Vol.2015-OS-132 No.11
2015/2/27

Linux と *AnT* オペレーティングシステムの連携機構の評価

福島 有輝[1]　山内 利宏[1]　乃村 能成[1]　谷口 秀夫[1]

概要：特定のサービスに適した独自 OS を利用し，さらにマルチコアプロセッサを活かして独自 OS からの既存 OS プログラム利用による独自 OS の利便性向上を目指した Linux と ***AnT*** オペレーティングシステムの混載システムを提案した．Linux は，多種の入出力機器の利用を可能とし，かつ既存 OS として普及しており，多くのサービスを実現している．一方，***AnT*** は，マイクロカーネル構造を有し，適応性と堅牢性を特徴とする独自 OS である．この両者の特徴を生かしたサービス提供のために，両 OS の連携機構を設計した．この連携機構は，***AnT*** から Linux にシステムコール代行実行を処理依頼する．本稿では，***AnT*** からの Linux システムコール代行実行の評価について報告する．具体的には，連携機構実現における工数と Linux システムコール代行実行の性能について述べる．

1. はじめに

マルチコアプロセッサを有する計算機の増加に伴い，マルチコアプロセッサを利用した効率的なサービス提供が求められている．これを実現するには，複数のコアに処理をうまく分散できることが重要である．また，様々なサービス提供を行うには，そのサービスに適した基盤ソフトウェアの利用が有効である．

そこで，特定のサービスに適した独自 OS を利用する方法がある．しかし，独自 OS では，独自 OS 用の応用プログラム（以降，AP）しか動作せず，利用できる入出力機器は少ない．独自 OS 上でも既存 OS の AP や入出力機器を利用できれば，独自 OS の利便性を向上できる．しかし，既存 OS のプログラムを独自 OS に移植する工数は大きい．そこで，マルチコアプロセッサを利用して，コア毎に独自 OS と既存 OS を独立して動作させ，両 OS で連携させれば，既存 OS のプログラムを移植なしに利用できる．

実際に，我々は，独自 OS と既存 OS を混載した Linux と ***AnT*** オペレーティングシステム [1]（以降，***AnT***）の混載システムを提案した [2]．Linux は，多種の入出力機器の利用を可能とし，かつ既存 OS として普及しており，多くのサービスを実現している．一方，***AnT*** は，マイクロカーネル構造 [3]～[6] を有し，適応性と堅牢性を特徴とする独自 OS である．この両者の特徴を生かしたサービス提供のために，両 OS の連携機構を設計した [7]．この連携機構は，***AnT*** から Linux にシステムコール代行実行を処理依頼する．

独自 OS から Linux にシステムコール代行実行を処理依頼する方式は，文献 [8]～[10] で提案されている．しかし，これらの方式は，独自 OS と Linux が密に協調して動作することを前提に設計されている．一方，我々の連携機構は，独自 OS と Linux をできる限り独立化した状態に保ちつつ，高速に代行実行することを目指している．

本稿では，***AnT*** からの Linux システムコール代行実行の評価について報告する．具体的には，連携機構実現における工数と Linux システムコール代行実行の性能について述べる．

2. Linux と *AnT* の混載システム

AnT は m-カーネルと p-カーネルの 2 種類のカーネルを有する．m-カーネルは，電源投入時に最初に起動するコアを制御し，スケジュール機能やメモリ管理機能といったマイクロカーネルに必要な全機能を有する．一方，p-カーネルは，m-カーネルが制御するコア以外のコアを制御し，機能を例外・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に絞り，軽量化を図っている．

AnT の仮想空間における特徴的な構造として，コア間通信データ域 (以降，ICA: Inter-core Communication Area) がある．ICA は，プロセス間で共用可能な空間に位置付けられているため，プロセス間の高速なデータ授受が可能である．Linux と ***AnT*** の連携機構は，この ICA を使用する．

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

© 2015 Information Processing Society of Japan

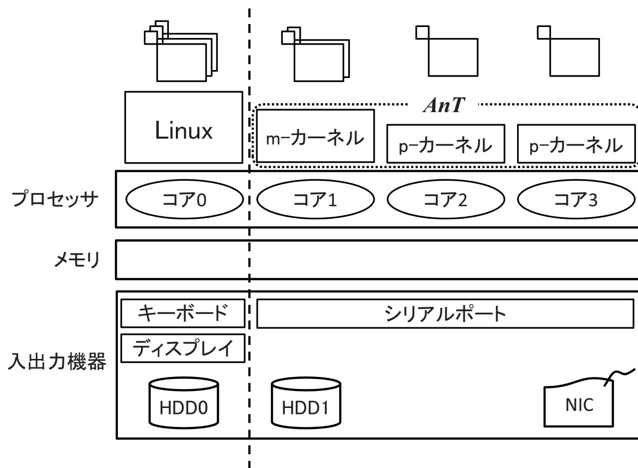

図 1 Linux と ***AnT*** の混載システムの構成例（4 コアの場合）

Linux と ***AnT*** の混載システムは，Mint [11] を拡張して実現している．Mint は，仮想化によらず 1 台の計算機上で複数の Linux を独立に同時走行させる OS である．Linux と ***AnT*** の混載システムの構成例（4 コアの場合）を**図 1** に示す．図 1 では，各 OS にプロセッサ，メモリ，および入出力機器を以下のように割り当てている．

（1）プロセッサ
Linux にコア 0，***AnT***（m-カーネル）にコア 1，***AnT***（p-カーネル）にコア 2 とコア 3 を割り当てる．

（2）メモリ
Linux と ***AnT*** に実メモリを空間分割して割り当てる．

（3）入出力機器
Linux にキーボード，ディスプレイ，および HDD0，***AnT*** にシリアルポート，NIC，および HDD1 を割り当てる．

なお，上記の構成から，Linux への複数コア割り当て，使用するメモリ量の変更，および入出力機器の構成の変更が可能である．

3. Linux と *AnT* の連携機構

3.1 設計方針

Linux システムコール実行の処理は，***AnT*** 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 ***AnT*** から Linux に転送して代行実行し，実行結果を Linux から ***AnT*** に返送し，結果を AP プロセスに返却する．

このLinux システムコール実行の処理を実現する設計方針を以下に示す．

（方針 1）高速化
高速な Linux システムコール実行処理を実現する．

（方針 2）多重化
Linux において，***AnT*** 上の複数 AP プロセスからの Linux システムコール発行を可能にし，Linux 上での Linux システムコール代行実行処理の多重化を実現する．

（方針 3）効率化
Linux システムコールには，getpid() のように単独で処理を行えるものと open()，read()，および close() のように複数のシステムコール間で関連するものの大きく二つに分類できる．この関連する Linux システムコール群について，処理の効率化を実現する．

（方針 4）独立化と局所化
連携において，両 OS が相手 OS 固有機能を意識しない形で実現する．つまり，各 OS の独立性を保つ．また，その実現においては，既存 OS 機能部分への変更を抑制し，局所化を図る．

3.2 課題

各方針を満たすための課題について述べる．課題として以下の三つがある．

（課題 1）データ転送方式
（方針 1）を満足するため，Linux と ***AnT*** 間で複写レスなデータ授受を可能にするデータ転送方式を確立する．

（課題 2）代行実行方式
（方針 2）を満足するため，Linux 上での Linux システムコール代行実行処理においては，複数の代行実行処理を行える代行プロセス形態が必要である．つまり，***AnT*** 上の同時に発行する各 AP プロセスからの依頼毎に Linux システムコールを実行できるような代行プロセス形態を確立する．

（課題 3）代行処理継続方式
（方針 3）を満足するため，関連システムコール群においては，代行処理を継続する方式を確立する．

3.3 対処

3.3.1 データ転送方式

Linux と ***AnT*** 間のデータ授受は，***AnT*** から始まり ***AnT*** で終わる．つまり，***AnT*** が処理を依頼し，Linux が実行して結果を返却し，***AnT*** が結果を受け取る．したがって，データ授受に使用する領域は，***AnT*** が確保し，解放する方法が良い．また，この領域の実メモリを両 OS で授受できれば，データ授受を複写レスで行うことができる．

3.3.2 代行実行方式

代行実行方式は，代行主体の決定，代行プロセスの管理方式，および代行プロセスの生成方式の三つからなる．それぞれについて以下で説明する．

（1）代行主体の決定
Linux 上で Linux システムコールを代行する主体として，プロセスとスレッドの二つが考えられる．ここでは，「Linux 上の代行主体同士の独立性を高めること」と「***AnT*** 上の実行主体（プロセス）と機能的に対応を取りやすいこと」により，Linux システムコール代行主体はプロセスとする．

（2）代行プロセスの管理方式
AnT 上の複数の AP プロセスからの処理依頼を代行する

© 2015 Information Processing Society of Japan

プロセスを生成するために，親プロセスを生成しておく．この親プロセスは，***AnT*** からの処理依頼を受け取り，受け取る度に fork() し，生成された代行プロセスが実際に Linux システムコールを代行実行する．

(3) 代行プロセスの生成方式

代行プロセスの生成方式として，静的生成方式と動的生成方式の二つがある．以下でそれぞれ説明する．

（方式 1）静的生成方式

AnT 上のプロセスと Linux 上のプロセスが一対一対応するように，***AnT*** のプロセス生成に同期して Linux 上に代行プロセスを生成する．

（方式 2）動的生成方式

AnT からの処理依頼を Linux 上の親プロセスが受け取る度に，代行プロセスを生成する．

静的生成方式は，***AnT*** 上のプロセスと対応するプロセスが Linux 上にすでに存在するため，処理依頼受取から Linux システムコール代行実行までの処理時間が短いという利点がある．しかし，両 OS で同期が必要である．具体的には，***AnT*** 上のプロセスの生成と終了に同期して，Linux 上でも対応するプロセスの生成と終了をしなければならない．これは，（方針 4）の独立化に反する．

動的生成方式は，両 OS で同期が不要である．しかし，処理依頼受取の度にプロセスを生成するため，処理依頼受取から Linux システムコール代行実行までの処理時間が長くなるという欠点がある．ただし，この欠点には，プロセスをある程度の数だけ事前生成しておくという対処が考えられる．

ここでは，（方針 4）により，動的生成方式を使用する．

3.3.3 代行処理継続方式

Linux システムコール代行処理の継続を実現するためには，以下の三つの対処が必要である．

（対処 1）Linux システムコール代行実行後に代行プロセスを継続

これにより，例えば，open() 後に得られるファイル記述子を保つことができる．

（対処 2）代行プロセスとデータ授受に使用する領域の先頭実アドレスの対応を保存

処理依頼受取後に保存した対応を確認し同一の先頭実アドレスを見つけた場合，継続している代行プロセスが存在していると判断できる．これにより，***AnT*** 上の AP プロセスからの 2 回目以降の処理依頼で，1 回目の処理依頼時と同じ Linux 上の代行プロセスを使用できるため，ファイル記述子を指定して read() や write() を実行できる．なお，（方針 4）の局所化により，Linux カーネルの既存の構造をできる限り改造しないようにするため，代行プロセスとデータ授受に使用する領域の先頭実アドレスの対応表は，カーネルではなく親プロセスにもたせる．

（対処 3）継続するか否かの通知機構を ***AnT*** に用意

これにより，代行処理完了後の代行プロセスの終了可否を制御可能となる．

4. 実現方式

4.1 Linux システムコールの内容を授受するシステムコール

4.1.1 基本機能

Linux システムコールの内容を授受するシステムコールでは，Linux システムコールの依頼内容と実行結果，および制御情報を格納する領域（以降，LCA: Linux system Call information Area）を使用する．LCA は，***AnT*** の ICA を使用して実現し，ページテーブル 1 エントリで構成される．このため，LCA1 つのサイズは 4 KB である．また，3.3.3 項の（対処 3）を実現するため，LCA に継続フラグをもたせ，継続する場合，継続フラグを立てて依頼する．なお，両 OS 間の通信には，プロセッサ間割り込み（以降，IPI: Inter-Processor Interrupt）を使用する．

Linux システムコールの内容を授受するシステムコールは，処理依頼，処理依頼受取，結果返却，および結果受取の 4 機能である．ただし，Linux システムコール実行は同期的に行うため，処理依頼と結果受取は一つのシステムコールとして実現する．（方針 4）に従い，機能の実現は Linux 側では LKM（Loadable Kernel Module）を使用し，***AnT*** でも局所化を図る．なお，LKM として ***AnT*** と連携を行う独自のキャラクタデバイス “/dev/ant”（以降，***AnT*** 連携デバイス）を作成し，Linux ではこのデバイスに対して操作を行うことで処理依頼受取と結果返却の機能を実現する．

4.1.2 ポインタ引数の扱い

Linux システムコールには，引数としてポインタを含むものがある．このポインタ先の情報を授受する方式として，大きく以下の 2 方式がある．

（方式 1）シリアライズ方式

ポインタ先のデータを LCA にシリアライズして格納する．

（方式 2）メモリマッピング方式

特定の仮想アドレス領域上にポインタ先のデータを格納する．この仮想アドレス領域を Linux と ***AnT*** で予約しておき，ページ例外を使用してオンデマンドでページテーブルを登録するか，または共有メモリとして使用する．

メモリマッピング方式では，両 OS のメモリ空間に相手 OS を意識した依存が生じてしまう．ここでは，（方針 4）により，両 OS 間に依存がないシリアライズ方式を使用する．

4.2 処理流れ

Linux システムコール実行の処理流れを**図 2** に示し，以下で説明する．

(1) 処理依頼

AnT 上の AP プロセスは，ICA を使用して LCA を確保す

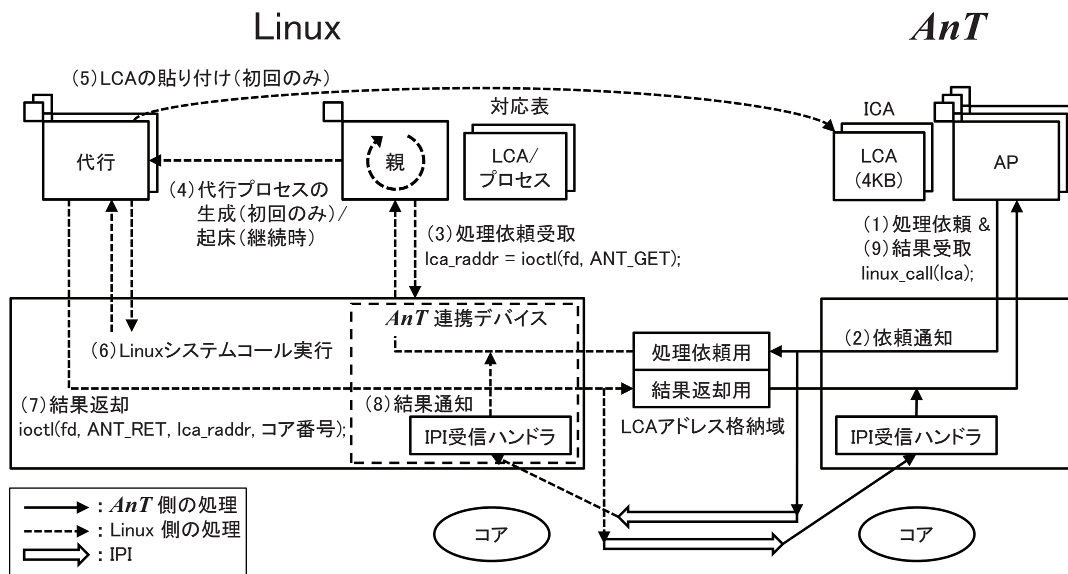

図 2　Linux システムコール実行の処理流れ

る．依頼する Linux システムコールの番号と引数（ポインタ引数の場合，シリアライズ）を確保した LCA に書き込み，LCA を指定して処理依頼システムコール linux_call() を発行する．なお，AP プロセスは結果受取までブロックされる．

（2）依頼通知

AnT のカーネルは，LCA の先頭実アドレスを LCA アドレス格納域（処理依頼用）に格納し，Linux が動作するコアに対して IPI を送信する．

（3）処理依頼受取

AnT 連携デバイスは，上記（2）の IPI 受信を契機として，***AnT*** 連携デバイス内の IPI 受信ハンドラを実行し，処理依頼受取システムコール ioctl(ANT_GET) で待ち状態となっている親プロセスを起床する．親プロセスは，処理依頼受取システムコールの延長で，LCA の先頭実アドレスを LCA アドレス格納域から読み出し，処理依頼を受け取る．

（4）代行プロセスの生成/起床

親プロセスは，LCA の先頭実アドレスをキーとして対応表を検索し，代行プロセスが存在しない（初回）場合，代行プロセスを生成する．代行プロセスが存在する（継続時）場合，当該代行プロセスを起床する．

（5）LCA の貼り付け

代行プロセスは，LCA を自身の仮想空間に貼り付ける．なお，継続時の場合，すでに貼り付いているため，この処理を行わない．

（6）Linux システムコール実行

代行プロセスは，LCA をもとに Linux システムコールの引数を再構築し，Linux システムコールを実行する．また，Linux システムコール実行の結果を LCA に書き込む．

（7）結果返却

代行プロセスは，***AnT*** 連携デバイスに対して，LCA の先頭実アドレスと返却先コア番号を指定して，結果返却システムコール ioctl(ANT_RET) を発行し，***AnT*** に結果を返却する．結果返却後，代行プロセスは LCA を剥がし，終了する．なお，継続時の場合，親プロセスからの起床待ち状態となる．

（8）結果通知

AnT 連携デバイスは，LCA の先頭実アドレスを LCA アドレス格納域（結果返却用）に格納し，***AnT*** が動作するコアに対して IPI を送信する．

（9）結果受取

AnT のカーネルは，上記（8）の IPI 受信を契機として，各カーネル固有の IPI 受信ハンドラを実行し，LCA の先頭実アドレスにより，***AnT*** 上のどの AP プロセスが結果受取待ちしているかを見つけ，当該プロセスを起床させる．起床したプロセスは，結果を受け取り，処理依頼を完了する．

4.3　期待される効果

以下の四つの効果が期待できる．

（効果 1）データ転送方式により，LCA の先頭実アドレスを両 OS 間でやり取りすることで，LCA の内容を複写レスで授受できるため，高速化が期待できる．

（効果 2）代行実行方式により，同時に複数の処理依頼を代行実行できるため，多重化が期待できる．

（効果 3）代行処理継続方式により，処理依頼毎の代行プロセス生成が不要になるため，効率化が期待できる．

（効果 4）Linux 側のカーネル空間の処理を ***AnT*** 連携デバイスに任せることで，Linux カーネルの改造量の抑制が期待できる．

表 1　Linux の改造量（追加のみ，変更なし）

項目	ファイル数（個）	コード量（LOC）
Linux カーネル	1	1
AnT 連携デバイス	2	172
親/代行プロセス	9	238

表 2　***AnT*** のカーネルの改造量

項目	ファイル数（個）	コード量（LOC）
機能追加前	235	15,266
機能追加後	238	15,393
追加/変更箇所	9（3.8%）	121（0.8%）

5. 評価

5.1 工数

Linux と ***AnT*** の連携機構の実現に要した工数を明らかにするため，Linux の改造量（Linux カーネル，***AnT*** 連携デバイス，および親/代行プロセス）と ***AnT*** のカーネルの改造量における追加/変更したファイル数とコード量について評価した．なお，本評価では，コード量の評価基準として，論理 LOC（Lines Of Code）を使用した．論理 LOC は，ソースコードの総行数から，記号だけの行，空白行，およびコメントのみの行を省いた総数である．論理 LOC の算出には，LocMetrics[12] を使用した．

まず，Linux の改造量（Linux カーネル，***AnT*** 連携デバイス，および親/代行プロセス）を**表 1** に示す．なお，Linux の改造量については，変更した箇所はなく，改造は追加のみである．表 1 より，Linux カーネルを改造したファイル数とコード量は，それぞれ 1 個と 1 行である．この改造量は Linux カーネル全体のコード量に比べて極めて小さい．このため，改造箇所を局所化できているといえる．なお，改造内容は，処理依頼受取で必要となる IPI 受信ハンドラを登録するベクタ管理表を LKM から操作可能にすることである．***AnT*** 連携デバイスと親/代行プロセスについて，これらのコード量は，合わせて 400 行程度であり，容易に実現できるといえる．

次に，***AnT*** のカーネルの改造量を**表 2** に示す．表 2 より，機能追加のために追加/変更したファイル数とコード量は，それぞれ 9 個と 121 行であり，機能追加前のファイル数とコード量のそれぞれ約 3.8%と約 0.8%にあたる．これにより，連携機構に依存する部分を局所化できているといえる．

5.2 性能

5.2.1 環境

評価環境を**表 3** に示す．プロセッサは Linux に 1 コア，***AnT*** に 3 コア割り当てた．ただし，***AnT*** で測定に使用するのは 1 コア (m-カーネル) だけである．Linux カーネルには，Linux 3.0.8 を使用し，Linux と ***AnT*** は，同一のコンパイラ（gcc-4.4.7）を使用してコンパイルした．なお，測定結果は，10 回試行した場合の平均処理時間である．

表 3　評価環境

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）

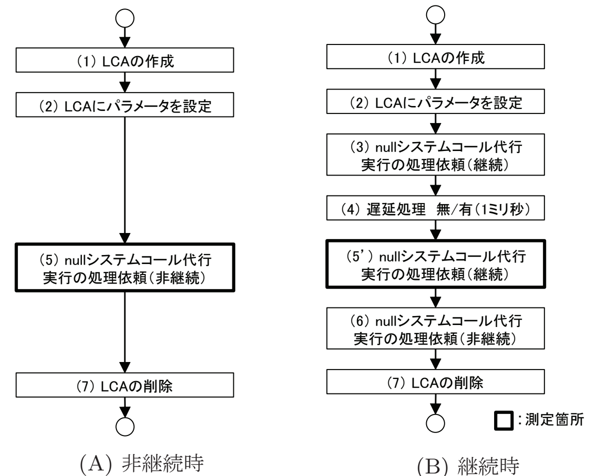

図 3　基本性能測定の処理流れ

5.2.2 基本性能

Linux システムコール代行実行の処理依頼 1 回におけるオーバヘッドを明らかにするため，何も処理を行わないシステムコール（以降，null システムコール）を作成した．これを使用して，代行処理を継続する場合（以降，継続時）と継続しない場合（以降，非継続時）の両方について，処理依頼にかかる処理時間を評価した．

基本性能測定の処理流れを**図 3** に示し，以下で説明する．なお，この処理流れは ***AnT*** 上の AP プロセスのものである．

（A）非継続時の基本性能測定では，(1) LCA を作成し，(2) LCA にパラメータを設定し，(5) null システムコール代行実行の処理依頼（非継続）をし，(7) LCA を削除する．一方，(B) 継続時の基本性能測定では，処理 (1)，(2)，および (7) は同じであるが，処理 (3) で Linux 側において事前に代行プロセスを生成させておき，(4) 遅延処理有の場合 1 ミリ秒だけ待ち，(5') null システムコール代行実行の処理依頼（継続）をし，(6) null システムコール代行実行の処理依頼（非継続）で代行プロセスを終了させる．遅延処理の有無により，継続時に連続して処理依頼を代行実行できる場合とできない（処理依頼の間に別の処理を行う）場合を比較する．なお，遅延処理の遅延時間を 1 ミリ秒としたのは，null システムコール 1 回の処理時間について，遅延時間を 10 ミリ秒，100 ミリ秒とした場合と 1 ミリ秒の場合がほぼ同様であったためである．

© 2015 Information Processing Society of Japan

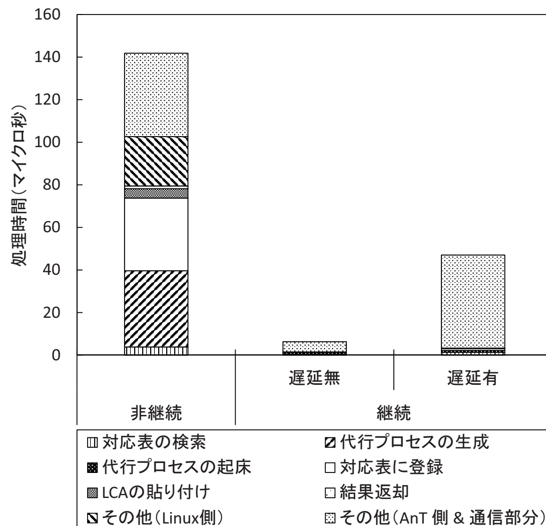

図 4 null システムコール 1 回の処理時間

表 4 null システムコールの部分処理の処理時間（マイクロ秒）

通番	処理内容	非継続時	継続時	
			遅延無	遅延有
(a)	対応表の検索	3.83	0.35	1.31
(b)	代行プロセスの生成	35.81	–	–
(c)	代行プロセスの起床	–	0.90	0.90
(d)	対応表に登録	34.16	–	–
(e)	LCA の貼り付け	4.37	–	–
(f)	結果返却	1.32	0.21	0.89
(g)	その他（Linux 側）	23.14	0.06	0.21
(h)	その他（***AnT*** 側 & 通信部分）	39.20	4.81	43.72
合計		141.83	6.33	47.03

null システムコール 1 回の処理時間を**図 4** に示す．また，null システムコールの部分処理の処理時間を**表 4** に示す．図 4 と表 4 から以下のことがわかる．

（1）非継続時の処理時間において，「(b) 代行プロセスの生成」と「(d) 対応表に登録」の全体に占める割合が大きい（約 49%（＝（35.81 ＋ 34.16）／ 141.83 ∗ 100））．これは，前者は fork()，後者は malloc() を内部で呼び出しており，これらの処理時間が長いためである．

（2）継続時の遅延処理無の処理時間は，三つの中で最も短い．これは，代行プロセスが起床待ち状態に移行した後，次の処理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，キャッシュが残ったままになっているためであると考えられる．

（3）継続時の遅延処理有の処理時間は，非継続時の処理時間に比べて短い．一方，継続時の遅延処理無の処理時間に比べて長い．継続時の遅延処理無の処理時間との差の大部分は，「(h) その他（***AnT*** 側 & 通信部分）」によるものである．この部分処理の処理時間の差は，約 38.91 マイクロ秒（＝（43.72 − 4.81））であり，全体の処理時間の差 40.70 マイクロ秒（＝ 47.03 − 6.33）の約 96%（＝ 38.91 / 40.70 ∗ 100）を占めている．また，この部分処理の処理内容は，***AnT*** 側の処理依頼，***AnT*** 側の結果受取，および Linux と ***AnT*** それぞれの IPI 送受信の処理である．この部分処理の長大化の原因は，遅延処理の間に 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，キャッシュミスが発生したためであると考えられる．

（4）継続時は，「(b) 代行プロセスの生成」，「(d) 対応表に登録」，および「(e) LCA の貼り付け」が不要である．一方，「(c) 代行プロセスの起床」が必要となる．結局，継続時は非継続時に比べて約 73.44 マイクロ秒（＝（35.81 ＋ 34.16 ＋ 4.37 − 0.90））だけ処理時間を削減できる．

以上のことから，以下の二つのことがいえる．

（1）1 回目の処理依頼は大きなオーバヘッドが生じる．

（2）2 回目以降（継続時）の処理依頼のオーバヘッドは小さいため，効率的に処理できる．

5.2.3 ファイル読み込み性能

アプリケーションを動作させた場合の性能を明らかにするため，ファイルの読み込み処理にかかる処理時間を Linux 上で実行した場合と ***AnT*** から処理依頼した場合で比較評価した．処理内容は，open()，read()，および close() を使用して，1,024 KB のファイルを 1 KB 単位で読み込むというものである．対象のファイルは同一とし，読み込むファイルサイズは，1 KB から 1,024 KB の範囲で 2 のべき乗ごとのサイズとした．なお，測定対象は，read() の発行にかかった合計処理時間とした．

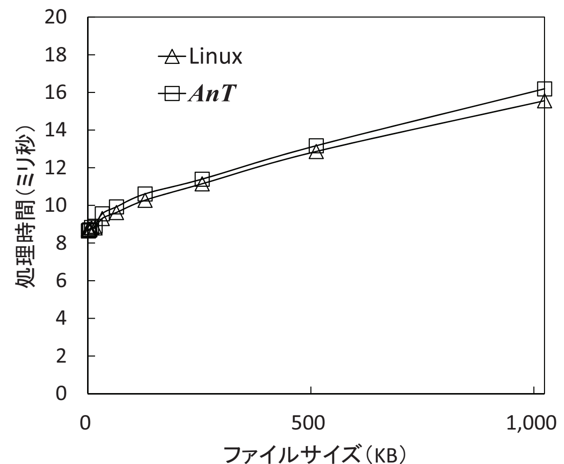

図 5 ファイル読み込みの処理時間

ファイル読み込みの処理時間を**図 5** に示す．図 5 より，以下のことがわかる．

（1）ファイルサイズが大きくなるにつれて，Linux 上で実行した場合と ***AnT*** から処理依頼した場合の処理時間の差は増加している．これは，ファイルサイズに比例して処理依頼の回数が増加し，結果としてオーバヘッドが増加するためである．

（2）ファイルサイズが 1〜16 KB の場合，***AnT*** から処理

依頼した場合の処理時間は，Linux 上で実行した場合の処理時間に比べ，ほぼ差がない．これは，処理依頼の回数が少なく，処理依頼によるオーバヘッドの影響が少ないためである．

(3) ファイルサイズが 32〜512 KB の場合，***AnT*** から処理依頼した場合の処理時間は，Linux 上で実行した場合の処理時間に比べ，約 0.2〜0.3 ミリ秒（約 2%〜3%）長い．1,024 KB の場合，約 0.6 ミリ秒（約 4%）長い．つまり，I/O 処理の処理時間が大きい場合，1,000 回程度の処理依頼を行ってもオーバヘッドは数%となる．

以上のことから，I/O 処理の処理時間が大きい場合，処理依頼のオーバヘッドは数%以内に収まるため，実際のアプリケーションにも十分耐えうる性能であるといえる．

6. おわりに

Linux と ***AnT*** の混載システムにおいて，両 OS の連携機構である ***AnT*** から Linux にシステムコール代行実行を処理依頼する方式について評価を報告した．

Linux システムコール代行実行の処理について，工数，代行実行処理の基本性能，およびファイル読み込み性能を評価した．連携機構を実現するための工数は，Linux では，Linux カーネルで 1 行，***AnT*** 連携デバイスと親/代行プロセスで約 400 行であった．一方，***AnT*** では，機能追加による追加/変更箇所は 9 ファイル 121 行であり，それぞれ全体の約 3.8%，約 0.8% であった．また，代行実行処理の基本性能では，null システムコール 1 回の処理依頼について，非継続時で約 141.83 マイクロ秒，継続時の遅延処理無で約 6.33 マイクロ秒，遅延処理有で約 47.03 マイクロ秒であることを示した．さらに，ファイル読み込み性能では，1,024 KB のファイルを 1 KB 単位で読み込んだ際，Linux で実行した場合と ***AnT*** から処理依頼した場合を比較すると，処理依頼時のオーバヘッドは約 4%であることを示した．

残された課題として，代行プロセスをある程度の数だけ事前生成した場合の評価がある．

謝辞　本研究の一部は，科学研究費補助金基盤研究（B）（課題番号：24300008）による．

参考文献

[1] 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).

[2] 福島有輝，山内利宏，谷口秀夫：Linux と ***AnT*** オペレーティングシステムの混載システム，情報処理学会研究報告，Vol.2014-OS-129，No.13，pp.1–8 (2014).

[3] Liedtke, J.: Toward real microkernels, *Communications of the ACM*, Vol.39, No.9, pp.70–77 (1996).

[4] Elphinstone, K. and Heiser, G.: From L3 to seL4 what have we learnt in 20 years of L4 microkernels?, *Proc. 24th ACM Symposium on Operating Systems Principles* (*SOSP'13*), pp.133–150 (2013).

[5] Tanenbaum, A.S., Herder, J.N. and Bos, H.: Can we make operating systems reliable and secure?, *IEEE Computer Magazine*, Vol.39, No.5, pp.44–51 (2006).

[6] Black, D.L., Golub, D.B., Julin, D.P., Rashid, R.F., Draves, R.P., Dean, R.W., Forin, A., Barrera, J., Tokuda, H., Malan, G. and Bohman, D.: Microkernel operating system architecture and mach, *Journal of Information Processing*, Vol.14, No.4, pp.442–453 (1992).

[7] 福島有輝，山内利宏，谷口秀夫：Linux と ***AnT*** オペレーティングシステムの混載と連携，情報処理学会研究報告，Vol.2014-DPS-161，No.12，pp.1–8 (2014).

[8] 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 (2013).

[9] 深沢　豪，佐藤未来子，長嶺精彦，坂本龍一，吉永一美，辻田祐一，堀　敦史，石川　裕，並木美太郎：マルチコア・メニーコア混在型計算機における演算コア側資源管理の代行方式，情報処理学会研究報告，Vol.2012-HPC-134，No.7，pp.1–8 (2012).

[10] 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 High Performance Computing* (*SBAC-PAD*), pp.211–218 (2012).

[11] 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).

[12] locmetrics.com: LocMetrics, available from 〈 http://www.locmetrics.com/ 〉 (accessed 2015-02-02).

© 2015 Information Processing Society of Japan