システムソフトウェアと
オペレーティング・システム 92－3
（２００３．２．２７）

二つの OS 共存に向けた起動方式

田渕　正樹†　　中島　雄作†　　伊藤　健一†
乃村　能成‡　　谷口　秀夫‡

いくつかの OS が固有に持つアプリケーションを同時に利用可能とし，利便性の向上を図るために，我々は一つの CPU を持つ計算機の上に複数の OS を共存させる構成法を提案し，検討している．ここでは，二つの OS の共存に際し，共存するために課題となる実メモリの分割方式と，各 OS の起動方式について述べる．具体的には，システム立ち上げ時に各 OS の使用するメモリ領域を排他的に確保する方式，及び先に起動した OS が単独走行の状態から二番目の OS を，割り当てられた実メモリ領域にロードし，起動させるまでの制御方式について述べる．

A Method for Booting Two Coexistent Operating Systems

Masaki TABUCHI†, Yusaku NAKAJIMA†, Ken-ichi ITOH†,
Yoshinari NOMURA‡ and Hideo TANIGUCHI‡

We have been developing a method for running multiple operating systems on a single computer for utilizing their individual application programs effectively. This paper describes how to boot up multiple operating systems safely in succession by sharing physical memory among them. Under the method, physical memory is exclusively divided and reserved for each operating system from beginning. Each operating system is controlled to boot up using their own memory region without having any bad effect on other operating systems.

1．はじめに

パーソナルコンピュータ（以降，PC と略す）に代表される計算機の普及は目覚しく，アプリケーション（以降，AP と略す）を基盤ソフトウェア，特にオペレーティングシステム（以降，OS と略す）上で走行させることで様々なサービスが提供できる．既存の AP にはネットワーク通信に関連するもの，文書処理を行うものなどがあるが，これらは個別の OS 対応のものが多く，OS の種類に関係なく，そのまま利用可能な AP はほとんどない．例えば，Web サーバやメールサーバなどには Unix 系 OS を，文書作成処理には Windows を，というように，サービスの特性に合わせた OS が利用されることが多い．

このように各 OS の特長を生かした利用が望まれており，一つの計算機上で複数の OS を利用できれば，利便性が向上すると共に，各 OS に対応している AP を，使用している OS に対応するよう改造することなく利用できるため，システム開発工数の削減も可能である．

そこで，我々はハードウェア資源を各 OS に直接割り当てることにより，複数の OS を独立に，かつハードウェア性能を十分に生かして，同一計算機上で時分割走行させる技術（複数実計算機（MRM:Multiple Real Machines）と名付ける）を提案した[1]．

ここでは，一つの計算機上で複数 OS が共存する環境を構成する過程において，二つの OS が共存するための各 OS の起動方式，及び二つの OS で共有しなければならない実メモリの利用方法について述べる．

2．複数実計算機（MRM）

2.1　MRM 開発方針

複数実計算機（MRM）は，以下の方針に基づいて構成する．

（１）OS には一切変更を伴わない．
（２）ハードウェアの入出力機能を十分に利用できる．
（３）各 OS はハードウェア性能を生かした性能を AP に提供できる．
（４）各 OS は独立に動作する．

以下，上記方針の理由を説明する．

（１）について，複数実計算機（MRM）では，

†株式会社 NTT データ技術開発本部
Research and Development Headquarters, NTT DATA Co.
‡九州大学大学院システム情報科学研究院
Faculty of Information Science and Electrical Engineering, Kyushu University

複数の OS は市販の OS を含めた様々な OS をそのまま利用したい．したがって複数実計算機（MRM）を構成するにあたり，OS の再コンパイルを伴う改造による実現方式は採用できない．これにより，市販 OS を改造するというソフトウェアライセンスに関わる問題を解決することができ，様々な種類の OS を容易に組み込むことが可能になる．

（２）について，ハードウェアの変遷は目覚しく，次から次へと新しい周辺機器が登場している．このような状況下では，一つの OS で全てのハードウェア入出力機能をサポートすることは難しい．また使い慣れた AP を使い続けたい要求もあるので，ひいては古いハードウェア入出力機能も必要となってくる．したがって，利用者の要求する様々なハードウェア入出力機能を利用できることが望ましい．

（３）について，一つの計算機上に複数 OS が走行することにより，各 OS の性能が低下することを防ぐ必要がある．特に，入出力装置の性能がプロセッサ（以降，CPU と略す）性能に比べて非常に低いため，AP は入出力装置の性能低下による影響を受けやすい．したがって，AP の利便性を確保するために，入出力機能の性能低下を防ぐ必要がある．

（４）について，既存の複数 OS 構成法である仮想計算機（VM: Virtual Machine）では，複数の OS が一つのハードウェアを共有する形で構成されている．また，共有されているハードウェアを制御するのは，主となる一つの OS（基盤 OS）であり，基盤 OS の上で他の OS が走行する．したがって，複数の OS 間に主従関係がある．そのため，基盤 OS の障害が他 OS に影響を及ぼす．この問題を解決するため，複数の OS 環境がそれぞれ独立した状態であるような構成法を実現する必要がある．

上記方針を満足するためには，我々は一つの計算機のハードウェアを複数の OS で共有せず，ハードウェアを非共有にすることが最良と考えた．

図１にハードウェア非共有の様子を示す．

入出力装置１～３は OS1 により占有制御され，入出力装置４～６は OS2 により占有制御されている．ただし，CPU とメモリについては，一つの計算機上に一つずつしかないため，全ての OS で共有せざるを得ない．

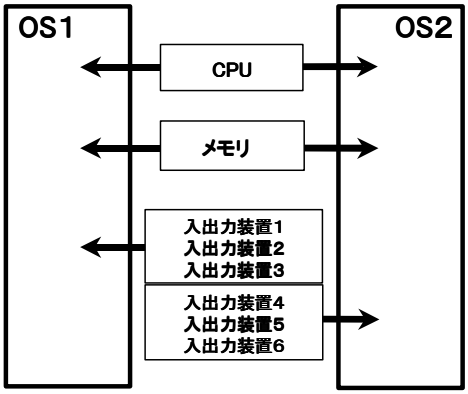

図1　ハードウェア非共有の様子

2.2　MRM 実現のための課題

ハードウェアをできる限り非共有にさせて複数の OS を共存させる構成法を実現する際の課題を，以下に挙げる．

（１）起動方式

（２）共有ハードウェアの扱い

以下，これら課題について説明する．

（１）について，「各 OS は独立に動作する」という方針を言い換えると，各 OS は他の OS が走行していることを関知せずに走行しなければならない．したがって，各 OS が影響を及ぼし合わないよう，独立した走行環境を構成する必要がある．一般に OS は，起動時に，メモリチェックや入出力機器のハードウェアの初期化と，メモリマッピングテーブルや割込みテーブルなどの OS の初期化を行う．したがって，各 OS の初期化処理において先に起動している OS の環境を破壊しないように起動する方法を検討する必要がある．

（２）について，複数実計算機（MRM）では，「複数の OS がハードウェア資源を非共有する」としているが，例えば，CPU やメモリのように，OS の動作に不可欠なものについては，それを共有せざるを得ない．これらは，あたかも非共有されているようにするため，分割して共有することになるが，その分割方式について検討する必要がある．

上記課題を解決するに際し，ここでは，先に一つの OS が起動している状態から別の OS を起動させる方式と，各 OS 間での実メモリの分割利用方式について述べる．

以降，先に起動する OS を OS2 と表記し，OS2 起動状態から起動させる OS を OS1 と表記する．

3．二つの OS の共存

3.1　二つの OS 共存への流れ

一つの計算機上に二つの OS が共存している様子を図 2 に示す.

各 OS は，実メモリ領域を分割占有し，仮想空間は各 OS が占有している実メモリ領域に対応する空間をそれぞれ利用する.

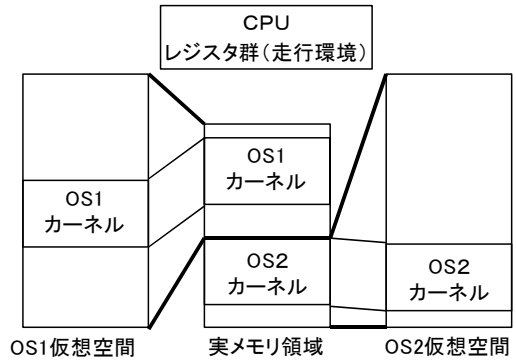

図2　二つの OS が共存している様子

二つの OS が共存する環境は以下の流れにより実現できる.

（１）OS2 が起動し，実メモリの上位アドレス部分を利用して動作を開始する．通常利用される下位アドレス部分は利用しないようにする.

（２）OS1 の起動を契機として，OS2 の走行環境を OS2 のメモリ領域に保存しておく.

（３）通常の起動動作，つまりリセットボタンを押したのと同じ状態で計算機を再起動する.

（４）ブートローダが HDD から起動する OS として OS1 を選択する.

（５）OS1 が実メモリ領域にロードされる．この際，上位アドレス部分には，動作を中断された（２）の状態の OS2 がそのまま残っている.

（６）OS1 は，初期化処理において，BIOS から OS1 が利用可能な実メモリ領域を取得する．この時，OS2 が使用している実メモリ領域を，使用不可と認識するようにする.

（７）OS1 は，OS2 が未使用の実メモリ領域を利用して起動する.

3.2　課題

前節の流れを実現するためには，以下の課題がある.

（１）OS2 の利用できる実メモリ領域を制限し，任意の位置で動作させる方法.

（２）OS1 が，OS2 にとって未使用の実メモリ領域を使用して走行する方法.

以下に各課題について説明する.

（１）について，実メモリ領域には二つの OS のカーネルがロードされることになるため，OS2 カーネルが OS1 カーネルに上書きされないように配置する必要がある．また，二つの OS でメモリを非共有とするために，OS2 の使用できる実メモリ領域を制限する必要がある.

（２）について，（１）と同様に，二つの OS でメモリを非共有とするために，OS1 は，OS2 が未使用の実メモリ領域を使用して走行する必要がある.

ここでは，前節の流れを OS2 と，OS1 カーネルの実メモリ領域へのブートローダに変更を加えることによって実現する．つまり，「OS には一切変更を伴わない」という MRM の開発方針実現の前段階として，少なくとも「OS1 カーネルには一切変更を伴わない」という方針で二つの OS を共存させる方法を採る．具体的には，OS2 としてソースコードが入手可能な Linux を採用し，任意のメモリアドレス上で実行可能になるように改造した．また，OS1 を起動する契機として，OS2 に対して OS1 起動を合図する割込みを発行する．OS2 に変更を加えることで，OS2 の使用可能な実メモリ領域の制限と，任意の位置での走行を可能とし，課題（１）が実現できる.

一方，（２）については，「OS1 カーネルには一切変更を伴わない」としたので，OS1 を変更せずに，OS1 の初期化処理から OS2 の走行環境を保護しなければならない．したがって，以下の課題が挙げられる.

(A) OS1 カーネルが初期化処理を行う前に一度 OS2 に制御を移行する方法.

(B) 移行後，OS2 が監視している状態で OS1 の起動を行う方法.

(C) OS1 が利用可能な制限された実メモリ領域を OS1 に認識させる方法.

(D) OS2 の仮想空間上で OS2 カーネルが対応付けられる位置の移動方法.

以下，各課題について説明する.

(A) について，「OS1 カーネルには一切変更を伴わない」としたので，OS1 は自分以外の OS が動作していることを認識せずに起動処理を行う．この際，OS1 が OS2 の走行環境を壊さないようにするため，OS2 が自身の走行環境を保護しながら OS1 の初期化処理を行う.

したがって，OS1 カーネルが初期化処理を実行する前に，一旦 OS2 に制御を移行する必要がある．

（B）について，OS2 が OS1 の初期化処理を行うが，OS2 は，OS1 の使用する実メモリ領域にアクセスできないように制限されている．したがって，OS2 が OS1 の初期化処理を行うためには，OS2 が OS1 の使用する実メモリ領域にアクセスできるようにする必要がある．また，OS2 は OS1 の初期化処理を監視し，処理内容に応じて適切な対処をする必要がある．なぜなら，OS1 の初期化処理が，CPU 内のレジスタ値を書き換えたり，OS2 が利用するメモリの内容を書き換えるなど，既に起動している OS2 の走行環境を破壊したり，影響を及ぼす可能性があるからである．したがって，OS2 の走行環境を壊すことを伴う OS1 の処理については，OS2 の走行環境を壊す部分の命令を，代替命令に置き換え実行させることで，OS2 の走行環境を保護する必要がある．

（C）について，OS2 が OS1 の初期化処理を行って OS1 を起動させる際，OS1 が OS2 の走行環境を壊さないように，OS2 の未使用の実メモリ領域を，OS1 の使用できる実メモリ領域として認識させる工夫を施す必要がある．

（D）について，OS2 は OS1 の初期化処理を，先に起動している OS2 の走行環境を壊すことなく，起動させる必要がある．OS2 は，OS2 の仮想空間上で走行するため，OS1 の初期化処理は OS2 の仮想空間上で実行される．つまり，一つの仮想空間上に二つ OS のカーネルが対応付けられる．したがって，各 OS のカーネルの仮想空間上での互いの位置が重ならない配置にする必要がある．この際，「OS1 カーネルには一切変更を伴わない」としたため，OS2 カーネルの位置を移動させ，OS1 カーネルの位置とは異なる配置にする必要がある．

4．設計方針

前節の課題を解決するため，以下の設計方針に基づいて開発を行う．

（１）OS2 カーネルの仮想空間上での位置の移動，及び OS1 カーネルロード後の OS2 への制御の移行は，OS2 と，OS1 カーネルの実メモリ領域へのブートローダに変更を加えることで実現する．

（２）BIOS が出力する実メモリ領域に関する情報を OS2 が変更することで，実メモリ領域の制限を行う．

（３）OS2 が OS1 の起動時処理を監視するために，ページディレクトリ，ページテーブルの内容を変更し，実メモリ領域の偽装と拡張を行う．

（４）OS1 の起動時処理内容への対処を行うために，CPU のトレースモードを利用する．

以下，各設計方針について説明する．

（１）について，OS2 カーネルの仮想空間上での位置の移動は OS2 に変更を加えることで実現する．また，OS1 カーネルを実メモリ領域にロードした後，OS2 へ制御を移行するために，OS1 カーネルの実メモリ領域へのロード処理部分に変更を加え，OS2 への制御移行ルーチンを実行するようにする．

（２）について，各 OS が一つの実メモリ上で独立に動作するためには，互いに不干渉な環境を構成する必要がある．そこで，各 OS に対して利用可能な実メモリ領域を制限して認識させる方法を考案し，各 OS があたかも異なる実メモリを用いて動作し，他 OS が使用している実メモリ領域にはアクセスしない環境を構成する必要がある．一方，「OS1 カーネルには一切変更を伴わない」としたので，OS の種類に依存せず，かつ各 OS に改造を加えることなく，各 OS が使用する実メモリ領域を制限する方法である必要がある．これらを満たすため，OS 初期化時に BIOS から取得する実メモリ情報を変更することで，各 OS の実メモリ領域を制限することができる．

（３）について，ページディレクトリ及びページテーブルのエントリ内容を変更することで，アクセスできるメモリ領域を拡張することができ，他 OS の使用しているメモリ領域までアクセスすることができる．

（４）について，本構成法では，CPU の走行モードをトレースモードにすることにより，ステップ実行毎に発生するデバグ例外を利用する．今回は実行前の命令を把握するため，例外処理ルーチンの中で命令列の先読みをする処理を行っている．

5．各 OS の起動方法

以降，前節の設計方針に基づいた，一つの計算機上に二つの OS を共存させるための起動方式と，各設計方針の具体的な実現方法について説明する．なお，PC は Intel 社製 Pentium の CPU を搭載した PC/AT 互換機，OS は Red Hat Linux7.2（kernel 2.4.7-10）を想定する．

以下に，OS2 が単独に走行している状態で，改造を加えていない OS1 を起動させる流れを述べる．

起動方法を説明する上で，次の技術が実現できていると仮定する．

（1）各 OS の使用できる実メモリ領域の制限，及び OS2 のみ仮想空間上の位置変更ができる．

（2）OS2 はトレースモードを利用し，OS1 の処理内容の取得，及び対処の実行ができる．

上記技術の具体的実現方法について，（1）については6章で説明し，（2）については7章で説明する．

5.1 OS2 の単独走行状態

一般的な OS は実メモリの低位アドレス部分にイメージを置く想定が多い．今回用いる Linux kernel 2.4.7-10 では，実メモリの1 MByte 番地にカーネルがロードされ，仮想空間上では 3GByte 番地に対応付けられる．ここで，OS2 は二番目に起動する OS1 がロードされる位置との衝突を避ける必要がある．したがって，メモリ領域の上位アドレス部分を OS2 が使用するメモリ領域として先に確保し，その領域のみを利用して走行させる．これは，OS2 のカーネルの書き換えを行うことにより実現できる．

OS2 が単独走行している状態を図 3 に示す．

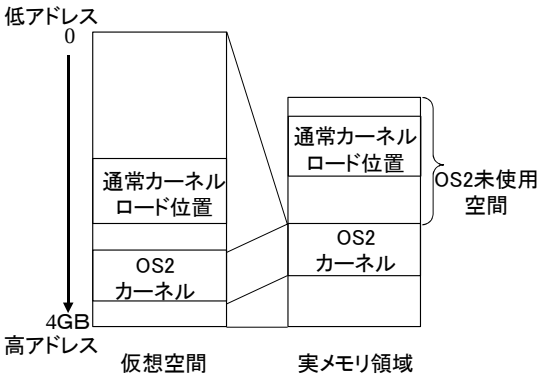

図 3　OS2 単独起動の様子

OS2 は，実メモリ領域の上位アドレス部分を利用して起動している．実メモリ領域の下位アドレス部分は，システム予約された領域として認識され，OS2 は利用できない．仮想空間においても，OS1 カーネルとの衝突を避けるため，通常対応付けられる仮想アドレスよりも上位の仮想アドレスに対応付けられる．

5.2 OS1 カーネルイメージのロード

制限された実メモリ領域上での走行が実現した後，OS2 は OS1 カーネルイメージを実メモリ上にロードする．ロードは以下の流れで行う．

（1）OS2 の走行環境の保存

（2）OS2 が再起動シーケンス実行

（3）OS1 のロード開始

（4）OS1 の初期化処理は実行せず OS2 に復帰

ここで走行環境とは，CPU 内のレジスタ値を指す．また，OS2 への復帰は，OS1 カーネルの実メモリ領域へのブートローダに変更を加えることで実現する．

5.3 OS2 のメモリ領域の拡張

OS1 カーネルは，OS1 の使用できる実メモリ領域にロードされている．したがって，OS2 は，OS1 が使用する領域を，使用できない領域と認識している．しかし，OS2 は，OS1 から自身を保護し，BIOS 情報を変更して OS1 に返すために，OS1 の初期化処理を監視しなくてはならない．そのため，OS1 がロードされている領域，つまり，OS2 が使用できないと認識している実メモリ領域も，OS2 がアクセスできる状態にする必要がある．これは，OS1 の実メモリ領域にも対応付けられているページディレクトリを作成し，そのページディレトリの実アドレスを制御レジスタ CR3 に設定することで可能となる．

作成するページディレクトリにおいて，OS1 の実メモリ領域の位置と，仮想空間との対応は，実アドレスとストレートマッピングさせる必要がある．ストレートマッピングとは，仮想アドレスが，そのまま実アドレスを表すことである．これは OS1 の初期化処理は OS2 の仮想空間上で実行されるが，OS1 の初期化処理は，リアルモード，ページング機能がセットされていないプロテクトモード，ページング機能がセットされたプロテクトモードと，モード移行しながら実行される．この際，リアルモード，ページングがセットされていないプロテクトモードでは，仮想アドレスがそのまま実アドレスとして認識されるため，仮想アドレスと実アドレスをストレートマッピングさせる必要がある．

OS1 カーネルがストレートマッピングされている様子を図 4 に示す．

実メモリ領域での OS1 カーネルの実アドレ

スと，仮想空間上で OS1 カーネルが対応付けられている仮想アドレスが同じ値となるようページディレクトリを作成する．

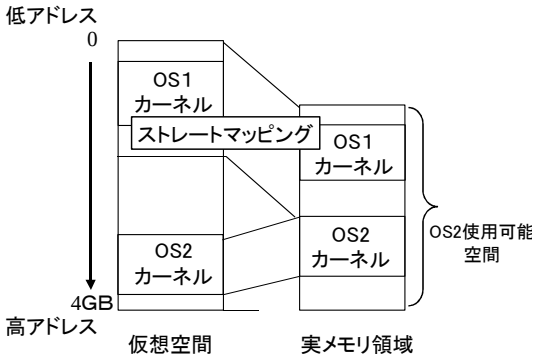

図 4　OS1 初期化処理の準備

5.4　OS2 の監視下での OS1 の初期化実行

OS2 が制御した状態で，トレースモードによるワンステップ実行により，OS1 の初期化処理を実行する．

CPU の動作モードとして，トレースモード実行がある．これは，機械語レベルでの命令実行において，ステップ命令毎にデバグ例外を発生させ，デバグ例外処理を実行するものである．したがって，OS2 が制御している状態で，CPU の走行モードをトレースモードにし，OS1 の初期化処理を実行させる．つまり，OS2 が，あたかも自分のコードを自分が監視し，実行内容を把握している中で処理を実行している状態にする．ステップ実行毎に発生するデバグ例外処理では，次に実行される処理内容を判別して，OS2 の走行環境を壊すような処理に対して，処理を行わない，あるいは代替処理を行う．

OS1 の初期化処理において，OS2 は次の処理を行う．

（1）OS1 が使用できる実メモリ領域の制限．
（2）OS2 の仮想空間上での，OS1 カーネルの位置の移動．

以下，各処理について説明する．

（1）について，OS1 の初期化処理において，OS1 の使用できる実メモリ領域を制限して認識させる必要がある．これは，OS 初期化時に BIOS から取得する実メモリ情報を変更することで，OS1 の実メモリ領域を制限することができる．具体的制限方法については 6 章で述べる．

（2）について，OS1 の初期化処理は，OS2 が監視しながら実行される．つまり，あたかも OS1 の初期化処理は，OS2 の一つのプロセスとして実行される．前述したように，OS の初期化処理は，リアルモード，ページング機能がセットされていないプロテクトモード，ページング機能がセットされたプロテクトモードと，モード移行を行いながら処理が行われていく．それぞれのモードでは，アドレスの認識の仕方に違いがある．しかし，OS2 の一つのプロセスとして，実行させるためには，モード移行に合わせて，OS2 が管理する仮想空間上の，二番目に起動する OS カーネルの位置を変更する必要がある．これは，ページディレクトリ及びページテーブルのエントリを変更し，仮想空間での対応アドレスを変更することで実現できる．仮想空間上での OS1 の位置変更の様子を図 5 に示す．

ページング機能がセットされたプロテクトモード移行時に合わせて，仮想空間上での OS1 カーネルの位置を変更する．

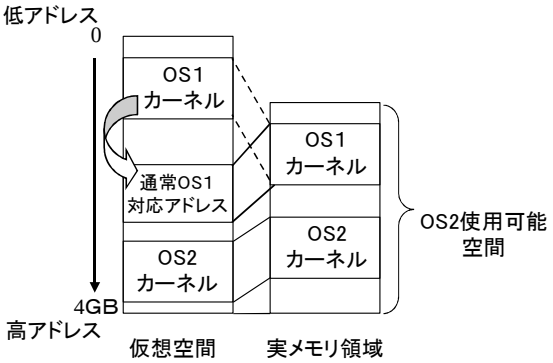

図 5　仮想アドレス変更の様子

以上の流れにより，一つの計算機上に，二つの OS が共存し，走行する環境が構成できる．

6．各 OS の実メモリ領域割り当て方法

6.1　メモリマップの制約方法

一般に PC 向けの OS は，使用できる実メモリ領域に関する情報を，BIOS から取得する．取得は以下の流れで行われる．

（1）OS は初期化処理時に，割込みベクタテーブルに登録されている実メモリ情報取得のための割込みを呼び出す．
（2）BIOS は実メモリに関する情報を実メモリ上に結果を出力する．
（3）OS は実メモリ上に出力された結果を用いて，メモリ管理に関する処理を実行する．

BIOS がメモリ上に出力した結果を変更するのみである本方法は，OS の種類に依存することなく，OS が使用可能な実メモリ領域の制限を行うことができる．また BIOS 自体の変更も一切伴わない．さらに，BIOS から取得した OS に渡すべき実メモリ情報のみを加工する本方法では，複雑なメモリ管理処理を改造することなく，実メモリ領域の制限が可能となる．

以下，BIOS が出力する実メモリ情報の具体的な変更方法について説明する．

BIOS が提供する実メモリ情報（メモリマップ）はエントリの列で成り立っており，エントリ一つひとつが，実メモリ上のある範囲の領域を表す．実メモリ領域の領域割り当ての様子を図 6 に示す．

実メモリ領域は，一エントリが表す範囲毎に分割され，属性に従って OS が使用可能な領域と，OS が使用不可能な領域とに分けられる．

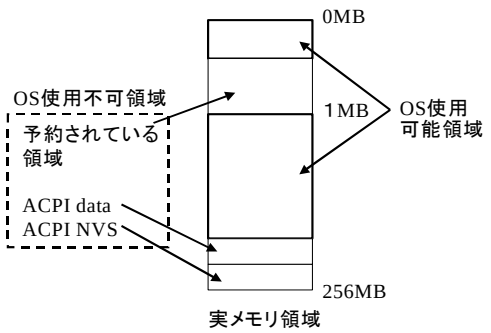

図 6　実メモリ領域の領域割り当て

各エントリには，対象とする実メモリの領域と属性が含まれる．属性には以下の種類がある．

（1）usable：OS が使用できる実メモリ領域．
（2）reserved：予約されている実メモリ領域．OS は使用できない．
（3）ACPI　data：ACPI が使用する領域．
（4）ACPI　NVS：ACPI が使用する領域．

OS は，「usable」領域を使用でき，他の属性の領域については使用できないと認識し，メモリ管理の初期化処理を行う．したがって，各 OS のメモリ管理の初期化処理前に，メモリマップのアドレス範囲，及び属性の書き換えを行うことにより，各 OS が使用できる実メモリ領域の制限が可能となる．

上記方法により，OS が使用できる実メモリ領域を制限して，認識させることができる．

6.2　OS ロード開始位置の変更

OS ロード開始位置が異なる場合にも，正しく OS が起動するようにするためには，OS が初期化処理の際に利用するページディレクトリ，ページテーブルを，OS が使用できる実メモリ領域及びカーネル開始位置に対応するように用意しておくことで解決できる．通常 OS が初期化時に利用するページディレクトリ，ページテーブルはカーネル内にあらかじめ領域を予約し，その領域にそれぞれのテーブルを用意して利用している．したがって，予約されたページディレクトリ及びページテーブル用領域のメモリ内容を，適切に変更することでカーネル開始位置が異なるページディレクトリ，ページテーブルを提供することができる．

7．トレース実行の処理方法

CPU 内のフラグレジスタのトレースビットをセットすることで，トレースモードに移行する．トレースモードで走行することで，一命令毎にデバグ例外が発生し，例外処理を実行する．

通常のデバグ例外発生時は以下の流れで例外処理を実行する．

（1）デバグ例外発生
（2）走行状態の保存
（3）デバグ例外処理
（4）走行状態の復帰
（5）例外終了

デバグ例外が発生すると，まずデバグ例外発生時の走行状態の保存が行われる．ここでいう走行状態とは，例外発生時に CPU 内のレジスタに格納されている各々の値を指す．

走行状態の保存処理は，CPU が行う処理と，OS が行う処理とに分けられる．CPU が行う処理で例外からの復帰アドレス，すなわち，次に実行される命令の仮想アドレスがスタックに積まれる．

OS が行う処理は，CPU が積むレジスタ以外のレジスタの値をスタックに積み，デバグ例外番号に対応するデバグ例外処理を呼び出すことである．デバグ例外処理を行った後，OS はスタックに保存されていた値をそれぞれのレジスタに格納し直すことで，デバグ例外発生時の走行環境に復帰する．

上記の処理の流れのうち，走行状態を保存した後のデバグ例外処理を利用して，OS2 は，以下の処理を行う．

（１）OS1 の初期化処理において，次に実行される命令内容を取得する．
（２）OS2 の走行環境を壊すような命令に対して，代替処理を実行するあるいは，命令を飛び越すことで，命令を実行させない．

以下，各処理の実現方法について説明する．

（１）について，次に行われる命令を把握するためには，次に実行される命令の仮想アドレスを獲得し，獲得した仮想アドレスが示す命令を取得することで，命令内容を知ることができる．次に実行される命令の仮想アドレスは前述したように，デバグ例外発生時の走行状態を保存する処理で，CPU が自動的にスタックに保存するので，CPU がスタックに積んだ値を，実行されるデバグ例外処理において，該当スタックから取得することで，次に行われる命令の仮想アドレス及び命令内容を知ることができる．

（２）について，デバグ例外処理の中で次に行われる命令の扱いを以下の通り分類する．

(a) そのまま実行させる．
(b) 実行させず，代替処理を実行させる．

以下，各分類項目について説明する．

(a) について，そのまま実行させる命令の場合，デバグ例外処理では何も行わず終了し，そのまま当該命令を実行させる．

(b) について，実行させない命令の場合には，命令内容に合わせた代替処理を行う．例えば，入出力装置をリセットする I/O 命令の場合，ニつ目に起動する OS が再び入出力装置をリセットする必要はないので，替わりにリセット結果が正常である応答を返す擬似命令に置き換えて実行させる．

実行させないと判別した命令を，実行させないためには，次に実行される命令を飛び越す必要がある．これは，デバグ例外処理終了後にスタックに保存されている値を利用して，デバグ例外発生時の走行環境に戻ることを利用する．つまり，スタックに積まれている次に実行される命令の仮想アドレスを変更することにより，命令を飛び越すことが実現できる．命令を飛び越す方法を図 7 に示す．

デバグ例外発生時に，スタックに積まれる次に実行される命令の仮想アドレスを取得することで，実行される命令の内容を知ることができる．命令内容に対して実行可，不可を判定し，命令の飛び越しを行う分，取得した仮想アドレスに加算した値を次に実行される命令の仮想アドレスとして入れ替えることにより，命令の飛び越しが実現できる．

デバグ例外を利用した次の命令取得により，処理速度の低下は免れないが，初期化処理終了後にデバグ例外を解除することにより，大きな影響はないと考えられる．

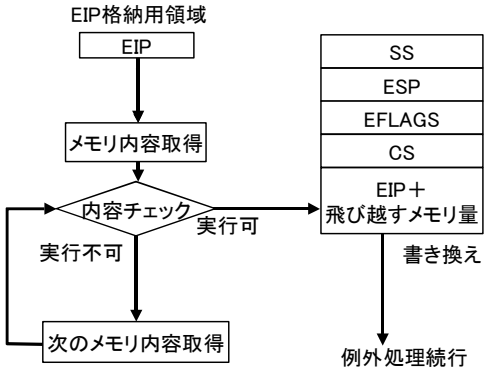

図 7　命令の飛び越し方法

8．おわりに

我々が提案した複数実計算機（MRM）の課題である各 OS の起動方式，及び起動方式において二つの OS で共有しなければならない実メモリの利用方法について述べた．具体的には，システム立ち上げ時に各 OS の使用するメモリ領域を排他的に確保する方式は，BIOS から取得するメモリ情報を変更することで，OS に依存することなく各 OS が使用できるメモリ領域を制限し分割割り当てすることを特徴とする．また，先に起動した OS が単独走行の状態から二番目の OS を割り当てられた実メモリ領域にロードし起動させる方式は，ニ番目に起動する OS を改造することなく，トレースモード下のデバグ例外処理を利用して命令の置き換えを行うことで二番目に起動する OS の初期化処理を行うという特徴を持つ．

今後は，提案したシステムを開発し，評価を行う予定である．

参考文献

[1]谷口秀夫，乃村能成，田中一男，大塚作一，井上友二，“ハードウェアを非共有する複数オペレーティングシステムの構成法，”情報処理学会研究報告，Vol.2002,No.79,pp.47-54,2002.

[2]箱守聰，谷口秀夫，“分散型リアルタイム OS:DIROS，”情報処理学会誌,Vol.36,No.8,pp.751-754,1995.