Vol.2009-EMB-14 No.5
2009/7/24

マルチプロセッサ対応 RTOS における ロードバランス機構の実現

石田利永子[†]　本田晋也[†]　高田広章[†]
福井昭也[††]　小川敏行[†††]　田原康宏[†††]

[†]名古屋大学大学院情報科学研究科
E-mail:r_ishida@nces.is.nagoya-u.ac.jp,{Honda,hiro}@ertl.jp
[††]株式会社ルネサスソリューションズ E-mail:fukui.akiya@renesas.com
[†††]株式会社ルネサステクノロジ
E-mail:{ogawa.toshiyuki2, tawara.yasuhiro}@renesas.com

マルチプロセッサ対応 RTOS において，RTOS 自身が各プロセッサの負荷を見て動的にロードバランスを行う SMP 型 OS では，リアルタイム性の保証が困難である．そこで TOPPERS プロジェクトにおいて，AMP 型 OS にタスクマイグレーション機能を追加した組込みシステム向けマルチプロセッサ対応 RTOS である TOPPERS/FMP カーネルが開発された．TOPPERS/FMP カーネルでは，RTOS が動的にロードバランスを行うのではなく，ユーザーの要求時（API）にのみタスクの移動を行うことで，RTOS 自体のリアルタイム性を高めている．

本稿では，TOPPERS/FMP カーネルが提供する API を使用して，ユーザーレベルで，ロードバランスの実現が可能か評価した．具体的には，Linux®**[a]**で行われているロードバランス機構を調査し，TOPPERS/FMP カーネルに適したタスクのマイグレーションの方法を 2 方式考案し，その方式を比較検討した．評価の結果，各プロセッサの負荷を取得する API と，指定した優先度のタスクを指定したプロセッサへマイグレーションする API の 2 つの API を追加することで，ユーザーレベルでロードバランス機構を実現できることを確認した．

Loadbalance Algorithm in a Real-TIme Operating System for Multiprocessors

Rieko Ishida[†]　Shinya Honda[†]　Hiroaki Takada[†]
Akiya Fukui[††]　Toshiyuki Ogawa[†††]　Yasuhiro Tawara[†††]

[†]Graduate School of Information Science Nagoya University
[††]Renesas Solutions Corp.
[†††]Renesas Technology Corp.

In a RTOS for multiprocessors, symmetric load balancing, in which the kernel itself watches a load of each processor and dynamically balances the load among processors, sometimes makes it difficult to guarantee real-time behavior of the system. To address this problem, we have developed the TOPPERS/FMP kernel. This kernel will improve the real-time behavior of the multiprocessor system by allowing a user to move a task to another processor on demand through an API rather than by letting the kernel itself perform dynamic load balancing.
In this work, we evaluated feasibility of user-level load balancing using the original API's of the TOPPERS/FMP kernel. After investigating the convention of load balancing employed by Linux, we created two candidates of load balancing methods which would be suitable for the TOPPERS/FMP kernel. Adding two new API's was necessary to implement either of the load balancing methods. One API returns a load of a processor, and another API moves a task of a designated priority to a designated processor. We weighed the two load balancing methods and selected one which was suitable for the TOPPERS/FMP kernel.

1. はじめに

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ほう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，対称性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の 2 つのタイプに分類できる．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．よって，組込みシステムの分野では，上記 2 つのタイプのマルチプロセッサのうち，機能分散型の密結合マルチプロセッサを用いることが一般的である[1].

一般的なマルチプロセッサで用いられる対称性 OS（SMP 型 OS）は，高いスループットを得るために各プロセッサへの動的なタスク分配を行う．

それに対し，組込みシステムではリアルタイム性保証や検証性の観点から，機能分散型の密結合マルチプロセッサを用いて，タスク配置を静的に行える方が都合が良い．そこで，組込みシステムでは，タスク配置を静的に行うマルチプロセッサ向け RTOS である，機能分散型 OS（AMP 型 OS）を用いるのが一般的である．一方で，タスク配

a)Linux は、Linus Torvalds 氏の日本およびその他の国における登録商標または商標です

置を静的に行うことは，負荷変動が発生しプロセッサ間に負荷の不均等が生じる場合にプロセッサの利用率が低下する事が考えられる．この結果，動的に負荷分散を行うSMP 型 OS と比較して，スループットが大幅に低下する可能性がある．[2]

TOPPERS プロジェクト[3]で開発された TOPPERS/FMP カーネルは，SMP 型 OS と AMP 型 OS の両方の利点を生かしたマルチプロセッサ用 RTOS である．SMP 型の利点である動的なロードバランスを実現する仕組みを用意しつつ，AMP 型 OS の利点であるリアルタイム性の確保も容易になっている．

アプリケーションによって，特性が異なるため，動的なロードバランスの目標を一概に決めることは難しい．例えば，制御系のアプリケーションと，マルチメディア系アプリケーションとでは，分散させる負荷対象も，ロードバランス方針も全く違うものになる．よって，ロードバランス方針を OS レベルで判断することはできない．TOPPERS/FMP カーネルでは，カーネルが動的にロードバランスを行うのではなく，ユーザが API を発行したときにのみ，タスクをマイグレーションする機能を提供した．つまり，ユーザがアプリケーションの特徴に応じて，ロードバランス方針を決定できるようにした．

そこで本稿では，TOPPERS/FMP が提供する API を使用して，ユーザレベルでロードバランスが実現可能か評価を行った．具体的には，Linux 2.6.10 で行われているロードバランスを調査し，TOPPERS/FMP カーネルに適したロードバランス手法の検討を行った．なお，Linux では，定期的，自プロセッサが idle 状態になるとき，プロセスの起動時，プロセスの起床時の 4 つのタイミングでロードバランスを行うが，本稿では，定期的に行うロードバランスについて検討した．

2. TOPPERS/FMP カーネル

TOPPERS/FMP カーネルは，SMP 型 OS の，動的なタスクの移動が可能という利点と，AMP 型 OS の，リアルタイム性を保証しやすいという両方の利点の両立を目指したマルチプロセッサ用 RTOS である．μITRON4.0 仕様の API をどのプロセッサに対しても，発行できるように拡張した．

TOPPERS/FMP カーネルは，タスクを実行するプロセッサを静的に決定するのが基本である．システム構築時に設計者がタスクをプロセッサに割り当てる．リアルタイム性の確保のため，カーネルは動的にロードバランスを行う機能を持たないが，図 1 に示すように，タスクをマイグレーションする API を提供することで，SMP 型 OS の利点である動的なロードバランスを実現する仕組みを用意している．

AMP 型 OS のリアルタイム性を確保しやすいという利点を享受するため，TOPPERS/FMP カーネルでは，図 1 に示すように，プロセッサごとにレディキューを持っている．各プロセッサのレディキューは個別に管理するため，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，他のプロセッサの影響を受けることはない．つまり，OS 内での排他制御が必要ないため，システムコール発行時の平均的なスループットが向上している．また，スケジューリングはプロセッサ毎に，プリエンプティブな優先度ベースのスケジューリングを行うので，シングルプロセッサとほとんど変わらずリアルタイム性を保証しやすいと言える．

このように，TOPPERS/FMP カーネルは，リアルタイム性とロードバランスの実現の両方を目指したマルチプロセッサ用 RTOS といえる．

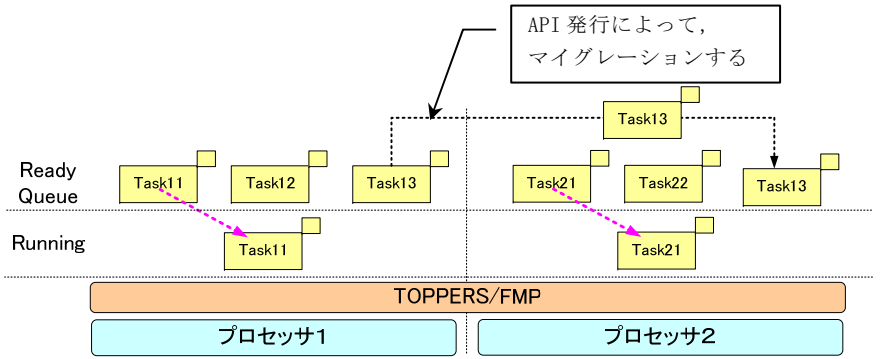

図 1　TOPPERS/FMP カーネル

TOPPERS/FMP カーネルで提供している API のうち，マルチプロセッサに特有の API について述べる．

(1)タスクマイグレーション機能
・mig_tsk(ID tskid, ID prcid)
　・tskid で指定したタスクを prcid で指定したプロセッサに移動させる．

TOPPER/FMP カーネルで提供しているタスクのマイグレーション機能は，API を発行したプロセッサ（以下，自プロセッサ）に割り付けられたタスクに対してのみ可能である．また，非タスクコンテキストからは本 API は発行できない．これらは，システムコールの最悪実行時間を抑えるための制約である．

(2)プロセッサ指定のタスク起動機能
・mact_tsk(ID tskid, ID prcid)/imact_tsk(ID tskid, ID prcid)

・tskid で指定したタスクを prcid で指定したプロセッサで起動する．

3. Linux におけるロードバランス機構

TOPPERS/FMP カーネルに適した，ロードバランス機構を検討するために，Linux で行われているロードバランス手法を調査した．

3.1 Linux のスケジューリングの特徴

ITORN 系 OS のレディキューに相当するものを，Linux では RUN キューと呼んでいる．Linux の RUN キューは，2 種類のキュー（active キュー，expired キュー）で構成されている．また ITRON 系 OS でタスクと呼んでいる処理単位を，Linux ではプロセスと呼んでいる．

・active キュー　：実行可能状態で，実行割り当て時間を待っているプロセスのキュー
・expired キュー：実行可能状態だが，実行割り当て時間を使い果たしたプロセスのキュー

この 2 種類で対となり RUN キューを形成している．

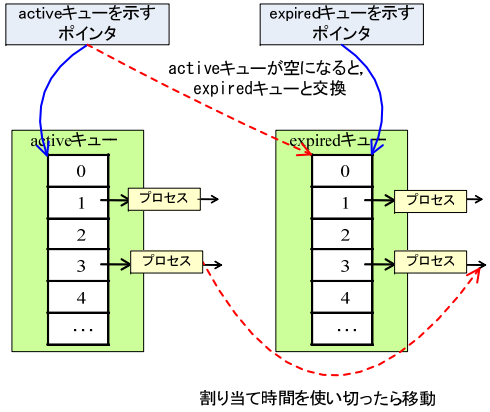

図 2　Linux の RUN キュー

実行可能なプロセスは RUN キューの該当する実行優先度にリンクされる．図 2 に示す様に，実行割り当て時間を使い切ったプロセスは，active キューから expire キューへ移動する．active キュー上にプロセスがいなくなると，expire キューを active キューとして処理を続ける．active キュー，expire キュー両方にプロセスがいなくなると，idle キューに登録された処理（idle プロセス）が実行される．idle プロセスは RUN キューにつながれない，カーネルプロセスとなる．

マルチプロセッサシステムに対応した Linux では，プロセッサごとにこの RUN キューを持つ．このことにより，TOPPERS/FMP カーネルと同じように，OS 内での排他制御が必要ないため，OS 内での平均的なスループットが向上している．

Linux には，スケジュールドメインとグループという概念がある．スケジュールドメインとは，カーネルが負荷の均衡を保つべきプロセッサの集合体である．そして，グループとは，スケジュールドメイン内のプロセッサの部分集合である．グループは，ロードバランス時の比較の単位となり，グループ間の負荷が均一化するようにロードバランスが行われる．グループ機能は，ハードウェア・マルチスレッディングや大規模システムのための機能である．組込みシステムでは比較的小規模なシステムを扱うことが多いため，本稿ではグループ機能は対象外とし，1 グループに 1 つのプロセッサが割り付けられた SMP 型の例を参考にした．

3.2 Linux におけるロードバランス手法

プロセッサ間の負荷を平準化するようにロードバランスが行われる．スケジューリングを行うプロセススケジューラは各プロセッサで独立して動作する．つまり 4 つのプロセッサで構成されるマルチプロセッサであれば，それぞれ 4 つのプロセッサにスケジューラが存在し，それぞれがロードバランスを行う．プロセッサ間で負荷に大きな隔たりが出たときにロードバランスが行われるが，キャッシュや TLB の有効利用のため，プロセスは可能な限り特定のプロセッサに割り付けられて動作する．

負荷（cpu_load）を RUN キュー上にある実行可能なプロセスの数（nr_running）を元に，式(1)で算出している．

$$cpu\_load = (cpu\_load + nr\_running * 128) / 2 \qquad (1)$$

この値は，定期的に更新される．Linux バージョン 2.6.10 では 1 ミリ秒ごと，バージョン 2.6.13 以降では，4 ミリ秒ごとに更新される．式(1)で算出される負荷は，実行可能なプロセス数の積分と考えてよい．積分するのは，一時的に増減するプロセス数の影響を受けないようし，マイグレーションの頻発を防ぐためである．不定期にプロセス数が急増，急減するようなシステムの場合，プロセス数をそのまま負荷因子としてしまうと，プロセスのマイグレーションが頻発する可能性がある．前述したように，キャッシュや TLB の有効利用を考えると，プロセスは一度起動したら，できるだけ同

じプロセッサ上で動作する方が有利であるので，プロセスのマイグレーションはできるだけしない方針である．よって，負荷をプロセス数の積分とすることで，プロセス数の変化を長期的にとらえようとしている．そこで，Linux ではプロセス数の積分を負荷因子としている．

Linux は，定期的，自プロセッサが idle 状態になるとき，プロセスの起動時，プロセスの起床時の 4 つのタイミングでロードバランスを試みている．本稿では，定期的に行うロードバランスを参考にした．

3.2.1 定期的に行うロードバランス機構

定期的に行うロードバランスの間隔は，idle プロセスが実行している時，つまり実行可能なプロセスが存在しない場合は 1 ミリ秒ごとに，他のプロセスが実行している場合は，64 ミリ秒ごとである．前述したように，プロセッサ間の負荷が均一になるようにプロセスのマイグレーションがロードバランスにより行われる．

定期的に行うロードバランスでは，全プロセッサの平均負荷を算出し，自プロセッサの負荷がそれより高ければ，プロセスのマイグレーションを行わない．反対に低ければ，負荷の高いプロセッサから自プロセッサへプロセスのマイグレーションを行う．このようなロードバランスを行うことで，全プロセッサの負荷を平均に近づけようとする．実際にプロセスのマイグレーションが行われるのは，自プロセッサより，1.25 倍以上負荷が大きいプロセッサが見つかった場合である．つまり，自プロセッサの負荷がある程度大きければ，自プロセッサはマイグレーションの必要がないと判断し，プロセスのマイグレーションは行われない．このように負荷の小さいプロセッサにより，プロセスのマイグレーションが行われる．以下と図 3 に，各プロセッサで行われるロードバランス処理詳細を示す．

（STEP1）ロードバランス用の各プロセッサの負荷を見積もる．
（式 1 ）で算出した負荷をそのままプロセッサの負荷とするのではなく，自プロセッサの負荷は，cpu_load と nr_runnning*128 を比較して大きい値を負荷とし，他プロセッサの負荷を見積もるときは，反対に，cpu_load と nr_running*128 の小さい値を負荷として見積もる．このように自プロセッサの負荷を大きく見積もることで，マイグレーションの頻発を防いでいる．

（STEP2）マイグレーション実行判断を行う．
自プロセッサの負荷(this_load)が平均負荷(ave_load)より小さい．かつ，最高負荷(max_load)との差が 1.25 倍以上のときに，マイグレーションが必要と判断する．

（STEP3）移動するプロセス数(imbalance)を式(2)で算出する．
最高負荷と平均負荷の差と，平均負荷と自プロセッサの負荷の差の小さい方を移動するプロセッサ数とすることで，マイグレーション後に自プロセッサの負荷が平均負荷を超えないようにしている．マイグレーションすることによって，自プロセッサの負荷が，平均負荷を超えてしまうと，さらにマイグレーションが発生してしまう可能性があるからである．

$$imbalance = \min((max\_load - ave\_load),(ave\_load - this\_load)) / 128 \quad (2)$$

式(2)より算出された imbalance が 1 未満となる場合には，最大負荷と自プロセッサの負荷の差が 2 タスク以上の場合は，1 タスク移動する．これは，最後にあるプロセッサにのみ複数のプロセスが残ってしまった場合にも，ロードバランスが行われるようにするための工夫である．

（STEP4）移動するプロセスの選択をする．
最高負荷プロセスのプロセスから，マイグレーションすることによる影響が少ないプロセスを選択する．

（STEP5）マイグレーションする．
式(2)で求めたプロセス数分，最高負荷プロセッサから自プロセッサにプロセスをマイグレーションする．

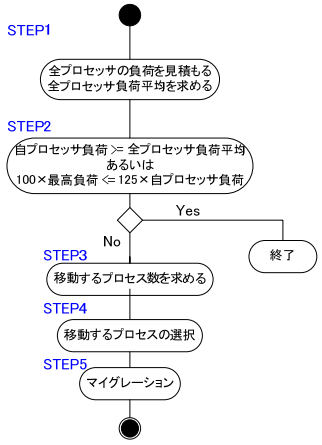

図 3　定期的に行うロードバランス

上記の処理を，プロセッサごとに行うことで，全体として負荷が平準化される．

3.2.2 その他のタイミングで行うロードバランス

Linux で行われるロードバランスのうち，定期的に行われるロードバランス以外のタイミングで行われるロードバランス概要について述べる．

(1)自プロセッサが idle 状態になるとき

active キューにも，expired キューにもプロセスがいなくなった場合には，idle プロセスが起動し，idle 状態となる．Linux では idle 状態になる前にも，ロードバランスを行う．その際に行われるロードバランスの方針も，定期的に行う場合と同じである．

(2)プロセスの起動時

プロセスの起動時に行われるロードバランスでは，まだ実行状態でないプロセスをどこのプロセッサで起動するか選択する．プロセスを起動するサービスコールを発行したプロセッサの負荷が最低負荷プロセッサの 1.12 倍以上大きい場合は，最低負荷プロセッサで起動した方が負荷が均一化されると判断して，最低負荷プロセッサで起動する．

(3)プロセスの起床時

プロセスの起床時に行われるロードバランスでは，起床するプロセスをどこのプロセッサで起床するか選択する．対象となるプロセスは，以前にどこかのプロセッサで実行していたため，マイグレーションすることによるキャッシュや TLB の影響は大きい．よって，できるだけ以前に実行していたプロセッサで起床するようにしている．以前実行していたプロセッサの負荷が，起床サービスコールを発行したプロセッサの負荷より，1.37 倍以上大きければ，起床サービスコールを発行したプロセッサで起床する．

4. TOPPRS/FMP カーネル向けのロードバランス機構

Linux で定期的に行われるロードバランスを参考にして，TOPPERS/FMP カーネルに適したロードバランス手法を検討した．

方針としては，ロードバランス機構をカーネル内部で実現するのではなく，ユーザレベルで，ミドルウェアとして実現する．ただし，カーネルに大きな影響を与えない場合に限り，汎用性がある機能や API であればその機能と API をカーネル内に追加する．また，TOPPERS/FMP カーネルのマイグレーション仕様は変更しないものとする．

本研究では，ロードバランスの対象となるタスクは全て同一優先度とし，それらのタスクは，特定の優先度のタスクの優先順位を回転する API(rot_rdq)を発行することで，ラウンドロビンスケジューリングでスケジューリングされているものとする．

4.1 Linux と TOPPERS/FMP カーネルの比較

本節では，Linux と TOPPERS/FMP カーネルのスケジュール方針と，マイグレーション方針の違いを述べる．

Linux では TTS（Time Sharing System）方式でスケジュールを行っている．実行可能状態となったプロセスに，実行割り当て時間を与える．そして実行状態となったプロセスは，実行割り当て時間を使い切ったら，他のプロセスに時間を明け渡す．実行割り当て時間を使い果たすと，スケジューリングの対象外となる．実行割り当て時間が残っている実行待ちプロセスが 1 つもいなくなると，すべてのプロセスに新たに実行割り当て時間を割り当てる．このように，必ずすべてのプロセスに実行権が割り当てられる保証をしている．TOPPERS/FMP カーネルで採用しているスケジュール方法は，プロセッサごとのプリエンプティブな優先度ベーススケジューリングであるので，大きな違いがある．しかし前述したように，本研究ではロードバランスの対象となるタスクは全て同一優先度で，かつ，ラウンドロビンスケジューリングでスケジューリングされているので，TTS 相当である．

また，Linux におけるプロセスのマイグレーションは，他のプロセッサに割り付けられたプロセスをマイグレーションすることも可能である．それに対して，TOPPERS/FMP カーネルでは，前述したようにシステムコールの最悪実行時間を抑えるため，自プロセッサに割り付けられたタスクのみをマイグレーション可能とする仕様としている．以上を表 1 にまとめる．

表 1　Linux と TOPPERS/FMP カーネルの比較

	Linux	TOPPERS/FMP カーネル
スケジューリング方式	TTS	プリエンプティブな優先度ベーススケジューリング ※ロードバランス対象のタスクは，ラウンドロビンスケジューリング
レディキュー	プロセッサごとに持つ	プロセッサごとに持つ
expire キュー	持つ	持たない
マイグレーション	他プロセッサに割り付けられたプロセスもマイグレーション可能	自プロセッサに割り付けられたタスクのみマイグレーション可能

4.2 追加した機能と API

カーネル内部に追加した機能としては，各プロセッサが 1 ミリ秒ごとに，式(1)を用

いて自プロセッサの cpu_load を更新するように追加変更を行った．本研究では，ロードバランス対象のタスク以外のタスクも含む全タスク数を元に，cpu_load を算出している．

TOPPERS/FMP カーネルが提供する API だけでは，ロードバランスを実現することができなかったので，新たに下記 API を２つ追加した．

・get_lod(ID prcid);
　prcid で指定したプロセッサの負荷を取得
・mig_pri(PRI pri, ID prcid);
　pri で指定された優先度のタスクを１つ, prcid で指定されたプロセッサへ mig_tsk と同じ方針でマイグレーションする．マイグレーションするタスクは，指定された優先度のタスクのうち，最低優先順位のタスクとする．

この２つの API を追加することで，ロードバランスをミドルウェアとして実現することが可能であることを確認した．

4.3 TOPPERS/FMP カーネルにおけるロードバランス方針

ロードバランスの目的は，各プロセッサの実行状態および，実行可能状態のタスク数を平準化することとする．各プロセッサの負荷(cpu_load)はレディキューにつながれているタスク数をもとに，式(1)により算出した．この値は各プロセッサがそれぞれ，周期ハンドラで定期的に自プロセッサの値を更新する．本研究では，1 ミリ秒ごとに更新することとしたが，更新間隔は，任意に設定することができる．

ロードバランス処理の概要としては，大きく判断処理とマイグレーション処理の 2 つの処理に分類される．

・判断処理
(1) プロセッサごとに追加 API である get_lod を用いて，全プロセッサの負荷を算出し，全プロセッサの負荷平均を算出する．その値を用いて，ロードバランスが必要か判断し，必要であれば移動タスク数を式(2)より求める．必要でなければ，図 4 に示すように，ロードバランスを終了する．図 3 で示す，Linux での処理 STEP1 から STEP4 まで相当する．
(2) マイグレーション処理を行うプロセッサに対して，データキューで通知を行う．

・マイグレーション処理
(3) データキューを受信する．
(4) 追加 API である，mig_pri を発行する．図 3 で示す，Linux での処理 STEP5 が相当する．

実装では，図 4 に示すように，各プロセッサに，判断処理を行う周期ハンドラ，マイグレーション処理を行うサーバタスクを準備した．そして，この周期ハンドラとサーバタスク間の通信は，データキューで行った．判断処理は，周期ハンドラで定期的に行う． 本研究では，この周期を 50 ミリ秒ごととしたが，この周期は任意に設定することができる．

自プロセッサに割り付けられたタスクのみマイグレーション可能という制約により，マイグレーション処理は，負荷の高いプロセッサが行う．

判断処理とマイグレーション処理は別プロセッサで行う方式を取ることもできるし，同一プロセッサで行う方式を取ることも可能である．本稿では，負荷の低いプロセッサが判断処理を行い，マイグレーション処理を行うプロセッサへマイグレーションを依頼する方式（依頼方式）と，負荷の高いプロセッサが判断処理を行い，マイグレーションする方式（追い出し方式）の 2 方式を検討した．依頼方式では，判断処理とマイグレーション処理を行うプロセッサは異なるのに対して，追い出し方式では，同一プロセッサが行う．

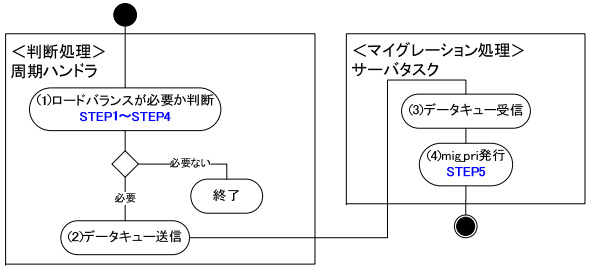

図 4　ロードバランス処理概要

4.4 依頼方式

依頼方式では，負荷の低いプロセッサがタスクのマイグレーションが必要か判断をする．判断処理を行うプロセッサの負荷がある程度低ければ，自プロセッサへのタスクのマイグレーションが必要と判断する．反対に，判断処理を行うプロセッサの負荷がある程度高ければ，自プロセッサにタスクをマイグレーションする必要はないと判断し，マイグレーションを行わない．

タスクのマイグレーションが必要と判断した場合には，最高負荷のプロセッサに対して，自プロセッサへマイグレーションするタスク数を伝え，マイグレーションを依

頼する．つまり，図 5 に示すように，最高負荷プロセッサがマイグレーション処理を実行する．そして，その依頼を受け取ったプロセッサのサーバタスクが，判断処理を行ったプロセッサへタスクをマイグレーションするという仕組みとした．

判断処理の(1)ロードバランスが必要か判断する処理では，図 6 に示すように，自プロセッサ負荷が平均負荷より小さい．かつ，最高負荷が自プロセッサ負荷の 1.25 倍以上のときにマイグレーションが必要と判断する．

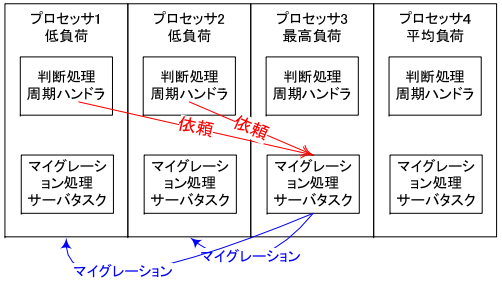

図 5　依頼方式

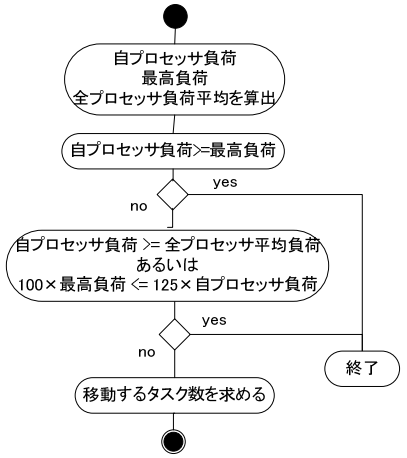

図 6　依頼方式での判断処理

マイグレーションが必要と判断した場合には，移動するタスク数を式(2)より算出する．この際，移動するタスク数が 1 タスク未満となる場合には，最大負荷と自プロセッサの負荷の差が 2 タスク以上の場合は，移動するタスク数を 1 とする．最高負荷プロセッサに対して，移動するタスク数と自プロセッサ ID をデータキューで送信し，マイグレーションを要求する．

この方式は，図 5 に示すように，判断処理を行ったプロセッサが，最高負荷プロセッサへマイグレーションを依頼することから，依頼方式と名付けた．

依頼方式では，判断処理とマイグレーション処理は別プロセッサで行うため，データキュー送信にはプロセッサ間割込みを伴う．

4.5 追い出し方式

判断処理の結果，負荷を平準化させるために，負荷の高いプロセッサから負荷の低いプロセッサに対してマイグレーションが行われる．依頼方式では，負荷の低いプロセッサが判断処理を行い，最高負荷のプロセッサへデータキュー送信によりマイグレーションを依頼する．このプロセッサ間データキュー送信は，データキューを送信したことを受信プロセッサに知らせるために，プロセッサ間割込みが伴う．一般にプロセッサ間割込みは実行オーバヘッドが大きい．そこで依頼方式に対して，負荷の高いプロセッサが自分自身で判断処理を行い，必要であればマイグレーションを行う追い出し方式を検討した．この方法であれば，負荷の高いプロセッサが判断処理を行う必要があるが，実行オーバヘッドの大きいプロセッサ間割込みが発生しない．

追い出し方式では，負荷の高いプロセッサが判断処理を行う．判断処理を行うプロセッサの負荷がある程度高ければ，自プロセッサからタスクのマイグレーションが必要と判断する．反対に，判断処理を行うプロセッサの負荷がある程度低ければ，自プロセッサからのタスクのマイグレーションが必要ないと判断し，マイグレーションを行わない．

タスクのマイグレーションが必要と判断した場合には，自プロセッサが最低負荷のプロセッサに対してタスクをマイグレーションする．つまり，図 7 に示すように，判断処理とマイグレーション処理は同一プロセッサが行う．自分自身がマイグレーション処理を行うにも関わらず，通知をデータキュー送信で行うのは，前述したように，mig_tsk は非タスクコンテキストからは発行できないからである．

判断処理の(1)ロードバランスが必要か判断する処理では，図 8 に示すように，自プロセッサ負荷が平均負荷より大きい．かつ，最低負荷との差が 1.25 倍以上のときに，マイグレーションが必要と判断する．

マイグレーションが必要と判断した場合には，移動するタスク数を式(2)より算出する．この際，移動するタスク数が 1 タスク未満となる場合には，最低負荷と自プロセッサの負荷の差が 2 タスク以上の場合は，移動するタスク数を 1 とする．自プロセッサのサーバタスクに対して，移動するタスク数と最低負荷プロセッサ ID をデータキ

ューで送信し，マイグレーションを行う．

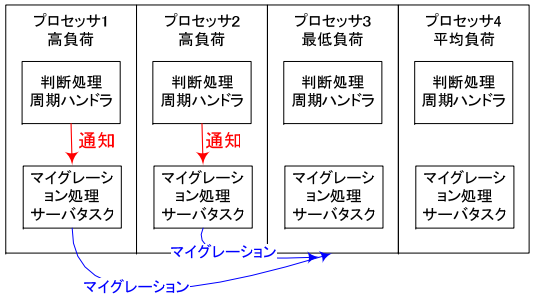

図 7　追い出し方式

この方式は，図 7 に示すように，判断処理を行うプロセッサ自身が，最低負荷プロセッサへタスクをマイグレーションすることから，追い出し方式と名付けた．

追い出し方式では，判断プロセッサとマイグレーション実行プロセッサは同一プロセッサであるため，データキュー送信にはプロセッサ間割込みは伴わない．

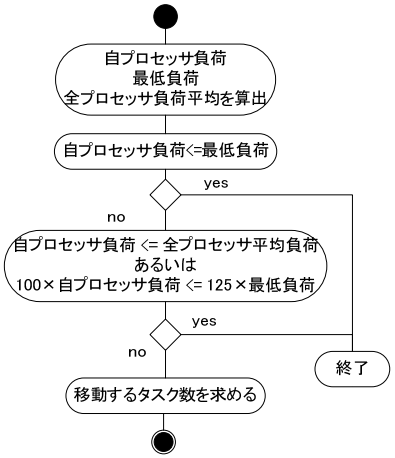

図 8　追い出し方式での判断処理

5.　評価

依頼方式と追い出し方式それぞれの実行オーバヘッド測定と，4 プロセッサによるロードバランスの確認と評価を行った．本研究で使用したマルチプロセッサ環境を表 2 に示す．

表 2 評価環境

プロセッサ名称	SH-X3
プロセッサ数	4 個
各プロセッサ動作周波数	600MHz

5.1 実行オーバヘッドの測定

実行オーバヘッドの測定は，プロセッサ 2 個で行った．残り 2 個のプロセッサは停止させている．ロードバランスによりタスクが 1 つマイグレーションする状況を 1000 回発生させた．測定は，図 9 に示すように 4 か所で行った．一連の処理全体にかかる処理を(A)全体として測定した．一連の処理を，図 9 に示すように，(B)ロードバランスが必要か判断，(C)データキュー送受信，(D)mig_pri 発行処理の 3 つの部分に分割し，それぞれにかかる時間を測定した．

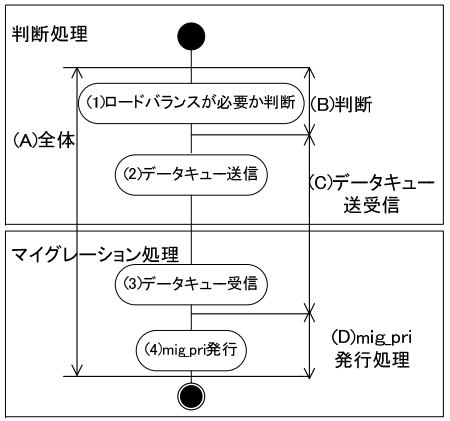

図 9　実行オーバヘッド測定箇所

測定結果を，表 3 に示す．

(A)全体の結果から，依頼方式の方が実行オーバヘッドがかかることが観測された．

(B)判断，(D)mig_pri 発行処理にかかる時間は，ほぼ同等となっている（タイマ精度は 80ns）．2 方式において異なるのは，(C)データキュー送受信の部分である．依頼方式では，前述したように，データキューを送信するプロセッサと，データキューを受信するプロセッサが異なるため，データキュー送信の API 中で，プロセッサ間割込みが発生する．それに対し，追い出し方式では，データキューを送信するプロセッサと受信するプロセッサは同一であるため，プロセッサ間割込みは発生しない．よって．(C)データキュー送受信の測定で観察される 2 方式における差は，プロセッサ間割込みによるものと言える．

表 3 で示す結果は，(B)(C)(D)の測定には測定オーバヘッドを含むため，(A)=(B)+(C)+(D)とはならない．依頼方式の方が(A)と(B)(C)(D)の和がずれているのは，プロセッサ間割込みの影響と考えられる．

以上より，依頼方式と追い出し方式の 2 方式をオーバヘッド時間で比較すると，依頼方式の方が，プロセッサ間割込みが発生する分時間がかかっていると言える．

表 3　実行オーバヘッド測定結果

	依頼方式	追い出し方式
(A)全体	4,960ns	4,480ns
(B)判断	400ns	480ns
(C)データキュー送受信	3,840ns	2,400ns
(D)mig_pri 発行	1,840ns	1,680ns
(B)+(C)+(D)	6,080ns	4,560ns

※タイマ精度：80ns

5.2 4 プロセッサによるロードバランス方式の評価

依頼方式と，追い出し方式の 2 つの方式によるロードバランスの確認と評価をするため，4 プロセッサで実験を行った．ロードバランス対象のタスクは全て実行可能状態で起動し，待ち状態に入ることはないものとする．カーネルスタート時には，あるプロセッサに突出してタスクを割り付け，プロセッサ間で負荷のばらつきが大きくなるようにし，ロードバランスによりタスクが平準化される様子を観察した．

使用したタスクは，キャッシュの影響を受けないループ処理を行う．そして，全タスクが実行可能状態で起動し，カーネルスタートとともに，全タスクが実行可能状態となる．すべてのタスクは同一優先度とし，rot_rdq を 1 ミリ秒ごとに発行することにより，優先順位を回転することで，ラウンドロビンスケジューリングを実現した．ロードバランス実行判断を行う周期ハンドラは，50 ミリ秒ごとに起動しロードバランスを行う．このように周期ハンドラは，1 ミリ秒ごとに rot_rdq を発行するためのものと，50 ミリ秒ごとにロードバランスを行うためのものの 2 つを実装した．起動時に割り付けるタスク数は，プロセッサ 1 に 11 タスク，プロセッサ 2，プロセッサ 3，プロセッサ 4 にそれぞれ 1 タスク，計 14 タスクとした．

平均 1 タスクの実行時間は，11.163 秒である．14 タスクを 4 プロセッサで実行するため，1 プロセッサあたりのタスク数は，平均 3.5 タスクとなる．よって，ロードバランスを行った場合の理想的な全タスク終了時間は，11.163×3.5=39.070 秒となる．

(1)依頼方式

依頼方式でロードバランスを行った場合の，各プロセッサに割り付けられたタスク数変化を図 10 に示す．実験開始直後のタスク数変化を図 11 に示す．

全タスク終了時間は，39.537 秒，ロードバランスによるマイグレーションは，14 回行われた．

依頼方式を 4 プロセッサで実行するマルチプロセッサ環境に適用する実験により，以下の２つの問題が観測された．

・依頼方式における問題点 1

この実験では，プロセッサ 2,プロセッサ 3,プロセッサ 4 がそれぞれ自プロセッサに割り付けられたタスク数を，タスク数平均に近づけるように一斉にロードバランスを行う．その結果，図 12 に示すように，プロセッサ 1 に対して，プロセッサ間割込みが多発する．

・依頼方式における問題点 2

複数プロセッサが，同一プロセッサに依頼した場合，図 10 の矢印，および図 12 に示すように最高負荷であったプロセッサ 1 の負荷が一時的に平均負荷より低くなる．これは，移動するタスク数を計算する際に，自プロセッサ負荷と最高負荷プロセッサとの負荷バランスのみを判断材料としているためで，他のプロセッサが最高負荷プロセッサからどれだけタスクをマイグレーションしようとしているかは，考慮していないためである．また，プロセッサ 2,3,4 の負荷が平均負荷を超えるのは，ロードバランス実行により，各プロセッサの負荷が図 11 に示す，5 タスク，3 タスク，3 タスク，3 タスクとなった時点で，式(2)により算出される移動するタスク数が 1 未満であるが，最大負荷との差が 2 タスク以上であったため，それぞれ 1 タスクずつプロセッサ 1 に対してマイグレーションを要求したためである．

Vol.2009-EMB-14 No.5
2009/7/24

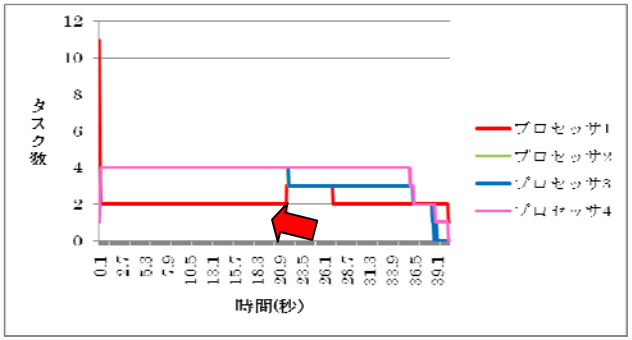

図 10　依頼方式におけるタスク数変化

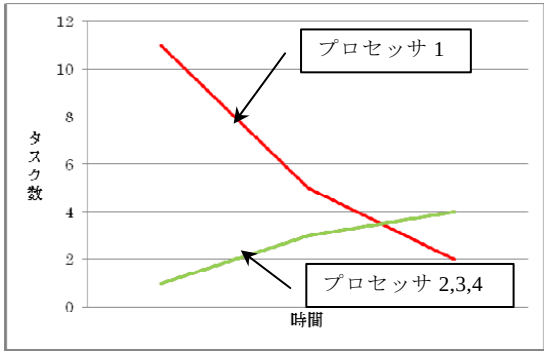

図 11 依頼方式におけるタスク数変化開始直後の様子

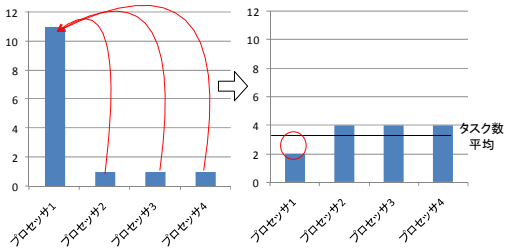

図 12　依頼方式に見られる問題点 1,2

(2)追い出し方式

追い出し方式でロードバランスを行った場合の，各プロセッサに割り付けられたタスク数変化を図 13 に示す．実験開始直後のタスク数変化を図 14 に示す．

全タスク終了時間は，39.346 秒，ロードバランスによるマイグレーションは，13 回行われた．

追い出し方式を 4 プロセッサで実行するマルチプロセッサ環境に適用する実験により，以下 2 つの問題が観測された．

・追い出し方式における問題点 1

この実験では，図 14，図 15 に示すように，プロセッサ 1 が，プロセッサ 2，プロセッサ 3，プロセッサ 4 へ順番にタスクをマイグレーションする．つまり，タスク数が平準化されるまでに，何度かマイグレーションが必要で，平準化されるまでに時間がかかる可能性がある．

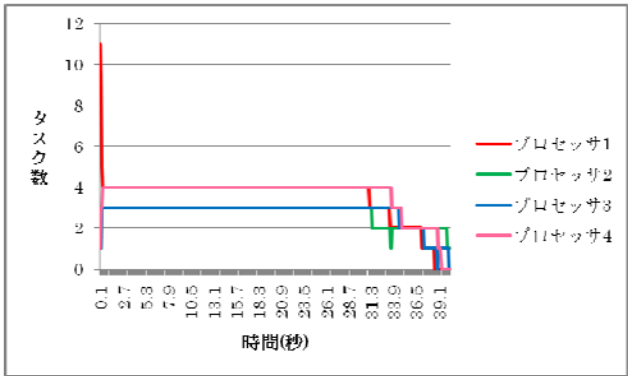

図 13　追い出し方式におけるタスク数変化

・追い出し方式における問題点 2

この実験では観察されなかったが，起動時に割り付けるタスク数を，プロセッサ 1 と 2 に 11 タスク，プロセッサ 3 に 1 タスク，プロセッサ 4 に 5 タスクとして実験を行ったところ，図 16 に示すように，最低負荷であったプロセッサ 3 の負荷が一時的に平均負荷より高くなる現象が観察された．これは，移動するタスク数を計算する際に，自プロセッサ負荷と最低負荷プロセッサとの負荷バランスのみを判断材料としているためで，他のプロセッサが最低負荷プロセッサにどれだけタスクをマイグレーションしようとしているかは，考慮していないためである．

Vol.2009-EMB-14 No.5
2009/7/24

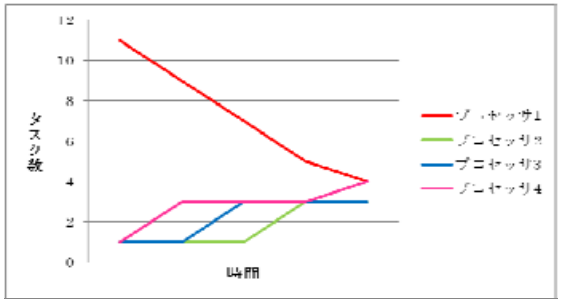

図 14　追い出し方式におけるタスク数変化開始直後の様子

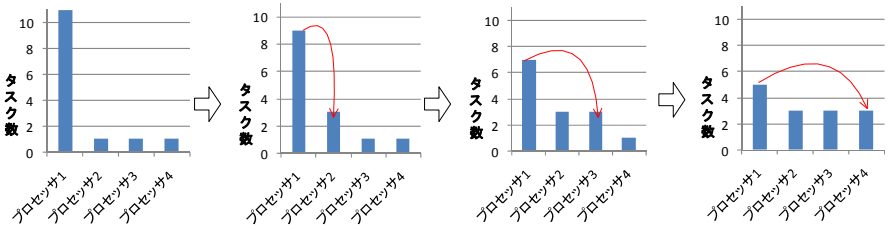

図 15　追い出し方式における問題点 1

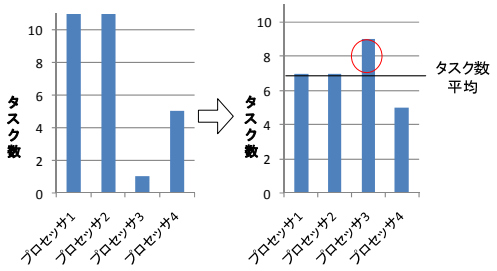

図 16　追い出し方式における問題点 2

5.3 考察

依頼方式と，追い出し方式の 2 つの方式によるロードバランスの確認と評価をするため，4 プロセッサで実験を行った結果，両方式とも，ロードバランスを行わない場合と比較して，スループットの向上が観測された．

前述したように，問題点が 4 つ観察された．そのうち依頼方式における問題点 2 と，追い出し方式における問題点 1,2 は，時間の経過とともに負荷は平準化されるので，さほど大きな問題ではない．また，一度に実行できるタスクは，各プロセッサにつき 1 つであるため，それほど迅速に負荷が平準化されなくても支障はないと判断した．

両方式のマイグレーションする際の実行オーバヘッドを比較すると，依頼方式の方がプロセッサ間割込み発生のため，実行オーバヘッドが大きい．依頼方式における問題点 1 で示したように，依頼方式ではプロセッサ間割り込みが多発する可能性がある．その際の実行オーバヘッドを考えると，TOPPERS/FMP カーネルにおいては，プロセッサ間割込みを伴わない追い出し方式が適していると判断できる．

6. おわりに

本稿では，TOPPERS/FMP カーネルが提供する API と，ロードバランス用に追加した 2 つの API を用いて，ユーザレベルでロードバランスの実現が可能であることを確認した．依頼方式と追い出し方式の 2 方式を検討及び実装し，両方式のロードバランスを行うための実行オーバヘッドを示した．また，4 つのプロセッサで実験を行い，現象を観測した．

結果として，マイグレーションを行うために，プロセッサ間割込みが発生する依頼方式に対して，追い出し方式の方が，TOPPERS/FMP カーネルには適していると判断した．

本稿では，負荷をレディキューにつながっているタスク数としてロードバランスを行った．今回の実験でタスク数が平準化されることは確認できた．

このタスク数を平準化する方式では，いったんプロセッサにタスクが割り当てられると，タスク数バランスが崩れるまで，そのまま実行を続ける．同一プロセッサに割り付けられたタスク数によって，各タスクが利用可能なプロセッサ時間は一律に定まり，各タスクの仕事の進捗度合いは公平ではない点が問題と言える．

今後の課題としては，アプリケーションを具体的に想定して，何を公平化するかを明確にして，それに応じて負荷を定めてロードバランス手法を検討する必要がある．また，自プロセッサが idle 状態になるとき，プロセスの起動時，プロセスの起床時の各タイミングでもロードバランスの適用を検討し，ライブラリ化をする予定である．

Vol.2009-EMB-14 No.5
2009/7/24

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) 深江輝昭，本田晋也，冨山宏之，高田広章: 機能分散マルチプロセッサ向け RTOS へのマイグレーション可能タスクの導入，情報処理, Vol.2007, No.27, pp.7-12.
3) TOPPERS project, http://www.toppers.jp/