修 士 論 文

題 目

Mint オペレーティングシステムにおける 高速で柔軟な起動方式に関する研究

指導教員

報 告 者

中原 大貴

岡山大学大学院自然科学研究科電子情報システム工学専攻

平成 24 年 2 月 8 日 提出

要約

計算機のハードウェアは高性能化が進んでおり，1 つのチップに数百個のコアの搭載されたメニーコアプロセッサが登場している．この際に OS の観点からは，メニーコアを効率的に使用する方法が課題になる．そこで，メニーコアを効率的に扱う手段として，1 台の計算機上で複数の OS を動作させることが考えられる．

1 台の計算機上で複数の OS を動作させる手法の 1 つとして，仮想計算機方式がある．仮想計算機方式は，仮想マシンモニタにより，実計算機を複数の仮想的な計算機に見せかけ，この上で OS をそれぞれ独立して走行させられる．しかし，仮想計算機方式は，デバイスの仮想化によるオーバヘッドが発生するほか，仮想計算機と仮想マシンモニタに依存性があり，実計算機と同等の性能では動作しない．

一方，Mint オペレーティングシステムは，仮想化によらずに 1 台の計算機上で複数の Linux を独立に同時走行させるため，仮想化によるオーバヘッドの問題を解決できる．そして，OS 間通信を利用し，役割の異なる複数のカーネルを連携させて走行させることを目指している．

本論文では，メニーコアの効率的な利用を考慮し，Mint における高速で柔軟な起動方式に関する研究について述べる．まず，Kexec を応用した起動方式を実現した．Kexec とは，Linux の高速な再起動方式である．これにより，カーネルを配置する実メモリ領域を起動時に自由に変更可能となり，柔軟な起動を実現できる．また，多数の OS の起動に要する時間を短縮するため，起動処理を並列化した．さらに，役割の異なる複数のカーネルの連携例として 32/64bit カーネルの混載を実現した．そして，評価により，Kexec を応用することによって後続 OS の起動処理に要する時間は，BIOS とブートローダを介する Linux の起動処理と比較して約 21.1s 短縮可能であることを示した．また，2 つの後続 OS を並列起動する場合，逐次起動の約 55%の時間で起動可能であることを示した．

目次

図 目 次

表 目 次

第 1 章

はじめに

計算機のハードウェアは高性能化が進んでおり，1 つのチップに数百個のコアの搭載されたメニーコアプロセッサが登場している [1]．この際にオペレーティングシステム (以降，OS と呼ぶ) の観点からは，メニーコアを効率的に使用する方法が課題になる．そこで，メニーコアを効率的に扱う手段として，1 台の計算機上で複数の OS を動作させることが考えられる．

1 台の計算機上で複数の OS を動作させる手法の 1 つとして，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[2] や VMware[3] がある．VM 方式は，仮想マシンモニタ (VMM : Virtual Machine Monitor) により，実計算機を複数の仮想的な計算機に見せかけ，この上で OS をそれぞれ独立して走行させられる．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．

一方，Mint(Multiple Independent operating systems with New Technology) オペレーティングシステム [4](以降，Mint と呼ぶ) は，仮想化によらずに 1 台の計算機上で複数の Linux を独立に同時走行させるため，仮想化によるオーバヘッドの問題を解決できる．そして，OS 間通信を利用し，役割の異なる複数のカーネルを連携させて走行させることを目指している．

本論文では，メニーコアの効率的な利用を考慮し，Mint における高速で柔軟な起動方式に関する研究について述べる．まず，Kexec[5] を応用した起動方式を実現する．Kexec とは，Linux の高速な再起動方式である．これにより，カーネルを配置する実メモリ領域を起動時に自由に変更可能となり，柔軟な起動を実現できる．また，多数の OS の起動に要する時間を短縮するため，起動処理を並列化する．その後，Mint の応用例，評価，および関連研究について述べる．

第 2 章

Mintオペレーティングシステム

2.1 特徴

Mint はLinux カーネルを改変することによって開発されている．Mint は以下の特徴を持つ.

(1) 1 台の計算機上で 2 つ以上の Linux が走行する．

(2) 各 OS は，1 つ以上のコアを占有する．

(3) 全ての OS が相互に処理負荷の影響を与えない.

(4) 全ての OS が入出力性能を十分に利用できる．

これらの特徴を実現するため，1 台の計算機上で CPU，メモリ，およびデバイスといったハードウェア資源を効果的に分割し，占有する必要がある．Mint の各ハードウェアの分割と共有方法を以下に示す.

(1) CPU：コア単位で分割して各 OS で占有する．

(2) メモリ：走行させる OS の数だけ実メモリを空間分割して各 OS で占有する．

(3) デバイス：デバイス単位で分割し，各 OS が仮想化によらずに直接占有制御する．

Mint は，同時走行の実現のため，最初に走行する OS が後から走行する OS を順次起動する．本論文では，最初に走行する OS を先行 OS，後から走行する OS を後続 OS と呼称する．複数の後続 OS を走行させる場合は，起動順に後続 OS1，後続 OS2，· · · とする．図 2.1 は，3 つの OS を走行させる場合の Mint の構成例である．なお，Mint において共有メモリ領域を利用した OS 間通信が設計されている．

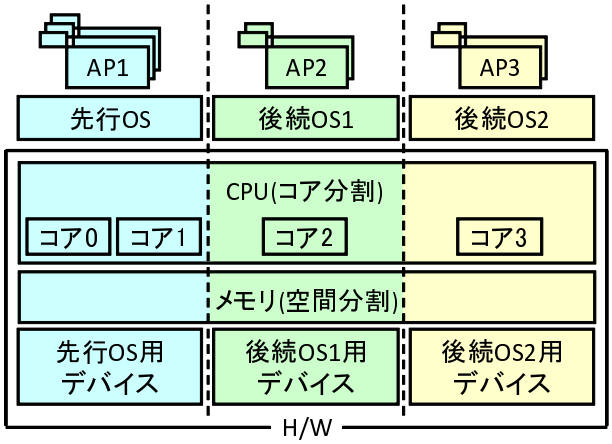

図　2.1　3 つの OS を走行させる場合の Mint の構成例

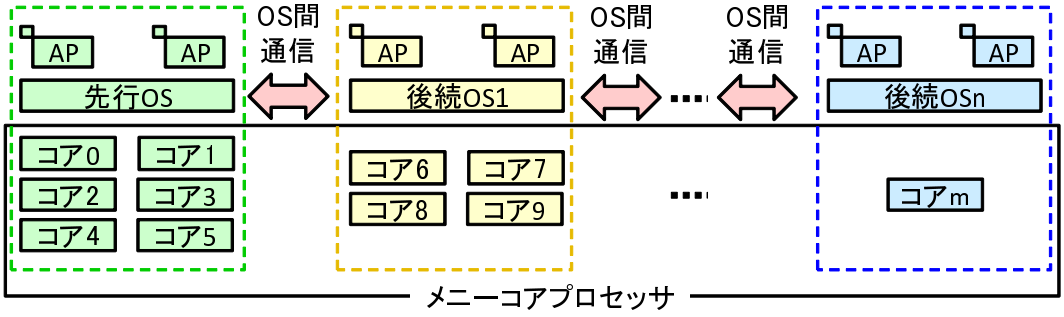

図　2.2　OS 間通信を利用した多数の OS の連携

2.2　メニーコアの利用に関する要求

Mint においてメニーコアを効率的に利用するため，同時に役割の異なる多数の OS を走行させ，OS 間通信を利用して密に連携させる．メニーコアの効率的な利用のため，OS 間通信を利用して多数の OS を連携させる様子を図 2.2 に示す．

Mint においてメニーコアを利用する際，起動に関して以下の 2 つの要求が存在する．

(要求 1) 柔軟な起動方式の実現

多数の OS を起動する場合，動作させる OS と同じ数のカーネルイメージが必要であるため，カーネルイメージの管理が煩雑になる．そこで，単一のカーネルイメージから複数の OS を起動可能となる柔軟な起動方式が必要になる．

(要求 2) 起動処理の高速化

Mint において，多数の OS を密に連携させる場合，全ての OS の起動処理を完了するための時間は長大になる．そこで，高速に多数の OS を起動させる必要がある．

(要求 1) を満たすため，3 章において，Kexec を応用した後続 OS 起動方式の実現について述べる．また，(要求 2) を満たすため，4 章において，後続 OS 起動処理の並列化について述べる．

第 3 章

Kexecを応用した後続OS起動方式の実現

3.1 Kexecを応用する目的

Kexec を後続 OS の起動に利用することを考える．Kexec とは，Linux の高速な再起動方式である．通常の再起動では，BIOS，ブートローダ，セットアップルーチンを経てカーネル本体の初期化に至る．これに対して，Kexec は，これらの処理を自身で代替し，カーネル本体の初期化から再起動を開始する．また，指定した実メモリ上に再起動用のカーネルイメージを配置する機能や，本来 BIOS やセットアップルーチンで行う初期化の結果をあらかじめ用意して再現する機能を持つ．これらの機能は，Mint において後続 OS の起動を実現するために有用である．特に，指定した実メモリ上に再起動用のカーネルイメージを配置する機能を利用することにより，単一のカーネルイメージを別々の実メモリ領域に配置可能となる．このことにより，単一のカーネルイメージから複数の OS を起動可能となる．そこで，Kexec を改変し，再起動処理の代わりに後続 OS の起動処理を実行する．

3.2 Kexecの処理流れ

まず，Kexec 本来の動作について，処理流れを図 3.1 に示し，説明する．

(1) 読み込み処理

Kexec は，まず，現在走行中のカーネルが保持するメモリマップを取得する．メモリマップには，実メモリの利用可能 (usable) 領域に関する情報が含まれており，この情報をもとに再起動用カーネルの配置先実アドレスを決定する．次に，再起動に必要な 4 つのデータを仮想メモリ上に配置する．

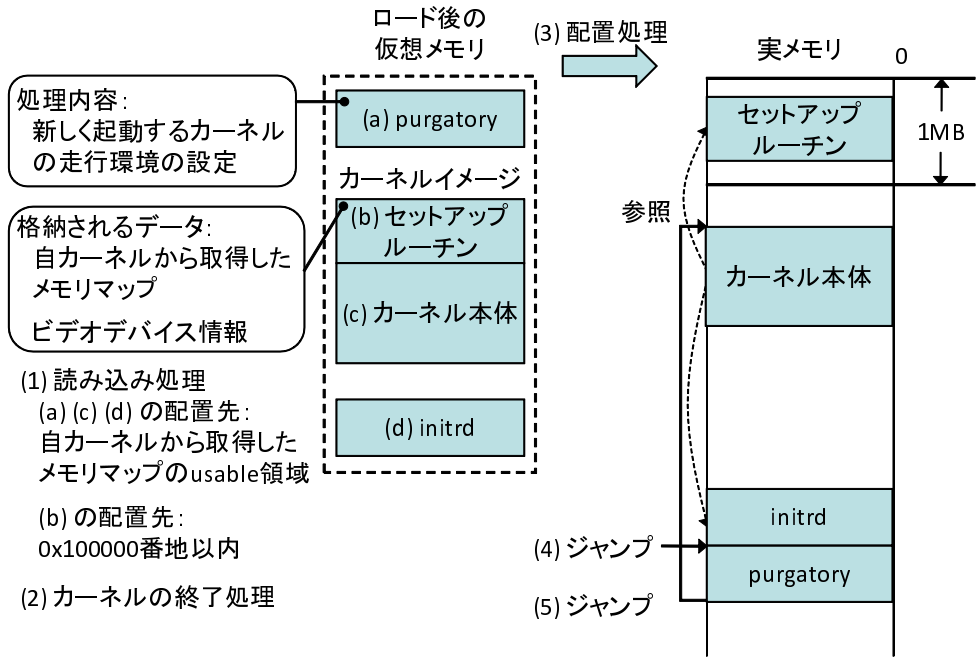

図　3.1　Kexec の処理流れ

(a)　purgatory (Kexec 固有の前処理)

(b)　セットアップルーチン

(c)　カーネル本体

(d)　initrd

これらは，(a) を除いて，通常のカーネル起動 (コールドブート) に利用されるデータと同一である．コールドブートでの (b) は，計算機に接続されているハードウェアの初期化やパラメータの取得を行い，カーネル起動に必要なデータを (b) のデータ領域に準備する．一方，Kexec では，(b) の実行で収集されるべきメモリマップやビデオデバイス情報などのハードウェア固有の情報は，再起動前のカーネル内部より取得し，(b) のデータ領域に再現する．このため，Kexec は (b) を走行しないものの，データを取得できる．また，(b) はリアルモードからプロテクトモードに切り替える処理も行うが，Kexec はプロテクトモードのままで再起動を実現するため，この処理は必要ない．そして，コアの初期化等の省略できない処理は (a) に代替させる．(b) のデータ領域は (c) によって利用されるため，実行コードとして (b) は不要なものの，(c) へのデータ受渡しのために必要とされる．

(2) カーネルの終了処理

再起動に備えて割り込みの無効化やコアの状態の初期化を行う．

(3) データ配置処理

(1) で仮想空間上に用意したデータを実メモリ上に配置する．

(4) purgatory へのジャンプ

purgatory では，スタック領域，GDT，各種セグメントレジスタ，および各種汎用レジスタの設定を行う．つまり，(b) の部分的な実行といえる．ただし，(b) がリアルモードで走行開始するのに対して，purgatory は，プロテクトモードで走行する．

(5) カーネル本体へのジャンプ

カーネル本体が起動する．

3.3 課題

Kexec は，カーネルを再起動する機能であるため，後続 OS 起動用に改変する必要がある．改変には，以下の 4 つの課題がある．

(課題 1) コアの起動

カーネルを再起動するのではなく，別のコアを起動させ，このコアにカーネルを起動させる必要がある．

(課題 2) プロテクトモードへの切り替え

起動直後のコアはリアルモードで走行する．一方，Kexec は，プロテクトモードのままで再起動を実現するため，リアルモードからプロテクトモードに切り替える処理を加える必要がある．

(課題 3) 後続 OS 用のメモリマップの用意

Kexec は，図 3.1-(1) の読み込み処理において，再起動のために必要なメモリマップを起動中のカーネルから取得する．しかし，後続 OS 起動のためには，メモリマップを起動中のカーネルから取得するのではなく，後続 OS に合わせて外部から与えられるようにする必要がある．

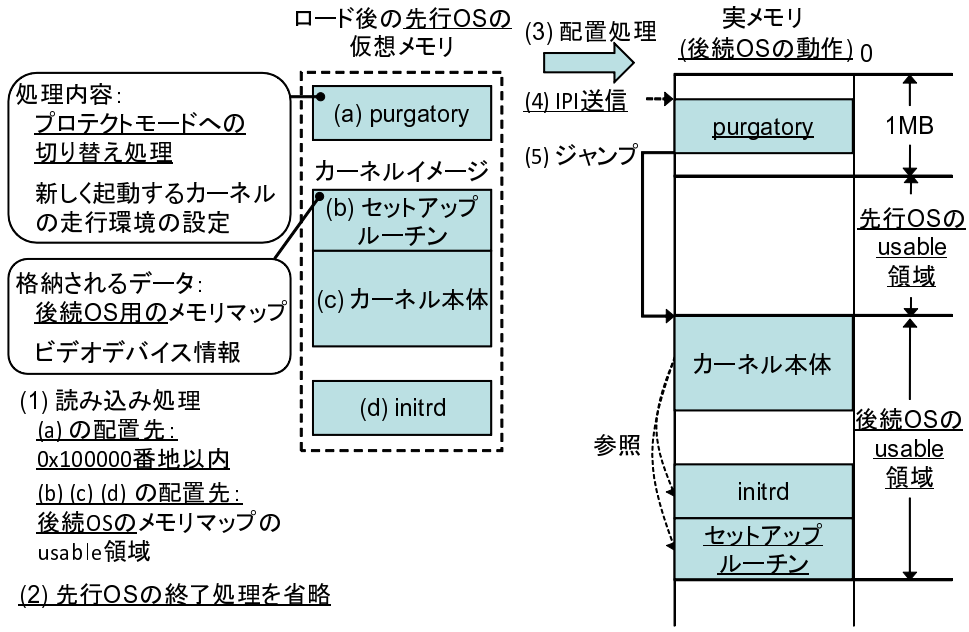

図　3.2　Kexec を応用した後続 OS 起動方式の処理流れ

(課題 4) 先行するカーネルの走行環境の保護

図 3.1-(2) のカーネルの終了処理は，そのまま実行すると先行するカーネルの走行環境を破壊する．これらの処理を省略する必要がある．

3.4　対処

3.3 節で示した (課題 1) から (課題 4) を解決するため，それぞれ (対処 1) から (対処 4) を示す．Kexec を応用した後続 OS 起動方式の処理流れを図 3.2 に示す．課題への対処のために Kexec に対して行った改変は，図 3.2 中では下線部に示している．

(対処 1) IPI の送信によるコアの起動

カーネルを実メモリ上に配置した後，purgatory の先頭アドレスにジャンプするのではなく，起動対象のコアに IPI(プロセッサ間割り込み) を送信するように改変した．このため，起動対象のコアを選択するオプションを Kexec に追加した．IPI を用いて purgatory の先頭アドレスの情報を送信することで，IPI を受け取ったコアは，purgatory の先頭アドレスから処理を実行する．起動直後のコアはリアルモードで走行するため，purgatory は，実メモリの 0x100000 番地以内に存在しなければならない．このため，purgatory

の配置場所を 0x100000 番地以内に固定した．また，IPI を送信したコア，つまり，先行 OS として動作するコアは，Kexec の処理を終了させるように改変した．

(対処 2) プロテクトモードへの切り替え処理の追加

リアルモードからプロテクトモードへの切り替え処理を purgatory の先頭に追加した．

(対処 3) メモリマップの改変

メモリマップの取得に関して，自カーネルの情報から取得する代わりに，外部から与えられた値を使用するように改変した．また，purgatory 以外のデータをここで作成したメモリマップの usable 領域に配置するように改変した．元来，セットアップルーチンはリアルモードで走行するため，実メモリの 0x100000 番地以内に配置しなければならない．しかし，3.3 節の (1) で示したように，セットアップルーチンはデータの受け渡し以外の目的では使われないため，0x100000 番地以上の usable 領域に配置するように改変した．

(対処 4) 終了処理の省略

先行するカーネルの走行環境を保護するため，終了処理を行う関数の呼び出しを省略した．また，以下の 5 つの処理を行わないように省略した．

(1) Local IRQ の無効化

(2) 各種セグメントレジスタの初期化

(3) GDT の初期化

(4) IDT の初期化

(5) 各種汎用レジスタの初期化

以上の対処により，Kexec を応用して後続 OS を起動することができる．

3.5 カーネルイメージの単一化

走行する OS ごとに異なるカーネルイメージを必要とする場合，カーネルの構成管理が複雑である．そこで，カーネルイメージを単一化し，Kexec を応用して単一化したカーネルイメージから複数の OS を起動する．

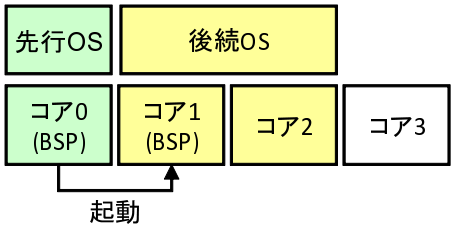

図 3.3 CPU の分割，指定方法

それぞれの OS の差分は，原則として占有するハードウェア資源のみである．このため，カーネルイメージを単一化するために，OS 起動時に占有するハードウェア資源を指定しなければならない．ここで注意として，先行 OS はブートローダから起動し，後続 OS は先行 OS から Kexec を応用して起動する．このため，先行 OS 起動時に指定可能なパラメータは，ブートオプションだけである．また，後続 OS 起動時に指定可能なパラメータは， Kexec の実行オプションとブートオプションである．

指定が必要なハードウェア資源は，以下の 3 種類である．

(1) CPU

各 OS が占有するコアを指定する必要がある．

(2) メモリ

各カーネルの配置先アドレスを指定する必要がある．また，各 OS が利用するメモリ領域を指定する必要がある．

(3) デバイス

各 OS が占有するデバイスを指定する必要がある．

これらのハードウェア資源を分割し，指定する方法をそれぞれ以下に示す．

(1) CPU の分割，指定方法

CPU の分割，指定方法を図 3.3 に示す．各 OS は，複数のコアを占有する可能性がある．このため，(A)BSP(起動対象のコア) と (B) 占有するコアの個数を指定する．先行 OS の BSP は，ブートローダから起動することで自動的に先頭のコア (コア 0) になるため，指定しない．後続 OS の BSP は，3.4 節の (対処 1) で独自に Kexec に追加したオプションを用いて指定する．占有するコアの個数の指定には，Linux の既存機能 maxcpus ブートオプションを用いる．

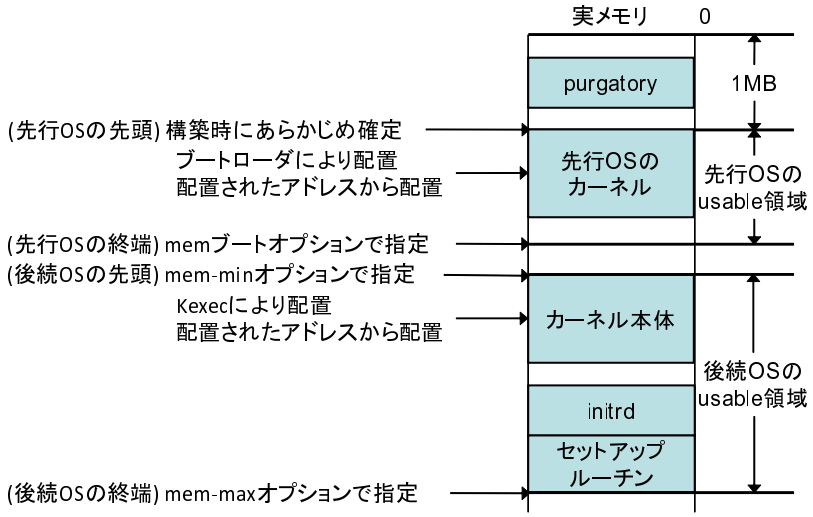

図 3.4 実メモリの分割，指定方法

(2) 実メモリの分割，指定方法

実メモリの分割，指定方法を図 3.4 に示す.

(A) 各カーネルの配置先アドレス

通常の Linux カーネルの配置先アドレスは，カーネルを構築した時点で決定される．このため，複数のカーネルをそれぞれ異なる実メモリ領域に配置するためには，複数のカーネルイメージを作成する必要があった．そこで，単一のカーネルイメージをそれぞれ異なるアドレスに配置するため，再配置可能な圧縮カーネルイメージを用いる．圧縮カーネルイメージを再配置可能にする機能は，Kexec を応用してカーネルクラッシュダンプを取得する kdump[6] のために Linux に用意されており，これを利用する．再配置可能な圧縮カーネルイメージは，再配置情報を所持しており，圧縮カーネルイメージが配置されたアドレスから配置される．

(a) 先行 OS のカーネルイメージは，ブートローダによって実メモリの先頭 (0x100000) 付近に配置される．このため，Mint でカーネルイメージを単一化する場合，先行 OS に限っては，任意の実アドレスからカーネルを配置することはできない.

(b) 後続 OS に関しては，再配置可能な圧縮カーネルイメージを任意のアドレスに配置するため，Kexec の既存のオプション mem-min を用いる．このオプションを用いることで，データの配置先の先頭アドレスを指定可能である．通常

は，3.2 節で示した 4 つのデータを全て mem-min で指定した実メモリアドレス以降の領域に配置する．しかし，purgatory は 0x100000 番地以内に配置する必要がある．このため，purgatory 以外のデータを mem-min で指定した実メモリ領域に配置するように改変した．purgatory は，mem-min の影響を受けず，0x100000 番地以内に配置する．以上により，任意のアドレスに後続 OS のカーネルを配置可能である．

(B) 各 OS が利用する実メモリ領域

利用する実メモリ (usable) 領域の先頭アドレスと終端アドレスを指定する．

(a) 先行 OS の先頭アドレスは，カーネル構築時のコンフィグオプションを用いて指定し，カーネル構築時に確定させておく．先行 OS の終端アドレスは，Linux の既存機能 mem ブートオプションを用いて指定する．

(b) 後続 OS では，3.4 節の (対処 3) で示したように，メモリマップを改変することで，先行 OS のために確定した実メモリ領域とは別の領域を指定可能である．後続 OS 用のメモリマップの先頭アドレスと終端アドレスは，それぞれ mem-min とデータ配置先の終端アドレスを指定するオプション mem-max を用いて指定する．

(3) デバイスの分割，指定方法

各 OS で占有するデバイスをあらかじめ確定させる．

例. 先行 OS : HDD1, VGA, キーボード
後続 OS1 : HDD2, シリアルポート
後続 OS2 : HDD3, NIC(Network Interface Card)

独自に追加したブートオプションを用いて占有するデバイスを指定する．デバイスドライバの登録処理を行う部分は，全てこのブートオプションの値によって処理を分岐させる．

以上の方法で単一のカーネルイメージに対して各 OS の起動時に異なるハードウェア資源を指定することにより，それぞれの OS に異なるハードウェア資源を割り当て可能となり，カーネルイメージの単一化が可能となる．

第 4 章

後続OS起動処理の並列化

4.1　起動処理の解析と目的

後続 OS の起動において多くの時間を要する部分を分析するため，Intel Core i7(2.8GHz) を搭載した計算機において後続 OS の起動に要する時間を測定した．後続 OS の起動に要する時間を図 4.1 に示し，それぞれの処理について以下で説明する．

(1) 読み込み処理

後続 OS の起動に必要なカーネルイメージ，initrd，および purgatory を先行 OS の HDD から読み込む．

(2) データ配置処理

(1) で読み込んだデータを実メモリ上に配置する．

(3) IPI の送信処理

IPI を利用して未使用のコアを起動し，起動したコアは purgatory の実行を開始する．

(4) purgatory の処理

purgatory においてコアの走行モードをリアルモードからプロテクトモードに切り替える．

(5) カーネル起動処理

メモリ管理，割り込み，およびデバイスといった各種初期化処理を実行し，カーネルを起動する．

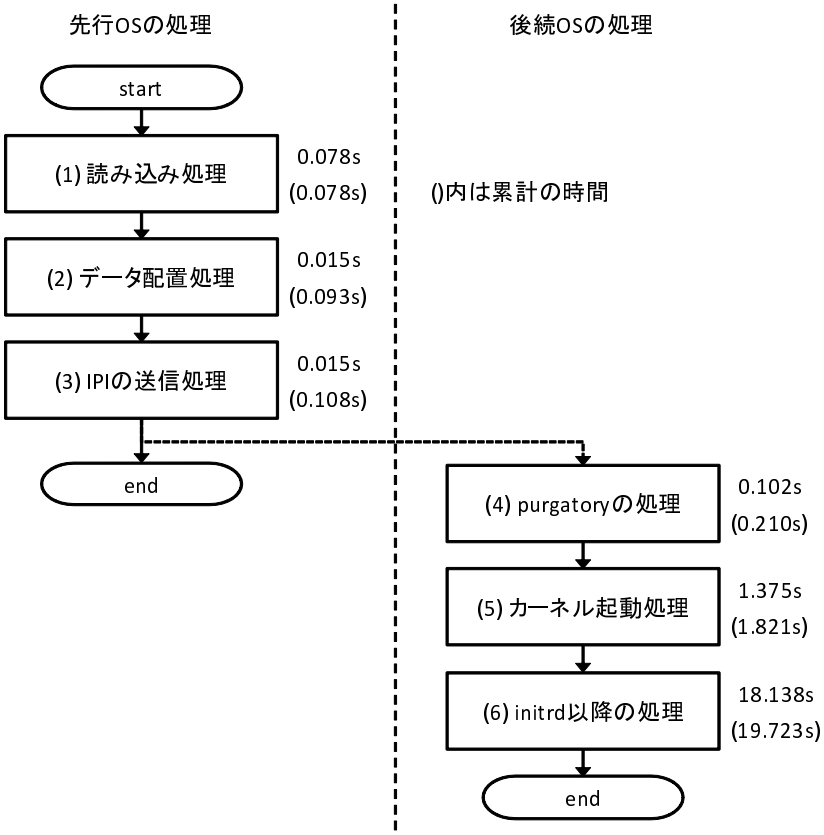

図　4.1　後続 OS の起動に要する時間

(6) initrd 以降の処理

initrd を実行し，後続 OS の HDD をマウントする．そして，デーモン起動処理を実行する．

(1) から (3) は先行 OS の処理であり，(4) から (6) は後続 OS の処理である．これらの処理の内，(6) の部分に要する時間は全体の約 92%に相当する．そこで，少なくとも後続 OS の起動の内の約 92%を要する (6) を並列に実行するだけでも大幅な起動時間の短縮を見込める．そこで，多数の OS の起動に要する時間を短縮するため，後続 OS を並列に起動する．

(6) の部分は既にカーネル起動処理が完了しているため，(6) 自体は並列に実行可能である．しかし，(6) 以外の部分に並列に実行不可能な箇所が存在する．そこで，並列に実行不可能な箇所に対して排他制御を実現することにより，(6) の部分を確実に並列実行する．なお，複数の後続 OS はそれぞれ自身の占有する HDD から読み込みを実行する．(6) の部分では入出力に要する時間が多いと予想されるが，Mint の各 OS は入出力性能を十分に発揮可能であるため，並列に実行しても入出力に要する時間が問題になることはない．

4.2　課題

複数の後続 OS を並列に起動するため，後続 OS での処理 (図 4.1 の (4) から (6)) を複数 OS で並列に実行する．この際，以下の 3 つに対して排他制御の要否と対処法を明らかにする必要がある．

(1) デバイス

(2) 実メモリ

(3) I/O APIC(割り込み制御)

4.3　対処

4.3.1　デバイス

Mint は各 OS がそれぞれ異なるデバイスを占有するため，デバイスに対する排他制御は不要である．

4.3.2　実メモリ

図 4.1 の (2) データ配置処理におけるデータ配置位置について図 4.2 に示し，以下で説明する．

カーネルイメージと initrd に関しては各 OS の usable 領域に排他的に配置するため，排他制御は不要である．一方，purgatory に関してはリアルモードで走行するため，実メモリの先頭から 1MB 以内の領域に配置する．この際，起動の並列化を考慮していないため，全ての後続 OS の purgatory を同じ位置に配置する．purgatory には，カーネルへのジャンプ先のアドレスが含まれているため，それぞれの後続 OS に対応した別々の purgatory が必要となる．このため，全ての後続 OS の purgatory を同じ位置に配置すると，並列に purgatory を実行不可能となり，排他制御が必要となる．

そこで，purgatory に対する排他制御の方法として，以下の 3 つが考えられる．

(対処案 1) 全ての後続 OS の purgatory を同じ位置に配置し，各 OS が purgatory の配置時にロックを獲得し，purgatory の走行が終わった時点でロックを解放する．

(対処案 2) 全ての後続 OS の purgatory をそれぞれ別々の位置に配置する．

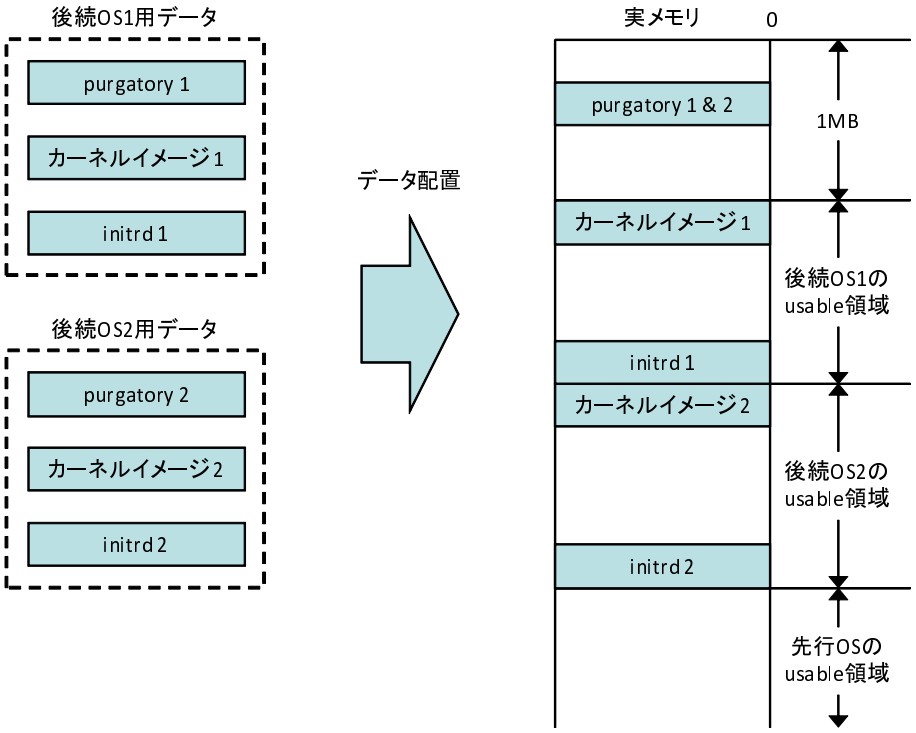

図　4.2　データ配置処理におけるデータ配置位置

(対処案 3) purgatory を 1 つに統合してカーネルへのジャンプ先アドレスを後続 OS ごとに変更する．

これらの対処方法の内，(対処案 2) が最も実装が容易であり，起動の並列化という目的に適している．そこで，(対処案 2) で対処する．

pugatory の配置場所を変更した後の配置例を図 4.3 に示し，purgatory の配置方法について以下で説明する．複数の後続 OS の purgatory をそれぞれ別々の位置に配置するため，起動対象のコア (BSP) のコア ID によって purgatory の配置位置を決定するようにした．具体的には，1 個あたりの purgatory のサイズは 0x9000(36KB) であるため，先頭から，0x9000 ×コア ID の位置に purgatory を配置する．

ただし，1 個あたりの purgatory のサイズは 0x9000(36KB) であるため，1MB 以内に配置できる purgatory の数は 28(1MB ÷ 36KB) 個以下に制限されるという問題がある．今後，29 コア以上の CPU を利用する場合は，29 コア目以降のコアを BSP にできない．この問題への対処として，(対処案 3) による対処が考えられる．

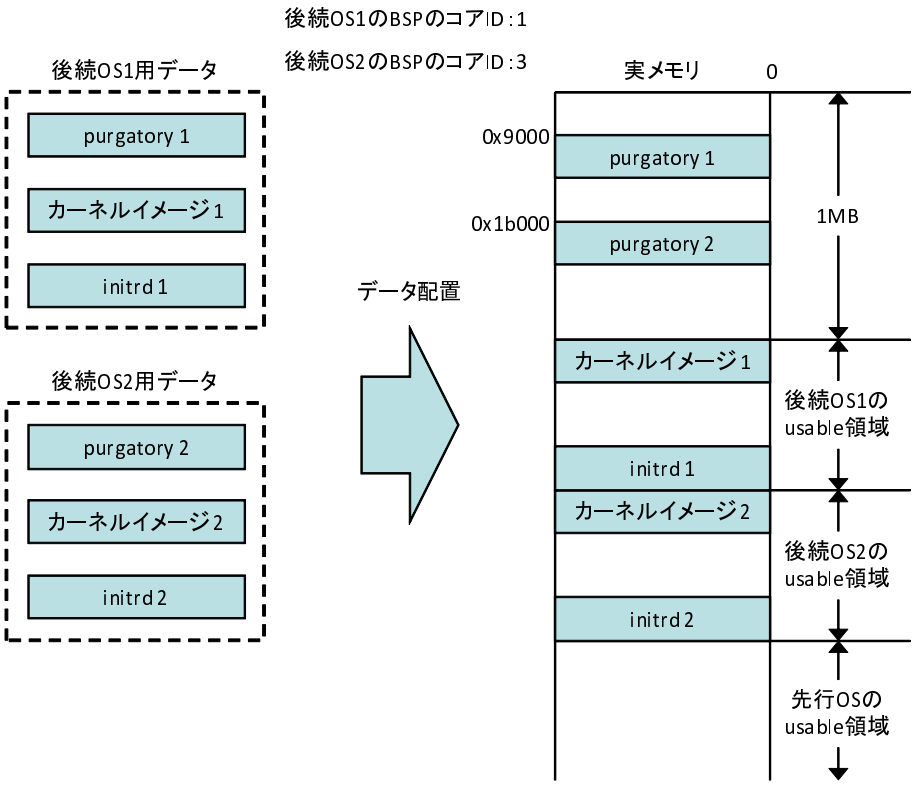

図　4.3　pugatory の配置場所を変更した後の配置例

4.3.3　I/O APIC

起動処理における割り込みの初期化やデバイスの登録処理では，I/O APIC を操作する．複数 OS でこれらの処理を同時に実行すると問題が生じる可能性がある．このため，図 4.1 の (5) における割り込みの初期化とデバイスの登録処理を含む部分に対し，共有メモリ領域を利用した排他制御機構を一括で実現することとする．

4.4　実現

4.3 節で示した排他制御が十分であることを確認するため，2 つの後続 OS の起動処理において複数の箇所で同期をさせつつ起動の可否を調査した．具体的には，purgatory の先頭，カーネル起動処理の先頭，および initrd 実行前で同期をさせた．この結果，purgatory と initrd 以降に関しては，同時に実行しても問題は発生しないことを確認した．また，カーネル起動処理に関しては，排他制御によって問題が発生することなく走行可能であることを確認した．

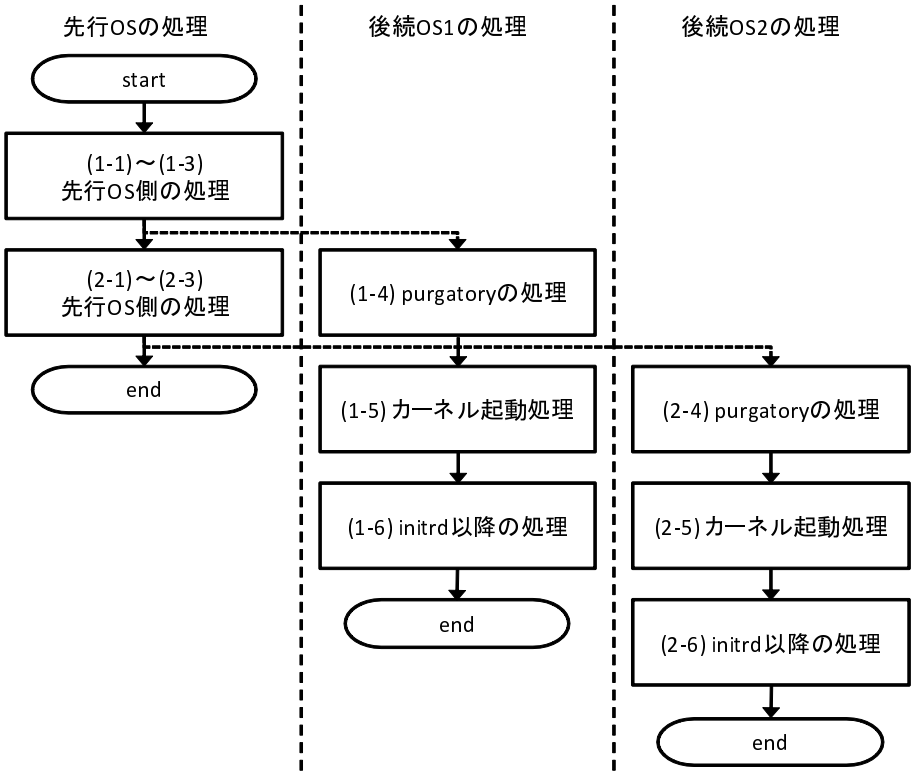

図　4.4　2 つの後続 OS の並列起動

なお，排他制御なしにカーネル起動処理を同時に実行した場合は問題が発生し，計算機が再起動する．

2 つの後続 OS の並列起動の様子を図 4.4 に示す．先行 OS 側の処理は逐次実行し，purgatory 以降の処理を並列実行する．

第 5 章

Mintの応用例

5.1 複数OSの連携

本章では，今後の Mint の応用例について述べる．Mint で走行する複数の OS はメモリを共有する．そこで，共有メモリ領域を作成し，共有メモリ領域を利用した OS 間通信が設計されている．今後の Mint は，OS 間通信を利用して役割の異なる複数のカーネルを連携させることを目指す.

役割の異なるカーネルの連携について以下に 3 つの例を示す.

(1) 32bit カーネルと 64bit カーネルを同時に走行させることにより，大量のメモリと 32bit OS 用のソフトウェアの両方を利用する．これについては，5.2 節で解説する．

(2) 複数の OS を監視用 OS とサービス用 OS に分け，ヘルスチェック機能を実現する [7]．ヘルスチェック機能とは，OS を検査して異常を検出するための機能である．ネットワークから切り離された監視用 OS がサービス用 OS を監視し，サービス用が改ざんを伴う攻撃を受けた場合，サービス用 OS を停止させる．Mint は複数の OS でメモリを共有するため，監視用 OS は共有されたメモリによって検査を行う．これにより，外部から監視用 OS の存在を隠蔽可能であり，信頼性の高いシステムを構築できる．

監視用 OS の役割は監視のみであり，少ないハードウェア資源で動作可能である．そこで，initrd を拡張して initrd 内で作業可能にすることにより，HDD の占有なしに監視用 OS の起動を実現している．

(3) 異なるディストリビューションの Linux を走行させ，それぞれのディストリビューションに適した処理を実行させる．

また，別の連携の例として OS 間の計算機資源の移譲が挙げられる．1 つの OS の処理負荷が高まった場合に，どの OS も占有していない計算機資源またはを他の OS が占有する計算機資源を処理負荷の高まった OS に移譲することで負荷の分散を行う．移譲の対象となる計算機資源は以下の 3 つである．

(1) CPU

コア単位で移譲する．Linux に存在する CPU のホットプラグ機能を利用して移譲を実現する．

(2) メモリ

実メモリの空間分割の割合を変更することで移譲を実現する．

(3) デバイス

デバイス単位で移譲する．割り込みルーティングを変更することで移譲を実現する．移譲の利用例として NIC を複数枚搭載した計算機において，通信負荷が増大した OS に NIC を移譲することで帯域を増やすことが挙げられる．

以上により，処理負荷に柔軟に対応可能なシステムを構築可能である．

5.2 32/64bit カーネル混載の実現

5.2.1 目的

64bit CPU の普及に伴い，OS も 64bit 化が進んでいる．64bit OS の下で走行するソフトウェアは，広大なアドレス空間を利用可能であり，一度に処理できるデータの量も大きいため，システム性能の大幅な向上が見込める．しかし，64bit OS は 32bit OS 用の一部のソフトウェアと互換性がない．特に，32bit OS 用に記述されたカーネルやデバイスドライバを始めとするシステムソフトウェアは，64bit OS では動作しない．

そこで，Mint において 32bit カーネルと 64bit カーネルを同時に走行させる．これにより，32bit OS では得られない広大なメモリ領域と処理性能，64bit OS では利用不可能な 32bit OS 用のソフトウェアをシステム全体として使用可能になる．

表 5.1 x86-64 アーキテクチャにおける CPU の走行モード

モード	サブモード	必要な OS	アドレスサイズ (bit)
Legacy モード	リアルモード	16bit OS	16
	プロテクトモード	32bit OS	32/16
	仮想 8086 モード	32bit OS	16
Long モード	64bit モード	64bit OS	64
	互換モード	64bit OS	32/16

5.2.2 64bit アーキテクチャの特徴

互換性の問題は，x86-64 アーキテクチャの特徴に起因する．x86-64 アーキテクチャは，x86 アーキテクチャを 64bit に拡張したアーキテクチャである．Intel Core 2 以降の Intel 製 CPU の大半は，x86-64 アーキテクチャである．x86-64 アーキテクチャの CPU が持つ走行モードを表 5.1 に示す.

x86-64 アーキテクチャの走行モードは大きく分けて Legacy モードと Long モードに分かれる．Legacy モードは，32bit OS 用のモードであり，従来の x86 アーキテクチャと完全な互換性を持つ．Legacy モードは，3 つのサブモードを持つ．一方，Long モードは，64bit OS 用のモードである．Long モードは，2 つのサブモードを持つ．各サブモードを以下で説明する．

(1) リアルモード

CPU の起動時の走行モードであり，16bit のソフトウェアを走行させるモードである．例えば，Linux の起動処理 (BIOS，ブートローダ，およびカーネルのセットアップルーチン) は，このモードで走行する．ページングは使用不可能であり，実メモリの先頭から 1MB までしかアクセスできない.

(2) プロテクトモード

16bit または 32bit のソフトウェアを走行させるモードである．x86 アーキテクチャ対応の OS，例えば Linux の 32bit 版はこのモードで走行する.

(3) 仮想 8086 モード

プロテクトモードにおいてリアルモードのソフトウェアを走行させるためのモードである.

(4) 64bit モード

64bit のソフトウェアを走行させるモードである．

(5) 互換モード

16bit のソフトウェアまたは 32bit のソフトウェアを走行させるモードである．4GB の仮想アドレス空間を持ち，従来の 32bit アプリケーションソフトウェアの大半を修正なしで走行させられる．OS の観点から見た本モードは，64bit モードとして機能する．具体的には，アドレス変換方式，割り込み処理，および例外処理について，64bit メカニズムが使用される．つまり，本モードは，64bit OS を要求する．

なお，Long モードの 2 つのサブモードは，コードセグメントごとに選択可能である．したがって，OS 制御により，64bit で走行する AP と 32bit で走行する AP を Long モードの CPU 上で同時に実行可能である．

5.2.3 互換性の問題

プロテクトモードと 64bit モードでは，アドレスのビット幅が異なる．ソフトウェアのバイナリファイルには，アドレスに関する情報が含まれている．したがって，アドレスのビット幅が異なることにより，32bit OS 用のソフトウェアのバイナリファイルは 64bit モードでは動作しない．

32bit OS 用のソフトウェアのソースコードが存在する場合，64bit モード用に再コンパイルすれば，64bit モードで動作可能になる場合がある．しかし，アドレスのビット幅について考慮されていない 32bit OS 用のソフトウェアは，ソースコードからコンパイルしても 64bit モードでは動作しない．例えば，C 言語で記述されたプログラムにおいて，ポインタと他のデータ型が同じサイズであるという前提でソースコードが記述されている 32bit OS 用のソフトウェアは，64bit モードでは動作しない．

AP に関しては，64bit モードと互換モードをコードセグメントごとに選択可能であるため，64bit OS においても 32bit OS 用の AP をバイナリファイルの状態で利用可能である．しかし，カーネルモードで走行するソフトウェアは，64bit モードと互換モードを切り替えることができない．このため，カーネルモードで走行するカーネルやデバイスドライバは 64bit モード用に記述されている必要がある．例えば，32bit OS 用のデバイスドライバがバイナリファイルでしか提供されていない場合，そのデバイスは 64bit OS で利用できない．

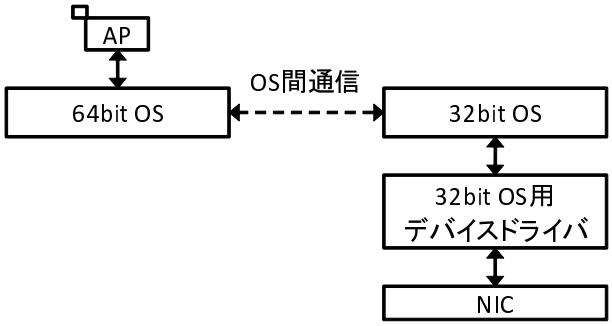

図 5.1 64bit OS では利用不可能な NIC を 32bit OS を介して利用する例

5.2.4 混載による利点

CPU の走行モードは，コアごとに切り替え可能である．そこで，互換性の問題を解決するため，Mint において 32bit カーネルと 64bit カーネルを同時に走行させる．これにより，32bit OS では得られない広大なメモリ領域と処理性能，64bit OS では利用不可能な 32bit OS 用のソフトウェアをシステム全体として使用可能になる．

例えば，64bit OS では利用不可能な NIC を 32bit OS を介して利用する例を図 5.1 に示す．64bit OS 用のデバイスドライバが提供されていない NIC を利用する場合，64bit の AP が NIC に処理を要求する際は，OS 間通信を利用して 32bit OS と 32bit OS 用デバイスドライバを介して NIC を利用する．

VM 方式では上記の方式を実現できない．なぜなら，VM 方式において VM を走行させる OS(以降，ホスト OS と呼ぶ) と VM 上で走行する OS(以降，ゲスト OS と呼ぶ) の間には，以下の制約が存在するためである．

(制約 1) ホスト OS を 32bit OS，ゲスト OS を 64bit OS とする構成はできない

(制約 2) ホスト OS で認識できないデバイスは，ゲスト OS から利用できない

つまり，ホスト OS を 32bit OS とした場合は，(制約 1) により，図 5.1 の構成は不可能であり，逆にホスト OS を 64bit OS とした場合は，(制約 2) が問題になる．

また，32bit カーネルは 64bit カーネルよりも必要な実メモリ量が少ないという特徴がある．具体的に，Fedora 14 において Linux カーネル 2.6.39 のバイナリファイルのサイズと走行時のメモリ使用量を調査した．この結果，64bit Linux は，32bit Linux と比較してカーネルのバイナリファイルのサイズが約 58%増加し，カーネル走行時の使用メモリ量が約 51%増加することを確認した．このため，32/64bit カーネルを混載する際の利用例として，4GB 以

上の実メモリを必要とする OS を 64bit で動作させ，実メモリをあまり必要としない OS を 32bit で動作させることが考えられる．これにより，32bit カーネルは 64bit カーネルよりも必要な実メモリ量が少ないため，メモリの節約が可能となる．

5.2.5 課題

Mint は x86 アーキテクチャにしか対応していないため，32/64bit カーネルの混載を実現するためには，Mint を x86-64 アーキテクチャに対応させる必要がある．このため，課題は Mint の x86-64 アーキテクチャへの対応となる．

Linux カーネルは x86 アーキテクチャと x86-64 アーキテクチャのソースコードの大部分を共有している．原則として共有部分の処理に関しては，Mint の x86-64 アーキテクチャ対応のために新しく改変する必要はない．

Mint を x86-64 アーキテクチャに対応させるための課題を以下に示す．

(課題 1) CPU の走行モードの切り替え

32bit OS は Legacy モードのプロテクトモードで走行する．一方，64bit OS は Long モードで走行し，個別のプロセスごとに 64bit モードか互換モードを選択する．また，起動直後のコアはリアルモードで走行する．後続 OS の起動時は，これらの CPU の走行モードの切り替えを意識する必要がある．

(課題 2) ソースコードが共有されていない処理の改変

x86 アーキテクチャ対応の Mint では，以下の 6 つを実現するために Linux カーネルを改変した [4]．

(1) 後続 OS の起動

(2) コアの分割と占有

(3) 実メモリの分割と占有

(4) デバイスの分割と占有

(5) 割り込みの制御

(6) 終了処理と再起動処理

これらの改変の中で x86 アーキテクチャと x86-64 アーキテクチャのソースコードが共有されていない部分を解析し，新しく改変する必要がある．

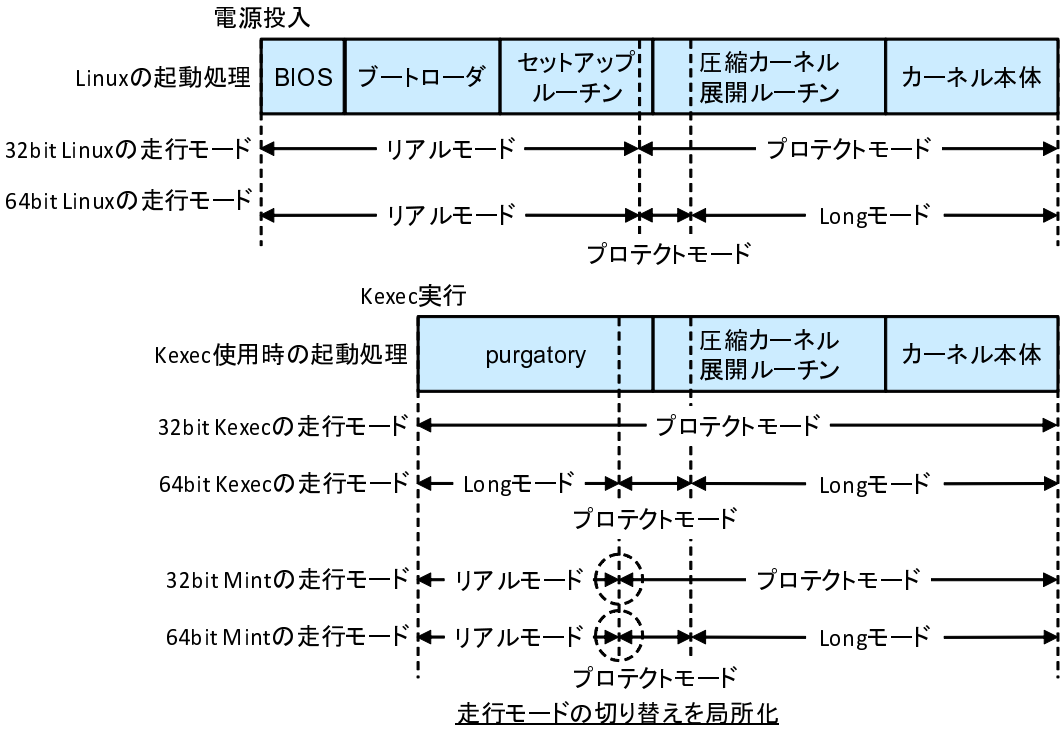

図　5.2　起動処理と CPU の走行モードの関係

5.2.6　対処

5.2.5 節で示した (課題 1) と (課題 2) を解決するため，それぞれ (対処 1) と (対処 2) を示す.

(対処 1) コア起動後にリアルモードからプロテクトモードに切り替える処理を追加

起動処理と CPU の走行モードの関係を図 5.2 に示し，以下で説明する.

通常の Linux に関して，起動直後のコアはリアルモードで走行し，BIOS とブートローダはリアルモードで走行する．そして，セットアップルーチンの後半にリアルモードからプロテクトモードに移行する．以降，32bit Linux の場合は，プロテクトモードの状態で圧縮カーネル展開ルーチンとカーネル本体を実行する．一方，64bit Linux の場合は，圧縮カーネル展開ルーチンにおいてプロテクトモードから 64bit モード (Long モード) に移行する．以降，プロテクトモード (Legacy モード) に戻ることは一般的にはない.

32bit Kexec の場合は，Kexec をプロテクトモードで実行し，プロテクトモードの状態で purgatory を実行する．一方，64bit Kexec の場合は，Kexec を Long モードで実行し，Long モードの状態で purgatory を実行する．purgatory 後の圧縮カーネル展開ルー

チンはプロテクトモードで実行する必要がある．このため，purgatory 内で Long モードからプロテクトモードに移行する．

Mint における後続 OS 起動処理は，Kexec を改変することによって自コアではなく，新しく起動した別のコアに purgatory 以降の処理を実行させる．起動直後のコアは必ずリアルモードで走行するため，続く処理を実行するためには，リアルモードからプロテクトモードに切り替える必要がある．このため，32bit Mint では purgatory にプロテクトモードに切り替える処理を追加した．一方，64bit Mint においても purgatory にプロテクトモードに切り替える処理を追加する必要がある．

以上のように，結果的には x86 アーキテクチャ対応の Mint と同様に，コア起動後にリアルモードからプロテクトモードに切り替える処理を purgatory を改変することで追加した．

(対処 2) 割り込み処理と Kexec に関連するルーチンの改変

(課題 2) であげた改変の内，(3) から (6) については，x86 アーキテクチャと x86-64 アーキテクチャのソースコードが共有されていた．このため，(3) から (6) については，改変することなく 32bit カーネルのソースコードを再コンパイルするだけで対処可能であった．一方，(1) 後続 OS の起動 と (2) コアの分割と占有 に関しては，ソースコードが共有されていない部分があったため，新しく改変が必要である．

(1) 後続 OS の起動 に関しては，64bit Kexec 関連のルーチンを新しく改変する必要があった．3.3 節で示した対処と同様に終了処理を省略し，新しく起動した別のコアに purgatory を実行させるという対処を 64bit Kexec に対して施した．

(2) コアの分割と占有 に関しては，論理 APIC ID に関連するルーチンを新しく改変する必要があった．論理 APIC ID は，IPI や I/O APIC からの割り込み発生時に割り込み先のコアを示す値として使用される値であり，OS によって設定される．したがって，論理 APIC ID は，各カーネル間で重複してはならないため，Mint では重複しないように論理 APIC ID の決定規則を変更している [4]．この論理 APIC ID に関連するルーチンが x86-64 アーキテクチャ用に新しく定義されていたため，x86 アーキテクチャ対応の Mint と同等の改変を施した．

以上の 2 つの対処により，Mint を x86-64 アーキテクチャに対応させ，64bit カーネルを複数同時に走行させることに成功した．また，32/64bit カーネルの混載として，32bit OS から 64bit OS の起動と 64bit OS から 32bit OS の起動がともに可能であることを確認した．

第 6 章

評価

6.1 評価項目

本論文で述べた起動方式について，以下の項目で評価する．

(評価 1) 起動処理に要する時間

(評価 2) コード改変量

(評価 1) として，Kexec を応用した後続 OS 起動方式を評価するために以下の 3 通りの起動処理に要する時間をそれぞれ測定し，比較する．

(1) 未改変の Linux の起動時間

(2) 未改変の Linux の Kexec による再起動時間

(3) Kexec の応用による Mint の後続 OS 起動時間

また，後続 OS 起動処理の並列化を評価するために以下の 2 通りの起動処理に要する時間をそれぞれ測定し，比較する．

(4) 単独での後続 OS 起動時間

(5) 2 つの後続 OS の並列起動時間

(評価 2) として，後続 OS を起動するために Linux カーネルに対して行った改変を行数の面から評価する．一般的なプログラム新規に作成する場合とは異なり，カーネルの改変は，改変すべき箇所を判断することが困難である．このため，改変量は少しでも減らしたいという要望が存在する．

表 6.1 kernel_init カーネルスレッド実行までの起動時間

(1) 未改変の Linux の起動時間	21.4s
(2) 未改変の Linux の Kexec による再起動時間	1.97s
(3) Kexec の応用による Mint の後続 OS 起動時間	0.309s

表 6.2 デーモン起動処理を含めた後続 OS の起動時間

(4) 単独での後続 OS 起動時間	19.7s
(5) 2 つの後続 OS の並列起動時間	21.6s

6.2 起動に要する時間

評価では，Intel Core i7(2.8GHz) を搭載した計算機を利用した．時間の測定には TSC(Time Stamp Counter) レジスタを用いた．

カーネル起動処理において，デーモン起動処理などは共通の処理である．6.1 節の (評価 1) で示した (1) から (3) では，測定の誤差を減らすため，電源の投入から kernel_init カーネルスレッド実行までを起動時間とする．このため，kernel_init カーネルスレッドを実行する直前に TSC レジスタの値を取得する．一方，(4) と (5) では，デーモン起動処理を並列に実行することによって起動を高速化するという目的が存在するため，デーモン起動処理完了までを起動時間とする．なお，測定は 10 回行い，平均値を算出した．

6.1 節の (評価 1) で示した (1) から (3) の測定結果を表 6.1 に示す．この結果の考察を以下に示す．

(1) Kexec の応用による Mint の後続 OS 起動時間は，未改変の Linux の起動時間と比較して約 21.1s(98.6%) 短縮できている．これは，BIOS，ブートローダ，およびセットアッププルーチンを走行しないためである．

(2) Kexec の応用による Mint の後続 OS 起動時間は，未改変の Linux の Kexec による再起動時間と比較して約 1.66s(84.3%) 短縮できている．これは，カーネルの終了処理を行わないためである．

また，(4) と (5) の測定結果を表 6.2 に示し，この結果の考察を以下に示す．

(3) デーモン起動処理まで含めた後続 OS 起動時間は，図 4.1 に示したのと同様に，19.7s である．各後続 OS の起動に要する時間は大差がないため，2 つの後続 OS を逐次起動

表 6.3 カーネルの改変量

評価対象	行数	ファイル数
Mint 全体	304 行	22
後続 OS 起動方式	55 行	6
起動並列化	18 行	1
64bit 対応	43 行	5

表 6.4 ブート用 AP の改変量

評価対象	行数	ファイル数
後続 OS 起動方式	130 行 (全体：10047 行)	9
起動並列化	4 行	3
64bit 対応	8 行	5

するのに要する時間は，約 39.4s となる．2 つの後続 OS を並列起動する場合，逐次起動の約 55%の時間で起動可能となる．なお，一方の OS が割り込みの初期化やデバイスの登録処理を実行している間は，もう一方の OS は排他制御機構によって処理を停止する．このため，2 つの後続 OS の並列起動時間は，単独での後続 OS 起動時間と比較して約 1.9s 程度長くなっている．

6.3　ソースコードの改変量

評価対象は，起動方式を実装した後の Mint 全体でのコード改変量，後続 OS 起動方式に関するコード改変量，起動並列化に関するコード改変量，および 64bit 対応のためのコード改変量である．この際，改変のあったファイル数も調査する．また，起動には，ブート用に AP を用いる．このブート用 AP についてもコード改変量を調査する．

カーネルの改変量の評価結果を表 6.3 に示す．また，ブート用 AP の改変量を表 6.4 に示す．これらの結果から以下のことがわかる．

(1) 表 6.3 を見ると，後続 OS 起動方式実現のための改変量は 55 行であり，Mint 全体の改変量の約 18%である．

(2) 表 6.4 を見ると，後続 OS 起動方式実現のためのブート用 AP のコード改変量は，130

行であり，元々のブート用 AP 全体の行数に対して約 1.2%である．

(3) 起動並列化と 64bit 対応は，後続 OS 起動方式実現と比較して少ない改変量で実現できている．

第 7 章

関連研究

Kexec を応用した起動方式の関連研究として，kboot[8] が存在する．kboot は，Kexec を用いたブートローダである．kboot は，まず，Kexec を扱うための最低限の機能を持ったカーネルを走行させる．そして，このカーネルから Kexec を用いて起動対象のカーネルを起動させる．これにより，汎用的な起動方式を実現できる．Kexec を再起動目的ではなく起動目的で使用する点において，本研究と類似している．しかし，本研究とは異なり，複数 OS の同時走行を目的としていないため，最初に走行するカーネルの走行環境を保護する必要はない．

起動の高速化の手法として，launchd[9] が存在する．launchd は，起動時に必要最小限のデーモンを起動し，残りのデーモンはオンデマンドで起動することにより，起動の高速化を実現している．デーモンの起動処理には多くの時間を要するため，Mint でも同様の対処を実現できれば，起動に要する時間を大幅に短縮可能である．

メニーコアの利用に関する関連研究として，Multikernel[10] が存在する．Multikernel は分散処理の考え方を取り入れ，新しく設計した OS を計算機内部で複数走行させ，ネットワークを構築する．しかし，このような新しい OS の仕組みでは，既存のソフトウェア資源を活用できないという問題がある．一方，Mint は Linux を走行させることにより，既存のソフトウェア資源を活用できる．

第 8 章

おわりに

メニーコアの効率的な利用を考慮し，Mint における高速で柔軟な起動方式に関する研究について述べた．カーネルを配置する実メモリ領域を起動時に自由に変更可能にするため，Kexec を応用した起動方式を実現した．これにより，単一のカーネルイメージから複数の OS を起動可能となった．また，多数の OS の起動に要する時間を短縮するため，起動処理を並列化した．その後，Mint の応用例として OS 間通信を利用した複数 OS の連携について説明し，連携の例として，32/64bit カーネルの混載について述べた．そして，評価により，Kexec を応用することによって後続 OS の起動処理に要する時間は，BIOS とブートローダを介する Linux の起動処理と比較して約 21.1s 短縮可能であることを示した．また，2 つの後続 OS を並列起動する場合，逐次起動の約 55%の時間で起動可能であることを示した．

謝辞

本研究を進めるにあたり，懇切丁寧なご指導をして頂きました乃村能成准教授に心より感謝の意を表します．また，数々のご助言を頂きました谷口秀夫教授，山内利宏准教授，ならびに後藤祐介助教に厚く御礼申し上げます．最後に，日頃の研究活動において，お世話になりました研究室の皆様ならびに本研究を行うにあたり，経済的，精神的な支えとなった家族に感謝いたします．

参考文献

[1] Shekhar Borkar, “Thousand core chips: a technology perspective, ” Proc. of the 44th annual Design Automation Conference, pp.746-749, 2007.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield, “Xen and the Art of Virtualization, ” Proc. of the 19th ACM Symposium on Operating Systems Principles, pp.164-177, 2003.

[3] Ole Agesen, Alex Garthwaite, Jeffrey Sheldon, and Pratap Subrahmanyam, “The Evolution of an x86 Virtual Machine Monitor, ” ACM SIGOPS Operating Systems Review Volume 44 Issue 4, pp.3-18, 2010.

[4] 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.

[5] Hariprasad Nellitheertha, “Reboot Linux faster using kexec, ” IBM, http://www.ibm.com/developerworks/linux/library/l-kexec/index.html

[6] Vivek Goyal, Neil Horman, Ken’ichi Ohmichi, Maneesh Soni, and Ankita Garg, “Kdump: Smarter, Easier, Trustier, ” 2007 Linux Symposium, Volume One, pp.167-178, 2007.

[7] 天野 正博, “Mint オペレーティングシステムにおけるヘルスチェック機能の実現, ” 岡山大学工学部情報工学科特別研究報告書, 2011.

[8] Werner Almesberger, “kboot A Boot Loader Based on Kexec, ” 2006 Linux Symposium, Volume One, pp.27-38, 2006.

[9] Brian Jepson, Ernest E. Rothman, and Rich Rosen, “Mac OS X for Unix geeks (Leopard), 4th Edition, ” O’Reilly Media, pp.69-70, 2008.

[10] 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.

発表論文

[1] 中原 大貴, 乃村 能成, 谷口 秀夫, “ Kexec によるコア別 Linux カーネルの起動方式,” 第 9 回情報科学技術フォーラム (FIT2010) 講演論文集, 第 1 分冊, pp.363-364 (2010.09).

[2] 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 (2010.11).

[3] 中原 大貴, 千崎 良太, 牛尾 裕, 片岡 哲也, 乃村 能成, 谷口 秀夫, “ Kexec を利用した Mint オペレーティングシステムの起動方式, ” 電子情報通信学会技術研究報告, vol.110, no.278, pp.35-40 (2010.11).

[4] 中原 大貴, 乃村 能成, 谷口 秀夫, “ 32/64bit カーネル混載方式の実現,” 電子情報通信学会技術研究報告, vol.111, no.255, pp.25-30 (2011.10).

[5] Yoshinari Nomura, Ryota Senzaki, Daiki Nakahara, Hiroshi Ushio, Tetsuya Kataoka, Hideo Taniguchi, “ Mint: Booting Multiple Linux Kernels on a Multicore Processor,” Proceedings of the 6th International Conference on Broadband, Wireless Computing, Communication and Applications (BWCCA2011), (2011.10).

[6] 中原 大貴, 乃村 能成, 谷口 秀夫, “ Mint オペレーティングシステムにおける起動並列化手法の検討,” 電子情報通信学会 2012 年総合大会講演論文集, (掲載予定).