

# OS処理の分散実行を効果的に利用できる 一括処理依頼機能の実現と評価

佐藤 将也<sup>1,a)</sup> 谷口 秀夫<sup>1</sup> 村岡 勇希<sup>1</sup> 山内 利宏<sup>1</sup>

受付日 2018年5月7日, 採録日 2018年11月7日

**概要:** マイクロカーネル OS では, OS 機能の一部を OS サーバとして実現している. このため, 複数プロセッサ環境において OS サーバを分散配置することで, OS 処理を分散できる. しかし, AP プロセスから OS サーバへの処理依頼は, OS サーバ間の通信を複数回行うことで実行されるため, 応答時間の向上は望めない. また, AP プロセスから OS サーバへの処理依頼は, プログラム記述が簡明で利便性が高いことや低処理オーバーヘッドであることから, 多くの場合, 完了型のインタフェースである. しかし, 完了型のインタフェースであるために, 複数の処理が分散実行可能な処理内容であり, かつ各処理を依頼する OS サーバが異なっても, 逐次実行せざるをえない. そこで, 本論文では, AP プロセスが完了型のインタフェースで一度に複数の処理依頼を OS サーバに行える一括処理依頼機能を提案する. また, 基本性能を明らかにするとともに, 分散実行によりサービスが複数処理からなる場合に応答時間を短縮できることを示す.

**キーワード:** 分散処理, マイクロカーネル OS, マルチコアプロセッサ

## Implementation and Evaluation of Batch Processing Request for Leveraging Distributed Execution of OS Processing

MASAYA SATO<sup>1,a)</sup> HIDEO TANIGUCHI<sup>1</sup> YUUKI MURAOKA<sup>1</sup> TOSHIHIRO YAMAUCHI<sup>1</sup>

Received: May 7, 2018, Accepted: November 7, 2018

**Abstract:** In microkernel operating systems (OSes), some parts of OS functions are implemented as processes, which called OS server. For this reason, OS functions can be distributed by placing OS servers to multiple processors. However, it is difficult to reduce the response time of a processing request from an application program (AP) to an OS server. This is due to invocation of multiple inter server communication for processing requests. In addition, an interface of processing requests from AP to OS servers is blocking in most cases. Hence, a processing request is forced to be done successively even though multiple processing are concurrently executable and related OS servers are independent. In this paper, we propose a batch process request function with blocking interface to request multiple processing to OS servers at one time. We also present evaluation results of basic performance, and distributed processing by the proposed function can reduce the response time for a service consists of multiple processing.

**Keywords:** distributed processing, microkernel operating system, multicore processor

### 1. はじめに

計算機性能の向上にとともに, 提供されるサービスは高度化し, 多様化している. また, サービスの提供を支えるシステムは複雑化している. このため, 多様なサービ

スを支える高い適応性と複雑化したシステムによる不具合に耐え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) が求められている. 高い適応性と堅牢性を有する OS のプログラム構造としてマイクロカーネル構造 [1], [2], [3], [4], [5], [6], [7], [8] がある.

マイクロカーネル OS は, OS 機能をカーネルと OS サーバで実現している. このため, AP プロセスは, サーバプログラム間通信を利用して処理を OS サーバに依頼する.

<sup>1</sup> 岡山大学大学院自然科学研究科  
Graduate School of Natural Science and Technology,  
Okayama University, Okayama 700-8530, Japan

<sup>a)</sup> sato@cs.okayama-u.ac.jp

また、マイクロカーネル OS では、OS サーバをプロセスごと分散させることで、OS 処理を分散できる。特に、マルチコアプロセッサ環境では、各コアに OS サーバを分散配置して実行することで、処理スループットの向上を図れる。

しかし、AP プロセスから OS サーバへ依頼された処理は、OS サーバ間の通信を複数回行うことで実行されるため、応答時間の向上は望めない。むしろ、OS サーバ間の通信がコア間通信を含むため低下してしまう。

一方、AP プロセスから OS サーバへの処理依頼は、多くの場合、完了型のインタフェースである。これは次の理由による。完了型のインタフェースでは、処理依頼、結果待ち、および結果取得を 1 回の OS サーバ呼び出しでできる。このため、プログラム記述が簡明で利便性が高く、かつ低処理オーバーヘッドである。これに対し、非完了型のインタフェースでは、処理依頼と結果取得が別インタフェースであるため、2 回の OS サーバ呼び出しが必要である。このため、処理依頼の結果取得を待たずに別の処理依頼をできるもの、AP プロセスのプログラム記述は複雑化し、かつ処理オーバーヘッドも大きい。

しかし、完了型のインタフェースであるために、AP プロセスは、複数の処理が分散実行可能な処理内容であり、かつ各処理を依頼する OS サーバが異なっても、逐次依頼せざるをえない。AP プロセスからマルチスレッドで処理依頼を行う方法も考えられるもの、プログラム記述が複雑化するうえに、モノリシックカーネルに比べ、マイクロカーネルでの利用は難しい。

そこで、本論文では、AP プロセスが、完了型インタフェースで一度に複数の処理依頼を OS サーバに行える一括処理依頼機能を提案する。具体的には、マイクロカーネル構造を持つ OS として、**AnT** オペレーティングシステム (An operating system with adaptability and tough-ness) (以降、**AnT**) を取り上げ、一括処理依頼機能について述べる。また、基本性能を明らかにするとともに、分散実行によりサービスが複数処理からなる場合に応答時間を短縮できることを示す。

なお、**AnT** は、マルチコア環境で複数の OS サーバを同時起動し、OS 処理を分散して実行することにより高スループットを実現する OS 処理分散機能 [9], [10] を有する。当然のことながら、OS サーバは、非完了型のインタフェースで処理を別の OS サーバに依頼でき、複数の処理を実行できる。

## 2. **AnT** のサーバプログラム間通信

### 2.1 複写レスデータ授受機能

マイクロカーネル構造を有する **AnT** は、プロセス間の通信を高速化するため、コア間通信データ域 (ICA: Inter-core Communication Area) を利用した複写レスデータ授受機

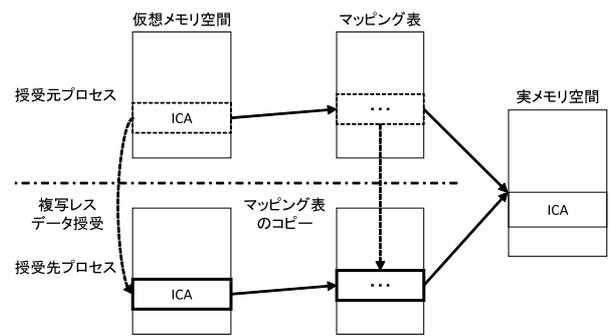


図 1 複写レスデータ授受

Fig. 1 Copyless data transfer.

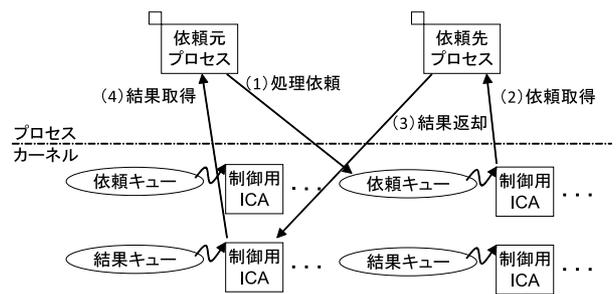


図 2 サーバプログラム間通信の基本機構

Fig. 2 Basic architecture of inter server program communication.

能を持つ。プロセス間の複写レスデータ授受の様子を図 1 に示す。ICA の特徴として、以下の 3 つがある。

- (1) ページ (4KBytes) を単位とする  $n$  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 仮想空間間での領域の貼り替え

ICA は、ページを最小単位として管理する領域であり、ICA へのアクセスは、プロセスごとの仮想空間のマッピング表を介して行われる。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でのデータ授受は、授受するデータを格納した ICA (以降、データ用 ICA と呼ぶ) 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われる。これらの操作をまとめて、ICA の貼り替えと呼ぶ。すなわち、通信する 2 つのプロセス間におけるデータ授受を 2 仮想空間の間における ICA の貼り替えにより実現することで、複写レスデータ授受を実現する。

### 2.2 高速なサーバプログラム間通信機構 [11]

**AnT** は、複写レスデータ授受機能を利用した高速なサーバプログラム間通信機構を有する。カーネルは、すべてのプロセスに対し、通信制御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用意する。サーバプログラム間通信の基本機構を図 2 に示し、以下で説明する。

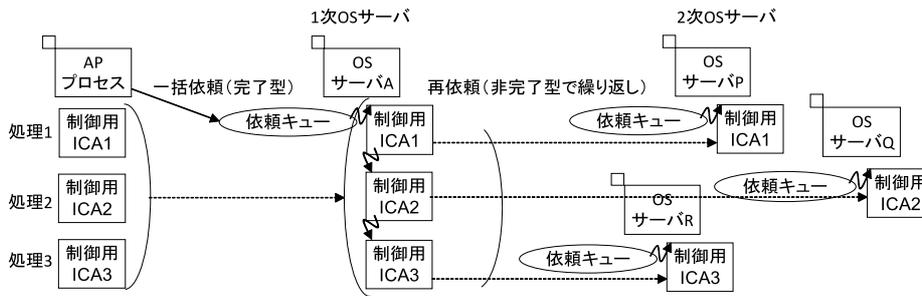


図 3 一括処理依頼機能

Fig. 3 Batch processing request function.

- (1) 依頼元プロセスが処理依頼を行うと、カーネルは、依頼先プロセスの依頼用リングバッファに依頼内容（以降、依頼情報と呼ぶ）を格納した制御用の ICA（以降、制御用 ICA と呼ぶ）を登録し、依頼先プロセスに貼り替える。
- (2) 依頼先プロセスは、依頼用リングバッファから依頼情報を格納した制御用 ICA を取得し、依頼された処理を実行する。
- (3) 依頼先プロセスが結果返却を行うと、カーネルは、依頼元プロセスの結果用リングバッファに処理の結果（以降、結果情報と呼ぶ）を格納した制御用 ICA を登録し、依頼元プロセスに貼り替える。
- (4) 依頼元プロセスは、結果用リングバッファから結果情報を格納した制御用 ICA を取得し、処理を終了する。

また、完了型と非完了型の通信インタフェースを同様な形式で提供し、両インタフェースを選択して利用できる。マイクロカーネル構造 OS では、OS サーバを多段に経由して処理依頼が発行される。これを多段依頼と名付ける。このとき、処理依頼のたびに制御用 ICA を確保すると、処理オーバーヘッドが大きい。そこで、経由する OS サーバ間で 1 つの制御用 ICA を使いまわし、依頼情報を積み重ねることで、オーバーヘッドを削減する。また、多段依頼の結果返却では、積み重ねられた情報をもとに、中継した OS サーバを経由する逐次的な返却が行われる。しかし、必ずしも逐次的な返却を行う必要がない場合、依頼元プロセスへ直接に返却することにより、処理を高速化する。具体的には、制御用 ICA に flag を設け、返却の可否を設定する。結果返却時にカーネルが flag を確認し、返却が必要な依頼元プロセスに直接返却を行う。

### 3. 一括処理依頼機能

#### 3.1 問題と対処

AP プロセスから OS サーバへの利用インタフェースは、完了型インタフェースである。完了型インタフェースでは、処理依頼、結果待ち、および結果取得を 1 回の OS サーバ呼び出しでできる。このため、プログラム記述が簡明で利便性が高く、かつ低処理オーバーヘッドである。

しかし、完了型インタフェースであるために、AP プロ

セスは、複数の処理が分散実行可能な処理内容であり、かつ各処理を依頼する OS サーバが異なっても、逐次依頼せざるをえない。このため、複数処理からなるサービスの応答時間は、各コアに分散配置された OS サーバ間でコア間通信を含むサーバ間の通信を複数回含んで実行される影響も受け、低下してしまうという問題がある。

上記問題を解決する対処として、AP プロセスが、完了型インタフェースで一度に複数の処理依頼を OS サーバに行える一括処理依頼機能を提案する。これにより、OS サーバ群は、一度に複数の処理依頼を受け取り、分散実行できる。つまり、本機能により、マルチコアプロセッサ環境において、AP プロセスは各コアに分散配置された OS サーバを効果的に利用できる。これにより、AP プロセスから OS サーバへの複数の処理依頼を必要とする処理において、サービス応答時間の短縮が期待できる。

#### 3.2 基本機構

一括処理依頼機能とは、一度に複数の処理依頼を行える機能である。これにより、AP プロセスが依頼した処理の処理時間を短縮できる。本機能の様子を図 3 に示す。AP プロセスは、3 つの処理を依頼する場合、制御用 ICA を 3 個確保する。次に、3 つの処理の依頼情報を制御用 ICA1, 2, および 3 へそれぞれ格納し、一度に OS サーバ A（1 次 OS サーバと呼ぶ）へ処理依頼を行う。OS サーバ A は、これらの制御用 ICA を非完了型インタフェースで OS サーバ P, Q, および R（これらを 2 次 OS サーバと呼ぶ）へ処理依頼する。OS サーバ A は、非完了型インタフェースで処理依頼を行うため、AP プロセスが依頼した処理は、分散実行される。

#### 3.3 制御機構

本機能を実現する制御機構は、既存のサーバプログラム間通信機構への変更を最小化かつ局所化して実現する。

まず、既存のサーバプログラム間通信機構を利用して、AP プロセスが一度に 1 つの処理を依頼する処理流れを図 4(A) に示し、以下に説明する。

- (1) AP プロセスは、処理依頼を行うために制御用 ICA を

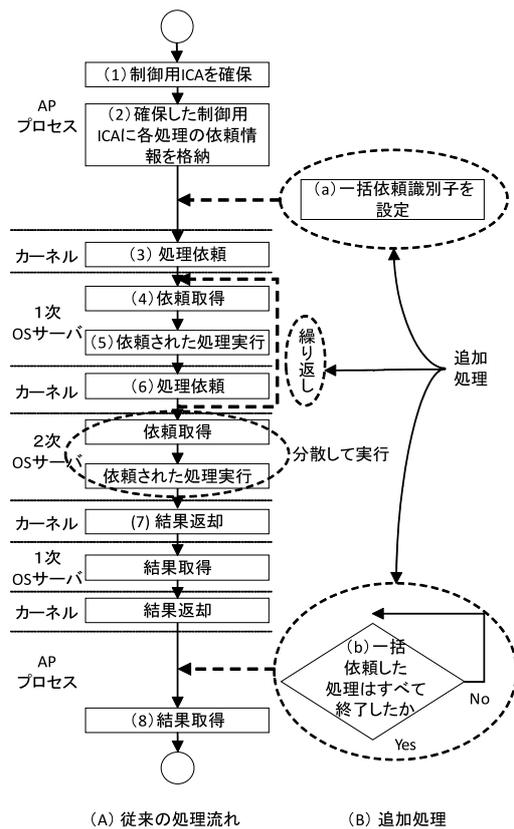


図 4 一括処理依頼機能の処理流れ

Fig. 4 Processing flow of batch processing request function.

確保する。

(2) AP プロセスは、確保した制御用 ICA に各処理の依頼情報を格納する。

(3) AP プロセスは、処理依頼のシステムコールを発行し、カーネルに処理が切り替わる。カーネルは、1次 OS サーバの依頼キューに制御用 ICA を登録する。

(4) 1次 OS サーバは、依頼キューから制御用 ICA を取得する。

(5) 1次 OS サーバは、依頼された処理を実行する。

(6) 1次 OS サーバは、依頼された処理を実行するために2次 OS サーバへ処理依頼を行う。

(7) 2次 OS サーバは、依頼取得し、処理実行し、1次 OS サーバへ結果返却を行う。1次 OS サーバは、結果を取得し、AP プロセスへ結果返却を行うために、AP プロセスの結果キューに制御用 ICA を登録する。

(8) AP プロセスは、処理の結果を取得する。

既存のサーバプログラム間通信機構を利用する場合は、(2) から (8) までを繰り返し、複数の処理を実行する。

上記の処理にいくつかの新しい処理を追加することで一括処理依頼機能を実現する。なお、複数の処理結果を取得する方法として、一括して1回で取得する方法(一括結果取得)と個別に複数回取得する方法(個別結果取得)がある。結果取得の回数が増えるとオーバーヘッドが大きくなるため、一括結果取得を実現する。

表 1 インタフェースの機能と形式

Table 1 Function and format of ncallsync() interface.

機能	形式	戻り値
一括依頼	ncallsync(n, p); n: 処理依頼する制御用 ICA の個数 p: 制御用 ICA リストの先頭の制御用 ICA のアドレス	成功: 0 失敗: -1

本機能を実現するために、1つのプロセスが一括依頼した処理依頼の数を管理し、制御する一括依頼識別子を導入する。この識別子は、制御用 ICA に設定する。この識別子の値は、依頼した処理の数(つまり、制御用 ICA の数)を示す。追加する処理を以下に述べる。

(a) 制御用 ICA の一括依頼識別子に値を設定する。この処理は、依頼元プロセス(図4では AP プロセス)が行う。  
(b) 一括結果取得の処理である。具体的には、各処理の結果返却が行われた際に、返却された結果の数が一括依頼識別子の値と等しくなったときに結果をまとめて取得する。この処理は、カーネルが行う。

処理(a)は(2)の直後に、処理(b)は(8)の直前に追加する。また、処理(4)から処理(6)を繰り返し実行する。図4における2次 OS サーバの処理は、1次 OS サーバから2次 OS サーバへ非完了型の処理依頼を行うため、分散して実行できる。

以上から、OS サーバを改変することなく本機能を実現できる。

### 3.4 利用インタフェース

利用インタフェースの機能と形式を表1に示す。依頼元プロセスは、一括依頼を行う制御用 ICA の個数(n)とキュー構造にした制御用 ICA のリストの先頭に登録されている制御用 ICA のアドレス(p)を引数として ncallsync() を発行する。これにより、1次 OS サーバの依頼キューへ複数個の制御用 ICA が一度に登録される。依頼した処理がすべて実行終了した後、各処理の結果情報が格納された制御用 ICA のリストを依頼元プロセスへ返す。

### 3.5 分散化サーバの導入

一括処理依頼機能は、一括依頼を受け取った OS サーバ(1次 OS サーバ)が非完了型のインタフェースを駆使して、複数存在する次の OS サーバ(2次 OS サーバ)に依頼することで分散処理の効果を発揮する。

一方、OS サーバは単一の OS 機能を実現している。したがって、同じ機能の処理依頼を一括して行えるものの、異なる機能の処理依頼を一括して行えない。この問題に対処するために分散化サーバを導入する。この様子を図5に示す。分散化サーバを1次 OS サーバとして導入し、ファイル操作機能や通信制御機能といった OS 機能を実現して

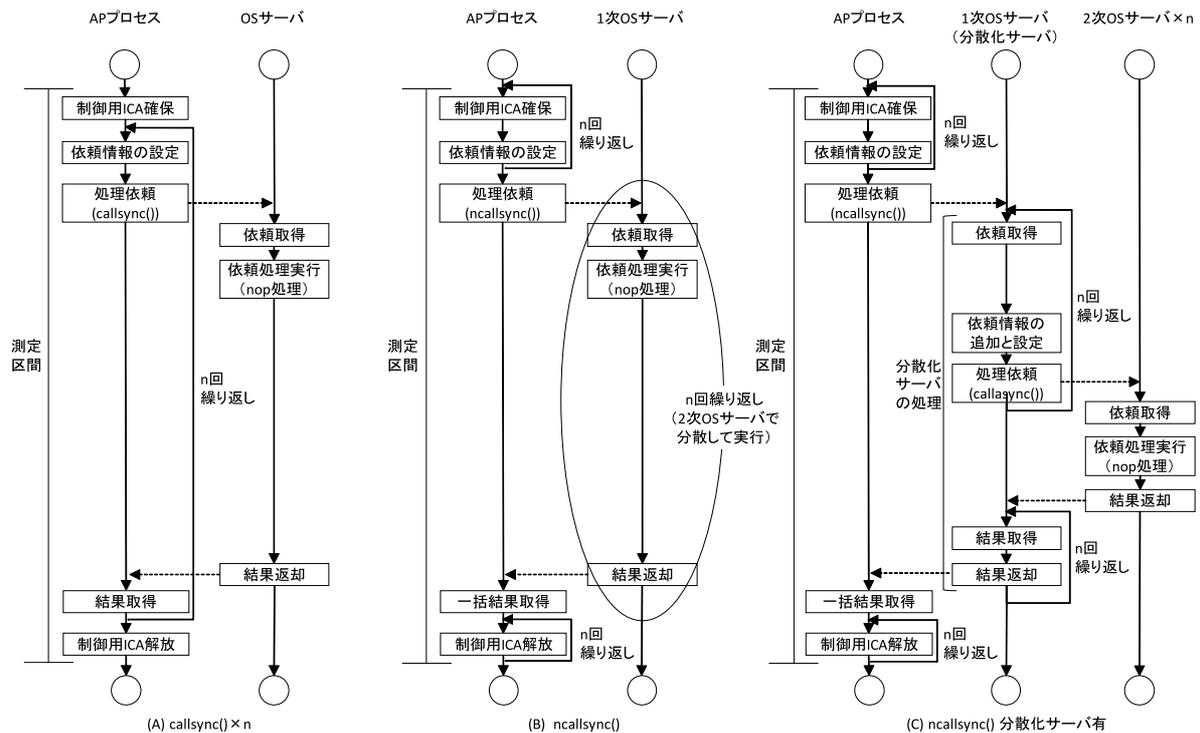


図 6 基本性能の評価の処理流れ  
 Fig. 6 Processing flow for evaluation of basic performance.

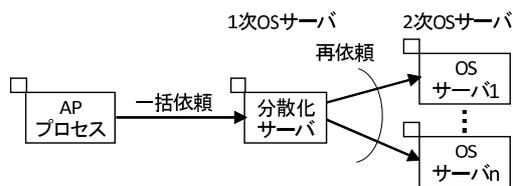


図 5 分散化サーバ  
 Fig. 5 Distribution server.

いる OS サーバを 2 次 OS サーバに位置付ける．これにより，異なる機能を実現している各 OS サーバにも処理依頼を一括して行える．

## 4. 評価

### 4.1 観点と評価環境

#### 4.1.1 観点

評価は 2 つの観点で行う．1 つは，基本評価として，処理依頼と結果返却における処理オーバーヘッドを明らかにする．具体的には，1 個の処理依頼と結果返却を繰り返す場合 (callsync() × n)，ncallsync() により n 個の処理依頼と結果返却を行う場合，および分散化サーバを利用した場合について，1 コアで実行される場合の処理オーバーヘッドを明らかにする．さらに，分散化サーバを利用した場合については，3 コアを利用して処理時間の分析結果を示す．もう 1 つは，応答時間評価として，サービスが複数の CPU 処理からなる場合，およびファイル読み込み処理とデータ送信処理からなるファイル転送処理の場合について，提案機能の効果を明らかにする．

#### 4.1.2 評価環境

評価は，送信側計算機で *AnT* を用い，受信側計算機で FreeBSD 6.3-RELEASE を用いた．いずれの計算機も，CPU に Intel Core i7-2600 (3.4 GHz, 4 コア)，メモリ 8 GB，Intel SSD 540s Series 120 GB，および Intel Pro/1000 MT NIC を搭載したものを用いた．

なお，*AnT* では，様々な独自 OS インタフェースを実現しやすいように，OS ライブラリ (libant) として OS インタフェースを実現し提供している．表 1 に示した ncallsync() も libant に組み込んで実現している．このため，評価プログラムの作成においては，コンパイル時にライブラリ libant をリンクするように指定し，ncallsync() を利用した．

### 4.2 基本評価

AP プロセスが OS サーバへ処理依頼を行い，OS サーバから AP プロセスへ結果返却が行われるまでの処理時間を評価する．この評価の処理流れを図 6 に示す．callsync() × n は，制御用 ICA を確保し，処理依頼と結果取得を n 回繰り返す，制御用 ICA を解放する．ncallsync() は，制御用 ICA を n 個確保し，処理依頼と結果取得を行い，制御用 ICA を n 個解放する．ncallsync() 分散化サーバ有は，ncallsync() の処理に加え，分散化サーバの処理が増える．OS サーバの依頼処理実行は，制御オーバーヘッドを明らかにするため，nop 処理を行う．コア数 3 の場合は，コア 0 上で AP プロセスと分散化サーバが走行し，コア 1 とコア 2 上で OS サーバが 5 個ずつ走行する．

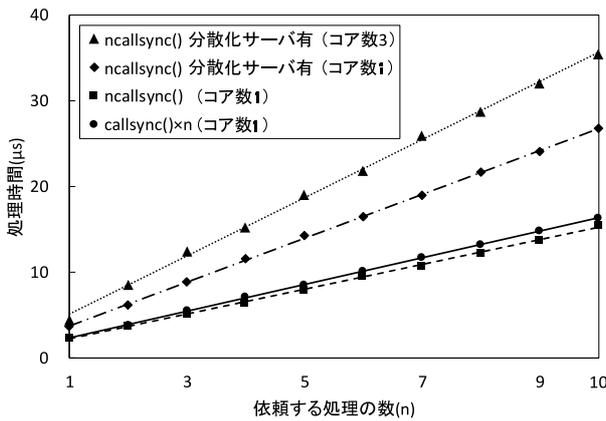


図 7 依頼する処理の数による基本性能の処理時間  
Fig. 7 Overheads for system calls.

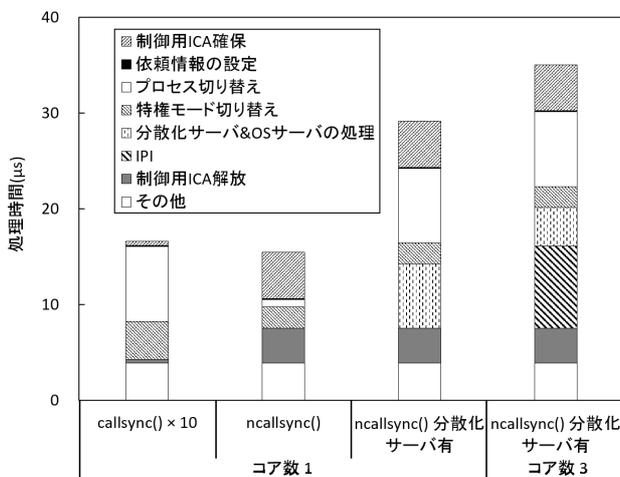


図 8 依頼する処理の数が 10 のときの各部分の処理時間  
Fig. 8 Processing time for each part (The number of requests: 10).

依頼する処理の数による処理依頼と結果取得にかかる処理時間について、測定結果を図 7 に示す。図 7 から以下のことが分かる。

(1) コア数 1 の場合、依頼する処理の数が 2 以上のとき、callsync() × n の処理時間より ncallsync() の処理時間の方が短い。つまり、AP プロセスの処理依頼のシステムコール発行回数を削減することは重要である。詳細は、図 8 で述べる。

(2) コア数 1 の場合、ncallsync() の分散化サーバ有無を比較すると、当然ながら、分散化サーバ無の処理時間が短い。分散化サーバ有による処理オーバーヘッドは、約 10 μs (n = 9 のとき) である。

また、処理時間を分析するため、依頼する処理の数が 10 個のときについて、各部分の処理時間を図 8 に示す。図 8 から以下のことが分かる。

(3) callsync() × 10 と ncallsync() を比較する。システムコール発行回数は、callsync() × 10 が 12 回、ncallsync() が 21 回であり、ncallsync() が多い。しかし、発行するシ

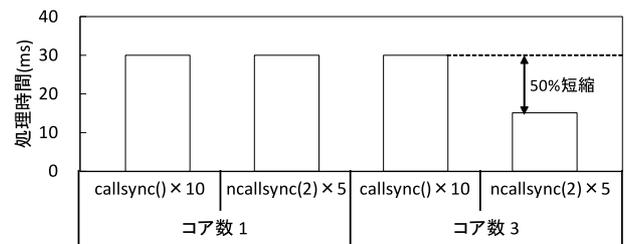


図 9 CPU 処理 (2 つの CPU 処理)  
Fig. 9 CPU processing (two processes).

ステムコールの内訳が異なっており、callsync() × 10 は、制御用 ICA 確保 1 回、処理依頼 10 回、制御用 ICA 解放 1 回である。これに対して、ncallsync() は、制御用 ICA 確保 10 回、処理依頼 1 回、制御用 ICA 解放 10 回である。1 回あたりの制御用 ICA 確保と解放の合計処理時間は、1 回あたりの処理依頼の処理時間より短いため、ncallsync() の処理時間が短い。

(4) ncallsync() 分散化サーバ有のとき、コア数 1 とコア数 3 を比較すると、コア数 1 の処理時間が短い。この差は、処理依頼や結果返却の際に別コアのプロセスを起床させるために必要な IPI (Inter-Processor Interrupt) 処理、および分散化サーバと OS サーバの関係に起因する。分散化サーバ自身の処理時間は、コア数にかかわらず同じである。しかし、コア数 1 のとき、10 個の OS サーバは、依頼された順に逐次的に処理を実行する。コア数 3 のとき、OS サーバは、2 つのコアに分かれて走行しているため、OS サーバの処理を分散実行できる。このため、分散化サーバ & OS サーバの処理時間は、コア数 3 が短い。しかし、IPI の処理時間がより長いいため、コア数 1 の処理時間が短い。

### 4.3 応答時間

#### 4.3.1 CPU 処理の分散効果

OS サーバの処理流れは、図 6 の OS サーバの依頼処理実行 (nop 処理) を 3 ms の CPU 処理に置き換え、CPU 処理の分散効果を評価した。なお、各 OS サーバへ 5 回処理依頼を行う。つまり、callsync() × n は (callsync() × 10) であり、ncallsync(2) は (ncallsync(2) × 5) である。

測定結果を図 9 に示す。図 9 から以下のことが分かる。

(1) コア数 1 のとき、callsync() × 10 と ncallsync(2) × 5 は、処理時間がほぼ同じである。ncallsync(2) × 5 は、一度に 2 つの処理を依頼するものの、OS サーバが同一コア上で走行しているため、逐次実行されるため処理時間を短縮できない。なお、先に述べたように、callsync() と ncallsync() はオーバーヘッドの差があるものの、このオーバーヘッドは CPU 処理の処理時間と比較して十分小さい。たとえば、callsync(1) × 8 では約 13 μs、ncallsync(2) × 4 では約 31 μs である。

(2) コア数 3 のとき、ncallsync(2) × 5 は、callsync() × 10 の処理時間の半分である。ncallsync(2) × 5 は、一括処理

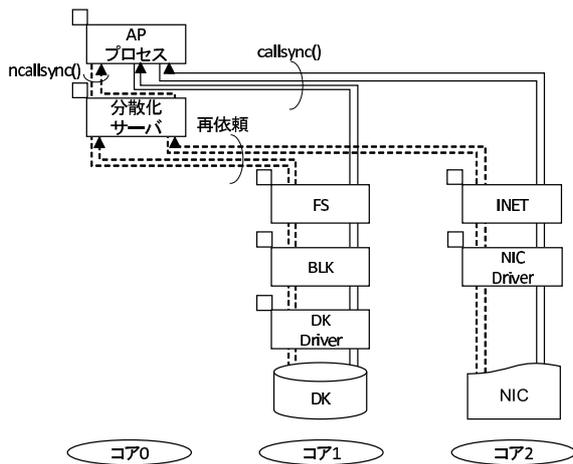


図 10 ファイル転送処理の様子 (コア数 3)

Fig. 10 Overview of file transfer (The number of cores: 3).

依頼により一度に2つの処理を依頼する。OSサーバは、それぞれ異なるコアで走行しているため、CPU処理を分散実行でき、処理時間を約50%に短縮できる。

したがって、サービスが複数のCPU処理からなる場合、一括処理依頼機能を利用することで、各CPU処理を実行する分散したOSサーバを効果的に利用でき、本機能を利用しない場合に比べ、サービス応答時間をn分の1 (nは分散したコア数)程度に短縮できる。もちろん、各CPU処理の処理時間が異なる場合は、最長CPU処理がサービス応答時間に大きく影響を与える。

#### 4.3.2 I/O処理の分散効果

I/Oをとまなう処理の分散効果を明らかにするために、ファイル転送処理を評価する。ファイル転送処理は、外部記憶装置(DK)からファイルを読み込み、読み込んだデータを別の計算機へ送信する処理である。

ファイル転送処理の様子を図10に示し、処理流れを図11に示す。ファイル管理サーバ(FS)はファイルシステムを管理し、ブロック管理サーバ(BLK)はブロックキャッシュを管理し、ディスクドライバプロセス(DK Driver)はDK入出力を制御する。ネットワークプロトコル制御サーバ(INET)はソケット生成やパケット送信処理におけるプロトコルヘッダの設定を行い、NICドライバプロセス(NIC Driver)はNIC入出力を制御する。なお、OSサーバの優先度は、同じであり、APプロセスより高い。

処理流れは、図11に示すように、callsync()の場合(従来)、ファイル読み込みとデータ送信を繰り返し行う。つまり、APプロセスは、FSとINETへ逐次的に処理依頼を繰り返し行うことでファイル転送処理を実行する(read(), send(), read(), send(), ...)。一方、ncallsync()の場合(本機能を利用)、APプロセスは、データ送信処理と次に送信するデータのファイル読み込み処理を一括依頼し、これを繰り返す。ただし、最初はファイル読み込み処理のみをFSへ処理依頼し、最後はデータ送信処理のみをINETへ処理

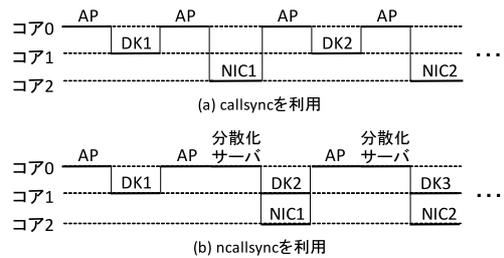


図 11 ファイル転送処理の処理流れ (コア数 3)

Fig. 11 Processing flow of file transfer (The number of cores: 3).

依頼する (read(), (send() + read()) × (n - 1), send())。

なお、コア数が3のとき、APプロセスと分散化サーバはコア0上、ファイル操作処理に関するOSサーバ群はコア1上、通信制御処理に関するOSサーバ群はコア2上で走行する。

ファイル転送処理時間を図12に示す。(A)は1KBのファイル読み込み処理と1KBのデータ送信処理を一括依頼した場合であり、(B)は2KBのファイル読み込み処理と1KBのデータ送信処理(同じ処理を2つ実行)を一括依頼した場合である。図12から以下のことが分かる。

(1) いずれの場合も、コア数1のとき、callsync() × nとncallsync()は、処理時間がほぼ同じである。ncallsync()は、APプロセスがデータ送信処理とファイル読み込み処理を分散化サーバへ一括依頼するため、分散化サーバは、処理依頼を分散実行できる。しかし、すべてのプロセスが同一コア上で走行するため、複数のCPU処理を同時に分散実行できない。さらに、いずれの場合も、1回のファイル読み込み処理時間(約0.18ms)に比べ、1回のデータ送信処理時間(約0.02ms)は非常に短いため、分散実行による効果は小さい。

(2) いずれの場合も、コア数3のとき、ncallsync()の処理時間は、約9.4msであり、callsync() × nより短い。これは、ファイル読み込み処理とデータ送信処理を分散実行できたためであると考えられる。なお短縮効果は、(B)の場合が大きい。これは、ファイル読み込み処理1回に対してデータ送信処理を2回行うと、ファイル読み込み処理時間とデータ送信処理時間の差が小さくなり、分散実行により短縮できる処理時間の割合が増加したためである。具体的には、1KBまたは2KBのファイル読み込み処理時間はどちらも約0.18msであり、1KBのデータ送信処理時間は約0.02msである。このため、(B)の場合は0.18msの処理と0.04ms(0.02 × 2)の処理を分散実行した。この結果、(A)の場合は約14%短縮できたのに対し、(B)の場合は約22%短縮できた。

上記(1)と(2)より、本機能がI/Oをとまなう処理の分散実行を可能とし、全体の処理時間を短縮できることを明らかにした。さらに、INETサーバに0.1msのCPU処

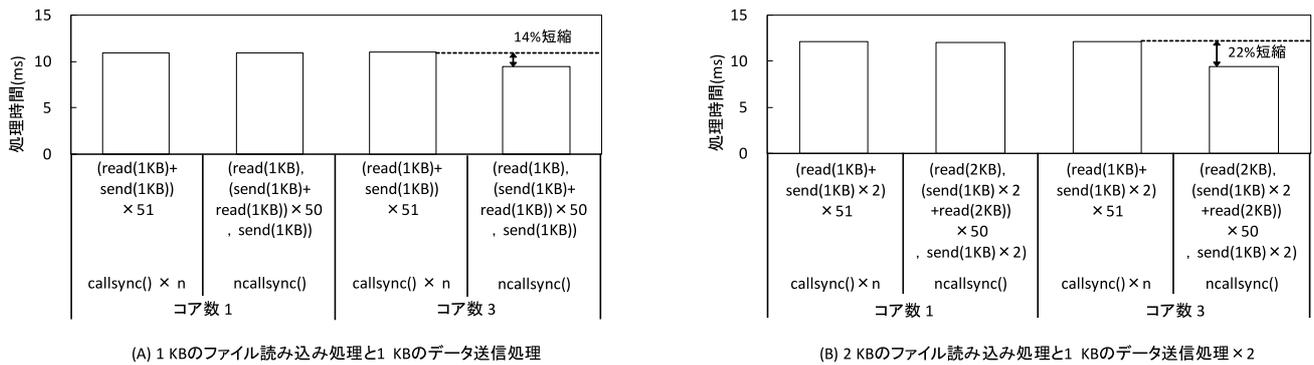


図 12 ファイル転送処理時間

Fig. 12 Processing time for file transfer.

理を追加し、ファイル読み込み処理とデータ送信処理の処理時間の差が小さくなるように変更し、評価した。この結果、コア数3のとき約40%短縮できた。したがって、分散実行する処理の処理時間の差が小さいと、短縮できる処理時間の割合が増え、本機能の効果が大きくなるといえる。

## 5. 関連研究

QNX [3] はマイクロカーネル OS であり、OS サーバ間の通信にメッセージを用いる。しかし、QNX の API には、一括して処理を依頼するインタフェースは提供されていない。L4 [5] では、OS サーバとの通信に完了型のインタフェースを用いる。OS サーバ間の通信は、共有メモリを用いて行われるため、*AnT* とは通信モデルが異なる。

OS カーネルをコアごとに配置して分散処理を実現するマルチカーネル方式 [12], [13], [14] がある。Barrelfish [12] は、マルチコア環境で OS をコアごとに分散配置して実行できる。Popcorn [13] は Linux をマルチカーネル方式に対応させたものであり、NetPopcorn [14] は Popcorn を改良し高速な通信を実現している。マルチカーネル方式では、OS 間の通信にメッセージ通信を用いることで、既存の分散処理方式を利用できる。このため、一度に複数の処理をまとめて依頼できる。*AnT* の一括処理依頼機能では、分散化サーバを用いることにより、AP プロセスから完了型インタフェースで一度に複数の処理依頼を OS サーバに行える方式を実現しており、簡明なインタフェースを維持しつつ複数の OS 処理を分散実行可能という点でマルチカーネル方式と異なる。

入出力処理を一括して処理依頼する方式として、sendmsg() や recvmsg() を用いる方法、Xen における I/O ring [15]、および virtio [16] がある。sendmsg() や recvmsg() は複数バッファの一括送受信が可能であり、sendmmsg() や recvmmsg() は複数メッセージの一括送受信が可能である。I/O ring は、Xen においてドメイン間通信の手段として用いられ、ドメインからの入出力要求も I/O ring を介して依頼される。I/O ring に複数の処理を登録してお

くことで、依頼先ドメインへ一括処理依頼が可能である。virtio は、仮想化環境においてゲストからハイパーバイザへ処理依頼を行うための Linux 向け準仮想化ドライバである。virtio では、ゲストからホストへの処理依頼を登録する virtqueue がある。ゲストは、virtqueue に複数バッファを登録することで、入出力処理を一括してハイパーバイザに依頼できる。これらの手法は、同じ機能の処理依頼を一括して行えるものの、異なる機能の処理依頼を一括して行えない。一方、提案した一括処理依頼機能は、分散化サーバの導入により、異なる機能を実現している各 OS サーバにも処理依頼を一括して行える。

モノリシックカーネル OS において、OS 機能呼び出すシステムコールの一括呼び出しのための機能が提案されている [17], [18]。これらの機能は、モノリシックカーネルにおいて、OS 機能呼び出す際に生じる走行モードの変更やコンテキストスイッチによるオーバヘッドを削減している。一方、本機能は、一括処理依頼により OS 機能を分散実行できることに着目しており、モノリシックカーネル OS における一括呼び出し機能とは目的が異なる。

uKACC [19] は、非完了型の通信を一括して行う手法をライブラリにより実現している。また、MPI における非完了型の一括入出力 [20] が実現されている。これらの手法はアプリケーションでの利用を目的としているが、一括処理依頼機能は OS 処理を分散実行するため、目的が異なる。

Linux に代表されるモノリシック OS では、AP プロセスのシステムコール発行により OS 処理 (割込み処理を除く) が実行されるため、マルチコア環境においても AP プロセスを分散実行することで OS 処理を分散実行できる。逆にいえば AP プロセスを分散できなければ、OS 処理を分散実行できない。これに対し、マイクロカーネル OS は、大半の OS 機能を OS サーバとして実現しているため、マルチコア環境において、AP プロセスが分散実行しているか否かに関係なく OS 処理の分散実行が可能である。このとき、分散した各 OS サーバに効率的に処理を依頼できることが強く求められる。この要求を満足するため、*AnT* で

は、提案した一括処理依頼機能により、効率的な処理の依頼を可能にしている。さらに、最も特徴的な事項は、分散化サーバの導入により、異なる機能を実現している各 OS サーバにも処理依頼を一括して行えることである。

## 6. おわりに

一度に複数の処理依頼を行える一括処理依頼機能を提案し、マイクロカーネル構造を有する **AnT** での実装を述べた。また、評価により、OS 処理の分散実行を効果的に利用できることを示した。本機能の実現において、従来のサーバプログラム間通信機構の変更を局所化することで、既存 OS サーバを変更することなく実現した。また、分散処理を促進するために、分散化サーバを導入した。

基本評価では、処理依頼と結果返却における制御オーバーヘッドを明らかにした。一括処理依頼により処理依頼の制御オーバーヘッドを削減できることを示した。また、分散実行した OS サーバの処理時間が短い場合、IPI の処理時間に起因して本機能の制御オーバーヘッドが大きいことを示した。応答時間の評価では、CPU 処理と I/O 処理において、分散実行によりサービス応答時間を短縮できることを示した。サービスが複数の CPU 処理からなる場合は、本機能を利用しない場合に比べ、サービス応答時間を  $n$  分の 1 ( $n$  は分散したコア数) 程度に短縮できることを示した。I/O をともなう処理の場合は、分散実行する処理の処理時間の差が小さいと、短縮できる処理時間の割合が増え、本機能の効果が大きくなることを示した。具体的には、ファイル読み込み処理とデータ送信処理からなるファイル転送処理を分散実行することで、サービス応答時間を約 22% 短縮できることを示した。また、分散実行する処理時間の差を小さくするために、データ送信処理を行う INET サーバに CPU 処理 (0.1 ms) を追加した場合、サービス応答時間を約 40% 短縮できることを示した。

## 参考文献

- [1] Bricker, A., Gien, M., Guillemont, M., Lipkis, J., Orr, D. and Rozier, M.: Architectural issues in microkernel-based operating systems: The CHORUS experience, *Computer Communications*, Vol.14, No.6, pp.347-357 (1991).
- [2] 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, *Proc. USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pp.11-30 (1992).
- [3] Hildebrand, D.: An Architectural Overview of QNX., *USENIX Workshop on Microkernels and Other Kernel Architectures*, pp.113-126 (1992).
- [4] Unrau, R.C., Krieger, O., Gamsa, B. and Stumm, M.: Hierarchical clustering: A structure for scalable multiprocessor operating system design, *The Journal of Supercomputing*, Vol.9, No.1-2, pp.105-134 (1995).
- [5] Liedtke, J.: Toward real microkernels, *Comm. ACM*, Vol.39, No.9, pp.70-77 (1996).
- [6] Tanenbaum, A.S., Herder, J.N. and Bos, H.: Can we make operating systems reliable and secure?, *Computer*, Vol.39, No.5, pp.44-51 (2006).
- [7] Kaiser, R. and Wagner, S.: Evolution of the PikeOS microkernel, *1st International Workshop on Microkernels for Embedded Systems*, pp.50-57 (2007).
- [8] Wentzlaff, D. and Agarwal, A.: Factored operating systems (fos): The case for a scalable operating system for multicores, *ACM SIGOPS Operating Systems Review*, Vol.43, No.2, pp.76-85 (2009).
- [9] 佐古田健志, 山内利宏, 谷口秀夫: 高スループットを実現する OS 処理分散法の実現, マルチメディア, 分散, 協調とモバイル (DICOMO2013) シンポジウム論文集, Vol.2013, No.2, pp.1663-1670 (2013).
- [10] 江原寛人, 河上裕太, 山内利宏, 谷口秀夫: ファイル操作に着目した OS 処理分散法, 情報処理学会研究報告, Vol.2015-OS-132, No.7, pp.1-7 (2015).
- [11] 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).
- [12] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A. and Singhanian, A.: The multikernel: A new OS architecture for scalable multicore systems, *Proc. ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pp.29-44, ACM (2009).
- [13] Barbalace, A., Ravindran, B. and Katz, D.: Popcorn: A replicated-kernel OS based on Linux, *Proc. Linux Symposium*, Ottawa, Canada (2014).
- [14] Ansary, S., Barbalace, A., Chuang, H.-R., Lazor, T. and Ravindran, B.: A Distributed Operating System Network Stack and Device Driver for Multicores, *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp.2646-2649, IEEE (2017).
- [15] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the Art of Virtualization, *ACM SIGOPS Oper. Syst. Rev.*, Vol.37, No.5, pp.164-177 (2003).
- [16] Russell, R.: virtio: Towards a De-Facto Standard for Virtual I/O Devices, *ACM SIGOPS Oper. Syst. Rev.*, Vol.42, No.5, pp.95-103 (2008).
- [17] 谷口秀夫: 新しい OS カーネル内外連携機構: DRD 機構の提案, 電子情報通信学会論文誌 (D), Vol.J85-D-I, No.2, pp.193-201 (2002).
- [18] 谷口秀夫, 日下部茂, 中山大士, 乃村能成, 雨宮真人: 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, 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 (HPS), Vol.44, No.SIG01 (HPS6), pp.81-92 (2003).
- [19] Nomura, A., Ishikawa, Y., Maruyama, N. and Matsuoka, S.: Design and implementation of portable and efficient non-blocking collective communication, *Proc. 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp.1-8 (2012).
- [20] Seo, S., Latham, R., Zhang, J. and Balaji, P.: Implementation and Evaluation of MPI Nonblocking Collective I/O, *Proc. 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp.1084-1091, IEEE (2015).



佐藤 将也 (正会員)

2010年岡山大学工学情報工学科卒業。2012年同大学大学院自然科学研究科博士前期課程修了。2014年同大学院同研究科博士後期課程修了。2013年日本学術振興会特別研究員(DC2)。現在、岡山大学大学院自然科学研究科助教。博士(工学)。コンピュータセキュリティ、仮想化技術に興味を持つ。2012年度情報処理学会論文賞受賞。電子情報通信学会会員。



谷口 秀夫 (正会員)

1978年九州大学工学部電子工学科卒業。1980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。1987年同所主任研究員。1988年NTTデータ通信株式会社開発本部移籍。1992年同本部主幹技師。1993年九州大学工学部助教授。2003年岡山大学工学部教授。2010年岡山大学工学部長。2014年岡山大学理事・副学長。博士(工学)。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「並列分散処理」(コロナ社)等。電子情報通信学会、ACM各会員。本会フェロー。



村岡 勇希 (正会員)

2016年岡山大学工学部情報系学科卒業。2018年同大学大学院自然科学研究科博士前期課程修了。オペレーティングシステムに興味を持つ。



山内 利宏 (正会員)

1998年九州大学工学部情報工学科卒業。2000年同大学大学院システム情報科学研究科修士課程修了。2002年同大学院システム情報科学府博士後期課程修了。2001年日本学術振興会特別研究員(DC2)。2002年九州大学大学院システム情報科学研究院助手。2005年岡山大学大学院自然科学研究科助教。現在、同准教授。博士(工学)。オペレーティングシステム、コンピュータセキュリティに興味を持つ。2010年度JIP Outstanding Paper Award, 2012年度情報処理学会論文賞受賞。電子情報通信学会、ACM, USENIX, IEEE各会員。本会シニア会員。