社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2008－OS－107 (16)
2008／1／31

# 動的なプロセッサコア資源の変動に適応するCPUスケジューラの検討

荒木 裕靖† 金城 聖†
元濱 努†† 片山 吉章†† 毛利 公一†††
† 立命館大学大学院理工学研究科 †† 三菱電機株式会社情報技術総合研究所
††† 立命館大学情報理工学部

仮想化技術を搭載したマルチコアプロセッサが登場し，仮想計算機 (VM) 上のゲストOSを効率的に動作させる技術が研究開発されている．ただし，現在は仮想計算機モニタ (VMM) と各ゲストOSが，各々の状況に適応して協調しながら，有限な資源であるプロセッサコアを効率的に共有する仕組みについての研究は十分なされているとは言えない．そこで，我々は，VMMによるVMへのプロセッサコアの割当て処理と，ゲストOSによるプロセスへのプロセッサ割当て処理を連携させ，システム全体として，見かけ上，プロセッサコア数よりも高いパフォーマンスを出すための方式を研究している．本論文では，研究の初期段階で行ったマイクロカーネルLavenderのマルチコア対応について述べ，プロセッサ割当ての連携手法の検討について述べる．

# Examination of Adaptive CPU Scheduler for Dynamic Change of Processor Core Resources

Hiroyasu ARAKI† Akira KANASIRO†
Tsutomu MOTOHAMA†† Yoshiaki KATAYAMA†† Koichi MOURI†††
†Graduate School of Science and Engineering, Ritsumeikan University
††Mitsubishi Electric Corporation
†††College of Information Science and Engineering, Ritsumeikan University

Multi-core processors equipped with the virtual technology have appeared, and technologies that efficiently operates guest OSes on Virtual Machines (VM) are researched and developed. However, researches of mechanisms for collaboration between Virtual Machine Monitor(VMM) and each guest OS have not been necessarily performed enough. For example, VMM and guest OSes cannot adapt to each current situation like load. Therefore, we propose a new method to achieve higher performance than real processor cores speciously. It is based on collaboration between two allocation algorithms. One is allocating appropriate number of processor cores to VMs by VMM. The other is allocating time slices to processes on each guest OS. In this paper, we discuss developing a microkernel Lavender on multi-core hardware, the first step of this research, and discuss examination of coordinated technique for processor allocation.

## 1 はじめに

近年，仮想化技術を搭載したマルチコアプロセッサが登場し，普及してきている．従来のシングルプロセッサでは，周波数の高速化で処理能力を向上させていたが，このような方法での処理能力の向上は，発熱や消費電力の問題がある．そのため，複数のプロセッサを利用し，これらに処理を分散させることで処理能力の向上を図る方法が注目されている．マルチコアプロセッサを搭載した計算機は，従来のシングルプロセッサを搭載した計算機と比べ，並列処理を利用したプロセスの実行が優位な環境である．このような優位点は対称型マルチプロセッシング (Symmetric

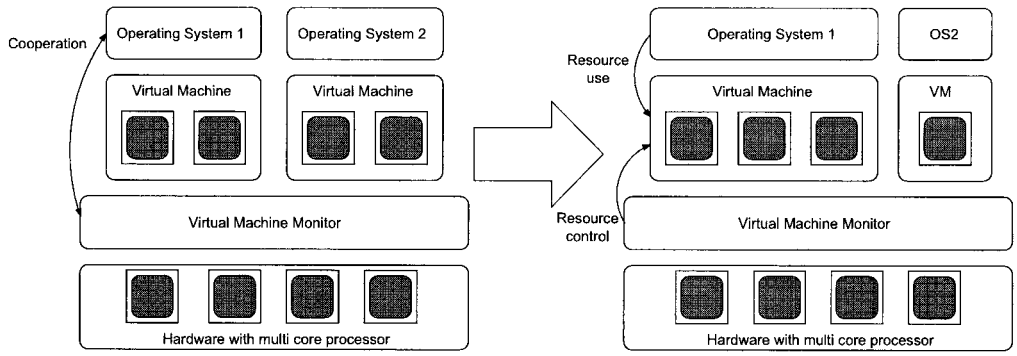


図 1　仮想計算機モニタと OS との協調

Multi-Processing，以下 SMP と記す) でも見られるが，マルチコアプロセッサではキャッシュを共有することでさらに効率的な処理が可能となっている．

また，近年注目されている仮想化技術は，物理的な計算機資源を OS から隠蔽し，OS に仮想計算機を提供する．1 台の計算機上に複数の OS を起動させることが可能となっており，このように提供されている仮想計算機 (Virtual Machine，以下 VM と記す) には，物理的な計算機の資源を自由に割り振ることが可能となっている．近年，高性能化している計算機資源では，今までの計算機 1 台に対して OS1 つといった利用方法と比べ，仮想化技術を用いて 1 台の計算機上で複数の OS を起動させることでより柔軟な資源利用が可能となる [1][2].

このようにマルチコアプロセッサと仮想化技術を組み合わせて利用することで，複数の VM に物理的なコアをそれぞれ割り当てることが可能となり，また，割り当てるコア数を指定することで，それぞれの VM の処理能力を指定することが可能となった．これにより，従来，計算機に依存していた OS の処理能力を自由に変更することが可能となり，OS の特性によってプロセッサの割当てを指定することで，計算機資源を有効に利用できるようになった．

しかし，計算機の利用法によっては，プロセッサ利用率が低い場合も多く，その場合はプロセッサの処理能力は常に必要ではない．一方で，数値計算やサーバなどでは利用率が高い場合が多い．このように，ある VM に割当てられているプロセッサが利用されておらず，別の VM に割り振られているプロセッサの処理能力が不足しているような場合，システム全体としては効率的な計算機利用が出来ているとはいえない．

そこで，複数の OS が 1 台の計算機上で動作するような VM 環境において，プロセッサコアを OS の負荷に応じて，適切に OS へ配分することによって，より効率的で適応的な資源利用を実現する．コア資源を動的に配分することによって，必要の無いと思われるコアを停止させ消費電力を抑えることが可能となり，また，動作していないコアを処理能力を必要としている OS に割り当てることで，各 OS の要求を満たすことが可能になる．

既存の OS では起動時にコア数を検出するが，その後，コア数が変動する場合を考慮していない．このため，OS は割り当てられたコア資源を負荷の高低にかかわらず独占する．そこで，OS の起動後でも動的にコア資源の変動を可能にすることによって，コア資源の占有を回避することができ，上記のようなコア資源の利用が可能となる．

このようなカーネルの構築のため我々は，Lavender[3] をターゲットとして用い，Lavender を改良することでこれを実現する．Lavender は筆者らの研究室で開発されたカーネルで，必要最小限の機能を備え，カーネルサイズも小さいため，今回，ターゲットとして利用する．

以下，2 章では，VM モニタ (Virtual Machine Monitor，以下 VMM と記す) とゲスト OS の協調した環境で，動的なプロセッサの変動を伴う CPU スケジューラを提案し，3 章では，VMM とゲスト OS とが協調した動作をするうえで，Lavender へ実装が必要な機能を述べる．4 章で，Lavender へ実装の必要がある機構の中でも，今回実装を行ったプロセッサの起動について述べ，5 章では実際に実装を行ったことを述べる．6 章では，OS 起動後でも CPU の交換が可能な CPU Hotplug と，OS の要求に応じて処理能力を変動させることが可能な共用プロセッサプールについて述べ，7 章でまとめとする．

## 2　VMM とゲスト OS の協調

1 章で述べた様な計算機資源の利用は，図 1 に示すような VMM と OS の協調によって実現される．図 1 左側では，4 つのコアを有するプロセッサ上で，2 つの VM が提供されており，それぞれ OS が動作している．各 VM には 2 つずつコアが割当てられている．このとき，OS1 の負荷

が高くなってきたとすると，図1右側のように，負荷の低いOS2からコアを回収し，それをOS1に割り当てる．本論文では，資源の中でも特に，プロセッサコアについて着目し，プロセッサコア数の変動に適応可能なマルチコア対応カーネルの構築を行う．このカーネルを利用することで，OSは起動しているアプリケーションの特性によってコア資源を動的に変動させることが可能となる．

以上で述べたような，VMMとOSの協調を実現するための手法としては，自律型と協調型スケジューリングが考えられる．以下，これらのスケジューリング手法について述べる．

### 2.1 自律型スケジューリング

VMMがVM上で起動しているOSの動作を観測し，その観測結果をもとに各VMに対しての資源管理を行う．OS上で動作するプロセスによる負荷が高くなった場合など，プロセッサが必要になった場合は，VMMがそれを観測することで，VMに割当てるプロセッサ資源を増加させる．プロセッサ資源の減少の場合もこれに準じ，OSの負荷が下がったようであれば，それをVMMが検出し，プロセッサをVMMがVMから開放することで他のVMに使用可能なものとする．

この方式でのスケジューリングの場合，VMMとの通信は原則として行われない．プロセッサ数の変化があったときのみ，シグナルを送ればよい．このため，協調型スケジューリング手法より容易に実装することができる．また，VMM，OSに対する変更も最小限のものとなり，OSは，動的にプロセッサ資源が変動する環境であれば，どのようなVMM上でも利用が可能となる．

### 2.2 協調型スケジューリング手法

自律型スケジューリング手法に対し，この手法はVMMとOSとの間で通信する機構を実装し，相互にプロセッサ利用率や，スケジューリング対象となっているプロセスといった情報をやり取りすることで，VMMと各OS間での総合的なスケジューリングを行う．

OSは自身のプロセスの要求，現在の状況などに従い，VMMへとプロセッサの増減要求を発行する．OSからのプロセッサ増減要求を受信したVMMは，全VM上で動作しているOSの状況を考慮し，VMの資源を管理する．その結果，VM上のプロセッサ資源の増加があった場合，VMMは対象のOSに対しプロセッサ資源の増加を通知する．VMMからの通知を受理したOSはプロセッサの確保を行い，自身のプロセスを確保しているプロセッサに対してスケジューリングを行う．また，VMMからプロセッサ資源の解放要求があった場合は，OSは自身が確保しているプロセッサを開放し，VMMが開放されたプロセッサをVM上から削除する．

この方式の場合，OSとVMMに，資源の増減に対応した機構を実装するだけでなく，双方向の通信機構を備える必要がある．しかし，自律型スケジューリング手法と比べ，よりVMMとOSの連携の取れたスケジューリングが可能であり，結果的にパフォーマンスの向上が見込まれる．

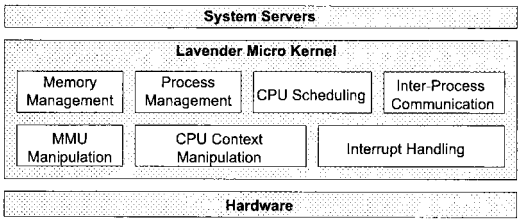


図2 Lavenderの機能構成

本研究では，この協調型スケジューリング手法を採用し実装を行う．以下で，協調型スケジューリングを実現する上でLavenderに実装すべき機能を述べる．

## 3 Lavenderの設計

### 3.1 Lavenderの概要

Lavenderの構成を図2に示す．Lavenderは，メモリ管理，プロセス管理，CPUスケジューリング，プロセス間通信，MMU操作，コンテキスト操作，割込み管理の機能を持った，マイクロカーネルベースのOSである．現在，Lavenderはシングルプロセッサ上で起動することを前提として設計されており，協調型スケジューリングを実現するためには，まずマルチコアプロセッサに対応させる必要がある．また，VMMとの協調のため，以下に述べるような機能が必要となる．

### 3.2 協調に必要な機能

VMMと協調した動作を行うカーネルには，VMMからの資源提供を認識する機構が必要となる．また，VMMはVM上の各OSの要求を受理し，それぞれの要求を統合的に判断して，各VMの資源を管理する．このような環境では，OSは物理的な計算機資源が動的に変動しているように見える．そのため，従来のOSでは，新たに提供された資源を利用することができず，また，起動時に識別した資源を解放することもできない．すなわち，本研究で開発するカーネルには，VMMからの要求や，自らの判断で，資源を解放したり，獲得を行う機構を備える必要がある．以下では，特にプロセッサに着目し，本カーネルの構築に必要な機構について述べる．

**プロセッサの起動，停止**

カーネルは，VM上に提供されたプロセッサを起動し，利用しなければならない．従来のカーネルであれば，このようなプロセッサの起動処理はカーネルの初期化時に行われ，以後，行われる必要はない．しかし，本研究で取り扱う環境では，OS起動後にプロ

セッサ数が変動するため，カーネルの起動時よりカーネル起動後にプロセッサの起動処理を行うケースが多い．そのため，本カーネルでは任意のタイミングでのプロセッサ起動を可能なものとすることで，動的なプロセッサの変動に対応する．

**同期機構**

シングルプロセッサの場合，プロセッサへの割込みを禁止するだけでプロセス間の同期がとれ，データの整合性を保つことができる．しかし，マルチプロセッサの場合，割込みを禁止するだけでなく，別途，同期機構を実装する必要がある．この同期機構も動的なプロセッサの変動に対応したものでなければならない．

**VMM との協調機構**

VMM とゲスト OS に通信機構を実装することは，2.2 節で述べた協調型スケジューリング手法を実現するために必要な機構である．VMM が VM に対して，どのようにプロセッサを割り当てるかを決定する情報として，各 VM 上で起動している OS のプロセッサ利用率，スケジューリングすべきプロセスの数，処理特性といったものが挙げられる．このような情報を VMM に送信する機構を OS 内部に実装する．また，送信だけでなく受信する機構も必要となる．VMM から OS に対しての送信には，どれだけのプロセッサ資源が利用可能であるか，といった情報や，プロセッサの追加報告，開放要求といったものが挙げられ，OS はこの通信の情報を管理する必要がある．

**CPU スケジューラ**

CPU スケジューラは自身が識別しているプロセッサに対して，スケジュールされるべきプロセスのスケジューリングを行う．また，VMM との協調を行うため，自身が起動している VM 上にあるプロセッサ数だけでなく，実計算機に搭載され，利用可能な総プロセッサ数を把握し，必要に応じて適応的にこれらを利用しスケジューリングを行う．

このようなスケジューリングは，まず，自身が確保しているプロセッサでスケジューリングを行い，リアルタイム性能といったような，各プロセスの要求を満たせない場合，VMM に対しプロセッサ追加要求を発行し，新たにプロセッサを確保することでプロセスの要求を満たす．VM 上の OS から上記のような要求を受理した VMM は，全ての OS の状況を把握した上で，総合的な要求を判断し，VM に対してプロセッサの管理を行う．

このようなスケジューラを実装することで，全システムを見通したスケジューリングが可能となる．今までのスケジューラでは，OS 自身のプロセスだけを把握し，スケジューリングを行っていたが，VMM 上で起動することを考慮し，VMM とシステムの資源利用率をやり取りすることで，総合的なスケジューリングが可能となる．

VMM との協調にはこのような機構が必要であるが，本論文では，特に Lavender のマルチコアへの対応について述べる．

## 4 プロセッサの起動手順

SMP においてプロセッサは 2 つのクラスのプロセッサに分類される．OS 起動時に動的に決定され，OS の起動に利用されるプロセッサをブートストラッププロセッサ (BootStrap Processor，以下 BSP と記す) と呼び，それ以外のプロセッサをアプリケーションプロセッサ (Application Processor，以下 AP と記す) と呼ぶ．マルチコアにおいても，各々のコアを上記のように分類する．

まず，マルチコア対応化のためには，自身で起動することが無い AP を起動させる必要がある．以下，AP 起動の主な流れを示す (図 3 参照)．

1. AP の Local APIC ID の取得
2. IPI の送信
   i. INIT 信号のアサート，デアサート
   ii. STARTUP 信号の送信
3. AP の初期化

なお，APIC(Advanced Programmable Interrupt Controller) とは，割込みコントローラであり，今回利用する Local APIC は各プロセッサ内に搭載され，割込みを管理する．この Local APIC によって IPI を送信し，これを利用してプロセッサ間通信を行うことができる．また，IPI とはプロセッサ間割込み (InterProcessor Interrupt) のことであり，プロセッサの起動，停止といったプロセッサ間通信を行うことができる．

### 4.1 Local APIC ID の取得

AP の起動には Local APIC を使用する．Local APIC を利用するためには，Local APIC が持っているレジスタをアクセスする必要である．各プロセッサの Local APIC のレジスタは，物理アドレス 0xFEE0 0000 から 4KB の空間へマップされており，これは，モデル固有レジスタ (Model Specific Register，以下 MSR と記す) の値を参照することで確認できる．このアドレスにアクセスするためには Local APIC が OS によって認識される必要があり，そのために次で述べるような Local APIC の初期化が必要となる．

Local APIC の初期化では，CPUID 命令によって Local APIC のサポートの有無を確認し，MSR からの値の取得によって Local APIC が利用可能な状態かを確認し，無効

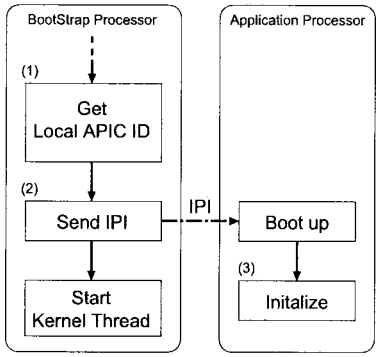


図 3　プロセッサ起動の流れ

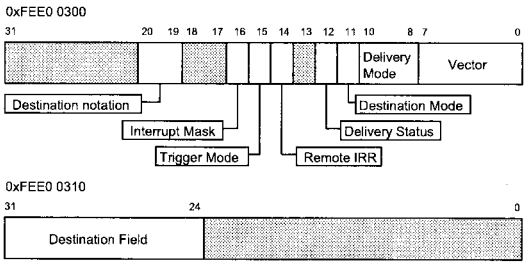


図 4　割込みコマンドレジスタ

になっていた場合 MSR へ書込むことで Local APIC を有効にする．これらの処理が終了すると，Local APIC を利用することが可能となる．

次に，Local APIC のレジスタを OS から参照可能なアドレスへキャッシュ不可でマップすることで，OS は Local APIC のレジスタにアクセスすることが可能となり，Local APIC の提供する Local APIC ID を確認することができ，Local APIC を利用した IPI の送信も可能となる．

BSP の Local APIC ID は上記のような方法で取得することが可能であるが，AP の Local APIC ID は，AP が起動しておらず，AP の Local APIC レジスタが参照不可なため，取得することが出来ない．そこで，Local APIC ID に連続した値が割り振られることを利用し，AP の Local APIC ID を決定する．

### 4.2 IPI の送信方法

IPI の送信は，Local APIC レジスタの一つである，割込みコマンドレジスタ (Interrupt Command Register，以下 ICR と記す) に適切な値を書込むことによって任意の IPI の送信を行うことができる．利用する ICR のアドレスは 0xFEE0 0300 と 0xFEE0 0310 でありそれぞれ図 4 に示すような領域を持っている．以下に，今回利用する項目を挙げる．

- Destination Field
  IPI を送信する対象のプロセッサの Local APIC ID を代入する．
- Delivery Mode
  送信する IPI の種類を選択する．
- Trigger Mode
  割込みのトリガがレベルか，エッジかを選択する．
- Remote IRR
  IRR は割り込み要求レジスタと呼ばれる領域で，エッジトリガの割込みの際，アサート，デアサートを選択する．
- Interrupt Vector
  今回は STARTUP IPI 利用時に AP が処理を開始するコードのアドレスを書込む．

このような項目を利用し，INIT IPI と STARTUP IPI を AP に送信する．これは，BSP が，IRC に書込むことによって実現する．

INIT IPI はハードウエアの初期化のために送信され，STARTUP IPI によって AP は自身の起動を行う．INIT IPI の場合，割込みはレベルトリガで行い，Remote IRR ビットをセットし，INIT 信号をアサートする．アサート出来たことを確認し，今度は Remote IRR ビットをリセットしデアサートを行う．この作業で対象 AP のハードウェア初期化が完了する．

次に，STARTUP IPI を送信する．割込みはエッジトリガを選択し，Interrupt Vector に AP が処理を開始するコードのアドレスを書込むことで STARTUP IPI を対象 AP に送信する．STARTUP IPI を AP が受理し，かつ，何もエラーがない場合 BSP による起動処理は終了する．

### 4.3 AP の初期化

AP は，STARTUP IPI を受信すると，STARTUP IPI に付随するスタートアップアドレスから処理を開始する．起動を開始した AP は，リアルモードで動作している．プロテクトモードへの以降は，プロセッサが持つレジスタの初期化を行い，CR0 レジスタの PE ビットに 1 をセットすることで実現する．また，ページディレクトリを作成し，初期化を行うことで AP の初期化は終了する．

## 5 Lavender への実装

4 章の手順を基に，Local APIC レジスタのマップ，AP の起動関数を，Lavender に実装した．

### 5.1 Local APIC レジスタのマップ

Local APIC の初期化が終了した段階では Local APIC が提供しているレジスタへとアクセスすることができないた

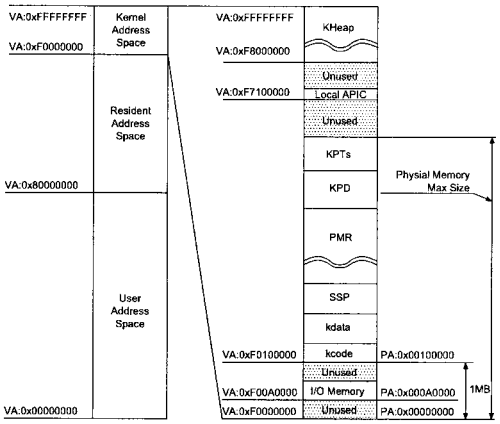


図 5　**Lavender のアドレスマップ**

op. Code

| | |
|---|---|
| 0x1000 | 0x0000b866 |
| 0x1004 | 0xc083d88e |
| 0x1008 | 0xe08ed08e |
| 0x100c | 0x31b0e88e |
| 0x1010 | 0x0eb4ff30 |
| 0x1014 | 0x03cd10cd |

"0x1000" is Start Address

Assembly Language

```
01  mov $0x00, %ax
02  mov %ax, %ds
03  mov %ax, %es
04  mov %ax, %ss
05  mov %ax, %fs
06  mov %ax, %gs
07  mov $'1', %al
08  xor %bh, %bh
09  mov $14, %ah
10  int $0x10
```

図 6　**実装したアプリケーションプロセッサ起動確認コード**

め，レジスタがマップされている物理アドレスを仮想アドレスへとマッピングする必要がある．

今回，Local APIC レジスタへとアクセスするため，Lavender へメモリマップのコードを追加した．Lavender では，図 5 に示すように，仮想アドレスの 0xF800 0000 より低位のアドレスに使用していないアドレス空間があるため，この使用していない 0xF710 0000 から 4KB の領域を 0xFEE0 0000 へとマップするようにコードを追加している．この追加したコードによって，Lavebder 上で 0xF710 0000 へとアクセスすることで，物理アドレスの 0xFEE0 0000 へとアクセスすることが可能となる．これにより，Local APIC の利用が可能となり，AP 起動等の IPI の利用が可能となった．

### 5.2　AP の起動

BSP の初期化が完了した後，4.2 節で述べた，IPI を AP に送信し AP の起動を行った．現在は，AP の起動確認は BIOS コードを用い，画面上に文字を出力することで行っている．これは，図 6 に示すような，機械語コードを直接メモリ上に配置することで実現している．

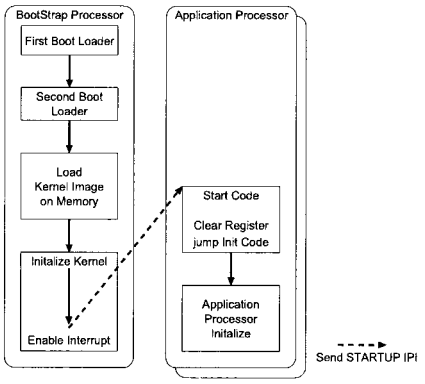


図 7　**Lavender でのアプリケーションプロセッサ起動**

今後，AP の初期化を行うが，これにはリアルモードで動作している AP をプロテクトモードへ移行させ，その際，ページテーブルの作成，初期化を行う必要がある．このコードは上記のような実装方法は取れないため，現在，図 7 に示すような実装方法を考えている．

Lavender は，ブート時には，ファーストブートローダ，セカンドブートローダの順に起動し，その後カーネルがメモリ上に読み込まれる．次に，カーネルの初期化が行われる．それらが完了したところで AP を起動する．この起動手順では，スタートアドレス以降のコードで最低限のレジスタの初期化を完了させ，AP 用のコードを記述したアドレスへとジャンプする．AP 用の初期化関数を作成し，ここで，ページテーブルの初期化，およびプロテクトモードへの移行を行う．

## 6　関 連 研 究

### 6.1　Linux の CPU Hotplug

Linux-2.6.13 以降のカーネルには，OS を停止させることなく CPU の数を変更することが可能な CPU Hotplug[4] という機能がある．これまでの SMP のための機構にいくつかの変更，追加を行うことでこれを実現している．CPU Hotplug を利用することで，CPU に障害が生じた場合，障害を持った CPU を OS 起動中にスケジューリング対象から除外し，停止させることでシステムを停止することなく CPU を計算機から取り外すことが可能となる．また，OS 起動中に新たに追加された CPU に対し起動処理を行うことで，CPU の追加も可能となり，システム起動中の CPU 交換が可能になっている．

CPU Hotplug では，利用可能な CPU 数を保持していた変数を可変に変更し，最大利用可能 CPU 数と現在利用可能な CPU 数の二つを保持している．前者は，システムで追加可能な最大 CPU 数を表し，後者は，OS が識別しス

ケジューリング対象となっている CPU 数を表している．

OS 起動後の CPU の追加に対応するため，これまでカーネルの初期化処理内で行っていた CPU の初期化処理を，任意のタイミングで行えるようにしており，CPU のレジスタや，タイマ，IRQ などの初期化処理を新たに実装している．これらの処理を任意のタイミングで実行することで，OS の再起動を行うことなく CPU の追加が可能となる．

また，CPU の停止に関しても，CPU が保持している構造体，変数などを開放し，IRQ の設定を変更を行う．スケジューラが管理している run キュー上のプロセスを他の CPU へ移動を行い，停止に伴うプロセスの消失を防止する．最後に，CPU のキャッシュをフラッシュし，CPU に対して停止命令を発行する．これによって，OS を停止せずに CPU の削除が可能となる．

Linux に実装されている CPU Hotplug は OS の再起動なしで CPU 数を変更可能だが，スケジューラは通常の SMP に対応したものであり，現時点で確保しているの CPU をすべて使用しスケジューリングを行う．

我々が開発しているカーネルでは，VMM との協調によってスケジューリングを行う予定であり，VMM がシステムを総合的に監視することで，各 VM 上の OS に CPU の起動，停止といった要求を発行する．この要求によって各 OS は，CPU の起動，停止を行う．CPU Hotplug は任意のタイミングで CPU の起動，停止が可能であるため，CPU の起動や停止部分の処理を本カーネルを構築するにあたり，参考にできる可能性がある．

### 6.2 共用プロセッサプール

共有プロセッサプール [5] とは，IBM が開発した VM 上の OS に対するプロセッサの処理能力を変動できる概念である．物理的なプロセッサを共有プロセッサプールというプロセッサの集合に登録しておき，各 VM に対しどれだけの処理能力を割当てるかを範囲で指定する．これによって，ある VM 上の OS が処理能力を必要とするとき，別の VM に割当てである処理能力のうち余剰分をその処理能力を必要としている VM へ分配することが可能となる．動的に処理能力を割り振ることにより，固定的に処理能力を割り当てるよりもシステム全体的にパフォーマンスの向上を可能としている．

我々の開発しているカーネルでも動的に処理能力を変動させているが，プロセッサコアを動的に変動させているという点で差異が見られる．

## 7 おわりに

本論文では，VMM と OS とが協調し，システム全体のパフォーマンスを向上させる手法について述べ，特に OS に焦点をあて，動的なプロセッサコアの変動に適応するカーネル Lavender の構築を行っていることを述べた．現在，VMM と OS はそれぞれが独立に動作し，お互いの状況を考慮した動作を行っているとはいえない．そのため，VM 上で起動する OS は，VM に割り振られているプロセッサだけでは，各プロセスの要求を満たせない場合や，他の VM 上の OS が処理能力を必要としているときに，過剰なプロセッサを占有する場合がある．Lavender に VMM との通信機構や，プロセッサの変動に対応した同期機構，スケジューラなどの機能を実装することで，VMM と協調した動作を行うことが可能となる．このような VMM との協調のために必要な機能についても述べた．

本カーネルは，カーネル起動後のプロセッサ数の変動に対応するため，従来のカーネルではカーネル初期化時にしか実行されることのない CPU 初期化処理を任意のタイミングで実行可能な実装方法を取っている点で特徴的である．実装については，Lavender のマルチコア化から進めている．

今後，同期機構についても，プロセッサ数の変動に伴う考察を行い実装を行ってゆく．さらに，プロセッサコアの変動に適応可能なスケジューラを実装し，VMM との協調を考慮に入れ評価を行う．

## 参考文献


[1] 遠藤 幸典，菅井 尚人，山口 義一，近藤 弘郁:"シングルチップマルチプロセッサ上のハイブリッド OS 環境の実現 –システムアーキテクチャ–"，第 66 回情報処理学会全国大会講演論文集，Vol.1，pp.9–10 (2004).

[2] 菅井 尚人，遠藤 幸典，山口 義一，近藤 弘郁:"シングルチップマルチプロセッサ上のハイブリッド OS 環境の実現 –OS 間インタフェースの実装–"，第 66 回情報処理学会全国大会講演論文集，Vol.1，pp.11–12 (2004).

[3] 毛利公一，大久保英嗣:"マイクロカーネル Lavender の設計と開発"，電子情報通信学会論文集 (D-I)，Vol.J82-D-I，No.6，pp.730-739 (1999).

[4] 青野 寛，河内 隆仁，菅沼 公夫:"Linux における CPU/Memory/IO の Hot Plug サポート"，Linux Conference 2002 (2002).

[5] "OS/400 V5R2 における iSeries の論理区画化"，http://www.e-bellnet.com/special/vision/vision_0305.html.