2004－OS－96　(15)
2004／6／18

カーネルウェア：アプリケーションプログラムを流用するOS機能拡張法の提案

佐藤 喬[†]　　多田 好克[†]

本研究では，アプリケーションプログラムを変更することなく，汎用 OS のカーネル内で実行する機構を提案する．近年，アプリケーションプログラムの機能をカーネルモジュールとして実装し，カーネル内に取り込む動きがある．しかし，アプリケーションプログラムはそのままカーネルモジュールとして利用することはできないため，大幅な書き換えが必要となり，開発の効率が悪い．そこで，アプリケーションプログラムを改変することなくカーネル内実行できれば開発効率は格段に向上するであろう．本機構を用いてカーネル内で実行するアプリケーションプログラムをカーネルウェアと呼ぶ．OS 開発者は，既存のアプリケーションプログラムをカーネルゥエア化することで，容易にアプリケーションプログラムの機能をカーネル内に実装し OS の機能拡張を行える．

Kernelware: An OS extension mechanism by reusing application programs

Takashi Satou [†]　and Yoshikatsu Tada[†]

Kernelware is an application program, which runs in kernel without any modification. Recently, a function of a application program is executed in kernel as a kernel module. But the application program can not be reused in the kernel module. Developers have to change the application program widely, and will pay much development cost. If the application program can run in kernel without any modifications, developers can improve it efficiently. Using our system, existing application programs can be executed in kernel, and developers can extend OS easily with using a function of a application program easily.

1　はじめに

Web サーバなどの I/O 操作を頻繁に行うアプリケーションプログラム (以下 AP) には CPU の動作モード遷移のオーバヘッドがボトルネックとなり，十分な性能を発揮できない問題がある [3]．これは，モード遷移があるたびに，AP は OS の保護境界を跨ぎ，コンテクストの保存や復帰といったオーバヘッドの大きな処理を必要とするためである．

そこで，この問題を解決するため AP が行うサービスをカーネル内で提供する手法がある [3, 5]．カーネル内であれば，モード遷移の必要がなくなり問題を解決できる．さらに，カーネル内の低レベル操作を使い，より効率的な I/O 処理を実現できる利点もある．

しかし，このような解決策は開発効率が悪いという新たな問題がある．なぜなら， カーネル内で実行されるコードの記述には AP のソースをそのまま再利用することができず，多くの変更が必要となるためである．これは，カーネル内では AP の利用しているインタフェイスやセマンティクスが利用できないことが原因である．

そこで我々は，カーネル内で AP の利用するインタフェイスとセマンティクスを提供し，AP に変更を加えることなくカーネル内で実行する機構を実現した．もし，AP を変更すること無しにカーネル内で実行する仕組みをつくれば，運用実績の豊富な既存の AP のサービスを容易にカーネル内の機能として提供でき，開発効率の問題を解決できる．

本機構を使ってカーネル内で実行される AP をカーネルウェアと定義する．AP をカーネルウェア化することで，モード遷移のオーバヘッドを削減できる．さらに，カーネルウェアからカーネル内資源を直接操作できる枠組を用意し，システムコールでは実現不可能な，AP の処理内容に特化した I/O 処理手段を開発者に提供する．ルータなどの最近の組み込みシステムでは，汎用 OS 上で少数の特定 AP を動作させていることが多い．このような場面にお

[†] 電気通信大学 大学院情報システム学研究科
Graduate School of Information Systems, The University of Electro–Communications

いて，本機構を使い特定 AP に的をしぼって改良するアプローチは有効である．

本機構は汎用 OS のひとつである NetBSD 上で実装した．そのため，動作実績のある AP が豊富に存在する．開発者はそれらの既存 AP をカーネルウェア化することで，容易にカーネル内に取り込むことができ，効率的なサービスを提供できる．

2 カーネルウェア化の流れ

本機構を用いて AP のカーネルウェア化を行う場合の流れを説明する．行う手順は大きく二段階に分けられる．

まず，開発者はカーネルウェア化したい AP を用意し，本機構を用いて図 1 のようにカーネル内実行する．この際，AP へ変更を加える必要はない．これは，本機構が AP に対してユーザモードで動作するのと同じインタフェイスとセマンティクスを提供するためである．この段階で，AP はモード遷移のオーバヘッドを削減することができ，その分のシステムコールの効率化が可能となる．

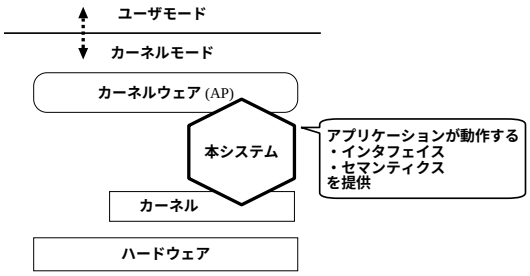

図 1: アプリケーションプログラム（AP）のカーネル内実行

次に，開発者は AP の処理内容に応じて，図 2 のように計算機資源を直接操作する改良を加え，I/O 処理のボトルネック部分を解消する．この際，改良箇所以外のソースは元の AP の物をそのまま流用できる．これにより，開発者は改良箇所にのみ集中すれば良い．そのため，全てを書き直す必要があるカーネルモジュールに比べ，格段に少ない手間で効率の良い開発ができる．

このように，既存 AP を再利用することで，カーネルモジュールよりも開発効率を高めながら，カーネルモジュールと同様の計算機資源を直接操作する改良を加えることができる．

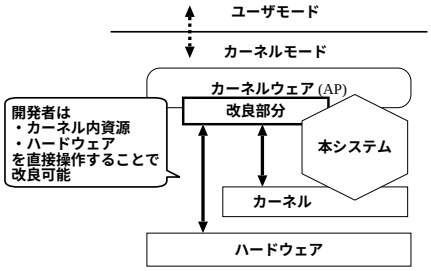

図 2: 計算機資源の直接利用による改良

3 設計方針

開発効率が高く，拡張が容易なカーネルウェアを実現するため，本機構では以下のような設計方針をたてた．

- AP の変更は最小限：

  AP の再利用性を高めるため，AP に必要な変更は少ないことが望ましい．AP 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 AP への影響は最小限：

  本機構を導入したことによって，OS や関係ない他の AP 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 AP が存在する．AP が豊富にあれば，本システムの利用局面を増やすことができる．

4 実装

本機構は NetBSD–1.5.3/i386 上に実装した．NetBSD は UNIX 系の OS であり，その上では豊富な AP が

動作している．カーネルに加えた変更は全て LKM（Loadable Kernel Modules） により実現した．そのため，本システムの導入は動的に行え，カーネルの再コンパイルは必要ない．LKM によりカーネルに組み込まれるモジュールは，2000 行の C プログラムと約 800 行のアセンブリプログラムで構築されている．

4.1 要求点

本機構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：

  AP をカーネル内で実行する機構が必要である．その際，AP 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 AP 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，開発者は処理内容に特化した改良を加えることができる．

- 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，処理を占有してはならない．そのため，割り込みが発生した際やシステムコール呼出し時に，必要ならば他の AP 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，本システムでは，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．

4.2 カーネル内実行環境

実行の開始は，AP を実行する exec システムコールをフックし，実行開始のコードセグメントをユーザからカーネルへ変更することで実現する．この exec のフックは特定プロセスのみシステムコールテーブルを書き換えることで行う．そのため，カーネルウェア以外の AP に対しては影響が無い．

フックした exec システムコールを使用することで，カーネルウェアは図 3 のようにユーザモードで動作する AP と同じメモリ配置になる．そのため，AP へ加える変更を抑えることができる．

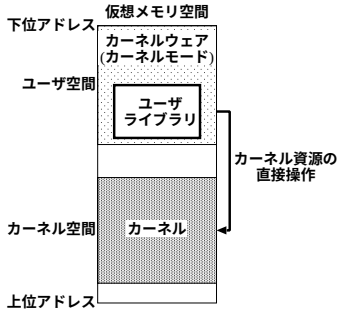

図 3: カーネル内実行時のメモリ配置

カーネルウェアは，カーネルモードで実行可能な点を除いて，ユーザモードで動作するプロセスと同じように管理されている．そのため，fork のようなプロセスを生成するシステムコールも利用可能である．fork によって生成された子プロセスの動作モードは選択可能にする予定であるが，現状ではカーネルモードでのみ実行可能としている．これは，Web

サーバなどのデーモン型のプログラムの場合，ファイル転送の I/O 処理を子プロセスで行っているためである．

また，カーネルウェア中の I/O 処理オーバヘッドの大きな処理のみをカーネル内実行するために，switch_to_kernel_mode() と switch_to_user_mode() という関数を用意した．これにより，カーネルウェア内から明示的にモード遷移を制御できる．

4.3 システムコールの関数呼出し

AP に変更を加えずにシステムコール呼出しのオーバヘッドを削減するため，本システムでは関数呼出し型のシステムコールを持つ libc ライブラリを作成した．カーネルウェアは，このライブラリをリンクすることで，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利用できる．大部分のシステムコールインタフェイスのソースはマクロを元に生成されているので，この変更はマクロを書き換えることで容易に実現できた．

この libc ライブラリは動的リンクに対応しているため，動的リンクを使用している AP であれば，再コンパイルの必要無くこのライブラリを使用できる．また，カーネルウェアの動作モードに応じてソフトウェア割り込みか関数呼出しかを切替えるハイブリッド仕様になっている．そのため，switch_to_kernel_mode() や switch_to_user_mode() を使用して動作モードを切替えても正しく動作する．

4.4 カーネル内シンボルの解決

カーネルウェアからカーネル内資源を利用した改良を加えるためには，カーネル内シンボルの参照を解決する必要がある．カーネル内のシンボル解決には，カーネルのシンボルとアドレスとが記述されているリンカスクリプトを用いる静的な方法を採用した．カーネル内のシンボルを使用するカーネルウェアはコンパイル時にこのリンカスクリプトを使用することで，シンボルのアドレス解決を行う．

現状では静的にシンボルを解決しているために，カーネルの変更ごとにカーネルウェアの再コンパイルが必要となっている．そのため，動的リンクを使った実行時のシンボル解決が今後の課題となっている．

4.5 プリエンプション

ユーザモードで動作する AP と同様，カーネルウェアもプリエンプティブでなければならない．NetBSD はユーザモードでの割り込みに対し，必要であれば他の AP へ処理を渡すことでプリエンプションを実現している．しかし，カーネルウェアはカーネルモードで動作しているため，割り込みが発生してもこのままではプリエンプションが発生しない．そこで，本システムでは割り込みをフックし，カーネルモードでの割り込みでも，カーネルウェアに対してのものであればプリエンプションを発生させる．

この判断には，モード遷移が起こらない場合にスタックの切替えが起こらないことを利用した．もし，割り込みが発生しカーネルに処理が渡った時，スタックポインタがユーザのメモリ空間を指していればカーネルウェアの実行中であると判断できる．なぜなら，他の AP やカーネル処理中の割り込みは全てカーネル空間のスタックを使っているためである．この方法であれば，フック処理の記述を簡潔にできるため，割り込み発生時のオーバヘッドを低く抑えることができる．

割り込みのフックは，LKM により IDT(Interrupt Descriptor Table) を上書きすることで実現している．割り込みが発生すると，本システムが提供するフックコードが実行されるようになる．本システムを取り外す場合は元の IDT に復旧することで，OS に対する影響を無くしている．一般に，他のアーキテクチャのマシンでも，この方法は適用可能である．

また，システムコールを関数呼出しした場合は割り込みが起こらないため，システムコール終了時にプリエンプションの処理が必要かどうかチェックしている．

4.6 安全性

本システムでは運用実績が豊富な AP を使用することで，安全性の向上をはかる．これは，運用実績が豊富な AP であれば，異常動作を起こす確率が低いと考えられるためである．また，カーネルモードで実行する部分を限定することで，異常動作により OS に影響がでることを抑えることができる．

カーネルウェアに対して加える変更に対しては，カーネル拡張コードが及ぼす悪影響を検知する鈴鹿

らの技法 [7] を利用し，安全性の向上をはかることを予定している．

5 実験

本システムを評価するため，3 つの実験を行った．まず，本システム導入による割り込みオーバヘッドの測定を行った．これにより本システムの導入によって，他の AP へどの程度の負担がかかるか測定できる．次にシステムコールを関数呼出し化したことでどの程度オーバヘッドを削減できるかを測定した．最後にシステムの利用例として AP のカーネルウェア化を行い，どの程度の効率化ができるか測定した．

これらの実験は CPU が Celeron 500MHz，メモリが 64MB の計算機を使用して行った．測定時間は CPU の TSC （Time Stamp Counter） を使ってクロック単位で測定した．

5.1 割り込みフックによるオーバヘッド

本システムでは，プリエンプション実現のため割り込みをフックしている．ここではそのフックがどの程度他の AP へ影響を与えるか測定した．

測定方法として，割り込み内での処理量が少ない getpid システムコールの処理時間を測定する．システム導入前と後を比較することで，フックが及ぼすオーバヘッドの測定ができる．ここで測定した getpid は，どちらも本システムを使わず，ユーザモードから呼出したものである．

各 1000 回ずつ呼出し測定した．図 4 は 1 回あたりのクロック数の平均と，最小値と最大値の範囲をグラフ化した物である．どちらも平均は 450 クロック程度で違いは見られない．これは割り込み処理に対してフックによる影響の小さいことを示す．つまり，システム導入によって OS や他の AP の割り込み処理が遅くなるような影響は出ないと考えて良い．

5.2 システムコールオーバヘッドの削減

カーネルウェアは，割り込みを使わない軽量な関数呼出し型のシステムコールを利用できる．そこ

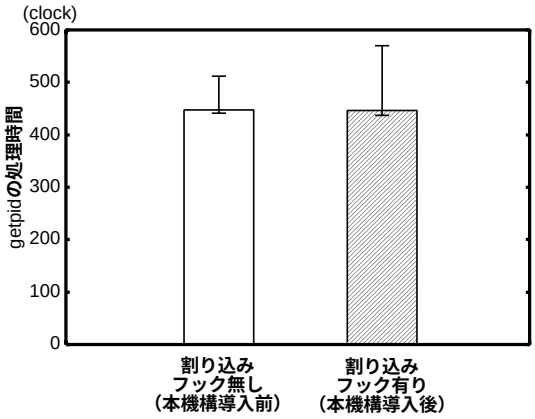

図 4: 割り込みをフックしたことによるオーバヘッド

で，通常のシステムコールと関数呼出し型のシステムコールの処理時間の比較を行った．ファイル操作を行うシステムコールは MFS （Memory based File System）上で測定した．これは，ディスク I/O 処理時間のばらつきを抑えるためである．

図 5 は，getpid と stat という，カーネル内での処理内容が少ないシステムコールを測定したものである．getpid は約 50%，stat は約 20%程度処理時間が削減されている．

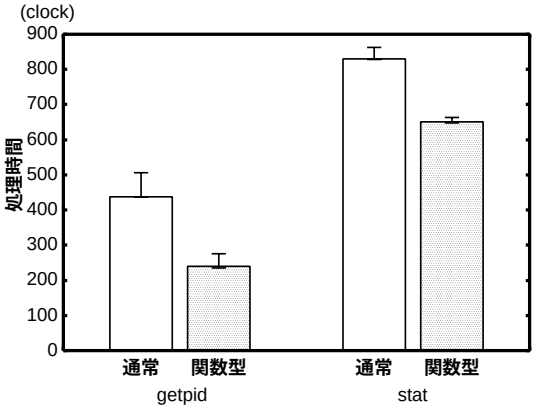

図 5: 内部処理が少ないシステムコールの処理時間

図 6 は，read と write で 64KB のファイルを扱うといった，カーネル内での処理内容が多いシステムコールの測定結果である．この場合，両者には差が見られない．これは，呼出しオーバヘッドの削減のみを行っているため，カーネル内の処理内容の増加とともに実行時間の比率が小さくなっているためである．もし，より処理時間を削減するには，カーネル内で動作することを利用をしたシステムコール

の内部処理の改良が必要である．

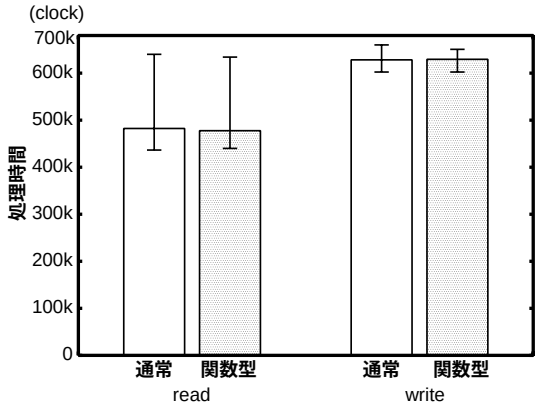

図 6: 内部処理が多いシステムコールの処理時間

5.3 AP のカーネルウェア化

本システムの利用例として，ls コマンドと cp コマンドのカーネルウェア化を行った．ls コマンドは stat などの処理量が少ないシステムコールを多量に使用する．そのため，ソースの変更を行わずに，システムコールを関数呼出し型に変更するだけで，性能向上が期待できる．一方，cp コマンドは read や write システムコールの処理量が多いため，システムコールを関数呼出し型に変更するだけでは性能向上に不十分である．そこで，カーネル内資源の直接操作を行う改良を加え，測定を行った．

5.3.1 ls コマンドのカーネルウェア化

MFS と HDD 上の 1000 個のファイルに対し通常の ls と関数呼出し型の ls の二つを使い測定を行った．本システムで実行される関数呼出し型の ls のソースは通常の ls と同じものであり，変更は加えていない．図 7 が測定結果である．

関数呼出し型の ls の方が，MFS，HDD のどちらのファイルに対しても，約 5% の処理時間が削減されている．これは，ls が stat などの処理量が少ないシステムコールを多量に使用しており，システムコール呼出しオーバヘッドの削減が性能向上に結び付いているためである．

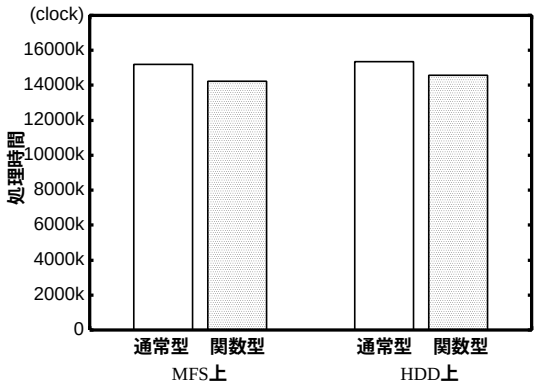

図 7: ls コマンドの処理時間

5.3.2 cp コマンドのカーネルウェア化

通常の cp はユーザ空間のバッファを介してファイル内容をコピーする．この際，ユーザ空間のバッファに変更は加えられないため，このコピーはカーネル内であれば無駄である．そこで，カーネル内資源を直接操作してユーザ空間のバッファを介さずに，バッファキャッシュ間で直接コピーするカーネルウェアを作成した．

このカーネルウェアは，通常の cp コマンドのソースにバッファキャッシュ間コピー用関数を呼び出すためコードを追加し作成した．追加したコードの量は，通常の cp のソース 897 行に対し 11 行だけであり，全体の約 1.2%と非常に少なくすんでいる．追加した 11 行のコードには，ユーザモードで呼ばれた場合に read と write システムコールを使った通常の cp 処理にジャンプしたり，エラー処理するためのコードも含まれている．

MFS 上の測定結果を図 8 に示す．測定は通常の cp，関数呼出し型システムコールを使った cp，そしてバッファキャッシュ間コピーを行う cp を使用した．通常の cp 以外は，本システムを使用している．関数呼出し型システムコールを使った場合，性能向上はみられなかった．これは，図 6 で示したように，read と write システムコールの処理が大きなためである．一方，バッファキャッシュ間のコピーを行った場合は，2MB のファイルコピーの際，約 26%の処理時間の削減が実現され，カーネル内資源の直接操作の有効性が確認できた．

HDD 上のファイルに対し測定を行った結果を図 9 に示す．HDD へのアクセスが発生するため，処理時間削減の割合が最大で約 3%に低下している．

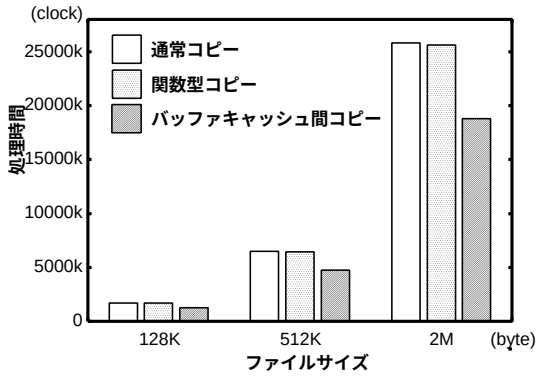

図 8: MFS 上のファイルコピー時間

より処理時間の削減をはかるには，田胡ら [8] が述べているような，データコピーに CPU が介在しないゼロコピー技術を利用することが考えられる．

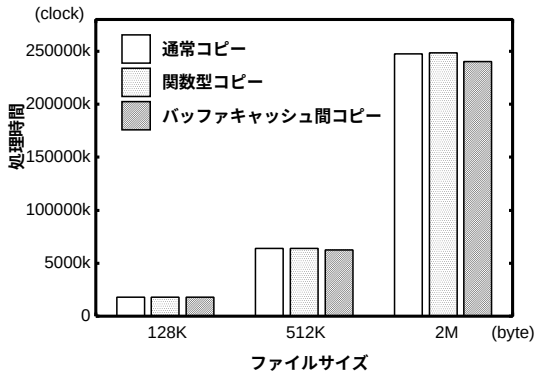

図 9: HDD 上のファイルコピー時間

6 今後の予定

ここでは，本システムを用いて今後実現する予定のカーネルウェアについて述べる．

6.1 Web サーバの高速化

Web サーバは I/O 処理が頻繁に行われる AP の一つである．既存の Web サーバをカーネルウェア化することで，I/O 処理オーバヘッドの削減を行い性能向上を実現する．

改良内容として，転送データが存在するバッファキャッシュの内容を直接 NIC(Network Interface Card) に転送し，データコピーの回数を減らすことを想定している．このような改良を行うのに必要な改良手間と，それによってもたらされる性能向上を調べ，本システムの有効性を確認する．

6.2 スクリプト言語処理系の組み込み

perl や ruby，lisp などのスクリプト言語処理系をカーネルウェア化し，これらのスクリプト言語によりカーネル内の機能を操作する枠組を作る．スクリプト言語の記述性を利用し，カーネル内の機能拡張を容易にするのが狙いである．また，スクリプト内容を変更することでカーネルの動作を動的に変更することも考えている．

7 関連研究

ここでは，関連研究について述べ，本機構との比較を行う．

7.1 カーネルモジュール

カーネルモジュールとして AP のサービスを実現する手法である．例えば Linux の kHTTPd[5] 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．

本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，本システムでは既存 AP を利用できるため，開発効率の面で有利である．

7.2 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 AP 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．

7.3 専用 OS

Exokernel[2] は計算機資源管理ポリシーを AP 側に任せることが可能な OS である．AP 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[1] は AP 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 AP に提供する手法に対し，本システムは汎用 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 AP を豊富に利用できる．

7.4 カーネル内実行 AP

Kernel Mode Linux[6] は AP 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[4] も Kernel Mode Linux と同様，AP 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．

本システムでは，カーネル内資源を直接操作できる枠組を用意している．これを用いてカーネルウェアの処理内容に特化した I/O 処理手段を提供している点が，他のカーネル内実行 AP 機構と異なる．

8 まとめと今後の予定

本研究では AP を変更すること無く汎用 OS のカーネル内で実行し，カーネルウェアとして OS 機能を拡張する手法を提案し，実装した．カーネル内で実行することにより，カーネルウェアは I/O 処理を行う場合，モード遷移のオーバヘッドを受けずに済む．さらに，カーネル内資源を直接操作する枠組を提供することで，開発者はカーネルウェアの I/O 処理内容に特化した改良を加えることができる．また，既に存在する豊富な AP をカーネルウェア開発の土台として利用できる．

実験により，本システム導入による OS や他の AP へ与えるオーバヘッドは少ないこと，システムコールを関数呼出しすることで，モード遷移のオーバヘッドを削減できることを確認した．また，システム利用例として，cp コマンドのカーネルウェア化を行い，カーネル資源の直接操作を行う改良を加え性能が向上したことを確認した．

今後の予定として，Web サーバのような大きな AP をカーネルウェア化し，I/O 処理オーバヘッドの削減方法について考察し，カーネルウェアで共有できる改良方法をライブラリ化する．また，安全性の向上や，カーネル内シンボルの動的解決について検討する．

参考文献

[1] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, Marc E. Fiuczynski, et al. Extensibility, safety, and performance in the spin operating system, In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 267–284, 1995.

[2] 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 15th ACM Symposium on Operating Systems Principles*, pp. 251–266, 1995.

[3] Philippe Joubert, Robert B. King, Rich Neves, Mark Russinovich, and John M. Tracey, High-performance memory-based web servers: Kernel and user-space performance, In *Proceedings of the USENIX Annual Technical Conference*, 2001.

[4] Amit Purohit, Charles P. Wright, Joseph Spadavecchia, and Erez Zadok, Cosy: Develop in user-land, run in kernel-mode, In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, 2003.

[5] Arjan van de Ven, khttpd linux http accelerator web pages, http://www.fenrus.demon.nl/.

[6] 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, 2002.

[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] 田胡和哉, 根岸康, 奥山健一, 村田浩樹, 松永拓也, オープンソフトウェアによる network attached storage の性能の解析および改善に関する一試み, 情報処理学会論文誌, Vol. 44, No. 2, pp. 344–352, 2003.