マルチスレッドプログラミングの基礎

－ 最適なマルチスレッドプログラミングのために －

2014年2月10日 初版

板垣 衛

改訂履歴

|  |  |  |  |
| --- | --- | --- | --- |
| 版 | リリース | 担当 | 改訂内容 |
| 初版 | 2014年2月10日 | 板垣 衛 | （初版） |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

目次

[ 概略 1](#_Toc378903591)

[ 目的 1](#_Toc378903592)

[ サンプルプログラムについて 1](#_Toc378903593)

[▼ サンプルプログラムのビルド・実行環境 1](#_Toc378903594)

[▼ サンプルプログラム記載の方針について 2](#_Toc378903595)

[ スレッドの仕組み 2](#_Toc378903596)

[▼ マルチタスク 3](#_Toc378903597)

[ ノンプリエンプティブなマルチタスク 3](#_Toc378903598)

[ プリエンプティブなマルチタスク 4](#_Toc378903599)

[ タイムスライス 4](#_Toc378903600)

[ 並行と並列 5](#_Toc378903601)

[▼ OSとカーネル 5](#_Toc378903602)

[ モノリシックカーネル 6](#_Toc378903603)

[ マイクロカーネル 7](#_Toc378903604)

[ ハイブリッドカーネル 7](#_Toc378903605)

[ カーネルモードとユーザーモード 7](#_Toc378903606)

[ BIOS 9](#_Toc378903607)

[▼ プロセスとスレッド 10](#_Toc378903608)

[ プロセス 10](#_Toc378903609)

[ スレッド 11](#_Toc378903610)

[ メモリ空間の保護 11](#_Toc378903611)

[ プロセスの終了時のリソースの解放 11](#_Toc378903612)

[ サービス／デーモン 12](#_Toc378903613)

[ スレッドとハードウェア 12](#_Toc378903614)

[▼ CPU 12](#_Toc378903615)

[ CISCとRISC 13](#_Toc378903616)

[▼ シングルプロセッサ／シングルコア 13](#_Toc378903617)

[▼ マルチプロセッサ／マルチコア／論理プロセッサ 14](#_Toc378903618)

[▼ ハイパースレッディング・テクノロジー 15](#_Toc378903619)

[▼ コプロセッサ 16](#_Toc378903620)

[▼ 浮動小数点演算装置（FPU）／SIMD 16](#_Toc378903621)

[ SIMD／MIMD／SISD／MISD 16](#_Toc378903622)

[▼ GPU 17](#_Toc378903623)

[▼ メインメモリ 17](#_Toc378903624)

[▼ RAMとROM 17](#_Toc378903625)

[▼ キャッシュメモリ 18](#_Toc378903626)

[▼ レジスタ 18](#_Toc378903627)

[▼ GPUメモリ／ユニファイドメモリ 19](#_Toc378903628)

[▼ DMAC 19](#_Toc378903629)

[▼ Cell 19](#_Toc378903630)

[▼ サーバー 20](#_Toc378903631)

[ マルチスレッドの意義 20](#_Toc378903632)

[▼ マルチプロセス 20](#_Toc378903633)

[▼ マルチスレッドの意義①：同時実行 21](#_Toc378903634)

[▼ マルチスレッドの意義②：同時接続 21](#_Toc378903635)

[▼ マルチスレッドの意義③：分散・高速化 21](#_Toc378903636)

[▼ ゲームの並行処理 22](#_Toc378903637)

[▼ ゲームでのスレッド活用 22](#_Toc378903638)

[ プログラムの動作原理 23](#_Toc378903639)

[▼ メモリ構造 23](#_Toc378903640)

[▼ 機械語／アセンブラ／プログラムカウンタ 27](#_Toc378903641)

[▼ スタック領域 28](#_Toc378903642)

[▼ スレッドとコンテキストスイッチ 30](#_Toc378903643)

[▼ スレッドスケジューリング 31](#_Toc378903644)

[▼ スレッド優先度 31](#_Toc378903645)

[ 様々なスレッド 33](#_Toc378903646)

[▼ fork 33](#_Toc378903647)

[▼ POSIXスレッドライブラリ版スレッド 37](#_Toc378903648)

[▼ Win32API版スレッド 42](#_Toc378903649)

[▼ C++11版スレッド 47](#_Toc378903650)

[▼ OpenMP 52](#_Toc378903651)

[▼ C++11版非同期関数 57](#_Toc378903652)

[▼ ファイバースレッド／コルーチン 61](#_Toc378903653)

[▼ SPURS／GPGPU／DirectCompute／CUDA／ATI Stream／OpenCL 65](#_Toc378903654)

[▼ クライアント・サーバー／クラウド／グリッド／RPC／ORB／Hadoop 66](#_Toc378903655)

[▼ 割り込み／システムコール 68](#_Toc378903656)

[ 割り込みの具体例：垂直同期割り込み（V-SYNC割り込み） 68](#_Toc378903657)

[ 割り込みとコンテキストスイッチ、スタック領域の配慮 69](#_Toc378903658)

[ 割り込みハンドラ／割り込みサブルーチン（ISR） 69](#_Toc378903659)

[ 割り込みハンドラの処理 70](#_Toc378903660)

[ ハードウェア割り込みとソフトウェア割り込み／イベントコールバック 70](#_Toc378903661)

[ シグナル 71](#_Toc378903662)

[ タイマー割り込み 71](#_Toc378903663)

[ システムコール 71](#_Toc378903664)

[ マルチスレッドで起こり得る問題① 71](#_Toc378903665)

[▼ データ破損（不完全なアトミック操作） 72](#_Toc378903666)

[ 解決策 72](#_Toc378903667)

[ 同期処理 73](#_Toc378903668)

[▼ スレッド間の情報共有の失敗①：コンパイラの最適化による問題 73](#_Toc378903669)

[ 解決策 76](#_Toc378903670)

[▼ スレッド間の情報共有の失敗②：CPUの最適化による問題 77](#_Toc378903671)

[ 問題点 77](#_Toc378903672)

[ アウト・オブ・オーダー実行 77](#_Toc378903673)

[ メモリ操作命令：ロード／ストア 78](#_Toc378903674)

[ メモリバリア 78](#_Toc378903675)

[ 解決策① 78](#_Toc378903676)

[ 別の解決策：ロックフリー 78](#_Toc378903677)

[ 解決策② 79](#_Toc378903678)

[▼ 困難なマルチスレッドプログラミング 80](#_Toc378903679)

[ スレッドの同期 80](#_Toc378903680)

[▼ ビジーウェイト 81](#_Toc378903681)

[▼ スリープ 81](#_Toc378903682)

[▼ スリープとYield 82](#_Toc378903683)

[ 様々な同期手法 83](#_Toc378903684)

[▼ 排他制御（ロック） 84](#_Toc378903685)

[▼ 排他制御：通常変数でロック（誤ったロック） 84](#_Toc378903686)

[ Win32API版 84](#_Toc378903687)

[▼ 排他制御：ミューテックス 87](#_Toc378903688)

[ POSIXスレッドライブラリ版 87](#_Toc378903689)

[ Win32API版（名前なし） 90](#_Toc378903690)

[ Win32API版（名前付き） 92](#_Toc378903691)

[ C++11版 95](#_Toc378903692)

[▼ 排他制御：スピンロック 98](#_Toc378903693)

[ POSIXスレッドライブラリ版 98](#_Toc378903694)

[▼ 排他制御：クリティカルセクション 100](#_Toc378903695)

[ Win32API版 101](#_Toc378903696)

[▼ 変数操作による排他制御 103](#_Toc378903697)

[▼ 変数操作による排他制御：volatile型修飾子（誤ったロック） 103](#_Toc378903698)

[ Win32API版 103](#_Toc378903699)

[▼ 変数操作による排他制御：インラインアセンブラ 106](#_Toc378903700)

[ Win32API版 106](#_Toc378903701)

[▼ 変数操作による排他制御：アトミック操作（インターロック操作） 109](#_Toc378903702)

[ Win32API版：インターロック操作 109](#_Toc378903703)

[ C++11版：アトミック操作 111](#_Toc378903704)

[ C++11版：アトミック操作（フラグ型） 113](#_Toc378903705)

[▼ 特殊な排他制御：リード・ライトロック 114](#_Toc378903706)

[ POSIXスレッドライブラリ版 114](#_Toc378903707)

[▼ 有限共有リソース使用権獲得 119](#_Toc378903708)

[▼ 有限共有リソース使用権獲得：セマフォ 120](#_Toc378903709)

[ SystemV版 120](#_Toc378903710)

[ POSIXスレッドライブラリ版（名前なし） 124](#_Toc378903711)

[ POSIXスレッドライブラリ版（名前付き） 127](#_Toc378903712)

[ Win32API版（名前なし） 131](#_Toc378903713)

[ Win32API版（名前付き） 134](#_Toc378903714)

[▼ モニター 138](#_Toc378903715)

[▼ モニター：条件変数 138](#_Toc378903716)

[ POSIXスレッドライブラリ版 139](#_Toc378903717)

[ C++11版 144](#_Toc378903718)

[▼ モニター：バリア 150](#_Toc378903719)

[ POSIXスレッドライブラリ版 150](#_Toc378903720)

[▼ モニター：イベント 157](#_Toc378903721)

[ Win32API版（名前なし） 157](#_Toc378903722)

[ Win32API版（名前付き） 163](#_Toc378903723)

[▼ モニター：アトミック操作（インターロック操作）によるビジーなモニター 169](#_Toc378903724)

[ 【用語解説】コンペア・アンド・スワップ（CAS）操作 169](#_Toc378903725)

[ Win32API版：インターロック操作 169](#_Toc378903726)

[ C++11版：アトミック操作 174](#_Toc378903727)

[ C++11版：アトミック操作のメモリオーダー指定 179](#_Toc378903728)

[▼ Call Once 182](#_Toc378903729)

[ POSIXスレッドライブラリ版 182](#_Toc378903730)

[ C++11版 184](#_Toc378903731)

[▼ 先物（promise と future） 185](#_Toc378903732)

[ C++11版 185](#_Toc378903733)

[▼ 割り込み：シグナル 186](#_Toc378903734)

[ POSIX版 187](#_Toc378903735)

[ 【用語解説】リエントラント 193](#_Toc378903736)

[ マルチスレッドで起こり得る問題② 194](#_Toc378903737)

[▼ デッドロック 194](#_Toc378903738)

[ 【問題】相互ロック 195](#_Toc378903739)

[ 【解決策】相互ロック 195](#_Toc378903740)

[ 【問題】再帰ロック 197](#_Toc378903741)

[ 【解決策】再帰ロック 197](#_Toc378903742)

[▼ 【モニターの問題】通知の取りこぼし 198](#_Toc378903743)

[ 解決策 198](#_Toc378903744)

[▼ 低速化 199](#_Toc378903745)

[ 解決策 199](#_Toc378903746)

[▼ メモリ不足 200](#_Toc378903747)

[▼ ゾンビスレッド（リソースの枯渇） 200](#_Toc378903748)

[ 様々なデータ共有手法 201](#_Toc378903749)

[▼ volatile型修飾子、アトミック型の使いどころ 201](#_Toc378903750)

[ volatile型修飾子の使いどころ 202](#_Toc378903751)

[ アトミック型の使いどころ 203](#_Toc378903752)

[▼ スレッドローカルストレージ（TLS） 204](#_Toc378903753)

[ スレッドローカルストレージの使い方 204](#_Toc378903754)

[ スレッドローカルストレージの使いどころ 205](#_Toc378903755)

[ スレッドローカルストレージの注意点① 205](#_Toc378903756)

[ スレッドローカルストレージの注意点② 206](#_Toc378903757)

[▼ プロセス間のデータ共有①：共有メモリ 206](#_Toc378903758)

[ SystemV共有メモリ 207](#_Toc378903759)

[ メモリマップトファイル 207](#_Toc378903760)

[▼ プロセス間のデータ共有②：プロセス間通信 207](#_Toc378903761)

[ シグナル 207](#_Toc378903762)

[ ウインドウメッセージ 207](#_Toc378903763)

[ パイプ 208](#_Toc378903764)

[ 名前付きパイプ（Uninx） 208](#_Toc378903765)

[▼ プロセス間のデータ共有③：ネットワーク通信 209](#_Toc378903766)

[ 名前付きパイプ（Windows） 209](#_Toc378903767)

[ ソケット通信 209](#_Toc378903768)

[ メッセージキューサービス 210](#_Toc378903769)

[ 並列処理の活用 210](#_Toc378903770)

[▼ MapReduce 210](#_Toc378903771)

[ 耐障害性／スケールアウト 212](#_Toc378903772)

# 概略

マルチスレッドプログラミングを行う上で、基礎となる用語や仕組み、技術を解説する。

適切なマルチスレッドプログラミングを行うためには、プログラムが動作するハードウェア環境、OS、プログラムの動作原理を意識する必要がある。マルチスレッドを解説する多くのドキュメントも、そうした背景知識を前提にしていることが多い。

本書は、マルチスレッドのプログラミング技法を説明する前に、こうした周辺知識・関連知識の説明を、広範囲に、順を追って説明する。

# 目的

本書は、マルチスレッドプログラミングに対する理解を深め、環境と目的に合わせた「最適な」マルチスレッドプログラム構造の設計に導くことを目的とする。

本書ではマルチスレッドプログラミングに関係する要素を広範囲に取り上げているが、一つ一つの要素はあまり深く掘り下げていない。そのため、実際のプログラミングの際は専門のドキュメントを参照することになるが、それらはとかく専門用語にあふれている。本書は、そのようなドキュメントを読む解くための手がかりとして、専門用語の説明書・手引き書となることも目的とする。

# サンプルプログラムについて

本書には多数のサンプルプログラムを、その実行結果とともに記載している。

## サンプルプログラムのビルド・実行環境

実行結果には、コンパイラのバージョンはもとより、並列処理の性能はハードウェアの影響を強く受けるため、サンプルプログラムのビルド・動作環境を提示しておく。

**【Windows系】**

* OS：Microsft Windos8.1, 64bit
* CPU：Intel® Core™ i7-3770K VPU @ 3.5GHz Core×4, HT有効化, Socket×1

⇒ 論理プロセッサ×8

* Compiler：Microsoft Visual Studio 2013 Professional

⇒標準の Release ビルド設定でビルドしたプログラムを実行

**【Unix系】**

* OS：CentOS 6.4, 64bit
* CPU：仮想プロセッサ×4 ※上記WindowsのHyper-V仮想マシンとして実行
* Compiler：GCC 4.4.7

⇒最適化レベル –O3 オプションを付けてビルドしたプログラムを実行

※一部Cygwin6.3,64bit + GCC 4.8.2 で実行したものもあり。

## サンプルプログラム記載の方針について

本書に記載するサンプルプログラムは、関数リファレンス的な性質のものではなく、なるべく各技術の性質を把握するための具体的な用例として示している。

そのため、やや長めのコードとなっているが、各技術のポイントとなる箇所には着色して分かり易いようにしている。また、各技術の機能を網羅的に紹介するものではないが、少しでも広範囲にポイントを抑えたものになるようにしている。

各サンプルプログラムは、同系統技術の比較のために、大部分が同じコードになっているものも多い。しかし、ドキュメントが全体的に長いこともあり、差分での記載とせず、それぞれ完全に動作する状態のサンプルを示す方針としている。これにより、部分的にドキュメントを参照したい時に、広範囲に読み返さなくても良いものとしている。

# スレッドの仕組み

まず、「スレッドとはなにか？」を解説する。

主に、スレッドの挙動とスレッドがなにによって管理されるのかを説明する。

## マルチタスク

「スレッド」とは、「マルチタスク」の実装手段の一つである。

「マルチタスク」には大きく分けて二つの方式があり、どの方式が使われるかはOSに依存する。

### ノンプリエンプティブなマルチタスク

OSが関与する部分が少なく、タスク自身が自発的にCPUを解放しないと他のタスクに制御が移らない方式。



Windows95より前のWindows3.1はこの方式。例えば「while(1){ }」と書いたプログラムを実行すると、同時に起動している「メモ帳」や「電卓」などのアプリケーションも動作できずに止まってしまう。

この方式はOSの負担が非常に軽いため、現在でも部分的に利用されている。「ファイバースレッド」の節で後述する。

### プリエンプティブなマルチタスク

OSの制御により、短い時間で区切って複数のタスクを順に実行する方式。

非常に短い間隔で素早く切り替え続けることで、複数のタスクが同時に実行されているように見せる。



今時のゲーム機向けのプログラミングも、通常のスレッドはこの方式で扱われるものと思って良い。

なお、少し古いゲーム機ではOSそのものが無いため、「ノンプリエンプティブ」な方式で扱うか、スレッドを使うこと自体がほとんどなかった。

### タイムスライス

タスクの実行を時間で区切ることを指して「タイムスライス」と呼ぶ。

### 並行と並列

ここまでの説明は、「複数のタスク」を「一つの演算装置」が「平行」（Concurrent）に実行する手法のことである。一人の人が複数の仕事を受け持っている状態である。

マルチプロセッサ、マルチコア、SPU、GPGPUなど、「複数の演算装置」を活用した処理では、「並列」（Parallel）のタスクが実現できる。複数の人が分担して仕事を行う状態である。



なお、厳密な用語の定義としては、「平行」は「並列」を包含する。「複数のタスク」を「複数の演算装置」が実行することもまた「平行」である。

あまり神経質に用語を使い分ける必要はないが、本書においては上記の通りの使い分けをしている。

## OSとカーネル

「マルチタスク」の制御は「OS」が行う。

「OS」とは「Operating System」の略語である。前述のタスクの制御や、メモリ管理、入出力装置（デバイス）の制御、ファイル操作などを行う基本プログラムを意味する。



「ゲーム」を含め、ユーザーが作成するアプリケーションプログラムは、このOSを通してデバイスにアクセスし、OSの制御のもとでCPUの割り当てやメモリの確保が行われる。

OSは多数のプログラムで構成されている。各種デバイスを制御する個々のプログラムを「デバイスドライバ」と呼び、タスク制御やメモリ管理を行うOSの中核プログラムを「カーネル」と呼ぶ。

スレッドの制御を行うのは「カーネル」である。カーネルには、大きく分けると2種類の方式がある。

### モノリシックカーネル

Wikipediaの説明を引用すれば、「『入出力機能やネットワーク機能、デバイスのサポートなどOSの一般的な機能』をカーネルと同一のメモリ空間に実装・実行する手法を言う」との事。



単純には、デバイスドライバがカーネルに一体化しているような状態。そのため、多くの機能を取り込んだカーネルは肥大化する。また、デバイスドライバに深刻な問題があると、OS全体が停止するような事態も招く。

もう一つの「マイクロカーネル」と比べると古い方式で、「時代遅れ」とされることもあったものの、比較的仕組みが単純なこともあり、オーバーヘッドが少ないため、今なおLinuxを含む多くのUnix系OSなどで採用されている。Linuxなどでは、デバイスドライバをOS起動後に読み込む「ローダブルモジュール」にも対応している。

語源は「一枚岩」を意味する「モノリス」から。（映画「2001年宇宙の旅」に登場する物体）

### マイクロカーネル

Wikipediaの説明を引用すれば、「OSが担う各種機能のうち、必要最小限のみをカーネル空間に残し、残りをユーザーレベルに移すことで全体の設計が簡素化でき、結果的に性能も向上できるという考え方。カーネル本体が小規模な機能に限定されるので『マイクロカーネル』と呼ばれるが、必ずしも小さなOSを構成するとは限らない。」との事。



旧来のモノリシックカーネルの大規模な改善を意図して設計されたOS。

カーネル自体は必要最小限にとどめ、デバイスドライバなどの他の「準カーネル」機能を切り離して構成している。そのため、OSの機能拡張やOS全体を止めずに一部の機能をアップデートすることなどが可能。反面、機能間の相互通信が多く、オーバーヘッドが大きく、メモリ使用量も大きい。

マイクロカーネルはWindowsNT以降のMicrosoft系OSで採用されているが、グラフィックスドライバーなどのオーバーヘッドが大きかったことから、カーネルから直接アクセスできるデバイスも設けられており、純粋なマイクロカーネルではない。

### ハイブリッドカーネル

「モノリシックカーネル」と「マイクロカーネル」のハイブリッド。

前述の通り、Microsoft の WindowsNT以降のOSはこの類に入る。

また、Apple のOS XはFreeBSDのモノリシックカーネルをベースにしたハイブリッドカーネル。

### カーネルモードとユーザーモード

OSの制御により、プログラムの実行は「カーネルモード」と「ユーザーモード」という二つの「動作モード」に切り分けて扱われる。

アプリケーションのパフォーマンスをより最適なものにするためには、これらの動作モードを（多少は）意識したほうがよい。

#### ユーザーモード

「ユーザーモード」はアプリケーション側の処理モードのことである。

他のアプリケーションのメモリに干渉することや、ファイルシステム、デバイスを破損させるといったことがないように保護された動作モードである。

ユーザーモードで動作するプログラムは、処理に問題があってもOS全体がクラッシュするようなことにはならない。

#### カーネルモード

「カーネルモード」はOS側の処理である。

ユーザーモード側で行われるメモリ確保やファイルアクセス、デバイス操作といった処理は、カーネルモード側の処理が呼び出されて実行される。

直接メモリやファイル、デバイスを操作する特権モードであり、この処理に問題があると、OS全体がクラッシュすることがある。

#### デバイスドライバ

「デバイスドライバ」の動作モードは、通常カーネルモードである。

しかし、マイクロカーネルの場合はユーザーモードで動作し、デバイスドライバからOSが保護される。

Windowsでは、パフォーマンスのために純粋なマイクロカーネルをやめて、グラフィックカードなどのデバイスドライバはカーネルモードで動作するようにしているが、一部のデバイスドライバはユーザーモードで動作する。

#### アプリケーション

OS上で実行させるプログラムを「アプリケーション」と呼ぶ。

アプリケーションはユーザーモードで実行される。

#### システムコールとコンテキストスイッチ

ユーザーモードからカーネルモードのプログラムを呼び出す際は、「システムコール」を使用する。

システムコールとは、OS側の処理の呼び出しのことである。システムコールが呼び出されると、「コンテキストスイッチ」という処理が発生し、動作モードをカーネルモードに切り替える。

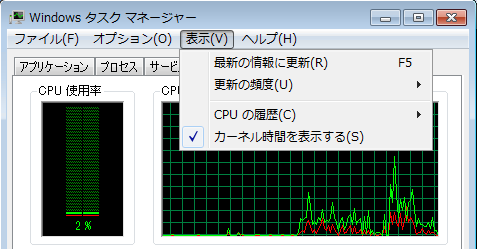
コンテキストスイッチには多少の（微少な）処理負荷かかるので、パフォーマンスのためにも、システムコールは多用しすぎないほうがよい。

システムコールをアプリケーション側のプログラムで直接使用することはあまりないが、APIの中でシステムコールを使用しているものは多い。例えば、malloc() 関数や fopen関数は内部でシステムコールを使用している。システムコールを使用しているかどうかは、SDKのマニュアルに明記されていることもある。

なお、「コンテキストスイッチ」とはプログラムの切り替え処理のことであり、タスク（スレッド）の切り替えの際も発生する。この説明については改めて後述する。

#### カーネル時間

WindowsではCPU使用率を表示する際に「カーネル時間を表示する」というオプションを指定できる。



この「カーネル時間」が「カーネルモード」の処理負荷を意味する。

カーネル時間が高まるのは、デバイスへのアクセスやグラフィックス、ネットワーク通信などが込み合った時である。

### BIOS

コンピュータ起動時に最初に実行されるプログラムは、OSではなく、ROMに組み込まれている「BIOS」（バイオス＝Basic Input/Output System）である。

BIOSがHDDやCDなどからOSをロードして実行している。

その昔は、BIOSからOSを介さず、直接FDやCD、カートリッジのゲームを起動していた。

## プロセスとスレッド

「マルチタスク」の実行単位は「スレッド」であり、スレッドは「プロセス」によって管理される。プロセスの管理とタスクの制御は「OS」が行う。



### プロセス

OSに対して「メモ帳」や「Excel」などの「プログラム」（.exeファイルや.elfファイルなど）の実行が呼び出されると、OSはまず「プロセス」を生成し、プロセスに対してプログラムをロードし、実行のためのメモリを割り当てる。

基本的に各プロセスは独立しており、互いに干渉することなく動作する。

同じプログラムを複数実行した場合（例えばメモ帳を二つも三つも起ち上げた状態）、それぞれ別のプロセスとして動作し、互いに干渉しない。

なお、ゲーム機の場合、「プロセスの概念そのものがない」か、「実行できるプロセスは一つだけ」というのが普通。

#### 子プロセス

プロセスは「子プロセス」というプロセスを生成できる。

両者はメモリ空間を共有できないが、Unix系OSでは多数のメモリ状態やオブジェクトがコピーされ、オープンしているファイルディスクリプタなどの共用ができ、Windows系OSではパイプによる入出力ハンドルの共有などができる。

親プロセスは子プロセスの終了を待ったり、子プロセスを終了させたりといった制御が可能。

### スレッド

プロセスが起動する際、「メインスレッド」を生成し、「スタック領域」を確保して、プログラムの実行を開始する。

一つのプロセス内には、メインスレッドのほか、複数のスレッドを動作させることができる。

スレッドの生成は、関数を呼び出す際に行う。スレッド生成用APIに対して関数とスタック領域を指定すると、OSがその関数をスレッドとして実行し、マルチタスクのスケジューリングの対象に加えて制御する。

### メモリ空間の保護

プロセス内の全スレッドはメモリ空間を共有できる。

しかし、別プロセスのメモリ空間にはアクセスできない。プロセスのメモリ空間は、OSが制御する「仮想メモリ空間（仮想アドレス）」によって保護されるため、メモリアドレスを直接指定しても、他のプロセスに干渉することはない。

なお、プロセス間で情報を共有する手段は幾つも用意されている。どのような手段があるかは後述する。

逆にスレッド内だけで（スコープが）保護されたメモリ領域を扱うこともできる。これは「TLS＝Thread Local Strage」（スレッドローカルストレージ）と呼ばれる仕組みである。詳しくは後述する。

### プロセスの終了時のリソースの解放

メインスレッドが終了すると、プロセスが終了する。

この時、「未解放のメモリ」（メモリリーク）、「クローズしていないファイルディスクリプタ（FILE\*など）」、「終了していないスレッド」、「クローズしていない同期オブジェクトなどの各種ハンドル」など全ての未解放リソースに対して、終了処理が行われる。

ただし、きちんと終了できずに「ゾンビプロセス」という形で残ってしまうケースもある。何より、メモリリーク等を見過ごしていると、長時間のアプリケーション実行でメモリ不足などの深刻な障害をもたらす。終了処理に頼らず、逐一きちんとリソースの解放を行うのがプログラミングの作法である。

【補足】forkによる子プロセスを生成した時、親プロセスが子プロセスの終了をwaitしないと子プロセスがゾンビ化する。

### サービス／デーモン

Windowsの「サービス」、Unix系OSの「デーモン」は、バックグラウンドで常時稼働する「プロセス」のことである。

GUIを持った「フロントエンド（プロセス）」に対して、姿の見えない「バックエンド（プロセス）」といった呼び方もする。

サービス／デーモンは、ログインしなくてもOS起動と同時に実行するようにOSが管理する。

Webサーバーなどのサーバー系のシステムに多く用いられる形態。何らかのリクエストが来るまで待ち続けるような処理が多い。

なお、「デーモン」は「demon」（悪魔）ではなく「daemon」（守護神）。

# スレッドとハードウェア

マルチスレッドプログラミングでは、対象となるハードウェアを意識しなければならない。

使用するハードウェア／OSに応じた判断要件がある。

例えば、「同期にスピンロックを使うべきか？（別コアにスレッドを分散させる必要あり）」、「SPUやGPU（GPGPU）のような副プロセッサを利用するべきか？（プロセッサ専用のプログラムが必要）」、「多数の副プロセッサがあるなら、一部のプロセッサは特定の処理に専門化して、専用プログラムを読み込む手間をなくようにすべきか？（SPUでよく使われる手法）」 など。

また、スレッドの扱い方や同期の方法など、プログラミング手法がOSによって異なる。

## CPU

「Central Processing Unit」（中央処理装置）のこと。コンピュータで最もメインとなる演算装置を指して「CPU」と呼ぶ。

微妙に意味の違う同義語があるので以下に幾つか列挙する。

MPU 「Micro-Processing Unit」。中央処理装置を一個の半導体チップに集積したもの。元来の「CPU」は複数の半導体チップの連携で演算処理を行う集合体のことを指していたが、もはやMPUとCPUは同義となっている。

PPE 「Power PC Processor Element」。PS3に採用された「Cell」プロセッサの中のメインプロセッサコア。

APU AMDが開発した、CPUとGPUを合成したプロセッサ。

### CISCとRISC

CPUは、その命令セットアーキテクチャによって大きく二つに分類される。

CISC（Complex Instruction Set Computer: シスク）と、RISC（Reduced Instruction Set Computer：リスク）である。

CPUの発展に伴って、その命令セットが複雑化してきたことを背景に、単純な命令を指向してRISCが考案された。複雑なことを１命令でこなせるように命令セットがふくれあがったCISCに対して、命令セットを減らして回路を単純化することで高速化を図ったのがRISC。

なお、CISCという言葉はRISCの誕生に伴って、対義語的に用いられるようになったものである。

RISCの登場はCISCの淘汰には至らず、現状はどちらも主流である。Intel系はCISCで、ARM系やPowerPC系はRISCである。どちらもゲーム機に採用されている。

## シングルプロセッサ／シングルコア

まず、「プロセッサ」と「プロセス」は言葉が似ているが全く別物なので注意。

「プロセス」は先に説明した通り論理的なプログラムのインスタンスであるが、「プロセッサ」は物理的な演算装置のことを意味する。

「シングルプロセッサ」は「コンピュータの中に一個のプロセッサ（物理的な演算装置パッケージ）が搭載された状態」のことで、「シングルコア」は「一個のプロセッサの中に一個のコア（演算装置の演算部）が搭載された状態」のことである。

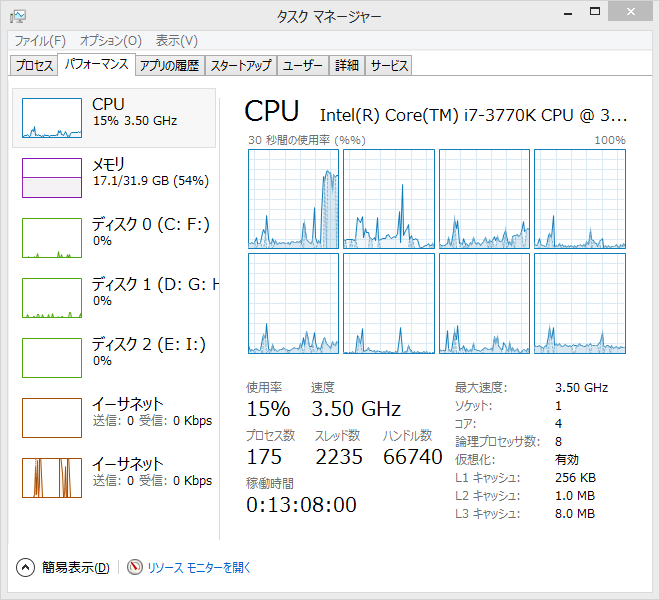
## マルチプロセッサ／マルチコア／論理プロセッサ



この図は、「一つのCPU」に「二つのコア」が搭載されている「マルチコア」プロセッサの例である。IntelのCPUでは、さらに「HT = Hyper-Threading technology」（ハイパースレッディング・テクノロジー）によって、一つのコアで「二つのスレッド」の平行処理に対応する。これをOSから見ると、四つのCPUがあるように見えるため、「四つの論理プロセッサ」と呼ぶ。

Windows8から見た、4コア8論理プロセッサのCPUの例を示す。L1～L3キャッシュメモリのサイズも表示されている。

Windows8のタスクマネージャの例：



同様の「プロセッサ」が複数ある状態が「マルチプロセッサ」である。上記の「タスクマネージャ」では「ソケット」と表記されている。

GPUもプロセッサの一つではあるが、メインのプロセッサではないので、通常マルチプロセッサに数えない。

マルチプロセッサは、複数のCPUソケットを持った基板（マザーボード）に、全く同じスペックのCPUを複数装着して動作させるのが通常である。サーバー系によく用いられる。

かつては複数の演算装置で並列処理をするにはマルチプロセッサにするしかなかったが、マルチコアの登場により、安価で低消費電力な並列演算装置が一般化し、それに伴ってマルチスレッド処理も並列化の意識が高まっているのが現状である。

余談になるが、一つのCPUに同種のコアのみを搭載しているものを「ホモジニアスマルチコア」と呼び、異種のコアを搭載しているモノを「ヘテロジニアスマルチコア」と呼ぶ。後者の代表例には、PS3に採用されている「Cell」（PPEコア×1＋SPEコア×8）、AMD社の「APU」（CPU + GPU）がある。

## ハイパースレッディング・テクノロジー

一つの物理的な演算装置（コア）上で二つの処理をスケジューリングすることで二つの「論理プロセッサ」（論理コア）に見せる技術である。

「整数処理」と「浮動小数点処理」のように、命令パイプラインの異なる処理の同時実行に特に有効とのこと。とはいえ、複数の処理系のコードが同時にキャッシュメモリに乗っていないと並列処理できないので、必ずしも最適化できるわけではない。

もともとCPUの稼働率を向上させるための技術であるため、単純に２倍の性能が得られるわけではない。

なお、「ハイパースレッディング」の呼称はインテルの商標であるが、同技術は他社の演算装置にも用いられており、サーバー用途に4, 8論理コアの製品もある。

## コプロセッサ

メインのCPU以外の演算装置を「コプロセッサ」（co-prosessor, 副処理装置）と呼ぶ。

コプロセッサにはFPUやGPUなどがある。

## 浮動小数点演算装置（FPU）／SIMD

本来のCPUは整数しか演算できないため、浮動小数点の演算は独自に実装する必要があった。詳細は省略するが、浮動小数点データの指数部に基づいて仮数部をビットシフトしてから演算するといった手間がかかる。

この手間のかかる演算を高速化するために、専用の演算装置が用いられる。それが「浮動小数点演算装置」（FPU = Floating Point Unit）である。

FPUの形態は、CPUと一体化しているものもあれば、独立した演算装置のものもある。例えば、PS2に搭載されていたVU（Vector Unit: ベクトル演算ユニット）は独立したFPUの一種。

現在のCPUでは、浮動小数点演算演算装置はCPUに統合され、CPUの命令セットが浮動小数点演算用に拡張されている。

さらには、複数の浮動小数点演算をまとめて行うための「SIMD」命令が拡張搭載されている。主なSIMD演算命令セットには「MMX」（MultiMedia eXtensions）、「SSE」（Streaming Simd Extensions）などがある。また、GPUの演算装置は基本的にSIMDである。

### SIMD／MIMD／SISD／MISD

「SIMD」（Single Instructoin, Multi Data stream：シムド）は、一回の命令で複数の演算をまとめて行うものである。このSIMDの活用は、ゲームプログラミングではとても意識する必要がある。

128bit（レジスタ）のSIMD演算の場合、4つの32bit値をまとめて処理できる。256bitなら8つである。レジスタ長が固定されるので、演算する値の数も固定される。3つの値を演算したい時も、一つはダミーとして、4つの値を演算することになる。

SIMD演算は、ベクトルや行列の演算に多用される。

SIMDは複数の値（＝値セット）に対して一つの命令をまとめて実行する。この値セットに対して、同時に二つの命令を実行するのが「MIMD」（Multiple Instruction, Multiple Data streams：ミムド）である。MIMDでは、一つの値セットを複数のプロセッサやコアに振り分けて、それぞれで別個の演算を行う。

他、単独の値に対する演算は、「SISD」（Single Instruction, Single Data Stream）「MISD」（Multiple Instruction, Single Data Stream）と呼ばれる。

## GPU

「GPU」（Graphics Processing Unit：グラフィックス演算装置）は、グラフィックス描画専用の演算装置である。多数のコアやシェーダーユニットなどで構成される。

グラフィックス描画以外の演算にもGPUを応用する技術が「GPGPU」（General-Purpose computing on GPU）。

## メインメモリ

プログラムの本体と各種データはメインメモリに置かれる。プログラムも突き詰めればデータの一種である。

CPUはプログラムとデータを逐次メインメモリから読み出して実行する。

## RAMとROM

通常「メモリ」と言えば「RAM」（Randam Access Memory：ラム）のことである。読み書き可能なメモリのこと。

それに対して、「ROM」（Read Only Memory：ロム）というのは、読み取り専用で書き込むことができないメモリのことである。BIOSプログラムの保存領域や、物理的に書き込みできないCD-ROM／ゲームカードなどに使われる言葉。

なお、ROMの中には「フラッシュROM」などの、書き換え可能なROMもある。

ゲームでは、データをROMからRAMに読み込んで処理するのが普通。

## キャッシュメモリ

CPUが読み出したメインメモリのデータは、CPU上の「キャッシュメモリ」に蓄えられる。

L1キャッシュ（Layer-1キャッシュ＝一次キャッシュ）→L2キャッシュ→L3キャッシュ→メインメモリの順に、アクセスが高速（L1キャッシュが一番速い）。L3キャッシュを搭載しているのは比較的最近のCPU。

CPUは、高速に動作するために、必要が生じるまでメインメモリに極力アクセスせず、キャッシュメモリを優先的に扱う。

キャッシュによる効率化は、C言語などの高級言語でプログラミングする際も多少は意識したほうが良い。

例えば、一般にインライン展開はプログラムサイズと引き替えに高速化されるものであるが、ループ処理の中で同じ関数を何度も呼んでいるような処理の場合、全てインライン展開されると長い処理となり、キャッシュに全部乗らなくなってしまうと、むしろ関数呼び出しの方がキャッシュの読み替えが起こらず、高速に動作する、といったことがある。

実際どの程度のサイズが妥当かは見極めが難しいが、何にしても、繰り返し実行されるような処理は短くまとめておくと、効率的になる。

ほかには、遅延評価が可能な演算に対しては（求めた計算結果をすぐに使って次の演算を行う必要がない場合）、命令と値をプールして、後でまとめて演算するような効率化手法もある。

長い計算がある場合、なるべく一本の計算式にまとめているほうが最適化されやすい（可読性は落ちるが）。

## レジスタ

CPUは演算に使う値を「レジスタ」という領域に保存して扱う。

レジスタは変数の一種のようなものであるが、C言語などのように自由な変数を使えるわけではなく、CPUによって扱えるレジスタの種類と数が固定されている。

C言語などが扱う多数の変数はメモリ上で扱われ、都度レジスタに読み込んで演算して書き戻すことで処理する。（メモリからレジスタへの読み込みを「ロード」と呼び、レジスタからメモリへの書き戻しを「ストア」と呼ぶ）

メモリアクセス（キャッシュメモリへのアクセスを含む）は、CPUの処理としては比較的遅いので、一度レジスタに取り込んだ変数は極力メモリに書き戻さないで処理する。

コンパイラはそのような処理になるように最適化を行う。特にライフサイクルの短い変数の場合、メモリに置くことすらせず、レジスタに直接値を与えて完結するようなプログラムにする。

変数の少ないプログラムは、このような事情から高速になる。

このような処理の最適化がコンパイラによって行われることは、マルチスレッドプログラミングでは非常に意識する必要がある。

レジスタだけで処理が完結してメモリに書き戻されない変数は、他のスレッドから見えず、値の変化をキャッチすることができないといった問題が生じるためである。

この問題については詳しく後述する。

## GPUメモリ／ユニファイドメモリ

GPUはグラフィック描画のための専用のメモリを持つ。VRM（Video RAM）とも呼ばれる。シェーダープログラムもGPUメモリ上に置かれる。

GPUメモリはメインメモリから独立したメモリであるため、CPUが直接アクセスすることができない。そのため、「DMAC」などの他のメモリコントローラーを使用してアクセスする。

GPUが専用メモリを持たず、CPUとメインメモリを共有するのが「ユニファイドメモリ」である。

AMDは、「APU」（Accelerated Processing Unit／AMD Fusionプロセッサ）でCPUとGPUを統合し、さらに、「hUMA」（heterogeneous Unified Memory Access：ヘテロジニアス・ユニファイドメモリアクセス）という名称のユニファイドメモリ技術により、メモリ空間の統合も図っている。

## DMAC

「DMAC」（Direct Memory Access Controller）は、メモリを転送する装置である。単に「DMA」とも呼ぶ。

CPUとは独立して動作し、メインメモリとGPUメモリの間のデータ転送や、メインメモリ内での大きなデータの転送などに用いられる。

## Cell

「Cell」（Cell Broadband Engine：セル）を構成する中で「制御」を担当する汎用プロセッサコアが「PPE」（Power PC Processor Element）。1個のPPEに対して8個の「SPE」（Synergistic Processor Element）が演算を担当する。計9個のプロセッサコアで一つのプロセッサを構成している。

「SPU」（Synertistic Prosessor Unit）は、「SPE」の中の演算装置コア本体のこと。SPUは自身が持つ256KBのローカルメモリ（LS = Local Store）にしかアクセスできず、かつ、PPU向けのプログラムとは別のSPU専用プログラムしか実行できない。

そのため、一部の関数をスレッド化するような用法は使えず、専用プログラムと演算対象のデータをメインメモリからSPUに転送して処理し、演算結果をまたメインメモリに転送する。

このような構造のため、細かい演算を頻繁に実行させるような用法には向いておらず、ある程度まとまったデータを一括処理させるような使い方をする。

なお、メインメモリとLSとの通信には、SPEのDMAユニットが用いられる。

## サーバー

コンピュータの物理的な垣根を越えて、ネットワーク通信を使って別のコンピュータに演算を行わせる手法がある。

この演算の要求を受け付けるコンピュータが「サーバー」である。

「グリッド・コンピューティング」では、CPUやGPUのみならず、サーバーも含めて「演算リソース」として抽象化し、処理を分散して並列処理を行う。

# マルチスレッドの意義

マルチスレッドをゲームプログラミングに適用する意義を考察する。

## マルチプロセス

まず、複数の処理を同時に実行するマルチタスクの意義について考えると、分かり易い所で「マルチプロセス」がある。（「マルチプロセッサ」とは別物なので注意）

マルチプロセスは、その名の通り複数のプロセスを同時に実行することである。一台のコンピュータ上で複数のアプリケーションを実行し、一台のコンピュータを複数のユーザーが同時に操作することを可能とする。

Widnowsユーザーにはイメージしにくいかもしれないが、その昔のコンピュータの使い方は、一つのホスト（サーバー）に、多数の端末（クライアント／ユーザー）が接続して同時に操作をしていた。

高価なコンピュータを何台も購入せずに、複数のユーザー、複数のアプリケーションを動作させることがマルチプロセスの意義である。

また、シングルプロセッサ、シングルコアの古いコンピュータでもマルチプロセスが実現できていたのは、一つのアプリケーション／ユーザーが、ミリ秒単位でコンピュータを占有しているわけではないため、それぞれの空き時間で十分に共有が行えたためである。

ユーザーがキーボードを連打する時のごく短い間隔であっても、コンピュータにとってはCPUを明け渡すのに十分な時間である。

## マルチスレッドの意義①：同時実行

「マルチスレッド」とは、一つのプロセスの中で複数の処理を同時に行うことである。

利用例としては、ユーザーがアプリケーションを操作し続ける裏で長く時間のかかる計算を実行するといったものがある。

ここで重視しているのは、操作と計算を「同時に」実行することである。

ユーザーの操作を禁止して計算に専念した方が処理は早いが、わざわざ同時に実行するのは、「（多少レスポンスが悪くなっても）ユーザーが操作できない時間を極力作らない」という意義があるからである。また、「計算を途中キャンセルできる」といった処理中の操作を受け付けたい場合もスレッドを活用したほうがよい。

## マルチスレッドの意義②：同時接続

「マルチスレッド」には「マルチプロセス」と同じ意義もある。

サーバー系の処理に多く、一つのアプリケーションが多数のPC（クライアント）に同時に接続して並行して処理するものである。Webサーバーやチャットサーバーなどがイメージし易い。

一つのクライアントに対して一つのスレッドで処理し、多数のクライアントが接続されればその分スレッドが増えていく方式である。（マルチプロセスで処理するサーバーもある）

これも一つ一つの接続が100%コンピュータを占有するわけではないので、シングルプロセッサ、シングルコアであっても共有を実現できる。

## マルチスレッドの意義③：分散・高速化

ここまで説明してきたとおり、そもそものスレッドの意義は、並行化の実現であって、複数の処理を同時に実行してもパフォーマンスが向上するようなものではない。

しかし、「メニーコア時代」と呼ばれ、安価なマルチコアプロセッサーが一般化した現在は、スレッドを高速化に利用する意義が生じている。

例えば、大量データのソートや集計は、データを分割して複数の演算装置で同時に演算したほうが高速に処理できる。なお、シングルコア環境ではスレッドを増やしたところで早くはならず、むしろスレッドの管理コスト分遅くなる。

## ゲームの並行処理

ゲームプログラミングはもともと並行処理の塊である。

多数のキャラを同時に動かし、それと同時にカメラを動かし、同時にメニューを動かし、同時にファイルを読み込んでいる。しかし、これらの処理をそれぞれスレッド化するようなことはあまりない。

ゲームプログラミングは、並行処理を自前で行うのが通常である。

ほとんどの並行処理要素は1フレームごとの処理単位を意識し、一コマ分ずつ一つずつ処理を進める。実際には直列処理である。

むやみにスレッド化すると、膨大な量のスレッドが出来上がり、メモリや処理に無駄が生じる（各スレッドはそれぞれスタック領域を持ち、スレッドを切り替えることにも処理コストはかかる）。スレッド間の処理の動機も難しくなる。

## ゲームでのスレッド活用

以上を踏まえ、ゲームでスレッドを活用する意義は、主に下記のような用途と考察する。

描画スレッド（GPU処理と並列化）

処理落ちの影響を受けてはいけない常時稼働処理（サウンドなど）

並列実行が可能な演算（アニメーション、物理演算、圧縮展開など）

演算結果が数フレーム後になっても良いもの（AIなど）

普段は待機状態で、要求に応じてウェイクアップするバックエンド処理（通信など）

メインループよりも頻繁に状態を監視・確認したいもの（入力デバイス制御、ファイル読み込みなど）

* 注：ファイル読み込みに関しては通常非同期読み込みをOSレベルでサポートしているので、必ずしもスレッド化の必要はない。スレッドで管理したほうが良いのは、読み込み要求をキューイングしているようなケース。ファイルディスクリプタの制限から同時実行できない読み込み要求をバックエンドで逐次処理するような場合にスレッド化。

マルチスレッドは、スレッド間の同期や共有リソース（メモリなど）の扱いにとても神経質にならなければならない。同期やリソースアクセスには厳格なルールを設けて扱う。

# プログラムの動作原理

基本的なプログラムの動作原理を理解していないままスレッドを扱うのは危険である。

特にスタック領域がどのように使われるかは把握しておく必要がある。

## メモリ構造

まず、プログラムのメモリ構造は下記の通りとなっている。セクション名などは、GCCでビルドした場合の構成だが、Windowsでも大きな違いはない。



実際にはもっと細かいセクションに分かれていたり、読み取り専用領域やヒープ領域が不連続な配置になっていたりする。

また、図では省略しているが、「.data」セクションのデータ（初期値）は最初に読み取り専用領域に配置され、読み書き専用領域を確保した後にコピーされる。

「.bss」セクションはゼロクリアされることが保証されている。なお、「bss」とは「Block Started by Symbol」（シンボル名でアドレスが示されているメモリブロック）のこと。

以下、実際のプログラムをサンプルに、各要素のセクションとの対応と、メモリ情報の確認結果を示す。

各種変数を定義したプログラムのサンプル：

|  |
| --- |
| #include <stdio.h>  int global\_without\_value; //[.bss]初期値なし大域変数  int global\_with\_value = 1; //[.data]初期値付き大域変数  const int global\_const\_value = 2; //[.rodata]大域const変数（定数）  //[.text]main() 関数  int main(const int argc, const char\* argv[])  {  printf("main():begin\n");    static int static\_without\_value; //[.bss]初期値なし局所静的変数  static int static\_with\_value = 3; //[.data]初期値付き局所静的変数  static const int static\_const\_value = 4; //[.rodata]局所静的const変数（定数）  int auto\_without\_value; //[スタック]初期値なしローカル変数  int auto\_with\_value = 5; //[スタック]初期値付きローカル変数　※初期値はプログラムコード化  const int auto\_const\_value = 6; //[スタック]ローカルconst変数（定数）　※初期値はプログラムコード化  const char\* auto\_str = "STRING"; //[スタック]初期値付きローカル変数　※[.rodata]文字列リテラル    global\_without\_value = 101;  global\_with\_value = 102;  static\_without\_value = 103;  static\_with\_value = 104;  auto\_without\_value = 105;  auto\_with\_value = 106;  printf("%d,%d,%d,%d,%d,%d,%d,%d,%d,\"%s\"\n",  global\_without\_value,  global\_with\_value,  global\_const\_value,  static\_without\_value,  global\_const\_value,  static\_without\_value,  static\_with\_value,  static\_const\_value,  auto\_without\_value,  auto\_with\_value,  auto\_const\_value,  auto\_str  );    printf("main():end\n");    return 0;  }  //[.text]クラス  class auto\_run\_test\_class  {  public:  auto\_run\_test\_class()  {  printf("auto\_run\_test\_class::constructor()\n");  }  ~auto\_run\_test\_class()  {  printf("auto\_run\_test\_class::destructor()\n");  }  };  //大域変数に直接インスタンスを生成したクラス  //[.init.text]コンストラクタ呼び出しは main() 関数より先に実行  //[.exit.text]デストラクタ呼び出しは main() 関数よの後に実行  auto\_run\_test\_class auto\_run\_test\_obj;  //[.init.text]初期化関数テスト１：main() 関数より先に実行  //※このアトリビュートは GCC の方言  \_\_attribute\_\_((constructor))  void auto\_run\_test\_func\_constructor\_1()  {  printf("auto\_run\_test\_fun\_constructor\_1()\n");  }  //[.init.text]初期化関数テスト２：main() 関数より先に実行  //※このアトリビュートは GCC の方言  \_\_attribute\_\_((constructor))  void auto\_run\_test\_func\_constructor\_2()  {  printf("auto\_run\_test\_fun\_constructor\_2()\n");  }  //[.exit.text]終了関数テスト１：main() 関数の後に実行  //※このアトリビュートは GCC の方言  \_\_attribute\_\_((destructor))  void auto\_run\_test\_func\_destructor\_1()  {  printf("auto\_run\_test\_fun\_destructor\_1()\n");  }  //[.exit.text]終了関数テスト２：main() 関数の後に実行  //※このアトリビュートは GCC の方言  \_\_attribute\_\_((destructor))  void auto\_run\_test\_func\_destructor\_2()  {  printf("auto\_run\_test\_fun\_destructor\_2()\n");  } |

↓（実行結果）

|  |
| --- |
| auto\_run\_test\_class::constructor()  auto\_run\_test\_fun\_constructor\_2()  auto\_run\_test\_fun\_constructor\_1()  main():begin  101,102,2,103,104,4,105,106,6,"STRING"  main():end  auto\_run\_test\_class::destructor()  auto\_run\_test\_fun\_destructor\_1()  auto\_run\_test\_fun\_destructor\_2() |

sizeコマンドによるサイズ確認例： ※GCC系のみ

|  |
| --- |
| $ size a.out  text data bss dec hex filename  2562 612 32 3206 c86 a.out ←\*.text + .rodataサイズ, .dataサイズ, .bss サイズ, 合計サイズ |

objdumpコマンドによるサイズ確認例： ※GCC系のみ

|  |
| --- |
| $ objdump -h a.out  a.out: file format elf64-x86-64  Sections:  Idx Name Size VMA LMA File off Algn  0 .interp 0000001c 0000000000400200 0000000000400200 00000200 2\*\*0  CONTENTS, ALLOC, LOAD, READONLY, DATA  1 .note.ABI-tag 00000020 000000000040021c 000000000040021c 0000021c 2\*\*2  CONTENTS, ALLOC, LOAD, READONLY, DATA  2 .note.gnu.build-id 00000024 000000000040023c 000000000040023c 0000023c 2\*\*2  CONTENTS, ALLOC, LOAD, READONLY, DATA  3 .gnu.hash 00000024 0000000000400260 0000000000400260 00000260 2\*\*3  CONTENTS, ALLOC, LOAD, READONLY, DATA  4 .dynsym 000000c0 0000000000400288 0000000000400288 00000288 2\*\*3  CONTENTS, ALLOC, LOAD, READONLY, DATA  5 .dynstr 000000ac 0000000000400348 0000000000400348 00000348 2\*\*0  CONTENTS, ALLOC, LOAD, READONLY, DATA  6 .gnu.version 00000010 00000000004003f4 00000000004003f4 000003f4 2\*\*1  CONTENTS, ALLOC, LOAD, READONLY, DATA  7 .gnu.version\_r 00000040 0000000000400408 0000000000400408 00000408 2\*\*3  CONTENTS, ALLOC, LOAD, READONLY, DATA  8 .rela.dyn 00000018 0000000000400448 0000000000400448 00000448 2\*\*3  CONTENTS, ALLOC, LOAD, READONLY, DATA  9 .rela.plt 00000078 0000000000400460 0000000000400460 00000460 2\*\*3  CONTENTS, ALLOC, LOAD, READONLY, DATA  10 .init 00000018 00000000004004d8 00000000004004d8 000004d8 2\*\*2  CONTENTS, ALLOC, LOAD, READONLY, CODE  11 .plt 00000060 00000000004004f0 00000000004004f0 000004f0 2\*\*2  CONTENTS, ALLOC, LOAD, READONLY, CODE  12 .text 00000358 0000000000400550 0000000000400550 00000550 2\*\*4  CONTENTS, ALLOC, LOAD, READONLY, CODE  13 .fini 0000000e 00000000004008a8 00000000004008a8 000008a8 2\*\*2  CONTENTS, ALLOC, LOAD, READONLY, CODE  14 .rodata 0000014c 00000000004008b8 00000000004008b8 000008b8 2\*\*3  CONTENTS, ALLOC, LOAD, READONLY, DATA  15 .eh\_frame\_hdr 00000064 0000000000400a04 0000000000400a04 00000a04 2\*\*2  CONTENTS, ALLOC, LOAD, READONLY, DATA  16 .eh\_frame 000001a4 0000000000400a68 0000000000400a68 00000a68 2\*\*3  CONTENTS, ALLOC, LOAD, READONLY, DATA  17 .ctors 00000028 0000000000600c10 0000000000600c10 00000c10 2\*\*3  CONTENTS, ALLOC, LOAD, DATA  18 .dtors 00000020 0000000000600c38 0000000000600c38 00000c38 2\*\*3  CONTENTS, ALLOC, LOAD, DATA  19 .jcr 00000008 0000000000600c58 0000000000600c58 00000c58 2\*\*3  CONTENTS, ALLOC, LOAD, DATA  20 .dynamic 000001c0 0000000000600c60 0000000000600c60 00000c60 2\*\*3  CONTENTS, ALLOC, LOAD, DATA  21 .got 00000008 0000000000600e20 0000000000600e20 00000e20 2\*\*3  CONTENTS, ALLOC, LOAD, DATA  22 .got.plt 00000040 0000000000600e28 0000000000600e28 00000e28 2\*\*3  CONTENTS, ALLOC, LOAD, DATA  23 .data 0000000c 0000000000600e68 0000000000600e68 00000e68 2\*\*2  CONTENTS, ALLOC, LOAD, DATA  24 .bss 00000020 0000000000600e78 0000000000600e78 00000e74 2\*\*3  ALLOC  25 .comment 00000058 0000000000000000 0000000000000000 00000e74 2\*\*0  CONTENTS, READONLY |

MAPファイル出力例： ※GCC系のコンパイル結果の一部抜粋

|  |
| --- |
| .text 0x0000000000400634 0x16b /tmp/ccg1iYOV.o  0x0000000000400634 main  0x000000000040070a \_Z32auto\_run\_test\_func\_constructor\_1v  0x000000000040071a \_Z32auto\_run\_test\_func\_constructor\_2v  0x000000000040072a \_Z31auto\_run\_test\_func\_destructor\_1v  0x000000000040073a \_Z31auto\_run\_test\_func\_destructor\_2v  \*fill\* 0x000000000040079f 0x1 90909090  .text.\_ZN19auto\_run\_test\_classC2Ev  0x00000000004007a0 0x18 /tmp/ccg1iYOV.o  0x00000000004007a0 \_ZN19auto\_run\_test\_classC2Ev  0x00000000004007a0 \_ZN19auto\_run\_test\_classC1Ev  .text.\_ZN19auto\_run\_test\_classD2Ev  0x00000000004007b8 0x18 /tmp/ccg1iYOV.o  0x00000000004007b8 \_ZN19auto\_run\_test\_classD1Ev  0x00000000004007b8 \_ZN19auto\_run\_test\_classD2Ev  .text 0x00000000004007d0 0x99 /usr/lib64/libc\_nonshared.a(elf-init.oS)  0x00000000004007d0 \_\_libc\_csu\_fini  0x00000000004007e0 \_\_libc\_csu\_init |

## 機械語／アセンブラ／プログラムカウンタ

「プログラム」もメモリに展開されている「データ」の一種である。

「機械語」（「マシン後」）のコードは16進数（というか2進数）のコードである。メモリから機械語コードを一つずつ読み出して、命令コードとして実行していく。

例えば、0xB8というコードは「mov eax, \*\*\*\*」という命令コードと解釈し、「続く4バイトの値をeax レジスタに格納する」という処理を行う。

上記の「mov eax, \*\*\*\*」は、機械語コー「0xB8」を、人が見て分かり易い表記に置き換えたものである。この表記法のことを「ニーモニック」と呼び、ニーモニック表記で書かれた機械語プログラムを「アセンブラ」と呼ぶ。

CPUは「現在どのメモリのプログラムを実行しているのか？」という情報を「プログラムカウンタ」という専用のレジスタで管理している。実行中のプログラムのメモリアドレスが格納されるレジスタである。

以下、機械語とアセンブラのサンプルを示す。（Visual C++ 2013）

アセンブラのサンプル：※左側の数字がアドレス（プログラムカウンタはこの値を保持）

|  |
| --- |
| int main(const int argc, const char\* argv[])  {  00051520 55 push ebp  00051521 8B EC mov ebp,esp  00051523 81 EC 00 01 00 00 sub esp,100h  00051529 53 push ebx  0005152A 56 push esi  0005152B 57 push edi  0005152C 8D BD 00 FF FF FF lea edi,[ebp-100h]  00051532 B9 40 00 00 00 mov ecx,40h  00051537 B8 CC CC CC CC mov eax,0CCCCCCCCh  0005153C F3 AB rep stos dword ptr es:[edi]  0005153E A1 00 80 05 00 mov eax,dword ptr ds:[00058000h]  00051543 33 C5 xor eax,ebp  00051545 89 45 FC mov dword ptr [ebp-4],eax  printf("main():begin\n");  00051548 8B F4 mov esi,esp  0005154A 68 E0 58 05 00 push 558E0h  0005154F FF 15 D8 90 05 00 call dword ptr ds:[590D8h]  00051555 83 C4 04 add esp,4  00051558 3B F4 cmp esi,esp  0005155A E8 09 FC FF FF call \_\_RTC\_CheckEsp (051168h)  static int static\_without\_value; //[.bss]初期値なし局所静的変数  static int static\_with\_value = 3; //[.data]初期値付き局所静的変数  static const int static\_const\_value = 4; //[.rodata]局所静的const変数（定数）  int auto\_without\_value; //[スタック]初期値なしローカル変数  int auto\_with\_value = 5; //[スタック]初期値付きローカル変数　※初期値はプログラムコード化  0005155F C7 45 E8 05 00 00 00 mov dword ptr [auto\_with\_value],5  const int auto\_const\_value = 6; //[スタック]ローカルconst変数（定数）　※初期値はプログラムコード化  00051566 C7 45 DC 06 00 00 00 mov dword ptr [auto\_const\_value],6  const char\* auto\_str = "STRING"; //[スタック]初期値付きローカル変数　※[.rodata]文字列リテラル  0005156D C7 45 D0 74 58 05 00 mov dword ptr [auto\_str],55874h |

スレッドごとにプログラムカウンタを記録しており、スレッドを切り替える時にプログラムカウンタ（とスタックポインタ）を切り替えることで、スレッドの挙動を実現する。

余談だが、かつてメモリ量が64KBにも満たないPCで動作していたプログラムは、少ないメモリをやりくりするために「自己書き変えコード」を積極的に活用していた。上記の「move eax, \*\*\*\*」の「\*\*\*\*」に該当するコードを直接書き換えたり、条件分岐のジャンプ先アドレスを直接書き換えたりといったことを、プログラム実行中に行う手法である。

今はプログラム領域が読み取り専用に保護されているので、このような手法はそうそう使えない。

## スタック領域

スタック領域とは、関数内のローカル変数と、関数呼び出しからの復帰のために使用されるメモリ領域のこと。

スタック領域を説明するために、まずはプログラムのサンプルを示す。

スタック領域確認のためのプログラムのサンプル：

|  |
| --- |
| //関数２  void func2()  {  printf("func2()\n");  int local\_var2 = 0;  printf("&local\_var2 =%p\n", &local\_var2);  }  //関数１  void func1()  {  printf("func1()\n");  int local\_var1a = 0;  int local\_var1b = 0;  int local\_var1c = 0;  printf("&local\_var1a=%p\n", &local\_var1a);  printf("&local\_var1b=%p\n", &local\_var1b);  printf("&local\_var1c=%p\n", &local\_var1c);  func2();  }  //メイン  int main(const int argc, const char\* argv[])  {  printf("main()\n");  int local\_var0 = 0;  printf("&local\_var0 =%p\n", &local\_var0);  func1();  func2();  return 0;  } |

↓（実行結果）

|  |
| --- |
| main()  &local\_var0 =010EF7F8  func1()  &local\_var1a=010EF708 ←関数の呼び出しによってリターンアドレスがスタックに積まれ、SPが上に移動している  &local\_var1b=010EF6FC ←ローカル変数がスタックに積まれ、SPが上に移動している  &local\_var1c=010EF6F0 ←（同上）  func2()  &local\_var2 =010EF60C ←func1から呼び出されたので、さらにSPが上に移動している  func2()  &local\_var2 =010EF708 ←mainに戻って（スタックを戻して）から再度呼び出したので、同じ関数でもSPが変わっている |

この時、スタック領域は下記のように使用される。スタックは領域の最下層から順に積まれていき、一つの関数が使用する範囲を「スタックフレーム」と呼ぶ。



スタック領域は、スレッドごとにそれぞれ個別に割り当てられる。

スレッドに割り当てるスタックは、OSに任せて既定の領域を確保さるか、自分でサイズを指定してOSに領域を確保させるか、自分で確保した領域を渡すかする。

スタックサイズを超える関数呼び出しや変数確保が行われると、「スタックオーバーフロー」が起こり、プログラム全体が停止する。

スタックオーバーフローのサンプル①：変数が大きすぎる

|  |
| --- |
| //関数１  void func1()  {  printf("func1()\n");  int local\_var[16 \* 1024 \* 1024] = {}; //←【問題の箇所】  }  //メイン  int main(const int argc, const char\* argv[])  {  printf("main()\n");  func1();  return 0;  } |

↓（実行結果）※Windowsの場合

|  |
| --- |
| main()  ハンドルされない例外が 0x00C11777 (test.exe) で発生しました: 0xC00000FD: Stack overflow (パラメーター: 0x00000000, 0x000E2000)。 |

↓（実行結果）※Linuxの場合

|  |
| --- |
| main()  セグメンテーション違反です (コアダンプ) |

スタックオーバーフローのサンプル②：無限再帰

|  |
| --- |
| //関数１  void func1()  {  func1(); //←【問題の箇所】  }  //メイン  int main(const int argc, const char\* argv[])  {  printf("main()\n");  func1();  return 0;  } |

※実行結果はサンプル①と同じ

GCC 4.6 以降のコンパイラを使用している場合、 -fstack-usege というオプションを付けてコンパイルすると、関数ごとのスタック所要量を確認できる。上記オーバーフローのサンプル①のコードの確認結果を示す。

スタック所要量の確認例：

|  |
| --- |
| $ g++ --version  g++ (GCC) 4.8.2  $ g++ -fstack-usage a.cpp  $ a.su  a.cpp:14:6:void func1() 67108896 static  a.cpp:27:5:int main(int, const char\*\*) 32 static |

## スレッドとコンテキストスイッチ

アクティブなタスクが切り替わることを「コンテキストスイッチ」と呼ぶ。

「コンテキスト」とは、タスクを実行中のCPUの状態のことで、スイッチする際に全てのレジスタの情報を退避・復元する。レジスタの中にはプログラムカウンタとスタックポインタも含まれる。

スレッドの挙動としては、スレッドがタイムスライスの所要時間（「クォンタム」とも言う）を使い切った時か、スレッドが待機状態（スリープ）に入った時にコンテキストスイッチが発生する。

なお、コンテキストスイッチはスレッドの切り替えだけではなく、OSのカーネルモードとユーザーモードが切り替わる時や、割り込み処理が発生した際にも行われる。

## スレッドスケジューリング

スレッドの切り替えはOSが管理する。スレッドの実行順序はOSにスケジューリングされる。スレッドは順番にタイムスライスで区切りながらローテーションで実行されていく。

## スレッド優先度

スレッドには優先度がある。

スレッドスケジューリングは優先度の高いスレッドを先に実行し、それらが全て待機状態にならない限り、低いスレッドに実行を移さない。優先度の同じスレッドはタイムスライスで順次実行する。

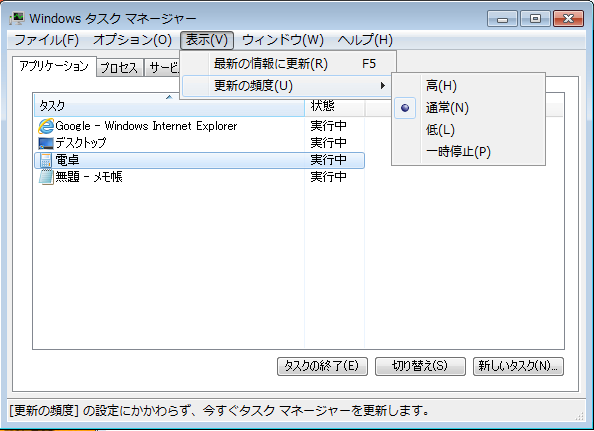
要するに、待ち状態に入らずにタイムスライスを使い切ってばかりの重いスレッドがあると、それより優先度の低いスレッドに処理が移らなくなる。

スレッド優先度のイメージ：



なお、スレッド優先度は、スレッド生成時、もしくは実行中に切り替えることができる。

Windowsでは、プロセスの優先度をタスクマネージャから切り替えることもできる。これは、プロセス内のスレッドの優先度に影響する。



余談になるが、Windows8.1の環境で試してみたところ、無限ループを行う処理を優先度最高で実行しても（プログラム内で最高レベルに強制変更しても）、（論理プロセッサの一つが）100%になるようなことにはならなかったので、ある程度処理が続いたら強制的に待機状態になっているようである。ただし、この挙動はOSによって異なると思われる。

# 様々なスレッド

幾つかのマルチスレッドプログラミング技法の説明とサンプルプログラムを示す。

ゲームプログラミングにおけるスレッドは、スタック領域のサイズやスレッド優先度に対して特に神経質になったほうがよい。できればC++11のスタイルで汎用性の高いプログラミングを行いたいところだが、今のところ、そうした細かい調整には対応できていないようである。ゲームプログラミングでは、ゲーム用SDKに基づいて細かく制御した方が良い。

## fork

まずは「fork」（フォーク）を説明する。

マルチスレッドプログラミングを学習すると、同じ言葉で意味の異なる「fork-joinモデル」という言葉もあり、混乱を招きかねないので、あえて旧来からのforkを説明する。なお、「fork-joinモデル」は、分散・並列処理の処理モデルのこと。

forkは、Unix系OSで伝統的なマルチタスクプログラミングの手法。マルチスレッドではなく、マルチプロセスである。マルチスレッドの原点的な技術。

関数をスレッド化する普通のマルチスレッドプログラミングを先に学習していると奇妙に見えるコードである。マルチスレッドよりもコードを短くまとめることができる。

マルチプロセスなのでメモリ空間の共有ができず、また、スレッドに比べて所用メモリ量が大きい。

forkのサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <errno.h>  #include <sys/types.h>  #include <unistd.h>  #include <sys/wait.h>  //fork テスト  int main(const int argc, const char\* argv[])  {  //プログラム実行開始  printf("- Start Fork Test -\n");  fflush(stdout);    //テスト用変数  const char\* test\_text = "TEST";  int test\_value = 1000;  int test\_loop\_max = 1;    //プロセス名を作成  // P + 子プロセスのレベル + ( 自プロセスID , 親プロセスID )  char process\_name[32];  sprintf(process\_name, "P0(%5d,%5d)", getpid(), getppid());  int level = 0;    //大元のプロセス判定用フラグ（初期値は true）  bool is\_root\_process = true;    //子プロセス生成ループ  const int CHILD\_THREAD\_NUM = 3;  pid\_t pid[CHILD\_THREAD\_NUM];//生成した子プロセスのプロセスID  for(int i = 0; i < CHILD\_THREAD\_NUM; ++i)  {  //子プロセス生成  pid\_t pid\_tmp = fork();  //※fork()が実行された瞬間に子プロセスが生成される。  //※子プロセスは、親プロセスの静的変数やスタック（ローカル変数）、実行位置（プログラムカウンタ）、  //　および、オープン中のファイルディスクリプタをコピーして、新しいプロセスを作る。  //※実行位置もコピーされるため、子プロセスはこの位置からスタートする。  //　すべての変数の値も引き継がられるが、メモリを共有するわけではない。  //　あくまでもその時点の内容がコピーされるだけ。    //生成結果確認  switch(pid\_tmp)  {  //fork 成功：自プロセスが「生成された子プロセス」の場合  case 0:  //大元のプロセス判定用フラグをOFF  is\_root\_process = false;    //プロセス名を作成  ++ level;  sprintf(process\_name, "C%d(%5d,%5d)", level, getpid(), getppid());  // C + 子プロセスのレベル + ( 自プロセスID , 親プロセスID )    //子プロセス開始メッセージ  printf("START: %s\n", process\_name);  fflush(stdout);    //テスト用変数を更新  ++test\_value;  ++test\_loop\_max;    //親プロセスが生成した子プロセスのIDのコピーされてしまっているのでクリアする  for(int ii = 0; ii <= i; ++ ii)  pid[ii] = -1;    //1秒スリープ  sleep(1);    break;    //fork 失敗  case -1:  //エラーメッセージ  if(is\_root\_process)  fprintf(stderr, "%s -> [%d] FAILED (errno=%d)\n", process\_name, i, errno);  else  fprintf(stderr, "%s -> [%d] FAILED (errno=%d)\n", process\_name, i, errno);  fflush(stderr);    //1秒スリープ  sleep(1);    break;    //fork 成功：子プロセスの生成に成功し、子プロセスのプロセスIDを取得  default:  //子プロセスのプロセスIDを記録（最後の終了待ち用）  pid[i] = pid\_tmp;    //子プロセス生成メッセージ  if(is\_root\_process)  printf("%s -> [%d] CREATED:C%d(%5d,%5d)\n", process\_name, i, level + 1, pid\_tmp, getpid());  else  printf("%s -> [%d] CREATED:C%d(%5d,%5d)\n", process\_name, i, level + 1, pid\_tmp, getpid());  fflush(stdout);    //2秒スリープ  sleep(2);    break;  }  }    //ダミー処理  for(int i = 0; i < test\_loop\_max; ++i)  {  //ダミーメッセージ表示  printf("[%s] ... Process(%d/%d): text=\"%s\",value=%d\n", process\_name, i + 1, test\_loop\_max, test\_text, test\_value);  fflush(stdout);  test\_value += 10;    //1秒スリープ  sleep(1);  }    //子プロセスの終了待ち  for(int i = 0; i < CHILD\_THREAD\_NUM; ++i)  {  pid\_t pid\_tmp = pid[i];  if(pid\_tmp > 0)  {  //ウェイト開始メッセージ  printf("[%s] ... Wait(%5d) ...\n", process\_name, pid\_tmp);  fflush(stdout);    //ウェイト：子プロセス終了待ち  int status = -1;  waitpid(pid\_tmp, &status, WUNTRACED);    //ウェイト完了メッセージ  printf("[%s] ... Wait(%5d) End status=%d\n", process\_name, pid\_tmp, status);  fflush(stdout);  }  }    //自プロセスが子プロセスならこの時点で実行終了  if(!is\_root\_process)  {  //子プロセス終了メッセージ  printf("END: %s\n", process\_name);  fflush(stdout);    //プロセス終了  exit(0);  }    //プログラム実行終了  printf("- End Fork Test -\n");  fflush(stdout);  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| $ ./fork.exe ※プロセスの親子関係が分かるように着色している  - Start Fork Test -  P0( 5348,12044) -> [0] CREATED:C1( 6464, 5348)  START: C1( 6464, 5348)  C1( 6464, 5348) -> [1] CREATED:C2( 6752, 6464)  START: C2( 6752, 6464)  P0( 5348,12044) -> [1] CREATED:C1(11556, 5348)  START: C1(11556, 5348)  C2( 6752, 6464) -> [2] CREATED:C3( 1760, 6752)  START: C3( 1760, 6752)  [C3( 1760, 6752)] ... Process(1/4): text="TEST",value=1003  C1( 6464, 5348) -> [2] CREATED:C2(12152, 6464)  START: C2(12152, 6464)  C1(11556, 5348) -> [2] CREATED:C2(10372,11556)  START: C2(10372,11556)  [C2( 6752, 6464)] ... Process(1/3): text="TEST",value=1002  [C3( 1760, 6752)] ... Process(2/4): text="TEST",value=1013  P0( 5348,12044) -> [2] CREATED:C1( 8972, 5348)  START: C1( 8972, 5348)  [C2(12152, 6464)] ... Process(1/3): text="TEST",value=1002  [C2(10372,11556)] ... Process(1/3): text="TEST",value=1002  [C2( 6752, 6464)] ... Process(2/3): text="TEST",value=1012  [C3( 1760, 6752)] ... Process(3/4): text="TEST",value=1023  [C1( 8972, 5348)] ... Process(1/2): text="TEST",value=1001  [C2(12152, 6464)] ... Process(2/3): text="TEST",value=1012  [C1( 6464, 5348)] ... Process(1/2): text="TEST",value=1001  [C1(11556, 5348)] ... Process(1/2): text="TEST",value=1001  [C2(10372,11556)] ... Process(2/3): text="TEST",value=1012  [C2( 6752, 6464)] ... Process(3/3): text="TEST",value=1022  [C3( 1760, 6752)] ... Process(4/4): text="TEST",value=1033  [P0( 5348,12044)] ... Process(1/1): text="TEST",value=1000  [C1( 8972, 5348)] ... Process(2/2): text="TEST",value=1011  [C1( 6464, 5348)] ... Process(2/2): text="TEST",value=1011  [C2(12152, 6464)] ... Process(3/3): text="TEST",value=1022  [C2(10372,11556)] ... Process(3/3): text="TEST",value=1022  [C1(11556, 5348)] ... Process(2/2): text="TEST",value=1011  END: C3( 1760, 6752)  [C2( 6752, 6464)] ... Wait( 1760) ...  [C2( 6752, 6464)] ... Wait( 1760) End status=0  END: C2( 6752, 6464)  END: C1( 8972, 5348)  [P0( 5348,12044)] ... Wait( 6464) ...  END: C2(12152, 6464)  [C1( 6464, 5348)] ... Wait( 6752) ...  END: C2(10372,11556)  [C1(11556, 5348)] ... Wait(10372) ...  [C1( 6464, 5348)] ... Wait( 6752) End status=0  [C1( 6464, 5348)] ... Wait(12152) ...  [C1( 6464, 5348)] ... Wait(12152) End status=0  END: C1( 6464, 5348)  [C1(11556, 5348)] ... Wait(10372) End status=0  END: C1(11556, 5348)  [P0( 5348,12044)] ... Wait( 6464) End status=0  [P0( 5348,12044)] ... Wait(11556) ...  [P0( 5348,12044)] ... Wait(11556) End status=0  [P0( 5348,12044)] ... Wait( 8972) ...  [P0( 5348,12044)] ... Wait( 8972) End status=0  - End Fork Test - |

↓（実行中のプロセスの状態）

|  |
| --- |
| $ ps –ef | grep fork ※多数のプロセスが生成されていることがわかる  user\_name 5348 12044 pty0 07:48:54 /cygdrive/c/Work/test/Program/C++/Thread/fork/fork  user\_name 1760 6752 pty0 07:48:56 /cygdrive/c/Work/test/Program/C++/Thread/fork/fork  user\_name 11556 5348 pty0 07:48:56 /cygdrive/c/Work/test/Program/C++/Thread/fork/fork  user\_name 12152 6464 pty0 07:48:57 /cygdrive/c/Work/test/Program/C++/Thread/fork/fork  user\_name 8972 5348 pty0 07:48:58 /cygdrive/c/Work/test/Program/C++/Thread/fork/fork  user\_name 6752 6464 pty0 07:48:55 /cygdrive/c/Work/test/Program/C++/Thread/fork/fork  user\_name 6464 5348 pty0 07:48:54 /cygdrive/c/Work/test/Program/C++/Thread/fork/fork  user\_name 10372 11556 pty0 07:48:57 /cygdrive/c/Work/test/Program/C++/Thread/fork/fork |

## POSIXスレッドライブラリ版スレッド

POSIXスレッドライブラリは、Unix系OSで標準的なライブラリ。

前述のfork版と同じ挙動のサンプルを示す。

関数の切り分けと変数の受け渡しを明示的に行う必要がある都合から、forkと同じ挙動にしようとすると、少し複雑になることが分かる。

fork版との比較用に用意したサンプルだが、スレッドのサンプルとしては少し複雑すぎる。後述する同期処理のサンプルがもっとシンプルで分かり易い。

このサンプルプログラムでは、malloc() で確保したメモリを子スレッドに受け渡している。子スレッド側では受け取ってすぐに不要になメモリだが、すぐにfree() していない。malloc() 時と同じスレッドでfree() しないとハングするのでそのままにしている。

あわせてTLS（スレッドローカルストレージ）の動作もテスト。

サンプルには示していないが、スレッド生成時の属性指定により、スタック領域のサイズやスレッド優先度などを設定することが可能。

POSIXスレッドライブラリ版スレッドのサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <errno.h>  #include <sys/types.h>  #include <pthread.h>  #include <unistd.h>  //スレッドローカルストレージ(TLS)テスト  \_\_thread pthread\_t tls\_pth = 0; //POSIXスレッドライブラリ版TLS指定  //テスト用情報  struct TEST\_INFO  {  const char\* text;//文字列  int value; //数値  int loop\_max; //テストループ回数  int start\_i; //スレッド生成ループ開始値  };  //スレッド情報  struct THREAD\_INFO  {  int level; //子スレッドレベル  pthread\_t parent\_pth;//親スレッド\_t  bool is\_root\_thread; //大元のスレッド判定用フラグ  };  //スレッド受け渡し情報  struct THREAD\_PARAM  {  TEST\_INFO test; //テスト用情報  THREAD\_INFO thread;//スレッド情報  };  //子スレッドの生成と終了待ち  extern void createChildThreads(const char\* thread\_name, TEST\_INFO& test, THREAD\_INFO& tinfo);  //スレッド関数  void\* threadFunc(void\* param\_p)  {  //パラメータ受け取り  TEST\_INFO test;  THREAD\_INFO tinfo;  {  THREAD\_PARAM\* param = (THREAD\_PARAM\*)param\_p;  memcpy(&test, &param->test, sizeof(TEST\_INFO));  memcpy(&tinfo, &param->thread, sizeof(THREAD\_INFO));  // free(param\_p); //この時点で受け取ったメモリは不要だが、元のスレッドでfree しないとハングする  }    //スレッド情報  pthread\_t pth\_tmp = pthread\_self();    //スレッドローカルストレージ(TLS)テスト  tls\_pth = pth\_tmp;    //大元のスレッド判定用フラグをOFF  tinfo.is\_root\_thread = false;    //スレッド名を作成  ++ tinfo.level;  char thread\_name[32];  sprintf(thread\_name, "C%d(%010p,%010p)", tinfo.level, pth\_tmp, tinfo.parent\_pth);  // C + 子スレッドのレベル + ( 自スレッド\_t , 親スレッド\_t )    //子スレッド開始メッセージ  printf("START: %s\n", thread\_name);  fflush(stdout);    //テスト用変数を更新  ++ test.value;  ++ test.loop\_max;    //親スレッド\_tを格納  tinfo.parent\_pth = pth\_tmp;    //1秒スリープ  sleep(1);    //子スレッドの生成と終了待ち  createChildThreads(thread\_name, test, tinfo);    //子スレッド終了メッセージ  printf("END: %s\n", thread\_name);  fflush(stdout);    return NULL;  }  //子スレッドの生成と終了待ち  void createChildThreads(const char\* thread\_name, TEST\_INFO& test, THREAD\_INFO& tinfo)  {  //子スレッド生成ループ  const int CHILD\_THREAD\_NUM = 3;  pthread\_t pth[CHILD\_THREAD\_NUM]; //生成した子スレッドのスレッド\_t  THREAD\_PARAM\* param[CHILD\_THREAD\_NUM];//スレッド受け渡し用データ  for(int i = 0; i < test.start\_i; ++i)  {  pth[i] = 0;  param[i] = NULL;  }  for(int i = test.start\_i; i < CHILD\_THREAD\_NUM; ++i)  {  //子スレッド用パラメータ作成  param[i] = (THREAD\_PARAM\*)malloc(sizeof(THREAD\_PARAM));  memcpy(&param[i]->test, &test, sizeof(TEST\_INFO));  memcpy(&param[i]->thread, &tinfo, sizeof(THREAD\_INFO));  param[i]->test.start\_i = i + 1;    //子スレッド生成  pthread\_t pth\_tmp = 0;  int ret = pthread\_create((pthread\_t\*)&pth\_tmp, NULL, threadFunc, (void\*)(param[i]));  if(ret != 0)  {  //子スレッド生成失敗  free(param[i]);  param[i] = NULL;    //エラーメッセージ  if(tinfo.is\_root\_thread)  fprintf(stderr, "%s -> [%d] FAILED (errno=%d)\n", thread\_name, i, ret, errno);  else  fprintf(stderr, "%s -> [%d] FAILED (errno=%d)\n", thread\_name, i, ret, errno);  fflush(stderr);    //1秒スリープ  sleep(1);  }  else  {  //子スレッド生成成功    //子スレッドのスレッド\_tを記録（最後の終了待ち用）  pth[i] = pth\_tmp;    //子スレッド生成メッセージ  if(tinfo.is\_root\_thread)  printf("%s -> [%d] CREATED:C%d(%010p,%010p)\n", thread\_name, i, tinfo.level + 1, pth\_tmp, tinfo.parent\_pth);  else  printf("%s -> [%d] CREATED:C%d(%010p,%010p)\n", thread\_name, i, tinfo.level + 1, pth\_tmp, tinfo.parent\_pth);  fflush(stdout);    //2秒スリープ  sleep(2);  }  }    //ダミー処理  for(int i = 0; i < test.loop\_max; ++i)  {  //ダミーメッセージ表示  printf("[%s] ... Process(%d/%d): text=\"%s\",value=%d (tls\_tid=%010p)\n", thread\_name, i + 1, test.loop\_max, test.text, test.value, tls\_pth);  fflush(stdout);  test.value += 10;    //1秒スリープ  sleep(1);  }    //子スレッドの終了待ち  for(int i = 0; i < CHILD\_THREAD\_NUM; ++i)  {  pthread\_t pth\_tmp = pth[i];  if(pth\_tmp > 0)  {  //ウェイト開始メッセージ  printf("[%s] ... Wait(%010p) ...\n", thread\_name, pth\_tmp);  fflush(stdout);    //ウェイト：子スレッド終了待ち  pthread\_join((pthread\_t)pth\_tmp, NULL);    //メモリ解放  free(param[i]);  param[i] = NULL;    //ウェイト完了メッセージ  printf("[%s] ... Wait(%010p) End\n", thread\_name, pth\_tmp);  fflush(stdout);  }  }  }  //posix thread テスト  int main(const int argc, const char\* argv[])  {  //プログラム実行開始  printf("- Start Thread Test -\n");  fflush(stdout);    //テスト用変数  TEST\_INFO test =  {  "TEST", //text  1000, //value  1, //loop\_max  0, //start\_i  };    //スレッド情報  THREAD\_INFO tinfo =  {  0, //level  pthread\_self(),  //pareant\_pth  true,//is\_root\_thread  };    //スレッドローカルストレージ(TLS)テスト  tls\_pth = tinfo.parent\_pth;    //スレッド名を作成  // P + 子スレッドのレベル + ( 自スレッド\_t , 親スレッド\_t )  char thread\_name[32];  sprintf(thread\_name, "P0(%010p,%010p)", tinfo.parent\_pth, 0);    //子スレッドの生成と終了待ち  createChildThreads(thread\_name, test, tinfo);    //プログラム実行終了  printf("- End Thread Test -\n");  fflush(stdout);  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| $ ./pthread.exe ※スレッドの親子関係は fork 版と同じなので着色は省略  - Start Thread Test - ※TLS属性のついたグローバル変数 tls\_pth が、スレッド毎に異なる値になっている点に注目  P0(0x80000038,0x00000000) -> [0] CREATED:C1(0x800203d0,0x80000038)  START: C1(0x800203d0,0x80000038)  C1(0x800203d0,0x80000038) -> [1] CREATED:C2(0x80020520,0x800203d0)  START: C2(0x80020520,0x800203d0)  P0(0x80000038,0x00000000) -> [1] CREATED:C1(0x80020720,0x80000038)  START: C1(0x80020720,0x80000038)  C2(0x80020520,0x800203d0) -> [2] CREATED:C3(0x80020690,0x80020520)  START: C3(0x80020690,0x80020520)  [C3(0x80020690,0x80020520)] ... Process(1/4): text="TEST",value=1003 (tls\_tid=0x80020690)  C1(0x800203d0,0x80000038) -> [2] CREATED:C2(0x80020930,0x800203d0)  START: C2(0x80020930,0x800203d0)  C1(0x80020720,0x80000038) -> [2] CREATED:C2(0x800209c0,0x80020720)  START: C2(0x800209c0,0x80020720)  [C2(0x800209c0,0x80020720)] ... Process(1/3): text="TEST",value=1002 (tls\_tid=0x800209c0)  [C3(0x80020690,0x80020520)] ... Process(2/4): text="TEST",value=1013 (tls\_tid=0x80020690)  [C2(0x80020930,0x800203d0)] ... Process(1/3): text="TEST",value=1002 (tls\_tid=0x80020930)  [C2(0x80020520,0x800203d0)] ... Process(1/3): text="TEST",value=1002 (tls\_tid=0x80020520)  P0(0x80000038,0x00000000) -> [2] CREATED:C1(0x80020bb0,0x80000038)  START: C1(0x80020bb0,0x80000038)  [C2(0x80020520,0x800203d0)] ... Process(2/3): text="TEST",value=1012 (tls\_tid=0x80020520)  [C2(0x800209c0,0x80020720)] ... Process(2/3): text="TEST",value=1012 (tls\_tid=0x800209c0)  [C2(0x80020930,0x800203d0)] ... Process(2/3): text="TEST",value=1012 (tls\_tid=0x80020930)  [C3(0x80020690,0x80020520)] ... Process(3/4): text="TEST",value=1023 (tls\_tid=0x80020690)  [C1(0x80020720,0x80000038)] ... Process(1/2): text="TEST",value=1001 (tls\_tid=0x80020720)  [C1(0x800203d0,0x80000038)] ... Process(1/2): text="TEST",value=1001 (tls\_tid=0x800203d0)  [C1(0x80020bb0,0x80000038)] ... Process(1/2): text="TEST",value=1001 (tls\_tid=0x80020bb0)  [C1(0x80020720,0x80000038)] ... Process(2/2): text="TEST",value=1011 (tls\_tid=0x80020720)  [C2(0x80020930,0x800203d0)] ... Process(3/3): text="TEST",value=1022 (tls\_tid=0x80020930)  [P0(0x80000038,0x00000000)] ... Process(1/1): text="TEST",value=1000 (tls\_tid=0x80000038)  [C1(0x800203d0,0x80000038)] ... Process(2/2): text="TEST",value=1011 (tls\_tid=0x800203d0)  [C3(0x80020690,0x80020520)] ... Process(4/4): text="TEST",value=1033 (tls\_tid=0x80020690)  [C1(0x80020bb0,0x80000038)] ... Process(2/2): text="TEST",value=1011 (tls\_tid=0x80020bb0)  [C2(0x80020520,0x800203d0)] ... Process(3/3): text="TEST",value=1022 (tls\_tid=0x80020520)  [C2(0x800209c0,0x80020720)] ... Process(3/3): text="TEST",value=1022 (tls\_tid=0x800209c0)  END: C1(0x80020bb0,0x80000038)  END: C2(0x80020930,0x800203d0)  END: C3(0x80020690,0x80020520)  END: C2(0x800209c0,0x80020720)  [P0(0x80000038,0x00000000)] ... Wait(0x800203d0) ...  [C1(0x800203d0,0x80000038)] ... Wait(0x80020520) ...  [C2(0x80020520,0x800203d0)] ... Wait(0x80020690) ...  [C1(0x80020720,0x80000038)] ... Wait(0x800209c0) ...  [C2(0x80020520,0x800203d0)] ... Wait(0x80020690) End  END: C2(0x80020520,0x800203d0)  [C1(0x80020720,0x80000038)] ... Wait(0x800209c0) End  END: C1(0x80020720,0x80000038)  [C1(0x800203d0,0x80000038)] ... Wait(0x80020520) End  [C1(0x800203d0,0x80000038)] ... Wait(0x80020930) ...  [C1(0x800203d0,0x80000038)] ... Wait(0x80020930) End  END: C1(0x800203d0,0x80000038)  [P0(0x80000038,0x00000000)] ... Wait(0x800203d0) End  [P0(0x80000038,0x00000000)] ... Wait(0x80020720) ...  [P0(0x80000038,0x00000000)] ... Wait(0x80020720) End  [P0(0x80000038,0x00000000)] ... Wait(0x80020bb0) ...  [P0(0x80000038,0x00000000)] ... Wait(0x80020bb0) End  - End Thread Test - |

↓（実行中のプロセスの状態）

|  |
| --- |
| $ ps -ef | grep pthread ※プロセスが一つしか生成されていることがわかる  user\_name 8140 11912 pty0 11:38:07 /cygdrive/e/Work/test/Program/C++/Thread/pthread/pthread |

## Win32API版スレッド

前述のfork版、POSIXスレッドライブラリ版スレッドと同じ挙動のサンプルを示す。

比較用に用意したサンプルだが、スレッドのサンプルとしては少し複雑すぎる。後述する同期処理のサンプルがもっとシンプルで分かり易い。

スレッド生成関数、終了待ち関数、Sleep()関数の扱いがPOSIXスレッドライブラリ版と異なる。

あわせて、Windows版のTLS（スレッドローカルストレージ）の動作もテスト。

サンプルには示していないが、スレッド生成時にスタック領域（のサイズ）やスレッド優先度などを設定することが可能。

Windowsのスレッド生成用関数には、CreateThread() 関数、\_beginthread() 関数、\_beginthreadex() 関数がある。

CreateThread() 関数で呼び出したスレッド関数の中ではCランタイムライブラリが使えず、また自身のスレッドハンドルをクローズできないという問題があるため、基本的に使用しないこと。

\_beginthread() 関数と\_beginthreadex() 関数の違いは、スレッド生成時に細かい制御ができるかどうかという点。

なお、fork と同様のコピープロセスを生成する仕組みはWindowsにはないが、実行ファイル（.exe）を指定してプロセス／子プロセスを作成する仕組みは用意されている。（サンプルは割愛）

Win32API版スレッドのサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <windows.h>  #include <process.h>  //スレッドローカルストレージ(TLS)テスト  \_\_declspec(thread) unsigned int tls\_tid = 0;//Visual C++固有版TLS指定  //テスト用情報  struct TEST\_INFO  {  const char\* text;//文字列  int value; //数値  int loop\_max; //テストループ回数  int start\_i; //スレッド生成ループ開始値  };  //スレッド情報  struct THREAD\_INFO  {  int level; //子スレッドレベル  unsigned int parent\_tid;//親スレッドID  bool is\_root\_thread; //大元のスレッド判定用フラグ  };  //スレッド受け渡し情報  struct THREAD\_PARAM  {  TEST\_INFO test; //テスト用情報  THREAD\_INFO thread;//スレッド情報  };  //子スレッドの生成と終了待ち  extern void createChildThreads(const char\* thread\_name, TEST\_INFO& test, THREAD\_INFO& tinfo);  //スレッド関数  unsigned int WINAPI threadFunc(void\* param\_p)  {  //パラメータ受け取り  TEST\_INFO test;  THREAD\_INFO tinfo;  {  THREAD\_PARAM\* param = (THREAD\_PARAM\*)param\_p;  memcpy(&test, &param->test, sizeof(TEST\_INFO));  memcpy(&tinfo, &param->thread, sizeof(THREAD\_INFO));  }  //スレッド情報  unsigned int tid\_tmp = GetCurrentThreadId();  //スレッドローカルストレージ(TLS)テスト  tls\_tid = tid\_tmp;  //大元のスレッド判定用フラグをOFF  tinfo.is\_root\_thread = false;  //スレッド名を作成  ++tinfo.level;  char thread\_name[32];  sprintf\_s(thread\_name, sizeof(thread\_name), "C%d(%5d,%5d)", tinfo.level, tid\_tmp, tinfo.parent\_tid);  // C + 子スレッドのレベル + ( 自スレッドID , 親スレッドID )  //子スレッド開始メッセージ  printf("START: %s\n", thread\_name);  fflush(stdout);  //テスト用変数を更新  ++test.value;  ++test.loop\_max;  //親スレッドIDを格納  tinfo.parent\_tid = tid\_tmp;  //1秒スリープ  Sleep(1000);  //子スレッドの生成と終了待ち  createChildThreads(thread\_name, test, tinfo);  //子スレッド終了メッセージ  printf("END: %s\n", thread\_name);  fflush(stdout);    //スレッド終了  \_endthreadex(0); //これを呼ぶ場合、呼び出し元でCloseHandle() してはダメ  //また、ローカル変数オブジェクトのデストラクタが呼び出されないので注意！  //呼ばない場合は、呼び出し元で CloseHandle() すること  return 0;  }  //子スレッドの生成と終了待ち  void createChildThreads(const char\* thread\_name, TEST\_INFO& test, THREAD\_INFO& tinfo)  {  //子スレッド生成ループ  const int CHILD\_THREAD\_NUM = 3;  HANDLE hthread[CHILD\_THREAD\_NUM]; //生成した子スレッドのスレッドハンドル  unsigned int tid[CHILD\_THREAD\_NUM]; //生成した子スレッドのスレッドID  THREAD\_PARAM\* param[CHILD\_THREAD\_NUM];//スレッド受け渡し用データ  for (int i = 0; i < test.start\_i; ++i)  {  hthread[i] = INVALID\_HANDLE\_VALUE;  tid[i] = 0;  param[i] = NULL;  }  for (int i = test.start\_i; i < CHILD\_THREAD\_NUM; ++i)  {  //子スレッド用パラメータ作成  param[i] = (THREAD\_PARAM\*)malloc(sizeof(THREAD\_PARAM));  memcpy(&param[i]->test, &test, sizeof(TEST\_INFO));  memcpy(&param[i]->thread, &tinfo, sizeof(THREAD\_INFO));  param[i]->test.start\_i = i + 1;  //子スレッド生成  unsigned int tid\_tmp = 0;  HANDLE hthread\_tmp = (HANDLE)\_beginthreadex(NULL, 0, threadFunc, (void\*)(param[i]), 0, &tid\_tmp);  if (hthread\_tmp == (HANDLE)1L || hthread\_tmp == INVALID\_HANDLE\_VALUE)  {  //子スレッド生成失敗  free(param[i]);  param[i] = NULL;  //エラーメッセージ  if (tinfo.is\_root\_thread)  fprintf(stderr, "%s -> [%d] FAILED (errno=%d)\n", thread\_name, i, errno);  else  fprintf(stderr, "%s -> [%d] FAILED (errno=%d)\n", thread\_name, i, errno);  fflush(stderr);  //1秒スリープ  Sleep(1000);  }  else  {  //子スレッド生成成功  //子スレッドのスレッドハンドルとIDを記録（最後の終了待ち用）  hthread[i] = hthread\_tmp;  tid[i] = tid\_tmp;  //子スレッド生成メッセージ  if (tinfo.is\_root\_thread)  printf("%s -> [%d] CREATED:C%d(%5d,%5d)\n", thread\_name, i, tinfo.level + 1, tid\_tmp, tinfo.parent\_tid);  else  printf("%s -> [%d] CREATED:C%d(%5d,%5d)\n", thread\_name, i, tinfo.level + 1, tid\_tmp, tinfo.parent\_tid);  fflush(stdout);  //2秒スリープ  Sleep(2000);  }  }  //ダミー処理  for (int i = 0; i < test.loop\_max; ++i)  {  //ダミーメッセージ表示  printf("[%s] ... Process(%d/%d): text=\"%s\",value=%d (tls\_tid=%d)\n", thread\_name, i + 1, test.loop\_max, test.text, test.value, tls\_tid);  fflush(stdout);  test.value += 10;  //1秒スリープ  Sleep(1000);  }  //子スレッドの終了待ち  for (int i = 0; i < CHILD\_THREAD\_NUM; ++i)  {  HANDLE hthread\_tmp = hthread[i];  unsigned int tid\_tmp = tid[i];  if (hthread\_tmp != (HANDLE)1L && hthread\_tmp != INVALID\_HANDLE\_VALUE)  {  //ウェイト開始メッセージ  printf("[%s] ... Wait(%5d) ...\n", thread\_name, tid\_tmp);  fflush(stdout);  //ウェイト：子スレッド終了待ち  WaitForSingleObject(hthread\_tmp, INFINITE);    //子スレッドのハンドルをクローズ  // CloseHandle(hthread\_tmp);//スレッド終了時に \_endthreadex() を呼んでいるなら CloseHandle() 不要    //メモリ解放  free(param[i]);  param[i] = NULL;  //ウェイト完了メッセージ  printf("[%s] ... Wait(%5d) End\n", thread\_name, tid\_tmp);  fflush(stdout);  }  }  }  //Windows thread テスト  int main(const int argc, const char\* argv[])  {  //プログラム実行開始  printf("- Start Thread Test -\n");  fflush(stdout);  //テスト用変数  TEST\_INFO test =  {  "TEST", //text  1000, //value  1, //loop\_max  0, //start\_i  };  //スレッド情報  THREAD\_INFO tinfo =  {  0, //level  GetCurrentThreadId(),  //parent\_tid  true,//is\_root\_thread  };  //スレッドローカルストレージ(TLS)テスト  tls\_tid = tinfo.parent\_tid;  //スレッド名を作成  // P + 子スレッドのレベル + ( 自スレッドID , 親スレッドID )  char thread\_name[32];  sprintf\_s(thread\_name, sizeof(thread\_name), "P0(%5d,%5d)", tinfo.parent\_tid, 0);  //子スレッドの生成と終了待ち  createChildThreads(thread\_name, test, tinfo);  //プログラム実行終了  printf("- End Thread Test -\n");  fflush(stdout);  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| ※スレッドの親子関係は fork 版と同じなので着色は省略  - Start Thread Test - ※TLS属性のついたグローバル変数 tls\_pth が、スレッド毎に異なる値になっている点に注目  P0(14168, 0) -> [0] CREATED:C1(13368,14168)  START: C1(13368,14168)  C1(13368,14168) -> [1] CREATED:C2( 472,13368)  START: C2( 472,13368)  P0(14168, 0) -> [1] CREATED:C1(11400,14168)  START: C1(11400,14168)  C2( 472,13368) -> [2] CREATED:C3( 336, 472)  START: C3( 336, 472)  C1(13368,14168) -> [2] CREATED:C2(12888,13368)  START: C2(12888,13368)  C1(11400,14168) -> [2] CREATED:C2(14400,11400)  START: C2(14400,11400)  [C3( 336, 472)] ... Process(1/4): text="TEST",value=1003 (tls\_tid=336)  [C2(12888,13368)] ... Process(1/3): text="TEST",value=1002 (tls\_tid=12888)  P0(14168, 0) -> [2] CREATED:C1(11304,14168)  START: C1(11304,14168)  [C2( 472,13368)] ... Process(1/3): text="TEST",value=1002 (tls\_tid=472)  [C3( 336, 472)] ... Process(2/4): text="TEST",value=1013 (tls\_tid=336)  [C2(14400,11400)] ... Process(1/3): text="TEST",value=1002 (tls\_tid=14400)  [C1(13368,14168)] ... Process(1/2): text="TEST",value=1001 (tls\_tid=13368)  [C2(12888,13368)] ... Process(2/3): text="TEST",value=1012 (tls\_tid=12888)  [C1(11304,14168)] ... Process(1/2): text="TEST",value=1001 (tls\_tid=11304)  [C1(11400,14168)] ... Process(1/2): text="TEST",value=1001 (tls\_tid=11400)  [C2( 472,13368)] ... Process(2/3): text="TEST",value=1012 (tls\_tid=472)  [C3( 336, 472)] ... Process(3/4): text="TEST",value=1023 (tls\_tid=336)  [C2(14400,11400)] ... Process(2/3): text="TEST",value=1012 (tls\_tid=14400)  [C1(13368,14168)] ... Process(2/2): text="TEST",value=1011 (tls\_tid=13368)  [P0(14168, 0)] ... Process(1/1): text="TEST",value=1000 (tls\_tid=14168)  [C1(11304,14168)] ... Process(2/2): text="TEST",value=1011 (tls\_tid=11304)  [C2(12888,13368)] ... Process(3/3): text="TEST",value=1022 (tls\_tid=12888)  [C1(11400,14168)] ... Process(2/2): text="TEST",value=1011 (tls\_tid=11400)  [C2( 472,13368)] ... Process(3/3): text="TEST",value=1022 (tls\_tid=472)  [C3( 336, 472)] ... Process(4/4): text="TEST",value=1033 (tls\_tid=336)  [C2(14400,11400)] ... Process(3/3): text="TEST",value=1022 (tls\_tid=14400)  [C1(13368,14168)] ... Wait( 472) ...  [P0(14168, 0)] ... Wait(13368) ...  END: C1(11304,14168)  END: C2(12888,13368)  [C1(11400,14168)] ... Wait(14400) ...  [C2( 472,13368)] ... Wait( 336) ...  END: C3( 336, 472)  [C2( 472,13368)] ... Wait( 336) End  END: C2( 472,13368)  END: C2(14400,11400)  [C1(13368,14168)] ... Wait( 472) End  [C1(13368,14168)] ... Wait(12888) ...  [C1(11400,14168)] ... Wait(14400) End  END: C1(11400,14168)  [C1(13368,14168)] ... Wait(12888) End  END: C1(13368,14168)  [P0(14168, 0)] ... Wait(13368) End  [P0(14168, 0)] ... Wait(11400) ...  [P0(14168, 0)] ... Wait(11400) End  [P0(14168, 0)] ... Wait(11304) ...  [P0(14168, 0)] ... Wait(11304) End  - End Thread Test - |

## C++11版スレッド

前述のWin32API版スレッドをベースにC++11版にしたサンプルを示す。

比較用に用意したサンプルだが、スレッドのサンプルとしては少し複雑すぎる。後述する同期処理のサンプルがもっとシンプルで分かり易い。

スレッド生成時にテンプレートで柔軟なパラメータを渡せるのが特徴。全体的にクラスベースの操作でシンプルなコードにできる。

TLS（スレッドローカルストレージ）は、Visual C++ 2013ではC++11版が使えないので、Windows版を使用。

なお、本書作成時点で、スタック領域サイズの指定方法とスレッド生成失敗時の判定方法を確認しきれず。スレッドクラスに .native\_handle() というメンバー関数があるので、それを使用して環境依存の調整を行う必要がありそうである。

C++11版スレッドのサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <thread>  //スレッドローカルストレージ(TLS)テスト  \_\_declspec(thread) std::size\_t tls\_tid = 0;//Visual C++固有版TLS指定  //thread\_local std::size\_t tls\_tid = 0;//C++11仕様版TLS指定：Visual C++ 2013 では未対応  //テスト用情報  struct TEST\_INFO  {  const char\* text;//文字列  int value; //数値  int loop\_max; //テストループ回数  int start\_i; //スレッド生成ループ開始値  };  //スレッド情報  struct THREAD\_INFO  {  int level; //子スレッドレベル  std::thread::id parent\_tid;//親スレッドID  bool is\_root\_thread; //大元のスレッド判定用フラグ  };  //スレッド受け渡し情報  struct THREAD\_PARAM  {  TEST\_INFO test; //テスト用情報  THREAD\_INFO thread;//スレッド情報  };  //子スレッドの生成と終了待ち  extern void createChildThreads(const char\* thread\_name, TEST\_INFO& test, THREAD\_INFO& tinfo);  //スレッド関数  void threadFunc(THREAD\_PARAM param)  {  //パラメータ受け取り  TEST\_INFO test;  THREAD\_INFO tinfo;  {  memcpy(&test, &param.test, sizeof(TEST\_INFO));  memcpy(&tinfo, &param.thread, sizeof(THREAD\_INFO));  }  //スレッド情報  std::thread::id tid\_tmp = std::this\_thread::get\_id();  //スレッドローカルストレージ(TLS)テスト  tls\_tid = tid\_tmp.hash();  //大元のスレッド判定用フラグをOFF  tinfo.is\_root\_thread = false;  //スレッド名を作成  ++tinfo.level;  char thread\_name[32];  sprintf\_s(thread\_name, sizeof(thread\_name), "C%d(0x%08x,0x%08x)", tinfo.level, tid\_tmp.hash(), tinfo.parent\_tid.hash());  // C + 子スレッドのレベル + ( 自スレッドID , 親スレッドID )  //子スレッド開始メッセージ  printf("START: %s\n", thread\_name);  fflush(stdout);  //テスト用変数を更新  ++test.value;  ++test.loop\_max;  //親スレッドIDを格納  tinfo.parent\_tid = tid\_tmp;  //1秒スリープ  std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));  //子スレッドの生成と終了待ち  createChildThreads(thread\_name, test, tinfo);  //子スレッド終了メッセージ  printf("END: %s\n", thread\_name);  fflush(stdout);  }  //子スレッドの生成と終了待ち  void createChildThreads(const char\* thread\_name, TEST\_INFO& test, THREAD\_INFO& tinfo)  {  //子スレッド生成ループ  const int CHILD\_THREAD\_NUM = 3;  std::thread\* thread[CHILD\_THREAD\_NUM];//生成した子スレッドのスレッドハンドル  std::thread::id tid[CHILD\_THREAD\_NUM];//生成した子スレッドのスレッドID  THREAD\_PARAM\* param[CHILD\_THREAD\_NUM];//スレッド受け渡し用データ  for (int i = 0; i < test.start\_i; ++i)  {  thread[i] = nullptr;  tid[i] = std::thread::id();  param[i] = NULL;  }  for (int i = test.start\_i; i < CHILD\_THREAD\_NUM; ++i)  {  //子スレッド用パラメータ作成  param[i] = (THREAD\_PARAM\*)malloc(sizeof(THREAD\_PARAM));  memcpy(&param[i]->test, &test, sizeof(TEST\_INFO));  memcpy(&param[i]->thread, &tinfo, sizeof(THREAD\_INFO));  param[i]->test.start\_i = i + 1;  //子スレッド生成  std::thread::id tid\_tmp;  std::thread\* thread\_tmp = new std::thread(threadFunc, \*param[i]);  if (thread\_tmp == nullptr)  {  //子スレッド生成失敗  free(param[i]);  param[i] = NULL;  //エラーメッセージ  if (tinfo.is\_root\_thread)  fprintf(stderr, "%s -> [%d] FAILED (errno=%d)\n", thread\_name, i, errno);  else  fprintf(stderr, "%s -> [%d] FAILED (errno=%d)\n", thread\_name, i, errno);  fflush(stderr);  //1秒スリープ  std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));  }  else  {  //子スレッド生成成功  //子スレッドのスレッドオブジェクトとIDを記録（最後の終了待ち用）  thread[i] = thread\_tmp;  tid\_tmp = thread\_tmp->get\_id();  tid[i] = tid\_tmp;  //子スレッド生成メッセージ  if (tinfo.is\_root\_thread)  printf("%s -> [%d] CREATED:C%d(0x%08x,0x%08x)\n", thread\_name, i, tinfo.level + 1, tid\_tmp.hash(), tinfo.parent\_tid.hash());  else  printf("%s -> [%d] CREATED:C%d(0x%08x,0x%08x)\n", thread\_name, i, tinfo.level + 1, tid\_tmp.hash(), tinfo.parent\_tid.hash());  fflush(stdout);  //2秒スリープ  std::this\_thread::sleep\_for(std::chrono::milliseconds(2000));  }  }  //ダミー処理  for (int i = 0; i < test.loop\_max; ++i)  {  //ダミーメッセージ表示  printf("[%s] ... Process(%d/%d): text=\"%s\",value=%d (tls\_tid=0x%08x)\n", thread\_name, i + 1, test.loop\_max, test.text, test.value, tls\_tid);  fflush(stdout);  test.value += 10;  //1秒スリープ  std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));  }  //子スレッドの終了待ち  for (int i = 0; i < CHILD\_THREAD\_NUM; ++i)  {  std::thread\* thread\_tmp = thread[i];  std::thread::id tid\_tmp = tid[i];  if (thread\_tmp)  {  //ウェイト開始メッセージ  printf("[%s] ... Wait(0x%08x) ...\n", thread\_name, tid\_tmp.hash());  fflush(stdout);  //ウェイト  thread\_tmp->join();    //子スレッドのオブジェクトを削除  delete thread\_tmp;  thread[i] = nullptr;  //メモリ解放  free(param[i]);  param[i] = NULL;  //ウェイト完了メッセージ  printf("[%s] ... Wait(0x%08x) End\n", thread\_name, tid\_tmp.hash());  fflush(stdout);  }  }  }  //C++11 thread テスト  int main(const int argc, const char\* argv[])  {  //プログラム実行開始  printf("- Start Thread Test -\n");  fflush(stdout);  //テスト用変数  TEST\_INFO test =  {  "TEST", //text  1000, //value  1, //loop\_max  0, //start\_i  };  //スレッド情報  THREAD\_INFO tinfo =  {  0, //level  std::this\_thread::get\_id(),  //parent\_tid  true,//is\_root\_thread  };  //スレッドローカルストレージ(TLS)テスト  tls\_tid = tinfo.parent\_tid.hash();  //スレッド名を作成  // P + 子スレッドのレベル + ( 自スレッドID , 親スレッドID )  char thread\_name[32];  sprintf\_s(thread\_name, sizeof(thread\_name), "P0(0x%08x,0x%08x)", tinfo.parent\_tid.hash(), 0);  //子スレッドの生成と終了待ち  createChildThreads(thread\_name, test, tinfo);  //プログラム実行終了  printf("- End Thread Test -\n");  fflush(stdout);  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| ※スレッドの親子関係は fork 版と同じなので着色は省略  - Start Thread Test -  START: C1(0xe7088d04,0x792b3321) ←Windows版と比較して、スレッドがすぐに開始していることがわかる  P0(0x792b3321,0x00000000) -> [0] CREATED:C1(0xe7088d04,0x792b3321)  START: C2(0x089c1d60,0xe7088d04)  C1(0xe7088d04,0x792b3321) -> [1] CREATED:C2(0x089c1d60,0xe7088d04)  START: C1(0xf8c2e2b0,0x792b3321)  P0(0x792b3321,0x00000000) -> [1] CREATED:C1(0xf8c2e2b0,0x792b3321)  START: C3(0xec799d21,0x089c1d60)  C2(0x089c1d60,0xe7088d04) -> [2] CREATED:C3(0xec799d21,0x089c1d60)  [C3(0xec799d21,0x089c1d60)] ... Process(1/4): text="TEST",value=1003 (tls\_tid=0xec799d21)  START: C2(0x19c48b6a,0xf8c2e2b0)  C1(0xf8c2e2b0,0x792b3321) -> [2] CREATED:C2(0x19c48b6a,0xf8c2e2b0)  START: C2(0xa2ce3a4e,0xe7088d04)  C1(0xe7088d04,0x792b3321) -> [2] CREATED:C2(0xa2ce3a4e,0xe7088d04)  START: C1(0x781ab907,0x792b3321)  [C3(0xec799d21,0x089c1d60)] ... Process(2/4): text="TEST",value=1013 (tls\_tid=0xec799d21)  [C2(0x089c1d60,0xe7088d04)] ... Process(1/3): text="TEST",value=1002 (tls\_tid=0x089c1d60)  P0(0x792b3321,0x00000000) -> [2] CREATED:C1(0x781ab907,0x792b3321)  [C2(0x19c48b6a,0xf8c2e2b0)] ... Process(1/3): text="TEST",value=1002 (tls\_tid=0x19c48b6a)  [C2(0xa2ce3a4e,0xe7088d04)] ... Process(1/3): text="TEST",value=1002 (tls\_tid=0xa2ce3a4e)  [C3(0xec799d21,0x089c1d60)] ... Process(3/4): text="TEST",value=1023 (tls\_tid=0xec799d21)  [C2(0x089c1d60,0xe7088d04)] ... Process(2/3): text="TEST",value=1012 (tls\_tid=0x089c1d60)  [C1(0x781ab907,0x792b3321)] ... Process(1/2): text="TEST",value=1001 (tls\_tid=0x781ab907)  [C2(0x19c48b6a,0xf8c2e2b0)] ... Process(2/3): text="TEST",value=1012 (tls\_tid=0x19c48b6a)  [C1(0xf8c2e2b0,0x792b3321)] ... Process(1/2): text="TEST",value=1001 (tls\_tid=0xf8c2e2b0)  [C1(0xe7088d04,0x792b3321)] ... Process(1/2): text="TEST",value=1001 (tls\_tid=0xe7088d04)  [C2(0xa2ce3a4e,0xe7088d04)] ... Process(2/3): text="TEST",value=1012 (tls\_tid=0xa2ce3a4e)  [C3(0xec799d21,0x089c1d60)] ... Process(4/4): text="TEST",value=1033 (tls\_tid=0xec799d21)  [P0(0x792b3321,0x00000000)] ... Process(1/1): text="TEST",value=1000 (tls\_tid=0x792b3321)  [C2(0x089c1d60,0xe7088d04)] ... Process(3/3): text="TEST",value=1022 (tls\_tid=0x089c1d60)  [C1(0x781ab907,0x792b3321)] ... Process(2/2): text="TEST",value=1011 (tls\_tid=0x781ab907)  [C2(0x19c48b6a,0xf8c2e2b0)] ... Process(3/3): text="TEST",value=1022 (tls\_tid=0x19c48b6a)  [C1(0xf8c2e2b0,0x792b3321)] ... Process(2/2): text="TEST",value=1011 (tls\_tid=0xf8c2e2b0)  [C1(0xe7088d04,0x792b3321)] ... Process(2/2): text="TEST",value=1011 (tls\_tid=0xe7088d04)  [C2(0xa2ce3a4e,0xe7088d04)] ... Process(3/3): text="TEST",value=1022 (tls\_tid=0xa2ce3a4e)  END: C3(0xec799d21,0x089c1d60)  [P0(0x792b3321,0x00000000)] ... Wait(0xe7088d04) ...  [C2(0x089c1d60,0xe7088d04)] ... Wait(0xec799d21) ...  [C2(0x089c1d60,0xe7088d04)] ... Wait(0xec799d21) End  END: C2(0x089c1d60,0xe7088d04)  END: C1(0x781ab907,0x792b3321)  END: C2(0x19c48b6a,0xf8c2e2b0)  [C1(0xf8c2e2b0,0x792b3321)] ... Wait(0x19c48b6a) ...  [C1(0xf8c2e2b0,0x792b3321)] ... Wait(0x19c48b6a) End  END: C1(0xf8c2e2b0,0x792b3321)  END: C2(0xa2ce3a4e,0xe7088d04)  [C1(0xe7088d04,0x792b3321)] ... Wait(0x089c1d60) ...  [C1(0xe7088d04,0x792b3321)] ... Wait(0x089c1d60) End  [C1(0xe7088d04,0x792b3321)] ... Wait(0xa2ce3a4e) ...  [C1(0xe7088d04,0x792b3321)] ... Wait(0xa2ce3a4e) End  END: C1(0xe7088d04,0x792b3321)  [P0(0x792b3321,0x00000000)] ... Wait(0xe7088d04) End  [P0(0x792b3321,0x00000000)] ... Wait(0xf8c2e2b0) ...  [P0(0x792b3321,0x00000000)] ... Wait(0xf8c2e2b0) End  [P0(0x792b3321,0x00000000)] ... Wait(0x781ab907) ...  [P0(0x792b3321,0x00000000)] ... Wait(0x781ab907) End  - End Thread Test - |

## OpenMP

OpenMPは、圧倒的に簡単にマルチスレッドの記述が可能である。マルチコアを活用した並列処理が書き易い。

常駐スレッドのようなスレッドを作るのにはあまり向いておらず、部分的な処理を分散処理で効率化することに向いている。

OpenMPは基本的に #pragam でスレッドを制御する。幾つかサンプルコードを示す。

OpneMPのサンプル①： Hello, World

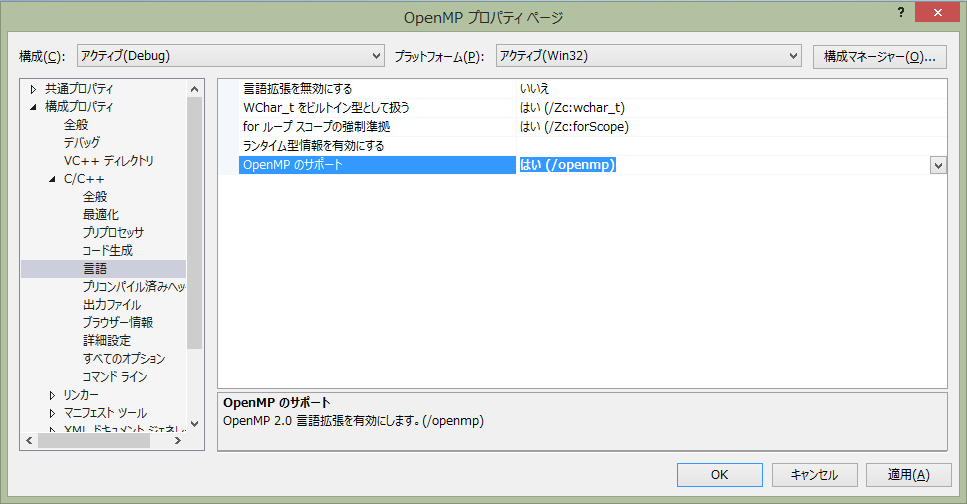
|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main(const int argc, const char\* argv[])  {  printf("[BEGIN]\n");  #pragma omp parallel  printf("Hello, OpenMP!\n");  printf("[END]\n");  return EXIT\_SUCCESS;  } |

↓（実行結果） ※OpenMP有効化前

|  |
| --- |
| [BEGIN]  Hello, OpenMP!  [END] |

下記のプロジェクト設定を変更し、OpneMPを有効化する。

Visual C++ 2013 の OpenMP有効化設定：



↓（実行結果） ※OpenMP有効化後

|  |
| --- |
| [BEGIN] ※１プロセッサ、４コア、HT有効 = ８論理プロセッサのPCで実行した結果  Hello, OpenMP!  Hello, OpenMP!  Hello, OpenMP!  Hello, OpenMP!  Hello, OpenMP!  Hello, OpenMP!  Hello, OpenMP!  Hello, OpenMP!  [END] |

OpenMPが無効の環境では、#pragma が無視されるだけなのでコンパイルエラーにはならない。

OpneMPのサンプル②： 論理プロセッサより少ないスレッド数を指定

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main(const int argc, const char\* argv[])  {  printf("[BEGIN]\n");  #pragma omp parallel num\_threads(3)  printf("Hello, OpenMP!\n");  printf("[END]\n");  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| [BEGIN]  Hello, OpenMP!  Hello, OpenMP!  Hello, OpenMP!  [END] |

OpneMPのサンプル③： 論理プロセッサより多いスレッド数を指定

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main(const int argc, const char\* argv[])  {  printf("[BEGIN]\n");  #pragma omp parallel num\_threads(15)  printf("Hello, OpenMP!\n");  printf("[END]\n");  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| [BEGIN]  Hello, OpenMP!  Hello, OpenMP!  Hello, OpenMP!  Hello, OpenMP!  Hello, OpenMP!  Hello, OpenMP!  Hello, OpenMP!  Hello, OpenMP!  Hello, OpenMP!  Hello, OpenMP!  Hello, OpenMP!  Hello, OpenMP!  Hello, OpenMP!  Hello, OpenMP!  Hello, OpenMP!  [END] |

OpneMPのサンプル④：処理ブロックを指定

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main(const int argc, const char\* argv[])  {  #pragma omp parallel num\_threads(3)  {  printf("[BEGIN]\n");  printf("Hello, OpenMP!\n");  printf("[END]\n");  }  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| [BEGIN]  Hello, OpenMP!  [END]  [BEGIN]  Hello, OpenMP!  [END]  [BEGIN]  Hello, OpenMP!  [END] |

OpneMPのサンプル⑤：OpenMPの関数を使用

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #ifdef \_OPENMP  #include <omp.h>  #endif  int main(const int argc, const char\* argv[])  {  #ifdef \_OPENMP  printf("omp\_get\_num\_procs()=%d\n", omp\_get\_num\_procs());  printf("omp\_get\_max\_threads()=%d\n", omp\_get\_max\_threads());  #endif  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| omp\_get\_num\_procs()=8  omp\_get\_max\_threads()=8 |

OpneMPのサンプル⑥： ワークシェアリング

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <Windows.h>  #ifdef \_OPENMP  #include <omp.h>  #endif  //素数判定  bool isPrime(const int n)  {  if (n < 2)  return false;  else if (n == 2)  return true;  else if (n % 2 == 0)  return false;  for (int i = 3; i <= n / i; i += 2)  {  if (n % i == 0)  return false;  }  return true;  }  //メイン関数  int main(const int argc, const char\* argv[])  {  //CPUスペック表示  #ifdef \_OPENMP  printf("omp\_get\_num\_procs()=%d\n", omp\_get\_num\_procs());  printf("omp\_get\_max\_threads()=%d\n", omp\_get\_max\_threads());  #endif    //処理時間計測準備  LARGE\_INTEGER freq;  QueryPerformanceFrequency(&freq);    //処理時間計測（開始）  LARGE\_INTEGER begin;  QueryPerformanceCounter(&begin);    //集計  int prime\_num= 0;  long long prime\_total = 0;  long long sum = 0;  {  int i, j, k;  //三重ループで集計  #pragma omp parallel for reduction(+:sum) private(j, k)  for (i = 0; i < 2000; ++i)  {  sum += i;  for (j = 0; j < 2000; ++j)  {  sum += j;  for (k = 0; k < 2000; ++k)  {  sum += k;  }  }  }  //素数を数える  #pragma omp parallel  {  static const int PRIME\_MAX = 200000;  static const int PRIME\_NUM\_MAX = 20000;  #pragma omp for  for (i = 0; i < PRIME\_MAX; ++i)  {  if (prime\_num < PRIME\_NUM\_MAX && isPrime(i))  {  #pragma omp atomic  ++prime\_num;  #pragma omp atomic  prime\_total += i;  }  }  }  }  //処理時間計測（終了）  LARGE\_INTEGER end;  QueryPerformanceCounter(&end);  float time = static\_cast<float>(static\_cast<double>(end.QuadPart - begin.QuadPart)  / static\_cast<double>(freq.QuadPart));    //結果表示  printf("prime\_num=%d, prime\_total=%lld, sum=%lld, time=%.6f sec\n", prime\_num, prime\_total, sum, time);    return EXIT\_SUCCESS;  } |

↓（実行結果）※OpneMP無効化時

|  |
| --- |
| prime\_num=17984, prime\_total=1709600813, sum=7999999999000, time=7.192203 sec |

↓（実行結果）※OpneMP有効化時

|  |
| --- |
| omp\_get\_num\_procs()=8  omp\_get\_max\_threads()=8  prime\_num=17984, prime\_total=1709600813, sum=7999999999000, time=1.870132 sec |

細かい解説は省略するが、処理によって効果が大きいことは確認できる。しかし、制約も多く、思い通りの制御は難しい。

慣れないと正確な指定がしにくい。例えば、「#pragma omp parallel for」の「for」が抜けても動作するが、誤った計算結果になる。ほか、「#pragma omp parallel private(var)」の「private(var)」でブロック内のローカル変数を指定しないと正しく変数の値が扱われない、といった面倒が多い。

OpenMP有効化時と無効化時を切り替えて検算し、ロジックに誤りがないことを確認することが必須となる。

スレッドの制御は通常のスレッドと同様に、OSによってスケジューリングされる。スタックサイズは直接指定できず、環境変数によって設定する。

ほか、ロック機構なども用意されている。

## C++11版非同期関数

OpenMPのような分散処理の手法が、C++11にも用意されている。

OpenMPに比べて、明示的な手続きで分散するので、処理の把握がし易い。

C++11の非同期関数のサンプルコードを示す。

ややコードが長めになるが、応用事例もサンプルに含める。1千万件のデータのソートのサンプルである。ラムダ式を用いてコードをコンパクトにできる点もC++11の特徴。

C++11非同期関数のサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <future>//非同期関数用  #include <thread>//スリープ用  #include <chrono>//時間計測用  #include <vector>//テスト計算用  #include <algorithm>//テスト計算用  //計算用関数A  int calcSubA()  {  //時間のかかる処理代わりのスリープ  std::this\_thread::sleep\_for(std::chrono::milliseconds(10));  return 1;  }  //計算用関数B  int calcSubB()  {  //時間のかかる処理代わりのスリープ  std::this\_thread::sleep\_for(std::chrono::milliseconds(10));  return 2;  }  //計算用関数C  int calcSubC()  {  //時間のかかる処理代わりのスリープ  std::this\_thread::sleep\_for(std::chrono::milliseconds(10));  return 3;  }  //通常関数処理  void testNormalCalc()  {  //時間計測開始  auto begin = std::chrono::high\_resolution\_clock::now();    //処理  int result = 0;  static const int TEST\_TIMES = 100;  for (int i = 0; i < TEST\_TIMES; ++i)  {  result += (calcSubA() + calcSubB() + calcSubC());  }    //時間計測終了  auto end = std::chrono::high\_resolution\_clock::now();  auto duration = static\_cast<float>(static\_cast<double>(std::chrono::duration\_cast<std::chrono::microseconds>(end - begin).count()) / 1000000.);    //結果表示  printf("----------------------------------------\n");  printf("subNormalCalc: result = %d (%.6f sec)\n", result, duration);  }  //非同期関数処理  void testAsyncCalc()  {  //時間計測開始  auto begin = std::chrono::high\_resolution\_clock::now();  //処理  int result = 0;  static const int TEST\_TIMES = 100;  for (int i = 0; i < TEST\_TIMES; ++i)  {  //未来(future)に結果が返る関数を非同期で実行  std::future<int> r1 = std::async(calcSubA),  r2 = std::async(calcSubB),  r3 = std::async(calcSubC);    //.get()メソッドは、処理の完了を待って結果を返す  result += (r1.get() + r2.get() + r3.get());  }    //時間計測終了  auto end = std::chrono::high\_resolution\_clock::now();  auto duration = static\_cast<float>(static\_cast<double>(std::chrono::duration\_cast<std::chrono::microseconds>(end - begin).count()) / 1000000.);  //結果表示  printf("----------------------------------------\n");  printf("subAsyncCalc: result = %d (%.6f sec)\n", result, duration);  }  //非同期関数【応用】  void testAsyncCalcEx()  {  //リスト作成  static const int LIST\_MAX = 10000000;  std::vector<int> list\_org;  list\_org.reserve(LIST\_MAX);  for (int i = 0; i < LIST\_MAX; ++i)  {  list\_org.push\_back(i);  }  //リストシャッフル  std::random\_shuffle(list\_org.begin(), list\_org.end());    //この時点のリストを表示  printf("----------------------------------------\n");  printf("subAsyncCalcEx: before\n");  printf("list =");  std::for\_each(list\_org.begin(), list\_org.begin() + 10, [](int val){printf(" %d", val); });  printf(" ... ");  std::for\_each(list\_org.end() - 10, list\_org.end(), [](int val){printf(" %d", val); });  printf("\n");  //通常ソート  {  //時間計測開始  auto begin = std::chrono::high\_resolution\_clock::now();    //処理準備：シャッフル済みリストをコピー  std::vector<int> list = list\_org;    //クイックソート  std::sort(list.begin(), list.end());    //時間計測終了  auto end = std::chrono::high\_resolution\_clock::now();  auto duration = static\_cast<float>(static\_cast<double>(std::chrono::duration\_cast<std::chrono::microseconds>(end - begin).count()) / 1000000.);    //結果表示  printf("----------------------------------------\n");  printf("subAsyncCalcEx: normal-sort: result (%.6f sec):\n", duration);  printf("list =");  std::for\_each(list.begin(), list.begin() + 10, [](int val){printf(" %d", val);});  printf(" ... ");  std::for\_each(list.end() - 10, list.end(), [](int val){printf(" %d", val); });  printf("\n");  }    //非同期関数でソート  //※大量にバッファを消費する。また、アルゴリズムを高速化できてもバッファコピーに時間がかかる。  {  //時間計測開始  auto begin = std::chrono::high\_resolution\_clock::now();  //処理準備：シャッフル済みリストをコピー  std::vector<int> list = list\_org;  const int list\_size = list.size();    //4分割で並列クイックソート  std::future<void> r\_a[4];  {  //非同期処理には、関数ポインタ以外にも、関数オブジェクトやラムダ式を渡すことができる  auto sort\_lambda = [&list](const int index, const int list\_begin, const int list\_end)  {  std::vector<int>::iterator ite\_begin = list.begin() + list\_begin;  std::vector<int>::iterator ite\_end = list.begin() + list\_end;  std::sort(ite\_begin, ite\_end);  };  r\_a[0] = std::async(sort\_lambda, 0, list\_size / 4 \* 0, list\_size / 4 \* 1);  r\_a[1] = std::async(sort\_lambda, 1, list\_size / 4 \* 1, list\_size / 4 \* 2);  r\_a[2] = std::async(sort\_lambda, 2, list\_size / 4 \* 2, list\_size / 4 \* 3);  r\_a[3] = std::async(sort\_lambda, 3, list\_size / 4 \* 3, list\_size);  }    //2分割で並列マージソート  std::future<void> r\_b[2];  {  //非同期処理には、関数ポインタ以外にも、関数オブジェクトやラムダ式を渡すことができる  auto merge\_sort\_lambda = [&r\_a, &list](const int index, const int list\_begin, const int list\_mid, const int list\_end)  {  r\_a[index \* 2 + 0].wait();  r\_a[index \* 2 + 1].wait();  std::vector<int>::iterator ite\_begin = list.begin() + list\_begin;  std::vector<int>::iterator ite\_mid = list.begin() + list\_mid;  std::vector<int>::iterator ite\_end = list.begin() + list\_end;  std::inplace\_merge(ite\_begin, ite\_mid, ite\_end);//※二つのリストからマージソート  //　する場合には std::merge() が使える  };  r\_b[0] = std::async(merge\_sort\_lambda, 0, list\_size / 2 \* 0, list\_size / 2 \* 0 + list\_size / 4,  list\_size / 2 \* 1);  r\_b[1] = std::async(merge\_sort\_lambda, 1, list\_size / 2 \* 1, list\_size / 2 \* 1 + list\_size / 4,  list\_size);  }    //マージソート（完了）  {  r\_b[0].wait();  r\_b[1].wait();  std::vector<int>::iterator ite\_begin = list.begin();  std::vector<int>::iterator ite\_mid = list.begin() + list\_size / 2;  std::vector<int>::iterator ite\_end = list.end();  std::inplace\_merge(ite\_begin, ite\_mid, ite\_end);//※二つのリストからマージソート  //　する場合には std::merge() が使える  }  //時間計測終了  auto end = std::chrono::high\_resolution\_clock::now();  auto duration = static\_cast<float>(static\_cast<double>(std::chrono::duration\_cast<std::chrono::microseconds>(end - begin).count()) / 1000000.);    //結果表示  printf("----------------------------------------\n");  printf("subAsyncCalcEx: async-sort: result (%.6f sec):\n", duration);  printf("list =");  std::for\_each(list.begin(), list.begin() + 10, [](int val){printf(" %d", val); });  printf(" ... ");  std::for\_each(list.end() - 10, list.end(), [](int val){printf(" %d", val); });  printf("\n");  }  }  //テスト  int main(const int argc, const char\* argv[])  {  //通常関数処理  testNormalCalc();    //非同期関数処理  testAsyncCalc();  //非同期関数応用  testAsyncCalcEx();  //終了  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| ----------------------------------------  subNormalCalc: result = 600 (3.300186 sec)  ----------------------------------------  subAsyncCalc: result = 600 (1.005057 sec)  ----------------------------------------  subAsyncCalcEx: before  list = 5288787 5235665 796859 1253125 4931978 5458993 4375385 5810937 5295160 844757 ... 7573541 8996297 9161292 7190808 1950095 9319072 7664002 7360610 8990105 2669525  ----------------------------------------  subAsyncCalcEx: normal-sort: result (1.193068 sec):  list = 0 1 2 3 4 5 6 7 8 9 ... 9999990 9999991 9999992 9999993 9999994 9999995 9999996 9999997 9999998 9999999  ----------------------------------------  subAsyncCalcEx: async-sort: result (0.431025 sec):  list = 0 1 2 3 4 5 6 7 8 9 ... 9999990 9999991 9999992 9999993 9999994 9999995 9999996 9999997 9999998 9999999 |

なお、このサンプルプログラムに含めた並列ソート処理は、MapReduceというプログラミングモデルを参考にしている。MapReduceについては、本書の最後に簡単な説明を添えている。

## ファイバースレッド／コルーチン

「ファイバースレッド」はノンプリエンプティブなマルチタスクの仕組みである。

複数のファイバースレッドを同時に稼働させることができるが、アクティブなスレッドの切り替えは明示的に行う必要がある。並列処理は不可能。OSの負担がほとんどかからず軽量なので、使いどころによっては効率的。通常のスレッドと同様に、それぞれのファイバースレッドがスタック領域を持つ。

ファイバースレッドは「コルーチン」とも言われる。

コルーチンは、LuaやSqirrelなどの汎用スクリプト言語や、C#などに実装されている。

汎用スクリプト言語のコルーチンは、ゲームでもよく用いられる。例えば、キャラクターのAIをコルーチン化することで、シーンに登場するキャラ一それぞれが並行してAI処理を実行する。処理の途中でサスペンドすることで処理を中断し、次のフレームでサスペンドした位置から再開する。この「サスペンド」によって、他のコルーチンやメインループに制御を切り替える。

ファイバースレッドの使いどころとして、「並列処理の待ち受け」というものがある。

外部プロセッサでプログラムを動作させた際に、その完了をファイバースレッドで待ち受ける手法である。

これにより、外部プログラムを透過的に自身のスレッドに見立てることができる。外部プログラム完了後の処理を行うが、ほとんどは単なる待ち受けであるため、普通のスレッドではなくファイバースレッドを用いることで、OSやリソース消費の負担を軽減する。

スタック領域の割り当てなど、多少のリソース消費は生じるが、コールバック処理にするよりも安全で、一括処理にするよりも局所化された柔軟なプログラム構造にできる。

この手法は、PS3のSPUの処理待ちによく用いられている。ほか、GPGPUやグリッド・コンピューティングなどへの利用が考えられる。

ファイバースレッドの活用例：



WIN32APIによるファイバースレッドのサンプルプログラムを示す。

WIN32API版ファイバースレッドのサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <Windows.h>  #include <Process.h>  //スレッドローカルストレージ(TLS)  #define \_TLS \_\_declspec(thread)//Visual C++固有版TLS指定  //#define \_TLS thread\_local //C++11仕様版TLS指定：Visual C++ 2013 では未対応  //ファイバースレッド管理情報  static const int FIBER\_MAX = 3; //ファイバースレッドの最大数  \_TLS LPVOID s\_pFiler[FIBER\_MAX] = {};//ファイバースレッドのインスタンスのポインタ  \_TLS int s\_fiberNum = 0; //稼働中のファイバースレッド数  \_TLS int s\_fiberNow = 0; //現在のファイバースレッド  //ファイバースレッドが存在するか？  //※管理用のスレッドは数に含めない  bool isExistFiberThread()  {  return s\_fiberNum > 1;  }  //ファイバースレッド切り替え  void switchNextFiber()  {  if (s\_fiberNum == 0)  {  fprintf(stdout, "Fiber thread is nothing.\n");  return;  }  s\_fiberNow = (s\_fiberNow + 1) % s\_fiberNum;  fprintf(stdout, "Fiber thread No.%d is switched.\n", s\_fiberNow);  SwitchToFiber(s\_pFiler[s\_fiberNow]);  }  //ファイバースレッドを追加  template<typename T>  bool addFiber(LPFIBER\_START\_ROUTINE func, const size\_t stack\_size, T param\_p)  {  if (s\_fiberNum >= FIBER\_MAX)  {  fprintf(stderr, "Fiber thread was not more added.\n");  return false;  }  fprintf(stdout, "Fiber thread No.%d was added.\n", s\_fiberNum);  s\_pFiler[s\_fiberNum++] = CreateFiber(stack\_size, func, (LPVOID)param\_p);  return true;  }  //現在のファイバースレッドを削除  void removeCurrentFiber()  {  for (int i = s\_fiberNow; i < s\_fiberNum - 1; ++i)  {  s\_pFiler[i] = s\_pFiler[i + 1];  }  --s\_fiberNum;  fprintf(stdout, "Fiber thread No.%d was removed.\n", s\_fiberNow);  if (s\_fiberNum == 0)  {  s\_fiberNow = 0;  }  else  {  s\_fiberNow = (s\_fiberNum + s\_fiberNow - 1) % s\_fiberNum;  switchNextFiber();//自身を削除したらすぐに次のファイバースレッドを実行する  }  }  //ファイバースレッド関数  void WINAPI fiberFunc(LPVOID param\_p)  {  //パラメータ受け取り  const char\* fiber\_name = reinterpret\_cast<const char\*>(param\_p);  //テスト処理  for (int i = 0; i < 3; ++i)  {  printf("fiberFunc(): [%s](%d)\n", fiber\_name, i);    switchNextFiber();//次のスレッドに制御を移す  }  //最後に自身を削除する（同時に次のスレッドに制御を移す）  removeCurrentFiber();    //※関数を return すると、その時点でスレッド（fiberMain）が  //　終了するので注意！  //　（ファイバースレッドの切り替えは、関数呼び出しと違って  //　単純なプログラムカウンタの切り替えなので、return すると、  //　fiberMain() の return とみなされてしまう）  }  //ファイバースレッド管理用スレッド　※必須  unsigned int WINAPI fiberMain(void\* param\_p)  {  //スレッドをファイバースレッドとして登録  //※他のファイバースレッドと処理を切り替える必要があるため  //　管理スレッドもファイバースレッドの一つにする。  //※main() から見たら、普通のスレッドであることに変わりはない。  s\_pFiler[s\_fiberNum++] = ConvertThreadToFiber(NULL);    //ファイバースレッドを追加  addFiber(fiberFunc, 1024, "太郎");  addFiber(fiberFunc, 1024, "次郎");    //ファイバースレッドを切り替えながら、全て完了するのを待つ  while (isExistFiberThread())  {  switchNextFiber();//ファイバースレッドの切り替え  fprintf(stdout, "- loop -\n");  Sleep(0);//他のスレッドに切り替え  }    //ファイバースレッドを削除  removeCurrentFiber();    return 0;  }  //メイン  int main(const int argc, const char\* argv[])  {  fprintf(stdout, "- Fiber Thread Test: BEGIN -\n");  //ファイバースレッド管理用のスレッドを生成  unsigned int tid = 0;  HANDLE hThread = (HANDLE)\_beginthreadex(NULL, 0, fiberMain, nullptr, 0, &tid);  //ファイバースレッド管理スレッド終了待ち  WaitForSingleObject(hThread, INFINITE);    fprintf(stdout, "- Fiber Thread Test: END -\n");  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - Fiber Thread Test: BEGIN -  Fiber thread No.1 was added.  Fiber thread No.2 was added.  - loop: start - fiberMainでファイバースレッドを切り替えると、連鎖的に全てに切り替わる  Fiber thread No.1 is switched.  fiberFunc(): [太郎](0)  Fiber thread No.2 is switched.  fiberFunc(): [次郎](0)  Fiber thread No.0 is switched.  - loop - fiberMainでファイバースレッドを切り替えると、連鎖的に全てに切り替わる  Fiber thread No.1 is switched.  fiberFunc(): [太郎](1)  Fiber thread No.2 is switched.  fiberFunc(): [次郎](1)  Fiber thread No.0 is switched.  - loop - fiberMainでファイバースレッドを切り替えると、連鎖的に全てに切り替わる  Fiber thread No.1 is switched.  fiberFunc(): [太郎](2)  Fiber thread No.2 is switched.  fiberFunc(): [次郎](2)  Fiber thread No.0 is switched.  - loop - fiberMainでファイバースレッドを切り替えると、連鎖的に全てに切り替わる  Fiber thread No.1 is switched.  Fiber thread No.1 was removed. ←ファイバースレッド終了  Fiber thread No.1 is switched.  Fiber thread No.1 was removed. ←ファイバースレッド終了  Fiber thread No.0 is switched.  - loop -  - loop: end -  Fiber thread No.0 was removed. ←fiberMain終了  - Fiber Thread Test: END - |

## SPURS／GPGPU／DirectCompute／CUDA／ATI Stream／OpenCL

より大規模な並列処理技術を簡単に解説する。

標題に記したのは、いずれもメインプロセッサ（コア）以外のプロセッサを使用して並列処理を行う為のフレームワークである。プロセッサ独自の処理コードが必要なため、「プログラム中の一部の関数をスレッド化する」といったことはできない。メモリもメインメモリから独立しているものが多く、専用のメモリ管理が必要になる。

SPURS PS3用のSDKで提供されているSPUのジョブタスクスケジューラ。稼働中のタスクだけではなく、予約待機状態のタスクも管理する。個々のタスクを「ジョブ」と呼ぶ。「Spurs Engine」と呼ばれる演算装置とは別物。SPUは独自のメモリ領域を持ち、専用のプログラムを必要とする。SPURSはプログラムとデータの転送、プログラムの実行をまかなう。

ちなみに、「Spurs Engine」はSPEを4基とエンコーダー／デコーダー処理チップ搭載してPPEは搭載しないコプロセッサ。

GPGPU 「General-Processing computing on Graphincs Processing Unit：GPUによる汎目的演算」のこと。描画用の処理装置を他の演算に活用する。浮動小数点演算に強く、プログラミングにはコンピュートシェーダーを用いる。なお、「GPGPU」自体は技術的概念を指す言葉であって、特定のフレームワークではない。

DirectCompute Microsoft によるGPGPU対応API。DirectX11以降に含まれる。

CUDA NVIDIAによるGPU向けのC言語の統合開発環境。NVIDIA製のGPUに特化したGPGPUのためのフレームワークである。「CUDA」は「Compute Unified Devide Architecture」の略。「クーダ」と読む。

ATI Stream AMDによるGPGPU技術。開発フレームワークには「OpenCL」を用いる。合併前の社名の「ATI」がその名に残る。

OpenCL 汎用的な並列コンピューティングの開発フレームワーク。GPGPU、Cell（SPU）、CPU（マルチプロセッサ／マルチコア）などの異種演算装置をひとまとめに扱う。

「SPURS」「DirectCompute」「CUDA」「OpenCL」が競合技術。

なお、「CUDA」「OpenCL」のサンプルプログラムを記載したいところだったが、本書作成時点で間に合わず。

## クライアント・サーバー／クラウド／グリッド／RPC／ORB／Hadoop

さらに大規模な並列処理技術を簡単に解説する。

標題に記したのは、いずれもネットワークを介して他のコンピュータを使用することで並列処理を行うための技術・概念・フレームワークである。

クライアント・サーバー 通信処理モデルの一形態のこと。1台のサーバーに複数のクライアントが接続して、情報の共有を行うためのシステム。

クライアント側では、サーバーのレスポンスを待たずに画面操作を受け付けるなど、並列処理を行う。

データベースサーバーやWebサーバーなど、「サーバー」と呼ばれるものの多くはこの形態。なお、ソフトウェア的な概念なので、（主に開発中に）同一PC上でサーバーとクライアントの両方を動かしてもよい。

余談になるが、「クライアント・サーバー通信」に対して、情報を一か所に集めず、各機器同士が直接情報を渡し合うのを「ピア・ツー・ピア通信」と呼ぶ。また、ゲーム機でおなじみの「アドホック通信」はもっとローレベルな通信のことで、互いの機器を認識して直接の通信を確立するための技術を指す。「アドホック通信」では周囲の機器としか通信できないため、必然的に「ピア・ツー・ピア通信」を行うことになる。

クラウド クラウド・コンピューティング。

インターネットを介して特定のサーバーシステムを提供するサービスの一形態。

オンラインストレージやグループウェア、ブログ、サーバーレンタルなどの広範囲なサービスを含む抽象的な用語。オンラインゲームもクラウドの一種と言える。

「クラウド」という用語は、これらのシステムの技術や形態を表すのではなく、これらのシステムをサービスとして広範囲に提供する業務形態を意味する。

細かくサービスを分類すると、ASP、SaaS、PaaS、IaaSなどがある。（説明は省略）

グリッド グリッド・コンピューティング。

他の機器の演算装置（演算資源）を利用して、大規模な並列処理を行う技術的概念。

RPC Remote Procedure Call（リモート・プロシージャ・コール）。

プログラミングの技術的概念の一つで、他のPC上の関数を呼び出す手法である。クラウド・サービスには、ユーザーが使えるAPIを提供しているものも多い。会社のファイヤーウォールを超えてAPIを利用できるように、「SOAP」などのhttp通信ベースのRPC技術が確立されている。

ORB Object Request Broker（オブジェクト・リクエスト・ブローカー）。

オブジェクトベースのRPC。サーバー側とクライアント側のそれぞれに透過的に一対のオブジェクトを配置して通信する技術。CORBA（コルバ）やJaveEEのEJBが有名。（説明は省略）

Hadoop Apache Hadoop（アパッチ・ハドゥープ）。

大規模分散処理のためのJavaフレームワーク。グリッド・コンピューティングのための汎用フレームワークとして特に有名。

このような分散技術は、ゲームプログラマーにとっても身近なところで活用されている。分散コンパイルや分散レンダリングなどは、グリッド・コンピューティングの一種である。

また、「Cellコンピューティング」の構想のもと、世界中のPS3を使ってタンパク質解析などの演算を行う「Life with Playstation」という、グリッド・コンピューティング的な取り組みもあった。現在終了しているとのこと。

## 割り込み／システムコール

原点に立ち戻り、「マルチスレッド」や「マルチプロセス」以前の並行処理について説明する。

「割り込み」は、昔からゲームプログラマーにもなじみ深いものであり、今でも利用されている。そのため、「使われている割り込み処理」と、「割り込み処理の中でどんなことが行われているか」は、マルチスレッドプログラミングを行う上でも、注意しなければならない。

まず、具体的な割り込みの例を説明する。

### 割り込みの具体例：垂直同期割り込み（V-SYNC割り込み）

テレビに画像を表示する際、左上から順に一点ずつ高速に点を打っていき、右下まで打ち終わったらまだ左上に戻る。

この「戻る時」が「垂直同期」と呼ばれる描画が行われないタイミングである。

この時、ゲーム機はハードウェア的に反応して「垂直同期割り込み」と呼ばれる割り込みを発生させる。（垂直同期は「V-SYNC」とも呼ばれる）

「割り込み」が発生すると、あらかじめ登録されたアドレスのプログラムを呼び出す。この割り込み処理で画面のフリップを行うことで、ゲームはティアリングの起こらない映像描画を実現する。

このタイミング以外でフリップすると、「画面の上部が前のフレームの画像で下部が次のフレームの画像」という二つの画像が混ざった「ティアリング」という現象が起こる。

#### 【余談】垂直同期割り込みの失敗 ～ 処理落ちとティアリング対策

余談になるが、処理落ちした時は、割り込み処理でフリップを実行せず、処理が完了したタイミングでフリップするためティアリングが起こる。

まるまる1フレームフリップを遅らせればティアリングは防げるが、その完了を待つため、処理が大きく遅れてしまう。

この処理落ちの対策として「トリプルバッファ」を用いる。

「ダブルバッファ」の場合、「現在表示中の画像」と「現在処理中（描画中）の画像」の二つを用いて交互にフリップする。

トリプルバッファでは、その二つの画像の間に「フリップ待ちの画像」を入れる。これにより、「垂直同期割り込み」に処理が間に合わなかった場合でも、フリップ待ちの画像として予約さえしておけば、すぐに次の処理（描画）を進めることができる。また、処理が間に合っている限りは、ダブルバッファと同様に即座にフリップしていけば、描画が常時遅延するようなことにもならい。

トリプルバッファの問題点としては、より多くのグラフィックメモリを必要とすることである。しかし、ティアリングはかなり見苦しいので、グラフィックメモリに余裕があるならトリプルバッファは検討した方がよい。

### 割り込みとコンテキストスイッチ、スタック領域の配慮

割り込みはとても乱暴な処理である。

メインスレッドや他のスレッドがどんな状態であっても、最優先で有無をいわさず処理が実行される。

むしろ、それぐらい乱暴でなければ、フリップ処理のような「タイミングが命」の処理を信頼して実行することができない。

割り込み時の「コンテキストスイッチ」では、コンテキスト（CPUのレジスタなど）の退避を、その時割り込んだスレッドのスタック領域に行う。

そのため、スレッドを設計する際は、そのスレッドの稼働中に割り込みが発生しても問題が起こらないように、スタック領域のサイズに余裕を持たせるなどの配慮が必要になる。これは、ファイバースレッドに対しても同様である。

「どの程度の余裕が必要か？」については、「どのような割り込み処理が実装されているか」による。どの割り込み処理が起こっても十分なスタック領域の余裕が必要である。

また、スレッドによっては割り込み禁止処理なども用いる必要がある。

### 割り込みハンドラ／割り込みサブルーチン（ISR）

割り込みで実行される処理は、「割り込みハンドラ」や「割り込みサブルーチン」と呼ぶ。割り込みサブルーチンの略語として「ISR」（Interrupt Service Routine）という用語もよく用いられる。

本書では、以降「割り込みハンドラ」で統一する。

OSに対して、特定の割り込みに対する割り込みハンドラを登録しておくと、その後条件が揃った時に割り込みハンドラが実行されるようになる。

実際には、「OSに登録を要求する」というより、「割り込みベクタ」と呼ばれるコールバックテーブルに処理のアドレスを登録しておくだけである。

### 割り込みハンドラの処理

割り込みハンドラの処理は、スレッドセーフではないことに留意しなければならない。

大きなローカル変数の使用やメモリ確保などは行ってはならず、printfなどの出力も避け、関数呼び出しのようなスタックを消費する要素もできるだけ排除すべきである。また、極めて短時間で処理を復帰させる必要がある。

割り込み処理は本当に必要に迫られたときにだけ使用し、どうしても使う場合は、「フラグを立ててタイミングを通知するだけ」にとどめ、他のスレッドが（若干遅延するが）そのフラグを見て処理を行うといった設計にしたほうがよい。

画面フリップのような、タイミングが重要な処理はそのまま割り込み処理内で実行する。

なお、割り込み処理中に同じ割り込みの条件が揃っても、割り込みは発生せずに無視される。

### ハードウェア割り込みとソフトウェア割り込み／イベントコールバック

ここまで、割り込み処理に対する厳しい制約を示したが、基本的には「ハードウェア割り込み」と呼ばれる割り込みのことを示している。ハードウェア割り込みは、特に神経質に扱う必要がある。

ハードウェア割り込みに対して「ソフトウェア割り込み」（SWI = SoftWare Interrupt）というものもある。

ソフトウェア割り込みは基本的にプロセスで保護されるため、安全性が高い。「例外」の発生や「ブレークポイント」、［Ctrl］＋［C］キーでプロセスに終了命令を出すのもソフトウェア割り込みである。

なお、VBやC#、MFCなど、多くの開発ツールでは、ボタンクリックやキー入力などに応じた「イベントコールバック」を定義できる。

これも割り込みの一種のようなものであるが、基本的に各アプリケーションで制御され、特にコンテキストの切り替えを行うこともなく、逐次処理される。

### シグナル

OSが用意するプロセス管理のためのソフトウェア割り込みが「シグナル」である。

Unix系OSに搭載されている仕組みである。シグナルについてはサンプルプログラムを含めたもう少し詳しい説明を後述する。

なお、Windowsでは、シグナルのように、プロセスの外から何らかの要求を行う際は、ウインドウメッセージを用いる。

### タイマー割り込み

タイマー割り込みは一定時間ごとにイベントハンドラを呼び出す割り込みである。

OSはハードウェアタイマー割り込みを使うことで、タスクのタイムスライスを実現している。

Unix系OSでは、一定間隔で「アラーム」というシグナルを発行するタイマー割り込みを用意している。

また、多くの開発ツールでは、イベントコールバックの形でソフトウェアタイマー割り込みに対応している。

### システムコール

「シグナル」のようなソフトウェア割り込みはOSが管理しているため、イベントハンドラの登録には、前述の「システムコール」を用いる。

# マルチスレッドで起こり得る問題①

マルチスレッドプログラミングは難しい。

安易な制御では結果が破損し、結果を保護しようとすればパフォーマンスが劣化する。

スレッドを作ること自体はとても簡単ではあるが、各スレッドが情報をどのように保護し、共有し、かつ、最大限のパフォーマンスを引き出すか、という点を適切に設計するのは難しい。

以降、マルチスレッドプログラミングで実際に起こりえる問題とその対策を説明していく。

## データ破損（不完全なアトミック操作）

スレッドの処理が、データを読み込んでから書き込むまでの間に他のスレッドの処理が入り込むと、データ破損が起こる。

データ破損の例：



### 解決策

この問題は、一方が処理を始めた時点で、他のスレッドが処理できないように「ロック」することで解消できる。

データ破損の改善例：



このように、読み込み～書き込みの過程で割り込まれては困る一続きの操作（処理）を、「アトミック操作」もしくは「不可分操作」と呼ぶ。

一旦読み出して書き戻すような操作ではなく、「a += 10」のような一回の式で済ませればロックする必要がないかと思えばそうでもない。「a += 10」のような処理では、機械語レベルでは「レジスタ = a → レジスタ += 10 → a = レジスタ」という処理に展開されるので、途中で割り込まれる事は十分に起こり得る。

こうした問題はタイミングによって生じる問題である。

シングルコア環境では中途半端なタイミングでコンテキスト切り替えが発生しない限りは問題が表面化しないため、潜在的な問題が埋もれている可能性がある。

マルチコア環境ではあらゆるタイミングで並列動作するので、長らく大丈夫だったはずの処理でも問題を起こす可能性がある。

### 同期処理

データ破損を防ぐ「同期」処理には様々な方法がある。

アトミック操作を保証するための「排他制御」（ロック）や、一方の準備ができるまで監視しながら待機する「モニター」など、様々な種類とそれぞれの問題がある。詳しくは後述する。

## スレッド間の情報共有の失敗①：コンパイラの最適化による問題

マルチスレッドプログラミングでは、他のスレッドが更新した情報を確実に扱えるように気にかける必要があるが、「そうしているつもりなのにうまくいかない」という問題が起こることがある。具体的な事例を示す。

スレッド間の情報共有に失敗するサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <Windows.h>  #include <Process.h>  //処理終了フラグ  bool s\_isFinished = false;  //スレッド１  unsigned int WINAPI func1(void\* param\_p)  {  //開始  printf("- func1(): start -\n");  fflush(stdout);  //処理...  printf("func1(): process ...\n");  fflush(stdout);  Sleep(1000);  //処理終了通知  s\_isFinished = true;  //終了  printf("- func1(): end -\n");  fflush(stdout);  return 0;  }  //スレッド２  unsigned int WINAPI func2(void\* param\_p)  {  //開始  printf("- func2(): start -\n");  fflush(stdout);  //処理終了待ち  printf("func2(): wait ...\n");  fflush(stdout);  while (!s\_isFinished){}  printf("func2(): ok!\n");  fflush(stdout);  //処理...  printf("func2(): process ...\n");  fflush(stdout);  Sleep(1000);  //終了  printf("- func2(): end -\n");  fflush(stdout);  return 0;  }  //テスト  int main(const int argc, const char\* argv[])  {  //開始  printf("- main(): start -\n");  fflush(stdout);  //スレッド開始  HANDLE hThread[2] =  {  (HANDLE)\_beginthreadex(nullptr, 0, func1, nullptr, 0, nullptr),  (HANDLE)\_beginthreadex(nullptr, 0, func2, nullptr, 0, nullptr)  };  //スレッド終了待ち  WaitForMultipleObjects(2, hThread, true, INFINITE);  //終了  printf("- main(): end -\n");  fflush(stdout);  return EXIT\_SUCCESS;  } |

↓（実行結果） ※Debugビルド

|  |
| --- |
| - main(): start -  - func1(): start -  func1(): process ...  - func2(): start -  func2(): wait ...  - func1(): end -  func2(): ok!  func2(): process ...  - func2(): end -  - main(): end - |

これは正常に動作した結果である。

Debugビルドで正常動作を確認できたので、Releaseビルドでも動作を確認してみる。

↓（実行結果） ※Releaseビルド

|  |
| --- |
| - main(): start -  - func1(): start -  func1(): process ...  - func2(): start -  func2(): wait ...  - func1(): end -  ※ここから先に進まない（エラーは出ない） |

なぜ挙動が変わったのか？それは、Releaseビルドの最適化処理の影響である。

説明のために、Debugビルド時とReleasビルド時のアセンブラコードを示す。（Visual C++ 2013 で調査）

Debugビルド時のアセンブラコード：　※一部抜粋

|  |
| --- |
| ...（略）...  21:  22: //処理終了通知  23: s\_isFinished = true;  00F84018 C6 05 30 81 F8 00 01 mov byte ptr ds:[0F88130h],1 ← s\_isFinished のメモリに1 をセット  ...（略）...  41: while (!s\_isFinished){} 【ループ処理：ここから】  00F84124 0F B6 05 30 81 F8 00 movzx eax,byte ptr ds:[0F88130h] ← s\_isFinished のメモリを読み込み  00F8412B 85 C0 test eax,eax ← 値が 0 かチェック  00F8412D 75 02 jne func2+0B1h (0F84131h) ← 0 以外なら 00F84131 にジャンプ  00F8412F EB F3 jmp func2+0A4h (0F84124h) ← (0 だったので) 00F84124 にジャンプ  42: printf("func2(): ok!\n"); 【ループ処理：ここまで】  00F84131 8B F4 mov esi,esp  ...（略）... |

Releaseビルド時のアセンブラコード：　※一部抜粋

|  |
| --- |
| ...（略）...  22: //処理終了通知  23: s\_isFinished = true;  24:  25: //終了  26: printf("- func1(): end -\n");  00BD1041 68 3C 21 BD 00 push 0BD213Ch  00BD1046 C6 05 68 33 BD 00 01 mov byte ptr ds:[0BD3368h],1 ← s\_isFinished のメモリに1 をセット  ...（略）...  00BD10A3 A0 68 33 BD 00 mov al,byte ptr ds:[00BD3368h] ← s\_isFinished のメモリを読み込み  00BD10A8 83 C4 10 add esp,10h  00BD10AB EB 03 jmp func2+40h (0BD10B0h)  00BD10AD 8D 49 00 lea ecx,[ecx]  41: while (!s\_isFinished){} 【ループ処理：ここから】  00BD10B0 84 C0 test al,al ← 値が 0 かチェック  00BD10B2 74 FC je func2+40h (0BD10B0h) ← 0 なら 00BD10B0 にジャンプ  【ループ処理：ここまで】  ...（略）... |

前述の「レジスタ」の説明で、コンパイラが最適化によって変数の参照を極力減らすことを解説した。この結果は、その影響を受けたものである。

Releaseビルドのアセンブラコードを見ると（最適化によってC++言語のコードとアセンブラの対応付けがきちんとできない状態になっているが）、「s\_isFinished」という変数をメモリから読み出しているのがループの外だと分かる。これでは他で変数の値が書き換わってもループ処理中に検出できない。そのため、ループが終了しなくなってしまったのである。

### 解決策

このような問題に対して、C++言語は言語仕様として解決策を用意している。

「volatile型修飾子」である。

この修飾子が付いた変数は、コンパイラが最適化せずに、常にメモリを参照するようになる。

なお、volatile型修飾子は、「揮発性型修飾子」と訳されたりもするが、「volatile = 変わりやすい、気まぐれ」の意味で捉えたほうが理解しやすい。

修正を加えたプログラムを示す。

スレッド間の情報共有に失敗するサンプル：※修正版の一部抜粋

|  |
| --- |
| …（略）…  //処理終了フラグ  volatile bool s\_isFinished = false;  …（略）… |

Releaseビルド時のアセンブラコード：　※修正版の一部抜粋

|  |
| --- |
| ...（略）...  22: //処理終了通知  23: s\_isFinished = true;  24:  25: //終了  26: printf("- func1(): end -\n");  00BD1041 68 3C 21 BD 00 push 0BD213Ch  00BD1046 C6 05 68 33 BD 00 01 mov byte ptr ds:[0BD3368h],1 ← s\_isFinished のメモリに1 をセット  ...（略）...  41: while (!s\_isFinished){} 【ループ処理：ここから】  00B810A6 80 3D 68 33 B8 00 00 cmp byte ptr ds:[0B83368h],0 ← s\_isFinished のメモリが 0 かチェック  00B810AD 74 F7 je func2+36h (0B810A6h) ← 0 なら 00B810A6 にジャンプ  【ループ処理：ここまで】  ...（略）... |

↓（実行結果） ※Releaseビルド（修正版）

|  |
| --- |
| - main(): start -  - func1(): start -  func1(): process ...  - func2(): start -  func2(): wait ...  - func1(): end -  func2(): ok!  func2(): process ...  - func2(): end -  - main(): end - |

## スレッド間の情報共有の失敗②：CPUの最適化による問題

後述する「ミューテックス」のような「安全だが低速な同期手法」に頼らず、高速化のためにと、自前で同期の制御を行おうとすると、一見してCPUの不具合とも思えるような不可思議な問題に直面することがある。

### 問題点

例えば、下記のような処理があるとする。

**[グローバル変数]**

|  |
| --- |
| volatile bool flg = false;  volatile int val = 0; |

**[スレッドA]**

|  |
| --- |
| val = 123; //先に値をセットしてから、  flg = true; //フラグを更新 |

**[スレッドB]**

|  |
| --- |
| if(flg) //フラグの更新状態をチェック  std::cout << "val=" << val << std::endl; //値を表示 |

**[処理結果]**※タイミングによって起こり得る問題

|  |
| --- |
| val=0 ←普通に考えて確実に 123 となるはずが、そうならない場合がある |

スレッドAは、変数 val に値をセットした上で flg を更新することで、スレッドBに値の更新を通知している。

スレッドBはフラグの更新を確認してから、変数 val を表示している。

valを先に更新してから flg を更新しているので、flgの更新が確認できればvalには確実に値が格納されているはずである。しかし、そうならないことがある。

### アウト・オブ・オーダー実行

CPUは、必ずしもプログラム（機械語・アセンブラ）の順序通りに実行するとは限らない。

CPUは、命令の実行効率を上げるために、「アウト・オブ・オーダー」（順序を守らない実行）という、最適化を実行時に行う。

CPUの内部では、機械語の命令コードを逐次処理していくのだが、この時、可能な限り複数の命令を同時に実行しようとする。これは、「スーパースカラー」（もしくは「スーパースケーラ」）と呼ばれる、命令パイプラインを並列に処理する技術である。

これにより、実行順序が逆転することが起こり得る。

本来はプログラムの整合性を損なわない範囲でこのような最適化を行っているのだが、マルチスレッドや割り込みによって途中に処理が割り込まれると、問題として表面化することが起こり得る。

なお、「アウト・オブ・オーダー実行」に対して、順序どおりに命令を実行することを「イン・オーダー実行」と呼ぶ。

### メモリ操作命令：ロード／ストア

このような問題は、CPUのメモリ操作命令の順序が損なわれると起こる。

CPUは、メモリ上のデータを操作する際、「ロード」命令によってメモリからレジスタ（CPU内の変数）に読み出し、値を更新した後、「ストア」命令によってレジスタからメモリに書き戻す。

x86系CPUにおける一般的なロード命令は「MOV reg, [mem]」で、ストア命令は「MOV [mem], reg」というものである。命令はどちらも「MOV」であるが、その操作が「レジスタ←メモリ」（ロード）か「メモリ←レジスタ」（ストア）で呼び分ける。ほか、ロードとストアをいっぺんに行う「XCHG」という命令もある。

### メモリバリア

このような問題の解決策は、CPUの命令レベルで用意されている。

CPUに対して、メモリ操作命令の順序性を保証するように働きかける命令があり、そのような命令を「メモリバリア」（もしくは「メモリフェンス」）と呼ぶ。

### 解決策①

このような問題は、前述の「データ破損」で示した解決策を用いることで対処できる。

開発言語やSDKによって用意されている「排他制御」（ロック）の仕組みを利用することで、メモリバリアの確実性も保証される。

### 別の解決策：ロックフリー

ちょっとした値の更新で逐一ロックを行うのは処理コストが高い。

ロックを用いずにデータ共有を保証することができれば、ロックを使わないで済む場面もある。

ロックを用いずに確実なデータ共有を保証する手法（アルゴリズム）を「ロックフリー」（Lock-Free）と呼ぶ。また、スレッドの処理全体がロックフリーな処理になっていることを「ウェイトフリー」（Wait-Free）と呼ぶ。

処理のパフォーマンスを追求するなら、このようなロックフリーの処理構造を追求しなければならない。

### 解決策②

ロックフリーな処理を実現するには、CPUのメモリバリア命令を使用することである。

C++11では、「アトミック型」と呼ばれるクラスが用意されており、これを使用すれば、コンパイラがCPUのメモリバリア命令を使用するようにコード生成する。

「アトミック型」は、ロックフリーでデータを共有する手段となる。これを用いた場合、前述の問題のコードはこのように修正する。

**[グローバル変数]**

|  |
| --- |
| std::atmic<bool> flg = false;  volatile int val = 0; |

**[スレッドA]**

|  |
| --- |
| val = 123; //先に値をセットしてから、  flg.store(true); //フラグを更新 |

**[スレッドB]**

|  |
| --- |
| if(flg.load()) //フラグの更新状態をチェック  std::cout << "val=" << val << std::endl; //値を表示 |

**[処理結果]**

|  |
| --- |
| val=123 ←確実に 123 という結果が得られる |

「アトミック型」にはメモリ操作方法を指定するオプションがあり、それを使うと、逆に「メモリバリアを使わない」という指定もできる。これにより、余計な命令が発行されなくなるため、わずかにパフォーマンスが向上する。厳密なパフォーマンスチューニングを行う場合に活用できる。

参考までに、メモリ操作方法を指定したコードは下記のようになる。

**[スレッドA]**

|  |
| --- |
| val = 123; //先に値をセットしてから、  flg.store(true, std::memory\_order\_release);//フラグを更新：ストア専用のメモリバリアを指定 |

**[スレッドB]**

|  |
| --- |
| if(flg.load(std::memory\_order\_relaxed)) //フラグの更新状態をチェック：メモリバリアなし  std::cout << "val=" << val << std::endl; //値を表示 |

「アトミック型」のデフォルトは、ロード／ストア共にメモリバリアすることである。

以上のように、C++11のアトミック型は、一見して単なるクラスライブラリだが、volatle型と同様に、コンパイラにコード生成の方法を指定するためのものであり、言語仕様に密着している。

このメモリ操作指定に関する説明は割愛するが、後述する「モニター：アトミック操作によるビジーウェイト」のサンプルプログラムに、コメントとして説明を記載している。

## 困難なマルチスレッドプログラミング

以上の例からも、マルチスレッドのプログラミングは非常に繊細であることがわかる。また、こうした問題について理解していないと、問題発生時に原因を探り当てることが非常に困難となり、闇雲にハードウェアやコンパイラの不具合を疑いだすことにもなりかねない。

そのため、面倒でも、マルチスレッドプログラミングは十分に理解を深めてから取り組むべきである。

マルチスレッドプログラミングでは、デバッガも有効に使えないことがあったり、デバッグ用の変数操作一つで挙動が変わったりする。後者は、メモリ操作が増えて、思いがけずメモリバリアの役割を果たし、問題が再現しなくなるといったことがある。

繰り返しになるが、このような十分な知識と理解がないと、マルチスレッドの問題を自力で解決することができない。

# スレッドの同期

スレッドの同期について解説する。

「同期」とは、先の「データ破損の問題」で示したように、複数のスレッドがタイミングを取り合うことである。

同期が必要なのは、「確実な情報の共有」や「共有リソースの排他的なアクセス」を行うためである。一方のスレッドが情報を作ったり更新したりしている間に、他方のスレッドはその完了を「待つ」ことになる。

この「待つ」という処理を適切に行わないと、大きくパフォーマンスを劣化させてしまうことになりかねないので注意が必要である。

具体的な同期の手法を説明する前に、まず、この「待つ」手法について説明する。

大きくは2種類の待ち方がある。

## ビジーウェイト

「ビジーウェイト」は、その名の通り「忙しく待つ」ことである。

ループ処理でフラグを監視するような処理が該当する。

また、「スピンロック」など、ライブラリが提供する機能にもビジーウェイトを扱うものがある。

ビジーウェイトは非常に軽量で、待機の必要がない状況ではほとんど無駄な処理をせずに素通りする。しかし、待機が発生すると、激しく状態を監視し続け、CPUを占有する。そのため、並列処理ができない環境（シングルプロセッサ／シングルコア）でビジーウェイト状態に入ると、必ずタイムスライスを使い切ることになり、非効率となる。

ビジーウェイトの使いどころは、並列処理環境（マルチプロセッサ／マルチコア）にある。

並列処理環境であれば、ビジーウェイト中にも他のスレッドが状態を更新できる。しかし、それでもCPU（コア）の一つは占有されるので、長時間の待機に用いるとやはり非効率となる。

このように、ビジーウェイトには何かと制約が多いので、大抵の場合は「スリープ」を用いるものと考えたほうが良い。

ビジーウェイトを用いるケースをまとめると、下記の条件が揃った時である。

並列処理環境にある

待機時間がごく短い（タイムスライスを超えるような長い待機にならない）

【できるだけ】排他関係にあるスレッドの並列稼働（別コアへの振り分け）をスケジューリングできる

## スリープ

「スリープ」とは、スレッドが一時的に稼働を停止することである。

スリープ中のスレッドは、OSのタスクスケジューリングから外れて、アクティブにならなくなる。

スレッドをスリープさせ、フラグ更新などの待機終了の条件が揃った時に自動的にOSに起こしてもらうことができる。

スレッドが稼働を停止するため、他のスレッドが効率的に稼働できる。

基本的には、スレッドの同期にはこのスリープを用いるのが普通。ただし、マルチプロセッサ／コア環境では、前述の通り、「ビジーウェイト」の方が効率的となる場面もあるため、環境に合わせて使い分けること。

## スリープとYield

「スリープ」を行うのは、「同期」が必要な時だけではない。

前述の「スレッド優先度」でも説明した通り、優先度の高いスレッドがタイムスライスを使い切ってばかりいると、優先度の低いスレッドが動作できなくなる。

そのため、スレッドの処理では、同期の必要がなくても、定期的にスリープを行うのが作法である。

この場合のスリープは「待機」目的ではなく、他のスレッドに制御を明け渡すことが目的である。そのような場合、時間指定なしのスリープ「Sleep(0)」を呼び出すとよい。

また、制御を明け渡すだけであれば、「Yiled」（イールド）という手法もある。ただし、Yiledの扱いには注意が必要。

Yieldは、「待機状態にならずに」優先度の同じ他のスレッドに制御を明け渡す。つまり、同じ優先度の他のスレッドの処理が済んだらまた自分のスレッドに制御が戻ってきて、下位の優先度のスレッドには制御が移らない。

POSIXスレッドライブラリの sched\_yield() はこの挙動である。

また、Win32APIのYieldは「廃止予定」であり、使用しないように勧められている。代わりは Sleep(0) である。

ほかにも、Win32APIにはSwitchToThread() という関数が用意されており、こちらは「再スケジューリングは、呼び出し側スレッドの優先順位と実行可能な他のスレッドの状態によって決定します」と説明されている。

C++11の std::this\_thread::yield() はOSに依存するようである。

他のゲーム用SDKやJava,C#などにもYieldがあるが、挙動が違う場合もあるので、リファレンスマニュアルをよく確認したほうがよい。

# 様々な同期手法

まず、様々な同期手法を列挙する。



以上に列挙したものは、とりわけ代表的なものである。これらの知識があれば、ゲーム用SDKにも応用が利く。

他には、OpenMP（ミューテックスなどの同期処理もある）や、Boost C++、.Net Framework、Java、さらに、Visual C++ 2010 で追加された「同時実行ランタイム」ライブラリなど、枚挙にいとまがない。

前述の「様々なスレッド」では、ゲームプログラミングでにおけるスレッド作成にはC++11が向いていないと説明したが、同期オブジェクトに関しては、C++11を優先的な選択肢と見た方が望ましい。後発なだけに、様々な機能に対応している上、汎用性の高いコーディングが行える。Visual C++ も C++11 の対応が進んでおり、Win32APIよりも高機能な面がある。

以降、各同期処理の説明とサンプルプログラムを示す。

## 排他制御（ロック）

前述の「データ破損」の問題で示したように、共有する情報に対して「不可分操作」（アトミック操作）を保証するために用いる制御手法。

様々なバリエーションがあるため、動作環境や処理に合わせて適切な技術を用いる。

ロックを多用することでデータ破損を免れることができても、過剰なロックはパフォーマンス低下やデッドロックなどの問題にもつながることに注意が必要。

## 排他制御：通常変数でロック（誤ったロック）

まず、他の処理との比較用に、開発言語やSDKで提供された機能を用いず、通常変数だけでロックを実装した場合のサンプルを示す。

これは誤ったプログラミングのサンプルである。

最も高速ではあるが、同期が成立していない。

### Win32API版

スレッド作成と処理時間計測にWin32APIを用いる。ロック制御はスタティック変数によるフラグで行う。

（Win32API版）通常変数によるロック制御のサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <Windows.h>  #include <Process.h>  //ロック取得用変数  static int s\_lock = 0;  //共有データ  static int s\_commonData = 0;  //スレッド固有データ  \_\_declspec(thread) int s\_tlsData = 0;  //スレッド  unsigned int WINAPI threadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);  //開始  printf("- begin:%s -\n", name);  fflush(stdout);  //処理  for (int i = 0; i < 3; ++i)  {  //ロック取得用変数でロック取得待ち  while (s\_lock != 0){} s\_lock = 1;    //データ表示（前）  printf("%s: [BEFORE] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～4 msec）  Sleep(rand() % 5);  //データ書き戻し  s\_commonData = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("%s: [AFTER] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //ロック取得用変数でロック解除  s\_lock = 0;  //スレッド切り替えのためのスリープ  Sleep(0);  // //スレッド切り替え  // SwitchToThread();//OSに任せて再スケジューリング  // Yield();//廃止  }  //終了  printf("- end:%s -\n", name);  fflush(stdout);  return 0;  }  //テスト  int main(const int argc, const char\* argv[])  {  //スレッド作成  static const int THREAD\_NUM = 3;  unsigned int tid[THREAD\_NUM] = {};  HANDLE hThread[THREAD\_NUM] =  {  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "太郎", 0, &tid[0]),  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "次郎", 0, &tid[1]),  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "三郎", 0, &tid[2])  };    //スレッド終了待ち  WaitForMultipleObjects(THREAD\_NUM, hThread, true, INFINITE);  //スレッドハンドルクローズ  for (int i = 0; i < THREAD\_NUM; ++i)  {  CloseHandle(hThread[i]);  }  //比較用に空ループで時間を計測  {  LARGE\_INTEGER freq;  QueryPerformanceFrequency(&freq);  LARGE\_INTEGER begin;  QueryPerformanceCounter(&begin);  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  while (s\_lock != 0){} s\_lock = 1;  s\_lock = 0;  }  LARGE\_INTEGER end;  QueryPerformanceCounter(&end);  float duration = static\_cast<float>(static\_cast<double>(end.QuadPart - begin.QuadPart) / static\_cast<double>(freq.QuadPart));  printf("Loop \* %d = %.6f sec\n", TEST\_TIMES, duration);  }  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:太郎 -  太郎: [BEFORE] commonData=0, tlsData=0  - begin:次郎 -  - begin:三郎 -  太郎: [AFTER] commonData=1, tlsData=1  次郎: [BEFORE] commonData=1, tlsData=0  三郎: [BEFORE] commonData=1, tlsData=0 ←ロックが失敗している  次郎: [AFTER] commonData=2, tlsData=1  三郎: [AFTER] commonData=2, tlsData=1  太郎: [BEFORE] commonData=2, tlsData=1  次郎: [BEFORE] commonData=2, tlsData=1 ←ロックが失敗している  次郎: [AFTER] commonData=3, tlsData=2  太郎: [AFTER] commonData=3, tlsData=2  三郎: [BEFORE] commonData=3, tlsData=1  三郎: [AFTER] commonData=4, tlsData=2  太郎: [BEFORE] commonData=4, tlsData=2  次郎: [BEFORE] commonData=4, tlsData=2 ←ロックが失敗している  次郎: [AFTER] commonData=5, tlsData=3  太郎: [AFTER] commonData=5, tlsData=3  - end:次郎 -  三郎: [BEFORE] commonData=5, tlsData=2  - end:太郎 -  三郎: [AFTER] commonData=6, tlsData=3 ←3回ロックに失敗したので、正しい処理結果になっていない  - end:三郎 -  Loop \* 10000000 = 0.006544 sec ←1千万回ループによる処理時間計測 |

## 排他制御：ミューテックス

「ミューテックス」は、最も代表的なロック手法。

スリープ方式の代名詞的な手法。なお、これに対して、ビジーウェイト方式の代表格は「スピンロック」。

### POSIXスレッドライブラリ版

LinuxなどのUnix系OSや、GCCベースのゲーム系SDKで一般的なスタイル。

ミューテックス生成時の属性により、再帰ロックやデッドロック検出に対応しているのが特徴。デフォルトではどちらも無効。また、Cygwin環境ではこれらの属性が使用できないなど、環境依存の機能。

再帰ロックとは、同一スレッドが何度もロックを取得できること。取得した数と同じ数だけ解放する必要がある。再帰ロック無効時は二度目のロックでデッドロックを起こす。（「デッドロック」については後述）

POSIXスレッドライブラリ版ミューテックスのサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <unistd.h>  #include <sys/time.h> //時間計測用  //CygwinのGCCではこの拡張定数が使えない  //#define PTHREAD\_MUTEX\_FAST\_NP (0)  //#define PTHREAD\_MUTEX\_RECURSIVE\_NP (1)  //#define PTHREAD\_MUTEX\_ERRORCHECK\_NP (2)  //ミューテックス  static pthread\_mutex\_t s\_mutex = PTHREAD\_MUTEX\_INITIALIZER;  //共有データ  static int s\_commonData = 0;  //スレッド固有データ  \_\_thread int s\_tlsData = 0;  //スレッド  void\* threadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);  //開始  printf("- begin:%s -\n", name);  fflush(stdout);  //処理  for (int i = 0; i < 3; ++i)  {  //ミューテックス取得  pthread\_mutex\_lock(&s\_mutex);  // pthread\_mutex\_trylock(&s\_mutex);//ロックを取得できない時に他の処理を行いたい場合は  // //pthread\_mutex\_trylock() を使用する    //データ表示（前）  printf("%s: [BEFORE] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～4 msec）  usleep((rand() % 5) \* 1000);    //データ書き戻し  s\_commonData = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("%s: [AFTER] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //ミューテックス解放  pthread\_mutex\_unlock(&s\_mutex);    //スレッド切り替えのためのスリープ  usleep(0);  // //スレッド切り替え  // sched\_yield();//同じ優先度の他のスレッドに切り替え  }  //終了  printf("- end:%s -\n", name);  fflush(stdout);  return 0;  }  //テスト  int main(const int argc, const char\* argv[])  {  //ミューテックス生成  //※PTHREAD\_MUTEX\_INITIALIZER で初期化している場合は不要  {  // pthread\_mutexattr\_t attr;  // pthread\_mutexattr\_init(&attr);  // pthread\_mutexattr\_setpshared(&attr, PTHREAD\_PROCESS\_PRIVATE);//単独プロセス専用　※デフォルト  // pthread\_mutexattr\_setpshared(&attr, PTHREAD\_PROCESS\_SHARED);//プロセス間で共有  // pthread\_mutexattr\_settype(&attr, PTHREAD\_MUTEX\_FAST\_NP);//高速（Fast）ミューテックス属性　※デフォルト  // pthread\_mutexattr\_settype(&attr, PTHREAD\_MUTEX\_RECURSIVE\_NP);//再帰的な (recursive) ミューテックス属性  // //※多重ロックに対して多重開放が必要  // pthread\_mutexattr\_settype(&attr, PTHREAD\_MUTEX\_ERRORCHECK\_NP);//エラー検査を行う (error checking)  // //ミューテックス属性  // //※デッドロック検出（pthread\_mutex\_lock()  // //　がエラーコード EDEADLK を返す）  // pthread\_mutex\_init(&s\_mutex, &attr);  }    //スレッド作成  static const int THREAD\_NUM = 3;  pthread\_t pth[THREAD\_NUM];  {  pthread\_attr\_t attr;  pthread\_attr\_init(&attr);  pthread\_attr\_setstacksize(&attr, 1024);//スタックサイズ指定  pthread\_create(&pth[0], &attr, threadFunc, (void\*)"太郎");  pthread\_create(&pth[1], &attr, threadFunc, (void\*)"次郎");  pthread\_create(&pth[2], &attr, threadFunc, (void\*)"三郎");  }    //スレッド終了待ち  for(int i = 0; i < THREAD\_NUM; ++i)  {  pthread\_join(pth[i], NULL);  }    //ミューテックスの取得と解放を大量に実行して時間を計測  {  struct timeval begin;  gettimeofday(&begin, NULL);  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  pthread\_mutex\_lock(&s\_mutex);  pthread\_mutex\_unlock(&s\_mutex);  }  struct timeval end;  gettimeofday(&end, NULL);  struct timeval duration;  if( end.tv\_usec >= begin.tv\_usec)  {  duration.tv\_sec = end.tv\_sec - begin.tv\_sec;  duration.tv\_usec = end.tv\_usec - begin.tv\_usec;  }  else  {  duration.tv\_sec = end.tv\_sec - begin.tv\_sec - 1;  duration.tv\_usec = 1000000 - begin.tv\_usec + end.tv\_usec;  }  printf("Mutex \* %d = %d.%06d sec\n", TEST\_TIMES, duration.tv\_sec, duration.tv\_usec);  }    //ミューテックス破棄  //※PTHREAD\_MUTEX\_INITIALIZER で初期化している場合は不要  {  // pthread\_mutex\_destroy(&s\_mutex);  }    return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:三郎 -  三郎: [BEFORE] commonData=0, tlsData=0  三郎: [AFTER] commonData=1, tlsData=1  三郎: [BEFORE] commonData=1, tlsData=1  - begin:太郎 -  三郎: [AFTER] commonData=2, tlsData=2  太郎: [BEFORE] commonData=2, tlsData=0  - begin:次郎 -  太郎: [AFTER] commonData=3, tlsData=1  三郎: [BEFORE] commonData=3, tlsData=2  三郎: [AFTER] commonData=4, tlsData=3  次郎: [BEFORE] commonData=4, tlsData=0  - end:三郎 -  次郎: [AFTER] commonData=5, tlsData=1  太郎: [BEFORE] commonData=5, tlsData=1  太郎: [AFTER] commonData=6, tlsData=2  次郎: [BEFORE] commonData=6, tlsData=1  次郎: [AFTER] commonData=7, tlsData=2  太郎: [BEFORE] commonData=7, tlsData=2  太郎: [AFTER] commonData=8, tlsData=3  次郎: [BEFORE] commonData=8, tlsData=2  - end:太郎 -  次郎: [AFTER] commonData=9, tlsData=3 ←全て正常にロックされたので、正しい処理結果になっている  - end:次郎 -  Mutex \* 10000000 = 0.579944 sec ←1千万回ループによる処理時間計測 |

### Win32API版（名前なし）

Win32API版のミューテックス。

同一スレッドが何度もロックを取得してもデッドロックを起こさず、また、一回の解放で済む点が特徴。これに対して、クリティカルセクションやPOSIXスレッドライブラリ版ミューテックスは再帰ロックであり、取得した数だけ解放する必要がある。

Win32API版ミューテックスのサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <Windows.h>  #include <Process.h>  //ミューテックスハンドル  static HANDLE s\_hMutex = INVALID\_HANDLE\_VALUE;  //共有データ  static int s\_commonData = 0;  //スレッド固有データ  \_\_declspec(thread) int s\_tlsData = 0;  //スレッド  unsigned int WINAPI threadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);  //開始  printf("- begin:%s -\n", name);  fflush(stdout);  //処理  for (int i = 0; i < 3; ++i)  {  //ミューテックス取得  WaitForSingleObject(s\_hMutex, INFINITE);//ロックを取得できない時に他の処理を行いたい場合は  //タイムアウト値を指定する    //データ表示（前）  printf("%s: [BEFORE] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～4 msec）  Sleep(rand() % 5);  //データ書き戻し  s\_commonData = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("%s: [AFTER] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //ミューテックス解放  ReleaseMutex(s\_hMutex);    //スレッド切り替えのためのスリープ  Sleep(0);  // //スレッド切り替え  // SwitchToThread();//OSに任せて再スケジューリング  // Yield();//廃止  }  //終了  printf("- end:%s -\n", name);  fflush(stdout);  return 0;  }  //テスト  int main(const int argc, const char\* argv[])  {  //ミューテックス生成  {  s\_hMutex = CreateMutex(nullptr, false, nullptr);    // //属性を指定して生成する場合  // SECURITY\_ATTRIBUTES attr = { sizeof(SECURITY\_ATTRIBUTES), nullptr, true };//子プロセスにハンドルを継承する  // SECURITY\_ATTRIBUTES attr = { sizeof(SECURITY\_ATTRIBUTES), nullptr, false };//子プロセスにハンドルを  // //継承しない　※デフォルト  // s\_hMutex = CreateMutex(&attr, false, nullptr);  }  //スレッド作成  static const int THREAD\_NUM = 3;  unsigned int tid[THREAD\_NUM] = {};  HANDLE hThread[THREAD\_NUM] =  {  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "太郎", 0, &tid[0]),  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "次郎", 0, &tid[1]),  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "三郎", 0, &tid[2])  };    //スレッド終了待ち  WaitForMultipleObjects(THREAD\_NUM, hThread, true, INFINITE);  //スレッドハンドルクローズ  for (int i = 0; i < THREAD\_NUM; ++i)  {  CloseHandle(hThread[i]);  }  //ミューテックスの取得と解放を大量に実行して時間を計測  {  LARGE\_INTEGER freq;  QueryPerformanceFrequency(&freq);  LARGE\_INTEGER begin;  QueryPerformanceCounter(&begin);  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  WaitForSingleObject(s\_hMutex, INFINITE);  ReleaseMutex(s\_hMutex);  }  LARGE\_INTEGER end;  QueryPerformanceCounter(&end);  float duration = static\_cast<float>(static\_cast<double>(end.QuadPart - begin.QuadPart)  / static\_cast<double>(freq.QuadPart));  printf("Mutex \* %d = %.6f sec\n", TEST\_TIMES, duration);  }  //ミューテックス破棄  CloseHandle(s\_hMutex);  s\_hMutex = INVALID\_HANDLE\_VALUE;  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:太郎 -  - begin:次郎 -  太郎: [BEFORE] commonData=0, tlsData=0  - begin:三郎 -  太郎: [AFTER] commonData=1, tlsData=1  次郎: [BEFORE] commonData=1, tlsData=0  次郎: [AFTER] commonData=2, tlsData=1  三郎: [BEFORE] commonData=2, tlsData=0  三郎: [AFTER] commonData=3, tlsData=1  太郎: [BEFORE] commonData=3, tlsData=1  太郎: [AFTER] commonData=4, tlsData=2  次郎: [BEFORE] commonData=4, tlsData=1  次郎: [AFTER] commonData=5, tlsData=2  三郎: [BEFORE] commonData=5, tlsData=1  三郎: [AFTER] commonData=6, tlsData=2  太郎: [BEFORE] commonData=6, tlsData=2  太郎: [AFTER] commonData=7, tlsData=3  - end:太郎 -  次郎: [BEFORE] commonData=7, tlsData=2  次郎: [AFTER] commonData=8, tlsData=3  - end:次郎 -  三郎: [BEFORE] commonData=8, tlsData=2  三郎: [AFTER] commonData=9, tlsData=3 ←全て正常にロックされたので、正しい処理結果になっている  - end:三郎 -  Mutex \* 10000000 = 4.880666 sec ←1千万回ループによる処理時間計測 |

### Win32API版（名前付き）

Win32版の名前付きミューテックス。

基本的には名前無しミューテックスと変わらないが、プロセス間での排他制御や、スレッドごとにミューテックスのアクセス権を変えたい場合に使用する。

Win32API版名前付きミューテックスのサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <Windows.h>  #include <Process.h>  //共有ミューテックス名  static const char\* COMMON\_MUTEX\_NAME = "Common Mutex";  //共有データ  static int s\_commonData = 0;  //スレッド固有データ  \_\_declspec(thread) int s\_tlsData = 0;  //スレッド  unsigned int WINAPI threadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);  //開始  printf("- begin:%s -\n", name);  fflush(stdout);  //名前付きミューテックスオープン  HANDLE hMutex = OpenMutex(SYNCHRONIZE, false, COMMON\_MUTEX\_NAME);    //処理  for (int i = 0; i < 3; ++i)  {  //ミューテックス取得  WaitForSingleObject(hMutex, INFINITE);//ロックを取得できない時に他の処理を行いたい場合はタイムアウト値を  //指定する    //データ表示（前）  printf("%s: [BEFORE] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～4 msec）  Sleep(rand() % 5);  //データ書き戻し  s\_commonData = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("%s: [AFTER] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //ミューテックス解放  ReleaseMutex(hMutex);    //スレッド切り替えのためのスリープ  Sleep(0);  // //スレッド切り替え  // SwitchToThread();//OSに任せて再スケジューリング  // Yield();//廃止  }  //名前付きミューテックスクローズ  CloseHandle(hMutex);    //終了  printf("- end:%s -\n", name);  fflush(stdout);  return 0;  }  //テスト  int main(const int argc, const char\* argv[])  {  //ミューテックスハンドル  HANDLE hMutex = INVALID\_HANDLE\_VALUE;    //名前付きミューテックス生成  {  hMutex = CreateMutex(nullptr, false, COMMON\_MUTEX\_NAME);  // //属性を指定して生成する場合  // SECURITY\_ATTRIBUTES attr = { sizeof(SECURITY\_ATTRIBUTES), nullptr, true };//子プロセスにハンドルを継承する  // SECURITY\_ATTRIBUTES attr = { sizeof(SECURITY\_ATTRIBUTES), nullptr, false };//子プロセスにハンドルを  // //継承しない　※デフォルト  // hMutex = CreateMutex(&attr, false, COMMON\_MUTEX\_NAME);  }  //スレッド作成  static const int THREAD\_NUM = 3;  unsigned int tid[THREAD\_NUM] = {};  HANDLE hThread[THREAD\_NUM] =  {  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "太郎", 0, &tid[0]),  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "次郎", 0, &tid[1]),  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "三郎", 0, &tid[2])  };    //スレッド終了待ち  WaitForMultipleObjects(THREAD\_NUM, hThread, true, INFINITE);  //スレッドハンドルクローズ  for (int i = 0; i < THREAD\_NUM; ++i)  {  CloseHandle(hThread[i]);  }  //ミューテックスの取得と解放を大量に実行して時間を計測  {  LARGE\_INTEGER freq;  QueryPerformanceFrequency(&freq);  LARGE\_INTEGER begin;  QueryPerformanceCounter(&begin);  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  WaitForSingleObject(hMutex, INFINITE);  ReleaseMutex(hMutex);  }  LARGE\_INTEGER end;  QueryPerformanceCounter(&end);  float duration = static\_cast<float>(static\_cast<double>(end.QuadPart - begin.QuadPart)  / static\_cast<double>(freq.QuadPart));  printf("Named Mutex \* %d = %.6f sec\n", TEST\_TIMES, duration);  }  //名前付きミューテックス破棄  CloseHandle(hMutex);  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:太郎 -  - begin:次郎 -  - begin:三郎 -  太郎: [BEFORE] commonData=0, tlsData=0  太郎: [AFTER] commonData=1, tlsData=1  次郎: [BEFORE] commonData=1, tlsData=0  次郎: [AFTER] commonData=2, tlsData=1  三郎: [BEFORE] commonData=2, tlsData=0  三郎: [AFTER] commonData=3, tlsData=1  太郎: [BEFORE] commonData=3, tlsData=1  太郎: [AFTER] commonData=4, tlsData=2  次郎: [BEFORE] commonData=4, tlsData=1  次郎: [AFTER] commonData=5, tlsData=2  三郎: [BEFORE] commonData=5, tlsData=1  三郎: [AFTER] commonData=6, tlsData=2  太郎: [BEFORE] commonData=6, tlsData=2  太郎: [AFTER] commonData=7, tlsData=3  - end:太郎 -  次郎: [BEFORE] commonData=7, tlsData=2  次郎: [AFTER] commonData=8, tlsData=3  - end:次郎 -  三郎: [BEFORE] commonData=8, tlsData=2  三郎: [AFTER] commonData=9, tlsData=3 ←全て正常にロックされたので、正しい処理結果になっている  - end:三郎 -  Named Mutex \* 10000000 = 4.903843 sec ←1千万回ループによる処理時間計測 |

### C++11版

C++11版の名前付きミューテックス。

言語の標準機能として用意されたミューテックス。

言語標準なのでコードの汎用性が高いことが大きな特徴。クラスオブジェクトとして操作できる点も扱い易い。

また、lock\_guard クラスなどのC++11の他のクラスと組み合わせて便利に扱うことができる点も強み。（lock\_guard クラスは、いわゆる「Scoped Lock Pattern」を提供するクラスで、処理ブロックを抜けるときに自動的にロックを解放する。）

タイムアウトを行うための「std::timed\_mutex」、再帰ロックを行うための「std::recursive\_mutex」、その両方を組み合わせた「std::recursive\_timed\_mutex」という亜種もある。

以下のサンプルは、スレッド生成、スリープ、処理時間計測も含めて、すべてC++11のライブラリのみで扱っている。

C++11版ミューテックスのサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <thread>  #include <mutex>  #include <chrono> //時間計測用  #include <random> //乱数生成用  //ミューテックスオブジェクト  static std::mutex s\_mutex;  //共有データ  static int s\_commonData = 0;  //スレッド固有データ  \_\_declspec(thread) int s\_tlsData = 0;  //thread\_local int s\_tlsData = 0;//Visual C++ 2013 では thread\_local キーワードが使えない  //スレッド  void threadFunc(const char\* name)  {  //開始  printf("- begin:%s -\n", name);  fflush(stdout);  //乱数  std::random\_device seed\_gen;  std::mt19937 rnd(seed\_gen());  std::uniform\_int\_distribution<int> sleep\_time(0, 4);  //処理  for (int i = 0; i < 3; ++i)  {  //ミューテックス取得  s\_mutex.lock();  // s\_mutex.try\_lock();//ロックを取得できない時に他の処理を行いたい場合は try\_lock() を使用する  //タイムアウトしたい場合は timed\_mutex クラスを使用する  //再帰ロックしたい場合は recursive\_mutex クラスを使用する  //データ表示（前）  printf("%s: [BEFORE] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～4 msec）  std::this\_thread::sleep\_for(std::chrono::milliseconds(sleep\_time(rnd)));  //データ書き戻し  s\_commonData = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("%s: [AFTER] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //ミューテックス解放  s\_mutex.unlock();    //スレッド切り替えのためのスリープ  std::this\_thread::sleep\_for(std::chrono::milliseconds(0));  // //スレッド切り替え  // std::this\_thread::yield();//OSに任せて再スケジューリング  }  //終了  printf("- end:%s -\n", name);  fflush(stdout);  }  //テスト  int main(const int argc, const char\* argv[])  {  //スレッド作成  //※スタックサイズ指定方法不明  std::thread thread\_obj1 = std::thread(threadFunc, "太郎");  std::thread thread\_obj2 = std::thread(threadFunc, "次郎");  std::thread thread\_obj3 = std::thread(threadFunc, "三郎");    //スレッド終了待ち  thread\_obj1.join();  thread\_obj2.join();  thread\_obj3.join();  //ミューテックスの取得と解放を大量に実行して時間を計測  {  auto begin = std::chrono::high\_resolution\_clock::now();  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  s\_mutex.lock();  s\_mutex.unlock();  }  auto end = std::chrono::high\_resolution\_clock::now();  auto duration = static\_cast<float>(static\_cast<double>(  std::chrono::duration\_cast<std::chrono::microseconds>(end - begin).count()) / 1000000.);  printf("Mutex \* %d = %.6f sec\n", TEST\_TIMES, duration);  }  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:太郎 -  太郎: [BEFORE] commonData=0, tlsData=0  - begin:次郎 -  - begin:三郎 -  太郎: [AFTER] commonData=1, tlsData=1  次郎: [BEFORE] commonData=1, tlsData=0  次郎: [AFTER] commonData=2, tlsData=1  三郎: [BEFORE] commonData=2, tlsData=0  三郎: [AFTER] commonData=3, tlsData=1  太郎: [BEFORE] commonData=3, tlsData=1  太郎: [AFTER] commonData=4, tlsData=2  次郎: [BEFORE] commonData=4, tlsData=1  次郎: [AFTER] commonData=5, tlsData=2  三郎: [BEFORE] commonData=5, tlsData=1  三郎: [AFTER] commonData=6, tlsData=2  太郎: [BEFORE] commonData=6, tlsData=2  太郎: [AFTER] commonData=7, tlsData=3  - end:太郎 -  次郎: [BEFORE] commonData=7, tlsData=2  次郎: [AFTER] commonData=8, tlsData=3  - end:次郎 -  三郎: [BEFORE] commonData=8, tlsData=2  三郎: [AFTER] commonData=9, tlsData=3 ←全て正常にロックされたので、正しい処理結果になっている  - end:三郎 -  Mutex \* 10000000 = 0.650037 sec ←1千万回ループによる処理時間計測 |

## 排他制御：スピンロック

「スピンロック」は古典的なロック手法。

ビジーウェイト方式の代名詞的な手法。

ミューテックスと異なりOSに依存する部分が少ないため、軽量かつ高速。

前述の「ビジーウェイト」で説明しているとおり、並列処理環境でない場合や長時間の排他処理では非効率になるので注意。

### POSIXスレッドライブラリ版

POSIXスレッドライブラリ版のスピンロック。

POSIXスレッドライブラリ版スピンロックのサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <unistd.h>  #include <sys/time.h> //時間計測用  //スピンロック  static pthread\_spinlock\_t s\_lock;  //共有データ  static int s\_commonData = 0;  //スレッド固有データ  \_\_thread int s\_tlsData = 0;  //スレッド  void\* threadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);  //開始  printf("- begin:%s -\n", name);  fflush(stdout);  //処理  for (int i = 0; i < 3; ++i)  {  //スピンロック取得  pthread\_spin\_lock(&s\_lock);  // pthread\_spin\_trylock(&s\_lock);//ロックを取得できない時に他の処理を行いたい場合は  // //pthread\_spin\_trylock() を使用する    //データ表示（前）  printf("%s: [BEFORE] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);    //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～4 msec）  usleep((rand() % 5) \* 1000);    //データ書き戻し  s\_commonData = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("%s: [AFTER] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //スピンロック解放  pthread\_spin\_unlock(&s\_lock);    //スレッド切り替えのためのスリープ  usleep(0);  // //スレッド切り替え  // sched\_yield();//同じ優先度の他のスレッドに切り替え  }  //終了  printf("- end:%s -\n", name);  fflush(stdout);  return 0;  }  //テスト  int main(const int argc, const char\* argv[])  {  //スピンロック生成  pthread\_spin\_init(&s\_lock, PTHREAD\_PROCESS\_PRIVATE);//単独プロセス専用  // pthread\_spin\_init(&s\_lock, PTHREAD\_PROCESS\_SHARED);//プロセス間で共有    //スレッド作成  static const int THREAD\_NUM = 3;  pthread\_t pth[THREAD\_NUM];  {  pthread\_attr\_t attr;  pthread\_attr\_init(&attr);  pthread\_attr\_setstacksize(&attr, 1024);//スタックサイズ指定  pthread\_create(&pth[0], &attr, threadFunc, (void\*)"太郎");  pthread\_create(&pth[1], &attr, threadFunc, (void\*)"次郎");  pthread\_create(&pth[2], &attr, threadFunc, (void\*)"三郎");  }    //スレッド終了待ち  for(int i = 0; i < THREAD\_NUM; ++i)  {  pthread\_join(pth[i], NULL);  }    //スピンロックの取得と解放を大量に実行して時間を計測  {  struct timeval begin;  gettimeofday(&begin, NULL);  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  pthread\_spin\_lock(&s\_lock);  pthread\_spin\_unlock(&s\_lock);  }  struct timeval end;  gettimeofday(&end, NULL);  struct timeval duration;  if( end.tv\_usec >= begin.tv\_usec)  {  duration.tv\_sec = end.tv\_sec - begin.tv\_sec;  duration.tv\_usec = end.tv\_usec - begin.tv\_usec;  }  else  {  duration.tv\_sec = end.tv\_sec - begin.tv\_sec - 1;  duration.tv\_usec = 1000000 - begin.tv\_usec + end.tv\_usec;  }  printf("Spinlock \* %d = %d.%06d sec\n", TEST\_TIMES, duration.tv\_sec, duration.tv\_usec);  }    //スピンロック破棄  pthread\_spin\_destroy(&s\_lock);    return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:三郎 -  三郎: [BEFORE] commonData=0, tlsData=0  三郎: [AFTER] commonData=1, tlsData=1  - begin:次郎 -  次郎: [BEFORE] commonData=1, tlsData=0  - begin:太郎 -  次郎: [AFTER] commonData=2, tlsData=1  太郎: [BEFORE] commonData=2, tlsData=0  太郎: [AFTER] commonData=3, tlsData=1  次郎: [BEFORE] commonData=3, tlsData=1  次郎: [AFTER] commonData=4, tlsData=2  三郎: [BEFORE] commonData=4, tlsData=1  三郎: [AFTER] commonData=5, tlsData=2  次郎: [BEFORE] commonData=5, tlsData=2  次郎: [AFTER] commonData=6, tlsData=3  太郎: [BEFORE] commonData=6, tlsData=1  - end:次郎 -  太郎: [AFTER] commonData=7, tlsData=2  三郎: [BEFORE] commonData=7, tlsData=2  三郎: [AFTER] commonData=8, tlsData=3  太郎: [BEFORE] commonData=8, tlsData=2  - end:三郎 -  太郎: [AFTER] commonData=9, tlsData=3 ←全て正常にロックされたので、正しい処理結果になっている  - end:太郎 -  Spinlock \* 10000000 = 0.188224 sec ←1千万回ループによる処理時間計測 |

## 排他制御：クリティカルセクション

「クリティカルセクション」という用語自体は、その名の通り「重大な処理部分」のことであり、アトミック操作が必要な処理のことを指す。

### Win32API版

Win32API版のクリティカルセクション。

Windowsの軽量ミューテックス的存在。OSがハンドルを管理しない。

ビジーウェイトとスリープのハイブリッドで扱える点が大きな特徴。

Win32API版クリティカルセクションのサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <Windows.h>  #include <Process.h>  //クリティカルセクション  static CRITICAL\_SECTION s\_criticalSection;  //共有データ  static int s\_commonData = 0;  //スレッド固有データ  \_\_declspec(thread) int s\_tlsData = 0;  //スレッド  unsigned int WINAPI threadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);  //開始  printf("- begin:%s -\n", name);  fflush(stdout);  //処理  for (int i = 0; i < 3; ++i)  {  //クリティカルセクション取得  EnterCriticalSection(&s\_criticalSection);  // TryEnterCriticalSection(&s\_criticalSection);//ロックを取得できない時に他の処理を行いたい場合は  // //TryEnterCriticalSection() を使用する  //データ表示（前）  printf("%s: [BEFORE] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～4 msec）  Sleep(rand() % 5);  //データ書き戻し  s\_commonData = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("%s: [AFTER] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //クリティカルセクション解放  LeaveCriticalSection(&s\_criticalSection);    //スレッド切り替えのためのスリープ  Sleep(0);  // //スレッド切り替え  // SwitchToThread();//OSに任せて再スケジューリング  // Yield();//廃止  }  //終了  printf("- end:%s -\n", name);  fflush(stdout);  return 0;  }  //テスト  int main(const int argc, const char\* argv[])  {  //クリティカルセクション初期化  InitializeCriticalSection(&s\_criticalSection);  // InitializeCriticalSectionAndSpinCount(&s\_criticalSection, 1000);//スピンロックカウント数を指定する場合  // //（スピンロック後に待機する）：  // //マルチプロセッサシステム専用  //スレッド作成  static const int THREAD\_NUM = 3;  unsigned int tid[THREAD\_NUM] = {};  HANDLE hThread[THREAD\_NUM] =  {  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "太郎", 0, &tid[0]),  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "次郎", 0, &tid[1]),  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "三郎", 0, &tid[2])  };    //スレッド終了待ち  WaitForMultipleObjects(THREAD\_NUM, hThread, true, INFINITE);  //スレッドハンドルクローズ  for (int i = 0; i < THREAD\_NUM; ++i)  {  CloseHandle(hThread[i]);  }  //クリティカルセクションの取得と解放を大量に実行して時間を計測  {  LARGE\_INTEGER freq;  QueryPerformanceFrequency(&freq);  LARGE\_INTEGER begin;  QueryPerformanceCounter(&begin);  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  EnterCriticalSection(&s\_criticalSection);  LeaveCriticalSection(&s\_criticalSection);  }  LARGE\_INTEGER end;  QueryPerformanceCounter(&end);  float duration = static\_cast<float>(static\_cast<double>(end.QuadPart - begin.QuadPart)  / static\_cast<double>(freq.QuadPart));  printf("Critical Section \* %d = %.6f sec\n", TEST\_TIMES, duration);  }  //クリティカルセクション破棄  DeleteCriticalSection(&s\_criticalSection);  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:太郎 -  太郎: [BEFORE] commonData=0, tlsData=0  - begin:次郎 -  - begin:三郎 -  太郎: [AFTER] commonData=1, tlsData=1  太郎: [BEFORE] commonData=1, tlsData=1  太郎: [AFTER] commonData=2, tlsData=2  太郎: [BEFORE] commonData=2, tlsData=2  太郎: [AFTER] commonData=3, tlsData=3  - end:太郎 -  次郎: [BEFORE] commonData=3, tlsData=0  次郎: [AFTER] commonData=4, tlsData=1  次郎: [BEFORE] commonData=4, tlsData=1  次郎: [AFTER] commonData=5, tlsData=2  次郎: [BEFORE] commonData=5, tlsData=2  次郎: [AFTER] commonData=6, tlsData=3  - end:次郎 -  三郎: [BEFORE] commonData=6, tlsData=0  三郎: [AFTER] commonData=7, tlsData=1  三郎: [BEFORE] commonData=7, tlsData=1  三郎: [AFTER] commonData=8, tlsData=2  三郎: [BEFORE] commonData=8, tlsData=2  三郎: [AFTER] commonData=9, tlsData=3 ←全て正常にロックされたので、正しい処理結果になっている  - end:三郎 -  Critical Section \* 10000000 = 0.197762 sec ←1千万回ループによる処理時間計測 |

## 変数操作による排他制御

最初に誤ったサンプルとして普通の変数によるロック制御の方法を示した。

この誤りを正して、自前の変数操作で排他制御を行うことを再考する。

## 変数操作による排他制御：volatile型修飾子（誤ったロック）

前述の「スレッド間の変数操作の失敗」で説明している通り、volatile 型修飾子は、コンパイラによる変数操作の最適化を禁止し、スレッド間で変数の状態を確実に共有できるようにする。

しかし、volatile型修飾子はあくまでもコンパイルの仕方の指定である。アトミック操作を保証するものではないため、結局うまくいかない。

### Win32API版

スレッド作成と処理時間計測にWin32APIを用いる。ロック制御はスタティック変数によるフラグで行う。最初のサンプルとの違いは、変数「s\_lock」の型宣言のみである。

（Win32API版）volatile型修飾子付き変数によるロック制御のサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <Windows.h>  #include <Process.h>  //ロック取得用volatile型修飾子付き変数  static volatile int s\_lock = 0;  //共有データ  static int s\_commonData = 0;  //スレッド固有データ  \_\_declspec(thread) int s\_tlsData = 0;  //スレッド  unsigned int WINAPI threadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);  //開始  printf("- begin:%s -\n", name);  fflush(stdout);  //処理  for (int i = 0; i < 3; ++i)  {  //ロック取得用変数でロック取得待ち  while (s\_lock != 0){} s\_lock = 1;  //データ表示（前）  printf("%s: [BEFORE] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～4 msec）  Sleep(rand() % 5);  //データ書き戻し  s\_commonData = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("%s: [AFTER] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //ロック取得用変数でロック解除  s\_lock = 0;  //スレッド切り替えのためのスリープ  Sleep(0);  // //スレッド切り替え  // SwitchToThread();//OSに任せて再スケジューリング  // Yield();//廃止  }  //終了  printf("- end:%s -\n", name);  fflush(stdout);  return 0;  }  //テスト  int main(const int argc, const char\* argv[])  {  //スレッド作成  static const int THREAD\_NUM = 3;  unsigned int tid[THREAD\_NUM] = {};  HANDLE hThread[THREAD\_NUM] =  {  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "太郎", 0, &tid[0]),  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "次郎", 0, &tid[1]),  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "三郎", 0, &tid[2])  };  //スレッド終了待ち  WaitForMultipleObjects(THREAD\_NUM, hThread, true, INFINITE);  //スレッドハンドルクローズ  for (int i = 0; i < THREAD\_NUM; ++i)  {  CloseHandle(hThread[i]);  }  //比較用に空ループで時間を計測  {  LARGE\_INTEGER freq;  QueryPerformanceFrequency(&freq);  LARGE\_INTEGER begin;  QueryPerformanceCounter(&begin);  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  while (s\_lock != 0){} s\_lock = 1;  s\_lock = 0;  }  LARGE\_INTEGER end;  QueryPerformanceCounter(&end);  float duration = static\_cast<float>(static\_cast<double>(end.QuadPart - begin.QuadPart)  / static\_cast<double>(freq.QuadPart));  printf("Loop \* %d = %.6f sec\n", TEST\_TIMES, duration);  }  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:太郎 -  太郎: [BEFORE] commonData=0, tlsData=0  - begin:次郎 -  - begin:三郎 -  太郎: [AFTER] commonData=1, tlsData=1  三郎: [BEFORE] commonData=1, tlsData=0  次郎: [BEFORE] commonData=1, tlsData=0 ←ロックが失敗している  三郎: [AFTER] commonData=2, tlsData=1  次郎: [AFTER] commonData=2, tlsData=1  太郎: [BEFORE] commonData=2, tlsData=1  三郎: [BEFORE] commonData=2, tlsData=1 ←ロックが失敗している  太郎: [AFTER] commonData=3, tlsData=2  三郎: [AFTER] commonData=3, tlsData=2  次郎: [BEFORE] commonData=3, tlsData=1  三郎: [BEFORE] commonData=3, tlsData=2 ←ロックが失敗している  次郎: [AFTER] commonData=4, tlsData=2  太郎: [BEFORE] commonData=4, tlsData=2  三郎: [AFTER] commonData=4, tlsData=3  次郎: [BEFORE] commonData=4, tlsData=2 ←ロックが失敗している  - end:三郎 -  太郎: [AFTER] commonData=5, tlsData=3  - end:太郎 -  次郎: [AFTER] commonData=5, tlsData=3 ←4回ロックに失敗したので、正しい処理結果になっていない  - end:次郎 -  Loop \* 10000000 = 0.008377 sec ←1千万回ループによる処理時間計測 |

## 変数操作による排他制御：インラインアセンブラ

変数操作でロックを行うには、「変数の値を確認する」「変数のフラグを更新する」という二つの操作をアトミック操作（不可分操作）として実現する必要がある。このごくわずかな二つの処理の間に割り込まれると、ロックが成立しなくなる。

この対応には、1サイクル（CPUの実行の最小単位）でメモリ上の変数の確認と更新を同時に行う必要がある。C/C++言語の演算子では対応できないので、インラインアセンブラを使用する。

### Win32API版

スレッド作成と処理時間計測にWin32APIを用いる。なお、このロックの手法は、Wikipediaの「スピンロック」の解説を参考にしている。

（Win32API版）インラインアセンブラによるロック制御のサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <Windows.h>  #include <Process.h>  //ロック取得用変数  static int s\_lock = 0;  //共有データ  static int s\_commonData = 0;  //スレッド固有データ  \_\_declspec(thread) int s\_tlsData = 0;  //スレッド  unsigned int WINAPI threadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);  //開始  printf("- begin:%s -\n", name);  fflush(stdout);  //処理  for (int i = 0; i < 3; ++i)  {  //ロック取得  //※xchg のような命令を用いないとロックが不確実。  //　C言語の演算子だけでは不十分。  \_\_asm //ロック取得用インラインアセンブラ（x86系)  {  mov eax, 1 //フラグ更新準備  lock\_loop: //ループ用ラベル  xchg eax, [s\_lock] //変数とレジスタの値を交換  test eax, eax //レジスタの値をチェック  jnz lock\_loop //レジスタの値が 0 以外ならジャンプ  }  //データ表示（前）  printf("%s: [BEFORE] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～4 msec）  Sleep(rand() % 5);  //データ書き戻し  s\_commonData = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("%s: [AFTER] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //ロック解放  \_\_asm //ロック解放用インラインアセンブラ（x86系)  {  mov eax, 0 //フラグ更新準備  xchg eax, [s\_lock] //変数とレジスタの値を交換  }  //スレッド切り替えのためのスリープ  Sleep(0);  // //スレッド切り替え  // SwitchToThread();//OSに任せて再スケジューリング  // Yield();//廃止  }  //終了  printf("- end:%s -\n", name);  fflush(stdout);  return 0;  }  //テスト  int main(const int argc, const char\* argv[])  {  //スレッド作成  static const int THREAD\_NUM = 3;  unsigned int tid[THREAD\_NUM] = {};  HANDLE hThread[THREAD\_NUM] =  {  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "太郎", 0, &tid[0]),  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "次郎", 0, &tid[1]),  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "三郎", 0, &tid[2])  };    //スレッド終了待ち  WaitForMultipleObjects(THREAD\_NUM, hThread, true, INFINITE);  //スレッドハンドルクローズ  for (int i = 0; i < THREAD\_NUM; ++i)  {  CloseHandle(hThread[i]);  }  //インラインアセンブラでロックの取得と解放を大量に実行して時間を計測  {  LARGE\_INTEGER freq;  QueryPerformanceFrequency(&freq);  LARGE\_INTEGER begin;  QueryPerformanceCounter(&begin);  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  \_\_asm //ロック取得用インラインアセンブラ（x86系)  {  mov eax, 1 //フラグ更新準備  lock\_loop: //ループ用ラベル  xchg eax, [s\_lock] //変数とレジスタの値を交換  test eax, eax //レジスタの値をチェック  jnz lock\_loop //レジスタの値が 0 以外ならジャンプ  }  \_\_asm //ロック解放用インラインアセンブラ（x86系)  {  mov eax, 0 //フラグ更新準備  xchg eax, [s\_lock] //変数とレジスタの値を交換  }  }  LARGE\_INTEGER end;  QueryPerformanceCounter(&end);  float duration = static\_cast<float>(static\_cast<double>(end.QuadPart - begin.QuadPart)  / static\_cast<double>(freq.QuadPart));  printf("Volatile \* %d = %.6f sec\n", TEST\_TIMES, duration);  }  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:太郎 -  - begin:次郎 -  - begin:三郎 -  次郎: [BEFORE] commonData=0, tlsData=0  次郎: [AFTER] commonData=1, tlsData=1  三郎: [BEFORE] commonData=1, tlsData=0  三郎: [AFTER] commonData=2, tlsData=1  太郎: [BEFORE] commonData=2, tlsData=0  太郎: [AFTER] commonData=3, tlsData=1  三郎: [BEFORE] commonData=3, tlsData=1  三郎: [AFTER] commonData=4, tlsData=2  太郎: [BEFORE] commonData=4, tlsData=1  太郎: [AFTER] commonData=5, tlsData=2  三郎: [BEFORE] commonData=5, tlsData=2  三郎: [AFTER] commonData=6, tlsData=3  太郎: [BEFORE] commonData=6, tlsData=2  - end:三郎 -  太郎: [AFTER] commonData=7, tlsData=3  次郎: [BEFORE] commonData=7, tlsData=1  - end:太郎 -  次郎: [AFTER] commonData=8, tlsData=2  次郎: [BEFORE] commonData=8, tlsData=2  次郎: [AFTER] commonData=9, tlsData=3 ←全て正常にロックされたので、正しい処理結果になっている  - end:次郎 -  Assembler \* 10000000 = 0.126005 sec ←1千万回ループによる処理時間計測 |

## 変数操作による排他制御：アトミック操作（インターロック操作）

インラインアセンブラを用いた方法では、プログラムコードの汎用性に問題がある。

開発言語やSDKが提供する機能には、変数操作のアトミック操作を保証するものがある。

### Win32API版：インターロック操作

Win32APIにはインターロック操作のAPIが用意されている。

Win32API版インターロック操作によるロック制御のサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <Windows.h>  #include <Process.h>  //インターロック操作用変数  static LONG s\_lock = 0;  //共有データ  static int s\_commonData = 0;  //スレッド固有データ  \_\_declspec(thread) int s\_tlsData = 0;  //スレッド  unsigned int WINAPI threadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);  //開始  printf("- begin:%s -\n", name);  fflush(stdout);  //処理  for (int i = 0; i < 3; ++i)  {  //インターロック操作でロック取得待ち  while (InterlockedExchange(&s\_lock, 1) == 1){}    //データ表示（前）  printf("%s: [BEFORE] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～4 msec）  Sleep(rand() % 5);  //データ書き戻し  s\_commonData = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("%s: [AFTER] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //インターロック操作でロック解放  InterlockedExchange(&s\_lock, 0);    //スレッド切り替えのためのスリープ  Sleep(0);  // //スレッド切り替え  // SwitchToThread();//OSに任せて再スケジューリング  // Yield();//廃止  }  //終了  printf("- end:%s -\n", name);  fflush(stdout);  return 0;  }  //テスト  int main(const int argc, const char\* argv[])  {  //スレッド作成  static const int THREAD\_NUM = 3;  unsigned int tid[THREAD\_NUM] = {};  HANDLE hThread[THREAD\_NUM] =  {  (HANDLE)\_beginthreadex(nullptr, 0, threadFunc, "太郎", 0, &tid[0]),  (HANDLE)\_beginthreadex(nullptr, 0, threadFunc, "次郎", 0, &tid[1]),  (HANDLE)\_beginthreadex(nullptr, 0, threadFunc, "三郎", 0, &tid[2])  };    //スレッド終了待ち  WaitForMultipleObjects(THREAD\_NUM, hThread, true, INFINITE);  //スレッドハンドルクローズ  for (int i = 0; i < THREAD\_NUM; ++i)  {  CloseHandle(hThread[i]);  }  //インターロック操作でロックの取得と解放を大量に実行して時間を計測  {  LARGE\_INTEGER freq;  QueryPerformanceFrequency(&freq);  LARGE\_INTEGER begin;  QueryPerformanceCounter(&begin);  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  while (InterlockedExchange(&s\_lock, 1) == 1){}  InterlockedExchange(&s\_lock, 0);  }  LARGE\_INTEGER end;  QueryPerformanceCounter(&end);  float duration = static\_cast<float>(static\_cast<double>(end.QuadPart - begin.QuadPart)  / static\_cast<double>(freq.QuadPart));  printf("Interlocked \* %d = %.6f sec\n", TEST\_TIMES, duration);  }  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:太郎 -  太郎: [BEFORE] commonData=0, tlsData=0  - begin:次郎 -  - begin:三郎 -  太郎: [AFTER] commonData=1, tlsData=1  次郎: [BEFORE] commonData=1, tlsData=0  次郎: [AFTER] commonData=2, tlsData=1  三郎: [BEFORE] commonData=2, tlsData=0  三郎: [AFTER] commonData=3, tlsData=1  太郎: [BEFORE] commonData=3, tlsData=1  太郎: [AFTER] commonData=4, tlsData=2  次郎: [BEFORE] commonData=4, tlsData=1  次郎: [AFTER] commonData=5, tlsData=2  太郎: [BEFORE] commonData=5, tlsData=2  太郎: [AFTER] commonData=6, tlsData=3  次郎: [BEFORE] commonData=6, tlsData=2  - end:太郎 -  次郎: [AFTER] commonData=7, tlsData=3  - end:次郎 -  三郎: [BEFORE] commonData=7, tlsData=1  三郎: [AFTER] commonData=8, tlsData=2  三郎: [BEFORE] commonData=8, tlsData=2  三郎: [AFTER] commonData=9, tlsData=3 ←全て正常にロックされたので、正しい処理結果になっている  - end:三郎 -  Interlocked \* 10000000 = 0.127556 sec ←1千万回ループによる処理時間計測 |

### C++11版：アトミック操作

C++11にはアトミック操作のテンプレートクラスが用意されている。

Win32API版のインターロック操作に比べると、データ型の制約が少ないことや、サポートされている操作（演算）が多い点などが特徴。

C++11版アトミック操作によるロック制御のサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <thread>  #include <atomic>  #include <chrono> //時間計測用  #include <random> //乱数生成用  //アトミック操作用変数  static std::atomic<int> s\_lock = 0;  //共有データ  static int s\_commonData = 0;  //スレッド固有データ  \_\_declspec(thread) int s\_tlsData = 0;  //thread\_local int s\_tlsData = 0;//Visual C++ 2013 では thread\_local キーワードが使えない  //スレッド  void threadFunc(const char\* name)  {  //開始  printf("- begin:%s -\n", name);  fflush(stdout);  //乱数  std::random\_device seed\_gen;  std::mt19937 rnd(seed\_gen());  std::uniform\_int\_distribution<int> sleep\_time(0, 4);  //処理  for (int i = 0; i < 3; ++i)  {  //アトミック操作でロック取得待ち  while (s\_lock.exchange(1) == 1){}    //データ表示（前）  printf("%s: [BEFORE] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～4 msec）  std::this\_thread::sleep\_for(std::chrono::milliseconds(sleep\_time(rnd)));  //データ書き戻し  s\_commonData = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("%s: [AFTER] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //アトミック操作でロック解放  s\_lock.exchange(0);  //スレッド切り替えのためのスリープ  std::this\_thread::sleep\_for(std::chrono::milliseconds(0));  // //スレッド切り替え  // std::this\_thread::yield();//OSに任せて再スケジューリング  }  //終了  printf("- end:%s -\n", name);  fflush(stdout);  }  //テスト  int main(const int argc, const char\* argv[])  {  //スレッド作成  std::thread thread\_obj1 = std::thread(threadFunc, "太郎");  std::thread thread\_obj2 = std::thread(threadFunc, "次郎");  std::thread thread\_obj3 = std::thread(threadFunc, "三郎");  //スレッド終了待ち  thread\_obj1.join();  thread\_obj2.join();  thread\_obj3.join();  //アトミック操作でロックの取得と解放を大量に実行して時間を計測  {  auto begin = std::chrono::high\_resolution\_clock::now();  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  while (s\_lock.exchange(1) == 1){}  s\_lock.exchange(0);  }  auto end = std::chrono::high\_resolution\_clock::now();  auto duration = static\_cast<float>(static\_cast<double>(  std::chrono::duration\_cast<std::chrono::microseconds>(end - begin).count()) / 1000000.);  printf("Atomic \* %d = %.6f sec\n", TEST\_TIMES, duration);  }  return EXIT\_SUCCESS;  }  // End of file |

↓（実行結果）

|  |
| --- |
| - begin:太郎 -  太郎: [BEFORE] commonData=0, tlsData=0  - begin:次郎 -  - begin:三郎 -  太郎: [AFTER] commonData=1, tlsData=1  三郎: [BEFORE] commonData=1, tlsData=0  三郎: [AFTER] commonData=2, tlsData=1  次郎: [BEFORE] commonData=2, tlsData=0  次郎: [AFTER] commonData=3, tlsData=1  太郎: [BEFORE] commonData=3, tlsData=1  太郎: [AFTER] commonData=4, tlsData=2  三郎: [BEFORE] commonData=4, tlsData=1  三郎: [AFTER] commonData=5, tlsData=2  太郎: [BEFORE] commonData=5, tlsData=2  太郎: [AFTER] commonData=6, tlsData=3  次郎: [BEFORE] commonData=6, tlsData=1  - end:太郎 -  次郎: [AFTER] commonData=7, tlsData=2  三郎: [BEFORE] commonData=7, tlsData=2  三郎: [AFTER] commonData=8, tlsData=3  次郎: [BEFORE] commonData=8, tlsData=2  - end:三郎 -  次郎: [AFTER] commonData=9, tlsData=3 ←全て正常にロックされたので、正しい処理結果になっている  - end:次郎 -  Atomic \* 10000000 = 0.124007 sec ←1千万回ループによる処理時間計測 |

### C++11版：アトミック操作（フラグ型）

アトミック型はテンプレートのため、任意の型の変数を扱えるが、単なるフラグ操作の場合は、あらかじめ用意されているフラグ型を使用するとさらに効率的である。

なお、前述の「atd::atomic<int>」型と比べて、処理時間はほとんど変わらない。

C++11版アトミック操作（フラグ型使用）によるロック制御のサンプル（一部抜粋）：

|  |
| --- |
| …（略）…  //アトミック操作用変数  static std::atomic\_flag s\_lock = ATOMIC\_FLAG\_INIT;  …（略）…  //アトミック操作でロック取得待ち  while (s\_lock.test\_and\_set()){}  …（略）…  //アトミック操作でロック解放  s\_lock.clear();  …（略）…  //アトミック操作用変数初期化  s\_lock.clear();  …（略）… |

## 特殊な排他制御：リード・ライトロック

比較的新しいロック手法。

リードロックとライトロックを使い分けてロックする。

リードロック中は、他のリードロックは許可され、ライトロックはブロックされる。

ライトロック中は、他の全てのロックがブロックされる。

特殊な制御ゆえに、ミューテックスよりもパフォーマンスが落ちる。

使いどころとしては、共有データを作成・更新するスレッドが一つに対して、多数のスレッドがその情報を読み込むような場合に非常に有効。

並列処理環境では、分散処理が活用されるため、有効な場面が多い。

ゲームプログラミングにおいては、例えば、メインループと描画スレッドでリソースを排他制御するのに有効である。

### POSIXスレッドライブラリ版

POSIXスレッドライブラリ版のリード・ライトロック。

POSIXスレッドライブラリ版リード・ライトロックのサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <unistd.h>  #include <sys/time.h> //時間計測用  //リード／ライトロック  static pthread\_rwlock\_t s\_lock = PTHREAD\_RWLOCK\_INITIALIZER;  //共有データ  static int s\_commonData = 0;  //スレッド固有データ  \_\_thread int s\_tlsData = 0;  //書き込みスレッド  void\* threadFuncW(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);  //開始  printf("- begin:(W)%s -\n", name);  fflush(stdout);  //若干ランダムでスリープ（0～499 msec）  usleep((rand() % 500) \* 1000);  //処理  for (int i = 0; i < 3; ++i)  {  //ライトロック取得  pthread\_rwlock\_wrlock(&s\_lock);  // pthread\_rwlock\_trywrlock(&s\_lock);//ロックを取得できない時に他の処理を行いたい場合は  // //pthread\_rwlock\_trywrlock() を使用する    //データ表示（前）  printf("(W)%s: [BEFORE] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);    //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～499 msec）  usleep((rand() % 500) \* 1000);    //データ書き戻し  s\_commonData = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("(W)%s: [AFTER] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);    //ライトロック解放  pthread\_rwlock\_unlock(&s\_lock);  //若干ランダムでスリープ（0～499 msec）  usleep((rand() % 500) \* 1000);  }    //終了  printf("- end:(W)%s -\n", name);  fflush(stdout);  return 0;  }  //読み込みスレッド  void\* threadFuncR(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);  //開始  printf("- begin:(R)%s -\n", name);  fflush(stdout);  //若干ランダムでスリープ（0～499 msec）  usleep((rand() % 500) \* 1000);  //処理  for (int i = 0; i < 3; ++i)  {  //リードロック取得  pthread\_rwlock\_rdlock(&s\_lock);  // pthread\_rwlock\_tryrdlock(&s\_lock);//ロックが取得できない時に他の処理を行いたい場合は  // //pthread\_rwlock\_tryrdlock() を使用する  // pthread\_rwlock\_wrlock(&s\_lock);//【パフォーマンス検証用】ライトロックの場合    //データ表示（前）  printf("(R)%s: [BEFORE] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);    //若干ランダムでスリープ（0～499 msec）  usleep((rand() % 500) \* 1000);    //データ表示（後）  printf("(R)%s: [AFTER] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);    //リードロック解放  pthread\_rwlock\_unlock(&s\_lock);  //若干ランダムでスリープ（0～499 msec）  usleep((rand() % 500) \* 1000);  }    //終了  printf("- end:(R)%s -\n", name);  fflush(stdout);  return 0;  }  //テスト  int main(const int argc, const char\* argv[])  {  //リード／ライトロック生成  //※PTHREAD\_RWLOCK\_INITIALIZER で初期化している場合は不要  {  // pthread\_rwlockattr\_t attr;  // pthread\_rwlockattr\_init(&attr);  // pthread\_rwlockattr\_setpshared(&attr, PTHREAD\_PROCESS\_PRIVATE);//単独プロセス専用　※デフォルト  // pthread\_rwlockattr\_setpshared(&attr, PTHREAD\_PROCESS\_SHARED);//プロセス間で共有  // pthread\_rwlock\_init(&s\_lock, &attr);  }    //スレッド作成  static const int THREAD\_NUM = 6;  pthread\_t pth[THREAD\_NUM];  {  pthread\_attr\_t attr;  pthread\_attr\_init(&attr);  pthread\_attr\_setstacksize(&attr, 1024);//スタックサイズ指定  pthread\_create(&pth[0], &attr, threadFuncR, (void\*)"太郎");  pthread\_create(&pth[1], &attr, threadFuncR, (void\*)"次郎");  pthread\_create(&pth[2], &attr, threadFuncR, (void\*)"三郎");  usleep(1);  pthread\_create(&pth[3], &attr, threadFuncW, (void\*)"松子");  pthread\_create(&pth[4], &attr, threadFuncW, (void\*)"竹子");  pthread\_create(&pth[5], &attr, threadFuncW, (void\*)"梅子");  }    //スレッド終了待ち  {  struct timeval begin;  gettimeofday(&begin, NULL);    for(int i = 0; i < THREAD\_NUM; ++i)  {  pthread\_join(pth[i], NULL);  }    struct timeval end;  gettimeofday(&end, NULL);  struct timeval duration;  if( end.tv\_usec >= begin.tv\_usec)  {  duration.tv\_sec = end.tv\_sec - begin.tv\_sec;  duration.tv\_usec = end.tv\_usec - begin.tv\_usec;  }  else  {  duration.tv\_sec = end.tv\_sec - begin.tv\_sec - 1;  duration.tv\_usec = 1000000 - begin.tv\_usec + end.tv\_usec;  }  printf("Time = %d.%06d sec\n", duration.tv\_sec, duration.tv\_usec);  }    //リード／ライトロックの取得と解放を大量に実行して時間を計測  {  struct timeval begin;  gettimeofday(&begin, NULL);  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  pthread\_rwlock\_wrlock(&s\_lock);  pthread\_rwlock\_unlock(&s\_lock);  }  struct timeval end;  gettimeofday(&end, NULL);  struct timeval duration;  if( end.tv\_usec >= begin.tv\_usec)  {  duration.tv\_sec = end.tv\_sec - begin.tv\_sec;  duration.tv\_usec = end.tv\_usec - begin.tv\_usec;  }  else  {  duration.tv\_sec = end.tv\_sec - begin.tv\_sec - 1;  duration.tv\_usec = 1000000 - begin.tv\_usec + end.tv\_usec;  }  printf("Read-WriteLock \* %d = %d.%06d sec\n", TEST\_TIMES, duration.tv\_sec, duration.tv\_usec);  }    //リード／ライトロック破棄  //※PTHREAD\_RWLOCK\_INITIALIZER で初期化している場合は不要  {  // pthread\_rwlock\_destroy(&s\_lock);  }    return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:(R)三郎 -  - begin:(W)梅子 -  - begin:(R)次郎 -  - begin:(W)竹子 -  - begin:(R)太郎 -  - begin:(W)松子 -  (R)次郎: [BEFORE] commonData=0, tlsData=0  (R)太郎: [BEFORE] commonData=0, tlsData=0 ←リードロックは同時取得可  (R)三郎: [BEFORE] commonData=0, tlsData=0 ←リードロックは同時取得可  (R)三郎: [AFTER] commonData=0, tlsData=0  (R)次郎: [AFTER] commonData=0, tlsData=0  (R)太郎: [AFTER] commonData=0, tlsData=0 ←リードロック中はライトロックをブロック  (W)松子: [BEFORE] commonData=0, tlsData=0  (W)松子: [AFTER] commonData=1, tlsData=1 ←ライトロック中は完全に排他（完了）  (W)梅子: [BEFORE] commonData=1, tlsData=0  (W)梅子: [AFTER] commonData=2, tlsData=1 ←ライトロック中は完全に排他（完了）  (W)竹子: [BEFORE] commonData=2, tlsData=0  (W)竹子: [AFTER] commonData=3, tlsData=1 ←ライトロック中は完全に排他（完了）  (W)松子: [BEFORE] commonData=3, tlsData=1  (W)松子: [AFTER] commonData=4, tlsData=2 ←ライトロック中は完全に排他（完了）  (R)太郎: [BEFORE] commonData=4, tlsData=0  (R)次郎: [BEFORE] commonData=4, tlsData=0 ←リードロックは同時取得可  (R)三郎: [BEFORE] commonData=4, tlsData=0 ←リードロックは同時取得可  (R)三郎: [AFTER] commonData=4, tlsData=0  (R)太郎: [AFTER] commonData=4, tlsData=0  (R)次郎: [AFTER] commonData=4, tlsData=0 ←リードロック中はライトロックをブロック（完了）  (W)梅子: [BEFORE] commonData=4, tlsData=1  (W)梅子: [AFTER] commonData=5, tlsData=2 ←ライトロック中は完全に排他（完了）  (W)松子: [BEFORE] commonData=5, tlsData=2  (W)松子: [AFTER] commonData=6, tlsData=3 ←ライトロック中は完全に排他（完了）  (W)竹子: [BEFORE] commonData=6, tlsData=1  - end:(W)松子 -  (W)竹子: [AFTER] commonData=7, tlsData=2 ←ライトロック中は完全に排他（完了）  (W)梅子: [BEFORE] commonData=7, tlsData=2  (W)梅子: [AFTER] commonData=8, tlsData=3 ←ライトロック中は完全に排他（完了）  (R)次郎: [BEFORE] commonData=8, tlsData=0  (R)太郎: [BEFORE] commonData=8, tlsData=0 ←リードロックは同時取得可  (R)三郎: [BEFORE] commonData=8, tlsData=0 ←リードロックは同時取得可  - end:(W)梅子 -  (R)次郎: [AFTER] commonData=8, tlsData=0  (R)太郎: [AFTER] commonData=8, tlsData=0  - end:(R)太郎 -  (R)三郎: [AFTER] commonData=8, tlsData=0 ←リードロック中はライトロックをブロック（完了）  (W)竹子: [BEFORE] commonData=8, tlsData=2  - end:(R)三郎 -  - end:(R)次郎 -  (W)竹子: [AFTER] commonData=9, tlsData=3 ←全てのライトロックが正常にロックされたので、正しい処理結果になっている  - end:(W)竹子 -  Time = 3.720456 sec ←処理時間  Read-WriteLock \* 10000000 = 0.995709 sec ←処理時間 |

※リードロックをライトロックに変えて、普通のミューテックスのような排他制御を行った場合の実行結果

↓

|  |
| --- |
| - begin:(R)三郎 -  (R)三郎: [BEFORE] commonData=0, tlsData=0  - begin:(R)太郎 -  - begin:(W)松子 -  - begin:(W)梅子 -  - begin:(R)次郎 -  - begin:(W)竹子 -  (R)三郎: [AFTER] commonData=0, tlsData=0 ←完全に排他（完了）  (R)太郎: [BEFORE] commonData=0, tlsData=0  (R)太郎: [AFTER] commonData=0, tlsData=0 ←完全に排他（完了）  (W)松子: [BEFORE] commonData=0, tlsData=0  (W)松子: [AFTER] commonData=1, tlsData=1 ←完全に排他（完了）  (W)梅子: [BEFORE] commonData=1, tlsData=0  (W)梅子: [AFTER] commonData=2, tlsData=1 ←完全に排他（完了）  (R)次郎: [BEFORE] commonData=2, tlsData=0  (R)次郎: [AFTER] commonData=2, tlsData=0 ←完全に排他（完了）  (W)竹子: [BEFORE] commonData=2, tlsData=0  (W)竹子: [AFTER] commonData=3, tlsData=1 ←完全に排他（完了）  (R)三郎: [BEFORE] commonData=3, tlsData=0  (R)三郎: [AFTER] commonData=3, tlsData=0 ←完全に排他（完了）  (R)太郎: [BEFORE] commonData=3, tlsData=0  (R)太郎: [AFTER] commonData=3, tlsData=0 ←完全に排他（完了）  (W)松子: [BEFORE] commonData=3, tlsData=1  (W)松子: [AFTER] commonData=4, tlsData=2 ←完全に排他（完了）  (W)梅子: [BEFORE] commonData=4, tlsData=1  (W)梅子: [AFTER] commonData=5, tlsData=2 ←完全に排他（完了）  (R)次郎: [BEFORE] commonData=5, tlsData=0  (R)次郎: [AFTER] commonData=5, tlsData=0 ←完全に排他（完了）  (W)竹子: [BEFORE] commonData=5, tlsData=1  (W)竹子: [AFTER] commonData=6, tlsData=2 ←完全に排他（完了）  (R)三郎: [BEFORE] commonData=6, tlsData=0  (R)三郎: [AFTER] commonData=6, tlsData=0 ←完全に排他（完了）  (R)太郎: [BEFORE] commonData=6, tlsData=0  - end:(R)三郎 -  (R)太郎: [AFTER] commonData=6, tlsData=0 ←完全に排他（完了）  (W)松子: [BEFORE] commonData=6, tlsData=2  - end:(R)太郎 -  (W)松子: [AFTER] commonData=7, tlsData=3 ←完全に排他（完了）  (W)梅子: [BEFORE] commonData=7, tlsData=2  - end:(W)松子 -  (W)梅子: [AFTER] commonData=8, tlsData=3 ←完全に排他（完了）  (R)次郎: [BEFORE] commonData=8, tlsData=0  - end:(W)梅子 -  (R)次郎: [AFTER] commonData=8, tlsData=0 ←完全に排他（完了）  (W)竹子: [BEFORE] commonData=8, tlsData=2  - end:(R)次郎 -  (W)竹子: [AFTER] commonData=9, tlsData=3  - end:(W)竹子 -  Time = 5.350463 sec ←処理時間　※リード処理が並行動作しなくなった分遅くなっている  　(各処理のウェイトがランダムとはいえ、はっきり結果が違う)  Read-WriteLock \* 10000000 = 0.895791 sec ←1千万回ループによる処理時間計測 |

## 有限共有リソース使用権獲得

「ミューテックス」や「スピンロック」は、一つのスレッドだけが相互排他的に動作するためのものであった。

これから説明する「セマフォ」は、二つ以上の共有リソースに対して、それ以上のスレッドが排他的にアクセスするような場面で用いる。

具体的な用途としては、例えば、「多数のクライアントとの通信セッションを一つ一つのスレッドで確立したた状態」において、「負荷を抑えるために、三つまでしか並行で動作しないようにする」といったことが考えられる。

## 有限共有リソース使用権獲得：セマフォ

セマフォを扱う上での注意点として、ミューテックスのようなスレッド保護の観点はないため、再帰ロックには対応しない。

一つのスレッドが二つ以上のセマフォを獲得するようなプログラミングを行うと、簡単にデッドロックを起こすので要注意。

### SystemV版

古典的なSystemV系Unix（Linux含む）のセマフォ。プロセス間の排他制御に多く用いられる。

OSが管理するため、他のプロセスと重複しないキーを指定してセマフォを生成しなければならない。その為、自身のプログラム専用の何らかのファイルのキー（ファイルシステムによって付けられたユニークID）を用いるのが慣例となっている。

なお、プログラム中でセマフォを生成すると、端末からipcsコマンドでそれを確認することができる。

SystemV版セマフォのサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <unistd.h>  #include <sys/sem.h>  #include <fcntl.h>  #include <sys/time.h> //時間計測用  //セマフォ  static int s\_semaphore;  //ミューテックス  static pthread\_mutex\_t s\_mutex = PTHREAD\_MUTEX\_INITIALIZER;  //共有リソース  static const int COMMON\_RESOURCE\_NUM = 2;  static int s\_commonResource[COMMON\_RESOURCE\_NUM] = {};//複数の共有リソース  static bool s\_usingCommonResource[COMMON\_RESOURCE\_NUM] = {};//共有リソース使用中フラグ  //スレッド固有データ  \_\_thread int s\_tlsData = 0;  //スレッド  void\* threadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);  //開始  printf("- begin:%s -\n", name);  fflush(stdout);  //処理  for (int i = 0; i < 3; ++i)  {  //セマフォ取得  {  struct sembuf sb[1] = {{0, -1, 0}};//インデックス0番のセマフォのカウンタを -1 する  semop(s\_semaphore, sb, 1); //この時カウンタが 0ならブロックされる  //セマフォ操作データ sembuf の要素数は一つ  //取得失敗時にエラー終了させたい場合は  //sembuf::sem\_flag に IPC\_NOWAIT を指定する  //タイムアウトしたい場合は semtimedop() を使用する  }    //共有リソースを獲得  int index = 0;  pthread\_mutex\_lock(&s\_mutex);  for (; index < COMMON\_RESOURCE\_NUM; ++index)  {  if (!s\_usingCommonResource[index])  break;  }  s\_usingCommonResource[index] = true;  pthread\_mutex\_unlock(&s\_mutex);    //データ表示（前）  printf("%s: [BEFORE] commonResource[%d]=%d, tlsData=%d\n", name, index, s\_commonResource[index], s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonResource[index];  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～4 msec）  usleep((rand() % 5) \* 1000);    //データ書き戻し  s\_commonResource[index] = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("%s: [AFTER] commonResource[%d]=%d, tlsData=%d\n",  name, index, s\_commonResource[index], s\_tlsData);  fflush(stdout);  //共有リソースを解放  s\_usingCommonResource[index] = false;    //セマフォ解放  {  struct sembuf sb[1] = {{0, 1, 0}};//インデックス0番のセマフォのカウンタを +1 する  semop(s\_semaphore, sb, 1); //セマフォ操作データ sembuf の要素数は一つ  }    //スレッド切り替えのためのスリープ  usleep(0);  // //スレッド切り替え  // sched\_yield();//同じ優先度の他のスレッドに切り替え  }  //終了  printf("- end:%s -\n", name);  fflush(stdout);  return 0;  }  //テスト  int main(const int argc, const char\* argv[])  {  //セマフォ生成  {  key\_t key = ftok("share.cpp", 'S');//セマフォを一意に識別するためのキーとして、  //実在するファイルのユニークIDを利用  s\_semaphore = semget(key, 1, S\_IRWXU|IPC\_CREAT);//セマフォを一つ生成、全ユーザー＆グループ読み書き許可  semctl(s\_semaphore, 0, SETVAL, 0);//インデックス0番のセマフォの値を 0 にする（取得できない状態）  // //セマフォの初期値を設定する場合は、semctl() に SETVALと初期値を渡して呼び出す  // union semun arg;  // arg.val = COMMON\_RESOURCE\_NUM;  // semctl(s\_semaphore, 0, SETVAL, arg); //インデックス0番のセマフォの値を COMMON\_RESOURCE\_NUM にする  }    //スレッド作成  static const int THREAD\_NUM = 4;  pthread\_t pth[THREAD\_NUM];  {  pthread\_attr\_t attr;  pthread\_attr\_init(&attr);  pthread\_attr\_setstacksize(&attr, 1024);//スタックサイズ指定  pthread\_create(&pth[0], &attr, threadFunc, (void\*)"太郎");  pthread\_create(&pth[1], &attr, threadFunc, (void\*)"次郎");  pthread\_create(&pth[2], &attr, threadFunc, (void\*)"三郎");  pthread\_create(&pth[3], &attr, threadFunc, (void\*)"四朗");  }  //若干スリープ  usleep(10 \* 1000);  //共有リソース初期化  for (int i = 0; i < COMMON\_RESOURCE\_NUM; ++i)  {  s\_commonResource[i] = 1000 \* (i + 1);  }  //若干スリープ  usleep(10 \* 1000);  //共有リソースの使用を許可（全セマフォ解放）  {  for (int i = 0; i < COMMON\_RESOURCE\_NUM; ++i)  {  struct sembuf sb[1] = {{0, 1, 0}};//インデックス0番のセマフォのカウンタを +1 する  semop(s\_semaphore, sb, 1); //セマフォ操作データ sembuf の要素数は一つ  }  printf("Common-resources have been prepared. (num=%d)\n", COMMON\_RESOURCE\_NUM);  fflush(stdout);  }    //スレッド終了待ち  for (int i = 0; i < THREAD\_NUM; ++i)  {  pthread\_join(pth[i], NULL);  }    //セマフォの取得と解放を大量に実行して時間を計測  {  struct timeval begin;  gettimeofday(&begin, NULL);  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  {  struct sembuf sb = {0, -1, 0};  semop(s\_semaphore, &sb, 1);  }  {  struct sembuf sb = {0, 1, 0};  semop(s\_semaphore, &sb, 1);  }  }  struct timeval end;  gettimeofday(&end, NULL);  struct timeval duration;  if( end.tv\_usec >= begin.tv\_usec)  {  duration.tv\_sec = end.tv\_sec - begin.tv\_sec;  duration.tv\_usec = end.tv\_usec - begin.tv\_usec;  }  else  {  duration.tv\_sec = end.tv\_sec - begin.tv\_sec - 1;  duration.tv\_usec = 1000000 - begin.tv\_usec + end.tv\_usec;  }  printf("Semaphore \* %d = %d.%06d sec\n", TEST\_TIMES, duration.tv\_sec, duration.tv\_usec);  }    //セマフォ破棄  {  semctl(s\_semaphore, 0, IPC\_RMID, 0);//インデックス0番のセマフォを破棄  }    return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:三郎 -  - begin:太郎 -  - begin:次郎 -  - begin:四朗 -  Common-resources have been prepared. (num=2) ←セマフォが解放されるまで全てのスレッドがブロック  太郎: [BEFORE] commonResource[1]=2000, tlsData=0  三郎: [BEFORE] commonResource[0]=1000, tlsData=0 ←4つスレッドのうち、2つだけが稼働  三郎: [AFTER] commonResource[0]=1001, tlsData=1  次郎: [BEFORE] commonResource[0]=1001, tlsData=0 ←1つスレッドが終了すると、次のスレッドが動く  太郎: [AFTER] commonResource[1]=2001, tlsData=1 （常に同時に2つまで）  四朗: [BEFORE] commonResource[1]=2001, tlsData=0  四朗: [AFTER] commonResource[1]=2002, tlsData=1  三郎: [BEFORE] commonResource[1]=2002, tlsData=1  次郎: [AFTER] commonResource[0]=1002, tlsData=1  太郎: [BEFORE] commonResource[0]=1002, tlsData=1  太郎: [AFTER] commonResource[0]=1003, tlsData=2  四朗: [BEFORE] commonResource[0]=1003, tlsData=1  三郎: [AFTER] commonResource[1]=2003, tlsData=2  次郎: [BEFORE] commonResource[1]=2003, tlsData=1  四朗: [AFTER] commonResource[0]=1004, tlsData=2  太郎: [BEFORE] commonResource[0]=1004, tlsData=2  次郎: [AFTER] commonResource[1]=2004, tlsData=2  四朗: [BEFORE] commonResource[1]=2004, tlsData=2  太郎: [AFTER] commonResource[0]=1005, tlsData=3  三郎: [BEFORE] commonResource[0]=1005, tlsData=2  四朗: [AFTER] commonResource[1]=2005, tlsData=3  次郎: [BEFORE] commonResource[1]=2005, tlsData=2  - end:四朗 -  - end:太郎 -  次郎: [AFTER] commonResource[1]=2006, tlsData=3 ←全て正常にロックされたので、正しい処理結果になっている  三郎: [AFTER] commonResource[0]=1006, tlsData=3 （commonResource[0] と [1] が計12進行している）  - end:三郎 -  - end:次郎 -  Semaphore \* 10000000 = 8.186604 sec ←1千万回ループによる処理時間計測 |

↓※ipcsコマンド実行結果

|  |
| --- |
| $ ipcs  ------ 共有メモリセグメント --------  キー shmid 所有者 権限 バイト nattch 状態  ------ セマフォ配列 --------  キー semid 所有者 権限 nsems  0x5302020a 851973 user\_name 700 1 ←生成されているセマフォが確認できる  ------ メッセージキュー --------  キー msqid 所有者 権限 使用バイト数 メッセージ |

### POSIXスレッドライブラリ版（名前なし）

POSIXスレッドライブラリ版のセマフォ。

POSIXスレッドライブラリ版セマフォのサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <unistd.h>  #include <semaphore.h>  #include <sys/time.h> //時間計測用  //セマフォ  static sem\_t s\_semaphore;  //ミューテックス  static pthread\_mutex\_t s\_mutex = PTHREAD\_MUTEX\_INITIALIZER;  //共有リソース  static const int COMMON\_RESOURCE\_NUM = 2;  static int s\_commonResource[COMMON\_RESOURCE\_NUM] = {};//複数の共有リソース  static bool s\_usingCommonResource[COMMON\_RESOURCE\_NUM] = {};//共有リソース使用中フラグ  //スレッド固有データ  \_\_thread int s\_tlsData = 0;  //スレッド  void\* threadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);  //開始  printf("- begin:%s -\n", name);  fflush(stdout);  //処理  for (int i = 0; i < 3; ++i)  {  //セマフォ取得  sem\_wait(&s\_semaphore);  // sem\_trywait(&s\_semaphore);//ロックを取得できない時に他の処理を行いたい場合は  // //sem\_trywait() もしくは sem\_timedwait() を使用する    //共有リソースを獲得  int index = 0;  pthread\_mutex\_lock(&s\_mutex);  for (; index < COMMON\_RESOURCE\_NUM; ++index)  {  if (!s\_usingCommonResource[index])  break;  }  s\_usingCommonResource[index] = true;  pthread\_mutex\_unlock(&s\_mutex);    //データ表示（前）  printf("%s: [BEFORE] commonResource[%d]=%d, tlsData=%d\n", name, index, s\_commonResource[index], s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonResource[index];  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～4 msec）  usleep((rand() % 5) \* 1000);    //データ書き戻し  s\_commonResource[index] = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("%s: [AFTER] commonResource[%d]=%d, tlsData=%d\n",  name, index, s\_commonResource[index], s\_tlsData);  fflush(stdout);  //共有リソースを解放  s\_usingCommonResource[index] = false;    //セマフォ解放  sem\_post(&s\_semaphore);    //スレッド切り替えのためのスリープ  usleep(0);  // //スレッド切り替え  // sched\_yield();//同じ優先度の他のスレッドに切り替え  }  //終了  printf("- end:%s -\n", name);  fflush(stdout);  return 0;  }  //テスト  int main(const int argc, const char\* argv[])  {  //セマフォ生成  {  sem\_init(&s\_semaphore, 0, 0);//プロセス間で共有しないセマフォ  //セマフォの初期値を 0にする（取得できない状態）  }    //スレッド作成  static const int THREAD\_NUM = 4;  pthread\_t pth[THREAD\_NUM];  {  pthread\_attr\_t attr;  pthread\_attr\_init(&attr);  pthread\_attr\_setstacksize(&attr, 1024);//スタックサイズ指定  pthread\_create(&pth[0], &attr, threadFunc, (void\*)"太郎");  pthread\_create(&pth[1], &attr, threadFunc, (void\*)"次郎");  pthread\_create(&pth[2], &attr, threadFunc, (void\*)"三郎");  pthread\_create(&pth[3], &attr, threadFunc, (void\*)"四朗 ");  }  //若干スリープ  usleep(10 \* 1000);  //共有リソース初期化  for (int i = 0; i < COMMON\_RESOURCE\_NUM; ++i)  {  s\_commonResource[i] = 1000 \* (i + 1);  }  //若干スリープ  usleep(10 \* 1000);  //共有リソースの使用を許可（全セマフォ解放）  {  for (int i = 0; i < COMMON\_RESOURCE\_NUM; ++i)  {  sem\_post(&s\_semaphore);  }  printf("Common-resources have been prepared. (num=%d)\n", COMMON\_RESOURCE\_NUM);  fflush(stdout);  }    //スレッド終了待ち  for (int i = 0; i < THREAD\_NUM; ++i)  {  pthread\_join(pth[i], NULL);  }    //セマフォの取得と解放を大量に実行して時間を計測  {  struct timeval begin;  gettimeofday(&begin, NULL);  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  sem\_wait(&s\_semaphore);  sem\_post(&s\_semaphore);  }  struct timeval end;  gettimeofday(&end, NULL);  struct timeval duration;  if( end.tv\_usec >= begin.tv\_usec)  {  duration.tv\_sec = end.tv\_sec - begin.tv\_sec;  duration.tv\_usec = end.tv\_usec - begin.tv\_usec;  }  else  {  duration.tv\_sec = end.tv\_sec - begin.tv\_sec - 1;  duration.tv\_usec = 1000000 - begin.tv\_usec + end.tv\_usec;  }  printf("Semaphore \* %d = %d.%06d sec\n", TEST\_TIMES, duration.tv\_sec, duration.tv\_usec);  }    //セマフォ破棄  {  sem\_destroy(&s\_semaphore);  }    return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:四朗 -  - begin:三郎 -  - begin:太郎 -  - begin:次郎 -  Common-resources have been prepared. (num=2) ←セマフォが解放されるまで全てのスレッドがブロック  三郎: [BEFORE] commonResource[1]=2000, tlsData=0  四朗: [BEFORE] commonResource[0]=1000, tlsData=0 ←4つスレッドのうち、2つだけが稼働  四朗: [AFTER] commonResource[0]=1001, tlsData=1  太郎: [BEFORE] commonResource[0]=1001, tlsData=0 ←1つスレッドが終了すると、次のスレッドが動く  三郎: [AFTER] commonResource[1]=2001, tlsData=1 （常に同時に2つまで）  次郎: [BEFORE] commonResource[1]=2001, tlsData=0  太郎: [AFTER] commonResource[0]=1002, tlsData=1  四朗: [BEFORE] commonResource[0]=1002, tlsData=1  次郎: [AFTER] commonResource[1]=2002, tlsData=1  太郎: [BEFORE] commonResource[1]=2002, tlsData=1  太郎: [AFTER] commonResource[1]=2003, tlsData=2  三郎: [BEFORE] commonResource[1]=2003, tlsData=1  四朗: [AFTER] commonResource[0]=1003, tlsData=2  次郎: [BEFORE] commonResource[0]=1003, tlsData=1  三郎: [AFTER] commonResource[1]=2004, tlsData=2  太郎: [BEFORE] commonResource[1]=2004, tlsData=2  次郎: [AFTER] commonResource[0]=1004, tlsData=2  四朗: [BEFORE] commonResource[0]=1004, tlsData=2  四朗: [AFTER] commonResource[0]=1005, tlsData=3  三郎: [BEFORE] commonResource[0]=1005, tlsData=2  太郎: [AFTER] commonResource[1]=2005, tlsData=3  次郎: [BEFORE] commonResource[1]=2005, tlsData=2  - end:四朗 -  - end:太郎 -  三郎: [AFTER] commonResource[0]=1006, tlsData=3 ←全て正常にロックされたので、正しい処理結果になっている  次郎: [AFTER] commonResource[1]=2006, tlsData=3 （commonResource[0] と [1] が計12進行している）  - end:次郎 -  - end:三郎 -  Semaphore \* 10000000 = 0.607549 sec ←1千万回ループによる処理時間計測 |

### POSIXスレッドライブラリ版（名前付き）

POSIXスレッドライブラリ版の名前付きセマフォ。

基本的には名前無しセマフォと変わらないが、プロセス間での排他制御で場合に使用する。セマフォ生成時に、ファイルと同様のアクセス権を設定できる。

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <unistd.h>  #include <semaphore.h>  #include <fcntl.h>  #include <sys/time.h> //時間計測用  //共有セマフォ名  static const char\* COMMON\_SEMAPHORE\_NAME = "Common Semaphore";  //ミューテックス  static pthread\_mutex\_t s\_mutex = PTHREAD\_MUTEX\_INITIALIZER;  //共有リソース  static const int COMMON\_RESOURCE\_NUM = 2;  static int s\_commonResource[COMMON\_RESOURCE\_NUM] = {};//複数の共有リソース  static bool s\_usingCommonResource[COMMON\_RESOURCE\_NUM] = {};//共有リソース使用中フラグ  //スレッド固有データ  \_\_thread int s\_tlsData = 0;  //スレッド  void\* threadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);  //開始  printf("- begin:%s -\n", name);  fflush(stdout);  //名前付きセマフォオープン  sem\_t\* semaphore = sem\_open(COMMON\_SEMAPHORE\_NAME, 0);//すでに存在しているセマフォをオープン  //処理  for (int i = 0; i < 3; ++i)  {  //セマフォ取得  sem\_wait(semaphore);  // sem\_trywait(&s\_mutex);//取得できない時に他の処理を行いたい場合は  // //sem\_trywait() もしくは sem\_timedwait() を使用する    //共有リソースを獲得  int index = 0;  pthread\_mutex\_lock(&s\_mutex);  for (; index < COMMON\_RESOURCE\_NUM; ++index)  {  if (!s\_usingCommonResource[index])  break;  }  s\_usingCommonResource[index] = true;  pthread\_mutex\_unlock(&s\_mutex);    //データ表示（前）  printf("%s: [BEFORE] commonResource[%d]=%d, tlsData=%d\n", name, index, s\_commonResource[index], s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonResource[index];  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～4 msec）  usleep((rand() % 5) \* 1000);    //データ書き戻し  s\_commonResource[index] = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("%s: [AFTER] commonResource[%d]=%d, tlsData=%d\n",  name, index, s\_commonResource[index], s\_tlsData);  fflush(stdout);  //共有リソースを解放  s\_usingCommonResource[index] = false;    //セマフォ解放  sem\_post(semaphore);    //スレッド切り替えのためのスリープ  usleep(0);  // //スレッド切り替え  // sched\_yield();//同じ優先度の他のスレッドに切り替え  }    //セマフォクローズ  sem\_close(semaphore);  //終了  printf("- end:%s -\n", name);  fflush(stdout);  return 0;  }  //テスト  int main(const int argc, const char\* argv[])  {  //名前付きセマフォ生成  sem\_t\* semaphore = NULL;  {  semaphore = sem\_open(COMMON\_SEMAPHORE\_NAME, O\_CREAT, S\_IRWXU, 0);//全ユーザー＆グループ読み書き許可の  //セマフォを生成  //セマフォの初期値を 0にする  //（取得できない状態）  }    //スレッド作成  static const int THREAD\_NUM = 4;  pthread\_t pth[THREAD\_NUM];  {  pthread\_attr\_t attr;  pthread\_attr\_init(&attr);  pthread\_attr\_setstacksize(&attr, 1024);//スタックサイズ指定  pthread\_create(&pth[0], &attr, threadFunc, (void\*)"太郎");  pthread\_create(&pth[1], &attr, threadFunc, (void\*)"次郎");  pthread\_create(&pth[2], &attr, threadFunc, (void\*)"三郎");  pthread\_create(&pth[3], &attr, threadFunc, (void\*)"四朗");  }  //若干スリープ  usleep(10 \* 1000);  //共有リソース初期化  for (int i = 0; i < COMMON\_RESOURCE\_NUM; ++i)  {  s\_commonResource[i] = 1000 \* (i + 1);  }  //若干スリープ  usleep(10 \* 1000);  //共有リソースの使用を許可（全セマフォ解放）  {  for (int i = 0; i < COMMON\_RESOURCE\_NUM; ++i)  {  sem\_post(semaphore);  }  printf("Common-resources have been prepared. (num=%d)\n", COMMON\_RESOURCE\_NUM);  fflush(stdout);  }    //スレッド終了待ち  for (int i = 0; i < THREAD\_NUM; ++i)  {  pthread\_join(pth[i], NULL);  }    //セマフォの取得と解放を大量に実行して時間を計測  {  struct timeval begin;  gettimeofday(&begin, NULL);  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  sem\_wait(semaphore);  sem\_post(semaphore);  }  struct timeval end;  gettimeofday(&end, NULL);  struct timeval duration;  if( end.tv\_usec >= begin.tv\_usec)  {  duration.tv\_sec = end.tv\_sec - begin.tv\_sec;  duration.tv\_usec = end.tv\_usec - begin.tv\_usec;  }  else  {  duration.tv\_sec = end.tv\_sec - begin.tv\_sec - 1;  duration.tv\_usec = 1000000 - begin.tv\_usec + end.tv\_usec;  }  printf("Semaphore \* %d = %d.%06d sec\n", TEST\_TIMES, duration.tv\_sec, duration.tv\_usec);  }    //セマフォ破棄  {  sem\_close(semaphore);//クローズ  sem\_unlink(COMMON\_SEMAPHORE\_NAME);//破棄（きちんと破棄しないとプロセス終了後も残るので注意）  }    return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:四朗 -  - begin:太郎 -  - begin:三郎 -  - begin:次郎 -  Common-resources have been prepared. (num=2) ←セマフォが解放されるまで全てのスレッドがブロック  四朗: [BEFORE] commonResource[0]=1000, tlsData=0  太郎: [BEFORE] commonResource[1]=2000, tlsData=0 ←4つスレッドのうち、2つだけが稼働  太郎: [AFTER] commonResource[1]=2001, tlsData=1  三郎: [BEFORE] commonResource[1]=2001, tlsData=0 ←1つスレッドが終了すると、次のスレッドが動く  四朗: [AFTER] commonResource[0]=1001, tlsData=1 （常に同時に2つまで）  四朗: [BEFORE] commonResource[0]=1001, tlsData=1  三郎: [AFTER] commonResource[1]=2002, tlsData=1  太郎: [BEFORE] commonResource[1]=2002, tlsData=1  四朗: [AFTER] commonResource[0]=1002, tlsData=2  次郎: [BEFORE] commonResource[0]=1002, tlsData=0  次郎: [AFTER] commonResource[0]=1003, tlsData=1  三郎: [BEFORE] commonResource[0]=1003, tlsData=1  太郎: [AFTER] commonResource[1]=2003, tlsData=2  四朗: [BEFORE] commonResource[1]=2003, tlsData=2  三郎: [AFTER] commonResource[0]=1004, tlsData=2  太郎: [BEFORE] commonResource[0]=1004, tlsData=2  四朗: [AFTER] commonResource[1]=2004, tlsData=3  三郎: [BEFORE] commonResource[1]=2004, tlsData=2  - end:四朗 -  三郎: [AFTER] commonResource[1]=2005, tlsData=3  次郎: [BEFORE] commonResource[1]=2005, tlsData=1  太郎: [AFTER] commonResource[0]=1005, tlsData=3  - end:三郎 -  - end:太郎 -  次郎: [AFTER] commonResource[1]=2006, tlsData=2 ←全て正常にロックされたので、正しい処理結果になっている  次郎: [BEFORE] commonResource[0]=1005, tlsData=2 （commonResource[0] と [1] が計12進行している）  次郎: [AFTER] commonResource[0]=1006, tlsData=3  - end:次郎 -  Semaphore \* 10000000 = 0.568886 sec ←処理時間 |

### Win32API版（名前なし）

Win32API版のセマフォ。

Win32API版セマフォのサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <Windows.h>  #include <Process.h>  //セマフォハンドル  static HANDLE s\_hSemaphore = INVALID\_HANDLE\_VALUE;  //クリティカルセクション  static CRITICAL\_SECTION s\_lock;  //共有リソース  static const int COMMON\_RESOURCE\_NUM = 2;  static int s\_commonResource[COMMON\_RESOURCE\_NUM] = {};//複数の共有リソース  static bool s\_usingCommonResource[COMMON\_RESOURCE\_NUM] = {};//共有リソース使用中フラグ  //スレッド固有データ  \_\_declspec(thread) int s\_tlsData = 0;  //スレッド  unsigned int WINAPI threadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);  //開始  printf("- begin:%s -\n", name);  fflush(stdout);  //処理  for (int i = 0; i < 3; ++i)  {  //セマフォ取得  WaitForSingleObject(s\_hSemaphore, INFINITE);//取得できない時に他の処理を行いたい場合はタイムアウト値を  //指定する  //共有リソースを獲得  int index = 0;  EnterCriticalSection(&s\_lock);//クリティカルセクション取得  for (; index < COMMON\_RESOURCE\_NUM; ++index)  {  if (!s\_usingCommonResource[index])  break;  }  s\_usingCommonResource[index] = true;  LeaveCriticalSection(&s\_lock);//クリティカルセクション解放  //データ表示（前）  printf("%s: [BEFORE] commonResource[%d]=%d, tlsData=%d\n", name, index, s\_commonResource[index], s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonResource[index];  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～4 msec）  Sleep(rand() % 5);  //データ書き戻し  s\_commonResource[index] = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("%s: [AFTER] commonResource[%d]=%d, tlsData=%d\n",  name, index, s\_commonResource[index], s\_tlsData);  fflush(stdout);  //共有リソースを解放  s\_usingCommonResource[index] = false;    //セマフォ解放  LONG prev\_count;  ReleaseSemaphore(s\_hSemaphore, 1, &prev\_count);  //スレッド切り替えのためのスリープ  Sleep(0);  // //スレッド切り替え  // SwitchToThread();//OSに任せて再スケジューリング  // Yield();//廃止  }  //終了  printf("- end:%s -\n", name);  fflush(stdout);  return 0;  }  //テスト  int main(const int argc, const char\* argv[])  {  //セマフォ生成  {  s\_hSemaphore = CreateSemaphore(nullptr, 0, COMMON\_RESOURCE\_NUM, nullptr);//初期状態はセマフォを全て  //使用中にする    // //属性を指定して生成する場合  // SECURITY\_ATTRIBUTES attr = { sizeof(SECURITY\_ATTRIBUTES), nullptr, true };//子プロセスにハンドルを継承する  // SECURITY\_ATTRIBUTES attr = { sizeof(SECURITY\_ATTRIBUTES), nullptr, false };//子プロセスにハンドルを  // //継承しない　※デフォルト  // s\_hSemaphore = CreateSemaphore(&attr, 0, COMMON\_RESOURCE\_NUM, nullptr);//初期状態はセマフォを全て  // //使用中にする  }  //クリティカルセクション生成  InitializeCriticalSectionAndSpinCount(&s\_lock, 100);  //スレッド作成  static const int THREAD\_NUM = 4;  unsigned int tid[THREAD\_NUM] = {};  HANDLE hThread[THREAD\_NUM] =  {  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "太郎", 0, &tid[0]),  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "次郎", 0, &tid[1]),  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "三郎", 0, &tid[2]),  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "四郎", 0, &tid[3])  };  //若干スリープ  Sleep(10);  //共有リソース初期化  for (int i = 0; i < COMMON\_RESOURCE\_NUM; ++i)  {  s\_commonResource[i] = 1000 \* (i + 1);  }  //若干スリープ  Sleep(10);  //共有リソースの使用を許可（全セマフォ解放）  {  LONG prev\_count;  ReleaseSemaphore(s\_hSemaphore, COMMON\_RESOURCE\_NUM, &prev\_count);  printf("Common-resources have been prepared. (num=%d -> %d)\n", prev\_count, COMMON\_RESOURCE\_NUM);  fflush(stdout);  }  //スレッド終了待ち  WaitForMultipleObjects(THREAD\_NUM, hThread, true, INFINITE);  //スレッドハンドルクローズ  for (int i = 0; i < THREAD\_NUM; ++i)  {  CloseHandle(hThread[i]);  }  //セマフォの取得と解放を大量に実行して時間を計測  {  LARGE\_INTEGER freq;  QueryPerformanceFrequency(&freq);  LARGE\_INTEGER begin;  QueryPerformanceCounter(&begin);  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  WaitForSingleObject(s\_hSemaphore, INFINITE);  ReleaseSemaphore(s\_hSemaphore, 1, nullptr);  }  LARGE\_INTEGER end;  QueryPerformanceCounter(&end);  float duration = static\_cast<float>(static\_cast<double>(end.QuadPart - begin.QuadPart)  / static\_cast<double>(freq.QuadPart));  printf("Semaphore \* %d = %.6f sec\n", TEST\_TIMES, duration);  }  //クリティカルセクション破棄  DeleteCriticalSection(&s\_lock);    //セマフォ破棄  CloseHandle(s\_hSemaphore);  s\_hSemaphore = INVALID\_HANDLE\_VALUE;  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:太郎 -  - begin:次郎 -  - begin:三郎 -  - begin:四郎 -  Common-resources have been prepared. (num=0 -> 2) ←セマフォが解放されるまで全てのスレッドがブロック  次郎: [BEFORE] commonResource[0]=1000, tlsData=0  太郎: [BEFORE] commonResource[1]=2000, tlsData=0 ←4つスレッドのうち、2つだけが稼働  次郎: [AFTER] commonResource[0]=1001, tlsData=1  太郎: [AFTER] commonResource[1]=2001, tlsData=1  三郎: [BEFORE] commonResource[0]=1001, tlsData=0 ←1つスレッドが終了すると、次のスレッドが動く  四郎: [BEFORE] commonResource[1]=2001, tlsData=0 （常に同時に2つまで）  四郎: [AFTER] commonResource[1]=2002, tlsData=1  三郎: [AFTER] commonResource[0]=1002, tlsData=1  太郎: [BEFORE] commonResource[0]=1002, tlsData=1  次郎: [BEFORE] commonResource[1]=2002, tlsData=1  太郎: [AFTER] commonResource[0]=1003, tlsData=2  次郎: [AFTER] commonResource[1]=2003, tlsData=2  四郎: [BEFORE] commonResource[0]=1003, tlsData=1  三郎: [BEFORE] commonResource[1]=2003, tlsData=1  三郎: [AFTER] commonResource[1]=2004, tlsData=2  四郎: [AFTER] commonResource[0]=1004, tlsData=2  太郎: [BEFORE] commonResource[0]=1004, tlsData=2  次郎: [BEFORE] commonResource[1]=2004, tlsData=2  次郎: [AFTER] commonResource[1]=2005, tlsData=3  太郎: [AFTER] commonResource[0]=1005, tlsData=3  - end:太郎 -  - end:次郎 -  三郎: [BEFORE] commonResource[0]=1005, tlsData=2  四郎: [BEFORE] commonResource[1]=2005, tlsData=2  四郎: [AFTER] commonResource[1]=2006, tlsData=3 ←全て正常にロックされたので、正しい処理結果になっている  三郎: [AFTER] commonResource[0]=1006, tlsData=3 （commonResource[0] と [1] が計12進行している）  - end:四郎 -  - end:三郎 -  Semaphore \* 10000000 = 4.636328 sec ←1千万回ループによる処理時間計測 |

### Win32API版（名前付き）

Win32版の名前付きセマフォ。

基本的には名前無しセマフォと変わらないが、プロセス間での排他制御や、スレッドごとにセマフォのアクセス権を変えたい場合に使用する。

Win32API版名前付きセマフォのサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <Windows.h>  #include <Process.h>  //共有セマフォ名  static const char\* COMMON\_SEMAPHORE\_NAME = "Common Semaphore";  //共有ミューテックス名  static const char\* COMMON\_MUTEX\_NAME = "Common Mutex";  //共有リソース  static const int COMMON\_RESOURCE\_NUM = 2;  static int s\_commonResource[COMMON\_RESOURCE\_NUM] = {};//複数の共有リソース  static bool s\_usingCommonResource[COMMON\_RESOURCE\_NUM] = {};//共有リソース使用中フラグ  //スレッド固有データ  \_\_declspec(thread) int s\_tlsData = 0;  //スレッド  unsigned int WINAPI threadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);  //開始  printf("- begin:%s -\n", name);  fflush(stdout);  //名前付きセマフォオープン  HANDLE hSemaphore = OpenSemaphore(SEMAPHORE\_ALL\_ACCESS, false, COMMON\_SEMAPHORE\_NAME);  //SEMAPHORE\_ALL\_ACCESS 権限が必要（セマフォの獲得とリリースを行うため）  //名前付きミューテックスオープン  HANDLE hMutex = OpenMutex(SYNCHRONIZE, false, COMMON\_MUTEX\_NAME);  //処理  for (int i = 0; i < 3; ++i)  {  //セマフォ取得  WaitForSingleObject(hSemaphore, INFINITE);//取得できない時に他の処理を行いたい場合はタイムアウト値を  //指定する    //共有リソースを獲得  int index = 0;  WaitForSingleObject(hMutex, INFINITE);//ミューテックス取得  for (; index < COMMON\_RESOURCE\_NUM; ++index)  {  if (!s\_usingCommonResource[index])  break;  }  s\_usingCommonResource[index] = true;  ReleaseMutex(hMutex);//ミューテックス解放  //データ表示（前）  printf("%s: [BEFORE] commonResource[%d]=%d, tlsData=%d\n", name, index, s\_commonResource[index], s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonResource[index];  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～4 msec）  Sleep(rand() % 5);  //データ書き戻し  s\_commonResource[index] = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("%s: [AFTER] commonResource[%d]=%d, tlsData=%d\n",  name, index, s\_commonResource[index], s\_tlsData);  fflush(stdout);  //共有リソースを解放  s\_usingCommonResource[index] = false;    //セマフォ解放  LONG prev\_count;  ReleaseSemaphore(hSemaphore, 1, &prev\_count);  //スレッド切り替えのためのスリープ  Sleep(0);  // //スレッド切り替え  // SwitchToThread();//OSに任せて再スケジューリング  // Yield();//廃止  }  //名前付きミューテックスクローズ  CloseHandle(hMutex);  //名前付きセマフォクローズ  CloseHandle(hSemaphore);  //終了  printf("- end:%s -\n", name);  fflush(stdout);  return 0;  }  //テスト  int main(const int argc, const char\* argv[])  {  //セマフォハンドル  HANDLE hSemaphore = INVALID\_HANDLE\_VALUE;  //ミューテックスハンドル  HANDLE hMutex = INVALID\_HANDLE\_VALUE;  //名前付きセマフォ生成  {  hSemaphore = CreateSemaphore(nullptr, 0, COMMON\_RESOURCE\_NUM, COMMON\_SEMAPHORE\_NAME);  //初期状態はセマフォを全て使用中にする    // //属性を指定して生成する場合  // SECURITY\_ATTRIBUTES attr = { sizeof(SECURITY\_ATTRIBUTES), nullptr, true };//子プロセスにハンドルを継承する  // SECURITY\_ATTRIBUTES attr = { sizeof(SECURITY\_ATTRIBUTES), nullptr, false };//子プロセスにハンドルを  // //継承しない　※デフォルト  // hSemaphore = CreateSemaphore(&attr, 0, COMMON\_RESOURCE\_NUM, COMMON\_SEMAPHORE\_NAME);//初期状態はセマフォを  // //全て使用中にする  }  //名前付きミューテックス生成  {  hMutex = CreateMutex(nullptr, false, COMMON\_MUTEX\_NAME);  }  //スレッド作成  static const int THREAD\_NUM = 4;  unsigned int tid[THREAD\_NUM] = {};  HANDLE hThread[THREAD\_NUM] =  {  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "太郎", 0, &tid[0]),  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "次郎", 0, &tid[1]),  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "三郎", 0, &tid[2]),  (HANDLE)\_beginthreadex(nullptr, 1024, threadFunc, "四郎", 0, &tid[3])  };  //若干スリープ  Sleep(10);  //共有リソース初期化  for (int i = 0; i < COMMON\_RESOURCE\_NUM; ++i)  {  s\_commonResource[i] = 1000 \* (i + 1);  }  //若干スリープ  Sleep(10);  //共有リソースの使用を許可（全セマフォ解放）  {  LONG prev\_count;  ReleaseSemaphore(hSemaphore, COMMON\_RESOURCE\_NUM, &prev\_count);  printf("Common-resources have been prepared. (num=%d -> %d)\n", prev\_count, COMMON\_RESOURCE\_NUM);  fflush(stdout);  }  //スレッド終了待ち  WaitForMultipleObjects(THREAD\_NUM, hThread, true, INFINITE);  //スレッドハンドルクローズ  for (int i = 0; i < THREAD\_NUM; ++i)  {  CloseHandle(hThread[i]);  }  //セマフォの取得と解放を大量に実行して時間を計測  {  LARGE\_INTEGER freq;  QueryPerformanceFrequency(&freq);  LARGE\_INTEGER begin;  QueryPerformanceCounter(&begin);  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  WaitForSingleObject(hSemaphore, INFINITE);  ReleaseSemaphore(hSemaphore, 1, nullptr);  }  LARGE\_INTEGER end;  QueryPerformanceCounter(&end);  float time = static\_cast<float>(static\_cast<double>(end.QuadPart - begin.QuadPart)  / static\_cast<double>(freq.QuadPart));  printf("Semaphore \* %d = %.6f sec\n", TEST\_TIMES, time);  }  //名前付きミューテックス破棄  CloseHandle(hMutex);    //名前付きセマフォ破棄  CloseHandle(hSemaphore);  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:太郎 -  - begin:次郎 -  - begin:三郎 -  - begin:四郎 -  Common-resources have been prepared. (num=0 -> 2) ←セマフォが解放されるまで全てのスレッドがブロック  次郎: [BEFORE] commonResource[0]=1000, tlsData=0  太郎: [BEFORE] commonResource[1]=2000, tlsData=0 ←4つスレッドのうち、2つだけが稼働  太郎: [AFTER] commonResource[1]=2001, tlsData=1  次郎: [AFTER] commonResource[0]=1001, tlsData=1  三郎: [BEFORE] commonResource[1]=2001, tlsData=0 ←1つスレッドが終了すると、次のスレッドが動く  四郎: [BEFORE] commonResource[0]=1001, tlsData=0 （常に同時に2つまで）  三郎: [AFTER] commonResource[1]=2002, tlsData=1  四郎: [AFTER] commonResource[0]=1002, tlsData=1  太郎: [BEFORE] commonResource[1]=2002, tlsData=1  次郎: [BEFORE] commonResource[0]=1002, tlsData=1  次郎: [AFTER] commonResource[0]=1003, tlsData=2  太郎: [AFTER] commonResource[1]=2003, tlsData=2  三郎: [BEFORE] commonResource[0]=1003, tlsData=1  四郎: [BEFORE] commonResource[1]=2003, tlsData=1  四郎: [AFTER] commonResource[1]=2004, tlsData=2  三郎: [AFTER] commonResource[0]=1004, tlsData=2  次郎: [BEFORE] commonResource[1]=2004, tlsData=2  太郎: [BEFORE] commonResource[0]=1004, tlsData=2  次郎: [AFTER] commonResource[1]=2005, tlsData=3  太郎: [AFTER] commonResource[0]=1005, tlsData=3  - end:次郎 -  四郎: [BEFORE] commonResource[0]=1005, tlsData=2  三郎: [BEFORE] commonResource[1]=2005, tlsData=2  - end:太郎 -  三郎: [AFTER] commonResource[1]=2006, tlsData=3 ←全て正常にロックされたので、正しい処理結果になっている  四郎: [AFTER] commonResource[0]=1006, tlsData=3  - end:四郎 -  - end:三郎 -  Semaphore \* 10000000 = 4.787152 sec ←1千万回ループによる処理時間計測 |

## モニター

状態の変化を監視し、それまでスレッドをスリープして待機する手法を「モニター」と呼ぶ。

大きな括りとしては排他制御と同様だが、順序性のある処理をリレーするような時に使える。

例えば、メインループで描画用のデータを構築した後、描画スレッドに処理を受け渡すような場面で使える。描画スレッドは、データ構築完了の通知が来るまでスリープする。

## モニター：条件変数

最も一般的なモニターの手法は「条件変数」を用いる事である。

条件変数は、「通知があるまで待機する」仕組み。ミューテックスと、状態確認のための変数と組み合わせて使用するのが一般的であり、若干使用方法が難しい。

通知のタイミングで必ず待機している必要があることにも注意。

例えば、スレッドAの通知を待ってスレッドBが処理を開始したい場合、スレッドBが待機状態に入る前にスレッドAが通知を行ってしまうと、スレッドBが待機に入っても通知を確認することができない。このため、既に通知が行われていたら待機処理を実行しないように、気を付けてプログラミングしなければならない。

### POSIXスレッドライブラリ版

POSIXスレッドライブラリ版の条件変数。

POSIXスレッドライブラリ版条件変数のサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <unistd.h>  #include <sys/time.h> //時間計測用  //スレッド情報  static const int FOLLOW\_THREAD\_MAX = 10; //後続スレッド最大数  static volatile int s\_followThreadNum = 0; //後続スレッド数  static volatile int s\_followThreadNo = 0; //後続スレッド番号発番用  static volatile bool s\_IsQuirProiorThread = false;//先行スレッド終了フラグ  static pthread\_mutex\_t s\_followThreadMutex = PTHREAD\_MUTEX\_INITIALIZER;//後続スレッド番号発番用ミューテックス  //条件変数  //※条件変数は、必ず条件変数＋ミューテックス＋変数のセットで扱う  static pthread\_cond\_t s\_followCond[FOLLOW\_THREAD\_MAX]; //後続スレッド処理完了条件変数  static pthread\_mutex\_t s\_followCondMutex[FOLLOW\_THREAD\_MAX];//後続スレッド処理完了条件変数ミューテックス  static volatile bool s\_followFinished[FOLLOW\_THREAD\_MAX]; //後続スレッド処理完了フラグ  //共有データ  static int s\_commonData = 0;  //スレッド固有データ  \_\_thread int s\_tlsData = 0;  //【処理説明】  //先行スレッドが共有データを作成し、それが完了したら  //複数の後続スレッドがスタート。  //後続スレッドは共有データを読み込むだけのため、並列で動作。  //後続スレッドの処理が全て完了したら、また先行スレッドが稼働。  //以上を何度か繰り返し、先行スレッドが終了したら全スレッド終了。  //※メインループ⇒描画スレッドのようなリレー処理への応用を想定。  //先行スレッド  void\* priorThreadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);    //開始  printf("- begin:(P)%s -\n", name);  fflush(stdout);  //初期化  s\_IsQuirProiorThread = false;//先行処理終了フラグ  //処理  static const int LOOP\_COUNT\_MAX = 3;  int loop\_counter = 0;  while (1)  {  //全後続スレッド処理完了待ち  for (int i = 0; i < s\_followThreadNum; ++i)  {  pthread\_mutex\_lock(&s\_followCondMutex[i]);//ロック取得  while (!s\_followFinished[i])//待機終了条件を満たしていない場合  pthread\_cond\_wait(&s\_followCond[i], &s\_followCondMutex[i]);  //待機（ロック開放→待機→待機終了→ロック取得が行われる）  pthread\_mutex\_unlock(&s\_followCondMutex[i]);//ロック解放  //※待機をタイムアウトさせたい場合は pthread\_cond\_timedwait() 関数を用いる。  }  //ループカウンタ進行＆終了判定  if (loop\_counter++ == LOOP\_COUNT\_MAX)  {  //処理終了  printf("(P)%s: [QUIT]\n", name);  fflush(stdout);    //先行スレッド終了フラグ  s\_IsQuirProiorThread = true;    //先行スレッド処理完了：全待機スレッドを起床  for (int i = 0; i < s\_followThreadNum; ++i)  {  pthread\_mutex\_lock(&s\_followCondMutex[i]);//ロック取得  s\_followFinished[i] = false;//後続スレッドの処理完了状態を解除  pthread\_cond\_signal(&s\_followCond[i]);//解除待ちしているスレッドに通知  pthread\_mutex\_unlock(&s\_followCondMutex[i]);//ロック解放  }  break;  }  //データ表示（前）  printf("(P)%s: [BEFORE] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～499 msec）  usleep((rand() % 500) \* 1000);  //データ書き戻し  s\_commonData = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("(P)%s: [AFTER] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //先行スレッド処理完了：全待機スレッドを起床  for (int i = 0; i < s\_followThreadNum; ++i)  {  pthread\_mutex\_lock(&s\_followCondMutex[i]);//ロック取得  s\_followFinished[i] = false;//後続スレッドの処理完了状態を解除  pthread\_cond\_signal(&s\_followCond[i]);//解除待ちしているスレッドに通知  pthread\_mutex\_unlock(&s\_followCondMutex[i]);//ロック解放  //※一つの条件変数で多数のスレッドを待機させている場合は、  //　pthread\_cond\_broadcast() 関数も使える。  }  //スレッド切り替えのためのスリープ  usleep(0);  // //スレッド切り替え  // sched\_yield();//同じ優先度の他のスレッドに切り替え  }  //終了  printf("- end:(P)%s -\n", name);  fflush(stdout);  return 0;  }  //後続スレッド  void\* followThreadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);    //後続スレッド数カウントアップ  pthread\_mutex\_lock(&s\_followThreadMutex);  int thread\_no = s\_followThreadNo++;  pthread\_mutex\_unlock(&s\_followThreadMutex);  //開始  printf("- begin:(F)[%d]%s -\n", thread\_no, name);  fflush(stdout);  //後続スレッド処理完了：待機スレッドを起床  {  pthread\_mutex\_lock(&s\_followCondMutex[thread\_no]);//ロック取得  s\_followFinished[thread\_no] = true;//後続スレッドの処理完了状態をON  pthread\_cond\_signal(&s\_followCond[thread\_no]);//解除待ちしているスレッドに通知  pthread\_mutex\_unlock(&s\_followCondMutex[thread\_no]);//ロック解放  }  //処理  while (1)  {  //先行スレッド処理完了待ち  {  pthread\_mutex\_lock(&s\_followCondMutex[thread\_no]);//ロック取得  while (s\_followFinished[thread\_no])//待機終了条件を満たしていない場合  pthread\_cond\_wait(&s\_followCond[thread\_no], &s\_followCondMutex[thread\_no]);  //待機（ロック開放→待機→待機終了→ロック取得が行われる）  pthread\_mutex\_unlock(&s\_followCondMutex[thread\_no]);//ロック解放  //※待機をタイムアウトさせたい場合は pthread\_cond\_timedwait() 関数を用いる。  }  //終了確認  if (s\_IsQuirProiorThread)  {  //処理終了  printf("(F)[%d]%s: [QUIT]\n", thread\_no, name);  fflush(stdout);    //後続スレッド処理完了：待機スレッドを起床  {  pthread\_mutex\_lock(&s\_followCondMutex[thread\_no]);//ロック取得  s\_followFinished[thread\_no] = true;//後続スレッドの処理完了状態をON  pthread\_cond\_signal(&s\_followCond[thread\_no]);//解除待ちしているスレッドに通知  pthread\_mutex\_unlock(&s\_followCondMutex[thread\_no]);//ロック解放  }    break;  }  //データ表示（前）  printf("(F)[%d]%s: [BEFORE] commonData=%d, tlsData=%d\n", thread\_no, name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++tls\_data;  //若干ランダムでスリープ（0～500 msec）  usleep((rand() % 500) \* 1000);  //データ書き戻し  s\_tlsData = tls\_data;  //データ表示（後）  printf("(F)[%d]%s: [AFTER] commonData=%d, tlsData=%d\n", thread\_no, name, s\_commonData, s\_tlsData);  fflush(stdout);    //後続スレッド処理完了：待機スレッドを起床  {  pthread\_mutex\_lock(&s\_followCondMutex[thread\_no]);//ロック取得  s\_followFinished[thread\_no] = true;//後続スレッドの処理完了状態をON  pthread\_cond\_signal(&s\_followCond[thread\_no]);//解除待ちしているスレッドに通知  pthread\_mutex\_unlock(&s\_followCondMutex[thread\_no]);//ロック取得  //※一つの条件変数で多数のスレッドを待機させている場合は、  //　pthread\_cond\_broadcast() 関数も使える。  }  //スレッド切り替えのためのスリープ  usleep(0);  // //スレッド切り替え  // sched\_yield();//同じ優先度の他のスレッドに切り替え  }  //終了  printf("- end:(F)[%d]%s -\n", thread\_no, name);  fflush(stdout);  return 0;  }  //テスト  int main(const int argc, const char\* argv[])  {  //条件変数＆ミューテックス生成  //※PTHREAD\_COND\_INITIALIZER, PTHREAD\_MUTEX\_INITIALIZER で初期化している場合は不要  {  for(int i = 0; i < FOLLOW\_THREAD\_MAX; ++i)  {  pthread\_cond\_init(&s\_followCond[i], NULL);  pthread\_mutex\_init(&s\_followCondMutex[i], NULL);  }  }    //スレッド作成  static const int FOLLOW\_THREAD\_NUM = 5;  static const int THREAD\_NUM = 1 + FOLLOW\_THREAD\_NUM;  //static\_assert(FOLLOW\_THREAD\_NUM <= FOLLOW\_THREAD\_MAX, "FOLLOW\_THREAD\_NUM is over.");  s\_followThreadNum = FOLLOW\_THREAD\_NUM;  for (int i = 0; i < THREAD\_NUM - 1; ++i)  s\_followFinished[i] = true;  pthread\_t pth[THREAD\_NUM];  {  pthread\_attr\_t attr;  pthread\_attr\_init(&attr);  pthread\_attr\_setstacksize(&attr, 1024);//スタックサイズ指定  pthread\_create(&pth[0], &attr, priorThreadFunc, (void\*)"先行");  pthread\_create(&pth[1], &attr, followThreadFunc, (void\*)"後続01");  pthread\_create(&pth[2], &attr, followThreadFunc, (void\*)"後続02");  pthread\_create(&pth[3], &attr, followThreadFunc, (void\*)"後続03");  pthread\_create(&pth[4], &attr, followThreadFunc, (void\*)"後続04");  pthread\_create(&pth[5], &attr, followThreadFunc, (void\*)"後続05");  }    //スレッド終了待ち  for(int i = 0; i < THREAD\_NUM; ++i)  {  pthread\_join(pth[i], NULL);  }  //条件変数の取得と解放を大量に実行して時間を計測  {  struct timeval begin;  gettimeofday(&begin, NULL);  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  pthread\_mutex\_lock(&s\_followCondMutex[0]);  s\_followFinished[0] = true;  pthread\_cond\_signal(&s\_followCond[0]);  pthread\_mutex\_unlock(&s\_followCondMutex[0]);  }  struct timeval end;  gettimeofday(&end, NULL);  struct timeval duration;  if( end.tv\_usec >= begin.tv\_usec)  {  duration.tv\_sec = end.tv\_sec - begin.tv\_sec;  duration.tv\_usec = end.tv\_usec - begin.tv\_usec;  }  else  {  duration.tv\_sec = end.tv\_sec - begin.tv\_sec - 1;  duration.tv\_usec = 1000000 - begin.tv\_usec + end.tv\_usec;  }  printf("Cond-Sginal \* %d = %d.%06d sec\n", TEST\_TIMES, duration.tv\_sec, duration.tv\_usec);  }  //条件変数＆ミューテックス破棄  //※PTHREAD\_COND\_INITIALIZER, PTHREAD\_MUTEX\_INITIALIZER で初期化している場合は不要  {  for(int i = 0; i < FOLLOW\_THREAD\_MAX; ++i)  {  pthread\_cond\_destroy(&s\_followCond[i]);  pthread\_mutex\_destroy(&s\_followCondMutex[i]);  }  }    return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:(F)[0]後続04 -  - begin:(F)[1]後続03 -  - begin:(P)先行 -  (P)先行: [BEFORE] commonData=0, tlsData=0 ←先行スレッドが処理を開始  - begin:(F)[2]後続02 - 　（後続スレッドは、スレッド開始時から先行スレッドの処理終了を待機）  - begin:(F)[3]後続05 -  - begin:(F)[4]後続01 -  (P)先行: [AFTER] commonData=1, tlsData=1  (F)[0]後続04: [BEFORE] commonData=1, tlsData=0 ←先行スレッドの処理が終了すると、後続スレッドが一斉スタート  (F)[1]後続03: [BEFORE] commonData=1, tlsData=0  (F)[3]後続05: [BEFORE] commonData=1, tlsData=0  (F)[4]後続01: [BEFORE] commonData=1, tlsData=0  (F)[2]後続02: [BEFORE] commonData=1, tlsData=0  (F)[1]後続03: [AFTER] commonData=1, tlsData=1  (F)[4]後続01: [AFTER] commonData=1, tlsData=1  (F)[2]後続02: [AFTER] commonData=1, tlsData=1  (F)[0]後続04: [AFTER] commonData=1, tlsData=1  (F)[3]後続05: [AFTER] commonData=1, tlsData=1  (P)先行: [BEFORE] commonData=1, tlsData=1 ←後続スレッドの処理が全て終了すると、また先行スレッドがスタート  (P)先行: [AFTER] commonData=2, tlsData=2  (F)[0]後続04: [BEFORE] commonData=2, tlsData=1 ←先行スレッドの処理が終了すると、また後続スレッドが一斉スタート  (F)[1]後続03: [BEFORE] commonData=2, tlsData=1  (F)[3]後続05: [BEFORE] commonData=2, tlsData=1  (F)[4]後続01: [BEFORE] commonData=2, tlsData=1  (F)[2]後続02: [BEFORE] commonData=2, tlsData=1  (F)[2]後続02: [AFTER] commonData=2, tlsData=2  (F)[1]後続03: [AFTER] commonData=2, tlsData=2  (F)[4]後続01: [AFTER] commonData=2, tlsData=2  (F)[3]後続05: [AFTER] commonData=2, tlsData=2  (F)[0]後続04: [AFTER] commonData=2, tlsData=2  (P)先行: [BEFORE] commonData=2, tlsData=2 ←後続スレッドの処理が全て終了すると、また先行スレッドがスタート  (P)先行: [AFTER] commonData=3, tlsData=3  (F)[0]後続04: [BEFORE] commonData=3, tlsData=2 ←先行スレッドの処理が終了すると、また後続スレッドが一斉スタート  (F)[3]後続05: [BEFORE] commonData=3, tlsData=2  (F)[1]後続03: [BEFORE] commonData=3, tlsData=2  (F)[4]後続01: [BEFORE] commonData=3, tlsData=2  (F)[2]後続02: [BEFORE] commonData=3, tlsData=2  (F)[4]後続01: [AFTER] commonData=3, tlsData=3  (F)[0]後続04: [AFTER] commonData=3, tlsData=3  (F)[3]後続05: [AFTER] commonData=3, tlsData=3  (F)[1]後続03: [AFTER] commonData=3, tlsData=3  (F)[2]後続02: [AFTER] commonData=3, tlsData=3  (P)先行: [QUIT] ←先行スレッド終了  - end:(P)先行 -  (F)[3]後続05: [QUIT] ←先行スレッドの終了に反応して後続スレッドも終了  - end:(F)[3]後続05 -  (F)[0]後続04: [QUIT]  - end:(F)[0]後続04 -  (F)[4]後続01: [QUIT]  - end:(F)[4]後続01 -  (F)[1]後続03: [QUIT]  - end:(F)[1]後続03 -  (F)[2]後続02: [QUIT]  - end:(F)[2]後続02 -  Cond-Sginal \* 10000000 = 0.944458 sec ←1千万回ループによる処理時間計測 |

### C++11版

C++11版の条件変数。ミューテックスと変数を別途用意して組み合わせて使うのは、POSIXスレッドライブラリ版と同じ。

C++11版条件変数のサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <thread>  #include <mutex>  #include <condition\_variable>  #include <atomic>  #include <chrono> //時間計測用  #include <random> //乱数生成用  //スレッド情報  static const int FOLLOW\_THREAD\_MAX = 10; //後続スレッド最大数  static volatile int s\_followThreadNum = 0; //後続スレッド数  static std::atomic<int> s\_followThreadNo = 0; //後続スレッド番号発番用  static volatile bool s\_IsQuirProiorThread = false;//先行スレッド終了フラグ  //条件変数  //※条件変数は、必ず条件変数＋ミューテックス＋変数のセットで扱う  static std::condition\_variable s\_followCond[FOLLOW\_THREAD\_MAX];//後続スレッド処理完了条件変数  static std::mutex s\_followCondMutex[FOLLOW\_THREAD\_MAX]; //後続スレッド処理完了条件変数ミューテックス  static volatile bool s\_followFinished[FOLLOW\_THREAD\_MAX]; //後続スレッド処理完了フラグ  //共有データ  static int s\_commonData = 0;  //スレッド固有データ  \_\_declspec(thread) int s\_tlsData = 0;  //thread\_local int s\_tlsData = 0;//Visual C++ 2013 では thread\_local キーワードが使えない  //【処理説明】  //先行スレッドが共有データを作成し、それが完了したら  //複数の後続スレッドがスタート。  //後続スレッドは共有データを読み込むだけのため、並列で動作。  //後続スレッドの処理が全て完了したら、また先行スレッドが稼働。  //以上を何度か繰り返し、先行スレッドが終了したら全スレッド終了。  //※メインループ⇒描画スレッドのようなリレー処理への応用を想定。  //先行スレッド  void priorThreadFunc(const char\* name)  {  //開始  printf("- begin:(P)%s -\n", name);  fflush(stdout);  //乱数  std::random\_device seed\_gen;  std::mt19937 rnd(seed\_gen());  std::uniform\_int\_distribution<int> sleep\_time(0, 499);  //初期化  s\_IsQuirProiorThread = false;//先行処理終了フラグ  //処理  static const int LOOP\_COUNT\_MAX = 3;  int loop\_counter = 0;  while (1)  {  //全後続スレッド処理完了待ち  for (int i = 0; i < s\_followThreadNum; ++i)  {  {  std::unique\_lock<std::mutex> lock(s\_followCondMutex[i]);//ロック取得  //（ブロックから抜ける時に自動解放）  while (!s\_followFinished[i])//待機終了条件を満たしていない場合  s\_followCond[i].wait(lock);//待機（ロック開放→待機→待機終了→ロック取得が行われる）  //※待機をタイムアウトさせたい場合は .wait\_for() メソッドを用いる。  }  }  //ループカウンタ進行＆終了判定  if (loop\_counter++ == LOOP\_COUNT\_MAX)  {  //処理終了  printf("(P)%s: [QUIT]\n", name);  fflush(stdout);    //先行スレッド終了フラグ  s\_IsQuirProiorThread = true;    //先行スレッド処理完了：全待機スレッドを起床  for (int i = 0; i < s\_followThreadNum; ++i)  {  {  std::unique\_lock<std::mutex> lock(s\_followCondMutex[i]);//ロック取得（ブロックから  //抜ける時に自動解放）  s\_followFinished[i] = false;//後続スレッドの処理完了状態を解除  s\_followCond[i].notify\_one();//解除待ちしているスレッドに通知  }  }  break;  }  //データ表示（前）  printf("(P)%s: [BEFORE] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～499 msec）  std::this\_thread::sleep\_for(std::chrono::milliseconds(sleep\_time(rnd)));  //データ書き戻し  s\_commonData = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("(P)%s: [AFTER] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //先行スレッド処理完了：全待機スレッドを起床  for (int i = 0; i < s\_followThreadNum; ++i)  {  {  std::unique\_lock<std::mutex> lock(s\_followCondMutex[i]);//ロック取得（ブロックから  //抜ける時に自動解放）  s\_followFinished[i] = false;//後続スレッドの処理完了状態を解除  s\_followCond[i].notify\_one();//解除待ちしているスレッドに通知  //※一つの条件変数で多数のスレッドを待機させている場合は、  //　.notify\_all() メソッドも使える。  }  }  //スレッド切り替えのためのスリープ  std::this\_thread::sleep\_for(std::chrono::milliseconds(0));  // //スレッド切り替え  // std::this\_thread::yield();//OSに任せて再スケジューリング  }  //終了  printf("- end:(P)%s -\n", name);  fflush(stdout);  }  //後続スレッド  void followThreadFunc(const char\* name)  {  //後続スレッド数カウントアップ  const int thread\_no = s\_followThreadNo++;  //開始  printf("- begin:(F)[%d]%s -\n", thread\_no, name);  fflush(stdout);  //乱数  std::random\_device seed\_gen;  std::mt19937 rnd(seed\_gen());  std::uniform\_int\_distribution<int> sleep\_time(0, 499);  //後続スレッド処理完了：待機スレッドを起床  {  std::unique\_lock<std::mutex> lock(s\_followCondMutex[thread\_no]);//ロック取得（ブロックから  //抜ける時に自動解放）  s\_followFinished[thread\_no] = true;//後続スレッドの処理完了状態をON  s\_followCond[thread\_no].notify\_one();//解除待ちしているスレッドに通知  }  //処理  while (1)  {  //先行スレッド処理完了待ち  {  std::unique\_lock<std::mutex> lock(s\_followCondMutex[thread\_no]);//ロック取得（ブロックから  //抜ける時に自動解放）  while (s\_followFinished[thread\_no])//待機終了条件を満たしていない場合  s\_followCond[thread\_no].wait(lock);//待機（ロック開放→待機→待機終了→ロック取得が行われる）  //※待機をタイムアウトさせたい場合は .wait\_for() メソッドを用いる。  }  //終了確認  if (s\_IsQuirProiorThread)  {  //処理終了  printf("(F)[%d]%s: [QUIT]\n", thread\_no, name);  fflush(stdout);    //後続スレッド処理完了：待機スレッドを起床  {  std::unique\_lock<std::mutex> lock(s\_followCondMutex[thread\_no]);//ロック取得（ブロックから  //抜ける時に自動解放）  s\_followFinished[thread\_no] = true;//後続スレッドの処理完了状態をON  s\_followCond[thread\_no].notify\_one();//解除待ちしているスレッドに通知  }    break;  }  //データ表示（前）  printf("(F)[%d]%s: [BEFORE] commonData=%d, tlsData=%d\n", thread\_no, name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++tls\_data;  //若干ランダムでスリープ（0～500 msec）  std::this\_thread::sleep\_for(std::chrono::milliseconds(sleep\_time(rnd)));  //データ書き戻し  s\_tlsData = tls\_data;  //データ表示（後）  printf("(F)[%d]%s: [AFTER] commonData=%d, tlsData=%d\n", thread\_no, name, s\_commonData, s\_tlsData);  fflush(stdout);    //後続スレッド処理完了：待機スレッドを起床  {  std::unique\_lock<std::mutex> lock(s\_followCondMutex[thread\_no]);//ロック取得（ブロックから  //抜ける時に自動解放）  s\_followFinished[thread\_no] = true;//後続スレッドの処理完了状態をON  s\_followCond[thread\_no].notify\_one();//解除待ちしているスレッドに通知  //※一つの条件変数で多数のスレッドを待機させている場合は、  //　.notify\_all() メソッドも使える。  }  //スレッド切り替えのためのスリープ  std::this\_thread::sleep\_for(std::chrono::milliseconds(0));  // //スレッド切り替え  // std::this\_thread::yield();//OSに任せて再スケジューリング  }  //終了  printf("- end:(F)[%d]%s -\n", thread\_no, name);  fflush(stdout);  }  //テスト  int main(const int argc, const char\* argv[])  {  //スレッド作成  static const int FOLLOW\_THREAD\_NUM = 5;  static const int THREAD\_NUM = 1 + FOLLOW\_THREAD\_NUM;  static\_assert(FOLLOW\_THREAD\_NUM <= FOLLOW\_THREAD\_MAX, "FOLLOW\_THREAD\_NUM is over.");  s\_followThreadNum = FOLLOW\_THREAD\_NUM;  for (int i = 0; i < THREAD\_NUM - 1; ++i)  s\_followFinished[i] = true;  std::thread thread\_obj00 = std::thread(priorThreadFunc, "先行");  std::thread thread\_obj01 = std::thread(followThreadFunc, "後続01");  std::thread thread\_obj02 = std::thread(followThreadFunc, "後続02");  std::thread thread\_obj03 = std::thread(followThreadFunc, "後続03");  std::thread thread\_obj04 = std::thread(followThreadFunc, "後続04");  std::thread thread\_obj05 = std::thread(followThreadFunc, "後続05");  //スレッド終了待ち  thread\_obj00.join();  thread\_obj01.join();  thread\_obj02.join();  thread\_obj03.join();  thread\_obj04.join();  thread\_obj05.join();  //条件変数の取得と解放を大量に実行して時間を計測  {  auto begin = std::chrono::high\_resolution\_clock::now();  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  {  std::unique\_lock<std::mutex> lock(s\_followCondMutex[0]);  s\_followFinished[0] = true;  s\_followCond[0].notify\_one();  }  }  auto end = std::chrono::high\_resolution\_clock::now();  auto duration = static\_cast<float>(static\_cast<double>(  std::chrono::duration\_cast<std::chrono::microseconds>(end - begin).count()) / 1000000.);  printf("Cond-Sginal \* %d = %.6f sec\n", TEST\_TIMES, duration);  }  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:(P)先行 -  (P)先行: [BEFORE] commonData=0, tlsData=0 ←先行スレッドが処理を開始  - begin:(F)[0]後続01 -  - begin:(F)[1]後続02 -  - begin:(F)[2]後続03 -  - begin:(F)[3]後続04 -  - begin:(F)[4]後続05 -  (P)先行: [AFTER] commonData=1, tlsData=1 ←先行スレッドの処理が終了すると、後続スレッドが一斉スタート  (F)[0]後続01: [BEFORE] commonData=1, tlsData=0 　（後続スレッドは、スレッド開始時から先行スレッドの処理終了を待機）  (F)[1]後続02: [BEFORE] commonData=1, tlsData=0  (F)[2]後続03: [BEFORE] commonData=1, tlsData=0  (F)[3]後続04: [BEFORE] commonData=1, tlsData=0  (F)[4]後続05: [BEFORE] commonData=1, tlsData=0  (F)[0]後続01: [AFTER] commonData=1, tlsData=1  (F)[4]後続05: [AFTER] commonData=1, tlsData=1  (F)[2]後続03: [AFTER] commonData=1, tlsData=1  (F)[3]後続04: [AFTER] commonData=1, tlsData=1  (F)[1]後続02: [AFTER] commonData=1, tlsData=1  (P)先行: [BEFORE] commonData=1, tlsData=1 ←後続スレッドの処理が全て終了すると、また先行スレッドがスタート  (P)先行: [AFTER] commonData=2, tlsData=2  (F)[0]後続01: [BEFORE] commonData=2, tlsData=1 ←先行スレッドの処理が終了すると、また後続スレッドが一斉スタート  (F)[4]後続05: [BEFORE] commonData=2, tlsData=1  (F)[3]後続04: [BEFORE] commonData=2, tlsData=1  (F)[1]後続02: [BEFORE] commonData=2, tlsData=1  (F)[2]後続03: [BEFORE] commonData=2, tlsData=1  (F)[0]後続01: [AFTER] commonData=2, tlsData=2  (F)[3]後続04: [AFTER] commonData=2, tlsData=2  (F)[1]後続02: [AFTER] commonData=2, tlsData=2  (F)[4]後続05: [AFTER] commonData=2, tlsData=2  (F)[2]後続03: [AFTER] commonData=2, tlsData=2  (P)先行: [BEFORE] commonData=2, tlsData=2 ←後続スレッドの処理が全て終了すると、また先行スレッドがスタート  (P)先行: [AFTER] commonData=3, tlsData=3  (F)[0]後続01: [BEFORE] commonData=3, tlsData=2 ←先行スレッドの処理が終了すると、また後続スレッドが一斉スタート  (F)[1]後続02: [BEFORE] commonData=3, tlsData=2  (F)[2]後続03: [BEFORE] commonData=3, tlsData=2  (F)[3]後続04: [BEFORE] commonData=3, tlsData=2  (F)[4]後続05: [BEFORE] commonData=3, tlsData=2  (F)[1]後続02: [AFTER] commonData=3, tlsData=3  (F)[2]後続03: [AFTER] commonData=3, tlsData=3  (F)[3]後続04: [AFTER] commonData=3, tlsData=3  (F)[0]後続01: [AFTER] commonData=3, tlsData=3  (F)[4]後続05: [AFTER] commonData=3, tlsData=3  (P)先行: [QUIT] ←先行スレッド終了  (F)[0]後続01: [QUIT] ←先行スレッドの終了に反応して後続スレッドも終了  (F)[1]後続02: [QUIT]  (F)[4]後続05: [QUIT]  (F)[3]後続04: [QUIT]  - end:(F)[1]後続02 -  - end:(F)[0]後続01 -  - end:(F)[4]後続05 -  - end:(P)先行 -  - end:(F)[3]後続04 -  (F)[2]後続03: [QUIT]  - end:(F)[2]後続03 -  Cond-Sginal \* 10000000 = 0.758042 sec ←1千万回ループによる処理時間計測 |

## モニター：バリア

手軽なモニターの手法として、「バリア」というものがある。

スレッドの処理中にバリアを設定することにより、規定数のスレッドがバリアに到達するまで待機させることができる。

### POSIXスレッドライブラリ版

POSIXスレッドライブラリ版のバリア。

「条件変数」のサンプルを元に、バリア処理を追加している。「【バリア処理追加】」と書かれた箇所以外は元のプログラムと同じ。

POSIXスレッドライブラリ版バリアのサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <unistd.h>  #include <sys/time.h> //時間計測用  //スレッド情報  static const int FOLLOW\_THREAD\_MAX = 10; //後続スレッド最大数  static volatile int s\_followThreadNum = 0; //後続スレッド数  static volatile int s\_followThreadNo = 0; //後続スレッド番号発番用  static volatile bool s\_IsQuirProiorThread = false;//先行スレッド終了フラグ  static pthread\_mutex\_t s\_followThreadMutex = PTHREAD\_MUTEX\_INITIALIZER;//後続スレッド番号発番用ミューテックス  //条件変数  //※条件変数は、必ず条件変数＋ミューテックス＋変数のセットで扱う  static pthread\_cond\_t s\_followCond[FOLLOW\_THREAD\_MAX]; //後続スレッド処理完了条件変数  static pthread\_mutex\_t s\_followCondMutex[FOLLOW\_THREAD\_MAX];//後続スレッド処理完了条件変数ミューテックス  static volatile bool s\_followFinished[FOLLOW\_THREAD\_MAX]; //後続スレッド処理完了フラグ  //【バリア処理追加】バリア  static pthread\_barrier\_t s\_barrier;  //共有データ  static int s\_commonData = 0;  //スレッド固有データ  \_\_thread int s\_tlsData = 0;  //【処理説明】  //先行スレッドが共有データを作成し、それが完了したら  //複数の後続スレッドがスタート。  //後続スレッドは共有データを読み込むだけのため、並列で動作。  //後続スレッドの処理が全て完了したら、また先行スレッドが稼働。  //【バリア処理追加】後続スレッドの処理完了時は、バリアで足並みを揃える。  //以上を何度か繰り返し、先行スレッドが終了したら全スレッド終了。  //※メインループ⇒描画スレッドのようなリレー処理への応用を想定。  //先行スレッド  void\* priorThreadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);    //開始  printf("- begin:(P)%s -\n", name);  fflush(stdout);  //初期化  s\_IsQuirProiorThread = false;//先行処理終了フラグ  //処理  static const int LOOP\_COUNT\_MAX = 3;  int loop\_counter = 0;  while (1)  {  //全後続スレッド処理完了待ち  for (int i = 0; i < s\_followThreadNum; ++i)  {  pthread\_mutex\_lock(&s\_followCondMutex[i]);//ロック取得  while (!s\_followFinished[i])//待機終了条件を満たしていない場合  pthread\_cond\_wait(&s\_followCond[i], &s\_followCondMutex[i]);  //待機（ロック開放→待機→待機終了→ロック取得が行われる）  pthread\_mutex\_unlock(&s\_followCondMutex[i]);//ロック解放  //※待機をタイムアウトさせたい場合は pthread\_cond\_timedwait() 関数を用いる。  }  //ループカウンタ進行＆終了判定  if (loop\_counter++ == LOOP\_COUNT\_MAX)  {  //処理終了  printf("(P)%s: [QUIT]\n", name);  fflush(stdout);    //先行スレッド終了フラグ  s\_IsQuirProiorThread = true;    //先行スレッド処理完了：全待機スレッドを起床  for (int i = 0; i < s\_followThreadNum; ++i)  {  pthread\_mutex\_lock(&s\_followCondMutex[i]);//ロック取得  s\_followFinished[i] = false;//後続スレッドの処理完了状態を解除  pthread\_cond\_signal(&s\_followCond[i]);//解除待ちしているスレッドに通知  pthread\_mutex\_unlock(&s\_followCondMutex[i]);//ロック解放  }  break;  }  //データ表示（前）  printf("(P)%s: [BEFORE] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～499 msec）  usleep((rand() % 500) \* 1000);  //データ書き戻し  s\_commonData = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("(P)%s: [AFTER] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //先行スレッド処理完了：全待機スレッドを起床  for (int i = 0; i < s\_followThreadNum; ++i)  {  pthread\_mutex\_lock(&s\_followCondMutex[i]);//ロック取得  s\_followFinished[i] = false;//後続スレッドの処理完了状態を解除  pthread\_cond\_signal(&s\_followCond[i]);//解除待ちしているスレッドに通知  pthread\_mutex\_unlock(&s\_followCondMutex[i]);//ロック解放  //※一つの条件変数で多数のスレッドを待機させている場合は、  //　pthread\_cond\_broadcast() 関数も使える。  }  //スレッド切り替えのためのスリープ  usleep(0);  // //スレッド切り替え  // sched\_yield();//同じ優先度の他のスレッドに切り替え  }  //終了  printf("- end:(P)%s -\n", name);  fflush(stdout);  return 0;  }  //後続スレッド  void\* followThreadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);    //後続スレッド数カウントアップ  pthread\_mutex\_lock(&s\_followThreadMutex);  int thread\_no = s\_followThreadNo++;  pthread\_mutex\_unlock(&s\_followThreadMutex);  //開始  printf("- begin:(F)[%d]%s -\n", thread\_no, name);  fflush(stdout);  //後続スレッド処理完了：待機スレッドを起床  {  pthread\_mutex\_lock(&s\_followCondMutex[thread\_no]);//ロック取得  s\_followFinished[thread\_no] = true;//後続スレッドの処理完了状態をON  pthread\_cond\_signal(&s\_followCond[thread\_no]);//解除待ちしているスレッドに通知  pthread\_mutex\_unlock(&s\_followCondMutex[thread\_no]);//ロック解放  }  //処理  while (1)  {  //先行スレッド処理完了待ち  {  pthread\_mutex\_lock(&s\_followCondMutex[thread\_no]);//ロック取得  while (s\_followFinished[thread\_no])//待機終了条件を満たしていない場合  pthread\_cond\_wait(&s\_followCond[thread\_no], &s\_followCondMutex[thread\_no]);  //待機（ロック開放→待機→待機終了→ロック取得が行われる）  pthread\_mutex\_unlock(&s\_followCondMutex[thread\_no]);//ロック解放  //※待機をタイムアウトさせたい場合は pthread\_cond\_timedwait() 関数を用いる。  }  //終了確認  if (s\_IsQuirProiorThread)  {  //処理終了  printf("(F)[%d]%s: [QUIT]\n", thread\_no, name);  fflush(stdout);    //後続スレッド処理完了：待機スレッドを起床  {  pthread\_mutex\_lock(&s\_followCondMutex[thread\_no]);//ロック取得  s\_followFinished[thread\_no] = true;//後続スレッドの処理完了状態をON  pthread\_cond\_signal(&s\_followCond[thread\_no]);//解除待ちしているスレッドに通知  pthread\_mutex\_unlock(&s\_followCondMutex[thread\_no]);//ロック解放  }    break;  }  //【バリア処理追加】若干ランダムでスリープ（0～500 msec）  //バリア処理の効果を確認するために追加した処理  //各スレッドの処理タイミングがバラバラになるように  usleep((rand() % 500) \* 1000);  //データ表示（前）  printf("(F)[%d]%s: [BEFORE] commonData=%d, tlsData=%d\n", thread\_no, name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++tls\_data;  //若干ランダムでスリープ（0～500 msec）  usleep((rand() % 500) \* 1000);    //【バリア処理追加】以降、バリア処理の効果を確認するために追加した処理  //各スレッドの処理タイミングがバラバラになるようにしている    //中間表示1（後）  printf("(F)[%d]%s: [STEP1] commonData=%d, tlsData=%d\n", thread\_no, name, s\_commonData, s\_tlsData);  fflush(stdout);  //若干ランダムでスリープ（0～500 msec）  usleep((rand() % 500) \* 1000);    //中間表示2（後）  printf("(F)[%d]%s: [STEP2] commonData=%d, tlsData=%d\n", thread\_no, name, s\_commonData, s\_tlsData);  fflush(stdout);  //若干ランダムでスリープ（0～500 msec）  usleep((rand() % 500) \* 1000);    //中間表示3（後）  printf("(F)[%d]%s: [STEP3] commonData=%d, tlsData=%d\n", thread\_no, name, s\_commonData, s\_tlsData);  fflush(stdout);  //若干ランダムでスリープ（0～500 msec）  usleep((rand() % 500) \* 1000);  //【バリア処理追加】バリア処理の効果を確認するために追加した処理は、ここまで    //データ書き戻し  s\_tlsData = tls\_data;    //【バリア処理追加】バリアにより、全後続スレッドの処理タイミングを揃える  pthread\_barrier\_wait(&s\_barrier);    //データ表示（後）  printf("(F)[%d]%s: [AFTER] commonData=%d, tlsData=%d\n", thread\_no, name, s\_commonData, s\_tlsData);  fflush(stdout);    //後続スレッド処理完了：待機スレッドを起床  {  pthread\_mutex\_lock(&s\_followCondMutex[thread\_no]);//ロック取得  s\_followFinished[thread\_no] = true;//後続スレッドの処理完了状態をON  pthread\_cond\_signal(&s\_followCond[thread\_no]);//解除待ちしているスレッドに通知  pthread\_mutex\_unlock(&s\_followCondMutex[thread\_no]);//ロック取得  //※一つの条件変数で多数のスレッドを待機させている場合は、  //　pthread\_cond\_broadcast() 関数も使える。  }  //スレッド切り替えのためのスリープ  usleep(0);  // //スレッド切り替え  // sched\_yield();//同じ優先度の他のスレッドに切り替え  }  //終了  printf("- end:(F)[%d]%s -\n", thread\_no, name);  fflush(stdout);  return 0;  }  //テスト  int main(const int argc, const char\* argv[])  {  //条件変数＆ミューテックス生成  //※PTHREAD\_COND\_INITIALIZER, PTHREAD\_MUTEX\_INITIALIZER で初期化している場合は不要  {  for(int i = 0; i < FOLLOW\_THREAD\_MAX; ++i)  {  pthread\_cond\_init(&s\_followCond[i], NULL);  pthread\_mutex\_init(&s\_followCondMutex[i], NULL);  }  }    //スレッド作成  static const int FOLLOW\_THREAD\_NUM = 5;  static const int THREAD\_NUM = 1 + FOLLOW\_THREAD\_NUM;  //static\_assert(FOLLOW\_THREAD\_NUM <= FOLLOW\_THREAD\_MAX, "FOLLOW\_THREAD\_NUM is over.");  s\_followThreadNum = FOLLOW\_THREAD\_NUM;  pthread\_barrier\_init(&s\_barrier, 0, s\_followThreadNum);//【バリア処理追加】バリア初期化  for (int i = 0; i < THREAD\_NUM - 1; ++i)  s\_followFinished[i] = true;  pthread\_t pth[THREAD\_NUM];  {  pthread\_attr\_t attr;  pthread\_attr\_init(&attr);  pthread\_attr\_setstacksize(&attr, 1024);//スタックサイズ指定  pthread\_create(&pth[0], &attr, priorThreadFunc, (void\*)"先行");  pthread\_create(&pth[1], &attr, followThreadFunc, (void\*)"後続01");  pthread\_create(&pth[2], &attr, followThreadFunc, (void\*)"後続02");  pthread\_create(&pth[3], &attr, followThreadFunc, (void\*)"後続03");  pthread\_create(&pth[4], &attr, followThreadFunc, (void\*)"後続04");  pthread\_create(&pth[5], &attr, followThreadFunc, (void\*)"後続05");  }    //スレッド終了待ち  for(int i = 0; i < THREAD\_NUM; ++i)  {  pthread\_join(pth[i], NULL);  }  //条件変数の取得と解放を大量に実行して時間を計測  {  struct timeval begin;  gettimeofday(&begin, NULL);  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  pthread\_mutex\_lock(&s\_followCondMutex[0]);  s\_followFinished[0] = true;  pthread\_cond\_signal(&s\_followCond[0]);  pthread\_mutex\_unlock(&s\_followCondMutex[0]);  }  struct timeval end;  gettimeofday(&end, NULL);  struct timeval duration;  if( end.tv\_usec >= begin.tv\_usec)  {  duration.tv\_sec = end.tv\_sec - begin.tv\_sec;  duration.tv\_usec = end.tv\_usec - begin.tv\_usec;  }  else  {  duration.tv\_sec = end.tv\_sec - begin.tv\_sec - 1;  duration.tv\_usec = 1000000 - begin.tv\_usec + end.tv\_usec;  }  printf("Cond-Sginal \* %d = %d.%06d sec\n", TEST\_TIMES, duration.tv\_sec, duration.tv\_usec);  }  //【バリア処理追加】バリア破棄  pthread\_barrier\_destroy(&s\_barrier);    //条件変数＆ミューテックス破棄  //※PTHREAD\_COND\_INITIALIZER, PTHREAD\_MUTEX\_INITIALIZER で初期化している場合は不要  {  for(int i = 0; i < FOLLOW\_THREAD\_MAX; ++i)  {  pthread\_cond\_destroy(&s\_followCond[i]);  pthread\_mutex\_destroy(&s\_followCondMutex[i]);  }  }    return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:(F)[0]後続03 -  - begin:(P)先行 -  (P)先行: [BEFORE] commonData=0, tlsData=0  - begin:(F)[1]後続02 -  - begin:(F)[2]後続05 -  - begin:(F)[3]後続04 -  - begin:(F)[4]後続01 -  (P)先行: [AFTER] commonData=1, tlsData=1  (F)[1]後続02: [BEFORE] commonData=1, tlsData=0  (F)[4]後続01: [BEFORE] commonData=1, tlsData=0  (F)[0]後続03: [BEFORE] commonData=1, tlsData=0  (F)[2]後続05: [BEFORE] commonData=1, tlsData=0  (F)[3]後続04: [BEFORE] commonData=1, tlsData=0  (F)[0]後続03: [STEP1] commonData=1, tlsData=0  (F)[0]後続03: [STEP2] commonData=1, tlsData=0  (F)[1]後続02: [STEP1] commonData=1, tlsData=0 ←各後続スレッドの進行が入り乱れている状態  (F)[0]後続03: [STEP3] commonData=1, tlsData=0  (F)[1]後続02: [STEP2] commonData=1, tlsData=0  (F)[3]後続04: [STEP1] commonData=1, tlsData=0  (F)[4]後続01: [STEP1] commonData=1, tlsData=0  (F)[2]後続05: [STEP1] commonData=1, tlsData=0  (F)[3]後続04: [STEP2] commonData=1, tlsData=0  (F)[2]後続05: [STEP2] commonData=1, tlsData=0  (F)[3]後続04: [STEP3] commonData=1, tlsData=0  (F)[1]後続02: [STEP3] commonData=1, tlsData=0  (F)[2]後続05: [STEP3] commonData=1, tlsData=0  (F)[4]後続01: [STEP2] commonData=1, tlsData=0  (F)[4]後続01: [STEP3] commonData=1, tlsData=0  (F)[0]後続03: [AFTER] commonData=1, tlsData=1 ←[AFTER]の前にバリアがあるので、このタイミングで足並みが揃う  (F)[3]後続04: [AFTER] commonData=1, tlsData=1  (F)[2]後続05: [AFTER] commonData=1, tlsData=1  (F)[1]後続02: [AFTER] commonData=1, tlsData=1  (F)[4]後続01: [AFTER] commonData=1, tlsData=1  (P)先行: [BEFORE] commonData=1, tlsData=1  (P)先行: [AFTER] commonData=2, tlsData=2  (F)[2]後続05: [BEFORE] commonData=2, tlsData=1  (F)[2]後続05: [STEP1] commonData=2, tlsData=1  (F)[1]後続02: [BEFORE] commonData=2, tlsData=1 ←各後続スレッドの進行が入り乱れている状態（2回目）  (F)[3]後続04: [BEFORE] commonData=2, tlsData=1  (F)[2]後続05: [STEP2] commonData=2, tlsData=1  (F)[1]後続02: [STEP1] commonData=2, tlsData=1  (F)[0]後続03: [BEFORE] commonData=2, tlsData=1  (F)[3]後続04: [STEP1] commonData=2, tlsData=1  (F)[0]後続03: [STEP1] commonData=2, tlsData=1  (F)[3]後続04: [STEP2] commonData=2, tlsData=1  (F)[4]後続01: [BEFORE] commonData=2, tlsData=1  (F)[0]後続03: [STEP2] commonData=2, tlsData=1  (F)[2]後続05: [STEP3] commonData=2, tlsData=1  (F)[1]後続02: [STEP2] commonData=2, tlsData=1  (F)[1]後続02: [STEP3] commonData=2, tlsData=1  (F)[3]後続04: [STEP3] commonData=2, tlsData=1  (F)[4]後続01: [STEP1] commonData=2, tlsData=1  (F)[0]後続03: [STEP3] commonData=2, tlsData=1  (F)[4]後続01: [STEP2] commonData=2, tlsData=1  (F)[4]後続01: [STEP3] commonData=2, tlsData=1  (F)[2]後続05: [AFTER] commonData=2, tlsData=2 ←[AFTER]の前にバリアがあるので、このタイミングで足並みが揃う(2回目)  (F)[4]後続01: [AFTER] commonData=2, tlsData=2  (F)[3]後続04: [AFTER] commonData=2, tlsData=2  (F)[0]後続03: [AFTER] commonData=2, tlsData=2  (F)[1]後続02: [AFTER] commonData=2, tlsData=2  (P)先行: [BEFORE] commonData=2, tlsData=2  (P)先行: [AFTER] commonData=3, tlsData=3  (F)[0]後続03: [BEFORE] commonData=3, tlsData=2  (F)[4]後続01: [BEFORE] commonData=3, tlsData=2  (F)[3]後続04: [BEFORE] commonData=3, tlsData=2  (F)[2]後続05: [BEFORE] commonData=3, tlsData=2  (F)[1]後続02: [BEFORE] commonData=3, tlsData=2  (F)[1]後続02: [STEP1] commonData=3, tlsData=2  (F)[4]後続01: [STEP1] commonData=3, tlsData=2  (F)[0]後続03: [STEP1] commonData=3, tlsData=2  (F)[0]後続03: [STEP2] commonData=3, tlsData=2  (F)[3]後続04: [STEP1] commonData=3, tlsData=2 ←各後続スレッドの進行が入り乱れている状態（3回目）  (F)[2]後続05: [STEP1] commonData=3, tlsData=2  (F)[1]後続02: [STEP2] commonData=3, tlsData=2  (F)[3]後続04: [STEP2] commonData=3, tlsData=2  (F)[4]後続01: [STEP2] commonData=3, tlsData=2  (F)[0]後続03: [STEP3] commonData=3, tlsData=2  (F)[3]後続04: [STEP3] commonData=3, tlsData=2  (F)[2]後続05: [STEP2] commonData=3, tlsData=2  (F)[1]後続02: [STEP3] commonData=3, tlsData=2  (F)[4]後続01: [STEP3] commonData=3, tlsData=2  (F)[2]後続05: [STEP3] commonData=3, tlsData=2  (F)[3]後続04: [AFTER] commonData=3, tlsData=3 ←[AFTER]の前にバリアがあるので、このタイミングで足並みが揃う(3回目)  (F)[2]後続05: [AFTER] commonData=3, tlsData=3  (F)[4]後続01: [AFTER] commonData=3, tlsData=3  (F)[1]後続02: [AFTER] commonData=3, tlsData=3  (F)[0]後続03: [AFTER] commonData=3, tlsData=3  (P)先行: [QUIT]  - end:(P)先行 -  (F)[1]後続02: [QUIT]  - end:(F)[1]後続02 -  (F)[4]後続01: [QUIT]  - end:(F)[4]後続01 -  (F)[3]後続04: [QUIT]  - end:(F)[3]後続04 -  (F)[0]後続03: [QUIT]  - end:(F)[0]後続03 -  (F)[2]後続05: [QUIT]  - end:(F)[2]後続05 -  Cond-Sginal \* 10000000 = 0.937633 sec |

## モニター：イベント

Win32APIでモニターを行うには、「イベント」を使用する。

イベントは条件変数に比べてシンプルに使用することができるが、単なる通知（「シグナル状態」と呼ぶ）しかできないので、スレッド間で相互に通知しあうような場合はそれぞれのイベントを設ける必要がある。（条件変数は任意の変数で状態を管理するので共用できる）

### Win32API版（名前なし）

Win32API版のイベント。

Win32API版イベントのサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <Windows.h>  #include <Process.h>  //スレッド情報  static const int FOLLOW\_THREAD\_MAX = 10; //後続スレッド最大数  static volatile LONG s\_followThreadNum = 0; //後続スレッド数  static volatile LONG s\_followThreadNo = 0; //後続スレッド番号発番用  static volatile bool s\_IsQuirProiorThread = false;//先行スレッド終了フラグ  //イベントハンドル  static HANDLE s\_hPriorEvent[FOLLOW\_THREAD\_MAX]; //先行スレッド処理完了イベント  static HANDLE s\_hFollowEvent[FOLLOW\_THREAD\_MAX];//後続スレッド処理完了イベント  //共有データ  static int s\_commonData = 0;  //スレッド固有データ  \_\_declspec(thread) int s\_tlsData = 0;  //【処理説明】  //先行スレッドが共有データを作成し、それが完了したら  //複数の後続スレッドがスタート。  //後続スレッドは共有データを読み込むだけのため、並列で動作。  //後続スレッドの処理が全て完了したら、また先行スレッドが稼働。  //以上を何度か繰り返し、先行スレッドが終了したら全スレッド終了。  //※メインループ⇒描画スレッドのようなリレー処理への応用を想定。  //先行スレッド  unsigned int WINAPI priorThreadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);  //開始  printf("- begin:(P)%s -\n", name);  fflush(stdout);  //初期化  s\_IsQuirProiorThread = false;//先行処理終了フラグ  //処理  static const int LOOP\_COUNT\_MAX = 3;  int loop\_counter = 0;  while (1)  {  //全後続スレッド処理完了待ち  WaitForMultipleObjects(s\_followThreadNum, s\_hFollowEvent, true, INFINITE);  //待機が完了しない時に他の処理を行いたい場合はタイムアウト値を指定する  //ループカウンタ進行＆終了判定  if (loop\_counter++ == LOOP\_COUNT\_MAX)  {  //処理終了  printf("(P)%s: [QUIT]\n", name);  fflush(stdout);    //先行スレッド終了フラグ  s\_IsQuirProiorThread = true;    //先行スレッド処理完了イベントセット  for (int i = 0; i < s\_followThreadNum; ++i)  SetEvent(s\_hPriorEvent[i]);  break;  }  //データ表示（前）  printf("(P)%s: [BEFORE] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～499 msec）  Sleep(rand() % 500);  //データ書き戻し  s\_commonData = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("(P)%s: [AFTER] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //先行スレッド処理完了イベントセット  for (int i = 0; i < s\_followThreadNum; ++i)  {  SetEvent(s\_hPriorEvent[i]);  }  //※PulseEvent() は SetEvent() のように、先行してシグナル状態を作るようなことができず、  //　PulseEvent() 実行のタイミングで待機しているものを開放するだけしかできない。  //　なので、PulseEvent() を使用せず、子の数分のイベントを使用する。  //スレッド切り替えのためのスリープ  Sleep(0);  // //スレッド切り替え  // SwitchToThread();//OSに任せて再スケジューリング  // Yield();//廃止  }  //終了  printf("- end:(P)%s -\n", name);  fflush(stdout);  return 0;  }  //後続スレッド  unsigned int WINAPI followThreadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);  //後続スレッド数カウントアップ  LONG thread\_no = InterlockedIncrement(&s\_followThreadNo) - 1;  //開始  printf("- begin:(F)[%d]%s -\n", thread\_no, name);  fflush(stdout);  //後続スレッド処理完了イベントセット  SetEvent(s\_hFollowEvent[thread\_no]);  //処理  while (1)  {  //先行スレッド処理完了イベント待ち  WaitForSingleObject(s\_hPriorEvent[thread\_no], INFINITE);  //待機が完了しない時に他の処理を行いたい場合はタイムアウト値を指定する    //終了確認  if (s\_IsQuirProiorThread)  {  //処理終了  printf("(F)[%d]%s: [QUIT]\n", thread\_no, name);  fflush(stdout);    break;  }  //データ表示（前）  printf("(F)[%d]%s: [BEFORE] commonData=%d, tlsData=%d\n", thread\_no, name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++tls\_data;  //若干ランダムでスリープ（0～500 msec）  Sleep(rand() % 500);  //データ書き戻し  s\_tlsData = tls\_data;  //データ表示（後）  printf("(F)[%d]%s: [AFTER] commonData=%d, tlsData=%d\n", thread\_no, name, s\_commonData, s\_tlsData);  fflush(stdout);    //後続スレッド処理完了イベントセット  SetEvent(s\_hFollowEvent[thread\_no]);  //スレッド切り替えのためのスリープ  Sleep(0);  // //スレッド切り替え  // SwitchToThread();//OSに任せて再スケジューリング  // Yield();//廃止  }  //終了  printf("- end:(F)[%d]%s -\n", thread\_no, name);  fflush(stdout);  return 0;  }  //テスト  int main(const int argc, const char\* argv[])  {  //イベント生成  {  for (int i = 0; i < FOLLOW\_THREAD\_MAX; ++i)  {  //先行スレッド→後続スレッド通知用イベント（PulseEventが信頼できないので、後続スレッド数分作成）  s\_hPriorEvent[i] = CreateEvent(nullptr, false, false, nullptr);  //手動リセット=false（自動リセット）, 初期状態=false（リセット状態）  //後続スレッド→先行スレッド通知用イベント  s\_hFollowEvent[i] = CreateEvent(nullptr, false, false, nullptr);  //手動リセット=false（自動リセット）, 初期状態=false（リセット状態）  }  // //属性を指定して生成する場合  // SECURITY\_ATTRIBUTES attr = { sizeof(SECURITY\_ATTRIBUTES), nullptr, true };//子プロセスにハンドルを継承する  // SECURITY\_ATTRIBUTES attr = { sizeof(SECURITY\_ATTRIBUTES), nullptr, false };//子プロセスにハンドルを  // //継承しない　※デフォルト  // for (int i = 0; i < FOLLOW\_THREAD\_MAX; ++i)  // {  // //先行スレッド→後続スレッド通知用イベント（PulseEventが信頼できないので、後続スレッド数分作成）  // s\_hPriorEvent[i] = CreateEvent(&attr, true, false, nullptr);  // //後続スレッド→先行スレッド通知用イベント  // s\_hFollowEvent[i] = CreateEvent(&attr, false, false, nullptr);  // }  }  //スレッド作成  static const int FOLLOW\_THREAD\_NUM = 5;  static const int THREAD\_NUM = 1 + FOLLOW\_THREAD\_NUM;  static\_assert(FOLLOW\_THREAD\_NUM <= FOLLOW\_THREAD\_MAX, "FOLLOW\_THREAD\_NUM is over.");  s\_followThreadNum = FOLLOW\_THREAD\_NUM;  unsigned int tid[THREAD\_NUM] = {};  HANDLE hThread[THREAD\_NUM] =  {  (HANDLE)\_beginthreadex(nullptr, 1024, priorThreadFunc, "先行", 0, &tid[0]),  (HANDLE)\_beginthreadex(nullptr, 1024, followThreadFunc, "後続01", 0, &tid[1]),  (HANDLE)\_beginthreadex(nullptr, 1024, followThreadFunc, "後続02", 0, &tid[1]),  (HANDLE)\_beginthreadex(nullptr, 1024, followThreadFunc, "後続03", 0, &tid[1]),  (HANDLE)\_beginthreadex(nullptr, 1024, followThreadFunc, "後続04", 0, &tid[1]),  (HANDLE)\_beginthreadex(nullptr, 1024, followThreadFunc, "後続05", 0, &tid[1]),  };    //スレッド終了待ち  WaitForMultipleObjects(THREAD\_NUM, hThread, true, INFINITE);  //スレッドハンドルクローズ  for (int i = 0; i < THREAD\_NUM; ++i)  {  CloseHandle(hThread[i]);  }  //イベントの取得と解放を大量に実行して時間を計測  {  LARGE\_INTEGER freq;  QueryPerformanceFrequency(&freq);  LARGE\_INTEGER begin;  QueryPerformanceCounter(&begin);  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  SetEvent(s\_hPriorEvent[0]);  // WaitForSingleObject(s\_hPriorEvent[0], INFINITE);  // ResetEvent(s\_hPriorEvent[0]);  }  LARGE\_INTEGER end;  QueryPerformanceCounter(&end);  float duration = static\_cast<float>(static\_cast<double>(end.QuadPart - begin.QuadPart)  / static\_cast<double>(freq.QuadPart));  printf("Event \* %d = %.6f sec\n", TEST\_TIMES, duration);  }  //イベント破棄  for (int i = 0; i < FOLLOW\_THREAD\_MAX; ++i)  {  CloseHandle(s\_hPriorEvent[i]);  s\_hPriorEvent[i] = INVALID\_HANDLE\_VALUE;  CloseHandle(s\_hFollowEvent[i]);  s\_hFollowEvent[i] = INVALID\_HANDLE\_VALUE;  }  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:(P)先行 -  - begin:(F)[2]後続03 -  - begin:(F)[1]後続02 -  - begin:(F)[0]後続01 -  - begin:(F)[3]後続04 -  - begin:(F)[4]後続05 -  (P)先行: [BEFORE] commonData=0, tlsData=0 ←先行スレッドが処理を開始  (P)先行: [AFTER] commonData=1, tlsData=1 　（後続スレッドは、スレッド開始時から先行スレッドの処理終了を待機）  (F)[0]後続01: [BEFORE] commonData=1, tlsData=0 ←先行スレッドの処理が終了すると、後続スレッドが一斉スタート  (F)[1]後続02: [BEFORE] commonData=1, tlsData=0  (F)[2]後続03: [BEFORE] commonData=1, tlsData=0  (F)[3]後続04: [BEFORE] commonData=1, tlsData=0  (F)[4]後続05: [BEFORE] commonData=1, tlsData=0  (F)[1]後続02: [AFTER] commonData=1, tlsData=1  (F)[2]後続03: [AFTER] commonData=1, tlsData=1  (F)[4]後続05: [AFTER] commonData=1, tlsData=1  (F)[3]後続04: [AFTER] commonData=1, tlsData=1  (F)[0]後続01: [AFTER] commonData=1, tlsData=1  (P)先行: [BEFORE] commonData=1, tlsData=1 ←後続スレッドの処理が全て終了すると、また先行スレッドがスタート  (P)先行: [AFTER] commonData=2, tlsData=2  (F)[0]後続01: [BEFORE] commonData=2, tlsData=1 ←先行スレッドの処理が終了すると、また後続スレッドが一斉スタート  (F)[2]後続03: [BEFORE] commonData=2, tlsData=1  (F)[3]後続04: [BEFORE] commonData=2, tlsData=1  (F)[1]後続02: [BEFORE] commonData=2, tlsData=1  (F)[4]後続05: [BEFORE] commonData=2, tlsData=1  (F)[4]後続05: [AFTER] commonData=2, tlsData=2  (F)[1]後続02: [AFTER] commonData=2, tlsData=2  (F)[3]後続04: [AFTER] commonData=2, tlsData=2  (F)[0]後続01: [AFTER] commonData=2, tlsData=2  (F)[2]後続03: [AFTER] commonData=2, tlsData=2  (P)先行: [BEFORE] commonData=2, tlsData=2 ←後続スレッドの処理が全て終了すると、また先行スレッドがスタート  (P)先行: [AFTER] commonData=3, tlsData=3  (F)[1]後続02: [BEFORE] commonData=3, tlsData=2 ←先行スレッドの処理が終了すると、また後続スレッドが一斉スタート  (F)[0]後続01: [BEFORE] commonData=3, tlsData=2  (F)[4]後続05: [BEFORE] commonData=3, tlsData=2  (F)[3]後続04: [BEFORE] commonData=3, tlsData=2  (F)[2]後続03: [BEFORE] commonData=3, tlsData=2  (F)[2]後続03: [AFTER] commonData=3, tlsData=3  (F)[4]後続05: [AFTER] commonData=3, tlsData=3  (F)[0]後続01: [AFTER] commonData=3, tlsData=3  (F)[1]後続02: [AFTER] commonData=3, tlsData=3  (F)[3]後続04: [AFTER] commonData=3, tlsData=3  (P)先行: [QUIT] ←先行スレッド終了  (F)[1]後続02: [QUIT] ←先行スレッドの終了に反応して後続スレッドも終了  (F)[2]後続03: [QUIT]  - end:(P)先行 -  - end:(F)[2]後続03 -  (F)[4]後続05: [QUIT]  (F)[0]後続01: [QUIT]  (F)[3]後続04: [QUIT]  - end:(F)[0]後続01 -  - end:(F)[3]後続04 -  - end:(F)[4]後続05 -  - end:(F)[1]後続02 -  Event \* 10000000 = 2.102039 sec ←1千万回ループによる処理時間計測（SetEventのみの処理時間） |

### Win32API版（名前付き）

Win32版の名前付きイベント。

基本的には名前無しイベントと変わらないが、プロセス間での排他制御や、スレッドごとにイベントのアクセス権を変えたい場合に使用する。これにより、待ち受け専用のスレッドが誤って通知（シグナル状態化）することを防ぐことができる。

Win32API版名前付きイベントのサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <Windows.h>  #include <Process.h>  //スレッド情報  static const int FOLLOW\_THREAD\_MAX = 10; //後続スレッド最大数  static volatile LONG s\_followThreadNum = 0; //後続スレッド数  static volatile LONG s\_followThreadNo = 0; //後続スレッド番号発番用  static volatile bool s\_IsQuirProiorThread = false;//先行スレッド終了フラグ  //イベント名  static const char\* PRIOR\_EVENT\_NAME[FOLLOW\_THREAD\_MAX] = //先行スレッド処理完了イベント  {  "Prior Event 01",  "Prior Event 02",  "Prior Event 03",  "Prior Event 04",  "Prior Event 05",  "Prior Event 06",  "Prior Event 07",  "Prior Event 08",  "Prior Event 09",  "Prior Event 10",  };  static const char\* FOLLOW\_EVENT\_NAME[FOLLOW\_THREAD\_MAX] =//後続スレッド処理完了イベント  {  "Follow Event 01",  "Follow Event 02",  "Follow Event 03",  "Follow Event 04",  "Follow Event 05",  "Follow Event 06",  "Follow Event 07",  "Follow Event 08",  "Follow Event 09",  "Follow Event 10",  };  //共有データ  static int s\_commonData = 0;  //スレッド固有データ  \_\_declspec(thread) int s\_tlsData = 0;  //【処理説明】  //先行スレッドが共有データを作成し、それが完了したら  //複数の後続スレッドがスタート。  //後続スレッドは共有データを読み込むだけのため、並列で動作。  //後続スレッドの処理が全て完了したら、また先行スレッドが稼働。  //以上を何度か繰り返し、先行スレッドが終了したら全スレッド終了。  //※メインループ⇒描画スレッドのようなリレー処理への応用を想定。  //先行スレッド  unsigned int WINAPI priorThreadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);  //開始  printf("- begin:(P)%s -\n", name);  fflush(stdout);  //名前付きイベントオープン  HANDLE hPriorEvent[FOLLOW\_THREAD\_MAX];  HANDLE hFollowEvent[FOLLOW\_THREAD\_MAX];  for (int i = 0; i < FOLLOW\_THREAD\_MAX; ++i)  {  hPriorEvent[i] = OpenEvent(EVENT\_MODIFY\_STATE, false, PRIOR\_EVENT\_NAME[i]);  //先行スレッド処理完了イベント：SetEvetn, ResetEvent許可  hFollowEvent[i] = OpenEvent(SYNCHRONIZE, false, FOLLOW\_EVENT\_NAME[i]);  //後続スレッド処理完了イベント：Wait許可  }    //初期化  s\_IsQuirProiorThread = false;//先行処理終了フラグ  //処理  static const int LOOP\_COUNT\_MAX = 3;  int loop\_counter = 0;  while (1)  {  //全後続スレッド処理完了待ち  WaitForMultipleObjects(s\_followThreadNum, hFollowEvent, true, INFINITE);  //待機が完了しない時に他の処理を行いたい場合はタイムアウト値を指定する  //ループカウンタ進行＆終了判定  if (loop\_counter++ == LOOP\_COUNT\_MAX)  {  //処理終了  printf("(P)%s: [QUIT]\n", name);  fflush(stdout);    //先行スレッド終了フラグ  s\_IsQuirProiorThread = true;    //先行スレッド処理完了イベントセット  for (int i = 0; i < s\_followThreadNum; ++i)  SetEvent(hPriorEvent[i]);  break;  }  //データ表示（前）  printf("(P)%s: [BEFORE] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～499 msec）  Sleep(rand() % 500);  //データ書き戻し  s\_commonData = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("(P)%s: [AFTER] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //先行スレッド処理完了イベントセット  for (int i = 0; i < s\_followThreadNum; ++i)  {  SetEvent(hPriorEvent[i]);  }  //※PulseEvent() は SetEvent() のように、先行してシグナル状態を作るようなことができず、  //　PulseEvent() 実行のタイミングで待機しているものを開放するだけしかできない。  //　なので、PulseEvent() を使用せず、子の数分のイベントを使用する。  //スレッド切り替えのためのスリープ  Sleep(0);  // //スレッド切り替え  // SwitchToThread();//OSに任せて再スケジューリング  // Yield();//廃止  }  //名前付きイベントクローズ  for (int i = 0; i < FOLLOW\_THREAD\_MAX; ++i)  {  CloseHandle(hPriorEvent[i]);  CloseHandle(hFollowEvent[i]);  }  //終了  printf("- end:(P)%s -\n", name);  fflush(stdout);  return 0;  }  //後続スレッド  unsigned int WINAPI followThreadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);  //後続スレッド数カウントアップ  LONG thread\_no = InterlockedIncrement(&s\_followThreadNo) - 1;  //開始  printf("- begin:(F)[%d]%s -\n", thread\_no, name);  fflush(stdout);  //名前付きイベントオープン  HANDLE hPriorEvent[FOLLOW\_THREAD\_MAX];  HANDLE hFollowEvent[FOLLOW\_THREAD\_MAX];  for (int i = 0; i < FOLLOW\_THREAD\_MAX; ++i)  {  hPriorEvent[i] = OpenEvent(SYNCHRONIZE, false, PRIOR\_EVENT\_NAME[i]);  //先行スレッド処理完了イベント：Wait許可  hFollowEvent[i] = OpenEvent(EVENT\_MODIFY\_STATE, false, FOLLOW\_EVENT\_NAME[i]);  //後続スレッド処理完了イベント：SetEvetn, ResetEvent許可  }  //後続スレッド処理完了イベントセット  SetEvent(hFollowEvent[thread\_no]);  //処理  while (1)  {  //先行スレッド処理完了イベント待ち  WaitForSingleObject(hPriorEvent[thread\_no], INFINITE);  //待機が完了しない時に他の処理を行いたい場合はタイムアウト値を指定する  //終了確認  if (s\_IsQuirProiorThread)  {  //処理終了  printf("(F)[%d]%s: [QUIT]\n", thread\_no, name);  fflush(stdout);    break;  }  //データ表示（前）  printf("(F)[%d]%s: [BEFORE] commonData=%d, tlsData=%d\n", thread\_no, name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++tls\_data;  //若干ランダムでスリープ（0～500 msec）  Sleep(rand() % 500);  //データ書き戻し  s\_tlsData = tls\_data;  //データ表示（後）  printf("(F)[%d]%s: [AFTER] commonData=%d, tlsData=%d\n", thread\_no, name, s\_commonData, s\_tlsData);  fflush(stdout);    //後続スレッド処理完了イベントセット  SetEvent(hFollowEvent[thread\_no]);  //スレッド切り替えのためのスリープ  Sleep(0);  // //スレッド切り替え  // SwitchToThread();//OSに任せて再スケジューリング  // Yield();//廃止  }  //名前付きイベントクローズ  for (int i = 0; i < FOLLOW\_THREAD\_MAX; ++i)  {  CloseHandle(hPriorEvent[i]);  CloseHandle(hFollowEvent[i]);  }  //終了  printf("- end:(F)[%d]%s -\n", thread\_no, name);  fflush(stdout);  return 0;  }  //テスト  int main(const int argc, const char\* argv[])  {  //イベント生成  HANDLE hPriorEvent[FOLLOW\_THREAD\_MAX];  HANDLE hFollowEvent[FOLLOW\_THREAD\_MAX];  {  //名前付きイベント  for (int i = 0; i < FOLLOW\_THREAD\_MAX; ++i)  {  //先行スレッド→後続スレッド通知用イベント（PulseEventが信頼できないので、後続スレッド数分作成）  hPriorEvent[i] = CreateEvent(nullptr, false, false, PRIOR\_EVENT\_NAME[i]);  //手動リセット=false（自動リセット）, 初期状態=false（リセット状態）  //後続スレッド→先行スレッド通知用イベント  hFollowEvent[i] = CreateEvent(nullptr, false, false, FOLLOW\_EVENT\_NAME[i]);  //手動リセット=false（自動リセット）, 初期状態=false（リセット状態）  }    // //属性を指定して生成する場合  // SECURITY\_ATTRIBUTES attr = { sizeof(SECURITY\_ATTRIBUTES), nullptr, true };//子プロセスにハンドルを継承する  // SECURITY\_ATTRIBUTES attr = { sizeof(SECURITY\_ATTRIBUTES), nullptr, false };//子プロセスにハンドルを  // //継承しない　※デフォルト  // for (int i = 0; i < FOLLOW\_THREAD\_MAX; ++i)  // {  // //先行スレッド→後続スレッド通知用イベント（PulseEventが信頼できないので、後続スレッド数分作成）  // hPriorEvent[i] = CreateEvent(&attr, false, false, PRIOR\_EVENT\_NAME[i]);  // //手動リセット=false（自動リセット）, 初期状態=false（リセット状態）  // //後続スレッド→先行スレッド通知用イベント  // hFollowEvent[i] = CreateEvent(&attr, false, false, FOLLOW\_EVENT\_NAME[i]);  // //手動リセット=false（自動リセット）, 初期状態=false（リセット状態）  // }  }  //スレッド作成  static const int FOLLOW\_THREAD\_NUM = 5;  static const int THREAD\_NUM = 1 + FOLLOW\_THREAD\_NUM;  static\_assert(FOLLOW\_THREAD\_NUM <= FOLLOW\_THREAD\_MAX, "FOLLOW\_THREAD\_NUM is over.");  s\_followThreadNum = FOLLOW\_THREAD\_NUM;  unsigned int tid[THREAD\_NUM] = {};  HANDLE hThread[THREAD\_NUM] =  {  (HANDLE)\_beginthreadex(nullptr, 0, priorThreadFunc, "先行", 0, &tid[0]),  (HANDLE)\_beginthreadex(nullptr, 0, followThreadFunc, "後続01", 0, &tid[1]),  (HANDLE)\_beginthreadex(nullptr, 0, followThreadFunc, "後続02", 0, &tid[1]),  (HANDLE)\_beginthreadex(nullptr, 0, followThreadFunc, "後続03", 0, &tid[1]),  (HANDLE)\_beginthreadex(nullptr, 0, followThreadFunc, "後続04", 0, &tid[1]),  (HANDLE)\_beginthreadex(nullptr, 0, followThreadFunc, "後続05", 0, &tid[1]),  };    //スレッド終了待ち  WaitForMultipleObjects(THREAD\_NUM, hThread, true, INFINITE);  //スレッドハンドルクローズ  for (int i = 0; i < THREAD\_NUM; ++i)  {  CloseHandle(hThread[i]);  }  //イベントの取得と解放を大量に実行して時間を計測  {  LARGE\_INTEGER freq;  QueryPerformanceFrequency(&freq);  LARGE\_INTEGER begin;  QueryPerformanceCounter(&begin);  static const int TEST\_TIMES = 10000000;  for (int i = 0; i < TEST\_TIMES; ++i)  {  SetEvent(hPriorEvent[0]);  // WaitForSingleObject(hPriorEvent[0], INFINITE);  // ResetEvent(hPriorEvent[0]);  }  LARGE\_INTEGER end;  QueryPerformanceCounter(&end);  float duration = static\_cast<float>(static\_cast<double>(end.QuadPart - begin.QuadPart)  / static\_cast<double>(freq.QuadPart));  printf("Event \* %d = %.6f sec\n", TEST\_TIMES, duration);  }  //名前付きイベントクローズ  for (int i = 0; i < FOLLOW\_THREAD\_MAX; ++i)  {  CloseHandle(hPriorEvent[i]);  CloseHandle(hFollowEvent[i]);  }  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:(P)先行 -  - begin:(F)[0]後続01 -  - begin:(F)[1]後続02 -  - begin:(F)[2]後続03 -  - begin:(F)[3]後続04 -  - begin:(F)[4]後続05 -  (P)先行: [BEFORE] commonData=0, tlsData=0 ←先行スレッドが処理を開始  (P)先行: [AFTER] commonData=1, tlsData=1 　（後続スレッドは、スレッド開始時から先行スレッドの処理終了を待機）  (F)[0]後続01: [BEFORE] commonData=1, tlsData=0 ←先行スレッドの処理が終了すると、後続スレッドが一斉スタート  (F)[1]後続02: [BEFORE] commonData=1, tlsData=0  (F)[3]後続04: [BEFORE] commonData=1, tlsData=0  (F)[4]後続05: [BEFORE] commonData=1, tlsData=0  (F)[2]後続03: [BEFORE] commonData=1, tlsData=0  (F)[4]後続05: [AFTER] commonData=1, tlsData=1  (F)[0]後続01: [AFTER] commonData=1, tlsData=1  (F)[3]後続04: [AFTER] commonData=1, tlsData=1  (F)[1]後続02: [AFTER] commonData=1, tlsData=1  (F)[2]後続03: [AFTER] commonData=1, tlsData=1  (P)先行: [BEFORE] commonData=1, tlsData=1 ←後続スレッドの処理が全て終了すると、また先行スレッドがスタート  (P)先行: [AFTER] commonData=2, tlsData=2  (F)[0]後続01: [BEFORE] commonData=2, tlsData=1 ←先行スレッドの処理が終了すると、また後続スレッドが一斉スタート  (F)[2]後続03: [BEFORE] commonData=2, tlsData=1  (F)[3]後続04: [BEFORE] commonData=2, tlsData=1  (F)[1]後続02: [BEFORE] commonData=2, tlsData=1  (F)[4]後続05: [BEFORE] commonData=2, tlsData=1  (F)[4]後続05: [AFTER] commonData=2, tlsData=2  (F)[3]後続04: [AFTER] commonData=2, tlsData=2  (F)[2]後続03: [AFTER] commonData=2, tlsData=2  (F)[1]後続02: [AFTER] commonData=2, tlsData=2  (F)[0]後続01: [AFTER] commonData=2, tlsData=2  (P)先行: [BEFORE] commonData=2, tlsData=2 ←後続スレッドの処理が全て終了すると、また先行スレッドがスタート  (P)先行: [AFTER] commonData=3, tlsData=3  (F)[0]後続01: [BEFORE] commonData=3, tlsData=2 ←先行スレッドの処理が終了すると、また後続スレッドが一斉スタート  (F)[2]後続03: [BEFORE] commonData=3, tlsData=2  (F)[4]後続05: [BEFORE] commonData=3, tlsData=2  (F)[3]後続04: [BEFORE] commonData=3, tlsData=2  (F)[1]後続02: [BEFORE] commonData=3, tlsData=2  (F)[1]後続02: [AFTER] commonData=3, tlsData=3  (F)[3]後続04: [AFTER] commonData=3, tlsData=3  (F)[4]後続05: [AFTER] commonData=3, tlsData=3  (F)[0]後続01: [AFTER] commonData=3, tlsData=3  (F)[2]後続03: [AFTER] commonData=3, tlsData=3  (P)先行: [QUIT] ←先行スレッド終了  (F)[0]後続01: [QUIT] ←先行スレッドの終了に反応して後続スレッドも終了  (F)[2]後続03: [QUIT]  - end:(P)先行 -  (F)[4]後続05: [QUIT]  (F)[3]後続04: [QUIT]  (F)[1]後続02: [QUIT]  - end:(F)[0]後続01 -  - end:(F)[2]後続03 -  - end:(F)[4]後続05 -  - end:(F)[3]後続04 -  - end:(F)[1]後続02 -  Event \* 10000000 = 2.079530 sec ←1千万回ループによる処理時間計測（SetEventのみの処理時間） |

## モニター：アトミック操作（インターロック操作）によるビジーなモニター

ごく短い処理をリレーするような場合、ビジーウェイトでモニターするのも有効である。

この場合、アトミック操作（インターロック操作）を活用する。

### 【用語解説】コンペア・アンド・スワップ（CAS）操作

ここで用いるのは、「コンペア・アンド・スワップ」（Compare-And-Swap = CAS）と呼ばれる手法で、ロックフリーなシステムの常套手段である。

ある変数の値をチェックし、それが指定の値であれば、新しい値に置換する。

CPUの命令レベルで対応しており、割り込まれることなく一瞬でその操作が完了する。

この仕組みをC++言語から利用する手段が提供されている。

なお、前述の「変数操作による排他制御：インラインアセンブラ」では、インラインアセンブラでXCHG命令を使用している。これは比較を行わない単純な交換命令。

### Win32API版：インターロック操作

Win32APIでは、InterlockedCompareExchange() を使用する。

Win32API版インターロック操作によるモニターのサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <Windows.h>  #include <Process.h>  //スレッド情報  static const int FOLLOW\_THREAD\_MAX = 10; //後続スレッド最大数  static volatile LONG s\_followThreadNum = 0; //後続スレッド数  static volatile LONG s\_followThreadNo = 0; //後続スレッド番号発番用  static volatile bool s\_IsQuirProiorThread = false;//先行スレッド終了フラグ  //モニター用変数  enum E\_PROCESS  {  PRIOR\_RUNNING, //先行処理実行中  PRIOR\_IDLE, //先行処理アイドル状態  FOLLOW\_RUNNING,//後続処理実行中  FOLLOW\_IDLE, //後続処理アイドル状態  };  static LONG s\_followFinished[FOLLOW\_THREAD\_MAX];//後続スレッド処理完了フラグ  //共有データ  static int s\_commonData = 0;  //スレッド固有データ  \_\_declspec(thread) int s\_tlsData = 0;  //【処理説明】  //先行スレッドが共有データを作成し、それが完了したら  //複数の後続スレッドがスタート。  //後続スレッドは共有データを読み込むだけのため、並列で動作。  //後続スレッドの処理が全て完了したら、また先行スレッドが稼働。  //以上を何度か繰り返し、先行スレッドが終了したら全スレッド終了。  //※メインループ⇒描画スレッドのようなリレー処理への応用を想定。  //先行スレッド  unsigned int WINAPI priorThreadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);  //開始  printf("- begin:(P)%s -\n", name);  fflush(stdout);  //初期化  s\_IsQuirProiorThread = false;//先行処理終了フラグ  //処理  static const int LOOP\_COUNT\_MAX = 3;  int loop\_counter = 0;  while (1)  {  //全後続スレッド処理完了待ち  for (int i = 0; i < s\_followThreadNum; ++i)  {  E\_PROCESS prev = E\_PROCESS::FOLLOW\_IDLE;  while (InterlockedCompareExchange(&s\_followFinished[i], E\_PROCESS::PRIOR\_RUNNING,  E\_PROCESS::FOLLOW\_IDLE) != E\_PROCESS::FOLLOW\_IDLE){}  }  //ループカウンタ進行＆終了判定  if (loop\_counter++ == LOOP\_COUNT\_MAX)  {  //処理終了  printf("(P)%s: [QUIT]\n", name);  fflush(stdout);    //先行スレッド終了フラグ  s\_IsQuirProiorThread = true;  //先行スレッド処理完了：全待機スレッドを起床  for (int i = 0; i < s\_followThreadNum; ++i)  {  s\_followFinished[i] = E\_PROCESS::PRIOR\_IDLE;  }  break;  }  //データ表示（前）  printf("(P)%s: [BEFORE] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～499 msec）  Sleep(rand() % 500);  //データ書き戻し  s\_commonData = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("(P)%s: [AFTER] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //先行スレッド処理完了：全待機スレッドを起床  for (int i = 0; i < s\_followThreadNum; ++i)  {  s\_followFinished[i] = E\_PROCESS::PRIOR\_IDLE;  }  //スレッド切り替えのためのスリープ  Sleep(0);  // //スレッド切り替え  // SwitchToThread();//OSに任せて再スケジューリング  // Yield();//廃止  }  //終了  printf("- end:(P)%s -\n", name);  fflush(stdout);  return 0;  }  //後続スレッド  unsigned int WINAPI followThreadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);  //後続スレッド数カウントアップ  LONG thread\_no = InterlockedIncrement(&s\_followThreadNo) - 1;  //開始  printf("- begin:(F)[%d]%s -\n", thread\_no, name);  fflush(stdout);  //後続スレッド処理完了：待機スレッドを起床  s\_followFinished[thread\_no] = E\_PROCESS::FOLLOW\_IDLE;  //処理  while (1)  {  //先行スレッド処理完了待ち  while (InterlockedCompareExchange(&s\_followFinished[thread\_no], E\_PROCESS::FOLLOW\_RUNNING,  E\_PROCESS::PRIOR\_IDLE) != E\_PROCESS::PRIOR\_IDLE){}  //終了確認  if (s\_IsQuirProiorThread)  {  //処理終了  printf("(F)[%d]%s: [QUIT]\n", thread\_no, name);  fflush(stdout);  break;  }  //データ表示（前）  printf("(F)[%d]%s: [BEFORE] commonData=%d, tlsData=%d\n", thread\_no, name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++tls\_data;  //若干ランダムでスリープ（0～500 msec）  Sleep(rand() % 500);  //データ書き戻し  s\_tlsData = tls\_data;  //データ表示（後）  printf("(F)[%d]%s: [AFTER] commonData=%d, tlsData=%d\n", thread\_no, name, s\_commonData, s\_tlsData);  fflush(stdout);  //後続スレッド処理完了：待機スレッドを起床  s\_followFinished[thread\_no] = E\_PROCESS::FOLLOW\_IDLE;  //スレッド切り替えのためのスリープ  Sleep(0);  // //スレッド切り替え  // SwitchToThread();//OSに任せて再スケジューリング  // Yield();//廃止  }  //終了  printf("- end:(F)[%d]%s -\n", thread\_no, name);  fflush(stdout);  return 0;  }  //テスト  int main(const int argc, const char\* argv[])  {  //初期化  for (int i = 0; i < s\_followThreadNum; ++i)  s\_followFinished[i] = E\_PROCESS::FOLLOW\_IDLE;  //スレッド作成  static const int FOLLOW\_THREAD\_NUM = 5;  static const int THREAD\_NUM = 1 + FOLLOW\_THREAD\_NUM;  static\_assert(FOLLOW\_THREAD\_NUM <= FOLLOW\_THREAD\_MAX, "FOLLOW\_THREAD\_NUM is over.");  s\_followThreadNum = FOLLOW\_THREAD\_NUM;  unsigned int tid[THREAD\_NUM] = {};  HANDLE hThread[THREAD\_NUM] =  {  (HANDLE)\_beginthreadex(nullptr, 0, priorThreadFunc, "先行", 0, &tid[0]),  (HANDLE)\_beginthreadex(nullptr, 0, followThreadFunc, "後続01", 0, &tid[1]),  (HANDLE)\_beginthreadex(nullptr, 0, followThreadFunc, "後続02", 0, &tid[1]),  (HANDLE)\_beginthreadex(nullptr, 0, followThreadFunc, "後続03", 0, &tid[1]),  (HANDLE)\_beginthreadex(nullptr, 0, followThreadFunc, "後続04", 0, &tid[1]),  (HANDLE)\_beginthreadex(nullptr, 0, followThreadFunc, "後続05", 0, &tid[1]),  };    //スレッド終了待ち  WaitForMultipleObjects(THREAD\_NUM, hThread, true, INFINITE);  //スレッドハンドルクローズ  for (int i = 0; i < THREAD\_NUM; ++i)  {  CloseHandle(hThread[i]);  }  //インターロック操作を大量に実行して時間を計測  {  LARGE\_INTEGER freq;  QueryPerformanceFrequency(&freq);  LARGE\_INTEGER begin;  QueryPerformanceCounter(&begin);  static const int TEST\_TIMES = 10000000;  s\_followFinished[0] = E\_PROCESS::FOLLOW\_IDLE;  for (int i = 0; i < TEST\_TIMES; ++i)  {  while (InterlockedCompareExchange(&s\_followFinished[0], E\_PROCESS::PRIOR\_RUNNING,  E\_PROCESS::FOLLOW\_IDLE) != E\_PROCESS::FOLLOW\_IDLE){}  while (InterlockedCompareExchange(&s\_followFinished[0], E\_PROCESS::PRIOR\_IDLE,  E\_PROCESS::PRIOR\_RUNNING) != E\_PROCESS::PRIOR\_RUNNING){}  while (InterlockedCompareExchange(&s\_followFinished[0], E\_PROCESS::FOLLOW\_RUNNING,  E\_PROCESS::PRIOR\_IDLE) != E\_PROCESS::PRIOR\_IDLE){}  while (InterlockedCompareExchange(&s\_followFinished[0], E\_PROCESS::FOLLOW\_IDLE,  E\_PROCESS::FOLLOW\_RUNNING) != E\_PROCESS::FOLLOW\_RUNNING){}  }  LARGE\_INTEGER end;  QueryPerformanceCounter(&end);  float duration = static\_cast<float>(static\_cast<double>(end.QuadPart - begin.QuadPart)  / static\_cast<double>(freq.QuadPart));  printf("Event \* %d = %.6f sec\n", TEST\_TIMES, duration);  }  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:(P)先行 -  - begin:(F)[1]後続02 -  - begin:(F)[0]後続01 -  - begin:(F)[2]後続03 -  - begin:(F)[3]後続04 -  - begin:(F)[4]後続05 -  (P)先行: [BEFORE] commonData=0, tlsData=0 ←先行スレッドが処理を開始  (P)先行: [AFTER] commonData=1, tlsData=1 　（後続スレッドは、スレッド開始時から先行スレッドの処理終了を待機）  (F)[3]後続04: [BEFORE] commonData=1, tlsData=0 ←先行スレッドの処理が終了すると、後続スレッドが一斉スタート  (F)[1]後続02: [BEFORE] commonData=1, tlsData=0  (F)[2]後続03: [BEFORE] commonData=1, tlsData=0  (F)[4]後続05: [BEFORE] commonData=1, tlsData=0  (F)[0]後続01: [BEFORE] commonData=1, tlsData=0  (F)[1]後続02: [AFTER] commonData=1, tlsData=1  (F)[0]後続01: [AFTER] commonData=1, tlsData=1  (F)[4]後続05: [AFTER] commonData=1, tlsData=1  (F)[2]後続03: [AFTER] commonData=1, tlsData=1  (F)[3]後続04: [AFTER] commonData=1, tlsData=1  (P)先行: [BEFORE] commonData=1, tlsData=1 ←後続スレッドの処理が全て終了すると、また先行スレッドがスタート  (P)先行: [AFTER] commonData=2, tlsData=2  (F)[2]後続03: [BEFORE] commonData=2, tlsData=1 ←先行スレッドの処理が終了すると、また後続スレッドが一斉スタート  (F)[1]後続02: [BEFORE] commonData=2, tlsData=1  (F)[4]後続05: [BEFORE] commonData=2, tlsData=1  (F)[3]後続04: [BEFORE] commonData=2, tlsData=1  (F)[0]後続01: [BEFORE] commonData=2, tlsData=1  (F)[4]後続05: [AFTER] commonData=2, tlsData=2  (F)[3]後続04: [AFTER] commonData=2, tlsData=2  (F)[2]後続03: [AFTER] commonData=2, tlsData=2  (F)[1]後続02: [AFTER] commonData=2, tlsData=2  (F)[0]後続01: [AFTER] commonData=2, tlsData=2  (P)先行: [BEFORE] commonData=2, tlsData=2 ←後続スレッドの処理が全て終了すると、また先行スレッドがスタート  (P)先行: [AFTER] commonData=3, tlsData=3  (F)[3]後続04: [BEFORE] commonData=3, tlsData=2 ←先行スレッドの処理が終了すると、また後続スレッドが一斉スタート  (F)[1]後続02: [BEFORE] commonData=3, tlsData=2  (F)[0]後続01: [BEFORE] commonData=3, tlsData=2  (F)[4]後続05: [BEFORE] commonData=3, tlsData=2  (F)[2]後続03: [BEFORE] commonData=3, tlsData=2  (F)[4]後続05: [AFTER] commonData=3, tlsData=3  (F)[0]後続01: [AFTER] commonData=3, tlsData=3  (F)[1]後続02: [AFTER] commonData=3, tlsData=3  (F)[2]後続03: [AFTER] commonData=3, tlsData=3  (F)[3]後続04: [AFTER] commonData=3, tlsData=3  (P)先行: [QUIT] ←先行スレッド終了  - end:(P)先行 -  (F)[2]後続03: [QUIT] ←先行スレッドの終了に反応して後続スレッドも終了  (F)[3]後続04: [QUIT]  - end:(F)[2]後続03 -  - end:(F)[3]後続04 -  (F)[1]後続02: [QUIT]  (F)[0]後続01: [QUIT]  (F)[4]後続05: [QUIT]  - end:(F)[0]後続01 -  - end:(F)[4]後続05 -  - end:(F)[1]後続02 -  Event \* 10000000 = 0.247101 sec ←1千万回ループによる処理時間計測（4回のステート更新） |

### C++11版：アトミック操作

C++11のアトミック操作によるモニター。

かなり細かい操作が可能。メモリオーダー指定を使用したパフォーマンスチューニングの方法も後述する。

C++11版のアトミック操作によるモニターのサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <thread>  #include <atomic>  #include <chrono> //時間計測用  #include <random> //乱数生成用  //スレッド情報  static const int FOLLOW\_THREAD\_MAX = 10; //後続スレッド最大数  static volatile int s\_followThreadNum = 0; //後続スレッド数  static std::atomic<int> s\_followThreadNo = 0; //後続スレッド番号発番用  static volatile bool s\_IsQuirProiorThread = false;//先行スレッド終了フラグ  //モニター用変数  enum E\_PROCESS  {  PRIOR\_RUNNING, //先行処理実行中  PRIOR\_IDLE, //先行処理アイドル状態  FOLLOW\_RUNNING,//後続処理実行中  FOLLOW\_IDLE, //後続処理アイドル状態  };  static std::atomic<E\_PROCESS> s\_followFinished[FOLLOW\_THREAD\_MAX];//後続スレッド処理完了フラグ  //共有データ  static int s\_commonData = 0;  //スレッド固有データ  \_\_declspec(thread) int s\_tlsData = 0;  //thread\_local int s\_tlsData = 0;//Visual C++ 2013 では thread\_local キーワードが使えない  //【処理説明】  //先行スレッドが共有データを作成し、それが完了したら  //複数の後続スレッドがスタート。  //後続スレッドは共有データを読み込むだけのため、並列で動作。  //後続スレッドの処理が全て完了したら、また先行スレッドが稼働。  //以上を何度か繰り返し、先行スレッドが終了したら全スレッド終了。  //※メインループ⇒描画スレッドのようなリレー処理への応用を想定。  //先行スレッド  void priorThreadFunc(const char\* name)  {  //開始  printf("- begin:(P)%s -\n", name);  fflush(stdout);  //乱数  std::random\_device seed\_gen;  std::mt19937 rnd(seed\_gen());  std::uniform\_int\_distribution<int> sleep\_time(0, 499);  //初期化  s\_IsQuirProiorThread = false;//先行処理終了フラグ  //処理  static const int LOOP\_COUNT\_MAX = 3;  int loop\_counter = 0;  while (1)  {  //全後続スレッド処理完了待ち  for (int i = 0; i < s\_followThreadNum; ++i)  {  E\_PROCESS prev = E\_PROCESS::FOLLOW\_IDLE;  while (!s\_followFinished[i].compare\_exchange\_weak(prev, E\_PROCESS::PRIOR\_RUNNING))  { prev = E\_PROCESS::FOLLOW\_IDLE; }  }  //ループカウンタ進行＆終了判定  if (loop\_counter++ == LOOP\_COUNT\_MAX)  {  //処理終了  printf("(P)%s: [QUIT]\n", name);  fflush(stdout);    //先行スレッド終了フラグ  s\_IsQuirProiorThread = true;    //先行スレッド処理完了：全待機スレッドを起床  for (int i = 0; i < s\_followThreadNum; ++i)  {  s\_followFinished[i].store(E\_PROCESS::PRIOR\_IDLE);  }  break;  }  //データ表示（前）  printf("(P)%s: [BEFORE] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++common\_data;  ++tls\_data;  //若干ランダムでスリープ（0～499 msec）  std::this\_thread::sleep\_for(std::chrono::milliseconds(sleep\_time(rnd)));  //データ書き戻し  s\_commonData = common\_data;  s\_tlsData = tls\_data;  //データ表示（後）  printf("(P)%s: [AFTER] commonData=%d, tlsData=%d\n", name, s\_commonData, s\_tlsData);  fflush(stdout);  //先行スレッド処理完了：全待機スレッドを起床  for (int i = 0; i < s\_followThreadNum; ++i)  {  s\_followFinished[i].stre(E\_PROCESS::PRIOR\_IDLE);  }  //スレッド切り替えのためのスリープ  std::this\_thread::sleep\_for(std::chrono::milliseconds(0));  // //スレッド切り替え  // std::this\_thread::yield();//OSに任せて再スケジューリング  }  //終了  printf("- end:(P)%s -\n", name);  fflush(stdout);  }  //後続スレッド  void followThreadFunc(const char\* name)  {  //後続スレッド数カウントアップ  const int thread\_no = s\_followThreadNo++;  //開始  printf("- begin:(F)[%d]%s -\n", thread\_no, name);  fflush(stdout);  //乱数  std::random\_device seed\_gen;  std::mt19937 rnd(seed\_gen());  std::uniform\_int\_distribution<int> sleep\_time(0, 499);  //後続スレッド処理完了：待機スレッドを起床  s\_followFinished[thread\_no].store(E\_PROCESS::FOLLOW\_IDLE);  //処理  while (1)  {  //先行スレッド処理完了待ち  E\_PROCESS prev = E\_PROCESS::PRIOR\_IDLE;  while (!s\_followFinished[thread\_no].compare\_exchange\_weak(prev, E\_PROCESS::FOLLOW\_RUNNING))  { prev = E\_PROCESS::PRIOR\_IDLE; }  //終了確認  if (s\_IsQuirProiorThread)  {  //処理終了  printf("(F)[%d]%s: [QUIT]\n", thread\_no, name);  fflush(stdout);  break;  }  //データ表示（前）  printf("(F)[%d]%s: [BEFORE] commonData=%d, tlsData=%d\n", thread\_no, name, s\_commonData, s\_tlsData);  fflush(stdout);  //データ取得  int common\_data = s\_commonData;  int tls\_data = s\_tlsData;  //データ更新  ++tls\_data;  //若干ランダムでスリープ（0～500 msec）  std::this\_thread::sleep\_for(std::chrono::milliseconds(sleep\_time(rnd)));  //データ書き戻し  s\_tlsData = tls\_data;  //データ表示（後）  printf("(F)[%d]%s: [AFTER] commonData=%d, tlsData=%d\n", thread\_no, name, s\_commonData, s\_tlsData);  fflush(stdout);    //後続スレッド処理完了：待機スレッドを起床  s\_followFinished[thread\_no].store(E\_PROCESS::FOLLOW\_IDLE);  //スレッド切り替えのためのスリープ  std::this\_thread::sleep\_for(std::chrono::milliseconds(0));  // //スレッド切り替え  // std::this\_thread::yield();//OSに任せて再スケジューリング  }  //終了  printf("- end:(F)[%d]%s -\n", thread\_no, name);  fflush(stdout);  }  //テスト  int main(const int argc, const char\* argv[])  {  //スレッド作成  static const int FOLLOW\_THREAD\_NUM = 5;  static const int THREAD\_NUM = 1 + FOLLOW\_THREAD\_NUM;  static\_assert(FOLLOW\_THREAD\_NUM <= FOLLOW\_THREAD\_MAX, "FOLLOW\_THREAD\_NUM is over.");  s\_followThreadNum = FOLLOW\_THREAD\_NUM;  for (int i = 0; i < THREAD\_NUM - 1; ++i)  s\_followFinished[i].store(E\_PROCESS::FOLLOW\_IDLE);  std::thread thread\_obj00 = std::thread(priorThreadFunc, "先行");  std::thread thread\_obj01 = std::thread(followThreadFunc, "後続01");  std::thread thread\_obj02 = std::thread(followThreadFunc, "後続02");  std::thread thread\_obj03 = std::thread(followThreadFunc, "後続03");  std::thread thread\_obj04 = std::thread(followThreadFunc, "後続04");  std::thread thread\_obj05 = std::thread(followThreadFunc, "後続05");  //スレッド終了待ち  thread\_obj00.join();  thread\_obj01.join();  thread\_obj02.join();  thread\_obj03.join();  thread\_obj04.join();  thread\_obj05.join();  //アトミック操作を大量に実行して時間を計測  {  auto begin = std::chrono::high\_resolution\_clock::now();  static const int TEST\_TIMES = 10000000;  s\_followFinished[0] = E\_PROCESS::FOLLOW\_IDLE;  E\_PROCESS prev = E\_PROCESS::FOLLOW\_IDLE;  for (int i = 0; i < TEST\_TIMES; ++i)  {  while (!s\_followFinished[0].compare\_exchange\_weak(prev, E\_PROCESS::PRIOR\_RUNNING)){}  while (!s\_followFinished[0].compare\_exchange\_weak(prev, E\_PROCESS::PRIOR\_IDLE)){}  while (!s\_followFinished[0].compare\_exchange\_weak(prev, E\_PROCESS::FOLLOW\_RUNNING)){}  while (!s\_followFinished[0].compare\_exchange\_weak(prev, E\_PROCESS::FOLLOW\_IDLE)){}  }  auto end = std::chrono::high\_resolution\_clock::now();  auto duration = static\_cast<float>(static\_cast<double>(  std::chrono::duration\_cast<std::chrono::microseconds>(end - begin).count()) / 1000000.);  printf("Watch-atomic \* %d = %.6f sec\n", TEST\_TIMES, duration);  }  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:(P)先行 -  (P)先行: [BEFORE] commonData=0, tlsData=0 ←先行スレッドが処理を開始  - begin:(F)[0]後続01 - 　（後続スレッドは、スレッド開始時から先行スレッドの処理終了を待機）  - begin:(F)[1]後続02 -  - begin:(F)[2]後続03 -  - begin:(F)[3]後続04 -  - begin:(F)[4]後続05 -  (P)先行: [AFTER] commonData=1, tlsData=1  (F)[0]後続01: [BEFORE] commonData=1, tlsData=0 ←先行スレッドの処理が終了すると、後続スレッドが一斉スタート  (F)[2]後続03: [BEFORE] commonData=1, tlsData=0  (F)[1]後続02: [BEFORE] commonData=1, tlsData=0  (F)[4]後続05: [BEFORE] commonData=1, tlsData=0  (F)[3]後続04: [BEFORE] commonData=1, tlsData=0  (F)[1]後続02: [AFTER] commonData=1, tlsData=1  (F)[0]後続01: [AFTER] commonData=1, tlsData=1  (F)[4]後続05: [AFTER] commonData=1, tlsData=1  (F)[3]後続04: [AFTER] commonData=1, tlsData=1  (F)[2]後続03: [AFTER] commonData=1, tlsData=1  (P)先行: [BEFORE] commonData=1, tlsData=1 ←後続スレッドの処理が全て終了すると、また先行スレッドがスタート  (P)先行: [AFTER] commonData=2, tlsData=2  (F)[0]後続01: [BEFORE] commonData=2, tlsData=1 ←先行スレッドの処理が終了すると、また後続スレッドが一斉スタート  (F)[1]後続02: [BEFORE] commonData=2, tlsData=1  (F)[4]後続05: [BEFORE] commonData=2, tlsData=1  (F)[3]後続04: [BEFORE] commonData=2, tlsData=1  (F)[2]後続03: [BEFORE] commonData=2, tlsData=1  (F)[1]後続02: [AFTER] commonData=2, tlsData=2  (F)[4]後続05: [AFTER] commonData=2, tlsData=2  (F)[2]後続03: [AFTER] commonData=2, tlsData=2  (F)[3]後続04: [AFTER] commonData=2, tlsData=2  (F)[0]後続01: [AFTER] commonData=2, tlsData=2  (P)先行: [BEFORE] commonData=2, tlsData=2 ←後続スレッドの処理が全て終了すると、また先行スレッドがスタート  (P)先行: [AFTER] commonData=3, tlsData=3  (F)[1]後続02: [BEFORE] commonData=3, tlsData=2 ←先行スレッドの処理が終了すると、また後続スレッドが一斉スタート  (F)[3]後続04: [BEFORE] commonData=3, tlsData=2  (F)[4]後続05: [BEFORE] commonData=3, tlsData=2  (F)[2]後続03: [BEFORE] commonData=3, tlsData=2  (F)[0]後続01: [BEFORE] commonData=3, tlsData=2  (F)[3]後続04: [AFTER] commonData=3, tlsData=3  (F)[4]後続05: [AFTER] commonData=3, tlsData=3  (F)[1]後続02: [AFTER] commonData=3, tlsData=3  (F)[2]後続03: [AFTER] commonData=3, tlsData=3  (F)[0]後続01: [AFTER] commonData=3, tlsData=3  (P)先行: [QUIT] ←先行スレッド終了  (F)[0]後続01: [QUIT] ←先行スレッドの終了に反応して後続スレッドも終了  (F)[2]後続03: [QUIT]  - end:(F)[0]後続01 -  (F)[1]後続02: [QUIT]  - end:(F)[2]後続03 -  (F)[3]後続04: [QUIT]  - end:(P)先行 -  (F)[4]後続05: [QUIT]  - end:(F)[3]後続04 -  - end:(F)[1]後続02 -  - end:(F)[4]後続05 -  Watch-atomic \* 10000000 = 0.499029 sec ←1千万回ループによる処理時間計測（4回のステート更新） |

### C++11版：アトミック操作のメモリオーダー指定

アトミック操作時にメモリオーダーを指定し、メモリバリアを明示的に使い分けた場合のサンプルを示す。

「アトミック操作を大量に実行して時間を計測」の一部だけ書き換える。

実はこれはパフォーマンスの差がほとんど得られなかったので、比較用に前述の「変数操作による排他制御：アトミック型」のサンプルに対してメモリオーダーを指定した場合の結果も示す。

C++11版のアトミック操作によるモニターのサンプル（一部抜粋）：メモリオーダー追加

|  |
| --- |
| …（略）…  while (!s\_followFinished[0].compare\_exchange\_weak(prev, E\_PROCESS::PRIOR\_RUNNING)){}  while (!s\_followFinished[0].compare\_exchange\_weak(prev, E\_PROCESS::PRIOR\_IDLE)){}  while (!s\_followFinished[0].compare\_exchange\_weak(prev, E\_PROCESS::FOLLOW\_RUNNING)){}  while (!s\_followFinished[0].compare\_exchange\_weak(prev, E\_PROCESS::FOLLOW\_IDLE)){}  …（略）…  ↓（メモリオーダー指定追加）  …（略）…  while (!s\_followFinished[0].compare\_exchange\_weak(prev, E\_PROCESS::PRIOR\_RUNNING, std::memory\_order\_relaxed,  std::memory\_order\_relaxed)){}  while (!s\_followFinished[0].compare\_exchange\_weak(prev, E\_PROCESS::PRIOR\_IDLE, std::memory\_order\_relaxed,  std::memory\_order\_relaxed)){}  while (!s\_followFinished[0].compare\_exchange\_weak(prev, E\_PROCESS::FOLLOW\_RUNNING, std::memory\_order\_relaxed,  std::memory\_order\_relaxed)){}  while (!s\_followFinished[0].compare\_exchange\_weak(prev, E\_PROCESS::FOLLOW\_IDLE, std::memory\_order\_relaxed,  std::memory\_order\_relaxed)){}  …（略）… |

↓（実行結果）

|  |
| --- |
| Watch-atomic \* 10000000 = 0.499029 sec  Watch-atomic \* 10000000 (memory barrier free) = 0.489027 sec ←メモリオーダー追加版（ほとんど効果なし） |

C++11版のアトミック操作によるロックのサンプル（一部抜粋）：

|  |
| --- |
| …（略）…  while (s\_lock.test\_and\_set()){}  s\_lock.clear();  …（略）…  ↓（メモリオーダー指定追加）  …（略）…  while (s\_lock.test\_and\_set(std::memory\_order\_seq\_cst)){}  s\_lock.clear(std::memory\_order\_release);  …（略）… |

↓（実行結果）

|  |
| --- |
| Atomic \* 10000000 = 0.126008 sec  Atomic \* 10000000 (memolry barrier free) = 0.080004 sec ←メモリオーダー追加版（効果あり） |

↓（メモリオーダーの説明） ※サンプルプログラムに書いたコメントからの抜粋

|  |
| --- |
| //http://www.justsoftwaresolutions.co.uk/threading/intel-memory-ordering-and-c++-memory-model.html  //【メモリオーダーの意味】（x86命令相当の場合）  //Memory Ordering, Store, Load  //------------------------- --------------- --------------  //std::memory\_order\_relaxed MOV[mem],reg MOV reg,[mem]  // ←メモリバリアなし：メモリ更新と読み込みの順序性を  // 　保証しない  //std::memory\_order\_consume (n/a) MOV reg,[mem] (Data-Dependency Ordering)  // ←読み込み専用のメモリバリア（少し弱い）  // ※ストア処理で使うとエラー（交換可）  //std::memory\_order\_acquire (n/a) MOV reg,[mem]  // ←読み込み専用のメモリバリア  // ※ストア処理で使うとエラー（交換可）  //std::memory\_order\_release MOV[mem],reg (n/a)  // ←書き込み専用のメモリバリア  // ※ロード処理で使うとエラー（交換可）  //std::memory\_order\_acq\_rel MOV[mem],reg MOV reg,[mem] (acquire & release)  // ←交換専用のメモリバリア  // ※ロード／ストア処理で使うとエラー  //std::memory\_order\_seq\_cst XCHG[mem],reg MOV reg,[mem] (Sequential consistency)  // ←読み書きメモリバリア（強力）  // ※デフォルト（必ずしも低速というわけでではない）  //・Load ... レジスタ ← メモリ  //・Store ... レジスタ → メモリ    //【処理サンプル】  //std::atomic<int> o(1);  //int v = 0;  //v = o.load(std::memory\_order\_relaxed);//OK  //v = o.load(std::memory\_order\_consume);//OK  //v = o.load(std::memory\_order\_acquire);//OK  ////v = o.load(std::memory\_order\_release);//例外：Invalid Memory Order  ////v = o.load(std::memory\_order\_acq\_rel);//例外：Invalid Memory Order  //v = o.load(std::memory\_order\_seq\_cst);//OK  //o.store(v, std::memory\_order\_relaxed);//OK  ////o.store(v, std::memory\_order\_consume);//例外：Invalid Memory Order  ////o.store(v, std::memory\_order\_acquire);//例外：Invalid Memory Order  //o.store(v, std::memory\_order\_release);//OK  ////o.store(v, std::memory\_order\_acq\_rel);//例外：Invalid Memory Order  //o.store(v, std::memory\_order\_seq\_cst);//OK  //o.exchange(2, std::memory\_order\_relaxed);//OK  //o.exchange(3, std::memory\_order\_consume);//OK  //o.exchange(4, std::memory\_order\_acquire);//OK  //o.exchange(5, std::memory\_order\_release);//OK  //o.exchange(6, std::memory\_order\_acq\_rel);//OK  //o.exchange(7, std::memory\_order\_seq\_cst);//OK  //v = 7;  //v = o.compare\_exchange\_weak(v, 7, std::memory\_order\_relaxed, std::memory\_order\_relaxed);//条件成立時：OK  //v = o.compare\_exchange\_weak(v, 7, std::memory\_order\_consume, std::memory\_order\_consume);//条件成立時：OK  //v = o.compare\_exchange\_weak(v, 7, std::memory\_order\_acquire, std::memory\_order\_acquire);//条件成立時：OK  ////v = o.compare\_exchange\_weak(v, 7, std::memory\_order\_release, std::memory\_order\_release);  // //条件成立時：Assertion failed! \_Order2 != memory\_order\_release  ////v = o.compare\_exchange\_weak(v, 7, std::memory\_order\_acq\_rel, std::memory\_order\_acq\_rel);  // //条件成立時：Assertion failed! \_Order2 != memory\_order\_acq\_rel  //v = o.compare\_exchange\_weak(v, 7, std::memory\_order\_seq\_cst, std::memory\_order\_seq\_cst);//条件成立時：OK  //v = o.compare\_exchange\_weak(v, 8, std::memory\_order\_relaxed, std::memory\_order\_relaxed);//条件不成立時：OK  //v = o.compare\_exchange\_weak(v, 8, std::memory\_order\_consume, std::memory\_order\_consume);//条件不成立時：OK  //v = o.compare\_exchange\_weak(v, 8, std::memory\_order\_acquire, std::memory\_order\_acquire);//条件不成立時：OK  ////v = o.compare\_exchange\_weak(v, 8, std::memory\_order\_release, std::memory\_order\_release);  // //条件不成立時：Assertion failed! \_Order2 != memory\_order\_release  ////v = o.compare\_exchange\_weak(v, 8, std::memory\_order\_acq\_rel, std::memory\_order\_acq\_rel);  // //条件不成立時：Assertion failed! \_Order2 != memory\_order\_acq\_rel  //v = o.compare\_exchange\_weak(v, 8, std::memory\_order\_seq\_cst, std::memory\_order\_seq\_cst);//条件不成立時：OK  //【メモリ更新と読み込みの順序性保証】  //CPUのメモリバリアを使うかどうか？  //（例）下記のような処理で、スレッドＢは変数 b をチェックしているにもか関わらず、  // 先に更新されたはずの変数 a の値が 12 と表示されないようなことが起こりえる。  // これは、CPUにメモリバリア命令が出されて順序性を保証しないと、CPUの最適化により、  // 更新の順序が入れ替わることがあるため。（「アウト・オブ・オーダー実行」と呼ばれる最適化）  //[グローバル変数]  //volatile bool flg\_a;  //volatile bool flg\_b;  //volatile int val\_a;  //volatile int val\_b;  //[スレッドA]  //val\_a = 123;  //flg\_a = true;  //if(flg\_b)  // std::cout << val\_b;  //[スレッドB]  //val\_b = 789;  //flg\_b = true;  //if(flg\_a)  // std::cout << val\_a;  //※この問題は、必要な箇所で必要なメモリバリアを明示することで対処できる  //[グローバル変数]  //std::atmic<bool> flg\_a;  //std::atmic<bool> flg\_b;  //volatile int val\_a;  //volatile int val\_b;  //[スレッドA]  //val\_a = 123;  //flg\_a.store(true, std::memory\_order\_release);  //if(flg\_b.load(std::memory\_order\_acquire))  // std::cout << val\_b;  //[スレッドB]  //val\_b = 789;  //flg\_b.store(true, std::memory\_order\_release );  //if(flg\_a.load(std::memory\_order\_acquire))  // std::cout << val\_a;  //http://msdn.microsoft.com/ja-jp/library/hh874684.aspx  //template<class Ty>  //bool std::atomic::compare\_exchange\_\*\*\*(Ty& Expected, Ty NewValue, std::memory\_order SuccessOrder, std::memory\_order FailureOrder) \_NOEXCEPT;  //・std::atomic::compare\_exchange\_weak() ... \*this で弱いアトミックの比較および交換操作を実行します。  //・std::atomic::compare\_exchange\_strong() ... \*this に対してアトミックの比較および交換の操作を実行します。  //【共通】  // このアトミックの比較および交換の操作では、\*this に格納されている値と Expected を比較します。  // 値が等しい場合、操作は read-modify-write 操作を使用して、  // SuccessOrder によって指定されたメモリ順序制約を適用して、  // \*this に格納された値を NewValue と置き換えます。  // 値が等しくない場合、操作は \*this に格納されている値を使用して Expected を置き換え、  // FailureOrder によって指定されたメモリ順序制約を適用します。  //  // 2 番目の memory\_order がないオーバーロードは、SuccessOrder の値に基づく暗黙の FailureOrder を使用します。  // SuccessOrder が memory\_order\_acq\_rel の場合、FailureOrder は memory\_order\_acquire です。  // SuccessOrder が memory\_order\_release の場合、FailureOrder は memory\_order\_relaxed です。  // 他のすべての場合、FailureOrder は SuccessOrder と等しくなります。  //  // 2 つの memory\_order パラメーターを受け取るオーバーロードの場合、  // FailureOrder の値は memory\_order\_release または memory\_order\_acq\_rel ではない必要があり、  // かつ SuccessOrder の値よりも強い値ではない必要があります。  //  //【compare\_exchange\_weak() のみの解説】  // 比較された値が同一の場合、弱いアトミック比較および交換操作は交換を実行します。  // 値が同じでない場合、操作による交換の実行は保証されません。  //  //【メソッドの使い分け】 http://d.hatena.ne.jp/yohhoy/20120725/p1  //・アルゴリズム上、CAS操作をループで括る必要があれば compare\_exchange\_weak を利用する。  //　（例：一般的なLock-Freeアルゴリズム実装など）  //・アルゴリズム上の制約が無く、かつspurious failureを許容しない場合は compare\_exchange\_strong を利用する。  //　（例：pthread spinlock相当のtrylock操作実装 \* 2）  //【メソッドの使い分け要約】  //・ループ処理は compare\_exchange\_weak でOK。  //・ループ処理以外の（交換の取りこぼしがあってはならない）場面では compare\_exchange\_strong を使用。  //【SuccessOrderとFailureOrderの使い分け要約】  //・SuccessOrderは、条件成立時の \*this ← NewValue に用いられる。「読み書き可能」なメモリオーダーである必要がある。  //・FailureOrderは、条件不成立時の Expected ← \*this に用いられる。「書き込み可能」なメモリオーダーである必要がある。 |

## Call Once

「Call Once」（コール・ワンス）により、多数のスレッドを実行した際に、最初の1回だけ実行したい処理を設定することができる。

初期化処理に有用であるため、いつインスタンス化されるかわからない「シングルトンパターン」に応用されたりしている。

CallOnceを活用したスレッドセーフなシングルトンパターンについては、別紙の「効率化と安全性のためのロック制御」にて詳述する。

### POSIXスレッドライブラリ版

POSIXスレッドライブラリ版のCallOnce。

POSIXスレッドライブラリ版のCallOnceのサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <unistd.h>  //CallOnce制御用変数  pthread\_once\_t s\_onceFlag = PTHREAD\_ONCE\_INIT;  //CallOnce処理関数  void callOnceFunc()  {  //CallOnce処理開始  printf("CallOnce処理：初期化中...\n");  fflush(stdout);  //1秒スリープ  sleep(1);  //CallOnce処理終了  printf("CallOnce処理：終了\n");  fflush(stdout);  }  //スレッド  void\* threadFunc(void\* param\_p)  {  //パラメータ受け取り  const char\* name = static\_cast<const char\*>(param\_p);  //開始  printf("- begin:%s -\n", name);  fflush(stdout);  //CallOnce処理  pthread\_once(&s\_onceFlag, callOnceFunc);  //終了  printf("- end:%s -\n", name);  fflush(stdout);  return 0;  }  //テスト  int main(const int argc, const char\* argv[])  {  //スレッド作成  static const int THREAD\_NUM = 3;  pthread\_t pth[THREAD\_NUM];  {  pthread\_attr\_t attr;  pthread\_attr\_init(&attr);  pthread\_attr\_setstacksize(&attr, 1024);//スタックサイズ指定  pthread\_create(&pth[0], &attr, threadFunc, (void\*)"太郎");  pthread\_create(&pth[1], &attr, threadFunc, (void\*)"次郎");  pthread\_create(&pth[2], &attr, threadFunc, (void\*)"三郎");  }    //スレッド終了待ち  for(int i = 0; i < THREAD\_NUM; ++i)  {  pthread\_join(pth[i], NULL);  }    return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:太郎 -  CallOnce処理：初期化中... ←3つスレッドを起動したが、CallOnce関数は最初のスレッドだけが実行している  - begin:次郎 -  - begin:三郎 -  CallOnce処理：終了 ←CallOnce関数完了まで、全てのスレッドがブロックされており、  - end:太郎 - 　完了後は一斉に動き出している  - end:次郎 -  - end:三郎 - |

### C++11版

C++11版のCallOnce。内部的にはミューテックスで排他制御している。

通常関数の呼び出しだけではなく、関数オブジェクトやラムダ式にも使えるため、関数スコープのstatic変数の初期化などにも安全に活用できる。

C++11版のCallOnceのサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <thread>  #include <mutex>  #include <chrono> //時間計測用  //スレッド  void threadFunc(const char\* name)  {  //開始  printf("- begin:%s -\n", name);  fflush(stdout);  //CallOnce制御用変数  static std::once\_flag s\_onceFlag;    //CallOnce処理  std::call\_once(s\_onceFlag, [&name]()  {  //ラムダ式で直接記述    //CallOnce処理開始  printf("CallOnce処理：初期化中(by %s)...\n", name);  fflush(stdout);  //1秒スリープ  std::this\_thread::sleep\_for(std::chrono::seconds(1));  //CallOnce処理終了  printf("CallOnce処理：終了\n");  fflush(stdout);  }  );  //終了  printf("- end:%s -\n", name);  fflush(stdout);  }  //テスト  int main(const int argc, const char\* argv[])  {  //スレッド作成  std::thread thread\_obj1 = std::thread(threadFunc, "太郎");  std::thread thread\_obj2 = std::thread(threadFunc, "次郎");  std::thread thread\_obj3 = std::thread(threadFunc, "三郎");    //スレッド終了待ち  thread\_obj1.join();  thread\_obj2.join();  thread\_obj3.join();  return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| - begin:太郎 -  CallOnce処理：初期化中(by 太郎)... ←3つスレッドを起動したが、CallOnce関数は最初のスレッドだけが実行している  - begin:次郎 - ※POSIX版と異なり、CallOnce処理にパラメータを渡すことができている  - begin:三郎 -  CallOnce処理：終了  - end:太郎 - ←CallOnce関数完了まで、全てのスレッドがブロックされており、  - end:三郎 - 　完了後は一斉に動き出している  - end:次郎 - |

## 先物（promise と future）

結果を格納する予定の変数を先に別スレッドに渡し、結果が出た時点でそのスレッド側でも参照可能になる変数を扱う事ができる。

### C++11版

C++11版の先物。

前述の「C++11の非同期関数」で紹介した「std::future<>」を用いる。

モニター処理の一種として活用できる。

C++11版の先物のサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <future>  #include <thread>  #include <chrono> //時間計測用  //結果表示スレッド  void threadFunc(const char\* name, std::shared\_future<int> value)  {  printf("%s: [BEFORE]\n", name);  fflush(stdout);  printf("%s: [AFTER] value=%d\n", name, value.get());//future::get() は、結果が格納されるまで処理がブロックされる  fflush(stdout);  }  //テスト  int main(const int argc, const char\* argv)  {  //処理開始  printf("main: [START]\n");  fflush(stdout);  //先物変数の用意  //※後で値を格納する変数  std::promise<int> p\_val;  //結果待機オブジェクトを用意  //※先物変数や非同期関数の結果を待機するオブジェクト。  //※結果を受け取る側がスレッドの場合、shared\_futureクラスを使用。  std::shared\_future<int> f\_val = p\_val.get\_future().share();  //スレッド生成  static const char\* names[] = { "太郎", "次郎", "三郎" };  for(auto name : names)  {  //スレッド生成  std::thread th = std::thread(threadFunc, name, f\_val);    //スレッドの完了を待たないので、切り離しておく  th.detach();  }  //1秒スリープ  printf("main: [sleep ...]\n");  fflush(stdout);  std::this\_thread::sleep\_for(std::chrono::seconds(1));  //値を格納  //※この時、スレッド側の future::get() が反応する  printf("main: [set\_value()]\n");  fflush(stdout);  p\_val.set\_value(123);  //1ミリ秒スリープ  std::this\_thread::sleep\_for(std::chrono::milliseconds(1));  //処理終了  printf("main: [END]\n");  fflush(stdout);    return EXIT\_SUCCESS;  } |

↓（実行結果）

|  |
| --- |
| main: [START]  太郎: [BEFORE]  次郎: [BEFORE]  三郎: [BEFORE]  main: [sleep ...] ←メインループで1秒のスリープ（この間スレッドは3つともブロックされた状態）  main: [set\_value()]  三郎: [AFTER] value=123 ←メインループで先物変数に値を格納すると、スレッドが一斉に動き出す  次郎: [AFTER] value=123  太郎: [AFTER] value=123  main: [END] |

## 割り込み：シグナル

最後にUnix特有の「シグナル」を説明する。

シグナルは基本的に割り込み処理である。例えば、[Ctrl]+[C]キーを押すと、プロセスに対して割り込みシグナル（SIGINT）が発行され、プログラムが中断される。

この時、プログラム中にSIGINTの割り込みハンドラ（関数）が設定されていると、割り込み発生時にハンドラ関数が呼び出され、中断を回避して独自の処理を行うことができる。

この仕組みを利用して、プログラム中で任意のタイミングでシグナルを発行し、ハンドラ関数に処理をリレーするプログラミング手法を使うことができる。

割り込み発生時は、スレッドの切り替えと同様のコンテキストスイッチが発生するが、専用のスレッドではなく、あくまでも割り込みであるため、その時動作中のスレッドのスタックが使用される。

割り込み処理中は、 malloc や printf などの「非割り込みセーフな関数」を使えないといった制約もある。

シグナルは、割り込みだけではなく、Windowsのウインドウメッセージのような通知に用いることもできる。

「同期シグナル」と呼ばれる手法で、スレッドにシグナルをキューイングして、ループ処理で一つずつ処理することができる。この場合、「非割り込みセーフな関数」の制約は受けない。

この仕組みをスレッド間の通知処理に利用して同期を取ることができる。

シグナルハンドラにせよ、同期シグナルにせよ、本来OSがプロセスの制御のために使用している仕組みなので、OSの影響を強く受け、自由度が少ないことには注意が必要。

### POSIX版

POSIX版のシグナル。

シグナルハンドラと、スレッドを用いた同期シグナルの両方のサンプルとしている。

アラームシグナルを利用し、1秒ごとに定期起動する割り込み処理も実装している。

なお、LinuxとCygwinで大きく挙動が異なったため、両方の結果を記載しておく。

POSIX版のシグナルのサンプル：

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <memory.h>  #include <pthread.h>  #include <unistd.h>  #include <signal.h>  #include <sys/time.h> //時間計測用  //共有データ  static int s\_commonData = 0;  //スレッド固有データ  \_\_thread int s\_tlsData = 0;  //スレッド情報  pthread\_t s\_threadWathc;  pthread\_t s\_threadSyncSignal;  //受信シグナルバッファ（シグナルハンドラ内処理用）  //※シグナルハンドラ内では「非同期安全な関数」以外使用できない。malloc や prnitf もNG。  //　そのため、シグナルハンドラで受信したシグナルをプールし、  //　専用の処理スレッドに受け渡すようにする。  //※サンプルとしてややこしく見えるが、「受信シグナルバッファ」は、  //　あくまでも受信したシグナルを画面に表示するために作った仕組みであり、  //　シグナルハンドラーや同期シグナルを扱うために必須の処理ではない  struct SIG\_INFO  {  int sig;  pthread\_t pth;  int tls;  SIG\_INFO& operator =(const SIG\_INFO& o)  {  memcpy(this, &o, sizeof(\*this));  return \*this;  }  };  static const int SIGNAL\_RCV\_MAX = 5; //最大受信シグナル数  static volatile int s\_sigBuffCnt = 0; //受信シグナル数  static SIG\_INFO s\_sigBuff[SIGNAL\_RCV\_MAX];//受信シグナルバッファ  static pthread\_cond\_t s\_sigBuffC = PTHREAD\_COND\_INITIALIZER; //受信シグナルバッファ用条件変数  static pthread\_mutex\_t s\_sigBuffM = PTHREAD\_MUTEX\_INITIALIZER;//受信シグナルバッファ用ミューテックス  //同期シグナル用のシグナルマスク  static sigset\_t s\_sigSet;  //シグナル受信バッファ登録  bool pushSigBuff(SIG\_INFO& info)  {  bool result = false;  pthread\_mutex\_lock(&s\_sigBuffM);  if(s\_sigBuffCnt < SIGNAL\_RCV\_MAX)  {  s\_sigBuff[s\_sigBuffCnt++] = info;  pthread\_cond\_signal(&s\_sigBuffC);  result = true;  }  pthread\_mutex\_unlock(&s\_sigBuffM);  return result;  }  //シグナル受信バッファ取り出し  SIG\_INFO popSigBuff(bool is\_lock = true)  {  SIG\_INFO info = {-1, 0, 0};  if(is\_lock)  pthread\_mutex\_lock(&s\_sigBuffM);  if(s\_sigBuffCnt > 0)  {  info = s\_sigBuff[0];  --s\_sigBuffCnt;  for(int i = 0; i < s\_sigBuffCnt; ++i)  {  s\_sigBuff[i] = s\_sigBuff[i + 1];  }  }  if(is\_lock)  pthread\_mutex\_unlock(&s\_sigBuffM);  return info;  }  //シグナル受信待ち受け  SIG\_INFO watchSigBuff()  {  //シグナル受信バッファへの登録待ち  pthread\_mutex\_lock(&s\_sigBuffM);  while (s\_sigBuffCnt == 0)  pthread\_cond\_wait(&s\_sigBuffC, &s\_sigBuffM);  SIG\_INFO info = popSigBuff(false);  pthread\_mutex\_unlock(&s\_sigBuffM);  return info;  }  //シグナル名取得  const char\* getSigName(int sig)  {  #define CASE\_TO\_STR(x) case x: return #x; break  switch(sig)  {  CASE\_TO\_STR(SIGINT);  CASE\_TO\_STR(SIGTSTP);  CASE\_TO\_STR(SIGCONT);  CASE\_TO\_STR(SIGQUIT);  CASE\_TO\_STR(SIGTRAP);  CASE\_TO\_STR(SIGUSR1);  CASE\_TO\_STR(SIGUSR2);  CASE\_TO\_STR(SIGALRM);  }  #undef CASE\_TO\_STR  static char unknown[32];  sprintf(unknown, "(unknown:%d)", sig);  return unknown;  }  //シグナル情報作成  SIG\_INFO makeSigInfo(int sig)  {  SIG\_INFO info = {sig, pthread\_self(), s\_tlsData};  return info;  }  //シグナル情報表示  void printSigInfo(SIG\_INFO& info)  {  printf("[%s] (thread=%x) tls=%d\n", getSigName(info.sig), info.pth, info.tls);  fflush(stdout);  }  void printSigInfo(int sig)  {  SIG\_INFO info = makeSigInfo(sig);  printSigInfo(info);  }  //汎用シグナルハンドラ関数（スレッドではない）  //※シグナルハンドラは「プロセス」に対するシグナルをハンドリングする関数  void signalHandler(int sig)  {  SIG\_INFO info = makeSigInfo(sig);  pushSigBuff(info);  }  //アラーム用シグナルハンドラ関数  void alarmFunc(int sig)  {  if(s\_tlsData++ < 10)  {  //汎用シグナルハンドラにシグナルを送る  signalHandler(sig);    //1秒後に再アラーム  alarm(1);  }  else  {  //規定回数アラームを発行したら終了する（再アラームしない）    //汎用シグナルハンドラにSIGINT（割り込み：CTRL + C）を送る  signalHandler(SIGINT);  }  }  //同期シグナルを送信  void sendSyncSignal(int sig)  {  //同期シグナルスレッドが受け付け可能なシグナルを送信  if(sigismember(&s\_sigSet, sig))  {  printf("sendSyncSignal:[%s] (thread=%x) tls=%d\n", getSigName(sig), pthread\_self(), s\_tlsData);  fflush(stdout);    //スレッドにシグナルを送信  pthread\_kill(s\_threadSyncSignal, sig);  }  }  //非同期シグナル受信バッファ監視スレッド  void\* threadFuncWatch(void\* param\_p)  {  //開始  printf("- begin:threadFuncWatch(%x) -\n", pthread\_self());  fflush(stdout);    //非同期シグナルバッファ受信処理  while(1)  {  //受信バッファが溜まるまで待機  SIG\_INFO info = watchSigBuff();    //受信したシグナルを表示  printSigInfo(info);    //シグナルをリレーする  sendSyncSignal(info.sig);    //シグナルがSIGINTなら終了  if(info.sig == SIGINT)  {  //同期シグナルに SIGTRAP を送って終了をうながす  sendSyncSignal(SIGTRAP);  break;  }  }    //若干スリープ  sleep(1);    //終了  printf("- end:threadFuncWatch(%x) -\n", pthread\_self());  fflush(stdout);  return 0;  }  //同期シグナルスレッド  void\* threadFuncSyncSignal(void\* param\_p)  {  //開始  printf("- begin:threadFuncSyncSignal(%x) -\n", pthread\_self());  fflush(stdout);    //joinせずにスレッドを終了させるために、スレッドをデタッチする  //※スレッド生成時に最初からデタッチ状態の属性を与えることも可能。  pthread\_detach(pthread\_self());    //シグナルマスクをセット  pthread\_sigmask(SIG\_BLOCK, &s\_sigSet, NULL);  //※これにより、ここで設定したシグナルは、このスレッドに  //　キューイングされるようになり、sigwait() で取り出せるようになる。  //【Linuxの挙動】  //※プロセスに対するシグナルには反応しない。  //【Cygwinの挙動】  //※プロセスに対するシグナルに反応する。  //　先に同じシグナルに対するシグナルハンドラが設定されていても、  //　こちらが優先される。（両方は反応しない）    //同期シグナル処理  while(1)  {  //シグナル受信待ち  //【Linuxの挙動】  //※マスク対象シグナル以外のシグナルが直接送られてくると  //　普通に受信するものの、キューから削除されず復帰できない。  //【Cygwinの挙動】  //※マスク対象シグナル以外のシグナルが直接送られてくると  //　sigwait() がエラーを返す。  //　キューから削除されないので、復帰できない。  int sig;  if(sigwait(&s\_sigSet, &sig) != 0)  {  fprintf(stderr, "sigwait() failed.\n");  fflush(stderr);  break;  }    //カウンターを進める  ++s\_tlsData;    //シグナル表示  printSigInfo(sig);    //シグナルがSIGINTもしくはSIGTRAPなら終了  if(sig == SIGINT || sig == SIGTRAP)  break;  }    //終了  printf("- end:threadFuncSyncSignal(%x) -\n", pthread\_self());  fflush(stdout);  return 0;  }  //テスト  int main(const int argc, const char\* argv[])  {  //開始  printf("- begin:main(%x) -\n", pthread\_self());  fflush(stdout);    //同期シグナル（スレッド）用のシグナルマスク作成  sigemptyset(&s\_sigSet);  // sigaddset(&s\_sigSet, SIGINT); //割り込み（CTRL + C）  sigaddset(&s\_sigSet, SIGQUIT);//終了とコアダンプ（CTRL + \）  sigaddset(&s\_sigSet, SIGCONT);//停止していれば再開  sigaddset(&s\_sigSet, SIGTSTP);//端末からの中断シグナル（CTRL + Z）  sigaddset(&s\_sigSet, SIGTRAP);//トレース/ブレークポイントによるトラップ  // sigaddset(&s\_sigSet, SIGUSR1);//ユーザー定義シグナル 1  sigaddset(&s\_sigSet, SIGUSR2);//ユーザー定義シグナル 2    //スレッド作成  static const int THREAD\_NUM = 1;  pthread\_t pth[THREAD\_NUM];  {  pthread\_attr\_t attr;  pthread\_attr\_init(&attr);  pthread\_attr\_setstacksize(&attr, 1024);//スタックサイズ指定  pthread\_create(&s\_threadSyncSignal, &attr, threadFuncSyncSignal, NULL);//同期シグナルスレッド  pthread\_create(&s\_threadWathc, &attr, threadFuncWatch, NULL); //非同期シグナル受信バッファ監視スレッド  }    //シグナルハンドラを登録　※非同期シグナル処理  signal(SIGINT, signalHandler);//割り込み（CTRL + C）  signal(SIGQUIT, signalHandler);//終了とコアダンプ（CTRL + \）  signal(SIGCONT, signalHandler);//停止していれば再開  signal(SIGTSTP, signalHandler);//端末からの中断シグナル（CTRL + Z）  // signal(SIGTRAP, signalHandler);//トレース/ブレークポイントによるトラップ  signal(SIGUSR1, signalHandler);//ユーザー定義シグナル 1  // signal(SIGUSR2, signalHandler);//ユーザー定義シグナル 2    //アラーム用のシグナルハンドラを登録　※非同期シグナル処理  signal(SIGALRM, alarmFunc);    //1秒後にアラーム  //※アラームとスリープを同時に使用すると、sleepが効かないので注意！  alarm(1);    //スレッド終了待ち  pthread\_join(s\_threadWathc, NULL);  //※同期シグナルスレッドはデタッチしているので終了待ちしない    //終了  printf("- end:main(%x) -\n", pthread\_self());  fflush(stdout);  return EXIT\_SUCCESS;  } |

実行用のシェルスクリプト：※外部からのシグナル発行

|  |
| --- |
| ./signal & ←プログラムを実行（プロセスを生成）  sleep 1  kill -USR1 %1 ←1秒後にSIGUSR1 シグナルをプロセスに送信  #この処理をLinuxで実行するとプログラムが停止する  #sleep 1  #kill -USR2 %1 ←1秒後にSIGUSR2 シグナルをプロセスに送信  sleep 1  kill -TSTP %1 ←1秒後にSIGTSTP シグナルをプロセスに送信  sleep 1  kill -CONT %1 ←1秒後にSIGCONT シグナルをプロセスに送信  sleep 1  kill -QUIT %1 ←1秒後にSIGQUIT シグナルをプロセスに送信  #この処理をLinuxで実行するとプログラムが停止する  #Cygwinで実行すると同期シグナルハンドラスレッドのみが停止する  #sleep 1  #kill -TRAP %1 ←1秒後にSIGTRAP シグナルをプロセスに送信  sleep 1  kill -INT %1 ←1秒後にSIGUINT シグナルをプロセスに送信 |

↓（実行結果：Linux）

|  |
| --- |
| - begin:main(2dce1720) - ←メインスレッド開始  - begin:threadFuncSyncSignal(2dcdf700) - ←同期シグナルスレッド開始  - begin:threadFuncWatch(2d2de700) - ←シグナル受信バッファ監視スレッド開始  [SIGUSR1] (thread=2dce1720) tls=0 ←メインスレッドでシグナルハンドラが起動：SIGUSR1受信  （同期シグナルスレッドが対応しないシグナル）  [SIGALRM] (thread=2dce1720) tls=1 ←メインスレッドでアラームが起動  [SIGALRM] (thread=2dce1720) tls=2 ←メインスレッドでアラームが起動  [SIGTSTP] (thread=2dce1720) tls=2 ←メインスレッドでシグナルハンドラが起動：SIGTSTP受信  sendSyncSignal:[SIGTSTP] (thread=2d2de700) tls=0 ←シグナル受信バッファ監視スレッドから、  [SIGTSTP] (thread=2dcdf700) tls=1 ←　同期シグナルスレッドに SIGTSTP をリレー  [SIGALRM] (thread=2dce1720) tls=3 ←メインスレッドでアラームが起動  [SIGCONT] (thread=2d2de700) tls=0 ←シグナル受信バッファ監視スレッドでシグナルハンドラが起動：  　　　　　　　　　　　　　　　　　　　　　　　　　SIGCONT受信  sendSyncSignal:[SIGCONT] (thread=2d2de700) tls=0 ←シグナル受信バッファ監視スレッドから、  [SIGCONT] (thread=2dcdf700) tls=2 ←　同期シグナルスレッドに SIGCONT をリレー  [SIGALRM] (thread=2dce1720) tls=4 ←メインスレッドでアラームが起動  [SIGQUIT] (thread=2dce1720) tls=4 ←メインスレッドでシグナルハンドラが起動：SIGQUIT受信  sendSyncSignal:[SIGQUIT] (thread=2d2de700) tls=0 ←シグナル受信バッファ監視スレッドから、  [SIGQUIT] (thread=2dcdf700) tls=3 ←　同期シグナルスレッドに SIGQUIT をリレー  [SIGALRM] (thread=2dce1720) tls=5 ←メインスレッドでアラームが起動  [SIGINT] (thread=2dce1720) tls=5 ←メインスレッドでシグナルハンドラが起動：SIGINT受信  sendSyncSignal:[SIGTRAP] (thread=2d2de700) tls=0 ←シグナル受信バッファ監視スレッドから、  [SIGTRAP] (thread=2dcdf700) tls=4 ←　同期シグナルスレッドに SIGTRAP としてをリレー  - end:threadFuncSyncSignal(2dcdf700) - ←同期シグナルスレッド終了  - end:threadFuncWatch(2d2de700) - ←シグナル受信バッファ監視スレッド終了  - end:main(2dce1720) - ←メインスレッド終了 |

↓（実行結果：Cygwin）

|  |
| --- |
| - begin:main(80000038) - ←メインスレッド開始  - begin:threadFuncSyncSignal(80048308) - ←同期シグナルスレッド開始  - begin:threadFuncWatch(80048398) - ←シグナル受信バッファ監視スレッド終了  [SIGALRM] (thread=80000038) tls=1 ←メインスレッドでアラームが起動  [SIGUSR1] (thread=80000038) tls=1 ←メインスレッドでシグナルハンドラが起動：SIGUSR1受信  （同期シグナルスレッドが対応しないシグナル）  [SIGALRM] (thread=80000038) tls=2 ←メインスレッドでアラームが起動  [SIGUSR2] (thread=80048308) tls=1 ←同期シグナルスレッドでシグナルを直接受信：SIGUSR2  （シグナルハンドラが対応しないシグナル）  [SIGALRM] (thread=80000038) tls=3 ←メインスレッドでアラームが起動  [SIGTSTP] (thread=80048308) tls=2 ←同期シグナルスレッドでシグナルを直接受信：SIGUSR2  　（シグナルハンドラも登録されていたが、同期シグナル優先で  　　反応していないか、先にスレッドが処理してしまった）  [SIGALRM] (thread=80000038) tls=4 ←メインスレッドでアラームが起動  [SIGCONT] (thread=80048308) tls=3 ←同期シグナルスレッドでシグナルを直接受信：SIGCONT  　（シグナルハンドラも登録されていたが、同期シグナル優先で  　　反応していないか、先にスレッドが処理してしまった）  [SIGALRM] (thread=80000038) tls=5 ←メインスレッドでアラームが起動  [SIGQUIT] (thread=80048308) tls=4 ←同期シグナルスレッドでシグナルを直接受信：SIGQUIT  　（シグナルハンドラも登録されていたが、同期シグナル優先で  　　反応していないか、先にスレッドが処理してしまった）  [SIGALRM] (thread=80000038) tls=6 ←メインスレッドでアラームが起動  [SIGINT] (thread=80000038) tls=6 ←メインスレッドでシグナルハンドラが起動：SIGINT受信  sendSyncSignal:[SIGTRAP] (thread=80048398) tls=0 ←シグナル受信バッファ監視スレッドから、  [SIGTRAP] (thread=80048308) tls=5 ←　同期シグナルスレッドに SIGTRAP としてをリレー  - end:threadFuncSyncSignal(80048308) - ←同期シグナルスレッド終了  - end:threadFuncWatch(80048398) - ←シグナル受信バッファ監視スレッド終了  - end:main(80000038) - ←メインスレッド終了 |

### 【用語解説】リエントラント

「リエントラント」（reentrant）とは「再入可能」という意味である。

割り込み処理後に元の処理に戻っても問題を起こさない「割り込みセーフ」なプログラムの意味である。

マルチスレッド環境下で、メインスレッド以外のスレッドで実行しても問題のないプログラムのことを「スレッドセーフ」と呼ぶ。それに対して、割り込み処理で実行しても問題のないプログラムのことを「リエントラント」と呼ぶ。

メモリ確保やファイルアクセスなどの「非割り込みセーフ」な処理や、他の処理が使用するグローバル変数を更新するような処理があると「リエントラント」な処理ではなくなる。

なお、「リエントラント」の定義は、シングルスレッドプログラミング環境が起源である。

処理中にロックを行うスレッドセーフな処理も、リエントラントにはならないので注意が必要である。

ロック中のスレッドで割り込みが発生して、その割り込み処理の中でスレッドセーフな処理を呼び出し、またロックを取得しようとするとデッドロックが起きてしまう。スレッドセーフでかつリエントラントな処理とするには、ロックの前後でさらに割り込みの禁止／許可を行う必要もある。

割り込みは旧来からの技術ではあるが、非常に扱いにくい。OSの内部などでは頻繁に活用されているものの、よほど効果的な場面でもない限りは安易に用いないほうが良い。マルチスレッドが最適な状態になるように注力するためにも、リエントラントな処理は不要な状態が望ましい。

# マルチスレッドで起こり得る問題②

ここまでは、同期の不備によって起こる問題と、それを解決する同期の手法を説明した。

ここからは、その同期によって起こる問題を説明する。

## デッドロック

「デッドロック」とは、その名の通り「死んだロック」のこと。解放される見込みがないロックのことである。

デッドロックはマルチスレッドプログラミングで最も典型的な問題の一つ。

### 【問題】相互ロック

二つのロックを用いてスレッドが互いにロックを取得し合うことで、スレッドが動かなくなってしまう問題である。



### 【解決策】相互ロック

上記の例ではっきりとした問題なのは、ロック取得の順序が二つの処理で異なっていることである。

スレッドA ①のロック取得　→　②のロックを取得　の処理順序

スレッドB ②のロック取得　→　①のロックを取得　の処理順序

このように、ロック取得の順序が異なる処理が混在するのが典型的なデッドロックのパターンである。

主な解決策は下記の通り。

* 【原則】ロック取得の順序を全ての処理で統一する。
* 【可能なら】複数のロック取得を同時に行わない。（それが可能かよく検討する）
* 【可能なら】ロックオブジェクトを一つに統一する。（それが可能かよく検討する）
* 【望ましくない解決策】ロック取得のタイムアウトやトライロック処理をつかってエラー処理する。（その後のつじつまあわせに要注意）
* 【最悪のケース：共通処理を利用する都合などから、どうしても順不同で複数のロック取得が必要な場合】

必要なアトミック操作の範囲をよく考え直し、処理全体を包括する大きなロックをさらに追加する。（パフォーマンスの劣化も招くため、望ましくない解決策）

* 上記の例の場合、さらに「ミューテックス③」を追加し、スレッドAもスレッドBも、一連の処理全体をミューテックス③でロックする。



* 【POSIXスレッドライブラリ版】デッドロック検出属性付きのミューテックスを使用する。ただし、これは動作の重いデバッグ用。

### 【問題】再帰ロック

一つのスレッドが2回ロックを取得しようとすることで、スレッドが動かなくなってしまう問題である。

再帰ロックに対しては、各同期処理で挙動が異なっている。前述の「様々な同期手法」の一覧には、それぞれがどのような再帰ロックの挙動となっているかをまとめている。



### 【解決策】再帰ロック

主な解決策は下記の通り。

* 【原則】再帰的なロック取得をしないようにする。
* 【望ましくない解決策】ロック取得のタイムアウトやトライロック処理をつかってエラー処理する。
* その後のつじつまあわせに要注意。例えば、「トライロックでロックを取得できなかった場合は再帰ロックとみなして処理を行えるものとする」とした場合、その内側の処理でロックを解放した時点で、外側の処理が残っていてもロックが解放されてしまう。特に、Windows版のミューテックスは再帰ロックを無視するので、デフォルトでこのような挙動となることに要注意。
* 【よく考えて用いるべき解決策】再帰ロック有効なロック手法を用いる。
* POSIXスレッドライブラリ版ミューテックス＋再帰属性、Win32APIのクリティカルセクション、C++11版の再帰ミューテックスなどが対応している。

## 【モニターの問題】通知の取りこぼし

条件変数やイベントなどの「モニター」を使用する場合、適切に使用しないと、通知を取りこぼしてデッドロックと同じ状態に陥ることがある。



### 解決策

主な解決策は下記の通り。

* 【原則】条件変数を適切に使用し、待機前に通知があったと判断したら、待機処理を行わない。
* 【Windowsの場合】Win32APIの「イベント」は、待機前に通知（「シグナル状態」と呼ぶ）があったら、シグナル状態が継続して、その後の待機が即座に完了するのでこの問題は起こらない。ただし、Windowsイベントには注意点も多い。
* 「シグナル状態」（通知状態）と「非シグナル状態」（リセット状態）という単純なフラグしか扱えないので、かなり融通が利かない。モニター処理では、２ステップ以上のステートを扱いたいことも多いが、イベントで扱えるのは１ステップの状態通知のみ。
* Windowsイベントは、シグナル状態（通知）に反応して待機が解除されると、自動的に非シグナル状態（リセット）になる。この動作を自動で行わせず、手動操作にするオプションがあるが、通知側の処理間隔が十分長い場合以外はかなり危険。通知を受けてから手動リセットまでの間に次の通知が来ると、それを取り下げることになってしまう。手動リセットは使用禁止にすべき。
* 待機中のスレッド全てにシグナル状態を通知する「PulseEvent」というメソッドを用いた場合、条件変数と同じくその時待機中のスレッドしか反応しないので要注意。「PulseEvent」は使用禁止にした方が賢明。
* 【代替策】アトミック操作（インターロック操作）を使用してモニターする。
* 待機時間が長いと分かっている処理なら、モニター処理ループ中に毎ループスリープを入れる。あまり長めのスリープにすると、通知に対して反応が遅れるので要注意。
* 特にマルチコア環境で、かつ、モニターする処理時間が極めて短いとわかっている時は（例えば0.1ミリ秒未満）、条件変数よりも効果的な選択となる。

## 低速化

マルチスレッドプログラミングにおいて、深く考えずにコーディングしていると深刻な低速化を招く。主な問題の事例を示す。

* 【問題】多すぎるスレッド
* 大量のスレッドを作っても高速化するわけではない。
* むしろ、コンテキストスイッチの機会が増えて低速化する。
* スレッドの生成自体も多少時間がかかる。
* 【問題】過剰なロック
* データの不整合を恐れて、とにかくロックしすぎる。
* 待機なしでロックを取得するだけでも、それなりの処理時間がかかる。
* 【問題】過剰なvolatile型修飾子、アトミック型
* volatile型修飾子、アトミック型を使いすぎて、コンパイラ・CPUの最適化が働かずに処理効率が悪い。

### 解決策

主な解決策は下記の通り。

* 【解決策】多すぎるスレッド
* 並行処理の必要がないものまでスレッド化しない。
* SPURSのようなジョブキューイング方式のシステムを作成し、並行処理の要求を待機させることが出来るようにし、対象ハードウェアに合わせて並行処理の数を制限する。
* 【あまりよくない案】スレッドはとりあえず大量に作り、同時稼働数をセマフォで制限する。スレッドを大量生成することによるメモリの問題はあるが、手っ取り早い解決策には使える。
* 【解決策】過剰なロック
* 可能な限りロックフリーな処理にする。
* ロックフリーな処理のためには、メモリバリアの考慮が不可欠なので、volatile型修飾子やアトミック型を適切に利用する。
* スレッド間で共有するデータを完全に把握する。
* 配列処理可能なハードウェアか、待機処理が長いかなどを考慮し、ミューテックス、クリティカルセクション、スピンロック、リードライトロックを使い分ける。
* 【解決策】過剰なvolatile型修飾子、アトミック型
* volatile型修飾子、アトミック型は、必要最小限に留める。
* 処理が長い場合、volatile型修飾子やアトミック型の変数をそのまま演算せず、一旦ローカル変数にコピーして処理し、最後に書き戻す。これにより、最適化の恩恵を受けつつ、不整合をなくす。

## メモリ不足

スレッドや同期オブジェクトが多いと、それだけメモリも多く消費し、あまりに大すぎるとメモリ不足も引き起こす。

特にスレッドは一つ当たり数KB～数MBのメモリをスタック領域として使用するため、注意が必要。

※ 【正確性に欠ける情報】スレッドのスタック領域のサイズは、必要に応じて自動拡張される仕組みのものもある。これは恐らく1MBなどの割と大きな単位で行われるものと思われる。仮想メモリアドレスの仕組みを利用すると動的なメモリ拡張が可能になるが、それは割と大きなさいずでしかできないはず。ゲームプログラミングで使用するスタックは、全て明示的にサイズを指定した方が無難。4KBもあれば十分な処理も多い。

スレッド生成時の指定により、スレッドに応じてスタック領域のサイズが異なるため、大きなローカル変数を使用する関数を呼び出すとハングする可能性がある。

開発プロジェクト内で、「非スレッドセーフな共通関数」を明確にしておくことが重要。

## ゾンビスレッド（リソースの枯渇）

スレッドは適切な終了処理が必要である。それを行わないと、終了したはずのスレッドが残り続けてリソースの枯渇を招く。

【スレッドの終了を待つ場合】

Windows \_beginthread()もしくは\_beginthreadex()から返されたハンドルを使用して、 WaitForSingleObject() もしくは WaitForMultipleObjects() 関数で待ち、その後、CloseHandle() でハンドルをクローズする。CreateThread() でスレッドを生成した場合も同様。

POSIX pthread\_create() から返されたスレッド識別子を使用して、pthread\_join() で待機する。

C++11 std::threadクラスのインスタンス生成でスレッドを生成し、.join() メソッドで待機する。

【スレッドの終了を待たない場合】

Windows スレッド側の処理で、処理終了時に \_endthread() もしくは、\_endthreadex() を実行する。スレッドを生成した側ではCloseHandle() しない。なお、この場合も WaoitFor\*\*\*() で待機可能。

POSIX pthread\_create() から返されたスレッド識別子を使用して、pthread\_detach() で待機（join）しないことを表明する。

C++11 std::threadクラスのインスタンス生成でスレッドを生成し、.detach() メソッドで待機（join）しないことを表明する。

# 様々なデータ共有手法

スレッド間でのデータ共有の手法について説明する。

## volatile型修飾子、アトミック型の使いどころ

ここまでに何度か説明してきたように、スレッド間で情報の共有を確実にするには、 volatile 型修飾子付き変数やアトミック型を使うのが確実である。

しかし、これによってコンパイラの最適化やCPUの最適化の恩恵を受けることができず、パフォーマンスが劣化する。そのため、できる限り使わないようにした方が良い。

では、何をもってその使いどころとするのか？以下にその条件をまとめる。

なお、以下に示す内容は、本書でこれまでに各所で説明した内容と重複する。重要な内容であるため、改めてここにまとめ直す。

以下はあくまでもまとめ直しであるため、前述の内容のほうが、詳しい解説となっており、用語の解説も省いている。

### volatile型修飾子の使いどころ

パフォーマンスを犠牲にしてでも安全性を保証したい箇所に用いる。

一つの関数の処理の途中で、別スレッドによる値の変更が期待されるもの。

|  |
| --- |
| volatile int global\_a; //グローバル変数  //関数  void func()  {  …(処理)…  if(global\_x == 0) ←参照①  {  …(処理)…  }  …(処理)…  if(global\_x == 1) ←参照②  {  …(処理)…  global\_x = 2; ←更新①  }  …(処理)…  while(global\_x == 2){} ←参照③  …(処理)…  global\_x ++; ←更新②  …(処理)…  } |

* 「参照①」と「参照②」の間で、他のスレッドがグローバル変数 global\_x を更新する可能性が期待されるなら、global\_s に volatile 型修飾子を付けるべき。
* その可能性がない、もしくは、それを無視して良いのであれば、volatile 型修飾子は不要である。
* 「参照③」については、見た目には一つの処理だが、ループ処理中の変化が期待されるので、このような処理がある場合は、volatile型修飾子を必要とする。
* 「更新①」については volatile 型修飾子の有無に依らず、値が更新されるので気にする必要はない。
* 「更新②」は参照を含んだ更新処理なので、やはり直近の参照処理との間に他のスレッドが値を更新することが期待されるなら、volatile型修飾子が必要。
* 要するに、コンパイラの最適化によって、メモリの再参照が省略される可能性があることが問題なので、関数の途中で変数の値が変化することがない、もしくは、それが問題にならない限りは、volatiile型修飾子は必要ない。

使いどころのポイントは以上が全て。

volatile型修飾子の回避策としては、グローバル変数の値が更新される前の処理と、更新された後の処理で、関数を切り分けるなら、なるべくそうしたほうがよい。その場合、（その一連の処理に関しては）volatile型修飾子は必要ない。

* コードの可読性の観点からも、そのような構造化は望ましい。
* inline関数の場合は最適化される可能性があるので、その限りではない。

### アトミック型の使いどころ

基本的には、ミューテックスなどのロック機構を用いずに、アトミック操作（不可分操作）を保証し、良好なパフォーマンスを得たい場合に使用する。

* 当然、アトミック操作は、値をチェックして交換するだけなどのような、一回の操作で完結するものに限られる。
* 二つ以上の計算を要するような処理のアトミック性を保証するには、ミューテックスなどのロック機構を用いなければならない。

|  |
| --- |
| std::atomic<int> global\_a; //グローバル変数  //関数  void func()  {  int prev = 1;  if(global\_a.compare\_exchange\_weak(prev, 2)) ←比較＆交換処理  {  …(処理)… |

* この「比較＆交換処理」は、「値が1なら2に更新してif文の処理ブロックに入る」という処理。その途中で他のスレッドに割り込まれることがないことを保証する。
* 仮にこの「比較＆交換処理」がアトミック操作を保証しない場合（かつ、この処理が複数のスレッドで同時に稼働している場合）、複数のスレッドが同時にif文のブロックに入り込む可能性がある。アトミック操作を保証している場合、そのようなことは起こらない。
* このような処理は、ロックを用いずにスレッド間の同期を保証するため、「ロックフリー」な処理と呼ばれ、マルチスレッド環境で良好なパフォーマンスを得ることができる。

別スレッドによる変数の更新を監視し、同時に更新された他の変数の値も間違いなく参照したい場合。（ミューテックスなどを用いず、ロックフリーな処理として実現したい場合）

|  |
| --- |
| std::atomic<bool> global\_a; //グローバル変数  int global\_b; //グローバル変数  //スレッド①  …(処理)…  global\_b = 123;  global\_a = true;  …(処理)…  //スレッド②  …(処理)…  if(global\_a.load() == true) ←アトミック型参照  {  int b = global\_b; ←アトミック型の前に更新された値の参照  …(処理)… |

* global\_a がアトミック型でなかった場合、CPUの「アウト・オブ・オーダー実行」という最適化により、global\_a が true であっても、global\_b がプログラムの順序通り、123 に更新されているとは限らない、という現象が起こり得る。
* この問題は、volatile型修飾子や関数の切り分けによって回避可能な問題とは全く別であり、CPU内の最適化によって起こる問題である。
* アトミック型は、CPUに対するメモリバリアを保証し、メモリ操作の順序性を保証する。
* この例の場合、global\_aは一連の更新処理の最後に更新され、その変化を監視する処理となっているため、global\_aに対するメモリバリアが指定されれば、その前の更新が保証される。
* 確実にこの問題を回避したい場合は、ミューテックスなどのロック機構を使用して変化を検出することである。ただし、それではパフォーマンスを損ねるため、ロックフリーを実現して高速な処理を行うために、厳密なメモリバリアをコントロールする。

## スレッドローカルストレージ（TLS）

スレットローカルストレージ（TLS）は、メモリ共有とは逆に、スレッド専用のデータを扱うために用いる。

### スレッドローカルストレージの使い方

まずはサンプルを示す。（C++11のサンプル）

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <thread>  #define thread\_local \_\_declspec(thread)//C++11のTLS修飾子Visual C++2013でまだ使えないので偽装する  static volatile int s\_data = 0;//通常変数  static thread\_local int s\_tlsData = 0;//TLS変数  //スレッド関数  void threadFunc(const char\* name)  {  ++s\_data;  ++s\_tlsData;  printf("[%s] s\_data=%d, s\_tlsData=%d\n", name, s\_data, s\_tlsData);  }  //C++11 thread テスト  int main(const int argc, const char\* argv[])  {  std::thread th1(threadFunc, "スレッド1");  std::thread th2(threadFunc, "スレッド2");  std::thread th3(threadFunc, "スレッド3");  th1.join();  th2.join();  th3.join();  return EXIT\_SUCCESS;  } |

↓（処理結果）

|  |
| --- |
| [スレッド1] s\_data=1, s\_tlsData=1  [スレッド2] s\_data=2, s\_tlsData=1  [スレッド3] s\_data=3, s\_tlsData=1 |

普通のグローバル変数である s\_data は、その更新が他のスレッドにも反映されているのに対して、s\_tlsData は、スレッドごとに個別の値を保持していることがわかる。

このように、スレッドローカルストレージは、一件同じグローバル／静的変数ではあるものの、スレッド固有の値を扱うものとなる。

### スレッドローカルストレージの使いどころ

例えば、標準ライブラリの関数には、関数を実行した結果失敗した場合、詳しいエラー内容をグローバル変数 errno　に格納するものがある。errno が完全なグローバル変数だと、他のスレッドの処理結果が影響してしまい、変数の値を信頼することができない。errno がスレッド毎に値を保証するなら、他のスレッドの動作状況を気にせず、errno に基づく処理を行うことができる。

このように、他のスレッドの処理が干渉しては困るが、どのスレッドからも実行可能な共通処理に、スレッドローカルストレージを使用することができる。

### スレッドローカルストレージの注意点①

スレッドの数だけ使用メモリが増大する点に注意が必要。

【補足】 TLSのメモリがどこから取られるかは調査しきれなかった。各スレッドのスタック領域の上部から取られるような挙動を期待したが、WindowsのTLSAlloc() というTLS領域を手動で確保する関数の説明を確認する限り、（Windowsに関しては）「TLS記憶領域」というプロセス内の共通領域からスレッドに随時割り当てられるようである。（余談だが、あるゲーム機の処理では、ローダブルモジュール（DLL）を読み込んだ際のプログラム領域として、スタックの上部の領域が使用される仕組みとなっていた…と記憶している。）

WindowsのTLS領域は、LPVOID型のデータ（つまり32bit値or 64bit値のデータ）が、最低でもTLS\_MINIMUM\_AVAILABLE個（64個）の配列（？）として構成される。配列要素の一つ一つを「スロット」と呼び、スレッドが作られるたびに空いているスロットをスレッドに割り当ててスレッド固有の変数として扱う。TLS\_MINIMUM\_AVAILABLE個のスロットを超えたら、（おそらく）グローバルヒープ（もしくはメモリページ）などからメモリを確保してTLS領域を拡張する。

ゲームプログラミングでTLSを使用する場合、グローバルヒープを極限まで削ってゲーム用に予約する可能性があるので、SDKのマニュアルをよく確認して挙動を確認したほうがよい。

### スレッドローカルストレージの注意点②

TLS領域にアクセスする処理はひと手間かかるので、通常のグローバル変数に比べてアクセスが遅い点にも注意が必要である。

他のスレッドが干渉することがないため、volatile型修飾子やアトミック型は不要だが、一旦ローカル変数に値をコピーして、演算後に書き戻すようにしたほうが良好なパフォーマンスが期待できる。

以下、通常変数とTLSでどれだけ処理に差が出るかを、逆アセンブルコードで示す。（あえてDebugビルドのコードで示す。最適化されていない。）

元の処理コード：

|  |
| --- |
| void threadFunc(const char\* name)  {  ++s\_data;//通常変数  ++s\_tlsData;//TLS変数 |

逆アセンブルコード：通常変数の処理

|  |
| --- |
| 14: ++s\_data;//通常変数  001A34BE A1 C8 C2 1A 00 mov eax,dword ptr ds:[001AC2C8h] ←メモリから変数の値を読み出し  001A34C3 83 C0 01 add eax,1 ←インクリメント  001A34C6 A3 C8 C2 1A 00 mov dword ptr ds:[001AC2C8h],eax ←メモリに値を書き戻し |

逆アセンブルコード：TLS変数の処理

|  |
| --- |
| 15: ++s\_tlsData;//TLS変数  001A34CB A1 D4 C2 1A 00 mov eax,dword ptr ds:[001AC2D4h] ←TLS領域のインデックスを取得（？）  001A34D0 64 8B 0D 2C 00 00 00 mov ecx,dword ptr fs:[2Ch] ←現在のスレッド用の  TLS領域のオフセット取得（？）  001A34D7 8B 14 81 mov edx,dword ptr [ecx+eax\*4] ←実際のTLS領域のアドレスを取得（？）  001A34DA 8B 82 04 01 00 00 mov eax,dword ptr [edx+104h] ←TLS領域から値を読み出し  001A34E0 83 C0 01 add eax,1 ←インクリメント  001A34E3 8B 0D D4 C2 1A 00 mov ecx,dword ptr ds:[1AC2D4h] ←（上記と同じTLS領域アドレス計算）  001A34E9 64 8B 15 2C 00 00 00 mov edx,dword ptr fs:[2Ch] ←（同上）  001A34F0 8B 0C 8A mov ecx,dword ptr [edx+ecx\*4] ←（同上）  001A34F3 89 81 04 01 00 00 mov dword ptr [ecx+104h],eax ←メモリに値を書き戻し |

C++言語の処理コードは同じ一行でも、実際の処理コードにはかなりの差があることがわかる。

TLSの使いどころは厳選すべきである。

## プロセス間のデータ共有①：共有メモリ

スレッドどうしは直接メモリ空間を共有できるため、以上のような変数ベースのデータ共有で十分である。

プロセス間でデータ共有する場合、普通はメモリ空間を共有できないので、「共有メモリ」という特殊なメモリ操作を行う。

プロセス間通信は、ゲームプログラミングで必要となることはそうそうなさそうだが、デバッグツールなどの使い道も考えられるので、簡単な技術紹介だけしておく。

### SystemV共有メモリ

Unixで一般的な共有メモリの手法。

SystemVセマフォと同様にOSが管理し、使用状況をipcsコマンドで確認することができる。

SystemVセマフォと同様に、特定のキー（ユニークID）を指定してメモリ空間を割り当てることにより、複数のプロセスで同じメモリへの読み書きを可能とする。

### メモリマップトファイル

Windowsで一般的な共有メモリの手法。

本来は、ファイルをメモリのアドレス空間にマッピングして、ファイル操作系のAPIを使わずにファイルにアクセスする手法。メモリ（ポインタ変数）への直接アクセスがファイルの読み書きとなる仕組み。

ファイルを指定せずにメモリマップトファイルを使用すると、ファイルがないまま、メモリ空間へのアクセスが可能となる。

名前付きミューテックスなどと同様に、名前を付けて管理することができるため、他のプロセスが同じ名前でメモリマップトファイルをオープンすると、メモリ空間を共有できる。

## プロセス間のデータ共有②：プロセス間通信

プロセス間は、直接データを送り合うことでデータを共有する方法がある。

これも簡単な技術紹介だけしておく。

### シグナル

Unixでもっとも一般的なプロセス間通信の手法。

先に説明しているとおり、OSによるプロセス制御に用いられる。

### ウインドウメッセージ

Windowsでもっとも手軽なプロセス間通信の手法。

Unixのシグナルと同様に、OSによるプロセス制御に用いられるが、割り込み方式ではなく、キューイングされたメッセージを逐一処理する。

マウスクリックやキー入力、アプリケーションのアクティブ化といった非常に多数オンイベントがメッセージとして扱われる。任意のメッセージや不定長のデータを送信することができる。

通信相手のプロセスのウインドウハンドルさえわかれば、どんなプロセス（ウインドウ）に対しても、メッセージを送ることが可能。

### パイプ

UnixでもWindows（MS-DOS）でも標準的なプロセス間通信の手法。

基本的には、親プロセスと子プロセスの二つのプロセス間で通信するための仕組み。

親プロセスの標準出力が子プロセスの標準入力に送られる。基本的に親プロセスから子プロセスへのシーケンシャルなデータ通信である。

Windowsでは、親プロセスが子プロセスを生成する際に、パイプによる連結を指定することができる。

このプロセス間通信はコマンドプロンプト上でも簡単に扱うことができる。

以下、Unix系のコマンドを使ったパイプによるプロセス間通信の例を示す。

パイプを使う前：（catコマンドの実行）

|  |
| --- |
| $ cat test.txt  すずき ←ファイルの内容が、cat コマンドの標準出力（画面）に出力。  いしい  さとう  たなか  やまもと  なかむら  こばやし  いとう  やまだ  よしだ |

パイプを使用：（catコマンドとsortコマンドを連結）

|  |
| --- |
| $ cat test.txt | sort ←「|」がパイプ  いしい ←パイプ「|」により、  いとう cat コマンドがファイルの内容を標準出力に出力すると、  こばやし それがsort コマンドの標準入力に送られる。  さとう sort コマンドは標準入力を処理して、その結果を  すずき 標準出力（画面）に出力。  たなか  なかむら  やまだ  やまもと  よしだ |

### 名前付きパイプ（Uninx）

Unixの名前付きパイプ。Windowsの同名技術とは全く別もの。

パイプ「|」の代わりに mkfifo コマンドで作成したファイル（例えば /tmp/test\_pipe といったファイルを作る）にリダイレクトすることで、パイプと同様のシーケンシャルな通信を行う。

名前付きパイプを使用：（catコマンドとsortコマンドを連結）

|  |
| --- |
| $ mkfifo /tmp/test\_pipe ←名前付きパイプを作成  $ cat test.txt > /tmp/test\_pipe & ←「>」が標準出力へのリダイレクト 「&」はプロセスの終了を待たずに  バックグラウンド実行させる指定  $ sort < /tmp/test\_pipe ←「<」が標準入力へのリダイレクト  いしい  いとう  こばやし  さとう  すずき  たなか  なかむら  やまだ  やまもと  よしだ  $ rm /tmp/test\_pipe ←名前付きパイプを削除 |

## プロセス間のデータ共有③：ネットワーク通信

ネットワーク通信もまたプロセス間通信の手法となりえる。

これも簡単な技術紹介だけしておく。

### 名前付きパイプ（Windows）

Windows用のプロセス間通信およびネットワーク通信の手法。

「パイプ」と名がつくものの、パイプとは別物であり、Unixの名前付きパイプとも全く異なる。

性質としては、ソケット通信に近い。

「\\ . \（パイプ名）」といったファイルパスのような形式で名前を付けて、パイプと同様のシーケンシャルなデータ通信を行う。パイプと異なり、双方向通信が可能である。

「\\（コンピュータ名）\（パイプ名）」というUNC（Universal Naming Convention = ネットワークパス）を用いることで、ネットワーク通信を介したデータ通信が可能。

通信プロトコルを完全に隠ぺいし、かつ、プロセス間通信もネットワーク通信も全く同じ手続きで行える、きわめて抽象度の高い通信手法。

### ソケット通信

ソケット通信。

TCP/IP環境で一般的なネットワーク通信手段。

ローカルアドレス（127.0.0.1）を使ってプロセス間通信に使用しても問題ない。

### メッセージキューサービス

MSMQなどのメッセージキューサービスアプリケーション（バックグラウンドプロセス）を介して、ネットワーク越しのプロセス間通信を行う手法。

メッセージキューアプリケーションがメッセージの永続化（ハードディスクへの保存）を行うので、PCを再起動しても未処理のメッセージが失われず、過去の通信状態も確認できる。

大掛かりな仕組みのため、あまり気軽に用いられるものではない。また、あまりに頻繁なメッセージには対応しきれない。需要なメッセージを確実に送受信しなければならないクリティカルな業務に使用する。

# 並列処理の活用

最後に、大規模な並列処理を紹介する。

近年「ビッグデータ」という言葉がはやっているように、各所に膨大なデータがある。

この膨大なデータを効率よく処理するには並列処理が欠かせない。

ゲーム制作でもまったく無縁の話ではないため、参考までに簡単な技術的説明を記載しておく。

## MapReduce

MapReduce（マップ・リデュース）は、Googleによって導入された、分散コンピューティングのプログラミングモデルである。

単純なイメージとして、「Mapフェーズ」で大量のノード（PCもしくは演算装置）に分散し、「Reduceフェーズ」で段階的にデータを集約して結果を得る手法である。

MapReduceの大雑把なイメージ：



※かなり大雑把なイメージであり、実際の処理モデルのイメージとしては適切なものではない。

もう少し具体的な処理のイメージを説明する。

同様の処理モデルを適用したサンプルプログラムを、本書中の「C++11版非同期関数」で示しており、その中のソート処理に適用している。

STLの<algorithm>が実装する std::sort() 関数はクイックソートである。クイックソートの処理時間は　平均＝O(n log n)、最悪=O(n2)　と言われていることから、件数が多ければ多いほど、処理時間は指数関数的に増大する。

この計算を高速化するために、「マップフェーズ」として、データを4分割してそれぞれクイックソートした後、次に「リデュースフェーズ」としてマージソート（std::inplace\_merge）を使用して2段階に分けて集約している。

件数が多いため、このような回りくどい処理が効果を出していることが確認できる。

このような処理モデルは、大量の演算装置を持つGPGPUを効果的に活用する上でも効果的なのではないかと考える。

### 耐障害性／スケールアウト

余談になるが、MapReduceには、高速化以外の重要な考え方がある。

それは、「耐障害性」である。

一部の処理ノード（コンピュータ）が障害（故障や電源切断など）によって計算結果を出せなかった場合、別のノードに再スケジューリングすることで処理を完了させることができる。

また、単純にノードを増やせば性能を向上させることができる構造のため、簡単に「スケールアウト」することができる。

「スケールアウト」とは、コンピュータ（主にサーバー）を増やす事で性能を向上させることを意味する用語である。

「スケールアウト」するには、あらかじめそのように設計されたシステムでなければならず、そうでないものは性能向上のために「スケールアップ」を行う事になる。

「スケールアップ」とは、メモリの増強や高性能なコンピュータへの入れ替えで性能を向上させることを意味する用語である。

■■以上■■

索引

Ａ

APU 13, 19

ATI Stream 66

Ｂ

BIOS 9

Ｃ

C# 61

CallOnce 182

C++11版 184

POSIXスレッドライブラリ版 182

CAS 169

Cell 19

PPE 12, 19

SPE 19

Concurrent 5

CPU 12

CISC 13

RISC 13

ソケット 15

CUDA 66

Ｄ

DirectCompute 66

DMAC 19

Ｆ

fork 33

fork-joinモデル 33

FPU 16

Ｇ

GCC

--fstack-usageオプション 30

GPGPU 65

GPU 17

GPUメモリ 19

Ｈ

Hadoop 67

HT 15

Ｉ

ipcsコマンド 120, 207

ISR 69

Ｌ

Life with Playstation 68

Lua 61

Ｍ

MapReduce 61, 210

MAPファイル 26

MIMD 16

MISD 16

MMX 16

MPU 12

Ｏ

objdumpコマンド 25

OpenCL 66

OpenMP 52

ORB 67

OS 5

Ｐ

Parallel 5

Ｒ

RAM 17

ROM 17

RPC 67

Ｓ

SIMD 16

SISD 16

sizeコマンド 25

SPU 20

LS 20

SPURS 65

Squirrel 61

SSE 16

Ｔ

TLS 204

Ｖ

volatile型修飾子 76, 103, 202

VRAM 19

VU 16

Ｙ

Yield 82

あ

アウト・オブ・オーダー実行 77

アセンブラ 27

アトミック型 79, 203

アトミック操作 73, 109

C++11版 111, 113

アプリケーション 8

い

イベント 157

WIN32API版 157

WIN32API版名前付きイベント 163

イベントコールバック 70

イン・オーダー実行 78

インターロック操作 109

WIN32API版 109

インラインアセンブラ 106

う

ウインドウメッセージ 207

ウェイトフリー 79

え

演算装置 13

か

カーネル時間 9

カーネル 5

ハイブリッドカーネル 7

マイクロカーネル 7

モノリシッックカーネル 6

き

機械語 27

キャッシュメモリ 18

共有メモリ 206

く

クライアント・サーバー 66

クラウド 67

ぐりっど 67

グリッド・コンピューティング 20

クリティカルセクション 100

WIN32API版 101

こ

コア 13

コプロセッサ 16

コルーチン 61

コンテキスト 31

コンテキストスイッチ 9, 30, 69

コンペア・アンド・スワップ 169

C++11版アトミック操作 174

WIN32API版インターロック操作 169

さ

サーバー 20

サービス 12

再帰ロック 197

先物 185

C++11版 185

サスペンド 61

サンプルプログラム 1

し

シグナル 186, 207

POSIX版 187

シグナル

同期シグナル 187

システムコール 8, 68, 71

条件変数 138

C++11版 144

POSIXスレッドライブラリ版 139

シングルコア 13

シングルプロセッサ 13

す

スケールアウト 212

スケールアップ 212

スタックオーバーフロー 29

スタックフレーム 29

スタックポインタ 29

スタック領域 28

スピンロック 98

POSIXライブラリ版 98

スリープ 81, 82

スレッド 2, 11

C++11版 47

POSIXスレッドライブラリ版 37

WIN32API版 42

スケジューリング 31

優先度 31

スレッドローカルストレージ 204

せ

セマフォ 120

POSIXスレッドライブラリ版 124

POSIX版名前付きセマフォ 127

SystemV版 120

WIN32API版 131

WIN32API版名前付きセマフォ 134

そ

ソケット通信 209

た

耐障害性 212

タイムスライス 4

クォンタム 31

て

デーモン 12

デッドロック 194

デバイスドライバ 6, 8

と

同期 73

動作モード 8

カーネルモード 8

ユーザーモード 8

な

名前付きパイプ(Unix) 208

名前付きパイプ(Windows) 209

に

ニーモニック 27

ね

ネットワーク通信 209

は

バイオス 9

排他制御 84

ハイパースレッディング・テクノロジー 15

パイプ 208

バリア 150

POSIXスレッドライブラリ版 150

ひ

ビジーウェイト 81

非同期関数

C++11版 57

ふ

ファイバースレッド 61

不可分操作 73

浮動所数点演算装置 16

プログラム 27

プログラムカウンタ 27

プロセス 10

子プロセス 10

バックエンド 12

フロントエンド 12

マルチプロセス 20

プロセス間通信 207

プロセッサ 13

へ

平行 5

並列 5

ま

マシン語 27

マルチコア 14

ヘテロジニアスマルチコア 15

ホモジニアスマルチコア 15

マルチタスク 3

ノンプリエンプティブ 3, 61

プリエンプティブ 4

マルチプロセッサ 14

み

ミューテックス 87

C++11版 95

POSIXスレッドライブラリ版 87

WIN32API版 90

WIN32API版名前付きミューテックス 92

め

メインメモリ 17

メッセージキューサービス 210

メニーコア時代 22

メモリオーダー 179

メモリ空間 11

メモリ構造 23

メモリ操作命令 78

ストア 78

メモリバリア 78

メモリフェンス 78

ロード 78

メモリマップトファイル 207

も

モニター 138

ゆ

ユニファイドメモリ 19

ら

ラムダ式 57

り

リード・ライトロック 114

POSIXスレッド版 114

リエントラント 193

れ

レジスタ 18

ろ

ローカル変数 28

ロック 72, 84

ロックフリー 78

論理プロセッサ 14

わ

割り込み 68

SWI 70

V-SYNC割り込み 68

シグナル 71

垂直同期割り込み 68

ソフトウェア割り込み 70

タイマー割り込み 71

ハードウェア割り込み 70

割り込みサブルーチン 69

割り込みハンドラ 69

マルチスレッドプログラミングの基礎

以　上