Vol.2014-DPS-161 No.12
Vol.2014-EIP-65 No.12
2014/9/19

Linuxと*AnT*オペレーティングシステムの混載と連携

福島 有輝[1]　山内 利宏[1]　谷口 秀夫[1]

概要：マルチコアプロセッサを有する計算機において，計算機全体のサービス提供を効率的に行うには，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動作が必要である．そこで，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．ここでは，Linux と ***AnT*** の混載システムにおいて，両 OS の連携により，***AnT*** 上の AP プロセスが Linux 機能を利用できる方式について述べる．具体的には，本方式の設計方針を述べ，設計における課題と対処について述べる．また，本方式の実現方式について述べる．

1. はじめに

LSI 技術の進歩は著しく，多くの機能が 1 つの LSI チップに実現できるようになっている．例えば，複数の演算実行ユニット（コア）をもつマルチコアプロセッサが登場している．コア数は，4 個，8 個と増加傾向にあり，低価格で複数コアを生かした分散処理が可能になっている．

マルチコアプロセッサを有する計算機において，計算機全体のサービス提供を効率的に行うには，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動作が必要である．

そこで，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，多くの入出力機器の利用が可能であり，かつ既存 OS として普及しており，多くのサービスが実現されている．一方，***AnT*** は，マイクロカーネル構造 [3]〜[6] を有する OS であり，適応性と堅牢性を特徴とする 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は，両 OS の連携が必要である．

ここでは，***AnT*** 上の応用プログラム（以降，AP）を実行する AP プロセスが Linux 機能を利用できる方式について述べる．具体的には，***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*** と Linux が連携しながら実行し，結果を AP プロセスに返却する方式を述べる．

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

2. Linuxと*AnT*の混載

2.1 *AnT* オペレーティングシステム

2.1.1 基本構造

AnT はマイクロカーネル構造を有する OS であり，現在，マルチコアプロセッサに対応している．以降，マルチコアプロセッサに対応した ***AnT*** を含めて ***AnT*** と呼ぶ．***AnT*** の基本構造を**図 1** に示す．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．OS サーバは，適応したシステムに必須なプログラム部分であり，通信制御処理や NIC ドライバ処理をプロセスとして提供する．サービスは，サービスを提供するプログラム部分である．

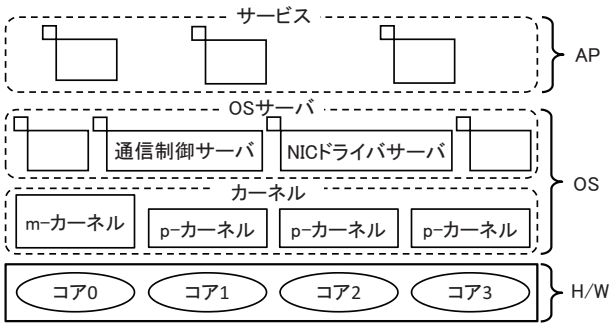

図 1　***AnT*** の基本構造

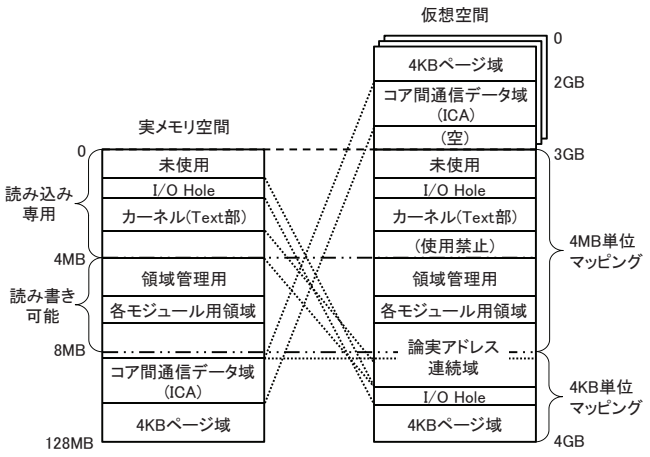

図 2 ***AnT*** の仮想空間の構成

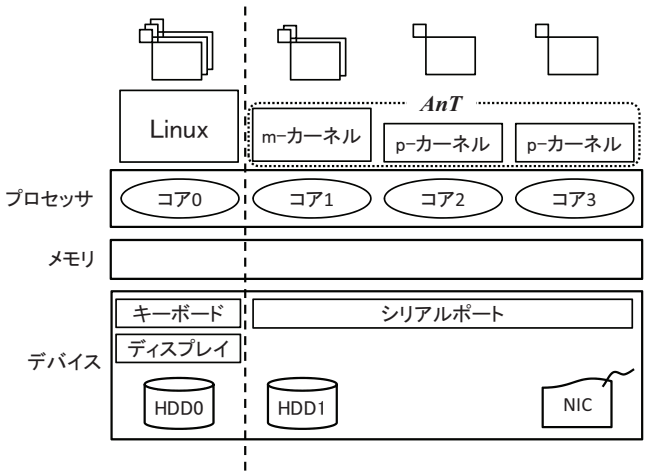

図 3 Linux と ***AnT*** の混載システムの構成例（4 コアの場合）

2.1.2 仮想空間の構成

AnT の仮想空間の構成を**図 2** に示す．仮想空間は，0 から 3GB をプロセス空間，3GB 以降をカーネル空間とし，多重仮想記憶を採用している．0 から 2GB のプロセス空間は，OS サーバやサービスのプログラムが使用する空間であり，4KB 単位でマッピングする．これに対し，2GB から 3GB のプロセス空間は，OS サーバとサービスのプログラムが相互にデータ通信に使用する空間であり，4KB 単位でマッピングする．この領域をコア間通信データ域 (以降，ICA : Inter-core Communication Area) と呼ぶ．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．3GB から 3GB+8MB のカーネル空間は，読み込み専用であるカーネルのテキスト部と読み書き可能なカーネルのデータ部をそれぞれ 4MB 単位でマッピングする．この領域は，実アドレスと仮想アドレスがすべて連続しており，実アドレスと仮想アドレスを容易に変換できる．以降の領域は，4KB 単位でマッピングする．

2.2 Linux と Mint

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．ここで, Mint は Linux を改造することで実現している．主な改造個所は，コアの分割，メモリの分割，デバイスの分割，および他 OS の起動処理の四つである．なお，Mint では，仮想化によらず各 OS を実計算機上で直接走行させているため，実計算機に近い性能で各 OS の走行を実現している．

2.3 Linux と *AnT* の混載システム

Linux と ***AnT*** の混載システムは，2.2 節の Mint を拡張して実現している．Linux と ***AnT*** の混載システムの構成例（4 コアの場合）を**図 3** に示す．図 3 では，各 OS にプロセッサ，メモリ，およびデバイス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）デバイス

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，およびデバイス構成の変更が可能である．

3. Linux と *AnT* の連携

3.1 設計方針

AnT 上の AP プロセスが Linux の入出力機能やファイル操作機能を利用できれば，***AnT*** 機能を最小化した状態で，AP プロセスが多くのサービスを実現できる．このため，Linux と ***AnT*** の連携により，Linux システムコール実行の機能を実現する．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．そこで，以下の処理とする．また，処理流れを**図 4** に示す．

（1）***AnT*** は，実メモリを確保し，***AnT*** の仮想空間に貼り付ける．

（2）***AnT*** は，その領域に依頼データを格納する．

（3）***AnT*** は, 実メモリの実アドレスを Linux に通知する．

（4）Linux は，受け取った実アドレスの実メモリを Linux の仮想空間に貼り付ける．

（5）Linux は，その領域の情報に基づき Linux システムコールを発行し，その領域に結果を格納する．

（6）Linux は，その領域の実メモリの実アドレスを ***AnT*** に通知する．

（7）Linux は，Linux の仮想空間から実メモリを剥がす．

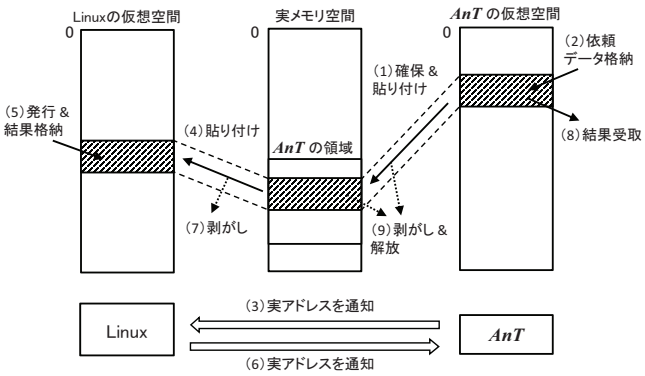

図 4 Linux と ***AnT*** 間の複写レスデータ授受の処理流れ

表 1 システムコール代行主体の比較

	プロセス	スレッド
利点	(1) 他プロセスに影響を与えない (2) ***AnT*** 上の実行主体に適合	生成が高速
欠点	生成が低速	(1) 資源共有による制限あり (2) ***AnT*** 上の他プロセスに影響あり

（8）***AnT*** は，受け取った実アドレスをもとに，対応する ***AnT*** の仮想空間から結果を受け取る．

（9）***AnT*** は，***AnT*** の仮想空間から実メモリを剥がし，実メモリを解放する．

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．それぞれの比較を**表 1** に示し，以下で説明する．

プロセスの場合，プロセス毎に独立したコンテキストを保持しているため，特定のプロセスで不具合が起きても他プロセスに影響しない．また，***AnT*** 上の実行主体はプロセスであるため，代行主体をプロセスにすると機能的に対応を取りやすい．一方，生成処理は低速である．

スレッドの場合，1 プロセス上で動作するため，資源共有により同時にオープン可能なファイル数に制限がかかる．また，Linux 上の他スレッドでエラーが生じた場合，***AnT*** 上の他プロセスにも影響が出てしまう．一方，生成処理は高速である．

ここでは，***AnT*** 上の実行主体がプロセスであるため，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

表 2 静的生成方式と動的生成方式の比較

	静的生成方式	動的生成方式
利点	処理依頼の実行が高速	両 OS で同期が不要
欠点	両 OS で同期が必要（***AnT*** 上のプロセスの生成と終了に同期）	処理依頼の実行が低速（処理依頼の度にプロセスを生成する必要あり）

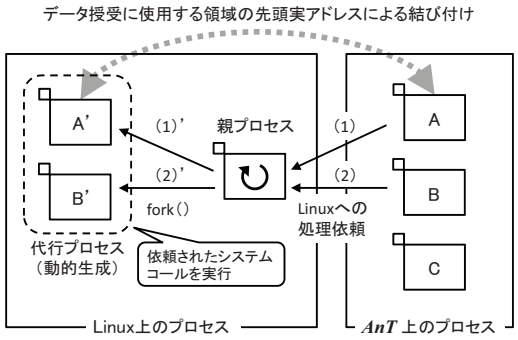

図 5 多重処理の実現の様子

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，Linux 上にプロセスを静的に事前に生成する．

（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，代行プロセスを動的に生成する．

上記二つの方式の比較を**表 2** に示し，以下で説明する．

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が高速である．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が低速という欠点がある．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．また，多重処理の実現の様子を**図 5** に示し，以下で説明する．図 5 では，Linux 上の親プロセスが ***AnT*** からの処理依頼を受け取る度に代行プロセスを生成している．図中の（1）と（2）に対応する処理は，それぞれ（1）’と（2）’である．なお，プロセス A とプロセス A’，プロセス B とプロセス B’ は，それぞれのデータ授受に使用する領域の先頭実アドレスにより結び付けられている．

以上より，Linux 上の Linux システムコール代行主体はプロセスとし，***AnT*** からの処理依頼は Linux 上の親プロセスが受け取り，処理依頼を受け取る度に親プロセスは代行プロセスを生成し，生成方式は動的生成方式とする．

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

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，代行プロセスの終了可否を制御可能となる．

なお，この効率化により，継続時には，3.3.1 項で述べた処理流れのうち（1），（4），（7），および（9）の 4 処理が不要となる．

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を使用する．以降，この領域を LCA（Linux system Call information Area）と呼ぶ．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 4KB である．なお，Linux と ***AnT*** 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．

LCA の構造を**表 3** に示し，以下で各エントリを説明する．cid は ***AnT*** 上の AP プロセスが動作しているコアの番号を表し，結果返却における IPI の送信先コアの特定に使用する．syscallno は Linux システムコール番号である．retval は Linux システムコールを実行した際の戻り値である．cflag は 3.3.3 項を実現するために使用するフラグである．継続する場合，この cflag を有効にしておく．args と

表 3 LCA の構造

エントリ名	内容
cid	結果返却先コア番号
syscallno	システムコール番号
retval	システムコールの戻り値
cflag	継続するか否かを表すフラグ
args	システムコールの引数
buffer	バッファ域

buffer は Linux システムコールの引数である．この二つのエントリを使用して Linux システムコールの引数を設定する．

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” 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．Linux システムコールの内容を授受するシステムコールの仕様を**表 4** に示す．

linux_call は，***AnT*** 側のシステムコールであり，Linux システムコール実行の処理を依頼する．これにより，***AnT*** は Linux に処理依頼を行うことができる．

ioctl（ANT_GET）は，Linux 側のシステムコールであり，Linux システムコール実行の処理依頼を受け取る．これにより，Linux は ***AnT*** から処理依頼を受け取ることができる．

ioctl（ANT_RET）は，Linux 側のシステムコールであり，Linux システムコール実行の結果を返却する．これにより，Linux は ***AnT*** に結果を返却できる．

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．

上記の 2 方式の比較を**表 5** に示し，以下で説明する．

表 5 シリアライズ方式とマッピング方式の比較

	シリアライズ方式	メモリマッピング方式
利点	両 OS 間で依存なし	(1) 複写レスのため高速 (2)Linux システムコールを識別する必要なし
欠点	(1) シリアライズのため低速 (2)Linux システムコールを識別して処理を行う必要あり	両 OS 間で依存あり

シリアライズ方式は，LCA を Linux の任意の仮想アドレスに貼り付けられるため，両 OS 間で依存がない．しかし，ポインタ引数をシリアライズするため，データ複写のオーバヘッドが生じる．また，ポインタ引数の位置は Linux システムコール毎に異なるため，Linux システムコールを識別して処理を行う必要がある．

メモリマッピング方式は，データ複写が発生しないため，高速である．また，両 OS で共通の仮想アドレスを使用するため，ポインタ引数に関連して Linux システムコールを識別する必要もない．しかし，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，シリアライズ方式を使用する．なお，データ複写のオーバヘッドとページ例外処理のオーバヘッドの比較が必要である．

表 4 Linux システムコールの内容を授受するシステムコールの仕様

形式	機能	備考
linux_call（raddr） unsigned int raddr;	LCA の先頭実アドレス raddr を指定して処理依頼し，結果受取まで待つ．LCA には，自分のコア番号，Linux システムコールの情報，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．
ioctl（fd, request） int fd; int request;	***AnT*** 連携デバイス “/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() は処理依頼受取を行う．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．	処理依頼受取までブロックされる．***AnT*** 連携デバイス “/dev/ant” を open() していることを前提とする．
ioctl（fd, request, raddr, cid） int fd; int request; unsigned int raddr; int cid;	***AnT*** 連携デバイス “/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() 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．	***AnT*** 連携デバイス “/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．

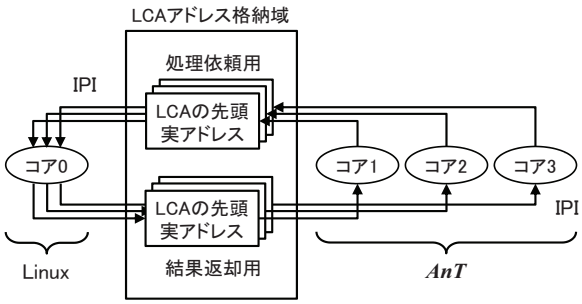

図 6 コア間通信方式

4.2 コア間通信方式

Linux と ***AnT*** のコア間通信では，IPI を使用する．IPI ではベクタ番号しか指定できないため，両 OS で LCA の先頭実アドレスを直接授受できない．そこで，LCA の先頭実アドレスを格納した両 OS で共有する領域（以降，LCA アドレス格納域）が必要となる．LCA アドレス格納域の使用方法を以下で説明する．

（1）処理依頼で ***AnT*** は LCA アドレス格納域に LCA の先頭実アドレス ***X*** を登録する．

（2）処理依頼受取で Linux は LCA アドレス格納域から ***X*** を取得する．

（3）結果返却で Linux は LCA アドレス格納域に ***X*** を登録する．

（4）結果受取で ***AnT*** は LCA アドレス格納域から ***X*** を取得する．

LCA アドレス格納域を使用した Linux と ***AnT*** のコア間通信方式を**図 6** に示す．Linux と ***AnT*** のコア間通信方式は，以下の二つの特徴をもつ．

（特徴 1）LCA の先頭実アドレスを格納する領域を処理依頼用と結果返却用で個別に用意
これにより，処理依取と結果返却を同時に行うことができる．

（特徴 2）LCA の先頭実アドレスを格納する領域をコア毎に用意
これにより，複数のコアから同時に処理依頼を行うことができる．なお，（特徴 2）は，処理依頼を行うコア毎に異なるベクタ番号を使用することで実現する．具体的には，図 3 の構成例では，必要なベクタ数は 3 となる．

4.3 処理流れ

Linux システムコール実行の処理流れを**図 7** に示し，以下で説明する．

（1）処理依頼（***AnT*** 上の AP プロセス）
AnT 上の AP プロセスは，ICA を使用して LCA を確保する．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 の先頭実アドレスを指定して処理依頼システムコールを発行する．なお，***AnT*** 上の AP プロセスは結果受取までブロックされる．

（2）IPI（***AnT*** → Linux）
AnT は，Linux が動作するコアに対して，IPI を送信する．なお，LCA の先頭実アドレスは LCA アドレス格納域に格納されている．

（3）処理依頼受取（Linux 上の親プロセス）
Linux 上の親プロセスは，処理依頼受取システムコールにより，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）代行プロセスを生成（Linux 上の親プロセス）
Linux 上の親プロセスは，多重処理を実現するため，代行プロセスを生成する．なお，継続の場合，LCA の先頭実アドレスに対応する代行プロセスを起床する．

（5）LCA の貼り付け（Linux 上の代行プロセス）
Linux 上の代行プロセスは，mmap() により，LCA を自身の仮想空間に貼り付ける．なお，継続の場合，すでに貼り付いているため，この処理は行わない．

（6）Linux システムコール実行（Linux 上の代行プロセス）

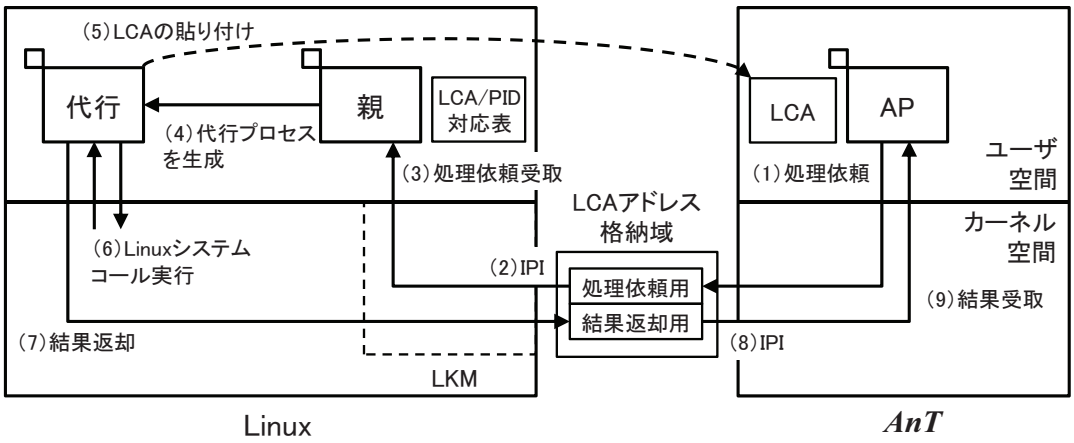

図 7 Linux システムコール実行の処理流れ

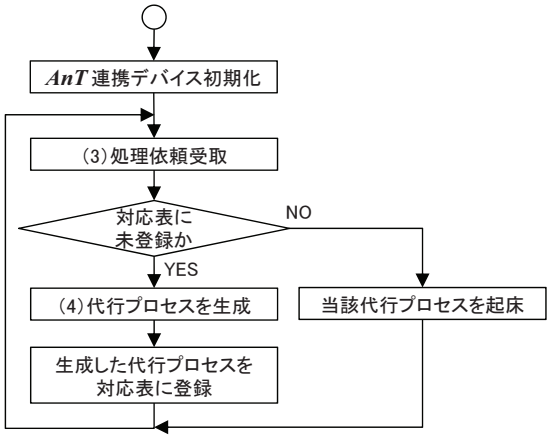

図 8 親プロセスの処理流れ

Linux 上の代行プロセスは，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（Linux 上の代行プロセス）

Linux 上の代行プロセスは，結果返却システムコールを発行し，***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，継続の場合，親プロセスからの起床待ちとなる．

（8）IPI（Linux → ***AnT***）

Linux は，***AnT*** が動作するコアに対して，IPI を送信する．なお，LCA の先頭実アドレスは LCA アドレス格納域に格納されている．

（9）結果受取（***AnT*** 上の AP プロセス）

AnT は，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．起床されたプロセスは，結果を受け取り，処理依頼を完了する．

Linux システムコール実行における親プロセスの処理流れを**図 8** に示し，以下で説明する．親プロセスは，***AnT*** 連携デバイスを open() し，初期化する．次に，処理依頼受取（図 7 の（3））を行った後，受け取った LCA の先頭実アドレスが対応表に未登録か否かを確認し，未登録であれば，代行プロセスを生成し（図 7 の（4）），生成した代行プロセスを対応表に登録する．一方，登録されている場合，LCA の先頭実アドレスに対応する代行プロセスを起床する．以降，再度（3）の処理に戻る．

Linux システムコール実行における代行プロセスの処理流れを**図 9** に示し，以下で説明する．代行プロセスは，LCA の貼り付け（図 7 の（5）），Linux システムコール実行（図 7 の（6）），および結果返却（図 7 の（7））を行った後，代行処理を継続する場合，親プロセスからの起床待ちとなる．一方，代行処理を継続しない場合，（5）で貼り付けた LCA を剥がし，終了する．なお，親プロセスは代行プロセスの終了通知を受け取ると，当該代行プロセスを対応表から削除する．

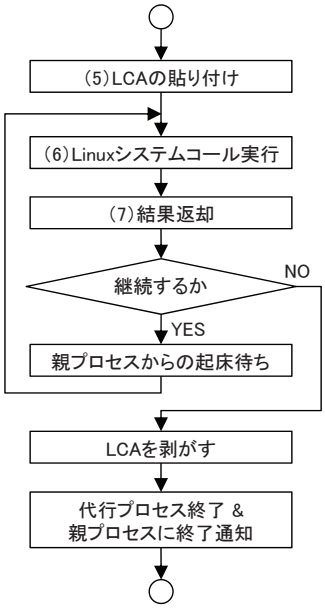

図 9 代行プロセスの処理流れ

5. 関連研究

組み込みシステムを対象として，Linux とリアルタイム OS（以降，RTOS）を混載したシステムが提案されている [7][8]．これらは，軽量な仮想マシンモニタを使用して実現されている．混載の目的は，Linux によるシステムの汎用性の向上，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が挙げられる．なお，文献 [7][8] は，混載した時点で目的を達成している．このため，本研究のような連携はない．

マルチコア・メニーコア混在型計算機を対象として，Linux と軽量 OS を混載し，連携するシステムが提案されている [9][10]．これらは，軽量 OS から Linux システムコールを利用できる方式を採用している．文献 [9] の特徴は，Linux と軽量 OS で同じ仮想アドレスを使用して，Linux システムコールを代行実行することである．これは 4.1.2 項のメモリマッピング方式である．これにより，Linux システムコールを識別する必要がなく，Linux のバージョンにも依存しないため，実装と保守が容易である．また，シリアライズ方式に比べ高速である．文献 [10] の特徴は，Linux 上のプロセスと軽量 OS 上のプロセスが一対一対応していることである．これは 3.3.2 項の静的生成方式である．これにより，代行プロセスの生成が処理要求時に不要であるため，高速である．また，Linux 上のプロセスはシステムコールの多重実行を実現するため，ワーカスレッドを作成している．文献 [9][10] では，独立化よりも高速化を重視し

ている．一方，本研究では，高速化よりも独立化を重視している．

6. おわりに

Linux と ***AnT*** の混載システムにおいて，***AnT*** 上の AP プロセスが Linux システムコールを利用できる方式について述べた．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というものである．

本方式の設計方針は，高速化，多重化，効率化，および独立化と局所化の四つである．これらの方針を満たすため，課題としてデータ転送方式，代行実行方式，および代行処理継続方式の三つを挙げ，それぞれについて対処を示した．本方式の特徴としては，***AnT*** から処理依頼を受け取る度に，Linux で代行プロセスを生成し，代行プロセスが Linux システムコールを代行実行することが挙げられる．また，システムコールの実行を継続する場合，代行プロセスを終了させないことが挙げられる．本方式の利点は，各 OS の独立性を保っていることである．

実現方式では，Linux システムコールにポインタ引数が含まれる場合の対処，コア間通信における構造について述べ，最後に本方式の処理流れを示した．

残された課題として，本方式の実現と評価がある．

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] Mitake, H., Lin, T., Shimada, H., Hinebuchi, Y., Li, N. and Nakajima, T.: Towards Co-existing of Linux and Real-Time OSes, *Linux Symposium*, pp.55–68, (2011).
[8] 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 (2010)．
[9] 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)．
[10] 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)．