汎用OSと専用OSを高効率に相互補完するナノカーネルの提案と実現

新井利明[†] 関口知己[†] 佐藤雅英[†]
木村信二[†] 大島訓[†] 吉澤康文[††]

オペレーティングシステム（OS）はこれまでに多くのものが開発されているが，ユーザの要求が多様であり，すべての要求を満足するOS開発は不可能に近い．そこで，1台のマシン上に汎用OSと特定の目的を持つ専用OSを共存させ各々機能補完する仮想計算機機能のナノカーネルを提案し，実現した．豊富なソフトウェア資産を活用できる汎用OSと特殊機能を有する専用OSを1台のマシン上に共存させ，互いに機能補完させることができる．ナノカーネルは，上記の目的を達成するために，(1) 複数OS共存オーバヘッドを削減するための資源分割機能，(2) OS間の機能補完を可能とするOS間連携機能，(3) OSの信頼性を向上させる障害監視，回復機能と擬似不揮発メモリ機能などで構成する．これらの限定した機能を実現することで，ナノカーネルは複数OSの共存を可能とし，補完環境をオーバヘッド2%以内で達成できることを確認した．また，汎用OSとリアルタイムOSの共存環境を構築し，汎用OS環境では不可能であったマイクロ秒単位の応答性を確保できることを確認し，ナノカーネルの持つOS間機能補完を実証した．さらに，専用の高信頼OSからの汎用OS障害情報の収集や汎用OSの再起動処理を実現し，システムの信頼性向上にも有効であることを確認した．

A Nanokernel Technology for a Multiple-OS Coexisting and Complementing Environment

Toshiaki Arai,[†] Tomiki Sekiguchi,[†] Masahide Sato,[†]
Shinji Kimura,[†] Satoshi Ooshima[†] and Yasufumi Yoshizawa[††]

Although various kinds of Operating Systems (OSs) have been developed so far, a user hardly finds the OS satisfying all the users' needs completely. We proposed and developed a kind of virtual machine called Nanokernel, which effectively enables a general purpose OS and a special purpose OS co-exist in one machine and complement each other. By complementing general purpose OS with rich software and special purpose OS with special function, Nanokernel realizes the environment which meets variety of users' needs effectively. We restrict Nanokernel functions to resource partitioning for lower overhead, communication between OSs for function complement between them, and failure detection and fast recovery for reliability, to achieve the purpose effectively. By restricting Nanokernel functions above, Nanokernel achieves the multi-OS co-existing environment less than 2% overhead. The Nanokernel environment with a general purpose OS and a realtime OS showed that Nanokernel relieves the lack of realtime property of the general purpose OS and establishes micro-second order response time. Moreover, even when the general purpose OS crashes, the realtime OS can survive, get the information for the failure from the general purpose OS and execute the general purpose OS restart process, so that Nanokernel is useful for enhancing system reliability.

1. はじめに

オペレーティングシステム（OS）は，これまでに多種多様なものが開発されてきているが，ユーザの要求も多岐にわたるため，すべてのユーザを満足するOSを開発することは不可能に近い．広く普及している，いわゆる汎用OSは，豊富なソフトウェアが利用可能であり，周辺装置やそれを制御するドライバなども整備されているため，これを用いることで比較的容易にシステムを構築することができる．しかし，リアルタイム性の保障や高信頼化への要求など，専用OSが提供している特殊機能が必要な場合には専用OSが必要

† 株式会社日立製作所システム開発研究所
Systems Development Laboratory, Hitachi, Ltd.
†† 東京農工大学大学院共生科学技術研究部
Tokyo University of Agriculture & Technology

となる．一方，専用 OS は，ある特殊な機能に特化しているため，特定の目的を満たすことはできる．しかし，利用目的が特定の領域に限定されていることが多いため，汎用 OS では利用できる豊富なソフトウェアの機能を使えないという欠点がある．つまり，利用可能なアプリケーションや整備された開発環境が必要な場合には汎用 OS が必要となる．

このように，汎用 OS と専用 OS は各々特徴があり，相互に補完関係にある．そこで，本論文では，2 種類の OS を相互に補完することを目的とした仮想計算機機能であるナノカーネルを提案する．過去に各種の仮想計算機実現方式が提案されているが，ここでは，上記の目的に限定した機能をナノカーネルとして提案し，開発した．その特長は，

(1) 複数 OS 共存オーバヘッド削減のための資源分割機能，
(2) OS 間の機能補完を可能とする OS 間連携機能，
(3) OS の信頼性を向上させる障害監視，回復機能と擬似不揮発メモリ機能，

である．これらの機能に限定することにより，ナノカーネルは 16 kstep 程度の開発量で複数 OS の動作基盤を実現している．同様に OS の動作基盤を提供する技術としてマイクロカーネル[1] があるが，ナノカーネルはより少ないステップ数で実現できており，ナノカーネルという名称の由来となっている．

このナノカーネルを開発し，効果を確認したのでここに報告する．

2. ナノカーネルの目的と概要

ナノカーネルの目的は，異なる機能を持つ OS を相互補完させて，単一の OS では実現不可能な機能を持つシステムを効率良く開発することである．具体的には，多くのアプリケーションと豊富な開発環境を持つ汎用 OS と特殊機能を持つ専用 OS の両者の特長を兼ね備えたシステムや，旧 OS 上の既存アプリケーションと新 OS 上の新アプリケーションを同時実行できるシステム，など，単一 OS で実現するためには多大な工数を必要とするシステムを効率良く開発できるようにすることである．

これらを可能とするために，複数 OS の同時実行機能，OS 間の連携機能，OS 障害の監視，回復機能，擬似不揮発メモリ，に限定して少ないオーバヘッドで実現する機能がナノカーネルである．1 台のマシン上に複数の OS を搭載する方式としては，メインフレームの VM（Virtual Machine）およびマイクロカーネル技術が知られている[2)~5)] が，いずれも複数 OS を独立して実行させるものであり，ナノカーネルのように OS 間の機能補完を目的とするものではない．実現方式に関しても，VM はハードウェアアーキテクチャをソフトウェアおよびファームウェアを用いてエミュレーションする技術であり，十分な性能を発揮するためには専用のハードウェアが必要である．また，マイクロカーネル[1] は，最小限の機能のみをカーネルモードで実行し，その他の機能をシステムサーバとしてユーザモードで実行する技術であるが，システムサーバの開発工数および性能劣化が問題となる．

これらのことから，ナノカーネルアプローチによる複数 OS 搭載機能では，性能の維持と既存 OS の変更を最小限度に抑えることを第 1 の目的としている．この目的を実現するため，ナノカーネルは，基本的に，各 OS に割り当てる計算機資源をエミュレーションによって仮想化するのではなく，物理的な計算機資源を各 OS に分割して振り分けることでオーバヘッドを少なくする．

ナノカーネル技術の次の特徴は，OS 間の連携機能を構築するためのプリミティブを提供する点にある．先に述べたように，ナノカーネルの目的は，複数 OS 間の連携によって，目的とする機能を，短期開発によって提供するところにある．これを可能とするため，異なる OS 上のプロセスあるいはタスク間の通信/同期などの機能をプリミティブとして用意することにした．また，これらの機能の応用として，OS 障害の検知や障害情報の収集，OS の再起動，さらには，OS 再起動時の処理引継ぎを高速化する擬似不揮発メモリ，などの機能を提供している．

以上のように，従来の仮想計算機技術のように単に複数 OS を同時実行させるだけでなく，OS 間の機能補完を実現する OS 間連携機能と OS を高信頼化する障害監視，回復機能および擬似不揮発メモリ機能を実現している点がナノカーネルの特長である．

ナノカーネルの概念構造を**図 1** に示す．ナノカーネルは，ハードウェアと OS の中間に位置するレイヤで

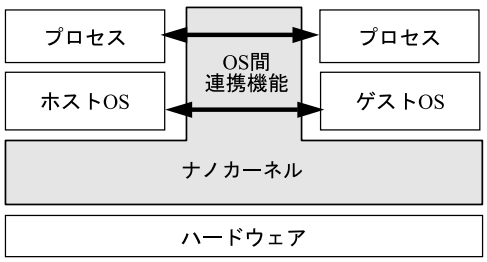

図 1 ナノカーネルの基本概念
Fig. 1 Nanokernel overview.

あり，1 台のハードウェア上への複数 OS の搭載機能と OS 間の連携機能を実現する．ナノカーネル上に搭載する複数の OS は概念的には対等であるが，マシン起動時に最初に立ち上がる OS をホスト OS，その後立ち上がる OS をゲスト OS と呼んでいる．これに関しては，実装の章で詳説する．

ナノカーネル機能を用いてホスト OS とゲスト OS を共存させることにより，単一 OS では実現が難しい様々な機能を提供できる．

(1) リアルタイム性強化

汎用 OS とリアルタイム専用 OS を共存させ，ナノカーネルの OS 間連携機能を用いることにより，リアルタイム性を要求する部分のみをリアルタイム専用 OS が受け持ち，その他の処理は汎用 OS の豊富なアプリケーションを利用する，というような機能分担が可能となる．この場合，ナノカーネルはリアルタイム専用 OS に高プライオリティを付加し，リアルタイム OS はそのリアルタイム性を保障する．このように，汎用 OS を適用したシステム環境に特定機能を簡単に追加することが可能である．

(2) 信頼性

独自の高信頼な OS が汎用 OS の監視を行い，汎用 OS に障害が発生した際に障害情報を収集し，オペレータへ通知することにより，障害原因の追求，回避策の検討，復旧手段の確保を行うことが可能となる．つまり，汎用 OS と独自開発の高信頼な監視 OS を共存させることにより，汎用 OS の信頼性を向上させることが可能となる．

また，障害が発生した OS の再起動後に，システムの状態を高速に回復させるための擬似不揮発を用意している．通常，OS の再起動処理は計算機ハードウェアのリセット処理をともなうため，メモリ内のデータはすべて消えてしまい，再起動後に使用することはできない．そこで，ナノカーネルでは，再起動後もクリアされない領域として擬似不揮発メモリを用意している．更新ログなどのシステムの状態に関する重要なデータをこの擬似不揮発メモリに格納しておくことにより，障害解析や障害時の引継ぎ処理の高速化が可能となる．

(3) 既存ソフトウェアとの互換性

多くのシステムでは，従来から運用を続けてきた OS（レガシ OS）上に膨大なアプリケーション財産が存在する．それらのプログラムはレガシ OS に依存する部分が多いため，新しい汎用 OS に移植するためには多くの工数が必要であった．しかし，ナノカーネルでは汎用 OS とレガシ OS を共存させることが可能であるため，蓄積されたソフト資産を有効に活用することができる．たとえば，従来機能をレガシ OS 上で実行しつつ，最新の汎用 OS 上に新規機能を構築することで，従来機能を活用しつつ最新 OS を利用することが可能となる．また，全機能をレガシ OS 上に搭載し，システムを運用しつつ徐々に機能を最新汎用 OS 上に移植する，といった段階的機能移植も可能である．

従来の仮想計算機技術と異なり，ナノカーネルでは異なる OS 上のプロセス間での連携機能を用意しているため，システムの拡張を効率良く短期間で実現することが可能である．たとえば，新しい周辺装置が開発され，その装置は最新の汎用 OS のみがサポートしている，というような状況においても，レガシ OS の I/O 処理部分のみを最新汎用 OS 上に再構築し，OS 間プロセス連携機能を利用してデータを送受信することでレガシ OS から新しい周辺装置を利用することが可能となる．

以上のように，ナノカーネル技術を利用することにより，拡張性に富んだ，高信頼，高機能なシステムを構築することが可能である．

3. ナノカーネル機能

本章では，ナノカーネルの実現方式を説明する[6)~9)]．ナノカーネルの目的は，複数 OS 実行環境と OS 間連携機能を簡便にかつオーバヘッド少なく実現することである．

1 台のマシン上で複数の仮想計算機（VM: Virtual Machine）を構築し，その各々で OS を同時実行させる仮想計算機モニタ（VMM: Virtual Machine Monitor）機能を実現するためには，プロセッサ，メモリ，I/O 装置などの計算機資源を，仮想的に各 VM に対して割り付ける必要がある．これを実現するため，VMM は実際にマシンに装備されている資源を仮想化し，各仮想計算機に割り付ける．そのためには，仮想化された資源の状態を管理し，実資源と仮想化された資源との関連を制御する必要がある．たとえば，VM 上の OS が仮想化された計算機資源に対する操作命令を発行した際に，その命令をトラップして，あたかも仮想資源に対する操作が完了したかのように仮想資源の状態を管理するテーブルの内容を変更し，実資源との対応関係を示すテーブルを変更する．このようにして資源を仮想化する，いわゆるエミュレーションによる仮想化方式は，一般的な方式であり，これまでにいくつかのシステムで実現されている[2)~5)]．

仮想化すべき計算機資源の主なものにはプロセッサ，メモリ，I/O 装置があるが，これらの中では特にメモ

表 1 ナノカーネル機能一覧
Table 1 Nanokernel functions.

#	項目		機能概要
1	資源分割機能	メモリ分割	物理メモリを分割し、ホストOS、ゲストOS、ナノカーネルおよび共有メモリに割付け
2		I/O装置分割	各々のI/O装置をホストOS、ゲストOSに占有
3		プロセッサ共用	プロセッサを時分割してOS間で共用
4		タイマ共用	タイマをOS間で共用
5	OS間連携機能	共有メモリ	OS間で共有するメモリ 擬似不揮発メモリとしても利用
6		ﾒｯｾｰｼﾞﾊﾟｼﾝｸﾞ	異種OS上のプロセス間のメッセージパシング
7		プロセス間同期	異種OS上のプロセス間の同期制御
8	障害監視、回復機能		障害発生を他OSに通知 障害OS以外のOSの継続実行を保証 障害OSのみの再起動

リと I/O はエミュレーションオーバヘッドが大きい．それは，それらの資源は状態変更の頻繁度が高く，かつ，エミュレーションのオーバヘッドが大きいためである．たとえば，I/O 装置を仮想化するためには，装置への I/O 発行や装置からの割込み発生のたびごとに，仮想装置の状態を示す管理テーブルの操作やその装置が使用するバッファ領域などの管理が必要である．そのため，エミュレーション方式では特に I/O が頻繁に発行される環境ではオーバヘッドが大きい．

そこで，ナノカーネルでは，計算機資源をエミュレーションによって仮想化して配分するのではなく，資源を分割して各 VM に提供する．これにより，資源の状態を変更する命令を直接実行させることが可能になり，エミュレーションオーバヘッドを削減できる．さらに，OS 間の機能連携を用意に実現するため，命令レベルでの相互アクセスが可能な共有メモリ方式も採用した．**表 1** にナノカーネル機能の一覧を示す．

(1) 資源分割機能

資源を分割して各 OS に占有して割り付けることにより，仮想化のオーバヘッドを削減する．

(a) メモリ分割機能

物理メモリを静的に分割し，各 OS，ナノカーネルおよび共有メモリに割り付ける．各 OS は割り付けられた仮想的な物理メモリをそのまま使用するため，物理メモリと仮想的な物理メモリ間のマッピングを管理する必要はない．ただし，この方式では，ゲスト OS 用のメモリはアドレス 0 から始まらないため，ゲスト OS の修正が必要となる．これに関しては，4 章で詳説する．この方式では，各 OS は与えられたメモリ以外はハード的に搭載されていないものとして動作するため，OS 障害時に誤って他 OS に割り付けられたメモリをアクセスする危険性はない．

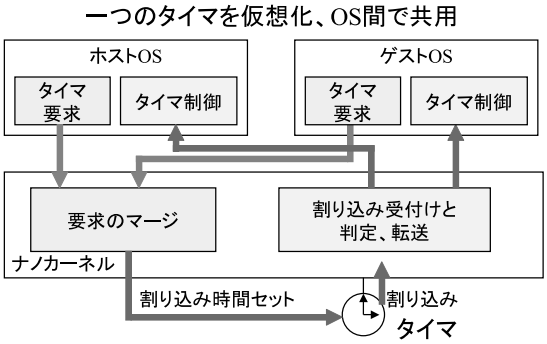

図 2 タイマ共用機能
Fig. 2 Virtual timer function.

(b) I/O 分割機能

OS 単位に各々が制御する I/O 装置を占有させる．I/O 装置は必ず 1 つの OS からのみ操作されるため，ナノカーネルが介在して管理する必要はない．OS は自分自身に割り付けられた I/O 装置以外はハード的に搭載されていないものとして実行するため，OS 障害時にも誤って他 OS に割り付けられた I/O 装置をアクセスする危険性はない．

(c) プロセッサ共用機能

プロセッサは多くのシステムでは 1 つであるため，各 OS に時分割で割り付ける．割付けのアルゴリズムおよび優先順位は共存する OS の特性に依存する．たとえば，汎用 OS とリアルタイム OS を共存させる場合には，リアルタイム OS を優先させ，リアルタイム OS に対する割込み事象が発生した場合には，汎用 OS の処理を中断してリアルタイム OS に制御を移す．

(d) タイマ共用機能

タイマも通常ハードウェアに 1 つのみ実装される装置であり，OS 間で共用する必要がある．タイマは指定された時間に割込みを発生させることが基本動作である．ナノカーネルは，各 OS からのタイマ割込み発生要求を受け取り，各要求をマージして，最も近い将来発生する割込みの時間を物理タイマにセットする．**図 2** に示すように，タイマ割込みが発生した際には，その割込みがどの OS からの要求に対応するものであるかを判定し，該当する OS のタイマ割込みハンドラルーチンへ制御を渡す．

(2) OS 間連携機能

ナノカーネルの特長は，単に複数の OS を搭載することだけではなく，OS 間の機能連携を可能とし，相互補完によるシステム全体の機能強化，信頼性向上を可能とする点にある．OS 間連携機能はこの実現のために不可欠な機能である．ナノカーネルではこのために，**図 3** に示すように，目的に応じて 3 種類の方法

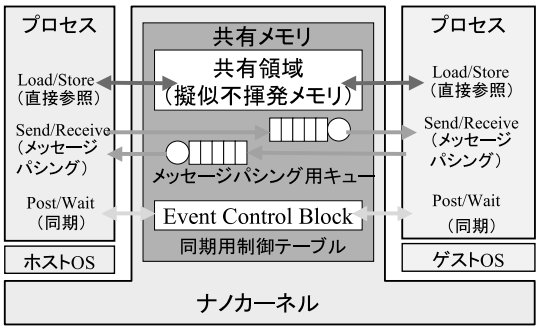

図 3 OS 間連携機能
Fig. 3 Intercommunication among OSs.

を利用できるようにした．

(a) 共有メモリ機能

OS 間で共通に参照可能となるメモリ領域を設定する機能である．先に述べたメモリ分割機能で確保した専用の物理メモリを共有メモリ用に使用し，この領域を各 OS からアクセス可能とする．ただし，記憶保護強化のため，物理メモリを直接参照するのではなく，論理化した仮想メモリとしてアクセスする方式とする．本領域は OS 間通信の重要な部分であるため，仮想記憶の参照にページフォールトが発生しないように実メモリの割付けを行う．

(b) OS 間メッセージパシング機能

相異なる OS 上のプロセス間でメッセージパシングによりデータの共用，転送，プロセスの同期制御などを実現する機能である．具体的には上記の共有メモリを用いて実装し，共有メモリ上にメッセージキューと送受信に関する制御テーブルなどを配置する．制御テーブルの排他制御用のインターロックも制御テーブル上に構築する．

(c) OS 間プロセス同期機能

異なる OS 上のプロセス間で同期をとる機能を提供する．上記の共有メモリ上に同期のための制御テーブルを配置することで実現する．ここでは，同期処理のためのイベントの発生状態を表すイベント変数と，そのイベントに関連するプロセスの識別子などの情報を，両方の OS からアクセス可能な共有メモリに配置することにより，異なる OS 上のプロセス間で各々共通のイベント変数名を指定するのみで同期をとることを可能としている．

(d) 擬似不揮発メモリ機能

OS の回復処理などを高速化する擬似不揮発メモリは，上記の共有メモリを用いて実現する．共有メモリの物理領域は，ホスト OS，ゲスト OS のいずれにも属さないため，OS が障害などでいったん終了し，再度起動した後でもその内容は保持されている．この特長を利用して，共用メモリを擬似不揮発メモリとして提供する．

擬似不揮発メモリを使用するプロセスは，プロセス起動時に，そのプロセス固有の識別子を指定して擬似不揮発メモリの割付けを要求する．これは共有メモリ割付け要求のオプションの 1 つとして実現する．ナノカーネルはプロセスからの擬似不揮発メモリ要求の履歴を共有メモリ上に記録しており，もし，そのプロセスが前回起動された際も同一の識別子を指定して擬似不揮発メモリを要求したと判明した場合には，前回割り付けた共有メモリのアドレスと同一のアドレスを割付け要求の結果として返す．また，そのプロセスからの最初の要求である場合には共有メモリの空領域を新たに割り付ける．これにより，ナノカーネルが継続して動作している間は，OS が異常終了し再立ち上げした場合にも，その OS 上のプロセスは共有メモリをあたかも不揮発メモリが装備されているかのように使用することができる．

(3) OS の独立性保障

ナノカーネル上で稼動する各 OS は，それぞれ独立に実行されている．すなわち，ある OS の状態が他の OS に影響を与えることはない．これは，OS に障害が発生し，異常終了した場合も同様である．これを実現するため，ナノカーネルは，メモリおよび I/O 装置を各 OS に占有させることで，OS 障害時に他 OS 用の資源を不正アクセスすることを防止している．プロセッサについては，ナノカーネルが障害発生をトラップすることで OS の障害を認知し，その OS へのプロセッサの割当てを抑止することで，障害が他の OS へ影響しないよう制御している．

さらに，ナノカーネルでは，他 OS の実行は継続したまま，異常終了した OS のみを，再起動することも可能としている．以上すべての機能は，各 OS を独立して制御するというナノカーネルの基本思想に基づいている．

(4) 障害監視，回復機能

OS の障害を監視し，障害情報の収集や障害が発生した OS の再起動処理を行う．ナノカーネルは，障害に関連する特定の割込みやシステムコールの発生をトラップすることで OS 障害の発生を監視している．通常，OS は OS 障害によりシステムの続行が不可能と判断した際には，特別な割込みやシステムコールを発行して停止する．ナノカーネルはこれをトラップすることで OS 障害を検知する．

ナノカーネルは，OS 障害を検知した場合には，他

方の OS にその障害発生を通知する．たとえば，ホスト OS で障害が発生した場合には，障害情報の収集やオペレータへの通知はゲスト OS が実施する．これは，ナノカーネルが通信機能を持たないことと，ゲスト OS の機能を有効に使用するためである．

ナノカーネルは障害 OS の再起動処理も実施する．ナノカーネル自身は I/O 装置を持たないため，OS をディスクから読み込むことはできない．そこで，ホスト OS のローディング時に，そのイメージを共有メモリ領域にコピーしておき，それをホスト OS 領域にコピーすることで OS の再ローディングを可能とする．

以上の方式により，ナノカーネルは 1 つのマシン上で，複数の OS が独立実行することを可能としている．

4. ナノカーネルの実装

4.1 ナノカーネルの起動とホスト OS，ゲスト OS の起動

ナノカーネルは，ホスト OS のドライバとして起動する．ドライバとして起動することにより，特権命令の実行やカーネル専用のシステムコールの使用が可能となり，計算機のすべての資源を制御することができるようになる．

たとえば，Linux をホスト OS とするナノカーネル（Linux ナノカーネル）では，Linux のカーネルに動的にプログラムをローディングする仕掛けであるカーネルモジュールを利用してナノカーネルを起動する[10),11)]．これにより，ホスト OS である Linux カーネル自身を修正する必要がなくなり，Linux カーネルから見てナノカーネルはある種のデバイスドライバと見なすことができるようになる．

ナノカーネルの起動は以下の手順で行う．

(1) ホスト OS の起動

ホスト OS の起動に際しては，ホスト OS が使用する物理メモリ領域を起動時パラメータによって指定する．これにより，ナノカーネルやゲスト OS および共有メモリが使用する物理メモリ領域が，ホスト OS によって使用されることのないようにする．

(2) ナノカーネルの起動

ナノカーネルはホスト OS が使用する物理メモリ領域の外にロードする必要があるため，ホスト OS のローディング時にその一部としてロードすることはできない．そのため，まず，ナノカーネルをロードするためのローダであるナノカーネル・ローダをホスト OS のドライバとしてロードする．そして，ホスト OS のカーネル初期設定後にドライバ初期設定処理の中で制御を受け，外部記憶に格納されたナノカーネル本体をローディングする．この際，ローディングする物理メモリ領域は，ホスト OS が認識する領域の範囲外とする．これは，ホスト OS とナノカーネルの独立性を保つために必要であり，これにより，ホスト OS のみの再起動処理などが実現できる．

Linux ナノカーネルでは，ナノカーネル・ローダは，ホスト OS である Linux に用意されているドライバのローディング機能を利用する．この際，ローディング先の仮想領域に対応するアドレス変換テーブル・エントリを書き換えて，該当仮想領域をナノカーネル用の物理領域にマッピングする．これにより，通常のドライバ用ローディング機構を利用しつつ，ホスト OS の領域外の物理領域にナノカーネル本体をローディングすることができる．

(3) ゲスト OS の起動

ゲスト OS もナノカーネルと同様の手順でローディングすることが可能である．

ここでの課題は，通常利用されている OS をゲスト OS として起動する際のアドレスに関する改造である．通常，ホスト OS が物理メモリの 0 番地を使用しているため，物理メモリの 0 番地からローディングされる OS をゲスト OS として使用する場合には，0 番地以外からローディングされるよう，改造が必要である．

たとえば，Linux をゲスト OS として動作させる場合には，仮想アドレスと物理アドレスの変換を実施するマクロを修正して，変換の際に一定のオフセット値を加算するように変更することにより，物理メモリの 0 番地を使用しないように改造することができる．Linux 以外の OS の場合にも，同様に，物理アドレスの 0 番地を使用しないよう修正することで，ゲスト OS として実装することができる．

以上に加えて，ゲスト OS として動作させるためには，割込み処理の修正が必要である．ナノカーネル方式ではすべての割込みはいったんナノカーネルが受け付け，必要に応じてホスト OS，ゲスト OS に振り分ける．そのため，ゲスト OS に割込み処理が転送された段階では，割込みに関する状態を示すフラグ類が変更されている可能性があり，これを修復するための処理が必要である．さらに，OS 間連携機能のインタフェースの調整も必要である．以上の修正に必要なコーティング量は Linux をゲスト OS とする場合には 0.5 kstep であった．

4.2 I/O リソースの分離

ナノカーネルでは，I/O 装置はホスト OS またはゲスト OS が占有する．これを実現するため，ナノカーネル・ローダは，ホスト OS のドライバ初期設定時に，

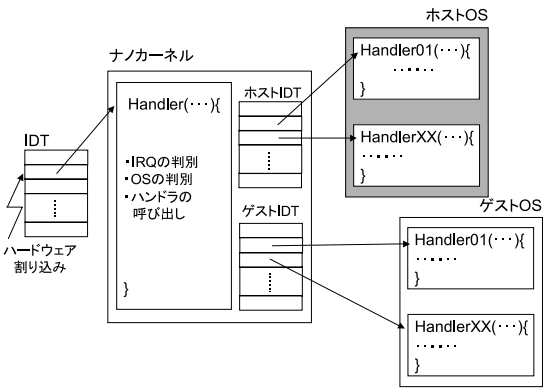

図 4 割込み振り分け処理
Fig. 4 Interruption management process.

ナノカーネルおよびゲスト OS に割り当てる I/O 装置のリソースを確保し，それらの I/O 装置をホスト OS からは使用不可能とする．その後，それらの I/O 装置に関する情報をゲスト OS に通知することより，ゲスト OS 専用として使用することが可能となる．すなわち，ホスト OS からは，ゲスト OS が使用している I/O 装置はナノカーネルローダが占有しているように見える．

次に，割り付けた I/O 装置をゲスト OS から利用可能とするためには，I/O 装置からの割込み処理を振り分ける必要がある．すなわち，ナノカーネルは，ホスト OS に割り付けられた I/O 装置からの割込みはホスト OS に，ゲスト OS に割り付けられた I/O 装置からの割込みはゲスト OS へ，と振り分けなければならない．これを実現するため，割込み発生時に制御の移行先を指定するテーブルである割込みテーブルを書き換え，割込み発生時に上記の振り分け処理を実施するプログラムが最初に起動されるように設定する．

上記の処理は，**図 4** に示すように，たとえば，IA-32 アーキテクチャ[12] 上の Linux ナノカーネルでは，I/O 装置のリソースとして IRQ と I/O アドレスを用いて I/O 装置を確保する．また，IA-32 の割込みハンドラを登録するテーブルである IDT（割込みベクタテーブル）を書き換え，すべてのハードウェア割込みをナノカーネルがインターセプトする．そして，割込み種別（IRQ）を判別し，それを所有する OS を判定して，該当する OS の割込みハンドラルーチンを呼び出す．そのために，ナノカーネル内にホスト OS 用の IDT およびゲスト OS 用の IDT を用意している．

4.3 OS ディスパッチ処理

ナノカーネルは，ホスト OS，ゲスト OS をディスパッチして各 OS に制御を渡す．この方法は，通常の OS がプロセスをディスパッチする方式と同様である．すなわち，OS ごとにコンテキストを用意しておき，時分割などのアルゴリズムに従って，コンテキスト切替え（OS 切替え）を行う．コンテキスト切替えの実行中は，ナノカーネルはホスト OS またはゲスト OS のコンテキストのどちらかで動く．次に，ナノカーネルが両 OS からの呼び出しによって，メッセージ通信などのサービスを提供するときは，それぞれの OS のコンテキストの延長で動くので，通常の処理ではナノカーネル自身のコンテキストは必要としない．ナノカーネル自身は，つねにどちらかの OS のコンテキストを利用して動作している．

ここでの課題は，障害発生にともなう OS 再起動中は，その OS の領域の中にあるコンテキストを利用できない点にある．障害が発生した OS では OS のコンテキスト自体が破壊されている危険性があるのがその理由である．そこで，ナノカーネルでは，緊急用のナノカーネル専用のコンテキストを用意しておき，OS のダウンを検知した際にこの緊急用コンテキストへ切り替えた後，OS の再ローディング処理を実行する．

以上の処理は，たとえば，IA-32 アーキテクチャでは，汎用レジスタ群（EAX，ECX など）と，OS がメモリ空間の管理などに使用するシステムレジスタ群（GDTR，CR3 など）をコンテキストとして使用することで実現する．

4.4 共有メモリと OS 間通信機能，擬似不揮発メモリ

共有メモリと OS 間通信機能は，I/O 処理として実装する．ナノカーネル・ローダは，ホスト OS からはドライバとして見えるため，そのドライバが使用する特別な I/O 装置への I/O 処理として，共有メモリ機能とそれを用いた OS 間通信機能を実現できる．

共有メモリの物理領域はナノカーネルのローディング処理の際に確保されている．その仮想領域を該当する I/O 装置用の I/O 空間の中に割り付け，あらかじめ用意した共有メモリ用物理領域をマッピングするようアドレス変換テーブルエントリを修正する．これにより，ホスト OS から見た共有メモリ領域の仮想アドレスと，ゲスト OS から見た共有メモリ領域の仮想アドレスが異なっている場合でも，同一の共有物理領域をマッピングすることが可能である．

ナノカーネルは，共有メモリの応用として，OS 間メッセージパシング機能と OS 間プロセス同期機能を提供している．いずれの機能も，共有メモリ上に制御テーブルを配置することで，ホスト OS からもゲスト OS からも操作することが可能である，という特徴を

表 2 ナノカーネルが提供する OS 間連携機能の例
Table 2 Examples of intercommunication function.

#	コマンド	機能
1	NK_SHM_MAP	共有メモリのマップ
2	NK_SHM_UNMAP	共有メモリのアンマップ
3	NK_MSG_OPEN	メッセージキューのオープン
4	NK_MSG_CLOSE	メッセージキューのクローズ
5	NK_MSG_CONNECT	通信相手接続待ち
6	NK_MSG_SEND	メッセージの送信
7	NK_MSG_RECV	メッセージの受信
8	NK_SYN_WAIT	イベント待ち
9	NK_SYN_POST	イベント通知

利用して実現している．また，共有メモリは擬似不揮発メモリとして見ることもできる．システム内で一意に定まる識別子を付加して共有メモリをアクセスすることで，OS 再起動後も同一の領域をアクセスすることが可能であるため，OS からはあたかもこの共用メモリが不揮発メモリであるように見える．

表 2 に，Linux ナノカーネル上に実装した OS 間連携機能の一例を示す．ここでは，I/O 装置の制御を指示するシステムコール（ioctl）を利用し，その引数に表 2 に示したコマンドを処理要求機能として指定できるようにしている．

4.5 障害監視，障害回復機能

ナノカーネルの特徴の 1 つに，OS の相互監視と障害発生時の OS 再起動処理がある．これらの機能は，これまでに説明した，メモリ分割，I/O 分割機能，および OS 間連携機能を用いて実現できる．

ここでの課題は OS の再起動である．

OS を再起動するためには，OS のバイナリコードを新たにローディングする必要があるが，ナノカーネル自体は I/O 装置を持たないため通常ファイルからのローディングは不可能である．そこで，未使用の物理メモリ領域を RAM ディスクとして使用し，OS のバイナリコードを格納することで OS の再ローディングを可能とした．

再起動時の他の課題は，再起動時は，通常の起動時とは初期設定処理が異なる，という点である．たとえば，タイマなどはすでにナノカーネルで使用しているので，初期設定してはならない．これを解決するため，初期設定処理をスキップしたポイントに制御を渡すことで実行してはならない初期設定処理を回避する．

たとえば，Linux ナノカーネルでは，Linux カーネルがプロテクトモードに移行し，プロセッサの設定を終えた後に実行される初期化関数のエントリポイント（start_kernel）から再スタートさせる．

表 3 ナノカーネルの開発ステップ数
Table 3 No. of steps required for Nanokernel implementation.

単位：kstep

#	機能	共通部	Windows 固有部	Linux 固有部
1	ローディング、初期設定	0.2	4.2	4.6
2	割り込み処理	2.0	0.4	0.2
3	タイマ処理	1.4	0.1	0.1
4	共有メモリ	1.6	0.3	0.2
5	OS切り替え処理	1.0	0.2	0.2
6	OS間連携処理	4.0	0.3	0.2
7	再起動処理	0.0	0.3	0.5
8	合計	10.2	5.8	6.0

4.6 ナノカーネルのホスト OS 依存性

ナノカーネルは，概念的にはホスト OS およびゲスト OS から独立しており，ホスト OS，ゲスト OS に依存しない．しかし，本章で説明したように，実装では，ホスト OS の機能を利用している部分があり，ホスト OS に依存している．これは，開発工数と開発期間を最小化するための選択である．以上のことから，ナノカーネルの実装はホスト OS ごとに異なる．

表 3 にホスト OS ごとのナノカーネル実装のステップ数を，共通部分とホスト OS 依存部に分けて示す．表 3 から分かるように，ナノカーネルの実装は，ローディングや初期設定などのように，ホスト OS の機能を利用して実現している部分はホスト OS への依存性が高い．しかし，初期設定完了後の部分に関しては共通部分は 80%以上であり，OS 依存部はインタフェースの調整のための改造に限定される．総実装ステップに対する OS 依存部分はホスト OS により異なるが 40%程度であり，共有部分は全体の 60%程度である．これらの合計は 16 kstep 程度であり，少ないステップ数で仮想計算機機能と OS 間連携機能を実現している．

5. ナノカーネルの評価

5.1 Linux ナノカーネルの性能評価

Linux をホスト OS とした Linux ナノカーネルのオーバヘッドを評価した．ここでは，ナノカーネルが動作していることによるオーバヘッドを測定するため，以下の 3 つの環境で測定を実施した．

(1) ナノカーネル未インストール環境

ナノカーネルをインストールせず，Linux がベア状態で動く環境．

(2) ナノカーネルインストール済みかつ未起動環境

ナノカーネルをインストールしているため，ホスト OS が使用できる物理メモリ量に制限がある環境．

測定環境	
HITACHI FLORA 330 (Pen. III 933MHz,128MB) RedHat LINUX 6.2J	
測定項目	Byte Benchmarks3.1
(1) 四則演算性能 (type = double)	
(2) 整数演算性能 (Dhrystone 2)	
(3) 新規プロセス起動性能	
(4) ファイルコピー性能(30秒)	
(5) 子プロセスへの双方向パイプを生成する際のオーバーヘッド	
(6) 負荷性能 (8同時実行)	
(7) 6結果の平均	

図 5 ベンチマークテスト環境
Fig. 5 Benchmark environment.

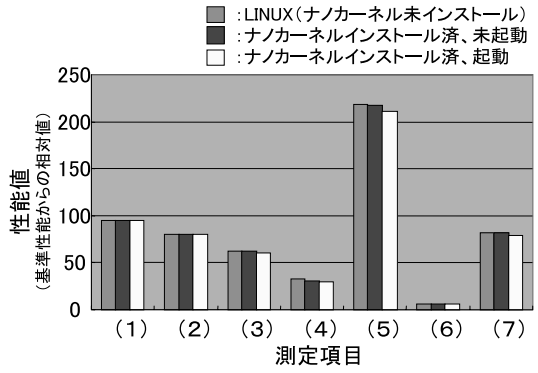

図 6 ベンチマークテスト結果
Fig. 6 Benchmark results.

(3) ナノカーネル起動環境

(2) に加え，ナノカーネルが起動されているため，すべての割込みをナノカーネルがインターセプトし，ホスト OS に転送している環境．

測定環境を**図 5** に示す．測定には，UNIX 系の OS 性能評価に広く使用されているベンチマークプログラムである Byte Benchmarks[13] を使用した．このベンチマークは図 5 に示す 7 種類の負荷プログラムを使用している．

測定結果を**図 6** に示す．図 6 より明らかなように，ナノカーネルのオーバヘッドはたかだか 2%程度であり，資源分割機能の効果を表している．

また，Linux ナノカーネルの大きさは約 60 KB であり，オーバヘッド少なく複数 OS の共存環境を構築する，という目的を達成している．

5.2 Windows ナノカーネルによる機能拡張性の評価

ナノカーネルの有効性を検証するため，Windows NT ®をホスト OS とする Windows ナノカーネル上に，ゲスト OS として独自リアルタイム OS を共存させた DARMA（Dependable Autonomous Real-time Management）システムを開発し[14]，評価した．ここでは，汎用 OS である Windows NT を制御システムに適用することを目的に，共存する OS として，監視制御向けのコントローラ用高信頼リアルタイム OS である CPMS（Compact Process Monitoring System）を選択した．これにより，リアルタイム OS のリアルタイム性および堅牢性と，Windows NT の豊富な GUI と開発環境をともに利用することが可能となり，あたかも Windows NT にリアルタイム制御機能を追加したかのように利用することが可能となる．また，万一，Windows NT に障害が発生した場合には，リアルタイム OS のみの実行継続が可能であり，かつ，Windows NT のみの再起動を行えるようにして連続的なシステム運用を可能としている[15)~18)]．

(1) リアルタイム性強化

Windows NT のタイマ機構は 10 ミリ秒周期であり，これより短い周期の要求には対応不可能である．また，リアルタイム用のプロセスをきめ細かく制御することはできない．さらに，ネットワークなどに関する割込み処理が集中し，CPU の負荷が増加した場合には応答性が大幅に悪化する．

そこで，ナノカーネルによりリアルタイム OS を共存させて，システムとしてのリアルタイム性を向上させ，電力，交通，鉄鋼などの制御システム分野での利用を可能とした．

ここでの課題は，Windows NT とリアルタイム OS 間の CPU の競合である．すなわち，Windows NT が CPU を占有してしまうとリアルタイム OS に制御がわたらず，リアルタイム OS の応答性が悪化する．そこで，DARMA システムでは，ゲスト OS であるリアルタイム OS に最高の優先度を与えている．すなわち，Windows NT 実行中にリアルタイム OS 用の割込みが発生した場合には，Windows NT がどのようなモードで実行していても強制的に制御を奪い，リアルタイム OS に制御をわたしている．ホスト OS である Windows NT はゲスト OS がアイドルのときにのみ制御を受け取る．

以上の制御により，リアルタイム OS が Windows NT と共存した場合にも，リアルタイム OS を単独実行時と同様の応答時間を確保し，リアルタイム性を保証することができる．

図 7 に，Windows NT 単独実行時と DARMA システムの応答時間の比較を示す．ここでは，プロセスを指定時刻に起こす処理において，指定した時刻と実際にプロセスが起こされた時間との誤差を応答時間として測定した．また，測定環境を高負荷状態とするた

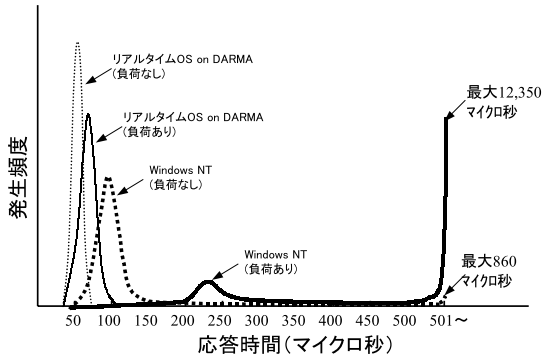

図 7 DARMA の効果（応答時間比較）
Fig. 7 Response time comparison.

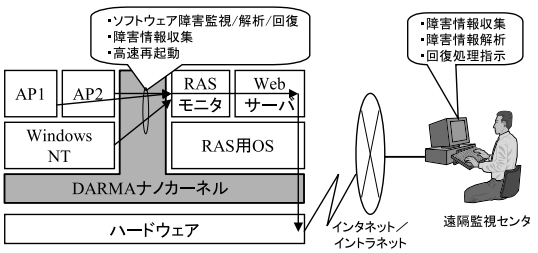

図 8 DARMA を用いた遠隔監視システム
Fig. 8 Remote monitoring system using DARMA System.

め，ディスクアクセスを頻発する低優先度のプロセスを共存させている．図 7 で，負荷なしの状態とは測定対象となっているプロセスのみが存在する状態であり，負荷ありの状態とは I/O がつねに実行中の状態であり，I/O 起動処理と I/O 完了割込み処理が頻発している状態である．

図 7 から分かるように，Windows NT 単独実行では，無負荷状態では応答時間の最頻値が 100 マイクロ秒，最大応答時間が 860 マイクロ秒であったものが，負荷状態では応答時間の最頻値が 240 マイクロ秒，最大応答時間は 12 ミリ秒以上と，負荷状態時のリアルタイム性が大きく悪化している．これに対して，DARMA システムのリアルタイム OS では，応答時間の最頻値は無負荷状態の 60 マイクロ秒から負荷状態では 70 マイクロ秒へ，最大応答時間は 80 マイクロ秒から 110 マイクロ秒へと，応答時間の延びは数十マイクロ秒にとどまっており，安定した応答時間を達成している．

(2) 信頼性向上

ナノカーネル技術の特長の 1 つに，OS 障害時の監視/解析/回復機能がある．**図 8** にその構成例を示す．ここでは，Windows ナノカーネルのゲスト OS として，ホスト OS である Windows NT およびその上で稼動するアプリケーションプログラムの動作を監視する小規模な高信頼 OS（RAS 用 OS）を共存させている．万一 Windows NT に障害が発生した場合には，RAS 用 OS 上の RAS モニタが障害を検知する．RAS モニタはナノカーネルを介して Windows NT の障害情報を収集することができ，障害発生原因の解明を迅速に行うことが可能である．さらに，Windows NT のみの再起動も短時間で実施することができるため，障害の影響を最小限度に抑えることができる．また，障害情報は RAS 用 OS 上の Web サーバを利用して遠隔地の Web クライアントから参照，解析することが可能であるため，集中監視によるコスト削減と障害解析時間の短縮を図ることができる．また，RAS 用 OS の障害がかえってシステムの信頼性に悪影響を与えることのないよう，RAS 用 OS はこれまで 20 年以上の稼動実績のある制御用 OS をベースとして，十分な検査を行い開発した．これにより，万一 Windows NT に障害が発生した場合でも，状況を迅速に把握することができ，さらに，高速に再起動することにより障害発生の影響を最小限度に抑えることができ，システム全体の信頼性を確保することができる．

ここに述べた信頼性向上についての説明は文献 18) により詳細に記述しているので，必要に応じて参照されたい．

以上のように，ナノカーネルを用いて，ゲスト OS として特徴ある独自 OS を共存させることにより，容易に機能拡張を実現することが可能となる．本章で示した拡張は，たとえば，製鉄所における制御システムなどのように，リアルタイム性と信頼性が要求される制御システムを，汎用 OS と専用 OS の連携により実現できることを示している．従来，このようなシステムは専用のアーキテクチャと専用の OS で実現してきたが，ナノカーネル方式により汎用 OS を利用することができ，短期間かつ低コストで開発することが可能となる．

6. 関連研究との比較

VMM 機能はこれまでにも多くの研究がなされており，特に近年，本論文と同様に IA-32 をベースとした VMM が紹介されている[19]．

VMWare ESX Server[20] は，エミュレーションを用いて VMM 機能を実現している．ここでは，メモリの仮想化を実現するため，VM 上の OS がメモリの管理テーブルであるページテーブルや TLB を操作するために発行する命令をトラップし，その操作をエミュレーションしている．また，実際にマシンに装備

された物理メモリを有効活用するため，ballooning 技術や，同一の内容を保持しているメモリを VM 間で共用する機能を採用している．

一方，ナノカーネルでは，メモリは VM 起動時に固定的に連続して割り付けることで，実行時のメモリの管理に必要なアドレス変換処理やメモリの参照履歴管理が不要とし，CPU オーバヘッドを小さく抑えている．また，ナノカーネルが提供する共用メモリは VM 間のデータ共用や OS 間の機能連携を低オーバヘッドで実現できる点に特徴がある．これにより，ナノカーネルでは，複数の OS が相互に機能補完することで単一の OS では実現不可能であった機能を 1 台のマシン上で実現させることを可能としている．

VMware Workstation [21] は，入出力装置の仮想化を簡便に実現するため，ホスト OS 上のドライバを用いてゲスト OS 側の入出力処理を仮想化している．ここでは，ゲスト OS が発行した入出力操作を実行する命令をトラップして，ホスト OS に切り替えることにより仮想的な装置をエミュレーションしている．しかし，入出力命令のトラップとホスト OS への切替えオーバヘッドが大きい，という欠点がある．

一方，ナノカーネル方式ではメモリおよび I/O 装置をゲスト OS またはホスト OS に占有させている．そのため，1 台の装置を OS 間で共用したり，仮想的な装置を VM に提供したりすることはできない．しかし，装置に対する入出力処理を直接実行するため，オーバヘッドは割込み発生時の OS 振り分け処理のみである．これにより，5 章に示したように，リアルタイム OS のような QoS 保証が要求される OS の搭載，共存も可能としている．

VM 実行時のオーバヘッド削減や VMM の開発工数を削減する目的で，VM 上で稼動するゲスト OS を修正することを前提とした方式もいくつか報告されている．Xen [22] は，VM 上の OS が仮想化された計算機資源を操作するためのインタフェースを設け，OS がそれらのインタフェースを使用するよう改造することで，計算機資源の仮想化を簡便化し，実行時のオーバヘッドを削減しようとするものである．ここでは，計算機資源であるメモリ，プロセッサ，I/O 装置仮想化するため 8 種類のインタフェースを用意している．これを利用して，Linux OS を Xen 上で実行させるために必要な修正量は 3 kstep である．

Xen の方式は，ゲスト OS を最小限修正することで性能を損なうことなく VM 上で実行させる，という基本的な方針はナノカーネルと同様である．しかし，Xen がゲスト OS に提供するインタフェースが多岐にわたるのに対し，ナノカーネルではメモリマップと割込みに関する部分のみである．そのため，Xen では OS の修正は 3 kstep が必要であるが，ナノカーネルでは 0.5 kstep である．また，Xen では Xen 上で稼動するすべての OS を修正する必要があるが，ナノカーネルではホスト OS は修正する必要がない．

Cooperative Linux [23] も，Xen と同様に，ゲスト OS の修正を必要としている．しかし，Cooperative Linux は Xen とは異なり，ホスト OS に共存する形で特権モードで動作するゲスト OS を動作させる．そのため，ゲスト OS はホスト OS の特殊なデーモンとして動作する．ゲスト OS の修正は，ドライバやメモリ管理割込み処理など多岐にわたり計 3.8 kstep の修正が必要であり，ナノカーネルの 0.5 kstep に比べ大きい．

ナノカーネルはホスト OS とゲスト OS が存在する点では Cooperative Linux と同様であるが，ナノカーネルではゲスト OS のローディング時にのみホスト OS の機能を利用し，ローディング以降はゲスト OS はホスト OS から独立している点が異なる．これにより，Cooperative Linux ではゲスト OS の実行時にホスト OS との間の切替え処理が発生する．

以上の従来方式は，いずれも VM の独立性を保つためのみの方式であるのに対し，ナノカーネルでは，これに加えて，ゲスト OS の性能確保とホスト OS とゲスト OS 間の連携強化による機能拡張の効率化を可能とする機能を提供している．これにより，オーバヘッドが少なく QoS の保証も可能となり，OS 間連携による新機能を容易に実現する点に特徴がある．

7. おわりに

汎用 OS と専用 OS を 1 台のマシン上に共存させ，互いに機能補完することで，多様化する計算機システムへの要求に短期間で対応できるナノカーネルについて述べた．OS 間の機能補完を実現する必要かつ十分な機能として，資源分割機能，OS 間通信機能，障害監視，回復機能，を選択し，ナノカーネル機能を実現した．開発後，数ケースへの適用と評価により，上記の機能のみで OS 間の機能補完による新システムを高効率に開発できることを確認した．

今後，汎用 OS はますます巨大化し，汎用 OS を独自に修正，保守することはほとんど不可能となりつつある．ナノカーネルはここに示したように，汎用 OS を補完するリアルタイム性，高信頼性などを強化した OS 機能を短期間に効率良く共存する方式として有効と考える．

(Windows NT ®は，米国 Microsoft Corporation の米国および他の国の登録商標です)．

参 考 文 献

1) The Mach Project. http://www-2.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html
2) 岡崎世雄ほか：VM，共立出版株式会社，ISBN4-320-02405-2.
3) Goldberg, R.P.: *Architectual Principles for Virtual Computer System*, Center for Research in Computing Technology, Harvard University (1971).
4) *Proc. Workshop on Virtual Computer Systems*, ACM, SIGARCH-SIGOPS (1973).
5) Bugnion, E., et al.: Disco: Running Commodity Operating Systems on Scalable Multiprocessors, *Proc. SOPS-16* (1997).
6) 新井ほか：ナノカーネル方式による異種 OS 共存技術「DARMA」の提案，情報処理学会第 59 回全国大会講演論文集（1），pp.139–140 (1999).
7) 佐藤ほか：ナノカーネル方式による異種 OS 共存技術「DARMA」の実装，情報処理学会第 59 回全国大会講演論文集（1），pp.141–142 (1999).
8) 木村ほか：異種 OS 共存環境ナノカーネル方式の Linux への適用，情報処理学会第 61 回全国大会講演論文集（1），pp.1-41–1-42 (2000).
9) 佐藤ほか：異種 OS 共存環境ナノカーネル方式の応用，情報処理学会第 61 回全国大会講演論文集（1），pp.1-43–1-44 (2000).
10) Bovet, D.P., et al.: *Understanding the LINUX Kernel*, O'REILLY, ISBN 0-596-0002-2.
11) Rubini, A., et al.: *Linux Device Drivers*, O'REILLY, ISBN 0-59600-008-1.
12) *IA-32 Intel Architecture Software Developer's Manual*, Intel.
13) Byte Unix Benchmarks. http://www.tux.org/~mayer/linux/bmark.html
14) Solomon, D.A.: *Inside Windows NT*, Microsoft Press, ISBN 1-57231-677-2.
15) 木村：PLC 市民権を得るソフトロジック，日経ディジタルエンジニアリング，1999 年 1 月 (1999).
16) 宮崎ほか：オープンでかつ頼れる新計測制御システム，計測制御，1999 年 4 月 (1999).
17) 新井ほか：異種 OS 共存技術「DARMA」の開発と制御システムへの適用，計測技術，Vol.27, No.7, pp.39–44, 日本工業出版 (1999).
18) 新井ほか：情報制御システム統合のための高信頼化技術，日本信頼性学会誌「信頼性」，Vol.22, No.5, pp.396–403 (2000).
19) Robin, J.S. and Irvine, C.E.: Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor, *Proc. 9th USENIX Security Symposium*, pp.129–144 (2000).
20) Waldspurger, C.A.: Memory resource management in VMware ESX server, *Proc. 5th symposium on Operating systems design and implementation* (*OSID 2002*), pp.181–194 (2002).
21) Sugerman, J., Venkitachalam, G. and Lim, B.-H.: Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor, *Proc. 2001 USENIX Annual Technical Conference* (2001).
22) 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, *Proc. 19th ACM symposium on Operating systems principles*, pp.164–177 (2003).
23) Aloni, D.: Cooperative Linux, *Proc. Linux Symposium* (2004). http://www.colinux.org/

(平成 16 年 8 月 23 日受付)
(平成 17 年 9 月 2 日採録)

新井 利明（正会員）

1976 年早稲田大学理工学部電気工学科卒業．1978 年同大学院理工学研究科電気工学専攻修了．同年（株）日立製作所システム開発研究所入所．以来，大型汎用 OS VOS3，UNIX，Windows，Linux 等のオペレーティングシステムの研究開発に従事．近年はストレージシステム，オフィスシステムにも興味を持つ．

関口 知己（正会員）

1970 年生．1992 年北海道大学工学部情報工学科卒業．1994 年同大学院工学研究科情報工学専攻修了．同年（株）日立製作所システム開発研究所入社．オペレーティングシステム，計算機システムの高信頼化に関する研究．特に，マルチ OS 共存技術，Windows による監視制御システムの高信頼化に関する研究開発に従事．

佐藤　雅英（正会員）

1965 年生．1990 年東京理科大科学大学院理工学研究科修士課程修了．同年（株）日立製作所システム開発研究所に入所．1990〜2002 年オペレーティングシステムの研究開発に従事．2002 年〜現在，ストレージシステムの運用管理技術の研究開発に従事．

木村　信二（正会員）

1983 年日立京浜工業専門学院ソフト工学科卒業．現在，（株）日立製作所システム開発研究所主任研究員．UNIX ワークステーションの基本ソフトウェア，ウィンドウシステムおよびストレージ応用システム等の研究開発に従事．Linux オペレーティングシステム，オープンソースソフトウェアに興味を持つ．

大島　　訓（正会員）

1972 年生．1998 年東京理科大学大学院理工学研究科情報科学専攻博士課程前期修了．現在の所属は Hitachi Computer Product（America）Inc. で，Massachusetts 州 Westford の Red Hat Corporation 内在勤．Linux Operating System の Scalability および信頼性の向上に取り組んでいる．主な研究テーマは，PC 向け Virtualization 技術，PC 向け OS の設計と実装等．参加している Open Source Project は，System Tap, Kexec-base crushdump，Linux-MM 等である．

吉澤　康文（フェロー）

1967 年東京工業大学卒業．同年（株）日立製作所中央研究所に勤務．HITAC 5020/TSS の研究開発に従事．1973 年システム開発研究所．仮想記憶，大規模 TSS，オンラインシステム等，大型計算機の性能向上と評価，記憶管理方式，OS テスト・デバッギングシステム，ハイエンドサーバ，超並列計算機，リアルタイムシステム等の研究開発を推進．東京農工大学大学院教授（工学博士）．当学会フェロー．