OSの処理を多く含む並列処理の効率化を指向した一括システムコール機能

谷口秀夫[†] 日下部茂[†] 中山大士[††]
乃村能成[†] 雨宮真人[†]

並列実行を要求する処理の多様化にともない，応用プログラム（以降，APと略す）内での並列化だけではなく，APとオペレーティングシステム（以降，OSと略す）カーネルとの連携が深い処理の効率的な並列実行が求められている．そこで，OSカーネルの処理を多く含む並列処理を効率的に実行できる機能として，ユーザプログラムがカーネルプログラムに処理を依頼する際，依頼のたびにシステムコールを発行せず，複数の依頼をまとめて1つのシステムコールとして発行する一括システムコール機能を提案する．この機能は，システムコールにともなう処理の負荷を軽減でき，かつカーネル呼び出し回数を削減できる特徴を持つ．また，この機能の実現例により，一括システムコール機能は，パイプライン段数が大きい高性能なプロセッサほど有効であることを示している．

A Wrapped System-call Mechanism of Parallel Processing with Heavy Interaction between User-mode and Kernel-mode

Hideo Taniguchi,[†] Shigeru Kusakabe,[†] Hiroshi Nakayama,[††]
Yoshinari Nomura[†] and Makoto Amamiya[†]

By an appearance of various parallel processing types, effective execution of parallel processing with heavy interaction between user and kernel is demanded. So this paper proposes a wrapped system-call mechanism that executes multiple requests as one system-call in a mass. This mechanism has the characteristics that can reduce load of processing with a system-call, and that can reduce kernel call frequency. And, by realization example of this mechanism, a wrapped system-call mechanism shows that parallel processing is executed effectively in high performance processor with many pipeline stages.

1. はじめに

プロセッサの高性能化，計算機の小型化，およびプロセッサ間を結ぶ通信路の高速化により，並列処理を並列計算機だけではなく高性能なPCを利用したクラスタシステム[1)]でも実現することが可能になってきた．これにともない，様々な処理について，並列実行による処理の効率化が望まれている．従来，並列処理の研究では計算処理を行う応用プログラム（以降，APと略す）のレベルでの並列化を目指す研究が，並列言語[2)]や並列化コンパイラ[3)]を利用してさかんに行われている．一方，並列実行を要求する処理の多様化にともない，AP内での並列化だけではなく，APとオペレーティングシステム（以降，OSと略す）カーネルとの連携が深い処理の効率的な並列実行が求められている．このことは，AP内での並列化が多い処理であっても，その処理を計算機クラスタシステム上で実行する際には，プロセス間あるいはスレッド間での同期や通信はOSカーネル呼び出しとなり，APとOSカーネルとの効率的な連携が必要になる．つまり，OSカーネルの処理を多く含む並列処理を効率的に実行できれば，より広範囲な処理について並列実行を行え効率化が期待できる．

APは，ユーザモードで走行するOSカーネル外のプロセスのプログラム（以降，ユーザプログラムと呼ぶ）であり，カーネルモードで走行するOSカーネル内のプログラム（以降，カーネルプログラムと呼ぶ）にシステムコールを使って処理を依頼する．システム

† 九州大学大学院システム情報科学研究院
Graduate School of Information Science and Electrical Engineering, Kyushu University
†† 九州大学大学院システム情報科学府
Graduate School of Information Science and Electrical Engineering, Kyushu University

コールは，処理の依頼に対し，いつ制御が戻ってくるかにより，同期型と非同期型に分類できる．同期型は，処理の実行完了を待って制御が戻ってくるものであり，初期の UNIX [4]に代表される．非同期型は，処理の受付により制御が戻ってくるものであり，トランザクション処理用 OS [5]に見られる．この場合，AP は，実行結果の取得を陽に行う必要がある．したがって，同期型のシステムコールは，AP の処理が簡単であるものの，多くの処理を行うためには多くのプロセスを必要とするため，OS カーネル呼び出しが多発するとともに OS カーネルのオーバヘッドが大きい．これに対し，非同期型のシステムコールは，AP が実行結果の取得を陽に行う必要があるものの，1 つのプロセスで多くの処理を行うことができる．2 つのシステムコールの形態をともに実現した研究[6]があるが，使い分けが難しく，利便性が良いとはいえない．また，高性能なプロセッサ実現のために，命令パイプラインのように連続な命令列の実行を高速化する技術が広く使われるようになっている．しかし，システムコールは，連続な命令列の実行を阻害するため，命令パイプライン機能が有効に働かず処理性能の低下を招く．

そこで，本論文では，ユーザプログラムがカーネルプログラムに処理を依頼する際，依頼のたびにシステムコールを発行せず，複数の依頼をまとめて 1 つのシステムコールとして発行する一括システムコール機能を提案する．この機能を AP への共通機能として OS カーネル外に実現することにより，同期型のシステムコールと同様に AP の処理が簡単であるにもかかわらず，OS カーネル呼び出しを抑制できる．以降では，OS カーネルの処理を多く含む並列処理が要求する事項を明らかにし，一括システムコール機構について述べる．さらに，この機構の評価結果を報告する．

2. OS カーネルの処理を多く含む並列処理の要求

OS カーネルの処理を多く含む並列処理では，ユーザプログラムからカーネルプログラムへの処理の依頼が頻繁に発生する．このため，処理の依頼と結果の取得を効率的に行えることが求められる．また，命令パイプラインのように連続な命令列の実行を高速化してプロセッサ性能を向上させる技術が広く使われている今日では，この処理の依頼が連続な命令列の実行を阻害することに留意する必要がある．

最初に，ユーザプログラムからカーネルプログラムへの処理の依頼であるシステムコールについて，同期型と非同期型の得失を**表 1** に示す．

表 1 同期型と非同期型
Table 1 Synchronous and asynchronous system-call.

方式	長所	短所
同期型	(1) AP 処理が簡単 (2) システムコール処理の負荷小	(1) プロセス数増加による OS オーバヘッド大 (2) OS 呼び出し多発による性能低下大
非同期型	(1) プロセス数増加を抑制	(1) AP 処理が複雑 (2) システムコール処理の負荷大 (3) OS 呼び出し多発による性能低下大

同期型は，処理の実行完了を待って制御が戻ってくるため，非同期型のように実行結果の取得を別に行う必要がなく，AP の処理が簡単である．また，システムコール処理の負荷は大きくないため，プロセッサ負荷が小さい場合は処理効率が良い．しかし，処理を並列に行うためには多くのプロセス（またはスレッド）を必要とするため，多数のプロセス（またはスレッド）制御のための OS カーネルのオーバヘッドが大きくなってしまう．さらに，高性能なプロセッサでは，命令実行のパイプライン段数が多く，かつメモリキャッシュの大きさも大きいため，関数呼び出しによるジャンプ命令が多くかつ走行モード変更をともなう OS カーネル呼び出し処理が多発すると性能が大きく低下すると予想される．

非同期型は，処理の受付により制御が戻ってくるため，別途，実行結果の取得を陽に行う必要がある．また，処理の受付と実行結果の取得の処理を分離するため，システムコール処理の負荷は大きくなってしまう．しかし，処理を並列に行う場合でも，少ないプロセス（またはスレッド）で処理を行え，OS カーネルのオーバヘッドが大きくならない．なお，OS カーネル呼び出しの回数は，同期型の 2 倍になるため，OS カーネル呼び出し処理の実行にともなう性能低下は深刻である．

以上に述べた同期型と非同期型の得失から，以下が求められる．

（要求 1） AP の処理が簡単
（要求 2） システムコール処理の負荷削減
（要求 3） 処理に必要なプロセス（またはスレッド）数の抑制

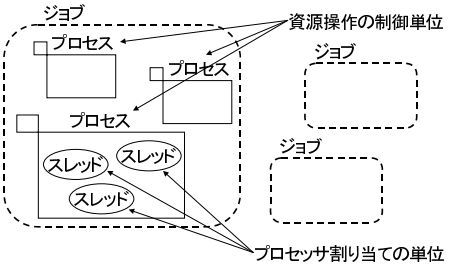


図 1 ジョブとプロセスとスレッドの関係
Fig. 1 Relations among job, process and thread.

（要求 4） カーネル呼び出し回数の削減

次に，システムコールが生み出す連続な命令列実行の阻害について述べる．ユーザプログラムからカーネルプログラムへの処理の依頼であるシステムコールでは，走行モードの変更が起こる．さらに，OS やプロセッサに依存するものの，メモリ空間の切替えが発生することもある．このため，命令やデータのキャッシュクリア，あるいは TLB のクリアが起こることがある．一方，プロセッサを高性能化する技術としては，連続な命令列の実行を高速化するパイプライン技術や，メモリへのアクセス回数を削減するキャッシュ技術が広く使われている．したがって，システムコールは，プロセッサを高性能化する技術の効果を低下させるものであり，その発行回数の削減（要求 4）が重要である．

3. 一括システムコール

3.1 モ デ ル

並列処理のシステム形態として，システムにおけるジョブとプロセスとスレッドの様子を**図 1** に示し，以下に説明する．ここで，ジョブ，プロセス，およびスレッドは，ユーザモードで走行するものとする．

(1) サービス処理を実現するジョブは，複数のプロセスから構成される．
(2) プロセスは 1 つ以上のスレッドから構成される．
(3) プロセスは，資源操作の制御単位であり，固有のメモリ空間を保有する．
(4) スレッドは，プロセッサ割当ての単位であり，同じプロセス下ではメモリ空間を共有する．

さらに，並列処理のために，

(5) 同じプロセス内に多数のスレッドが存在する．

と仮定する．このようなシステムモデルは，多くの並列処理で見られるシステム形態である．

3.2 機 能

一括システムコール機能とは，ユーザプログラムがカーネルプログラムに処理を依頼する際，依頼のたびにシステムコールを発行せず，複数の依頼をまとめて

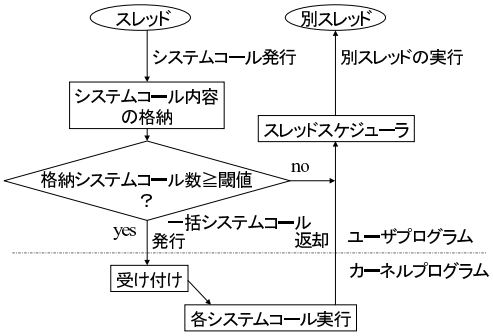


図 2 一括システムコールの処理流れ
Fig. 2 Flowchart of the wrapped system-call.

1 つのシステムコール（一括システムコール）として発行する機能である．つまり，複数のシステムコールをまとめて一括システムコールとして 1 つのシステムコールにより，処理を依頼する．

一括システムコールの処理の流れを**図 2** に示し，以下に説明する．

(1) スレッドがシステムコールを発行すると，その内容を格納域に格納する．
(2–A) 格納したシステムコール数が閾値以上の場合，一括システムコールを発行する．一括システムコールは，カーネルプログラムで受け付けられ，各システムコールが実行され，実行結果が返却される．この返却を契機にスレッドスケジューラが呼ばれる．
(2–B) 格納したシステムコール数が閾値未満の場合，スレッドスケジューラを呼び出す．
(3) スレッドスケジューラは，次に実行するスレッドを選定し制御を移行する．

なお，図 2 には示していないが，スレッドスケジューラにおいて実行可能なスレッドが存在しない場合にも，すでに格納したシステムコール内容を一括システムコールとして発行する．

このように，一括システムコールは，同じプロセス内の各スレッドが発行する各システムコールをまとめて発行する．ただし，一括システムコールとして，各システムコールをまとめる際には，以下の点に留意する必要がある．

(1) システムコールは，処理の性質から大きく 3 つに分類できる．1 つ目は，キーボード入力待ちやセマフォ待ちのように，待ちを解除される時間が予測不可能なシステムコールである．2 つ目は，磁気ディスク装置へのデータ書き出しのように，待ちを解除されることが明らかなシステムコールある．3 つ目は，メッセージ送信や

セマフォ送信のように，待ちが発生しないシステムコールである．一括システムコールでは，まとめた各システムコールの処理がすべて終了して制御が戻ってくる（同期型）ため，基本的には，待ちが発生しないシステムコールのみをまとめる必要がある．そうしないと，処理の応答時間の長大化を抑制できない．待ちを解除されることが明らかなシステムコールも一括システムコールとしてまとめて発行できるが，待ち時間が処理の応答時間に悪影響を与えてしまう．なお，一括システムコールそのものを非同期型で実現する方法も考えられるが，利用インタフェースや実現機構が複雑化してしまう問題がある．

(2) 1つのスレッドが発行する複数のシステムコールをまとめる場合がある．たとえば，スレッドAが一括処理するシステムコールを発行し，スレッドスケジューラが別スレッドとしてスレッドB実行し，スレッドBが一括処理するシステムコールを発行したとき，スレッドスケジューラが別スレッドとしてスレッドAを実行した場合である．このような場合には，1つのスレッドが発行する複数のシステムコール間で依存関係がないことが必要である．システムコール間の依存関係をコンパイル段階で把握し，うまく一括システムコールを利用する方法が考えられる．しかし，依存関係を完全に把握することは難しい．このため，最も簡単な方法として,「一括システムコールとしてまとめるシステムコールは，1つのスレッドあたり最大1つのシステムコールとする」方法がある．この場合，一括処理するシステムコールを発行したスレッドはブロックされ，一括システムコールの閾値は，せいぜい1つのプロセス内のスレッド総数である．

3.3 特　　徴

一括システムコールは，2章で示した要求に対し，(要求2)と(要求4)を満足することができる．また，システムコールとして同期型とすることで,(要求1)を満足することができる．しかしながら，同期型の場合，システムコールを発行してから結果を受け取るまでの時間（これをシステムコール応答時間と名付ける）が長くなる．これらについて，以降で説明する．

カーネルプログラムに対し複数のシステムコールをまとめて一括システムコールとして発行するため，システムコールにともなう処理の負荷を削減できる可能性がある．ユーザプログラムがカーネルプログラムに処理を依頼するたびにシステムコールを発行する場合（以降，逐次システムコールの場合と名付ける）と一括システムコールの場合について，システムコール処理を比較する．図2から以下のことが分かる．

(1) ユーザプログラムでは，システムコール内容の格納処理（処理時間Sとする）の分だけ，一括システムコールの場合の負荷が多い．

(2) カーネルプログラムでは，システムコールの受付け処理および返却時の復帰処理（処理時間Aとする）の分だけ，逐次システムコールの場合の負荷が多い．

システムコール内容の格納処理はシステムコール引数の数のメモリ間複写処理が大半であるのに対し，システムコールの受付け・復帰処理は，例外処理を行うため単純ではない．このため，$S < A$ と考えられ，一括システムコールは，逐次システムコールに比べ，システムコール処理の負荷を削減できる．

カーネル呼び出し回数を削減できることは自明である．これにより，命令実行のパイプライン段数が多く，かつメモリキャッシュの大きさも大きい高性能なプロセッサであっても，APとOSカーネルとの効率的な連携が可能である．

システムコールとして同期型とすることで（要求1）を満足できるものの，システムコール応答時間が長くなってしまう．システムコール応答時間は，一括システムコールの発行と判断する閾値に依存する．当然のことながら，閾値が大きくなるほどシステムコール応答時間は長くなる．一方，閾値を大きくすればカーネル呼び出し回数を大きく削減でき，処理量（スループット）の向上を見込める．したがって，サービス処理の要求に合わせて，閾値を設定する必要がある．ユーザプログラムからカーネルプログラムへの処理の依頼が頻繁に発生し，並列に動作可能なスレッドが多い（並列度が高い）場合には，システムコール応答時間よりも処理量（スループット）の向上が大切である．

4. 評　　価

4.1 カーネル呼び出し回数の削減効果

一括システムコールの長所であるカーネル呼び出し回数削減について，その効果があることを明確にするために，システムコール処理時間とプロセッサ種別の関係を明らかにした．

同種であるが動作クロック周波数が異なるプロセッサを搭載した各計算機において，同じLinux（カーネル2.4.0-test10）を走行させgetpidシステムコール

表 2 各プロセッサにおける getpid システムコール処理時間
Table 2 Execution time of getpid for each CPU.

動作クロック数（MHz）	プロセッサ種別	パイプライン段数	実測値（クロック数）	処理時間（マイクロ秒）
1,800	Pentium 4	20	1,552	0.862
1,500	Pentium 4	20	1,544	1.029
466	Celeron	10	302	0.648
450	Pentium III	10	301	0.669
366	Celeron	10	302	0.825
200	MMX Pentium	5	134	0.670

※処理時間は，動作クロック数と実測値（クロック数）から算出

の処理時間を測定した．測定には，プロセッサが持つハードウェアクロックカウンタを用いた．10 回実行し，キャッシュ・ミスの影響がない 2 回目以降の 9 回の値の平均値を測定値とした．なお，9 回の値は，ほとんど同じ値だった．getpid システムコールは，呼び出したプロセスの識別子を返却する非常に簡易な機能のシステムコールであり，その処理の大半は各システムコールに共通な処理と見なすことができる．また，カーネル呼び出しにともなう多数の関数呼び出しや走行モード変更の影響を把握しやすいと考えられる．

実測結果を**表 2** に示す．表 2 において，理想的には，動作クロック数に関係なく処理時間（クロック数）は一定になるはずであり，このようであれば，動作クロック数に比例して処理時間は短くなる．しかし，処理時間（クロック数）は一定ではなく，処理時間（マイクロ秒）は，動作クロック数に比例して短くなっていない．むしろ，動作クロック数が 1,500 MHz や 1,800 MHz では処理時間（マイクロ秒）が増加している．

動作クロック数が最も小さくパイプライン段数の小さい 200 MHz のときの実測値（クロック）が 134 であることから，getpid システムコール処理の実行命令数は最大 134 命令と考えられる．これに対し，各プロセッサのメモリキャッシュは大きいため，命令キャッシュによる影響は小さいと推察できる．一方，パイプライン段数が等しい（10 段）場合の処理時間（クロック数）は，ほぼ同じである．したがって，この現象にはパイプライン段数が大きく影響していると考えられる．そこで，パイプライン段数と処理時間（クロック数）の関係を**図 3** に示す．図 3 より，パイプライン段数が大きくなると処理時間（クロック数）が非常に大きくなることが分かる．また，パイプライン段数が 10 段の場合について，実測値と処理時間（クロック数）を 200 MHz（パイプライン段数は 5）と同じ（134 クロック）と仮定した場合の理想値の関係を**図 4** に示す．理想に比べ，処理時間が 2 倍以上遅いことが分かる．この傾向は，パイプライン段数が大きくなると非常に強くなり，パイプライン段数が 20 段の場合は，理想

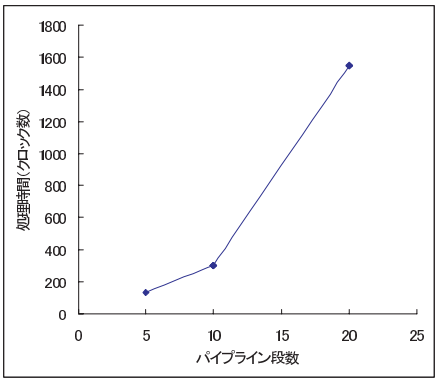


図 3 パイプライン段数と処理時間の関係
Fig. 3 Pipeline depth vs. execution time.

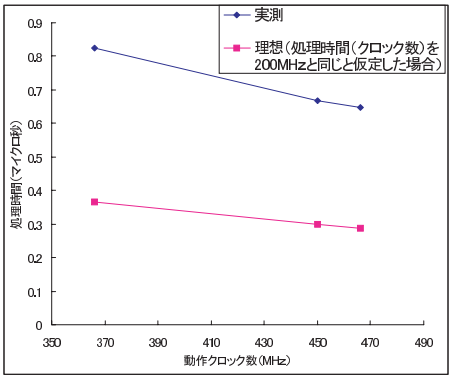


図 4 動作クロック数と処理時間の関係（パイプライン段数 10 の場合）
Fig. 4 CPU clock vs. execution time.

に比べ，処理時間が 10 倍以上遅くなっていることが分かる．

次に，関数呼び出しによるジャンプ命令や走行モード変更の命令がプロセッサ性能へ与える影響の程度について考察する．getpid システムコール処理は，getpid のシステムコール C ライブラリ，OS カーネル内のシステムコール受付け処理，および OS カーネル内の getpid 処理からなる．そこで，関数呼び出しによるジャンプ命令数が getpid システムコール処理と同じであるが，走行モード変更の命令を含まない処理を

APプログラムとして作成し，その処理時間（クロック数）を測定した．その結果，動作クロック数やパイプライン段数の違いによる処理時間（クロック数）の違いはほとんど見られなかった．したがって，関数呼び出しによるジャンプ命令が多いことは，パイプライン段数の増加による性能低下にあまり影響を与えないといえる．

以上のことから，命令実行のパイプライン段数が多い高性能なプロセッサでは，関数呼び出しによるジャンプ命令が多いことを抑制することよりも，走行モード変更をともなうOSカーネル呼び出し処理の多発を抑制することが非常に重要であるといえる．したがって，一括システムコールは，高性能なプロセッサにおいて，ユーザプログラムがカーネルプログラムに処理を依頼する手法としい非常に有効であるといえる．

4.2 実装と評価

4.2.1 実装内容

一括システムコールを並列分散オペレーティングシステムCEFOS（Communication-Execution Fusion Operating System）[7),8)]に実装した．CEFOSは，細粒度マルチスレッドをプロセッサ割当ての基本単位とし，計算機における内部情報処理と通信の融合を目指したOSである．システムは，Pentium系プロセッサを搭載した複数の計算機を高速な通信路で結んで構築され，3.1節に述べたモデルに基づく．なお，プロセス数の増加を抑制し，1つのプロセスを1個以上のスレッド（数百個）から構成する．これにより，複数プロセスによる並列処理ではなく，複数スレッドによる並列処理が可能になり，処理の効率化を目指している．また，各スレッドは，停止することなく連続して走行し，走行最長時間は1ミリ秒程度を想定している．このため，スレッドはシステムコール発行とともに処理を終了する．

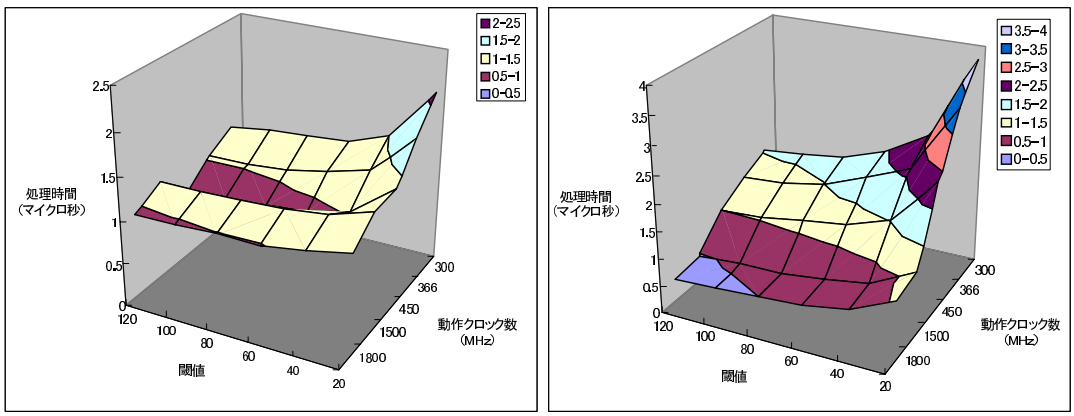
各スレッドのプログラム記述は，逐次システムコールと同じ形式である．コンパイル時に一括システムコールライブラリをリンクすることにより，一括システムコールを利用できる．また，DRD機構[9)]を利用してプロセスとカーネルの共有領域を設け，余計なメモリ間複写を抑制した．一括システムコールライブラリ機能は，主に図2に示した内容である．ただし，3.2節で述べた留意点を考慮し，一括システムコールとしてまとめないシステムコール（システムコールの種類で判断）は，逐次システムコールとして発行している．また，一括システムコールとしてまとめるシステムコールは，1つのスレッドあたり最大1つのシステムコールとしている．

4.2.2 評価

一括システムコールにより，システムコールにともなう処理の負荷を削減できるか否かを明らかにするため，処理の内容が単純なgetpidシステムコールを用いて評価した．

getpidシステムコールをn回繰り返し発行する逐次システムコール処理とgetpidシステムコールn個分をまとめて1回で発行する一括システムコール処理の処理時間を測定した．ここで，nは一括システムコールの処理における閾値と見ることができる．測定には，プロセッサが持つハードウェアクロックカウンタを用いた．

1つのgetpidシステムコール処理時間について，逐次システムコールと一括システムコールの処理時間を**図5**に示す．図5において，横軸は閾値（n），縦軸はプロセッサの動作クロック数，高さ軸は1つのgetpid



（A）逐次システムコール　　（B）一括システムコール

図5 1つのgetpidシステムコール処理時間
Fig. 5 Execution time of single getpid system-call.

システムコール処理時間（実測した処理時間を n で割った値）である．また，処理時間の傾斜を示すため，0.5 間隔で模様分けしている．図 5 から以下のことが分かる．

(1) 逐次システムコール（図 5 (A)）について

(a) いずれの動作クロック数においても，閾値が大きくなると，処理時間は必ず短くなっている．これは，getpid システムコールを繰り返し実行するための処理オーバヘッドが，繰返し回数の増加により，1 つの getpid システムコールあたり小さくなるためである．

(b) 動作クロック数を 300 MHz から 450 MHz に大きくした場合（パイプライン段数 10），あるいは動作クロック数を 1,500 MHz から 1,800 MHz に大きくした場合（パイプライン段数 20）は，動作クロック数が大きいほど処理時間が短くなっている．つまり，パイプライン段数が同じ（10 または 20）場合は，動作クロック数が大きいほど処理時間は短くなる．これは，動作クロック数が大きくなることにより処理が高速化されたためである．

(c) 動作クロック数が 450 MHz（パイプライン段数 10）と 1,500 MHz（パイプライン段数 20）の場合の処理時間を比較すると，1,500 MHz の場合の処理時間が長い．また，1,500 MHz の場合，閾値を大きくしても処理時間が短くなる程度は小さい．これは，パイプライン段数が大きくなるとシステムコール呼び出しによるモード変更の処理時間が増加するためである．つまり，動作クロック数を大きくしてもパイプライン段数が大きくなると処理時間は長くなる可能性があるといえる．

(2) 一括システムコール（図 5 (B)）について

(a) いずれの動作クロック数においても，閾値が大きくなると，処理時間は必ず短くなっている．これは，システムコール呼び出しによるモード変更の処理が 1 回しか発生しないためである．具体的には，1 つの getpid システムコール処理時間には，モード変更処理時間の閾値分の 1 が含まれるためである．

(b) いずれの閾値においても，動作クロック数が大きくなると，処理時間は必ず短くなっている．これは，システムコール呼び出しによるモード変更の処理が 1 回しか発生せず，かつパイプライン段数の違いが小さい（10 と 20）ためである．したがって，閾値が小さい場合は，1 つの getpid システムコール処理時間がモード変更処理の影響を受けやすいため，パイプライン段数を非常に大きくすると動作クロック数を大きくしても処理時間が長くなってしまう可能性がある．

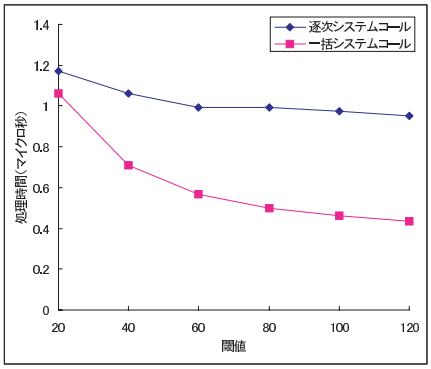


図 6 閾値と処理時間の関係（動作クロック数が 1,800 MHz の場合）

Fig. 6 Threshold vs. execution time (CPU clock = 1,800 MHz).

次に，閾値を変化させた場合の様子として，動作クロック数が 1,800 MHz の場合を**図 6** に示す．処理時間は，1 つの getpid システムコール処理に相当する時間を示している．図 6 より，閾値が小さい場合は，逐次システムコールと一括システムコールの処理時間に大差はないものの，閾値が大きくなると処理時間は非常に異なってくる．これは，システムコール呼び出しによるモード変更の処理の実行回数が，逐次システムコールの場合は変化しないのに対し，一括システムコールでは少なくなるためである．

また，動作クロック数を変化させた場合の様子として，閾値が 20 の場合を**図 7** に示す．処理時間は，1 つの getpid システムコール処理に相当する時間を示している．図 7 より，閾値が 20 と小さい場合でも，動作クロック数 450 MHz（パイプライン段数 10）と動作クロック数 1,500 MHz（パイプライン段数 20）の間でパイプライン段数が変化（10 から 20）し，逐次システムコールの方が一括システムコールより処理時間が長くなっている．つまり，動作クロック数が小さい場合は逐次システムコールの処理時間が短いものの，動

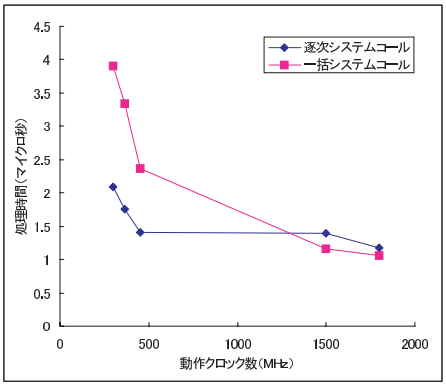


図 7 動作クロック数と処理時間の関係（閾値が 20 の場合）
Fig. 7 CPU clock vs. execution time (threshold = 20).

作クロック数が大きくなりパイプライン段数が大きくなると一括システムコールの処理時間の方が短くなる．なお，3.3 節では，「システムコール内容の格納処理に比べシステムコールの受付け処理および返却時の復帰処理は長い（S < A）と考えられる」と記した．にもかかわらず，動作クロック数 450 MHz 以下では逐次システムコールの処理時間が短い原因は，今回評価に利用した CEFOS は，開発工数を削減するため，Linux をベースにしており，一括システムコールの各システムコール処理をカーネル内で実行する際に Linux カーネルの各システムコール処理を利用しているためである．具体的には，Linux カーネルの各システムコール処理を利用するために，一括システムコールとして格納された各システムコールの形式を，Linux カーネルの各システムコールの呼び出し インタフェースに整合させる処理を行っている．一括システムコールに適合した形式で各システムコール処理を呼び出せるようにすることにより，この問題は解決できると考える．

以上のことから，一括システムコールによりシステムコールにともなう処理の負荷を削減できる効果は，パイプライン段数が大きい高性能なプロセッサほど有効であるといえる．

4.3 実システムでの効果予測

実システムとして，文献 5) に示される端末制御計算機におけるトランザクション処理を取り上げ，一括システムコールの効果を予測する．この端末制御計算機は，MC68020 もしくは MC68040 のプロセッサを搭載したものである．4.1 節や 4.2 節の評価は，Pentium 系プロセッサを基にしている．モトローラ系と Pentium 系では，命令セットに違いがあるため，以降で述べるトランザクション処理を Pentium 系プロセッサで実行した場合，命令数は異なってくる．しかし，1 トランザクションで発行されるシステムコールの数や内容は同じである．この点に注意して，効果予測を行う．具体的には，1 トランザクションの命令数が，モトローラ系プロセッサでは 475522 命令の場合，Pentium 系プロセッサでは 475522 × Pen 命令と換算する．

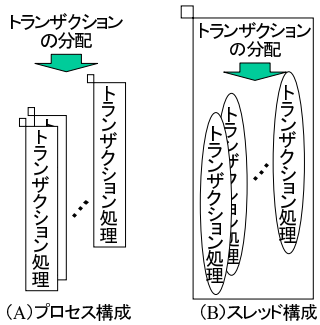


図 8 トランザクション処理の様子
Fig. 8 Transaction assignment.

この端末制御計算機は，文献 5) の図 1 に示されるように，ATM や CD などの端末をネットワークを介して集約して制御し，端末とホスト計算機とのデータ通信を制御するものである．搭載された OS は，トランザクション処理用に開発された TRADE と名付けられた OS である．TRADE は，非同期型のシステムコールを提供しており，様々なシステム構成を実現するための工夫がなされている．また，端末制御計算機上には，TRADE のほかに，業務処理を共通管理するサービス管理モジュールと業務処理を行う業務処理モジュールがある．端末制御計算機に接続される ATM や CD などの端末の数は，構築されるシステムにより，数十から百以上になる．このため，サービス管理モジュールと業務処理モジュールからなる実行プログラムを作成し，そのプログラムから，接続された端末数と同じプロセスの数だけプロセスを生成し処理を制御する．つまり，各プロセスは同じプログラムから生成されている．この様子を**図 8** (A) に示す．また，トランザクション処理の内容を以下に説明する．

(1) 1 つのトランザクション処理は，定型の処理である．
(2) トランザクションを複数並列に実行するため，同じプログラムによるプロセスが数十から百以上存在する．

また，システム構築を支援するため，トランザクション処理に関する多くのデータを収集しており，典型的な 1 トランザクションは以下の特徴を有する．

(1) 1 トランザクションの命令数は，475522 命令で

表 3 1トランザクションで発行されるシステムコールの種類と回数
Table 3 System-calls emitted in a typical transaction.

システムコール名	機能（利用条件）	回数	性質
osgtpbf	メモリ域の確保	12	待ちが発生しない（○）
osfrpbf	メモリ域の開放	12	待ちが発生しない（○）
ossenmsp	メッセージの送信	4	待ちが発生しない（○）
osrcvmspr	メッセージの受信	4	待ちを解除される時間が予測不可能（×）
osanyrl	非同期要求の受取り	5	待ちを解除される時間が予測不可能（×）
oscanrq	非同期要求の取消し	2	待ちが発生しない（○）
ossigsp	セマフォの送信	6	待ちが発生しない（○）
oswaitsp	セマフォの受信	6	待ちを解除される時間が予測不可能（×）
ossttim	インターバルタイマの設定	2	待ちが発生しない（○）→（×）
osgtdat	時刻の取得	3	待ちが発生しない（○）→（×）
osdelay	指定時間の待ち	2	待ちを解除されることが明らか（△）
osreadd	通信回線からのデータ受信	1	待ちを解除される時間が予測不可能（×）
oswrited	通信回線へのデータ送信	4	待ちを解除されることが明らか（△）
osiioctl	通信回線の状態制御	1	待ちを解除されることが明らか（△）
osioctl	通信回線の状態取得	2	待ちを解除されることが明らか（△）
oswritd	ファイルへの書き込み	1	待ちを解除されることが明らか（△）
osflush	ファイル書き込み用のバッファ内容を実出力	1	待ちを解除されることが明らか（△）
合計		68	

ある．

(2) 1トランザクションで発行されるシステムコール数は，68個である．なお，システム開発初期では，116個のシステムコールが発行されたが，システムコール発行のオーバヘッドを削減するため，2つ以上のシステムコール機能を複合したシステムコールの提供などにより，発行回数を削減した．

(3) 1トランザクションで発行される68個のシステムコールの種類と回数を**表3**に示す．表3の性質に示すように，通信回線からのデータ受信待ちやセマフォ待ちのように，待ちを解除される時間が予測不可能なシステムコールは16個である．磁気ディスク装置へのデータ書き出しのように，待ちを解除されることが明らかなシステムコールは11個である．メッセージ送信やセマフォ送信のように，待ちが発生しないシステムコールは41個である．

次に，使用する計算機のプロセッサをMMX200 MHzから1,500 MHzに変更したと仮定した場合について，逐次システムコールの影響を考察する．ここで，両プロセッサとも平均Cクロック/命令（RISCでは，大半の命令は$C=1$であるが，$C>1$の命令もあるため，このように仮定した）とすると，MMX200 MHzの場合，1トランザクションのクロック数は$475522\times Pen\times C$クロックである．一方，getpidシステムコールの処理の大半は，68個の各システムコールに共通な処理相当と見なすことができる．この部分のクロック数は，表2より，MMX200 MHzの場合134×68クロックであり，1,500 MHzの場合1544×68クロックである．したがって，1,500 MHzの場合，1トランザクションのクロック数は475522×Pen×C−(134×68)+(1544×68)クロックとなる．以上のことから，MMX200 MHzの場合に比べ，1,500 MHzの場合は，1トランザクションのクロック数が95880クロック（−(134×68)+(1544×68)クロック）増加する．つまり，1,500 MHzの場合，たとえば，$C=1$とすると1トランザクションのクロック数が約20%（$95880/(475522\times Pen)\times 100\%$，$Pen=1$）も増加し，C=1.5としても1トランザクションのクロック数が約13%も増加する．このため，クロック数の向上に見合った性能向上が得られないといえる．

最後に，1,500 MHzのプロセッサの計算機について，一括システムコールの効果を予測する．予測に際し，以下の仮定を行う．

(1) 1つのトランザクション処理は定型の処理であるため，並列実行可能な実行形態をプロセスではなくスレッドで実現する．この様子を図8(B)に示す．なお，文献5)に示すシステムの第1段階開発着手時（昭和60年頃）は各トランザクション処理間の保護などの理由からスレッドは導入されなかったため，この実システムでは，図8(A)の処理の様子で実現された．並列実行可能な実行形態をプロセスではなくスレッドとすることにより，プロセス切替えに比べスレッド切替えはオーバヘッドが少ないため，処理の効率化が図れる．しかし，ここでは，この観点ではなく，一括システムコール利用の観点から評価する．

(2) 同時走行スレッド数を n とする．
(3) 1トランザクション処理が発行するシステムコールの α% が一括システムコールとしてまとめて発行できるとする．

これらの仮定により，各スレッドが 1 トランザクションを処理するとき，一括システムコールとして発行できるシステムコールの数は，平均 $n \times \alpha/100$ 個である．ただし，3.2 節で述べたように，実行可能なスレッドが存在しない場合にも一括システムコールを発行するものの，ここでは，予測を簡単化するため，このようなことは起こらないと仮定する．ここで，表 3 に示した 1 トランザクションで発行されるシステムコールの種類，回数，および性質から，待ちが発生しないシステムコール 41 個が一括システムコールとしてまとめて発行できる．しかし，待ちが発生しないシステムコール 41 個の中で，時間や時刻を扱うシステムコール 5 個（ossttim と osgtdat）は，一括システムコールとしてまとめるとその精度が劣化するため，好ましくない．そこで，$\alpha = (41-5)/68 \times 100$ である．なお，待ちを解除されることが明らかなシステムコール 9 個も一括システムコールとしてまとめて発行できるが，待ち時間がトランザクションの応答時間に悪影響を与えるため除外する．また，プロセスが数十から百以上存在するため，これらをスレッドに置き換え，ここでは $n = 100$ とすると，一括システムコールとして発行できるシステムコールの数は約 53 個（$100 \times (36/68 \times 100)/100$ 個）である．一方，図 5 に示した測定では，1,500 MHz で閾値 60 の場合，1 つの getpid システムコール処理時間は，逐次システムコールが 1.169 μ秒，一括システムコールが 0.612 μ秒であった．また，この値は，閾値が 53 の場合もほぼ同様である．ここで，getpid システムコールの処理は，すべてのシステムコールに共通な処理相当と考えられる．このため，100 個のスレッドが行う 100 トランザクション処理で発行される 6,800 個のシステムコールの共通な処理に相当する時間は，すべて逐次システムコールによる場合 7,949 μ秒，53 個（$\fallingdotseq$ 閾値 60）を一括システムコールとし残りを逐次システムコールとする場合 5,677 μ秒となる．したがって，一括システムコールを利用することにより 2,272 μ秒/100 トランザクションの短縮が可能である．つまり，$C = 1$ とすると，1 トランザクションのクロック数は 571402 クロック（$475522 \times \mathrm{Pen} + 95880$ クロック，$\mathrm{Pen} = 1$）であり，380.93 μ秒/トランザクションとなる．これらにより，約 6%（$2272/(380.93 \times 100) \times 100\%$）の性能向上が一括システムコールにより得られると推察できる．

なお，ここでの効果予測は，1,500 MHz（パイプライン段数 20）について行った．今後，プロセッサを高性能化するために，動作クロック数を高くしパイプライン段数を大きくすると，一括システムコールによる性能向上がさらに期待できる．

4.4 効果の定式化

4.3 節に述べた実システムによる効果予測を基に，パイプライン段数の増加をともなう高い動作クロック数が招く性能低下，および一括システムコールの効果を定式化する．

定式化のために，以下を仮定する．

Dh：動作クロック数 h における
　システムコール共通処理の必要クロック数
P：システムコール共通処理の命令数
I：1 トランザクションの命令数
S：1 トランザクションで発行される
　システムコール数
C：1 命令の平均クロック数

最初に，パイプライン段数の増加をともなう高い動作クロック数が招く性能低下を定式化する．先に示した仮定により，動作クロック数 h における 1 トランザクション処理のクロック数（Ah）は，以下のようになる．

$$Ah = C \times (I - S \cdot P) + Dh \cdot S \quad (1)$$

したがって，$h1 < h2$ のとき，$h2$ では以下に示す割合の性能低下が発生する．

$$(Ah2 - Ah1)/Ah1 = \{(Dh2 - Dh1)S\}/ \{C \times (I - S \cdot P) + Dh1 \cdot S\} \quad (2)$$

式 (2) が示すように，性能低下は，システムコール共通処理の必要クロック数の差分（$Dh2 - Dh1$）に比例する．

次に，一括システムコールが有効に働く閾値について定式化する．先に示した仮定により，システムコール共通処理のクロック数は，以下のようになる．

逐次システムコールの場合：

$$Dh \quad (3)$$

一括システムコールの場合：

$$(Dh + tN)/N \quad (4)$$

ここで，t は図 2 に示したシステムコール内容の格納の処理に必要なクロック数，N は閾値である．一括システムコールが有効である場合は式 (3) に比べ式 (4) が小さい場合であるから，以下の式を満足する閾値（N）で一括システムコールの処理を行う必要がある．

$$N > Dh/(Dh - t) \quad (5)$$

もちろん，$Dh > t$ であることが必須である．また，

スレッド数を n とすると，N は下式を満足する必要がある．ここで，α は 1 トランザクションで発行されるシステムコールの中で一括システムコールとしてまとめて発行できる割合（%）である．

$$N \leq n\alpha/100 \tag{6}$$

式 (5) が示すように，一括システムコールで行うシステムコール内容の格納の処理に必要なクロック数を小さくすることで，閾値を大きくできる．ただし，式 (6) が示すように，並列に動作する処理の相関や実行形態により制限を受ける．

最後に，一括システムコールがトランザクション処理に与える効果の程度について定式化する．1 トランザクションのクロック数は，全システムコールを逐次システムコールで発行する場合は，式 (1) で表せる．一方，全システムコールの α% を一括システムコールで発行する場合は，以下のようになる．

$$\begin{aligned} Ah' = \quad & C \times (I - S \cdot P) + \\ & (1 - \alpha/100) \times Dh \cdot S + \\ & \alpha/100 \times ((Dh + tN)/N) \end{aligned} \tag{7}$$

したがって，一括システムコールを利用することによる性能向上の程度（%）は，下式のようになる．

$$\begin{aligned} & (Ah - Ah')/Ah \times 100 = \\ & \quad \alpha S \times \{Dh(1 - 1/N) - t\}/ \\ & \{C \times (I - S \cdot P) + Dh \cdot S\} \times 100 \end{aligned} \tag{8}$$

式 (8) が示すように，1 トランザクションで発行されるシステムコールの中で一括システムコールとしてまとめて発行できる割合（α）や閾値（N）が大きいほど，一括システムコールを利用することによる性能向上の程度は大きい．

先の測定では，$D1500Mhz = 1544$，$D200Mhz = 134$，$I = 475522$，$S = 68$，であり，多くの場合 P は 100 命令前後である．なお，TRADE では 80 命令強であった．4.3 節の前半は，式 (2) を利用でき，P を 100 とすると $CI \gg (Dh1 - CP)S$ と考えられ，先に記述した結果を得る．4.3 節の後半では，式 (8) を利用して同様な結果を得ることができる．

5. ま と め

OS カーネルの処理を多く含む並列処理を効率的に実行できる機能として，ユーザプログラムがカーネルプログラムに処理を依頼する際，依頼のたびにシステムコールを発行せず，複数の依頼をまとめて 1 つのシステムコールとして発行する一括システムコール機能を提案した．この機能は，システムコールにともなう処理の負荷を軽減でき，かつカーネル呼び出し回数を削減できる特徴を持つ．また，この機能の実現例により，プロセッサの動作クロック数と一括システムコール機能の有効性の関係を示し，一括システムコール機能は，パイプライン段数が大きい高性能なプロセッサほど有効であることを明らかにした．さらに，銀行系システムにおける端末制御計算機のトランザクション処理の実システムを取り上げた効果予測では，一括システムコールにより約 6%の性能向上が見込まれることを示した．

今後，プロセッサを高性能化するために，パイプライン段数の増加といったような実行命令順序の連続性を仮定するプロセッサ・アークテクチャが増加するほど，一括システムコール機能は有効な機能になるといえる．

残された課題として，種々のトランザクション処理モデルにおいて一括システムコールが有効に働く閾値と応答時間の検討，およびシステムコールの要求内容あるいはシステムコールを要求したスレッドの優先度に基づくシステムコール実行スケジュール法の検討がある．

謝辞 カーネル呼び出し回数の削減効果を明らかにするため，測定に協力いただいた九州大学大学院システム情報科学府の棚林拓也君および九州大学工学部電気情報工学科の松尾隆利君に感謝します．

本研究は，通信・放送機構の創造的情報通信技術研究開発推進制度に係る研究開発課題「次世代型インテリジェント・マルチメディア情報通信網の基盤技術に関する研究」による．

参 考 文 献

1) 佐藤三久，原田 浩，長谷川篤史，石川 裕：Cluster-enabled OpenMP：ソフトウェア分散共有メモリシステム SCASH 上の OpenMP コンパイラ，情報処理学会論文誌：ハイパフォーマンスコンピューティングシステム，Vol.42, No.SIG9(HPS3), pp.158–169 (2001).
2) Frigo, M., Leiserson, C.E. and Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language, *ACM SIGPLAN Notices*, Vol.33, No.5, pp.212–223 (1998).
3) Bacon, D.F., Graham, S.L. and Sharp, O.J.: Compiler transformations for high-performance computing, *ACM Computing Surveys*, Vol.26, No.4 (1994).
4) Ritchie, D.M. and Thompson, K.: The UNIX Time-sharing System, *The Bell System Technical Journal*, Vol.57, No.6, pp.1905–1929 (1978).
5) 箱守 聰，谷口秀夫：高負荷オンライントランザクション処理を実現するオペレーティングシステム，信学論（D-I），Vol.J84-D-I, No.6, pp.627–

638 (2001).
6) Maggio, M.D. and Krumme, D.W.: A Flexible System Call Interface for Interprocessor Communication in a Distributed Memory Multicomputer, *SIGOSR*, Vol.25, No.2, pp.4–21 (1991).
7) Amamiya, M., Taniguchi, H. and Matsuzaki, T.: An Architecture of Fusing Communication and Execution for Global Distributed Processing, *Parallel Processing Letters*, Vol.11, No.1, pp.7–24 (2001).
8) 日下部茂，富安洋史，村上和彰，谷口秀夫，雨宮真人：並列分散オペレーティングシステム CEFOS（Communication-Execution Fusion OS），信学技報，Vol.99, No.251, pp.25–32 (1999).
9) 谷口秀夫：新しい OS カーネル内外連携機構：DRD 機構の提案，信学論（D-I），Vol.J85-D-I, No.2, pp.193–201 (2002).

(平成 14 年 6 月 6 日受付)
(平成 14 年 10 月 1 日採録)

谷口　秀夫（正会員）

昭和 53 年九州大学工学部電子工学科卒業．昭和 55 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．昭和 62 年同所主任研究員．昭和 63 年 NTT データ通信（株）開発本部移籍．平成 4 年同本部主幹技師．平成 5 年九州大学工学部助教授．平成 8 年九州大学大学院システム情報科学研究科助教授．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「オペレーティングシステム」（昭晃堂）等．電子情報通信学会，日本ソフトウェア科学会，ACM 各会員．博士（工学）．

日下部　茂（正会員）

昭和 41 年生．平成元年九州大学工学部情報工学科卒業．平成 3 年同大学大学院総合理工学研究科情報システム学専攻修士課程修了．同年より同専攻助手．平成 10 年より九州大学大学院システム情報科学研究科情報工学専攻助教授．平成 12 年マサチューセッツ工科大学計算機科学科客員研究員．関数型言語，細粒度並列言語処理系と実行時システムの研究に従事．電子情報通信学会，ACM，IEEE 会員．博士（工学）．

中山　大士

昭和 52 年生．平成 12 年九州大学工学部電気情報工学科卒業．平成 14 年同大学大学院システム情報科学府知能システム学専攻修士課程修了．オペレーティングシステムに興味を持つ．現在（株）東芝にてミドルウェアの開発に従事．

乃村　能成（正会員）

昭和 44 年生．平成 5 年九州大学工学部電子工学科卒業．平成 7 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．平成 8 年九州大学大学院システム情報科学研究科助手．オペレーティングシステムとソフトウェア開発環境，グループ支援環境に興味を持つ．博士（情報科学）．

雨宮　真人（正会員）

昭和 17 年生．昭和 42 年九州大学工学部電子工学科卒業．昭和 49 年同大学大学院工学研究科修士課程修了．同年日本電信電話公社武蔵野電気通信研究所入所．以来，プログラミング言語・処理系，自然言語理解，データフロー・アーキテクチャ，並列処理，関数型/論理型言語，知能処理アーキテクチャ等の研究に従事．現在，九州大学大学院システム情報科学研究院知能システム学部門教授．電子情報通信学会，ソフトウェア科学会，人工知能学会，IEEE，ACM，AAAI 各会員．工学博士．