Vol.2017-OS-139 No.1
2017/3/1

Mint オペレーティングシステムにおけるカーネルイメージのメモリ常駐化によるOS ノード起動の高速化

末武 悠[1]　乃村 能成[1]　谷口 秀夫[1]

概要：仮想化によらず 1 台の計算機上で複数の OS を独立走行させる Mint オペレーティングシステムが提案されている．Mint でプロセッサ数と同数の OS を起動させた場合，多コア環境では多数の OS の起動が必要となる．Mint の OS 起動処理は，Linux と同様にカーネルイメージをディスクから読み込む．このため，多数の OS を起動させる場合，読み込み処理時のディスク I/O により起動時間が長大化する．そこで，本稿では，カーネルイメージをメモリ上に常駐化させることにより，OS 起動を高速化する手法について述べる．具体的には，Mint における OS 起動方式である Kexec-Mint に提案手法を適用し，OS 起動を高速化する．

1. はじめに

近年，計算機のハードウェアは高性能化が進み，多コアプロセッサが登場している．また，メモリの容量も増加し，デバイスも多様化している．しかし，1 つの OS が全てのコアやメモリ，デバイスを効率良く利用することは困難である．

これらの計算機資源を効率良く利用する方法として，multikernel[1] がある．multikernel は，分散システムの考えに基づいた OS である．コア間共有を行わず，計算機を独立したコアのネットワークとして扱い，複数の OS を走行させる．しかし，このような OS では，Linux に代表される既存のソフトウェアを利用することができない．もう 1 つの方法として，VMware[2] や Xen[3] といった仮想計算機方式がある．仮想計算機方式は，ソフトウェアで仮想的な計算機を作成することで，1 台の計算機上で複数の OS を走行させる．しかし，仮想計算機方式は仮想化によるオーバヘッドが発生し，実計算機よりも性能が低下する．また，OS 間の処理負荷の影響が大きい．

そこで，仮想計算機方式を用いずに複数の Linux を独立に走行させる方式として，Mint（Multiple Independent operating systems with New Technology）[4] が提案されている．Mint は，各 Linux が計算機資源を直接占有して動作する．また，Mint は Linux に基づいた OS であるため，Linux 上の既存ソフトウェアを利用可能である．

Mint はプロセッサのコア数と同数の OS を走行可能なため，1 台の計算機で分散処理環境を構築可能である．Mint で分散処理環境を構築する場合，OS を頻繁に起動または終了させたいという要求がある．これは，処理の内容によって OS の構成を変化させ，処理内容に適した OS の構成にするためである．Mint において OS を起動させるには，Linux と同様にディスクからカーネルイメージを読み込む必要がある．したがって，頻繁に OS を起動させる場合，起動時のディスク I/O によって OS の起動時間が遅くなり，高速な起動ができないという問題がある．また，Mint はハードウェアを各 OS に占有させる方式であるため，1 つのコアに対して 1 つの I/O デバイスが必要となる．しかし，計算機に搭載可能なハードウェアの数には限度がある．

そこで本稿では，この問題への対処として，カーネルイメージのメモリ常駐化による OS 起動の高速化手法を提案する．本手法では，OS 起動時にカーネルイメージをディスクから読み込まない．すべての OS が読み込み可能なメモリ領域にカーネルイメージを常駐化させ，その常駐化させたカーネルイメージを利用し，OS を起動させる．これにより，ディスクからの読み込みがなくなり，OS 起動を高速化できる．

まず，既存 Mint の OS 起動方式の問題点を述べ，その問題点を解決するために提案したカーネルイメージのメモリ常駐化について，設計方針，処理流れ，および評価につ

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

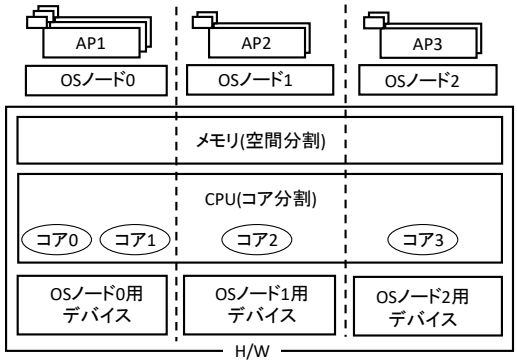

図 1 Mint の構成例

いて述べる．

2. Mint オペレーティングシステム

2.1 設計方針

Mint は仮想化方式を用いず，1 台の計算機上で複数の Linux を独立に走行させる方式である．Mint の設計方針として以下の 2 つがある．

(1) 全ての OS が相互に処理負荷の影響を与えない．
(2) 全ての OS が入出力性能を充分に利用できる．

2.2 構成

Mint では，1 台の計算機上で CPU，メモリ，およびデバイスを分割し，各 OS で占有する．Mint の構成例を**図 1** に示し，以下で説明する．本稿では，Mint を構成する OS を OS ノードと呼ぶ．Mint では，最初に起動する OS を OS ノード 0 とし，それ以降に起動する OS は起動順に OS ノード 1，OS ノード 2，... とする．

(1) CPU: コア単位で分割し，各 OS ノードは 1 つ以上のコアを占有する．
(2) メモリ: 空間分割し，各 OS ノードは分割領域を占有する．
(3) デバイス: デバイス単位で分割し，各 OS ノードが指定されたデバイスを占有する．

2.3 Mint におけるチップ内分散環境

現在，様々なプロセッサが開発されており，1,000 個のコアを有する多コアプロセッサ [5] も存在する．このようなプロセッサの開発が進むことにより，将来的には 1 台の計算機に数千個のコアを持つプロセッサが搭載されることが考えられる．Mint は，1 台の計算機上で複数の OS ノードを独立に走行可能であり，コア数と同数の OS ノードを走行可能である．このため，Mint は，いわゆるチップ内分散処理環境を計算機で構築できる．分散処理環境を Mint で実現する場合，以下の要求がある．

(要求) OS ノードの頻繁な起動と終了

ある処理を実行する際，処理の種類によって，複数のコアを占有する単一の OS に実行させたい場合と，1 つのコアを占有する複数の OS で実行させたい場合が考えられる．Mint において，処理の種類によって OS ノードの構成を変化させることが出来れば，処理を効率的に実行できる．処理の種類によって OS ノードの構成を変化させるには，OS ノードの頻繁な起動と終了が必要となる．

Mint では，OS ノードの起動に改変した Kexec (以降，Kexec-Mint) を利用し，OS を起動する度に Kexec-Mint を実行する．上記の要求を実現するには，高速な Kexec-Mint の実行が必須である．

3. Kexec-Mint を利用した OS ノードの起動

3.1 Kexec

Kexec[6] は，計算機全体を再起動することなく OS を再起動する Linux 既存の機能である．Kexec は，カーネルイメージを指定した実メモリ上に配置する機能や，BIOS やセットアップルーチンの実行によって得られるデータを再現する機能を持つ．Kexec は，カーネルイメージを指定した実メモリに配置した後，そのカーネルイメージの先頭から処理を開始することにより，OS を再起動する．これにより，Kexec は，計算機全体を再起動する必要なく，BIOS やブートローダを介さずに再起動できる．

3.2 Kexec-Mint の処理流れ

Mint では，OS ノードの起動に Kexec-Mint[7] を利用する．Kexec は，カーネルイメージを実メモリへ配置した後，そのカーネルイメージの先頭から処理を実行することにより OS を再起動する．対して Kexec-Mint は，カーネルイメージの配置後，別のコアを起動する．そのコアにカーネルイメージの処理を実行させることより，他 OS ノードを起動する．

Kexec-Mint を利用した OS ノード起動処理流れについて**図 2** に示し，以下で説明する．Kexec-Mint を実行する OS ノードを起動元 OS ノードと呼ぶ．Kexec-Mint の実行によって起動させる OS ノードを起動対象 OS ノードと呼ぶ．

(1) 起動元 OS ノード側の処理
  (A) ロード処理
    (a) データの読み込み
      起動元 OS ノードのディスクから，カーネルイメージ (bzImage)，初期 RAM ディスクイメージ (initrd.img) をバッファに読み込む．
    (b) セグメントの作成
      (1Aa) で読み込んだデータを管理するため，セグメントを作成する．bzImage はセットアップルーチンと圧縮カーネルに分割し，それぞれ別のセグメントで管理される．また，Kexec 固有の処理である purgatory のセグメ

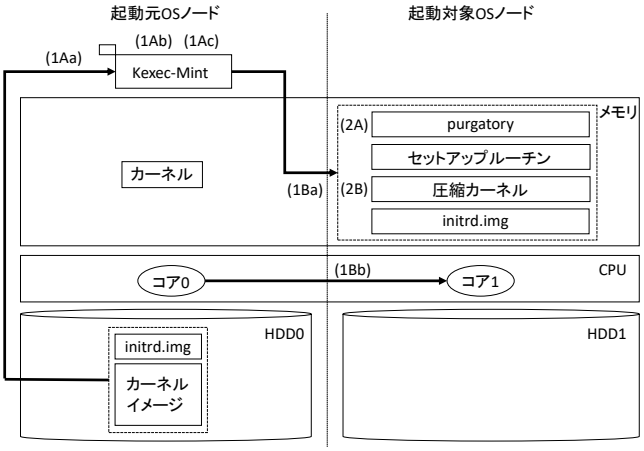

図 2 Kexec-Mint を利用した OS ノード起動処理流れ

ントも作成する．purgatory の処理については，後述する．

(c) セグメント展開マップの作成
セグメント展開マップを作成する．セグメント展開マップは配置先のページアドレスとセグメントのページアドレスから構成される．

(B) 実行処理

(a) セグメントの展開
セグメントを実メモリ上に展開する．セグメント展開マップを先頭から参照し，配置先のページとセグメントのページの内容を入れ替えることでセグメントを実メモリ上に配置する．

(b) IPI(Inter-processor interrupt) の送信
Kexec-Mint はセグメントの展開を終えた後，起動元 OS ノードのコア 0 から起動対象 OS ノードのコア 1 へ IPI を送信する．この際，IPI によって purgatory の先頭アドレスの情報を送信する．これにより，起動対象 OS ノードのコアは purgatory の処理を実行する．

(2) 起動対象 OS ノード側の処理

(A) purgatory の実行
IPI によって起動したコアは，Kexec 固有の処理である purgatory を実行する．これは通常の OS 起動時に実行される BIOS，ブートローダ，およびセットアップルーチンの代替となる．purgatory は IDT（Interrupt Descriptor Table），GDT（Global Descriptor Table），スタック領域，および各種レジスタの設定を行う．また，Kexec-Mint では，これらの処理に加え，コアの動作モードを切り替える．起動直後のコアの動作モードはリアルモードであるため，プロテクトモードに切り替える．処理を終えた後，圧縮カーネルの先頭の処理へ移る．

(B) カーネルの起動
圧縮カーネルの先頭は展開ルーチンである．展開ルーチンは圧縮カーネルを展開する．その後，展開されたカーネルがセットアップルーチンと initrd.img を参照しながら，カーネルを起動する．セットアップルーチンは，Kexec-Mint を利用しての起動では実行せず，カーネルがセットアップルーチンのデータ部を参照するのみである．

3.3 問題

Kexec-Mint を利用した OS ノードの起動には，Linux の起動と同様にディスクからのカーネルイメージの読み込みが必要である．このため，起動時間が長いという問題がある．Kexec-Mint の処理において，このカーネルイメージを読み込むためのディスク I/O が処理時間の大部分を占めている．このため，頻繁に OS ノードを起動させる場合，このディスク I/O により OS ノード起動時間が遅くなる．

4. 高速な起動手法

4.1 目的

本手法では，ディスクからのカーネルイメージの読み込みを削除し，ディスク I/O を削減する．Kexec-Mint を利用した OS ノード起動では，ディスク I/O が処理時間の大部分を占めている．このため，ディスク I/O を削減することにより OS ノード起動を高速化させる．

また，Mint にはディスクを占有せずに OS ノードを起動させたいという要求がある．これは，走行可能な OS ノード数は計算機に搭載されている I/O デバイスの数に依存するという問題に対処するためである．このような要求に対しても，提案手法は有用である．

4.2 提案手法

目的を達成するための手法として，以下の 2 つが挙げられる．

(手法 1) 走行中カーネルからのメモリイメージ複製
(手法 2) 起動前のカーネルイメージのメモリ常駐化

(手法 1) は，既に走行している OS ノードのカーネルを利用する．具体的には，Kexec-Mint を利用した起動対象 OS ノード起動時，起動元 OS ノードのカーネルを複製する．複製したカーネルを起動対象 OS ノードのカーネルとして利用する．これにより，起動対象 OS ノード起動時にカーネルイメージをディスクから読み込む必要がなくなり，ディスク I/O を削減できる．

(手法 2) は，すべての OS ノードが読み込み可能なメモリ領域 (以降，共有領域) に別途常駐化したカーネルイメージを利用する．具体的には，Kexec-Mint を利用した起動元 OS ノード起動時，共有領域の起動前のカーネルイメージを複製し，起動対象 OS ノードのカーネルとして利用す

表 1 提案手法の比較

	(手法 1) 走行中カーネルからのメモリイメージ複製	(手法 2) 起動前のカーネルイメージのメモリ常駐化
利点	カーネルの起動処理を省略可	カーネルの変更の必要無
欠点	カーネルの変更の必要有	カーネルの起動処理を省略不可

る．この手法においても，カーネルイメージをディスクから読み込む必要がなくなり，ディスク I/O を削減できる．

各手法の比較について**表 1** に示し，以下で説明する．

(手法 1) の利点は，カーネルの起動処理を省略できることである．カーネルの起動処理では，圧縮カーネルの展開処理を実行する．しかし，(手法 1) で利用するカーネルは既に展開された状態のカーネルである．このため，圧縮カーネルの展開処理を実行する必要はなく，処理を省略できる．(手法 1) の欠点は，カーネルの内容を変更する必要があることである．(手法 1) で利用するカーネルは起動前のカーネルと比較し，テキスト部とデータ部が変更されている．テキスト部の変更要因として，LKM のロードやページテーブルエントリの書き換えがある．このため，カーネルが自身の変更内容を検知し，その内容を起動対象 OS ノードの実行環境に適した内容に変更する必要がある．

(手法 2) の利点は，カーネルを変更する必要がないことである．(手法 2) で利用するカーネルは，起動前のカーネルイメージを利用するため，起動元 OS ノードのカーネルの内容が変更されても，影響がない．(手法 2) の欠点はカーネルの起動処理をすべて実行することである．(手法 2) は起動前のカーネルイメージを利用するため，(手法 1) のようにカーネルの起動処理を省略できない．

(手法 1) について，変更内容はカーネルのバージョンや Kexec-Mint の実行契機に依存する．また，その内容の変更には非常に多くの工数が必要となる．このため，ここでは (手法 2) を実現し，その効果を確認する．

4.3 設計方針

提案手法を実現する方法として，Kexec-Mint のロード処理を改変する．改変した Kexec-Mint (以降, 改 Kexec-Mint) は，1 回目の実行時にディスクから読み込んだ bzImage と initrd.img を共有領域に配置する．また，2 回目以降の実行時には共有領域の bzImage と initrd.img を利用し，OS ノードを起動させる．これにより，2 回目以降の改 Kexec-Mint 実行時の実行時間が高速化される．

1 回目の改 Kexec-Mint の実行時の実メモリ上の様子と 2 回目以降の改 Kexec-Mint の実行時の実メモリの様子を**図 3** に示し，以下で説明する．

(a) 1 回目
  (1) Kexec-Mint と同様にディスクから bzImage と initrd.img をバッファにコピーする．
  (2) (a1) と同様のデータをバッファから共有領域に配置する．
  (3) セグメント展開マップを作成し，bzImage と initrd.img を起動対象 OS ノードの領域に配置する．

(b) 2 回目以降
  (1) 共有領域に bzImage と initrd.img が配置済みであるため，ディスクからこれらのデータの読み込みは行わない．共有領域のデータをバッファにコピーする．
  (2) セグメント展開マップを作成し，bzImage と initrd.img を起動対象 OS ノードの領域に配置する．

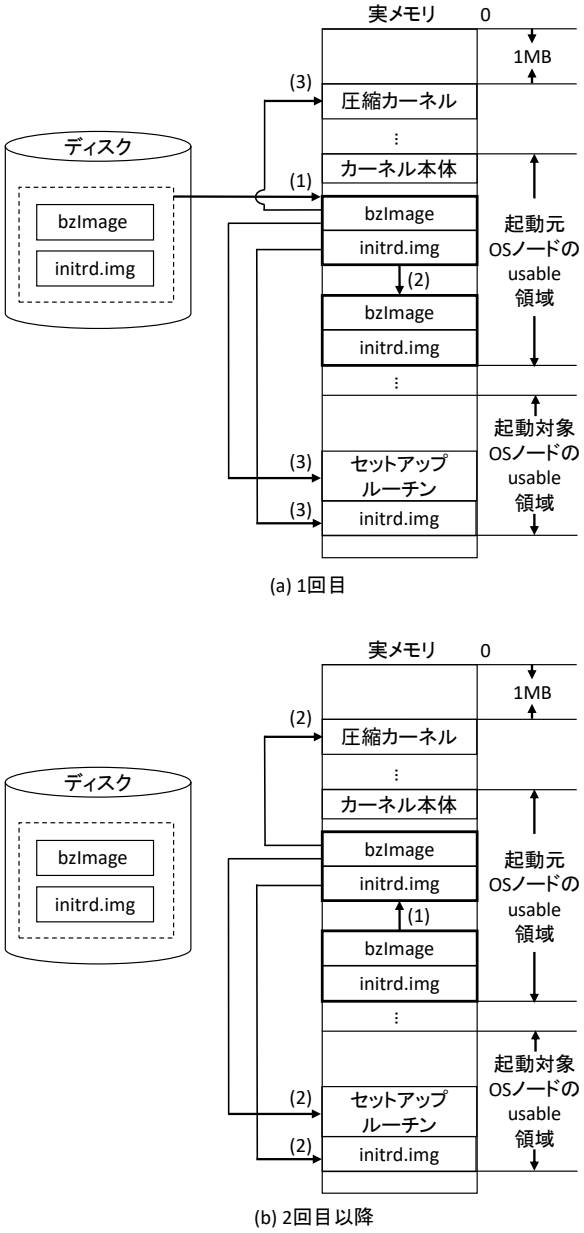

図 3 改 Kexec-Mint 実行時の実メモリ上の様子

4.4 改 Kexec-Mint の処理流れ

改 Kexec-Mint は，1 回目の実行と 2 回目以降の実行でロード処理で異なる処理を実行する．実行回数の判定には

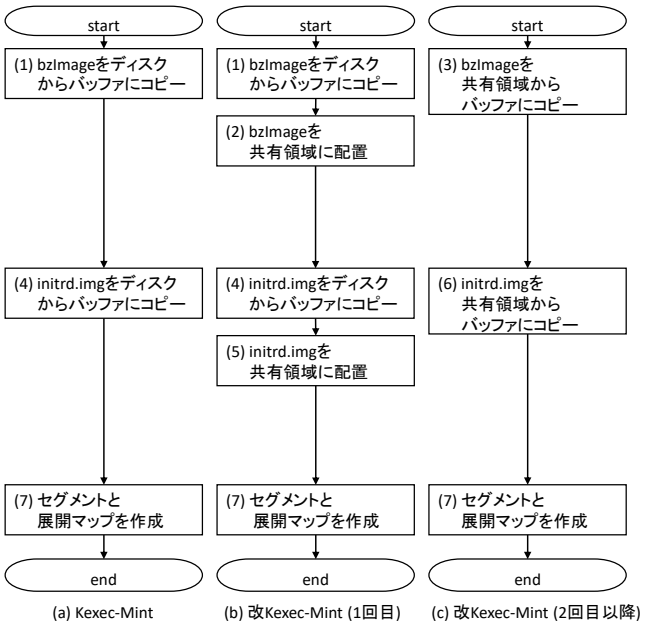

図 4 各 Kexec-Mint における ロード処理の処理流れ: (a)Kexec-Mint の処理流れ，(b) データをディスクからコピーした後，データを共有領域に配置，(c) ディスクからのコピーはせず，共有領域からデータをコピー

ロード処理の実行時に引数として渡す OS ノードの番号を利用する．番号が 1 ならば 1 回目の実行とし，番号が 2 以上であれば 2 回目以降の実行としてロード処理を実行する．

各 Kexec-Mint のロード処理流れについて**図 4** に示し，改 Kexec-Mint の改変内容について説明する．

(a) は Kexec-Mint のロード処理流れである．(b) の改 Kexec-Mint (1 回目) では，bzImage と initrd.img は共有領域に配置されていないため，共有領域に配置する処理を追加する．(c) の改 Kexec-Mint (2 回目以降) は，bzImage と initrd.img は共有領域に配置されているため，ディスクからの読み込み処理は行わず，共有領域からコピーする．

改 Kexec-Mint (1 回目) と改 Kexec-Mint (2 回目以降) のロード処理流れを以下で説明する．

(b) 改 Kexec-Mint (1 回目)
- (1) Kexec-Mint と同様に bzImage をディスクからバッファにコピーする．
- (2) (b1) でバッファにコピーした bzImage を共有領域に配置する．
- (4) Kexec-Mint と同様に initrd.img をディスクからバッファにコピーする．
- (5) (b4) でバッファにコピーした initrd.img を共有領域に配置する．
- (7) バッファにコピーしたデータを管理するためのセグメントを作成し，そのセグメントを展開するためのマップを作成する．

(c) 改 Kexec-Mint (2 回目以降)
- (3) Kexec-Mint ではディスクから bzImage をバッファにコピーしたが，共有領域からバッファにコピーする．
- (6) Kexec-Mint ではディスクから initrd.img をバッファにコピーしたが，共有領域からバッファにコピーする．
- (7) バッファにコピーしたデータを管理するためのセグメントを作成し，そのセグメントを配置するためのマップを作成する．

1 回目の場合，改 Kexec-Mint は Kexec-Mint の処理に加えて，bzImage と initrd.img を共有領域に配置する処理 (図 4 における (2) と (5)) を実行する．このため，Kexec-Mint のロード処理の処理時間よりも処理時間が長大化する．

一方，2 回目以降の場合，Kexec-Mint の処理と比較して，bzImage と initrd.img のディスクからのコピー処理 (図 4 における (1) と (4)) を実行せず，共有領域からのコピー処理 (図 4 における (3) と (6)) をするのみである．このため，Kexec-Mint のロード処理の処理時間より処理時間を短縮化できる．

5. 評価

5.1 評価項目

4 章で提案したカーネルイメージのメモリ常駐化による OS ノード起動の高速化について，以下の 2 つについて評価する．

(評価 1) Kexec-Mint の処理時間
(評価 2) コード改変量

(評価 1) として，以下の 3 通りにおける Kexec-Mint の実行時間をそれぞれ評価し，比較する．

(1) Kexec-Mint
(2) 改 Kexec-Mint (1 回目)
(3) 改 Kexec-Mint (2 回目以降)

(評価 2) として，OS ノード起動の高速化のために Kexec-Mint に対して行った改変を行数の面から評価する．

評価環境を**表 2** に示す．

表 2 評価環境

項目名	環境
OS	Mint (Linux 3.15.0 を改変)
CPU/クロック数	Intel(R) Core(TM) i7-4770 / 3.40GHz
メモリ	16.0 GB

5.2 Kexec-Mint の処理時間

提案手法では，Kexec-Mint のロード処理に改変を加え，カーネルイメージのメモリ常駐化を実現した．ロード処理より後の処理には，改変を加えていない．このため，測定箇所として，ロード処理の処理時間のみを測定する．各 Kexec-Mint におけるロード処理の処理時間の測定結果を

表 3 各 Kexec-Mint のロード処理時間

	Kexec-Mint (ミリ秒)	改 Kexec-Mint (1 回目) (ミリ秒)	改 Kexec-Mint (2 回目以降) (ミリ秒)
(1) bzImage をディスクからコピー	83.02	83.0	-
(2) bzImage を共有領域に配置	-	0.60	-
(3) bzImage を共有領域からコピー	-	-	1.45
(4) initrd.img をディスクからコピー	29.50	28.90	-
(5) initrd.img を共有領域に配置	-	0.16	-
(6) initrd.img を共有領域からコピー	-	-	0.60
(7) セグメントと展開マップを作成	33.11	33.05	33.10
その他	0.51	0.52	0.40
ロード処理全体	146.14	146.23(+0.09)	35.55(-110.59)

表 3 に示す．表 3 より，以下のことがわかる．

(1) 改 Kexec-Mint (1 回目) の処理時間は，Kexec-Mint と同等

改 Kexec-Mint (1 回目) のロード処理全体の処理時間は 146.23 ミリ秒であり，Kexec-Mint のロード処理全体の処理時間は 146.14 ミリ秒である．つまり，改 Kexec-Mint (1 回目) は Kexec-Mint と比較して，処理時間が 0.09 ミリ秒 (0.06%) 長くなっている．したがって，改 Kexec-Mint (1 回目) によって加わったオーバヘッドは，Kexec-Mint の処理時間にはほとんど影響を与えていない．

(2) 改 Kexec-Mint (2 回目以降) の処理時間は Kexec-Mint に比べて，約 1/4 倍

改 Kexec-Mint (2 回目以降) のロード処理全体の処理時間は 35.55 ミリ秒である．Kexec-Mint のロード処理全体の処理時間は 146.14 ミリ秒である．したがって，改 Kexec-Mint (2 回目以降) は Kexec-Mint と比較して，処理時間が 110.59 ミリ秒 (75.7%) 短くなっている．これは，改 Kexec-Mint (2 回目以降) は処理時間の長い bzImage と initrd.img のディスクからのコピー処理 (表 3 中における (1) と (4)) を省略し，処理時間の短い共有領域からのコピー処理 (表 3 中における (3) と (6)) を実行しているためである．

Kexec-Mint において，ロード処理以外の実行処理の処理時間 (28.98 ミリ秒) を加えると，処理全体の時間は，175.12 ミリ秒であった．つまり，本手法により，Kexec-Mint における起動元 OS ノード側の処理時間は 63.15%向上した．

また，Kexec-Mint を利用した Linux の起動全体には約 7 秒が必要である．この起動時間に対して，本稿による貢献は，約 1.5%と非常に小さい．しかし，2.3 節で述べたように，Mint では，OS ノードを頻繁に起動させることが考えられる．このような場合には，各 OS ノード起動時に必要なディスク I/O が競合し，各 OS ノードの起動がディスクからのデータの読み込み処理に左右される．そこで，本研究の成果であるディスク I/O を不要とすることにより，OS ノード起動時間がディスク I/O に左右されない起動が可能となる．

5.3 コード改変量

評価対象は改 Kexec-Mint である．Kexec は Linux 既存の機能であり，現在も開発は継続されている [8]．このため，開発によるバージョンアップに追従するためには，改変量は少量かつ局所的であることが望ましい．Kexec-Mint(行数: 24,472 行，ファイル数: 163 個) に対して，改 Kexec-Mint の改変行数は 24 行 (0.1%) であり，改変のあったファイル数は 2 個 (1.2%) である．以上のことにより，改変は少量で局所的であるといえる．

6. 関連研究

Mint のように 1 台の計算機上に複数の OS 環境を提供する技術として Docker[9][10] がある．Docker は，オープンソースソフトウェアである．`Namespace` や `Cgroups` といった Linux 既存の技術を利用し，コンテナと呼ばれるプロセス毎に隔離したユーザ空間を提供し，コンテナ毎に異なる OS 環境を実現する．コンテナの起動は OS からは単にプロセスが起動しているだけである．このため，通常のプロセスの起動と同等の速度で起動でき，非常に高速である．

しかし，Docker は以下の様な問題がある．

(1) カーネルの脆弱性を利用した攻撃をされた場合，他のコンテナに影響が出る可能性がある．Docker では，各コンテナはカーネルを共有している．このため，例えば 1 つのコンテナが攻撃を受け，カーネルの内容が書き換えられた場合，他のコンテナにも影響がでてしまう．

(2) カーネルに関わる操作はできない．上記のように，各コンテナはカーネルを共有している．このため，各コンテナ毎に異なるカーネルへの操作は不可能である．例えば，コンテナ毎に異なるモジュールをロードすることはできない．

一方，Mint では各 OS ノードはそれぞれ別のカーネルを利

用して起動する．このため，(1) のように，1 つの OS ノードのカーネルが書き換えられたとしても，他の OS ノードに影響はない．また，各 OS ノードでカーネルに異なる操作が可能であり，(2) のような問題はない．

7. おわりに

Mint オペレーティングシステムを用い，カーネルイメージのメモリ常駐化による OS ノード起動の高速化手法について述べた．具体的には，Mint の OS 起動方式である Kexec-Mint に本手法を適用し，評価を行い，OS 起動が高速化されたことを示した．

本手法は，1 回目の Kexec-Mint を利用した OS ノード起動時にカーネルイメージをメモリ上に常駐化させ，2 回目以降の起動時には，常駐化させたカーネルイメージを利用する．こうすることにより，2 回目以降の起動時には，カーネルイメージを読み込む必要がなくなり，起動が高速化される．

評価では，改 Kexec-Mint の処理時間とコード改変量について評価した．改 Kexec-Mint (2 回目以降) のロード処理は，Kexec-Mint と比較し，110.59 ミリ秒だけ処理時間が短くなり，改 Kexec-Mint の起動元 OS ノード側の処理時間は，63.15%だけ向上した．コード改変量の評価では，改 Kexec-Mint は改変前に比べ，0.1%のみの改変であり，改変量は非常に少量である．

参考文献

[1] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A. and Singhania, A.: The multikernel: a new OS architecture for scalable multicore systems, *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ACM, pp. 29–44 (2009).

[2] Adams, K. and Agesen, O.: A comparison of software and hardware techniques for x86 virtualization, *ACM Sigplan Notices*, Vol. 41, No. 11, pp. 2–13 (2006).

[3] 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, *ACM SIGOPS Operating Systems Review*, Vol. 37, No. 5, pp. 164–177 (2003).

[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] Bohnenstiehl, B., Stillmaker, A., Pimentel, J., Andreas, T., Liu, B., Tran, A., Adeagbo, E. and Baas, B.: A 5.8 pJ/Op 115 billion ops/sec, to 1.78 trillion ops/sec 32nm 1000-processor array, *VLSI Circuits (VLSI-Circuits), 2016 IEEE Symposium on*, IEEE, pp. 1–2 (2016).

[6] Hariprasad Nellitheertha: Reboot Linux faster using kexec, IBM (online), available from ⟨https://www.ibm.com/developerworks/linux/library/l-kexec.html⟩ (accessed 2017-01-20).

[7] 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).

[8] Simon Horman: index : kernel/kexec/kexec-tools.git, (online), available from ⟨http://git.kernel.org/cgit/utils/kernel/kexec/kexec-tools.git/log/⟩ (accessed 2017-01-20).

[9] Felter, W., Ferreira, A., Rajamony, R. and Rubio, J.: An updated performance comparison of virtual machines and linux containers, *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, IEEE, pp. 171–172 (2015).

[10] Joy, A. M.: Performance comparison between linux containers and virtual machines, *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in*, IEEE, pp. 342–346 (2015).