マルチスレッドプログラミングの基礎

– スレッド制御のための基礎 –

2014 年 1 月 20 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 1 月 20 日	板垣 衛	（初版）

■ 目次

■ 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を解説する。

かなり広範囲な解説を記述しているのは、そのところどころにスレッドの理解につながる要素があるためである。周辺知識を一まとめに解説する。

■ 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、最適なマルチスレッドプログラミングを行えるようにすることを目的とする。

■ スレッドの仕組み

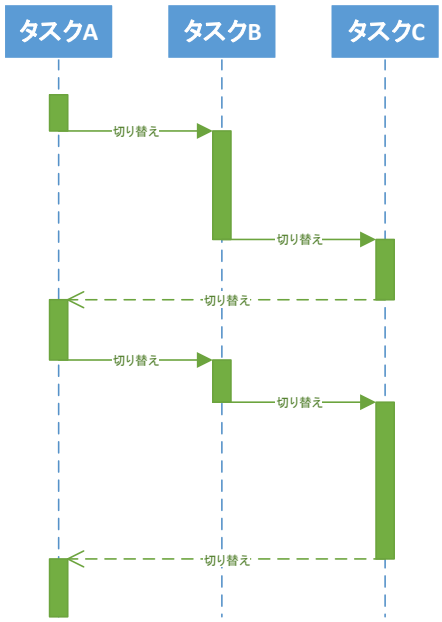
▼ マルチタスク

「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。



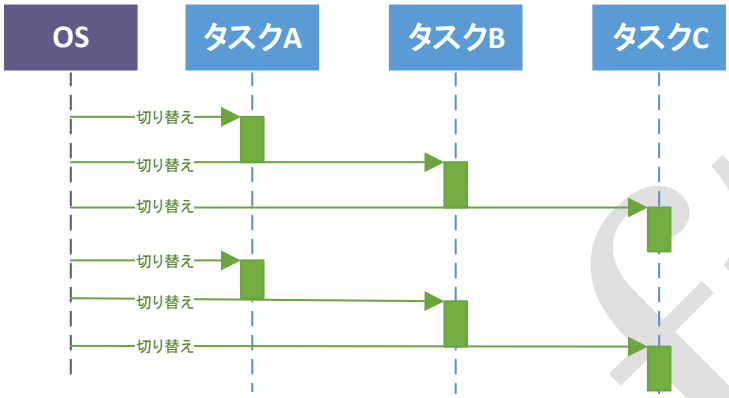
Windows95 より前の Windows3.1 はこの方式。例えば「`while(1){}`」と書いたプログラムを実行すると、同時に起動している「メモ帳」や「電卓」などのアプリケーションも動作できずに止まってしまう。

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。

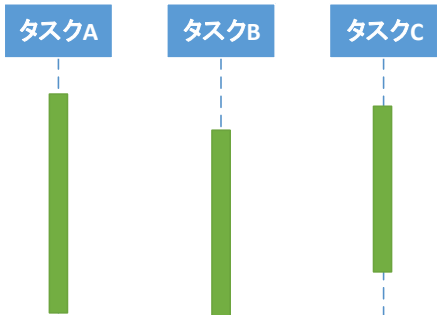
● タイムスライス

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。



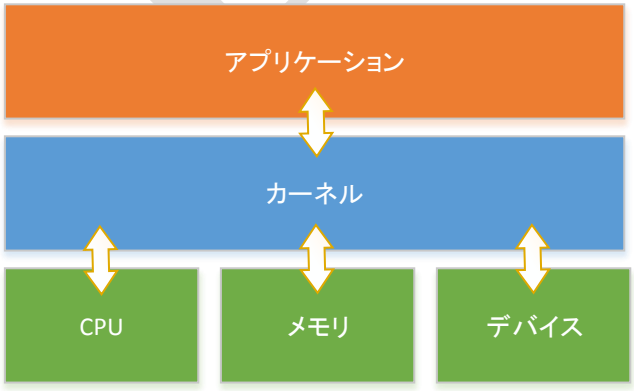
なお、厳密な用語の定義としては、「平行」は「並列」を包含する。「複数のタスク」を「複数の演算装置」が実行することもまた「平行」である。

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

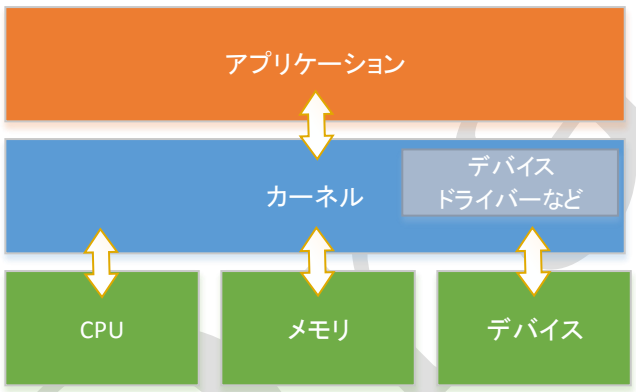
してデバイスにアクセスし、OS の制御のもとで CPU の割り当てやメモリの確保が行われる。

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「デバイスドライバー」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。



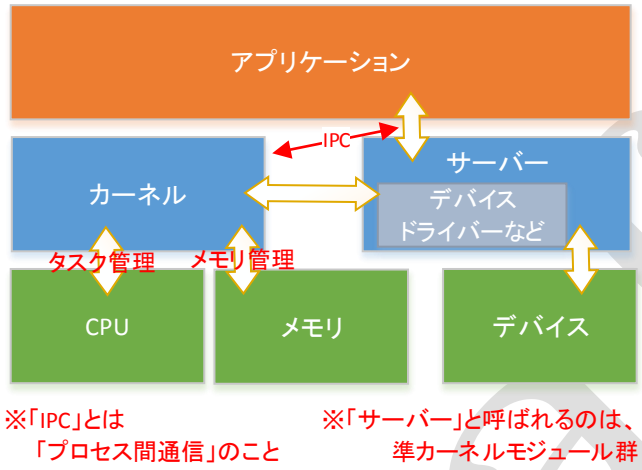
単純には、デバイスドライバーがカーネルに一体化しているような状態。そのため、多くの機能を取り込んだカーネルは肥大化する。また、デバイスドライバーに深刻な問題があると、OS 全体が停止するような事態も招く。

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、デバイスドライバーを 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、デバイスドライバーなどの他の「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。

マルチスレッド制御の観点で言えば、より高度な機能が提供されていると言える。例えば、後述する「名前付きパイプ」では、高度な通信の抽象化が実現できている。「名前付きパイプ」は、OS に対してユーザーがインストールした通信プロトコル（TCP/IP や NetBUEI、IPX/SPX など）を意識することなく、全く同じ手続きでプロセス間通信もネットワーク通信も可能な通信手段である。

● 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。

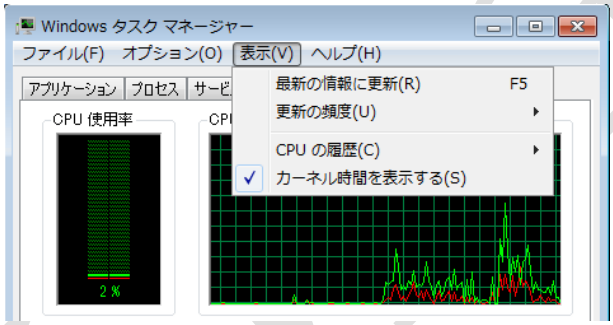
● カーネルモードとユーザーモード

マルチスレッドプログラミングでカーネルの方式を意識する必要性はあまりないが、OS が使用するメモリ量や CPU は意識したいところ。PS3 の開発では、システムバージョンアップの際に所用メモリ量が削減されることを切実に願うことがあった。

また、パフォーマンスチューニングの際、ユーザーモード側（ゲーム側）の処理負荷が高いのか、カーネルモード側の処理負荷が高いのかは、可能な限り意識したい。

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。全体の CPU 使用率のうち、カーネルが占めている割合を示す。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、複数のスレッドを並行で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できない。

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保護される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、ゲームプログラミングでもかなり利用価値が高い。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、長時間のゲームプレイではやがてメモリ不足などの問題を引き起こすので、きちんと明示的に全ての解放を行うべき。

【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。

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スリープして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ロードの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。

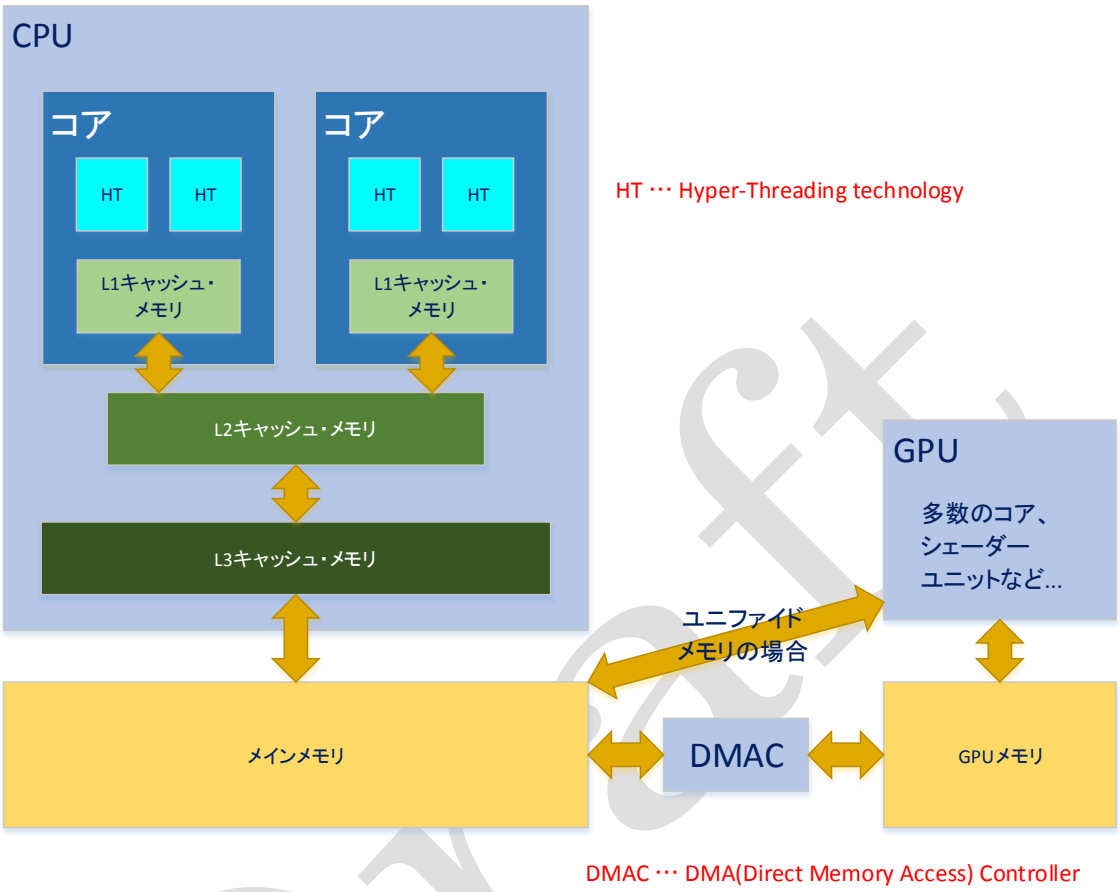
▼ シングルプロセッサ／シングルコア

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。

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。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

一つの演算装置上で二つの処理をスケジューリングすることで二つ以上の「論理プロセッサ」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。

元々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。

▼ 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。

ゲームプロでは、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）。

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であるが、C 言語などのように自由な変数が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だり書き戻したりすることで処理する。（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。

▼ 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。「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。

「Cell コンピューティング」もこのような構想のもと、世界中の PS3 を使ってタンパク質解析などの演算をする「Life with Playstation」を展開していた。

■ 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：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。

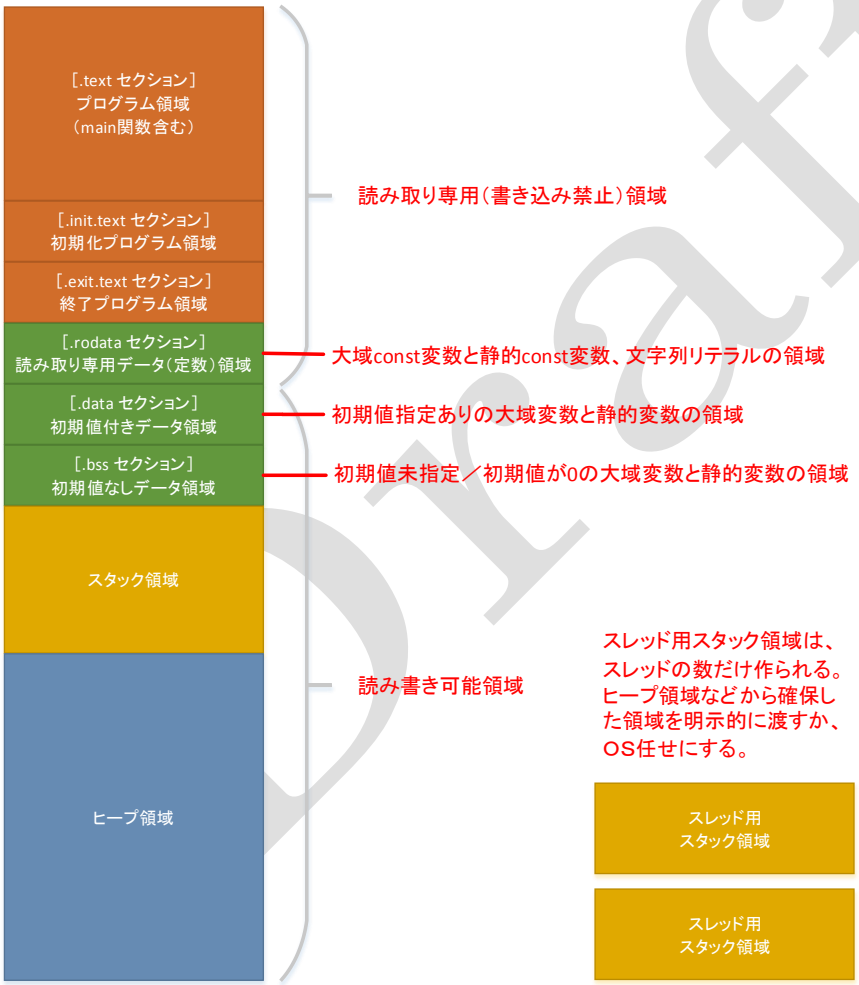
マルチスレッドは、スレッド間の同期や共有リソース（メモリなど）の扱いにとても神経質にならなければならない。同期やリソースアクセスには厳格なルールを設けて扱う。

■ 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。

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。

PC … プログラムカウンタ（プログラムの実行アドレス） ※関数から return する際、スタックから取り出して、前の位置の次の位置に戻す

SP … スタックポインタ（現在使用中のスタック領域のトップアドレス） ※関数から return する際、関数の呼び出し前の位置に戻す

BP … スタックのベースアドレス（関数ごとのスタック領域の基底のアドレス） ※関数から return する際、スタックから取り出して前の位置に戻す

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ーの情報を退避・復元する。レジスターの中にはプログラムカウンタとスタックポインターも含まれる。

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。

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。

■ 様々なスレッド

▼ fork（Unix 系）

まずは fork（フォーク）。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。

関数をスレッド化する通常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スレッドに比べて所用メモリ量が大きく、メモリ空間の共有もできない。

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    5348   12044 pty0     07:48:54 /cygdrive/c/Work/test/Program/C++/Thread/fork/fork
    user    1760    6752 pty0     07:48:56 /cygdrive/c/Work/test/Program/C++/Thread/fork/fork
    user   11556    5348 pty0     07:48:56 /cygdrive/c/Work/test/Program/C++/Thread/fork/fork
    user   12152    6464 pty0     07:48:57 /cygdrive/c/Work/test/Program/C++/Thread/fork/fork
    user    8972    5348 pty0     07:48:58 /cygdrive/c/Work/test/Program/C++/Thread/fork/fork
    user    6752    6464 pty0     07:48:55 /cygdrive/c/Work/test/Program/C++/Thread/fork/fork
    user    6464    5348 pty0     07:48:54 /cygdrive/c/Work/test/Program/C++/Thread/fork/fork
    user   10372   11556 pty0     07:48:57 /cygdrive/c/Work/test/Program/C++/Thread/fork/fork
```

▼ Posix スレッド（Unix 系）

Posix スレッドは、Unix 系 OS で標準的なスレッドライブラリ。

fork 版と同じ挙動のサンプル。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。

malloc() で獲得したメモリは、子スレッドに受け渡すとすぐに不要になるが、元のスレッドで free() しないとハングする。

ついでに TLS（スレッドローカルストレージ）の動作もテスト。C++11 仕様の指定方法はエラー（GCC 4.8）。

Posix 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;//CC 版 TLS 指定
//thread_local pthread_t tls_pth = 0;//C++11 仕様版

//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    8140   11912 pty0      11:38:07 /cygdrive/e/Work/test/Program/C++/Thread/pthread/pthread
```

▼ Win32 スレッド（Windows 系）

Win32API によるスレッド。

fork 版、Posix スレッドライブラリ版と同じ挙動のサンプル。スレッド生成関数、終了待ち関数、Sleep()関数の扱いが Posix スレッド版と異なることに注目。

Windows 版の TLS（スレッドローカルストレージ）の動作もテスト。C++11 仕様の指定方法はエラー（Visual C++ 2013）。

このサンプルの後に、Windows 版で用意されている二種類のスレッド生成用関数の違いを説明する。

Windows 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 <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
//thread_local unsigned int 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
	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);
```

```
	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);

			//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
※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

```
- Start Thread Test -
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 -
```

▼ OpenMP

▼ ファイバースレッド

▼ SPU／GPGPU／CUDA／ATI Stream／OpenCL

▼ クライアント・サーバー／クラウド／グリッド／RPC／ORB

▼ 割り込み／システムコール

■ マルチスレッドで起こり得る問題①

▼ 不完全な不可分操作によるデータ破損

▼ スレッド間の情報共有の失敗

■ スレッドの同期

▼ ビジーウェイト

▼ スリープウェイト

▼ スリープ／Yield

■ 様々な同期手法

▼ Volatile 型変数

▼ スピンロック

▼ ミューテックス

▼ 軽量ミューテックス

▼ 名前付きミューテックス

▼ クリティカルセクション

▼ インターロック操作

▼ セマフォ／名前付きセマフォ

▼ モニター

- 条件変数

- イベント／名前付きイベント

▼ シグナル

■ マルチスレッドで起こり得る問題②

▼ デッドロック

- **相互ロック**

- **自己ロック**

▼ 低速化

▼ モニターのタイミングずれ

▼ ゾンビスレッド

■ 様々なデータ共有手法

▼ グローバル変数／スタティック変数

▼ スレッドローカルストレージ（TLS）

▼ 共有メモリ

▼ メモリマップトファイル

▼ メッセージ

▼ メッセージキュー

▼ パイプ

▼ 名前付きパイプ

▼ ソケット

■ その他のスレッド制御

▼ スレッド優先度

▼ イールド（Yield）

▼ Map Reduce 理論

■■以上■■

■ 索引

索引項目が見つかりません。

マルチスレッドプログラミングの基礎

以　上