Vol.2012-OS-122 No.5
2012/8/1

マルチコア *AnT* における処理分散機能

佐古田 健志[1]　枡田 圭祐[1]　井上 喜弘[1]　谷口 秀夫[1]

概要：マルチコア ***AnT*** は，マイクロカーネル構造を有し，マルチコアプロセッサ上で動作する．マルチコアプロセッサにおいて，OS サーバを複数のコア上に分散動作させることで，OS 機能の処理分散を実現できる．本稿では，マルチコア ***AnT*** の OS 機能の処理分散を実現する際の課題として，プロセスを分散する方式と分散したプロセスが別コアの OS 機能を利用する方式について課題と対処を示し，実現方式を述べる．また，評価として，マルチコア ***AnT*** に実現した OS 機能の処理分散機能の性能について報告する．

Process Distribution Mechanism for Multi-core *AnT*

Takeshi SAKODA[1]　Keisuke MASUDA[1]　Yoshihiro INOUE[1]　Hideo TANIGUCHI[1]

Abstract: Multi-core ***AnT*** is an operating system based on microkernel architecture. Multi-core ***AnT*** works on a multi-core processor. In the multi-core processor, process distribution for OS functions is realized by distributed OS server execution on multiple cores. This paper discusses the methods to distribute process and to use OS functions that work on different core, describes implementation of the proposed methods. In addition, this paper shows the result of evaluations for performance of the process distribution mechanism based on Multi-core ***AnT***.

1. はじめに

近年，プロセッサ性能の向上や入出力ハードウェアの進歩は著しい．更に，通信路の伝送速度も向上している．これにより，様々な場面で計算機が利用され，提供するサービス種別も増大している．一方，計算機の利用形態の多様化により，プログラムの相互影響による不具合の発生，およびネットワークを介した攻撃の脅威は大きな問題になっている．このような背景から，計算機の多様な利用を支える高い適応性，および不具合や攻撃に耐え得る高い堅牢性を有するオペレーティングシステム（以降，OS）が必要になっている．そこで，我々は，高い適応性と堅牢性をあわせ持つ ***AnT*** オペレーティングシステム [1]（An operating system with adaptability and toughness）（以降，***AnT***）を研究開発している．***AnT*** は，高い適応性と堅牢性を実現するためにマイクロカーネル構造 [2]~[4] を採用している．マイクロカーネル構造は，スケジューラ機能やメモリ管理機能といった最小限の OS 機能をカーネルとして実現し，ファイル管理機能やディスクドライバといった大半の OS 機能をプロセス（以降，OS サーバ）として実現するプログラム構造である．

トランザクション処理に代表される処理は，大半が OS 機能の利用である．そこで，OS 機能の処理分散を可能にする第一歩として，文献 [5] の ***AnT***（以降，マルチコア ***AnT***）は，マルチコアプロセッサ [6] 上での動作を可能にした．

本稿では，マルチコア ***AnT*** の OS 機能の処理分散を実現する際の課題として，プロセスを分散する方式と分散したプロセスが別コアの OS 機能を利用する方式について課題と対処を示し，実現方式を述べる．また，実現した処理分散機能について，基本評価，および OS 機能を分散した場合の評価について結果を報告する．

2. マルチコア *AnT*

2.1 基本構造

マルチコア ***AnT*** は，マイクロカーネル構造を有する OS である．以下で，マルチコア ***AnT*** の基本構造と仮想空間

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

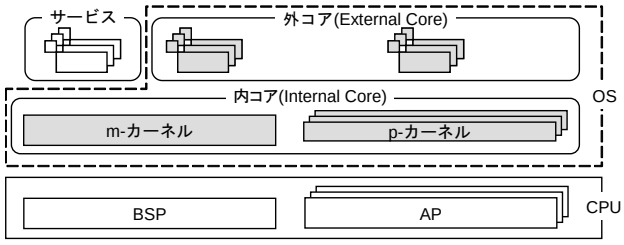

図 1 マルチコア ***AnT*** の基本構造

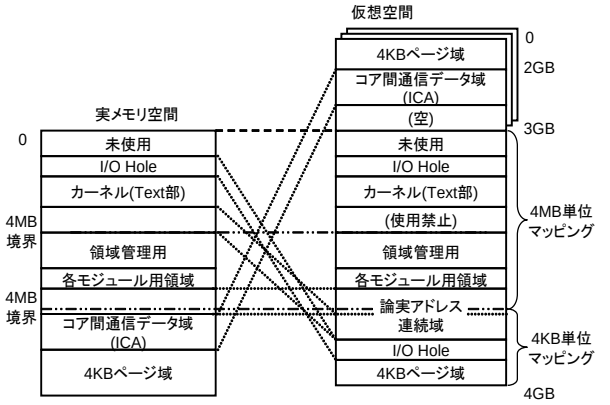

図 2 仮想空間の構成

の構成について述べる．

マルチコア ***AnT*** の基本構造を**図 1** に示す．OS は，内コアとプロセス (OS サーバ) として動作する外コアからなる．内コアは，m-カーネル（最初に起動するカーネル）と p-カーネル（m-カーネルにより起動されるカーネル）の二種類のカーネルからなり，カーネルごとに異なった機能を有する．m-カーネルは，スケジューラ機能やメモリ管理機能といったカーネルに必要な全機能を有する．一方，p-カーネルは，機能を例外・割り込み機能，サーバプログラム間通信機能，およびスケジューラ機能に絞り，軽量化を図っている．内コアは，最小のシステムの動作を保証するプログラム部分である．主な機能として，スケジューラ機能やメモリ管理機能がある．外コアは，適応したシステムに必須なプログラム部分であり，例えば，ファイル管理機能やディスクドライバを OS サーバとして提供する．サービスは，サービスを提供するプログラム部分である．

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，多重仮想記憶である．0 から 2GB のプロセス空間は，外コアやサービスのプログラムが利用する空間であり，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，内コア，外コア，およびサービスのプログラムが相互のデータ通信に利用する空間である．この領域をコア間通信データ域（以降，ICA : Inter-core Communication Area）と呼ぶ．

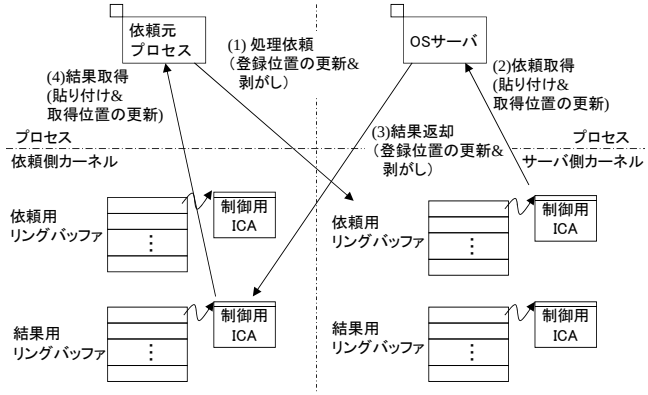

図 3 マルチコア ***AnT*** のサーバプログラム間通信機構の処理流れ

2.2 サーバプログラム間通信機構

AnT は，高速なサーバプログラム間通信機構 [7] を有している．さらに，マルチコア ***AnT*** では，マルチコアでのプログラム間通信機構を高速化している [5] ．具体的には，OS サーバ間の通信相手を制限し，通信時のキュー操作を２排他制御にしている．さらに，リングバッファを用いた制御機構により，排他制御を行なわない制御用 ICA の貼り替えを実現している．また，ICA の授受方式を改善し，各プロセスの動作するコア上のカーネルがサーバプログラム間通信時における ICA の貼り替えを行なうことにした．

以上に基づくサーバプログラム間通信機構の処理流れを**図 3** に示し，以下で説明する．

(1) 依頼元プロセスが処理依頼を行なうと，依頼側カーネルは，まず，OS サーバの依頼用リングバッファの登録位置が示すエントリに制御用 ICA のアドレスと保護情報を登録し，登録位置を更新する．次に，依頼元プロセスから制御用 ICA を剥がす．

(2) サーバ側カーネルは，依頼用リングバッファの取得位置が示すエントリに登録された制御用 ICA のアドレスと保護情報をもとに，OS サーバに制御用 ICA を貼り付け，取得位置を更新する．

(3) OS サーバが結果返却を行なうと，サーバ側カーネルは，まず，依頼元プロセスの結果用リングバッファの登録位置が示すエントリに制御用 ICA のアドレスと保護情報を登録し，登録位置を更新する．次に，OS サーバから制御用 ICA を剥がす．

(4) 依頼側カーネルは，結果用リングバッファの取得位置が示すエントリに登録された制御用 ICA のアドレスと保護情報をもとに，依頼元プロセスに制御用 ICA を貼り付け，取得位置を更新する．

3. 処理分散機能

3.1 考え方

マルチコア ***AnT*** では，大半の OS 機能をプロセス（OS サーバ）として実現している．このため，プロセスの分散により，処理を分散できる．プロセスの分散を実現するには，以下に示す２つの課題がある．

（課題１）プロセスを分散する方式

（課題２）分散したプロセスが別コアの OS 機能を利用する方式

プロセスを分散する方式を確立する必要がある．また，p-カーネルの機能は軽量化のために絞られているため，p-カーネル上へ分散したプロセスが m-カーネルの機能を利用する方式を確立する必要がある．

3.2 プロセス分散方式

3.2.1 基本方式

プロセスを分散する方式として，以下の案がある．

（案１）プロセス生成時に分散

（案２）プロセス動作中に移動で分散

（案１）では，一度生成した OS サーバは，動作するコアを変更できないため，処理途中での動的な負荷分散は難しい．どうしても必要になった場合は，分散対象の OS サーバを終了させ，再度，同一の OS サーバを分散先のコアに生成する必要がある．

（案２）は，m-カーネルが自コアへプロセスを生成後，別コアへ分散する方法である．（案２）では，一部のコアに OS サーバが集中した場合，OS サーバを終了させることなく，動的に分散することが可能である．

OS 機能の再分散を容易にできれば，処理の偏りを抑制でき，多くの時点において負荷の分散を行なえる．（案１）では，OS サーバの分散を実行するたびにプロセスの終了と生成を繰り返す．一方，（案２）では，OS サーバの分散時にプロセスの終了と生成を実行しないため，プロセスの終了と生成によるオーバヘッドなしにプロセスの分散を実現できる．したがって，（案２）により，プロセスの移動（プロセスマイグレーション）[8] を実現する．

マルチコア ***AnT*** では，マルチコア環境に適したプロセス移動（プロセスマイグレーション）機能を実現する．プロセス移動機能の実現において，マルチコア環境とネットワーク環境の大きな違いは，メモリを共有できるか否かである．また，マルチコア ***AnT*** は，コアの独立性を高める設計方針により実現している [5]．したがって，プロセス移譲後のコア間の連携は疎にする．

以降では，「プロセスを移動させる機能」をプロセス移譲機能と名づける．

3.2.2 プロセス移譲機能

マルチコア ***AnT*** におけるプロセス移譲機能は，移譲を実行するコア（以降，移譲元コア）上で走行しているプロセスを任意のコア（以降，移譲先コア）に移譲する機能である．このため，プロセス移譲機能の実現には，移譲対象プロセスを移譲元コア上で動作するカーネル（以降，移譲元カーネル）のスケジュールキュー（以降，移譲元スケジュールキュー）から削除し，移譲先コア上で動作するカーネル（以降，移譲先カーネル）のスケジュールキュー（以降，移譲先スケジュールキュー）に接続する必要がある．プロセス移譲機能を実現するには，以下に示す２つの課題がある．

（課題Ａ）移譲可能なプロセスの処理状態の明確化

（課題Ｂ）移譲方式の確立

マルチコア ***AnT*** において，プロセスは，依頼や結果を受け取る専用のキューを持つ．このキューについて，以下の２つの状態がある．

（１）依頼や結果用のキューが空の状態

（２）依頼や結果用のキューが空でない状態

依頼や結果用のキューが空の状態である場合，プロセスは，処理を行なっていない．一方，依頼や結果用のキューが空でない状態である場合，プロセスは，処理を行なっている途中である．今回，プロセス移譲処理の複雑化を避けるため，（１）のキューが空の状態を前提とし，（課題Ａ）に対処する．

移譲方式の確立は，以下の２つの方法により実現される．

（ａ）移譲可能なプロセス状態の判定と状態保存方法

（ｂ）移譲処理方法

まず，プロセスは，以下の３つの状態を有する．

（１）RUN 状態

（２）READY 状態

（３）WAIT 状態

マルチコア ***AnT*** では，これら３つの状態のうち，RUN 状態にはスケジュールキューが存在せず，READY 状態と WAIT 状態は操作対象であるスケジュールキューが異なる．また，プロセスは，生成されてから終了するまでに状態の遷移を繰り返すため，プロセス移譲時のプロセスの状態を予測できない．したがって，プロセス移譲機能の処理内容をプロセス移譲時のプロセスの状態によって変更する必要がある．対処として，プロセス移譲機能の実行時に移譲対象プロセスの状態を調査し，状態によって実行する処理を変更する．

（１）RUN 状態の場合，プロセスの状態を READY 状態に変更し，移譲前に動作中のコンテキストを保存する

（２）READY 状態では，移譲元スケジュールキューの内，READY キューから移譲対象プロセスを削除する

（３）WAIT 状態では，移譲元スケジュールキューの内，WAIT キューから移譲対象プロセスを削除する

移譲対象プロセスの状態に応じて上記の３つの処理から適

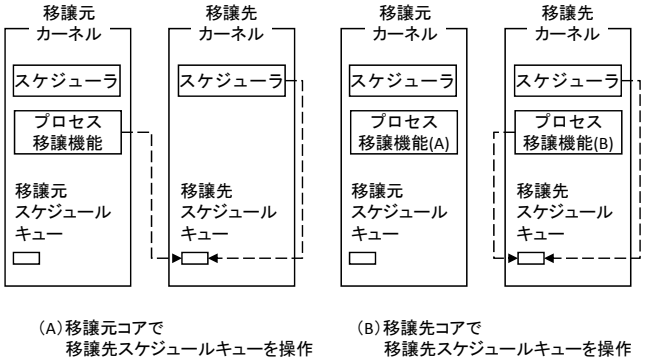

図 4 スケジュールキュー操作の様子

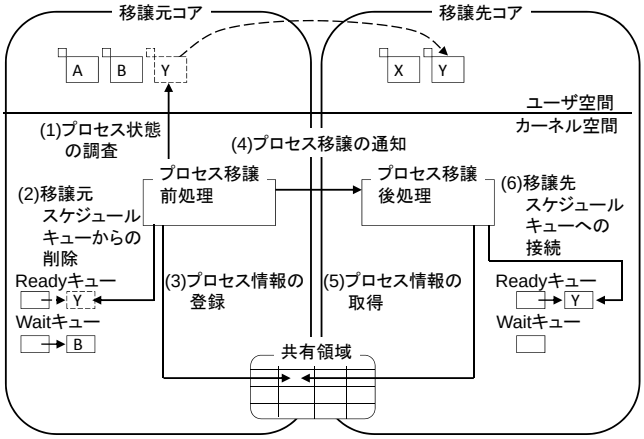

図 5 プロセス移譲処理の流れ

切な処理を選択することで，移譲可能なプロセス状態の判定と状態保存方法を確立する．

また，プロセス移譲機能は，移譲先スケジュールキューの操作を必要とする．この処理を移譲元コアで実行した場合，移譲元コアによる移譲先スケジュールキューの操作と移譲先コアのスケジューラ機能による移譲先スケジュールキューの操作が同時に発生する可能性がある．対処として，以下の案がある．

（案１）移譲元コアで移譲先スケジュールキューを操作
（案２）移譲先コアで移譲先スケジュールキューを操作

（案１）を用いたスケジュールキュー操作の様子を**図 4** (A) に示す．（案１）では，同一のスケジュールキューを複数のカーネルが同時に操作する可能性があるため，各カーネルのスケジュールキュー操作に対して排他制御を利用し，複数のカーネルから同時にスケジュールキューを操作できないようにする必要がある．しかし，スケジュールキューの操作は，各カーネルのスケジューラ機能により頻繁に行なわれる．したがって，排他制御のオーバヘッドが増大し，性能が著しく低下すると考えられる．

次に，（案２）を用いたスケジュールキュー操作の様子を図 4（B）に示す．（案２）では，プロセス移譲処理を分割し，移譲元コアと移譲先コアで分担実行し，移譲元コアと移譲先コアのスケジュールキュー操作を各カーネルで実行する．この際，移譲先コアで同時実行可能な機能は高々１つであるため，移譲先コアにおいて，プロセス移譲機能によるスケジュールキュー操作とスケジューラ機能によるスケジュールキュー操作は同時に実行されず，スケジュールキュー操作に対する排他制御を不要にできる．しかし，プロセス移譲処理を分担実行するため，移譲先コアに移譲対象プロセスのプロセス情報を通知する必要がある．

（案１）を用いた対処の場合，既存のスケジューラ機能を改造する必要があり，改造工数が多い．さらに，排他制御を用いるため，性能が著しく低下する．一方，（案２）を用いた対処の場合，既存のスケジューラ機能を改造する必要はなく，改造工数が少ない．したがって，改造工数の少ない（案２）を採用し移譲処理方法を確立する．

マルチコア ***AnT*** では，移譲元コアで実行する処理をプロセス移譲前処理，移譲先コアで実行する処理をプロセス移譲後処理として実現している．また，（案２）におけるプロセス移譲を通知する方法として，IPI(Inter-Processor Interrupt) を利用する．IPI は，通知先コアへ割り込み番号のみを通知する機構であるため，移譲先コアへ移譲対象プロセスのプロセス情報を通知できない．そこで，移譲元コアと移譲先コアの双方が読み書き可能な領域 (以降，共有領域) を利用し，移譲先コアへ移譲対象プロセスのプロセス情報を通知する．

3.2.3 処理の流れ

実現したプロセス移譲処理の流れを**図 5** に示し，以下で説明する．

（１）プロセス状態の調査では，移譲可能なプロセス状態の判定を行なう．
（２）移譲元スケジュールキューからの削除では，移譲対象プロセスが RUN 状態の場合，プロセスの状態を READY 状態に変更し，プロセスのコンテキストを保存する．また，移譲対象プロセスが READY 状態，または WAIT 状態の場合，それぞれのスケジュールキューからプロセスを削除する．
（３）プロセス情報の登録では，移譲対象プロセスのプロセス情報を共有領域へ書き込む．
（４）プロセス移譲の通知では，IPI を用いて移譲先コアへプロセス移譲を通知する．
（５）プロセス情報の取得では，移譲対象プロセスのプロセス情報を共有領域から読み込む．
（６）移譲先スケジュールキューへの接続では，（２）の処理により，全てのプロセスの状態は READY 状態か WAIT 状態のどちらかに変更されているため，該当のスケジュールキューへプロセスを接続する．

また，プロセス情報の通知に利用する共有領域の構造を

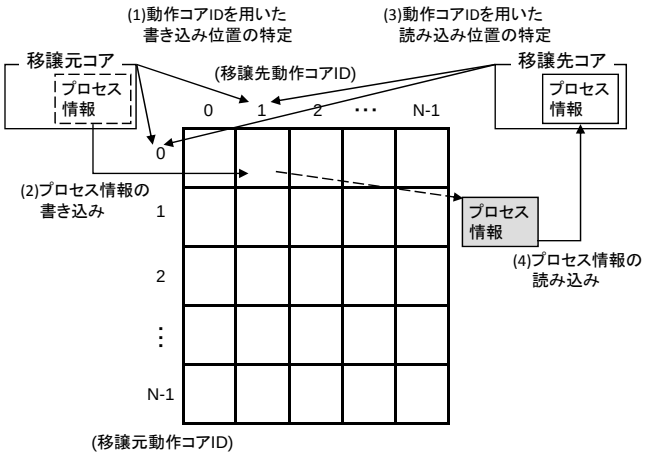

図 6 共有領域

図 6 に示す． 共有領域は，全コアで読み書き可能な領域とする．マルチコア ***AnT*** では，複数コアによるプロセス情報の同時書き込みを防ぐため，N × N 配列の形で共有領域を実現している（N はコア数）．共有領域へのプロセス情報の登録と取得には，移譲元カーネルの情報と移譲先カーネルの情報 (以降，動作コア ID) を利用する．移譲元コアによる共有領域への登録では，移譲元カーネルの動作コア ID と移譲先カーネルの動作コア ID を用いて共有領域内の書き込み位置を特定し，プロセス情報を書き込む．移譲先コアによる共有領域からの取得では，移譲元カーネルの動作コア ID と移譲先カーネルの動作コア ID を用いて共有領域内の読み込み位置を特定し，プロセス情報を読み込む．

3.3 別コア OS 機能利用方式

3.3.1 基本方式

分散したプロセスが別コアの OS 機能を利用する方式として，以下の案がある．

（案１）p-カーネルへの機能追加
（案２）m-カーネルへの処理依頼
（案３）m-カーネル専用機能の共用化

（案１）では，m-カーネル専用機能と同等の処理で構成される p-カーネル専用機能を p-カーネルに追加する．p-カーネル専用機能は，m-カーネル専用機能と同等の処理で構成されるため，m-カーネル専用機能と同等の処理時間でカーネル機能を利用できる．しかし，m-カーネルと同等の処理を p-カーネルに実現するため，p-カーネルのカーネルサイズが増大する．

次に，（案２）では，m-カーネルへの処理依頼を行ない，m-カーネルが実行した m-カーネル専用機能の結果を受け取る機能を新規に p-カーネルに実現する．また，p-カーネルからの処理依頼を受け取り，結果返却する機能を m-カーネル専用機能に追加する．このため，m-カーネルへの処理依頼機能は，処理依頼機能と結果返却機能を実現する必要がある．処理依頼機能は，処理を依頼し，結果をプロセスへ返却するのみであり，改造工数は少ない．また，結果返却機能は，依頼を受け取り，結果を返却するのみであり，改造工数は少ない．このため，p-カーネル専用機能を追加する（案１）に比べ，カーネルサイズを抑制できる．しかし，m-カーネルへの処理依頼時にコア間通信が発生するため，（案１）に比べ処理時間が増大する．

表 1 m-カーネル専用処理の利用案の比較

	（案１）機能追加	（案２）処理依頼	（案３）共有化
オーバヘッド	変化なし	大きい	大きい（削減可）
改造工数	多い	少ない	多い
カーネルサイズの増加量	非常に多い	少ない	多い

（案３）では，m-カーネル専用機能を共用機能に改造し，p-カーネルからも利用可能にする．しかし，メモリ資源操作といった m-カーネル専用機能は，資源操作を行なう機能であるため，複数のカーネルから同時に実行されると問題になる．したがって，m-カーネル専用機能を共有機能として p-カーネルからも利用可能とするためには，排他制御を用いて，複数のカーネルから同時に実行されない機能として実現する必要がある．（案３）では，排他制御を用いるため，排他制御処理のオーバヘッドが増加し，（案１）に比べ処理時間が増大する．

上記３つの案の比較を**表 1** に示す．表 1 より，（案１）は，マルチコア ***AnT*** 全体のカーネルサイズを抑制する設計方針に反することがわかる．このため，（案１）は採用しない．次に，（案２）は，改造工数とカーネルサイズを抑制できるが，オーバヘッドが大きいことがわかる．最後に，（案３）は，改造工数は多いものの，オーバヘッドを抑制しながら，（案１）よりもカーネルサイズを抑制できることがわかる．したがって，別コアの OS 機能を利用する方式として，カーネル機能の処理内容によって，（案２）と（案３）を使い分ける方式を実現する．以降では，別コア OS 機能利用方式として m-カーネルへの処理依頼と m-カーネル専用機能の共有化を使い分ける機能をカーネルコール処理分担機能と名づける．

3.3.2 m-カーネルへの処理依頼

m-カーネルへの処理依頼の流れを**図 7** に示し，以下で説明する．

（１）移譲前の m-カーネル上のプロセスにおける処理流れ
（A）プロセスは，カーネルコールを発行し，m-カーネルに処理を依頼する．
（B）m-カーネルは，依頼された処理を実行し，結果をプロセスへ返却する．
（２）移譲後の p-カーネル上のプロセスにおける処理流れ
（A’）プロセスは，カーネルコールを発行し，p-カーネルに処理を依頼する．
（C）依頼されたカーネルコール処理が m-カーネル専用機

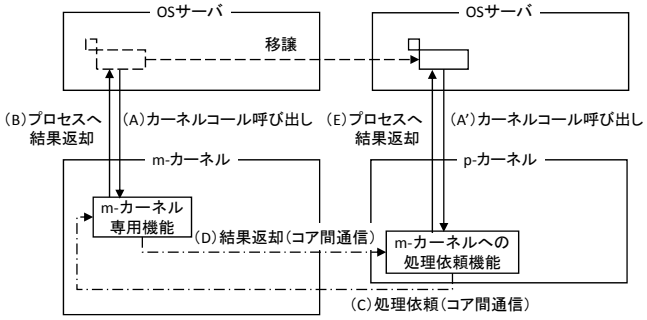

図 7 m-カーネルへの処理依頼の流れ

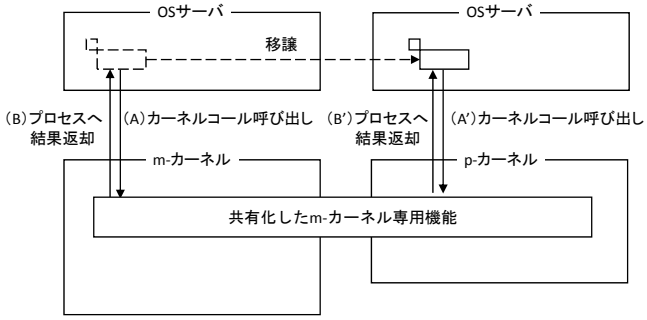

図 8 共有化した m-カーネル専用機能の流れ

能を利用する場合，p-カーネルは，m-カーネルに対し処理を依頼する．

（D）m-カーネルは，依頼された処理を実行し，結果を p-カーネルへ返却する．

（E）p-カーネルは，m-カーネルから返却された結果をプロセスへ返却する．

（1）と（2）の処理流れからわかるように，移譲後の p-カーネル上で動作するプロセスは，m-カーネルへの処理依頼のため，コア間通信が発生する．マルチコア ***AnT*** におけるコア間通信は，プロセス切り替えと同等の処理時間が必要になるため，オーバヘッドは大きい．また，コア間通信を用いて処理依頼を行なう p-カーネルは，依頼した処理が完了するまで他の処理を実行できない．このため，m-カーネルへの処理依頼は，m-カーネルと p-カーネルの 2 つのカーネルにおいて処理を並列に実行できなくなり，処理の並列性は低減する．さらに，m-カーネルにおいても p-カーネルから依頼された処理を実行するため，他の処理を実行できなくなる．

また，m-カーネルから p-カーネルへの結果返却機能の実現のため，m-カーネル専用機能の改造が必要になる．しかし，結果返却機能は，依頼を受け取り，結果を返却するのみであり，改造工数とオーバヘッドは少なく，m-カーネル専用機能の処理時間に影響を及ぼさない．このため，m-カーネル上のプロセスが m-カーネル専用機能を使用する場合，高速に実行できる．

3.3.3 m-カーネル専用機能の共有化

共有化した m-カーネル専用機能の流れを**図 8** に示し，以下で説明する．

（1）移譲前の m-カーネル上のプロセスにおける処理流れ

（A）プロセスは，カーネルコールを発行し，m-カーネルに処理を依頼する．

（B）m-カーネルは，依頼された処理を実行し，結果をプロセスへ返却する．

（2）移譲後の p-カーネル上のプロセスにおける処理流れ

（A'）プロセスは，カーネルコールを発行し，p-カーネルに処理を依頼する．

表 2 m-カーネルへの処理依頼と m-カーネル専用機能の共有化の特徴

	処理依頼	共有化
オーバヘッド	大きい	大きい（削減可）
処理の並列性	低い	高い
m-カーネル専用処理の処理時間	微増	増加
m-カーネルの改造	一部必要	必要
改造工数	少ない	多い

（B'）p-カーネルは，依頼された処理を実行し，結果をプロセスへ返却する．

（1）と（2）の処理流れからわかるように，m-カーネル専用機能の共有化は，p-カーネルからも m-カーネル専用機能を利用可能にする．共有化するにあたり，m-カーネル専用機能に排他制御を用いる必要がある．この際，排他制御としてジャイアントロックを用いると，カーネルレベルでの並列性が失われるため，マルチコア ***AnT*** におけるコアの独立性がなくなり，処理分散機能が有効に働かない．したがって，マルチコア ***AnT*** において排他制御を用いる場合は，細粒度ロックを用いる．細粒度ロックを用いると，オーバヘッドは増大するものの，ロックの競合は低減するため，処理の並列性は高くなる．排他制御を用いた m-カーネル専用機能では，ロック処理の箇所を考慮することでオーバヘッドを削減できる．そこで，オーバヘッドの削減と処理の並列性を高めるため，複数のロック粒度を用いた排他制御を実現する．例えば，メモリ資源における ICA の剥がし，貼り付けといったサーバプログラム間通信時にプロセスごとに並列に実行される処理に対しては，ロック粒度を細かくし，処理の並列性を高める．一方，実メモリ確保といった処理速度の求められる処理に対しては，ロック粒度を粗くし，オーバヘッドを削減する．

3.3.4 OS 機能による利用方法の分類

m-カーネルへの処理依頼と m-カーネル専用機能の共有化の特徴を**表 2** に示す．表 2 より，プロセス生成といった処理速度の求められない m-カーネル専用機能は，改造工数の少ない m-カーネルへの処理依頼による実現が求められる．また，メモリ資源操作といった処理速度の求められ

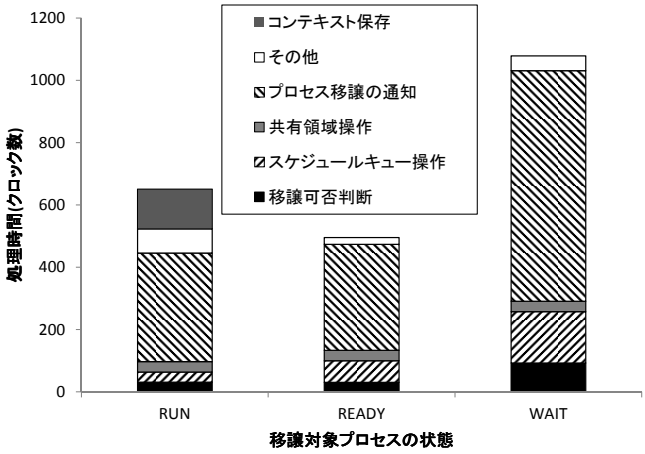

図 9 プロセス移譲機能の基本処理時間

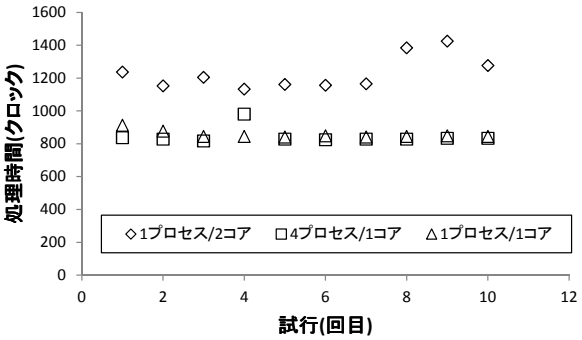

図 10 4KB の ICA 確保処理時間の分布

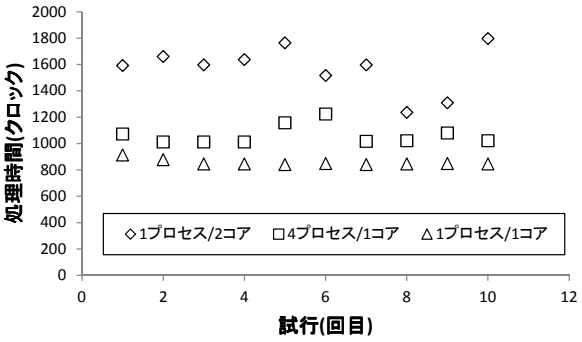

図 11 16KB の ICA 確保処理時間の分布

る m-カーネル専用機能は，m-カーネル専用機能の共有化による実現が求められる．

4. 評価

4.1 測定環境

プロセス移譲機能とカーネルコール処理分担機能をマルチコア ***AnT*** に実現し，Core i7-2600（3.4GHz）の計算機で走行させた．

4.2 基本評価

4.2.1 プロセス移譲機能

プロセス移譲機能を用いて，RUN 状態，READY 状態，および WAIT 状態のプロセスを移譲する処理の性能を明らかにする．

動作コア ID が 0 のコアから動作コア ID が 1 のコアへのプロセス移譲を 100 回試行した際の平均処理時間を**図 9** に示す． 図 9 より以下のことがわかる．

（1）移譲対象プロセスの状態が RUN 状態の場合，移譲対象プロセスの状態が READY 状態の場合に比べ，処理時間が約 150 クロック増加している．これは，移譲対象プロセスの状態が RUN 状態の場合，移譲元コアと移譲先コアのスケジュールキュー操作だけでなく，移譲対象プロセスのコンテキストの保存処理（約 130 クロック）も行なっているためである．

（2）移譲対象プロセスの状態が WAIT 状態の場合，他の状態に比べ，処理時間が約 2 倍になっている．これは，移譲対象プロセスの状態が WAIT 状態である場合，プロセス状態の調査と WAIT スケジュールキュー操作を行なっているためであると考えられる．

4.2.2 カーネルコール処理分担機能

マルチコア ***AnT*** に実現したカーネルコール処理分担機能において，共有化した m-カーネル専用処理である createica() カーネルコールについて評価を行なう．createica() は，メモリ資源を確保し，確保したメモリ資源を ICA として扱い，確保した ICA を発行元プロセスの仮想空間に貼り付けるカーネルコールである．このカーネルコールにおいて，ICA のサイズによる処理時間を明らかにする．ICA のサイズを 4KB，または 16KB とし，createica() を 10 回試行した際の処理時間の分布を**図 10** と**図 11** に示す．図 10 と図 11 より以下のことがわかる．

（1）カーネルコールを発行するプロセスを 1 コアで 1 プロセス動作させた場合と 4 プロセス動作させた場合の処理時間は，同等である．これは 1 コアで複数のプロセスを動作させても，同時にカーネルコールを発行するプロセスは高々 1 つであり，ロックの競合は発生しないためである．

（2）カーネルコールを発行するプロセスを 2 コアでそれぞれ 1 プロセス動作させた場合の処理時間は，1 コアで動作させた場合の処理時間に比べ約 400 クロック増加している．これは，資源操作時にロックの競合が発生したためであると考えられる．

（3）図 11 において，2 コアで動作させた場合の処理時間は，8，9 回目の際，約 400 クロック短縮している．これは，資源操作時にロックの競合が発生しなかったためであると考えられる．

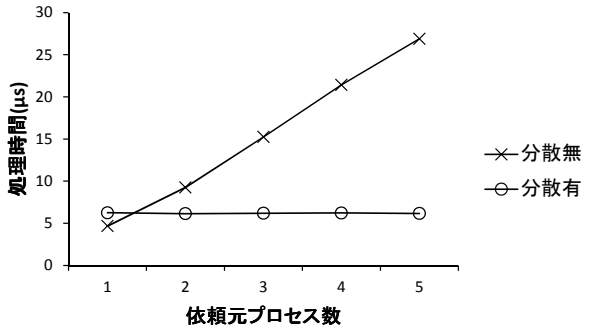

図 12 データ読み込み処理の応答時間

4.3 OS 機能の処理分散機能

マルチコア ***AnT*** のファイル管理機能を用いて，ファイルのデータの読み込みに要する応答時間を測定する．マルチコア ***AnT*** のファイル管理機能は，ファイル管理部とブロック管理部の２つの OS サーバに機能を分割している．ファイル管理部は，サービスからのファイルのデータ読み込み処理依頼を受け，操作対象ファイルへのアクセス権限の確認を行ない，ブロック管理部へのファイルのデータ読み込み処理依頼を行なう．ブロック管理部は，ファイル管理部からの処理依頼を受け，自身が管理するファイルキャッシュを探索し，ファイルのデータを返却する．これら２つの OS サーバとサービスを別々のコア上で分散動作させた場合と同一のコア上で動作させた場合を比較し，処理分散の性能を明らかにする．

応答時間は，依頼元プロセスが OS サーバへ処理依頼を行なった後，結果を受け取るまでの処理時間である．この処理を 10 回試行し，平均処理時間を算出した．なお，OS サーバの優先度はサービスより高く，サービスの優先度は全て同じである．

データ読み込み処理の応答時間を**図 12** に示す． 図 12 より以下のことがわかる．

(1) OS サーバを分散している場合の応答時間は，サービス数によらず一定である．これは，OS サーバを分散している場合，OS サーバとサービスは異なるコアで動作しており，OS サーバは常に RUN 状態になるためである．つまり，OS サーバは，サービスの要求を即時に実行可能であり，応答時間は一定になる．

(2) サービス数が 1 のとき，OS サーバを分散している場合の応答時間は，OS サーバを分散していない場合に比べ，約 1.5 μ 秒長い．これは，OS サーバを分散している場合，OS サーバを起床するためには，プロセス切り替えに加え，p-カーネルへの IPI 送信（本評価では IPI 送信回数は 3 回であり， 1 回の IPI 送信時間は約 0.5 μ 秒のため，約 1.5 μ 秒）を行なうためである．

5. おわりに

マルチコア ***AnT*** における処理分散機能として，プロセス分散方式と別コア OS 機能利用方式について述べた．具体的には，プロセス移譲機能として，移譲可能なプロセスの処理状態を明確化し，移譲方式を述べた．また，カーネルコール処理分担機能として，m-カーネルへの処理依頼と m-カーネル専用機能の共有化を使い分けることについて述べた．

プロセス移譲機能の評価では，移譲対象プロセスが RUN 状態の場合, コンテキストの保存処理を行なうため, READY 状態に比べ処理時間が約 130 クロック増加することを明らかにした．また，カーネルコール処理分担機能の評価では，ロックの競合が発生した場合，処理時間が約 400 クロック増加することを明らかにした．さらに，OS 機能の処理分散機能の評価では，ファイル管理機能を 1 コア上で動作させた場合，依頼元プロセス数の増加により応答時間が増加するのに対し，各コアで分散動作させた場合，依頼元プロセス数によらず応答時間が一定になることを明らかにした．

残された課題として，依頼や結果用のキューが空でない場合の移譲方式の検討がある．

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] 谷口秀夫, 乃村能成, 田端利宏, 安達俊光, 野村裕佑, 梅本昌典, 仁科匡人, “適応性と堅牢性をあわせもつ ***AnT*** オペレーティングシステム,” 情報処理学会研究報告, vol.2006-OS-103, pp.71-78 (2006.07).

[2] J. Liedtke, “Toward real microkernels,” Commun. ACM, vol.39, no.9, pp.70-77 (1996.09).

[3] A.S. Tanenbaum, J.N. Herder, and H. Bos, “Can we make operating systems reliable and secure?,” IEEE Computer Magazine, vol.39, no.5, pp.44-51 (2006.05).

[4] D.L. Black, D.B. Golub, D.P. Julin, R.F. Rashid, R.P. Draves, R.W. Dean, A. Forin, J. Barrera, T. Hideyuki, G.R. Malan, and D. Bohman, “Microkernel operating system architecture and mach,” J. Inf. Process., vol.14, no.4, pp.442-453 (1992.03).

[5] 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 ,” 研究報告マルチメディア通信と分散処理 (DPS) , vol.2012-DPS-150, no.37, pp.1-8 (2012.02).

[6] Rotem. E., Naveh. A., Rajwan. D., Ananthakrishnan. A., and Weissmann. E., “POWER-MANAGEMENT ARCHITECTURE OF THE INTEL MICROARCHITECTURE CODE-NAMED SANDY BRIDGE,” Micro, IEEE, vol.32, no.2, pp.20-27, (2012.03-04).

[7] 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.1977-1989 (2010.10).

[8] Milojicic. D.S., Douglis. F., Paindaveine. Y., Wheeler. R., Zhou. S., “Process migration,” ACM Computing Surveys, vol.32, no.3, pp.241-299 (2000).