Vol.2013-OS-124 No.3
2013/2/28

マルチコア向けFIFO式プリエンプティブ同期プロトコルのスケジューラビリティ評価

藤谷 隆宏[1,a)]　松原 豊[1]　加藤 真平[1]　高田 広章[1]

概要：マルチコアリアルタイムシステムでは，システムの時間制約を維持するため，コア間のリソース共有の最大ブロック時間を導出できることが重要となる．これまで様々なマルチコア向け同期プロトコルが提案されてきているが，その多くが高優先度タスクのスケジューラビリティの低い，ノンプリエンプティブ方式を採用したものであり，同時に複数のロックを取得するネストをサポートしていないものであった．本論文では，ネストをサポートし，プリエンプティブなスピンを採用した，高優先度タスクのスケジューラビリティの改善するプロトコルを提案した．さらに，静的優先度パーティショニングスケジューリング方式の際の，タスクの最大ブロック時間計算式を導出し，タスクの最悪応答時間計算式を導出した．評価の結果，従来の方式よりも，高優先度タスクの最悪応答時間が最大で 13%，スケジューラビリティが最大 6.5%改善されたことを示す．

キーワード：リアルタイムシステム，同期プロトコル，マルチコア，最悪応答時間

1. はじめに

近年，大規模化・複雑化が進む組込みシステムにおいても，マルチコアシステムの重要性が増している．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，コア数を増やしたほうが有利であるという状況がある．特に，複数のコアを 1 つの LSI 上に集積したマルチコアプロセッサは，処理性能面からも消費電力面からも利点が大きく，広範囲な組込みシステムへの適用が期待される．組込みシステムの多くは，最悪応答時間が予測できること，高速な割込み応答が可能である，といったリアルタイム性を保証することが重要となる．このような性質を持ったシステムをリアルタイムシステムと呼ぶ．

マルチコア向けリアルタイムシステムでは，I/O デバイスや，タスク情報，レディーキューなどのリソースを，異なるコア上で動作する複数のタスク間で共有するため，リソースの排他制御を行う必要がある．この排他制御を実現するために，ロックと呼ばれるリソースに関連付けられオブジェクトを取得したタスクのみが，クリティカルセクションの処理を実行できるようにする手法が用いられる場合が多い．クリティカルセクションの実行が終了したタスクは，ロックを解放し，他のタスクが処理を実行できる．

シングルコアにおいては，優先度上限プロトコルや，優先度継承プロトコルが用いられているが，マルチコアにおいては，複数のコアで実行されるタスクでリソース共有を行うため，シングルコア環境では発生しないリモートブロッキングが発生する．そのため，シングルコア向けプロトコルをそのまま利用することは難しい．リモートブロックキングとは，あるタスクが，既に他のコアのタスクが取得しているロックを要求することによって発生するブロックを指す．反対に，同コア内の低優先度タスクによるブロックをローカルブロッキングと呼ぶ．一般に，リアルタイムシステムではクリティカルセクションが短いことから，ロックを取得できたかどうかを繰返し確認するスピンと呼ばれる方式でブロックが実現されている．しかし，既に提案されているマルチコア向け同期プロトコルの多くが，高優先度タスクに実行が切り替わらないノンプリエンプティブ方式が採用されているため，スピン時間が伸びた場合，高優先度タスクのスケジューラビリティが悪化する可能性があると考えられる．さらに，我々がこれまでに開発してきたマルチコア向けリアルタイム OS である TOPPERS/FDMP[7]，FMP カーネル [8] では，多くの API が複数個のロックを同時に取得するネストを行う．このように，多くのリアルタイムシステムではリソースのネストが行われているが，デッドロックの危険や，最大ブロック時間の導出が困難で

[1] 名古屋大学大学院情報科学研究科
Graduate School of Information Science, Nagoya University
[a)] fujitani@ertl.jp

あることから，ネストをサポートしている同期プロトコルは少ない．

本論文では，高優先度タスクのスケジューラビリティの改善を目的とした，ネストをサポートする，プリエンプティブ方式のスピンを採用した同期プロトコルを提案する．リアルタイムシステムで頻繁に用いられる静的優先度パーティショニングスケジューリングを対象としており，その際のタスクの最大ブロック時間の計算式を導出することにより，スケジューラビリティを保証しやすくなる．マルチコア向けリアルタイム同期プロトコルの内，プリエンプティブ方式を採用した際の最大ブロック時間計算式を導出しているものは現在知られていないため，本論文は有用であると言える．また，既存のプロトコルに本手法を適用し，評価した結果，高優先度タスクの最悪応答時間が最大13%改善され，高優先度タスクのスケジューラビリティが6.5%改善された．本手法は，高優先度なハードリアルタイムタスクと低優先度なソフトリアルタイムタスクが混在するマルチコアリアルタイムシステムに有効であると言える．

本論文の構成を以下に示す．2 章では，既存のマルチコア向けリアルタイム同期プロトコルについて述べ，3 章では本論文で扱うシステムのモデル化を行う．4 章で，提案する同期プロトコルについて述べ，5 章では本手法の評価について述べ，6 章でまとめと今後の課題について述べる．

2. 関連研究

マルチコア向けリアルタイム同期プロトコルとしては，D-PCP[6] や M-PCP[5] が知られており，これらは静的優先度パーティショニングスケジューリングアルゴリズムに適用することができる．

D-PCP では，コア c_1 のあるタスクが他のコア c_2 のリソースにアクセスする場合，コア c_2 内のローカルエージェントと呼ばれる別のタスクが，代わりにリソースにアクセスする．リソース要求を迅速に完了させるため，ローカルエージェントは通常より高い優先度となる．D-PCP ではタスクが直接リソースにアクセスしないため，共有メモリを使用しない分散システムに適している．M-PCP ではリソースは特定のコアに配置されず，どのコアのタスクからも直接アクセスすることができ，リソースへのアクセスがブロックされたタスクは，要求が満たされるまでサスペンドされる．しかし，どちらのプロトコルも，デッドロックを回避するためリソースのネストをサポートしていないという問題がある．

Block らは 2007 年に，ネストをサポートし，動的優先度なパーティショニング・グローバルスケジューリング両方に適用可能なマルチコア向けリアルタイム同期プロトコル，FMLP(Flexible Multiprocessor Locking Protocol) [2] を提案している．FMLP では，クリティカルセクションの短いものをショートリソース，長いものをロングリソースとし，リソースの同期にはそれぞれ FIFO キューとセマフォが用いられている．タスクがブロックされた場合は，ショートリソースの場合はスピンで，ロングリソースの場合はサスペンドしてロックの取得を待つ．スピンは，リソースのロックを取得できるまでコアを専有し続けるという特徴を持つため，一旦ロックを取得したタスクは，早期にロックを解放するよう実装する必要があり，通常はロック取得中のタスクのプリエンプトが行われないノンプリエンプティブ方式で実装される．そのため，FMLP でもスピンはノンプリエンプティブ方式で実装されている．

さらに，M-PCP，D-PCP では制限されていたリソース要求のネストを実現するために，FMLP ではリソースグループを導入している．リソースグループとは，ネストする可能性のある複数のリソースを 1 つにまとめたものであり，タスクがグループ内のリソースにアクセスするためには，グループに関連付けられたグループロックを取得する必要がある．グループロックを取得したタスクはグループ内のリソースに自由にアクセスすることができる．どのリソースがどのグループに所属するかは，アプリケーション開発者が自由に決めることができるが，ショート・ロングリソース両方を含んだグループを作ることはできない．グループロックを複数個取得するネストも可能であるが，デッドロックを回避するため，ショートグループロックを取得した後にロンググループロックを取得するようなネストは禁止されている．グループロックは粒度の大きなロックと考えることができるため，本論文では，ロックの最小単位をグループロックとする．

Brandenburg らは，FMLP を静的優先度パーティショニングスケジューリング用に改良し，LITMUS[4] と呼ばれる Linux をマルチコアリアルタイムシステム研究用に拡張した試験用プラットフォームに実装・評価し，さらにタスクの最大ブロック時間の計算式を導出した [3]．Brandenburg らは，サスペンドから復帰したタスクは優先度をシステム内で最大値まで上昇させ，他のタスクよりも優先的にクリティカルセクションを実行できるようにしている．優先度上昇しているタスクが複数個存在する場合は，FIFO 順にスケジューリングされる．本論文では，静的優先度パーティショニングスケジューリングを対象としているため，以降 FMLP は論文 [3] で提案されているプロトコルを指すものとする．FMLP は，ネストをサポートし，静的優先度パーティショニングスケジューリングに適用可能なプロトコルであるが，ノンプリエンプティブ方式のスピンを採用しており，高優先度タスクのスケジューラビリティが悪いという問題がある．

PPIQL[9] は，マルチコアリアルタイムシステム向けに提案された，プリエンプティブな優先度継承キューイングスピン同期プロトコルである．PPIQL では，2 段階のネストをサポートし．最大ブロック時間がコア数のリニアオー

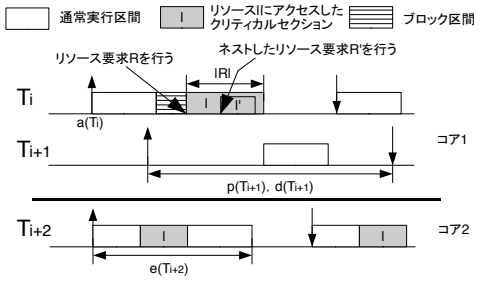

図 1　タスクのパラメータ

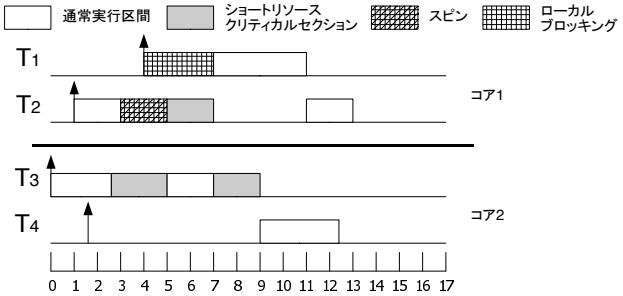

図 2　ノンプリエンプティブ方式のアルゴリズムを用いた場合の例

ダで抑えられるといった特徴があるが，実際の最大ブロック時間や最悪応答時間の計算式を導出されておらず，スケジューラビリティ解析による評価が行えないという問題がある．

3. システムモデル

本論文では，m 個のコアと n 個の周期タスクから構成されるタスクセットを対象としており，タスク T_i の起動時刻 $a(T_i)$ や相対デッドライン時刻 $d(T_i)$，周期 $p(T_i)$，クリティカルセクションを含む最大実行時間 $e(T_i)$ が定義され，相対デッドライン時刻 $d(T_i)$ と周期 $p(T_i)$ は等しいものとする．T_i の優先度は i とし，値の小さいもの程高優先度である．タスク T_i の使用率 $U(T_i)$ は $e(T_i)/p(T_i)$ で表され，システム使用率 U_{sys} は $U_{sys} = \sum_{j=1}^{i} U(T_j)/m$ で定義される．さらに，タスク間には共有リソースがあり，1 回のリソース要求を R と定義し，R について，アクセスするリソース l，クリティカルセクションの長さ $|R|$，ネストする場合は，ネストするリソース要求 R' を定義する．スケジューリング方式は，リアルタイムシステムで多く採用されている静的優先度パーティショニングスケジューリング方式を対象とする．図 1 は，タスクのパラメータを図示したものである．

4. 提案手法

本章では，提案する同期プロトコルについて述べ，タスクの最大ブロック時間を導出することにより，最悪応答時間計算式の導出とスケジューラビリティ解析を可能にする．

4.1 プリエンプティブ方式のスピン

本論文では，FMLP をベースにプリエンプティブ方式のスピンを採用した同期プロトコルを提案する．FMLP では，2 章で述べたとおり，ショートリソースを要求した際のブロックがスピンで実現されている．そこで，ショートリソースのロックの取得・解放のアルゴリズムに改良を加え，プリエンプティブ方式のスピンを実現する．

図 2 に，ノンプリエンプティブ方式の場合の，2 つのコアに 4 つのタスクが割り当てられ，1 つのショートリソースを共有している例を示す．タスク T_1，T_2 はコア 1，タスク T_3，T_4 はコア 2 に割り当てられてている．時刻 2.5 において，T_3 がショートリソースを要求し，ロックを取得する．時刻 3 において T_2 がショートリソースを要求するが，ロックを取得できないため，待ちキューにならび，スピンしてロックが渡されるのを待つ．続いて，時刻 4 において T_1 が起動するが，T_2 がスピンしているため実行できず，ブロックされる．その後，時刻 5 において T_3 がロックを T_2 へ渡し，T_2 が時刻 7 においてクリティカルセクションを終了するまで T_1 は実行することができず，T_1 は時刻 11 において全ての処理を完了する．Algorithm1 は，FMLP

Algorithm get/release locks algorithm using preemptive spin

```
 1: Ti requires a lock g of a resource l
 2: if other tasks have g then
 3:     put Ti in a wait queue q of a resource l
 4:     while don't hold a lock g do
 5:         if a high-priority task arrives then
 6:             remove Ti from a wait queue q
 7:             goto line 1
 8:         end if
 9:     end while
10: else
11:     hold a lock g
12:     execute its critical section
13:     release a lock g
14:     if some tasks exist in a wait queue q then
15:         remove a head task q
16:         goto line 12
17:     end if
18: end if
```

ショートリソースのロック取得・解放時のスピンをプリエンプティブ方式にした際のアルゴリズムである．タスクがロックを取得できなかった場合スピンしてロックが渡されるのを待つが，その間に高優先度タスクが起動した場合，そのタスクに処理を切り替えることでプリエンプティブ方式を実現する (4〜9 行目)．図 3 に，図 2 と同じタスクセットに対し，プリエンプティブ方式のスピンを採用した場合の例を示す．図 2 同様，時刻 3 において T_2 がブロックされ，スピンするが，時刻 4 においてより高い優先度を持つ T_1 が起動するため，T_2 はショートリソースの待ちキュー

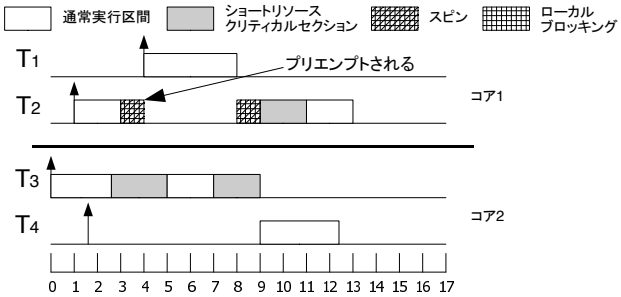

図 3　プリエンプティブ方式のアルゴリズムを用いた場合の例

から外され，T_1 にプリエンプトされる．その結果，T_1 は時刻 8 で処理を完了させることができるので，ノンプリエンプティブ方式の場合よりも応答性が良くなる．

4.2　最大ブロック時間計算式の導出

プリエンプティブ方式を導入することにより，タスクの最大ブロック時間が変わるため，本論文では FMLP での最大ブロック時間計算式をベースに，スピンがプリエンプティブ方式の場合の，タスクの最大ブロック時間計算式を導出する．

FMLP では，タスクのブロック時間は以下の 5 つの要素からなり，タスク T_i のそれぞれの最大時間を $\mathrm{BB}(T_i)$, $\mathrm{AB}(T_i)$, $\mathrm{SB}(T_i)$, $\mathrm{LB}(T_i)$, $\mathrm{DB}(T_i)$ とする．

BB(Boost Blocking)　低優先度タスクがロングリソースにアクセスし，優先度上昇しているために発生するローカルブロッキング

AB(Arrival Blocking)　ローカルな低優先度タスクがショートリソースにアクセスし，ノンプリエンプティブにスピンしているために発生するローカルブロッキング

SB(Short Blocking)　リモートタスクがアクセスしているショートリソースを要求した時に発生するリモートブロッキング

LB(Long Blocking)　リモートタスクがアクセスしているロングリソースを要求した時に発生するリモートブロッキング

DB(Deferral Blocking)　高優先度タスクがサスペンドしたことにより低優先度タスクが受ける遅延

これらの内，スピンをプリエンプティブ方式にすることにより変化する，$\mathrm{AB}(T_i)$, $\mathrm{LB}(T_i)$, $\mathrm{SB}(T_i)$ の修正を行う．スピンをプリエンプティブ方式で行うことで，高優先度タスクがローカルブロッキング時間が短くなるので $\mathrm{AB}(T_i)$, $\mathrm{LB}(T_i)$ はは短くなる．それに対して $\mathrm{SB}(T_i)$ は，プリエンプトされるタスクの最大ブロック回数が増えることを考慮すると，既存の式よりも長くなる．

ここで，最大ブロック計算式の改良で用いる記号群について述べる．

$tsk(R)$　リソース要求 R を行うタスク
$C(T_i)$　T_i が割当てられているコア
$partition(c)$　コア c に割り当てられているタスクの集合
$competing(R, c)$　コア c 内のリソース要求の内，R が要求するリソースを要求するリソース要求の集合
$narr(T_i)$　タスク T_i が周期内に起動，またはサスペンドする最大回数．
$SR(T_i)$　タスク T_i で行うショートリソース要求の集合
$res(R)$　リソース要求 R で要求されるリソース
$G(l)$　リソース l のグループ

4.2.1　最大 AB 時間計算式の導出

従来の最大 **AB** 時間計算式について

AB は，ローカルな低優先度タスクがショートリソースのクリティカルセクションを実行しているために，高優先度タスクが起動しても実行できず，ローカルブロックされている時間であり，図 2 の時刻 4〜7 において，T_1 で発生しているブロックのことである．スピンがノンプリエンプティブ方式である場合，$\mathrm{AB}(T_i)$ は低優先度タスクのショートリソース要求時のクリティカルセクションとスピン時間の和を用いて計算するため，各タスクのショートリソース要求 R ごとの最大スピン時間 $spin(R)$ を計算し，$\mathrm{AB}(T_i)$ を計算する．タスクのショートリソース要求が即座に満足されなかった場合，待ち FIFO キューに並びスピンしてロックを待つ．ノンプリエンプティブ方式のスピンであるため，最大で $m-1$ 個のタスクがキューに並んでいる可能性があることから，$spin(R)$ は下式で表すことができる．

$$spin(R) = \sum_{\substack{c=1 \\ c \neq C(tsk(R))}}^{m} \max(\{|R_c| | R_c \in competing(R, c)\}) \tag{1}$$

T_i が他のローカルな低優先度タスクからブロックされる回数は，T_i の実行中に，他のローカルな低優先度タスクがショートリソースにアクセスする回数 $|abr(T_i)|$ と，T_i が起動，またはサスペンドから復帰する回数 $narr(T_i)$ の，どちらか小さい方であるので，$a = narr(T_i)$, $s = |abr(T_i)|$ とすると，$\min(a, s)$ となり，$\mathrm{AB}(T_i)$ は下式で表すことができる．

$$\mathrm{AB}(T_i) = \sum_{l=1}^{\min(a,s)} |R_l| + |spin(R_l)| \tag{2}$$

となる．$abr(T_i)$ は，T_i の実行中に，他のローカルな低優先度タスクが行うショートリソース要求の集合であり，R_l は $wcsx(T_x, T_i)$ 内の各要素をクリティカルセクション $|R|$ と $spin(R)$ の和のの昇順に並び替えた l 番目のリソース要求である．

プリエンプティブ方式への改良

スピンをプリエンプティブ方式にした場合，図 3 の時刻

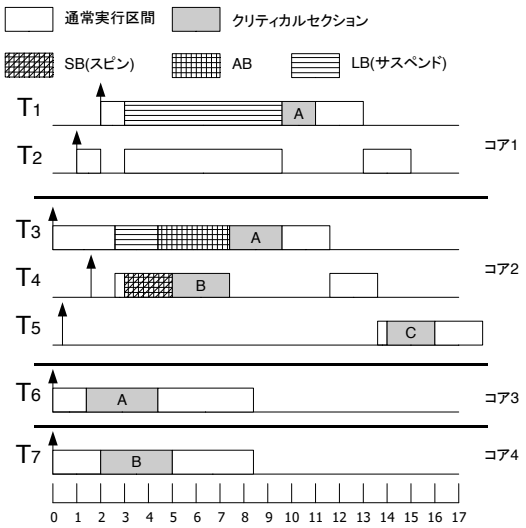

図 4　TRAB が発生する例

4 の T_1 のように．スピンによって高優先度タスクがブロックされることはない．それゆえ，各ショートリソース要求毎の最大スピン時間 $spin(R)$ をブロック時間として考慮する必要はなく，クリティカルセクション長 $|R_l|$ のみを用いて計算することができるため，AB(T_i) は下式のように修正される．

$$\mathrm{AB}(T_i) = \sum_{l=1}^{\min(a,s)} |R_l| \qquad (3)$$

4.2.2　最大 LB 時間計算式の導出

従来の最大 LB 時間計算式について

LB は，タスクが，リモートなタスクがロックを取得しているロングリソースを要求する時に発生するブロックである．LB は Direct Blocking，Transitive Remote Boost Blocking(TRBB)，Transitive Remote Arrival Blocking(TRAB) の 3 つに分類することができる．LB による最大ブロック時間 LB(T_i) は Direct Blocking と TRBB の最大ブロック時間 $rbl(T_i)$ と TRAB の最大ブロック時間 $rbs(T_i)$ から導出できる．ここでは，本手法による計算式の改善に直接関係のある TRAB の説明のみ行う．

TRAB は，タスクが要求するリソースのロックを取得しているリモートタスクが，同コア内の低優先度タスクに AB されていることにより発生するブロックである． 図 4 は，ロングリソース A とショートリソース B を 4 つのコアに割当てられた 5 つのタスクで共有している例を示している．時刻 2.5 においてロングリソース A を要求するが，T_4 によってブロックされていた T_2 は，時刻 4.5 において T_4 が解放したロックを取得する．しかし，T_3 がスピンしているため，起動できずローカルブロッキング (AB) される．したがって，時刻 3 でロングリソース A を要求した T_1 は，T_2 がロックを解放する時刻 9.5 までブロックされ続ける．このように，リモートタスクがクリティカルセクションを実行している時間に加え，時刻 4.5〜7.5 のようにロックを取得しているタスクが AB されることで発生するブロックのことを TRAB と呼ぶ．

TRAB の最大ブロック時間 $rbs(T_i)$ の計算式は，各コアからの T_i への最大ブロック時間を解析することで導出できる．コア c から受ける最大ブロック時間を $rbsp(T_i, c)$ とすると，$rbs(T_i)$ は下式で表すことができる．

$$rbs(T_i) = \sum_{\substack{l=1 \\ c \neq C(T_i)}}^{m} rbsp(T_i, c) \qquad (4)$$

T_i はコア c 内の 1 つのタスクから最大 1 回ブロックされるため，T_i の実行中にコア c 内の全タスクが要求するショートリソース要求の集合を $wcsp(T_i, c)$ と定義し，$b = ndbp(T_i, c)$，$s = |wcsp(T_i, c)|$ とすると，$rbsp(T_i, c)$ は下式で表すことができる．

$$rbsp(T_i, c) = \sum_{l=1}^{\min(b,s)} |R_l| + spin(R_l) \qquad (5)$$

ここで，$ndbp(T_i, c)$ は T_i がコア c のタスクからブロックされる最大回数であり，R_l は $wcsp(T_x, T_i)$ 内の各要素をクリティカルセクション $|R_l|$ とスピン時間 $spin(R_l)$ の和の昇順に並び替えた l 番目のリソース要求である．

プリエンプティブ方式の計算式の導出

スピンをプリエンプティブ方式にすることにより，$rbsp(T_i, c)$ で最大スピン時間 $spin(R)$ を考慮する必要がなくなるため，$rbsp(T_i, c)$ を以下のように修正され，LB(T_i) が短くなる．

$$rbsp(T_i, c) = \sum_{l=1}^{\min(b,s)} |R_l| \qquad (6)$$

4.2.3　最大 SB 時間計算式の導出

従来の最大 SB 時間計算式について

SB はタスクがショートリソースを要求することにより発生するリモートブロックである．ショートリソースの待ちキューは FIFO キューであるため，T_i はショートリソース要求を行う度に各コアからそれぞれ 1 回ずつ，最大で $m-1$ 回ブロックされる．s をコア c 内のタスクが T_i の実行中に行うリソースグループ g へのリソース要求の回数 $|wcspg(T_i, c, g)|$，b を T_i のリソースグループ g のリソースの要求回数とすると，コア c のタスクからの最大ブロック時間 $sbgp(T_i, g, c)$ は，下式で表すことができる．

$$sbgp(T_i, g, c) = \sum_{l=1}^{\min(b,s)} |R_l| \qquad (7)$$

ここで，R_l は $wcspg(T_i, c, g)$ をクリティカルセクションの昇順に並び替えた l 番目のリソース要求である．したがって，T_i がグループ g のリソースを要求した時に，全てのコアから受けるブロック時間の和を $sbg(T_i, g)$ とすると，SB(T_i) は下式で表すことができる．

$$\mathrm{SB}(T_i) = \sum_{g \in \{\mathrm{G}(R) | R \in \mathrm{SR}(T_i)\}} sbg(T_i, g) \quad (8)$$

プリエンプティブ方式への改良

ノンプリエンプティブ方式の場合はショートリソース要求 1 回に対し最大 $m-1$ 回のブロックが発生するが，プリエンプティブ方式の場合は，スピン中にプリエンプトされた場合，一旦待ちキューから外すため，1 つのコアから複数回ブロックされる可能性がある．図 2 では，T_2 は T_3 に時刻 3 で 1 回のみブロックされるが，図 3 では，T_2 は T_3 に，時刻 3 と時刻 8 の 2 回ブロックされている．タスクは，ショートリソース要求を行う時と，プリエンプトされ高優先度タスクの処理が終了し，再びタスクに処理が戻ってくる時にブロックされる可能性があるため，コア c のタスクが要求しているリソースを，T_i が要求する回数を $b(T_i, c)$，高優先度タスクにプリエンプトされる最大回数を $preempt(T_n)$ とすると，それぞれ下式で表すことができ，T_i がコア c のタスクにブロックされる最大回数は $b(T_i, c) + preempt(T_i)$ となる．

$$b(T_i, c) = \bigcup_{R \in SR(T_i)} \{R | C(tsk(R')) = c \wedge G(res(R')) = G(res(R))\} \quad (9)$$

$$preempt(T_i) = \max\left(\left\lceil \frac{p(T_i)}{p(T_x)} \right\rceil \middle| T_x \in partition(\mathrm{C}(T_i)) \wedge x < i \right) \quad (10)$$

また，T_i の実行中に，コア c のタスクで要求される可能性のあるリソース要求の内，T_i でも要求されるリソースへのリソース要求の回数 s を $s = |wcspx(T_i, c)|$ とすると，T_i がコア c のタスクにブロックされる最大回数は $min(b(T_i, c) + preempt(T_i), s)$ と表すことができる．したがって，T_i がコア c のタスクによって引き起こされる最大ブロック時間 $sbp(T_i, c)$ は下式で表すことができる．

$$sbp(T_i, c) = \sum_{l=1}^{\min(b(T_i,c)+preempt(T_i),s)} |R_l| \quad (11)$$

R_l は $wcspx(T_i, c)$ 内の各要素をクリティカルセクションの昇順に並び替えた l 番目のリソース要求である．以上より，$\mathrm{SB}(T_i)$ は下式のように修正される．

$$\mathrm{SB}(T_i) = \sum_{\substack{c=1 \\ c \neq \mathrm{C}(tsk(R))}}^{m} sbp(T_i, c) \quad (12)$$

4.3 スケジューラビリティ解析式

タスクの最大ブロック時間が計算可能になったことにより，タスクの最悪応答時間を求めることができる．本論文では，優先度を周期の短い順に割り付けるレートモノトニックスケジューリング (RMS) を用いているため，RMS の CPU 使用率によるスケジューラビリティ解析の式

$$\sum_{j=1}^{n} \frac{e(T_i)}{p(T_i)} \leq n(2^{1/n} - 1) \quad (13)$$

を用いることができる．しかし，上式は十分条件式でありタスクがスケジューリング可能な CPU 使用率の上限値が保守的な結果となるため，本論文では，タスクごとの最悪応答時間を求め，各タスクが時間制約を満たすかどうかを計算することで，スケジューラビリティ解析を行う．リソース共有のない場合のタスク T_i の最悪応答時間 $W(T_i)$ は下式で求めることができ，$W^{(n+1)}(T_i) = W^{(n)}(T_i)$ となった時の $W^{(n+1)}(T_i)$ が，タスク T_i の最悪応答時間である．

$$W^{n+1}(T_i) = \sum_{\substack{x<i \\ C(T_x)=C(T_i)}} e(T_i) \left\lceil \frac{W^n(T_i)}{p(T_x)} \right\rceil \quad (14)$$

ここで，$W^0(T_i)$ は $W^0(T_i) = e(T_i)$ とする．本論文ではリソース共有を行うシステムを対象としているため，式 (15) に，タスク間のリソース共有も考慮して拡張したものを示す．

$$W^{n+1}(T_i) = e(T_i) + \mathrm{AB}(T_i) + \mathrm{BB}(T_i) + \mathrm{SB}(T_i) + \mathrm{LB}(T_i) + \sum_{\substack{x<i \\ C(T_x)=C(T_i)}} hpt(T_x) \quad (15)$$

$$hpt(T_x) = \left\lceil \frac{W^n(T_i)}{p(T_x)} \right\rceil (e(T_x) + \mathrm{SB}(T_x) + \mathrm{LB}(T_x)) \quad (16)$$

$W^0(T_i)$ は $W^0(T_i) = e(T_i) + \mathrm{AB}(T_i) + \mathrm{BB}(T_i) + \mathrm{SB}(T_i) + \mathrm{LB}(T_i)$ とする．式 (15) における $e(T_i) + \mathrm{AB}(T_i) + \mathrm{BB}(T_i) + \mathrm{SB}(T_i) + \mathrm{LB}(T_i)$ は，高優先度タスクにプリエンプトされない場合の T_i の最悪応答時間の計算式である．$\mathrm{AB}(T_i) + \mathrm{BB}(T_i) + \mathrm{SB}(T_i) + \mathrm{LB}(T_i)$ は．リソース要求時の最大ブロック時間を表している．ここで，$\mathrm{DB}(T_i)$ は高優先度タスクのサスペンドによる低優先度タスクの遅延であり，最悪応答時間を計算する場合は，後に説明する高優先度タスクによるプリエンプト時間 $hpt(T_x)$ において $\mathrm{LB}(T_i)$ を計算しているため，考慮する必要はない．$hpt(T_x)$ は，高優先度タスク T_x によるプリエンプト時間である．$\mathrm{LB}(T_i)$ は，T_i がサスペンドする時間であるが，サスペンドしている間必ずタスクが実行されるとは限らないため，考慮する必要がある．$\mathrm{AB}(T_i)$，$\mathrm{BB}(T_i)$ はローカルブロッキングであり，高優先度タスクがローカルブロッキングされている間，低優先度タスクが実行していることになるため，考慮する必要はない．

5. 評価

本章では，提案手法の評価を行うため，静的優先度パーティショニングスケジューリングに提案する同期プロトコルを適用し，各タスクの最悪応答時間の改善率と高優先度

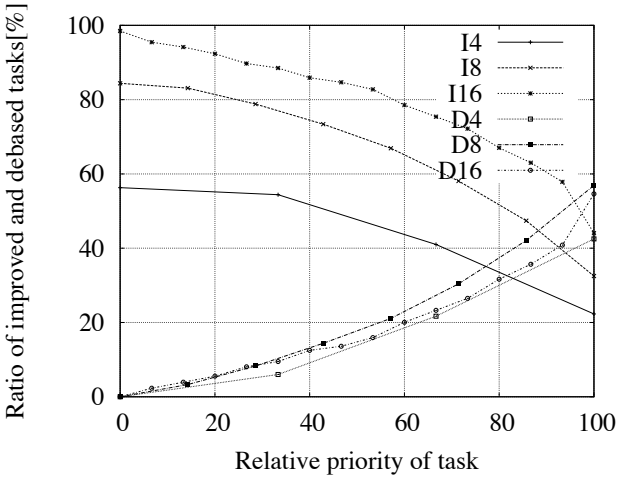

図 5　最悪応答時間が改善するタスクセットの割合

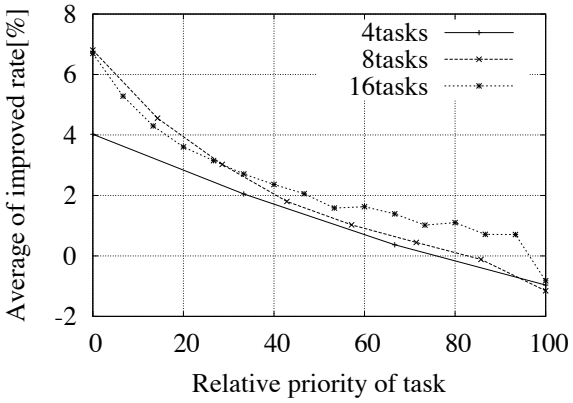

図 6　平均改善率

タスクのスケジューラビリティの評価を行う．

5.1　タスクの最悪応答時間改善率

本評価で用いるタスクセットは以下の方法で作成する．コア数 m は 2，タスクセット内のタスク数 N を $\{4, 8, 16\}$ 個の 3 パターンとし，各タスクの使用率は $[0.01, 0.1]$ としてタスクセットを生成する．タスクは WorstFitDecreasing アルゴリズムを用いて各コアに割り当てる．文献 [1] を参考に，各タスクの実行時間 $e(T_i)$ は平均 1000 の指数分布に従い，$[100, 10000]$ の範囲でランダムに作成する．リソースは，クリティカルセクションが $[10.0, 20.0]$ と $[25.0, 30.0]$ の 2 種類用意する．それぞれショートリソース，ロングリソースとして，ショートリソースは N 種類，ロングリソースは $N/2$ 種類作成する．実際のアプリケーションでは，実行時間の短いタスクは，実行時間の長いタスクに比べリソース要求回数が少ない．そのため，タスクごとに最大で 4 回リソース要求を行うように，$L_{max} = \lceil e(T_i)/2500 \rceil$ 個のショートリソースを割当てロングリソースはタスクセット内のタスクをランダムに N 個選んで割当てる．

タスクセットの各タスクの最大ブロック時間を，本手法適用前と後の最大ブロック時間計算式を用いて計算し，式 (15) を用いて最悪応答時間を計算する．タスクセットは上記のパラメータを用いてタスク数毎に 1000 個ずつ作成し，最悪応答時間が改善するタスクセットの割合，最悪応答時間が改悪するタスクセット率，平均改善率を計測した．改善率とは，1 – [改良後の最悪応答時間]/[改善前の最悪応答時間] を指す．なお，作成したタスクセットの内，最悪応答時間が周期の 2 倍を超えるタスクを含むタスクセットについては，評価を行わず無視するものとする．

図 5 は，作成したタスクセットの内，各タスクの最悪応答時間が改善，改悪するタスクの割合を示している．X 軸は，タスクの相対優先度を表し，数字が低いほど高優先度タスクを指し，I4 は 4 タスクのタスクセットでの改善したタスクの割合，D16 は 16 タスクでの改悪したタスクの割合を示している．優先度の高いタスクほど低優先度タスクをプリエンプトする可能性が高いため，AB(T_i) や LB(T_i) が改良された効果が現れ，最悪応答時間の改善率が高い．逆に，低優先度なタスクはスピンしている低優先度なタスクが少ないため，改善率が低い．最低優先度タスクには，AB(T_i) を改良した効果は現れないが，LB(T_i) による最大ブロック時間の改善がみられるため，最悪応答時間が改善されるタスクセットも少なくない．最高優先度タスクの最悪応答時間が改善されるタスクセットは 16 タスクの場合は全体の 98.5%，4 タスクの場合でも 56.3%であり，最低優先度タスクの最悪応答時間は，16 タスクセットの場合は 44.0%，4 タスクの場合も 22.3%のタスクセットで改善が見られた．

最悪応答時間の改悪は，ショートリソースを要求する際の最大ブロック回数が，プリエンプトにより多くなることが原因で発生する．これは SB(T_i) を修正したことによる影響である．最高優先度タスクはプリエンプトされることがないため，改悪するタスクセットはない．優先度が低くなるにつれてプリエンプトされる回数が増えるため，改悪される確率が高くなる．

図 6 は，各タスクの平均改善率である．タスク数が多いほど，高優先度タスクの改善率は上がっており，16 タスクの最高優先度タスクの平均改善度は 6.7%であり，タスクセットによっては最大で 13%改善される場合もみられた．低優先度タスクは改善度が低い，または改悪される傾向にあるが，優先度順上位 75%以上のタスクの最悪応答時間はいずれのタスク数の場合も改善されている．

5.2　高優先度タスクのスケジューラビリティ改善率

本論文では，高優先度タスクのスケジューラビリティ改善を目的としているため，優先度順上位 50%のタスクと全てのタスクのスケジューラビリティを計測した．

スケジューラビリティ解析は以下の方法で行われる．

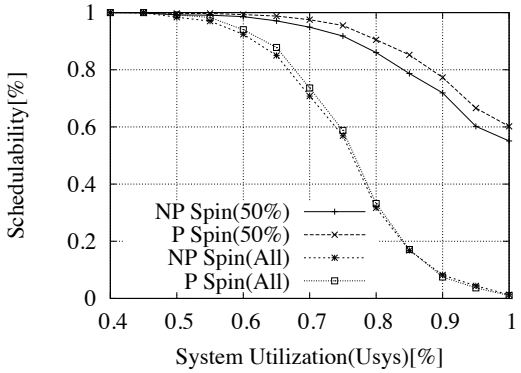

図 7　優先度順上位 50%と全タスクのスケジューラビリティ

タスクセットから，システム使用率が U_{sys} になるまで WorstFitDecreasing でタスクを 1 つずつコアに割当て，U_{sys} を超えた時点で最後に割当てたタスクをコアから外し，上で示した上位 50%と全てのタスクのスケジューラビリティを計測する．タスクセットは 5.1 節とほぼ同じ方法で作成するが，タスク使用率の範囲を $[0.01, 0.4]$ とする．5.1 節では，最悪応答時間が周期の 2 倍を超えるタスクセットは無視していたが，本評価では，その制限は設けないものとする．

図 7 は，割当てられたタスクの内，優先度順に上位 50%のタスクと全タスクのスケジューラビリティを示している．X 軸にシステム使用率の上限 U_{sys} を表し，Y 軸がスケジューラビリティを表す．U_{sys} が上がるにつれ，スケジューラビリティは悪くなるが，上位 50%のタスクのスケジューラビリティは，既存の FMLP よりも常に良い値となっており，最大で 6.5%の改善が見られる．

一方，全タスクのスケジューラビリティは，システム使用率 U_{sys} が 0.85 を超えると，ノンプリエンプティブ方式の方がよいスケジューラビリティを示している．しかし，それ以下の場合はプリエンプティブ方式の方が良いスケジューラビリティとなっており，最大で 2.8%のスケジューラビリティの改善が見られる．

提案したプロトコルでは，スピンをプリエンプティブ方式にすることで，高優先度タスクの応答性は良くなるが，低優先度タスクはブロック回数の見積もりが増え．最大ブロック時間が伸びるため，最悪応答時間が悪くなる．本論文の目的は高優先度タスクのスケジューラビリティ改善であり，図 7 より，高優先度タスクのスケジューラビリティは改善されていることがわかるため，本手法は有効であると言える．

6. おわりに

本論文では，マルチコアリアルタイムシステムにおける静的優先度パーティショニングスケジューリングを対象とした，ネストをサポートし，高優先度タスクのスケジューラビリティを改善する同期プロトコルを提案した．評価では，リソース要求を行う際に発生するブロックを実現するスピンを，既存のノンプリエンプティブ方式からプリエンプティブ方式に改良した際のタスク毎の最大ブロック時間と最悪応答時間計算式を導出し，高優先度タスクの最悪応答時間，およびスケジューラビリティの改善が見られることを確認した．静的優先度パーティショニングスケジューリングを対象とした，ネストをサポートし，プリエンプティブ方式を採用した際の最悪応答時間計算式の導出と，スケジューラビリティ解析が行われている論文は私が知る限り存在しないため，その点でも本論文は有用だといえる．

今後の課題として，実機での評価と，ネストが行われる場合の評価が挙げられる．本論文では，計算式よりタスクの最悪応答時間を求め，スケジューラビリティ解析を行った．しかし，実際にはプリエンプトによるオーバヘッドが生じるため，TOPPERS/FDMP, FMP カーネルのアプリケーションに本手法を適用し，評価を行う予定である．また，本評価で作成したタスクのリソース要求は全て単一のロックを取得するものであり，ネストしてロックを取得する様なリソース要求が発生する場合の評価は行われていない．実際のリアルタイムシステムでネストするケースは少なくないため，ネストを考慮した評価も行う予定である．

参考文献

[1] T. Baker. A comparison of global and partitioned EDF schedulability tests for multiprocessors. In *International Conf. on Real-Time and Network*, 2005.
[2] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *RTCSA 2007.*, pp. 47 –56, aug. 2007.
[3] B. B. Brandenburg and J. H. Anderson. An Implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP Real-Time Synchronization Protocols in LITMUS^RT. In *2008 14th IEEE International Conference on Embedded and*, pp. 185–194. IEEE, Aug. 2008.
[4] B. Brandenburg, A. D. Block, J. M. Calandrino, U. C. Devi, H. Leontyev, and J. H. Anderson. LITMUSRT: A status report. *Proceedings of the 9th RealTime Linux Workshop*, 4(3):107–123, 2007.
[5] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. *Proceedings.,10th International Conference on Distributed Computing Systems*, pp. 116–123, 1990.
[6] R. Rajkumar, L. Sha, and J. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings. RTSS*, pp. 259–269. IEEE Comput. Soc. Press, 1988.
[7] 本田, 高田. 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 D, Apr 2008.
[8] 石田, 本田, 高田. マルチプロセッサ用 rtos. 電気関連学会東海支部連合大会, Sep 2009.
[9] 一場, 松原, 本田, 高田. 中断可能な優先度継承キューイングスピンロックとそのハードウェア実装. 情報処理学会論文誌 コンピューティングシステム, 4(3):133–146, May 2011.