2006－OS－101（4）
2006／2／16

走行モード変更機構を利用したデバイスドライバの実現

野村裕佑[†]　乃村能成[†]　横山和俊[†,††]　谷口秀夫[†]　丸山勝巳[†††]

既存の多くのオペレーティングシステムにおいて，デバイスドライバはカーネルに組み込まれて動作する．そのため，デバイスドライバの信頼性がオペレーティングシステム全体の信頼性に大きく影響する．ここでは，デバイスドライバをカーネルから分離し，ユーザプロセスとして実現する手法を提案する．提案手法は，応用プログラムの走行モード変更機構を利用する．デバイスドライバをユーザプロセスとして実装することにより，オペレーティングシステム全体の信頼性の向上が期待できる．また，提案手法を FreeBSD4.3 上に実装し，基本的なオーバヘッドを評価した．

An implementation of device drivers using running mode switch mechanism

Yusuke Nomura,[†] Yoshinari Nomura,[†] Kazutoshi Yokoyama,[†,††]
Hideo Taniguchi[†] and Katsumi Maruyama[†††]

Conventional operating systems used to implement device drivers as kernel modules or embedded objects. Therefore, maturity of a device driver influences the reliability of a whole system. In this paper, we propose a method for constructing isolated device drivers in the form of user process. Our proposed method uses the running mode switch mechanism of application programs. User process device drivers enhance the reliability of the operating system. We evaluated the basic overhead of our method.

1. はじめに

計算機の進歩に伴い，次々に新たな入出力機器が登場している．オペレーティングシステム（以降，OS と略す）において，これら入出力機器はデバイスドライバと呼ばれるプログラムによって制御される．OS を最新の入出力機器に対応させるためには，デバイスドライバを順次開発する必要がある．このため，デバイスドライバは様々な時期に様々な開発元によって作成されることが多い．したがって，デバイスドライバの信頼性は開発元により様々なものとなる．

既存の多くの OS では，デバイスドライバはカーネルに組み込まれ，カーネルの一部として動作する．そのため，デバイスドライバの信頼性が OS 全体の信頼性に大きく影響する．

OS の信頼性を向上させる方法として，デバイスドライバをユーザプロセスとしてカーネルから分離する方法がある．ここでは，応用プログラムの走行モード変更機構[1]を利用し，ユーザプロセスとして動作するデバイスドライバの実現について述べる．

†　岡山大学大学院自然科学研究科
††　(株)NTT データ技術開発本部
†††　国立情報学研究所

2. 走行モード変更機構

2.1 概要

システムコールは，応用プログラム（以降，AP と略す）が OS に処理を依頼する機能を提供する．システムコールによる OS 処理への移行は，走行モード変更の処理を伴うためオーバヘッドが大きい．システムコールのオーバヘッドを削減する手法として，AP をスーパバイザモードで実行する手法がある[2,3,4]．これらの手法では，AP から OS のシステムコール処理を直接呼び出すことができる．このため，従来の OS で発生する走行モード変更の処理が不要となる．これらの研究では，プロセスが AP を実行する走行モードを
OS の起動時やプロセス生成時に決定する．

このため，プロセスの走行モード変更はプロセス停止を伴い, サービスが中断してしまう．

走行モード変更機構[1]は，AP を実行する走行モードを，サービスを停止することなく実行中の任意の時点でユーザモードからスーパバイザモードに，またはスーパバイザモードからユーザモードに変更する機構である．

図 1 に走行モード変更機構を利用したプロセス実行の概念を示す．プロセスは，走行モード変更システムコールにより任意の時点で走行モードを変更できる．switch_supervisor システムコールは，プロセスをスーパバイザモード走行に変更する．逆に， switch_user システムコールは，プロセスをユーザモード走行に変更する．走行モード変更機構を利用したプロセスは，ユーザモード走行時はソフトウェア割り込みによるシステムコールを用い，スーパバイザモード走行時は関数呼出しによるシステムコールを用いる．このため，スーパバイザモード走行時のプロセスはシステムコールのオーバヘッドを削減できる．また，スーパバイザモード走行時のプロセスのユーザ領域は，OS 領域と同様にページアウトの対象外としてメモリに常駐する．これにより，スーパバイザモードプロセスではユーザ領域のページインとページアウトが発生しないため，さらにプロセスの実行を効率化することができる．

2.2 実現方式

Pentium4 プロセッサと FreeBSD を例にしたモード変更システムコールの動作を示す．Pentium4 プロセッサでは，プロセスの走行モードは CS レジスタにより制御される．FreeBSD では，AP 実行から OS 実行に移行する際に，CS レジスタの値をカーネルスタック上に退避する．また，OS 実行から AP 実行に移行する際, カーネルスタック上の CS レジスタ値を回復することで AP 実行と OS 実行の走行モードを変更する．

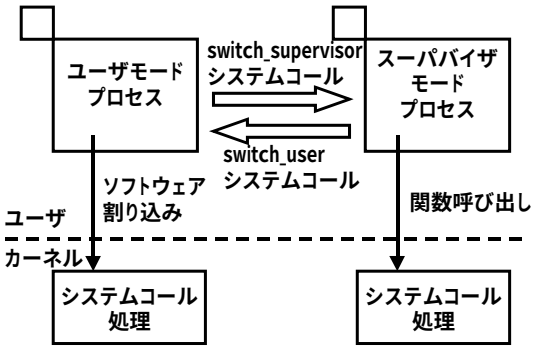

図1 走行モード変更機構の概要

図 2 はプロセス A がプロセス B の走行モードを変更する場合を示している．このとき，プロセス A の発行したシステムコールはプロセス B のモード変更要求を登録する．次に，プロセス B が任意のシステムコールを発行した時点で走行モード変更要求を確認し，要求が登録されている場合は自身のモードフラグと CS レジスタを更新し，走行モードを変更する．

3. デバイスドライバの実現

3.1 概要

走行モード変更機構を利用したデバイスドライバの概要を図 3 に示し, 以下に説明する．スーパバイザモードで走行するユーザプロセスは，通常保護されているメモリ領域へのアクセス，および I/O 命令を発行することができるので，ハードウェアを直接制御することができる．以下ではユーザプロセスとして実現したデバイスドライバをコアと呼ぶ．

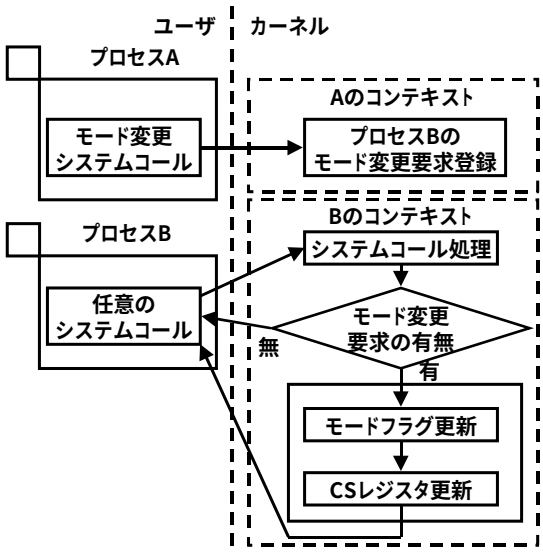

図2 走行モード変更機構の実現方式

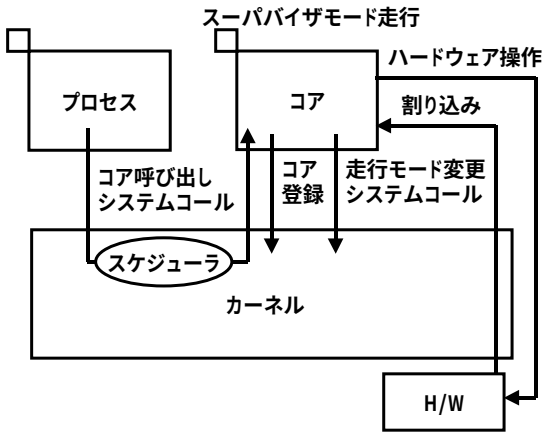

図3 デバイスドライバの概要

コアは，初期化時に走行モード変更システムコールを発行し，自身をスーパバイザモードプロセスとする．これによりハードウェアを直接制御し，ハードウェアの初期化処理を行う．初期化処理の末尾でコア登録システムコールを発行し，ドライバの種類，ドライバ処理，および割り込み処理などの必要な情報を登録する．コアの機能は，コア呼び出しシステムコールを経て，スケジューラを介して呼び出される．また，割り込みを利用して処理を行う場合は，割り込み発生時にカーネルが割り込み番号に該当するコアの割り込み処理を呼び出す．

3.2 コア呼び出しの設計

プロセスがコア呼び出しを行う際の動作概要を図 4 に示す．

コアは，機能を他プロセスから利用可能にするため，初期化時にコア登録システムコールにより自身の情報を OS に登録する．登録後はそのまま OS 内部で要求待ち状態に移行する．処理要求を行うプロセスは，コア呼び出しシステムコールにより処理要求の登録と処理対象コアの起動を行い，コアの処理が終了するまで処理待ち状態で待機する．プロセス切り替え後にコアに実行権が移ると，コアは要求内容を取得し，コアのドライバ処理を呼び出す．ドライバ処理が終了するとコアは要求元プロセスを起動し，再び要求待ち状態で待機する．そしてプロセス切り替え後に要求元プロセスに実行権が移ると，要求元プロセスは要求を削除し，呼び出し元に復帰する．

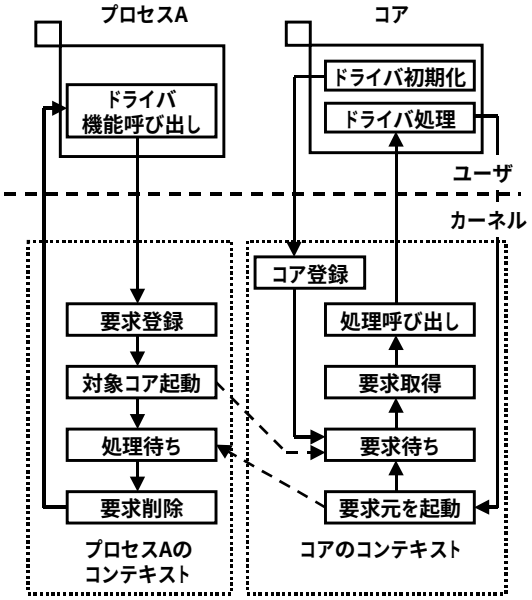

図4 コア呼び出しの動作概要

3.3 割り込み処理の設計

割り込み処理の実現方式としては次の 2 つの方式が考えられる．

(方式 1)

割り込みを処理処理するコアが実行権を得るまで待ち，コアのコンテキストで割り込み処理を実行する．つまり，割り込み発生時にコアへの割り込み処理要求を登録し，コアの待ち状態を解除する．その後，コアのコンテキストに切り替わった時に割り込み処理要求を確認し，割り込み処理を実行する．

(方式 2)

割り込み発生時に，即時割り込み処理を呼び出す．つまり，割り込み発生時に，仮想空間を割り込み処理を実行するコアの仮想空間に切り替え，コアの割り込み処理ルーチンを呼び出す．

2 つの方式の比較を表 1 に示す．方式 1 では，割り込み発生からコアのコンテキストに切り替わり，割り込み処理を実行するまでに遅延が生じる．そのため，割り込み要求が複数回発生した場合の要求管理や，コアのコン

表 1 割り込み処理の実現方式の比較

	方式1	方式2
割り込み処理実行時期	実行が遅延	実行が即時
他プロセスへの影響	少ない	性能低下の可能性あり
処理の複雑さ	複雑	単純

テキストにおける割り込み処理呼び出し処理を行う必要があるため，処理が複雑化する．しかし，OS のプロセス制御の枠組み内で動作するため，他プロセスへの影響は最小限に抑えられる．一方，方式 2 では，割り込み発生時に即座に仮想空間を切り替えて割り込み処理を実行するため，割り込み発生から割り込み処理実行までの遅延は少ない．また，割り込み発生時に即座に関数呼び出しで割り込み処理を呼び出すため，処理が単純である．しかし，Pentium4 プロセッサでは仮想空間切り替え時に TLB のフラッシュが行われるため，仮想空間を切り替えて割り込み処理を行った後，割り込み発生前に走行していたプロセスの処理において TLB のミスヒットが発生し，性能が低下するおそれがある．

以降では，割り込み処理実行の即時性と処理の単純さを優先し，方式 2 を用いた実現について述べる．

3.4 実現方式

3.4.1 コア ID

それぞれのコアは，受け持つサービスの種類を特定するためのコア ID を持つ．コア ID は 32 ビットの整数値であり，図 5 に示す形式をとる．それぞれのフィールドについて以下に説明する．

(1) 処理内容

処理内容は，read, write のようなコアに要求する処理の内容を示す．コア登録処理時はコアが請け負う処理の種類を表す．

(2) 処理種別

処理種別は，ディスク入出力やメッセージ通信など，デバイスやコアの種類を表す．

(3) 処理指定子

処理指定子は，UNIX 系 OS におけるデバイスのマイナー番号に相当し，どのデバイスがどのように処理を行うかといった処理の詳細の指定に利用する．

3.4.2 インタフェース

表 2 にコア関連システムコールのインタフェース形式を示し，詳細を説明する．

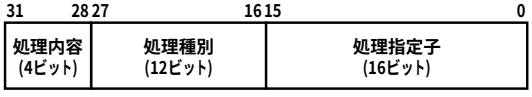

図5 コア ID の形式

(1) registcore()

registcore()は，コア呼び出しを行う際に必要な情報をコア自身がカーネル内に登録する操作である．registcore()を発行したコアは，そのままカーネル内で待ち状態に移行する．このとき，登録するコア ID の処理内容と処理指定子は，コアが請け負う処理内容および機能を示す．これらのビットをすべて 1 にすることにより，ある処理種別におけるすべての処理を請け負うことを示す．ifunc は，コアがデバイスから割り込みを受け付ける際の割り込み処理ルーチンのアドレスである．ufunc は，callcore()による要求を処理するルーチンのアドレスである．ifunc と ufunc の形式を表 3 に示す．また，retval は，callcore()により要求された処理の戻り値を格納するメモリ領域のアドレスである．

(2) callcore()

callcore()は，プロセスからコアに対して処理要求を行う操作である．このとき指定したコア ID に基づいて，カーネルは要求先コアを決定する．引数バッファ buffp は，要求先コアに与えるデータを格納したメ

表 2 コア関連インタフェース形式

インタフェース	引数
registcore()	unsigned int coreid; 　登録するコアID int *ifunc; 　割り込み処理へのポインタ int *ufunc; 　ドライバ処理へのポインタ void *retval; 　戻り値バッファへのポインタ
callcore()	unsigned int coreid; 　要求先コアのコアID void *buffp; 　引数バッファへのポインタ

表 3 登録する関数の形式

インタフェース	引数
ifunc()	unsigned int itnrid; 　割り込み識別子 unsigned int coreid; 　コアID
ufunc()	unsigned int coreid; 　コアID void *buffp; 　引数バッファへのポインタ

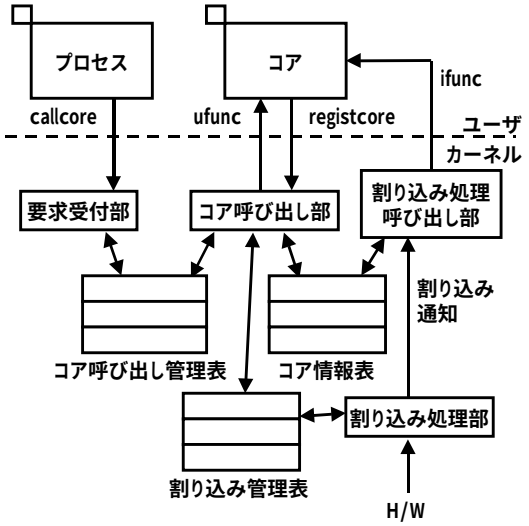

図6 実現方式の構成

モリ領域である.

3.4.3 構成

図 6 に実現方式の構成を示し，それぞれのモジュールの処理概要を以下に示す.

(1) コア登録処理

コアは, 対応するデバイスおよびコア自身の初期化処理の最後に registcore システムコールによりカーネル内にコア自身を登録する．このとき，コア ID はコアの種類や請け負う操作を表し, コア呼び出し部はコアの情報を表4に示すコア情報表に登録する. 割り込み処理を登録する場合は, 割り込み処理部に対して coreid と割り込み番号の対応を登録する．最後に，登録されたコアを要求待ち状態に移行させる.

(2) 要求受付処理

要求受付部は，callcore システムコールによりコアへの処理要求を受け付ける. 要求受付部は， callcore の引数として渡されたコア ID から要求先コアを特定し，要求元プロセスの情報とともにコア呼び出し管理表に登録する．このとき，引数バッファとして与えられたメモリ領域を複写する. コア呼び出し管理表の形式を表 5 に示す．次に，要求先コアの要求待ち状態を解除し, 要求元プロセスを処理待ち状態に移行させる. 要求元プロセスの処理待ち状態が解除されると, コア呼び出し管理表に設定された戻り値を callcore システムコールの戻り値として設定し, 要求をコア呼び出し管理表から削除する.

表 4 コア情報表のエントリ形式

メンバ	内容
pid	コアのPID
coreid	コアID
*intr	割り込み処理関数へのポインタ
*ufunc	ドライバ処理関数へのポインタ
*retval	戻り値を格納するバッファへのポインタ

(3) コア呼び出し処理

コア呼び出し部は, コア呼び出し管理表とコア情報表を参照し, 適切なコアの処理を呼び出す. コアの要求待ち状態が解除されコアのコンテキストに切り替わった時, コア呼び出し部はコア呼び出し管理表を参照し，未処理の要求を取り出す．次に, コア ID を元にコア情報表との対応を取り, 登録された ufunc を実行する．ここで ufunc は，ufunc 内で処理が終了した場合は処理完了を示す値を返し, 終了しなかった場合は処理中を示す値を返す. このときの ufuncの戻り値に応じて以下の2つの処理に分岐する.

(場合 1) 処理終了値を返す場合

コアの処理は終了しているので, コアへの要求の終了処理を行う．つまり，コア呼び出し管理表に処理の戻り値を設定し, 要求元プロセスの処理待ち状態を解除し, コアは再度処理要求を確認後, 要求が登録されていなければ要求待ち状態に移行する. 要求が登録されていれば再びコア呼び出し処理を行う.

(場合 2) 処理未終了値を返す場合

コアの処理は終了していないので, 要求元プロセスは処理待ち状態のまま, コアも処理中の状態で待ち状態に移行する. 以降の処理は割り込みを契機として実行する.

表 5 コア呼び出し管理表のエントリ形式

メンバ	内容
source pid	要求元プロセスのプロセスID
coreid	要求先コアのコアID
*buffp	コア処理の引数バッファへのポインタ
flag	コアの状態を表すフラグ
retval	コア処理の戻り値

表 6 割り込み管理表のエントリ形式

メンバ	内容
intrtype	割り込み種別
coreid	コアID

(4) 割り込み処理

割り込み処理部には, コア登録時にコア ID と割り込み番号の対応が登録される. 割り込み管理表の形式を表 6 に示す. 割り込みが発生した時, 割り込み処理部は必要な前処理を行った後, 割り込み管理表を参照する. 割り込み番号に対応するコア ID が登録されている場合はコア ID を割り込み処理呼び出し部に通知する.

(5) 割り込み処理呼び出し処理

割り込み処理呼び出し部は, 通知されたコア ID にもとづき, 割り込み処理を実行する. 割り込み処理呼び出しの処理概要を図 7 に示す. 割り込み処理部により割り込みを処理するコアのコア ID が通知されると, 仮想空間を処理対象コアの仮想空間に切り替え, コア情報表に登録された ifunc を関数呼び出しで呼び出す. コアの割り込み処理終了後は仮想空間を元に戻し, 割り込みから復帰する. このとき, ifunc は戻り値としてコアに要求された処理が終了したかどうかを示す値を返す. 処理未終了を示す値を返した場合は, そのまま割り込み処理から復帰する. 一方, 処理終了を示す値を返した場合は, ufunc と同様にコアへの要求の終了処理を行う.

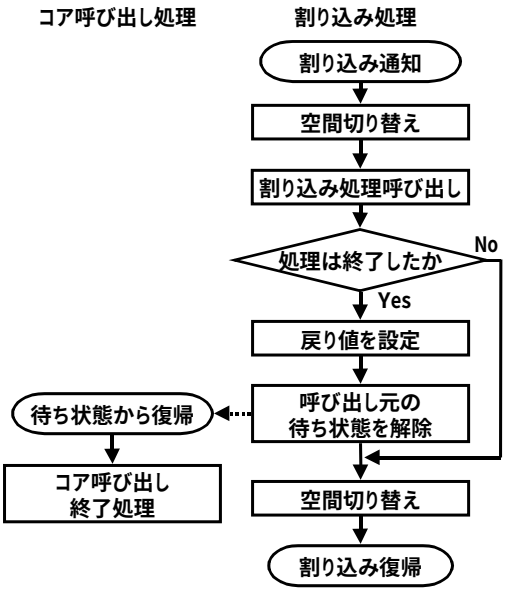

図7 割り込み処理呼び出しの処理概要

4. 評価

コア呼び出し処理を FreeBSD4.3 上に実装し, Pentium4 プロセッサ 2.8GHz を用いてオーバヘッドの評価を行った. 測定はハードウェアクロックを利用し, 以降の各処理時間は 10 回の測定の平均値である.

4.1 基本オーバヘッドの測定

カーネルに組み込まれて動作するデバイスドライバの大半は, 割り込み処理をのぞき, デバイスドライバの機能を呼び出すプロセスのコンテキストで動作する. 一方, 提案方式では, デバイスドライバの機能を呼び出すプロセスとは異なるコンテキストでデバイスドライバが動作する. つまり, カーネルに組み込まれたデバイスドライバを使用する場合と比較すると, プロセスがコア呼び出しを行い, デバイスドライバの機能が実行されるまでの間にプロセススケジューリング, コンテキストスイッチ, および引数バッファ領域複写のためのオーバヘッドが発生する.

呼び出された後, 何も処理を行わずに処理を完了するコアを実装し, このオーバヘッドの測定を行った. コアの引数バッファのサイズは 4KB とする. 何も処理を行わず完了するだけのシステムコールとの比較結果を表 7 に示す. 表 7 より, 割り込み処理を伴わないコア呼び出し処理のオーバヘッドは約 26600 クロックであることがわかる.

4.2 割り込みを伴う処理オーバヘッドの評価

割り込みを利用したデバイスドライバ処理の例として, カーネル組み込み型と提案方式の 2 つの方式を採用したデータ送信のみを行う RS-232C ドライバを実装し, 評価を行った.

ユーザプロセスから 2 種類の RS-232C ドライバに 1 バイトのデータ送信を要求し, 処理終了までの時間を計測した. 図 8 に, それぞれのデバイスドライバにおける 1 バイト

表 7 コア呼び出し処理の基本オーバヘッド

処理内容	処理時間(クロック数)
システムコール	1338
コア呼び出し	27947

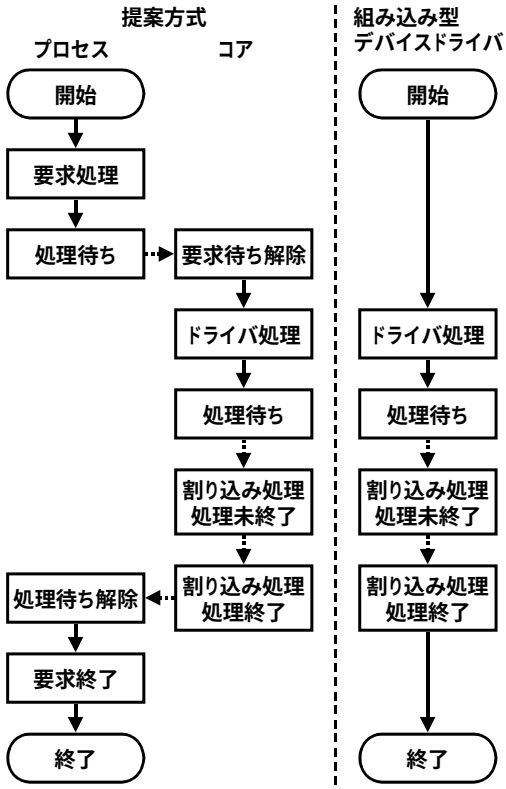

図8 測定用デバイスドライバの処理概要

表 8 デバイスドライバの処理時間の比較

実現方式	処理時間(クロック数)
組み込み型	797633
提案方式	825391

データ送信の処理概要を示す．ここで実装したデバイスドライバは，1 バイトのデータを送信する際に 2 度の割り込み処理を伴う．

測定結果を表 8 に示す．カーネル組み込み型デバイスドライバと提案方式の処理時間の差は約 27800 クロックである．これは，割り込みを伴わないコア呼び出し処理のオーバヘッドとほぼ同じであることから，提案方式の割り込み処理にかかるオーバヘッドは，RS-232C のような低速なデバイスに対しては軽微であるといえる．

5. おわりに

OS の信頼性を向上させるため，応用プログラムの走行モード変更機構を利用し，デバイスドライバをカーネルから分離してユーザプロセスとして構成する手法を述べた．

デバイスドライバの実現のため，デバイスドライバへの要求，および割り込み処理の処理方式を示した．また，提案手法を FreeBSD4.3上に実装し，基本的なオーバヘッドの評価を行った．評価の結果，割り込み処理のオーバヘッドは軽微であるものの，コア呼び出しの要求処理に大きなオーバヘッドがかかることがわかった．

今後の課題は，プロセスとデバイスドライバの通信の高速化が考えられる．また，提案手法の一般的な様々な事例における検討，および評価が必要である．

<参考文献>

[1] 横山和俊, 乃村能成, 谷口秀夫, 丸山勝巳, “応用プログラムの走行モード変更機構の評価,” 情報処理学会研究報告 2005-OS-100, Vol.2005, No.79, pp.49-56, Aug. 2005.

[2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chamvers, “Extemsibility, safety, and performance in the SPIN operating system,” Pro. Of 15th ACM symposium on Operating systems Principles, pp.267-284, Colorado, Dec. 1995.

[3] 前田, 住井, 米澤, “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,” 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予稿集, March, 2003.

[4] 佐藤, 安田, 中村, 多田, “カーネルウェア : アプリケーションプログラム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.SIG 11(ACS7), pp.248-256, Oct. 2004.