処理単位の概念と実装モデル

～プロセス、スレッド、コルーチン、非同期タスクの理解～

第1章　はじめに

1.1 背景と目的

コンピュータは、複数の処理を同時またはほぼ同時に進行させる必要がある状況に頻繁に直面する。たとえば、ユーザー入力の受付、ファイルの読み書き、ネットワーク通信、UIの更新などが同時に求められる場合がある。これらの並行した処理を実現するために、コンピュータシステムとプログラミング言語は様々な処理単位の概念を導入してきた。

このレポートの目的は、代表的な処理単位であるプロセス、スレッド、コルーチン、非同期タスクなどの動作モデル、メモリ管理、制御機構について正確に理解することである。さらに、各処理単位がどのようなシステム資源を消費し、どのようなアプリケーションに適しているかを比較検討する。対象読者は情報系大学1年生とし、基本的なプログラミングとオペレーティングシステムの知識を前提とするが、専門的な知識がなくても理解できるように配慮する。

1.2 並行処理とその必要性

並行処理（concurrency）は、複数の計算を同時に進行させるための設計手法である。現代の計算機資源、特にマルチコアCPUや非同期I/O装置の普及により、単一の制御フローでは資源を効率的に活用することが難しくなっている。このため、アプリケーションの実行効率を高めたり、応答性を確保したりするために並行処理は不可欠な技術となっている。

また、リアルタイム性やユーザインタラクションの応答性を確保する観点からも、並行処理は重要である。たとえば、ネットワークサーバは複数のクライアントからのリクエストを並行に処理する必要があり、GUIアプリケーションはユーザーの操作をブロックせずにバックグラウンドで計算処理を進める必要がある。

1.3 このレポートで扱う処理単位の概要

本レポートでは、以下の処理単位を中心に、それぞれの構造、挙動、実装モデル、適用例について論じる。

- ●プロセス（Process）：OSにより管理される最も基本的な処理単位。アドレス空間が分離されており、安全性が高い。
- ●スレッド（Thread）：同一プロセス内で複数の処理を実行可能にする軽量な実行単位。アドレス空間を共有する。
- ●・コルーチン（Coroutine）：関数呼び出しを一時中断・再開できる構造を持ち、非同期処理や逐次的な制御に向く。

●・非同期処理（Async/Await）：イベントループやタスクベースのモデルにより、I/O待ちの間も他の処理を継続できる機構。

●・その他（Fiber、Goroutine、Actor Model など）：各言語や環境に特有の処理単位についても必要に応じて触れる。

これらの処理単位は、CPU時間やメモリ、入出力デバイスといった計算機資源の有効活用に直結する設計要素であり、正しい理解は高効率かつ安定したプログラムの実装に不可欠である。

第2章　プロセス（Process）

2.1 定義と役割

プロセスは、オペレーティングシステム（OS）によって管理される基本的な実行単位である。実行中のプログラムは、プロセスとして扱われ、その中には実行コード、メモリ空間、開かれたファイル、ネットワークソケット、入出力状態などが含まれる。プロセスは、他のプロセスとは独立した資源を持ち、それぞれが独立してスケジューリングされ、実行される。

プロセスの主な役割は、プログラムの安全かつ効率的な実行を可能にすることである。OSは、複数のプロセスを管理し、時間的に並行して動作するように切り替えを行うことで、ユーザに複数のタスクが同時に進行しているように見せる。

2.2 メモリ構造

各プロセスは、OSから割り当てられた独自の仮想アドレス空間を持ち、他のプロセスからのアクセスを遮断している。この隔離により、あるプロセスのバグが他のプロセスに影響を与えることは基本的にない。仮想アドレス空間は、通常コード領域、データ領域、ヒープ領域、スタック領域に分かれており、それぞれの用途に応じて動的に拡張・縮小される。

仮想記憶は、物理メモリよりも大きなアドレス空間をプロセスに提供する技術であり、ページングやスワッピングによって実現される。仮想アドレスはページ単位で物理アドレスにマッピングされ、必要なときにのみ実際のメモリにロードされる。このしくみにより、複数のプロセスが効率的にメモリを共有しつつ、安全に隔離された環境で動作できる。

2.3 関数の実行とスタック

プロセスは、関数の呼び出し履歴とローカル変数の格納に使用されるスタック領域を保持する。スタックはLIFO（Last-In, First-Out）構造であり、関数の呼び出しごとにスタックフレームが積み重ねられ、関数の終了時に対応するフレームが破棄される。各プロセスは独自のスタックを持ち、スレッドを使わない限りこのスタックは1つのみである。正しいスタック管理は、プログラムの信頼性に直結する。

2.4 実行制御の仕組み

プロセスの実行順序とCPU割り当ては、OSのスケジューラが管理する。スケジューリングアルゴリズムには、ラウンドロビン、優先度ベース、マルチレベルキューなどがあり、状況に応じてプロセスを切り替えながら実行する。OSはプロセスの状態（実行中、待機中、終了など）を追跡し、コンテキストスイッチにより、あるプロセスから別のプロセスへの切り替えを実現する。

新しいプロセスを作成するためには、システムコールが利用される。UNIX系OSでは、forkシステムコールが親プロセスの複製を生成し、その後execシステムコールで別のプログラムに置き換えることができる。この組み合わせにより、新しいプロセスを任意の実行バイナリで開始することが可能になる。Windowsでは、CreateProcess関数がこれに相当する機能を提供する。

第3章　スレッド（Thread）

3.1 定義と特徴

スレッドは、プロセス内における軽量な実行単位であり、同一のアドレス空間内で複数のスレッドが同時に実行されることにより、並行性を高める手段となる。各スレッドは独立した実行コンテキスト（プログラムカウンタ、レジスタ、スタックなど）を持つが、プロセスのコード、データ、ヒープなどの資源はすべてのスレッドで共有される。

スレッドを用いることで、リソースの共有が容易になり、コンテキストスイッチのオーバーヘッドがプロセス間よりも小さくなる利点がある。一方で、共有資源の保護が必要となり、同期処理が不可欠となるため、設計と実装には注意が求められる。

3.2 メモリ構造

スレッドは同一プロセス内に存在するため、コード領域やヒープ領域はすべてのスレッドで共有される。一方、各スレッドは独立したスタック領域を持ち、関数呼び出しやローカル変数はこのスタック上で管理される。スタックはスレッドごとに割り当てられ、スレッドの実行中にスタックフレームの積み重ねと除去が行われる。

このように、共有と独立を適切に分離することで、効率的かつ安全な実行が可能となる。ただし、スタックオーバーフローやヒープとの境界管理には注意が必要である。

3.3 関数の実行と再入性

複数のスレッドが同じ関数を同時に実行する場合、その関数は再入可能（reentrant）でなければならない。再入可能性を確保するためには、関数内部で使用される変数をスレッドごとのローカル変数として扱い、共有状態の変更には適切な同期処理（ミューテックス、セマフォなど）を用いる必要がある。

スレッドセーフな設計は、データ競合（race condition）を防ぎ、意図しない振る舞いを回避する上で不可欠である。また、ライブラリ関数などを使用する際には、その関数がスレッドセーフであるかを事前に確認する必要がある。

3.4 実行制御の仕組み

スレッドは、OSによって直接管理されるカーネルスレッドと、ユーザ空間で管理されるユーザスレッドに分類される。カーネルスレッドは、OSスケジューラによって直接CPUに割り当てられるが、コンテキストスイッチのコストがやや高い。一方、ユーザスレッドは軽量だが、カーネルによる個別の管理がなされないため、1つのスレッドのブロックが全体に影響を及ぼすことがある。

スレッドの実行順序は、OSのスケジューラによって決定される。プリエンプティブ（preemptive）スケジューリングを採用しているOSでは、スレッドが一定時間実行された後、強制的に別のスレッドへと切り替えられる。これにより、公平性と応答性が確保される。

一方、ユーザレベルで協調的に制御する場合（たとえば一部のコルーチン系実装）では、明示的な制御移譲によってスレッドの切り替えが行われるため、予測可能性が高くなる反面、プログラムの責任で制御を正しく行う必要がある。

第4章　コルーチン（Coroutine）

4.1 定義と特徴

コルーチンは、実行中の関数を任意の位置で一時中断し、後からその続きから再開できる制御構造である。通常の関数呼び出しが一方向であるのに対し、コルーチンは呼び出し元との間で制御を双方向にやり取りできる。これにより、逐次的な記述でありながら、非同期的または協調的な処理を実現できる。

コルーチンは、非同期プログラミングや状態機械の実装、反復処理、協調的マルチタスクなどに適しており、特にI/O待ちを含む処理において効率を発揮する。言語によっては、ジェネレータと呼ばれることもある。

4.2 メモリ構造

コルーチンは関数の途中で実行を停止し、再開する必要があるため、実行状態（プログラムカウンタ、レジスタ、スタックの一部など）を保存しておく必要がある。このため、コルーチン実装では、スタックのスナップショットを保存・復元する機構が必要となる場合がある。保存されたスタックは、次回の再開時にそのまま復元され、直前の状態から再び実行を継続できる。

一部の言語では、コルーチンの局所変数や中断点の状態がヒープに保存されることで、関数の中断・再開を可能にしている。この方式では、スタックそのものを退避するのではなく、変数の状態と実行位置をヒープ領域に構造化して保持し、再開時にその情報を元に実行環境を再構築する。

4.3 関数の実行モデル

コルーチンは、通常の関数とは異なり、実行中に明示的な命令（たとえばyield）により中断される。呼び出し元はこの中断により制御を受け取り、必要に応じて再びコルーチンをresumeすることで、続きの処理を再開できる。

コルーチンが再開された場合、そのコルーチンは再開を命じた関数のコンテキスト上で実行される。したがって、中断前とは異なる関数（あるいはスレッド）によって再開されることも可能であり、その動作は通常の関数呼び出しとは異なる。コルーチンが終了した場合、制御は最後にそのコルーチンをresumeした呼び出し元に戻る。これは、コルーチンが呼び出し関数の中に積み重なるのではなく、独立した実行単位としてスケジュールされるためである。

このモデルにより、イベント駆動型の非同期処理を、状態遷移に基づく分岐ではなく、逐次的なコード構造として記述できる。これは可読性と保守性の向上に寄与する。

4.4 実行制御の仕組み

コルーチンは多くの場合、OSによるスケジューリングを受けず、ユーザ空間のライブラリやランタイムによって実行の中断・再開が制御される。これにより、コンテキストスイッチのオーバーヘッドが小さく、軽量な並行実行が実現される。

Python、Lua、C++20、Kotlinなど、多くの現代的なプログラミング言語は、コルーチンを言語レベルまたはランタイムでサポートしている。これらの実装では、コルーチンの状態管理、スタック保存、スケジューリング機構などが自動化されており、プログラマは中断・再開のポイントを定義するだけでコルーチンの使用が可能となる。

第5章　非同期処理（Async/Await）

5.1 非同期と並行の違い

非同期（asynchronous）処理は、ある操作の完了を待つことなく次の処理に進む実行モデルである。対して、並行（concurrent）処理は、複数の処理を同時に進める構造や設計を指す。非同期処理は、必ずしも並列や並行とは限らず、単一のスレッド上で複数のタスクを切り替えることで、効率的にI/O待ち時間を活用する。

非同期処理では、タスクが非ブロッキングで進行し、待機中の操作の完了を通知するイベントやコールバックにより制御が復帰する。このため、I/O待ちが長い処理（ファイル読み書き、ネットワーク通信など）に対して極めて有効であり、スレッド数を増やさずに高スループットを実現できる。

5.2 メモリ構造とタスクの保存

非同期処理においては、非同期関数が一時停止した状態や再開地点を保持するために、コールバックキューやタスクキューと呼ばれる待機構造が使われる。この構造は、通常ヒープ上に存在し、実行順序に従ってイベントループが順次再開処理を呼び出す。

async/await構文を用いた実装では、awaitに到達した時点でその関数の実行コンテキスト（局所変数、現在位置、戻り先など）が中断され、ヒープ上のオブジェクトに保持される。再開のタイミングは、非同期処理の完了通知によってトリガーされる。

5.3 関数の実行と中断点

非同期関数は、通常の関数とは異なり、中断可能な状態で動作する。asyncで定義された関数は、実行開始時にただちに戻り値としてPromiseやFutureに類するオブジェクトを返す。awaitキーワードによって、一時的にその実行が中断され、非同期操作の完了を待機する。

この中断点で関数のスタックフレームは通常の意味では残らず、代わりにランタイムがヒープ上に関数の状態を管理する。非同期操作の完了により、保持された状態が再構築され、関数の残り部分が再開される。再開はイベントループの文脈で行われ、通常は別のコールスタック上で処理される。

5.4 実行制御の仕組み

非同期処理の中核となるのがイベントループである。イベントループは、準備が整った非同期タスクを順次処理するループ構造であり、待機状態の処理を監視し、再開可能となったタスクをコールバックキューから取り出して実行する。これにより、非同期関数の中断と再開を効率的に管理できる。

イベントループは通常1スレッドで動作し、スレッドセーフな設計を簡略化できるが、ブロッキング操作を避ける必要がある。Node.jsやPythonのasyncioなど、多くの非同期フレームワークがこのモデルを採用している。

非同期I/Oの実現には、OSが提供する非同期I/Oシステムコールが使用される。Linuxではepoll、BSD系OSではkqueue、WindowsではIOCPがこれに該当する。これらは、複数のファイル記述子に対して非ブロッキングでのイベント検出を可能にする機構であり、イベント駆動の非同期プログラミングに不可欠である。

これらのシステムコールにより、イベントループは低オーバーヘッドで多くのI/O操作を同時に監視し、効率的な非同期タスクのスケジューリングと実行を実現する。

第6章　ファイバー（Fiber）

6.1 スレッドとの比較

ファイバーは、スレッドと似た実行単位であるが、OSによるスケジューリングを受けない点が大きく異なる。スレッドはプリエンプティブにOSが制御するのに対し、ファイバーはユーザ空間で明示的に制御を移譲する協調的マルチタスクである。そのため、ファイバーは非常に軽量であり、数千〜数万単位の並行処理にも対応可能である。

また、スレッドはカーネルリソースを使用するため作成と切り替えに比較的コストがかかるが、ファイバーはカーネルに依存しないため高速に切り替えが可能である。ただし、1つのスレッド内で動作するため、1つのファイバーがブロックすると全体が停止するという制限がある。

6.2 メモリとスタックの切り替え

ファイバーはそれぞれ独立したスタックを持ち、スタックポインタやレジスタの状態を保存・復元することでコンテキストの切り替えを行う。この切り替えはユーザ空間で行われるため、OSの介入なしに高速なコンテキストスイッチが実現される。

メモリの面では、各ファイバーに割り当てられるスタックは比較的小さく、動的に確保される場合もある。ヒープは共有されるため、適切な同期が必要になるが、ファイバーの制御は単一スレッド内で完結するため、一般には同期の必要性は低い。

6.3 協調的マルチタスクの実行モデル

ファイバーは協調的マルチタスクを採用しており、明示的に他のファイバーへ制御を移譲することでタスクを切り替える。これは、yieldやswitchといった専用の操作によって実現される。したがって、制御の移譲タイミングはプログラム側が責任を持って管理する必要がある。

このモデルでは、プリエンプティブな割り込みが存在しないため、スケジューリングの予測性が高く、リアルタイム制御や状態管理に適している反面、ひとつのファイバーが長時間制御を保持すると他のファイバーの実行が遅延するというリスクがある。

6.4 実行制御とユーザモードの切替

ファイバーの実行制御は完全にユーザ空間において行われ、OSカーネルのスケジューラとは無関係である。これにより、スレッド間の切り替えに伴うシステムコールが不要となり、きわめて軽量な切り替えが可能である。

ファイバーを使用するには、ユーザモードで明示的にファイバーの生成・開始・中断・再開を行うAPIを用いる必要がある。WindowsではCreateFiber/ConvertThreadToFiber関数、Rubyや一部のゲームエンジンでもファイバーに類似する機構が提供されている。

第7章　Goroutine（Go言語）

7.1 概要と特徴

Goroutineは、Go言語における非常に軽量な並行処理の単位であり、関数の並列実行を簡潔に記述できる構文を提供する。goキーワードを用いて関数呼び出しを行うことで、新たなGoroutineが生成され、非同期的に実行される。GoroutineはOSのスレッドとは独立した存在であり、Goランタイムが独自にスケジューリングを行う。

Goroutineの特徴は、作成と切り替えのコストが極めて小さいことである。数千から数十万単位での同時実行が現実的であり、大規模な並行システムの構築に適している。また、スレッドよりも安全かつ簡潔な記法で、共有状態の管理にはチャネルという構文上の機構が用意されている。

7.2 メモリ構造

Goroutineは各自に独立したスタックを持ち、その初期サイズは非常に小さい（数KB）。スタックは実行に伴って動的に拡張され、必要な分だけメモリが割り当てられる。これにより、多数のGoroutineを同時に動作させても、メモリ使用量を抑えることができる。

このスタックの拡張と縮小はGoランタイムが自動的に行い、プログラマはその詳細を意識せずに使用できる。この仕組みにより、Goroutineの効率的な資源管理が実現されている。

7.3 実行モデル

Go言語では、Goroutine間の通信と同期にチャネル（channel）という専用の構文が用意されている。チャネルは、Goroutine間でデータを受け渡すための型付きのパイプであり、明示的なロック機構を使用せずに、スレッドセーフなデータ共有が可能となる。

チャネルを用いることで、Goroutineは協調的に動作し、明確なデータフローを維持したまま通信が可能になる。このモデルは、共有よりも通信を重視するCSP（Communicating Sequential Processes）に基づいて設計されている。

7.4 実行制御の仕組み

Goランタイムは、M個のOSスレッド上にN個のGoroutineを動的に割り当てるM:Nスケジューリングモデルを採用している。これにより、複数のGoroutineが少数のスレッド資源で効率的に実行される。スレッド数を制限しつつ、多数のGoroutineを処理できる点が特徴である。

Goランタイムは、Goroutineの生成、停止、再開、スタック管理、スケジューリングといった処理を内部で担っている。プリエンプティブなスケジューリングも一部導入されており、長時間CPUを占有するGoroutineがあれば強制的に切り替えられる。

このように、Goroutineの実行はOSに依存せず、Go独自のランタイムによってきめ細かく制御される。これにより、高いスケーラビリティと制御性が両立されている。

第8章　アクターモデル（Actor Model）

8.1 分散・並行処理における意義

アクターモデルは、並行処理と分散処理を安全かつ拡張性高く実現するための計算モデルであり、各アクター（actor）が独立して状態を保持し、非同期メッセージを介して相互作用する仕組みを提供する。このモデルは、並行性の制御における競合状態やロックの複雑さを回避し、システムのモジュール性とスケーラビリティを高める。

アクターモデルは特に分散システムやリアルタイム通信、耐障害性が求められる環境において有効であり、ErlangやAkka（Scala）などのプラットフォームが代表的な実装例である。

8.2 メモリと状態の分離

アクターは他のアクターと状態を共有せず、自身のローカルなメモリ空間に状態を保持する。このため、アクター間の直接的なデータアクセスは許されておらず、すべての情報のやり取りはメッセージを通じて行われる。これにより、スレッドセーフな構造が自然に保証される。

アクターは状態を内部で隠蔽することができ、外部の影響を受けにくいため、変更に強く、再利用性や保守性の高い構造を作ることが可能となる。

8.3 関数の非同期呼び出しとメッセージ処理

アクター同士の通信は、非同期メッセージの送受信により行われる。メッセージは送信された後、即座に処理されるとは限らず、各アクターは自身の受信キュー（メールボックス）にメッセージを蓄積し、順次取り出して処理を行う。

この非同期性により、送信元と受信先の実行タイミングを独立に保つことができ、システム全体の応答性と柔軟性が向上する。関数呼び出しの代替としてメッセージを使用することで、呼び出し元が結果を待つ必要がなく、全体の処理が滞りなく進行する。

8.4 実行制御の仕組み

アクターは内部に受信ループを持ち、そこに到達したメッセージを順番に処理する。通常、メッセージ処理は1つずつ逐次的に実行され、他のアクターとの並行性はアクター単位で保証される。これにより、アクター内部ではロック機構を使用せずに状態を安全に変更することができる。

実行制御は、アクターランタイム（たとえばErlang VMやAkkaフレームワーク）により行われ、アクターのスケジューリング、メッセージ配送、エラーハンドリングなどが自動的に管理される。スケジューラはアクターごとに均等に処理時間を分配し、高スケーラビリティな並行システムの構築を可能にする。

第9章　タスク／Future／Promise

9.1 非同期処理の抽象化

タスク、Future、Promiseは、非同期処理の結果を表現・制御するための抽象であり、非同期プログラミングにおいて広く利用されている。これらの構造は、非同期的に進行する処理の状態や結果、エラーを保持し、後続の処理を定義する手段を提供する。

Futureは「将来得られる値」を表し、Promiseはその値を外部から設定する機構である。タスクは、処理の実行単位とその完了状態を表現し、しばしばFutureやPromiseの実装と結びつけられる。これらにより、非同期処理の実行と管理が関数型に近いスタイルで記述可能となる。

9.2 メモリ上のタスク管理

非同期処理の抽象は、ヒープ上にタスクオブジェクトや状態マシンを構築することによって実現される。タスクの実行中は、進行状況や待機中の非同期操作、例外状態などがこのオブジェクトに格納され、非同期処理全体の状態を追跡する。

メモリ上では、依存関係を持つ複数のタスクがツリー構造やグラフ構造として管理されることもあり、それぞれのタスクが独立または連鎖的に完了状態へ移行する。ランタイムはこれらの依存関係を解決し、必要に応じて次のタスクを自動的にスケジューリングする。

9.3 関数の非同期チェーン実行

タスク、Future、Promiseはいずれも「then」や「continueWith」などの構文により、非同期処理をチェーンとして記述できる。これにより、複雑なコールバックのネストを避け、処理の流れを直線的に記述することが可能となる。

各ステップは、前の処理が完了した後に非同期的に実行されるため、非同期連鎖の中でも効率的なスケジューリングとリソース管理が行われる。失敗時の処理（例外捕捉）も統一的なインターフェースで取り扱うことができ、堅牢な非同期プログラムを構築しやすくなる。

9.4 実行制御と完了通知モデル

タスクの完了は、イベントや通知機構によって監視され、登録されたハンドラが呼び出されることで後続処理が起動する。これには、イベントループによるディスパッチや、内部キューの活用が含まれる。

Promiseは、外部から明示的に完了状態（成功または失敗）を設定する点が特徴であり、複数の非同期操作を統合する場合にも利用される。複数のタスクを集約する構文（たとえばPromise.allやFuture.wait）によって、並列実行されたタスクの全体完了を待ち合わせる処理も可能である。

このようなモデルにより、非同期処理の設計はより宣言的かつ制御しやすいものとなり、複雑な処理フローの整理や保守性の向上に寄与する。

9.5 Promiseの詳細と構造

Promiseは、非同期処理の完了通知を抽象化し、安全かつ再利用可能な方法で非同期の結果を取り扱う構造である。Promiseは生成された時点では保留中（pending）という状態にあり、最終的には成功（fulfilled）または失敗（rejected）いずれかの状態へ一度だけ遷移する。この不可逆な状態遷移によって、重複呼び出しや状態競合が回避される。

Promiseは2段階のコールバック構造を持つ。1つは正常完了時に呼び出される成功コールバック（resolve相当）、もう1つは異常時に呼び出される失敗コールバック（reject相当）である。これらを別個に登録できることにより、エラーハンドリングが明確化される。

さらに、Promiseは連鎖可能な設計となっている。あるPromiseが完了した後に、次の非同期処理を登録し、それを連続的に記述することで「コールバックチェーン」を構成できる。この構造により、非同期処理を逐次的なフローとして記述することが可能になり、可読性と保守性が大きく向上する。

設計パターンの観点では、Promiseは「2段階コールバック」と「コールバックチェーン」の組み合わせを構成しており、非同期制御を関数型スタイルで記述可能にする抽象インターフェースである。さらに、状態管理、例外処理、依存関係の表現が統一された仕組みにより、プログラマは複雑な非同期処理を安全に構築することができる。

第10章　イベントループと非同期I/Oの基盤

10.1 イベント駆動モデルの歴史と基盤技術

イベント駆動モデルは、非同期処理のための基本的なアーキテクチャとして、GUIアプリケーション、サーバ、OSカーネルのインフラなどで長年にわたり採用されてきた。代表的な初

期の応用例には、UNIXにおけるファイルディスクリプタ監視機構や、Windowsにおけるウィンドウメッセージ処理などがある。

このモデルでは、アプリケーションが明示的にI/O操作を待つのではなく、発生したイベント（入出力の完了、ユーザー操作など）を通知として受け取り、それに対応する処理を行う。イベントを検出し、処理を分配する中心的な仕組みが「イベントループ」である。これにより、アプリケーションは単一スレッドであっても高い応答性とスケーラビリティを持つことが可能となる。

10.2 メモリ管理とキューの利用

イベントループは、実行中の状態、待機中のイベント、次に処理すべきタスクなどを内部的に管理する。これらの情報は、通常ヒープ上のイベントキューまたはタスクキューに保持される。キューはFIFO構造であることが多く、外部イベントの到着順に処理を行うことができる。

非同期処理においては、タスクオブジェクト、コールバック関数、ステートマシンの状態などを保持するためにヒープメモリが広範に使用される。ランタイムは、タスクの生成・登録・スケジューリング・破棄を自動的に管理し、ユーザが個別にメモリを解放する必要はない設計が多い。

10.3 システムコール（select, poll, epoll, IOCP）

イベントループがI/Oの完了を効率的に検出するためには、OSが提供する低レベルの非同期I/O通知機構を利用する。これには以下のような代表的なシステムコールがある。

selectは、最も古くから存在する機構であり、複数のファイルディスクリプタの状態を同時に監視できる。ただし、監視対象数の制限やスケーリングの課題がある。

pollは、selectの制限を克服するために導入され、より多くのディスクリプタを扱えるが、毎回の監視対象全体の再スキャンが必要である。

epollはLinuxで導入された高効率な監視機構で、監視対象を事前に登録することで変更のない部分を効率的に処理できる。大量の接続を持つサーバにおいて重要な技術である。

IOCP（I/O Completion Port）はWindowsにおける非同期I/Oの基盤であり、イベント駆動とスレッドプールを組み合わせた高度な制御が可能である。これにより、スレッド数を抑制しつつ高スループットが実現される。

10.4 ユーザ空間でのイベントディスパッチ

イベントループは、ユーザ空間において非同期イベントを逐次的に処理する。これは、OSからのI/O完了通知を受け取った後に、登録されたコールバック関数を順次呼び出す構造であり、イベントディスパッチとも呼ばれる。

ユーザ空間でのディスパッチ処理は、非同期処理の中断と再開を効率的に実現する上で中心的な役割を果たす。たとえばJavaScriptのNode.jsやPythonのasyncioなどは、シングルスレッドで動作するイベントループモデルを採用しており、非同期I/Oとイベント処理を統一的に管理している。

この構造により、同期的なコードと見た目が近い非同期プログラムの記述が可能になり、開発効率と可読性が高まると同時に、スレッドの過剰使用を避けつつ高い並行性を実現できる。

第11章　実装例と言語ごとの違い

11.1 Pythonにおけるasyncioとスレッド

Pythonは標準ライブラリとして非同期処理用のフレームワークasyncioを提供しており、イベントループに基づいた非同期I/O処理を記述するための構文としてasync defとawaitを導入している。これにより、Pythonはコールバックベースの非同期プログラミングから、より可読性の高い構造的な記述に移行した。

Pythonのスレッド機能はthreadingモジュールとして提供されているが、グローバルインタプリタロック（GIL）の存在により、CPUバウンドなタスクの並列処理には制限がある。一方、I/Oバウンドな処理では、asyncioを用いた単一スレッドのイベントループによる並行処理が有効に機能する。

11.2 JavaScriptにおけるPromiseとイベントループ

JavaScriptは非同期処理を主とする設計において、最初からイベントループモデルを中心に構築されている。ブラウザ環境でもNode.jsでも、単一スレッド上でイベントとコールバックを効率的に処理するモデルが採用されている。

非同期操作は、かつてはコールバック関数によって記述されていたが、現在ではPromiseとasync/await構文が主流となっている。Promiseは2段階コールバックとチェーン構造を備え、非同期フローを宣言的に記述することができる。イベントループは、マクロタスクとマイクロタスクのキューにより実行タイミングを細かく制御し、高速かつ安定した応答性を実現している。

11.3 Luaにおけるコルーチン

Luaは言語仕様として軽量なコルーチン機構を標準で組み込んでおり、明示的な中断と再開の操作により協調的マルチタスクを簡単に実装できる。coroutine.createとcoroutine.resume、coroutine.yieldといったAPIを使用することで、制御フローを柔軟に制御できる。

Luaのコルーチンはユーザ空間で制御され、OSスレッドを用いずに複数の論理的な流れを切り替えることが可能である。そのため、組込み用途やゲームスクリプトなどでの効率的な状態制御に向いており、イベント駆動の実装にも適している。

11.4 Goにおけるgoroutineとチャネル

Go言語は、軽量スレッドであるgoroutineと、データの受け渡しを行うチャネルを組み合わせた並行処理モデルを採用している。goキーワードによって関数を非同期に実行することができ、数千から数十万のgoroutineが効率的にスケジューリングされる。

goroutineはランタイムによってM:Nスケジューリングされ、スタックも動的に拡張されるため、メモリ効率に優れている。チャネルはデータと制御の同期を同時に扱える構文として提供され、排他制御なしに安全な通信が可能となる。これにより、明確な並行構造を簡潔に表現できる。

11.5 Erlangにおけるプロセスとアクター

Erlangはアクターモデルを言語とランタイムの中核に据えた設計となっており、並行処理の単位として軽量プロセスを使用する。これらのプロセスはOSスレッドとは無関係に実行され、各プロセスは独立したメッセージキューと状態を持つ。

プロセス間の通信はメッセージパッシングによって行われ、状態の共有は一切行われない。これにより、スケーラブルかつフォールトトレラントな分散システムを構築することが可能となる。Erlangランタイムはプロセスの監視、回復、再起動といった機能を内包し、高可用性が要求される通信インフラで広く利用されている。

第12章　まとめと考察

12.1 処理単位ごとの特徴比較

本レポートでは、プロセス、スレッド、コルーチン、ファイバー、Goroutine、アクター、非同期タスクなど、複数の処理単位について、それぞれの構造、メモリモデル、実行制御の仕組みを比較してきた。各処理単位は異なる抽象レベルとトレードオフを持ち、用途や対象となるシステムの性質によって適否が分かれる。

プロセスはアドレス空間を分離することで安全性を高める一方、切り替えコストが高い。スレッドは共有メモリによる高速な通信が可能だが、競合制御が複雑である。コルーチンとファイバーは、ユーザ空間での軽量な協調的切り替えが特徴で、実装の簡素さと実行の効率性を兼ね備える。Goroutineはこれらをランタイムによって管理し、高スケーラビリティを実現している。アクターモデルは状態の隔離とメッセージパッシングによって、高い堅牢性と分散性を提供する。非同期処理は、スレッドに頼らずイベントループを通じて高効率なI/Oを実現する。

12.2 実装における設計判断

これらの処理単位を選択する際には、タスクの性質、パフォーマンス要件、安全性、開発効率、プラットフォーム依存性などを考慮する必要がある。たとえば、リアルタイム性が求められるシステムでは明示的な制御ができるコルーチンやファイバーが適している。逆に、高信頼性と分散性が求められるシステムでは、アクターのように状態を分離したモデルが有効である。

また、非同期タスクやPromiseのような抽象は、ユーザコードから低レベルの実行制御を隠蔽しつつ、効率的な処理記述を可能にする。このような高レベル抽象の導入は、実装の容易さとバグの少なさという観点で大きな利点を持つ。

12.3 将来的な技術の方向性

処理単位の設計は、ハードウェアの進化やアプリケーション要件の変化に応じて進化し続けている。マルチコア、NUMA構成、GPUや専用アクセラレータとの連携など、並列処理の基盤はさらに多様化しており、それに対応する実行モデルの柔軟性が求められている。

今後は、より自動化されたスケジューリング、並行性と安全性を両立する言語レベルでの制約（たとえばRustの所有権モデル）、分散処理とローカル並行処理の統合的管理などが注目される領域となる。また、リアクティブプログラミング、関数型リアクティブモデル、コンティニュエーションなど、制御構造のさらなる抽象化と高階化も進むと予測される。

今後のソフトウェア開発においては、これらの処理単位の特性を正確に理解し、適材適所で適用する設計判断が、性能・信頼性・保守性を大きく左右する重要な技術的基盤となる。