2005－OS－100（7）
2005／8／3

応用プログラムの走行モード変更機構の評価

横山和俊[†,††]　乃村能成[†]　谷口秀夫[†]　丸山勝巳[‡]

あらまし

システムコールの処理は，プロセスの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．ここでは，プロセスが AP を実行する走行モードを，プロセス走行中に自由に変更する手法の実現と評価について述べる．提案手法は，プロセス走行中に，ユーザモードで AP を実行するプロセスを，スーパバイザモードで AP を実行するプロセスに変更できる．また，この逆も可能である．スーパバイザモードで AP を実行するプロセスでは，AP が OS のシステムコール処理を関数呼び出しの形態で使用できるため，オーバヘッドを削減できる．さらに，実装評価により，提案手法はシステムコールの処理を 800〜900 クロック短くできることを示す．

Evaluation of dynamic running mode switch of application programs

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

Abstract

System-call is a high cost operation because it causes a running mode switch of a process. This paper presents an implementation and evaluation of a dynamic running mode switch mechanism of application programs. Our proposed mechanism has a characteristic of on-demand and on-the-fly mode switching of a process from user mode to supervisor mode and vice versa. While a process is in the supervisor mode, it is able to emit a system-call as a corresponding simple function call. Thus, it eliminates the overhead of a conventional trap-based system-call procedure. The evaluation shows that our implementation runs 800-900 clocks faster than a conventional system-call based implementation does.

1. はじめに

近年, 計算機で扱うサービスの規模が増大しており, 高負荷なサービスを効率良く実行することが求められている．また, 計算機で扱うサービスの重要性も高まっており, 例えばトランザクション処理では，24 時間無停止でサービスを提供することが望まれている．一方, サービスを提供する形態も大きく変化している. サービスの提供形態は, サービス毎に専用のシステムを構築する形態から, データセンタのように, 同一システム上に複数種類のサービスを実現する形態が出現している．このようなサービスの提供形態では，負荷の高いサービスが時刻とともに変化する. つまり, 変化する負荷に応じて, 処理の効率化が必要となるサービスが変化する．

†　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

サービスを実現する応用プログラム（以降，AP と略す）は，プロセスとして実行される．従来のオペレーティングシステム（以降，OS と略す）では，プロセスが AP を実行する時の走行モードはユーザモードであり，OS を実行する時の走行モードはスーパバイザモードである．

システムコールは，AP 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，AP がソフトウェア割込み命令を発行することで実現される. OS 処理からの復帰は, OS がプロセッサに対して，割り込みからの復帰命令を発行することで実現される. これらの処理は, 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. 特に，トランザクション処理のように短時間で多数のシステムコール発行を伴うサービスでは無視できないオーバヘッドである．

システムコール処理のオーバヘッドを削減する手法として，プロセスが，AP をスーパバイザ

モードで実行する手法がある[1,2,3]. 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 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の変更は, プロセス停止を伴いサービスが中断してしまう．

ここでは，プロセスが AP を実行する走行モードを, 走行中に変更することを可能にする動的走行モード変更機構 DMSM（Dynamic running Mode Switch Mechanism）の実現と評価について述べる．

2. 動的走行モード変更機構

2. 1 概要

DMSM は，AP をユーザモードで実行するプロセス（以降，Umode プロセスと呼ぶ）を，任意の時点で，AP をスーパバイザモードで実行するプロセス（以降，Smode プロセスと呼ぶ）へ変更することができる．逆に，任意の時点で Smode プロセスを Umode プロセスに変更することも可能である．

DMSM を用いたプロセス実行の概念を図１に示し，以下に説明する．プロセスは，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， Smode プロセスに変更する．逆に，switch_user システムコールは， Umode プロセスに変更する．Umode プロセスは，ソフトウェア割込みによるシステムコール（以降，割込み型システムコール呼ぶ）を用い，Smode プロセスは，関数呼出しによるシステムコール（以降，関数型システムコールと呼ぶ）を用いる．このシステムコールの形態を, 共存型システムコール形態と呼ぶ.

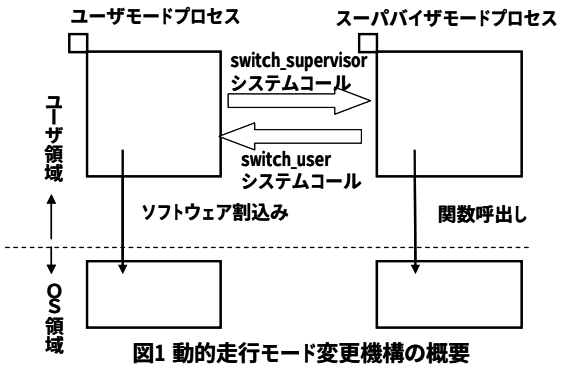

図1 動的走行モード変更機構の概要

Smode プロセスは，システムコールを関数型システムコールで使用するため, システムコールのオーバヘッドが削減できる．さらに，Smode 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メモリに常駐する. これにより，Umode 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するものの, Smode プロセスでは発生しないので, さらなる効率化が期待できる.

2. 2 要求条件

DMSM の実現に際して，以下の要求がある．

（要求１）プログラムの互換性

DMSM を用いたプロセス実行では，同じプロセスが, 割込み型と関数型のシステムコールを使い分ける．つまり，同一 AP により，両方のシステムコール形態が利用できる必要がある．また，Umode プロセスとして開発された豊富な既存 AP を再利用することが望まれる．特に，近年のシステム開発ではパッケージソフトウェアを用いることが多い. パッケージソフトウェアは, 通常，ソースコードが提供されないため，既存 AP のソースコードを修正せずに再利用できることに加え, バイナリコードで提供される既存 AP に対応することが求められる.

（要求２）他プロセス走行への影響の抑制

OS の重要な機能の一つにプリエンプションがある. プリエンプションは, 割込みを契機として, 優先度の高いプロセスにプロセスを切り替える機能である．DMSM を用いたプロセス実行では, Smode プロセスは, AP をスーパバイザモードで実行する．このため，スーパバイザモード走行時のプリエンプションを提供していない OS では, Smode プロセスが AP 実行時の場合でも，処理を継続してしまう．このため，Umode プロセスと同様に，Smode プロセスが AP を実行している場合についてプリエンプションを可能にし, 必要ならば他のプロセスに切り替える必要がある.

（要求３）自由な走行モード変更

DMSM では，サービスを継続するため，プロセスを停止することなく, プロセス走行中の自由な走行モード変更を実現する. この実現において,

Umode プロセスは割込み型システムコールを必ず使用し，Smode プロセスは関数型システムコールを必ず使用することを保証する必要がある．

以上 3 つの要求に加え, これらの要求の解決に際し，OS への変更量の最小化と局所化を考慮する必要がある．

3. 動的走行モード変更機構の設計

3. 1 実現方式

ここでは，DMSM の基本的な実現方式について述べる．

先ず, 共存型システムコール形態の実現方式を図 2 に示し，詳細を説明する．

（1）共存制御領域

プロセスは，Umode プロセスか Smode プロセスであるかを示すモードフラグを持つ．また，OS の関数型システムコールを受け付ける関数受付部のアドレスを持つ. これら 2 つの領域を共存制御領域と呼ぶ. 共存制御領域は，システムコールライブラリにより参照される．この領域は，Umode プロセスの場合，ユーザモードで参照される．一方，Smode プロセスの場合，スーパバイザモードで参照される．このため，両方の走行モードで参照可能なユーザ領域上に実現する．

（2）システムコールライブラリ

AP にリンクされるシステムコールライブラリは, 割込み型と関数型の両方のシステムコール形態を提供する．具体的には，共存制御領域を参照し, プロセスが Umode プロセスであるか Smode プロセスであるかを判定する．Umode プロセスの場合，ソフトウェア割込み命令（Pentium4 プロセッサの場合，int 命令）を発行する．Smode モードプロセスの場合，関数型受付部を，関数呼出しの形態（Pentium4 プロセッサの場合，call 命令）で呼び出す．

（3）システムコール受付部

OS には，割込み受付部と，関数受付部の 2 つがある．両者とも，システムコール番号により，該当するシステムコール処理を呼び出す. 割込み受付部は，割込みからの復帰命令（Pentium4 プロセッサの場合，iret 命令）でシステムコールライブラリへ復帰する．関数受付部は，関数呼出しの復帰命令(Pentium4 プロセッの場合 ret 命令)によりシステムコールライブラリへ復帰する．

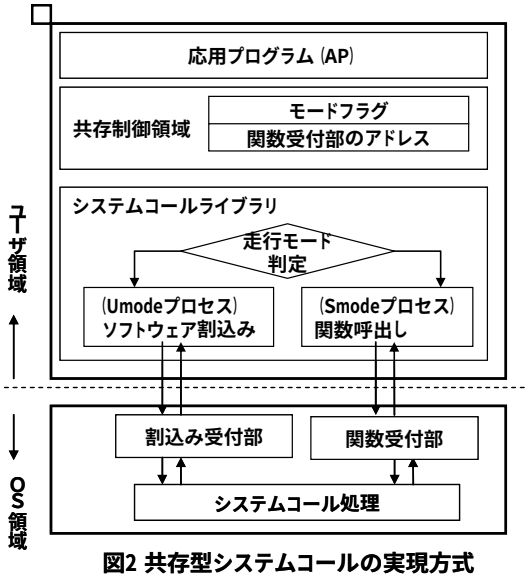

図2 共存型システムコールの実現方式

表1 モード変更システムコールのインタフェース形式

インタフェース	引数
switch_supervisor(pid)	long pid ; /* 変更対象プロセスのプロセスID */
switch_user(pid)	long pid ; /* 変更対象プロセスのプロセスID */

次に, モード変更システムコールの実現方式を説明する. 表 1 にモード変更システムコールを示す．switch_supervisor(pid)は，pid で指定されたプロセスを Smode プロセスに変更する. 一方, switch_user(pid)は，pid で指定されたプロセスを，Umode プロセスに変更する．pid で指定されたプロセスを変更対象プロセスと呼ぶ．pid が 0 の場合は，自プロセスを意味する．すなわち，pid が 0 の場合，モード変更システムコールを発行したプロセスの走行モードが変更される．

図 3 は，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している．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 実行に移行する時に，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．

モード変更システムコールは, 主に 2 つの処理を行う．モードフラグを変更する処理と，カーネルスタック上の CS レジスタの値を変更する処理である．switch_supervisor システムコールでは，モードフラグと CS レジスタ値をスーパバイザモ

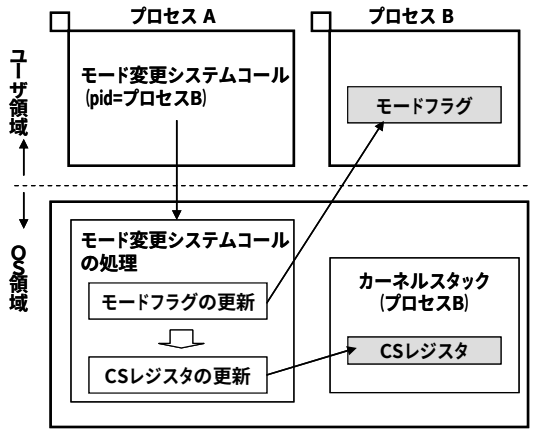

図3 モード変更システムコールの基本処理

ードに変更する．これにより，プロセスBがOS実行から AP 実行に移行する際に，プロセスがAP を実行する時の走行モードが，スーパバイザモードとなる．逆に，switch_user システムコールは，モードフラグとカーネルスタック上の CS レジスタ値をユーザモードに変更する. これにより, プロセスが AP を実行する時の走行モードが, ユーザモードとなる.

3．2 要求への対処

3．2．1 プログラムの互換性

最初に, 同一の AP で割込み型と関数型の両方のシステムコール形態に対応する方法について考察する．3.1 節で示した実現方式では，走行モードを判定し, システムコール形態を選択する処理は, すべてシステムコールライブラリで行なっている．AP は，Umode プロセスと Smode プロセスの両者で, システムコールインタフェースを呼び出すだけである．つまり，システムコールライブラリの修正によって, 同一の AP で割込み型と関数型の両方のシステムコール形態が実現できる.

次に, 既存 AP との互換性への対処を説明する. 既存 AP を用いたプロセスに対する DMSM の適用の様子を図 4 に示す. 既存 AP により生成されたプロセスとは別に, プロセスの走行モードを変更する管理プロセスを新規に構築する. 管理プロセスは，変更対象プロセスの pid を指定して，モード変更システムコールにより, 変更対象プロセスの走行モードを切り替える．これにより，変更対象プロセスを実現する既存 AP を修正するこ

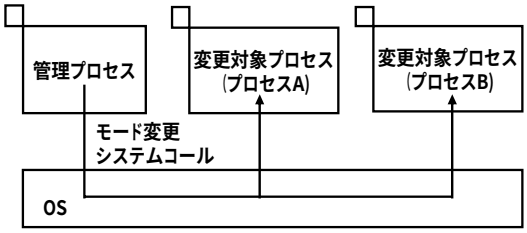

図4 既存APを用いたプロセスの走行モード変更処理

となく，走行モード変更を実現できる.

既存 AP の再コンパイルの必要性は, 共存制御領域の確保方法と AP のリンク形態により異なる．共存制御領域の確保方式として，以下の 2 つがある.

（方式 a）AP が走行する前に初期プログラム（UNIX 系 OS の場合 crt0.s）により，明示的にメモリ領域を確保するシステムコール（sbrk）や static 変数定義により領域を確保する．また, 確保した領域のアドレスを OS に通知する.

（方式 b）プロセス生成時（exec）時に，プロセスの AP 領域の固定アドレスに OS が領域を確保する.

方式 a は, AP を再コンパイルする必要がある. また, 確保した領域を OS に通知するシステムコールを追加する必要がある．さらに，OS がプロセス毎に異なる制御領域のアドレスを管理するため, OS の制御が複雑化する. 一方, 方式 b は, OS の exec 処理に修正を加える必要があるものの，AP の再コンパイルが不要である.

一方，AP のリンク形態では，動的リンクと静的リンクの 2 つの場合がある. 動的リンクを利用する AP の場合, システムコールライブラリが実行時にリンクされるため, 再コンパイルの必要がない．一方，静的リンクの場合，AP を再コンパイルし, システムコールライブラリを再リンクする必要性がある.

3．2．2 他プロセスへの影響の抑制

ここでは, スーパバイザモード走行時のプリエンプションを提供していない OS での対処を述べる．従来の多くの OS では，割込み発生時の走行モードにより, プリエンプションの必要性の有無を判定している．つまり，ユーザモードの場合プリエンプションを行い, スーパバイザモードの

場合はプリエンプションを行わない．一方，Smode プロセスは, AP 実行時にもスーパバイザモードで走行する．このため，割込み発生時に AP 実行であると判断できず，プリエンプションが行われない．この対処として，スタックポインタの値による判断することが，文献[3]で示されている．DMSM でのプリエンプションの実現についても，同じ手法で対処した．

3．2．3 自由な走行モード変更

プロセスの走行中に，AP の観点から自由に走行モードを変更するためには, 走行モードの変更が可能なプロセスの走行状態を明確化する必要がある．ここで，プロセスの走行状態とは，プロセスが実行しているプログラム部分を意味している．変更対象プロセスは，その時点でどのプログラム部分を実行しているかわからない. このため, プロセスがどのプログラム部分を実行しているかに関わらず, 走行モードと使用するシステムコール形態の間に矛盾が生じないように, 走行モード変更を実現する必要がある．

プロセスの走行状態は, 図 5 に示すように, 以下のように分類できる．

（状態 X）プロセスが AP を実行中である．

（状態 Y1）プロセスがシステムコールライブラリを実行中であり, 走行モードを判定していない状態である．

（状態 Y2）プロセスがシステムコールライブラリを実行中であり，走行モードを判定し，システムコールを呼び出す直前までの状態である．

（状態 Y3）プロセスがシステムコールライブラリを実行中であり，システムコールが終了し，AP へ復帰するまでの状態である．

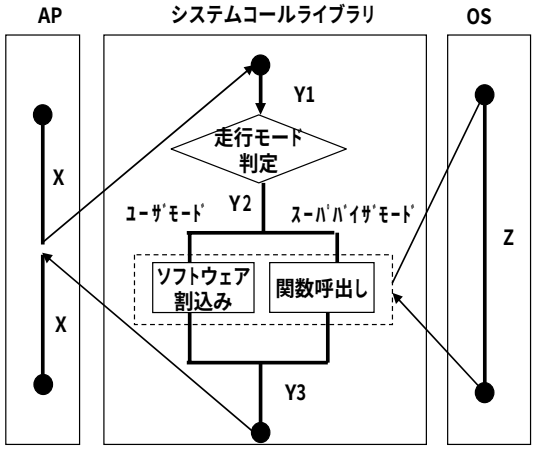

図5 プロセスの走行状態

（状態 Z）プロセスが OS を実行している状態である．

走行モード変更ができないのは，状態 Y2 の場合である．状態 Y2 において，走行モードを変更すると, 走行モードと使用するシステムコール形態の間に矛盾が生じる．例えば，Smode プロセスで，関数型システムコールを呼び出す直前に Umode プロセスに変更された場合，保護された OS 領域にユーザモードでのアクセスが生じ，アクセス権違反となる．状態 Y2 以外の状態は，走行モード判定が再度行われるため矛盾が発生せず，矛盾のない変更が可能である．

次に, モード変更システムコールの実現方式は, 以下の 2 つがある．

（方式 a）モード変更システムコール内で直ちに変更を行う方式である．つまり，モード変更システムコールを発行したプロセスのコンテキストで変更対象プロセスの走行モードを変更する. 図 6(a)に方式 a の動作の様子を示す．図 6(a)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．この場合，プロセス A のシステムコール処理でプロセス B のモードフラグと CS レジスタ値が変更される．

（方式 b）変更対象プロセスが発行した任意のシステムコールの末尾で, 走行モードを変更する方式である．つまり，変更対象プロセスのコンテキストで走行モードの変更が行われる．図 6(b)に，この様子を示す．図 6(b)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．この場合，プロセス A が発行したシステムコール処理は, プロセスＢの走行モード変更の要求を登録するのみである．プロセス切り替え後, プロセスＢが任意のシステムコールを発行した場合, システムコール処理の末尾で走行モード変更要求の有無を確認する. 走行モード変更の要求されている場合, プロセスＢが自分自身を対象に, 走行モードを変更する．

2 つの方式の比較を表 2 に示す．方式 a では，走行モード変更時に変更対象プロセスが取り得る走行状態は，状態 X～状態 Z のすべてである．方式 a で矛盾のない走行モード変更行うためには，変更対象プロセスの走行状態が，状態 Y2 であるかを判定する必要がある．さらに，走行状態

が状態 Y2 であった場合，変更対象プロセスが他の走行状態へ遷移するまで，走行モードの変更を遅延させる必要がある．この実現のためには，プロセスの走行状態を監視し，状態 Y2 への遷移と状態 Y2 からの遷移を検出する必要がある．このため，OS 処理やモード変更システムコールの処理が複雑化すると共に，OS 処理のオーバヘッドが増加する．

一方，方式 b は，変更対象プロセスがシステムコールを実行している場合に限定できる．つまり，方式 b では，状態 Z の場合に限定でき，矛盾のない走行モード変更を実現することが容易である．しかし，方式 b の欠点として，走行モードの変更時期が，変更対象プロセスがシステムコールを発行するまで遅れることがある．ただし，変更要求が登録され走行モードの変更がなされるまでに発行されるシステムコール数は，高々１回である．このため，多数のシステムコールが発行される AP の処理効率の向上という本来の目的には影響が少ないと考えられる．以上のことより，方式 b が有効である．

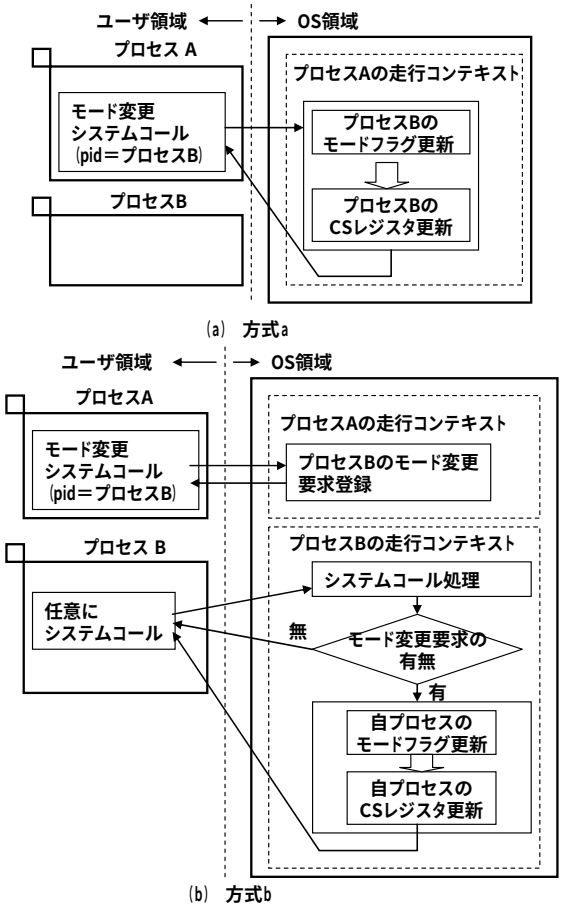

図6 モード変更システムコールの実現方式

表2 システムコール実現方式の比較

	走行モード変更の時期	実現の複雑さ
方式a	切り替えが即時	プロセスの走行状態を監視する必要があるため，複雑.
方式b	切り替えが遅延	プロセスの走行状態が限定されるため，簡易.

4．評価

4．1 評価環境

DMSM を FreeBSD4.3 の OS に実装し，Pentium4 プロセッサ 2.6GHz を用いて評価を行った．共存制御領域の確保方式とモード変更システムコールの実現方式は，両者とも方式 b を実装した．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 回測定した平均値である．

4．2 システムコールの処理時間

DMSM を用いると，プロセスは，関数型のシステムコールを用いることで，処理の効率化が図れる．ここでは，DMSM を用いたシステムコールと従来の割込み型システムコールの処理時間を比較する．評価は，getpid，read，write，sbrk システムコールを用いた．getpid は，OS でのシステムコール処理が最も少ないシステムコールの一つである．また，read，write，sbrk システムコールは，トランザクション処理において頻繁に使用されるシステムコールである．read，write システムコールでは，OS のバッファキャッシュにヒットし，実 I/O は発生しない状態とした．なお，比較対象の従来の割込み型システムコールは，FreeBSD4.3 のシステムコールを用いた．また，測定は，静的リンクと動的リンクの場合について行った．測定結果を図 7 に示す．図 7 より，以下のことが分かる．

（１）4 つのシステムコールすべてにおいて，DMSM（関数型）は，従来の FreeBSD に比べて，システムコールの処理時間が短い．両者の処理時間の差は，約 800～900 クロックである．この時間は，割込み型システムコールの問題点であった走行モードの変更にかかる時間であり，すべての割込み型システムコールで固定的に発生する時間である．つまり，DMSM（関数型）では，す

べてのシステムコールにおいて, 約 800～900 クロックの処理時間が削減できる. 従って, 例えば静的リンクの場合, DMSM（関数型）の処理時間は, getpid では, FreeBSD の処理時間の約 34%であり, 66%の削減を達成している. 同様に, sbrk では約 57%（43%削減）, read では約 80%（20%削減）, write では約 81%（19%削減）である. つまり, getpid のようにシステムコールの処理量の少ないシステムコールほど, 処理時間の削減効果が大きい. 一方, 処理量の多い read や write では, 削減効果が小さい. このため, DMSM は, 処理量の小さいシステムコールを頻繁に用いる AP に対して, 特に有効であるといえる.

（2）FreeBSD と DMSM（割込み型）を比較することにより, 走行モード判定処理に伴う処理時間の増加の程度を見ることができる. 4 つのシステムコールの測定結果から, DMSM（割込み型）の処理時間は, FreeBSD に比べ, 約 0%～2%増加する. しかし, 全体の処理時間に占める割合は小さい. つまり, 走行モード判定処理による処理量の増加は軽微であるといえる.

（3）静的リンクと動的リンクの各場合の処理時間を比較すると, 静的リンクの場合の処理時間は, 約 0%～6%程度短い. これは, 動的リンクの場合, 実行時にアドレスを解決するため, 変数や関数のアドレス参照処理の増加が原因である. なお, 割込み型あるいは関数型であることの影響はない. このため, ソースコードが入手できる既存 AP は, 静的リンクを用いて再コンパイルすることにより, 処理時間を削減できる.

4．3 モード変更システムコールの処理時間

ここでは, モード変更システムコールの処理時間について評価する. 自プロセスを変更する場合について, switch_supervisor と switch_user システムコールの処理時間を測定した. 自プロセスを変更する場合, モード変更システムコールでは, 走行モード変更要求の登録処理と, 走行モードを変更するためのモードフラグと CS レジスタ値の変更が実行される. 図 8 に測定結果を示す. 図 8 より, switch_supervisor の処理時間は約 1200 クロック, switch_user の処理時間は約 1100 クロックである. これらの処理時間は, 処理量の少ない getpid システムコールの FreeBSD の処理時間より短い. つまり, 走行モード変更の処理時

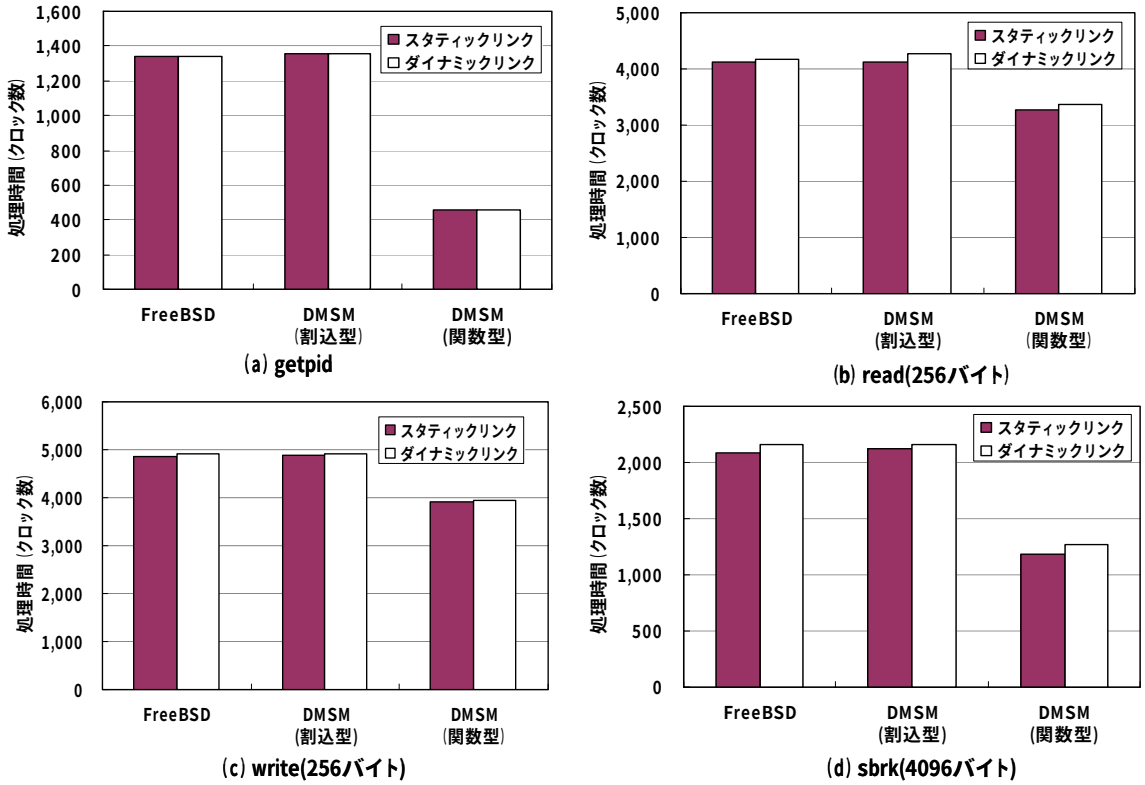

図7 システムコールの処理時間

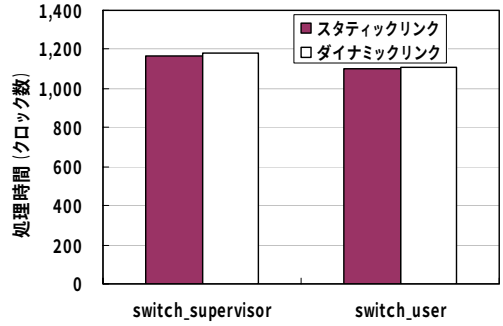

図8 モード変更システムコールの処理時間

間は非常に短い, 多数のシステムコールを発行する AP では, 変更伴う処理時間の増加は軽微であるといえる.

4．4 既存コマンドによる評価

多数のシステムコール処理を伴う既存コマンドに対する DMSM の有効性の検証について述べる. ここでは, ls コマンド例に有効性を考察する.

FreeBSD4.3 の ls コマンドについて，DMSM を適用した. ls コマンドは, 比較的処理が少ない stat システムコールを多用するコマンドである. 測定結果を図 9 に示す. 図 9(a)は，ファイル数 5 の場合について測定した結果であり，図 9(b)は，ファイル数 408 の場合（/usr/bin）である. なお，図 9 は静的リンクの場合である. 図 9 より，次のことが分かる.

（1）DMSM（関数型）の処理時間は，FreeBSD（割込み型）の処理時間に比べ，5 ファイルの場合では約 91%（9%削減），408 ファイルの場合では約 95%（4%削減）の処理時間である. これらにより, 関数型システムコールを使用したことによるシステムコールのオーバヘッド削減の効果が確認できる.

（2）5 ファイルの場合に比べ，408 ファイルの場合の削減効果が小さい理由は, 画面表示などの実 I/O の処理量が増加しているためである.

5．まとめ

システムコールのオーバヘッドを削減するため, プロセスが AP を実行する走行モードを自由に変更できる動的走行モード変更機構の実現と評価について述べた.

提案機構の実現に際し, プログラムの互換性を確保するため, 既存 AP に修正を加えることなく, 割込み型と関数型の両方のシステムコール形態

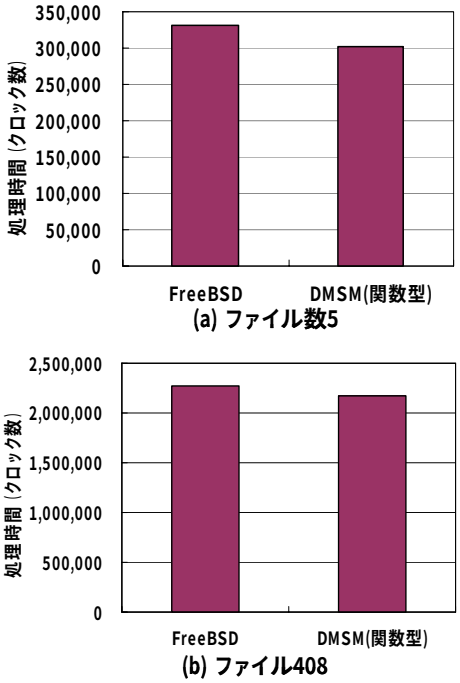

図9 lsコマンドの処理時間

を使用できる実現方式を示した. また, 自由な走行モード変更を可能にするため, 変更可能なプロセス走行状態を明確にし, 変更対象プロセスのコンテキストで走行モードを変更する方式を示した. さらに, 実装評価により, 提案機構はシステムコールの処理時間を 800～900 クロック（Pentium4 プロセッサ 2.6GHz の場合）短くできることを示した. また, 走行モードを変更する処理量は，getpid システムコールの処理より小さく, プロセスの実行時間に与える影響は小さいことを示した.

残された課題として，Web サーバなどのサービス処理で評価することがある.

＜参考文献＞

[1] 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, ” Proc. of 15th ACM Symposium on Operating systems Principles, pp.267-284, Colorado, Dec. 1995.

[2] 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.

[3] 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.