情報処理学会論文誌　Vol.60　No.12　2279–2290 (Dec. 2019)

AnT オペレーティングシステムにおける Linux 入出力操作機能の利用手法

山内 利宏[1,a)]　福島 有輝[1]　乃村 能成[1]　谷口 秀夫[1]

受付日 2019年3月11日, 採録日 2019年9月11日

概要：サービスを提供する応用プログラム（AP）の処理内容に合わせ効率的に実行するには，そのサービスに適した独自のオペレーティングシステム（OS）が有効である．しかし，独自 OS 機能をサービス提供に特化させるために，多数のドライバや高機能な OS 機能を開発する工数は大きい．このため，既存 OS のドライバ機能やファイル管理機能を利用できるようにし，利用に必要な OS の入出力操作の機能を実現する工数を最小化することが望まれる．そこで，本論文では，マルチコアプロセッサを利用して，独自 OS と既存 OS を独立に走行させ，既存 OS の入出力操作の機能を独自 OS が利用する手法の実現方式について述べる．具体的には，独自 OS でありマイクロカーネル構造を持つ ***AnT*** オペレーティングシステム（***AnT***）と既存 OS の Linux を共存して走行させ，***AnT*** 上の AP から Linux の入出力操作の機能を利用する手法を述べる．また，提案手法の実現方式と処理構造を述べ，工数と Linux 入出力操作の機能利用の性能を明らかにする．

キーワード：ファイル入出力操作，システムコール代行実行，ドライバ，オペレーティングシステム，マルチコアプロセッサ

Method for Delegating I/O Functions to Linux in *AnT* Operating System

Toshihiro Yamauchi[1,a)]　Yuuki Fukushima[1]　Yoshinari Nomura[1]　Hideo Taniguchi[1]

Received: March 11, 2019, Accepted: September 11, 2019

Abstract: A specialized operating system (OS) can provide some order-made service efficiently. However, development of the specialized OS takes a large amount of cost (e.g., device driver, file management functions). Thus, it is important to reduce the cost. In this paper, we describe a method for introducing the specialized OS and Linux run independently in multicore processors, and the specialized OS uses Linux file I/O functions. This paper describes the design and the implementation of the proposed method for ***AnT*** operating system as the specialized OS, and reports the evaluation results of the proposed method.

Keywords: file I/O operation, system call delegation, driver, operating system, multicore processors

1. はじめに

サービスを提供する応用プログラム（以降，AP）の処理内容に合わせ効率的に実行するための OS の選択肢として，既存 OS を利用する方法と新規に独自のオペレーティングシステム（以降，OS）を開発する方法がある．一方で，様々なデバイスの登場とともに，これらを制御するプログラム（以降，ドライバ）の種類は増加し，各 OS では，多数のドライバが必要となっている．新規 OS 開発の場合，デバイスドライバ開発または移植の工数は小さくないため，特に OS 開発者の数が限られる場合，工数の削減が重要である．

ドライバ開発工数を削減する研究として，ドライバを 3 階層で抽象化し，これらの階層の組合せでドライバを生成

[1] 岡山大学大学院自然科学研究科
Graduate School of Natural Science and Technology, Okayama University, Okayama 700–8530, Japan
[a)] yamauchi@cs.okayama-u.ac.jp

© 2019 Information Processing Society of Japan

する手法 [1] が提案されている．また，ドライバの開発に特化したプログラミング言語を設計し，設計した言語を利用してドライバを生成する手法 [2]～[4] も提案されている．しかし，これらの手法は，デバイスを制御するためのノウハウが必要であり，移植に比べ，その工数は小さくない．また，既存 OS のドライバを移植する研究として，Linux の LKM 形式ドライバのプロセス化手法 [5] がある．この手法は，LKM 形式ドライバをソースコード不要でそのまま流用可能である．ただし，独自 OS と Linux のカーネル機能の変換処理を作成する必要があり，独自 OS と Linux に熟知している必要がある．したがって，より工数を削減できる手法が必要になっている．さらに，多数のドライバに加え，複雑かつ高機能な OS 機能として，ファイル管理機能があり，この開発工数も大きい．

そこで，本研究では，独自 OS の開発において，必要なドライバや入出力操作の機能の開発工数を削減することを目指す．これを実現するために，本論文では，マルチコアプロセッサを利用して，独自 OS と既存 OS を独立に走行させ，既存 OS のドライバ機能とファイル管理機能を利用する入出力操作の機能を独自 OS が利用する手法の実現方式について述べる．具体的には，独自 OS でありマイクロカーネル構造を持つ ***AnT*** OS（以降，***AnT***）[6]，[7] と既存 OS の Linux を Mint OS [8] を用いて，共存して走行させる．また，***AnT*** 上の AP から Linux の入出力操作の機能を利用する手法の設計について述べる．さらに，これらの実現方式と処理構造を述べ，工数と Linux 入出力操作の機能利用の性能を明らかにする．

提案手法を実現することで，1 つのコアを占有して，既存 OS を起動するため，独自 OS が利用できるコアが 1 つ減り，またメモリも空間分割となるデメリットがある．しかし，新規 OS 開発では，限られた時間と人的リソースのなかで OS 開発を行うため，提案手法を 1 度適用することで，対応できるデバイスドライバやファイルシステムが増えることは，非常に有用である．

提案手法と同様に，Linux に処理を依頼する独自 OS の例として，McKernel [9] がある．McKernel は，HPC アプリケーション向けに開発されており，Linux にシステムコール処理をオフロードする機能がある．一方，提案手法は次の 3 つの点で異なる手法である．1 つは，入出力に関する処理のみを Linux に依頼する点で異なる．2 つ目は，独自 OS と Linux 間での連携方式 [10] において，両 OS でプロセスを同期させて存在させるか否かという点で異なる．さらに，連携方式において，引数の受け渡しを行うためのメモリ共有方式で異なる．具体的には，提案手法では，引数，およびポインタ引数の参照先を共有メモリに格納し，物理アドレスを Linux に通知し，Linux 側でマッピングして共有する．このため，独自 OS と Linux の仮想記憶空間の構成が異なっていた場合にも適用できる．一方で，McKernel は，ポインタ引数の場合でも，仮想アドレスをそのまま Linux に通知して，アクセス時にページ例外でメモリをマッピングする．McKernel では，ポインタ引数をそのまま依頼できる利点があるものの，仮想アドレスを意識するため，独自 OS の仮想記憶空間の構成を Linux に合わせる必要がある．

2. *AnT* と Mint

2.1 *AnT* オペレーティングシステム

マイクロカーネル OS である ***AnT*** は，マルチコアプロセッサに対応しており，m-カーネルと p-カーネルの 2 種のカーネルを用意している [7]．m-カーネルはメモリ管理機能も含めマイクロカーネルに必要な全機能（ファイル管理機能とドライバ管理機能を除く）を有し，一方，p-カーネルは，例外・割り込み管理機能，サーバプログラム間通信機能，およびスケジュール機能のみを保有することで軽量化を図っている．

2.2 Mint

Mint は，仮想化によらずマルチコアプロセッサを搭載した計算機上で複数の Linux を独立に走行させる OS である．たとえば，4 コアの計算機上において，コア 0 上で 1 つ目の Linux，コア 1～3 上で 2 つ目の Linux を独立に走行させることができる．Mint は，Linux ベースであり，開発工数が小さい．なお，実メモリは空間分割で割り当てられ，デバイスは各 OS がデバイス単位で占有する．また，Mint は他 OS 起動機能を有しており，専用のシステムコールなしに，他の Linux の起動を実現している．

3. Linux 入出力操作機能の利用手法

3.1 方針

マルチコアプロセッサを利用して，独自 OS が既存 OS の入出力操作機能を利用可能にする手法の設計方針について述べる．

（方針 1）独自 OS と既存 OS は別コアで走行させる．これにより，OS 間の相互影響を抑制する．

（方針 2）入出力操作の処理オーバヘッドを小さくする．実入出力を含むからといっても，その処理は高速であることが望ましく，特にプロセッサ負荷を小さくして他サービスへの影響を抑制することが重要である．

（方針 3）開発工数を抑制する．本手法が有効であるためには，既存 OS のプログラムを移植すること等に比べ，工数が小さいことが重要である．

（方針 1）により，既存 OS を別コアで実行しても，既存 OS 側の処理によって，バスや CPU のキャッシュの共有による性能への影響がある．提案手法でこの影響を許容するためには，既存 OS 側では基本的に代行処理以外の AP を走行させないことが要件となる．また，既存 OS が入出

© 2019 Information Processing Society of Japan

力操作のために1つのコアを占有するため，CPUがボトルネックとなるAPの場合にはデメリットが大きい．一方で，提案手法には，1つのコアを入出力操作で占有させたとしても，それにより，利用できるデバイスやファイルシステムの選択肢を増やすことができる利点がある．このため，入出力操作の選択肢を増やしたい場合には，提案手法は有用である．

以降では，3.2節において両OSが共存して走行する手法，3.3節において独自OS上のAPがLinuxの入出力システムコールを利用できる手法について述べる．

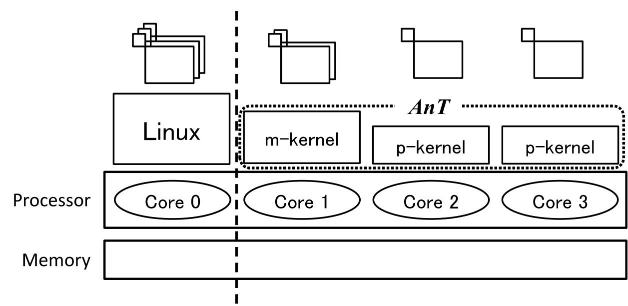


図1　独自OS（***AnT***）とLinuxの共存走行の構成

Fig. 1　Overview of coexistence of specialized OS (***AnT***) and Linux.

3.2　共存走行法

3.2.1　設計

異なるコアでOSを共存走行させる方式として，仮想マシンモニタを用いる方式と，ハードウェアをOS間で分割して利用する方式がある．本研究では，独自OSを実計算機に近い性能で動作させたいことから，複数のLinuxを共存走行できるMintを基盤とし，後者の方式を採用する．

また，共存走行する複数のLinuxの内1つを独自OSとする．共存走行の構成を図**1**に示す．Mintから***AnT***を起動するために，***AnT***のカーネルをLinuxカーネルと同じELF形式（bzImage）で作成する．これにより，Linuxと同様に***AnT***を専用システムコールなしに起動できるようにした．各OSの起動順は，Linux，***AnT***とし，起動処理に関する改造を***AnT***のみ局所化した．これにより，（方針1）と（方針3）を満足できる．

なお，本方式を実現する共存走行環境への要件として，コアを排他的に分割，メモリを空間分割，および各デバイスを排他的に専有できることがある．また，既存OSと独自OS間でプロセッサ間割り込み（以降，IPI：Inter-Processor Interrupt）の送受信が可能であることも要件となる．上記条件に合致すれば，Mint以外でも利用できる．

3.2.2　実現方式

共存走行の実現には，以下の2つの課題がある．

（課題1）　ハードウェア資源の分割方式
独自OSは，Linuxによって後から起動されるため，独自OSの初期化において，Linuxが使用しているコア，メモリ，およびデバイスの環境を破壊してはならない．

（課題2）　起動方式
開発工数抑制のため，***AnT***のみの改造でMINTの2番目のOS起動方式に対応させる必要がある．
各課題の対処について，以降に説明する．

（課題1）について，まずコアを分割するために，Linuxの使用コア数を起動オプションで指定し，LinuxがBoot Strap Processor（BSP）のみを占有するように起動する．Mintは，Linux起動後，Kexec機能によって，Linuxの未使用コアで***AnT***カーネルを実行させる．Mintでは，実行中のカーネルを終了せずに未使用のコアで新しいカーネルを起動するためにKexecにいくつか改変している[11]．主な改変内容は，IPIを送信してCPUを起動する処理の追加，CPUモードの移行処理，および起動するカーネルの利用可能な物理メモリ領域の指定機能である．具体的には，Mintの上記の既存機能を利用して，Linuxから***AnT***を起動させる2番目のコアにIPIを送信し，***AnT***の起動処理部分から走行させる．したがって，***AnT***起動の際，すべてのコアを初期化するのではなく，LinuxのBSP以外を初期化するように***AnT***を改変する．

Mintでは，BIOSコールの結果を改変し，2番目以降に起動するOSが利用できるメモリ域を設ける．MintのKexec機能は，2番目以降のOS起動時に，このメモリ域を指定する．***AnT***では，この利用できるメモリ域で起動するように，コンパイル時に指定する．

Mintにおけるデバイス分割は，PCIバスのスキャン処理を改変することによって実現されている．具体的には，ハードウェア初期化の際に特定のPCIデバイスIDを持つハードウェアのみを捕捉し，他を無視する．Mintでは，もともと設定ファイルを記述することにより，特定のPCIデバイスのみを捕捉する（初期化する）ように指定可能であることから，***AnT***にも同等の仕組みを導入し，デバイス利用の競合を防ぐ．この際，***AnT***が占有するデバイスの割り込み通知を自身が占有するコアに向けるようにする．

（課題2）についての対処について述べる．Mintは2番目のOS起動にKexec機能を利用しており，Kexec機能は，LinuxのbzImageの元となるvmlinux（ELF形式）と同じ形式のプログラムを起動する．

そこで，***AnT***にラベルを追加し，***AnT***をLinuxのbzImageと認識させ，MINTから2番目のOSとして起動可能にする．

AnTのbzImage作成手順を図**2**に示し，以下で説明する．

(1) ***AnT***でbzImage作成に必要なラベルを追加
(2) ***AnT***のソースコードから“ant”（ELF形式）をビルド
(3) “ant”（ELF形式）を“vmlinux”にリネーム
(4) カーネルの展開先アドレスを変更

© 2019 Information Processing Society of Japan

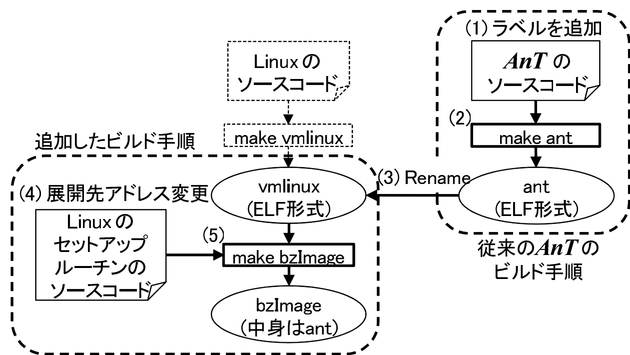


図 2 ***AnT*** の bzImage 作成手順
Fig. 2 Procedure for creating bzImage of ***AnT***.

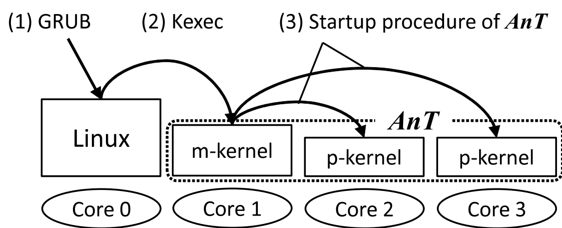


図 3 各 OS の起動順序
Fig. 3 Startup order of each OS.

(5) “vmlinux”(中身は ant)と展開先アドレスを変更したセットアップルーチンのソースコードをもとに bzImage を作成

上記により,セットアップルーチンのソースコードを独自 OS に移植しなくてよいため,改造工数は小さい.

3.2.3 起動の処理流れ

共存走行のための起動の処理流れを図 **3** に示し,以下で説明する.

(1) 電源投入後に GRUB により,コア 0 で Linux を起動
(2) Kexec により,コア 1 で m-カーネル(***AnT***)を起動
(3) ***AnT*** 固有の起動処理により,m-カーネル(***AnT***)がコア 2 とコア 3 でそれぞれ p-カーネル(***AnT***)を起動

3.3 Linux 入出力システムコール利用法

3.3.1 設計

多数のドライバやファイル管理機能の利用を実現するために必須なシステムコールとして,open/close/read/write/stat/fstat/lseek システムコールの 7 つがあげられる.本方式では,これらのシステムコールを利用可能とすることを目標とする.

独自 OS 上の AP が Linux の入出力システムコールを利用する処理の流れについて述べる.まず,独自 OS 上の AP プロセスが Linux システムコールを発行し,その内容を独自 OS から Linux に転送する.次に,Linux 上で当該システムコールを代行実行し,実行結果を Linux から独自 OS に返送し,結果を AP プロセスに返却する.この処理における設計を述べる.

両 OS 間でのシステムコール実行に関するデータ授受では,両 OS 間で実メモリを共有させ,複写レスでのデータ授受する方式,および実メモリを共有せず複写する方式がある.実メモリ共有の場合,両 OS で実メモリを共有する機構の検討が必要であるものの,複写レスで高速にデータ授受できる.このため,実メモリ共有方式を採用する.具体的には,データ授受を行う領域の実アドレスを送信元 OS から送信先 OS に通知することで共有を実現する.

また,Linux 上での代行実行は,プロセス,もしくはスレッドで行う方式が考えられる.プロセスは,不具合が他のプロセスに影響を与えず,***AnT*** の実行主体であるプロセスと対応を取りやすい利点がある.一方,スレッドはプロセスに比べ,生成処理が高速である利点がある.***AnT*** の実行主体との対応を優先し,Linux 上の代行実行はプロセス(以降,代行プロセス)で行う.なお,代行プロセスを複数存在させることで,代行実行の多重化を可能にする.

次に,代行プロセスの生成方式として,***AnT*** 上のプロセスと Linux 上のプロセスが 1 対 1 で対応するように,***AnT*** のプロセス生成時に Linux 上で代行プロセスを生成する方法,および ***AnT*** からの処理依頼時に代行プロセスを生成する方法がある.前者の方式は後者の方式に比べ,処理の実行が高速である利点,および両 OS 間でプロセスを同期して存在させるために,***AnT*** でのプロセスの生成と消滅に合わせて Linux で対応するプロセスの生成と消滅を行う必要があるという欠点がある.OS 間のプロセスの生成と消滅時の相互の影響を抑制するため,後者の処理依頼時に代行プロセスを生成する方法を採用する.これにより,Linux の入出力システムコール実行の度に代行プロセスの生成/実行/消滅の処理を行うものの,システムコール実行の度にプロセスの生成と消滅を行うため,オーバヘッドが大きい.

このオーバヘッド増加に対処するため,相互に関係するシステムコール(例:open/read/close システムコール)は,連続して実行されることに着目し,この場合の Linux の代行実行処理は,システムコールの度にプロセスを消滅させず,生成/実行/$\cdots$/実行/消滅とする.ここで,実行を 2 回以上機能を継続機能と呼ぶ.上記により,オーバヘッドを削減でき,他サービスへの影響を抑制(方針 2)できる.

なお,代行プロセスを消滅させるか継続させるかの効果は,プロセス生成消滅処理のオーバヘッドと継続時のメモリ使用量のトレードオフになる.この点については,評価(4.3.1 項)で述べる.また,提案手法では,代行プロセスを生成する手法を利用しており,かつ入出力操作を Linux カーネルに任せているため,提案手法における排他制御は不要である.

3.3.2 実現方式

Linux 入出力システムコール利用の実現には,以下の 3 つの課題がある.

© 2019 Information Processing Society of Japan

(課題 1) 両 OS 間での実メモリ共有

両 OS で実メモリをどのように確保と解放を行い，共有するのか検討する必要がある．

(課題 2) 継続機能

代行プロセスを管理し，代行プロセスのシステムコールの実行回数を制御する必要がある．

(課題 3) 実現するプログラムの局所化

Linux 側のプログラムの修正や追加カ所を局所化し，開発工数を抑制する必要がある．

（課題 1）について，データ授受する領域の利用は独自 OS から始まり独自 OS で終わるため，領域の確保/解放は独自 OS で行う．つまり，共有する実メモリの管理は独自 OS が行う．Linux は，授受した実アドレスを適当な仮想アドレス位置にマップする．これにより，両 OS 間で実メモリを共有する．なお，授受するデータサイズがページサイズを超える場合，独自 OS 側で実メモリ連続の領域を確保する．

図 **4** に Linux と ***AnT*** 間のデータ授受に関係するメモリマッピングを示す．***AnT*** は，実メモリ連続を保証しているコア間通信データ域（以降，ICA：Inter-core Communication Area）[6] を利用して領域を確保する．***AnT*** における ICA の実アドレスは仮想アドレスと 1 対 1 で対応しており，ICA の領域はマップにより確保され，アンマップにより削除される．ICA に確保した両 OS の共有メモリ域を LCA（Linux system Call information Area）と呼ぶ．また，LCA の実アドレスを管理するために，LCA の実アドレスを格納した領域（以降，LCA 管理表（LCA management table））をあらかじめ，両 OS で共有する実メモリ領域に確保する．LCA 管理表には，LCA の実アドレスを格納する領域を依頼用と結果用で個別にコア数分，用意してあり，生成した LCA 用の ICA の実アドレスを登録する．

これに加え，***AnT*** から Linux に IPI を送信する際には，自コアに対応するベクタ番号で IPI を送信する．これにより，IPI を受け取った Linux 側では，ベクタ番号により，IPI の送信元コアを特定できる．この機構とコアごとに LCA を用意することにより，依頼と結果返却を複数のコアから同時に行うことを実現できる．

LCA には，以下の 5 つの情報を格納する．

(1) cid：***AnT*** 上の AP プロセスが動作しているコアの番号．結果を返却する際の IPI で使用
(2) syscallno：Linux システムコール番号
(3) retval：Linux システムコールを実行した際の戻り値
(4) cflag：継続機能の ON/OFF を示すフラグ
(5) args：Linux システムコールの引数

（課題 2）について，Linux 上に代行プロセスを管理するプロセス（以降，親プロセス）を用意し，代行実行の前に親プロセスを経由させることで対処する．親プロセスは，代行プロセスと LCA の対応を保存しておく管理表（以降，代行プロセス管理表）を有し，複数の代行プロセスを管理する．独自 OS から LCA の cflag を設定することにより，Linux で継続機能を制御する．たとえば，getpid/stat/open システムコールの順にシステムコール利用が発生する場合，getpid と stat システムコール要求時に継続を示すフラグを ON にしておくことで，後続の stat と open システムコール利用の際に，代行プロセスの生成処理を削減できる．

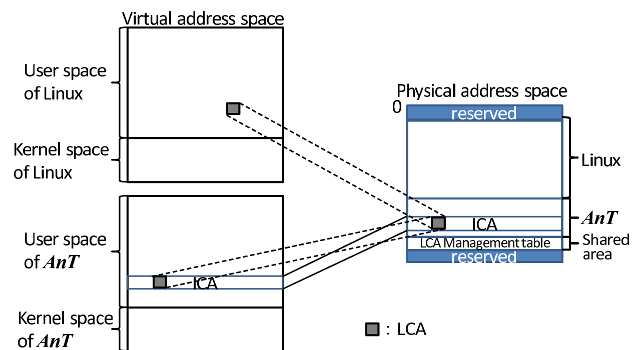


図 **4** Linux と ***AnT*** 間のデータ授受に関係するメモリマッピング
Fig. 4 Memory mapping between Linux and ***AnT*** for data transfer.

（課題 3）について，Linux において，多くの機能は，可能な限りユーザ空間で処理し，カーネル空間での処理を最小化する．また，カーネル空間での処理は LKM として実現する．たとえば，Linux では，独自のキャラクタ型デバイスファイル "/dev/ant" を LKM として作成し，改造箇所を局所化する．

3.3.3 処理構造

Linux 入出力システムコール利用の処理は，依頼処理，代行処理，および連携処理からなる．処理の構造を図 **5** に示し，各処理の内容を以下に示す．なお，依頼処理と代行処理が使用するシステムコールの仕様を表 **1** に示す．

依頼処理は，独自 OS 上の AP プロセスが行う処理であり，Linux システムコールの代行実行を依頼し，結果を受け取る．処理の流れを以下に説明する．

(1) 独自 OS 上の AP プロセスは，LCA を確保する．
(2) LCA に Linux システムコールの引数と制御情報を格納する．
(3) linux_call システムコール（&LCA）を発行する．
(4) Linux 連携ドライバ内で仮想アドレスを実アドレスに変換する．次に，実アドレスを LCA 管理表の自コアの依頼用のエントリに格納する．
(5) Linux が動作しているコア 0 に，自コアに対応するベクタ番号で IPI を送信する．
(6) linux_call システムコール（&LCA）の結果を受け取る．
(7) LCA に格納された Linux システムコール実行結果を受け取る．

代行処理は，Linux システムコールを代行実行する処理である．親プロセスと代行プロセスで構成される．親プロ

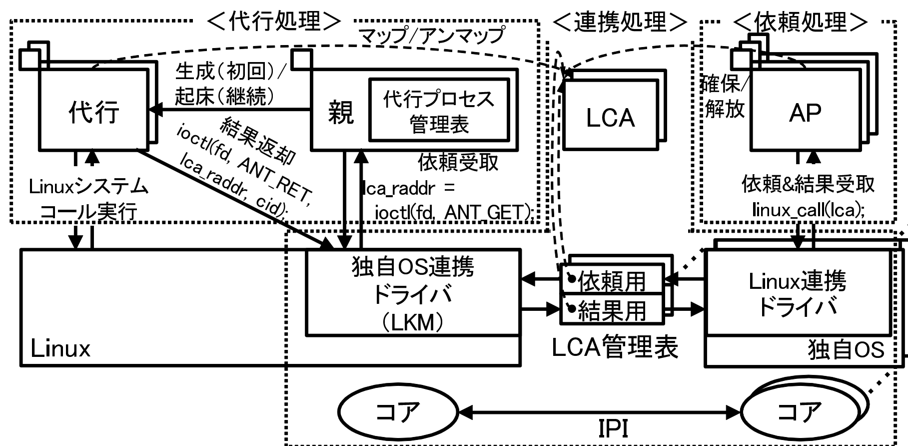


図 5　Linux 入出力システムコール利用の処理構造
Fig. 5　Procedure for utilization of Linux I/O system call.

表 1　Linux 入出力システムコール利用を実現するシステムコールの仕様
Table 1　Specification of Linux I/O system call.

機能	形式	処理概要	備考
依頼 & 結果受取	linux_call（lca） unsigned int lca;	LCA の先頭仮想アドレス lca を指定して依頼し，結果受取まで待つ．LCA には，自身が動作しているコア番号，Linux システムコールの引数，および継続するか否かの情報を格納しておく．	結果受取までブロックされる．
依頼受取	ioctl（fd, request） int fd; int request;	“/dev/ant” を open して得られるファイル記述子 fd に対して，request に ANT_GET を指定して発行する．request が ANT_GET の場合，ioctl システムコールは依頼受取を行う．このシステムコールの戻り値として，LCA の先頭実アドレス（raddr）を得る．	依頼受取までブロックされる．“/dev/ant” を open していることを前提とする．
結果返却	ioctl（fd, request, raddr, cid） int fd; int request; unsigned int raddr; int cid;	“/dev/ant” を open して得られるファイル記述子 fd に対して，request に ANT_RET を指定して発行する．request が ANT_RET の場合，ioctl システムコールは結果返却を行う．結果返却では，返却する結果を格納した LCA の先頭実アドレス raddr を指定して，結果返却先コア番号 cid のコアに IPI を送信する．	“/dev/ant” を open していることを前提とする．実装では，raddr と cid は 1 つの構造体にまとめ，この構造体を第 3 引数とする．

セスと連携処理の独自 OS 連携ドライバが協力して，以下の処理を繰り返し行う．

(1) 親プロセスは，ioctl（ANT_GET）システムコールを発行する．
(2) 独自 OS から依頼の IPI を受信するまでブロックされる．
(3) IPI の受信後，Linux は，ベクタ番号から受信した IPI の送信元コアを判別する．
(4) 独自 OS 連携ドライバは，LCA 管理表の対応するコアの依頼用エントリから，実アドレスを取得し，親プロセスに ioctl（ANT_GET）システムコールの戻り値として，実アドレスを返却する．
(5) 親プロセスは，ioctl（ANT_GET）システムコールの結果として，独自 OS からの Linux システムコール代行実行の依頼を受け取る．
(6) 代行プロセス管理表を検索し，当該 LCA を使用した依頼が初回か継続を確認する．
(7) 初回であれば実アドレスを引数として代行プロセスを生成し，継続であれば対応する代行プロセスを起床する．

連携処理は，依頼処理と代行処理の間で LCA を授受する処理である．この処理は OS 間の処理であり，各 OS への組み込みを容易するために連携ドライバとして実現した．代行プロセスと連携処理の独自 OS 連携ドライバが協力して，以下の処理を行う．

(1) 代行プロセスは，初回であれば，引数として受け取った実アドレスを用いて，LCA をメモリ空間にマップする（継続であれば，すでにマップしてあるため不要）．
(2) LCA の内容に基づいて，Linux システムコールを代行実行する．
(3) Linux システムコール実行結果を LCA に書き込む．
(4) ioctl（ANT_RET）システムコールを実行して，結果を連携処理に通知する．
(5) 独自 OS 連携ドライバは，結果返却コアに対応する LCA 管理表の結果用のエントリに LCA の実アドレスを格納する．
(6) Linux から結果送信先コアに IPI を送信する．
(7) 代行プロセスは，継続を示すフラグが ON であれば休眠し，そうでなければ LCA をアンマップして終了する．

© 2019 Information Processing Society of Japan

4. 評価

4.1 評価項目

3章で述べた手法を独自OSである ***AnT*** に実現した．以降では，提案手法の評価項目と評価の目的を示す．

（評価1）工数
　共存走行法とLinux入出力システムコール利用法それぞれの実現における ***AnT*** とLinuxの工数を明らかにし，OS機能やドライバの移植に対し，工数当たりの利用できる機能を比較評価した．

（評価2）pid取得とファイル操作システムコール処理時間
　pid取得とファイル操作システムコール利用において，各システムコールの処理時間を測定した．

（評価3）システムコール処理の分析
　nullシステムコールを利用して，継続時と非継続時のシステムコール処理を測定し，分析した．

（評価4）read/writeシステムコール処理時間
　read/writeシステムコールを利用して，シーケンシャルアクセスとランダムアクセスそれぞれについて処理時間を評価した．

（評価5）複数のLinuxシステムコール同時実行
　システムコール利用を行うAPプロセスを複数同時に実行し，その際のシステムコール利用オーバヘッドと代行実行の多重化の効果を評価した．

上記の（評価2～5）で使用した評価環境を**表2**に示す．***AnT*** からLinux入出力システムコール利用を行った場合の測定（以降，***AnT***-Linux）は，Linuxに1コア，***AnT*** に3コアを割り当てて行った．***AnT***-Linuxと比較するためのLinuxの測定は，Linuxに1コアを割り当てて行った．

表2 評価環境
Table 2 Evaluation environment.

プロセッサ	Intel(R) Core(TM) i7-3770，3.4 GHz（Linux：1コア，***AnT***：3コア）
メモリ	4,096 MB
OS	Linux 3.0.8（64 bit），***AnT***（32 bit）
HDD	ST500DM002（500 GB，7200 rpm，キャッシュ16 MB）

表3 共存走行実現における改造量（追加・変更）
Table 3 Amount of modified code to implement coexistence of Linux and ***AnT***.

OS	項目	ファイル数（個）	コード量（行）
AnT	メモリマップ関連	12	48
	その他	10	57
Linux（Mint）	カーネル，ブート用AP	7	374

表4 Linux入出力システムコール利用実現における改造量（追加・変更）
Table 4 Amount of modified code to implement Linux I/O system call interface to ***AnT***.

OS	項目	ファイル数（個）	コード量（行）
AnT	Linux連携ドライバ	9	121
Linux	Linuxカーネル	1	1
	独自OS連携ドライバ（LKM）	2	172
	親プロセス	7	161
	代行プロセス	2	100

4.2 工数

AnT のカーネル全体のファイル数165個，コード量18,505行に対して，***AnT*** の共存走行実現における改造量を**表3**に示す．***AnT*** の改造項目はメモリマップ関連とその他の大きく2つに分類できる．***AnT*** の改造量は，***AnT*** のカーネル全体に対して，ファイル数で約13%（=(12+10)/165 ∗ 100），コード量で約0.57%（=(48+57)/18505∗100）である．一方，Linux（Mint）は，文献[11]で実現されたMintをそのまま利用しており，Linuxと ***AnT*** が利用するコア，メモリ，デバイス指定の設定を変更するだけで利用可能である．これは，Kexecを利用することでMint側に独自のシステムコールを不要としたためである．文献[11]のLinux（Mint）の改造量は，カーネルの改変量がファイル数6個，コード量257行，およびブート用APの改変量が117行（全体：10,047行）であり，Linuxカーネルやブート用APのサイズに比べて，小さい．したがって，***AnT*** の改造量が ***AnT*** のカーネル全体のコード量の1%以下であり，Linux（Mint）改造量もカーネルやブートAPのコード行数に比べると小さいことから，開発工数を抑制できたといえる．メモリマップ関連の改造ファイル数が多い理由は，メモリマップを意識した処理が ***AnT*** の各ファイルに点在するためであった．***AnT*** のソースコードにおいて，メモリマップ関連の変数をdefine文で1つのファイルで定義する等の対処がなされていれば，修正量は少なくできたと推察する．また，その他の処理としては，コアの初期化処理，デバイス分割処理，および起動処理があり，改造量が少ないものの，機能がいろいろあるため改造ファイルが多い．

Linux入出力システムコール利用法の実現では，最初に機能を実現すれば以降の変更が不要である「Linux入出力システムコール利用実現に必須の項目」と，システムコール利用の種類を追加した場合に変更が必要である「ファイル操作に依存した項目」の2つがある．具体的には，システムコール利用実現に必須の項目は，Linux連携ドライバ，Linuxカーネル，独自OS連携ドライバ（LKM），および親プロセスの4つであり，ファイル操作に依存した項目は，代行プロセスの1つである．

Linux入出力システムコール利用法における改造量を**表4**に示す．表4より，Linux入出力システムコール利用実現に必須の項目のコード量は455行（= 121 + 1 + 172 + 161）であり，ファイル操作に依存した項目のコード量（代行プロセス）は100行である．なお，この100行には，

© 2019 Information Processing Society of Japan

open/close/read/write/stat/fstat/lseek システムコールの基本的なファイル操作を実現するためのシステムコール利用が含まれている．

システムコール利用の種類を追加する場合，Linux 入出力システムコール利用実現に必須の項目の変更は不要である．このため，この場合の修正箇所は，ファイル操作に依存した項目である代行プロセスに限定され，その工数は小さいといえる．

また，上記の基本的なファイル操作のシステムコール利用により，Linux で対応しているファイルシステムをすべて利用でき，これらのシステムコールで操作可能なドライバにも対応できる．一方，文献 [5] では，同様の機能を実現する場合，複数のドライバを移植する必要があるうえに，ファイルシステムを利用できない．したがって，提案手法は，少なくとも 1 コアを使用できなくなるものの，工数を削減でき，さらに独自 OS が未対応のファイルシステムを利用できる点で文献 [5] よりも効果が大きいといえる．

4.3　性能

4.3.1　pid 取得とファイル操作システムコール処理時間

表 5 に示す各システムコールの組合せで測定した．getpid と stat システムコールの場合は，代行プロセスの生成と消滅をともない，代行実行処理が単独で完結する．これに対し，open システムコールを含む場合は，代行プロセスの生成，継続，消滅が対応システムコールに応じて行われる．ここで，各システムコールのシステムコール利用オーバヘッドを明らかにするため，実入出力の処理時間を除いて測定した．なお，対象ファイルはすべて同一とし，read/write システムコールが扱うデータ長は 1 KB とした．各システムコールの処理時間を**図 6** に示す．

図 6 より，Linux のシステムコール処理時間に対するオーバヘッドは，各場合で約 0.09～0.1 ミリ秒であり，想定デバイスであるハードディスクの実入出力時間と比べて小さい．このため，システムコール利用オーバヘッドは十分小さいといえる．

また，いずれの場合においても ***AnT***-Linux の 1 つ目のシステムコール処理時間（代行プロセスの生成処理を含む）は，約 0.09 ミリ秒であり，これに対し，2 つ目以降のシステムコールの処理時間は，約 0.01 ミリ秒と短いことが分かる．このことから，継続機能を利用することで，システムコール利用オーバヘッドを削減できることが分かる．入出力システムコール利用は，継続を含んだ処理（例：open/･･･/close）が多いと考えられ，このような場合に継続機能は有効である．

なお，継続機能を利用した際の 1 プロセスあたりのメモリ使用量は，約 176 KB となった．継続機能により，処理時間のオーバヘッド削減は可能であるが，上記のメモリ使用量のオーバヘッドが生じる．処理時間とメモリ使用量のどちらを削減する方が有効かはシステムによって変わってくるため，本方式ではどちらかをユーザが制御できるようにした．

表 5　pid 取得とファイル操作システムコールの組み合わせ
Table 5　Combination of getpid and file operation system call.

組合せ	代行プロセスの動作
getpid	単独
stat	単独
open-close	生成–消滅
open-fstat-close	生成–継続–消滅
open-lseek-close	生成–継続–消滅
open-read(1 KB)-close	生成–継続–消滅
open-write(1 KB)-close	生成–継続–消滅

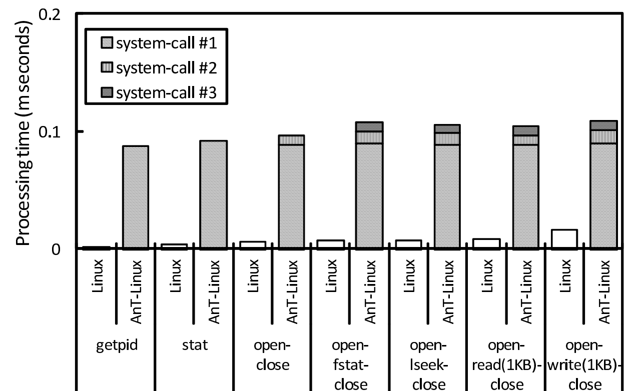


図 6　表 6 の各システムコールの処理時間
Fig. 6　Processing time of each system call of Table 6.

表 6　null システムコールの処理時間
Table 6　Processing time of null system call.

処理内容	処理時間
非継続時	141.8 μ 秒
継続時–遅延無	6.3 μ 秒
継続時–遅延有（1 ミリ秒）	47.0 μ 秒

4.3.2　システムコール処理の分析

図 6 のオーバヘッドの分析のために，***AnT*** 上の AP プロセスから null システムコール利用を行い，継続時と非継続時の処理時間を測定した．継続時については，連続して依頼した場合（継続時-遅延無）と ***AnT*** 上の他の処理により連続して依頼できない場合を考慮し，継続時において事前の依頼から遅延（1 ミリ秒）を入れて依頼した場合（継続時-遅延有）の処理時間を測定した．null システムコールの処理時間を**表 6** に示す．表 6 より，以下のことが分かる．

(1) 非継続時の処理時間は，継続時より長い．これは，非継続時の処理に代行プロセス生成が含まれており，代行プロセス生成の処理時間が長いためである．
(2) 継続時–遅延無の処理時間は，3 つのなかで最も短い．これは，代行プロセスが起床待ち状態に移行した後，次の依頼で親プロセスから起床されるまでの間，別の処理が何も行われず，CPU のメモリキャッシュにヒッ

© 2019 Information Processing Society of Japan

トしているためであると考えられる．

(3) 継続時–遅延有の処理時間は，非継続時の処理時間に比べて短い．一方，継続時–遅延無の処理時間に比べて長い．これは，遅延の間に Linux と ***AnT*** それぞれでタイマ割り込み処理が動作したことで，元のコンテキストに戻った後，CPU のメモリキャッシュミスが発生したためであると考えられる．実際に，***AnT*** の linux_call システムコールの最初と最後の区間において，L2 キャッシュミス回数を計測したところ，継続時–遅延無の場合は 117 回であり，継続時–遅延有の場合は 439 回であった．

以上から，代行プロセスの生成オーバヘッドが大きいことが分かる．また，継続機能を利用する際には，遅延の有無により，効果が変わることを示した．

4.3.3 read/write システムコール処理時間

read/write システムコールを利用して，シーケンシャルアクセスとランダムアクセスを行うことで，***AnT***-Linux と Linux の処理時間を比較した．

シーケンシャルアクセスは，open システムコールを実行し，読み込みの場合 read システムコール，書き込みの場合 write システムコールを 1,000 回繰り返し実行し，close システムコールを実行する．ランダムアクセスは，シーケンシャルアクセスの各 read/write システムコールの直前にランダムな箇所を指定した lseek システムコールを加えて実行する．測定区間は，いずれも open システムコールの直後から close システムコールの直前までとした．I/O 処理時間を含めた処理時間を明らかにするため，測定では毎回ディスクキャッシュをクリアして行った．なお，対象ファイルのサイズは 200 MB である．

シーケンシャルアクセスの処理時間を図 **7** に示す．図 7 より，以下のことが分かる．

(1) 多くの場合，***AnT***-Linux の処理時間は Linux より長い．これは，システムコール利用オーバヘッドのためである．なお，データ長が 2 KB の write システムコールの場合，***AnT***-Linux の処理時間は Linux より短い．これは，write システムコールの発行間隔の違いにより，DK アクセス時間に差が生じているためと推察される．

(2) ***AnT***-Linux の Linux に対するオーバヘッドは read システムコールについて約 0.3〜4.5 ミリ秒，write システムコールについて約 −2.8〜5.0 ミリ秒である．各 read/write システムコールが扱うデータ長が 1 KB の場合，オーバヘッドの割合は，read システムコールで約 3%，write システムコールで約 2%である．

ランダムアクセスの処理時間を図 **8** に示す．図 8 より，以下のことが分かる．

(1) ***AnT***-Linux の処理時間は Linux とほぼ同様である．これは，I/O 処理時間が長く，I/O 処理時間に比べてシステムコール利用オーバヘッドが小さいためである．なお，

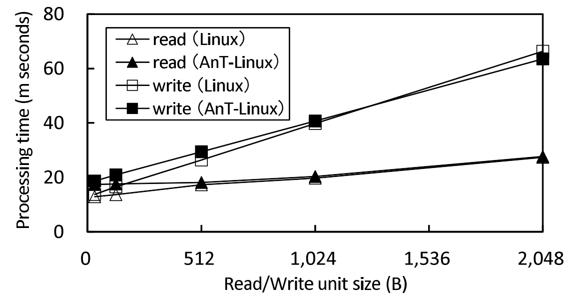


図 **7** シーケンシャルアクセスの処理時間
Fig. 7 Processing time of sequential access.

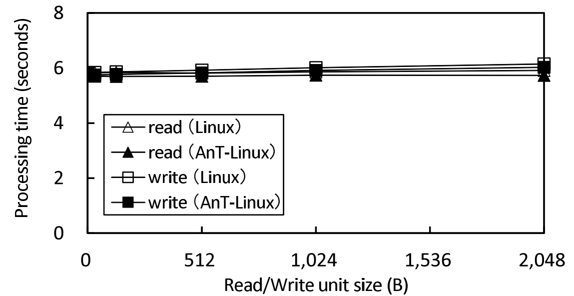


図 **8** ランダムアクセスの処理時間
Fig. 8 Processing time of random access.

AnT-Linux の処理時間は Linux より全体的に少し短い．これも，シーケンシャルアクセスと同様に，DK アクセス時間に差が生じているためと推察される．

(2) ***AnT***-Linux の Linux に対するオーバヘッドは read システムコールについて約 −0.11〜−0.18 秒，write システムコールについて約 −0.09〜−0.11 秒である．各 read/write システムコールが扱うデータ長が 1 KB の場合，オーバヘッドの割合は，read システムコールで約 −2%，write システムコールで約 −2%である．

図 7 と図 8 より，実入出力を含めたシステムコール利用において，オーバヘッドを数%程度に抑えられていることが分かる．

4.3.4 複数の Linux システムコール同時実行

シーケンシャルアクセス/ランダムアクセス読み込みで read システムコールが扱うデータ長を 1 KB とした 4.3.3 項の測定プログラムを利用し，これを複数コア上で複数プロセス（N コア M プロセス）同時実行した．測定区間は，測定 AP プロセス生成処理の直前から全測定 AP プロセス終了待ち処理の直後までとした．読み込む対象ファイルは，同じサイズのファイルをプロセス数分用意した．以降，シーケンシャルアクセスを利用した場合を多重シーケンシャルアクセス，ランダムアクセスを利用した場合を多重ランダムアクセスと呼ぶ．また，それぞれの場合において代行するシステムコールは，多重シーケンシャルアクセスで open/read/close システムコール，多重ランダムアクセスで open/lseek/read/close となる．なお，複数プロセスを同時実行した場合でも，システムコール代行の際のシ

© 2019 Information Processing Society of Japan

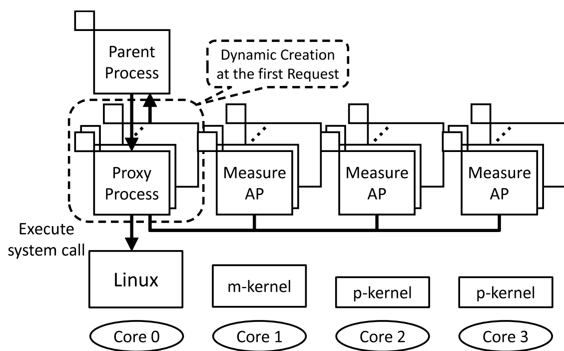


図 9　複数の Linux システムコール同時実行の様子
Fig. 9　Workflow of simultaneous on Linux system calls.

ステムコール発行順は AP プロセス/代行プロセスのペアにおいて保証される．複数の Linux システムコール同時実行の様子を**図 9** に示し，以下に説明する．

(1) 1 コア M プロセス

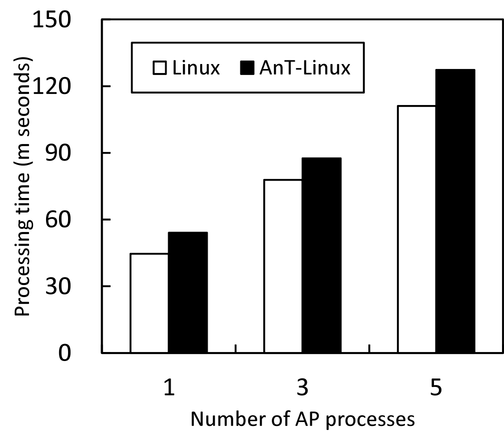
AnT-Linux と Linux の処理時間を比較した．***AnT***-Linux の場合，Linux はコア 0，***AnT*** はコア 1 を使用し，コア 0 上で親プロセスと代行プロセス，コア 1 上で測定 AP プロセスが複数動作する．Linux の場合，Linux はコア 0 のみ使用し，コア 0 上で測定 AP プロセスが複数動作する．

(2) N コア M プロセス

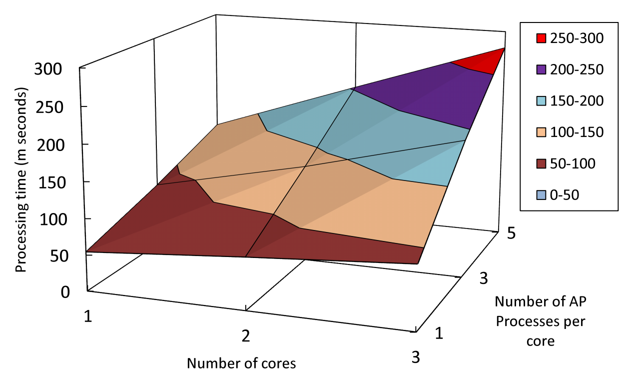
コア 0 上で親プロセスと代行プロセスが動作し，コア 1，コア 2，およびコア 3 上で測定 AP プロセスが複数動作する．

多重シーケンシャルアクセスの処理時間を**図 10** に示す．図 10 (a) より，いずれのプロセス数においても，***AnT***-Linux の処理時間は Linux より少し長い程度であり，システムコール利用オーバヘッドは小さいといえる．1 コア 5 プロセスの場合，***AnT***-Linux の処理時間は Linux の 115%である．多重ランダムアクセスも同様の傾向が得られ，1 コア 5 プロセスの場合，104%であった．図 10 (b) より，処理時間の長大化は測定 AP プロセス総数の影響を受け，測定 AP プロセスが走行するコア総数に影響されないことが分かる．

したがって，Linux が 1 コアで動作し代行実行するため，プロセス数を増加させてシステムコール実行を多重に要求できても，代行実行時の実入出力がシーケンシャルに行われ，この処理がボトルネックになり，多重要求できる効果は確認できなかった．たとえば，図 10 (a) でプロセス数 3 の場合，処理時間の大半（Linux の場合 94%，***AnT***-Linux の場合 81%）が実入出力時間であり，これを裏付けている．以上のこのことから，代行実行においてシーケンシャルな処理にならない場合（たとえば，異なる磁気ディスク装置へのアクセス）には，効果があると推察する．



(a) One core M processes (Linux vs ***AnT***-Linux)



(b) N cores M processes (***AnT***-Linux)

図 10　多重シーケンシャルアクセスの処理時間
Fig. 10　Processing time of simultaneous sequential access.

5. 関連研究

独自 OS と Linux を共存走行させ，独自 OS 上で発行された Linux システムコールを Linux に代行実行させる手法 [10], [12], [13] が提案されている．

文献 [10] では，Linux 上で動作する既存 AP をそのまま独自 OS 上で動作させるため，Linux の完全なシステムコール互換を目指している．また，両 OS 間で Linux システムコール引数に利用する領域を予約しておくことで，システムコール利用の種類を追加する際，コードの追記を不要としている．しかし，これを実現するには，独自 OS のメモリ空間を Linux を意識したものに改造する必要がある．一方，提案手法は，独自 OS 側の AP が Linux の入出力機能を実現するものであり，独自 OS 側で利用するシステムコールの種類が少ないことから，工数増加の抑制する設計を行っている．

文献 [12] では，独自 OS から Linux システムコールの代行実行を非同期で依頼することで代行実行のスループット

© 2019 Information Processing Society of Japan

を向上させている．これを実現するには，依頼や結果を溜めておく処理が必要となる．提案手法は，同期で Linux システムコール処理を依頼することで工数を小さくしている．

文献 [13] では，独自 OS において 1 コア上に 1 プロセスのみ配置することで，多重化によるシステムコール利用オーバヘッドを削減している．この手法は，独自 OS におけるプロセス配置に制限が生じるため，提案手法には適用できない．

組み込みシステムを対象として，軽量な仮想マシンモニタ上に Linux とリアルタイム OS（以降，RTOS）を共存したシステム [14]～[16] が提案されている．これらの目的は，Linux による汎用性向上，RTOS 上で動作する AP の再利用，および RTOS のリアルタイム性の保証である．文献 [16] では，さらに両 OS 間でデータ授受を実現している．具体的には，両 OS 間で双方向の通信用チャネルを用意し，共有メモリとイベント通知により実現している．これらの研究は複数 OS が共存するものの，提案手法とは一方の OS が他方の OS に入出力機能を提供するために共存し，このための入出力操作を利用する機能を OS 間で実現している点が異なる．

6. おわりに

AnT オペレーティングシステムにおける Linux 入出力操作機能の利用手法について述べた．具体的には，マルチコアプロセッサを利用して，Mint を基盤として ***AnT*** と Linux を共存走行させる手法を述べた．また，Linux 入出力操作機能として Linux 入出力システムコールを ***AnT*** から利用する手法を示し，依頼処理，代行処理，および連携処理について述べた．

次に，共存走行法と Linux 入出力システムコール利用法それぞれの実現における ***AnT*** と Linux の工数，および Linux 入出力システムコール利用のオーバヘッドを評価した．工数の評価では，共存走行実現の工数が小さいこと，および Linux 入出力システムコール利用実現が文献 [5] のドライバ単位での移植に比べて，一度に複数のドライバやファイル操作機能が利用できるようになるため，工数に対する効果が大きいことを明らかにした．

また，pid 取得とファイル操作システムコール処理時間，read/write システムコール処理時間，および複数の Linux システムコール同時実行の性能について明らかにした．具体的には，pid 取得とファイル操作システムコール処理時間では，いずれのシステムコール利用でもシステムコール利用オーバヘッドが約 0.1 ミリ秒であること，および継続機能の効果を示した．read/write システムコール処理時間では，実入出力を含めたシステムコール利用において，オーバヘッドを数%程度に抑えられていることを明らかにした．複数の Linux システムコール同時実行の評価では，***AnT*** 上の複数コア，複数 AP プロセスから同時にシステムコール利用を行った場合，処理時間の長大化は測定 AP プロセス総数の影響を受け，測定 AP プロセスが走行するコア総数に影響されないことを明らかにした．

残された課題として，他の独自 OS に本手法を適用した際の工数と性能に関する評価がある．

参考文献

[1] 奥野幹也, 片山徹郎, 最所圭三, 福田　晃：UNIX 系 OS におけるデバイスドライバの抽象化と生成システムの実現，情報処理学会論文誌，Vol.41, No.1, pp.1755–1765 (2000).

[2] Zhang, Q.-L., Zhu, M.-Y. and Chen, S.-Y.: Automatic generation of device drivers, *ACM SIGPLAN Notices*, Vol.38, No.6, pp.60–69 (2003).

[3] O'Nils, M. and Jantsch, A.: Operating system sensitive device driver synthesis from implementation independent protocol specification, *Proc. Design, Automation and Test in Europe Conference and Exhibition*, pp.563–567 (Mar. 1999).

[4] Thibault, S.A., Marlet, R. and Consel, C.: Domain-specific languages: From design to implementation application to video device drivers generation, *IEEE Trans. Softw. Eng.*, Vol.25, No.3, pp.363–377 (1999).

[5] 島崎　泰，山内利宏，乃村能成，谷口秀夫：***AnT*** オペレーティングシステムにおける Linux の LKM 形式ドライバのプロセス化手法，電子情報通信学会論文誌（D），Vol.J93-D, No.10, pp.1990–2000 (2010).

[6] 岡本幸大，谷口秀夫：***AnT*** オペレーティングシステムにおける高速なサーバプログラム間通信機構の実現と評価，電子情報通信学会論文誌（D），Vol.J93-D, No.10, pp.1977–1987 (2010).

[7] 井上喜弘，佐古田健志，谷口秀夫：マルチコアプロセッサ上での負荷分散を可能にする ***AnT*** オペレーティングシステム，情報処理学会研究報告，Vol.2012-DPS-150, No.37, pp.1–8 (Mar. 2012).

[8] Nomura, Y., Senzaki, R., Nakahara, D., Ushio, H., Kataoka, T. and Taniguchi, H.: Mint: Booting Multiple Linux Kernels on a Multicore Processor, *Proc. 6th International Conference on Broadband, Wireless Computing, Communication and Applications*, pp.555–560 (Oct. 2011).

[9] McKernel — R-CCS Sys-Soft, available from ⟨http://www.sys.r-ccs.riken.jp/ResearchTopics/os/mckernel/⟩ (accessed 2019-07-01).

[10] 佐伯裕治，清水正明，白沢智輝，中村　豪，高木将通，Balazs Gerofi，思　敏，石川　裕，堀　敦史：ヘテロジニアス計算機上の OS 機能委譲機構，情報処理学会研究報告，Vol.2013-OS-125, No.15, pp.1–8 (Apr. 2013).

[11] 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 (Nov. 2010).

[12] Sato, M., Fukazawa, G., Nagamine, K., Sakamoto, R., Namiki, M., Yoshinaga, K., Tsujita, Y., Hori, A. and Ishikawa, Y.: A Design of Hybrid Operating System for a Parallel Computer with Multi-Core and Many-Core Processors, *Proc. 2nd International Workshop on Runtime and Operating Systems for Supercomputers*, No.9, pp.1–8 (June 2012).

[13] Park, Y., Hensbergen, E.V., Hillenbrand, M., Inglett, T., Rosenburg, B., Ryu, K.D. and Wisniewski, R.W.: FusedOS: Fusing LWK Performance with FWK Functionality in a Heterogeneous Environment, *Proc. IEEE 24th International Symposium on Computer Architecture and*

© 2019 Information Processing Society of Japan

High Performance Computing, pp.211–218 (Oct. 2012).
[14] Mitake, H., Lin, T.-H., Shimada, H., Kinebuchi, Y., Li, N. and Nakajima, T.: Towards Co-existing of Linux and Real-Time OSes, *Linux Symposium 2011*, Vol.6, pp.55–68 (2011).
[15] 杉本　健，野尻　徹，平松義崇，寺田光一：組み込み向けマルチコアプロセッサにおける複数 OS 実行環境の構築技術，情報処理学会研究報告，Vol.2010-OS-114, No.3, pp.1–8 (Apr. 2010).
[16] Sangorrin, D., Honda, S. and Takada, H.: Reliable and Efficient Dual-OS Communications for Real-Time Embedded Virtualization, Computer Software, Vol.29, No.4, pp.182–198 (2012).

山内 利宏　（正会員）

1998 年九州大学工学部情報工学科卒業．2000 年同大学大学院システム情報科学研究科修士課程修了．2002 年同大学院システム情報科学府博士後期課程修了．2001 年日本学術振興会特別研究員（DC2）．2002 年九州大学大学院システム情報科学研究院助手．2005 年岡山大学大学院自然科学研究科助教授．現在，同准教授．博士（工学）．オペレーティングシステム，コンピュータセキュリティに興味を持つ．2010 年度 JIP Outstanding Paper Award，2012 年度情報処理学会論文賞等受賞．電子情報通信学会，ACM，USENIX，IEEE 各会員．本会シニア会員．

福島 有輝

2014 年岡山大学工学部情報工学科卒業．2016 年同大学大学院自然科学研究科博士前期課程修了．同年，三菱電機マイコン機器ソフトウエア（株）入社．オペレーティングシステムに興味を持つ．

乃村 能成　（正会員）

1993 年九州大学工学部電子工学科卒業．1995 年同大学大学院情報工学専攻修士課程修了．同年九州大学工学部助手．1996 年九州大学大学院システム情報科学研究科助手．2003 年岡山大学工学部講師．現在，岡山大学大学院自然科学研究科准教授．博士（情報科学）．オペレーティングシステム，ソフトウェア開発環境，グループウェアに興味を持つ．電子情報通信学会会員．本会シニア会員．

谷口 秀夫　（正会員）

1978 年九州大学工学部電子工学科卒業．1980 年同大学大学院修士課程修了．同年日本電信電話公社電気通信研究所入所．1987 年同所主任研究員．1988 年 NTT データ通信株式会社開発本部移籍．1992 年同本部主幹技師．1993 年九州大学工学部助教授．2003 年岡山大学工学部教授．2010 年岡山大学工学部長．2014 年岡山大学理事・副学長．博士（工学）．オペレーティングシステム，実時間処理，分散処理に興味を持つ．著書『並列分散処理』（コロナ社）等．電子情報通信学会，ACM 各会員．本会フェロー．

© 2019 Information Processing Society of Japan