UNIX へ実時間性と柔軟性を付加した 組込み機器向け基盤ソフトウェアに関する研究

佐藤　喬

電気通信大学
大学院情報システム学研究科

博士（工学）の学位申請論文

2014 年 3 月

UNIX へ実時間性と柔軟性を付加した組込み機器向け基盤ソフトウェアに関する研究

博士論文審査委員会

主査	多田 好克	教授
委員	大森 匡	教授
委員	末廣 尚士	教授
委員	田野 俊一	教授
委員	古賀 久志	准教授
委員	小宮 常康	准教授

著作権所有者

佐藤 喬

2014年

Adding Real-Time Functionality and Flexibility to UNIX for Embedded System

Takashi Sato

Abstract

With improvement in the performance of existing computer hardware, embedded systems require to use several functions, such as network stacks, file systems, graphical user interfaces (GUI), and web interfaces. However, it is difficult for existing embedded operating systems (OS) to deal with such functions because embedded OSes do not possess rich software resources. Developers have to develop or buy such software resources, which results in an increase in the required time and monetary cost incurred during the development process.

Therefore, UNIX has been preferred for use as the embedded OS since it already possesses rich software resources, such as device drivers, network stacks, and application programs. Many of these software resources are open source and royalty free. Using UNIX as the embedded OS resolves the problem of increased development cost that mars other embedded OSes. However, a general purpose OS such as UNIX does not possess the real-time functionality and flexibility required of an embedded OS since a general purpose OS focuses mainly on the equality and efficiency of processing.

In this paper, we propose Unitron system and Kexec system. The Unitron system the kernel module of a general purpose OS with an additional real-time functionality. The Unitron system is a UNIX kernel module that is obtained from the μITRON kernel. The μITRON system is an embedded OS possessing real-time functionality. The Unitron system provides the real-time functionality to UNIX. The Kexec system the kernel module of a general purpose OS with an additional flexibility. The Kexec system allows user programs to run in kernel mode. The user programs in kernel mode can access kernel resources directly. These systems are loadable kernel module (LKM) and can be dynamically added and removed from the general purpose OS. By using these systems, UNIX becomes to be suitable for an embedded system.

UNIX へ実時間性と柔軟性を付加した組込み機器向け基盤ソフトウェアに関する研究

佐藤 喬

概要

組込みシステムは、ある特定の役割を持つ機器を制御する計算機システムである。現在、組込みシステムは、情報家電やロボット、携帯電話、自動車など様々な機器に搭載されており、世界中で広く利用されている。

組込み機器に搭載された組込みシステムは、デスクトップ PC（Personal Computer）のような汎用の計算機システムと同じように、CPU とメモリ、I/O（Input/Output）装置のハードウェア群とそれらを統括する基盤ソフトウェアから構成される。ただし、組込みシステムは、その用途が限定されているため、用途に必要十分なハードウェア資源と基盤ソフトウェアだけから構成される。そのため、既存の組込み OS は、性能の低い CPU と少量のメモリ上で動作するように単純な機能のみの構成になっている。更に低価格化のためにハードウェア資源が制限されるような用途の組込みシステムは、OS を使用せずに対象となる用途専用のソフトウェアのみで構成されることもある。

他方で組込み機器の高機能化が進み、汎用 OS(Operating System) である UNIX を組込み機器向けの基盤ソフトウェアとして利用する事例が増えてきた。UNIX を利用する利点は、既に動作している豊富なソフトウェア資産を流用することで開発コストを削減できる事である。

例えば、UNIX を使った情報家電の TV は、画面上のメニューを既存の GUI(Graphical User Interface) ライブラリを基に構築する事ができる。また、インターネットから番組情報を取得するような処理も、UNIX の持つドライバ、ネットワークスタック、アプリケーションといった豊富なソフトウェア資産を利用して実現可能である。一方、組込み OS は、UNIX に比べソフトウェア資産の蓄積が少ない。そのため、ソフトウェアをゼロから開発するか、商用のソフトウェアを別途購入する必要があり、開発にコストがかかってしまう。UNIX は、豊富なソフトウェア資産を持ち、多くの UNIX 実装はロイヤリティフリーかつオープンソースである。UNIX を組込み機器向けの基盤ソフトウェアとして利用すれば、これらの豊富なソフトウェア資産を利用、改良することで、既存の組込み OS が抱えていた開発コストの増加を抑えられる。

このように、UNIX は豊富なソフトウェア資産を提供する事で、組込み機器向けの基盤システムとして開発上の利点をもっている。しかし、UNIX は汎用計算機向けとして発展

してきたため、組込み機器を制御する上で以下に挙げる 4 つの不足点がある。

- 実時間性の不足
- 用途に特化する柔軟性の不足
- 省資源性の不足
- 信頼性の不足

これらの内、省資源性の不足に対しては μCLinux などのシステムが、信頼性に関しては SELinux などの研究があるが、実時間性の不足と用途に特化する柔軟性の不足に対しては有効な手段が見いだせていない。本研究では、この 2 つの不足点の解決を目標とする。まず、実時間性について説明する。実時間性は、ハードウェア割込みへの応答時間やソフトウェア処理時間をある時間内に収めることを保証する性質である。組込みの世界では、数 μ 秒オーダの実時間性を求められるが、汎用 OS では処理の効率や平等性を重視した結果、実時間性を持たないことが多い。次に柔軟性について説明する。組込みシステムは、使用される用途が限定されるため、ソフトウェアもその処理内容に特化する必要がある。μITRON のような組込み OS では、CPU は常にカーネルモードで動作し、ハードウェアを含めたカーネル内資源に低オーバヘッドにアクセスできる柔軟性を持つ。一方、汎用 OS は、多目的に使用されるため、多種多様なアプリケーションソフトウェアを動作させなければならない。そこで CPU は、それらのアプリケーションをユーザモードで動作させ、システムコールという制限された枠組みを通してカーネルモードへ遷移し、カーネル内資源へアクセスする必要がある。この方法ではカーネル内資源へのアクセス手段が制限されてしまい、用途に特化する柔軟性が不足する。汎用 OS を組込みシステムとして利用するには、カーネル内資源にアクセスできる柔軟性を確保し、用途に応じて特化可能にすべきである。

そこで、本研究では「実時間性の不足」と「用途に特化する柔軟性の不足」を解決するため、実時間性を提供する unitron システムと、用途に特化する柔軟性を提供する kexec システムを提案する。unitron システムは、組込み用 OS の μITRON を LKM(Loadable Kernel Module) 化することで、UNIX カーネル内部に取り込み実時間性を提供する。kexec システムは、UNIX アプリケーションをそのままカーネルモードで動作させ、カーネル内資源を直接操作可能とする。これにより、用途に特化する柔軟性を提供する。unitron システムと kexec システムにより、豊富なソフトウェア資産を持つという利点を生かしつつ、UNIX をより組込み機器向けの基盤ソフトウェアとして活用できるようになる。

目次

第 1 章　背景　1
1.1　組込みシステム　1
1.2　構成　4

第 2 章　提案　5
2.1　unitron システムの概要　5
2.2　kexec システムの概要　5

第 3 章　unitron システム　9
3.1　背景　9
3.2　設計方針　10
3.3　実装　12
3.4　評価　31
3.5　関連研究　32
3.6　まとめ　33

第 4 章　kexec システム　35
4.1　背景　35
4.2　目的　36
4.3　既存手法の問題　36
4.4　本研究の提案　41
4.5　システム概要　42
4.6　kexec システムを実現するにあたっての問題　44
4.7　設計方針　47
4.8　実装　48
4.9　カーネル内シンボルの解決　60
4.10　スタックの切り替え　61

4.11 評価 62
4.12 関連研究 75
4.13 まとめ 76

第 5 章 まとめ 79

参考文献 81

図目次

2.1 unitron システムの概要 6
2.2 kexec システムの概要 6

3.1 unitron のメモリ配置 13
3.2 libunitron.a の構成法 14
3.3 LKM オブジェクト unitron.o 15
3.4 TOPPERS の実行開始 18
3.5 割込み配送部 20

4.1 保護機構の概略 37
4.2 システムコール処理の流れ 38
4.3 Web サーバのファイル転送 39
4.4 アプリケーションのカーネルモード実行 43
4.5 カーネル内資源の直接操作 44
4.6 カーネルモード実行時のメモリ配置 45
4.7 kexec システムの構成 49
4.8 カーネルモードへの移行方法 51
4.9 関数型システムコールの処理の流れ 56
4.10 通常のシステムコール呼び出し 59
4.11 インターポジショニングが起きた場合 60
4.12 リンカスクリプトの内容 61
4.13 `getpid()` システムコールの処理時間 68
4.14 `read()` と `write()` システムコールを用いるファイルコピー 70
4.15 バッファキャッシュ間の直接コピー 70
4.16 ファイルコピー速度の比較 75

表目次

3.1 周期ハンドラの起動間隔（単位：μs） 31
3.2 ソース行数 32

4.1 システムの比較 44
4.2 評価に用いた計算機環境 62
4.3 システムコールの処理時間の比較 69

リスト目次

3.1 LKM オブジェクトのロードと確認 15
3.2 unitron スタートプログラム 16
3.3 unitron スタート用割込みベクタ 17
3.4 TOPPERS スタートアップルーチン 19
3.5 割込み配送部コード 19
3.6 割込みベクタテーブルの初期化 21
3.7 割込みベクタテーブルの後処理 23
3.8 OS コンテクスト保存用構造体 24
3.9 コンテクストスイッチ部 24
3.10 TOPPERS のタスクスイッチ部 26
3.11 タイマ処理部 27
3.12 OS 間スケジューラ 30
4.1 `kexec_main()` 関数のソース 52
4.2 カーネルモード実行部のソース 53
4.3 エントリポイントの変更法 55
4.4 自動生成された置換用ソース (`read()` システムコール) 57
4.5 一部変更が必要な置換用ソース (`pipe()` システムコール) 58
4.6 スタックのアドレスを取得、設定するマクロ 62
4.7 `print_pid.c` の内容 63
4.8 コンパイル方法（通常版） 64
4.9 実行結果（通常版） 64
4.10 コンパイル方法（カーネルモード実行版） 64
4.11 カーネルモジュールの組み込み 64
4.12 実行結果（カーネルモード実行版） 65
4.13 `kern_print_pid.c` の内容 65
4.14 コンパイル方法（カーネル内資源利用版） 66

4.15 実行結果（カーネル内資源利用版） 66
4.16 バッファキャッシュ間でデータコピーをする関数 71

第 1 章

背景

本研究は、汎用 OS である UNIX を組込み向け基盤ソフトウェアに適するよう改良することを目的とする。本章では、汎用 OS（Operating System）と組込み向け基盤ソフトウェアの関係について述べる。組込み機器が高性能化した結果、これまで汎用計算機向けとして使われてきた OS を組込み向け基盤ソフトウェアとして使用する例が増えている。なぜなら、汎用 OS には豊富なソフトウェア資産がそろっており、それを活用することができれば、開発のコストを抑えることができるためである。

1.1　組込みシステム

組込みシステムとは、ある役割を持つ機器の制御を行う計算機システムである。現在、組込みシステムは、情報家電やロボット、携帯電話、自動車など様々な機器に搭載されており、世界中で広く利用されている。

組込み機器に搭載された組込みシステムは、デスクトップ PC（Personal Computer）のような汎用の計算機システムと同じように、CPU とメモリ、I/O（Input/Output）装置のハードウェア群とそれらを統括する基盤ソフトウェアから構成される。ただし、組込みシステムは、その用途が限定されているため、用途に必要十分なハードウェア資源と基盤ソフトウェアから構成されている。既存の組込み OS は、性能の低い CPU と少量のメモリ上で動作するように単純な機能のみで構成になっている。更にハードウェア資源が制限されるような用途の組込みシステムは、OS を使用せずに対象となる用途専用のソフトウェアのみで構成されることもある。

しかし、情報家電をはじめとした組込み機器の用途の拡大により、組込みシステムは、GUI（Graphical User Interface）やネットワーク、ストレージ、Web インタフェースの機能が求められるようになってきた。また、CPU やメモリなどのハードウェア資源の性能向上と低価格化により、高性能なハードウェア資源を搭載する組込み機器が可能になっ

た。このような状況から、組込みシステムの基盤ソフトウェアに UNIX の一種である Linux や NetBSD のような汎用 OS を利用する例が増えている。例えば、情報家電 TV である SHARP 社の AQUOS は、Linux を用いネットワーク接続を行い、GUI 描画に DirectFB と呼ばれるグラフィカルライブラリを使用している [1]。他にも、ネットワーク接続機能を持った二足歩行ロボット [2]、ネットワークルータ [3]、Web インタフェースを持つネットワークカメラ [4] などに NetBSD が利用された例もある。

組込み機器向けの基盤ソフトウェアに UNIX を利用する理由は、μITRON のような既存の組込み OS に、デバイスドライバやネットワークスタック、アプリケーションプログラムなどのソフトウェア資産が不足しているためである。既存の組込み OS で、上記に挙げたような用途に対応するためには、自前で必要なソフトウェアを開発するか、既に存在するサードパーティー製のソフトウェアのロイヤリティを購入する必要がある。これは開発にかかる時間的、金銭的コストの増加を意味する。一方、UNIX は、豊富なソフトウェア資産を持ち、多くの UNIX 実装はロイヤリティフリーかつオープンソースである。UNIX を組込み機器向けの基盤ソフトウェアとして利用すれば、これらの豊富なソフトウェア資産を利用、改良することで、既存の組込み OS が抱えていた開発コストの増加を抑えられる。

しかし、UNIX のような汎用 OS は組込み向け基盤ソフトウェアとして以下の 4 つの問題点がある。

- 実時間性の不足
- 用途特化の柔軟性の不足
- 省資源性の不足
- 信頼性（堅牢性）の不足

これらの問題点の内、「省資源性の不足」と「信頼性（堅牢性）の不足」については改良が進められている。しかし、「実時間性の不足」と「用途特化の柔軟性の不足」の問題は残ったままである。そこで、本研究では「実時間性の不足」と「用途特化の柔軟性の不足」に焦点をあてる。以下に、4 つの問題点について説明する。

1.1.1　実時間性の不足

実時間性は、ハードウェア割込みへの応答時間やソフトウェア処理時間をある時間内に収めることを保証する性質である。組込みの世界では、数 μs オーダの実時間性を求められるが、汎用 OS では処理の効率や平等性を重視した結果、実時間性を持たないことが多い。

1.1.2　用途特化の柔軟性の不足

汎用 OS は、多目的に使用されるため、多種多様なアプリケーションソフトウェアを動作させなければならない。CPU は、それらのアプリケーションをユーザモードで動作させ、システムコールという制限された枠組みを通してカーネルモードへ遷移し、カーネル内資源へアクセスする。

一方、組込みシステムは、用途が限定されるため、ソフトウェアもその処理内容に特化する必要がある。μITRON のような組込み OS では、CPU は常にカーネルモードで動作し、ハードウェアを含めたカーネル内資源に低オーバヘッドにアクセスできる柔軟性を持つ。

汎用 OS を組込みシステムとして利用するには、この柔軟性を確保し、用途に応じて特化可能にすべきである。

1.1.3　省資源性の不足

汎用機器に比べ用途が限定される組込み機器は、機器性能の上限が明確であることが多い。機器の作成コストを抑えるためには、その性能上限に近い必要十分なハードウェア構成となる。このようなハードウェア構成においては、動作するソフトウェアは限られたハードウェア資源の上で動作する必要がある。

UNIX では、省資源性を実現する技術として以下のような試みが行われている。busybox[5] は、UNIX の基本的なアプリケーション群を一つのファイルにまとめ、別々のファイルで構成した場合に生じていた冗長なディスク消費量を抑えた。また、300 種類のアプリケーションを一つにまとめており、その機能も必要な処理に絞って小型化を図っている。μClibc[6] は、アプリケーションの基本ライブラリの libc ライブラリを、組込み機器向けに小型にしたものである。busybox と合わせて多くの組込み機器向け UNIX で使用されている。linux-tiny プロジェクト [7] は、Linux カーネルの小型化を目指すプロジェクトである。

このように、組込み機器のハードウェア資源が制約されている環境に適応した UNIX を構築する技術が存在している。

1.1.4　信頼性（堅牢性）の不足

組込みシステムがネットワークに接続されることによって、セキュリティ上の信頼性の問題が生じる。また、組込みシステムは、長時間運用されること、不意に電源が落とされることなどに対応できる堅牢性を持たなければならない。

SELinux[8] は Linux カーネル用のセキュリティ拡張機能で、セキュリティ上の防御と監視機能を持つ。万が一、管理者権限を奪われたとしても、管理者のアクセス権限を分割、制限することができ、信頼性を確保している。

JFFS2[9] は、組込み機器のストレージとして利用されることの多い Flash 用のファイルシステムである。JFFS2 はジャーナリングファイルシステムを持っており、不意に電源が落とされたとしても、ファイルシステムの一貫性を保つことが出来る。また、書き込み回数の制限される Flash ストレージに対し、書き込みの局所化を避け、Flash の長時間運用に貢献している。

1.2　構成

本論文の構成を本節で述べる。第 1 章では、背景として組込み基盤システムの現状と本研究で対象とする問題点について述べた。第 2 章では、提案として第 1 章で挙げた問題点に対する提案として unitron システムと kexec システムの概要を述べる。第 3 章では unitron システムを、第 4 章では unitron システムを説明する。最後に第 5 章でまとめを述べる。

第 2 章

提案

本章では、第 1 章で述べた問題の中で対策が欠けている「実時間性の不足」と「用途特化の柔軟性の不足」の二つの問題に対して、unitron システムと kexec システムの二つを提案する。unitron システムは、実時間処理 OS を汎用 OS のモジュールとして動作させることで、実時間性の不足を補う。一方、kexec システムは、アプリケーションをカーネルモード実行することで、用途特化の柔軟性の不足を補う。これら二つのシステムを使用することで、汎用 OS を組込み向け基盤ソフトウェアに近づけることを目指す。

2.1 unitron システムの概要

unitron システムは、実時間性を持つ組込み OS の 1 つの μITRON カーネルをカーネルモジュール化し、UNIX カーネルに取り込むシステムである [10, 11]。μITRON のカーネルモジュールは、UNIX に実時間性を付加する。概要を図 2.1 に示す。

開発者は、unitron システムでの実時間処理を μITRON のタスクとして記述し、実時間性が必要ない処理を UNIX のプロセスとして記述する。このように、実時間処理を実現しつつ、UNIX が持つ豊富なソフトウェア資産を流用できることが利点である。また、μITRON の機能を LKM（Loadable Kernel Module）を利用し UNIX に取り込んでいるため、動的に機能の取り付け、取り外しができる。

2.2 kexec システムの概要

kexec システムは、UNIX アプリケーションをカーネルモードで動作させるシステムである [12]。概要を図 2.2 に示す。kexec システムにより実行される UNIX アプリケーションをカーネルウェアと呼ぶ。カーネルモードで動作するカーネルウェアは、UNIX カーネル内資源を直接操作する柔軟性を持つ。カーネルウェアは UNIX アプリケーションを流

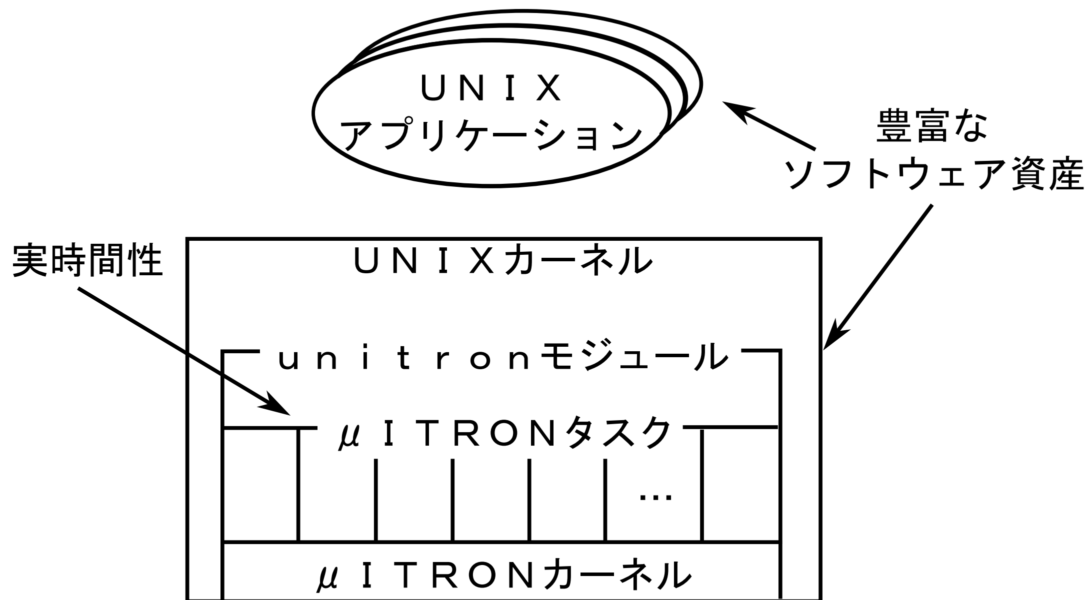

図 2.1 unitron システムの概要 : UNIX カーネル内にカーネルモジュールとして μITRON を取り込み、実時間性を提供する。

用して作成できるため、初めからカーネルモジュールを作成する場合に比べ開発のコストを抑えられる。

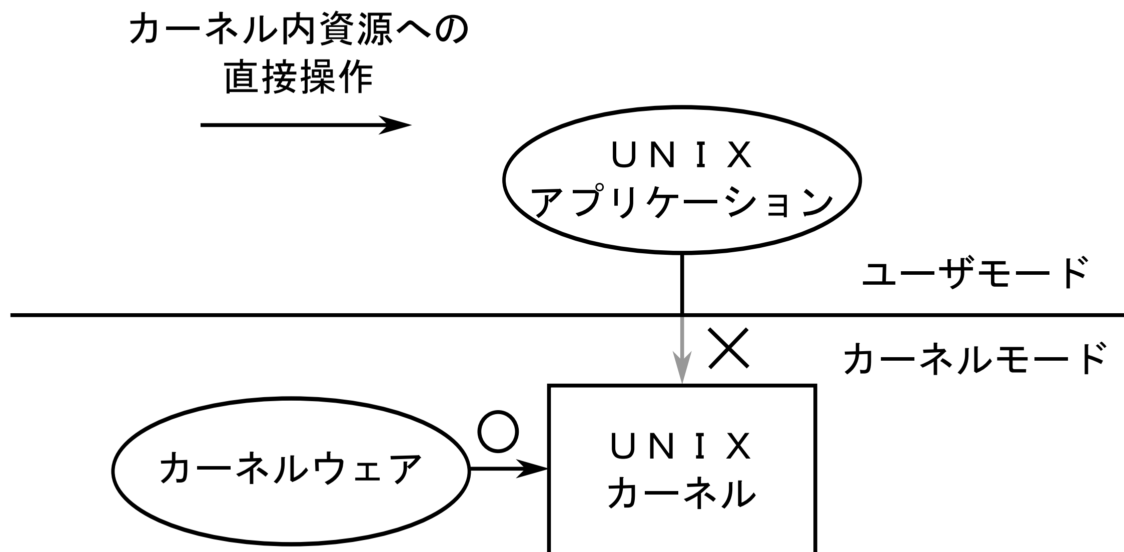

図 2.2 kexec システムの概要 : カーネルモードで動作するカーネルウェアはカーネル内資源への直接操作が可能になり、用途に特化する柔軟性を確保する。

組込み機器は用途が限定されるため、その用途に特化したシステムが求められる。kexec システムを使用することで、用途に応じたアプリケーションを流用しつつ、そのソースへ

カーネル内資源を直接操作する変更を加え、用途に特化できる。

第 3 章

unitron システム

本章では、unitron システムについて説明する。unitron システムは、実時間 OS をカーネルモジュール化し、汎用 OS に取り込む枠組みである。これにより、実時間性を持たない汎用 OS に実時間性を持たせることが可能になる。

3.1 背景

情報家電をはじめとした組込み機器の用途の拡大により、組込みシステムは、ネットワークやファイルシステム、GUI、Web インタフェースなどの機能が求められるようになってきた。また、CPU やメモリなどのハードウェア資源の性能向上と低価格化により、高性能なハードウェア資源を搭載する組込み機器が可能になった。このような状況から、組込みシステムに UNIX のような汎用 OS を利用する例が増えている。

既存の組込み OS で、上記のような機能を提供するためには、自前で必要なソフトウェアを開発するか、既に存在するサードパーティー製のソフトウェアのロイヤリティを購入する必要がある。これは開発にかかる時間的、金銭的コストの増加を意味する。一方、UNIX は、ネットワークやファイルシステムを提供するカーネル機能、GUI ライブラリ、Web サーバのようなアプリケーションプログラムなどの豊富なソフトウェア資産を既に持っている。これらのソフトウェア資産の多くは、ロイヤリティフリーかつオープンソースであり、UNIX を組込み OS として利用すれば、既存の組込み OS が抱えていた開発コストの増加を抑えられる。

しかし、UNIX のような汎用 OS は組込み OS として必要な実時間性が不足している。実時間性は、ハードウェア割込みへの応答時間やソフトウェア処理時間に対し、ある時間制約を保証することである。汎用 OS では処理の効率や平等性を重視した結果、実時間性を持たないことが多い。

豊富なソフトウェア資産と実時間性を持つ組込み OS に対する要求から、実時間 OS を

汎用 OS のモジュールとして動作させせることで、実時間性の不足を補う unitron システムを提案し、実装した。

3.2　設計方針

unitron システムの設計方針は以下の 3 つである。

- 組込みシステムに適した OS の選択
- μITRON の実時間性を確保
- UNIX、μITRON への変更を極力抑える

以下、各方針について説明する。

3.2.1　組込みシステムに適した OS の選択

組込みシステムでは、その用途に応じて多種多様な CPU アーキテクチャが使用されている。そのため、ベースとして使用する OS も多種多様な CPU アーキテクチャに対応していることが望ましい。

そこで、unitron システムで使用する UNIX と μITRON の実装は、それぞれ NetBSD と TOPPERS を選択した。選択の理由は、これらの実装が豊富な CPU アーキテクチャに対応しており、組込みシステムでの利用実績があるためである。両実装の特徴として、CPU アーキテクチャ依存部と非依存部が明確に切り分けられており、新たな CPU アーキテクチャへの移植が容易であるという利点もある。

3.2.2　μITRON の実時間性確保

NetBSD へ TOPPERS を組み込むにあたり、TOPPERS の実時間性を損なわないことは重要な課題である。この課題を達成するためには、以下 3 つの問題を解決する必要がある。

- TOPPERS タスクの優先的な実行
- TOPPERS に対する低遅延の割込み配送
- NetBSD による割込み禁止処理への対応

1 つめの問題は、TOPPERS タスクと NetBSD との優先度の問題である。TOPPERS タスクの動作の方が NetBSD よりも実時間性が求められるため、TOPPERS タスクは NetBSD よりも高い優先度で動作する必要がある。

そこで、NetBSD と TOPPERS 間のコンテクストスイッチの起点となる割込み復帰毎に、実行可の TOPPERS タスクがあるかチェックし、そのようなタスクが無い場合に NetBSD へコンテクストスイッチする。これにより、NetBSD に対し TOPPERS タスクを優先的に実行できる。また、NetBSD カーネルはプリエンプティブカーネルであるため、NetBSD が実行中であっても割込みが発生すれば、現在の処理を休止し、TOPPERS 側へ処理を渡す事が可能である。

2 つめの問題は、割込み処理に関するものである。タイマや TOPPERS が制御する周辺機器からの割込みが発生した場合、できる限り低遅延で TOPPERS へ処理を渡さなければならない。そのため、NetBSD の割込み処理を介して TOPPERS へ割込み配送することは避けるべきである。

そこで、NetBSD と TOPPERS へ割込み情報を振り分ける割込み配送部を実装した。割込み配送部は、CPU が呼び出す割込み処理が記述された割込みベクタを置換し、割込み処理をすべき NetBSD か TOPPERS へ直接割込みを配送する。このようにすることで、NetBSD を介することなく、TOPPERS では低遅延の割込み処理ができる。また、タイマのような両者が必要とする割込みは、TOPPERS へ先に配送し、TOPPERS の割込み終了後に NetBSD へ配送する。

3 つめの問題は、NetBSD による割込み禁止処理への対応についてである。NetBSD 側で割込みを禁止されてしまうと、割込み配送部へ割込み情報が届かなくなってしまう。もしこの状態の時に、TOPPERS 側でタイムクリティカルな処理を行いたいとしても、割込みがブロックされているため、処理を切り替えることができない。このことは、実時間性を損ねる原因となる。

そこで、NetBSD 内部で呼び出される割込み禁止と許可を置換する方法を用いる。この方法は、保田らが TOPPERS 上で Linux をタスク化し実行した際にも使用されている [13]。これは、Linux の割込み禁止の `cli` と許可の `sti` を、タスク例外処理禁止状態にする `dis_tex` と許可状態にする `ena_tex` へ変更し、CPU への割り込み禁止処理をタスクへの割り込み禁止という方法で実現している。

同様に unitron でも NetBSD 中の割込み禁止と許可を置換し、NetBSD が割込みを禁止しても、TOPPERS 側へは割込みが配送されるようにする。置換内容は、割込みを実際には禁止せず、割込み配送部まで割込みを到達させるというものである。そして、配送部からは NetBSD へ割込みを通知しないようにする。こうすることで、TOPPERS へは割込みの通知が可能になるため、TOPPERS の実時間性を損ねてしまうことを避けられる。

なお、割込み禁止処理については実装が完了していないため、実装方針を述べるに留める。

3.2.3 UNIX と μITRON への変更

UNIX、μITRON への変更を極力抑えることについて説明する。使用する UNIX と μITRON の実装である NetBSD と TOPPERS はオープンソースとして開発が続けられている。そのため、それぞれのソースを直接編集し大幅な変更を加えると、新しいバージョンへ追随することが難しくなる。

そこで、unitron システムでは NetBSD と TOPPERS について次のように対応する。NetBSD に対する変更は LKM を利用する。LKM を使うことで、NetBSD のカーネルソースへは変更を加える必要が無くなり、新しいバージョンへの追随が容易になる。ただし、作成した LKM プログラムに新しいバージョンへの依存部が存在した場合は修正が必要である。

TOPPERS に対する変更は直接ソース変更を行う。そのため、NetBSD の LKM 側で吸収できる変更はそちらで行い、TOPPERS に対する変更の量を抑える。また、TOPPERS に対する変更は、NetBSD の LKM を一つの計算機アーキテクチャとして考え、アーキテクチャ依存部として記述する。このことにより、TOPPERS のアーキテクチャ非依存部への変更を無くし、修正箇所を減らす。

3.3 実装

3.3.1 実装に使用したプラットフォーム

unitron の実装に使用した OS は、UNIX として NetBSD を、μITRON として TOPPERS を選択した。こうした理由は、これらの OS が豊富な CPU アーキテクチャに対応しているためである。設計で述べたように対応 CPU アーキテクチャの豊富さから、それぞれのバージョンは、NetBSD-5.0.1、TOPPERS/JSP-1.4.3 である。JSP は Just Standard Profile の略称であり、μITRON 4.0 仕様に準拠した実装となる。

今回の実装では、対象とする CPU アーキテクチャを Intel IA-32 アーキテクチャを選択した。選択の理由は、Intel IA-32 アーキテクチャに対する開発技術・知識が、他の CPU アーキテクチャのそれに比べ我々の研究室に多く蓄積されているためである。組込み機器に多く使用されている、ARM や PowerPC、MIPS などの CPU アーキテクチャへの対応は今後の課題とする。

3.3.2　メモリ配置

図 3.1 に示すように、NetBSD カーネル領域内の LKM 領域のメモリに unitron は配置される。そのため、unitron のシンボル解決は、LKM ロード時に動的に行われる。両 OS は同一のメモリ空間に配置されるため、互いの通信は共有メモリを介して行うことができる。

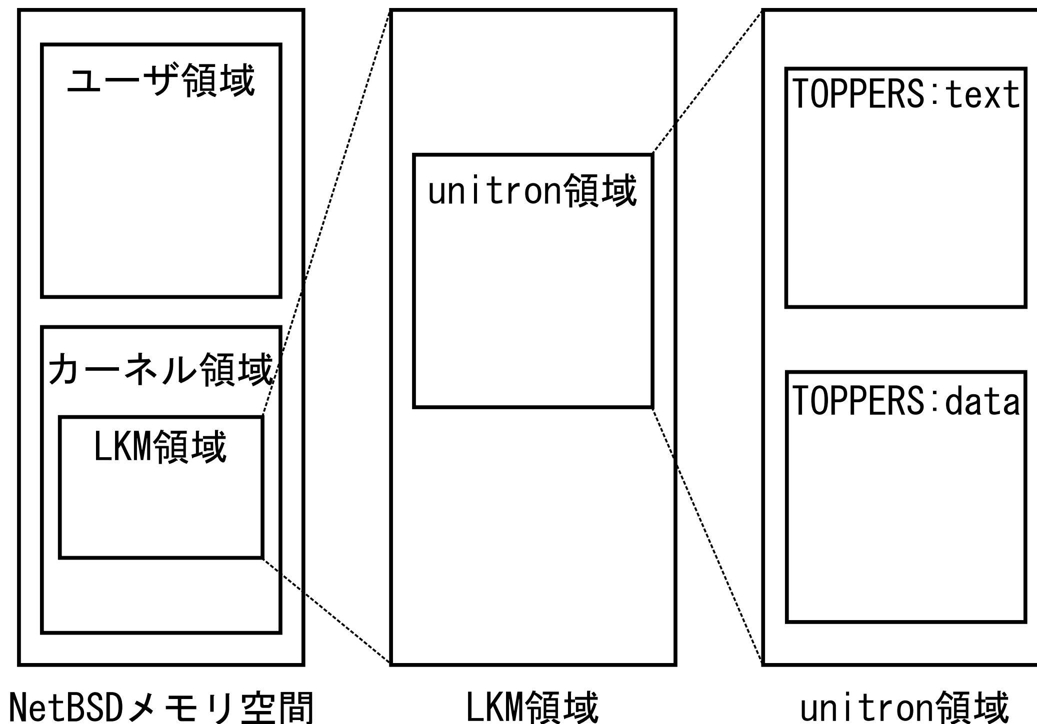

図 3.1　unitron のメモリ配置：unitron は、LKM により NetBSD 内のメモリ空間へ動的に配置される。つまり、TOPPERS のコードとデータも NetBSD のメモリ空間へ配置される。

元となる TOPPERS は、ハードウェアインタフェースの関係から、一部のシンボルのアドレスを静的に持っていた。例えば、実行開始位置を示すエントリポイントに対応するシンボルなどである。これは、ハードウェアがメモリへアクセスする場合、アドレスを直接指定しなければならないためである。

このような静的なアドレスを持つシンボルは、CPU アーキテクチャ依存部に限られ、非依存部には影響が無い。調査の結果これらのシンボルは、TOPPERS 実行イメージのエントリポイントとメモリ配置場所指定に使われるのみで、LKM として使用する場合には、動的に解決しても問題とはならなかった。

3.3.3 LKM オブジェクトの構成

LKM オブジェクトは、LKM 機構を使ってカーネルにロードするオブジェクトファイルを指す。この節では、unitron の LKM オブジェクトの構成について述べる。

unitron では、TOPPERS 側の機能を図 3.2 に示すように、ライブラリ化した。ライブラリ化した理由は、TOPPERS 側の独立性を保つためである。独立性を保つことで、TOPPERS のバージョンアップへ追随することが容易になる。

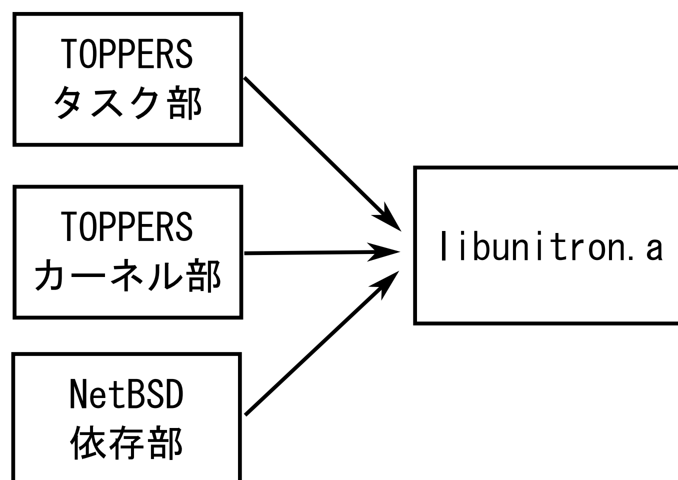

図 3.2 libunitron.a の構成法 : TOPPERS 側の機能をライブラリ化し提供する。

ライブラリには、TOPPERS タスク部とカーネル部、NetBSD 依存部が含まれる。TOPPERS タスク部は、μITRON のタスクから構成されている。unitron システムを利用する場合、組込みシステムの実時間部開発者はこのタスク部を記述することになる。

TOPPERS カーネル部には、CPU アーキテクチャ非依存のカーネルコードが含まれる。unitron の実装では、TOPPERS の CPU アーキテクチャ非依存部/依存部の関係を壊さずに実装したため、この CPU アーキテクチャ非依存のカーネルコードは一切の変更を加えていない。

NetBSD 依存部は、本来ならば CPU アーキテクチャ依存部となる部分だが、本実装では、NetBSD を一つの CPU アーキテクチャと見立てて実装した。ただし、この部分には、TOPPERS タスクのコンテクストスイッチ部も含まれており、現実装では Intel IA-32 のアセンブリコードも含まれてしまっている。CPU アーキテクチャに依存するコードを除去し、純粋に NetBSD のみに依存させれば、このライブラリの CPU アーキテクチャ間の移植が容易になる。

図 3.3 のように、TOPPERS 側の機能をまとめたライブラリ libunitron.a と、NetBSD

側の機能を合わせて LKM オブジェクト unitron.o とした。NetBSD 側の機能には実行開始部、割込み配送部、OS 間コンテクストスイッチ部が含まれる。上記 3 つの NetBSD 側の機能の詳細は次節以降で順次説明する。

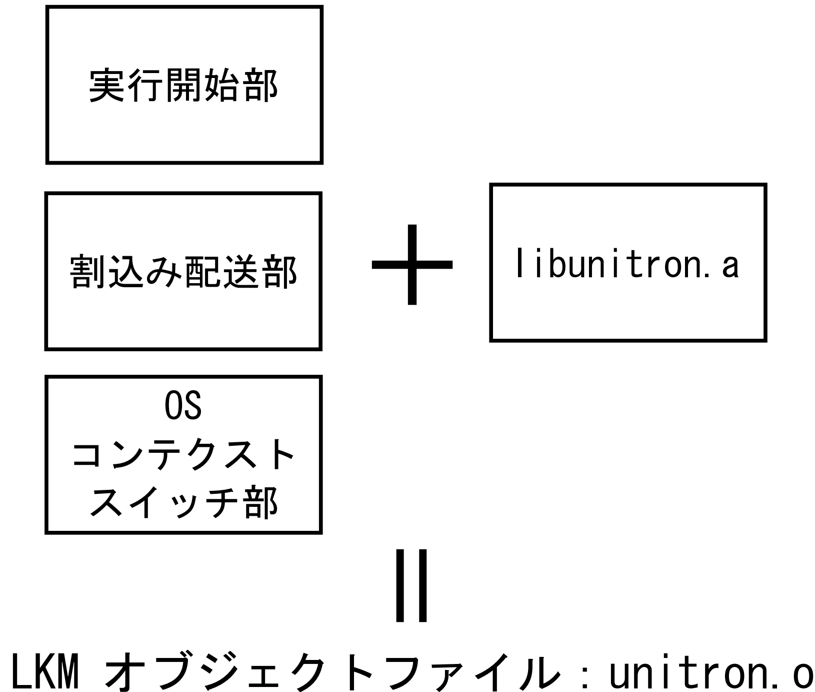

図 3.3　LKM オブジェクト unitron.o : libunitron.a と NetBSD 側の機能を結合することで作成する。

この LKM オブジェクトファイルを `modload(8)` コマンドでカーネルにロードする。ロードされた状態は、`modstat(8)` コマンドで確認できる（リスト 3.1 参照）。[htbp]

```
1 % sudo modload unitron.o
2 Module loaded as ID 0
3 % modstat
4 Type    Id   Offset Loadaddr Size Info  Rev Module Name
5 DEV      0  -1/194 c4730000 0170 c4736f40 2 unitron
```

リスト 3.1　LKM オブジェクトのロードと確認

実行開始部

LKM としてロードした unitron の TOPPERS 部の実行は、スタートプログラムからの専用ソフトウェア割込みの発生をトリガに行われる。実行開始を LKM のロードと同時

に行う方法もあるが、ロード時の初期化処理と実行開始を分けた方がプログラムコードの各処理内容が明確になる。また、ソフトウェア割込みをトリガとする事で、割込み突入時のコンテクスト保存を NetBSD 側のコンテクスト保存として利用できる。

このスタートプログラムは、リスト 3.1 に示す NetBSD のユーザプログラムとして作成されており、専用に用意されたソフトウェア割込みを起こす。ソフトウェア割込みは `unitron_start`関数の内部から、asm 文を用いて IA-32 CPU 命令の `int <ソフトウェア割込み番号>`で発生させる。この `unitron_start`関数をユーザプログラムの `main`関数から呼び出せば、unitron の TOPPERS 部の実行を開始するカーネル処理へ遷移する。今回使用したソフトウェア割込み番号は、NetBSD と TOPPERS のどちらも使用していない 133 番を使用した。

```
 1  #include <stdio.h>
 2
 3  #define UNITRON_START_INTR_NUM 133
 4
 5  inline static int
 6  unitron_start(int debug)
 7  {
 8      int ret = 0;
 9
10      asm volatile (
11                          "movl %1, %%eax\n\t"
12                          "int %2\n\t"
13                          "movl %%eax, %0\n\t"
14                          : "=g" (ret) // 出力
15                          : "g" (debug), "n" (UNITRON_START_INTR_NUM) // 入力
16                          : "%eax" );
17      return ret;
18  }
19
20  int
21  main(int argc, char *argv[])
22  {
```

```
23      int ret;
24
25      ret = unitron_start(0);
26      printf("ret = %x\n", ret);
27
28      return 0;
29  }
```

リスト 3.2 unitron スタートプログラム

このソフトウェア割込みに対し、TOPPERS 部を起動する割込みベクタが CPU から呼ばれる (リスト 3.3)。unitron は NetBSD のコンテクストを `osc_netbsd`という名前の構造体へ保存し、既に TOPPERS の実行開始部のエントリポイントが登録されている`osc_toppers`という名前の構造体のデータを使用し TOPPERS のコンテクストへスイッチする。

```
1   /* TOPPERS のスタートアップ部分 */
2   UNITRON_IDTVEC(start)
3       pushl  $0;
4       pushl  $UNITRON_START_INTR_NUM
5       INTR_ENTRY
6
7       /* NetBSD コンテクスト保存 */
8       movl   $osc_netbsd, %edi
9       movl   %esp, OSC_ESP(%edi)
10      movl   $_C_LABEL(ret_intr), OSC_EIP(%edi)
11
12      /* TOPPERS コンテストへ切り替え */
13      movl   $osc_toppers, %esi
14      movl   OSC_ESP(%esi), %esp
15      pushl  $0 /* EFLAGS */
16      pushl  $GSEL(GCODE_SEL, SEL_KPL) /* CS */
17      pushl  OSC_EIP(%esi) /* EIP */
18      movl   %esi, _C_LABEL(curos)
```

```
19      iret
20  UNITRON_IDTVEC_END(start)
```

リスト 3.3　unitron スタート用割込みベクタ

TOPPERS のコンテクストは、図 3.4 に示すように TOPPERS のカーネルスタックを指すスタックポインタと、割込み復帰のコードを指す命令ポインタから成る。TOPPERS のカーネルスタックには、割込み復帰時のリターンアドレスとして TOPPERS スタートアップルーチンのアドレスが格納されている。このようすることで、コンテクストスイッチ後の割込み復帰時に TOPPERS が実行開始されるようにした。なお、NetBSD から呼び出したスタートプログラムは、次回、NetBSD にコンテクストスイッチした際に、ソフトウェア割込みから復帰する。

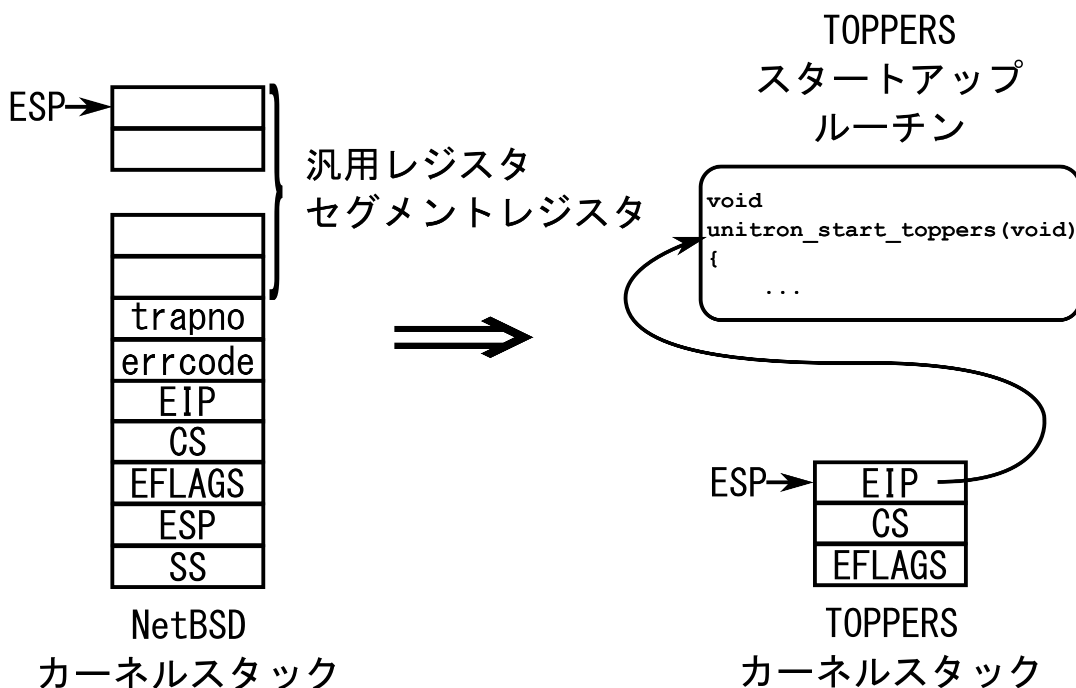

図 3.4　TOPPERS の実行開始：ソフトウェア割込みにより、NetBSD カーネル処理に遷移し、カーネルスタックを変更することでスタートアップルーチンを呼び出す。

リスト 3.4 に TOPPERS スタートアップのルーチンを示す。7 行目の `kernel_start` 関数を呼び出すことで、TOPPERS は起動する。`kernel_start` 関数は TOPPERS のアーキテクチャ非依存の関数であり、内部に変更は加えておらず、他のアーキテクチャで利用されている関数と同一である。

```
1 void
2 unitron_start_toppers(void)
3 {
4     if (sense_lock() == FALSE) {
5         sys_printf("Warn: sense_lock() == FALSE!\n\tThis process must be
              interrupt disable.\n");
6     }
7     kernel_start();
8 }
```

リスト 3.4　TOPPERS スタートアップルーチン

割込み配送部

unitron には NetBSD と TOPPERS という 2 つの OS が存在している。それぞれの OS には各割込みに対して割込みハンドラが用意されている。そのため、図 3.5 に示すように、CPU が参照する IDT（Interrupt Descriptor Table）から割込み配送部を呼び出し各 OS 側で処理すべき割込みハンドラへ割込み情報を配送する。

割込み配送部のコードをリスト 3.5 に示す。割込み配送部は TOPPERS 側の実時間性を確保するために極力少ない処理量に抑える事が望ましい。そのため、今回の実装ではアセンブリ言語を用いて記述した。割込み配送部は、呼び出されるとすぐに TOPPERS 側へ割込み配送が必要かチェックする (7 行目)。チェックは、TOPPERS 用の割込みベクタテーブルである `toppers_idt_handler`の該当割込み番号の要素に有効な割込みベクタが登録されているかで判断を行う。もし、有効な割込みベクタが登録されているのであれば、その割込みベクタに処理を移し (9 行目)、そうでなければ NetBSD 側へ配送する (12 行目)。各割込みに対する割込み配送部は 4 つのアセンブリ命令 (7〜12 行目) のみで構築されており、処理量は少なく抑えられている。この割込み配送部はマクロで記述され、17 行目以降のように各割込み番号に対応して関数として作成した。この関数は、次に述べる割り込みベクタテーブルの初期化において IDT に登録する。

```
1 CHAIN_HANDLER name, num
2     .text
3     .align 16
4     .global \name
```

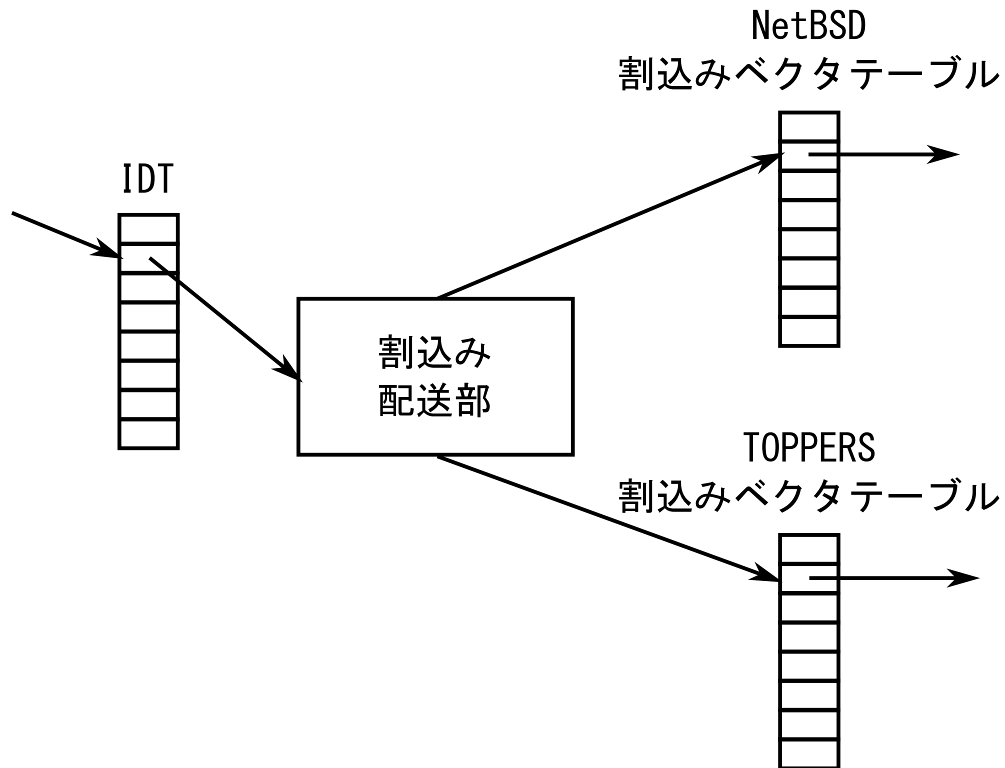

図 3.5　割込み配送部 : IDT の内容を変更し、割込み配送部から割込みを受け取るべき OS 側の割込みハンドラを呼び出す。

```
 5      .type \name, @function;
 6  \name:
 7      cmpl   $0, (toppers_idt_handler + (\num*4))
 8      je  1f
 9      jmp *(toppers_idt_handler + (\num*4))
10      /* NOTREACHED */
11  1:
12      jmp *(netbsd_idt_handler + (\num*4))
13      /* NOTREACHED */
14      .size \name, . - \name
15  .endm
16
17  CHAIN_HANDLER unitron_chain_00, 0
18  CHAIN_HANDLER unitron_chain_01, 1
```

```
19 CHAIN_HANDLER unitron_chain_02, 2
20 CHAIN_HANDLER unitron_chain_03, 3
21 CHAIN_HANDLER unitron_chain_04, 4
22 CHAIN_HANDLER unitron_chain_05, 5
23 CHAIN_HANDLER unitron_chain_06, 6
24 CHAIN_HANDLER unitron_chain_07, 7
25 CHAIN_HANDLER unitron_chain_08, 8
26 CHAIN_HANDLER unitron_chain_09, 9
27 CHAIN_HANDLER unitron_chain_0a, 10
28 CHAIN_HANDLER unitron_chain_0b, 11
29 CHAIN_HANDLER unitron_chain_0c, 12
30 CHAIN_HANDLER unitron_chain_0d, 13
31 CHAIN_HANDLER unitron_chain_0e, 14
32 CHAIN_HANDLER unitron_chain_0f, 15 // 以降省略
```

リスト 3.5 割込み配送部コード

NetBSD 側の割込みベクタテーブルには、LKM オブジェクトをロードする際に IDT に登録されていた割込みハンドラ情報をコピーし、LKM オブジェクトをアンロードする際に IDT へ書き戻すようにした。これにより、unitron システムを動的に追加、削除しても NetBSD の割込みハンドラが正しく呼ばれる。

割込みベクタテーブルの初期化処理をリスト 3.6 に示す。初期化処理では、割込みテーブル情報を変更するため、割込み禁止にする必要がある。割込み禁止の区間は 34 行目の `splhigh`の呼び出しから 49 行目の `splx`の呼び出しまでである。この区間中に、IDT に登録されていた NetBSD 用の割込みベクタを `init_netbsd_idt_handler`関数により `netbsd_idt_handler`へコピーしている (36 行目)。また、TOPPERS 用の割込みベクタを `init_toppers_idt_handler`関数により `toppers_idt_handler`へコピーしている (37 行目)。そして 39 行目から 46 行目において、割込み配送部のコードを IDT に登録する。

```
1 static void
2 init_netbsd_idt_handler(void)
3 {
4     int i;
5
```

```
 6     memset(netbsd_idt_handler, 0, sizeof (netbsd_idt_handler));
 7     for (i = 0; i < NIDT; i++) {
 8         struct gate_descriptor *gd = &idt[i];
 9         if (SDTYPE(gd) != SDT_SYS386IGT) {
10             continue;
11         }
12         netbsd_idt_handler[i] = GD2HANDLER_PTR(gd);
13     }
14 }
15
16 static void
17 init_toppers_idt_handler(void)
18 {
19     memset(toppers_idt_handler, 0, sizeof (toppers_idt_handler));
20     toppers_idt_handler[0xc0] = (vector_t)&UNITRON_IDTVEC(timer);
21 }
22
23 int
24 init_idt(void)
25 {
26     int x;
27     int i;
28
29     if (is_idtelem_valid(UNITRON_START_INTR_NUM)) {
30         /* 使用中のIDT 要素なので、使用することは× */
31         return EBUSY;
32     }
33
34     x = splhigh();
35
36     init_netbsd_idt_handler();
```

```
37     init_toppers_idt_handler();
38     memcpy(idt_backup, idt, sizeof (idt_backup));
39     for (i = 0; i < NIDT; i++) {
40         struct gate_descriptor *gd = &idt[i];
41         if (SDTYPE(gd) != SDT_SYS386IGT) {
42             continue;
43         }
44         gd->gd_looffset = HANDLER2GD_LOOFFSET(unitron_idt_chain_handler[i
               ]);
45         gd->gd_hioffset = HANDLER2GD_HIOFFSET(unitron_idt_chain_handler[i
               ]);
46     }
47     setgate(&idt[UNITRON_START_INTR_NUM], &UNITRON_IDTVEC(start), 0,
          SDT_SYS386IGT, SEL_UPL, GSEL(GCODE_SEL, SEL_KPL));
48 
49     splx(x);
50 
51     return 0;
52 }
```

リスト 3.6　割込みベクタテーブルの初期化

割込みベクタテーブルの後処理をリスト 3.7 に示す。後処理では、初期化処理時に idt_backup へ退避していた IDT 内容を書き戻す (8 行目) ことで、unitron 導入前の状態へ復元する。

```
1  int
2  fini_idt(void)
3  {
4      int error = 0;
5      int x;
6  
7      x = splhigh();
8      memcpy(idt, idt_backup, sizeof (idt_backup));
```

```
 9        splx(x);
10
11        return error;
12    }
```

リスト 3.7 割込みベクタテーブルの後処理

今回の実装では、NetBSD と TOPPERS で共有する割込み対象はタイマのみであった。タイマ割込みが発生した場合は、TOPPERS 側へ先に配送し、その後 NetBSD 側へ配送することで、両者へタイマ割込みを配送しつつ、TOPPERS 側へ低遅延な割込み配送を実現した。

OS コンテクストスイッチ部

NetBSD としての処理、TOPPERS としての処理を行うため、OS コンテクストのスイッチが必要となる。OS のコンテクストは、CPU のレジスタ群からなり、コンテクストをスイッチする際には、レジスタ群をスタックへ積み上げ、そのスタックポインタと復帰コードを指す命令ポインタを保存する。OS のコンテクストを保存する構造体 `struct os_context`をリスト 3.8 に示す。

```
1    struct os_context {
2        int esp; //スタックポインタ
3        int eip; //復帰コード命令ポインタ
4    };
```

リスト 3.8 OS コンテクスト保存用構造体

このように、保存すべきコンテクストは、命令ポインタやスタックポインタ、スタック上の CPU レジスタなど必要最小限にとどめ、コンテクストスイッチのオーバヘッドを抑える。なお、このようなコンテクストスイッチに関して我々は et/MRSA[14] として FreeBSD 上への構築経験を有している。

次に、コンテクストスイッチのコードを説明する。リスト 3.9 にコンテクストスイッチを行うコードを示す。現在動作している OS のコンテクスト変数へのポインタが `curos`に、切り替える先の OS コンテクスト変数へのポインタが `nextos`に格納された状態でコンテクストスイッチのコードは呼び出される。

```
1    /*
2     * curos, nextos が設定されていること
```

```
3   * %eax に curos の復帰ポインタが指定されていること
4   * jmp で突入
5   *
6   * nextos の復帰ポイントへ jmp する
7   */
8  UNITRON_ENTRY(do_os_ctsw)
9      movl   curos, %edi
10     cmpl   $0, %edi
11     je  debug_do_os_ctsw
12     movl   %esp, OSC_ESP(%edi)
13     movl   %eax, OSC_EIP(%edi)
14
15     movl   nextos, %esi
16     cmpl   $0, %esi
17     je  debug_do_os_ctsw
18     movl   OSC_ESP(%esi), %esp
19     movl   OSC_EIP(%esi), %eax
20
21     movl   %esi, curos
22     movl   $0, nextos
23     jmp *%eax
24     /* NOTREACHED */
25 debug_do_os_ctsw:
26     int3
27 UNITRON_ENTRY_END(do_os_ctsw)
```

リスト 3.9 コンテクストスイッチ部

処理の内容は、現在動作している OS のコンテクストを curosが指す先に保存し (9〜13 行目)、nextosが指すコンテクスト変数のデータを CPU へセットする (15〜19 行目)。その後、curosの指す先を nextosへ変更し、nextosの値を NULLにしたうえで、コンテクスト切り替え後の処理を行う (21〜23 行目)。割込み配送部と同様、こちらも極力処理量を抑えて実装した。

コンテクストスイッチが呼び出される部分は本実装では二か所存在する。一つは、TOPPERS のタスクスイッチ部であり、もう一つはタイマ処理部である。

リスト 3.10 にタスクスイッチ部を示す。TOPPERS のタスクは割込みドリブンに記述されることが一般的である。本実装では、TOPPERS 側のタスクが割込み待ち状態になった時に、NetBSD が動作するようにした。このようにした理由は、TOPPERS 側の全タスクよりも NetBSD の実行優先度を低い状態にし、TOPPERS 側の実時間性を確保するためである。また、NetBSD のカーネルはプリエンプティブであるため、TOPPERS 側が管理する割込みが発生した場合は、NetBSD 側の処理は直ちに放棄され TOPPERS 側に処理が切り替わる。

TOPPERS 側の全タスクが割込み待ちになるという事は、タスクスイッチ部で実行可能タスクが存在しないという意味になる。通常の TOPPERS であれば、全タスクが割込み待ちになった場合、30 行目から 32 行目のコメントアウトされたコードのように割込みを許可した状態で動作を停止し、割込みを待つ。unitron では、この部分を利用し NetBSD へコンテクストスイッチする (22～26 行目)。

```
 1 /*
 2  *  Task dispatcher
 3  */
 4     .text
 5     .globl dispatch
 6     .globl exit_and_dispatch
 7 dispatch:
 8     pusha
 9     movl   runtsk, %ebx
10     movl   %esp, TCB_esp(%ebx)
11     movl   $dispatch_r, %eax
12     movl   %eax, TCB_eip(%ebx)
13 exit_and_dispatch:
14 dispatch_loop:
15     movl   schedtsk, %ebx
16     movl   %ebx, runtsk
17     cmpl   $0, %ebx
18     jne dispatch_1
```

```
19      movl    $0, enadsp
20      incl    nest
21  #if 1
22      /* NetBSD へコンテクストスイッチ */
23      movl    $osc_netbsd, %eax
24      movl    %eax, nextos
25      movl    $dispatch_ret_os_ctsw, %eax
26      jmp do_os_ctsw
27  dispatch_ret_os_ctsw:
28      cli
29  #else
30      sti
31      hlt
32      cli
33  #endif
34      decl    nest
35      movl    $1, enadsp
36      jmp dispatch_loop
37  dispatch_1:
38      movl    TCB_esp(%ebx), %esp
39      movl    TCB_eip(%ebx), %eax
40      jmp *%eax
41
42  dispatch_r:
43      call    calltex
44      popa
45      ret
```

リスト 3.10　TOPPERS のタスクスイッチ部

次に、リスト 3.11 にタイマ処理部を示す。タイマ割込みの内部では、TOPPERS と NetBSD のタイマ割込みベクタを呼出す他に、両 OS のスケジューリングする。

```
1  /*
2   * - TOPPERS の割込みハンドラ実行を最優先
3   *   + NetBSD コンテクストの場合は、TOPPERS にカーネル内処理中と思わせる
4   */
5  UNITRON_IDTVEC(timer)
6      pushl  $0
7      pushl  $0xc0
8      INTR_ENTRY
9      cmpl   $0, (exc_table + (0xc0*4))
10     je  to_netbsd_handler
11
12     movl   curos, %eax
13     cmpl   $osc_toppers, %eax
14     je  1f
15     incl   nest
16 1:
17     /* TOPPERS 用の割込み処理 */
18     call   *(exc_table + (0xc0*4))
19
20     movl   curos, %eax
21     cmpl   $osc_toppers, %eax
22     je  2f
23     decl   nest
24 2:
25     call   tick_search_nextos
26     cmpl   $0, nextos
27     jne context_switch
28     /* コンテクストスイッチが発生しない */
29     movl   curos, %eax
30     cmpl   $osc_netbsd, %eax
31     je  to_netbsd_handler
```

```
32     /* TOPPERS コンテクストで割込みからの復帰 */
33     INTR_EXIT
34     /* NOTREACHED */
35 
36 context_switch:
37     /* OS 間のコンテクストスイッチ */
38     movl   curos, %edi
39     movl   %esp, OSC_ESP(%edi)
40     movl   $_C_LABEL(ret_intr), OSC_EIP(%edi)
41     movl   nextos, %esi
42     movl   OSC_ESP(%esi), %esp
43     movl   OSC_EIP(%esi), %eax
44     movl   %esi, curos
45     movl   $0, nextos
46     jmp *%eax
47     /* NOTREACHED */
48 
49 to_netbsd_handler:
50     /* NetBSD 用の割込み処理 */
51     INTR_RESTORE
52     jmp *(netbsd_idt_handler + (0xc0*4))
53     /* NOTREACHED */
54 
55 debug0:
56     int3
57 UNITRON_IDTVEC_END(timer)
```

リスト 3.11 タイマ処理部

まず、割込み処理は、TOPPERS 側のタイマ割込み用ベクタを最優先で実行する。ただし、もし割込み発生時のコンテクストが NetBSD であった場合は、TOPPERS 側であった場合へスイッチする。そして TOPPERS 側の割込みベクタから復帰すると、`tick_search_nextos`関数 (リスト 3.12) を呼び出し、OS 間のコンテクストスイッチが必

要か判断する。なお、本実装では両 OS とも 5ms のタイムスライスを与えられ交互にコンテクストスイッチする。ただし、割込み配送部の説明で述べたように、TOPPERS 側で処理すべき割込みが発生した場合は、TOPPERS 側の処理が優先的に実行される。つまり、NetBSD 側のタイムスライス間の実行中であっても、TOPPERS 側への割込みをブロックする事は無い。

```
1  /*
2   * タイマ割込み時にOS のコンテクストスイッチをすべきか判断
3   */
4  void
5  tick_search_nextos(void)
6  {
7      ctxsw_msec -= decr_msec;
8      if (ctxsw_msec > 0) {
9          nextos = NULL;
10         return;
11     }
12
13     if (curos == &osc_netbsd) {
14         nextos = &osc_toppers;
15         ctxsw_msec = toppers_msec;
16     } else if (curos == &osc_toppers) {
17         nextos = &osc_netbsd;
18         ctxsw_msec = netbsd_msec;
19     } else {
20         panic("curos[%p] does not point to osc_netbsd[%p] or osc_toppers
               [%p]\n", curos, &osc_netbsd, &osc_toppers);
21     }
22 }
```

リスト 3.12 OS 間スケジューラ

表 3.1 周期ハンドラの起動間隔 （単位：μs）

測定対象	最小値	最大値	平均値	標準偏差
TOPPERS	9203.2	10629.6	10023.8	123.3
unitron 負荷無	9273.8	10783.6	10035.3	125.2
unitron 負荷有	9356.9	10726.9	10036.4	179.6
user proc. 負荷無	8505.4	11580.6	10021.5	296.6
user proc. 負荷有	3724.9	25288.6	10022.3	458.3

3.4 評価

unitron の実時間性を評価するために、TOPPERS の周期ハンドラの周期性を測定した。ハードウェアタイマが 1ms 毎に割込みをかける環境において、10ms 周期で動作する周期ハンドラを呼び出し、起動時刻を取得する。時刻は、CPU の TSC（Time Stamp Counter）レジスタの値を使用した。この周期ハンドラ起動時刻の間隔から実際の周期性を測定した。

測定対象は、ネイティブ TOPPERS のタスクと unitron で動作する TOPPERS タスクから `sta_cyc` によって開始される周期ハンドラ、NetBSD のユーザプロセスから `ualarm` によって呼び出されるハンドラの 3 種とした。unitron と NetBSD においては、NetBSD のユーザプロセスが動作することによる影響を見るため、負荷が有る、無しの 2 状態で測定した。与えた負荷は、コマンドプロンプトから `ls -lR /`を実行し、ファイルシステムへアクセスするものとした。なお、測定環境は NFS（Network File System）を使用しているため、この負荷はネットワーク I/O となる。

測定環境は、Windows7 で動作する VMware Workstation 7.1.6 の仮想計算機上に構築した。物理 CPU は Intel Xeon 2.66GHz（4 コア）、物理メモリは 12GB で、仮想計算機側には、CPU1 コアとメモリ 128MB を割り当てた。測定は、測定対象のハンドラの起動間隔を 10,000 回測定し、その最小値、最大値、平均値、標準偏差を μs 単位で求めた。表 3.1 にその結果を示す。

TOPPERS に対し、unitron は標準偏差が負荷無で 1.9μs の増加に抑えられている。また、負荷有で 56.3μs の増加となった。現在の実装では、第 3.2.2 節で述べた通り NetBSD 側の割込み禁止命令に未対応である。そのため、この 56.3μs という増加量は、負荷によって発生するネットワーク I/O の割込み禁止状態が原因と考えられる。なお、これは、ユーザプロセスが `ualarm` を使用した場合の、負荷無での 173.3μs、負荷有での 335.0μs とい

表 3.2 ソース行数

対象	C（行）	ASM（行）
NetBSD 側	988	856
TOPPERS 側（NetBSD 依存部）	861	89

う増加量に比べ少なく抑えられている。

表 3.2 に unitron システムに関連した C 言語とアセンブリ言語のソース行数を求めた。対象は、NetBSD 側の機能を構成するソースと TOPPERS 側の NetBSD 依存部となるソースである。NetBSD 依存部とは、ネイティブ TOPPERS における CPU アーキテクチャ依存部にあたる。

ネイティブ TOPPERS の Intel IA-32 実装では、CPU アーキテクチャ依存部のソース行数は、C 言語で 1654 行、アセンブリ言語で 237 行となっている。それに対し、unitron では、約半分に抑えられている。これは、TOPPERS の起動に必要な処理や CPU 依存部分の多くを NetBSD へ任せることができるためである。NetBSD 側の機能を構成するソースは、割込み配送部など CPU アーキテクチャに依存する部分を記述する必要があるため、アセンブリ言語による記述が多い。ただし、この内の半分以上となる 525 行は、IDT 用のコードを perl スクリプトにより自動生成したものである。

3.5 関連研究

RTLinux[15] は Linux に実時間処理タスクの実行機能を追加する。タスクに対する割込み処理は通常の Linux カーネル内部の処理を通さないため、細粒度の実時間性を実現している。しかし、実時間処理の記述に独自のライブラリを用いたプログラミングが必要とされる。一方、unitron システムの実時間処理は、組込み OS として利用されている μITRON のタスクとして記述できる。

ART-Linux[16] は Linux のユーザプロセスに実時間性を与える。ユーザプロセスで実時間処理を記述できる点が利点である。一方、unitron システムはユーザプロセスで直接実時間処理を記述することはできないが、実時間処理記述の実績のある μITRON の API を使用して実時間処理を記述できる。また、ART-Linux は Linux カーネルソースを直接編集して実装しているが、unitron システムは LKM を用いているため、カーネルソースの編集と再コンパイル作業は不要である。

et/MRSA[14] は FreeBSD で動作するカーネル外スレッド機構である。et/MRSA は独自のコンテクストを持ち、カーネル内のタイムアウト関数を使用することで、FreeBSD

のカーネルコンテクストと et/MRSA のコンテクストを切り替えながら動作する。et/MRSA は LKM によりカーネルモジュール化されており、プログラム中からカーネル内の関数や変数にアクセスできる。ただし、コンテクスト切り替えがタイムアウト関数処理に依存しているため、実時間処理の粒度は中粒度にとどまっている。

Linux on TOPPERS[13] は、TOPPERS の 1 タスクとして Linux を実行するハイブリッド OS である。豊富なソフトウェア資産を Linux から流用し、実時間処理は TOPPERS のタスクとして記述できる。unitron システムは、TOPPERS を LKM 化し NetBSD カーネルへ取り込むため、TOPPERS の機能を動的に追加、削除が可能である。これを利用して、ネットワーク越しに別の TOPPERS タスクを持つ LKM オブジェクトファイルと交換できるといった利点がある。

仮想計算機を用い、汎用 OS と実時間 OS を同時に動作させる手法がある。両 OS に対する変更を抑えたうえに互いの安全性を確保しながら、両 OS の利点を享受することができる。しかし、仮想計算機の代表格である Xen[17] は実時間処理に対応しておらず、割込みの配送のオーバヘッドが大きい。実時間処理に対応している RTH（Real Time Hypervisor）[18] や SafeG[19] があるが、CPU の仮想化支援機能が必要である。それに対し、unitron システムは CPU の仮想化支援機能は必要ないため、より低コストの CPU を利用できる利点がある。

3.6 まとめ

情報家電などの組込み機器の高性能化に伴い、豊富なソフトウェア資産を利用でき、かつ、実時間性を持った組込み OS が求められている。そこで、LKM 化した μITRON を UNIX カーネルへ動的に追加と削除ができる unitron システムを提案し、実装した。

unitron システムを使用することで、UNIX の持つ、デバイスドライバやネットワーク機構、アプリケーションプログラムなどの豊富なソフトウェア資産と、μITRON の優れた実時間性の両方の恩恵を受けることができる。

現実装は、NetBSD の割込み禁止処理への対応が実現されていない。割込み禁止と許可を行う処理は、NetBSD 中のアーキテクチャ依存部にある `spl` 関数とアーキテクチャ依存部にある `cli`、`sti` 命令で行われる。ソース中にあるこれらの処理を修正し、実際に割込み禁止と許可を行うのではなく、割込み配送部が NetBSD 側へのみ割込み配送を行わないようにする。ただし、この方法では、ソースの変更が不要であるという LKM 実装の利点を損ねてしまう。そこで、上記実装で得られた知見を生かし、unitron システムをカーネルにロードする際、メモリ上にある実行コード中の `cli` と `sti` 命令を動的に書き換える予定である。このようにすることで、ソースの変更が不要という LKM 実装の利点を保持したまま、NetBSD の割込み禁止処理へ対応する。

通信は、TOPPERS が NetBSD カーネルの一部として同じメモリ空間で動作していることを利用し、共有データ領域を介して行っている。今後、TOPPERS タスクと NetBSD のユーザプロセスとで通信を行うため、TOPPERS 側のメールボックス機能への対応と NetBSD 側のインタフェースライブラリの構築を進める。インタフェースライブラリは、カーネルからユーザプロセスへデータを渡すためのデバイスドライバを用いて実現する予定である。

第 4 章

kexec システム

本章では、kexec システムについて説明する。kexec システムはアプリケーションをカーネルモードで実行する枠組みを提供する。カーネルモードで実行されるアプリケーションに対し、カーネル内資源を直接操作する変更を開発者は加えることができる。この枠組みを使用することで、アプリケーションを流用しながら組込みシステムの用途特化が可能になる。

4.1 背景

OS は CPU や記憶装置、入出力装置等の計算機資源を抽象化している。OS は抽象化した計算機資源をシステムコールによってユーザに提供している。個々のユーザやプロセスは OS の保護機構により守られており、アプリケーションはシステムコールを使うことで安全に計算機資源を利用できる。

システムコールは汎用性と堅牢性を重視しているため、一般に時間のかかる処理である。Web サーバや NFS サーバ等の I/O 操作を頻繁に行うアプリケーションはシステムコールのオーバヘッドにより、十分な性能が発揮できないとの報告がある [20]。

そこでアプリケーションが行うサービスを保護機構が無いカーネル空間で提供し、システムコールよりも低レベルの操作を使い、必要最小限の処理を行うことでボトルネックを解消する手法がある。ここで低レベル操作とは、バッファキャッシュの内容を直接扱う等、カーネル内でのみ可能な処理のことをいう。

Linux の kHTTPd[21] はカーネル内で Web サービスを行うカーネルモジュールである。kHTTPd はファイル転送の部分をカーネル内の低レベル操作で実装し、データのコピーを抑えており、ユーザ空間で動作する Apache 等の Web サーバよりも高い性能を示している。

しかし、既存のアプリケーションが行うサービスをカーネルモジュールで提供する場

合、アプリケーションのソースをカーネルモジュールにそのまま流用することはできないため、開発の効率が悪い。これは、カーネルモジュールとアプリケーションとでは、プログラムのインタフェイスやセマンティクスが異なるためである。インタフェイスの相違として、アプリケーションはユーザライブラリを使用できるのに対し、カーネルモジュールはユーザライブラリを使用できないことが挙げられる。セマンティクスの相違としては、アプリケーションはプリエンプティブに動作するのに対し、カーネルモジュールはノンプリエンプティブに動作することが挙げられる。

4.2 目的

本研究では、カーネル開発者を対象とし、アプリケーションが行うサービスをカーネルで容易に提供する環境を構築する。前述した背景を踏まえ、カーネルモジュールよりも効率の良い開発環境の提供を目的とする。

本研究でとった手法は、アプリケーションをカーネルモードで実行する [12] というものである。カーネルモードで実行することにより、アプリケーションはカーネルモジュールと同様にカーネル内資源を直接操作できる。カーネル内資源を直接操作することで、アプリケーションはシステムコールを使用する場合に比べ低いオーバヘッドで計算機資源の利用が可能になる。また kexec システムは、通常のアプリケーションが使用するのと等価なインタフェイスとセマンティクスを提供することができる。そのため、アプリケーションに変更を加えなくともカーネルモードで実行できる。開発者は、アプリケーションの計算機資源利用を効率化したい部分や、カーネル特有の機能を使いたい部分のみに集中して改良を加えることができ、実装にかかるコストを抑えることができる。

4.3 既存手法の問題

アプリケーションは、カーネルの管理する計算機資源利用にオーバヘッドが伴うという問題がある。一方、カーネルモジュールは保護やデバッグの観点から開発効率が悪いという問題がある。本節では、両者の問題について議論し、本研究での解決手法を提案する。

4.3.1 アプリケーション

アプリケーションとは、エディタやコンパイラ、Web サーバなど、利用者が計算機を利用する際に使用するソフトウェアのことである。利用者はこれらのソフトウェアを自由に作成、実行することができる。そのため、アプリケーションは本質的に不正な処理を行う可能性のあるソフトウェアである。しかし、あるアプリケーションが不正な処理を行っ

たためにカーネルが異常動作を起こしてしまうことは、計算機システムの利便性を損ねてしまい、問題である。

そこでカーネルは CPU の動作モードを変更することで自分自身を保護している。CPU の動作モードは二つ存在する。アプリケーションが動作するユーザモードと、カーネルが動作するカーネルモードである。

図 4.1 に示すようにユーザモードでは、保護機構が働くためカーネル空間のメモリや、ハードウェアを直接操作することができない。そのため、アプリケーションは計算機資源を使用する際、必ずシステムコールを用いてカーネルを呼び出す必要がある。

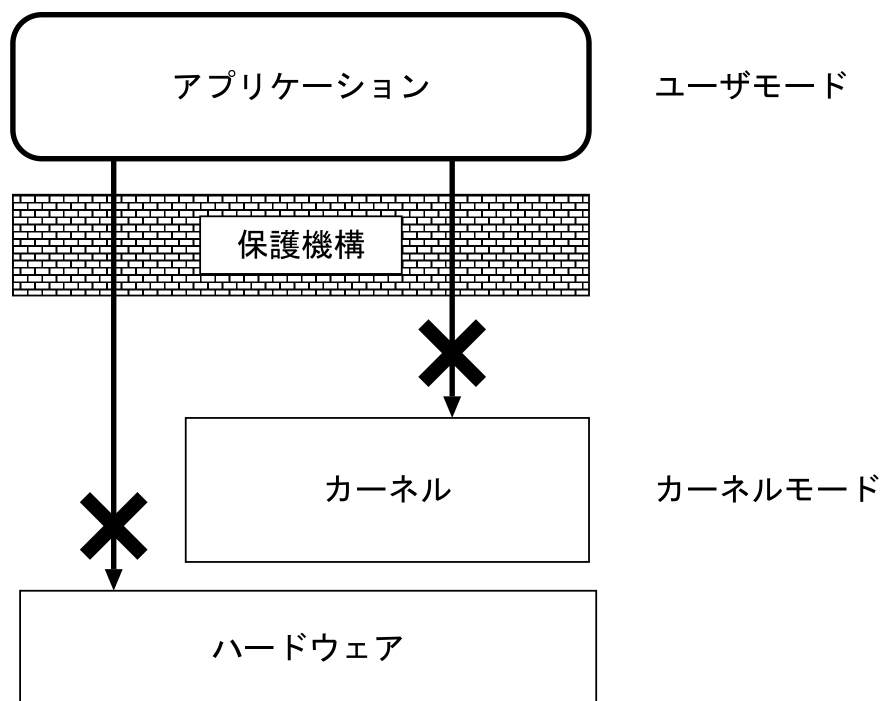

図 4.1　保護機構の概略

アプリケーションの問題

アプリケーションが使用するシステムコールは、汎用性、堅牢性、安全性を維持するために多くの処理を必要とし、オーバヘッドを伴う。I/O 処理を頻繁に行う Web サーバのようなアプリケーションはシステムコールのオーバヘッドにより性能が低下してしまう問題がある。オーバヘッドの要因として以下のことが挙げられる。

CPU の動作モード遷移の省略

図 4.2 にシステムコール処理の流れを示す。システムコールはカーネルを呼び出すため、CPU の動作モードをカーネルモードへ遷移させる必要がある。カーネルモードへの遷移は一般的にソフトウェア割り込みを用いて行う [22]。呼び出されたカーネルは、CPU レジスタの保存を行う。そして、カーネル内のシステムコール処理関数をディスパッチし、実行する。実行終了後、CPU レジスタの復帰を行いユーザモードへ復帰しシステムコール処理を完了する。アプリケーションが計算機資源を利用するときのみカーネルモードに遷移することで、OS は堅牢性を維持している。しかし、これらの処理はオーバヘッドが大きく、関数呼び出しに比べ時間のかかる処理である。

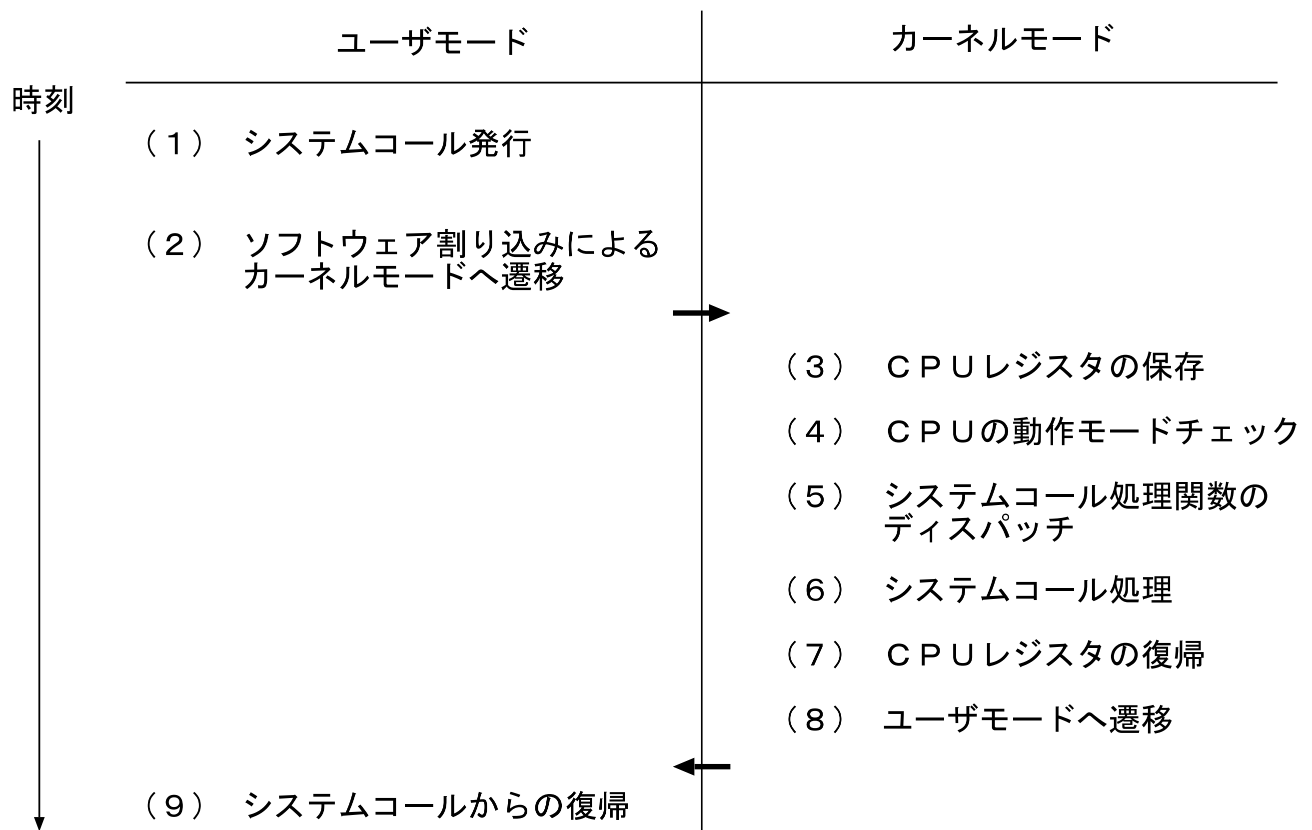

図 4.2　システムコール処理の流れ

システムコールの内部処理

システムコールの内部処理ではユーザメモリ空間とカーネルメモリ空間でのデータコピーが発生する。これらのデータコピーは本質的に冗長であることが多い [23]。例えば、Web サーバがクライアントの要求でファイル転送をする場合、ファイルのデータは図 4.3

のような流れでコピーされる。Web サーバは `read()` システムコールでファイルの内容を送信用データバッファにコピーし、それを `send()` システムコールによりクライアントに送信する。`read()` してから `send()` するまでの間、送信用データバッファは変更されない。もし、Web サーバが計算機資源を直接操作できるなら、ディスクから直接ファイル内容を NIC（Network Interface Card）に転送することで、送信データバッファへのコピーオーバヘッドを削減することが可能である。しかし、このような操作は、OS によって禁止されているため、実際には不可能である。

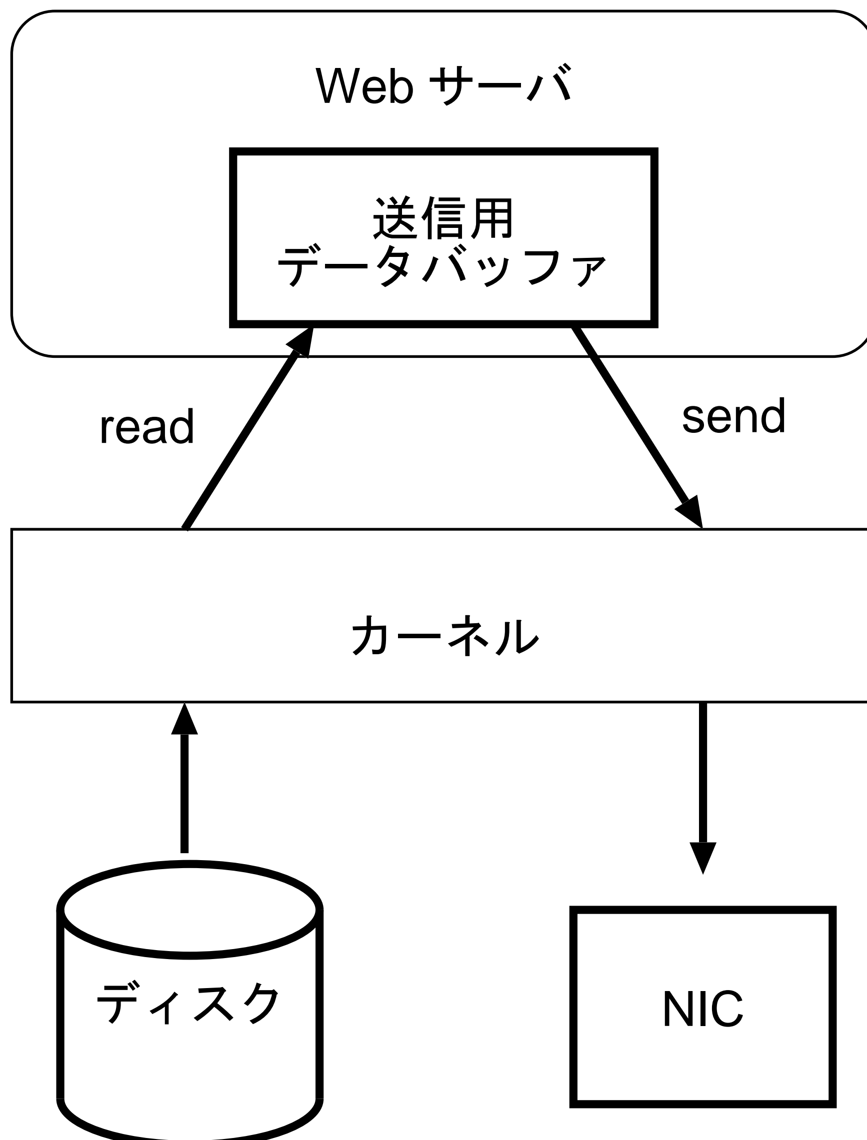

図 4.3　Web サーバのファイル転送

4.3.2　カーネルモジュール

カーネルモジュールはカーネルの機能を提供する部品である。例えば、あるハードウェアに対応したデバイスドライバの実装やファイルシステムの構築に使用される。

カーネルモジュールの問題

アプリケーションのサービスをカーネルモジュールで提供する場合、アプリケーションのソースをそのままカーネルモジュールに流用することは難しく、開発の効率が悪い。これはカーネルモジュールとアプリケーションプログラムとでは、プログラムのインタフェイスやセマンティクスが異なるためである。カーネルモジュールはアプリケーションプログラムと比較して、以下のような相違点が挙げられる。

- ユーザライブラリが使えない
  カーネルモジュールはアプリケーションが使用しているユーザライブラリが使えない。そのため、ユーザライブラリが提供する機能を別に用意する必要があり、実装のコストが大きい。
  ユーザライブラリの多くは内部でシステムコールを呼び出しているが、カーネルモジュールからはシステムコールを利用できない。そのため、カーネルモジュールからはユーザライブラリが使用できない。カーネルモジュールがシステムコールを利用できない理由として以下のことが挙げられる。
  - プロセスコンテクストの有無
    アプリケーションが利用しているメモリやファイルなどの計算機資源に関する情報はプロセスコンテクストに保存されている。システムコールは呼び出し元のアプリケーションが持つプロセスコンテクストを参照、変更しながら動作している。しかし、カーネルモジュールは独自のプロセスコンテクストを持たないため、システムコールを利用することができない。
  - メモリ空間の相違
    アプリケーションはユーザメモリ空間に配置されるのに対し、カーネルモジュールはカーネルメモリ空間に配置される。システムコールはアプリケーション呼び出されることが前提のため、入出力データがカーネルメモリ空間にあると、システムコール内部処理でエラーになり正常に動作しない。
  - 動作モードの相違
    システムコールによりソフトウェア割り込みでカーネルモードに遷移した際、カーネルは割り込み処理関数において、割り込み発生箇所の CPU 動作モード

のチェックを行う。呼び出し元がカーネルモードで動作している場合、このチェックでエラーになりシステムコールを利用することはできない。

- ノンプリエンプティブ
  アプリケーションはプリエンプティブに動作している。そのため一つのアプリケーションが処理を占有することは無く、公平にスケジューリングされている。それに対し、カーネルモジュールはノンプリエンプティブに動作しており、自発的に処理を放棄しない限り他のプロセスやカーネルに処理が渡ることが無い。プリエンプティブ動作を前提とするアプリケーションは自発的に処理を放棄することが無いため、ノンプリエンプティブに動作させると処理を占有する問題がある。
- 不正な処理に対するシステムの脆弱性
  アプリケーションはユーザモードで動作しており、不正な処理はカーネルによって検知され、他のアプリケーションやカーネルに影響を与えることは無い。一方、カーネルモジュールはカーネルモードで動作しており、カーネルに対し不正な処理を行った場合、OS は異常動作、もしくは異常停止してしまう。このことはカーネルモジュールのデバックを難しくしている [24]。

4.4　本研究の提案

アプリケーションは計算機資源を利用する際、OS の保護境界をまたぐオーバヘッドを伴う。そのため、I/O 操作を頻繁に行う Web サーバのようなアプリケーションは性能が低下してしまう問題がある [20]。また、アプリケーションは、システムコールという限られたインタフェイスでしか計算機資源利用ができないため、柔軟性に欠ける問題がある。

そこで、これらの問題を解決するために、アプリケーションのサービスをカーネルモジュールで実装する手法がある。この手法により、以下のことが可能になる。

- CPU の動作モード遷移の削除
  カーネルモジュールはカーネルモードで実行されるため、計算機資源を操作する際に CPU の動作モード間の遷移が必要無い。これにより、アプリケーションが計算機資源利用のたびに必要であった CPU の動作モード遷移のオーバヘッドを削減できる。
- カーネル内資源の直接操作
  計算機資源利用の際、アプリケーションはシステムコールという限られたインターフェイスしか持たず、柔軟性に欠ける。それに対しカーネルモジュールは、ハードウェアやカーネル内資源を直接操作でききため、あるサービスに特化した機能を柔軟に構築できる。例えば Linux の kHTTPd はデータコピーを削減したファイル転

送機構を独自に実装することで効率的な Web サービスを実現している [21]。

しかし、カーネルモジュールはインタフェイスとセマンティクスの相違によりアプリケーションのソースを流用することができない。そのため、アプリケーションの初期化部分やエラー処理部分といった、計算機資源利用と関係無い部分も実装し直す必要があり、実装効率が悪いという問題がある。

そのため、アプリケーションのサービスをより容易にカーネルに取り込む機構が望まれている。

そこで、本研究では、このような機構を実現するために、アプリケーションをカーネルモードで実行する手法を提案する。カーネルモードで動作するアプリケーションには、kexec システムにより通常のアプリケーションと同様のインタフェイスとセマンティクスが提供される。kexec システムを使用することで、開発者はアプリケーションのソースに変更を加えなくとも、カーネルの一部としてそのまま実行できる。カーネルモードで動作するアプリケーションは、カーネルモジュールと同様にハードウェアやカーネル内資源を直接操作することができる。この操作をアプリケーションの一部に加えることで、高速化やカーネル特有の機能を使えるように改良することが可能である。

例えば、ある開発者が kexec システムを用いて、カーネルに Web Proxy の機能を追加したい場合、以下のような手順を踏む。

1. 既存の Web Proxy のソースを用意する。
2. kexec システムを用いて Web Proxy をカーネルモードで動作させる。
3. 開発者はファイル転送などの計算機資源利用部分を改良する。

改良箇所以外のソースはそのまま使用でき、開発者は改良部分にのみ集中して実装をすすめることができる。そのため、ソースが流用できないカーネルモジュールに比べ、kexec システムはより容易な実装環境を提供する。

4.5 システム概要

通常のアプリケーションは計算機資源利用にオーバヘッドを伴うという問題がある。一方、カーネルモジュールは低オーバヘッドな計算機資源利用が可能であるが、実装コストが大きいという問題がある。

そこで本研究ではこれらの問題を解決するため、アプリケーションをカーネルモードで実行するシステムを実現する。本節ではこのシステムの概要について述べる。

kexec システムはカーネル開発者を対象とし、アプリケーションのサービスを容易にカーネルへ取り込む機構を提供する。カーネル開発者は、カーネルに取り込みたい既存の

アプリケーションを用意し、図 4.4 のように kexec システムと組み合わせてカーネルモードで実行する。kexec システムは通常のアプリケーションが使用するインタフェイスとセマンティクスをカーネル内で提供することで、アプリケーションをカーネルの一部として実行する。常にカーネルモードで動作するアプリケーションは、CPU 動作モード遷移が必要無く、その分のオーバヘッドを削減できるため、効率的な計算機資源利用が可能になる。

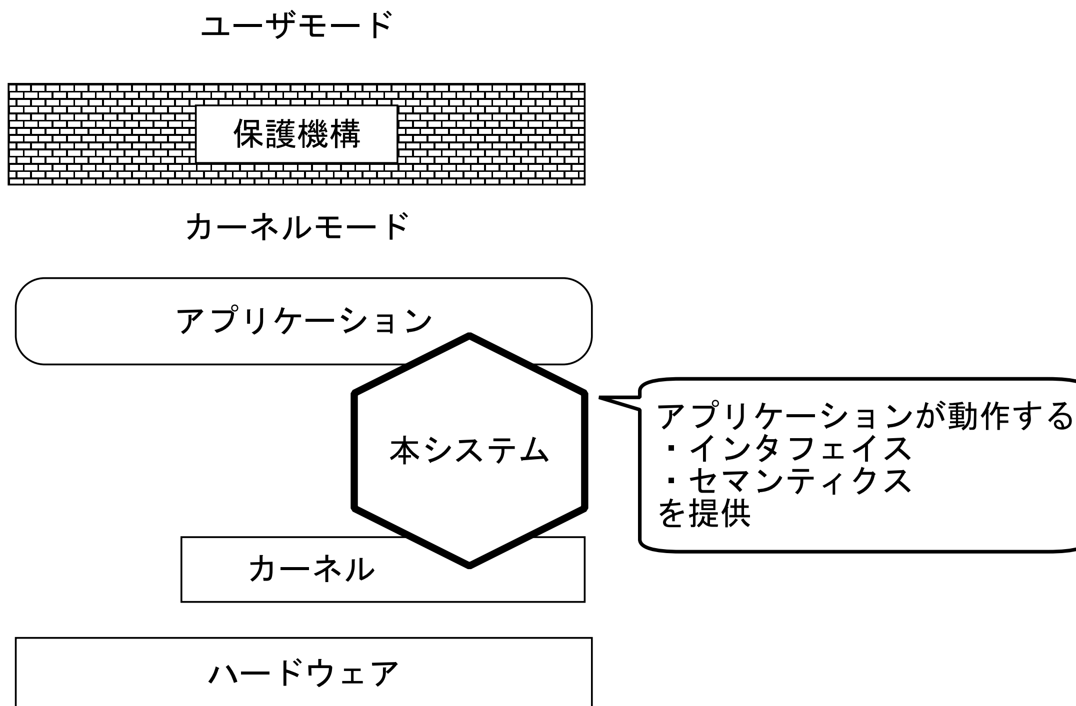

図 4.4　アプリケーションのカーネルモード実行

アプリケーションはカーネルモードで動作することで、カーネル内資源やハードウェアを直接操作することができる。このことを利用して開発者は、図 4.5 に示すように計算機資源の利用効率を高めたり、カーネル特有の機能をアプリケーション内に追加することで、カーネルとアプリケーションの融合をより進めることができる。

kexec システムによって実行されるアプリケーションとカーネルモジュール、ユーザモードで動作するアプリケーションの比較を表 4.1 に示す。また、kexec システムによりカーネルモードで動作するアプリケーションの概要を、メモリ配置図を用いて図 4.6 に示す。通常のユーザプログラムと同様に、アプリケーションとユーザライブラリはユーザのメモリ空間に配置される。アプリケーションはカーネルモードで動作することでカーネル資源を直接扱うことができる。

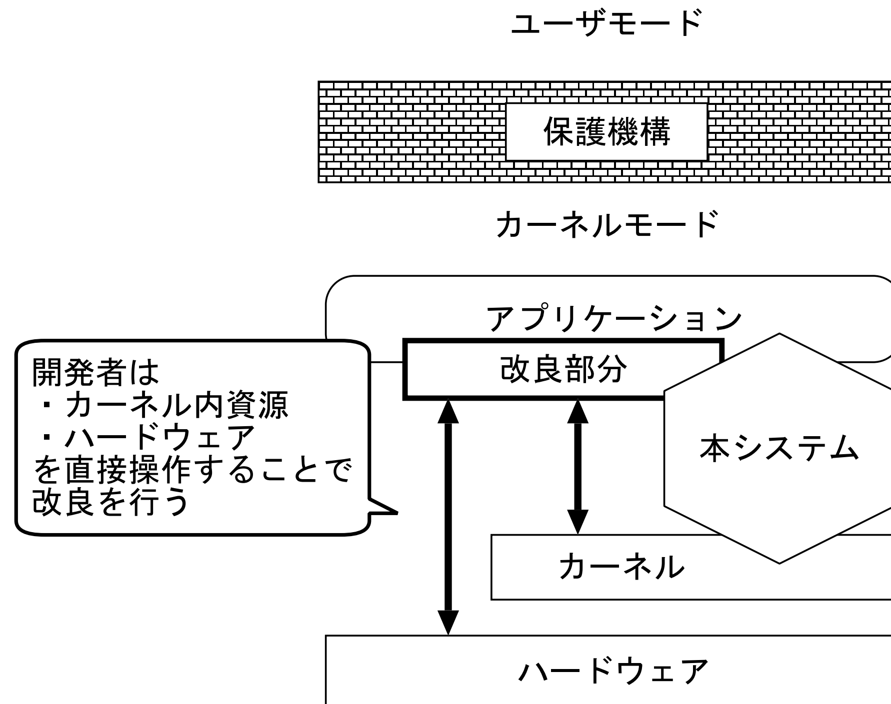

図 4.5　カーネル内資源の直接操作

表 4.1　システムの比較

	カーネル モジュール	kexec システム	ユーザ プログラム
hline 動作モード	カーネル	カーネル	ユーザ
メモリ空間	カーネル	カーネル/ユーザ	ユーザ
スタックのあるメモリ空間	カーネル	カーネル/ユーザ	ユーザ
システムコール呼び出し速度	N/A	○	×
カーネル内資源の直接操作	○	○	×
ユーザライブラリの使用	×	○	○
プリエンプション	×	○	○
デバッグの容易さ	×	△	○

4.6　kexec システムを実現するにあたっての問題

kexec システムを実現するために、以下の問題を解決する必要がある。

- カーネルモード実行機構

  OS はアプリケーションをユーザモードで実行する機構を提供しているが、カー

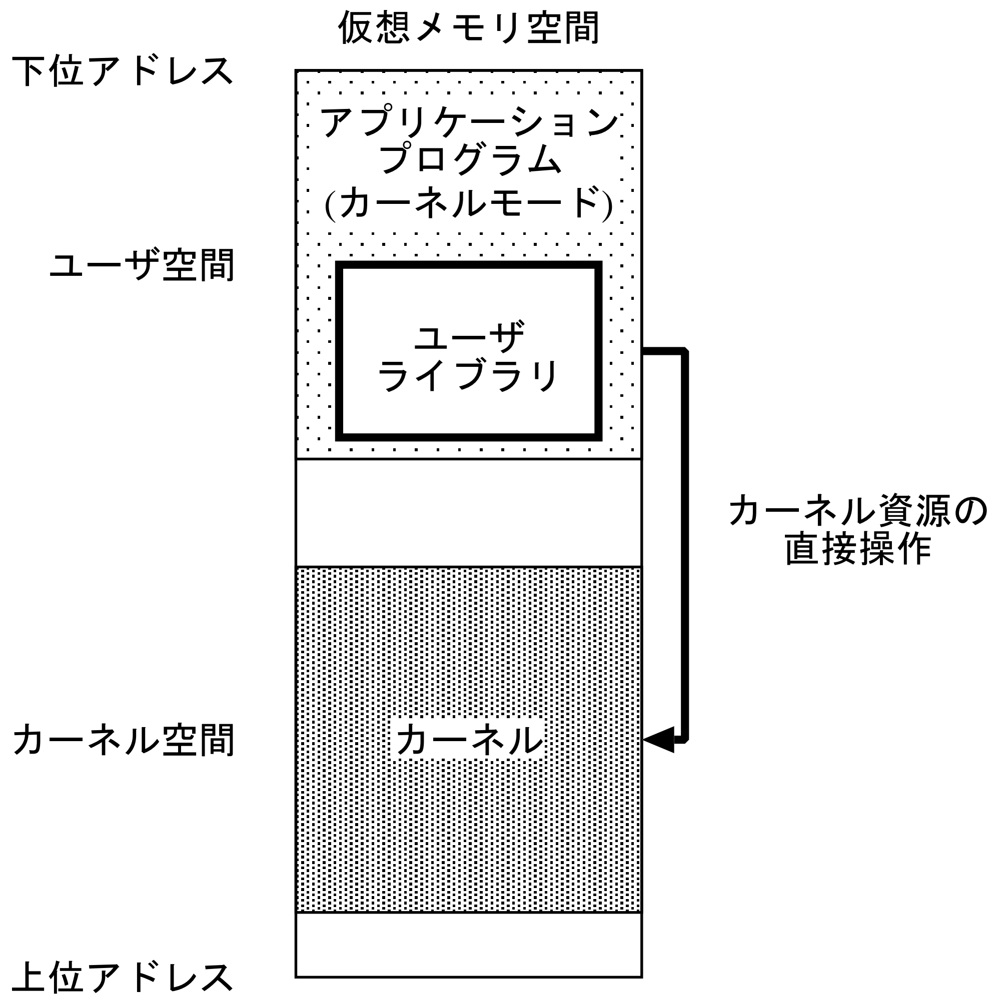

図 4.6　カーネルモード実行時のメモリ配置

ネルモードで実行する機構は提供していない。そのため、kexec システムはアプリケーションをカーネルモードで実行する機構を実装する必要がある。その際、以下のことを実現しなければならない。

– プロセスコンテクストの確保
  使用中のファイルなどの計算機資源に関する情報は、カーネル中のプロセスコンテクストに保存される。計算機資源を扱うシステムコールは、プロセスコンテクストが必須である。そのため、実行開始機構はプロセスコンテクストを確保する必要がある。
– 実行イメージのメモリ読み込み
  アプリケーションを実行するためには、実行イメージをファイルからメモリ上に読み込む必要がある。CPU は読み込まれた命令列を順次実行することで、処理を行う。

- ユーザライブラリの使用
  通常のアプリケーションはユーザライブラリを使用して構築されている。kexec システムによりカーネルモードで実行されるアプリケーションでも、同様にユーザライブラリを使用できなければならない。機能の豊富なユーザライブラリを利用できることはアプリケーションの開発効率を高める。
- システムコールの置換
  カーネルモードで動作するアプリケーションは、システムコールを呼び出した際、CPU の動作モード遷移を行う必要が無い。そのため、動作モード遷移に伴う CPU レジスタ保存といった処理を省略でき、オーバヘッドを低減することが可能である。しかし、カーネルはシステムコール用の割り込み処理関数において、呼び出し元の CPU 動作モードのチェックを行う。カーネルモードで呼び出された場合、カーネルはエラーと判断してシステムコールの実行を中断してしまう。そこで、カーネルモードでも呼び出し可能なシステムコールを用意し、それを呼び出すようにアプリケーション中のシステムコールを置換する必要がある。
- カーネル内シンボルの解決
  カーネルモードで動作することにより、アプリケーションはカーネル内資源を直接操作することができる。その際、開発者はカーネル内資源を関数名や変数名といったシンボルを介して操作や参照を行う。そのため、カーネル内シンボルの解決を行う機構が必要である。
- スケジューリング
  通常のアプリケーションはプリエンプティブに動作している。そのため、一つのアプリケーションが処理を占有することはなく、公平にスケジューリングされている。kexec システムによりカーネルモードで実行されるアプリケーションでも、同様にプリエンプティブに動作させ、自発的に処理を放棄しなくとも他のプロセスやカーネルに処理を渡さなければならない。

アプリケーションがカーネルモードで動作することは、システムの安全性を損ねる可能性がある。カーネルモードで動作するアプリケーションがカーネルに対し不正な処理をした場合、OS は異常動作、もしくは異常停止してしまう。このことはデバッグを難しくするため、何らかの対策が必要であるが、現在、kexec システムでは安全性の対策はとっておらず、今後の課題となっている。

しかし、アプリケーションが流用できる kexec システムはカーネル内資源を直接操作する部分がカーネルモジュールに比べ少なく、デバッグの範囲を限定することができるため、デバッグはカーネルモジュールよりも容易である。

4.7　設計方針

開発効率が高く、拡張の容易なカーネルウェアを実現するため、以下のような設計方針をたてた。

- アプリケーションの変更は最小限：
  アプリケーションの再利用性を高めるため、アプリケーションに必要な変更は少ないことが望ましい。アプリケーションを再コンパイルせずともカーネル内実行できることが理想である。
- カーネルウェアから計算機資源が直接操作可能：
  I/O 処理の効率化を行うには計算機資源、つまりカーネル内資源を直接扱えることが必要である。そのために、カーネル内の関数や変数をカーネルウェア内から扱えなければならない。
- OS や他のアプリケーションへの影響は最小限：
  kexec システムを導入したことによって、OS や関係ない他のアプリケーションへ影響を与えることは望ましくない。
- OS は広く利用されている物を使用：
  広く利用されている OS には、その上で動作する豊富なアプリケーションが存在する。アプリケーションが豊富にあれば、kexec システムの利用局面を増やすことができる。

4.7.1　要求点

kexec システムはカーネルウェアに対し、ユーザモード動作と同じインタフェイスとセマンティクスを提供し、かつ、カーネル内実行を利用した拡張性も提供する。そのための要求点を以下にあげる。

- カーネル内実行環境の提供：
  アプリケーションをカーネル内で実行する機構が必要である。その際、アプリケーションの変更は最小限に抑える。また、カーネルウェア中の一部の処理のみをカーネル内実行できるように、開発者側で動作モードの切替えを可能にするインタフェイスを提供する。これにより、I/O 処理のオーバヘッドが大きな部分のみをカーネル内実行し、オーバヘッドを削減できる。
- システムコールの関数呼出し化：
  カーネルウェアは、ソフトウェア割り込みを使った通常のシステムコールを使わず

とも良い。そこで、呼出しオーバヘッドの少ない関数呼出し型のシステムコールを提供する。このシステムコールを使うことで、開発者はアプリケーションに変更を加えなくとも性能向上を実現できる。

- カーネル内シンボルの解決：
  カーネル内の関数や変数をカーネルウェアから扱うには、それらのシンボルとアドレスの対応を解決する仕組みが必要である。これにより、カーネルウェアからカーネル内の資源を直接操作し、処理内容に応じた改良することができる。
- プリエンプティブ：
  カーネルウェアは、ユーザモードで動作するときと同様にプリエンプティブであり、処理を占有してはいけない。そのため、割り込みが発生した際、必要ならば他の アプリケーション へ処理を明け渡す必要がある。
- 安全性：
  カーネルウェアが異常動作をした場合、OS に対して悪影響を与える可能性がある。そのため、安全性に対して対策をとらなくてはならない。ただし、最近の OS は、個人用、もしくは専用サーバとして使われることが多い。そのため、kexec システムでは、保護の厳格さよりも、導入のコストや実行時のオーバヘッドを抑えることに重点を置く。

4.8　実装

本節では kexec システムの実装について述べる。まず、kexec システムの構成を述べ、その後実装方法について議論する。

4.8.1　システム構成

kexec システムの実装は NetBSD 上で行った。NetBSD は Unix の一種であり、世の中で広く使われている OS である。また、NetBSD はソースが公開されており自由に改変ができるため、カーネルに手を加える必要がある kexec システムのプラットフォームに適している。

kexec システムの構成は、図 4.7 に示すように大きく二つのモジュールに分割される。カーネルモジュールとアプリケーション用追加モジュールである。この二つのモジュールを使用することで、kexec システムはアプリケーションをカーネルモードで実行する。以下にそれぞれのモジュールの概要を述べる。

- カーネルモジュール

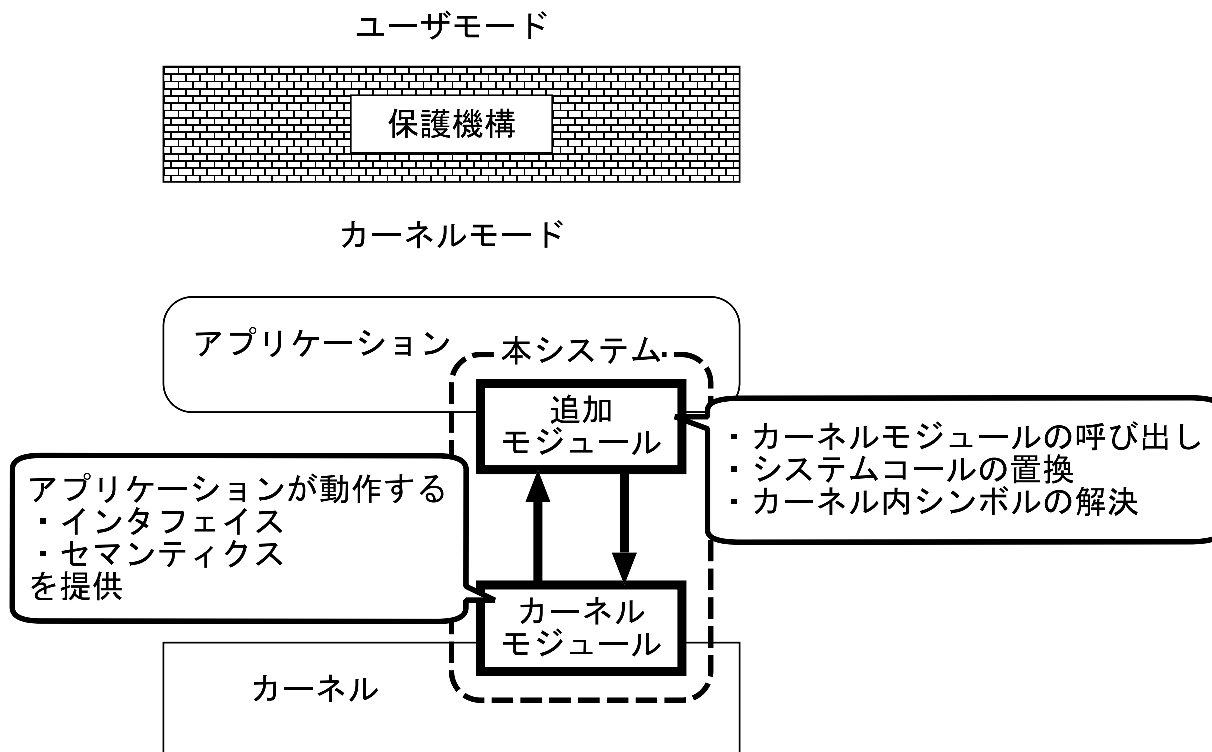

図 4.7 kexec システムの構成

カーネルモジュールは、カーネルモードで動作するアプリケーションに対し、通常のアプリケーションと同じインタフェイスとセマンティクスを提供する機能を持つ。カーネルモジュールは NetBSD の LKM 機構を用いて実装する。LKM は実行中のカーネルに対して動的にカーネルモジュールの追加、削除が行える。そのため、カーネルソースの変更やカーネルの再コンパイルは必要無く、kexec システムの導入コストを抑えることができる。

- アプリケーション用追加モジュール

  アプリケーション用の追加モジュールは、上述したカーネルモジュールを呼び出す機能と、システムコールを関数呼び出しにする機能、カーネル内のシンボルを解決する機能を持つ。追加モジュールは、アプリケーションから暗黙的に呼び出されるよう実装したため、明示的に呼び出す必要は無い。そのためこの追加モジュールを使用するには、アプリケーションのソースを変更する必要は無く、追加モジュールを再リンクするだけで良い。また、この追加モジュールはアプリケーションの種類を問わず同一であり、アプリケーションごとに異なる追加モジュールを用意する必要は無い。

4.8.2 カーネルモード実行機構

ここではアプリケーションをカーネルモードで実行する機構について述べる。

実行イメージのメモリ配置方法

アプリケーションを実行するには、実行イメージをファイルからメモリ上に読み込む必要がある。また、実行イメージをメモリ上に読み込む際に、プロセスコンテクストをカーネル内に確保しなければならない。これは、アプリケーションのメモリ使用情報や、実行中に使用する計算機資源の情報をカーネルが管理するためである。

この処理を実現するには二つ方法が考えられる。

- 既存手法の利用

  これは既存の `fork()` システムコールと `exec()` システムコールを利用する方法である。アプリケーションは、通常のプロセス生成と同様に `fork()` システムコールによりメモリ空間とプロセスコンテクストが確保され、その後 `exec()` システムコールにより実行される。

  この方法は、既存の `fork()` と `exec()` を使うことで、kexec システムの実装コストを抑えることができる利点がある。しかし、通常のプロセス実行と同じであり、実行開始時の CPU 動作モードはユーザモードとなるため、何らかの方法でカーネルモードに移行する仕組みが必要となる。

- 独自実装

  これは、`fork()` や `exec()` が行うプロセスコンテクストの確保や実行イメージのロードを独自実装で行う方法である。この方法は既存手法を使うのに比べ柔軟性があり、実行開始時の CPU 動作モードをカーネルモードにすることも可能である。しかし、kexec システムの実装コストが大きくなる。

kexec システムは、既存手法を用いる方法で実装を行った。これは、kexec システムの実装コストを抑えたかったことと、既存手法を用いた場合に必要となるカーネルモードへの移行機構が容易に実現できるためである。カーネルモードへの移行機構については次に述べる。

カーネルモードでの処理開始方法

kexec システムは、既存の `fork()` システムコールと `exec()` システムコールを用いてプロセスコンテクストの確保と、実行イメージのメモリへの読み込みを行う。`exec()` により実行開始したアプリケーションは、この段階ではユーザモードであるため、カー

ネルモードへ移行する仕組みが必要である。kexec システムではこの仕組みをカーネルモジュールを呼び出すことにより実現する。

カーネルモードへ移行する機構を図 4.8 に示す。図 4.8 中の `main()` 関数がアプリケーションの本体であり、カーネルモードで実行を開始したい部分である。`exec()` から処理が渡る実行開始部分のエントリポイントは後述する方法で `_kexec_start()` 関数に変更する。処理の流れを以下に挙げる。

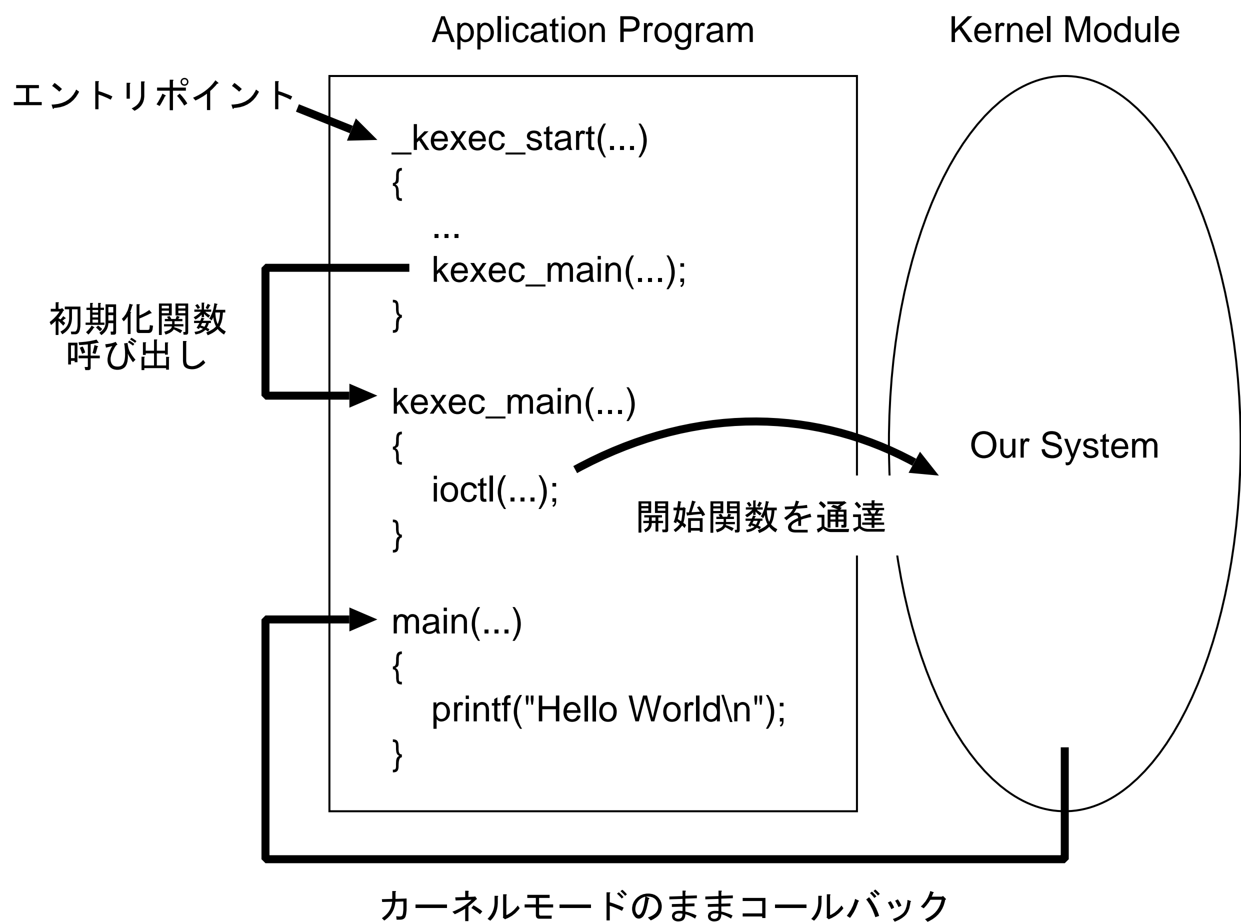

図 4.8　カーネルモードへの移行方法

1. `_kexec_start()` 関数
   `_kexec_start()` は `exec()` から処理が渡されるエントリポイントの関数である。内部では動的リンクライブラリの実行時リンクや、コマンドライン引数をスタックに積み上げる処理をする。それらの処理をした後、 `kexec_main()` 関数を呼び出す。

2. kexec_main() 関数
   kexec_main() はカーネルモードに移行するためにカーネルモジュールを呼び出す関数である。リスト 4.1 にソースを示す。main() 関数を呼び出すために必要な情報を構造体に格納（24〜33 行）し、ioctl() システムコールを用いてカーネルモジュールに通達（38 行）する。
3. カーネルモジュール
   カーネルモジュールは、ioctl() によって通達された情報を元に、main() 関数をカーネルモードで実行する。ソースをリスト 4.2 に示す。カーネルモジュールはカーネルモードで動作しているため、そのまま main() 関数をコールバックする（21 行）ことで、main() 関数をカーネルモードで実行する。この時、カーネルモジュールは以下の処理も行う。
   - システムコールのエントリの変更（12、13 行）
   - スタックの切り替え（16〜18 行）

   これらの処理内容については後述する。

```
1  #define DEV_FILE "/dev/kexec"
2
3  /*
4   * カーネルモジュールを呼び出し、
5   * main 関数をカーネルモードで実行する。
6   */
7  int
8  kexec_main(int argc, char **argv)
9  {
10     int ret;
11     int fd;
12     caddr_t esp;
13     struct callback_args kargs;
14     extern syscall_entry_t syscall_entry;
15
16     if ((fd = open(DEV_FILE, O_RDWR)) < 0) {
17         return ENOENT;
18     }
```

```
19
20      /*
21       * カーネルモジュール中で必要な情報を構造体に格納
22       */
23       /* 現在のスタックのアドレスを得る */
24      GET_ESP(esp);
25      kargs.syscall_entry_pp = &syscall_entry;
26      /* main 関数が使用するスタックのアドレス */
27      kargs.user_stack_addr = esp - KEXEC_USER_ESP_OFFSET;
28      /* main 関数のアドレス */
29      kargs.main_ptr = main;
30      /* main に渡すコマンド引数の個数 */
31      kargs.argc = argc;
32      /* コマンド引数 */
33      kargs.argv = argv;
34
35      /*
36       * カーネルモジュールを呼び出す
37       */
38      if ((ret = ioctl(fd, KEXEC_IO_CALLBACK, &kargs)) < 0) {
39          return ret;
40      }
41      ret = kargs.ret;
42
43      return ret;
44  }
```

リスト 4.1　kexec_main() 関数のソース

```
1  /*
2   * 通達された main 関数をカーネルモードのままコールバック
3   */
```

```
4  static int
5  callback_main(struct proc *p, struct callback_args *uap)
6  {
7      main_t main_ptr;
8      syscall_entry_t user_syscall;
9      int ret;
10
11     /* システムコールエントリの変更 */
12     user_syscall = *(uap->syscall_entry_pp);
13     *(uap->syscall_entry_pp) = kernel_syscall;
14
15     /* スタックの切り替え */
16     UserESP = uap->user_stack_addr;
17     GET_ESP(KernelESP);
18     SET_ESP(UserESP);
19
20     /* main 関数の実行 */
21     main_ptr = uap->main_ptr;
22     ret = (*main_ptr)(uap->argc, uap->argv);
23
24     SET_ESP(KernelESP);
25     *(uap->syscall_entry_pp) = user_syscall;
26     uap->ret = ret;
27
28     return 0;
29 }
```

リスト 4.2　カーネルモード実行部のソース

4.8.3　エントリポイントの変更

上述したように、カーネルモードで動作するアプリケーションは、通常の実行方法と異なり、実行開始部分で kexec システムのカーネルモジュールを呼び出す必要がある。通常のアプリケーションは、システムの用意したエントリポイントである`_start()`関数から処理を開始し、`main()`関数を呼び出す。kexec システムはエントリポイントを`_start()`関数から`_kexec_start()`関数に変更することで、カーネルモジュールを呼び出す処理を実行する。

エントリポイントの変更はリスト 4.3 に示すように、コンパイラへのオプションで実現でき、特別な機構を用意する必要は無い。`_kexec_start()`関数は、kexec システムのアプリケーション用追加モジュールとして用意されており、アプリケーションのコンパイル時にリンクされる。

```
1 % gcc -e _kexec_start source.c ...
```

リスト 4.3　エントリポイントの変更法

4.8.4　システムコールの置換

既存手法の問題において、カーネルモジュールからシステムコールが呼び出せない理由として、以下の三つを挙げた。

- メモリ空間の相違
- プロセスコンテクストが無い
- 動作モードの相違

これらの内、上の二つは上述したカーネルモード実行機構において解決した。しかし、動作モードの相違による問題を解決しなければシステムコールの利用ができない。

この問題は、カーネル内の割り込み処理関数における CPU 動作モードのチェック部が原因である。このチェック部でシステムコールの呼び出し元の CPU 動作モードのチェックを行い、もしカーネルモードで呼び出しているならば、エラーとなってシステムコールの処理を中断する。このチェックは、カーネル内で誤った割り込みが発生した際の誤動作を防止する働きがある。

このチェックを回避する方法として、以下の二つの方法が考えられる。

- CPU 動作モードのチェックの削除
  これは、カーネル内の割り込み処理関数で行われる CPU 動作モードのチェックを

削除する方法である。チェック自体を削除することで、カーネルモードからでもシステムコールを呼び出せるようになる。しかし、この方法ではカーネル内で誤った割り込みが発生した際の誤動作を防止することができなくなる欠点がある。

- システムコールを関数呼び出しへ置換

  これは、システムコールの呼び出し方法を割り込み型から関数型へ置換する方法である。アプリケーションがカーネルモードで動作しているならば、ソフトウェア割り込みを起こして CPU 動作モードを遷移する必要は無い。

  システムコールを関数呼び出しにすることで、カーネル内の割り込み処理関数を介さずにシステムコールが利用できる。この方法により割り込み処理関数内のチェックを回避する。また、割り込み処理関数中の CPU レジスタ保存も省略することができるため、低オーバヘッドなシステムコール呼び出しが可能になる。

kexec システムはシステムコールを関数呼び出しに置換する方法を採用した。採用の理由は、こちらの方法はカーネルの動作に対する影響が少ない点と、低オーバヘッドな呼び出しが可能である点を考慮したためである。

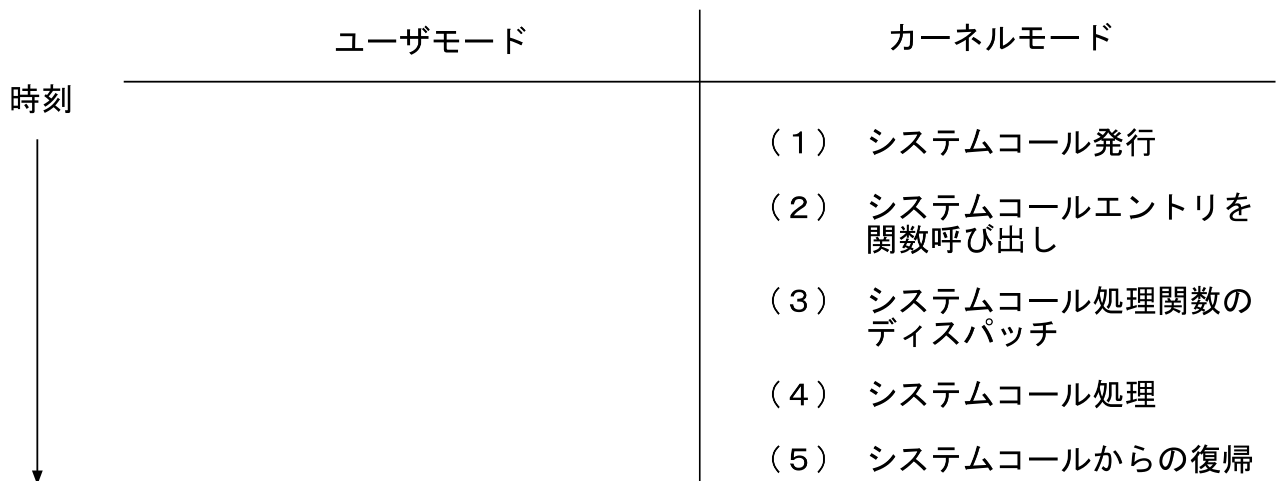

図 4.9 関数型システムコールの処理の流れ

関数型システムコールの処理の流れを図 4.9 に示す。通常のシステムコールの処理（p.38 の図 4.2 参照）に比較して、関数型システムコールは CPU 動作モード遷移に伴う CPU レジスタの保存、復帰の処理を省略できる。そのため、システムコール利用に伴うオーバヘッドが削減され、システムコールを頻繁に利用するアプリケーションは、ソースを変更しなくとも高速に動作するようになる。

アプリケーションから呼び出されるシステムコールを関数型に置換するために、kexec システムでは後述するインターポジショニングという手法を用いる。NetBSD では約 200

個のシステムコールが提供されている。これらのシステムコールの名前、返り値、引数が記述されたテンプレートファイルがあり、置換用のソースはほぼ自動生成可能である。

リスト 4.4 に自動生成したソースの一部を示す。呼び出されたシステムコールは、CPU の動作モード遷移をせずにシステムコールエントリを呼び出し、対応するカーネル内の処理関数をディスパッチする。

```
 1  /*
 2   * syscall: "read" num: 3
 3   */
 4
 5  ssize_t
 6  _read(int fd, void *buf, size_t nbyte)
 7  {
 8      struct kexec_frame frame;
 9      ssize_t ret;
10
11      init_kexec_frame(&frame);
12      /* システムコールエントリの呼び出し */
13      syscall_entry(&frame, SYS_read, fd, buf, nbyte);
14      if (frame.is_error) {
15          errno = frame.eax;
16          ret = (ssize_t)-1;
17      } else {
18          ret = frame.eax;
19      }
20      return ret;
21  }
22  ssize_t
23  read() __attribute__ ((alias("_read")));
```

リスト 4.4　自動生成された置換用ソース (`read()` システムコール)

自動生成したシステムコールの内、約 20 個は手で変更を加える必要がある。これはライブラリが提供するシステムコールのインタフェイスと、OS が想定するインタフェイス

の相違のためである。このようなシステムコールの例としてリスト 4.5 に示す pipe() システムコールがある。pipe() システムコールは、カーネル内の対応関数の処理が終了した後、自前でファイルディスクリプタの設定を行わなければならない（18〜21 行）。これらの変更は軽微であるため、kexec システムの開発にかかるコストは少ない。

```
1  /*
2   * syscall: "pipe" num: 42
3   */
4
5  int
6  _pipe(int *fdp)
7  {
8      struct kexec_frame frame;
9      int ret;
10     10
11     init_kexec_frame(&frame);
12     /* システムコールエントリの呼び出し */
13     syscall_entry(&frame, SYS_pipe, fdp);
14     if (frame.is_error) {
15         errno = frame.eax;
16         ret = (int)-1;
17     } else {
18         /* ファイルディスクリプタの設定 */
19         fdp[0] = frame.eax; /* read */
20         fdp[1] = frame.edx; /* write */
21         ret = 0;
22     }
23     return ret;
24 }
25 int
26 pipe() __attribute__ ((alias("_pipe")));
```

リスト 4.5 一部変更が必要な置換用ソース (pipe() システムコール)

kexec システムによって実行されるアプリケーションは、実行開始の段階ではユーザモードで動作している。そのため、ソフトウェア割り込み型のシステムコールも利用できなくてはならない。そこで、置換したシステムコールは関数ポインタを変更することで、ユーザモードではソフトウェア割り込みを発生させ、カーネルモードでは関数呼び出しとなるように実装されている。関数ポインタの変更はカーネルモジュールから `main()` 関数をコールバックする時のみ起きる（p.53 リスト 4.2 の 11、12 行参照)。

4.8.5　インターポジションング

ここではシステムコールの置換方法であるインターポジショニングについて述べる。図 4.10 に示すように、通常のアプリケーションはライブラリに含まれるシステムコールを呼び出す。同様にライブラリ中の関数も同じシステムコールを呼び出す。そのため、アプリケーションとライブラリから呼ばれるシステムコールを置換しなければならない。

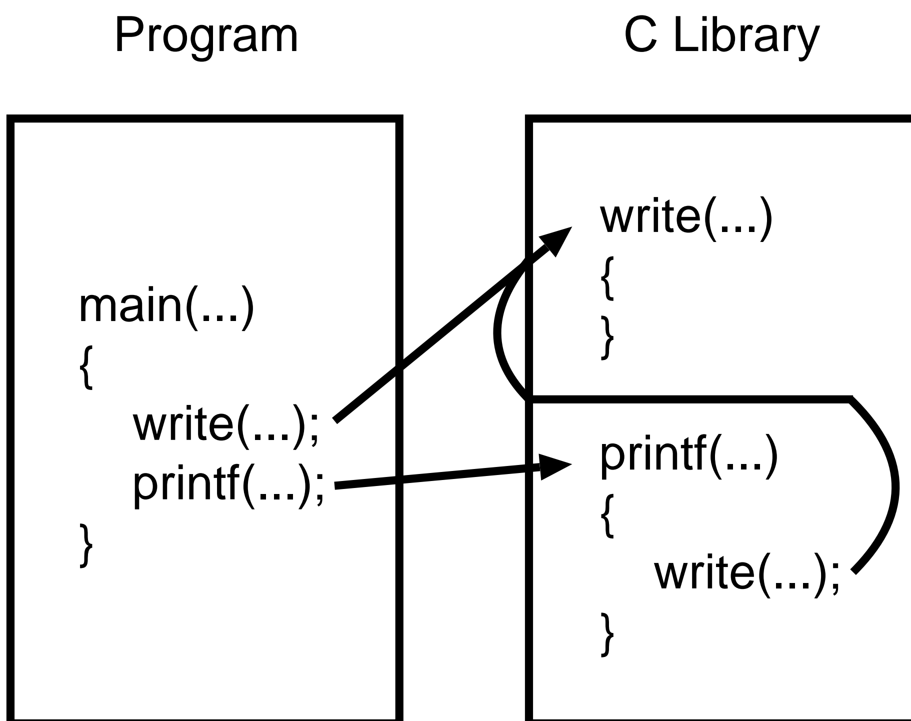

図 4.10　通常のシステムコール呼び出し

そこで、kexec システムは図 4.11 に示すように、インターポジショニングという手法を利用してシステムコールの置換を実現している。インターポジショニングとはライブラリ関数をユーザ定義関数によって置換することである。置換された関数は他のライブラリ関数からも呼び出されるようになる。kexec システムではこの性質を利用して、ソフトウェ

ア割り込み型のシステムコールを関数型のシステムコールに置換する。関数型のシステムコールは、アプリケーション用の追加モジュールとして実装されており、コンパイル時にリンクすることで置換を行う。

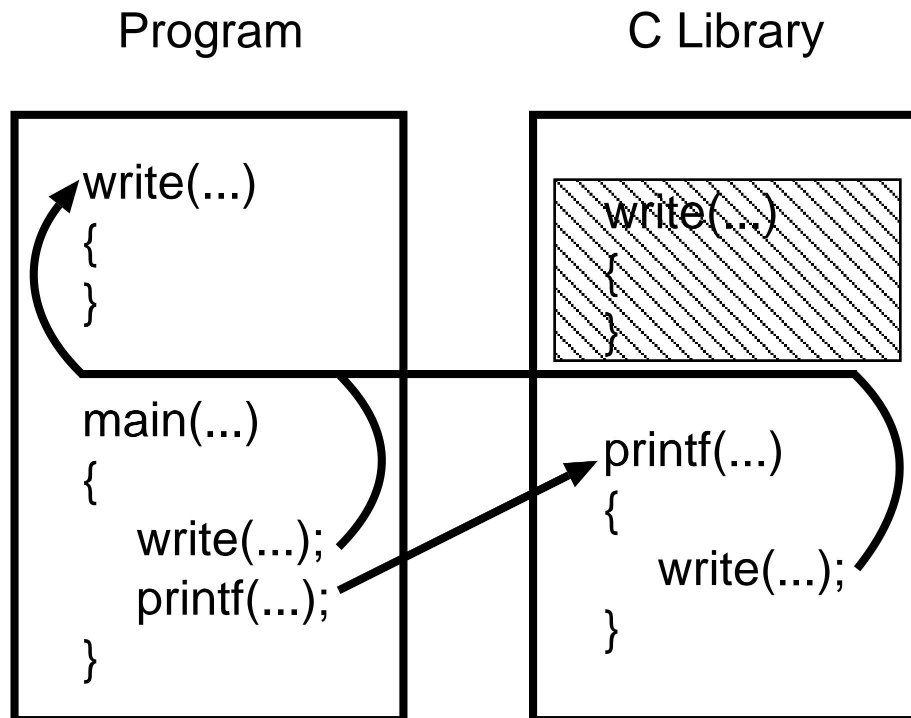

図 4.11 インターポジショニングが起きた場合

4.9 カーネル内シンボルの解決

カーネルモードで動作することにより、アプリケーションはカーネル内資源を直接操作することができる。その際、開発者はカーネル内資源を関数名や変数名といったシンボルを介して操作を行う。そのため、カーネル内シンボルの解決を行う機構が必要である。シンボルの解決とは、関数名や変数名といったプログラム中に使用されるシンボル名を、実際のメモリアドレスに結びつけることを言う。

kexec システムでは、リンカスクリプトを用いてカーネル内シンボルの解決を行う。リンカスクリプトとは、プログラムをリンクする際の制御用スクリプトであり、特定のシンボルにメモリアドレスを割り当てることができる。アプリケーションをリンクする際にリンカはこの仕組みを利用し、カーネル内のシンボルとそのシンボルに対応するメモリアドレスを知り、カーネル内シンボルの解決を行う。

リンカスクリプトは、図 4.12 に示すように、シンボルとメモリアドレスの組が記述さ

れたテキストファイルである。カーネル内のシンボルは約 6000 個と数が多い。しかし、リンカスクリプトの生成は、`nm` コマンドを用いてカーネルのオブジェクトファイルから、シンボルとアドレスの対応を取得し自動的に作成する。そのため、リンカスクリプトの生成にかかるコストは小さい。

```
...
...
...
bcopy = 0xc01003f0;
bdwrite = 0xc01cae2c;
bread = 0xc01caab8;
...
callout_init = 0xc01a4d18;
curproc = 0xc04c4500;
...
...
kern_printf = 0xc01b6ecc;
...
...
...
```

図 4.12　リンカスクリプトの内容

4.10　スタックの切り替え

スタックは関数の呼び出しにおいて、リターン・アドレスや関数の引数、局所変数を格納する領域として使われる。通常のアプリケーションプログラムは、ユーザのメモリ空間にあるスタックを使って動作している。

それに対し、カーネルモードで動作するアプリケーションプログラムは、カーネルモ

ジュールからコールバックされるため、スタックポインタがカーネルメモリ空間にあるスタックを指している。システムコールはユーザメモリ空間にデータがあるものとして動作するため、カーネルのメモリ空間上のスタックに扱うデータが存在するとエラーが発生してしまう。

そこで kexec システムではカーネルモード実行時に、スタックポインタをユーザのメモリ空間にあるスタックを指すように切替える。スタックのアドレスを取得、設定する操作は、リスト 4.6 に示すようにアセンブリ言語を用いて記述している。切替えるスタックのアドレスは `ioctl()` システムコールを用いて、kexec システムのカーネルモジュールに通達される。

```
1  #define GET_ESP(var) __asm__ volatile \
2                  ("movl %%esp,%0" : "=r" (var) : )
3  #define SET_ESP(var) __asm__ volatile \
4                  ("movl %0,%%esp" : : "r" (var) )
```

リスト 4.6 スタックのアドレスを取得、設定するマクロ

4.11 評価

本節では kexec システムの評価を行う。評価の内容は、まず kexec システムの利用例を挙げ、アプリケーションをどの程度容易にカーネルモードで動作させることができるのかを考察する。次に kexec システムによりどの程度システムコールオーバヘッドが削減されるかを測定する。最後にカーネル内資源を直接操作する例を挙げ、どの程度性能向上が果たせるか測定する。

評価を行う前に、評価環境と時刻の測定法を以下に挙げる。

評価環境

評価に用いた計算機環境を表 4.2 に示す。

表 4.2 評価に用いた計算機環境

OS	NetBSD–1.5.2
CPU	Intel Celeron 366MHz
主記憶	128 MB

時刻測定

時刻の測定は CPU 内部にある TSC[25, 26] を使用した。TSC は 1 クロックごとにインクリメントされる 64 bit の値である。TSC の値は数命令で読み込むことができ、ほぼ CPU のクロックと同じ精度で時刻測定が可能である。

4.11.1 運用の容易さの評価

ここでは、kexec システムの利用例を挙げ、アプリケーションをどの程度容易にカーネルモードで実行できるかを考察する。アプリケーションは、自分のプロセス ID を表示するという単純な内容とし、kexec システムの利用方法に焦点を当てる。

通常のアプリケーション

ここではユーザモードで動作する通常のアプリケーションを作成する。アプリケーションのソースはファイル名を `print_pid.c` として作成し、その内容はリスト 4.7 に示したようになる。ソースの内容は `getpid()` システムコールで自分のプロセス ID を取得し、`printf()` 関数で出力するというものである。

```
 1  #include <sys/types.h>
 2  #include <unistd.h>
 3  #include <stdio.h>
 4
 5  int
 6  main(int argc, char *argv[])
 7  {
 8      pid_t pid;
 9
10      pid = getpid();
11      printf("proc(%d): Hello, world.\n", pid);
12      return 0;
13  }
```

リスト 4.7 print_pid.c の内容

次に `print_pid.c` をリスト 4.8 に示すようにコンパイルする。実行ファイルは `print_pid` という名前で作成される。

```
1 % gcc -o print_pid print_pid.c
```

リスト 4.8 コンパイル方法（通常版）

作成した実行ファイルは、リスト 4.9 に示すように自分のプロセス ID を表示する。

```
1 % ./print_pid
2 proc(4680): Hello, world.
```

リスト 4.9 実行結果（通常版）

以上が、プロセス ID を表示するアプリケーションの作成から実行までの流れとなる。このアプリケーションは kexec システムを用いなくても動作する。

カーネルモードで動作するアプリケーション

ここでは kexec システムを用い、アプリケーションをカーネルモードで実行する方法を述べる。用いるアプリケーションは、先程作成した自分のプロセス ID を表示するものである。

アプリケーションのソースは先ほど示したリスト 4.7 とまったく同じである。このソースをリスト 4.10 に示すようにコンパイルする。通常のアプリケーションとの変更点は、エントリポイントを変更したことと、kexec システムのアプリケーション用の追加モジュールをリンクしたことである。エントリポイントの変更は kexec システムのカーネルモジュールを呼び出す初期化処理を行うためであり、追加モジュールのリンクはシステムコールを関数呼び出しにするために必要である。

```
1 % gcc -o print_pid print_pid.c -e _kexec_start libkexec.o
```

リスト 4.10 コンパイル方法（カーネルモード実行版）

kexec システムを利用するためには、カーネルモジュールをカーネルに組み込む必要がある。リスト 4.11 にカーネルモジュールの組み込み方法を示す。`kexec_mod.o` が kexec システムのカーネルモジュールであり、そのカーネルモジュールを現在動作しているカーネル`/netbsd` に組み込んでいる。カーネルモジュールの組み込みは、一度行えばアプリケーションの実行ごとに行う必要は無い。また、組込んだままでも通常の処理のセマンティクスは変わらない。

```
1 % modload -A/netbsd kexec_mod.o
```

リスト 4.11 カーネルモジュールの組み込み

カーネルモジュールを組み込み、先程作成したアプリケーションを実行する。実行結果をリスト 4.12 に示す。通常のアプリケーションと同じように動作していることがわかる。

```
1 % ./print_pid
2 proc(4681): Hello, world.
```

リスト 4.12 実行結果（カーネルモード実行版）

以上が、アプリケーションをカーネルモードで実行する方法である。ソースは通常のアプリケーションと同じ物が使用でき、再コンパイルを行うだけでカーネルモードで実行できる。

カーネル内資源を利用するアプリケーション

ここでは、カーネル内資源を直接操作する方法を述べる。作成するアプリケーションは、自分のプロセス ID を表示するもので、前節で使用していたアプリケーションを流用する。

アプリケーションのソース kern_print_pid.c をリスト 4.13 に示す。このソースはカーネル内資源を直接操作する変更が加えられている。まず、プログラムは自分のプロセス ID を出力する他に、カーネル用のメッセージバッファに出力する。メッセージバッファへの出力は、18 行目の kern_printf() カーネル内関数を使用し行っている。また、getpid() システムコールを使用しプロセス ID を取得していたのを、カーネル内変数の curproc を使用し取得するように変更した。curproc 変数には現在動作しているプロセスのコンテクストが格納されており、そこから現在動作しているプロセスのプロセス ID、つまり自分のプロセス ID を取得できる。

```
1  #define _KERNEL
2  #define _LKM
3  #include <sys/types.h>
4  #include <sys/param.h>
5  #include <sys/proc.h>
6  #undef _LKM
7  #undef _KERNEL
8
9  #include <stdio.h>
10
11 int
```

```
12 main(int argc, char *argv[])
13 {
14     pid_t pid;
15
16     pid = curproc->p_pid;
17     printf("proc(%d): Hello, user.\n", pid);
18     kern_printf("proc(%d): Hello, kernel.\n", pid);
19     return 0;
20 }
```

リスト 4.13　kern_print_pid.c の内容

カーネル内の関数や変数を使用したこれらの変更は、通常のアプリケーションが関数を呼び出したり、変数を参照したりするのと同様に記述できる。また、ユーザ側のライブラリに含まれる printf() 関数も同時に使用できる。これは通常のアプリケーションや、カーネルモジュールでは不可能な記述である。

変更したソースをリスト 4.14 に示すようにコンパイルする。カーネル内シンボルの解決をするため、リンカスクリプト KERN_SYM をコンパイラオプションに追加する。KERN_SYM には curproc や kern_printf といったカーネル内のシンボルと、そのメモリアドレスが記述されている。

```
1 % gcc -o kern_print_pid kern_print_pid.c -e _kexec_start libkexec.o -Wl
      ,-RKERN_SYM
```

リスト 4.14　コンパイル方法（カーネル内資源利用版）

作成したアプリケーションの実行結果をリスト 4.15 に示す。ユーザライブラリの printf() 関数の出力とカーネル内関数の kern_printf() の出力が確認できる。kern_printf() 関数の出力内容は、メッセージバッファの内容を出力する dmesg コマンドを使用した。

```
1 % ./kern_print_pid
2 proc(4682): Hello, user.
3 % dmesg
4 ...
5 proc(4682): Hello, kernel.
```

```
6 ...
```

リスト 4.15 実行結果（カーネル内資源利用版）

考察

kexec システムの利用法を、簡単なアプリケーションを用いた例で紹介した。まず、通常のアプリケーションを用意し、次にそれをカーネルモードで実行する方法を示した。そして最後に、カーネル内資源を直接扱う処理を追加する方法を示した。

カーネルモードで実行するアプリケーションは、ソースに変更を加えること無く、再コンパイルするだけで作成できる。また、カーネル内資源を扱う操作をソースを変更することだけで追加できる。その際、ユーザ側のライブラリも同時に使用することが可能である。これは通常のアプリケーションや、カーネルモジュールでは不可能である。

これらのことより、アプリケーションが行うサービスをカーネル内で提供する場合、カーネルモジュールと比較して kexec システムを利用した方が、ソースを流用できる分、実装が容易であることがわかる。開発者は kexec システムを用いることで、計算機資源を効率的に利用したい部分や、機能を拡張したい部分を集中的に実装することができる。

4.11.2 システムコールオーバヘッドの削減

kexec システムはカーネルモードで動作することを利用し、オーバヘッドの少ない関数呼び出し型のシステムコールを利用している。ここでは、システムコールの処理にかかる時間を測定し、どの程度オーバヘッドが削減できているかを定量的に示す。

測定

比較対象は、通常のアプリケーションで使用されるソフトウェア割り込み型のシステムコールの処理時間と、kexec システムによる関数呼び出し型のシステムコールの処理時間である。処理時間は、各システムコールの処理時間を 100 万回測定しその平均クロック数として算出する。以下のシステムコールをそれぞれ測定した。

- getpid
  呼び出したプロセスのプロセス ID を返す。
- chdir
  呼び出したプロセスのカレントディレクトリを変更する。
- gettimeofday
  現在の時刻を得る。

- read–1
  ローカルファイルよりデータを 1byte 読み込む。
- read–8192
  ローカルファイルよりデータを 8192byte 読み込む。
- write–1
  ローカルファイルへデータを 1byte 書き込む。
- write–8192
  ローカルファイルへデータを 8192byte 書き込む。

結果

図 4.13 に `getpid()` システムコールの処理時間のグラフを示す。通常のソフトウェア割り込み型の `getpid()` システムコールに比べ、kexec システムの関数型の `getpid()` システムコールの処理時間は半分以下に短縮されている。`sys_getpid()` 関数は各 `getpid()` システムコールから共通に呼び出されるカーネル内のシステムコール処理関数となっており、この関数の処理時間は両方の `getpid` システムコールで共通である。短縮された処理時間は `getpid()` システムコールの呼び出しと復帰の部分の処理時間である。

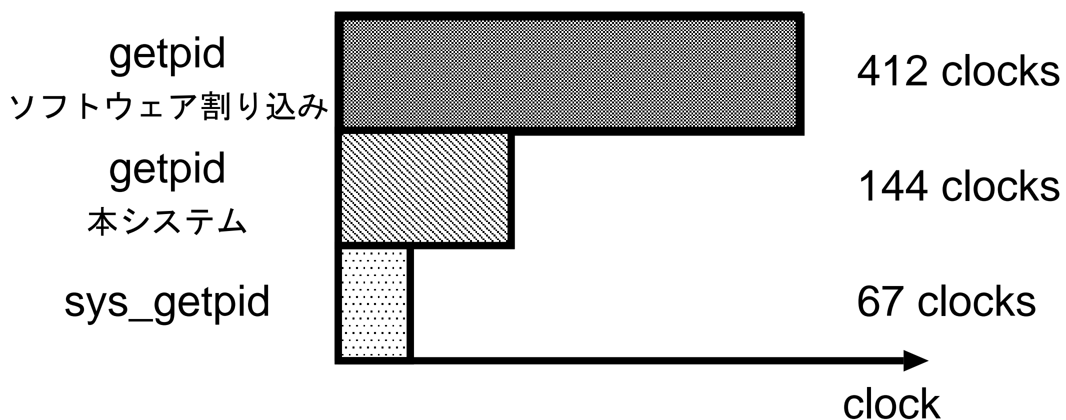

図 4.13 `getpid()` システムコールの処理時間

表 4.3 に kexec システムを使用することで、各システムコールがどの程度短縮されるかを示す。通常のシステムコールに比較して、kexec システムのシステムコールのオーバヘッドが削減されていることがわかる。ただし、kexec システムで削減したシステムコールの処理時間は、システムコールの呼び出し部分と復帰部分である。そのため、カーネル内のシステムコール処理関数の処理量が増加するとシステムコールの処理時間比は増加してしまう。

表 4.3　システムコールの処理時間の比較

	本手法 (clock)	ソフトウェア割り込み (clock)	処理時間比
getpid	144	412	35 %
chdir	794	1184	67 %
gettimeofday	1626	1967	83 %
read–1	1306	1751	75 %
read–8192	5664	6105	93 %
write–1	1652	2149	77 %
write–8192	5609	6886	81 %

以上の測定により kexec システムを使用することでシステムコールを高速化できることが確認できた。システムコールを頻繁に利用するアプリケーションは、kexec システムを使うことで高速に動作することが期待できる。

4.11.3　ファイルコピーの高速化

ここでは、カーネル内資源を直接操作する例としてファイルコピーの高速化を取り上げる。比較対象は、通常の `read()` と `write()` システムコールを使用するファイルコピーと、カーネル内資源であるバッファキャッシュを直接操作する高速化したファイルコピーである。

バッファキャッシュについて

バッファキャッシュは、ディスク上のファイル内容をメモリ上に保持する仕組みである。ディスクの入出力操作は遅い処理であるため、バッファキャッシュにファイルの内容を保持することで、ファイル操作を効率的に行う。

通常の `read()` と `write()` システムコールを使用するファイルコピーは、図 4.14 に示すように、コピー元のバッファキャッシュの内容をユーザのメモリ空間にあるバッファへコピーし、そのデータをコピー先のバッファキャッシュに書き込む。この間にデータの変更は無いため、ユーザのメモリ空間にあるバッファへのコピーは無駄である。しかし、`read()` と `write()` システムコールを使用する限り、ユーザのメモリ空間にあるバッファへのコピーは省くことができない。

そこで kexec システムを使用して、図 4.15 に示すようにバッファキャッシュ間の直接コピーを実現する。ユーザのメモリ空間にあるバッファへのコピーを省略し、ファイルコピーの高速化を実現する。

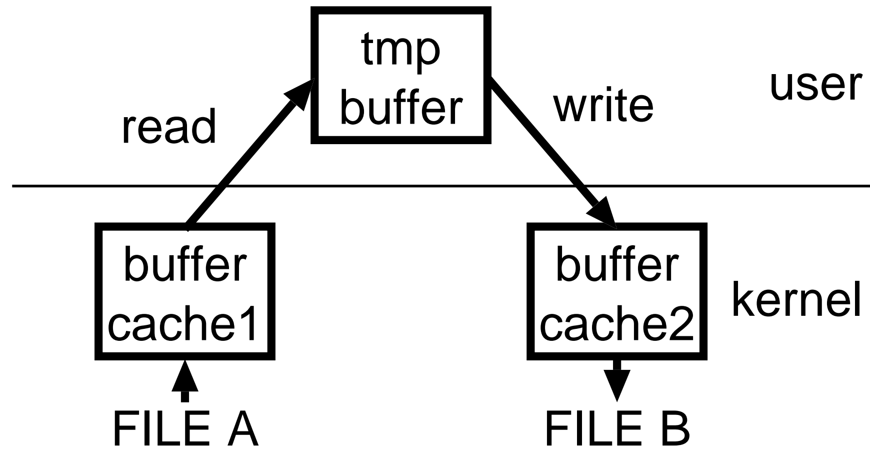

図 4.14　`read()` と `write()` システムコールを用いるファイルコピー

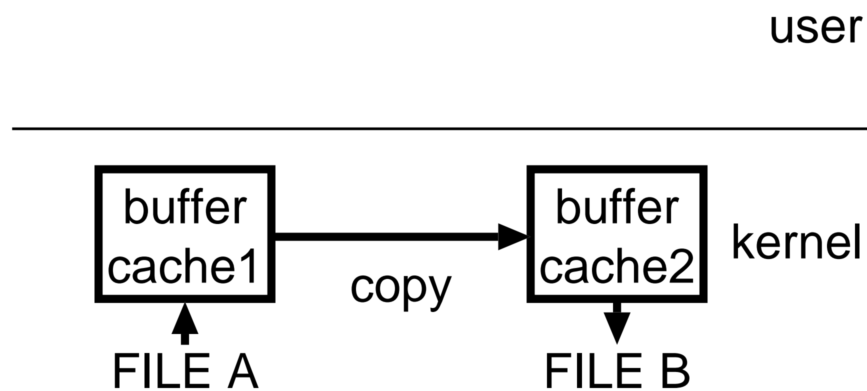

図 4.15　バッファキャッシュ間の直接コピー

リスト 4.16 にバッファ間の直接コピーを行う関数のソースを示す。バッファキャッシュはカーネル資源であり、この関数を実行するためには、kexec システムを用いてアプリケーションをカーネルモードで実行する必要がある。

```
 1  int
 2  kern_copy_file(int ifd, int ofd)
 3  {
 4      int error = 0;
 5      struct proc *p = curproc;
 6      struct filedesc *fdp = p->p_fd;
 7
 8      struct file *ifp, *ofp;
 9      struct vnode *ivp, *ovp;
10      struct inode *iip;
11      struct buf *ibp, *obp;
12
13      long ixfersize;
14      off_t ifile_offset = 0, ofile_offset = 0;
15
16      /*
17       * file 構造体
18       */
19      if ((ifd >= fdp->fd_nfiles) ||
20          (ifp = fdp->fd_ofiles[ifd]) == NULL ||
21          (ifp->f_iflags & FIF_WANTCLOSE) != 0 ||
22          (ifp->f_flag & FREAD) == 0) {
23          return EBADF;
24      }
25      if ((ofd >= fdp->fd_nfiles) ||
26          (ofp = fdp->fd_ofiles[ofd]) == NULL ||
27          (ofp->f_iflags & FIF_WANTCLOSE) != 0 ||
28          (ofp->f_flag & FWRITE) == 0) {
29          return EBADF;
30      }
31      FILE_USE(ifp);
```

```
32      FILE_USE(ofp);
33
34      /*
35       * vnode 構造体
36       */
37      ivp = (struct vnode *)ifp->f_data;
38      VOP_LEASE(ivp, p, ifp->f_cred, LEASE_READ);
39      vn_lock(ivp, LK_EXCLUSIVE | LK_RETRY);
40
41      ovp = (struct vnode *)ofp->f_data;
42      VOP_LEASE(ovp, p, ofp->f_cred, LEASE_READ);
43      vn_lock(ovp, LK_EXCLUSIVE | LK_RETRY);
44
45      /*
46       * inode 構造体
47       */
48      iip = VTOI(ivp);
49
50      /*
51       * ファイルチェック
52       */
53      if ((ivp->v_type != VREG) || (ivp->v_tag != VT_UFS)) {
54          error = EBADF;
55          goto cleanup;
56      }
57      if ((ovp->v_type != VREG) || (ovp->v_tag != VT_UFS)) {
58          error = EBADF;
59          goto cleanup;
60      }
61
62      for (ibp = NULL; ; ibp = NULL) {
```

```
63         /* read 側のバッファキャッシュの確保 */
64         ixfersize = read_buf(ifile_offset, ivp, &ibp);
65         if (ixfersize <= 0) {
66             break;
67         }
68         ifile_offset += ixfersize;
69 
70         /* write 側のバッファキャッシュの確保 */
71         if (get_new_buf(ofile_offset, ovp, &obp, ixfersize, ofp->f_cred)
                < 0) {
72             error = ENOBUFS;
73             break;
74         }
75 
76         /* バッファキャッシュ間でのファイル内容コピー */
77         kcopy((char *)ibp->b_data, (char *)obp->b_data, ixfersize);
78 
79         write_buf(ovp, &obp, ixfersize);
80         ofile_offset += ixfersize;
81         brelse(ibp);
82     }
83 
84     if (ibp != NULL) {
85         brelse(ibp);
86     }
87     if (!(ivp->v_mount->mnt_flag & MNT_NOATIME)) {
88         iip->i_flag |= IN_ACCESS;
89     }
90 
91     /*
92      * 後処理
```

```
 93      */
 94 cleanup:
 95     VOP_UNLOCK(ivp, 0);
 96     VOP_UNLOCK(ovp, 0);
 97     FILE_UNUSE(ifp, p);
 98     FILE_UNUSE(ofp, p);
 99     return error;
100 }
```

リスト 4.16 バッファキャッシュ間でデータコピーをする関数

測定

ファイルコピーの速度を測定する。比較対象は、通常の read() と write() システムコールを使用するファイルコピーと、kexec システムを使用しバッファキャッシュを直接操作することによって高速化したファイルコピーである。

測定はサイズの異なる複数のファイルに対し行った。ファイルコピーの速度は、各ファイルのコピーにかかる時間を 100 回測定し、その平均から算出した。

結果

測定結果を図 4.16 に示す。通常の read() と write() システムコールを使用したファイルコピーに比べ、kexec システムを使用しバッファキャッシュを直接操作するファイルコピーの処理速度の方が速い。ファイルサイズが 100KB 以上では両方のファイルコピー速度が低下している。これはファイルサイズの増加に伴い、バッファキャッシュにファイルの内容が収まらなくなり、ディスク I/O が発生するため処理速度が低下すると考えられる。

以上の結果から、カーネル内資源を直接操作することでファイルコピーの高速化が可能であることを確認した。kexec システムを使い、アプリケーションの一部を変更することで、高速化やカーネル特有の機能を追加することが可能になる。

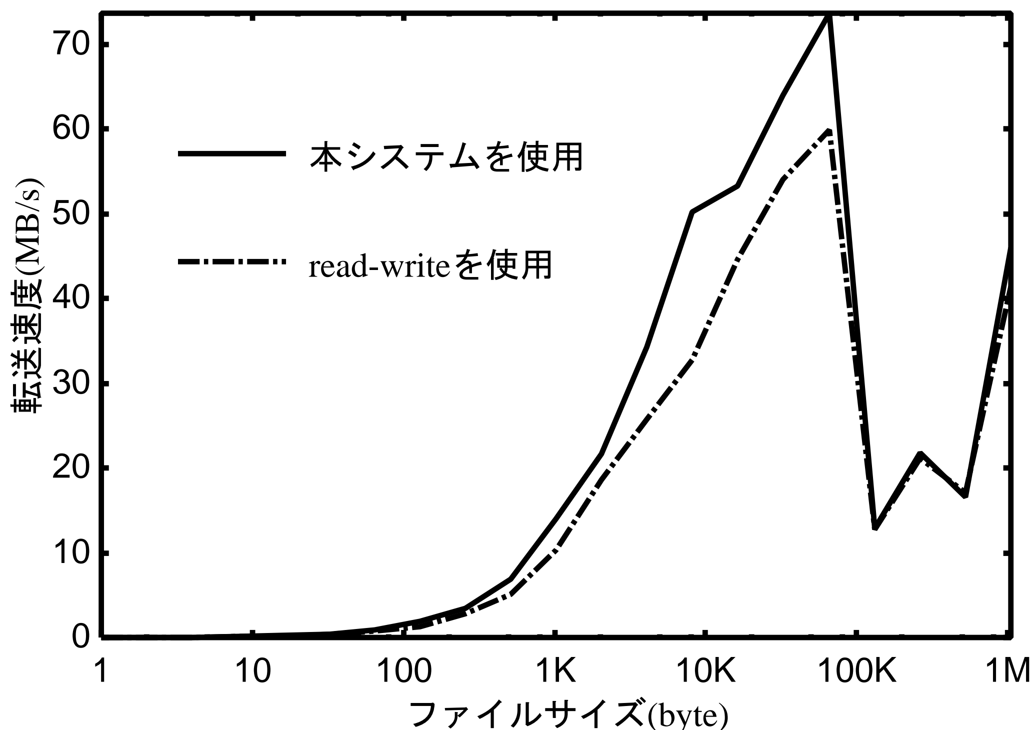

図 4.16 ファイルコピー速度の比較

4.12 関連研究

4.12.1 LKM を用いたアプリケーション実装

カーネルモジュールとしてアプリケーションのサービスを実現する手法である。カーネル内で動作するため、モード遷移のオーバヘッドが必要無く、更に、カーネル内資源を直接操作して処理内容に特化した改良を加えることが可能である。

例えば Linux の kHTTPd[21] はカーネル内で Web サービスを行うカーネルモジュールである。kHTTPd はモード遷移を必要としないことに加え、カーネル内の低レベル操作を使いデータコピーの少ないファイル転送を実装しているため、ユーザランドで動作する Web サーバよりも高い性能を実現している。

kexec システムと比較して、カーネルモジュールはゼロから作成するため、OS よりに実装され、高い性能を発揮することが予想される。しかし、kexec システムでは既存 アプリケーション を利用できるため、開発効率の面で有利である。

4.12.2 特殊システムコール

ファイル転送を効率的に行う sendfile のように、ある特定の処理に特化したシステムコールを アプリケーション に提供する手法である。

カーネルウェアでは、システムコールを使う以外に、カーネル内資源を直接操作する改良を処理内容に応じて柔軟に加えることができる。

4.12.3 専用 OS

Exokernel[20] は計算機資源管理ポリシーをアプリケーション側に任せることが可能な OS である。アプリケーション側で計算機資源管理を行うことで、オーバヘッドが少ない計算機資源利用が可能である。

SPIN[27] は アプリケーション の一部のコードをカーネル内に取り込み、実行することができる OS である。カーネル内でコードを実行することでモード遷移を抑え、オーバヘッドの少ない資源利用を行える。カーネル内実行コード記述に Modula–3 を使うことで安全性を高めている。

専用 OS を用いて効率的な I/O 操作をアプリケーションに提供する手法に対し、kexec システムは汎用 OS 上に実装した。汎用 OS を使用することで、動作実績のあるアプリケーションを豊富に利用できる。

4.12.4 カーネル内実行アプリケーション用フレームワーク

Kernel Mode Linux[28] は アプリケーション をカーネル内実行する機構である。実行コードは型検査されるため、安全性を確保することができる。

Cosy[29] も Kernel Mode Linux と同様、アプリケーション をカーネル内実行する機構である。実行時にコードの安全性を検査する。また、システムコールの内部処理を改良することで、コピー回数の削減などを実現している。

kexec システムでは、カーネル内資源を直接操作できる枠組を用意することで、カーネルウェアの処理内容に最適化した I/O 処理手段を提供している点が異なる。

4.13 まとめ

アプリケーションは計算機資源利用にオーバヘッドを伴うという問題があり、カーネルモジュールは開発効率が悪いという問題があった。そこで本研究では、これらの問題を解決するためアプリケーションをカーネルモードで実行する手法を提案した。アプリケー

ションはカーネルモードで動作することにより、カーネルモジュールと同様に低レベルな計算機資源利用が可能になる。また、kexec システムは通常のアプリケーションが使用するインタフェイスとセマンティクスを提供しており、アプリケーションのソースに変更を加えること無く、カーネルの一部として実行できる。

我々は、kexec システムを NetBSD 上に実装して評価を行った。その結果、アプリケーションのソースに変更を加えること無く、カーネルモードで実行できることを確認した。また、システムコールをソフトウェア割り込みから関数呼び出しにすることで、処理時間を削減できた。更に、カーネル内資源を直接操作することで、効率的な計算機資源利用が可能になることを示した。

第 5 章

まとめ

本章では、本研究について総括する。情報家電などの組込み機器の高性能化に伴い、豊富なソフトウェア資産を利用できる UNIX への注目が高まってきた。組込み機器向けの基盤ソフトウェアに UNIX を利用する理由は、μITRON のような既存の組込み OS に、デバイスドライバやネットワークスタック、アプリケーションプログラムなどのソフトウェア資産が不足しているためである。

既存の組込み OS で、上記に挙げたような用途に対応するためには、自前で必要なソフトウェアを開発するか、既に存在するサードパーティー製のソフトウェアのロイヤリティを購入する必要があった。これは開発にかかる時間的、金銭的コストの増加を意味する。一方、UNIX は、豊富なソフトウェア資産を持ち、多くの UNIX 実装はロイヤリティフリーかつオープンソースである。UNIX を組込み機器向けの基盤ソフトウェアとして利用でき、これらの豊富なソフトウェア資産を利用、改良することで、既存の組込み OS が抱えていた開発コストの増加を抑えられる。

しかし、現状の UNIX は、汎用 OS として発展してきた背景から、組込み機器の制御に適さない面もある。本研究では、「実時間性の不足」と「用途に特化する柔軟性の不足」という問題点に着目し、unitron システムと kexec システムを提案した。unitron システムは、実時間 OS である TOPPERS をカーネルモジュール化し、NetBSD に組込むことで実時間性の不足を補う。kexec システムは、アプリケーションをカーネルモードで実行する枠組みを NetBSD に持たせた。カーネルモードで実行されるアプリケーションはカーネル内資源を直接操作する変更が可能であるため、組込みシステムの用途に合わせ特化する柔軟性を持つことができる。

unitron システムにおいて、開発者は実時間処理を μITRON のタスクとして記述し、実時間性が必要ない処理を UNIX のプロセスとして記述する。このように、実時間処理を実現しつつ、UNIX が持つ豊富なソフトウェア資産を流用できることが利点である。μITRON の機能は LKM を利用し UNIX に取り込んでいるため、動的に機能の取り付

け、取り外しができる。

kexec システムにより実行される UNIX アプリケーションをカーネルウェアと呼ぶ。カーネルモードで動作するカーネルウェアは、UNIX カーネル内資源を直接操作する柔軟性を持つ。カーネルウェアは UNIX アプリケーションを流用して作成できるため、初めからカーネルモジュールを作成する場合に比べ開発のコストを抑えられる。組込み機器は用途が限定されるため、その用途に特化したシステムが求められる。kexec システムを使用することで、用途に応じたアプリケーションを流用しつつ、そのソースへカーネル内資源を直接操作する変更を加え、用途に特化できる。

本研究では、unitron システムと kexec システムとを別個に実装、評価した。今後、両システムのインタフェースを洗練し、両システムの連携をはかりながら、更に UNIX を組込み機器の制御に適した基盤ソフトウェアとする事を目指す。

参考文献

[1] SHARP 社. 液晶テレビ aquos サポートステーション | ソースコード公開：シャープ. http://www.sharp.co.jp/support/aquos/source/download/index.html. 2012 年 10 月 1 日閲覧.

[2] SPEECYS 社. スピーシーズは、動くインターネットエンタテインメントを創造する会社です。http://www.speecys.com. 2012 年 10 月 1 日閲覧.

[3] IIJ 社. Seil.jp. http://www.seil.jp. 2012 年 10 月 1 日閲覧.

[4] BRAINS 社. 株式会社ブレインズ. http://www.brains.co.jp/. 2012 年 10 月 1 日閲覧.

[5] Denys Vlasenko. Busybox. http://www.busybox.net. 2012 年 10 月 1 日閲覧.

[6] Erik Andersen. uclibc. http://uclibc.org. 2012 年 10 月 1 日閲覧.

[7] Matt Mackall. Linux tiny. http://elinux.org/Linux_Tiny. 2012 年 10 月 1 日閲覧.

[8] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pp. 29–42, Berkeley, CA, USA, 2001. USENIX Association.

[9] David Woodhouse. Jffs : The journalling flash file system. In *Proceedings of the 2001 Ottawa Linux Symposium*, 2001. http://sourceware.org/jffs2/jffs2-html/.

[10] 佐藤喬, 多田好克. uitron のカーネルモジュール化による組込み用 unix への融合. 情報処理学会第 51 回プログラミング・シンポジウム予稿集, pp. 89–94, 2010.

[11] Takashi Sato and Yoshikatsu Tada. Unitron: Loadable kernel module for adding real-time functionality of uitron to unix kernel. In *CSA 2013 : 1st International Workshop on Computer Systems and Architectures*, pp. 248–256, 2013.

[12] 佐藤喬, 安田絹子, 中村嘉志, 多田好克. カーネルウェア：アプリケーションのカーネル内実行による os 機能拡張法の提案. 情報処理学会論文誌：コンピューティングシ

ステム, Vol. 45, No. SIG11(ACS7), pp. 248–256, 2004.

[13] 保田信長, 飯山真一, 冨山宏之, 高田広章, 中島浩. Linux と itron によるハイブリッド os の設計と実装. 情報処理学会研究報告 (SLDM), Vol. 2004, No. 33, pp. 45–50, 2004.

[14] 多田好克, 福田伸彦, 鈴鹿倫之, 中村嘉志. カーネルの外部で走行するカーネルスレッドの提案とその実装法. 電子情報通信学会論文誌, Vol. J85-D-I, No. 3, pp. 286–293, 2002.

[15] Michael Barabanov and Victor Yodaiken. Introducing real-time linux. *Linux Journal*, Vol. 34, pp. 19–23, 1997.

[16] 加賀美聡, 石綿陽一, 西脇光一, 梶田秀司, 金広文男, 尹祐根, 安藤慶昭, 佐々木洋子, サイモントンプソン, 松井俊浩. Art-linux の複数コア利用機能によるソフトウェアディペンダビリティー向上. 第 12 回計測自動制御学会システムインテグレーション部門講演会論文集, pp. 728–731.

[17] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pp. 164–177, New York, NY, USA, 2003. ACM.

[18] Real-Time Systems GmbH. Real-time systems - embedded hypervisor for intel x86 multicore. `http://www.real-time-systems.com/real-time_hypervisor`. 2012 年 10 月 1 日閲覧.

[19] Daniel Sangorri'n, Shinya Honda, and Hiroaki Takada. Reliable and efficient dual-os communications for real-time embedded virtualization. *Information and Media Technologies*, Vol. 8, No. 1, pp. 1–17, 2013.

[20] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pp. 251–266, New York, NY, USA, 1995. ACM.

[21] Arjan van de Ven. khttpd - linux http accelerator. `http://www.fenrus.demon.nl`. 2012 年 10 月 1 日閲覧.

[22] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. ADDISON WESLEY, 1996.

[23] Matt Welsh, Anindya Basu, and Thorsten von Eicken. Atm and fast ethernet network interfaces for user-level communication. In *In Proceedings of The Third International Symposium on High Performance Computer Architecture*, pp. 332–

342, 1997.

[24] 鈴鹿倫之, 中村嘉志, 多田好克. カーネル拡張のための効率的な開発環境. 情報処理学会第 41 回プログラミング・シンポジウム報告集, pp. 57–64, 2000.

[25] インテル社. インテル・アーキテクチャ・ソフトウェア・ディベロッパーズ・マニュアル上巻: 基本アーキテクチャ.

[26] インテル社. インテル・アーキテクチャ・ソフトウェア・ディベロッパーズ・マニュアル中巻: 命令セット・リファレンス.

[27] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin G. Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility safety and performance in the spin operating system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP ’95, pp. 267–283, New York, NY, USA, 1995. ACM.

[28] 前田俊行, 住井英二郎, 米澤明憲. Linux/tal: 型付きアセンブリプログラムのカーネルモード実行方式. 日本ソフトウェア科学会第 4 回プログラミングおよびプログラミング言語ワークショップ論文集, pp. 62–73, 2002.

[29] Amit Purohit, Charles P. Wright, Joseph Spadavecchia, and Erez Zadok. Cosy: develop in user-land, run in kernel-mode. In *Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9*, HOTOS’03, pp. 109–114, Berkeley, CA, USA, 2003. USENIX Association.

謝辞

本研究をおこなうにあたり、多くの方からご理解、ご支援、ご指導を頂きました。お世話になったすべての方に深く感謝の意を表します。

まず、指導教員としてご指導頂きました多田好克教授、小宮常康准教授、大森匡教授に感謝します。いくつかの転機において、先生方からのご指導がなければ研究を進めることはできませんでした。特に多田好克教授にはソフトウェア生産管理学講座（現：基盤ソフトウェア学講座）へ博士前期課程で配属された時からご指導頂きました。ソフトウェアの作成法や研究に対する考え方など、研究者としての基礎部分を学ぶことができました。中村嘉志助手（当時)、安田絹子助手（当時）に感謝します。研究の方向性や論文の構成などで多くのご助言とご指導を頂きました。本論文の審査をして頂きました末廣尚士教授、田野俊一教授、古賀久志准教授に感謝します。先生方の専門的な見地からのコメントを頂くことで、多くの気づきを得ることができました。

最後に、学位取得に理解を示し、励まし、そして自由にさせてくれた父 和美、母 葉子に心から感謝します。

関連論文の印刷公表の方法及び時期

1. 全著者名：Takashi Sato, Yoshikatsu Tada
   論文題目：Unitron: Loadable Kernel Module for Adding Real-Time Functionality of uITRON to UNIX Kernel
   印刷公表の方法及び時期：1st International Workshop on Computer Systems and Architectures (CSA'13), pp.373–377, 2013
   (第 4 章の内容)
2. 全著者名：佐藤喬, 安田絹子, 中村嘉志, 多田好克
   論文題目：カーネルウェア：アプリケーションのカーネル内実行による OS 機能拡張法の提案
   印刷公表の方法及び時期：情報処理学会論文誌：コンピューティングシステム, Vol. 45, No. SIG11(ACS7), pp.248–256, 2004
   (第 4 章の内容)
3. 全著者名：Takashi Sato, Yoshikatsu Tada
   論文題目：Execution system for user programs in kernel mode
   印刷公表の方法及び時期：Journal of Communication and Computer, Vol. 10, No. 10, pp.1307-1311. 2013
   (第 3 章の内容)

著者略歴

佐藤　喬（さとう　たかし）

1977 年 4 月　山形県に生まれる

1996 年 4 月　電気通信大学 電気通信学部 情報工学科 入学

2000 年 3 月　電気通信大学 電気通信学部 情報工学科 卒業

2000 年 4 月　電気通信大学 大学院情報システム学研究科 情報システム設計学専攻 修士 入学

2002 年 3 月　電気通信大学 大学院情報システム学研究科 情報システム設計学専攻 修士 修了

2002 年 4 月　電気通信大学 大学院情報システム学研究科 情報システム設計学専攻 博士 入学

2004 年 5 月　電気通信大学 大学院情報システム学研究科 情報システム設計学専攻 博士 退学

2004 年 6 月　電気通信大学 大学院情報システム学研究科 助手 着任

2007 年 4 月　電気通信大学 大学院情報システム学研究科 助教 着任

2011 年 4 月　電気通信大学 大学院情報システム学研究科 情報システム基盤学専攻 博士 入学

2011 年 4 月　電気通信大学 大学院情報システム学研究科 情報システム基盤学専攻 博士 休学

2011 年 5 月　電気通信大学 大学院情報システム学研究科 助教 退任

2011 年 6 月　電気通信大学 大学院情報システム学研究科 情報システム基盤学専攻 博士 復学

2013 年 4 月　東京都立産業技術高等専門学校 ものづくり工学科 准教授 着任

2014 年 3 月　電気通信大学 大学院情報システム学研究科 情報システム基盤学専攻 博士 修了予定