排他制御を局所化するマルチコア向け*Tender*の実現

山本貴大[†] 山内利宏[†] 谷口秀夫[†]

マルチコアプロセッサ（以降，マルチコア）を搭載した計算機の登場により，オペレーティングシステム（以降，OS）のマルチコア対応が行われている．FreeBSD や Linux は，すべてのコアで OS の資源を共有し，互いに排他制御することでカーネル処理の並列化を実現している．処理の並列性を向上させるためには，細粒度ロックを使用することが望ましい．しかし，細粒度ロックは実現する際，修正量が非常に多い．また，ロック処理が頻発し，オーバヘッドが大きくなりやすい．ここでは，***Tender*** について，特有の OS 構造である資源インタフェース制御において排他制御することで，細粒度ロックであり，かつ一括した排他制御を実現する機構を述べる．また，マルチコア対応の際の修正量の抑制と処理の並列性の向上について既存 OS と比較し，評価する．

Implementation of the Localized Exclusive Control for Multi-core *Tender*

Takahiro Yamamoto,[†] Toshihiro Yamauchi[†]
and Hideo Taniguchi[†]

Operating system has supported multi-core environment because of the appearance of a computer equipped with a multi-core processor. Linux and FreeBSD achieved parallel processing of the kernel by sharing the resources of the OS with all cores, and making exclusive control each core. It is desirable to use the fine-grained lock to improve concurrency of the processing. However the fine-grained lock needs to modify many points. In addition, because exclusive control processing occurs frequently, an overhead tends to increase. This paper describes a mechanism to achieve local and fine-grained lock in ***Tender***. We compared the OS that we developed to existing OSes about a number of modified point and performance.

1. はじめに

マルチコアプロセッサ[1]（以降，マルチコア）が普及し，コア数も年々増加する傾向にある[2]．マルチコア環境においてオペレーティングシステム（以降，OS）がアプリケーションプログラムの処理だけでなく OS の処理も並列に行うためには，OS のマルチコアへの対応が必要となる．

OS が制御し，管理する対象である資源のマルチコア環境における扱い方として，FreeBSD や Linux のようにコア間で OS の資源を共有して利用する方式がある．この方式では，すべてのコアが任意のタイミングで任意の資源を利用するため，コア間で共有する資源を排他制御し，データの不整合を防ぐ必要がある．

コア間で OS の資源を共有して利用する方式において，カーネル処理の並列性を向上させるためには，細粒度ロックを利用することが望ましい．しかし，細粒度ロックを利用するためには，すべてのコードを見直し，ロックに必要な箇所を特定する必要がある．また，排他制御の必要な箇所が非常に多くなるため，細粒度ロックを利用したマルチコア対応では，修正量が非常に多くなる．

本論文では，***Tender*** のマルチコア対応において排他制御を局所化し，かつカーネル処理の並列性を向上させる方式について述べる．***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，これらの資源を ***Tender*** 特有の OS 構造である資源インタフェース制御により，一括して管理している．そこで，***Tender*** のマルチコア対応では，コア間で OS の資源を共有して利用する際，コア間で共有して利用する資源を資源インタフェース制御において一括して排他制御する．これにより，排他制御箇所を局所化し，ロックの取得と解放に必要な修正量を抑制する．また，資源インタフェース制御が有する情報を基に排他制御することで，カーネル処理

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

の並列性を向上させる．具体的には，次の 2 つの方式を実現する．1 つは，資源の種類ごとに排他制御する方式（粗粒度ロックによる方式）であり，もう 1 つは，個々の資源ごとに排他制御する方式（細粒度ロックによる方式）である．

評価では，マルチコア対応の際の修正量について排他制御量の観点から既存 OS と比較した．また，性能についてプロセス生成，メモリの確保と解放，およびプロセス間通信の機能の並列性を既存 OS と比較し，評価した．

細粒度ロックを実現する提案方式は，プログラム部品が扱う個々の資源の単位で一括した排他制御を実現する新しい方式である．しかし，一括した排他制御では，排他できない共有データが一部，存在する．これらの共有データは，コアごとに分割することでプログラム部品内の排他制御をすべて排除できる．これを実現することで，マルチコア環境において排他制御を考慮せずにプログラム部品を開発や改版でき，既存 OS よりも OS 機能の修正量が少なくなると推察できる．また，共有データをすべて排除できない実装でも，コア数の増加に対して既存 OS と同等程度の性能向上率であることを示しており，共有データを排除することで，さらなる性能向上を期待できる．

2. 関 連 研 究

マルチコア環境における並列性制御の手法に関する近年の研究について述べる．まず，コア間で OS の資源を共有して利用する方式がある．この方式において，カーネル内の並列性を制御する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は，プロセスがカーネル処理を行う際に単一の大域的なロックを取得することで，カーネル内のすべての資源を排他制御する．ジャイアントロックは，ロックの取得と解放の修正箇所が限定されるため，実現は容易である．しかし，大域的なロックを利用するため，同時に排他制御区間を実行できるコアは，1 つだけとなり，並列性が失われ，性能が低下するという欠点がある．FreeBSD や Linux において，マルチコアへの対応が初めて行われた際は，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，最新の FreeBSD や Linux 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[3),4)]．

しかし，細粒度ロックは，ロックに必要な箇所を特定するため，コードを見直す必要がある．また，ジャイアントロックに比べ，ロック箇所が増加するため，修正量が増加し，対応に多くの工数を要する欠点がある．マルチコア ***Tender*** は，細粒度ロックによるマルチコア対応を行う際，ロック箇所を資源インタフェース制御に局所化することで細粒度ロックの実現における修正量の増加を抑制することができる．

次に，各コアに OS の資源を分散させて利用する方式がある．Factored Operating System[5)]，GenerOS[6)]，および NIX[7)] では，カーネル処理を実行するコアとアプリケーションプログラム（以降，AP）を実行するコアに，各コアを分割する．これらの OS は，マイクロカーネル構造[8)] を有し，カーネルサービスを OS サーバとして提供している．このため，AP は，カーネルコア上の OS サーバへ処理を依頼することでカーネル処理を実行する．各コアに OS の資源を分散させて利用する方式では，AP を実行するコアとカーネル処理を実行するコアを分離する．これにより，AP を実行するコア上でカーネル処理を実行することによるキャッシュの汚染[9)] や無関係な割り込みによる実行の妨害[10)] を AP を実行するコアから排除する．また，各 OS サーバは，異なるカーネル処理を実行し，資源の共有を排除することで排他制御が不要となる．しかし，この場合，AP を各コアに分散しても，各 AP が特定のカーネル処理を頻繁に利用すると，処理負荷が特定のコアに集中し，処理性能が低下する．マルチコア ***Tender*** は，すべてのコアにおいてカーネル処理を実行できる．したがって，各コアに AP を分散させることで，特定のコアに処理負荷が集中することを回避できる．

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[11)] がある．AUTOSAR 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．マルチコア ***Tender*** は，すべての OS 機能をすべてのコアにおいて提供するため，OS 機能を頻繁に利用するプロセスを実行する場合，単一のコアでのみ OS 機能を提供する場合よりも性能を向上させることができる．

3. *Tender* オペレーティングシステム

Tender では，OS が制御し，管理する対象を資源と呼び，資源を分離と独立化している．例えば，***Tender*** では，既存 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」「実メモリ」の 6 つの資源に分割している．また，***Tender*** は，分離と独立化した資源を資源インタフェース制御

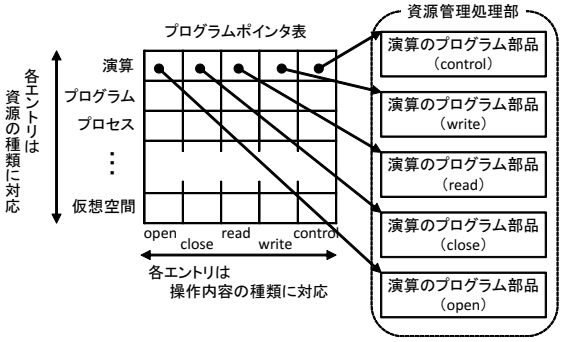


図 1 表プログラム構造

により管理している．資源インタフェース制御[12] は，**図 1** に示す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品（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構成される．図 1 で示すように，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できる．

また，生成した個々の資源に対して資源識別子を付与し，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．資源名管理部は，**図 2** に示す資源名管理木とフリーノードリストから構成される．資源の生成時には，フリーノードリストからノードを切り離し，資源名管理木に追加する．資源の削除時には，削除対象の資源に対応するノードを資源名管理木から削除し，削除したノードをフリーノードリストに繋ぐ．このように，資源名管理部は，資源の生成（open 系）と削除（close 系）において資源名管理木とフリーノードリストを更新する．

4. *Tender* のマルチコア対応の実現方式

4.1 目　　的

マルチコア環境において処理の並列性を向上させるにあたり，カーネル処理を多く利用するサービスの性能を向上させるためには，ユーザ処理だけでなく，カーネル処理の並列性を向上させる必要がある．また，各コアで OS の資源を共有して利用する場合，カーネル処理の並列性を向上させるためには，細粒度ロックを使用して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は，排他制御箇所が多く，マルチコア対応に要する修正量が増加する．

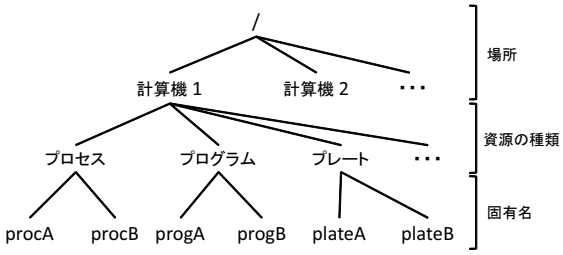


図 2 資源名管理木

Tender のマルチコア対応では，OS の資源の分離と独立化に着目し，資源を一括して管理している資源インタフェース制御において排他制御することで，以下に示す 2 種類の排他制御粒度のマルチコア対応を実現し，カーネル処理の並列性を向上させ，修正量を抑制する．

（方式 1）資源種別単位での排他制御
　本方式では，資源の種類に着目し，排他制御することで，粗粒度ロックによるマルチコア対応を実現する．

（方式 2）資源識別子単位での排他制御
　本方式では，個々の資源に与えられた識別子ごとに排他制御することで，細粒度ロックによるマルチコア対応を実現する．

4.2 基 本 方 式

マルチコア ***Tender*** の OS 構造を**図 3** に示す．マルチコア ***Tender*** では，資源インタフェース制御において把握できる情報に基づき，資源インタフェース制御で資源を一括して排他制御する．資源名管理部は，資源名管理部の処理時に排他制御する．資源の操作では，資源操作の共通インタフェースからプログラム部品を呼び出す．この際，呼びだすプログラム部品に対してロックを取得する．プログラム部品の実行終了後，プログラム部品に対するロックを解放する．その後，資源操作が生成，または削除であった場合，資源名管理部の処理としてノードの追加，または削除を行う．この際，資源名管理部の処理前に資源名管理部に対応するロックを取得し，資源名管理部の処理後に取得したロックを解放する．

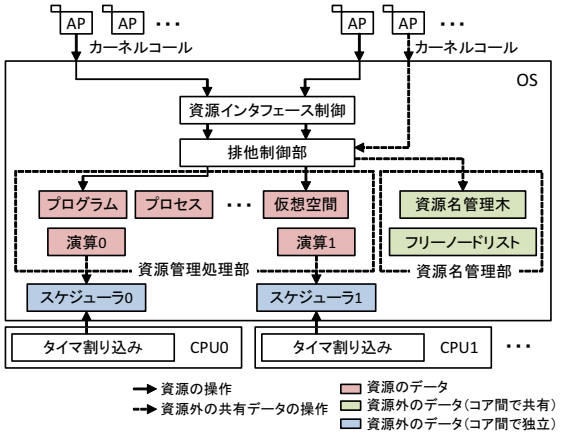


図 3 マルチコア ***Tender*** の OS 構造

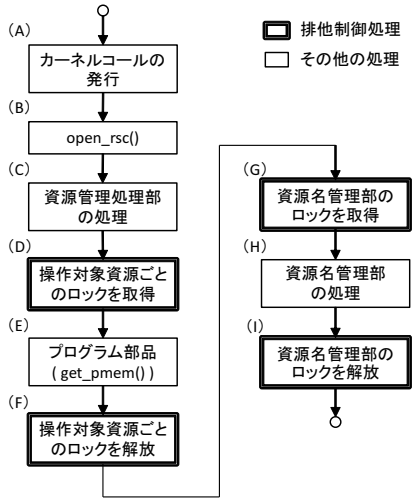
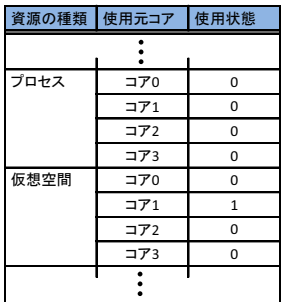


図 4 資源操作の処理流れ（資源「実メモリ」生成）

資源操作の例として，マルチコア ***Tender*** における資源「実メモリ」生成の資源操作の処理流れを**図 4** に示す．マルチコア ***Tender*** における資源操作では，まず資源「実メモリ」生成のカーネルコールが発行されると資源生成の共通インタフェースである open_rsc() が呼び出される．次に，open_rsc() は，資源管理処理部において，実行するプログラム部品をプログラムポインタ表より，生成する資源の資源名に基づき決定する．次に，実行するプログラム部品に対してロックを取得し，実メモリの open 操作のプログラム部品である get_pmem() を実行する．get_pmem() の実行終了後，プログラム部品に対するロックを解放する．次に，資源名管理部に対するロックを取得し，資源名管理部の処理を実行する．ここでの資源名管理部の処理では，open 操作により，生成した資源を資源名管理木の資源「実メモリ」のノード以下に新たなノードとして追加する．最後に資源名管理部に対するロックを解放する．



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

図 5 資源種別排他制御方式における使用状態記録表

このとき，プログラム部品に対するロックと資源名管理部に対するロックは，ロック粒度の異なる 2 種類の排他制御方式を実現する．1 つは，粗粒度ロックを利用した排他制御方式として資源種別単位での排他制御を実現する．もう 1 つは，細粒度ロックを利用した排他制御方式として資源識別子単位での排他制御を実現する．

4.3 資源種別単位での排他制御

資源種別単位での排他制御（以降，資源種別排他制御）では，プログラム部品に対するロック時に資源の種類ごとに排他制御する．この手法では，資源管理処理部の処理において資源識別子が有する情報である資源の種類情報に基づき，操作対象の資源の種類に応じたロックを取得する．このとき，資源のロックの取得状況は，**図 5** に示す使用状態記録表と呼ばれる表により管理する．図 5 では，コア 1 が「仮想空間」を使用している状態となっている．このように，ロックを取得したときにロックを取得した資源の種類とコア番号に対応する使用状態の要素をインクリメントすることで排他制御する．

この手法では，資源インタフェース制御において資源の種類ごとに一括して排他制御でき，異なる種類の資源であれば，各コアで操作を並列に実行できる．

また，資源名管理部の排他制御は，資源名管理部を単一のロックで一括して排他制御する．このため，この手法では，資源名管理部を同時に処理できるコアは，最大 1 つまでである．

4.4 資源識別子単位での排他制御

資源識別子単位での排他制御（以降，資源識別子排他制御）では，プログラム部品に対するロック時に資源の種類と同一種類内の通番に基づき，排他制御する．この手法では，資源管理処理部の処理において資源識別子が有する資源の種類と同一種類内の通番情報に基づき，操作対象の個々の資源に対応したロックを取得

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 6 資源識別子排他制御方式における使用状態記録表

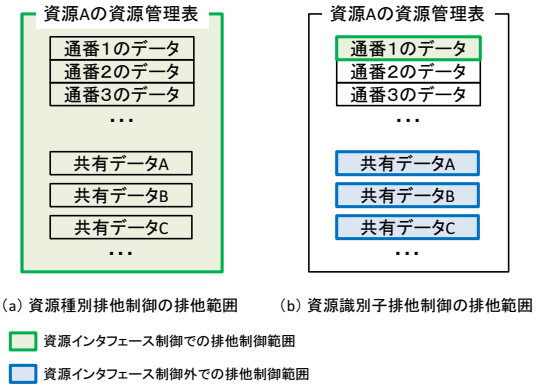


図 7 資源の種類 A の通番 1 の資源を操作する場合における方式別の排他制御範囲

する．このとき，資源のロックの取得状況は，**図 6** に示すように図 5 の使用状態記録表を拡張した使用状態記録表を用いて管理する．図 6 では，コア 2 が「プロセス」の通番 0 の資源，コア 0 が「プロセス」通番 1 の資源，およびコア 1 が「仮想空間」通番 0 の資源を使用している状態となっている．

この手法では，資源インタフェース制御において個々の通番ごとの資源に対してロックを取得するため，細粒度に排他制御できる．このため，同じ種類の資源でも通番が異なれば，各コアで操作を並列に実行できる．

また，資源名管理部の排他制御は，資源名管理木の資源の種類のノードごとに排他制御する．また，ノードの追加と削除処理をコアごとに独立して実行できるようにするため，フリーノードリストは，コアごとに保有させる．これにより，各コアは，異なる種類の資源であれば，資源名管理部の処理を並列して実行することができる．

資源種別排他制御と資源識別子排他制御における排他制御範囲の違いを**図 7** に示す．図 7(a) に示すように資源種別排他制御は，1 つの種類の資源の資源管理表を一括して排他制御する．一方，資源識別子排他制御は，図 7(b) に示すように資源管理表内の 1 つの通番に対応するデータのみを排他制御対象とする．このため，資源識別子排他制御は，資源種別排他制御に対して，より細粒度に排他制御することができる．しかし，図 7(b) 中の共有データは，資源インタフェース制御での排他制御対象とならないため，資源インタフェース制御において一括して排他制御できない．したがって，個別に排他制御する必要がある．なお，これらの共有データは，コアごとに独立して保持させることが可能であり，将来的には，排他制御が不要となる．

5. 評　　価

5.1 修正量の評価

5.1.1 評 価 方 式

修正量の評価では，***Tender***，FreeBSD，および Linux におけるマルチコア対応の際の修正量の違いを明らかにするため，排他制御量の観点から評価を行う．このため，各 OS のソースコード内における排他制御箇所を調査した．FreeBSD と Linux は，排他制御粒度ごとに**表 1** に示す 3 種類のバージョンの OS を対象とした．なお，以降では，FreeBSD 3.5-RELEASE を FreeBSD 3.5-R，FreeBSD 6.4-RELEASE を FreeBSD 6.4-R，および FreeBSD 8.4-RELEASE を FreeBSD 8.4-R とする．***Tender*** は，資源種別排他制御方式により実現したマルチコア ***Tender***（以降，***Tender***-Rkind）と資源識別子排他制御方式により実現したマルチコア ***Tender***（以降，***Tender***-Rid）を対象とした．

また，FreeBSD と Linux は，サポートするプラットフォームやデバイスの種類が多いため，***Tender*** よりもソースコード規模が大きくなり，排他制御量が増加する．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各 OS において比較するソースコード規模を合わせる．具体的には，FreeBSD において include/，sys/dev/ata/，sys/dev/kbd/，sys/dev/io/，sys/kern/，sys/boot/，sys/i386/，sys/net/，sys/sys/，sys/vm/，sys/ufs/，sys/libkern/，および sys/isa/を調査対象ディレクトリとし，Linux において arch/x86/，drivers/block/，fs/ufs/，include/，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．

表 1 ***Tender***，FreeBSD，および Linux における排他制御量の比較

OS の種類	ロック粒度	行数			ファイル数		
		全体	修正行数	割合 (%)	全体	修正ファイル数	割合 (%)
Tender-Rkind	粗粒度ロック (資源種別排他制御)	196,469	128	0.07	579	24	4.2
Tender-Rid	細粒度ロック (資源識別子排他制御)	202,049	653	0.32	581	62	10.7
FreeBSD 3.5-R	ジャイアントロック	589,442	33	0.01	1,268	8	0.01
FreeBSD 6.4-R	ジャイアントロック，細粒度ロック	643,688	2,623	0.41	1,518	235	0.15
FreeBSD 8.4-R	細粒度ロック	768,212	2,837	0.37	1,691	249	0.15
Linux 2.0.40	ジャイアントロック	130,532	12	0.01	229	4	1.8
Linux 2.4.37	ジャイアントロック，細粒度ロック	220,396	1,076	0.49	372	88	23.7
Linux 2.6.39	細粒度ロック	737,679	5,033	0.68	1,567	330	21.1

また，排他制御箇所は，調査対象となるソースコードからデータをロックとアンロックする関数（Linux のスピンロックであれば，“spin_lock”と“spin_unlock”）を含む行，またはファイルの総量を算出した．

5.1.2 評価結果

Tender，FreeBSD，および Linux における行数とファイル数についての排他制御量を表 1 に示す．評価は，各 OS のバージョンの違いによる差異，また粗粒度ロック，および細粒度ロックにおける OS 間での差異について比較する．まず，***Tender***-Rid は，***Tender***-Rkind に比べ，行数における排他制御量が約 5.1 倍に増加している．***Tender***-Rid は，資源インタフェース制御において同一種類内の資源の通番ごとに排他制御しているため，資源管理表内の通番ごとのデータ以外の共有データは，排他制御できない．このため，***Tender***-Rid は，資源管理表内の共有データをアクセス時ごとに排他制御しているため，***Tender***-Rkind に比べ，排他制御量が増加している．資源管理表内の共有データを利用するファイルは，資源の種類ごとに存在しているため，ファイル数における排他制御量も増加している．

次に，FreeBSD は，FreeBSD 3.5-R から FreeBSD 6.4-R への改版時において行数における排他制御量が急激に増加している．これは，FreeBSD 3.5-R は，ジャイアントロックにより排他制御されており，FreeBSD 6.4-R に改版したとき，ジャイアントロックが細粒度ロックに置き換えられたためである．一方，FreeBSD 6.4-R から FreeBSD 8.4-R に改版したとき，排他制御量の増加は少ない．これは，実行管理やメモリ管理といったカーネルの主要機能における排他制御の細粒度化は，FreeBSD 6.4-R において十分完了していることを示している．また，ファイル数における排他制御箇所数も行数における排他制御箇所数と同様に増加している．

次に，Linux は，Linux 2.0.40 から Linux 2.4.37 の改版時において行数における排他制御量が急激に増加している．これは，FreeBSD と同様に Linux 2.0.40 で利用していたジャイアントロックが Linux 2.4.37 において細粒度ロックに置き換えられたためである．また，Linux 2.4.37 から Linux 2.6.39 の改版時において行数における排他制御量が約 4.7 倍に増加している．この際，Linux 2.6.39 における，ディレクトリごとの排他制御量は，kernel/の割合が最も高く約 45%であり，次に割合が高いディレクトリは，mm/で約 17%となっている．kernel/や mm/には，実行管理やメモリ管理といったカーネルの主要機能のファイルが含まれているため，Linux 2.6.39 においてカーネルの主要機能部分の排他制御が細粒度化されているといえる．

次に，粗粒度ロックを利用した OS として ***Tender***-Rkind，FreeBSD 6.4-R，および Linux 2.4.37 を比較すると ***Tender***-Rkind の排他制御量が最も少ない．この際，***Tender***-Rkind は，FreeBSD 6.4-R より約 95%，Linux 2.4.37 より約 88%だけ行数における排他制御量を抑制している．また，割合の観点からも ***Tender***-Rkind が最も行数における排他制御量の割合が少ないことが分かる．

さらに，細粒度ロックを利用した OS として ***Tender***-Rid，FreeBSD 8.4-R，および Linux 2.6.39 を比較すると ***Tender***-Rid の排他制御量が最も少ない．この際，***Tender***-Rid は，FreeBSD 8.4-R より約 77%，Linux より約 87%だけ行数における排他制御量を抑制している．また，割合の観点からも ***Tender***-Rid が最も行数における排他制御量が少ないことが分かる．この際，***Tender***-Rid は，Linux 2.6.39 と比べ，約 1/2 の割合で細粒度ロックを実現している．

以上より，***Tender***-Rkind と ***Tender***-Rid は，排他制御量を FreeBSD や Linux よりも抑制しているといえる．

5.2 既存 OS との比較評価

5.2.1 評価方式

本評価では，OS の 3 種類の機能についてマイクロベンチマークプログラムを使用し，処理の並列性を評価する．評価環境を**表 2** に示す．マイクロベンチマークプログラムは，評価対象機能を実行するプロセスを

表 2　評価環境

OS	***Tender***-Rkind, ***Tender***-Rid FreeBSD 6.4-R, FreeBSD 8.4-R Linux 2.4.37, Linux 2.6.39
CPU	3.4 GHz（4 コア）
RAM	8,192 MB

任意の数だけ各コア上に配置し，実行する．この際，各プロセスは，コア番号が 0 のコアから順に配置する．各コアに配置したプロセスが処理を開始してから，すべてのプロセスが処理を終了するまでの処理時間を測定する．また，***Tender*** は，***Tender*** 独自のカーネルコールにより評価対象機能を実現する．

評価では，コア数を 1 コアから 4 コアに増加させた場合の処理時間の削減率を評価する．各 OS における処理時間の削減率を比較するため，評価結果は，1 コアを基準とした相対値で示す．以下に評価対象機能について述べる．

(1) プロセス生成

プロセス生成の評価では，fork() と execve() により，プロセスを生成する処理を繰り返す評価用プロセスを各コアに配置し，実行する．***Tender*** は，資源「プロセス」と「演算」[13] のカーネルコールにより，FreeBSD や Linux と同等の処理を実行するプロセスの生成を実現する．

生成するプロセスは，起動後，即座に return 文により終了するプロセスであり，***Tender*** では，テキスト部が約 2,000 B，データ部が 0 B，BSS 部が 0 B のプログラムであり，FreeBSD と Linux では，テキスト部が約 1,000 B，データ部が約 200 B，BSS 部が約 10 B のプログラムである．評価では，上記の評価用プロセスを 24 プロセス起動し，処理回数 100 回で実行する．

(2) メモリの確保と解放

メモリの確保と解放の評価では，mmap() と munmap() により，メモリの確保と解放の処理を繰り返す評価用プロセスを各コアに配置し，実行する．***Tender*** は，資源「仮想領域」と「仮想ユーザ空間」のカーネルコールにより，メモリの確保と解放を実現する．

メモリの確保と解放は，4 KB のメモリ領域の確保と解放を繰り返す．評価では，上記の評価用プロセスを 24 プロセス起動し，処理回数 100 回で実行する．

(3) プロセス間通信

プロセス間通信の評価では，pipe() によるプロセス間通信を繰り返す評価用プログラムを各コアに配置し，実行する．pipe() によるプロセス間通信では，write() により，データを送信し，read() により，データを受信する．***Tender*** は，資源「コンテナ」と「コンテナボックス」のカーネルコールにより，プロセス間通信を実現する．

プロセス間通信は，2 つのプロセス間において 4 KB のデータの送受信を 100 回繰り返す．また，プロセス間通信を実行するプロセスグループを 24 グループ，つまり送受信のプロセスを合計して 48 プロセスだけ実行する．この際，送信プロセスと受信プロセスの組は，同一コア上で実行する．

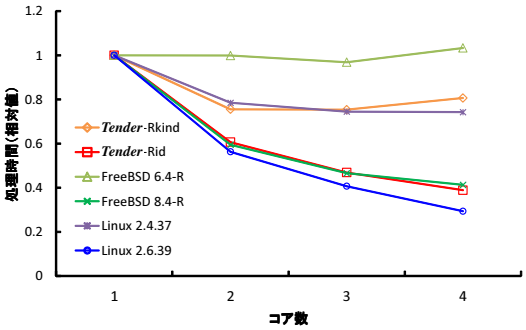


図 8　プロセス生成の処理時間

5.2.2　プロセス生成の評価結果

評価結果を**図 8** に示す．まず，FreeBSD 6.4-R は，コア数が増加しても処理時間は，削減していないことが分かる．これは，FreeBSD 6.4-R は，プロセスに対して各コアへ負荷分散を行う機能が提供されていないため，評価用プロセスが 1 つのコア上でのみ実行されているからである．

次に，***Tender***-Rkind と Linux 2.4.37 は，同程度の処理時間削減率である．***Tender*** では，プロセスの生成において資源「プロセス」と「演算」の 2 種類の資源を利用する．***Tender***-Rkind は，資源の種類ごとに排他制御しているため，***Tender***-Rkind における本評価では，2 種類の資源を並列に操作できる．このため，2 コアの場合において処理時間の削減率は，最も高くなり，約 24.5%となっている．その後，コア数の増加につれて処理時間の削減率は低下し，4 コアの場合，約 19.3%となる．

また，***Tender***-Rid は，FreeBSD 8.4-R と同程度の処理時間削減率であり，4 コアの場合，***Tender***-Rid は約 61.1%，FreeBSD 8.4-R は約 58.7%となっている．Linux 2.6.39 は，***Tender***-Rid と FreeBSD 8.4-R よりも処理時間の削減率は高く，約 70.6%となっている．***Tender***-Rid は，資源の種類と通番ごとに排他

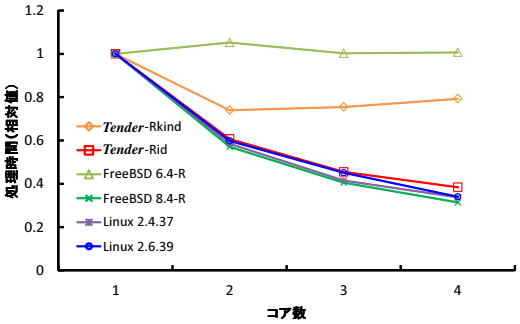


図 9 メモリの確保と解放の処理時間

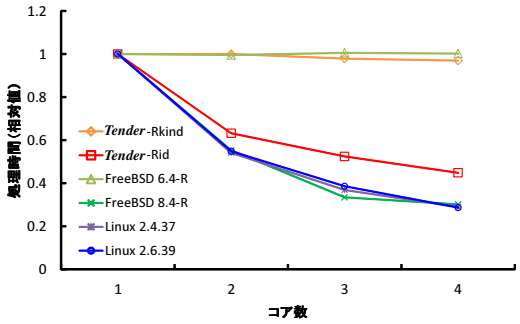


図 10 プロセス間通信の処理時間

制御するため，***Tender***-Rkind に比べ，資源操作の競合を抑制できる．しかし，資源の生成に伴う資源名管理部の排他制御がボトルネックとなるため，Linux 2.6.39 よりもコア数の増加に対して処理時間の削減率が低下している．

5.2.3 メモリの確保と解放の評価結果

評価結果を**図 9** に示す．まず，FreeBSD 6.4-R は，プロセス生成と同様の理由により，処理時間が削減されなかった．次に，***Tender***-Rkind のコア数の増加に対する処理時間削減率は，他 OS に比べて低い．***Tender***-Rkind では，メモリの確保と解放において，資源「仮想空間」と「仮想ユーザ空間」の 2 種類の資源を利用する．このため，プロセス生成と同様に，***Tender***-Rkind における本評価では，2 種類の資源を並列に操作できる．したがって，2 コアの場合において処理時間の削減率は，最も高くなり，約 26.0%となっている．その後，コア数の増加につれて処理時間の削減率は低下し，4 コアの場合，約 20.8%となる．***Tender***-Rkind では，資源の種類ごとに排他制御するため，並列に扱う資源の種類が増えれば，コア数の増加に対する処理時間削減率も高くなると予想される．

また，その他の OS は，コア数の増加に対して同程度の処理時間削減率となっている．4 コアの場合，***Tender***-Rid は約 61.6%，FreeBSD 8.4-R は約 68.6%，Linux 2.4.37 は約 66.2%，Linux 2.6.39 は，約 66.0%となっている．

5.2.4 プロセス間通信の評価結果

評価結果を**図 10** に示す．まず，FreeBSD 6.4-R と ***Tender***-Rkind は，コア数が増加しても処理時間が削減されていない．FreeBSD 6.4-R は，プロセス生成と同様の理由により，処理時間が削減されなかった．また，***Tender*** におけるプロセス間通信では，データの送受信において資源「コンテナ」のカーネルコールのみを利用するため，プロセス間通信におけるデータの送受信は，1 種類の資源のみで実現される．このため，***Tender***-Rkind は，プロセス間通信におけるデータの送受信において資源「コンテナ」を利用する際，競合が発生し，常に 1 つのコアでしか送受信処理を実行できない．このため，並列に処理を実行できず，処理時間を削減できない．

また，***Tender***-Rid は，FreeBSD 8.4-R，Linux 2.4.37，および Linux 2.6.39 と比べ，コア数の増加に対する処理時間の削減率が低い．4 コアの場合において ***Tender***-Rid は，約 55.2%の削減率，FreeBSD 8.4-R，Linux 2.4.37，および Linux 2.6.39 は，約 70%の削減率となっている．***Tender***-Rid は，資源インタフェース制御で排他制御できない資源管理表内の共有データに対する排他制御がボトルネックとなっているため，FreeBSD 8.4-R，Linux 2.4.37，および Linux 2.6.39 よりもコア数の増加に対する処理時間の削減率が低下している．

5.2.5 排他制御箇所数あたりの処理時間削減率の評価

1 コアから 4 コアにコア数が増加した場合の排他制御箇所数あたりの処理時間削減率を**表 3** に示す．表 3 より，***Tender***-Rkind は，プロセス生成とメモリの確保と解放において排他制御箇所数あたりの処理時間削減率が最も高い．これは，***Tender***-Rkind は，処理時間削減率において ***Tender***-Rid，FreeBSD 8.4-R，および Linux 2.6.39 に劣るが，これらの OS に比べ，排他制御量が少ないためである．しかし，***Tender***-Rkind は，プロセス生成とメモリの確保と解放の性能評価より，3 コア以上で性能が向上していないため，コア数の増加に対して有用な方式ではないといえる．また，プロセス間通信では，***Tender***-Rkind は，処理時間を削減できなかったため，排他制御箇所数あたりの処理時間削減率が低くなっている．

Linux 2.4.37 は，プロセス生成において処理時間の

表 3　1→4 コアへのコア数の増加における排他制御箇所数あたりの処理時間削減率（%）

OS の種類	ベンチマークの種類		
	プロセス生成	メモリの確保と解放	プロセス間通信
Tender-Rkind	0.00151	0.00162	0.00024
Tender-Rid	0.00094	0.00094	0.00085
FreeBSD 6.4-R	-0.00001	-0.00001	-0.00001
FreeBSD 8.4-R	0.00021	0.00024	0.00025
Linux 2.4.37	0.00024	0.00062	0.00066
Linux 2.6.39	0.00014	0.00013	0.00014

削減率が低かったため，排他制御箇所数あたりの処理時間削減率が他の機能に比べ，低くなっている．しかし，メモリの確保と解放やプロセス間通信は，***Tender***-Rid に次いで排他制御箇所数あたりの処理時間削減率が高い．一方，FreeBSD 8.4-R と Linux 2.6.39 は，各機能において排他制御箇所数あたりの処理時間削減率は，同程度の値となっている．しかし，Linux 2.6.39 の修正量は，FreeBSD 8.4-R よりも多いため，Linux 2.6.39 の方がより値が小さくなっている．

一方，***Tender***-Rid は，Linux 2.6.39 に比べ，排他制御箇所数あたりの処理時間削減率が約 6.7 倍となっている．このため，***Tender***-Rid の方式を用いて排他制御箇所を局所化して排他制御することは，有用であると推察できる．また，同様に FreeBSD 8.4-R と比較した場合，排他制御箇所数あたりの処理時間削減率が約 4.0 倍となっており，FreeBSD 8.4-R と比べ，約 1/4 の修正量で同程度だけ性能を向上している．以上より，***Tender***-Rid は，他 OS に比べ，排他制御箇所数あたりの処理時間削減率が高く，少ない修正量で性能を向上させることができるといえる．

6. おわりに

Tender において排他制御を局所化し，かつカーネル処理の並列性を向上させるマルチコア対応方式について述べた．マルチコア対応にあたり，OS の資源を一括して管理している ***Tender*** 特有の OS 構造である資源インタフェース制御に基づき資源を排他制御することで，排他制御箇所を資源インタフェース制御に局所化した．また，資源インタフェース制御が有する情報として資源種別単位と資源識別子単位の 2 種類のロック粒度の排他制御方式を実現した．

修正量の評価では，排他制御量の観点より，評価を行った．細粒度ロックにより実現された既存 OS と比較すると細粒度ロックとして資源識別子排他制御を実現したマルチコア ***Tender*** は，行数における排他制御量が FreeBSD 8.4-R よりも約 77%，Linux 2.6.39 よりも約 87%だけ少ないことを示した．

性能評価では，細粒度ロックにより実現されたマルチコア ***Tender*** は，コア数の増加に対する処理時間削減率がプロセス生成とメモリの確保と解放において，FreeBSD 8.4-R や Linux 2.6.39 と同程度であることを示した．また，排他制御箇所数あたりの処理時間削減率の観点では，細粒度ロックにより実現されたマルチコア ***Tender*** は，既存 OS よりもプロセス生成，メモリの確保と解放，およびプロセス間通信において，コア数の増加に対する修正量あたりの処理時間削減率が高いことを示した．

残された課題として資源識別子排他制御方式における資源管理表内の共有データの削減がある．

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)，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. and Vora, S.: A 45 nm 8-Core Enterprise Xeon Processor, *IEEE J. Solid-State Circuits*, Vol. 45, No. 1, pp. 7–14 (2010).
3) 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, *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, pp. 1–16 (2010).
4) Lehey, G.: Improving the FreeBSD SMP implementation, *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pp. 155–164 (2001).
5) Wentzlaff, D. and 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).
6) Yuan, Q., Zhao, J., Chen, M. and Sun, N.: GenerOS: An Asymmetric Operating System Kernel for Multi-core Systems, *Parallel & Dis-*

tributed Processing, 2010 IEEE International Symposium on, pp. 1–10 (2010).
7) Ballesteros, F. J., Evans, N., Forsyth, C., Guardiola, G., McKie, J., Minnich, R. and Soriano-Salvador, E.: NIX: A Case for a Manycore System for Cloud Computing, *Bell Labs Technical Journal*, Vol. 17, No. 2, pp. 41–54 (2012).
8) Liedtke, J.: Toward Real Microkernels, *Communications of the ACM*, Vol. 39, No. 9, pp. 70–77 (1996).
9) 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, *SIGMETRICS '08: Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, Vol. 36, No. 1, pp. 229–240 (2008).
10) Regehr, J. and Duongsaa, U.: Preventing Interrupt Overload, *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, Vol. 40, No. 7, pp. 50–58 (2005).
11) 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).
12) 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).
13) 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).