特 別 研 究 報 告 書

題 目

Mint オペレーティングシステムにおける 柔軟な起動順序の実現

指導教員

谷 口 秀 夫

報 告 者

深井貴明

岡山大学工学部 情報工学科

2013 年 2 月 7 日 提出

要約

計算機に搭載される CPU のコア数や実メモリ量は増加し，計算機の性能は向上している．これらの計算機資源を効率的に利用する方法として，1 台の計算機上で複数の OS を同時に走行させる方式が研究されている．この方式の一つとして，仮想計算機方式が広く利用されている．

一方，仮想化によらず 1 台のマルチコア計算機上で複数の Linux を同時に走行させる方式の OS として，Mint (Multiple Independent operating systems with New Technology) オペレーティングシステムがある．Mint では，1 台のマルチコア計算機上で複数の OS ノードを走行できる．ここで，OS ノードは，一つの Linux OS に対応する．Mint では，複数の OS ノードを同時に独立走行させるため，CPU，実メモリ，および入出力機器を分割し，これらを各 OS ノードが仮想化によらず占有する．

Mint では，特定の OS ノードのみが新たな OS ノードを起動できる．また，一部の OS ノードは，単独で再起動できない．これらの理由から，OS ノードの起動および終了の順序は限定されている．また，OS ノードの起動順序が限定されることで発生する問題がある．この問題を解決するため，柔軟な起動順序を実現する．

本論文では，柔軟な起動順序を実現する課題について述べ，対処を明らかにした．次に，評価により，柔軟な起動順序の実現のためにソースコードを変更した量は少ないことを示した．また，柔軟な起動順序の実現により，OS ノードの起動時間は増加したものの，その増加量は 0.02 秒と小さいことを示した．

目次

1 はじめに **1**

2 Mint オペレーティングシステム **2**

2.1 設計方針 2

2.2 構成 2

2.3 OS ノードの起動順序 3

2.4 Mint の問題点 4

3 複数 OS ノードの柔軟な起動順序 **6**

3.1 目的 6

3.2 課題 6

4 他 OS ノード起動機能と単独再起動機能の共存 **9**

4.1 Kexec 9

4.2 Linux を再起動する際の実行処理 10

4.3 他 OS ノードを起動する際の実行処理 10

4.4 単独で再起動する際の実行処理 12

4.5 対処 12

4.6 実現方式 12

5 後続 OS ノード走行中の先行 OS ノードの起動 **15**

5.1 すべての OS ノードで共有するハードウェアの初期化 15

5.2 後続 OS ノード走行中に先行 OS ノードを起動する際の問題と対処 16

5.3 実現方式 16

6 評価 **17**

6.1 評価目的 17

6.2 機能実装によるコードの変更量 17
6.3 OS ノードの起動時間 18
6.3.1 評価環境 18
6.3.2 評価方法 18
6.3.3 結果 19
6.3.4 考察 20

7 おわりに **22**

謝辞 **23**

参考文献 **24**

図 目 次

2.1 Mint の構成例 3
2.2 複数 OS ノードを起動する流れ 4
2.3 Mint において問題が発生する例 5

3.1 柔軟な起動順序 7
3.2 柔軟な起動順序による問題解決の例 7

4.1 Kexec の実行処理 11
4.2 対処後の Kexec の実行処理 14

6.1 電源投入時の起動の測定 19
6.2 単独再起動時の測定 19
6.3 他 OS ノード起動の測定 20

表 目 次

6.1 評価に用いた OS および OS ノード 18
6.2 起動時間の測定結果 21

第 1 章

はじめに

計算機に搭載される CPU のコア数や実メモリ量は増加し，計算機の性能は向上している．これらの計算機資源を効率的に利用する方法として，1 台の計算機上で複数のオペレーティングシステム (以降，OS と呼ぶ) を同時に走行させる方式が研究されている．この方式の一つとして，仮想計算機方式 [1][2] が広く利用されている．

一方，仮想化によらず 1 台のマルチコア計算機上で複数の Linux を同時に走行させる方式の OS として，Mint (Multiple Independent operating systems with New Technology) オペレーティングシステム [3](以降，Mint と呼ぶ) がある．Mint は，1 台のマルチコア計算機上で複数の OS ノードを走行できる．ここで，OS ノードは，一つの Linux OS に対応する．Mint は Linux のカーネルに変更を加えることで実現されている．Mint では，複数の OS ノードを同時に独立走行させるため，CPU，実メモリ，および入出力機器を分割し，これらを各 OS ノードが仮想化によらず占有する．これにより，OS ノード間の独立性を実現している．

Mint では，特定の OS ノードのみが新たに OS ノードを起動できる．また，一部の OS ノードは，単独で再起動できない．これらの理由から，OS ノードの起動および終了の順序は限定されている．また，OS ノードの順序が限定されることで発生する問題がある．

本論文では，2 章で，Mint について説明する．3 章で，柔軟な起動順序について，実現の目的と課題を述べる．4 章と 5 章で，課題への対処を明らかにする．6 章で，対処を実現するために変更したコード量と OS ノードの起動時間の観点から，対処を評価する．7 章で，本論文のまとめを示す．

第 2 章

Mint オペレーティングシステム

2.1 設計方針

1 台のマルチコア計算機上で複数の Linux を同時に独立走行させる方式として，Mint を開発している．Mint の設計方針は，以下の二つである．

(方針 1) OS ノード間に依存関係は存在しない．
ある OS ノードの処理負荷が高くなっても，他の OS ノードの処理性能は低下しない．また，ある OS ノードが走行もしくは停止となる状態は，他の OS ノードの走行状態および起動の可否に影響しない．

(方針 2) 各 OS ノードは十分な入出力性能を利用できる．
実計算機上での走行時と同じ入出力性能で走行する．

2.2 構成

Mint では，独立走行を実現するため，CPU，実メモリ，および入出力機器を仮想化によらず分割し，各 OS ノードで占有する．

(1) CPU
コア単位で分割し，各 OS ノードは一つ以上のコアを占有する．

(2) 実メモリ
空間分割し，各 OS ノードに分割領域を分配する．

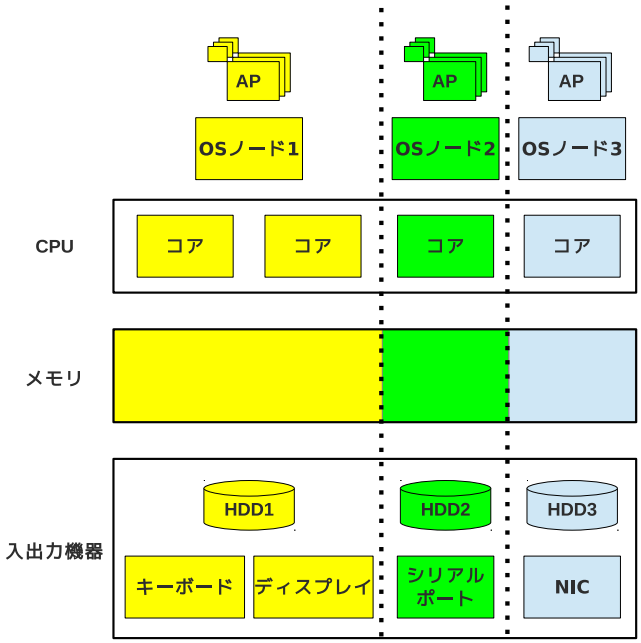

図　2.1　Mint の構成例

(3) 入出力機器
　デバイス単位で分割し，各 OS ノードは直接に占有制御する．

Mint の構成例を図 2.1に示し，以下で説明する．図 2.1では，Mint により，1 台の計算機上で OS ノード 1〜3 の 3 つの OS ノードを走行させる例を示している．この例では，CPU について，OS ノード 1 がコアを 2 つ，OS ノード 2 がコアを 1 つ，OS ノード 3 がコアを 1 つ，それぞれ占有する．実メモリについて，各 OS ノードは分割領域を占有する．また，入出力機器について，OS ノード 1 はキーボード，ディスプレイおよび HDD1 を占有する．同様に，OS ノード 2 はシリアルポートと HDD2，OS ノード 3 は NIC と HDD3 をそれぞれ占有する．

2.3　OS ノードの起動順序

Mint は，複数の OS ノードを順番に起動する．Mint において，複数の OS ノードを起動する流れを図 2.2に示し，以下で説明する．

(1) 計算機の電源投入時に OS ノードを一つ起動する．この OS ノードは静的に決定されている．以降，この OS ノードを先行 OS ノードと呼ぶ．

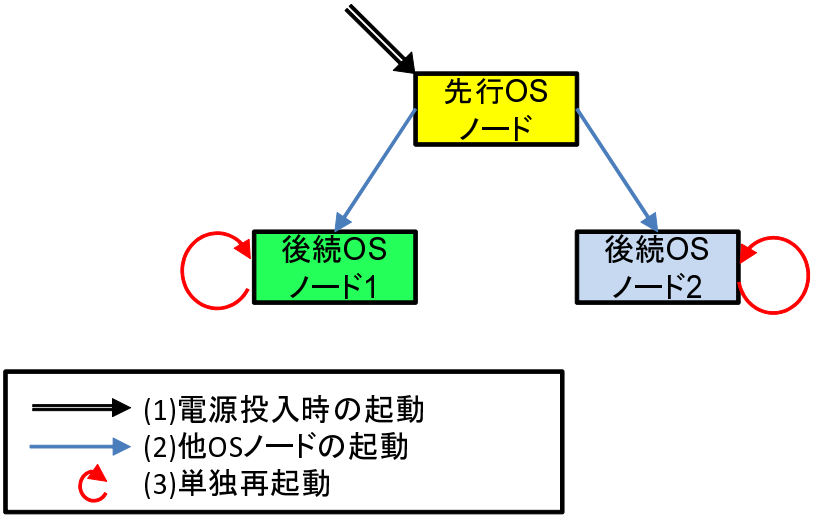

図　2.2　複数 OS ノードを起動する流れ

(2) 先行 OS ノード起動後，先行 OS ノード上で Mint 用に改変した Kexec[4] を用いて，他の OS ノードを起動する [5]．以降，先行 OS ノード以外の OS ノードを後続 OS ノードと呼ぶ．

(3) 後続 OS ノードは，通常の Kexec を用いて，単独で再起動できる．

Kexec は，Linux に標準で搭載されている高速再起動方式である．

また，すべての OS ノードは，単一のカーネルイメージを用いて起動できる．このため，OS ノードごとにカーネルイメージを用意する必要はなく，カーネルの構成と管理が容易である．

2.4　Mint の問題点

Mint には以下の問題がある．

(問題) 先行 OS ノードを停止すると，新たに OS ノードを起動できない．

Mint で問題が発生する例を図 2.3に示し，以下で説明する．まず，電源投入時に先行 OS ノードを起動し，先行 OS ノードが後続 OS ノード 1 を起動する．その後，先行 OS ノードを停止する．この場合，先行 OS ノード停止後，電源を再投入せずに，先行 OS ノードおよび後続 OS ノード 2 を起動できない．これは，後続 OS ノードを起動できるのは先行 OS ノードのみであり，先行 OS ノードは電源投入時以外に起動できないためである．計算機の電源を再投入するためには，計算機上で走行する OS ノードをすべて停止する必要がある．

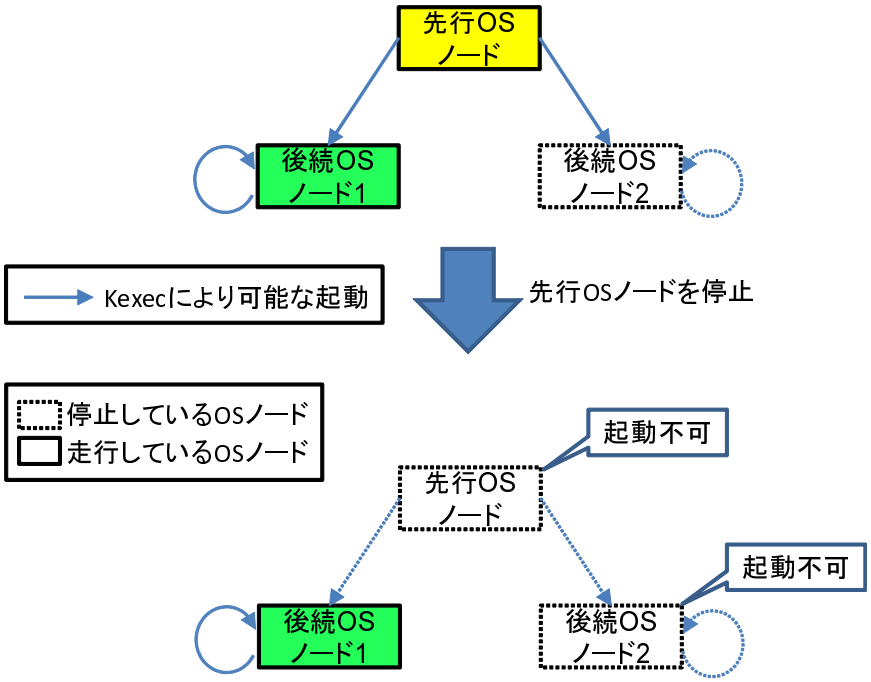

図　2.3　Mint において問題が発生する例

第 3 章

複数OSノードの柔軟な起動順序

3.1　目的

Mint において，複数 OS ノードの柔軟な起動順序を実現する．柔軟な起動順序について，図 3.1に示し，以下で説明する．柔軟な起動順序は，以下の二つの条件を満たす．

(1) すべての OS ノードは任意の OS ノードを起動できる．

(2) すべての OS ノードは単独で再起動できる．

柔軟な起動順序を実現することで，先行 OS ノードと後続 OS ノードの起動に関する機能は同じになる．このため，すべての OS ノードは任意の OS ノードを起動でき，先行 OS ノードが走行していなくても，任意の OS ノードを起動できる．故に，柔軟な起動順序を実現した Mint において，2.4節で述べた問題は発生しない．

柔軟な起動順序で問題が解決される例を図 3.2に示し，以下で説明する．図 2.3と同様に，先行 OS ノード，後続 OS ノード 1 の順に起動した後，先行 OS ノードを停止する．この場合，電源を再投入せずに，先行 OS ノードと後続 OS ノード 2 はともに後続 OS ノード 1 から起動できる．

3.2　課題

柔軟な起動順序を実現するため，以下の三つの起動を実現する必要がある．

(1) 後続 OS ノードから後続 OS ノードを起動

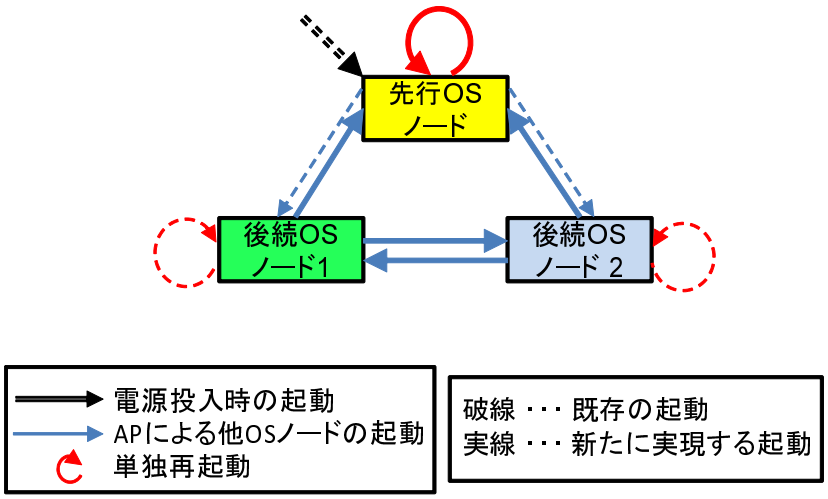

図　3.1　柔軟な起動順序

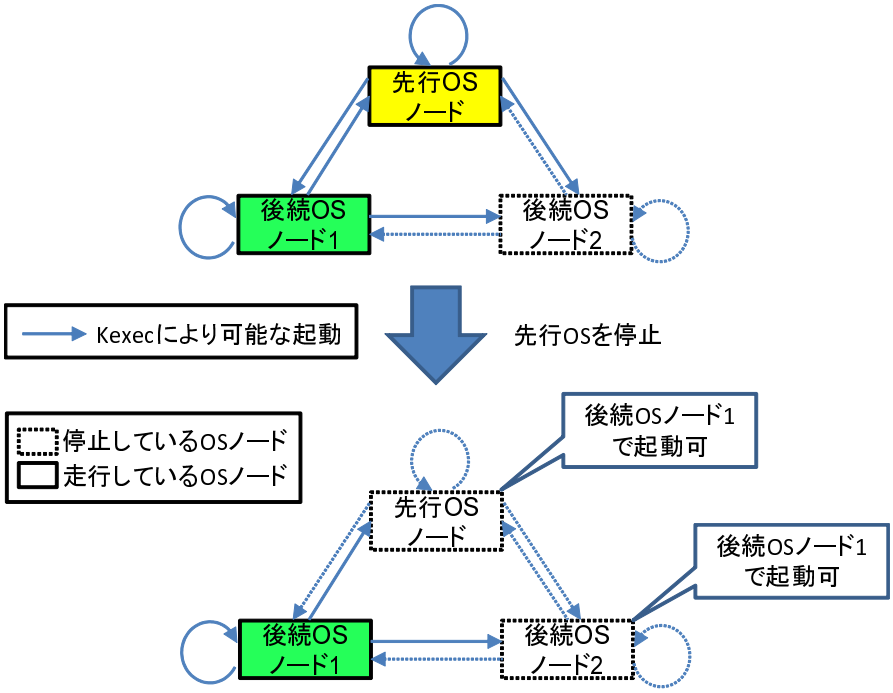

図　3.2　柔軟な起動順序による問題解決の例

(2) 後続 OS ノードから先行 OS ノードを起動

(3) 先行 OS ノード単独による再起動

これらの起動を実現するためには，以下の二つの課題がある．

(課題 1) すべての OS ノードが他 OS ノード起動機能と単独再起動機能の両方をもつ必要がある．

従来の Mint では，先行 OS ノードから他 OS ノードを起動できるが，後続 OS ノードは他 OS ノードを起動できない．また，後続 OS ノードは単独で再起動できるが，先行 OS ノードは単独で再起動できない．柔軟な起動順序では，後続 OS ノードから他 OS ノードを起動でき，先行 OS ノードは単独で再起動できる．つまり，すべての OS ノードは他 OS ノードの起動と単独再起動の両方が可能である必要がある．

(課題 2) 後続 OS ノード走行中に先行 OS ノードを起動可能にする必要がある．

従来の Mint では，先行 OS ノードは計算機の電源投入時にのみ起動する．計算機の電源投入時，後続 OS ノードは走行していない．このため，先行 OS ノードの起動処理は，後続 OS ノード走行中の起動を考慮していない．一方，柔軟な起動順序を実現した Mint では，後続 OS ノードが走行している状態で先行 OS ノードを起動または再起動できる．このため，先行 OS ノードの起動処理を変更し，後続 OS ノード走行中でも先行 OS ノードを起動可能にする必要がある．

(課題 1) に対処することで，後続 OS ノードから後続 OS ノードを起動できる．また，(課題 1) と (課題 2) の両方に対処することで，後続 OS ノードからの先行 OS ノードの起動，および後続 OS ノード走行中における先行 OS ノードの単独再起動が可能となる．

第 4 章

他OSノード起動機能と単独再起動機能の共存

4.1　Kexec

Mint では，Kexec を用いて，他 OS ノード起動と OS ノード単独再起動を実現している．Kexec では，BIOS，ブートローダ，およびセットアップルーチンの処理を省略することで，高速に再起動する．Kexec の処理は，ロード処理と実行処理で構成される．カーネルのロード処理では，カーネルイメージを読み込み，セグメントと呼ばれるデータを作成する．ロード処理で作成するセグメントは以下の 4 つである．

(1) purgatory (Kexec 固有の前処理)
電源投入時の起動処理における BIOS，ブートローダ，およびセットアップルーチンの処理のうち，Kexec で省略できない処理を代わりに行う実行コードである．

(2) セットアップルーチン
通常のカーネル起動で実行された後のセットアップルーチンと同じである．Kexec の再起動では，このセグメントは実行されずに，カーネル本体から参照される．通常のカーネル起動のセットアップルーチンで収集されるメモリマップやビデオデバイス情報といったハードウェア固有の情報を再起動前のカーネル内部より取得し，セットアップルーチンのデータ領域に再現する．

(3) カーネル本体
通常のカーネル起動で利用されるカーネル本体と同じである．

(4) initrd
通常のカーネル起動で利用される initrd と同じである．

また，実行処理では，ロード処理で作成したセグメントを実メモリに展開し，再起動の処理を行う．

以降の節では，通常の Linux を再起動する際の実行処理，他 OS ノードを起動する際の実行処理，および OS ノードを単独で再起動する際の実行処理について述べる．

4.2　Linux を再起動する際の実行処理

通常の Kexec を用いて Linux を再起動する際の実行処理について，図 4.1(a) に示し，以下で説明する．

(1) reboot システムコールの発行
reboot システムコールを発行する．Kexec の再起動は，reboot システムコールを契機に行われる．

(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 CPU の状態の初期化を行う．

(3) セグメントを実メモリへ展開
ロード処理で作成したセグメントを実メモリ上に展開する．この後，展開した purgatory にジャンプする．

(4) 走行環境の設定
(3) で展開した purgatory を走行させ，コアの走行環境を設定する．具体的には，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．この後，展開したカーネル本体へジャンプする．

(5) カーネルの起動処理
カーネル本体を起動する．

4.3　他 OS ノードを起動する際の実行処理

Mint において，他 OS ノードを起動するために改変した Kexec の実行処理について，図 4.1(b) に示し，以下で説明する．

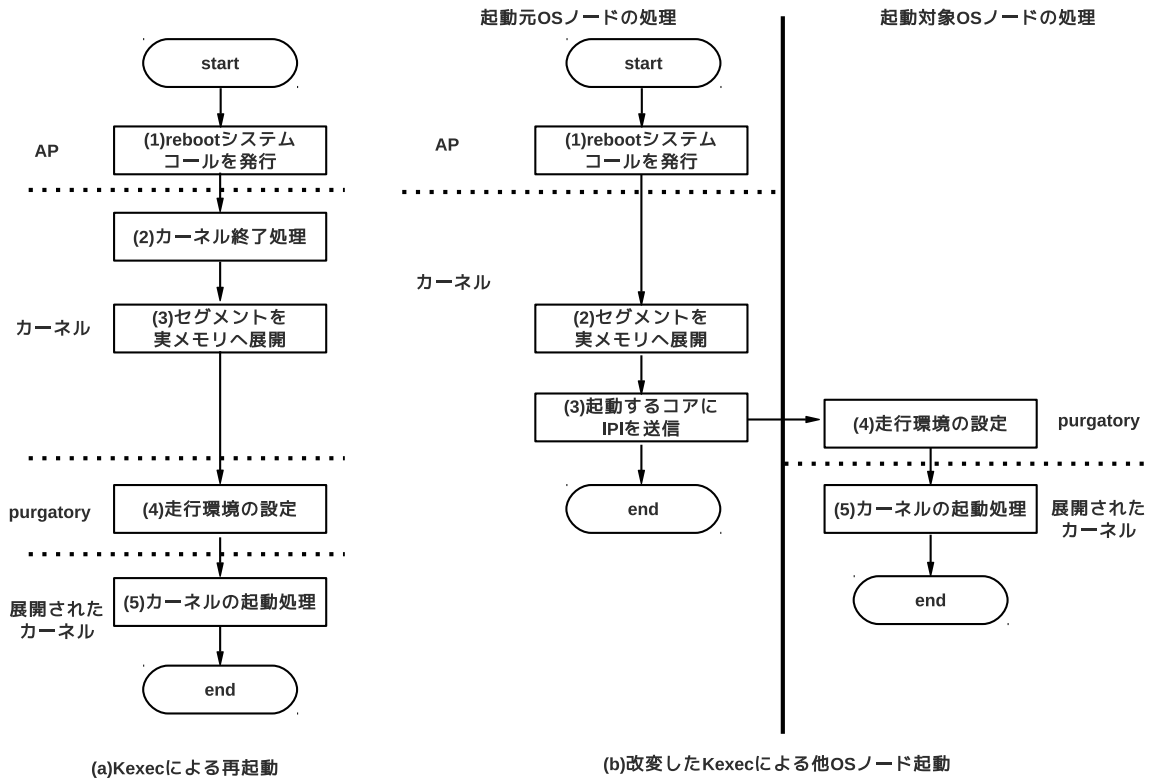

図　4.1　Kexec の実行処理

(1) reboot システムコールの発行
reboot システムコールを発行する．Kexec の再起動は，reboot システムコールを契機に行われる．

(2) セグメントを実メモリへ展開
カーネルのロード処理で作成したセグメントを実メモリ上へ展開する．

(3) 起動するコアに IPI (InterProcessor Interrupts) を送信
起動対象 OS ノードを走行させるコアに IPI を送信し，コアを起動する．IPI の送信後，起動元 OS ノードの処理は終了する．また，起動されたコアは，展開された purgatory の先頭から処理を開始する．

(4) 走行環境の設定
(2) で展開した purgatory を走行させ，コアの走行環境を設定する．具体的には，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．この後，展開したカーネル本体へジャンプする．

(5) カーネルの起動処理
　　カーネル本体を起動する．

通常の Kexec からの変更点は，以下の二つである．

(変更点 1) カーネル終了処理を省略する．

(変更点 2) 起動元 OS ノードは，セグメント展開後，purgatory に処理をジャンプせず，IPI を送信し，他コアを起動する．

4.4　単独で再起動する際の実行処理

OS ノードを単独で再起動する際の実行処理は，Linux を再起動する際の実行処理と同じである．つまり，4.2節で述べた実行処理で，OS ノードは単独で再起動する．

4.5　対処

柔軟な起動順序では，すべての OS ノードは単独再起動の機能と他 OS ノード起動の機能をもつ．後続 OS ノードがもつ単独再起動機能は，4.2節で述べた機構で実現している．また，先行 OS ノードがもつ他 OS ノード起動機能は，4.3節で述べた機構で実現している．このため，すべての OS ノードは，これら二つの機構を両方もつ必要がある．

対処として，単独再起動用の Kexec と他 OS ノード起動用の Kexec をそれぞれ用意し，すべての OS ノードにもたせる方法が考えられる．しかし，この方法では，実現のために追加するコード量が多くなることが予想される．そこで，4.2節で述べた機構と 4.3節で述べた機構を同じ Kexec に共存させる方法が考えられる．この方法では，実現のために追加するコード量は少ないと考えられる．よって，二つの機構を同じ Kexec に共存させる方法で対処する．

4.6　実現方式

4.2節で述べた機構と 4.3節で述べた機構の処理を同じ Kexec に共存させるため，他 OS ノード起動の場合と OS ノード単独再起動の場合で，Kexec の処理を分岐させる．つまり，再起動の場合は 4.2節で述べた処理を行い，他 OS ノード起動の場合は 4.3節で述べた処理を行う．対処後の Kexec の処理流れを図 4.2に示し，以下で説明する．

分岐箇所は以下の二つである．

(分岐 1) カーネル終了処理

単独再起動の場合，自身のカーネルを一度終了する必要がある．このため，カーネル終了処理を行う．一方，他 OS ノードを起動する場合，起動元 OS ノードのカーネルは，Kexec 実行後も終了せずに走行を続ける．このため，カーネル終了処理を行わない．

(分岐 2) セグメント展開後の処理

単独再起動の場合，展開した purgatory の先頭に自身の処理をジャンプする．一方，他 OS ノードを起動する場合，セグメント展開後，IPI を送信し，起動対象 OS ノードを走行させるためのコアを起動する．IPI を送信後，起動元 OS ノードは Kexec の処理を終了する．

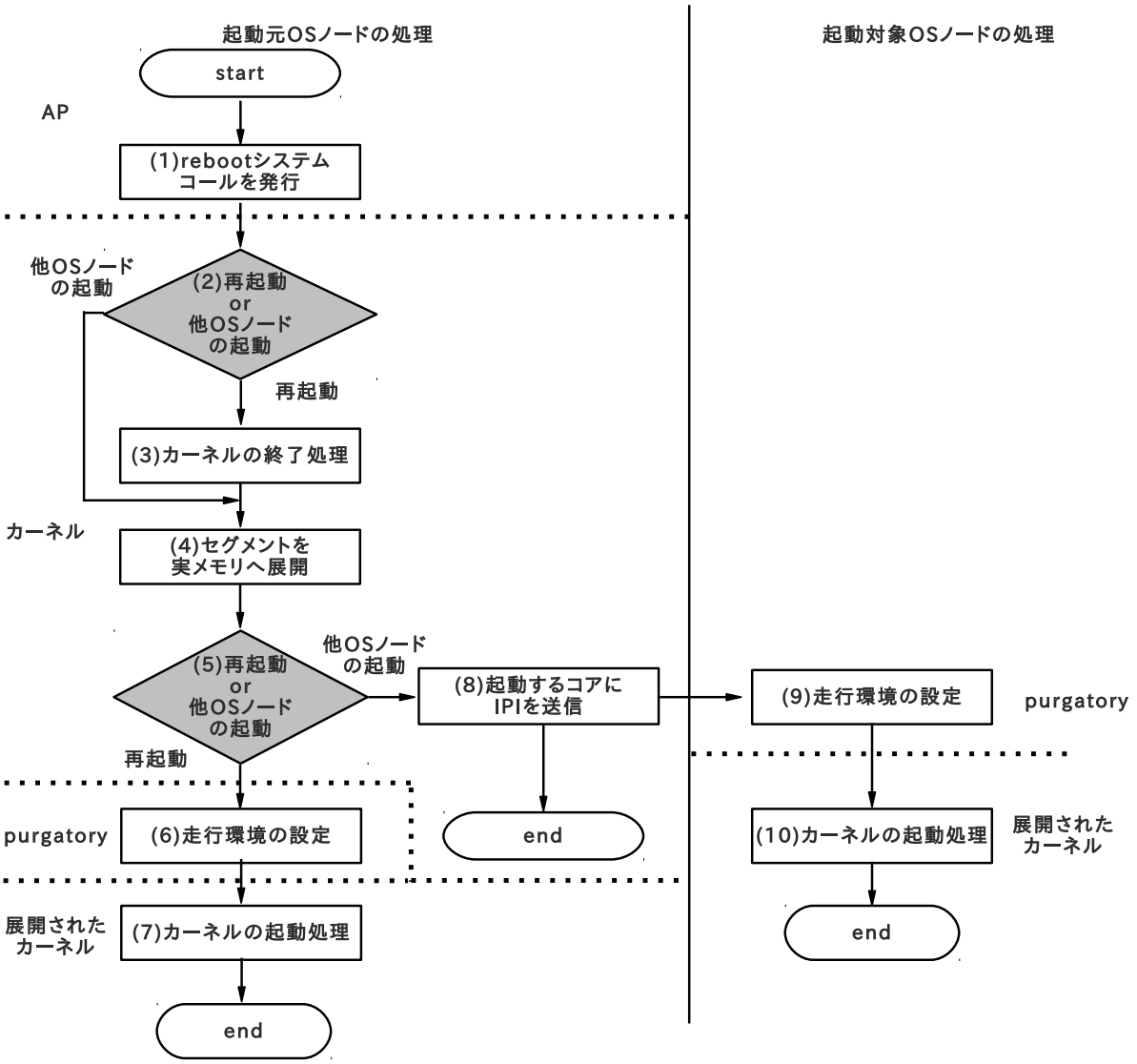

図　4.2　対処後の Kexec の実行処理

第 5 章

後続OSノード走行中の先行OSノードの起動

5.1　すべてのOSノードで共有するハードウェアの初期化

Mint で走行するすべての OS ノードは，以下の二つのハードウェアを共有する．

(1) I/O APIC
I/O APIC は割り込みコントローラである．通常，1 台の計算機は I/O APIC を一つだけ搭載する．このため，Mint で走行する OS ノードへの割り込みは，すべて一つの I/O APIC で制御する．

(2) High Precision Event Timer (HPET)
HPET はタイマである．HPET は，一つのカウンタと複数のマッチレジスタで，複数のタイマを提供する．Mint では，HPET で提供されるタイマを各コアのローカルタイマとして利用している．このため，すべての OS ノードは，一つの HPET のカウンタを共有している．

Mint では，先行 OS ノード起動時に，上記二つのハードウェアの初期化を行う．また，後続 OS ノード起動時は，これらのハードウェアの初期化処理を省略する．

5.2　後続 OS ノード走行中に先行 OS ノードを起動する際の問題と対処

後続 OS ノード走行中に 5.1節で述べた二つのハードウェアを初期化すると，以下の二つの問題が発生する．

(1) I/O APIC に登録された割り込み制御のための情報がクリアされる．

(2) HPET のメインカウンタがクリアされ，タイマ割り込みが正しく発生しなくなる．

このため，先行 OS ノードを後続 OS ノードから起動しようとすると，走行中の OS ノードの環境を破壊することになり，問題である．したがって，後続 OS ノード走行中に先行 OS ノードを起動する際は，5.1節で述べた二つのハードウェアの初期化を行わないようにする必要がある．

5.3　実現方式

実現方式では，後続 OS ノード走行中に先行 OS ノードを起動する際に，5.1節で述べた二つのハードウェアの初期化を行わないようにする．これを実現する方法として，他 OS ノード走行中に先行 OS ノードを起動するためのカーネルイメージを新たに用意する方法が考えられる．2.3節で述べたとおり，Mint では，単一のカーネルイメージを用いて複数の OS ノードを起動できる．新たにカーネルイメージを用意する方法では，他 OS ノード走行中に先行 OS ノードを起動するときのみ，異なるカーネルイメージを用いなければならない．これは，カーネルの構成と管理を複雑にするため問題である．

そこで，ブートオプションを用いる方法を考える．従来の Mint で用いているカーネルに，5.1節で述べた二つのハードウェアの初期化を行うか否かを指定するブートオプションを作成する．電源投入時は，これら二つのハードウェアの初期化を行うように指定する．一方，後続 OS ノード走行中に先行 OS ノードを起動する際は，これら二つのハードウェアの初期化を行わないように指定する．この方法を用いることで，柔軟な起動順序におけるあらゆる起動で，単一のカーネルで OS ノードを起動できる．

第 6 章

評価

6.1　評価目的

本論文では，以下の二つを評価する．

(1) 機能実装によるコードの変更量
　Mint は，Linux に変更を加えることで実現している．このため，Linux のバージョンが更新された際は，Mint を実現するための変更を新たな Linux に適用することが考えられる．このとき，ソースコードの変更量が少ないほど，Mint を実現するための変更を適用することが容易である．

(2) OS ノードの起動時間
　これまでに述べた機能の実装によるオーバヘッドを明らかにする．

以降の節では，これらの評価について述べる．

6.2　機能実装によるコードの変更量

機能実装によるコードの変更量について，以下の二つが分かった．

(1) 変更箇所について
　今回の機能実装による変更箇所は，従来の Mint で Linux に変更を加えていた箇所のみである．つまり，今回の機能実装によって，Mint 全体の変更箇所は増加していない．

表 6.1 評価に用いた OS および OS ノード

OS および OS ノード	メモリ	占有するデバイス
Vanilla Linux	192MB	ディスプレイとキーボード
先行 OS ノード (対処前)	192MB	ディスプレイとキーボード
後続 OS ノード (対処前)	192MB	シリアルポート
先行 OS ノード (対処後)	192MB	ディスプレイとキーボード
後続 OS ノード (対処後)	192MB	シリアルポート

(2) 変更行数について
機能実装によるコードの変更行数は 32 行であった．一方，Mint を実現するためのコードの変更行数は，全体で 664 行である．このため，機能実装によるコードの変更行数は，全体の 4.8%である．以上より，本機能を実装するためのコードの変更量は，十分に少ないと言える．

6.3 OS ノードの起動時間

6.3.1 評価環境

評価では，Intel Core i7 860 (2.8GHz) を搭載した計算機を利用した．評価に用いた OS および OS ノードを表 6.1に示し，以下で説明する．評価では，変更していない Linux (以降，Vanilla Linux と呼ぶ) と 4 種類の OS ノードを用いた．Mint 上の OS ノードとして，従来の Mint の OS ノード (以降，OS ノード (対処前) と呼ぶ) と，柔軟な起動順序を実現した Mint の OS ノード (以降，OS ノード (対処後) と呼ぶ) をそれぞれ用いた．すべての OS ノードは，192MB のメモリをもつ．また，Vanilla Linux と先行 OS ノードは，ディスプレイとキーボードを占有する．一方，後続 OS ノードは，シリアルポートを占有する．

6.3.2 評価方法

測定には，TSC (Time Stamp Counter) レジスタの値を用いた．このレジスタは，電源投入時からの合計クロック数を格納する．測定は 25 回行い，平均値を算出した．また，kernel_init スレッド以降は，すべての起動で同じ処理を行う．このため，測定する起動時間は，kernel_init

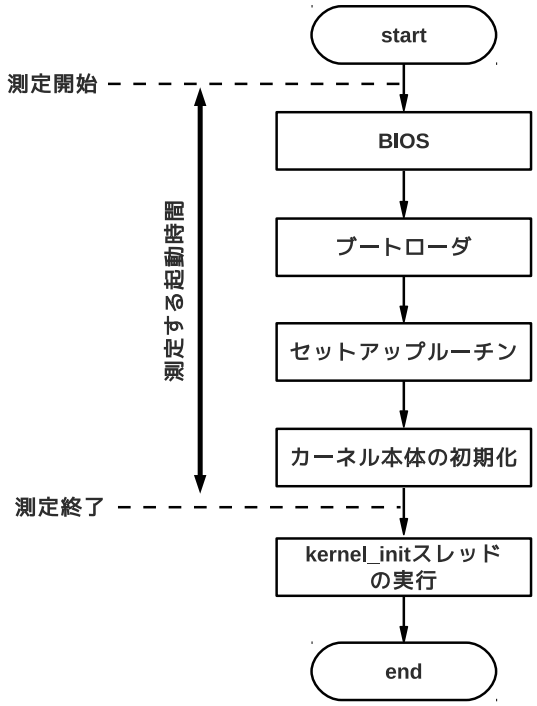

図 6.1 電源投入時の起動の測定

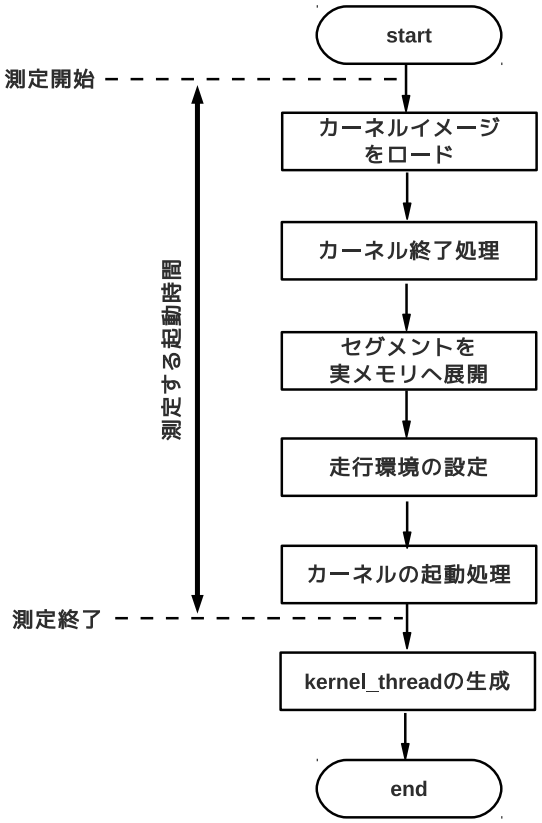

図 6.2 単独再起動時の測定

スレッドを生成する直前までとする．測定の開始時点と終了時点について，図6.1，図6.2および図6.3に示す．電源投入時の起動では，kernel_initスレッドを生成する直前のTSCレジスタの値を実行時間とする．他OSノード起動は，起動元OSノードのカーネルイメージをロードする直前を測定の開始時点とし，起動対象のOSノードのカーネルスレッドを生成する直前を測定の終了時点とする．単独再起動は，再起動するOSノードのカーネルイメージをロードする直前を測定の開始時点とし，再起動したOSノードのkernel_initスレッド実行直前までを測定の終了時点とする．

6.3.3 結果

測定結果を表6.2に示す．電源投入時の起動時間は，Vanilla Linux，対処前の先行OSノード，および対処後の先行OSノードの間で差はないことが分かる．また，他OSノード起動では，対処前のMintに比べて，対処後のMintの方が0.02秒遅いことが分かる．さらに，単

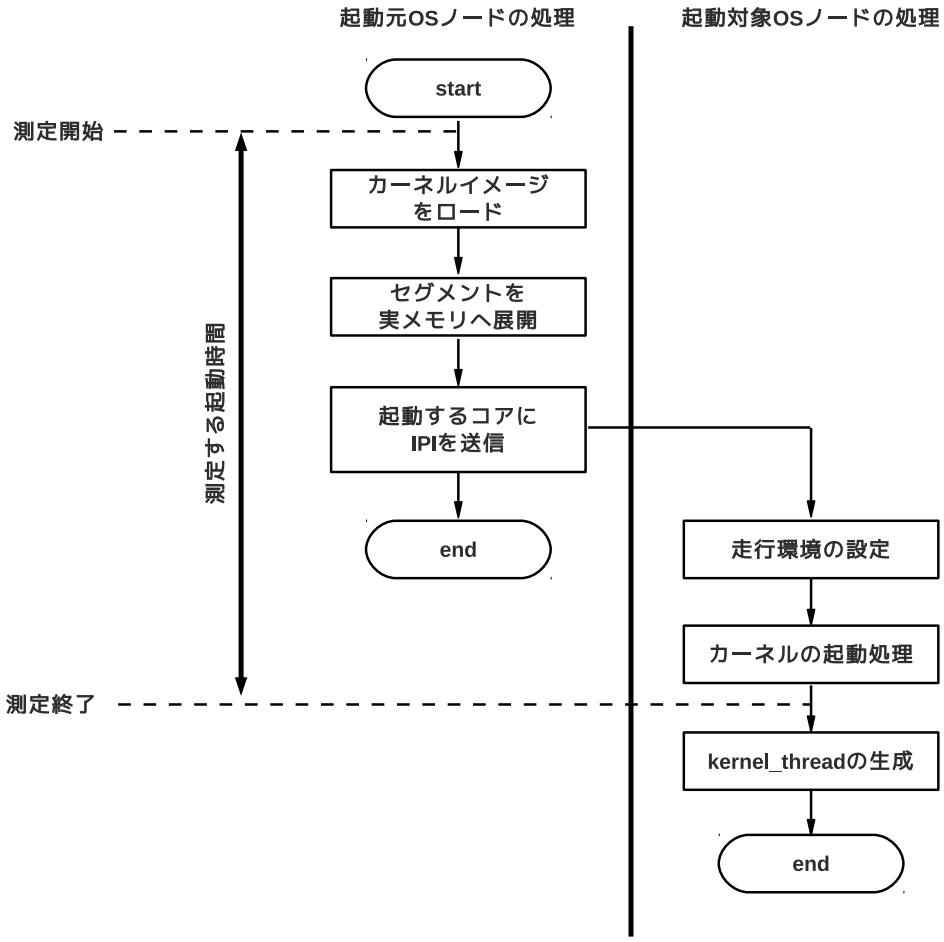

図 6.3 他 OS ノード起動の測定

独再起動では，Vanilla Linux に比べて対処後の Mint の方が 0.02 秒遅いことが分かる．

6.3.4 考察

他 OS ノード起動の処理時間について，対処前と対処後を比べると，対処後の方が 0.02 秒遅い．また，単独再起動では，Vanilla Linux に比べて対処後の Mint の方が 0.02 秒遅い．これは，Kexec に単独再起動機能と他 OS ノード起動機能を共存するための処理によるオーバヘッドが原因である．しかし，他 OS ノード起動時間の起動時間および単独再起動時間のオーバヘッドは，全体の 1%未満である．このため，機能実装によるオーバヘッドは十分に小さいと言える．

表 6.2 起動時間の測定結果

起動種別	OS ノード	時間 (秒)
電源投入時	Vanilla Linux	25.8
	先行 OS ノード (対処前)	25.8
	先行 OS ノード (対処後)	25.8
他 OS ノード起動	Mint(対処前)	4.08
	Mint(対処後)	4.10
単独再起動	Vanilla Linux	2.75
	先行 OS ノード (対処後)	2.77

第 7 章

おわりに

Mint において，柔軟な起動順序を実現するための問題点について説明し，課題を示した．また，課題への対処を明らかにした．

柔軟な起動順序の実現への課題は二つあった．一つ目は，すべての OS ノードに他 OS ノード機能と単独再起動機能の両方をもたせることである．この課題は，OS ノードの単独再起動の機能と他 OS ノードを起動する機能を共存させることで対処した．二つ目は，後続 OS ノード走行中に先行 OS ノードを起動可能にすることである．この課題は，起動処理中の一部のハードウェアに対する初期化を行うか否かをブートオプションで指定可能にすることで対処した．

さらに，上記の二つの対処を評価した．評価では，機能実現のためにソースコードに加えた変更量が少ないことを示した．また，機能実現によって OS ノードの起動時間は増加したものの，この増加量は 0.02 秒と小さいことを示した．

残された課題として，電源投入時に起動する OS ノードを選択する機能の実現がある．

謝辞

本研究を進めるにあたり，懇切丁寧なご指導をしていただきました谷口秀夫教授，ならびに乃村能成准教授に心より感謝の意を表します．また，研究活動において，様々なご指導やご助言を与えていただきました山内利宏准教授，ならびに後藤佑介助教に心から感謝申し上げます．

また，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に感謝いたします．

最後に，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] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A.: Xen and the Art of Virtualization, Proc. of the 19th ACM Symposium on Operating Systems Principles, pp.164-177 (2003).

[2] Sugerman, J., Venkitachalam, G., and Lim, B.-H.: Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor, Proc. of the General Track: 2002 USENIX Annual Technical Conference, pp.1-14, (2001).

[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] Hariprasad Nellitheertha: Reboot Linux faster using kexec, IBM (online), available from <http://www.ibm.com/developerworks/linux/library/l-kexec/index.html> (accessed 2012-02-07)

[5] 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).