社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2004－OS－95 (10)
2004／2／27

Pico-Kernel を用いた OS デバッグ方法

竹内 理
（株）日立製作所システム開発研究所

近年、PC/AT 互換機上に高速 I/O 機能を持つ独自 OS を開発するニーズが増大している。しかし、従来の PC/AT 互換機上で動作する OS 向けの開発環境は、開発環境の安定稼働が保証できる、様々な OS や I/O デバイスに大きな開発なく適用できる、デバッグ時にも高い I/O 性能で動作させることができる、の 3 条件を同時に充足することができなかった。本論文では、これらの 3 条件を同時に充足する独自 OS 開発方式として、Pico-Kernel という小型組み込みモジュールを用いた独自 OS 開発方式を提案し、実装を行う。そして、上記条件の同時充足には、独自 OS からのハードウェアアクセス資源方式を複数準備して資源の種類ごとに使い分ける、などの実装上の工夫が有効であったことを示す。さらに、本開発方式を実際に 3 種類の既存 OS に適用した結果、必要な開発コード量は 2K 行以下に抑えられたことも示す。最後に本開発方式によるデバッグ時の I/O 性能の定量的な評価も行う。評価の結果、デバッグ時にも、I/O 性能の劣化は 9.5% 程度に抑えられることが明らかになった。

OS Debugging Method Using Pico-Kernel

Tadashi Takeuchi
Systems Development Laboratory, Hitachi, Ltd.

Recently, demands for implementing original operating systems which can achieve high I/O performance on PC/AT compatible hardware have been increasing. However, conventional operating system development environments have not been able to satisfy the following demands at the same time: 1) assuring the stability of the development environments even if the operating system under development does not execute proper processing due to operating system bugs, 2) an easy customization of the development environment for new operating systems and new I/O devices, 3) efficient I/O execution of the operating system under the development environment.

In this paper, we propose a novel operating system development method and environment using the Pico-Kernel. We show that the Pico-Kernel should have some features such as hardware accessing method optimization in order to satisfy the above three demands. We also added modifications to three existing operating systems so that the operating system can work under this proposed environment, and we confirmed that the necessary customizations were below 2K steps in each operating system. Finally, we evaluated I/O performances of an operating system working under this environment, and confirmed that the I/O performance degradation caused by this environment is only 9.5%.

1. はじめに

近年、PC/AT 互換機上に自社開発した独自 OS を搭載し、I/O 性能において差別化をはかるアプライアンスサーバ製品が数多く市場に出回ってきている。Novel 社のキャッシュサーバ製品である Volera Media Excelarator [1][1]はその典型例である。また、Kasenna Streaming Accelerator [2][2]のように、独自改変を加えた Linux を用いている製品もある。日立グループにおいても、自社開発したストリーミング専用 OS HiTacti [3～5] を搭載した映像配信サーバ [6] を製品化している。

これに伴い、PC/AT 互換機上で高速 I/O 機能を提供する独自 OS を開発するニーズが増大している。ICE 等のデバッグ用ハードウェアが提供されていない PC/AT 互換機では、従来、OS の開発環境として、

1) 汎用 OS 上に構築されたハードウェアシミュレータ（または仮想計算機）及び当該ハードウェアシミュレータと連動するソフトウェアデバッガ [7～9]
2) ソフトウェアリモートデバッガ¥cite [10]
3) OS 内部に組み込まれた当該 OS 専用のデバッガ [11]

などを利用していた。しかし、上記のはいずれも、A) 開発環境の安定稼働が保証できる、B) 様々な OS や I/O デバイスに大きな開発なく適用できる、C) デバッグ時にも高い I/O 性能で OS を動作させることができる（結果として、OS の I/O 性能エンハンス時にも開発環境が利用できる）、の 3 条件を同時に充足していない。(1) は、I/O デバイス（または I/O プロセッサ）の動作シミュレーションをハードウェアシミュレータ（または仮想計算機）にて行うので、B) C) の条件を充足できない。ハードウェアシミュレータ（または仮想計算機）でも、I/O 処理以外であれば、実計算機上での命令の直接実行によりある程度の高速動作を行える。しかし、I/O デバイス（または I/O プロセッサ）の動作シミュレーションを、そのデバイスの仕様に依存せずに高速実行することが困難であるため、I/O の高速実行を保証できなかった。(2) は、OS のバグがリモートデバッグ動作（例えばシリアル通信）を阻害する可能性があるため A) の条件を充足できない。また、(3) は、OS が

[1] Volera Media Excelerator は米国 Novel 社の登録商標です。
[2] Kasenna Streaming Accelerator は米国 Kasenna 社の登録商標です。

変わると多大な開発が必要となる上、OS とデバッガ間のメモリ保護等も実現できていないため、A) B) の条件を充足できない。

本研究の目的は、上記 3 条件を同時に充足する、PC/AT 互換機上で動作する独自 OS の開発方式として、Pico-Kernel を用いる方式を新規に提案することにある。さらに、この開発方式を複数の OS (HiTactix、μITRON 仕様 OS[12]、BSD/OS[13]) に実際に適用し、必要な開発量を検証する。最後に、本開発方式によるデバッグ時の I/O 性能の定量的な評価を行う。

以下、2 章において、新規に提案する Pico-Kernel を用いた独自 OS 開発方式の概要について述べる。さらに、3 章において、提案方式の中核をなす Pico-Kernel を実現するにあたり解決すべき技術課題と、その解決策について明らかにする。次に、4 章において Pico-Kernel の実装の概要について述べる。5 章では、本開発方式の既存 OS への適用事例について述べ、この適用が容易に行えたことを明らかにする。最後に 6 章で、本方式の開発環境上で動作する独自 OS の I/O 性能の評価結果について述べる。

2. Pico-Kernel を用いた独自 OS 開発方式の概要

本章では、本論文で新規に提案する Pico-Kernel を用いた独自 OS 開発方式の概要について説明する。

本開発方式は、PC/AT 互換機上で動作する独自 OS の開発の際に、以下の条件を充足する開発環境を提供することを目標としている。

1) 開発環境の安定稼働が保証できる。
2) 様々な OS や I/O デバイスに大きな開発なく適用できる
3) デバッグ時にも高い I/O 性能で動作させることができる

上記目的を達成するために、本開発方式では、図 1 に示す構成を持つ開発環境を提供する。

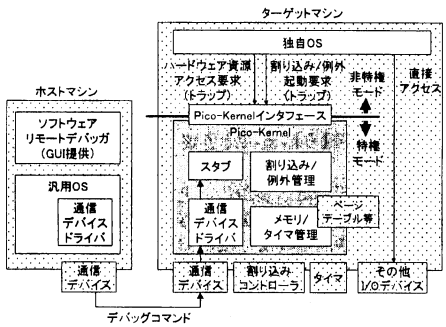

図 1 開発環境の構成（機能デバッグモード）

本開発環境は、従来のソフトウェアリモートデバッガを利用する際の開発環境と類似した構成を持つ。開発環境は、ホストマシンとターゲットマシンからなり、ホストマシン上ではリモートデバッグ機能を持つソフトウェアデバッガが動作する。ソフトウェアデバッガは、デバッグコマンド（ターゲットマシンのメモリ参照／更新、レジスタ参照／更新など）をユーザから受け取り、当該コマンドをターゲットマシンに転送する。

しかし、ターゲットマシン上で、独自 OS とは独立に開発された Pico-Kernel が予め開発環境として組み込まれている点が従来のソフトウェアリモートデバッガと異なる。Pico-Kernel は、上記デバッグコマンドの受信、コマンドの実行、コマンドの実行結果の返信の際に必要となる機能のみを提供する。具体的には、リモートデバッガのスタブ、通信デバイスドライバ、割り込み／例外管理機能、メモリ管理機能、タイマ管理機能を提供する。Pico-Kernel は、上記機能を実現するため、ページテーブル、割り込みコントローラ、タイマデバイス等のハードウェア資源を管理する。上記機能は CPU を特権モードで動作させることにより実現する。

さらに、ターゲットマシン上では、開発すべき独自 OS が動作する。独自 OS は非特権モードにて動作をする。Pico-Kernel が管理するハードウェア資源へのアクセスや、独自 OS を動作させるために必要となる割り込み処理や例外処理の起動は、独自 OS 自身が実行せずに Pico-Kernel に代理実行を要求する。Pico-Kernel は上記要求を受け付けるためのインタフェースを独自 OS に提供する。但し、Pico-Kernel が利用しない I/O デバイス（例えば SCSI コントローラ、Ethernet[3]カードなど）は、Pico-Kernel を介さず、独自 OS から直接アクセスする。

このような構成をとれば、開発する独自 OS は非特権モードで動作し、Pico-Kernel の動作を阻害しないため、目標 (1) が達成できる。また、通信デバイス以外の I/O デバイスへのアクセスは、Pico-Kernel を介さずに独自 OS が直接行うため、I/O デバイスが代わっても Pico-Kernel 部分を改変する必要はない。また、既存 OS への本開発方式の適用も、既存 OS の一部を Pico-Kernel 提供インタフェースを利用するように改変するだけで可能になり、目標 (2) も達成できる。この時必要となる既存 OS の改変量の詳細については、5 章で明らかにする。

しかし、図 1 では、目標 (3) を部分的にしか達成できない。独自 OS から I/O デバイスへの直接アクセスは実現できているものの、独自 OS から Pico-Kernel が管理するハードウェア資源（割り込みコントローラなど）へのアクセス要求等を発行する際にトラップ発行が必要となり、I/O 性能の劣化が予想されるためである。このため、本開発方式では、独自 OS の初期段階の機能レベルのデバッグ時には図 1 の構成（以下「機能デバッグモード」の構成、と呼ぶ）で動作させるが、開発が進み I/O 性能のエンハンスを行う段階では、図 2 の構成（以下「性能デバッグモード」の構成、と呼ぶ）に変更する。具体的には、Pico-Kernel の部分を Pico-Kernel ライブラリに置き換え、独自 OS を非特権モードではなく、特権モードで動作させる。Pico-Kernel ライブラリは Pico-Kernel と同一インタフェースを提供させ、この構成変更に伴う独自 OS の改変を不要にする。但し、この構成変更後は、独自 OS の機能レベルのバグにより、開発環境が正常稼働しなくなる可能性が生じる。構成を変更した後に機能レベルの不具合が判明した場合は、再び機能デバッグモー

[3] Ethernet は、米国 Xerox 社の商品名称です

ドの構成に戻して独自 OS のデバッグを行う必要がある。

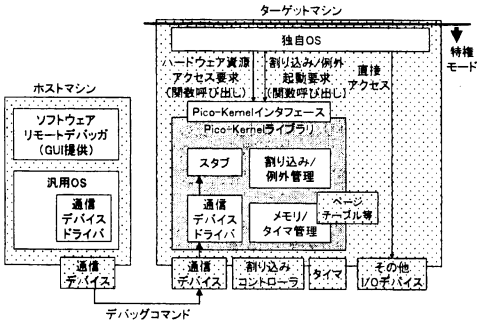

図 2 開発環境の構成（性能デバッグモード）

3. Pico-Kernel の技術課題と解決策

本章では、前章で述べた独自 OS 開発方式で使用する Pico-Kernel 及び Pico-Kernel ライブラリ（以下、「Pico-Kernel 群」と略して表記する）の実現にあたり解決する必要が生じた技術課題と、その解決策について明らかにする。

前章で示した通り、Pico-Kernel は独自 OS からいかなる不法なハードウェア資源へのアクセス要求が到達しても、Pico-Kernel によるリモートデバッグの動作を阻害しないことを保証しなければならない。一方で、Pico-Kernel ライブラリは、Pico-Kernel と同一インタフェースを提供しつつ、高速に独自 OS からのハードウェア資源アクセスを実現しなければならない。また一方で、色々な既存 OS にも前章で述べた開発方式を容易に適用可能にするため、既存 OS を Pico-Kernel 群の上で動作させる際に必要となる既存 OS の改変量を抑える必要がある。

上記を実現するため、Pico-Kernel 群の実装の際に、以下の 2 点の技術課題の解決が必要になった。

1) ハードウェア資源アクセス方式の最適化
2) スタック切替えのエミュレーションの実現

以下、これらの技術課題の詳細と、その解決策について順に述べる。

3.1. ハードウェア資源アクセス方式

Pico-Kernel は、独自 OS がいかなる不法なハードウェア資源アクセス要求を発行しても Pico-Kernel の動作を阻害しないことを保証するため、Pico-Kernel が使用するハードウェア資源を保護する必要がある。一方、Pico-Kernel ライブラリは、Pico-Kernel と同一インタフェースを保ちつつ、高速なハードウェア資源アクセスを実現しなければならない。さらに、既存 OS を Pico-Kernel 群の上で動作させる際に必要となる既存 OS の改変量を抑えるため、ハードウェア資源アクセスは、できる限り Pico-Kernel 群を介することなく、従来通りにアクセスできることが望ましい。以上の技術課題を解決するため、Pico-Kernel 群は、図 3 に示す 3 種類のハードウェア資源アクセス方式を、ハードウェア資源の種類に応じ使い分けている。さらに、同じハードウェア資源でも、Pico-Kernel と Pico-Kernel ライブラリでは異なるハードウェア資源アクセス方式を用いることもある。

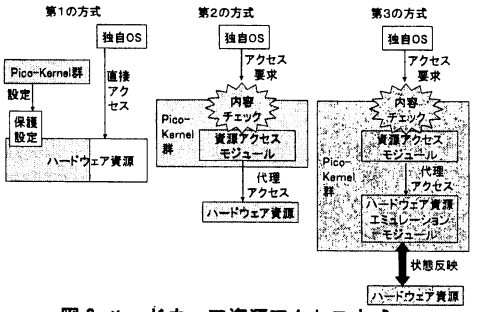

図 3 ハードウェア資源アクセス方式

第 1 のハードウェア資源アクセス方式は、Pico-Kernel 群が初期化時にハードウェアの保護設定を行うことにより、Pico-Kernel 群が使用するハードウェア資源に独自 OS が不法にアクセスすることを防ぐ方式である。独自 OS は、Pico-Kernel 群を介さずにハードウェア資源にアクセスする。

第 2 のハードウェア資源アクセス方式は、Pico-Kernel 群がハードウェア資源アクセスモジュールを保持し、上記モジュールに対するインタフェースを独自 OS に提供する方式である。独自 OS はハードウェア資源に直接アクセスする代わりに、上記インタフェースを呼び出す。

第 3 のハードウェア資源アクセス方式は、Pico-Kernel 群がハードウェア資源のエミュレーションモジュール、及び上記エミュレーションモジュールに対する資源アクセスモジュールを保持し、資源アクセスモジュールに対するインタフェースを独自 OS に提供する方式である。第 2 の方式と同様に、独自 OS はハードウェア資源を直接操作する代わりに、上記インタフェースを呼び出す。また、Pico-Kernel 群は、エミュレートされているハードウェア資源の状態を、適宜実ハードウェア資源の状態に反映させる。

第 1 の方式は、高速アクセスと、従来のアクセス方式との互換性の維持が達成できる。そのため、できる限り第 1 の方式を用いることにした。現在の実装では、物理メモリ、I/O メモリ、I/O ポートへのアクセスは、本方式により制御している。なお、Pico-Kernel と Pico-Kernel ライブラリ間で、異なるハードウェア保護設定をする場合がある。例えば、Pico-Kernel が使用する物理メモリは独自 OS からアクセス不可能な保護設定にしているが、Pico-Kernel ライブラリはこの保護を行わない。Pico-Kernel ライブラリはこの保護を行わない代わりに、第 2 や第 3 の方式で用いる資源アクセスモジュールを独自 OS から高速に呼び出す（トラップを使用しないで呼び出す）ことを可能にしている。

第 2 の方式は、従来のアクセス方式との互換性維持はできないが、第 3 の方式と比すれば高速アクセスが実現できる。また、この方式により、ハードウェアレベルでの保護設定のできない資源の保護も可能になる。そのため、第 1 の方式を利用できないハードウェア資

源に対しては、できる限り第2の方式を用いてアクセスすることにした。ページテーブル[4]、セグメントテーブル(LDT)などは本方式を用いて制御している。これらのハードウェア資源へのアクセス要求が独自OSから発行された際には、Pico-Kernel群が要求内容をチェックし、不正アクセスにはエラーを返す。このチェックにより、当該要求によるPico-Kernel群の動作阻害を防いでいる。

しかし、割り込み制御ビット(EFLAGSレジスタのIFビット)、割り込みコントローラ(8259Aチップ)に関しては第2の方式でもPico-Kernelの動作を阻害する可能性があるため、Pico-Kernelでは第3の方式を用いた。

以下、Pico-Kernelにおいて割り込み制御ビットと割り込みコントローラの制御を行う際に第3の方式が必要となる理由と、Pico-Kernelにおけるこれらのエミュレーション方法の概要について示す。独自OSが割り込み制御ビットや割り込みコントローラの制御を誤り、Pico-Kernelの動作が阻害される場合として、以下の3ケースが考え得る。

1) 独自OSが割り込み制御ビットを割り込み禁止状態に設定したまま、無限ループに陥る。
2) 独自OSが管理しているI/Oデバイスから割り込み通知が到達しているにも関わらず、独自OSが割り込みコントローラにおけるマスクを設定しない。
3) 独自OSが管理しているI/Oデバイスから割り込み通知が到達した後に、独自OSが割り込みコントローラにEOI命令を発行しない。

(1)の場合は、通信デバイスからの割り込みが発生しても、割り込み制御ビットが割り込み禁止状態のままであるため、当該割り込みをPico-Kernelが受けられず、リモートデバッグが継続不可能になる。(2)の場合も、独自OSが管理するI/Oデバイスの割り込みが発生し続け、この連続発生により通信デバイス等からの割り込みをPico-Kernelが受け取れなくなる可能性がある。(3)の場合は、EOI命令が発行されず割り込みコントローラが次の割り込み通知をCPUに行わないため、Pico-Kernelは通信デバイスからの割り込み受理を行えない。

上記いずれの場合も、独自OSがPico-Kernelに対して発行した要求が不正であったためにPico-Kernelの動作が阻害されたわけではない。このことは、第2の方式でPico-Kernelの正常動作を保証することはできず、第3の方式の適用が必要であることを意味する。

独自OSからの割り込み制御ビットと割り込みコントローラの制御要求は、表1に示すインタフェースにより受理する。

表1 割り込み制御要求一覧

インタフェース名	機能概要
pk_get_intstat	割り込み制御ビットの状態を取得する
pk_disable_int	割り込み制御ビットを割り込み禁止状態にする
pk_enable_int	割り込み制御ビットを割り込み許可状態にする
pk_get_icustat	割り込みコントローラのマスク状態を取得する
pk_mask_icu	割り込みコントローラのマスク設定をする
pk_unmask_icu	割り込みコントローラのマスクセットを解除する
pk_EOI_icu	割り込みコントローラのインサービスビットをクリアし、次の割り込み通知を可能にする
pk_iret	割り込み処理が完了したことを知らせる

Pico-Kernelが保持するエミュレーションモジュールは、割り込みのマスク状態、ペンディング状態を管理する。さらに、割り込み発生や表1に示すインタフェースの呼び出し時に、この状態を更新する。さらに、必要に応じて、独自OSへの割り込み発生通知やこの通知の保留を行う。このエミュレーションは、従来のハードウェアエミュレーションと比べると、レジスタアクセスのエミュレーション等を行う必要がないため高速実行が可能である。

なお、Pico-Kernelライブラリも表1に示すインタフェースを独自OSに提供している。しかしPico-Kernelライブラリはエミュレーションを行わない。これらの要求を受けると、割り込み制御ビットや割り込みコントローラを直接操作する。

3.2. スタック切り替えエミュレーション

Pico-Kernelは独自OSを非特権モードで動作させるが、Pico-Kernelライブラリは独自OSを特権モードで動作させる。独自OS上で動作するアプリケーションも含めた動作モードの関係は図4に示す通りになる。

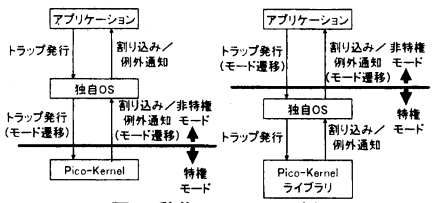

図4 動作モードの関係

この動作モードの違いは、割り込み、例外、トラップ(アプリケーションから独自OSへのシステムコール発行)発生時におけるスタック切替え動作の違いを生じさせる。PC/AT互換機で使用されているCPU(IA-32チップ)は、割り込み、例外、トラップ発生時に動作モードの遷移が起きるか否かでスタック切替えを行うか否かの動作が変わるためである。

この違いを独自OSから隠蔽するため、Pico-Kernel群から独自OSに割り込み、例外の発生を通知する時、及びトラップ発生時にスタック切替えのエミュレーションを行う。このエミュレーションにより、Pico-Kernel上で動作している時も、Pico-Kernelライブラリ上で動作している時も、割り込み、例外、トラップ発生後に独自OSに制御が遷移する際には、スタックの状態が同一になる。

また一方で、スタック切替えエミュレーションの際

---

[4] 独自OSがアクセス可能な物理メモリ領域の範囲はPico-Kernelが初期化時に決定する。しかし、独自OSはさらにページテーブルに対するアクセス要求を発行することで、アクセス可能な物理メモリ領域を自由に仮想空間にマップしたり、読み／書き／実行権限の設定を行える。

に実行中コンテキストの格納形式の変換も行うことにより、既存OSへの適用を容易にしている。既存OSは、アプリケーション定義のシグナルハンドラの起動などの際に、格納されている実行中コンテキストを操作する。そして、この際に仮定している実行中コンテキストの格納形式はOSごとに異なる。既存OSが仮定する格納形式を変更せずにPico-Kernel群上で動作させることを可能とするため、実行中コンテキストの格納形式の変換も、スタック切替えのエミュレーション時に行うことにした。

アプリケーション実行中に割り込み（または例外）が発生し、Pico-Kernel群から独自OSに割り込み（または例外）を通知する時を例にして、スタック切替えエミュレーションの動作手順を説明する。図5は、Pico-Kernel動作時のスタック切替えエミュレーション手順を示している。また、図6は、Pico-Kernelライブラリ動作時のスタック切替えエミュレーション手順を示している。

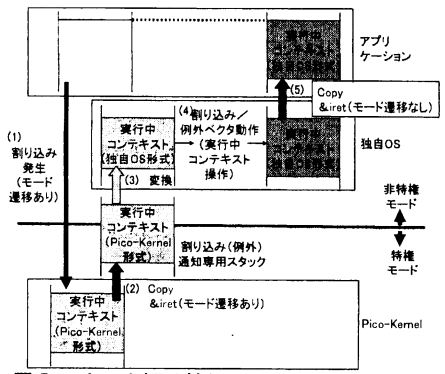

図5 スタック切り替えエミュレーション手順（Pico-Kernel）

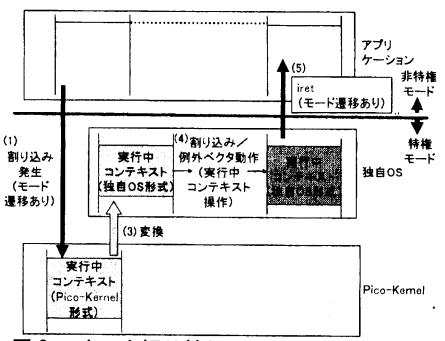

図6 スタック切り替えエミュレーション手順（Pico-Kernelライブラリ）

Pico-Kernel動作時のスタック切替えエミュレーションの動作手順は以下の通りである。

1) 割り込み（または例外）が発生すると、非特権モードから特権モードに遷移し、アプリケーションスタックからPico-Kernelスタックへのスタック切替えがCPUにより行われる。Pico-Kernelは、Pico-Kernelスタックに Pico-Kernelが仮定する形式で実行中コンテキストを格納する。
2) Pico-Kernelから独自OSに割り込み(または例外)通知を行う際に、まず割り込み（または例外）通知専用スタックに実行中コンテキストをコピーする。そして、独自OS定義の割り込み(または例外)ベクタを起動する。この起動は、(1)で格納した実行中コンテキストの操作後、ハンドラリターン命令（iret命令）を発行することにより行う。この操作の際に、格納されているスタックポインタも更新し、使用スタックを割り込み（または例外）通知専用スタックに変更する。なお、割り込み(または例外）通知専用スタックは、Pico-Kernelが物理メモリ上に確保しているスタックだが、この通知の際に使用できるように、非特権モードからもアクセス可能にしている。
3) 独自OS定義の割り込み（または例外）ベクタは、起動後すぐ、割り込み（または例外）通知専用スタックから独自OSスタックに実行中コンテキストをコピーする。この際、格納形式の変換も行う。さらに、スタックポインタを更新し、使用スタックを独自OSスタックに変更する。
4) 独自OS定義の割り込み（または例外）ベクタは、独自OSスタックを用いて動作する。シグナルハンドラ起動時などには、独自OSスタック内の実行中コンテキストを更新する。
5) 独自OS定義の割り込み（または例外）ベクタは、処理を完了したら、独自OSスタックの実行中コンテキストをアプリケーションスタックにコピーする。そして独自OS定義ベクタは、スタックポインタを更新し、使用スタックをアプリケーションスタック変更する。そしてハンドラリターン命令（iret命令）を発行する。この命令発行時にCPUの動作モード遷移は起こないので、スタック切替えはアプリケーションスタックのまま、アプリケーションの実行が再開する。

(2)で、実行中コンテキストを独自OSスタックに直接コピーせず、一度割り込み（または例外）通知専用スタックにコピーする。これは、独自OSにバグがあり独自OSスタックが正しく設定されていないことに起因する例外（ページ不在例外など）発生を避けるためである。

一方、Pico-Kernelライブラリ動作時のスタック切替えエミュレーションの動
作手順は以下の通りである。

1) 割り込み（または例外）が発生すると、非特権モードから特権モードに遷移し、アプリケーションスタックからPico-Kernelライブラリスタックへのスタック切替えがCPUにより行われるPico-Kernelライブラリは、Pico-Kernelライブラリスタックに Pico-Kernelライブラリが仮定する形式で実行中コンテキストを格納する。
2) Pico-Kernelライブラリから独自OSに割り込み（または例外）通知を行う。この通知はジャンプ命令（jmp命令）により行われ、使用スタックは変更されない。
3) 独自OS定義の割り込み（または例外）ベクタは、起動後すぐ、Pico-Kernelライブラリスタックか

ら独自 OS スタックに実行中コンテキストをコピーする。この際、格納形式の変換も行う。さらに、スタックポインタを更新し、使用スタックを独自 OS スタックに変更する。

4) 独自 OS 定義の割り込み（または例外）ベクタは、独自 OS スタックを用いて動作する。シグナルハンドラ起動時などには、独自 OS スタック内の実行中コンテキストを更新する。

5) 独自 OS 定義の割り込み（または例外）ベクタは、処理を完了したらハンドラリターン命令（iret 命令）を発行する。この命令発行により CPU の動作モードが特権モードから非特権モードに遷移し、CPU により使用スタックがアプリケーションスタックに変更される。そしてアプリケーションの実行が再開する。

割り込み（または例外）ベクタ内で実行中コンテキストを操作しないことがわかっている場合は、(3) の格納形式の変換やスタックポインタの更新処理を省略すれば（Pico-Kernel スタックで独自 OS 定義の割り込み／例外ベクタを動作させれば）、高速に割り込み（または例外）ベクタを起動することもできる。

図 5 と図 6 は、(4) の独自 OS 定義の割り込み／例外ベクタ動作時にはスタック状態が同一であることを示している。すなわち (4) の独自 OS 定義の割り込み／例外ベクタは同一コードを使用できる。

4. Pico-Kernel の実装概要

本章では、前章で示した特徴を持つ Pico-Kernel 及び Pico-Kernel ライブラリの実装の概要について述べる。まず、実装環境を説明した後、Pico-Kernel 群の提供機能の概要について述べる。

4.1. 実装環境

Pico-Kernel 群の実装環境として、ホストマシンとターゲットマシンは PentiumIII[5]搭載の PC/AT 互換機を使用した。両マシンは、IEEE1394 を用いて接続している。ホストマシン上には、FreeBSD 4.8-R 上で動作する gdb5.3 を搭載した。gdb5.3 のリモートデバッグ機能に一部改変を加え、IEEE1394 を介したリモートデバッグを可能にした。ターゲットマシン上には、Pico-Kernel 以外に、独自 OS が動作する。現在稼働している独自 OS は、HiTactix、TOPPERS/JSP （μITRON 仕様 OS）である。BSD/OS に関しても開発中で、現在α版の開発が完了している。

4.2. 提供機能

Pico-Kernel 群のモジュール構成を図 7 に示す。Pico-Kernel 群は、独自 OS とリンクしないで動作させる Pico-Kernel 群本体と、独自 OS とリンクして動作させる Pico-Kernel 群接続ライブラリからなる。

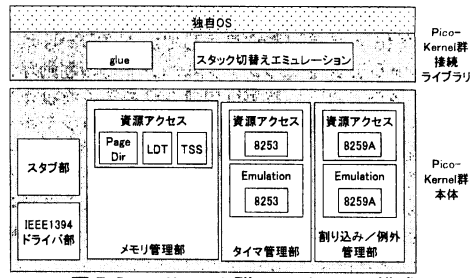

図 7 Pico-Kernel 群のモジュール構成

Pico-Kernel 群本体には、IEEE1394 ドライバ部、スタブ部、メモリ管理部、タイマ管理部、割り込み／例外管理部が存在する。現在の実装では、コンパイルオプションの指定を変えることで、同一のソースツリーから、Pico-Kernel 本体と Pico-Kernel ライブラリ本体の 2 種類のオブジェクトを生成可能にしている。Pico-Kernel 群本体は、物理メモリ領域、I/O メモリ領域、I/O ポートの保護設定を初期化時に行う。独自 OS からこれらのハードウェア資源をアクセスする際には Pico-Kernel を介する必要はない。しかし、1 段目のページテーブル（ページディレクトリ、CR3 レジスタ）、セグメントテーブル（LDT、LDTR レジスタ）、コンテキスト格納領域（TSS、TR レジスタ）に関しては資源アクセスモジュールを提供し、独自 OS はこのモジュールに対して要求を発行することで、ハードウェア資源にアクセスする。さらに、割り込み制御ビット、割り込みコントローラ（8259A チップ）、タイマ（8253 チップ）に関しては資源アクセスモジュールの他にエミュレーションモジュールも提供している（但し、Pico-Kernel ライブラリは割り込み制御ビット、割り込みコントローラのエミュレーションモジュールを使用しない）。

Pico-Kernel 群接続ライブラリには、Pico-Kernel に対するトラップ要求発行（または Pico-Kernel ライブラリのエントリポイントへのジャンプ）を行う glue モジュールと、3.2 節で示した動作を行うスタック切替えエミュレーションモジュールからなる。glue モジュールには、Pico-Kernel 対応版と Pico-Kernel ライブラリ対応版の 2 モジュールが存在する。また、スタック切替えエミュレーションモジュールは、Pico-kernel 群上で動作する独自 OS ごとに異なるモジュールを使用する。スタック切替えエミュレーションモジュールは、独自 OS を Pico-Kernel 群上で動作させる際に新規に実装する必要がある。

5 既存 OS への適用事例

本章では、HiTactix、TOPPERS/JSP、BSD/OS を Pico-Kernel 群の上で動作させるために行った改変内容の概要と、必要となった改変量の大きさについて述べる。

上記 3 つの既存 OS を Pico-Kernel 群上で動作させるために必要となった主な改変項目を以下に列挙する。

1) ブート処理の改変

2) 割り込みコントローラドライバの削除、割り込みコントローラエミュレーションドライバの新規開発、割り込み制御ビットアクセスインタフェース

[5] PentiumIII は米国 Intel 社の登録商標です。

の変更
3) タイマドライバの削除、タイマエミュレーションドライバの新規開発
4) セグメントテーブルアクセスインタフェースの変更
5) ページテーブルアクセスインタフェースの変更
6) コンテキスト格納領域アクセスインタフェースの変更
7) スタック切替えエミュレーションモジュールの新規実装

(1)は、Pico-Kernel 上で動作する際には独自 OS から一切アクセスする必要のないハードウェア資源(GDT など) の初期化処理の省略や、割り込みコントローラエミュレーションドライバやタイマエミュレーションドライバの初期化処理の追加からなる。(2)～(6)は、独自OSがPico-Kernel群を介してハードウェア資源アクセスを行うように、アクセスインタフェースを変更している。

表 2 に、HiTactix、TOPPERS/JSP、BSD/OS のそれぞれについて、上記(1)～(7)の改変項目ごとに、削除及び新規に追加したソースコードの行数を表している。また、上記以外の改変項目の改変量の合計についても併せて示している。

表 2 改変量一覧

No	HiTactix		TOPPERS/JSP		BSD/OS	
	削除	新規	削除	新規	削除	新規
(1)	13	19	54	2	22	8
(2)	46	89	49	19	180	338
(3)	0	121	1	1	99	50
(4)	0	0	0	0	9	18
(5)	0	0	0	0	128	227
(6)	59	25	0	0	1	0
(7)	0	64	0	64	47	472
others	1	1	1	0	189	159
Total	119	319	105	86	675	1332
	438		191		1997	

HiTactix、TOPPERS/JSP は仮想記憶を使用していないため、(4) (5) の改変は不要であった。また、TOPPERS/JSP と BSD/OS は TSS を使用していないため、(6) の改変はほとんど不要であった。

表から明らかな通り、必要な改変量 2K 行以下に抑えられており、既存 OS を Pico-Kernel 群上で動作させる際に大きな改変は必要ない。また、(1)～(7)以外に、Pico-Kernel 上で動作させる際の機能制限(GDT の使用、hlt 命令の使用、コプロセッサのコンテキスト切替え遅延機能が使用が不可能になること) に起因する改変が必要になった。

6 性能評価

本章では、提案した OS デバッグ方式の開発環境上で動作する独自 OS の I/O 性能の評価結果について示し、Pico-Kernel を用いずに独自 OS を動作させた時と比しても遜色のない I/O 性能を達成できることを明らかにする。

本 I/O 性能評価に用いた実験環境を図 8 に示す。本実験では、PC/AT 互換機 (PentiumIII 600MHz 搭載) 上で、単独で動作する HiTactix、Pico-Kernel 上で動作する HiTactix、Pico-Kernel ライブラリ上で動作する HiTactix を動作させる。そして、上記の各 HiTactix 上で、ギガビット Ethernet の MTU サイズ (1500 バイト) に相当する UDP パケットを指定した送信レートで送信するアプリケーションを動作させた。そして、指定する送信レートを変動させた場合における PC/AT 互換機の CPU 負荷の変動を測定した。上記 3 つの HiTactix においてこの測定を行い、I/O 性能の比較をした。

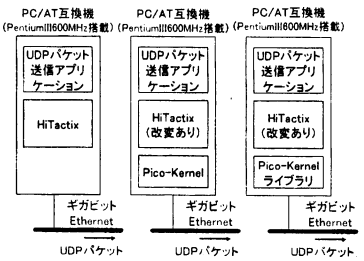

図 8 実験環境

測定結果を図 9 に示す。グラフの横軸は PC/AT 互換機から送信されている UDP パケットの送信レートを、縦軸にその時の PC/AT 互換機の CPU 負荷を示している。

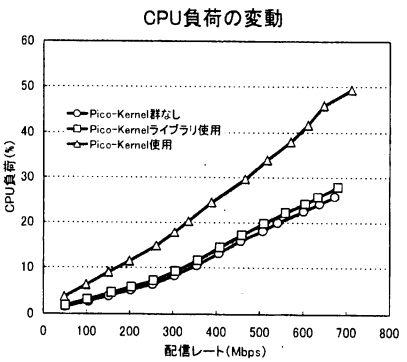

図 9 I/O 性能評価結果

この結果から以下が明らかになった。まず、Pico-Kernel 上で動作する HiTactix は、単独で動作する HiTactix と比して 52%の I/O 性能の劣化が発生する (同じ送信レートの際の CPU 負荷が、Pico-Kernel 上で動作する HiTactix は、単独で動作する HiTactix と比して、平均で 100 / (100-52) 倍になる)。しかし、Pico-Kernel ライブラリ上で動作する HiTactix は、単独で動作する HiTactix と比しても 9.5%しか性能低減が発生しない。

文献[8]によると、PC/AT 互換機の上で、本提案方式と同等に安定稼働可能な開発環境を構築しうるハードウェアシミュレータは、表 3 に示す性能で動作している。表で、「パフォーマンス」の項は、実ハードウェアとの動作性能比を示している。すなわち、この値から 1 を引いた値が「エミュレーションオーバヘッド」と

なり、I/O性能の劣化の要因になる。提案方式の場合、前述のI/O性能の評価結果から、1.1～2.1のパフォーマンスを得られると予測した。

表3 ハードウェアシミュレータとの比較

シミュレータ名	パフォーマンス	エミュレーションオーバヘッド
Shade	3～6	2～5
GSS	30	29
Mable	20～80	19～79
Specemu	40～150	39～149
SESP	18～50	17～49
提案方式	1.1～2.1	0.1～1.1

「エミュレーションオーバヘッド」の項を比較すればわかる通り、本提案方式は、最速のShadeと比較しても、Pico-Kernel使用時で1/2、Pico-Kernelライブラリ使用時では1/20にエミュレーションオーバヘッドを低減していることがわかる。

7 まとめ

本研究では、開発環境の安定稼働が保証できる、様々なOSやI/Oデバイスに大きな開発なく適用できる、デバッグ時にも高いI/O性能で動作させることができる、の3条件を充足するPC/AT互換機上の独自OS開発方式としてPico-Kernelを用いる方式を新規に提案した。

提案した開発方式は、従来のソフトウェアリモートデバッグ方式の改良である。従来と異なり、ターゲットマシン上でPico-Kernelを動作させる。Pico-Kernelはリモートデバッグに必要最小限の機能のみを提供する特権モード動作モジュールである。そして、開発すべき独自OSはPico-Kernel上で非特権モード動作する。これにより開発環境の安定稼働を保証した。さらに、Pico-Kernel上で動作するOSは、Pico-Kernelが管理するハードウェア資源にアクセスする時のみPico-Kernel提供インタフェースを呼び出す必要があるが、それ以外のハードウェア資源、特にI/Oデバイスには直接アクセスできるため、本開発の開発環境はOSやI/Oデバイスが変わっても大きな開発を必要としない。さらに、Pico-Kernelと同一インタフェースを提供しつつも、独自OSを特権モードで動作させるPico-Kernelライブラリも開発環境に組み込んで提供することで、デバッグ時のI/O性能の劣化を防いだ。

さらに、Pico-Kernelを用いた開発環境が既存OSにどれ程容易に適用可能であるか評価するために、HiTactix、μITRON仕様OS、BSD/OSをPico-Kernel上で動作させるように改変を加えた。その結果、どのOSも2K行以下の改変でPico-Kernel上で動作させることができ、本開発環境を用いたリモートデバッグが可能になることを確認できた。

最後に、Pico-Kernelを用いた開発環境上で動作する既存OS（HiTactix）が、Pico-Kernelを用いないで動作する時と比して、どの程度のI/O性能の劣化が発生するのかを評価した。評価の結果、I/O性能の劣化は9.5¥%程度に抑えられること、このI/O性能の劣化はハードウェアシミュレータを用いた場合と比べると1/2～1/20になることがわかった。

参考文献

[1] Novel Media Excelerator, http://www.novel.com/products/volera/media.html

[2] Kasenna Streaming Accelerator, http://www.kasenna.com/newkasenna.products/Kasenna-Streaming-Accelerator-IP-Datasheet.08.15.031.pdf

[3] M. Iwasaki, et. al., "Isochronous Scheduling and its Application to Traffic Control", 19th IEEE Real-Time System Symposium, Dec. 1998.

[4] 竹内理他, 「HiTactix-BSD連動システムを応用した大規模双方向ストリームサーバの設計と実装」, 情報処理学会論文誌 Vol. 43, No. 1, Jan. 2002.

[5] 竹内理他, 「外付けI/Oエンジン方式を用いたストリームサーバの実現」, 情報処理学会論文誌 Vol. 44, No. 7, Jul. 2003.

[6] HEC21/VS. http://www.hitachi-hec.co.jp/hec21vs/hec2vs01.html

[7] L. Albertsson, et. al., "Using Complete System Simulation for Temporal Debugging of General Purpose Operating Systems and Workloads", IEEE 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems 2000, Aug. 2000.

[8] 安田絹子他, 「オペレーティングシステム開発環境の設計と実装」, 情報処理学会研究報告「システムソフトウェアとオペレーティングシステム」, Vol. 072-011, Jun. 1996.

[9] 清水正明他, 「OSデバッグ環境の設計と実装」, 情報処理学会研究報告「システムソフトウェアとオペレーティングシステム」, Vol. 057-001, Oct. 1992.

[10] kgdb: linux kernel source level debugger, http://kgdb.sourceforge.net

[11] KDB(Built-in Kernel Debugger), http:// oss.sgi.com/products/kdb

[12] TOPPERSプロジェクト, http://www.toppers.jp

[13] BSDI Internet Server Edition, http://www.networks.macnica.co.jp/windriver/index.html