2005－OS－99（10）
2005／5／26

継続概念を用いた Zero-Wait 方式による OS 構成法の提案

日下部　茂[†]　乃村能成[††]
谷口秀夫[††]　雨宮真人[†]

次のような特徴を持つスレッドによりオペレーティングシステムを構成する方式を提案する：スレッド実行開始後は内部の命令を wait なしに実行する，スレッドは継続点を持ちスレッド間の実行制御は継続概念によって行う，スレッドから構成されるプログラムは逐次実行列をノードとし継続関係をエッジとするようなデータフローグラフとなる．この方式により汎用オペレーティングシステムの内部構造単純化と効率的な多重並行処理環境の実現を目指す．構成法の概要に加え，入出力処理を例にした予備評価について論じる．

Proposal of an Operating System Architecture Using Continuation-Based Zero-Wait Threads

Shigeru KUSAKABE,[†] Yoshinari NOMURA,[††]
Hideo TANIGUCHI[††] and Makoto AMAMIYA[†]

In this paper, we propose an operating system architecture using continuation-based zero-wait threads. A thread is supposed to run to completion without suspension, and execution control between threads are performed based on continuations attached to threads. This approach enables us to reduce thread management overhead and to realize flexible scheduling schemes. We discuss the operating system architecture and examines its effectiveness in handling I/O requests.

1. はじめに

データフロー方式は本質的に並列・並行処理に適した実行方式である．プロセッサアーキテクチャにおいてはデータフロー概念から派生した様々な技術が提案されている．データフロー方式のプロセッサや既存命令セットアーキテクチャのプロセッサ高速化技術への応用に加え，革新的な命令セットアーキテクチャの一つとして一種のデータフロー実行方式を実現するアーキテクチャおよびそのコンパイラが提案されている[1)6)7)8)9)]．しかしながら，論理的には多数の並列・並行処理を扱うオペレーティングシステムにおいては，データフロー概念の積極的な導入はなされていない．近年は並行・並列実行の単位としてスレッドがサポートされているが，それらは逐次処理を前提に作成された複数のプログラムを多重に実行するためのプロセスモデルから派生したものである．

我々は，オペレーティングシステムにもデータフロー概念を導入することが有効と考える．オペレーティングシステムは基本的に受動的でイベントの発生に対応した処理を行う．計算機システムの利用形態の多様化にともない，多数の非同期イベントを効率的に扱うことの重要性は増大すると考える．イベントが非同期かつ大量に発生する場合には，逐次実行方式を基本とするより，本質的に非同期実行であるデータフローモデルをベースにしたスレッド実行方式を基本とする方がよいと考える．

本稿では，次のような特徴を持つ Zero-wait スレッドによりオペレーティングシステムを構成する方式を提案する：スレッド実行開始後は内部の命令を wait なしに実行する，スレッドは 0 個以上の継続点を持ちスレッド間の実行制御は継続概念によって行う．スレッドから構成されるプログラムは逐次実行列をノードとし，継続関係をエッジとするようなデータフローグラフとなる[☆]．このようなスレッドを用いることで汎用オペレーティングシステムの内部構造単純化と効率的な多重並行処理環境の実現を目指す．

オペレーティングシステムのカーネルには，遅延を

† 九州大学大学院システム情報科学研究院
Faculty of Information Science and Electrical Engineering, Kyushu University
†† 岡山大学工学部
Faculty of Engineering, Okayama University

☆ 以降，特に説明が無いかぎりスレッドといえばこの継続点をもつ zero-wait スレッドを指すこととする．

生じる可能性のある処理が多いため，耐遅延性を高めるスケジューリングが容易なスレッド実行方式を導入する利点が大きいと考える．遅延を生じる可能性のある処理においては，処理の要求と結果受け取りをそれぞれ別のスレッドとして分離して扱うスプリットフェーズ方式を用いる．遅延を生じる処理とスレッド実行処理をオーバラップさせ，遅延を隠蔽し全体としての処理スループットを向上することができる．また，継続点によるスレッド実行制御を行うため，継続対象の処理結果受け取りスレッドを特定して一意に実行可能状態にすることができる．これによりスループットの向上に加え応答時間の改善も期待できる．

カーネルでの処理において処理待ち状態を生じる高遅延の処理の 1 つに入出力処理がある．本稿では，提案方式と UNIX 系オペレーティングシステムのひとつである Linux との間で，特にこの入出力処理を中心に比較を行う．近年，Web サーバやデータベースなど，高い並行度を持ち多数の入出力処理を同時並行に行う応用プログラムも多くなっており，今後もこの傾向は続くと考えられる．入出力処理で，wait キューにより処理待ちスレッドの管理をする方式では，同時に多くの入出力要求が出されたときにスレッド起床処理のオーバヘッドが大きくなる．一方，Zero-wait スレッド実行のように継続点によるスレッド実行制御を用いる方式では，入出力要求数に左右されることなく安定した性能を達成できると考えられる．

本稿の構成は以下の通りである．第 2 節では Zero-wait スレッド実行方式について説明する．次に第 3 節では Zero-wait スレッド実行方式によるカーネル構成の概要とその利害得失について考察する．次に第 4 節では，特に入出力処理に着目し，Zero-wait スレッド実行方式の利点について評価を行う．

2. Zero-wait スレッド実行方式

本節では，Zero-wait スレッド実行方式の概要について説明する．Zero-wait スレッドは以下のような特徴を持つ．

- プログラムはスレッドをノードとするデータフローグラフ様のものとして構成される．そのエッジは単方向でスレッド間の継続 (依存) 関係を表す．
- 各スレッドはひとつの実行開始点を持ち，いったん実行が開始されるとスレッドは wait なしに走り切る命令列である．実行は開始点から始まり命令列の途中から始まることはない．
- スレッドの実行順序は半順序関係である．スレッドは次に実行すべきスレッドを継続命令で指定できる．ただし直接の継続関係があるスレッドでもそれらは必ずしも連続して実行されるわけではない．継続命令によってスレッド間で制御が移るのは，指定されたスレッドの先頭の実行開始点である．以降，継続先スレッドの実行開始点を継続点と呼ぶ．また一つのスレッドが複数の継続点を指定できる．
- 一つのスレッドは複数の先行スレッドを持ち得る．その場合，全ての先行スレッドの継続命令の完了後，スレッド間の依存関係が解消し実行可能となる．
- スレッドは既存の逐次コードに類似したものであるが，スレッド実行制御のための界面を持つ．スレッドの途中を継続点とすることはできない．

スレッドには wait 状態がなく，ひとたび実行されると走り切る．ただし，継続処理が完了し，実行可能状態になってから，プロセッサ資源が割り当てられるまでの実行待ち状態はある．実行可能状態になったスレッド同士の実行順序は任意であるため，実行状況に応じた柔軟なスケジューリングを行うことも可能である．

Zero-wait スレッド実行方式ではスレッドの待ち状態がなくなる．したがって，明示的に待ち状態を記述することはない．一方，例えば POSIX スレッドのような場合だと，遅延が生じる可能性のある処理呼び出しに加え，明示的に待ち状態の記述が必要である (**図 1**)．Zero-wait スレッド実行方式では，そのような箇所はスレッドの切れ目として扱う．ハードウェアレベルのマルチスレッド処理では，スレッドが明示的なものと暗黙的なものがある[10]．本方式はソフトウェアレベルのもので明示的なスレッドを扱う．そのため静的に予測できない例外などはスレッドの切れ目とすることはできない．

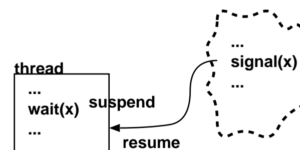

図 1 スレッドの wait

本方式では遅延が生じる可能性のある処理は，処理要求と処理結果受け取りをそれぞれ別のスレッドとして分離して扱うスプリットフェーズ処理で行う (**図 2**)．結果を待つ間，他の実行可能なスレッドの実行をオーバラップさせ，全体として遅延を隠蔽することができる．これにより，全体としての処理スループットを向上することができる．

3. Zero-wait スレッド実行方式によるカーネル

本節では，前述の Zero-wait スレッドを用いてカーネルを構成することを検討する．文献 3) でも Zero-wait スレッド方式を導入する提案をしているが主に

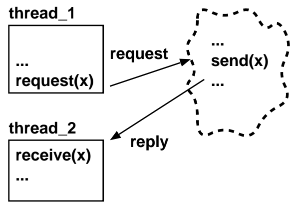

図 2 Zero-wait スレッドによるスプリットフェーズ実行

ユーザ空間での議論である．一方，本方式はカーネルに Zero-wait スレッドを用いる．

3.1 Zero-wait スレッドによるカーネル

オペレーティングシステムは基本的に受動的で，非同期なイベントの発生に対応した処理を行う．我々は，イベントが非同期かつ大量に発生する場合には，逐次実行方式を基本とするより，本質的に非同期実行であるデータフローモデルをベースにしたスレッド実行方式を基本とする方がよいと考える．Zero-wait スレッド実行方式では，スレッド間の実行制御はデータフロー方式のように継続概念に基づく方式で行われる．カーネルは，ユーザアプリケーションからのサービス要求，下位のハードウェアレベルからの割り込みといったイベントに対処するハンドラスレッドを持つ．カーネルの界面をまたぐ処理も継続概念に基づいて行われ，カーネル内のこれらのイベントに対処する処理を起動するにはハンドラスレッドを継続点として指定する．

例えば，システムコールを経由してユーザ空間からカーネルサービスを利用するような，システムコールの界面をまたぐ処理も継続概念に基づく．このため，ユーザ空間側でも，システムコールの呼び出しの前後では処理要求と処理結果受け取りのスレッドに分離される．システムコールの呼び出しでは，ユーザ空間で処理待ち状態になり得ると考えられるが，スレッドのスケジューリングにより遅延を隠蔽できる．

3.2 Zero-wait スレッド方式のための実行機構

3.2.1 同 期 機 構

Zero-wait スレッドの実行制御を行うためには，以下のような機構により継続処理にともなう同期機構を実現する必要がある．

(1) ひとつのスレッドが複数の先行スレッドから継続として指定され得る．そのためスレッドは，同期カウンタを持つ．その初期値は先行スレッド数である．

(2) 先行スレッドからの継続処理を行うと，対象スレッドのカウンタ値をデクリメントする．もしカウンタ値が 0 になった場合は，対象スレッドは実行可能状態になる．

3.2.2 カーネル内スレッドスケジューラ

同期が成立し実行可能になったスレッドをスケジュールするスケジューラがカーネル内に必要である．スケジューラは，スレッド終了処理を受け，実行可能なスレッドから次に実行するスレッドを選び出し，切り替え処理を行う．

3.2.3 自己継続処理

カーネル内の処理では，排他的な処理を行う箇所がある．このような箇所をスレッド化する場合には，スレッドの排他的な実行を保証するための機構が必要である．そのひとつとして，自分自身への継続処理を行う方法がある．この処理を以下に示す．以降，自分自身への継続処理を自己継続処理と呼ぶ．

同一スレッドに対して複数の実行を行う場合は，異なる実行インスタンスのデータが干渉しないようにする必要がある．動的なデータフロー方式では，スレッドの多重並行処理のためタグやカラーなどを用いて異なるインスタンスの区別を行う．しかしながら排他実行スレッドは多重に起動できないため，静的データフロー方式のような制御機構で干渉を防ぐ必要がある．本方式においては，多重に起動できない排他スレッドに対して多重にスレッド起動を試みる場合は，対象スレッドの同期カウンタ値がすでに 0 になっている．このようなスレッドに対して継続処理を行った場合は，処理要求をキューイングする．排他スレッドは終了時に以下のような自己継続処理を行い，非同期に発生する複数の起動要求に対処する．

(1) 自スレッドの同期カウンタ値を指定された値でリセットする．

(2) 自スレッドに対して通常の継続処理を行う．また，バッファリングされている継続処理要求の有無を調べ，あれば処理する．

3.3 利 害 得 失

本節では，Zero-wait スレッド実行方式でカーネルを構築することによる利害得失について議論する．

3.3.1 スレッド管理

3.3.1.1 スレッド状態管理

Zero-wait スレッド実行方式の場合，スレッドのとりうる状態および状態遷移が少なくなるため，スレッド状態を管理する処理が簡潔になると考えられる．前述のように，スレッド実行方式では，遅延を伴う箇所では処理要求と処理結果受け取りをそれぞれ別スレッドとして分離しているため，スレッドの処理待ち状態がなくなる．

スレッド状態を管理する処理が簡潔になれば，スレッド実行制御のハードウェア（HW）化が容易になると考える．スレッド管理部を HW 化することにより，スレッド切り替え処理の高速化が望める．この場合，スプリットフェーズ実行以外の箇所においてもスレッドを分割することで，積極的な細粒度化の恩恵を受けることも可能となる[5]．

3.3.1.2 スレッドの切り替えコスト

走りきりでないスレッドでの wait に相当する部分を，処理要求と処理結果受け取りのスレッドに分割するため，スレッドが取り得る状態数は減少するものの

スレッド数は増加する．この場合，走りきりの前提のないスレッドによるカーネルでのスレッド数と比べて，大まかに見積もってスレッド数は定数倍程度増えると考えられる．このため，スレッド切り替えのコストは小さくなると考えられるものの，スレッド切り替えの回数は増加すると考えられる．

3.3.2　スループット

オペレーティングシステムのカーネルの処理には，遅延が生じる可能性のある処理が多いため，Zero-wait スレッドを用い耐遅延性を高めるスケジューリングを行う利点は大きいと考える．遅延が生じる可能性のある処理においては，処理の要求と結果受け取りをそれぞれ別のスレッドとして分離して扱うスプリットフェーズ方式を用いる．遅延を生じる処理とスレッド実行処理をオーバラップさせ，遅延を隠蔽し全体としての処理スループットを向上することができる．また，継続点によるスレッド実行制御を行うため，継続対象の処理結果受け取りスレッドを特定して一意に実行可能状態にすることができる．これによりスループットの向上に加え応答時間の改善も期待できる．

3.3.3　応　答　性

3.3.3.1　プリエンプション

割り込み処理スレッドなど優先度が高いスレッドが実行可能状態になった場合の対処としては，プリエンプション機構によって優先度の逆転を回避できる．しかし，カーネル内プリエンプションを実現する際には，スレッドの実行中断および再開処理機構が必要である．

一方，スレッド終了処理毎の別スレッドへ切り替えまで，優先度の高いスレッドの実行を遅延させることも考えられる．この場合，別途カーネル内プリエンプション機構がなくとも，スレッドの粒度程度の粗さでカーネル内プリエンプションに相当する処理が実現できる (準プリエンプションと呼ぶ)．スレッドを細粒度化し，時間閾値に比べてスレッド実行時間が短いように設計することで，妥当な応答性を確保できると考える．この準プリエンプションによる効果は，カーネル内スレッドとユーザレベルスレッドという違いはあるものの，文献 4) に述べられた効果と同様である．

3.3.3.2　割り込み処理に続く処理

スレッド化されたカーネルでは，割り込み処理をスレッドとして実行し，割り込み処理スレッド終了時にスケジューリングの機会が与えられる．割り込み処理スレッドに続く一連の処理を行うスレッドの優先度を高く設定しておくことで，スケジューリングの際に選択される可能性を高めることができる．これにより，割り込み処理復帰後には割り込まれた処理を続けて行う方式に比べて，割り込み処理に続く一連の処理の応答性の向上が期待できる．

3.3.4　クリティカルセクションの局所化

カーネル処理にはクリティカルセクションがある．この対処として，カーネル実行パス全体をロックした場合，優先度逆転が起こり，カーネル処理の実時間性が実現できない．このため，リアルタイム OS では，カーネル実行パスのうち，クリティカルセクションを細かく区切り，カーネル実行パスの切り替えを可能にしている．しかし，このような設計および実装は難しく，動作検証も非常に困難である．

一方，本 Zero-wait スレッド実行方式では，割り込み処理をスレッドとして実行する．また，自己継続処理により，スレッドの排他実行制御が可能である．したがって，自己継続処理を適切に利用することで，クリティカルセクションをスレッドとして陽に記述し，局所化して実現することができる．なお，継続処理をソフトウェアで実現する際には，継続処理自体がクリティカルセクションになり，それ自体にはコストがかかる．しかし，クリティカルセクションのロックに相当する処理を継続処理に局所化することができる．

4. 入出力処理における Zero-wait スレッド実行方式の効果

本節では，UNIX 系 OS の I/O 処理 (以降 UNIX 方式) と，Zero-wait スレッド実行方式による I/O 処理 (以降 Zero-wait 方式) を比較する．ここでは，カーネル内の処理を想定しており，例として read() システムコールなどによりカーネルに突入したあとの処理について論じる．

4.1　入出力処理

4.1.1　UNIX 系 OS の I/O 処理

UNIX 方式では，基本的に**図 3** に示すような処理を行っており，以下にその説明を行う．処理を細分化し通番をふり，Zero-wait 方式の各処理との対応付けを行えるようにしている☆．

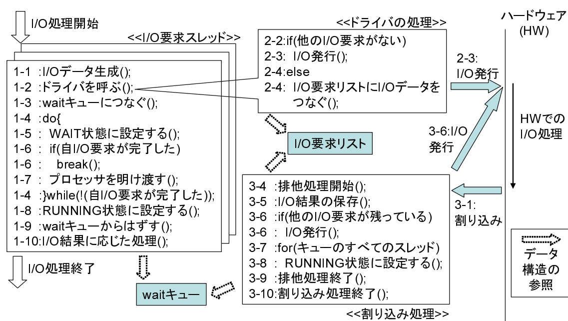

図 3　UNIX 系 OS の I/O 処理

4.1.1.1　I/O 要求スレッド

I/O 要求スレッドでは，まず，I/O に必要なデータ

☆ なお，UNIX 方式における「スレッド」は，スレッド化されていないカーネルの場合はユーザのスレッドコンテキスト上で実行されるカーネルの処理を，スレッド化されたカーネルの場合はカーネルスレッドを意味する．

を生成する（1-1）．そして，ドライバの処理として，他にI/O 要求がなければ，HW に対してI/O を発行する（2-2，2-3）．他にI/O 要求があれば，I/O 要求リストにI/O データをつなぐ（2-4）．これは，一般的に，HW で実行中のI/O が完了しないと次のI/O を発行できないからである．

ドライバの処理から復帰すると，自分自身を wait キューにつなぐ（1-3）．そして，事象待ち状態になり，自分が出したI/O が完了するまでプロセッサを明け渡す処理を繰り返す（1-4，1-5，1-7）．これは，キューに事象待ちのスレッドをつないでおくことで，事象待ち解除処理が容易になるためである．

自分が出したI/O が完了すると，wait キューから自分自身を取り除き，I/O 結果に応じた処理を行う（1-9，1-10）．

なお，プロセッサを明け渡す処理の前に自分が出したI/O が完了しているかを判断する処理や，それに伴い do ループから抜けたあとに，RUNNING 状態に設定する処理を行うのは（1-6，1-8），スレッドの状態設定を厳密に行う必要があるからである．

4.1.1.2　割り込み処理

割り込み処理では，まず，割り込み処理を排他的に行うために割り込みレベルの設定を行う（3-4）．次に，I/O 結果の保存を行う（3-5）．I/O 要求リストに他のI/O 要求が残っている場合は HW に対してI/O を発行する（3-6）．続いて，wait キューにつながっているすべてのスレッドに対して事象待ち解除処理を行う（3-7，3-8）．最後に割り込みレベルを元に戻し，割り込み処理を終了する（3-9，3-10）．

4.1.2　Zero-wait スレッド実行による I/O 処理

Zero-wait スレッド実行方式によるI/O 処理の概要を**図 4** に示す．以下に各処理を説明する．

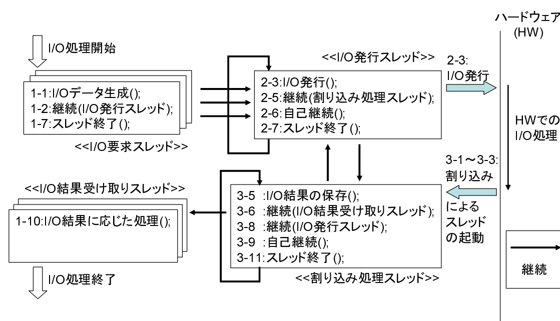

図 4　Zero-wait スレッド実行方式による I/O 処理

4.1.2.1　I/O 要求スレッド

I/O 要求スレッドでは，まず，I/O に必要なデータを生成する（1-1）．そして，I/O 発行スレッドに対して継続処理を行う（1-2）．継続処理では，I/O データの情報と I/O 結果受け取りスレッドの情報を I/O 発行スレッドに渡す．

4.1.2.2　I/O 発行スレッド

I/O 発行スレッドでは，HW に対して I/O 発行を行う（2-3）．また，割り込み処理スレッドに対して継続処理を行い，I/O データの情報と I/O 結果受け取りスレッドの情報を渡す（2-5）．さらに自己継続処理を行い，次の I/O 発行に備える（2-6）．I/O 発行スレッドの実行には，割り込み処理スレッドからの継続が必要である．これにより，HW で実行中の I/O が完了しないと次の I/O が発行されないことが保証される☆．

また，I/O を発行する処理を 1 つのスレッドとして分離するのは，I/O を発行する処理を排他的に実行できるように I/O 発行スレッドとして分離するためである．たとえば，複数のプロセスから同時に I/O 要求が出された場合でも排他的に I/O 処理を実行できる．

4.1.2.3　割り込み処理スレッド

割り込み処理スレッドでは，I/O 結果の保存を行った後，I/O 結果受け取りスレッドに対して継続処理を行い，I/O 結果が保存された I/O データの情報を渡す（3-5，3-6）．また，I/O 発行スレッドに対して継続処理を行い，次の I/O 発行許可を与える（3-8）．さらに，自己継続処理を行い，スレッドの排他的な実行を保証する（3-9）．これは，同じ種類の割り込み処理を同時に実行しないことを保証するためである．

4.1.2.4　I/O 結果受け取りスレッド

I/O 結果受け取りスレッドでは，I/O 結果に応じた処理を行う（1-10）．

4.2　I/O 完了時の起床処理と直接起動の比較

本節では，I/O 処理における Zero-wait スレッド実行方式の効果について議論する．主に，UNIX 方式でのキューを用いた起床処理と，Zero-wait スレッド方式の継続点による直接起動に関する比較を行う．

4.2.1　UNIX 系 OS の起床処理

UNIX 方式ではキューを用いてI/O 要求スレッドを一括管理しているため，キューを見るだけで当該I/O 処理に関与しているすべてのスレッドの情報を得ることができる．しかしながら，割り込み処理の中で，キューにつながっているすべてのスレッドに対して起床処理を行わなければならない．キューにつながっているすべてのスレッドが起床されるため，I/O 要求スレッドは，自分の I/O 要求が完了したかを確認する必要がある．さらに自分の I/O 要求の完了でなかった場合は再びプロセッサを明け渡す必要がある．このオーバヘッドは，同時に多くの I/O 要求が出され，キューに多数のスレッドがつながるに従い顕著になってくる．そのため近年では，既存の枠組みの中で，複数のキューに I/O 要求スレッドを分散させるといった改善もなされている．たとえば，最近の Linux kernel ではディ

☆ 1 回目の I/O 発行を可能にするため，割り込み処理スレッドからの継続は初期値では与えられているものとする．

スク I/O 時のページの読み込み処理において，ハッシュ関数を用いることにより，I/O 要求スレッドを分散している．

しかしながら，このような改善を行っても，自分の I/O 要求が完了したかを確認する必要があり，自分の I/O 要求の完了ではなかった場合は再びプロセッサを明け渡す必要がある．さらに，一括管理による効果が減少してしまう．

非同期 I/O を用いることにより，不必要なスレッドまで起床してしまうことはない．しかしながら，非同期 I/O では I/O 要求者が自分で I/O 結果を別途確認する処理を行う必要がある．そのため，もし I/O が完了していなかったら再び確認する処理を行わなければならない．この処理は，システムコールを介してユーザプログラムが行う必要があり，システムコールの呼び出しと復帰の処理だけオーバヘッドが大きいと考えられる．

4.2.2 Zero-wait 方式による直接起動

Zero-wait 方式の場合，当該 I/O 処理に関与しているすべてのスレッドの情報を一度に得ることが難しい．また，I/O 結果受け取りスレッドの情報を逐一受け渡しながら処理を行う必要がある．しかしながら，UNIX 方式のように，キューにつながっているすべてのスレッドを起床する処理が必要なくなる（3-7）．したがって，同時に多くの I/O 要求が出されたとしても，その影響を受けずに処理を行うことができると考えられる．さらに，図 3 と図 4 の比較からわかるように，UNIX 方式での条件判定やループ処理などが必要なくなる（1-4，1-6，2-2，2-4，3-6，3-7）．これは，継続処理を活用するすることで，条件判定やループ処理に相当する処理を継続処理で実現できることを意味する．スレッドに処理待ち状態がないため，4.1.1.1 小節で述べた，UNIX 方式のようにスレッドの状態設定のタイミングを厳密に考える必要もなくなる．

4.2.3 切り替え処理回数の比較

I/O 要求が n 個ほぼ同時に発生したときに，すべての I/O 要求が完了するまでの切り替えの処理回数について比較する．

UNIX 方式では，はじめに，通番 1-7 の処理（プロセッサの明け渡し処理）で n 回切り替えが起こる．次に，割り込み処理により，wait キューにつながっているすべてのスレッドが起床される．これにより n 回の切り替えが起こる．しかし，通番 1-4 の do ループを抜け出すことができるスレッドは 1 つである．したがって，残りの $n-1$ 個のスレッドは，通番 1-7 で再びプロセッサを明け渡すことになり，$n-1$ 回の切り替えが起こる．次の割り込み処理では，$n-1$ 個のスレッドが起床され，$n-2$ 個のスレッドが再びプロセッサを明け渡すことになる．したがって，切り替え回数の合計を計算すると，次のようになる．

$$\begin{aligned} & n + (n + (n-1)) + \cdots + (1+0) \\ & = n + \sum_{k=1}^{n} (k + (k-1)) = n^2 + n \end{aligned}$$

4.2.1 小節で述べたように，ハッシュによりスレッドの分散を行った場合は次のようになる．ただし，wait キューの個数を h とする．

$$\begin{aligned} & n + \frac{(n + (n-1))}{h} + \cdots + \frac{(1+0)}{h} \\ & = n + \sum_{k=1}^{n} \frac{(k + (k-1))}{h} = \frac{n^2}{h} + n \end{aligned}$$

Zero-wait スレッド方式では，はじめに，通番 1-7 の処理（I/O 要求スレッド終了処理）で n 回切り替えが起こる．次に，I/O 発行スレッドで最初の I/O を発行した後，通番 2-7 の処理（I/O 発行スレッド終了処理）で 1 回の切り替えが起こる．続いて，割り込み処理スレッドが I/O 結果受け取りスレッドに継続し，通番 3-11 の処理（割り込み処理スレッド終了処理）で 1 回のスレッド切り替えが起こる．次の I/O 発行時にも同様に，通番 2-7 の処理と通番 3-11 の処理の 2 回のスレッド切り替えが起こる．したがって，切り替え回数の合計を計算すると，次のようになる．

$$n + (1+1) + \cdots + (1+1) = n + \sum_{k=1}^{n} (1+1) = 3n$$

以上の考察により，I/O 要求数 n に対する切り替え回数のオーダーは，UNIX 方式では $O(n^2)$，Zero-wait スレッド方式では $O(n)$ になる☆．

たとえば Web サーバプログラムの場合は，多くの I/O 要求が同時に発生する可能性が高い．このため，I/O 要求数 n が大きくなる傾向がある．その結果，$O(n^2)$ と $O(n)$ の差は大きくなるため，Zero-wait スレッド方式は効果があると考えられる．

4.2.4 切り替え処理の削減効果

本項では，Zero-wait スレッドの継続点による直接起動を実現した場合の，切り替え処理削減効果の予備評価について述べる．Zero-wait スレッド方式によるカーネルはまだ存在していないため，以下に述べる Linux スレッド での直接起床方式の測定データ，および Zero-wait スレッド方式を実現した場合の基本機構の予想コストにもとづいて予備評価を行った．Zero-wait スレッド方式の実行機構のコストは文献 2) での測定値を参考にした。直接起床方式の I/O 処理の流れを**図 5** に示し，以下にその説明を行う．

直接起床方式特有の処理には，通番 4-1 および 4-2 を振っている．直接起床方式では，UNIX 方式のよう

☆ なお，本項で考察した切り替えの回数は，I/O 処理に関わっていないスレッドやマルチプロセッサでの並列処理といったことの影響は考慮していない近似的なものである．

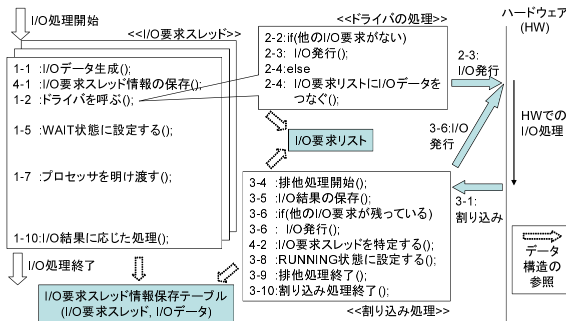

図 5 直接起床方式の I/O 処理の流れ

に I/O 完了待ちスレッドを wait キューで管理しない．その代わりに I/O 要求スレッドが，自スレッドと自 I/O データの情報の組を，I/O 要求スレッド情報保存テーブルに保持する（4-1）．これにより，割り込み処理の中で，当該 I/O データを発行したスレッドを一意に指定し（4-2），起床することができる．

直接起床方式では，コンテキスト切り替え処理が UNIX 方式と同じものになる．一方で，Zero-wait スレッド方式のスレッド切り替えは UNIX 方式のコンテキスト切り替えとは異なるので，直接的な比較は難しい．しかしながら，切り替えの回数に関しては，Zero-wait スレッド方式と直接起床方式には傾向が同じであると考えることができる．また，Zero-wait スレッド方式の方が UNIX 方式よりスレッドの粒度が小さく一回あたりのコストも小さいと予想され，この比較は実際より効果が控え目の結果になると考える．

実装および測定は以下の方針で行う．今回はコンテキスト切り替えのコストの差に注目して測定を行う．まず，実際の I/O 処理にかかわる部分（1-2 およびドライバの処理すべて，1-10，3-1，3-4，3-6，3-9，3-10）を除いたものを抜き出す．その処理の流れを**図 6** および**図 7** に示す．次に，抜き出した処理を直接呼ぶシステムコール（図 6 中のシステムコール A，B，および図 7 中の C，D）を作成する．続いて，測定用プログラムを作成する．このプログラムは上記のシステムコールを呼ぶ．測定用プログラムの処理の流れを以下に示す．

(1) 子スレッドを指定個生成する．
  (a) 子スレッドは，システムコール A(または C) を発行し，WAIT 状態になり，プロセッサ資源を明け渡す．
(2) 開始時の clock tick を取得する．
(3) （1）で生成した子スレッドの個数分次の処理を繰り返す．
  (a) システムコール B(または D) を発行し，子スレッドをひとつ起床する．
    (i) 子スレッドは，起床されると 10 秒後にそのまま終了する．
(4) 終了時の clock tick を取得する．つまり，子スレッドの終了処理にかかる時間は測定されない．
(5) 開始時と終了時の clock tick の差分から，所要時間を計算する．

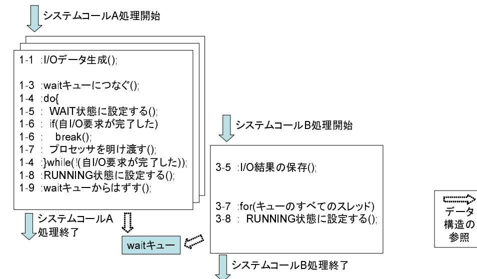

図 6 UNIX 方式の I/O 処理を除いた処理の流れ

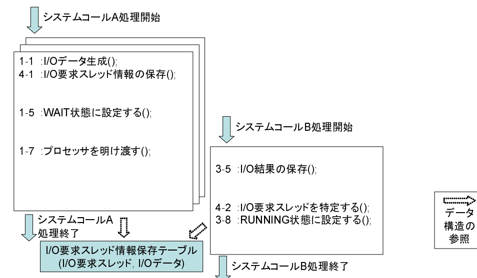

図 7 直接起床方式の I/O 処理を除いた処理の流れ

I/O 発行処理時間および HW での I/O 処理時間は，この比較には含まれないことになる．また，測定用プログラムでは，I/O 要求数 n に対するはじめの n 回の切り替えについては計測しない点にも注意が必要である．ただし，はじめの n 回の切り替えを除いた場合においても，切り替え回数のオーダに影響はない．測定結果を**表 1** に示す．また，グラフにしたものを**図 8** に示す．なお，実装環境および測定環境は CPU が PentiumIII 1GHz (メモリ 512MB)，OS が Red Hat Linux 9 (kernel 2.4.20-8) である．

測定結果より，切り替えのコストにかなりの差が出ていることがわかる．今回は Zero-wait スレッド方式の代替として直接起床方式での比較である．しかし，Zero-wait スレッド方式でも同様の傾向を持つと予想する．また，Zero-wait スレッド方式の方が UNIX 方式よりスレッドの粒度が小さく一回あたりのコストも小さいと予想される．そのため，Zero-wait スレッド方式の方が，切替えのコストが低く，CPU の計算資源をより有効に利用することができると考えられる．

表 1　切り替えコストの比較（単位:マイクロ秒）

スレッド数	1	32	64	96	128	160	192	224	256
UNIX 方式	2	129	863	1896	3565	5166	7937	10572	13531
直接起床方式	2	32	65	104	146	192	242	296	362

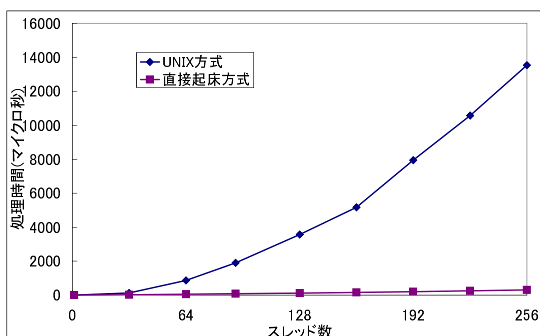

図 8　コンテキスト切り替えコストの比較

5. おわりに

本稿では，まず，Zero-wait スレッドを用いたオペレーティングシステムカーネルの構成を検討した．予想される短所として，継続情報の明示的受渡し，スレッド切り替えオーバヘッド増加，といった点がある．しかしながら利点として，スレッド状態管理の簡潔化，スループットの向上，割り込み処理スレッドに続く処理の応答性向上，準プリエンプション処理，クリティカルセクションの局所化，といった点が期待できる．

本稿では，I/O 処理に焦点をあて，Zero-wait スレッド実行方式の定性的な利害得失について論じ，切り替え回数やコストの比較を行った．I/O 処理に関して予想される欠点としては，UNIX 方式では容易な当該 I/O 処理に関与するスレッド情報の一括管理が困難，といった点があった．しかし I/O 処理に関して期待できる利点として，スレッド処理の簡潔化，I/O 結果受け取りスレッドの直接起動効果によるオーバヘッド削減，といった点がある．切り替え回数のオーダーは，同時並行 I/O 要求数 n に対して，UNIX 系 OS の I/O 処理の場合は $O(n^2)$ であるのに対し，Zero-wait スレッド実行方式による I/O 処理の場合は $O(n)$ であることを示した．さらに，切り替え処理の削減効果の予備評価を行い，同時に多数の I/O 要求が起こった場合には，Zero-wait スレッド実行方式による I/O 処理が有利であることを示した．

今後さらに Zero-wait スレッド実行方式の詳細な検討の具体化をすすめ，評価を行う予定である．

参考文献

1) 雨宮 聡史, 松崎 隆哲, 雨宮 真人．排他実行マルチスレッド実行モデルに基づくオンチップ・マルチプロセッサの設計．**情処研報**,*ARC-155*, pp. 51–56, 2003.

2) 飯尾 賢太郎, 日下部 茂, 谷口 秀夫, 雨宮 真人．性能モニタリングカウンタによる一括システムコール機構の評価．**情処研報**,*OS-97*, pp. 49–56, 2004.

3) 日下部 茂, 富安 洋史, 村上 和彰, 谷口 秀夫, 雨宮 真人．並列分散オペレーティングシステム CEFOS(Communication-Execution Fusion OS)．**信学技報** *CPSY99-50*, pp. 25–32, 1999.

4) 棚林 拓也, 中山 大士, 日下部 茂, 谷口 秀夫, 雨宮 真人．並列分散オペレーティングシステム CEFOS における準プリエンプション機能．**情処研報**,*OS-89*, pp. 55–62, 2002.

5) 福冨 和弘, 乃村 能成, 日下部 茂, 谷口 秀夫, 雨宮 真人．マルチスレッド実行機構を考慮したプログラム実行制御法．**情処研報**,*OS-96*, pp. 135–140, 2004.

6) Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, pp. 1–6, 1990.

7) Doug Burger and et al. Scaling to the end of silicon with edge architectures. *IEEE Computer*, 37(7):44–55, 2004.

8) Jack B. Dennis and Guang R. Gao. *Multithreaded computer architecture: A summary of the state of the art*, chapter Multithreaded Architectures: Principles, Projects, and Issues, pp. 1–74. Kluwer academic, 1994.

9) Sean Ryan, José N. Anaral, Guang Gao, Zachary Ruiz, Andres Marquez, and Kevin Theobald. Coping with very high latencies in petaflop computer systems. In *Proceedings of second International Symposium on High Performance Computing, ISHPC'99*, pp. 71–82, 1999.

10) Theo Ungerer, Borut Robič, and Jurij Šilc. A survey of processors with explicit multithreading. *ACM Computing Surveys*, 35(1):29–63, 2003.