JAIST Repository
https://dspace.jaist.ac.jp/

Title	異種タスク混在型リアルタイム組込みシステムに おけるタスクスケジューリング方式の研究
Author(s)	Ly, Luong Cong
Citation	
Issue Date	2014-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/12007
Rights	
Description	Supervisor: 田中　清史, 情報科学研究科, 修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

修 士 論 文

異種タスク混在型リアルタイム組込みシステムにおけるタスクスケジューリング方式の研究

北陸先端科学技術大学院大学
情報科学研究科情報科学専攻

Ly Cong Luong

2014 年 3 月

修 士 論 文

異種タスク混在型リアルタイム組込みシステムにおけるタスクスケジューリング方式の研究

指導教員　田中 清史 准教授

審査委員主査　田中 清史 准教授
審査委員　井口 寧 教授
審査委員　金子 峰雄 教授

北陸先端科学技術大学院大学
情報科学研究科情報科学専攻

1110751Ly Cong Luong

提出年月: 2014 年 2 月

Copyright © 2014 by Ly Cong Luong

概 要

本稿では、今日のリアルタイム組込みシステムにおけるアプリケーションの多種多様化により、異なる種類・重要性のタスクが混在する状況下でのタスクスケジューリング方式が重要になってきている。例えば、周期的に起動されるハードリアルタイムタスクと、非周期に起動されるソフトリアルタイム（あるいは非リアルタイム）タスクが混在するシステムが増加してきており、それらのタスク群をリアルタイム制約を満たすようにスケジューリングする必要がある。一般的に、周期的なハードタスクはデッドライン制約を満たすことが重視され、ソフト（あるいは非リアルタイム）タスクは平均応答時間が小さいことが求められる。本研究では、周期タスクと非周期タスクの混在タスクセットを対象としたスケジューリングアルゴリズムである Total Bandwidth Server を改良することで、重要度の高い周期タスクのデッドラインに関するスケジューラビリティを確保しつつ、非周期タスクの応答時間を可能な限り小さくするスケジューリング方式を提案することを目的とする。提案スケジューリング方式を実際の ITRON ベースのリアルタイムオペレーティングシステムに組み込んで評価した結果、ハードリアルタイムタスクセットの CPU 使用率が 60 %の場合に、ソフトタスクの応答時間が最大で 29.9%短縮されることを確認した。

目次

第 1 章　はじめに 4
1.1 背景、目的 4
1.2　論文の構成 5
第 2 章　関連研究 6
2.1　スケジューラの概要 [6]..... 6
2.2　ITRON 概要 7
2.3 Earliest deadline first(EDF)の概要..... 11
2.4　非周期タスクサーバアルゴリズム..... 13
2.5　Total Bandwidth Server(TBS)の概要..... 14
2.6　最悪実行時間 16
第 3 章　TBS の改良方法の提案 18
3.1　概要 18
3.2　デッドライン計算について..... 19
3.3 提案方式の適用例..... 21
3.4 スケジューラビリティ..... 23
3.5 実装の複雑さ 23
第 4 章　スケジューラの実装 25
4.1　既存 RTOS について..... 25
4.2　デッドライン算出タイミング..... 25
4.3　実装 26
第 5 章　評価 32
5.1 評価環境 32
5.2 タスクセット 32
5.3 評価結果 38
第 6 章　まとめ 40
6.1　総括 40
6.2　今後の課題 40
謝辞 41
参考文献 42

表目次

2.1 周期タスクセット 12
2.2 EDF でスケジュールされるタスクセットの例 12
2.3 タスクセット 15
3.1 デッドライン計算のためのパラメータ 19
3.2 タスクセット 21
5.1 Up=0.6 の周期タスクセット 1 33
5.2 Up=0.6 の周期タスクセット 2 33
5.3 Up=0.6 の周期タスクセット 3 33
5.4 Up=0.6 の周期タスクセット 4 33
5.5 Up=0.6 の周期タスクセット 5 33
5.6 Up=0.7 の周期タスクセット 1 34
5.7 Up=0.7 の周期タスクセット 2 34
5.8 Up=0.7 の周期タスクセット 3 34
5.9 Up=0.7 の周期タスクセット 4 34
5.10 Up=0.7 の周期タスクセット 5 34
5.11 Up=0.8 の周期タスクセット 1 35
5.12 Up=0.8 の周期タスクセット 2 35
5.13 Up=0.8 の周期タスクセット 3 35
5.14 Up=0.8 の周期タスクセット 4 35
5.15 Up=0.8 の周期タスクセット 5 35
5.16 Up=0.9 の周期タスクセット 1 36
5.17 Up=0.9 の周期タスクセット 2 36
5.18 Up=0.9 の周期タスクセット 3 36
5.19 Up=0.9 の周期タスクセット 4 36
5.20 Up=0.9 の周期タスクセット 5 36
5.21 非周期タスクの情報 1 37
5.22 非周期タスクの情報 2 37
5.23 非周期タスクの情報 3 37
5.24 非周期タスクの情報 4 37
5.25 非周期タスクの情報 5 37
5.26 非周期タスクの応答時間 38
5.27 従来の TBS に対する改善率 38

図目次

2.1 μITRON でのシステム構造 8
2.2 μITRON でのタスク状態遷移 10
2.3 EDF の動作タイミング図 13
2.4 TBS での動作タイミング図 16
3.1 TBS 21
3.2 提案方式 21
3.3 提案方式でのデッドライン再計算タイミング 22
4.1 デッドライン計算 26
5.1 非周期タスクの応答時間 39

第1章 はじめに

1.1 背景、目的

組込みシステムは身の回りに存在している。家庭内にあるDVD Player、Game機器、エアコン、冷蔵庫などは組込みシステムである。車のエンジン制御装置、ブレーキ制御装置、ネットワークルータ、プリンタも組込みシステムである。リアルタイムオペレーティングシステム（Real-Time Operating System: RTOS）は今日の多くの組込みシステムの要として、アプリケーションを構築する際のプラットフォームを提供する。小規模の（コード量が少ない）アプリケーションで構成される組込みシステムではRTOSを必要としない場合があるが、中規模～大規模のソフトウェアを要する組込みシステムの開発ではRTOSは不可欠であると言われている。

RTOS の中核はリアルタイムカーネルである。スケジューラはカーネルに含まれており、どのタスクをいつ実行するかは、そのスケジューラが実現するスケジューリングアルゴリズムによって決定される。従来から様々なリアルタイムスケジューリング方式が提案されてきたが、それらの主な目的はリアルタイム性を向上させることである [3]。

リアルタイム性とは、単に高速の計算処理や応答時間が短いことではなく、システムが定められた時間条件を満たして動作する性質をいう。多くの組込みシステムは装置の制御要求で定まる時間条件を満たして動作しなければならない。リアルタイム性が求められるシステムがリアルタイムシステムである。

リアルタイムシステムは時間制約を守れなかった場合の許容度により、ハードリアルタイムシステムとソフトリアルタイムシステムに分類できる。ハードリアルタイムシステムでは、タスクの実行がデッドライン内に終了しなかった時（デッドラインミス）、システム全体にとって致命的損害が生じる、といった理由から、デッドライン内での終了が保証されなければならないシステムである。タスクは実行毎に実行時間が変動するため、ハードリアルタイムシステムでは、その時間制約厳守の必要性から、タスクの実行に関して最悪実行時間（Worst-Case Execution Time: WCET）を想定して保障することになる。一方、ソフトリアルタイムシステムではデッドラインミスが起こっても、システム全体に致命的なダメージを与えることはない。また、非リアルタイムシステムは、デッドラインを持たないシステムである。したがって、ソフトリアルタイムあるいは非リアルタイムシステムでは、タスクの実行時間としてWCETを想定する必要性はないと言える。

一つのリアルタイムシステムの中にもハードリアルタイムタスク（原則として周期タスク）と、ソフトリアルタイムまたは非リアルタイムタスク（周期タスクあるいは非周期タスク）が混在することがある。重要度の高いハードリアルタイムタスクのデッドラインに

関するスケジューラビリティを確保しつつ、ソフトあるいは非リアルタイムタスクの応答時間を可能な限り小さくできれば、システム全体のレイテンシ（latency)が小さくなってシステム全体の性能が改善する。

本研究ではRTOS内のスケジューラのスケジューリングアルゴリズムを研究対象とし、周期（ハード）タスクのリアルタイム性を確保しつつ非周期タスクの応答時間を短縮する方式を提案・評価することを目的とする。提案する方式は、繰り返し実行されるタスクの実行時間が実行毎に変動することに着目し、非周期タスクに対して早期終了を予測してスケジュールすることにより、応答時間を短縮するものである。また、非周期タスクの応答時間が短縮しつつ、周期タスクの時間制約を満たすことが保証されることが提案方式の特長である。

1.2　論文の構成

本論文は以下の構成をとる。

第2章　関連研究

RTOSのスケジューラについての概念と用語を説明し、続いて提案方式の実装対象となるITRON環境について概略する。更に周期タスクの基本的なスケジューリング法であるEarliest Deadline First（EDF）アルゴリズムと、周期タスクと非周期タスクの混在タスクセットを対象としたスケジューリングアルゴリズムの一つであるTotal Bandwidth Server（TBS）を紹介する。

第3章　TBSの改善方法の提案

TBSを改良し、タスクの実行時間の変動を利用してタスク実行の早期終了を予測する方法と、予測時間を利用して非周期タスクの応答時間を更に短縮する方法を提案し、定式化する。

第4章　スケジューラの実装

提案するTBSの改良方式を既存のITRONベースのリアルタイムオペレーティングシステムに実装する方法を述べる。

第5章　評価

実プログラムからなるタスクセット（周期・非周期混在）を用意し、従来のTBSと提案するTBSの改良方式をシミュレーションにより比較評価し、その結果を議論する。

第6章　まとめ

研究結果のまとめと今後の課題について述べる。

第 2 章　関連研究

本章では，RTOS のスケジューラについて各概念と用語を説明し，本研究で提案されるスケジューリング方式の実装対象となる ITRON 環境について概略する。続いて，提案スケジューリング方式に関連する Earliest deadline first(EDF)[1]と Total bandwidth server(TBS)[2]を紹介する.

2.1　スケジューラの概要 [6]

スケジューラは、オペレーティングシステムのカーネルにある。スケジューラはどのタスクがいつ実行されるかを決定するためにスケジューリングアルゴリズムを実行する。スケジューラがどのように機能するかを理解するために、次のトピックについて説明する。

・スケジュール可能なエンティティ
・マルチタスク
・コンテキストの切り替え
・スケジューリングアルゴリズム

2.1.1 スケジュール可能なエンティティ

スケジュール可能なエンティティは、スケジューラのスケジューリングアルゴリズムに基づいて、システム上で実行時間（実行のためのハードウェア資源）を分け合うカーネルオブジェクトである。タスクやプロセスは、スケジュール可能なエンティティの例である。タスクやプロセスは、独立した実行スレッドであり、一連の命令を持つ。本研究ではスケジュール可能なエンティティとしてタスクを対象とする。

2.1.2 マルチタスク

マルチタスクは複数のスケジュール可能なエンティティを切り替えて実行する機能である。この機能は、タスク実行に内在する各種待ち時間を隠蔽し、効率の良いシステム実行を実現するために有効である。マルチタスクはスケジューラによって実現される一つの機能である。

2.1.3 コンテキストの切り替え

各タスクが自身のコンテキストを持つ。コンテキストとは、タスク実行の途中状態であり、実際には各種 CPU レジスタの値が相当する。タスク実行を中断する際は後に再開できるように、当該タスクのコンテキストを退避し、切り替え先のタスクの（退避されていた）コンテキストを復帰して実行を再開する。これをコンテキスト切り替えと呼ぶ。

新しいタスクが作成されるたびに、カーネルは関連付けられたタスク制御ブロック（Task Control Block: TCB）を作成し、管理する。 TCB はシステムデータ構造であり、カーネルが TCB を使用してタスク固有の情報を保持する。カーネルが特定のタスクについて知るために、必要となるすべての情報が TCB に含まれている。タスクが実行されている際にタスクのコンテキストは動的に変化する。この動的なコンテキストが TCB に保持される。タスクが実行されていない場合は、コンテキストは、TCB 内で凍結される。タスクの次回実行時に復元する。

2.1.4 スケジューリングアルゴリズム　[7]

スケジューラは、スケジューリングアルゴリズムによってどのタスクを実行するかを決定する。スケジューリングが発生する可能性がある様々な状況が存在する。下記の状況が発生した場合に次に実行されるタスクを選択する必要があり、スケジューラが実行される。

・タスクの実行が終了したとき
・タスクがＩ／Ｏ処理またはセマフォでブロックされたとき

論理的には必須ではないが、以下の通り、スケジューリングが通常行われる他の 3 つの状況がある。

・新しいタスクが作成されるとき。
・Ｉ／Ｏ割り込みが発生したとき。
・クロック割り込みが発生したとき。

2.2　ITRON 概要

2.2.1 μITRON の概要

ITRON は、TRON プロジェクトが策定・維持している組込み OS・リアルタイム OS カーネルの仕様である。μITRON は ITRON のサブセット的な仕様である。各社の実装により動作に多少の違いがあるが、μITRON 仕様は公開されており、システムコールや API 名称などが決められている。

2.2.2 μITRON でのシステム構造

μITRON ではカーネルとアプリケーションは同一の空間で動作している。カーネルは直接、ハードウェア資源を利用できる。また、アプリケーションから直接デバイスドライバを呼び出せる。以下の図 2.1 はμITRON でのシステム構造を示す。

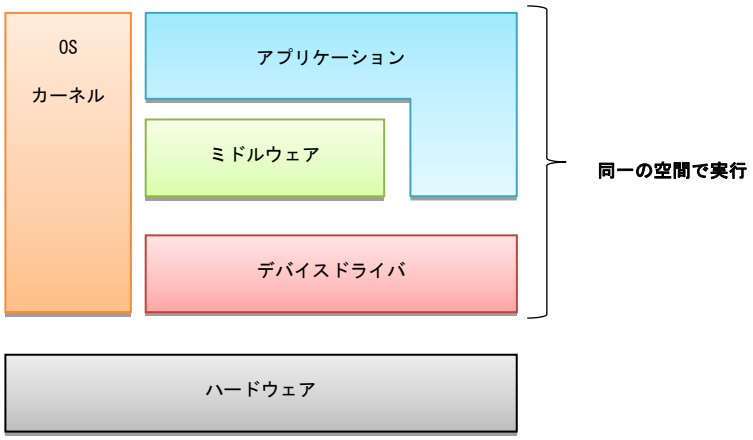

図 2.1 μITRON でのシステム構造

2.2.3 μITRON カーネルの基本機能[5]

2.2.3.1 タスク管理機能

タスク管理機能はタスクの状態を直接的に操作/参照するための機能である。タスクを生成/削除する機能、タスクを起動/終了する機能、タスクに対する起動要求をキャンセルする機能、タスクの優先度を変更する機能、タスクの状態を参照する機能が含まれる。

2.2.3.2 タスク付属同期機能

タスク付属同期機能は、タスクの状態を直接的に操作することによって同期を行うための機能である。タスクを起床待ちにする機能とそこから起床する機能、タスクの起床要求をキャンセルする機能、タスクの待ち状態を強制解除する機能、タスクを強制待ち状態へ移行する機能とそこから再開する機能、自タスクの実行を遅延する機能が含まれる。

2.2.3.3 タスク例外処理機能

タスク例外処理機能はタスクに発生した例外事象の処理を、タスクのコンテキストで行うための機能である。タスク例外処理ルーチンを定義する機能、タスク例外処理を要求する機能、タスク例外処理を禁止/許可する機能、タスク例外処理に関する状態を参照する機能が含まれる。

2.2.3.4 同期・通信機能

同期・通信機能はタスクとは独立したオブジェクトにより、タスク間の同期・通信を行うための機能である。セマフォ、イベントフラグ、データキュー、メールボックスの各機能が含まれる。

2.2.3.5 拡張同期・通信機能

拡張同期・通信機能はタスクとは独立したオブクジェクトにより、タスク間の高度な同期・通信を行うための機能である。ミューテックス、メッセージバッファ、ランデブの各機能が含まれる。

2.2.3.6 メモリプール管理機能

メモリプール管理機能はソフトウェアによって動的なメモリ管理を行うための機能である。固定長メモリプール、可変長メモリプールの各機能が含まれる。

2.2.3.7 時間管理機能

時間管理機能は時間に依存した処理を行うための機能である。システム時刻管理、周期ハンドラ、アラームハンドラ、オーバランハンドラの各機能が含まれる。

2.2.3.8 システム状態管理機能

システム状態管理機能はシステムの状態を変更/参照するための機能である。タスクの優先順位を回転する機能、実行状態のタスク ID の参照する機能、CPU ロック状態へ移行/解除する機能、タスクディスパッチを禁止/解除する機能、コンテキストやシステム状態を参照する機能が含まれる。

2.2.3.9 割り込み管理機能

割り込み管理機能は外部割込みによって起動される割り込みハンドラおよび割り込みサービスルーチンを管理するための機能である。割り込みハンドラを定義する機能、割り込みサービスルーチンを生成/削除する機能、割り込みサービスルーチンの状態を参照する機能、割り込みを禁止/許可する機能、割り込みマスクを変更/参照する機能が含まれる。

2.2.3.10 サービスコール管理機能

サービスコール管理機能は拡張サービスコールの定義と呼びだしを行うための機能である。拡張サービスコールを呼びだすための機能は、標準のサービスコールを呼び出すために用いることもできる。

2.2.3.11 システム構成管理機能

システム構成管理機能にはCPU例外ハンドラを定義する機能、システムのコンフィギュレーション情報やバージョン情報を参照する機能、初期化ルーチンを定義する機能が含まれる。初期化ルーチンは、システム初期化時に実行されるルーチンである。

2.2.4 μITRONにおけるタスク状態遷移

ITRONのタスク状態は基本的に以下の4つに分けられている。

(1) 実行状態：現在そのタスクを実行中であるという状態。

(2) 実行可能状態：タスク側の実行の準備は整っているが、そのタスクよりも優先度が高いタスクが実行中であるため、そのタスクの実行はなされていないという状態。

(3) 待ち状態:タスクの実行を継続できる条件が整わないため、実行が止まっている状態。

(4) 休止状態。タスクがまだ起動されていない状態、または終了後の状態。

ITRONのタスク状態と、状態間の遷移を図2.2に示す。

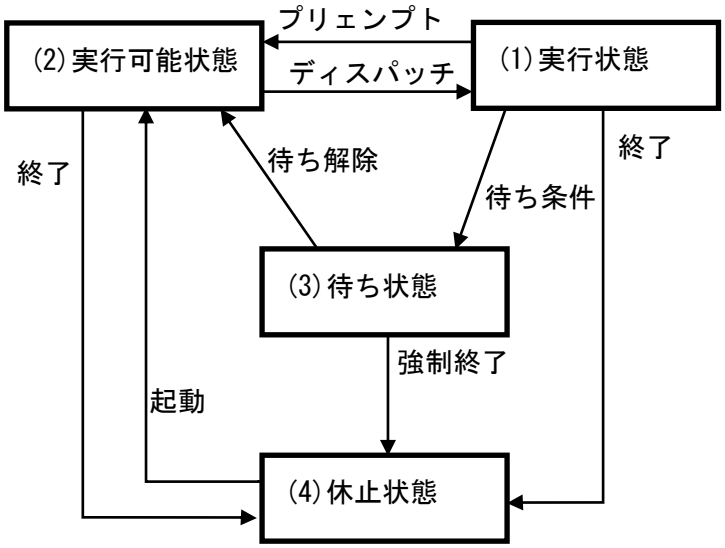

図2.2　μITRONでのタスク状態遷移

2.3 Earliest deadline first(EDF)の概要

2.3.1 Earliest Deadline First(EDF)とは [1]

EDFはRTOSで使用される動的スケジューリングアルゴリズムの一種である。周期タスク、非周期タスクの両方に適用可能である。（ただし、タスク同士は依存やリソース競合が無く、独立であるとする。）スケジューリングイベント（タスク実行終了、新規タスク生成）などが発生すると、動作可能な（実行可能状態の）タスクからなるレディキューを探索し、最もデッドラインが近いタスクを選んで次に実行すべきものとしてスケジュールする。（または、タスクが実行可能状態になる際、レディキューがデッドラインに関して昇順にソートされた順になるようにタスクを挿入し、スケジューリング実行の際はレディキューの先頭タスクを選択する。）なお、本方式はプリエンプションを仮定する。すなわち、あるタスクの実行中に、より高い優先度のタスクの実行要求が発生した場合は、高優先度タスクへの実行切り替え（コンテキストの切り替え）が行われる。

2.3.2 EDFのスケジューラビリティについて

EDFは周期タスクセットに対し、CPU使用率の合計が100%を超えない限り全てのタスクのデッドラインを守ることを保証できる。すなわち、EDFは全周期タスクのCPU使用率の合計をUpとした場合、Up <= 1の条件を満たすシステムでは、全てのタスクのデッドラインを守ることを保障する。

全周期タスクのCPU使用率Upは下記の式によって算出される。

$$U_p = \sum_{i=1..N} \frac{C_i}{T_i}$$

U_p： CPU使用率

T_i： タスクｉの周期

C_i： タスクｉの実行時間

例として、表 2.1 の周期タスクセットに対して CPU 使用率 Up を算出する。
各タスクの使用率の合計を求めることにより、

U_p = 2/10 + 1/8 + 3/15 = 0.2 + 0.125 + 0.2 = 52.5%

となる。

表 2.1 周期タスクセット

タスク	実行時間	周期
1	2	10
2	1	8
3	3	15

2.3.3 EDF の動作説明

表 2.2 の非周期タスクセットで EDF の動作を説明する。このタスクセットを EDF に従ってスケジュールした例を図 2.3.3 に示す。時刻 0 の時、T1 のデッドラインが T2 より近いため T1 が実行され、時刻 1 になったとき T1 の実行が完了し、T2 が実行状態に遷移する。時刻 2 の時、T3 の起動要求が発生し T3 のデッドラインが T2 より近いため、T2 の実行が中断されて T3 が実行状態になる。時刻 3 の時に T4 の起動要求が発生するが、T4 のデッドラインが T3 より遅いので T3 の実行が継続され、時刻 4 に実行完了する。その時に Ready キューに存在するタスクは T2 と T4 である。T2 のデッドラインが T4 より近いため、スケジューラが T2 を選んで実行する。時刻 5 になったとき T2 の実行が完了し、T4 が実行状態になる。時刻 6 の時、T5 の起動要求が発生し、T5 のデッドラインが T4 より近いため、T4 の実行が中断されて T5 が実行状態になる。そして、T5 の実行が時刻 8 に完了して T4 が実行状態になり、時刻 9 に完了する。この例では、全てのタスクのデッドラインが守られていることがわかる。

表 2.2　EDF でスケジュールされるタスクセットの例

タスク名	到着時間	実行時間	デッドライン
T1	0	1	2
T2	0	2	5
T3	2	2	4
T4	3	2	10
T5	6	2	9

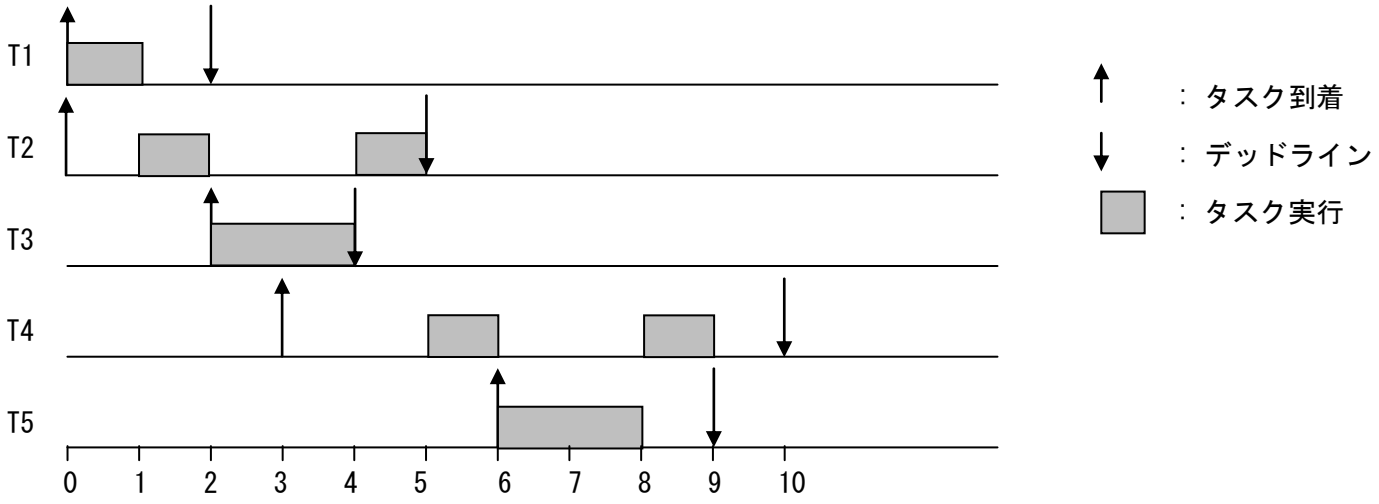

図 2.3　EDF の動作タイミング図

2.4　非周期タスクサーバアルゴリズム

前節の EDF は周期タスクセットに対してスケジューラビリティ解析を可能とする方式であった。すなわち、非周期タスクが存在した場合、使用率によってスケジューラビリティを保証することが不可能である。

周期タスクと非周期タスクが混在するタスクセットに対して、スケジューラビリティを保証するアルゴリズムがいくつか存在する。それらは Rate Monotonic 法[1]をベースとする固定優先度サーバと、EDF をベースとする動的優先度サーバに分類される。固定優先度サーバの例として、Deferrable Server [10]、Priority Exchange [10]、Sporadic Server [11]、Slack Stealing [12]などがあり、これらは RM をベースとするため、高優先度タスクのジッタや応答時間を小さく保つ特徴がある。一方、動的優先度サーバの例として、Dynamic Priority Exchange [2]、 Dynamic Sporadic Server [2]、Total Bandwidth Server [2]、Earliest Deadline Late Server [2]、Constant Bandwidth Server [13]などがあり、EDF をベースとするため、プロセッサの使用率を 100%まで向上させることができるのが特徴である。

これらの中で、Total Bandwidth Server は非周期タスクの応答時間性能が比較的高く、複雑な実装を必要としないといった特徴があり、実際に実装例が存在する [14]。実用性の高い方式であるため、本研究において基本となるスケジューリング方式として採用する。

2.5 Total Bandwidth Server(TBS)の概要

2.5.1 Total Bandwidth Server(TBS)とは

周期タスク（ハードリアルタイムタスク）と非周期タスク（ソフトリアルタイムあるいは非リアルタイムタスク）を明確に区別してスケジューリングを行う方式がある。その代表的なものの一つに Total Bandwidth Server(TBS)[2] がある。これは周期タスクのスケジューラビリティを確保しつつ、非周期タスクの応答時間を小さくできるスケジューリング手法である。シミュレーション結果により、TBS のパフォーマンスは良好であることが示されており、また実装がシンプルであることが示されているため、TBS は実用的なシステムのための有力なスケジューリング方式の候補である。

TBS において、周期タスクは周期のタイミングと等しいデッドラインを持つが、非周期タスクはソフトリアルタイムタスクであるためデッドラインを持たない。そこで、周期タスクと非周期タスクの全てを EDF によってスケジュールすることを可能とするために、非周期タスクに仮のデッドラインを計算して与える。

非周期タスクの実行を担当するサーバのバンド幅を Us（= 1 - Up）とし、TBS では、非周期タスク k の（仮の）デッドライン dk は以下の式に従って算出される。

$$d_k = \max(r_k,\ d_{k-1}) + c_k/U_s$$

k ： 非周期タスクの到着順番
r_k ： k 番目の非周期タスクの到着時刻
d_{k-1} ： k-1 番目の非周期タスクのデッドライン時刻
c_k ： k 番目の非周期タスクの実行時間
U_s ：非周期タスクの実行を担当するサーバのバンド幅

上記の右辺第一項は、連続する 2 つの（k 番目と k-1 番目の）非周期タスクの実行において、それぞれの与えられたバンド幅が重ならないようにするためのものである。すなわち、k 番目の非周期タスクの起動要求時刻が、k-1 番目の非周期タスクの仮のデッドライン時刻よりも早い場合は、k 番目の非周期タスク実行のためのバンド幅区間の始まり時刻として後者を選択することになる。右辺第二項が、サーバのバンド幅から計算される、当該タスク実行のためのデッドラインの長さである。

2.5.2 TBS の動作説明

表 2.3 のタスクセットを使用して TBS の動作を説明する。タスクセットは 2 つの周期タスク（T1、T2）と、3 つの非周期タスク（T3、T4、T5）からなる。表より、周期タスクのプロセッサ使用率は Up = 3/6 + 2/8 = 0.75 となる。したがって、非周期タスクのプロセッサ使用率の合計を Us = 1 - Up = 0.25 とする。

このタスクセットを TBS によってスケジュールした例を図 2.4.2 に示す。最初の非周期タスク（T3）が t=3 に到着し、デッドラインの計算が行われ、$d_1 = r_1 + C_1/U_s = 3+1/0.25 = 7$ となる。デッドライン d_1 は同じく実行可能状態である T2 のデッドラインよりも近いため、非周期タスク T3 が選ばれて実行される。同様に、非周期タスク T4 が t=9 に到着し、d2 = r2 + C2/Us =17 となる。ここで、t=9 のとき、T2 のデッドライン(=16)のほうが近いため、当該非周期タスクはすぐには実行されない。T2 の当該実行が完了したときに、T4 が実行される。t=12 のとき、T1 の起動要求が発生するが、T4 のデッドライ(=17)が T1 のデッドライン(=18)よりも近いため、T4 の実行を継続し、t=13 で T4 の実行が終了する。続いて、非周期タスク T5 が t=14 に到着し、デッドラインが計算され、$d_3 = \max(r_3, d_2) + C_3/U_s = 21$ となる。このとき、周期タスク T1 のデッドラインのほうが近いため、T1 の当該実行の完了後、t=16 で実行が開始されている。

表 2.3 タスクセット

タスク	周期	実行時間	起動時刻
T1	6	3	周期
T2	8	2	周期
T3	-	1	3
T4	-	2	9
T5	-	1	14

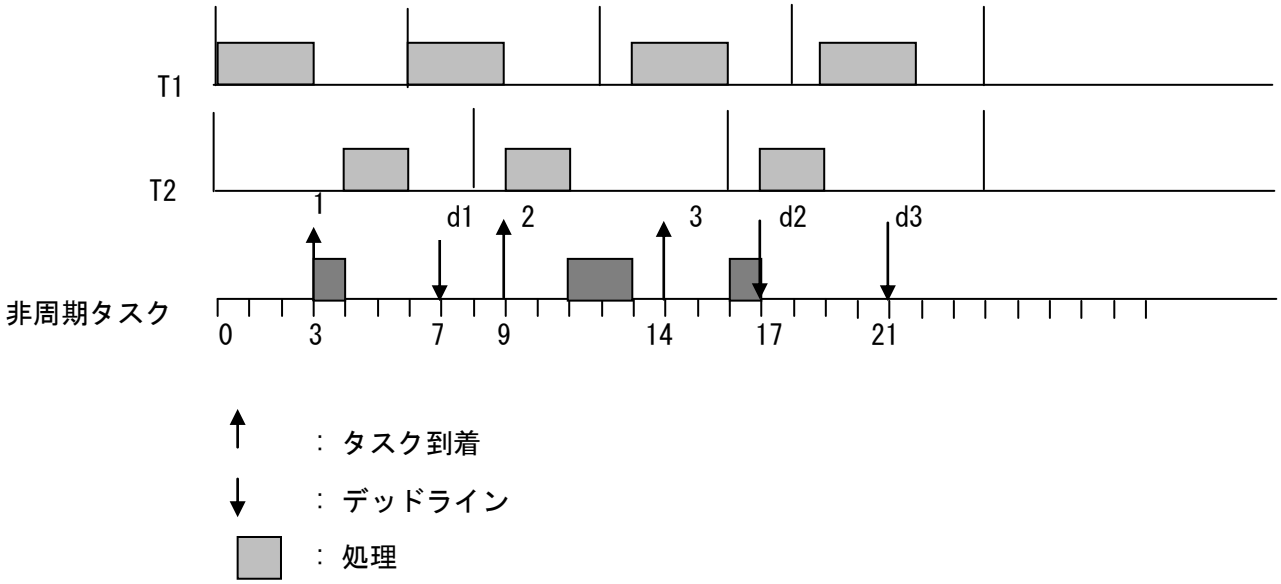

図 2.4 TBS での動作タイミング図

2.6　最悪実行時間

リアルタイムスケジューリングにおいて、スケジューラビリティの保証の必要性から、タスクの実行は最悪実行時間（Worst-Case Execution Time: WCET）を費やすものと仮定されることが一般的である。短い実行時間を仮定した場合、実行時間が予定以上の長さとなった場合、デッドラインミスが発生する可能性があるためである。

近年のプロセッサやソフトウェアは大規模化・複雑化が進み、タスクの最悪実行時間の見積／解析は困難になる傾向がる。例えば、命令の深いパイプライン実行やアウトオブオーダー実行は正確な実行サイクルの見積を困難にしている。また、多数のタスクを含むシステムでは、頻繁なタスクスイッチが原因となり、各メモリ参照がキャッシュメモリでヒットするかミスするかを判断することは実質的に不可能である [15]。更に、実行されるプログラムは多数の分岐やループなどの制御構造を含み、実行パスの探索範囲は膨大であるため、最悪実行時間を求めることは事実上不可能である [16]。したがって、最悪実行時間の見積は悲観的にならざるをえず、実際の実行時間との隔たりは大きくなる傾向がある。この隔たりは、タスクの実行時間に基づいて実行順序を決定するスケジューリング方式においては、最適なスケジュールを得ることへの障害となる。例えば、Shortest Job First（SJF）や Shortest Remaining Time First（SRTF）アルゴリズム等ではタスクの実行時間が実行順序に直接反映されるため、予定した最悪実行時間よりも短い実行時間となった場合は、平均ターンアラウンドタイムは最小とはならない [17]。

本研究では、タスクの実行時間が実行ごとに変動すること、特に最悪実行時間よりも短くなることを利用し、非周期タスクの応答時間を短縮することを目指す。TBS は実行時間に依存してデッドラインを決定するアルゴリズムであり、このことが本研究において TBS を対象とした理由の一つである。また、短い実行時間を仮定することになるが、このことがタスクセットのスケジューラビリティに影響を及ぼさないことが重要である。3 章で提案する方式は、短い実行時間を仮定することで非周期タスクの応答時間を短縮し、かつスケジューラビリティを保障するものである。

第 3 章　TBS の改良方法の提案

3.1　概要

従来の TBS は、非周期タスク（ソフトあるいは非リアルタイムタスク）を単に周期タスクのバックグラウンドで実行する方式と比較し、バンド幅を考慮したデッドラインを計算して使用することから、非周期タスクの応答時間を短くする特徴がある[3]。一方、デッドライン計算において、タスクの実行時間としては事前に見積もった最悪実行時間を使用するため、大きすぎるデッドライン時刻を計算する可能性が高い。このことから、EDF の性質により、応答時間の短縮には限界がある。

これに対する改善方法の一つに、Improved TBS 方式がある[3]。この方法は TBS と比較し、非周期タスクのデッドラインを短くする特徴がある。具体的には、非周期タスクの要求が発生した時点で、TBS にしたがって将来のスケジュールを構築し、そこから当該非周期タスクの終了予定時刻を求め、この予定終了時刻を新たにデッドラインとして設定し、再度スケジュールを構築する。予定終了時刻が早くなる限り、このステップを繰り返す。ただし、非周期タスクの予定の終了時刻を知るために、毎ステップで全タスクのスケジュールを構築することが必要である。このための計算オーバヘッドは大きく、この方式は実用性が低いことが予想される。

本研究では、TBS のデッドライン計算において、非周期タスクの最悪実行時間の代わりに予測実行時間を使用してデッドラインを計算することにより、より短いデッドラインが設定できることを利用して応答時間を短縮させる方法を試みる。これによって、従来の TBS と比較し、非周期タスクの応答時間がより小さくなることが期待できる。提案する方式では、非周期タスクの最悪実行時間や過去の実行時間に基づいて予測した実行時間を使用することにより、小さなデッドライン時刻を設定可能である。実際のタスクの実行時に、この予測した実行時間を経過した場合は、最悪実行時間を使用して計算されるデッドラインに再設定して、残りの実行を再スケジュールする。この手法は、周期的なハードタスクのスケジューラビリティには影響を及ぼさず、かつ定性的に従来の TBS よりも応答時間を短縮できることが特色である。

3.2　デッドライン計算について

3.2.1 最悪実行時間の半分の使用

短いデッドラインを算出する方法の一つの候補として、最悪実行時間の半分の値の使用を検討する。実際の実行時間は変動するが、その時間の多くは最悪実行時間の半分以下に分布する非周期タスクの場合、この方法が適切である可能性がある。（しかし、組込みシステムのアプリケーションは多種多様であるため、全てのアプリケーションに対して万能であることは期待できない。）

3.2.2 タスクの前回の実行時間の使用

計算機におけるプログラム実行／振る舞いには様々な局所性あり、動作条件の急速変化がない限り、タスク実行時間は同一タスクの前回の実行とほぼ同じであると期待できる場合がある。この方法を実現するためには、RTOS はタスクの実行が完了した時にそのタスク実行時間を記憶し、次回のタスク起動要求が発生したときにデッドライン計算に使用することになる。

3.2.3 タスク実行時間の平均値の使用

前節と同じく過去のタスク実行時間に基づくが、過去の全実行時間の平均値を使用してデッドラインを算出する。非周期タスクの実行毎にタスクの実行時間を記録し、記録した実行時間を使用して過去のタスク実行時間の平均値を算出する。この方法を数学的に説明するために、表 3.1 の記号を定義する。

表 3.1 デッドライン計算のためのパラメータ

パラメータ	用途
k	非周期タスクの起動要求の番号
d_k	TBS で算出される k 番目の非周期タスクの絶対デッドライン時刻
d_{k-1}	k-1 番目の非周期タスクの絶対デッドライン時刻
WCETk	k 番目の非周期タスクの最悪実行時間
AET	タスクの実際の実行時間
U_s	非周期タスクに割当てられるバンド幅
pd_k	改良 TBS で算出される k 番目の非周期タスクのデッドライン時刻
AVE	タスク実行時間の平均値
tim	システム時刻

k-1 番目の非周期タスクの実行が完了したときにタスク実行時間の平均値を以下の式に従って更新する。この式からわかるように、その時点までの平均値と直前の実行時間との平均（加重平均）となっている。

$$AVE = (AVE + AET)/2$$

また、k 番目の非周期タスクが到着したときに、デッドラインを以下の式によって算出する。

$$pd_k = \max(r_k,\ d_{k-1}) + AVE/U_s$$

スケジューラは pd_k を使用してスケジューリングを行う。k 番目の非周期タスクの実行が完了する時刻が pd_k より早い場合、期待通りである。実行完了時に次の k+1 番目のタスクのデッドライン計算のための d_{k-1} が pd_k で設定（上書き）される。一方、システム時刻が pd_k になっても k 番目の非周期タスクの実行が完了していない場合、以下の式によってデッドラインを再計算し、レディキューに当該非周期タスクを再挿入する。

$$d_k = pd_k + (\text{最悪実行時間} - \text{実行した時間})/\text{サーバの使用率}$$

$$= pd_k + (WCETk - actually_executed_time)/Us$$

k 番目の非周期タスクを実行している最中に k+1 番目の非周期タスクの起動要求が発生した場合、k+1 番目の非周期タスクのデッドラインの算出にあたって d_k が必要となるが、以下の式から算出される d_k を使用する。(k 番目のタスクが k+1 番目の起動要求より前に、かつ pd_k 以内に完了したときは pd_k で上書きされる。)

$$d_{k+1} = \max(r_{k+1},\ d_k) + WCETk/U_s$$

評価で用いる ITRON ベースの OS ではシステムタイマタスクの静的優先度が他のタスクより高く設定されている。このため、どのタスクが実行されていてもシステムタイムティックになったらそのタスクを中断してシステムタイマタスクが実行される。システムタイマタスクは、中断されたタスクの TCB 内の実行時間の情報を更新することで、タスクの実行時間を記録することができる。

3.3　提案方式の適用例

表 3.3 のタスクセットを使用して提案方式の動作を説明する。タスクセットは 2 つの周期タスク（T1、T2）と、1 つの非周期タスクからなる。表より、周期タスクのプロセッサ使用率は Up = 3/6 + 2/8 = 0.75 となる。したがって、非周期タスクのプロセッサ使用率を Us = 1 - Up = 0.25 とする。

表 3.2 タスクセット

タスク	周期	実行時間	WCET	起動時刻
T1	6	3	-	周期
T2	8	2	-	周期
非周期タスク	-	2	4	3

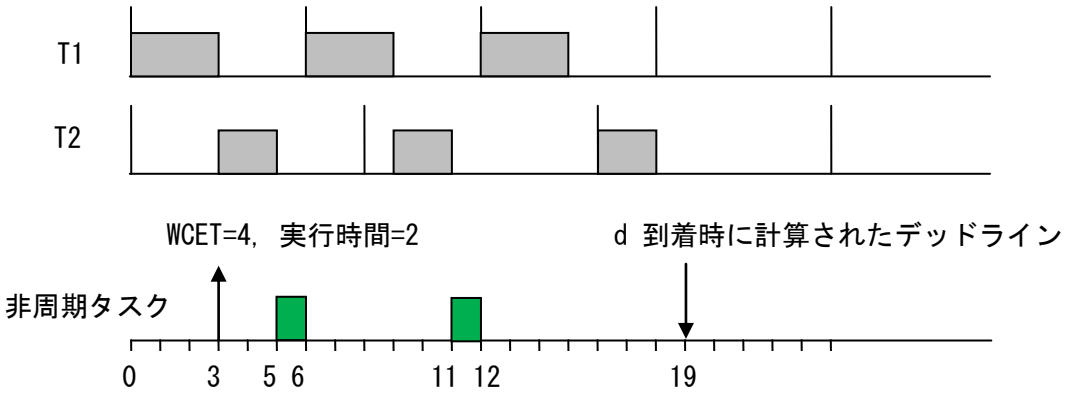

図 3.1 TBS

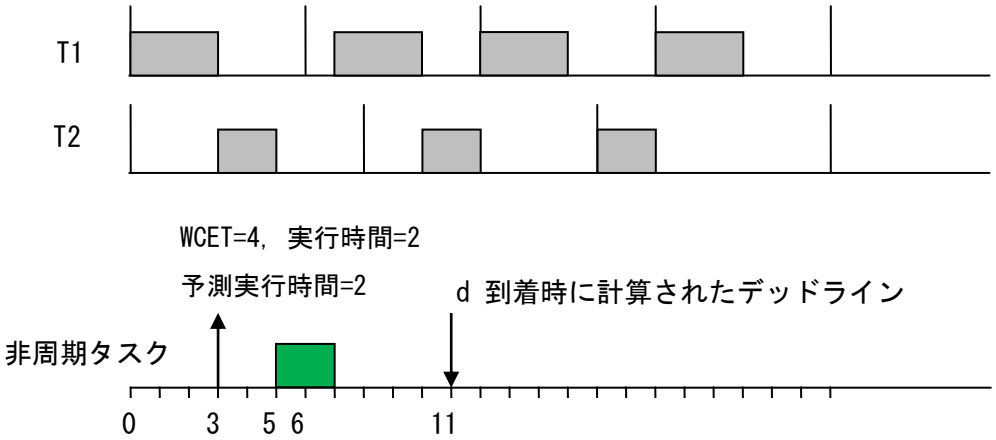

図 3.2 提案方式

図 3.3.1 に従来の TBS でのタスクスケジューリング例を示す。非周期タスクが時刻 t=3 に到着し、デッドラインの計算が行われ、d = r + WCET/U_s = 3+4/0.25 = 19 となる。スケジューリングは EDF にしたがい、当該タスクは時刻 t=5 で実行を開始し、時刻 t=6 で中断し、時刻 t=11 で再開し、時刻 t=12 に終了する。結果的に、非周期タスクの応答時間 = 12 - 3 = 9 となる。

図 3.3.2 に提案方式でのタスクスケジューリングを示す。非周期タスクが時刻 t=3 に到着し、デッドラインの計算が行われ、d = r + 予測時間/U_s = 3+2/0.25 = 11 となる。d は同じく実行可能状態である T2 のデッドラインよりも遠いため、周期タスク T2 が選ばれて実行される。時刻 t=5 に周期タスク T2 の実行が完了したら非周期タスクの実行が開始される。周期タスク T1 が時刻 t=6 に到着し、T1 のデッドラインの計算が行われ、T1 のデッドライン = 6 + 周期 = 12 となる。非周期タスクのデッドラインが周期タスクより近いため、非周期タスクが選ばれて実行される。非周期タスクの実行が時刻 t=7 に完了する。この場合、提案方式での非周期タスクの応答時間 = 実行完了時刻 - 到着時刻 = 7 - 3 = 4 である。

一方、提案方式でのタスクスケジューリングを行う際に、予測時間が実際の実行時間より小さく設定されたとき、計算されたデッドラインになっても非周期タスクの実行が完了しなかったらデッドラインを再計算する必要がある。図 3.3.3 にデッドライン計算ミスの例を示す。非周期タスクは予測実行完了時刻(t=7)になっても実行が完了していないため、非周期タスクのデッドラインが再計算される。この例では、再計算されたデッドラインを使用して EDF によりスケジュールした結果、残りの実行が時刻 t=15 で開始され、時刻 t=17 で終了している。

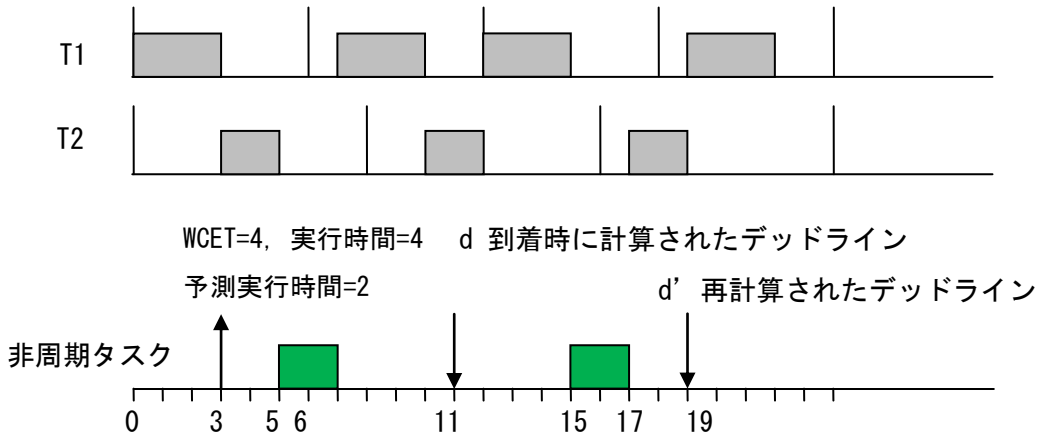

図 3.3 提案方式でのデッドライン再計算タイミング

3.4　スケジューラビリティ

提案方式のスケジューラビリティは以下のように説明可能である。

提案方式では、一つの非周期タスクの実行を、前半部分（予測した実行時間に達するまでの実行）と後半部分（予測した実行時間後の実行）の二つのサブインスタンスに分割する。これらの二つのサブインスタンスを、同時に起動要求が発生した二つの独立したタスク実行とみなすことにより、結果的に、従来の TBS の場合と同じデッドラインが”それぞれに”割り当てられ、TBS と同様にふるまうことになる。二つのサブインスタンスによる使用率は、以下のようにオリジナル TBS と等しい。（式内で actually_executed_time = AVE である。）

$$Updk = AVE / (pdk - max(rk, dk-1)) = Us$$

$$Udk = (WCETk - actually_executed_time) / (dk - pdk) = Us$$

このことから、提案方式のスケジューラビリティは文献[2]におけるものと同一となり、Up + Us <= 1 のときタスクセットはスケジュール可能となる。

3.5　実装の複雑さ

提案スケジューリングアルゴリズムでは、非周期タスク実行は 2 つのサブインスタンスに分解される。しかし、オペレーティングシステムは一つのタスクを一つの情報セット（タスク制御ブロック：TCB）で管理するべきである。このことは、以下のように実現可能である。予測した実行時間が経過し、タスク実行が未完了の場合、デッドラインを再設定し、レディキューに再挿入する。この点がオリジナルの TBS との唯一の相違点である。

タスク実行が予測した実行時間に達したことを検出するために、ティックタイミング毎にスケジューラを実行することが必要であるが、これはティックタイミングで発生する割り込み時にスケジューラを呼び出すことを意味するが、（実は）これはオペレーティングシステムが通常行う手続きである。

デッドライン計算のオーバヘッドは以下のように考察される。非周期タスクの要求発生時に、（3.2.3 節の）以下のデッドライン計算を行う。

$$pdk = max(r_k, d_{k-1}) + AVE/Us$$

AVE は前回の非周期タスク実行時に算出されたものである。この値とサーババンド幅である Us で除算を行うが、Us が定数であることを考慮すると、乗算に還元可能であり、小さいオーバヘッドで計算可能となる。

予測した実行時間の経過時に、(3.2.3 節の）以下のデッドラインの再計算を行う。

$$dk = pdk + (\text{最悪実行時間} - \text{実行した時間})/\text{サーバの使用率}$$

$$= pdk + (WCETk - actually_executed_time)/Us$$

ここで、actually_executed_time は AVE であるため、前出の式を代入することにより、

$$dk = \max(r_k,\ d_{k-1}) + AVE/Us + (WCETk - AVE)/Us$$

$$= \max(r_k,\ d_{k-1}) + WCETk/Us$$

となる。WCETk と Us が定数（不変）であることを考慮すると、第二項は定数となり、実際の計算は第一項と定数との加算のみとなり、オーバヘッドは小さいことがいえる。

第 4 章　スケジューラの実装

4.1　既存 RTOS について

本研究で使用するリアルタイムオペレーティングシステム(RTOS)はμITRON4.0 のスタンダードプロファイル仕様の RTOS であり、2.2 節で紹介した全機能を有している[8]。本 RTOS 内では各タスクが持つ静的優先度に従うスケジューリングを行っている。本研究では、本 RTOS のスケジューラを提案する方式を実現するものに変更する。

4.2　デッドライン算出タイミング

4.2.1 概要

既存の RTOS にはスケジューリング関数があり、スケジューリングイベントが発生するとレディキューを探索して最も優先順位が高いタスクを選んで次に実行すべきものとしてスケジューリングする。このレディキューは静的優先度毎に存在し、最も優先順位が高いタスクは対応する優先度のレディキューの先頭にある。また、最も優先順位が低いタスクは対応する優先度のレディキューの末尾にある。このため、レディキューにタスクの TCB(Task Control Block)情報を挿入する時に、タスクのデッドラインが近い順に挿入するようにすることにより、既存 OS のスケジューラから、TBS の基本ポリシーとして使用される EDF のスケジューラに変更することができる。

既存 RTOS では、アプリケーションがシステムコール act_tsk/iact_tsk を使用することにより、カーネルにタスクの起動を要求する。また、起動要求付きで新規タスク生成(cre_tsk)を行うこともあることから、非周期タスクのデッドラインの算出は act_tsk/iact_tsk/cre_tsk の内部で実装することとする。

4.2.2　デッドライン計算

図 4.1 に TBS におけるデッドライン計算のチャート図を示す。デッドライン計算時に、タスクの種類をチェックする。周期タスクの場合、デッドラインはタスク到着時刻にタスクの周期を加算することにより算出される。一方、非周期タスクの場合、3.2 節における算出式に従って（タスク到着時刻と前回の算出されたデッドラインのうち大きい方を選び、タスク実行時間/Us と加算して）デッドラインを算出する。

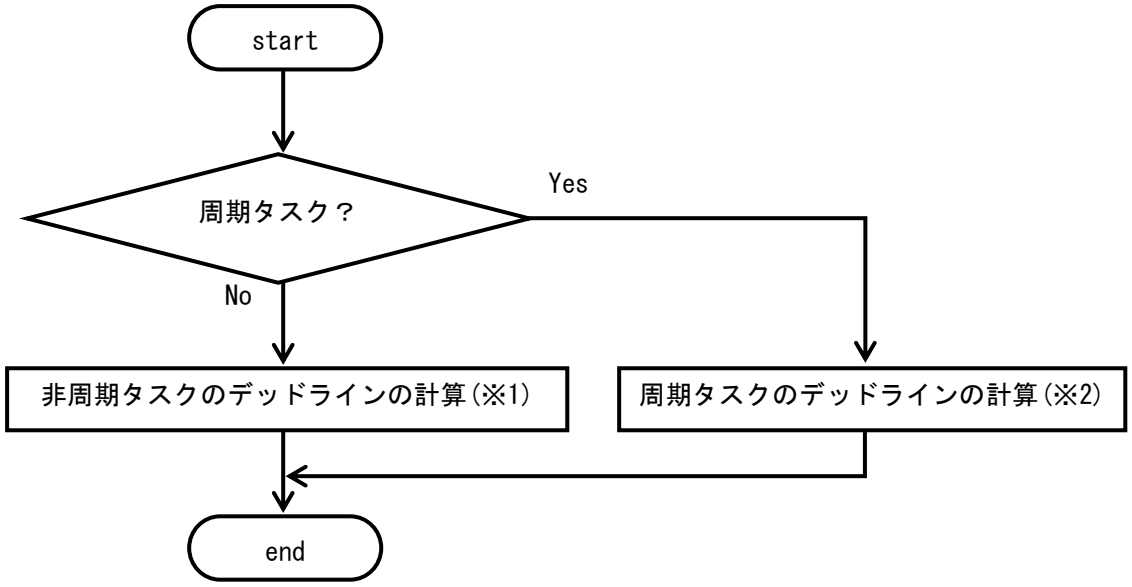

図 4.1 デッドライン計算

※1 ： デッドライン = max(到着時刻、前回のデッドライン) + タスク実行時間/Us
※2 ： デッドライン = 到着時刻 + タスクの周期

4.3 実装

4.3.1 iact_tsk

ITRON 環境では割り込みや CPU 例外（非タスクコンテキスト）のルーチン／ハンドラで iact_tsk をコールすることによってタスク起動要求を発行することができる。iact_tsk のソースコードの一部を以下に示す。

```
ER iact_tsk ( ID tskid )
{
        ----------------------------省略----------------------------------------
      if( tcb -> type == APERIODIC_TASK)
      {
         tcb ->operated_time = 0;        // 実行した時間を初期化
         tcb -> response_time = 0;                                          (1)
         tcb -> ave_response_time = 0;
         tcb -> act_time         = (INT)_kernel_systim;
         time  =  _kernel_current_task_deadline>=(TIME)_kernel_systim?
                  _kernel_current_task_deadline : (TIME)_kernel_systim;     (2)

         tcb -> deadline = time + (TIME)((long)(100*tcb -> wcet) / (long)(30));(3)
                _kernel_current_task_deadline = tcb -> deadline;
      }
      else {
            tcb ->operated_time = 0;     // 実行した時間を初期化
            tcb -> deadline = (TIME)_kernel_systim + (TIME)tcb -> period;   (4)
      }
     _kernel_queue_insert_prev (&_kernel_ready_q[tcb -> tskpri],
                                (_KERNEL_QUEUE *) &tcb -> queue );          (5)
      /* スケジュール関数 */
     _kernel_sched ( 0 );                                                   (6)

 ----------------------------省略----------------------------------------
}
```

上記の iact_tsk のソースコードを使用してデッドライン計算について説明する。

コード内の(1)では、非周期タスクの実行時間変数、起動時刻変数などを初期化している。続いて、(2)、(3)においてデッドライン計算を行う。(2)では記録されている（前回の非周期タスク実行の）デッドライン（*_kernel_current_task_deadline*）とシステム時刻（*_kernel_systim*）を比較して大きな方を選び、これに対して(3)でサーババンド幅に基づく計算を行い、加算を行っている。(コード内では、タスクの実行時間とサーバ使用率(0.3)に対して 100 を掛けている。これは評価環境の都合上、浮動小数演算を整数演算で代替するためである。)

本コードは(3)の計算で、タスクの実行時間としてタスクの最悪実行時間（*tcb -> wcet*）を使用するものである。すなわち、従来の TBS におけるデッドライン計算となっている。この部分が従来 TBS と 3.2 節で述べた提案手法で異なることになり、以下のように整理される。

・<u>従来の TBS で Us = 0.3 の場合</u>

デッドライン = max(前回のデッドライン, システム時刻) + 最悪実行時間/Us
= max(前回のデッドライン, システム時刻) + 100*最悪実行時間/30

・<u>案①:最悪実行時間の半分を使用する。Us = 0.3 の場合</u>

デッドライン = max(前回のデッドライン, システム時刻) + 最悪実行時間/(Us×2)
= max(前回のデッドライン, システム時刻) + 100*最悪実行時間/60

・<u>案②：前回の実行時間を使用する。Us = 0.3 の場合</u>

デッドライン = max(前回のデッドライン, システム時刻) + 前回の実行時間/Us
= max(前回のデッドライン, システム時刻) + 100*前回の実行時間/30

・<u>案③：過去実行時間の平均値を使用する。Us = 0.3 の場合</u>

デッドライン = max(前回のデッドライン, システム時刻) + 過去実行時間の平均値/Us
= max(前回のデッドライン, システム時刻) + 100*過去実行時間の平均値/30

当該システムコールによって起動されるタスクが周期タスクの場合は、コード内の(4)で実行時間変数の初期化とデッドライン計算（デッドライン = システム時刻 + タスクの周期）を行う。

本システムコールは、上記のデッドライン計算を行った後、(5)において *_kernel_queue_insert_prev* をコールしてレディキューにタスクを挿入し、最後にスケジュール関数（*_kernel_sched*）をコールする。

4.3.2 act_tsk

ITRON 環境ではタスクコンテキスト（割り込みや CPU 例外以外の通常のタスク実行時）で act_tsk をコールすることによりタスク起動要求を発行する。act_tsk の実行コンテキストは iact_tsk と異なるが、処理はほとんど同じであるため、本項目の説明は省略する。詳細は 4.3.1 節を参照。

4.3.3 cre_tsk

ITRON 環境では TA_ACT 属性を指定してタスク生成 cre_tsk をコールすると、カーネルがタスク生成を行った後に、自動的にタスク起動要求も発行するため、cre_tsk でもデッドランを算出する場合がある。デッドライン計算は iact_tsk での処理と同じであるため本項目の説明も省略する。詳細は 4.3.1 節を参照。

4.3.4 レディキューへの挿入

以下にタスクの TCB をレディキューに挿入する関数 _kernel_queue_insert_prev のソースコードを示す。

```
void _kernel_queue_insert_prev ( _KERNEL_QUEUE *queue, _KERNEL_QUEUE *entry )
{
        _KERNEL_QUEUE *ptr;
        TIME dl = entry -> self -> deadline;

        for ( ptr = queue -> next; ptr != queue; ptr = ptr -> next )
        {
                if ( dl < ptr -> self -> deadline )
                  break;
        }
        entry -> prev = ptr -> prev;
        entry -> next = ptr;
        ptr -> prev -> next = entry;
        ptr -> prev = entry;
}
```

(1)

(2)

以下、レディキューに対してデッドラインが近い順にタスク（TCB）を挿入する処理を説明する。

コード内の(1)において、挿入されるタスクのデッドラインと、レディキューに接続されているタスクのデッドラインを比較して挿入タスクの位置を確定する。続いて、(2)においてタスクの位置を確定した後に、タスクをレディキューに挿入する。（これにより、キュー内の接続タスクのデッドラインは常に昇順となり、スケジューリングにおいて先頭タスクを選択することにより EDF が実現される。）

4.3.5 システムタイマタスク

以下にシステムタイマタスクの処理のうち、デッドライン管理と関連するソースコードを示す。

```
void _kernel_timer ( void )
{
  ----------------------------省略----------------------------------
   tsk = (_KERNEL_QUEUE *)&_kernel_ready_q[2];                    (1)
   if ( tsk->next != tsk )
   {
     tcb = tsk->next->self;
     // 非周期タスクに対して
     if (tcb->type == APERIODIC_TASK)
     {
        // 実行時間を更新
       tcb->operated_time = tcb->operated_time + 1;               (2)
       // デッドラインチェック
       if (tcb->deadline <= _kernel_systim)
       {// デッドラインを超えたら再度設定
            tcb->deadline = _kernel_systim + (TIME)((long)(100*(tcb->wcet -
                                  tcb->operated_time) ) / (long)(30));   (3)
          _kernel_queue_remove((_KERNEL_QUEUE *) &tcb->queue);    (4)
           _kernel_queue_insert_prev ( &_kernel_ready_q[tcb->tskpri],
                                        (_KERNEL_QUEUE *) &tcb->queue );   (5)
        }
      }
     ------------------------------省略----------------------------------
  return;
}
```

コード内の(1)において、レディキューの先頭にあるタスク情報を取得する。続いて(2)において、レディキューの先頭にあるタスクが非周期タスクの場合、タスク実行時間を更新する。(3)では、タスクのデッドラインがシステム時刻以下になっている時に、タスクのデッドラインを再計算している。続いて(4)においてデッドラインが再計算されたタスクをレディキューから取り出す。最後に(5)においてデッドラインが再計算されたタスクをレディキューに挿入する。

4.3.6 ext_tsk

ITRON環境では、タスクの実行が終了する際に、システムコールext_tskがコールされる。以下にext_tskの処理の中で、実行時間の記録と応答時間の算出に関連するソースコードを示す。

```
void ext_tsk ( void )
{
-----------------------------省略-----------------------------------
  if (tcb -> type == APERIODIC_TASK)
  {
    INT tmp;

    tcb -> save_operated_time = tcb -> operated_time;                    (1)
    tcb -> operated_average_time = (tcb->operated_average_time +
                                    tcb->operated_time)/2;               (2)
    // 応答時間 = 終了時刻 - 起動時刻
    tmp = (INT)_kernel_systim - tcb -> act_time;                         (3)

   -----------------------------省略-----------------------------------
}
```

コード内の(1)において、同一タスクの次回の起動要求が到着した時にデッドラインを算出する際に必要となる情報として、タスク実行時間を保存する。続いて(2)において当該タスクの平均実行時間を更新する。最後に(3)において当該非周期タスクの応答時間を算出する。

第5章　評価

5.1 評価環境

本研究では評価環境として ITRON オペレーティングシステムを使用するバイナリを実行可能な既存の CPU シミュレータを利用する。この CPU シミュレータは Cygwin の X Windows で動作する。シミュレータの構成を以下に示す。

・クロックサイクル毎の動作をシミュレート。
・SPARC version 8 の整数命令セットを実行[9]。
・各命令は基本的に 1 クロックサイクルで実行。（乗算や除算などのいくつかの命令は複数クロックサイクルで実行。）
・命令／データ分離型キャッシュ。（4 ウェイ・セット・アソシアティブ方式。LRU 置換方式。サイズは各 16KB。ブロックサイズは 32B。）
・キャッシュアクセスレイテンシはヒット時 1 サイクル、ミス時 10 サイクル。

本シミュレータは ITRON RTOS のカーネルデータを解釈・表示可能であり、評価のための各種情報を提供する。

5.2 タスクセット

本研究では、重要度の高い周期タスクのデッドラインに関するスケジューラビリティを確保しつつ、非周期タスクの応答時間を可能な限り小さくすることを目的としている。したがって、提案する方式が従来の TBS と比較し、どの程度非周期タスクの応答時間を短縮するかをシミュレーション実験で評価する。

タスクの応答時間はタスク起動要求の時刻とタスク実行終了時刻との間の差である。本研究では周期タスクによる CPU 使用率が Up = 0.6, 0.7, 0.8, 0.9 となる周期タスクセットを用意して、いくつかの動作条件（実行時間、タスク使用率）の非周期タスクを混在させて評価する。

5.2.1　Up=0.6 の周期タスクセット

プロセッサ使用率が 0.6 となる周期タスクセットを 5 つ用意した。各周期タスクセットは 4 つのタスク（task1～4）からなり、それぞれの周期、実行時間は表 5.1～5.5 の通りである。なお、実行時間はシステムタイマタスクが管理するティック単位である。

表 5.1　Up=0.6 の周期タスクセット 1

No.	タスク名	周期	実行時間
1	task1	8	1
2	task2	12	3
3	task3	16	2
4	task4	20	2

表 5.2　Up=0.6 の周期タスクセット 2

No.	タスク名	周期	実行時間
1	task1	24	3
2	task2	12	3
3	task3	16	2
4	task4	20	2

表 5.3　Up=0.6 の周期タスクセット 3

No.	タスク名	周期	実行時間
1	task1	8	1
2	task2	24	6
3	task3	16	2
4	task4	20	2

表 5.4　Up=0.6 の周期タスクセット 4

No.	タスク名	周期	実行時間
1	task1	8	1
2	task2	12	3
3	task3	32	4
4	task4	20	2

表 5.5　Up=0.6 の周期タスクセット 5

No.	タスク名	周期	実行時間
1	task1	8	1
2	task2	12	3
3	task3	32	4
4	task4	40	4

5.2.2　Up=0.7の周期タスクセット

プロセッサ使用率が 0.7 となる周期タスクセットを 5 つ用意した。各周期タスクセットは 4 つのタスク（task1～4）からなり、それぞれの周期、実行時間は表 5.6～5.10 の通りである。

表 5.6　Up=0.7の周期タスクセット 1

No.	タスク名	周期	実行時間
1	task1	5	1
2	task2	10	2
3	task3	15	3
4	task4	20	2

表 5.7　Up=0.7の周期タスクセット 2

No.	タスク名	周期	実行時間
1	task1	25	5
2	task2	10	2
3	task3	15	3
4	task4	20	2

表 5.8　Up=0.7の周期タスクセット 3

No.	タスク名	周期	実行時間
1	task1	5	1
2	task2	30	6
3	task3	15	3
4	task4	20	2

表 5.9　Up=0.7の周期タスクセット 4

No.	タスク名	周期	実行時間
1	task1	5	1
2	task2	10	2
3	task3	45	9
4	task4	20	2

表 5.10　Up=0.7の周期タスクセット 5

No.	タスク名	周期	実行時間
1	task1	5	1
2	task2	10	2
3	task3	15	3
4	task4	40	4

5.2.3　Up=0.8 の周期タスクセット

プロセッサ使用率が 0.8 となる周期タスクセットを 5 つ用意した。各周期タスクセットは 4 つのタスク（task1～4）からなり、それぞれの周期、実行時間は表 5.11～5.15 の通りである。

表 5.11　Up=0.8 の周期タスクセット 1

No.	タスク名	周期	実行時間
1	task1	4	1
2	task2	5	1
3	task3	15	3
4	task4	20	3

表 5.12　Up=0.8 の周期タスクセット 1

No.	タスク名	周期	実行時間
1	task1	8	2
2	task2	5	1
3	task3	15	3
4	task4	20	3

表 5.13　Up=0.8 の周期タスクセット 3

No.	タスク名	周期	実行時間
1	task1	4	1
2	task2	10	2
3	task3	15	3
4	task4	20	3

表 5.14　Up=0.8 の周期タスクセット 4

No.	タスク名	周期	実行時間
1	task1	4	1
2	task2	5	1
3	task3	25	5
4	task4	20	3

表 5.15　Up=0.8 の周期タスクセット 5

No.	タスク名	周期	実行時間
1	task1	8	2
2	task2	10	2
3	task3	15	3
4	task4	20	3

5.2.4 Up=0.9 の周期タスクセット

プロセッサ使用率が 0.9 となる周期タスクセットを 5 つ用意した。各周期タスクセットは 4 つのタスク（task1～4）からなり、それぞれの周期、実行時間は表 5.16～5.20 の通りである。

表 5.16　Up=0.9 の周期タスクセット 1

No.	タスク名	周期	実行時間
1	task1	8	2
2	task2	4	1
3	task3	15	3
4	task4	30	6

表 5.17　Up=0.9 の周期タスクセット 2

No.	タスク名	周期	実行時間
1	task1	16	4
2	task2	4	1
3	task3	15	3
4	task4	30	6

表 5.18　Up=0.9 の周期タスクセット 3

No.	タスク名	周期	実行時間
1	task1	8	2
2	task2	12	3
3	task3	15	3
4	task4	30	6

表 5.19　Up=0.9 の周期タスクセット 4

No.	タスク名	周期	実行時間
1	task1	8	2
2	task2	4	1
3	task3	45	9
4	task4	30	6

表 5.20　Up=0.9 の周期タスクセット 5

No.	タスク名	周期	実行時間
1	task1	8	2
2	task2	4	1
3	task3	15	3
4	task4	5	1

5.2.5 非周期タスク情報

各非周期タスクセットにおいて、起動要求タイミングは 5 回あり、起動ごとにタスク実行時間が異なるモデルを使用する。最悪実行時間は 13 とする。非周期タスクの起動要求タイミングと実行時間は表 5.21～5.25 の通りである。

表 5.21　非周期タスクの情報 1

パターン	実行時間	起動要求タイミング
Case 1	1	1
Case 2	2	10
Case 3	3	19
Case 4	4	29
Case 5	7	44

表 5.22　非周期タスクの情報 2

パターン	実行時間	起動要求タイミング
Case 1	8	1
Case 2	2	10
Case 3	3	19
Case 4	4	29
Case 5	7	44

表 5.23　非周期タスクの情報 3

パターン	実行時間	起動要求タイミング
Case 1	1	1
Case 2	8	10
Case 3	3	19
Case 4	4	29
Case 5	7	44

表 5.24　非周期タスクの情報 4

パターン	実行時間	起動要求タイミング
Case 1	1	1
Case 2	2	10
Case 3	9	19
Case 4	4	29
Case 5	7	44

表 5.25　非周期タスクの情報 5

パターン	実行時間	起動要求タイミング
Case 1	1	1
Case 2	2	10
Case 3	3	19
Case 4	8	29
Case 5	7	44

シミュレーションでは、5.2.1 節から 5.2.4 節までの周期タスクセットと上記の非周期タスクセットを組み合わせて実行して、評価を行った。

5.3 評価結果

5.2 節で用意した周期タスクセットと非周期タスクを混在させ、シミュレーションを行った。評価の対象は非周期タスクの応答時間であり、非周期タスクの起動要求が到着した時のシステム時刻を記録し、その非周期タスクの終了時に ext_tsk がコールされるため、ext_tsk の内部でシステム時刻と記録した要求時のシステム時刻の差を当該非周期タスクの応答時間として出力させた。CPU 使用率毎の非周期タスクの応答時間の平均値を表 5.26 に示す。

表 5.26　非周期タスクの応答時間

Up	従来 TBS	案①	案②	案③
0.6	13.79	13.75	13.76	9.66
0.7	16.57	16.53	17.33	12.83
0.8	24.03	21.61	23.95	16.97
0.9	51.94	46.06	54.35	44.36

表 5.21　非周期タスクの応答時間から表 5.27 に従来の TBS に対する改善率を示す。

表 5.27　従来の TBS に対する改善率

Up	案①	案②	案③
0.6	99.7%	99.8%	70.1%
0.7	99.8%	104.6%	77.4%
0.8	89.9%	99.7%	70.6%
0.9	88.7%	104.6%	85.4%

(*1) 案①　最悪実行時間の半分を使用してデッドラインを算出する方式
　　案②　前回の実行時間を使用してデッドラインを算出する方式
　　案③　過去実行時間の平均値を使用してデッドラインを算出する方式

表 5.26 をグラフ化したものが図 5.1 である。

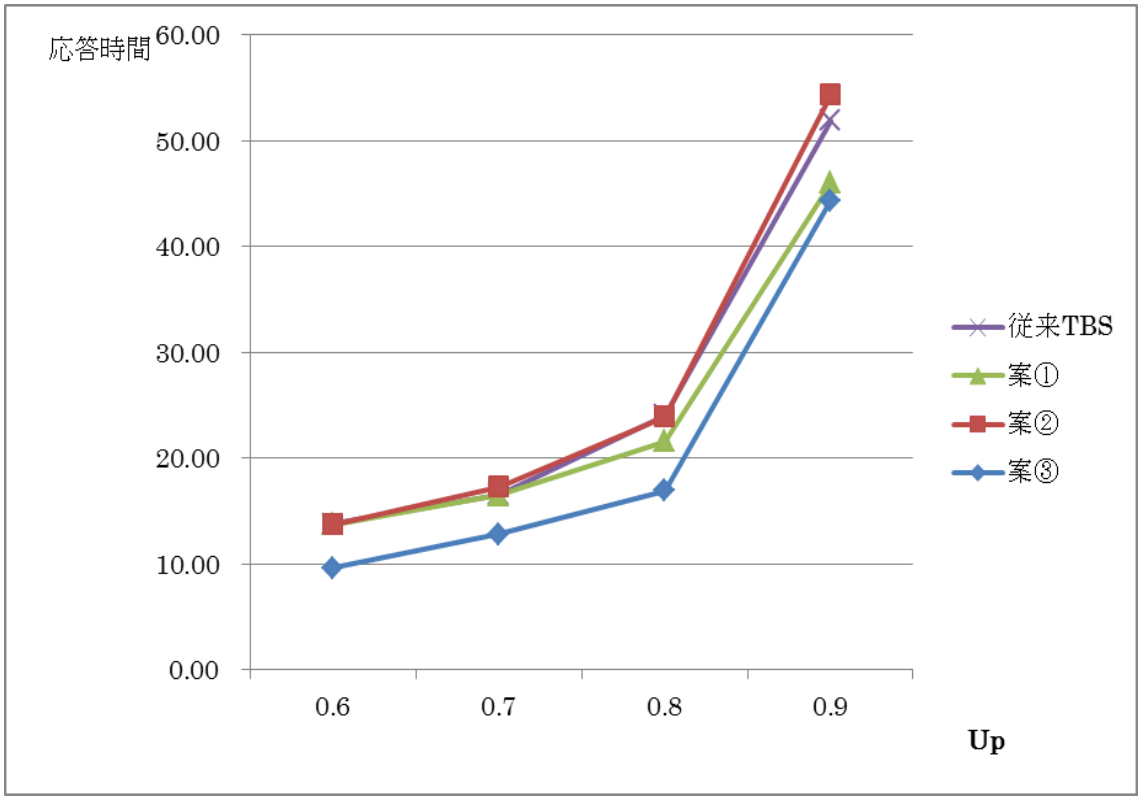

図 5.1　非周期タスクの応答時間

以下には評価結果について考察する。

1. 案①での非周期タスクの応答時間が従来の TBS と比べると周期タスクの CPU 使用率が 0.6 や 0.7 の場合、あまり改善されないが、周期タスクの CPU 使用率が 0.8 や 0.9 の場合、それぞれ 10.1%, 11.3%短縮される。
2. 案②での非周期タスクの応答時間が従来の TBS と比べると周期タスクの CPU 使用率が 0.6 や 0.8 の場合、あまり改善されないし、周期タスクの CPU 使用率が 0.7 や 0.9 の場合は長くなってしまう。原因としてはこれからの実行時間が前回と大幅長くなると、デッドラインの再計算が発生することにより、非周期タスクの応答時間が従来の TBS より多少長くなることと考えられる。
3. 案③で周期タスクの CPU 使用率が 60%の場合に、非周期の応答時間が最大で 29.9%短縮される。また、周期タスクの CPU 使用率が 90%の場合は非周期タスクの応答時間が最小で 14.6%である。

評価結果により、提案する方式（案③）での非周期タスクの応答時間が従来の TBS と比較し、最大で 29.9%短縮されることを確認した。この短縮率はタスクセット／アプリケーションに依存している。

第6章　まとめ

6.1　総括

本研究では、周期タスクと非周期タスクの混在タスクセットを対象としたスケジューリングアルゴリズムであるTotal Bandwidth Serverを改良することで、重要度の高い周期タスクのデッドラインに関するスケジューラビリティを確保しつつ、非周期タスクの応答時間を可能な限り小さくするスケジューリング方式を提案することを目的とした。提案スケジューリング方式を実際の ITRON ベースのリアルタイムオペレーティングシステムに組み込んで評価した結果、周期タスクセットの CPU 使用率が 60%の場合に、非周期タスクの応答時間が最大で 30%短縮されることを確認した。

6.2　今後の課題

第5章でCPU使用率（Up = 0.6, 0.7, 0.8, 0.9）となる周期タスクと非周期タスクを用意してシミュレーションを行い、非周期タスクの応答時間を確認したが、組み込みシステムによってタスクの動作条件は異なり、また一つのシステムの中でも様々な動作条件がある。これらの条件の変化によってRTOSのタスクスケジューリングは影響されると考えられる。例えば、タスク同士の同期通信によってタスクがロック状態になることもあり、制御対象機器の特性によってタスク動作が変わることもある。すなわち、CPU使用率だけではなく他の条件によって非周期タスクの応答時間が変動するため、本研究で提案する方式が実際に使用されるためには幅広い評価を行わなければならない。今後は、提案する方式を実際の稼働システム上で評価する予定である。

また、最近、高性能化への要求に応えるために組み込みシステムにおいてマルチコアが一般的に使わる見通しである。このため、提案する方式がマルチコアにおいて有用であることを示していくことが今後の課題として挙げられる。

謝辞

本研究を進めるにあたり、終始熱心かつ寛容なご指導をいただきました、田中清史准教授に心から感謝致します。

また、中間審査時や学位申請書の提出時に適切なご指摘をしていただいた田中 清史准教授、井口 寧教授、金子 峰雄教授に深く感謝致します。

参考文献

[1] C.L.Liu, J.W.Layland, “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment,” Journal of the Association for Computing Machinery, Vol.20, No.1, pp.46-61, 1973.

[2] M.Spuri, G.Buttazzo, “Efficient Aperiodic Service Under Earliest Deadline FirstScheduling,” Proc. of IEEE Real-Time Systems Symposium, pp.2-11, 1994.

[3] G.C.Buttazzo, “HARD REAL-TIME COMPUTING SYSTEMS,” 2nd edition, Springer, 2005.

[4]坂村健, “ITRON・μITRON 標準ハンドブック”, 初版, パーソナルメディア株式会社, 1990.

[5] ITRON Committee, TRON ASSOCIATION. μITRON4.0 Specification Ver.4.00.00.

[6] Qing Li/Caroline Yao, “Real-Time Concepts for Embedded Systems,” CMP Books.

[7] Andrew S. Tanenbaum, “Operating Systems Design and Implementation,” 3rd edition, Prentice Hall, January 04, 2006

[8] K.Tanaka, “Real-Time Adaptive Task Schedulint,” Proc. of International Conference on Embedded Systems and Applications (ESA’05), pp.24-30, 2005.

[9] SPARC International, Inc. “The SPARC Architecture Manual Version 8,” Prentice-Hall Inc., 1992.

[10] J.P.Lehoczky, L.Sha, J.K.Strosnider, “Enhanced Aperiodic Responsiveness in Hard Real-Time Environments,” Proc. of IEEE Real-Time System Symposium, pp.261-270, 1987.

[11] B.Sprunt, L.Sha, J.Lehoczky, “Aperiodic Task Scheduling for Hard-Real-Time Systems,” Journal Real-Time Systems, Vol.1, No.1, pp.27-60, 1989.

[12] J.P.Lehoczky, S.Ramos-Thue, “An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems,” Proc. of IEEE Real-Time Systems Symposium, pp.110-123, 1992.

[13] L.Abeni, G.Buttazzo, “Integrating Multimedia Applications in Hard Real-Time Systems,” Proc. of IEEE Real-Time Systems Symposium, pp.4-13, 1998.

[14] S.Rho, B.K.Choi, R.Bettati, “Design Real-Time Remote Method Invocation: A Server-Centric Approach,” Proc. of Intl. Conf. on Parallel and Distributed Computing Systems, pp.269-276, 2005.

[15] T.Lundqvist, P.Stenström, “Timing Anomalies in Dynamically Scheduled Microprocessors,” Proc. of IEEE Real-Time Systems Symposium, pp.12-21, 1999.

[16] R.Wilhelm, J.Engblom, A.Ermedahl, N.Holsti, S.Thesing, D.Whally, G.Bernat, C.Ferdinand, R.Heckmann, T.Mitra, F.Mueller, I.Puaut, P.Puschner, J.Staschulat, P.Stenström, "The Worst-Case Execution Time Problem – Overview of Methods and Survey of Tools," ACM Trans. Embedded Computing Systems, Vol.7, No.3, pp.1-53, 2008.

[17] A.Silberschatz, P.B.Galvin, G.Gagne, "Operating System Concepts," John Wiley&Sons, Inc., 8th edition, 2009.