平成13年度 修士論文

アプリケーションプログラムを流用するカーネル機能拡張法

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

0050013　佐藤 喬

指導教官　多田 好克　助教授
大森 匡　　助教授
岡本 敏雄　教授

提出日　平成14年2月1日

目 次

図 目 次

表 目 次

第 1 章

背景と目的

1.1 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 [2]。

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[1] 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。

1.2 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 [8] 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。また本システムは、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。

第 2 章

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。本章では、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。

2.1 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 2.1 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。

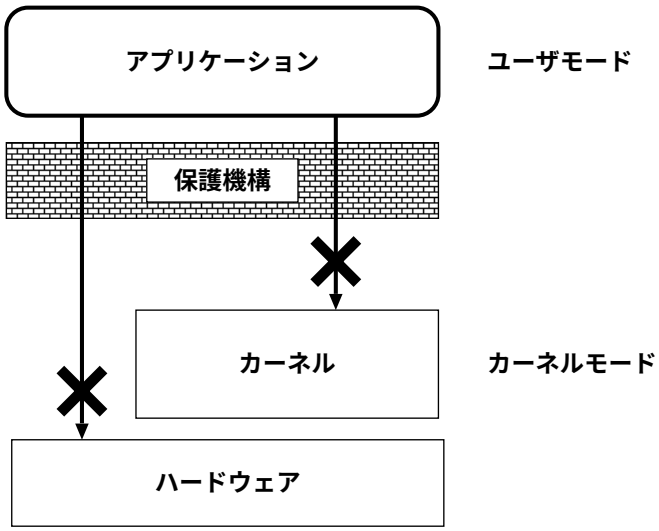

図 2.1: 保護機構の概略

2.1.1 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 2.2 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 [11]。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

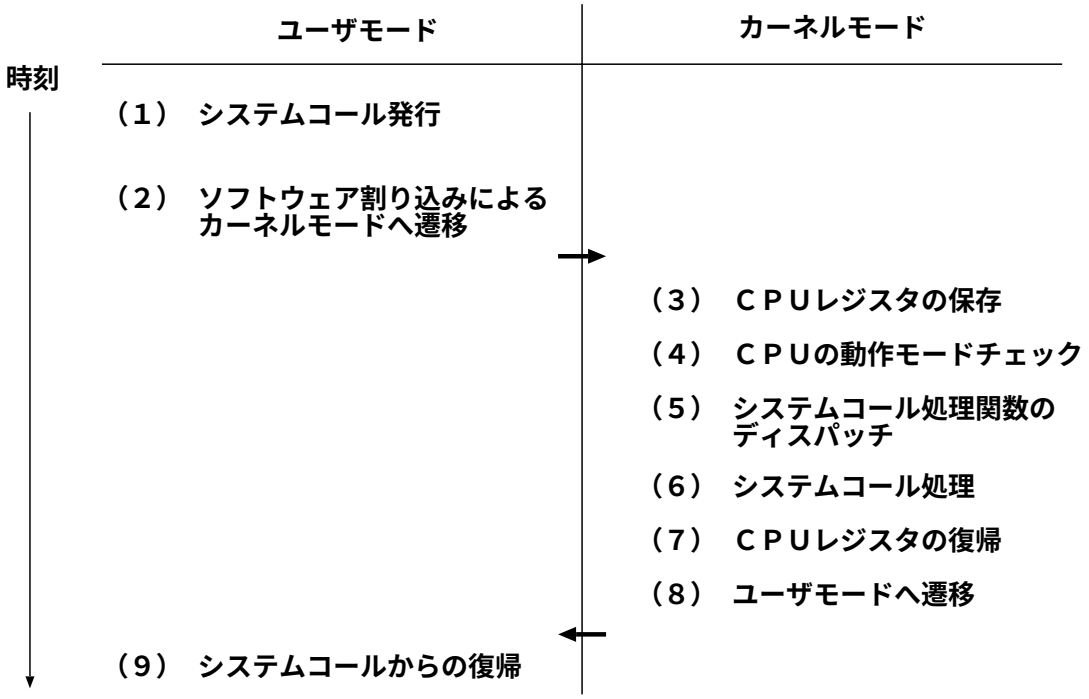

図 2.2: システムコール処理の流れ

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。

- 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 [5]。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 2.3 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(ネットワークインターフェイスカード) 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。

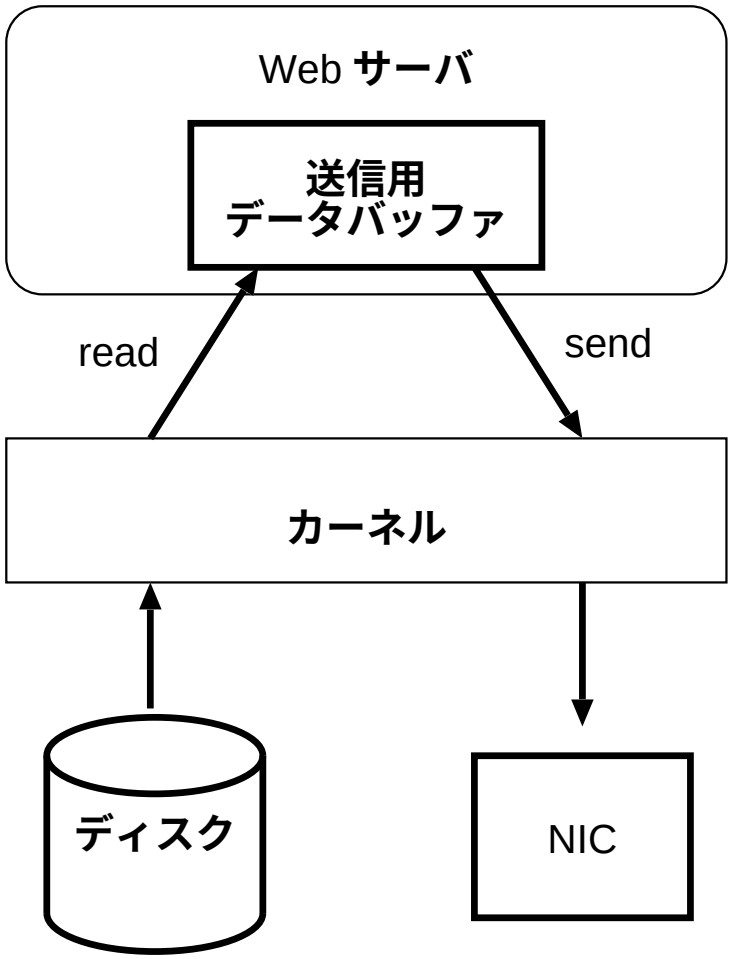

図 2.3: Web サーバのファイル転送

2.2 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。

2.2.1 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 [7]。

2.3 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 [2]。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 [1]。

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、本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。本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、ある開発者が本システムを用いて、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. 本システムを用いて 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、本システムはより容易な実装環境を提供する。

第 3 章

システムの概要

2 章の既存手法の問題で述べたように、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。本章ではこのシステムの概要について述べる。

3.1 本システムの概要

まず、本システムの概要を述べる。本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 3.1 のように本システムと組み合わせてカーネルモードで実行する。本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。

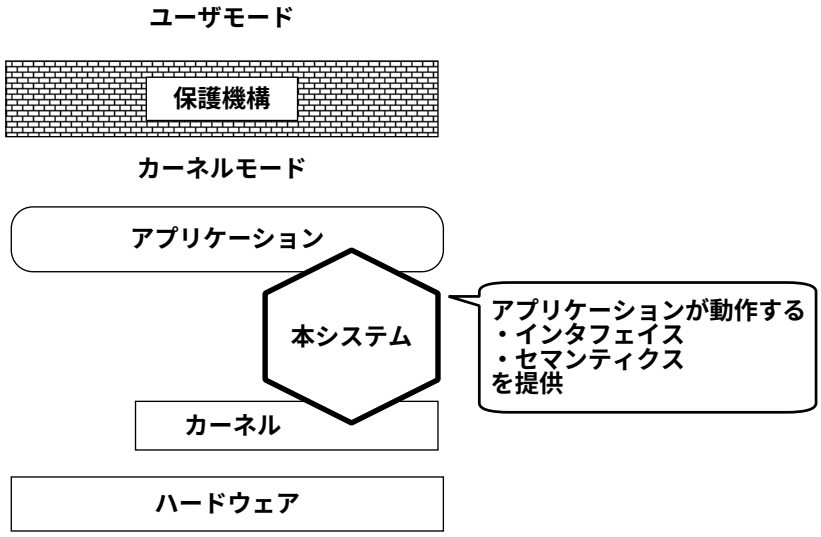

図 3.1: アプリケーションのカーネルモード実行

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 3.2 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。

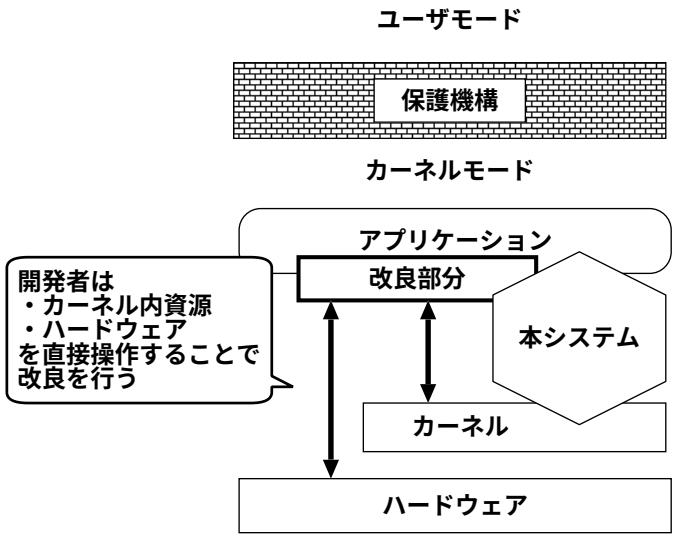

図 3.2: カーネル内資源の直接操作

本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 3.1 に示す。また、本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 3.3 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。

表 3.1: システムの比較

	カーネル モジュール	本システム	ユーザ プログラム
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	×	△	○

*現在未実装 (cf. p.38)

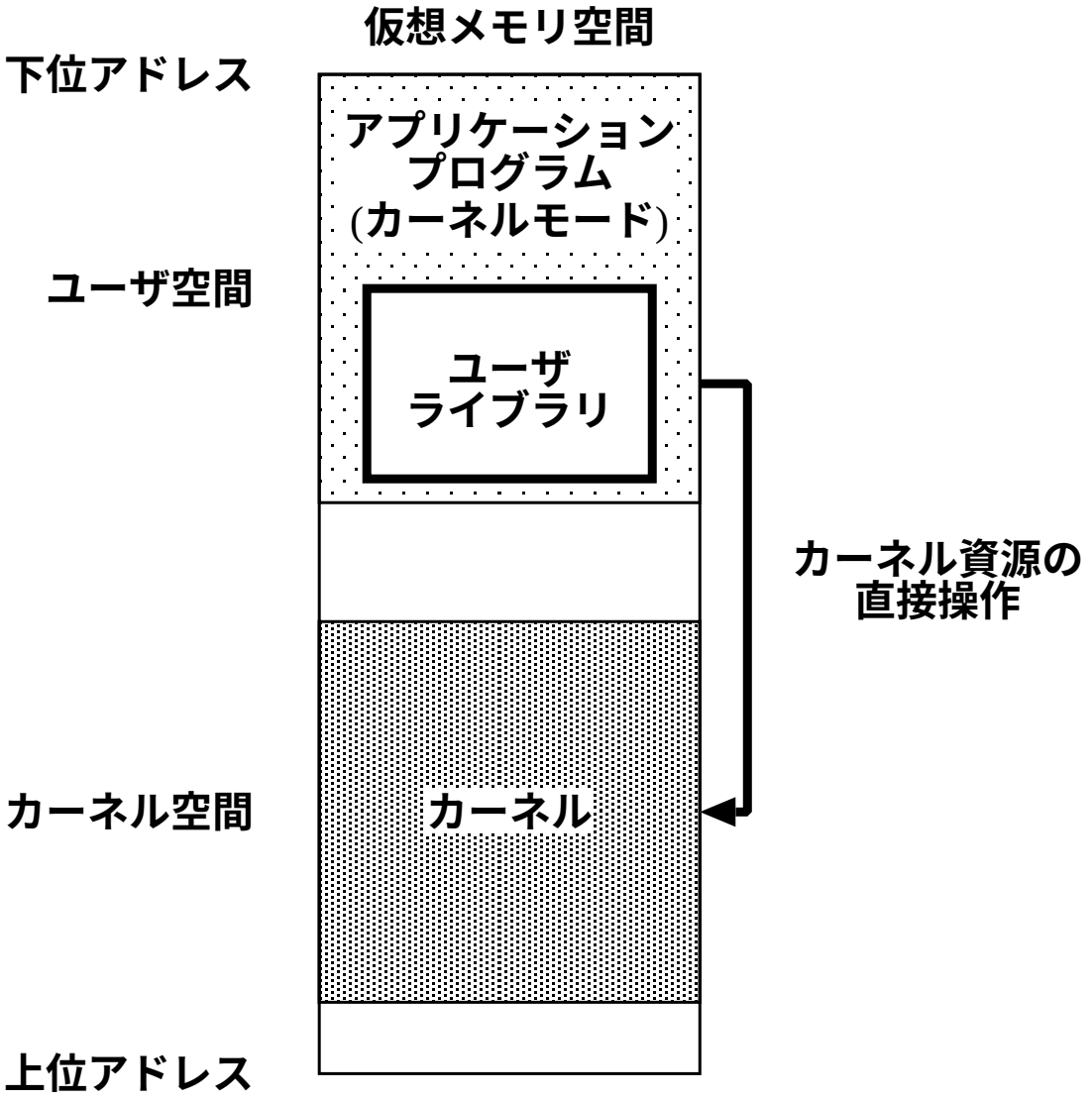

図 3.3: カーネルモード実行時のメモリ配置

3.2 本システムを実現するにあたっての問題

本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、カーネルモードで実行する機構は提供していない。そのため、本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。本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。本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、本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本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。

3.3 設計方針

本システムの設計方針を以下に挙げる。

- アプリケーションの変更は最小限

  カーネルモードで実行するアプリケーションへ加える変更は最小限に抑える。変更を最小限に抑えることにより、開発者へかかる負担を抑えられる利点がある。

- カーネルの変更は最小限

  本システムを実装するためにカーネルに加える変更は最小限に抑える。変更を最小限に抑えることにより、本システムを導入することが容易になる利点がある。

- 本システムの実装は簡潔性を重視

  本システムの実装は簡潔性を重視する。実装を簡潔にすることは、システムへのバグの混入を減らすことにつながる。また、OS がバージョンアップした際に、システムのソース変更量を削減することができる利点がある。

- OS や他のアプリケーションへの影響を無くす

  本システムを導入することで、今まで正常に動作していた OS や他のアプリケーションが異常動作を起こすことは避けなくてはならない。

- 利用者の多い OS

  実装に使用する OS は、既存の良く利用されている物を使用する。良く利用されている OS を使用することは、対応するアプリケーションが豊富に存在する利点がある。

上記の方針にしたがって実装を行う。

第 4 章

システムの実装

本章では本システムの実装について述べる。まず、本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。

4.1 システム構成

本システムの実装は NetBSD 上で行った。NetBSD は Free の PC–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本システムのプラットフォームに適している。

本システムの構成は、図 4.1 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、本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

  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 (Loadable Kernel Module) 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

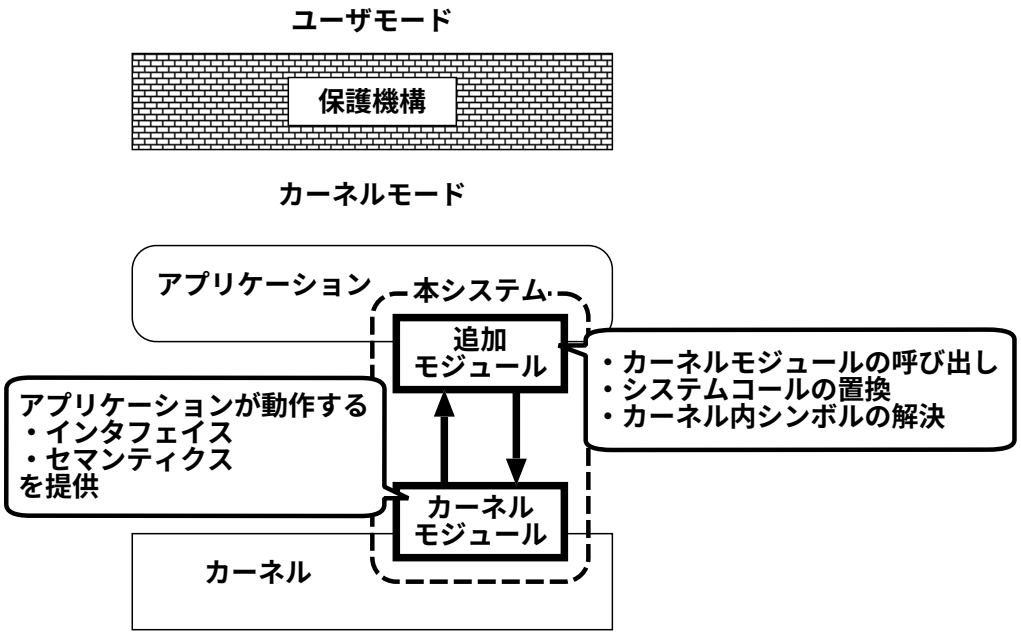

図 4.1: 本システムの構成

更やカーネルの再コンパイルは必要無く、本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。

4.2 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。

4.2.1 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、本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機構が必要となる。

- 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、本システムの実装コストが大きくなる。

本システムは、既存手法を用いる方法で実装を行った。これは、本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。

4.2.2 カーネルモードでの処理開始方法

本システムは、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機構が必要である。本システムではこの機構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 4.2 に示す。図 4.2 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。

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。図 4.3 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。

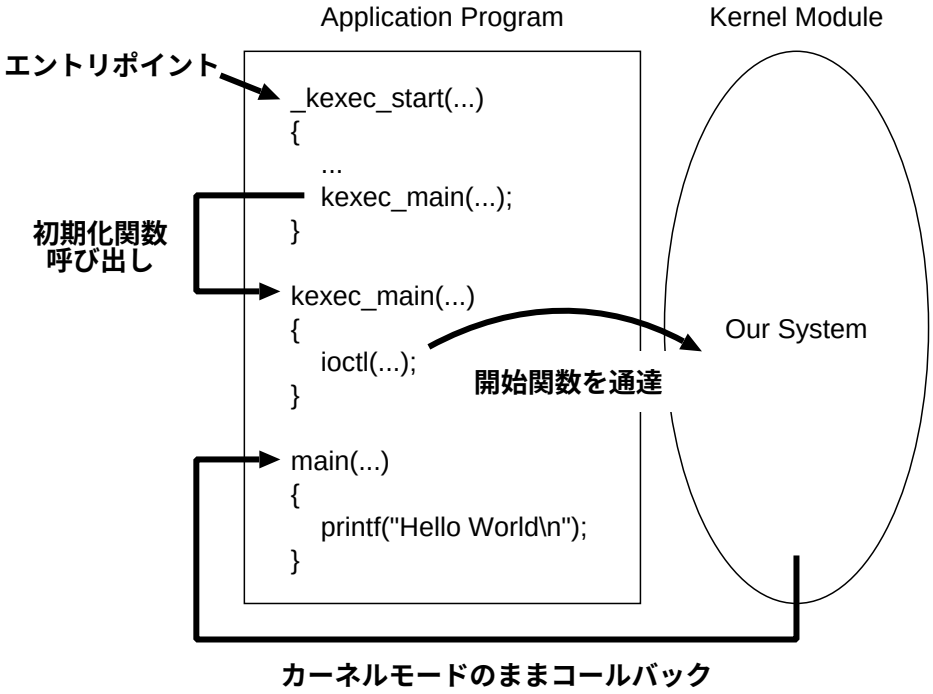

図 4.2: カーネルモードへの移行方法

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。ソースを図 4.4 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)
- 割り込みレベルの変更 (現在未実装)

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  }
```

図 4.3: 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  }
```

図 4.4: カーネルモード実行部のソース

4.2.3 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、実行開始部分で本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。本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図 4.5 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、本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。

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

図 4.5: エントリポイントの変更法

4.3 システムコールの置換

2 章の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。

本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。

関数型システムコールの処理の流れを図 4.6 に示す。通常のシステムコールの処理 (p.10 の図 2.2 参照) に比較して、関数型システムコールは CPU 動作モード

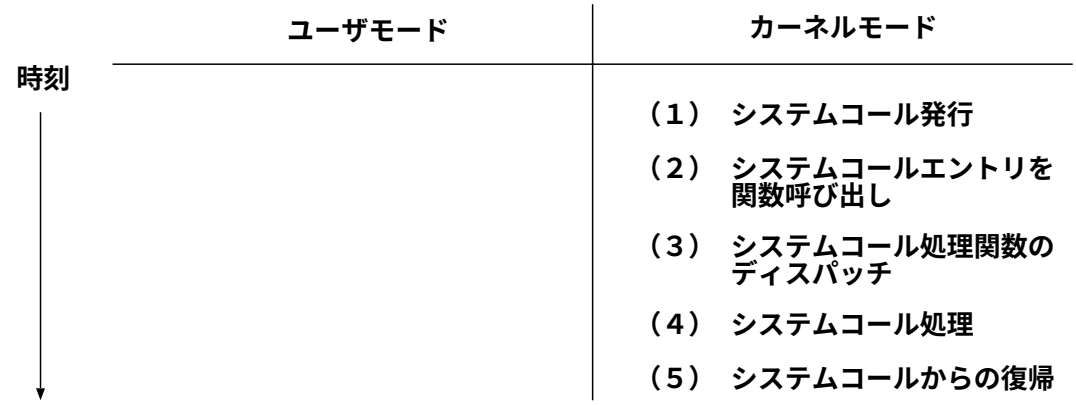

図 4.6: 関数型システムコールの処理の流れ

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、本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。

図 4.7 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。

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図 4.8 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プリ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、本システムの開発にかかるコストは少ない。

```
 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")));
```

図 4.7: 自動生成された置換用ソース (read() システムコール)

本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.29 図 4.4 の 11、12 行参照)。

```
     1  /*
     2   * syscall: "pipe" num: 42
     3   */
     4
     5  int
     6  _pipe(int *fdp)
     7  {
     8      struct kexec_frame frame;
     9      int 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_pipe, fdp);
    14      if (frame.is_error) {
    15          errno = frame.eax;
    16          ret = (int)-1;
    17      } else {
+   18          /* ファイルディスクプリタの設定 */
+   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")));
```

図 4.8: 一部変更が必要な置換用ソース (pipe() システムコール)

4.3.1 インターポジショニング

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 4.9 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。

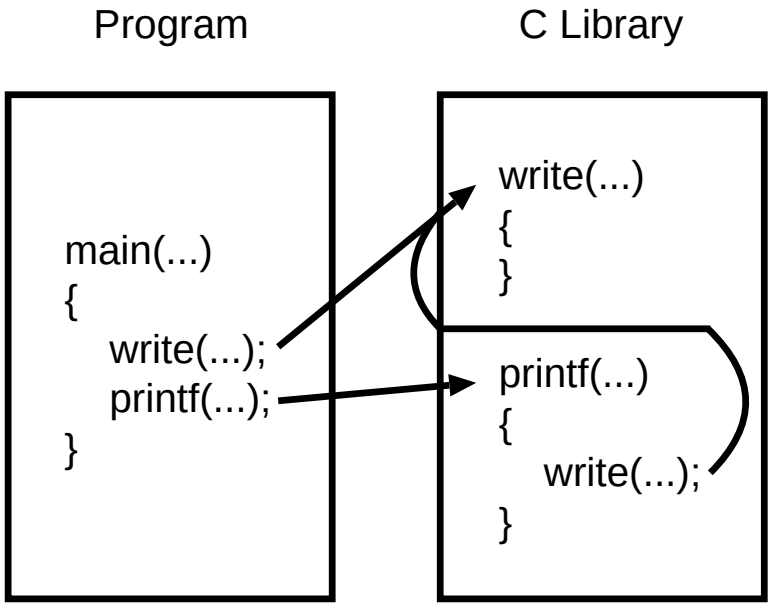

図 4.9: 通常のシステムコール呼び出し

そこで、本システムは図 4.10 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。本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。

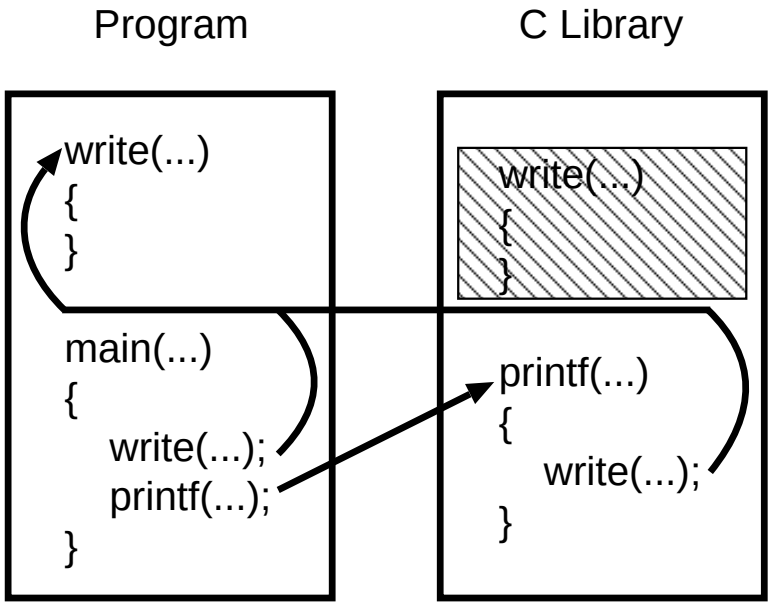

図 4.10: インターポジショニングが起きた場合

4.4 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事を言う。

本システムでは、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 4.11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。

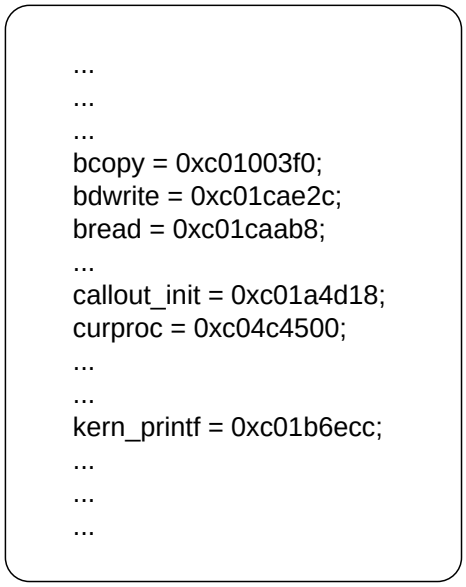

図 4.11: リンカスクリプトの内容

4.5 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。

そこで本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、図 4.12 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、本システムのカーネルモジュールに通達される。

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

図 4.12: スタックのアドレスを取得、設定するマクロ

4.6 スケジューリング

通常のアプリケーションプログラム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プログラム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。

本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プログラムは、ioctl() システムコールで呼び出されたカーネルモジュールからコールバックされる。NetBSD ではカーネルプログラム実行中はノンプリエンプティブであるため、本システムで実行されるアプリケーションプログラムはノンプリエンプティブで動作することになる。これでは処理が占有されてしまい、他のアプリケーションプログラムが動作できなくなってしまう。

現在、スケジューリング機構は未実装であり、実装は今後の課題となっている。

第 5 章

システムの評価

本章では本システムの評価を行う。評価の内容は、まず本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。次に本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 5.1に示す。

表 5.1: 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 (Time Stamp Counter)[9, 10] 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。

5.1 運用の容易さの評価

ここでは、本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、本システムの利用方法に焦点を当てる。

5.1.1 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、その内容は図 5.1 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  }
```

図 5.1: print_pid.c の内容

次にprint_pid.c を図 5.2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。

作成した実行ファイルは、図 5.3に示すように自分のプロセス ID を表示する。

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本システムを用いなくても動作する。

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

図 5.2: コンパイル方法 (通常版)

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

図 5.3: 実行結果 (通常版)

5.1.2 カーネルモードで動作するアプリケーション

ここでは本システムを用い、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図 5.1と同じ内容である。このソースを図 5.4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、本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本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。これらの詳細については、4 章のシステムの実装で述べた。

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

図 5.4: コンパイル方法 (カーネルモード実行版)

本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。図 5.5にカーネルモジュールの組み込み方法を示す。`kexec_mod.o` が本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。

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

図 5.5: 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。実行結果を図 5.6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。

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

図 5.6: 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ことが確認できた。

5.1.3 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 を図 5.7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  }
```

図 5.7: 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。

変更したソースを図 5.8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。

作成したアプリケーションの実行結果を図 5.9に示す。ユーザライブラリの

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

図 5.8: コンパイル方法 (カーネル内資源利用版)

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。

```
% ./kern_print_pid
proc(4682): Hello, user.
% dmesg
...
proc(4682): Hello, kernel.
...
```

図 5.9: 実行結果 (カーネル内資源利用版)

5.1.4 考察

本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。

5.2 システムコールオーバヘッドの削減

本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。

5.2.1 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、本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。

5.2.2 結果

図 5.10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、本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。

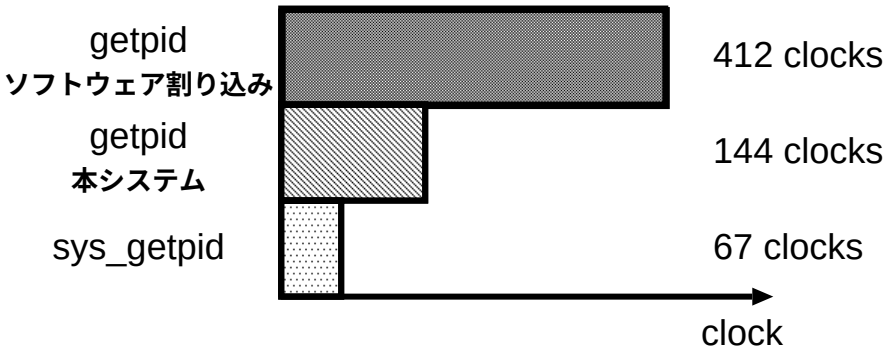

図 5.10: getpid() システムコールの処理時間

表 5.2に本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、本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、本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。

以上の測定により本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

表 5.2: 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 %

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、本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。

5.3 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。

5.3.1 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 5.11 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

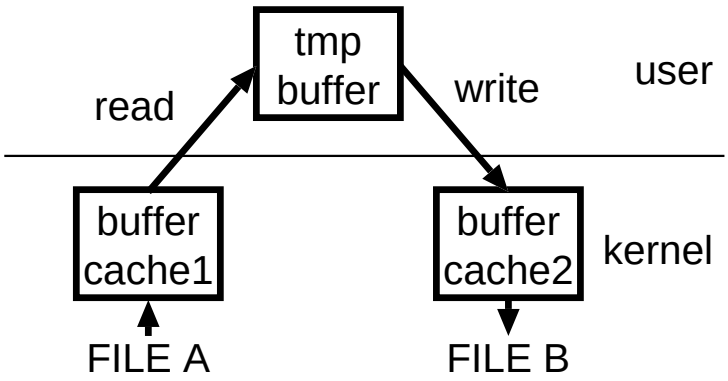

図 5.11: read() と write() システムコールを用いるファイルコピー

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。

そこで本システムを使用して、図 5.12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。

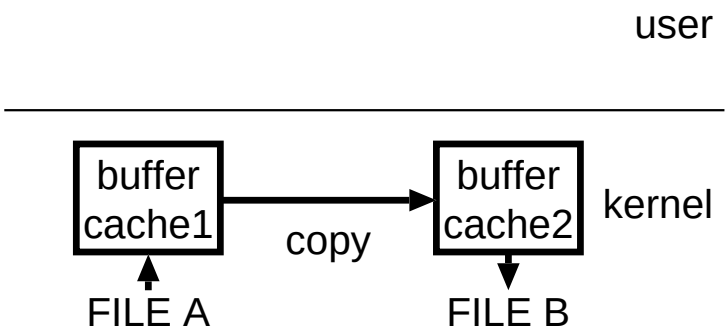

図 5.12: バッファキャッシュ間の直接コピー

図 5.13 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、本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  {

[... 省略...](初期化処理)

  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      }

[... 省略...](後処理)

  99      return error;
 100  }
```

図 5.13: バッファキャッシュ間でデータコピーをする関数 (一部省略)

5.3.2 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、本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。

5.3.3 結果

測定結果を図 5.14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、本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。本システムを使い、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。

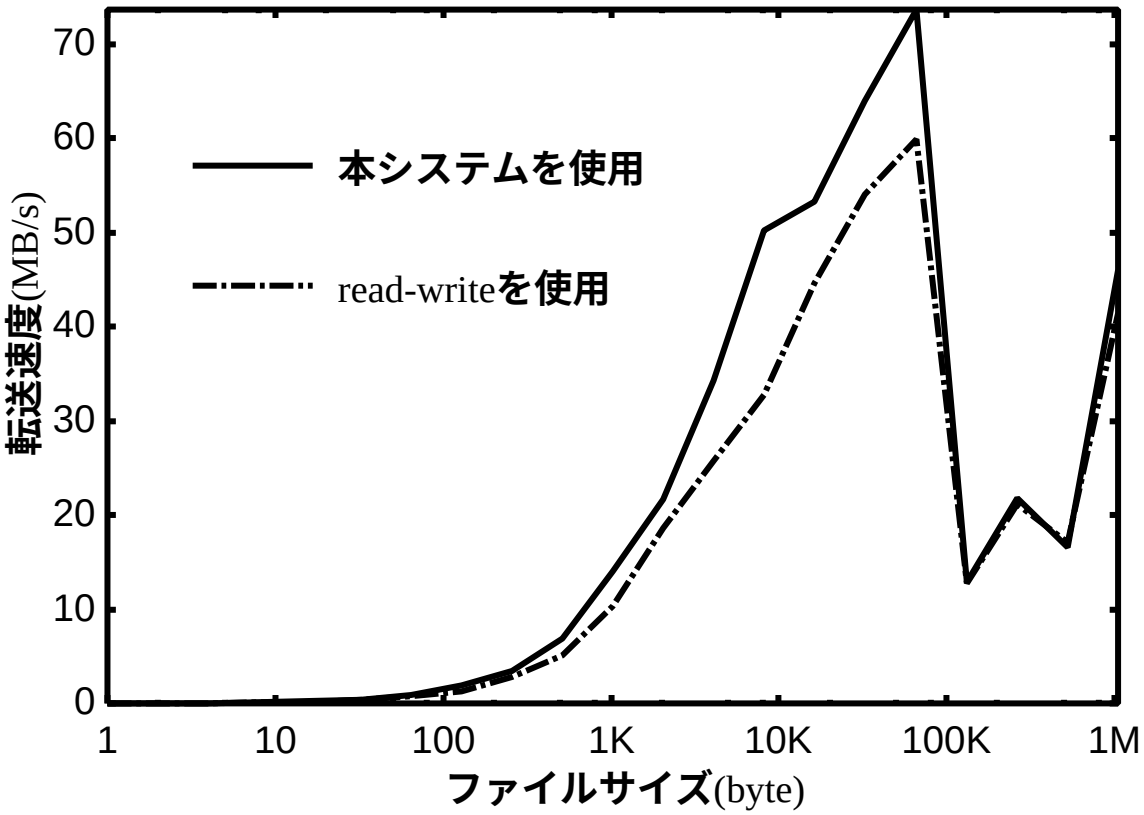

図 5.14: ファイルコピー速度の比較

第 6 章

関連研究

この章では関連研究について述べ、本システムとの比較を行う。

- kHTTPd

  kHTTPd[1] は Web サーバの機能を Linux のカーネルモジュールとして実装したものである。静的なデータ転送は全てカーネル内で処理し、Web サービスの高速化を実現している。本研究と比較した場合、kHTTPd はカーネルモジュールであるためユーザ側のライブラリが利用できず実装が難しい。

- Exokernel

  Exokernel[2] 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。Exokernel は研究用 OS であるのに対し、本研究は利用者数が多い NetBSD を使用しているため運用の面で利点がある。

- SPIN

  SPIN[3] 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させることが可能な OS である。アプリケーションは一部のコードをカーネル内で実行することで、ユーザとカーネルのモード遷移を抑えオーバヘッドの少ない計算機資源利用を行う。Exokernel と同様に研究用 OS であるため、本システムの方が運用の面で利点がある。

- Mach のサーバをマイクロカーネル内で実行

  文献 [4, 6] はマイクロカーネルである Mach のサーバをカーネル内で実行する機構についての研究である。マイクロカーネルはシステムコール呼び出しにメッセージパッシングを使うため、このオーバヘッドをカーネル内実行により削減している。カーネル内で使用できる低レベル操作は行わない方針をとっており、この点が本研究と異なる。

第 7 章

今後の課題

7.1 スケジューリング

本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はノンプリエンプティブで実行されるため、自発的に処理を放棄しない限り他のプロセスや OS に処理が渡らない。そのため、カーネルモードで動作するアプリケーションが処理を占有してしまう。

そこで、カーネルモードで動作するアプリケーションを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 OS にスケジューリングさせる必要がある。現在、スケジューリング機構は未実装であり、実装は今後の課題となっている。

7.2 マルチプロセスへの対応

Apache などの既存の Web サーバは複数プロセスを生成し、サービスを提供している。しかし、本システムはシングルプロセスのみ対応しているため、複数プロセスをカーネルモードで実行できず、Apache などのアプリケーションを実行できない。

本システム上で複数のプロセスをカーネルモードで実行するには、カーネルモジュールに記録する動作中のアプリケーションの情報を多重化する必要がある。

7.3 安全性の向上

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て不正な処理を行った場合、OS は異常動作、もしくは異常停止する。アプリケーションをカーネルモードで実行することは、システムの安全性を損ねる可能性がある。そこで鈴鹿らによって提案されたカーネル保護機構 [7] を利用し、システムの安全性を高めることが考えられる。

第 8 章

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、本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。

筆者は本システムを 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。

カーネル内資源を直接操作する改良は、全てのアプリケーションに有効では無い。数値計算処理のように I/O 処理をほとんど行わないアプリケーションはカーネル内資源を直接操作する必要が無いため、本システムを用いて得る恩恵は少ない。

本システムが想定するアプリケーションは、Web サーバなどの I/O 処理を頻繁に行う種類の物である。これらの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でき、効率的なサービス提供が可能になる。

謝辞

本研究を遂行するにあたって、いろいろな方々にお世話になりました。

まず、指導教官の多田好克先生には日頃から熱心なご指導、そしてご鞭撻を賜わりました。また、ご多忙中にもかかわらず論文の草稿を丁寧に読んで下さり、大変貴重なご助言をいただきました。また、研究を進めるにあたり、中村嘉志助手と安田絹子助手には貴重な助言を頂きました。ここに厚く御礼申し上げます。

そして、本研究が行えたことは、研究方針や方法論について議論をしてくださった福田伸彦さん、山之内暢彦さんら博士後期課程のみなさんのおかげです。また、多田研の学生諸氏とは互いに励まし合い研究を進めることができ、感謝しています。

最後に、本システムの開発環境として度重なるハングアップに耐えぬいた Panasonic Let's NOTE AL-N1 と、論文執筆環境として使用した SGI O2 の計算機達に感謝します。

参考文献

[1] kHTTPd Linux HTTP Accelerator Web Pages, `http://www.fenrus.demon.nl/`

[2] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr.: "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pp. 251–266 (1995).

[3] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers: "Extensibility, Safety, and Performance in the SPIN Operating System," *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pp. 267–284 (1995).

[4] Jay Lepreau, Mike Hibler, and Bryan Ford: "In-Kernel Servers on Mach 3.0: Implementation and Performance," *Proc. of the USENIX MACH III Symposium* (1993).

[5] Matt Welsh, Anindya Basu, and Thorsten von Eicken: "ATM and Fast Ethernet Network Interfaces for User–level Communication," *Proc. of the 3rd IEEE International Symposium on High Performance Computer Architecture* (1997).

[6] 黒岩 実, 石井 秀造, 高野 陽介, 横田 実: "Real-Time Mach におけるカーネル内タスクの実装と評価", 情報処理学会コンピュータシステム・シンポジウム論文集, pp.43–50 (1997).

[7] 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).

[8] 佐藤 喬, 中村 嘉志, 多田 好克: "アプリケーションプログラムのカーネル内実

行による高速化”, 情報処理学会第 43 回プログラミング・シンポジウム報告集, pp.153–160 (2002).

[9] Intel, Corp.: “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”, ftp://download.intel.co.jp/jp/developer/jpdoc/24319002_j.pdf, (1999).

[10] Intel, Corp.: “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”, ftp://download.intel.co.jp/jp/developer/jpdoc/24319102_j.pdf, (1999).

[11] 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, ISBN 0-201-54979-4, (1996).