Vol.2013-OS-126 No.16
2013/8/1

# マルチコア *Tender* における排他制御の細粒度化による並列性向上手法

山本 貴大[1]　山内 利宏[1]　谷口 秀夫[1]

**概要**：近年，マルチコアプロセッサの普及に伴い，オペレーティングシステム（以降，OS）のマルチコア対応が必要となっている．OS のマルチコア対応における課題として処理の並列性の向上がある．***Tender*** オペレーティングシステム（以降，***Tender***）では，***Tender*** 特有の OS 構造である資源インタフェース制御において一元的に排他制御することで修正工数を抑制し，マルチコア対応を実現した．このとき，資源インタフェース制御において資源の種類ごとに排他制御することで異なる種類の資源の並列処理を実現した．しかし，本手法では，同じ種類の資源を並列に処理することができない．本稿では，マルチコア向け ***Tender***（以降，マルチコア ***Tender***）において一元的な排他制御構造を維持しつつ，排他制御を細粒度化する手法について述べる．これにより，同じ種類の資源の並列処理を実現し，処理の並列性を向上させる．このとき，修正に要した工数について評価する．また，マイクロベンチマークを使用し，Linux，および FreeBSD と比較することでマルチコア ***Tender*** の性能を評価する．

## 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（以降，マルチコア）[1] が普及し，コア数も年々増加する傾向にある [2]．マルチコア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．ここで，OS のマルチコア対応における課題としてカーネル処理の並列性の向上がある．

カーネル処理の並列性を向上させるためには，細粒度なロックを使用し，資源を排他制御することが望ましい．しかし，細粒度ロックによる排他制御では，排他制御箇所が増大し，修正コストが膨大になる．一方で，ジャイアントロックといった大局的なロックを使用し，カーネル資源を排他制御する場合，排他制御箇所は少なく実現は容易である．しかし，カーネル処理の並列性が失われ，性能が低下する問題がある．

***Tender*** オペレーティングシステム（以降，***Tender***）では，OS の資源を分離と独立化しており，それらの資源を ***Tender*** 特有の構造である資源インタフェース制御により一元的に管理している [3]．マルチコア向け ***Tender***（以降，マルチコア ***Tender***）では，資源の排他制御箇所を資源インタフェース制御に限定し，一元的に排他制御することでマルチコア対応に要する修正工数を抑制し，マルチコア対応を実現した [4]．このとき，資源インタフェース制御において資源の種類ごとに排他制御することで異なる種類の資源の並列操作を実現した．しかし，この手法では，同じ種類の資源を並列に操作することができない．このため，同じ種類の資源を複数のコアから同時に操作する場合，競合が発生し，処理性能が低下する．

そこで，本稿では，マルチコア ***Tender*** において一元的な排他制御構造を維持しつつ，同じ種類の資源内の通番ごとに排他制御することで排他制御を細粒度化する手法について述べる．これにより，同じ種類の資源の並列操作を実現し，カーネル処理の並列性を向上させる．評価では，修正に要した工数を排他制御箇所数の観点から Linux と比較し，評価する．また，マイクロベンチマークを使用し，カーネル処理を並列に実行した場合の処理時間を Linux，および FreeBSD と比較することでマルチコア ***Tender*** の性能を評価する．

## 2. 関連研究

マルチコア，またはメニーコア向け OS に関する近年の研究について述べる．まず，モノリシックカーネルにおけるカーネル内の並列性制御の手法として，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局的なロックを取

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

得する．この手法では，排他制御箇所がカーネル処理への移行時に限定されるため，修正工数は少なく実現は容易である．しかし，カーネル処理の並列性が失われ，性能が低下するという欠点がある．

この欠点を克服するため，Linux[5] や FreeBSD[6] では，ジャイアントロックを撤廃し，細粒度ロックに置き換えている．しかし，細粒度ロックによるマルチコア対応は，排他制御箇所が膨大となるため，修正工数が増大する．

また，マルチコア上で複数のカーネルを走行させるマルチカーネル方式 OS として Barrelfish[7] がある．Barrelfish は，コア間通信を明示的に行い，各コアにおけるカーネル状態の複製の通信パターンを明示的に制御する．これにより，ハードウェアによるキャッシュ一貫性プロトコルのオーバヘッドを抑制し，処理性能を向上させる．しかし，本手法は，開発者が計算機内部のデータの一貫性条件を十分に理解する必要があるため，開発者の負担が増加する．

また，メニーコア向けの OS として Factored Operating System（fos）[8] がある．fos は，OS の機能を固有なサービスに分離し，各 OS サービスを空間的に分散したコアへ分割する．これにより，スケジューリングの観点を時間多重化によるスケジューリングから空間多重化によるスケジューリングに置き換える．fos はマイクロカーネル構造 [9] により実現されており，各 OS サービスはサーバとしてアプリケーションプログラムに提供される．この際，メッセージパッシングを使用し，サーバと通信する．また，サーバ内ではプリエンプションが発生しないため，1 つのスレッドのみがサーバ上で動作する．このため，共有メモリのロックが不要となり，性能が向上する．

また，マルチコア向け非対称型 OS として GenerOS[10] がある．GenerOS は，マルチコア環境において複数あるコアをアプリケーションプログラムが走行するコア，カーネルサービスを提供するコア，および割り込みサービスを提供するコアの 3 つに分類する．これにより，アプリケーションプログラムの実行においてカーネルサービスの動作によるキャッシュ汚染 [11] や無関係な割り込みによる実行の妨害 [12] を排除することができ，性能を向上させることができる．しかし，性能を最大限発揮するためには，3 つに分類されたコアの最適な配置を探索する必要がある．その際，アプリケーションプログラムの再起動を繰り返す必要がある．また，性能を向上するためには十分な数のコアが必要となる．

その他のマルチコア対応の例として，車載システムの一種である AUTOSAR[13] がある．AUTOSAR では，単一のコアでのみドライバやサービスといった大部分の OS 機能を提供するという手法を用いてマルチコア対応を実現している．しかし，この手法では，ジャイアントロックを用いた場合よりも性能の点で劣るという結果が示されている．

## 3. *Tender* オペレーティングシステム

### 3.1 資源の分離と独立化

***Tender*** では，OS が制御し，管理する対象を資源と呼び，26 種類の資源を分離と独立化して管理している．例えば，***Tender*** では，既存の OS のプロセスを「プロセス」，「プログラム」，「仮想ユーザ空間」，「仮想空間」，「仮想領域」，および「実メモリ」の 6 つの資源に分割している．

### 3.2 資源インタフェース制御

資源インタフェース制御は，***Tender*** 特有の OS 構造であり，表プログラム構造と呼ばれるプログラム管理構造に基づき，資源を管理しているプログラム部分（以降，資源管理処理部）の呼び出しを制御している．

表プログラム構造は，資源管理処理部を独立化させるための機構である．表プログラム構造は，プログラム部品とプログラム部品へのポインタからなる．プログラムポインタ表の行要素と列要素は，操作する資源の種類と操作内容に対応している．資源管理処理部は，資源への操作を資源の生成（open 系），削除（close 系），入力（read 系），出力（write 系），および制御（control 系）の 5 つに分類し，各々をプログラム部品として実現する．

プログラムポインタ表は，資源インタフェース制御により管理される．資源インタフェース制御は，プログラム部品の呼び出しを制御している．つまり，プログラム部品の呼び出しは，資源インタフェース制御へ依頼し，資源インタフェース制御がプログラム部品を呼び出す機構としている．このため，資源インタフェース制御では，プログラム部品の呼び出し状況を把握することができる．

また，個々の資源は，資源識別子と資源名を付与し，資源名管理部と呼ばれる部分により管理する．資源識別子は，資源の場所，種類，および同一種類内の通番を情報として有する数字である．また，資源名は，場所名，種類名，および固有名からなる文字列である．資源名管理部は，資源名管理木とフリーノードリストから構成される．資源名管理木は，資源識別子と資源名の変換機能を提供し，各枝は資源の種類に対応し，葉（ノードと呼ぶ）は個々の資源に対応する．資源名管理部は，資源の生成（open 系）と削除（close 系）時に操作する．このとき，資源名管理木へのノードの追加と削除は，フリーノードリストを繋ぎ変えることで行う．

## 4. 旧方式における課題

***Tender*** は，文献 [4] で実現したマルチコア対応（以降，旧方式）により，修正工数を抑制した上でマルチコア環境において処理の並列性を向上させた．しかし，旧方式では，以下の課題がある．

（課題 1）同じ種類の資源は複数のコアで同時に操作不可
旧方式では，資源の種類ごとに排他制御するため，同じ種類の資源を複数のコアから同時に操作できない．このため，同じ種類の資源を複数のコアから同時に操作しようとした場合，ロックを取得したコア以外のコアは，ロックが解放されるまで処理待ちとなり，処理の並列性が低下する．

（課題 2）資源名管理部は複数のコアで同時に操作不可
旧方式では，資源名管理部を一括して排他制御するため，複数のコアから資源名管理部を操作することができない．このため，資源名管理部を複数のコアから同時に操作しようとした場合，ロックを取得したコア以外のコアは，ロックが解放されるまで処理待ちとなり，処理の並列性が低下する．資源名管理部は，open 系，close 系の操作時に操作するため，資源の生成と削除における処理の並列性が低下する．

本稿では，以上の 2 つの課題に対処する．また，対処するにあたり修正工数の抑制を図るため，旧方式の排他制御方式である資源インタフェース制御での一元的な排他制御構造を維持した対処方式を実現する．

## 5. 実現方式

### 5.1 方針

（課題 1）は，資源の通番ごとに排他制御することで対処する．資源の通番ごとに排他制御することで，同じ種類の資源でも通番が異なれば，同時に操作できるため，複数のコアから同時に同じ種類の資源を操作できる．このとき，資源の通番情報は，資源識別子により資源インタフェース制御で把握できる．このため，資源インタフェース制御において排他制御することで一元的な排他制御構造を維持する．実現方式の詳細は，5.2 節で述べる．

（課題 2）は，資源名管理木を資源の種類ごとに排他制御することで異なる種類の資源での同時操作を実現する．また，各コアで共有して利用していたフリーノードリストをコアごとに保持させることで，フリーノードリストをコアごとに独立して操作できるようにする．これにより，資源名管理木へのノードの追加と削除時における並列性を向上させる．実現方式の詳細は，5.3 節で述べる．

### 5.2 資源管理処理部における排他制御方式

資源インタフェース制御で把握できる情報として資源識別子がある．そこで，資源識別子の有する情報に基づいて排他制御することで資源インタフェース制御における排他制御構造を維持する．旧方式では，資源の種類の情報を利用し，資源の種類ごとに排他制御していた．また，資源の場所は，計算機の単位で割り当てられる情報であるため，排他制御対象としては不適切である．同一種類内の通番は，同じ種類の資源内において個々の資源を指すため，同一種類内の通番の情報を利用して排他制御することで複数のコアから同じ種類の資源の操作を同時にできるようにする．具体的には，旧方式において資源の利用状況を管理していた使用状態記録表を**図 1** に示すように拡張し，各資源の種類において通番ごとにロック構造体を保持することで，コアごとに個々の資源の使用状態を管理する．図 1 では，資源「プロセス」の通番 0 に対するロックをコア 2 が取得し，通番 1 に対するロックをコア 0 が取得している．また，資源「仮想空間」の通番 0 に対するロックをコア 1 が取得している．本方式の実現により，同一種類の資源であっても通番が異なれば同時に操作することができるようになり，処理の並列性が向上する．

| 資源の種類 | 通番 | 使用元コア | 使用状態 |
|---|---|---|---|
| プロセス | 0 | コア0 | 0 |
| | | コア1 | 0 |
| | | コア2 | 1 |
| | | コア3 | 0 |
| | 1 | コア0 | 1 |
| | | コア1 | 0 |
| | | コア2 | 0 |
| | | コア3 | 0 |
| 仮想空間 | 0 | コア0 | 0 |
| | | コア1 | 1 |
| | | コア2 | 0 |
| | | コア3 | 0 |
| | 1 | コア0 | 0 |

**図 1** 使用状態記録表

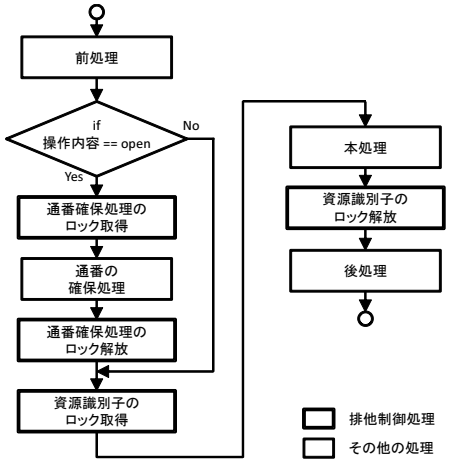


**図 2** 資源操作の処理流れ

また，資源操作の open 系は，資源識別子の情報を有さない．このため，資源操作の open 系は，プログラム部品の内部において通番を確保した後，確保した通番に対応する資源識別子に対してのロックを取得する．資源操作の処理流れを**図 2** に示す．図 2 に示すように open 系では，通番の確保処理において通番が重複しないように処理を排他制御する．通番を確保後，確保した通番に対応する資源識別子に対してロックを取得し，本処理を実行する．

表 1 ***Tender*** と Linux における排他制御インタフェースの使用数の比較

| | 行数 | | | ファイル数 | | |
|---|---|---|---|---|---|---|
| | 全体 | 使用数 | 割合 (%) | 全体 | 使用数 | 割合 (%) |
| 旧方式 ***Tender*** | 196,469 | 128 | 0.07 | 579 | 24 | 4.2 |
| 新方式 ***Tender*** | 202,049 | 653 | 0.32 | 581 | 62 | 10.7 |
| Linux 2.0.40 | 130,532 | 12 | 0.01 | 229 | 4 | 1.8 |
| Linux 2.4.37 | 220,396 | 1,076 | 0.49 | 372 | 88 | 23.7 |
| Linux 2.6.39 | 737,679 | 5,033 | 0.68 | 1,567 | 330 | 21.1 |

### 5.3 資源名管理部における排他制御方式

資源名管理部での排他対象となる資源名管理木は，木構造を有しており，資源の種類ごとにノードを形成している．このため，資源の種類ごとに排他制御することで並列性を向上させる．これにより，異なる種類であれば，複数のコアから同時に資源名管理木を操作できる．

また，旧方式では，フリーノードリストを各コアで共有して利用していた．排他制御の細粒度化により，複数のコアからの資源名管理木の同時操作を実現した．しかし，旧方式どおりフリーノードリストを各コアで共有して利用すると，ノードの追加と削除時にフリーノードリストへの操作が競合し，処理の並列性が低下する．このため，フリーノードリストは，各コアで独立して保持させる．これにより，各コアは，ノードの追加と削除時に，自コアに対応するフリーノードリストのみを操作すればよいため，排他制御は不要となり，処理の並列性が向上する．

## 6. 評価

### 6.1 修正工数の評価

#### 6.1.1 評価方式

修正工数の評価では，***Tender*** と Linux において排他制御インタフェースの使用数を調査した．評価対象 OS は，***Tender*** では，旧方式のマルチコア ***Tender***（以降，旧方式 ***Tender***）と新方式により実現されたマルチコア ***Tender***（以降，新方式 ***Tender***）である．また，Linux では，ジャイアントロックを使用して初めてマルチコア対応した Linux のバージョンである Linux 2.0.40，ジャイアントロックをすべて撤廃し，細粒度ロックに置き換えた Linux のバージョンである Linux 2.6.39，およびジャイアントロックと細粒度ロックを混載したバージョンである Linux 2.4.37 である．

また，Linux は，サポートするデバイスドライバの種類が多いため，***Tender*** よりもソースコード規模が大きくなり，排他制御インタフェースの使用数も増加する．このため，調査対象のソースコードは，カーネルの主要機能のみを対象とすることで ***Tender*** と Linux において比較するソースコード規模を均一化する．具体的には，Linux において/arch/x86，/fs/ufs，/init，/ipc，/kernel，/mm，/net/ipv4，および/net/ethernet を調査対象ディレクトリとした．

また，排他制御インタフェースの使用数は，調査対象となるソースコードから排他制御インタフェースの文字列（スピンロックであれば，“spin_lock”）を含む行とファイルをカウントし，算出した．

#### 6.1.2 評価結果

***Tender*** と Linux における行数とファイル数についての排他制御インタフェース使用数を**表 1** に示す．新方式 ***Tender*** は，排他制御の細粒度化により，旧方式 ***Tender*** よりも排他制御インタフェースの使用数が増加している．また，Linux も改版を重ねるにつれて排他制御インタフェースの使用数が増加している．

まず，すべてのカーネルにおいて行数における排他制御インタフェースの使用数だけで評価すると Linux 2.0.40 が最も排他制御インタフェースの使用数が少ない．これは，Linux 2.0.40 では，ジャイアントロックにより，カーネルを単一のロックで排他制御しているためである．しかし，Linux 2.0.40 は，この大局的なロックのため，マルチコア環境におけるカーネル処理の性能が低下している．一方，旧方式 ***Tender*** は，資源インタフェース制御での一元的な排他制御により，Linux 2.0.40 の約 10 倍の排他制御インタフェースの使用で異なる種類の資源操作における処理の並列性を実現している．

次に，細粒度なロックを使用したマルチコア対応方式では，行数における排他制御インタフェースの使用数において新方式 ***Tender*** は，Linux 2.4.37 と比べ，約 40%，Linux 2.6.39 と比べ，約 87%だけ排他制御インタフェースの使用数を抑制している．このため，ソースコードの規模に関わらず，新方式 ***Tender*** における排他制御インタフェースの使用数は，Linux 2.4.37 と Linux 2.6.39 よりも抑制されているといえる．

また，ファイル数で評価した場合でも新方式 ***Tender*** は，排他制御インタフェースの使用数と割合ともに Linux 2.4.37 と Linux 2.6.39 よりも少ない．このとき，ディレクトリごとにおける排他制御を使用しているファイル数の割合は，Linux 2.6.39 の場合，/kernel における割合が最も高く約 53.8%である．次に割合が高いディレクトリは，/mm で約 50.7%である．また，同様に Linux 2.0.40 と Linux 2.4.37 でも/kernel と/mm において排他制御を使

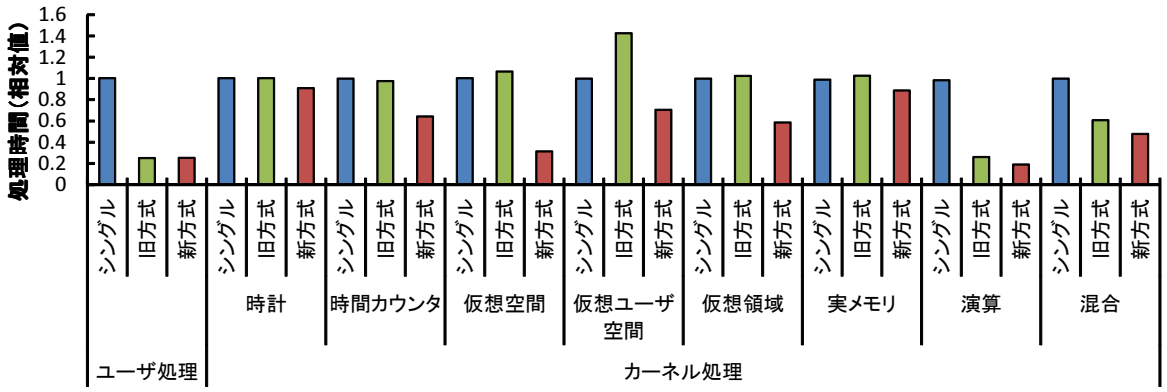


**図 3** 資源の並列操作における処理時間

用するファイル数の割合が高いことが分かった．/kernel と/mm には，実行管理やメモリ管理といった主要なカーネルコードが置かれているため，/kernel と/mm における排他制御数の割合が高いのは妥当だといえる．

一方，***Tender*** では，旧方式 ***Tender*** の場合，資源インタフェース制御に関する部分に多くの排他制御が使用されている．しかし，新方式 ***Tender*** では，資源インタフェース制御以外に，プロセス管理やメモリ管理といった部分にも排他制御が使用されている．これは，排他制御の細粒度化により，資源インタフェース制御での排他制御だけでは排他制御できない共有データが現れたためである．この共有データに排他制御が必要となったため，新方式 ***Tender*** では，資源インタフェース制御以外のプロセス管理やメモリ管理といった部分に排他制御を使用している．

以上より，新方式 ***Tender*** は，旧方式 ***Tender*** よりも排他制御数が増加したが，Linux 2.4.37 と Linux 2.6.39 よりも排他制御インタフェースの使用を抑制しているため，修正工数を抑制できているといえる．

### 6.2 旧方式 *Tender* との比較評価

#### 6.2.1 評価方式

本評価では，旧方式 ***Tender*** と新方式 ***Tender*** において各コア上で資源を並列に操作した場合の性能を評価する．評価対象の処理は，ユーザ処理，同じ種類の資源，および異なる種類の資源の並列処理である．このとき，ユーザ処理は，変数のインクリメントを繰り返す処理を行い，各資源の操作では，open，close，read，write，および control の操作を呼び出すカーネルコールを順に呼び出し，処理を実行する．また，同じ種類の資源の操作では，7 種類の資源について評価し，異なる種類の資源の操作では，「時計」，「時間カウンタ」，「仮想空間」，および「仮想領域」の 4 つの資源を並列に操作した場合について評価する．測定は，各種類の資源の操作を任意の回数だけ行うプロセスを 4 コア上に配置し，実行した場合の処理時間を測定する．

#### 6.2.2 評価結果

旧方式 ***Tender*** と新方式 ***Tender*** における資源の並列操作における処理時間の測定結果を**図 3** に示す．図 3 の横軸における「シングル」は，シングルコア環境で動作させた旧方式 ***Tender*** 上での結果，「旧方式」は旧方式 ***Tender*** 上での結果，および「新方式」は新方式 ***Tender*** 上での結果を示している．横軸における「ユーザ処理」は，ユーザ処理を並列処理した場合，「混合」は異なる種類の資源を並列処理した場合を示している．また，カーネル処理の「混合」以外は，同じ種類の資源の操作として，図 3 中で示した資源名の資源を各コアで並列処理した場合を示している．また，縦軸は，シングルを 1 とした場合の処理時間を相対値で示している．

まず，ユーザ処理は，競合なしに処理されるため，排他制御が発生しない．このため，旧方式と新方式ともに処理時間がシングルの約 4 分の 1 になっている．次に，同じ種類の資源の並列処理では，すべての場合において新方式は，シングルと旧方式よりも処理時間が短い．しかし，資源「時計」と資源「実メモリ」における処理時間の削減量は，他の場合よりも少ない．これは，資源「時計」と資源「実メモリ」では，資源の操作について生成（open）と削除（close）の操作割合が高いためである．open 系と close 系の操作は，資源名管理部における排他制御の影響を大きく受ける．このため，新方式における資源名管理部の操作では，open 系と close 系の操作を並列に実行すると資源名管理部の排他制御の影響を大きく受け，並列性が低下する．

また，旧方式における資源「仮想ユーザ空間」の処理時間がシングルを大きく上回っている．これは，資源「仮想ユーザ空間」では，仮想ユーザ空間の処理内部で多段に他資源の操作を実行しており，排他制御の影響を大きく受けたためである．また，資源「演算」について旧方式と新方式は，差が僅かである．これは，資源「演算」は旧方式からの設計方針であるスケジューラの独立化のため，例外的に旧方式からコアごとに独立して動作するよう設計されているためである．

表 2　評価環境

| | |
|---|---|
| OS | 新方式 ***Tender*** |
| | FreeBSD 8.2-RELEASE |
| | Linux 2.6.37 |
| CPU | 3.4 GHz（4 コア） |
| RAM | 8,192 MB |

また，旧方式 ***Tender*** から異なる種類の資源における並列操作を実現していた．このため，混合における旧方式はシングルよりも処理時間が削減されている．このとき，新方式では排他制御の細粒度化により，異なる種類の資源の操作においても旧方式より処理時間が削減されている．以上より，新方式 ***Tender*** は，同じ種類の資源の並列処理を実現しており，旧方式 ***Tender*** よりも性能が向上していることが明らかとなった．

### 6.3 他 OS との比較評価

#### 6.3.1 評価方式

本評価では，新方式 ***Tender*** と Linux，および FreeBSD において各コアで OS 機能を並列に実行するマイクロベンチマークを使用し，処理時間を測定した．また，評価結果は，評価観点ごとに基準を設定して相対値で示す．**表 2** に評価環境を示し，以下に評価対象機能と評価観点について述べる．

＜評価対象機能＞

(1) プロセス管理
プロセスの生成と削除における処理性能を評価する．本評価では，プロセスの生成と削除を繰り返すプログラムの処理時間を測定する．

(2) プロセス間通信
2 つのプロセス間でのプロセス間通信における処理性能を評価する．本評価では，同一コア上における 2 つのプロセス間において 4 KB のデータを通信する場合の処理時間を測定する．

＜評価観点＞

(1) コア数の変化による処理性能
24 個のプロセスを各コア上に均一に分散させて走行させる．このとき，コア数を 1~4 で変動させて処理時間の変化を評価する．このとき，コア数が 1 の場合を基準とした相対値で評価結果を示す．

(2) プロセス数の変化による処理性能
4 コア上でプロセスを均一に分散させて走行させる．このとき，プロセス数を 4~32 で変化させて処理時間の変化を評価する．このとき，プロセス数が 4 の場合を基準とした相対値で評価結果を示す．

#### 6.3.2 マイクロベンチマークについて

マイクロベンチマークは，1 つの親プロセスと複数の子プロセスから構成される．親プロセスは，子プロセスを各コアに配置し，実行させた後，すべての子プロセスが終了するまで待機状態となる．子プロセスは，配置されたコア上で評価対象の OS 機能を実行する．すべての子プロセスが終了した後，親プロセスは実行を再開し，測定結果を出力する．このとき，子プロセスの生成数は，ユーザにより任意に指定され，また，子プロセスはコア番号が 0 のコアから順番に配置される．測定区間は，親プロセスが待機状態となり子プロセスが実行を開始してから子プロセスの実行が終了し，親プロセスが実行を再開するまでとする．

表 3　プロセスの生成と削除，およびプロセス間通信のインタフェース

| 形式 | 機能 |
|---|---|
| proccreate(name, path, argv, vmid) | path で指定された内容と引数 argv を持つ資源名 name のプロセスを仮想空間 vmid 上に生成する． |
| procgetexec(pid, mips, core) | pid で指定されたプロセスに演算の程度 mips でコア番号 core に所属する演算を関連付ける． |
| execprocdelete(pid) | pid で指定されたプロセスに割り当てられた演算とプロセスをすべて削除する． |
| ctainercreate(name, reqaddr, size) | reqaddr で指定されたアドレスに貼り付いた大きさが size で資源名 name のコンテナを生成する． |
| ctainerboxcreate(name) | 資源名 name でコンテナボックスを生成する． |
| ctainersend(ctainerid, ctainerbxid, reqaddr, op) | コンテナ ctainerid を送信モード op でコンテナボックス ctainerbxid に貼り付け希望アドレス reqaddr を指定して送信する． |
| ctainerreceive(ctainerid, ctainerbxid, reqaddr, op) | コンテナ ctainerid を受信モード op でコンテナボックス ctainerbxid に貼り付け希望アドレス reqaddr を指定して受信する． |

***Tender*** では，プロセスに関連付けた演算 [14] が属するコア上でのみプロセスが走行する．このため，各コアへのプロセスの配置方法として ***Tender*** では，生成した子プロセスに特定のコア上に属する演算を関連付けることで各コアへ子プロセスを配置する．ここで，演算とは，***Tender*** 独自の資源であり，プロセスをプロセッサへ割り当てる程度を管理する資源である．Linux では，sched_setaffinity()，FreeBSD では，cpuset_setaffinity() を使用し，子プロセスの走行するコアを指定する．また，***Tender*** では，**表 3** に示す ***Tender*** 独自のカーネルコールを使用し，評価対象 OS 機能の処理を実現する．以下に評価対象機能ごとの子プロセスの処理について述べる．

＜プロセス管理＞

***Tender*** では，まず，proccreate() によりプロセスを生成する．このとき，生成するプロセスは，テキスト部が約 2,000 B，データ部と BSS 部はともに 0 B のプロセスである．次に procgetexec() により，プロセスに演算を関連付けプロセスを走行させる．その後，execprocdelete() により，プロセスと演算を削除する．

Linux と FreeBSD では，まず，fork() により，プロセスを生成する．次に生成されたプロセスが execve() により，プロセスイメージを上書きする，このとき，上書きするプロセスイメージは，テキスト部が約 1,000 B，データ部が約 200 B，BSS 部が約 10 B のプロセスである．その後，生成したプロセスを kill() により削除し，wait() により生成したプロセスの終了を待機する．

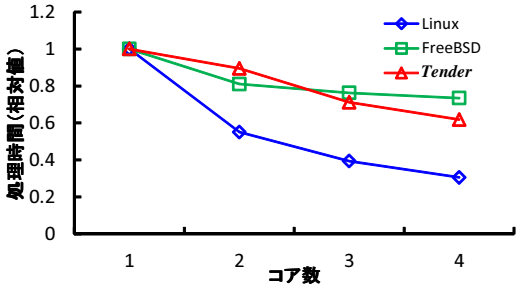


図 4　コア数の変化による処理時間（プロセス管理）

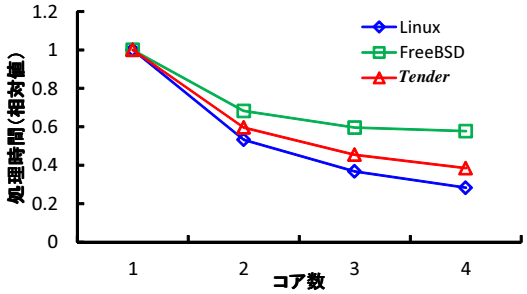


図 6　コア数の変化による処理時間（プロセス間通信）

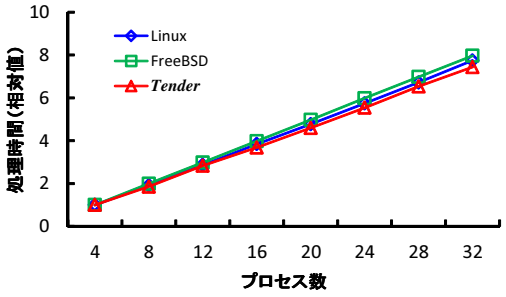


図 5　プロセス数の変化による処理時間（プロセス管理）

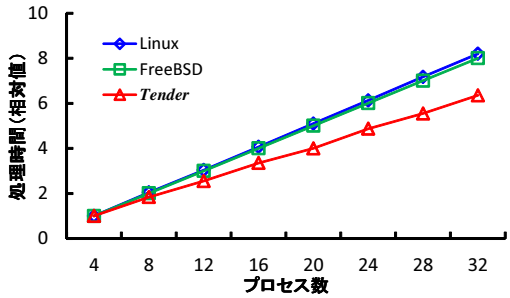


図 7　プロセス数の変化による処理時間（プロセス間通信）

また，生成するプロセスは，***Tender***，Linux，および FreeBSD ともに while 文により無限ループを行うプログラムである．このとき，処理内容が同じにも関わらず，Linux や FreeBSD よりも ***Tender*** のプログラムの方がテキスト部のサイズが大きい．これは，***Tender*** は，プログラムのコンパイル時に ***Tender*** のカーネルコールライブラリとシステムコールライブラリに静的にリンクするためである．

＜プロセス間通信＞

プロセス間通信では，同一コア上の 2 つのプロセス間においてサイズが 4 KB のデータを通信する．***Tender*** では，資源「コンテナ」と資源「コンテナボックス」によりプロセス間通信を実現する [15]．まず，送信側プロセスは，ctainercreate() によりコンテナを生成し，データを書き込む．次に ctainerboxcreate() によりコンテナボックスを生成し，生成したコンテナボックスへ移動モードで ctainersend() によりコンテナを送信する．受信側プロセスは，ctainerreceive() により，コンテナボックスから移動モードでコンテナを受信し，データを読み込む．

Linux と FreeBSD では，パイプを介したプロセス間通信を行う．まず，pipe() により，パイプを生成する．送信側プロセスは，write() により，パイプへデータを書き込む．受信側プロセスは，read() により，パイプからデータを読み込む．

### 6.3.3　評価結果

＜プロセス管理＞

まず，プロセス管理の評価におけるコア数の変化による処理性能の測定結果を**図 4** に示す．図 4 より，***Tender*** は，FreeBSD と比べるとコア数の増加につれ，処理時間の削減率が高いことがわかる．処理時間の削減率は，4 コアの場合，***Tender*** は約 38%，FreeBSD は約 26%である．一方，Linux は，4 コアの場合，約 69%の削減率である．4 コアで動作した場合，処理時間の理想の削減率は 75%であるため，Linux は，***Tender*** や FreeBSD に比べ，競合が少なく理想値に近いことがわかる．***Tender*** においてプロセスの生成は，open 系の操作であるため，資源名管理木の操作と通番の確保を行う．このとき，処理の競合が発生するため，排他制御が必要となり，資源名管理木の操作と通番の確保は，逐次処理として実行される．しかし，資源名管理木の操作と通番の確保が完了した後は，競合なく処理できるため，コア数の増加につれて処理時間を削減できた．

次に，プロセス管理の評価におけるプロセス数の変化による処理性能の測定結果を**図 5** に示す．図 5 より，プロセス数が 32 の場合，***Tender*** は，Linux に比べ約 4.0%，FreeBSD よりも約 6.7%だけ性能低下率が低い．これは，***Tender*** では，プロセスをスケジューリングする際，各コアが独立してスケジューリングするため，スケジューリングによる他コアへの影響がない．したがって，スケジューリングの際のオーバヘッドが Linux や FreeBSD よりも小さくなり，性能低下率が低くなっている．

以上より，***Tender*** は，コア数の増加に対する性能向上率は，FreeBSD より高く，プロセス数の増加に対する性能低下率は，Linux や FreeBSD と同等以上といえる．

＜プロセス間通信＞

まず，プロセス間通信の評価におけるコア数の変化による処理性能の測定結果を**図 6** に示す．図 6 より，プロセス管理と同様に ***Tender*** は，FreeBSD と比べるとコア

数の増加に伴う，処理時間の削減率が高いことがわかる．処理時間の削減率は，4 コアの場合，***Tender*** は約 61%，FreeBSD は，約 42%となっている．また，4 コアの場合における Linux の処理時間削減率は，約 71%であり，***Tender*** との差は約 10%となっている．***Tender*** におけるプロセス間通信では，データを通信する際，各コアは異なる資源識別子を持った資源により通信するため，競合が発生しない．したがって，プロセス管理に比べ，並列処理が可能な処理区間が多く，処理時間を削減できた．

次に，プロセス間通信の評価におけるプロセス数の変化による処理性能の測定結果を**図 7** に示す．図 7 より，***Tender*** は，プロセス数が 32 の場合，FreeBSD よりも約 20%，Linux よりも約 22%だけ性能低下率が低い．これは，プロセス間通信の場合もプロセス管理と同様に ***Tender*** は，スケジューリングによるオーバヘッドが Linux や FreeBSD よりも低いため，プロセス数の増加に対して性能低下率が低くなっている．

以上より，***Tender*** は，コア数の増加に対して FreeBSD よりも性能向上率が高く，プロセス数の増加に対して Linux や FreeBSD よりも性能低下率が低いといえる．

## 7. おわりに

***Tender*** において資源インタフェース制御での一元的な排他制御構造を維持し，並列性を向上させる手法について述べた．旧方式によるマルチコア ***Tender*** の課題を解決するため，資源インタフェース制御が把握可能な情報として資源識別子の情報を利用し，同一種類の資源内の通番ごとに排他制御することで排他制御の細粒度化を実現した．また，資源名管理部においては，資源名管理木を資源の種類ごとに排他制御し，フリーノードリストをコアごとに保持させることで処理の並列性を向上させた．

また，修正工数の評価より，ソースコード行数における排他制御インタフェースの使用割合において新方式によるマルチコア ***Tender*** は，Linux 2.4.37 よりも約 0.17%，Linux 2.6.39 よりも約 0.36%だけ排他制御インタフェースのコード行数を抑制していることを示した．性能評価では，新方式によるマルチコア ***Tender*** は，プロセス管理において Linux には劣るものの，FreeBSD よりも約 12%だけコア数の増加による処理時間削減率が高く，性能向上率が高いことを示した．また，プロセス間通信の性能評価では，***Tender*** は，コア数の増加による処理時間の変化において FreeBSD より性能向上率が高く，プロセス数の増加による処理時間の変化において Linux と FreeBSD より性能低下率が低いことを示した．

**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)，および科学研究費補助金若手研究 (B)（課題番号 ： 25730046）による．

### 参考文献

[1] Hammond, L., Nayfeh, B. and Olukotun, K.: A Single-Chip Multiprocessor, *IEEE Computer*, Vol.30, pp.79–85 (1997).

[2] Rusu, S., Tam, S., Muljono, H., Stinson, J., Ayers, D., Chang, J., Varada, R., Ratta, M., Kottapalli, S., Vora, S.: A 45 nm 8-Core Enterprise Xeon Processor , *IEEE J. Solid-State Circuits*, Vol.45, No.1, pp.7–14 (2010).

[3] 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)．

[4] 山本貴大，長井健悟，山内利宏，谷口秀夫：マルチコア ***Tender*** の開発，情報処理学会研究報告，Vol.2012-OS-122，No.4，電子媒体（2012）．

[5] Boyd-Wickizer, S., Clements, A. T., Mao, Y., Pesterev, A., Kaashoek, M. F., Morris, R. and Zeldovich, N.: An Analysis of Linux Scalability to Many Cores, In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, pp.1–16 (2010).

[6] Lehey, G.: Improving the FreeBSD SMP implementation, In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pp.155–164 (2001).

[7] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A. and Singhania, A.: The Multikernel: A New OS Architecture for Scalable Multicore Systems, In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp.29–44 (2009).

[8] Wentzlaff, D., Agarwal, A.: Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores, *ACM SIGOPS Operating Systems Review*, Vol.43, No.2, pp.76–85 (2009).

[9] Liedtke, J.: Toward Real Microkernels, *Communications of the ACM*, Vol.39, No.9, pp.70–77 (1996).

[10] Yuan, Q., Zhao, J., Chen, M. and Sun, N.: GenerOS: An Asymmetric Operating System Kernel for Multi-core Systems, In *Parallel & Distributed Processing, 2010 IEEE International Symposium on*, pp.1–10 (2010).

[11] Bao, Y., Chen, M., Ruan, Y., Liu, L., Fan, J., Yuan, Q., Song, B. and Xu, J.: HMTT: A Platform Independent Full-System Memory Trace Monitoring System, In *SIGMETRICS '08: Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pp.229–240 (2008).

[12] Regehr, J., Duongsaa, U.: Preventing Interrupt Overload, In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pp.50–58 (2005).

[13] Böhm, N., Lohmann, D., Schröder-Preikschat, W. and Erlangen-Nuremberg, F.: *A Comparison of Pragmatic Multi-Core Adaptations of the AUTOSAR System, 7th annual workshop on Operating Systems Platforms for Embedded Real-Time applications*, pp.16–22 (2011).

[14] 田端利宏，谷口秀夫：***Tender*** オペレーティングシステムの資源「演算」によるプログラム実行速度調整の実現と評価，情報処理学会論文誌，Vol.40，No.6，pp.2523–2533 (1999)．

[15] 田端利宏，谷口秀夫：***Tender*** オペレーティングシステムにおけるプロセス間通信機能の実現と評価，情報処理学会研究報告，1999-OS-81,Vol.99，No.32，pp.95–100 (1999)．