機能分散マルチプロセッサ用リアルタイム OS : TOPPERS/FDMP カーネル

（第 8 回 LSI IP デザイン・アワード応募書類 特徴の説明書）

本田晋也[†] 椙山剛[‡] 高田広章[‡]

[†] 名古屋大学 情報連携基盤センター [‡] 名古屋大学 大学院情報科学研究科

1 概要

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。その背景には、消費電力の増大を消費電力の増大を抑えつつ、処理性能の向上をはかるためには、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 1 つの LSI 上に集積したオンチップマルチプロセッサは、処理性能面からも消費電力面からも利点が大きく、広範な組み込みシステムへの適用が期待される。

マルチプロセッサと一口にいっても、OS でサポートするという観点からは、対称型の密結合マルチプロセッサ、機能分散型の密結合マルチプロセッサ、粗結合マルチプロセッサの 3 つのタイプへに分類できる。このうち、組込みシステムの分野では、その特性により機能分散の密結合マルチプロセッサ（以降、単に機能分散マルチプロセッサと呼ぶ）を用いることが一般的である [1]。

現状の機能分散マルチプロセッサ上のソフトウエア開発では、アプリケーションレベルでプロセッサ間の同期・通信を実現しているため、開発コストや保守性に問題がある。そのため、プロセッサ間の同期・通信を OS レベルで実現するリアルタイム OS が必要とされている。

そこで、我々は、国内でリアルタイム OS 仕様のデファクト標準となっている μITRON 仕様 [3] を、機能分散マルチプロセッサ向けに拡張した（以下、これを拡張仕様と呼ぶ）[4]。また、この拡張仕様に基づいて、TOPPERS/FDMP カーネル（以下、 FDMP カーネルと呼ぶ）を実装した[1]。

拡張仕様に関しては、機能分散マルチプロセッサ独自の機能は極力設けないものとし、プロセッサを跨いで μITRON4.0 仕様のシステムコールを実行することが可能である。そのため、既存の μITRON4.0 仕様 OS 向けのソフトウェア資産が活用可能である。実装に関しては、プロセッサ数に対するスケーラビリティの確保や、リアルタイム性（特に割込みに対する応答性）を損なわないための工夫を行った。

FDMP カーネルは、幅広く誰でも自由に利用・変更・配布できるように、TOPPERS プロジェクトからオープンソースとして公開する予定である。

2 組込みシステム向けマルチプロセッサシステムとそのリアルタイム OS

2.1 機能分散マルチプロセッサ

前述のように、マルチプロセッサシステムは、OS でサポートするという観点からは、3 つのタイプへに分類できる。密結合マルチプロセッサとは各プロセッサからアクセスできる共有メモリを持つものをいい、そうでないものを粗結合マルチプロセッサとすることが一般的であるが、その境界は曖昧である。密結合マルチプロセッサの中で、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）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、ある用途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わかっているならば、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。そのため、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、有利であると言える。

2.2 リアルタイム OS のサポート

これまで、機能分散マルチプロセッサ上に組込みソフトウェアを開発する場合、各プロセッサで独立にリアルタイム OS を動作させ、プロセッサ間の同期・通信はアプリケーションレベルで実現する方法が取られてきた。

ところがこの方法には、次の 2 つの問題がある。

(1) プロセッサ間の同期・通信処理は、開発の難しいプログラムであり、アプリケーション毎に開発するとコストがかかる。

[1]TOPPERS とは、ITRON 仕様の技術開発成果を出発点として、組み込みシステム構築の基盤となる各種のオープンソースソフトウェアを開発するプロジェクトの名称である。FDMP カーネルもこのプロジェクトの開発成果の 1 つである。

(2) 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。

(1) の問題については、プロセッサ間の同期・通信処理をライブラリとして用意しておくことで解決可能であるが、(2) の問題については、同じプロセッサ上のタスクとの同期・通信と、異なるプロセッサ上のタスクとの同期・通信が、同一のインタフェースで行えることが必要である。

このことから、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・通信ができるような OS があると、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。

3　TOPPERS/FDMP カーネルの機能

3.1　システムコール

FDMP カーネルの仕様は、μITRON4.0 仕様のスタンダードプロファイルをベースにした。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。そのため、FDMP カーネルは、μITRON4.0 仕様のスタンダードプロファイルで定められている 72 のサービスコールと 11 の静的 API を全てサポートする。

スタンダードプロファイルでは、タスクやセマフォ等のカーネルオブジェクトはすべて静的に生成されることを想定している。FDMP カーネルにおいては、図 1 に示すように、生成されたオブジェクトはプロセッサに固定される。これらのオブジェクトはシステム稼働中に実行プロセッサを変更することはない（システム設計時の静的な移動は可能）。しかしながら、各プロセッサ上のタスクは、システム上の全てのカーネルオブジェクトにアクセスすることが可能である。

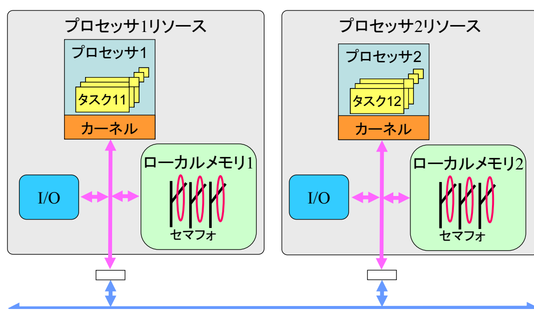

図 1: オブジェクト管理の概念図

3.2　プロセッサ間の同期・通信機能

FDMP カーネルは、プロセッサを跨いで通常の μITRON4.0 仕様のシステムコールを実行することが可能である。μITRON4.0 仕様では、オブジェクトを操作するシステムコールは、操作対象のオブジェクトの ID 番号をパラメータとして指定する。FDMP カーネルでは、図 2 に示すように ID 番号を拡張して、オブジェクトが属するプロセッサを表す。具体的には、ID 番号の上位ビットで属するプロセッサを、下位ビットでそのプロセッサ内でのオブジェクトの識別番号を表す。この拡張により、オブジェクトを操作するシステムコールを、そのままの形で機能分散マルチプロセッサに対応させることができる。

3.3　システム状態

μITRON4.0 仕様では、主に排他制御を実現するために、割込み及びタスク切換えを禁止する CPU ロック状態や、タスク切換えのみを禁止するディスパッチ禁止状態を持つ。

拡張仕様では、これらの状態をプロセッサ毎に独立に管理する。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 ロックやディスパッチ禁止を用いて、他のプロセッサのタスクや割込みハンドラとの排他制御を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 FDMP カーネル上に移植する場合には、CPU ロック状態やディスパッチ禁止により排他制御を行っている部分を、同期オブジェクト（典型的にはセマフォ）を用いて書き換える必要がある。

3.4　静的 API

μITRON4.0 仕様のスタンダードプロファイルでは、カーネルオブジェクトは静的に（システム設計時に）生成する。カーネルオブジェクトの生成情報は、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。

拡張仕様では、全てのカーネルオブジェクトはいずれかのプロセッサに属するため、オブジェクトの登録などを行う静的 API は、プロセッサの囲みの中に記述する。図 8 に、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の例を示す。この例では、各プロセッサで 2 つのタスクと 1 つの周期ハンドラを生成している。このように記述することにより、プロセッサ間でのオブジェクトの（静的な）移動が容易になる。例えば TASK3 を PE2

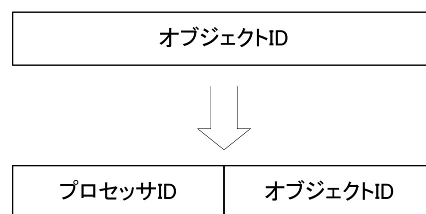

図 2: ID 番号の拡張

から PE1 に移動させるには、CRE_TSK(TASK3..) の行を PE1 の囲みの中に移動するだけでよい。

```
local_class PE1{
    CRE_TSK(TASK1, {TA_HLANG, (VP_INT), 1, ..});
    CRE_TSK(TASK2, {TA_HLANG, (VP_INT), 1, ..});
    CRE_SEM(SEM1, {TA_TFIFO, 1,1});
}
local_class PE2{
    CRE_TSK(TASK3, {TA_HLANG, (VP_INT), 1, ..});
    CRE_TSK(TASK4, {TA_HLANG, (VP_INT), 1, ..});
    CRE_SEM(SEM2, {TA_TFIFO, 1,1});
}
```

図 3: 静的 API の例

4 TOPPERS/FDMP カーネルの特徴

4.1 μITRON4.0 仕様の API によるプロセッサ間の同期・通信

FDMP カーネルは μITRON4.0 仕様の API によるプロセッサの同期・通信をサポートしている。そのため、プロセッサを跨いで通常の μITRON4.0 仕様のシステムコールを実行することが可能である。これにより、プロセッサ間の同期・通信機能とタスク間の同期・通信に互換性を持たせることができ、開発効率の向上やソフトウェアのポーティングを容易化が実現できる。

4.2 TOPPERS/JSP カーネルをベースに

FDMP カーネルの実装にあたっては、μITRON4.0 仕様のスタンダードプロファイルに準拠した TOPPERS/JSP カーネル（以下、JSP カーネル）をベースに用いた。JSP カーネルをベースにしたことにより、FDMP カーネルは JSP カーネル持つ以下の特徴を踏襲している。

- 読みやすく改造しやすいソースコード
- 他のターゲットデバイスへのポーティングが用意な構造
- 高い実行性能と小さい RAM 使用量

また、プロセッサ依存した部分のコードは JSP カーネルとほぼ同じであるため、すでに JSP カーネルがサポートしているターゲットプロセッサへのポーティングは容易である。JSP カーネルでサポートしていた NiosII へのポーティングは、1 日で完了した。

4.3 直接操作法によるオブジェクトの操作

機能分散マルチプロセッサの利点を活かすため、対称型マルチプロセッサ用の OS と異なり、FDMP カーネルでは、プロセッサ毎に独立したオブジェクトコードが動作し、OS の内部情報を管理するデータ構造はプロセッサ毎に持たせている。

このような構成において、異なるプロセッサで実行されるタスク間の同期・通信を実現する方法として、2 つが方法が考えられる。以下では、あるプロセッサ 1 で実行されるタスク A が、他のプロセッサ 2 で実行されるタスク B を起動する状況を例に説明する。

1 つめの方法は、メモリを共有していることを活用して、タスク B の状態を管理するデータ構造（これを、管理ブロックと呼ぶ）をプロセッサ 1 から直接アクセスする方法である。これを直接操作法と呼ぶ。タスク B を起動した結果、プロセッサ 2 でのタスク切換えが必要になった場合には、プロセッサ 1 からプロセッサ 2 に割込みをかけ、タスク切換えを依頼する。

もう 1 つの実現方法として、プロセッサ 1 からプロセッサ 2 にタスク B を起動する旨の依頼を送り、タスク B の管理ブロックへのアクセスは、プロセッサ 2 が行う方法が考えられる。これを遠隔呼出し法と呼ぶ。

この二つの方式を比較した場合、操作対象のプロセッサのメモリ（リモートメモリ）へのアクセスの遅延が大きい場合には、遠隔呼出し法の方が有効である。しかしながら、遠隔呼出し法は、操作対象のオブジェクトが属するプロセッサの動作を阻害してしまい、リアルタイム性が悪くなってしまう。そのため、リモートメモリへのアクセスの遅延が大きくない場合は、直接操作法の方がメリットがあると考え、FDMP カーネルは直接操作法を用いることとした。

4.4 プロセッサ数に対するスケーラビリティーの確保

FDMP カーネルでは、直接操作法により、他のプロセッサのオブジェクトを操作する。そのため、オブジェクトの管理ブロックの操作にあたっては、プロセッサ間での排他制御が必要となる。この排他制御はスピンロックにより実現し、1 つのロックで排他制御を行うデータ構造の集合をロック単位と呼ぶ。組み込みシステムで求められるリアルタイム性と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には、ロック単位を適切な大きさに設定する必要がある。

ロック単位を最も大きくする方法は、OS 内のすべてのデータ構造を 1 つのロックで排他制御する方法（ジャイアントロック）である。この方法は、お互いに関連しない処理を阻害してしまい、スケーラビリティの観点で問題がある。例えばプロセッサ 1 とプロセッサ 2 で、それぞれプロセッサ内に閉じた管理ブロックの操作をしている場合でも、ロック取得により互いの実行を阻害してしまう。これを防ぐために、少なくともプロセッサ毎のロック単位は別々にするべきある。これは自プロセッサのオブジェクト操作で、他のプロセッサの処理を阻害するべきではないからである。一方、あまり細かい粒度でロック単位を設定してしまうと、1 つの処理を行うために取得が必要なロックが多くなり、（特に最悪時の）オーバヘッドが増大してしまう。

また、デッドロック回避が必要なケースを最小限

にするために、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のが望ましい。

これらのことを踏まえ、μITRON4.0 仕様のシステムコールを分析した結果より、プロセッサ毎にタスクロックとオブジェクトロックの 2 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 [5]。タスクロックは、タスク管理に関わるデータ構造用のロックで、具体的にはタスク管理ブロックや時間管理関係のデータ構造を操作する際に取得する。オブジェクトロックは、同期・通信オブジェクトに関わるデータ構造用のロックである。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を操作する際に取得する。

4.5 リアルタイム性の確保

システムコールの実行には、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、前者はスピンロックにより、後者は割込み禁止により実現している。この 2 つの排他制御はお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、注意が必要である。

例えば、プロセッサ間ロックを取得してから割込みを禁止する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要求があると、それを受け付けることになる。受け付けた割込みの処理を行っている間は、ロックを取得した状態となるため、その間、他のプロセッサを無駄に待たせてしまう。逆に割込みを禁止してからロックの取得を試みると、ロックの取得を待つ間、割込みが禁止状態となる。結果的に、割込み禁止時間が長くなり、割込み応答性が低下することになる。

このようにどちらの排他制御を先に実行してもしても問題が発生するため、2 つの排他制御をアトミックに実行することが必要になる。

この問題を解決するために、FDMP カーネルでは次の方法をとっている。まず最初に割込みを禁止し、その後、Test&Set ロックによりロックの取得を試みる。ロック取得に成功しなかった場合（言い換えると、ロック取得を待っている場合）には、割込み要求が発生しているかをチェックし、発生していた場合には、割込みを許可して割込みを受け付ける。

さらに、オブジェクトロックを取得した状態で、タスクロックを取得する場合には、次の方法をとっている。タスクロックの取得を試みている間に割込みが要求があった場合には、取得済みのオブジェクトロックを一旦解放し、割込みを許可して割込みを受け付ける。その後、1 つめのオブジェクトロックの取得からやり直す。このように、タスクロックの取得ができない場合、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ため、この間コードは非破壊コードである必要がある。

この方法により、プロセッサ内排他制御とプロセッサ間排他制御の問題を一応解決しているものの、そもそも Test&Set ロックは、ロックが取得できるまでの上限時間が定まらないため、厳密な時間制約が課せられるリアルタイムシステムには適当ではない。厳密な時間制約が求められる場合の実装技術については、今後の課題である。

5 動作環境と使用したツール

現状の FDMP カーネルは、以下の前提を満たすマルチプロセッサシステムで動作する。

(1) 各プロセッサのためのデータを置くメモリが、全てのプロセッサから同一のアドレスでアクセス可能であること

(2) 任意のプロセッサに割込み（プロセッサ間割込み）を発生可能であること

(3) プロセッサ間での排他制御のための機構を持ち、これを用いてプロセッサ数の 2 倍の個数のロックを実現できること

(1) と (2) に関しては、直接操作法により他のプロセッサのオブジェクトを操作するためである。(3) の排他制御のための機構は、OS 内部のデータ構造の排他制御に必要なもので、プロセッサがそのための命令（例えば test&set 命令）を持っているか、排他制御を実現するハードウェアモジュールを持っていることが必要である。個数の条件に関しては、排他制御をハードウェアモジュールで実現する場合に制限になる場合がある。

現在、表 1 に示すターゲットプロセッサ上での動作を確認している。

メーカー	プロセッサ
東芝	MeP（Media embedded Processor）
Altera	NiosII
Xilinx	Microblaze

表 1: ターゲットプロセッサ

開発ツールに関しては、それぞれのプロセッサ用の GCC（GNU Compiler Collection）を用いている。

6 性能評価

FDMP カーネルとそのベースとなったシングルプロセッサ用のリアルタイム OS である JSP カーネルとのコードサイズ、システムコールの実行時間を比較する。

評価環境としては、東芝社の MeP を用いたシステム LSI である T5V[2] を用いた。T5V はプロセッサコアを 4 つ搭載しており、メモリはローカルメモリと共有メモリを持っている。今回の評価では、ローカルメモリは使用せず、共有メモリのみを用いキャッ

シュは使用していない。また、動作周波数は150MHzである。

6.1 コードサイズ

簡単なサンプルアプリケーションとカーネルをリンクした場合のメモリ使用量を表2に示す。プログラムサイズ（text）は55%ほど増大している。これは、マルチプロセッサ化による排他制御のためのコードによるところが多い。一方、データ（data）や初期値無しデータ（bss）はJSPカーネルから大きくは増大していない。

カーネル	text	data	bss	計
JSPカーネル	17718	40	18748	36506
FDMPカーネル	27554	41	19056	46779

表2: コードサイズ（単位はbyte）

6.2 システムコール実行時間

セマフォの解放を行うシステムコールであるsig_semの処理時間を測定した。sig_semはセマフォの解放の結果、ディスパッチを伴わないパターンを測定した。測定条件は、JSPカーネルで実行した場合、FDMPカーネルでsig_semを呼び出すタスクとセマフォが同じプロセッサに属する場合（プロセッサ内）、異なるプロセッサに属するセマフォを操作する場合（プロセッサ間）の3つの条件で測定した。なお、バス負荷の条件を同じにするため、どの測定条件においてもプロセッサ2個を動作させている。

測定はそれぞれ1000回行い分布を求めた。測定結果を表3に示す。測定結果を見るとそれぞれの測定条件において、測定値が大きく2つに分かれているの。これは、測定中にシステムに存在する1msec周期のカーネル用のタイマ割込みが入り、その割込みハンドラの処理ために測定時間が増加してしまうためである。この増加した時間は割込みハンドラの処理時間と言える。

JSPカーネルとFDMPカーネル（プロセッサ内）を比較すると、sig_semの処理時間は12.7μsec増大している。割込み処理時間はJSPカーネルが21.9μsec、FDMPカーネル（プロセッサ内）が、24.9μsecとなっており、FDMPカーネル（プロセッサ内）の方が3.0μsec増大している。

JSPカーネルとFDMPカーネルの割込み処理を比較すると、一つのタスクロックの取得と解放を行う以外はほぼ同じコードとなっている。そのため、一つのロックの取得と解放に必要な時間は3.0μsec程度であると言える。一方、sim_semの処理に内部には、タスクロックとオブジェクトロックの二つのロックを取得するため、処理時間の増加分の6.0μsecはロックの処理に要している。残りの6.7μsecは、マルチプロセッサ対応による処理の増加である。

FDMPカーネルのプロセッサ内とプロセッサ間では、処理時間はほぼ同じになっている。これは今回評価に用いた環境が共有メモリを用いて動作しているため、自プロセッサのオブジェクトと他プロセッサのオブジェクトのアクセスに要する時間が変わらないためである。

以上の結果より、シングルプロセッサ用のリアルタイムタイムOSと比較視して、sig_semの実行時間は86%増、割込み処理時間は14%増に抑えられ、実用に十分な性能となっている。

条件	時間（μsec）	回数
JSPカーネル	14.6	981
	36.5	19
FDMPカーネル	27.3	969
（プロセッサ内）	52.2	31
FDMPカーネル	27.3	973
（プロセッサ間）	52.2	27

表3: sig_semの処理時間

6.3 まとめと今後の課題

FDMPカーネルは、2005年4月よりTOPPERSプロジェクトの会員向けに配付しており、近日中には、オープンソースソフトウェアとして一般公開する予定である。

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。その他の課題としては、FDMPカーネル上でアプリケーションを開発する際に必要となる、シミュレーション環境やデバッグ環境に関する研究を挙げることができる。

謝辞

本ソフトウェアの開発の一部は、2004年度IPA未踏ソフトウェア事業の援助を受けた。プロジェクトマネージャーの中島達夫先生に感謝致します。

参考文献

[1] 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 (Jan, 2006).

[2] Fujiyoshi, T. et.al.: An H.264/MPEG-4 Audio/Visual CODEC LSI with Module-Wise Dynamic Voltage/Frequency Scaling, Proc. ISSCC (Feb. 2005).

[3] 坂村健監修, 高田広章編: μITRON4.0仕様 Ver.4.02.00, トロン協会 (2004).

[4] 高田広章: 機能分散マルチプロセッサ向けリアルタイムカーネル仕様, http://www.toppers.jp/documents.html, TOPPERSプロジェクト (2005).

[5] Hiroaki Takada and Ken Sakamura : "Inter- and Intra-Processor Synchronizations in Multiprocessor Real-Time Kernel", Proc. 4th International Workshop on Parallel and Distributed Real-Time Systems (Apr. 1996).