社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2005－OS－100 (8)
2005／8／3

Responsive Multithreaded Processor における排他制御機構の設計と実装

中村 哲朗† 小林 秀典† 山崎 信行†

Responsive MultiThreaded Processor (RMTP) は優先度を持つ複数のハードウェアスレッドが同時に実行可能であり，ハードウェア資源の割り当てはこの優先度に従って行われる．スレッド間で排他制御を行う際，固定的な優先度に従ってスレッドを実行させるだけでは，優先度逆転問題が起きる可能性がある．そこで本論文では，優先度逆転問題を解決し，スレッド間の排他制御をソフトウェアで効率的に行う機構を設計・実装する．実装した排他制御機構を評価するために，異なる優先度を持つ複数のスレッドが共有資源にアクセスするプログラムを RMTP のシミュレータ上で実行し，各スレッドの実行終了までの時間を測定した．その結果，排他制御を行う際に高い優先度を持つスレッドがより低い優先度のスレッドによりブロックされる時間を，固定的な優先度に従って実行される排他制御機構と比較して大幅に削減することができた．

TETSURO NAKAMURA ,† HIDENORI KOBAYASHI †
and NOBUYUKI YAMASAKI †

Responsive MultiThreaded Processor (RMTP) executes multiple hardware threads with priority simultaneously. Resources of the processor are allocated based on the priority. When we execute multiple threads exclusively, threads' execution with a fixed priority may cause a priority inversion problem. To solve this problem, in this paper we designed and implemented an efficient exclusive control mechanism with software. To evaluate our mechanism, we executed the program that multiple threads with different priority access a shared resource on the RMTP simulator, and measured the execution time of each thread. Simulation results show that our mechanism decreases the duration that a higher priority thread is blocked by a lower priority thread when these threads are executed exclusively, more greatly than a exclusive control mechanism with a fixed priority.

1. はじめに

リアルタイムシステムとは，主に時間制約 (デッドライン等) を持つタスクを扱うシステムのことである．時間制約を満たすためには，各タスクに対して適切に優先順位を決定し，処理することが重要である．複数の処理が計算資源を共有し，並列または並行に処理を行なう必要がある場合，全ての時間制約を守った上で，全体として効率的に処理が進行するように競合する資源アクセスを調停することが必要である．

我々は，リアルタイムシステムにおける時間制約をハードウェアレベルから保証することを目的とした Responsive MultiThreaded Processor (以下，RMTP)[7] の研究開発を行っている．RMTP は，Simultaneous MultiThreading (SMT) と同様に，複数のハードウェアスレッドで同時実行可能な構成をとっている．複数の処理を同時に行うプログラムを実行させる際には，共有資源に対するアクセスの競合を避けるためにスレッド間で排他制御を行う必要がある．

RMTP のハードウェアスレッドは優先度を持つ．プロセッサ内のハードウェア資源の割り当てはこの優先度に従ってハードウェアにより行われる．スレッド間で排他制御を行う必要がある場合，固定的な優先度に従ってスレッドを実行させるだけでは高優先度スレッドがクリティカルセクションを実行中の低優先度スレッドにブロックされるという優先度逆転問題が生じる可能性がある．この問題が生じた場合，高優先度スレッドがブロックされる最大時間が決定できないためにデッドラインミスが生じる可能性がある．

そこで本論文では優先度逆転問題を解決し，スレッド間の排他制御を効率的に行う機構として，ブロックしている側のスレッドがブロックされた側のスレッドの最高優先度を継承する排他制御機構を RMTP 用に設計する．また，それを RMTP を対象として研究開発を行っているリアルタイムオペレーティングシステ

† 慶應義塾大学
Keio University

ム RT-Frontier 上に実装する．

本論文の構成は次の通りである．まず，第 2 章で設計及び実装対象である RMTP，RT-Frontier について述べる．第 3 章では，現在までに提案されてきたいくつかの資源共有プロトコルについて述べる．第 4 章で本論文で提案する RMTP 用の排他制御機構の設計及び実装について述べる．第 5 章で実装した機構の性能，オーバヘッドの評価及びその考察について述べ，第 6 章で結論を述べる．

2. 実装対象

2.1 Responsive Multithreaded Processor

RMTP[7] はリアルタイム処理をハードウェアレベルで支援する RMT Processing Unit (RMT PU) をプロセッシングコアに持ち，リアルタイム通信機構，コンピュータ用周辺機能，各種周辺制御機能を 1 つのチップに集積した System-on-a-Chip (SoC) である．

RMT PU では 8 つのコンテキストをプロセッサ内に保持して実行することができるが，スレッド間で演算器やキャッシュシステム等のハードウェア資源の競合が起こった場合は，RMT PU はスレッド毎に設定された優先度を基に優先度のより高いスレッドに対して先にハードウェア資源を割り当てる．正に同一クロックで同時に実行できるスレッド数は，3〜4 個である．

9 個以上のスレッドを実行する場合は，コンテキストスイッチが発生する．RMT PU ではコンテキストを格納するための専用キャッシュをオンチップに用意し，レジスタファイルとの間を広いバスで接続している．コンテキストキャッシュは 32 個のコンテキストを格納することができ，コンテキストスイッチをハードウェアにより 4 クロックサイクルで行う．これによりコンテキストスイッチにかかるオーバヘッドを大幅に削減する．

RMTP のスレッドは次の 2 つに分類される．

アクティブスレッド　レジスタファイルやプログラムカウンタなどの資源が確保され，プロセッサ内で実行可能なスレッド

キャッシュスレッド　コンテキストキャッシュ内に保持されているスレッド

RMTP が実行するスレッドはアクティブスレッドで実行状態にあるスレッドのみである．スレッドの優先度は 8bit 256 レベルで表され，値が大きいほど優先度は高くなる．

2.2 RT-Frontier

RT-Frontier[3] は，マイクロカーネルを基本としたサーバベースドアーキテクチャのリアルタイムオペレーティングシステムである．オペレーティングシステム内のコンポーネントとしては，マイクロカーネル，デバイスドライバ群，アプリケーションプログラムインタフェース (API) 及びシステムサービスを提供するためのサーバ群が存在する．その概念的な構成を図 1 に示す．

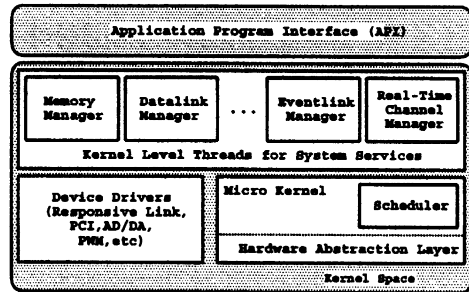

図 1 RT-Frontier の構成図

また，RT-Frontier は $1ms$ を単位時間としたスケジューリングを行うことができ，タスクの実行時間は $1\mu s$ を単位として管理することが可能である．

3. 関連研究

文献 6) は，ユニプロセッサにおける優先度逆転問題を解決する代表的な資源共有プロトコルとして，Basic Priority Inheritance Protocol を提案している．この基本的な考え方は，あるタスクがクリティカルセクションを実行中に 1 つ以上のより高い優先度を持つタスクをブロックしたとき，ブロックしているタスクは元の優先度を無視して，ブロックされている全タスクの最高優先度を継承させるというものである．そしてクリティカルセクションの実行が終了すると元の優先度に戻す．Basic Priority Inheritance Protocol は優先度逆転問題を解決するが，デッドロック，チェーンブロッキングという 2 つの問題を起こす可能性がある．デッドロックは 2 つのタスクが入れ子の形で互いが保持しているロックを獲得しようとしたときに起きる．またチェーンブロッキングとは，高優先度タスクがロックを獲得しようとするたびに，先にロックを獲得していた複数の低優先度タスクにブロックされて，結果的に高優先度タスクがブロックされる時間が長くなってしまうことである．

さらに文献 6) では，Basic Priority Inheritance Protocol に優先度シーリングという概念を追加した Priority Ceiling Protocol を提案している．優先度シーリングとは各ロックに対して設定され，その値はそのロックを利用する全タスクの最高優先度とする．あるタスクがロックを獲得しようとする時，そのタスクの優先度が，その時点でその他のタスクにより保持されている全ロックの優先度シーリングよりも高い場合のみ，ロックを獲得しクリティカルセクションへ入ることができる．これにより，Basic Priority Inheritance Protocol では防ぐことが不可能なデッドロッ

クとチェーンブロッキングを防ぐことが可能となる．しかし，あらかじめ各ロックを利用するタスクの優先度が分かっていないといけないため，ある程度アプリケーションに特化したプロトコルといえる．その点，Basic Priority Inheritance Protocol は優先度を継承させるだけなので汎用性の高いプロトコルといえる．

文献 4) 5) は，文献 6) の Priority Ceiling Protocol をマルチプロセッサ用に拡張した Multiprocessor Priority Ceiling Protocol を提案している．このプロトコルでは，資源が同一プロセッサのタスク間で共有される場合，異なるプロセッサのタスク間で共有される場合，それぞれに Priority Ceiling Protocol を適用する．

文献 1) 2) は，スピンロックに優先度継承の概念を取り入れた，Priority Inheritance Spin Lock を提案している．基本的な考え方は，Priority Inheritance Protocol と同様であり，ロックを保持しているプロセッサが，そのプロセッサによりブロックされた全プロセッサ中の最高優先度を継承するというものである．

文献 10) 9) は，RMTP においてスレッド間の排他制御と同期をハードウェアでサポートする機構を設計・実装している．この機構では，ブロックされたスレッドをアクティブのままストールさせることにより資源の浪費を防ぐ．命令がハードウェアにより実装されているため，オーバヘッドはソフトウェアでの実装と比較して小さい．しかし排他制御命令では，ブロックされたスレッドはアクティブのままであり，同時に 9 個以上のスレッドが同じロックにアクセスを試みる場合は，ユーザ側でブロックされたスレッドをキャッシュに退避したり，アクティブに戻したりしなければならないため，柔軟性に乏しい．

4. 設計と実装

本論文では，非常に短い実行区間の排他制御を対象として，スピンロックの設計と実装を行う．スピンロックはロックを利用した共有資源へのアクセスの排他制御機構であり，ロックを獲得できるまでロック獲得命令を繰り返すといった形で，資源の競合が起こるクリティカルセクションへの複数スレッドの侵入を防ぐ．しかし，この方法ではロック獲得までに費す時間が長くなる場合，その間，常に命令を実行し続けるため実行資源の浪費が大きくなってしまい他のスレッドの実行を阻害する．

また，長い実行区間の排他制御を対象として，セマフォの設計と実装を行う．セマフォは共有資源へのアクセスの際にはスピンロックを用いて排他制御を行い，もしブロックされたならそのスレッドの実行を停止させ優先度順に待ち行列に追加するという機構である．クリティカルセクション内のスレッドがクリティカルセクションから抜け出せば，待ち行列の先頭のスレッドが実行を開始しクリティカルセクションへ入ることができる．この方法ではブロックされたスレッドはその間停止しているため他スレッドの実行を阻害することはない．しかし，この方法を用いる上では待ち行列への追加，待ち行列からの削除といった際に生じるオーバヘッドを考慮に入れなければならない．

RMTP にはハードウェアレベルでの優先度が存在し，優先度のより高いスレッドに優先的にハードウェア資源が割り当てられる．したがって OS レベルの優先度を RMTP 用の優先度にマッピングする必要がある．そこで，RT-Frontier では OS レベルでの優先度を変更する際には，それに対応してハードウェアレベルでの優先度をスケジューラにより変更することにした．

4.1 スピンロック

スピンロックは通常 test_and_set, fetch_and_store, compare_and_swap といった 1 語のアトミックな read-modify-write 命令，もしくは load linked と store conditional といった 1 組でアトミックに load と store を行うことができる read-modify-write 命令によって実装される．

RMTP には 1 語のアトミックな read-modify-write 命令は存在せず，その代わりに load linked と store conditional 命令が用意されている．これらの命令は対になるように用い，もし load linked によって指定されたメモリアドレスの内容が，store conditional によってアクセスされる前に書き換えられると，store conditional は失敗し 0 を返す．もしこの 2 命令の間にプロセッサがコンテキストスイッチを起こせば，この場合も失敗する．成功すれば 1 を返す．スピンロックはこの 2 命令により排他制御を実現する．

4.1.1 優先度逆転問題

RMTP においてスピンロックを単純に用いると制御不能な優先度逆転問題，さらにはシステム全体が擬似的に実行停止状態を引き起こしてしまう．前々章で述べたように RMTP は 8 つのコンテキストをプロセッサ内に保持しているが，同時実行できるスレッド数は 3〜4 個である．ここで，RMTP が同時実行できるスレッド数を 3 と仮定する．したがって，もし 4 個以上のアクティブスレッドが同時に存在する場合，優先度が最も高い 3 スレッドに優先的にハードウェア資源が割り当てられる．これにより，例えばロックを保持しているスレッドよりも高い優先度を持つ 3 個以上のスレッドが同じロックを獲得しようとした場合，それらのスレッドはロック獲得に失敗しスピンを続けることになる．このとき，ロックを保持しているスレッドにはほとんどハードウェア資源が割り当てられず，システム全体としては実行が停止しているように見える．

また，クリティカルセクションを実行中のスレッドよりも高い優先度を持つ 1 個のスレッドがそのロック

を獲得しようとしたとする．このとき，2個以上の中優先度のスレッドが実行を開始すると，それらのスレッドの実行が終了するまで高優先度のスレッドは待たされたままスピンを続けるという制御不能な優先度逆転問題が起きる．

4.1.2 RMTP 用のスピンロック

RMTP でスピンロックを用いる場合，前述のような制御不能な優先度逆転問題を回避し，システム全体の実行の停止を起こさないことが必要である．そのためにはクリティカルセクション内のスレッドがハードウェア資源を奪われてはいけない．また，スピンロックは実行区間が短いクリティカルセクションを対象とするので，優先度継承を行なうとその手続きによるオーバヘッドの影響が大きくなってしまう．このため次のような RMTP 用のスピンロックを実装する．

- ロックを獲得したスレッドのハードウェアレベルでの優先度を OS レベルでの優先度とは無関係に RMTP で設定できる最高優先度 255 に設定する．
- クリティカルセクションの実行を終了した時点で，元の優先度に戻す．

```
 1:          GETOTID R1              ;R1 <- thread id
 2:          GETTT   R2, R1          ;R2 <- thread table
 3:          ANDI    R3, R2, 0xff    ;R3 <- my priority
 4: lockit:  LL      R2, 0(R5)       ;R2 <- lock(load linked)
 5:          ANDI    R2, 0xff        ;R2 <- lock[7:0]
 6:          BNEZ    R2, lockit      ;branch if locked
 7:          LI      R4, 0xff        ;R4 <- 0xff: highest priority
 8:          OR      R2, R3          ;R2 <- my priority
 9:          CHGPR   #0, R1, R4      ;promote my priority to highest
10:          SC      R2, 0(R5)       ;lock variable <- my priority
11:          BEQZL   R2, lockit      ;brunch if SC failed
12:          CHGPR   #0, R1, R3      ;restore my priority
```

図 2 RMTP 用のスピンロックのアセンブリコード

図 2 にアセンブリコードを示す．getotid, gettt, chgpr は RMTP 固有のスレッド制御命令である．getotid は自スレッドのスレッド ID を返す．gettt は指定したスレッド ID のスレッドテーブルを返す．chgpr は指定したスレッド ID の優先度を変更する．その他の命令は MIPS 準拠となっている．R5 はロック変数であり，ロック変数が 0 より大きければ locked であり，0 であれば unlocked となる．locked の状態ではロック変数の値として自スレッドの優先度を用いる．

1〜3 行目で自スレッドのハードウェアレベルでの優先度を求める．なおスレッドテーブルの下位 8 ビットは優先度を保持する．4〜6 行目でロック変数の下位 8 ビットを求め，unlocked であれば先に進むが，locked であればまた 4〜6 行目を繰り返す．9 行目で自スレッドの優先度を最大優先度 255 に設定する．10〜12 行目で store conditional によりロック変数に自スレッドの元の優先度を store し，失敗したら元の優先度に戻して load linked 命令に戻る．成功したらクリティカルセクションに入る．

一方，ロックを解放する際には，ロック変数に保持しておいた自スレッドの元の優先度を取り出し，ロック変数には 0 を与え，unlocked の状態にする．そして自スレッドの優先度を元に戻す．

RMTP 用のスピンロックを用いるとクリティカルセクションを実行中のスレッドに最優先にハードウェア資源が与えられるので，システム全体の実行の停止状態を回避することができる．

4.2 セマフォ

セマフォにはクリティカルセクションの実行権を獲得できなかったスレッドのための待ち行列を用意する．ブロックされたスレッドはアクティブスレッドではなくなり，その待ち行列で優先度順に待たすようにする．セマフォにアクセスする際には前述の RMTP 用のスピンロックを用いて排他制御を行う．なお，sem_down() 関数では実行権を獲得できなかった場合にブロックされることになるスレッドの待ち行列への追加を行い，逆に sem_up() 関数では待ち行列からの削除を行なう．

4.2.1 優先度逆転問題

セマフォを用いて排他制御を行う場合，ブロックされて待ち行列中にいるスレッドはアクティブスレッドではない．そのため前述のような一般的なスピンロックを用いた場合に起こり得るシステム全体の実行停止状態にはならない．しかしクリティカルセクションを実行中のスレッドよりも高い優先度のスレッドが待ち行列中に存在し，その中間の優先度のスレッドがクリティカルセクション外を実行しているとき，クリティカルセクションを実行中のスレッドはその中間の優先度のスレッドに多くのハードウェア資源を奪われる．そのような場合，クリティカルセクションの実行時間が長引き，ブロックされている高優先度のスレッドにとっては好ましくない．

図 3 は，スレッド th_1, th_2, th_3 (優先度はそれぞれ P_1, P_2, P_3 であり $P_1 > P_2 > P_3$) があるとき，th_1, th_3 がセマフォS_1 によって保護されているクリティカルセクション CS_1 にアクセスしようとして優先度逆転問題を起こす例を示している．

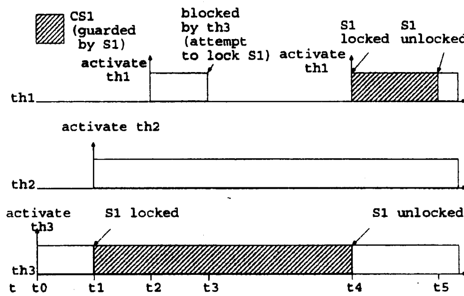

図 3 RMTP においてセマフォを用いた場合の優先度逆転問題

図 3 の例では，時刻 t ≥ t2 においてクリティカルセクションを実行中の th_3 よりも優先度が高い th_2

にハードウェア資源が多く割り当てられるため，クリティカルセクションの実行時間が長くなり，それに伴って th_1 が自分よりも低い優先度のスレッドにブロックされる時間も長引くことになる．

4.2.2 RMTP 用のセマフォ

セマフォを用いる排他制御ではクリティカルセクションの実行時間が長い場合を想定する．そのため前述の RMTP 用のスピンロックのようにクリティカルセクション内のスレッドの優先度をシステム内の全スレッドの最高優先度よりも高く設定すると，クリティカルセクション外を実行中のスレッドに与える影響が大きくなってしまう．

前述のような優先度逆転問題を解決するために，セマフォにおいてはクリティカルセクションを実行中のスレッドに待ち行列中のスレッドの最高優先度を継承させる．

```
struct sem_struct{
        volatile int counter;
        struct thread_struct *ewq;
        struct thread_struct *owner;
        unsigned int org_prio;
};
```

図 4 sem_struct 構造体

セマフォを管理する構造体 (図 4) はあらかじめ OS 内に静的に確保しておく．sem_down() では counter を 1 デクリメントし，sem_up() では counter を 1 インクリメントする．待ち行列が空で，どのスレッドもクリティカルセクションを実行していないときは counter は 1 に設定しておく．また，ewq はクリティカルセクションの実行を待つスレッドの待ち行列である．

優先度を継承するために，新たに以下のメンバを追加する．

owner はロックを保持しているスレッド管理構造体へのポインタを保持する．org_prio はロックを獲得したスレッドの元の優先度を保持しており，クリティカルセクションを実行中のスレッドが存在しない場合は 0 を設定する．

sem_down

sem_down() 関数における処理の過程を以下に示す．

(1) counter を 1 デクリメントする．

(2) counter < 0 であれば (クリティカルセクションを実行中のスレッドが存在すれば) 以下を実行する．
  (a) 優先度順に ewq に自スレッドを追加する．
  (b) org_prio に owner の優先度を保存する．
  (c) owner のスレッドの優先度と自スレッドの優先度を比較し，自スレッドの優先度の方が高ければ owner のスレッドの優先度を自スレッドの優先度に書き換える．
  (d) 自スレッドをアクティブスレッドから外す．
  (e) その後，sem_up() により activate されたら owner に自スレッドを設定し，クリティカルセクションを実行する．

(3) counter = 0 であれば owner に自スレッドを設定し，クリティカルセクションを実行する．

sem_up

sem_up() 関数における処理の過程を以下に示す．

(1) counter を 1 インクリメントする．
  (a) ewq が空ならば次に進む．
  (b) ewq が空でないならば，ewq から先頭のスレッドを取り出してアクティブスレッドにし，ewq の以降のスレッドを順次繰り上げる．

(2) owner に NULL を設定する．

(3) 自スレッドが別のスレッドの優先度を継承している場合，org_prio に保持しておいた元の値に自スレッドの優先度を設定する．

(4) org_prio に 0 を設定する．

図 3 と同様のスレッドセットに対して RMTP 用のセマフォを用いて排他制御を行い優先度逆転問題を回避した例を図 5 に示す．

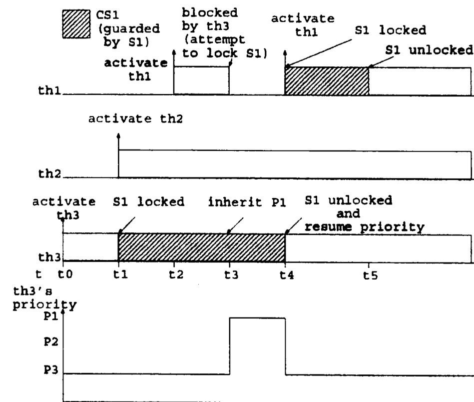

図 5 RMTP 用のセマフォによる優先度逆転問題の回避

図 5 の例では，クリティカルセクションを実行中の th_3 が時刻 t = t3 において S_1 を獲得しようとした th_1 をブロックし，P_1 を継承した時点で th_2 より優先度が高くなり，th_2 よりも多くのハードウェア資源が割り当てられる．そのためクリティカルセクションの実行時間を短縮することができ，それに伴って th_1 が自分よりも低い優先度のスレッドにブロックされる時

間も短縮できる．

4.3 まとめ

RMTP 用のスピンロックでは，クリティカルセクションを実行中のスレッドの優先度を設定できる最高優先度まで引き上げるため，実行の途中でハードウェア資源を完全に奪われることはない．そのためシステム全体の実行が停止することは無い．しかし，優先度を引き上げたスレッドよりもデッドラインが近いスレッドがクリティカルセクション外を実行中の場合，それらのスレッドに悪影響を及ぼす．さらに，ブロックされたスレッドはその間停止しているわけではなく，スピンを続けており，その他のスレッドの実行を阻害する．しかし，少ない操作で排他制御を実現することができるため，短いクリティカルセクションに対しては有効であると考えられる．

それに対して RMTP 用のセマフォでは，ブロックされたスレッドはその間アクティブではなくなるので他スレッドの実行を阻害することはない．また，クリティカルセクションを実行中のスレッドは待ち行列中のスレッドの最高優先度を継承し，次にクリティカルセクションを実行するスレッドは待ち行列中のその最高優先度スレッドである．そのため，排他制御が原因で生じる，低優先度スレッドが高優先度スレッドにブロックされる時間を削減することができる．これらにより，上記の RMTP 用のスピンロックでは効率的に保護できないような比較的実行時間の長いクリティカルセクションの保護を実現することが可能である．しかし優先度の継承，待ち行列への優先度順の追加，待ち行列からの削除といった処理を行わないといけないため，RMTP 用のスピンロックと比較してオーバヘッドが大きくなってしまうことが予想される．

5. 評価

本章では，RMTP 用のスピンロック，RMTP 用のセマフォにおけるオーバヘッドを計測し評価を行なう．また，異なる優先度を持つスレッドで資源の競合が起きた場合の，優先度継承を行わないセマフォと RMTP 用のセマフォを用いた場合の IPC の計測を行い，評価を行なう．

5.1 評価環境

本機構の実装はアセンブリ言語及び C 言語で行ない，コンパイルは RMTP のアーキテクチャ用に修正した GNU gcc，RMTP の独自命令を追加した GNU binutils を用いて行った．gcc のオプションには- O2 フラグを用いてコンパイルを行なった．コンパイラのバージョンは以下の表 1 の通りである．

RMT PU は命令供給機構が改良されたもの[12] を使用した．命令供給機構のハードウェア資源が全アクティブスレッド間で共有されることにより，全体的な性能及び，単一スレッドの実行効率が現行の RMT PU[11] よりも向上している．しかしながら，この機構を持つ RMT PU は現在チップ化作業中のため，評価は Cadence 社の NC-Verilog を用いたクロックレベルシミュレーションにより行なった．CPU クロックは 100MHz に設定した．使用した RMT PU の命令発行機構の概要を表 2 に，命令演算機構[8] の概要を表 3 にそれぞれ示す．

表 1 コンパイルに用いたツール群

ベース	バージョン
GNU gcc	3.4.0
GNU binutils	2.14
Newlib	1.12.0

表 2 命令発行ユニットの概要

アクティブスレッド数	8
キャッシュスレッド数	32
優先度の指定	256level
命令フェッチ数	8
同時命令発行数	4
同時命令完了数	4
整数レジスタ数	32bit × 32entry × 8set
整数リネームレジスタ数	32bit × 64entry
浮動小数点レジスタ数	64bit × 8entry × 8set
浮動小数点リネームレジスタ数	64bit × 64entry

表 3 命令演算ユニットの概要

整数演算器	4 + 1 (Divider)
浮動小数点演算器	2 + 1 (Divider)
64bit 整数演算器	1
整数ベクトル演算器	1 (8IU × 2 line)
浮動小数点ベクトル演算器	1 (4FPU × 2 line)
分岐ユニット	2
メモリアクセスユニット	1
同期ユニット	1

5.2 オーバヘッドの計測

スピンロック，セマフォの各関数を単一スレッドで実行させた時の実行終了までの時間をオーバヘッドとして，RMTP の 64-bit の CPU Count Register を用いて計測を行った．ただし，計測したオーバヘッドには，CPU Count Register の読み出しにかかる時間 (0.04 μsec) も含まれる．

表 4 スピンロックのオーバヘッド（単位は μsec）

spin lock	spin unlock	spin lock for RMTP	spin unlock for RMTP
0.35	0.11	0.66	0.33

表 5 スピンロックのオーバヘッド増加率

spin lock	spin unlock
89%	200%

スピンロックの場合，表 4 から全体的にオーバヘッドは小さいことが分かる．通常のスピンロックは元々

命令数が少ない．また，RMTP 用のスピンロックは，lock で引き上げる優先度は最初から決まっており，unlock では元の優先度に戻すだけでよいため追加する命令数も少ない．よって，これは当然の結果といえる．

表 5 を見るとオーバヘッドの増加率はかなり大きな値となっているが，これは通常のスピンロックのオーバヘッドが絶対的に小さいため，多少命令数を増加させただけで相対的に大きくなってしまっていることによる．

表 6 セマフォのオーバヘッド (単位は μsec)

down	up	down for RMTP	up for RMTP
1.19	1.83	1.47	1.85

表 7 セマフォのオーバヘッド増加率

down	up
24%	1%

一方，セマフォの場合は表 6 からスピンロックに比べて全体的にオーバヘッドが大きいのが分かる．これはセマフォにおいてはスピンロックによる排他制御の処理に加えて，セマフォ管理構造体の各変数の設定があるためである．

表 7 では，down, up 共に増加率はスピンロックに比べると小さい．これは，通常のセマフォの操作がスピンロックと比較して元々多いためである．また，追加した機構に伴うオーバヘッドが非常に小さく，オーバヘッドにほとんど差がないことを示す結果といえる．

5.3 RMTP 用のセマフォと通常のセマフォの比較

3 個のスレッド th_0, th_1, th_2 (優先度は $th_0 > th_1 > th_2$) において，低優先度の th_2 と高優先度の th_0 が資源の共有をセマフォにより実現することを想定したプログラムを実行した．また，th_1 は全く資源を要求しないものとした．

th_2 と th_0 は同じプログラムを用いており，どちらもリリースの直後に資源を要求する．資源の占有時間は 1 スレッド単体で実行させた場合で 200 μsec であり，資源の解放後直ちに終了する．

シミュレーションでは，OS がブートした直後に th_2, th_1, th_0 の順にスレッドを起動した．

図 6，図 7 はそれぞれ 優先度の変更を行わないセマフォ，優先度を継承するセマフォを用いた場合の各スレッドの IPC を示す．

図 6 では，シミュレーションの開始から約 471μsec 経過したところで 3 つのスレッドが順次起床しているのが分かる．このとき，th_0 よりも先に起床した th_2 が資源を先に獲得したために，th_0 はブロックされ，th_2 が資源を解放するまで IPC が 0 となっている．また，th_2 よりも優先度が高い th_1 がハードウェア資源をより多く使用するため，図 8 を見ても分かる通り，本来 200μsec で終了するはずの th_2 によるクリティカルセクションの実行時間が約 730μsec もかかってしまっている．

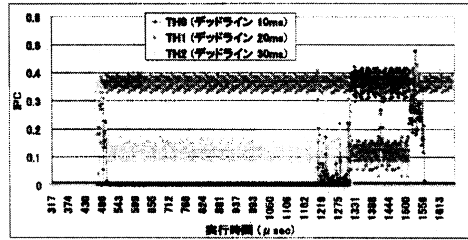

図 6 優先度継承を行わないセマフォを用いた場合の各スレッドの IPC

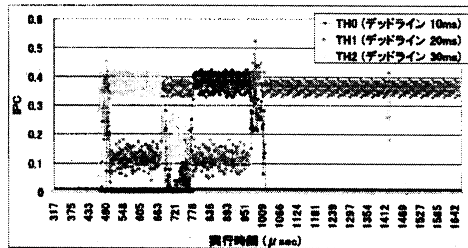

図 7 RMTP 用のセマフォを用いた場合の各スレッドの IPC

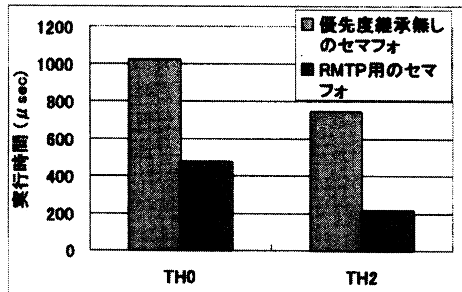

図 8 クリティカルセクションの実行完了までの時間の比較

それに対して RMTP 用のセマフォでは th_2 が th_0 の優先度を継承するため，図 7 が示すように th_1 の約 3 倍のハードウェア資源を使用することができる．これにより，図 8 を見ても分かるとおり th_2 によるクリティカルセクションの実行時間を約 220μsec まで削減することができた．それに伴い，最も高い優先度を持つ th_0 の実行完了までの時間を約 1020μsec から，約 480μsec まで削減することができた．

6. 結論と今後の課題

本研究では，RMTP において資源共有により発生する優先度逆転問題を回避するために，RMTP 用のスピンロックとセマフォを設計・実装した．

RMTP で単純にスピンロックを用いて排他制御を行う場合，共有資源へアクセスするスレッド数，またその優先度によっては，クリティカルセクションを実行中のスレッドのハードウェア資源が奪われシステム全体が停止してしまう可能性がある．この問題を解決するために，提案したスピンロックでは，スピンロックにより保護されたクリティカルセクションを実行中のスレッドに RMTP で設定可能な最高優先度を与える．このスピンロックによるオーバヘッドは小さく，クリティカルセクションの実行時間が非常に短い場合には有効であることが分かった．またこのとき，優先度を最高に設定することによる他のスレッドに対する悪影響は小さく抑えられる．

一方，セマフォを用いて資源共有を実現する場合には，資源の獲得に失敗したスレッドはブロック状態になる様にした．さらに，資源の獲得に成功したスレッドはブロックされたスレッドの最高優先度を継承することで優先度逆転問題を解決した．シミュレーションによる評価では，このセマフォを用いることでクリティカルセクションを実行中のスレッドがクリティカルセクション外のスレッドにより受ける影響を小さくし，ブロックされたより高い優先度を持つスレッドのブロッキング時間を大幅に削減することができた．しかし，スピンロックに比べてセマフォはオーバヘッドが大きいためクリティカルセクションの実行時間が短い場合には不向きであることが分かった．

今後の課題としては，ブロックされたスレッドの状態をアプリケーションにより指定できるようにすることが考えられる．本論文ではスレッドの実行コンテキストは OS のスケジューラが管理しているが，ブロックされたスレッドのコンテキストがコンテキストキャッシュに退避されるのか，メモリに書き戻されるのかをクリティカルセクションの長さ等に応じて変えることができれば，より効率のよい排他制御が可能となると考えられる．また，ハードウェア排他制御命令[10)9)] をソフトウェアにより拡張してセマフォにおける排他制御命令として用いることができれば，よりオーバヘッドの小さい排他制御機構が可能となると考えられる．

謝辞 本論文の研究は，科学技術振興機構 CREST の支援による．

参 考 文 献

1) Craig, T.: Queuing Spin Lock Algorithms to Support Timing Predictability, *Proceedings of Real-Time Systems Symposium*, pp. 148–157 (1993).

2) Johnson, T. and Harathi, K.: A Prioritized Multiprocessor Spin Lock, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, No. 9, pp. 926–933 (1997).

3) Kobayashi, H. and Yamasaki, N.: RT-Frontier: A Real-Time Operating System for Practical Imprecise Computation, *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, pp. 255–264 (2004).

4) Rajkumar, R., Sha, L. and Lehoczky, J.: Real-time Synchronization Protocols for Multiprocessors, *Proceedings of the Real-time Systems Symposium*, pp. 259–269 (1988).

5) Rajkumar, R., Sha, L. and Lehoczky, J.: Real-Time Synchronization Protocols for Shared Memory Multiprocessors, *Proceedings of the 10th International Conference on Distributed Computing Systems*, pp. 116–123 (1990).

6) Sha, L., Rajkumar, R. and Lehoczky, J.: Priority Inheritance Protocols: An Approach to Real-Time Synchronization, *IEEE Transactions on Computers*, Vol. 39, No. 9, pp. 1175–1185 (1990).

7) Yamasaki, N.: Responsive Multithreaded Processor for Distributed Real-time Systems, *Journal of Robotics and Mechatronics*, Vol.17, No.2, pp. 130–141 (2005).

8) 伊藤務, 山崎信行: Responsive Multithreaded Processor の命令実行機構, 情報処理学会論文誌: コンピューティングシステム, Vol. 44, No. SIG 11(ACS 3) (2003).

9) 村中延之, 伊藤務, 新井誠一, 山崎信行: Responsive Multithreaded Processor のスレッド間同期機構の設計と実装, 電子情報情報通信学会技術研究報告:組込技術とネットワークに関するワークショップ (ETNET2005), pp. 31–36 (2005).

10) 薄井弘之, 内山真郷, 伊藤務, 山崎信行: Responsive Multithreaded Processor の同期機構の設計と実装, 情報処理学会研究報告 2003-ARC-145, pp. 37–42 (2003).

11) 薄井弘之, 内山真郷, 伊藤務, 山崎信行: Responsive Multithreaded Processor における実時間処理用命令供給機構, 電子情報通信学会技術研究報告: 実時間処理に関するワークショップ (RTP2004), pp. 15–20 (2004).

12) 薄井弘之, 内山真郷, 伊藤務, 山崎信行: Responsive Multithreaded Processor の命令供給機構, 情報処理学会論文誌: コンピューティングシステム, Vol. 45, No. SIG 11(ACS 7) (2004).