平成29年度 特別研究報告書

メモリ共有を用いた
システムコールの並列化手法

龍谷大学 理工学部 情報メディア学科

学籍番号 ： T140482

氏名 ： 中村 公亮

指導教員 ： 三好 力 教授，芝 公仁 助教

内容梗概

多くのオペレーティング・システムにおいて，プロセスが特権的な処理を行う場合には，システムコールを発行する．システムコールを発行したスレッドは走行モードが特権モードへと遷移し，必要な処理を行うことができる．一方でシステムコールの発行時には，プロセッサの走行モードを特権モードに変更するため，カーネルスタックにレジスタの値を保存したり，Translation Look-aside Buffer がクリアされたりする．これらの原因により，システムコール発行のためオーバヘッドが発生し，アプリケーションのパフォーマンスが低下する．

本論文では，システムコール要求を行うスレッドと処理を行うスレッドを分け，メモリを共有して必要な情報を受け渡すことで上記の問題を解決する方法について述べる．また，処理を行うスレッドを複数用意し，並列にシステムコール処理を行う方法を提案し，その有効性を確認するため評価実験を行う．評価実験の結果，提案手法がシステムコールの処理速度を向上させることを確認した．

目 次

1 はじめに **1**

2 関連研究 **2**

3 提案手法 **3**

3.1 従来型のシステムコール発行 3

3.2 並列システムコール 4

4 動作 **7**

4.1 全体の動作 7

4.2 処理スレッド 7

4.3 要求スレッド 10

5 評価 **11**

5.1 並列システムコールのオーバヘッド計測 11

5.2 並列システムコールによる要求処理時間 13

6 おわりに **16**

謝辞 **17**

参考文献 **18**

1 はじめに

プロセスが動作するとき，ファイル I/O やメモリの確保，ソケット通信などの処理を行うことがある．しかし，これらの処理はセキュリティ上の観点から非特権モードで実行されるプロセスがそのまま行うことはできない．オペレーティング・システムでは，通常，非特権モードで実行されるプロセスが特権的な処理を行うために，システムコールが用意されている．システムコールを発行した場合，プロセッサは特権モードへと移行し，カーネル内で必要な処理を行うことができる．

一方でシステムコールの発行時には，プロセッサの走行モードを特権モードに変更するため，カーネルスタックにレジスタの値を保存したり，パイプラインのフラッシュが起こる．また，Translation Look-aside Buffer (TLB) と呼ばれる，CPU が仮想アドレスと論理アドレスとを対応させた情報を一時的に保管しておくバッファメモリや，命令キャッシュにおいても特権モードへの移行による汚染が起こる．また，システムコール発行後はスレッドの処理が中断され，システムコールから復帰するまで別の処理が行えなくなる．これらの原因により，システムコール発行によるオーバヘッドが発生し，アプリケーションのパフォーマンスが低下する．

本論文では，システムコールによるオーバヘッドを削減するため，システムコールの処理を行うスレッドと，要求を行うスレッドに分ける手法を提案する．提案手法では，カーネル内でシステムコールの処理を行うスレッドとその要求を行うスレッドは，共有メモリを使用してシステムコール番号や引数，返り値などの受け渡しを行う．また，この機構を使えばユーザスレッドはシステムコール発行後ブロックされる事なく他の動作を行う事ができる．従来型のシステムコール発行より柔軟な発行処理が行え，より効率的にシステムコールを実行できる．

以下，第 2 章では関連研究について概説し，第 3 章では提案手法とその目的について述べる．第 4 章では提案手法の動作について述べ，第 5 章では実際に実装を行なった提案手法を用いた実験とその結果に対しての考察について記述する．第 6 章では本稿のまとめを行う．

2 関連研究

システムコールのオーバヘッドを削減する手法については，これまでも幾つかの手法が提案されている．

文献 [1] では，カーネル空間とユーザ空間において同期的なプロセッサ例外を用いずにシステムコールを発行する仕組みを提案している．共有メモリ上にエントリを作成し，ユーザ空間とカーネル空間で別スレッドが非同期的に連携しシステムコールを発行する手法で，FlexSC (Flexible System Call) と呼ばれる．FlexSC では，syscall page と呼ばれる，例外を用いないシステムコール発行のためのエントリを共有メモリ上に用意し，システムコール番号や引数などの必要な情報を格納しておく．ユーザ空間のスレッドは共有メモリ内で使用されていないエントリを探し，適切な値を設定する．ユーザ空間のスレッドが責任を持たなければならないのはこの様な値の設定と，エントリ内の終了ステータスを読み取ることで，実行が完了したかどうかを確認する事だけである．カーネル空間では syscall thread と呼ばれるカーネルスレッドが動き，これが syscall page からリクエストを取得して，実行を行う．

文献 [2] では，個別に発行するシステムコールを複数個まとめて発行する手法により，ユーザ空間からカーネル空間へのコンテキストスイッチ回数自体を削減する手法を提案している．提案手法では新たなシステムコールとして，発行する個数，システムコール番号や引数などが格納された構造体を引数にとる，マルチコールを実装している．よって，任意の組み合わせのシステムコールを複数個まとめて実行できる．しかし実際は，一つのシステムコールの結果が処理分岐の条件になっていたり，コード上でシステムコールの組みが連続していない場合には使用できない．文献 [2] では，マルチコールを使用可能なコード上の部分をクラスタリング可能領域と呼び，条件を述べている．一方で従来型の Linux カーネルの設計に準拠し，同期的なシステムコール発行になっている．そのため，マルチコールから復帰した時点で結果を受け取っている．

3　提案手法

本章では，従来型のシステムコール発行と，メモリ共有を用いたシステムコール発行 (以降，並列システムコール) について述べる．

3.1　従来型のシステムコール発行

プロセスがシステムコールを発行すると，CPU はカーネルモードに切り替わり，カーネルコードの実行を始める．Linux カーネルにおけるシステムコールの発行は，CPU の種類によって異なるが，どの場合でもアセンブリ言語で書かれたカーネル内のシステムコールハンドラを呼び出す．例えば，Intel 製の 64 bit CPU では，システムコールの発行に SYSCALL という CPU 命令が使用され，SYSCALL は特権モードでシステムコールハンドラを起動する．これは，rip レジスタに IA32_LSTAR MSR からシステムコールハンドラのアドレスをロードすることで行われる．また，復帰には SYSRET という CPU 命令が使用され，CPU の走行モードをユーザモードに変更してから，プロセスの実行が再開される．これは，rcx レジスタから rip レジスタに実行が再開されるべき場所のアドレスをロードすることで行われる [3]．

プロセスとカーネル間におけるシステムコール番号や引数の受け渡しは，通常の関数の引数のようにスタックへ積むことはせずに，CPU レジスタに書き込む．カーネルは CPU レジスタ上の引数をカーネルモードスタックに書き込むことで，引数を受け取る．また，引数をレジスタ経由で渡すため，引数の数が 6 個以下であること，INTEGER クラスまたは MEMORY クラスであることの制限が設けられている [4]．

図 1 は，システムコールを発行したアプリケーションプログラム，対応するラッパールーチン，システムコールハンドラ，システムコールサービス関数の関係を表す．矢印は関数間の実行の流れを表し，ユーザモードとカーネルモードの間にあるソフトウェア割り込みが，上述の SYSCALL CPU 命令と SYSRET CPU 命令にあたる．Linux カーネルのシステムコールハンドラは，次のような処理を行う [5]．

1. カーネルモードスタック上に，ほぼ全てのレジスタの内容を退避する
2. システムコールサービス関数と呼ばれる，C の関数を呼び出して，システムコールを処理する

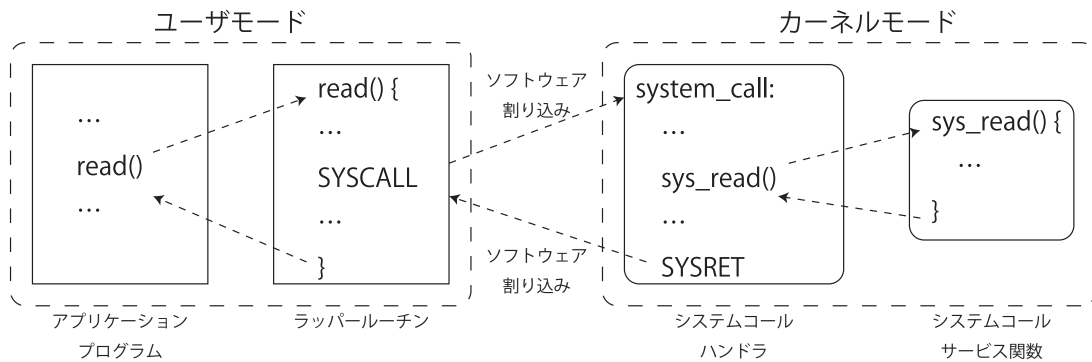

図 1: システムコール発行

3. カーネルモードスタック上に退避してあったレジスタ値を読み込み，CPU をカーネルモードからユーザモードに切り替える

カーネルは，各システムコール番号をサービス関数と関連づけるために，システムコール分岐テーブルを利用する．このテーブルは sys_call_table 配列に格納されている．sys_call_table の n 番目のエントリには，システムコール番号 n のサービス関数のアドレスが格納されている．システムコールハンドラは eax レジスタから受け取ったシステムコール番号を基にして，sys_call_table 内の対応する番号の関数を呼び出す．

3.2 並列システムコール

並列システムコール (Parallel System Call) を実現するため，Linux カーネルに新たなシステムコールを追加する．以下にそれらを記述する．

1. 処理スレッドとして動作するためのシステムコール (以降，parasc_proc システムコール)

2. システムコール要求を置いた事を通知するためのシステムコール (以降，parasc_wake_up システムコール)

3. 要求の処理完了を待つためのシステムコール (以降，parasc_wait システムコール)

提案手法の全体図を図 2 に示す．提案手法では，図中の並列システムコール処理機構を提供し，それはカーネル内にある．ユーザプロセス内には，処理スレッドと要求

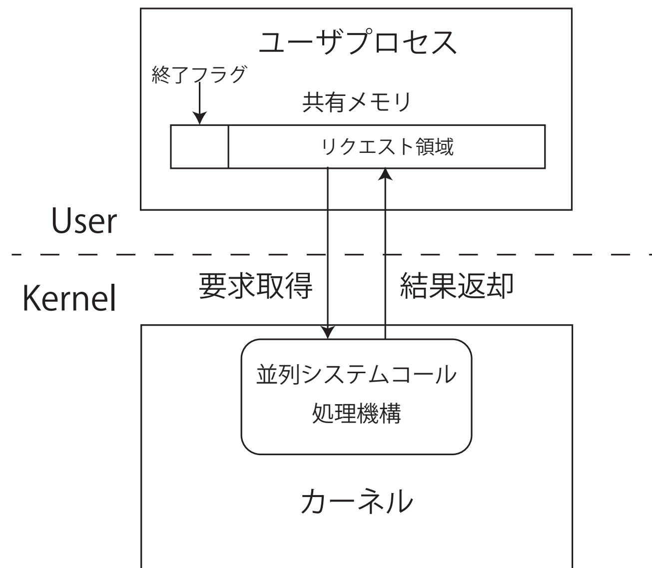

図 2: 提案手法の全体図

や返り値の受け渡しを行う共有メモリがある．共有メモリには，要求をまとめた構造体を置くリクエスト領域と，処理スレッドに対して処理を終了するよう通知する終了フラグがある．それらは，先頭 1 ワードに終了フラグがあり，それ以外がリクエスト領域となる．

並列システムコール処理機構の内部では，各プロセスに属するスレッドが動作する．それらは，カーネル内でシステムコール要求を処理するための専用スレッドである．(以降，処理スレッド)．システムコールを要求するスレッド (以降，要求スレッド) は，処理スレッドと共有メモリを介して要求を置いたり処理スレッドの復帰を通知したりする．提案手法においては，処理スレッドと要求スレッドは多対一，一対多，一対一，多対多とすることができる．要求の受け渡しには，システムコール発行に必要な情報をまとめた構造体 (以降，リクエスト構造体) を用いる．そのメンバを表 1 に示す．要求スレッドはリクエスト領域にそれを書き込むことによって，システムコール要求とする．また，処理スレッドは終了フラグの値を参照し，parasc_proc システムコールから復帰する．

この手法ではシステムコール要求を行うスレッドとその処理を行うスレッドが異なる．また，システムコールの要求は共有メモリに置かれ，処理はそれを参照するだけ

表 1: リクエスト構造体のメンバと役割

メンバ	役割
status	要求，処理中，完了など要求の状態を示す
sysnum	要求するシステムコール番号を格納する
args_num	引数の数を指定する
args[6]	それぞれに引数を格納する
ret_val	処理が完了したときの返り値を格納する

であるため，ソフトウェア割り込みが省略されプロセスが中断されない．さらに，処理スレッドを生成するときに複数個のスレッドを生成することで，複数個の要求を並列に処理できる．

4　動作

本章では，前章の構成を基にする全体の動作と，処理スレッド，要求スレッドの動作について述べる．

4.1　全体の動作

図 3 に提案手法の全体動作の図を示す．図中には，プロセス A とプロセス B という 2 つのプロセスがある．プロセスは内部にカーネル内の処理スレッドと共有可能な，共有メモリを持つ．各スレッドは共有メモリにシステムコールの要求を置く．並列システムコール処理機構は，3.2 節で述べたように処理スレッドを持つ．その役割は図 1 中のシステムコールハンドラにあたり，共有メモリに置かれた要求を基にサービスルーチンを呼び出す．また，従来型のシステムコール発行の手順とは異なり，システムコール番号や引数は共有メモリに置かれているので，メモリアクセスだけでそれらを取得できる．処理スレッドは，リクエスト領域の先頭から順に要求が置かれていないかを探索する．要求が見つかれば，処理スレッドは sys_call_table の n 番目のエントリに格納されているアドレスを基に，システムコール番号 n のサービスルーチンを呼び出す．サービスルーチンから結果を受け取った処理スレッドは，共有メモリに結果を返却する．システムコール要求を行ったスレッドは，共有メモリから結果を取得することができる．

共有メモリと処理スレッドは，図 3 に示したように各プロセスで別個に用意される．つまり，プロセス A の要求スレッドはプロセス A の共有メモリに対してアクセスを行い，プロセス A の処理スレッドはプロセス A の要求スレッドからの処理要求を処理する．

4.2　処理スレッド

処理スレッドの処理手順を図 4 に示す．処理スレッドが行わなければならないのは，共有メモリへアクセス可能かどうかの確認，リクエストを参照してシステムコールを発行し結果を返す，自身がまだ必要であるのか必要でないのかの確認である．

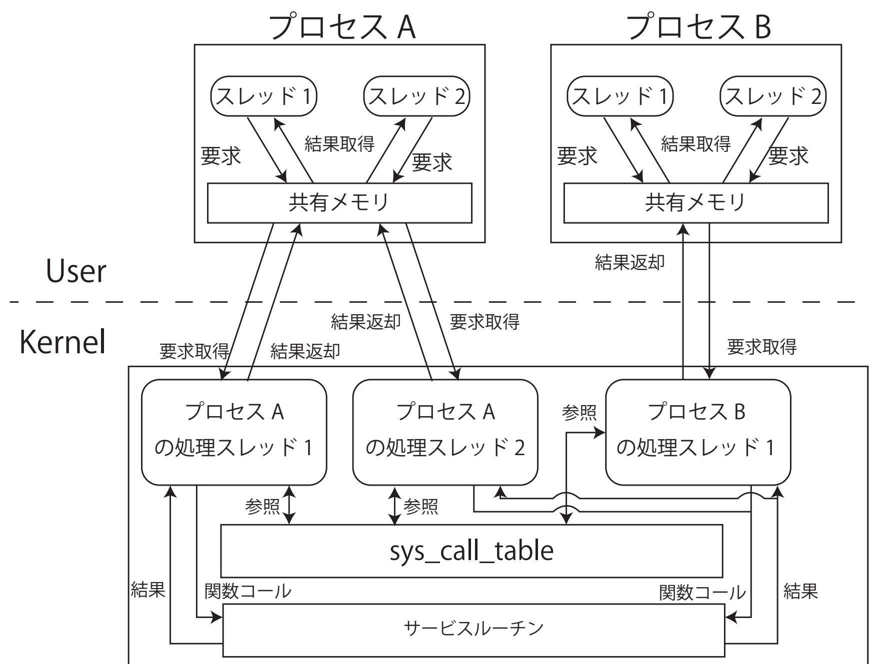

図 3: 提案手法の全体動作

処理スレッドはまず，共有メモリへアクセス可能かを確認する．ユーザスレッドが確保した共有メモリ領域に対して書き込み可能である事が確認されなければ，その時点でエラーとして終了する．次に，共有メモリ中の終了フラグとリクエスト領域の先頭アドレスを別変数に格納しておく．終了フラグのアドレスに格納されている値を確認し，終了フラグが設定されていない間，繰り返しシステムコールの処理を行う．図 4 中の (1) では，処理スレッドはリクエスト領域の先頭アドレスから末尾までを順に探索し，要求が置かれていないかを確認している．要求が置かれている場合にはすぐにその処理を始める．要求が置かれていない場合には待機状態となる．要求が置かれているかどうかは，リクエスト構造体の status を参照することで判別できる．status メンバが REQUEST に設定されている領域は，要求スレッドによってすでにリクエストが完成している．

要求状態のリクエスト構造体を見つけた処理スレッドは，システムコール処理を始める．その処理手順を図 5 に示す．図中の (2) と (3) は，スレッド間での同期処理が必要となる部分である．まず，(2) においては要求の処理が処理スレッド間で競合しないようにする必要がある．これには，コンペア・アンド・スワップ (CAS) 命令を用い

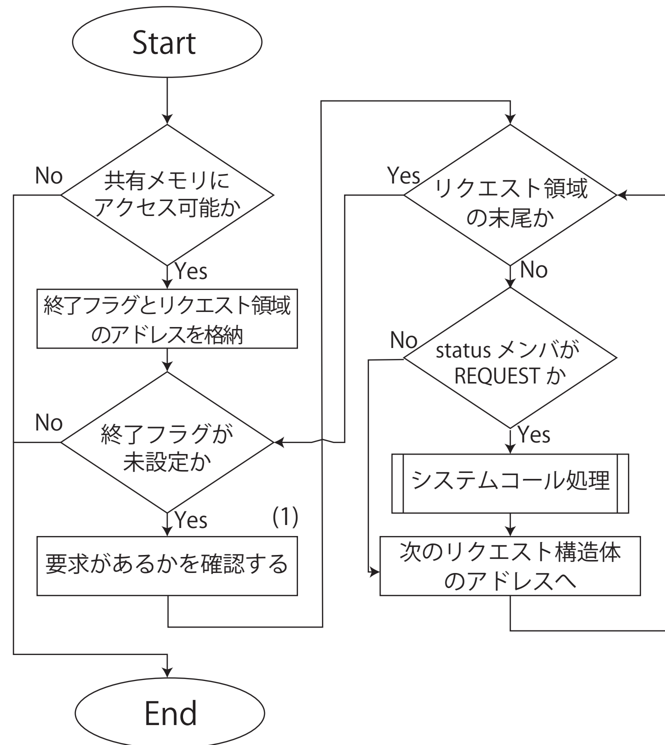

図 4: 処理スレッドのアルゴリズム

る．CAS はメモリの内容を元の値と更新したい値で比較し，値が異なる場合のみ不可分に新しい値へ更新する CPU 命令である．CAS によってリクエスト構造体の status を REQUEST から PROCESSING に変更することで，同時に複数の処理スレッドが一つの要求を処理することを防ぐ．要求の処理を始めた処理スレッドは，共有メモリ上のシステムコール番号と引数を基に，sys_call_table のエントリに格納されたサービスルーチンを呼び出す．返り値を受け取った後，それを ret_val メンバに戻し，返り値に従って status メンバを SUCCESS または ERROR に変更する．最後に，図中の (3) で処理を行った回数を加算する．この回数は，同時に複数の処理スレッドが加算を行ったり，加算対象の変数が 3.2 節に記述した parasc_wait システムコール内で参照される．そのため，処理を行う前に占有ロックをかけ，処理を行った後にロックを解放する必要がある．提案手法では，これをスピンロックを用いて行った．

要求の処理を終えた処理スレッドは，次のリクエスト構造体へ参照するアドレスを進める．もし未処理の要求があれば同様に処理を行い，未処理の要求が無いことを確認した後は，終了フラグの条件分岐へと戻る．

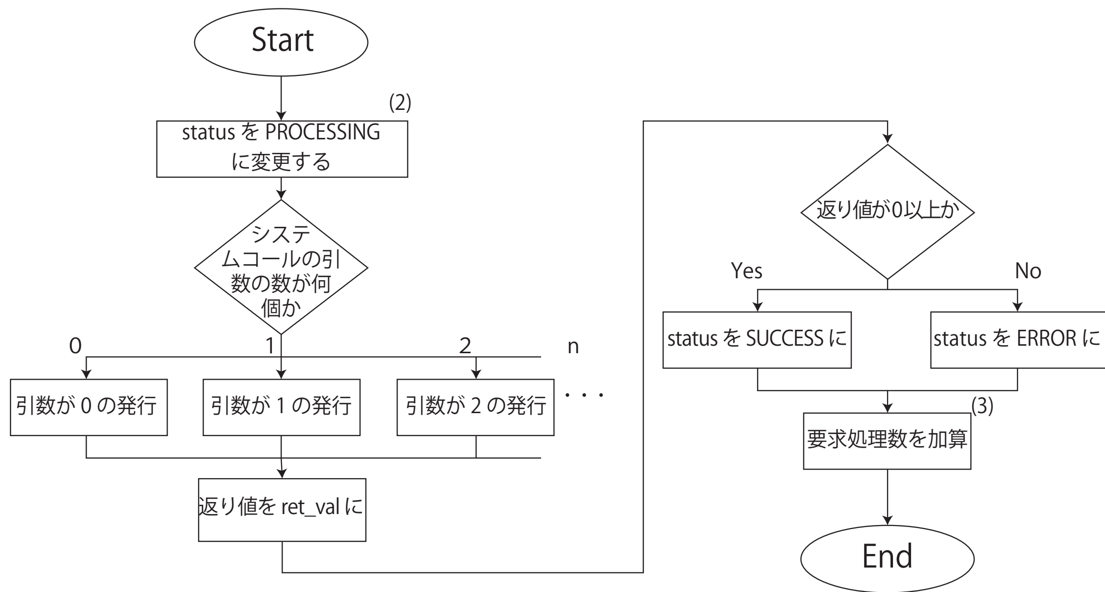

図 5: システムコール発行部分のアルゴリズム

4.3　要求スレッド

要求スレッドが行わなければならない事は，共有メモリの確保，スレッドの生成，各システムコールの呼び出し，リクエストの生成と設定と結果の読み込み，終了フラグの設定，スレッドの待ち合わせである．

最初に並列システムコールを使用する要求スレッドは，まず，共有メモリを確保する．処理スレッドとなるスレッドは parasc_proc システムコールを発行して処理スレッドとなる．その間，要求スレッドはシステムコール要求を共有メモリに作成する．作成する際には，他スレッドと競合しないように status メンバを REQUESTING にして，作成後は REQUEST に変更する．要求を作成した要求スレッドは，parasc_wait システムコールを使用して処理完了を待つことができる．また，システムコールに関連せず他に処理しなければならないことがあれば，それを処理できる．parasc_wait システムコールから復帰した，もしくは結果が必要になった要求スレッドは，共有メモリを参照することで結果を回収する．その後，status メンバを FREE にする．処理スレッドを parasc_proc システムコールから復帰させる場合には，終了フラグの値を設定する．

5 評価

本章では，提案手法の有効性を確かめるために行った実験と，結果および考察について記述する．実験は表 2 に示す構成のマシンで行った．

5.1 並列システムコールのオーバヘッド計測

提案手法では，従来型のシステムコール発行が用いていた値のレジスタ渡しを共有メモリを用いる方法に変更し，またシステムコールの処理を行うスレッドを複数使って処理の効率化を目指す．一方で，システムコールの発行を行うために共有メモリを確保しなければならないことや，スレッドを生成すること，要求をコピーすること，処理された結果を回収することなど従来型のシステムコール発行にはなかった手順を踏む必要がある．そこで，まず，提案手法を用いてシステムコール発行を行うとき，行わなければならない各手順にかかる処理時間を測定する．これは，gettid システムコールを一回，提案手法と従来型の手続きによって発行することで行う．そして，それを 500 回行い平均をとった．

まず，提案手法の各手順でかかる処理時間を計測した結果を表 3 に示す．以下の処理時間を図 6 に示す．グラフの縦軸は処理時間を表す．

init 共有メモリを確保してから，処理スレッドを生成するまでの時間

struct リクエスト構造体を作成する時間

gettid リクエスト構造体を共有メモリにコピーしてから，処理スレッドが処理を行い，結果を取得するまでの時間

表 2: 実験マシンの構成

カーネル	Linux-4.14.9
CPU	core-i5 4690
メモリ	DDR3-16000 8 G バイト
SSD	CSSD-S6T128NHG6Q
共有メモリ	要求数によって変化し，4096 バイトから 20,480 バイトまで
処理スレッド数	1 から 3 まで

表 3: 各処理ごとの処理時間の値

処理	時間 [ns]
init	24,759
struct	59
gettid	16,138
end	32,649
normal	379

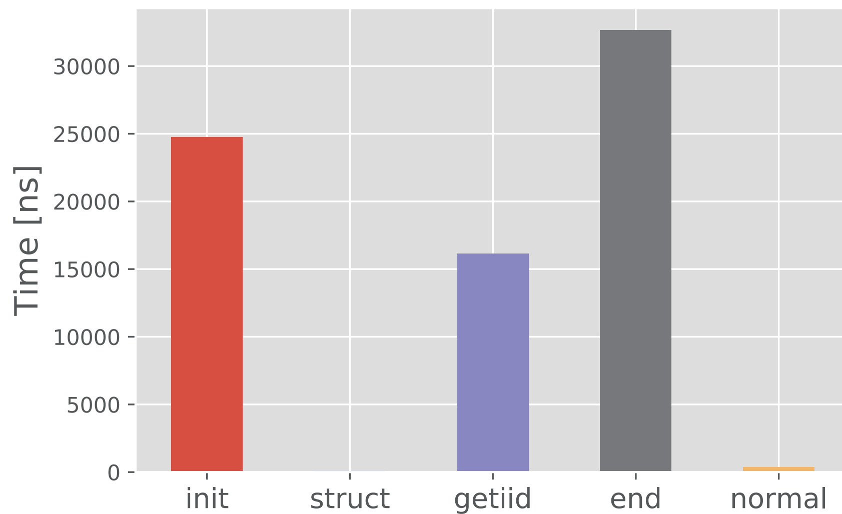

図 6: 各処理ごとの処理時間

end 終了フラグを設定してから，プロセスに復帰した処理スレッドと待ち合わせるまでの時間

normal 従来型の手続きで gettid システムコールを発行するのにかかった時間

グラフから，最も処理時間がかかっているのは end の処理である．また，gettid の処理時間 16,138 ナノ秒は normal の処理時間 379 ナノ秒と比べ大きく増加している．gettid はほとんどカーネル内で処理を行わないシステムコールである．よって，従来型の手続きと提案手法を用いた gettid 発行にかかる処理時間は，その差 16,138 - 379 = 15,759 ナノ秒が提案手法のオーバヘッドだと言える．

各項目の処理時間に対して考察を行う．提案手法を用いた gettid のシステムコール発行は従来型の手続きと比べ約 50 倍の時間がかかった．これは，共有メモリへのリクエスト作成から結果の取得の間に，システムコール処理の完了まで要求スレッドが待機状態になっていた事が原因であると考えられる．提案手法において，システムコー

ルの要求を行うスレッドと処理を行うスレッドは分けられる．よって，要求スレッドが要求を行った後に結果を取得するには，共有メモリを直接参照するか，parasc_wait システムコールを用いる．今回の実験では，要求スレッドは parasc_wait システムコールを使用し，処理が 1 つ以上完了するまで待機状態になった．しかし，処理スレッドを生成してからすぐに共有メモリへリクエストを置いた場合，処理スレッドは前準備を行っている途中で，システムコールの処理可能状態にないことがある．そのため，処理スレッドが要求を発見し処理を行ってから，再び要求スレッドを実行可能状態にするまでの時間が加算されたのだと考えられる．

5.2　並列システムコールによる要求処理時間

従来型のシステムコール発行と提案手法のシステムコール発行の処理時間の差を計測する．合わせて，提案手法ではスレッド数による処理時間の変化を計測する．これは，read システムコールを使い /dev/zero から 4096 バイトを読み出すことによって行い，従来型，提案手法ともに，要求数を 1，5，10 と 5 刻みで 50 まで，50 からは 50，75，100 と 25 刻みで 300 個まで変化させる．このとき，共有メモリのサイズは要求数によって動的に変化させ，1 から最大 300 までの要求を一度に置くことができるようにする．また，提案手法の計測は全ての要求を共有メモリにコピーするところから，結果を全て回収するところまで計測する．そして，それぞれの要求数，手法において 500 回ずつ計測を行い，平均をとった．提案手法のみ，処理スレッド数を変え同じ実験を行う．

従来型のシステムコール発行と提案手法のシステムコール発行の処理時間の差を図 7 に示す．グラフの縦軸は処理時間であり，横軸はシステムコールの発行回数である．1 から 50 回までの発行回数では，従来型の手続きの処理時間が提案手法の処理時間よりも下回っているのが確認できる．一方で，50 以降の発行回数では，従来型の手続きは線形に処理時間が増え，提案手法も線形に処理時間が増えるものの，従来型の手続きより増加が緩やかである．それは，処理スレッドの数が増えるほど顕著であり，75 回あたりの発行回数では全ての処理スレッド数において，完全に逆転している．結果，従来手続きとの差が最も大きくなった処理スレッドが 3 個の場合では，処理時間が従来型の手続きの約 50% となった．

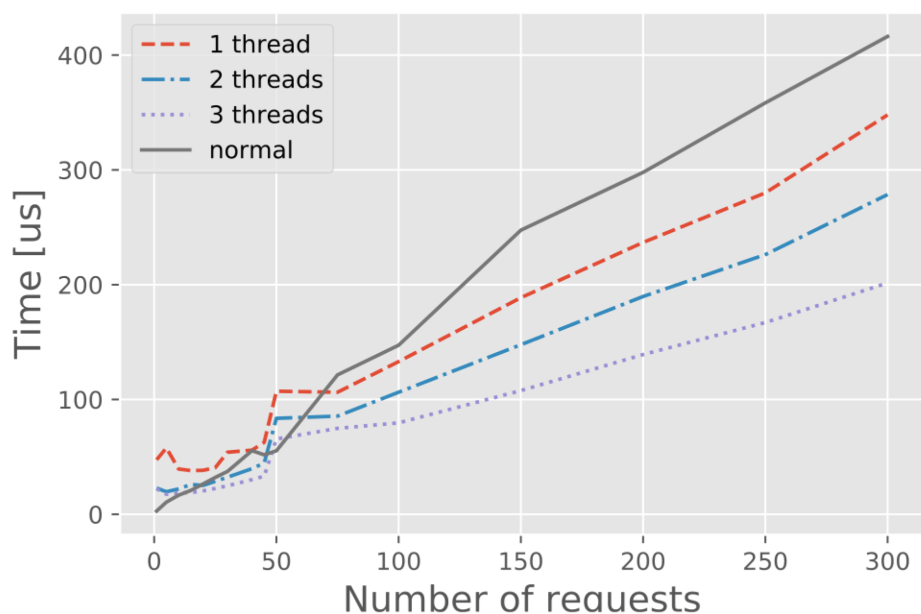

図 7: 提案手法と従来型の処理時間

従来型のシステムコール発行と提案手法のシステムコール発行の処理時間の差について考察を行う．少ない発行回数では従来型の手続きの方が処理時間が短かった理由としては，上記のことが考えられる．一方で，発行回数が多くなるにつれ提案手法によるシステムコール発行の方が処理時間が短くなった理由として，ソフトウェア割り込みが発生しなくなったことや，引数や返り値がレジスタ渡しでは無くメモリアクセスになり，オーバヘッドが削減されたことが考えられる．加えて，提案手法では処理スレッドは独立して動作する．よって，要求スレッドが従来型の手続きで parasc_wait システムコールを発行したりそれから復帰したりしている間に，処理スレッドは共有メモリ内の要求を処理できる．要求スレッドは共有メモリの先頭から末尾まで順に結果を回収し，不足している場合に parasc_wait を発行すれば良いので，全体の発行回数が減少したことによる結果だと考えられる．また，処理スレッド数が増えれば，発行回数ごとの処理時間が短くなった理由として，並列化の効果が考えられる．一つの処理要求を一つの処理スレッドが実行している間，他の処理スレッドは別の処理要求を実行できる．よって，処理スレッドで処理要求を分担でき，全ての要求を実行し終えるまで時間が短くなったと考えられる．しかし，3 並列で処理要求を処理した場合に，単純に処理時間が 3 分の 1 とはならなかった．これは処理スレッド間で，要求に対する競合や共通変数に対する競合が発生した場合に，コンペア・アンド・スワップ

やロックによる同期処理によって，処理がブロックされていることが考えられる．また，プロセスに生成されたスレッドが parasc_proc システムコールを発行し処理スレッドとなる．だが，生成されたスレッドが処理を開始するまでの時間にばらつきがあり，最初から最後までを 3 並列で行えなかった可能性がある．よって，要求スレッドと処理スレッドの同期的処理手順のさらなる検討や，処理スレッド間の同期処理を十分に検証する必要がある．

6 おわりに

本論文では，従来型のシステムコール発行手続きを変更し，要求を行うスレッドと処理を行うスレッドに分けることにより，システムコールのオーバヘッド削減と実行の効率化を行う方法を述べた．また，提案手法を基に並列システムコール機構を実装した．そして，提案手法の有効性を確認するための実験を行った．その結果，提案手法によるシステムコール発行のオーバヘッドが 15,759 ナノ秒であった．また，システムコールの発行回数が多くなるにつれ，提案手法の処理時間が通常の手続きより短くなることが確認できた．そして，処理を行うスレッドの数を増やせば，単位時間あたりの処理可能数が増加し，より短い時間で処理を終えることを確認した．

今後の課題として，提案手法が効果を発揮すると考えられる，Web サーバやデータベースサーバなどシステムコールが多発する環境での実験を行う必要がある．また，提案手法を用いたアプリケーションやサーバなどを作成し，従来型の手続きを用いたそれらより性能が向上することを検証する必要がある．

謝辞

本研究を行なうにあたり，終始熱心なご指導とご鞭撻を頂いた，三好力教授，芝公仁助教に心から感謝致します．また，本研究を進めるにあたり有意義な御助言を頂いた，芝研究室の院生の方に深く感謝致します．最後に，日頃参考となる貴重な意見やご協力を頂いた芝研究室及び三好研究室の皆様に感謝致します．

参考文献

[1] Livio Soares and Michael Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *OSDI*, 2010.

[2] Mohan Rajagopalan, Saumya K. Debray, Matti A. Hiltunen, and Richard D. Schlichting. System call clustering: A profile-directed optimization technique. 2002.

[3] Intel, https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 2b edition, September 2016.

[4] Jan Hubcika, Andreas Jaeger, Michael Matz, Mark Mitchell, https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf. *System V Application Binaly Interface AMD64 Architecture Processer Supplement*, July 2013.

[5] Marco Cesati Daniel P. Bovet. 詳解 LINUX カーネル. オライリー・ジャパン, 第 3 版, 2007.