

学習向け組込みOSの実装(技術記録)

要旨

近年, 実用OSのソースコードは年々巨大化及び複雑化し, コードレベルからの学習及び研究は非常に困難である。学習向けOSも存在するが, ソースが複雑化し, OS学習者は理解が難しい。さらに, 学習OSでは, 機能ごとの実装法と管理アルゴリズムは, 一般的に一通りである。そのため学習者は単一のOSコードレベルで機能ごとの複数実装方式が可読できず, かつそれらの動作状況を体験できない。

本研究の目的は, 複数のOSソースを可読せず, 単一のカーネルコードレベルから機能ごとの複数実装法を可読でき, それらの動作状況を体験できるフルスクラッチOSを実装することである。そのためには, 可読性を向上させ, OS各機能の既存方式を調査及び特徴や得失を考察する。さらに, 考察結果より新規方式を提案し実装する。また, それらの動作状況を体験するために, システムコールを使用したユーザタスクライブラリ集を自作する。

複数方式を実装する機構は, OSの基本的機能であるタスクスケジューラ, 同期と排他に使用するセマフォと mutex, タスク間通信に使用するメールボックス, 時間管理に使用するOSアラーム機能と周期タイマ機能, 優先度逆転防止機構とし, 複雑な機能は実装しない。

モノリシックカーネル設計とし, カーネルコンフィギュレーションによるスケーラビリティの配慮は行わず, 組込みOSの仕様である μITRON4.0 を参考に本研究OSの仕様を一般化する。さらに, 組込みシステム設計規格である MISRA C2004 のルールを実装するOSに適用し, 実装するOSコードのコメント比率を上げ, アセンブラー比率を下げる。また, 可読性を向上させるツール類への対応を行う。

これにより, ユーザはさまざまなOS分野の機能ごとの複数管理方式を本研究OSから一度に知る事ができ, それらのレスポンス結果及びレスポンスタイムを体験できる。

目次

1. はじめに
 - 1.1. 背景 1
 - 1.2. 目的 1
2. 研究方針 2
 - 2.1. 研究方針 2
 - 2.2. オープンソースの紹介 3
 - 2.3. 対象 3
 - 2.4. 環境 4
3. システム概要 3
 - 3.1. 組込みシステム全体構成 3
 - 3.2. OS機能全体構成 3
 - 3.3. OS各処理全体構成 6
4. タスク管理 9
 - 4.1. タスクの状態 9
 - 4.2. タスク管理データ構造 10
5. 割込み管理及びシステム管理 22
 - 5.1. 割込み管理データ構造 22
 - 5.2. 各割込み詳細 24
 - 5.3. 非タスクコンテキストシステムコールとサービスコール 26
 - 5.4. ディスパッチャ管理データ構造
 - 5.5. システムロールバック機能
6. スケジューラ及びレディー管理 30
 - 6.1. 各スケジューラ詳細 31
 - 6.2. レディー管理データ構造 38
 - 6.3. スケジューラ考察 42
 - 6.4. スケジューリングの新規方式(Priority Fair Scheduling) 45
 - 6.5. スケジューラ動的取替え機能 46
7. 時間管理 50
 - 7.1. ハードタイマドライバ 50
 - 7.2. ソフトタイマドライバ 50
 - 7.3. 周期ハンドラ管理データ構造
 - 7.4. アラームハンドラ管理データ構造
8. 同期・排他・通信機能 55
 - 8.1. セマフォ管理データ構造 55

8.2.	メールボックス管理データ構造
8.3.	mutex 管理データ構造 55
9.	優先度逆転防止機構管理
9.1.	各優先度逆転防止機構の詳細 56
9.2.	優先度逆転防止機構の考察 58
9.3.	優先度逆転防止機構の新規方式(Virtual Priority Inheritance Protocol) 63
9.4.	本研究OS 上の実装
10.	init 管理
10.1.	ブートシーケンスと OS メモリマップ
10.2.	カーネル init
10.3.	タスクのスタートアップとエンド
11.	他OSとの可読性比較調査結果 71
12.	おわりに 73
12.1	研究の成果 73
12.2	今後の課題 73
13.	参考文献とツール類

1. はじめに

1.1. 背景

近年、実用 OS のソースコードは年々巨大化及び複雑化し、コードレベルからの学習及び研究は非常に困難である。学習向け OS も存在するが、ソースが複雑化し、OS 学習者は理解が難しい。また、OS はリンク制御及びブートローダと連携する内部処理があり、単に OS をコードリーディングしただけでは理解できない。さらに、学習 OS では、機能ごとの実装法と管理アルゴリズムは、一般的に一通りである。そのため学習者は単一の OS コードレベルで機能ごとの複数実装法による動作状況が体験できない。

1.2. 目的

本研究の目的は、複数の OS ソースを可読せず、单一のカーネルコードレベルから機能ごとの複数実装法を可読でき、それらの動作状況を体験できる OS を実装することである。そのためには、リンク制御及びブートローダを含むフルスクラッチ OS であり、可読性を向上させ、OS 各機能の既存方式を調査及び特徴や得失を考察する。さらに、考察結果より新規方式を提案し実装する。また、それらの動作状況を体験するために、システムコールを使用したユーザタスクライブラリ集を自作する。

2. 方針

2.1. 研究方針

本研究OSは学習向けOSであるため、資源汎用性ではなく、ハードウェアの機能も比較的小ないH8/3069Fマイコンを対象とするフルスクラッチ組込みOSにする事にした。そのため、OS記述言語は一般的なC言語、アセンブラー言語、リンクスクリプトとする。

OS及びブートローダ、リンク制御、ライブラリすべて最初から自作していっては膨大な時間がかかるため、オープンソース(参考文献[18])を母体として使用する事にした(オープンソースOSは2.2節参照)。

本研究では、このオープンソースOSの内部データ構造を理解し、目的に沿うように改造する。可読性として(1)～(2)の考慮を行い、(2)については他OSの可読性比較調査を行う事にした。目的で掲げた機能ごとの複数実装法は(3)の考慮を行う事にした。

(1) 組込みシステムレベルとしての可読性考慮

(1-1) カーネル設計

カーネル設計は大きく、マイクロカーネル設計とモノリシックカーネル設計がある。マイクロカーネル設計では一部のOS機能がスレッド化し、プリエンプトの回数が多くなり、学習者には動作状況がわかりにくい。そのため、本研究OSはモノリシックカーネル設計にする事にした。

また、実装する学習向けOSは、組込みでは代表的な割込みドリブン型を採用し、厳しいメモリ制約がなく、リアルタイム処理及びカーネルスケーラビリティが求められないものとした。

(1-2) コンフィギュレータ

OSビルド前にカーネルコンフィギュレーションさせ、カーネルオブジェクト類を制限または自動生成する機能があるとスケーラビリティは向上するが、OS可読性は低くなる。また、コンフィギュレータについては、ある程度しかOS標準規格で定められていないので、使用しないものとし、すべて参考文献[16]のμITRON仕様と参考文献[17]のT-kernel仕様にある動的API仕様とした。システムコールは仕様を一般化できるため、参考文献[16]と参考文献[17]を参考とした。なお、本研究OSではユーザがID管理の手間を軽減できるよう、カーネルオブジェクト生成系システムコールはすべて、カーネルID自動割付けにする事にした。

(1-3) カーネルの複雑な機能

メモリの仮想化及びサブシステム、TCP/IPプロトコルスタック、多重割込みを実装するとカーネルソースが複雑化する。そのため、これらの機能は実装しない事にした。

(2) コードレベルとしての可読性考慮

(2-1) MISRA C規格(参考文献[15])

C言語プログラミングでは可読性が低下するとされている構文がある。そのため、可読

性, 安全性, 可搬性を目的とした MISRA C2004 規格を参考にコーディングを行い, 半分以上のルールを本研究OSに適用した.

(2-1) コメントとアセンブラー

一般的に可読性は, コメント数が多く, アセンブラコードが少ないほうが向上する. そのため, コード全体のコメント比率を多くし, アセンブラ比率が少ない実装とする. なおコメントの記述方法は参考文献[6], [15]のとを参考にした.

(2-3) ツール類への対応

可読性及び管理性を高めるために doxygen(参考文献[42], 参考文献[43]), graphviz の対応コーディングを行った. これにより, コードの可視化とコードをドキュメントとして管理できる.

(3) 複数の実装法

OSにおいて最も重要な機能であるスケジューラを複数実装する事にした. 実装するスケジューラはさまざまな実用OSコード及び学習OSコード, 文献を調査し本研究OSに適用できるものとした結果, 13種となった.

- ・既存の方式であるスケジューラを 12種
- ・新規の方式であるスケジューラを 1種

複数のスケジューラ実装に補いレディー管理を 5通り実装した. さらに, 基本的なOS機能であるタスク管理, 同期及び排他に使用するセマフォ, 完全な排他に使用する mutex, 時間管理に使用するアラーム機能と周期タイマ機能を実現するためのOS内部データ構造を 2通り実装した.

また, mutex の属性である優先度逆転防止機構を複数実装する事にした. 実装する優先度逆転防止機構も OSコード及び文献を調査し本研究OSに適用できるものとした結果, 5種となった.

- ・既存の方式である優先度逆転防止機構を 4種
- ・新規の方式である優先度逆転防止機構を 1種

2.2. 母体としたオープンソースOS

このオープンソースOSは専門家である坂井氏によって, C言語, アセンブラ言語とリンクカスクリプトで記述された 3000Step 程度のフルスクラッチOSである(参考文献[18]). 主に, このオープンソースは簡易OS, 簡易ブートローダ, 簡易Cプログラミングライブラリ, リンカ制御を含む. 簡易ブートローダはシングルブートとなる. 簡易OSは割込みドリブン型, マイクロカーネル設計であり, 最小限の機能となっている.

2.3. 本研究OS可読対象者

本研究システムは, カーネル(C言語とアセンブラ言語で記述), ブートローダ(C言語とアセンブラ言語で記述), リンカ制御(リンクカスクリプトで記述), 簡易アプリケーション

ライブラリ, 簡易標準Cライブラリを含んでいる。したがって, C言語及びアセンブラー言語, リンカスクリプトの知識があり, 基本的なハードウェアとハードウェア制御プログラミングの知識, 基本的なOS概念の知識, マルチスレッドプログラミングの知識がある初心者を対象とする。

2.4. 環境

本研究OSを使用する場合は, ホストOSにDebian GNU/Linux, ハードウェアにH8/3069Fマイコン, ROMライタであるH8write(参考文献[36]またはkz_h8write(参考文献[37])), エミュレータであるminicom(参考文献[38])またはkermit(参考文献[39]), OS実行ファイルを転送するlrzsz(参考文献[40])が必要となる。コードリーディングをする場合は, doxygen(参考文献[41]), graphviz(参考文献[42])があると便利である。本研究OSを改造する場合は, h8 gcc-3.4.6クロスコンパイラ(参考文献[43]), アセンブラーとリンカのユーティリティであるbinutils-2.19.1(参考文献[44])が必要となる。またバージョン管理のgit(参考文献[45])があると便利である。これらは11節のツール類で入手できる。

3. システム概要

3.1. 組込みシステム全体構成

本節ではシステムの概要について述べる。組込みシステムの完成後の構成を下図 3-1 に示す。

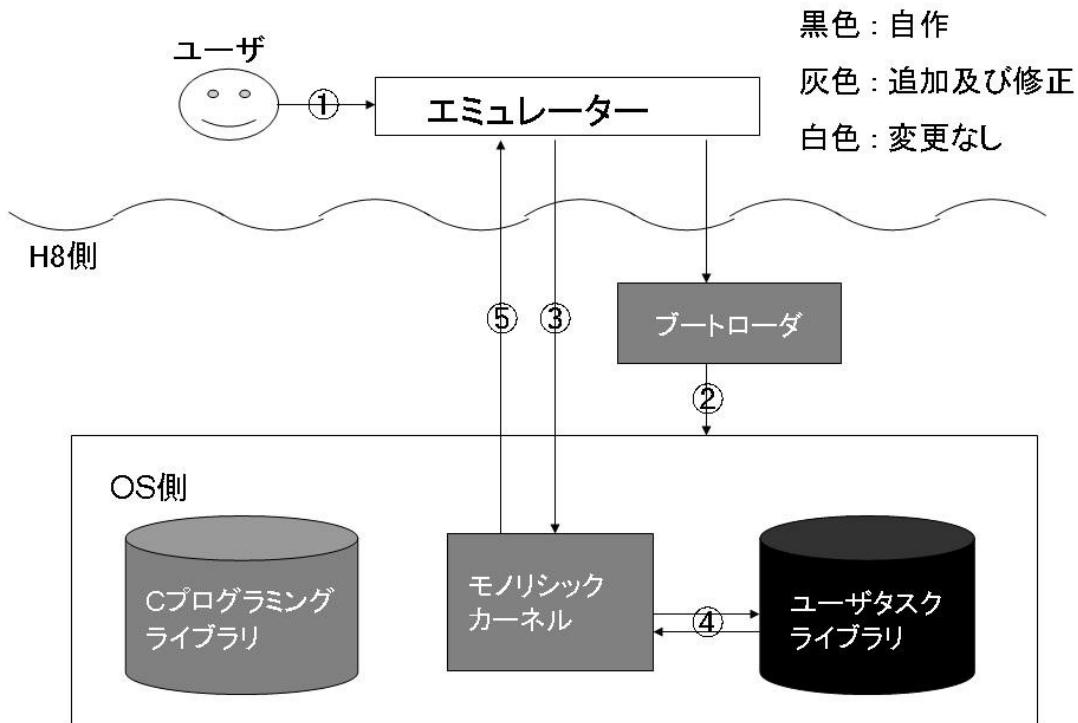


図3-1 完成後の組込みシステム全体構成

本研究システムは大きくカーネル、ブートローダ、ユーザタスクライブラリ、Cプログラミングライブラリで構成されている。また図 3-1 は、すでにブートローダが ROM 化されているものとする。

- ① ユーザはエミュレータを通して OS 実行ファイルを Xmodem 転送プロトコルによりブートローダへ送信し、OS の読み込み指示をする。
- ② ブートローダが OS をメモリへ読み込む。
- ③ エミュレータを通してユーザタスクセットの選択を行い、カーネルへ指示する。
- ④ カーネルがユーザタスクライブラリから指示されたユーザタスクセットを起動する。
- ⑤ ユーザタスクが起動し、結果がエミュレータへ返却される。

3.2. 機能から見た OS 全体構成

OS を外面の機能から見た全体構成を下図 3-2 に示す。外面からの機能のみとなるので、図 3-2 に依存関係は示さない。

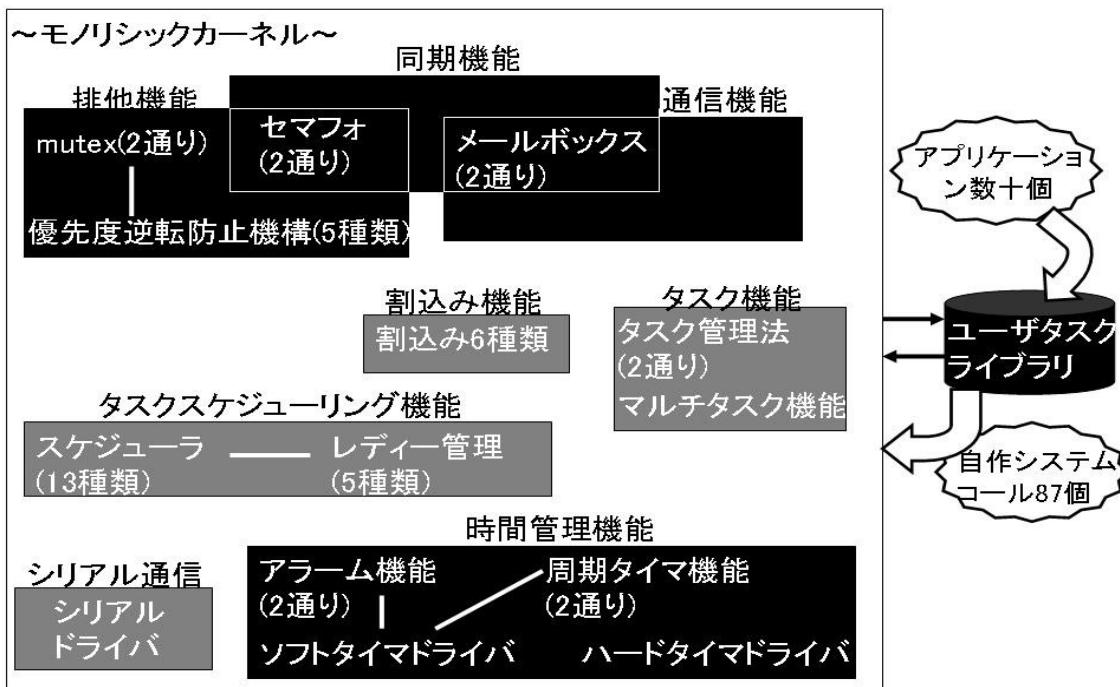


図3-2 実装したOS全ての機能

- ① タスク機能については、OS 内部でのタスク管理データ構造を 2通り実装した。さらに、マルチタスク機能がある。これらは、4節と 5節で説明する。
- ② 排他、同期機能はセマフォを使用して行う。セマフォ機能については、OS 内部での管理データ構造を 2通り実装した。セマフォについては 8節で説明する。
- ③ 完全な排他機能は mutex を使用して行う。mutex 機能については、OS 内部での管理データ構造を 2通り実装した。mutex については 8節で説明する。
- ④ 優先度逆転防止機構は 2.1.(3) 説で簡単に述べた。これらは 8節で詳しく説明する。
- ⑤ 割込みはシステムコール割込み、シリアル送信割込み、シリアル受信割込み、タイマ割込み、NMI 割込み、例外モデルとなる。これらは 5節で説明する。
- ⑥ スケジューリング機能は 2.1.(3) 説で簡単に述べた。これらは 6節で説明する。
- ⑦ ②の実装によりレディー管理構造が 4通りとなった。これらは 6節で説明する。
- ⑧ アラーム機能については、OS 内部での管理データ構造を 2通り実装した。7節で説明する。
- ⑨ 周期タイマ機能については、OS 内部での管理データ構造を 2通り実装した。7節で説明する。
- ⑩ ソフトタイマドライバはタイマ要求の並行管理ができる、多少の時間誤差が生じるドライバである。主に、スケジューラ、アラーム機能と周期タイマ機能が使用する。また、操作 API によってはセマフォ、mutex も使用する。これは 7節で説明する。
- ⑪ ハードタイマドライバはタイマ要求の並行管理はできないが、精度の高いタイマドライバである。主に、スケジューラ、アラーム機能と周期タイマ機能が使用する。これは 7節で説明する。

イバである。主に、ユーザタスクが使用する。これは7節で説明する。

- ⑫ OSのシリアル送信処理と受信処理の制御を行う。マイクロカーネル設計からモノシリックカーネル設計へ変更するにあたって追加、修正を行った。5節で説明する。
- ⑬ 組込み向け簡易アプリケーション集である。
- ⑭ 本研究OSには、自作システムコールが87個実装した。システムコールの詳細は付録で述べる。
- ⑮ ユーザがアプリケーション作成時にCプログラミングを行えるように、Cの簡易標準ライブラリを実装している。このCの簡易標準ライブラリは必要最低限の関数のみ(13個)の構成となる。オープンソースから多少の追加及び修正を行ったが、どのような関数があるかは付録で述べる。

3.3. OS各処理の関連図

カーネルとユーザタスクライブラリのインターフェース及びカーネルとブートローダの連携処理を含めたOS各処理部の関連を下図3-3に示す。各部依存関係を矢印で示す。

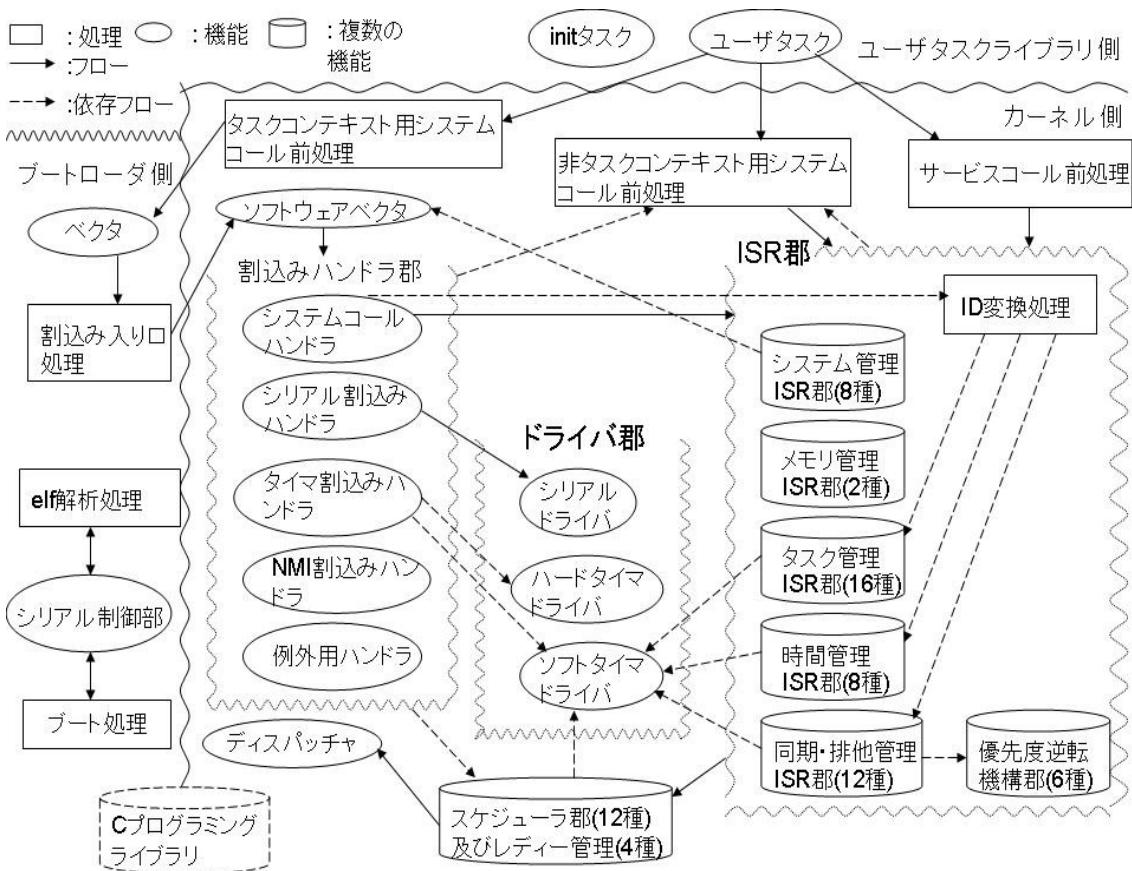


図3-3 OS内部全体構成

主に、ユーザタスクはAPI(システムコール及びサービスコール)を用いてOSの機能を使用する。APIが発行されると各割込みハンドラが起動し、その延長で割込みサービスルー

チング(以下 ISR)処理に移り、スケジューラがタスクをスケジューリングし、ディスパッチャがタスクをディスパッチする(図 3-3 の実線部). なお、ブートローダの elf 解析処理及びシリアル制御部については本論文では述べない.

4. タスク管理

本節では組込みOSの核となるタスク管理について述べる。本研究で開発したOSではタスクは独立であり、並行処理可能な単位として扱う。そのため、マルチタスクとなる。またタスクに親子関係はないものとし、タスク単一の粒度を独立して並行動作させるので、タスクがスレッドとなる。

4.1. タスクの状態

(1) タスクの状態と遷移

タスクの状態遷移を下図4-1に示す。

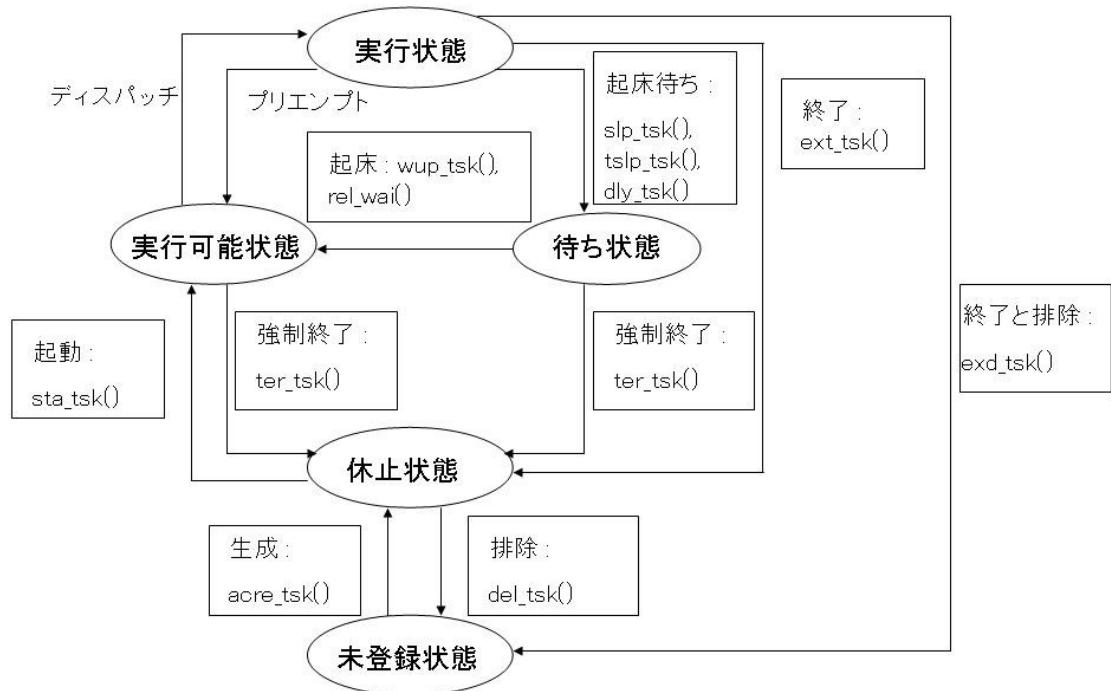


図4-1 タスク状態遷移

タスクを下記の5つの状態に分類する。

実行状態(activ)・・・タスクにCPUが割り付けられ、実行している状態である。OS内部ではレディーのデータ構造で管理されていて、実行状態カウンタを設定している。

実行可能状態(ready)・・・実行の準備が整い、レディーのデータ構造内で管理されている

状態である。OS内部ではレディーのデータ構造で管理されている。

待ち状態(wait)・・・何らかのカーネルオブジェクト等の取得を待っていて、OS内部で

はレディーのデータ構造内では管理されておらず、待ちカウンタを設定している状態である。

休止状態(dormant)・・・タスクは生成されているが、OS内部ではレディーのデータ構造内で管理されておらず、休止カウンタを設定している状態である。

未登録状態(non-existent)・・・タスクは生成されていない(メモリに存在しない)状態である。

各状態の遷移はシステムコールで行う。これらのシステムコールはタスク管理システムコールとタスク付属同期機能システムコールに分類され、4.3節と4.4節で説明する。また、実行可能状態から実行状態の遷移であるディスパッチと実行状態から実行可能状態の遷移であるプリエンプトはOSの機能となる。

実行状態、実行可能状態、待ち状態、未登録状態は基本的にOSにサポートされる状態である。

本研究OSでは、さらに効率化のため休止状態を実装した。具体的には、周期タスクが存在する場合、タスクが終了の度に未登録状態へ遷移すると、タスクを起動する際にタスク生成から行わなくてはならないからである。タスク生成のシステムコールは処理が重く、周期機能に誤差が蓄積する。

4.2. タスク管理データ構造

本節ではOS内部で扱うのタスク管理データ構造について述べる。

(1) タスク構造体

タスク構造体(以下TCB)の情報を下図4-2に示す。この構造体は本研究OSにおける最も重要な構造体となり、タスク1つの制御ごとに持つべき情報を格納した構造体である。

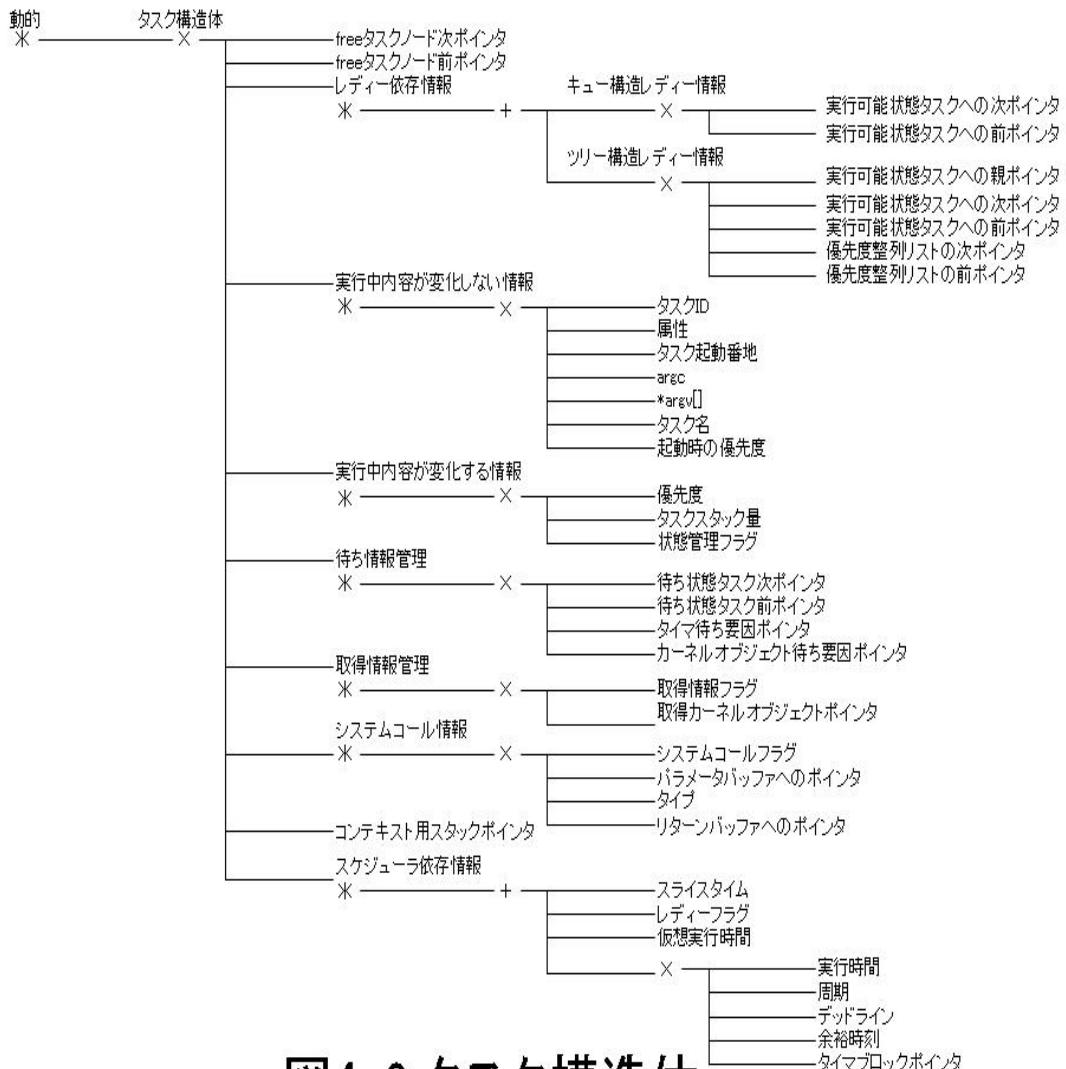


図4-2 タスク構造体

free タスクノードへのポインタは 4.2.(4), レディー依存情報は 6.1, 実行中内容が変化しない情報は 4.4.(1), 実行中内容が変化する情報は 4 節及び 5.1 節, 待ち情報は 4.2.(2), 取得情報は 4.4.(3), システムコール情報は 5 節スケジューラ依存情報は 6 節で述べる。

(2) タスク状態管理と待ち要因管理

各タスクの状態は 4.2.(1) 図 4-2 の状態管理フラグ下位 2 ビットで管理する。また、残りのビットをタスク待ち要因管理として使用する。

待ち要因をすべて列挙すると以下のようになる。

① タイムアウト待ちだけのタスク

タイムアウト付き起床待ち, タイムアウト付き自タスクの遅延である。

② カーネルオブジェクト待ちだけのタスク

セマフォ取得待ち, mutex ロック待ちである。

③タイムアウト待ちとカーネルオブジェクト待ちの両方のタスク

タイムアウト付きセマフォ取得待ち, タイムアウト付き mutex ロック待ちである。

①～③のそれぞれは, 同時には起こらないため, 4.2.(1).節の図 4-2 の状態管理フラグ 3, 4, 5, 6 ビット目で管理する. 下図 4-3 に示す.

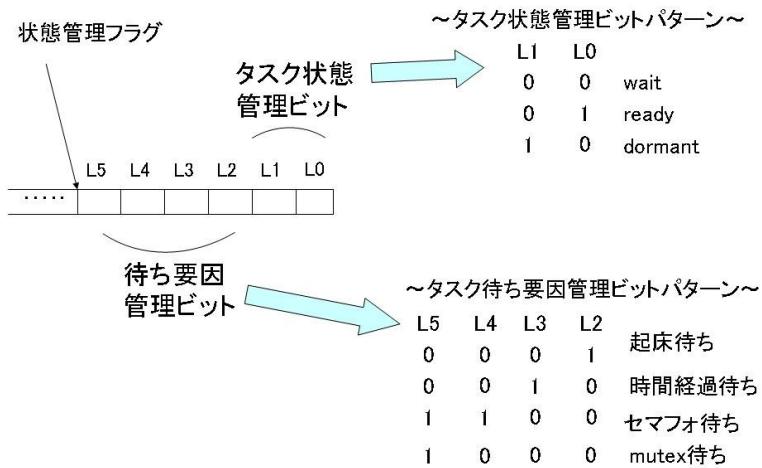


図4-3 タスク状態管理と待ち状態管理

また, タイムアウト付きの待ち要因はタイマコントロールブロックを持っている. 各カーネルオブジェクトでタイマ管理を分けていないので, ②と③についてタイムアウト付きの場合と別途フラグ管理はしない. タイマコントロールブロックへのアクセスを高速化するために, 4.2.(1).節の図 4-2 のタイマ待ち要因ポインタで管理する. 下図 4-4 に示す.

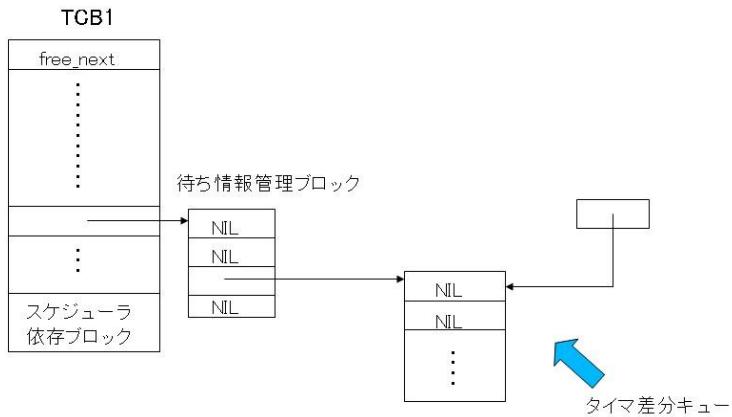


図4-4 タイムアウト付き待ち要因の場合

さらに、どの ID のカーネルオブジェクトに繋がれているかについて、4.2.(1)図 4-2 の k カーネルオブジェクト待ち要因ポインタで管理する。タイムアウト待ちとカーネルオブジェクト待ちの両方持っている場合を下図 4-5 に示す。

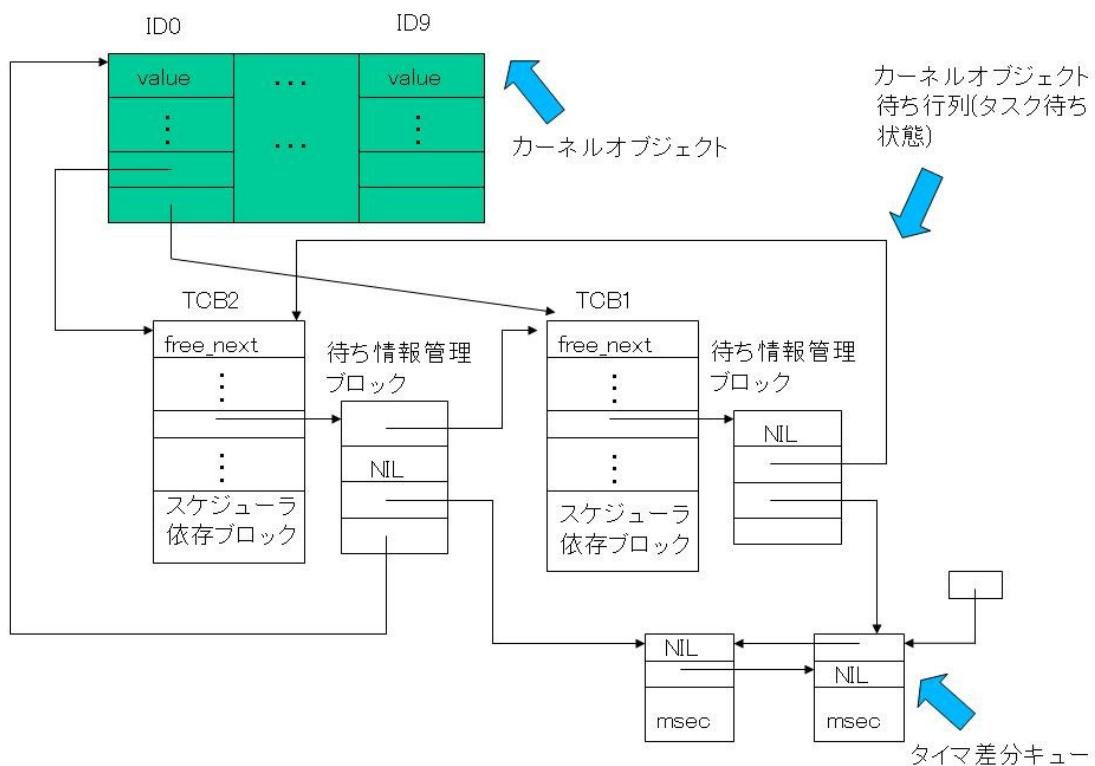


図4-5 タイムアウトとカーネルオブジェクト待ち両方の場合

この方法によって、タスクは ID 変換テーブルでアクセスできるので、対象カーネルオブジェクト待ち行列を先頭から走査する必要がなくなる(タスクは ID 変換テーブルで 0(1) アクセス 4.2.(4)節参照).

(3) タスク取得情報管理

タスクが実行状態から休止状態または未登録状態に遷移した時にカーネルオブジェクトを保持していたならば、カーネルが解放しなければならない。初学者向け OS のため、アプリケーション設計において、カーネルオブジェクトの解放忘れが懸念されるからである。

解放する方法として、2通り考えた。

(3.1) タイマを使用する方法

カーネルオブジェクトを取得時に一定のタイマをかける。解放されない時は、タイマ割込みハンドラの延長で、カーネルが自動解放を行う。この方法ではアプリケーション側でタイマ時間の設計を行う必要があり難しくなる。

(3.2) 取得情報を管理する方法

この方法はタスクがカーネルオブジェクトを取得する度に、取得情報を保持しておく。解放されない時にその情報をチェックし、カーネルが自動解放を行う。具体的には、タスクが休止状態または未登録状態へ遷移するシステムコールの割込みサービスルーチン内で 4.2.(1)図 4-2 の取得フラグを見てどのカーネルオブジェクトを取得しているか知り、4.2.(1)図 4-2 の取得カーネルオブジェクトポインタでその取得しているカーネルオブジェクトの位置へアクセスする。取得フラグと取得カーネルオブジェクトポインタの管理方法は 4.2.(2) と同様である。この方法では、タスクが何個のオブジェクトを取得するかわからぬいため、取得情報の管理コードが膨大となり、処理が重くなる。

本研究 OS では(3.2)の方法を採用した。しかし、この自動解放機能は最後に取得したカーネルオブジェクト以外は自動解放の動作保障しない。対象のカーネルオブジェクトはセマフォと mutex となる。

(4) TCB 管理データ構造及び ID 変換処理

本研究 OS は OS 機能ごとのさまざまな実装を行っている。TCB 管理のデータ構造として、静的型と動的型の 2 通り実装した。静的型は(1)を配列で管理し、動的型は(1)をリンクドリストで管理する。また静的型の場合は、配列の index(タスク ID に相当)で 0(1) アクセス可能だが、動的型の場合はリンクドリストを検索するため、0(n) アクセス(n : TCB の数)となってしまう。

動的型のアクセスを 0(1) にする方法として、静的型 TCB と動的型 TCB のポインタをタスク ID とし、ユーザ側で管理する方法が考えられる。しかし、OS 内部のデータ構造のアクセスをユーザ側で許してしまうと、ユーザ側から OS 内部のデータ構造を自由に書き換える事ができるので、セキュリティの観点から危険であるため、ユーザ空間と OS 空間は切り分けなくてはならない。

また、ID は上限値があるため、ID 変換テーブル(可変長配列)を採用する。具体的には

acre_tsk()システムコールで ID 変換テーブルを設定し、変換テーブルの index をタスク ID として返却する。以降、ID をパラメータとして使用するシステムコールはすべて、ID 変換テーブルで ID を変換する。また、acre_tsk()時に静的型か動的型を選択し、ID 変換テーブルを設定することによって、他のタスク管理システムコール使用時にユーザは静的型と動的型を意識する必要がなくなる。そして、del_tsk()システムコールで ID 変換テーブルの設定を解除し、TCB を初期状態へ戻す。なお、本研究OSは学習向けであるため、多量のタスクが生成されることは懸念されないため、ハッシュは引かないものとする。ID が枯渇した場合の状況も考えて、TCB 管理データ構造とそのデータ構造へのアクセス方法(主に動的型)を何通りか考えた。(4.1), (4.2)に示す。

(4.1) alloc リストと free リストの 2つ持つ実装

init 時に ID 変換テーブルの index と同じ数だけ、静的型の初期化された可変長配列を作成し、動的型の初期化された free リストを作成しておく。下図 4-6 に示す。なお、free リストの作成は TCB のメンバである 4.2.(1) 図 4-2 の free タスクノードへの次ポインタと free タスクノードへの前ポインタを使用する。

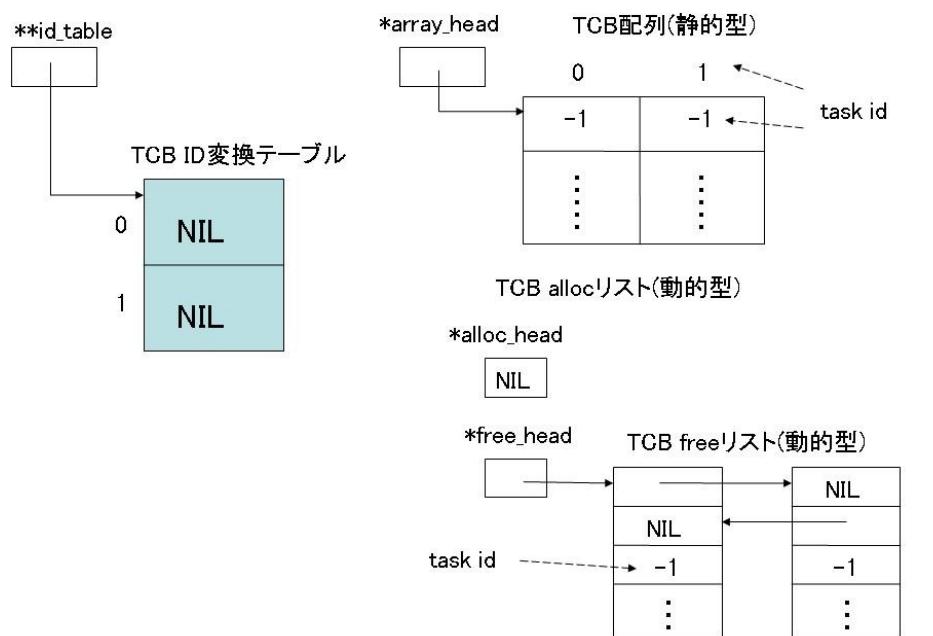


図4-6 init時の管理データ構造

acre_tsk()で動的型が指定された時は、free リストから TCB を抜き取り、alloc リストに挿入する。ID0 が作成された状態を下図 4-7 に示す。

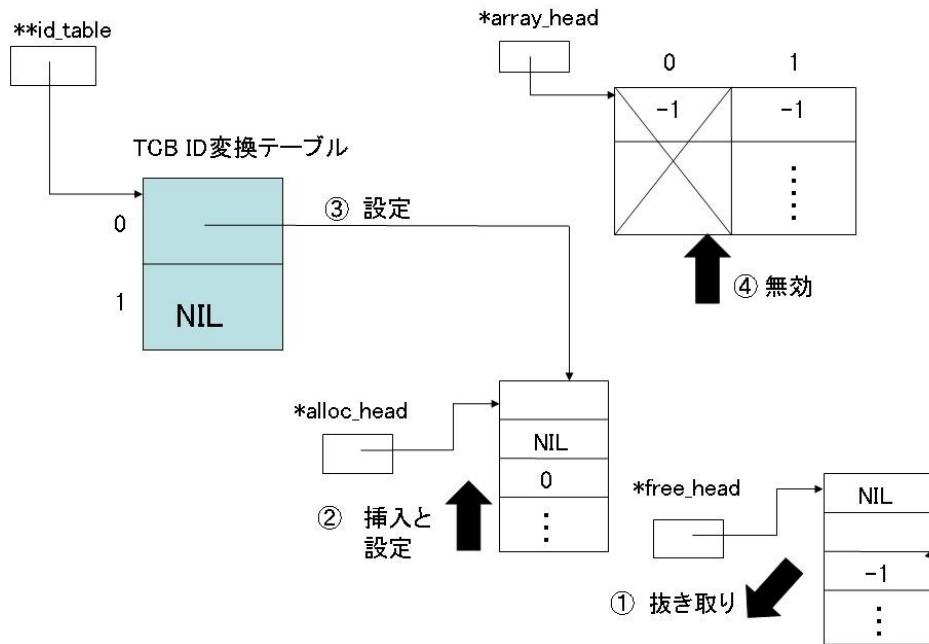


図4-7 タスクID0をcre_tsk()で生成した状態

`del_tsk()`で動的型が指定された時は、alloc リストから抜き取り、free リストへ挿入する。ID0 が排除した状態を下図 4-8 に示す。

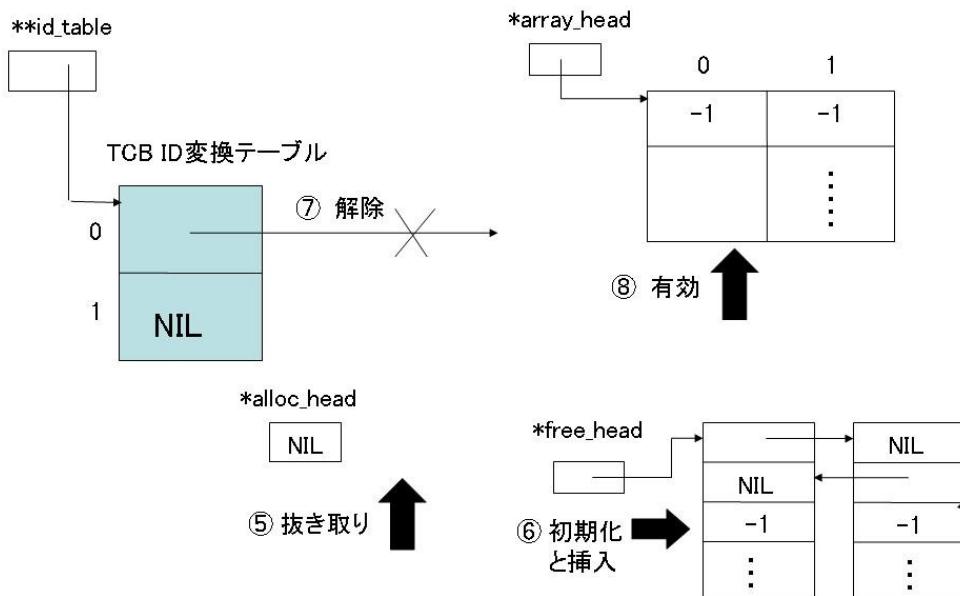


図4-8 タスクID0をdel_tsk()で排除した状態

タスク ID0～ID1 が使用されていて、ID が不足した時の状態を下図 4-9 に示す。倍の ID 変換テーブルの領域をとってきて、古い ID 変換テーブルをコピーする。静的型の可変長配

列も倍の領域をとってきて、古い可変長配列をコピーする。動的型は倍の free リストの領域をとってきて、そこから、古い alloc リストをコピーし、新しい alloc リストを形成する。状態を下図 4-9 に示す。

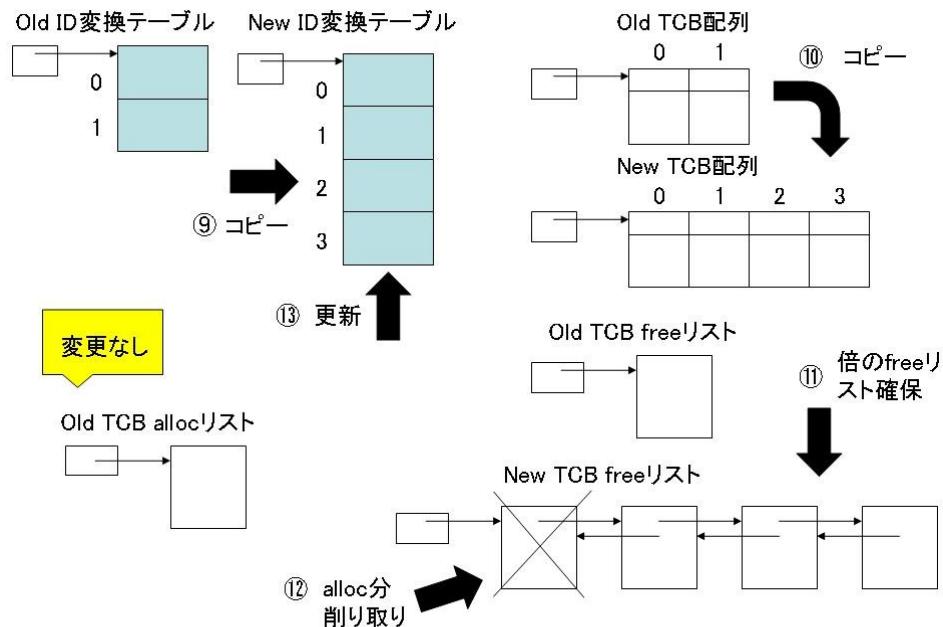


図4-9 タスクID不足時の対応

(4.2) 単一のリストで alloc と free を管理する実装

(4.1)と同様に、init 時に静的型は初期化された可変長配列を作成し、動的型は初期化された free リストを作成しておく。下図 4-10 に示す。

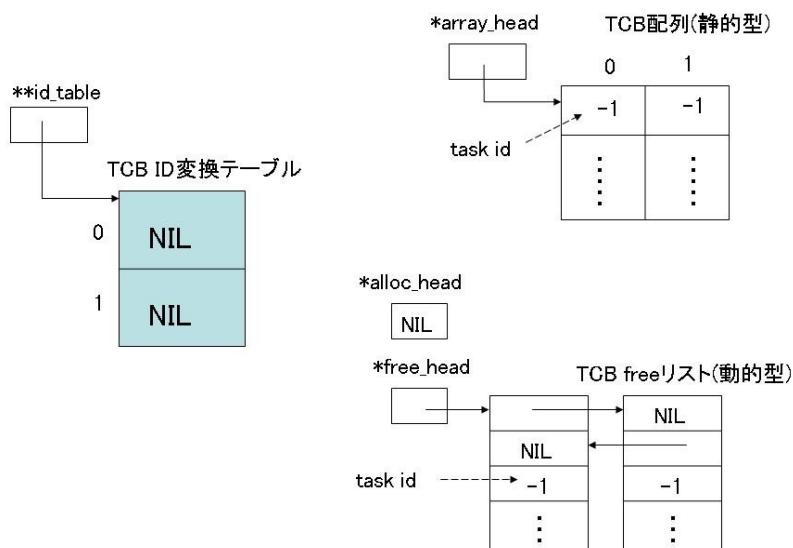


図4-10 init時の管理データ構造

acre_tsk()で動的型が指定された時は、alloc リストの head に free リストの head を代入し、free リストの head を一つ進める。ID0 が作成された状態を下図 4-11 に示す。

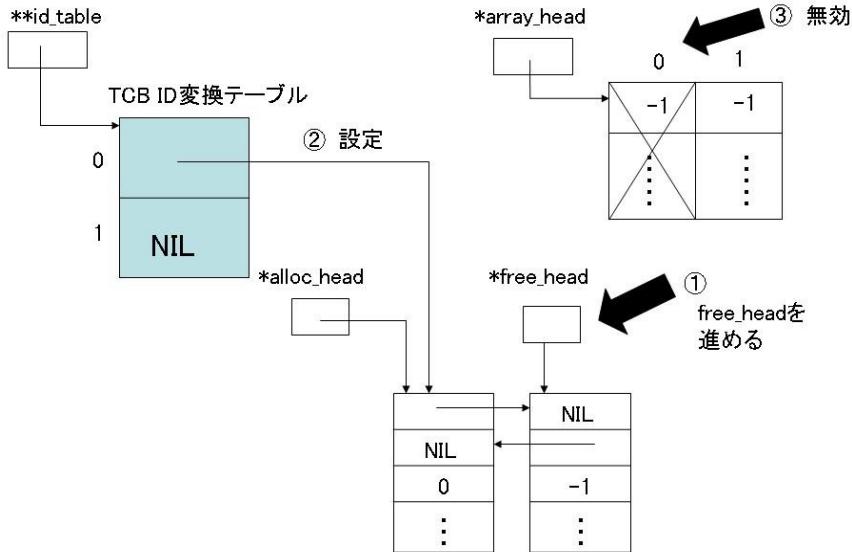


図4-11 タスクID0をcre_tsk()で生成した状態

del_tsk()で動的型が指定された時は、alloc リストから抜き取り、free リストの head へ挿入する。ID0 が排除した状態を下図 4-12 に示す。

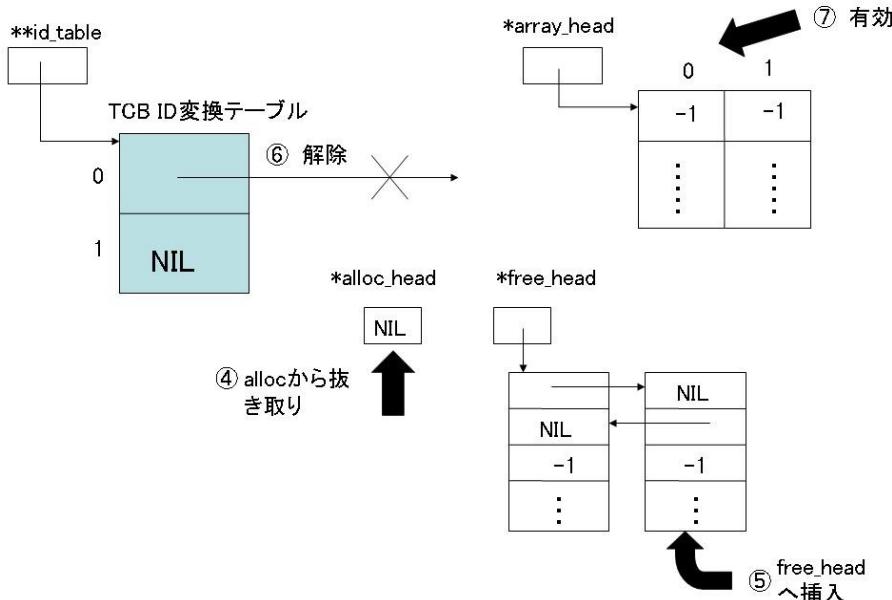


図4-12 タスクID0をdel_tsk()で排除した状態

(4.1)と同様に、タスク ID0～ID1 が使用されていて、ID が不足した時の状態を下図に示す。倍の ID 変換テーブルの領域をとってきて、古い ID 変換テーブルをコピーする。静的

型の可変長配列も倍の領域をとってきて、古い可変長配列をコピーする。動的型は古い ID 変換テーブルと同じ数の free リストの領域をとってきて、古い free リストの head へつなげる。状態を下図 4-13 に示す。

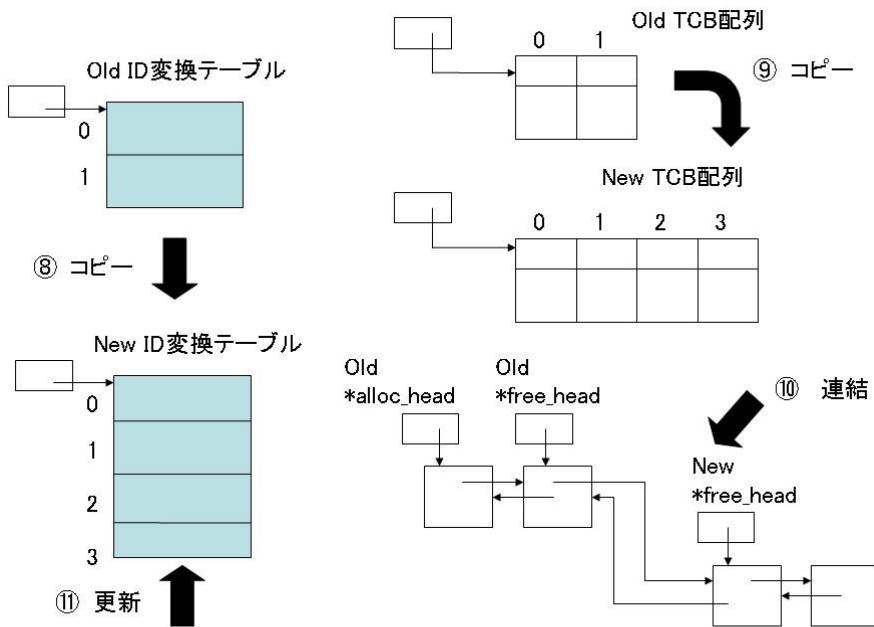


図4-13タスクID不足時の対応

生成時の free リストの抜き取り操作と ID 不足時より、(4.2)の方が効率が良いので本研究OSでは、(4.2)を採用した。(4.2)の方式を管理するための構造体をタスク情報構造体とし、図 4-14 に示す。

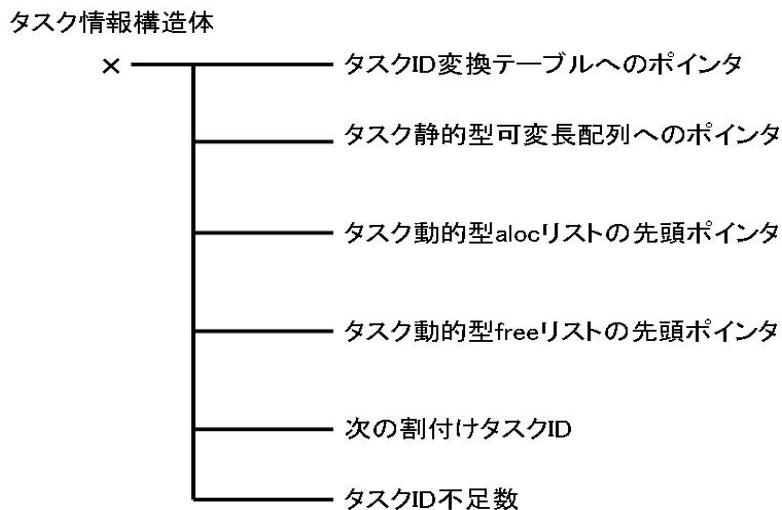


図4-14 タスク情報構造体

次の割付け ID は、値保持とシステムロールバックの関係(ローカル静的変数は使用不可 5.5.(1)参照)からタスク情報構造体にまとめた。ID 不足回数は可変長配列の大きさを知るためにメンバにもたせている。

5. 割込み管理

本節では割込みについて述べる。ターゲットハードウェアである H8 マイコンはベクタ割込み方式を採用し割込みコントローラを内蔵している。本研究 OS ではリアルタイム性を考慮しないので、多重割込み及び割込み優先度は未実装となる。また、割込みドリブン型 OS なので、割込みを契機にタスクを切替える(割込みを契機にタスクをレディー管理データ構造から抜き取り、タスクが待ちにならない場合、再度レディー管理データ構造に繋げる)。ただし、シリアル割込みではタスクの切り替えを行わない。これについては、5.2.(2)節で説明する。

実装した割込みはシステムコール割込み、シリアル送受信割込み、タイマ割込み、NMI 割込み、例外となる。シリアル出力の指定ができるため、シリアル送受信割込みハンドラはユーザが自由に作成及び登録できる実装とし、他の割込みハンドラは init タスク生成時の段階でカーネル内に組込む方式とした。そのため、ベクタは read/execiton メモリである ROM にマップされ登録できないので、DRAM 側にソフトウェアベクタを設け、このソフトウェアベクタへシリアル割込みハンドラを登録する実装とした。

5.1. 割込み管理データ構造

(1) 割込み発行時のカーネル状態

割込みでカーネルへ処理が移ると,カーネルはビッグカーネルロックをし,割込み無効モードとなる. 割込み無効モードとしたのは,多重割込み未実装だからである. よって,本研究OSの特権モードは割込み有効モード(主にタスク空間)と無効モード(主にカーネル空間)の2種類となる. 下図5-1に示す.

タスク空間

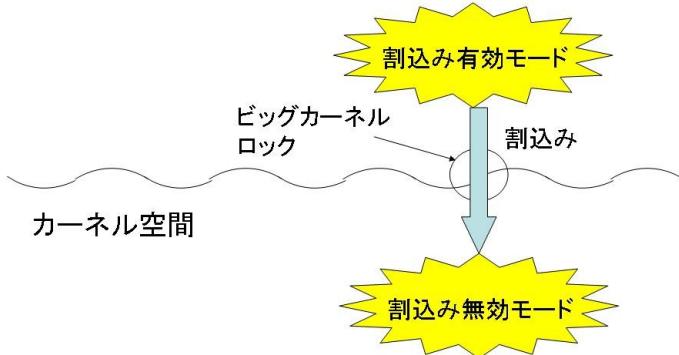


図5-1 割込み有効モードと無効モード

(2) 割込み発行のシーケンス

タスク及びターゲットハードウェアから割込みが発行されると,ROM側にマップしたベクタに信号が入り,タスクコンテキストが退避され,ソフトウェアベクタへ移行する. ここまででは,ROM側のブートローダが行うOSとの連携処理となる. ソフトウェアベクタへ移行後は,カーネルへ処理が移り,登録した各割込みハンドラを呼び,実行状態タスクをレディー管理データ構造から取り除く. システムコール割込みならば,その延長線上で割込みISRまたは非タスクコンテキストシステムコールを呼び,取り除いたタスクを再度レディー管理データ構造へ管理させる. そして,次に実行可能なタスクを決定するために,スケジューラを起動し,タスクを実行状態へ遷移させるディスパッチャを起動する(図3-3参照).

(3) 割込み時のタスクコンテキスト退避及び復旧とマルチタスク機能

退避及び復旧(復旧はシリアル送受割込みのみとなり,5.2.説で説明する)するタスクコンテキストはプログラムカウンタ,汎用レジスタ,フラグレジスタ,OS特権モードを記憶するモードレジスタとなる. これらをタスクごとに管理させるための領域をタスクスタックとしてメモリセグメントを定義した. タスクスタックはread/write/executionメモリであり,速度性を考え,DDR3SDRAMに配置した. タスクスタックへ退避した場所を4.2.(1)のTCBコンテキスト情報で管理する. この実装によりマルチタスク機能が実現できる. 下図5-2に示す.

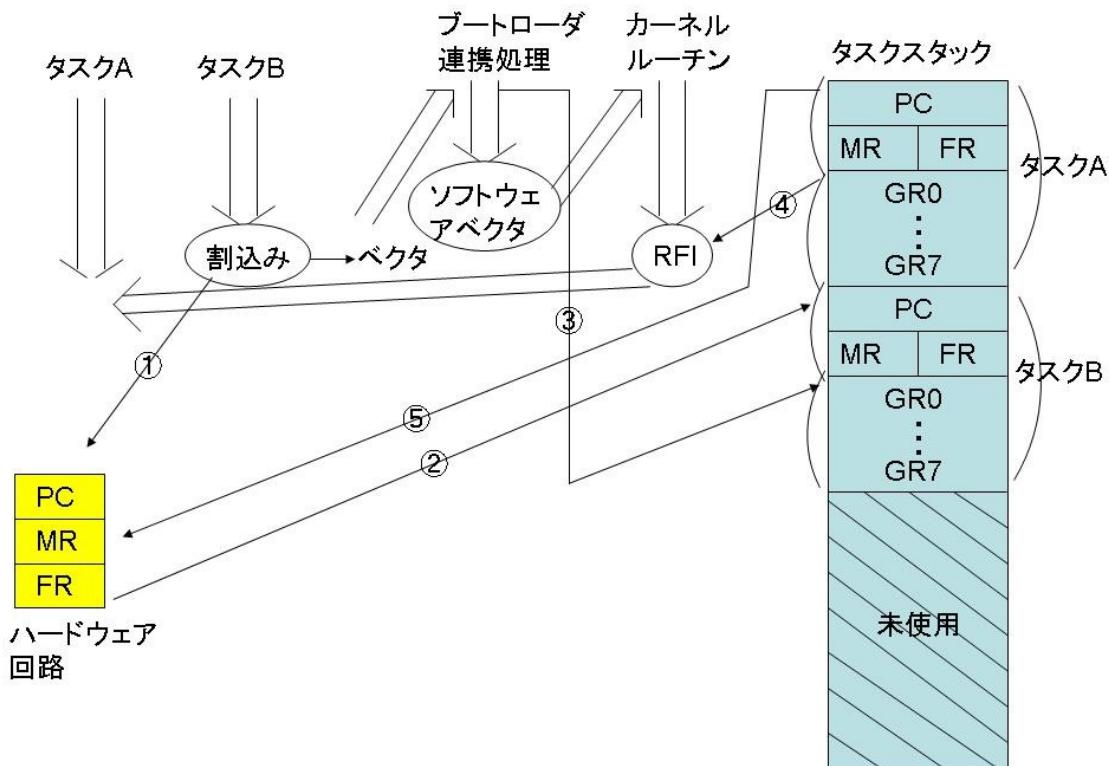


図5-2 マルチタスク機能によるコンテキスト退避と復旧

- ① 割込みをターゲットハードウェアが検知する。
- ② ハードウェア回路がプログラムカウンタ, モードレジスタ, フラグレジスタをタスクBのタスクstackへ退避する。
- ③ 汎用レジスタの退避をする。これはプログラムで行う。
- ④ 復帰命令より, 次に切替えるタスクの汎用レジスタをタスクstackから復旧させる。これはプログラムで行う。
- ⑤ ハードウェア回路がフラグレジスタ, モードレジスタ, プログラムカウンタを復旧させる。

なお, モードレジスタは特権モードを切替える前の割込み有効モードを退避させる。これにより, 復旧させタスクに実行を移す時に割込み有効モード引き継ぐ事ができる。また, ターゲットハードウェアはシャドーレジスタ機能はないので, 制御していない。

(4) タスクstack管理の問題点

現在のタスクstack管理は, タスクがstack獲得時, タスクstackメモリセグメントを下方伸長で切り出す方式としている。つまり, allocはつねにメモリの下方から行い, 下方より上にある未使用領域を再利用できない。よって, この実装だと, タスクstackが一部穴あき状態となり, メモリ効率が悪い。下図5-3に示す。

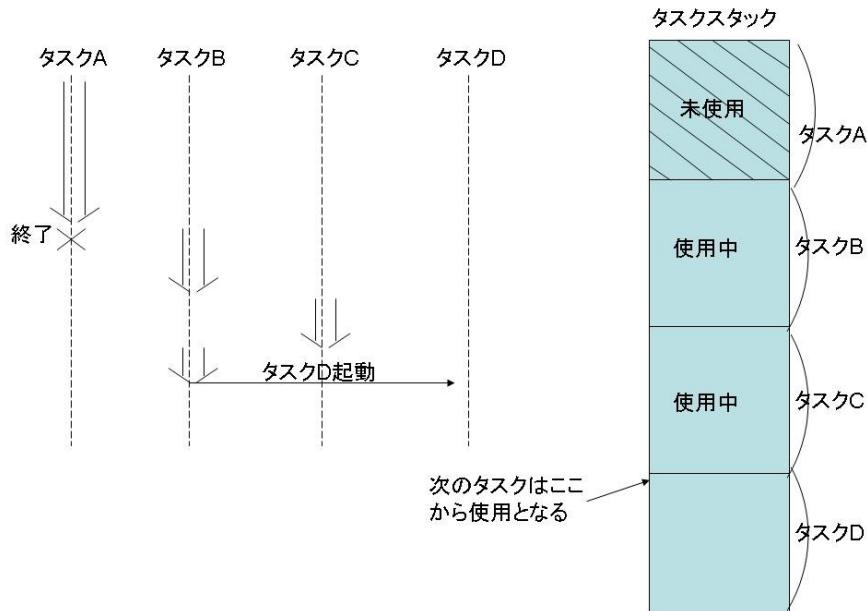


図5-3 タスクスタック一部穴あき状態

一部穴あきとなったメモリを再度使用できるようにタスクスタックはリンクドリストによって、管理させるべきである。これは、まだ未実装であり今後の課題とする。

5.2. 各割込み詳細

(1) システムコール割込み

システムコール割込みはトラップを発行して行う。トラップを使用して強制的に処理を切替えるので、パラメータ及びリターンパラメータの退避が必要となる。この退避領域をシステムコール構造体とした。

なお、システムコールのパラメータ情報をコピーする処理、5.2.(1-1)で説明するタスク構造体にシステムコール情報を登録する処理とトラップを発行する処理が、3.3.(3-3)のタスクコンテキスト用システムコール前処理に相当する。

(1-1) システムコール構造体

システムコールのパラメータ情報及びリターンパラメータ情報は单一のシステムコール構造体としてバッファ管理させる。システムコール構造体を下図5-4に示す。

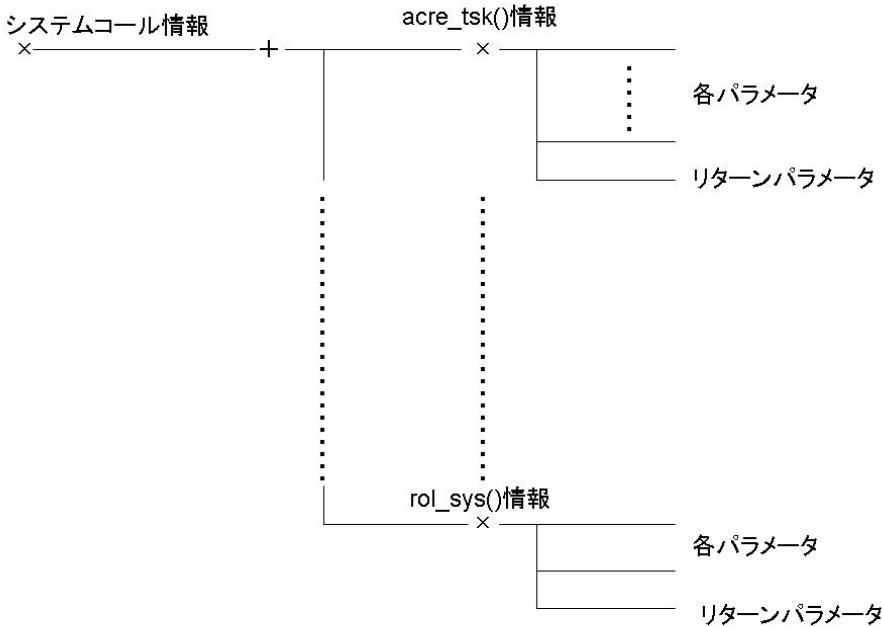


図5-4 システムコール構造体

なお,1つのタスクから,同時に2つ以上のシステムコール構造体が利用されることはないのでunion定義としている。そして,システムコール発行後は,この構造体のパラメータバッファとリターンパラメータバッファのポインタを4.2.(1)の図3-2のシステムコール情報で管理する。パラメータバッファのポインタを持つのは,カーネルルーチン内でシステムコールのパラメータをまとめて渡せるからである。リターンパラメータバッファのポインタを持つのは,他のタスクのシステムコールからそのリターンパラメータバッファを書き換える処理があるからである。

また,実装している54個のシステムコールに番号を割付け4.2.(1)の図3-2のシステムコールタイプで管理する。これにより,タスクが現在発行しているシステムコールの種類が判別できる。

(1-2) システムコールのデザイン

本研究OSはすべて参考文献[16], [17]にある動的API使用としているため,カーネルオブジェクト生成はコンフィギュレーションではなくシステムコールで行う。カーネルオブジェクト生成システムコールはパラメータ情報が多く,ユーザは使用しづらい。そのためパラメータ情報を格納した構造体を渡す仕様とした。

(1-3) システムコールのISR呼出し時のチェック

3.3節の図3-3に示したように,システムコールが発行されると延長でISRが発行される。システムコールによっては,ISRに処理が移る前にID変換処理を行うものがある。このときに,ユーザによって設定されたシステムコールパラメータ類が適切か,カーネルの状態やメモリ状態,各カーネルオブジェクトの状態等をチェックを行う。このチェックは

システムコールによって異なる。どのようなチェックがあるかは付録で述べる。

(2) シリアル送受信割込み

シリアル送受信割込みハンドラはシリアル入力を行う時に、ユーザがシステムコールで(`def_inh()` : ハンドラの登録)登録する。また、ターゲットハードウェアはキャラクタ型デバイスドライバのため、1文字受信の度に割込みが入る。1文字ごとにタスクの切換えを行うと、文字列の出力がわかりづらくなる。さらに、タスクコンテキストスイッチングにより、OSのオーバーヘッドが大きくなる。そのため、シリアル送受信割込みではタスクは切り替えない実装とした。

(3) タイマ割込み

タイマ割込みはハードタイマドライバ及びソフトタイマドライバで制御される。

(4) NMI 割込み

本研究OSでのNMI割込みはターゲットにNMI配線が施されており、外部スイッチによる発行となる。NMI割込みハンドラではinitタスク生成段階へロールバックさせる実装とした。この実装としたのは、処理をブートローダへ戻さず(リセットベクタはブートローダへ処理を移し、OSのリロードが必要)、タスクコンテキスト及び非タスクコンテキスト両方に応する組込みシステム暴走用として使用するためである。タスクコンテキストからNMI割込みが発行された時のシーケンスを図5-6に示す。なお、ロールバックに関しては、5.5.(5)節で述べる。

(5) 例外

本研究OSはマルチタスク機能があるため、ユーザタスクに不正がある場合はそのユーザタスクの実行処理を終了する必要がある。シグナルを簡素化し、トラップを使用する方式としてタスク例外処理機能を実装した。しかし、タスクごとに例外処理ハンドラを作成及び登録処理ができない。よって、他のタスクに終了要求を発行やタスクデッドライン通知ができない。また、例外処理モデルとして、CPU例外処理が未実装である。これは今後の課題とする。

5.3. 非タスクコンテキストシステムコールとサービスコール

(1) 非タスクコンテキスト用システムコール

タスクコンテキスト用システムコールはタスクから発行するものであり、過渡的な状態(カーネル実行中)、タスク独立部実行中状態(割込みハンドラ実行中)、準タスク部実行中(拡張SVSハンドラ実行中)からは呼べない(タスクコンテキスト情報の扱いがあるため)。また、本研究OSはユーザが登録できるシリアル割込みハンドラ及びアラーム機能、周期タイマ機能を実装している。よって、異なるコンテキストからもシステムコールが発行可能にしなくてはならない。これは、非タスクコンテキスト用システムコールとした(参考文献[16], [17]を参考)。

非タスクコンテキスト用システムコールの実装は、シリアル割込みハンドラ、アラーム

拡張ハンドラ、周期タイマ拡張ハンドラで使用が懸念され、タスクのスイッチングを必要とするシステムコールとした。具体的には、ista_tsk(タスクの起動), ichg_pri(タスク優先度変更), iwup_tsk(タスクの起床), irot_rdq(タスク優先度回転)となる。機能自体はシステムコールと同じとなる。3.3節の図3-3の非タスクコンテキスト用システムコール前処理では、非タスクコンテキストから呼ばれても不整合が起きないようにし、タスクのスイッチングをさせない処理をする。さらに、非タスクコンテキストシステムコール発行時を4.2.節の図4-2のシステムコール情報タイプでフラグ管理する。

(2) サービスコール

タスクコンテキスト用システムコールはOS内部でトラップを発行し、ソフトウェア割込みで行い、タスクを切替えていた。しかし、割込みで行うシステムコールは処理が重く、タスクを切替える必要がないシステムコールがある。そのため、応答性を高めたシステムコールとしてトラップを使用せずに割込みを行い、タスクを切替えない方式としてサービスコールを実装した。

5.4. ディスパッチャ管理データ構造

本節では、ディスパッチャ管理データ構造について述べる。本研究OSでは、ディスパッチャ禁止状態とディスパッチ許可状態がある。OS内部構造簡素化のため、この2つの状態は、ディスパッチャ状態フラグとし、ディスパッチャ起動番地とともに構造体(2つのメンバを持つ)として管理させる方式とした。3.3節の図3-3にあるディスパッチャ呼出しでは、カーネルがディスパッチャ状態フラグを検査し、ディスパッチャ許可状態ならば、ディスパッチャ起動番地にアクセスし、タスクのディスパッチを行う。

タスクディスパッチの処理は、スケジューラによって変化せず、5.1.節(3)の④, ⑤を行い、タスクスタートアップ関数を呼び出す(10.3.(1)も参考していただきたい)。

5.5. システムロールバック機能

本節では、システムロールバック機能について述べる。システムのロールバックはユーザが明示的に行えるように、本研究OS実装独自のrol_sys()システムコールを用いて行う方式とした。このシステムコールは、システム暴走用として使用するか、スケジューラ動的切替えシステムコール(6.5.節で説明)と併用して使用する。スケジューラ動的切替えの柔軟化と、タスクのレスポンスを見やすくするため、属性にはタスク実行中にロールバックする方式とタスク終了後にロールバックする方式が指定できる実装とした。ISR内では、ロールバック処理を行う。ロールバック処理で考慮すべき点を(1)に示す。

(1) ロールバック時の考慮

ロールバック後はOS内で不整合が起こらないようにしなくてはならない。下記の(1-1)～(1-3)の問題点があり、それぞれ対応を行った。

(1-1) 動的メモリ

問題・・・動的メモリを使用したままロールバックすると、OSのメモリプール(動的メモリ)が枯渇または不整合となる。

対応・・・ロールバック処理で使用している動的メモリの解放を行う。

(1-2) グローバル変数及び静的グローバル変数

問題・・・前のグローバル変数と静的変数の状態が引き継がれるので、不整合が起きる。

対応・・・ロールバック処理で使用しているグローバル変数と静的グローバル変数をすべて0初期化しておく。

(1-3) 静的ローカル変数

問題・・・(1-2)で述べたように、不整合が起きる。

対応・・・使用しない。

また、rol_sys()システムコールはNMI発行と同じ処理となる。タスクコンテキストからNMIが発行された時のシーケンスを図5-5に示す。

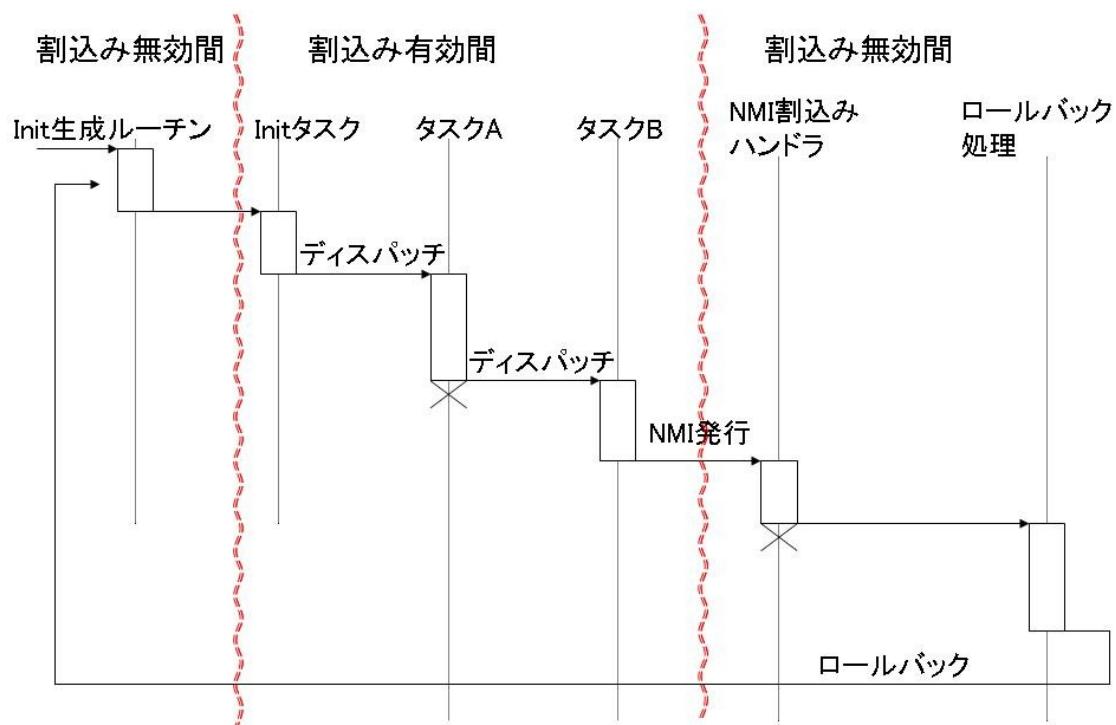


図5-5 タスクコンテキストから発行されたNMIのシーケンス

図5-5はタスクB実行中にNMIが発行され、その延長でNMI割込みハンドラが起動する。ロールバック処理では、(1)の処理を行う。

なお、内部構造簡素化のため、rol_sys()システムコールはNMI割込みハンドラとinitタスク生成段階へのロールバック処理を共有する。

6. スケジューラ及びレディー管理

本節ではスケジューラについて述べる。また、スケジューラはタスクレディー管理とも密接に関わるので、レディー管理について 6.2. 節で述べる。本研究 OS では複数のスケジューラを実装した。下記の①と②に示す。

① 既存の方式・・・First Come First Saved, ラウンドロビンスケジューリング, 優先度スケジューリング, ラウンドロビン×優先度スケジューリング, Multilevel Feedback Queue, 簡易O(1)スケジューリング, Fair Scheduling, Rate Monotonic, Deadline Monotonic, Earliest Deadline First, Least Laxity First, μ ITRON型ラウンドロビンスケジューリングとなる.

② 新規の方式・・・Priority Fair Scheduling

本節ではこれらのスケジューラ詳細を6.1.節で述べる。6.3.節で特徴及び得失を考察し、既存の方式であるスケジューラの問題点を述べ、6.4.節で問題解決として新規スケジューリングの詳細を述べる。

また、13個のスケジューラはメモリヘリロードせずに動的に13個のスケジューラを切替えることができる。これは、6.5.節で述べる。

(1) スケジューリングの分類

スケジューリングは、さまざまな観点から複数の分類法がある。ここでは、一般的にリアルタイム性を意識した組込みOS等で使用されるスケジューリング、公平性を意識した汎用系OS等で使用されるスケジューリング、上記の両OS共通で使用されるスケジューリングというOS環境下で分類する。図6-1に示す。



図6-1 スケジューリングの分類

また、スケジューラによってはソフトタイマドライバと連携処理がある。連携処理に関しては、6.1.(4)で述べる。

6.1. 各スケジューラ詳細

(1) 両OS共通して使用されるスケジューリング

(1-1) First Come First Served スケジューリング(以下FCFS)

このスケジューリングは、レディーキューにタスクが挿入された順(実行要求の順)にスケジューリングする方式である。

本研究OSの実装としては、レディー構造は6.2.(2)とし、FCFSスケジューラの起動では、レディーキューにある先頭のTCBを次に実行するタスクとして設定し、ディスパッチャへ引き渡す。また、タスクの優先度は設定せず、優先度に依存するシステムコール処理は未サポートエラーとする。具体的には、優先度取得システムコール、優先度変更システムコール、優先度回転システムコール、優先度逆転機構(すべての優先度逆転解消プロトコル)、他タスク生成システムコールやタスク強制終了システムコールなどである。

(1-2) ラウンドロビンスケジューリング(以下RR)

このスケジューリングは、FCFSスケジューリングにスライスタイムを加えたスケジューリング方式である。

本研究OSの実装としては、(1-1)と同様にレディー構造は6.2.(2)とし、RRスケジューラ起動では、レディーキューの先頭のTCBを実行対象タスクとする。また、タスクの優先度は設定せず、優先度に依存するシステムコール及びOS機構は未サポートエラーとする。

また、RRでは4.2.(1)節の図4-2のTCBのスケジューラ依存情報はスライスタイムとなる。

(1-3) 優先度スケジューリングの拡張(優先度スケジューリング以下PR)

このスケジューリングはタスクに固定優先度を割り付け、優先度の高いタスクから実行していくスケジューリング方式でオープンソースOSに実装されていた。固定優先度の割付けはユーザがシステムコールを用いて行う一般的な方式とした。実装としては、レディー構造は6.2.(3)とし、PSスケジューラ起動では、優先度レベルの高いレディーキューにある先頭のTCBを次に実行するタスクとして設定し、ディスパッチャへ引き渡す。同優先度レベルはFCFSスケジューリングとなる。また、タスクスライスタイムは設けていない(タスクスライスタイム付きは(1-4)となる)。

優先度スケジューリングの拡張として、優先度レベルのレディーキューの検索を高速化するために、ビットマップテーブルを用意し、ビットサーチアルゴリズムを実装した。なお、本研究OSであるターゲットハードウェアはインストラクションセットとして、ビットサーチ命令を持っていない。また、このスケジューリングは優先度があるので、優先度に依存するシステムコールを発行許可する。

(1-4) ラウンドロビン×優先度スケジューリング(以下RPS)

このスケジューリングは、優先度スケジューリングに対してタスクスライスタイムを加えたスケジューリング方式である。

本研究OSの実装としては、(1-3)と同様に固定優先度があり、レディー構造は6.2.(3)とし、RPSスケジューラ起動では、優先度レベルの高いレディーキューの先頭TCBを実行対象タスクとする。同優先度レベルはラウンドロビンスケジューリングとなる。

また、RPSでは4.2.(1)節の図4-2のTCBのスケジューラ依存情報はスライスタイムとなる。

(2) 公平性を意識した汎用系OS等で使用されるスケジューリング

(2-1) Multilevel Feedback Queueスケジューリング(以下MFQ)

このスケジューリングは、BSDカーネルの4BSDスケジューラを学習向けOS用に簡素化したものである(参考文献[34], [35]参照)。(1-4)のスケジューリングを拡張したもので、優先度の高いタスクほど短いスライスタイムを割付け、タイムアウトしたタスクに対して優先度を一つ下げていくスケジューリングである。

本研究OSの実装としては、(1-4)と同様に固定優先度があり、レディー構造は6.2.(3)とする。最低優先度タスクがタイムアウトした場合は、優先度無効範囲となってしまうため、優先度は下げる。

(2-2) 簡易O(1)スケジューリング(以下O(1))

このスケジューリングはLinuxカーネル2.6~2.6.22に実装されているスケジューリングである。

本研究OSの実装としては、参考文献[24], [25], [27]を参考に、構造を簡素化し、データ構造と基本スライスタイムのみを実装した(簡素化したため、簡易O(1)スケジューリングとした)。(1-4)を拡張したもので、優先度の高いタイムアウトしたタスクよりも優先度の低いタイムアウトしていないタスクを優先的にスケジューリングしていく方式である。(1-4)と同様に固定優先度があり、レディー構造は6.2.(4)とする。O(1)スケジューラでは、実行タスクの決定はactivキューから行い、activキューが空になったら、activキューとexpiredキューを入れ替え、再度activキューから実行タスクを決定する実装である。

なお、参考文献[24]のLinuxカーネル2.6.11で実装されている動的優先度、CPUバウンドとI/Oバウンドの発見アルゴリズムにヒューリスティックを用いる方法、プロセスの優先度別スケジューリング(スケジューリングクラス)は実装していない。

また、O(1)では、4.2.(1)節の図4-2のTCBのスケジューラ依存情報はレディーキューフラグとなる。レディーキューフラグによって、タスクが存在するレディーキュー(activキュー、expiredキュー)の区別が可能である。

(2-3) Fair Scheduling(以下FRS)

このスケジューリングは、ヒューリスティックを使用せずにCPUバウンドタスクとI/Oバウンドタスクの実行時間(タスク実行状態の時間)をFairness(公平)に扱う。汎用OSなどで、実装されるスケジューリングである。

本研究OSの実装としては、固定優先度を持たずに(ユーザは優先度の設定をしない)、タスク総実行時間が少ないタスクほど、スケジューラが動的に優先度を割り付け、優先度の高い順にスケジューリングする。レディー構造は6.2.(5)とする。また、固定優先度を持っていないので、優先度に依存するシステムコール及びOS機構は未サポートエラーとする。

また、FRSでは4.2.(1)節の図4-2のTCBのスケジューラ依存情報は仮想実行時間となる。

(3) リアルタイム性を意識した組込みOS等で使用されるスケジューリング

(3-1) デッドライン保障によるリアルタイム属性

一般論では、リアルタイム性を意識したスケジューリングはデッドライン保障が重要となる。

本研究OSではタスクがデッドラインをミスした場合は、スケジューラごとに用意したデッドラインミスハンドラでOSをフリーズさせる。フリーズさせるのはdomino effect(将棋倒し現象)を防ぐためである。デッドラインミスハンドラを呼ぶタイミング(デッドライン保障属性)として、次の(3-1-1).①～(3-1-1).④の方式が考えられる。

(3-1-1).① 石橋をたたいて渡る型:Guarantee型でタスク生成時

図6-2に示す。

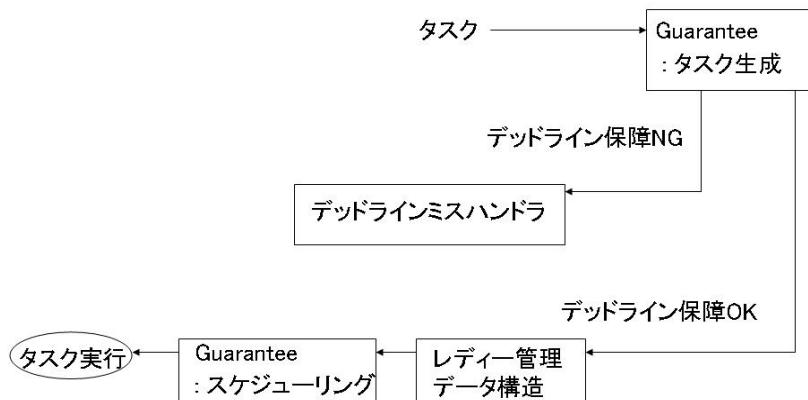


図6-2 Guarantee型でタスク生成時

タスク生成の段階で呼ぶと、どこまでタスクが実行されたかレスポンスがわかりづらくなる。

(3-1-1).② 石橋をたたいて渡る型:Guarantee型でスケジュール時
図6-3に示す。

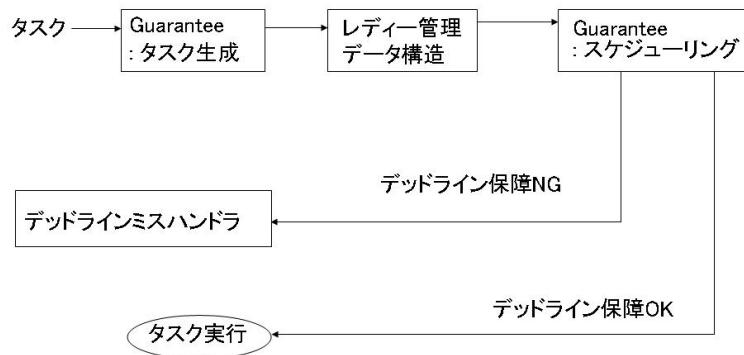


図6-3 Guarantee型でスケジュール時

スケジュールの段階で呼ぶと、どこまでタスクが実行されたかのレスポンスがわかりやすい。

(3-1-1).③ 最善をつくす型:Best Effect型

図6-4に示す。

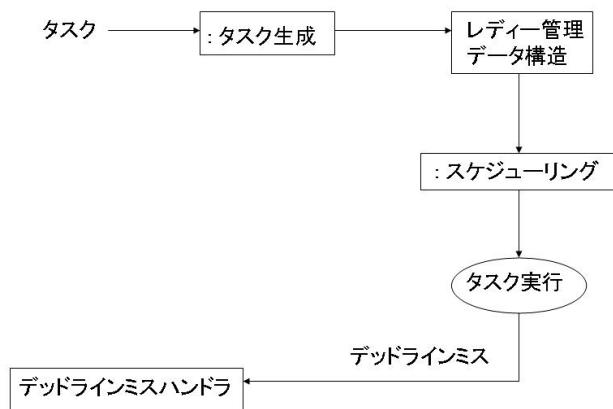


図6-4 Best Effect型

タスク実行中に呼ぶと、レスポンスはわかりやすい。Best Effect型はEDFスケジューリング及びLLFスケジューリングで使用している

(3-1-2).④ 敗者復活型:Robust型

図6-5に示す。

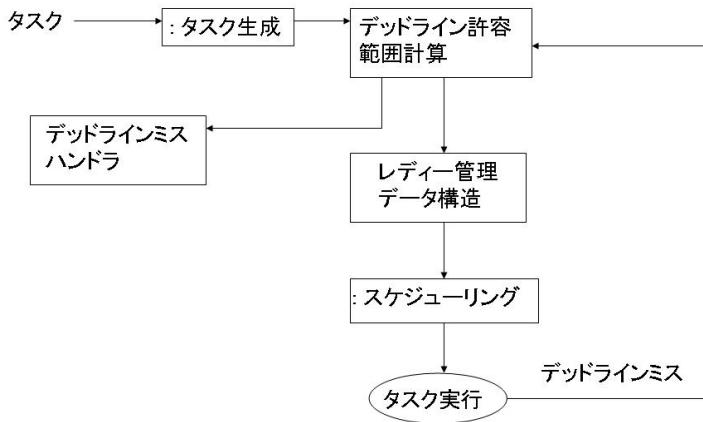


図6-5 Robust型

この保障属性は、タスクがデッドラインをオーバーしないうちは何度もリトライするので、(3-1-1).①よりタスクのレスポンスがわかりづらくなる。

学習向けOSなので、(3-1-1).②を(3-2)のスケジューリング(3-3)のスケジューリングに適用し、(3-1-1).③を(3-4)のスケジューリング及び(3-5)のスケジューリングに適用した。

(3-2) Rate Monotonic(以下RM)

このスケジューリングは周期タスクセットに対して、起動周期の短いタスクほど高い固定優先度を割り付けるスケジューリングである。固定優先度の割付けは、ユーザが4.4.(1)のacre_tsk()システムコールまたは4.4.(10)のrun_tsk()システムコールで行う。相対デッドラインは周期と同じものとし、次の周期の起動時刻までに実行を終了させる。

本研究OSの実装としては、(1-3)と同様に固定優先度があり、レディー構造は6.2.(3)とし、RMスケジューラ起動では、優先度レベルの高いレディーキューの先頭TCBを実行対象タスクとする。また、RMスケジューラ内で不整合が起こるので、優先度に依存するシステムコール及びOS機構は未サポートエラーとする。

(3-2-1) RM展開スケジュール

相対デッドラインをオーバーするかは、タスク生成時の相対デッドラインと起動周期で判断できることが証明されている。具体的には、全タスク起動周期の最小公倍数時間と全タスク実行時間の最小公倍数時間で違反がないかである。

本研究OSの実装としては、では起動周期と実行時間の最小公倍数時間を求めるのに、ユークリッド互除法のアルゴリズムを適用した。

また、RMでは、4.2.(1)節の図4-2のTCBのスケジューラ依存情報は実行時間と周期となる。

(3-3) Deadline Monotonic(以下 DM)

このスケジューリングは周期タスクセットに対して、相対デッドラインの短いタスクほどユーザがシステムコールで高い固定優先度を割り付ける。(3-1)と異なるところは、相対デッドラインと周期は異なるものとし、相対デッドラインまでに実行を終了させる。

本研究OSの実装としては、(3-1)と同様である。

また、DMでは、4.2.(1)節の図4-2のTCBのスケジューラ依存情報は実行時間とデッドラインとなる。

(3-2-1) DM展開スケジュール

本研究OSの実装としては、DM展開スケジュールは全タスクの相対デッドラインの最小公倍数時間と全タスクのデッドライン時間で行う。

(3-4) Earliest Deadline First(以下 EDF)

このスケジューリングは、非周期的及び周期的に起動されるタスクについてデッドラインの早い順に動的に優先度を割付けるスケジューリングである。

本研究のOS実装としては、固定優先度は使用せず、動的優先度のみを使用する。動的優先度はユーザがシステムコールで割付けるのではなく、スケジューラがデッドラインの早い順に動的に割付ける。レディー構造は6.2.(5)とし、スケジューラが割付けた動的優先度の最も高いTCBを次に実行するタスクとし、ディスペッチャへ引き渡す。また、固定優先度を持っていないので、優先度に依存するシステムコール及びOS機構は未サポートエラーとする。

EDFでは、4.2.(1)節の図4-2のTCBのスケジューラ依存情報はデッドラインとタイマップロックポインタとなる。

(3-4-1) EDF展開スケジュール

本研究OSの実装としては、EDFのスケジュール可能性判断はデッドラインのみで行い、タイマ機構を使用する。

(3-5) Least Laxity Firstスケジューリング(以下 LLF)

このスケジューリングは、非周期的及び周期的に起動されるタスクについて余裕時刻の早い順に動的に高い優先度を割付けるスケジューリングである。.

本研究OSの実装としては、(3-4)と同様に、動的優先度のみを使用し、スケジューラが余裕時刻の早い順に動的に割付ける。

LLFでは、4.2.(1)節の図4-2のTCBのスケジューラ依存情報は余裕時刻とタイマプロックポインタとなる。

(3-5-1) LLF展開スケジュール

本研究OSの実装としては、LLFのスケジュール可能性判断は余裕時刻のみで行い、タイマ機構を使用する。

(3-6) μ ITRON型ラウンドロビンスケジューリング

このスケジューリングは、優先度スケジューリング環境下(優先度レベルでのレディー構造時)でラウンドロビンスケジューリングを行う μ ITRON準拠OSで使用されるスケジューリングである。参考文献[22]あるtoppers/ASPカーネルで実装されている方式で構造を簡単化せずに、本研究OSに実装した。カーネルではなく、ユーザがrot_rdq()システムコールを本研究OSの周期タイマ機能を用いて、周期的に発行する。よって、周期タイマ値がタスクスライスタイムとなる。このスケジューリングのサンプルとしてユーザタスクライブラリに実装している。

6.2. スケジューラ管理データ構造

(1) スケジューラ構造体

スケジューラを制御させる構造体を下図6-6に示す。なお、スケジューラはOS稼動中は必ず1つしか使用できないため、各スケジューラが個別に持つべき情報はunion定義している。

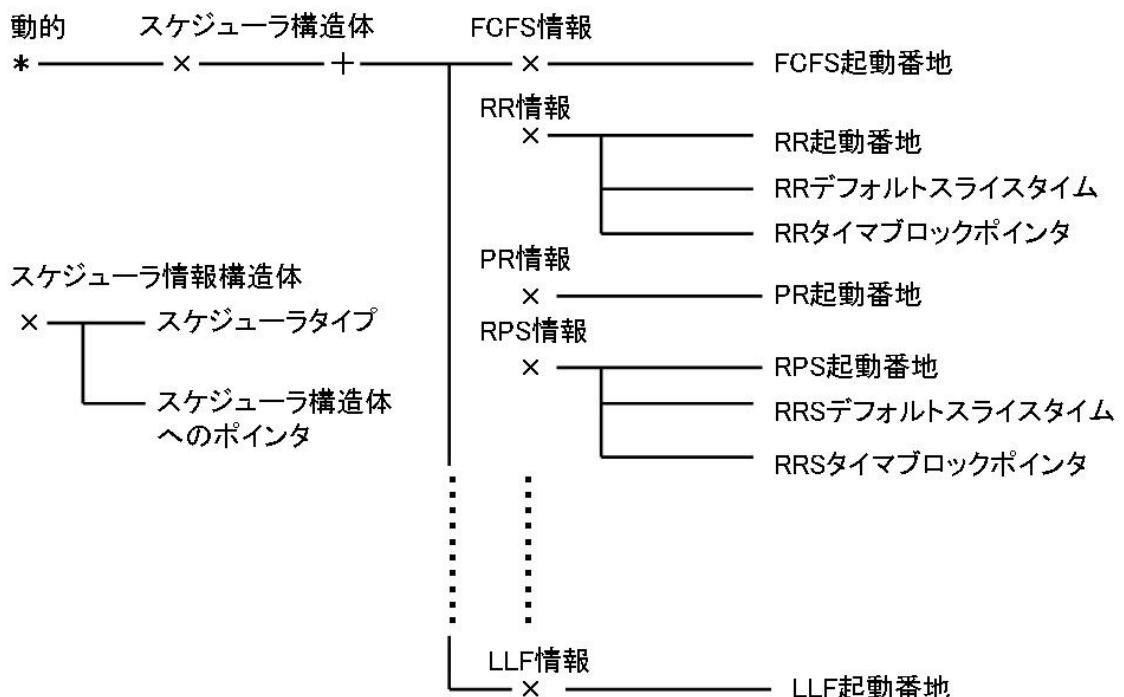


図6-6 スケジューラ構造体とスケジューラ情報構造体

図6-6のスケジューラタイプで現在有効化されているスケジューラ番号を登録しておく。

スケジューラを起動する時は、このスケジューラ番号を参照し、図のスケジューラ起動番地へアクセスする方式である。

(2) タイマ機構を使用するスケジューラ

本研究OSでは、スケジューラによってタスクスライスタイムの決定及びタイマ使用要求と使用解除が異なる。分類を図6-7に示す。なお、タイムスライスの制御は chg_slt()システムコール及びget_slt()システムコールで行う(4.4.(8)と4.4.(9)で説明済み)。なお、意味がないため init タスクはスライスタイムで切っていかない。

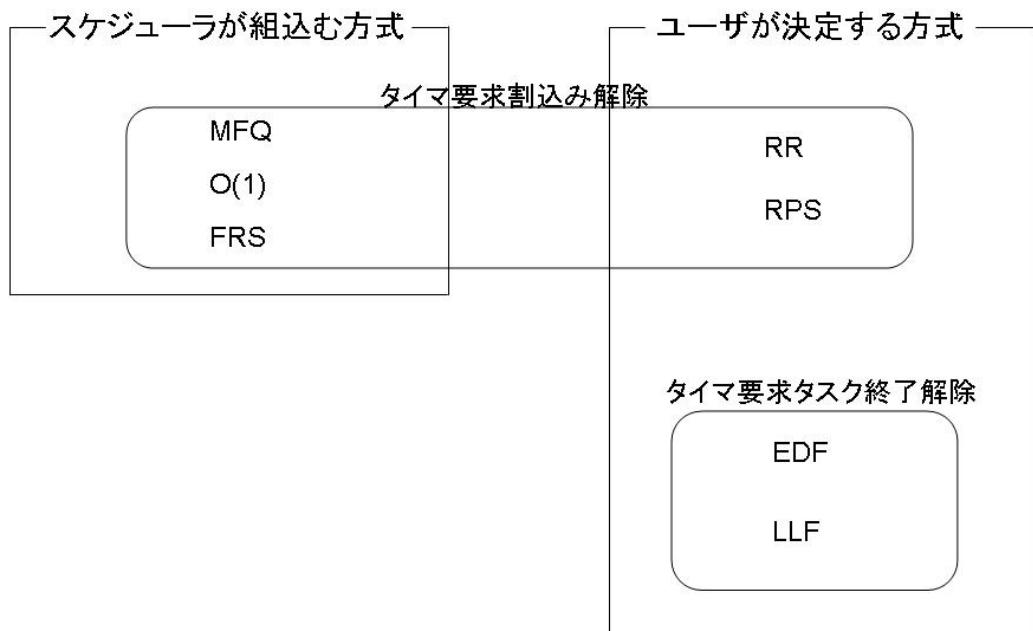


図6-7 タスクスライスタイムの決定及びタイマ使用要求と使用解除

① スケジューラが決定する方式

具体的には、ユーザがシステムコールで制御せずに、すべてのタスクスライスタイムをスケジューラが実行タスクを決定するときに、タスクに組込む方式である。よって、スケジューラがタスクスライスタイム値または重み値と使用するタイマコントロールブロック情報を図のスケジューラ構造体で別途持つておく方式である。

② ユーザが決定する方式

具体的には、ユーザが acre_tsk()システムコールまたは run_tsk()システムコールで図4-2のタスク構造体のメンバであるスライスタイムで管理する。よって、スケジューラ構造体ではタスクタイムスライス値とタイマコントロールブロック情報は持たない。

なお、本研究OSは割込みでタスクを切り替える。タイマ割込みでない場合またはスケジ

ューラ以外のタイマ割込みの場合、①と②ともにカーネルがタイマ解除処理を行う。

③ タイマ要求を割込み時で解除

具体的には、スケジュール段階でタイマ機構へタイマ使用要求を発行し、タスクを切替える割込み発生でタイマ機構へタイマの使用解除を発行する。

④ タイマ要求をタスク終了時で解除

具体的には、スケジュール段階でタイマ機構へタイマ使用要求を発行し、タスク終了である ext_tsk()システムコールまたは、タスク終了と排除である exd_tsk()システムコールでタイマ機構へタイマの使用解除を発行する。

(4-1) MFQ のタイマ周り

MFQ は、タスクスライスタイムの決定は 6.1.(4).① の方式であり、タイマ解除処理は 6.1.(4).③ に方式となる。MFQ を①としたのは、タスク優先度によってタスクスライスタイム大小が決まるからである。本研究 OS は学習向け OS のため、スケジューラによるタスクスライスタイムの計算は、タスク優先度と重みを乗算する実装とした。

(4-2) O(1) のタイマ周り

O(1) は、タスクスライスタイムの決定は 6.1.(4).① の方式であり、タイマ解除処理は 6.1.(4).③ に方式となる。スケジューラによるスライスタイムの計算は、参考文献[24]の Linux カーネル 2.6.11 と同様とした。ただし、ターゲットハードウェアはクロックが遅いので、重みを乗算させる方式としている。

(4-3) FRS のタイマ周り

FRS は、タスクスライスタイムの決定は 6.1.(4).① の方式であり、タイマ解除処理は 6.1.(4).③ に方式となる。スケジューラの構造を簡単化するために、タスクスライスタイムは全タスクで一定時間とした。

(4-4) RR のタイマ周り

RR は、タスクスライスタイムの決定は 6.1.(4).② の方式であり、タイマ解除処理は 6.1.(4).③ に方式となる。6.1.(4).② の方式を採用しているが、スケジューラはデフォルトのスライスタイムを持ち、一時的に使用する。それは、タスクを生成及び起動して、タイムスライスタイムをタスクに設定するには、2つのシステムコールを使用する。よって、タスク生成及び起動システムコールとタスクスライスタイム変更システムコールの実行区間ではタスクスライスタイムは未設定となるからである。

(4-5) RPS のタイマ周り

(4-4) と同様である。

(4-6) EDF のタイマ周り

EDF はタスクスライスタイムの決定は 6.1.(4).② の方式であり、タイマ解除処理は 6.1.(4).④ の方式となる。タスクスライスタイムは 4.2.(1) の TCB メンバであるデッドラインをタスクスライスタイムとして扱う。この方法により、タスクがタイムアウトするとデッドラインをミスする事となる。つまり、デッドラインミスハンドラはタイムアウトルーチ

ンから呼ばれる。

(4-7) LLF のタイマ周り

LLF は 6.1.(4).②の中の 6.1.(4).④に含まれる。実装としては、(4-6)と同様である。ただし、デッドラインではなく余裕時刻をタスクスライスタイムとして扱う。

6.2. 各レディー管理データ構造

(1) レディー管理におけるスケジューリングクラス

本研究OSでは、すべてのスケジューラにおけるレディー管理は、init タスクとユーザタスクとで2つに切り分けている。切り分けたのは、本研究OSには動的スケジューラが実装しているため、動的スケジューラ適用時に init タスクかユーザタスクかのチェック処理が必要なくなるからである。2つのスケジューリングクラスは init スケジューリングクラス、ユーザタスクスケジューリングクラスとした。init スケジューリングクラスは 6.2.(2)の図のメンバである init クラスで、ユーザタスククラスは 6.2.(2)の図のメンバであるユーザタスククラスで管理させる。ユーザタスククラスから init タスククラスへの切替え時期は、ユーザタスクがすべて終了後となる。

(2) レディー管理構造体

レディーを制御させる構造体を下図 6-8 に示す。スケジューラと同様に、OS稼動中は必ず 6.6.(2)～6.2.(5)のデータ構造は1つに決まるため、各レディーが個別に持つべき情報は union 定義としている。

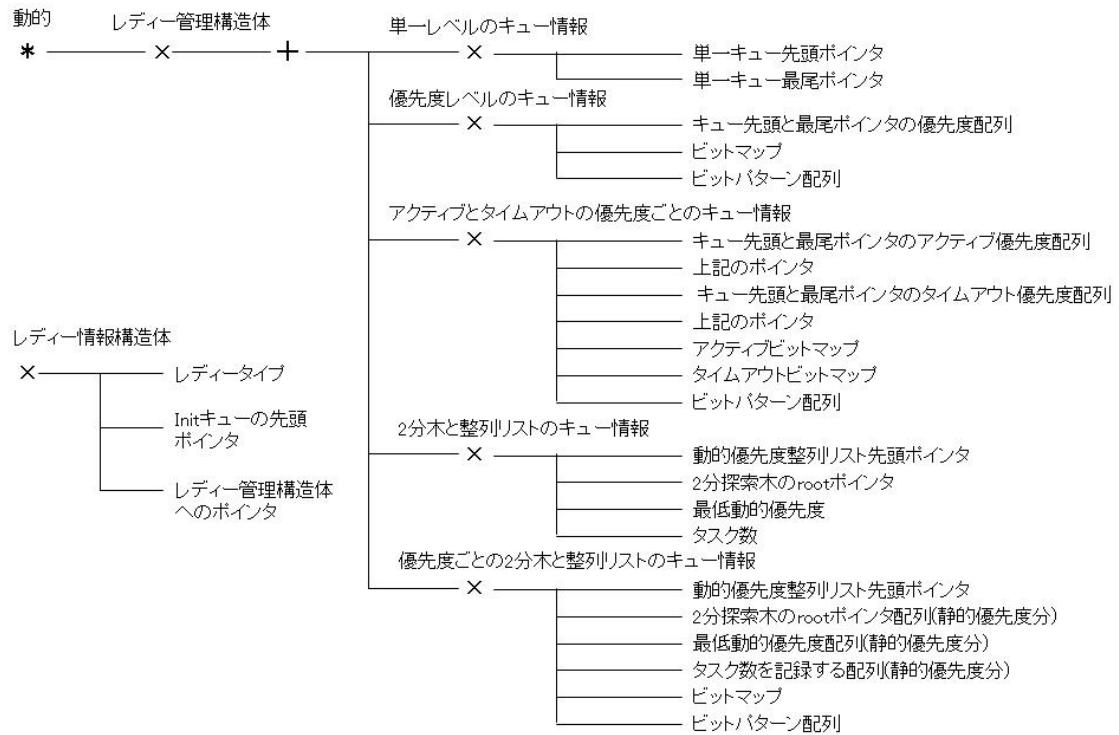


図6-8 レディー管理構造体とレディー情報構造体

図6-8のレディータイプで現在有効化されているレディー番号を登録しておく。レディー管理データ構造を操作させる時は、このレディー番号を参照し、図のレディー管理データ構造操作関数の起動番地へアクセスする方式である。なお、レディー管理データ構造操作関数は、実行状態タスクをレディー管理データ構造から抜き取る処理、指定されたタスクをレディー管理データ構造から抜き取る処理、実行状態タスクをレディー管理データ構造へ繋げる処理となる。

(3) 単一のレディーキュー

主に、FCFSとRRで使用されるレディー構造である。単一のレディーキューとしたのは、FCFSとRRは固定優先度がなくかつ、本研究OSはマルチタスクOSのため、並列的にタスク起動要求が発行されるからである。单一のレディー構造を図6-9に示す。

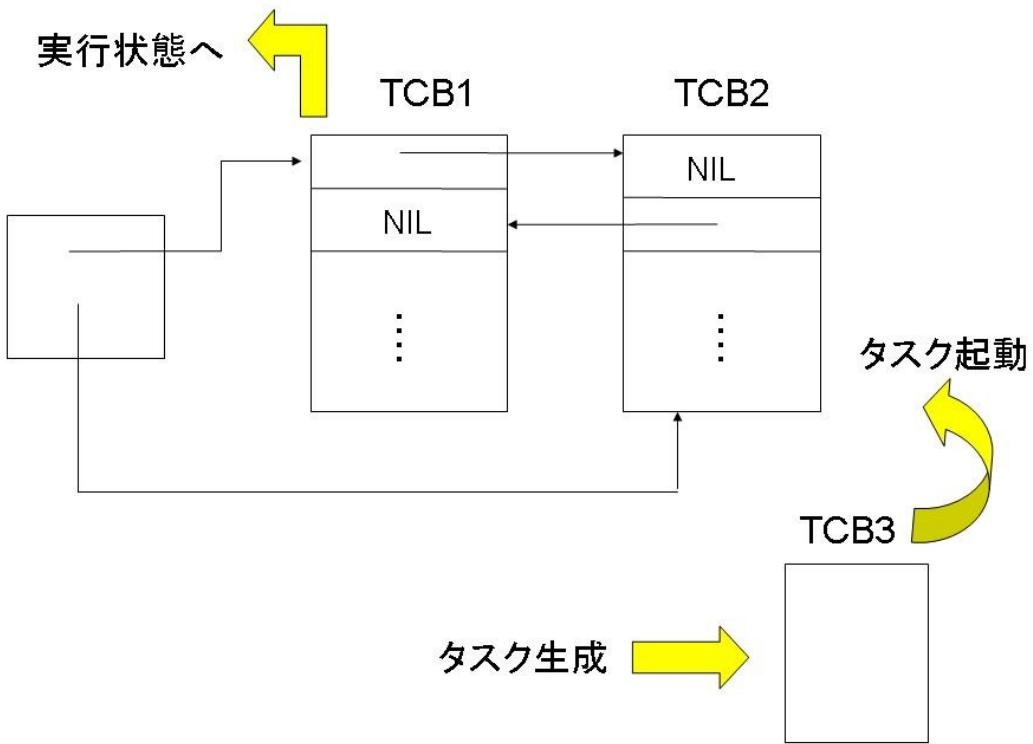


図6-9 単一のレディーキュー

このレディー管理データ構造では、TCBを接続するための先頭ポインタと最尾ポインタを6.2.(2)の図のレディー管理構造体で個別に持つ。

(4) 優先度レベルのレディーキュー

この方式は、PSスケジューリングが使用するレディー構造としてオープンソースOSに実装されていた。

本研究OSでは、RPS, MFQ, RM, DMスケジューリングで使用できるようにした。また、ターゲットハードウェアはインストラクションセットにビットサーチ命令を持っていない配列検索計算量を一定とするために、優先度配列に対応したビットマップテーブルを割付けビットサーチアルゴリズムを適用した。ビットサーチアルゴリズムは2分探索を応用した方式で実装した。具体的には、ビットマップテーブル半分ずつシフトしていく、残り4ビットでビットパターンテーブルとつき合わせる方式となる。このビットサーチアルゴリズム周りにバグがあり、現在では固定優先度の上限を32個しかできない。上限が設定可能を今後の課題とする。

ビットマップテーブルと拡張した優先度レベルでのレディーキュー構造を図6-10に示す。

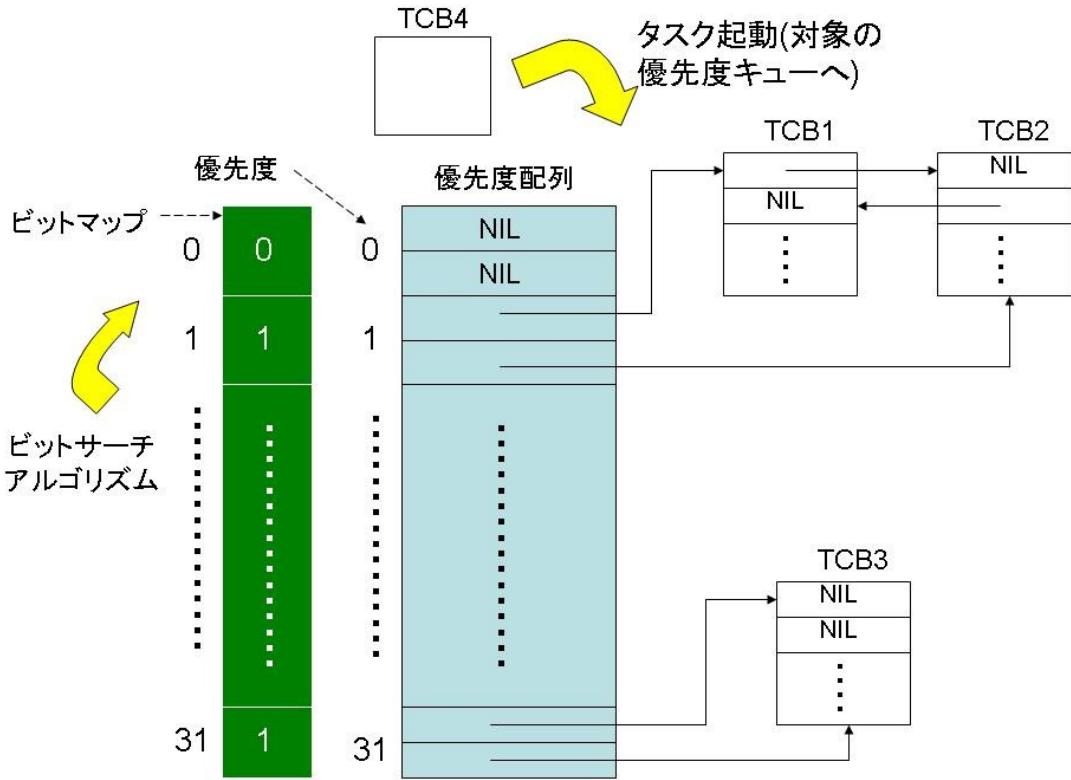


図6-10 優先度レベルのレディーキュー

このレディー管理データ構造では、優先度数分の TCB を接続するための先頭ポインタと最尾ポインタ、ビットマップ情報を 6.2.(2)の図のレディー管理構造体で個別に持つ。

(5) 優先度とタイムアウトレベルのレディーキュー

主に、0(1)スケジューリングで使用されるレディー構造である。このレディー構造は、Linux カーネル 2.6~2.6.22 で実装されていた方式を本研究 OS に実装した(参考文献 [29], [30])。連続してそのタスクをスケジューリング対象管理としないために、優先度レベルのキューを 2つ持つ。一方を activ キュー(実行可能状態タスクをキューイング)、もう一方を expired キュー(タイムアウトしたタスクをキューイング)とする。タスク起動である `sta_tsk()` システムコールまたは、`run_tsk()` システムコール発行では activ キューへ TCB を接続し、タイムアウトした場合は expired キューへ接続する。そして、activ キューにタスクがなくなり次第、expired キューと activ キューを入れ替える。図 6-11 に示す。

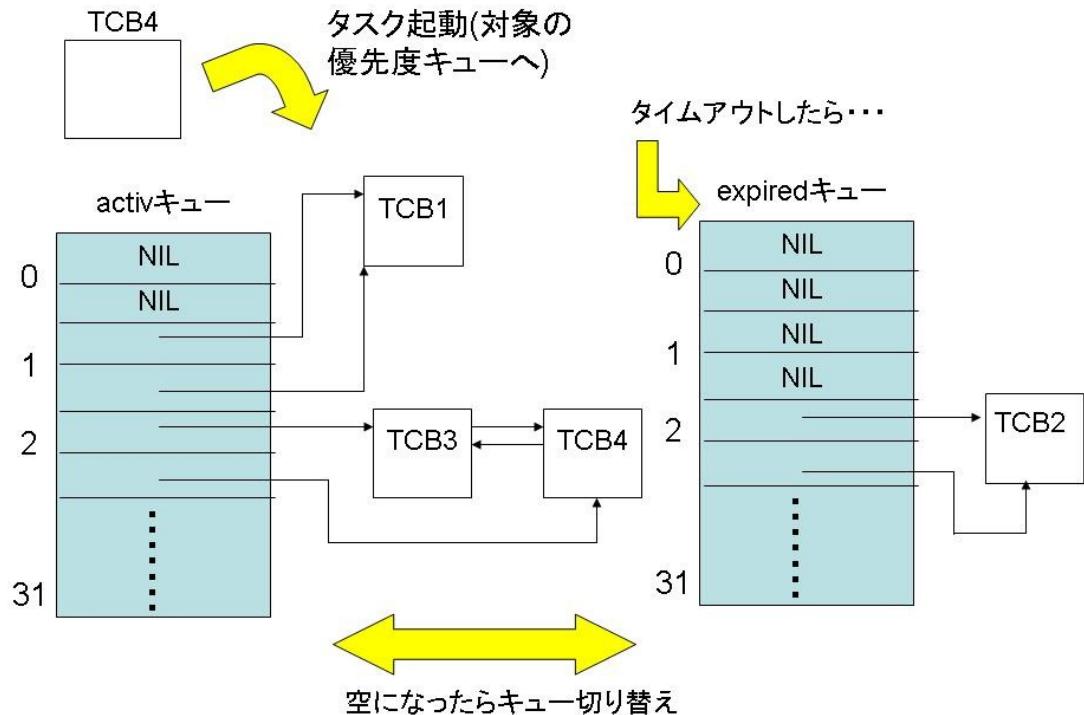


図6-11 タイムアウトと優先度レベルのレディーキュー

このレディー管理データ構造では、優先度数分の TCB を接続するための先頭ポインタと最尾ポインタ、ビットマップ情報のそれぞれ 2 組(activ キュー, expired キュー)を 6.2.(2)の図のレディー管理構造体で個別に持つ。

(6) 単一のツリーレディー

主に、FRS, EDF, LLF スケジューリングで使用するレディー構造である。ツリー構造のレディーとしたのはタスク総実行時間、デッドライン、余裕時刻に上限がない事と、TCB 插入時の計算量よりツリー構造が適している(单一のレディーキューでの ERS, EDF, LLF を実装できるが、挿入時の計算量がツリー構造より大きい)。ツリーの種類を下記に示す。

- ① 半順序木 → 広く知られているデータ構造で、木を一定のルールで走査しても、親以外は整列しないため優先度取得システムコールなどで正しい優先度が取得できないため除外。
- ② 2 分探索木 → 広く知られているデータ構造で、木を一定のルールで走査すると、整列するため、その走査番号を優先度とすれば動的優先度を管理できる。
- ③ AVL 木 → 広く知られているデータ構造だが、実装が難しく、回転の操作が膨大となる。
- ④ 赤黒木 → 広く知られているデータ構造で効率的だが、実装が難しい。
- ⑤ AA 木 → あまり知られていない。

また、初学者向け OS ということもあり、双方向リンクリストによる 2 分探索木にした。さらに、動的優先度管理の計算量を少なくするために、動的優先度管理双方向のリンクド

リストと組み合わせて実装した。図 6-12 に示す。

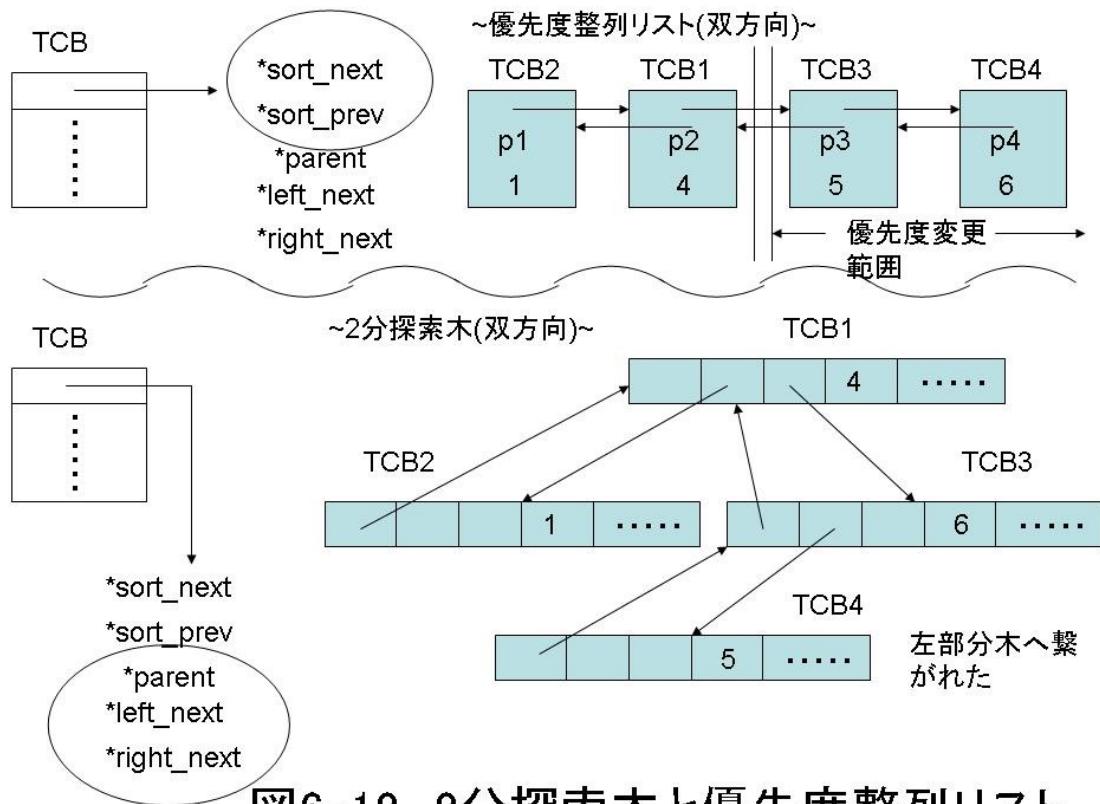


図6-12 2分探索木と優先度整列リスト

このレディー管理データ構造では、TCB を接続するための tree ルートポインタと優先度整列リストの先頭ポインタを 6.2.(2)の図のレディー管理構造体で個別に持つ。

本研究OSの実装として、動的優先度に上限(ツリー構造内で管理するタスク数)を設けている。上限を設けていると、タスクが dormant と生成を繰り返すと優先度が繰り上がっていき枯渇する。図 6-13 に示す。

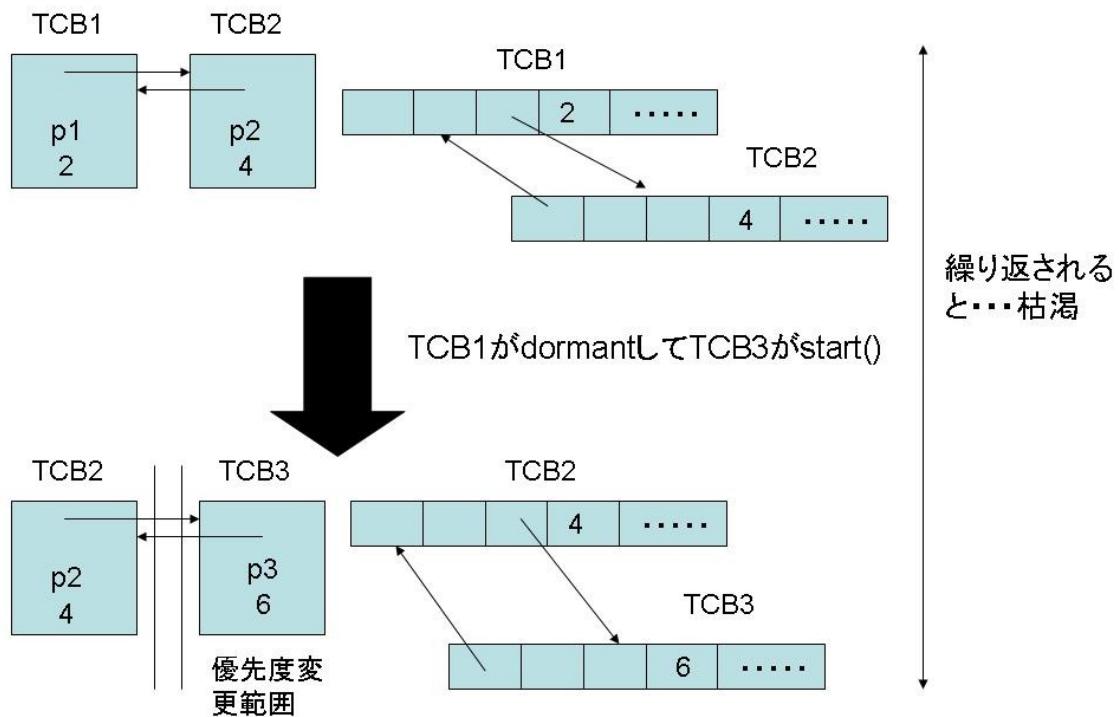


図6-13 優先度枯渇

よって、2分探索木に存在する最低優先度がツリー構造内で管理するタスク数の半分の時点で、すべてのタスクの優先度を最調整する。そのため、6.2.(2)の図のレディー管理構造体にレディーに存在するタスク数と最小優先度を個別に持つ。

また、ツリー構造では、管理元対象がある。管理元対象は実行時間(Fair スケジューリング適用時)、デッドライン時刻(Earliest Deadline First 適用時)、余裕時刻(Least Laxity First 適用時)となる。これらは属性として扱い、6.2.(2)の図のレディー管理構造体で個別に持つ。

6.3. スケジューラ考察

リアルタイム性を意識した組込みOS等で使用されるスケジューリングの分類と、公平性を意識した汎用系OS等で使用されるスケジューリングの分類ごとにスケジューラを考察する。

(1) リアルタイム性を意識した組込みOS等で使用されるスケジューリングの考察

この分野では、いかにデッドラインを保障しつつ、コンテキストスイッチングを減らせるかが目標となる。

RMとDMスケジュールの根本的な可能性判断は同等となり、同じくEDFとLLFもスケジュール可能性判断は同等となる。よって、ここではRMとEDFで考察する。

RMとEDFとの差異を表す周期タスクセットのスケジュール例を図に示す。(EDFは周期タ

スクに対しても適用できる。)

RMはデッドライン違反をし、スケジュールできない。対して、EDFはスケジュールできている。さらに、RMではコンテキストスイッチは5回に対して、EDFはコンテキストスイッチが1回となっている。つまり、OSのオーバーヘッドを軽減できている。また、RMは参考文献[11]でタスクの周期に特別な考慮をしない場合、CPUの能力を88%までしか使用できないとされている。よって、EDFはRMよりも最適にCPUを使用できる事から、リアルタイム性を意識したスケジューリング分野では最適スケジューリング方式となる。

(2) 公平性を意識した汎用系OS等で使用されるスケジューリングの考察

この分野では、いかにタスクの総実行時間を均等にし、CPUバウンドタスクとI/Oバウンドタスクを公平にスケジュールするかが目標となる。

① FCFS

この方式は、タスクごとのCPU割付け回数は均等にできる。しかし、スライスタイムを持っていないため、タスクの総実行時間までは均等に扱えない。よって、(2)冒頭の目標が達成できない。

② RR

この方式は、タスクのスライスタイムをすべて同一とし、タスクがすべてタイムアウトするという条件化では、タスクの総実行時間の均等配分ができる。しかし、I/OバウンドタスクとCPUバウンドタスクを混在させてレディーヘキューイングしているため、I/Oバウンドタ CPUバウンドタスクとI/Oバウンドタスクのタスク総実行時間の公平性は確保できない。CPUバウンドタスクのみの総実行時間の公平配分は可能である。

③ PS

この方式は、優先度の高いタスクがCPUを占有するので、同優先度タスクごとのCPU割付け回数は均等にできるが、全タスクの総実行時間までは均等に扱えず、(2)冒頭の目標が達成できない。

④ RPS

この方式は、同優先度タスクのスライスタイムをすべて同一とし、タスクがすべてタイムアウトするという条件化では、同優先度タスクの総実行時間の均等配分ができる。さらに、優先度によって、タスク総実行時間の均等配分が可能となる。しかし、I/OバウンドタスクとCPUバウンドタスクを混在させてレディーヘキューイングしているため、CPUバウンドタスクとI/Oバウンドタスクの公平性は確保できない。

⑤ MFQ

この方式は、適切なスライスタイムを設ける事によって、優先度レベルでタスク総実行時間の均等配分ができ、CPUバウンドタスクとI/Oバウンドタスクの公平性が確保できる。しかし、グループ制の考慮はできない。これは(2-1)で述べる。

⑥ O(1)

この方式は、適切なスライスタイムを設ける事によって、優先度レベルでタスク総実行時間の均等配分ができる。しかし、グループ制は確保できない。

考察結果を図6-9にまとめた。なお、図6-14はスケジューラが最大限に活用できた場合の評価である。

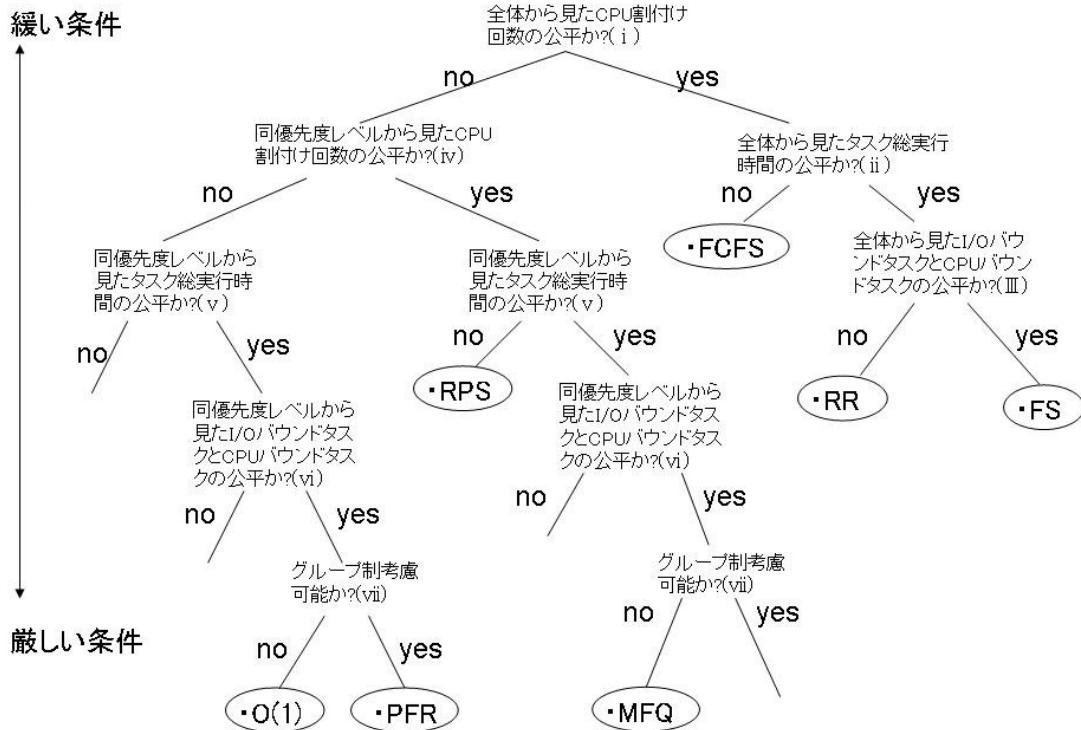


図6-14 公平性を意識したスケジューリングの考察まとめ

上図の見方として、木を root から葉(スケジューラ)に向かって深さ優先探索でトラバースし、枝の yes,no に基づいてトラバースしたノードの条件を各スケジューラはクリアできる。なお、上図にある(i)と(iv)の条件は特に(2)の冒頭にある目標とは直接関係しない。

図より、MFQ と O(1)スケジューリングが(2)の冒頭にある目標を達成可能である。

(2-1) MFQ と O(1)の問題点

MFQ と O(1)の問題点として、グループ制を考慮できない。グループ制とはユーザが設定できる小グループ(優先度)であり、グループを優先してスケジューリングが可能かである(これは本来の優先度スケジューリングの機能)。例を下図6-15に示す。

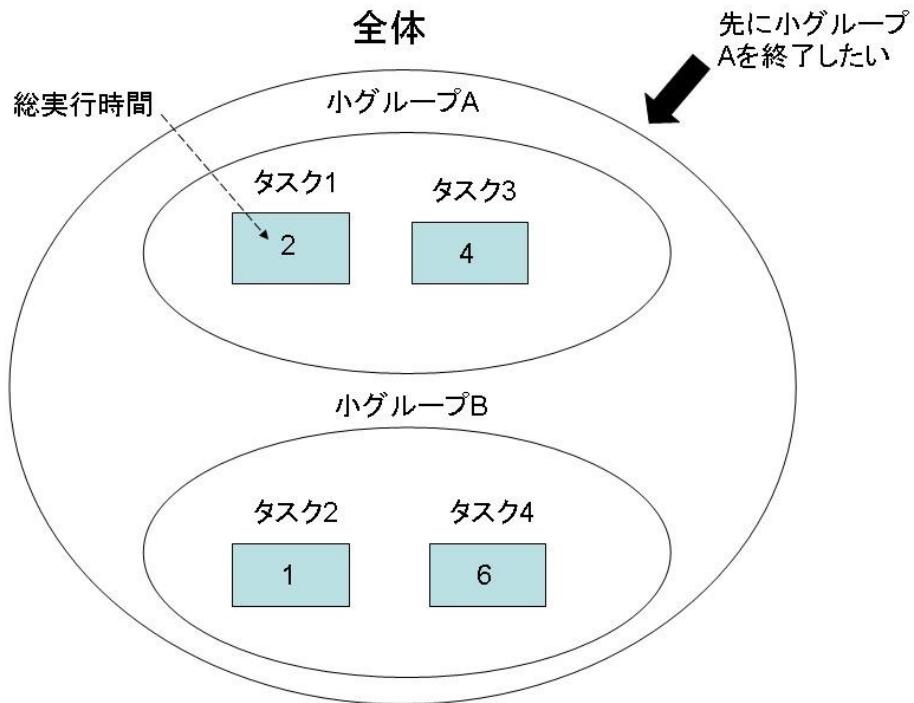


図6-15 小グループごとにタスクを見た場合

MFQがグループ制を確保できないのは、タスクがタイムアウトする度に、ユーザが割付けた優先度をスケジューラが変更するからである。つまり、タスクが異なるグループに属される事となる。

対してO(1)がグループ制を確保できないのは、タスクがタイムアウトすると、他のすべてのタスクがタイムアウトするまで、そのタスクは実行権を得られないからである。つまり、スケジューラが次のグループをスケジューリング対象とする事となる。

グループ制を確保できないもう一つの理由として、優先度レベルでのキューイングを行っているからである。

(3) 公平性の限界

6.3.(2)のFRSを適用すれば、全体としての公平性は確保できる。短いスライスタイムを定めるほど、公平性は高くなる。しかし、タスクが細切れになるので、タスクのプリエンプト及びディスパッチが多発する現象が起こる。つまり、短いスライスタイムを設定すればするほど、公平性は高くなるが、OSのオーバーヘッドが大きくなる相反する条件となる。よって、完全公平性スケジューリングというものはできない。

6.4. スケジューリングの新規方式(Priority Fair Scheduling(以下PFS))

6.3.(2-1)の問題点を解消するためのスケジューラとしてPriority Fair Schedulingを提案する。このスケジューリングは、タスクの総実行時間を均等にし、CPUバウンドタス

クと I/O バウンドタスクを公平にスケジュールでき、かつグループ制の考慮ができる方式である。グループの作成は、ユーザが acre_tsk() システムコールで固定優先度を設定し、レディー構造は(1)とした。タスクスライスタイムの扱いは、6.1.(4-3)と同様とした。固定優先度割付け後は、タスク総実行時間が少ないタスクほど、スケジューラが動的優先度を割付ける。ここまでを図 6-16 に示す。

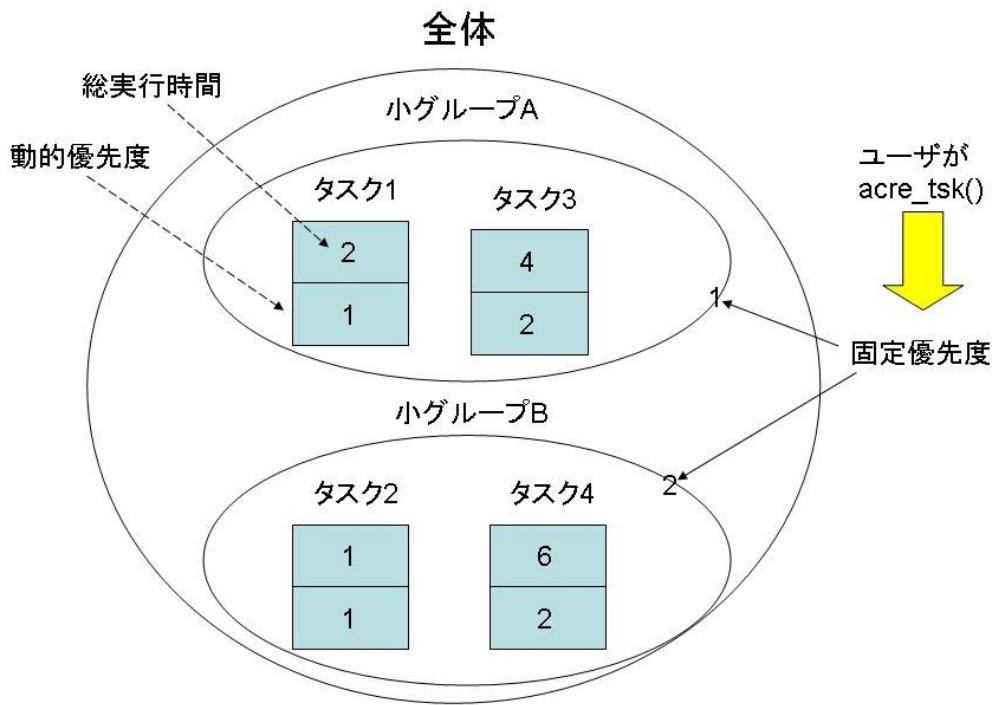


図 6-16 Priority Fair スケジューリング

PFR スケジューラ起動では、固定優先度の高い動的優先度管理リスト((1)の図 6-12)を参照し、その先頭(先頭が最も動的優先度が高い)の TCB を実行対象タスクとする。この実装によって、優先度に依存するシステムコール及び OS 機構を有効化できる。

なお、動的優先度管理リストのタスクがすべて待ちとなった場合は、次の固定優先度の高い動的優先度管理リストを参照する。

(1) 優先度レベルのツリーレディー

このレディー管理データ構造は、PFR スケジューリングで適用できるようにした。具体的には、固定優先度には上限があるため固定優先度配列を設け、その固定優先度配列の分だけ、6.2.(5)のツリーレディー構造を持つ。ツリーレディー構造を持たせたのは、タスクの総実行時間を均等にし、CPU バウンドタスクと I/O バウンドタスクを公平にスケジュールするためである。また、固定優先度配列の検索は、6.2.(3)と同様のビットサーチアルゴリズムで行う。優先度レベルでのツリーレディー構造を図 6-17 に示す。

また、PFRでは4.2.(1)節の図4-2のTCBのスケジューラ依存情報は仮想実行時間となる。

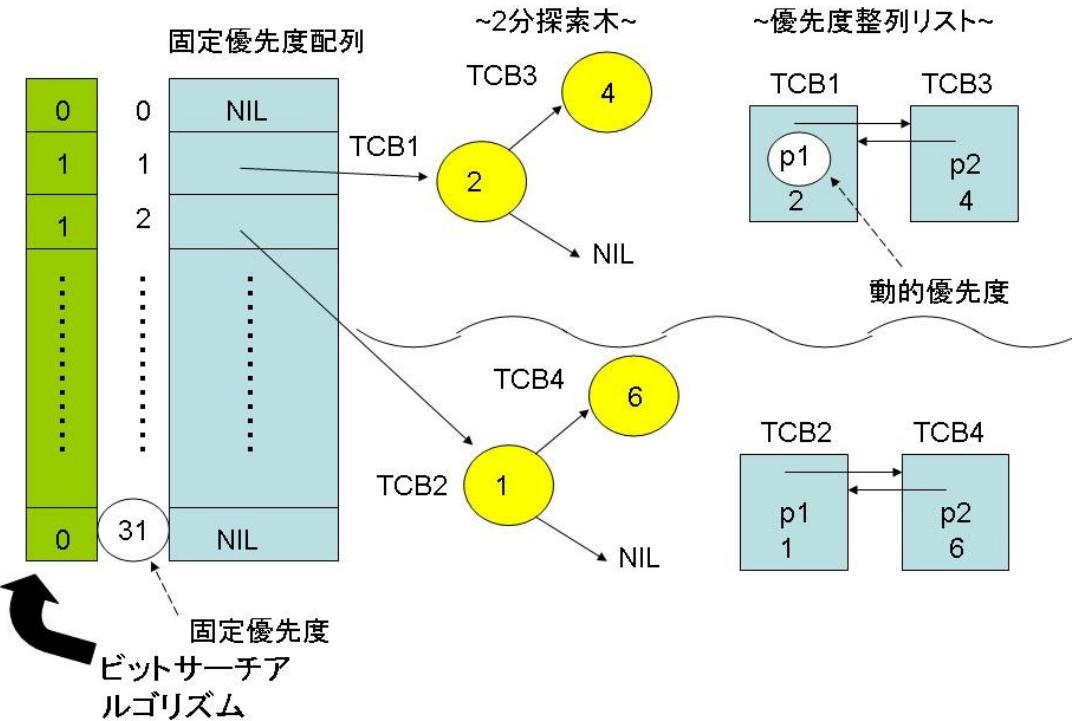


図6-17 Priority Fairスケジューリング内部構造

このレディー管理データ構造では、6.2.(6)の単一のツリーレディーで管理させる情報の優先度数分とビットマップテーブルを6.2.(2)の図のレディー管理構造体で個別に持つ。

6.5. スケジューラ動的切替え機能

本研究OSでは、13個のスケジューラはメモリヘリロードせずに動的に切替えることができる。切替えは①～③の方式で実装した。

- ① タスクがsel_schdul()システムコールを発行して異なるスケジューリングへ切替える方式
- ② タスクがsel_schdul()システムコールを発行した後にrol_sys()システムコールを発行または、NMI発行してinitタスク生成前段階へシステムをロールバックして切替える方式
- ③ initタスクでsel_schdul()システムコールを発行する方式

①を実装したのは単純に実行途中から切替えた場合のシステムレスポンスを見るためであり、②と③を実装したのは同一タスクセットに対して異なるスケジューリングを適用したレスポンスを見るためである。

sel_schdul()の仕様については付録のシステムコールの仕様を参考にしていただきたい。また、後続にrol_sys()システムコールのタスク終了ロールバックが指定される事があ

るため、スケジューラの情報を登録する専用のメモリセグメント（スケジューラ情報メモリセグメント）を設け、sel_schdul()システムコールでは、このメモリセグメントへ登録する。スケジューラ適用時にスケジューラ情報メモリセグメントから読み出す。これは、sel_schdul()の属性問わず行う。メモリマップを図 6-18 に示す。なお、スケジューラのコードは DRAM へ配置している。

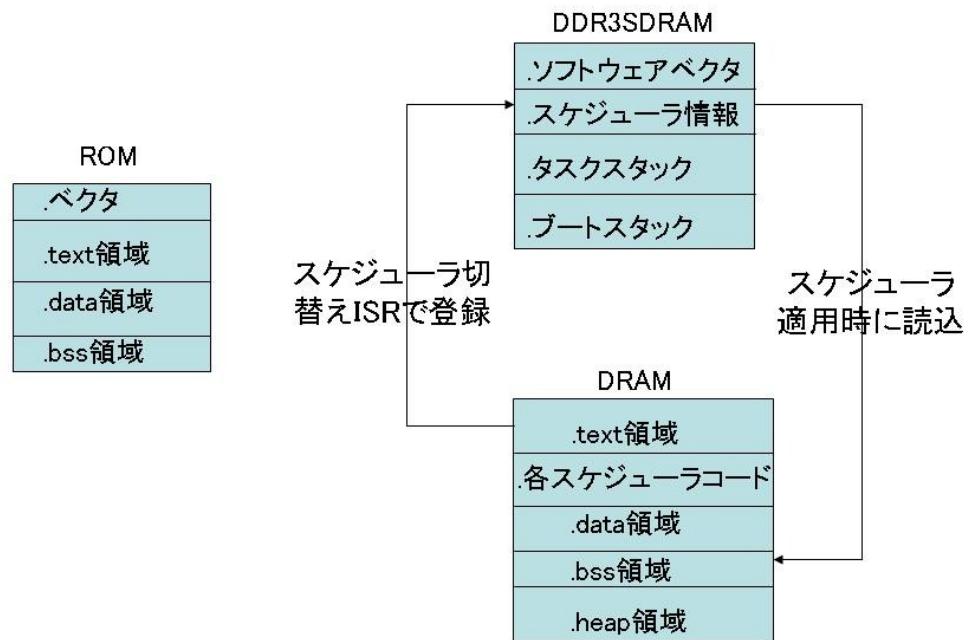


図6-18 OSメモリマップ

6.5.①のシーケンスを図 6-19 に示す。6.5.②のシーケンス図 6-20 に示す。

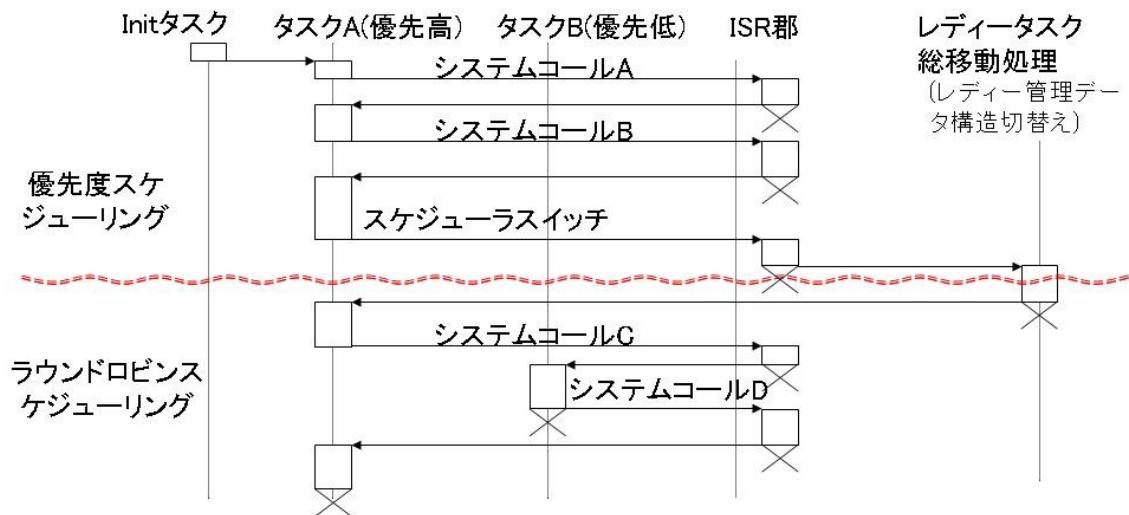


図6-19 ①のシーケンス

図6-14はデフォルトで優先度スケジューリングを行い、`sel_schedul()`システムコール発行後はラウンドロビンスケジューリングに切替えた場合である。スケジューラ切替えシステムコールでは、単純に切替える属性を指定する。ISRでは、スケジューラ情報メモリセグメントへ次に切替えるスケジューラ情報を登録し、スケジューラを切替えられるかチェックする。そして、その延長で全タスクのレディー間移動を行う。

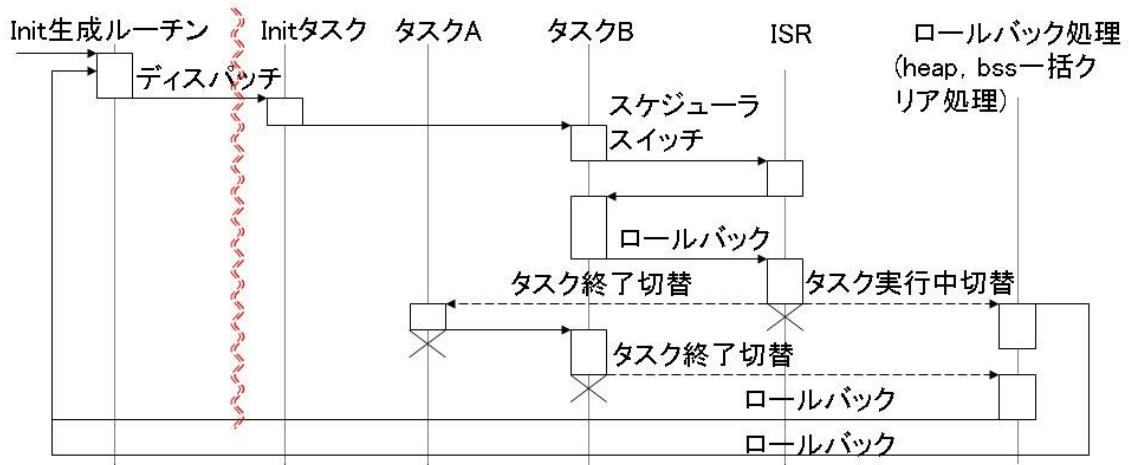


図6-20 ②のシーケンスとメモリマップ

図 6-18 は `sel_schedul()` システムコールを発行後, `rol_sys()` システムコールを発行し, `init` タスク生成段階へロールバックするシーケンスである。 `sel_schedul()` の ISR では, スケジューラ情報メモリセグメントへ次に切替えるスケジューラ情報を登録する。次に, `rol_sys()` システムコールが発行し, その後は依存フローとなる。依存フローとなるタスク終了切替え(タスク終了ロールバック)とタスク実行中切替え(実行中ロールバック), ロールバック処理は 5.5.(7) で述べた。

7. 時間管理

本節では、時間管理について述べる。ターゲットハードウェアに配線されている 8 ビットタイマ資源を 4 つ使用し、クロックを 8192 分割してカスケード接続し、16 ビットタイマとして 2 本実装した。この実装としたのは、長いタイマ要求時間を計れるようするためである。1 つをハードタイマドライバが使用し、残りの 1 つをソフトタイマドライバが使用する。

ハードタイマドライバ、ソフトタイマドライバともに精度は、1 msec となる。

7.1. ハードタイマドライバ

本研究 OS のハードタイマドライバは、タイマ精度が高いタイマドライバである。しかし、タスクからのタイマ資源以上の並行要求を受付ける事ができない。

ハードタイマドライバへの並行要求が発行されても不整合が起きないように、排他制御を行う方法とタイマ要求をキュー管理する方法がある。前者である排他制御を行う方法だと、タイマへの要求が遅延し正確なタイマのレスポンスを得る事ができない。よって、後者のタイマ要求をキュー管理する方式としてソフトタイマドライバを実装した。

7.2. ソフトタイマドライバ

本研究 OS のソフトタイマドライバは、タイマ精度がハードタイマドライバより低いタイマドライバである。しかし、タスクからのタイマ並行要求を受け可能である。タイマの精度が低くなる現象は後述する。

内部データ構造は、タイマ要求(システムコールで行う)をリンクリストによるキュー管理する。また、並行要求に整合性を持たせるために差分させる差分リンクリストキュー管理方式を実装した。1 つのタイマ要求を管理するタイマ構造体を(1)で説明し、差分リンクリストキュー管理は(2)で述べる。

(1) タイマ構造体

タイマ構造体を図 7-1 に示す。この構造体の制御は(2)で行う。

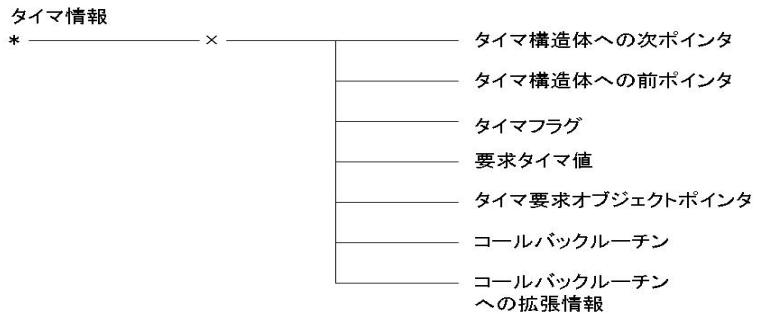


図7-1 タイマ構造体

タイマ要求の度にタイマ構造体を生成する。図7-1の情報をセットし、(2)で管理される。なお、ユーザから見たタイムカウントは、システムコールによるタイマ要求発行時からの相対時間とする。

(2) 差分リンクリストキュー管理方式

まず、タイマ資源へタイマ値((1)の図7-1のメンバである要求タイマ値)のスタートはキュー先頭のタイマ構造体とし、キューにタイマ構造体が挿入された時点とする。タイマ満了(タイマ割込み)で(1)の図7-1のメンバであるコールバックルーチンを呼び、先頭のタイマ構造体を抜取り、そのタイマ構造体を排除する。この流れを図7-2に示す。

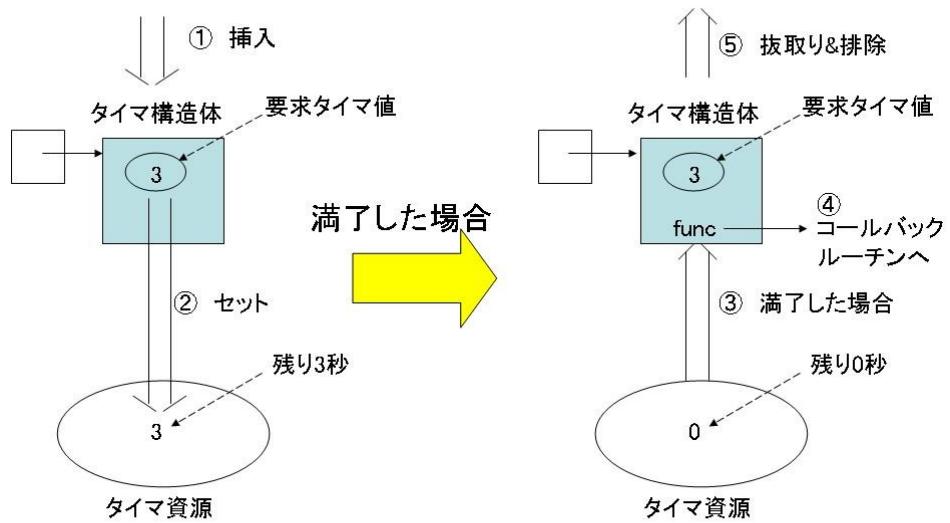


図7-2 タイマのセットと満了

なお、要求が周期タイマの場合((1)の図7-1のメンバであるタイマ要求オブジェクトポインタが設定されている)，タイマ満了時にタイマ構造体を排除せずに、再度キューへ挿入する。

図7-2の状態で次のタイマ要求がきた場合、現在のタイマカウント時間を取り得し、要求タイマ値((1)の図7-1のメンバである要求タイマ値)から差分する。そして、タイマ要求値が昇順に並ぶようにキューの先頭の次のタイマ構造体(先頭はタイマをすでにかけている)からリンクドリストを走査し、走査したタイマ構造体の分だけ要求タイマ値を差分する。この流れを図7-3に示す。なお、図7-3の状態でタイマが満了した時は、次のタイマ構造体のタイマ値をセットする。

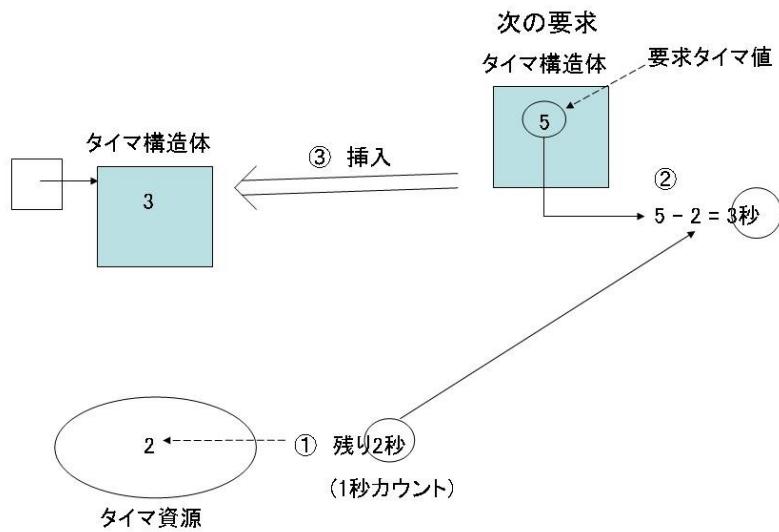


図7-3 差分処理

走査した後続にタイマ構造体が存在する場合、そのすべてのタイマ構造体の整合性を保つために差分処理を行う。

この管理方式の難しいところは、タイマ要求によるシステムコールとタイマ割込みによるタイマの満了処理(タイマ構造体の排除、また外部割込みなので IRQ ピンのアサートで割込み発生)がすれ違いに起きる事である。図 7-3 に示す。

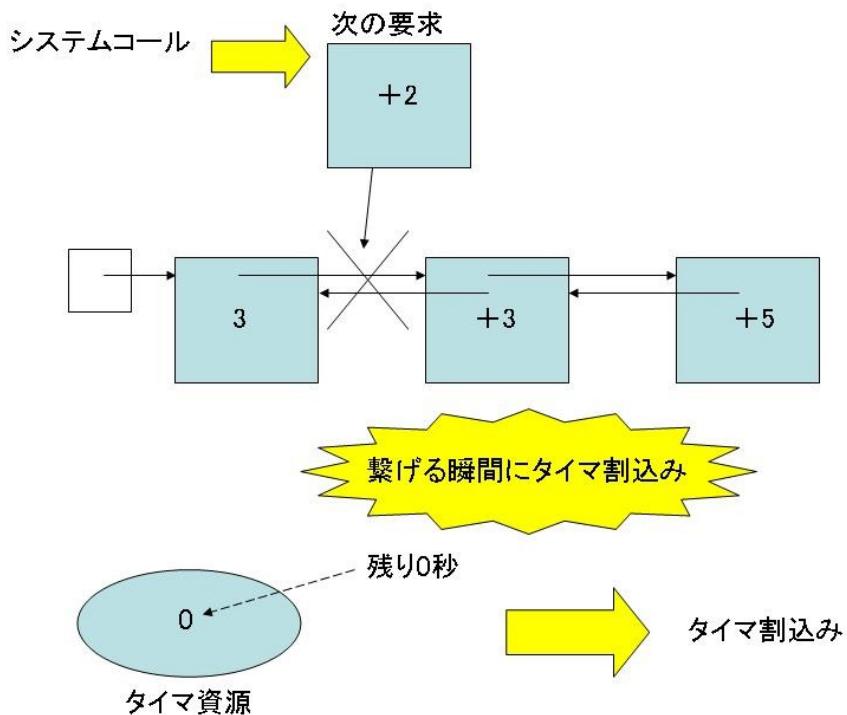


図7-4 割込みのすれ違い

(3) 差分リンクリストキュー管理方式の欠点

(2)で述べた、差分リンクリストキュー管理方式は、要求タイマ値をキューに管理する時にリストを走査する必要がある。また、ユーザから見たタイムカウントはシステムコール発行時となるので、リストのタイマ構造体の数が増加するほど、タイマ構造体の管理に時間がかかる。よって、タイマ満了の誤差が生じる。誤差を減らすためには、2分木による差分方式が有効である。しかし、本研究OSには未実装であり、今後の課題とする。

7.3. 周期ハンドラ管理データ構造

本節では、本研究OS内部で扱う周期ハンドラ管理データ構造について述べる。本研究OSの周期ハンドラは、マルチタスクが使用できる周期タイマ機能となり、周期的な処理を記述できる。周期ハンドラ ISR群では、7.2.節のソフトタイマドライバへの要求処理関数をラップ(周期ハンドラ ISR群の内部で、ソフトタイマドライバ関連の関数を呼んでいる)させている。

なお、周期タスク生成は周期ハンドラを利用して行う。具体的には、周期ハンドラ内に非タスクコンテキスト用のシステムコール `ista_tsk()`を記述する。

(1) 周期ハンドラ使用上の注意

注意点を以下の①と②に示す。

- ① 各タスクからの周期ハンドラ制御処理(`sta_cyc()`, `stp_cyc()`)が終わるまでは、作成

した周期ハンドラ自動割付 ID を書き換えたり、解放されたりしないよう(周期ハンドラを生成したタスクが自動割付 ID をローカルに確保し、そのタスクが retnrun で終了すると、自動割付 ID は解放される)にする。

② 周期ハンドラはタイマ割込みの延長上で起動するので、非タスクコンテキスト部である拡張 SVC ハンドラとなり、割込み無効モードとして実行される。つまり、最高優先度として実行されるので、周期ハンドラ内の処理を重くすると、タスクの応答時間が短くなる。

(1) 周期ハンドラ構造体

周期ハンドラを制御させる構造体の情報を下図 7-5 に示す。この構造体は、周期ハンドラ 1 つの制御ごとに持つべき情報を格納した構造体である。

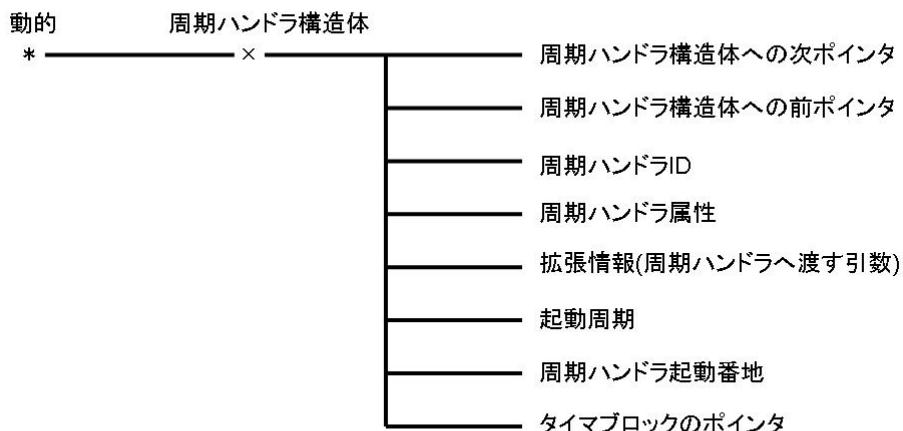


図7-5 周期ハンドラ構造体

図の free ノードへの次ポインタと free ノードへの前ポインタはカーネル init 時(10.2. 節参照)に設定する。周期ハンドラ ID, 属性(静的型または動的型を記録), 拡張情報, 起動周期, 周期ハンドラ起動番地は acre_cyc() の ISR で設定する。タイマブロックポインタは sta_cyc() の ISR で設定する。

(2) 周期ハンドラ情報構造体

タスク管理データ構造と同様に、周期ハンドラ管理データ構造に静的型と動的型を実装した。静的型は(1)を配列で管理し、動的型は(1)をリンクドリストで管理する。さらに、周期ハンドラ ID 変換テーブルを採用し、4.2.(4.2)節の方式とした。これらを管理するための構造体を周期ハンドラ情報構造体とし、図 7-6 に示す。

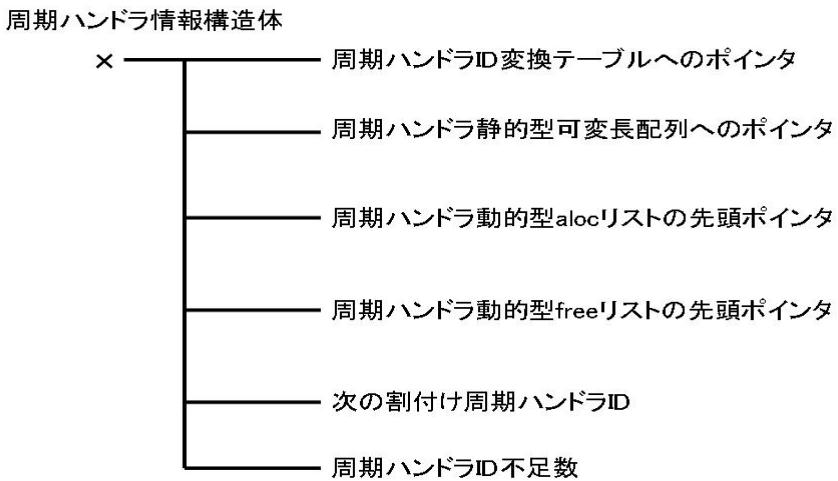


図7-6 周期ハンドラ情報構造体

各メンバは4.2.(4.2)節と同様である。

7.4. アラームハンドラ管理データ構造

本節では、本研究OS 内部で扱うアラームハンドラ管理データ構造について述べる。本研究OSのアラームハンドラは、マルチタスクが使用できるアラーム機能となり、定義時間に動作する処理を記述できる。アラームハンドラ ISR群では、7.2.節のソフトタイマドライバへの要求処理関数をラップ(アラームハンドラ ISR群の内部で、ソフトタイマドライバ関連の関数を呼んでいる)させている。

(1) アラームハンドラ使用上の注意

周期ハンドラと同様であるが、注意点を以下の①と②に示す。

① 各タスクからのアラームハンドラ制御処理(sta_alm(), stp_alm())が終わるまでは、作成したアラームハンドラ自動割付 ID を書き換えたり、解放されたりしないよう(アラームハンドラを生成したタスクが自動割付 ID をローカルに確保し、そのタスクが retnr で終了すると、自動割付 ID は解放される)にする。

② アラームハンドラはタイマ割込みの延長上で起動するので、非タスクコンテキスト部である拡張 SVC ハンドラとなり、割込み無効モードとして実行される。つまり、最高優先度として実行されるので、アラームハンドラ内の処理を重くすると、タスクの応答時間が短くなる。

(2) アラームハンドラ構造体

アラームハンドラ構造体の情報を下図 7-7 に示す。この構造体は、アラームハンドラ 1 つの制御ごとに持つべき情報を格納した構造体である。

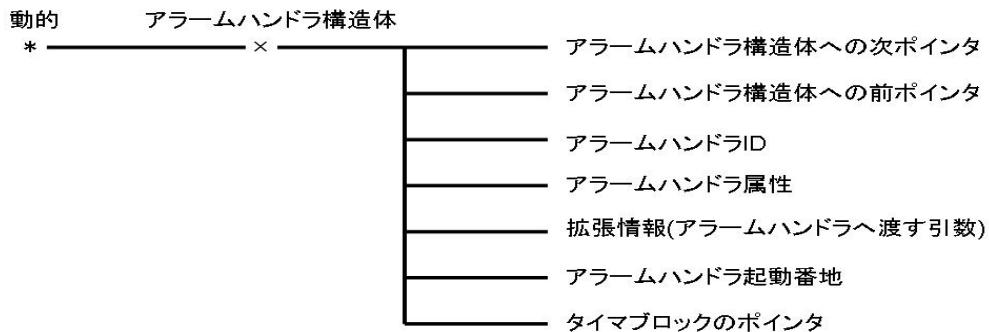


図7-7 アラームハンドラ構造体

図の free ノードへの次ポインタと free ノードへの前ポインタはカーネル init 時(10.2. 節参照)に設定する。アラームハンドラ ID, 属性(静的型または動的型を記録), 拡張情報, アラームハンドラ起動番地は acre_alm() の ISR で設定する。タイマブロックポインタは sta_alm() の ISR で設定する。なお, アラームハンドラ動作開始時間は sta_alm() で設定する。

(3) アラームハンドラ情報構造体

タスク管理データ構造と同様に, アラームハンドラ管理データ構造に静的型と動的型を実装した。静的型は(1)を配列で管理し, 動的型は(1)をリンクドリストで管理する。さらに, アラームハンドラ ID 変換テーブルを採用し, 4.2.(4.2)節の方式とした。これらを管理するための構造体をアラームハンドラ情報構造体とし, 図 7-8 に示す。

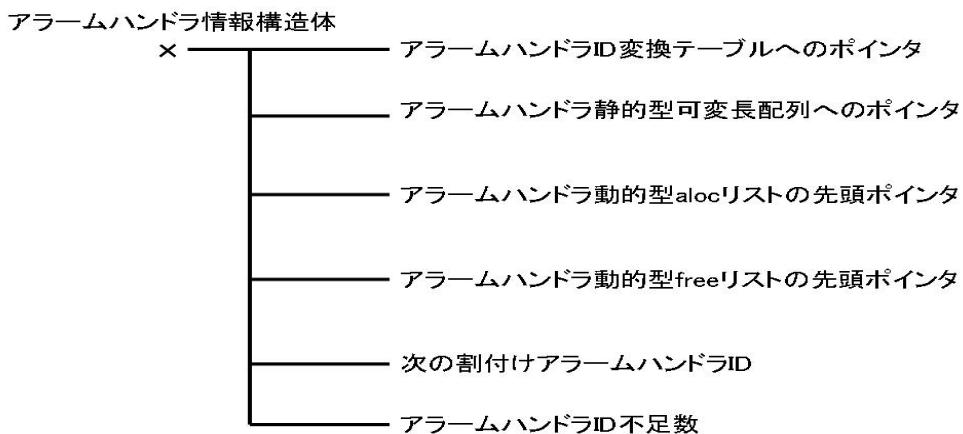


図7-8 アラームハンドラ情報構造体

各メンバは4.2.(4.2)節と同様である。

8. 同期・排他・通信機能

8.1. セマフォ管理データ構造

本節では、本研究OS内部で扱うセマフォ管理データ構造について述べる。本研究OSのセマフォは同期・排他に使用する。なお、ゼネラルセマフォとしても使用する事もできる。主に、排他はグローバル変数へのアクセス時となる。

(1) セマフォ使用上の注意

注意点を以下の①と②に示す。

- ① 各タスクからのセマフォ制御処理(wai_sem(), sig_sem()など)が終わるまでは、作成したセマフォ自動割付IDを書き換えたり、解放されたりしないよう(セマフォを生成したタスクが自動割付IDをローカルに確保し、そのタスクがretrunで終了すると、自動割付IDは解放される)にする。
- ② セマフォの取得及び解放は複数のタスクからできる。セマフォを取得したタスクを終了させる場合、注意が必要である。なお、本研究OSでは、4.2.(3)節のタスク取得情報管理によるカーネルオブジェクト自動解放機能を実装している。

(2) セマフォ構造体

セマフォ構造体の情報を下図8-1に示す。この構造体は、セマフォ1つの制御ごとに持つべき情報を格納した構造体である。

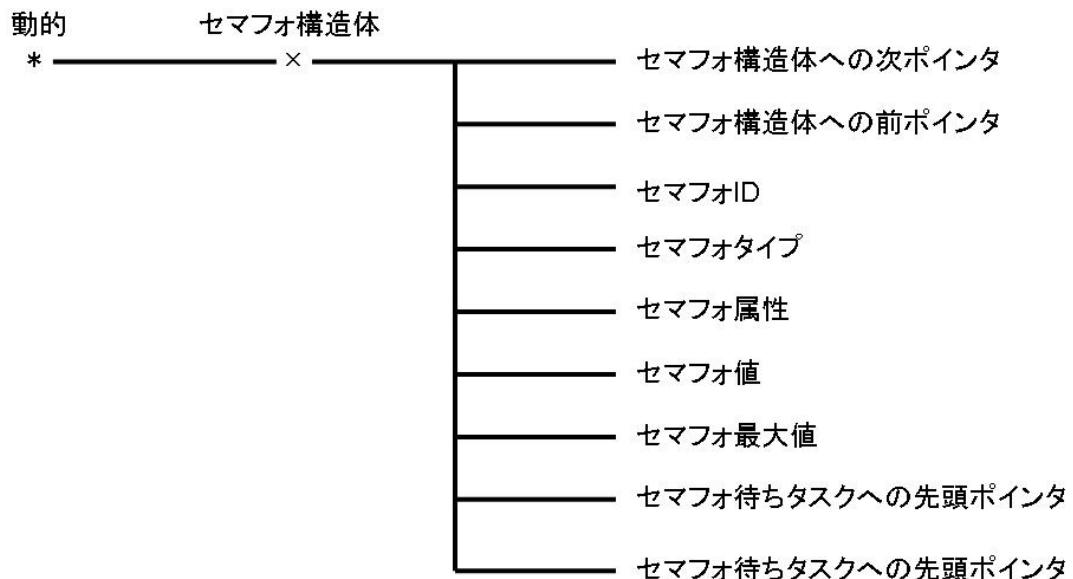


図8-1 セマフォ構造体

図の free ノードへの次ポインタと free ノードへの前ポインタはカーネル init 時(10.2. 節参照)に設定する。セマフォ ID, セマフォタイプ(静的型または動的型を記録), 属性(タスクをレディーへ戻すアルゴリズム), セマフォ最大値(ゼネラルセマフォを実現するため), セマフォ値は acre_sem() の ISR で初期設定する。セマフォ待ちタスク次ポインタと前ポインタは wai_sem() と sig_sem() の ISR で操作する。

(3) セマフォ情報構造体

タスク管理データ構造と同様に, セマフォ管理データ構造に静的型と動的型を実装した。静的型は(1)を配列で管理し, 動的型は(1)をリンクリストで管理する。さらに, セマフォ ID 変換テーブルを採用し, 4.2.(4.2)節の方式とした。これらを管理するための構造体をセマフォ情報構造体とし, 図 8-2 に示す。

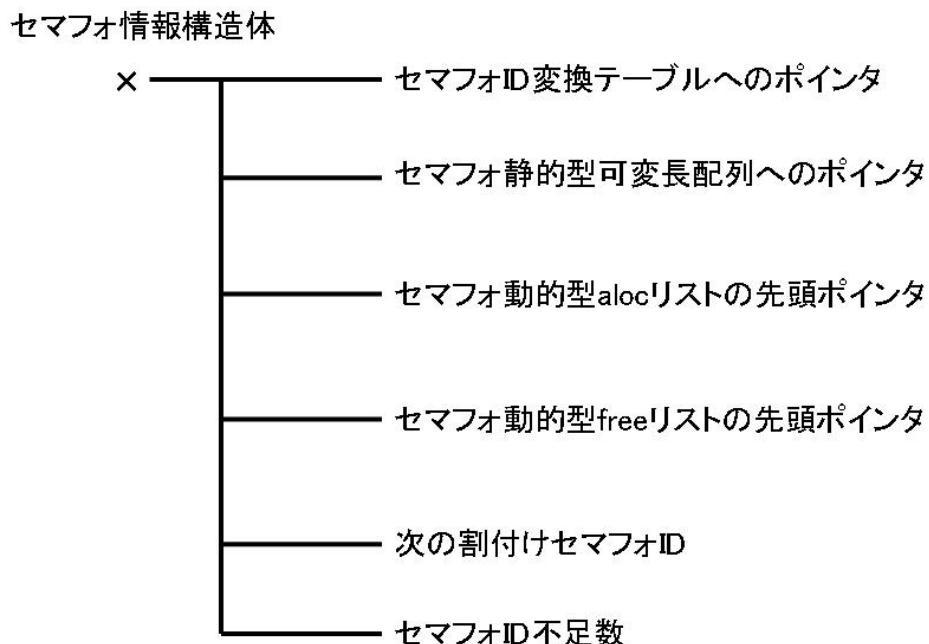


図8-2 セマフォ情報構造体

各メンバは 4.2.(4.2)節と同様である。

(4) タスクのセマフォ待ち

タスクにセマフォ待ちが発生した時は, セマフォ待ちをした ID のセマフォ構造体ヘキューリングし, タスクをレディー管理データ構造へ繋げず(シリアル割込み以外の割込み発生でタスクをレディー管理データ構造から抜き取る), タスク構造体へ 4.2.(2)節の処理(タスク構造体へ待ち情報の設定)を行う。

8.2. メールボックス管理データ構造

本節では、本研究OS内部で扱うメールボックス管理データ構造について述べる。本研究OSのメールボックスは、同期・通信に使用する。なお、メッセージに優先度を設けて緊急メッセージのハンドシェイク的なやりとりが可能である。

(1) メッセージ構造体

1つのメッセージを管理するためのメッセージ構造体を下図8-3に示す。

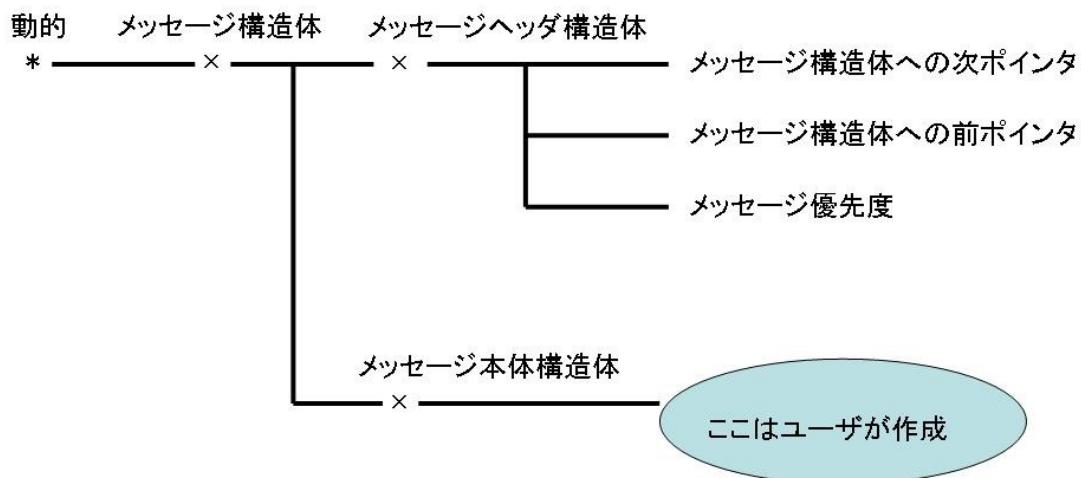


図8-3 メッセージ構造体

メッセージ構造体は、メッセージヘッダ構造体(カーネルで定義されている)を先頭のメンバーとし、ユーザ側で用意する。メッセージ本体構造体部はユーザが自由に設定できる(静的メモリや動的メモリ使用可能)。そして、`snd_mbx()`及び`rcv_mbx()`システムコール時に、メッセージ構造体をメッセージヘッダ構造体としてキャストしてカーネルに渡す実装とした。下図8-4に示す。

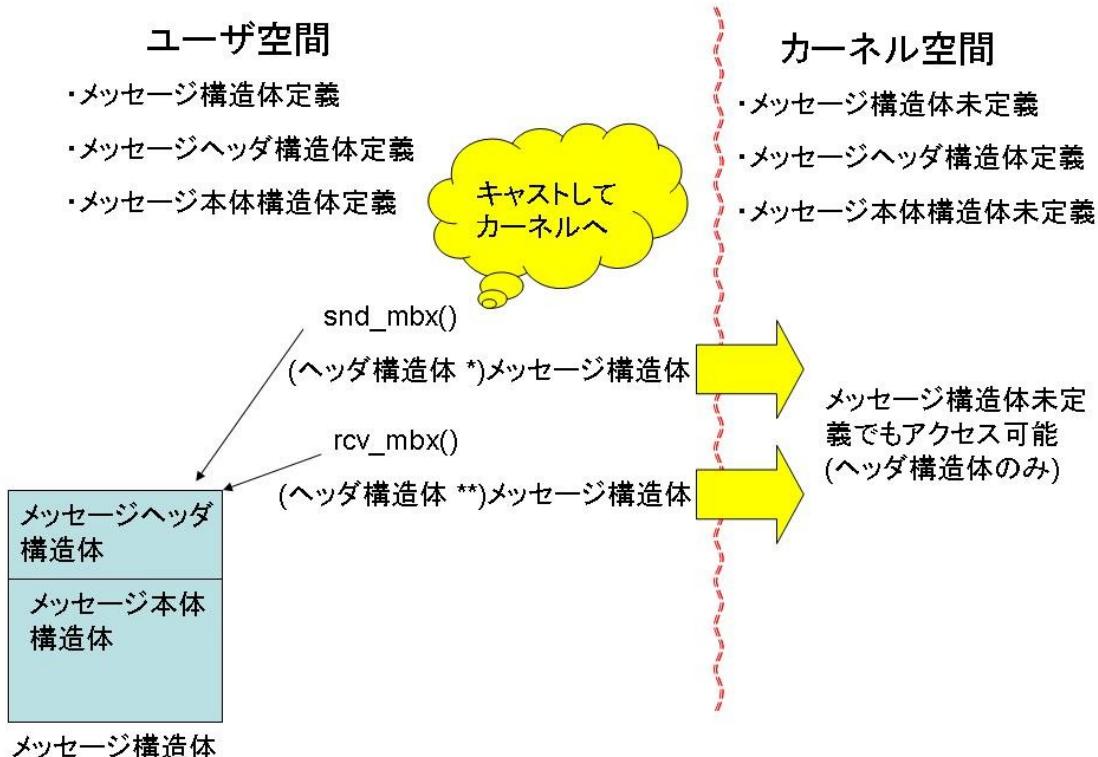


図8-4 snd_mbx()とrecv_mbx()時のメッセージ構造体制御

この実装としたのは、`snd_mbx()`及び`recv_mbx()`のISRではメッセージヘッダ構造体を操作する必要があるからである。また、メッセージ本体とメッセージヘッダをまとめて構造体として持ておく事によって、ISR内での処理が簡潔化する(メッセージ本体とメッセージヘッダ部分を繋げておく必要がない)。ちなみに、定番な実装法である。

図のように、ISR内の処理では、本来の型ではない型を使用して構造体を制御する(type-punning(型もじり)という)。型もじりはコンパイラの最適化によって、バグとなってしまう場合がある。

型もじりの問題について簡単に述べる。高度なコンパイラは、スーパースカラのような実行ユニットが複数あるプロセッサを認識し、それに向けた最適化を行う。連続する2つの命令が別の実行ユニットで並列・同時に実行される様、命令の順番を入れ替える(再配置)テクニックはその筆頭と言える。この再配置は当然、演算の依存関係に注意して処理の意味(内容)が変わらないように行わなければならない。しかし、変数に型の違う別名(エイリアス)を付けられてしまうと、コンパイラもその依存関係を追いかげなくなる(特にポインタ演算を含む場合)。本来は依存関係により再配置してはならない命令を再配置してしまうと、一見問題のないコードなのに不正な挙動を行う、発見しづらいバグとなるインライン関数の展開を含んだりすると、一層わかりづらい。

再配置機能自体は、ハードウェア(CPU)側のアウトオブオーダー実行機能として有名である。CPU側では比較的近い範囲でのアウトオブオーダー実行しか実現できないため、よ

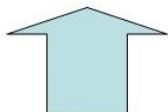
り広域ではコンパイラがコンパイルの段階に意識するほうが、効率がよい(高速な)コードとなる。むしろ、IA-64のようなVLIW・EPICアーキテクチャでは複雑化するCPU側でのアウトオブオーダー実行を諦め、コンパイラに全てを任せた方向ですらある。

なお、intに対するunsigned int等の共通性がある型や、共用体でアクセスする分には問題ない。char*でのアクセスは本来は違反にしたいのだろうが、バイト単位でアクセスしたいケースはあまりに多く、既存のコードで問題が出すぎるためか、OKになっている。ただし、元々共用体型でない変数を共用体のポインタを受けるようにしたり、共用体メンバを別のポインタの受け直してしまうと違反になる(参考文献[1])。

本研究では、gccクロスコンパラを使用してバグが発生した。バグ解消法として、以下の2つの方法がある。図8-5に示す。

i コンパイル段階で行なう

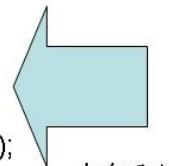
```
% gcc -O2 -fno-strict-aliasing ファイル名…
```



実行ファイル全体で型もじり有効化

ii ソースファイル内で行なう

```
タスクルーチン(int argc, char *argv[]) {  
    :  
    USER_MSG *msg __attribute__((my_alias));  
    :  
}
```



この変数のみ
型もじり有効化

ちなみに…

GCC3.X以降のみ可能

図8-5 型もじりのバグ解消法

図のiの方法だと、コード全体で型もじりを許すので、コーディングミス(誤って型もじりを行ってしまった場合)によるコンパイラの警告が発生せず、危険性が高くなる。図のiiの方法だと、特定の変数のみ型もじりのコンパイラ警告を発生をなくすので、危険性は低い。上記より、本研究OSでは図のiiの方法でバグを回避した。

(2) メールボックス使用時の注意

注意点を以下の①～④に示す。

- ① 各タスクからのメールボックス制御処理(snd_mbx(), recv_mbx()など)が終わるまでは、

作成したメールボックス自動割付 IDを書き換えたり、解放されたりしないよう(メールボックスを生成したタスクが自動割付 IDをローカルに確保し、そのタスクが retnrun で終了すると、自動割付 IDは解放される)にする。

- ② メールボックスでは送信されたメッセージがメールボックス構造体のリストで管理されるため、送信可能メッセージ上限がなく、送信側は待ち状態にはならない。
- ③ メールボックスではメッセージのポインタを snd_mbx()または recv_mbx()システムコールで制御させる。そのため、受信側がメッセージを処理し終わるまでは、送信したメッセージ構造体が壊されたり、その領域が解放されたり(送信側タスクがメッセージ領域をローカルで確保し、return でタスクが終了すると、メッセージ領域は解放される)しないように注意する。
- ④ 送信側のタスクは snd_mbx()システムコールを呼んだ後、メッセージ本体構造体を操作してはならない。これは、本研究 OS はマルチタスク OSなので、タイミング依存のバグが発生する可能性がある。

(3) メールボックス構造体

メールボックス構造体の情報を下図 8-6 に示す。この構造体は、メールボックス 1 つの制御ごとに持つべき情報を格納した構造体である。

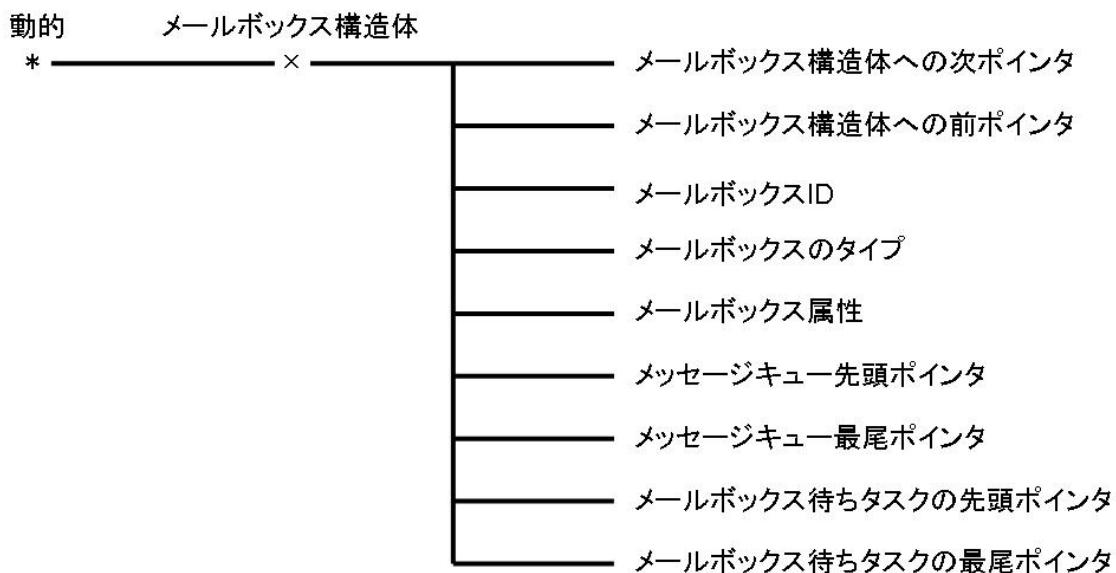


図8-6 メールボックス構造体

図の free ノードへの次ポインタと free ノードへの前ポインタはカーネル init 時(10.2.

節参照)に設定する。メールボックス ID, メールボックスタイプ(静的型または動的型を記録), メッセージ属性(メッセージをタスクに与えるアルゴリズム), 待ちタスク属性(タスクをレディーへ戻すアルゴリズム)は acre_mbx() の ISR で初期設定する。メッセージ次ポインタと前ポインタ, メールボックス待ちタスク次ポインタと前ポインタは snd_mbx() と rcv_mbx() の ISR で操作する。

(4) メールボックス情報構造体

タスク管理データ構造と同様に, メールボックス管理データ構造に静的型と動的型を実装した。静的型は(1)を配列で管理し, 動的型は(1)をリンクドリストで管理する。さらに, メールボックス ID 変換テーブルを採用し, 4.2.(4.2)節の方式とした。これらを管理するための構造体をメールボックス情報構造体とし, 図 8-7 に示す。

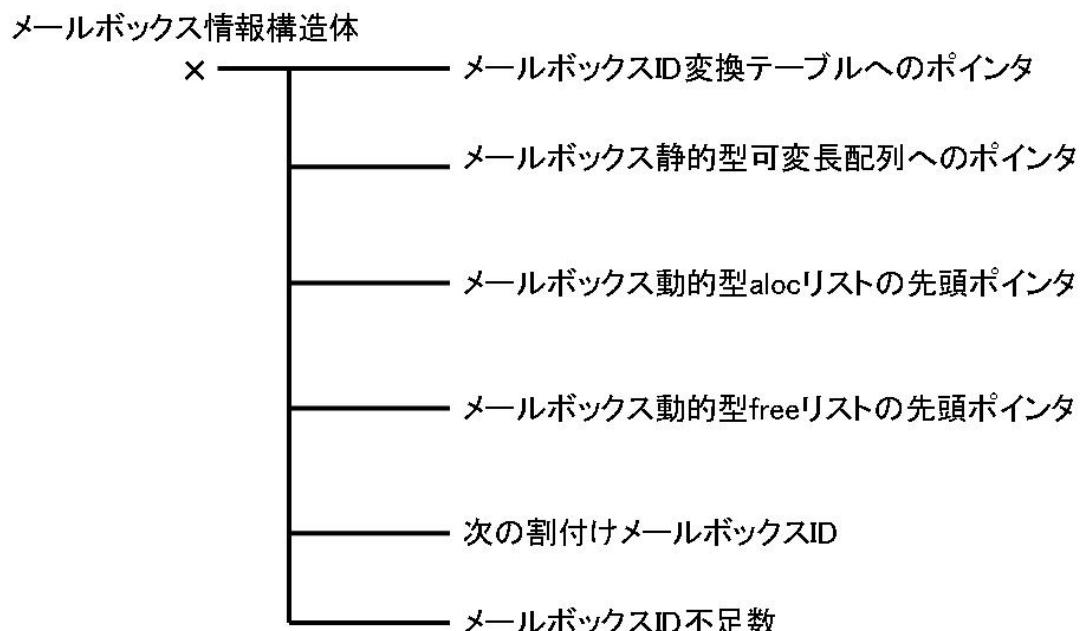


図 8-7 メールボックス情報構造体

各メンバは 4.2.(4.2)節と同様である。

(5) メールボックスによるタスク構造体依存処理

メールボックスによるタスク構造体依存処理を①と②に示す。

① タスクのメッセージ待ち

タスクにメッセージ待ちが発生した時は, メッセージ待ちをした ID のメールボックス構造体へキューイングし, タスク構造体をレディー管理データ構造へ繋げず(シリアル割込み以外の割込み発生でタスク構造体をレディー管理データ構造から抜き取る), タスク構造体へ 4.2.(2)節の処理(タスク構造体へ待ち情報の設定)を行う。なお, 待ちタスクが存在し

ない時に、メッセージがポストされると、そのメッセージをポストされたメールボックス構造体へメッセージをキューイングする。

② タスクのメッセージ取得情報

受信側タスクは、送信側タスクのメッセージ領域の参照のみとなるので、受信側タスクが終了しても自動解放処理は必要ない。よって、メールボックスでは4.2.(1)節の図4-2のメンバである取得情報管理は行わない。

8.3. mutex 管理データ構造

本節では、本研究OS内部で扱うmutex管理データ構造について述べる。本研究OSのmutexは完全的な排他に使用する。完全的な排他(セマフォとの相違点)とは、loc_mtx()でロックしたタスクからのみ、そのmutexをunl_mtx()できる(mutexのオーナシップ機能)。下図8-8に示す。

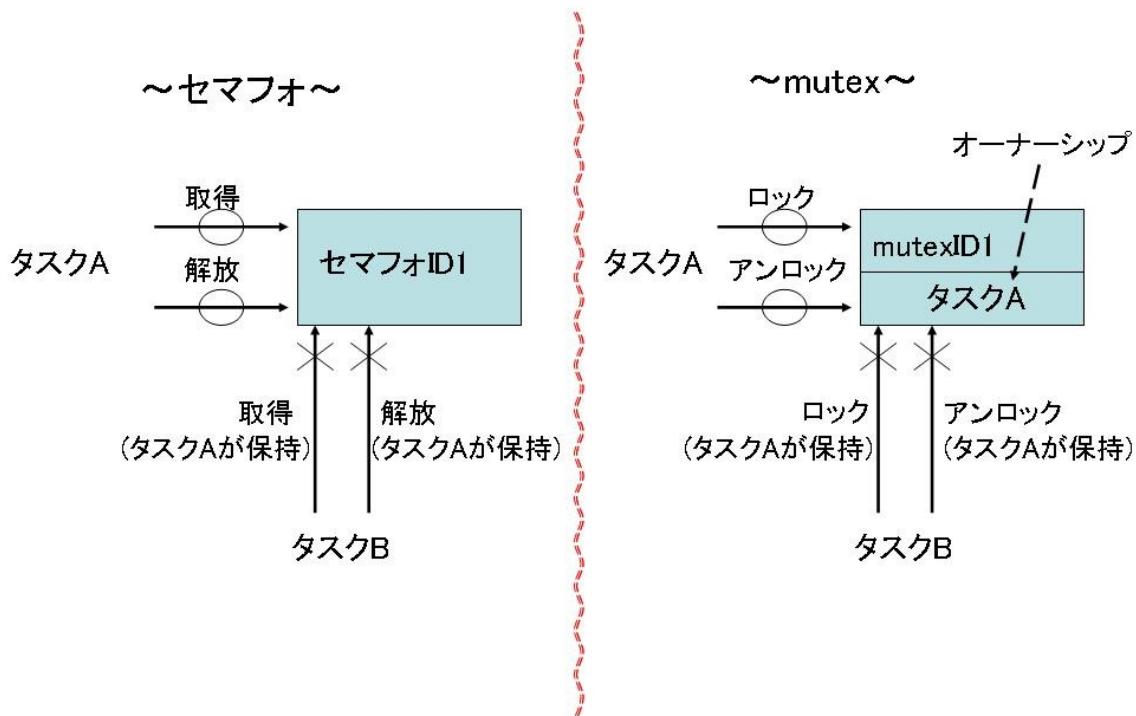


図8-8 セマフォとmutexの相違

なお、本研究OSのmutexは再帰ロック機能(ロックしたタスクは複数回ロック可能)を実装している。さらに、優先度逆転現象を防止する複数の優先度逆転防止アルゴリズムを備えている。これは9章で述べる。

(1) mutex 使用上の注意

注意点を以下の①と②に示す。

- ① 各タスクからのmutex制御処理(loc_mtx(), unl_mtx())が終わるまでは、作成した

mutex 自動割付 ID を書き換えたり、解放されたりしないよう(mutex を生成したタスクが自動割付 ID をローカルに確保し、そのタスクが retrun で終了すると、自動割付 ID は解放される)にする。

② mutex のロック及びアンロックは複数のタスクからできる。mutex を取得したタスクを終了させる場合、注意が必要である。なお、本研究 OS では、4.2.(3)節のタスク取得情報管理によるカーネルオブジェクト自動解放機能を実装している。

(2) mutex 構造体

mutex 構造体の情報を下図 8-9 に示す。この構造体は、mutex1 つの制御ごとに持つべき情報を格納した構造体である。

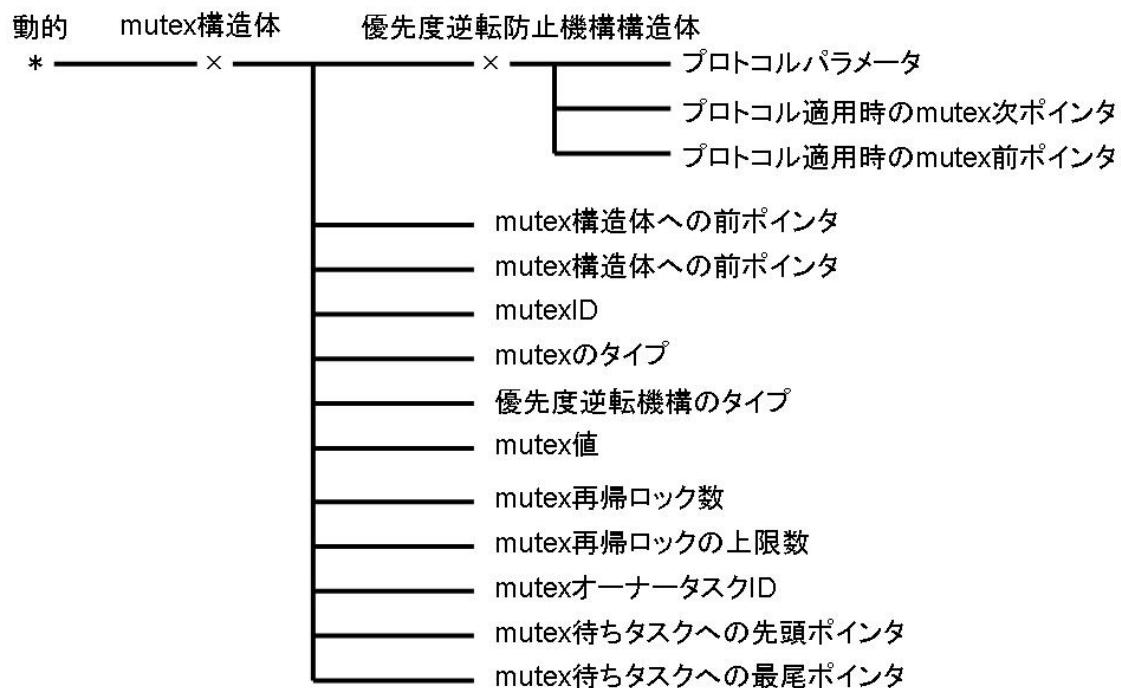


図8-9 mutex構造体

図の free ノードへの次ポインタと free ノードへの前ポインタはカーネル init 時(10.2. 節参照)に設定する。mutexID、mutex タイプ(静的型または動的型を記録)、属性(タスクをレディーへ戻すアルゴリズム)、優先度逆転防止機構のタイプ、オーナー ID(オーナシップ機能を実現するため)、再帰ロック最大数(再帰ロックを機能を実現にするため)、再帰ロック数(mutex 値をカウントさせれば、再帰ロック数はいらないかも)、mutex 値は acre_mtx() の ISR で初期設定する。mutex 待ちタスク次ポインタと前ポインタは loc_mtx() と unl_mtx() の ISR で操作する。mutex 構造体には mutex のオーナー優先度を設けていない。オーナー優先度を取得するには、オーナー ID を利用してタスク ID 変換テーブルで対象タスクにアクセスし、4.2.(1)節の図 4-2 の実行中内容が変化する情報→優先度にアクセスす

る。

なお、図の各プロトコル情報エリアは9章で具体的に述べる。

(3) mutex 情報構造体

タスク管理データ構造と同様に,mutex 管理データ構造に静的型と動的型を実装した。静的型は(1)を配列で管理し、動的型は(1)をリンクリストで管理する。さらに,mutexID 変換テーブルを採用し,4.2.(4.2)節の方式とした。これらを管理するための構造体を mutex 情報構造体とし,図 8-10 に示す。



図8-10 mutex情報構造体

各メンバは4.2.(4.2)節に加えて、4つのメンバがある。これについては、9章で述べる。

(4) タスクの mutex 待ち

タスクに mutex 待ちが発生した時は,mutex 待ちをした ID の mutex 構造体へキューイングし、タスクをレディー管理データ構造へ繋げず(シリアル割込み以外の割込み発生でタスクをレディー管理データ構造から抜き取る),タスク構造体へ4.2.(2)節の処理(タスク構造体へ待ち情報の設定)を行う。

9. 優先度逆転防止機構管理

9.1. 各優先度逆転防止機構の詳細

本節では、複数の実装方式を搭載した優先度逆転防止機構について述べる。具体的には優先度逆転防止ができる既存方式の割込みロック方式，Priority Inheritance Protocol，Delay Highest Locker's Protocol，Immediate Highest Locker's Protocol，Priority Ceiling Protocol，Stack Resource Policyについて述べる。

(1) 割込みロック方式

最も単純な方式であり、タスクがクリティカルセクション(以下CS)に入っている間、すべての割込みを禁止する悲観的方式である。

本研究OSでは、dis_dsp()システムコール及びena_dsp()システムコールで割込み無効モード及び有効モードを制御する実装である。

(2) Priority Inheritance Protocol(以下PIP)

この方式は、あるタスクがmutexロック中(CS実行中)に自分より優先度の高いタスクが、そのmutexを要求して待ち状態となる時に、一時的にmutexをロック(CS実行中)しているタスクまで優先度を継承する。そして、継承したタスクがmutexをアンロック(CS解除)した時に、元の優先度に戻す方式である。

本研究の実装では、mutexロックとアンロックISRに、別途PIPのISRを追加する実装である。

(3) Delay Highest Locker's Protocol(以下DHL)

この方式は、あるタスクがmutexロック中(CS実行中)に自分より優先度の高いタスクが、そのmutexを要求して待ち状態となる時に、一時的にmutexをロック(CS実行中)しているタスクの優先度をHighest Lockerへ上昇させる遅延型である。そして、そのmutexをアンロック(CS解除)した時に、元の優先度に戻す楽観的方式である。Highest Lokerはmutexをロックする可能性のある最も高い優先度タスクの優先度とする。この方式は、静的にアプリケーション側がHighest Lockerの計算が必要となる。

(4) Immediate Highest Locker's Protocol(以下IHL)

この方式は、他のタスクからすでにロック済みmutexへのロック要求(CS要求)が発行された時ではなく、mutexを初期ロック(CS要求)した瞬間に、Highest Lockerへ上昇させる即時型である。そして、そのmutexをアンロック(CS解除)した時に、元の優先度に戻す楽観的方式である。(2)と同様にHighest Lokerはmutexをロックする可能性のある最も高い優先度のタスクの優先度とし、事前にHighest Lockerの計算が必要となる。

(5) Priority Ceiling Protocol(以下PCP)

この方式は、あるタスクがmutexロック要求(CS要求)した時に、他のタスクによりロックされているすべてのmutexのCeiling Priorityよりも高ければ、mutexをロックできる(CSに入る事ができる)。そうでない場合は、mutexをロックする前にブロックする。つまり、mutex(CS)をまだロックされていなくても、ブロックされる事がある。mutexが1つもロック(CSがない)されていない時は、Ceiling Priorityはなく、すでにロックされているmutex(CS)を他のタスクが要求した場合は、(1)のPIP方式となる。そして、そのmutexをアンロック(CS解除)した時に、元の優先度へ戻す楽観的方式である。Ceiling Priorityは、mutex(CS)をロックする可能性のある最も高い優先度タスクの優先度とし、静的にPriority Ceilingの計算が必要となる。また、他のタスクがロックしているmutex(CS)のCeiling Priorityで最高のものを、Priority Ceilingとする。

(6) Stack Resource Policy(以下SRP)

この方式は、タスクがロック中のmutexをアクセスしようとした時に、ブロックされるのではなく、ロック中のmutexをアクセスするタスクを起動する時にブロックする。タスクのPreemption Levelがその時のSystem Ceilingより高くないと、そのタスクは起動できず、起動時の段階でタスクがブロックされる楽観的方式である。

本研究OSではSRPは未実装となる。これは今後の課題とする。

9.2. 優先度逆転防止機構の考察

本節では、9.1.節で述べた各優先度逆転防止機構の利害及び得失を考察(1)で行い、問題点を(2)で述べ、(3)で新規方式を提案する。

(1) 利害及び得失

① 割込みロック方式

利点として、全体で1つのCSを持つのでデッドロックの誘発、推移的優先度継承現象(チャーンブロッキング現象)、連続ブロッキング現象、デッドロックは発生しない。

欠点として、すべての割込みを無効とするので、CSの実行中は他のタスクは一切動けなくなり、制限が強い。つまり、CS実行中はイベント待ち及びメッセージ受信待ちはできず、タイマ拡張ルーチンやシリアル割込みハンドラも使用できない。つまり、拡張ルーチンを介したタスクの動的追加ができない。これは、DHL及びIHLのタスクが最高値固定となった場合と同様である。

② PIP方式

利点として、動的にタスクの優先度を継承させてるので、動的なタスクの追加やオーバーロード対応、イベント待ちやメッセージ待ち、タイマ拡張ルーチンが使用できる。さらに、動的なタスクの追加やオーバーロードへ対応できる。

欠点として、デッドロックの誘発、チェーンブロッキング現象、連続ブロッキング現象が発生する(②.1~②.3で説明)。ただし、デッドロックの誘発は、レディーへタスクを戻すアルゴリズムによって予防する事ができる。また、デッドロックの予防はできない(②.4で説明)。

②.1 デッドロックの誘発現象が発生するタスクセット

図9-1に示す。なお、タスクをレディーへ戻すアルゴリズムは優先度順としている

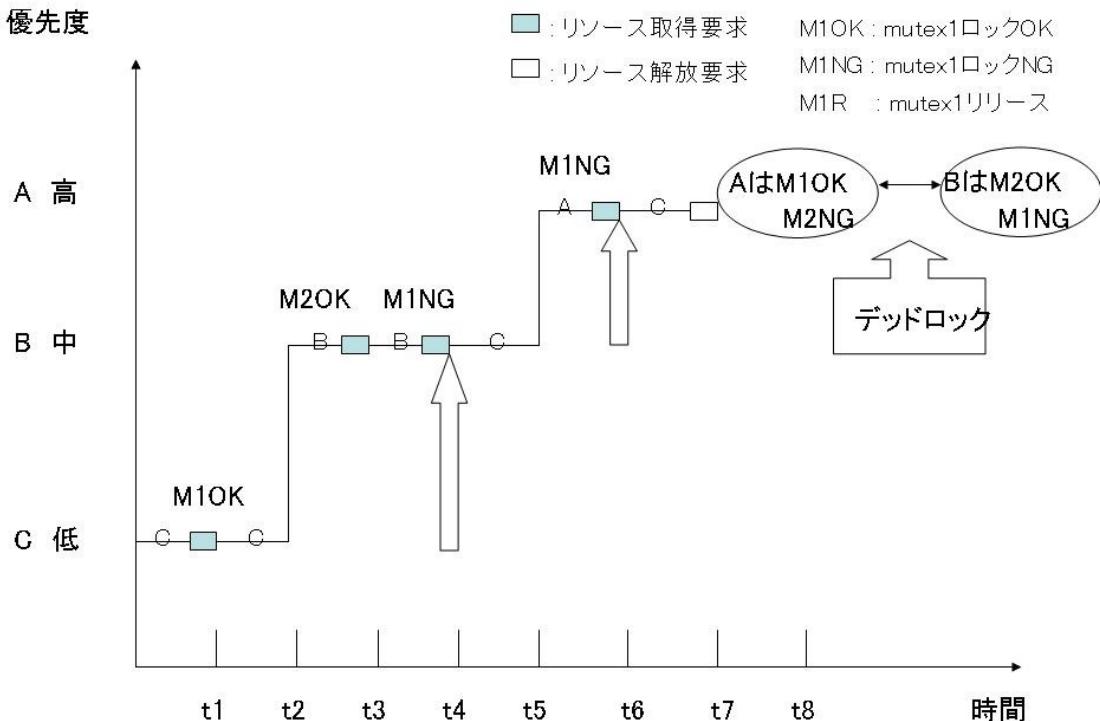


図9-1 デッドロックの誘発現象

- ・t1で低優先度タスクCがmutex1をロックする。
 - ・t2で中優先度タスクBが起動する。
 - ・t3で中優先度タスクBがmutex2をロックする。
 - ・t4で中優先度タスクBはmutex1をロックすることができず、低優先度タスクCの優先度を中優先度タスクBの優先度まで継承する。
 - ・t5で高優先度タスクAが起動する。
 - ・t6で高優先度タスクAはmutex1をロックすることができず、低優先度タスクCの優先度を高優先度タスクAまで継承する。
 - ・t7で低優先度タスクCがmutex1をアンロックし、高優先度タスクにmutex1を与える。
 - ・t8で高優先度タスクAはmutex2をロックしようとするが、中優先度タスクBによってすでにロックされているので待ち状態となる。ここでデッドロックとなる。
- t7の時点でmutex1の待ちタスクを優先度順でレディーへ戻したのでデッドロックとな

る。FIFO順でレディーへ戻せば,mutex1は中優先度タスクBへ与える事になり,デッドロックを誘発しない。

②.2 チェーンブロッキング現象が発生するタスクセット

図9-2に示す。なお,各mutexの待ち行列に繋がれるタスクは1つのみとなるので,タスクをレディーへ戻すアルゴリズムはFIFO順及び優先度順ともに結果は同じとなる。

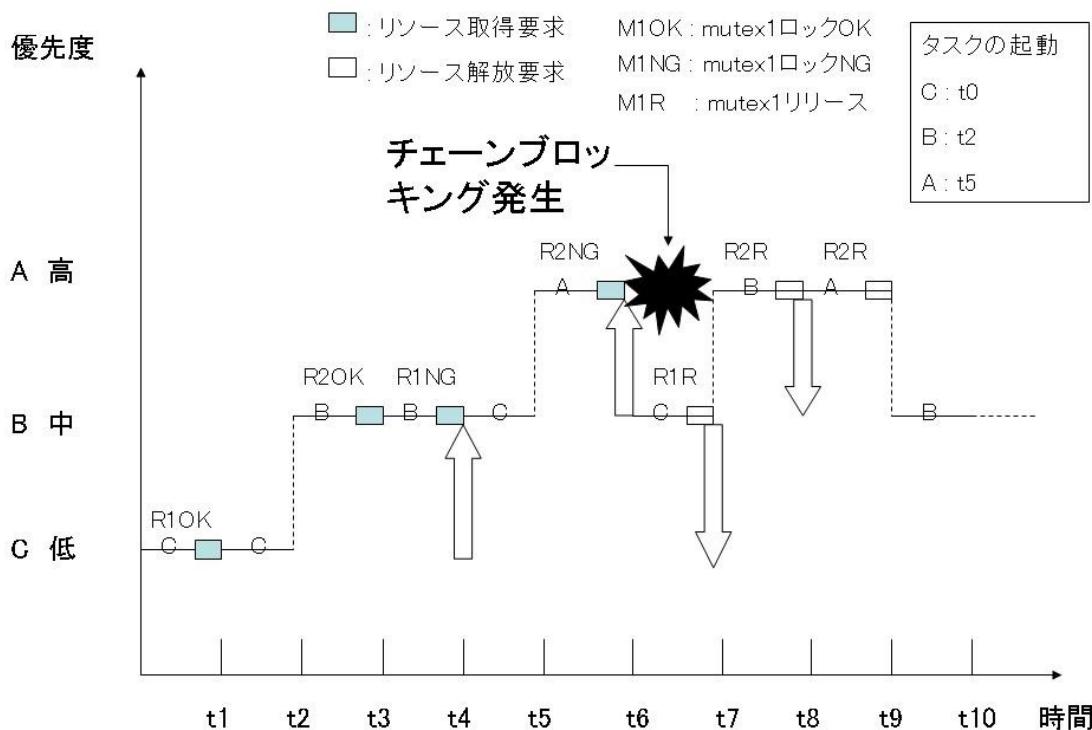


図9-2 チェーンブロッキング現象

- ・t1で低優先度タスクCがmutex1をロックする。
- ・t2で中優先度タスクBが起動する。
- ・t3で中優先度タスクBがmutex2をロックする。
- ・t4で中優先度タスクBはmutex1をロックすることができず,低優先度タスクCの優先度を中優先度タスクBの優先度まで継承する。
- ・t5で高優先度タスクAが起動する。
- ・t6で高優先度タスクAはmutex1をロックすることができず,中優先度タスクBの優先度を中優先度タスクAの優先度まで継承する。しかし,中優先度タスクは待ち状態となっているため,実行ができず低優先度タスクCへ実行が移る。ここでネストブロックし,チェーンブロッキング現象が発生する。
- ・t7で低優先度タスクCがmutex1をアンロックし,中優先度タスクBにmutex1を与える。
- ・t8で中優先度タスクBがmutex2をアンロックし,高優先度タスクAにmutex2を与える。

- ・ t9 で高優先度タスク A が mutex2 をアンロックする。
- チェーンブロッキング現象は、優先度の異なるタスクの数とロックをする mutex の数が増加するほど、ネストブロックされ、高優先度タスクの応答時間が予測できなくなる。

②.3 連続ブロッキング現象が発生するタスクセット

図 9-3 に示す。なお、②.2 と同様に、タスクをレディーへ戻すアルゴリズムは FIFO 順及び優先度順とともに結果は同じとなる。

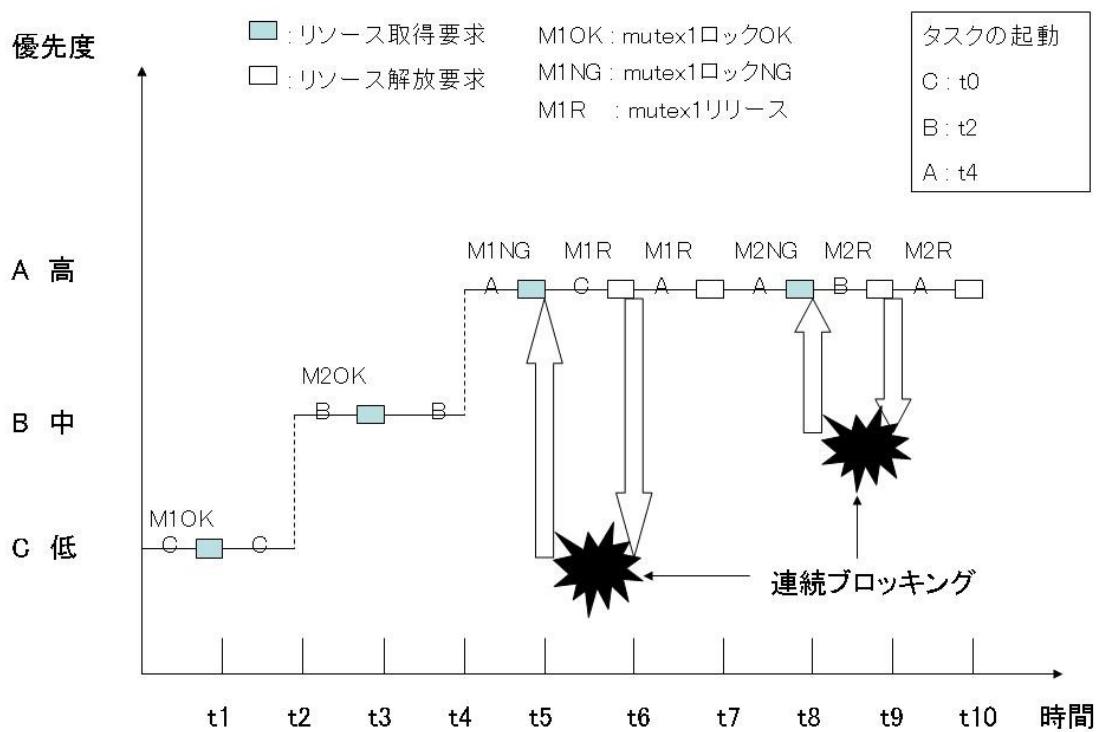


図 9-3 連続ブロッキング現象

- ・ t1 で低優先度タスク C が mutex1 をロックする。
- ・ t2 で中優先度タスク B が起動する。
- ・ t3 で中優先度タスク B が mutex2 をロックする。
- ・ t4 で高優先度タスク A が起動する。
- ・ t5 で高優先度タスク A は mutex1 をロックすることができず、低優先度タスク C の優先度を高優先度タスク A の優先度まで継承する。
- ・ t6 で低優先度タスク C が mutex1 をアンロックし、高優先度タスク A に mutex1 を与える。
- ・ t7 で高優先度タスク A が mutex1 をアンロックする。
- ・ t8 で高優先度タスク A は mutex2 をロックすることができず、中優先度タスク B の優先度を高優先度タスク A の優先度まで継承する。
- ・ t9 で中優先度タスク B が mutex1 をアンロックし、高優先度タスク A に mutex1 を与える。

- ・ t7 で高優先度タスク A が mutex1 をアンロックする。

高優先度タスクは t5 でブロックされ待ち状態となり、さらに t8 でブロックされ待ち状態となる。これはタスクが mutex を分散してロックするほど、高優先度タスクがブロックされる回数が増える。これが連続ブロッキング現象であり、何回も高優先度タスクがブロックされても優先度を高く設定した意味がなくなる。

②.4 デッドロックが発生するタスクセット

図 9-4 に示す。なお、②.2 と同様に、タスクをレディーへ戻すアルゴリズムは FIFO 順及び優先度順とともに結果は同じとなる。

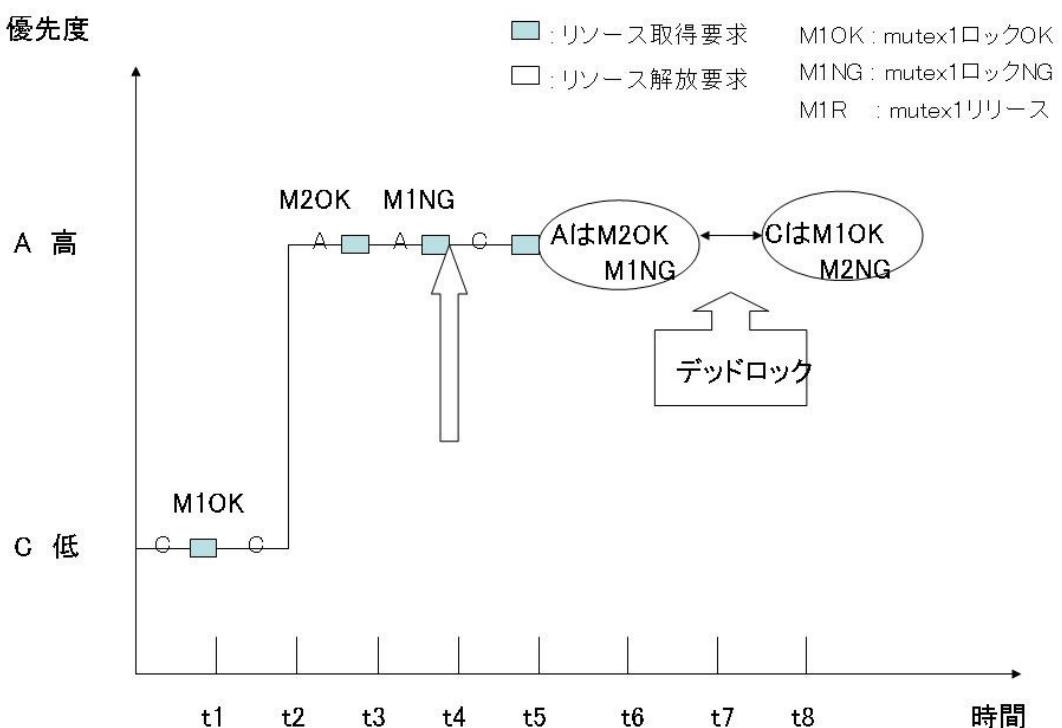


図9-4 デッドロックの発生

- ・ t1 で低優先度タスク B が mutex1 をロックする。
- ・ t2 で高優先度タスク A が起動する。
- ・ t3 で高優先度タスク A が mutex2 をロックする。
- ・ t4 で高優先度タスク A は mutex1 をロックすることができず、低優先度タスク B の優先度を高優先度タスク A の優先度まで継承する。
- ・ t5 で低優先度タスク B は mutex2 をロックしようとするが、高優先度タスク A によってすでにロックされているので待ち状態となる。ここでデッドロックとなる。

③ DHL 方式

利点として,デッドロックの誘発現象を防ぐ事ができる。

欠点として,最高値固定の上昇を遅延するので,チェーンブロッキング現象及び連続ブロッキング現象を防ぐ事ができず,デッドロックの予防はできない。また,アプリケーション側が最高値固定を静的に決定するので,動的優先度を使用するスケジューリングとの併用が困難であり,動的なタスクの追加やオーバーロードへの対応ができない。また,最高値固定となっている間は,イベント待ちやメッセージ待ちは使用できない。

④ IHL 方式

利点として,初期ロックの時点で最高値固定に上昇させてるので,デッドロックの誘発,チェーンブロッキング現象,連続ブロッキング現象を防ぐ事ができる。

欠点として,デッドロックの予防はできない。また,③と同様に動的スケジューリングと併用が困難であり,動的なタスクの追加やオーバーロードへの対応ができない。イベント待ちやメッセージ待ちも使用できない。

⑤ PCP 方式

利点として,デッドロックの誘発,チェーンブロッキング現象,連続ブロッキング現象を防ぐ事ができる。また,9.1.(5)節の機能より,デッドロックの予防が可能である。

欠点として,Ceiling Priority はアプリケーション側が静的に決定するので,③と同様に動的スケジューリングとの相性が悪く,動的なタスクの追加及びオーバーロード非対応となる。

⑥ SRP 方式

タスクを起動段階でブロックするので,動的スケジューリング法と併用できる。ただし,Preemption Level を静的に決定する事より,動的なタスクの追加やオーバーロードに対応できない。

(2) 既存方式の問題点

優先度逆転防止機構を使用する時に,静的に決定する項目があると,動的なタスクの追加ができず,アプリケーション側の制限が強くなる。また,カーネル内部構造によって,mutex ロック順リストの作成が必要となり,オーバーヘッドが大きくなる。よって,PIP 方式を選択できる。しかし,PIP 方式を採用する時は,デッドロックの誘発,チェーンブロッキング現象,連続ブロッキング現象を考えなくてはならず,アプリケーション側の煩瑣なテストが必要となる。

9.3. 優先度逆転防止機構の新規方式 (Virtual Priority Inheritance Protocol(以下 VPI))

9.2.(2)節の問題点を解決するために, Virtual Priority Inheritance Protocol(以下 VPI)を提案する。この優先度逆転防止機構は,VPI を使用する時に,静的に決定する項目がなく,動的なタスクの追加及びオーバーロードへの対応ができる。そして,イベント待ちや

メッセージ待ちに対応できる。さらに、デッドロックの誘発、チェーンブロッキング現象、連続ブロッキング現象を防ぐ事ができ、デッドロックの予防ができる方式である。

アルゴリズムは、mutex のロックをタスク 1 つに留め、そのタスクが CS をロックしている時は、他のタスクは一切 mutex をロックできない。また、mutex をロックできたタスクは他の mutex をロックできる。つまり、全体から見て mutex をロックできるのは 1 つのタスクのみとなる制限をつける。ここまでを図 9-5 に示す。

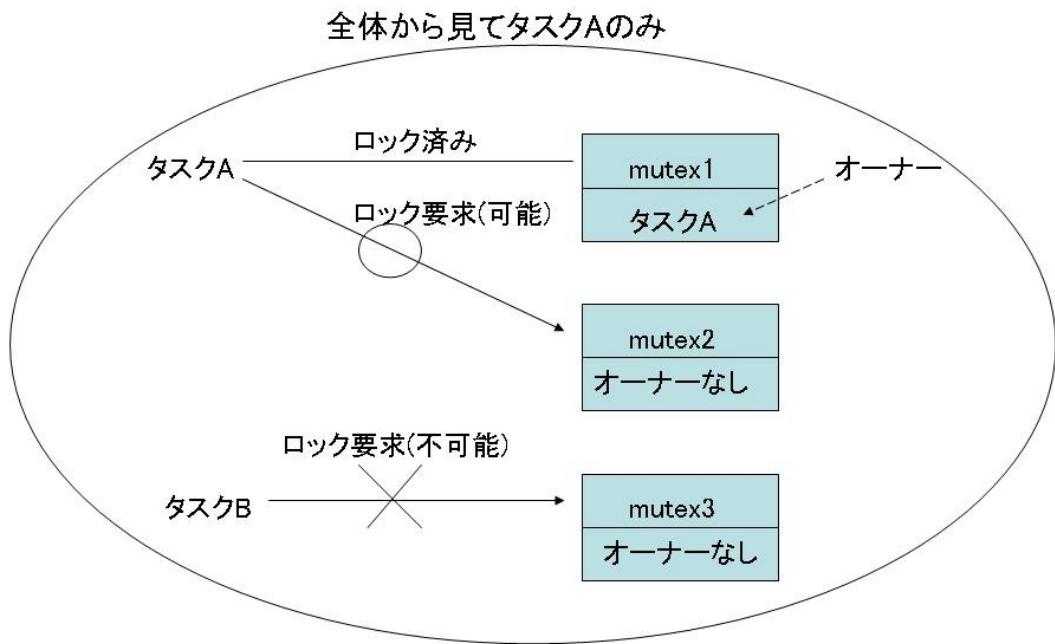


図9-5 mutexの制限

図 9-5 の機能を実現するために、Virtual mutex(実際には存在しない。機能は mutex と同じ)を設け、要求した physical mutex(実際に要求した mutex)をロックするには、まずタスクは Virtual mutex をロックしなくてはならない。この Virtual mutex がロック出来れば、要求した physical mutex を無条件でロックできる 2 段階方式とする。図 9-6 に示す。

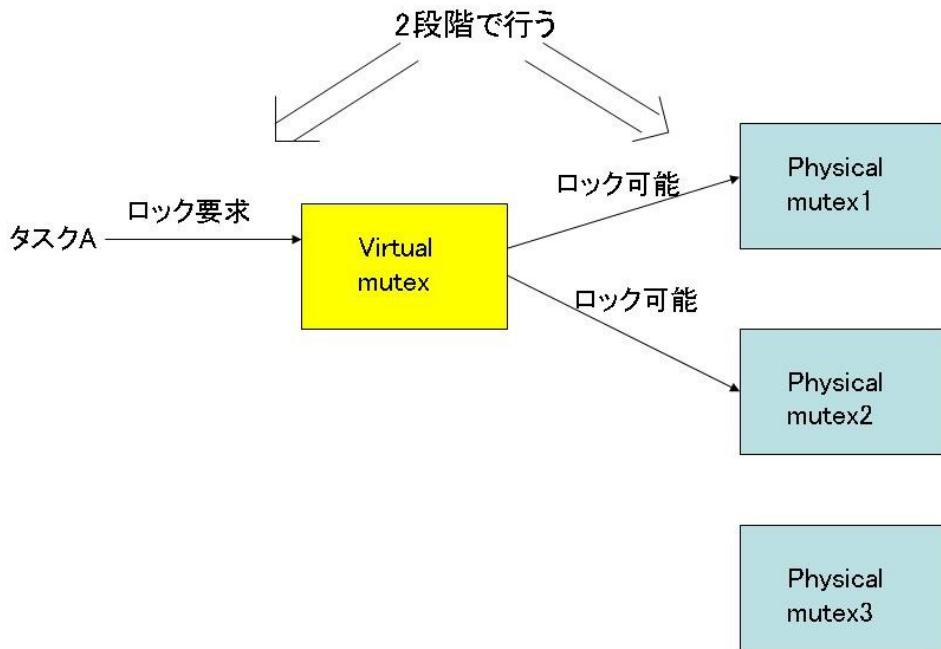


図9-6 タスクによるPhysical mutexのロック

Virtual mutexをロックできるのは、一度に1つまでのタスクであり、Virtual mutexをロックできない時は、Virtual mutexの待ち行列にタスクを管理する。また、優先度逆転現象を防ぐために、タスクがVirtual mutexを要求して待ち状態となる時に、一時的にVirtual mutexをロックしているタスクまで優先度を継承する。図9-7に示す。

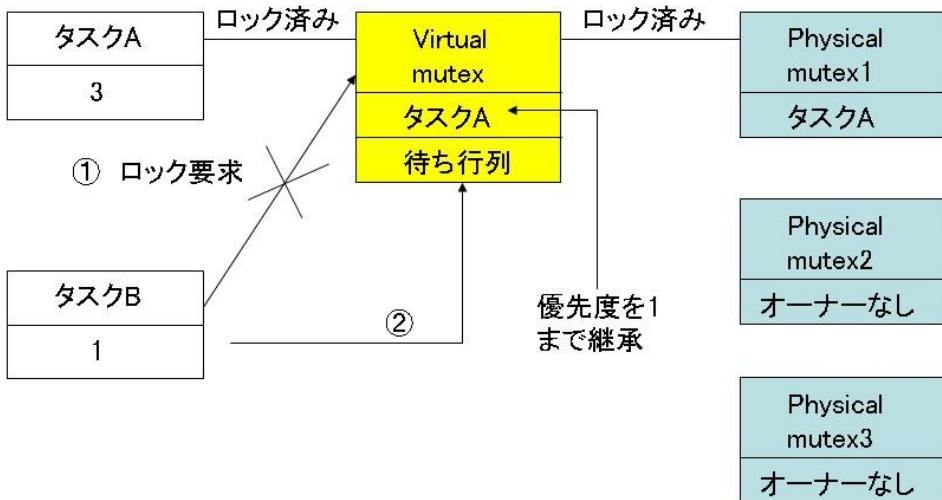


図9-7 Virtual mutex待ちと優先度継承

- I タスク B がすでにロックしている Virtual mutex を要求
 - II タスク B はロックできず, Virtual mutex の待ち行列へつなぐ
 - III タスク B の優先度を Virtual mutex オーナーであるタスク A まで継承
- そして, 繙承したタスクが mutex をアンロック(CS 解除)した時に, 元の優先度に戻す方式である。

この方式は, (1)の割込みロック方式を応用し, (2)のPIP 方式を加えたものである。全体で 1 つのタスクしか mutex をロックしていないならば, 優先度逆転現象も, デッドロックも起こらないという考えに基づいた。以下①~④に VIP を使用した時のデッドロックの誘発防止, チェーンブロッキング現象防止, 連続ブロッキング防止, デッドロック予防を示す。なお, タスクセットは(1).②.1~(1).②.4 を使用する。

- ① デッドロック誘発の防止

図 9-8 に示す。

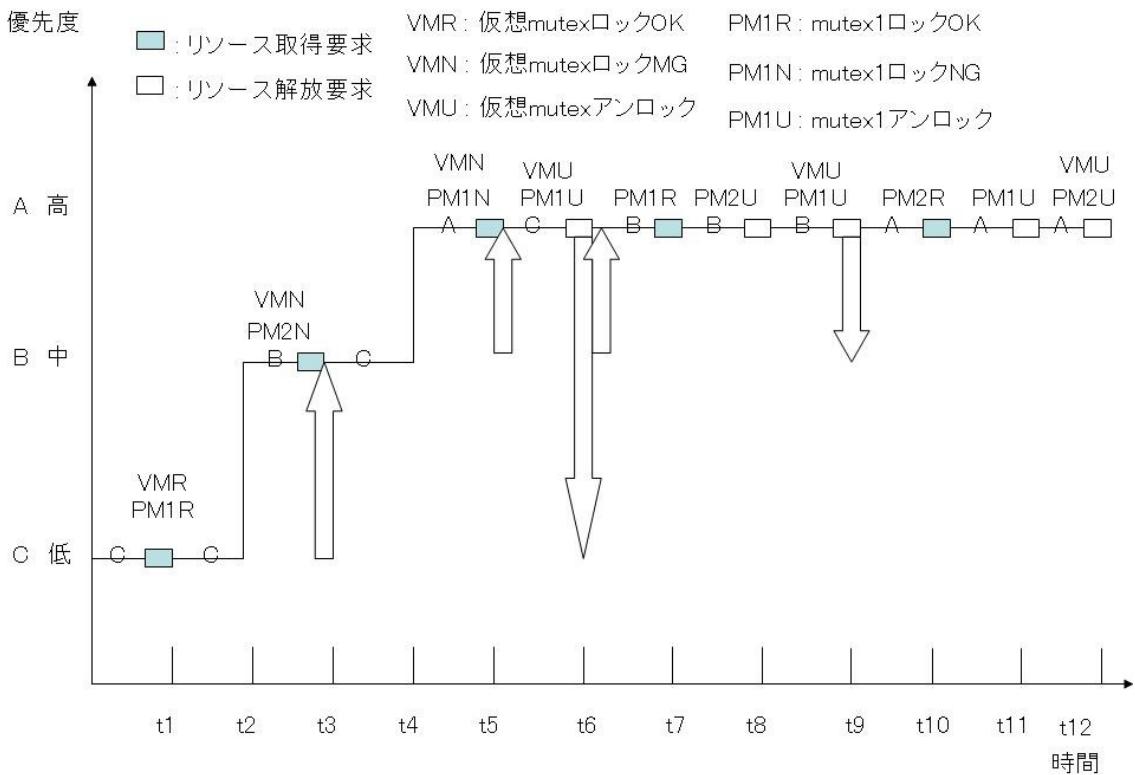


図9-8 VPIによるデッドロックの誘発現象防止

- ・t1で低優先度タスクCがvirtual mutexとphysical mutex1をロックする。
- ・t2で中優先度タスクBが起動する。
- ・t3で中優先度タスクBはvirutal mutexをロックできず、待ちとなり、低優先度タスクCが優先度を継承する。
- ・t4で高優先度タスクAが起動する。
- ・t5で高優先度タスクAはvirtual mutexをロックできず、待ちとなり、低優先度タスクCが優先度を継承する。
- ・t6で低優先度タスクCはvirtual mutexとphysical mutex1をアンロックし、優先度を元に戻す。この時、待ちとなっていた中優先度タスクBにphysical mutex2を与え、優先度を継承する。
 - ・t7で中優先度タスクBはphysical mutex1をロックする。
 - ・t8でphysical mutex2をアンロックする。
 - ・t9で中優先度タスクBはvirtual mutexとphysical mutex1をアンロックし、優先度を元に戻す。この時、待ちとなっていた高優先度タスクAにvirtual mutexとphysical mutex1を与える。
- ・t10で高優先度タスクAはphysical mutex2をロックする。
- ・t11で高優先度タスクAはphysical mutex1をアンロックする。
- ・t12で高優先度タスクAはvirtual mutexとphysical mutex2をアンロックする。

② チェーンブロッキング現象防止

図9-9に示す。

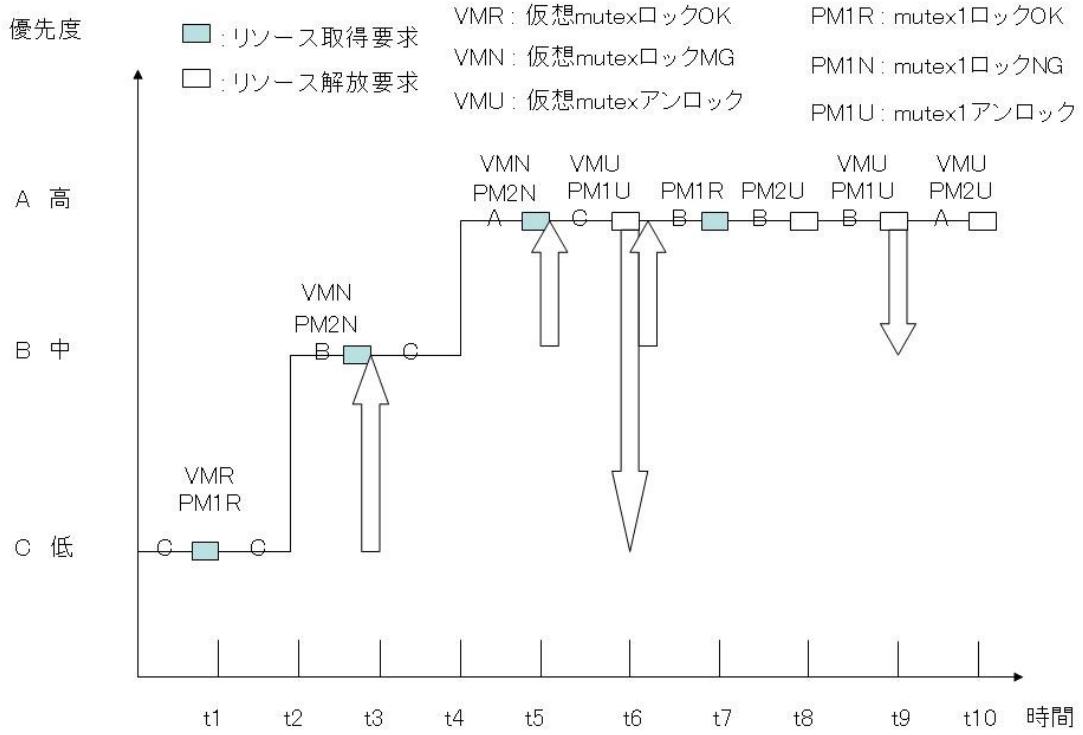


図9-9 VPIによるチェーンブロッキング現象防止

- ・t1で低優先度タスクCがvirtual mutexとphysical mutex1をロックする。
- ・t2で中優先度タスクBが起動する。
- ・t3で中優先度タスクBはvirutal mutexをロックできず、待ちとなり、低優先度タスクCが優先度を継承する。
- ・t4で高優先度タスクAが起動する。
- ・t5で高優先度タスクAはvirtual mutexをロックできず、待ちとなり、低優先度タスクCが優先度を継承する。
- ・t6で低優先度タスクCはvirtual mutexとphysical mutex1をアンロックし、優先度を元に戻す。この時、待ちとなっていた中優先度タスクBにphysical mutex2を与え、優先度を継承する。
- ・t7で中優先度タスクBはphysical mutex2をロックする。
- ・t8で中優先度タスクBはphysical mutex2をアンロックする。
- ・t9で中優先度タスクBはvirtual mutexとphysical mutex1をアンロックし、優先度を元に戻す。この時、待ちとなっていた高優先度タスクAにphysical mutex1を与える。
- ・t10で高優先度タスクAはvirtual mutexとphysical mutex2をアンロックする。

③ 連続ブロッキング現象防止

図9-10に示す。

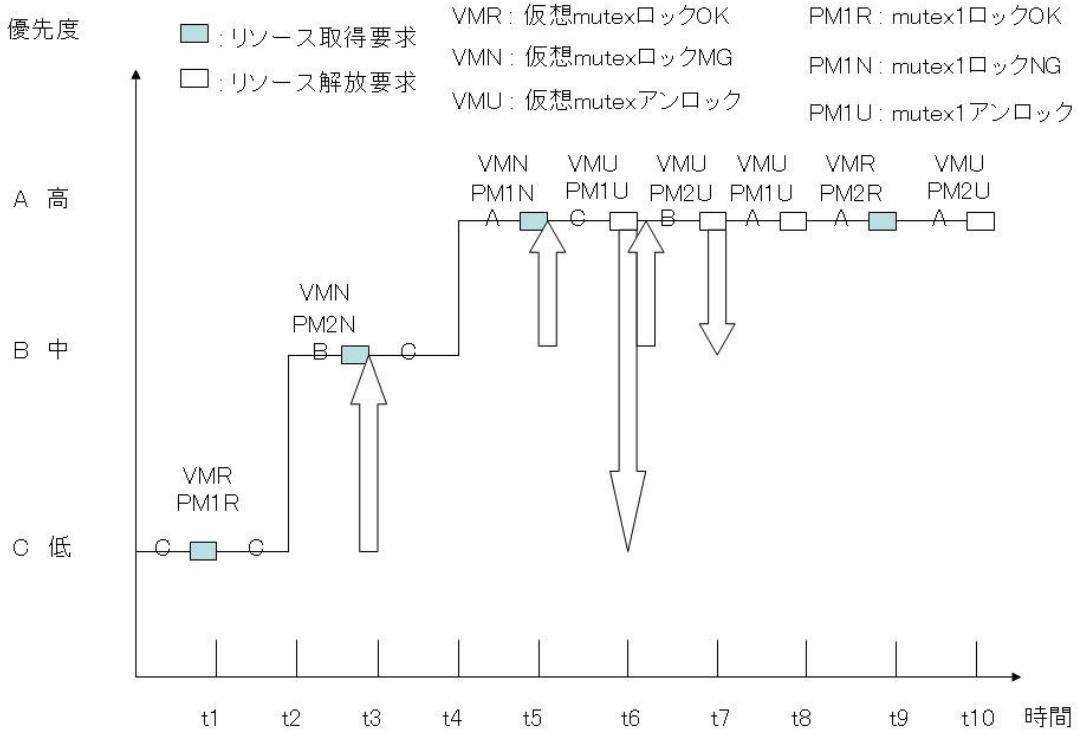


図9-10 VPIによる連続ブロッキング現象防止

- ・t1で低優先度タスクCがvirtual mutexとphysical mutex1をロックする。
- ・t2で中優先度タスクBが起動する。
- ・t3で中優先度タスクBはvirutal mutexをロックできず、待ちとなり、低優先度タスクCが優先度を継承する。
- ・t4で高優先度タスクAが起動する。
- ・t5で高優先度タスクAはvirtual mutexをロックできず、待ちとなり、低優先度タスクCが優先度を継承する。
- ・t6で低優先度タスクCはvirtual mutexとphysical mutex1をアンロックし、優先度を元に戻す。この時、待ちとなっていた中優先度タスクBにphysical mutex2を与え、優先度を継承する。
- ・t7で低優先度タスクCはvirtual mutexとphysical mutex1をアンロックし、優先度を元に戻す。この時、待ちとなっていた高優先度タスクAにphysical mutex1を与える。
- ・t8で高優先度タスクAはvirtual mutexとphysical mutex1をアンロックする。
- ・t9で高優先度タスクAがvirtual mutexとphysical mutex2をロックする。
- ・t10で高優先度タスクAはvirtual mutexとphysical mutex1をアンロックする。

④ デッドロック予防

図9-11に示す。

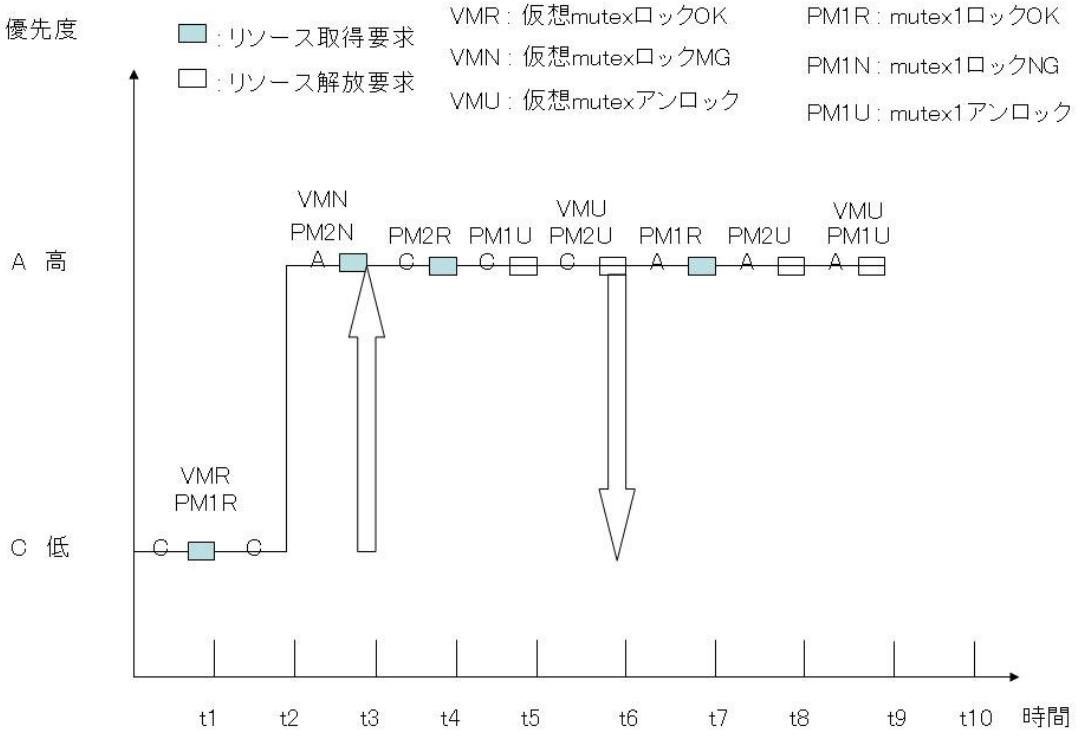


図9-11 VPIによるデッドロック予防

- ・t1で低優先度タスクCがvirtual mutexとphysical mutex1をロックする。
- ・t2で高優先度タスクAが起動する。
- ・t3で高優先度タスクAはvirutal mutexをロックできず、待ちとなり、低優先度タスクCが優先度を継承する。
- ・t4で低優先度タスクはphysical mutex2をロックする。
- ・t5で低優先度タスクはphysical mutex1をアンロックする。
- ・t6で低優先度タスクCはvirtual mutexとphysical mutex2をアンロックし、優先度を元に戻す。この時、待ちとなっていた高優先度タスクAにphysical mutex2を与える。
- ・t7で高優先度タスクAはphysical mutex1をロックする。
- ・t8で高優先度タスクAはphysical mutex2をアンロックする。
- ・t9で高優先度タスクAはvirtual mutexとphysical mutex1をアンロックする。

(4) VIPの欠点

VIPの利点は(3)で述べた。VPIは欠点として、PCP同様にタスクがmutexをロックする場合、必ず1回はブロックされてしまう。よって、アクセスが集中するmutexにのみVPIを使用し、集中しない場合は優先度逆転防止機能なしで制御させる事が望ましい。

9.4. 本研究OS上の実装

本節では、9.1.(2)～(6)節で述べた方式と9.3.節で述べた新規方式の本研究OSにおける実装を説明する。なお、9.1.(1)節の割込みロック方式はdis_dsp()とena_dsp()システムコールの実装のみとなるので、本節では述べない。

また9.1.(5)節のPCP方式と9.3.のVPI方式は、バグがあり、連続ブロックへの対応ができない。9.1.(6)節のSRP方式は未実装となる。これは、今後の課題とする。

9.2.節で述べたように、優先度逆転防止機構には静的型(DHL方式、IHL方式、PCP方式)と動的型(PIP方式、PCP方式)がある。パラメータ情報の有無があるので、静的型と動的型で実装が異なる。なお、動的型であるPIP方式とVPI方式でも実装が多少異なる。

これらを実装するにあたって、2通りの方式を考えた。(1)～(2)に示す。

(1) 各タスクにmutexの取得順と取得数をテーブルで管理

8.3.(2)節の図では、各プロトコル情報エリアはプロトコルパラメータのみとし、8.3.(3)節の図にPCP方式とDHL方式、IHL方式ごとに取得順と取得数テーブルを設ける。そして、cre_mtx()システムコール発行前に取得順と取得数テーブルを設定するシステムコールを発行し、Ceiling PriorityまたはHighest Lockerはカーネルが計算し算出する方式である。取得順と取得数テーブルを設ける事によって、(2)で説明する欠点を解消する事ができる。

この方式の欠点として、取得順と取得数テーブルを設定するシステムコールは、パラメータの設定が多くなり、ユーザにとって煩瑣な操作となる。さらに、同時に複数の優先度逆転機構を導入するとOS内部での管理が複雑化し、ユーザ側も管理が複雑化する。

(2) Ceiling PriorityまたはHighest lockerとなったmutexを連結リストで管理

8.3.(2)節の図の各プロトコル情報エリアは、Ceiling Priority(PCP方式)またはHighest locker(DHL方式とIHL方式)となったmutexを接続しておく(Ceiling PriorityまたはHighest Lockerはユーザが決定する)。プロトコルパラメータでは、PCPでは、そのmutexのCeiling Priorityを記録する。DHLとIHLでは、Highest lockerを記録する。そして、先頭を8.3.(3)節の図で保持しておく。

なお、静的型を上記の実装としたのは、PCP方式だと連結リストを探索する事によってPriority Ceiling値の取得が容易となる。また、ユーザから優先度逆転防止機構の使用時のパラメータ設定の煩瑣をなくすのと、OS内部での管理のしやすくするためである。

この方式と欠点として、Ceiling PriorityまたはHighest Lockerとした優先度のタスクがmutex制御処理(loc_mtx(), unl_mtx())を終えた後も、Ceiling PriorityまたはHighest Lockerは変更されない点である。下図9-12に示す。

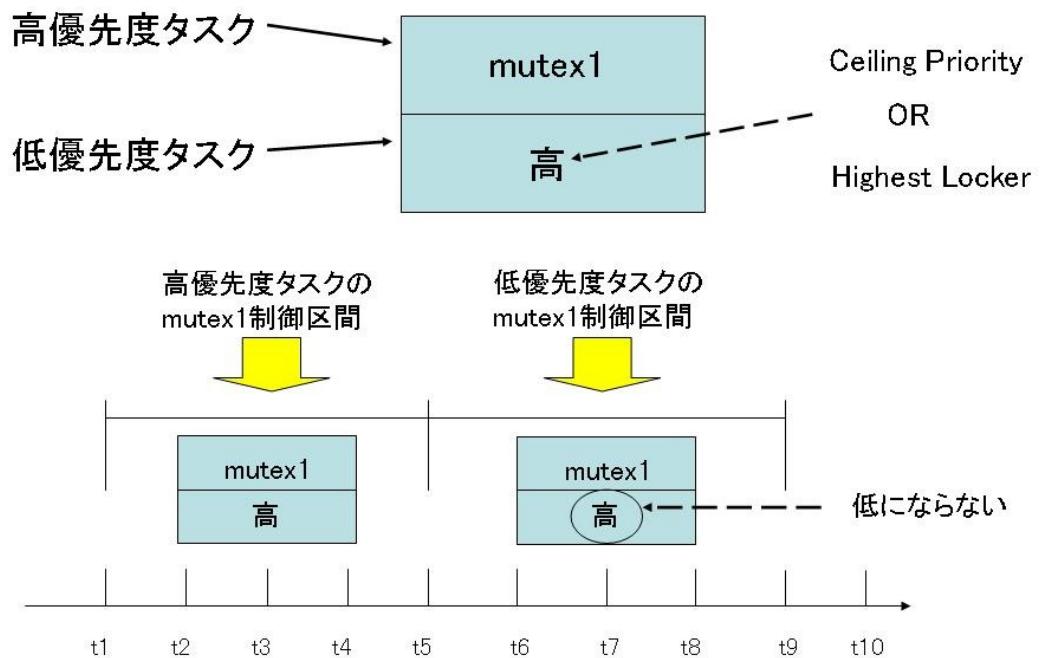


図9-12 Ceiling PriorityまたはHighest Lockerが未変更時

(1)と(2)の欠点の差異から、管理の容易さに着目し(2)の方式を採用した。Ceiling Priority または Highest Locker は cre_mtx() の時にユーザが設定を行なう。

また、VPI 方式の仮想 mutex は初回の cre_mtx() で VPI が指定された時に作成し、8.3.(3)節の図に接続しておく。二回目の cre_mtx() で VPI が指定されても仮想 mutex の作成は行なわない。

10. init 管理

10.1. ブートシーケンスと OS メモリマップ

本節では、ブートシーケンスと OS メモリマップについて述べる。

(1) ブートシーケンス

本研究OSでは、ブートローダの起動をリセットベクタで行う。ブートローダをリセットベクタで起動させるので、OSは使用する事はできない。そのため、OSの起動はセグメントのエントリーポイントの取得(OS側のリンクスクリプトによって取得処理を記述する)によって行う。

セグメントとは、リンク後に生成され、主にブートローダがロードの際に参照するメモリ展開の単位である(主に実行ファイル)。そして、ブートローダは実行ファイルのセグメント情報を参照して、そこに記述されているように実行ファイルの領域を(セグメント単位)DRAM上に展開する。

OSを起動させる方法であるOSセグメント展開について以下の(1)~(3)が考えられる。

① セグメント2段階で扱う(OS実行に使用するDRAMで行う)

OSを起動するにあたって、ブートローダのloadコマンドにより、XMODEMを通じてOSのELFファイル(実行ファイル)を転送する。転送したELFファイルをOSメモリマップの.buffer領域に退避させる。次にブートローダのrunコマンドにより、.buffer領域をセグメント単位でOSが実効できる領域へコピーする、という方式である。この方式だと内部構造が簡単化する。しかし、OSが使用できるDRAM領域を消費するので、メモリ効率が悪い。

② セグメントを2段階で扱う(ブートローダ実行に使用するROMで行う)

OSを起動するにあたって、ブートローダのloadコマンドにより、XMODEMを通じてOSのELFファイル(実行ファイル)を転送する。転送したELFファイルはブートローダメモリマップの.buffer領域に退避させる。次にブートローダのrunコマンドにより、.buffer領域をセグメント単位でDRAMへコピーする、という方式である。この方式だと内部構造が簡単化する。

③ セグメントを1段階で扱う

OSを起動するにあたって、ブートローダのload & runコマンドにより、XMODEMを通じてOSのELFファイルを転送する。受信時に.buffer領域を設けずにセグメントを直接DRAMへ展開する、という方式である。この方式だとメモリ効率が良いが、内部構造が複雑化する。

学習向けOSなので、2の方式を採用している。なお、ブートローダの構造は、オープンソースから大幅な改良を施していないので、本論文では述べない(オープンソースとほぼ同じとなるので参考文献[]参照)。

エントリーポイント取得後は、OSのスタートアップ(スタートアップでH8アーキテクチャ依存部となるブートスタックの設定を行う)が実効され、次にOSのmain関数が実効され

る。OSのmain関数ではinitタスク生成処理を呼び出す。これは10.2節で述べる。

(2) OSメモリマップ

本研究OSのメモリマップは、6.5節の図6-18を見ていただきたい。

OSは外部RAMであるDRAMと内部RAMであるDDR3SDRAMで動作させる。DRAMのメモリマップはコードを配置するコードセグメントとなる。DDR3SDRAMのメモリマップは、割込みハンドラを登録するセグメントであるソフトウェアベクタセグメント、マルチタスクを実現するためのスタックであるユーザスタックセグメント、割込みに使用する割込みスタックセグメント、スケジューラのスイッチングに使用するスケジューラセグメントとなる。スケジューラセグメントは本研究OSオリジナルセグメントである。これらはリンクスクリプトで生成する。

10.2. カーネルinit

本節では、カーネルのinitタスク生成ルーチン及びinitタスクについて述べる。

(1) initタスク生成ルーチン

initタスク生成ルーチンはカーネルルーチンとなるので、割込み無効モードで起動する。initタスク生成ルーチンの処理を下図10-1に示す。なお、この初期化処理の順番が重要となる。

```
void start_init_tsk(タスク起動番地:*func, タスク名:*name, 優先度:priorit, タスクスタック量:stacksize,
                     タスク第一引数:argc, タスク第二引数:*argv[])
{
    tskid; //initのタスクID

    set_schedul_isr(スケジューラ情報); //デフォルトでのスケジューラ設定

    dispatch_init(); //ディスパッチャ初期化
    mem_init(); //heap領域の初期化
    tmrdriver_init(); //タイマドライバ初期化
    schdul_init(); //スケジューラ初期化
    ready_init(); //レディー管理データ構造初期化
    tsk_init(); //タスク周り初期化
    sem_init(); //セマフォ周り初期化
    mbx_init(); //メールボックス周りの初期化
    mtx_init(); //mutex周りの初期化
    alm_init(); //アラームハンドラ周りの初期化
    cyc_init(); //周期ハンドラ周りの初期化
    memset(handler, 0, ハンドラサイズ); //割込みハンドラ初期化

    kernelrte_def_inf(システムコールハンドラ情報); //システムコールハンドラの登録
    kernelrte_def_inf(例外用ハンドラ情報); //例外用ハンドラの登録
    kernelrte_def_inf(タイマ割込みハンドラ情報); //タイマ割込みハンドラの登録
    kernelrte_def_inf(NMI割込みハンドラ情報); //NMI割込みハンドラの登録

    tskid = kernelrte_acre_tsk(タイプ, func, name, priority, stacksize, 0, 0, 0, 0, argc, argv); //タスク生成
    kernelrte_sta_tsk(tskid); //タスク起動

    (*ディスパッチャ起動番地)(起動するタスクのスタックポインタ); //ディスパッチャの呼び出し
    //ここには戻ってこない
}
```

図10-1 initタスク生成ルーチンの擬似コード

図 10-1 に示したように、最初にデフォルトのスケジューラを登録する処理(6.5 節のスケジューラセグメント登録処理)を行う。これは、スケジューラの初期化でスケジューラセグメントを読み込む処理があるからである。また、スケジューラの初期化処理やカーネルオブジェクト初期化処理(レディー、タスク、セマフォ、メールボックス、mutex、アラームハンドラ、周期ハンドラ、割込みハンドラ等)で動的メモリ(heap 領域)を使用するので、スケジューラ初期化よりも動的メモリ初期化を先に行う。

システムコールハンドラ、例外用ハンドラ、タイマ割込みハンドラ、NMI 割込みハンドラはユーザは登録できず、カーネルで設定するので、init タスク生成ルーチンで設定している。

最後に init タスクを生成して、ID 変換テーブルに設定(ID0 として設定)し、起動してレディー管理データ構造へ挿入する。そして、ディスパッチャを呼び出す(init タスクのみとなるので、スケジューラを呼ぶ必要はない)。なお、生成と起動はカーネルルーチン実行中から呼ぶので、システムコールは使用できない。

(2) init タスク

init タスクでは、シリアルドライバの初期化とシリアル割込みハンドラの設定を行い、init タスク生成ルーチンから引き継がれている割込み無効モードを有効にし、ユーザからのタスクセットの操作を待機する。この待機している時は、init タスクは省電力モードへ以降し、デーモンとなっている。init タスクはユーザからのタスクセットの操作を受けると、シリアル割込みハンドラの延長でユーザタスクセットを生成及び起動し、ユーザタスクセットが終了すると init タスクへ戻り、再びユーザからの操作待ちをする。なお、ユーザからのタスクセット操作については、付録の操作マニュアルを見ていただきたい。

10.3. タスクのスタートアップとエンド

本節では、タスクのスタートアップとエンドについて述べる。

(1) タスクのスタートアップ

5.1.(3)で述べたように、タスク中断時(割込み発行時)ではプログラムカウンタ、モードレジスタ、フラグレジスタはハードウェア回路がスタックへ退避する。タスクスタート時では、割込みは発行されないので、これらを手動で設定する。具体的には、`cre_tsk()` の ISR 内でリンククリプトで定義しているタスクスタック領域から必要量を確保し、初期化を行う。下図 10-2 に示す。なお、H8 アーキテクチャでは、フラグレジスタとモードレジスタはCCR としてまとめられ、プログラムカウンタと同一ワードで管理される。

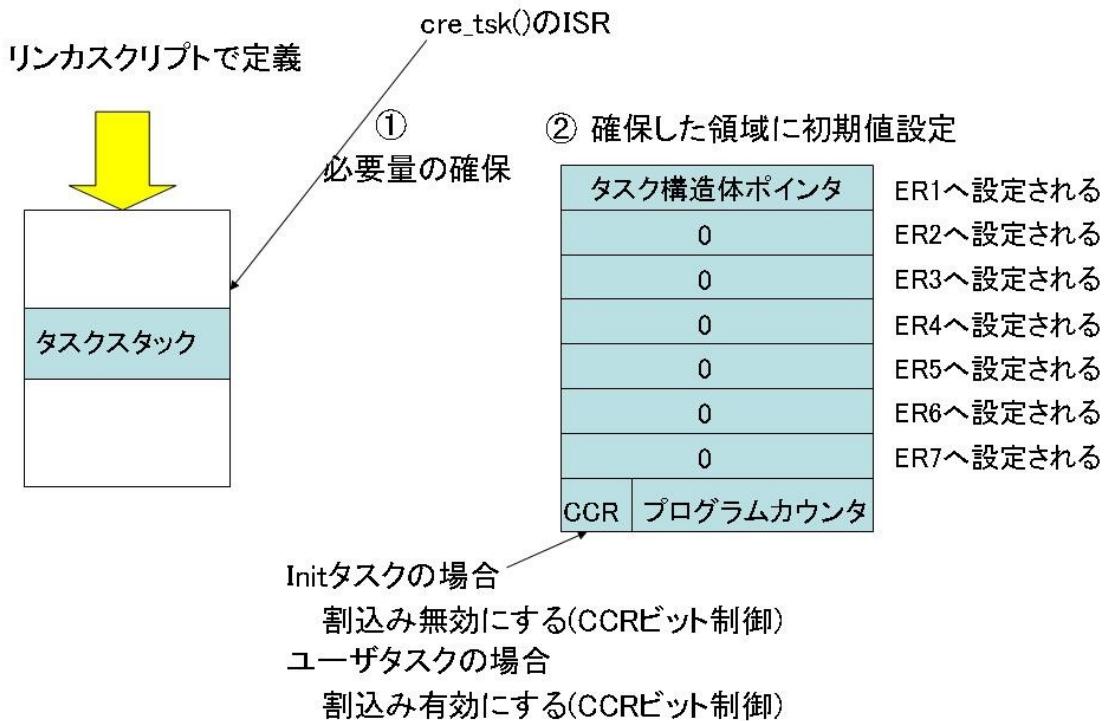


図10-2 タスクstackの確保と設定

① タスクstack必要量を確保

タスクstackから必要量を確保する

② タスクstackへ初期値設定

図のように汎用レジスタに設定する値は、スタートアップ時は0(ER0にはタスク構造体のポインタを設定(ディスパッチャがタスクスタートアップ関数を呼び出すときのパラメータとして使用する))とし、プログラムカウンタにはタスクスタートアップ関数を設定する。initタスクの場合は割込み禁止タスクとし、ユーザタスクは割込み有効タスクとするため、CCRの値を制御している。そして、タスクスタートアップ関数で、タスク構造体のargcとargvをパラメータとしてタスク起動番地を呼び出す(タスクスタートアップ関数はディスパッチャが呼ぶ)。なお、汎用レジスタER2とER1にタスクのargvとargcを設定すれば、タスク構造体のargcとargvの領域は不要となるが、移植性の観点から行わない。

(2) タスクのエンド

タスクがreturnで終了すると、(1)で述べたタスクスタートアップ関数へ戻ってくる。ここで、自タスクの終了であるext_tsk()システムコールを発行する実装である。なお、タスクスタートアップ関数もタスク自体の処理となる(そのため、システムコールを発行できる)。

11. 他OSとの可読性比較調査結果

本節では、他OSとの可読性比較調査の結果について述べる。

比較するOSは本研究OS、kozos(参考文献[18])、H8/OS(参考文献[20])、toppers/sspカーネル(参考文献[21])とする。

② kozos・・・使用したオープンソースOS

対象とした理由：この一年間でどの程度の本研究がされたか確認するためである。

③ H8/OS・・・本研究OS及び②とは関係ないH8専用学習オープンソースOS

対象とした理由：本研究OSはH8専用の学習組込みOSであるため、他のH8専用の学習組込みOSとどう異なるかを確認するためである。

④ topplers/sspカーネル・・・本研究OS及び②、③とは関係ない学習オープンソースOS

対象とした理由：④は完成度が高く学習向け組込みOSの代表格であるため(資源汎用性がある)である。

比較項目はコメント行、ソースコード行、コメント比率、アセンブラー行、アセンブラ比率、条件付きコンパイルマクロ件数、goto文件数、関数マクロ件数となる。

⑤ コメント行

理由：コメントを多くすれば、コードの可読性が上がると考えられるためである。なお、約400のOS内部関数ヘッダコメント及びマクロコメント、選択及び反復処理の入り口にコメントを記載した(付録のソースコード参照)。

⑥ ソースコード行(コメント+コード)

理由：OSの大きさを確認するためである。

⑦ コメント比率($⑤ \div ⑥ \times 100$)

理由：OS全体から見た場合にどの程度のコメントを含むか確認するためである。

⑧ アセンブラー行(アセンブラー行+オンラインアセンブラー行)

理由：一般的にアセンブラーはコードリーディングしづらいため、どの程度あるか確認するためである。

⑨ アセンブラー比率($⑧ \div ⑥ \times 100$)

理由：OS全体から見た場合にどの程度のアセンブラーを含むか確認するためである。

⑩ 条件付きコンパイルのマクロ件数

⑪ goto文件数

⑫ 関数マクロ件数

⑬ 可読性を高めるツール類への対応である。doxygen(参考文献[42])、graphviz(参考文献[43])はコードの可視化ができる。

⑩、⑪、⑫の比較項目はMISRA C(参考文献[15])を参考にした。MISRA Cとはコードの可読性、安全性、可搬性を確保する事を目的としたC言語のためのソフトウェア設計標準規格である。MISRA C2004では、121件の必須ルールと20件の推奨ルールが設けられている。

(半分程度のルールは本OSに適用済み)

⑩は必須ルール、⑪及び⑫は推奨ルールから選出した。コードの比較結果を下表11-1に示す。

表11-1 他OSとの可読性比較結果

	⑤コメント行	⑥ソースコード行	⑦コメント比率(%)	⑧アセンブラー行
①本研究OS	6322	22064	29%	140
②kozox(母体OS)	350	2090	16%	112
③H8/OS	304	10726	2.8%	197
④toppers/sspカーネル	2579	20880	12%	625

⑨アセンブラー比率(%)	⑩goto文	⑪条件付きマクロ制限	⑫関数マクロ	⑬doxygen, graphvizへの対応
1%	0件	0件	2件	○
5%	0件	0件	4件	×
1.8%	0件	169件	2件	×
2.9%	10件	100件	32件	×

表11-1より、本研究OSは他OSより2倍以上コメントがあり、アセンブラーが少なく、コードリーディングを低下させる構文がなく可読性が向上したと考えられる。

12. おわりに

12.1 研究の成果

目的で掲げた複数のカーネルソースを可読せずに单一のカーネルコードレベルから機能ごとの複数実装法を可読できるOSを実装することができた。複数方式では、OS機能で代表的なタスクスケジューラ、同期及び排他に使用するセマフォとmutex、時間管理にアラーム機能と周期タイマ機能、優先度逆転機構を実装できた。特に、タスクスケジューラは、汎用系OSに備わるタイプやRTOS系に備わるタイプ、共通して備わるタイプ等複数実装できた。これにより、ユーザは本研究OSからそれらの異なる分野のスケジューラを一度に可読及び操作でき学習可能となった。さらに、ユーザはアプリケーションライブラリより、本研究OSに備わる全ての複数方式のレスポンス結果を体験できる。また、タイマ機構を実装したので、複数方式によるレスポンスタイムを知る事ができる。

新規方式では、スケジューラに1種、優先度逆転防止機構に1種提案でき、本研究OSに実装できた。

しかし、本研究ではレスポンス結果を表示するところまで実装できたが、レスポンスがCUI表示となり、ユーザにとって煩瑣となる。

12.2 今後の課題

本研究では、メモリ管理機構について複数方式を実装できなかった。メモリ管理機構もスケジューラと同様に、汎用系OSで使用されるメモリ管理方式、RTOS系で使用されるメモリ管理方式、共通の分野で使用されるメモリ管理方式がある。今後の課題として、それらを調査し、本研究OSに実装する。

また、10.1で述べたように、ユーザにレスポンス結果をわかりやすいように表示できる機構を本研究OSに組み込む事を今後の課題とする。

13. 参考文献とツール類

13.1. 参考文献

- [1] B.W.カーニハン/D.M.リッチャー著, プログラム言語C 第2版, 共立出版株式会社
- [2] 前橋和弥著, C言語ポインタ完全制覇, 技術評論社
- [3] ピーター.ヴァン.デ.リンデン, エキスパートCプログラミング—知られざるCの深層, アスキー出版局
- [4] 茨木俊秀著, Cによるアルゴリズムとデータ構造, 株式会社昭?堂
- [5] 奥村晴彦著, アルゴリズム辞典, 技術評論社
- [6] Brian W.Kernighan/Rob Pike/福崎 敏博訳, プログラミング作法
- [7] 社団法人トロン著/坂村健(監修), 組込み実践プログラミングガイド ~ITRON仕様OS/T-kernel 対応, 技術評論社
- [8] 鹿取祐二著, SuperHで学ぶ μ ITRON仕様OS, 電波新聞社
- [9] Qing Li/Caroline Yao著, リアルタイム組込みOS, 翔泳社
- [10] Wayne Wolf著, 組込みシステム設計の基礎, 日経BP社
- [11] 藤倉俊幸著, リアルタイム/マルチタスクシステムの徹底研究, CQ出版社
- [12] 薄地 輝尚著, はじめて読む486, アスキー出版
- [13] 日立製作所, h8/3069 f-ztat ハードウェアマニュアル
- [14] 秋月電子, h8/3069f ボードマニュアル
- [15] MISRA C ルール一覧
 - : http://www.openrtp.jp/wiki/_hara/ja/RtORB/MISRA-C-RULE.html
- [16] μ ITRON4.0 仕様書(ver4.02.00, 日本語版)
 - : <http://www.ert1.jp/ITRON/SPEC/mitron4-j.html>
- [17] T-kernel 仕様書(ver1.00.01, 日本語版)
 - : <http://www.t-engine.org/ja/specifications>
- [18] 母体としたオープンソースOS「kozos」
 - : http://kozos.jp/kozos/osbook/osbook_03/12/
- [19] 坂井弘亮著, 組込み自作入門, 株式会社カットシステム
- [20] H8/OS ソースコード : <http://mes.sourceforge.jp/h8/index-j.html>
- [21] TOPPERS/SSP カーネル 1.1.0 : <http://www.toppers.jp/ssp-kernel.html>
- [22] TOPPERS/ASP カーネル 1.3.1 : <http://www.toppers.jp/asp-d-download.html>
- [23] リアルタイムOSの内部構造を見てみよう
 - : <http://www.nces.is.nagoya-u.ac.jp/NEXCESS/blog/index.php?catid=7&blogid=4>
- [24] Linux カーネルソースコード : <http://www.kernel.org/>
- [25] Daniel P.Bovet, Marco Cesati 著/高橋浩和 監訳/杉田由美子/清水正明/高杉昌督/平松雅巳/安井隆宏訳, 読解Linux カーネル第3版
- [26] 平田豊著, Linux カーネル解析入門, 工学社

- [27] 高橋浩和/小田逸郎/山幡為佐久著, Linux カーネル診断室:, ソフトバンククリエイティブ
- [28] Multilevel Feedback Queue Scheduling
: <http://pages.cs.wisc.edu/~remzi/OSFEP/cpu-sched-mlfq.pdf>
- [29] Fair Share Scheduling
: <http://pirlo21.net/article/Fair-Share-Scheduler.html#1>
- [30] developerWorks Linux カーネル 2.6 Completely Fair Scheduling
: <http://www.ibm.com/developerworks/jp/linux/library/l-completely-fair-scheduler/>
: <http://www.ibm.com/developerworks/jp/linux/library/l-cfs/>
- [31] omicron
: <http://tiki.is.os-omicron.org/tiki.cgi>
- [32] priority Inheritance Protocol & Priority Ceiling protocol & Highest Locker's Protocol
:<http://user.it.uu.se/~yi/courses/rts/dvp-rts-07/notes/ResourceAccessProtocols.pdf>
<http://user.it.uu.se/~yi/courses/rts/dvp-rts-08/notes/synchronization-resource-sharing.pdf>
- [33] コード可視化
: http://skazami.web.infoseek.co.jp/tools/Graphviz_Doxygen.htm
- [34] 4.4BSD カーネルの設計と実装
: Marshall Kirk McKusick, Michael J. Karels, John S. Quarterman, 砂原秀樹, 七丈直弘 訳
- [35] BSD カーネルの設計と実装
: Marshall Kirk McKusick, George V. Neville Neil, 歌代和正 訳
[] gcc option(type-punning(型もじり))
: <http://up-cat.net/gcc%2Boption.html>

13.2. ツール類

- [36] h8write(ROM ライタ) : <http://mes.sourceforge.jp/h8/writer-j.html>
- [37] kz_h8write : <http://sourceforge.jp/projects/kz-h8write/>
- [38] minicom : http://sourceforge.jp/projects/freshmeat_minicom/releases/
- [39] kermit : http://sourceforge.jp/projects/sfnet_kermit/releases/
- [40] lrzs : <http://packages.debian.org/ja/sid/lrzsz>
- [41] doxygen : <http://www.stack.nl/~dimitri/doxygen/>
- [42] graphviz : <http://www.graphviz.org/>

- [43] gcc コンパイラ : <http://core.ring.gr.jp/pub/GNU/gcc/>
- [44] binutils : <http://core.ring.gr.jp/pub/GNU/binutils/>
- [45] git : <http://git-scm.com/download>