Vol.2012-OS-122 No.3
2012/8/1

# Mintオペレーティングシステムにおけるコア管理機能の実現

池田 騰[1]　乃村 能成[1]　谷口 秀夫[1]

**概要**：
計算機資源を有効に活用するために，複数の OS を 1 台の計算機で動作させる技術として仮想計算機方式が研究されている．しかし，仮想計算機方式は，計算機の仮想化によるオーバヘッドのため，実計算機よりも性能が低下する．また，OS 間での処理負荷の影響が存在する．そこで，性能の低下を抑えるため，マルチコアプロセッサ上で複数の Linux を独立に走行させる方式として Mint が研究開発されている．Mint では，静的に計算機資源を割り当てている．しかし，OS の負荷は一定ではないため，負荷に応じて計算機資源を割り当てたいという要求がある．本稿では，Mint において，OS 間でのコアの移譲を可能にするコア管理機能について，課題と対処を述べ，実現する．

# Core Manegement Function on Mint Operating System

Noboru Ikeda[1]　Yoshinari Nomura[1]　Hideo Taniguchi[1]

***Abstract:***
Virtualization is a novel technology to run multiple commodity OSes on a single PC. However, virtualization has considerable performance overheads. To solve this performance issue, we have implemented the Mint operating system, which runs multiple Linux instances by partitioning physical cores, memory blocks, and I/O devices (interruptions). Although, non-virtualization feature of Mint makes the most efficient use of available cores, Mint does not have any elastic/dynamic features on resource assignment. In this paper, we describe a new dynamic core management scheme of Mint and its concrete implementation.

## 1. はじめに

計算機性能の向上に伴い，1 台の計算機上に複数のオペレーティングシステム (以降，OS と呼ぶ) を走行させる研究が活発に行われている．代表的なものとして，仮想計算機 (VM : Virtual Machine) を利用する方式 (VM 方式) があり，代表的な研究として，Xen[1] や VMware[2] がある．VM 方式は，仮想マシンモニタ (VMM : Virtual Machine Monitor) により，実計算機に複数の仮想計算機を動作させ，この上でゲスト OS をそれぞれ独立して走行させる．しかし，VM 方式は，デバイスの仮想化によるオーバヘッドが発生するほか，VM と VMM に依存性があり，実計算機と同等の性能では動作しない．そこで，仮想化による性能の低下を防ぐため，我々は，マルチコアプロセッサ上で複数の Linux を独立に走行させる方式として Mint オペレーティングシステム [3] を研究開発している．Mint では，1 台の計算機ハードウェアにおいて，CPU やメモリ，入出力機器といった各資源について静的に分配し，各 OS で独立に管理を行う．これにより，仮想化によらずに各 OS を実計算機上で直接走行させる．このため，Mint では各 OS が実計算機とほぼ同等の性能で走行可能である．Mint と同様に各 OS に計算機資源を静的に分割することで複数の OS を走行させる方式として，SIMOS[4][5] や Twin-Linux[6] が存在する．Mint，SIMOS，および Twin-Linux は計算機資源を各 OS の起動時に静的に割り当てており，動的に割り当てることを検討していない．

OS の負荷は一定ではないため，OS の負荷に応じて計算

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

機資源を動的に割り当てたいという要求がある．Mint では，この要求を満たすために，各 OS の独立性を維持したまま計算機資源を移譲可能にするという課題がある．この課題の対処の 1 つとして，Mint において，OS 間でのコアの移譲を可能とするコア管理機能を実現する．

本稿では，Mint においてコア管理機能を実現する上での問題と課題を列挙し，対処を述べる．そして，対処を実装し，コア管理機能を実現する．

## 2. Mint の特徴と目的

### 2.1 特徴

Mint は，仮想化によらず 1 台の計算機上で複数 OS を動作させる技術である．Mint では，CPU をコア単位で分割し，OS 毎に割当てる．Mint の構成例を**図 1** に示し，CPU 分割方法を以下で説明する．Mint では，各 OS の起動時にそれぞれの OS が占有するコア数を静的に決定することで OS 間でのコアの分割を行っている．この際，最初に起動する OS1 はコア 0 を Boot Strap Processor(以下，BSP) とする．後に起動する OS1 および OS2 はそれぞれ起動時に指定したコア 2 とコア 3 を BSP とする．これにより，Mint では，全 OS(図中では OS1 から OS3) が互いに独立した Linux として動作する．つまり，Mint は，Multikernel [7] のようにチップ内分散処理を指向した構成を取りながら，それぞれがフルセットの Linux として動作可能である．したがって，将来の CPU のメニーコア化に対する柔軟性を持ちつつ，仮想化のオーバヘッドを排し，また，既存アプリケーションが Linux 上でそのまま動作するといった特徴を持っている．

### 2.2 目的

現在の Mint では，コアの分配を起動時に静的に決定している．チップ内分散処理の観点からいえば，コアの分配は起動後に動的に変更できることが望ましい．そこで，起動後にコアの再分配が可能なコア管理機能を実現したい．

オリジナルの Linux には，CPU ホットプラグという，動的に CPU コアを着脱する機能がある．しかしながら，この機能は単一の Linux が全資源を管理していることが前提になっているため，そのままでは Mint に適用できない．そこで，本稿では，CPU ホットプラグを Mint に導入する際の問題点と対処を明かにし，実装を示す．

## 3. 問題点

コア管理機能を実現する際の問題点として，以下の 3 つが存在する．

**(問題 1)** ローカルタイマ割り当ての重複

Linux では，各コアのタイマ割り込みの供給源として High Precision Event Timer(以下，HPET) を利用している．Linux は，HPET0, 1 をグローバルタイマとユーザ空間用のイベントタイマとし，残りの HPET2 以降を各コア用として分配する．

Linux は，起動時に利用可能なタイマを全て検出，初期化し，HPET 管理表に登録する．起動時や CPU ホットプラグなどのコア初期化時には，管理表を参照し，未使用の HPET を順に利用する．

Mint では，各 OS の HPET 管理表間で整合性を保つために，OS 毎で利用しない HPET を静的に決定していた．これが動的なコア分配の際に問題となる．

**図 2** に従来の Mint における HPET の割り当て例を示し，説明する．従来の Mint では，HPET の競合を防ぐために，後から起動した OS は自身の BSP が何番目のコアであるかをオフセットとして持ち，オフセット分を検索対象外としていた．図は，OS2 が HPET2 を避けて HPET3 を利用中，同様に OS3 が HPET4,5 を利用中の様子を示している．コアの割り当てが静的な場合，この処理で他の OS が利用する HPET を避けられた．しかし，コアの割り当てを動的にした場合，問題となる．例として，OS3 から OS1 にコア 3 を移譲することを考える．OS1 はコア 3 を初期化時に新たに HPET を割り当てようとする．この際，OS1 は HPET3 以降を利用可能としているため，OS2 の HPET3 を横取りしてしまう．

**(問題 2)** 各種識別 ID の重複

Linux がはコアに対する ID として，Local APIC ID (以下 LAPIC ID)，コア ID，論理 APIC ID の 3 つを持っている．LAPIC ID は，OS 起動前に設定され固

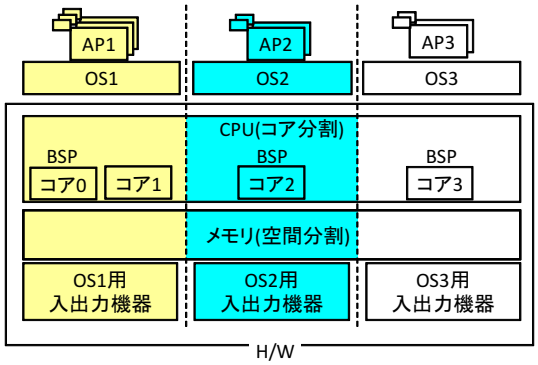



**図 1** Mint の構成例

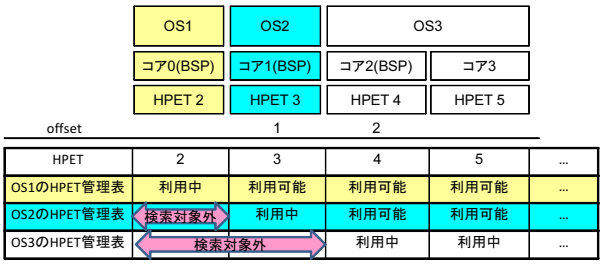
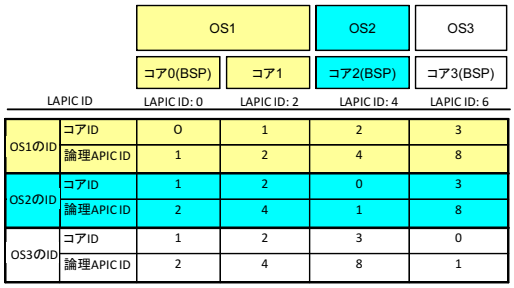



**図 2** 従来の Mint における HPET の割り当て例




| | | | | | |
|---|---|---|---|---|---|
| OS1のID | コアID | 0 | 1 | 2 | 3 |
| | 論理APIC ID | 1 | 2 | 4 | 8 |
| OS2のID | コアID | 1 | 2 | 0 | 3 |
| | 論理APIC ID | 2 | 4 | 1 | 8 |
| OS3のID | コアID | 1 | 2 | 3 | 0 |
| | 論理APIC ID | 2 | 4 | 8 | 1 |

**図 3** Linux におけるコア識別子の割り当てをそのまま複数 OS に割り当てた例

定であるため，重複は起こらないが，コア ID，論理 APIC ID の重複が問題となる．

Linux における両者の役割と，Mint における問題点について説明する．

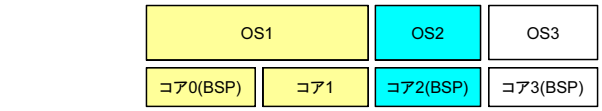
(1) コア ID

コア ID は，Linux が自身の占有するコアの識別のために広く使用する ID である．BSP はコア ID 0 でなければならず，以降コアの検出順に 1,2,··· と設定する．Mint では，OS 毎に BSP が存在するため，コア ID 0 が複数のコアに付与される．その結果，例えば，**図 3** では，コア 2 のコア ID は，2，0，3 と OS ごとに異なっている．コア ID は，OS 内でのみ使用する ID であるため，各 OS で重複していていも問題にはならないが，CPU ホットプラグの際には，この重複が問題となる．

(2) 論理 APIC ID

論理 APIC ID は，割り込みの通知先を指定する際に用いられる ID である．デバイスからの割り込みやコア間割り込みのために全コアで一意でなければならない．Linux では，1 をコア ID だけビットシフトした値を論理 APIC ID としている．そのため，コア ID がの重複を許していた従来の Mint では，以下のように重複を回避していた．**図 4** を例に，従来の Mint での静的な解決方法を説明する．

OS1 より後から起動した OS2 は，自身の BSP が何番目のコアであるかをオフセット (図中では 2) として持ち，BSP の論理 APIC ID を 2 のオフセット乗とすることによって BSP について OS1 の論理 APIC ID (4) と一致させる．ここで，OS2 がコア 0 に対して論理 APIC ID 2 を付与しており，これは，OS1 のコア 1 に対して付与した値と重複している．しかし，OS2 がコア 0 を扱うことはないので，これは，問題とならない．すなわち，自身の占有するコアに付けられた論理 APIC ID が他 OS の占有するコアに付与されていなければ問題とならない．ここで避けたいケースは，OS2 がコア 2 に対して，OS1 が既に使用している 1 または 2 を付与することである．説明した手法は，このケースを避けて要件を満たしている．




| | | | | |
|---|---|---|---|---|
| OS1の論理APIC ID | 1 | 2 | 4 | 8 |
| OS2の論理APIC ID | 2 | 4 | 4(2^offset) | 8 |
| OS3の論理APIC ID | 2 | 4 | 8 | 8(2^offset) |

**図 4** 従来の Mint における論理 APIC ID の算出例

この手法は，各 OS の起動時に静的に前から順番にコアを割り当てていることに依存している．つまり，各 OS は自身の BSP よりも若番側のコアを占有しない事実を利用して，論理 APIC ID の重複を避けている．しかも，全論理 APIC ID の重複を避ける手法ではない．したがって，コアの割り当てを動的にした場合，手法の前提は崩れ，問題となる．

**(問題 3)** OS 間にまたがるコア管理機構の欠如

Linux では，OS 内で閉じた ID で各コアを識別しており，CPU ホットプラグの際にもこの値を利用する．当然ながら，OS 間でコアをやりとりすることを想定していないため，他の OS がどのコアを占有しているといった情報を管理することもない．Mint においても，コアの割り当て状況は静的に固定されており，各 OS は独立にコアを管理するため，OS 間でコアの利用状況を統一的に管理する機構もない．OS 間でコアの移譲を行うためには，他の OS が占有するコアの情報を管理する必要がある．

これらの問題を解決するために，新たなローカルタイマの割り当て方式，OS 間でコアを一意に識別する方式，および OS 間にまたがるコアの利用状況管理の枠組みが必要となる．

## 4. 課題

3 章で述べた問題点を解決するためには，以下の 3 つの課題が存在する．

**(課題 1)** HPET の割り当て方式の変更

3 章の (問題 1) の対処のため，HPET を各 OS に重複して割り当てることがないように HPET の割り当て方式を変更する．

**(課題 2)** OS 間でのコア識別方式の導入

OS 間でのコアの移譲を可能とするためには，同一のコアに対しては同一の論理 APIC ID で識別する必要がある．そこで，OS 間でのコアの識別方式を導入し，論理 APIC ID について OS 間で整合性を保つ．

**(課題 3)** コア利用状況管理機能の導入

OS 間でのコアの移譲を可能とするためには，各コア

の利用状況を管理する必要がある．しかし，もともと Linux は複数の OS で走行することを想定していないため，他の OS が持つコアを確認する手段を持っていない．また，OS はコア ID を元に自身の占有しているコアを管理しており，このコア ID は OS によって異なる．このため，OS 間をまたがり，統一のコア識別子を用いてコアの利用状況を管理する機能を導入する必要がある．このため，以下の 2 つの機能を導入する．

**(機能 1)** コアの占有状況を管理する機能

OS が全コアの占有状況を把握可能にするために，コアの占有状況について管理するための機能である．

**(機能 2)** コア識別子変換機能

OS 間でコアの識別に対する整合性を保つための機能である．

### 4.1 HPET の割り当て方式の変更

各 OS 間で HPET の利用状況を共有するためには，OS 間通信機能を利用して HPET の利用状況を他の OS に伝える方法が考えられる．しかし，OS 間の独立性が低下してしまう．そこで，各コアに一意に HPET を割り当てるルールを統一することで対処する．具体的には，OS 外で付与される一意な LAPIC ID と HPET を 1 対 1 で対応させ，各コアは自身の持つ LAPIC ID に対応する HPET のみを利用するように変更する．

### 4.2 OS 間でのコア識別方法

OS 間でコアの識別に関する整合性を保つ対処案として，以下の 2 つの方式が存在する．

**(対処案 1)** コア ID 統一方式

Linux のコア ID 決定規則を変更し，全 OS で同じコアには同じコア ID を付与する方法である．この方式の利点として，OS 間でのコア識別が Linux の枠組み内の識別方式と一致するため，OS 間でのコア移譲や資源管理がしやすい．また，論理 APIC ID はコア ID から決定されるため，論理 APIC ID も自然に統一される．この方式の欠点は，実装の工数が高いことである．もともと Linux では BSP の異なる複数の OS が同時に走行することを想定していないため，Linux はコア ID 0 を BSP とする記述が随所にある．このため，コア ID 0 を BSP と想定している記述を漏れなく改変する工数は大きい．

**(対処案 2)** 論理 APIC ID の算出規則変更方式

コア ID の重複を許可するものの，コア ID から論理 LAPIC ID への算出規則を変更することで，論理 APIC ID の重複を防ぐ方式である．もともと，コア ID は OS 内でのみ使われる閉じた ID であるため，OS 間で重複しても問題ない．このため，この方式では，論理 APIC ID の重複のみを防ぐ対処とする．この方式の利点として，改変箇所が論理 APIC ID 算出部分のみとなり，OS の改変箇所を局所化できる．この方式の欠点として，コア ID は統一されないため，OS 間でコアを移譲する際に別の識別方式が必要となる．これについては，4.4 節で述べる．

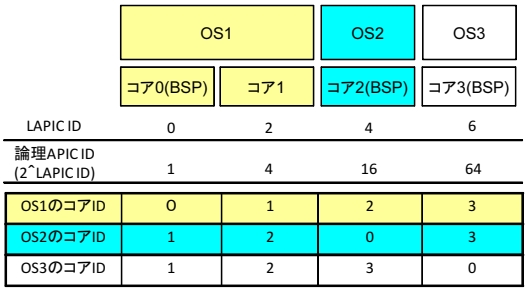



**図 5** (対処案 2) を採用後の Mint におけるコア識別子の割り当て例

OS 間でコアの識別に関する整合性を保つ方法としては，Linux の枠組みの内でコアの識別ができる (対処案 1) の方が望ましい．しかし，(対処案 1) はカーネルに対する改変量が大きく，実装が困難である．そこで，OS の改変を局所化，極小化できる (対処案 2) を採用する．

(対処案 2) を採用するにあたり，論理 APIC ID の算出規則で，OS 間で一意な値をもつ LAPIC ID を用いるように変更する．具体的には，コア ID から直接論理 APIC ID に変換する方式から，コア ID から一度 LAPIC ID に変換し，この LAPIC ID を論理 APIC ID に変換する．**図 5** に (対処案 2) を採用した Mint におけるコア識別子の割り当て例を示し，説明する．OS1，OS2，および OS3 において，コア 2 のコア ID はそれぞれ 2，0，3 と異なる．各 OS はそれぞれ独自に自身の持つコア ID と LAPIC ID の対応表を持つため，これを用いてコア ID から LAPIC ID に変換する．これにより，コア 2 に関して，各 OS で LAPIC ID 4 という識別子で統一される．この LAPIC ID をもとに論理 APIC ID を算出することで各 OS の論理 APIC ID を統一できる．図 5 では，2 の LAPIC ID 乗を論理 APIC ID とし，LAPIC ID 4 から論理 APIC ID 16 を算出している．論理 APIC ID は，割り込み時のビットマスクとして使用されるため，LAPIC ID ごとに 1bit を割り当てる形で算出している．

### 4.3 コア利用状況管理

4 章で述べた (課題 3) への対処として，以下の 2 つの方式が考えられる．

**(方式 1)** 特定の OS によるコアの集中管理

特定の OS(以下，管理 OS) がコアの占有状況の管理を行う．

**図 6** に示すように，コア移譲の際には管理 OS が移譲元の OS と移譲先の OS にそれぞれの OS にコアの解放と占有を命令する．

**表 1** 各管理方式の利点と欠点

| 管理方式 | 利点 | 欠点 |
|---|---|---|
| 管理 OS によるコアの集中管理 | (1) 管理 OS 以外の OS はコアの把握が不要 | (1) OS 間に依存関係が存在 |
| 各 OS によるコアの分散管理 | (1) 他の OS に依存しない | (1) 全 OS でコアの把握が必要<br>(2) OS 間で排他制御が必要 |

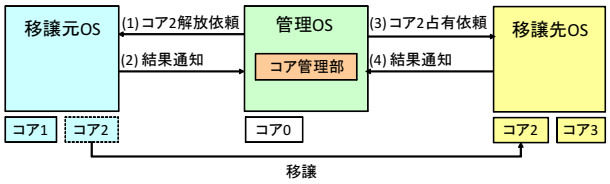



**図 6** 管理 OS によるコアの集中管理を行う場合のコアの移譲機能の実現例

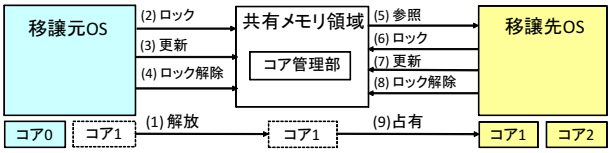



**図 7** 各 OS によるコアの分散管理を行う場合のコア移譲の実現例

**(方式 2)** 各 OS によるコアの分散管理

コアの占有状況を格納したコア管理部を各 OS とは独立に作成し，OS ごとにコア管理部を用いてコアを管理する．まず，コア管理専用の実メモリ領域を用意し，各 OS から操作可能な共有メモリ領域とする．この共有メモリ領域にコア管理部を配置する．この方式では，**図 7** に示すように，コア移譲では移譲元 OS と移譲先 OS がそれぞれ以下の処理を行う．

**(移譲元 OS)** コアを解放して占有可能なコアとして管理部に登録

**(移譲先 OS)** 占有可能なコアを管理部から確認して占有

対処として，(方式 2) の各コアによる分散管理方式を採用する．(方針 1) と (方針 2) の利点と欠点を**表 1** に示し，以下で説明する．(方針 1) 管理 OS によるコアの集中管理では，共有メモリ領域を作成する必要がなく，OS 間でのコア管理部の排他制御を行う必要がない．ただし，OS 間に依存関係があり，単一障害点が存在する．これに対し，(方針 2) 各 OS によるコアの分散管理では，OS 間に依存関係がなく，単一障害点が存在しない．ただし，OS 間で共通で使用する領域があるため OS 間で排他制御が必要となる．

(方針 1) のように，OS 間で依存関係がある場合，依存先の OS が不具合を起こした場合に全ての OS に影響を与えることになる．このように，1 つの OS の不具合が全体に影響を与えることは問題である．一方，(方針 2) では，全ての OS でコアの占有状況を把握する必要があり，OS 間での排他制御によるオーバヘッドも問題となる．しかし，コア管理部を使用する頻度は低いと想定されるため，コア管理部に対する排他制御の回数は多くならないと予想される．このため，排他制御によるオーバヘッドは大きくならない．これらの点を踏まえると，各 OS によるコアの分散管理は，全ての OS でコアの把握が必要となるものの，OS 間で依存関係が存在しないため，この課題の対処に適している．

### 4.4 コア識別子変換機能の導入

4 章で述べた (課題 2) への対処として，以下の 2 つの対処案が考えられる．

**(対処案 1)** 全ての OS で同じコア ID を用いるように改変する．

**(対処案 2)** 全ての OS で一意に設定されるコア識別子を用意し，コア ID の代わりにコア移譲の際にのみ用いる．

(対処案 1) と (対処案 2) に関する検討内容は，3 章の (課題 2) と同様である．(対処案 1) による，OS 間でのコア ID の統一は実現が難しいため，(対処案 2) を採用する．全ての OS で一意に設定されるコア識別子には，4.2 節の対処と同様に LAPIC ID を用いる．

## 5. 実現

### 5.1 コア管理機能の基本構成

**図 8** にコア管理機能を実現した際の，Mint におけるコア移譲の例を示し，以下で Mint におけるコア管理機能の構成要素とコア移譲の手順を説明する．まず，コアを解放する際の手順を以下で示す．

**(A)** コア解放 AP

**(1)** 解放するコア数を引数として，ユーザがコア解放 AP を実行する．

**(2)** コア移譲インタフェース部を呼び出し，解放するコアの検索を依頼する．

**(B)** コア移譲インタフェース部

**(3)** 解放可能なコアを検索し，検索結果から解放するコアを決定する．解放するコアのコア ID をコア解放 AP に返す．

**(A)** コア解放 AP

**(4)** 解放するコアに対し，CPU ホットプラグ機能を用いてコアを解放する．

**(5)** コア移譲インタフェース部に解放したコアについてコア管理部の更新を依頼する．

**(B)** コア移譲インタフェース部

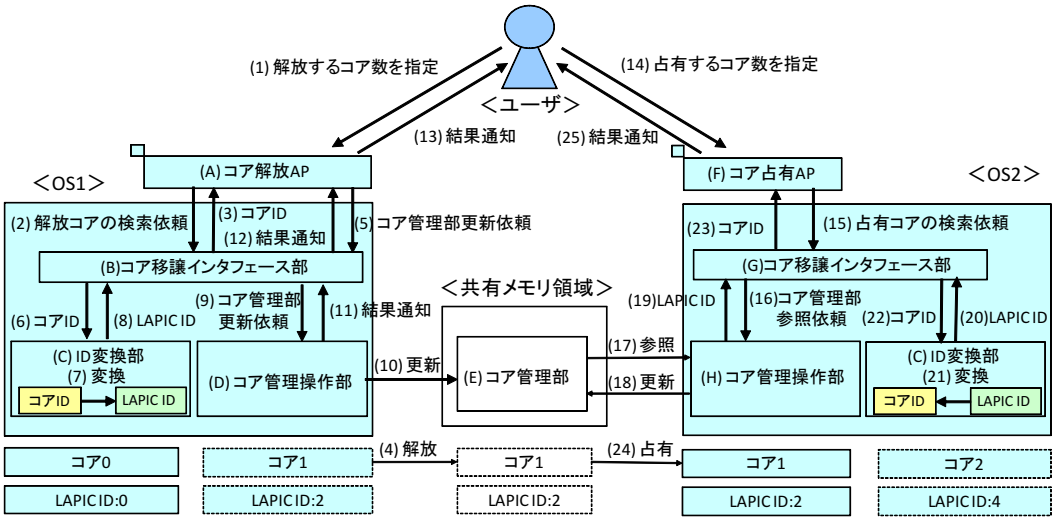



**図 8** Mint におけるコア移譲の例

**(6)** コア管理部の更新依頼をされたコアのコア ID について，ID 変換部に LAPIC ID への変換を依頼する．

**(C)** ID 変換部

**(7)** コア ID を LAPIC ID に変換する．

**(8)** LAPIC ID をコア移譲インタフェース部に返す．

**(B)** コア移譲インタフェース部

**(9)** 更新するコアの LAPIC ID を用いてコア管理操作部にコア管理部の更新を依頼する．

**(D)** コア管理操作部

**(10)** コア管理部の更新を行う．

**(11)** コア管理部の更新結果をコア移譲インタフェース部に返す．

**(B)** コア移譲インタフェース部

**(12)** コア解放 AP にコア管理部の更新結果を返す．

**(A)** コア解放 AP

**(13)** ユーザにコア解放の結果を通知する．

次に，コアの占有を行う際の手順を以下に示す．

**(F)** コア占有 AP

**(14)** 占有するコア数を引数として，ユーザがコア占有 AP を実行する．

**(15)** コア移譲インタフェース部を呼び出し，占有するコアの検索を依頼する．

**(B)** コア移譲インタフェース部

**(16)** コア管理操作部に占有するコアの検索を依頼する．

**(H)** コア管理操作部

**(17)** コア管理部を参照し，占有可能なコアを検索する．検索結果から占有するコアを決定する．

**(18)** 占有するコアについてコア管理部の内容を占有済みの状態に更新する．

**(19)** コア移譲インタフェース部に占有するコアの LAPIC ID を返す．

**(G)** コア移譲インタフェース部

**(20)** 占有するコアの LAPIC ID のコア ID への変換をコア ID 変換部に依頼する．

**(I)** ID 変換部

**(21)** LAPIC ID をコア ID に変換する．

**(22)** コア ID をコア移譲インタフェース部に返す．

**(G)** コア移譲インタフェース部

**(23)** 占有するコアの LAPIC ID をコア占有 AP に返す．

**(F)** コア占有 AP

**(24)** 占有するコアに対し，CPU ホットプラグ機能を用いてコアを占有する．

**(25)** ユーザにコアの占有結果を通知する．

## 6. おわりに

本稿では，コア管理機能の実現について述べた．まず，Mint の特徴と目的について述べ，コア管理機能を実現する際の問題について述べた．そして問題を解決するための課題について述べ，対処を検討した．コア管理機能の実現には，ローカルタイマ割り込みの重複，各種識別 ID の重複，および OS 間にまたがるコア管理機構の欠如という 3 つの問題があった．これらの問題を解決するための課題として，HPET の割り当て方式の変更，OS 間でのコア識別方式の導入，コア利用状況管理機能の導入があった．これらの課題に関して，HPET の割り当て方式の変更と OS 間でのコア識別方式の導入には，OS 間で一意な値をもつ LAPIC ID を利用することで対処した．コア利用状況管理機能の導入では，OS 間に共有領域を作成し，各 OS によるコアの分散管理を行うことで対処した．これら対処を実装し，動作を確認した．

残された課題として，コア移譲の際の割り込み通知先の変更処理の実現と CPU の使用率により自動でコアの解放

と占有を行う方式の検討がある．

**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] Jeremy Sugerman，Ganesh Venkitachalam，and Beng-Hong Lim，“Virtualizing I/O De-vices on VMware Workstation’s Hosted Virtual Machine Monitor，” Proc. of the GeneralTrack: 2002 USENIX Annual Technical Conference，pp.1-14，2001.

[2] P.Barham，B.Dragovic，K.Fraser，S.Hand，T.Harris，A.Ho，R.Neugebauer，I.Pratt，and A. Warfield. “Xen and the art of virtualization，” In Proceedings of the ACM symposiumon Operating Systems Principles，pages 164-177，2003.

[3] 千崎良太，中原大貴，牛尾裕，片岡哲也，粟田祐一，乃村能成，谷口秀夫，“ マルチコアにおいて複数の Linux カーネルを走行させる Mint オペレーティングシステムの設計と評価，”電子情報通信学会技術研究報告，vol.110，no.278，pp.29-34，2010.

[4] T. Shimosawa,, H. Matsuba,, Y. Ishikawa，“Logical Partitioning without Architectural Supports，” Proc. of the 2008 Annual IEEE International Computer Software and Applications Conferece，pp.355-364，2008.

[5] 下沢拓， 藤田肇， 石川裕，“マルチコア SH における複数カーネル実行機構の設計と実装，”情報処理学会研究報告 2008-OS-109，pp.25-32，2008.

[6] Swapnil Pimpale，Adhiraj Joshi，Mandar Naik，Swapnil Rathi，Kiran Pawar，“Twin-Linux: Running independent Linux Kernels simultaneously on separate cores of a multicore system，” Proc. of the Linux Symposium 2010，Ottawa，Ontario，Canada，pp.101-108，2010.

[7] Andrew Baumann，Paul Barham，Pierre-Evariste Dagand，Tim Harris，Rebecca Isaacs，Simon Peter，Timothy Roscoe，Adrian Schupbach，and Akhilesh Singhania，“The Multikernel:A New OS Architecture for Scalable Multicore Systems，” Proc. of the 22nd ACM Symposium on Operating Systems Principles，pp.29-44，2009.