Vol.2012-OS-122 No.4
2012/8/1

マルチコア*Tender*の開発

山本 貴大[1]　長井 健悟[1]　山内 利宏[1]　谷口 秀夫[1]

概要：近年，プロセッサのコア数は増加してきており，処理性能を向上させるためには，OS のマルチコアプロセッサへの対応が必要である．OS のマルチコア対応の手法として，ジャイアントロックがある．しかし，ジャイアントロックは，カーネル処理の並列性が失われるため，性能が低下するという欠点がある．したがって，マルチコアにおいてカーネル処理の性能を向上するためには，細粒度ロックを用いることが望ましい．本稿では，***Tender*** をマルチコアに対応させる際の方式について述べる．マルチコアへの対応を行う上で，***Tender*** の OS 構造に着目することにより，修正工数の削減を図りつつ，OS 処理の並列性を向上させることを目指す．また，マルチコアへ対応した ***Tender*** の修正工数と性能を評価した．

Development of Multicore *Tender* Operating System

Takahiro Yamamoto[1]　Kengo Nagai[1]　Toshihiro Yamauchi[1]　Hideo Taniguchi[1]

Abstract: In recent years, the number of processor cores has been increasing, therefore it is necessary to support multicore processor of the OS, in order to improve processing performance. There is a giant lock as a method of multicore support of the OS. However, the giant lock has the disadvantage that performance decreases, because parallelism of the kernel processing is lost. Therefore, in order to improve the performance of kernel processing on multicore, it is desirable to use a fine-grained lock. In this paper, we describe the method of multicore support in ***Tender***. The goal is to reduce modified quantity, and to improve that of the parallelism of OS processing by focus on OS structure in supporting to multicore. In addition, we evaluate modified quantity to support multicore, and the performance of the multicore ***Tender***.

1. はじめに

2000 年代より，搭載するコア数を増加させることで，プロセッサの処理性能の向上が図られるようになった．このため，近年ではマルチコアプロセッサが普及し，コア数も年々増加する傾向にある [1]．また，マルチコアプロセッサを利用して並列に処理を行うためには，マルチコアに対応したオペレーティングシステム (以降，OS) が必要となる．

カーネル内における並列性を制御する方式として，ジャイアントロックがある．ジャイアントロックでは，プロセスの処理がカーネル処理に移行する際に単一の大域的なロックを取得し，プロセスの処理がユーザ処理に戻る際にロックを解放する．ジャイアントロックは，ロックの取得と解放のための修正箇所が限られるため，実装は容易である．しかし，カーネル処理の並列性が失われるため，性能が低下するという欠点がある．Linux や FreeBSD において，マルチコアへの対応が初めて行われた際に，ジャイアントロックが利用されていた．しかし，前述の性能が低下するという欠点のために，Linux 2.6.37 や FreeBSD 6.0-RELEASE では，ジャイアントロックに代わり，細粒度ロックが導入されている [2][3]．マルチコアにおいて，カーネル処理の性能を向上するためには，細粒度ロックを用いることが望ましい．

本稿では，***Tender*** オペレーティングシステム [4](以降，***Tender***) の特徴的な OS 構造である資源の分離と独立化に着目したマルチコアへの対応方式について述べる．具

[1] 岡山大学大学院自然科学研究科
Graduate School of Natural Science and Technology, Okayama University

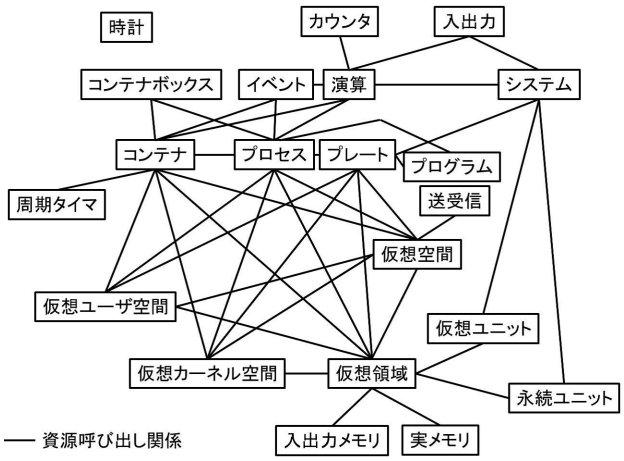

図 1 資源種別の一覧

体的には，***Tender*** において，OS の資源を操作するプログラムの呼び出しを制御している資源インタフェース制御 (以降，RIC: Resource Interface Controller) に着目し，RIC により，一元的に排他制御する．これにより，マルチコア対応への修正工数を削減し，また，資源種別毎に排他制御することで，ジャイアントロックよりも細粒度なロックを実現する．さらに，スケジューラが操作する資源において排他制御を必要としないように資源の管理表をコア毎に分割させることで，スケジューラを各コアで独立して動作させ，プロセスの効率的な動作を実現する．

2. 設計

2.1 *Tender* オペレーティングシステム

Tender は，OS が制御し，管理する対象を資源と呼び，資源種別毎に資源を分離している．***Tender*** の持つ資源種別の一覧を**図 1** に示す．資源種別には，プロセスをプロセッサに割り当てる程度を制御する資源「演算」や，OS の持つ仮想空間を管理する資源「仮想空間」などがある．また，資源種別毎に管理表を保持し，資源の操作時に管理表を更新する．

資源の操作は，資源種別毎に基本操作 (OPEN，CLOSE，READ，WRITE，CONTROL) として定義している 5 つの操作プログラムにより行う．各資源種別の 5 つの操作プログラムは，資源種別，操作種別毎に区切られたプログラムポインタ表で管理している．***Tender*** が有するプログラムポインタ表を**図 2** に示す．資源の操作は，常に RIC を介して行われる．RIC は，資源操作要求を受け取ると，要求に対応する操作プログラムをプログラムポインタ表から呼び出す．

また，各資源種別において確保した資源は，資源名管理木により，管理する．資源名管理木を**図 3** に示す．資源名管理木は，資源種別と固有の文字列により，確保した資源を管理する．資源名管理木は，資源の OPEN(確保) と CLOSE(削除) 操作時に RIC により，操作される．以上より，***Tender*** は，RIC での一元的な資源管理構造を有している．

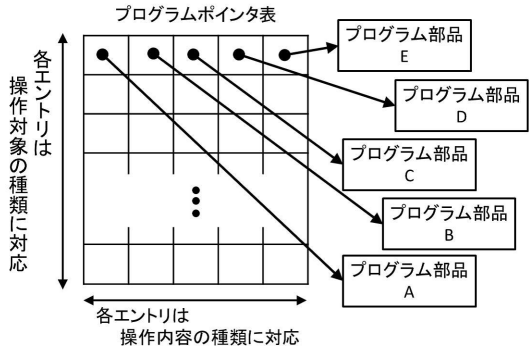

図 2 プログラムポインタ表

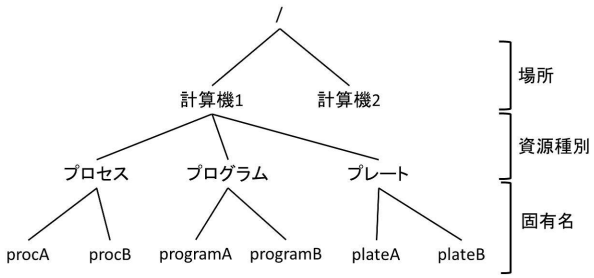

図 3 資源名管理木

2.2 目的

Tender のマルチコア対応を実現する目的として，一元的な排他制御による修正工数の削減と OS 処理の並列性の向上がある．***Tender*** のマルチコア対応では，RIC により，ロックの取得と解放を一元的に行うことでロック箇所を削減し，修正工数の削減を図る．また，***Tender*** では，OS の各資源を資源種別毎に分離して制御し，管理している．このため，資源種別毎に排他制御することで，ジャイアントロックよりも細粒度なロックを実現し，より少ない修正工数で OS 処理の並列性の向上を目指す．

2.3 マルチコア環境

マルチコアでは，各コアは主記憶を共有し，並列してアクセスする．また，デバイスからの割り込みは，割り込みコントローラにより，どのコアに割り込みを配送するかを設定することができる．また，この割り込みコントローラ経由の割り込みとは別に，各コアにはローカル割り込みが存在する．ローカル割り込みには，タイマや温度センサなどの割り込みがあり，各コアは，デバイスからの割り込みに加えてローカル割り込みを受信する．

2.4 方針

マルチコアに対応した ***Tender***(以降，マルチコア ***Tender***) の OS 構成を**図 4** に示す．マルチコア ***Tender***

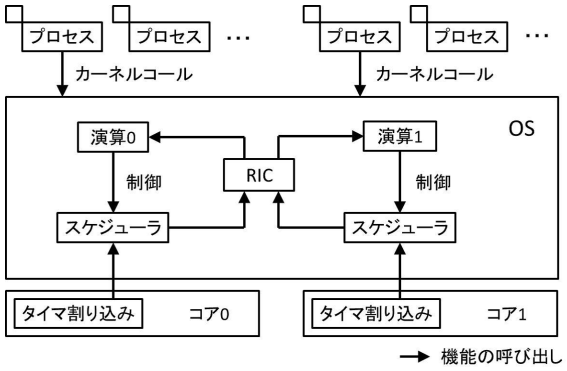

図 4 マルチコア ***Tender*** の OS 構成

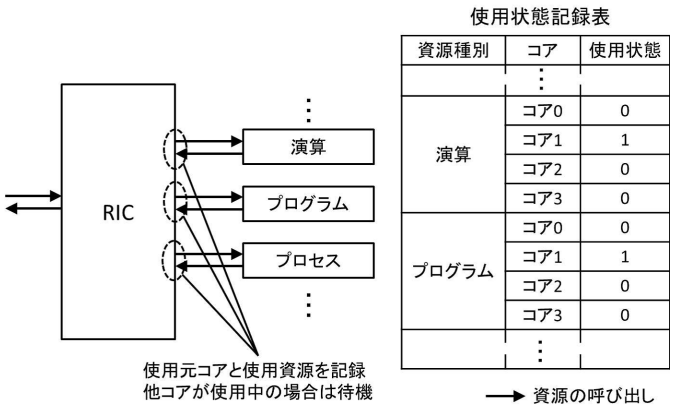

図 5 RIC による一元的な排他制御

では，1 つの OS を複数のコアで並列して動作させる．各コア上で動作する OS は，資源を共有して利用する．以上をもととしたマルチコア ***Tender*** の設計方針を述べる．

(方針 1)複数のコアで協調して OS を走行

各コアにおいて，カーネルのテキスト部とデータ部を共有し，スタック部のみを独立して保有する．各コアは，協調して動作し，資源を共有して利用する．

(方針 2)コア間通信の排除

カーネルコールと OS 内の各機能呼出は，自コアにおいて処理し，他コアへの処理依頼を行わない．

(方針 3)スケジューラの独立化

スケジューラは，各コアで独立して動作し，自コア上のプロセスのみをスケジューリングする．

(方針 1) により，カーネル処理の多いサービスの性能を向上できる．(方針 2) により，カーネル処理は，自コア内で完結する．したがって，他コアへの処理依頼待ちによるオーバヘッドの発生を回避できる．(方針 3) により，1 つのコアが全プロセスのスケジューリングを行う場合に比べ，スケジューリングを高速に行える．また，スケジューラが独立して動作するため，他コアの状態に影響されず，スケジューリングを行える．

スケジューラを独立させて動作させるため，各コアはタイマ割り込みを必要とする．このため，タイマ割り込みは，ローカル割り込みを利用し，各コアへ配送する．他の割り込み処理については，1 つのコアにのみ配送し，同コアにおいて処理するようにする．また，他のコアの OS 処理が入出力デバイスを利用する際には，ポーリングにより入出力処理を行い，割り込みを必要としないようにする．これは，実装を簡易なものにするためである．

3. 実現方式

3.1 基本方式

マルチコア ***Tender*** では，資源を RIC により，一元的に排他制御し，管理する．**図 5** に排他制御の様子を示す．具体的には，資源種別の粒度で排他制御し，資源操作の直前にロックの取得，直後にロックの解放を行う．ロックの取得と解放の際には，使用状態記録表に資源の使用元コアと使用資源種別を記録する．資源「プロセス」においてロックを取得，解放する場合の処理流れを例として，以下に手順を説明する．

(1) 使用状態記録表を参照し，他コアがプロセス管理を使用しているか否かをチェックする．他コアにより使用されていれば，すべての他コアの使用状態が 0 になるまでビジーループにより待機する．使用されていなければ，処理 (2) に進む．
(2) 使用状態記録表の資源種別「プロセス」において自コアの使用状態を 1 加算し，ロックを取得する．
(3) プログラムポインタ表を参照し，プロセス管理の該当操作に対応する操作プログラムを呼び出し，資源に対する操作を行う．
(4) 使用状態記録表の資源種別「プロセス」の自コアにおいて使用状態を 1 減算し，ロックを解放する．

また，資源名管理木は，資源の OPEN と CLOSE 操作時に RIC により，操作される．このため，資源名管理木は，資源名管理木全体にロックを取得することで排他制御を行う．

3.2 排他制御方式

3.2.1 デッドロックへの対処

3.1 節で述べた基本方式では，デッドロックが発生する場合がある．ここでは，マルチコア ***Tender*** においてデッドロックを回避するための排他制御方式について述べる．

これまで ***Tender*** は，資源の分離と独立化を主な設計方針として開発されており，各資源の操作や呼び出し順序は，意識せずに開発されている．このため，***Tender*** における各資源は各々の操作プログラムの中で任意の資源種別の資源を操作する．したがって，マルチコア ***Tender*** では，複数のコアが異なる順序で資源のロックを取得しようとし，デッドロックが発生する．

デッドロックを回避するため，マルチコア ***Tender*** では，資源種別のロックを資源種別の呼び出し関係に従って

（演算，イベント）
（演算，プレート）
（演算，仮想領域）
（演算，入出力）
（イベント，仮想領域）
（イベント，プロセス）
（入出力，仮想領域）
（入出力，プロセス）
（プレート，プロセス）
（プロセス，仮想領域）
（仮想カーネル空間，仮想空間）
（仮想カーネル空間，仮想領域）
（仮想空間，仮想領域）
（仮想領域，仮想ユーザ空間）

図 6 デッドロックとなる組合せ

取得するように設定する．具体的には，他の資源種別の操作プログラムを呼び出す資源種別に対して，資源を操作する前に呼び出す可能性のある資源種別のロックを事前に取得する．これにより，任意な順序でのロックの取得を禁止する．また，プロセスの切り替え処理 (ディスパッチ) の前後で，ロックの使用状態を保存，復元することでディスパッチ前後でも，任意な順序でのロックの取得を許さないようにする．

3.2.2 ロックの取得順序の設定

マルチコア ***Tender*** において実際に OS の動作ログを記録し，その解析結果を元にロックの取得順序を設定した．***Tender*** の動作ログの解析により判明したデッドロックとなる資源種別の組合せを**図 6** に示す．図 6 に基づいてロックの取得順序を設定する．以下の方針により，ロックの取得順序を設定した．

(1) 資源「仮想領域」を含む組合せの場合

図 6 より，デッドロックとなる組合せは，資源「仮想領域」を含むものが多い．このため，資源「仮想領域」を含む組合せに対して，資源「仮想領域」でない方のロックを先に取得した場合，資源「仮想領域」の操作時に多量のロックを事前取得することになる．この状態では，資源「仮想領域」の操作時に並列して操作できる資源の数が減少してしまう．このため，資源「仮想領域」を含む組合せの場合，資源「仮想領域」のロックを先に取得する．

(2) 資源「演算」を含む組合せの場合

資源「演算」は，スケジューラにより操作される．スケジューラは，タイマ割り込みにより頻繁に呼び出されるため，資源「演算」の利用頻度は高い．このため，資源「演算」の操作時に多数のロックを事前取得すると，資源「演算」により，ロックされた資源はスケジューラの動作時に使用できなくなり，利用効率が低下する．したがって，資源「演算」を含む組合せの場合，資源「演算」のロックを先に取得する．

以上の方針に基づいて設定した，ロックの事前取得の設定を**表 1** に示す．表 1 において，デッドロックとなる組合せは存在しない．したがって，マルチコア ***Tender*** では，表 1 の設定に従いロックの事前取得が必要な資源種別は，ロックの事前取得を行う．

表 1 ロックの事前取得順序の設定

取得する資源種別	ロックの取得順序
イベント	仮想領域 → 演算 → プロセス → イベント
演算	仮想領域 → 演算
入出力	仮想領域 → 演算 → プロセス → 入出力
プレート	仮想領域 → 演算 → プロセス → プレート
プロセス	仮想領域 → プロセス
仮想カーネル空間	仮想領域 → 仮想空間 → 仮想カーネル空間
仮想空間	仮想領域 → 仮想空間
仮想ユーザ空間	仮想領域 → 仮想ユーザ空間

表 2 ロックの取得順序

順序	資源種別	順序	資源種別
1	時計	12	システム
2	コンテナ	13	送受信
3	コンテナボックス	14	入出力
4	仮想領域	15	イベント
5	周期タイマ	16	時間カウンタ
6	演算	17	仮想ユーザ空間
7	プロセス	18	仮想空間
8	プログラム	19	仮想カーネル空間
9	プレート	20	実メモリ
10	永続ユニット	21	入出力メモリ
11	仮想ユニット		

3.2.3 ディスパッチ前後におけるロックの使用状態の保存と復元

ディスパッチ処理の前後で，適切な順序でロックの解放と取得を行う．これにより，走行しないプロセスが，ロックを取得し続けること，およびデッドロックを回避する．

具体的には，プロセスが切り替わる前後においてロックの使用状態の保存と復元を行うことでデッドロックを回避する．また，ディスパッチ後にロックの使用状態を復元する場合，複数のロックを取得する場合がある．複数のロックを取得する場合，デッドロックの発生しない順序でロックを取得する必要がある．このため，3.2.2 項に基づいて，ロックの使用状態を復元する際の取得順序を設定する．決定したロックの取得順序を**表 2** に示す．以上より，ディスパッチの前後では，ロック状態の保存と復元を行い，復元を行う際は，表 2 の順序に従いロックを取得する．

3.3 スケジューラの独立化

マルチコア ***Tender*** では，プロセスの効率的な動作を実現するため，スケジューラを各コアで独立して動作させる．このため，スケジューラが操作する資源種別は各コア

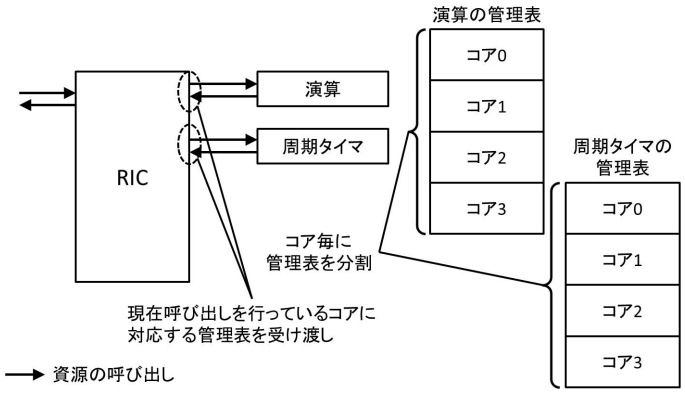

図 7 管理表の分割による資源呼び出し

で独立して扱う必要がある．スケジューラが操作する資源種別として以下の 2 つがある．

(1) 資源「演算」
(2) 資源「周期タイマ」

資源「演算」は，プロセスをプロセッサに割り当てる程度を管理しており，スケジューラは，資源「演算」が割り当てられたプロセスから，次に走行すべきプロセスを決定する [5]．また，ディスパッチの発生時には，資源「演算」の管理表が更新される．資源「周期タイマ」は，スケジューラの呼び出しを管理しており，スケジューラは，タイマ割り込みの発生を契機に資源「周期タイマ」を介して呼び出される．この際，資源「周期タイマ」の管理表へアクセスが行われる．

3.1 節で述べた基本方式では，資源種別の粒度で排他制御しており，各資源種別の資源はコア間で共有し，利用される．したがって，スケジューラを独立して動作させるためには，資源「演算」，資源「周期タイマ」を基本方式とは別の方式により，マルチコア対応し，独立化させる必要がある．

資源「演算」，資源「周期タイマ」を独立化させる方式として，資源「演算」，資源「周期タイマ」の管理表を各コアで保持するように分割させる方式により，実現する．管理表の分割による資源呼び出しの様子を**図 7** に示す．

本方式では，RIC により，資源の使用元コアに対応した管理表を選択し，操作プログラムへ受け渡すことで実現する．資源「演算」においてコア番号 0 のコアが資源を操作する場合の処理流れを例として，以下に手順を説明する．

(1) 資源の使用元コアのコア番号を取得する．
(2) 取得したコア番号に対応する管理表の先頭アドレスを取得する．
(3) 取得した管理表の先頭アドレスを引数として操作プログラムを呼び出し，資源に対する操作を行う．

これにより，資源の操作プログラムは，引数として受け渡された管理表のみにアクセスし，更新する．したがって，各コアは，自コアの有する管理表にしかアクセスできないため，各コアは他コアを意識せずに独立して資源を操作することができる．しかし，プロセスの起床や，休眠時に全プロセスの状態を確認する．このため，資源「演算」に関しては，全プロセスの状態を確認する処理でのみ，他コアの管理表における資源の状態を確認する．

表 3 ***Tender*** と Linux における修正工数の比較

	行数		ファイル数	
	全体	修正数	全体	修正数
Tender	220,964	3,302	688	75
Linux 2.0	330,150	7,822	883	49
Linux 2.0’	143,743	3,635	506	35

表 4 ロック使用箇所数の比較

	ロック使用箇所数
Tender	36
Linux 2.0	4

4. 評価

4.1 修正工数の評価

4.1.1 評価内容

Tender におけるマルチコア対応のための修正箇所について，マルチコア対応を行う前の ***Tender*** と差分をとり，追加と変更された行数とファイル数をカウントした．また，修正工数の評価において比較対象とする OS として Linux 2.0 を対象とした．Linux 2.0 は，Linux においてはじめてマルチコア対応が行われたバージョンであり，ジャイアントロックを用いてマルチコア対応が行われている [6]．

Linux における修正工数の評価では，マルチコア対応以外の修正箇所を排除するため，拡張子が s，c，h であり，ファイル内に “smp” の文字列を含むファイルを対象とした．また，Linux は ***Tender*** に比べ，サポートしているアーキテクチャやドライバの数が多い．このため，アーキテクチャ依存のコードに関する修正箇所は，i386 のみを対象とし，ドライバに関する修正箇所は，ブロック型，キャラクタ型，およびネットワークに関するドライバのみを対象とした．また，マルチコア対応が行われる前の Linux として，Linux 1.3.9 を使用した．

4.1.2 評価結果

Tender と Linux における修正工数の比較を**表 3** に示す． 表 3 より，マルチコア対応に必要な修正行数は，***Tender*** の方が少ないことがわかる．これは，Linux では，ネットワークやファイルシステムに関するコード量が多く，これに多くの修正を要しているためである．表中の Linux 2.0’ に，Linux 2.0 においてネットワークやファイルシステムのコードを除外した場合を示している．これより，ネットワークやファイルシステムに関するコードを除外しても，修正行数は 3,635 行となり，***Tender*** の修正行数とほぼ同等となっている．修正のあったファイル数では，***Tender***

よりも Linux の方が少ない．これは，***Tender*** のソースコードは，***Tender*** の設計方針である資源の分離と独立化のため，カーネルのコードを記述したファイルが資源種別毎に分散しているためである．このため，カーネルのコードを記述したファイルを修正する際に，***Tender*** は，Linux よりも修正するファイル数が増加する．

また，ロックの使用箇所数の比較を**表 4** に示す．ロックの使用箇所数では，***Tender*** は，Linux よりも使用箇所数が多くなっている．これは，***Tender*** では資源以外の箇所について，排他制御が必要な箇所には，個別にスピンロックを用いて排他制御しているためである．

以上より，***Tender*** におけるマルチコア対応では，Linux においてジャイアントロックを用いたマルチコア対応の修正工数と同等以下の修正工数で，ジャイアントロックよりも細粒度なロックを用いた排他制御を実現し，マルチコア対応を行えたといえる．

4.2 性能評価の内容と環境

4.2.1 評価内容

マルチコア ***Tender*** の評価では，測定用のプログラムとして，4 つのプログラム (プログラム A，プログラム B，プログラム C，およびプログラム D) を用いる．各プログラムの内容は評価項目により変更する．シングルコアでは，プログラム A〜D を逐次実行する．マルチコアでは，プログラム A〜D を各コアに分散して実行する．ここで，各プログラムの処理に要した時間を測定し，比較する．評価観点として，以下の観点から評価を行う．

(1) シングルコア用の ***Tender*** のカーネル (以降，シングルコアカーネル)，またはマルチコア用の ***Tender*** のカーネル (2 コア，4 コア)(以降，マルチコアカーネル) を使用する場合
(2) シングルコアの計算機，またはマルチコアの計算機で動作する場合
(3) ユーザ処理，またはカーネル処理を実行する場合
(4) カーネル処理において，依存関係のない処理，または依存関係のある処理を実行する場合

ただし，ここでの依存関係とは，ある資源種別が操作プログラムの中で，別の資源種別の操作プログラムを呼び出すことを指す．

4.2.2 評価環境

CPU は，シングルコアの計算機では，Pentium4 2.8GHz を用い，マルチコアの計算機では，Core i7-2600 3.40GHz を用いた．各計算機上でシングルコアカーネルとマルチコアカーネルを動作させて評価を行った．

4.2.3 使用するプログラム

本評価で使用するプログラムを以下に説明する．

＜ユーザ処理の評価＞

ユーザ処理の評価では，プログラム A〜D として，変数のインクリメント処理を 10,000,000 回繰り返す処理をユーザが指定した回数だけ繰り返すプログラムを使用した．また，プログラムの処理回数は，シングルコアカーネルで実行した場合に処理時間が約 1.0 秒となるように指定した．

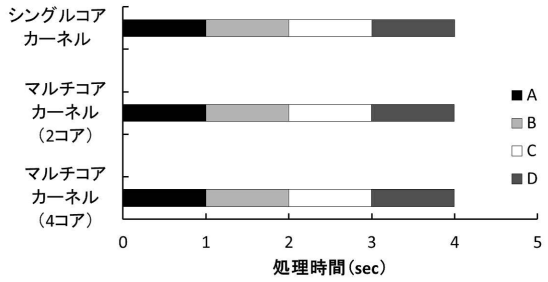

図 8 シングルコアの計算機におけるユーザ処理の測定結果

＜カーネル処理の評価＞

カーネル処理の評価では，依存関係のないカーネル処理を行う場合と依存関係のあるカーネル処理を行う場合の 2 通りを測定する．以下に各場合において使用するプログラムの概要を述べる．

(1) 依存関係のないカーネル処理を行う場合
　依存関係のない資源種別の資源を操作するプログラム A〜D を使用し，測定を行う．プログラム A〜D は，依存関係のない資源種別として資源「時計」，資源「コンテナ」，資源「時間カウンタ」，および資源「入出力」の各基本操作プログラムに相当するカーネルコールをユーザから指定した回数だけ呼び出し，資源操作を行う．各プログラムの処理回数は，シングルコアカーネルで処理させた場合に処理時間が約 1.0 秒となるように指定した．
(2) 依存関係のあるカーネル処理を行う場合
　依存関係のある資源種別の資源を操作するプログラム A〜D を使用し，測定を行う．プログラム A〜D として，プロセスの操作を行うプログラムを用いて評価を行った．今回用いたプログラムはプロセスの操作に関するカーネルコールをユーザから指定された回数だけ呼び出す．プログラムの処理回数は，シングルコアカーネルで処理させた場合に処理時間が約 1.0 秒となるように指定した．以下にプログラムが呼び出すカーネルコールの概要を示す．

(A) プロセスの生成 (proccreate)
(B) プロセスの再起動 (procreset)
(C) プロセスの削除 (procdelete)
(D) プロセスの動作する空間の移動 (procmove)
(E) プロセスの休眠 (execwait)
(F) プロセスの起床 (execwakeup)
(G) 動作するプロセスの切り替え (prochange)

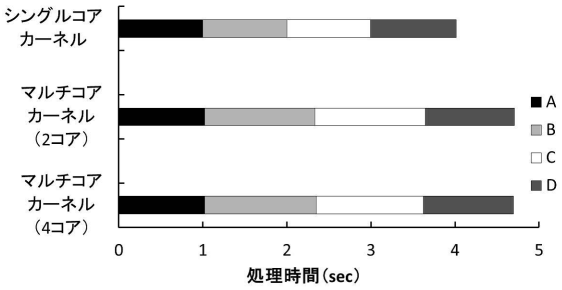

図 9 シングルコアの計算機における依存関係のないカーネル処理の測定結果

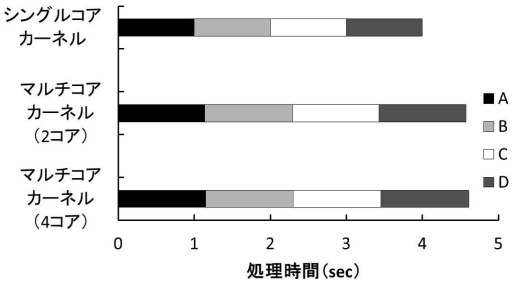

図 10 シングルコアの計算機における依存関係のあるカーネル処理の測定結果

4.3 シングルコアの計算機における性能評価

シングルコアの計算機における測定結果を**図 8**，**図 9**，および**図 10** に示す．

図 9 と図 10 においてマルチコアカーネル (2 コア，4 コア) の処理時間は，図 8，またはシングルコアカーネルの処理時間よりも，約 16%増加している．これは，マルチコアカーネル (2 コア，4 コア) では，排他制御処理が実行されているためである．しかし，実際にはシングルコアの計算機上では，排他制御は不要である．このため，図 9 と図 10 より，排他制御のオーバヘッドは，カーネル処理における依存関係に関係なく一定である．

4.4 マルチコアの計算機における性能評価

4.4.1 ユーザ処理の測定結果

ユーザ処理における測定結果を**図 11** に示す．縦軸は，使用したカーネルの種類を示し，数値は，コア番号を示している．図 11 より，シングルコアカーネル，マルチコアカーネルのどちらについても各プログラムの処理時間は約 1.0 秒であり，マルチコアの計算機においてプログラムを並列して実行させた場合もシングルコアの計算機で実行した場合と比較すると処理時間は変化しない．これは，ユーザ処理では排他制御によるオーバヘッドが発生しないためである．処理に要する全体の時間は，シングルコアの場合は約 4.0 秒，マルチコアで 4 コアの場合は約 1.0 秒であり，並列に実行することにより，処理時間を 4 分の 1 に削減できている．

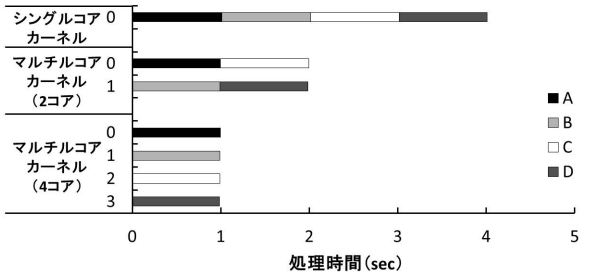

図 11 マルチコアの計算機におけるユーザ処理の測定結果

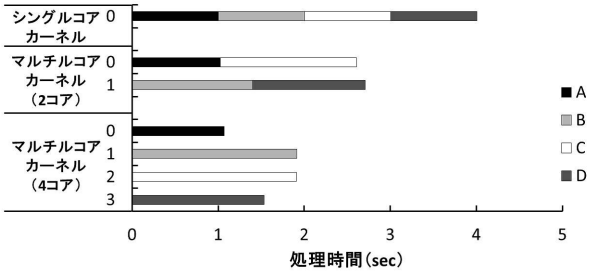

図 12 マルチコアの計算機における依存関係のないカーネル処理の測定結果 (コア別処理時間)

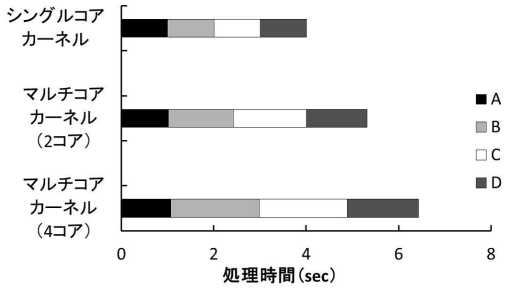

図 13 マルチコアの計算機における依存関係のないカーネル処理の測定結果 (合計処理時間)

4.4.2 依存関係のないカーネル処理の測定結果

依存関係のないカーネル処理における測定結果を**図 12** と**図 13** に示す．図 12 は，各コアに分散させたプログラム A〜D の処理時間を示している．図 12 より，各プログラムは各コアに処理が分散されていることがわかる．シングルコアカーネルにおける処理時間は，約 4.0 秒であった．これに対し，マルチコアカーネル (2 コア) における処理時間は，約 2.7 秒であった．したがって，2 コアで処理を分散させた場合，処理時間を約 32%削減できている．さらに，マルチコアカーネル (4 コア) における処理時間は，約 1.9 秒であり，4 コアで処理を分散させた場合，処理時間を約 52%削減できている．

また，図 13 は，コア数別におけるプログラム A〜D の処理時間の合計を示している．図 13 より，コア数が増加するにつれて処理時間が増加していることがわかる．これは，資源の OPEN(確保)，CLOSE(削除) 操作時にすべての資源種別は資源名管理木にアクセスし，資源名管理木の更新を行っているためである．資源名管理木は，アクセス時

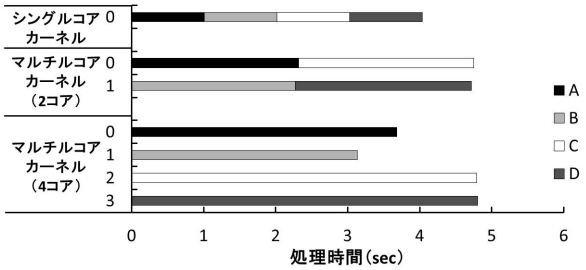

図 14 マルチコアの計算機における依存関係のあるカーネル処理の測定結果 (コア別処理時間)

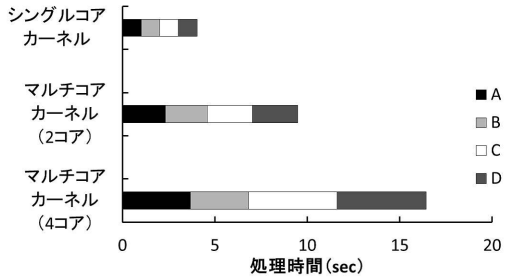

図 15 マルチコアの計算機における依存関係のあるカーネル処理の測定結果 (合計処理時間)

に管理木全体をロックするため，資源の OPEN，CLOSE 操作時に排他制御のオーバヘッドが生じている．

4.4.3 依存関係のあるカーネル処理の測定結果

依存関係のあるカーネル処理における測定結果を**図 14** と**図 15** に示す．図 14 は，各コアに分散させたプログラム A〜D の処理時間を示している．図 14 より，マルチコアカーネルでは，各プログラムの処理時間がシングルコアカーネルに比べ著しく増加していることがわかる．これは，各コアが同時に同じ資源種別の資源を操作しようとした結果，排他制御のオーバヘッドが増大しているためである．

また，図 15 は，コア数別におけるプログラム A〜D の処理時間の合計を示している．図 15 より，シングルコアカーネルにおける各プログラムの処理時間の合計は，約 4.0 秒であり，対してマルチコアカーネル (2 コア) における各プログラムの処理時間の合計は，約 9.5 秒である．さらに，マルチコアカーネル (4 コア) における各プログラムの処理時間の合計は，約 16.4 秒である．これより，マルチコアカーネルでは，2 コアの場合，プログラムの処理時間の約 57%が，4 コアの場合，プログラムの処理時間の約 75%が排他制御のオーバヘッドとなっている．

5. おわりに

Tender のマルチコア対応方式では，***Tender*** の特徴的な OS 構造に着目し，マルチコア対応を行った．まず，資源インタフェース制御 (RIC: Resource Interface Controller) に着目し，RIC により一元的に排他制御することで，修正工数の削減を図った．次に，資源の分離と独立化に着目し，資源種別毎に排他制御することで，OS 処理の並列性を向上した．また，プロセスの効率的な動作を実現するため，各コアにおいてスケジューラの独立した動作を実現した．

修正工数の評価では，***Tender*** は，Linux においてジャイアントロックを用いてマルチコア対応を行った場合と同等以下の修正工数で，ジャイアントロックよりも細粒度なロックにより，マルチコア対応を行うことができ，明らかにした．

また，性能評価では，OS におけるユーザ処理，カーネル処理の観点から性能を評価した．評価の結果，ユーザ処理，およびカーネル処理における依存関係のない資源種別の操作では，各コアでの並列処理が実現されていることを明らかにした．

残された課題として，依存関係のあるカーネル処理と資源名管理木の操作における並列性の向上がある．

謝辞 本研究の一部は，科学研究費補助金基盤研究 (B) (課題番号：24300008) による．

参考文献

[1] L.Gwennap, “Sandy Bridge spans generations,” Microprocessor Report, September, 2010.
[2] Andi Kleen, “Linux multi-core scalability, ” Linux-Kongress 2009, 2009.
[3] Attilio Rao, “The locking infrastructure in the FreeBSD kernel, ” AsiaBSDCon 2009, 2009.
[4] 谷口秀夫，青木義則，後藤真孝，村上大介，田端利宏，“資源の独立化機構による ***Tender*** オペレーティングシステム，” 情報処理学会論文誌，vol.41，no.12，pp.3363-3374，2000.
[5] 田端利宏，乃村能成，谷口秀夫，“***Tender*** オペレーティングシステムにおける資源「演算」を利用したプロセスグループの実行性能調整法，” 電子情報通信学会論文誌，Vol.J87-D-I，No.11，pp.961-974，2004.
[6] M.Beck, H.Bohme, M.Dziadzka, U.Kunitz, R.Magnus, and D. Verworner, “Linux Kernel Internals, ” Addison-Wesley, second edition, 1998.