論文

ITRON 仕様 OS の機能分散マルチプロセッサ拡張

本田　晋也[†a)]　　高田　広章[†]

Extension of ITRON Specification OS for Function-Distributed Multiprocessors

Shinya HONDA[†a)] and Hiroaki TAKADA[†]

あらまし　近年，組込みシステムの分野においてもマルチプロセッサシステムの利用が進んでいる．本論文では，国内の組込みソフトウェア開発で広く使われている ITRON 仕様のリアルタイム OS の機能分散マルチプロセッサ拡張について述べる．本リアルタイム OS を用いると，シングルプロセッサと互換のシステムコールをプロセッサ間で実行することが可能となる．実装にあたっては，プロセッサ数が増加しても，組込みシステムで求められる最悪実行時間や最悪割込み応答時間といったリアルタイム性をできるだけ損なわないよう工夫した．具体的には，プロセッサ間の排他制御のために用いるロックの単位を，要件を満たしつつデッドロック防止の必要性を最小限にするように設定した．また，システムコールの実行時に必要となる，ロックによるプロセッサ間の排他制御とプロセッサ内の排他制御に関して，プロセッサ数に対する最悪割込み応答時間のスケーラビリティを確保するための機構を導入した．そして，実装したリアルタイム OS の基本性能や実装技術の有効性を評価した．

キーワード　組込みシステム，リアルタイム OS，マルチプロセッサ，割込み応答性

1. まえがき

近年，組込みシステムの分野においても，マルチプロセッサシステムの重要性が急速に増している．この背景には，消費電力の増大を抑えつつ，処理性能の向上を図るためには，クロック周波数を上げるよりも，プロセッサ数を増やした方が有利であるという状況がある．特に，複数のプロセッサを一つの LSI 上に集積したオンチップマルチプロセッサ [1] は，処理性能面からも消費電力面からも利点が大きく，広範な組込みシステムへの適用が期待されている．

組込みシステムのソフトウェア開発においては，リアルタイム OS が用いられることが一般的であり，国内では ITRON 仕様 [2] のリアルタイム OS が広く使われている．そのため，マルチプロセッサシステムの利用が広がるにつれて，マルチプロセッサをサポートする ITRON 仕様のリアルタイム OS が必要となる．

マルチプロセッサは，OS でサポートするという観点からは，対称型のマルチプロセッサと機能分散型のマルチプロセッサに分類することができる．対称型マルチプロセッサ（Symmetric Multiprocessor，SMP）とは，各プロセッサから計算機のすべての資源にアクセスすることができ，（機能的には）どの処理をどのプロセッサでも実行できるものをいう．それに対して機能分散マルチプロセッサ（Function Distributed Multiprocessor，FDMP）(注1)とは，各プロセッサからアクセスできる資源に違いがあり，どの処理をどのプロセッサで行うかが，あらかじめ決まっているようなシステムをいう．

組込みシステムは，ある応用に専用化された計算機システムであり，どのような処理を行う必要があるかは，あらかじめ決まっているのが通常である．実行すべきタスクとその実行時間が分かっているならば，各プロセッサの負荷が均衡し，プロセッサ間の通信量が少なくなるよう，タスクをプロセッサに割り付けることができる．更に，タスクをプロセッサに固定すると，OS のリソースをプロセッサごとに独立して管理できるため，プロセッサ間でのリソースの使用に伴う排他制御の必要性が少なくなり，システムのスループッ

† 名古屋大学大学院情報科学研究科情報システム学専攻，名古屋市
Dept. of Information Engineering, Graduate School of Information Science, Nagoya Univ., Nagoya-shi, 464–8601 Japan
a) E-mail: honda@ertl.jp

(注1)：非対称型マルチプロセッサ（Asymmetric Multiprocessor, AMP または ASMP）と呼ばれる場合もある．

 ©（社）電子情報通信学会 2008

トが向上する [3], [4]．以上より，機能分散マルチプロセッサを採用した方が，ハードウェアコストの低減，消費電力の低減，スケーラビリティの確保，リアルタイム性の確保の観点から，有利であるといえる [5]．

既存の ITRON 仕様の OS を用いた機能分散マルチプロセッサシステムでは，ITRON 仕様と互換性のない新規のシステムコールやアプリケーションレベルにより，プロセッサ間の通信を実現している [6]．そのため，複数のタスクで構成されているシングルプロセッサ用のアプリケーションを，変更なしに機能分散マルチプロセッサ上の複数のプロセッサに割り当てて動作させることは不可能である．

そこで我々は，ITRON 仕様を機能分散マルチプロセッサ向けに拡張し（以下，拡張仕様と呼ぶ），TOPPERS/FDMP カーネル（以下，FDMP カーネルと呼ぶ）として実装した．

拡張仕様は，ITRON 仕様で定められているシステムコールの呼出し方は変更せず，引数で指定する操作対象のオブジェクトの ID 番号を拡張することで，他のプロセッサのオブジェクトを指定可能としている．ID 番号はカーネル構築時にカーネルに付随するツールにより自動決定され，アプリケーションは，この自動決定される ID 番号を使うように記述されているため，シングルプロセッサ用のアプリケーションを変更なしに機能分散マルチプロセッサ上で動作させることが可能である．FDMP カーネルの実装にあたっては，プロセッサ数が増加しても，組込みシステムで求められるリアルタイム性をできるだけ損なわないよう工夫した．具体的には，プロセッサ間の排他制御のために用いるロックの単位を，要件を満たし，デッドロック防止の必要性を最小限にするように，ITRON 仕様のシステムコールを分析して決定した．また，システムコールの実行時に必要となるロックによるプロセッサ間の排他制御とプロセッサ内の排他制御に関して，最悪割込み応答時間のプロセッサ数に対するスケーラビリティを確保するための機構を導入した．

本論文では，まず，想定する機能分散マルチプロセッサのアーキテクチャと，そのための OS の要件について述べる．次に，拡張仕様を解説したのち，FDMP カーネルの実装に用いた技術について述べる．最後に FDMP カーネルの基本性能や，実装に用いた技術の評価を行う．

2. 機能分散マルチプロセッサ向けリアルタイム OS

ITRON 仕様を機能分散マルチプロセッサに拡張するにあたって，想定したシステムと，定めた要件について述べる．

2.1 想定システム

本研究の対象とする機能分散マルチプロセッサは，ITRON 仕様がターゲットとしている小規模でリアルタイム性の高い組込みシステムとする．このようなシステムでは，メモリサイズは小さく，リアルタイム性のため MMU は使わないことが多い．

想定するアーキテクチャを図 1 に示す．各プロセッサはローカルバスに接続されたローカルメモリと I/O 装置をもつ．ローカルメモリは，グローバルバスを介して他のプロセッサからアクセス可能である．

タスクや割込みハンドラは，特定のプロセッサで固定的に実行する．このような構成することにより，プロセッサ間の通信量が少なくなるようタスクをプロセッサに割り付けると，他のプロセッサの実行を阻害しなくなり，シングルプロセッサに近いリアルタイム性が確保でき，プロセッサ数に対するスケーラビリティを確保しやすい．

2.2 機能分散マルチプロセッサ向けリアルタイム OS の要件

機能分散マルチプロセッサ向けリアルタイム OS の要件を，前述の想定システムの利点を活用し，組込みリアルタイムシステムに求められるリアルタイム性を満たすよう，次のように定めた．

（1） プロセッサ間の同期・通信のためのシステムコールは，シングルプロセッサ用のシステムコールと互換性をもつこと．

（2） 各プロセッサがプロセッサ内に閉じた処理（システムコールも含む）を実行している限りは，互いの処理を阻害せず，プロセッサ数に関係なく最悪実行時

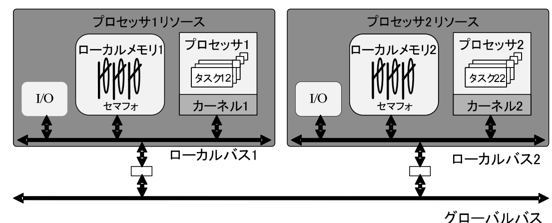

図 1　想定アーキテクチャ
Fig. 1　Assumed architecture.

間が定まること．

（3） 各プロセッサがプロセッサ内に閉じた処理（システムコールも含む）を実行している限りは，互いの処理を阻害せず，プロセッサ数に関係なく最悪割込み応答時間が定まること．

（4） プロセッサをまたぐ処理の最悪実行時間が定まること．

（5） プロセッサをまたぐ処理を実行した場合の最悪割込み応答時間が定まること．

（1）は，互換性のないシステムコールでは，ある処理を別のプロセッサに移そうとすると（設計時の移し換えのみを考えており，動的な移動は考えていない），同期・通信部分のプログラムの作り直しが必要になる．シングルプロセッサ用のシステムコールと互換性をもつことにより，どのプロセッサで実行されているかによらず，いずれのタスクとも同じ方法で同期・通信ができるようになり，機能分散マルチプロセッサ上でのソフトウェア開発を効率化することができるため定めた．

（2），（3）を満たすと，各プロセッサがプロセッサ内に閉じた処理をする限りは，プロセッサ数を増加させても，個々のプロセッサのリアルタイム性は低下しない，すなわち，プロセッサ数に対するスケーラビリティの確保のため定めた．

（4），（5）は，リアルタイムシステム性の要件として定めた．なお，最悪実行時間がプロセッサ数と比例して長くなることは本質的に避けられない．

3. 拡張仕様

本章では，ITRON 仕様の機能分散マルチプロセッサ拡張について述べる．まず，ベースとした μITRON4.0 仕様のスタンダードプロファイルについて，概要及び，拡張仕様のシステムコールの互換性の説明で必要となるカーネルオブジェクトの ID 番号の扱いについて説明する．その後，拡張仕様の詳細について説明する．

拡張仕様では，μITRON4.0 仕様から以下の点を機能分散マルチプロセッサ向けに拡張した．

- カーネルオブジェクトのクラス分け
- ID 番号の割付け方法
- システムコール
- システム状態
- 静的 API

```
静的 API の記述例（タスク生成）
CRE_TSK(TASK1, task_1, .....);

ID 番号自動割付け結果ヘッダファイルの例
#define TASK1 1

システムコールの記述例（タスク起動）
act_tsk(TASK1);
```

図 2　オブジェクト ID 割付け例
Fig. 2　Example of object ID assignment.

3.1 μITRON4.0 仕様の概要

μITRON4.0 仕様のスタンダードプロファイルは μITRON4.0 仕様で定められている標準的な機能セットである．

スタンダードプロファイルでは，システム全体が一つのモジュールにリンクされ，カーネルオブジェクトはすべて静的に生成される．システム全体が一つのモジュールにリンクされることから，システムコールはサブルーチンコールで呼び出すことになる．また，プロテクションの機能はもたない．

カーネルオブジェクトの静的な生成情報（名前や属性）は，静的 API と呼ばれる記法によりコンフィギュレーションファイルに記述する．コンフィギュレーションファイルは，カーネル構築（コンパイル）時にコンフィギュレータと呼ばれるプログラムにより処理されカーネル構成・初期化ファイル（C 言語のコード）が生成される．同時に，コンフィギュレータは，カーネルオブジェクトに ID 番号を割り付け，カーネルオブジェクト名のマクロとして定義したものを，「ID 番号自動割付け結果ヘッダファイル」として生成する．

カーネルオブジェクトを操作するシステムコールは引数に操作対象のカーネルオブジェクトの ID 番号を指定する．前述のように ID 番号はカーネル構築時まで決定されないため，プログラム中では，ID 番号の代わりにカーネルオブジェクトの名を指定して記述する（図 2）．そして，「ID 番号自動割付け結果ヘッダファイル」をインクルードすることにより，コンパイル時に実際の ID 番号を指定することが可能となる．

3.2 カーネルオブジェクトのクラス分け

タスクやセマフォなどのカーネルオブジェクトは，システム中のいずれかのプロセッサに属し，あるプロセッサに属するオブジェクトの集合をクラスと呼ぶ．カーネルオブジェクトをどのプロセッサに所属させるかは，後で述べるコンフィギュレーションファイルに

よって静的に決定する．クラスは 1 から連続する ID 番号で識別する．

以下に示す処理単位となるカーネルオブジェクトはそれが属するプロセッサでのみ実行される．そのため，OS はプロセッサごとに独立にタスクスケジューリングを行う．

- タスク/タスク例外処理ルーチン
- 周期/割込み/CPU 例外ハンドラ

3.3 ID 番号

カーネルオブジェクトは，ID 番号を拡張することにより所属するクラスを表す．具体的には，ID 番号の上位ビットで所属するクラスを下位ビットでそのクラス内でのオブジェクトの識別番号を表す．

3.4 システムコール

要件（1）を満たすため，ITRON 仕様と互換のシステムコールにより，システムに存在するすべてのカーネルオブジェクト（属するプロセッサが異なっても）に対してアクセス可能である．

カーネルオブジェクトを操作するシステムコールの引数には，操作対象オブジェクトの ID 番号を指定する．拡張仕様ではオブジェクト ID の上位ビットにクラス ID を含むため，引数に直接 ID 番号を記述しているプログラムは，そのままでは機能分散マルチプロセッサ上で動作させることはできない．しかしながら，引数には **3.1** で述べたように直接 ID 番号を記述せず，カーネルオブジェクト名を記述する．拡張仕様用のコンフィギュレータはクラス ID を付加した ID 番号の定義マクロを「ID 番号自動割付け結果ヘッダファイル」として生成するため，シングルプロセッサ用のプログラムは書換えなしに，機能分散マルチプロセッサに対応させることができる．

3.5 システム状態

ITRON 仕様では，特定のタスクを独占的に実行するための状態として，割込み及びタスク切換を禁止する CPU ロック状態や，タスク切換を禁止するディスパッチ禁止状態をもつ．これらの状態は排他制御の目的で使われる場合が多い．拡張仕様では，これらの状態をプロセッサごとに独立に管理する．すなわち，あるプロセッサで CPU ロック状態やディスパッチ禁止状態になったとしても，他のプロセッサにおいてはそれらは禁止されない．

そのため，CPU ロック状態とディスパッチ禁止状態を用いて他のプロセッサの処理単位との排他制御は実現することができない．シングルプロセッサ用のアプリケーションを複数のプロセッサに分割して動作させる場合には，CPU ロック状態とディスパッチ禁止状態により排他制御を行っている部分を同期オブジェクトを用いた方法に書き換える必要がある．

一方，互換性のため，シングルプロセッサと同様にシステム全体で一つの状態を管理することも可能であるが，要件（2),（3）を満たせなくなることや，実現オーバヘッドが大きいため，プロセッサごとに独立に管理することとした．

```
local_class PE1{
   CRE_TSK(TASK1, {TA_HLNG, ..});
   CRE_TSK(TASK2, {TA_HLNG, ..});
   CRE_CYC(CYCHDR1, {TA_HLNG, ..});
}
local_class PE2{
   CRE_TSK(TASK3, {TA_HLNG, ..});
   CRE_TSK(TASK4, {TA_HLNG, ..});
   CRE_CYC(CYCHDR2, {TA_HLNG, ..});
}
```

図 3 静的 API の例
Fig. 3 Example of static API.

3.6 静的 API

すべてのカーネルオブジェクトはいずれかのクラスに属するため，カーネルオブジェクトの登録を行う静的 API は，クラス（プロセッサ）ごとの囲みの中に記述するよう拡張した．これにより複数のプロセッサの静的 API 記述を一つのファイルに記述可能である．

PE1 と PE2 の二つのプロセッサで構成されているシステムのコンフィギュレーションファイルの例を図 3 に示す．それぞれのプロセッサで二つのタスクと一つの周期ハンドラを生成している．このように複数のプロセッサの生成情報を一つのファイルに記述することにより，プロセッサ間でのカーネルオブジェクトの（静的な）移動が容易になる．例えば TASK3 を PE2 から PE1 に移動させるには，CRE_TSK(TASK3.. の行を PE1 のクラスの囲みの中に移動するだけでよい．

4. FDMP カーネルの実装技術

本章では，拡張仕様を実装した FDMP カーネルの実装技術について述べる．

4.1 プロセッサごとに独立した管理ブロック

機能分散マルチプロセッサでは，タスクや割込みハンドラといったカーネルオブジェクトは，特定のプロセッサに固定化されているため，それらのコードや

データは，プロセッサごとに独立にもつことが可能である．

そのため，カーネルオブジェクトを管理するデータ構造（以下，管理ブロックと呼ぶ）及び，カーネルのコードとデータをプロセッサごとにもつこととした．すなわち，プロセッサごとに独立したオブジェクトコードで動作するよう実装した．

プロセッサごとのオブジェクトコードをローカルメモリにおくことにより，プロセッサ内に閉じた処理を実行する限りは，他のプロセッサのメモリアクセスを阻害せず，要件（2）を満たすことができる．

プロセッサをまたぐシステムコールを実現する場合は，他のプロセッサの管理ブロックを操作する必要がある．FDMP カーネルでは，他のプロセッサの管理ブロックを直接操作する直接操作法で実現した [5]．

直接操作法を実現するためには，他のプロセッサの管理ブロックの配置場所を知る必要がある．そのため，各プロセッサごとの管理ブロックへのポインタをクラスコントロールブロック（CCB）と呼ぶ構造体に登録して，カーネル内で用いる．

4.2 ロック

FDMP カーネルでは，システムコールを直接操作法を用いて他のプロセッサの管理ブロックを操作することで実現している．プロセッサ間で共有する管理ブロックの操作にあたっては，ロックによるプロセッサ間での排他制御が必要となる．

ロックはスピンロックで実現した．ロックが必要となる ITRON 仕様の全システムコールは，排他が必要なデータ構造に対し書込みを伴う場合がある．そのため，ロックの実現方法として，Linux で採用されている reader-writer spinlock，Seqlock，Read-Copy Update（RCU）等 [7] の one-writer/many-readers タイプの排他制御を用いることはできない．

4.3 ロック単位

一つのロックで排他制御を行うリソースの単位をロック単位と呼ぶ．要件（2），(4）を満たすためには，ロック単位を適切に設定する必要がある．

最も大きなロック単位はシステム全体で一つのロックを用いる方法（ジャイアントロック）であるが，この方法では，各プロセッサで閉じた処理を実行した場合でも，ロックの競合が発生し，互いの処理を阻害するため，要件（2）を満たさない．

しかしながら，あまり細かい粒度でロック単位を設定してしまうと，ロック取得のネスト段数が多くなりオーバヘッド（最悪時性能）が増大してしまう．

また，デッドロック防止が必要なシステムコールを最小限にするため，できるだけ多くのシステムコールでロックの取得順序が一定になるようにロック単位を決定する必要がある．定めた順序でロックを取得できないシステムコールに関しては，デッドロック防止のためのコードを加える必要がある．

4.3.1 プロセッサごとのロック単位

ジャイアントロックの次に粒度が細かいロック単位として，プロセッサごとに別々のロック単位を設定する方法がある．FDMP カーネルは，管理ブロックをプロセッサごとにもつため，プロセッサごとに別々のロック単位を設定すると，各プロセッサが，プロセッサに閉じた処理を実行している限りは，ロックの競合は発生せず，要件（2）を満たすことが可能である．

しかしながら，単にプロセッサごとに一つのロックを設ける方法では，システムコールをプロセッサをまたいで実行した場合に，FDMP カーネルがサポートする 74 個のシステムコールうち 24 個のシステムコールでデッドロックが発生する可能性がある．具体的には，タスク管理にかかわる管理ブロックと，同期・通信オブジェクトにかかわる管理ブロックの両方を操作する可能性があるシステムコールでデッドロックが発生する．これらのシステムコールはセマフォやイベントフラグ等を操作するものであり，一般に使用頻度が高い．

デッドロックの発生をセマフォを例に説明する．プロセッサ 1 のタスク 1 が取得待ちとなっているプロセッサ 2 のセマフォ2 に対してセマフォを返却するシステムコール（sig_sem）を発行すると，まずセマフォ2 の管理ブロックを操作するため，プロセッサ 2 のロックを取得する（1a）．セマフォ2 の管理ブロックのアクセスにより，タスク 1 を起床させる必要があることが分かる．次にタスク 1 の管理ブロックを操作するため，プロセッサ 1 のロック取得する（1b）．すなわち，プロセッサ 2 のロック（1a）→ プロセッサ 1 のロック（1b）の順にロックを取得する．一方，プロセッサ 2 のタスク 2 が取得待ちとなっているプロセッサ 1 のセマフォ1 に対して sig_sem を発行すると，先ほどと逆順（プロセッサ 1 のロック（2a）→ プロセッサ 2 のロック（2c））でロックを取得する．そして，それぞれのシステムコールが異なるプロセッサ上のタスクにより実行され，片方のタスクが（1a）を実行した後，(1b）を実行するまでの間に，もう片方のタスクが（2a）を実行すると，デッドロックが発生する．

4.3.2 FDMP カーネルの分析とロック単位の決定

最適なロック単位を決定するため，プロセッサ間で共有する管理ブロックの種類と，それに対するアクセスパターンを整理した結果，プロセッサごとに二つのロックを設けると，デッドロック防止が必要な処理をシステムコール 3 個（rel_wai，chg_pri，ter_tsk）と，タイムアウト処理に限定できることが分かった [8]．具体的には，プロセッサごとにタスクロックとオブジェクトロックの二つのロックを設け，システムコール内でのロックの取得順序をオブジェクトロック → タスクロックの順に定めた．

タスクロックは，タスク管理にかかわるデータ構造用のロックで，タスク管理ブロックの操作の前に取得する．また，時間管理関係の内部データ（タイムイベントヒープ）もタスクロックにより排他制御する．オブジェクトロックは，同期・通信オブジェクトにかかわるデータ構造用のロックで，セマフォやイベントフラグ，データキューの管理ブロックの操作の前に取得する．

プロセッサごとにタスクロックとオブジェクトロックをもつことにより，前述のセマフォの返却の例では，プロセッサ 2 のセマフォ2 に対してセマフォを返却するシステムコールは，プロセッサ 2 のオブジェクトロックを取得後，プロセッサ 1 のタスクロックを取得し，プロセッサ 1 のセマフォ1 に対してセマフォを返却するシステムコールは，プロセッサ 1 のオブジェクトロックを取得後，プロセッサ 2 のタスクロックを取得するため，デッドロックは発生しない．

4.4 デッドロック防止

オブジェクトロック → タスクロックの順でロックを取得できないシステムコールに対しは，デッドロック防止が必要となる．

これらのシステムコールは，まずタスクロックをロックしてタスク管理ブロック（TCB）にアクセスした後で初めてロックするべきオブジェクトロックが分かる．そのため，一度タスクロックを取得したのち，ロックするべきオブジェクトロックが分かった時点でいったんタスクロックを解放して，改めてオブジェクトロック → タスクロックの順序でロックを取得するようにする．

この方法では，いったんタスクロックを解放して，オブジェクトロック → タスクロックの順でロックを再取得する間に，タスクの状態が変化する可能性がある．例えば，タスクが別のオブジェクトに対する待ち状態に変化した場合には，ロックするべきオブジェクトロックが変化するため，両方のロックを解放して，新しいオブジェクトロック → タスクロックの順でロックを再取得する必要がある．この方法では，再取得回数の上限が抑えられず，要件（2),（4）を満たせない．

この問題を解決するため，デッドロック防止が必要なシステムコールごとに，そのシステムコールを実行中であることを示すフラグ（保留フラグ）を TCB 中に用意する．そして，デッドロック防止のためにいったんロックを解放する前に，このフラグをセットし，システムコールが保留中であることを示す．

ロック解放後に他のプロセッサがロックを取得して，タスクの状態を変更する場合は，TCB 中のフラグをチェックし，保留されているシステムコールがあれば，代わりに対象のタスクに対する処理を行う．そして，TCB 内の保留フラグをクリアする．一方，保留中のシステムコールはロック取得後，保留フラグをチェックし処理の必要性の有無をチェックして，保留フラグがクリアされていなければタスクの状態を変更し，クリアされていれば何も行わずシステムコールの実行を終了する．

4.5 プロセッサ間とプロセッサ内の排他制御の問題

システムコールの実行（管理ブロックへのアクセス）には，プロセッサ間での排他制御と，プロセッサ内での排他制御が必要となる．FDMP カーネルでは，前者は前述のロック（スピンロック）で後者は割込み禁止で実現している．

この二つの排他制御は互いに関連しており，リアルタイム性を確保するためには適切に扱わなければならない．例えば，ロックの取得と割込みの禁止という順に実行すると，ロックの取得と割込みの禁止の間に割込みが入りそれを受け付ける可能性がある．割込みを受け付けると，ロックを取得しているため，割込みを受け付けている間は他のプロセッサを待たせてしまい要件（4）を満たせない．一方，割込みを禁止してロックの取得を試みると，ロックの取得を試みる間，割込みが禁止となって割込み禁止時間が長くなってしまい，要件（5）を満たせない．このようにどちらを先に実行しても問題が発生してしまう．理想的な実現方法は二つを同時ロックすることであるが，1 命令でロックの取得と割込みの禁止を行えるプロセッサは存在しないため，実現は困難である．

4.5.1 Test&Set ロックによる実装

プロセッサ間とプロセッサ内の排他制御の問題に関

しては，FDMP カーネルの現在の実装では理想的な解法に至っておらず今後の課題としている．

FDMP カーネルでは，まず割込みを禁止してから，Test&Set ロックによりロックの取得を試みる．そして，ロックの取得を試みるたびに（ロックの取得を試みた結果ロックが取得できなかった場合に），割込み要求をチェックし，割込み要求があれば，割込みを許可して割込みを受け付ける．割込みを先に禁止することにより，ロックを取得したまま割込みハンドラを実行することを防ぎ，ロックの取得を試みるたびに割込み要求をチェックすることにより要件（5）を満たす．

オブジェクトロックとタスクロックの二つのロックを取得する必要がある場合のコードを図 4 に示す．2 段目のロックの取得である（3）のタスクロックの取得ができなかった場合は，割込み要求をチェックして，割込み要求があれば，取得済みのオブジェクトロックを解放し，割込みを許可して割込みを受け付ける．その後，（1）のオブジェクトロックの取得からやり直す．

```
retry:
  /* (1) オブジェクトロックの取得 */
  disable_interrupt();
  while(!test_and_set(obj_lock)){
    /* ロックが取得できずかつ割込みが発生して */
    /* いれば，割込みを受け付け，その後 retry */
    /* から再開する */
    if(interrupt_request()){
      enable_interrupt();
      goto retry;
    }
  }

  /* (2) この箇所のコードは繰り返し実行される可能 */
  /* 性があるため，カーネル内部のデータを変更して */
  /* はならない */
  ...

  /* (3) タスクロックの取得 */
  while(!test_and_set(tsk_lock)){
    if(interrupt_request()){
      /* ロックが取得できずかつ割込みが発生して */
      /* いればオブジェクトロックを解放し，割込 */
      /* みを受け付け，その後 retry から再開する */
      release_lock(obj_lock);
      enable_interrupt();
      goto retry;
    }
  }

  /* (4)Critical Section */
  ...
```

図 4　プロセッサ内排他制御とプロセッサ間排他制御
Fig. 4　Inprocess exclusion and interprocess exclusion.

このように，タスクロックの取得ができない場合，図中の（2）のオブジェクトロックの取得からタスクロックの取得の間のコードは，繰り返し実行される可能性がある．そのための，この箇所のコードでは，タスクやオブジェクトの管理ブロックといったカーネル内のデータを変更してはならない．

Test&Set ロックは，ロック取得までの上限時間が定まらないため，要件（4）を満たせず，厳密な時間制約が求められるリアルタイムシステムには向かない．上限時間が定まるロック手法として，キューイングスピンロック [9] がある．ロックが一つの場合は，キューイングスピンロックを拡張して要件（4）を満たすことが可能であるが，二つのロックを取得する場合は要件（4），（5）を満たすことができない [10]．

5. 評　　価

本章では，FDMP カーネルの評価を行う．まず，FDMP カーネルのベースとなった μITRON4.0 仕様のスタンダードプロファイルに準拠した JSP カーネルと，コードサイズと実行時間について，デッドロック防止が必要なシステムコールに特に着目して比較する．次に，プロセッサ間とプロセッサ内の排他制御の問題に関して，最悪割込み応答時間を計測し，提案手法の有効性を評価する．また，プロセッサごとに独立のメモリをもつ場合と単一の共有メモリをもつ場合との性能差を評価する．

5.1 評価環境

評価環境としては，プロセッサに Altera 社の NiosII を用いた．NiosII は，FPGA 用のソフトコアであり，任意の数のプロセッサをもつシステムを FPGA 上に構築可能である．評価では，4 プロセッサのシステムを用いた．各プロセッサはそれぞれ 4 kByte の命令キャッシュをもつ．メモリは FPGA 内蔵メモリを，それぞれ個別のバスインタフェースをもつ 4 個のメモリに分け，各プロセッサのオブジェクトコード（4 個）はこの 4 個メモリにそれぞれ配置した．NiosII のバスである Avalon は，クロスバースイッチであり，今回のシステムでは，各プロセッサと各メモリは完全結合している．そのため，各プロセッサが自分のオブジェクトコードが配置されたメモリをアクセスしている限りはバスの衝突は発生しない．また，他のプロセッサとの衝突がなければ，自プロセッサのオブジェクトが配置されたメモリに対するアクセス時間と他プロセッサのオブジェクトが配置されたメモリに対するアクセス時

表 1　JSP カーネルと FDMP カーネルのコードサイズ
Table 1　Code size of JSP kernel and FDMP kernel.

カーネル	text	data	bss
JSP	26671 Byte	5 Byte	68 Byte
FDMP	42707 Byte	6 Byte	76 Byte

表 2　一般のシステムコールの text サイズ
Table 2　Text size of normal systemcall.

システムコール	JSP	FDMP	ロック取得数	増加率
act_tsk	200 Byte	392 Byte	1	1.96
slp_tsk	184 Byte	280 Byte	1	1.52
sig_sem	348 Byte	568 Byte	2	1.63
wai_sem	168 Byte	408 Byte	2	2.43

間は同等である．プロセッサとメモリともに動作周波数は 50 MHz である．

FDMP カーネルでは，ロックはスピンロックで実現しているが，NiosII はメモリに対する Test&Set 命令をサポートしておらず，Test&Set 命令に相当する機能を実現するバス接続のペリフェラル（ロック回路）が提供されている．プロセッサは，ロックを取得する場合，このロック回路に対して，スピンを行う．評価環境では，ロックごとにこのロック回路を用意した．前述のとおり，プロセッサ間との接続はメッシュ構造の完全結合であるため，あるプロセッサがスピン中でも，他のプロセッサが同じロックを使わなければ実行時間に影響を及ぼさない．

5. 2　コードサイズ

JSP カーネルと FDMP カーネルのコードサイズの比較を表 1 に示す．text のサイズは，JSP カーネルの 1.6 倍となっている．これは，各システムコールにロックの取得と解放のルーチンが挿入されたためと，デッドロック防止ルーチンのためである．一方，data/bss はほとんど増加していない．また，CCB に関しては，プロセッサごとに 124 Byte 必要となる．TCB はタスクごとに 6 Byte 増加している．

デッドロック防止ルーチンを必要としない一般的なシステムコールをマルチプロセッサ対応にすることによってどの程度コード量が増加するかその傾向を評価するため，デッドロック防止ルーチンが必要ないいくつかのシステムコールの text サイズを JSP カーネルと FDMP カーネルで比較した．表 2 の比較結果を見ると，約 1.5 倍から 2.4 倍の増加率となっており，ロック取得数と増加率には相関はないことが分かる．

デッドロック防止ルーチンを組み込んだシステムコールのサイズの比較を表 3 に示す．表 2 の一般のシステムコールと比較して増加率が高い．

表 3　デッドロック防止が必要なシステムコールの text サイズ
Table 3　Text size of deadlock prevented systemcall.

システムコール	JSP	FDMP	増加率
ter_tsk	296 Byte	836 Byte	2.82
chg_pri	264 Byte	648 Byte	2.45
rel_wai	192 Byte	544 Byte	2.83

5. 3　実 行 時 間

JSP カーネルと FDMP カーネルにより，同一プロセッサのオブジェクトに対してシステムコールを発行した場合（プロセッサ内）と，FDMP カーネルにより，他プロセッサのオブジェクトに対してシステムコールを発行した場合（プロセッサ間）について比較する．

測定は，まず起床待ち状態のタスクに対して，そのタスクを起床させるシステムコール（wup_tsk）を発行し，その実行時間を測定した．wup_tsk は，内部でタスクロックを取得する．測定は 2 種類の条件で行った．一つはディスパッチが発生しない条件とし，wup_tsk の実行に要する時間を測定した．もう一つは，ディスパッチが発生する条件とし，wup_tsk の実行が開始され，起床されたタスクの実行が再開されるまでの時間を計測した．また，タスクロックとオブジェクトロックの両方を取得する sig_sem に関しても同様の条件で計測した．それぞれの測定結果を表 4 と表 5 に示す．なお，プロセッサ間では，システムコール発行対象のタスクが所属するプロセッサでは特にタスクを実行せず，割込み待ちになっている条件で測定した．割込み待ちでは，データはアクセスせず，命令は命令キャッシュがあるため，メモリへのアクセスは発生しない．

JSP カーネルと比較すると，プロセッサ内の実行であっても実行時間は増加している．これはロックの取得のためのオーバヘッドと，CCB によるアクセスのためである．また，ロックの取得数の多い sig_sem の方が実行時間の増加率が高くなっている．ディスパッチなしの条件で，プロセッサ間とプロセッサ内の実行時間は同じである．これは今回の評価環境では，自プロセッサのメインメモリと他プロセッサのメインメモリのアクセス時間が同じであり，かつシステムコール発行対象のタスクが所属するプロセッサでは，前述のようにメインメモリに対するアクセスが発生していないため，システムコールを発行したプロセッサとのメモリアクセスの衝突が発生しないためである．

ディスパッチありの条件で，プロセッサ内と比較して，プロセッサ間の実行時間が大きく増加しているの

表 4 システムコールの実行時間（ディスパッチなし）
Table 4 Execution time of systemcall without dispatch.

条件	wup_tsk	sig_sem
JSP	5 μs	5 μs
FDMP（プロセッサ内）	9 μs	10 μs
FDMP（プロセッサ間）	9 μs	10 μs

表 5 システムコールの実行時間（ディスパッチあり）
Table 5 Execution time of systemcall with dispatch.

条件	wup_tsk	sig_sem
JSP	7 μs	6 μs
FDMP（プロセッサ内）	11 μs	13 μs
FDMP（プロセッサ間）	17 μs	18 μs

表 6 デッドロック防止が必要なシステムコールの実行時間
Table 6 Execution time of deadlock avoided systemcall.

条件	ter_tsk	rel_wai	chg_pri
JSP	4 μs	5 μs	3 μs
FDMP（プロセッサ内）	11 μs	14 μs	10 μs
FDMP（プロセッサ間）	11 μs	15 μs	10 μs

は，プロセッサ間ディスパッチ要求は，割込みで伝えられ，実行された割込みハンドラの出口でタスク切換を行うため，単にレジスタ切換を行うだけのプロセッサ内よりオーバヘッドが大きいためである．

次にデッドロック防止ルーチンを組み込んだシステムコールの実行時間を表 6 に示す．デッドロック防止ルーチンを組み込んでいない表 4 のシステムコールと比較すると増加量が大きい．ほぼ同じ処理を行う sig_sem（ディスパッチなし）と rel_wai を比較すると，4〜5 μs の差があり，これはデッドロック防止のためのオーバヘッドである．

5.4 プロセッサ間とプロセッサ内の排他制御

プロセッサ内排他制御とプロセッサ間排他制御の問題に関して，単に割込みを禁止してからロックの取得を試みる方式（方式 1）と，FDMP カーネルで採用している方法（方式 2）に関して，プロセッサ数を増加させた場合の最悪割込み応答時間を比較する．

測定は，各プロセッサで同じロックを取得するシステムコールを実行するタスクを動作させ，1 ms ごとに発生するシステムタイマの割込みが発生してから，その割込みハンドラが実行されるまでの時間（割込み応答時間）の最悪値を測定した．測定値は割込みを 1 万回実行した結果である．測定結果を図 5 に示す．

なお，システムタイマの割込みが発生してから，その割込みハンドラが実行されるまでの時間は，割込み

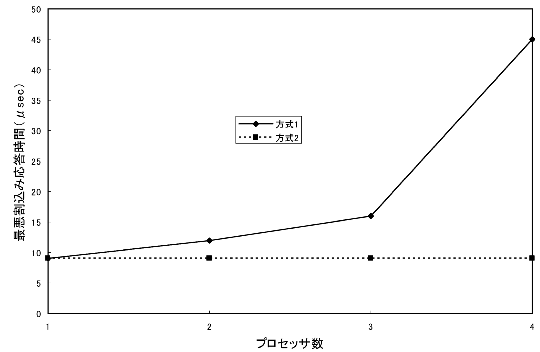

図 5 最悪割込み応答時間
Fig. 5 Worst case interrupt response.

ハンドラの入り口でシステムタイマのカウント値を読み込むことで取得している．システムタイマはオートリロードタイプで，タイムアウトにより割込みを発生させた後，即座に 0 からカウントを再開するため，システムタイマにより起動される割込みハンドラの先頭でタイマのカウント値を読み込むと，その値はタイマがタイムアウトして割込み要求をプロセッサに送ってから，割込みハンドラが実行されるまでの時間となる．

方式 1 では，プロセッサ数の増加に伴って，最悪割込み応答時間が長くなっている．これは，プロセッサ数が増加することにより，ロック取得までの時間が長くなるのに対して，その間を割込み禁止としているためである．一方，方式 2 では，プロセッサ数が増加しても，最悪割込み応答時間は増加しておらず要件（5）を満たしている．

5.5 メモリ構成による性能差

FDMP カーネルは，プロセッサごとに独立したオブジェクトコードをもつ．そのため，プロセッサごとに独立したメモリをもつハードウェアアーキテクチャを用い，プロセッサごとのオブジェクトコードをそれぞれのメモリに配置すると，プロセッサ内に閉じた処理を実行している限りは，他のプロセッサの実行を阻害しないというメリットがある．一方，単一の共有メモリのみをもつハードウェアアーキテクチャの場合は，すべてのプロセッサのオブジェクトコードをこの単一の共有メモリに配置するため，メモリアクセスの衝突が発生する．

衝突によりどの程度の性能が低下するか評価するため，プロセッサに閉じたシステムコールの実行時間を，プロセッサごとに独立したメモリをもち，それぞれのメモリにプロセッサごとのオブジェクトコードを配置した条件（独立メモリ）と，単一の共有メモリにすべ

表 7　独立メモリでの wup_tsk の実行時間
Table 7　Execution time of wup_tsk in individual memory architecture.

	プロセッサ			
プロセッサ数	CPU1	CPU2	CPU3	CPU4
1	11 μs	-	-	-
2	11 μs	13 μs	-	-
3	11 μs	12 μs	13 μs	-
4	11 μs	12 μs	12 μs	13 μs

表 8　単一共有メモリでの wup_tsk の実行時間
Table 8　Execution time of wup_tsk in single shared memory architecture.

	プロセッサ			
プロセッサ数	CPU1	CPU2	CPU3	CPU4
1	11 μs	-	-	-
2	12 μs	14 μs	-	-
3	12 μs	15 μs	16 μs	-
4	13 μs	16 μs	19 μs	19 μs

てのプロセッサのオブジェクトコードを配置した条件（単一共有メモリ）で測定した．

測定結果を表 7 と表 8 に示す．測定したシステムコールは wup_tsk で，ディスパッチを発生する条件とした．このシステムコールをすべてのプロセッサで実行し，プロセッサ数を 1 個から 4 個に増加させ実行時間を測定した．

独立メモリでは，プロセッサが増加しても各プロセッサでの実行時間は大きくは増加せず，プロセッサ数とは独立して実行時間が定まっていることが分かる．一方，単一共有メモリでは，プロセッサ数の増加とともに実行時間が増加している．なお，表 8 に示した時間はすべてのプロセッサで wup_tsk を実行した場合の結果であり，あるプロセッサでの実行時間は，他のプロセッサで実行されている処理の内容によって増減する．

6. 関連研究

文献 [11] では，本研究と同様に ITRON 仕様の OS をマルチプロセッサ向けに拡張している．拡張仕様は，メモリ空間をプロセッサ間で共有しない分散型マルチプロセッサとメモリ空間を共有する共有メモリの両方に対応し，ITRON 仕様で定められたシステムコールと互換のシステムコールをもつ．FDMP カーネルと比較すると，一部の ITRON 仕様のシステムコールしかサポートしてない．また，ロック単位とその実現方法については述べられていない．

マルチプロセッサ向けのリアルタイム OS である Atlanta [12] では，システムコールの実行時間は十分に短いとして，システム全体で一つのロックを用いている．そのため，FDMP カーネルと比較してプロセッサ数に対するスケーラビリティが確保できない．

文献 [13] で述べられているリアルタイム OS では，FDMP カーネルと同様に，プロセッサごとに独立したコードとデータをもつことにより，プロセッサ数に対するスケーラビリティを確保している．しかしながら，ロック単位とその実現方法については言及されていない．

各種のハードウェアを追加することにより，マルチプロセッサ上でのリアルタイム OS の高速実行や効率の良いロックを実現する研究が多くなされている．文献 [13], [14] では，システムコールを実行するための専用のハードウェアについて述べている．専用のハードウェアはシステムに 1 個のみ存在し，すべてのプロセッサからアクセスされるため，ジャイアントロックと同様にシステムコールの実行はプロセッサ間で排他的になってしまう．文献 [15], [16] では，ロックのためのハードウェアを提案している．これら専用ハードウェアを用いる手法は，有用だが特定のハードウェアアーキテクチャに依存するため汎用性がない．FDMP カーネルは，特殊なハードウェアを前提としないため，汎用性が高く，現時点で 4 種類のプロセッサ（Nios2, Microblaze，ARM MPCore，MeP）をサポートしている．

また，前述の研究では，プロセッサ間とプロセッサ内の排他制御の問題については，言及されていない．

7. む す び

本論文では，機能分散マルチプロセッサ向けリアルタイム OS に求められる要件を挙げ，ITRON 仕様を機能分散マルチプロセッサ拡張し，拡張仕様に準拠した FDMP カーネルの実装に用いた技術について述べ，基本性能や実装技術の有効性を評価した．

拡張仕様では，ITRON 仕様と互換のシステムコールを用いて，プロセッサ間の同期・通信が可能である．実装技術に関しては，他プロセッサのオブジェクトの操作に直接操作法を用いるために必要となる，ロック単位の設定や，デッドロック防止，プロセッサ間とプロセッサ内の排他制御に関して述べ，それらのオーバヘッドや効果について評価した．

その結果，機能分散マルチプロセッサ向けリアルタイム OS に求められる五つの要件のうち四つを満たしていることを確認した．具体的には，シングルプロ

セッサ用のシステムコールと互換システムコールでプロセッサ間通信が可能であること，各プロセッサがプロセッサに閉じた処理を実行している限りはプロセッサ数に関係なく最悪実行時間と最悪割込み応答時間が定まること，プロセッサ間とプロセッサ内の排他制御に関して，プロセッサをまたいだ処理を実行した場合のプロセッサ数に対する最悪割込み応答時間のスケーラビリティがあることを確認した．

しかしながら，残り一つの要件である，プロセッサをまたぐ処理を実行した場合の最悪実行時間については理想的な解放に至っておらず，今後の課題とする．

現在，FDMP カーネルを用いたアプリケーションソフトウェアの開発にも取り組んでおり，それを通じて，有用性を評価する計画である．

謝辞　TOPPERS/FDMP カーネルの実装に対しては，情報処理推進機構（IPA）が実施した 2004 年度未踏ソフトウェア創造事業の支援を受けた．

文　　献

[1] T. Fujiyoshi, et al., “An H.264/MPEG-4 Audio/Visual CODEC LSI with module-wise dynamic voltage/frequency scaling,” Proc. ISSCC, 2005.

[2] 坂村　健（監修），高田広章（編），“μITRON4.0 仕様 Ver.4.02.00,” トロン協会，2004.

[3] M. Kravetz and H. Franke, “Implementation of a multi-queue scheduler for Linux,” http://lse.sourceforge.net/scheduling/mq1.html, 2001.

[4] M. Dobson, P. Gaughen, M. Hohnbaum, and E. Focht, “Linux support for NUMA hardware,” Proc. Linux Symposium 2003, 2003.

[5] 高田広章，本田晋也，“機能分散マルチプロセッサ向けのリアルタイム OS,” 情報処理，vol.47, no.1, pp.36–45, 2006.

[6] A. Suga and S. Imai, “FR-V single-chip multicore processor: FR1000,” FUJITSU Sci. Tech, J., vol.42, no.2, pp.190–199, 2006.

[7] C. Lameter, “Effective synchronization on Linux/NUMA systems,” Gelato Conference, 2005.

[8] 高田広章，坂村　健，“マルチプロセッサリアルタイムカーネルのスケーラビリティを重視した実装,” 信学技報，vol.95, no.603, pp.1–6, March 1996.

[9] J.M. Mellor-Crummey and M.L. Scotto, “Algorithms for scalable synchronization on shared-memory multiprocessors,” ACM Trans. Comput. Syst., vol.9, pp.21–65, 1991.

[10] H. Takada, C. Wang, and K. Sakamura, “Issues for realizing a scalable real-time kernel for function-distributed multiprocessors,” Proc. Work in Progress Session of 17th IEEE Real-Time Systems Symposium, pp.23–26, Dec. 1996.

[11] 鈴木貴久，上方輝彦，“マルチプロセッサ向け μITRON OS の開発,” 情処学研報，計算機アーキテクチャ研報，vol.2005, no.120, pp.57–61, 2005.

[12] D. Sun, D.M. Blough, and V.J. Mooney, “Atalanta: A new multiprocessor RTOS kernel for system-on-a-chip applications,” Technical Report GIT-CC-02-19, Georgia Institute of Technology, Atlanta, Georgia, 2002.

[13] T. Samuelsson, M. Akerholm, P. Nygren, J. Starner, and L. Lindh, “A comparison of multiprocessor real-time operating systems implemented in hardware and software,” International Workshop on Advanced Real-Time Operating System Services (ARTOSS), Porto, Portugal, 2003.

[14] B.D. Theelen, A.C. Verschueren, V.V.R. Suárez, M.P.J. Stevens, and A. Nunez, “A scalable single-chip multi-processor architecture with on-chip RTOS kernel,” J. Systems Architecture: the EUROMICRO Journal, vol.49, no.12-15, pp.619–639, 2003.

[15] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, “Efficient synchronization for embedded on-chip multiprocessors,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.14, no.10, pp.1049–1062, 2006.

[16] J. Lee and V.J. Mooney, III, “A novel deadlock avoidance algorithm and its hardware implementation,” Proc. 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, pp.200–205, 2004.

（平成 19 年 5 月 16 日受付，9 月 14 日再受付）

本田　晋也　（正員）

名古屋大学大学院情報科学研究科附属組込みシステム研究センター助教．2002 豊橋技術科学大学大学院情報工学専攻修士課程了．2005 同大学院電子・情報工学専攻博士課程了．2005 名古屋大学情報連携基盤センター名古屋大学組込みソフトウェア技術者人材養成プログラム産学官連携研究員．2006 から現職．リアルタイム OS，ソフトウェア・ハードウェアコデザインの研究に従事，博士（工学）．

高田　広章　（正員）

名古屋大学大学院情報科学研究科情報システム学専攻教授．1988 東京大学大学院理学系研究科情報科学専攻修士課程了．同専攻助手，豊橋技術科学大学情報工学系助教授等を経て，2003 より現職．2006 より大学院情報科学研究科附属組込みシステム研究センター長を兼務．リアルタイム OS，リアルタイムスケジューリング理論，組込みシステム開発技術等の研究に従事．オープンソースのリアルタイム OS 等を開発する TOPPERS プロジェクトを主宰．博士（理学）．IEEE，ACM，日本ソフトウェア科学会各会員．