組込みリアルタイムアプリケーション統合のための階層型スケジューリング

松原　豊

組込みリアルタイムアプリケーション統合のための階層型スケジューリング

要旨

近年，ハードリアルタイム性を要求される組込みシステムの分野においても，ソフトウェアの大規模化，複雑化が著しく，その信頼性をどのように確保・向上させるかが大きな課題となっている．ハードリアルタイム性を要求される代表的な組込みシステムである自動車制御システムも，低燃費化，走行性能の向上，排気ガス規制への対応などの目的から，大規模化，複雑化が進んでいる．自動車に搭載される制御用コンピュータを ECU（Electronic Control Unit；電子制御ユニット）と呼ぶが，1 台の自動車に搭載される ECU の数は，多いもので，1998 年では 30 個程度であったが，2003 年には 70 個，2010 年には 100 個程度まで増加しているといわれている．ECU の数は，さらに増加する傾向にあり，コストの増加や ECU 搭載スペースの不足という問題を引き起こしている．

自動車に搭載される ECU の数が増加する理由の 1 つとして，エンジン，ステアリング，ブレーキといった制御対象毎に別々の ECU が用いられ，その統合が進んでいないことが挙げられる．それに加えて，自動車制御システムに新しい機能を追加する場合にも，サプライヤから提供される ECU を，自動車メーカが制御システムにその都度追加する開発プロセスも 1 つの要因である．

自動車に搭載される ECU の数を減らすためには，複数の ECU を 1 つの ECU に統合する必要がある．しかし，ECU で動作するソフトウェア（アプリケーションと呼ぶ）が大規模，複雑化することで，ECU を統合する開発プロセスも複雑化することや，アプリケーションの開発，検証コストが増加することなどの理由により，統合が進んでいない．その背景にある技術的な問題としては，ECU に使われている OS が保護機能を持っていないことが挙げられる．OS に保護機能がないと，複数の ECU で動作しているアプリケーションを 1 つの ECU に統合する場合に，あるアプリケーションの不具合により，別のアプリケーションが障害を引き起こす可能性がある．その結果，アプリケーションの信頼性が低下するだけでなく，統合した状況において発生した障害の切り分けが困難になる．

これまで，組込みリアルタイムアプリケーションの統合を容易に実現するための保護機能を持つリアルタイム OS（RTOS と呼ぶ）が開発されている．RTOS の保護機能には，メモリ保護に代表される，資源へのアクセス保護機能と，時間多

重される資源に対する時間保護機能がある．これまで提案された時間保護機能は，統合後のアプリケーションのリアルタイム性を保証しないものが多く，アプリケーション統合における検証の負荷を低減させる標準的な技術にはなっていない．

本論文では，分散制御システムにおいて，個別のプロセッサで統合前に固定優先度ベーススケジューリングでスケジュール可能であったアプリケーションを，1つのプロセッサに統合して動作させる状況を想定し，アプリケーション単位でプロセッサ時間を保護する階層型スケジューリングアルゴリズムを提案する．提案するアルゴリズムを，既存の RTOS に実装して処理時間を評価することで，提案アルゴリズムの実現可能性と実用性を明らかにする．

本論文の具体的な内容は，次の 4 つである．

1つ目に，時間保護の機能要件を明確にするため，スケジューリングアルゴリズムが満たすべき要件を 3 つに整理し，それらの要件を満たす階層型スケジューリングアルゴリズム（時間保護アルゴリズム 1 と呼ぶ）を提案する．時間保護アルゴリズム 1 は，2 階層のスケジューリングアルゴリズムである．各アプリケーションのタスクをスケジュールするローカルスケジューラは，固定優先度ベーススケジューリングを，アプリケーションをスケジュールするグローバルスケジューラは EDF（Earliest Deadline First）を採用している．このアルゴリズムは，タスクの起動時刻のタイミングで，デッドラインまでに使用可能なプロセッサ時間（バジェットと呼ぶ）を割り当てる．したがって，タスクの正確な実行時間の情報を必要とせず，タスクの起動時刻と相対デッドラインの 2 つの情報が得られることを前提とする．時間保護アルゴリズム 1 の詳細と，具体的なスケジュール例を示し，時間保護の 3 つの要件を満たすことを証明する．

2つ目に，タスクの起動時刻が不明なアプリケーションにも対応するために，BSS（Bandwidth Sharing Server）アルゴリズムをベースに時間保護アルゴリズム 1 を拡張したアルゴリズム（時間保護アルゴリズム 2 と呼ぶ）を提案する．時間保護アルゴリズム 2 は，時間保護アルゴリズム 1 のローカルスケジューラに対してタスクの起動を遅延する仕組みを加え，グローバルスケジューラに対して BSS アルゴリズムをベースにしたバジェット管理手法を導入したものである．タスクの起動時刻が不明なアプリケーションにも適用できるため，時間保護アルゴリズム 1 と比べて，より多くのアプリケーションに時間保護機能を適用できる．しかしながら，実行時の CPU 負荷により，実行時間が変化するよう QoS 制御されたタスクを含むアプリケーションに対しては，時間保護を実現できないという制限がある．時間保護アルゴリズム 2 を用いてアプリケーションを統合することで，統合前に固定優

先度スケジューリングによりスケジュール可能なアプリケーションは，タスクの優先度を変更することなく，統合後にもスケジュール可能であることを証明する．

3つ目に，提案した階層型スケジューリングアルゴリズムの実現可能性を確認し，性能を評価するための柔軟なスケジューリングフレームワークを開発する．満たすべき時間要件や，統合段階で既知であるパラメータはアプリケーションごとに異なる場合が多い．そのため，単一の階層型スケジューリングで多様なアプリケーションに対応することは困難である．このフレームワークでは，提案した階層型スケジューラの共通点を整理し，ローカルスケジューラとグローバルスケジューラ間の標準インタフェースを定義する．このインタフェースを用いると，複数のスケジューリングアルゴリズムを同一の RTOS に実装できるようになる．RTOS に複数のアルゴリズムを実装できると，アプリケーション開発者は，アプリケーションごとに適切なアルゴリズムを選択できるようになる．提案フレームワークを用いて，時間保護アルゴリズム 1 に加えて，アプリケーションを周期的に起動するアルゴリズムの 2 種類を実装する．性能評価として，同一アプリケーション内でのタスク切替時間と，異なるアプリケーション間でのタスク切替時間を測定する．その結果，提案フレームワークを用いて実装した RTOS が実用可能な範囲のオーバヘッドで，実装可能であることを明らかにする．

4つ目に，これまで提案した階層型スケジューリングに対して，容易に追加可能な割込み処理スケジューリングアルゴリズムを 2 つ提案する．これらのアルゴリズムにより，タスクに加えて，割込み処理が含まれるアプリケーションにも対応できるようになる．1 つ目のアルゴリズムは，割込み処理をスケジュールする際に，予め設定された，割込み処理の優先度を，すべての実行可能な処理（すなわち，割込み処理とタスク）と比較するアルゴリズム（グローバル固定優先度スケジューリングと呼ぶ）である．2 つ目は，割込み処理の優先度を，所属するアプリケーションの実行可能な処理に限定して比較するアルゴリズム（ローカル固定優先度スケジューリングと呼ぶ）である．既存プロセッサのもつ割込みコントローラでは，発生した割込みの優先度が，実行中の処理の優先度よりも高い場合に，即座に割込み処理を実行するものが多い．したがって，グローバル固定優先度スケジューリングは，既存の割込みコントローラを用いて容易に実装可能である．それに対して，ローカル固定優先度スケジューリングをそのまま実装するのは難しい．そこで，ローカル固定優先度スケジューリングを既存の割込みコントローラでも容易に実装できるようにするため，割込み処理を，割込みハンドラと割込みサービスタスクの 2 つに分割した割込み処理モデルを提案する．このモデルを使用すること

で，割込みハンドラの処理時間を短く抑えることができ，割込み処理の本質的な部分である割込みサービスタスクを通常のタスクと同様に扱うことが可能となる．提案する 2 つのアルゴリズムをスケジューリングフレームワークを用いて実装し，割込み応答性と，割込み処理起動時のオーバヘッドを評価する．そして，これら 2 つの手法を，アプリケーションの独立性，実装容易性，割込み応答性能等の観点で比較する．さらに，割込み禁止区間を含むアプリケーションを統合する場合のスケジュール可能性を解析する手法を提案する．この解析手法を用いることで，システムの設計段階で，アプリケーションの詳細な情報を知ることなく，アプリケーション統合の可否を判定することができる．

本研究では，まず，ハードリアルタイム性を要求されるアプリケーションを統合する際の課題を指摘し，それらを解決する時間保護の要件を明確に定義した．それらの要件を満たす階層型スケジューリングアルゴリズムとして，時間保護アルゴリズム 1 と 2 を提案し，時間保護の要件を満たすことを証明した．さらに，提案したアルゴリズムを含め，多様なアルゴリズムを同一の RTOS に実装するためのスケジューリングフレームワークと，割込み処理を含むアプリケーションに対応するためのスケジューリングアルゴリズムを提案した．提案したアルゴリズムを実際に実装し，実用性の高いアルゴリズム及び RTOS を実現可能であることを明らかにした．これらの成果が，組込みリアルタイムアプリケーションの統合を促進することにつながり，さらに RTOS 技術の発展に貢献することを期待する．

A Hierarchical Scheduling for Integrating Embedded Real-time Applications

Abstract

In the recent years, scale and complexity of embedded real-time applications have been rapidly growing. The automotive control system, which is a typical embedded system with hard real-time constraints, have been just in the situation to satisfy various requirements such as low fuel consumption, low exhaust emission and driving stability. Therefore, reliability of the applications is one of the most important challenges in the real-time systems development. The number of ECUs (Electronic Control Units) have been continually increasing. For example, in the case of a luxury automobile, the number of ECUs was about 30 in 1998. It became about 70 in 2003; moreover it becomes about 100 in 2010. This trend will continue in the future. The increase of ECUs directly affects cost of production and system development. It also causes lack of necessary space for placing the ECUs with the wiring.

One of the reasons for increasing the number of ECUs in automotive control systems is that each ECU has been individually developed. To reduce the number of ECUs, it is necessary to integrate independently developed real-time applications into one ECU. However, integration of them has been rarely implemented. Because a real-time OS (RTOS) used in ECUs does not support protection functions such as memory protection and execution time protection, a failure in an application affects behavior of another applications in same ECU. Therefore, it is difficult to find an underlying causes of deadline misses on shared ECU.

In this research, we target Application-specific Execution Time Protection (AETP) for a shared processor that is the most important resource shared by multiple applications. We propose new two-level hierarchical scheduling algorithms for AETP. In order to confirm the realizability and practicality of them, a prototype of a scheduling framework we also proposed is developed based on a open-source RTOS. In this thesis, we describe following four contributions.

The first contribution is that three requirements for AETP that hierarchical scheduling algorithms should satisfy are defined. The most characteristic one of the

requirements is to guarantee that a real-time application that was schedulable in a dedicated ECU by fixed-priority scheduling (FPS) is also schedulable in a shared ECU. Then we propose a basic hierarchical scheduling, called Application-specific Execution Time Protection Scheduling algorithm 1 (AETPS1), that can satisfy the all requirements. The AETPS1 is a two-level hierarchical scheduling algorithm. The scheduler of the AETPS1 consists of local schedulers and a global scheduler. Each local scheduler corresponding to an integrated application, schedules tasks of the application by FPS. The global scheduler determines which an application should be executed in a shared processor by EDF. At release time of each task, the AETPS1 allocates execution time, called "budget", to the application the task belongs to. The budget of an application means upper bounded time to execute tasks of the application until the earliest time among release time and absolute deadline of them. Therefore, the AETPS1 needs information on release time and relative deadline of all tasks. We show details of the AETPS1 and prove that the AETPS1 satisfies all requirements for AETP.

The second contribution is to propose Application-specific Execution Time Protection Scheduling algorithm 2 (AETPS2). In order to support an application that includes a task whose release time cannot be known a priori, we extends the AETPS1 based on the Bandwidth Sharing Server (BSS) algorithm. We add a delayed activation of tasks mechanism to local scheduling of the AETPS1, and introduce budget management mechanism based on BSS to global scheduling of the AETPS1. The AETPS2 can integrate practical applications because the AETPS2 needs only information of relative deadline of tasks (i.e., information of release time of tasks is not required). However, the AETPS2 cannot support applications including QoS controlled tasks, whose execution time changes depend on processor loading at run-time, while the AETPS1 can support them. Simulation results indicate that if an application is schedulable in FPS, then the application is in isolation also schedulable without any modifications of priorities of tasks when it is integrated with other applications into the same processor by the AETPS2.

The third contribution is to propose a flexible scheduling framework in order to implement the AETPS1 and other hierarchical scheduling algorithms. It is difficult to schedule diverse applications by a single hierarchical scheduling algorithm because timing constraints and timing parameters known at integration phase,

such as release time, worst-case execution time and deadline, vary in each application. We define standard interface functions between local schedulers and a global scheduler. By using the interface functions, multiple scheduling algorithms can be implemented to one RTOS. At the RTOS configuration, application developers can choose an adequate scheduling algorithm. As a performance evaluation, execution time of task scheduling in a same application and between different applications are measured. The results of the evaluation indicate that the framework can be implemented based on an open source RTOS with a little overhead.

The fourth contribution is to present two kinds of scheduling algorithms for interrupt routines in a two-level hierarchical scheduling. One algorithm is a global FPS (GFPS). The other one is a local FPS (LFPS). In the GFPS, when an interruption occurs, the priority of an interrupt routine assigned to the interruption is compared to the priorities of all of runnable interrupt routines and runnable tasks in the system. In general interrupt controllers, an interrupt routines assigned to an occurring interruption is immediately executed if the priority of the interrupt routine is higher than the priority of a running interrupt handler or a running task. Therefore, the GFPS can be easily implemented with conventional hierarchical scheduling on the general interrupt controllers. On the other hand, in the LFPS, the priority of the interrupt handler is locally compared to all runnable interrupt routines and runnable tasks in the same application. In order to easily implement the LFPS on the general interrupt controllers, we introduce a divided interrupt routine model. In this model, an interrupt routine is divided into an Interrupt Handler (IH), providing device-depended service, and an Interrupt Service Task (IST), providing application-specific service as a main part of the interrupt routine. By using this model, execution time of the IH can be minimized and then negligible in many cases. The IST is treated as a task by global EDF scheduling. We implement the GFPS and the LFPS on the scheduling framework to evaluate the response time and overhead in activating an interrupt routine. Furthermore, we analyze the schedulability of the applications including interrupt-disabled time, and then a simple schedulability test method is proposed. This method helps system integrators such as automotive manufacturers to easily determine which applications can be integrated to the system without detail knowledge of each application in the system design phase.

In this research, we explained problems in integrating of real-time applications. In order to solve them, three requirements for AETP were defined. We proposed two hierarchical scheduling algorithms, which are called the AETPS1 and the AETPS2. The proposed flexible scheduling framework allows application developers to choose the adequate scheduling algorithm for necessary AETP. The results of the evaluation indicate that the scheduling framework can be implemented with a little overhead. To support interrupt routines, the scheduling framework was extended. As conclusions, the results of this research indicate that the proposed scheduling algorithms and the RTOS based on the proposed framework can facilitate integration of real-time applications in practical embedded system development.

目 次

図 目 次

表 目 次

第1章　序論

1.1　研究の背景

特定の機能を実現するために，機械や電子機器に組み込まれるコンピュータシステムは，組込みシステムと呼ばれる．組込みシステムの多くは，処理結果の正しさに加えて，特定の時刻（デッドライン）までに処理結果が出力されることを要求されるリアルタイムシステムである．処理がデッドラインまでに完了しない場合，システムの基本的動作や人命に対して，致命的な影響を及ぼす可能性のあるシステムはハードリアルタイムシステムと呼ばれる．

従来の組込みハードリアルタイムシステムは，機械技術や電子技術を中心に構築され，制御ロジックが単純であったことから，システムに搭載されるソフトウェアの規模は小さかった．そのため，テストや妥当性検証によりソフトウェアの不具合を防ぐことが現実的に可能であり，仮に不具合があったとしても，その不具合がシステム全体に与える影響は限定的であった．しかし近年，リアルタイム性を要求されない汎用システムだけでなく，ハードリアルタイム性を要求される組込みシステムの分野においても，ソフトウェアの大規模化・複雑化が著しく，その信頼性をどのように確保・向上させるかが大きな課題となっている．

ハードリアルタイム性を要求される代表的な組込みシステムのひとつに自動車制御システムがある．自動車制御システムは，低燃費化，走行性能の向上，排気ガス規制への対応などの目的により，システムの大規模化・複雑化が進んでいる．自動車に搭載される制御用のコンピュータをECU（Electronic Control Unit；電子制御ユニット）と呼ぶが，現在は，ECUと，そのECU上で動作する制御ソフトウェアをサプライヤが一体で開発し，自動車メーカがそれらを組み合わせることで，複雑な制御システムを実現している．

システムに新しい機能を追加する場合，センサやアクチュエータへのインタフェース，それらを制御するマイコン，制御ソフトウェアが一体になったECUを新しく追加する必要がある．自動車制御システムの大規模化・複雑化により，1台の自動車に搭載されるECUの数は，一部の高級車など多いもので，1998年では30個程

度であったが，2003 年には 70 個程度，2010 年には 100 個まで増加しているといわれている．ECU の数はさらに増加する傾向にあり，コスト増や ECU 搭載スペースの不足といった問題を起こしている．

自動車に搭載される ECU の数が増加している理由の 1 つとして，エンジン，ステアリング，ブレーキといった制御対象毎に別々の ECU が用いられ，その統合が進んでいないことが挙げられる．そのため，自動車に新しい機能を追加する度に ECU が増えていくことになる．

システムに搭載される ECU の数を削減するためのアプローチとして，個別に開発・検証された制御ソフトウェア（アプリケーションと呼ぶ）を単一の ECU に統合して動作させる（アプリケーション統合と呼ぶ）手法がある．例えば，航空機の制御システムでは，自動車制御システムの ECU に相当するコンピュータシステムは IMA （Integrated Modular Avionics）と呼ばれ，単一の IMA では，異なる機能をもつ複数のアプリケーションが統合されて動作している．IMA に搭載されるリアルタイム OS（RTOS と呼ぶ）は，各アプリケーションの実行タイミングを保証する機能を持っており，あるアプリケーションの動作が別のアプリケーションの動作に影響を及ぼすことのないよう，各アプリケーションを保護している．

現在の自動車制御システムに搭載される ECU の RTOS は，保護機能を持っていない場合がほとんどである（そもそも，RTOS を使っていないことも多い）．ECU に搭載される RTOS が保護機能を持っていないと，複数の ECU で動作しているアプリケーションを 1 つの ECU に統合する場合に，あるアプリケーションの不具合が，正常に動作している別のアプリケーションの動作に影響を及ぼし，障害を起こす可能性がある．そのため，各アプリケーションが別々に開発されている場合には，特に，発生した障害の原因や，責任の切り分けが難しくなる．このことが，自動車制御システム開発において，ECU の統合が進まない大きな理由と考える．

自動車制御システムにおいて，アプリケーション統合を容易に実現するためには，ECU に搭載される RTOS に保護機能を持たせる必要がある．自動車制御システム向けの保護機能は，航空機の制御システムとは異なり，厳しいリアルタイム制約を満たすことができること，さらに，既存の制御システムで動作しているアプリケーションが非常に多いため，これらを容易に再利用出来ることが重要となる．具体的には，統合する前のシステムにおいて，単体で正常に動作するアプリケーションが，そのタスク構成や優先度設計を変更することなく，統合後も正常に動作することを保証できることが望ましい．これが実現できると，複数のアプリケーションが統合された複雑なシステムの動作検証においても，発生する障害の原因

の切り分けが容易になり，アプリケーション統合における開発者の負担を大幅に減らすことができる．

このような背景から，ハードリアルタイム性を要求される組込みシステム向けに，保護機能を持ったRTOSが開発されている．一般的に，RTOSに求められる保護機能には，メモリ保護のように，資源へのアクセスを空間的に保護する機能に加えて，資源へのアクセスを時間的に保護する時間保護機能がある．組込みシステムにおいて，時間的に多重アクセスされる資源の中で最も重要なのがプロセッサである．プロセッサの時間保護とは，プロセッサで実行される処理が使用するプロセッサ時間を，他の処理が奪うことを防ぐ機能である．アプリケーション統合において，アプリケーション単位のプロセッサ時間保護を適用することにより，統合前に単独で実行した場合に時間制約を満たせるアプリケーションは，複数のアプリケーションを統合して実行する場合でもデッドラインを満たして実行できることを保証できる．その結果，ハードリアルタイム性を要求される組込みシステム開発におけるアプリケーション統合を促進することができる．

1.2　論文の概要

本研究の目的は，分散制御システムにおいて，個別のプロセッサで単独動作するアプリケーションを，1つのプロセッサに統合して動作させる状況を想定し，プロセッサの時間保護を実現する階層型スケジューリングアルゴリズムを提案することである．アルゴリズムの検討においては，実際に既存のRTOSに実装して処理時間を評価することで，アルゴリズムの実現可能性と実用性を明らかにする．ここで，階層型スケジューリングアルゴリズムとは，タスクやアプリケーションなどの処理の実行順序を決定する機構（スケジューラ）を，階層的にもつスケジューラを前提とするスケジューリングアルゴリズムである．本論文で提案するアルゴリズムのスケジューラは，各アプリケーションのタスクをスケジュールするローカルスケジューラと，アプリケーションをスケジュールするグローバルスケジューラを2階層に配置したスケジューラである．ローカルスケジューラは，統合前のアプリケーションの振る舞いを再現する機能をもつ．グローバルスケジューラは，各アプリケーションのタスクがデッドラインを満たすように実行するアプリケーションを決定し，かつアプリケーションの振る舞いが，別のアプリケーションの動作に影響を及ぼさないようアプリケーションを保護する機能をもつ．このように，2階層の階層型スケジューラは，アプリケーション統合に適するスケジューラ構成である．

本研究の具体的な内容は，次の 4 つである．

1 つ目は，時間保護の機能を明確に分類するために，スケジューリングアルゴリズムが満たすべき要件を，(1) アプリケーションのプロセッサ利用率が保護されること，(2) タスクがデッドラインを満たすことを保証されること，(3) システムの負荷に応じて処理内容が変化するタスク（QoS 制御されたタスクと呼ぶ）を含むアプリケーションに対しても (2) の要件を満たすことの 3 つに整理し，これらの要件をすべて満たす階層型スケジューリングアルゴリズム（時間保護アルゴリズム 1 と呼ぶ）を提案する．これまで提案されているアルゴリズムで要件 (3) を満たせるものは少ないが，時間保護アルゴリズム 1 は，この要件も満たせることが特徴である．時間保護アルゴリズム 1 は，統合前にプリエンプティブな固定優先度スケジューリングによりスケジュール可能であったアプリケーションを単一のプロセッサに統合することを想定しており，ローカルスケジューラはプリエンプティブな固定優先度スケジューリングを，グローバルスケジューラは EDF（Earliest Deadline First）スケジューリングを採用する．時間保護アルゴリズム 1 を適用するための前提条件は，タスクの正確な実行時間の情報を必要とせず，タスクの起動時刻と相対デッドラインの 2 つの情報が得られることである．アルゴリズムの詳細な動作と具体的な動作例を示し，時間保護アルゴリズム 1 が時間保護の 3 つの要件を満たすことを証明する．

2 つ目は，起動時刻が不明なタスクを含むアプリケーションへの対応である．リアルタイムアプリケーションは，外部イベントが発生したタイミングで動作を開始するというような処理を含む場合が多く，このような処理の正確な起動時刻を，事前に把握することは困難である．時間保護アルゴリズム 1 では，適用するための条件として，タスクの起動時刻とデッドラインの 2 つの情報が得られることを前提としており，このようなアプリケーションに適用することができない．そこで，タスクのデッドライン情報のみを使って時間保護の要件を満たすスケジューリングアルゴリズム（時間保護アルゴリズム 2 と呼ぶ）を提案する．時間保護アルゴリズム 2 でも，統合前にプリエンプティブな固定優先度スケジューリングによりスケジュール可能であったアプリケーションを単一のプロセッサに統合することを想定している．時間保護アルゴリズム 2 の特徴は，適用するために必要な情報が時間保護アルゴリズム 1 より少なくてすむが，その一方で，時間保護の要件 (3) を満たすことができない．そのため，QoS 制御されたタスクが含まれないアプリケーションにのみ適用できる．時間保護アルゴリズム 2 の詳細な動作を示し，タスクのスケジューリングに特化したシミュレータを用いてアルゴリズムの正当性を確認

する．さらに，時間保護の要件 (1) と (2) を満たせることを証明する．

3つ目は，スケジューリングフレームワークの提案である．このフレームワークは，階層型スケジューラの構成を整理し，上位のスケジューラと下位のスケジューラ間の共通インターフェースを定義したものである．階層型スケジューラの構成が，特定のアルゴリズムに強く依存しないため，このフレームワークに，複数の異なるスケジューリングアルゴリズムを実装できる．したがって，アプリケーションの時間要件や統合段階で既知のパラメータに応じて，アプリケーションごとに適切なアルゴリズムを選択できるようになる．提案フレームワークに，アプリケーションを周期的に起動するアルゴリズムと，時間保護アルゴリズム1の2種類を実装し，動作時の処理オーバヘッドを測定する．その結果，実用上許容できる程度のオーバヘッドで実現できることを明らかにする．

4つ目に，タスクと割込み処理で構成されるアプリケーションも統合することを目的として，階層型スケジューリングアルゴリズムにおける割込み処理のスケジューリングアルゴリズムを2つ提案する．1つ目のアルゴリズムは，割込み処理の優先度をすべての実行可能な処理（割込み処理とタスク）と比較してスケジュールするグローバル固定優先度スケジューリングである．このアルゴリズムは，既存の割込みアーキテクチャと従来の階層型スケジューリングに容易に実装可能である．2つ目は，割込み処理の優先度を比較する対象を，所属するアプリケーションの実行可能な処理に限定するローカル固定優先度スケジューリングである．既存の割込みアーキテクチャでは，発生した割込み要因に対応する割込み処理の優先度が，実行中の処理の優先度よりも高い場合は，即座に割込み処理が実行されるため，このアルゴリズムを実装することは難しい．そこで，割込み処理を，割込みハンドラと割込みサービスタスクの2つに分割した割込み処理モデルを提案する．このモデルを使用することで，割込み処理の本質的な部分である割込みサービスタスクを，通常のタスクと同様に扱うことが可能となる．提案する2つのアルゴリズムをスケジューリングフレームワークのプロトタイプに実装して，実現可能性を確認する．さらに，割込み応答性を評価し，システムに要求される時間制約により，これら2つの手法のどちらかを選択するのが適切であるかを議論する．最後に，割込み禁止区間を含むアプリケーションのスケジュール可能性を解析する手法を提案する．この解析手法を用いることで，システムの設計段階で，アプリケーションの詳細な情報を知ることなく，アプリケーションの統合の可否を判定することができる．

1.3　論文の構成

本論文の構成を述べる．第 1 章では，本研究の背景と概要について述べた．第 2 章では，本研究の前提となる事項について，リアルタイムスケジューリングと時間保護の要件を中心に整理する．さらに，本研究に関連する技術と従来研究について述べる．第 3 章では，時間保護を実現する基本的なスケジューリングアルゴリズムである時間保護アルゴリズム 1 について述べる．第 4 章では，時間保護アルゴリズム 1 を改良し，起動時刻が不明なタスクを含むアプリケーションに対応したスケジューリングアルゴリズムである時間保護アルゴリズム 2 について述べる．第 5 章では，複数のスケジューリングアルゴリズムを同一の RTOS に実装することを目的として開発したスケジューリングフレームワークについて述べる．第 6 章では，割込み処理を含むリアルタイムアプリケーションに対応したスケジューリングアルゴリズムについて述べる．第 7 章で，本研究の結論と今後の課題について述べる．

第2章　時間保護による組込みリアルタイムアプリケーションの統合

2.1　リアルタイムシステムとスケジューリング

2.1.1　リアルタイムシステム

コンピュータシステムにより処理した結果の価値が，結果の内容だけではなく，出力される時刻にも依存するシステムはリアルタイムシステムと呼ばれる．すなわち，リアルタイムシステムには，正しい処理結果が得られることに加えて，その処理結果が特定の時刻（デッドラインと呼ぶ）までに得られることが要求される．リアルタイムシステムに対する要求事項には，満たすべき時間的な要求（時間制約と呼ぶ）がデッドラインとして与えられ，そのデッドラインを満たすための仕組みとそれを保証する手段が必要となる．

デッドラインまでに処理が必ず完了することを保証するためには，その処理の実行時間を予め予測できる必要がある．特に，処理の実行時間が最も長くなる場合の時間が予測できると，その実行時間と，時間制約として与えられたデッドラインとを比較することで，最悪時の実行時間がデッドラインよりも早く完了することを保証できる．このように，最悪時の処理の実行時間を保証できる処理は，リアルタイム性（もしくは予測可能性）をもつといわれる．この定義から，リアルタイム性は，処理の実行時間の長さとは独立した概念であることが分かる．

一般に，リアルタイムシステムは，処理がデッドラインを満たせなかった場合に発生する結果により，次の 3 つに分類される．

- ハードリアルタイムシステム（hard real-time system）

  処理がデッドラインまでに完了しない場合に，システムの基本的な動作や人命などに致命的な影響を及ぼすシステムである．例えば，自動車制御システ

ムのブレーキ制御では，運転者がブレーキを踏むと，ブレーキキャリパを動かすシステムに対して正しい制御信号を出力することに加えて，要求される時間以内に制御信号を出力する必要がある．

- ソフトリアルタイムシステム（soft real-time system）

  ハードリアルタイムシステムとは異なり，デッドラインまでに処理を完了することが出来ない場合でも，システム全体に致命的な影響を及ぼさないシステムである．例えば，携帯電話やカーナビゲーションシステムのユーザインタフェース処理は，入力に対して，ある程度の応答性を要求されるが，デッドラインを満たせない場合でも，システム全体に致命的な影響を及ぼすことや，利用者に対して人的被害をもたらすことは考えにくい．

- ファームリアルタイムシステム（firm real-time system）

  ハードリアルタイムシステムと同様に，厳しい時間制約をもつが，時間制約を満たせない場合でも致命的な結果は発生しないシステムである．ソフトリアルタイムシステムと異なるのは，デッドラインを過ぎた後の処理結果の価値はないと判断することにあり，このような処理結果は破棄して処理を強制終了する．例えば，音声や動画などのマルチメディア処理が該当する．音声や動画のエンコード処理は，各フレームが特定の時間内に処理されることが要求されるが，時間制約が満たされない場合，出力の品質低下が発生する．処理がデッドラインを過ぎた場合には，その計算結果には価値がないと判断して処理を停止し，次のフレームの処理に移る．

以上より，リアルタイムシステムの処理完了時刻とその処理結果のもつ価値の関係は，次のように整理できる．デッドラインを満たして実行が完了した場合には，すべてのリアルタイムシステムにおいて一定の価値がある．それに対して，時間制約を満たせなかった場合の処理結果の価値は，大きく異なり．すなわち，ハードリアルタイムシステムでは，致命的な被害をもたらすことから，処理結果の価値は $-\infty$ と考えられる．ソフトリアルタイムシステムでは，時間経過に伴い徐々に低下していく．ファームリアルタイムシステムでは処理結果を破棄するので，0 と考えられる．

2.1.2　タスクのモデル化と分類

本論文では，システムが提供する機能を実現する処理の単位をタスクと呼ぶ．タスクは，図 2.1 に示すように，複数のパラメータでモデル化できる．各パラメータの意味は，次の通りである．

- 起動時刻（Release time）

  タスクが実行できる状態になった時刻．図中では r で表す．

- 開始時刻（Start time）

  タスクが実行を開始した時刻．図中では s で表す．

- 実行時間（Computation time）

  タスクが処理を完了するために必要なプロセッサ時間．ただし，実行を開始してから完了するまでの間で，別のタスクに実行を邪魔されている時間を除く．図中では c で表す．

- 終了時刻（Finishing time）

  タスクの実行が完了した時刻．図中では f で表す．

- 絶対デッドライン（Absolute deadline）

  タスクが実行を完了すべき絶対時刻．リアルタイム性を要求されないタスクの絶対デッドラインは，∞ と考える．図中では d で表す．

- 相対デッドライン（Relative deadline）

  絶対デッドライン d と起動時刻 r の差の時間．すなわち，タスクの起動時刻から，そのタスクの実行が完了するべき時刻までの相対時間である．一般に，タスクの相対デッドラインは，タスクの処理内容に対する時間制約の 1 つとして設計者が定める．リアルタイム性を要求されないタスクの相対デッドラインは，∞ と考える．図中では D で表す．

- 応答時間（Response time）

  タスクの起動時刻から終了時刻までの時間．図中では R で表す．

一般に，リアルタイムシステムのタスクを分類する場合には，次の 2 つの観点で分類されることが多い．1 つ目の観点は，タスクが絶対デッドラインまでに実行を完了できなかった場合にシステムへ与える影響の度合である．

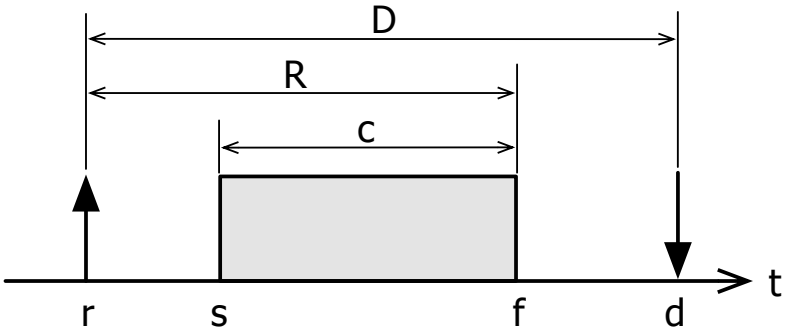

図 2.1: タスクのモデル

- リアルタイムタスク（real-time task）

  タスクが実行を完了すべき時刻が明確に決まっており，それが時間要件として与えられているタスクである．時間制約としては，相対デッドラインが指定されることが多い．リアルタイムタスクは，デッドラインを満たせなかった場合に発生する影響により，ハードリアルタイムタスクとソフトリアルタイムタスクに分類できる．ハードリアルタイムタスクは，タスクの実行がデッドラインまでに完了しない場合に，システムの動作に致命的な影響を及ぼすタスクである．ソフトリアルタイムタスクは，タスクの実行がデッドラインまでに完了しない場合に，システムの性能や出力の質が低下するタスクである．ただし，システムの基本動作に致命的な影響は及ぼさない．本論文では，ハードリアルタイムタスクとソフトリアルタイムタスクを区別せず，リアルタイムタスクと呼ぶ．

- 非リアルタイムタスク（non real-time system）

  タスクが実行を完了すべき時刻が明確に定められていない（相対デッドラインが ∞ である）タスクである．非リアルタイムタスクは，デッドラインが定められていないことから分かるように，実行時間が実行される状況や入力により大きく変動するため，最悪時の実行時間を予測しにくい場合が多い．例えば，システムの故障を診断するタスクのように，プロセッサがアイドル状態になると，不定期に診断処理を処理を実行するものがある．

2 つ目の観点として，タスクの起動時刻の規則性から，次のように分類する．

- 周期タスク（periodic task）

  起動する間隔が一定のタスク．

- 非周期タスク（aperiodic task）

  起動する間隔が一定ではないタスク．非周期タスクのうち，起動する間隔の最小値（最小起動間隔）が定まるタスクを，離散タスク（sporadic task）と呼ぶこともある．

代表的な理論体系である Rate Monotonic Analysis（RMA）では，すべてのタスクが周期タスクであることを制約条件の 1 つとしている．システムを周期的な処理の枠組みで捉えることができると，スケジュール可能性解析の適用が容易になる．

一般に，センサによる外部情報の取得や，アクチュエータの制御などでは，ある一定間隔で外部デバイスを操作する処理があり，それらは周期タスクとしてモデル化できる．外部イベントにより起動し，実行中の処理に優先して実行する処理（割込み処理と呼ぶ）は，処理開始の起点となる外部割込みの発生を予め予測することが困難であるため，非周期タスク（割込みの発生する最小間隔が分かる場合には，離散タスク）としてモデル化できる．

2.1.3　対象とするアプリケーション

本論文では，1 つ以上のタスクで構成されるタスクの集合をアプリケーションと呼ぶ．基本的には，アプリケーションに割込み処理は含まれないものとする（割込み処理が含まれるアプリケーションは，第 6 章で扱う）．

アプリケーションは，リアルタイムタスクのみで構成されるアプリケーションと，リアルタイム性を要求されるタスクに加えて，非リアルタイムタスクも混在しているアプリケーションの 2 つに分類できる．詳細は 2.2.2 項で述べるが，非リアルタイムタスクがアプリケーションに含まれると，非リアルタイムタスクの動作がリアルタイムタスクの動作に影響を及ぼすことがある．その結果，リアルタイムタスクの予測可能性が低下することになる．この理由から，リアルタイムタスクのみで構成されるアプリケーションに比べて，非リアルタイムタスクの含まれるアプリケーションのリアルタイム性を保証することは難しい．

タスク起動時刻の規則性の観点では，周期タスクのみで構成されるアプリケーションと，周期タスクに加えて非周期タスクが混在しているアプリケーションの 2 つに分類できる．非周期タスクがアプリケーションに含まれると，タスクが起動

する時刻を正確に把握できないため，アプリケーションの処理負荷変動を予測することが難しくなる．この問題に対しては，例えば，離散タスクが常に最小起動間隔で起動するという悲観的な前提を置くことで，負荷の予測可能性を確保できるが，その解析結果も現実に比べて悲観的な結果となってしまう．このことから，周期タスクのみで構成されるアプリケーションに比べて，非周期タスクも含まれるアプリケーションのリアルタイム性を保証することは難しい．

アプリケーションに含まれるタスクの処理時間の予測可能性と，起動時刻の予測可能性が，アプリケーションのリアルタイム性を保証する上で重要であることを述べた．実システムで動作するアプリケーションは非常に複雑であるため，処理時間と起動時刻の両方の予測可能性をもつアプリケーションは非常に少ない．そこで，本論文では，まず，第 3 章では，すべてのタスクの起動時刻と相対デッドラインが分かることを前提として，非リアルタイムタスクを含むアプリケーションを扱う．次に，第 4 章では，タスクの起動時刻が分かることを前提とせず，相対デッドラインのみが分かることを前提として，リアルタイムタスクのみで構成されるリアルタイムアプリケーションを扱う．

2.1.4　リアルタイムスケジューリング

初期のリアルタイムシステムでは，ソフトウェアに要求される機能が比較的少なかったことから，そのリアルタイム性を保証することも容易であった．例えば，ソフトウェアで実現すべき処理を機能毎に関数に分け，1 つのループ処理内にそれらを順番に呼び出すことで実現している場合がある．このようなシステムのリアルタイム性を保証する基本的な方法として，最悪時の実行時間を測定する．すわなち，関数毎の実行時間が最も長くなる状況を想定し，各関数の最悪実行時間の合計時間をループ処理全体の実行時間と考え，それが満たすべきデッドライン以内であることを確認する．

システムに要求される機能が増加し大規模化してくると，機能を複数のタスクに分割し，それらを並列に実行する必要が出てきた．複数のタスクが存在する場合には，タスクをどの順番で実行するかを決定するアルゴリズムが必要となり，数多くのタスクスケジューリングアルゴリズムが提案された．

タスクスケジューリングアルゴリズムの目的は，汎用システムでは主にプロセッサのスループットを高めることにあるのに対して，リアルタイムシステムでは各タスクのリアルタイム性を保証し易いことに主眼が置かれる．スケジュールする

タスクの集合が与えられたときに，それらが特定のスケジューリングアルゴリズムによりリアルタイム性を保証するための必要十分条件や，スケジュール可能な，プロセッサ利用率の上限について理論的な研究が行なわれた．これがリアルタイムスケジューリング理論である．

1970 年代は，特に時代背景などの理由から，ハードリアルタイムシステムのリアルタイム性に関する古典的なスケジューリング理論の研究が盛んに行なわれた．その後は，古典的な理論をベースに，対象とするタスクモデルの制約条件を緩めることにより，さまざまな状況下で理論を適用できるよう拡張されてきた．2000 年代になると，リアルタイムシステムのハードウェアアーキテクチャの進化に伴い，シングルプロセッサだけでなく，数個のプロセッサコアをもつマルチコアプロセッサや，さらに多くのプロセッサコアをもつメニーコアプロセッサを対象としたスケジューリングアルゴリズムも活発に研究されている．

リアルタイム性の保証を困難にする要因は，ソフトウェアの大規模化だけではない．すなわち，キャッシュの影響，バス競合，分岐予測など，コンピュータシステムのアーキテクチャの進化により，ソフトウェアの実行時間を正確に予測することが困難になっている．そのため，最悪時の実行時間と平均的な実行時間との差が大きくなり易く，最悪実行時間を正確に見積もることは困難である．

本研究では，スケジューリングアルゴリズムの検討において，タスクの正確な最悪実行時間の情報が分かることを前提とせず，タスクの起動時刻や相対デッドラインなど，タスクの設計時に決定する情報のみを使用する．ただし，スケジュール可能性解析においては，古典的なスケジュール可能性解析手法と同様に，タスクの最悪実行時間を測定可能であることを前提とする．

2.1.5　対象とするスケジューリングアルゴリズム

リアルタイムスケジューリング理論研究の重要な成果の 1 つに，与えられたタスクの集合に含まれるすべてのタスクが，デッドラインを満たして実行可能であるかを判定するスケジュール可能性解析がある．現実のアプリケーションのスケジュール可能性を解析する場合には，各タスクの振る舞いに予測可能性があることと，システムの最大負荷を決定できることが重要となる．

これまで，リアルタイムシステムに用いられるタスクスケジューリングアルゴリズムは数多く提案されているが，タスクに優先度を与え，優先度の高いタスクから順番に実行していくという，優先度をベースにしたスケジューリングが一般

的である．優先度をどのように決定するかは，アルゴリズムにより異なるが，ここでは，まず，優先度割当てに用いられるパラメータが決定されるタイミングの観点で次の 2 つに分類する．

- 静的スケジューリング（static scheduling）

  タスクの優先度を静的に割当てるスケジューリング手法である．すなわち，システムが動作する前（例えば設計時）に，タスクの優先度を決定する．一旦，システムが動作を開始すると，その後は優先度が変化しないことから，固定優先度スケジューリングと呼ばれる．優先度を決定する方法としては，タスクの起動周期が短い順に高い優先度を割当てるレートモノトニックスケジューリングや，タスクの相対デッドラインが短い順に高い優先度を割当てるデッドラインモノトニックなどがある．

- 動的スケジューリング（dynamic scheduling）

  タスクの優先度をシステムの動作時に割当てるスケジューリング手法である．システムの動作中に，時間的に変動するパラメータを用いて優先度を決定するため，タスクの優先度も時間的に変動する．したがって，タスクが起動する度に，処理（ジョブ）の優先度が変わる場合がある．優先度を決定する方法としては，タスクの絶対デッドラインが早い順に，高い優先度を割り当てる EDF スケジューリングや，相対デッドラインと残り実行時間の差が小さい順に，高い優先度を割り当てる LL（Least Laxity）スケジューリングなどがある．

次に，実行中のタスクより優先度の高いタスクが実行可能になった場合の振舞に着目すると，次の 2 つに分類できる．

- ノンプリエンプティブスケジューリング（non-preemptive scheduling）

  一旦タスクの実行を開始すると，そのタスクが終了するか，イベント発生待ちなどの理由で実行を継続できなくなるまで実行を継続する．あるタスクの実行中に，実行中タスクよりも優先度の高いタスクが実行可能になっても，実行中タスクの処理を継続する．実行中タスクが実行を継続できなくなると，その時点でもっとも高い優先度をもつ実行可能なタスクを実行する．

- プリエンプティブスケジューリング（preemptive scheduling）

タスクの実行中に，実行中タスクよりも優先度の高いタスクが実行可能になると，実行中タスクの処理を一時中断し，高い優先度のタスクを実行する．高優先度タスクの実行が終了するか，イベント待ちなどで実行を継続できなくなると，その時点でもっとも高い優先度をもつ実行可能なタスクを実行する．

本研究では，プリエンプティブスケジューリングを前提としており，スケジューリングアルゴリズムの説明において，特に言及がない場合には，プリエンプティブスケジューリングを前提とする．

本研究では，主に，次の 2 つのスケジューリングアルゴリズムを対象とする．

- 静的優先度割当て固定優先度スケジューリング

  タスクに対して静的に優先度が割り当てられ，タスク実行中にタスクの優先度は変わらない．システム動作中はプリエンプティブにスケジューリングする．タスクの優先度を決定する方法は，処理内容の重要性により設計者が優先度を割り当てる方法や，起動周期の短い順に高い優先度を割り当てるレートモノトニック，相対デッドラインの短い順に高い優先度を割り当てるデッドラインモノトニックなどがある．既存の RTOS で最も多く採用されているタスクスケジューリングアルゴリズムである．

- EDF スケジューリング

  タスクの起動時と終了時において，その時点でもっとも早い絶対デッドラインをもつタスクを最高優先度のタスクとする．タスクの優先度は，時間経過により変動するため，動的スケジューリングアルゴリズムである．同じ絶対デッドラインをもつタスクが複数存在する場合には，それらのタスクの中でどのタスクを実行しても構わない．タスク実行中に，より早い絶対デッドラインをもつタスクが起動した場合には，そのタスクに実行を切り替える．

2.2　アプリケーション統合のためのスケジューリングアルゴリズム

2.2.1　アプリケーション統合の状況設定

本研究では，図 2.2 に示すように，分散リアルタイムシステムにおいて，独立に動作するコンピュータシステム上でデッドラインを満たして動作するリアルタイ

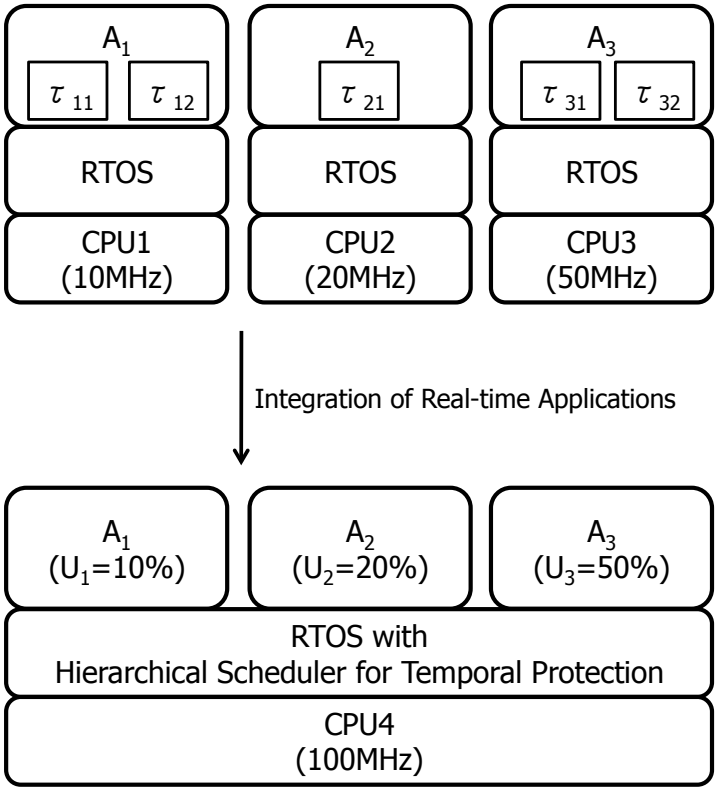

図 2.2: リアルタイムアプリケーションの統合

ムアプリケーション（A_1, A_2, A_3）を，高性能なコンピュータシステムに統合して動作させること（アプリケーション統合と呼ぶ）を想定する．各アプリケーションは 1 つ以上のタスクで構成される．ここで，1 つのアプリケーションのみを動作させるための統合前のプロセッサのことを個別プロセッサと呼ぶ．複数のアプリケーションを動作させるためのプロセッサを統合プロセッサと呼ぶ．

統合プロセッサは，個別プロセッサに比べて高い処理性能をもつ．統合プロセッサでは，各アプリケーションに対して，プロセッサ利用率を設定する．本研究では，プロセッサ性能と処理量の関係を単純化し，単位時間当たりの処理量はプロセッサの性能に比例するものとする．例えば，性能 1 のプロセッサで，2 単位時間で実行を完了するタスクは，性能 2 のプロセッサでは，1 単位時間で実行を完了できるものとする．

各アプリケーションには，統合プロセッサにおけるプロセッサ利用率（シェアと呼ぶ）と，実行可能なプロセッサ時間（バジェットと呼ぶ）の 2 つのパラメータをもつ．シェアは，アプリケーションが統合プロセッサを占有できる割合の上限であ

る．アプリケーションに設定するシェアの値は，統合プロセッサの性能に対する個別プロセッサの性能の相対値とする．例えば，統合後の統合プロセッサ上でN個のアプリケーションを動作させる場合，統合前に性能U_iの個別プロセッサで動作するアプリケーションのシェアは$U_i/\sum_{j=1}^{N} U_j$に設定する．統合するアプリケーションに設定するシェアの合計は，1以下であるものとする．バジェットは，統合後の統合プロセッサ上でそのアプリケーションを実行できるプロセッサ時間を意味する変数値である．アプリケーション内のタスクが実行されると，その実行時間分だけ減少する．別のアプリケーションに実行が切り替わると，その後に実行されたタスクの実行時間分のバジェットが，切り替わった先のアプリケーションのバジェットから減少する．

個別プロセッサで動作するアプリケーションを，統合プロセッサ上に容易に統合するためには，個別プロセッサでデッドラインを満たせるアプリケーションは，統合プロセッサへの統合後にもデッドラインを満たすことが望ましい．ここで，アプリケーションがデッドラインを満たすとは，アプリケーション内のすべてのタスクがデッドラインを満たすことをいう．タスクに与えられる時間制約は，その処理内容によって，応答時間，ジッタ，応答時間のぶれ幅など多様であるが，本研究では，タスクのデッドラインのみを対象とする．

本論文では，特に記載がない限り，次のことを前提とする．

- アプリケーションはタスクのみで構成され，割込み処理は含まれない（ただし，第6章では，割込み処理を含むアプリケーションを扱う）．
- タスクは待ち状態にならない．
- タスク間の通信はない．
- アプリケーションのタスク構成は設計時に決定され，システム動作中は変わらない．
- アプリケーションA_iは，統合プロセッサの性能を1とする相対性能U_iをもつ個別プロセッサで，固定優先度スケジューリングによりスケジュール可能である．

表 2.1: アプリケーション A のタスクセット

	起動周期	相対デッドライン	最悪実行時間	プロセッサ利用率 (%)	最悪応答時間
高優先度タスク	4	4	2	50.0	2
中優先度タスク	5	3	1	20.0	3
低優先度タスク	15	15	4	26.6%	15

2.2.2　統合後にデッドラインを満たせなくなる要因

複数のアプリケーションを 1 つの高性能なプロセッサで統合して動作させる場合，RTOS がアプリケーション間の保護機能をもたないと次のような問題が発生する．

1. あるアプリケーションの動作に時間的な不具合があった場合，その影響がシステム全体に及ぶこと
2. 高性能なプロセッサ上ですべてのアプリケーションがデッドラインを満たすことを再検証する必要があること
3. 統合後のシステムで発生したデッドラインミスの原因となったタスクを発見することが困難であること

ここでは，複数のアプリケーションを統合プロセッサに統合した結果，アプリケーションがデッドラインを満たせなくなる 3 つの要因を具体的に述べる．

1 つ目の要因は，別のアプリケーションがプロセッサ時間を使いすぎることにより，当該アプリケーションに割り当てられるプロセッサ時間が不足することである．これを防ぐためには，アプリケーション毎に利用可能なプロセッサ時間を管理し，それを超えてアプリケーションを実行しない仕組みが必要である．

2 つ目の要因は，アプリケーション内でのスケジュールが変化することである．アプリケーションが複数のタスクで構成されている場合には，アプリケーションに割り当てられるプロセッサ時間が十分であっても，タスクの実行順序が変化することで，一部のタスクがデッドラインをミスする場合がある．

この例を表 2.1 のアプリケーションを例に説明する．アプリケーション A は，3 つの周期タスクで構成され，固定優先度スケジューリングに基づいてスケジュールされる．最悪実行時間は個別プロセッサにおける実行時間，最悪応答時間は Critical Instant 定理により導かれる最悪時の応答時間である．

アプリケーション A は，性能 1 の個別プロセッサ上では図 2.3 のように実行され，デッドラインを満たせる．図中の上方向矢印はタスクの起動を，縦の太線は絶

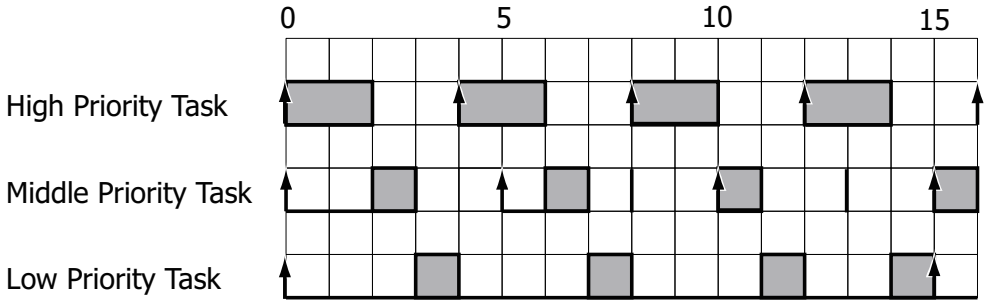

図 2.3: アプリケーション A の個別プロセッサ上での動作

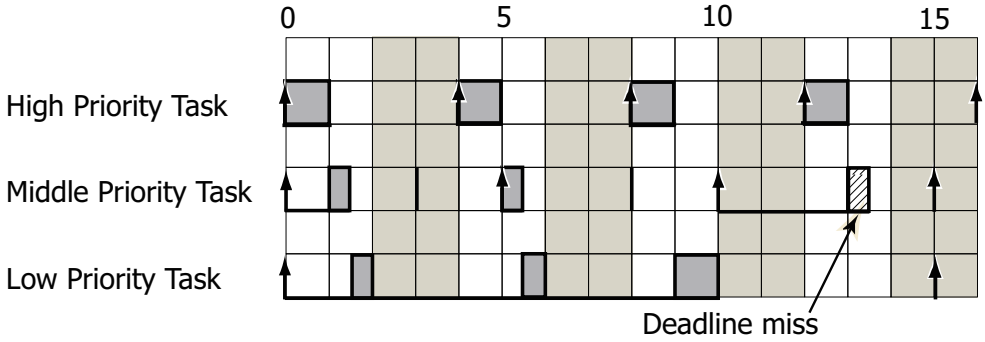

図 2.4: 統合後のスケジュール変化によるデッドラインミス

対デッドラインを（絶対デッドラインが起動時刻と同じ場合は省略），起動時刻の上矢印から横方向に伸びる下線はタスクが実行可能であることを示す．ここでは，性能 2 の統合プロセッサ上で，アプリケーション A をプロセッサ利用率 50%以内でデッドラインを満たせるようにスケジューリングできるアルゴリズムを考える．

まず，単純な方法として，高優先度タスクの起動周期である 4 単位時間毎に最大 2 単位時間のプロセッサ時間を与えるアルゴリズムを適用する．アプリケーション A は，与えられたプロセッサ時間を周期の最初から使えると仮定すると図 2.4 のように実行される（この例では濃淡部分のうち，淡い色の部分で実行可能タスクを実行する）．時刻 0 で 3 つのタスクが実行可能になると，まず高優先度タスクが実行され，個別プロセッサでの半分の時間で処理を完了する．次に，中優先度タスクが実行され，0.5 単位時間で処理を完了する．さらに，低優先度タスクが 0.5 単位時間実行される．時刻 2 になると時刻 0 で与えられたプロセッサ時間がなくなるため，アプリケーション A はこれ以上実行を継続できない．時刻 4 で，再び 2 単位時間のプロセッサ時間が得られると，この時最も優先順位の高い高優先度タスクが実行される．時刻 8 から時刻 12 の間のタスク実行順序に着目すると，高優先

表 2.2: QoS 制御されたタスクを含むアプリケーション A' のタスクセット

	起動周期	相対デッドライン	最悪実行時間	プロセッサ利用率 (%)	最悪応答時間
高優先度タスク	4	4	2	50.0	2
中優先度タスク	5	3	1	20.0	3
QoS 制御されたタスク	15	15	4(6)	26.6(40.0)	15

度タスク，低優先度タスクの順番で実行されており，個別プロセッサでスケジュールした場合（高優先度タスク，中優先度タスクの順番に実行される）と異なる．その結果，時刻 13 で中優先度タスクがデッドラインをミスしてしまう．このように，アプリケーション A が得られるプロセッサ時間は十分であるにも関わらず，タスクの実行順序が変更することにより，個別プロセッサでデッドラインを満たすタスクが，統合プロセッサではデッドラインを満たせなくなる場合がある．

これを防ぐ 1 つの方法として，条件付きではあるが，BSS アルゴリズム [23] もしくは PShED アルゴリズム [9] を適用できるといわれている[1][22]．BSS アルゴリズムと PShED アルゴリズムは，スケジューラの設計構造は異なるが，スケジューリングの結果に違いはないので，ここでは，PShED アルゴリズムに統一して表記する．PShED アルゴリズムでは，アプリケーションに対して周期的にバジェットを与えるのではなく，実行可能なタスク毎に，起動時刻から絶対デッドラインまでの間で使用可能なバジェットを管理する．タスクが使用したプロセッサ時間を，そのタスクよりデッドラインの長いタスクのバジェット計算にも反映することで，アプリケーション内でもっとも長いデッドラインまでのプロセッサ利用率をシェア以下に制限する．新しくタスクが起動すると，これまでに使用したバジェットの量に応じて，使用できるプロセッサ時間を計算して割当てる．アプリケーション内のタスクは，タスクの優先度に基づいて，固定優先度スケジューリングにより実行される．

アプリケーション A を PShED アルゴリズムでスケジュールすると，図 2.5 のように実行される．短い時間幅でのプロセッサ利用率は 50%を超えるが，時刻 0 から図中でもっとも長い絶対デッドライン（時刻 15）までの間のプロセッサ利用率は 50%となる．このように，アプリケーション A に対して PShED アルゴリズムを適用すると，アプリケーションのプロセッサ利用率を制限するだけではなく，個別プロセッサ上でのタスクの実行順序を維持し，かつすべてのタスクがデッドラインを満たすことができる．すなわち，この例では，PShED アルゴリズムは 2 つ

[1]ただし，我々の研究では，文献 [22] で提案されている条件を満たす場合でも，スケジュールできない例が見つかっている

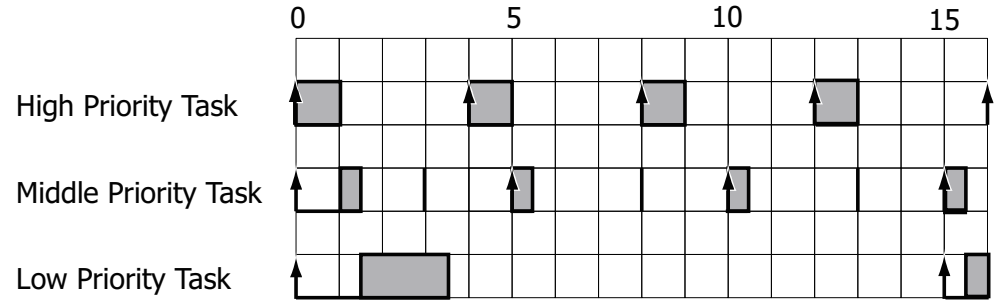

図 2.5: アプリケーション A に PShED アルゴリズムを適用した場合の動作

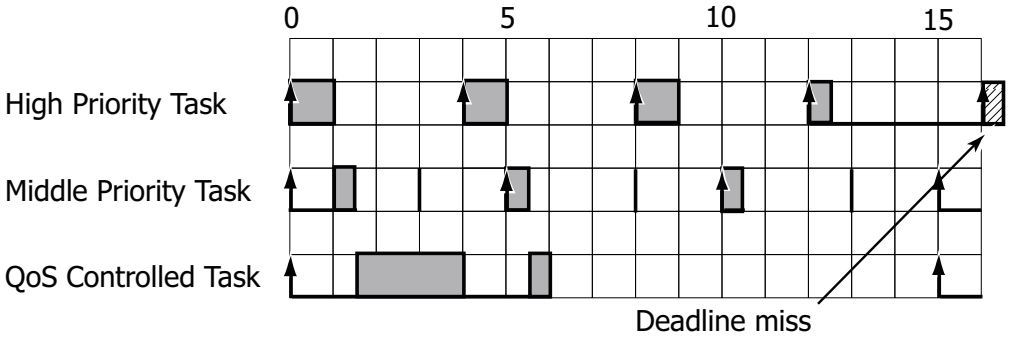

図 2.6: QoS 制御されたタスクを含むアプリケーション A' に PShED アルゴリズムを適用した場合の動作

目の要因に対応できている．しかしながら，次に説明する 3 つ目の要因には対応できない．

3 つ目の要因は，QoS 制御されたタスクの存在である．QoS 制御されたタスクは，システムの負荷に応じて処理内容を変化させている場合など，統合プロセッサで他のアプリケーションと統合して実行させると，個別プロセッサで実行したときと比べて，要求する処理量が変化するタスクである．QoS 制御されたタスクの例としては，実行中にデッドラインになったときに，他に実行可能なタスクが存在する場合には直ちに実行を終了するが，他に実行可能なタスクが存在しない場合には，さらにいくらか実行を継続する処理がある．

アプリケーション内に QoS 制御されたタスクが存在すると，PShED アルゴリズムを適用しても，統合後に一部のタスクがデッドラインを満たせない場合がある．このことを，表 2.2 に示すアプリケーション A' をスケジュールする例で示す．アプリケーション A' は，アプリケーション A の低優先度タスクを QoS 制御されたタスクに置き換えたものである．アプリケーション A' の QoS 制御されたタスクは，

アプリケーション内で最も優先度が低く，プロセッサ利用率が高い場合は 4 単位時間（個別プロセッサにおける処理時間）だけ要求するが，アイドル状態などプロセッサに余裕がある場合は 6 単位時間の処理を要求するタスクである．

図 2.6 に示すように，QoS 制御されたタスクは，時刻 1.5 から時刻 4 までと，時刻 5.5 から時刻 6 までの合計 3 単位時間実行されている．これは，将来起動するタスクのバジェットを先使いしてしまったことを意味している．バジェットの先使いにより，高優先度タスクは時刻 16 のデッドラインをミスしてしまう．QoS 制御されたタスクによる，バジェットの先使いを防ぐためには，タスクの起動時刻をスケジューラが把握し，将来起動する高優先度タスクがデッドラインを満たすために必要となるバジェットを残しておく必要がある．

2.2.3　スケジューリングアルゴリズムが満たすべき要件

アプリケーション統合により，統合プロセッサにおいて発生するデッドラインミスの 3 つの要因を分析した結果を踏まえ，統合後にすべてのタスクがデッドラインを満たすために，スケジューリングアルゴリズムが満たすべき要件を 3 つに整理する．本論文では，これらを時間保護の要件と呼ぶ．

- 要件 (1) アプリケーションのプロセッサ利用率が保護されること

  アプリケーションに所属するタスクが，ある周期ごとに，特定の割合の時間分だけ実行できることを保証する保護機能を，プロセッサ利用率の保護と呼ぶ．統合するアプリケーションのプロセッサ利用率を保護できると，統合の前後で，アプリケーションが得られる処理量が等しくなることを保証できる．

  非リアルタイムタスクのみで構成されるアプリケーションを統合する場合，各タスクは厳密なデッドラインを持たないため，アプリケーションのプロセッサ利用率を保証できれば十分である場合が多い．統合プロセッサでは複数のアプリケーションが動作するため，アプリケーションの周期の中で，タスクが実行される時刻までは保証されない．したがって，リアルタイムタスクが存在すると，そのタスクのデッドラインまでに必要なバジェットが得られるとは限らないため，デッドラインをミスしてしまう可能性がある．したがって，リアルタイムアプリケーションを統合する場合には，プロセッサ利用率が保護されるだけでは不十分である．

- 要件 (2) すべてのタスクがデッドラインを満たすことが保証されること

統合プロセッサで動作するときに，アプリケーションに所属するタスクがデッドラインを満たすことを保証することをタスクのデッドライン保証と呼ぶ．タスクのデッドラインを保証できると個別プロセッサにおけるアプリケーションの動作検証の結果は，統合後の統合プロセッサにおいても有効となる．これにより，アプリケーション統合における再検証の負担が大幅に軽減できることから，アプリケーションを統合することを目的として利用されるスケジューリングアルゴリズムは，この要件を満たす必要がある．

- 要件 (3)QoS 制御されたタスクを含むアプリケーションに対しても (2) の要件を満たすこと

  デッドラインミスが発生する 3 つ目の要因の分析から，時間保護を実現するスケジューリングアルゴリズムが満たすべき 3 つ目の要件として，QoS 制御されたタスクが存在する場合にも，個別プロセッサでデッドラインを満たすタスクは，統合プロセッサへの統合後もデッドラインを満たすことを保証することを挙げる．この要件を満たす必要があるかどうかは，アプリケーションのタスクの性質に依存する．

アプリケーション統合においては，これらの要件のうち，満たす必要のある要件を選択し，それを実現できるスケジューリングアルゴリズムを適用する必要がある．要件を満たさないアルゴリズムを使用する場合には，アプリケーション開発者が，検証によりデッドラインが満たされることを保証する必要がある．

時間保護のためのスケジューリングアルゴリズムを検討する際には，より多くの要件を満たせることが重要であるが，実システムに適用することを考えると，適用するための前提条件を出来る限り少なくすることも重要である．この観点から，本論文では，すべての要件を満たす時間保護アルゴリズム 1 と，要件 (1) と (2) のみを満たすが，前提条件を緩めた時間保護アルゴリズム 2 を提案する．

時間保護の要件を満たすスケジューリングアルゴリズムを実際のシステムで使用する最も容易な方法としては，既存のシステムで用いられている RTOS を拡張する形で実装する方法が挙げられる．これまで，時間保護の要件のうちいくつかを満たす RTOS が開発されている．既存の RTOS が有する時間保護機能と，本研究で提案する時間保護機能を違いを整理するため，次章から，保護機能をもつ組込みシステム向け RTOS について述べる．

2.3　保護機能をもつ組込みシステム向け RTOS

2.3.1　組込みシステムの特徴

多くの組込みシステムは，リアルタイム性を要求されるリアルタイムシステムであり，大規模な計算システムや汎用 PC などと比較すると，利用目的や利用形態が多岐に渡るという特徴がある．そのため，ハードウェアやソフトウェアの構成は，製品ごとに専用のものを開発することが多い．しかしながら，組込みシステムに対する要求や前提には共通する部分もある．ここでは，次の 4 つを挙げる．

1. 製品出荷後に，ハードウェアやソフトウェアの構成が変わらない
2. ソフトウェア全体が単一のオブジェクトにリンクされ，ROM に配置される
3. 要求仕様を満たす必要最低限のハードウェアが用いられる
4. システム全体に高い信頼性と安全性が要求される

1 と 2 は，最近の携帯電話のような高性能・高機能化が著しいシステムや，ネットワークに接続して外部と通信したり，後から新しい機能を追加するようなシステムでは成り立たなくなっている．しかしながら，ハードリアルタイム性を要求される制御系の組込みシステムにおいては，現在でも共通する特徴である．

3 については，コスト削減要求が高まっている製品には共通する要求である．特に，自動車内の ECU や家電製品など出荷数が多いシステムでは，要求仕様を満たす範囲で可能な限り，低コスト化が図られる．

4 は，近年の組込みシステムの高機能化，高性能化の影で大きな問題となっている品質，信頼性の問題である．特に，大規模化しているソフトウェアでは，品質や信頼性をどのように確保し，さらに向上させていくかが大きな課題となっている．安全性については，最近では，IEC（Internationa Electrotechnical Commission）[18] が，組込みシステムを含む電気・電子機器などのハードウェア及びソフトウェアに関して，機能安全に関する国際規格 IEC-61508 を発行しており，システムの安全性を確保するための取り組みが活発化している．

2.3.2　組込みシステム向け RTOS

リアルタイム性を要求される組込みシステムで，ソフトウェアが大規模化すると，複数のタスクを切替えたり，タスク間で同期や通信を行ないたいという要求

が出てくる．これらの機能は，タスクや割込みの処理内容とは独立に，共通化できる機能である．そこで，組込みシステムに OS を導入すると，容易にマルチタスクシステムを実現できることや，割込みの出入り口処理，タスク間同期，タスク通信などの機能を利用できることにより，アプリケーション部分の開発に注力できる利点がある．加えて，複数のアプリケーションで共通の OS を使用することで，アプリケーションの再利用性が高まり，ソフトウェア全体の開発効率が向上する．その一方で，OS を導入することの弊害もある．例えば，タスク切替処理や割込み処理におけるオーバヘッドの発生，メモリ消費量の増加などの欠点がある．そのため，必ずしも全てのリアルタイムシステムに，OS が必要であるとは限らない．例えば，自動車制御システムでは，現在でも OS を使用しない ECU が存在する．

組込みシステム向けの OS としては，リアルタイムシステム向けの RTOS を用いる場合と，マイクロソフト社の Windows，オープンソースの Linux に代表される汎用システム向け OS（汎用 OS と呼ぶ）を組込みシステム向けに拡張して用いる場合がある．どのような OS を採用するかは，ハードウェア構成やアプリケーションが要求する性能，時間制約から決定することになるが，RTOS と汎用 OS では異なる点が多い．例えば，一般的に，応答速度は RTOS の方が早く，カーネルサイズやメモリ消費量も圧倒的に小さい．しかしながら，扱えるデバイス数やカーネル以外のミドルウェア，ファイルシステム，開発ツールなどは汎用 OS の方が豊富である．

RTOS で最も重要な特徴は，カーネルの内部処理がリアルタイム性をもつことである．リアルタイムタスクには，リアルタイム性を満たすことが要求されるため，OS を用いて動作する場合には，OS 自身がリアルタイム性を持つことが必要である．ハードリアルタイムシステムにおいては，OS の内部処理時間の上限が定まることに加えて，短い応答時間を要求されることも多い．

2.3.3　RTOS に求められる保護機能

ハードリアルタイム性を要求される組込みシステムの分野では，ソフトウェアの品質を確保して信頼性を向上させるために，保護機能をもつ RTOS の研究，開発が進んでいる．組込みシステムにおける保護機能の目的として，次の 3 つが挙げられる．

1. 信頼性向上のための保護機能

システムの一部分に問題が発見された場合に，その影響でシステム全体が停止することを防ぐ．保護機能により，要求される信頼性の異なるタスクやアプリケーションを同一システムで実行することが可能となる．

2. デバッグ支援のための保護機能

   不正な，予期しないタスクの動作を検出することにより，システム上に問題があることを発見することができる．その結果，デバッグの効率が向上し，潜在的な問題を発見しやすくなる．

3. セキュリティ確保のための保護機能

   悪意を持ったプログラムを追加して実行した場合に，システム全体が破壊されることを防ぐ．

一般的に，RTOS に求められる保護機能には，メモリ保護のように，資源へのアクセスを空間的に保護する機能に加えて，資源へのアクセスを時間的に保護する時間保護機能がある．

メモリ保護は，メモリ領域をタスク，もしくはアプリケーションの単位で保護する機能である．組込みシステムのメモリ保護は，リアルタイム性の保証，低いオーバヘッドなど，汎用システム向けのメモリ保護とは異なる要件を満たす必要がある．多くのプロセッサに搭載される MMU（Memory Management Unit）を用いてメモリ保護を実現する場合，アドレス変換の過程において，TLB ミスが発生することでリアルタイム性が低下する問題がある．そこで，アドレス変換をせず，メモリ領域のアクセス制御を主眼に置いた MPU（Memory Protection Unit；メモリ保護機構）を用いるメモリ保護方式が提案されている [44]．MMU や MPU を利用してメモリ保護を実現するための RTOS 仕様としては，μITRON 仕様を拡張した μITRON/PX 仕様などが提案されている [39] や，AUTOSAR OS 仕様 [5] がある．メモリ保護を導入することで，単一のプロセッサ上で複数のアプリケーションを動作させる場合に，あるアプリケーションの不具合が別アプリケーションのメモリ領域を破壊することを防ぐことができる．

組込みシステムにおいて，時間的に多重アクセスされる資源の中で最も重要なのがプロセッサである．プロセッサの時間保護とは，プロセッサで実行される処理（タスク，割込み処理，アプリケーションなど）が使用するプロセッサ時間を保護する機能である．RTOS にプロセッサの時間保護機能があると，ある処理が想定し

た以上のプロセッサ時間を使ったために，他の処理がデッドラインを満たせなくなることを防ぐことができる．

時間保護の対象は，タスク，割込み処理，アプリケーションなど，さまざまなものがあるが，複数のアプリケーションを統合して動作する場合には，2.2.3 項で述べたように，アプリケーション単位の時間保護機能が有用である．アプリケーション単位の時間保護が実現できると，あるアプリケーションが想定した以上のプロセッサ時間を使ったために，他のアプリケーションのプロセッサ時間が少なくなり，デッドラインを満たせなくなることを防ぐことができる．特に，要件 (2) を満たす機能を実現できると，統合前に単独で実行した場合にデッドラインを満たせるアプリケーションは，複数のアプリケーションを統合して実行する場合でもデッドラインを満たして実行できることを保証できる．

2.4　既存 RTOS の時間保護機能

ここでは，これまでに提案されてきたハードリアルタイムシステム向けの組込み RTOS 仕様及び RTOS を紹介し，それらのもつ時間保護機能と，本論文で述べる時間保護機能の違いについて述べる．

2.4.1　ITRON 仕様

ITRON は，1984 年に開始された TRON プロジェクトの一環として検討，標準化されている組込み制御用 RTOS の仕様である．1987 年に ITRON1 仕様が公開されてから，8 ビットから 32 ビット MCU（Micro Control Unit）への適用，ソフトウェア移植性の向上など，さまざまな機能が拡張され，現在は μITRON4.0 仕様が公開されている [19]．国内では，ITRON 仕様に準拠した RTOS が数多く開発，製品化され，RTOS のデファクトスタンダードとなっている．2004 年に，日本システムハウス協会とトロン協会が行なったアンケート調査によると，製品に組み込んだ OS の API（Application Program Interface）の約 56%が，ITRON 仕様 API であるという結果が出ている．今後も，リソース制約の厳しい組込み機器や，ハードリアルタイムシステムで広く利用されていくものと考えられる．なお，TRON プロジェクトの活動は，実質的に T-Engine フォーラムに引き継がれている．

ITRON 仕様では，時間保護の要件を満たすための機能は提供されていないが，バグなどにより，予め想定した時間よりもタスクが長く実行されてしまうこと（オー

バランと呼ぶ）を防ぐための機能としてオーバランハンドラが提供されている．オーバランハンドラを使用することで，アプリケーションの中にオーバランしたタスクを検出し，他のタスクの実行に影響することを防ぐことができる．

2.4.2　OSEK OS 仕様

ドイツ，フランスを中心とした欧州の自動車業界の主要企業を中心に設立された標準化団体である OSEK/VDX（Offene Systeme und deren schnittstellen fur die Elektronik im Kraftfahrzeug/Vehicle Distributed eXecutive）[32] は，自動車制御システムの ECU に搭載される基本ソフトウェアの API を標準化することを目的に，RTOS の仕様である OSEK OS，ECU 内/ECU 間通信機能の仕様である OSEK COM，ネットワーク管理機能の仕様である OSEK NM の 3 つの仕様を公開している．OSEK/VDX の活動は，実質的に AUTOSAR（AUTomotive Open System ARchitecture）に引き継がれている．

OSEK OS 仕様は，利用環境に応じて必要な機能を取捨選択できるように工夫されている [33]．OSEK OS 仕様の特徴として，次のこと挙げられる．

- マルチタスクでリアルタイム性をもつ動作
- 複数のコンフォーマンスクラス（BCC1，BCC2，ECC1，ECC2)
- 2 つのタスクモデル（基本/拡張タスク）
- 3 つのスケジューリング機能（Non-preemptive/preemptive/Mix-preemptive）
- タスク間通信機能を OSEK COM として分離

タスクモデルは，running（実行状態），ready（実行可能状態），suspended（休止状態）の 3 状態をもつ基本タスクを採用する（BCC1 もしくは BCC2）か，基本タスクの 3 状態に waiting（待ち状態）を加えた 4 状態をもつ拡張タスクを採用する（ECC1 もしくは ECC2）かを選択できる．これとは直交した選択肢として，同一優先度のタスクは 1 つでタスクの多重起動を許さない（BCC1 もしくは ECC1）か，同一優先度タスクが複数存在してタスクの多重起動を許す（BCC2 もしくは ECC2）かを選択できる．さらに，OSEK COM 仕様の範囲を OS から分離することで，通信機能を使用しないシステムでは，メモリ消費量を削減できる．

OSEK OS 仕様の機能は，μITRON に比べると少ないが，自動車制御システムへの適用を念頭に仕様が作成されていること，欧州メーカが中心となって策定したことなどを理由に欧州で広く利用されている．2005 年には，自動車制御システム用 OS の国際標準 ISO17356 シリーズとして ISO で標準化された．現在は，欧州だけでなく，国内の自動車メーカも採用している．

OSEK/VDX の主要メンバにより構成された団体である HIS（Hersteller Initiative Software）[15] は，OSEK OS 仕様に対して，アプリケーションのメモリ保護機能と，タスクのデッドライン監視機能を追加する拡張仕様を公開している [16, 17]．デッドライン監視機能は，時間保護の要件を満たす機能ではないが，あるタスクがオーバランした結果デッドラインミスが発生した場合にそれを検出する機能である．オーバランしたタスクの影響で正常に動作しているタスクがデッドラインをミスする場合もあるため，デッドラインをミスしたタスクを特定できたとしても，その根本的な原因をもつタスク（すなわち，オーバランしたタスク）を特定することはできない．

2.4.3　AUTOSAR OS 仕様

世界の自動車メーカ，サプライヤ，ツールベンダ，ソフトウェア開発企業により構成されている標準化団体 AUTOSAR（AUTomotive Open System ARchitecture）[4] は，RTOS だけでなく，自動車制御システム向けソフトウェアアーキテクチャの標準仕様を公開している．

AUTOSAR の公開する仕様に含まれる AUTOSAR OS 仕様では，備えるべき保護機能を，段階的に 4 つのスケーラビリティクラス（SC）として規定している [5]．まず，SC1 として，OSEK OS 仕様をベースに細かい改良を加えた仕様を規定している．SC1 に対して，タイミング保護機能を加えた SC2，アプリケーション単位のメモリ保護機能を加えた SC3 を規定しており，さらに，SC2 と SC3 の機能を合わせもつ SC4 が規定されている．2009 年に公開されたバージョン 4.0 では，マルチコアプロセッサ向けの RTOS 仕様も追加された．

AUTOSAR OS のタイミング保護機能は，ISR2 と呼ばれる割込み処理ルーチンとタスクに対して，実行時間，起動間隔，割込み禁止時間，リソース確保時間を保護する機能である．実行中に，これらの時間が，予め設定した上限時間を越えると，保護違反が発生したと判断して，ProtectionHook と呼ばれる異常対応処理が呼び出される．ProtectionHook の仕様では，異常への対応処理として，保護違

反を発生した処理（ISR2 もしくはタスク）を強制終了させることや，その処理が含まれるアプリケーションを単独で停止，再起動すること，さらに，システム全体を停止，再起動する機能が規定されている．

AUTOSAR OS では，アプリケーションの概念や，アプリケーション単位のメモリ保護機能を規定するなどアプリケーション統合を想定した仕様になっている．しかしながら，先に述べたように，タイミング保護は処理ごとのプロセッサ時間の保護に留まっており，本研究の目的であるアプリケーション単位の保護機能ではない．そのため，既存のシステムで単独で動作しているアプリケーションを統合する場合には，各アプリケーションのタスクの優先度設計を見直す必要がある．加えて，統合後に動作を再検証をする必要があるために，統合検証のコストが大きくなることが問題となる．本研究で検討するアプリケーション単位の保護機能と，AUTOSAR OS のタイミング保護機能は，保護の単位が異なるため，これらを組み合わせることで，より強固な保護機能を実現することも可能である．

2.4.4　ARINC 653

米国 Aeronautical Radios（ARINC）社の APplication/EXecutive（APEX）ワーキンググループは，航空機制御システムの IMA 向け基盤ソフトウェアの標準規格である ARINC 653 を規定している [3]．ARINC 653 では，IMA で複数のアプリケーションを動作させる状況においても高い安全性，信頼性を実現することを目的に，パーティションとよばれる単位でメモリとプロセッサ時間を保護するための API を規定している．パーティション間の保護機能はパーティショニングと呼ばれ，近年では，自動車制御システムを対象とした安全規格 ISO-26262 にも盛り込まれるなど，航空機以外の分野においても注目が高まっている．

ARINC 653 の提供する時間保護機能では，まずシステムの構築段階で，パーティションごとに実行周期と実行時間を決める．次に，パーティションの実行順序と実行開始時間，実行時間を定めたスケジュールテーブルをシステム全体で 1 つ生成する．システム動作時は，そのスケジュールテーブルにしたがって，パーティションを繰り返し実行することで，プロセッサ時間を保護する．この機能は，本論文で定めた時間保護の 3 つの要件のうち，要件 (1) のアプリケーションのプロセッサ利用率の保護を実現することができるが，残りの 2 つの要件は満たしていない．すなわち，2.2.2 項で述べたように，統合後にタスクがデッドラインをミスしてしまう可能性がある．したがって，アプリケーション開発者が，統合後にタスクがデッ

ドラインを満たすよう設計，検証を行う責任をもつ．それに対して，本論文では，時間保護の要件 (2) を満たすスケジューリングアルゴリズムを提案しており，統合後にタスクがデッドラインを満たすことをスケジューラが保証する点において大きく異なる．なお，ARINC 653 では，さらにパーティション内のタスク（ARINC 653 ではプロセスと呼ぶ）のデッドラインを監視する機能も規定している．

ARINC 653 に準拠した RTOS としては，米国 Wind River 社（2009 年に，米国 Intel 社により買収）の VxWorks を拡張した VxWorks 653，米国 Green Hills 社の INTEGRITY を拡張した INTEGRITY-178B などがある．これら以外にも，複数の仮想マシンを管理するための小規模なソフトウェアである仮想マシンモニタ（ハイパーバイザーとも呼ばれる）が，仮想マシン間の保護（すなわち，パーティショニング）を提供することで ARINC 653 に準拠する製品も登場している．具体的には，Wind River 社の Wind River Hypervisor，独 SYSGO 社の PikeOS などがある．ハードウェアを仮想化する技術を用いて時間保護を実現する場合には，OS 間の切り替えオーバヘッドが大きいため，高性能なプロセッサを採用する必要がある．

2.5　関連研究

時間保護によるアプリケーション統合に関連する従来研究について述べる．最初に，アプリケーション統合を目的として，時間保護を実現するスケジューリングアルゴリズムが満たすべき要件に関する関連研究について述べる．

複数の個別プロセッサで動作するアプリケーションを統合プロセッサに容易に統合することを目的とした過去の研究では，統合前の個別プロセッサでデッドラインを満たすタスクが，統合後にデッドラインを満たせなくなる要因を整理して，スケジューリングアルゴリズムが満たすべき要件を定義しているものは少ない．特に，2.2.2 項で指摘した QoS 制御されたタスクの存在を考慮して要件を定義している研究はない．本論文では，2.2.3 項に列挙したように，リアルタイムアプリケーションを統合することを目的として，時間保護を実現するスケジューリングアルゴリズムが満たすべき要件を 3 つに整理した．

次に，時間保護を実現するスケジューリングアルゴリズムに関する関連研究について述べる．

アプリケーションの QoS 制御を主な目的とした CPU リソース管理手法が数多く研究されている．例えば，Oikawa らは，CPU リソース管理機能をもつ小型のカーネル Resource Kernel を Linux に実装し，汎用 PC では無視できる程度のオー

バヘッドで QoS 制御を実現した [31]．この手法では，アプリケーション毎のプロセッサ利用率を保護できることから，時間保護の要件 (1) を満たすことができる．しかし，要件 (2) を満たすことを保証していない．さらに，QoS 制御されたタスクを考慮していないため，要件 (3) を満たすことを保証していない．

アプリケーション統合に適用できるアルゴリズムとして，バジェットをタスク単位で管理するサーバアルゴリズムが提案されている [38, 37, 1, 2, 6]．これらのアルゴリズムは，もともと周期タスクのリアルタイム性保証と，非周期タスクの応答性の改善を目的としており，これらの手法をアプリケーション統合に適用して各タスクのリアルタイム性を保証するためには，すべてのタスクの動作を考慮してスケジュール可能性を検証する必要がある．したがって，統合するアプリケーションの組み合わせが変化する度に，スケジュール可能性を解析し直さなければならない．

サーバアルゴリズムにおいて，1 つのサーバに複数のタスクを割り当てることにより，要件 (1) を満たすことが可能である．さらに，割り当てられたタスクの最大公約数の周期でサーバにプロセッサ時間を割当てることで，サーバ上のすべてのタスクがデッドラインを満たせるようスケジュールできる（すなわち，要件 (2) を満たせる）．しかし，すべてのタスクの周期の最大公約数が非常に小さな値になる場合には，タスクの切替えが頻発する問題がある．加えて，QoS 制御されたタスクによるバジェットの先使いを考慮していないため，要件 (3) を満たすことは保証されない．サーバに割り当てられたバジェットが余った場合に，それを他のサーバが再利用することで，統合プロセッサの利用率を向上させるアルゴリズムも提案されている [24, 8, 29, 7]．これらの手法では，余りバジェットをどのタスクが再利用するかを実行時に決定するため，同一のアプリケーションに所属するタスクが利用できるとは限らない．タスク数が多い場合は，各タスクのバジェットや余りバジェットを管理するための計算が多くなる問題がある．

Deng らは，すべてのタスクの起動時刻と，実行時間の上限値が既知であることを前提として，時間保護の要件をすべて満たす階層型スケジューリングを提案している [10, 11]．しかし，2.1.4 項で述べたように，現実のシステムでは，タスクの実行時間の上限値を正確に見積もることは容易ではない．

本論文では，第 3 章において，Deng らのアルゴリズムをベースに，タスクの実行時間の上限値に代えて，相対デッドラインを用いるように修正した時間保護アルゴリズム 1 を提案する [46]．

Lipari らは，タスクの起動時刻が既知であることを前提とせず，デッドラインのみで，統合後の統合プロセッサ上のすべてのタスクがデッドラインを満たすこと

を保証する BSS（Bandwidth Sharing Server）アルゴリズム [23] と，それを改良した PShED（Processor Sharing with Earliest Deadline）アルゴリズム [9] を提案している．これらのアルゴリズムは，タスクの相対デッドラインが分かれば適用できるため，多くのアプリケーションを対象にできる．BSS アルゴリズムと PShED アルゴリズムは，階層型スケジューリングを採用しており，ローカルスケジューリングとグローバルスケジューリングの両方に EDF スケジューリングを用いる場合に，時間保護の要件 (1) と (2) を満たすことを証明している．しかし，ローカルスケジューリングに固定優先度スケジューリングを用いる場合には，要件 (2) を満たすことができないことが分かっている [22]．さらに，要件 (3) は保証されない．

本論文では，第 4 章において，BSS アルゴリズムと PShED アルゴリズムをベースに，ローカルスケジューリングに固定優先度スケジューリング（厳密には固定優先度スケジューリングと同一ではないが，高い優先度を持つタスクが，その実行中に低い優先度をもつタスクに邪魔されることはないという性質をもつアルゴリズム）を採用する場合でも，要件 (1) と (2) を満たせる時間保護アルゴリズム 2 を提案する [45].

Shin や Lipari らは，タスクの起動周期と最悪実行時間の情報を利用して，アプリケーション内のすべてのタスクがデッドラインを満たす (P,Q) の組を求める手法を提案している [36, 25]．P はアプリケーションの実行周期，Q は周期毎に必要なバジェットである．実行時にはすべてのアプリケーションを周期実行するため比較的容易に実装できるが，起動周期を設計する段階では，各タスクの起動周期（もしくは最小到着間隔），最悪実行時間，相対デッドラインの情報が必要である．これらの手法は，時間保護のすべての要件を満たすことができると思われる．しかし，アプリケーション内に周期の長いタスクと短いタスクが混在すると，アプリケーションの実行周期を周期の短いタスクに合わせて設定する必要があり，PShED アルゴリズムや時間保護アルゴリズム 1 と 2 のように，タスクの相対デッドラインを用いる手法に比べて，デッドラインに余裕があるタスクの切替え回数が増加する可能性がある．さらに，アプリケーションの構成によっては，アプリケーションに設定するシェアを，各タスクが必要とするプロセッサ利用率を合計した正味のプロセッサ利用率に比べて，30%程度高く設定する必要がある [36]．よって，プロセッサ性能に余裕のない制御システムに適用する場合には，本論文で提案する手法の方が適する．

多様なサーバアルゴリズムを実装できるリアルタイムカーネルとしては，S.Ha.R.K（Soft and Hard Real-time Kernel）[12, 26, 13] がある．S.Ha.R.K は，カーネルの

スケジューリング部分を，外部から定義するための Generic Kernel と呼ばれるモジュールを持っており，スケジューリングアルゴリズム開発者が独自のアルゴリズムを容易に実装できる．タスク単位の保護を基本としているが，ローカルスケジューラで，サーバを 2 レベル以上に階層化することで，アプリケーション単位の保護を実現できると思われる．しかしながら，アルゴリズム開発者により実装されたローカルスケジューラでアプリケーションのデータ構造を管理する必要があるため，アプリケーションもしくはローカルスケジューラに不具合が発生すると，別のローカルスケジューラやアプリケーションにも影響を及ぼしてしまう．その結果，アプリケーショの間の保護が崩れてしまう可能性がある．

Oikawa らは，マイクロカーネル上で複数の RTOS を動作させ，OS レベルでリアルタイムアプリケーションを統合する手法を提案している [30]．OS の多重化は，統合するアプリケーションの構造を変更することなく適用できる利点がある．しかし，本論文で提案するような，ローカルスケジューラを多重化する手法に比べると，メモリ消費が多く，OS 切替えによる処理オーバヘッドも大きい．

本論文では，第 5 章において，これまで提案された階層型スケジューリングアルゴリズムに共通する部分と異なる部分を整理し，異なる部分については，アプリケーション毎に選択できるよう柔軟性を高めたスケジューリングフレームワークを提案する [47]．具体的には，アプリケーションごとに，アプリケーションに要求される時間要件や，統合段階で既知であるパラメータに応じて，ローカルスケジューラのアルゴリズムを選択できるようにし，グローバルスケジューラは共通部分として統一する．ローカルスケジューラとグローバルスケジューラ間の共通インターフェースを定義し，ローカルスケジューラは，このインタフェースを介してグローバルスケジューラと情報をやりとりする．これにより，特定のアプリケーションやローカルスケジューラの時間的な動作に不具合が，別のアプリケーションの動作に影響することを防ぐことができる．既存の RTOS をベースに提案フレームワークを実装し，実用上，許容できるオーバヘッドで実現できることを明らかにする．

Regehr らは，階層型スケジューリングの汎用的フレームワーク HLS を提案している [35, 34]．このフレームワークでは，割込み処理を含むアプリケーションを動作させる場合，割込み処理を，システムに存在するすべてのタスクに優先して実行することが前提とされているために，アプリケーション統合に際してアプリケーションの構成を変更する必要があるという問題がある．

RTOS や汎用 OS を対象とした割込み処理に関する研究としては，これまでいく

つかの割込み処理モデルが提案されている [21, 43]．しかしながら，これらの研究は階層型スケジューリングを対象としたものではない．割込み処理のスケジュール可能性解析手法としては，[20] が提案されている．この手法では，割込み処理とタスクが周期的に発生することを前提としており，そのままでは，階層型スケジューラでの解析に利用することはできない．

本論文では，第 6 章において，タスクと割込み処理で構成されるリアルタイムアプリケーションも統合できるようにするために，第 5 章で述べた階層型スケジューリングフレームワークに対して容易に追加できる，割込み処理のスケジューリングアルゴリズムを提案する [28]．さらに，割込み処理の最大応答時間を解析する．

2.6　本研究の有用性

最後に，本研究で提案するスケジューリングアルゴリズムとそれを実装した RTOS の有用性を整理する．

- リアルタイムアプリケーション統合の促進

  時間保護の要件を満たすアルゴリズムを開発することにより，個別プロセッサでデッドラインを満たすタスクが，統合プロセッサへの統合後にもデッドラインを満たすことを保証できると，個別プロセッサにおけるアプリケーションの動作検証の結果は，統合後の統合プロセッサ上でも有効となる．その結果，統合に際してアプリケーションの再検証の負担は大幅に軽減でき，統合が容易になったということができる．特に，本研究の適用対象としている自動車制御システムにおいては，時間保護機能により ECU の統合を促進できる．その結果，自動車に搭載される ECU 数だけでなく，ECU 間通信のネットワークケーブル（ハーネス）の数も削減できる．

- 統合検証における問題の切り分け

  複数のアプリケーションが動作するシステムにおいて，あるタスクが，デッドラインを満たして実行することができず，デッドラインをミスしてしまった場合，時間保護機能がない環境では，そのタスクに問題があるのか，別のタスクの実行時間が延長した影響を受けたことが原因で，時間制約を満たせなかったのかを特定することが困難である．しがたって，どのアプリケーションに不具合が存在するのかを特定することも難しい．本論文で提案するアル

ゴリズムを適用することで，個別プロセッサ上での動作が統合プロセッサ上でも保証されるため，統合プロセッサ上で発生したデッドラインミスの要因は，そのデッドラインをミスしたタスクが属するアプリケーションにあると特定できる．

- 異なる信頼性，安全性をもつアプリケーションの統合

  組込みシステムに搭載されるアプリケーションは，動作環境に応じて適切な信頼性，安全性を確保する必要がある．IEC61508 や ISO26262 などの国際標準安全規格では，いくつかのレベル分けされた検証基準から，適切な検証基準を選択して，動作検証を行なうことが規定されている．検証レベルの異なるアプリケーションを統合する場合，従来の RTOS では，最も高い検証レベルを必要とするアプリケーションに合わせてシステム全体を検証する必要があった．それに対して，アプリケーション単位の時間保護を導入すると，アプリケーションの動作が互いに影響を及ぼさないことが保証されるので，異なる検証レベルのアプリケーションを比較的容易に統合することができる．さらに，既存の個別プロセッサで動作を確認済みであるアプリケーションは，統合プロセッサでもデッドラインを満たして動作するため，統合前の検証レベルを維持して統合することを容易になる．

2.7　まとめ

本章では，自動車制御システムに代表される分散リアルタイムシステムにおいて，プロセッサ数が増加している問題を整理し，その問題を解決するために，高性能な統合プロセッサに，複数のアプリケーションを統合して動作させるアプリケーション統合が必要であることを述べた．さらに，アプリケーション統合を容易に実現するためには，アプリケーションごとのプロセッサ時間を保護する時間保護が有効であることを述べ，スケジューリングアルゴリズムが満たすべき要件を整理した．時間保護の要件の観点から，本研究で提案するアルゴリズムの時間保護機能と，既存の RTOS 仕様で規定されている時間保護機能の違いを述べた．関連研究としては，これまで提案されたスケジューリングアルゴリズムと本論文で述べるスケジューリングアルゴリズムとの違いを述べた．最後に，本研究の有用性を整理した．

第3章　時間保護のための階層型スケジューリングアルゴリズム

3.1　概要

本章では，Deng らのアルゴリズムをベースに，見積もりにくい実行時間の上限値に代えて，デッドラインを用いるように修正した時間保護アルゴリズム 1 を提案する．時間保護アルゴリズム 1 は，統合前に固定優先度スケジューリングによりスケジュール可能であったアプリケーションを単一のプロセッサに統合することを想定しており，ローカルスケジューラは固定優先度スケジューリングを，グローバルスケジューラは EDF スケジューリングを採用する．時間保護アルゴリズム 1 を適用するための前提条件は，タスクの正確な実行時間の情報を必要とせず，タスクの起動時刻と相対デッドラインの 2 つの情報が得られることである．アルゴリズムの詳細な動作と具体的な動作例を示し，時間保護アルゴリズム 1 が時間保護の 3 つの要件をすべて満たすことを証明する．

本章の構成を述べる．3.2 節では，提案する時間保護アルゴリズム 1 について詳細に述べる．3.3 節では，提案アルゴリズムが時間保護機能の要件を満たすことを示す．最後に，3.4 節で結論を述べる．

3.2　時間保護アルゴリズム 1

3.2.1　前提と用語定義

提案アルゴリズムでは，タスクの起動時刻とデッドラインの 2 つの情報が既知であることを前提とする．タスクの起動時刻とデッドラインをタスクイベントと呼ぶ．アプリケーションに含まれるタスクのタスクイベントを，アプリケーションイベントと呼ぶ．さらに，ある時刻において最も早く発生するタスクイベントをそのタスクの次タスクイベントと呼び，最も早く発生するアプリケーションイベントをそのアプリケーションの次アプリケーションイベントと呼ぶ．

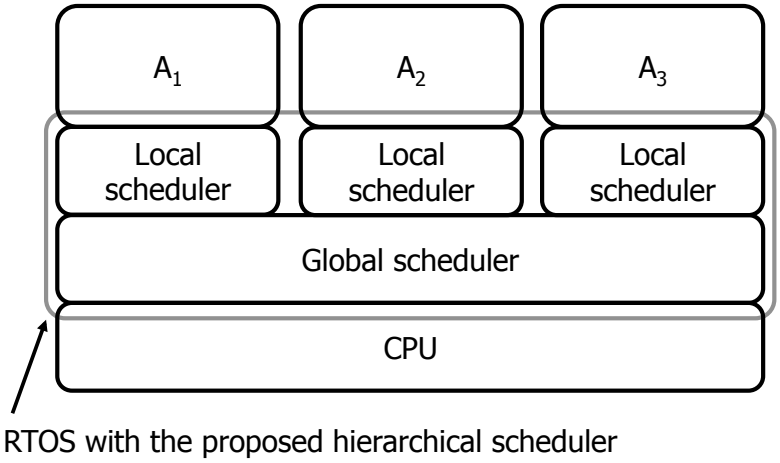

図 3.1: 階層型スケジューラの構成

3.2.2 スケジューラの構成

提案アルゴリズムは，図3.1 に示すように，アプリケーションをスケジューリングするグローバルスケジューラと，アプリケーション内のタスクをスケジューリングするローカルスケジューラを階層的に配置した階層型スケジューラで実現する．

ローカルスケジューラは，アプリケーションと 1 対 1 に対応し，アプリケーションに所属するタスクを，統合前に個別プロセッサで動作していたときのアルゴリズムと同じアルゴリズムでスケジューリングする．本章では固定優先度スケジューリングを前提とする．グローバルスケジューラは，アプリケーションの実行順序を決定するスケジューリングと，アプリケーションに対するバジェット割当ての 2 つの機能をもつ．

3.2.3 スケジューリングアルゴリズム

グローバルスケジューラは，バジェットが 0 でなく，次アプリケーションイベントが最も早いアプリケーションを最高優先順位アプリケーションとして選択する．アプリケーションイベントでは，現在時刻から，そのアプリケーションの次アプリケーションイベントまでの時間と，シェアの積をバジェットとして割り当てる．グローバルスケジューラは，実行中のアプリケーションのバジェットが 0 になると，次に優先順位の高いアプリケーションに実行を切り替える．

提案アルゴリズムでは，ハードウェアや RTOS に対する特別な前提機能はない．

すなわち，アプリケーションに対するバジェットの割当てや，バジェットが 0 になったときにアプリケーションを切り替える機能などは，多くのマイコンが備えるハードウェアタイマや割込みの機能を利用することで容易に実現できる．本研究の主な適用対象である自動車制御システムの ECU（主に 16 ビットから 32 ビットのマイコンが搭載されるものが多い）に適用する場合には，RTOS のスケジューラを修正することで，提案アルゴリズムを実装することができる．

具体的なスケジューリングアルゴリズムを示す．ここでは，アプリケーション $A_i(i = 1, 2, \ldots, N)$ のタスク $T_{ij}(j = 1, 2, \ldots, M)$ の k 回目のジョブを $J_{ijk}(k = 1, 2, \ldots)$ と表記する．

- 初期化
  1. すべてのアプリケーションのバジェットを 0 にする．
- タスク T_{ij} の k 回目のジョブ J_{ijk} が時刻 r_{ijk} で起動されたときの動作
  1. ローカルスケジューラはアプリケーション A_i 内のタスクをスケジューリングする．
  2. タスク T_{ij} の次タスクイベント NTE_{ij} を起動時刻 r_{ijk} から絶対デッドライン d_{ijk} に更新する．
  3. アプリケーション A_i の次アプリケーションイベント NAE_i を更新する．
  4. アプリケーション A_i のシェア U_i を用いて，アプリケーション A_i のバジェット B_i を次の式を用いて更新する．

$$B_i = (NAE_i - r_{ijk}) \times U_i$$

  5. グローバルスケジューラはアプリケーションをスケジューリングする．アプリケーション A_i が最高優先順位アプリケーションになった場合は，実行中アプリケーションのバジェットを保存する．
  6. 最高優先順位アプリケーションの最高優先順位タスクを実行する．
- ジョブ J_{ijk} の実行が完了したときの動作
  1. 次タスクイベント NTE_{ij} をジョブ J_{ijk} のデッドライン d_{ijk} から，次のジョブの起動時刻 $r_{ij(k+1)}$ に更新する．
  2. 次アプリケーションイベント NAE_i を更新する．

表 3.1: アプリケーション B のタスクセット

	起動周期	相対デッドライン	最悪実行時間	プロセッサ利用率 (%)	最悪応答時間
中優先度タスク	5	5	3	60.0	3

3. アプリケーション A_i に実行可能タスクがある場合は，次のように動作する．
   (a) ローカルスケジューラがタスクをスケジューリングする．
4. アプリケーション A_i に実行可能タスクがなければ，次のように動作する
   (a) バジェット B_i を 0 にする．
5. グローバルスケジューラはアプリケーションをスケジューリングする．最高優先順位アプリケーションがアプリケーション A_i 以外のアプリケーションになった場合は，バジェット B_i を保存する．
6. 最高優先順位アプリケーションの最高優先順位タスクを実行する．

- ジョブ J_{ijk} の実行中にバジェットが 0 になったときの動作
  1. ジョブ J_{ijk} の実行を中断する．
  2. アプリケーション A_i のバジェット B_i を保存する．
  3. グローバルスケジューラはアプリケーションをスケジューリングする．
  4. 最高優先順位アプリケーションの最高優先順位タスクを実行する．

3.2.4 動作例

提案アルゴリズムの動作を，表 2.2 のアプリケーション A’ と，表 3.1 のアプリケーション B を統合プロセッサ上に統合する例を用いて説明する．まず，図 3.2 に示すように，アプリケーション A’ とアプリケーション B は，性能 1 の個別プロセッサでそれぞれデッドラインを満たす．図には，アプリケーション毎の次アプリケーションイベントも示している．次に，性能 2 の統合プロセッサ上で，2 つのアプリケーションにそれぞれシェア 0.5 を設定し提案アルゴリズムを適用すると，図 3.3 に示すように，両アプリケーションともにデッドラインを満たす．

時刻 0 では，アプリケーション A’ の次アプリケーションイベントは時刻 3 なので，アプリケーション A’ に対してバジェットが 1.5 単位時間割当てられる．同様に，アプリケーション B の次アプリケーションイベントは時刻 5 なので，アプリ

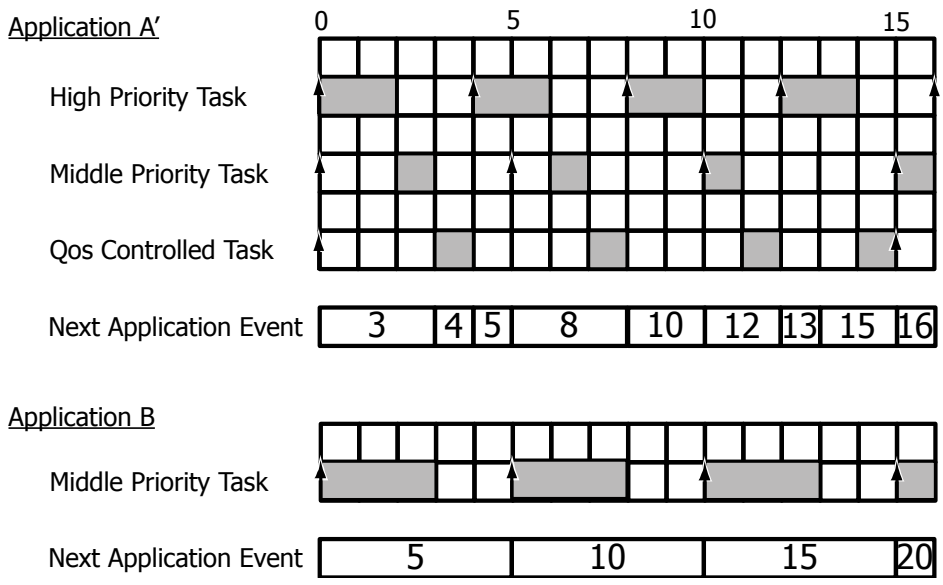

図 3.2: 個別プロセッサでのアプリケーションの動作

ケーション B に対してバジェットが 2.5 単位時間割当てられる．このとき，どちらのアプリケーションにも実行可能タスクが存在するが，次アプリケーションイベントはアプリケーション A' の方が早いので，アプリケーション A' のタスクから実行される．アプリケーション A' は時刻 1.5 でバジェットを使い切るので，アプリケーション B のタスクに実行が切り替わる．

時刻 12 では，アプリケーション B のタスク実行中に，アプリケーション A' の高優先度タスクが起動し，アプリケーション A' の次アプリケーションイベント（時刻 13）がアプリケーション B の次アプリケーションイベント（時刻 15）より早くなる．この場合には，アプリケーション B の実行を中断してアプリケーション B の残りバジェットを保存し，アプリケーション A' の高優先度タスクに実行が切り替わる．

アプリケーション A' がバジェットを使い切ると，先に保存した残りバジェットを使ってアプリケーション B のタスクの実行を再開する．このようにして，各アプリケーションがアプリケーションイベント間で利用可能なバジェットが制限される．これにより，各アプリケーションのプロセッサ利用率は，シェアを越えることはなく，アプリケーション間でプロセッサ時間を保護できる．

提案アルゴリズムでは，タスクのデッドラインだけではなく，起動時刻もアプリケーションイベントとして管理する．そのため，アプリケーション内に QoS 制御

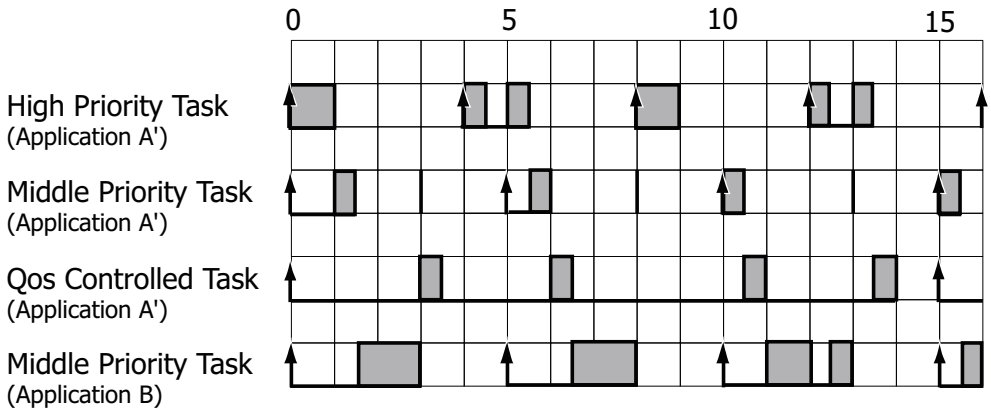

図 3.3: 統合プロセッサでのアプリケーション A’ の動作

されたタスクが存在する場合でも，QoS 制御されたタスクが，同じアプリケーション内のより高い優先度をもつタスクが起動される前に，バジェットを先使いしてしまうことを防ぐことができる．その結果，PShED アルゴリズムを適用した場合には，時刻 16 でデッドラインをミスしたアプリケーション A’ の高優先度タスクは，提案アルゴリズムを適用するとデッドラインを満たせる．

図 3.2 と図 3.3 を比較すると，低性能プロセッサと統合プロセッサでタスク実行順序が一致していることがわかる．例えば，時刻 0 から時刻 5 における，アプリケーション A’ のタスクの実行順序は，どちらも高優先度タスク，中優先度タスク，低優先度タスク，高優先度タスクの順序で実行されている．このように，提案アルゴリズムでは，個別プロセッサと統合プロセッサにおけるタスクの実行開始時刻は異なるが，実行順序は保存されるという性質がある．

アプリケーション毎のバジェットと累積実行時間の推移を図 3.4 に示す．統合前と統合後で，各アプリケーションが得られるアプリケーションイベント間の処理量は一致していることがわかる．累積実行時間は，アプリケーションイベントの時点で必ず一致している．例えば，個別プロセッサ上でのアプリケーション A’ は，時刻 0 から時刻 10 の間で 10 単位時間実行されているが，統合プロセッサでは，5 単位時間実行されており，プロセッサの性能差を考慮した処理量は一致している．

3.3 証明

ここでは，時間保護アルゴリズム 1 がもつ，タスクの実行順序が一致する性質と，アプリケーションイベント間で処理量が一致する性質をそれぞれ証明し，そ

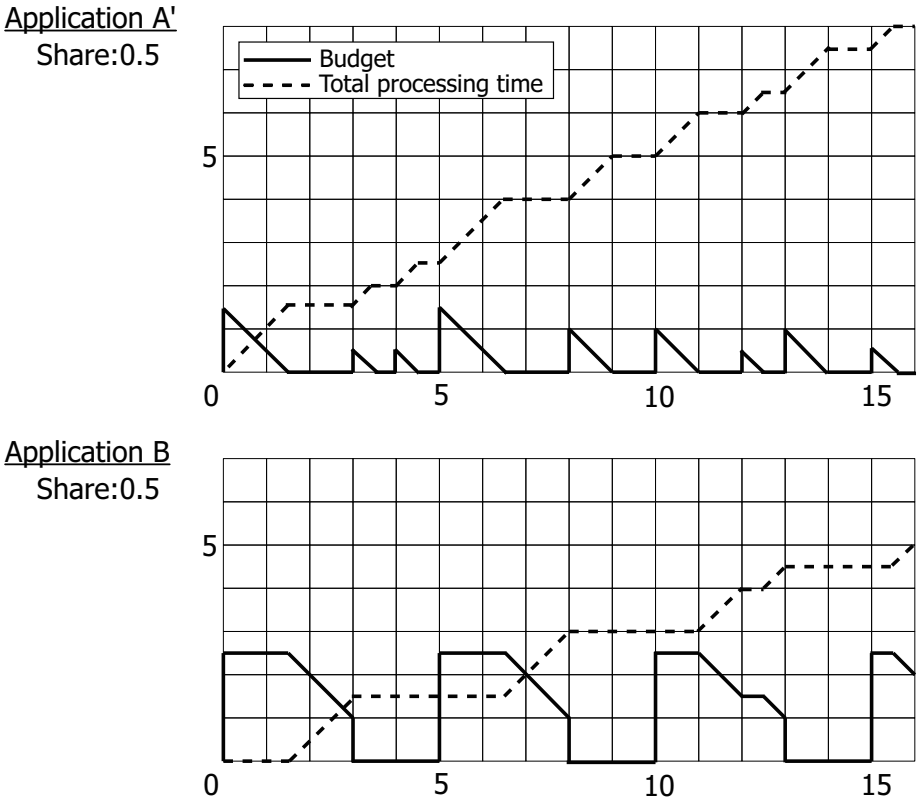

図 3.4: 2 つのアプリケーションのバジェットの推移

れを使って，時間保護の 3 つの要件を満たすことを示す．まず，提案アルゴリズムによりスケジュールされるアプリケーションについて，次の補題 1 が成り立つ．

補題 1 統合プロセッサで動作するアプリケーションは，アプリケーションイベントで割当てられたバジェットを次アプリケーションイベントまでに使用できる．

(証明) まず，アプリケーションイベントをそのアプリケーションの起動時刻，次アプリケーションイベントをそのアプリケーションのデッドラインと，それぞれ置き換えると，アプリケーションの動作をタスクの動作と同様に扱うことができる．さらに，グローバルスケジューラのスケジューリングアルゴリズムは，次アプリケーションイベントをデッドラインとする EDF スケジューリングであると考えることができる．

このことを踏まえて，アプリケーションのスケジュール可能性は，次のように示すことができる．アプリケーション $A_i(i = 1, 2, \ldots, N)$ のシェアは，高性能プロセッサの性能に対する A_i が動作する個別プロセッサの性能 U_i の割合と定義している．統合プロセッサにおける A_i の最大プロセッサ利用率 U_i^f は，$U_i / \sum_{j=1}^{N} U_j$ とな

る．したがって，統合プロセッサで動作するアプリケーションの最大プロセッサ利用率の合計は，次のようになる．

$$\sum_{i=i}^{N} U_i^f = \sum_{i=1}^{N}(U_i / \sum_{j=1}^{N} U_j) = 1 \tag{3.1}$$

プロセッサ利用率の合計が 1 以下となるタスクセットは，EDF スケジューリングによりスケジューリング可能であることが知られており [6]，全てのアプリケーションは，提案アルゴリズムによりスケジュール可能である．よって，統合プロセッサで動作するアプリケーションは，割当てられたバジェットを次アプリケーションイベントまでに使用できる．

さらに，次の補題 2 と補題 3 が成り立つ．

補題 2 すべてのアプリケーションについて，アプリケーションイベント間の処理量は個別プロセッサ上と統合プロセッサ上で一致する．

(証明) 補題 1 より，バジェットは，統合プロセッサにおけるアプリケーションイベント間での処理量の上限値であると考えることができる．2 つのアプリケーションイベント t_1 から t_2 における，プロセッサの性能差を考慮した処理量を $P(t_1, t_2)$ と表記すると，任意のアプリケーションイベント t において，個別プロセッサの処理量 P^l と統合プロセッサの処理量 P^h について，次の式が成り立つ．

$$P^l(0, t) \leq P^h(0, t) \tag{3.2}$$

$P^l < P^h$ となる状況とは，個別プロセッサ上に実行可能なタスクが存在せず，統合プロセッサ上に実行可能なタスクが存在し，かつ，そのタスクが実行される場合である．そこで，時刻 t' に，統合プロセッサ上にこのような実行可能タスク T が存在すると仮定する．個別プロセッサ上と統合プロセッサ上では，タスクの起動時刻が一致するので，時刻 t' までに起動されるタスクも一致する．時刻 t' までに実行可能となるタスクが一致し，タスク T が統合プロセッサ上で実行可能であるということは，時刻 0 から時刻 t' における個別プロセッサの処理量が統合プロセッサの処理量よりも多いことになる．すなわち，次の式が成り立つ．

$$P^l(0, t') > P^h(0, t') \tag{3.3}$$

これは，明らかに式 (3.2) に矛盾するため，$P^l < P^h$ は成り立たない．よって，時刻 0 から任意のアプリケーションイベント t の間での処理量について，次の式が成り立つ．

$$P^l(0,t) = P^h(0,t) \tag{3.4}$$

この関係は，任意の 2 つのアプリケーションイベント t_1 から t_2 についてもいえる．そのため，次の式が成り立つ．

$$P^l(t_1,t_2) = P^h(t_1,t_2) \tag{3.5}$$

以上より，すべてのアプリケーションについて，任意のアプリケーションイベント間の処理量は，個別プロセッサ上と統合プロセッサ上で一致する．

補題 3 タスクの実行順序は個別プロセッサ上と統合プロセッサ上で一致する．

(証明) 補題 3 を示す方針として，両プロセッサ上でタスク切換えが発生する状況において，最高優先順位のタスクが必ず一致することを示す．提案アルゴリズムのローカルスケジューラは，プリエンプティブなスケジューリングアルゴリズムを前提としているので，アプリケーション内のタスク切換えは，次のいずれかの状況で発生する．

1. 実行可能なタスクが存在しないときにタスクが起動する
2. 実行中のタスクよりも高優先度のタスクが起動する
3. 実行中のタスクの実行が終了する

個別プロセッサ上と統合プロセッサ上では，タスクの起動時刻が一致するため，状況 1 と状況 2 によるタスク切り替えは同じ時刻で発生する．補題 2 より，アプリケーションイベントまでに実行可能になったタスクと，それらのタスクに対する処理量は両プロセッサで一致する．ローカルスケジューラのスケジューリングアルゴリズムが統合の前と後で同じであるという前提より，実行したタスクとその順序も一致する．よって，アプリケーションイベントでは，実行可能なタスク（実行可能になった時刻も）とその優先順位は一致する．よって，個別プロセッサ上で起動されて最高優先順位となるタスクは，統合プロセッサ上でも，アプリケーション内の最高優先順位タスクとなる．

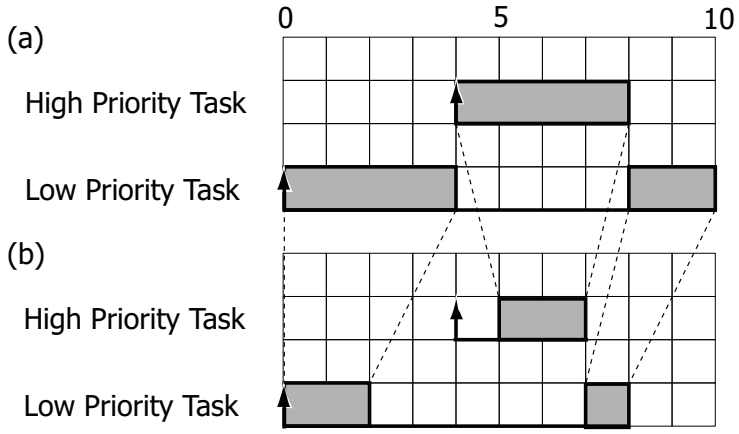

図 3.5: 同一アプリケーション内でのタスク切換え

状況 2 によりタスクがプリエンプトされる状況について具体的な例を用いて説明する．図 3.5 は，性能 1 の個別プロセッサ上と性能 2 の統合プロセッサ上において，時刻 4 で同時にタスクがプリエンプトされる例である．高性能プロセッサの場合には，複数のアプリケーションが存在するため，アプリケーション内の最高優先順位タスクが切り替わっても即座に実行されるとは限らない．図の例では，高優先度タスクと低優先度タスクのデッドラインは，どちらも時刻 10 とする．さらに，統合プロセッサ上では，時刻 4 から時刻 5 の間に別のアプリケーションが実行されているものとする．

低優先度タスクが起動する時刻 0 から，高優先度タスクが起動して低優先度タスクがプリエンプトされる時刻 4 までの間で，アプリケーションが得られる処理量は両プロセッサで一致している．したがって，個別プロセッサ上でプリエンプトされるタスクは，統合プロセッサ上でも必ずプリエンプトされるため，同一アプリケーション内でタスク切換えが発生する場合でもタスクの実行順序は一致する．

状況 3 はアプリケーションイベントの間で発生するため，タスクの切り替えタイミングは両プロセッサ上で一致しない可能性がある．しかし，アプリケーションイベントでは実行可能なタスクとその優先順位が一致することから，実行中のタスクが終了した後に実行されるタスクは一致する．以上より，タスク切替えが発生する状況では，両プロセッサ上で同一のタスクが実行されることから，すべてのタスクの実行順序は一致するといえる．

補題 2 と 3 より，次の定理が導かれる．

定理 1 個別プロセッサ上でデッドラインを満たして実行可能なタスクは，高性能

プロセッサ上でもデッドラインを満たして実行可能である．

（証明）補題 2 と補題 3 より，両プロセッサ上の任意のアプリケーションイベント間でタスクの実行順序と処理量が一致する．したがって，任意のタスクの起動時刻から，そのタスクの実行が終了した直後のアプリケーションイベントの間で実行されるすべてのタスクの実行順序と得られる処理量は一致することになる．よって，個別プロセッサ上で起動時刻からデッドラインまでに実行を完了するタスクは，統合プロセッサ上でもデッドラインまでに実行を完了する．

定理 1 により，個別プロセッサでデッドラインを満たせるタスクは高性能プロセッサ上でもデッドラインを満たして実行可能であることが示された．よって，時間保護アルゴリズム 1 は時間保護の 3 つの要件を満たす．

3.4　まとめ

本章では，ハードリアルタイム性を要求される組込みシステムを対象として，時間保護の 3 つの要件をすべて満たすことができる時間保護アルゴリズム 1 を提案した．このアルゴリズムは，Deng らのアルゴリズムを修正したもので，タスクの起動時刻とデッドラインが既知であることを前提として時間保護機能を実現する階層型スケジューリングである．最後に，提案アルゴリズムが時間保護の要件を満たすことを証明した．

第4章　起動時刻が不明なタスクを含むアプリケーションへの対応

4.1　概要

リアルタイムアプリケーションは，外部イベントが発生したタイミングで動作を開始するというような処理を含む場合が多く，このような処理の正確な起動時刻を，事前に把握することは困難である．時間保護アルゴリズム 1 では，適用するための条件として，タスクの起動時刻とデッドラインの 2 つの情報が得られることを前提としており，起動時刻が不明なタスクが含まれるアプリケーションに適用することができない．そこで，BSS アルゴリズムと PShED アルゴリズムをベースに，タスクのデッドライン情報のみを使って時間保護の要件を満たすスケジューリングアルゴリズム（時間保護アルゴリズム 2 と呼ぶ）を提案する．時間保護アルゴリズム 2 でも，統合前に固定優先度ベーススケジューリングによりスケジュール可能であったアプリケーションを単一のプロセッサに統合することを想定する．時間保護アルゴリズム 2 の特徴は，適用するために必要な情報が時間保護アルゴリズム 1 より少なくてすむが，その一方で，時間保護の要件 (3) を満たすことができない．そのため，QoS 制御されたタスクが含まれないアプリケーションにのみ適用できる．時間保護アルゴリズム 2 の詳細な動作を示し，タスクのスケジューリングに特化したシミュレータを用いてアルゴリズムの正当性を確認する．さらに，時間保護の要件 (1) と要件 (2) を満たせることを証明する．

本章の構成を述べる．まず 4.2 節で，アプリケーション統合におけるシステムモデルを整理する．4.3 節では，BSS アルゴリズムと PShED アルゴリズムの問題点を指摘し，4.4 節で，時間保護アルゴリズム 2 について述べる．4.5 節では，時間保護アルゴリズム 2 が時間保護の要件 (1) と (2) を満たすことを証明する．4.6 節では，スケジューリングシミュレータを用いて BSS とスケジュール可能性を比較する．4.7 節で結論を述べる．

4.2 システムモデル

本章では，統合プロセッサはシングルプロセッサで，N 個のアプリケーション $A_1, A_2, \ldots, A_n$ を動作させるものとする．アプリケーション A_i は，2 項組 (U_i, d_i) で表す．ここに，U_i はシステム構築時に設定されるシェア，d_i はアプリケーションの絶対デッドライン（アプリケーションデッドラインと呼ぶ）である．アプリケーションデッドラインは，アプリケーションに属する実行可能なタスクの絶対デッドラインの中でもっとも早い時刻であり，システムの動作中に随時変化する．

A_i は，タスクの集合 τ_i=$\{\tau_{i1}, \tau_{i2}, \ldots\}$ で構成される．タスク τ_{ij} は，4 項組 $(P_{ij}, D_{ij}, C_{ij}, d_{ij})$ で表される．ここに，P_{ij} は実行優先度，D_{ij} は相対デッドライン，C_{ij} は最悪実行時間である．P_{ij} と D_{ij} は，タスクの設計時に決定するものとする．タスク τ_{ij} と τ_{ik} の優先度が $P_{ij} > P_{ik}$ であるとき，τ_{ij} は，τ_{ik} より高い優先度をもつものとする．タスクの実行時間 C_{ij} は，性能 1 をもつ統合プロセッサでの実行時間である．したがって，性能 1 のプロセッサで実行時間 C_{ij} のタスクは，統合前の性能 U_i をもつ個別プロセッサでは，$\frac{C_{ij}}{U_i}$ で実行が完了する．なお，C_{ij} は，4.5 節で述べる証明でのみ使用するパラメータであり，タスクをスケジューリングする際には使用しない．d_{ij} はタスクの絶対デッドラインで，システムの動作中に，タスクの起動時刻に相対デッドラインを加算して計算する．

その他に，本章では次の前提をおく．

- タスクの起動要求は周期的に発生するとは限らない．すなわち，周期タスクに加えて，非周期タスクも対象とする．
- タスク間の通信が存在する場合には非同期とする．タスクが待ち状態になるような同期通信はない．
- タスクの優先度は，デッドラインモノトニックにより割当てられる．

4.3 BSS アルゴリズムと PShED アルゴリズムの問題点

BSS アルゴリズムおよび，それを改良した PShED アルゴリズム（本章では，BSS アルゴリズムに統一して表記する）は，時間保護の要件のうち，次の 2 つの要件を満たすことが証明されている．

- 要件 (1) アプリケーションのプロセッサ利用率の保証
  グローバルスケジューリングアルゴリズムとして EDF スケジューリングを採用すると，各アプリケーションは，設定されたシェアに基づくバジェットが割当てられ，アプリケーションデッドラインまでに使い切ることができる．

- 要件 (2) タスクのスケジュール可能性の保証
  各アプリケーションのタスクについて，統合前に，EDF スケジューリングによりスケジュール可能なタスクは，ローカルスケジューリングアルゴリズムとして EDF スケジューリングを採用する BSS アルゴリズムにより統合後もスケジュール可能である．

要件 (1) は，アプリケーション間でプロセッサ時間が保護されることを保証している．しかし，アプリケーションに属するタスクがそれぞれのデッドラインを満たすことは保証していない．このことを保証するのが要件 (2) であるが，BSS アルゴリズムの場合は，ローカルスケジューリングアルゴリズムが EDF スケジューリングであることを前提としており，2.2.2 項で述べたように，任意のアプリケーションに対して固定優先度スケジューリングを適用できるわけではない．

このことを簡単な例を用いて示す．いま，周期 5, (相対性能 0.5 の個別プロセッサにおける）最悪実行時間 3 の高優先度タスク τ_{11} と周期 12，最悪実行時間 4 の低優先度タスク τ_{12} の 2 タスクで構成されるアプリケーション A_1 と，周期 12，最悪実行時間 12 の 1 タスクで構成されるアプリケーション A_2 を性能 1 のプロセッサに統合することを考える．どちらのアプリケーションも，個別プロセッサでスケジュール可能で，A_1 に着目すると個別プロセッサでは図 4.1 に示すようにスケジュールされる．ここで，A_1 と A_2 にそれぞれシェアを 50%割当て，ローカルスケジューリングアルゴリズムにプリエンプティブな固定優先度スケジューリングを用いた BSS アルゴリズムでスケジュールする．図 4.2 に示すように，A_1 の τ_{12} は時刻 12 でデッドラインをミスしてしまう．これは，A_1 と A_2 の 2 つのアプリケーションが動作した結果，τ_{12} が，τ_{11} の 3 回目の起動の前に実行を完了できなかったことが原因である．時刻 12 までに各アプリケーションが使用したバジェットを考えると，A_1 と A_2 は共に 6 単位時間であり，プロセッサ利用率は 50%である．

得られたプロセッサ利用率は十分であるにも関わらず，A_1 のタスクがデッドラインをミスしてしまった原因を考えると，τ_{11} の 3 回目の起動要求が発生したときのアプリケーションデッドライン（すなわち，τ_{12} の絶対デッドラインである時刻 12）より遅いデッドライン（時刻 15）をもつ高優先度タスク τ_{11} が，低優先度タス

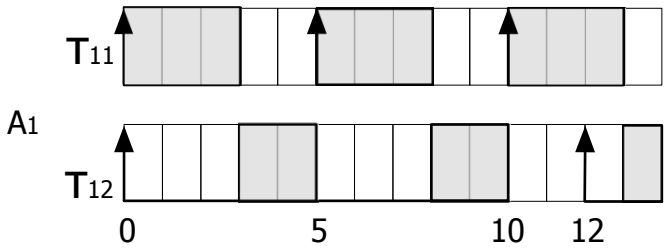

図 4.1: 固定優先度スケジューリングによる A_1 のスケジューリング例

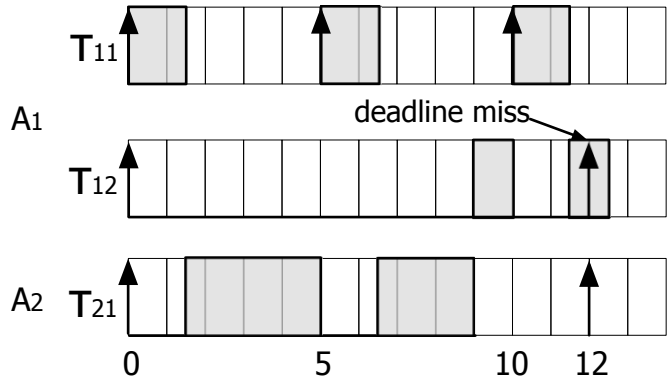

図 4.2: BSS アルゴリズムのローカルスケジューラに固定優先度スケジューリングを使用した場合の A_1 と A_2 のスケジューリング例

ク τ_{12} を実行するためのバジェットを使って，低優先度タスクに優先して実行されたと考えることができる．

4.4 時間保護アルゴリズム 2

4.4.1 アルゴリズムが満たす性質

本章では，ローカルスケジューリングにおいて，アプリケーションデッドラインより遅い絶対デッドラインをもつ高優先度タスクの起動時刻を遅延させることで，低優先度タスクがデッドラインをミスすることを防ぐアルゴリズムを提案する．高優先度タスクの起動を遅延させ，実行中の低優先度タスクを，処理が完了するまで優先的に実行する．この意味で，提案アルゴリズムのローカルスケジュー

リングアルゴリズムは，厳密な優先度ベーススケジューリングではない．しかしながら，優先度ベーススケジューリングの重要な性質の 1 つである，高い優先度を持つタスクが，その実行中に低い優先度をもつタスクに邪魔されることはない，という性質を満たすことができる．これにより，低い優先度をもつタスクには実行を妨げられないことを前提として，高い優先度をもつタスクでは（セマフォやミューテックスなどを用いて）明示的な排他制御をせずに，データの排他制御を実現している場合でも，統合後に排他制御が崩れてしまうことはない．

4.4.2　タスクモデル

提案アルゴリズムにおけるタスクは，図 4.3 に示すように 4 つの状態がある．システムの動作開始時には，すべてのタスクは T_DORMANT 状態である．タスクの起動要求が発生すると，そのタスクが起動遅延条件（詳細は，4.4.5 節で述べる）を満たすかどうかをチェックする．起動遅延条件を満たさない場合は，タスクは T_READY 状態に遷移する（図 4.3 (1)）．一方，起動遅延条件を満たす場合には，タスクは T_DELAYED 状態に遷移する（図 4.3(2)）．T_DELAYED 状態のタスクは，起動遅延条件を満たさなくなった時点で T_READY 状態に遷移する（図 4.3 (3)）．T_READY 状態で最も優先度の高いタスクは T_RUNNING 状態になる（図 4.3 (4)）．T_RUNNING 状態のタスクより高い優先度をもつタスクが T_READY 状態になると，T_RUNNING 状態のタスクは T_READY 状態になり（図 4.3 (5)），新しく起動した高優先度タスクが T_RUNNING 状態に遷移する．T_RUNNING 状態のタスクの実行が完了すると，T_RUNNING 状態から T_DORMANT 状態に遷移する（図 4.3 (6)）．

4.4.3　アプリケーションモデル

アプリケーションには，図 4.4 に示すように 4 つの状態がある．アプリケーションの状態は，そのアプリケーションに属するタスクの状態とバジェットの量で決まる．まず，システムの動作開始時などで，実行できる状態（T_READY 状態もしくは T_RUNNING 状態）のタスクが存在しないアプリケーションは A_DORMANT 状態になる．A_DORMANT 状態のアプリケーションに属するタスクのうち 1 つでも T _READY 状態になり，かつ，バジェットが 0 でない場合，アプリケーションは A_READY 状態に遷移する（図 4.4(1)）．一方，T_READY 状態のタスクが

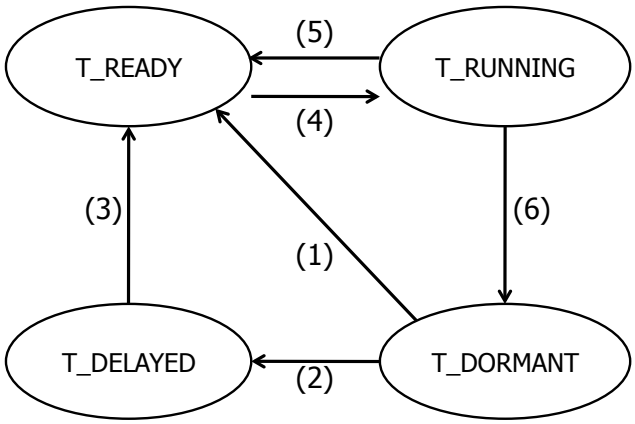

図 4.3: タスクの状態遷移図

存在しても，バジェットが 0 の場合，アプリケーションは A_DORMANT 状態から A_EXPIRED 状態に遷移する（図 4.4(2)）．A_EXPIRED 状態のアプリケーションにバジェットが割当てられると，A_READY 状態に遷移する（図 4.4(3)）．A_READY 状態のアプリケーションの中で最も絶対デッドラインの早いアプリケーションは，A_RUNNING 状態に遷移する（図 4.4(4)）．A_RUNNING 状態のアプリケーションより，絶対デッドラインの早いアプリケーションが A_READY 状態になると，A_RUNNING 状態のアプリケーションは A_READY 状態になり（図 4.4 (5)），新しく A_READY 状態になったアプリケーションが A_RUNNING 状態に遷移する．A_RUNNING 状態のアプリケーションの T_RUNNING 状態のタスクが実行された結果，実行できるタスクがなくなった場合には，アプリケーションは A_RUNNING 状態から A_DORMANT 状態に遷移する（図 4.4(6)）．

4.4.4 スケジューラ構成

提案アルゴリズムは，アプリケーションに属するタスクをスケジュールするローカルスケジューラと，どのアプリケーションのタスクをプロセッサで実行するかを決定するグローバルスケジューラを 2 階層に配置した階層型スケジューリングアルゴリズムである．階層型スケジューラの内部構成を図 4.5 に示す．

ローカルスケジューラは，固定優先度スケジューリングに対して，タスク起動遅延の仕組みを導入したアルゴリズムにより，タスクをスケジュールする．ローカルスケジューラでは，次の 3 つのキューを管理する．

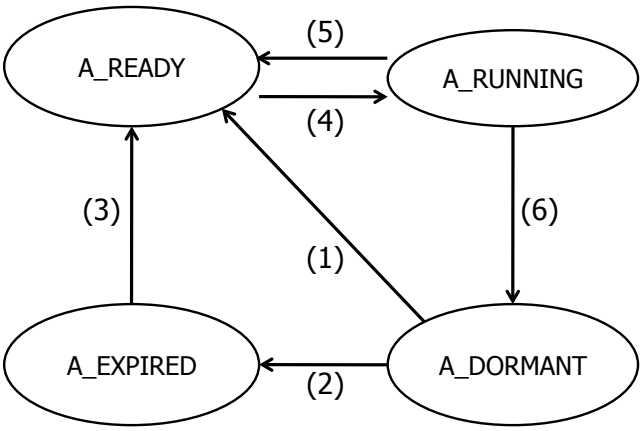

図 4.4: アプリケーションの状態遷移図

- タスクデッドラインキュー（Task deadline queue）
  絶対デッドラインが設定されている状態（すなわち，T_DELAYED 状態，T_READY 状態，T_RUNNING 状態のいずれかの状態）のタスクが，絶対デッドラインの早い順に接続されたキューである．タスクの起動要求が発生した時点で接続される．

- タスクレディキュー（Task ready queue）
  実行できる状態（すなわち，T_READY 状態もしくは T_RUNNING 状態）のタスクが，優先度の高い順に接続されたキューである．優先度が同じ場合は，FIFO 順に接続される．

- タスク起動遅延キュー（Activation-delayed task queue）
  タスクの起動遅延条件を満たしている（すなわち，T_DELAYED 状態）のタスクが，FIFO 順に接続されたキューである．

グローバルスケジューラは，EDF スケジューリングでアプリケーションをスケジュールし，最も早い絶対デッドラインをもつアプリケーションの T_RUNNING 状態のタスクを実行する．アプリケーションごとに使用できるバジェットは，バジェットリストで管理される（詳細は，4.4.6 項で述べる）．あるアプリケーションの絶対デッドラインが変わると，すべてのアプリケーションを再スケジュールする．加えて，新しい絶対デッドラインに対して使用できるバジェットを，そのアプリケーションに設定されたシェアを用いて計算する．グローバルスケジューラは，その時点での絶対デッドラインに対して割当てられているバジェットを上限として，

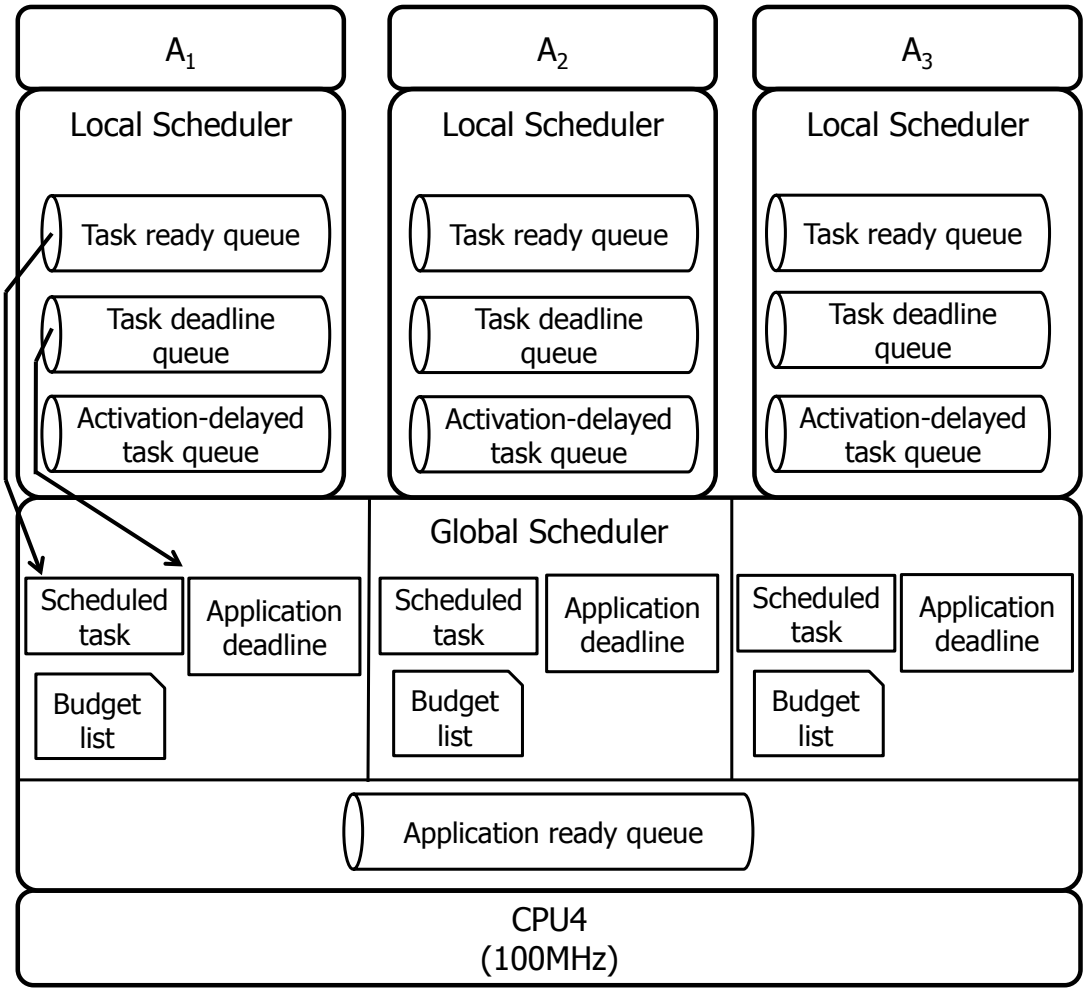

図 4.5: 時間保護アルゴリズム 2 における階層型スケジューラの内部構成

アプリケーションのタスクを実行する．アプリケーションのタスクを実行すると，その実行時間分だけバジェットを減らす．バジェットが 0 になると，次に絶対デッドラインの早いアプリケーションに，強制的に，実行を切り替える．

グローバルスケジューラでは，アプリケーション毎に次のデータ構造を管理する．

- 実行すべきタスク（Scheduled task）
  ローカルスケジューラでスケジュールされた T_RUNNING 状態のタスクである．アプリケーションがスケジュールされると，ここに格納されたタスクがプロセッサで実行される．

- アプリケーションレディーキュー（Application ready queue）
  実行できる状態（A_READY 状態もしくは A_RUNNING 状態）のアプリケー

ションが，絶対デッドラインの早い順番に接続されたキューである．アプリケーションが実行できる状態に遷移した時点で接続される．

- アプリケーションデッドライン（Application deadline）
  アプリケーションに属するタスクの絶対デッドラインの中でもっとも早い時刻（すなわち，ローカルスケジューラのタスクデッドラインキューの先頭に接続されたタスクの絶対デッドライン）である．グローバルスケジューラは，ここに格納された絶対デッドラインを用いて，アプリケーションをスケジュールする．

- バジェットリスト（Budget list）
  アプリケーションのバジェットを管理するためのデータ構造である．詳細は，4.4.6 節で述べる．

4.4.5　スケジューリングアルゴリズム

まず，タスクの起動要求が発生した時の流れを説明する．ここでは，時刻 t に，アプリケーション A_i のタスク τ_{ij} の起動要求が発生したとする．

1. τ_{ij} の相対デッドライン D_{ij} を用いて，次の計算式で絶対デッドライン d_{ij} を計算し，τ_{ij} をタスクデッドラインキューに挿入する．

$$d_{ij} = D_{ij} + t$$

2. τ_{ij} がタスクデッドラインキューの先頭に接続された場合，A_i の新しいデッドライン d_{ij} をグローバルスケジューラに通知する．グローバルスケジューラは，必要に応じて d_{ij} に対応するバジェット要素をバジェットリストに追加する．その後，アプリケーションの絶対デッドライン d_i を d_{ij} に更新する．

3. τ_{ij} と同じアプリケーションに属し，かつ時刻 t においてタスクレディキューに接続されたタスク τ_{ik} に対して，τ_{ij} が次の起動遅延条件を満たすかどうかをチェックする．

   - τ_{ij} の優先度が τ_{ik} の優先度より高い（$P_{ij} > P_{ik}$）．
   - τ_{ij} の絶対デッドラインが τ_{ik} の絶対デッドラインより遅い（$d_{ij} > d_{ik}$）．

4. τ_{ij} が起動遅延条件を満たす τ_{ik} が 1 つでも存在する場合は，τ_{ij} をタスク起動遅延キューの最後尾に挿入する．一方，τ_{ij} が起動遅延条件を満たさない場合は，τ_{ij} をタスクレディーキューに挿入する．

5. τ_{ij} がタスクレディーキューの先頭に接続された場合には，アプリケーション内で実行するタスクを切り替えるため，グローバルスケジューラに τ_{ij} を通知する．グローバルスケジューラは，アプリケーションの実行すべきタスクを τ_{ij} に更新する．

6. グローバルスケジューラは，絶対デッドラインの最も早いアプリケーションの実行すべきタスクをプロセッサで実行する．

次に，タスクの実行が中断もしくは完了した時の流れを説明する．ここでは，タスク τ_{ij} が時間 e だけ実行されたものとする．

1. τ_{ij} の属する A_i のバジェットリストを更新する．τ_{ij} の実行が完了した場合には，以降の (2) から (5) を実行する．

2. τ_{ij} をタスクデッドラインキューとタスクレディーキューから削除する．

3. タスク起動遅延キューの先頭に接続されているタスクから順に，そのタスク（τ_{ik} とする）が起動遅延条件を満たすかどうかをチェックする．

4. τ_{ik} が起動遅延条件を満たす場合は，そのままタスク起動遅延キューに接続しておく．起動遅延条件を満たさない場合は，τ_{ik} をタスク起動遅延キューから削除し，タスクレディーキューに挿入する．

5. 同様に，τ_{ik} の次に接続されたタスクが起動遅延条件を満たすかどうかをチェックする．タスク起動遅延キューに接続されたすべてのタスクについて，起動遅延条件をチェックしたら終了する．

4.4.6 バジェット管理アルゴリズム

提案アルゴリズムでは，グローバルスケジューラでアプリケーションのバジェットを管理するために，バジェットリストを用いる．このアルゴリズムは，アプリケーションが使用できるバジェットを計算するもので，本質的には，BSS アルゴリズムのバジェット管理アルゴリズム [23] と違いはない．

A_i のバジェットリストは，バジェット要素 $l_{ij} = (d_{ij}, b_{ij})$ で構成される．ここに，d_{ij} はバジェットの絶対デッドライン，b_{ij} は d_{ij} までのバジェットである．すなわち，l_{ij} は，A_i が d_{ij} までに b_{ij} のバジェットを使用できることを意味する．バジェットリスト内では，バジェット要素が絶対デッドラインの早い順に並ぶ．ここでは，バジェットリストに対する 3 つの操作を定義する．

バジェット要素の追加

A_i のローカルスケジューラが，グローバルスケジューラに対してアプリケーションのデッドライン d_{ij} を通知した時点（時刻 t とする）で，d_{ij} に一致するデッドラインをもつバジェット要素がバジェットリスト中に存在しない場合，次の処理により新しい要素 l_{ij} をバジェットリストに追加する．

1. 次の条件を満たす，l_{ij} の挿入位置を探す．

$$\exists l_{i(k-1)}, l_{ik}\ d_{i(k-1)} < d_{ij} < d_{ik}$$

2. d_{ij} までに使用できるバジェット b_{ij} を次の式で計算する．A_i のデッドライン d_i が早い時刻になる場合（$d_i > d_{ij}$）と，遅い時刻になる場合（$d_i < d_{ij}$）とで計算式が異なる．

$$b_{ij} = \begin{cases} \min\{D_i * U_i, (d_{ij} - d_{i(k-1)}) * U_i + b_{i(k-1)}, b_{ik}\} & (d_i > d_{ij}) \\ \min\{(d_{ij} - d_{i(k-1)}) * U_i + b_{i(k-1)}, b_{ik}\} & (d_i < d_{ij}) \end{cases}$$

3. l_{ij} を，$l_{i(k-1)}$ と l_{ik} の間に挿入する．

バジェットリストの更新

次の事象が発生したとき，バジェットリストを更新する．

- タスクの実行を完了したとき
- アプリケーションのバジェットが 0 になったとき
- アプリケーション内で実行するタスクを切り替えるとき
- 実行するアプリケーションを切り替えるとき

タスクを実行した時間を e，アプリケーションの絶対デッドラインに対応するバジェット要素 l_{ij} がバジェットリストの j 番目にあるとすると，次の処理を実行する．

- $b_{ik} \leftarrow b_{ik}$ - e (k $\geq$ j)

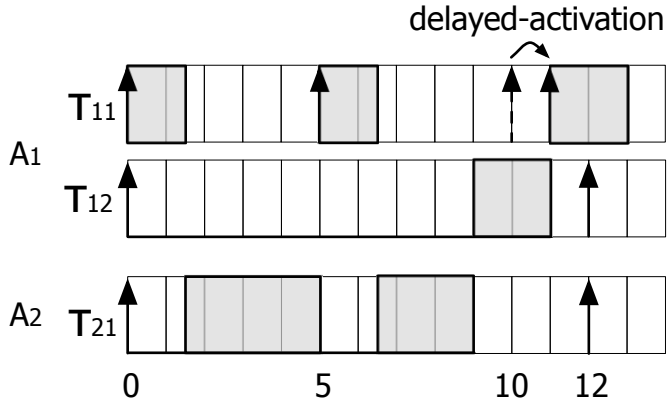

図 4.6: 時間保護アルゴリズム 2 によるスケジューリングの例

- l_{ik} を削除する．$(\mathrm{k} < \mathrm{j} \wedge b_{ik} > b_{ij})$

バジェット要素の削除

バジェットリストのバジェット要素のうち，今後のバジェット計算で使用されない要素はバジェットリストから削除できる．時刻 t で，デッドラインが d_{ik} に一致するタスクがすでに完了しており，かつ，次のいずれかの条件を満たすとき，その要素 l_{ik} を削除できる [23].

- $d_{ik} \leq t$
- $b_{ik} > (d_{ik} - t)U_i$

4.4.7 動作例

図 4.2 のアプリケーションを提案アルゴリズムによりスケジュールした例を図 4.6 に示す．時刻 10 で発生した高優先度タスク τ_{11} の起動要求は，実行中のタスク τ_{12} に対して起動遅延条件を満たすため，τ_{11} の起動は τ_{12} が完了するまで遅延する．その結果，τ_{12} は τ_{11} に優先して実行され，デッドラインである時刻 12 までに完了できる．さらに，起動が遅延した τ_{11} も，デッドラインである時刻 15 までに実行を完了できる．

4.5　証明

時間保護アルゴリズム 2 の正当性を証明するために，BSS アルゴリズムの証明手法 [22] をベースに，時間保護の要件 (1) と要件 (2) が成り立つことを示す．時間保護の要件 (1) を満たすことの証明は，グローバルスケジューリングアルゴリズムとバジェット管理手法に依存し，ローカルスケジューリングアルゴリズムには依存しない．そのため，BSS アルゴリズムの証明方法をそのまま適用できる．そこで，本論文では，ローカルスケジューリングにタスク起動遅延の仕組みを導入した固定優先度スケジューリングを適用する場合に，要件 (2) を満たせることを証明する．

まず，ある時間内におけるアプリケーションの処理要求に関して次の定義をする．

定義 1 区間 $[a, b]$ におけるアプリケーション A_i の処理要求を次のように定義する．

$$D_i(a, b) = \sum_{\substack{a_{ij} \geq a \\ d_{ij} \leq b}} C_{ij} \tag{4.1}$$

この処理要求の定義を用いて，アプリケーションに属するタスクのスケジュール可能性について，次の定理が成り立つ．

定理 2 性能 *1* の統合プロセッサにおいてシェア U_i を設定したアプリケーション A_i が次の式を満たすとき，A_i は提案アルゴリズムによりスケジュール可能である．

$$\forall a, b \ a < b \ D_i(a, b) \leq (b - a)U_i \tag{4.2}$$

（証明） 定理 2 の式 (4.2) を満たす状況において，A_i のタスクがデッドラインをミスすると仮定して矛盾を導く．いま，時刻 t_{ov} で A_i のバジェット l_{ij} が 0 になったとする．すなわち，バジェットリストには，$l_{ij} = (d_{ij}, 0)$ が存在する．ここで，次の条件を満たすバジェットリスト中の要素 $l_{ik} = (d_{ik}, b_{ik})$ について考える．

- k $\geq$ j
- $b_{ik} = 0$

言い換えると，l_{ik} は，バジェットリストの中でバジェットが 0 である，最もデッドラインの遅いバジェット要素である．このとき，バジェットリストは次のようになっている．

$$\begin{aligned} l_{i(k-1)} &= (d_{i(k-1)}, b_{i(k-1)}) = (d_{i(k-1)}, 0) \\ l_{ik} &= (d_{ik}, b_{ik}) = (d_{ik}, 0) \\ l_{i(k+1)} &= (d_{i(k+1)}, b_{i(k+1)}) \leftarrow (d_{i(k+1)} \neq 0) \end{aligned}$$

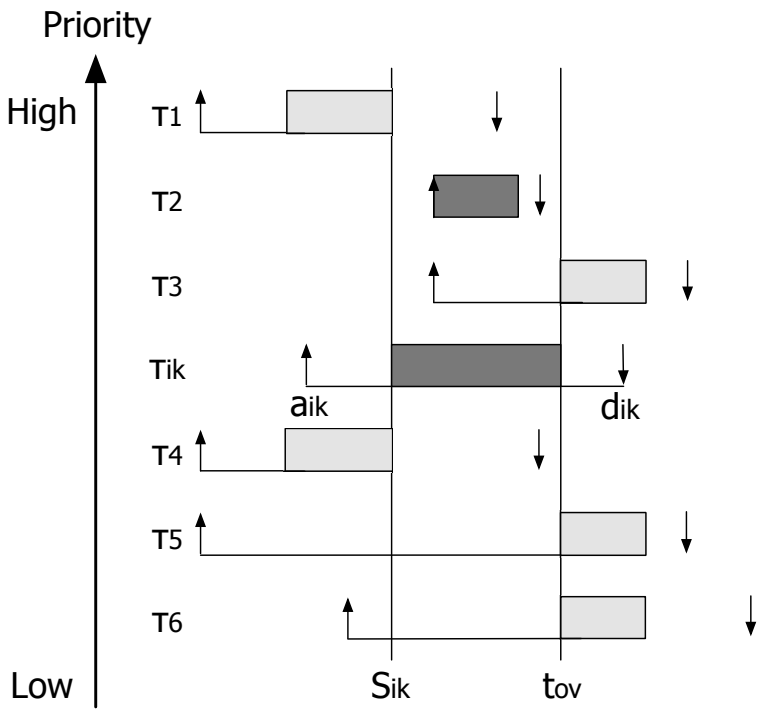

図 4.7: バジェット b_{ik} を使用して実行するタスク

l_{ik} がバジェットリストに追加された時刻を s_{ik} とすると，s_{ik} の時点で d_{ik} は最も早いデッドラインである．時刻 s_{ik} では，提案アルゴリズムのバジェット管理アルゴリズムにより，d_{ik} に対するバジェット b_{ik} が計算され割当てられる．ここで，b_{ik} を使用して実行されるタスクの条件を考えると，図 4.7 に示すように，$[s_{ik}, t_{ov}]$ で実行されるタスクの中で最も早く起動したタスクの起動時刻を a_{ik} とすると，a_{ik} より後に起動し，かつデッドラインが d_{ik} より早いタスクに限定される．その理由は次の通りである．すなわち，タスク起動遅延付きの固定優先度スケジューリングでは，τ_1 や τ_4 のようなタスクは τ_{ik} に優先して実行され，s_{ik} より早く完了するため，b_{ik} を使用して実行されることはない．さらに，τ_3，τ_5，τ_6 のようなタスクは，τ_{ik} が完了するまでは実行されず，バジェット b_{ik} を使用して実行されることはない．さらに，タスクの優先度がデッドラインモノトニックで割当てられることを前提としていることから，τ_{ik} より優先度が高いかつ相対デッドラインが長いタスク，および，τ_{ik} より優先度が低いかつ相対デッドラインが短いタスクは存在しない．したがって，b_{ik} を使用して実行するタスクは，τ_2 と τ_{ik} 自身のように，a_{ik} より後に起動し，かつデッドラインが d_{ik} より早いタスクである．

時刻 t_{ov} でバジェット b_{ik} は 0 になるという前提から，b_{ik} を使用して実行するタスクの合計実行時間は b_{ik} を越えることになる．すなわち，次の式が成立する．

$$\sum_{\substack{a_{ij}\geq a_{ik}\\ d_{ij}\leq d_{ik}}} C_{ij} > b_{ik} \tag{4.3}$$

ここで，b_{ik} の値は計算方法により，次の 3 つの場合に分かれる．それぞれの場合に，式 (4.3) が成立したときに矛盾が発生することを示す．

- 場合 A：$b_{ik} = b_{i(k+1)}$
  b_{ik} として，l_{ik} の次の要素の残りバジェット $b_{i(k+1)}$ が割当てられる場合である．この場合，b_{ik} が 0 になった時点で，$b_{i(k+1)}$ も 0 になっているはずである．しかし，l_{ik} は，バジェットが 0 で絶対デッドラインの最も遅いバジェット要素であるから，この状況は発生しない．

- 場合 B：$b_{ik} = D_{ik} * U_i$
  s_{ik} の時点で，相対デッドライン D_{ik} に対するシェア分のバジェットが得られた場合である．このとき，次の式が成り立つ．

  $$\sum_{\substack{a_{ij}\geq a_{ik}\\ d_{ij}\leq d_{ik}}} C_{ij} > (d_{ik} - a_{ik})U_i$$

  これは，明らかに式 (4.2) に矛盾する．

- 場合 C：$b_{ik} = b_{i(k-1)} + (d_{ik} - d_{i(k-1)})U_i$
  l_{ik} の前のバジェット要素のバジェット $b_{i(k-1)}$ に依存する場合である．時刻 $s_{i(k-1)}$ で，l_{ik} の前の要素 $l_{i(k-1)}$ が挿入されたとする．$[s_{i(k-1)}, s_{ik}]$ の間に起動したタスクを実行した結果，s_{ik} の時点で，バジェット $b_{i(k-1)}$ を次のように更新する．

  $$b_{i(k-1)} \leftarrow b_{i(k-1)} - \sum_{\substack{a_{ij}\geq a_{i(k-1)}\\ d_{ij}\leq d_{i(k-1)}}} C_{ij}$$

  よって，バジェット b_{ik} は，次のように計算される．

  $$b_{ik} = b_{i(k-1)} - \sum_{\substack{a_{ij}\geq a_{i(k-1)}\\ d_{ij}\leq d_{i(k-1)}}} C_{ij} + (d_{ik} - d_{i(k-1)})U_i$$

  $b_{i(k-1)}$ について，さらに次の 3 つの場合に分かれる．

– 場合 C-A：$b_{i(k-1)} = b_{ik}$
　場合 A と同様の理由で，このような状況はあり得ない．

– 場合 C-B：$b_{i(k-1)} = (d_{i(k-1)} - a_{i(k-1)})U_i$

$$\begin{aligned}\sum_{\substack{a_{ij}<a_{ik}\\ d_{ij}\le d_{ik}}} C_{ij} &> b_{ik} = b_{i(k-1)} - \sum_{\substack{a_{ij}\ge a_{i(k-1)}\\ d_{ij}\le d_{i(k-1)}}} C_{ij} + (d_{ik} - d_{i(k-1)})U_i \\ &= (d_{i(k-1)} - a_{i(k-1)})U_i - \sum_{\substack{a_{ij}\ge a_{i(k-1)}\\ d_{ij}\le d_{i(k-1)}}} C_{ij} + (d_{ik} - d_{i(k-1)})U_i \\ \sum_{\substack{a_{ij}\ge a_{i(k-1)}\\ d_{ij}\le d_{ik}}} C_{ij} &> (d_{ik} - a_{i(k-1)})U_i\end{aligned}$$

　となる．これは，$a = a_{i(k-1)}, b = d_{ik}$ と考えると，式 (4.2) に矛盾する．

– 場合 C-C：$b_{i(k-1)} = b_{i(k-2)}$
　さらに，前のバジェット $b_{i(k-2)}$ の計算方法により 3 つの場合に分かれるが，これまでと同様に $[a_{i(k-2)}, d_{ik}]$ の区間を考えると，場合 A と場合 B では矛盾が発生する．場合 C が続くと，最終的には，タスクの最初の起動時刻までさかのぼることになるが，このときは場合 B に該当するので，やはり式 (4.2) に矛盾する．

以上より，いずれの場合でも式 (4.2) に矛盾が発生するため，定理 2 は成立する．

次に，統合前の前提について，次の定理が成り立つ．

定理 3 個別プロセッサでアプリケーションが固定優先度スケジューリングによりスケジュール可能であるとき，次の式を満たす．

$$\forall a, b \; a \le b \; \frac{D_i(a,b)}{U_i} \le (b-a) \tag{4.4}$$

（証明） 式 (4.4) は，EDF スケジューリングによるスケジュール可能性の必要十分条件であることが証明されている [23]．一方，EDF スケジューリングは最適なスケジューリングアルゴリズムであることが知られており，固定優先度スケジューリングでスケジュール可能なアプリケーションは EDF スケジューリングでもスケジュール可能である [27]．よって，固定優先度スケジューリングでスケジュール可能なアプリケーションは式 (4.4) 満たす．以上より，定理 3 が証明された．

最後に，定理 2 と定理 3 より，次の定理 4 が成り立つ．

定理 4 性能 U_i の個別プロセッサでアプリケーション A_i が固定優先度スケジューリング（ただし，デッドラインモノトニックによる優先度割当てを前提とする）によりスケジュール可能であるとき，性能 *1* の統合プロセッサにおいてシェア U_i を設定すれば，アプリケーション A_i のタスクは提案アルゴリズムによりスケジュール可能である．

（証明）定理 3 の式 (4.4) より，次の式が成り立つ．

$$\forall\ a,b\ a \le b \sum_{\substack{a_{ij}\ge a \\ d_{ij}\le b}} \frac{C_{ij}}{U_A} \quad \le \quad (b-a)$$

$$\forall\ a,b\ a \le b \sum_{\substack{a_{ij}\ge a \\ d_{ij}\le b}} C_{ij} \quad \le \quad (b-a)U_i$$

よって，個別プロセッサで固定優先度スケジューリングによりスケジュール可能なアプリケーション A_i は定理 3 を満たすため，性能 1 の統合プロセッサにおいてシェア U_i を設定すれば，提案アルゴリズムにより A_i はスケジュール可能である．以上より，定理 4 が証明された．

4.6　評価

同一のアプリケーションをスケジューリングし，そのスケジュール可能性を比較するため，ローカルスケジューリングに固定優先度スケジューリングを採用した BSS アルゴリズムと，提案アルゴリズムを実装したタスクスケジューリング・シミュレータを開発した．

シミュレーションによる評価の流れは，次の通りである．まず，スケジュール可能かどうかを判定する評価対象のアプリケーションとして，固定優先度スケジューリングでスケジュール可能なアプリケーションを生成する．評価対象アプリケーションは複数のタスクで構成され，起動周期や最悪実行時間などのパラメータは，一様分布に従う乱数で生成する．相対デッドラインは，次の起動時刻に一致するものとする．次に，評価対象アプリケーションとテストベンチアプリケーションを統合し，それぞれに同じシェアを割当て，ローカルスケジューリングに固定優先度スケジューリングを採用した BSS アルゴリズムと，提案アルゴリズムの 2 つのアルゴリズムでスケジューリングする．ここで，テストベンチアプリケーションとは，評価対象アプリケーションがスケジュールされるタイミングを，様々に変化させることを目的としたアプリケーションであり，タスク 1 つで構成される．このタスク

表 4.1: 評価条件

評価番号	テストベンチアプリケーション数	タスクの起動属性	平均プロセッサ利用率 (%)	平均タスク数
1	1	周期	89.49	4.08
2	1	非周期	89.49	4.08
3	1	非周期	96.61	8.45
4	3	非周期	94.03	3.80

表 4.2: シミュレーション結果

評価番号	評価対象アプリケーション数	スケジュール可能なアプリケーション数	
		BSS	時間保護アルゴリズム 2
1	10,000	8,330	10,000
2	10,000	9,355	10,000
3	10,000	6,142	10,000
4	1,000	978	1,000

の相対デッドラインを，一様分布に従う乱数で実行時に決定し，テストベンチアプリケーションの絶対デッドラインを様々に変化させる．これにより，評価対象アプリケーションが先に実行される状況や，逆に，評価対象アプリケーションに優先してテストベンチアプリケーションが実行される状況を作り出すことができる．最後に，生成したアプリケーション数に対するスケジュール可能なアプリケーション数（スケジュール可能率と呼ぶ）を比較する．実施した 4 つの評価の条件設定を表 4.1 に，シミュレーション結果を表 4.2 に示す．

評価 1

評価 1 として，周期タスクのみで構成され，デッドラインモノトニックで優先度を割当てた評価対象アプリケーションを 10,000 個生成した．アプリケーションを構成するタスクの周期の最大値は 50 単位時間で，最悪実行時間の最大値は 10 単位時間である．これらのアプリケーションは，固定優先度スケジューリングによりスケジュール可能であることを確認している．評価対象アプリケーションと，テストベンチアプリケーション 1 つを統合し，それぞれのシェアを 50%に設定した．シミュレーション時間は，10,000 単位時間である．シミュレーションの結果，BSS アルゴリズムのスケジュール可能率は約 83%であるのに対して，提案アルゴリズ

ムでは 100%であった．

評価 2

評価 2 として，評価対象アプリケーションを構成するタスクを非周期タスクに変更してシミュレーションした．テストベンチアプリケーションの数，タスクのパラメータを生成する際の最大値は，評価 1 と同様である．非周期タスクの起動間隔は，シミュレーションの実行時に，次の式で計算している．

$$p - \frac{\log(1 - rand())}{\lambda}$$

ここに，第 1 項 p は，タスクの起動周期として生成した値で，ここでは最小起動間隔を意味する．第 2 項は，指数分布に従う乱数である．λ は 4 に固定した．$rand()$ は，[0,1] の実数をランダムに生成する関数である．シミュレーションの結果，BSS アルゴリズムのスケジュール可能率は約 93%であるのに対し，提案アルゴリズムは 100%であった．評価 2 で使用した評価対象アプリケーションの平均タスク数と平均プロセッサ利用率は評価 1 と同程度である．なお，プロセッサ利用率は，すべての非周期タスクが，最小到着間隔で起動した場合での値である．評価対象アプリケーションのプロセッサ利用率が評価 1 と同じであるにも関わらず，BSS アルゴリズムのスケジュール可能率が評価 1 に比べて高くなった理由として，最小到着間隔に基づいて計算したプロセッサ利用率よりも，シミュレーション中のプロセッサ利用率が低いため，BSS アルゴリズムでスケジュール可能なアプリケーション数が増加したと考えられる．

評価 3

評価 3 として，評価対象アプリケーションを構成するタスク数を増やして評価するため，評価 2 のタスク生成時のパラメータ設定を変更してシミュレーションした．具体的には，アプリケーションを構成するタスクの周期の最大値は 50 単位時間で変更せずに，最小値を 20 単位時間に設定した．さらに，最悪実行時間の最大値を 10 単位時間から 4 単位時間に変更した．BSS アルゴリズムのスケジュール可能率は約 62% に対して，提案アルゴリズムでは 100%であった．BSS アルゴリズムのスケジュール可能率が評価 2 に比べて低くなった理由として，評価対象アプリケーションを構成するタスク数が評価 2 に比べて増加したことで，4.3 節で述べた，BSS アルゴリズムでデッドラインをミスする状況が発生しやすくなったためと考えられる．評価 3 においても，提案アルゴリズムではすべての評価対象アプリケーションをスケジュールできている．

評価 4

評価 4 として，3 つ以上のアプリケーションを統合する状況で評価するため，評価 2 で用いた評価対象アプリケーションに対して，テストベンチアプリケーションを 3 つ統合した．シミュレーション時間は，10,000 から 100,000 単位時間に増やしてシミュレーションした．動作するアプリケーションが 4 つに増加したため，各アプリケーションのシェアをそれぞれ 25%に設定した．アプリケーションを構成するタスクの周期の最大値は 50 単位時間，最悪実行時間の最大値は 10 単位時間のままである．BSS アルゴリズムのスケジュール可能率は約 97%に対して，提案アルゴリズムでは 100%であった．

以上より，すべての評価において，提案アルゴリズムを適用することで，評価対象アプリケーションをスケジュールできることを確認した．

4.7　まとめ

本章では，統合前に固定優先度スケジューリングによりスケジュール可能なリアルタイムアプリケーションの優先度設計を変更することなく統合できる時間保護アルゴリズム 2 を提案した．提案アルゴリズムが，時間保護の要件 (1) と (2) を満たすことを証明した．さらに，スケジューリングシミュレータを用いて，BSS アルゴリズムとスケジュール可能性を比較した．その結果，BSS アルゴリズムでは統合後にデッドラインをミスしてしまうアプリケーションを，提案アルゴリズムによりスケジュール可能であることを確認した．

第5章　階層型スケジューリングフレームワーク

5.1　概要

リアルタイムアプリケーションを統合するためのスケジューリングアルゴリズムとして，第3章で時間保護アルゴリズム1を，第4章では，時間保護アルゴリズム1の前提条件を緩めた時間保護アルゴリズム2を提案した．その他にも，これまで数多くのスケジューリングアルゴリズムが提案されているが，それぞれのアルゴリズムが満たすことのできる，時間保護の要件や，アルゴリズムを適用するための前提条件は異なる．

自動車制御システムを始め，実際の分散リアルタイムシステムのアプリケーションでは，要求される時間要件（リアルタイム性，応答性，出力のタイミングなど）や，開発者が統合段階で知ることのできるパラメータ（アプリケーションに所属するタスクの起動時刻，最悪実行時間，デッドラインなど）はそれぞれ異なる場合が多い．したがって，統合するすべてのアプリケーションに対して同一のスケジューリングアルゴリズムを適用することは困難である．

本章では，これまで提案された階層型スケジューリングアルゴリズムにおいて，共通する部分と異なる部分を整理し，異なる部分については，アプリケーション毎に選択できるよう柔軟性を高めたスケジューリングフレームワークを提案する．具体的には，アプリケーションごとに，アプリケーションに要求される時間要件や，統合段階で既知であるパラメータに応じて，ローカルスケジューラのアルゴリズムを選択できるようにし，グローバルスケジューラは共通部分として統一する．共通のグローバルスケジューラ上で，多様なローカルスケジューリングアルゴリズムを実装できるように，ローカルスケジューラとグローバルスケジューラ間の共通インターフェースであるSPI（Scheduler Programming Interface）を定義する．ローカルスケジューラは，SPIを介してグローバルスケジューラと情報をやりとりする．既存のRTOSに対して提案フレームワークを実装することで実現可能性を

確認し，スケジューラの処理オーバヘッドを評価する．性能評価の結果，実用上，許容できる程度のオーバヘッドで実現できることを確認する．

本章の構成を述べる．まず 5.2 節で，これまで提案されたスケジューリングアルゴリズムを，満たすことのできる時間保護の要件の観点で分類し，フレームワークに実装するスケジューリングアルゴリズムを決定する．さらに，提案スケジューリングフレームワークにおいて，アプリケーションごとにアルゴリズムを選択する際の方針を述べる．5.3 節では，提案フレームワークの設計について詳細に述べる．5.4 節では，提案フレームワークのプロトタイプの実装について述べ，メモリ消費量とタスク切替処理時間などの観点で評価する．5.5 節で結論を述べる．

5.2　選択可能なスケジューリングアルゴリズム

5.2.1　時間保護要件によるスケジューリングアルゴリズムの分類

アプリケーションに要求される時間要件によって選択すべきプロセッサ時間の保護機能が異なる．加えてアプリケーションのパラメータの中で，統合段階で既知であるものも，それぞれ異なる場合が多い．例えば，周期タスクのみで構成されるアプリケーションでは，すべてのタスクの起動時刻は既知であるが，割込みなどのイベントにより起動するタスクを含むアプリケーションでは，起動時刻が既知ではないタスクも存在することになる．そのため，既知であるパラメータの異なるアプリケーションに対しては，同一のスケジューリングアルゴリズムを適用することは困難である．

本章では，時間保護の要件を満たすスケジューリングアルゴリズムを複数種類用意し，アプリケーションごとに，スケジューリングアルゴリズムを選択できるフレームワークを提案する．表 5.1 に，時間保護の要件ごとに，それを満たすことのできるスケジューリングアルゴリズムを整理する．

整理した結果，提案スケジューリングフレームワークで対象とするスケジューリングアルゴリズムは，アプリケーションを周期的に起動するアルゴリズムと，時間保護の要件をすべて満たすアルゴリズムの 2 種類とした．時間保護の要件 (1) と要件 (2) の 2 つを満たせるアルゴリズムを対象としない理由は，バジェットの管理手法が他のアルゴリズムに比べて複雑であるため，共通化するグローバルスケジューラの構造が複雑化してしまうためである．グローバルスケジューラの構造が複雑化すると，アプリケーション周期起動アルゴリズムや時間保護アルゴリズム 1 など

表 5.1: 満たす時間保護の要件とスケジューリングアルゴリズム

満たす要件	アルゴリズム	必要なパラメータ
(1)	単純な周期起動	アプリケーションの周期
(1),(2)	BSS, PShED	タスクの相対デッドライン
	時間保護アルゴリズム 2	タスクの相対デッドライン
(1),(2),(3)	Deng らのアルゴリズム	タスクの起動時刻と最悪実行時間
	時間保護アルゴリズム 1	タスクの起動時刻と相対デッドライン
	Lipari らの手法を用いた周期起動	タスクの周期と最悪実行時間

の，比較的単純なアルゴリズムの処理オーバヘッドが大きくなってしまう可能性がある．時間保護の要件 (1) と要件 (2) の 2 つを満たせるアルゴリズムを実装できるスケジューリングフレームワークの検討は，今後の課題とする．

5.2.2　スケジューリングアルゴリズムの選択方針

ここでは，アプリケーションごとに，どのアルゴリズムを選択すべきかの方針を述べる．

アプリケーション周期起動アルゴリズムは，アプリケーションごとに，シェアに加えて，アプリケーションを実行する周期を設定し，その周期ごとにシェアに応じた一定量のバジェットの割り当てを要求するアルゴリズムである．アプリケーションのタスクを実行した結果，アプリケーションに割当てられたプロセッサ時間が 0 になると，次の周期でプロセッサ時間が補充されるまで，そのアプリケーション内のタスクは実行されることはない．したがって，このアルゴリズムを適用することにより，時間保護の要件 (1) を満たすことができる．

タスクの起動時刻と正確な最悪実行時間が得られる場合は，Lipari らの設計手法 [25] を用いて，設定するシェアと起動周期を決定し，アプリケーション周期起動アルゴリズムを選択すれば，すべての要件を満たすことができる．しかし，ほとんどの場合，正味のプロセッサ利用率（各タスクが必要とする最低限のプロセッサ利用率を合計した値）を越えるシェアをアプリケーションに設定する必要がある [36]．

周期タスクのみで構成されるアプリケーションに対しては，周期タスクの起動周期の最大公約数をアプリケーションの実行周期として設定することで，アプリケーションの周期起動アルゴリズムでも時間保護の要件 (2) を満たせる場合がある．しかし，一般には，周期タスクの起動周期の関係がハーモニックでない場合が多く，アプリケーションの起動周期を非常に短くしなければならない．その結果，実行時のアプリケーションの切換え回数が増加することで，RTOS の処理オーバヘッ

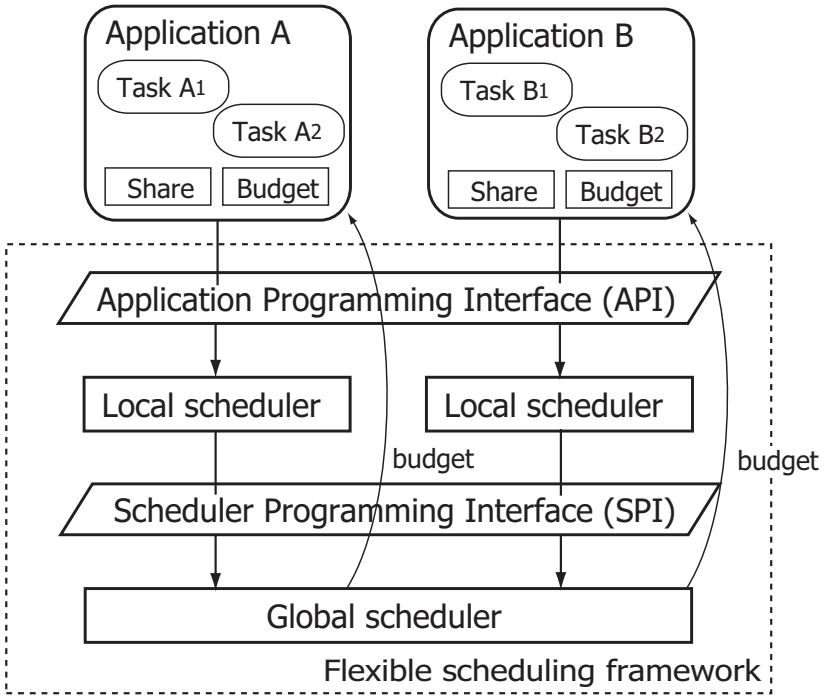

図 5.1: スケジューリングフレームワークの構成

ドが問題となる．以上の議論より，アプリケーションの周期起動アルゴリズムは，要件 (2) と要件 (3) を満たすためのアルゴリズムとしては適さない．

時間保護の要件 (1) 以外も満たす必要がある場合に，タスクの起動時刻と最悪実行時間が既知である場合には，Deng らのアルゴリズム [10, 11] アルゴリズムを適用できる．タスクの実行時間の代わりに，相対デッドラインが既知である場合は，第 3 章で述べた時間保護アルゴリズム 1[46] を適用できる．これらのアルゴリズムの違いは，Deng らのアルゴリズムがタスクの最悪実行時間を用いるのに対して，時間保護アルゴリズム 1 は相対デッドラインを用いる点であるが，スケジューリングの動作に本質的な違いはない．

5.3　スケジューリングフレームワーク

5.3.1　スケジューリングフレームワークの構成

提案するスケジューリングフレームワークの概要について述べる．提案フレームワークでは，図 5.1 に示すように，ローカルスケジューラとグローバルスケジューラを 2 階層に配置した階層型スケジューラを採用する．

ローカルスケジューラは，アプリケーションごとに割当てられ，所属するタスクをスケジューリングするタスクスケジューリング機能と，グローバルスケジューラに対してバジェットを要求する機能を持つ．本章では，タスクの実行順序を決定するためのスケジューリングアルゴリズムをタスクスケジューリングアルゴリズム，アプリケーションがバジェットを要求するためのスケジューリングアルゴリズムをバジェット要求アルゴリズムと呼ぶ．

ローカルスケジューラのバジェット要求アルゴリズムは，アプリケーションにバジェットを補充してほしい時刻を決定し，それをグローバルスケジューラに通知する．バジェットの補充を要求する時刻を,「アプリケーションを個別プロセッサと統合プロセッサで同時に実行を開始した場合に，個別プロセッサで得る処理量と統合プロセッサで得る処理量が一致する時刻」という意味で平衡時刻と呼ぶ．

グローバルスケジューラは，どのアプリケーションのタスクをプロセッサで実行するかを決定するアプリケーションスケジューリング機能と，アプリケーションに補充するバジェット量を決定するバジェット補充機能を持つ．本章では，アプリケーションの実行順序を決定するためのスケジューリングアルゴリズムをアプリケーションスケジューリングアルゴリズムと呼び，アプリケーションに補充するバジェット量を計算して補充するアルゴリズムをバジェット補充アルゴリズムと呼ぶ．

ローカルスケジューラとグローバルスケジューラ間のインタフェースが SPI である．ローカルスケジューラは，グローバルスケジューラに対して，SPI を介して情報の登録やタスク切換えなどの処理を要求する．提案フレームワークでは，SPI を，ローカルスケジューラのバジェット要求アルゴリズムに依存しないよう定義することで，特定のスケジューリングアルゴリズムに依存せず，多様なアプリケーションに対応できる．

5.3.2　タスクモデル

タスクは，提案フレームワークにおける最小の実行単位であり，いずれかのアプリケーションに所属する．タスクの状態は，*suspended*，*ready*，*running* の 3 状態ある．タスクがローカルスケジューラのタスクスケジューリングアルゴリズムによりスケジュールされると，図 5.2 に示す状態遷移図に従って，その状態が遷移する．初期状態は *suspended* で，システムサービスにより実行可能になると *ready* に遷移する．アプリケーション内の *ready* タスクの中で最も優先度の高いタスクが *running* に遷移する．同一優先度の *ready* タスクが複数存在する場合には，最も早

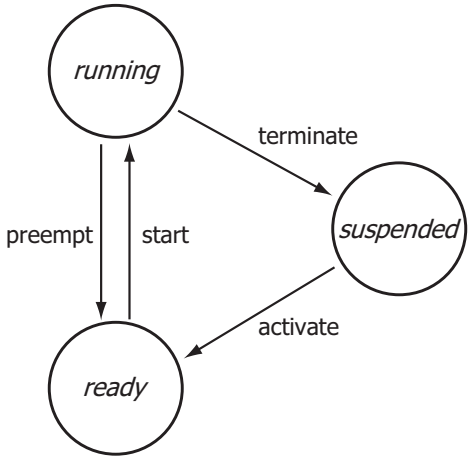

図 5.2: タスクの状態遷移図

く *ready* になったタスクが *running* となる．*running* タスクは，実行が終了すると *suspended* に遷移し，*ready* タスクの中で最も優先度の高いタスクが *running* に遷移する．タスクスケジューリングアルゴリズムは，プリエンプティブであることを前提としており，*running* タスクの実行中に，より優先度の高いタスクが *ready* になると，*running* タスクは *ready* に遷移し，優先度の高いタスクが *running* に遷移する．

5.3.3　アプリケーションモデル

アプリケーションは 1 つ以上のタスクで構成される，タスクの集合である．アプリケーションの状態は，表 5.2 に示すように，*suspended*，$replenishment - waiting$，*ready*，*running*，*exhausted* の 5 状態がある．アプリケーションの状態は，アプリケーションに所属するタスクの状態遷移とバジェット消費に伴い，図 5.3 の状態遷移図に従って遷移する．アプリケーションの初期状態は *suspended* である．アプリケーションに *ready* タスクが存在すると，$replenishment - waiting$ に遷移してバジェットの補充を待つ．グローバルスケジューラが，$replenishment - waiting$ アプリケーションにバジェットを補充すると，そのアプリケーションは *ready* に遷移する．*ready* アプリケーションの中で，平衡時刻の最も早いアプリケーションは *running* に遷移する．グローバルスケジューラは，*running* アプリケーションの *running* タスクを実行する．*running* アプリケーションのすべての *ready* タスクの実行が完了するか，バジェットが 0 になると，そのアプリケーションは *exhausted* に遷移し，

表 5.2: アプリケーションの状態

状態名	説明
suspended	バジェットが 0 で，実行できるタスク（*ready* タスクもしくは *running* タスク）が存在しない状態.
replenishment − *waiting*	バジェットが 0 で，実行できるタスクが存在する状態．平衡時刻が決定すれば，バジェットが補充されて *ready* に遷移する.
ready	バジェットが 0 ではなく，実行できるタスクが存在する状態. ただし，*running* タスクは，実際にプロセッサで実行されてはいない.
running	*running* タスクがプロセッサで実行されている状態.
exhausted	バジェットが 0 になり，平衡時刻が経過するのを待っている状態．所属するすべての *ready* タスクの実行が完了するか，バジェットが 0 になると，この状態に遷移する.

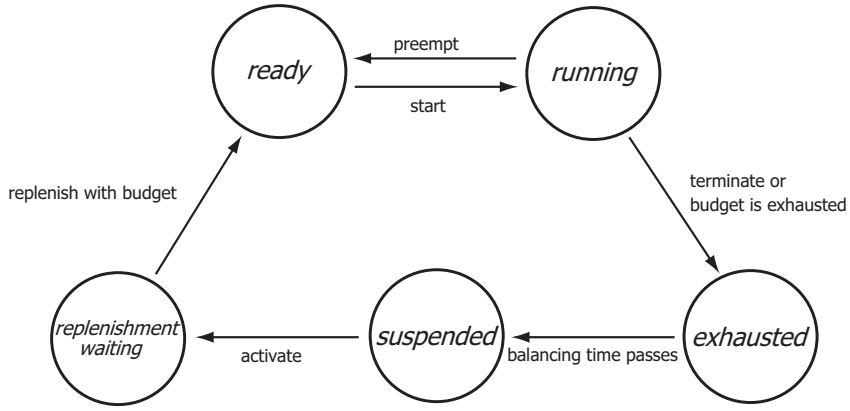

図 5.3: アプリケーションの状態遷移図

ready アプリケーションの中で次に平衡時刻の早いアプリケーションが *running* に遷移する．システム時刻が *exhausted* アプリケーションの平衡時刻を経過すると，そのアプリケーションは *dormant* に遷移する．アプリケーションが *dormant* に遷移した際に，*ready* タスクが存在する場合は，即座に *replenishment* − *waiting* に遷移して，バジェットの補充を待つ.

5.3.4　ローカルスケジューラ

ローカルスケジューラはアプリケーションと 1 対 1 に対応し，アプリケーションに所属するタスクをスケジューリングするタスクスケジューリング機能と，平衡

表 5.3: バジェット要求アルゴリズムと平衡時刻

バジェット要求アルゴリズム	平衡時刻
周期要求	アプリケーションの次の起動時刻
非周期要求	タスクの起動時刻とデッドラインの中で最も早い時刻

時刻を決定しグローバルスケジューラに通知するバジェット要求機能を持つ．管理するパラメータは，バジェット，シェア，平衡時刻の 3 つである．

タスクスケジューリングアルゴリズムは，アプリケーションに所属するタスクをスケジュールし，SPI を介して *running* タスクの ID をグローバルスケジューラに通知する．タスクスケジューリングのアルゴリズムは，統合前と同一とする．

システムの設計者は，アプリケーションに対する時間要件と，統合段階で既知であるバラメータから，適用するバジェット要求アルゴリズムを選択する．例えば，5.2.2 項におけるスケジューリングアルゴリズムの選択として，アプリケーションの周期起動アルゴリズムもしくは，Lipari らの設計手法 [25] に基づく周期起動アルゴリズムを選んだ場合には，バジェット要求アルゴリズムは周期要求アルゴリズムを選択すれば良い．バジェット周期要求アルゴリズムは，アプリケーションの起動周期に従い，アプリケーションの次の起動時刻を平衡時刻としてグローバルスケジューラに通知する．

一方，時間保護アルゴリズム 1 を選択した場合，バジェット要求アルゴリズムは非周期要求アルゴリズムを選択すれば良い．非周期要求アルゴリズムは，アプリケーションに所属するタスクの起動時刻とデッドラインを管理し，その時刻の中で最も早い時刻を，平衡時刻としてグローバルスケジューラに通知する．5.2.2 項で Deng らのアルゴリズムを選択した場合には，非周期要求アルゴリズムを若干修正する必要がある．具体的には，タスクの絶対デッドラインの代わりに，タスクの最悪時の終了時刻を管理し，その時刻の中で最も早い時刻を平衡時刻としてグローバルスケジューラに通知するよう変更する．このように修正したバジェット要求アルゴリズムは，時間保護アルゴリズム 1 を選択した場合の非周期要求アルゴリズムと本質的には変わらない．つまり，時間保護アルゴリズム 1 用の非周期要求アルゴリズムを実装できれば，Deng らのアルゴリズムに対応するよう修正することは容易である．そのため，本論文では，時間保護アルゴリズム 1 用の非周期要求アルゴリズムのみを実装することにする．

表 5.4: スケジューラプログラミングインタフェース（SPI）

関数インタフェース	機能
`dispatch()`	*running* タスクを中断し，タスク切替えを要求する
`exit_and_dispatch()`	*running* タスクを終了し，タスク切替えを要求する
`update_balancing_time(aid, time)`	アプリケーション aid の平衡時刻を time に更新する
`update_schedtsk(aid, tid)`	アプリケーション aid の実行するべきタスクの ID を tid に更新する

5.3.5　グローバルスケジューラ

グローバルスケジューラは，どのアプリケーションの *running* タスクを実行するかを決定するアプリケーションスケジューリング機能と，アプリケーションに対してバジェットを計算して補充するバジェット補充機能を持つ．アプリケーションスケジューリングアルゴリズムは，*ready* アプリケーションの中で，平衡時刻の最も早いアプリケーションの *running* タスクを実行する．バジェット補充アルゴリズムは，$replenishment - waiting$ アプリケーションの中で，平衡時刻が登録されたアプリケーションに対してバジェットを補充する．補充するバジェットの量は，その時点のシステム時刻から平衡時刻までの時間と，シェアの積である．このように，グローバルスケジューラは，アプリケーションの状態，平衡時刻，*running* タスクのみ管理することで，ローカルスケジューラのタスクスケジューリングアルゴリズムやバジェット要求アルゴリズムには依存せずに動作する．

5.3.6　スケジューラ間インタフェース

ローカルスケジューラとグローバルスケジューラのインタフェースである SPI のうち，スケジューリングに関連する SPI の一覧を表 5.4 に示す．ローカルスケジューラは，グローバルスケジューラに対して情報を通知したり，処理を要求する場合に，SPI を呼び出す．

update_schedtsk 関数は，ローカルスケジューラのタスクスケジューリングの結果，アプリケーション内で実行するべきタスク（*running* タスク）が変化した場合

に，新しく *running* なったタスクの ID を通知するために使用する．

dispatch 関数は，update_schedtsk 関数の後に呼び出される関数で，実行中の *running* タスクの処理を一時中断し，新しい *running* タスクに実行を切り替えるために使用する．exit_and_dispatch 関数は，*running* タスクの実行が終了したときに呼び出される関数で，次に優先度の高いタスクに実行を切り替えるために使用する．

update_balancing_time 関数は，グローバルスケジューラに登録されている，アプリケーションの平衡時刻を更新するために使用する．ローカルスケジューラが呼び出すタイミングは，バジェット要求アルゴリズム毎に異なる．例えば，非周期要求アルゴリズムの場合には，タスクの起動や終了するタイミングで呼び出す．一方，周期要求アルゴリズムでは，アプリケーションの起動時刻に呼び出す．

5.3.7　動作例

異なるバジェット要求アルゴリズムを適用した場合の，アプリケーションの動作の違いを具体的な例を用いて説明する．ここでは，アプリケーション 1 とアプリケーション 2 の 2 つのアプリケーションを統合するものとし，アプリケーション 1 の動作に着目する．アプリケーション 1 は，タスク 1 とタスク 2 の 2 つのタスクで構成され，タスク 2 はタスク 1 より高い優先度を持つ．各タスクの相対デッドラインは，タスク 1 が 10，タスク 2 が 14 とする．ローカルスケジューラのシェアは 50%が設定され，固定優先度ベースのプリエンプティブスケジューリングアルゴリズムでタスクをスケジュールする．

まず，バジェット要求アルゴリズムとして，非周期要求アルゴリズムを適用した場合の動作例を，図 5.4 に示す．非周期要求アルゴリズムの場合は，タスク起動により平衡時刻が変化するタイミング（時刻 0，時刻 15）や，タスクの実行終了により平衡時刻が変化するタイミング（時刻 5）で，update_balancing_time を呼び出し，アプリケーションの平衡時刻の変化を通知する．時刻 21 では，タスク 2 の実行が終了して，タスク 1 に実行が切り替わるが，平衡時刻は変化しない（タスク 2 のデッドラインである時刻 29 のまま）ため，update_balancing_time を呼び出す必要はない．タスクスケジューリングにより *running* タスクが変化するタイミング（時刻 0，時刻 5，時刻 15，時刻 21）では，update_schedtsk を呼び出し，グローバルスケジューラに *running* タスクの ID を通知することに加え，dispatch もしくは，exit_and_dispatch を呼び出して，実行するタスクの切替えを要求する．アプ

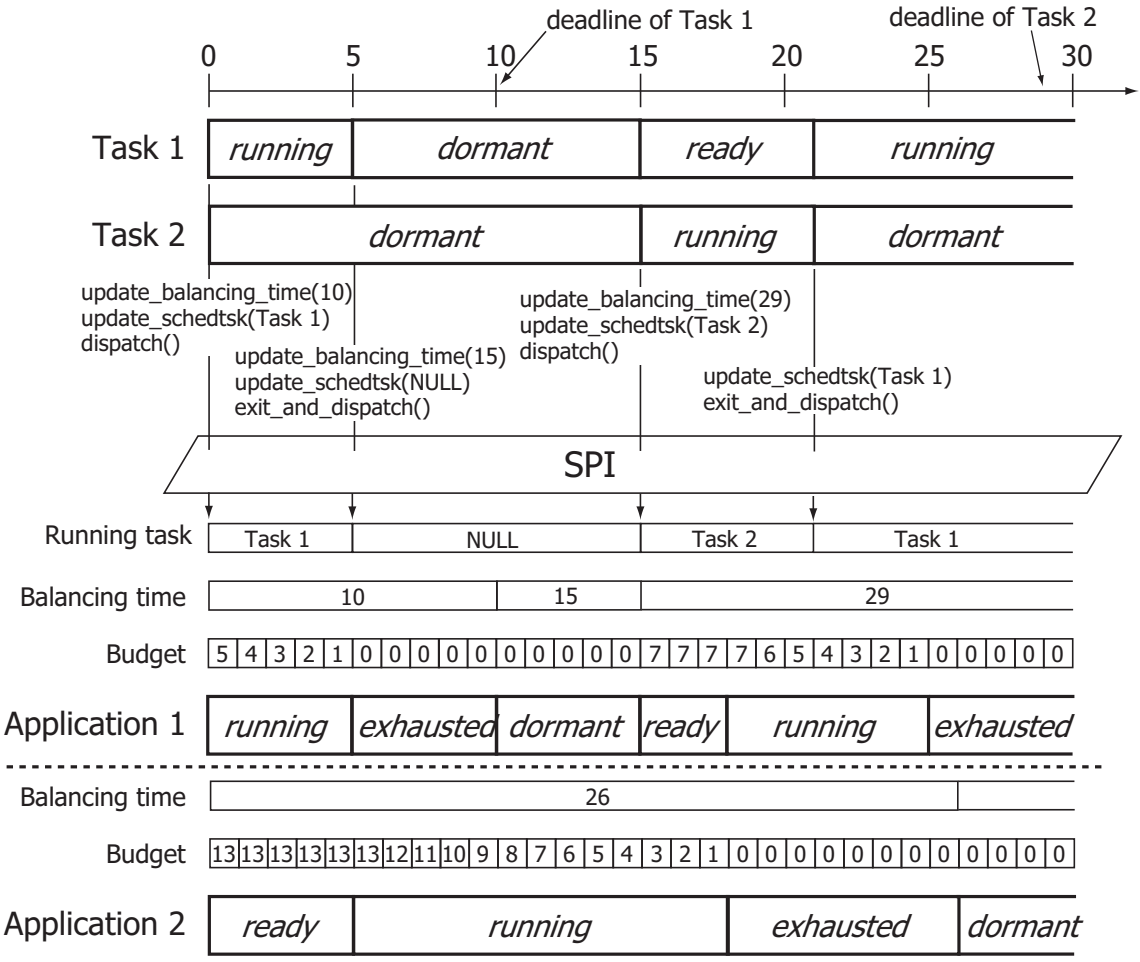

図 5.4: 非周期要求アルゴリズムを適用した場合のアプリケーションの動作例

リケーションが *ready* であっても，より平衡時刻の早いアプリケーションが存在する場合には，*running* に遷移しない．例えば，時刻 15 でアプリケーション 1 は *ready* であるが，平衡時刻が時刻 29 よりも早いアプリケーション 2 が *running* であるため，アプリケーション 2 がバジェットを使い切る時刻 18 までは *ready* のままである．

次に，バジェット要求アルゴリズムとして，周期要求アルゴリズムを適用した場合の動作例を，図 5.5 に示す．アプリケーション 1 のバジェットの補充周期は 10 とする．この場合，通常であれば，ローカルスケジューラは定期的にバジェットを要求するために，周期 10 で update_balancing_time を呼び出す必要がある．提案フレームワークでは，この処理をグローバルスケジューラ内で自動的に実行するよう実装することで，周期要求アルゴリズムのローカルスケジューラが update_balancing_time を呼び出す処理を省略できる．タスクスケジューリングにより *running* タスクが

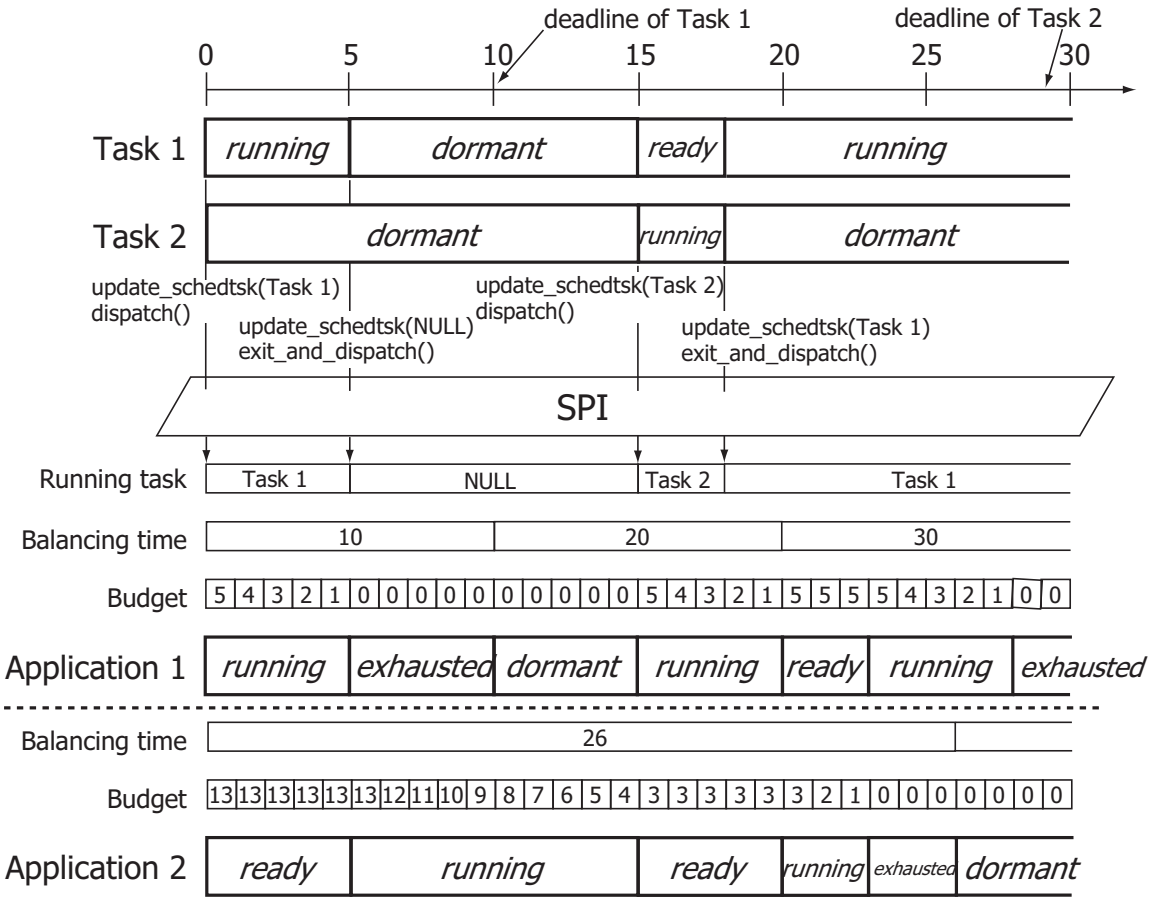

図 5.5: 周期要求アルゴリズムを適用した場合のアプリケーションの動作例

変化する時刻（時刻 0，時刻 5，時刻 15，時刻 18）では，update_schedtsk を呼び出して，*running* タスクの ID をグローバルスケジューラに通知し，dispatch もしくは，exit_and_dispatch を呼び出す．

5.4　実装と評価

5.4.1　実装

提案フレームワークのプロトタイプを，OSEK OS 仕様準拠の TOPPERS/ATK1（Automotive Kernel Version 1）[41] をベースに実装した．タスクスケジューリングアルゴリズムは，固定優先度ベースのプリエンプティブスケジューリングである．なお，提案フレームワークが OSEK OS 仕様に依存する部分はなく，同一のタスクモデルを導入できる仕様（例えば，μITRON 仕様）であれば，他の RTOS に実装することも容易である．

ターゲットボードは，Renesas 社製 M32R-II ソフトコアプロセッサ（動作周波数：10MHz，キャッシュ：OFF）を搭載した M3A-ZA36 ボードである．メモリは，32K バイトのコア内蔵 SRAM と，64M バイトの外部 SDRAM（動作周波数：10MHz）を持つ．このプロセッサの性能は，現在ハイエンドな組込み向け機器に用いられるものに比較すると劣る．特に，動作周波数については，実際の自動車制御システムの制御系で用いられるマイコンの動作周波数の 1/4 から 1/10 程度である．そのため，性能評価の段階ではこの点を考慮する必要がある．

プロトタイプを実装するに当たり，2 つのハードウェアタイマを使用した．それぞれ，アプリケーションのバジェットを管理するタイマと，システム時刻を管理するタイマである．バジェット管理用タイマの割込み処理では，バジェットが 0 になった後に，カーネルの動作が一貫性を保てる中で出来る限り早く，実行するアプリケーションを切り替える必要がある．今回は，システム時刻を管理する割込み処理の優先度の次に高い優先度を設定した．このように，プロトタイプを実装するためには，タイマ毎に割込み優先度を設定できる割込みコントローラが必要となる．

5.4.2　評価

実装したプロトタイプについて，実行形式サイズ，タスク切替え時間，アプリケーションのスケジューリング時間の観点で評価する．

実行形式サイズ

まず，プロトタイプのメモリ消費量を評価する．ここでは，実行形式ファイルにアプリケーションの領域は含めず，カーネル本体のみを測定対象とする．TOPPERS/ATK1 とプロトタイプの実行形式ファイルを表 5.5 に示す．プロトタイプの評価値の括弧内は，TOPPERS/ATK1 の値に対する増加率を示す．

プロトタイプの ROM と RAM の使用量は，TOPPERS/ATK1 に比較して，ROM が 6.4KB，RAM が 0.7KB 増加した．主な要因は，スケジューラを階層化することによるコードの増加や，バジェット管理用タイマを操作するコードなど追加したことである．測定結果より，ROM と RAM 使用量の増加量は非常に少なく，実用上許容できる範囲である．

表 5.5: カーネルの実行形式 (ELF) ファイルサイズ

カーネル	ROM(KB)	RAM(KB)
TOPPERS/ATK1	11.6	1.6
プロトタイプ	18.0 (+55.1%)	2.3 (+43.8%)

タスク切替え時間

次に，提案フレームワークの主な処理であるスケジューリング，ディスパッチ，タスク切替え処理時間を評価する．

評価に用いたアプリケーションは，リアルタイム性を要求される周期タスクのみで構成されるリアルタイムアプリケーションと，リアルタイム性を要求されない周期タスクとバックグラウンドタスクで構成される非リアルタイムアプリケーションの 2 つである．各アプリケーションのタスク数は，それぞれ 2 個（シェアはそれぞれ 40%），4 個（シェアはそれぞれ 20%），8 個（シェアはそれぞれ 10%）と変化させる．

リアルタイムアプリケーションは，バジェット要求アルゴリズムとして非周期要求アルゴリズムを選択し，時間保護アルゴリズム 1 でスケジュールする．一方，非リアルタイムアプリケーションは，バジェット要求アルゴリズムとして周期要求アルゴリズムを選択し，周期実行アルゴリズムでスケジュールする．

TOPPERS/ATK1 の固定優先度スケジューリングと，プロトタイプに実装した周期実行アルゴリズム及び時間保護アルゴリズム 1 について，同一アプリケーション内のタスク切替え処理時間（OSEK OS 仕様で規定されているサービスコールの ActivateTask の実行時間）を表 5.6 に示す．括弧内は，TOPPERS/ATK1 の固定優先度スケジューリングの値に対する増加率を示す．スケジューリング時間とは，*running* タスクより優先度の高いタスクを起動するサービスコールを呼出してから，タスクコンテキストの退避処理を開始するまでの時間である．ディスパッチ時間とは，タスクのコンテキスト退避処理を開始してから，優先度の高いタスクの処理が開始されるまでの時間である．

測定結果によると，同一アプリケーション内のタスク切替え処理においては，周期実行アルゴリズムの場合は，スケジューリング時間が 41μs，ディスパッチ時間が 17μs，タスク切替え全体の時間は 57μs それぞれ増加した．一方，時間保護アルゴリズム 1 の場合は，スケジューリング時間が 141μs，ディスパッチ時間は 17μs，タスク切替え全体の時間は 161μs それぞれ増加した．同一アプリケーションのタ

表 5.6: 同一アプリケーションのタスクへの切替え処理時間

スケジューリング アルゴリズム	スケジューリング 時間 (μs)	ディスパッチ 時間 (μs)	タスク切替え 時間 (μs)
固定優先度スケジューリング	49	25	77
周期実行アルゴリズム	90 (+83.7%)	42 (+68.0%)	134 (+74.0%)
時間保護アルゴリズム 1	190 (+287.8%)	42 (+68.0%)	234 (+239.0%)

表 5.7: 時間保護アルゴリズム 1 における別アプリケーションのタスクへの切替え処理時間

スケジューリング アルゴリズム	スケジューリング 時間 (μs)	ディスパッチ 時間 (μs)	タスク切替え 時間 (μs)
時間保護アルゴリズム 1	248	42	292

スクへの切替えでは，タスクスケジューリングを O(1) で処理できるため，これらの測定結果は，アプリケーションのタスク数により変化しない．

次に，時間保護アルゴリズム 1 において，異なるアプリケーション間のタスク切替え処理時間を 2 つの状況で評価する．1 つ目の状況は，実行中のアプリケーション内で実行するタスクを切り替えた結果，平衡時刻が更新され，その更新された平衡時刻が別のアプリケーションの平衡時刻より遅くなったために，実行するアプリケーションが切り替わる場合である．評価結果を表 5.7 に示す．スケジューリング時間が，同一アプリケーション内の場合に対して，さらに 58μs 増加した．2 つ目の状況は，実行中のアプリケーションがバジェットを使い切り，別のアプリケーションに切替える場合である．この処理時間は，130μs 程度であった．

測定した処理時間の増加量については，評価ボードのプロセッサ性能が，現実の自動車制御 ECU に搭載されるプロセッサ性能に対して 1/10 から 1/2 程度であることを考慮に入れると，多くの場合，実用上許容できる範囲であると考えらえる．ただし，高い応答性を要求されるアプリケーションでは，この処理オーバヘッドが許容できない可能性があるため，特に，時間保護アルゴリズム 1 については，さらにオーバヘッドを小さくするよう実装方法を改善する必要がある．

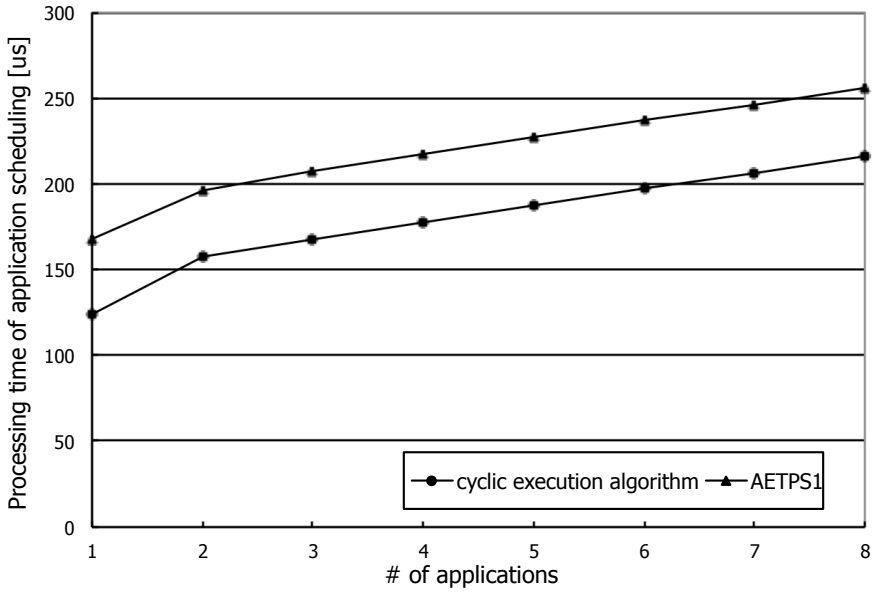

図 5.6: アプリケーションスケジューリング時間

アプリケーションスケジューリング時間

次に，統合するアプリケーションの数に依存して変動するオーバヘッドとして，グローバルスケジューラにおけるアプリケーションのスケジューリング時間を評価する．1 つのアプリケーションに属するタスク数を 4 つに固定し，アプリケーションの数を変化させたときの最長スケジューリング処理時間を測定する．

グローバルスケジューラにおいて，実行可能なアプリケーションは，平衡時刻の早い順に一方方向のキューに接続されている．したがって，最長スケジューリング処理時間とは，アプリケーションの平衡時刻が実行可能なアプリケーションの中で最も遅い状況において，測定対象のアプリケーションが実行可能になった時刻から，キューの最後尾に接続されるまでの処理時間となる．図 5.6 に，周期実行アルゴリズム（図中では cyclic execution algorithm と表記）と時間保護アルゴリズム 1（図中では AETPS1 と表記）のそれぞれのスケジューリング時間を示す．

提案フレームワークでは，実行可能なアプリケーションを管理するキューに対して，新たに実行可能になったアプリケーションの挿入位置を線形探索する．そのため，適用するスケジューリングアルゴリズムによらず，スケジューリングの最長処理時間は，統合するアプリケーション数に比例して増加する．測定結果から，統合するアプリケーション数が 1 つ増加すると，約 10μs 程度のオーバヘッドで済む．この増加量はタスクのスケジューリング処理時間に比べて小さいため，統合

するアプリケーション数が増加しても，実用的な範囲のオーバヘッドで実装可能であることが明らかになった．

5.5　まとめ

本章では，分散システムにおいて，各システムで単独で動作するアプリケーションを 1 つのプロセッサ上に統合することを目的としたスケジューリングフレームワークを提案した．提案フレームワークでは，アプリケーション毎に適切なバジェット要求アルゴリズムを選択することで，アプリケーションに要求される時間要件や，統合段階で既知であるパラメータが異なるアプリケーションに，柔軟に対応できる．そのため，従来の統合スケジューリングアルゴリズムでは，容易に統合できなかった分散システムに適用できると考える．提案フレームワークを OSEK OS 仕様準拠の RTOS をベースに実装し，メモリ消費量，タスク切替え処理時間，アプリケーションスケジューリング時間の観点から評価した結果，実用上許容できる程度のオーバヘッドで実現できることを確認した．よって，提案フレームワークは小型のマイコンに対しても適用できる．

第6章　割込み処理を含むアプリケーションへの対応

6.1　概要

割込み処理は，タスクより高い優先度が割り当てられ，かつその処理時間も，タスクの処理時間に比べると短い場合が多く，タスクのスケジューリングにのみ着目する場合には，処理時間を無視できるという前提を置く場合が多い．これまで提案された階層型スケジューリングアルゴリズムでも，タスクのみで構成されるアプリケーションを対象としており，割込み処理のスケジューリングについてはほとんど言及されていない．

実際のリアルタイムアプリケーションは，タスクと割込み処理で構成されることが多い．さらに，多様な割込み要因をもつ場合や，割込み発生回数が多い場合があり，割込み処理を含むアプリケーションを統合する状況において，割込み処理の処理時間をすべて無視することは妥当な前提とは言えない．例えば，実際の自動車制御システムには，一回の割込みハンドラの処理時間は短いが，割込み回数が非常に多いアプリケーションが存在する．このような場合には，プロセッサ利用率の設定においては，OS オーバヘッドと割込みハンドラの処理時間を軽視できない．

統合の段階で，タスクの処理だけでプロセッサ時間のほとんどを占有するようシェアを割り当ててしまうと，割込み処理がプロセッサを占有した時間を無視した結果，割り当てられたバジェットを使い切ることができないアプリケーションがでてくる可能性がある．この場合，アプリケーション間のプロセッサ時間の保護が破綻してしまう．

本章では，タスクと割込み処理で構成されるリアルタイムアプリケーションも統合できるようにするため，第 5 章で述べた階層型スケジューリングフレームワークに対して容易に追加可能な割込み処理スケジューリングアルゴリズムを提案する．提案アルゴリズムをスケジューリングフレームワークのプロトタイプに実装して，割込み応答性を性能を評価する．

本章の構成を述べる．6.2 節で，対象とする階層型スケジューリングについて簡単に述べる．6.3 節では，割込み処理のスケジューリングについて述べる．6.4 節で，既存の階層型スケジューラへ実装するための設計を述べ，性能を評価する．6.5 節では，割込み禁止時間をもつアプリケーションの統合可否判定式について述べる．6.6 節で，提案するスケジューリングアルゴリズムの有効性について議論する．最後に，6.7 節で，結論を述べる．

6.2　対象とする階層型スケジューリング

本章では，第 5 章のスケジューリングフレームワークに適用可能な階層型スケジューリングを対象とする．各アプリケーションは，外部イベントにより起動する割込み処理と，タスクで構成される．アプリケーションに属するすべての処理は，個別プロセッサで固定優先度のプリエンプティブスケジューリングによりスケジュール可能であることを前提とする．

スケジューリングフレームワークは，それぞれのアプリケーションに対応するローカルスケジューラと，システムで 1 つあるグローバルスケジューラで構成され，スケジューラ間のインタフェースである SPI を規定している．フレームワーク上で動作するアプリケーションの振る舞いは，ローカルスケジューラがグローバルスケジューラに対して SPI を介して通知する平衡時刻を決定するアルゴリズムに依存する．例えば，一定の時間間隔で一定量のバジェットが割り当てられることを期待する場合には，周期要求アルゴリズムにより，グローバルスケジューラに対して，一定周期で次の周期の開始時刻を平衡時刻として通知する．

6.3　割込み処理のスケジューリングアルゴリズム

6.3.1　アルゴリズムの概要

割込み処理は，割込みを発生する外部イベントに対応する処理であり，その割込み要因の重要性に応じて割込み処理の優先度が決定される．本章では，割込み処理のパラメータとして，優先度と，割込み発生時刻からの相対デッドラインが与えられているものとする．これらのパラメータは，実行中に変更されないものとする．割込み処理の優先度は，同じアプリケーションに属するすべてのタスクの優先度よりも高いものとするが，この前提は，アプリケーション間では成立す

る必要はない．割込み処理のスケジューリングとして，グローバル固定優先度スケジューリングとローカル固定優先度スケジューリングの 2 方式を提案する．

6.3.2　グローバル固定優先度スケジューリング

グローバル固定優先度スケジューリングでは，割込みが発生した際に，その割込み要因に対応する割込み処理を起動するかを判断するために，割込み処理の優先度をシステム全体を対象に比較する．すなわち，割込み処理は，優先度が高ければ，別のアプリケーションに属する処理が実行中であっても，実行中のその処理に優先して実行される．逆に，優先度が同じか低い場合には，割込み要求は受け付けずに，割込み処理は実行されない．システム全体のスケジューリングを整理すると，割込み処理は固定優先度スケジューリング，タスクは階層型スケジューリングという組合せになる．

グローバル固定優先度スケジューリングの特徴としては，一般的な RTOS やハードウェアアーキテクチャ（プロセッサ，割込みコントローラ）に対して親和性が高く，実装が比較的容易であるという利点がある．その一方で，割込み処理の優先度は，システム全体に対して影響をもつことになるため，アプリケーションの独立性が低下する．具体的には，アプリケーションの設計においては，統合するすべてのアプリケーションとの優先度関係を考慮する必要があり，単独で開発，動作検証したアプリケーションを，その詳細な設計を把握せずに統合することは困難となる．

6.3.3　ローカル固定優先度スケジューリング

ローカル固定優先度スケジューリングは，割込み処理のスケジューリングにおける優先度比較の対象を，属するアプリケーション内に限定する．すなわち，割込みが発生した際に，その割込み要因に対応する処理の優先度が，属するアプリケーション内でもっとも高い場合にのみ割込み処理を起動する．それ以外の場合には，割込み要求は受け付けず，割込み処理も起動しない．これを実現するために，割込み処理を，割込みハンドラと割込みサービスタスクの 2 つに分割したモデルを導入する．割込みハンドラは，デバイスに依存する処理で，非常に短い時間内に最小限の処理のみを実行するものとする．割込みサービスタスクは，デバイスに依存せず，割込みに対応して実行したい処理である．割込みハンドラでは，

最小限の処理を実行した後に，割込みサービスタスクを起動する．割込みサービスタスクの優先度は，割込み処理の優先度と同じとする．

このモデルを導入することで，割込み処理の処理時間の大部分を占める部分を割込みサービスタスクに分離することができる．階層型スケジューリングでは，割込みサービスタスクを通常のタスクと同様に扱うことができるため，アプリケーションの独立性を維持できる．したがって，ローカル固定優先度スケジューリングにより割込み処理をスケジュールするという考え方は，アプリケーション統合に適する．しかしながら，アプリケーションのデッドラインが短いアプリケーションが存在すると，選択したグローバルスケジューリングアルゴリズムによっては，別のアプリケーションのタスクが，割込みサービスタスクに優先して実行される場合があるため，割込みサービスタスクの応答性が低下する（すなわち，割込み応答性が低下してしまう）．

6.4　実装と評価

6.4.1　実装環境

割込み処理のスケジュール方式の実装容易性と，割込み処理の応答性能を評価するため，第 5 章で述べたスケジューリングフレームワークのプロトタイプにそれぞれの方式を実装する．実装ターゲットボードは，オークス電子社製の OAKS32R ボードである．このボードには，M32R プロセッサ（動作周波数は 66MHz，キャッシュは OFF）を搭載し，メモリは，64K バイトのコア内蔵 SRAM と，8M バイトの外部 SDRAM を持つ．今回は，すべて外部 SDRAM 上に配置する．

実装のベースとするプロトタイプは，オープンソースの TOPPERS/ASP カーネル [42] にスケジューリングフレームワークを実装したプロトタイプシステムである．このプロトタイプシステムの基本性能を表 6.1 に示す．

6.4.2　割込み処理の流れ

次に，割込み処理の流れについて述べる．割込み処理は，プロセッサや割込みコントローラ（IRC）のアーキテクチャに依存するため，本章では，TOPPERS/ASP カーネルが準拠している標準割込み処理モデル [40] を想定とする．これを，図 6.1 に示す．この図では，すべてをハードウェアで実現されていると想定して描かれて

いる．周辺デバイスからの割込み要求が発生すると，割込みコントローラ（IRC）を経由して，プロセッサに伝えられる．プロセッサは，NMI 以外の割込み要求は，次の 4 つの条件を満たす場合に割込み要求を受け付ける．

1. 割込み要求禁止フラグがクリアされていること
2. 割込み要求の持つ割込み優先度が，現在の割込み優先度マスクの現在値よりも高いこと
3. 全割込みロックフラグがクリアされていること
4. CPU ロックフラグがクリアされていること

各フラグはハードウェアで実現できるプロセッサもあれば，ソフトウェアにより実現しなければならないプロセッサもある．M32R では，CPU ロックフラグ（カーネル管理の割込みの受付を禁止するフラグ）のみソフトウェアにより実現する．割込み要求が受け付けられた後の処理は，割込み処理のスケジューリングにより異なるが，基本的には次の処理が実行される．詳細は，次節にて述べる．

1. 割込み入口処理
2. 割込みハンドラの実行
3. 割込みサービスルーチンの実行（グローバル固定優先度方式の場合）
4. 割込みサービスタスクの起動（ローカル固定優先度方式の場合）
5. 割込み出口処理

表 6.1: プロトタイプシステムの基本性能

処理内容	命令名称	実行時間（us）
タスク起動（タスク切り替え有り）	act_tsk	62
タスク起動（タスク切り替えなし）	act_tsk	5
タスク起動（非タスクコンテキスト）	iact_tsk	4
システム時刻更新とバジェット管理	signal_time	10
バジェット計測開始	budget_timer_start	6
バジェット計測停止	budget_timer_stop	7
アプリケーションの最高優先順位タスク ID の通知（SPI）	set_schedtsk	5
アプリケーションの平衡時刻の通知（SPI）	set_balancing_time	10
アプリケーションの絶対デッドラインの通知（SPI）	set_deadline	1

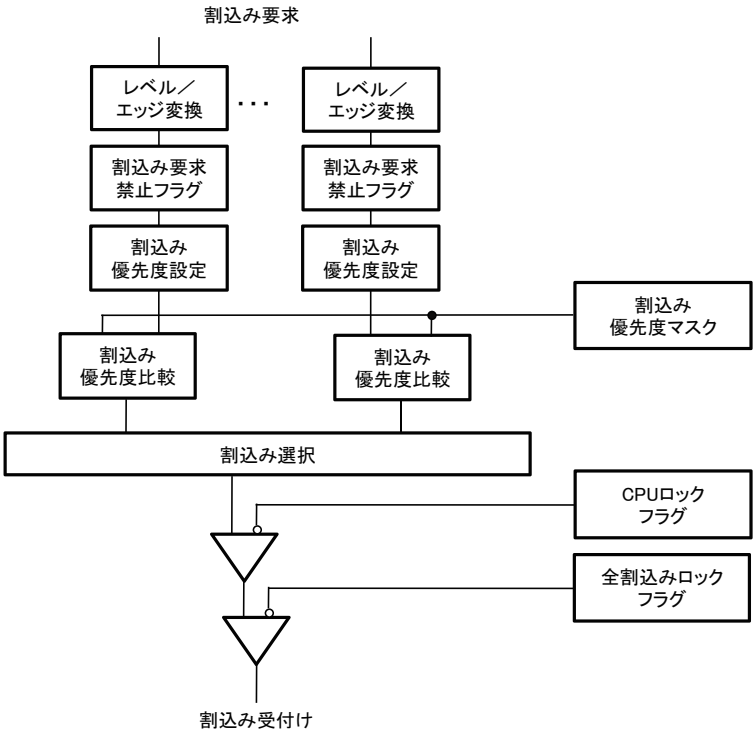

図 6.1: 割込み処理モデル

6.4.3　実装方法

割込み要求を受け付け後の処理を，割込み処理のスケジューリングごとに説明する．まず，グローバル固定優先度スケジューリングにおける処理の流れを図 6.2 に示す．グローバル固定優先度スケジューリングは，割込みが発生した際に，その割込みに対応する処理の優先度が，実行中の処理の優先度より高い場合に，実行中の処理に優先して割込み処理を実行する．この場合は，先の割込み処理モデルをそのまま用いることができ，割込み要求受け付け後に，割込みハンドラから，アプリケーションが登録した割込みサービスルーチンを実行することで実現する．割込み応答時間は，割込み要求受付けから，割込みサービスルーチンを起動するまでの時間とする．なお，割込みサービスルーチンの実行時間が非常に短いアプリケーションにおいて，割込みサービスルーチンの実行時間を階層型スケジューラのバジェット管理の対象から外す（無視する）場合には，割込みサービスルーチンの実行前後にあるバジェット計測操作は不要となる．

次に，ローカル固定優先度スケジューリングにおける処理の流れを図 6.3 に示す．ローカル固定優先度スケジューリングは，割込み処理が属するアプリケーション

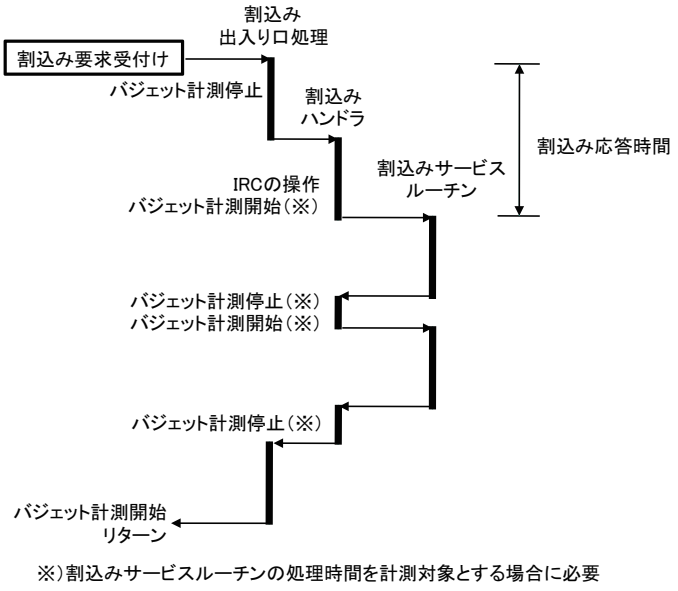

図 6.2: グローバル固定優先度スケジューリングにおける処理の流れ

内で優先度を比較する必要がある．まず，割込み要求発生後，カーネル処理などの割込み禁止区間（割込みロックされている状態）でない場合を除いては，一端割込み要求を受け付ける．その後で，割込み要求に対応する割込みサービスタスクを起動する．ここでタスクを起動しても，即座に実行を開始できるのではなく，タスクを実行可能な状態にして，アプリケーション内でのスケジューリングされるだけである．割込みサービスルーチンは，呼び出されると即座に実行されるのに対して，割込みサービスタスクは，起動直後に実行されない．すなわち，起動後に，アプリケーションのデッドラインを更新し，アプリケーションのデッドラインがもっとも早ければ，実行中の処理から割込みサービスタスクに実行を切り替えるディスパッチ要求を出す．ディスパッチ要求が発生しているアプリケーションでは，割込み出口処理で，割込みサービスタスクに実行が切り替わる．このように，割込み処理をタスク化している理由は，アプリケーション内に限定した優先度スケジューリングを容易に実現できることと，アプリケーション間を EDF スケジューリングでスケジュールすることで，割込み処理実行中であっても別のアプリケーションに属するタスクを実行できるよう実行状態を保存する必要があるためである．

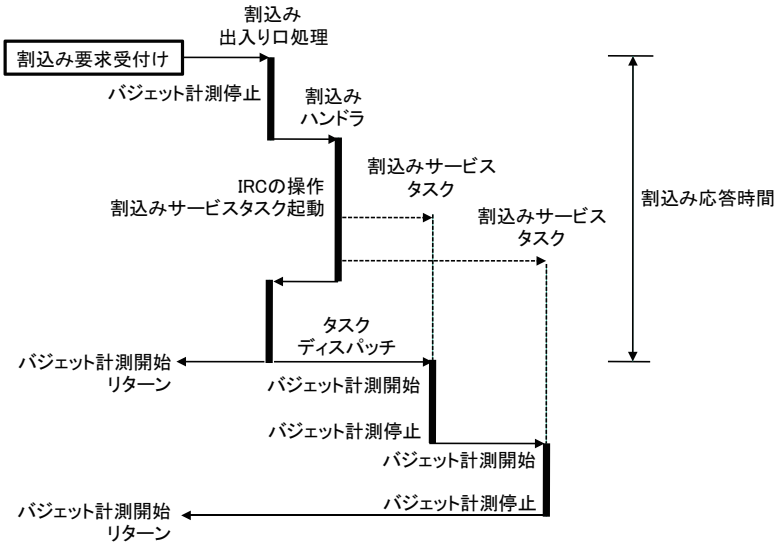

図 6.3: ローカル固定優先度スケジューリングにおける処理の流れ

表 6.2: 割込み処理の応答時間

割込み処理	応答時間（us）
割込みハンドラ	3
割込みサービスルーチン	10(3)
割込みサービスタスク	37

6.4.4　性能評価

割込み処理ごとに，割込み応答時間を測定した結果を表 6.2 に示す．測定した応答時間は，割込み要求が発生してから，それぞれの割込み処理が起動されるまでの時間である．それぞれの割込み処理の優先度は，システム内で最高であるものとし，他のアプリケーションや割込み処理に実行を邪魔されることはない．すなわち，割込要求の発生に対して，もっとも早く応答できる状況を想定している．

割込みハンドラは，割込み処理スケジューリングに関係なく，最初に起動する処理である．グローバル固定優先度スケジューリングにおける割込みサービスルーチンの応答時間は，割込みハンドラの応答時間に対して，割込みサービスルーチン呼び出しとバジェット管理の処理時間が加わり，10μ 秒となった．バジェット計測処理がない場合の応答時間は 3μ 秒となり，割込みハンドラと同等である．

一方，ローカル固定優先度スケジューリングにおける割込みサービスタスクの応答時間は，割込みハンドラ中に起動されてから割込み出口処理からタスク切り替

えが発生するまで実行を待たされるため，割込みサービスルーチンの応答時間に比べて 3.7 倍に増加した．この原因は，アプリケーション内でのタスクスケジューリング，アプリケーションのデッドライン更新，アプリケーションのスケジューリングなどの処理時間である．これらの処理の一部をハードウェアで実現することで，ローカル固定優先度スケジューリングの割込み応答時間を短縮できると思われるが，それは今後の課題とする．

6.5　解析

6.5.1　用語の定義

ここでは，スケジュール可能性を解析するための用語を定義する．統合プロセッサでは，N 個のアプリケーションが動作し，それぞれのアプリケーションは，A_i=(U_i,D_i,IDT_i) で表す．ここに，U_i は A_i に設定されるシェア，D_i は A_i の相対デッドライン，IDT_i は，A_i の最大割込み禁止時間である．時間保護アルゴリズム 1，2 のように，相対デッドラインが実行時に変動する場合には，最小の相対デッドラインとする．

6.5.2　スケジュール可能性解析

グローバル固定優先度スケジューリングにおけるスケジュール可能性解析手法としては，割込み処理は RMA による解析で，アプリケーションは EDF スケジューリングの解析方法を別々に適用する手法 [27] を適用できる．この場合でも，アプリケーションに属する個々のタスクがスケジュール可能であることを解析するには，さらに検討が必要であるが，これは今後の課題とする．

次に，ローカル固定優先度スケジューリングでは，タスクスケジューリングが固定優先度スケジューリングであり，さらに，割込みサービスタスクは通常のタスクと同様に扱うことができることから，第 4 章で述べた時間保護アルゴリズム 2 の証明を利用できる．したがって，タスクと割込みサービスタスクをスケジュール可能であることは，4.5 節より明らかである．タスクスケジューリングが EDF ス

ケジューリングの場合は，BSS アルゴリズムのスケジュール可能性解析の結果 [23] を利用できる．

6.5.3　割込み禁止時間を考慮した統合可否判定式

ローカル固定優先度スケジューリングにおいて，割込み禁止時間を考慮したアプリケーションの統合可否判定式について述べる．この式を使用することで，アプリケーションを統合する際に，アプリケーションの詳細な情報を知ることなく，すべてのアプリケーションを統合可能であるかを容易に判断することができる．

前節では，ローカル固定優先度スケジューリングおいては，割込み禁止時間が存在しない場合に，統合するアプリケーションの割込みサービスタスク及び通常のタスクについて，時間保護の要件 (2) を満たすことが示された．タスクの処理中に割込み禁止区間が存在すると，その間は割込みハンドラの実行開始が遅延するため，割込みサービスタスクの起動も遅延する．遅延している間は，優先度の高い割込みサービスタスクよりも，優先度の低いタスクが優先して実行される．その結果，割込みサービスタスクがデッドラインをミスしてしまう可能性がある．

ここでは，タスクに割込み禁止区間が存在する場合でも，時間保護の要件 (2) を満たすための十分条件を検討する．割込みが遅延する振る舞いは，ノンプリエンプティブスケジューリングにおいて，低優先度タスクの実行中に，高優先度タスクが実行可能になって実行が待たされる状況に相当する．割込み禁止時間のアプリケーション内の影響は，タスク内に割込み禁止区間が存在する場合でも，統合前の個別プロセッサですべてのタスクがスケジュール可能であったという前提より明らかである．割込み禁止時間のアプリケーション間の影響は，ノンプリエンプティブな EDF スケジューリングのスケジュール可能性条件式 [14] を用いて，次の定理を得ることができる．

定理 5 割込み禁止区間をもつアプリケーション A_i が次の条件式を満たすとき，このアプリケーションは性能 *1* の統合プロセッサでスケジュール可能である．

$$\forall d_i \in [0, \max_i\{D_i\}],$$
$$d_i \geq \max_{i<j}\{IDT_j * U_j\} + \sum_{D_k \leq d_i} (d_i * U_k) \tag{6.1}$$

(証明) この条件式は，文献 [14] の定理 12 から容易に導くことができる．$\max_{i<j}\{IDT_j * U_j\}$ は，A_i より長い相対デッドラインをもつアプリケーションの割込み禁止時間により，実行が遅延する時間である．$\sum_{D_k \le d_i}(d_i * U_k)$ は，A_i より絶対デッドラインの早いアプリケーションにより，実行が遅延する時間の合計と，A_i 自身の実行時間である．

この条件式は，スケジュール可能であるための必要十分条件式であるが，アプリケーションに含まれる処理の割込み禁止時間と相対デッドラインを把握する必要があることと，計算が複雑であるという問題がある．そこで，各アプリケーションの相対デッドライン D_i と IDT_i のみで計算できるように条件式を簡単化する．

$$\begin{aligned} d_i &\ge \frac{\max_{i<j}\{IDT_j * U_j\}}{1 - \sum_{D_k \le d_i}(U_k)} \\ &\ge \frac{\max_{i<j}\{IDT_j * U_j\}}{U_j} \\ &\ge \max_{i<j}\{\frac{IDT_j * U_j}{U_j}\} \\ &\ge \max_{i<j}\{IDT_j\} \\ &\ge \max_{i}\{IDT_i\} \end{aligned}$$

さらに，左辺を最小化すると，次の十分条件が得られる．

$$\min_i\{D_i\} \ge \max_i\{IDT_i\} \tag{6.2}$$

このスケジュール可能性条件式を利用して，アプリケーションの統合可否を判定する例を 2 つ示す．1 つ目の例は，3 つのアプリケーションを統合する場合で，それぞれのパラメータは，$A_1 = (0.25, 6, 1)$，$A_2 = (0.25, 6, 1)$，$A_3 = (0.5, 8, 7)$ とする．この場合，最小の相対デッドラインは，$\min\{6, 6, 8\} = 6$ で，最大割込禁止時間は，$\max\{1, 1, 7\} = 7$ である．このとき，条件式 (6.2) は満たされない．したがって，いずれかのアプリケーションがデッドラインをミスする可能性がある．2 つ目の例も 3 つのアプリケーションを統合する場合で，それぞれのパラメータは，$A_1 = (0.25, 6, 2)$，$A_2 = (0.25, 6, 2)$，$A_3 = (0.5, 20, 4)$ とする．最小の相対デッドラインは $\min\{6, 6, 20\} = 6$，最大割込禁止時間は $\max\{2, 2, 4\} = 4$ である．この場合，

表 6.3: 割込み処理のスケジューリングの比較

スケジューリング	優先度比較の範囲	階層型スケジューラへの適用性	実装容易性	割込み応答性
グローバル固定優先度	システム全体	×	○	○
ローカル固定優先度	アプリケーション内	○	△	△

条件 (6.2) は満たさせる．したがって，これらのアプリケーションは統合プロセッサでスケジュール可能である．

提案する条件式は，十分条件であり必要条件ではないため，条件式を満たさない場合には，厳密は条件式によりスケジュール可能性を詳細に解析するか，条件式を満たせるように，割込み禁止時間を短くするようアプリケーションの構成を改善することが必要となる．

6.6　議論

性能評価結果を踏まえ，割込み処理スケジューリングを表 6.3 に整理し，有用性を議論する．グローバル固定優先度スケジューリングは，割込み処理をアプリケーションから分離し，システム上のすべてのタスクに優先してスケジューリングする．割込み処理モデルに対して実装が容易であり，割込み応答性も高い．それに対して，ローカル固定優先度スケジューリングは，タスクと同様に，割込み処理もアプリケーションごとにスケジューリングするため，割込みサービスタスクを導入することで対応する．割込み処理の主要部分をタスク化した割込みサービスタスクの割込み応答時間は，割込みサービスタスクの 10μ 秒に対して 37μ 秒に増加した．階層型スケジューラへの適用性は，割込み処理も階層型スケジューラでそのまま扱えるという意味で，ローカル固定優先度スケジューリングの方が高い．このことは，アプリケーションのスケジュール可能性解析において，これまでの解析手法を利用できるという利点がある．それに対して，グローバル固定優先度スケジューリングは，割込み処理とタスクのスケジューリングが異なるため，アプリケーションのスケジュール可能性解析においては，これまでの解析手法をそのまま利用することができない．以上より，既存の割込みアーキテクチャにおいて，容易に実現する場合には，グローバル固定優先度スケジューリングの方が有利であるが，ローカル固定優先度スケジューリングおける実装方法と割込み応答性を改善できれば，階層型スケジューラにおいては，ローカル固定優先度スケジューリングが適すると考える．

6.7　まとめ

本章では，タスクのみを対象とした階層型スケジューリングに対して，割込み処理のスケジューリングを組み合わせることで，タスクと割込み処理で構成されるアプリケーションに対応する手法を提案した．まず，割込み処理のスケジューリングをシステム全体で行うグローバル固定優先度方式と，個々のアプリケーション内で行うローカル固定優先度方式の 2 つに整理した．次に，それぞれの方式を階層型スケジューラに実装する方法を述べ，割込み応答性を評価した．さらに，割込み禁止時間をもつアプリケーションの統合可否判定式について述べた．性能評価と解析の結果を通して，ローカル固定優先度方式は，実装が容易なグローバル固定優先度方式に比較して，30μ 秒程度，応答時間が長くなることが明らかになったが，階層型スケジューラへの適用性が高いため，アプリケーション統合においては，ローカル固定優先度方式が適するという結論を得た．

第7章　結論

7.1　まとめ

本研究では，分散制御システムにおいて，個別のプロセッサで単独動作するアプリケーションを，1つのプロセッサに統合して動作させる状況を想定し，プロセッサの時間保護を実現する階層型スケジューリングアルゴリズムを提案した．さらに，提案アルゴリズムを，実際に既存の RTOS に実装して処理時間を評価することで，アルゴリズムの実現可能性と実用性を明らかにした．この結果から，開発したスケジューリングアルゴリズムと RTOS は，実際のリアルタイムアプリケーションの統合に適用できる可能性があると考える．しかしながら，自動車制御システムには，本研究の前提に合致しないアプリケーションも多いため，今後さらなる改良が必要である．

本研究の主な貢献は次の 4 つである．

1つ目は，時間保護を実現するための基本的な階層型スケジューリングアルゴリズムを提案したことである．時間保護を定義するために，タスクのスケジューリングアルゴリズムに対する要求事項を 3 つに整理し，その要求事項をすべて満たすスケジューリングアルゴリズムを提案した．アルゴリズムを適用するための制約条件としては，タスクの正確な実行時間の情報を必要とせずに，タスクの起動時刻と相対デッドラインの 2 つの情報が得られる前提を置いた．提案アルゴリズムの動作を記述し，時間保護の 3 つの要求事項をすべて満たすことを証明した．

2つ目は，起動時刻が不明なタスクを含むアプリケーションに対応したスケジューリングアルゴリズムを提案したことである．最初に開発した時間保護スケジューリングアルゴリズムでは，適用するための条件として，タスクの起動時刻とデッドラインの 2 つの情報が得られることを前提としているのに対して，このアルゴリズムは，タスクのデッドライン情報のみを使い，時間保護の 3 つの要求事項のうち，特に重要な 2 つを満たすことができるスケジューリングアルゴリズムを開発した．提案アルゴリズムの動作を記述し，時間保護の 2 つの要求事項を満たすことを証明した．さらに，タスクのスケジューリングに特化したシミュレータを開発し，提

案アルゴリズムの正当性を確認した．

3つ目は，スケジューリングフレームワークの提案である．提案した階層型スケジューリングアルゴリズムを実装し，スケジューリング動作時の処理オーバヘッドを測定・評価するためのスケジューリングフレームワークを提案したことである．このフレームワークの特徴は，アプリケーションごとに適用するスケジューリングアルゴリズムを選択できることで，我々が提案する階層型スケジューリングアルゴリズムだけでなく，過去に提案された多くのスケジューリングアルゴリズムを選択することができる点にある．実装のベースとした TOPPERS/ATK1 と各処理時間比較した．その結果，使用 ROM/RAM 容量は，ROM が 6.4KB，RAM が 0.7KB それぞれ増加する程度で実装可能であることが確認できた．アプリケーション内のタスク切替時間の増加は，TOPPERS/ATK1 の 49μs に対して，周期実行アルゴリズムで 41μs，基本的な時間保護アルゴリズムで 141μs それぞれ処理時間が増加した．アプリケーション間のタスク切替では約 250μs となった．実装ターゲットの CPU の動作周波数が 10MHz であり，現実の ECU に利用されるプロセッサを 100MHz 程度と想定すると，単純計算で処理時間は 1/10 になる．この点を考慮すると，タスク切替え処理で，数十 μsec 程度のオーバヘッドとなることが予想される．この数値は，多くの組込みリアルタイムシステムにおいては許容可能であると考える．

4つ目は，割込み処理を含むアプリケーションに対応するためのスケジューリングを提案したことである．過去に提案されたアルゴリズム及び，先に我々が提案したスケジューリングアルゴリズムは，タスクのみを対象としており，高い応答性が要求される割込み処理を考慮していなかった．階層型スケジューリングアルゴリズムに適用できる割込み処理のスケジューリングアルゴリズムとして，グローバル優先度スケジューリングとローカル優先度スケジューリングを提案した．グローバル固定優先度スケジューリングは，割込み処理をアプリケーションから分離し，システム上のすべてのタスクに優先してスケジューリングする．割込み処理モデルに対して実装が容易であり，割込み応答性も高い．それに対して，ローカル固定優先度スケジューリングは，タスクと同様に，割込み処理もアプリケーションごとにスケジューリングする．既存の割込み処理モデルに対しては，実装は困難で，割込み処理をタスク化した割込みサービスタスクによる実装では，割込み応答時間は，割込みサービスタスクの 10μ 秒に対して 37μ 秒に増加した．提案した 2 つの手法を，階層型スケジューラへの適用性，実装容易性，割込み応答性の観点で比較した結果，既存の割込みアーキテクチャにおいて，容易に実現する場合

には，グローバル固定優先度スケジューリングの方が有効であるが，ローカル固定優先度スケジューリングおける実装方法と割込み応答性を改善できれば，階層型スケジューラにおいては，ローカル固定優先度スケジューリングの方が適することを明らかにした．

7.2　今後の課題

今後の課題としては，まず第5章で述べたスケジューリングフレームワークに対して，第4章で述べた階層型スケジューリングアルゴリズムを実装して，実機での性能評価を実施することがあげられる．この階層型スケジューリングアルゴリズムは，基本的なアルゴリズムに比較して，バジェットの管理方法が複雑化する．そのため，グローバルスケジューリングでのバジェット管理方法が，選択可能なアルゴリズムの中で最も複雑なアルゴリズムの仕組みに合わせる必要が出てくる．この場合の効率的な設計と実装方法を検討する必要がある．

時間保護のスケジューリングアルゴリズムについては，タスクモデルをより実用的な方向に拡張する必要がある．現在のタスクモデルは，タスクが待ち状態になることを考慮していないが，例えば，セマフォやミューテックスを用いるタスク間通信では，タスクの待ち状態を考慮する必要がある．

発展的な課題として，現在はプロセッサ間のネットワークを介している通信を，アプリケーション間通信に置き換える手法を検討する．自動車制御システムでは，ECU 間のデータ通信はハーネスを介して行なっているが，ECU を統合した環境では，ECU 間通信をアプリケーション間の通信として変換できる．これが実現できると，ECU の数だけではなく，ハーネスの数を削減することができる．

謝辞

本博士論文に関する研究全過程を通じて，御指導と御助言を頂きました名古屋大学大学院情報科学研究科情報システム学専攻の高田広章教授に，心から感謝致します.

日頃より有益な御討論，御指導を頂きました名古屋大学大学院情報科学研究科附属組込みシステム研究センターの本田晋也准教授に感謝致します.

本論文に関する研究の一部においては，2005 年度，2006 年度 IPA 未踏ソフトウェア創造事業の援助を頂きました．その際，プロジェクトマネージャーとして大変有効な御助言を頂きました東京農工大学の並木美太郎教授に感謝致します.

本研究を進めるに当たり，日々御指導，御討論頂いた名古屋大学大学院情報科学研究科情報システム学専攻高田研究室の皆様と，附属組込みシステム研究センターの皆様に感謝致します.

最後に，長い間，研究活動をサポートしてくれた家族に心から感謝致します.

参考文献

[1] Abeni, L. and Buttazzo, G.: Integrating multimedia applications in hard real-time systems, Proc. IEEE Real-Time Systems Symposium, 1998.

[2] Abeni, L. and Buttazzo, G.: Resource reservations in dynamic real-time systems, Vol. 27, pp. 123–165, Real-Time Systems, 2004.

[3] Airlines Electronic Engineering Committee: AVIONICS APPLICATION SOFTWARE STANDARD INTERFACE, 2008.

[4] AUTOSAR: http://www.autosar.org/.

[5] AUTOSAR: Specification of Operating System V4.0.0, 2009.

[6] Buttazzo, G.: HARD REAL-TIME COMPUTING SYSTEMS: Predictable Scheduling Algorithms and Applications, Second Edition, Springer, 2005.

[7] Caccamo, M., Buttazoo, G., Thomas, D.: Efficient reclaiming in reservation-based real-time systems with variable execution times, IEEE Trans. on Computers, Vol. 54, pp. 198–213, 2005.

[8] Caccamo, M., Buttazzo, G., Sha, L.: Capacity sharing for overrun control, Proc. IEEE 21th Real-Time System Symposium, 2000.

[9] Carpenter, J., Lipari, G., Baruah, S.: A framework for acheving inter-application isolation in multiprogrammed, hard real-time environments, Proc. IEEE Real-time Systems Symposium, pp. 217–226, 2000.

[10] Deng, Z., Liu, J.W.-S., Sun, J.: A scheme for scheduling hard real-time applications in open system environment, Proc. 9th Euromicro Workshop on Real-Time Systems, pp 191–199, 1997.

[11] Deng, Z., Liu, J.W.-S. Zhang, L., Mouna, S., Frei, A.: An open environment for real-time applications, Real-Time Systems Journal, volume 16, pp. 155–185, 1999.

[12] Gai, P., Abeni, L., Giorgi, M., Buttazzo, G.: A new kernel approach for modular real-time systems development, Proc. IEEE 13th Euromicro Conference on Real-Time Systems, 2001.

[13] Gai, P., Lipari, G., Abeni, L., di Natale, M., Bini, E.: Architecture for a portable open source real time kernel environment, Proc. 2nd Real-Time Linux Workshop and Hand's on Real-Time Linux Tutorial, 2000.

[14] George, L., Rivierre, N., Spuri, M.: Preemptive and non-preemptive real-time uni-processor scheduling, INRIA RECQUENCOURT, September 1996.

[15] HIS: http://portal.automotive-his.de

[16] HIS: Requirements for Protected Applications under OSEK Version 1, 2002.

[17] HIS: OSEK OS Extensions for Protected Applications Version 1.0, 2003.

[18] International Electrotechnical Commision: http://www.iec.ch

[19] ITRON Project: http://www.assoc.tron.org/jpn/itron-doc.html.

[20] Jeffay, K., Stone, D.L.: Accounting for interrupt handling costs in dynamic priority task systems, Proc. the 14th IEEE Real-Time Systems Symposium, 1993.

[21] Leyva-del Foyo, L. E., Mejia-Alvarez, P., de Niz, D.: Predictable interrupt scheduling with low overhead for real-time kernels, Proc. the 12th IEEE International Conference on Embedded and Real-Time Computing System and Applications, 2006.

[22] Lipari, G.: Resource reservation in real-time systems, Ph.D Thesis, Scuola Superiore S.Anna, 2000.

[23] Lipari, G., Baruah, S.: Efficient scheduling of real-time multi-task applications in dynamic systems, Proc. IEEE Real-Time Technology and Applications Symposium, 2000.

[24] Lipari, G., Baruah, S.: Greedy reclaimation of unused bandwidth in constant-bandwidth servers, Proc. IEEE 12th Euromicro Conference on Real-Time Systems, pp. 193–200, 2000.

[25] Lipari, G., Bini, E.: A methodology for designing hierarchical scheduling systems, Journal fo Embedded Computing, Cenbridge International Science Publishing, 2003.

[26] Lipari, G., Gai, P., Trimarchi, M., Guidi, G., Ancilotti, P.: A hierarchical framework for component-based real-time systems, Proc. IEEE International Symposium on Component-based Software Engineering, 2004.

[27] Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment, Journal of the ACM (JACM), pp. 46–61, 1973.

[28] Matsubara, Y., Honda, S., Takada, H.: Hierarchical Scheduling for Integrating Real-time Applications with Interrupt Routines, Proc. IEEE International SoC Design Conference, 2009.

[29] Nogueira, L., Pinho, L. M.: Capacity sharing and stealing in dynamic server-based real-time systems, Proc. IEEE International Parallel and Distributed Processing Symposium, 2007.

[30] Oikawa, S., Ishikawa, H., Iwasaki, M., Nakajima, T.: Providing protected execution environments for embedded operating systems using a u-kernel, Proc. International Conference on Embedded and Ubiquitous Computing, pp. 153–163, 2004.

[31] Oikawa ,S., Rjkumar, R.: Portable RK: A portable resource kernel for guaranteed and enforced timing behavior, Proc. IEEE Real Time Technology and Applications Symposium, 1999.

[32] OSEK/VDX: http://www.osek-vdx.org

[33] OSEK/VDX: Operating System Specification Version 2.2.3, 2005.

[34] Regehr, J., Reid, A., Webb, K., Parker, M., Lepreau, J.: Evolving real-time systems using hierarchical scheduling and concurrency analysis, Proc. the 24th IEEE Real-Time Systems Symposium, 2003.

[35] Regehr, J., Stankovic, J. A.: HLS:a framework for composing soft real-time schedulers, Proc. IEEE 22nd Real-Time System Symposium, 2001.

[36] Shin, I., Lee, I.: Compositional real-time scheduling framework, Proc. IEEE Real-time Systems Symposium, pp. 57–67, 2004.

[37] Spuri, M., Buttazzo, G.: Scheduling aperiodic tasks in dynamic priority systems, Real-Time Systems Journal, Vol. 10, pp. 179–210, 1996.

[38] Strosnider, J., Lehoczky, J.P., Sha, L.: The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments, IEEE Transactions on Computers, Vol. 44, 1995.

[39] The Version-Up WG of the TRON Association :Protection Extension of μITRON4.0 Specification.

[40] TOPPERS Project: TOPPERS 新世代カーネル統合仕様書 Release 1.1.0, 2009.

[41] TOPPERS/ATK1: http://toppers.jp/atk1.html.

[42] TOPPERS/ASP カーネル: http://toppers.jp/asp.html.

[43] Zhang, Y., West, R.: Process-aware interrupt scheduling and accounting, Proc. the 27th IEEE Real-Time Systems Symposium, 2006.

[44] 西部満, 本田晋也, 冨山宏之, 高田広章: ハードリアルタイムシステムに適したメモリ保護機構の提案と評価, 情報処理学会 組込み技術とネットワークに関するワークショップ, 2005.

[45] 松原豊, 本田晋也, 高田広章: 時間保護のためのタスク起動遅延付き階層型スケジューリングアルゴリズム, 情報処理学会研究報告, Vol. 2010-EMB-19, 2010.

[46] 松原豊, 本田晋也, 冨山宏之, 高田広章: 時間保護のためのリアルタイムスケジューリングアルゴリズム, 情報処理学会論文誌 コンピューティングシステム, Vol. 48, pp. 192–202, 2007.

[47] 松原豊, 本田晋也, 冨山宏之, 高田広章: リアルタイムアプリケーション統合のための柔軟なスケジューリングフレームワーク, 情報処理学会論文誌組込みシステム工学特集, Vol. 49, pp. 3508–3519, 2008.

研究業績

学術論文

- 松原豊, 本田晋也, 冨山宏之, 高田広章, ”時間保護のためのリアルタイムスケジューリングアルゴリズム”, 情報処理学会論文誌コンピューティングシステム, Vol.48, No.SIG 8(ACS 18), pp. 192-202, May 2007.
- 松原豊, 本田晋也, 冨山宏之, 高田広章, ”リアルタイムアプリケーション統合のための柔軟なスケジューリングフレームワーク”, 情報処理学会論文誌 組込みシステム工学特集, Vol.49, No.10, pp. 3508-3519, Oct 2008.
- 一場利幸, 松原豊, 本田晋也, 高田広章, ”中断可能な優先度継承スピンロックとそのハードウェア実装”, 情報処理学会論文誌コンピューティングシステム, 2011.（採録決定）

国際会議論文

- Yutaka Matsubara, Midori Sugaya, Ittetsu Taniguchi, Yasuaki Murakami, Hayato Kanai and Hiroaki Takada, ”SSEST: Summer School on Embedded System Technologies”, Proceedings of the 1st Asia-Pacific Workshop on Embedded System Education and Research (APESER), Hsinchu, Taiwan, Dec 2007.
- Hideki Takase, Takuya Azumi, Ittetsu Taniguchi, Yutaka Matsubara, Hayato Kanai, Shintaro Hosoai and Midori Sugaya, ”History of Summer School on Embedded System Technologies Organized by Students and Young Engineers”, 2009 Workshop on Embedded Systems Education (WESE’09), pp. 19-26, Grenoble, France, Oct 2009.
- Yutaka Matsubara, Shinya Honda and Hiroaki Takada, ”Hierarchical Scheduling for Integrating Real-time Applications with Interrupt Routines”, IEEE International SoC Design Conference (ISOCC) 2009, Busan, Korea, Nov 2009.

国内会議発表論文（査読あり）

- 松原豊, 冨山宏之, 高田広章, ”階層型スケジューラによる自動車制御システム向け時間保護環境”, 組込みソフトウェアシンポジウム 2005 論文集, pp. 110–116, 東京都, Sep 2005.
- 菅谷みどり, 今井陽平, 殷中翔, 大山将城, 谷口一徹, 谷崎裕明, Chaiwat Sathawornwichit, 野田厚志, 松原豊, 茂田井寛隆, ”スキル＆コミュニケーションの向上を目的とした学生主催の教育プロジェクトの運営と実施”, 組込みシステムシンポジウム 2006 論文集, Vol.2006 No.13, pp.118-122, 東京, Oct 2006.
- 松原豊, 本田晋也, 冨山宏之, 高田広章, ”OSEK アプリケーション統合のための柔軟なスケジューリングフレームワーク”, 組込みシステムシンポジウム 2007, Vol.2007, No.8, pp. 33-41, Oct 2007.（2007 年度 CS 領域奨励賞受賞）
- 一場利幸, 松原豊, 本田晋也, 高田広章, ”中断可能なキューイングスピンロックのハードウェア実装と評価”, 組込みシステムシンポジウム 2010 (ESS2010) 論文集, 東京, Oct 2010.
- 金周慧, 松原豊, 高田広章, ”状態遷移図に着目した安全要求分析手法”, 第 8 回クリティカルソフトウェアワークショップ（WOCS2011）, 東京, Jan 2011.

研究会発表論文（査読なし）

- 菅谷みどり, 今井陽平, 殷中翔, 大山将城, 谷口一徹, 谷崎裕明, Chaiwat Sathawornwichit, 野田厚志, 松原豊, 茂田井寛隆, ”組込みシステム技術に関するサマースクール（SSEST1）”, 第 7 回組込みシステム技術に関するサマーワークショップ（SWEST7）予稿集, pp.120-121, 浜松, Jul 2005.
- 松原豊, 小田哲也, 金井勇人, 谷口一徹, 中野俊和, 野田厚志, 御村武生, 村上靖明, ”第 2 回組込みシステム技術に関するサマースクール（SSEST2）”, 第 8 回組込みシステム技術に関するサマーワークショップ（SWEST8）予稿集, pp.107-109, 浜松, Aug 2006.
- 松原豊, 本田晋也, 冨山宏之, 高田広章, ”タスク優先度を考慮した時間保護スケジューリングアルゴリズム”, 電子情報通信学会技術研究報告コンピュータシステム, Vol.107, No.558, pp. 173-178, 屋久島, Mar 2008.

- 松原豊, 本田晋也, 高田広章, ”割込み処理を含むリアルタイムアプリケーション統合のための階層型スケジューリング”, 情報処理学会研究報告, Vol.2009-EMB-14, No.7, 名古屋, Jul 2009.
- 金周慧, 松原豊, 高田広章, ”小規模な組込みシステム設計における安全要求分析の事例”, 第 6 回ディペンダブルシステムシンポジウム（DSS2009）, Vol.2009, No.X, 大阪, Dec 2009.
- 松原豊, 本田晋也, 高田広章, ”タスクのデッドラインのみを用いる時間保護スケジューリングアルゴリズム”, 情報処理学会研究報告, Vol.2010-EMB-15, No.8, 横浜, Jan 2010.
- 一場利幸, 松原豊, 本田晋也, 高田広章, ”中断可能なキューイングスピンロックのハードウェア実装と評価”, 組込み技術とネットワークに関するワークショップ (ETNET2010), 八丈島, Mar 2010.
- 金周慧, 松原豊, 高田広章, ”小規模な組込みシステム設計における安全機能の適用事例”, 安全工学シンポジウム 2010, 東京, Jul 2010.
- 石谷健, 山崎二三雄, 長尾卓哉, 山田真大, 松原豊, 本田晋也, 高田広章, ”車載 ECU 統合向け異種 OS 間通信ミドルウェア”, 情報処理学会研究報告, Vol.2010-EMB-17, No.8, 東京, Jul 2010.
- 長尾卓哉, 山崎二三雄, 山田真大, 石谷健, 松原豊, 本田晋也, 高田広章, ”DUOS: 車載ECU 統合向け RTOS フレームワーク”, 情報処理学会研究報告, Vol.2010-EMB-17, No.8, 東京, Jul 2010.
- 石川拓也, 松原豊, 高田広章, ”分散リアルタイムシステムの端点間処理における応答時間の確率的解析”, 情報処理学会研究報告, Vol.2010-EMB-18, 函館, Aug 2010.
- 松原豊, 本田晋也, 高田広章, ”時間保護のためのタスク起動遅延付き階層型スケジューリングアルゴリズム”, 情報処理学会研究報告, Vol.2010-EMB-19, 熊本, Dec 2010.

ポスター発表

- Zoohaye Kim, Yutaka Matsubara, Hiroaki Takada, ”A Case Study of Safety Requirements Analysis in Small Embedded System Design”, The 28th International System Safety Conference (ISSC2010), Minneapolis, Aug 2010.
- 加藤希, 松原豊, 本田晋也, 高田広章, ”時間保護 OS を用いた二輪倒立振子ロボット制御システムの開発”, 第２回名古屋大学組込みシステム研究センターシンポジウム, 名古屋, Sep 2010.

受賞等

- 2005 年度上期 IPA 未踏ソフトウェア創造事業 「時間保護機能をもつ組込みシステム向け RTOS の開発」 採択
- 2006 年度下期 IPA 未踏ソフトウェア創造事業 「時間保護のための組込みシステム向け階層型リアルタイム OS」 採択
- 情報処理学会 2007 年度コンピュータサイエンス領域奨励賞
- TOPPERS of the Year 2008「メモリ保護と時間保護を有する自動車向けリアルタイム OS」株式会社ヴィッツ（服部博行，大西秀一，片岡歩），名古屋大学情報科学研究科（松原豊，高田広章）
- 2008 年 第 10 回 LSI IP デザイン・アワード（企業部門）IP 優秀賞「オープンソース保護 OS:メモリ保護と時間保護を有する自動車向けリアルタイム OS」株式会社ヴィッツ（服部博行，大西秀一，片岡歩），名古屋大学情報科学研究科（松原豊，高田広章）