Vol.2020-DPS-185 No.10
2020/12/22

マルチコア *AnT* を用いた処理分散効果の比較

小林 優也[1]　佐藤 将也[1]　谷口 秀夫[1]

概要：マルチコアを搭載した計算機が普及している．そこで，プロセスを複数のコアに配置して動作させることで処理分散が可能になる．オペレーティングシステム（以降，OS）には，マイクロカーネル構造 OS とモノリシックカーネル構造 OS がある．マイクロカーネル構造 OS は，最小限の OS 機能をカーネルとして実現し，その他の大半の OS 機能をプロセス（以降，OS サーバ）として実現する．モノリシックカーネル構造 OS は，すべての OS 機能をカーネルとして実現する．この 2 種類の OS は，実現できる処理の分散形態が異なる．本稿では，マイクロカーネル構造 OS である ***AnT*** と Fiasco.OC（Fiasco Object-oriented Capability system）について，プロセスを複数の CPU コアに分散配置することによる処理分散効果を比較する．また，***AnT*** とモノリシックカーネル構造 OS である Linux について，応用プログラム（以降，AP）処理と OS 処理の分散形態の違いに着目した性能を比較評価する．

キーワード：マイクロカーネル OS，オペレーティングシステム，負荷分散

1. はじめに

プロセッサ内のコア数の増加に伴い，処理を各コアに分散することが重要になっている．プロセッサを管理する OS には，マイクロカーネル構造 OS [1], [2], [3], [4], [5] とモノリシックカーネル構造 OS [6], [7] がある．マイクロカーネル構造 OS は，必要最低限の OS 機能をカーネルとして実現し，他の大半の OS 機能をプロセスとして実現する．このため，AP プロセスを各コアに分散できるとともに，AP プロセスの分散形態に関係なく，OS サーバを各コアに分散することで，OS 処理も分散できる [8]．一方，モノリシックカーネル構造 OS は，AP プロセスのシステムコール発行によって OS 処理を同じコアで実行する．したがって，マイクロカーネル構造 OS とモノリシックカーネル構造 OS は実現できる処理の分散形態が異なる．

我々は，マイクロカーネル構造を有するマルチコア ***AnT*** オペレーティングシステム（An operating system with adaptability and tough-ness）[9]（以降，***AnT***）を提案した．***AnT*** は，プロセスを任意のコアに移譲する機能を備えている．この機能を用いることで，プロセスを各コアに分散し，負荷分散を行うことができる．

我々は，***AnT*** を用いて，ディスク IO が頻繁に発生する場合の AP と OS の処理分散効果を評価し，モノリシックカーネル構造 OS である Linux と比較評価した [10]．しかし，キャッシュの影響によりディスク IO がほとんど発生しないような処理における評価が十分ではない．

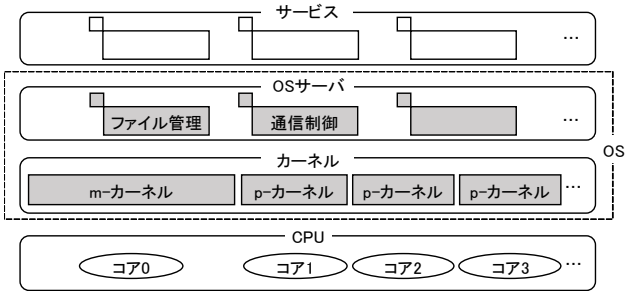


図 1　***AnT*** の基本構造

本稿では，マイクロカーネル構造 OS の ***AnT*** と Fiasco.OC（Fiasco Object-oriented Capability system）[11] について，プロセスを複数の CPU コアに分散配置する場合の性能を比較評価する．Fiasco.OC は，オープンソースとして広く利用されており，マルチコア対応，および高いリアルタイム性を実現した組み込み機器向けのマイクロカーネル構造 OS である．また，***AnT*** とモノリシックカーネル構造 OS の Linux について，ディスク IO がほとんど発生しない場合の AP と OS の分散形態における性能を比較評価する．

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

© 2020 Information Processing Society of Japan

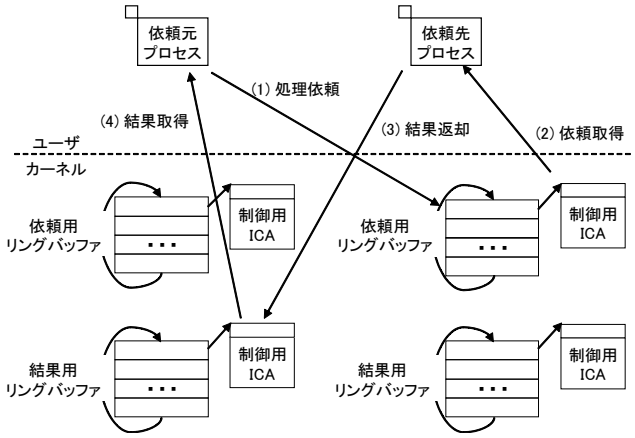


図 2　サーバプログラム間通信の基本機構

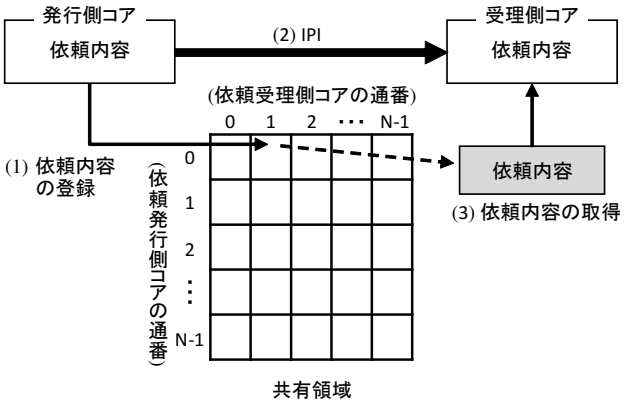


図 3　コア間通信の処理流れ

2. *AnT* オペレーティングシステム

2.1 基本構造

AnT はマルチコアに対応したマイクロカーネル構造 OS である．マルチコア ***AnT*** の基本構造を図 **1** に示し，以下に説明する．カーネルはシステムの動作に必要な最小限の OS 機能を実現するプログラム部分である．また，***AnT*** のカーネルは，コア毎にカーネルを配置する構造（マルチカーネル構造 [12]）となっており，マスタカーネル（m カーネル）とピコカーネル（p カーネル）の 2 種類のカーネルからなる．m カーネルはマイクロカーネルのすべての機能を有する．p カーネルは，プロセス実行制御機能，コア間通信制御機能，およびプログラム間通信機能のみを有する．コア毎にカーネルを配置することにより，各コアで独立したスケジューラを実現し，排他制御を抑制できる．さらに，m カーネルと p カーネルの構成にすることで，カーネルのテキスト部の増加を抑制できる．OS サーバは，OS 機能をプロセス化した部分であり，ファイル管理サーバや通信制御サーバがある．サービスは，サービスを提供するプログラム部分である．

2.2 複写レスデータ授受

AnT は，AP プロセスと OS サーバが相互に通信する際に用いるコア間通信データ域 (ICA: Inter-core Communication Area) を持つ．ICA は以下の 3 つの特徴を持つ．

（1）ページを単位とし，n ページ分の領域の確保と解放
（2）確保した領域（n ページ）の実メモリ連続の保証
（3）2 仮想空間での領域の張替え

ICA は，ページを最小単位として管理される領域である．ICA へのアクセスは，プロセス毎のマッピング表を通して行う．ここで，マッピング表への書き込みを貼り付けと呼び，マッピング表からの削除を剥がしと呼ぶ．ICA を利用したプロセス間のデータ授受は，授受するデータを格納した ICA をデータ授受元プロセスの仮想空間から剥がし，データ授受先プロセスの仮想空間に貼り付けることで行う．

2.3 サーバプログラム間通信機構

2.3.1 基本構造

サーバプログラム間通信機構を図 **2** に示し，以下に説明する．この機構では，ICA を用いて，プロセス間のデータ授受を複写レスで実現している．具体的には，依頼先プロセスへ渡す引数や通信制御の情報（以降，依頼情報）を制御用の ICA（以降，制御用 ICA）に格納し，扱うデータをデータ用の ICA（以降，データ用 ICA）に格納する．また，カーネルは，プロセス毎に通信のための依頼用リングバッファと結果用リングバッファを持つ．

コアを跨いだサーバプログラム間通信では，プロセスを起床させる処理において，コア間通信を伴う．各コアのカーネルは，スケジュールキューを持ち，プロセスの実行を制御する．このため，他コアのプロセスの起床は，コア間通信を用いて当該コアに依頼する．また，全てのプロセスは，コア数と同数の依頼用リングバッファと結果用リングバッファを持つ．そして，他コアのプロセスに対して依頼情報や結果情報の登録を行う際は，自身の走行するコアに対応したリングバッファに登録する．これにより，排他制御オーバヘッドを削減している．

2.3.2 コア間通信

コア間通信は，コア間で共有している領域（共有領域）を利用し，依頼内容の授受を行うことで実現する．

共有領域は $N \times N$（N はコア数）の配列として実現されており，依頼発行側コア（i）と依頼受理側コア（j）のコア間通信は，共有領域の（i，j）番目のエントリを利用して依頼内容を授受する．これにより，複数のコアが 1 つのコアにコア間通信を要求する際の排他制御処理を不要としている．

依頼内容の登録を通知するために IPI（Inter-Processor Interrupt）を利用する．コア間通信の処理流れを図 **3** に示し，以下に説明する．

© 2020 Information Processing Society of Japan

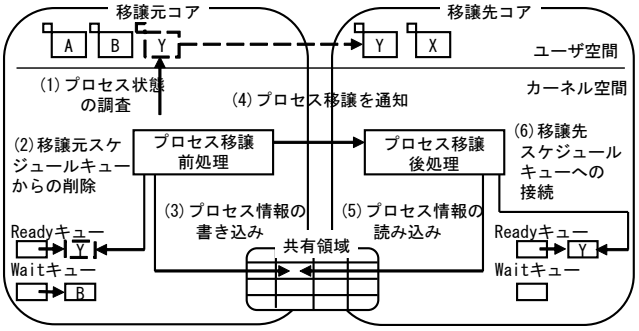


図 4　プロセス移譲機能の処理流れ

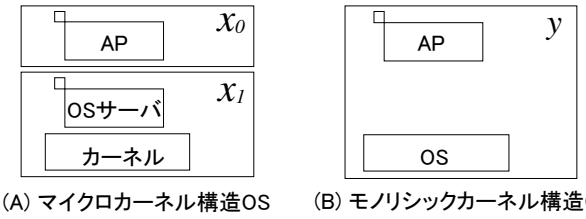


図 5　処理の分散単位

（1）依頼発行側コアは依頼内容を共有領域に登録する．
（2）依頼発行側コアは依頼受理側コアに IPI を送信し，依頼受理側コアに依頼内容の登録を通知する．
（3）依頼受理側コアは，IPI の受信を契機に，共有領域から依頼内容を取得し，依頼内容の処理を実行する．

2.4　プロセス移譲機能

コア毎に独立したスケジューラは他コアのスケジュールキューを操作しない．プロセスのコア分散は，プロセス移譲機能で行う．プロセス移譲機能の処理流れを**図 4** に示し，以下に説明する．
（1）状態の調査では，移譲可能なプロセス状態の判定を行う．
（2）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）書き込みでは，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）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）読み込みでは，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）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．

表 1　分散形態実現の可否

AP 処理	OS 処理	マイクロ	モノリシック
集中	集中	可	可
分散	集中	可	不可
集中	分散	可	不可
分散	分散	可	可

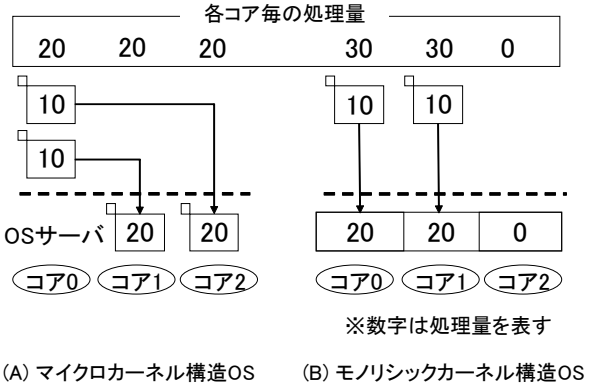


図 6　AP 処理と OS 処理の分散形態の例

3.　処理の分散形態

3.1　処理の分散単位

AP と OS の処理の分散単位を**図 5** に示す．マイクロカーネル構造 OS（図 5（A））は，AP プロセスと OS サーバを各コアに分散することで，AP 処理と OS 処理を別のコアで実行できる．つまり，AP 処理と OS 処理を別の単位（x_0 と x_1）で分散できる．
モノリシックカーネル構造 OS（図 5（B））は，AP プロセスのシステムコール発行によって OS 処理を実行する．このため，AP プロセスを各コアに分散することで OS 処理を分散できる．いいかえれば，AP プロセスを各コアに分散しない場合，OS 処理を分散できない．つまり，AP 処理と OS 処理を 1 つの単位（y）として分散する．

3.2　比較

AP 処理と OS 処理の 4 種類の分散形態について，各 OS が実現可能か否かを**表 1** に示す．「集中」は，複数の処理を同一コア上で実行することを示し，「分散」は，複数の処理を各々別コア上で実行することを示す．マイクロカーネル構造 OS は，全形態を実現できる．また，「集中，集中」と「分散，分散」の場合において，AP 処理と対応する OS 処理を同一コア上で実行しなくてもよい．一方，モノリシックカーネル構造 OS は，AP 処理と OS 処理が同じ形態（「集中」または「分散」）でなければ実現できない．さらに，「集中，集中」と「分散，分散」の場合，AP 処理と対応する OS 処理を同一コア上で実行する．
AP 処理と OS 処理の分散形態の例を**図 6** に示す．プロセッサ（PU）処理の量を数字で表す．AP 処理の量を 10，

© 2020 Information Processing Society of Japan

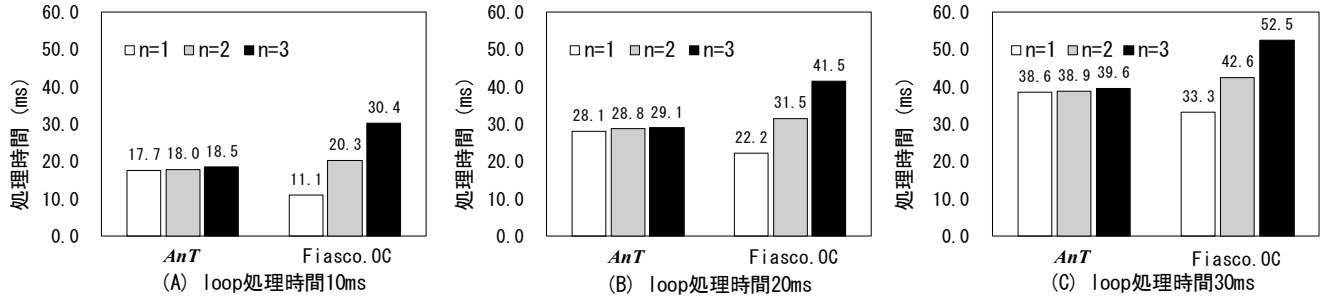


図 8 マイクロカーネル構造 OS の処理分散効果

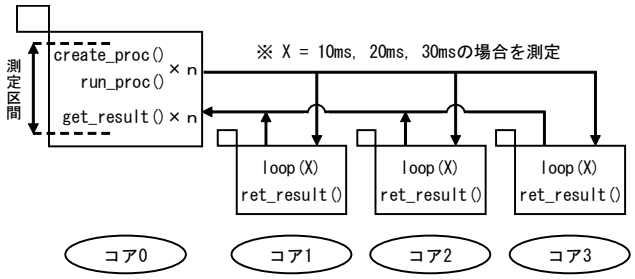


図 7 処理分散効果の評価モデル

対応する OS 処理の量を 20 としている．マイクロカーネル構造 OS の場合（図 6（A）），2 つの AP 処理をコア 0，対応する OS 処理（OS サーバ）をコア 1 とコア 2 に分散させることで，各コアの PU 処理量を均等化（20）できる．一方，モノリシックカーネル構造 OS の場合（図 6（B）），AP 処理をコア 0 とコア 1 に分散させると，対応する OS 処理もコア 0 とコア 1 で実行される．このため，コア 0 とコア 1 の各 PU 処理量は 30，コア 2 の PU 処理量は 0 となり，負荷分散がうまくできない．

4. 評価

4.1 マイクロカーネル構造 OS の処理分散効果

4.1.1 評価の環境と内容

評価用計算機は，CPU として Intel Core i7-2600 （3.4 GHz），ハードディスクドライブとして Western Digital WD1600AAJS（8 MB cache，7,200 rpm），および 8,192 MB のメモリを搭載している．マイクロカーネル構造 OS において，処理を各コアに分散した場合の処理時間を評価する．***AnT*** と Fiasco.OC における処理分散効果の評価モデルを**図 7** に示す．評価プログラムは，1 から 3 個のプロセスを生成し，各コアに分散する．生成されたプロセスは，10ms，20ms，または 30ms の loop 処理を実行した後，結果を返却する．評価では，プロセスの生成から全ての結果を取得するまでの処理時間を測定する．

4.1.2 結果と考察

測定結果を**図 8** に示し，以下に考察を述べる．

（1）loop 処理時間の長短に関係なく，分散プロセス数の変化に対し，***AnT*** の処理時間の変化は非常に少なく，Fiasco.OC の処理時間の変化（分散プロセス数の増加による処理時間の増加）は大きい．例えば，loop 処理時間が 10ms の場合（A），分散プロセス数を 1 個から 3 個に増加させたときの処理時間の増加は，***AnT*** では 0.8ms（18.5ms−17.7ms）と小さい．一方，Fiasco.OC では 19.3ms（30.4ms−11.1ms）と大きい．これは，***AnT*** では，各コアに独立したスケジューラを実装し，他コアのスケジュールキュー操作における排他制御を不要としているため，分散プロセス数の増加によるオーバヘッドの増加が小さいためである．Fiasco.OC では，コア毎に独立したスケジューラでないため，他コアのスケジュールキューの操作に排他制御が必要となり，分散プロセス数の増加によるオーバヘッドの増加が大きい．

（2）分散プロセス数が 1 の場合，loop 処理時間の長短に関係なく，Fiasco.OC の処理時間は ***AnT*** の処理時間より短い．例えば，loop 処理時間が 10ms の場合（A），***AnT*** よりも Fiasco.OC の方が 6.6ms（17.7ms−11.1ms）短い．これは，分散プロセス数が少ない場合，組み込み機器向けにリアルタイム性を重視して開発された Fiasco.OC の OS オーバヘッドが低いためである．

（3）Fiasco.OC の処理時間は，分散プロセス数の増加に伴い処理時間が増加し，その増加の程度は，loop 処理時間が短いほど大きい．例えば，loop 処理時間が 10ms の場合（A），分散プロセス数を 1 個から 3 個に増加させたとき，Fiasco.OC の処理時間の増加は約 2.73 倍であり，（C）の場合（1.58 倍）よりも大きい．これは，Fiasco.OC の場合，分散する処理の処理時間が短いと，AP 処理に比べ相対的に処理分散の OS オーバヘッドが大きくなるためである．

4.2 ***AnT*** と Linux の比較

4.2.1 評価の環境と内容

AP と OS の分散形態による処理時間を評価する．分散効果の評価環境は，4.1.1 項と同様である．マイクロカーネル構造 OS では，AP プロセスと OS 処理をそれぞれ別コアで実行できるのに対して，モノリシックカーネル構造 OS では別コアで実行できないことを述べた．この違いによる影響を明確化するため，各 OS で 2 種類ずつの分散形態について評価する．***AnT*** と Linux 5.3.1（CentOS7）に

© 2020 Information Processing Society of Japan

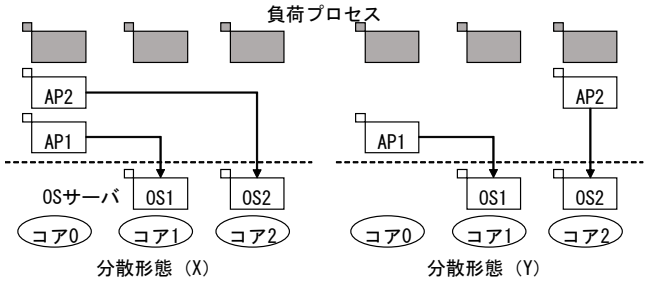


図 9　***AnT*** の分散形態

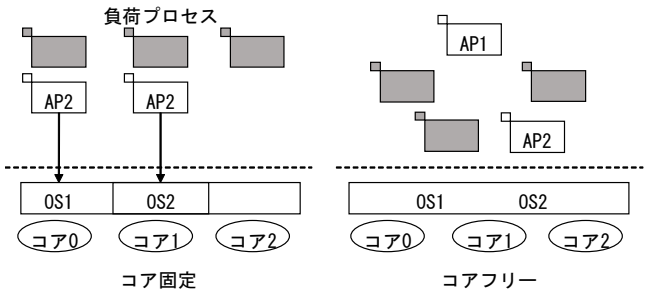


図 10　Linux の分散形態

おいて AP 処理と OS 処理を分散した形態を**図 9** と**図 10** に示す．

AP の処理流れを**図 11** に示す．AP は，RAW デバイスからのランダムリード（4 KB）を n 回繰り返す．なお，ランダムの範囲は 2 GB または 512 KB とした．シーケンシャルリードを測定する際は lseek() は実行しない．***AnT*** の場合，RAW デバイスからの読み込み処理は，制御用 ICA に依頼情報を格納し，サーバプログラム間通信を用いてディスクドライバサーバ（図 9 では OS1 と OS2）に依頼する．Linux の場合，read システムコールを用いて OS に RAW デバイスからの読み込みを依頼する．

AP 処理と OS 処理の PU 負荷の大小が分散効果に与える影響を明らかにするため，AP 処理と OS 処理のそれぞれに PU 処理（0，2，3ms）を追加した．また，コアの PU 負荷が大きい場合の分散効果を評価するために，負荷プロセスをコア当たり 1 個または 2 個共存走行させる．負荷プロセスは，PU 処理（0，5，10ms）と WAIT 状態（1ms）を繰り返し実行する．各処理の優先度は（OS 処理）>（負荷プロセス）>（AP プロセス）であり，AP1 と AP2 の優先度は等しい．なお，負荷プロセスと AP2（繰り返し回数 $n \gg 100$）は，AP1（繰り返し回数 $n = 100$）が処理を終えるまで処理を継続し，AP1 の処理時間を測定した．

4.2.2　結果と考察

ランダム範囲が 2 GB の場合の評価は文献 [10] で報告済みである．文献 [10] では，評価結果より以下のことを述べた．

（1）AP と OS の PU 処理が 0ms の場合，プロセッサ利用量は非常に少なく，これらの処理は，負荷プロセスが待ち状態（1ms）の間に行われる．したがって，処理分散効果はなく，処理時間はコア間通信を必要としない Linux が ***AnT*** より短い．ただし，その差は非常に小さい．

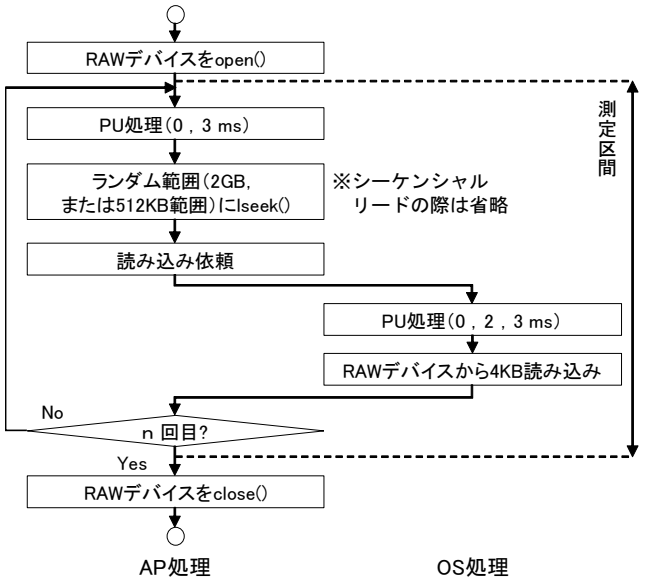


図 11　AP の処理流れ

（2）PU 処理（OS）が増える場合，Linux の処理時間が大きく増加し，***AnT*** より長くなる．これに対し，***AnT*** の処理時間はほとんど増加しない．したがって，***AnT*** は Linux に比べ処理分散効果が大きい．

（3）***AnT*** の分散形態（X，Y）による処理時間の影響は，PU 処理（AP）が大きい場合に明らかである．AP の処理分散を行っている分散形態（Y）の処理時間は，AP の処理分散を行っていない分散形態（X）より短い．

（4）Linux の分散形態（コア固定，コアフリー）による処理時間は，コアを自由に使える分散形態（コアフリー）の処理時間が，走行するコアを指定する分散形態（コア固定）より短い．

（5）コア当たりの負荷プロセスが 2 個になると，コアの負荷が大きくなり，項目（1）や（2）の傾向が強く表れる．

（6）コア当たりの負荷プロセスが 2 個になると，Linux の分散形態（コア固定，コアフリー）の処理時間は，分散形態（コア固定）の方が早くなっている．これは，プロセス数が多いため，コアを自由に利用できる効果がなくなったためである．

負荷プロセスが 1 個，ランダム範囲が 512 KB の場合の処理時間を**図 12** に示す．図 12 より以下がわかる．

（1）PU 処理が 0ms の場合（A），AP 処理と OS 処理のプロセッサ利用量は非常に少なく，これらの処理は，負荷プロセスが待ち状態（1ms）の間に行われる．したがって，処理分散効果はなく，処理時間はコア間通信を必要としない Linux が ***AnT*** より短い．ただし，その差は非常に小さい．これは，ランダム範囲が 2GB の場合の傾向と同様である．

（2）PU 処理（OS）が増える場合（（A）から（C）），Linux

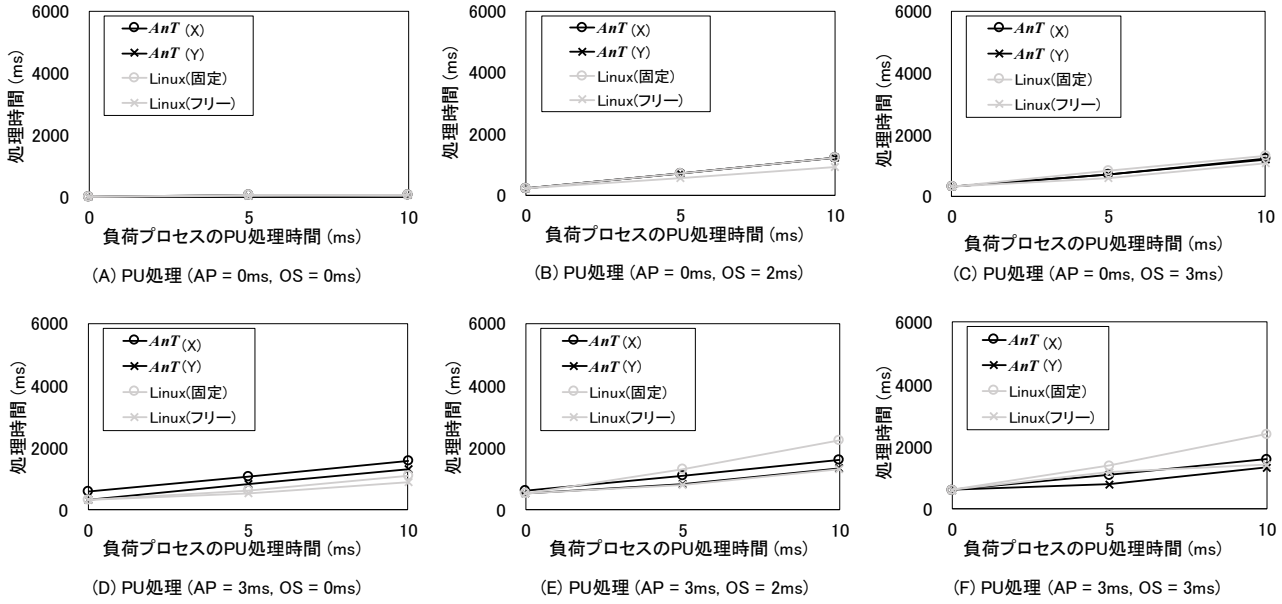


図 12 処理時間（負荷プロセス 1 個，ランダム範囲 512 KB）

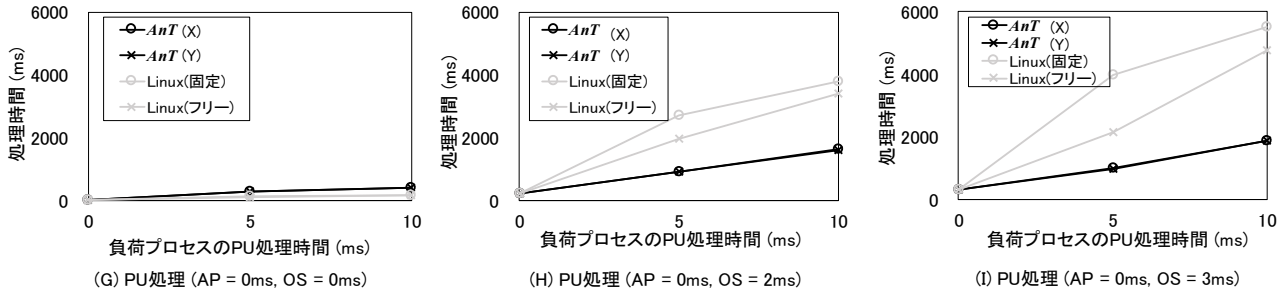


図 13 処理時間（負荷プロセス 2 個，ランダム範囲 512 KB）

（コアフリー）の処理時間の増加が最も小さい．しかし，（B）から（C）に着目すると，Linux の処理時間は増加しているのに対して，***AnT*** の処理時間は増加していない．したがって，***AnT*** は Linux に比べ処理分散効果が大きい．また，PU 処理（AP）が 3ms の場合も同様である．これは，ランダム範囲が 2GB の場合の傾向と同様である．

（3）***AnT*** の分散形態（X，Y）による処理時間の影響は，PU 処理（AP）が大きい場合（3ms，（D）から（F））に明らかである．AP の処理分散を行っている分散形態（Y）の処理時間は，AP の処理分散を行っていない分散形態（X）より短い．この傾向はランダム範囲が 2GB の場合よりも強く表れている．これは，IO 発生回数が少なくなった結果，処理の大部分を PU 処理が占めるようになり，AP プロセスの分散効果が大きく影響するためである．

（4）Linux の分散形態（コア固定，コアフリー）による処理時間の影響は，（A）から（F）のすべての場合で明らかである．コアを自由に使える分散形態（コアフリー）の処理時間は，走行するコアを指定する分散形態（コア固定）より短い．この傾向はランダム範囲が 2GB の場合よりも強く表れている．これは，（3）の場合と同様の理由である．

負荷プロセスが 2 個，ランダム範囲が 512 KB の場合の処理時間を**図 13** に示す．

（5）負荷プロセスが多いため，項目（1）や（2）の傾向が強く表れる．これは，ランダム範囲が 2GB の場合の傾向と同様である．

（6）Linux の分散形態（コア固定，コアフリー）の処理時間は，コアを自由に使える分散形態（コアフリー）の処理時間が分散形態（コア固定）よりも短い．この傾向は（4）と同様である．

アクセスパターンをランダムからシーケンシャルに変えた場合の傾向の変化は，ランダム範囲を 2 GB から 512 KB に変えた場合の傾向の変化と同様である．しかし，その傾向の表れ方は小さい．これは，IO 発生回数が，512 KB とシーケンシャルの方が近いためである．

4.2.3 処理時間の増加の割合

傾向が強く表れる負荷プロセスが 2 個の場合を例に，処理時間の増加をまとめる．各 OS におけるディスクドライバの性能の違いを無視して比較するために，各処理時間を基本処理時間を 1 としたときの割合で表す．基本処理時間は，AP プロセスと OS サーバを同じコアに配置し，負荷プ

© 2020 Information Processing Society of Japan

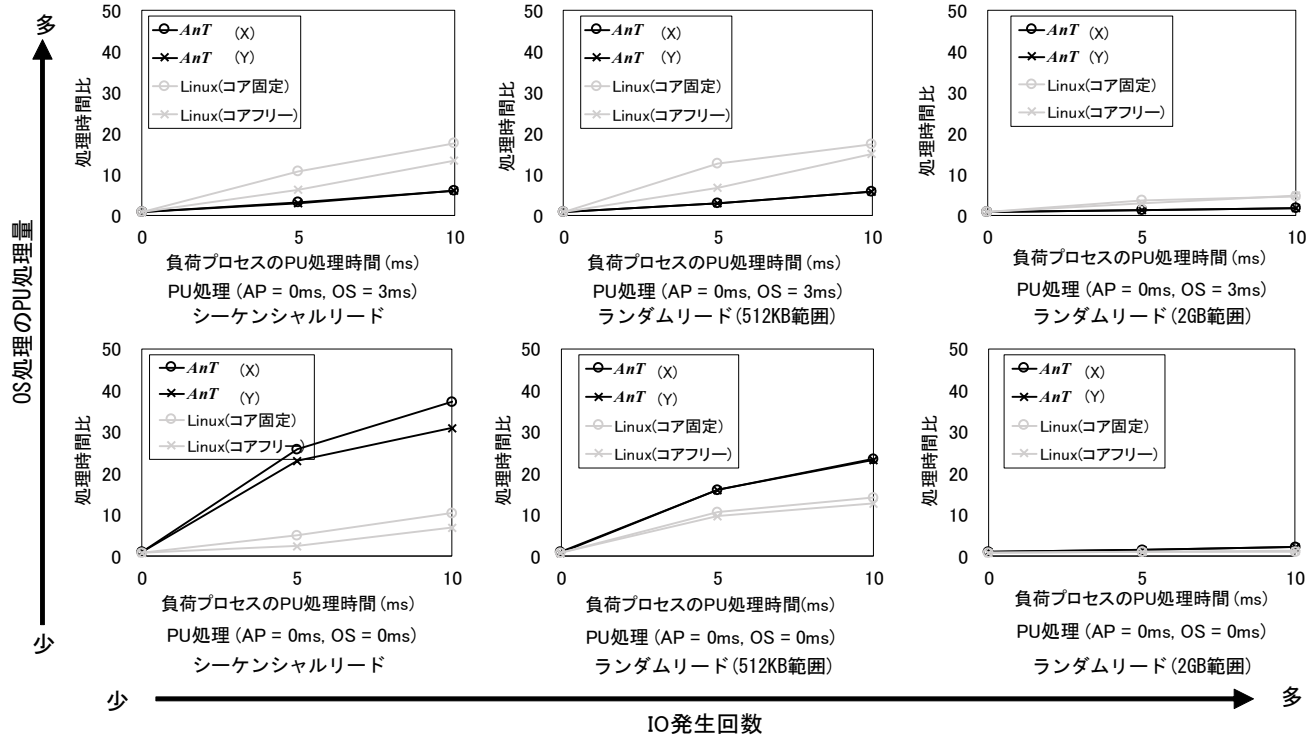


図 14　基本処理時間と負荷プロセスが 2 個の場合の各処理時間の割合

ロセスを無しとした場合の処理時間である．基本処理時間と負荷プロセスが 2 個の場合の各処理時間の割合を図 **14** に示す．図 14 から以下が分かる．

（1）OS 処理の PU 処理量が少なく，IO 発生回数が少ないとき，処理時間の増加率は，***AnT*** よりも Linux の方が小さい．これは，基本の read 処理時間が ***AnT*** よりも Linux の方が短いためである．

（2）OS 処理の PU 処理量が少なく，IO 発生回数が多いとき，***AnT*** と Linux の処理時間の増加率はほとんど等しい．これは，IO 発生回数が多い場合，処理時間全体に占める IO 待ち時間の割合が大きくなり，***AnT*** の処理分散効果の影響が小さく表れるためである．

（3）OS 処理の PU 処理量が多いとき，***AnT*** の処理時間の増加は Linux より小さい．これは，処理時間全体に占める PU 処理時間の割合が大きくなり，***AnT*** の PU 処理の分散効果が大きく影響するためである．

5. 関連研究

マルチカーネル構造を有するマイクロカーネル構造 OS に関する研究として，Multi Microkernel [13] がある．Multi Microkernel はマスタマイクロカーネルとスレーブマイクロカーネルの 2 種類のカーネルを持つ．マスタマイクロカーネルは，マイクロカーネルの全てのカーネル機能を持つ．一方，スレーブマイクロカーネルは，AP プロセスの実行に必要な最小限のカーネル機能のみを持つ．よって，OS 機能を有する唯一のカーネルであるマスタマイクロカーネルに処理が集中し，スループットが低下してしまう．***AnT*** は OS サーバを各コアに分散することで，スループットを向上できる．

マイクロカーネル構造 OS における処理の分散方法に関する研究として，factored operating systems [14]，NIX [15]，および GenerOS [16] がある．factored operating systems では，OS 処理は類似する処理（例えば，ファイルシステムサーバとディスクドライバサーバ）ごとに分類され，グループで各コアに静的に分散される．このため，グループとしてまとめられた OS 処理のスループットを向上できない．***AnT*** では，OS サーバを各コアに分散することで，特定の OS 処理のスループットを向上できる．また，NIX [15] と GenerOS [16] は，OS 処理のみを実行するコアと AP プロセスのみを実行するコアを静的に決定する．分散した OS 処理と応用プログラムの間の通信に排他制御が必要であり，排他制御オーバヘッドが大きい．***AnT*** では，コアごとのリングバッファや共有領域を用いた通信により，排他制御を抑制している．

マイクロカーネル構造 OS におけるプロセス間通信の高速化に関する研究として，SkyBridge [17] がある．Skybridge は，仮想化のためのハードウェア機能を利用して，ユーザ空間とカーネル空間の切り替えが発生しないプロセス間通信を実現している．***AnT*** では，複写レスデータ授受を利用して，プロセス間通信処理時間を高速化している．

© 2020 Information Processing Society of Japan

6. おわりに

マイクロカーネル構造 OS である ***AnT*** と Fiasco.OC について，処理の分散性能を述べた．また，***AnT*** とモノリシックカーネル構造 OS である Linux について，キャッシュの影響によりディスク IO がほとんど発生しないような処理における AP 処理と OS 処理の処理分散効果を述べた．

AnT と Fiasco.OC による AP 処理の分散性能の比較評価では，分散プロセス数を増加しても，コア毎に独立したスケジューラにより排他制御を抑制した ***AnT*** の処理時間の増加は小さいことを示した．具体的には，分散プロセス数を 1 個から 3 個に増加したとき，処理時間の増加は，***AnT*** では 0.8ms（約 5%）と小さいのに対して，Fiasco.OC では 19.3ms（約 174%）も増加する．これは，***AnT*** では，各コアに独立したスケジューラにより他コアのスケジュールキューの操作における排他制御が不要であり，分散プロセス数の増加によるオーバヘッドの増加が小さいためである．一方，Fiasco.OC では，コア毎に独立したスケジューラでないため，他コアのスケジュールキューの操作に排他制御が必要となり，分散プロセス数の増加によるオーバヘッドの増加が大きくなる．

AnT と Linux の AP 処理と OS 処理の処理分散効果の比較評価では，***AnT*** の場合，ディスク IO がほとんど発生しない場合（大半が PU 処理の場合）でも，AP 処理と OS 処理を各コアに自由に分散することで負荷を分散でき，Linux に比べ処理分散効果が高いことを示した．具体的には，OS 処理の PU 処理量が増加した時，AP 処理と OS 処理を別のコアで実行する ***AnT*** の方が，AP 処理と OS 処理を同じコアで実行する Linux よりも，処理時間の増加が小さいことを示した．なお，文献 [10] で示したディスク IO が頻繁に発生する場合と比べ，ディスク IO がほとんど発生しない場合は ***AnT*** の処理分散効果が高くなる．これは，ディスク IO がほとんど発生しない場合，処理時間全体に占める PU 処理時間の割合が大きくなり，PU 処理の分散効果が大きく影響するためである．

謝辞 本研究の一部は，JSPS KAKENHI 18K11244 による．

参考文献

[1] Black, D.L., Golub, D.B., Julin, D.P., Rashid, R.F., Draves, R.P., Dean, R.W., Forin, A., Bar-rera, J., Tokuda, H., Malan, M., and Bohman. D.: Microkernel Operating System Architecture and March, Journal of Information Processing, Vol. 14, No. 4, pp. 442–453 (1992).

[2] Bricker, A., Gien, M., Guillemont, M., Lipkis, J., Orr, D. and Rozier, M.: Architectural issues in microkernel-based operating systems: The CHORRUS experience, *Computer Communications*, Vol. 14, No. 6, pp. 347–357 (1991).

[3] Hartig, H., Hohmuth, M., Liedtke, J., Wolter, J. and Schonberg, S.: The performance of microkernel-based systems, Proc. 16th ACM Symp. Operating System Principles, pp.66–77 (1997).

[4] Liedtke, J.: Towards Real Microkernels, Communications of The ACM, vol. 39, issue 9, pp. 70–77 (1996).

[5] Tanenbaum, A.S., Herder, J.N., Bos, H.: Can We Make Operating Systems Reliable and Secure?, IEEE Computer Magazine, Vol. 39, No. 5, pp. 44–51 (2006).

[6] The FreeBSD Project. https://www.freebsd.org. (2020.11.24)

[7] The Linux Kernel Archives https://www.kernel.org. (2020.11.24)

[8] Hamad, M., Schlatow, J., Prevelakis, V., Ernst, R.: A communication framework for distributed access control in microkernel-based systems, In 12th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT16) pp. 11–16 (2016).

[9] 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，Vol. J93-D, No. 10, pp. 1977–1989 (2010).

[10] Kobayashi, Y., Sato, M., Taniguchi, H.: Evaluation of Processing Distribution for Application Program and OS in Microkernel OS, Proceedings of 2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW), pp. 440–444 (2019).

[11] The Fiasco microkernel Overview. https://os.inf.tu-dresden.de/fiasco/. (2020.11.24)

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

[13] Matarneh, R.: Multi Microkernel Operating System for MUlti-Core Processor, Journal of Computer Science, vol. 5, No. 7, pp. 493–500 (2009).

[14] Wentzlaff, D., Agarwal, A.: Factored operating systems (fos): the case for a scalable operating system for multicores, ACM SIGOPS Oper. Systems Review, Vol. 43, pp. 76–85 (2009).

[15] Ballesteros, F. J., Evans, N., Forsyth, C., Guardiola, G., McKie, J., Minnich, R., Salvador, E., S.: NIX: a case for a manycore system for cloud computing, Bell Labs Tech. J., Vol. 17, pp. 41–54 (2012).

[16] Yuan, Q., Zhao, J., Chen, M., Sun, N.: GenerOS: An asymmetric operating system kernel for multicore systems, Proc. 24th IEEE InternationalParallel & Distributed Processing Symposium, pp. 1–10 (2010).

[17] Mi, Z., Li, D., Yang, Z., Wang, X., Chen, H.: SkyBridge: Fast and Secure Inter-Process Communication for Microkernels, Proc. Fourteenth EuroSys Conference 2019, pp. 9:1–9:15 (2019).

[18] Shimosawa, T., Shimizu, M., Hori, A., Ishikawa, Y.: Interface for Heterogeneous Kernels: A Framework to Enable Hybrid OS Designs targeting High Performance Computing on Manycore Architectures, Proc. HiPC'14, pp. 1–10 (2014).

© 2020 Information Processing Society of Japan