「サービス」によるマルチスレッドの効率化

– 透過的な通信サービスとジョブスケジューラ –

2014 年 2 月 18 日　初稿

板垣 衛

■ 改訂履歴

稿	改訂日	改訂者	改訂内容
初稿	2014 年 2 月 18 日	板垣 衛	（初稿）

■ 目次

■ 概略

常駐スレッドを「サービス」に見立てて活用する手法について説明する。
とりわけスレッドを透過的な通信サービスに見立てて処理を簡単にする手法と、ハードウェアに合わせた適切な並行処理スケジューリングの手法についてを扱う。

■ 目的

本書は、別紙の「マルチスレッドプログラミングの基礎」を踏まえ、実際にゲーム開発で効果的なマルチスレッドプログラミングを実践することを目的とする。

別紙の「効率化と安全性のためのロック制御」が基本的なマルチスレッド処理の仕組みにフォーカスしていたのに対して、本書はゲームシステムの基本アーキテクチャにマルチスレッドを組み込むことにフォーカスする。

■ 「サービス」とは？

ここでいう「サービス」とは、Windows の「サービス」（あるいは Unix 系の「デーモン」）と同様の意味で、バックグラウンドで独立して動作するシステムのことである。

基本的に、ゲーム起動時から稼働する常駐スレッドで、他のスレッドもしくは外部（ネットワーク）からの要求に応じて処理する。

「サービス」としてイメージしやすいものとしては、「Web サーバー」や「データベースサーバー」などのサーバー系システムがある。外部からの要求があるまでは何もせず待機し、要求があったら稼働して、要求に応じたデータを（非同期で）返す。

このような「サービス」としての機能を持たせた常駐スレッドは、ゲームにおいても十分に利用価値がある。

■ 主な常駐スレッド

「常駐スレッド」＝「サービス」として定義づけるものではない。
通常、ゲームで利用する常駐スレッドとは「フレームに跨がって継続的に処理が必要な

もの」である。

以下、ゲームで利用する主な常駐スレッドを示す。

・ サウンド ..... メインループの状態に左右されず、ストリーミング再生やフェード（音量）制御を行うためにはスレッド化が必須。最優先のプライオリティが設定されることが多い。
・ ファイルシステム ..... 非同期読み込みのために必要。OS の非同期読み込みやミドルウェアに依存するケースも多いので、タイトル固有のファイルシステムスレッドは必須ではない。しかし、読み込み要求のキューイングや任意の圧縮解凍処理を組み込むならスレッド化したほうが効率的である。
・ 描画スレッド ..... GPU による描画処理を並列に行うために、専用スレッドを設ける。
・ 物理演算 ..... 処理が重いので、バックグラウンドで計算し続けるものもある。
・ ネットワーク通信 ..... （意外と）ブロック型の処理も多く、スレッド化が必須。外部のサーバーに要求を出した後のレスポンスの待ち受けや、外部からの接続要求を待ち受けに使用する。
・ 並列処理スケジューラ ..... SPURS などのジョブキューイングシステム。並列処理の要求を受けて、空いている演算装置（SPU や GPU）に処理を投入するためのスケジューリングを行う。

本書は、このうちの「ネットワーク通信」と「並列処理スケジューラ」について扱う。

■ 透過的な通信サービス

一言に「通信サービス」といっても、取り扱う通信処理の「レイヤー」によってその形態が変わるが、ここでは「アプリケーション層」の通信を扱う。

ゲーム（スレッド）側にとっては、通信の作法・過程を気にせず、単純にゲーム固有の通信要求と結果だけを扱えることが重要である。「スレッド」を通信先のサーバーに見立てて、スレッドとのやり取りを透過的なサーバー通信とする。

このような、ゲーム側の処理を「単純化」することを目的に、「サービス」としてのスレッドを確立する。

▼ OSI 参照モデル

「アプリケーション層」について簡単に説明する。

これは、「OSI 参照モデル」の通信層の一つである。

OSI 参照モデルとは、「OSI = Open System Interconnection（開放型システム間相互接続）という「データ通信のためのネットワーク構造の設計方針」に基づいた通信機能のことで、7 層に分割される。

層	名称	役割	主な例
第7層	アプリケーション層	具体的な通信サービス（例えばファイル・メールの転送、遠隔データベースアクセスなど）を提供。HTTPやFTP等の通信サービス。	HTTP・DHCP・SMTP・SNMP・SMB・FTP・Telnet・AFP・X.500
第6層	プレゼンテーション層	データの表現方法（例えばEBCDICコードのテキストファイルをASCIIコードのファイルへ変換する）。	SMTP・SNMP・FTP・Telnet・AFP
第5層	セッション層	通信プログラム間の通信の開始から終了までの手順（接続が途切れた場合、接続の回復を試みる）。	TLS・NetBIOS・NWLink・DSI・ADSP・ZIP・ASP・PAP・名前付きパイプ
第4層	トランスポート層	ネットワークの端から端までの通信管理（エラー訂正、再送制御等）。	TCP・UDP・SCTP・SPX・NBF・RTMP・AURP・NBP・ATP・AEP
第3層	ネットワーク層	ネットワークにおける通信経路の選択（ルーティング）。データ中継。	IP・ARP・RARP・ICMP・IPX・NetBEUI・DDP・AARP
第2層	データリンク層	直接的（隣接的）に接続されている通信機器間の信号の受け渡し。	イーサネット・トークンリング・アークネット・PPP・フレームリレー
第1層	物理層	物理的な接続。コネクタのピンの数、コネクタ形状の規定等。銅線−光ファイバ間の電気信号の変換等。	RS−232・RS−422（EIA−422, TIA−422）・電話線・UTP・ハブ・リピータ・無線・光ケーブル

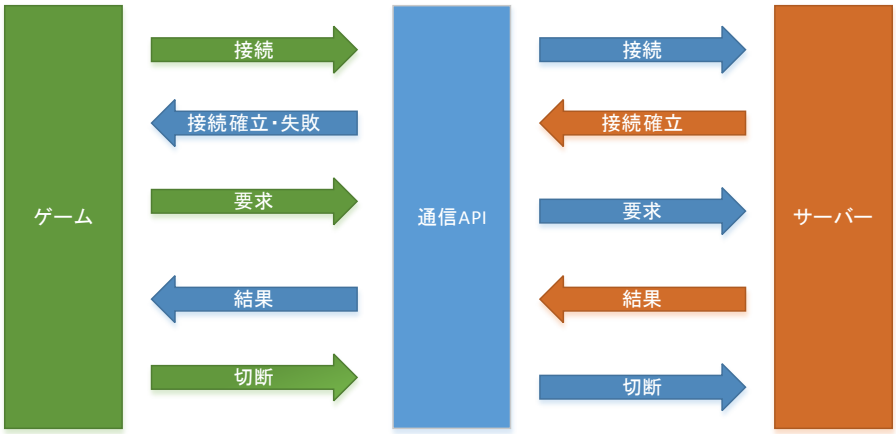
※Wikipedia より引用

なお、「ソケット通信」などの、SDK で提供される通信機能は、主に「セッション層」である。この層の API を使うと、オンラインゲームなどの任意の通信（プロトコル）をプログラミングできる。

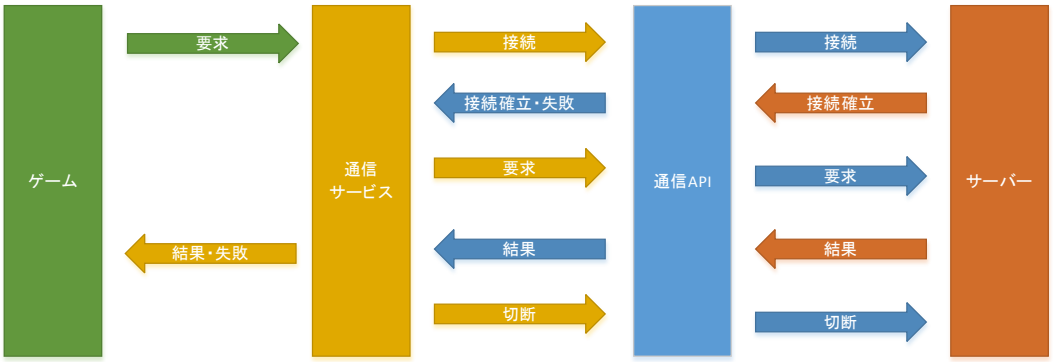
▼「サービス」を用いたプログラミングのイメージ

「サービス」を用いたプログラミングでは、ゲーム側の処理を単純化する。また、通信処理がゲーム側の処理をブロックすることもない。

単純に API に基づいてプログラミングしたイメージ：



「サービス」を使用したイメージ：



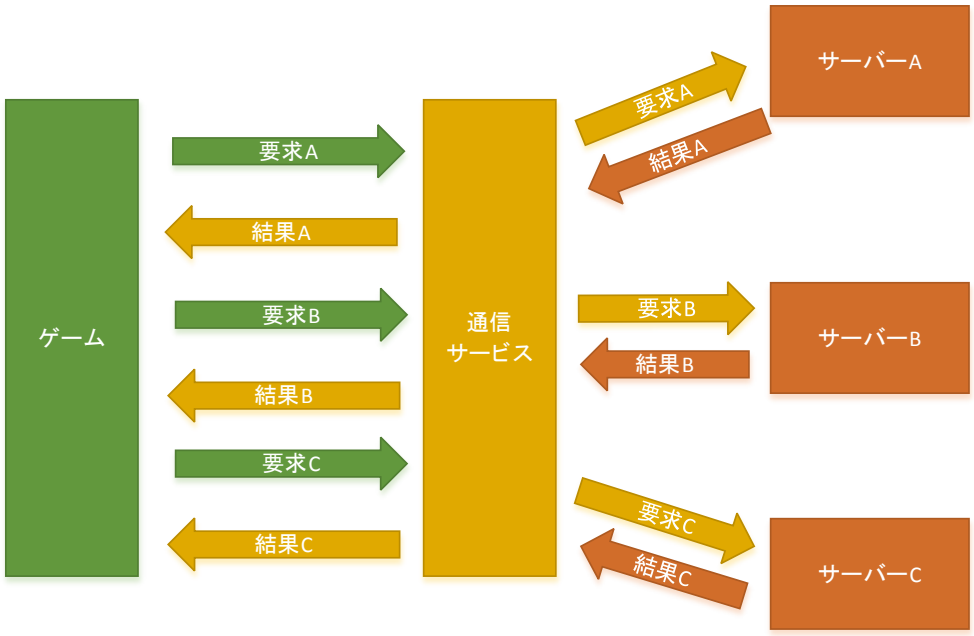
通信サービスが汎用的な通信処理とデータ構造で通信するのに対して、ゲーム側と通信サービスの間は、ゲーム固有の処理とデータ構造で扱う。

ゲーム側からは、通信サービスがあたかもゲーム固有のサーバーであるかのように見立てる。

▼ 統合型通信サービス

複数の通信要件がある場合、むやみにスレッドを増やさずに、一つの通信サービスに統合する。

「統合型サービス」のイメージ：



「通信要件ごとにスレッドを分けたほうが分かり易い」ということもあるかもしれないが、ゲーム側の処理を単純化して、かつ、ゲーム側の処理をブロックすることが無いようにすることが目的なので、できれば少しでも無駄は省きたい。スレッドが増えること自体に、多少のリソース（メモリ）と処理負荷を要する。

なお、ブロック型処理の API を要する通信は統合できないので注意。

▼ 処理要件

このようなサービスの処理は、下記の処理要件に基づいて実装する。

- 通信サービスを常駐スレッドとして稼働させる。
- 通信サービスは、ゲーム側からの要求がない時は、完全にスリープして待機する。
- 通信サービスは、ゲーム側から要求があった時に処理を開始（ウェイクアップ）する。
- 通信サービスは、何らかの処理中に別の処理要求があったら、実行可能な状態になるまで要求をプールする。
- 通信サービスの待機時と稼働時に処理負荷の格差があってはならない。
  - サービスが稼働したとたんにゲーム側が処理落ちするようなことがないようにする。
- ゲーム終了時は、ゲーム側から通信サービス側に、サービスの終了要求を出して終了さ

せる。

▼ 実装方法

以上の要件に基づいて、実装方法を示す。

- 通信サービスのスレッドの優先度は、ゲーム側よりも低く設定する。
- 通信サービスの待機処理は、「条件変数」などの「モニター」の手法を用いてスリープとウェイクアップを行う。
  - いつ有効になるかわからない処理は、できる限りスリープして他の処理を妨げないようにする。
  - 条件変数とモニターについては、別紙の「マルチスレッドプログラミングの基礎」参照。
- 通信サービスへの処理要求は別途キューを設けて記録する。
  - このため、ウェイクアップ処理自体は単純なフラグの通知で行う。
  - プライオリティキューとして扱い、終了要求などは常に最優先で扱う。
- 通信サービス内で負荷の高い処理は行わない。
  - どうしても必要なら頻繁にスリープする。
  - 迅速なレスポンスが求められる要求は基本的にないはずなので、高負荷になるぐらいなら、時間をかけて処理する。

■ 並行ジョブスケジューラサービス

マルチコア（マルチプロセッサ）環境でのゲームのパフォーマンスを最適化するために、並行処理を前提とした基本ゲームシステムを設計する。

少しでも多く、手軽で最適な並行処理を行うために、「並行ジョブスケジューラサービス」を実装する。

▼ 「並行」と「並列」

別紙の「マルチスレッドプログラミングの基礎」にも記述しているように、「並行」（Concurrent）と並列（Parrarel）は厳密には異なる意味である。

本当に物理的に複数の処理を同時に実行できるのが「並列」であり、そうである場合もそうでない場合も含めて、複数の処理を同時に（見せかけて）動作させるのが「並行」である。（「並行」は「並列」を内包する）

▼ マルチスレッドとパフォーマンス

マルチスレッドの意義は、前述している通り、「メインループをブロックすることなく別の処理を実行したい」「フレームに跨がる継続的な処理を実行したい」といった「並行」処理の要件が基本である。

ここで示すのは「パフォーマンス向上」のための「並行」処理である。

そのため、「並列」処理が可能なハードウェア環境でなければ意味がない。そうでない場合、むしろコンテキストスイッチの機会が増えて、パフォーマンスが劣化する。メモリも無駄に要することになる。

▼ 「パフォーマンス向上」のための並行処理の要件

パフォーマンス向上のために並行化できる処理の要件は多い。

アニメーション処理、移動処理、コリジョン判定、AI、HUD、描画処理など。およそ依存関係にない処理はなんでも並行化できる。

当然処理には「移動」→「コリジョン判定」のような順序性が必要となるが、それぞれの処理系の中で並行実行可能なものは多い。

ここで重要なのは、各処理系が並行処理を簡単かつ安全かつ最適に実装できるようにすることである。

そのために、汎用的なジョブスケジューラサービスを使用する。

▼ 並列処理の形態

メインプロセッサのマルチコアを利用した並列処理と、外部プロセッサ（SPU や GPU など）を利用した並列処理では扱い方が異なる。

前者は、ごく普通のスレッド（プログラム中の関数など）を並列実行するもので、プログラミングの負担も小さい。後者は、専用プログラムを作成し、実行時の際に別途プログラムのロードを行う必要もあるので負担が大きく制御が難しい。

本書は、前者の処理を扱うことを前提とする。そのため、多数のスレッドの動作状況に応じて並列実行は保証しきれないので、「並行」と表記している。

なお、SPURS は後者のジョブスケジューラであり、並列以外の並行実行はありえない。

▼ 「ジョブ」と汎用スケジューラサービス

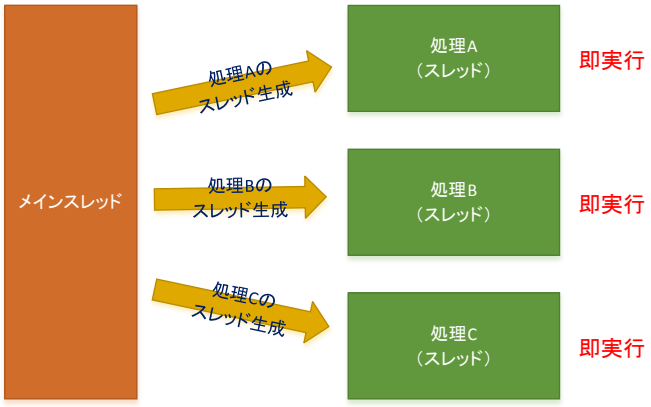
通常、マルチスレッドプログラミングでは、「createthread」や「std::thread」といった

関数／クラスを用いて、指定の処理を「スレッド」化し、並行実行する。

「ジョブ」はスレッドの一種ではあるが、即座に実行しないことが大きな違いである。

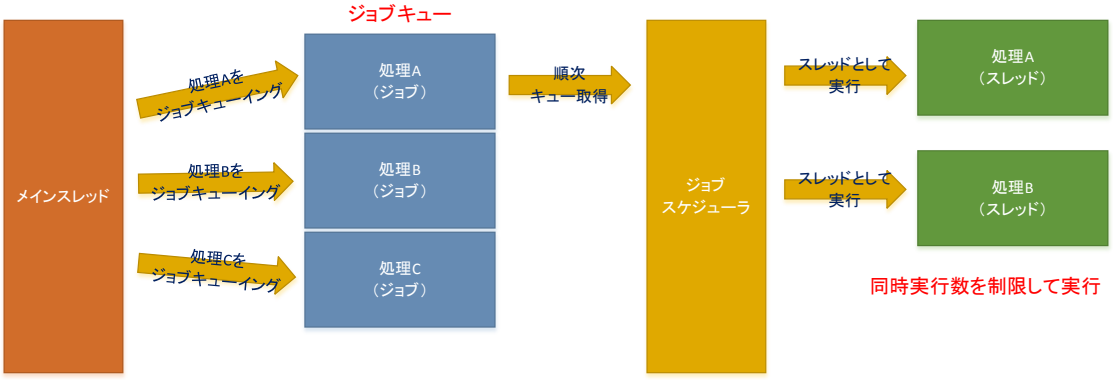
ジョブは実行待ちのキューに溜め込まれ、「ジョブスケジューラ」がキューからジョブを拾って実行する。この時、可能な限り並行実行するが、規定数を超える並行処理は行わない。

通常スレッドの実行イメージ：



・過剰なスレッドが作られると、過負荷、リソース（メモリ）不足が起こりえる。
・処理の依存関係を解消するためには、メインスレッドもしくはそれぞれの処理系が独自に判定する必要がある。

ジョブスケジューラを通したスレッドの実行イメージ：



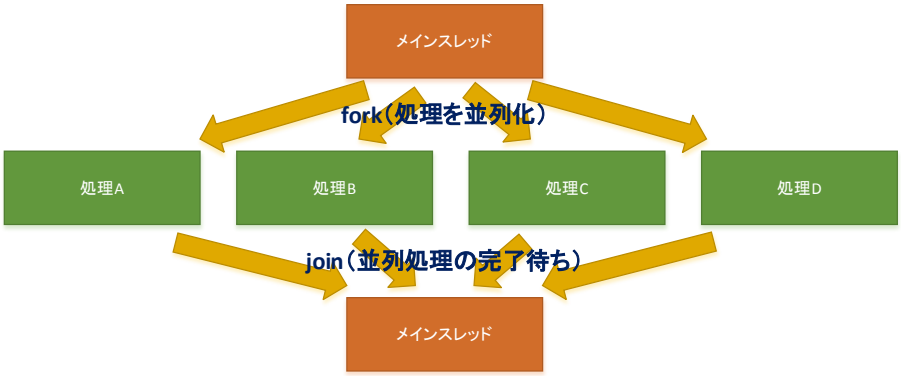
▼ fork-join モデル

ジョブ（スレッド）は、基本的にはどこかで完了を待つ必要がある。

ものによっては実行だけさせて放っておいてもよいが、大抵は次の処理フェーズに進む前に、一通りの並行処理の完了を待つ必要がある。

このような並列処理モデルを「fork-join モデル」（フォーク・ジョインモデル）と呼ぶ。

fork-join モデル：



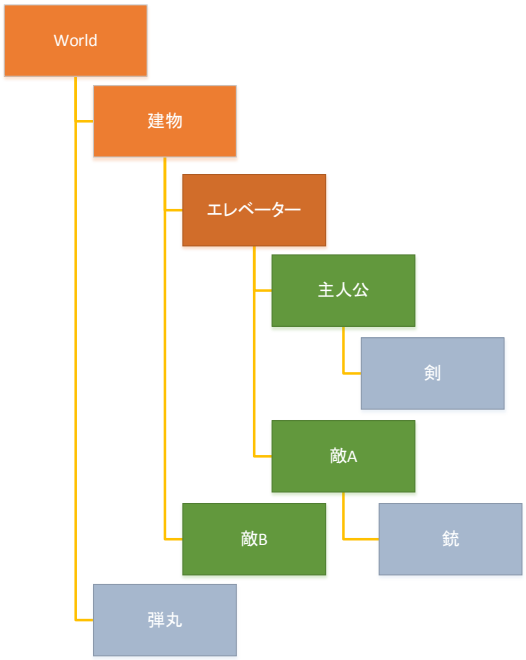
また、ジョブ間には依存関係があり、依存するジョブが終了してからでないと実行できない場合がある。

以上のように、「ジョブの完了を待つ」処理が必要になる点には留意が必要である。

▼ 【具体的な処理の例】アニメーション・移動処理の並行化

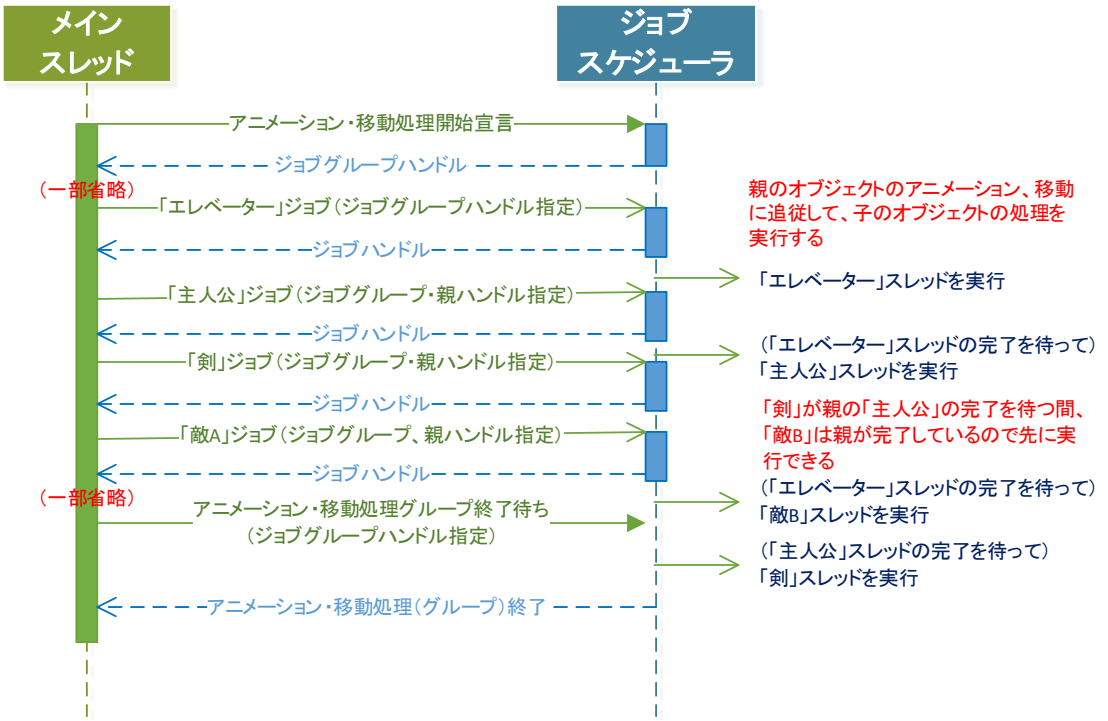
例えば、次のようなシーングラフの処理を行う。

シーングラフ：



このシーングラフでのアニメーション・移動処理の処理シーケンスはこのようになる。

アニメーション・移動処理シーケンス：



処理を解説する。

- 一連の処理の完了を待つ「fork-join モデル」の対応は、処理開始前にジョブスケジューラから「ジョブグループ」のハンドルを発行することで簡単に扱えるようにする。
- まず、ジョブグループ発行後、ジョブを投入する際に、都度ジョブグループを指定する。
- ジョブを投入すると、ジョブスケジューラはすぐに「ジョブハンドル」を返し、ジョブは実行待ちとしてキューイングする。
- 親子関係にあるジョブを投入する際は、ジョブグループと共に、親のジョブハンドルを指定する。
- ジョブスケジューラは、キューイングされた順にジョブを実行していくが、親の（ハンドルの）ジョブが完了していないものは実行せず、実行可能なものを優先して実行する。
- すべてのジョブが完了したかどうかは、ジョブグループのハンドルを指定して待ち受ける。

▼ 実装方法（処理要件）

このようなサービスの処理は、下記の処理要件に基づいて実装する。

- 並行ジョブスケジューラサービスを常駐スレッドとして稼働させる。
- 並行ジョブスケジューラサービスは、ゲーム側からのジョブの投入が無い限りは、完全にスリープして待機する。
- 並行ジョブスケジューラの処理は、メインスレッド側の処理とサービス側の処理に分かれる。

  【メインスレッド側の処理】
  - ジョブグループハンドルの発行
  - ジョブのキューイングとジョブハンドルの発行
  - ジョイン処理
    - ジョブハンドル／ジョブグループハンドルを指定して終了待ち
    - メインスレッドはスリープして待機する

  【並行ジョブスケジューラサービス側の処理】
  - キューイングされたジョブの実行（スレッド化）
  - ジョブの依存関係に基づくスケジューリング
  - 現在メインスレッドで終了待ち中のジョブ／ジョブグループがあれば、可能な限り優先的に実行する

▼ 実装方法

以上の要件に基づいて、実装方法を示す。

- 並行ジョブスケジューラサービスのスレッドの優先度は、メインスレッドより高めに設定する。
- 並行ジョブスケジューラサービスが稼働させる処理スレッドの優先度は、メインスレッドと同じに設定する。
  - ジョブ投入時の指定によって変更可能。
- 並行ジョブスケジューラサービスの待機処理は、「条件変数」などの「モニター」の手法を用いてスリープとウェイクアップを行う。
  - いつ有効になるかわからない処理は、できる限りスリープして他の処理を妨げないようにする。
  - 条件変数とモニターについては、別紙の「マルチスレッドプログラミングの基礎」参照。
- メインスレッド側のジョイン処理もモニターを活用し、スリープして待機する。

・ サービス側の処理は、基本的に瞬間的に完了するばかりを行う。
  - そのほとんどは待ち受け処理。
  - メインスレッドからの処理要求の待ち受け、実行したスレッドの完了の待ち受けを行う。
  - キューイングされたジョブに即時反応できるように、長くブロックするような処理があってはならない。
  - 例えば、C++11 ライブラリの「std::thread::join()」関数は、スレッドの完了をブロックして待つ関数である。このようなブロック処理は一切使用してはいけない。モニターなどを活用して、できる限り「普段はスリープ」「何かあれば即時反応」を実現する。

■ マルチスレッド処理の最適化のために

以上の処理をさらに最適化するための手法を示す。

▼ 【最適化】条件変数で待ち受け処理を最適化

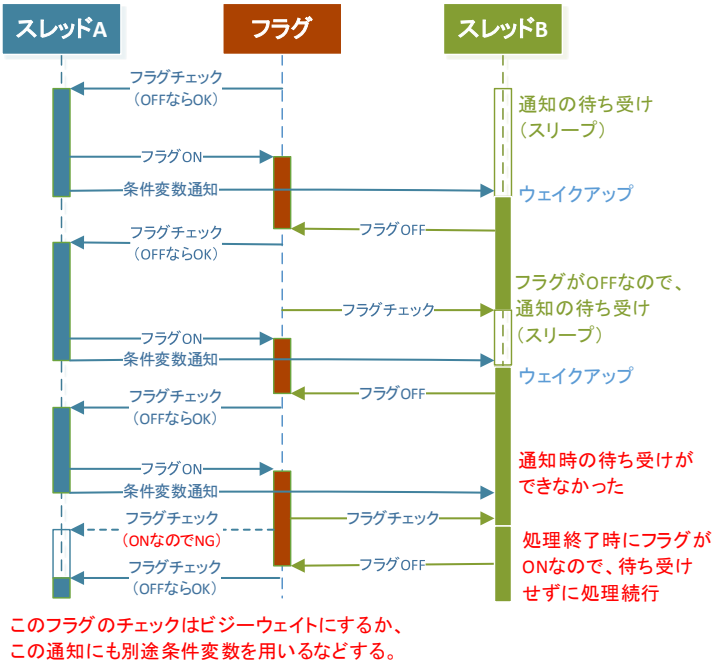
いつ有効になるかわからないスレッドの処理は、フラグをビジーウェイトで監視するような処理を行わず、できる限りスリープして他の処理を妨げないようにする。

このような要件に対しては、条件変数を使用した「モニター」の手法が有効。

条件変数とモニターについては、別紙の「マルチスレッドプログラミングの基礎」に詳しく解説しているが、処理を行う上での注意点があるので、以下に処理のイメージを示す。

通知のタイミングで待ち受けしていないと通知を取りこぼしてしまうので、別途フラグを設けて確実に通知を察知するするようにする。

条件変数を用いた効率的な待ち受けとウェイクアップ処理：



▼ 【最適化】スレッドの再利用で安定化

並行ジョブスケジューリングサービスは、キューイングされた処理を別スレッドで実行する。しかし、頻繁にスレッドを作ったり破棄したりしていると、状況によってスタック領域の確保に失敗するなどの恐れがあり、安定動作に不安がある。

そこで、最大同時実行数のスレッドはあらかじめ作成しておき、未実行状態（待機状態）のスレッドにキューの処理を割り当てて実行するようにする。

これにより、ゲーム中でスレッド数の変動をなくする。

▼ 【最適化】ロックフリーキューの活用

CAS 操作（Compare And Swap：コンペア・アンド・スワップ）を用いることにより、ロックせずにデータをキューイング／スタックするアルゴリズムがある。

「Lock-Free Queue」もしくは「Wait-Free Quee」などのキーワードで検索すると、簡単に見つかる。

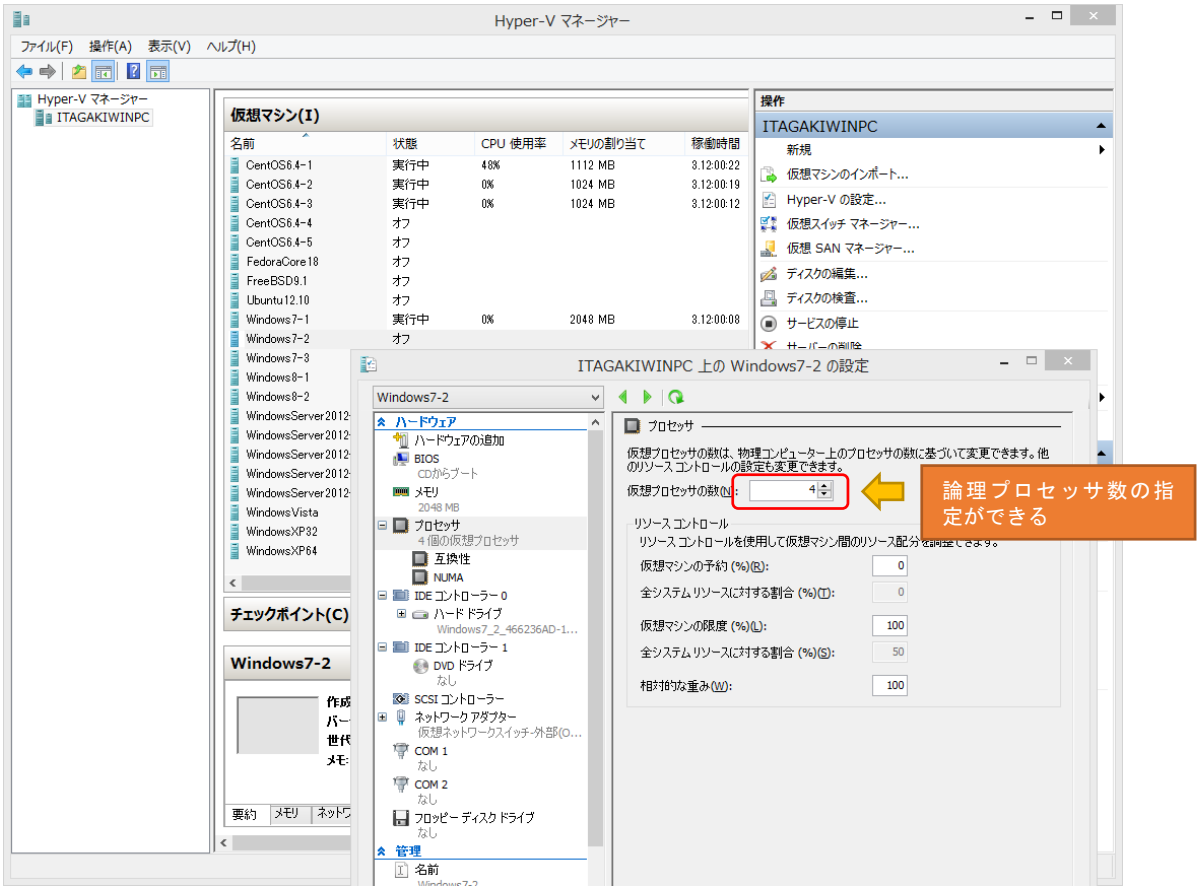
（プライオリティキューだと簡単にはいかないかもしれないが）このような処理手法を用いて、可能な限りロックフリー（ロックしない）を実現する。

▼ 【最適化】最適なパフォーマンスの検証方法

「コア数以外は同じ条件」のハードウェア環境で動作比較ができると便利である。

これを簡単に実現するには、「仮想マシン」が有効である。

「仮想マシン」には、Microsoft の「Hyper-V」や、「VMWare」などが有名。とくに Hyper-V は、「Windows8 Professional 版」にも標準で実装されているのでかなり手軽。



なお、Hyper-V は、「Windows 2012 Server」に実装されているものでないと DirectX が使用できない点に注意。

■■以上■■

■ 索引

索引項目が見つかりません。

「サービス」によるマルチスレッドの効率化

以　上