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

| | |
|---|---|
| Title | 組込みプロセッサの高速化機構と協調するＲＴＯＳの実装に関する研究 |
| Author(s) | 島田，信行 |
| Citation | |
| Issue Date | 2005-03 |
| Type | Thesis or Dissertation |
| Text version | author |
| URL | http://hdl.handle.net/10119/1924 |
| Rights | |
| Description | Supervisor:田中　清史，情報科学研究科，修士 |

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

修 士 論 文

# 組込みプロセッサの高速化機構と協調するRTOSの実装に関する研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

島田 信行

2005年3月

修 士 論 文

# 組込みプロセッサの高速化機構と協調するRTOSの実装に関する研究


指導教官　田中清史 助教授

審査委員主査　田中清史 助教授
審査委員　日比野靖 教授
審査委員　井口寧 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

310050 島田 信行


提出年月: 2005 年 2 月


Copyright © 2005 by Shimada Nobuyuki

## 概 要

適応型動的優先度方式は，動的にタスクの優先度を変更し，デッドラインオーバーするタスク数を減少させることを目的とする手法である．

本研究では，適応型動的優先度方式を採用し，組込み CPU の高速化機構と協調するリアルタイムオペレーティングシステムの実装方法を検討し，実際に実装を行うことにより，静的優先度方式との比較，評価を行う．また，動的優先度の計算に発生するオーバヘッドを正確に見積もることにより，オーバヘッドについても考察を行う．

# 目 次

**第 1 章　はじめに** **1**
1.1　研究の背景 . . . . . 1
1.2　研究の目的 . . . . . 1
1.3　本論文の構成 . . . . . 2

**第 2 章　リアルタイムシステム** **3**
2.1　タスクの状態と属性 . . . . . 3
2.1.1　タスクの状態 . . . . . 3
2.1.2　タスクの属性 . . . . . 6
2.2　リアルタイムスケジューリング . . . . . 7
2.2.1　ハードリアルタイムとソフトリアルタイム . . . . . 7
2.2.2　スケジューリングアルゴリズムの分類 . . . . . 8
2.2.3　WCET . . . . . 8
2.2.4　リアルタイムスケジューリングアルゴリズム . . . . . 9
2.3　スケジューラ . . . . . 12
2.3.1　スケジューラの構成 . . . . . 12
2.3.2　優先度更新方法 . . . . . 14
2.4　組込みプロセッサ . . . . . 15
2.4.1　高速割込み応答機構 . . . . . 15
2.4.2　データキャッシュ制御機構 . . . . . 17

**第 3 章　$\mu$ITRON4.0 インターフェースでの実装方法** **18**
3.1　リアルタイムオペレーティングシステムの構成 . . . . . 18
3.1.1　タスク管理 . . . . . 19
3.1.2　サービスコール . . . . . 22
3.2　評価環境の構成 . . . . . 25
3.3　内部トラップ . . . . . 30
3.4　割込み . . . . . 32

**第 4 章　評価** **34**
4.1　シミュレーションモデル . . . . . 34

4.2 動的優先度計算によるオーバヘッド . . . . . 34
4.3 静的優先度方式との比較 . . . . . 36
4.3.1 計測 1 . . . . . 36
4.3.2 計測 2 . . . . . 41
4.3.3 スケジューラのオーバヘッド . . . . . 43
4.3.4 補助計測 . . . . . 45
4.4 組込みプロセッサの評価 . . . . . 48
4.4.1 高速割込み応答機構の評価 . . . . . 48
4.4.2 データキャッシュ制御機構の評価 . . . . . 51

**第 5 章 おわりに** **52**
5.1 まとめ . . . . . 52
5.2 今後の課題 . . . . . 52

# 図 目 次


2.1 タスクの状態遷移 . . . . . 5
2.2 タスクの属性 . . . . . 6
2.3 (a) ハードリアルタイム，(b) ソフトリアルタイム . . . . . 7
2.4 静的優先度方式 . . . . . 10
2.5 提案手法でのレディキュー構造 . . . . . 13
2.6 実余裕時間に応じた優先度更新値 . . . . . 15
2.7 割込み発生時のレジスタセット切り替え . . . . . 16

3.1 評価環境 . . . . . 25
3.2 システム初期化手順 . . . . . 26
3.3 一つのバイナリファイルとして CPU シミュレータに渡される様子 . . . . . 28
3.4 CPU シミュレータの仮想的なアドレス空間 . . . . . 29
3.5 フラグ動作 . . . . . 31

4.1 計測 1 の動作 . . . . . 38
4.2 計測 1-1 . . . . . 39
4.3 計測 1-2 . . . . . 39
4.4 計測 2 の動作 . . . . . 41
4.5 計測 2-1 . . . . . 42
4.6 計測 2-2 . . . . . 42
4.7 計測 1 に除算を行った場合 . . . . . 44
4.8 計測 2 に除算を行った場合 . . . . . 44
4.9 計測 1 を変更 . . . . . 46
4.10 計測 2 を変更 . . . . . 46
4.11 計測 3-1 . . . . . 49
4.12 計測 3-2 . . . . . 49

# 第1章　はじめに

## 1.1　研究の背景

組込みシステムにおいて，機械／機器を制御するという意味合いから，汎用システム以上にリアルタイム性が求められる傾向にある．リアルタイム処理はオペレーティングシステム側のタスクスケジューリングに大きく依存するが，制御 CPU が他からの個々の要求への高速なレスポンスを達成することにより，リアルタイム処理における制約時間に余裕を持たせることが可能となる．組込みシステムにおける制御 CPU は，機器／周辺回路からの割込み要求に応答するのが通常であり，リアルタイム処理の実現のためには，この割込みに対する高速な応答機構が求められる．しかし，組込みシステム開発において，汎用プロセッサを低コスト／低消費電力化のために機能削減したものを，制御 CPU として使用しているに過ぎないのが現状である．

リアルタイムオペレーティングシステムとして現在広く使われている $\mu$ITRON では，タスクに与えた静的優先度に従いスケジューリングを行い，同じ優先度であれば先入れ先出しというスケジューリング方式がとられている．この方法では静的な優先度順位の低いタスクを処理できない可能性があることに着目し，タスク実行の時間要因に基づいた予測実行時間を利用し，動的なスケジューリングを行うことによって，制限時間を守るタスク数を増やす適応型動的優先度方式 [1] が提案された．

## 1.2　研究の目的

本研究では，低コストで組み用途に特化したプロセッサアーキテクチャ [2] をベースとし，その上で本アーキテクチャの持つ高速割込み応答機構，キャッシュ制御機構などの高速化機構を最大限利用するリアルタイムオペレーティングシステムを実装することを目的とする．また，組込みシステム分野におけるアプリケーション開発効率を考慮し，インタフェースは標準化された $\mu$ITRON4.0[3] の仕様に従う．また，従来のタスクスケジューラに変更を加え，従来の静的優先度方式を基本とし，タスクの実行終了時に適応型動的優先度方式を使用する構成を採る．

その他，CPU の高速割込み応答機構，キャッシュ制御機構等の高速化機構を最大限活用して実現する．

## 1.3 本論文の構成

本論文の構成を以下に示す.

第２章　リアルタイムシステム全般について説明し，提案手法であるスケジューラの構成について示す.

第３章　$\mu$ITRON4.0 仕様でのリアルタイムオペレーティングシステムの実装方法を示す.

第４章　静的優先度方式と適応型動的優先度方式についての評価を示す．また，適応型動的優先度方式での優先度計算のオーバヘッドについて考察を行う．更に，本研究で使用する CPU の高速割込み応答機構の評価を示す.

第５章　本論文のまとめ，及び今後の課題について.

# 第2章　リアルタイムシステム

## 2.1　タスクの状態と属性

本節では，本論文で使用する用語について $\mu$ITRON の仕様に基づき定義を行う．

### 2.1.1　タスクの状態

タスク[1]の状態は，大きく次の 5 つに分類される．この内，広義の待ち状態は，さらに 3 つの状態に分類される．また，実行状態と実行可能状態を総称して，実行できる状態と呼ぶ．

- 実行状態（RUNNING）
  現在そのタスクを実行中であるという状態．ただし，非タスクコンテキスト[2]を実行している間は，別に規定されている場合を除いて，非タスクコンテキストの実行を開始する前に実行していたタスクが実行状態であるものとする．

- 実行可能状態（READY）
  そのタスクを実行する準備は整っているが，そのタスクよりも優先順位の高いタスクが実行中であるために，そのタスクを実行できない状態．言い換えると，実行できる状態のタスクの中で最高の優先順位になればいつでも実行できる状態．

- 広義の待ち状態
  そのタスクを実行できる条件が整わないために，実行ができない状態．言い換えると，何らかの条件が満たされるのを待っている状態．タスクが広義の待ち状態にある間，プログラムカウンタやレジスタなどのプログラムの実行状態を表現する情報（コンテキスト）は保存されている．タスクを広義の待ち状態から実行再開する時には，プログラムカウンタやレジスタなどを広義の待ち状態になる直前の値に戻す．
  広義の待ち状態は，さらに次の 3 つの状態に分類される．

[1] ここでいうタスクとは，プログラムの並行実行の単位のことである．また，サービスコールを呼び出したタスクを「自タスク」と呼ぶ．

[2] タスク以外の処理が利用する環境のこと．割込みハンドラなどが利用する動作環境など．また，一般にプログラムの実行される環境を「コンテキスト」と呼ぶ．

- 待ち状態（WAITING）
  何らかの条件が整うまで自タスクの実行を中断するサービスコールを呼び出したことにより，実行が中断された状態．WAITING 状態は，その要因によって次のように分類することができる．
  * 起床待ち（slp_tsk，tslp_tsk）[3]
  * 時間待ち（dly_tsk）
  * イベントフラグ成立待ち（wai_flg，twai_flg）
  * セマフォ獲得待ち（wai_sem，twai_sem）
  * メールボックスでのメッセージ受信待ち（rcv_mbx，trcv_mbx）
  * 固定長メモリブロック獲得待ち（get_mpf，tget_mpf）
- 強制待ち状態（SUSPENDED）
  他のタスクによって，強制的に実行を中断させられた状態．ただし $\mu$ITRON4.0 仕様では，自タスクを強制待ち状態にすることもできる[4]．
- 二重待ち状態（WAITING-SUSPENDED）
  待ち状態と強制待ち状態が重なった状態．待ち状態にあるタスクに対して，強制待ち状態への移行が要求されると，二重待ち状態に移行させる．

- 休止状態（DORMANT）
  タスクがまだ起動されていないか，実行を終了した後の状態．タスクが休止状態にある間は，コンテキストは保存されていない．タスクを休止状態から起動する時には，タスクの起動番地から実行を開始する．また，別に規定されている場合を除いて，レジスタの内容は保証されない．

- 未登録状態（NON-EXISTENT）
  タスクがまだ生成されていないか，削除された後の，システムに登録されていない仮想的な状態．

実行可能状態に移行するタスクが，現在実行中のタスクよりも高い優先順位を持つ場合には，実行可能状態への移行と同時にディスパッチ[5]が起こり，即座に実行状態へ移行する場合がある．この場合，それまで実行状態であったタスクは，新たに実行状態へ移行したタスクにプリエンプトされたという．

実装したタスクの状態を図 2.1 に示す．未登録状態 (NON-EXISTENT）については本研究では実装しない．

実装での待ち状態に移行するサービスコールと待ち解除を行うサービスコールを①，②に示している．図のように，タスクは全部で 6 つの状態を持つことになる．

[3]括弧内は対応するサービスコール名．
[4]$\mu$ITRON4.0 以前の仕様では，自タスクを強制待ち状態にすることはできない．
[5]タスクの実行切り替え．

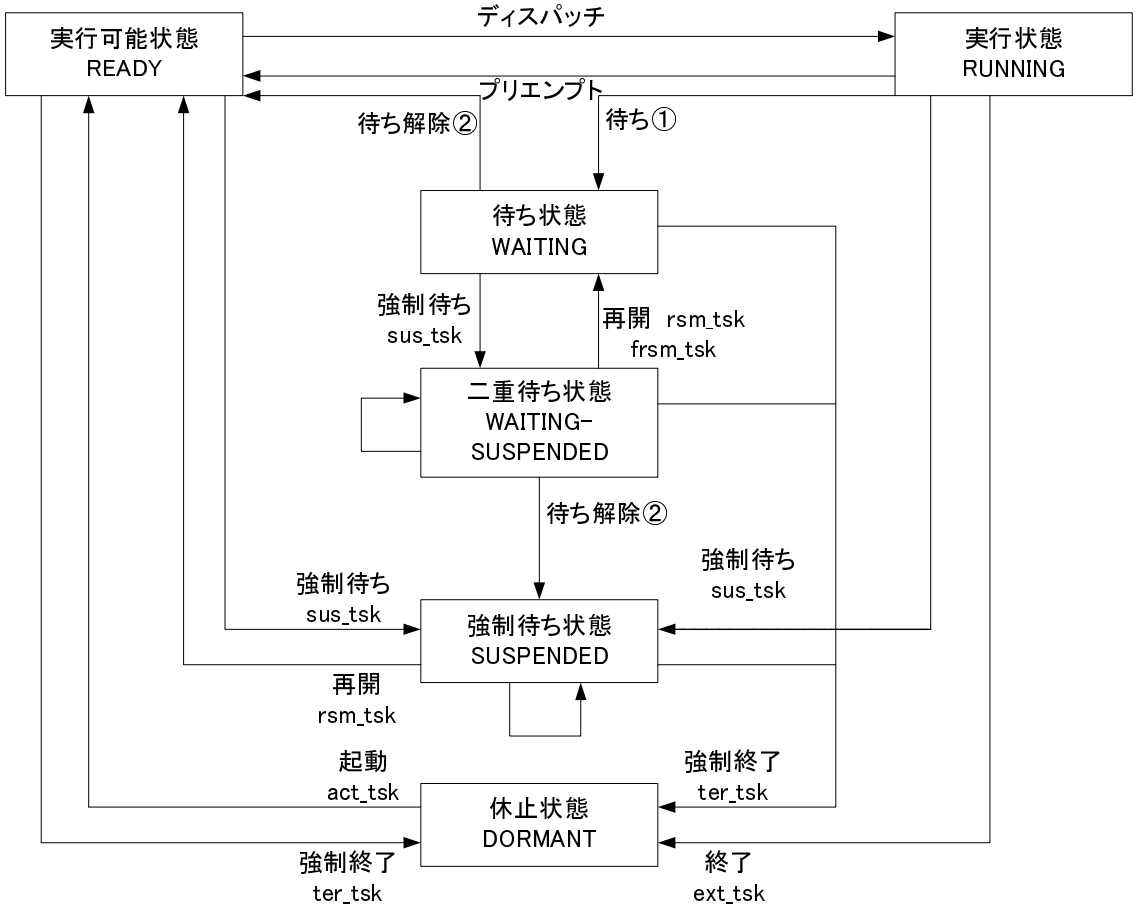


図 2.1: タスクの状態遷移

プリエンプティブなマルチタスク OS では，あるタスクの実行中に，それより優先度の高いタスクの実行が割込む場合がある．タスク切り替えの起きるタイミングとして，次の 4 通りがある．

1. 実行中のタスクが，自分より高優先度のタスクを起動，あるいは，待ち解除するようなサービスコールを発行した．
2. 非タスクコンテキスト（割込みハンドラ）から，実行中タスクより高優先度のタスクを起動，あるいは，待ち解除するようなサービスコールが発行された．
3. 実行中タスクより高優先度のタスクの待ち状態が，タイムアウトで解除された．
4. 実行中のタスクが，自ら待ち状態に入った，優先度を下げた，あるいは終了した．

逆にいえば，全てのサービスコールでタスク切り替えが起きるわけではない．実行中タスクより優先度が低いタスクに対して起動や待ち解除の操作を行っても，即座にはタスク切り替えは起こらない．

### 2.1.2 タスクの属性

タスクの時間に関する属性を図 2.2 に示す．タスクは休止状態から外部事象を受け，起動要求が発生した時刻 (activation time）に実行可能状態となる．次に，オペレーティンスシステムのスケジューラによりスケジューリングされ，実行権を得た後に実行される．タスクの処理が終了した場合は休止状態となり，次回の起動要求がくるまで実行されることはない．

タスクの終了時刻（finish time）から起動時刻を引いたものを応答時間（response time）という．終了時間がタスク起動毎に決められている締切時間（deadline)）を守るように，スケジューリングおよび実行されなければならない．締切時間を越えることなく実行が完了する実行開始時の最大遅延時間を余裕時間（laxity time）という．

一般に，タスクは起動の規則性によって周期的（periodic)，非周期的（aperiodic)，偶発的（sporadic）の 3 種類に分けられる．周期タスクは，一定の周期 T に従って規則的に起動する．非周期タスクは，再発するのが規則的ではない．偶発的タスクは，起動時刻を予期できないので非周期タスクの一種と考えることができる．ただし，最小起動間隔が決まっている点が異なる．

偶発的タスクを非周期タスクに含めて，単純に周期タスクと非周期タスクとに分類している文献も多い．本論文では，再発するか否かにに関係なく，規則的（周期的）でないタスクを全て非周期タスクと呼ぶ．

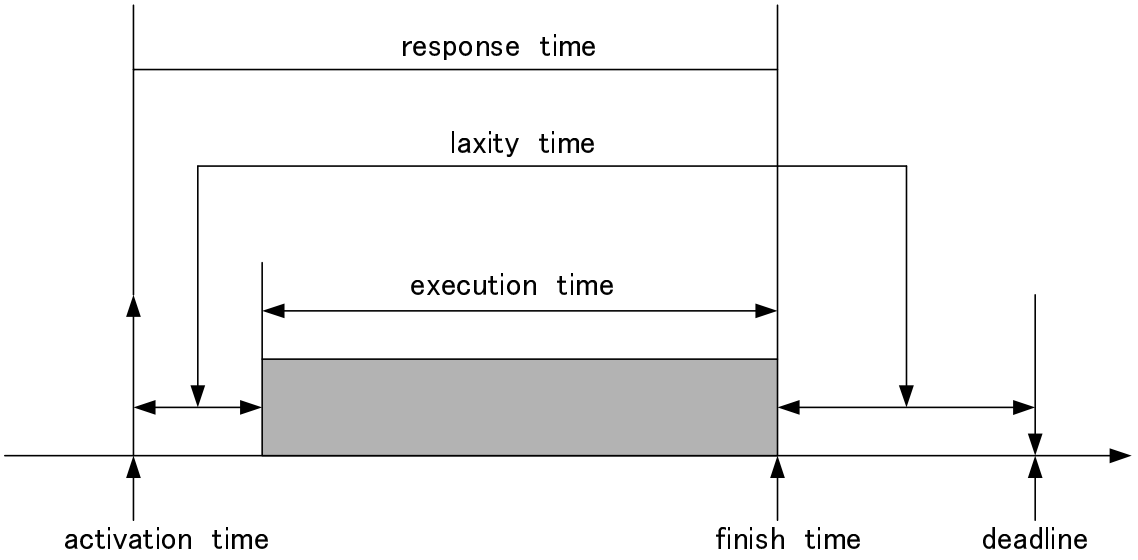


図 2.2: タスクの属性

## 2.2 リアルタイムスケジューリング

### 2.2.1 ハードリアルタイムとソフトリアルタイム

リアルタイム性には，大きく分けて次の 2 種類がある．

- ハードリアルタイム（hard real-time）
  ハードリアルタイムの場合，タスクは時間制約を必ず守らなければならず，少しでもデッドラインを過ぎてはならない．ハードリアルタイムにおいて，あるタスクがデッドラインオーバーを起こした場合，そのタスク（またはシステム）の価値はゼロになる (図 2.3(a)）．システムに損害を与える可能性があるので，システムを緊急停止する措置がとられることもある．ハードリアルタイムシステムの例としては，航空宇宙・自動車関連の機器や，発電所などが挙げられる．

- ソフトリアルタイム（soft real-time）
  ソフトリアルタイムの場合，過負荷状態の時などはデッドラインを多少過ぎても許容される．タスクの価値は，デッドラインを過ぎると徐々に減少していくが，システム自体に損害が及ぶことはない（図 2.3（b））．ソフトリアルタイムシステムの例としては，電話交換器や家電製品などが挙げられる．

システム全体がハードリアルタイムかソフトリアルタイムか，と決めることもできるが，現実的には一つのシステム内に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が混在することもある．

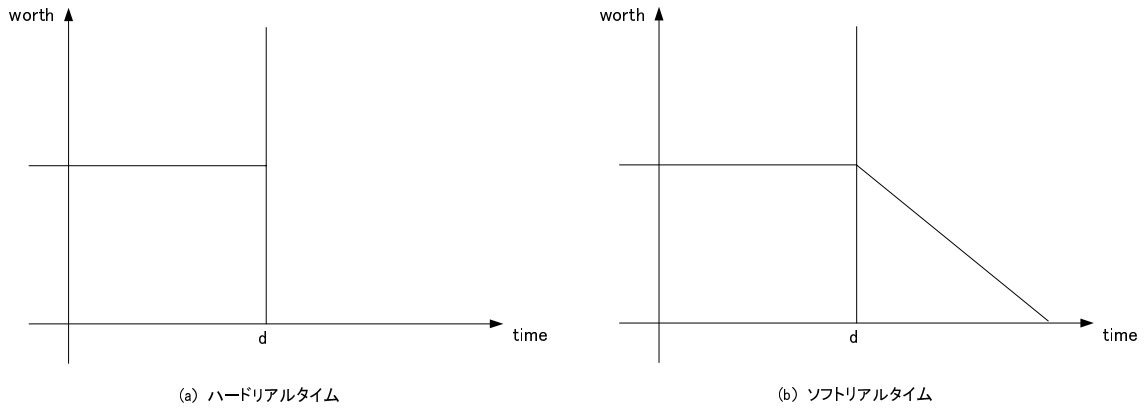


図 2.3: (a) ハードリアルタイム，(b) ソフトリアルタイム

### 2.2.2 スケジューリングアルゴリズムの分類

スケジューリングアルゴリズムの分類法にはいくつかある．ハードリアルタイムかソフトリアルタイムかはその一つであるが，それ以外の主なものを以下に挙げる．

- 最適（optimal），発見的（heuristic）
  スケジュール評価関数の最適値（通常は最小値）を与えるスケジューリングを，最適であるという．それに対して，最適性を保証できないものの，実用に堪える程度のスケジューリングを発見的であるという．

- オフライン（off-line），オンライン（on-line）
  システム稼動前（例えばコンパイル時など）にスケジューリングを済ませておく方式をオフライン方式という．事前に全てのタスクについて起動要求時刻などが既知でなければならず，柔軟性にも乏しい．しかし，最適なスケジューリングを行うことが可能であり，実行時のオーバヘッドが少ないという特徴がある．それに対して，オンライン方式ではシステム稼動中にスケジューリングを行う．具体的にはタスクの起動時と終了時にそれぞれスケジューリングが行われる．この方式の特徴はオフライン方式のそれと反対で，柔軟性はあるが最適なスケジューリング結果を得ることが難しく，実行時のオーバヘッドが大きくなる．

- 静的（static），動的（dynamic）
  タスクの優先度が静的か動的かという意味の分類である．静的優先度（固定優先度（fixed priority）とも呼ばれる.）とは，一度決められたら変化しないタスク優先度のことである．それに対して，タスクの時間属性などに基づいて変化する可能性があるのが動的優先度である[6]．

- プリエンプティブ（preemptive），ノンプリエンプティブ（non-preemptive）[7]
  タスク A が処理を実行している途中，さらに高い優先度を持つタスク B が起動された場合に，タスク A の処理を中断してタスク B に処理が移ることをプリエンプション（preemption）と呼び，プリエンプションの発生を許す性質をプリエンプティブという．逆に，タスク B の優先度がどれほど高くてもタスク A の処理が完了するまで実行権が移動しないなら，ノンプリエンプティブであるという[8]．

### 2.2.3 WCET

リアルタイムシステムに限らず，プログラムの実行時間は入力データなど条件の違いによって変動する．しかし，タスクの実行時間が分からなければデッドラインを保証するス

[6]動的／静的スケジューリングという言葉がそれぞれオンライン／オフラインスケジューリングの意味で用いられることもあるので注意が必要である．

[7]preemptive という言葉は「先取権がある」「中断可能である」「横取りできる」などと訳される．

[8]優先度の高いタスクが優先度の低いタスクから実行権を奪うことを「プリエンプトする」という．

ケジューリングは不可能なので，最長の実行時間を見積もることが行われる．あらゆる条件でタスクを実行したとして，その中で最長の実行時間を最悪実行時間（Worst-Case Execution Time：WCET）と呼ぶ．

文献 [1] および [4] では，WCET の見積もりについて詳細が書かれている．

### 2.2.4 リアルタイムスケジューリングアルゴリズム

単一プロセッサにおけるリアルタイムスケジューリングは従来から広く研究されており，様々な手法が提案されている．一般に，リアルタイムシステムでは優先度に基づくスケジューリングが行われるが，タスクの優先度を決定する方法には主に次の 2 種類がある．

- 設計者が事前に静的優先度を設定する
- タスクの属性に応じて優先度を決定する

前者については議論の余地は少ないが，後者については実際に数多くのアルゴリズムが提案されており，条件によっては最適性が証明されているものも存在する．”スケジューリングアルゴリズム”と呼ぶことが多いが，正確にはタスクの優先度付け方法である．ここでは，その代表的なものを紹介する．

- 静的優先度方式
  静的優先度方式ではタスク間の優先順位を基準にしてスケジューリングが行われる．タスク間の優先順位はアプリケーション開発者が各タスクの重要度に合わせ，あらかじめ段階的に決定した優先度を基準として，次のように決定される．実行可能なタスクが複数ある場合，その中で最も優先度の高いタスクが実行される．異なる優先度を持つタスクの優先順位は，高い優先度を持つタスクの優先順位が高くなる．等しい優先度のタスクの場合は FIFO（First In First Out）形式が採られる．
  図 2.4 に動作例を示す．この図では，塗りつぶしの部分が実際の実行している箇所となっている．

  始めにタスク 2 が起動され実行される．タスク 2 の実行中にタスク 1 が起動されるが，タスク 1 のほうが優先度が高いために，実行権が変わる．タスク 2 からタスク 1 に実行が切り替わり，今まで実行されていたタスク 2 が実行状態から実行可能状態に移行される．タスク 1 に実行が移り，タスク 1 の実行が終了する．その間にタスク 3 が起動されるが，優先度が低いため実行可能状態で待機する．タスク 1 の実行が終了した時，タスク 2 およびタスク 3 が実行可能状態にいるが，優先度の高いタスク 2 が先に実行される．タスク 2 の実行が終了した時点で，タスク 3 の実行が開始される．
  上述のように静的優先度方式については，高優先度のタスクを実行するスケジューリングポリシーである．

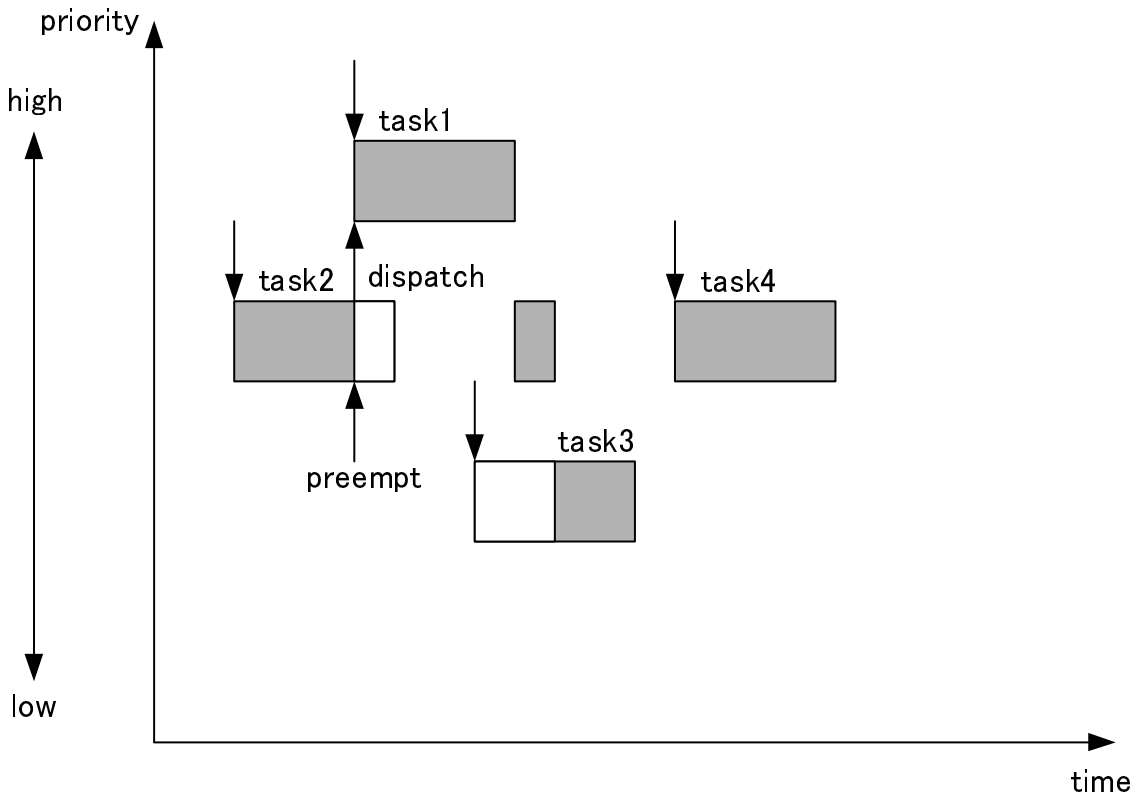


図 2.4: 静的優先度方式

- RM（Rate Monotonic）[5]
  周期タスク用のアルゴリズムであり，周期と相対デッドラインが等しいという条件の下，周期の短いタスクの優先度を高く設定する．他の条件として，タスクは互いに独立で順序制約がないこと，資源制約がないこと，プリエンプティブであることが必要である．
  RM 法を用いてスケジュールできない周期タスクセットは，他のアルゴリズムを用いてもスケジュールできないことが証明されており，その意味で最適である．なお，周期は静的に決定されているという観点から，RM は静的優先度方式の 1 つとしてみなされる．

- EDF（Earliest Deadline First）[5]
  絶対デッドラインの早いタスクに高い優先度を与える方法. Deadline Driven Scheduling とも呼ばれる．非周期タスクも扱える．前提条件は，タスクは互いに独立で順序制約がないこと，資源制約がないこと，そしてプリエンプティブであることである．
  スケジュール可能なタスクセットであれば必ず EDF でスケジュールできること，最大遅れ時間を最小にすることがそれぞれ証明されており，最適スケジューリングであるといえる．ただし，各タスクの実行時間を考慮に入れないため，将棋倒し的

にデッドラインオーバーを起こす可能性がある．

- LLF（Least Laxity First）[6]
  余裕時間の短いタスクに高い優先度を与える方法．前提とする条件は，EDF 法の場合と同じである．
  LLF 法も最適スケジューリングであると言われているが，余裕時間は時々刻々と変化するので，厳密に実現しようとするとスケジューリングやコンテキスト切り替えが頻発しオーバヘッドが大きくなるという欠点がある．

- DM（Deadline Monotonic）[7]
  RM 法を拡張した方法であり，周期ではなく相対デッドラインに基づいてタスク優先度を決める方法．RM 法では周期と相対デッドラインが等しいという前提条件であったが，実際のシステムでは次の起動要求時刻より先にデッドラインを迎えることがある．

- FCFS（First Come First Service）
  タスクを到着順に処理していく方法．リアルタイムシステムのためのスケジューリング方法というわけではないので，デッドラインについては考慮されていない．μ ITRON で静的優先度に基づくスケジューリングが行われていて，同じ優先度を持つタスクが複数個到着したとき，FCFS で実行される．

- 適応型動的優先度方式（ADP，Adaptive Dynamic Priority）[1]
  静的優先度を基準とし，これに動的要因を導入することによって優先度を更新する優先度付け方法．アプリケーション開発者が各タスクの重要度，および実行順序を考慮して設定した値は重要であるため，静的優先度を基準と考え，動的要因として，タスクの余裕時間と周期タスクの場合の周期を取り入れている．LLF に従い，余裕時間に応じて優先度を高めに変更する．動的優先度 $Pri_{dynamic}$ は次式で表される[9]．

$$Pri_{dynamic} = Pri_{static} - (\frac{\gamma}{T} + \frac{\sigma}{L})$$

$Pri_{dynamic}$：動的優先度
$Pri_{static}$：静的優先度
T：周期
L：余裕時間
$\gamma$,$\sigma$：定数

ADP 法では余裕時間の見積もりに関し，WCET ではなく PET（Predictive execution time）を用いる．PET とは，タスクの実際の実行時間に基づき動的に変動する見積もり時間である．各タスクはシステム起動後に複数回実行される可能性があ

[9] $Pri_{dynamic}$，$Pri_{static}$ は値が小さいほど優先度が高い．

るため，実行毎にタスクの実行時間に基づいてPETの再計算を行い，その結果が次回の実行時間の見積もりに反映される．この計算はタスクの終了時に行われ，終了したタスクの実行時間と前回のPETの値の加重平均から算出する．これにより，経験的に実際の実行時間を反映するPETの値を得る．あるタスクのi回目の実行終了時のPETの再計算は以下の式で与えられる．ここで，$\beta$は定数である．

$$PET_i = \beta \times PET_{i-1} + (1 - \beta) \times (measured\ exec.\ time)$$

文献[1]の結果より，本研究における実装では$\beta = 0.8$とする．

## 2.3 スケジューラ

本節では，実装するオペレーティングシステムにおけるスケジューラの構成について述べる．

### 2.3.1 スケジューラの構成

本研究でターゲットとしている$\mu$ITRONでは，タスクの管理方法として，実行可能状態のタスクを優先度順でキュー[10]に繋げて管理をしている．静的優先度方式では，優先度順に繋がれているレディキューを順に走査し，優先度の一番高いキューに繋がれているタスクの先頭を次に実行するタスクと決定している．

実装するスケジューラは静的優先度方式，適応型動的優先度方式を関数で用意し，基本は静的優先度方式を使用し，部分的に適応型動的優先度方式を使用する．レディキューの構造については変更することがない構成となっている．構造を変えないことにより，複数のスケジューラを容易に変更できる．

適応型動的優先度方式では，タスクの優先度計算におけるオーバヘッドが大きくなる可能性がある．また，全てのパターン[11]において優先度計算を行った場合，実行可能状態のタスク数が多い場合にオーバヘッドが過大になる．

オーバヘッドを削減する方法として以下の方法を提案する．

1. スケジューラは複数のサービスコールから呼び出されるが，適応型動的優先度方式については自タスクの終了（ext_tsk）のみに使用する．
2. 全てのタスクについて優先度計算を行うのではなく，レディキューに繋がれている優先度の高いタスク上位n個を対象とし，優先度計算にかかるオーバヘッドを削減する．
3. 優先度1[12]のレディキューに関しては，適応型動的優先度方式のみが利用する構成を採る．

[10] このキューのことをレディキューと呼ぶ．

[11] ここでいうパターンとは，サービスコールにおけるタスクの起動／終了時などから呼び出されるスケジューラのことである．

[12] 最高優先度．

1 について，スケジューラがサービスコールから呼び出されるタイミングとは，現在のタスクよりも優先度の高いタスクが起動された，割込みにより優先度の高いタスクが起動された，優先度の高いタスクが待ち解除された，優先度の高いタスクがタイムアウトした，現在実行中のタスクが終了した時である．全てのタスクに対して動的優先度の計算を行わないので，オーバヘッドの削減が期待できる．

2 での上位 n 個については，実装では n の値は 10 としている．実行可能状態のタスクが多い状況で実験した結果，n が上位 10 個以上の場合だとオーバヘッドによりデッドラインオーバーを起こす状況が増加したため，この値とした．

3 の構成にすることにより，デッドラインオーバーが近いタスクを一時的に最高優先度にすることにより，ミスを防ぐことが可能となる．しかし，優先度 1 のレディキューに対しても FIFO で行うと，デッドラインが近くて優先度の高いタスクが，自分よりも優先度の低いタスクを実行している間にデッドラインオーバーを起こす可能性がある．そのため，優先度 1 のレディキューに関しては起動時の優先度順につなぐ．

図 2.5 に提案手法でのレディキューの構造を示す．

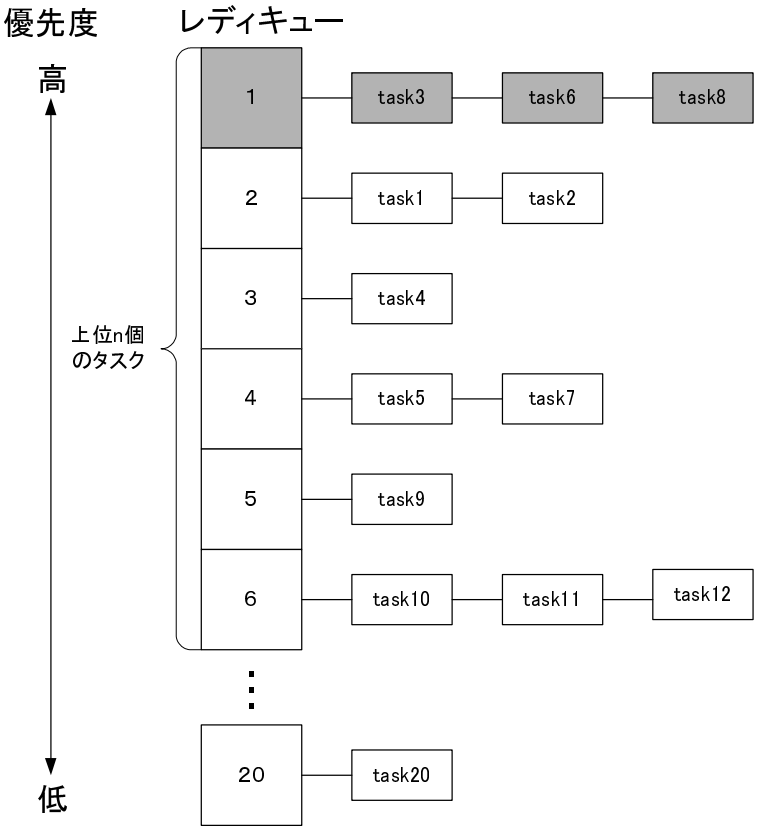


図 2.5: 提案手法でのレディキュー構造

塗りつぶしの箇所が優先度 1（最高優先度）のキューである．この状態ではタスク 3 がスケジューラによって次に実行するタスクとなる．また，動的優先度の計算対象となるタ

スクは，キューにつながれている上位 10 個のタスクが計算対象となる．この図では，優先度 2 のキューにつながれているタスク 2 までが動的優先度の計算対象となるタスクである．

### 2.3.2 優先度更新方法

適応型動的優先度方式では，動的優先度計算のために毎回除算を行う．しかし，タスクの動的優先度の再計算時に毎回除算を用いることは，オーバヘッドの増大につながる．この除算のオーバヘッドを削減するために，余裕時間の値に従って離散的な幅値が設定された表を用意し，このテーブルから値を得る方法を採る．周期に対する優先度更新の除算に関しては，周期が静的に決定しているので静的に更新値を計算しておくことで，実行時に除算を行わない方法を採る．このように，先に利用できる値を用意しておくことにより計算にかかるコストを省略する．

余裕時間の値に従う離散値は，タスクの実余裕時間が短くなるほど優先度を大きく更新し，逆にタスクの実行に余裕があればほとんどあるいはまったく更新しないような形をとる．実装では以下の方針に従う．

- 優先度の更新幅は最大 $2^p$ 段階．
- タスクセットのタスクの平均実行時間を c とする．
- タスクの実余裕時間を $l$ とする
- $2^{p+1-q}$ を変更幅とする．ここで，$q$ は $1 \leq q$，$(q-1) \cdot c \quad \leq l < q \cdot c$ となる整数値である．

実余裕時間の残りとタスク平均実行時間の割合いに応じて，2 のべき乗で動的優先度が修正される方法である．上記の更新法は以下の方針からなる．例えばスケジューリング時にあるタスクの実余裕時間が平均実行サイクル c 未満のときに他のタスクに実行権を与えた結果，前者のタスクは後に実行権が戻ってきた場合にはデッドラインオーバーとなる可能性が高い．このような場合には，このタスクは時間効率の面からは最優先タスクとし，デッドラインを超える前に終了させることが望ましい．そこで，上記方法の最後の項目に従い優先度を $2^p$ 段階上昇させることにより，最優先タスクとなり得る．$2^p$ は優先度の最大更新幅であり，これを Maximum Rising Priority（MRP）という．MRP を適切に設定することにより，実余裕時間が極端に短くなった場合に優先度の向上が過大となることを防ぐ．この様子を表したものが図 2.6 である．

文献 [1] によると，MRP を変動させ，15 個のタスクからなるタスクセットを実行した結果，MRP は 16 が最適となっている．本研究における実装においても，この値に従っている．

周期による優先度更新の効果は，2.2.4 節の式中の $\gamma$ の値を変化させることで変更可能

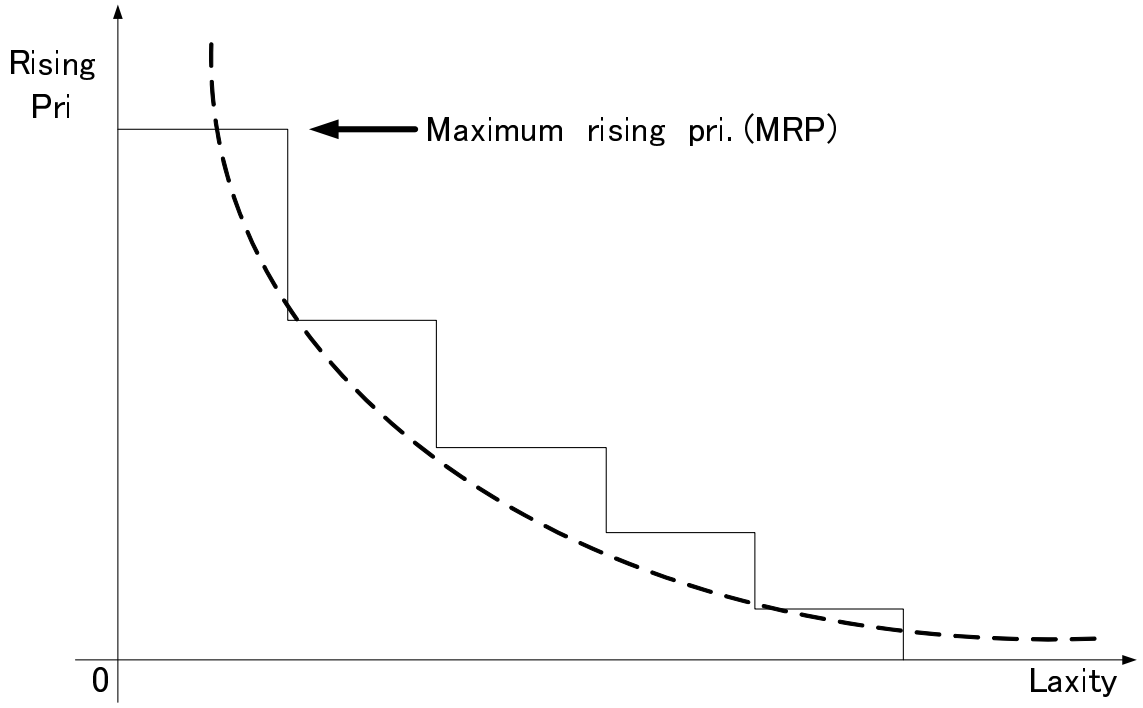


図 2.6: 実余裕時間に応じた優先度更新値

である．文献 [1] では，$\gamma$ の値を過大にすると周期タスクの過度の優先度の上昇を引き起こすため，周期のみによって実行順序が決定されるスケジューリングとなり，これは RM 法に他ならない．$\gamma$ の値が最小周期の 4 倍のとき十分ミスタスク数が減少するという結果から，実装でもこの値をとる．

## 2.4 組込みプロセッサ

### 2.4.1 高速割込み応答機構

本研究で使用する CPU(Casablanca[2]) は，複数の割込み専用レジスタセットを内蔵し，割込み発生時にアクティブなレジスタセットをハードウェアにより自動的に切り替えて使用することにより，多重の割込みに対して高速な応答を可能にする．これはマルチコンテキストアーキテクチャ[8] を割込み処理のコンテキストに応用したものである．表 2.1 に 8 つのレジスタセットを内蔵した場合のそれぞれのレジスタセットモードの用途を示す．

現在実行中のスレッドが使用するレジスタセットはプロセッサ状態レジスタ（PSR）内の”CRSP（current register-set pointer）”フィールドが指定する．以前の実行スレッドを”PRSP（Previous register-set pointer）”フィールドが指定する．CRSP の実体は 1 つであるが，PRSP はレジスタセット毎に独立して存在する．

| CRSP | モード | 用途 |
|---|---|---|
| 0 | Normal exec | 通常の実行時 |
| 1 | Internal trap | 内部トラップ（例外）ハンドラの実行時 |
| 2~7 | Interrupt | 外部割込みハンドラ実行時 |

表 2.1: レジスタセットモード

図 2.7 にレジスタセット 3 を使用する割込みを処理中に，レジスタセット 2 を使用する優先度のより高い割込みを受けた場合のアクティブなレジスタセットの切り替えの様子を示す．図中の実線が前者の割込み処理時，点線が後者の割込み処理時を表している．後者の割込みを受けたとき，CRSP の値は 3 から 2 に変化し，レジスタセット 2 に対応する PRSP に 3 がセットされる．後者の割込み処理が終了する際には，リターントラップ命令によって PRSP の値（3）が CRSP にセットされ，前者の割込み処理に復帰する．

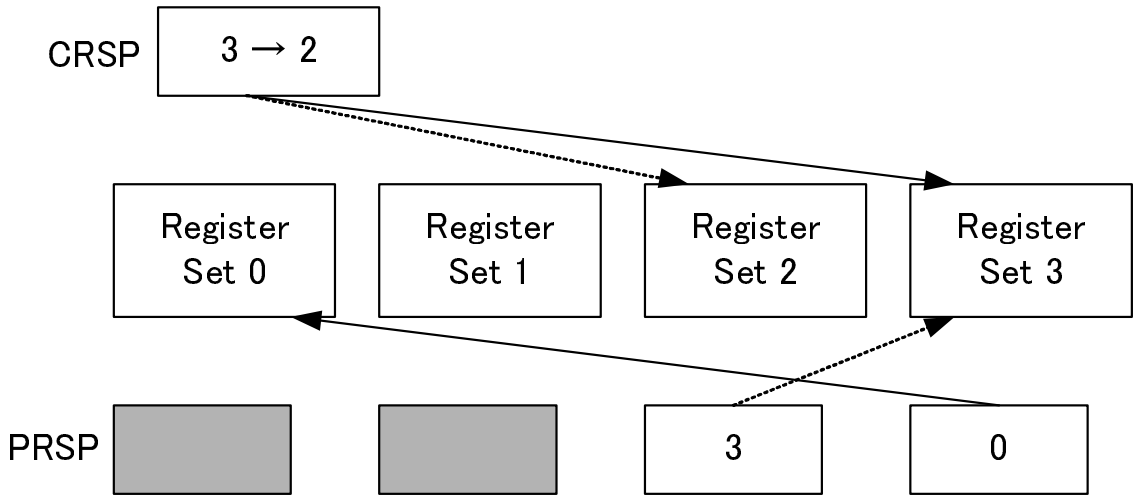


図 2.7: 割込み発生時のレジスタセット切り替え

プロセッサ状態レジスタ（PSR）内の一部のフィールド，プログラムカウンタ（PC），SPARC の Y レジスタなどはプロセッサコンテキストの一部として扱う必要があり，これらについてレジスタセットと同様に多重化し，CRSP の値にしたがって切り替えて使用する．このような切り替え機構により，同一レジスタセットを使用する割込みを多重に受け付けない方針で使用する限り，コンテキストを退避する必要はない．同一レジスタセットを使用する割込みを多重に処理する場合は，そのレジスタセットに限り従来の退避／復帰の方式で使用する．

### 2.4.2 データキャッシュ制御機構

従来の組込み用 CPU は，ハードウェアコスト削減のためにキャッシュにスヌープ機構は採用していなかった．このことから，外部メモリを介して機器/周辺回路とデータの授受を行う場合は，キャッシュを介さない処理，すなわちノンキャッシュブルなメモリアクセスを行うため大きなオーバヘッドを生じ，DMA 機構を使用する際に非効率である．本研究で使用する CPU は組込み用 CPU におけるこの反省点から，専用の命令実行により，キャッシュと外部メモリ間のコンシステンシを保つことを可能としている．この命令を使用するライブラリを提供することで，タスクと割込みハンドラとの間で高速なデータの授受が可能となる．この専用命令を以下に示す．

- FF-Load（強制キャッシュfill）
  - キャッシュアクセスにおいて，ヒット/ミスにかかわらず外部メモリからデータを読み込み，キャッシュにフィルインする命令．したがって，キャッシュ内に当該アドレスのラインが存在していた場合は上書きされる．
- FU-Store（強制キャッシュupdate）
  - キャッシュアクセスにおいて，当該アドレスがヒットかつそのラインが更新（dirty）されていた場合に，外部メモリにライトバックする命令．なお，ストア命令内の当該ラインの状態は clean に移行する．

# 第3章　$\mu$ITRON4.0インターフェースでの実装方法

本章では，実装したリアルタイムオペレーティングシステムの全体構成と実装方法について示す．

## 3.1　リアルタイムオペレーティングシステムの構成

本研究で実装したリアルタイム OS の主な機能を以下に示す．

1. マルチタスク機構
2. タスク間同期・通信機能
3. 割込み管理
4. 時間管理

マルチタスク機構は，複数のタスクを同時に実行させる環境を提供するものであり，いわば CPU の仮想化／多重化を行うものである．OS は，複数あるタスクに一定の規則[1]にしたがって CPU の使用権を与え，マルチタスク環境を実現している．

一つの目的を複数のタスクで実現する場合，タスク間で同期（待ち合わせ）や通信を行う場合がある．この手段を提供するのが，タスク間同期・通信機能である．

割込み管理は，CPU の割込みとユーザープログラムが用意した割込みハンドラとを対応させる役割を担うものである．

リアルタイムシステムは，時間と密接に関係するため，一定の時間経過を待つなどの機能を提供する．

上記を実現する各種サービスコールの実装において，それらのインターフェースは $\mu$ITRON4.0 仕様 [3] に従った．以下に $\mu$ITRON4.0 仕様に準拠するために必要な最低限の機能を記す．

1. タスクを生成できること．タスクは少なくとも，実行状態，実行可能状態，休止状態の 3 つの状態を持つこと．
2. $\mu$ITRON4.0 仕様のスケジューリング規則に従ったタスクスケジューリングを行うこと．ただし，優先度毎のタスクを 1 つに制限することや，優先度を 1 段階に制限する

[1] これをスケジューリングポリシーという．

ことは許される．

3. 割込みハンドラ（または割込みサービスルーチン）を登録できること．
4. タスクおよび割込みハンドラ（または割込みサービスルーチン）から，タスクを起動する（休止状態から実行できる状態にする）手段が用意されていること．
5. 自タスクを終了する（実行状態から休止状態にする）手段が用意されていること．

実装では 1，3，4，5 を満たしている．2 に関してはスケジューラを適応型動的優先度方式に置き換えを行っている部分が存在するため，完全に $\mu$ITRON4.0 仕様に準拠しているとはいえない．

実装で $\mu$ITRON4.0 仕様に従うことは，アプリケーション開発効率を考慮し，インターフェースを標準化したものを扱うという意味である．実際の構成方法などは仕様とは異なっているが，サービスコールを使用する際などに開発者が構成を気にすることはなく問題はない．

### 3.1.1 タスク管理

ITRON 仕様では，並行処理するプログラムの単位をタスクと呼ぶ．タスクを管理するための主な管理情報はタスクコントロールブロック (TCB) に格納される．以下に実装で定義された TCB の内容を示す．

```
typedef struct task_control_block {
    QUEUE       tasi_queue;     /* タスクキュー */
    ID          task_id;        /* タスク ID */
    ATR         tskatr;         /* タスク属性 */
    FP          task;           /* タスクの起動番地 */
    PRI         itskpri;        /* タスクの起動時優先度 */
    INT         stksz;          /* スタック領域のサイズ */
    VP          stk;            /* スタック領域の先頭番地 */
    PRI         tskpri;         /* 現在優先度 */
    UINT        tstat;          /* タスクの状態 */
    UINT        tskwait;        /* タスクの待ち要因 */
    ID          wobjid;         /* 待ち対象オブジェクトの ID */
    TMO         lefttmo;        /* タイムアウトする時間 */
    UINT        actcnt;         /* 起動要求キューイング数 */
    UINT        wupcnt;         /* 起床要求キューイング数 */
    UINT        suscnt;         /* 強制待ち要求ネスト数 */
    BOOL        enatex;         /* タスク例外処理許可状態 */
    UINT        texptn;         /* 保留例外要因 */
    T_MSG       *pk_msg;        /* メールボックス受信用 */
```

```
    T_CTXB      tskctxb;        /* タスクコンテキストブロック(sp,pc,npc) */
    BOOL        dyn_flg;        /* 実行中を判断するフラグ */
    INT         period_pri      /* 周期タスクの優先度更新値 */
    TIME        exe_time;       /* 予測実行時間(PET) */
    TIME        deadline;       /* 絶対デッドライン */
    TIME        period;         /* 周期(非周期=0)
    TIME        cam_time;       /* 累積実行時間 */
    TIME        wcet;           /* 最悪実行時間 */
}TCB;


  typedef struct queue{
     struct queue *next;       /* 次エントリへのポインタ */
     struct queue *prev;       /* 前エントリへのポインタ */
     TCB          *myself;     /* 自分参照用 */
  }QUEUE;

  typedef struct task_context_block{
     VP       sp;          /* スタックポインタ */
     FP       pc;          /* プログラムカウンタ */
     FP       npc;         /* ネクストプログラムカウンタ */
  }T_CTXB;

  typedef struct t_msg{
     struct t_msg   *next;              /* 次のメッセージへのポインタ */
     VB             msgcont[MSGS]       /* メッセージの内容 */
  }T_MSG;
```

タスク管理機能は，タスクの状態を直接的に操作/参照するための機能である．タスクを生成/削除する機能，タスクを起動/終了する機能，タスクに対する起動要求をキャンセルする機能，タスクの優先度を変更する機能，タスクの状態を参照する機能が含まれる．タスクはID番号で識別されるオブジェクト[2]である．タスクのID番号をタスクIDと呼ぶ．

タスクは，実行順序を制御するために，起動時優先度と現在優先度を持つ．単にタスクの優先度といった場合には，タスクの現在優先度を指す．タスクは実行中に優先度を変更するサービスコール（chg_pri）によって優先度を変更可能であるが，そのタスクの起動時および実行が終了する時に起動時の優先度に初期化される．

タスクに対する起動要求は，キューイングされる．すなわち，すでに起動されているタ

[2]タスクなどサービスコールの操作対象となるものを総称してオブジェクトと呼ぶ．

スクを更に起動すると，その記録が残り，後でそのタスクが終了した時にタスクを自動的に再起動する．起動要求のキューイングのために，TCB は起動要求キューイング数を持つ．タスクの起動要求キューイング数は，タスクの生成時に 0 にクリアされる．

カーネルは，タスクの終了時にタスクが獲得した資源[3]を解放する処理を行わない．タスク終了時に資源を解放するのはアプリケーションの責任である．これらの操作に関しては，全て TCB の情報によって行われる．

タスクが生成されると，タスクの情報を持つ TCB がタスク ID 順にテーブルに登録される．act_tsk(task_id) により起動されたタスクは，レディキューに優先度順に繋がれる．

レディキューに繋がれたタスクは，スケジューラによって選択されてから実行する．実行が終了したタスクに関しては，ext_tsk() などにより終了処理を行う．タスクの終了時に行う主な処理として，レディキューからの削除，TCB のメンバの初期化処理，実行時間の加重平均をとるための計算処理，起動要求がキューイングされていた場合の処理，次に実行するタスクを決めるためのスケジューリング処理などが挙げられる．

タスクの起動要求をキャンセルする処理は，TCB のメンバにある起動要求カウント数（actcnt）を 0 に初期化するのみである．タスクの優先度を変更する処理は，レディキュー中のタスクの現在の優先度を変更し，レディキューから一度削除し，再度新しい優先度でレディキューに繋ぐ．タスクの状態を参照する処理は，現在の TCB の状態を表示する．

### 実行タスクが存在しない場合の振舞い

組込みシステムであるとしても，常にシステム機能を実現するためのタスクが動作しているわけではない．このため，システムの負荷が下がると実行可能状態のタスクがない，すなわちレディキューに TCB がなくなる状況がありえる．

この場合の OS の動作として，次の二つの方法が考えられる．

一つは，次にカレントタスクにすべきタスクがない場合は，たとえば CPU の停止処理命令を実行し，次の割込みを待つ．

もう一つは，レディキューからタスクがなくならないようにする方法が考えられる．これは，最低優先度に何もしない無限ループのタスクを OS で用意しておくものである．こうすることで，レディキューからタスクがなくなることはないので，タスクスケジューリング処理でレディキューに TCB がない状況を想定しなくてもよい．言い換えると，待ちに入らないタスクが存在していればよいという意味である．

実装では後者の方を選択している．OS 側が何もしないタスク[4]を用意し，実際に実行するタスクがなくなった場合は，このアイドルタスクを実行する構成を採る．

[3] セマフォ資源，メモリブロックなど．
[4] これをアイドルタスクとよぶ．

### 3.1.2 サービスコール

実装したサービスコールを表 3.1 に示す[5].

| | |
|---|---|
| タスク管理機能 | |
| act_tsk | タスクの起動 |
| iact_tsk | タスクの起動（非タスクコンテキスト用） |
| can_act | タスク起動要求のキャンセル |
| ext_tsk | 自タスクの終了 |
| ter_tsk | タスクの強制終了 |
| chg_pri | タスク優先度の変更 |
| get_pri | タスク優先度の参照 |
| ref_tsk | タスクの状態参照 |
| ref_tst | タスクの状態参照（簡易版） |
| タスク付属同期機能 | |
| slp_tsk | 起床待ち |
| tslp_tsk | 起床待ち（タイムアウトあり） |
| wup_tsk | タスクの起床 |
| iwup_tsk | タスクの起床（非タスクコンテキスト用） |
| can_wup | タスク起床要求のキャンセル |
| rel_wai | 待ち状態の強制解除 |
| irel_wai | 待ち状態の強制解除（非タスクコンテキスト用） |
| sus_tsk | 強制待ち状態への移行 |
| rsm_tsk | 強制待ち状態からの再開 |
| frsm_tsk | 強制待ち状態からの強制再開 |
| dly_tsk | 自タスクの遅延 |
| 同期・通信機能 | |
| セマフォ | |
| sig_sem | セマフォ資源の返却 |
| isig_sem | セマフォ資源の返却（非タスクコンテキスト用） |
| wai_sem | セマフォ資源の獲得 |
| pol_sem | セマフォ資源の獲得（ポーリング） |
| twai_sem | セマフォ資源の獲得（タイムアウトあり） |
| イベントフラグ | |
| set_flg | イベントフラグのセット |
| iset_flg | イベントフラグのセット（非タスクコンテキスト用） |
| clr_flg | イベントフラグのクリア |
| wai_flg | イベントフラグ待ち |
| pol_flg | イベントフラグ待ち（ポーリング） |
| twai_flg | イベントフラグ待ち（タイムアウトあり） |
| メールボックス | |
| snd_mbx | メールボックスへの送信 |
| rcv_mbx | メールボックスからの受信 |
| prcv_mbx | メールボックスからの受信（ポーリング） |
| trcv_mbx | メールボックスからの受信（タイムアウトあり） |
| メモリプール管理機能 | |
| 固定長メモリプール | |
| get_mpf | 固定長メモリブロックの獲得 |
| pget_mpf | 固定長メモリブロックの獲得（ポーリング） |
| tget_mpf | 固定長メモリブロックの獲得（タイムアウトあり） |
| 時間管理機能 | |
| システム時刻管理 | |
| get_tim | システム時刻の参照 |
| システム状態管理機能 | |
| loc_cpu | CPU ロック状態への移行 |
| iloc_cpu | CPU ロック状態への移行（非タスクコンテキスト用） |
| unl_cpu | CPU ロック状態の解除 |
| iunl_cpu | CPU ロック状態の解除（非タスクコンテキスト用） |

表 3.1 実装したサービスコール一覧

[5]内容の詳細は [3] に記されている.

実装/実装外の機能に関して以下に示す.

- タスク管理機能
  sta_tsk,exd_tsk に関しては未実装である．sta_tsk は，タスクの起動をするサービスコールであるが，$\mu$ITRON3.0 仕様との互換性のために存在するものなので，実装外とする．
  exd_tsk は，自タスクの終了と削除を行うサービスコールである．削除とは，タスクの状態を未登録状態に移行させることである．タスクの状態で未登録状態を実装しないため，実装外とする．タスク管理機能のその他のサービスコールは全て実装対象とする．

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

- 同期/通信機能
  同期/通信機能は，タスクとは独立したオブジェクトにより，タスク間の同期/通信を行うための機能である．セマフォ，イベントフラグ，データキュー，メールボックスの各機能が含まれる．
  セマフォ，イベントフラグ，データキュー，メールボックスの中で，データキュー以外は実装対象である．

- 拡張同期/通信機能
  拡張同期/通信機能は，タスクとは独立したオブジェクトにより，タスク間の高度な同期/通信を行うための機能である．ミューテックス，メッセージバッファ，ランデブの各機能が含まれる．
  拡張同期/通信機能に関しては，本研究では実装対象外とする．

- メモリプール管理機能
  メモリプール管理機能は，ソフトウェアによって動的なメモリ管理を行うための機能である．固定長メモリプール，可変長メモリプールの各機能が含まれる．
  固定長メモリプールのみ実装対象とする．

- 時間管理機能
  時間管理機能は，時間に依存した処理を行うための機能である．システム時刻管理機能，周期ハンドラ，アラームハンドラ，オーバランハンドラの各機能が含まれる．周期ハンドラ，アラームハンドラ，オーバランハンドラを総称して，タイムイ

ベントハンドラと呼ぶ．システム時刻管理機能の一つである，システム時刻の参照 (get_tim(TIME *p_systim)) のみを実装対象とする．これは，内部トラップにより実装されている．

- システム状態管理機能

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

  システム状態管理で実装対象とするのは，CPU ロック状態への移行/解除する機能である．システムは，CPU ロック状態か解除状態かのいずれかの状態をとる．CPU ロック状態では，カーネルの管理を除くすべての割込みが禁止され，ディスパッチも起こらない．また，割込みが禁止されていることから，タイムイベントハンドラの起動[6]も保留される．CPU ロック状態は，実行中の処理の優先順位が，他のいずれの処理よりも高くなった状態とみなすこともできる．

  CPU ロック状態に移行することを「CPU ロックする」，CPU ロック解除状態にいこうすることを「CPU ロックを解除する」ともいう．

- 割込み管理機能

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

  割込みに関しては，サービスコールとして提供しない．割込みハンドラは事前に登録されており，割込み処理としてはタスクの起動のみを行う．

- サービスコール管理機能

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

  サービスコール管理機能は実装対象外とする．

- システム構成管理機能

  システム構成管理機能には，CPU 例外ハンドラを定義する機能，システムのコンフィギュレーション情報やバージョン情報を参照する機能，初期化ルーチンを定義する機能が含まれる．

  システム構成管理機能は実装対象外とする．

[6]本研究では実装対象外である．

## 3.2 評価環境の構成

評価環境の全体構成を図 3.1 に示す.

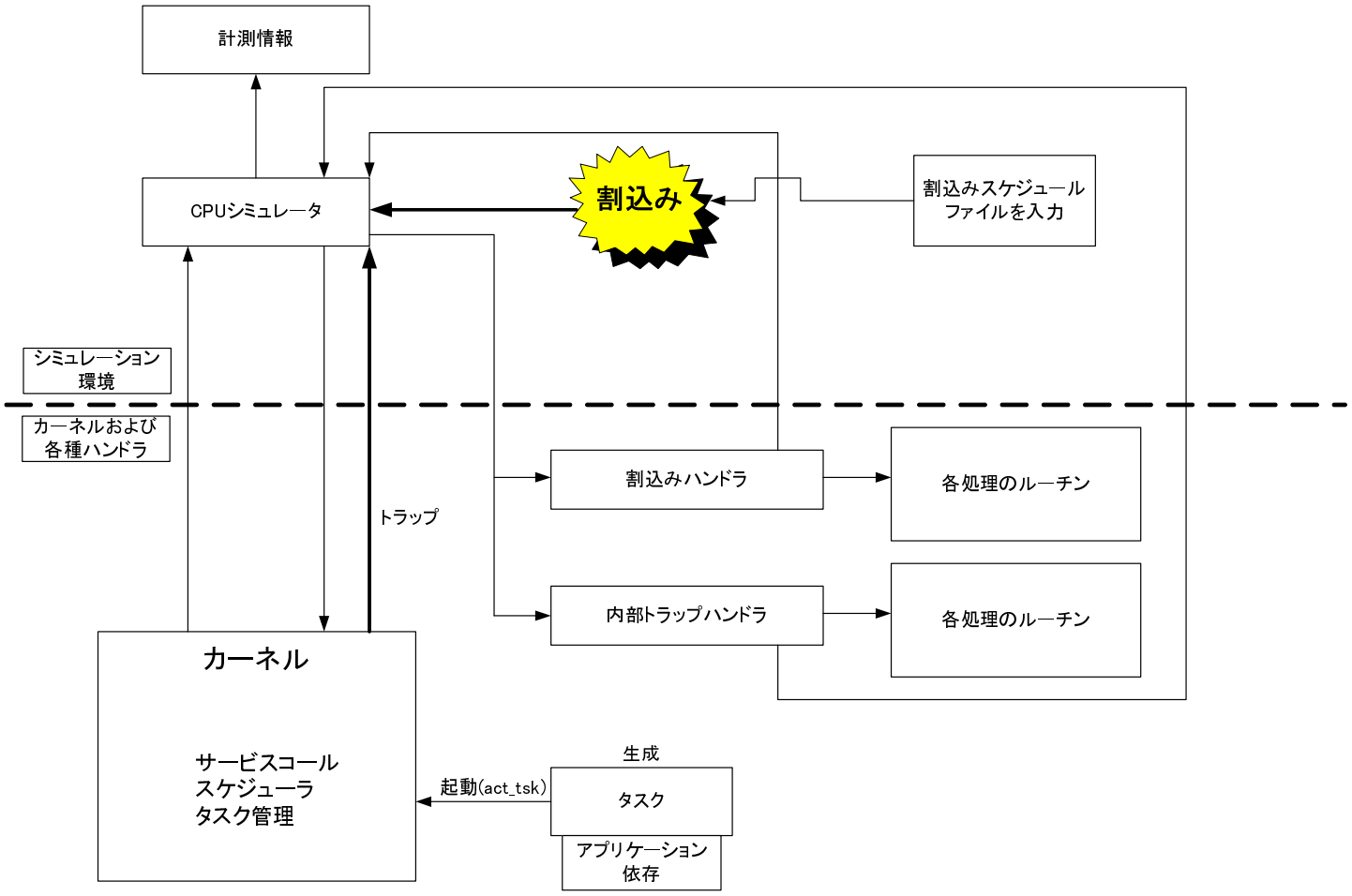


図 3.1: 評価環境

図の点線より上部はシミュレーション環境であり，下部はカーネルおよび各種ハンドラルーチンである．評価する環境として CPU シミュレータを使用しているが，下部の部分は変更することなく，他のターゲットマシンで動作することが可能となる．これにより，現実的な評価を行うことが可能である．

タスクは静的に生成されてから起動し，実行をする．タスクの実行中にトラップが発生すると，トラップの命令を読み込んだ CPU シミュレータはトラップハンドラを実行する．ハンドラの中で，どのトラップなのかを判断し，そのトラップに対しての処理を行うルーチンを実行する．割込みが発生すると，CPU シミュレータは割込みハンドラを実行する．ハンドラでは割込み信号線から読み込んだ割込みに対して行うべき処理を判断し，その処理のルーチンを実行する．割込みの発生方法は，割込みスケジューリングファイルによ

り，周期割込み，非周期割込みを記述する．詳細については後述する．

最初にシステムの初期化処理を行う．図 3.2 でシステム初期化処理の項目を示す．

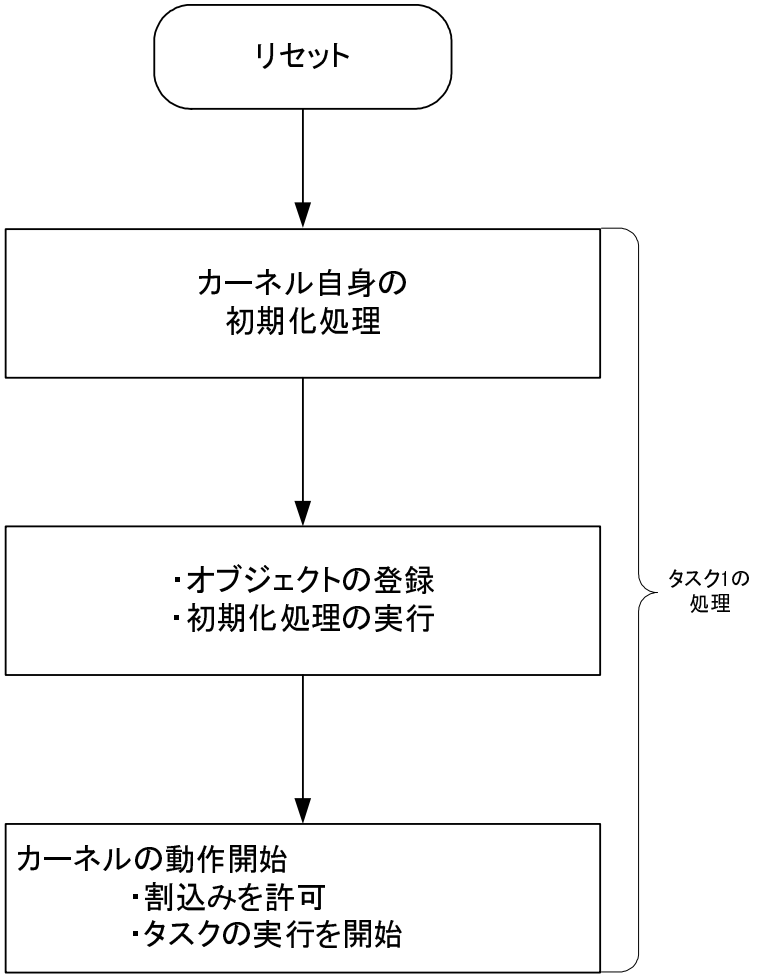


図 3.2: システム初期化手順

タスク 1 が最初の初期化処理を行う構成を採る．このタスクが実行中は他のタスクは起動されることが許されず，かつ割込みが発生することも許されない．このタスク 1 で図の処理を全て行う．

カーネル自身の初期化処理として，CPU シミュレータの仮想的なアドレスの割り当て処理，タスクの情報を持つ TCB（Task Control Block）やレディキューの初期化処理などを行う．

オブジェクトの登録は，メールボックスやセマフォ，イベントフラグ，固定長メモリプールなどの登録を行い，それぞれに対して初期化処理を行う．

最後にカーネルの動作を開始する．全ての登録および初期化処理が終了したら，割込みを許可し，他のタスクを実行を起動する．その後，システムの初期化処理を行っていたタ

スク 1 は終了され，再度起動されることはない．$\mu$ITRON では，カーネルやソフトウェア部品[7]の構成やオブジェクトの初期状態を定義するためのシステムコンフィギュレーションファイルを作成する．システムコンフィギュレーションファイルには，カーネルやソフトウェア部品の静的 API と ITRON 共通静的 API [8]に加えて，C 言語処理系のプリプロセッサディレクティブを記述することが可能となっている．システムコンフィギュレーションファイル中の静的 API を解釈して，カーネルやソフトウェア部品を構成するためのツール（コンフィギュレータ）を使用する．これらの詳細については $\mu$ITRON4.0 仕様書 [3] に記されている．

実装では，コンフィギュレータの作成は省略し，コンフィギュレータで行うべき処理を全てタスク 1 で処理している．

実際に実行される際，図 3.1 でのカーネルおよび各種ハンドラ，タスクの処理ルーチンは全て一つのバイナリファイルとして CPU シミュレータに渡される．この様子を図 3.3 に示す．

TCB を登録するテーブル，レディキュー，メールボックスを登録するテーブル，セマフォを登録するテーブル，イベントフラグを登録するテーブル，固定長メモリプールを登録するテーブル，タイムアウト待ちのタスクを登録するキューなどが，CPU シミュレータの仮想的なアドレスに割り当てられる．この割り当てに関しても全てタスク 1 の初期化処理にて行われる．

タスクを管理する TCB やレディキューは，カーネル初期化処理の際に初期化が行われる．オブジェクトの登録はセマフォ，メールボックス，イベントフラグ，固定長メモリプールなどが登録される．これらは全て，CPU シミュレータの仮想的なアドレス空間に割り当てられる．実装ではこのアドレス空間に割り当てる処理を静的に行っている．TCB はテーブルを持ち，このテーブルにタスクがタスク ID 順に登録されている．レディキューはキュー優先度 1 ～ 20 までのキューを用意しており，タスクが起動されるとそれぞれの優先度に TCB がつながれていく構成になっている．セマフォやメールボックスなどのオブジェクトに関しては，それぞれの管理ブロックがテーブルに登録されている．例えば，メールボックスの場合はメールボックスコントロールブロックがテーブルに登録され，メールボックス 1 を使用する場合は，テーブル 1 を指定してメッセージの送受信を行う．

CPU シミュレータの仮想的なアドレス空間に割り当てられた TCB，レディキュー，オブジェクトなどを図 3.4 に示す．

TCB のテーブルは 0x12000 ～ 0x20000 番地の間に割り当てられるれる．レディキューは 0x20000 ～ 0x30000 番地の間に割り当てられ，メールボックスは 0x30000 ～ 0x31000，セマフォは 0x31000 ～ 0x32000，イベントフラグは 0x32000 ～ 0x33000，固定長メモリプールは 0x33000 ～ 0x34000，タイムアウト待ちキューは 0x34000 ～ 0x35000 の間に割り当てられる．タイムアウト待ちキューには，タスクが何らかの処理を待つような状態で，その待つ時間が限られているタスクが繋げられる．

[7] ミドルウェアのこと．

[8] 静的 API とは，システムコンフィギュレーションファイル中に記述し，カーネルやソフトウェア部品の構成を決定したり，オブジェクトの初期状態を定義するためのインタフェースのことである．

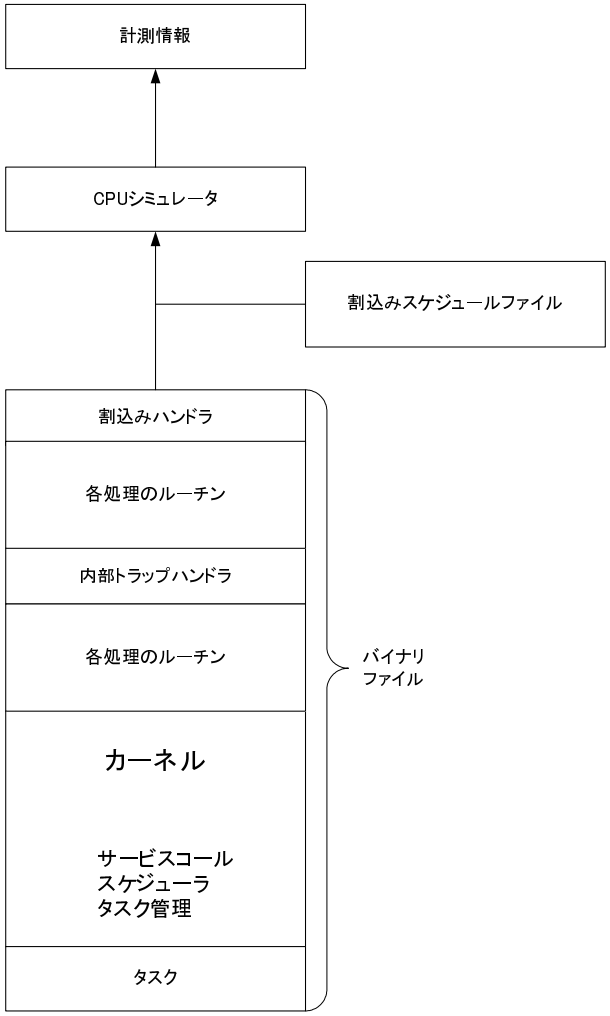


図 3.3: 一つのバイナリファイルとして CPU シミュレータに渡される様子

TCB（タスク）のスタック領域は 0x10003000 から 0x1000 ずつ割り当てられる．固定長メモリプールはメールボックスのみに使用するため，種類は一である．ここでは固定長メモリプールを一つ獲得すると，16byte のメモリ領域を獲得する．しかし，必要な場合は，領域サイズを指定してメモリブロックを獲得することは可能である．

g_runtsk，g_schedtsk に関して，g_runtsk は実行中の TCB を指し，g_schedtsk はスケジューラにより次に実行するタスクを決定した際に，その TCB を指す．

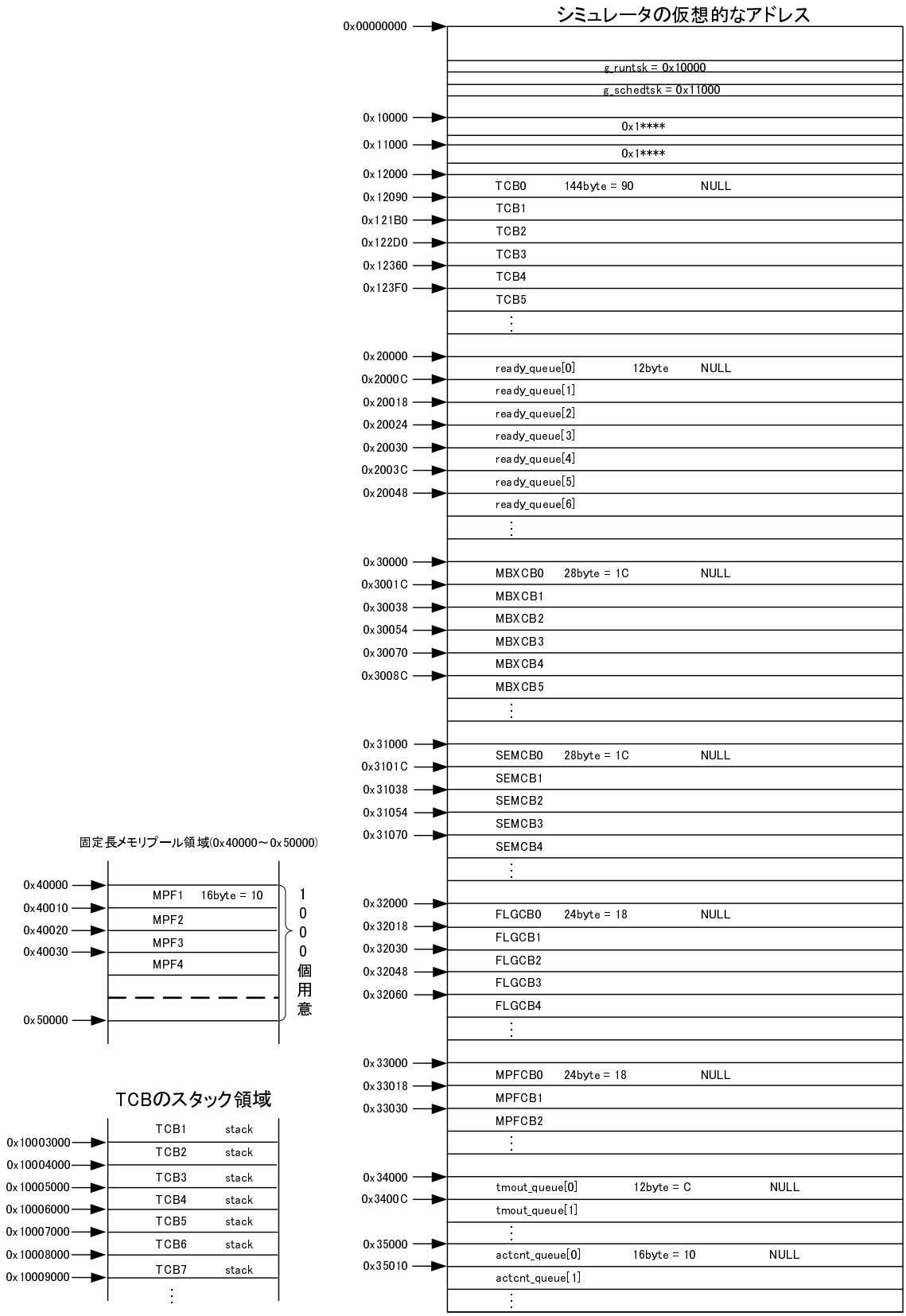


図 3.4: CPU シミュレータの仮想的なアドレス空間

## 3.3 内部トラップ

トラップ命令はC言語の関数の中で呼ばれる．関数の中ではインラインアセンブラで，それぞれのトラップに対応する命令が記述されている[9]．トラップの番号により，処理する内容が決まっている．トラップ番号は8bitで指定することができ[10]，最高で128パターンのトラップ処理を記述することが可能となっている．表3.1にそれぞれのトラップに関する処理内容を示す．

| No | trap |
|---|---|
| 0 | dispatch |
| 1 | loc_cpu |
| 2 | unl_cpu |
| 3 | get_tim |
| 4 ～ 8 | reserved |
| 9 | deadline_check |
| 10 | get_period |
| 11 | get_th |
| 12 ～ | reserved |

表 3.1: トラップ割り当て

トラップ0の処理はディスパッチャとなっている．トラップ1はCPUのロック（割込み禁止），トラップ2はCPUのアンロック（割込み許可），トラップ3は現在の時間を得る処理となっている．トラップの4～8に関しては未定義である．トラップ9はタスクがデッドラインを超えたかどうかの判断，トラップ10はタスクの周期を得る処理，トラップ11はタスクが現在何回目の実行を行っているのかを判断する処理となっている．

ディスパッチャでは，現在実行中のタスクのプログラムカウンタ，ネクストプログラムカウンタ，スタックポインタをTCBに保存する．次にレジスタのsaveを行い，次に実行するタスクのレジスタをrestoreする．タスクの実行終了時にディスパッチャが呼ばれたか，割込みによりディスパッチャが呼ばれたかを判断する必要がある．これに関しては，TCBのメンバにあるdyn_flgのビットをにより判断する．このフラグ動作を図3.5に示す．

saveでフラグの値が0の場合はレジスタのsaveは行わない．フラグが1の時は，レジスタのsave処理を行う．restoreでフラグの値が0の場合は，次に実行するタスクが初期状態なので，フラグに1をセットする．フラグが1の場合は，実行途中のタスクの状態レジスタをrestoreする．また，ext_tsk()で自タスクの終了が行われる際に，プログラムカウンタ，ネクストプログラムカウンタ，スタックポインタなどを初期化しておく．このフ

[9]本研究におけるシミュレーション環境では，SPARC Architecture Version 8[9]を命令セットとするCPUシミュレータを作成し，使用した．

[10]最上位1ビットは1に固定されている．

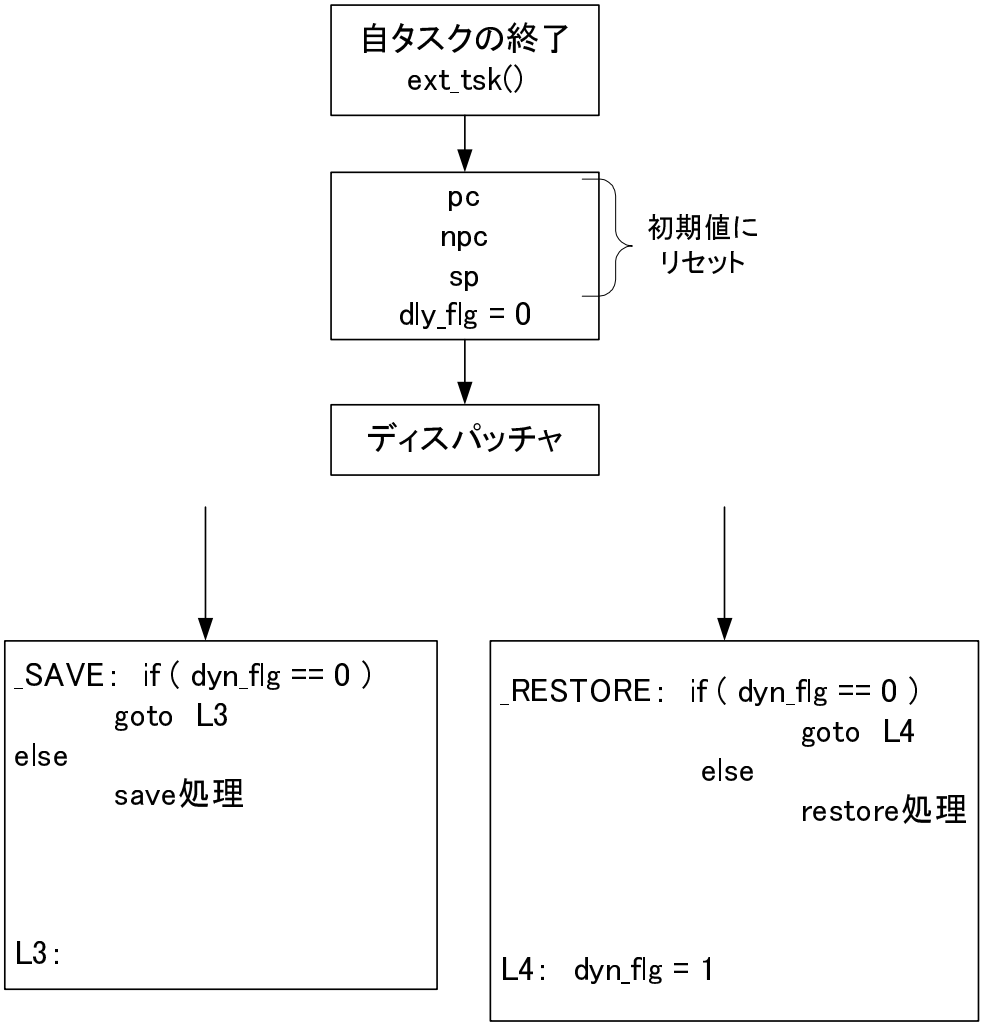


図 3.5: フラグ動作

ラグにより，プリエンプトされたタスクに対してもコンテキストの退避/復帰に関して正常に動作する．

タスクが実行途中に割込みによりタスクの切り替えが発生し，実行タスクが切り替わる．切り替わったタスクの実行が終了し，実行途中だったタスクに戻る時は，TCB のネクストプログラムカウンタの値から再開する．

loc_cpu は，割込みマスクを 15 にセットしている．トラップが通常通り発生するには，プロセッサ・ステータス・レジスタ (Processor State Register：PSR) の ET(enable traps) ビットが 1 でなければならない．割込みには許可ビットとマスクフィールドが用意されている．許可ビットとは，割込みの受け付けを許可するか否かを指定するビットである．割込み発生時に新たな割込みを受け付けないようにする機構は，この許可ビットを自動的に受け付け禁止に設定することで実現している．一方，マスクフィールドとは，割込みを割

込み優先度によってマスクするためのフィールドである．

実装する CPU では割込み優先度の最大値が 15 であり，マスクの値を 15 とすることにより，全ての割込みを禁止することが可能である． トラップ 9 ～ 11 については，全てトラップハンドラはすぐにリターンする処理をしているだけである．これらは評価のために使用するトラップであり，CPU シミュレータが返り値のためのレジスタ（%o0）に値を入れる．

## 3.4 割込み

割込みスケジュールファイルを CPU シミュレータに与え，外部割込を発生させる．割込みの処理により，指定されたタスクを起動させる．割込みスケジュールファイルの例を表 3.2 に示す．

| 物理線 | タスク ID | 周期/非周期 | 周期 | period |
|---|---|---|---|---|
| 1 | 2 | 1 | 1000000 | 4 |
| 1 | 3 | 1 | 1000000 | 4 |
| 1 | 4 | 1 | 1500000 | 2 |
| 1 | 5 | 1 | 2000000 | 2 |
| 1 | 6 | 1 | 2500000 | 1 |
| 1 | 7 | 1 | 3000000 | 1 |
| 1 | 8 | 1 | 6000000 | 0 |
| 1 | 9 | 1 | 12000000 | 0 |
| 1 | 10 | 0 | 12000000 | 0 |
| 1 | 11 | 0 | 12000000 | 0 |
| 1 | 12 | 0 | 12000000 | 0 |
| 1 | 13 | 0 | 12000000 | 0 |
| 1 | 14 | 0 | 12000000 | 0 |
| 1 | 15 | 0 | 12000000 | 0 |
| 1 | 16 | 0 | 12000000 | 0 |
| 1 | 17 | 0 | 12000000 | 0 |
| 1 | 18 | 0 | 12000000 | 0 |
| 1 | 19 | 0 | 12000000 | 0 |

表 3.2: 割込みスケジュールファイル

割込みの物理線は 15 本あるが，この例では 1 本のみ使用している．割込みスケジュールファイルでは，最初に物理線を指定し，次に起動するタスクの ID 番号，そのタスクが

周期タスクか非周期タスクかを指定，そのタスクの周期クロック[11]，周期タスクの場合は適応型動的優先度方式のスケジューラで優先度をいくつ上げるのかを静的に指定する．それぞれはタブで区切られたファイルになっている．

指定された周期の時間になると割込みを発生させ，CPU シミュレータが割込みハンドラを起動する．

この場合のタスクの起動は非タスクコンテキスト，すなわり割込みルーチンから呼ばれるため，iact_tsk サービスコールを使用する．スケジューラ，ディスパッチャに関しても割込み専用のものを用意している．この割込み専用スケジューラでは，次に実行するタスクの選定を行うのみで，スケジューラからディスパッチャが呼ばれることはない．次に実行するタスクの選定が行われた後は，割込みハンドラ側で，割込み専用のディスパッチャを呼ぶかどうかを判断する．

[11]非周期の場合は，指定したクロックのみに起動される．

# 第4章　評価

本章では実装したスケジューラの評価を示す．評価環境は第 3 章で述べた構成である．

## 4.1　シミュレーションモデル

カーネルおよび各種ハンドラをバイナリファイルとして入力する CPU シミュレータを作成した．CPU シミュレータの命令セットは SPARC Architecture Version8[9] とした．

命令，データキャッシュはそれぞれ 4 ウェイセットアソシアティブでトータル 16KB である．キャッシュのブロックサイズを 32byte として 1 ブロックを 8 ワードで構成する．キャッシュヒット時は 1 サイクルで読み書き可能であるが，ミス時のペナルティは 10 サイクルとした．乗算，除算によるペナルティサイクルはそれぞれ 3 サイクル，10 サイクルとした．今回の計測では二次キャッシュは用意していない．

## 4.2　動的優先度計算によるオーバヘッド

本研究で実装した適応型動的優先度方式では，動的優先度を計算するためのオバーヘッドが存在する．このオーバヘッドを，正確に計測する．

タスクの最大個数を 20 とし，優先度の最大値を 20 とし，それぞれの優先度キューにタスクが繋がれる状況で計測を行った．

計測対象として，全てのタスクを動的優先度計算の対象とするものと，上位 5 個，上位 10 個のタスクを動的優先度計算の対象とするものに分た．また，その中で周期と余裕時間に関する除算計算を行うが，それぞれ除算をしない方針[1]でのオーバヘッドをあわせて示す．これにより，除算を行わないことによるオーバヘッドの削減が評価可能である．以下に計測項目を示す．また，結果を表 4.1 に示す．なお，オーバヘッドはクロックサイクル単位となっている．

[1]周期に関する除算を行わない場合は，周期に関する優先度上昇を静的に与えておく．余裕時間に関する除算を行わない場合は，第 3 章で述べたように優先度上昇度の表を使用する．

1. 全てのタスクを計算対象とし，全てのタスクで除算を行う.
2. 全てのタスクを計算対象とし，周期の除算を行わず，余裕時間の除算は行う.
3. 全てのタスクを計算対象とし，余裕時間の除算を行わず，周期の除算は行う.
4. 全てのタスクを計算対象とし，除算を行わない.
5. 上位 10 個のタスクを計算対象とし，全てのタスクで除算を行う.
6. 上位 10 個のタスクを計算対象とし，周期の除算を行わず，余裕時間の除算は行う.
7. 上位 10 個のタスクを計算対象とし，余裕時間の除算を行わず，周期の除算は行う.
8. 上位 10 個のタスクを計算対象とし，除算を行わない.
9. 上位 5 個のタスクを計算対象とし，全てのタスクで除算を行う.
10. 上位 5 個のタスクを計算対象とし，周期の除算を行わず，余裕時間の除算は行う.
11. 上位 5 個のタスクを計算対象とし，余裕時間の除算を行わず，周期の除算は行う.
12. 上位 5 個のタスクを計算対象とし，除算を行わない.

| 項目 | 計測項目 | overhead |
|---|---|---|
| 1 | 全タスク:除算有り | 41,061 |
| 2 | 全タスク:周期除算なし | 23,727 |
| 3 | 全タスク:余裕時間除算なし | 24,433 |
| 4 | 全タスク:除算なし | 7,399 |
| 5 | 上位 10 個タスク:除算有り | 23,583 |
| 6 | 上位 10 個タスク:周期除算なし | 14,005 |
| 7 | 上位 10 個タスク:余裕時間除算なし | 14,409 |
| 8 | 上位 10 個タスク:除算なし | 3,906 |
| 9 | 上位 5 個タスク:除算有り | 14,439 |
| 10 | 上位 5 個タスク:周期除算なし | 8,663 |
| 11 | 上位 5 個タスク:余裕時間除算なし | 9,133 |
| 12 | 上位 5 個タスク:除算なし | 2,688 |

表 4.1: 適応型動的優先度方式でのオーバヘッド計測

次に，従来手法である静的優先度方式でのオーバヘッドを示す.

静的優先度方式では，最高優先度のタスクを次に実行するタスクと決定するスケジューリング方式なので，レディキューの最低優先度のレディキューまで走査した時がオーバヘッドが一番大きくなる．表 4.2 に，最低優先度のレディキューを走査したオーバヘッドおよび，最高優先度を走査した際のオーバヘッドを示す.（最低優先度は 20.）また，中間地点として，優先度 10 のレディキューを走査した際のオーバヘッドを示す.

| 項目 | overhead |
|---|---|
| 最高優先度 | 271 |
| 中間地点（レディキュー 10） | 627 |
| 最低優先度 | 1,087 |

表 4.2: 静的優先度方式でのオーバヘッド計測

**考察**

表 4.1 から分かるように，除算を行う場合はオーバヘッドが大きい．除算を行わず動的優先度の計算を行うことにより，オバーヘッドを軽減することができる．最大で 33,662 クロックのオーバヘッドを削減することができた．

全タスクの場合でも，除算を行わない場合は十分に小さいオーバヘッドとなっているが，今回ではタスク数の最大が 20 となっているため，上位 n 個としている場合との差が大きくない．20 個の範囲ならば，動的優先度計算によるオーバヘッドの影響でタスクがデッドラインオーバーを起こすケースは少ない．しかし，タスク数が 100 個を超えるような場合には，動的優先度計算のオーバヘッドにより，タスクがデッドラインオーバーを起こすケースが出てくる．

上記の結果から，タスク数が n 個以上実行可能状態となっている場合に，上位 n 個のタスクを動的優先度計算の対象とすることで，オーバヘッドを削減することができる．また，除算なしで実行することにより，更にオーバヘッドの削減が可能である．

## 4.3 静的優先度方式との比較

静的優先度方式との比較として，次の二つの項目を評価した．

- 周期タスクを数回起動した際に，いくつかの周期タスクの起動が衝突するポイントで非周期タスクを起動させる．
- 周期タスクが存在している中で，非周期タスクをランダムに起動させる．

前者を計測 1 と呼び，後者を計測 2 と呼ぶ．

### 4.3.1 計測 1

計測 1 では周期タスク 5 個，非周期タスク 10 個を実行する．それぞれの起動クロック（周期）を表 4.3 に示す．

| task_id | 周期/非周期 | 起動 clock |
|---|---|---|
| 2 | 周期タスク | 1,000,000 |
| 3 | 周期タスク | 1,000,000 |
| 4 | 周期タスク | 2,000,000 |
| 5 | 周期タスク | 3,000,000 |
| 6 | 周期タスク | 6,000,000 |
| 10 | 非周期タスク | 3,000,000 |
| 11 | 非周期タスク | 3,000,000 |
| 12 | 非周期タスク | 3,000,000 |
| 13 | 非周期タスク | 3,000,000 |
| 14 | 非周期タスク | 3,000,000 |
| 15 | 非周期タスク | 12,000,000 |
| 16 | 非周期タスク | 12,000,000 |
| 17 | 非周期タスク | 12,000,000 |
| 18 | 非周期タスク | 12,000,000 |
| 19 | 非周期タスク | 12,000,000 |

表 4.3: 計測 1 での起動クロック

表 4.3 の動作例を図 4.1 に示す.

タスク 2 ～ 5 が 3,000,000 クロックで起動される．そのクロックに非周期タスクを 5 個起動する．タスク 2 ～ 6 が 12,000,000 クロックで起動される．そのクロックに非周期タスクを 5 個起動する．周期が短いものほど起動時優先度が高く設定されている．また，周期タスクの実行時間は周期の約 10%としており，タスクの起動回数により実行時間が長くなる．すなわち，一つの周期タスクの実行時間は，起動される毎に長くなり，毎回実行時間が異なる周期タスクとした.

周期タスクの起動が重なるクロックに非周期タスクを複数個起動させることにより，デッドラインオーバーを起こすタスクセットとした．これにより，静的優先度方式と適応型動的優先度方式での比較を行った．また，適応型動的優先度方式で上位 10 個のタスクを動的優先度更新の対象とした場合と全てのタスクを動的優先度の計算対象とした場合を評価た.

また，タスクセットの変更は行わず，周期タスクの実行時間を変更することにより，CPU 使用率を変動させた．CPU 使用率に対して，適応型動的優先度方式の有効性を示す.

CPU 使用率に対して，タスクのトータルデッドラインオーバー数を図 4.2 に示し，CPU 使用率に対してのデッドラインオーバーしたタスクの起動時優先度の平均を図 4.3 に示す.

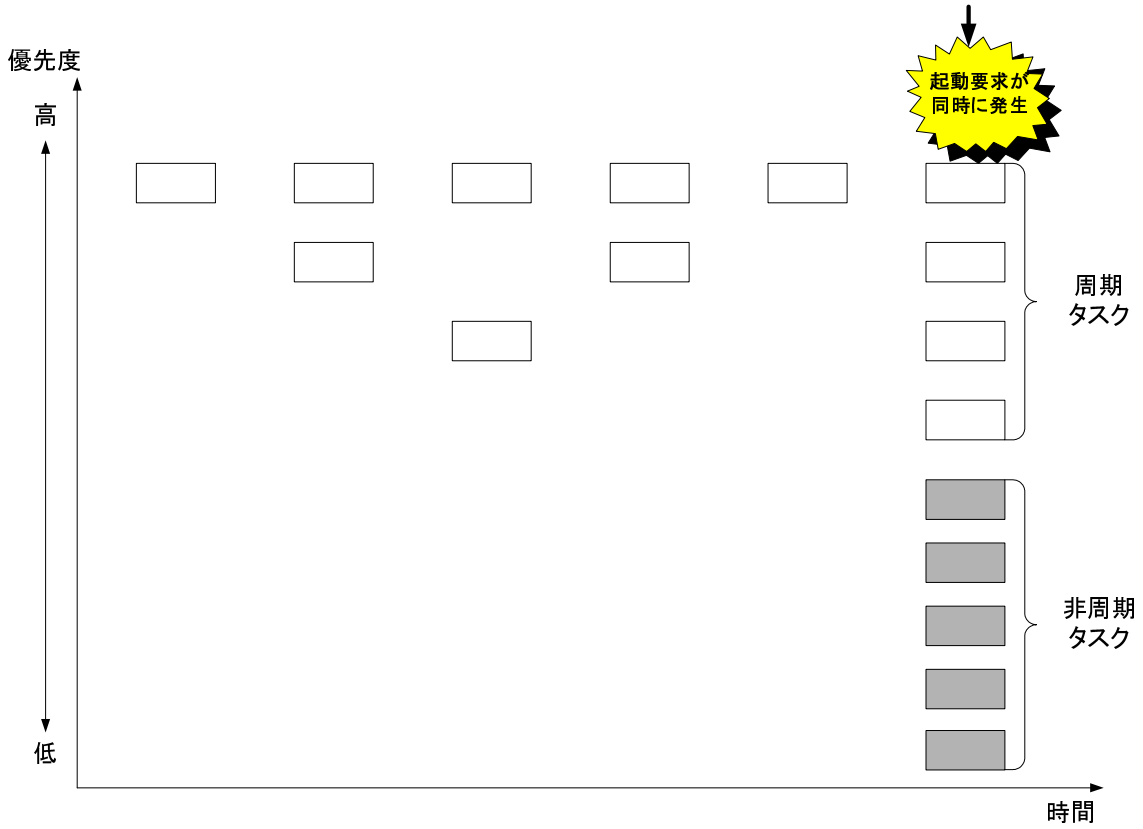


図 4.1: 計測 1 の動作

図 4.2 の TDOC_S は静的優先度方式でのトータルデッドラインオーバー数で，TDOC_D は適応型動的優先度方式での動的優先度計算の対象タスクが上位 10 個の場合のトータルデッドラインオーバー数である．また，TDOC_Dall は，動的優先度計算の対象タスクが全ての場合である．

図 4.3 の AVE_DOP_S は静的優先度方式でのデッドラインオーバーしたタスクの起動時優先度の平均値で，AVE_DOP_D は適応型動的優先度方式でのデッドラインオーバーしたタスクの起動時優先度の平均値で，上位 10 個を動的優先度計算の対象とした場合である．また，AVE_DOP_Dall は，適応型動的優先度方式でのデッドラインオーバーしたタスクの起動時優先度の平均値で，動的優先度計算の対象が全てのタスクの場合である．

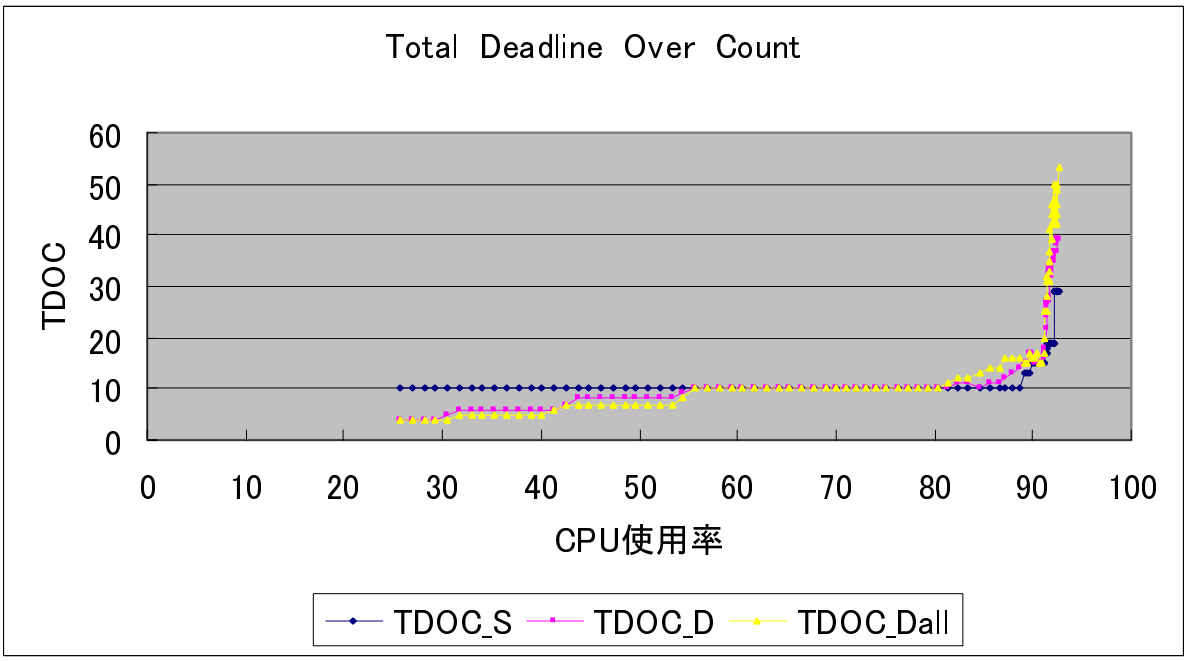


図 4.2: 計測 1-1

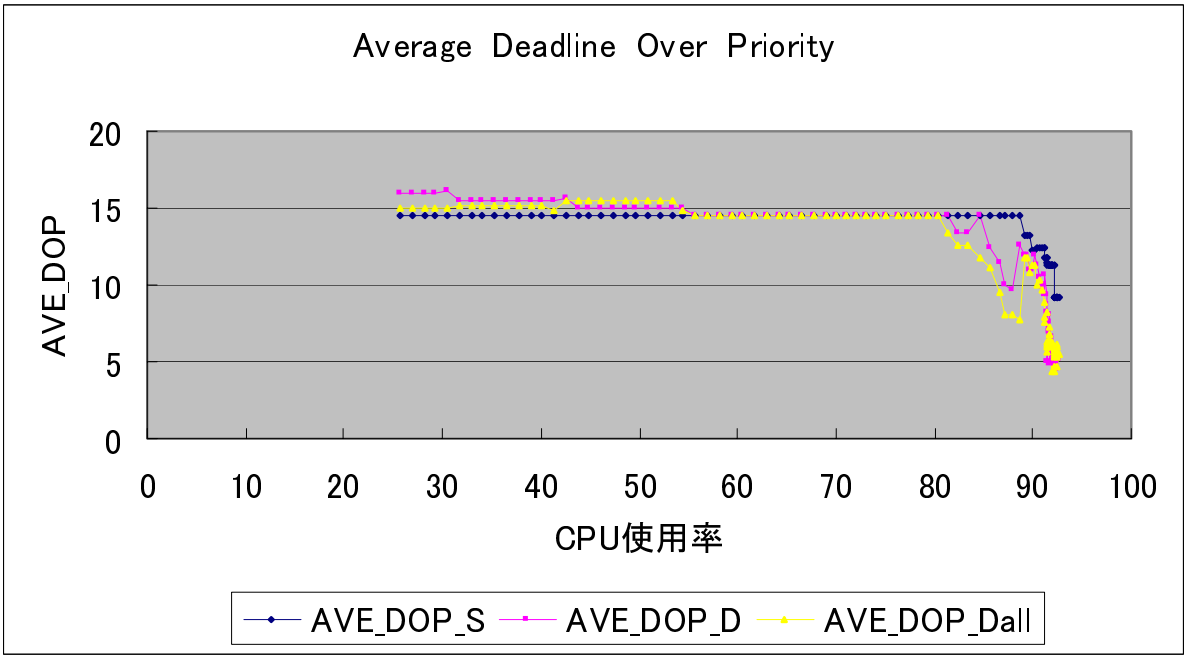


図 4.3: 計測 1-2

### 考察

図 4.2 では，適応型動的優先度方式で CPU 使用率が 55%以下の範囲で静的優先度方式よりもデッドラインオーバー数を減らしている．これは，周期タスクの余裕時間の間に，非周期タスクの優先度を動的に上げることにより，複数のタスクのデッドラインオーバーを回避しているためである．局所的に非周期タスクを起動しているため，55%以上は非周期タスクが全てデッドラインオーバーをしている．

82%付近から周期タスクもデッドラインオーバーを起こし始めている．これはスケジューラのオーバヘッドが影響している．CPU 使用率が 82%を超えてから，スケジューリング困難に陥り，デッドラインオーバーを起こすタスクが増加している．しかし，CPU 使用率が 82%以上のシステムは既に破綻状態となっているシステムとみなす事ができる．そのような状況でタスクを救うことを目的としていないため，CPU 使用率 80%以下の状況を評価対象とする．表 4.4 にそれぞれのトータルデッドラインオーバー数を示す．

| スケジューラ | 全体 | 80%以上 | 80%以下 |
|---|---|---|---|
| TDOC_S | 1,458 | 983 | 470 |
| TDOC_D | 1,738 | 1,355 | 383 |
| TDOC_Dall | 1,987 | 1,623 | 364 |

表 4.4: デッドラインオーバー数

全体では静的優先度方式がデッドラインオーバー数の合計が少なくなっている．これは，80%以上から動的優先度方式のデッドラインオーバー数が増加しているためである．評価では 80%以下をターゲットとしているため，そこに注目する．TDOC_D は TDOC_S に比べて 87 個のタスクを救っている．TDOC_Dall は TDOC_S に比べて 106 個のタスクを救っている．80%以下の状況で静的優先度方式と比較すると，適応型動的優先度方式を使用することにより，タスクのデッドラインオーバー数が削減されることが示された．

図 4.3 において，適応型動的優先度方式の場合，CPU 使用率が 82%以上からデッドラインオーバーしたタスクの平均値が下がっている．平均値が下がるということは，優先度の高いタスクがデッドラインオーバーしたことを意味する．スケジューラのオーバヘッドや，優先度の低いタスクを救った代償として，優先度の高いタスクもデッドラインオーバーを起こしている．

### 4.3.2 計測 2

計測 2 では，周期タスク 5 個，非周期タスク 50 個を実行した．計測 2 の動作例を図 4.4 に示す．

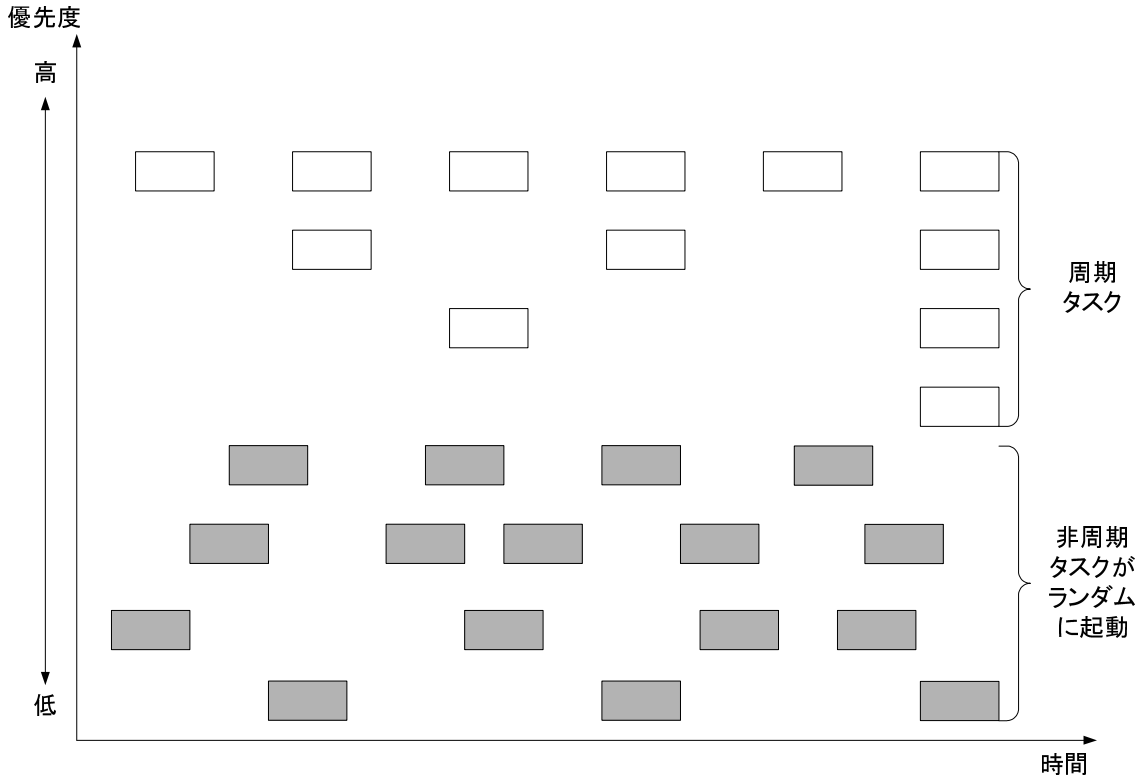


図 4.4: 計測 2 の動作

非周期タスクに関しては，事前に擬似乱数で生成されたクロックにそれぞれのタスクを起動させた．計測 1 と同様に，CPU 使用率に対してのデッドラインオーバー数を評価する．異なる点は非周期タスクがランダムに起動していることである．図 4.5 に CPU 使用率に対する，タスクのトータルデッドラインオーバー数を示し，CPU 使用率に対してのデッドラインオーバーしたタスクの起動時優先度の平均を図 4.6 に示す．

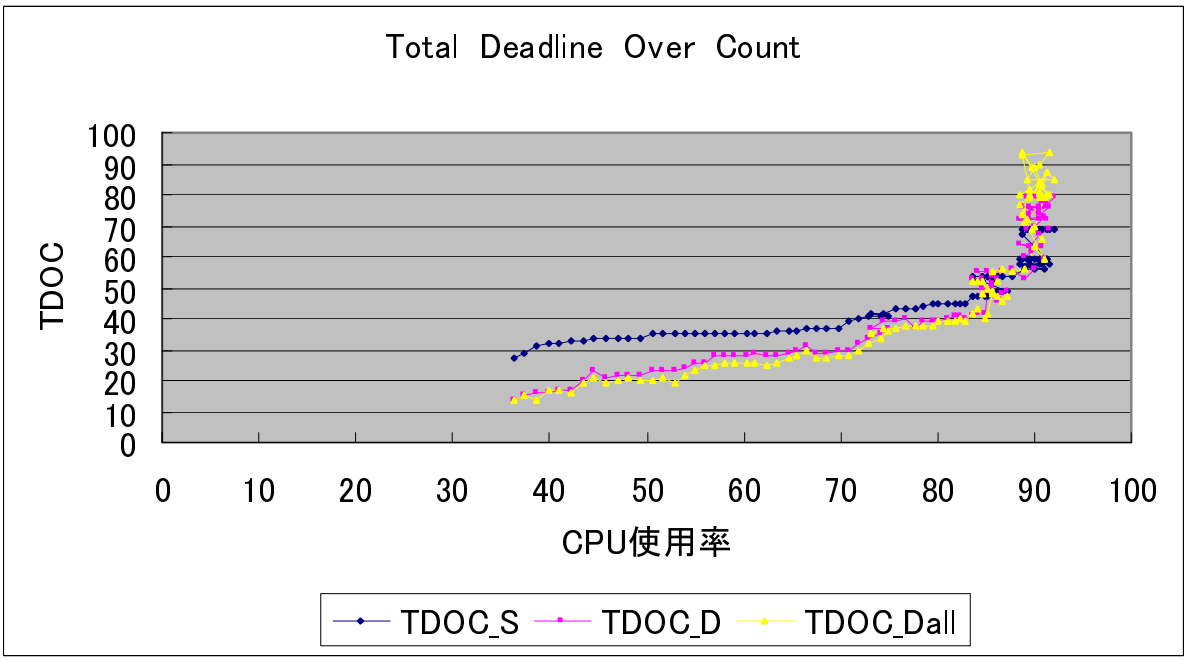


図 4.5: 計測 2-1

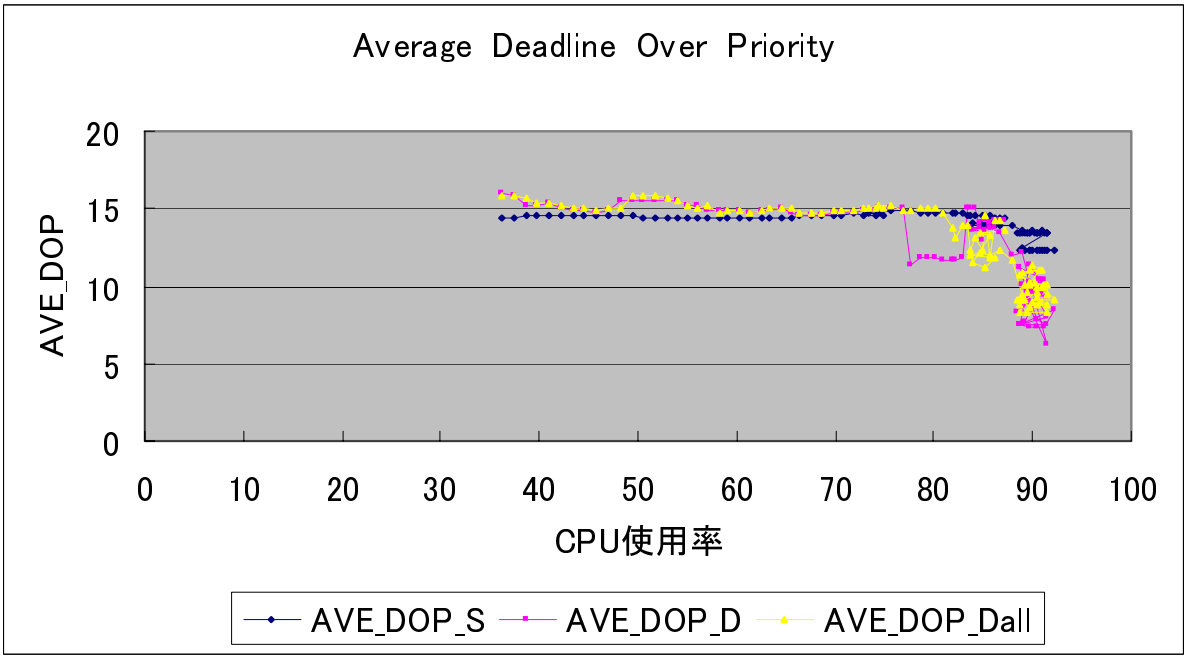


図 4.6: 計測 2-2

**考察**

図 4.5 において，適応型動的優先度方式が CPU 使用率が 84%以下の範囲で静的優先度方式よりもデッドラインオーバー数を削減している．84%以下では，TDOC_D よりも TDOC_Dall のほうがデッドラインオーバー数が少ない．これは，動的優先度のオーバヘッドがタスクの実行に与える影響が小さいためである．84%を超えると，スケジューリングのオーバヘッドが影響し，周期タスクがデッドラインオーバーを起こしている．しかし，計測 1 で前述したように，このような状況のシステムはターゲットとせず，CPU 使用率 80%以下をターゲットとして評価を行う．表 4.5 にそれぞれのトータルデッドラインオーバー数を示す．

| スケジューラ | 全体 | 80%以上 | 80%以下 |
|---|---|---|---|
| TDOC_S | 4,803 | 3,236 | 1,567 |
| TDOC_D | 4,661 | 3,478 | 1,183 |
| TDOC_Dall | 4,847 | 3,774 | 1,073 |

表 4.5: 計測 2 デッドラインオーバー数

80%以上を含めて全体値としても，TDOC_D が TDOC_S よりもタスクのトータルデッドラインオーバー数が少ない．80%以下に関しては，TDOC_D は TDOC_S に対して 384 個のタスク実行のデッドラインオーバーを回避している．TDOC_Dall は TDOC_S に対して 494 個のタスク実行のデッドラインオーバーを回避している．

計測 1，計測 2 ともに，CPU 使用率 82%以下の状況では，静的優先度方式よりもデッドラインオーバーを起こすタスク数を削減することができた．結果，実システムにおいても適応型動的優先度方式が有効性のあるスケジューラであることがいえる．

図 4.6 では，CPU 使用率が上がるにしたがって，適応型動的優先度方式でのタスクがデッドラインオーバーした起動時優先度の平均値が下がっている．これは，計測 1 での結果と同じような傾向を示している．

### 4.3.3 スケジューラのオーバヘッド

計測 1，計測 2 での結果に適応型動的優先度方式の除算を行った場合の結果を図 4.7，図 4.8 に示す．それぞれ，TDOC_DN が適応型動的優先度方式で除算を行った場合となる．

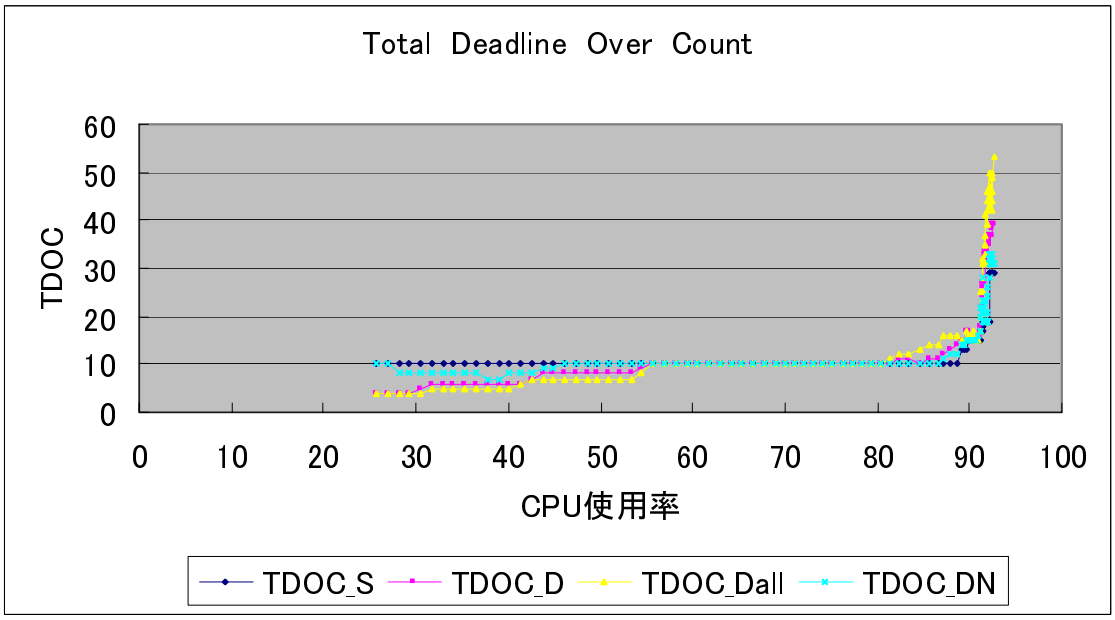


図 4.7: 計測 1 に除算を行った場合

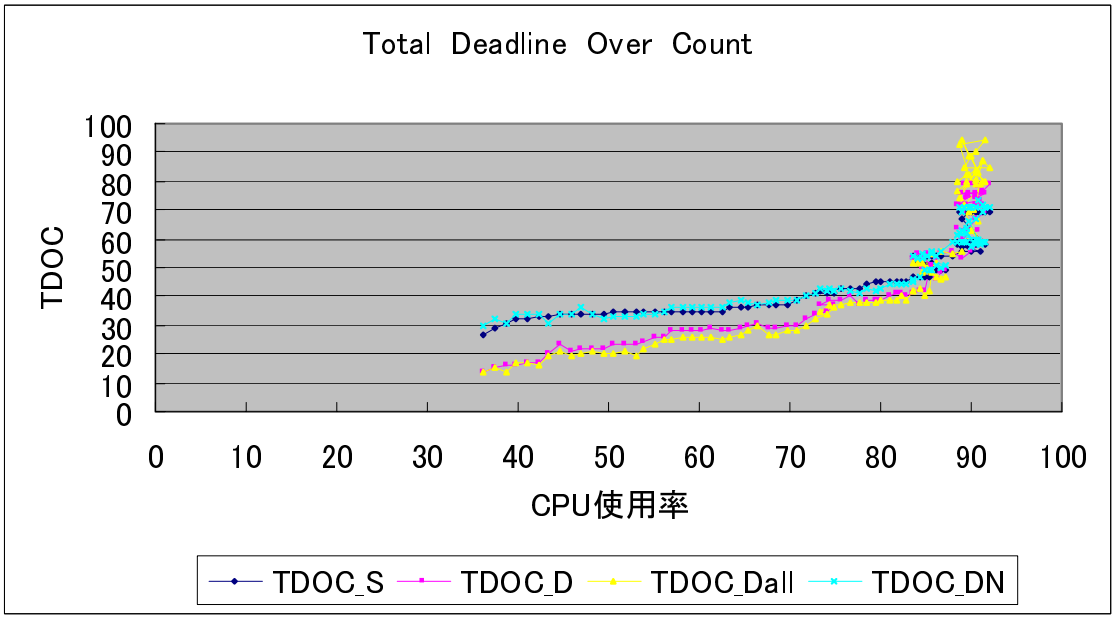


図 4.8: 計測 2 に除算を行った場合

**考察**

図から分かるように，除算を行った場合のオーバヘッドがタスクの実行に影響を与えているのが分かる．これにより，除算を行った場合と比較して，除算を行わない場合の方が有効であることを示した．

計測 1 で除算を行った場合の結果を表 4.6 に，計測 2 で除算を行った場合の結果を表 4.7 に示す．

| スケジューラ | 全体 | 80%以上 | 80%以下 |
|---|---|---|---|
| TDOC_S | 1,458 | 938 | 470 |
| TDOC_D | 1,738 | 1,355 | 383 |
| TDOC_Dall | 1,987 | 1,623 | 364 |
| TDOC_DN | 1,543 | 1,103 | 440 |

表 4.6: 計測 1 に除算を行った場合

| スケジューラ | 全体 | 80%以上 | 80%以下 |
|---|---|---|---|
| TDOC_S | 4,803 | 3,236 | 1,567 |
| TDOC_D | 4,661 | 3,478 | 1,183 |
| TDOC_Dall | 4,847 | 3,774 | 1,073 |
| TDOC_DN | 4,920 | 3,337 | 1,583 |

表 4.7: 計測 2 に除算を行った場合

表 4.6 では，適応型動的優先度方式での除算有りの場合において，静的優先度方式との差が 30 ある．静的優先度方式よりはタスクのデッドラインオーバーは減少しているが，除算なしで行った場合と比較すると，スケジューラ計算のオーバヘッドがタスクの実行に影響し，デッドラインオーバー数が増加している．

表 4.7 では，適応型動的優先度方式の除算有りの場合が，静的優先度方式よりもデッドラインオーバ数が 16 個増加している．これらの結果より，スケジューラの優先度計算のオーバヘッドがタスクの実行に影響を与えていることを示した．

### 4.3.4 補助計測

計測 1，計測 2 のタスクセットを変更し，デッドラインオーバーするタスクを，CPU 使用率 55%付近から出現させるような状況で計測を行った．図 4.9 に計測 1 を変更した場合を示し，図 4.10 に計測 2 を変更した場合を示す．

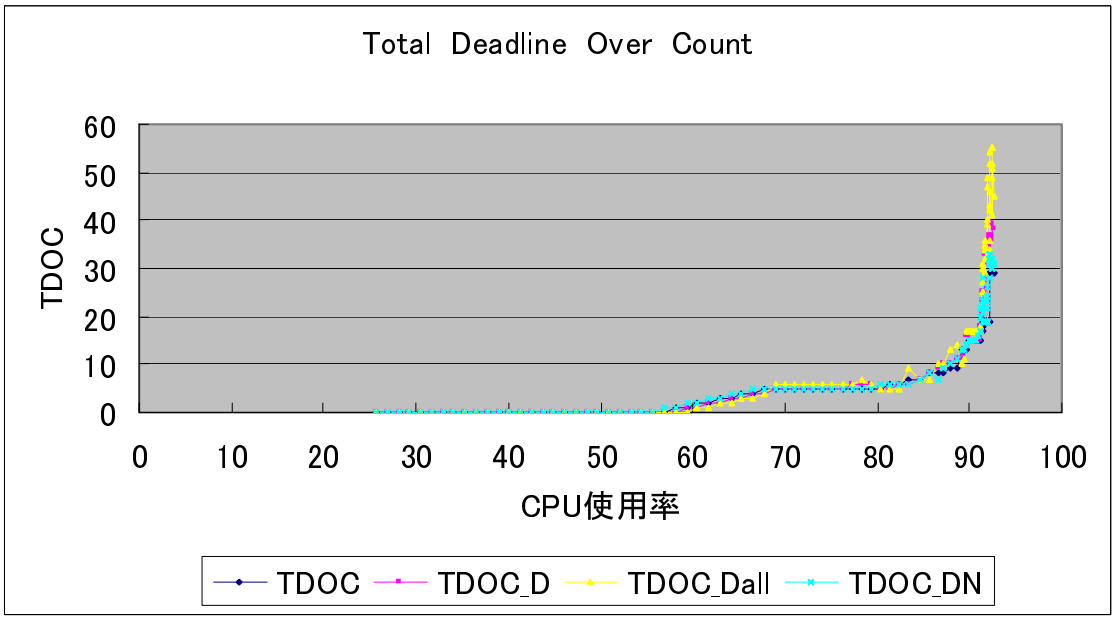


図 4.9: 計測 1 を変更

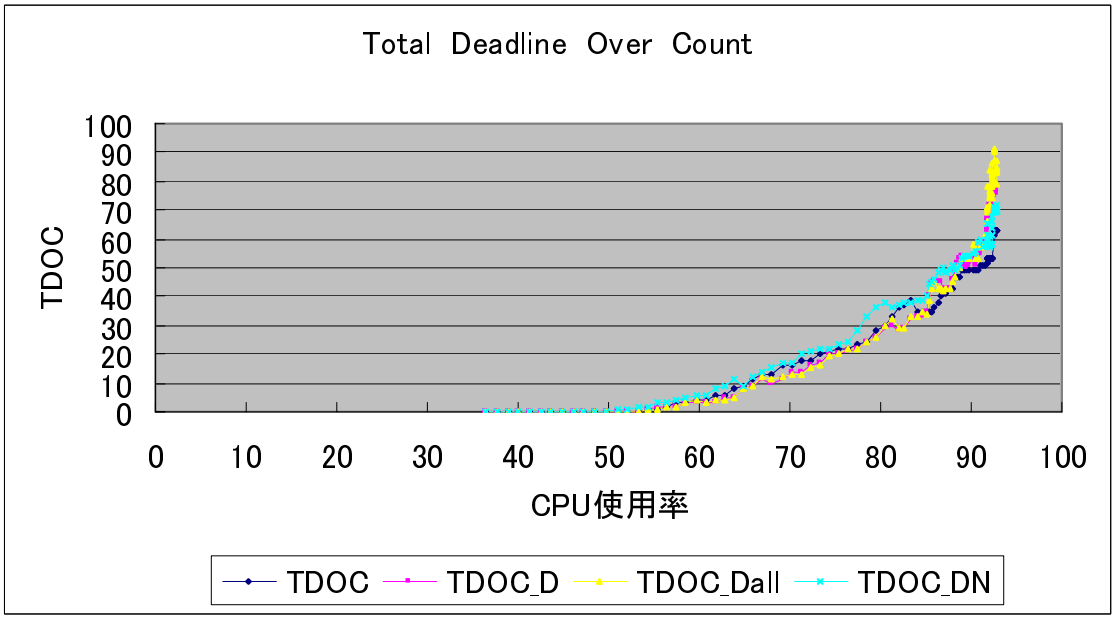


図 4.10: 計測 2 を変更

**考察**

図 4.9，図 4.10 での結果をそれぞれ表 4.8，表図 4.9 に示す．

| スケジューラ | 全体 | 80%以上 | 80%以下 |
|---|---|---|---|
| TDOC_S | 1,036 | 956 | 80 |
| TDOC_D | 1,344 | 1,266 | 78 |
| TDOC_Dall | 1,616 | 1,533 | 83 |
| TDOC_DN | 1,157 | 1,072 | 85 |

表 4.8: 計測 1 を変更したデッドラインオーバー数

| スケジューラ | 全体 | 80%以上 | 80%以下 |
|---|---|---|---|
| TDOC_S | 3,287 | 2,975 | 312 |
| TDOC_D | 3,755 | 3,483 | 272 |
| TDOC_Dall | 4,005 | 3,734 | 271 |
| TDOC_DN | 3,746 | 3,372 | 374 |

表 4.9: 計測 2 を変更したデッドラインオーバー数

計測 1 および計測 2 では，デッドラインオーバーをするタスクを故意的に発生させた．これによりスケジューラの傾向を示している．しかし，デッドラインオーバーするタスクが CPU 使用率 30%付近から頻繁に発生しているシステムは，そのシステムが破綻しているように見える．そのため，CPU 使用率 55%付近からタスクのデッドラインオーバーを発生させるような補助計測を行い，そのような場合でも静的優先度方式よりもタスクのデッドラインオーバー数を減少させることを示す必要がある．

表 4.8 では，TDOC_D だけが TDOC_S よりもデッドラインオーバー数が 80%以下で減少している．TDOC_Dall は 70%以上から TDOC_S よりもデッドラインオーバー数が 1 つ程度多くなっているため，トータルで見ると TDOC_S よりも 3 個のタスクがデッドラインオーバーしている．これは，スケジューラのオーバーヘッドが影響しているためである．また，TDOC_DN についても同じである．

表 4.9 では，TDOC_DN 以外は TDOC_S よりもデッドラインオーバー数を減少している．TDOC_DN に関しては上記の理由と同じで，スケジューラのオーバーヘッドが影響しているためである．

これら全ての結果により，適応型動的優先度方式は静的優先度方式よりもタスクのデッドラインオーバー数を削減できることを示した．また，スケジューラのオーバーヘッドを削減することにより，このような結果を得ることができたことを示した．

## 4.4 組込みプロセッサの評価

### 4.4.1 高速割込み応答機構の評価

本研究で使用する CPU(Casablanca) は，割込みに対して高速な応答を可能とすることを 2.4.1 項で述べた．4.3 での計測 1 および計測 2 では，Casablanca の機能を持つシミュレータで計測している．本項では，計測 2 で行った評価を通常の CPU の機能のみ持つシミュレータを使用することにより，割込み処理のオーバヘッドがタスクのデッドラインオーバーに影響を与えることを示し，本研究で使用する CPU の有効性を示す．タスクセットは計測 2 と同じものを使用する．また，今回の計測の方法を計測 3 とよぶ．

図 4.11 に，CPU 使用率に対する，タスクのトータルデッドラインオーバー数を示し，CPU 使用率に対するデッドラインオーバーしたタスクの起動時優先度の平均を図 4.12 に示す．

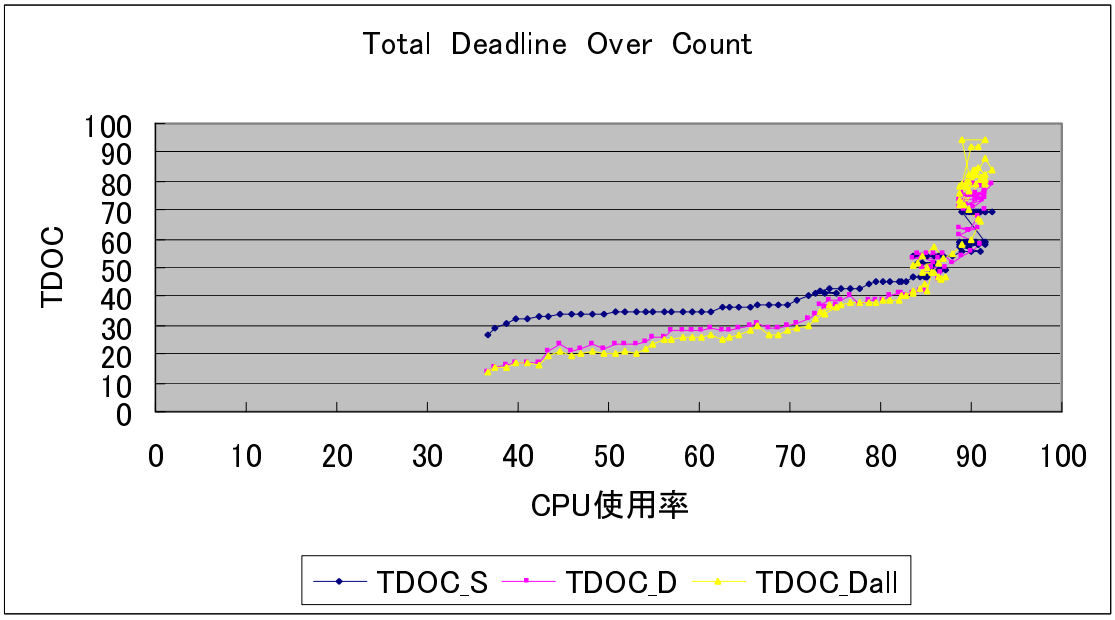


図 4.11: 計測 3-1

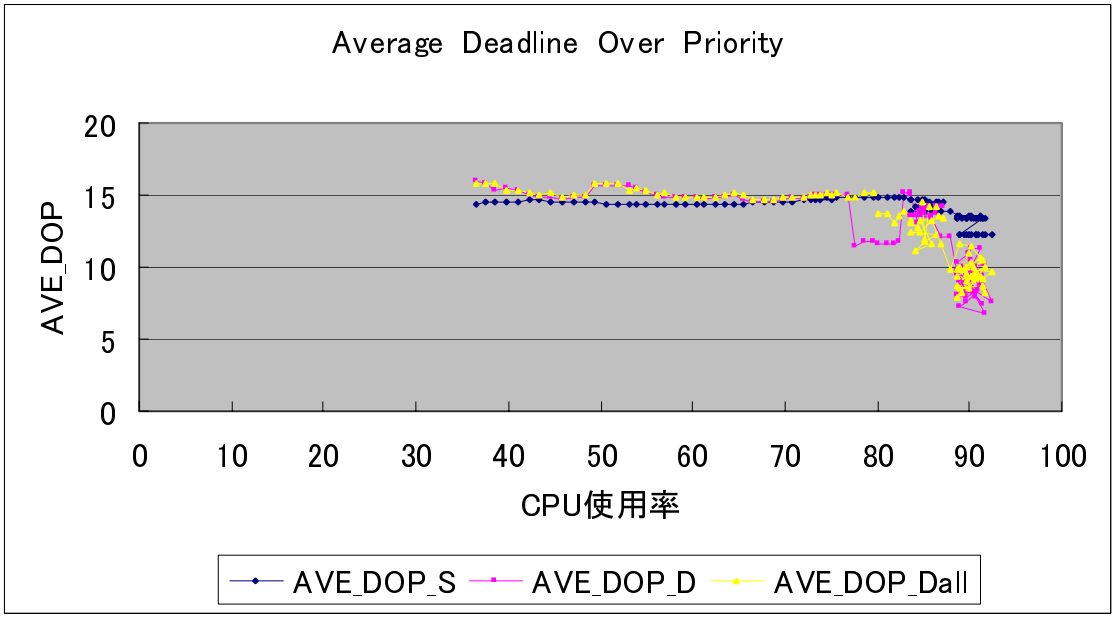


図 4.12: 計測 3-2

| スケジューラ | 全体 | 80%以上 | 80%以下 |
|---|---|---|---|
| TDOC_S | 4,808 | 3,239 | 1,569 |
| TDOC_D | 4,675 | 3,487 | 1,188 |
| TDOC_Dall | 4,844 | 3,729 | 1,115 |

表 4.10: 計測 3 デッドラインオーバー数

| スケジューラ | 全体 | 80%以上 | 80%以下 |
|---|---|---|---|
| TDOC_S | 4,803 | 3,236 | 1,567 |
| TDOC_D | 4,661 | 3,478 | 1,183 |
| TDOC_Dall | 4,847 | 3,774 | 1,073 |

表 4.11: 計測 2 デッドラインオーバー数

**考察**

計測結果を表 4.10 に示し，比較対象の計測 2 の結果を表 4.11 に示す.

表 4.11 の計測 2 が本研究で使用する CPU であり，表 4.10 の計測 3 が通常 CPU での結果である.

計測 2 に対して計測 3 の TDOC_S では，全体でのデッドラインオーバー数が 5 大きい. これにより，割込みで起こるコンテキストの退避によるオーバヘッドが，タスクのデッドラインオーバーに影響していることが分かる.

TDOC_D では，全体のデッドラインオーバー数が 14 増加している．これに対しても，コンテキスト退避のオーバヘッドが影響している.

TDOC_Dall では，全体で 3 個デッドラインオーバーが減少している．これは，全てのタスクを対象としているため，コンテキスト退避のオーバヘッドにより，タスクの余裕時間が短くなり，タスクの優先度が上がり結果的にデッドラインオーバーを回避する状況が存在するためである．しかし本研究における評価のターゲットである 80%以下では，全てにおいて本研究で使用する CPU によるデッドラインオーバー数を減らすことが示された.

### 4.4.2 データキャッシュ制御機構の評価

本研究で使用する CPU は，小規模のハードウェアによりデータキャッシュと外部メモリの間の一貫性を維持するデータキャッシュブロック強制 fill/update 機構を持つ．この機構により，外部メモリを介して周辺回路とのデータの授受を行うアプリケーションの高速実行を可能とする．

評価として，従来手法である store 命令でメモリに直接書き込む場合と本法による FU-Store 命令でメモリにライトバックした場合の速度差を評価する．

評価方法として，1KB のデータをメモリに書き込むときのオーバヘッドを従来手法と本手法で計測する．計測結果を表 4.12 に示す．

| 手法 | 実行時間 |
|---|---|
| 従来手法 | 10,930 |
| 本手法 | 4,754 |

表 4.12: メモリ書き込み実行時間

従来手法と比較して，本手法では 6,176 クロックのオーバヘッドを軽減した．

# 第5章　おわりに

## 5.1　まとめ

本研究で実装したオペレーティングシステムカーネルにおいて，スケジューラを適応型動的優先度方式にすることにより，従来から使用されていた静的優先度方式よりもタスクのデッドラインオーバー数を削減することが可能であることが示された．また，提案手法により効率よく優先度キューを使用することが可能となり，デッドラインオーバー数の削減につながった．カーネルのスケジューラは，柔軟に複数のスケジューラに対応させることが可能であり，動的にスケジューラを再構成することも可能である．また，適応型動的優先度方式での動的優先度の計算にかかるオーバヘッドを十分に小さいオーバヘッドで実装可能であることが示された．

## 5.2　今後の課題

今後の課題として，タスク数 100 以上での計測を行い，全てのタスクに対して動的優先度計算を行った場合のオーバーヘッドを見積もる必要がある．また，このオーバーヘッドが分かることにより，タスク数によって動的にスケジューラを変更し，実行状況によって変化させることを検証する．加えて，CPU 使用率が 80% 以上になった場合は，静的優先度方式にすることで，システム全体でのタスクのデッドラインオーバー数の総和が確実に減少することが期待できる．

# 謝辞

本研究を行うにあたり御指導，御鞭撻を頂いた北陸先端科学技術大学院大学情報科学研究科 田中清史 助教授に深く感謝するとともに，ここに御礼申し上げます.

適切な御意見，御助言を頂きました本学の日比野 靖 教授，井口 寧 助教授に誠に深く感謝致します.

多種多様な疑問や問題に対して親切なアドバイスをして頂いた田中研究室学友の請園智玲氏，今井俊晴氏，高橋礼彦氏，上田憲吾氏，南里洋亮氏，森田充氏，その他貴重なご意見，討論を頂いきました田中研究室の皆様をはじめ多くの方々に対して厚く御礼申し上げます.

最後に日頃よりあたたかく見守ってくださった両親に深く感謝致します.

# 参考文献


[1] 栗谷一路，田中清史，リアルタイム OS における適応型スケジューリング方式，電子情報通信学会技術研究報告 CPSY，Vol.102，No.478，pp.127–132,2002.

[2] Kiyofumi Tanaka, Takashi Matsumoto, "Casablanca: A Real-Time RISC Core for Embedded Systems ", International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the INternet, 2001, CD-ROM.

[3] $\mu$ITRON4.0 仕様 Ver. 4.00.00
http://222.sakamura-lab.org/TRON/ITRON/

[4] C.A.Healy, D.B.Whalley and M.G.Harmon, "Integrating the Timing Analysis of Pipelining and Instruction Caching," RTSS, 1995.

[5] C.L.Liu and J.W.Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," Journal of the ACM, Vol.20,No.1,pp.46-61,1973.

[6] A.K.Mok, "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment," Ph.D.thesis,MIT Cambridge MA, 1983.

[7] J.Y.T.Leung and J.Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic,Real-Time Tasks," Performance Evaluation, Vol.2,No.4,pp.237-250,Dec.1982.

[8] Wolf-Dietrich Weber and Anoop Gupta：Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture：Preliminary Results.In Proc.of the $16^{th}$ ISCA,p273 $\sim$ 280(1989)

[9] SPARC International,Inc.：The SPARC ARchitecture Manual Version 8,Prentice-Hall,Inc.(1992)