してください。

Cocoaアプリケーションを作成する場合は、`applicationShouldTerminate:`デリゲートメソッドを使用してアプリケーションの終了をしばらく遅延させるか、まとめてキャンセルできます。終了を遅延させる場合、アプリケーションでは、すべてのクリティカルスレッドがタスクを終了するまで待機してから`replyToApplicationShouldTerminate:`メソッドを呼び出さなければならない場合があります。これらのメソッドの詳細については、『*NSApplication Class Reference*』を参照してください。

例外処理

例外処理メカニズムでは、例外がスローされたときに、現在のコールスタックに依存して必要なすべてのクリーンアップを実行します。各スレッドは独自のコールスタックを持つため、各スレッドでスレッド独自の例外をキャッチします。二次スレッドで例外をキャッチしなかった場合の影響は、メインスレッドで例外をキャッチしなかった場合の影響と同じです。スレッドのオーナープロセスが終了されます。キャッチされていない例外を異なるスレッドにスローして処理させることはできません。

メインスレッドなど別のスレッドに現在のスレッドの例外状況を通知する必要がある場合は、単に例外をキャッチして、状況を示すメッセージをほかのスレッドに送信します。例外をキャッチしたスレッドでは、採用しているモデルおよび実行しようとしている作業に応じて、処理の続行（可能な場合）、指示の待機、または単に終了することができます。

注： Cocoaの`NSException`オブジェクトは自己完結型のオブジェクトであり、キャッチしてからスレッド間で受け渡しできます。

例外ハンドラは自動的に作成される場合があります。たとえば、Objective-Cの`@synchronized`ディレクティブには、暗黙的な例外ハンドラが含まれています。

スレッドをきちんと終了させる

スレッドを終了させる一般的に最良な方法は、メインエントリーポイントルーチンの終わりまで実行させることです。スレッドを即座に終了させる関数がありますが、これらの関数は最後の手段としてのみ使用してください。自然なエンドポイントに到達する前にスレッドを終了させた場合、スレッドでは処理後のクリーンアップを実行できません。スレッドでメモリの割り当て、ファイルのオープン、およびその他のタイプのリソース取得を行っていた場合、コードでそれらのリソースを回収できないおそれがあり、その結果メモリリークなどほかの潜在的な問題が発生します。

スレッドを終了させる適切な方法の詳細については、「[スレッドの終了](#)」（32 ページ）を参照してください。

ライブラリのスレッドの安全性

アプリケーションの開発者はアプリケーションを複数スレッドで実行するかどうかを制御できますが、ライブラリ開発者には制御できません。ライブラリを開発する場合は、呼び出し元のアプリケーションはマルチスレッドであるか、随時マルチスレッドに切り替えられると想定する必要があります。したがって、常にロックを使用してコードのクリティカルセクションを保護する必要があります。

ライブラリを開発する場合に、アプリケーションがマルチスレッド化される場合に限りロックを作成することは賢明ではありません。コードをロックする必要がある場合は、ライブラリを使用する早い場所、できればライブラリを初期化するための何らかの明示的な呼び出しで、ロックオブジェクトを作成してください。スタティックライブラリ初期化関数を使用してそのようなロックを作成することもできますが、他に方法がない場合に限定してください。初期化関数を実行するとライブラリのロードに必要な時間が長くなるため、パフォーマンスを低下させるおそれがあります。

注：ライブラリ内でのミューテックスロックのロックとロック解除をバランスを保って呼び出すよう、常に留意してください。呼び出し元のコードにスレッドセーフな環境の準備を任せるのではなく、ライブラリのデータ構造体のロックにも留意する必要があります。

Cocoaライブラリを開発する場合は、`NSWillBecomeMultiThreadedNotification`のオブザーバを登録して、アプリケーションがマルチスレッド化されたときに通知を受け取ることができます。ただし、この通知はライブラリコードが呼び出されないうちにディスパッチされる可能性があるため、この通知を受け取れることを前提にしないでください。

第 1 章

スレッドプログラミングの概要

スレッドの管理

Mac OS XおよびiOSの各プロセス（アプリケーション）は、1つ以上のスレッドで構成されます。各スレッドは、アプリケーションのコードを介して単一の実行パスを表します。すべてのアプリケーションは単一のスレッドで開始されます。このスレッドがアプリケーションのmain関数を実行します。アプリケーションは、それぞれ特定の機能のコードを実行する、追加のスレッドを作成できます。

アプリケーションで作成する新しいスレッドは、アプリケーションのプロセス空間内の独立したエンティティになります。各スレッドは独自の実行スタックを持ち、カーネルによって実行時間を別々にスケジューリングされます。スレッドでは、ほかのスレッドおよびほかのプロセスとの通信や、I/O操作の実行など必要なすべての処理を実行できます。ただし、スレッドは同じプロセス空間内にあるため単一のアプリケーションに含まれるすべてのスレッドが同じ仮想メモリ空間を共有し、プロセス自体と同じアクセス権限を持ちます。

この章では、Mac OS XおよびiOSで使用可能なスレッドテクノロジーの概要とアプリケーションでそのテクノロジーを使用する方法の例を示します。

注： Mac OSのスレッド処理アーキテクチャの歴史および、スレッドのさらに基礎的な情報については、「Technical Note TN2028: Threading Architectures」を参照してください。

スレッドのコスト

スレッド処理には、メモリの使用とパフォーマンスの面でプログラム（およびシステム）に対して現実のコストがあります。各スレッドは、カーネルメモリ空間およびプログラムのメモリ空間の両方でメモリの割り当てを必要とします。スレッドを管理するためおよびスレッドのスケジューリングを調整するために必要なコア構造は、固定メモリを使用してカーネルに格納されます。スレッドのスタック空間およびスレッドごとのデータはプログラムのメモリ空間に格納されます。これらの構造の大部分は、スレッドを最初に作成するときに作成および初期化されます。スレッドは、カーネルとのやり取りを必要とするために比較的成本の大きいプロセスになることがあります。

表 2-1に、ユーザレベルのスレッドをアプリケーションに新規作成する場合の概算コストを示します。二次スレッドのスタック空間の割り当て量など、このコストの一部は設定可能です。スレッド作成の時間コストはおおまかな値であるため、相対的な比較のためにだけ使用してください。スレッド作成時間は、プロセッサの負荷、コンピュータの速度、および使用可能なシステムメモリとプログラムメモリの量によって大きく変わります。

表 2-1 スレッド作成コスト

項目	概算コスト	メモ
カーネルのデータ構造体	約1KB	このメモリはスレッドのデータ構造体および属性の格納に使用されます。このメモリの大部分は固定メモリとして割り当てられるため、ディスクにページングできません。

項目	概算コスト	メモ
スタック空間	512 KB (二次スレッド) 8 MB (Mac OS Xのメインスレッド) 1 MB (iOSのメインスレッド)	二次スレッドに許可される最小スタックサイズは16KBであり、スタックサイズは4KBの倍数である必要があります。このメモリの領域は、スレッドの作成時にプロセス空間に確保されますが、このメモリと関連付けられた実際のページは必要になってから作成されます。
作成時間	約90マイクロ秒	この値は、スレッド作成の初期呼び出しからスレッドのエントリポイントルーチンが実行を開始するまでの時間を示します。これらの数値は、2 GHzのCore Duoプロセッサと1 GBのRAMを搭載し、Mac OS X v10.5を稼動しているIntelベースのiMacでスレッドを作成して得られた平均値および中央値を分析して算出されています。

注： オペレーションオブジェクトの基礎にはカーネルのサポートがあるため、通常は、これより高速にスレッドを作成できます。オペレーションオブジェクトでは、スレッドを毎回最初から作成する代わりに、すでにカーネルに存在しているスレッドのプールを使用して割り当て時間が節約されます。オペレーションオブジェクトの使用法の詳細については、『*Concurrency Programming Guide*』を参照してください。

スレッド化されたコードを作成するときに考慮する必要のあるもう1つのコストとして、製造コストがあります。スレッド化されたアプリケーションを設計するには、アプリケーションのデータ構造体の編成方法を根本的に変更しなければならない場合があります。同期の使用を避けるためにこのような変更が必要になる可能性があります。設計の悪いアプリケーションでは、同期自体が多大なパフォーマンス上の不利益をもたらすおそれがあります。このデータ構造体を設計するためおよびスレッド化されたコードで検出された問題をデバッグするために、スレッド化されたアプリケーションの開発に長時間かかることがあります。ただし、これらのコストを逃れようとしてスレッドでロックを待機する時間や何も実行しない時間が余りに長くなれば、実行時により大きな問題が発生します。

スレッドの作成

低レベルのスレッドは、比較的簡単に作成できます。いずれの場合にも、スレッドのメインエントリポイントとして機能する関数またはメソッドが必要であり、使用可能ないずれかのスレッドルーチンを使用してスレッドを開始する必要があります。以降の各セクションでは、一般的なスレッドテクノロジーを使用する、基本的な作成処理について説明します。これらの手法を使用して作成するスレッドは、使用するテクノロジーによって決まる、一連のデフォルト属性を継承します。スレッドの設定方法の詳細については、「[スレッド属性の設定](#)」（28ページ）を参照してください。

NSThreadの使用

NSThreadクラスを使用してスレッドを作成する方法は2通りあります。

- `detachNewThreadSelector:toTarget:withObject:` クラスメソッドを使用して新しいスレッドを作成する。
- 新しいNSThreadオブジェクトを作成して、`start`メソッドを呼び出す（iOSおよびMac OS X v10.5以降でのみサポートされています）。

いずれの手法でも、デタッチされたスレッドがアプリケーションに作成されます。デタッチされたスレッドの場合、スレッドのリソースはスレッドの終了時にシステムによって自動的に回収されます。つまり、作成したスレッドを明示的に合流させる必要もありません。

`detachNewThreadSelector:toTarget:withObject:` メソッドはMac OS Xの全バージョンでサポートされているため、スレッドを使用する既存のCocoaアプリケーションでよく使用されています。新しいスレッドをデタッチする場合は、スレッドのエントリポイントとして使用するメソッドの名前（セレクタとして指定）、このメソッドを定義するオブジェクト、および起動時にスレッドに渡すすべてのデータを指定するだけです。次の例は、現在のオブジェクトのカスタムメソッドを使用してスレッドを作成する、このメソッドの基本呼び出しを示します。

```
[NSThread detachNewThreadSelector:@selector(myThreadMainMethod:) toTarget:self
withObject:nil];
```

Mac OS X v10.5の前までは、スレッドを作成する前にNSThreadクラスを使用していました。NSThreadオブジェクトを取得して一部のスレッド属性にアクセスできる一方で、スレッドを実行した後でスレッド自体からだけアクセス可能でした。Mac OS X v10.5では、対応する新しいスレッドをすぐに作成しないNSThreadオブジェクト作成がサポートされるようになりました（このサポートはiOSでも有効です）。このサポートにより、スレッドを開始する前にさまざまなスレッド属性を取得および設定できるようになりました。このスレッドオブジェクトを後で使用して、実行中のスレッドを参照することもできるようになりました。

Mac OS X v10.5以降でNSThreadオブジェクトを初期化するには、`initWithTarget:selector:object:`メソッドを使用する方法が簡単です。このメソッドは、`detachNewThreadSelector:toTarget:withObject:`メソッドとまったく同じ情報を受け取り、新しいNSThreadインスタンスを初期化するために使用します。ただし、スレッドの開始は行いません。スレッドを開始するには、次の例に示すようにスレッドオブジェクトの`start`メソッドを明示的に呼び出します。

```
NSThread* myThread = [[NSThread alloc] initWithTarget:self
                                     selector:@selector(myThreadMainMethod:)
                                     object:nil];

[myThread start]; // 実際にスレッドを作成
```

注： `initWithTarget:selector:object:`メソッドを使用する代わりにNSThreadをサブクラスにして`main`メソッドをオーバーライドすることもできます。メソッドをオーバーライドする方法は、スレッドのメインエントリポイントを実装する場合などに使用します。詳細については、『*NSThread Class Reference*』のサブクラス処理のメモを参照してください。

スレッドを実行しているNSThreadオブジェクトがあり、このスレッドにメッセージを送信するには、たとえば、アプリケーションのほぼすべてのオブジェクトにある`performSelector:onThread:withObject:waitUntilDone:`メソッドを使用します。メインスレッド以外のスレッドでセレクタを実行する機能は、Mac OS X v10.5からサポートされるようになった、スレッド間通信のための便利な機能です（このサポートはiOSでも有効です）。この手法を使用して送信したメッセージは、ほかのスレッドで通常の実行ループ処理の一部として直接実行されます（もちろん、ターゲットスレッドで実行ループを実行している必要があるということを意味しては

いません。「[実行ループ](#)」(35 ページ)を参照してください)。この方法で通信する場合でもなんらかの同期を必要とすることはありますが、スレッド間に通信ポートを設定する方法と比べて簡単です。

注： `performSelector:onThread:withObject:waitUntilDone:` メソッドは、スレッド間の散発的な通信に適していますが、時間制約が厳しいか、頻繁なスレッド間の通信には使用しないでください。

これ以外の選択可能なスレッド通信の一覧については、「[スレッドのデタッチ状態の設定](#)」(29 ページ)を参照してください。

POSIXスレッドの使用

Mac OS XおよびiOSでは、POSIXスレッドAPIを使用して、C言語ベースでスレッドを作成できます。このテクノロジーは、CocoaアプリケーションやCocoa Touchアプリケーションなどあらゆるタイプのアプリケーションで実際に使用でき、複数プラットフォーム向けのソフトウェアを作成する場合に適しています。スレッドの作成に使用するPOSIXルーチンは、その機能にふさわしく、`pthread_create`という名前です。

リスト 2-1に、POSIX呼び出しを使用してスレッドを作成する、2個のカスタム関数を示します。`LaunchThread`関数は、メインルーチンを`PosixThreadMainRoutine`関数で実装する、新しいスレッドを作成します。POSIXのデフォルトでは合流可能としてスレッドが作成されるため、この例では、デタッチされたスレッドを作成するようにスレッドの属性を変更しています。デタッチされたスレッドにすれば、スレッドの終了直後にシステムでスレッドのリソースを回収するチャンスがあります。

リスト 2-1 C言語でのスレッドの作成

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

void* PosixThreadMainRoutine(void* data)
{
    // 必要な作業を実行します。

    return NULL;
}

void LaunchThread()
{
    // POSIXルーチンを使用してスレッドを作成します。
    pthread_attr_t attr;
    pthread_t      posixThreadID;
    int            returnVal;

    returnVal = pthread_attr_init(&attr);
    assert(!returnVal);
    returnVal = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    assert(!returnVal);

    int threadError = pthread_create(&posixThreadID, &attr,
    &PosixThreadMainRoutine, NULL);
```

```

returnVal = pthread_attr_destroy(&attr);
assert(!returnVal);
if (threadError != 0)
{
    // エラーを報告します。
}
}

```

ここにリストしたコードをソースファイルの1つに追加してLaunchThread関数を呼び出すと、デタッチされた新しいスレッドがアプリケーションに作成されます。このコードを使用して作成された新しいスレッドは、当然ながら何の作業も実行しません。スレッドは起動されるとほとんど瞬時に終了されます。何らかの処理を実際に実行するコードをPosixThreadMainRoutine関数に追加すれば、もう少し面白みのあるコードになります。実行する作業をスレッドで確実に判別できるようにするために、スレッドの作成時にデータのポインタを渡すことができます。このポインタは、pthread_create関数の最後のパラメータとして渡します。

新しく作成したスレッドからアプリケーションのメインスレッドへの通信で情報を戻すには、ターゲットスレッド間に通信パスを確立する必要があります。C言語ベースのアプリケーションには、ポート、条件変数、または共有メモリの使用など、スレッド間通信の方法が複数あります。長時間存続するスレッドの場合は、アプリケーションのメインスレッドでスレッドのステータスを確認したり、アプリケーションを終了するときにクリーンアップしてからシャットダウンしたりするために、通常は何らかのスレッド間通信メカニズムを設定する必要があります。

POSIXスレッド関数の詳細については、pthreadのmanページを参照してください。

NSObjectを使用したスレッドの作成

iOSおよびMac OS X v10.5以降では、新しいスレッドを作成し、作成したスレッドの任意のメソッドを実行することがすべてのオブジェクトで可能です。performSelectorInBackground:withObject:メソッドでは、デタッチされた新しいスレッドを作成し、指定されたメソッドを新しいスレッドのエントリポイントとして使用します。たとえば、変数myObjで示すオブジェクトがあり、このオブジェクトにあるdoSomethingというメソッドをバックグラウンドスレッドで実行する場合は、次のようなコードを使用します。

```
[myObj performSelectorInBackground:@selector(doSomething) withObject:nil];
```

このメソッドの呼び出し結果は、現在のオブジェクト、セレクタ、およびパラメータオブジェクトをパラメータとしてNSThreadのdetachNewThreadSelector:toTarget:withObject:メソッドを呼び出した結果と同じです。デフォルト設定を使用して新しいスレッドがすぐに作成され、実行を開始します。セレクタ内では、ほかのスレッドと同様にスレッドを設定する必要があります。たとえば、ガベージコレクションを使用しない場合に自動解放プールを設定したり、スレッドの実行ループを使用したりする場合に実行ループを設定する必要があります。新しいスレッドの設定方法の詳細については、「[スレッド属性の設定](#)」（28 ページ）を参照してください。

その他のスレッド処理テクノロジーの使用

低レベルのスレッドの作成に推奨されるテクノロジーはPOSIXルーチンおよびNSThreadクラスですが、Mac OS Xには、他にも使用可能なC言語ベースのテクノロジーがあります。そのうち、検討に値する唯一のテクノロジーがマルチプロセッシングサービスです。マルチプロセッシングサービスは、POSIXスレッドを使用して実装されています。マルチプロセッシングサービスは、もともと以前のバージョンのMac OS向けに開発され、その後Mac OS XのCarbonアプリケーションで利用できるよ

うになりました。マルチプロセッシングサービスを使用する既存のコードがある場合、そのコードは引き続き使用できますが、スレッド関連のコードをPOSIXに移植することも検討してください。マルチプロセッシングサービスは、iOSでは使用できません。

マルチプロセッシングサービスの使用方法の詳細については、『*Multiprocessing Services Programming Guide*』を参照してください。

CocoaアプリケーションでのPOSIXスレッドの使用

Cocoaアプリケーションでスレッドを作成するときの主なインターフェイスはNSThreadクラスですが、代わりにPOSIXスレッドを適宜使用できます。たとえば、POSIXスレッドを使用するコードがすでにあり、このコードをそのまま使用する場合にPOSIXスレッドを使用します。CocoaアプリケーションでPOSIXスレッドを使用しない場合でも、Cocoaとスレッドのやり取りに留意し、以降の各セクションで説明するガイドラインに従う必要があります。

Cocoaフレームワークの保護

Cocoaフレームワークでは、マルチスレッドアプリケーションについては、ロックおよびその他の形態の内部同期を使用してアプリケーションを正しく動作させます。一方、単一スレッドの場合にこれらのロックによってパフォーマンスを低下させないために、Cocoaでは、アプリケーションでNSThreadクラスを使用して新しいスレッドを作成するまでロックを作成しません。POSIXスレッドルーチンだけを使用してスレッドを作成した場合は、アプリケーションのマルチスレッド化を認識するために必要な情報がCocoaに伝わりません。この場合、Cocoaフレームワークに関連する操作が不安定になったり、アプリケーションがクラッシュしたりするおそれがあります。

NSThreadクラスを使用して単一のスレッドを作成し、このスレッドをすぐに終了させるだけで、複数スレッドを使用するつもりであることをCocoaに通知できます。スレッドのエントリポイントでは、何も実行する必要はありません。NSThreadを使用してスレッドを作成するという動作だけあれば、必要なロックをCocoaフレームワークに配置させるために十分です。

アプリケーションのマルチスレッド化をCocoaが認識しているかどうか不明な場合は、NSThreadのisMultiThreadedメソッドを使用して確認できます。

POSIXおよびCocoaのロックの混合

同じアプリケーションでPOSIXのロックとCocoaのロックを混用しても安全です。Cocoaのロックオブジェクトおよび条件オブジェクトは、本質的には、POSIXのミューテックスおよび条件変数のラッパーにすぎません。ただし、特定の1つのロックについては、常に同じインターフェイスを使用してそのロックを作成および操作する必要があります。つまり、pthread_mutex_init関数を使用して作成したミューテックスをCocoa NSLockオブジェクトで操作することはできず、この逆も同様です。

スレッド属性の設定

スレッドを作成した後や、場合によっては作成する前に、スレッド環境のさまざまな部分を設定する必要がある場合があります。以降の各セクションでは、実施できる変更の一部についておよびその変更を必要とするケースについて説明します。

スレッドのスタックサイズの設定

作成する新しいスレッドごとに、そのスレッドのスタックとして機能する一定量のメモリが、システムによってプロセス空間に割り当てられます。このスタックはスタックフレームを管理するだけではありません。スレッドのすべてのローカル変数はスタックに宣言されます。スレッドに割り当てられるメモリの量については、「[スレッドのコスト](#)」（23 ページ）を参照してください。

特定のスレッドのスタックサイズを変更する場合は、スレッドを作成する前に変更する必要があります。スタックサイズを設定する何らかの方法がすべてのスレッド処理テクノロジーに用意されていますが、NSThreadを使用するスタックサイズの設定はiOSおよびMac OS X v10.5以降でのみ使用可能です。表 2-2 に、各テクノロジーのさまざまなオプションを示します。

表 2-2 スレッドのスタックサイズの設定

テクノロジー	オプション
Cocoa	iOSおよびMac OS X v10.5以降で、NSThreadオブジェクトの割り当てと初期化を行います（detachNewThreadSelector: toTarget: withObject: メソッドを使用しない）。スレッドオブジェクトのstartメソッドを呼び出す前にsetStackSize:メソッドを使用して、新しいスタックサイズを指定します。
POSIX	新しいpthread_attr_t構造体を作成し、pthread_attr_setstacksize関数を使用してデフォルトのスタックサイズを変更します。スレッドを作成するときに、pthread_create関数に属性を渡します。
マルチプロセッシングサービス	スレッドを作成するときに、適切なスタックサイズ値をMPCreateTask関数に渡します。

スレッドのローカルストレージの設定

各スレッドは、スレッド内のすべての場所からアクセスできる、キーと値のペアの辞書を保持しています。この辞書は、スレッドが実行されている間継続させる必要のある情報を格納するために使用できます。たとえば、スレッドの実行ループが複数回反復される間継続させる必要のある状態情報を格納するために使用できます。

CocoaとPOSIXではスレッド辞書の格納方法が異なるため、この2つのテクノロジーに対する呼び出しを組み合わせることはできません。ただし、スレッドコード内で1方のテクノロジーだけを使用する限り、最終結果に大差はありません。Cocoaの場合は、NSThreadオブジェクトのthreadDictionaryメソッドを使用してNSMutableDictionaryオブジェクトを取得します。このオブジェクトに対してスレッドに必要なすべてのキーを追加できます。POSIXの場合は、pthread_setspecific関数およびpthread_getspecific関数を使用して、スレッドのキーおよび値を設定および取得します。

スレッドのデタッチ状態の設定

高レベルのスレッドテクノロジーの大部分は、デフォルトでは、デタッチされたスレッドを作成します。デタッチされたスレッドは、スレッドの完了後すぐにシステムでスレッドのデータ構造体を解放できるため、通常よく使われます。デタッチされたスレッドでは、プログラムとの明示的なや

り取りも必要としません。スレッドから結果を取得する方法は、開発者が決定できます。これに対し、合流可能スレッドのリソースは、別のスレッドによって明示的にそのスレッドが合流されるまで回収されません。この過程で合流先のスレッドがブロックされる可能性もあります。

合流可能スレッドは子スレッドのようなものと見なすことができます。これらのスレッドは、まだ独立したスレッドとして実行されますが、合流可能スレッドのリソースをシステムで回収するには、まず別のスレッドと合流させる必要があります。合流可能スレッドは、終了するスレッドから別のスレッドにデータを渡す手段にもなります。合流可能スレッドでは、データポインタまたはその他の戻り値を終了直前に`pthread_exit`関数に渡すことができます。別のスレッドでは、`pthread_join`関数を呼び出してこのデータを要求できます。

重要： デタッチされたスレッドはアプリケーションの終了に合わせて即座に終了できますが、合流可能スレッドは即座に終了できません。プロセスを終了するには、まず各合流可能スレッドを合流させる必要があります。したがって、ディスクへのデータの保存など中断しないほうがよい重要な作業をスレッドで実行する場合は、一般に、合流可能スレッドのほうが適切です。

合流可能スレッドは、POSIXスレッドを使用することによってのみ作成できます。POSIXで作成されるスレッドは、デフォルトでは、合流可能です。スレッドをデタッチまたは合流可能にするには、スレッドを作成する前に`pthread_attr_setdetachstate`関数を使用してスレッド属性を変更します。スレッドの開始後は、`pthread_detach`関数を呼び出すことにより、合流可能スレッドをデタッチされたスレッドに変更できます。これらのPOSIXスレッド関数の詳細については、`pthread`のmanページを参照してください。スレッドと合流させる方法の詳細については、`pthread_join`のmanページを参照してください。

スレッド優先度の設定

作成するすべての新しいスレッドには、デフォルトの優先度が関連付けられます。カーネルのスケジューリングアルゴリズムでは、スレッド優先度を勘案して実行するスレッドが決定されます。その結果、優先度の低いスレッドよりも優先度の高いスレッドの実行ほうが開始されやすくなります。優先度が高いからといってスレッドに一定の実行時間が保証されるわけではありません。優先度の低いスレッドに比べ、スケジューラによって選択される確率が高まるだけです。

重要： 通常は、スレッドの優先度をデフォルト値のままにすることをお勧めします。一部のスレッドの優先度を上げると、優先度の低いスレッドがなかなか選択されない割合も高くなります。互いにやり取りする必要のある優先度の高いスレッドと優先度の低いスレッドを含むアプリケーションでは、優先度の低いスレッドがなかなか選択されない結果、ほかのスレッドがブロックされてパフォーマンス上のボトルネックが生じるおそれがあります。

スレッドの優先度を変更する場合は、CocoaおよびPOSIXの両方に、そのための方法が用意されています。Cocoaスレッドの場合は、`NSThread`の`setThreadPriority:`クラスメソッドを使用して、現在実行中のスレッドの優先度を設定できます。POSIXスレッドの場合は、`pthread_setschedparam`関数を使用します。詳細については、『*NSThread Class Reference*』を参照するか、`pthread_setschedparam`のmanページを参照してください。

スレッドエントリルーチンの作成

スレッドエントリポイントルーチンの構造は、Mac OS Xでもほかのプラットフォームと大体同じです。データ構造体を初期化した後で任意の作業を実行するか必要に応じて実行ループを設定し、その後スレッドのコードの終了時にクリーンアップを行います。設計によっては、エントリルーチンを作成するために必要なステップがさらに増えます。

自動解放プールの作成

Objective-Cフレームワークでリンクされるアプリケーションの場合、通常は、それぞれのスレッドの内部で1つ以上の自動解放プールを作成する必要があります。オブジェクトの保持および解放をアプリケーションで処理するマネージドモデルを使用するアプリケーションの場合は、スレッドから自動解放されるすべてのオブジェクトを自動解放プールでキャッチします。

マネージドメモリモデルの代わりにガベージコレクションを使用するアプリケーションの場合は、自動解放プールを作成しなくてもかまいません。ガベージコレクションを使用するアプリケーションに自動解放プールが存在していても悪影響はなく、通常は無視されるだけです。必要であれば、コードモジュールでガベージコレクションとマネージドメモリモデルの両方をサポートできます。その場合、自動解放プールは、マネージドメモリモデルのコードをサポートするために存在している必要があります。ガベージコレクションを有効にしてアプリケーションを実行する場合には無視されるだけです。

マネージドメモリモデルを使用するアプリケーションの場合は、スレッドエントリルーチンでまず自動解放プールを作成する必要があります。同様に、この自動解放プールの破棄は、スレッドの最後の処理として実行する必要があります。このプールによって自動解放されたオブジェクトは確実にキャッチされます。ただし、オブジェクトはスレッド自体が終了するまで解放されません。リスト 2-2に、自動解放プールを使用する基本的なスレッドエントリルーチンの構造を示します。

リスト 2-2 スレッドエントリポイントルーチンの定義

```
- (void)myThreadMainRoutine
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init]; // 最上位プール

    // スレッドの作業を実行します。

    [pool release]; // プール内のオブジェクトを解放します。
}
```

最上位の自動解放プールではスレッドの終了までオブジェクトを解放しないため、長時間存続するスレッドの場合はさらに自動解放プールを作成して、もっと頻繁にオブジェクトを解放する必要があります。たとえば、実行ループを使用するスレッドで、実行ループの1回ごとに自動解放プールを作成および解放します。オブジェクトを頻繁に解放すれば、パフォーマンス上の問題の原因となるアプリケーションのメモリ占有量の過剰な増大を防止できます。ただし、パフォーマンス関連の動作については常に同様ですが、コードの実際のパフォーマンスを測定しながら、自動解放プールの使用方法を適宜調整してください。

メモリ管理および自動解放プールの詳細については、『*Memory Management Programming Guide*』を参照してください。

例外ハンドラの設定

アプリケーションで例外をキャッチおよび処理する場合は、発生する可能性のあるすべての例外をスレッドコードでキャッチできるよう準備する必要があります。例外の発生しそうな場所で例外を処理できれば最も理想的ですが、スローされた例外をスレッドでキャッチできない場合は、アプリケーションが終了します。スレッドのエントリルーチンの最後にtryおよびcatchを組み込めば、すべての不明な例外をキャッチして適切な応答を指定できます。

Xcodeでプロジェクトを構築する場合は、C++またはObjective-Cの例外処理のスタイルを使用できます。Objective-Cで例外を発生およびキャッチする方法の設定については、『*Exception Programming Topics*』を参照してください。

実行ループの設定

別スレッドで実行するコードを作成する場合、2通りの実装方法があります。まったく割り込みなし、ほとんど割り込みなしで長時間実行される1つのタスクとしてスレッドのコードを作成し、タスクの終了時にスレッドを終了させる方法が1つ。スレッドをループに入れ、要求を受け取った時点でスレッドに要求を処理させる方法がもう1つです。最初の方法では、コードの特別な設定は不要であり、実行する必要がある処理をそのまま開始します。一方、2番目の方法では、スレッドの実行ループを設定する必要があります。

MacOSXおよびiOSでは、組み込まれているサポートによって、すべてのスレッドに対する実行ループを実装できます。Cocoa、Carbon、およびUIKitの場合、アプリケーションのメインスレッドの実行ループは自動的に開始されますが、二次スレッドを作成する場合は、実行ループを設定して手動で開始する必要があります。

実行ループの使用および設定については、「[実行ループ](#)」（35 ページ）を参照してください。

スレッドの終了

スレッドを終了させるときは、スレッドのエントリポイントルーチンを自然に終了させる方法をお勧めします。Cocoa、POSIX、およびマルチプロセッシングサービスには、スレッドを直接強制終了するルーチンがありますが、それらのルーチンではできるだけ使用しないでください。スレッドを強制終了すると、スレッドによるクリーンアップが妨げられます。スレッドによって割り当てられたメモリがリークしたり、スレッドで使用中のその他のリソースが適切にクリーンアップされなかったりするおそれがあり、後で問題を発生させる可能性があります。

操作の途中でスレッドを終了させる必要があるような場合は、キャンセルメッセージや終了メッセージに応答するようにスレッドをあらかじめ設計しておく必要があります。このことは、長時間実行される操作の場合に作業を定期的に停止して、そのようなメッセージの到着を検査することを意味するかもしれません。スレッドの終了を要求するメッセージが到着すれば、それをきっかけに必要なすべてのクリーンアップを実行してからスレッドを適切に終了できます。到着しなければ、そのまま作業に戻って、次の一まとまりのデータを処理できます。

キャンセルメッセージに応答する1つの方法として、実行ループの入力ソースを使用してキャンセルメッセージを受け取る方法があります。リスト 2-3に、スレッドのメインエントリルーチンにこのコードを実装するときの構造を示します（この例は、メインループの部分だけを示しており、自動解放プールを設定するステップや、実際に実行する作業を設定するステップは含まれていません）。この例では、いずれかのスレッドから受け取るメッセージなどである場合が多いカスタム入

カソースを実行ループに組み込みます。入力ソースの設定については、「[実行ループソースの設定](#)」（46 ページ）を参照してください。スレッドでは総作業量の一部を実行した後で、実行ループを短時間実行して、入力ソースにメッセージが届いているかどうかを検査します。到着していない場合は即座に実行ループが終了されて、次の一まとまりの作業に対してループが続行されます。ハンドラではexitNowローカル変数に直接アクセスできないため、終了条件は、スレッド辞書のキーと値のペアを介して通信されます。

リスト 2-3 長時間処理に組み込んだ終了条件の検査

```
- (void)threadMainRoutine
{
    BOOL moreWorkToDo = YES;
    BOOL exitNow = NO;
    NSRunLoop* runLoop = [NSRunLoop currentRunLoop];

    // BOOL型のexitNowをスレッド辞書に追加します。
    NSMutableDictionary* threadDict = [[NSThread currentThread] threadDictionary];
    [threadDict setValue:[NSNumber numberWithInt:exitNow]
    forKey:@"ThreadShouldExitNow"];

    // 入力ソースを組み込みます。
    [self myInstallCustomInputSource];

    while (moreWorkToDo && !exitNow)
    {
        // 大量の作業のうち、一まとまりの作業を実行します。
        // 作業を完了した場合は、Boolean型のmoreWorkToDoの値を変更します。

        // 実行ループを実行しますが、入力ソースで作動を待機していなければすぐにタイムアウト
        // します。
        [runLoop runUntilDate:[NSDate date]];

        // 入力ソースハンドラによってexitNowの値が変更されたかどうかを確認します。
        exitNow = [[threadDict valueForKey:@"ThreadShouldExitNow"] boolValue];
    }
}
```


実行ループ

実行ループは、スレッドに関連する基本インフラストラクチャの一部です。**実行ループ**は、作業のスケジューリングと受信イベントの受信側の調整に使用するイベント処理ループです。実行ループは、実行する作業がある間スレッドをビジーにし、作業がないときはスレッドをスリープ状態にすることを目的としています。

実行ループ管理には自動化されていない部分があります。やはり、適切なタイミングで実行ループを開始して受信イベントに応答するようにスレッドのコードを設計する必要があります。CocoaおよびCore Foundationの両方に、スレッドの実行ループの設定および管理に役立つ**実行ループオブジェクト**が用意されています。アプリケーションでこれらのオブジェクトを明示的に作成する必要はありません。アプリケーションのメインスレッドも含む各スレッドに、関連付けられた実行ループオブジェクトが存在します。ただし、二次スレッドだけは、スレッドの実行ループを明示的に実行する必要があります。CarbonアプリケーションおよびCocoaアプリケーションの両方で、アプリケーション起動プロセスの一環としてメインスレッドが自動的に設定され、実行ループを実行します。

以降の各セクションでは、実行ループの詳細とアプリケーションに合わせて設定する方法を説明します。実行ループオブジェクトの追加情報については、『*NSRunLoop Class Reference*』および『*CFRunLoop Reference*』を参照してください。

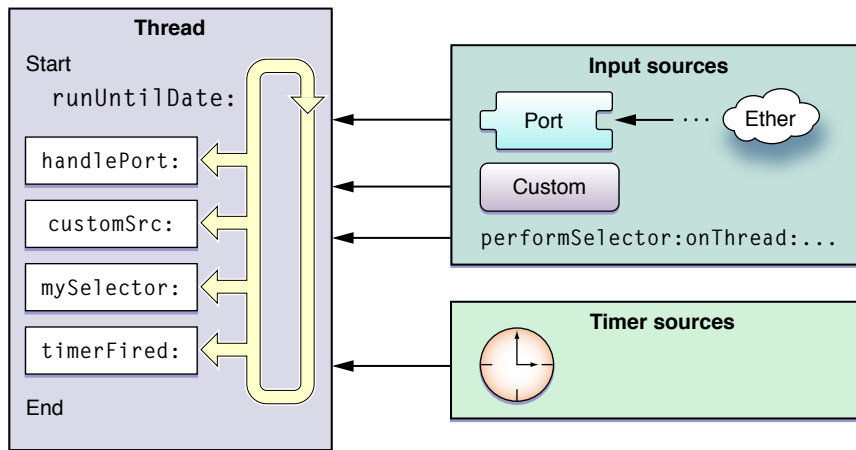
実行ループの構造

実行ループは、ほぼその名前から推測されるとおりのものです。スレッドは、実行ループに入り、受信イベントに응答してイベントハンドラを実行するために実行ループを使用します。実行ループの実際のループ部分を実装する制御文はコードで作成します。つまり、実行ループを駆動するwhileループまたはforループをコードで作成します。このループ内で実行ループオブジェクトを使用して、イベントを受け取って組み込んだハンドラを呼び出す、イベント処理コードを「実行」します。

実行ループは2つの異なるタイプのソースからイベントを受け取ります。**入力ソース**は、通常は別のスレッドまたは別のアプリケーションからのメッセージである、非同期イベントを配信します。**タイマーソース**は、スケジューリングされた時間または反復する間隔で発生する同期イベントを配信します。いずれのタイプのソースでも、アプリケーション固有のハンドラルーチンを使用して、到着したイベントを処理します。

図 3-1に、実行ループの概念的な構造およびさまざまなソースを示します。入力ソースは非同期イベントに対応するハンドラに配信します。その結果、スレッドに関連付けられたNSRunLoopオブジェクトで呼び出すrunUntilDate:メソッドが終了されます。タイマーソースは対応するハンドラルーチンにイベントを配信しますが、実行ループは終了されません。

図 3-1 実行ループの構造およびソース



実行ループでは、入力のソースの処理に加え、実行ループの動作に関する通知の生成も行います。登録した**実行ループオブザーバ**では、これらの通知を受け取り、これを使用してスレッドに対する追加の処理を実行できます。スレッドへの実行ループオブザーバの組み込みには、**Core Foundation**を使用できます。

以降の各セクションでは、実行ループの構成要素および実行ループが稼動するモードについて詳しく説明します。イベント処理のさまざまな時期に生成される通知についても説明します。

実行ループモード

実行ループモードは、監視対象の入力ソースおよびタイマーのコレクションであり、通知対象の実行ループオブザーバのコレクションです。実行ループを実行するたびに、明示的または暗黙的に特定の実行「モード」を指定します。その実行ループパスでは、そのモードと関連付けられているソースだけが監視対象になり、イベントの配信を許可されます（同様に、そのモードと関連付けられているオブザーバだけに実行ループの進行状況が通知されます）。ほかのモードと関連付けられているソースは、実行ループにおける後続のパスが適切なモードになるまで、すべての新しいイベントを保持します。

コードでは、名前でモードを識別します。**Cocoa**および**Core Foundation**の両方に、デフォルトのモードおよびよく使用するモードと、コードでこれらのモードを指定するための文字列が定義されています。モード名を表すカスタム文字列を指定するだけで、カスタムモードを定義できます。カスタムモードに割り当てる名前は任意ですが、カスタムモードの内容は決まっています。有効なモードを作成するには、いずれのモードにも、1つ以上の入力ソース、タイマー、または実行ループオブザーバを追加する必要があります。

モードを使用して、実行ループにおける特定のパスで、不要なソースからのイベントを除外できます。通常はシステム定義の「デフォルト」モードで実行ループを実行することになります。一方、モーダルパネルを「モーダル」モードで実行する場合もあります。このモードでは、モーダルパネルに関連するソースだけがイベントをスレッドに配信します。二次スレッドの場合は、カスタムモードを使用して、優先度の高い操作の実行中に優先度の低いソースからのイベントが配信されないようにできます。

注：モードでは、イベントのタイプではなくソースに基づいてイベントを区別します。たとえば、マウスダウンイベントだけやキーボードイベントだけと合致するモードは使用しません。モードを使用すると、異なる1組のポートの監視、タイマーの一時的な中断、または監視対象であるソースおよび実行ループオブザーバの変更などを実行できます。

表 3-1に、CocoaおよびCore Foundationで定義されている標準のモードおよびそのモードを使用する状況を示します。名前カラムは、モードを指定するためにコードで使用する実際の定数です。

表 3-1 定義済みの実行ループモード

モード	名前	説明
デフォルト	NSDefaultRunLoopMode (Cocoa) kCFRunLoopDefaultMode (Core Foundation)	デフォルトモードは、大部分の操作で使用するモードです。大半の時間はこのモードを使用して、実行ループを開始したり、入力ソースを設定したりします。
接続	NSConnectionReplyMode (Cocoa)	Cocoaでは、NSConnectionオブジェクトと組み合わせてこのモードを使用することにより、返信を監視します。このモードを独自に使用する必要はめったにありません。
モーダル	NSModalPanelRunLoopMode (Cocoa)	Cocoaでは、モーダルパネルを対象とするイベントの識別にこのモードを使用します。
イベントトラッキング	NSEventTrackingRunLoopMode (Cocoa)	Cocoaでは、マウスドラッグループおよびその他の種類のユーザインターフェイストラッキングループの間、受信イベントを制限するためにこのモードを使用します。
共通モード	NSRunLoopCommonModes (Cocoa) kCFRunLoopCommonModes (Core Foundation)	設定可能なよく使用するモードのグループです。入力ソースをこのモードに関連付けると、グループ内の各モードにも関連付けられます。Cocoaアプリケーションの場合、デフォルトでは、デフォルトモード、モーダルモード、およびイベントトラッキングモードがこのセットに含まれます。当初、Core Foundationが含むモードは、デフォルトモードだけです。CFRunLoopAddCommonMode関数を使用してこのセットにカスタムモードを追加できます。

入力ソース

入力ソースは、イベントを非同期でスレッドに配信します。イベントのソースは入力ソースのタイプによって異なり、通常は次の2カテゴリーのいずれかです。ポートベースの入力ソースは、アプリケーションのMachポートを監視します。カスタム入力ソースは、イベントのカスタムソースを監視します。実行ループに関する限り、入力ソースがポートベースなのかカスタムなのかは重要ではありません。通常は、両方のタイプの入力ソースが、そのまま使用できる状態でシステムに実装されています。この2つのソースはシグナルを受信する方法だけが異なります。ポートベースのソースにはカーネルによって自動的にシグナルが送信されます。カスタムソースには別のスレッドから明示的にシグナルを送信する必要があります。

作成した入力ソースを実行ループの1つ以上のモードに割り当てます。モードは、任意の時点においていずれの入力ソースを監視するのかに影響します。大半の時間は、デフォルトモードで実行ループを実行しますが、カスタムモードも指定できます。現在監視されているモードで実行されていない入力ソースの場合、その入力ソースが生成するすべてのイベントは、合致するモードで実行ループが実行されるようになるまで保持されます。

以降の各セクションでは、一部の入力ソースについて説明します。

ポートベースソース

CocoaおよびCore Foundationでは、組み込まれたサポートによって、ポートベースの入力ソースを作成したり、ポート関連のオブジェクトおよび関数を使用したりできます。たとえば、Cocoaでは、入力ソースを直接作成する必要はまったくありません。ポートオブジェクトを作成し、NSPortのメソッドを使用してこのポートを実行ループに追加するだけです。必要な入力ソースの作成および設定はポートオブジェクトによって処理されます。

Core Foundationの場合は、ポートおよび実行ループソースの両方を手動で作成する必要があります。いずれの場合も、ポート不透過型 (CFMachPortRef、CFMessagePortRef、またはCFSocketRef) に関連付けられた関数を使用して、適切なオブジェクトを作成します。

カスタムポートベースソースの設定方法の例については、「[ポートベース入力ソースの設定](#)」 (53 ページ) を参照してください。

カスタム入力ソース

カスタム入力ソースを作成するには、Core FoundationでCFRunLoopSourceRef不透過型に関連付けられている関数を使用する必要があります。カスタム入力ソースを設定するには、複数のコールバック関数を使用します。Core Foundationでは、さまざまなポイントでこれらの関数を呼び出して、ソースの設定およびすべての受信イベントの処理を行い、ソースが実行ループから除去された場合にソースを破棄します。

イベントが到着したときのカスタムソースの動作を定義するだけでなく、イベント配信メカニズムも定義する必要があります。ソースのこの部分を別のスレッドで実行して、入力ソースへのデータの提供と、データが処理できる状態になったときのシグナルの送信を行います。任意のイベント配信メカニズムを採用できますが、複雑になりすぎないようにしてください。

カスタム入力ソースを作成する方法の例については、「[カスタム入力ソースの定義](#)」 (46 ページ) を参照してください。カスタム入力ソースの参考情報については、『*CFRunLoopSource Reference*』も参照してください。

Cocoa実行セクタソース

Cocoaでは、ポートベースソースに加え、任意のスレッドでセクタを実行できるカスタム入力ソースを定義します。実行セクタ要求は、ポートベースソース同様にターゲットスレッド上でシリアルライズされます。その結果、1つのスレッドで複数メソッドを実行することに伴う潜在的な多くの同期問題が軽減されます。ポートベースソースと異なり、実行セクタソースは、セクタを実行した後で実行ループから自動的に除去されます。

注： Mac OS X v10.5の前までは、主にメインスレッドにメッセージを送信するために実行セクタソースを使用しましたが、Mac OS X v10.5以降およびiOSでは、任意のスレッドにメッセージを送信するために実行セクタソースを使用できます。

別のスレッド上のセクタを実行する場合は、ターゲットスレッドにアクティブ実行ループが存在する必要があります。独自に作成するスレッドの場合であれば、スレッドでの実行ループの明示的な開始をセクタで待機することを意味します。一方、メインスレッドでは独自の実行ループを開始するため、アプリケーションデリゲートの`applicationDidFinishLaunching:`メソッドをアプリケーションで呼び出した直後から、スレッド上の呼び出しを発行できます。実行ループでは、ループの反復ごとに1個ずつ処理するのではなく、ループのたびにキューにあるすべての実行セクタ呼び出しを処理します。

表 3-2に、ほかのスレッド上のセクタを実行するために使用できる、`NSObject`に定義されているメソッドをリストします。これらのメソッドは`NSObject`に宣言されているため、POSIXスレッドなど、Objective-Cのオブジェクトにアクセスできるすべてのスレッドから使用できます。これらのメソッドでは、セクタを実行するために、新しいスレッドの実際の作成を行いません。

表 3-2 ほかのスレッド上にあるセクタの実行

メソッド	説明
<code>performSelectorOnMainThread: withObject: waitUntilDone:</code> <code>performSelectorOnMainThread: withObject: waitUntilDone:modes:</code>	スレッドの次の実行ループサイクルの間に、アプリケーションのメインスレッドで指定されたセクタを実行します。これらのメソッドにより、セクタが実行されるまで現在のスレッドをブロックできるようになります。
<code>performSelector: onThread:withObject: waitUntilDone:</code> <code>performSelector: onThread:withObject: waitUntilDone:modes:</code>	<code>NSThread</code> オブジェクトを取得している任意のスレッドで、指定されたセクタを実行します。これらのメソッドにより、セクタが実行されるまで現在のスレッドをブロックできるようになります。
<code>performSelector: withObject: afterDelay:</code> <code>performSelector: withObject: afterDelay:inModes:</code>	次の実行ループサイクルの間に、オプションの遅延期間が過ぎてから、現在のスレッド上の指定されたセクタを実行します。次の実行ループサイクルまでセクタの実行が待機されるため、これらのメソッドによる遅延は現在実行中のコードからの自動最小遅延になります。複数のセクタがキューに入っている場合は、キューに入れられた順序で次々に実行されます。
<code>cancelPreviousPerformRequestsWithTarget:</code> <code>cancelPreviousPerformRequestsWithTarget: selector:object:</code>	<code>performSelector: withObject: afterDelay:</code> メソッドまたは <code>performSelector: withObject: afterDelay:inModes:</code> メソッドを使用して現在のスレッドに送信したメッセージをキャンセルできます。

この各メソッドの詳細については、『*NSObject Class Reference*』を参照してください。

タイマーソース

タイマーソースは、事前設定された時刻にスレッドにイベントを同期配信します。タイマーは、スレッド自体に何らかの処理を実行するよう通知する方法の1つです。たとえば、検索フィールドでタイマーを使用して、ユーザからの連続的なキーストロークの間隔が一定時間空いたときに自動検索を開始できます。この遅延時間があると、検索を開始する前に、目的の検索文字列を必要なだけ入力する機会がユーザに提供されます。

タイマーは時間ベースの通知を生成しますが、リアルタイムメカニズムではありません。タイマーは、入力ソース同様、実行ループの特定のモードと関連付けられます。実行ループで監視中のモードでないタイマーは、タイマーでサポートしているいずれかのモードで実行ループを実行するまで作動しません。同様に、実行ループでハンドラルーチンを実行中に作動したタイマーは、実行ループが終わりまで実行されて、ハンドラルーチンを次回起動するまで待機します。実行ループが実行されない場合、タイマーは作動しません。

タイマーはイベントを1回だけ生成するか繰り返し生成するように設定できます。反復タイマーは、実際の作動時刻ではなくスケジューリングされた作動時刻に基づいて自動的に再スケジューリングされます。たとえば、特定の時刻とそれ以降5秒ごとに作動するようスケジューリングされているタイマーの場合、実際の作動時刻が遅延したとしてもスケジューリングされた作動時刻は常に元の5秒間隔になります。作動時刻が大幅に遅延してスケジューリングされた作動時刻に作動しなかったタイマーがある場合は、この作動していない期間に対して1度だけ作動します。作動していなかった期間に対して作動したタイマーは、スケジューリングされている次の作動時刻用に再スケジューリングされます。

タイマーソースの設定の詳細については、「[タイマーソースの設定](#)」（51 ページ）を参照してください。参考情報については『*NSTimer Class Reference*』または『*CFRunLoopTimer Reference*』を参照してください。

実行ループオブザーバ

該当する非同期イベントまたは同期イベントが発生したときに作動する実行ループソースとは対象的に、実行ループオブザーバは実行中の実行ループの特別な場所で作動します。実行ループオブザーバは、特定のイベントを処理するためのスレッドの準備や、スリープ状態になる前のスレッドの準備に使用できます。実行ループオブザーバは、実行ループ内の次のイベントと関連付けることができます。

- 実行ループに入る。
- 実行ループでタイマーを処理しようとしている。
- 実行ループで入力ソースを処理しようとしている。
- 実行ループがスリープ状態になろうとしている。
- 実行ループはスリープ解除されているが、実行ループをスリープ解除させたイベントをまだ処理していない。
- 実行ループから抜ける。

実行ループオブザーバはCocoaアプリケーションおよびCarbonアプリケーションの両方に追加できますが、実行ループオブザーバを定義して実行ループに追加するには、**Core Foundation**を使用する必要があります。実行ループオブザーバを作成するには、CFRunLoopObserverRef不透過型の新しいインスタンスを作成します。この型は、カスタムコールバック関数および関連するアクティビティを追跡します。

実行ループオブザーバもタイマー同様、1回または繰り返し使用できます。1回だけのオブザーバは作動後に実行ループから自動で除去されます。これに対し、反復オブザーバは接続されたままになります。オブザーバを1回実行するのか繰り返し実行するのかは、作成時に指定します。

実行ループオブザーバを作成する方法の例については、「[実行ループの設定](#)」（43 ページ）を参照してください。参考情報については、『*CFRunLoopObserver Reference*』を参照してください。

イベントの実行ループのシーケンス

スレッドの実行ループは、実行されるたびに未処理イベントを処理し、接続されているすべてのオブザーバに対して通知を生成します。この処理の順序は、次のとおりに極度に限定されています。

1. 実行ループに入ったことをオブザーバに通知します。
2. 作動可能なすべてのタイマーが作動しようとしていることをオブザーバに通知します。
3. ポートベースでないすべての入力ソースが作動しようとしていることをオブザーバに通知します。
4. ポートベースでない作動可能な状態のすべての入力ソースを作動させます。
5. ポートベースの入力ソースが作動可能になっていて作動を待機している場合は、イベントをすぐに処理します。ステップ9に進みます。
6. スレッドがスリープ状態になろうとしていることをオブザーバに通知します。
7. 次のいずれかのイベントが発生するまでスレッドをスリープ状態にします。
 - ポートベースの入力ソースに対応するイベントが到着する。
 - タイマーが作動する。
 - 実行ループに設定されているタイムアウト値を経過する。
 - 実行ループが明示的にスリープ解除される。
8. スレッドが今スリープ解除したことをオブザーバに通知します。
9. 保留中のイベントを処理します。
 - ユーザ定義のタイマーが作動した場合は、タイマーイベントを処理し、ループを再開します。ステップ2に進みます。
 - 入力ソースが作動した場合は、イベントを配信します。
 - 実行ループが明示的にスリープ解除されてまだタイムアウトしていない場合はループを再開します。ステップ2に進みます。

10. 実行ループが終了したことをオブザーバに通知します。

タイマーおよび入力ソースのオブザーバ通知は、対応するイベントが実際に発生する前に配信されるため、通知時刻と実際のイベントの時刻に差があることがあります。これらのイベント間のタイミングが極めて重要な場合は、スリープ通知およびスリープ解除通知を使用して、実際のイベント間のタイミングを相関させることができます。

タイマーおよびその他の定期イベントは実行ループを実行するときに配信されるため、実行ループを実行するまでこれらのイベントは配信されません。この動作の代表的な例としては、ループを開始して、アプリケーションからイベントを繰り返し要求することにより、マウストラッキングルーチンを実装する場合があります。本来のようにイベントのディスパッチをアプリケーションにまかせるのではなく、コードがイベントを直接処理しているため、マウストラッキングルーチンが終了してアプリケーションに制御が戻るまでアクティブタイマーは作動できません。

実行ループは、実行ループオブジェクトを使用して明示的にスリープ解除できます。ほかのイベントによって実行ループがスリープ解除することもあります。たとえば、ポートベースでない別の入力ソースを追加すると、ほかの何らかのイベントが発生するまで待たずに入力ソースをすぐ処理できるよう、実行ループがスリープ解除されます。

実行ループを使用する状況

実行ループは、アプリケーションで二次スレッドを作成する場合に限り、明示的に実行する必要があります。アプリケーションのメインスレッドの実行ループは、欠かせないインフラストラクチャの1つです。したがって、メインアプリケーションループを実行してこのループ自動的に開始するコードがCocoaおよびCarbonの両方に用意されています。iOSのUIApplication（またはMac OS XのNSApplication）のrunメソッドは、通常の起動シーケンスの一環としてアプリケーションのメインループを開始します。同様に、RunApplicationEventLoop関数はCarbonアプリケーションのメインループを開始します。Xcodeテンプレートプロジェクトを使用してアプリケーションを作成する場合、これらのルーチンを明示的に呼び出す必要はありません。

二次スレッドの場合は、実行ループが必要かどうかを判断し、必要に応じて、独自に実行ループを設定および開始する必要があります。スレッドの実行ループを開始しなくてよい場合があります。たとえば、長時間実行される何らかの規定のタスクをスレッドによって実行する場合、通常は、実行ループを開始する必要はありません。実行ループは、頻繁にスレッドとやり取りする状況で使用することを目的としています。実行ループ開始する必要があるのは、たとえば、次のような処理を実行する場合です。

- ポートまたはカスタム入力ソースを使用してほかのスレッドと通信する。
- スレッド上でタイマーを使用する。
- Cocoaアプリケーションで、任意のperformSelector...メソッドを使用する。
- スレッドを保持して定期的なタスクを実行する。

実行ループを使用する場合は簡単に設定できます。ただし、すべてのスレッドプログラミングに共通していますが、二次スレッドを適宜終了させることを検討する必要があります。スレッドは、いずれの場合にも、強制終了するのではなくきちんと終了させるほうが適切です。実行ループを設定および終了する方法については、「[実行ループオブジェクトの使用](#)」（43 ページ）を参照してください。

実行ループオブジェクトの使用

実行ループオブジェクトは、入力ソース、タイマー、および実行ループオブザーバを実行ループに追加してからこの実行ループを実行するための主要インターフェイスになります。すべてのスレッドには、実行ループオブジェクトが1つ関連付けられています。Cocoaの場合、このオブジェクトは、NSRunLoopクラスのインスタンスです。CarbonアプリケーションまたはBSDアプリケーションの場合、これはCFRunLoopRef不透過型へのポインタです。

実行ループオブジェクトの取得

現在のスレッドの実行ループを取得するには、次のいずれかを使用します。

- Cocoaアプリケーションの場合は、NSRunLoopのcurrentRunLoopクラスメソッドを使用してNSRunLoopオブジェクトを取得する。
- CFRunLoopGetCurrent関数を使用する。

toll-free bridgingに対応している型ではありませんが、必要に応じてNSRunLoopオブジェクトからCFRunLoopRef不透過型を取得できます。NSRunLoopクラスには、Core Foundationルーチンに渡せるCFRunLoopRef型を戻すgetCFRunLoopメソッドが定義されています。NSRunLoopオブジェクトとCFRunLoopRef不透過型の両方で同じ実行ループを参照するため、必要に応じてこれらのオブジェクトの呼び出しを混合できます。

実行ループの設定

二次スレッドで実行ループを実行するには、まず、1つ以上の入力ソースまたはタイマーを実行ループに追加する必要があります。監視対象のソースがない実行ループを実行しようとした場合は、すぐに終了されます。実行ループにソースを追加する方法の例については、「[実行ループソースの設定](#)」（46 ページ）を参照してください。

ソースを組み込むだけでなく、実行ループオブザーバを組み込んで実行ループのさまざまな実行段階を検出するために使用することもできます。実行ループオブザーバを組み込むには、CFRunLoopObserverRef不透過型を作成し、CFRunLoopAddObserver関数を使用して実行ループに追加できます。実行ループオブザーバの作成には、Cocoaアプリケーションでも、Core Foundationを使用する必要があります。

リスト 3-1に、実行ループオブザーバを実行ループに接続するスレッドのメインルーチンを示します。この例では、実行ループオブザーバの作成方法を示すことを目的としているため、このコードでは、単に実行ループのすべてのアクティビティを監視するように実行ループオブザーバを設定します。基本ハンドラルーチン（ここでは省略）では、タイマー要求を処理するときに実行ループのアクティビティをそのまま記録します。

リスト 3-1 実行ループオブザーバの作成

```
- (void)threadMain
{
    // このアプリケーションはガベージコレクションを使用するため、自動解放プールは不要です。
    NSRunLoop* myRunLoop = [NSRunLoop currentRunLoop];
```

```

// 実行ループオブザーバを作成して、実行ループに接続します。
CFRunLoopObserverContext context = {0, self, NULL, NULL, NULL};
CFRunLoopObserverRef observer =
CFRunLoopObserverCreate(kCFAllocatorDefault,
                        kCFRunLoopAllActivities, YES, 0, &myRunLoopObserver, &context);

if (observer)
{
    CFRunLoopRef cfLoop = [myRunLoop getCFRunLoop];
    CFRunLoopAddObserver(cfLoop, observer, kCFRunLoopDefaultMode);
}

// タイマーを作成して、スケジューリングします。
[NSTimer scheduledTimerWithTimeInterval:0.1 target:self
                        selector:@selector(doFireTimer:) userInfo:nil repeats:YES];

NSInteger loopCount = 10;
do
{
    // 実行ループを10回実行して、タイマーを作動させます。
    [myRunLoop runUntilDate:[NSDate dateWithTimeIntervalSinceNow:1]];
    loopCount--;
}
while (loopCount);
}

```

長時間存続するスレッドの実行ループを設定する場合は、入力ソースを1つ以上追加してメッセージを受け取ることをお勧めします。タイマー1つだけを接続した実行ループに入ることはできますが、作動したタイマーは通常は無効になり、実行ループが終了されます。反復タイマーを接続すれば実行ループを長期間実行できますが、タイマーを定期的に作動してスレッドを作動状態にしておく必要があります。これは、事実上、形を変えたポーリングです。これに対して、イベントの発生を待機する入力ソースの場合は、イベントが発生するまでスレッドをスリープ状態にできます。

実行ループの開始

実行ループを開始させる必要があるのは、アプリケーションの二次スレッドの場合だけです。実行ループは、監視対象の入力ソースまたはタイマーを1つ以上持つ必要があります。1つも接続されていない場合、実行ループは即座に終了されます。

実行ループは、次のような複数の方法で開始できます。

- 無条件
- 時間制限の設定
- 特定のモード

無条件に実行ループに入る方法が一番簡単ですが、一番お勧めしない方法でもあります。無条件に実行ループを実行するとスレッドは永久ループに入ります。永久ループでは、実行ループ自体をコードでほとんど制御できません。入力ソースおよびタイマーの追加および除去は可能ですが、実行ループを停止する方法は強制終了のみです。カスタムモードで実行ループを実行する方法もあります。

実行ループを条件なしで実行する代わりに、タイムアウト値を指定して実行ループを実行することをお勧めします。タイムアウト値を使用する場合、実行ループはイベントが到着するか、割り当てられた時間が経過するまで実行されます。イベントが到着した場合は、ハンドラにそのイベントがディスパッチされて処理され、実行ループが終了します。その後、実行ループを再開して次のイベントを処理できます。イベントが到着しないで割り当てられた時間が経過した場合は、単に実行ループを再開することもできれば、必要な管理処理に時間をかけることもできます。

タイムアウト値の他に、特定のモードを使用して実行ループを実行することもできます。モードとタイムアウト値は相互に排他的でなく、実行ループを開始するときに両方を使用できます。モードによって実行ループにイベントを配信するソースのタイプが制限されます。詳細については、「[実行ループモード](#)」(36 ページ)を参照してください。

リスト 3-2 にスレッドのメインエントリールーチンの骨格を示します。この例の主要部分は実行ループの基本構造を示します。要約すると、入力ソースおよびタイマーを実行ループに追加してからルーチンの1つを繰り返し呼び出すことによって実行ループを開始しています。実行ループルーチンから戻るときに、スレッドを適切に終了するための何らかの条件が生じているかどうかを検査します。この例では、戻った結果を検査して、実行ループが終了した理由を判別できるように **Core Foundation** の実行ループルーチンを使用します。**Cocoa** を使用しており、戻り値を検査する必要がある場合は、`NSRunLoop` クラスにあるメソッドを使用して実行ループを実行することもできます (`NSRunLoop` クラスのメソッドを呼び出す実行ループの例については、[リスト 3-14](#) (54 ページ) を参照してください)。

リスト 3-2 実行ループの実行

```
- (void)skeletonThreadMain
{
    // ガベージコレクションを使用しない場合は、ここで自動解放プールを設定します。
    BOOL done = NO;

    // ソースまたはタイマーを実行ループに追加し、その他のすべての設定を行います。

    do
    {
        // 実行ループを開始しますが、各ソースの処理後に復帰します。
        SInt32 result = CFRunLoopRunInMode(kCFRunLoopDefaultMode, 10, YES);

        // ソースが明示的に実行ループを停止したか、ソースもタイマーも
        // ない場合は、続行して終了します。
        if ((result == kCFRunLoopRunStopped) || (result == kCFRunLoopRunFinished))
            done = YES;

        // その他のすべての終了条件を検査し、
        // 必要に応じてdone変数を設定します。
    }
    while (!done);

    // コードをクリーンアップします。割り当てられたすべての自動解放プールを必ず解放してください。
}
```

実行ループは繰り返し実行できます。つまり、`CFRunLoopRun`、`CFRunLoopRunInMode`、または任意の `NSRunLoop` メソッドを呼び出して、入力ソースまたはタイマーのハンドラルーチンから実行ループを開始できます。その場合は、外側の実行ループで使用中のモードなど任意のモードを使用して、ネストされた実行ループを実行できます。

実行ループの終了

実行ループでイベントを処理する前に実行ループを終了させる方法は2通りあります。

- タイムアウト値付きで実行するように実行ループを設定する。
- 実行ループに停止を指示する。

管理可能であれば、通常はタイムアウト値を使用することをお勧めします。タイムアウト値を指定した場合、実行ループでは、実行ループオブザーバに対する通知の配信などの通常の処理を完了してから終了できます。

CFRunLoopStop関数を使用して実行ループを明示的に停止した場合、タイムアウト同様の結果になります。実行ループは、残りのすべての実行ループ通知を送信してから終了します。この手法は、無条件で開始した実行ループで利用できる点が異なります。

実行ループの入力ソースおよびタイマーを除去することによって実行ループを終了できる場合もありますが、この方法による実行ループの停止には信頼性がありません。一部のシステムルーチンでは、入力ソースを実行ループに追加することによって、必要なイベントを処理します。この入力ソースをコードで認識できない場合があるためこの入力ソースを除去できず、その結果実行ループが終了されません。

スレッドの安全性と実行ループオブジェクト

スレッドの安全性は実行ループの操作に使用するAPIによって異なります。Core Foundationの関数は、通常スレッドセーフであり、すべてのスレッドから呼び出せます。ただし、実行ループの設定を変更する操作を実行する場合は、やはり、できるだけその実行ループを含むスレッドから操作することをお勧めします。

CocoaのNSRunLoopクラスは、Core Foundationの対応する関数とは異なり、本質的にスレッドセーフなわけではありません。NSRunLoopクラスを使用して実行ループを変更する場合は、その実行ループを含むスレッド自体から変更する必要があります。別のスレッドに含まれる実行ループに入力ソースまたはタイマーを追加すると、コードがクラッシュしたり予期しない動作になったりします。

実行ループソースの設定

以降の各セクションでは、CocoaおよびCore Foundationの両方について、さまざまなタイプの入力ソースを設定する方法の例を示します。

カスタム入力ソースの定義

カスタム入力ソースを作成する場合は、次の内容を定義する必要があります。

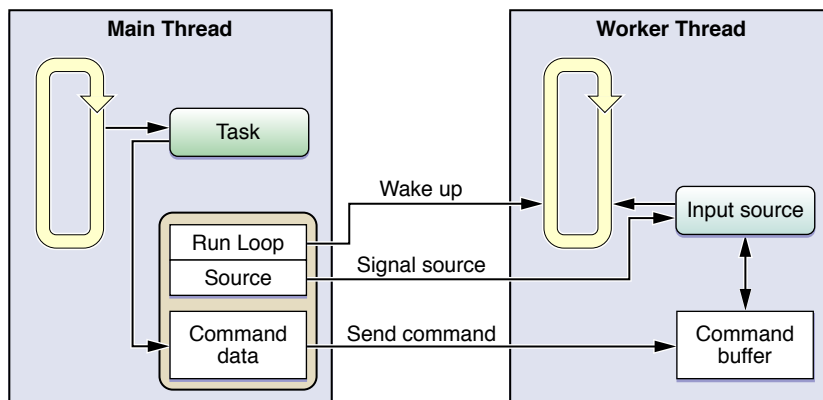
- 入力ソースで処理する情報。
- 入力ソースとの通信方法を関連するクライアントに通知するスケジューラルーチン。

- 任意のクライアントから送信された要求を実行するハンドラルーチン。
- 入力ソースを無効にするキャンセルルーチン。

カスタム入力ソースは、カスタム情報を処理するために作成します。したがって、実際の設定は柔軟な設計になっています。スケジューラルーチン、ハンドラルーチン、およびキャンセルルーチンは、カスタム入力ソースでほぼ必ず必要になる重要なルーチンです。一方、入力ソースのこれ以外の動作は、大部分がこれらのハンドラルーチンの外部で行われます。たとえば、入力ソースにデータを渡すメカニズムや入力ソースが存在していることをほかのスレッドに通知するメカニズムを決めるのは開発者です。

図 3-2 にカスタム入力ソースの設定のサンプルを示します。この例では、アプリケーションのメインスレッドで、入力ソースへの参照、この入力ソース用のカスタムコマンドバッファ、およびこの入力ソースを組み込む実行ループを管理します。ワーカースレッドに渡すタスクがある場合、メインスレッドでは、ワーカースレッドでタスクを開始するために必要なすべての情報とともにコマンドをコマンドバッファに送ります（メインスレッドでもワーカースレッドの入力ソースでもこのコマンドバッファにアクセスできるため、アクセスを同期させる必要があります）。コマンドが送られると、メインスレッドが入力ソースにシグナルを送り、ワーカースレッドの実行ループをスリープ解除させます。スリープ解除コマンドを受け取った実行ループは、入力ソースのハンドラを呼び出します。ハンドラはコマンドバッファで見つかったコマンドを処理します。

図 3-2 カスタム入力ソースの操作



以降の各セクションでは、前の図に示したカスタム入力ソースの実装について説明し、実装する必要のありそうなコードの主要部を示します。

入力ソースの定義

カスタム入力ソースを定義する場合は、Core Foundationルーチンを使用して実行ループソースを設定し、このソースを実行ループに接続する必要があります。基本ハンドラはC言語ベースの関数ですが、これらの関数のラッパーを作成し、Objective-CまたはC++を使用してコードの本体を実装することもできます。

図 3-2（47 ページ）に示した入力ソースでは、Objective-Cのオブジェクトを使用して、コマンドバッファの管理と実行ループとの連携を行います。リスト 3-3 にこのオブジェクトの定義を示します。RunLoopSource オブジェクトは、コマンドバッファを管理し、このバッファを使用してほかのスレッドからメッセージを受け取ります。このリストには RunLoopContext オブジェクトの定義も

含まれています。このオブジェクトは、純粹に単なるコンテナオブジェクトで、RunLoopSourceオブジェクトおよび実行ループの参照をアプリケーションのメインスレッドに渡すために使用されません。

リスト 3-3 カスタム入力ソースオブジェクトの定義

```
@interface RunLoopSource : NSObject
{
    CFRunLoopSourceRef runLoopSource;
    NSMutableArray* commands;
}

- (id)init;
- (void)addToCurrentRunLoop;
- (void)invalidate;

// ハンドラメソッド
- (void)sourceFired;

// 処理するコマンドを登録するためのクライアントインターフェイス
- (void)addCommand:(NSInteger)command withData:(id)data;
- (void)fireAllCommandsOnRunLoop:(CFRunLoopRef)runloop;

@end

// CFRunLoopSourceRefコールバック関数
void RunLoopSourceScheduleRoutine (void *info, CFRunLoopRef rl, CFStringRef mode);
void RunLoopSourcePerformRoutine (void *info);
void RunLoopSourceCancelRoutine (void *info, CFRunLoopRef rl, CFStringRef mode);

// RunLoopContextは、入力ソースを登録するときに使用されるコンテナオブジェクトです。
@interface RunLoopContext : NSObject
{
    CFRunLoopRef      runLoop;
    RunLoopSource*    source;
}
@property (readonly) CFRunLoopRef runLoop;
@property (readonly) RunLoopSource* source;

- (id)initWithSource:(RunLoopSource*)src andLoop:(CFRunLoopRef)loop;
@end
```

Objective-Cのコードで入力ソースのカスタムデータを管理しますが、入力ソースを実行ループに接続するにはC言語ベースのコールバック関数が必要です。最初の関数はリスト 3-4に示す関数で、実際に実行ループソースを実行ループに接続するときに呼び出されます。この入力ソースのクライアントは1つ（メインスレッド）だけであるため、入力ソースでは、スケジューラ関数を使用してメッセージを送信することにより、このスレッドのアプリケーションデリゲートに入力ソースを登録します。入力ソースとの通信が必要となった場合、デリゲートでは、RunLoopContextオブジェクトに含まれている情報を使用して通信します。

リスト 3-4 実行ループソースのスケジューリング

```
void RunLoopSourceScheduleRoutine (void *info, CFRunLoopRef rl, CFStringRef mode)
{
    RunLoopSource* obj = (RunLoopSource*)info;
```

```

AppDelegate* del = [AppDelegate sharedApplication];
RunLoopContext* theContext = [[RunLoopContext alloc] initWithSource:obj
andLoop:r1];

[del performSelectorOnMainThread:@selector(registerSource:)
                      withObject:theContext waitUntilDone:NO];
}

```

最も重要なコールバックルーチンの1つとして、入力ソースにシグナルが送信されたときにカスタムデータを処理するために使用されるルーチンがあります。リスト3-5に、RunLoopSourceオブジェクトと関連付けられたコールバックルーチン実行を示します。この関数では、単にsourceFiredメソッドに要求を転送して作業を実行させます。コマンドバッファに含まれているすべてのコマンドが処理されます。

リスト 3-5 入力ソースに含まれている作業の実行

```

void RunLoopSourcePerformRoutine (void *info)
{
    RunLoopSource* obj = (RunLoopSource*)info;
    [obj sourceFired];
}

```

CFRunLoopSourceInvalidate関数を使用して実行ループから入力ソースを除去すると、入力ソースのキャンセルルーチンが呼び出されます。このルーチンは、この入力ソースが有効でなくなっており、入力ソースへのすべての参照を除去する必要があることを、クライアントに通知するために使用できます。リスト3-6に、RunLoopSourceオブジェクトに登録されているキャンセルコールバックルーチンを示します。この関数では、もう1つのRunLoopContextオブジェクトをアプリケーションデリゲートに送信しますが、今回は、実行ループソースへの参照を除去するようデリゲートに指示します。

リスト 3-6 入力ソースの無効化

```

void RunLoopSourceCancelRoutine (void *info, CFRunLoopRef r1, CFStringRef mode)
{
    RunLoopSource* obj = (RunLoopSource*)info;
    AppDelegate* del = [AppDelegate sharedApplication];
    RunLoopContext* theContext = [[RunLoopContext alloc] initWithSource:obj
andLoop:r1];

    [del performSelectorOnMainThread:@selector(removeSource:)
                      withObject:theContext waitUntilDone:YES];
}

```

注： アプリケーションデリゲートのregisterSource:メソッドおよびremoveSource:メソッドのコードを「[入力ソースのクライアントとの連携](#)」（50 ページ）に示します。

実行ループへの入力ソースの組み込み

リスト3-7に、RunLoopSourceクラスのinitメソッドおよびaddToCurrentRunLoopメソッドを示します。initメソッドでは、実行ループに実際に接続する必要のあるCFRunLoopSourceRef不透過型を作成します。コンテキスト依存の情報としてRunLoopSourceオブジェクト自体を渡すことにより、コールバックルーチンはオブジェクトのポインタを持ちます。入力ソースは、ワーカースレッ

ドで`addToCurrentRunLoop`メソッドを呼び出すときになってから組み込まれます。この時点で`RunLoopSourceScheduleRoutine`コールバック関数が呼び出されます。入力ソースが実行ループに追加されると、スレッドで実行ループを実行して、入力ソースを待機できます。

リスト 3-7 実行ループソースの組み込み

```
- (id)init
{
    CFRunLoopSourceContext context = {0, self, NULL, NULL, NULL, NULL, NULL,
                                      &RunLoopSourceScheduleRoutine,
                                      RunLoopSourceCancelRoutine,
                                      RunLoopSourcePerformRoutine};

    runLoopSource = CFRunLoopSourceCreate(NULL, 0, &context);
    commands = [[NSMutableArray alloc] init];

    return self;
}

- (void)addToCurrentRunLoop
{
    CFRunLoopRef runLoop = CFRunLoopGetCurrent();
    CFRunLoopAddSource(runLoop, runLoopSource, kCFRunLoopDefaultMode);
}
```

入力ソースのクライアントとの連携

入力ソースを有用なものとするには、入力ソースを操作する必要があり、別のスレッドから入力ソースにシグナルを送る必要があります。入力ソースは、入力ソースと関連付けられているスレッドで実行する作業が生じるまで、スレッドをスリープ状態にすることに要点があります。したがって、アプリケーションのほかのスレッドに、入力ソースを認識させ、入力ソースと通信する方法を与える必要があります。

クライアントに入力ソースを通知する方法としては、入力ソースを初めて実行ループに組み込むときに登録要求を送信する方法などがあります。必要な数のクライアントに入力ソースを登録することもできれば、限られた数の集中型のエージェントだけに登録して、このエージェントから関連するクライアントに入力ソースを登録することもできます。リスト 3-8 に、アプリケーションデリゲートによって定義され、`RunLoopSource` オブジェクトのスケジューラ関数が呼び出されるときに呼び出される登録メソッドを示します。このメソッドは、`RunLoopSource` オブジェクトで準備した `RunLoopContext` オブジェクトを受け取り、ソースのリストに追加します。このリストには、実行ループから入力ソースが除去されるときに、入力ソースを登録解除するためのルーチンも示してあります。

リスト 3-8 アプリケーションデリゲートへの入力ソースの登録と除去

```
- (void)registerSource:(RunLoopContext*)sourceInfo;
{
    [sourcesToPing addObject:sourceInfo];
}

- (void)removeSource:(RunLoopContext*)sourceInfo
{
    id objToRemove = nil;

    for (RunLoopContext* context in sourcesToPing)
```

```

    {
        if ([context isEqual:sourceInfo])
        {
            objToRemove = context;
            break;
        }
    }

    if (objToRemove)
        [sourcesToPing removeObject:objToRemove];
}

```

注： このリストのメソッドを呼び出すコールバック関数を [リスト 3-4](#)（48 ページ）および [リスト 3-6](#)（49 ページ）に示します。

入力ソースへのシグナル送信

クライアントは、データを入力ソースに渡した後でソースにシグナルを送り、実行ループをスリープ解除させる必要があります。ソースにシグナルを送ることにより、ソースが処理できる状態になっていることを実行ループで認識できます。また、シグナルが送信されたときにスレッドがスリープ状態になっている場合があるため、実行ループは明示的にスリープ解除させる必要があります。スリープ解除させないと、入力ソースの処理が遅延するおそれがあります。

リスト 3-9にRunLoopSourceオブジェクトのfireCommandsOnRunLoopメソッドを示します。クライアントは、バッファに追加したコマンドをソースで処理できる状態になったときに、このメソッドを呼び出します。

リスト 3-9 実行ループのスリープ解除

```

- (void)fireCommandsOnRunLoop:(CFRunLoopRef)runloop
{
    CFRunLoopSourceSignal(runLoopSource);
    CFRunLoopWakeUp(runloop);
}

```

注： カスタム入力ソースにメッセージを送信することによって、SIGHUPまたはその他のタイプのプロセスレベルのシグナルを処理しないでください。実行ループをスリープ解除させるためのCore Foundation関数はシグナルセーフではないため、アプリケーションのシグナルハンドラルーチン内では使用しないでください。シグナルハンドラルーチンの詳細については、sigactionのmanページを参照してください。

タイマーソースの設定

タイマーソースを作成するために必要な作業は、タイマーオブジェクトの作成と、実行ループでのタイマーオブジェクトのスケジューリングがすべてです。CocoaではNSTimerクラスを使用して新しいタイマーオブジェクトを作成します。Core Foundationでは、CFRunLoopTimerRef不透過型を使用します。NSTimerクラスを内部から見れば、同じメソッドを使用してタイマーを作成およびスケジューリングする機能など複数の便利な機能を提供する、単なるCore Foundationの拡張機能です。

Cocoaでは、次のいずれかのクラスメソッドを使用して、タイマーの作成およびスケジューリングを同時に実行できます。

- `scheduledTimerWithTimeInterval:target:selector:userInfo:repeats:`
- `scheduledTimerWithTimeInterval:invocation:repeats:`

これらのメソッドでは、タイマーを作成し、デフォルトモード（`NSDefaultRunLoopMode`）で現在のスレッドの実行ループに追加します。`NSTimer`オブジェクトを作成してから`NSRunLoop`の`addTimer:forMode:`メソッドを使用して実行ループに追加することにより、タイマーを手動でスケジューリングすることもできます。いずれの手法を使用しても結果は基本的に同じですが、タイマーの設定を制御できるレベルは異なります。たとえば、タイマーを作成して実行ループに手動で追加する場合は、デフォルトモード以外のモードを使用できます。リスト 3-10に、両方の手法を使用してタイマーを作成する方法を示します。最初のタイマーは、初期遅延は1秒ですが、その後は0.1秒ごとに定期的に作動します。2番目のタイマーは、0.2秒の初期遅延後に作動を開始し、その後は0.2ごとに作動します。

リスト 3-10 NSTimerを使用したタイマーの作成およびスケジューリング

```
NSRunLoop* myRunLoop = [NSRunLoop currentRunLoop];

// 最初のタイマーを作成およびスケジューリングします。
NSDate* futureDate = [NSDate dateWithTimeIntervalSinceNow:1.0];
NSTimer* myTimer = [[NSTimer alloc] initWithFireDate:futureDate
                                interval:0.1
                                target:self
                                selector:@selector(myDoFireTimer1:)
                                userInfo:nil
                                repeats:YES];
[myRunLoop addTimer:myTimer forMode:NSDefaultRunLoopMode];

// 2番目のタイマーを作成およびスケジューリングします。
[NSTimer scheduledTimerWithTimeInterval:0.2
                                target:self
                                selector:@selector(myDoFireTimer2:)
                                userInfo:nil
                                repeats:YES];
```

リスト 3-11に、**Core Foundation**の関数を使用してタイマーを設定するために必要なコードを示します。この例では、コンテキスト構造体でユーザ定義の情報を渡していませんが、タイマーで必要なすべてのカスタムデータを渡すためにこの構造体を使用できます。この構造体の内容の詳細については、『*CFRunLoopTimer Reference*』に記載されている説明を参照してください。

リスト 3-11 Core Foundationを使用したタイマーの作成およびスケジューリング

```
CFRunLoopRef runLoop = CFRunLoopGetCurrent();
CFRunLoopTimerContext context = {0, NULL, NULL, NULL, NULL};
CFRunLoopTimerRef timer = CFRunLoopTimerCreate(kCFAllocatorDefault, 0.1, 0.3,
0, 0,
                                &myCFTimerCallback, &context);

CFRunLoopAddTimer(runLoop, timer, kCFRunLoopCommonModes);
```

ポートベースの入力ソースの設定

CocoaおよびCore Foundationの両方に、スレッド間またはプロセス間で通信するためのポートベースのオブジェクトが用意されています。以降の各セクションでは、複数の異なるタイプのポートを使用してポート通信を設定する方法を示します。

NSMachPortオブジェクトの設定

NSMachPortオブジェクトとのローカル接続を確立するには、ポートオブジェクトを作成し、プライマリスレッドの実行ループに追加します。二次スレッドを起動するときに、同じオブジェクトをスレッドのエントリポイント関数に渡します。二次スレッドでは、同じオブジェクトを使用してプライマリスレッドにメッセージを返信できます。

メインスレッドのコードの実装

リスト 3-12に、二次ワーカースレッドを起動するためのプライマリスレッドのコードを示します。Cocoaフレームワークではポートおよび実行ループを設定するために多数の中間ステップを実行するため、launchThreadメソッドはCore Foundationの対応する関数（[リスト 3-17](#)（56 ページ））に比べてステップ数がかなり少なくなっています。ただし、この2つのフレームワークの動作はほぼ同じです。このメソッドでは、ローカルポートの名前をワーカースレッドに送信する代わりに、NSPortオブジェクトを直接送信する点などが異なります。

リスト 3-12 メインスレッドの起動メソッド

```
- (void)launchThread
{
    NSPort* myPort = [NSMachPort port];
    if (myPort)
    {
        // このクラスでは、受信するポートメッセージを処理します。
        [myPort setDelegate:self];

        // 入力ソースとしてポートを現在の実行ループに組み込みます。
        [[NSRunLoop currentRunLoop] addPort:myPort forMode:NSDefaultRunLoopMode];

        // スレッドをデタッチします。ワーカーにポートを解放させます。
        [NSThread detachNewThreadSelector:@selector(LaunchThreadWithPort:)
            toTarget:[MyWorkerClass class] withObject:myPort];
    }
}
```

スレッド間の双方向通信チャネルを設定するために、チェックインメッセージに入れた独自のローカルポートをワーカースレッドからメインスレッドに送信する場合があります。チェックインメッセージを受け取ることによって二次スレッドの起動がすべて正常であることをメインスレッドで認識できるだけでなく、メッセージをこのスレッドにさらに送信できるようにもなります。

リスト 3-13にプライマリスレッドのhandlePortMessage:メソッドを示します。このメソッドは、スレッドの自身のローカルポートにデータが到着したときに呼び出されます。チェックインメッセージが到着した場合、このメソッドはポートメッセージから二次スレッド用のポートを直接取得し、後で使用するために保存します。

リスト 3-13 Machポートのメッセージの処理

```
#define kCheckinMessage 100
```

```
// ワーカースレッドからの応答を処理します。
- (void)handlePortMessage:(NSPortMessage *)portMessage
{
    unsigned int message = [portMessage msgid];
    NSPort* distantPort = nil;

    if (message == kCheckinMessage)
    {
        // ワーカースレッドの通信ポートを取得します。
        distantPort = [portMessage sendPort];

        // 後で使用するためにワーカーポートを保持および保存します。
        [self storeDistantPort:distantPort];
    }
    else
    {
        // ほかのメッセージを処理します。
    }
}
```

二次スレッドのコードの実装

二次ワーカースレッドでは、スレッドを設定し、指定されたポートを使用して通信することにより、プライマリスレッドに情報を戻す必要があります。

リスト 3-14 に、ワーカースレッドを設定するためのコードを示します。メソッドでは、スレッドの自動解放プールを作成した後で、スレッドを実行させるワーカーオブジェクトを作成します。[リスト 3-15](#) (55 ページ) に示すワーカーオブジェクトの `sendCheckinMessage:` メソッドでは、ワーカースレッド用のローカルポートを作成して、メインスレッドをチェックインメッセージに返信します。

リスト 3-14 Machポートを使用したワーカースレッドの起動

```
+(void)LaunchThreadWithPort:(id)inData
{
    NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];

    // このスレッドとメインスレッドの間の接続を設定します。
    NSPort* distantPort = (NSPort*)inData;

    MyWorkerClass* workerObj = [[self alloc] init];
    [workerObj sendCheckinMessage:distantPort];
    [distantPort release];

    // 実行ループに処理を実行させます。
    do
    {
        [[NSRunLoop currentRunLoop] runMode:NSDefaultRunLoopMode
        beforeDate:[NSDate distantFuture]];
    }
    while (![workerObj shouldExit]);

    [workerObj release];
    [pool release];
}
```


NSMachPortを使用する場合は、ローカルおよびリモートのスレッドで、スレッド間の片方向の通信に同じポートオブジェクトを使用できます。つまり、一方のスレッドで作成したローカルポートオブジェクトは、もう一方のスレッドのリモートポートオブジェクトになります。

リスト 3-15に、二次スレッドのチェックインルーチンを示します。このメソッドは、以降の通信用に独自のローカルポートを設定してから、チェックインメッセージをメインスレッドに返信します。このメソッドでは、LaunchThreadWithPort:メソッドで受け取ったポートオブジェクトを、メッセージのターゲットとして使用します。

リスト 3-15 Machポートを使用したチェックインメッセージの送信

```
// ワークスレッドのチェックインメソッド
- (void)sendCheckinMessage:(NSPort*)outPort
{
    // 以降で使用するためにリモートポートを保持および保存します。
    [self setRemotePort:outPort];

    // ワークスレッドのポートを作成および設定します。
    NSPort* myPort = [NSMachPort port];
    [myPort setDelegate:self];
    [[NSRunLoop currentRunLoop] addPort:myPort forMode:NSDefaultRunLoopMode];

    // チェックインメッセージを作成します。
    NSPortMessage* messageObj = [[NSPortMessage alloc] initWithSendPort:outPort
                                                                    receivePort:myPort components:nil];

    if (messageObj)
    {
        // メッセージの設定を完了し、すぐに送信します。
        [messageObj setMsgid:kCheckinMessage];
        [messageObj sendBeforeDate:[NSDate date]];
    }
}
```

NSMessagePortオブジェクトの設定

スレッド間でポートオブジェクトを受け渡すだけでは、NSMessagePortオブジェクトとのローカル接続を確立できません。リモートメッセージポートは名前を取得する必要があります。Cocoaでこれを実現するには、固有の名前でローカルポートを登録し、この名前をリモートスレッドに渡すことによって、リモートスレッドで通信用の適切なポートオブジェクトを取得できるようにする必要があります。リスト 3-16に、メッセージポートを使用する場合にポートを作成および登録するための処理を示します。

リスト 3-16 メッセージポートの登録

```
NSPort* localPort = [[[NSMessagePort alloc] init] retain];

// オブジェクトを設定し、現在の実行ループに追加します。
[localPort setDelegate:self];
[[NSRunLoop currentRunLoop] addPort:localPort forMode:NSDefaultRunLoopMode];

// 固有の名前を使用してポートを登録します。名前は固有である必要があります。
NSString* localPortName = [NSString stringWithFormat:@"MyPortName"];
[[NSMessagePortNameServer sharedInstance] registerPort:localPort
                                                    name:localPortName];
```

Core Foundationでのポートベース入力ソースの設定

このセクションでは、**Core Foundation**を使用して、アプリケーションのメインスレッドとワーカースレッドの間に双方向通信チャンネルを設定する方法を説明します。

リスト 3-17に、ワーカースレッドを起動するために、アプリケーションのメインスレッドから呼び出すコードを示します。このコードでは、まずワーカースレッドからのメッセージを監視する `CFMessagePortRef` 不透過型を設定します。ワーカースレッドで接続を確立するためにポートの名前が必要になるため、ワーカースレッドのエントリポイント関数に対して文字列値を配信します。通常は、現在のユーザコンテキストで固有のポート名にする必要があります。固有でない場合は、競合が発生します。

リスト 3-17 Core Foundationメッセージポートの新しいスレッドへの接続

```
#define kThreadStackSize      (8 * 4096)

OSStatus MySpawnThread()
{
    // 応答を受け取るローカルポートを作成します。
    CFStringRef myPortName;
    CFMessagePortRef myPort;
    CFRunLoopSourceRef r1Source;
    CFMessagePortContext context = {0, NULL, NULL, NULL, NULL};
    Boolean shouldFreeInfo;

    // 文字列を使用してポート名を作成します。
    myPortName = CFStringCreateWithFormat(NULL, NULL,
    CFSTR("com.myapp.MainThread"));

    // ポートを作成します。
    myPort = CFMessagePortCreateLocal(NULL,
        myPortName,
        &MainThreadResponseHandler,
        &context,
        &shouldFreeInfo);

    if (myPort != NULL)
    {
        // ポートが正常に作成されました。
        // 次に、ポート用の実行ループソースを作成します。
        r1Source = CFMessagePortCreateRunLoopSource(NULL, myPort, 0);

        if (r1Source)
        {
            // 現在の実行ループにソースを追加します。
            CFRunLoopAddSource(CFRunLoopGetCurrent(), r1Source,
            kCFRunLoopDefaultMode);

            // これらは、組み込みが終われば、解放できます。
            CFRelease(myPort);
            CFRelease(r1Source);
        }
    }

    // スレッドを作成し、処理を続行します。
    MPTaskID taskID;
    return(MPCreateTask(&ServerThreadEntryPoint,
```

```

        (void*)myPortName,
        kThreadStackSize,
        NULL,
        NULL,
        NULL,
        0,
        &taskID));
}

```

ポートが組み込まれ、スレッドが起動されたため、メインスレッドでは、スレッドのチェックインを待機しながら通常の実行を続行できます。到着したチェックインメッセージは、リスト 3-18に示すメインスレッドのMainThreadResponseHandler関数にディスパッチされます。この関数では、ワーカースレッド用のポート名を抽出し、以降の通信で使用するコンジットを作成します。

リスト 3-18 チェックインメッセージの受信

```

#define kCheckinMessage 100

// メインスレッドのポートメッセージハンドラ
CFDataRef MainThreadResponseHandler(CFMessagePortRef local,
                                     SInt32 msgid,
                                     CFDataRef data,
                                     void* info)
{
    if (msgid == kCheckinMessage)
    {
        CFMessagePortRef messagePort;
        CFStringRef threadPortName;
        CFIndex bufferSize = CFDataGetLength(data);
        UInt8* buffer = CFAllocatorAllocate(NULL, bufferSize, 0);

        CFDataGetBytes(data, CFRangeMake(0, bufferSize), buffer);
        threadPortName = CFStringCreateWithBytes (NULL, buffer, bufferSize,
        kCFStringEncodingASCII, FALSE);

        // 名前でもリモートメッセージポートを取得する必要があります。
        messagePort = CFMessagePortCreateRemote(NULL,
        (CFStringRef)threadPortName);

        if (messagePort)
        {
            // 以降で参照するためにスレッドの通信ポートを保持および保存します。
            AddPortToListOfActiveThreads(messagePort);

            // ポートは前の関数によって保持されているため、ここで
            // 解放します。
            CFRelease(messagePort);
        }

        // クリーンアップ。
        CFRelease(threadPortName);
        CFAllocatorDeallocate(NULL, buffer);
    }
    else
    {
        // ほかのメッセージを処理します。
    }
}

```

```

    return NULL;
}

```

これでメインスレッドの設定が終わったため、後は、新しく作成したワーカースレッドのために独自のポートを作成し、チェックインするだけです。リスト 3-19に、ワーカースレッドのエントリポイント関数を示します。この関数はメインスレッドのポート名を抽出し、これを使用してメインスレッドへのリモート接続を作成します。次に、関数自体のためのローカルポートを作成し、スレッドの実行ループにこのポートを組み込んで、ローカルポート名を含むチェックインメッセージをメインスレッドに送信します。

リスト 3-19 スレッド構造の設定

```

OSStatus ServerThreadEntryPoint(void* param)
{
    // メインスレッドへのリモートポートを作成します。
    CFMessagePortRef mainThreadPort;
    CFStringRef portName = (CFStringRef)param;

    mainThreadPort = CFMessagePortCreateRemote(NULL, portName);

    // paramで渡された文字列を解放します。
    CFRelease(portName);

    // ワーカースレッド用のポートを作成します。
    CFStringRef myPortName = CFStringCreateWithFormat(NULL, NULL,
        CFSTR("com.MyApp.Thread-%d"), MPCurrentTaskID());

    // 後で参照するために、このスレッドのコンテキスト情報にポートを格納します。
    CFMessagePortContext context = {0, mainThreadPort, NULL, NULL, NULL};
    Boolean shouldFreeInfo;
    Boolean shouldAbort = TRUE;

    CFMessagePortRef myPort = CFMessagePortCreateLocal(NULL,
        myPortName,
        &ProcessClientRequest,
        &context,
        &shouldFreeInfo);

    if (shouldFreeInfo)
    {
        // ローカルポートを作成できなかったため、スレッドを強制終了します。
        MPExit(0);
    }

    CFSRunLoopSourceRef r1Source = CFMessagePortCreateRunLoopSource(NULL, myPort,
0);
    if (!r1Source)
    {
        // ローカルポートを作成できなかったため、スレッドを強制終了します。
        MPExit(0);
    }

    // 現在の実行ループにソースを追加します。
    CFSRunLoopAddSource(CFSRunLoopGetCurrent(), r1Source, kCFSRunLoopDefaultMode);

    // これらは、組み込みが終われば、解放できます。
    CFRelease(myPort);
    CFRelease(r1Source);
}

```

```

// ポート名を組み込んでチェックインメッセージを送信します。
CFDataRef returnData = nil;
CFDataRef outData;
CFIndex stringLength = CFStringGetLength(myPortName);
UInt8* buffer = CFAllocatorAllocate(NULL, stringLength, 0);

CFStringGetBytes(myPortName,
                 CFRangeMake(0,stringLength),
                 kCFStringEncodingASCII,
                 0,
                 FALSE,
                 buffer,
                 stringLength,
                 NULL);

outData = CFDataCreate(NULL, buffer, stringLength);

CFMessagePortSendRequest(mainThreadPort, kCheckinMessage, outData, 0.1, 0.0,
NULL, NULL);

// スレッドデータ構造体をクリーンアップします。
CFRelease(outData);
CFAllocatorDeallocate(NULL, buffer);

// 実行ループに入ります。
CFRunLoopRun();
}

```

実行ループの開始後は、スレッドのポートに送信される以降のすべてのイベントが `ProcessClientRequest` 関数によって処理されます。この関数の実装は、スレッドで実行する作業のタイプによって異なり、ここには示してありません。

同期

1つのアプリケーションに複数のスレッドが存在していると、実行される複数のスレッドからリソースに対して安全にアクセスするという潜在的な問題が発生します。同じリソースに対する2つのスレッドによる変更が、意図しない方法で互いに干渉する場合があります。たとえば、一方のスレッドによってもう一方の変更を上書きしたり、アプリケーションを潜在的に無効で不明な状態にしたりする場合があります。運がよければ、リソースの破損はパフォーマンス上の問題としてはっきり示されたり、リソースの破損を原因とするクラッシュを比較的容易に追跡および修正できたりします。一方、運が悪ければ、破損を原因とするエラーが微細でずっと後にならないと検出されなかったり、基礎にあるコーディングの前提条件を徹底的に見直さなければならないエラーが生じたりする場合があります。

スレッドの安全性に関して言えば、優れた設計が最大の防御になります。共有リソースを使用せず、スレッド間のやり取りを最小化すれば、それらのスレッドが互いに干渉する可能性がずっと低下します。ただし、完全に干渉のない設計が常に可能なわけではありません。スレッド間のやり取りが必要な場合は、同期ツールを使用して、やり取りの安全性を確保する必要があります。

MacOSXおよびiOSでは、相互のアクセスを排他的にするためのツールから、アプリケーション内でイベントを正しい順序にするツールまで、非常にさまざまな同期ツールを使用できます。以降の各セクションでは、これらのツールについて説明し、これらのツールをコードで使用してプログラムのリソースに対する安全なアクセスを制御する方法を説明します。

同期ツール

異なるスレッドによる予期しない形でのデータの変更を防ぐには、同期の問題が生じないようにアプリケーションを設計するか、同期ツールを使用します。同期の問題を完全に回避できれば理想的ですが、常に可能なわけではありません。以降の各セクションでは、使用可能な同期ツールの基本カテゴリを説明します。

アトミック操作

アトミック操作は、単純なデータ型を操作する単純な形態の同期です。アトミック操作には、競合するスレッドをブロックしないという長所があります。カウンタ変数のインクリメントなど単純な操作では、ロックを使用するよりもずっとよいパフォーマンスを得られることがあります。

MacOSXおよびiOSには、32ビットおよび64ビットの値に対して数学的演算および論理演算を実行するための大量の操作が組み込まれています。これに含まれる操作として、コンペアアンドスワップ、テストアンドセット、およびテストアンドクリア操作のアトミックバージョンがあります。サポートされているアトミック操作のリストについては、`/usr/include/libkern/OSAtomic.h`ヘッダファイルを参照するか、`atomic`のmanページを参照してください。

メモリバリアおよびvolatile変数

コンパイラでは、パフォーマンスを最適化するために、プロセッサの命令パイプラインの空きをできるだけ減らすようにアセンブルレベルの命令を並べ替えることがよくあります。この最適化で、並べ替えによってデータの正確性を失うことはないコンパイラが判定し、メインメモリにアクセスする命令を並べ替える場合があります。残念ながら、メモリ依存の一部の操作は、コンパイラで検出できない場合があります。無関係に見える変数が実際には互いに影響していると、コンパイラ最適化によってこれらの変数が誤った順序で更新され、正しくない結果になる場合があります。

メモリバリアは、確実に正しい順序でメモリ操作を実行させるための、ブロックのないタイプの同期ツールです。メモリバリアはフェンスのように機能して、バリアより前に配置されているすべてのロード操作およびストア操作をプロセッサで必ず完了しないと、バリアより後に配置されているロード操作およびストア操作を実行できないようにします。通常は、ほかのスレッドから可視の特定のスレッドによるメモリ操作を、常に期待する順序で実行させるためにメモリバリアを使用します。このような場合にメモリバリアがないと、一見発生しそうなない結果がほかのスレッドに示されるおそれがあります（例については、[Wikipediaのmemory barriers](#)のページを参照してください）。メモリバリアは、コードの適切なポイントでOSMemoryBarrier関数を呼び出すだけで使用できます。

volatile変数では、別のタイプのメモリ制約を個々の変数に適用します。コンパイラは、変数の値をレジスタにロードすることによってコードを最適化場合があります。ローカル変数であれば、これが問題となることはまずありません。一方、別のスレッドから可視の変数の場合は、この最適化の結果、ほかのスレッドで変数に対する変更を認識できなくなります。変数にvolatileキーワードを指定すれば、この変数を使用するたびに必ずメモリからロードされるようになります。コンパイラで検出できない可能性のある外部ソースによって変数の値が随時変更されることがある場合などに、変数をvolatile宣言します。

メモリバリアおよびvolatile変数の両者ともコンパイラで実行できる最適化を減らすことになるため、頻繁には使用せず、正確性を確保するために必要な場所でのみ使用してください。メモリバリアの使用法については、OSMemoryBarrierのmanページを参照してください。

ロック

ロックは最もよく使われる同期ツールの1つです。ロックを使用するとコードの**クリティカルセクション**を保護できます。クリティカルセクションとは、一度に1つのスレッドだけにアクセスを許可するコードのセグメントです。たとえば、特定のデータ構造体を操作する部分や、一度にサポートするクライアントが1つまでの何らかのリソースを使用する部分がクリティカルセクションになります。このセクションの前後にロックを配置すると、コードの正確性に影響する可能性のある変更がほかのスレッドによって行われなくなります。

表 4-1に、プログラミングでよく使用するロックの一部を示します。Mac OS XおよびiOSでは、このロックのタイプの大部分を実装できますが、一部は実装できません。サポートされていないロックのタイプについては、プラットフォームでそのロックを直接実装していない理由を説明カラムに示してあります。

表 4-1 ロックのタイプ

ロック	説明
ミューテックス	リソースを囲む保護バリアとして機能する相互に排他的なロック（ミューテックス）ミューテックスは、一度に1つのスレッドだけにアクセスを許可する一種のセマフォです。ミューテックスの使用中にそのミューテックスを取得しようとする別のスレッドは、元のスレッドによってミューテックスが解放されるまでブロックされます。複数のスレッドが同じミューテックスを取得しようとした場合、一度に1つだけがミューテックスへのアクセスを許可されます。
再帰ロック	再帰ロックは、ミューテックスロックの変形の1つです。再帰ロックでは、ロックを取得した単一のスレッドで、そのロックを解放する前に複数回取得できます。ロックのオーナーが取得と同じ回数だけロックを解放するまで、ほかのスレッドはブロックされたままになります。再帰ロックは主に再帰的な反復で使用しますが、複数メソッドで別々にそのロックを取得する必要がある場合にも使用できます。
読み取り／書き込みロック	読み取り／書き込みロックは、共有排他ロックとも呼ばれます。このタイプのロックは大規模操作で使うことが多く、保護するデータ構造体の読み取り頻度が高く、ときどきしか変更しない場合に、パフォーマンスを大幅に向上できます。通常の操作の間は、複数の読み取り操作で同時にそのデータ構造体にアクセスできます。一方、このデータ構造体へ書き込もうとするスレッドは、すべての読み取り操作がロックを解放するまでブロックされ、解放後にロックを取得して構造体を更新できます。書き込みスレッドでロックを待機している間、新しい読み取りスレッドは、書き込みスレッドが完了するまでブロックされます。システムでは、POSIXスレッドを使用した読み取り／書き込みロックだけをサポートしています。これらのロックを使用する方法の詳細については、pthreadのmanページを参照してください。
分散ロック	分散ロックでは、プロセスレベルの相互に排他的なアクセスを実装できます。本物のミューテックスと異なり、分散ロックでは、プロセスをブロックせず、プロセスの実行は妨げられません。ロックがビジーであることを通知するのみで、処理方法はプロセスまかせです。
スピンロック	スピンロックでは、条件が真になるまでロックの条件を繰り返しポーリングします。スピンロックは、予期されるロックの待機時間が短いマルチプロセッサシステムで最もよく使われます。そのような状況では、スレッドをブロックするよりもポーリングするほうが通常は効率的です。スレッドをブロックする場合は、コンテキストを切り替え、スレッドのデータ構造体を更新する必要があるためです。スピンロックはポーリングを本質とするため、システムには、スピンロックを実装するための用意はありませんが、状況に応じて簡単に実装できます。カーネルでのスピンロックの実装については、『 <i>Kernel Programming Guide</i> 』を参照してください。
ダブルチェックロック	ダブルチェックロックは、ロックを取得する前にロック条件を検査することによってロックの取得に関わるオーバーヘッドを削減しようとする試みです。ダブルチェックロックは安全でない可能性を秘めているため、システムでは明示的にサポートしておらず、使用しないことをお勧めします。

注： 大部分のタイプのロックには、メモリバリアも組み込まれており、必ず先行するロード命令およびストア命令が完了してからクリティカルセクションに入ります。

ロックを使用する方法の詳細については、「[ロックの使用](#)」（71 ページ）を参照してください。

条件変数

条件変数は、特定の条件に該当する場合にスレッドで互いにシグナルを送ることができる、もう1つのタイプのセマフォです。条件変数は、通常、リソースの可用性を示すためや、必ず特定の順序でタスクを実行させるために使われます。スレッドの条件判定の際には、すでにその条件に該当している場合を除いてブロックされます。このスレッドは、ほかの何らかのスレッドによって条件が明示的に変更されて、シグナルが送信されるまでブロックされたままになります。ミューテックスロックと異なり、複数のスレッドが同時にその条件へのアクセスを許可されます。条件変数は、いわば門番のようなもので、指定された何らかの基準に応じてさまざまなスレッドに門を通過させます。

条件変数の使用方法の1つとして、未処理イベントのプールを管理する場合があります。イベントキューにイベントがあれば、条件変数を使用して、待機中のスレッドにシグナルが送信されます。1つのイベントがキューに届いている場合は、それに応じてシグナルを送ります。すでに待機中のスレッドがある場合は、スリープ解除され、すぐにキューからイベントを取り出して処理します。2つのイベントがほぼ同時にキューに到着した場合は、キューは条件のシグナルを2度送信して、2つのスレッドをスリープ解除させます。

システムの複数のテクノロジーで条件変数がサポートされています。一方、条件変数を適切に実装するには緻密なコーディングが必要であるため、独自のコードで条件変数を使用する前に「[条件変数の使用](#)」（76 ページ）に示されている例を参照してください。

セレクトアルーチンの実行

Cocoaアプリケーションには、同期された方法で単一のスレッドにメッセージを配信するための便利な方法があります。NSObjectクラスには、アプリケーションのいずれかのアクティブスレッドでセレクトを実行するためのメソッドが宣言されています。これらのメソッドにより、ターゲットスレッドでは必ず同期して実行されることを前提として、スレッドで非同期にメッセージを配信できます。たとえば、分散計算の結果をアプリケーションのメインスレッドまたは指定されたコーディネータスレッドに配信するために実行セレクトメッセージを使用できます。セレクトを実行する各要求はターゲットスレッドの実行ループでキューに格納されてから受け取った順序で順次処理されます。

実行セレクトアルーチンの概要および使用方法の詳細については、「[Cocoa実行セレクトソース](#)」（38 ページ）を参照してください。

同期のコストおよびパフォーマンス

同期によってコードの正確性は確保されますが、パフォーマンスが犠牲になります。同期ツールを使用すると、競合のない場合であっても遅延が生じます。ロック操作およびアトミック操作では、メモリバリアを使用することと、カーネルレベルの同期によりコードを適切に保護することが一般的です。また、ロックの競合がある場合は、スレッドがブロックされて遅延がさらに大きくなることがあります。

表 4-2 に、競合のない場合のミューテックスおよびアトミック操作に関する概算コストの一部を示します。この測定値は、数千回のサンプルから得られた平均時間を表します。スレッド作成時間同様、ミューテックス取得時間は競合のない場合でも、プロセッサの負荷、コンピュータの速度、および使用可能なシステムメモリとプログラムメモリの量によって大きく変わります。

表 4-2 ミューテックスおよびアトミック操作のコスト

項目	概算コスト	メモ
ミューテックス取得時間	約0.2マイクロ秒	これは、競合のない場合のロック取得時間です。別のスレッドによってロックが保持されている場合、取得時間はずっと長くなることがあります。これらの数値は、2 GHzのCore Duoプロセッサと1 GBのRAMを搭載し、Mac OS X v10.5を稼働しているIntelベースのiMacを使用してミューテックスを取得するときに得られた平均値および中央値を分析して算出されています。
アトミックコンペアアンドスワップ	約0.05マイクロ秒	これは、競合のない場合のコンペアアンドスワップ時間です。これらの数値は、これらの操作に対する平均値および中央値を分析した値であり、2 GHzのCore Duoプロセッサと1 GBのRAMを搭載し、Mac OS X v10.5を稼働しているIntelベースのiMacで得られた値です。

並行タスクを設計する場合は、常に正確性が最も重要な要因ですが、パフォーマンス要因も考慮する必要があります。複数スレッドで正しく実行されても、単一のスレッドで同じコードを実行する場合より遅くなっては、改良と呼ばえません。

既存の単一スレッドアプリケーションを作り変える場合は、必ず主要タスクのパフォーマンスに対する一連の基礎量を測定しておく必要があります。その後、追加のスレッドを追加したときに、その同じタスクの測定値を取得して、マルチスレッドの場合と単一スレッドを場合のパフォーマンスを比較する必要があります。コードを調整してもスレッド処理によってパフォーマンスが向上しない場合は、個々の実装やスレッドの使用自体を再検討する必要がある可能性があります。

パフォーマンスおよびメトリックを収集するツールについては、『*Performance Overview*』を参照してください。ロックおよびアトミック操作に関する固有の情報については、「[スレッドのコスト](#)」（23 ページ）を参照してください。

スレッドの安全性とシグナル

スレッド化されたアプリケーションの場合に最も懸念されたり、混乱させられたりする問題は、シグナルの処理です。シグナルは、プロセスに情報を配信したり、プロセスに何らかの操作を加えたりするために使用できる低レベルのBSDメカニズムです。一部のプログラムでは、シグナルを使用して、子プロセスの終了などの特定のイベントを検出します。システムでは、暴走したプロセスを終了するためや、その他の種類の情報を通信するためにシグナルを使用します。

シグナルに関する問題は、その機能ではなく、アプリケーションにスレッドが複数ある場合の動作に関するものです。単一スレッドアプリケーションでは、すべてのシグナルハンドラがメインスレッドで実行されます。マルチスレッドアプリケーションでは、不正な命令など特定のハードウェアエラーと結びついていないシグナルは、その時点でたまたま実行されていた任意のスレッドに配信されます。複数のスレッドが同時に実行されている場合、シグナルはシステムが随意で選択したスレッドに配信されます。つまり、シグナルは、アプリケーションのいずれのスレッドにも配信される可能性があります。

アプリケーションにシグナルハンドラを実装するときは、どのスレッドでシグナルを処理するかについて考える必要はありません。特定のスレッドに特定のシグナルを処理させるには、そのシグナルが届いたときにそのスレッドに通知する何らかの方法を考案する必要があります。スレッドにシグナルハンドラを組み込むことで、その同じスレッドにシグナルが配信されるとそのまま仮定することはできません。

シグナルおよびシグナルハンドラの組み込みの詳細については、`signal`および`sigaction`の`man`ページを参照してください。

スレッドセーフな設計のヒント

同期ツールはコードをスレッドセーフにするために有用ですが、万能薬ではありません。ロックおよびその他のタイプの同期プリミティブを使用しすぎると、アプリケーションをスレッド化した場合のパフォーマンスのほうが悪化しない場合のパフォーマンスよりかえって低下するおそれがあります。安全性とパフォーマンスの適切なバランスを見出すには、経験を伴う技術力が必要とされます。以降の各セクションでは、アプリケーションの適切なレベルの同期を選択するために有用なヒントを示します。

同期を完全に避ける

これから取り掛かる新規プロジェクトだけでなく、既存のプロジェクトであっても、同期を不要とするようにコードおよびデータ構造体を設計できるのであれば、それ以上の解決策はありません。ロックおよびその他の同期ツールは有用ですが、すべてのアプリケーションのパフォーマンスに影響を与えます。全体的な設計によって特定のリソースをめぐる競合が激しくなる場合、スレッドの待機時間はずっと長くなるおそれがあります。

並行タスク間のやり取りおよび相互依存を少なくすることが、並行処理を実装する最適な方法です。各タスクでそのタスク自体のプライベートデータセットを操作する場合は、ロックを使用してデータを保護する必要はありません。2つのタスクで共通のデータセットを共有する場合でも、このデータセットを分割するか、各タスクに独自のコピーを提供する方法を見つけることができます。データセットをコピーするにも、もちろんそれなりにコストがかかるため、コピーのコストと同期のコストを比較検討したうえで決める必要があります。

同期の制限事項を理解する

同期ツールは、アプリケーションのすべてのスレッドで一貫して使用した場合に限り効果を発揮します。特定のリソースへのアクセスを制限するミューテックスを作成する場合は、すべてのスレッドでそのリソースを操作する前に同じミューテックスを取得する必要があります。抜けがあれば、プログラマエラーであり、ミューテックスによる保護は失敗します。

コードの正確性を脅かす要因に注意する

ロックおよびメモリバリアを使用する場合は、コードでそれらを配置する場所に、常に注意する必要があります。適切に配置したように見えるロックであっても、実際には安心してはいけない場合があります。次の一連の例では、完全に安全に見えるコードに含まれている欠陥を指摘することにより、この問題を明らかにしようとしています。不変オブジェクトを1組含む可変配列があることを大前提としています。配列の先頭オブジェクトにあるメソッドを呼び出す場合を考えます。ここでは、次のコードを使用します。

```
NSLock* arrayLock = GetArrayLock();
NSMutableArray* myArray = GetSharedArray();
id anObject;
```

```
[arrayLock lock];
anObject = [myArray objectAtIndex:0];
[arrayLock unlock];
```

```
[anObject doSomething];
```

配列は可変であるため、目的のオブジェクトを取得するまで、配列の前後のロックによってほかのスレッドは配列を変更できなくなります。また、取得するオブジェクト自体は不変であるため、doSomethingメソッドの呼び出しの前後にロックは不要です。

ただし、この例には問題があります。ロックを解放してからdoSomethingメソッドを実行する機会を得るまでに、別のスレッドが割り込んですべてのオブジェクトを配列から除去するとどうなるでしょう。ガベージコレクションを使用しないアプリケーションでは、コードで保持しているオブジェクトが解放されて、anObjectは無効なメモリアドレスを指すようになる可能性があります。この問題は、次に示すように、doSomethingを呼び出した後でロックを解放するよう既存のコードを単に並べ替えるだけで解決できます。

```
NSLock* arrayLock = GetArrayLock();
NSMutableArray* myArray = GetSharedArray();
id anObject;
```

```
[arrayLock lock];
anObject = [myArray objectAtIndex:0];
[anObject doSomething];
[arrayLock unlock];
```

doSomethingの呼び出しをロックの内側に移動することによって、メソッドを呼び出すときにオブジェクトがまだ有効であることを保証するコードになります。このコードではロックを長時間保持するため、あいにくdoSomethingメソッドの実行に長時間かかる場合は、パフォーマンスのボトルネックになることがあります。

このコードの問題は、クリティカル領域を適切に定義していないことではなく、実際の問題が理解されていないことです。実際の問題は、ほかのスレッドが存在することによってのみ引き起こされるメモリ管理の問題です。別のスレッドによってanObjectが解放される可能性があるため、ロックを解放する前に保持するとよいでしょう。この方法では、潜在的なパフォーマンス上の悪影響をもたらすことなく、オブジェクトが解放されてしまうという真の問題に対処しています。

```

NSLock* arrayLock = GetArrayLock();
NSMutableArray* myArray = GetSharedArray();
id anObject;

[arrayLock lock];
anObject = [myArray objectAtIndex:0];
[anObject retain];
[arrayLock unlock];

[anObject doSomething];

```

前述の例は、このうえなく単純ですが、非常に重要なポイントを明らかにしています。つまり、正確性に関しては、自明でない問題についても検討する必要があります。メモリ管理および設計のその他の側面は、複数のスレッドが存在することによっても影響を受ける場合があるため、それらの問題について前もって検討する必要があります。さらに、安全性に関しては、コンパイラによる操作が最悪の結果になる可能性を常に想定する必要があります。このような認識を持って警戒すれば、潜在的な問題の回避につながり、コードを確実に正しく動作させるために有用です。

プログラムをスレッドセーフにする方法の別の例については「[Technical Note TN2059: "Using Collection Classes Safely in Multithreaded Applications"](#)」を参照してください。

デッドロックおよびライブロックへの配慮

単一のスレッドで複数のロックを同時に取得しようとする場合は、常にデッドロックが発生する可能性があります。デッドロックは、それぞれ互いのスレッドで必要なロックを保持している2つのスレッドが、もう一方のスレッドによって保持されているロックを取得しようとしたときに発生します。各スレッドはもう1つのロックを取得できないため、永久にブロックされる結果になります。

ライブロックはデッドロックと似ており、2つのスレッドが同じリソースのセットを取得しようとして競合する場合に発生します。ライブロック状況のスレッドは、第2のロックを取得しようとする過程で第1のロックを解放します。第2のロックを取得できれば、立ち返って、第1のロックを再度取得しようとしします。実際の作業にではなく、1つのロックの解放と別のロックの取得にすべての時間を費やしたため、スレッドは停止します。

一度に1つのロックだけを取得することが、デッドロック状況およびライブロック状況の両方を避けるための最良の方法です。一度に複数のロックを取得する必要がある場合は、ほかのスレッドで同様の処理をしていないことを確認する必要があります。

volatile変数の適切な使用

すでにミューテックスを使用してコードのセクションを保護していても、そのセクションに含まれている重要な変数をvolatileキーワードで保護しなければならないと、条件反射的に考えないでください。ミューテックスには、ロード操作およびストア操作を必ず適切に順序付けするためにメモリバリアが含まれています。クリティカルセクションに含まれている変数にvolatileキーワードを

追加すると、この変数にアクセスするたびに値が必ずメモリからロードされます。場合によってはこの2つの同期手法を組み合わせなければなりません、パフォーマンスは大幅に低下します。ミューテックスだけで変数を十分保護できる場合は、`volatile`キーワードを省いてください。

これも重要な点ですが、ミューテックスの使用を避ける目的で`volatile`変数を使用しないでください。通常は、ミューテックスおよびその他の同期メカニズムによってデータ構造体の完全性を保護するほうが、`volatile`変数を使用するより適切です。`volatile`キーワードによって保証されるのは変数をメモリからロードすることだけであり、レジスタへの格納ではありません。このキーワードを使用したからといって、変数に正しくアクセスするコードになるわけではありません。

アトミック操作の使用

ロックによる影響を受けることなく一部のタイプの操作を実行するための方法の1つとして、ブロックのない同期があります。ロックは2つのスレッドを同期させるための効果的な方法ですが、競合のない場合であってもロックの取得操作には比較的成本がかかります。これに対し、多くのアトミック操作は短時間で完了されてロックと同程度に効果的な場合があります。

アトミック操作では、32ビットおよび64ビットの値に対して単純な数学的演算および論理演算を実行できます。これらの操作では、特別なハードウェア命令およびオプションのメモリバリアに頼って、特定の操作を必ず完了してから、その操作の影響を受けるメモリへのアクセスが再開されるようにします。マルチスレッドの場合は、常にメモリバリアを組み込んだアトミック操作を使用し、メモリをスレッド間で必ず適切に同期させる必要があります。

表 4-3に、使用可能な数学的演算と論理演算および対応する関数名を示します。このすべての関数は、`/usr/include/libkern/OSAtomic.h`ヘッダファイルに宣言されています。このヘッダファイルには、完全な構文も示されています。これらの関数の64ビットバージョンは64ビットプロセスでのみ使用可能です。

表 4-3 アトミックな数学的演算および論理演算

演算	関数名	説明
追加	<code>OSAtomicAdd32</code> <code>OSAtomicAdd32Barrier</code> <code>OSAtomicAdd64</code> <code>OSAtomicAdd64Barrier</code>	2つの整数値を加算し、指定された変数のいずれかに結果を格納します。
インクリメント	<code>OSAtomicIncrement32</code> <code>OSAtomicIncrement32Barrier</code> <code>OSAtomicIncrement64</code> <code>OSAtomicIncrement64Barrier</code>	指定された整数値を1インクリメントします。

演算	関数名	説明
デクリメント	OSAtomicDecrement32 OSAtomicDecrement32Barrier OSAtomicDecrement64 OSAtomicDecrement64Barrier	指定された整数値を1デクリメントします。
論理OR	OSAtomicOr32 OSAtomicOr32Barrier	指定された32ビット値と32ビットマスクの論理ORを実行します。
論理AND	OSAtomicAnd32 OSAtomicAnd32Barrier	指定された32ビット値と32ビットマスクの論理ANDを実行します。
論理XOR	OSAtomicXor32 OSAtomicXor32Barrier	指定された32ビット値と32ビットマスクの論理XORを実行します。
コンペアアンドスワップ	OSAtomicCompareAndSwap32 OSAtomicCompareAndSwap32Barrier OSAtomicCompareAndSwap64 OSAtomicCompareAndSwap64Barrier OSAtomicCompareAndSwapPtr OSAtomicCompareAndSwapPtrBarrier OSAtomicCompareAndSwapInt OSAtomicCompareAndSwapIntBarrier OSAtomicCompareAndSwapLong OSAtomicCompareAndSwapLongBarrier	指定された古い値と変数を比較します。2つの値が等しい場合、この関数は指定された新しい値を変数に代入します。等しくない場合は何もしません。この関数は、比較と代入を1つのアトミック操作で実行して、実際に入れ替えを行ったかどうかを示すブール値を返します。
テストアンドセット	OSAtomicTestAndSet OSAtomicTestAndSetBarrier	指定された変数内のビットをテストし、このビットに1を設定して、元のビットの値をブール値として返します。ビットは、 $((\text{char}^*)\text{address} + (\text{n} \gg 3))$ バイトの式 $(0 \times 80 \gg (\text{n} \& 7))$ に従ってテストされます。ここで、nはビット番号、addressは変数のポインタです。この式は、変数を8ビットサイズのチャンクに分割し、各チャンクのビットの順序を反転させます。たとえば、32ビット整数の最下位ビット（ビット0）をテストする場合、実際に指定するビット番号は7になります。同様に、最上位ビット（ビット32）をテストする場合は、ビット番号に24を指定します。

演算	関数名	説明
テストアンドクリア	OSAtomicTestAndClear OSAtomicTestAndClearBarrier	指定された変数内のビットをテストし、このビットに0を設定して、元のビットの値をブール値として返します。ビットは、 <code>((char*)address + (n >> 3))</code> バイトの式 <code>(0x80 >> (n & 7))</code> に従ってテストされます。ここで、 <code>n</code> はビット番号、 <code>address</code> は変数のポインタです。この式は、変数を8ビットサイズのチャンクに分割し、各チャンクのビットの順序を反転させます。たとえば、32ビット整数の最下位ビット（ビット0）をテストする場合、実際に指定するビット番号は7になります。同様に、最上位ビット（ビット32）をテストする場合は、ビット番号に24を指定します。

大部分のアトミック関数は比較的わかりやすい、予期したとおりの動作をします。一方、リスト4-1は、アトミックなテストアンドセット操作およびコンペアアンドスワップ操作の動作を示しており、これは多少複雑です。OSAtomicTestAndSet関数の初めの3つの呼び出しは、整数値に対するビット演算式の使用法および予想される値とは異なっている可能性のある演算結果を示します。最後の2つの呼び出しは、OSAtomicCompareAndSwap32関数の動作を示します。いずれの場合も、ほかのスレッドでこの値を操作していない競合のない状態でこれらの関数を呼び出しています。

リスト 4-1 アトミック操作の実行

```
int32_t  theValue = 0;
OSAtomicTestAndSet(0, &theValue);
// theValueは128です。

theValue = 0;
OSAtomicTestAndSet(7, &theValue);
// theValueは1です。

theValue = 0;
OSAtomicTestAndSet(15, &theValue);
// theValueは256です。

OSAtomicCompareAndSwap32(256, 512, &theValue);
// theValueは512です。

OSAtomicCompareAndSwap32(256, 1024, &theValue);
// theValueはまだ512です。
```

アトミック操作については、atomicのmanページおよび`/usr/include/libkern/OSAtomic.h`ヘッダファイルを参照してください。

ロックの使用

ロックは、スレッドプログラミングのための基礎的な同期ツールです。ロックを使用すると、コードの大規模セクションを簡単に保護して、コードの正確性を確保できます。Mac OS XおよびiOSには、すべてのアプリケーションタイプで利用できる基本ミューテックスロックが用意されており、

Foundationフレームワークには、特別な状況で使用するミューテックスロックの補足的なバリエーションが複数定義されています。以降の各セクションでは、このうち複数のロックタイプについて使用方法を示します。

POSIXミューテックスロックの使用

POSIXミューテックスロックは、いずれのアプリケーションからも、非常に簡単に使用できます。ミューテックスロックを作成するには、`pthread_mutex_t`構造体を宣言して初期化します。ミューテックスロックをロックおよびロック解除するには、`pthread_mutex_lock`関数および`pthread_mutex_unlock`関数を使用します。リスト4-2に、POSIXスレッドミューテックスロックを初期化および使用するために必要な基本的なコードを示します。ロックが不要になった場合は、そのまま`pthread_mutex_destroy`を呼び出してロックデータ構造体を解放してください。

リスト4-2 ミューテックスロックの使用

```
pthread_mutex_t mutex;
void MyInitFunction()
{
    pthread_mutex_init(&mutex, NULL);
}

void MyLockingFunction()
{
    pthread_mutex_lock(&mutex);
    // 作業を実行します。
    pthread_mutex_unlock(&mutex);
}
```

注： 前述のコードは、POSIXスレッドミューテックス関数の基本的な使用方法を示すことを目的とした、単純化された例です。実際のコードでは、これらの関数が返すエラーコードを検査して、適宜処理する必要があります。

NSLockクラスの使用

NSLockオブジェクトには、Cocoaアプリケーション用の基本的なミューテックスが実装されています。NSLockなどすべてのロックのインターフェイスを実際に定義しているのは、NSLockingプロトコルです。このプロトコルに`lock`メソッドおよび`unlock`メソッドが定義されています。任意のミューテックスの場合とまったく同様に、これらのメソッドを使用してロックを取得および解放します。

NSLockクラスには、標準的なロック動作に加えて`tryLock`メソッドおよび`lockBeforeDate:`メソッドが追加されています。`tryLock`メソッドでは、ロックの取得を試行しますが、ロックできない場合はブロックしないで、そのままを`NO`返します。`lockBeforeDate:`メソッドはロックの取得を試行しますが、指定された時間制限内にロックを取得しない場合は、スレッドをブロック解除して`NO`を返します。

次の例では、複数のスレッドによってデータを計算している視覚表示の更新を、NSLockオブジェクトによって調整する方法を示します。このスレッドでは、ロックをすぐに取得できない場合、ロックを取得できるまで単に計算を続行してから表示を更新します。

```
BOOL moreToDo = YES;
NSLock *theLock = [[NSLock alloc] init];
```

```

...
while (moreToDo) {
    /* Do another increment of calculation */
    /* until there's no more to do. */
    if ([theLock tryLock]) {
        /* Update display used by all threads. */
        [theLock unlock];
    }
}

```

@synchronizedディレクティブの使用

@synchronizedディレクティブは、Objective-Cコードにオンザフライでミューテックスロックを作成する便利な方法です。@synchronizedディレクティブは、ほかのすべてのミューテックスロックと同じ機能を持ち、別のスレッドで同じロックを同時に取得できないようにします。ただし、この場合は、ミューテックスもロックオブジェクトも直接作成する必要はありません。次の例に示すように、任意のObjective-Cオブジェクトをロックとして使用するだけです。

```

- (void)myMethod:(id)anObj
{
    @synchronized(anObj)
    {
        // 中括弧で囲まれているすべての要素が@synchronizedディレクティブによって保護されま
        す。
    }
}

```

@synchronizedディレクティブに渡すオブジェクトは、保護されるブロックを識別するために使用される一意の識別子です。それぞれのスレッドでanObjパラメータに別々のオブジェクトを渡して、2つの別々のスレッドで前述のメソッドを実行した場合、各スレッドは独自のロックを取得し、他方によってブロックされないで処理を続行します。一方、両方に同じオブジェクトを渡した場合は、スレッドの1つが先にロックを取得し、もう一方は最初のスレッドでクリティカルセクションを完了するまでブロックされます。

@synchronizedブロックを使用すると、暗黙のうちに、予防策としての例外ハンドラが保護コードに追加されます。このハンドラは、例外がスローされる場合にミューテックスを自動的に解放します。つまり、@synchronizedディレクティブを使用するには、Objective-Cの例外処理をコードで有効にする必要があります。暗黙の例外ハンドラによる余分なオーバーヘッドを生じさせないためには、ロッククラスの使用を検討する必要があります。

@synchronizedディレクティブの詳細については、『*The Objective-C Programming Language*』を参照してください。

その他のCocoaロックの使用

以降の各セクションでは、その他の複数のCocoaロックについて、そのロックを使用するためのプロセスを説明します。

NSRecursiveLockオブジェクトの使用

NSRecursiveLockクラスは、スレッドをデッドロックにしないで同じスレッドによって複数回取得できるロックを定義します。再帰ロックでは、正常に取得されているロックの回数を常にカウントしています。ロックを正常に取得できた回数だけ、対応するロック解除を呼び出して、バランスを保つ必要があります。すべてのロック呼び出しとロック解除呼び出しのバランスが取れているときに限り、ロックが実際に解放されて、ほかのスレッドでロックを取得できるようになります。

このタイプのロックは、名前が暗示しているように、再帰関数の内部で再帰によってスレッドをブロックしないためによく使われます。同様に再帰しない場合に使用して、やはりロックを取得する必要のある動作を持つ関数を呼び出すことができます。再帰を介してロックを取得する単純な再帰関数の例を次に示します。このコードでNSRecursiveLockオブジェクトを使用しない場合、スレッドはこの関数を再度呼び出したときにデッドロックになります。

```
NSRecursiveLock *theLock = [[NSRecursiveLock alloc] init];
```

```
void MyRecursiveFunction(int value)
{
    [theLock lock];
    if (value != 0)
    {
        --value;
        MyRecursiveFunction(value);
    }
    [theLock unlock];
}
```

```
MyRecursiveFunction(5);
```

注： 再帰ロックは、すべてのロック呼び出しとバランスが取れるだけのロック解除呼び出しを実行したときに解放されるため、パフォーマンス上の潜在的な影響を慎重に比較検討したうえでパフォーマンスロックの使用を決定する必要があります。ロックを長期間保持している場合、ほかのスレッドは再帰が完了するまでブロックされます。コードを書き直して再帰呼び出しをなくすことができたり、再帰ロックを使用する必要性をなくしたりできるのであれば、通常は、パフォーマンスを向上できます。

NSConditionLockオブジェクトの使用

NSConditionLockオブジェクトは、特定の値を使用してロックおよびロック解除できるミューテックスロックを定義します。このタイプのロックと条件変数（「[条件変数](#)」（64 ページ）を参照）を混同しないでください。動作は条件変数と多少似ていますが、実装方法は大きく異なります。

通常は、一方のスレッドで使用するデータをもう一方のスレッドで生成する場合など、スレッドで特定の順序でタスクを実行する必要がある場合にNSConditionLockオブジェクトを使用します。プロデューサの実行中にコンシューマはプログラム独自の条件を使用してロックを取得します（条件自体は、独自に定義する整数値です）。プロデューサが終了すると、ロックが解除されてロックの条件に適切な整数値を設定することによりコンシューマスレッドをスリープ解除させます。コンシューマスレッドはデータの処理に進みます。

NSConditionLockオブジェクトで応答するロックメソッドおよびロック解除メソッドは、任意に組み合わせで使用できます。たとえば、lockメッセージとunlockWithCondition:を組み合わせたり、lockWhenCondition:メッセージとunlockを組み合わせたりできます。もちろん、この後者の組み合わせの場合は、ロックが解除されても特定の条件値を待機しているいずれのスレッドも解放されないことがあります。

次の例は、条件ロックを使用して、このプロデューサとコンシューマの問題を処理する方法を示します。データのキューを含むアプリケーションがあるとします。プロデューサスレッドはキューにデータを追加し、コンシューマスレッドはキューからデータを取り出します。プロデューサは特定の条件を満たすまで待機する必要はありませんが、データをキューに安全に追加するには、ロックが使用可能になるまで待機する必要があります。

```
id condLock = [[NSConditionLock alloc] initWithCondition:NO_DATA];

while(true)
{
    [condLock lock];
    /* Add data to the queue. */
    [condLock unlockWithCondition:HAS_DATA];
}
```

ロックの初期条件はNO_DATAに設定されているため、プロデューサスレッドでは最初ロックを簡単に取得できます。プロデューサスレッドはキューにデータを設定して条件にHAS_DATAを設定します。プロデューサスレッドのその後の反復では、キューが空になっているかまだデータを含んでいるかを問わず、到着した新しいデータを追加できます。プロデューサスレッドは、コンシューマスレッドがキューからデータを取り出しているときに限り、ブロックされます。

コンシューマスレッドでは、処理対象のデータを必要とするため、特定の条件を使用してキューで待機します。プロデューサがキューにデータを追加すると、コンシューマスレッドがスリープ解除してロックを取得します。コンシューマスレッドはキューからデータを取り出してキューのステータスを更新できます。次の例は、コンシューマスレッドの処理ループの基本構造を示します。

```
while (true)
{
    [condLock lockWhenCondition:HAS_DATA];
    /* Remove data from the queue. */
    [condLock unlockWithCondition:(isEmpty ? NO_DATA : HAS_DATA)];

    // データをローカルで処理します。
}
```

NSDistributedLockオブジェクトの使用

NSDistributedLockクラスは、複数ホストの複数アプリケーションでを使用して、ファイルなど何らかの共有リソースへのアクセスを制限できます。このロック自体は、事実上は、ファイルやディレクトリなどのファイルシステム項目を使用して実装されたミューテックスロックです。NSDistributedLockオブジェクトは、そのオブジェクトを使用するすべてのアプリケーションからロックを書き込める場合に限り使用可能です。これは、通常は、アプリケーションを実行するすべてのコンピュータからアクセスできるファイルシステムにこのオブジェクトを配置することを意味します。

ほかのタイプのロックと異なり、NSDistributedLockはNSLockingプロトコルに準拠しておらず、したがってlockメソッドを持ちません。lockメソッドがあればスレッドの実行をブロックすることになり、所定の速度でロックをポーリングするようシステムに要求します。この不利な条件をコードに課す代わりに、NSDistributedLockにはtryLockメソッドが用意されており、ポーリングを実行するかどうかを決めることができます。

NSDistributedLockオブジェクトは、ファイルシステムを使用して実装されているため、オーナーが明示的に解放しない限り解放されません。分散ロックを保持している間にアプリケーションがクラッシュした場合、ほかのクライアントは保護されたリソースにアクセスできなくなります。この場合は、breakLockメソッドを使用して既存のロックを破棄することにより、ロックを取得できます。通常は、オーナープロセスが停止しておりロックを解放できないことが確実な場合を除き、ロックを破棄すべきではありません。

ほかのタイプのロック同様、NSDistributedLockオブジェクトを使用し終えた場合は、unlockメソッドを呼び出してオブジェクトを解放します。

条件変数の使用

条件変数は、必要な順序に合わせて操作を進行させるために使用できる、特別なタイプのロックです。ミューテックスロックとは少し異なります。ある条件で待機しているスレッドは、その条件のシグナルが別のスレッドから明示的に送られるまでブロックされたままになります。

オペレーティングシステムの実装に関する特性により、コードから実際にシグナルが送信されなかった場合でも、偽の成功を報告して条件ロックから復帰することが許容されています。こうした偽のシグナルを原因とする問題を回避するには、常に述語と組み合わせて条件ロックを使用する必要があります。述語は、スレッドで処理を進めてよいかどうかを判別する上で、現実性の高い方法になります。スレッドにシグナルを送って述語を設定できるまでは、条件によってスレッドは単にスリープ状態のままになります。

以降の各セクションでは、コードで条件変数を使用する方法を示します。

NSConditionクラスの使用

NSConditionクラスの動作はPOSIXの条件変数と同じですが、ロックと条件変数の両方に必要なデータ構造体を単一のオブジェクトにラップしています。その結果、ミューテックスのようにロックでき、条件変数のように待機できるオブジェクトになっています。

リスト4-3に、NSConditionオブジェクトを待機しているイベントのシーケンスを説明するコードを示します。cocoaCondition変数は、NSConditionオブジェクトを含んでいます。timeToDoWork変数は、条件のシグナルを送信する直前に別のスレッドからインクリメントされる整数です。

リスト 4-3 Cocoa条件変数の使用

```
[cocoaCondition lock];
while (timeToDoWork <= 0)
    [cocoaCondition wait];

timeToDoWork--;

// 実際の作業を実行します。
```

```
[cocoaCondition unlock];
```

リスト 4-4にCocoa条件のシグナル送信と述語変数のインクリメントに使用されるコードを示します。シグナルを送信する前に必ず条件をロックする必要があります。

リスト 4-4 Cocoa条件のシグナル送信

```
[cocoaCondition lock];
timeToDoWork++;
[cocoaCondition signal];
[cocoaCondition unlock];
```

POSIX条件変数の使用

POSIXスレッド条件ロックでは、条件データ構造体とミューテックスの両方を使用する必要があります。この2つは単独のロック構造体ですが、ミューテックスロックは、実行時に条件構造体と密接に結合されます。シグナル待機のスレッドでは、常に同じミューテックスロックおよび条件構造体の組み合わせを使用する必要があります。組み合わせを変更するとエラーが発生することがあります。

リスト 4-5に、条件および述語の基本的な初期設定および使用方法を示します。待機するスレッドは、条件およびミューテックスロックの両方の初期設定後に、`ready_to_go`変数を述語として使用してwhileループに入ります。述語が設定され、それに続いて条件のシグナルが送信されたときだけ、待機しているスレッドはスリープ解除して作業の実行を開始します。

リスト 4-5 POSIX条件変数の使用

```
pthread_mutex_t mutex;
pthread_cond_t condition;
Boolean ready_to_go = true;

void MyCondInitFunction()
{
    pthread_mutex_init(&mutex);
    pthread_cond_init(&condition, NULL);
}

void MyWaitOnConditionFunction()
{
    // ミューテックスをロックします。
    pthread_mutex_lock(&mutex);

    // 述語がすでに設定されている場合、whileループは迂回されます。
    // 設定されていない場合、スレッドは述語が設定されるまでスリープ状態になります。
    while(ready_to_go == false)
    {
        pthread_cond_wait(&condition, &mutex);
    }

    // 作業を実行します（ミューテックスはロックされたままです）。

    // 述語をリセットし、ミューテックスを解放します。
    ready_to_go = false;
    pthread_mutex_unlock(&mutex);
}
```

```
}
```

シグナルを送信するスレッドで、述語の設定と条件ロックへのシグナル送信の両方を行います。リスト 4-6にこの動作を実装するためのコードを示します。この例では、待機しているスレッド間で競合状態を発生させないために、ミューテックスの内部で条件のシグナルを送信しています。

リスト 4-6 条件ロックへのシグナル送信

```
void SignalThreadUsingCondition()
{
    // この時点で、実行する必要がある作業がほかのスレッドにあります。
    pthread_mutex_lock(&mutex);
    ready_to_go = true;

    // 作業を開始するようほかのスレッドにシグナルを送信します。
    pthread_cond_signal(&condition);

    pthread_mutex_unlock(&mutex);
}
```

注： 前述のコードは、POSIXスレッド条件関数の基本的な使用方法を示すことを目的とした、単純化された例です。実際のコードでは、これらの関数が返すエラーコードを検査して、適宜処理する必要があります。

スレッドの安全性のまとめ

この付録では、Mac OS XおよびiOSの主要なフレームワークに関する高レベルのスレッドの安全性について説明します。この付録に示す情報は、変わる可能性があります。

Cocoa

複数のスレッドからCocoaを使用する場合は、次のようなガイドラインが適用されます。

- 不変オブジェクトは、一般にスレッドセーフです。一度これらのオブジェクトを作成すれば、スレッド間で安全に受け渡しできます。一方、可変オブジェクトは、一般にスレッドセーフではありません。スレッド化されたアプリケーションで可変オブジェクトを使用するには、アプリケーションを適切に同期させる必要があります。詳細については、「[可変と不変](#)」（82 ページ）を参照してください。
- 「スレッドセーフでない」と見なされる多くのオブジェクトが安全でないのは、複数スレッドから使用する場合に限ってのことです。これらのオブジェクトの多くは、同時に1つのスレッドしかない場合は、任意のスレッドから使用できます。アプリケーションのメインスレッドに厳密に限定されているオブジェクトの場合は、メインスレッドで呼び出されます。
- アプリケーションのメインスレッドでは、イベントを処理します。ほかのスレッドがイベントパスに含まれていてもApplication Kitは動作しますが、操作の実行順序が変わる可能性があります。
- スレッドを使用してビューを描画する場合は、すべての描画コードをNSViewのlockFocusIfCanDrawメソッドとunlockFocusメソッドで囲む必要があります。
- POSIXスレッドとCocoaを併用するには、まず、Cocoaをマルチスレッドモードにする必要があります。詳細については、「[CocoaアプリケーションでのPOSIXスレッドの使用](#)」（28 ページ）を参照してください。

Foundationフレームワークのスレッドの安全性

Foundationフレームワークはスレッドセーフで、Application Kitフレームワークはスレッドセーフでないという誤解があります。残念ながら、これは極端な一般化であり、多少語弊があります。それぞれのフレームワークにスレッドセーフな部分とスレッドセーフでない部分があります。以降の各セクションでは、Foundationフレームワークのスレッドの安全性全般について説明します。

スレッドセーフなクラスおよび関数

次のクラスおよび関数は、一般に、スレッドセーフであると見なされます。あらかじめロックを取得することなく、複数のスレッドから同じインスタンスを使用できます。

NSArray
NSAssertionHandler
NSAttributedString
NSDate
NSCalendarDate
NSCharacterSet
NSConditionLock
NSConnection
NSData
NSDate
NSDecimal関数
NSDecimalNumber
NSDecimalNumberHandler
NSDeserializer
NSDictionary
NSDistantObject
NSDistributedLock
NSDistributedNotificationCenter
NSException
NSFileManager (Mac OS X v10.5以降)
NSHost
NSLock
NSLog/NSLogv
NSMethodSignature
NSNotification
NSNotificationCenter
NSNumber
NSObject
NSPortCoder
NSPortMessage
NSPortNameServer
NSProtocolChecker
NSProxy
NSRecursiveLock
NSSet
NSString
NSThread
NSTimer
NSTimeZone
NSUserDefaults
NSValue
オブジェクト割り当て関数および保持カウント関数
ゾーン関数およびメモリ関数

スレッドセーフでないクラス

次のクラスおよび関数は、一般にスレッドセーフではありません。通常は、一度に1つのスレッドからだけ使用する限り、任意のスレッドからこれらのクラスを使用できます。その他の詳細については、クラスの資料を参照してください。

NSArchiver
NSAutoreleasePool
NSBundle
NSCalendar
NSCoder
NSCountedSet
NSDateFormatter
NSEnumerator
NSFileHandle
NSFormatter
NSHashTable関数
NSInvocation
NSJavaSetup関数
NSMapTable関数
NSMutableArray
NSMutableAttributedString
NSMutableCharacterSet
NSMutableData
NSMutableDictionary
NSMutableSet
NSMutableString
NSNotificationQueue
NSNumberFormatter
NSPipe
NSPort
NSProcessInfo
NSRunLoop
NSScanner
NSSerializer
NSTask
NSUnarchiver
NSUndoManager
ユーザ名関数およびホームディレクトリ関数

オブジェクトNSSerializer、NSArchiver、NSCoder、およびNSEnumerator自体はスレッドセーフですが、これらのオブジェクトにラップされているデータオブジェクトを使用中に変更すると安全でないため、ここにリストしてあることに注意してください。たとえば、アーカイバの場合に、アーカイブ中のオブジェクトグラフを変更することは安全ではありません。列挙子の場合、いずれのスレッドにおいても、列挙されたコレクションを変更することは安全ではありません。

メインスレッド専用クラス

次のクラスは、アプリケーションのメインスレッドからだけ使用する必要があります。

NSAppleScript

可変と不変

不変オブジェクトは一般にスレッドセーフです。一度作成すれば、スレッド間で安全に受け渡しできます。当然ながら、不変オブジェクトを使用するときは、やはり参照カウントを適切に使用するよう留意する必要があります。保持しなかったオブジェクトを解放することは適切でなく、後で例外が発生するおそれがあります。

可変オブジェクトは、一般にスレッドセーフではありません。スレッド化されたアプリケーションで可変オブジェクトを使用する場合は、ロックを使用して可変オブジェクトへのアクセスを同期させる必要があります（詳細については「[アトミック操作](#)」（61 ページ）を参照してください）。一般に、NSMutableArray、NSMutableDictionaryなどのコレクションクラスは、変更に関してスレッドセーフではありません。つまり、1つ以上のスレッドで同じ配列を変更していると、問題が発生することがあります。読み取りおよび書き取りが行われる箇所を囲むようにロックを設定してスレッドの安全性を確保する必要があります。

不変オブジェクトを返すことになっているメソッドであっても、返されるオブジェクトは不変であるとそのまま想定しないでください。返されるオブジェクトは、メソッドの実装に応じて可変と不変のいずれの場合もあります。たとえば、NSStringの戻り型を持つメソッドであっても、実装によって実際にはNSMutableStringを返すことがあります。不変オブジェクトを確実に入手するには、不変コピーを作成する必要があります。

再入可能性

再入は、同じオブジェクトまたは異なるオブジェクトのほかの操作を「呼び出す」操作がある場合にのみ発生します。オブジェクトの保持および解放は、見逃されがちですが、そのような「呼び出し」の1つです。

次の表に、再入可能であることが明白なFoundationフレームワークのクラスを示します。ほかのすべてのクラスは、再入可能な場合と再入可能でない場合があるか、今後再入可能になる可能性があります。再入可能性の完全な分析は実施されておらず、このリストは網羅的でない場合があります。

分散オブジェクト

- NSConditionLock
- NSDistributedLock
- NSLock
- NSLog/NSLogv
- NSNotificationCenter
- NSRecursiveLock
- NSRunLoop
- NSUserDefaults

クラスの初期化

Objective-Cランタイムシステムは、すべてのクラスオブジェクトに対して`initialize`メッセージを送信します。各クラスは、ほかのいずれのメッセージよりも先にこのメッセージを受け取ります。その結果、クラスの使用に先立ってクラス独自のランタイム環境を準備できます。このランタイムにより、マルチスレッドアプリケーションでは、必ず1つのスレッドだけ、具体的にはたまたま最初のメッセージをクラスに送信したスレッドだけが`initialize`メソッドを実行するようになります。最初のスレッドがまだ`initialize`メソッドを実行している間に2番目のスレッドがこのクラスにメッセージを送信しようとした場合、2番目のスレッドは`initialize`メソッドの実行が終了するまでブロックされます。この間、最初のスレッドでは、クラスのほかのメソッドを引き続き呼び出すことができます。`initialize`メソッドでは、そのクラスの2つ目のスレッドを呼び出すメソッドを使用しないでください。使用すると2つのスレッドがデッドロックを起こします。

Mac OS Xバージョン10.1.x以前にはバグがあり、クラスの`initialize`メソッドを別のスレッドで実行しているうちに、スレッドからそのクラスにメッセージを送信できました。メッセージの送信後、まだ完全に初期化されていない値にアクセスでき、アプリケーションをクラッシュさせていました。この問題が発生する場合は、ロックを取り入れて初期化後まで値へのアクセスを防ぐか、マルチスレッド化する前にクラスを必ず初期化する必要があります。

自動解放プール

各スレッドは、独自の`NSAutoreleasePool`オブジェクトのスタックを管理しています。Cocoaは、現在のスレッドのスタックで自動解放プールが常に利用可能であると予期しています。自動解放プールが利用できない場合は、オブジェクトは解放されないためメモリリークが発生します。`NSAutoreleasePool`オブジェクトは、アプリケーションのメインスレッドでは、`Application Kit`に基づいて自動的に作成および破棄されます。ただし、二次スレッドおよび`Foundation`のみのアプリケーションでは、Cocoaを使用する前に独自に作成する必要があります。スレッドが長時間継続し、大量の自動解放オブジェクトを生成する可能性がある場合は、（`Application Kit`のメインスレッドで行っているのと同様に）定期的に自動解放プールを破棄して作成する必要があります。この処理を組み込まない場合、自動解放オブジェクトが蓄積されてメモリ占有量が増えます。デタッチされたスレッドでCocoaを使用しない場合は、自動解放プールを作成する必要はありません。

実行ループ

すべてのスレッドには実行ループが必ず1つだけあります。一方、各実行ループ、つまり各スレッドは、実行ループの実行中に監視する入力ソースを決める、独自の一組の入力モードを持ちます。実行ループに定義されている入力モードが別の実行ループに定義されている入力モードと同じ名前であったとしても、入力モードは互いに影響しません。

`Application Kit`に基づくアプリケーションの場合、メインスレッドの実行ループは自動的に実行されますが、二次スレッドおよび`Foundation`のみのアプリケーションでは、独自に実行ループを実行する必要があります。デタッチされたスレッドが実行ループに入らない場合、デタッチされたメソッドの実行が終了され次第、スレッドが終了されます。

外見の印象と異なり、`NSRunLoop`クラスはスレッドセーフではありません。このクラスのインスタンスメソッドは、必ずこのクラスのオーナーであるスレッドから呼び出す必要があります。

Application Kitフレームワークのスレッドの安全性

以降の各セクションでは、`Application Kit`フレームワークのスレッドの安全性全般について説明します。

スレッドセーフでないクラス

次のクラスおよび関数は、一般にスレッドセーフではありません。通常は、一度に1つのスレッドからだけ使用する限り、任意のスレッドからこれらのクラスを使用できます。その他の詳細については、クラスの資料を参照してください。

- `NSGraphicsContext` 詳細については「[NSGraphicsContextの制約事項](#)」（85 ページ）を参照してください。
- `NSImage` 詳細については「[NSImageの制約事項](#)」（85 ページ）を参照してください。
- `NSResponder`
- `NSWindow` およびすべての下位クラス。詳細については「[ウィンドウの制約事項](#)」（84 ページ）を参照してください。

メインスレッド専用クラス

次のクラスは、アプリケーションのメインスレッドからだけ使用する必要があります。

- `NSCell` およびすべての下位クラス
- `NSView` およびすべての下位クラス。詳細については「[NSViewの制約事項](#)」（85 ページ）を参照してください。

ウィンドウの制約事項

二次スレッドにウィンドウを作成できます。**Application Kit**では、競合状態を避けるために、ウィンドウに関連するデータ構造体を必ずメインスレッドで割り当てます。多数のウィンドウを並行処理するアプリケーションでは、ウィンドウオブジェクトのリークが発生するおそれがあります。

二次スレッドにモーダルウィンドウを作成できます。**Application Kit**では、メインスレッドでモーダルループを実行している間、呼び出し中の二次スレッドをブロックします。

イベント処理の制約事項

アプリケーションのメインスレッドでは、イベントを処理します。メインスレッドは、`NSApplication` の `run` メソッド内でブロックされるスレッドで、通常はアプリケーションの `main` 関数内で呼び出されますほかのスレッドがイベントパスに含まれている場合に **Application Kit** が作業を続行している間は、操作の実行順序が変わる可能性があります。たとえば、別々の2つのスレッドでキーイベントに応答している場合は、受け取るキーの順序が変わる可能性があります。メインスレッドでイベントを処理すれば、ユーザ体験の一貫性を向上できます。受け取ったイベントは、必要に応じてさらに処理するために二次スレッドにディスパッチできます。

二次スレッドから `NSApplication` の `postEvent:atStart:` メソッドを呼び出すと、メインスレッドのイベントキューにイベントを送ることができます。ただし、ユーザ入力イベントに関して順序は保証されません。イベントキューのイベントは、やはりアプリケーションのメインスレッドで処理します。

描画の制約事項

NSBezierPathクラスおよびNSStringクラスなどのグラフィックス関数およびクラスによって描画する場合、Application Kitは、一般にスレッドセーフです。個々のクラスの詳細な使用方法を以降の各セクションで説明します。描画とスレッドに関するその他の詳細情報については、『Cocoa Drawing Guide』で説明されています。

NSViewの制約事項

NSViewクラスは、わずかな例外を除いて、一般にスレッドセーフです。NSViewオブジェクトの作成、破棄、サイズ変更、移動、およびその他の操作は、必ずアプリケーションのメインスレッドから実行する必要があります。二次スレッドからの描画は、lockFocusIfCanDrawおよびunlockFocusの呼び出しで描画用の呼び出しを囲んでいれば、スレッドセーフです。

アプリケーションの二次スレッドからメインスレッドにビューの一部分を再描画させる場合、display、setNeedsDisplay:、setNeedsDisplayInRect:、およびsetViewsNeedDisplay:などのメソッドは使用しないでください。代わりに、メインスレッドにメッセージを送信するか、performSelectorOnMainThread:withObject:waitUntilDone:メソッドを使用してこれらのメソッドを呼び出してください。

ビューシステムのグラフィックス状態（gstates）はスレッドごとです。グラフィックス状態の使用は、単一スレッドアプリケーションよりも優れた描画性能を実現する方法の1つでしたが、現在ではあてはまりません。グラフィックス状態の使用方法を誤ると、実際にはメインスレッドで描画するより効率の悪い描画コードになるおそれがあります。

NSGraphicsContextの制約事項

NSGraphicsContextクラスは、基礎のグラフィックスシステムによる描画コンテキストを表します。各NSGraphicsContextインスタンスは、独自の独立したグラフィックス状態を持ちます。これには、座標系、クリッピング、現在のフォントなどが含まれます。NSWindowのインスタンスごとに、このクラスのインスタンスがメインスレッドに自動的に作成されます。二次スレッドから描画を実行する場合は、必ずNSGraphicsContextの新しいインスタンスがそのスレッド専用で作成されます。

二次スレッドから描画を実行する場合は、必ず、手動で描画呼び出しをフラッシュする必要があります。Cocoaでは、二次スレッドから描画された内容でビューを自動的に更新しないため、描画を完了した時点でNSGraphicsContextのflushGraphicsメソッドを呼び出す必要があります。アプリケーションでメインスレッドだけから内容を描画する場合は、描画呼び出しをフラッシュする必要はありません。

NSImageの制約事項

1つのスレッドで、NSImageオブジェクトを作成し、画像バッファに描画し、メインスレッドに画像バッファを渡して描画させることができます。基礎の画像キャッシュはすべてスレッドで共有されます。画像およびキャッシュの機能の詳細については、『Cocoa Drawing Guide』を参照してください。

Core Dataフレームワーク

Core Dataフレームワークは、スレッド処理をおおむねサポートしていますが、いくつかの使用上の注意事項があります。これらの注意事項については、『*Core Data Programming Guide*』の「Concurrency with Core Data」を参照してください。

Core Foundation

Core Foundationは、注意深く正しくプログラミングすれば十分にスレッドセーフであるため、スレッドの競合に関連する問題は発生しません。不変オブジェクトの照会、保持、解放、および受け渡しなどの一般的なケースでは、スレッドセーフです。複数のスレッドから照会される場合がある中央管理の共有オブジェクトであっても、確実にスレッドセーフです。

Cocoa同様、Core Foundationは、オブジェクトまたはオブジェクトの内容の変更に関しては、スレッドセーフではありません。たとえば、可変データまたは可変配列オブジェクトを変更することは、当然ながらスレッドセーフではありません。ただし、いずれの場合も、不変配列に格納されているオブジェクトは変更されません。この動作の背景の1つはパフォーマンスです。パフォーマンスは、このような場合非常に重要です。さらに、このレベルでは、通常は、完全なスレッドの安全性を実現できません。たとえば、コレクションから取得したオブジェクトを保持する結果生じる不定な動作を排除できません。格納されているオブジェクトを保持するための呼び出しを行う前に、コレクション自体が解放される可能性があります。

Core Foundationオブジェクトに複数のスレッドからアクセスして変更する場合は、アクセスポイントでロックを使用することによって、同時アクセスからの保護を実装する必要があります。たとえば、Core Foundation配列のオブジェクトを列挙するコードでは、列挙ブロックを囲む適切なロック呼び出しを使用して、それ以外の場所からの変更から配列を保護する必要があります。

用語解説

アプリケーション (application) **プログラム**のうち、グラフィカルインターフェイスを使ってユーザとやりとりするもの。

条件変数 (condition) リソースへのアクセスを同期するために使う構成体。ある条件を満たすまで待機するスレッドは、ほかのスレッドがシグナルとして明示的に条件を伝えるまで、処理を続行できません。

クリティカルセクション (critical section) コードのうち、同時には1つのスレッドでしか実行されない部分。

入力ソース (input source) スレッドに送信される非同期イベントのソース。ポートベースのものとして直接トリガをかけるものがあり、スレッドの実行ループにアタッチする必要があります。

合流可能なスレッド (joinable thread) スレッドのうち、終了時にリソースが直ちに回収されないもの。明示的にデタッチするか、ほかのスレッドに合流してからでないと、リソースを回収できません。合流される側のスレッドに戻り値を返します。

メインスレッド (main thread) **スレッド**のうち、プロセス生成時に自動的に生成されるもの。プログラムのメインスレッドが終了すれば、プロセスも終了します。

ミューテックス (mutex) 共有リソースに対する相互排除アクセスを可能にするロック。同時には1つのスレッドだけが保持できます。ほかのスレッドが保持しているミューテックスを獲得しようとする、そのスレッドは、ロックを獲得できるまでの間スリープ状態になります。

オペレーションオブジェクト (operation object) NSOperationクラスのインスタンス。タスクに関連するコードとデータを、実行可能な単位にラップします。

オペレーションキュー (operation queue)

NSOperationQueueクラスのインスタンス。オペレーションオブジェクトの実行を管理します。

プロセス (process) アプリケーションやプログラムの実行時インスタンス。ほかのプログラムとは独立した、自分だけの仮想メモリ空間とシステムリソース（ポート権限を含む）を持ちます。プロセスにはスレッドが少なくとも1つある（メインスレッド）ほか、いくつでも追加で生成できます。

プログラム (program) 何らかのタスクを実行できる、コードとリソースを組み合わせたもの。グラフィカルアプリケーションもプログラムの一種ですが、単にプログラムと言った場合、グラフィカルユーザインターフェイスはなくても構いません。

再帰ロック (recursive lock) ロックのうち、同じスレッドによって複数回ロックできるもの。

実行ループ (run loop) イベントを受信して適切なハンドラにディスパッチする、イベント処理ループ。

実行ループモード (run loop mode) 入力ソース、タイマーソース、実行ループオブザーバを一式集めたものに、名前を与えて識別できるようにしたもの。実行ループをある「モード」で動かすと、関係するソースやオブザーバのみを監視するようになります。

実行ループオブジェクト (run loop object)

NSRunLoopクラスまたはCFRunLoopRef不透過型のインスタンス。スレッドにイベント処理ループを実装するためのインターフェイスを提供します。

実行ループオブザーバ (run loop observer) 実行ループの各実行フェーズにおける通知の受け手。

セマフォ (semaphore) 共有リソースへのアクセスを制限する、保護された変数。ミューテックスや条件変数はセマフォの一種です。

タスク (task) 実行すべき処理の量。ほかの技術（特にCarbon Multiprocessing Services）ではこの用語を別の意味で使っていますが、一般的には、実行すべき処理を何らかの形で定量化した、抽象的な概念を意味するものとして使います。

スレッド (thread) プロセスにおける、実行フローの単位。各スレッドには独自のスタック空間がありますが、それ以外のメモリは、同じプロセス内のほかのスレッドと共有します。

タイマーソース (timer source) スレッドに送信される同期イベントのソース。タイマーは、スケジューリングされた未来時に、イベントを1回または反復して生成します。

書類の改訂履歴

この表は「スレッドプログラミングガイド」の改訂履歴です。

日付	メモ
2010-04-28	誤字を訂正しました。
2009-05-22	オペレーションオブジェクトについての情報を『Concurrency Programming Guide』に移動しました。本書の内容をスレッドにのみ絞り込みました。
2008-10-15	オペレーションオブジェクトおよびオペレーションキューに関連したサンプルコードを更新しました。
2008-03-21	iOS用に更新しました。
2008-02-08	スレッド関連の概念およびタスクについて大幅な書き換えと更新を行いました。
	スレッドの設定についての詳細情報を追加しました。
	同期ツールに関するセクションを1つの章に再編し、アトミック操作、メモリバリア、およびvolatile変数についての情報を追加しました。
	実行ループの使用と設定についての詳細情報を追加しました。
	『Multithreading Programming Topics』から文書名を変更しました。
2007-10-31	NSOperationオブジェクトおよびNSOperationQueueオブジェクトについての情報を追加しました。
2006-04-04	実行ループについて、いくつかの新しいガイドラインを追加し、情報を更新しました。
2005-03-03	分散オブジェクトに関するサンプルコードの正確さを検証し、またその他の事項におけるサンプルコードをいくつか更新しました。
2005-01-11	NSMessagePortの代わりにNSPortを用いたポート使用例を更新しました。
	文書を再編し、Cocoaのスレッドに関するテクニック以外の内容も盛り込まれるように拡充しました。
	スレッドの概念に関する情報を更新し、MacOSXの別のスレッド処理パッケージについての情報を追加しました。
	Core Foundationのマルチスレッドに関する文書からの情報を追加しました。
	スレッド間でのソケットベースの通信に関する情報を追加しました。

日付	メモ
	Carbonスレッドの作成と使用に関する情報とサンプルコードを追加しました。
	スレッドの安全性に関するガイドラインを追加しました。
	POSIXスレッドとロックについての情報を追加しました。
	ポートベースの通信を実現するサンプルコードを追加しました。
	本書は、以前には『 <i>Multithreading</i> 』として公開されていた、スレッドについての情報を置き換えるものです。
28.07.03	サードパーティ製のライブラリにおいてライブラリのロックを使用するためのアドバイスを更新しました。
08.04.03	サードパーティ製のライブラリにおけるロック/ロック解除のバランスについての情報を明確にしました。
12.11.02	既存のトピックに改訂履歴を追加しました。