Vol.2015-SE-188 No.8
Vol.2015-EMB-37 No.8
2015/6/5

位置透過性のあるシステムコールを有するマルチコアプロセッサ向けリアルタイムOS

石橋　航太[1,a)]　横山 幸太郎[1,†1]　兪 明連[1,b)]　横山 孝典[1,c)]

概要：本論文では，位置透過性のあるタスク管理，イベント制御，およびリソース管理が可能な非対称型マルチコアプロセッサ向けリアルタイム OS を提案する．本リアルタイム OS は，自動車分野の標準リアルタイム OS 仕様である OSEK OS を拡張したもので，単一 CPU コア上のみでなく，CPU コア間でシステムコールを発行し，タスクの起動やタスク間の同期を可能としている．また，OSEK 優先度上限プロトコルを拡張し，MSRP(Multiprocessor Stack Resource Policy) をベースとしたリソースアクセスプロトコルを採用することで，CPU コア間の排他制御を実現している．これにより，コア内だけでなく，コア間でも同一のシステムコールを用いた位置透過性のある共有リソース管理が可能になる．そして，提案したリアルタイム OS のリソース管理システムコールの実行時間を測定し，実用上問題ないことを確認した．

キーワード：リアルタイム OS，組込みシステム，並列処理，マルチコアプロセッサ

1. はじめに

組込みシステムの大規模化および複雑化にともない，処理性能の向上が求められており，マルチコアプロセッサの重要性が増している. 特に複数の CPU コア上で独立に OS を動作させる非対称型あるいは機能分散型と呼ばれるマルチコア並列処理が有用視されている．そのため，シングルコアプロセッサ環境に限らず，マルチコアプロセッサ環境も包括的にサポートすることのできる組込システム向けのリアルタイム OS が求められている．しかし，従来のリアルタイム OS の多くはシングルコアプロセッサ向けであり，マルチコアプロセッサ向けのリアルタイム OS はまだ少ない．

機能分散マルチプロセッサ向けのリアルタイム OS として，μITRON4.0 仕様 [1] を拡張した TOPPERS/FDMP カーネル [2] がある．このリアルタイム OS ではプロセッサ間でシステムコールを実行可能としており，異なるプロセッサ上のタスクの起動やタスク間の同期が可能である．ただし，TOPPERS/FDMP カーネルではタスク間の排他制御をセマフォで行っており，この方式ではデッドロックが発生する恐れがある．また，自動車分野の業界標準仕様の策定を行っている AUTOSAR が，マルチコアプロセッサ向けのリアルタイム OS の仕様を定めている [3]. AUTOSAR OS は OSEK OS 仕様 [4] を拡張したリアルタイム OS であり，タスク管理やイベント制御を行うシステムコールをコア間でも発行可能とすることで，マルチコアプロセッサ向けの仕様としている．OSEK OS ではリソースアクセスプロトコルとして，タスクの優先度を変更することでデッドロックを発生せずに排他制御を行う OSEK 優先度上限プロトコル (OSEK Priority Ceiling Protocol，OSEK PCP) を採用しており，AUTOSAR OS でも同一コア上のタスク間に関しては同じ仕様のシステムコールを規定している．一方，コア間の排他制御にはスピンロックを用いたシステムコールを規定している．コア内とコア間とで使用するシステムコールを使い分ける必要があり，位置透過性に欠ける．コア内とコア間で同一のシステムコールを用いた位置透過性のあるリソース管理が可能となるマルチコアプロセッサ向けリアルタイム OS が求められている．

これまでにマルチコアプロセッサ向けのリソースアクセスプロトコルがいくつか提案されており，代表的なものとして，マルチプロセッサ優先度上限プロトコル (Multiprocessor Priority Ceiling Protocol，MPCP)[5][6] とマルチプロセッサスタックリソースポリシー (Multiprocessor Stack Resource Policy，MSRP)[7] がある．そのうち MPCP は，シングルコア向けのリソースアクセスプロトコルである優

[1] 東京都市大学
Tokyo City University
[†1] 現在，日立製作所
Presently with Hitachi, Ltd.
[a)] g1581501@tcu.ac.jp
[b)] yoo@cs.tcu.ac.jp
[c)] yokoyama@cs.tcu.ac.jp

先度上限プロトコル (Priority Ceiling Protocol，PCP)[8] をマルチコア向けに拡張したものであり，コア間で共有するリソースを獲得したタスクの優先度を変更することで排他制御を行っている．また，MSRP はスタックリソースポリシー (Stack Resource Policy，SRP)[9] を拡張したプロトコルで，コア間ではタスクの優先度を変更するとともにスピンロックを用いる．

しかし，これまでのところマルチコア向けのリソースアクセスプロトコルを実装した組込みシステム向けのリアルタイム OS はまだ発表されていない．

本論文では，マルチコアプロセッサ環境において，CPU コア間でシステムコールを発行し，タスクの起動やタスク間の同期が可能であるとともに，位置透過性のあるリソース管理が可能な非対称型マルチコアプロセッサ向けリアルタイム OS を提案する．マルチコアプロセッサ向けの機能のうち，タスク管理とイベント制御については既に報告している [10] ので，本論文ではリソース管理を中心に述べる．

本論文の構成は以下の通りである．まず 2 章で提案するリアルタイム OS の仕様について述べ，3 章でリソースアクセスプロトコルの詳細について述べる．4 章では本リアルタイム OS の実装について述べる．そして，5 章で実装したリアルタイム OS の性能評価を行い，6 章で本論文のまとめと今後の課題について述べる．

2. リアルタイム OS の仕様

2.1 拡張対象システムコール

本研究では OSEK OS 仕様に基づくリアルタイム OS である TOPPERS/ATK1[11] を対象に，同一コア上のタスクのみでなく，マルチコアプロセッサにおける他コア上のタスクも同一のシステムコールを用いて管理できるようにする．また，同一コア内のタスク間で共有するリソースのみでなく，異なるコア上のタスク間で共有するリソースも同一のシステムコールで管理できるようにする．以下，本論文では CPU コア内で共有するリソースをローカルリソース，コア間で共有するリソースをグローバルリソースと呼称する．

OSEK OS が提供するシステムコール (システムサービス) には，タスク管理，イベント制御，アラーム機能，リソース管理，割り込み制御，OS の実行制御がある．このうち，コア間での管理や制御が必要と考えられるタスク管理，イベント制御，リソース管理のシステムコールを拡張の対象とする．表 1 は OSEK OS 仕様で規定されているタスク管理，イベント制御，リソース管理のシステムコールを示す．6 つのタスク管理のためのシステムコールのうち，引数で対象タスクを指定するのは ActivateTask()，ChainTask()，GetTaskState() の 3 つである．また，4 つのイベント制御のためのシステムコールのうち，引数で対象タスクを指定する必要があるのは SetEvent()，GetEvent() の 2 つである．また，リソース管理のための 2 つのシステムコール GetResource()，ReleaseResource() はいずれも拡張対象とする．

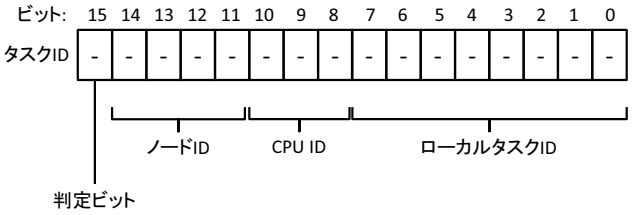

図 1 タスク ID

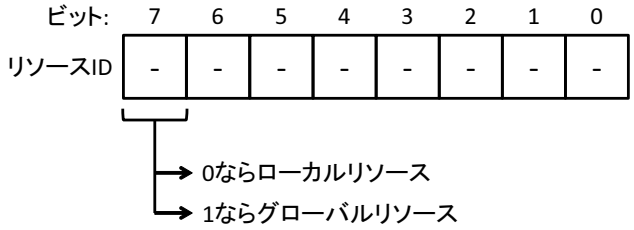

図 2 リソース ID

2.2 タスク ID とリソース ID

システム全体でタスクを一意に指定可能とするため全コアで共通のタスク ID を定義する．複数のノードをネットワーク接続した分散システムを対象とした分散リアルタイム OS[12] との互換性を保つため，タスク ID は 16 ビットとし，ノードのみでなく CPU コアを識別できるようにする．すなわち，図 **1** に示すように，上位 1 バイトでノードおよび CPU コアを，下位 1 バイトで OSEK OS と同一のタスク ID を表現する．上位 1 バイトのうち，最上位ビットはマルチコアかシングルコアを判別するもので，1 のときマルチコア，0 のときシングルコアとする．マルチコアの場合は，残り 7 ビットのうち上位 4 ビットでノード ID を，下位 3 ビットで CPU コアを表す．シングルコアの場合は 7 ビット全てでノード ID を表す．

また，TOPPERS/ATK1 ではリソースの判別を 8 ビットから構成されるリソース ID で行っている．リソースはそれぞれ固有のリソース ID を割り付けられ，その値をもとに管理されている．そこで本研究で開発する OS は，リソース ID の最上位ビットの値をもとにそのリソースがローカルリソースなのかグローバルリソースなのかを判断する．具体的には図 **2** に示すように，最上位ビットが 0 ならその ID の割り付けられたリソースはローカルリソースであり，最上位ビットが 1 ならグローバルリソースであると判断する．この ID 情報をもとにリソース獲得，解放等の管理を行う．

表 1　拡張対象システムコール

分類	機能	システムコール名 (引数)	タスク指定	拡張対象
タスク管理	タスク起動	ActivateTask(Task)	○	○
	自タスク終了	TerminateTask()	×	×
	自タスク終了およびタスク起動	ChainTask(Task)	○	○
	高優先度タスク実行	Schedule()	×	×
	タスク ID 取得	GetTaskID(TaskRef)	×	×
	タスク状態取得	GetTaskState(Task, StateRef)	○	○
イベント制御	イベント設定	SetEvent(Task, Event)	○	○
	イベントクリア	ClearEvent(Event)	×	×
	イベント状態取得	GetEvent(Task, Event)	○	○
	イベント待ち	WaitEvent(Event)	×	×
リソース管理	リソース獲得	GetResource(Resource)	-	○
	リソース解放	ReleaseResource(Resource)	-	○

2.3　拡張 OIL

OSEK OS では，アプリケーションで用いるタスク，イベント，リソース等の設定（コンフィギュレーション）のために OIL（OSEK Implementation Language）[13] と呼ばれる専用言語を用いる．OIL によりタスクの宣言やイベントの設定などを記述し，SG（System Generator）に通すことでアプリケーション依存の構成データを記述したソースコードを出力する．本研究では，マルチコア向けリアルタイム OS 向けに 1 つのファイルに複数の CPU コアの設定を記述可能とするとともに，CPU コア間で共有するリソースを宣言できるように OIL を拡張する．

拡張 OIL の記述例を**図 3** に示す．図 3 は，マルチコアプロセッサ multi1 の設定情報を記述した拡張 OIL の例である．CPU の設定情報の中に複数のコアに関する記述とともにコア間で共有するリソースの設定情報の記述が可能となっている．

```
OIL_VERSION = "2.5";
IMPLEMENTATION Standard {
. . . . . .
}
MULTI multi1 {
  CPU cpu11 {
    . . . . . .
    TASK task11 {
      . . . . . .
      RESOURCE = glbres1;
      RESOURCE = res11;
      . . . . . .
    };
    TASK task12 { . . . . . };
    RESOURCE res11 { . . . . . };
    . . . . . .
  };
  CPU cpu12 {
    . . . . . .
    TASK task21 {
      . . . . . .
      RESOURCE = glbres1;
      RESOURCE = res21;
      . . . . . .
    };
    TASK task22 { . . . . . };
    RESOURCE res21 { . . . . . };
    . . . . . .
  };
  RESOURCE glbres1 {
    . . . . . .
  };
  . . . . . .
};
```

図 3　拡張 OIL の記述例

3.　リソースアクセスプロトコル

3.1　概要

OSEK OS で規定されている OSEK PCP と呼ばれるリソースアクセスプロトコルは，SRP をもとに固定優先度スケジューリングに特化して，実装を簡略化したものである．そこで，本論文で提案する OS のリソースアクセスプロトコルは，SRP をマルチコアプロセッサ向けに拡張した MSRP をベースとする．ただし，MSRP はタスクのみを対象にしているのに対し，OSEK OS ではタスクだけでなく，割込み処理（Interrupt Service Routine，ISR）もリソースを利用することが可能であるため，ISR を考慮した優先度変更を行うように MSRP を拡張する．以下，これを拡張 MSRP と呼ぶことにする．なお，OSEK OS では，OS の管理外であるカテゴリ 1 の ISR と OS が介在するカテゴリ 2 の ISR の 2 種類が存在する．リソースを利用可能なのはカテゴリ 2 の ISR のみなので，本論文で述べる ISR とはカテゴリ 2 の ISR を指すものとする．

3.2　ローカルリソース管理

コア内のリソース管理は，OSEK PCP と同一である．すなわち，ローカルリソースを利用するタスクまたは ISR の中で最高の優先度をそのローカルリソースの上限優先度とし，ローカルリソースを獲得したタスクまたは ISR の優先度をそのローカルリソースの上限優先度に変更することで排他制御を行う．ローカルリソース解放後にはタスクまたは ISR の優先度を元に戻す．コア内でローカルリソースを共有する際の動作例を**図 4** に示す．タスク 1 とタスク 2 はローカルリソースを共有しており，タスク 3 はリソース

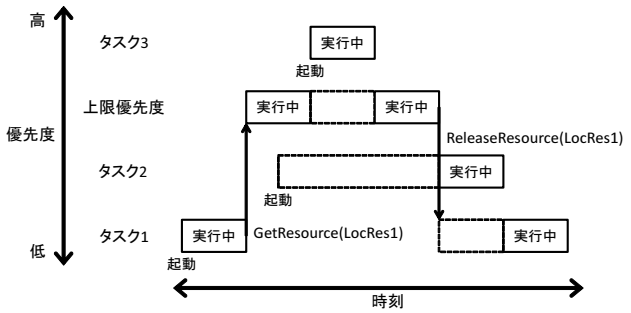

図 4　ローカルリソースを共有する場合の例

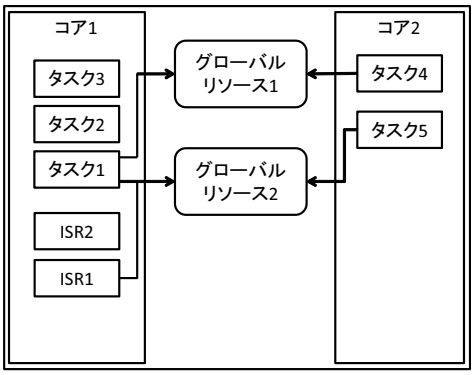

図 5　タスク及びリソースの構成例

を共有しないものとする．また，この図 4 ではタスクの位置が上にあるほど優先度が高いものとする．流れとしてはまず，タスク 1 がローカルリソース 1 を獲得することで優先度が上がり，タスク 2 よりも優先的に実行されることとなる．次にタスク 1 実行中にタスク 3 が起動され，タスク 3 はローカルリソース 1 を共有しておらず，排他制御の必要がないので，タスク 1 よりも優先的に実行される．最後にタスク 1 がローカルリソース 1 を解放した後は，最初の優先度でタスクが実行される。

3.3　グローバルリソース管理

コア間のリソース管理は拡張 MSRP により行う．MSRP では複数のグローバルリソースをネストして要求することはできないため，拡張 MSRP も同じ仕様とする．タスクがグローバルリソースを獲得する際，そのグローバルリソースがタスクのみから利用される場合は，MSRP に従い，グローバルリソースを獲得するタスクの優先度をコア内のタスクの中で最高の優先度にする．一方，タスクのみでなく ISR からも利用される場合は，グローバルリソースを獲得するタスクまたは ISR の優先度をそのコア内のグローバルリソースを共有する ISR の中で最高の優先度にする．

グローバルリソースにアクセスする具体的な動作について例を用いて説明をする．この例のタスク，ISR およびリソースの構成を**図 5** に示す．コア 1 上のタスク 1 とコア 2 上のタスク 4 がグローバルリソース 1 を共有し，コア 1 上のタスク 1 と ISR1 およびコア 2 上のタスク 5 がグローバルリソース 2 を共有している．

まず，異なるコア上のタスク間でコア間でリソースを共有する場合の動作例を**図 6** を用いて説明する．なお，図 6 では上の 2 つのタスクはコア 1 上，下の 2 つのタスクはコア 2 上のタスクであり，それぞれのコアにおいてタスクの位置が上にあるほど優先度が高いものとする．流れとしては，まずタスク 1 がグローバルリソース 1 を獲得し，優先度がコア 1 内のタスクの中で最高の優先度となる．次にタスク 4 がグローバルリソース 1 の獲得要求を出すが，この時点ではまだタスク 1 がグローバルリソース 1 を占有しているので，スピンロックでグローバルリソース 1 の解放を待つ．タスク 1 がグローバルリソース 1 を解放すると優先度は元に戻り，コア 1 内のタスクは最初の優先度で実行される．また，タスク 1 がグローバルリソース 1 を解放するとタスク 4 はスピンロックを解除し，グローバルリソース 1 を獲得する．タスク 4 がグローバルリソース 1 を用いた処理を終え，グローバルリソース 1 を解放すると優先度は元に戻り，コア 2 内においても最初の優先度でタスクが実行される．

次にタスクと ISR の両者がリソースを共有する場合の例を**図 7** を用いて説明する．タスク 1，タスク 3 と ISR2 はコア 1 に割り付けられたタスクと ISR であり，優先度は上に位置するほど高いものとする．グローバルリソース 1 はコア 1 ではタスクからのみ利用されるため，タスク 1 がグローバルリソース 1 を獲得するとタスク 1 の優先度は，コア 1 内のタスクの中で最高の優先度となる．それに対しグローバルリソース 2 はコア 1 では ISR からも利用されるグローバルリソースであるので，タスク 1 がグローバルリソース 2 を獲得すると，タスク 1 の優先度はコア 1 内のグローバルリソースを共有する ISR の中で最高の優先度となる．なお，ISR2 はグローバルリソース 2 を共有しないので，タスク 1 がグローバルリソース 2 を占有中でも，優先的に実行される．

4.　実装

4.1　対象アーキテクチャ

本研究で対象とするマルチコアプロセッサは，どの CPU コアからも利用できる共有メモリを有するものとする．また，コア間の共有リソースの排他制御をサポートするため，Test-and-Set 命令等の，リソース状態の確認と変更をアトミックな操作で行うことができる機構が必要である．

本研究で実装に使用する評価ボード M3A-HS50 の構成を**図 8** に示す．M3A-HS50 は 2 つの SH2A CPU コアから構成されるデュアルコアプロセッサ SH7205 を搭載している．それぞれの CPU がローカルメモリとして利用できる内蔵 RAM のほか，全ての CPU が利用できる共有 RAM が組み込まれている．また，CPU 間の共有リソースの排他制御をサポートするセマフォレジスタが組み込まれてい

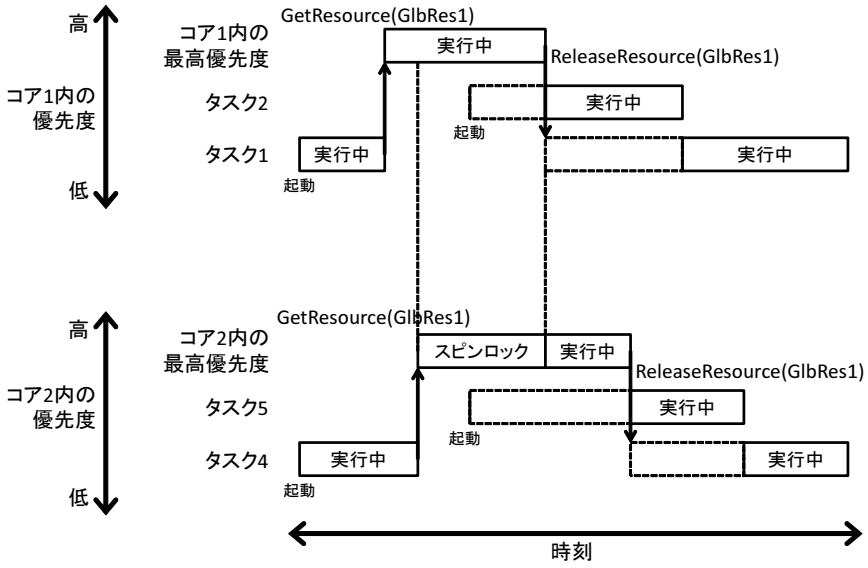

図 6　タスクがグローバルリソースを共有する場合の動作例

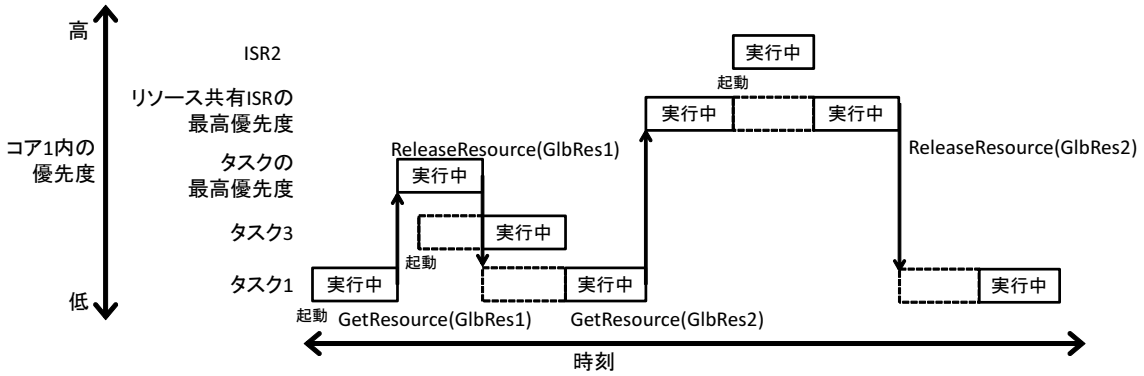

図 7　タスクと ISR がグローバルリソースを共有する場合の優先度変更

る．また，M3A-HS50 には外部メモリとしてプログラムコードを格納するフラッシュ ROM が搭載されている．

4.2　コア間遠隔システムコール

コア間のタスク管理やイベント制御はコア間遠隔システムコールにより実現する．本 OS で，コア間遠隔システムコールの動作を図 **9** に示す．

アプリケーションタスクがシステムコールを発行すると，タスク位置判定処理が，システムコールの対象タスクがどのコア上にあるか判定する．対象タスクが他コア上にある場合，メモリ経由要求送信処理がコア間要求メッセージを生成し，共有メモリ上のメッセージバッファに格納する．コア間要求メッセージは，要求元 CPU ID，要求先 CPU ID，システムコール発行元タスク ID，対象タスク ID，システムコールの種類，システムコールの引数で構成される．次に対象タスクのあるコアに対して割込みを発行し，ビジーウェイトで返答を待つ．

割込みを受けたコアでは，メモリ経由要求受信処理が共

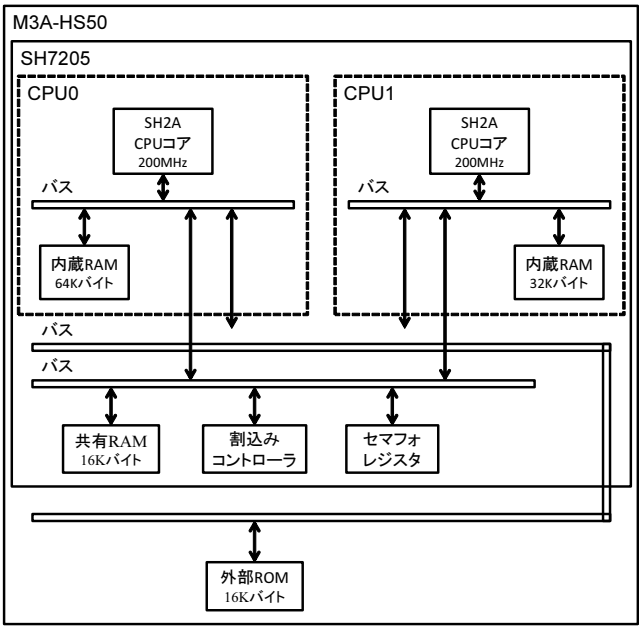

図 8　評価ボード構成図

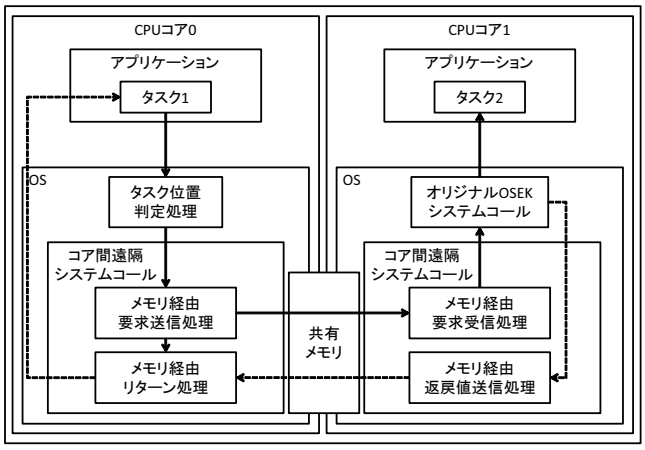

図 9 コア間システムコールの動作

有メモリ上のコア間要求メッセージを解析し，指定されたシステムサービスを実行する．システムコールを実行した後，メモリ経由返戻値送信処理が返戻値を含むコア間返答メッセージを共有メモリに格納する．要求側コアでは返答待ちをしていたタスクがメモリ経由リターン処理を行って返答メッセージを解析し，返戻値を受け取り，アプリケーションの処理に戻る．

4.3 スピンロック

本研究では，スピンロックの実現に SH7205 プロセッサが提供するセマフォレジスタを利用する．セマフォレジスタは，CPU 間の排他制御をサポートする機構で，格納された値が 1 ならそのレジスタに対応するリソースは空き状態，0 なら占有状態と判断する．セマフォレジスタは値を読み出すと自動的に 0 にクリアされるため，リソース状態の確認と変更をアトミックな操作で行うことができる．

4.4 構成データ

本 OS の構成データには，TOPPERS/ATK1 がもともと必要としていた構成データに加え，本 OS で追加したマルチコア並列処理向けの構成データがある．これらの構成データを生成する SG は現在開発中であるため，現時点では，OIL 記述を参照して手作業で構成データのソースファイルを記述している．

5. 評価

リソースの獲得・解放を行うシステムコールを使用した際の本 OS の実行時間を計測し，実用上問題のない性能か評価する．タスク管理およびイベント制御のシステムコール処理時間についてはすでに報告済み [10] なので，本論文ではリソース管理のシステムコールの処理時間について報告する．

本論文で提案した OS と TOPPERS/ATK1 において，それぞれリソースの獲得をするシステムコール GetResource() の実行時間と，解放をするシステムコール ReleaseResource() の実行時間をリソースがローカルリソースとグローバルリソースの 2 つの場合に分けて測定した．実行時間の測定は 100 回行い，その平均値で評価を行った．

表 2 グローバルリソース利用時の測定結果

対象 OS	GetResource()[μsec]			ReleaseResource()[μsec]		
	なし	タスク	ISR	なし	タスク	ISR
開発 OS	1.15	1.18	1.24	1.15	1.15	1.21

表 3 ローカルリソース利用時の測定結果

対象 OS	GetResource()[μsec]			ReleaseResource()[μsec]		
	なし	タスク	ISR	なし	タスク	ISR
開発 OS	0.93	0.97	1.03	1.06	1.06	1.12
ATK1	0.90	0.93	1.00	1.03	1.03	1.06

グローバルリソースの場合の GetResource(), ReleaseResource() の実行時間を表 2 に, 開発 OS と TOPPERS/ATK1 においてローカルリソースの場合の GetResource()，ReleaseResource() の実行時間を測定した結果を表 3 に示す．実行時間はぞれぞれ優先度変更がない場合，タスクレベルで優先度変更がある場合，ISR レベルで優先度変更がある場合の値を計測した．

開発 OS においてグローバルリソースを利用する際はセマフォレジスタへのアクセス処理などが追加されているため，ローカルリソースを利用する際に比べてシステムサービス GetResource() の実行時間が最大約 24 %，システムサービス ReleaseResource() の実行時間が最大約 8 %増大しているが実用上問題になるほどの増大ではないと考えている．また，開発 OS ではローカルリソースを利用する際は利用するリソースがグローバルリソースかローカルリソースかの判定処理などが追加されているため，TOPPER/ATK1 と比較して，システムサービス GetResource() の実行時間は最大約 4 %，システムサービス ReleaseResource() の実行時間は最大約 6 %増大している．しかし，どの場合も増大率は 10 %未満に収まっているので，これらの実行時間も実用上問題のあるものではないと考える．

本リソースアクセスプロトコルは，スピンロックを用いているため，クリティカルセクションが大きくなると効率が落ちる．対象としている自動車制御等のアプリケーションでは，クリティカルセクションが想定以上に大きくなることはないと考えるが，タスクを待ち状態に遷移させる方がよいアプリケーションへの対応については今後の課題である．

6. おわりに

CPU コア間でシステムコールを発行し，タスクの起動やタスク間の同期が可能であるとともに，位置透過性のあるリソース管理が可能な，非対称型マルチコアプロセッサ

向けリアルタイム OS を提案した．そして，実装及び評価を行い，実用上問題ない性能であることを確認した．

また現在，提案したマルチコアコアプロセッサ向けのリアルタイム OS に対応した構成データの生成が可能な SG を開発中である．今後の課題としては，スピンロックでなくタスクを待ち状態に遷移させるリソースアクセスプロトコルへの対応が考えられる．

我々はまた，マルチコア並列処理と分散処理が混在する環境に対応したリアルタイム OS と分散共有メモリ機能を有する分散リアルタイム OS も開発している [12][14]．本論文で提案したリアルタイム OS とこれらの OS を統合化することで，単一 CPU による処理，マルチコア並列処理，および分散処理を意識せずに組込み制御システムを開発可能なリアルタイム OS を実現したいと考えている．

謝辞

本研究で使用した TOPPERS/ATK1 の開発者に感謝する．本研究の一部は JSPS 科研費 24500046 および 15K00084 の助成を受けたものである．

参考文献

[1] 坂村健監修，高田広章編:μ ITRON4.0 仕様 Ver.4.02.00 トロン協会 (2004)．
[2] 本田晋也，高田広章:ITRON 仕様 OS の機能分散マルチプロセッサ拡張，電子情報通信学会論文誌，Vol.J91-D，No.4，pp.934-944 (2008)．
[3] AUTOSAR: *Specification of Multi-Core OS Architecture V1.1.0 R4.0 Rev 2* (2010).
[4] OSEK/VDX: *Operating System, Version 2.2.3* (2005).
[5] Ragunathan Rajkumar, Lui Sha, John P. Lehoczky: Real-Time Synchronization Protocols for Multiprocessors, *Proceedings of Real-Time Systems Symposium 1988.*, pp.259–269 (1988).
[6] Paolo Gai, Giuseppe Lipari, Alberto Ferrari, Claudio Gabellini, Paolo Marceca: A Comparison of MPCP and MSRP When Sharing Resources in the Janus Multiple-Processor on a Chip Platform, *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp.189–198 (2003).
[7] Paolo Gai, Giuseppe Lipari, Marco Di Natale: Minimizing Memory Utilization of Real-Time Task Sets in Single and Multi-Processor Systems-on-a-chip, *Proceedings of the 2001 IEEE Real-Time Systems Symposium*, pp.78–83 (2001).
[8] Lui Sha, Ragunathan Rajkumar, John P. Lehoczky, Priority Inheritance Protocols: an Approach to Real-Time Synchronization, *IEEE Transactions on Computers, Vol.39 , Issue 9*, pp.1175–1185 (1990).
[9] Theodore P. Baker: Stack-Based Scheduling of Realtime Processes, *Real-Time Systems, Vol.3, Issue 1*, pp.67–99 (1991).
[10] 横山幸太郎, 齊藤政典, 兪明連, 横山孝典, マルチコア並列処理・分散処理統合機能を有するリアルタイム OS, 第 13 回情報科学技術フォーラム, 第一分冊, pp.239–240 (2014).
[11] TOPPERS/OSEK カーネル，TOPPERS/ATK1: http://www.toppers.jp/atk1.html.
[12] 知場貴洋，齊藤政典，伊丹悠一，兪明連，横山孝典：位置等価性のあるシステムコールを有する組み込み制御システム向け分散リアルタイムＯＳ，情報処理学会論文誌，Vol.53，No.12，pp.2702–2714 (2012)．
[13] OSEK VDX, *System Generation OIL: OSEK Implementation Language Version 2.5* (2004).
[14] Chiba, T., Yoo, M., Yokoyama, T.: A Distributed Real-Time Operating System with Distributed Shared Memory for Embedded Control Systems, *Proceedings of the IEEE 11th International Conference on Dependable, Autonomic and Secure Computing (DASC),* pp.248–255 (2013).